Rewriting Declarative Query Languages

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

vorgelegt von

Diplom-Wirtschafts-Informatiker
Matthias Brantner

aus Schramberg

Mannheim, 2007

Dekan: Professor Dr. Matthias Krause, Universität Mannheim
Referent: Professor Dr. Guido Moerkotte, Universität Mannheim
Korreferent: Professor Dr. Donald Kossmann, ETH Zürich

Tag der mündlichen Prüfung: Donnerstag, den 11.10.2007

Zusammenfassung

Anfragen an Datenbanken werden mit Hilfe deklarativer Anfragesprachen gestellt. Beispiele hierfür sind die relationale Anfragesprache SQL und XPath oder XQuery für Anfragen an XML-Daten. Auf Grund der Deklarativität muss der Anfragesteller nichts über die Techniken wissen, die benutzt werden, um eine Anfrage zu bearbeiten. Stattdessen kann der Anfragebearbeiter eines Datenbanksystems hierfür beliebige Algorithmen auswählen.

Im relationalen Kontext wird die Anfragebearbeitung gewöhnlich mit Hilfe einer relationalen Algebra durchgeführt. Hierfür wird eine Anfrage in eine logische Algebra übersetzt. Diese Algebra besteht aus logischen Operatoren, die es erlauben, zahlreiche Optimierungstechniken anzuwenden. Beispielsweise können Ausdrücke in einer logischen Algebra umgeschrieben werden, um Ausdrücke zu erhalten, die effizienter auszuwerten sind.

Um Anfragen an Daten zu stellen, die im XML Format gespeichert sind, wurden die Anfragesprachen XPath und XQuery entwickelt. Beide sind deklarativ und haben dadurch ebenfalls ein groŝes Optimierungspotential. Insbesondere können sie mit einer Algebra ausgewertet werden. Leider sind die existierenden Ansätze (z.B. aus dem relationalen Kontext) jedoch nicht direkt anwendbar.

Das Ziel dieser Dissertation besteht aus zwei Teilen. Im ersten Teil werden die eben genannten Defizite der Auswertung von XML-Anfragespachen beseitigt. Es wird ein algebraisches Rahmenwerk entwickelt, um XPath und XQuery effizient auszuwerten. Dadurch soll es möglich werden, XPath und XQuery im groŝtechnischen Einsatz zu benutzen. Hierfür wird zuerst eine ordnungserhaltende logische Algebra definiert und danach eine Übersetzung von XPath in diese Algebra vorgestellt. Darüber hinaus werden Regeln vorstellt, mit denen ein Algebra-Ausdruck umgeformt werden kann, um den resultierenden algebraischen Ausdruck schneller ausgewerten zu können.

Im zweiten Teil der Arbeit werden neue Optimierungen im relationalen Kontext entwickelt, mit deren Hilfe sich geschachtelte SQL-Anfragen mit Disjunktionen entschachteln lassen. Hierfür werden algebraische Äquivalenzen vorgestellt, welche geschachtelte algebraische Ausdrücke in algebraische Ausdrücke mit ”Bypass-Operatoren” überführen. Die entwickelten Äquivalenzen können Anfragen mit einem disjunktiven Verbindungsprädikat und Anfragen, bei denen das Korrelationsprädikat disjunktiv vorkommt, entschachteln. Damit lassen sich sowohl Unteranfragen mit mengenwertigen Ergebnissen, als auch Anfragen mit skalaren Ergebnissen, entschachteln.

Für alle Optimierungen wurden Experimente durchgeführt. Ihre Ergebnisse, die in dieser Arbeit vorgestellt werden, zeigen die Wirksamkeit aller entwickelten Ansätze zeigen.

Abstract

Queries against databases are formulated in declarative languages. Examples are the relational query language SQL and XPath or XQuery for querying data stored in XML. Using a declarative query language, the querist does not need to know about or decide on anything about the actual strategy a system uses to answer the query. Instead, the system can freely choose among the algorithms it employs to answer a query.

Predominantly, query processing in the relational context is accomplished using a relational algebra. To this end, the query is translated into a logical algebra. The algebra consists of logical operators which facilitate the application of various optimization techniques. For example, logical algebra expressions can be rewritten in order to yield more efficient expressions.

In order to query XML data, XPath and XQuery have been developed. Both are declarative query languages and, hence, can benefit from powerful optimizations. For instance, they could be evaluated using an algebraic framework. However, in general, the existing approaches are not directly utilizable for XML query processing.

This thesis has two goals. The first goal is to overcome the above-mentioned misfits of XML query processing, making it ready for industrial-strength settings. Specifically, we develop an algebraic framework that is designed for the efficient evaluation of XPath and XQuery. To this end, we define an order-aware logical algebra and a translation of XPath into this algebra. Furthermore, based on the resulting algebraic expressions, we present rewrites in order to speed up the execution of such queries.

The second goal is to investigate rewriting techniques in the relational context. To this end, we present rewrites based on algebraic equivalences that unnest nested SQL queries with disjunctions. Specifically, we present equivalences for unnesting algebraic expressions with bypass operators to handle disjunctive linking and correlation. Our approach can be applied to quantified table subqueries as well as scalar subqueries.

For all our results, we present experiments that demonstrate the effectiveness of the developed approaches.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

Relational database systems are the prevalent choice for managing large amounts of data. These systems offer, for example, efficient and reliable storage, access control, multiuser synchronization, or assure data integrity. Moreover, besides these features, the efficient processing of queries is a major factor of success for these systems. In general, queries are formulated in a declarative language such as SQL. Using a declarative query language, the querist does not need to know about or decide on anything about the actual strategy a system uses to answer the query. Instead, the system can freely choose among the algorithms it employs to answer a query — as long as the result is correct. Predominantly, query processing in the relational context is accomplished using a relational algebra. To this end, the query is translated into a logical algebra. The algebra consists of logical operators which facilitate the application of various optimization techniques. For example, logical algebra expressions can be rewritten in order to yield more efficient expressions. Such rewrites can be formally proven for their validity. In addition, implementing the logical algebra in an iterator-based, pipelined query execution engine scales well to large data volumes [49].

XML (eXtensible Markup Language) is a standardized format [20] to store semi-structured documents. Thanks to its flexibility and its ease of use it is nowadays widely used in a vast number of application areas. It has emerged as the predominant mechanism for representing and exchanging data. For example, it is used in document-centric applications for modeling and exchanging data in the financial (FIXML), chemical (CML), or biological (BIOML) sector. Moreover, web standards such as Web Services use XML for communicating with clients or among each other. Because of its widespread use and the large amounts of data that are represented in XML, the necessity arises for efficiently managing and storing XML data. This need led the major commercial database system vendors (i.e. [8, 77, 92]) as well as many open source projects (e.g. eXist, MonetDB, MySQL) to integrate

XML support in their systems.

In order to query XML data, XPath [25] and XQuery [38] have been developed. Both are declarative query languages and, hence, can benefit from powerful optimizations. For instance, they could be evaluated using an algebraic framework. However, in general, the existing approaches are not directly utilizable for XML query processing. There are two main reasons for this: (1) The mismatch between the relational model and the tree-based data model of XML and (2) the order indifference of relations in contrast to XML, which takes the order of nodes within an XML document into account. The goal of this thesis is twofold.

The first goal is to overcome the above-mentioned misfits of XML query processing, making it ready for industrial-strength settings. Specifically, we develop an algebraic framework that is designed for the evaluation of XPath and XQuery. With this framework, we want to contribute to the development of efficient evaluation techniques for these query languages. To this end, we define an order-aware logical algebra and a translation of XPath into this algebra. Based on the resulting algebraic expressions, we present rewrites, for example, to unnest nested XPath and XQuery expressions. We validate each of our techniques with an experimental evaluation, comparing our approaches against several existing systems.

The second goal is to investigate rewriting techniques in the relational context. To this end, we present rewrites based on algebraic equivalences that unnest nested SQL queries with disjunctions. Nested queries with disjunctions seem to become more and more relevant in practice (cf. [105]). To the best of our knowledge, there does not exist a solution to unnest these kinds of subqueries so far. To proof the effectiveness of our rewrites, we present an extensive performance study that compares our approach against the canonical approach (which seems to be common in practice) and three major commercial database systems.

The detailed contributions together with the outline of this thesis are described in the following subsection.

1.2 Contributions

1.2.1 Algebraic XPath Evaluation

In Chapter 2 of this thesis, we present a complete algebraic approach for evaluating XPath 1.0 [1]. We present a logical algebra that is capable of evaluating all of XPath and develop a complete translation function into this algebra. Although this translation results in a rather naïve way of evaluating XPath, we develop a technique to remedy the exponential runtime behavior (of the naïve evaluation) that has been identified by Gottlob et al. [47]. At the end, we complement the logical algebra with a physical algebra and describe the implementation of these operators in the

[1] In the remainder of this thesis, we always use XPath when talking about XPath 1.0 [25].

runtime system of the native XML database management system Natix [40]. Using the Natix physical algebra, we present an experimental study to validate our approach. The logical algebra, the translation of XPath into this algebra, as well as the presentation and evaluation of the physical algebra can also be found in the following two publications.

[11] Matthias Brantner. Algebraische Auswertung von XPath in Natix. Masters thesis, University of Mannheim, Mannheim, Germany, March 2004. (in German).

[12] Matthias Brantner, Sven Helmer, Carl-Christian Kanne, and Guido Moerkotte. Full-fledged algebraic XPath processing in Natix. In ICDE, pages 705-716, 2005.

1.2.2 Unnesting XPath Expressions

Based on our full-fledged algebraic XPath approach, we develop optimization techniques that target further shortcomings of the naïve approach in Chapter 3. Therefore, we classify nested XPath expressions, and, for each class, we present optimizations in the form of algebraic equivalences. They have expressions resulting from our naïve translation (see previous subsection) on the left-hand side and an optimized counterpart on the right-hand side. Primarily, the right-hand side employs unnesting strategies that are already known from the context of SQL (e.g. [70]), OQL (e.g. [29]), and lately XQuery (e.g. [82]). Parts of the presented techniques have already been presented in the following poster paper.

[17] Matthias Brantner, Carl-Christian Kanne, Guido Moerkotte, and Sven Helmer. Algebraic optimization of nested XPath expressions. In ICDE, page 128. IEEE Computer Society,

1.2.3 Disjunctive Unnesting for XPath

The techniques mentioned in the previous subsection are a first step to unnest nested XPath expressions. However, in a second step, we take unnesting techniques a step forward. Specifically, we present unnesting techniques that are not limited to nested queries occurring in conjunctions but are also capable to unnest disjunctive queries (see Chapter 4). So far, we are not aware of any technique to unnest nested queries in the presence of disjunctions. Our techniques can not only be exploited for unnesting XPath but can also be applied to nested XQuery FLWORs that occur disjunctively. The results presented in Chapter 4 have already been published in a workshop paper and a technical report.

[13] Matthias Brantner, Sven Helmer, Carl-Christian Kanne, and Guido Moerkotte. Kappa-join: Efficient execution of existential quantification in XML query

languages. In XSym, volume 4156 of Lecture Notes in Computer Science, pages 1-15. Springer, 2006.

[14] Matthias Brantner, Sven Helmer, Carl-Christian Kanne, and Guido Moerkotte. Kappa-join: Efficient execution of existential quantification in XML query languages. Technical Report, University of Mannheim, 2006.

1.2.4 Disjunctive Unnesting for SQL

Encouraged by the results we achieved by unnesting disjunctive nested XPath and XQuery queries, we diverge from optimizing the execution of XPath and XQuery. In Chapter 5, we present algebraic unnesting techniques for SQL queries that occur disjunctively. We found this extremely useful because SQL queries containing nested queries with disjunctions seem to become more and more relevant in practice (cf. [36, 105]). We have already published the unnesting of scalar nested queries in a conference paper [19] and the unnesting of both scalar and table subqueries in a technical report [18].

[19] Matthias Brantner, Norman May, and Guido Moerkotte. Unnesting scalar SQL queries in the presence of disjunction. In ICDE, pages 46–55, 2007.

[18] Matthias Brantner, Norman May, and Guido Moerkotte. Unnesting SQL queries in the presence of disjunction. Technical report, University of Mannheim, March 2006.

1.2.5 Preparing XQuery for Plan Generation

After our excursion into the relational world, we are back with XML in the last Chapter 6. In this chapter, we investigate rewrites that merge XQuery FLWOR blocks. These rewrites are useful to support plan generation. Plan generators generate optimal plans only for a single query block. Hence, bigger query blocks usually imply a bigger search space for a plan generator and better query execution plans are possible. The rewrite toolkit developed has already been published in a workshop paper and, an extended version, in the according technical report.

[16] Matthias Brantner, Carl-Christian Kanne, and Guido Moerkotte. Let a Single FLWOR Bloom (to improve XQuery plan generation). In XSym, Lecture Notes in Computer Science. Springer, 2007.

[15] Matthias Brantner, Carl-Christian Kanne, and Guido Moerkotte. Let a Single FLWOR Bloom. Technical report, University of Mannheim, 2007.

In the last chapter (Chapter 7), we conclude this thesis and give an outlook onto future work.

Chapter 2

Algebraic XPath Processing

The efficient processing of XML data hinges on fast evaluation techniques for XPath expressions, because XPath is an essential part of widely used XML processing languages like XSLT and XQuery. We present the first complete translation of XPath into an algebra.

Such a translation of XPath expressions into algebraic expressions (1) renders possible algebraic optimization approaches as found in most modern query optimizers, and (2) facilitates the application of iterator-based, pipelined query execution engines that scale well to large data volumes and have proven their performance e.g. in relational systems. For the same reasons, algebra-based XQuery evaluation is attractive, requiring algebra-based XPath evaluation as an essential ingredient.

The main contributions of this chapter are:

- We introduce an algebra capable of expressing *any* XPath query
- We show exactly *how* all XPath constructs can be translated into algebraic expressions

These contributions are not intended to be purely theoretical exercises. To show their usefulness in implementing XPath evaluators, we also discuss our compiler and algebra implementation, and give some performance results.

In our approach we translate XPath 1.0 expressions[1] into a logical algebra working on *ordered tuple sequences*. The main task here is to avoid unnecessary work by *eliminating duplicates* in intermediate results or memoizing already computed results (such as location steps or predicates) if duplicate elimination is not immediately possible due to the semantics of XPath predicates. This is very important, as the presence of duplicates may lead to an exponential run time [48, 47, 46]. Another important point we cover is the efficient evaluation of predicates in XPath. We pay particular attention to *position-based predicates* using *position()* or *last()*.

[1] We will only write XPath in the following, always meaning XPath 1.0 except when explicitly stated otherwise.

For query execution, we use the physical algebra of our native XML database system Natix [40]. It implements the operators of the logical algebra in an iterator-based fashion [49]. We do not need to construct a complete main memory representation of an XML document in order to evaluate XPath expressions. Our approach directly accesses the physical storage layout of the XML documents on disk. Finally, our XPath evaluator is implemented in a modular way, allowing the integration of several different optimization techniques.

The remainder of this chapter is organized as follows: In Sec. 2.1, we summarize the XPath semantics and introduce our logical algebra. Sec. 2.2 describes the canonical translation of XPath expressions into our algebra, and Sec. 2.3 shows how to avoid an exponential run-time of the queries. Implementation details concerning our physical algebra are given in Sec. 2.4. In Sec. 2.5, preliminary performance results show that even without further optimization, our approach compares favorably to main-memory based evaluators, and scales better to large document sizes. Sec. 2.7 summarizes the contributions of this chapter.

2.1 Translation Input and Output

This section explains domain and range of our translation function: We give a brief summary of XPath expression semantics and introduce our logical algebra.

2.1.1 XPath Semantics

The primary syntactic construct in XPath is an expression. When evaluating an expression, the result has one of the following four basic types: a node-set (an unordered collection of nodes without duplicates), a boolean value ('true' or 'false'), a number (a floating-point number), or a string (a sequence of characters). The evaluation of an expression considers a context, which consists of the following: a node (also called the context node), a pair of non-negative integers (the context position and context size), a set of variable bindings (a map from variable names to values), a function library, and a set of namespace declarations.

Please note that in XPath 1.0, the node-sets themselves are unordered. However, there exists the notion of document order, totally ordering all nodes of a document. Document order is relevant in the evaluation of location steps, but not in the representation of node-sets. Hence, we do not always return result sequences in document order. For XPath 2.0 (and integration into XQuery), if ordered results are required, additional sorting is sometimes [59] necessary.

2.1.2 Logical Algebra

Before going into the details of the translation, we have to define the target algebra and some associated notions.

Universe

The universe of our algebra is the union of the domains of the atomic XPath types (`string, number, boolean`) and the set of ordered sequences of tuples[2].

A tuple is a mapping from a set of attributes to values. We allow nested tuples, i.e. the value of an attribute may be a sequence of tuples. In addition to sequences, attribute values may be document nodes or values of the atomic XPath types.

Conventions

Before defining the main algebra operators below, we introduce the notations used in their definition and in the description of the translation process:

The set of attributes defined for a tuple t is written as $\mathcal{A}(t)$. All the tuples $t \in e$ of a sequence-valued expression e have the same attributes $\mathcal{A}(t)$, which are also denoted as $\mathcal{A}(e)$. The set of free variables of an expression e is defined as $\mathcal{F}(e)$.

Single tuples are constructed by using the standard $[\cdot]$ brackets. The concatenation of tuples and functions is denoted by $\circ$.

The projection of a tuple on a set of attributes A is denoted by $|_A$. We also define $t|_{\overline{A}} := t|_{A(t)\setminus A}$. For brevity reasons, we identify a tuple containing a single attribute with the value of that attribute.

For an expression e possibly containing free variables, and a tuple t, we denote by $e(t)$ the result of evaluating e, where bindings of free variables are taken from attribute bindings provided by t. Of course this requires $\mathcal{F}(e) \subseteq \mathcal{A}(t)$. In general, accesses to identifiers are resolved by lookup in the tuple; if no mapping can be found, the tuples of the surrounding algebra expressions are checked successively. Ultimately, the free variables of the complete expressions must be bound by a top-level map supplied as execution context for the expressions. This top-level map also has to provide bindings for XPath variables and the context node for the execution.

For sequences e, we use $\alpha(e)$ to denote the first element of a sequence. We identify single element sequences and elements. The function τ retrieves the tail of a sequence, and $\oplus$ concatenates two sequences. We denote the empty sequence by ϵ. As a first application, we construct from a sequence of non-tuple values e a sequence of tuples denoted by $e[a]$. It is empty if e is empty. Otherwise, $e[a] = [a : \alpha(e)] \oplus \tau(e)[a]$.

By *id* we denote the identity function.

Operators

The main operators of our algebra are sequence-valued analogs of traditional relational algebra operators. An overview of the formal definitions of the sequence-valued operators is given in Fig. 2.1. More detailed comments about the operators

[2]We use ordered sequences instead of node-sets since predicate evaluation (with $position()$ and $last()$) and embedding of XPath into other languages is sensitive to the order of the returned nodes.

Selection σ	selects qualifying tuples according to predicate p: $\sigma_p(e) := \begin{cases} \alpha(e) \oplus \sigma_p(\tau(e)) & \text{if } p(\alpha(e)) \\ \sigma_p(\tau(e)) & \text{else} \end{cases}$
Projection $\mathbf{\Pi}$	projects on attributes in A (duplicate elimination version called Π^D_A, duplicate elimination without projection denoted by Π^D, attribute renaming version denoted by $\Pi_{a':a}$): $\Pi_A(e) := \alpha(e)\|_A \oplus \Pi_A(\tau(e))$ $\Pi_{a':a}(e) := \alpha(e)\|_a \circ [a' : a] \oplus \Pi_A(\tau(e))$
Map χ	extends each tuple t_i in e_1 with attribute a with value of $e_2(t_i)$: $\chi_{a:e_2}(e_1) := \alpha(e_1)\|_{Attr(e_1)\setminus\{a\}} \circ [a : e_2(\alpha(e_1))] \oplus \chi_{a:e_2}(\tau(e_1))$
Product $\overline{\times}$	connects single tuple t_1 to each tuple in e_2: $t_1 \overline{\times} e_2 := (t_1 \circ \alpha(e_2)) \oplus (t_1 \overline{\times} \tau(e_2))$
Cross product $\times$	connects all tuples in e_1 to all in e_2: $e_1 \times e_2 := (\alpha(e_1) \overline{\times} e_2) \oplus (\tau(e_1) \times e_2)$
D-join $<>$,$\bowtie$	joins each tuple t_i in e_1 to all tuples in e_2, which depend on t_i: $e_1 <e_2> := \alpha(e_1) \overline{\times} e_2(\alpha(e_1)) \oplus \tau(e_1) <e_2>$
Semi-join $\ltimes$	p checks for tuple existence in e_2 to decide on including tuple in e_1: $e_1 \ltimes_p e_2 := \begin{cases} \alpha(e_1) \oplus (\tau(e_1) \ltimes_p e_2) & \text{if } \exists x \in e_2 \;\; p(\alpha(e_1) \circ x) \\ \tau(e_1) \ltimes_p e_2 & \text{else} \end{cases}$
Unnesting μ	unnests a sequence-valued nested attribute: $\mu_g(e) := (\alpha(e)\|_{\overline{\{g\}}} \times \alpha(e).g) \oplus \mu_g(\tau(e))$
Unnest-Map $\mathbf{\Upsilon}$	abbreviated notation for a map operator followed by an unnest operator (μ): $\Upsilon_{a:e_2}(e_1) := \mu_g(\chi_{g:e_2[a]}(e_1))$
Binary Grouping $\bowtie\!\!\!\!\!\!\Gamma$	Adds to e_1 an attribute based on aggregation of e_2: $e_1 \bowtie\!\!\!\!\!\!\Gamma_{g;A_1\theta A_2;f}\ e_2 := \alpha(e_1) \circ [g : G(\alpha(e_1))] \oplus (\tau(e_1) \bowtie\!\!\!\!\!\!\Gamma_{g;A_1\theta A_2;f}\ e_2)$ $G(x) := f(\sigma_{x\|_{A_1}\theta A_2}(e_2))$
Aggregation $\mathfrak{A}$	Aggregates input sequence into a singleton sequence with a single attribute a: $\mathfrak{A}_{a;f}(e) := \{[a : f(e)]\}$
Sorting $\mathbf{S}ort$	Sorts input sequence based on attribute a: $Sort_a(e) := \begin{array}{l} Sort_a(\sigma_{a<\alpha(e).a}(\tau(e))) \oplus \alpha(e) \oplus \\ Sort_a(\sigma_{a\geq\alpha(e).a}(\tau(e))) \end{array}$
Singleton Scan $\square$	Returns singleton sequence consisting of the empty tuple: $\square := \{[\,]\}$

Figure 2.1: Sequence-valued operators of the target algebra

and their usage is embedded in the description of our translation process in the remainder of the chapter.

Except if explicitly stated otherwise, unary operators produce ϵ if their input is ϵ, and binary operators produce ϵ if their left input is ϵ. The d-join has two notations,

one to be used in visualizations of query trees ($\bowtie$) which designates the side that provides tuples as input to the other side using a filled triangle. The second representation is used for textual expressions where the dependent side is parenthesized ($<>$).

In addition, our target algebra provides counterparts for functions (e.g. contains) and operators (e.g. $+, *, /, =$) defined on the XPath basic types, including explicit and implicit conversions. For those functions that have node-sets as inputs (e.g. count), their algebraic counterpart has sequence-valued input. Note that for some XPath functions and operators, special translation rules are given in Sec. 2.2 (in particular node-set comparison, see Sec. 2.2.6). These functions or operators have no direct equivalent in our algebra.

2.2 Translation into Algebra

In a first translation step, we decide for each expression a mapping onto algebraic operators. In a second step (see Sec. 2.3), we enhance the translation to avoid exponential complexity of the evaluation process. The description of our translation process follows loosely the XPath grammar as found in the W3C recommendation [25].

When translating XPath into our algebra, we denote the translation of an expression e by $\mathcal{T}[e]$. The result of our translation function is an algebraic expression which may or may not be sequence-valued.

2.2.1 Location Paths

The most important construct in XPath is a location path. Location paths are applied to context nodes and produce as a result a node-set (Sec. 2.1.1).

We have to distinguish between absolute and relative location paths. An *absolute path* starts at the root element of an XML document. A *relative path* can start at an arbitrary context node. After that, both location paths are handled in the same manner.

The starting context node for a location path is provided by the variable cn. Note that for top-level location paths, cn is free and must be bound by the execution context; this is the mechanism for the execution engine to provide the initial context node.

Canonical Translation

A path expression $\pi = \pi_1/s_1/\ldots/s_{n-1}/s_n$ consists of a sequence of location steps (denoted by s_i).

For the moment, we assume that π starts with a partial expression π_1, consisting of the first location step of π, possibly prefixed by an initial $/$. We take a closer look

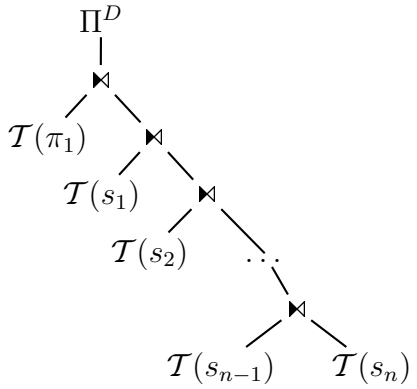


Figure 2.2: Canonical translation

at π_1 when distinguishing absolute from relative paths below. The individual steps are evaluated sequentially, i.e. the output of a location step s_i serves as the context for the following step s_{i+1}.

We translate a path expression into a chain of dependency joins (d-joins). In a d-join, the free variables in the expression on the right-hand side are bound with values supplied from a tuple generated by the expression on the left-hand side. We use this mechanism of a d-join to hand over the context from one location step to the next, one node at a time. The independent (left) subexpression of the d-join enumerates the context nodes from the previous step. The dependent subexpression of the d-join has the current step's context node as a free variable. Hence, each evaluation of the dependent subexpression corresponds to one result context of the location step.

We call a translation into d-joins the *canonical translation* of π:

$$\mathcal{T}[\pi] := \Pi^D(\mathcal{T}[\pi_1] < \mathcal{T}[s_1] > \ldots < \mathcal{T}[s_n] >)$$

We always want the cn attribute in a tuple sequence to contain the node attribute that was last added to the tuple. This makes it easy to treat all sequence-valued algebraic expressions uniformly.

We also have to add a projection operator that eliminates duplicates, as by definition the result of an XPath expression may not contain any duplicates (see also 2.1.1). The duplicate elimination only operates on the relevant context node attribute cn of the tuple, without projecting away the remaining attributes. Fig. 2.2 shows a graphical representation of the translated expression.

Absolute and Relative Paths

The initial context of a location path depends on whether it is an absolute or relative path, i.e. whether π_1 is prefixed by a slash or not. For absolute location paths, a map

operator is used supplying the input context node cn for the first location step s_1:

$$\mathcal{T}[\pi] := \Pi^D(\chi_{cn:root(cn)}(\Box) < \mathcal{T}[s_1] > \ldots < \mathcal{T}[s_n] >)$$

In the following, we abbreviate the algebraic expression $\chi_{cn':cn}(\Box)$ by $\Box_{cn':cn}$. It takes as input the free variable cn and creates a sequence containing a single tuple with a single attribute cn' containing the value of cn. This shortcut is called *context scan*.

$$\mathcal{T}[\pi] := \Pi^D(\Box_{cn:root(cn)} < \mathcal{T}[s_1] > \ldots < \mathcal{T}[s_n] >)$$

For relative location paths, the map operator can be omitted because cn is already bound to the context in which π has to be evaluated.

Unions

The union of path expressions $(\pi_1|\pi_2|\ldots|\pi_n)$ is translated into a series of concatenation operators followed by a duplicate elimination:

$$\mathcal{T}[\pi_1|\pi_2|\ldots|\pi_n] := \Pi^D(\mathcal{T}[\pi_1] \oplus \mathcal{T}[\pi_2] \oplus \ldots \oplus \mathcal{T}[\pi_n])$$

Note that the translation of π_i already binds cn to the produced context node.

2.2.2 Location Steps

A location step consists of three parts: an axis (which specifies the relationship between the result set of nodes and the context node), a node test (which specifies the node type and name of the selected nodes), and an arbitrary number of predicates (which use additional expressions to further refine the set of selected nodes). We will look at predicates in more detail in the following section, here we address axes and node tests. So for the moment, a step s_i is defined by an axis a_i and a node test t_i, i.e. is of the form $a_i :: t_i$

We translate the evaluation of a location step with the help of an auxiliary translation function Ψ:

$$\mathcal{T}[a_i :: t_i] := \Psi[a_i :: t_i] \circ \Box_{cn':cn},$$

where the actual translation of the location step uses an unnest map operator as follows:

$$\Psi[a_i :: t_i] := \Pi_{\overline{cn'}} \circ \Upsilon_{cn:cn'/a_i::t_i}$$

The unnest map operator takes cn' as input and creates bindings for the new context attribute cn. It is successively bound to the results produced by evaluating axis a and node test t with context cn'. At the end, the old context node cn' is discarded using $\Pi_{\overline{cn'}}$.

We explain in Sec. 2.4.2 how the evaluation of the subscript is performed. Here, it is sufficient to note that the result sequence is in the proper order for the specified axis.

For two neighboring location steps, e.g. in the absolute location path $/a_1 :: t_1/a_2 :: t_2$, it can be seen quite nicely that the result of the location step $a_1 :: t_1$ is used during the evaluation of location step $a_2 :: t_2$:

$$\Pi^D((\Box_{cn:root(cn)}) < \Pi_{\overline{cn'}} \circ \Upsilon_{cn:cn'/a_1::t_1}(\Box_{cn':cn}) >< \Pi_{\overline{cn'}} \circ \Upsilon_{cn:cn'/a_2::t_2}(\Box_{cn':cn}) >)$$

Example As an example, consider the XPath expression `//student/name`, which will be used (and extended) to illustrate problems and solutions throughout this thesis. In XPath, `//student` is an abbreviation for the two consecutive location steps `/descendant::node()/child::student`. Translating the full expression yields:

$$\begin{aligned}\mathcal{T}[//\text{student}/\text{name}] \;=\; & \Pi^D((\Box_{cn:root(cn)}) \\ & < \Pi_{\overline{cn'}} \circ \Upsilon_{cn:cn'/\text{descendant::node()}}(\Box_{cn':cn}) > \\ & < \Pi_{\overline{cn'}} \circ \Upsilon_{cn:cn'/\text{child::student}}(\Box_{cn':cn}) > \\ & < \Pi_{\overline{cn'}} \circ \Upsilon_{cn:cn'/\text{child::name}}(\Box_{cn':cn}) >)\end{aligned}$$

2.2.3 Predicates

A location step s_i may contain an arbitrary number h of predicates p_k and has the general form $a_i :: t_i[p_1] \ldots [p_h]$. The pattern for translating a location step $a_i :: t_i[p_1] \ldots [p_h]$ with predicates is

$$\Phi[p_h] \circ \cdots \circ \Phi[p_1] \circ \mathcal{T}[a_i :: t_i],$$

where Φ is an auxiliary translation function for predicates, returning a filtering functor which operates on algebraic expressions. We now elaborate on Φ.

An individual predicate p_k is represented as conjunction of several clauses l_{kj}, i.e. $p_k = \bigwedge_{j=1}^{m_k} l_{kj}$. Depending on whether or not the conjuncts contain function calls to the position-based functions $position()$ and $last()$, we have to translate them differently.

Simple Clauses & Nested Paths

Translating a predicate $p_k = l_{k1} \wedge \cdots \wedge l_{km_k}$ that does not include positional clauses simply results in a translation into selection operators:

$$\Phi[l_{k1} \wedge \cdots \wedge l_{km_k}] := \sigma_{\mathcal{T}[l_{km_k}]} \circ \cdots \circ \sigma_{\mathcal{T}[l_{k1}]}$$

After the semantic analysis, all clauses are broken down into function calls: $l_{kj} = f_1 \circ \cdots \circ f_r$. For example, `or`, `not`, and comparisons are all evaluated by function calls. All implicit conversions have also been added as function calls. We cover the translation of these calls in Sec. 2.2.6.

If a nested path is not used inside an aggregate function, the translation will add a conversion to boolean in the form of our internal $exists()$ aggregate function (see also Sec. 2.2.6).

Example To illustrate the translation of an XPath expression containing a nested path, let us extend our example to selecting only students that took at least one exam: `//stundent[exam]/name`.

$$\begin{aligned}\mathcal{T}[//\text{student}[\text{exam}]/\text{name}] \;=\; & \Pi^D((\Box_{cn:root(cn)}) \\ & <\Pi_{\overline{cn'}} \circ \Upsilon_{cn:cn'/\text{descendant::node()}}(\Box_{cn':cn})> \\ & <\sigma_{\mathfrak{A}_{x,exists()}(\Box<\Pi_{\overline{cn'}}\circ\Upsilon_{cn:cn'/\text{child::exam}}(\Box_{cn':cn})>)} \circ \\ & \Pi_{\overline{cn'}} \circ \Upsilon_{cn:cn'/\text{child::student}}(\Box_{cn':cn})> \\ & <\Pi_{\overline{cn'}} \circ \Upsilon_{cn:cn'/\text{child::name}}(\Box_{cn':cn})>)\end{aligned}$$

In this example, the value of the attribute `x` is `true` if there exists at least one exam for a given student and `false` otherwise. The selection operator checks this attribute, i.e. it compares `x` with `true`.

Clauses with $position()$

If at least one of the clauses in p_k contains $position()$ (but none of them contains $last()$), we have to count the number of context nodes that are produced. We do this with the help of a map operator that labels the tuples of the resulting nodes with their appropriate position within the current context (introducing a new attribute `cp`):

$$\Phi[l_{k1} \wedge \cdots \wedge l_{km_k}] := \sigma_{\mathcal{T}[l_{km_k}]} \circ \cdots \circ \sigma_{\mathcal{T}[l_{k1}]} \circ \chi_{cp:counter(p_k)++}$$

Calls to $position()$ are then translated into attribute accesses to cp:

$$\mathcal{T}[position()] := cp$$

Example For selecting only the first student, our algebra expression is as follows:

$$\begin{aligned}\mathcal{T}[//\text{student}[1]/\text{name}] \;=\; & \Pi^D((\Box_{cn:root(cn)}) \\ & < \Pi_{\overline{cn'}} \circ \Upsilon_{cn:cn'/\text{descendant::node()}}(\Box_{cn':cn}) > \\ & < \sigma_{cp=1} \circ \chi_{cp:counter(p_1)++} \circ \\ & \Pi_{\overline{cn'}} \circ \Upsilon_{cn:cn'/\text{child::student}}(\Box_{cn':cn}) > \\ & < \Pi_{\overline{cn'}} \circ \Upsilon_{cn:cn'/\text{child::name}}(\Box_{cn':cn}) >)\end{aligned}$$

Clauses with $last()$

The most difficult case are clauses that contain $last()$. Here we have to compute the context size to be able to evaluate the clause. We do this with the help of our new Tmp^{cs} operator that first materializes the context and then adds a context size attribute cs to all the tuples belonging to the current context. In the canonical translation, the context is exactly the result of the dependent subexpression of the current location step. Hence, on a logical level Tmp^{cs} is just shorthand for[3]

$$Tmp^{cs}(e) := \mathfrak{A}_{cs;count}(e) \overline{\times} e$$

This leads to the translation of a predicate relying on full positional information as

$$\Phi[l_{k1} \wedge \cdots \wedge l_{km_k}] \;:=\; \sigma_{\mathcal{T}[l_{km_k}]} \circ \cdots \circ \sigma_{\mathcal{T}[l_{k1}]} \circ \text{Tmp}^{cs} \circ \chi_{cp:counter(p_k)++}$$

with

$$\mathcal{T}[last()] := cs$$

2.2.4 Filter Expressions

XPath allows to filter any expression of type node-set using predicates. As with location path predicates, we use a different translation in case there are position-based clauses.

Without Position-Based Predicates

If the predicates p_i in the filter expression $e[p_1] \ldots [p_h]$ do not contain $position()$ or $last()$, we have as translation:

$$\mathcal{T}\,[e[p_1] \ldots [p_h]] := \Phi[p_h] \circ \cdots \circ \Phi[p_1] \circ \mathcal{T}[e]$$

Note that the sequence-valued e already has cn bound to the correct node, so we do not need to add a map operator.

[3] We explain in Sec. 2.4.2 how to implement Tmp^{cs} efficiently.

With Position-Based Predicates

Position-based predicates in filter expressions are evaluated with respect to the child axis, i.e. in document order. In location step predicates, the input sequence (the context) always results from a single location step and, hence, is properly ordered. In filter expressions, the input sequence may contain an arbitrary node sequence. To make the counting mechanisms from Sec. 2.2.3 work for filter expressions, we must guarantee that the input sequence is in document order. Hence, we introduce a sort operator which establishes document order before evaluating the predicates[4]. So, if there is any predicate p_k in the filter expression $e[p_1]\ldots[p_h]$ containing $position()$ or $last()$, its translation is

$$\mathcal{T}\,[e[p_1]\ldots[p_h]] := \Phi[p_h] \circ \cdots \circ \Phi[p_1] \circ Sort_{cn}(\mathcal{T}[e])$$

2.2.5 Path Expressions

Path Expressions are a more general form of relative location paths. They comprise a node-set expression e and a relative location path π. All the nodes in the node-set are used as context nodes for the location path, and a union of the results is returned.

Our translation of path expressions uses a d-join to feed all nodes from e as context nodes to the relative location path:

$$\mathcal{T}[e/\pi] := \Pi^D(\mathcal{T}[e] <\mathcal{T}[\pi]>)$$

The duplicate elimination operator is required since the evaluation of π for several context nodes may introduce duplicates, just as in location paths.

Note that the tuple sequence from e has an attribute cn containing the nodes it provides.

2.2.6 Function Calls

We distinguish between simple function calls, node-set-based function calls and node-set-valued function calls. Simple function calls are characterized by the fact that they neither get node-sets as parameters nor return node-sets, while node-set-based function calls have node-sets as parameters and return simple values. Node-set-valued function calls may return node-sets.

Simple Functions

Examples for simple functions in XPath are functions to deal with strings, numbers, or Boolean values (e.g. `string-length`, `floor`, `ceiling`, `true`, `false`,

[4]The input sequence may already be in document order, for example because it resulted from a location path that returned a sorted result[59]. We defer the determination of interesting orders in XPath and the resulting optimization of sort operations, as we are primarily concerned with a complete translation, but not an optimized one yet.

etc.). They are mostly used as subscripts of algebraic operators. Translating simple functions is quite straightforward (f is translated into its algebra counterpart):

$$\mathcal{T}[f(e_1,\ldots,e_n)] := f(\mathcal{T}[e_1],\ldots,\mathcal{T}[e_n])$$

Node-Set-Based Functions

We classify the node-set-based function calls further into aggregate functions and comparison operators between two node-sets. If f is an aggregate function (sum or count in XPath), we translate it into the corresponding aggregate operator $\mathfrak{A}_f$ of our algebra. $\mathfrak{A}_f$ aggregates the tuples of the input node-set applying the function f and returns a tuple containing the answer. Let e be a node-set value, then

$$\mathcal{T}[f(e)] \quad := \quad \mathfrak{A}_{a;f}(\mathcal{T}[e]) \tag{2.1}$$

Our aggregation operator $\mathfrak{A}$ formally has sequence-valued input *and* output. However, here we use it according to our conventions (Sec. 2.1.2) as an atomic value. We explain in Sec. 2.4.2 how this conversion is actually implemented.

For the comparison operators on node-sets, it is important to know that they have an existential semantics. That is, if we can find two elements (one in each node-set) that satisfy the condition, the comparison operator returns true. To implement this, we have the additional internal aggregation functions $exists()$, $max()$ and $min()$. $exists()$ is boolean-valued and returns false for empty sequences and true otherwise. $max_a()$ and $min_a()$ return the maximum or minimum of an attribute a in a tuple sequence, where for node attributes, each node content is converted to a number by means of the $number()$ function.

For (in)equality, we have

$$\mathcal{T}[e_1\theta e_2] \quad := \quad \mathfrak{A}_{x;exists()}(\mathcal{T}[e_1] \ltimes_{cn\theta cn'} \Pi_{cn':cn}(\mathcal{T}[e_2])) \tag{2.2}$$

with $\theta \in \{=, \neq\}$.

For $\theta \in \{<, \leq\}$ (recall that the nodes produced by sequence-valued e_2 are assigned to attribute cn):

$$\mathcal{T}[e_1\theta e_2] := \mathfrak{A}_{x;exists()}(\sigma_{cn\theta\mathfrak{A}_{max_{cn}(\mathcal{T}[e_2])}}\mathcal{T}[e_1]) \tag{2.3}$$

Finally, for $\theta \in \{>, \geq\}$:

$$\mathcal{T}[e_1\theta e_2] := \mathfrak{A}_{x;exists()}(\sigma_{cn\theta\mathfrak{A}_{min_{cn}(\mathcal{T}[e_2])}}\mathcal{T}[e_1]) \tag{2.4}$$

Example For selecting good students, i.e. those with an exam that was graded better than 'B', let us translate the following XPath query p:
`//student[examination/@id = //exam[grade < 'B']/@id]`.

$$\begin{aligned}\mathcal{T}[p] \;=\; & \Pi^D((\Box_{cn:root(cn)}) \\ & < \Pi_{\overline{cn'}} \circ \Upsilon_{cn:cn'/\text{descendant::node()}}(\Box_{cn':cn}) > \\ & < \sigma_{\mathfrak{A}_{x;exists()}(\mathcal{T}[\text{examination/@id}] \ltimes_{cn=cn'} \Pi_{cn':cn}(\mathcal{T}[//exam[...]/@id]))} \circ \\ & \Pi_{\overline{cn'}} \circ \Upsilon_{cn:cn'/\text{child::student}}(\Box_{cn':cn}) >\end{aligned}$$

To anticipate, these translation equations are labeled because they are on target for our new optimizations described in Chapters 3 and 4.

Node-Set-Valued Functions

The only node-set-valued function in XPath 1.0 is $id()$. We translate $id()$ by first converting the input into a sequence of IDs. Then, the individual IDs are dereferenced using a dereference function which converts a single ID string into a node [5]. The result is a sequence-valued expression with the result nodes assigned to cn.

The input conversion depends on whether the input is of type node set or not.

For a node set input e, we just convert the nodes to strings:

$$\mathcal{T}[id(e)] := \chi_{cn:deref(string(c'))}(\Pi_{c':cn}(\mathcal{T}[e]))$$

In the case of an e which is not of type node-set, we convert e to a string and use unnest map with a tokenizing function to return the sequence of embedded string tokens:

$$\mathcal{T}[id(e)] := \chi_{cn:deref(t)}(\Upsilon_{t:tokenize(string(\mathcal{T}[e]))}(\Box))$$

2.2.7 Constants and Variables

Constants and variables are very easy to translate into our algebra. For the translation of a constant c we have $\mathcal{T}[c] = c$. This means that, the expression is left as is and no algebraic operator is necessary to process it. The same applies to XPath $ variables, as they are bound to values before evaluating expressions.

2.3 Improved Translation

After presenting the canonical translation into the algebra, we go into some details on how to improve the translation step. In particular, Gottlob et al. have shown how

[5] We do not elaborate on the implementation of $deref()$ here, as it depends too much on the details of the storage environment.

XPath expressions can be evaluated in polynomial time in the worst case [46, 47]. In this section, we reveal how this can be done in an algebra-based approach.

2.3.1 Pushing Duplicate Elimination

We divide all location steps into two different groups: one that potentially produces duplicates (ppd) and one that does not ($\neg ppd$). Axes that belong to ppd are:

- `following`,
- `following-sibling`,
- `preceding`,
- `preceding-sibling`,
- `parent`,
- `ancestor`,
- `ancestor-or-self`,
- `descendant`, and
- `descendant-or-self`.

Instead of such a simple axis-wise treatment, we could incorporate the work by Hidders et al. for determining if a sequence of location steps will produce duplicates or not [59]. We skip this because it does not affect asymptotical complexity and is straightforward to implement.

The canonical translation eliminates duplicates only in a final step to preserve the duplicate-free semantics of XPath location paths. However, duplicates may be generated after every single step. The single final duplicate elimination means that the effect of producing duplicates in several intermediate steps will multiply, as we generate duplicates of duplicates.

Hence, we introduce additional duplicate eliminations after ppd axes. This reduces the input size of the following steps. Also, the duplicate elimination works on smaller data sets. For the translation of a location path $\pi_0/a :: t$ with a step s comprised of axis a and node test t, this means:

$$\mathcal{T}[\pi_0/a :: t] := \begin{cases} \Pi^D(\mathcal{T}[\pi_0] < \mathcal{T}[a :: t] >) & \text{if } ppd(s) \\ \mathcal{T}[\pi_0] < \mathcal{T}[a :: t] > & \text{else.} \end{cases}$$

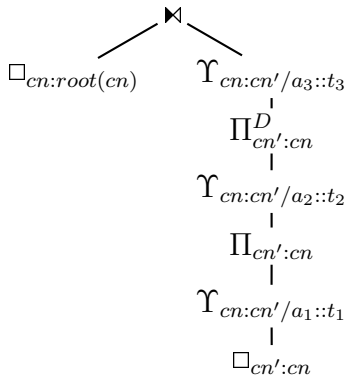


Figure 2.3: Stacked translation for $/a_1 :: t_1/a_2 :: t_2/a_3 :: t_3$ with $ppd(a_2 :: t_2)$

2.3.2 Location Paths

When improving on the translation of location paths, we have to distinguish between outer and inner paths. An *inner path* appears within a predicate, an *outer path* does not. We discern between these two cases because we can translate outer paths in a more efficient way, deviating from the canonical d-join translation. For inner location paths, we run the risk of having to evaluate expressions multiple times for the same context node. We avoid this by memoizing already evaluated paths.

Outer Paths

For outer location paths, we concatenate the evaluation of the steps, avoiding the overhead of a d-join operator. We replace the singleton scan in the dependent branch with the left subexpression of the d-join. For an outer path π/s, we get the following *stacked translation*:

$$\mathcal{T}[\pi/a :: t] := \begin{cases} \Pi^D_{cn':cn}(\Psi[a :: t](\mathcal{T}[\pi])) & \text{if ppd}(a) \\ \Pi_{cn':cn}(\Psi[a :: t](\mathcal{T}[\pi])) & \text{else,} \end{cases}$$

where $ppd(a)$ means that location step s with axis a potentially produces duplicates. The context nodes cn are now directly handed over from step to step instead of using a d-join to explicitly bind them. Fig. 2.3 shows an example for the translation of path $/a_1 :: t_1/a_2 :: t_2/a_3 :: t_3$, using the stacked translation if $a_2 :: t_2$ is potentially producing duplicates.

Inner Paths

When looking at inner location paths, we have to distinguish between relative and absolute inner paths. A *relative inner path* gets its context from the corresponding step of the outer path, an *absolute inner path* sets its own context. The translation

of the actual inner path then takes place during the translation of the predicate (see also Section 2.2.3).

Absolute inner paths can be translated like outer paths, while we have to avoid unnecessary work in relative inner paths as follows.

In the XPath expression

$$\pi[\text{count}(./\text{descendant} :: c/\text{following} :: *) = 1000],$$

when evaluating the predicate for the context nodes produced by π, we may reach the same c elements over and over again, computing and counting the nodes produced by the `following` axis multiple times.

For the location path $\pi_0[\pi_1/s]$, the inner path is translated as $\mathcal{T}[\pi_1] < \mathcal{T}[s] >$. We want to avoid computing the right-hand side of the d-join when getting handed a context node from the left-hand side for which s was already evaluated before.

In order to avoid this, we apply a memoization strategy using a *MemoX operator* ($\mathfrak{M}$). In contrast to the memoizing function call operator from [57], the MemoX operator is a *sequence-valued* unary operator typically used in the dependent subexpression of a d-join. It is subscripted with a set of variables which are free in its producer expression.

Every time the MemoX operator is evaluated, it checks if the variables have already been bound with these specific values in a prior evaluation. If not, the MemoX operator evaluates the subexpression and stores the result in an associative data structure with the given variable values as key. Finally, it also returns the result to its consumer. If the same variable values have already been used in an earlier evaluation of the MemoX operator, it just looks up the previously computed result and returns it without engaging the producer operator.

In the case of the translation of inner paths, the producer operator is the next location step, and the free variable is the current context node from the previous location step. So the translation of the inner path s/π_1 actually looks like:

$$\mathcal{T}[s/\pi_1] \quad := \quad \begin{cases} \mathcal{T}[s] < \mathcal{T}[\pi_1] > & \text{if } \neg ppd(s) \text{ and } \neg ppd(\pi_1) \\ \Pi^D(\mathcal{T}[s] < \mathfrak{M}_{cn}(\mathcal{T}[\pi_1]) >) & \text{if } ppd(s) \text{ and } \neg ppd(\pi_1) \\ \Pi^D(\mathcal{T}[s] < \mathcal{T}[\pi_1] >) & \text{if } \neg ppd(s) \text{ and } ppd(\pi_1) \\ \Pi^D(\mathcal{T}[s] < \mathfrak{M}_{cn}(\mathcal{T}[\pi_1]) >) & \text{if } ppd(s) \text{ and } ppd(\pi_1) \\ \mathcal{T}[s] & \text{if } \pi_1 \text{ is empty} \end{cases}$$

2.3.3 Predicate Evaluation

Predicates and Stacked Translation

In the canonical translation, all tuples produced on the right hand side of the d-join for a given tuple on the left hand side belong to the same context. So all contexts are clearly separated from each other by separate evaluations of dependent d-join

subexpressions. This makes it easy to determine context position and context size by counting the tuples in one complete evaluation of a dependent subexpression.

In the stacked translation, all tuples belonging to a location step are part of the same tuple stream flowing through the pipeline of operators. The different contexts are separated by the input context nodes; a new context begins whenever the input context node cn changes. This requires a slightly different handling of predicate evaluation.

Whenever a cn value different from the last processed tuple is detected, the map performing the position count must reset its counter.

We also have to be careful when evaluating predicates of outer location paths containing `last`. As we have already mentioned in Section 2.3.2, outer location paths are evaluated in a stack-based fashion. The Tmp^{cs} operator now has to be able to recognize the boundaries of the contexts. For this task, we define a Tmp^{cs}_{cn} operator parameterized with the context node attribute cn:

$$\text{Tmp}^{cs}_{cn}(e) := e \mathbin{⋉\!\!\!\Gamma}_{cs;cn=cn'';count} \Pi_{cn'':cn}(e)$$

This operator performs the same task as Tmp^{cs} but does not aggregate the whole input sequence. It only aggregates those tuples that were generated for the same context node cn.

When evaluating predicates, the new operator is used in the same way as the Tmp^{cs} operator in Sec. 2.2.3 but is parameterized with the input context node as Tmp^{cs}_{cn}.

Example Fig. 2.4 shows the graphical representation of a translation for the complex XPath expression `//student[exam][position()=last()]/name` using stacked translation.

Avoiding Evaluation of Expensive Predicates

We classify the clauses l_{kj} of a predicate p_k into the sets

$$\begin{aligned}
\text{cheap}(p_k) &:= \{l_{kj} \mid l_{kj} \text{ is cheap to evaluate}\} \\
\text{exp}(p_k) &:= \{l_{kj} \mid l_{kj} \text{ is expensive to evaluate}\} \\
\text{pos}(p_k) &:= \{l_{kj} \mid l_{kj} \text{ contains } \texttt{position()}, \\
&\qquad\qquad \text{but no } \texttt{last()}\} \\
\text{last}(p_k) &:= \{l_{kj} \mid l_{kj} \text{ contains } \texttt{last()}\}
\end{aligned}$$

For classification into cheap() and exp(), a simple cost model is used which considers the number of instructions that are necessary to evaluate a clause. For the translation of a predicate $p_k = l_{k1} \ldots l_{km_k}$, this means

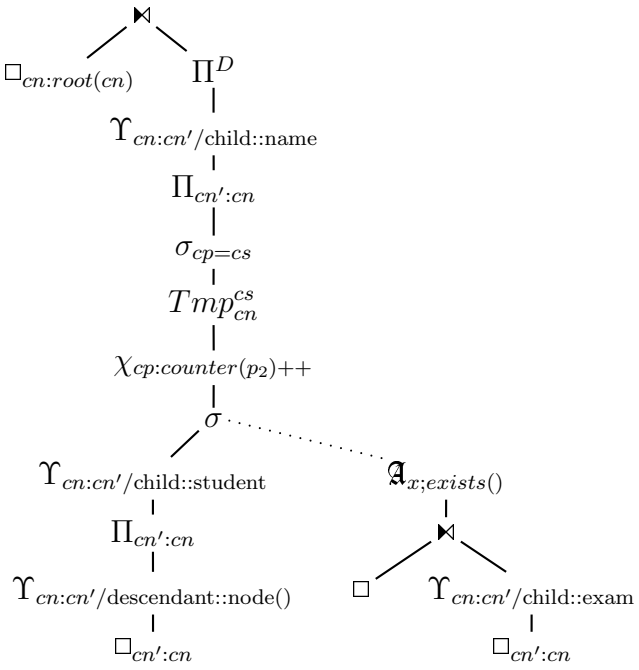


Figure 2.4: Translation of $//student[exam][position() = last()]/name$

$$\Phi[p_k] := \sigma^{mat}_{exp(p_k)} \circ \overbrace{\sigma_{cheap(p_k)\cap last(p_k)} \circ \mathrm{Tmp}^{cs}}^{\text{only for } \texttt{last}} \circ \sigma_{cheap(p_k)\setminus last(p_k)} \circ \underbrace{\chi_{cp:counter(p_k)++}}_{\text{only for } \texttt{pos} \text{ or } \texttt{last}}$$

Each expensive expression e in a clause l_{kj} that is a member of $\texttt{exp}(p_k)$ is replaced by a variable v. We compute the value of v with the help of χ_{mat} operators. The operators memoize function evaluation results similar to the approach by Hellerstein and Naughton [57]. In the translation above, $\sigma^{mat}_{exp(p_k)}$ is an abbreviation for this sequence of χ_{mat} operators and the final selection.

In the above translation, Tmp^{cs} has to be replaced by Tmp^{cs}_c if the predicate occurs in a stacked translation.

2.4 Implementation

2.4.1 Compiler

The translation was implemented as an XPath compiler module written in C++, taking XPath expressions as strings and generating an execution plan for the NQE (see

below). The compilation process comprises six steps, listed here with some of the tasks they perform: (1) Parsing (generating an abstract syntax tree (AST)) (2) Normalization (classifies and sorts predicates as explained in Sec. 2.2.3 and 2.3.3) (3) Semantic analysis (4) Rewrite (constant folding) (5) Translation into algebra (6) Code generation (generate NQE execution plan). The query is handed from step to step using a single data structure, starting out as an AST which is annotated and modified until it has become an algebraic expression. The resulting expression is traversed by step (6), which returns an execution plan in the NQE syntax. A detail worth noting is that our translation includes a lot of map and projection operations, particularly to guarantee that the context node attribute is always called cn, and cn' is projected away. The compiler does not emit actual copy operations in these cases. Instead, an attribute manager which is part of the compiler ensures that code emitted for aliased attribute accesses uses the proper memory locations directly.

2.4.2 Physical Algebra

The Query Execution Engine (NQE) of the Natix system implements the logical algebra from Sec. 2.1.2 in C++. Below, we focus on implementation aspects relevant for XPath. More details can be found in [40], [58], and [83].

Iterators

All the sequence-valued operators in our logical algebra (Fig. 2.1) have a corresponding implementation as an *iterator* [49] in the physical algebra. Whenever possible, they avoid to copy and/or materialize intermediate results, passing them by reference and/or in a pipelining fashion.

Natix Virtual Machine

The remaining (i.e. non-sequence valued) operators of our logical algebra are implemented using assembler-like programs interpreted by the Natix Virtual Machine (NVM). XPath basic type functions and operators are evaluated using single NVM commands or small command sequences.

Location step navigation and node tests are performed via NVM commands that directly access the persistent representation of the documents in the Natix page buffer, thus avoiding an expensive representation change into a separate main memory format. In the buffer, the XML documents are stored in a recoverable, updatable form which does not require a fixed DTD. There are also NVM commands for access to text node contents. However, we transcode the stored, space-saving string encoding to UTF-16, which is the encoding used for strings in NVM.

Nested Iterators

NVM programs are primarily used to evaluate non-sequence-valued subscripts of iterators, and the NVM commands operate on tuples. Sometimes subscripts also need to evaluate XPath functions that have sequences as input, and to convert the sequence-valued result of the $\mathfrak{A}$ operator into an atomic value. The NVM provides commands that can access results of nested iterators.

Context Size Operators

In Sec. 2.2.3 and Sec. 2.3.3, we introduced the special operators Tmp^{cs} and Tmp^{cs}_{c}, which determine the context size and concatenate it to all output tuples. The logical definition of these operators requires the input sequence twice, once for determining the number of tuples, and once to return the actual result annotated by the size.

The actual implementation does not evaluate the input context twice. Instead, each context is evaluated once and then materialized. Note that the input tuples for Tmp^{cs}_{c} always contain cp, the position counter. The cp value of the final tuple equals the context size cs, which is remembered. When delivering the result, the materialized sequence is reread, adding cs to each tuple as it is returned.

Tmp^{cs} and Tmp^{cs}_{cn} only differ in how they determine the input context to materialize. Tmp^{cs} counts the complete input sequence, while Tmp^{cs}_{cn} only materializes those input tuples generated for the same input context node. The materialization stops when the input context node attribute c changes (compare Sec. 2.3.3).

Actually, there is just one implementation Tmp^{cs}_{cn}, which covers Tmp^{cs} as a special case.

Smart Aggregation

The aggregation functions used by $\mathfrak{A}$ are also implemented as small NVM programs. The interface between the $\mathfrak{A}$ operator and these programs allows to signal a premature end of the aggregation. For example, when evaluating an $exists()$ function, it is not necessary to evaluate the complete argument sequence. If one tuple is found, the remaining input sequence may be ignored, and the $\mathfrak{A}$ operator may return $true$.

2.5 Evaluation

Our ultimate goal in providing a complete algebraic translation is performance. While we do not discuss advanced optimization techniques on the algebraic level in this chapter (see Chapters 3 and 4), another performance-related aspect of the algebraic approach is the fact that queries can be executed in a scalable way through an iterator-based approach.

To verify that this goal has been met, we compared our implementation against some purely main-memory based XPath interpreters. We chose those freely avail-

Number	Path
1	`/child::xdoc/desc::*/anc::*/desc::*/@id`
2	`/child::xdoc/desc::*/pre-sib::*/fol::*/@id`
3	`/child::xdoc/desc::*/anc::*/anc::*/@id`
4	`/child::xdoc/child::*/par::*/desc::*/@id`

Figure 2.5: Queries against generated documents

able interpreters which support the complete XPath specification, including all axes, namely `xsltproc` (libxslt 1.1.2) and Xalan 1.6.0.

We are aware that the following is not a comprehensive performance evaluation, and that it leaves open a lot of questions. The measurements below are intended to give a proof-of-concept of our approach, and we gathered them only to make sure that we are on the right track.

2.5.1 Environment

The environment used to perform the experiments consisted of a PC with an Intel Pentium 4 CPU at 2.80GHz and 1 GB of RAM, running Linux 2.6.4. The Natix C++ library and the test executable were compiled with gcc 3.3 at the O2 optimization level.

2.5.2 Results

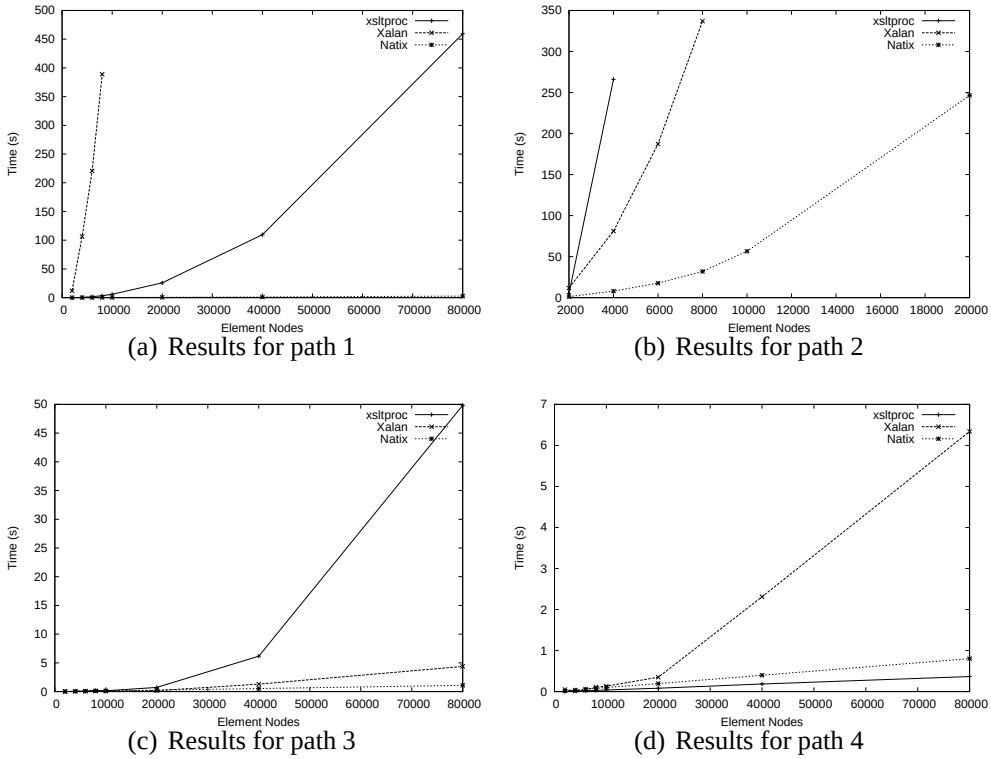
Below, we list the time needed to compile and execute a query. To make them comparable across the different evaluators, the times do not include the time to parse/load the document. The measurements are averaged over several runs.

Generated Documents

The documents on which the queries in Fig. 2.5 are executed were generated. They differ in the number of elements, fanout and document depth. The document generator follows a breadth first algorithm and fills every depth of the document with the given fanout until the maximum number of elements or depth is reached. The root element of every document has the name `xdoc`. Every element contains an attribute `id` which is consecutively numbered.

The concrete documents having between 2000-8000 elements were generated with a fanout of six and a depth of four. The documents having between 10000-80000 elements were generated with a fanout of ten and a depth of five.

The queries were obtained by systematically generating all XPath location paths of length 3 with a node test checking for any element node in each step. There are several typical patterns in the results, and we selected sample queries as examples for these patterns.



(a) Results for path 1 (b) Results for path 2

(c) Results for path 3 (d) Results for path 4

Figure 2.6: Results for paths 1-4

Fig. 2.6(a)–Fig. 2.6(d) show the selected results. Without going into detail, we observe (1) that we can keep up with the performance of the main-memory based interpreters, (2) the high memory requirements sometimes cause the main-memory interpreters to fail on large documents (this is the reason the curves sometimes stop before reaching the end of the x-axis), and (3) the constants in the asymptotic behavior of the algebraic approach are promisingly small.

In some queries like the one in Fig. 2.6(d), one or both main-memory evaluators outperform Natix by a constant factor. Profiling NQE has provided us with hints on how to lower this factor. Specifically, returning the query result in our implementation involves several unnecessary conversions and memory allocations, caused by unoptimized Unicode support. This points out some optimization potential.

DBLP Data

Fig. 2.7 shows the execution times for queries executed on DBLP data collected in one big XML document [75]. This document has a size of 216 mega-bytes.

`xsltproc` was not able to load the document, probably due to memory requirements.

As above, the results are promising, as we can compete with a main-memory-

based evaluation. For some queries (those below the horizontal line in the center), we are slower. Again, profiling points to engineering details in NQE: output processing and string management seem to be the cause. Here, the problem is even worse than in the previous section, because the string representation in NVM is UTF-16, and we transcode the contents of the stored nodes into UTF-16 for comparison. As a remedy, we plan to supply NVM with the ability to transcode the string constant from the query into the persistent encoding, reducing the comparison overhead (instead of tens of thousands of comparisons).

	time[s]	
Path	Xalan	Natix
/dblp/article/title	6.50	3.97
/dblp/*/title	17.73	8.10
/dblp/article[position() = 3]/title	24.51	1.51
/dblp/article[position() < 100]/title	25.22	1.55
/dblp/article[position() = last()]/title	23.99	2.22
/dblp/article[position()=last()-10]/title	24.32	2.31
/dblp/article/title/dblp/inproceedings/title	157.98	14.23
/dblp/article[count(author)=4]/@key	0.9	2.91
/dblp/article[year='1991']/@key | /dblp/inproceedings[year='1991']/@key	3.90	8.69
/dblp/*[author='Guido Moerkotte']/@key	4.2	9.78
/dblp/inproceedings[@key='conf/er/LockemannM91']/title	3.22	4.28
/dblp/inproceedings[author='Guido Moerkotte'] [position()=last()]/title	4.59	6.71

Figure 2.7: Results (in sec.) of queries against DBLP

2.6 Related Work

The approaches for evaluating XPath can be divided into several different categories:

First of all, we have (main-memory-based) interpreters (e.g. Xalan, XSLTProc). Although most of them support the full XPath standard, they have high memory requirements and do not scale very well to large documents.

Second, many papers were published investigating the efficient evaluation of individual location steps [2, 21, 58, 67]. For some location steps, very efficient operators have been developed, but a complete framework for supporting the full XPath standard still seems to be missing, e.g. there is no support for nested expressions or position-based predicates.

Third, we have approaches relying on relational databases [41, 52, 55, 103]. Here, the XML data is transformed and stored in relations. Queries containing XPath expressions are translated into SQL and processed using the (possibly extended) engine of the underlying database system.

Fourth, there are other algebra-based approaches for the evaluation of queries over XML data [7, 64, 98], However, these approaches give no translation function for all of constructs of XPath, in particular the whole set of axes.

Finally, there exist streaming-based approaches for evaluating XPath [34, 51, 66]. Some of these approaches are used for document filtering [34, 51] using XPath. Document filters do not require to retrieve the full result of the query, but only have to make a binary decision if there exists at least one result or not. The approach by Josifovski et al. [66] (called TurboXPath) is also used in the streaming context. Hence, they have to buffer their input for evaluating specific predicates or backward axis. A discussion about memory requirements in the streaming context is given in [5]. TurboXPath follows a holistic approach, i.e. it uses a single operator for evaluating XPath expressions. Moreover, their approach is specifically designed for evaluating multiple XPath expressions (for example in an XQuery) in a single run over the according input. However, because of following a holistic approach, they have limitations when it comes to advanced optimization techniques (e.g. unnesting) as described in the next chapter. TurboXPath is the foundation for the XNav operator used in DB2 [8].

Further, there are optimizations that are orthogonal to our approach and can be integrated. Mathis et al. have shown how to integrate structural joins ([2]) into our algebraic approach and, amongst others, use them for evaluating predicates [78]. Moreover, there exist path rewriting techniques [58, 90], schema-based rewritings [9, 73], and equivalences that use properties of the intermediate results to avoid duplicate elimination and sorting [59].

Gottlob et al. have shown that the presence of duplicates during evaluation may lead to an exponential run-time [46, 47, 48]. Their bottom-up approach in [46] computes all potential contexts to avoid an exponential runtime. However, they perform unnecessary computations. Their top-down approach is not algebraic and requires materialization of all intermediate results, which we circumvent by using our algebraic approach. Helmer et al. presented an approach to avoid the generation of duplicates during evaluation [58]. Their approach is orthogonal to our approach and can be integrated by modifying the translation function for location steps. Diao et al. developed a memoization approach for avoiding the computation of equal or shared XQuery expressions [35]. Their Memo Table is similar to our approach. Specifically, it is a combination of our MemoX and MapMat operators. The MapMat operator is similar to the approach by Hellerstein and Naughton [57].

The binary grouping operator has been introduced in [29, 28, 102]. An analysis and implementations for binary grouping can be found in [83].

2.7 Conclusion

In this chapter, we have explained how to translate XPath queries into algebraic expressions. The proposed translation method covers the complete set of XPath features including all axes, position-based predicates, nested paths, filter expressions, general path expressions, and node-set functions.

Apart from providing an effective translation as a first step, we were also concerned with efficiency. In a second step, we extended our simple approach, achieving polynomial worst-case complexity. To this end, we incorporated the memoization techniques pioneered by Gottlob et al. [47] in the context of XPath interpreters.

We implemented an XPath compiler based on the concepts outlined in this chapter. A complementary iterator-based physical algebra was used to evaluate the generated algebraic query plans. First measurements demonstrate that our approach is viable.

Having established that an algebraic approach to XPath is reasonable, we can now turn to the next challenges. In the next two chapters, we present algebraic optimization techniques for optimizing XPath expressions.

Chapter 3

Unnesting XPath Expressions

In the previous chapter, we presented a translation of XPath into algebraic expressions. Our translation is complete and avoids the exponential runtime behavior of the naïve evaluation (e.g. [47]). However, there are still a lot of shortcomings and, hence, there is a lot of optimization potential. In this chapter, we illustrate these shortcomings and present algebraic techniques to fix them.

Up to now, optimizing the evaluation of XPath addressed the evaluation of chains of simple location steps (e.g. [2, 21, 54, 55, 66]), streaming-based approaches for evaluating XPath (e.g. [34, 51, 66]), or techniques for avoiding unnecessary work (e.g. [47, 46, 58]). However, when used as a standalone language or within XSLT, the expressions used to formulate queries can become quite complex, containing many predicates. Let us demonstrate this with a sample XPath query (called Q1).

```
//student[examination/@id = //exam[grade<'B']
  [belongsto/@lecture = //lecture[title='NCS']/@id]          Q1
    /@id]/name
```

Q1 selects all students who have taken an exam in 'Numerical Computer Science' (NCS) and have achieved a grade better than 'B'. The naïve interpretation employs a strategy that evaluates the steps in order of their occurrence in the query. Expressions within predicates are evaluated for every result of the corresponding location step. In a first step, all students are selected (which can be quite a large number). Then, for each student, we select the exams that have a grade better than 'B' and belong to a lecture with the title NCS (the predicate checking for the lectures is again nested). Obviously, evaluating the above query in a nested fashion is not very efficient. As we show, we can improve the performance significantly by unnesting nested XPath expressions.

Therefore, the approach presented in this chapter is as follows: First, we give a complete classification of nested XPath expressions and introduce six new algebraic operators that are particularly useful for unnesting. Second, based on this classification, we introduce novel rewrite and optimization techniques for each class. We

present them by means of algebraic equivalences that allow to unnest nested algebraic expressions. The left-hand side of our equivalences results from the (canonical) translation introduced in the previous chapter. Pursuing an algebraic approach for unnesting has two basic advantages: (1) We can (partially) reuse current optimization techniques (e.g. [29, 82]), and (2) we can supply rigorous correctness proofs for the equivalences (see Appendix A). Usually, our rewrite techniques will improve performance. However, contrary to the situation in unnesting SQL, OQL, or XQuery, there will be cases where performance is difficult to improve by unnesting. In these cases, we refer to [106] for optimizations. Third, we validate our techniques by presenting performance measurements showing the improvements that are possible. This way, we also compare our approach with several other XPath evaluators.

The remainder of this chapter is organized as follows. In the next section, we establish a classification scheme for nested XPath expressions. After that, we present six algebraic operators that we use for unnesting (see Sec. 3.2). The core of this chapter is Section 3.3. It covers the actual optimization techniques by means of algebraic equivalences. For convenience, two figures contain all equivalences. In Section 3.4, we show the effectiveness of our approach by comparing it with other approaches using example queries. Section 3.5 discusses related work. At the end of this chapter, Section 3.6 summarizes our contributions.

3.1 XPath Expressions

First, we establish a classification of nested XPath expressions. In later sections, we develop optimization techniques for each of these classes. In more detail, the outline of this section is as follows: In order to keep the number of classes low, we first normalize XPath expressions. Then, we classify expressions by two properties of expressions: *dependency* on the local context and expected *result cardinality*. The expected result cardinality distinguishes between expressions returning a single value and expressions that return a node-set. The former are further subdivided into those that contain an aggregate function as the top most operator and those that do not. The final step then consists of deriving the classification of nested binary predicates. Since normalization turns unary predicates (i.e. expressions returning a boolean) into binary predicates, this covers all cases.

3.1.1 Normalization

Normalization performs the following steps:

Explicit Comparisons Makes all implicit comparisons explicit:

- If an expression evaluates to a number, a comparison with the context position is introduced.

- If an expression of a non-comparison expression is a boolean, a comparison with `true` is inserted.
- If an expression evaluates to a string, its result is converted to boolean.
- If an expression evaluates to a node or a node-set, an `exists` function call is introduced, comparing the result with true. Note that there is no `exists` function call in XPath. It is an internal function of ours.

Predicate CNF Transforms predicates into conjunctive normal form (CNF).

Extremum Rewrite The query is rewritten using maximum and minimum aggregation functions. For instance, a nested expression of the form `not(a < b)` is rewritten into `a` $\geq$ `max(b)` if a is single-valued. Similarly to the `exists` function call, the `max` function call is not defined in XPath 1.0, but inserted for internal use.

3.1.2 Expression Classification

The basic construct of XPath is an expression. We distinguish expressions according to their dependency on the context and the cardinality of their result.

Context Dependency

An important concept in XPath is the context. Every expression is evaluated with respect to a given context. For the purposes of this chapter, it suffices for a context to contain three components: the *context node* (cn), the *context position* (cp), and the *context size* (cs). The evaluation of an XPath expression requires a *global context* to be specified. Subexpressions of the top-level query are evaluated with respect to a *local context*. The local context is derived from the result of the evaluation of a former step or filter expression. We distinguish two classes of subexpressions:

Dependent Expression: Expressions depending on a local context that is different from the global context (denoted by *Type D*).

Independent Expression: Expressions depending only on the global context (denoted by *Type I*).

Location steps, relative location paths, calls to `position()` and `last()` are evaluated with respect to the local context. All other expressions are evaluated with respect to the global context.

In our example query Q1, the subexpression `//student` is independent. The last location step `name` depends on the local context resulting from the evaluation of the previous step and hence is dependent. Further, the absolute location path `//exam` is independent, whereas location path `examination/@id` within the first predicate is dependent.

Cardinality

For every XPath expression, the static type analysis can detect the type of its result. An expression has either one of the three atomic types (`boolean`, `number` or `string`) or is node-set typed. We distinguish expressions according to the cardinality of their result.

Single-valued expressions with no top-level aggregate function are classified as *Type S*, and those with a top-level aggregate function are classified as *Type A*.

Set-valued expressions (*Type M*) have a node-set as result. If an empty set is detected, its expression is associated with the class of set-valued expressions.

Detecting a single value or empty node-set requires schema information and is beyond the scope of this thesis.

In our example query from the introduction, the attributes `id` are single-valued (Type S), whereas the path expression `//student` has a set-valued result (Type M).

Combined Expression Classification

We have presented the two expression properties dependency and cardinality of the result. Similar to Kim [70] within the relational context and Cluet et al. [28, 29] in the OO-context, we can combine these properties and define the following four types of expressions:

- **Type I[S|A]** Independent subexpressions that return a single element, i.e. it either comes from a top-level aggregate (Type IA) or not (Type IS).
- **Type D[S|A]** Dependent subexpressions that return a single element that comes from a top-level aggregate (DA) or not (DS).
- **Type IM** Independent subexpressions that have a set-valued result.
- **Type DM** Dependent subexpressions that have a set-valued result.

Each of these types has individual properties that can particularly be exploited by the optimization techniques developed in Sec. 3.3.

3.1.3 Nested Expression Classification

Our primary focus of this work is the optimization of comparison-based predicates.

A predicate p_k can consist of arbitrary XPath expressions and is represented as a conjunction of several clauses l_{kj} (i.e. $p_k = \bigwedge_{j=1}^{m_k} l_{kj}$). The CNF results from our normalization phase. After making all implicit comparisons explicit, each conjunct

consists of binary comparison expressions. They exhibit the comparison operators $\in \{=, \neq, <, >, <=, >=\}$. Note, that if either of the operands is set-valued, the comparison operator has an existential semantics.

Corresponding to the previous classification, we establish a classification for comparison expressions. The properties of this classification allow for particular optimization techniques of each of the classes. The classification simply consists of all combinations of the previously introduced types and is shown in the following table.

		Independent		Dependent	
		I[S|A]	IM	D[S|A]	DM
Independent	I[S|A]	I[S|A]/I[S|A]	I[S|A]/IM	I[S|A]/D[S|A]	I[S|A]/DM
	IM	IM/I[S|A]	IM/IM	IM/D[S|A]	D[S|A]/DM
Dependent	D[S|A]	D[S|A]/I[S|A]	D[S|A]/IM	D[S|A]/D[S|A]	D[S|A]/DM
	DM	DM/I[S|A]	DM/IM	DM/D[S|A]	DM/DM

This table can be divided into three groups. The first group (top left) is composed only of independent operands. The second group (top right and down left) includes one independent and one dependent operand. For expressions in this group, we introduce the term *semi-independent*. The last group (down right) is made up only of dependent operands.

Nested comparison expressions are a crucial factor for an efficient evaluation of XPath predicates. This has several reasons:

- Expressions are evaluated in a nested fashion. In particular, expressions can be deeply nested, and many execution orders for independent expressions may exist. Some of them are more, some less efficient.
- Further, based on the classification, we observe that
  - independent comparisons (Types I[S|A] and IM) are evaluated unnecessarily for every context resulting from the outer expression.
  - comparisons with an independent and a dependent argument are a *correlation predicate* between the outer and the inner expression.
  - expressions which share common paths or consist of common subexpressions would require their evaluation more than once.

3.2 Logical Algebra

Our optimizations in Section 3.3 are based on the algebra described in the previous chapter. Moreover, all optimizations are presented by means of algebraic equivalences. They have the canonical translation (also presented in the previous chapter)

on the left-hand side. However, for our unnesting techniques, we need some additional algebraic operators. In this section, we formally define them. Before defining the operators, we repeat the notations that are used for their definition.

Our algebra is defined on sequences of tuples. A sequence-valued expression e results in several tuples which all have the same attributes $\mathcal{A}(e)$. The attributes of a single tuple are also referred to as $\mathcal{A}(t)$. Within the definitions, A (A_i) abbreviates $\mathcal{A}(e)$ ($\mathcal{A}(e_i)$). The projection of a tuple onto a set of attributes A is denoted by $|_A$. In the context of projections ($|$, Π), overlined attributes are discarded, i.e. only the complement is retained. A single tuple is constructed using the brackets $[\cdot]$. The concatenation of tuples and functions is denoted by $\circ$. For a sequence e, $\alpha(e)$ returns the first element of the sequence and $\tau(e)$ the tail of the sequence. Sequences are concatenated using $\oplus$. Given a sequence of non-tuple values e, we construct a sequence of tuples by $e[a]$. It is empty if e is empty. Otherwise $e[a] := [a : \alpha(e)] \oplus \tau(e)[a]$. In the following, e, e_1, and e_2 are sequence-valued expressions.

First, we define the **left outer-join**:

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

The left outer-join is used for unnesting queries with aggregation functions. It is needed to prevent the "count bug" [71]. Specifically, the left outer-join is a regular join with the exception that all tuples in e_1 that do not find a join partner in e_2 also contribute to the result. Those tuples are concatenated with $[\mathcal{A}(e_2) \setminus \{g\}] \circ [g : f(\emptyset)]$. The function $f(\emptyset)$ in the superscript of the operator assigns a meaningful value to the attribute g of those tuples. For example, if f is the function count, $\mathrm{count}(\emptyset)$ assigns 0 to g. For construction, the attributes $\mathcal{A}(e_2)$ are initialized with NULL by default.

In our optimizations, we use the left outer-join together with a unary grouping operator. Our **unary grouping** operator is defined in terms of the binary grouping operator ($\mathbin{⋈\!\!\!\Gamma}$; see Section 2.1.2):

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

In this definition, A and A' are sets of attributes ($A \in \mathcal{A}(e)$ and $A' \in \mathcal{A}(e)$) called grouping attributes. For each group (i.e. those tuples having the same value for all grouping attributes A), Γ computes an aggregation function and adds the result to the attribute g.

Fig. 3.1 presents an example for the unary grouping operator. It shows the result of applying the unary grouping operator $\Gamma_{g;=\mathrm{A};\mathrm{count}}$ to the sequence e_1. The result is a sequence of three tuples, each having two attributes, i.e. A and g. Each tuple results from one of the three distinct values in e_1. The attribute g contains the number of tuples having equal values for A.

e_1

A
1
1
3
3
4

$\Gamma_{g;=\mathrm{A};\mathrm{count}}$

A	g
1	2
3	2
4	1

Figure 3.1: Unary grouping example

To avoid unnecessary navigations in predicate expressions, we define the max operator. The **max** operator was originally introduced in [29].

$$Max_{g;m;a=}(e) = [m : max(\{x.a | x \in e\}), g :< [g' : x] | x \in e, x.a = m >]$$

The result of the operator has two attributes: One containing the maximum value m for a group of nodes and one (g) containing the resulting nodes that are equal to the calculated maximum m. The motivation for the max operator will become clear in later sections. The **min** operator is defined analogously.

Some of our unnesting techniques introduce duplicates of tuples that must not be in the result. To identify such falsely introduced tuples, we use the auxiliary ν operator (cf. ''[82]). It numbers tuples in a sequence by adding an additional attribute A holding their position (pos) in the sequence. It is defined as follows:

$$\nu_A(e) := \alpha(e) \circ [A : pos] \oplus \nu_A(\tau(e))$$

After unnesting, we have to eliminate multiple occurrences of tuples that have the same attribute value for A. However, we want to keep the order of the input. Therefore, we introduce an **order-preserving duplicate elimination projection** Π^A_B:

$$\Pi^A_B(e) := \begin{cases} \alpha(e)|_B \oplus \Pi^A_B(\tau(e)) & \text{if } \tau(e) = \epsilon \text{ or} \\ & \alpha(e).A \notin \alpha(\tau(e)).A \\ \Pi^A_B(\tau(e)) & \text{otherwise} \end{cases}$$

This projection keeps the first tuple for a given attribute value A and throws away the remaining tuples with the same value. At the same time, this operator can be used to project on a set of attribute B. Analogously, we have a **renaming order-preserving duplicate elimination projection** $\Pi^A_{B':B}$.

Last, we define a new operator that directly implements an unnesting strategy. The operator is called **kappa-join** and is a ternary operator, i.e. it has three argument expressions e_1, e_2, and e_3. It is defined by the following equation:

e_1

cn
1
2
3
4

$e_2(e_1)$

cn	c
1	11
2	22
2	222
3	33

e_3

c'
222
44

Figure 3.2: Kappa-join example

$$e_1 \kappa^{e_2}_{c=c'} e_3 \quad := \quad \sigma_{\mathfrak{A}_{x;exists(e_2 \ltimes_{c=c'} e_3)}}(e_1).$$

The right-hand side of this equation results from our translation of comparison expressions that use $=$ as a comparison predicate (see Sec. 2.2.3). In this equation, c is an attribute defined in the sequence-valued expression e_2, and c' is defined in e_3, i.e. $c \in \mathcal{A}(e_2)$ and $c' \in \mathcal{A}(e_3)$. The attribute x contains the result of the aggregation operator, i.e. true if the result sequence of the semi-join consists at least of one tuple and false otherwise. Accordingly, the selection filters tuples whose x value is false. As for conventional join operators, we denote the producer expressions e_1 and e_3 as *outer producer* and *inner producer*, respectively. The second producer expression e_2 (in the superscript) is called *link producer* because it acts as a link between the outer and inner producer within the join predicate. The outer expression e_1 and the inner expression e_3 are independent expressions, i.e. they do not depend on any of the kappa-join's other arguments. In contrast, the expression e_2 is dependent on e_1, i.e. it refers to free variables that are defined in e_1.

Informally, the result sequence of the operator contains all tuples from the outer producer (e_1) for which there exists at least one tuple in the link producer (e_2), when evaluated with respect to the current tuple of e_1, that satisfies a comparison of attributes of e_2 and attributes of the the inner producer e_3. Consider for an example Fig 3.2. It shows three sequences: (1) The sequence for expression e_1 contains one attribute named cn. (2) The sequence-valued expression e_2 consisting of the attribute c that depends on values from e_1. For example, for cn value 2 it contains two tuples, i.e. 22 and 222. (3) The independent sequence for e_3 contains the values 222 and 44 for attribute c'. The result sequence of the expression $e_1 \kappa^{e_2}_{c=c'} e_3$ on this input contains one tuple with attribute cn that has the value 2.

3.2.1 XPath Context in the Algebra

As already discussed in Section 3.1.2, the notion of context is of utmost importance for XPath. The context is needed to evaluate XPath expressions and, hence, must be handled within our algebra. We briefly repeat the hooks.

Context Representation For the purpose of this chapter, it suffices to think of an XPath context as consisting of three context items, each represented by an attribute: the attributes *cn*, *cp*, and *cs* represent *context node*, *position*, and *size*, respectively.

Each of these attributes has a counterpart in the form of a free variable: *cn*, *cp* and *cs*. Whenever an expression has no producer, but refers to an attribute, or refers to a non-existing attribute, the appropriate free variable is used.

Context Producer and Dependency In the formal description of our optimization techniques, we utilize the notions of *context producer* and *context dependent*. An expression e_1 is called *context producer* for an expression e_2 if (1) it creates new bindings for one or more context item attributes and (2) e_2 exhibits at least one free variable corresponding to one of these attributes. If so, expression e_2 is called *context dependent* on e_1.

3.3 Optimization

We now introduce optimizations for our canonical translation of nested XPath expressions. Given the classification established in Section 3.1 and the (canonical) algebraic translation from Chapter 2, we provide optimization techniques for each of the classes of nested comparison expressions. We formulate them by means of equivalences whose left-hand side matches the result of the canonical translation and whose right-hand side contains the optimized expression.

The section is structured according to the three groups of classes presented in Section 3.1. We start out with independent comparison expressions, continue with semi-independent expressions, and, at the end, examine the group containing only dependent expressions.

3.3.1 Independent Comparison Expressions

Optimizing independent comparison expressions is easy and does not depend on the cardinality. We can simply execute the independent parts separately and materialize the result. This is like constant folding[1] in compiler construction and saves evaluating independent parts more than once. Note that even if the operands of an expression are set-valued, its evaluation always results in a single boolean value. Hence, materialization costs are negligible.

For nested expressions of the form $e_1 \theta e_2$ with $\theta \in \{<, \leq, >, \geq\}$, the minimum/-maximum aggregation within Rules 2.3 and 2.4 (see Sec. 2.2.3) can additionally be pulled outside, as the expression e_2 is independent and single-valued. This way, we can avoid multiple evaluations of the same subexpression.

[1] Due to the global context, this term is a slight misnomer. However, from the evaluation point of view, the global context can be considered as a constant.

3.3.2 Semi-Independent Comparison Expressions

Nested comparison expressions, where one of the operands is independent and the other is dependent, describe a correlation predicate between the outer and inner independent expression. Their evaluation is expensive, as the independent expression is evaluated several times without necessity.

Now, we discuss algebraic unnesting techniques for each of the classes of semi-independent comparison expressions. The corresponding equivalences are summarized in Figure 3.3. Within all equivalences, expression e_2 is context dependent on expression e_1, whereas expression e_3 is independent. We discuss the equivalences in the order of increasing complexity: I[S|A]/D[S|A], IM/DM, I[S|A]/DM, and, finally, IM/D[S|A].

$$\sigma_{\mathcal{T}[e_2]=\mathfrak{A}_{m:max_{cn}}(\mathcal{T}[e_3])}(\mathcal{T}[e_1]) \equiv \Pi_{cn:g'}(Max_{g;m;cn'=}(\Pi_{cn':cn}(\mathcal{T}[e_1])<\mathcal{T}[e_2]>).g) \tag{3.1}$$

if $\mathcal{T}[e_1]<\mathcal{T}[e_2]>=\mathcal{T}[e_3]$ and e_2 is single-valued and dependent on e_3, and e_3 is independent

$$\sigma_{\mathfrak{A}_{x;exists(\mathcal{T}[e_2\theta e_3])}}(\mathcal{T}[e_1]) \equiv \Pi^A_{cn:cn'',\overline{A}}(E)\ltimes_{cn\theta cn'}\Pi^D_{cn':cn}(\mathcal{T}[e_3]) \tag{3.2}$$

with $E=(\nu_A(\Pi_{cn'':cn}(\mathcal{T}[e_1])))<\mathcal{T}[e_2]>$
for $\theta\in\{=,\neq\}$

$$\sigma_{\mathfrak{A}_{x;exists(\mathcal{T}[not(e_2\theta e_3)])}}(\mathcal{T}[e_1]) \equiv \Pi_{cn:cn'',\overline{A}}(\sigma_{c=0}(\nu_A(\mathcal{T}[e_1]))\bowtie\Gamma_{c;A=A';count}E) \tag{3.3}$$

$$\text{with } E = (\Pi_{cn'':cn}((\nu_{A'}(\mathcal{T}[e_1]))<\mathcal{T}[e_2]>)\ltimes_{cn\theta cn'}(\Pi_{cn':cn}(\mathcal{T}[e_3])))$$

$\theta\in\{=,\neq\}$

$$\sigma_{\mathfrak{A}_{x;exists(\mathcal{T}[e_2\theta e_3])}}(\mathcal{T}[e_1]) \equiv \Pi^A_{cn:cn',\overline{A}}(\sigma_{cn\theta x}((\nu_A(\Pi_{cn':cn}(\mathcal{T}[e_1])))<\mathcal{T}[e_2]>)) \tag{3.4}$$

where x results from $\mathfrak{A}_{x;a_{cn}}(\mathcal{T}[e_3])$,
with a $=$ min if $\theta\in\{>,\geq\}$ or a $=$ max if $\theta\in\{<,\leq\}$,

$$\sigma_{\mathfrak{A}_{x;exists(\mathcal{T}[not(e_2\theta e_3)])}}(\mathcal{T}[e_1]) \equiv \Pi_{cn:cn'',\overline{A}}(\sigma_{c=0}(\nu_A(\mathcal{T}[e_1])\bowtie\Gamma_{c;A=A';count}(E)) \tag{3.5}$$

with $E=\sigma_{cn\theta x}((\Pi_{cn'':cn}(\nu_{A'}(\mathcal{T}[e_1])))<\mathcal{T}[e_2]>)$
where x results from $\mathfrak{A}_{x;a_{cn}}(\mathcal{T}[e_3])$,
with a $=$ min if $\theta\in\{>,\geq\}$ or a $=$ max if $\theta\in\{<,\leq\}$,

$$\sigma_{\mathfrak{A}_{x;exists(\mathcal{T}[e_2]=(\sigma_p(\mathcal{T}[e_3])))}}(\mathcal{T}[e_1]) \equiv \sigma_{g>0}(E) \tag{3.6}$$

$$\text{with } E = \Pi_{cn:cn',\overline{A}}(\Gamma_{g;=A;count\circ\sigma_p}((\nu_A(\Pi_{cn':cn}(\mathcal{T}[e_1])))<\mathcal{T}[e_2]>))$$

and $(\mathcal{T}[e_1]<\mathcal{T}[e_2]>)=\mathcal{T}[e_3]$

$$\sigma_{\mathfrak{A}_{x;exists(\mathcal{T}[e_2\theta e_3])}}(\mathcal{T}[e_1]) \equiv \Pi^A_{cn:cn',\overline{A}}(\sigma_{cn\theta x}((\nu_A(\Pi_{cn':cn}(\mathcal{T}[e_1])))<\mathcal{T}[e_2]>)) \tag{3.7}$$

where x results from e_3 (single-valued), and $\theta\in\{=,\neq,>,\geq,<,\leq\}$

$$\sigma_{\mathfrak{A}_{x;f(\mathcal{T}[e_2])\theta\mathcal{T}[e_3]}}(\mathcal{T}[e_1]) \equiv \Pi_{cn:cn'',\overline{A}}(E)\ltimes_{g\theta cn'}(\Pi^D_{cn':cn}(\mathcal{T}[e_3])) \tag{3.8}$$

with $E=\Gamma_{g;=A;f}(\nu_A(\Pi_{cn'':cn}(\mathcal{T}[e_1])))<\mathcal{T}[e_2]>$
if $\theta\in\{=,\neq,<,\leq,>,\geq\}$

Figure 3.3: Equivalences for semi-independent comparison expressions

Classes I[S|A]/D[S|A]

Classes I[S|A]/D[S|A] contain two single-valued expressions where both operands are dependent. In a first optimization step, constant folding of the independent part could be applied for both classes (I[S|A]/D[S|A]). However, sometimes even more efficient techniques are possible. An interesting query, which allows for advanced optimization techniques, is the following one:

```
/university/exam[not(grade < /university/exam/grade)]
    /belongsto/@lecture.
```
Q2

During normalization, the query (selecting the worst exams) is rewritten to[2]

```
/university/exam[grade = max(/university/exam/grade)]
       /belongsto/@lecture.
```
Q2

and falls into the class IA/DS. Note that the rewrite is only correct because the grades will be the same on both sides of the comparison.

The canonical translation of this query is

$$\begin{aligned} e &= \mathcal{T}[\text{belongsto/@lecture}](e_1)) \\ e_1 &= \sigma_{\mathcal{T}[\text{grade}]=\mathfrak{A}_{m;max_{cn}(e_2)}}(\mathcal{T}[\text{/university/exam}]) \\ e_2 &= \mathcal{T}[\text{/university/exam/grade}] \end{aligned}$$

Note that from now on, we abbreviate the algebra expression for location steps and paths and simply denote them with our translation function $\mathcal{T}$ or our helper function Φ, respectively. Applying constant folding, the independent maximum aggregation is evaluated only once. However, since the outer and inner independent expression share a common path, evaluating the common expression more than once can be avoided. This optimization described in Equivalence 3.1 uses the max operator.

This operator uses the scan needed for the evaluation of the inner expression to simultaneously evaluate the outer expression. The result of the operator has two attributes: One containing the maximum value m for a group of nodes and one containing the resulting nodes that are equal to the calculated maximum. Note that the resulting sequence is empty if the input sequence is empty. The application of Equivalence 3.1 results in

$$\begin{aligned} e' &= \mathcal{T}[\text{belongsto/@lecture}](e'_1)) \\ e'_1 &= \Pi_{cn:g}(Max_{g;m;cn'=}(e'_2)).g \\ e'_2 &= (\Pi_{cn':cn}(\mathcal{T}[\text{/university/exam}])) < \mathcal{T}[\text{grade}] > \end{aligned}$$

[2]Remember that there is no maximum function in XPath 1.0. We fall back on our own function here.

Sharing the scan with the help of the max operator halves the execution time for this query. The evaluation of this query in Sec. 3.4 proofs our claim and Section A.1 in the appendix presents a proof of correctness. Of course, the minimum can be treated analogously.

Class IM/DM

We now consider comparison expressions of type IM/DM, i.e. those where both operands are set-valued. In this case, the XPath semantics requires a quantified evaluation. Using our algebraic approach, we are able to revert to the well-established idea of using semi-joins (e.g. for unnesting XQuery [82]). However, the application of known equivalences is not possible. There are two main reasons for this. First, all known equivalences suffer from at least one of two problems: (a) the outer operator of the nested algebraic expression is a map and not (the needed) selection, or (b) they have conditions not fulfilled here (e.g. at least one single-valued argument). Second, context handling has not been investigated.

In the following, we look at different algebraic evaluation strategies for different query patterns exhibiting correlation predicates. Therefore, consider the following query patterns that are made up of different comparison operators ($=, \neq$, and $\theta \in \{<, \leq, >, \geq\}$) and negation, where E denotes any path expression, DM an set-valued expression dependent on E, and IM an independent set-valued expression.

1. $E[DM = IM]$
2. $E[DM \neq IM]$
3. $E[\text{not}(DM = IM)]$
4. $E[\text{not}(DM \neq IM)]$
5. $E[DM\theta IM]$
6. $E[\text{not}(DM\theta IM)]$

For introducing the idea, we start with an elaborate discussion of optimizations for query pattern 1. Existential quantification (i.e. query pattern 1) is one of the most frequently used patterns in XPath. Hence, we not only present the application of known unnesting technique using semi-joins. We also extend the algebra with our new kappa-join operator for particularly boosting the performance of XPath queries with query pattern 1. [3] At the end, we briefly discuss the remaining query patterns and their associated equivalences.

Query Pattern 1 We begin with a shortened version of the query from the introduction, which is an instance of query pattern 1 (selecting all students having exams better than B).

```
//student[examination/@id =
    //exam[grade<'B']/@id]/name
```
Q3

[3] However, the kappa-join can also be used for evaluating other query languages, e.g. XQuery.

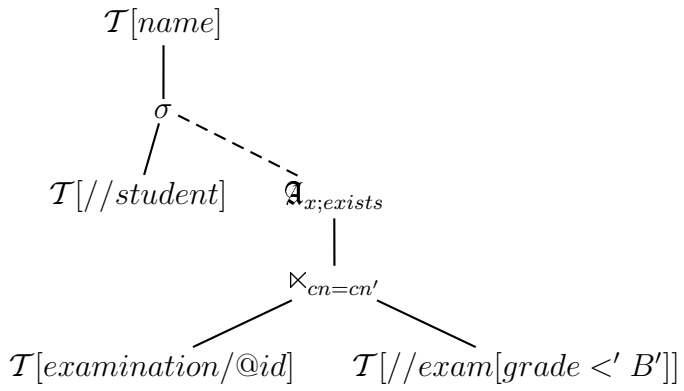


Figure 3.4: Translation sketch for Q3

The canonical evaluation of this query provides an aggregation with a semi-join as producer.

$$
\begin{aligned}
e &= \Psi[\text{name}](\sigma_{e_1}(\mathcal{T}[\text{//student}])) \\
e_1 &= \mathfrak{A}_{x;exists}(\mathcal{T}[\text{examination}] < \mathcal{T}[\text{@id}] > \ltimes_{cn=cn'}(e_2)) \\
e_2 &= \Pi_{cn':cn}(\Psi[\text{@id}](\sigma_{\mathcal{T}[\text{grade}]<'\text{B}'}(\mathcal{T}[\text{//exam}])))
\end{aligned}
$$

For convenience, Fig. 3.4 shows the algebra expression as a tree.

The problems that occur have already been explained in the introduction to this chapter. The expressions within predicates are evaluated for every result of the corresponding location step. That is, the subscript of the selection (in Fig. 3.4 denoted by a dashed line) is evaluated for every student resulting from the outer expression. Thus, evaluating the above query in a nested correlated fashion is not very efficient. The idea of the optimization that is shown in Equivalence 3.2 is to pull the semi-join into the main evaluation thread.

The resulting optimized translation is as follows:

$$
\begin{aligned}
e' &= \Psi[\text{name}](\Pi^{A}_{cn:cn'}(e'_1)) \\
e'_1 &= \nu_A(\Pi_{cn':cn,\overline{A}}(\mathcal{T}[\text{//student}])) < e'_2 > \ltimes_{cn=cn'}(e'_3)) \\
e'_2 &= \Pi_{cn':cn}(\Psi[\text{examination/@id}](\Box_{cn':cn}))) \\
e'_3 &= \Psi[\text{@id}](\sigma_{\mathcal{T}[\text{grade}]<'\text{B}'}(\mathcal{T}[\text{//exam}]))
\end{aligned}
$$

The dependent location path `examination/@id` is connected to the outer expression using a d-join [82]. As the dependent expression could produce duplicate students, the ν operator is needed to identify the tuples resulting from the outer expressions. At the end, we remove duplicates of the same ν-value for A

with our order-preserving duplicate elimination ($\Pi^{A}_{cn:cn',\overline{A}}$). Moreover, this projection reestablishes the right context attribute cn from cn' and discards the ν attribute A. Fig. 3.5 sketches the optimized algebra expression graphically.

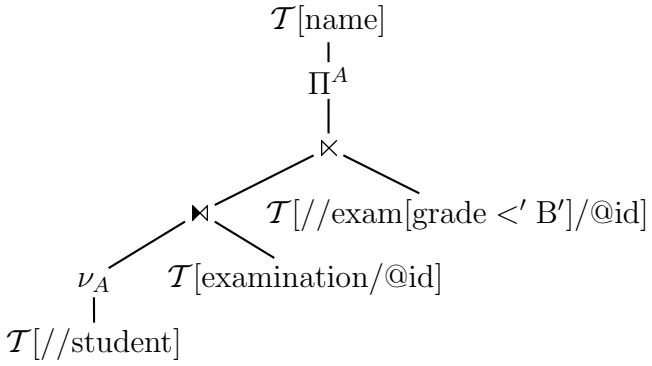


Figure 3.5: Unnesting strategy for Q3

Resulting from that, we avoid the repeated evaluation of the inner independent expression. If this equivalence is applied to the introductory query, we can additionally choose the optimal execution order for the semi-joins, which gives us an even higher optimization potential.

Clearly, the main advantage of this approach is that the independent expression is evaluated only once. In addition, if the implementation of the semi-join uses a custom data structure (e.g. a hash-table) to improve performance, this data structure has to be initialized only once, compared to one initialization per student in the naïve correlated evaluation. However, unnesting comes at a price: The outer expression produces duplicates which have to be eliminated. Below, we show how we can avoid them using our novel kappa-join. Our evaluation in Sec. 3.4 confirms this claim.

To avoid the above-mentioned generation of duplicates, but nevertheless gain performance by avoiding unneeded evaluations of the independent expression, we use the kappa-join operator. It combines the advantages of the evaluation strategies from Equivalence 3.2 and the canonical translation into one operator and capitalizes on efficient implementation techniques.

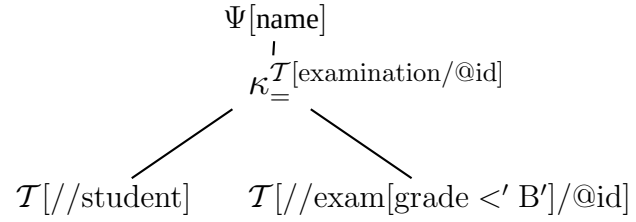


Figure 3.6: Unnesting strategy for Q3 with kappa-join

Fig. 3.6 contains the resulting algebra expression for Q3. Here, the location path `//student` is mapped to the outer producer of the kappa-join. The inner location path `examination/@id` is the (dependent) link producer, and the independent expression `//exam[grade<'B']/@id` is mapped to the inner producer.

```
OPEN
1  while T ← INNERPRODUCER.NEXT
2        do HASHTABLE.INSERT(T)

NEXT
 1  while T_1 ← OUTERPRODUCER.NEXT
 2        do
 3           LINKPRODUCER.OPEN(T_1)
 4           while T_2 ← LINKPRODUCER.NEXT
 5                 do
 6                    if HASHTABLE.LOOKUP(T_2)
 7                      then
 8                           LINKPRODUCER.CLOSE
 9                           return T_1
10
11           LINKPRODUCER.CLOSE
12  return nil
```

Figure 3.7: Pseudocode for the kappa-join

The secret of the kappa-join lies in its simple, yet efficient implementation. It improves performance beyond that of the operator combination in its logical definition. Fig. 3.7 shows the pseudocode for the implementation of the kappa-join as an iterator.

In its `open` method, the kappa-join builds a data structure, e.g. a hash-table, containing the attributes from the inner producer that are part of the join predicate. In its `next` method, the kappa-join initializes the link producer for every tuple T_1 from its outer producer. Like a semi-join, it then probes the hash-table with tuples T_2 from the link producer until a matching one is found, and returns the outer tuple as soon as it finds a match. The kappa-join does not always enumerate all tuples from the dependent link producer, while building the hash-table only once. Hence, the worst-case complexity is $O(|e_1| \times |e_2| + |e_3|)$, assuming constant hash-table insert and lookup, respectively. However, the average complexity depends on the distribution of the data and is usually much better. Compared to the canonical translation, the plan in Fig. 3.6 using the kappa-join has three main advantages: (1) It avoids to enumerate all tuples from the link producer because it immediately

returns a result if one match is found (see Line 9). (2) It does not produce duplicates of tuples from the outer producer because the result contains at most one tuple from $\mathcal{T}[//\mathrm{student}]$, and (3) consequently saves the cost of a final duplicate elimination. These effects combine to yield the speedup that can be achieved (see Sec. 3.4).

We now briefly discuss the remaining query patterns.

Query Pattern 2 So far, we have discussed the optimization of XPath queries that match query pattern 1. This query pattern uses an equality comparison operator and resembles the semantics of an existential quantification. However, XPath queries can also exhibit a non-equality comparison operator. In this case, shown in query pattern number 2, the semantics is that of a negated existential quantification. As for existential quantification, we can also evaluate such queries with a semi-join, but using a negated comparison. This procedure is also captured in Equivalence 3.2. Moreover, we can also use the kappa-join operator.

Query Pattern 3 If such a comparison expression is encapsulated in a `not` function-call, the semantic resembles that of a universal quantification. In the first case (see query pattern 3), the semantics requires that there must not exist any match that is equal, i.e. all items are different. Secondly, if the comparison contains a non-equals sign, the semantics is inverted, i.e. all items are equal. Both universally quantified query patterns can be unnested using Equivalence 3.3. In this equivalence, the trick is to count the number of matches (cf. [26]) that satisfy $e_2 = e_3$ (resp. $e_2 \neq e_3$) for every tuple resulting from e_1 and store the result in the attribute c. To this end, we use the binary grouping operator with e_1 as outer and the comparison between e_2 and e_3 as inner expression. Note that for computing e_2, we use the same trick as shown previously, i.e. using a d-join for computing e_2 that is dependent on e_1. Thereby, all tuples from e_1 are uniquely identified using the attribute A' that is added by a ν operator. To accomplish the match between tuples from the left- and right-hand side of the binary grouping operator, we also (deterministically) mark the tuples from e_1 on the left-hand side and store their ν-value in the attribute A. At the end, only those tuples from e_1 qualify for the result whose value for c is equal to zero, i.e. either all tuples are different ($=$) or equal ($\neq$), respectively.

Query Pattern 5 Query pattern 5 again resembles an existential quantification but with an arbitrary comparison operator $\theta \in \{<, \leq, >, \geq\}$. Unnesting such queries is accomplished using Equivalence 3.4. To this end, we either compute the maximum (for $\theta \in \{<, \leq\}$) or minimum (for $\theta \in \{>, \geq\}$) value for items resulting from the independent expression (e_3). Then, we can select only those tuples from e_1 whose e_2 value satisfies the according comparison with this maximum or minimum value, respectively. Again, we use a combination of the ν, d-join, and Π^A operators to evaluate e_2 depending on e_1. Note that in a first step, the maximum/minimum

aggregation remains subscript of the select operator. However, in a second optimization step the evaluation of the independent aggregation can be pushed outside and evaluated on its own.

Query Pattern 6 The last query pattern 6 describes a comparison expression encapsulated in a not function-call that uses an arbitrary comparison operator $\theta \in \{<, \leq, >, \geq\}$. To unnest queries that comply with this query pattern, we combine the previously presented Equivalences 3.3 and 3.4. The result is captured in Equivalence 3.5. In this equivalence, we compute the maximum/minium value of e_3 independently and count the number of matches for every tuple from e_1 with the help of a binary grouping operator. The count is stored in the attribute c, and tuples from e_1 whose c-count is equal to zero contribute to the result.

With Equivalence 3.6, we introduce an optimization that uses a strategy similar to the max operator. This equivalence avoids duplicate evaluation of common expressions if the inner and outer expression share the same path, i.e. $\mathcal{T}[e_1] < \mathcal{T}[e_2] >= \mathcal{T}[e_3]$. In fact, Equivalence 3.1 is a special case of Equivalence 3.6.

Class I[S|A]/DM and IM/D[S|A]

The last class within the group of semi-independent comparison expressions contains one single- and one set-valued expression. Optimizing them combines the evaluation strategies of the previous equivalences.

If the independent expression is single-valued (class I[S|A]/DM), Equiv. 3.7 can be applied, which is similar to 3.4. It pushes the evaluation of the independent part outside and does the comparison of the dependent part using a selection.

For the second case, where the independent part is set-valued (e_3) and the dependent part comes from an aggregation (class IM/DA), Equiv. 3.8 can be applied. It uses a grouping operator to compute the aggregation and does the comparison using a semi-join. As the input for the grouping operator is already sorted by the grouping attribute, an efficient implementation for the grouping operator can be used. For class IM/DS, Equivalence 3.2 can be applied. Generally, the equivalences from the previous sections can be used for all cases.

3.3.3 Dependent Comparison Expressions

Last but not least we discuss optimizations for comparison expressions that are made up only of dependent operands. The last two sections considered optimizations for comparison expressions where at least one of the operands is independent (i.e. that the independent part is evaluated redundantly). If both operands are dependent, unnesting techniques in the narrower sense cannot be applied. In this section, we present a few algebraic optimization techniques for optimizing such queries. Optimizing such queries requires factorization or schema knowledge. Although schema-based optimization is beyond the scope of this thesis (see outline in

Chapter 7), we present examples that demonstrate the optimization potential. For a more extensive treatment of optimizations for queries with dependent comparison expressions, we refer to [107].

The equivalences we present in this thesis are shown in Fig. 3.8.

$$\sigma_{f(\sigma_{n=c_1}(\chi_{n:f'(cn)}(\mathcal{T}[\pi])))=f(\sigma_{n'=c_2}(\chi_{n':f'(cn)}(\mathcal{T}[\pi])))}(e) \equiv e < \Pi\overline{}_{y,y'}(\sigma_{y'=y}(\chi_{y':2}(\Box) \quad (3.9)$$
$$\ltimes^{y:2}_{true}(\Gamma_{y;=x;count}(\Gamma_{x;=n;f}$$
$$(\sigma_{n=c_1 \vee n=c_2}(\chi_{n:f'(cn)}(\mathcal{T}[\pi])))))) >$$
and $f \in \{\text{sum}, \text{count}\}$,
and π is context dependent on e

$$\sigma_E(\mathcal{T}[(\pi)[e_2]]) \equiv \Pi^D_{\text{string-value}(cn)}(\mathcal{T}[(\pi)[e_2]] < \mathcal{T}[e_1] >) \quad (3.10)$$
with $E = \text{not}(\mathcal{T}[e_1] = \mathcal{T}[((pre :: *)|(anc :: *))[e_2]/e_1])$
if e_1 is single-valued,
e_1 is context dependent on $(\pi)[e_2]$ and,
e_2 does not contain a call to position or last

Figure 3.8: Equivalences for dependent comparison expressions

In the following three subsections, we discuss comparison expressions of type DA/DA, DS/DM, and DM/DM. We present an example query for each type and demonstrate their effectiveness in our evaluation section.

Class DA/DA

The first class contains comparison expressions where both operands are single-valued. Obviously, it is very challenging to improve the performance of nested single-valued expressions that do not result from aggregation. On the other hand, expressions that include aggregation (DA/DA) are more amenable to optimization since they can be more expensive to evaluate. As an example, consider the query selecting all students that take as many exams as they have attended lectures:

```
//student[count(descendant::examination)
    = count(descendant::attends)]/name
```
Q4

The canonical translation is as follows:

$$\begin{aligned} e &= \Psi[\text{name}](\sigma_{e_1=e_2}(\mathcal{T}[\text{//student}])) \\ e_1 &= \mathfrak{A}_{x;count}(\mathcal{T}[\text{descendant::examination}]) \\ e_2 &= \mathfrak{A}_{y;count}(\mathcal{T}[\text{descendant::attends}]) \end{aligned}$$

The last two sections considered optimizations for comparison expressions where at least one of the operands is independent (i.e. that the independent part is evaluated

redundantly). As in the above example, all expressions are dependent, unnesting techniques in the narrower sense cannot be applied. However, all descendant nodes of students are visited twice, once for each dependent expression. We can do better if we find a common superset with the following property: the overall evaluation costs – the costs to compute the superset plus the costs to complete the computations for both expressions – are cheaper than the evaluation costs of evaluating both expressions independently.

For the above example, the two optimized expressions can also be expressed using XPath filter expressions. For instance,

- `(descendant::*)[name(.)='examination']` and
- `(descendant::*)[name(.)='attends']`.

Again, the idea is to use a common scan to evaluate the superset and to evaluate the name tests afterwards. Applying Equivalence 3.9 results in the following optimized translation:

$$
\begin{aligned}
e' &= \Psi[\text{name}](\mathcal{T}[\text{//student}] < e'_1 >) \\
e'_1 &= \Pi_{\overline{y,y'}}(\sigma_{y'=y}(\chi_{y':2}(\Box) \mathbin{⟗}^{y:2}_{true}(e'_2)) \\
e'_2 &= \Gamma_{y;=x;count}(\Gamma_{x;=n;count}(e'_3))) \\
e'_3 &= (\sigma_{n=(\text{'examination'})\vee n=(\text{'attends'})}(e'_4))) \\
e'_4 &= (\chi_{n:\text{name(cn)}}(\mathcal{T}[\text{descendant::*}]))
\end{aligned}
$$

The inner expression e'_1 is connected using a d-join. It starts with evaluating all descendant nodes of the context nodes and subsequently adds the tagname as attribute n to each tuple. The following selection filters the nodes according to the required names. The first grouping operator creates groups for each of the names and adds an attribute containing the group sizes. The second grouping operator creates groups for each of the group sizes and counts its members. The result of the second grouping qualifies if it consists of one group and $y = 2$. The outer join is needed to handle empty groups.

Class DS/DM

The next class contains one single-valued and one set-valued expression. A common technique (cf. Sec. 3.14 in [38]) to eliminate duplicates using XPath is shown in the following example query:

```
//lecture[not(title =
    preceding-sibling::lecture/title)]/title
```
Q5

Within our document, we have a lot of lectures (all on the same level), some having the same title. Reasoning about this fact requires schema information. We

want to get a list of all lecture titles without duplicates. Therefore, our XPath query selects the lectures with a title that does not follow in the remainder of the document. The canonical translation is as follows :

$$\begin{aligned} e &= \Psi[\text{title}](\sigma_{not(\mathfrak{A}_{x;exists e_1 \ltimes_{cn=cn'} e_2})}(\mathcal{T}[\text{//lecture}])) \\ e_1 &= \mathcal{T}[\text{title}] \\ e_2 &= \Pi_{cn':cn}(\mathcal{T}[\text{preceding-sibling::lecture}] < \mathcal{T}[\text{title}] >) \end{aligned}$$

Obviously, the costs to evaluate all preceding-siblings for every lecture are huge. However, using information about the semantics of the above query, we can do better and apply Equivalence 3.10. This equivalences relies on a our duplicate elimination which exactly resembles the required semantics. It keeps the first tuple with a given string-value (string-value(cn)) and throws away the remaining ones.

$$e' = \Pi^D_{\text{string-value(cn)}}(\mathcal{T}[\text{//lecture/title}])$$

In fact, Equivalence 3.10 is more general. However, with some knowledge about the schema, in which all lectures appear on the same level, we can apply a rewrite and use the `preceding-sibling` axis instead of the union of `preceding` and `ancestor`.

Class DM/DM

This class is the most difficult to optimize, as we have two dependent set-valued expressions (i.e. it may be quite challenging to factorize common subexpressions). Most of the (reasonable) optimizations that are possible involve the semantics of the query, that is, knowledge of the schema. As schema-based optimizations are beyond the scope of this thesis, we just give an example here and some evaluations later. Consider the following query in which we are looking for all research assistants who share a research topic with another assistant:

```
//employee[assistant/topic =
  (following-sibling::employee |
    preceding-sibling::employee)
      /assistant/topic]/name
```

Q6

As all assistants can be found on the same level in the document, it is sufficient to scan through them twice and join them on their topics (taking care not to join a tuple with itself). After that, we just need to eliminate duplicates.

$$\begin{aligned} e &= \Psi[\text{name}]\Pi^D_e((e_1 \bowtie_{e \neq e' \wedge t = t'} e_2)) \\ e_1 &= \Pi_{t:cn}(\Psi[\text{topic}](\Psi[\text{assistant}](\Pi_{e:cn}(\mathcal{T}[\text{//employee}])))) \\ e_2 &= \Pi_{t':cn}(\Psi[\text{topic}](\Psi[\text{assistant}](\Pi_{e':cn}(\mathcal{T}[\text{//employee}])))) \end{aligned} \tag{3.11}$$

3.4 Evaluation

To show the effectiveness of our approach, we measured different XPath evaluation engines against our canonical and optimized translations. We chose the freely available engines

- Xalan C++ 1.8.0 using Xerces C++ 2.6.0,
- Berkley DB XML 2.0.9 (DBXML) using libpathan 1.99 as XPath engine,
- MonetDB 4.8.0 using MonetDB-XQuery-0.8.0 [10],
- the evaluator provided by the XMLTaskForce [72] (XTF), and
- Natix for the canonical and optimized approach.

3.4.1 Environment

The environment used to perform the experiments consisted of a PC with an Intel Pentium 4 CPU at 2.40GHz and 1 GB of RAM, running SuSE Linux 2.6.8. The Natix C++ library was compiled with gcc 3.3.5 using optimization level 2.

For Xalan and XTF, we measured the net time to execute the query. The time used to parse the document and generate the main memory representation is subtracted from the elapsed evaluation time.

For the evaluation of MonetDB, Berkley DB XML, and Natix, we imported the documents into the database. The Natix instance was created using a page size of 8kB and a buffer size of 1000 pages. The cache for Berkley DB XML was configured with the same size. MonetDB was used out of the box. The queries were executed with an empty buffer pool on a cold instance and without any indexes. The query execution engine of the Natix system implements the logical algebra defined in the last and this chapter.

3.4.2 Documents

We generated two different sets of documents. The first is used for our example queries Q1-Q6 that are based on the university schema. These documents were generated using the ToXgene data generator [6] [4]. We generated 6 documents: The smallest document contains 50 employees, 100 students, 10 lectures, and 30 exams. With each document we quadrupled these numbers, so that the biggest document contains 51200 employees, 102400 students, 10240 lectures, and 30720 exams. Overall, this lead to moderate document sizes between 59kB and 43MB.

The second set (called synthetic data set) is used for the comparison of the kappa-join with our unnesting strategy according to Equivalence 3.2. We generated seven documents structured according to the following template:

[4] The DTD is shown in Appendix B

```
<?xml version='1.0'?>
<gen>
<e1 id='0'> <e2 id='0'/> ... i-e2 nodes <e2 id='i'/> </e1>
                         ...
<e1 id='0'> <e2 id='0'/> ... i-e2 nodes <e2 id='i'/> </e1>
<e3 id='RandomNumber'/>
</gen>
```

Each of the documents contains 1000 e_1 nodes and 1000 e_3 nodes. For each document, we varied the number of e_2 nodes (below an e_1 node) between 10 and 500 nodes. This led to documents with sizes between 252kB and 13MB.

3.4.3 Queries

We executed performance measurements for all example queries (Q1-Q6) presented throughout this chapter. All queries were executed as printed on all the evaluators listed above.

Additionally, we executed performance measurements that compare the unnesting strategy according to Equivalence 3.2 with our kappa-join operator. Therefore, we executed the following query on our synthetic data set:

```
/gen/e1[e2/@id = /gen/e3/@id]
```
Q7

Query Evaluation Plans for Natix For Natix, we generated several different NVM access plans for each of the queries. One plan always implements the canonical translation. Further, we generated the following alternative plans:

Q1: **unnested** Access plan which uses Equivalence 3.2 twice. The exams and lectures are compared first, before completing the result with the students.

Q2: **unnested** Access plan using constant folding.

max Access plan which uses the max operator as described in Section 3.3.2, i.e. Equivalence 3.1.

Q3: **unnested** Access plan with Equivalence 3.2 applied (see Fig. 3.5).

kappa Access plan using the kappa-join (see Fig. 3.6).

Q4: **optimized** Access plan which uses Equivalence 3.9.

Q5: **dupelim** Access plan which uses Equivalence 3.10.

Q6: **join** Access plan according to Expression 3.11.

Q7: **unnested** Access plan according to Equivalence 3.2.

kappa Access plan exploiting the kappa-join.

Evaluator	Documents 1	2	3	4	5	6
Xalan	**0.01**	**0.03**	0.44	7.71	123.92	2008
DBXML	0.03	0.14	1.32	4.09	59.06	863.56
MonetDB	0.28	0.35	1.36	22.70	DNF	DNF
XTF	0.49	5.65	111.40	DNF	DNF	DNF
Natix						
• canonical	0.16	0.81	11.12	176.59	12330	DNF
• unnested	0.12	0.13	0.17	0.31	0.93	3.37
• max	0.12	0.13	**0.14**	**0.22**	**0.51**	**1.69**

Figure 3.9: Results (in sec.) for Q2

3.4.4 Results and Interpretation

Figures 3.9, 3.10, and 3.11 present the results of our performance measurements. All tables show the elapsed time in seconds. We executed all queries that finished within six hours. Those that did not finish in this time limit are marked by DNF. For MonetDB, the evaluation of some queries ran out of memory on bigger documents (denoted by OOM). Moreover, for some queries the evaluator provided by the XMLTaskForce and MonetDB crashed for unknown reasons. These cases are denoted by n/a. The best execution time for each column is printed in bold face. For almost all queries, our optimized approaches perform and scale best.

IA/DS Fig. 3.9 presents the performance evaluation for Q2. From the figure we can see that our unnested approach and the strategy using the max operator outperform the fastest of the other evaluators (i.e. Xalan) by three orders of magnitude on the biggest document. On this document, almost all other approaches — including our canonical approach — did not finish within six hours. Moreover, the evaluation demonstrates that the max operator gives us an additional speedup of almost 50% when compared to the ”simple” unnested approach.

IM/DM For demonstrating the performance of our unnesting approach for queries whose comparison operands are both set-valued, we performed experiments with Q1 and Q3 on the university schema and Q7 on our synthetic schema For all queries on all documents, our unnesting approach performs and scales best.

For Q3, the execution times of the unnesting approach using Equivalence 3.2 behave similar to those of the kappa-join. This is because all students took very few exams, i.e. only between one and three. For this reason, we compared those two strategies on the synthetic data set. Subfigure 3.10(c) contains a comparison between the two strategies. The execution times of the unnesting strategy without kappa-join grow linearly with the number of `e2` nodes per `e1` node. This is because it has to enumerate all `e2` nodes and finally perform a duplicate elimination on the

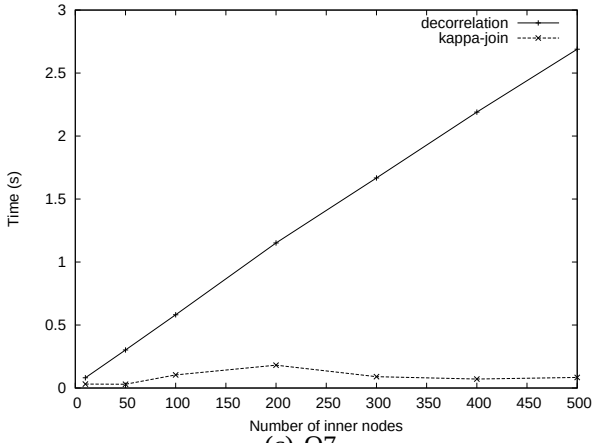
appropriate `e1` nodes. The execution times of the kappa-join operator are almost constant. This is because the kappa-join does not need to enumerate all `e2` nodes and saves the cost of a final duplicate elimination.

	Documents					
Evaluator	1	2	3	4	5	6
Xalan	1.07	45.40	2805	DNF	DNF	DNF
DBXML	0.98	40.83	2634	DNF	DNF	DNF
MonetDB	0.53	4.43	OOM	OOM	OOM	OOM
XTF	0.22	4.54	67.00	1376	DNF	DNF
Natix						
• canonical	2.83	119.40	7154	DNF	DNF	DNF
• unnested	**0.12**	**0.14**	**0.20**	**0.44**	**1.53**	**5.59**

(a) Q1

	Documents					
Evaluator	1	2	3	4	5	6
Xalan	0.30	0.38	6.17	95.6	1552	DNF
DBXML	0.07	0.66	11.6	336	DNF	DNF
MonetDB	0.31	0.38	2.05	36.1	OOM	OOM
XTF	0.40	4.72	82.8	DNF	DNF	DNF
Natix						
• canonical	0.25	2.62	38.2	583	9637	DNF
• unnested	**0.02**	**0.03**	**0.06**	**0.19**	**0.75**	2.99
• kappa	**0.02**	**0.03**	**0.06**	**0.19**	**0.75**	**2.88**

(b) Q3



(c) Q7

Figure 3.10: Results (in sec.) for Q1, Q3, and Q7

DA/DA, DS/DM, and DM/DM Finally, Fig. 3.11 presents the evaluation of our unnesting strategies targeting dependent comparison expressions.

Again, our approaches dominate almost all other evaluators. In some cases the main memory-based interpreter Xalan is faster than our unnested approach. Espe-

cially, on very small documents and when faced with an aggregation function as in Q4, Xalan is almost twice as fast as our approach.

However, our unnesting techniques for queries Q5 and Q6 again outperform all other evaluators. Especially our canonical approach cannot keep up with the unnested approaches and is outperformed by several orders of magnitude. Moreover, for Query Q6, for example, our unnesting technique outperforms even the fastest of the other evaluators (Xalan) by four orders of magnitude. These numbers show the potential that is possible to exploit with the help of schema-based optimization.

	Documents					
Evaluator	1	2	3	4	5	6
Xalan	**0**	**0**	**0.04**	**0.10**	**0.42**	**1.65**
DBXML	0.04	0.08	0.32	1.34	5.66	23.26
MonetDB	0.21	0.21	0.27	0.58	1.37	5.93
XTF	0.21	2.32	38.17	593.83	9489	DNF
Natix						
• canonical	0.12	0.13	0.17	0.33	1.00	3.76
• optimized	0.12	0.13	0.16	0.30	0.87	3.13

(a) Q4

	Documents					
Evaluator	1	2	3	4	5	6
Xalan	**0**	**0**	**0.06**	0.56	12.22	346.08
DBXML	0.03	0.10	0.64	8.17	232.54	DNF
MonetDB	0.26	0.27	0.40	1.80	23.87	n/a
XTF	n/a	n/a	n/a	n/a	n/a	n/a
Natix						
• canonical	0.12	0.14	0.31	2.97	43.45	851.44
• dupelim	0.12	0.13	0.15	**0.26**	**0.72**	**2.39**

(b) Q5

	Documents					
Evaluator	1	2	3	4	5	6
Xalan	**0.01**	**0.08**	1.49	29.90	795.9	33552
DBXML	0.15	0.62	9.65	451.06	DNF	DNF
MonetDB	0.28	0.47	2.62	35.60	DNF	DNF
XTF	n/a	n/a	n/a	n/a	n/a	n/a
Natix						
• canonical	0.16	0.62	8.18	126.73	2522	DNF
• join	0.21	0.22	**0.26**	**0.46**	**1.30**	**4.77**

(c) Q6

Figure 3.11: Results (in sec.) for Q4, Q5, and Q6

3.5 Related Work

We have already discussed existing evaluation techniques and evaluators for XPath in the related work of the previous chapter. However, all of the mentioned approaches have their drawbacks when faced with nested/correlated XPath queries.

Clearly, our main competitor is the approach by Gottlob et al. [46, 47] They presented a complete and efficient method for evaluating XPath. Although their evaluation algorithm has a polynomial worst-case run time, they have some shortcomings in optimizing the evaluation process further. For example, for independent nested expressions they materialize intermediate results for all different contexts (which is not necessary). We, however, can avoid this by unnesting the expression.

This brings us to the initial work on unnesting nested queries. Apart from introducing a classification for nested queries, Kim [70] was the first to rephrase nested SQL queries to contain joins or grouping. However, the validity of these rewrites depends on important restrictions. They mainly concern empty results for the inner query block, NULL values, and duplicate handling. Subsequent research found more unnesting techniques for SQL [32, 36, 43, 44, 69, 71, 99], OQL [28, 29, 39, 101, 102], and XQuery [81, 82, 93].

All these works have in common that they unnest nested queries where one query block contains another query block. In SQL and OQL, a query block corresponds to a SFW expression, whereas in XQuery, it is a FLWOR expression. Since XPath does not exhibit query blocks, the applicability of these approaches is limited.

Strategies for the evaluation of nested queries are discussed in [50]. However, currently the full potential for optimization is only available when queries are unnested. First results to lift this limitation are presented in [56]. Additional work for avoiding unnecessary navigation in XQuery subqueries that is similar to the max operator has been done in [33].

Contrary to the situation in unnesting SQL, OQL, or XQuery, there are cases in XPath that are difficult to unnest. These are the cases in which both expressions in a comparison expression are dependent. In these cases, we refer to [107] for optimizations. In contrast to [107] our classification defines classes for expressions of a single XPath query. They discuss the optimization of multiple correlated path expressions within one query by utilizing the connections between different path expressions. However, they have a similar definition of our term dependent, which they call correlated.

3.6 Conclusion

The translation of XPath queries into algebraic expressions provides a solid foundation for efficient evaluation. However, when faced with nested expressions, a simple canonical translation, as shown in the previous chapter, still suffers from

inadequate performance similar to other evaluation approaches, such as interpretation. In this chapter, we have showed how the algebraic approach can be leveraged to significantly improve evaluation time for nested XPath queries.

We have classified XPath expressions on properties relevant for unnesting predicates. For each of the resulting classes, we have developed equivalences for unnesting the algebraic expressions that have been obtained by the canonical translation of XPath into our algebra. In an experimental study, we have demonstrated that access plans optimized in this way are superior to other state-of-the-art XPath evaluators. We have gained several orders of magnitude in terms of performance by unnesting nested expressions.

However, our unnesting techniques do not apply when concerned with nested queries occurring in a disjunction. This also holds for all other unnesting techniques in the context of SQL, OQL, and XQuery. Hence, in the next chapter, we extend our approach to also support nested XPath queries with disjunctions.

Chapter 4

Disjunctive Unnesting for XPath

Almost every XML query language features a construct that allows to express an existentially quantified comparison of two node-set valued subexpressions in a concise manner. Unfortunately, current XML query processors lack efficiency and scalability when facing such constructs. Query Q1 from the beginning of the last chapter is an example for such a query. The corresponding semantics resembles that of nested and correlated subqueries, which are notoriously difficult to evaluate efficiently. To this end, we presented solutions in form of algebraic equivalences for efficiently evaluating such queries.

However, Q3 is ”simple” because the correlation predicate occurs on its own. What if correlation predicates become more complex? For an example, consider the following XPath query:

```
//student[examination/@id = //exam[grade<'B']/@id or
        semester > 5]/name
```

Q8

In this query, which could be used for searching students that are eligible for assistantship, we search for *either* good *or* senior students. If the two clauses were combined with **and**, we could use the techniques presented in the previous chapter. If the clauses were not correlation predicates, we could use techniques to improve performance for disjunctive predicates (e.g. bypass operators [27]). However, none of the existing techniques is able to unnest *disjunctively* occurring correlation predicates.

Hence, in this chapter, we combine the bypass technique and our unnesting technique which allows an unnested evaluation of disjunctively occurring correlation predicates. So far, this has not been accomplished for any query language.

The main contributions of this chapter are as follows:

- We combine the bypass technique with our unnesting approach to allow for efficient query execution plans in the presence of disjunction.
- We introduce a bypass variant of the kappa-join that allows us to extend our technique to queries where two or more correlation predicates occur in a disjunction.

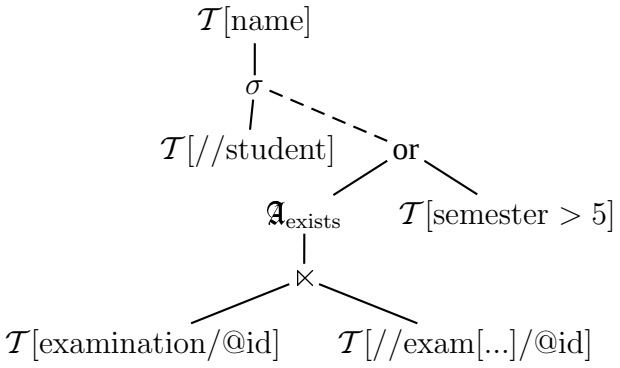


Figure 4.1: Translation sketch for Q8

- We provide experimental results, demonstrating the superiority of our new approach compared to other evaluation techniques and freely available XPath evaluators.

The remainder of this chapter is organized as follows: In the next section, we illustrate the problem existing unnesting techniques have in the presence of disjunction. Section 4.2 presents the bypass technique. In Sec. 4.3, we investigate the case of disjunctive correlation and present our novel bypass kappa-join. We experimentally confirm the efficiency of our approach in Sec. 4.4, discuss related work in Sec. 4.5, and conclude the chapter in the last Section 4.6.

4.1 Problem

Consider the canonical algebra plan for Query Q8 (see Fig. 4.1). This algebra expression is similar to the one presented in Fig. 3.4 for Q3 (see Sec. 3.3.2), except for the *or* function call in the subscript of the selection. Disjunctively occurring literals are translated using an *or* function call. It evaluates to true if either of its producer expressions does.

Because of the extra literal and the *or* function call, the pattern used for the correlation predicate does not match the left-hand side of Equivalence 3.2 or the definition of the kappa-join. Hence, we cannot proceed as for Query Q3. The only technique currently available to improve the canonical plan is the so-called shortcut evaluation of the disjunction, which means that we can avoid evaluation of the expensive correlation predicate for those students where the cheaper literal $\text{semester} > 5$ is true. Below, we recall the bypass technique, which does exactly that.

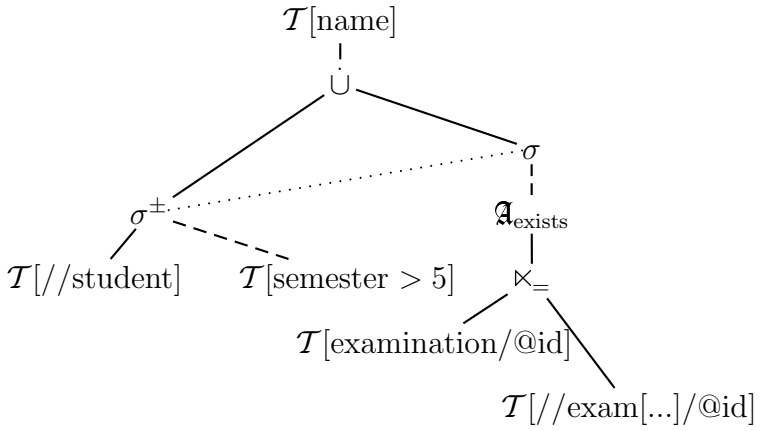


Figure 4.2: Unnesting strategy for Q8 with bypass selection

4.2 Bypass Technique

The bypass technique is used to prevent the unnecessary evaluation of predicates that occur disjunctively [27]. For this, the bypass technique adds a new class of operators to the conventional algebra. In contrast to regular operators, bypass operators have *two* output sequences. The first sequence contains the tuples that qualify for the predicate of the operator. The second sequence consists of those tuples that do not qualify this predicate. The two disjoint sequences are called *true-* and *false-sequence*. The existing bypass technique provides a bypass selection, a bypass join, and a bypass semi-join [27]. For our purposes, we only need the bypass selection.

Consider as a first example the algebra representation of Q8 extended by a bypass selection operator ($\sigma^{\pm}$) for evaluating the cheaper predicate $\text{semester} > 5$. Fig. 4.2 shows the resulting plan. Here and in the following, the false-sequence is indicated by dotted lines. The evaluation according to this plan starts with computing all result tuples for the outer expression ($//\text{student}$).

The bypass selection divides these tuples into two disjoint sequences. The true-sequence contains the students that fulfill the predicate $\text{semester} > 5$. Accordingly, the false-sequence contains the tuples that fail this predicate. The tuples of both sequences form two separate paths, which are merged by a disjoint union ($\dot{\cup}$). The tuples from the false-sequence have to pass the second selection operator computing the complex correlation predicate. This operator is responsible for filtering out those tuples that do not qualify for any of the two predicates. The two sequences are disjoint. Hence, no duplicate elimination is required by $\dot{\cup}$. However, as the XPath semantics requires its result to be in document order, a merge as in merge-sort may be required. This can be done, for example, by numbering the tuples before use or use node ids if they allow to rebuild the order. The final processing of $\mathcal{T}[\text{name}]$ completes the result.

Looking at Fig. 4.2, we are in for a surprise: The bypass selection we introduced

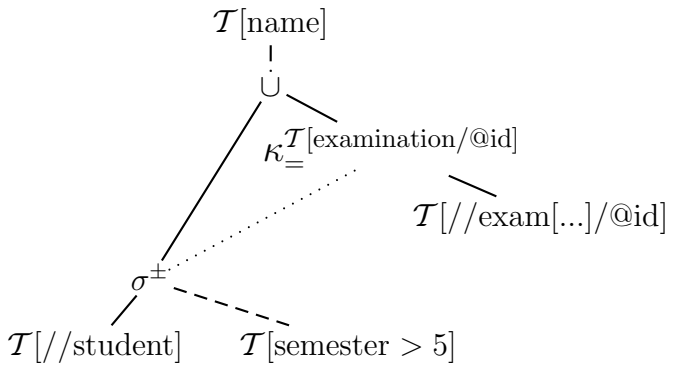


Figure 4.3: Unnesting strategy for Q8 with bypass selection and kappa-join

to allow shortcut evaluation of the disjunction made the pattern required for Equivalence 3.2 or the kappa-join reappear! We discuss in the following subsection how to leverage this for the unnesting of disjunctive queries with a single correlation predicate.

4.3 Disjunctive Unnesting

4.3.1 Unnesting a Single Disjunctive Correlation Predicate

Query Q8 contains a single correlation predicate within a disjunction. Bypass plans have the advantage that the expression in the false-sequence can be optimized separately. In general, whenever there is only a single correlation predicate per disjunction, we can apply unnesting. As seen in Fig. 4.2, we can again recognize the pattern that allows us to integrate the kappa-join for the conjunctive case. In the false-sequence of Fig. 4.2, we can use the kappa-join, yielding the expression shown in Fig. 4.3. Of course, we could also apply Equivalence 3.2 to the pattern in the false-sequence.

In this case, the plan takes advantage of both: (1) shortcut evaluation of the literals connected by disjunction and (2) unnesting of correlation predicates allowing efficient execution if the cheaper predicate in the disjunction fails.

4.3.2 Unnesting Multiple Disjunctive Correlation Predicates

We have seen that the bypass technique facilitates unnesting if there is only one correlation predicate in the disjunction. Unfortunately, if there is more than one, we are again at a loss. Consider as an example the following query. In addition to the good students, we also want to query the database for students that have already been a teaching assistant for a given lecture.

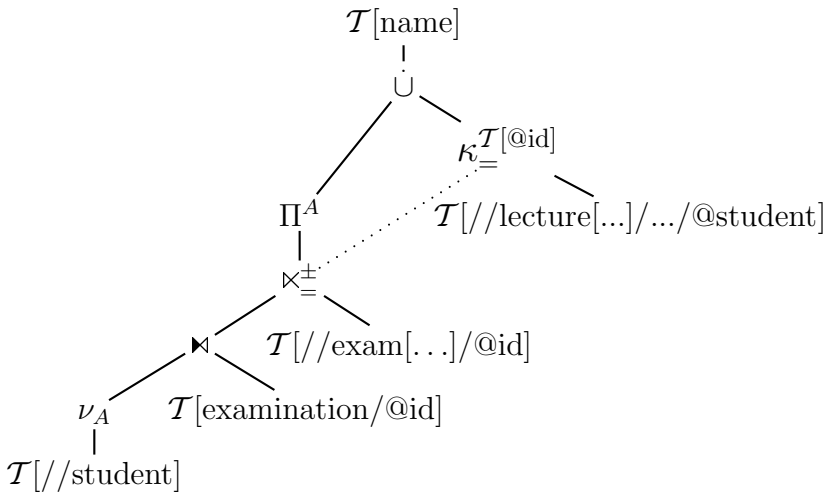


Figure 4.4: Incorrect unnesting strategy for Q9

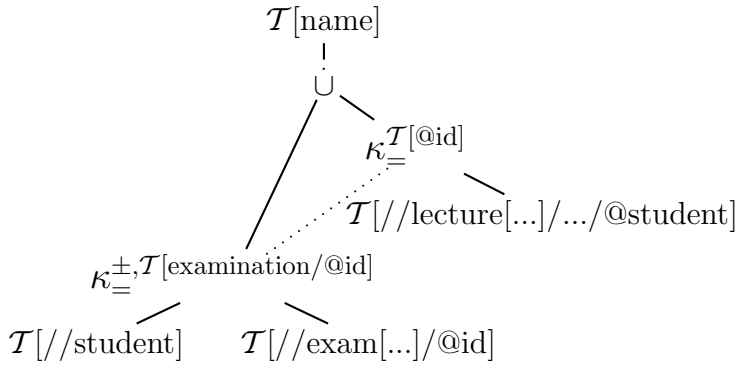


Figure 4.5: Unnesting strategy for Q9 with kappa-join

```
//student[examination/@id= //exam[grade < 'B']/@id or
      @id = //lecture[title='NCT']                          Q9
            /helpers/helper/@student]/name
```

We would like to unnest both correlation predicates. At first glance, it is tempting to apply the unnesting strategy according to Equivalence 3.2. Fig. 4.4 shows an algebra expression for Q9 applying this technique, but using a bypass semi-join instead of a regular semi-join. However, this approach is not feasible. The first d-join on the leftmost branch of the plan eliminates those items produced by $//\text{student}$ for which the dependent expression $\text{exmination}/@\text{id}$ produces an empty result. If we had a conjunctive query, this would not be a problem.

However, the $//\text{student}$ items failing the first disjunct could still qualify for the second disjunct, and dropping them as in Fig. 4.4 produces an incorrect result. Note that the bypass semi-join does not help: It ”comes too late”. Problems of this kind are often solved by using an outer-join [71] or, in this case, outer d-join. However,

this would still require duplicate elimination on ν_A, as shown in the true-sequence.

It turns out that we can do much better by applying the *bypass kappa-join*. As every bypass operator, the bypass kappa-join has two result sequences. The true-sequence is the same as for the regular kappa-join. The tuples in the false-sequence are the ones from the outer producer for which there was no match in the inner producer or for which the link producer returned an empty result. In the false-stream, we now have our familiar pattern and can employ the unnesting strategy as if the correlation predicate was a single correlation predicate. Fig. 4.5 shows the result. This plan finally has everything we want: (1) the evaluation of both correlation predicates can be done in a decorrelated fashion, (2) the kappa-join avoids unneeded duplicate generation and elimination for both correlation predicates, and (3) we have shortcut evaluation and only evaluate the second correlation predicate if the first one fails.

4.4 Evaluation

To show the effectiveness of our approach, we ran experiments with different XPath evaluation engines against our canonical and optimized approaches. We chose the freely available engines

- Xalan C++ 1.8.0 using Xerces C++ version 2.6.0,
- Saxon for Java 8.7.1,
- Berkeley DB XML 2.0.9 (DBXML) using libpathan 1.99 as XPath engine,
- MonetDB 4.8.0 using MonetDB-XQuery-0.8.0,
- the evaluator provided by the XMLTaskForce (XTF), and
- Natix for the execution of the canonical and unnested plans.

We performed the experiments within the environment presented in the last chapter. Moreover, we also generated the same documents as described in Sec. 3.4. For Xalan, Saxon, and XTF, we measured the net time to *execute* the query. The time needed to parse the document and generate the main memory representation is subtracted from the elapsed evaluation time. For the evaluation of MonetDB, Berkeley DB XML, and Natix, we imported the documents into the database. The time needed for this is not included in the execution times. The queries were executed several times with an empty buffer pool and without any indexes.

Queries

We executed performance measurements for both example queries (Q8 and Q9) presented in this chapter. For Natix, we generated several different query evaluation plans. For each of the queries, we generated the canonical plan as specified in Chapter 2. For example, Fig. 4.1 shows the plans for Q8. Further, we generated plans incorporating our optimization strategies. The following table gives a mapping from names for optimized query evaluation plans to figures that illustrate the techniques used.

Query	Name	Figure
Q7	bypass	Fig. 4.2
	kappa	Fig. 4.3
Q8	bypasskappa	Fig. 4.5

4.4.1 Results and Interpretation

Fig. 4.6 contains the results of our performance measurements (elapsed time in seconds). The best execution time(s) for each column in all tables are printed in bold face. Those that did not finish within six hours are marked by DNF (did not finish). For MonetDB, the evaluation of some queries ran out of memory on bigger documents. These cases are denoted by OOM.

Subfigures 4.6(a) and 4.6(b) show the execution times for Q8 and Q9, respectively. For all queries on all documents, our unnested approach performs and scales best. The performance of all other approaches drops considerably when executed on bigger documents. In contrast, our plans containing the kappa-join (Q8) and bypass kappa-join (Q9) almost scale linearly with the size of the document.

4.5 Related Work

We have already discussed related work for unnesting XPath, SQL, OQL, and XQuery at the end of the previous chapter. However, none of the mentioned approaches can handle unnesting in the presence of disjunction.

Several optimization techniques for queries containing disjunctive predicates have been proposed [22, 27, 65]. One of them is the bypass technique [27] which we extend with support for unnesting. Because bypass operators have two output streams, which are unioned later, the resulting expression forms a directed acyclic graph (DAG). Strategies for implementing bypass operators and query evaluation engines that support DAG-structured query plans can be found in [27, 87, 97].

Evaluator	Documents 1	2	3	4	5	6
Xalan	**0.02**	0.23	3.63	54.7	893	12453
DBXML	0.06	0.39	6.87	207	DNF	DNF
MonetDB	0.25	0.36	2.02	36.2	OOM	OOM
Saxon	0.22	0.30	0.62	1.44	7.82	85.4
XTF	0.76	8.60	9180	DNF	DNF	DNF
Natix						
• canonical	0.16	1.64	20.9	333	5598	DNF
• bypass	0.16	1.59	20.7	323	5436	DNF
• kappa	0.03	**0.05**	**0.16**	**0.60**	**2.51**	**9.91**

(a) Q8

Evaluator	Documents 1	2	3	4	5	6
Xalan	0.06	0.75	12.6	199	3201	DNF
DBXML	0.30	1.61	30.2	4057	DNF	DNF
MonetDB	0.31	0.50	3.29	62.9	OOM	OOM
Saxon	0.20	0.28	0.54	1.48	10.9	138
XTF	0.48	5.14	94.8	DNF	DNF	DNF
Natix						
• canonical	0.37	3.49	DNF	DNF	DNF	DNF
• bypasscanonical	0.37	3.43	48.1	749	12492	DNF
• bypasskappa	**0.02**	**0.04**	**0.10**	**0.35**	**1.44**	**5.91**

(b) Q9

Figure 4.6: Results (in sec.) for Q8 and Q9

4.6 Conclusion

We have demonstrated how to efficiently evaluate XPath queries featuring existentially quantified correlation predicates that occur in a disjunction. By injecting the kappa-join — introduced in the previous chapter — with the bypass technique, we are also able to perform an unnested evaluation if the correlation predicate occurs in a disjunction. All other approaches cannot evaluate such a case efficiently. Our performance measurements show that the bypass kappa-join outperforms existing approaches by up to two orders of magnitude.

Our novel unnesting technique is not only applicable to XPath queries containing disjunctions, but also to other query languages. For example, in [13] we have presented the applicability to XQuery. Moreover, in the next chapter, we transfer our approach for optimizing SQL queries in the presence of disjunction.

Chapter 5

Disjunctive Unnesting for SQL

In the last two chapters, we have seen that nested queries can easily become a performance bottleneck because in many cases, they demand a nested-loop evaluation. For XPath, we presented a technique to unnest XPath comparison expressions if the expression occurs either in a conjunction or a disjunction. Moreover, for conjunctive SQL and OQL predicates this problem has also been addressed successfully, e.g. [29, 70, 99]. However, despite the fact that disjunctions occurring inside nested queries are common in practice, we are not aware of any work that treats unnesting nested SQL queries which contain disjunctions, i.e. the linking or correlation predicate occur in a disjunction. Hence, in this chapter, we introduce a technique to unnest disjunctive nested SQL queries.

Key Idea As an example, let us consider a sample analytical query. Assume that we are interested in all European suppliers that deliver a certain part with minimum supply costs *or* have a minimal amount of this part available on stock. In SQL, this query can be formulated as follows:

```
SELECT s_acctbal, s_name, n_name, p_partkey,
       p_mfgr, s_address, s_phone, s_comment
FROM   part, supplier, partsupp, nation, region
WHERE  p_partkey = ps_partkey
AND    s_suppkey = ps_suppkey AND p_size = 15
AND    p_type LIKE '%BRASS'
AND    s_n_key = n_n_key AND n_r_key = r_r_key
AND    r_name = 'EUROPE'
AND    (ps_supplycost=(SELECT min(ps_supplycost)
                       FROM   partsupp, supplier,
                              nation, region
                       WHERE  s_suppkey = ps_suppkey
                       AND    p_partkey = ps_partkey
                       AND    s_n_key = n_n_key
```

```
                    AND    n_r_key = r_r_key
                    AND    r_name = 'EUROPE')
      OR ps_availqty > 2000)
ORDER BY s_acctbal desc, n_name, s_name, p_partkey
```

This query is very similar to TPC-H Query 2. Hence, we refer to it as Query 2d. It exhibits two key components: (1) it features a nested, correlated subquery, and (2) it contains a disjunction. Our unnesting strategy is capable of optimizing nested queries whose linking or correlation predicates occur disjunctively. The key idea is that the nested query block needs to be evaluated only for those tuples of the outer query block that do not pass the cheap and simple predicate `ps_availqty > 2000`. For those tuples, we are currently restricted to an inefficient nested-loop evaluation. However, our novel unnesting technique allows to employ more efficient evaluation algorithms. Consequently, our approach exploits both the short-cut evaluation of the disjunction and the power of unnesting nested queries.

Our Approach The starting point of our approach is to translate SQL into the relational algebra extended with bypass operators [27, 68]. Then, we apply our novel unnesting equivalences, which can cope with disjunctions on a large variety of nested queries. As a result, nested query blocks are removed, and the resulting queries can be evaluated much more efficiently.

As already mentioned in previous chapters, applying unnesting at the algebraic level has mainly three advantages: (1) It is possible to give rigorous correctness proofs for the unnesting equivalences. (2) Unnesting techniques stated as algebraic equivalences are query language independent as long as the query language is translatable into the algebra. (3) Unnesting equivalences can be used during plan generation. This allows to apply them in a cost-based manner. The latter is especially important in our case, since some unnesting strategies do not always result in better plans.

Contributions The main contributions of this chapter are:

- We present equivalences for unnesting algebraic expressions with bypass operators to handle disjunctive linking and correlation predicates where the linking and/or the correlation predicate involves a comparison operator $\theta \in \{=, \neq, <, \leq, >, \geq\}$.

- We present how our approach can be applied to quantified table subqueries with the operators EXISTS, NOT EXISTS, IN, and NOT IN.

- We show how they can be used to effectively unnest SQL queries with scalar subqueries featuring an arbitrary aggregation function in the `select` clause.

- Our techniques can be applied not only to queries with exactly one nested block (simple queries), but also to queries whose nesting has a linear or even a tree structure [85].
- We provide experimental results demonstrating the performance improvements that can be achieved by using our approach.

Limitations As a current limitation, we restrict ourselves to queries exhibiting direct correlation: That is, for linear queries the correlation predicate only refers to attributes of the current block and the direct outer block.

Further, we do exclude linking predicates with linking operators θ `SOME/ANY`, or θ `ALL` with $\theta \in \{<, \leq, >, \geq\}$ from our discussion. However, using aggregate functions that are aware of NULL values, we can still optimize these queries by turning these quantifiers into the aggregate functions $\texttt{min}_{\text{NULL}}$ or $\texttt{max}_{\text{NULL}}$ (cf. ‘’[44]).

Structure of This Chapter The remainder of this chapter is organized as follows: Section 5.1 briefly introduces preliminaries. In Section 5.2, we present our unnesting technique for table subqueries. Section 5.3 contains our unnesting techniques for scalar subqueries. After introducing these approaches, we show their effectiveness with an experimental study (Sec. 5.4). At the end, we summarize related work in Section 5.5 and conclude the chapter with Section 5.6.

5.1 Preliminaries

To start with, we briefly establish a common terminology and repeat Kim’s classification for nested SQL queries [70]. After that, we present the algebra on which our unnesting approach is based on.

5.1.1 Terminology

A *query block* is a `select-from-where` expression. A query containing a query block nested in another query block is called a *nested query*. The containing query block is called *outer* query block, and the contained block is called *inner* query block. An inner query block is also called *nested* query block. Let p be a predicate occurring in the `where` clause of an inner query block. If p refers to attributes defined in the outer query block and to attributes defined in the inner query block, p is called a *correlation predicate*, and the inner query block is called *correlated*. A predicate q in the `where` clause of the outer query block which contains the inner query block as an argument is called *connection predicate*. The operator used in the connection predicate is called *connection operator*. Connection predicates are also called *linking predicates* [23]. In the following, we will stick to the latter term.

If a linking predicate occurs in a disjunction as, for example, in the introductory query, this is called *disjunctive linking*. Analogously, if the correlation predicate occurs in a disjunction, this is called *disjunctive correlation*.

5.1.2 Classification

Kim introduced four types of nested query blocks [70]: *A*, *N*, *JA*, and *J*. Let us refer to a nested query block as *B*. If *B* contains an aggregate function, *B* is of type *A* or *JA* and is called *scalar* subquery. *B* must return a single column. If *B* contains a correlation predicate, it is of type *J* or *JA*. A nested query block *B* that neither has an aggregate function nor a correlation predicate is of type *N*. Query blocks with an aggregation function but no correlation predicate are of type *A*. Nested query blocks of type *N* or *J* are called *table* subqueries. They are connected to their outer query block using the *positive linking operators* `EXISTS`, `SOME/ANY`, and `IN` or *negative linking operators* `NOT EXISTS`, `ALL`, and `NOT IN`, respectively.

While Kim concentrates on classifying single nested query blocks, Muralikrishna additionally classifies queries according to the nesting structure [85]. He subdivides queries with more than one nested block into linear and tree queries: A *Linear (Nested) Query* is a query where at most one block is nested within any block. A *Tree (Nested) Query* is a query with at least one block, which has two or more blocks nested within at the same level. We complete this classification and call a query with exactly one nested block a *Simple (Nested) Query*.

5.1.3 Algebra for Sets

In contrast to previous chapters, the domain of the relational algebra consists of sets of tuples. The core algebra we use in this chapter contains the following operators: union ($\cup$), intersection ($\cap$), set-difference ($\setminus$), projection (Π), renaming operator (ρ), selection (σ), theta-join ($\bowtie$), semi-join ($\ltimes$), and anti semi-join ($\triangleright$) [45]. We denote a disjoint union by $\dot\cup$.

For the purpose of this chapter, we extend this core algebra by five operators: a unary grouping operator (Γ), a binary grouping operator ($\bowtie\!\!\!\!\Gamma$) [29, 102], a left outer-join ($\mathbin{⟕}^{g:f(\emptyset)}$) [29, 31], a numbering operator (ν), and a map operator (χ).

Given the definition of the binary grouping operator in Fig. 5.1, we define the unary grouping operator. The left outer-join ($\mathbin{⟕}^{g:f(\emptyset)}$) is required to address the "count bug" [69, 44], i.e. losing a tuple due to an empty group. Therefore, the function f assigns meaningful values to the attribute g for tuples that have no join partner on the right-hand side. The numbering operator (ν) characterizes each tuple with a unique deterministic number (e.g. a physical tuple identifier). We use the map operator to apply a function to each tuple. Figure 5.1 summarizes the formal definition of the five additional operators.

As a final important extension of our algebra, we allow subscripts to contain algebraic expressions. In our case, such subscripts result from translating nested

query blocks in the `where` clause, i.e. algebraic operators appear in selection predicates.

To translate the linking predicates `EXISTS` and `IN`, we employ the internal function $\exists_p(e)$; for the negated form we use $\nexists_p(e)$. Each of these operators returns true if there exists (does not exist) at least one element in their argument e that satisfies the predicate p. We write $\exists(e)$ and $\nexists(e)$ if the predicate is true, i.e. there is no linking predicate (e.g. `EXISTS` and `NOT EXISTS`). To translate aggregate functions, we use the aggregation operators, e.g. `SUM`, `COUNT`, `MAX`, `MIN`, and `AVG`.

In the previous chapter, we have presented bypass operators [27, 68] in order to effectively deal with disjunction. In this chapter, we use them again in variants that can be used for sets. Hence, we call their outputs positive and negative stream instead of sequence (as in the previous chapter). For example, a selection produces a *positive stream* containing all those tuples for which the selection predicate evaluates to true; the *negative stream* contains the remaining tuples. To denote the positive and negative streams of a bypass operator, we use the superscripts $+$ and $-$, respectively.

For this chapter, we need a bypass selection ($\sigma^{\pm}$), a bypass join ($\bowtie^{\pm}$), a bypass semi-join ($\ltimes^{\pm}$) and a bypass anti semi-join ($\triangleright^{\pm}$). Their definitions on sets are given in the bottom part of Figure 5.1.

Although the algebra is based on sets of tuples, our approach is also applicable for an algebra on bags. In two dedicated sections (see Sec. 5.2.7 and 5.3.7) and in the Appendix C, we elaborate on the correctness of our techniques if the algebra is based on bags.

5.2 Unnesting Table Subqueries

We now present our detailed unnesting techniques along the lines of the classification introduced in Section 5.1.2. As table subqueries (i.e. types *N* & *J*) are less demanding, we start out with them. As type *N* queries can be unnested by applying the unnesting techniques for type *J* queries, we deal with them later in Sec. 5.2.3.

This section is organized as follows: First, we discuss the basic idea of our approach by means of two queries based on a synthetic schema. Second, we present the general solution in the form of algebraic unnesting equivalences. On their left-hand side, they have a selection whose predicate contains disjunctively a quantified algebraic expression. On their right-hand side they introduce a bypass operator. Then, we move on to more advanced issues, i.e. tree queries, linear queries, and duplicate handling.

5.2.1 Disjunctive Linking

The first example query exhibits a nested query block whose disjunctively occurring linking predicate uses the linking operator `IN`. Because the nested block has a

Non-standard operators:

$$
\begin{aligned}
e_1 \ltimes\!\!\!\Gamma_{g;A_1\theta A_2;f} e_2 &:= \{x.A_1 \circ [g:G] | x \in e_1 \wedge \\
&\quad G = f(\{y | y \in e_2 \wedge x.A_1 \theta y.A_2\})\} \\
\Gamma_{g;=A;f}(e_1) &:= \Pi_{A:A'}(\Pi_{A':A}(e_1) \ltimes\!\!\!\Gamma_{g;A=A';f}(e_1)) \\
e_1 \overset{g:f(\emptyset)}{\ltimes\!\!\!\!\!\!\rightthreetimes}_p e_2 &:= e_1 \ltimes_p e_2 \cup \{x \circ z | x \in e_1 \wedge \\
&\quad \nexists y \in e_2 : p(x,z) \wedge \mathcal{A}(z) = \mathcal{A}(e_2) \wedge \\
&\quad g \in \mathcal{A}(e_2) \wedge \forall a \in (\mathcal{A}(e_2) \setminus g) : \\
&\quad (z.a : \mathrm{NULL} \wedge z.g : f(\emptyset))\} \\
\nu_A(e) &:= \{t_i \circ [A:i] | e = \{t_1, \ldots, t_n\}\} \\
\chi_{a:e_2}(e_1) &:= \{x \circ [a : e_2(x)] | x \in e_1\}
\end{aligned}
$$

Bypass operators:

$$
\begin{aligned}
\sigma_p^+(e) &:= \{x | x \in e \wedge p(x)\} \\
\sigma_p^-(e) &:= e \setminus \sigma_p^+(e) \overset{*}{=} \{x | x \in e \wedge \neg p(x)\} \\
e_1 \ltimes_p^+ e_2 &:= \{x \circ y | x \in e_1 \wedge y \in e_2 \wedge p(x,y)\} \\
e_1 \ltimes_p^- e_2 &:= (e_1 \times e_2) \setminus (e_1 \ltimes_p^+ e_2) \\
&\overset{*}{=} \{x \circ y | x \in e_1 \wedge y \in e_2 \wedge \neg p(x,y)\} \\
e_1 \ltimes\!\!\!\!\!\!\rightthreetimes_p^+ e_2 &:= \{x | x \in e_1 \wedge \exists y \in e_2 \wedge p(x,y)\} \\
e_1 \ltimes\!\!\!\!\!\!\rightthreetimes_p^- e_2 &:= e_1 \setminus (e_1 \ltimes\!\!\!\!\!\!\rightthreetimes_p^+ e_2) \\
&\overset{*}{=} \{x | x \in e_1 \wedge \nexists y \in e_2 \wedge p(x,y)\} \\
e_1 \triangleright_p^+ e_2 &:= \{x | x \in e_1 \wedge \nexists y \in e_2 \wedge p(x,y)\} \\
e_1 \triangleright_p^- e_2 &:= e_1 \setminus (e_1 \triangleright_p^+ e_2) \\
&\overset{*}{=} \{x | x \in e_1 \wedge \exists y \in e_2 \wedge p(x,y)\}
\end{aligned}
$$

* only valid for two-valued logic (cf. [27, 68] for details). $[\cdot]$ denotes tuple construction. $\circ$ denotes tuple concatenation. $\mathcal{A}(\mathrm{R})$ is the set of attributes of relation R.

Figure 5.1: Operators of the algebra

correlation predicate in its `where` clause ($R.A_2 = S.B_3$), it is of type *J*.

```
SELECT  DISTINCT *
FROM    R
WHERE   R.A1 IN (SELECT S.B4
                 FROM S
                 WHERE R.A2 = S.B3)
        OR R.A4 > 1500;
```

Q10

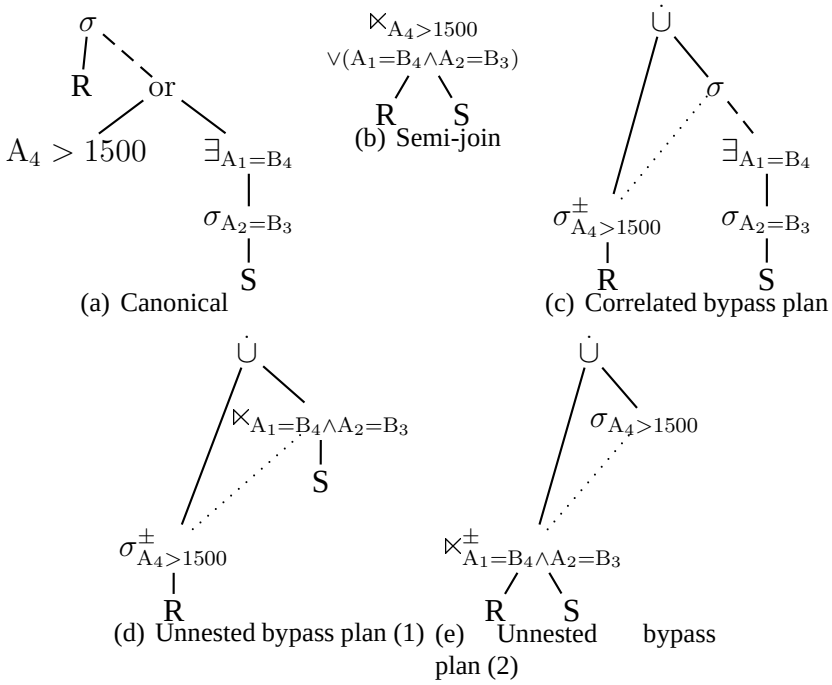


Figure 5.2: Unnesting strategy for Q10 (sketch)

Due to the existential nature of the `IN` operator, the algebraic expression resulting from the translation of the query has an existential quantifier subscripted with the linking predicate. The argument of this quantifier itself is again an algebraic expression. As the existential quantifier occurs in a selection operator, the nesting of the query blocks in the query is reflected by a nesting of algebraic expressions, i.e. the subscript of an algebraic operator again contains an algebraic expression. Translating the query into a nested algebraic expression yields the following:

$$\sigma_{(\exists_{A_1=B_4}(\sigma_{A_2=B_3}(S)))\vee A_4>1500}(R).$$

Fig. 5.2(a) sketches the more readable tree form of this expression. Evaluating the predicate which contains the inner query block for every tuple produced by the outer query block (R) is not very efficient.

To avoid this nested-loop-like evaluation, we would like to unnest the subquery. In the conjunctive case, nested queries are usually unnested by applying (semi-)joins. Recast into the algebraic framework, this amounts to applying the following equivalence:

$$\sigma_{\exists_{A_1=B_1}(\sigma_{A_2=B_2}(S))}(R) \quad \equiv \quad R \ltimes_{A_1=B_1\wedge A_2=B_2} S. \tag{5.1}$$

Let us see what happens if we apply this traditional technique to our translated query. The resulting expression (called semi-join plan) is shown in Fig. 5.2(b).

The problem is that efficient implementations of (semi-)joins only exist for equi-joins, whereas in our case, the semi-join condition contains a disjunction at the top level. Implementations other than a simple nested-loop evaluation are beyond reach. Thus, we are again stuck with a nested-loop evaluation.

Let us take a closer look at the query. Assume that a tuple from R satisfies `R.A4` $>$ `1500`. Then, we do not have to check `R.A1 IN` ... for it: it qualifies independently of the outcome of this check. Further, if a tuple from R does not satisfy `R.A4` $>$ `1500`, it must satisfy `R.A1 IN` ... in order to qualify. Thus, it does make sense to split the tuple stream produced by scanning R into two independent streams: one containing those tuples satisfying `R.A4` $>$ `1500` and one with the remaining tuples. The latter then needs to be filtered by `R.A1 IN` .... Finally, as the two streams are disjoint, a disjoint union ($\dot\cup$) on these two streams suffices to produce the final result. Bypass operators capture exactly this kind of reasoning. This is why we want to use them for unnesting. Let us introduce a bypass selection with predicate `R.A4` $>$ `1500`. Fig. 5.2(c) shows the result. The positive stream of the bypass selection (denoted by a solid line) directly contributes to the final result whereas the negative stream (denoted by dots) is filtered by a selection with the algebraic equivalent of `R.A1 IN` .... This equivalent, $\exists_{\mathrm{A}_1=\mathrm{B}_4}(\sigma_{\mathrm{A}_2=\mathrm{B}_3}(\mathrm{S}))$, is the filter predicate of a top-level selection and itself contains an algebraic expression (especially a scan of S). Hence, we still have a rather inefficient nested algebraic expression demanding a nested-loop evaluation. However, we are prepared for the final, performance-improving step.

We now introduce a semi-join to unnest the query (according to Equivalence 5.1). Although the details are given in the next subsection, we still would like to give the result:

$$\begin{aligned} \mathrm{e} &= \mathrm{e}_1 \dot\cup \mathrm{e}_2 \\ \mathrm{e}_1 &= \sigma^{+}_{\mathrm{A}_4>1500}(\mathrm{R}) \\ \mathrm{e}_2 &= (\sigma^{-}_{\mathrm{A}_4>1500}(\mathrm{R}))\ltimes_{\mathrm{A}_1=\mathrm{B}_4\wedge\mathrm{A}_2=\mathrm{B}_3}(\mathrm{S}). \end{aligned}$$

Fig. 5.2(d) shows this expression in a more readable form. The semi-join now operates on the negative stream of the bypass selection and the scan of S. Since its condition is now a conjunction of two equality predicates, it can be evaluated very efficiently. We verify this claim in our experiments in Section 5.4.

Remark. It is important to recognize that commuting the bypass selection with the semi-join (see Fig. 5.2(e)) is also feasible. This enables further optimization potential. Assume that the second predicate is expensive to evaluate. Then it may be cheaper to perform the semi-join first. This situation is recognized by comparing ranks of the predicates: the one with the lower rank should be evaluated first [100]. For a predicate p, the rank ($\mathrm{rank}(p)$) is defined as $\frac{s-1}{c}$, where s is the selectivity of predicate p, and c is the cost required to evaluate p.

5.2.2 Disjunctive Correlation

In the previous subsection, we have shown a technique to unnest (nested) queries whose linking predicate occur in a disjunction. In contrast, the following query contains a disjunctively occurring correlation predicate, i.e. disjunctive correlation:

```
SELECT  DISTINCT *
FROM    R
WHERE   R.A1 IN (SELECT S.B4
                 FROM S
                 WHERE R.A2 = S.B3
                 OR S.B4 > 1500)
```

Q11

Fig. 5.3(a) depicts the canonical translation of this query as a sketch. Note that we cannot unnest this query with the technique of the first example, because the `where` clause of the nested query contains a disjunction with two predicates, one of which is the correlation predicate ($R.A_2 = S.B_3$).

Consider a tuple r of R. If there exists a tuple s in S such that s passes the tests $s.B_4 > 1500$ and $r.A_1 = s.B_4$, then r is contained in the result. This is expressed by the bypass semi-join in Fig 5.3(b). If no such tuple in S exists, r becomes part of the negative output of the bypass semi-join. For r to qualify, there must be a tuple s in S such that $r.A_1 = s.B_4$ and $r.A_2 = s.B_3$. As those tuples in S with $s.B_4 > 1500$ have been checked before, we only need to consider those $s \in S$ not having $s.B_4 > 1500$. Thus, we have to perform a semi-join on the negative output of the bypass semi-join and the negative output of the selection (see again Fig 5.3(b)).

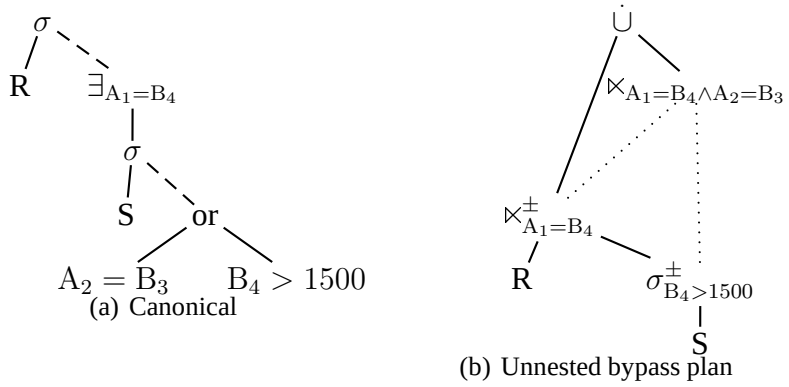


Figure 5.3: Unnesting strategy for Q11 (sketch)

$$\sigma_{\exists_{A_1=B_1}(S)\vee p}(R) \equiv e_1 \dot{\cup} e_2 \quad (5.2)$$
$$e_1 := \sigma_p^+(R)$$
$$e_2 := (\sigma_p^-(R)) \ltimes_{A_1=B_1} S$$
$$\sigma_{\exists_{A_1=B_1}(S)\vee p}(R) \equiv e_1 \dot{\cup} e_2 \quad (5.3)$$
$$e_1 := R \ltimes^+_{A_1=B_1} S$$
$$e_2 := \sigma_p(R \ltimes^-_{A_1=B_1} S)$$
$$\sigma_{\nexists_{A_1=B_1}(S)\vee p}(R) \equiv e_1 \dot{\cup} e_2 \quad (5.4)$$
$$e_1 := \sigma_p^+(R)$$
$$e_2 := (\sigma_p^-(R)) \triangleright_{A_1=B_1} S$$
$$\sigma_{\nexists_{A_1=B_1}(S)\vee p}(R) \equiv e_1 \dot{\cup} e_2 \quad (5.5)$$
$$e_1 := R \triangleright^+_{A_1=B_1}(S)$$
$$e_2 := \sigma_p(R \triangleright^-_{A_1=B_1} S)$$

Figure 5.4: Equivalences for disjunctive N queries

5.2.3 Equivalences

After having worked out the general idea by means of two examples, we now introduce the general form of our technique by means of our novel equivalences. They are shown in Figure 5.4 for type *N* subqueries and Figure 5.5 for type *J* subqueries. We provide proofs of these equivalences in Appendix C. These equivalences allow us, for example, to formally derive the unnested plans presented before. On their left-hand side, they contain a selection with a predicate that results from the translation of a nested type *N* or type *J* query block. On their right-hand side, they have an unnested algebraic expression with bypass operators.

We first discuss the equivalences for disjunctive linking, then those for disjunctive correlation.

Disjunctive Linking

We split the discussion of the equivalences for disjunctive linking into two parts: one for positive and one for negative linking predicates.

Positive Linking Predicates Equivalences 5.6 and 5.7 have been implicitly applied to our sample query Q10. The former yields the plan shown in Fig. 5.2(d), the latter the one in Fig. 5.2(e).

Both equivalences unnest table subqueries that exhibit a positive linking predicate occurring in a disjunction. The former employs the bypass technique to a

$$\sigma_{\exists_{\mathrm{A}_1=\mathrm{B}_1}(\sigma_{\mathrm{A}_2=\mathrm{B}_2}(\mathrm{S}))\vee p}(\mathrm{R}) \equiv e_1 \dot{\cup} e_2 \quad (5.6)$$
$$e_1 := \sigma_p^+(\mathrm{R})$$
$$e_2 := (\sigma_p^-(\mathrm{R}))\ltimes_{\mathrm{A}_1=\mathrm{B}_1\wedge\mathrm{A}_2=\mathrm{B}_2}\mathrm{S}$$
$$\sigma_{\exists_{\mathrm{A}_1=\mathrm{B}_1}(\sigma_{\mathrm{A}_2=\mathrm{B}_2}(\mathrm{S}))\vee p}(\mathrm{R}) \equiv e_1 \dot{\cup} e_2 \quad (5.7)$$
$$e_1 := \mathrm{R}\ltimes^+_{\mathrm{A}_1=\mathrm{B}_1\wedge\mathrm{A}_2=\mathrm{B}_2}\mathrm{S}$$
$$e_2 := \sigma_p(\mathrm{R}\ltimes^-_{\mathrm{A}_1=\mathrm{B}_1\wedge\mathrm{A}_2=\mathrm{B}_2}\mathrm{S})$$
$$\sigma_{\nexists_{\mathrm{A}_1=\mathrm{B}_1}(\sigma_{\mathrm{A}_2=\mathrm{B}_2}(\mathrm{S}))\vee p}(\mathrm{R}) \equiv e_1 \dot{\cup} e_2 \quad (5.8)$$
$$e_1 := \sigma_p^+(R)$$
$$e_2 := (\sigma_p^-(R))\triangleright_{\mathrm{A}_1=\mathrm{B}_1\wedge\mathrm{A}_2=\mathrm{B}_2}(\mathrm{S})$$
$$\sigma_{\nexists_{\mathrm{A}_1=\mathrm{B}_1}(\sigma_{\mathrm{A}_2=\mathrm{B}_2}(\mathrm{S}))\vee p}(\mathrm{R}) \equiv e_1 \dot{\cup} e_2 \quad (5.9)$$
$$e_1 := \mathrm{R}\triangleright^+_{\mathrm{A}_1=\mathrm{B}_1\wedge\mathrm{A}_2=\mathrm{B}_2}\mathrm{S}$$
$$e_2 := \sigma_p(\mathrm{R}\triangleright^-_{\mathrm{A}_1=\mathrm{B}_1\wedge\mathrm{A}_2=\mathrm{B}_2}\mathrm{S})$$
$$\sigma_{\exists_{\mathrm{A}_1=\mathrm{B}_1}(\sigma_{\mathrm{A}_2=\mathrm{B}_2\vee p}(\mathrm{S}))}(\mathrm{R}) \equiv e_1 \dot{\cup} e_2 \quad (5.10)$$
$$e_1 := \mathrm{R}\ltimes^+_{\mathrm{A}_1=\mathrm{B}_1} e_3$$
$$e_2 := (\mathrm{R}\ltimes^-_{\mathrm{A}_1=\mathrm{B}_1} e_3)\ltimes_{\mathrm{A}_1=\mathrm{B}_1\wedge\mathrm{A}_2=\mathrm{B}_2}(\sigma_p^-(\mathrm{S}))$$
$$e_3 := \sigma_p^+(\mathrm{S})$$
$$\sigma_{\nexists_{\mathrm{A}_1=\mathrm{B}_1}(\sigma_{\mathrm{A}_2=\mathrm{B}_2\vee p}(\mathrm{S}))}(\mathrm{R}) \equiv e_1 \dot{\cup} e_2 \quad (5.11)$$
$$e_1 := \mathrm{R}\triangleright^+_{\mathrm{A}_1=\mathrm{B}_1} e_3$$
$$e_2 := (\mathrm{R}\triangleright^-_{\mathrm{A}_1=\mathrm{B}_1} e_3)\triangleright_{\mathrm{A}_1=\mathrm{B}_1\wedge\mathrm{A}_2=\mathrm{B}_2}\sigma_p^-(\mathrm{S})$$
$$e_3 := \sigma_p^+(\mathrm{S})$$

Figure 5.5: Equivalences for disjunctive J queries

disjunctively occurring subquery and unnests the subquery in the negative stream of a bypass selection. The positive stream contains all tuples that match a lower-ranked predicate p [100]. A final union merges both streams without having to eliminate duplicates. The latter equivalence uses the same idea, but the subquery is evaluated first, and the evaluation of a higher-ranked (expensive) predicate p is postponed into the negative stream.

Equivalences 5.2 and 5.3 are similar but can be used for unnesting subqueries of type *N*. As a result of the missing correlation predicate, the unnested query only contains a simple join condition.

Negative Linking Predicates Equivalences 5.8 and 5.9 unnest queries with a negative linking predicate (i.e. of the form `NOT IN`), which occurs disjunctively. Both

equivalences are similar to those for positive linking predicates but feature an anti semi-join. The first equivalence takes advantage of an anti semi-join in the negative stream to unnest the subquery. Note that in order to evaluate the correlation predicate it also becomes a join predicate of the anti semi-join. The second equivalence uses a bypass anti semi-join for evaluating the subquery first and postpones the evaluation of a higher-ranked predicate p into the negative stream.

Equivalences 5.4 and 5.5 are again similar but useful for unnesting subqueries of type *N*, i.e. exhibiting a predicate containing `NOT EXISTS`. The anti semi-join on the right-hand side of both equivalences only features a single join predicate for evaluating the negative linking predicate.

Disjunctive Correlation

Equivalences 5.10 and 5.11 unnest queries with a disjunctive correlation predicate. Again, we discuss those exhibiting a positive linking predicate first and then discuss unnesting of queries featuring a negative linking predicate. In both equivalences, the predicate p can be a simple predicate or a nested query itself.

Positive Linking Predicates Equivalence 5.10 is used for unnesting queries whose linking operator is `IN`. The core benefit of this equivalence results from the clever filtering of tuples in R. First, the linking predicate is only checked for tuples of S that match the cheaper predicate p. Only the remaining tuples of R are checked for matches that pass the correlating predicate. This equivalence can also be used for unnesting queries whose linking operator is `EXISTS`. In this case, there is no linking and, hence, $\mathrm{A}_1 = \mathrm{B}_1$ is set to true within the equivalence.

Negative Linking Predicates Equivalence 5.11 handles the linking operators `NOT EXISTS` and `NOT IN`. It applies the same strategy as explained in the previous equivalence that handles positive linking predicates. However, note that now an anti semi-join replaces the semi-join to check for the negative linking operator. In case of the linking operator `NOT EXISTS`, the linking predicate within the equivalence is true.

5.2.4 Completeness of Equivalences

It is important to make sure that the equivalences suffice to unnest all nested queries with linking predicate `IN` or `EXISTS` and their negated counterparts. The reason is that the translation of these queries results exactly in the patterns on the left-hand side of our equivalences. When no correlation predicate exists, the equivalences from Fig. 5.4 can be applied. Otherwise, we apply those from Fig. 5.5.

5.2.5 Tree Queries

Obviously, the equivalences introduced in the last subsection are capable of unnesting simple nested queries, i.e. those containing just a single nested block. It might be less obvious that they also allow us to unnest tree and linear queries. In this subsection and the following one, we demonstrate that this is indeed the case.

Let us start with the following example of a tree query:

```
SELECT  DISTINCT *
FROM    R
WHERE   A1 NOT IN (SELECT B1
                   FROM    S
                   WHERE   A2 = B2)
        OR
        A3 IN   (SELECT C1
                 FROM    T
                 WHERE   A4 = C2)
```

Q12

In this query, we have two nested query blocks on the same level, one using `NOT IN` and the other using `IN`. Their linking predicates are connected by a disjunction. Additionally, both query blocks are correlated, i.e. of type *J*.

We briefly demonstrate that Equivalences 5.9 and 5.1 enable us to unnest this query. The canonical translation of the Query Q12 is given in Fig. 5.6(a). First, we can unnest this query by applying Equivalence 5.9. This introduces a bypass anti semi-join (for the negative linking predicate) whose join predicates are the linking and correlation predicates. The evaluation of the second nested query is postponed into the negative stream of this bypass anti semi-join. Second, we apply Equivalence 5.1 — the equivalence used for the conjunctive case — to the query in the negative stream. Fig. 5.6(b) shows the final unnested result.

Analogously, we can also apply Equivalence 5.7 for unnesting the subquery connected with the positive linking operator. This equivalence introduces a bypass semi-join and postpones the second nested query into the false stream of this operator. Then, we can apply the following equivalence to the operators in the false stream.

$$\sigma_{\exists_{\neq \mathrm{A}_1=\mathrm{B}_1}(\sigma_{\mathrm{A}_2=\mathrm{B}_2}(\mathrm{S}))}(\mathrm{R}) \quad \equiv \quad \mathrm{R} \triangleright_{\mathrm{A}_1=\mathrm{B}_1 \wedge \mathrm{A}_2=\mathrm{B}_2} \mathrm{S}. \tag{5.12}$$

This equivalence was developed for unnesting the conjunctive case with negative linking operators and is similar to Equivalence 5.1 for positive linking operators.

5.2.6 Linear Queries

In the following, we demonstrate that all of the strategies, we developed for simple type *N* or type *J* queries, also work for linear queries. Moreover, in this section, we

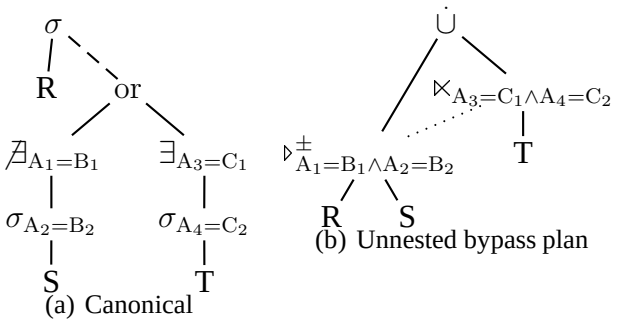


Figure 5.6: Unnesting strategy for Q12 (sketch)

also show that our equivalences can be applied in either a top-down or a bottom-up fashion on the algebra tree.

Let us start with a demonstration of our technique using a linear query with two subqueries of type *N*.

```
SELECT  A1
FROM    R
WHERE   A1 IN   (SELECT B1
                 FROM   S
                 WHERE  B2 IN (SELECT C1
                               FROM T)
                        OR c2)
        OR c1
```

Q13

Fig. 5.7 compares both strategies for Query Q13. In Fig. 5.7(a), we present the canonical algebra expression of this query. Starting with this expression, we have two choices: We can either start to unnest the deepest subquery that has a nested query in one of its predicates (see Fig. 5.7(b)) or start with unnesting the top-level query (see Fig. 5.7(d)). In the former case, we apply Equivalence 5.2 to the subquery in the middle, in the latter case, we apply the same equivalence to the top most query. In either case, Equivalence 5.2 does not modify or influence the expressions which represent the remaining subqueries. Hence, as we can also see in Figures 5.7(c) and 5.7(e), in both cases, there are no blind alleys, and both approaches yield a correctly unnested plan.

Next, we briefly demonstrate that the same holds for linear queries whose subqueries are of type *J*. Therefore, consider the following query containing two subqueries in a linear chain, each of which is correlated:

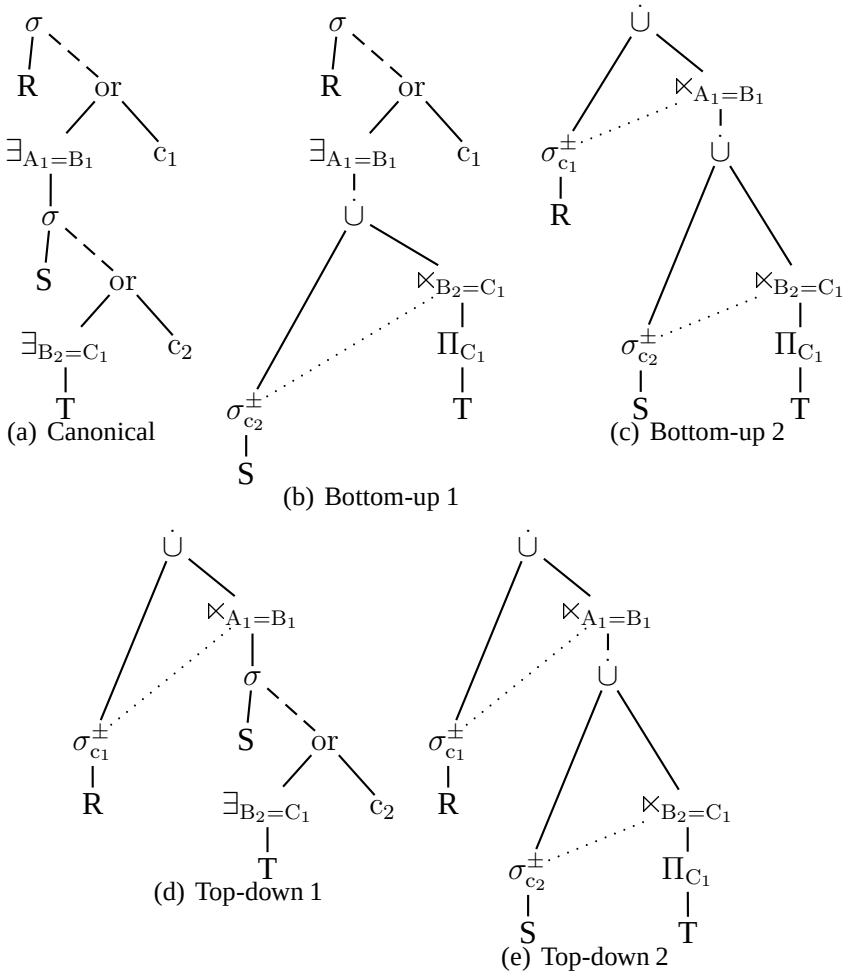


Figure 5.7: Unnesting strategy for Q13 (sketch)

```
SELECT A1
FROM   R
WHERE A1 IN (SELECT B4
             FROM    S
             WHERE   A2 = B3
             OR      B1 IN (SELECT C4
                            FROM T
                            WHERE B2 = C3))
```

Q14

In this query, the deepest nested query block is connected to its outer query block by a disjunction. Note that we have restricted ourselves to queries whose correlation predicate consists of attributes or variables that are defined in a directly adjacent outer block. We depict our top-down unnesting strategy in Figure 5.8. Subfigure 5.8(a) contains the canonical translation. Applying Equivalence 5.10 for positive linking operators, as already shown for Query Q11, yields the intermediate

plan from Fig. 5.8(b). Although the middle query block is already unnested, we would like to unnest the deepest nested block, too. This can finally be done applying Equivalence 5.1. Subfigure 5.8(c) shows the final result.

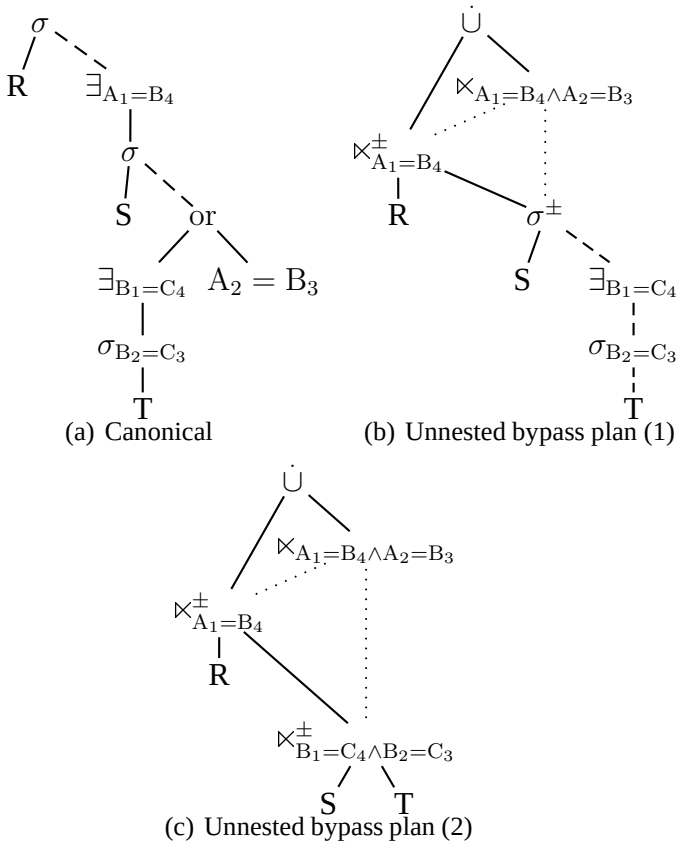


Figure 5.8: Unnesting strategy for Q14 (sketch)

We have already shown that it does not matter whether we apply our unnesting strategy in a top-down or a bottom-up fashion for subqueries of type *J*. Note that for a generally nested tree or linear query, we have a choice: We can apply all equivalences in a top-down or a bottom-up fashion. Taking the former approach, we would successively apply the equivalences to the outermost query block. For the bottom-up approach, we would use the innermost block.

5.2.7 Duplicate Handling

Our unnesting equivalences are defined for an algebra over sets of tuples. However, we now briefly argue that all of the equivalences presented in Figures 5.4 and 5.5 are also correct for an algebra over multisets. For formal proofs instead of a verbal argumentation, we refer to Appendix C. The validity of our equivalences for mul-

tisets is necessary because by default, SQL queries do not remove duplicates, i.e. they yield multisets of tuples.

For our equivalences for table queries, two issues have to be considered. The first is the bypass technique. Bypassing does not cause any problems because it splits its input into two disjoint multisets, i.e. equal tuples go the same way. The final (disjoint) union merges both inputs without duplicate elimination. Hence, no duplicates are falsely eliminated. Further, no new (false) duplicates are introduced as long as there are no expressions producing duplicates in any of the two streams.

Hence, the second source of possible problems are the operators that are applied in the streams. All equivalences from Figures 5.5 employ a semi-join or an anti semi-join. For both operators, implementations are conceivable which adhere to the selection-like semantics, i.e. they neither wrongly eliminate nor generate duplicates. Hence, it is safe to apply our unnesting techniques to multisets.

The correctness of the equivalences from Fig. 5.4 can be verified by replacing the correlation predicate of the according equivalence for type *J* queries with `true`.

5.3 Unnesting Scalar Subqueries

Unnesting scalar queries is difficult and error-prone. Particularly, empty groups and duplicates (cf. [69]) have been sources of errors. As a new challenge, we now support unnesting queries with disjunctive linking or correlation.

Analogically to the last section, this section is organized as follows: First, we start with a discussion of our approach by means of two simple queries. Second, we present our unnesting equivalences for simple queries. Last, we elaborate on the unnesting of linear queries, tree queries, and finish with a discussion of correctness in the presence of duplicates.

According to Kim's classification, scalar subqueries can be of type *A* or *JA* [70]. Subqueries of type *A* are easy to handle. Their result can be computed independently of the outer query, and the materialization costs are negligible. Thus, it suffices to materialize the computed result. As their treatment is so simple, we do not discuss them any further but concentrate on the more challenging type *JA* queries.

5.3.1 Disjunctive Linking

In the following query, the subquery is of type *JA*, as it contains a predicate which refers to the attribute A_2, which is defined in the outer block, and the attribute B_2, which is defined in the inner block:

```
SELECT  DISTINCT *
FROM    R
WHERE   A1 =  (SELECT COUNT(DISTINCT *)
               FROM    S
               WHERE   A2 = B2)
        OR A4 > 1500
```

Q15

The linking predicate compares the attribute A_1 with the result of the aggregation (i.e. count) from the `select` clause of the inner query. Moreover, this linking predicate occurs in a disjunction. Translating this query into the algebra yields the following expression:

$$\sigma_{\mathrm{A}_1=\mathrm{count}(\sigma_{\mathrm{A}_2=\mathrm{B}_2}(\mathrm{S}))\vee\mathrm{A}_4>1500}(\mathrm{R}).$$

Fig. 5.9(a) presents this canonical evaluation plan in a more readable form.

For the evaluation of this query, the inner query has to be evaluated for every tuple produced by the outer query block, i.e. in nested loops. Obviously, this is not very efficient. In order to unnest type *JA* queries in the conjunctive case, it is common practice to apply grouping on the correlation attributes of the inner query to perform the aggregation. Then, an outer-join is used to accomplish the match with the tuples from the outer query block with the grouped and aggregated result [70, 71]. The following algebraic equivalence captures this procedure:

$$\sigma_{\mathrm{A}_1\theta\mathrm{f}(\sigma_{\mathrm{A}_2=\mathrm{B}_2}(\mathrm{S}))}(\mathrm{R}) \equiv \Pi_{\mathcal{A}(R)}(\sigma_{\mathrm{A}_1\theta\mathrm{g}}(\mathrm{R}\mathbin{⟕}^{g:f(\emptyset)}_{\mathrm{A}_2=\mathrm{B}_2}(\Gamma_{g;=\mathrm{B}_2;\mathrm{f}}(\mathrm{S})))). \quad (5.13)$$

If the predicate in the outer query block of our example query was a conjunction, we could apply this equivalence without hesitation. However, if we apply this equivalence to the algebra expression of the query, the resulting plan contains an outer-join with a disjunctive join predicate. In this case, the only known implementation is the rather inefficient nested-loop implementation.

Equivalence 5.13 utilizes grouping and an outer-join to unnest the correlated subquery. However, it is restricted to the correlation predicate being an equality comparison. The binary grouping operator in the following equivalence roughly results from merging the grouping operator and the outer-join [29].

$$\sigma_{\mathrm{A}_1\theta_1\mathrm{f}(\sigma_{\mathrm{A}_2\theta_2\mathrm{B}_2}(\mathrm{S}))}(\mathrm{R}) \equiv \sigma_{\mathrm{A}_1\theta_1\mathrm{g}}(\mathrm{R}\bowtie\Gamma_{\mathrm{g};\mathrm{A}_2\theta_2\mathrm{B}_2;\mathrm{f}}\mathrm{S}) \quad (5.14)$$

This equivalence requires that A_2 is a super key of `R` but allows for general comparisons θ_2 of the correlation predicate. For both Equivalences 5.13 and 5.14, it is required that $\mathrm{g} \notin \mathcal{A}(\mathrm{R}) \cup \mathcal{A}(\mathrm{S})$.

Let us take a closer look at example Query Q15. Assume that a tuple from R satisfies $\mathrm{A}_4 > 1500$. Then, we do not have to check $\mathrm{A}_1 = \ldots$ for it: it qualifies independently of the result of this check. Further, if a tuple from R does not satisfy $\mathrm{A}_4 > 1500$, it must satisfy $\mathrm{A}_1 = \ldots$ in order to qualify. Thus, it does make sense

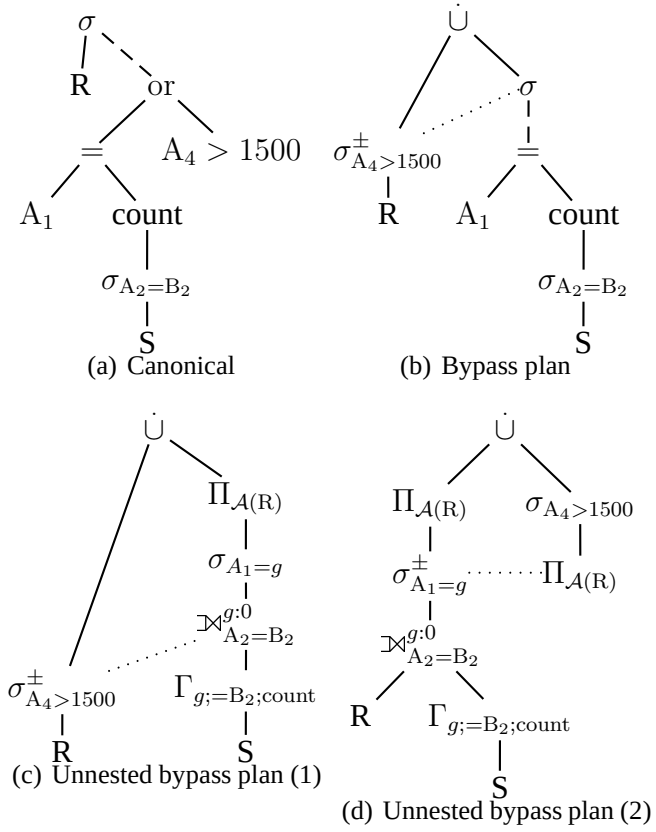


Figure 5.9: Unnesting strategy for Q15 (sketch)

to split the tuple stream produced by scanning R into two independent streams: one containing those tuples satisfying $\mathrm{A}_4 > 1500$ and one with the remaining tuples. The latter then needs to be filtered by $\mathrm{A}_1 = \ldots$. Finally, as the two streams are disjoint, a disjoint union ($\dot\cup$) on them suffices to produce the final result. Let us therefore introduce a bypass selection with predicate $\mathrm{A}_4 > 1500$. The following algebraic expression results from this:

$$\begin{aligned} \mathrm{e} &= \mathrm{e}_1 \mathbin{\dot\cup} \mathrm{e}_2 \\ \mathrm{e}_1 &= \sigma^{+}_{\mathrm{A}_4>1500}(\mathrm{R}) \\ \mathrm{e}_2 &= \sigma_{\mathrm{A}_1=\mathrm{count}(\sigma_{\mathrm{A}_2=\mathrm{B}_2}(\mathrm{S}))}(\sigma^{-}_{\mathrm{A}_4>1500}(\mathrm{R})). \end{aligned}$$

Fig. 5.9(b) shows the more readable result. The positive stream of the bypass selection checking $\mathrm{A}_4 > 1500$ (denoted by a solid line) directly contributes to the final result. In addition, the negative stream (denoted by dots) is filtered by a selection with the algebraic equivalent of $\mathrm{A}_1 = \ldots$.

With this expression as a starting point, we can derive the unnested bypass plan shown in Fig. 5.9(c). Those tuples of R that satisfy the predicate $\mathrm{A}_4 > 1500$ directly

contribute to the result. Only for the remaining tuples, we need to check the condition expressed by $\mathrm{A}_1 = \ldots$. This check is represented in the plan by the same trick used to unnest conjunctively nested queries. In a first step, we group by the linking attribute B_2 of the inner query and calculate the aggregate. Then, we perform an outer-join. For those tuples of R that do not find a join partner, the default handling of the outer-join assures correctness. Last, we evaluate the linking predicate. It has been rewritten since the aggregation result has been materialized in the attribute g. A final projection on the attributes of R guarantees the same schema in the positive as well as the negative stream before unioning the two streams.

Remark. As for disjunctive table queries, we can commute the bypass selection with the selection in the negative stream (see Fig. 5.9(d)). That is, if the predicate $\mathrm{A}_4 > 1500$ was a very expensive one, we could evaluate the subquery first. In this case, the selection checking the linking predicate turns into a bypass selection, and the predicate $\mathrm{A}_4 > 1500$ is evaluated only in the negative stream of the bypass selection. A projection on the attributes of R in both streams ensures the final schema.

5.3.2 Disjunctive Correlation

Not only the linking predicate can occur in a disjunction. The following query contains a disjunctively occurring correlation predicate, i.e. disjunctive correlation:

```
SELECT  DISTINCT *
FROM    R
WHERE   A1 =  (SELECT COUNT(*)
               FROM   S
               WHERE  A2 = B2
                      OR B4 > 1500)
```

Q16

The aggregation function in the `select` clause of the nested query combines all tuples that pass the correlation predicate $\mathrm{A}_2 = \mathrm{B}_2$ *or* the simple predicate $\mathrm{B}_4 > 1500$.

Similar to the canonical translation of Query Q15, but with the disjunction in the selection predicate of the nested selection, the canonical translation gives us (see also Figure 5.10(a))

$$\sigma_{\mathrm{A1}=\mathrm{count}(\sigma_{\mathrm{A_2}=\mathrm{B_2}\vee\mathrm{B_4}>1500}(\mathrm{S}))}(\mathrm{R}).$$

Unnesting is not possible with any of the existing techniques. For the following, we refer to the plan in Fig. 5.10(b). The general idea to unnest this query is based on two facts: (1) the aggregation function (in this case count) is decomposable [30], and (2) the predicate $\mathrm{B}_4 > 1500$ can be evaluated independently of the outer query. This allows us to calculate the total count of the inner query from adding up the counts calculated for two disjoint subsets. Take a look at the bottom of the plan in Fig. 5.10(b). In the positive stream of the bypass selection (denoted by a solid line), we count all tuples from relation S that satisfy the predicate $\mathrm{B}_4 > 1500$.

Those tuples of S that do not satisfy $\mathrm{B}_4 > 1500$ go into the negative stream. Here, they have to pass the correlation predicate before they contribute to the total count. Hence, we group them and evaluate the count function for each group. Analogously to the general unnesting strategy (see Equivalence 5.13), we apply an outer-join to perform the match with the outer relation R and — in order to avoid the count bug — assign 0 to the attribute g_1 for those tuples from R that do not have a join partner. At the end, we need a map operator to add up the separately calculated values for g_1 and g_2 to give the total count g. The subsequent selection with predicate $\mathrm{A}_1 = \mathrm{g}$ checks the linking predicate. The final projection assures that the result only contains attributes from R.

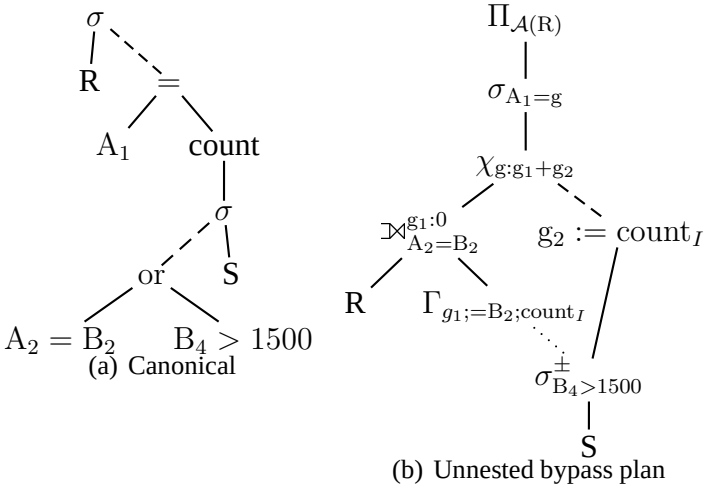


Figure 5.10: Unnesting strategy for Q16 (sketch)

5.3.3 Equivalences

Having presented the general approach, we present our general unnesting rewrites for scalar queries of type *JA*. However, first we need to define *decomposability* of aggregate functions [30]. Let X, Y, and Z be sets with $X = Y \dot\cup Z$ and $Y \dot\cap Z = \emptyset$. A scalar aggregate function $f : X \to \mathcal{N}$ is *decomposable* if there exist functions

$$\begin{aligned} f_I : X &\to \mathcal{N}' \\ f_O : \mathcal{N}', \mathcal{N}' &\to \mathcal{N} \end{aligned}$$

with $f(X) = f_O(f_I(Y), f_I(Z))$. Fortunately, the SQL aggregation functions used most often are decomposable:

$$\begin{aligned}
\text{count(X)} &\equiv \text{count}_I\text{(Y) + count}_I\text{(Z)} \\
\text{sum(X)} &\equiv \text{sum}_I\text{(Y) + sum}_I\text{(Z)} \\
\text{avg(X)} &\equiv \tfrac{\text{sum}_I(Y)+\text{sum}_I(Z)}{\text{count}_I(Y)+\text{count}_I(Z)} \\
\text{min(X)} &\equiv \text{min}_O\text{(min}_I\text{(Y),min}_I\text{(Z))} \\
\text{max(X)} &\equiv \text{max}_O\text{(max}_I\text{(Y),max}_I\text{(Z)).}
\end{aligned}$$

However, there is another restriction for decomposability. The partitions Y and Z must contain disjoint values. To see why this must be the case, consider the following example: `COUNT(DISTINCT(`$\{$`1,1,1,2,2,2`$\}$`))` $= 2$, but some partitionings of the bag yield the wrong result: `COUNT(DISTINCT(`$\{$`1,1,2`$\}$`))` + `COUNT(DISTINCT(`$\{$`1,2,2`$\}$`))` $= 2 + 2 \neq 2$.

The discussion of our equivalences (see Fig. 5.11) is split into two parts. In the first part, we discuss unnesting equivalences for queries with disjunctive linking. In the second part, we advance to unnesting equivalences for queries with disjunctive correlation. The proofs for all equivalences can be found in Appendix C.

Disjunctive Linking

In all the equivalences, let f be an aggregation function.

Equivalences 5.15 and 5.17 Equivalences 5.15 and 5.17 are used to unnest scalar queries whose linking predicate occurs disjunctively.

The former postpones the evaluation of the unnested subquery into the negative stream of a bypass selection. Basically, the unnesting technique is adapted from Equivalence 5.13 in the conjunctive case. Note that also the same restrictions hold, i.e. we require an equality comparison. The idea of this equivalence has already been explained using Query Q15. Fig. 5.9(c) depicts this strategy.

The latter equivalence is used for first evaluating the unnested subquery, i.e. the linking predicate, and postpone the evaluation of the second predicate into the negative stream of the bypass selection. Fig. 5.9(d) visualizes this strategy.

Equivalences 5.16 and 5.18 The above equivalences allow unnesting only for queries whose correlation predicate features an equality comparison. In Equivalences 5.16 and 5.18, the sequence of unary grouping and left outer-join is replaced by a binary grouping operator. The advantage of using binary grouping is that these equivalences support an arbitrary correlation predicate $\text{A}_2\theta_2\ \text{B}_2$ with $\theta \in \{=, \neq, <, \leq, >, \geq\}$. However, for their validity they require the correlation predicate A_2 to be a key of R. This is necessary to be able to preserve the other attributes of R, e.g. A_1 or a free attribute of p.

$$\sigma_{\mathrm{p}\vee\mathrm{A}_1\theta(\mathrm{f}(\sigma_{\mathrm{A}_2=\mathrm{B}_2}(\mathrm{S})))}(\mathrm{R}) \equiv e_1 \dot{\cup} e_2 \tag{5.15}$$
$$e_1 := \sigma_p^+(\mathrm{R})$$
$$e_2 := \Pi_{\mathcal{A}(R)}(\sigma_{\mathrm{g}\theta\mathrm{A}_1}((\sigma_p^-(\mathrm{R}))\ltimes^{\mathrm{g:f}(\emptyset)}_{\mathrm{A}_2=\mathrm{B}_2} (\Gamma_{\mathrm{g};=\mathrm{B}_2;\mathrm{f}}(\mathrm{S}))))$$
$$\Pi_{A_1,A_2}(\sigma_{p\vee\mathrm{A}_1\theta_1(\mathrm{f}(\sigma_{\mathrm{A}_2\theta_2\mathrm{B}_2}(\mathrm{S})))}(\mathrm{R})) \equiv \Pi_{A_1,A_2}(e_1 \dot{\cup} e_2) \tag{5.16}$$
$$e_1 := \sigma_p^+(\mathrm{R})$$
$$e_2 := \Pi_{\mathcal{A}(R)}(\sigma_{\mathrm{g}\theta_1\mathrm{A}_1}((\sigma_p^-(\mathrm{R}))\ltimes\Gamma_{\mathrm{g};\mathrm{A}_2\theta_2\mathrm{B}_2;\mathrm{f}}(\mathrm{S})))$$
$$\sigma_{\mathrm{p}\vee\mathrm{A}_1\theta(\mathrm{f}(\sigma_{\mathrm{A}_2=\mathrm{B}_2}(\mathrm{S})))}(\mathrm{R}) \equiv \Pi_{\mathcal{A}(R)}(e_1 \dot{\cup} e_2) \tag{5.17}$$
$$e_1 := \sigma^+_{\mathrm{g}\theta\mathrm{A}_1}((\mathrm{R})\ltimes^{\mathrm{g:f}(\emptyset)}_{\mathrm{A}_2=\mathrm{B}_2}(\Gamma_{\mathrm{g};=\mathrm{B}_2;\mathrm{f}}(\mathrm{S})))$$
$$e_2 := \sigma_p(\sigma^-_{\mathrm{g}\theta\mathrm{A}_1}((\mathrm{R})\ltimes^{\mathrm{g:f}(\emptyset)}_{\mathrm{A}_2=\mathrm{B}_2}(\Gamma_{\mathrm{g};=\mathrm{B}_2;\mathrm{f}}(\mathrm{S}))))$$
$$\Pi_{A_1,A_2}(\sigma_{\mathrm{p}\vee\mathrm{A}_1\theta_1(\mathrm{f}(\sigma_{\mathrm{A}_2\theta_2\mathrm{B}_2}(\mathrm{S})))}(\mathrm{R})) \equiv \Pi_{A_1,A_2}(e_1 \dot{\cup} e_2) \tag{5.18}$$
$$e_1 := \sigma^+_{\mathrm{g}\theta_1\mathrm{A}_1}((\mathrm{R})\ltimes\Gamma_{\mathrm{g};\mathrm{A}_2\theta_2\mathrm{B}_2;\mathrm{f}}(\mathrm{S}))$$
$$e_2 := \sigma_p(\sigma^-_{\mathrm{g}\theta_1\mathrm{A}_1}((\mathrm{R})\ltimes\Gamma_{\mathrm{g};\mathrm{A}_2\theta_2\mathrm{B}_2;\mathrm{f}}(\mathrm{S})))$$
$$\sigma_{\mathrm{A}_1\theta\mathrm{f}(\sigma_{\mathrm{A}_2=\mathrm{B}_2\vee p}(\mathrm{S}))}(\mathrm{R}) \equiv \Pi_{\mathcal{A}(R)}(\sigma_{\mathrm{A}_1\theta\mathrm{g}}(\chi_{\mathrm{g}:\mathrm{f}_O(\mathrm{g}_1,\mathrm{e}_2)}(\mathrm{e}_1))) \tag{5.19}$$
$$e_1 := R\ltimes^{\mathrm{g}_1:\mathrm{f}_I(\emptyset)}_{\mathrm{A}_2=\mathrm{B}_2}(\Gamma_{\mathrm{g}_1;=\mathrm{B}_2;\mathrm{f}_I}(\sigma_p^-(\mathrm{S})))$$
$$e_2 := \mathrm{f}_I(\sigma_p^+(\mathrm{S}))$$
$$\Pi_{A_1,A_2}(\sigma_{A_1\theta_1 f(\sigma_{A_2\theta_2 B_2\vee p}(S))}(R)) = \Pi_{A_1,A_2}(\sigma_{A_1\theta_1 g}(\chi_{g:f_O(g_1,e_2)}(e_1))) \tag{5.20}$$
$$e_1 := (R)\ltimes\Gamma_{g_1;A_2\theta_2 B_2;f_I}(\sigma_p^-(S))$$
$$e_2 := f_I(\sigma_p^+(S))$$
$$\sigma_{\mathrm{A}_1\theta\mathrm{f}(\sigma_{\mathrm{A}_2=\mathrm{B}_2\vee p}(\mathrm{S}))}(\mathrm{R}) \equiv \Pi_{\mathcal{A}(R)}(\sigma_{\mathrm{A}_1\theta\mathrm{g}}((R')\ltimes\Gamma_{\mathrm{g};t1=t1';\mathrm{f}} (\rho_{\mathrm{t}1'\leftarrow\mathrm{t}1}(e_1 \dot{\cup} e_2)))) \tag{5.21}$$
$$R' := \nu_{t1}(\mathrm{R})$$
$$e_1 := R'\ltimes^+_{\mathrm{A}_2=\mathrm{B}_2}\mathrm{S}$$
$$e_2 := \sigma_p(R'\ltimes^-_{\mathrm{A}_2=\mathrm{B}_2}\mathrm{S})$$

Figure 5.11: Equivalences for disjunctive *JA* queries

Disjunctive Correlation

Equivalences 5.19 and 5.20 handle queries whose correlation predicate occurs in a disjunction. Their limitation is that the predicate expression p must not be a subquery itself. Moreover, these equivalences require the aggregation function to be

decomposable and the correlation predicate to be an equality predicate[1]. Fig. 5.10 illustrates the idea of Equivalence 5.19 for the query from Section 5.3.2.

For both equivalences, the main idea is to generate partial, intermediate results, which are then combined by the subsequent map operator. The first partial result consists of those tuples that qualify for the non-correlation predicate. The other part of the result contains those tuples that satisfy the correlation predicate. These tuples are either checked by a sequence of unary grouping and left outer-join operator (see Equivalence 5.19) or a binary grouping operator (see Equivalence 5.20). For the former, the correlation predicate must exhibit an equality comparison. For the latter, the correlation predicate may use an arbitrary correlation predicate (using $\theta \in \{=, \neq, <, \leq, >, \geq\}$) but must hold the restriction that A_2 must be a super key of R.

Equivalence 5.21 in contrast, is more generally applicable. There are no restrictions on the aggregate function. In addition, predicate p may contain a nested query, i.e. the query is a linear query. The bypass join generates one positive stream for the tuples which satisfy the correlation predicate and a complementary negative one where p is checked. Beforehand, we need to introduce a numbering operator ν, which enables us to correctly reassemble the results during the binary grouping.

5.3.4 Completeness of Equivalences

Our equivalences handle all cases of scalar subqueries with disjunctive linking and correlation. Thereby, the linking predicate can consist of an arbitrary linking operator ($\{=, \neq, <, \leq, >, \geq\}$).

Let us make sure that the canonical translation of a scalar subquery always leads to a pattern that matches the left-hand side of one of our equivalences. In this situation, the canonical translation results in an aggregate function call f as top-level member of a selection predicate, which is part of the linking predicate.

In Equivalence 5.15 and 5.17, this corresponds to disjunctive linking. The argument of the aggregation function is again a selection checking for the correlation predicate, which in Equivalences 5.19 and 5.21 occurs in a disjunction. Remember that the former equivalence is a special case of the latter one, where p must not be a subquery itself and the aggregation function f must be decomposable.

5.3.5 Tree Queries

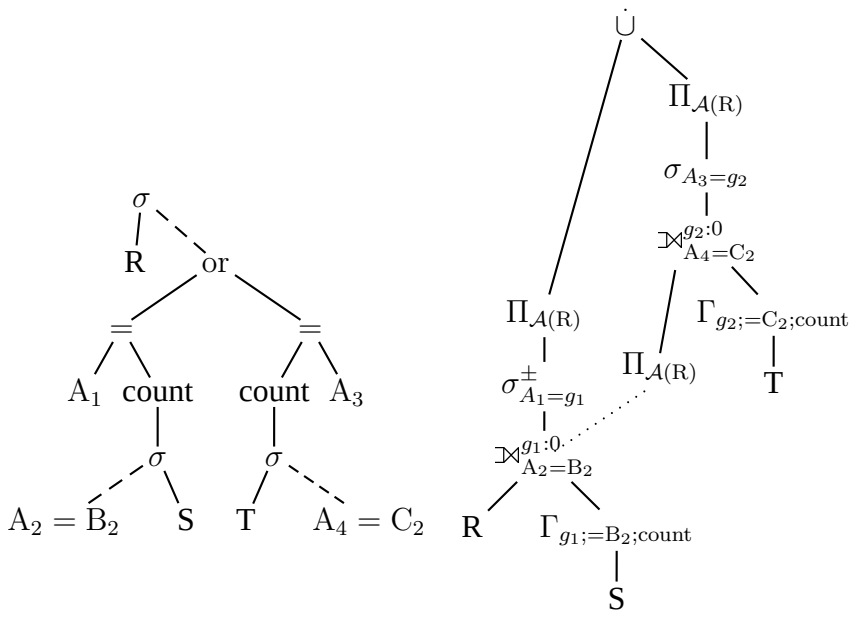
Tree queries of type *JA* can be unnested quite easily by successive applications of our known equivalences. Consider the following tree query:

[1]Note that the DISTINCT versions of the aggregation functions `COUNT`, `SUM`, and `AVG` are not decomposable. In this case, Equivalence 5.21 must be applied.

```
SELECT  DISTINCT *
FROM    R
WHERE   A1 =(SELECT COUNT(DISTINCT *)
             FROM    S
             WHERE   A2 = B2)
        OR
        A3 = (SELECT COUNT(DISTINCT *)
              FROM T
              WHERE A4 = C2)
```

Q17

Figure 5.12 illustrates the canonical translation and the result of the following two steps. In a first step, we unnest the query. For this, we apply Equivalence 5.15 to the predicate with the lowest rank. In the second step, we have to choose: either we apply Equivalence 5.15 again, if there exists another subquery on the same level, or we apply Equivalence 5.13, if this is not the case. Because none of the subqueries contains a nested query, we then apply Equivalence 5.13.



(a) Canonical (b) Unnested bypass plan

Figure 5.12: Unnesting strategy for Q17 (sketch)

We can also replace Equivalences 5.13 with 5.14 or Equivalence 5.15 with 5.16, respectively.

5.3.6 Linear Queries

Linear JA queries are a special case of disjunctive correlation. The second predicate in the disjunction is again a linking predicate, as shown in the following query:

```
SELECT  DISTINCT *
FROM    R
WHERE   A1 =(SELECT COUNT(DISTINCT *)
             FROM    S
             WHERE   A2 = B2
             OR
             B3 = (SELECT COUNT(DISTINCT *)
                   FROM T
                   WHERE B4 = C2))
```

Q18

Fig. 5.13 presents the unnesting procedure. We start with the canonical translation (see Fig. 5.13(a)) and unnest in a top-down fashion. In a first step, we apply Equivalence 5.21. The result is shown in Fig. 5.13(b). From here, it becomes obvious that for the deepest nested expression Equivalence 5.13 can be applied which yields the final plan shown in Fig. 5.13(c).

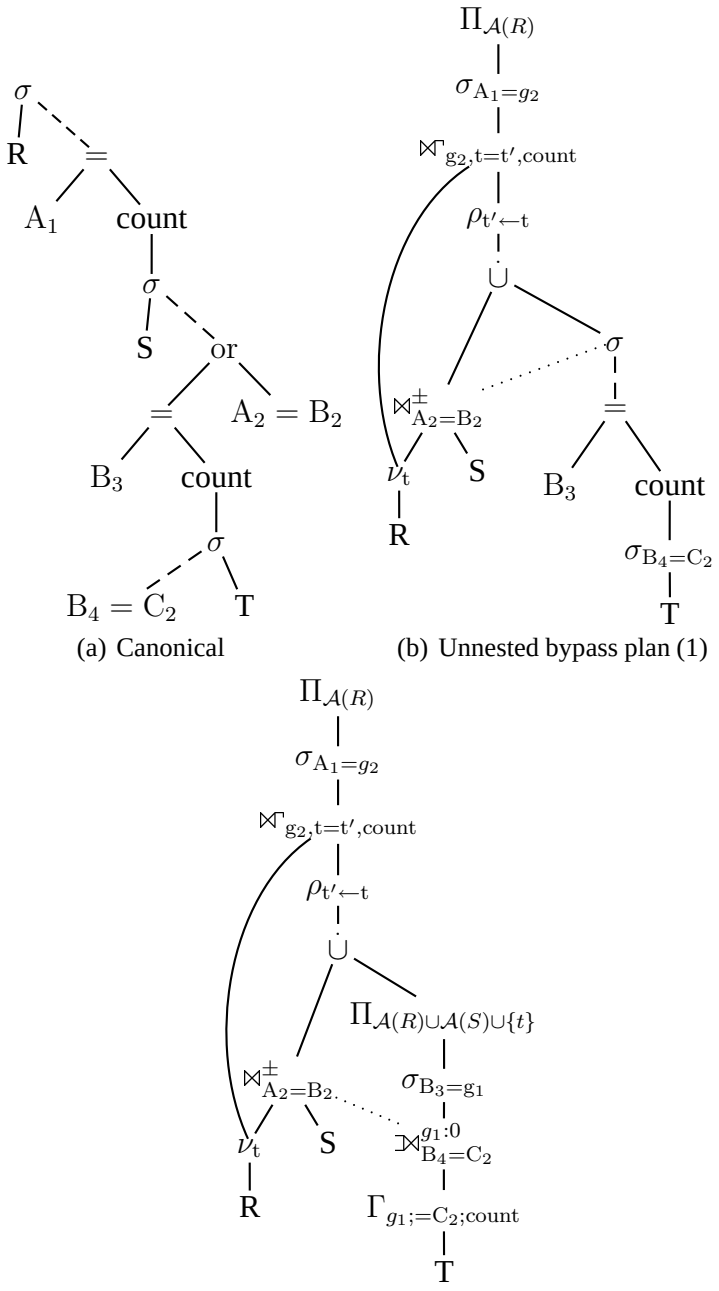
5.3.7 Duplicate Handling

Let us make sure that all equivalences mentioned in this section are also correct when they are based on an algebra over multisets. The right-hand side of Equivalences 5.15, 5.17, and 5.19 contains a unary grouping of the input of the nested query block, followed by a left outer-join. We observe that after grouping on the correlation attribute of the inner query, each value of the grouping attributes occurs exactly once. This key is later used as join attribute in the left outer-join. As a result, this join either finds exactly one matching tuple for each tuple resulting from the outer query block, or it keeps the outer block's tuple in order to preserve empty groups. Hence, the cardinality of the left outer-join is exactly the one of the outer relation R.

In Equivalence 5.19, we have already ensured correctness of the duplicate semantic for expression e_1 above. The map operator does not influence duplicates, as it only computes the correct aggregate value.

The numbering operator ν in Equivalence 5.21 turns the multiset R into a set and thereby ensures correctness for multisets.

Each equivalence introduces one bypass operator. In the unnested plan, this operator partitions its input into two disjoint sets. Thus, it neither creates duplicates nor discards any tuples. Moreover, the final union can be defined for multisets, ensuring that duplicates are handled correctly.



(a) Canonical (b) Unnested bypass plan (1)

(c) Unnested bypass plan (2)

Figure 5.13: Unnesting strategy for Q18 (sketch)

5.4 Evaluation

To demonstrate the effectiveness of our unnesting techniques, we performed an extensive evaluation. Specifically, we measured the execution times of the canonical

and the unnested plans using our hybrid relational and XML DBMS Natix [40]. Additionally, we compared the resulting evaluation times with those measured for three major commercial database management systems, for anonymity reasons henceforth nicknamed S 1, S 2, and S 3. For the same reason, we cannot present specific query evaluation plans for the commercial systems. However, as we will see, the strategy can be predicted by comparing these evaluation times with those resulting from the execution of the canonical plans in Natix.

After a brief description of the experimental setup, we first present performance results for unnesting table subqueries and then proceed to scalar subqueries. In both cases, we present evaluation results for simple, tree, and linear queries.

5.4.1 Datasets

The evaluation is performed on several datasets based on three schemas:

1. the schema of the TPC-H benchmark [104],
2. the schema of the preliminary draft of the TPC-DS benchmark [105], and
3. the schema RST used for the example queries.

The latter schema contains three tables (R, S, and T), each consisting of four columns $\mathrm{A}_i \in \mathcal{A}(\mathrm{R})$, $\mathrm{B}_i \in \mathcal{A}(\mathrm{S})$, and $\mathrm{C}_i \in \mathcal{A}(\mathrm{T})$ for $i = 1 \ldots 4$.

The datasets for the TPC-H benchmark are generated using the benchmark generator (dbgen) with scaling factors (SF) 0.01, 0.05, 0.5, 1, 5, and 10. This results in moderate database sizes of 11MB - 11GB.

For the TPC-DS benchmark, we generated the qualification database using the benchmark generator dbgen2. The resulting database has a size of 1GB.

For the independently scaled relations of the RST schema, we generated instances with scaling factors (SF) 1, 5, and 10. This led to 10.000, 50.000, and 100.000 rows and amounts to small tables of 178KB, 1.1MB, and 2.1MB. In the evaluations, $SF1$ denotes the scaling factor of the outer query block and $SF2$ the scaling factor of the inner query block. We did not use larger scaling factors because neither the canonical plans nor the commercial systems scaled well.

5.4.2 Settings

For the experiments, we used two comparable PCs with 1 GB of RAM each. Not all commercial systems are available for the same operating system. We executed Natix and two of the commercial systems on one of the PCs running Linux 2.6.11. The other commercial system ran on the other PC under Windows XP. All queries were executed with a cold buffer. Further, for optimizing the queries using the commercial systems, we used the highest optimization level possible. However, we did not create any indexes.

Because of the necessity to use two different systems, the resulting evaluation times are not exactly comparable. However, the growth of the resulting evaluation times already demonstrates the effectiveness of our unnesting approaches.

5.4.3 Table Subqueries

Disjunctive Linking

First, we present a performance evaluation for Query Q10 on our synthetic schema and the Query 4d on the TPC-H schema. Query 4d is similar to Query 4 from the TPC-H benchmark but extended to also select urgent orders. The predicate selecting the urgent orders is disjunctively connected to the linking predicate of the nested (correlated) query. The query is shown in the following.

```
SELECT  o_orderpriority, count(*) as order_count
FROM    orders
WHERE   o_orderpriority = '1-URGENT'
        OR EXISTS (
          SELECT *
          FROM  lineitem
          WHERE l_commitdate < l_receiptdate
          AND   o_orderkey = l_orderkey)
GROUP BY o_orderpriority ORDER BY o_orderpriority
```

We executed both queries in the three commercial systems and in our database system Natix. For the evaluation with Natix, we generated three alternative evaluation plans for both queries. For Query Q10 these are:

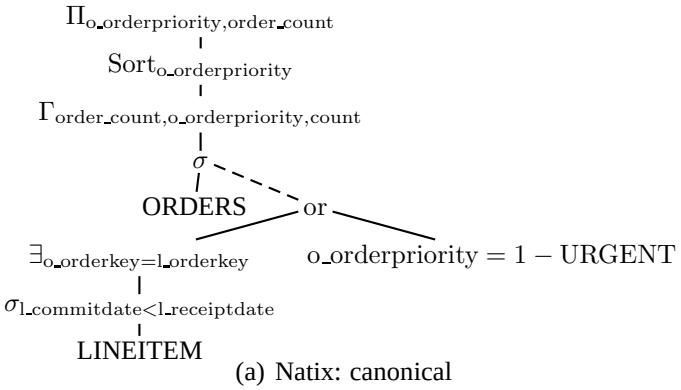
Canonical: A correlated plan, as depicted in Fig. 5.2(a). Clearly, this plan performs a nested-loop like evaluation.

Semi-join: An unnested plan, as shown in Fig. 5.2(b) using a nested-loop semi-join implementation.

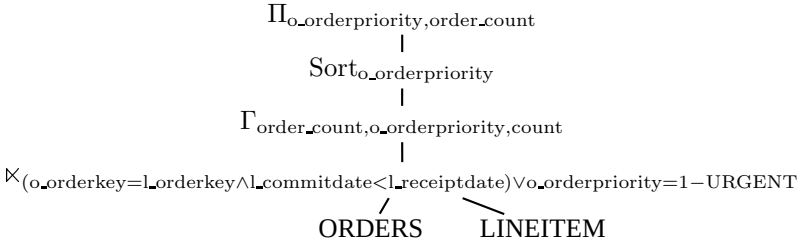
Unnested: An unnested bypass plan, as given in Fig. 5.2(d).

For Query 4d we generated the same alternatives. Fig. 5.14 depicts their plan sketches.

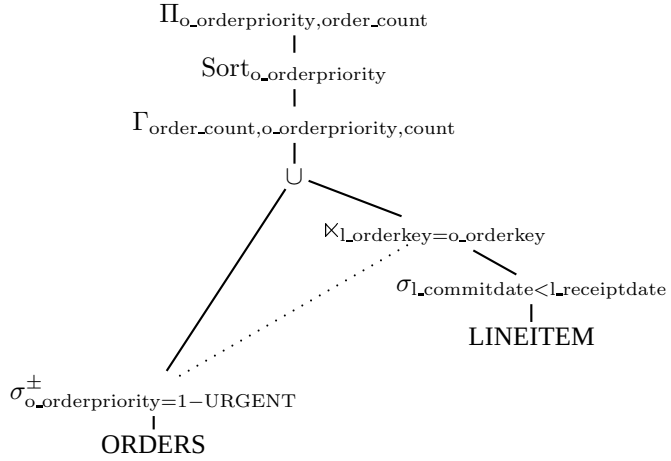
The first plan (see Fig. 5.14(a)) implements a nested-loop strategy. In this plan, the correlated subquery is executed for every tuple from the ORDERS relation that does not qualify for the predicate checking for an urgent orderpriority. The second plan (see Fig. 5.14(b)) features a semi-join operator for the unnested evaluation of the subquery and the predicate connected to the subquery using a disjunction. For this reason, the implementation of the semi-join performs a nested-loop strategy.



(a) Natix: canonical



(b) Natix: semi-join



(c) Natix: unnested

Figure 5.14: Query plan sketches for Query 4d

Efficient hash- or sort-based implementations are out of reach in this case. The third and last plan (see Fig. 5.14(c)) exhibits a bypass selection for the evaluation

of the predicate checking for urgent priorities. In the false stream of the bypass selection, a hash-based semi-join is used for the evaluation of the subquery. The remaining operators (i.e. grouping, sort, and projection operator) are the same in all three plans.

SF1	1			5			10		
SF2	1	5	10	1	5	10	1	5	10
System									
S 1	10.1	51.3	102	50.6	260	520	100	522	1043
S 2	**0.21**	0.28	**0.17**	0.86	0.83	0.89	1.75	1.84	1.86
S 3	7.78	41.4	83.3	33.8	175	363	66.1	342	663
Natix									
• canonical	10.8	53.1	104	45.5	228	452	86.2	431	852
• semi-join	4.0	4.05	4.0	4.01	4.03	3.96	4.01	4.02	3.97
• unnested	**0.21**	**0.2**	0.2	**0.21**	**0.2**	**0.2**	**0.21**	**0.2**	**0.23**

(a) Q10

	TPC-H Scaling Factor					
System/Technique	0.01	0.05	0.5	1	5	10
S 1	84.0	3715	n/a	n/a	n/a	n/a
S 2	**0.08**	1.83	24.7	43.9	290	616
S 3	62.8	1742	n/a	n/a	n/a	n/a
Natix						
• canonical	79.7	3631	n/a	n/a	n/a	n/a
• semi-join	17.7	470	n/a	n/a	n/a	n/a
• unnested	0.19	**0.48**	**3.67**	**15.6**	**79.3**	**189**

(b) Query 4d

Figure 5.15: Results (in sec.) for Q10 and 4d

Fig. 5.15 shows the execution times (in seconds) of these queries. We aborted the execution of queries after six hours. These cases are denoted by n/a. The fastes execution time for each dataset is denoted using a bold face. The first table (see 5.15(a)) compare the runtimes for Query Q10 executed on the commercial systems and Natix. The bottom table (see 5.15(b)) presents the results for Query 4d.

For both queries, our unnested approach outperforms even the fastest commercial system S 2. For the largest scaling factors, our approach outperforms S 2 by a factor of three or more. The remaining commercial systems show a performance similar to our canonical plan. These results indicate that they perform a rather naïve evaluation. The unacceptable evaluation times of these systems and the canonical plan underline the importance of unnesting nested queries. In general, the unnested plans of the two example queries finish up to four orders of magni-

tude faster than the naïve nested-loop evaluation. Morever, even the Natix plan that exhibits a nested-loop semi-join for unnesting Query 4d does not produce a result within six hours for the dataset with scaling factor 0.5.

Disjunctive Correlation

Besides the evaluation of type *J* queries with disjunctive linking, we also performed an evaluation for queries with disjunctive correlation. Therefore, we executed Query Q11 on our synthetic dataset and generated two alternative query execution plans for Natix. The first alternative is based on the canonical translation (see Fig. 5.3(a)), and the second is the unnested plan (see Fig. 5.3(b)). Fig. 5.16 presents the results of this study.

SF1	1			5			10		
SF2	1	5	10	1	5	10	1	5	10
System									
S 1	**0.06**	**0.17**	**0.24**	**0.08**	**0.35**	**0.56**	**0.11**	**0.38**	**0.70**
S 2	0.13	0.27	0.38	0.11	0.61	0.95	0.13	0.63	1.23
S 3	8.78	35.9	87.1	46.9	220	400	94.7	469	885
Natix									
• canonical	11.9	48.3	72.9	66.3	278	661	163	633	1348
• unnested	0.22	0.39	0.54	0.26	0.69	1.02	0.31	0.75	1.27

Figure 5.16: Results (in sec.) for Q11

In the experiments, systems S 1 and S 2 perform as fast as our unnested plan. However, in contrast to the acceleration of nested queries, unnesting gives the cost-based optimizer more opportunities for reordering operators and the selection of physical implementations. Finally, we point out the enormous performance gains, compared to the naïve nested-loop evaluation chosen by system S 3 and to our canonical plan.

Tree Table Subqueries

To evaluate queries of type *J* with a tree structure, we executed two queries. The first is based on the our synthetic dataset and shown in the following.

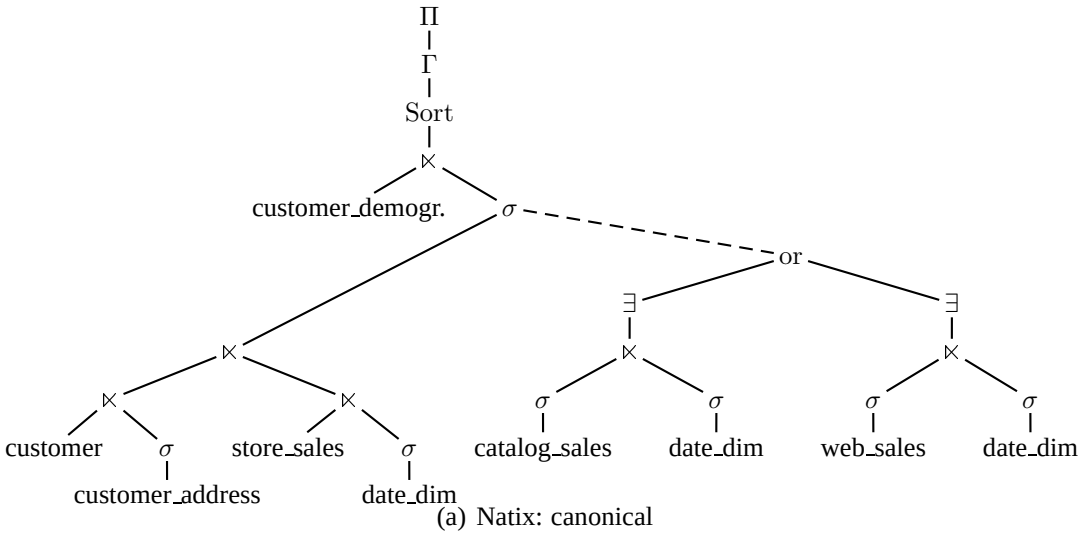
```
SELECT DISTINCT *
FROM R
WHERE A1 IN (SELECT B4
             FROM S
             WHERE A2 = B3)                    Q19
      OR
      A1 IN (SELECT C4
             FROM T
             WHERE A2 = C3)
```

Query Q19 is a tree query because it has two nested query blocks nested inside the top-level query block.

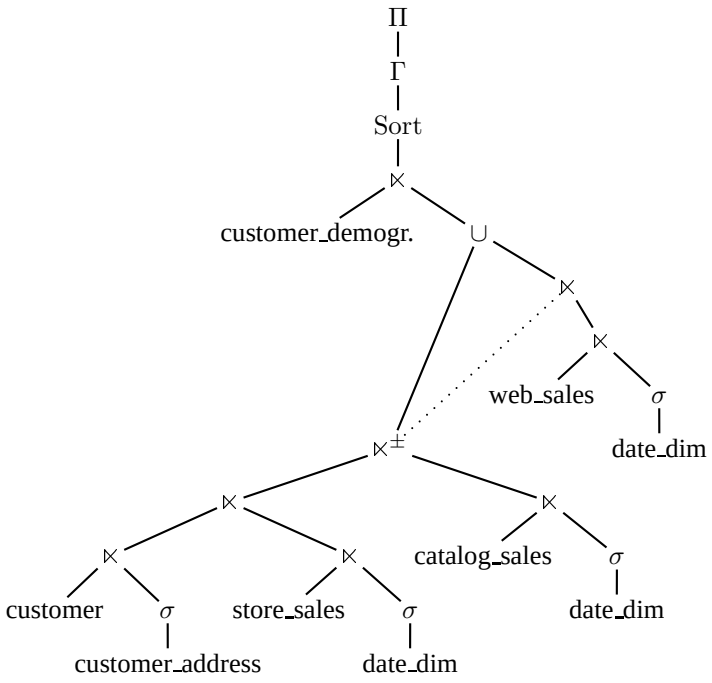
The second is Query 10 taken from the TPC-DS benchmark and entirely shown here.

```
SELECT cd_gender, cd_marital_status, cd_education_status,
       count(*), cd_purchase_estimate, count(*), cd_credit_rating,
       count(*), cd_dep_count, count(*), cd_dep_employed_count,
       count(*), cd_dep_college_count, count(*)
FROM   customer c,customer_address ca,customer_demographics
WHERE  c.c_current_addr_sk = ca.ca_address_sk and
       ca_county in ('Rush County','Toole County',
       'Jefferson County', 'Dona Ana County','La Porte County')
       and cd_demo_sk = c.c_current_cdemo_sk and
       EXISTS (SELECT *
                      FROM   store_sales,date_dim
                      WHERE  c.c_customer_sk = ss_customer_sk and
                             ss_sold_date_sk = d_date_sk and
                             d_year = 2000 and
                             d_moy between 3 and 3+3) and
       (EXISTS (SELECT *
                      FROM   web_sales,date_dim
                      WHERE  c.c_customer_sk = ws_bill_customer_sk
                             and ws_sold_date_sk = d_date_sk and
                             d_year = 2000 and
                             d_moy between 3 ANd 3+3) or
       EXISTS (SELECT *
                      FROM   catalog_sales,date_dim
                      WHERE  c.c_customer_sk = cs_ship_customer_sk
                             and cs_sold_date_sk = d_date_sk and
                             d_year = 2000 and
                             d_moy between 3 and 3+3))
GROUP BY  cd_gender, cd_marital_status, cd_education_status,
          cd_purchase_estimate, cd_credit_rating, cd_dep_count,
          cd_dep_employed_count, cd_dep_college_count
ORDER BY  cd_gender, cd_marital_status, cd_education_status,
          cd_purchase_estimate, cd_credit_rating, cd_dep_count,
          cd_dep_employed_count, cd_dep_college_count
```

The TPC-DS query contains three nested correlated queries, two of which are connected by a disjunction. For the (almost) canonical plan (see Fig. 5.17(a) for a sketch), we unnested the subquery that occurs conjunctively with the help of a semi-join. The remaining two subqueries that are connected by a disjunction are evaluated in the subscript of the selection. The grouping operator utilizes the sorted grouping attributes. The unnested bypass plan (see Fig. 5.17(b) for a sketch) exploits a bypass semi-join for the evaluation of the subquery that joining catalog_sales and date_dim. The evaluation of the other subquery is postponed into the false stream.



(a) Natix: canonical



(b) Natix: unnested

Figure 5.17: Query plan sketches for TPC-DS Query 10

Fig. 5.18 contains the results of our evaluation. For the synthetic dataset (see Subfigure 5.18(a)) SF1 denotes the scaling factor for the outer relation. SF2 denotes the scaling factor of both inner relations, i.e. S and T. Subfigure 5.18(b) presents the

results for the evaluation of TPC-DS Query 10 on the qualification database of the TPC-DS benchmark.

SF1	1			5			10		
SF2	1	5	10	1	5	10	1	5	10
System									
S 1	23.3	119	305	114	598	1524	231	1199	3008
S 2	0.34	**0.14**	**0.16**	**0.13**	**0.19**	**0.27**	**0.16**	**0.22**	**0.34**
S 3	17.2	91.9	179	84.3	496	920	166	944	1873
Natix									
• canonical	26.8	133	273	127	695	1323	266	1291	3003
• unnested	**0.23**	0.68	2.04	0.27	1.23	3.21	0.41	1.97	4.66

(a) Q19

	Qualification Database (1GB)
S 1	7304
S 2	228
S 3	537
Natix	
• canonical	82.9
• unnested	14.7

(b) TPC-DS Query 10

Figure 5.18: Results (in sec.) for Q19 and TPC-DS Query 10

For the synthetic dataset, S 2 is the fastest of all systems. We also note, that in the case of tree queries S 3 (which was the slowest before) is faster than S 1, for both, the synthetic and the TPC-DS dataset. However, on the TPC-DS benchmark, our unnested approach is the fastest, i.e. an order of magnitude faster than S 2. On this dataset, our canonical approach, which executes the conjunctively connected subquery in an unnested manner, is faster than S 2.

Linear Table Subqueries

Within the last table subquery experiment, we measured the performance gains that can be achieved by unnesting linear queries. Consider the following linear query that contains two subqueries of type *J*.

```
SELECT *
FROM R
WHERE R.A1 IN (SELECT S.A4
        FROM S
        WHERE R.A2 = S.A3
        OR S.A1 IN (SELECT T.A4
                    FROM T
                    WHERE S.A2 = T.A3))
```

Q20

Fig. 5.19 contains the according results of its evaluation. In the table, SF1 denotes the factor used to scale the outer relation. SF2 denotes the scaling factor of both inner relations, i.e. S and T.

SF1	1			5			10		
SF2	1	5	10	1	5	10	1	5	10
System									
S 1	8.71	191	902	8.36	497	1515	8.35	818	1995
S 2	0.19	0.75	1.64	0.31	0.91	1.73	0.53	1.14	1.94
S 3	25.6	418	1376	85.6	742	1974	149	1060	2925
Natix									
• canonical	24.2	348	1347	79.6	6574	1891	140	1057	2519
• unnested	**0.17**	**0.37**	**0.74**	**0.24**	**0.43**	**0.81**	**0.36**	**0.52**	**0.89**

Figure 5.19: Results (in sec.) for Q20

For linear table queries, our unnested approach dominates all other approaches. The execution times of S 1 and S 3, compared to the execution times of our naïve evaluation plan, indicate that the commercial systems use a nested-loop like evaluation. However, we also note that S 2 also performs very well.

5.4.4 Scalar Subqueries

In the following section, we present the results for our performance study for scalar subqueries. Similar to the last section, we start with presenting our evaluation for simple queries — with disjunctive linking and correlation — and then move on to queries that have a tree and linear structure.

Disjunctive Linking

We selected Query Q15, and based on the TPC-H schema, the introductory Query 2d to evaluate simple queries with disjunctive linking.

Query Q15 has one nested correlated query that is disjunctively connected to the outer query block. The same yields for Query 2d, which we depicted in the introduction of this chapter. For both queries, we executed two query execution plans in Natix. The first plan implements a canonical translation. Figures 5.9(a)

and 5.21(a) show the canonical plan for Query Q15 and 2d, respectively. The second plan results from the application of Equivalence 5.15. It unnests the type *JA* subquery using a binary grouping operator in the false stream of a bypass selection. Figures 5.9(c) and 5.21(b) illustrate these strategies for Query Q15 and 2d, respectively.

SF1	1			5			10		
SF2	1	5	10	1	5	10	1	5	10
System									
S 1	10.6	55.7	111	49.4	259	520	98.3	515	1029
S 2	**0.19**	0.33	0.52	0.92	1.17	1.30	1.95	2.13	2.52
S 3	5.06	25.1	50.1	25.7	144	267	49.8	259	558
Natix									
• canonical	10.9	54.9	109	46.8	235	474	88.5	450	899
• unnested	**0.2**	**0.24**	**0.3**	**0.78**	**0.87**	**0.98**	**1.6**	**1.65**	**1.74**

(a) Q15

	TPC-H Scaling Factor (SF)					
	0.01	0.05	0.5	1	5	10
System						
S 1	0.14	0.36	52.5	123	n/a	n/a
S 2	**0.10**	2.00	29.0	67.0	328	766
S 3	0.27	0.57	48.7	234	n/a	n/a
Natix						
• canonical	79.7	3631	n/a	n/a	n/a	n/a
• unnested	0.14	**0.19**	**0.82**	**1.49**	**23.1**	**49.5**

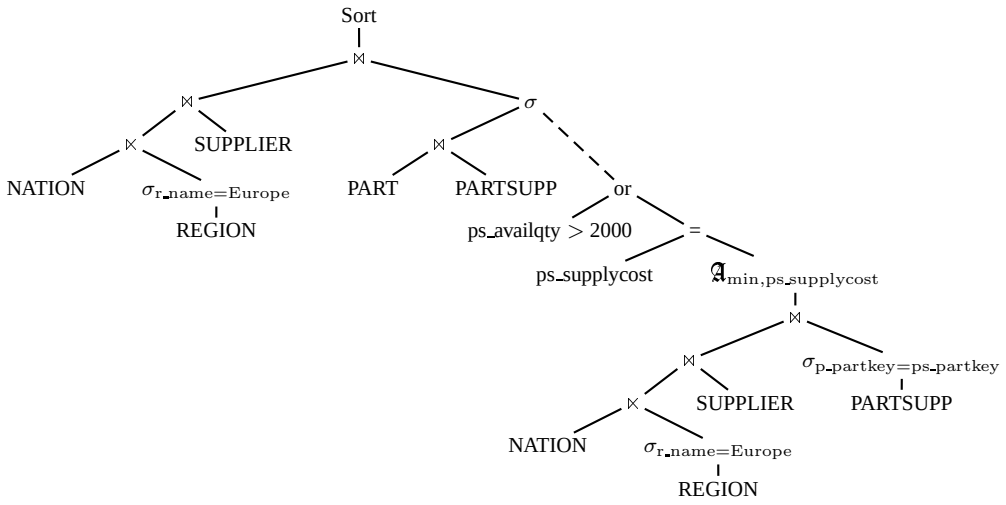
(b) Query 2d

Figure 5.20: Results (in sec.) for Q15 and 2d

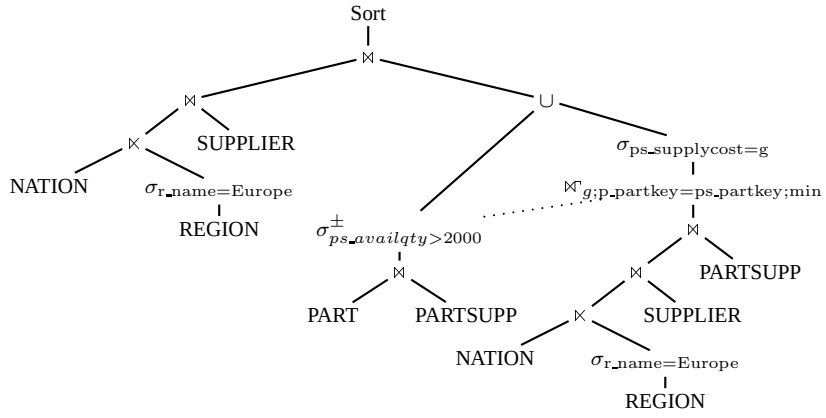
Fig. 5.20(a) and 5.20(b) present the results for these queries. We observe that our unnested approach excels all other approaches — for the RST as well as the TPC-H dataset. In comparison with our canonical approach, the performance numbers of the commercial systems for the RST dataset allow to deduce that these systems execute the nested query in a nested-loop like fashion. Only the commercial system S 2 almost keeps up with our unnested approach. However, for the TPC-H dataset our unnested approach even outperforms this system by one order of magnitude. The remaining commercial systems are surpassed by three to four orders of magnitude for the cases that finished within six hours.

Disjunctive Correlation

Besides the evaluation of JA queries with disjunctive linking, we performed an evaluation for queries with disjunctive correlation. Therefore, we executed Query Q16



(a) Natix: canonical



(b) Natix: unnested

Figure 5.21: Query plan sketches for Query 2d

using the commercial systems on our synthetic dataset and generated two alternative query execution plans for Natix. The first alternative is based on a canonical translation (see Fig. 5.10(a)). The second was derived by applying a strategy based on Equivalence 5.19 (see Fig. 5.10(b)).

Figure 5.22 presents our performance measurements for these plans. The assessments indicate that all commercial systems evaluate this query similarly to our

SF1	1			5			10		
SF2	1	5	10	1	5	10	1	5	10
System									
S 1	16.7	90.3	184	82.7	445	905	165	892	1803
S 2	8.55	46.3	95.5	42.9	235	479	85.7	466	971
S 3	11.6	59.7	120	71.4	378	737	143	753	1519
Natix									
• canonical	16.0	98.6	208	79.8	470	897	166	1237	1768
• unnested	**0.12**	**0.14**	**0.15**	**0.22**	**0.24**	**0.26**	**0.38**	**0.41**	**0.42**

Figure 5.22: Results (in sec.) for Q16

canonical plan. For the moderate size of 2.1MB of the largest synthetic dataset — scaling factor 10 for both the inner and outer query block —, our unnested approach outperforms the others by three to four orders of magnitude. Moreover, evaluation times up to half an hour for 2.1MB data seem unacceptable to us.

Tree Scalar Subqueries

To evaluate tree queries, we executed the following query on the synthetic RST dataset. Fig. 5.23 presents the results.

```
SELECT *
FROM R
WHERE A1 = (SELECT COUNT(*)
                FROM S
                WHERE A2 = B3)                                  Q21
        OR
        A2 = (SELECT COUNT(*)
                FROM T
                WHERE A4 = C4)
```

SF1	1			5			10		
SF2	1	5	10	1	5	10	1	5	10
System									
S 1	31.8	190	539	225	1039	2435	395	1976	4874
S 2	**0.14**	0.47	0.92	**0.36**	0.72	1.17	**0.66**	1.02	1.58
S 3	25.6	136	274	147	717	1391	289	1446	2874
Natix									
• canonical	25.8	136	262	136	689	1342	257	1384	2693
• unnested	0.19	**0.29**	**0.45**	0.4	**0.58**	**0.79**	0.81	**0.96**	**1.23**

Figure 5.23: Results (in sec.) for Q21

Similar to the results for queries of type *JA* with disjunctive linking, our unnested approach is the fastest. However, S 2 can almost keep up with us. The other commercial systems perform a naïve evaluation strategy.

Linear Scalar Subqueries

In the last subsection, we show an evaluation for linear queries whose subqueries are all of type *JA*. For this purpose, consider the following linear query.

```
SELECT *
FROM R
WHERE R.A1 = (SELECT count(S.B4)
              FROM S
              WHERE R.A2 = S.B3
                    OR S.B1 = (SELECT count(T.C4)
                               FROM T
                               WHERE S.B2 = T.C3))
      OR R.A4 < 1500
```

Q22

This query has two scalar subqueries of type *JA* arranged in a linear structure. Fig. 5.24 shows the according results of its evaluation.

SF1	1			5			10		
SF2	1	5	10	1	5	10	1	5	10
System									
S 1	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
S 2	**152**	**784**	**1602**	**744**	**3884**	**9237**	**1517**	**7923**	n/a
S 3	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
Natix									
• canonical	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
• unnested	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a

Figure 5.24: Results (in sec.) for Q22

As we can see, the evaluation of this query is very hard for almost all systems. Even our unnested approach cannot produce a result within six hours. The reasons for this are that the bypass join is, in fact, equally costly as a cross product because it has to enumerate the same number of tuples. However, in this case, our equivalences still increase the search space for an cost-based optimizer and may, for example, But as the evaluation of S 2 shows, evaluation techniques as in [50] become more important in such cases.

5.5 Related Work

The optimization of nested queries has been researched successfully. We have already discussed the existing techniques for unnesting SQL, OQL, and XQuery in the last two chapters. For example, Kim [70] established the classification for nested SQL queries to which we adhered to in this chapter. Moreover, he also proposed the first strategies for unnesting nested queries. With this preliminary work, he opened a huge research field for unnesting SQL [32, 36, 43, 44, 69, 99], OQL [29, 39, 101, 102], and XQuery [81, 82]. In particular, all these approaches focused on merging two or more query blocks into one. However, none of the approaches considered nested queries whose linking or correlation predicate occurs in a disjunction. We presented a novel approach that is capable of merging query blocks in order to unnest nested queries even if one of these predicates occurs in a disjunction.

In almost all cases, our approach significantly outperforms other existing approaches in terms of execution time. Admittedly, not all of our unnested plans are superior to evaluating queries in a nested fashion. The reason is that unnesting strategies do not always result in better plans if, for example, the result of the outer query is small. However, in any case, our algebraic unnesting approach increases the search space of a cost-based optimizer.

For the cases in which unnesting does not increase performance or unnesting is not possible, Elhemali et al. [36] and Graefe [50] propose approaches for efficiently evaluating subqueries in their nested structure. This is, for example, achieved by exhibiting available indices, materialized views, techniques similar to magic decorrelation ([99]), or prefetching.

The former approach by Elhemali et al. also suggests a solution for dealing with subqueries and their linking predicate occurring in disjunctions. Their approach utilizes the translation of disjunctions into union expressions in order to execute each argument of a union efficiently. However, this may require evaluating the two common input expression twice, which we avoid with bypass operators.

Optimizations for queries containing disjunctions have been presented in [22, 27, 65]. Specifically, the bypass technique that we extend for unnesting was introduced in [27, 68]. Strategies for implementing bypass operators and query evaluation engines that support DAG-structured query plans are presented in [27, 87, 97].

5.6 Conclusion

We believe that nested queries containing disjunctive predicates have not yet attracted the attention they deserve. In our experimental study, we have shown that evaluating nested queries in a nested-loop-like fashion leads to an unacceptable performance. With our novel unnesting strategy, we are able to remedy this situation and to substantially improve query execution times. Although most runtime

systems and optimizers do not incorporate bypass plans, it is possible to transfer bypass plans into plans without bypass operators. This can, for example, be done by tagging every tuple whether it belongs to the positive or negative stream.

Based on a common terminology, a classification, and an algebra, we have presented unnesting equivalences containing bypass operators for the first time. We have shown how to unnest scalar subqueries whose linking or correlation predicate occurs in a disjunction. Furthermore, we have demonstrated the our equivalences are applicable to simple, linear, and queries. Our equivalences are also valid for bags, and, hence, can be applied in real world applications.

Bypass operators in combination with our novel unnesting strategy provide an efficient evaluation for nested scalar queries with disjunctive linking or correlation. Our comprehensive experimental study compares the nested and unnested approaches — using our hybrid relational and XML database system Natix — against three commercial database systems. Our optimized approach dominates almost all other approaches, most of them by several orders of magnitude.

Chapter 6

Beyond XPath

In the last two years, the importance of XPath as a standalone language decreased, disposing the field to the more powerful XQuery language [38]. At the same time, XQuery evaluators become more and more mature in terms of features and performance, and XQuery is being integrated into mainstream DBMS products as a native language [8, 77, 92].

Because XQuery processing research is still missing some fundamental tools to facilitate the development of industrial-strength XQuery optimizers, we are concerned with filling one of these gaps in this chapter. Specifically, we provide a rewrite toolkit that allows to reduce the number of query blocks in a query expression. This widens the search space for plan generators by making more information visible to a single run of the plan generation algorithm. Let us begin by stressing the importance of our goal:

Industrial-strength query optimizers proceed in a two-phase manner. In a first phase, the query is translated into an internal representation, and heuristical rewrite rules are applied to simplify and normalize the query. In a second phase, a plan generator enumerates alternative execution plans, determines their cost, and chooses the optimal plan. Alternative plans can differ in the access paths used for the basic input sets (e.g. whether to use an index or not), in the order in which the basic input sets are joined, and in the position of other operators, such as grouping or sorting.

However, efficient plan generation algorithms cannot take arbitrary query structures as input. Instead, the unit of plan generation is the *query block*. Depending on the design of the query compiler, a query block can be represented in a variety of ways, for example as a source language construct (SELECT FROM WHERE in SQL, or FLWOR in XQuery), as a node in an internal graph representation (such as the Query Graph Model QGM [94]), or as an algebraic expression. Some queries exhibit a nested structure, where a query block references subquery blocks. In such cases, the plan generator is called in a bottom-up fashion, generating plans for all subquery blocks before the surrounding query block is processed. It is easy to see that in such cases, the search space examined by the plan generator is limited, because only locally good solutions are computed. For globally optimal plans, it is

desirable to reduce the number of query blocks to have more information available in a single run of the plan generator, creating a larger search space of alternative plans. For this reason, in the first phase of optimization, queries are rewritten by merging as many query blocks as possible. This is state-of-the-art for SQL query processing (e.g. [32, 44, 99]), but not highly developed for XQuery.

For an industrial strength approach to XQuery optimization, such a rewriting step to merge query blocks is particularly necessary:

- In XQuery expressions in real applications, a nested query structure is the norm rather than an exception. This is due to a number of reasons, including the construction of hierarchical XML results, the absence of a grouping construct, the generation of queries using visual editors, and, last but not least, the inlining of (nonrecursive) XQuery functions that contain FLWOR expressions.

- XML query processing can benefit from holistic n-way joins [21] which perform single-pass tree-pattern matching instead of constructing results just using binary joins. The detection of tree patterns and the decision when to use regular joins and when to use pattern matching is a global decision during plan generation that requires access to as much of the query as possible.

An example for a highly nested query (inspired by XMark Query 3) is shown here:

```
let $auction := doc("auction.xml") return
  let $euro:=for $o in $auction/site/open_auctions/open_auction
             for $i in $auction/site/regions/europe/item/@id
             where $o/itemref/@item eq $i
             return $o
  for $a in $euro
  where zero-or-one($a/bidder[1]/increase/text()) * 2
        <= $a/bidder[last()]/increase/text()
  return
    for $p in $auction/site/people/person[profile/@income > 5000]
    for $w in $p/watches/watch
    where $a/@id = $w/@open_auction
    return <auction id="{$a/@id}">
             <increase first="{$a/bidder[1]/increase/text()}"
                       last="{$a/bidder[last()]/increase/text()}"/>
             <watched_by id="{$p/@id}"/>
           </auction>
```

The query body is constructed of four FLWOR expressions, three of which are nested inside other FLWORs. However, these are only the explicit FLWOR blocks. Depending on the compiler design, the number of nested query blocks may be even deeper. For example, with a plan generator that focuses on purely structural tree pattern matching, nested value-based predicates such as `profile/@income > 5000` may be separate query blocks.

Without further processing, such a query is optimized using several runs of the plan generation algorithm, where each plan for a FLWOR expression is used in the plan for the surrounding FLWOR. This separate optimization of subqueries impedes the discovery of good overall execution plans. This is demonstrated by our

example, in which there are two value-based joins, one joining the Open Auctions to the European Items, and one joining the Open Auctions to the Persons with an income higher than 5000. However, the join conditions in the `where` clauses are in different FLWORs, prohibiting the plan generator to see both of the joins and optimize their order. Join order optimization is a cornerstone of efficient relational query processing and just as important in XQuery processing [80].

As in many other cases, the nested structure of the query is not required to obtain the query result, but is used because this way the query is simpler to write. In fact, the whole query above can be formulated using a single FLWOR block. One alternative to do so is shown below, with the results of each processing step bound to a separate variable:

```
let $auction := doc("auction.xml"), $x32 := $auction/site
for $o in $x32, $x13 in $o/open_auctions, $a in $x13/open_auction
for $i in $x32, $x15 in $i/regions, $x16 in $x15/europe
for $x17 in $x16/item, $x18 in $x17/@id
let $x4 := $a/itemref, $x19 := $x4/@item
let $x33 := $a/bidder[1], $x34 := $x33/increase, $x35 := $x34/text()
let $x36 := $a/bidder[last()], $x37 := $x36/increase, $x38 := $x37/text()
let $x39 := $a/@id
for $p in $x32, $x20 in $p/people, $x21 in $x20/person
for $w in $x21/watches, $x22 in $w/watch
let $x8 := $x21/@id, $x10 := $x22/@open_auction
let $x13 := $x21/profile, $x27 := $x13/@income
for $x1 in  <auction id="{$x39}">
              <increase first="{$x35}"
                        last="{$x38}"/>
              <watched_by id="{$x8}"/>
            </auction>
where zero-or-one($x35) * 2 <= $x38 and $x39 = $x10
      and $x27 > 5000 and $x19 eq $x18
return $x1
```

While this form of the query is less readable and more difficult to write, it is easier to optimize because all the basic operations, intermediate results, input sets, and their dependencies are uniformly represented in a single, top-level FLWOR construct.

The goal of this chapter is to provide a toolkit for developers of XQuery evaluators to transform XQuery expressions into expressions with as few query blocks as possible. This toolkit takes the form of rewrite rules merging the inner and outer FLWOR expressions into single FLWORs. These unnesting rules are supplemented by some helpful normalization rewrites. We have chosen to present our rules using regular XQuery syntax because other representations (such as QGM or algebraic expressions) are less universal and would be more difficult to adapt to different evaluators. We do not use the XQuery Core sublanguage because it does not have a query block construct suitable for plan generation. It is, instead, inherently nested, even for quite simple XQuery expressions.

The remainder of this chapter is structured as follows: First, in Section 6.1, we give an overview of the rewrite toolkit. The main Sections 6.2, 6.3, and 6.4 present normalization and FLWOR merging rules, respectively. We then present a short evaluation (see Sec. 6.5), demonstrating the effect of our rules when generating

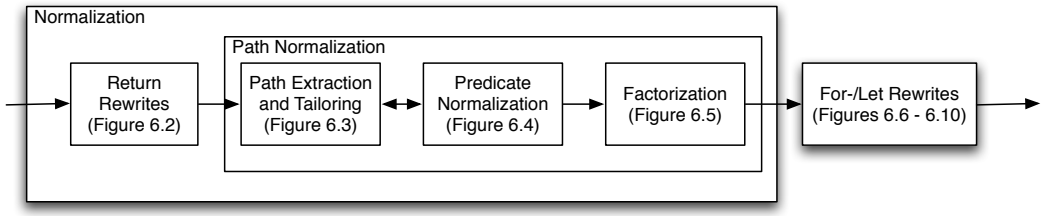


Figure 6.1: Processing model

execution plans. At the end, we discuss related work in Section 6.6 and conclude the chapter in Sec. 6.7.

6.1 Overview

The overall goal of this chapter is to flatten an XQuery expression, i.e. merge as many query blocks (i.e. FLWOR expressions) as possible. To achieve this goal, we basically proceed in two phases: (1) Normalization and (2) FLWOR Merging. Fig. 6.1 gives an overview of our processing model.

In both phases, we apply a set of rules based on XQuery syntax to a query. A separate figure presents one set of rules for each normalization and rewriting step. The overview Figure 6.1 contains references to each of them.

Normalization

comprises two major subtasks:

1. All `ExprSingle` expressions[1] from the `return` clause are moved to the expression creating the binding sequence of a new **for** expression.

2. Path expressions are normalized (as far as possible). In particular, (1) all path expressions not directly associated with a **for** clause are bound to variables using `let`, (2) path expressions are taken to single steps, (3) predicates are moved into the `where` clause, and (4) common location steps are factorized.

FLWOR Merging

Starting from this normalized form, we remove as many query blocks (FLWOR expressions) as possible. Specifically, we present rewrite rules that eliminate or merge inner FLWORs occurring in the **for** or `let` clause, respectively.

[1]Note that `ExprSingle` is the expression produced by the grammar rules from [38].

Notation

Our rewrite rules are formulated using XQuery syntax [38]. However, to simplify the presentation, we use the following abbreviations for frequently used clauses:

$$\texttt{ForOrLetClause} := \texttt{ForClause} \mid \texttt{LetClause}$$
$$\texttt{ForOrLetClauses} := \texttt{ForOrLetClause}*$$

Moreover, we assume that all variable names are unambiguous. Since we sometimes introduce new variables or change the bindings of existing ones, we introduce a notation for variable substitution: $\texttt{Expr}[\$x2 \leftarrow \$x1]$ denotes `Expr` with all free occurrences of $\$x2$ replaced by $\$x1$.

Running Example

We illustrate the application of our rules on the query from the introduction. Applying our rules to the example query yields a query having a single FLWOR block.

```
let $auction := doc("auction.xml") return
  let $euro:=for $o in $auction/site/open_auctions/open_auction
             for $i in $auction/site/regions/europe/item/@id
             where $o/itemref/@item eq $i
             return $o
  for $a in $euro
  where zero-or-one($a/bidder[1]/increase/text()) * 2
        <= $a/bidder[last()]/increase/text()
  return
    for $p in $auction/site/people/person[profile/@income > 5000]
    for $w in $p/watches/watch
    where $a/@id = $w/@open_auction
    return <auction id="{$a/@id}">
             <increase first="{$a/bidder[1]/increase/text()}"
                       last="{$a/bidder[last()]/increase/text()}"/>
             <watched_by id="{$p/@id}"/>
           </auction>
```

In practice, this query could well be the result of an inlined XQuery function. XQuery functions are often used as *views* to increase data independence, or simply to make queries more readable, similar to views in SQL. In our case, the sequence bound to `$euro` could be an inlined function to retrieve European Auctions, whereas the bottommost FLWOR expression could be a function to retrieve all watchers for a given auction. The results of these functions are joined using the surrounding FLWOR block. In such a context, the application of our rewrite rules can also be described as *view merging*, allowing the plan generator to optimize join orders beyond view borders.

6.2 Normalization

Normalization does not decrease the FLWOR nesting level of a query. Instead, it transforms the query such that the unnesting rewrite rules can still be applied in

case of minor syntactical variations. In addition to this preparatory character, normalization also directly helps to achieve our ultimate goal of preparing queries for plan generation: XQuery allows several different ways of formulating predicates (e.g. the `where` clause and XPath predicates). However, the plan generator requires a single unified formulation of all the constraints on all the variables in the currently considered query block to systematically explore the search space of alternative plans. Execution plan alternatives for value-based predicates include, but are not limited to, placement of selection operators, use of joins, and index selection. Which of these alternatives is used, and in which order the different predicates of a query are evaluated, should not depend on the nesting level or placement of the predicates. This robustness is achieved by our normalization phase.

Normalization proceeds in several consecutive steps, as shown in in Fig. 6.1. We first enforce a simple form for all `return` clauses before we break down complex locations paths into primitives, with an emphasis on predicate normalization. Finally, we eliminate common subexpressions.

6.2.1 Return Normalization

In order to allow a uniform treatment of nested expressions in `return` and `let` clauses, we move all `ExprSingle` expressions from `return` clauses to `let` clauses (see Rewrite 6.1). This way, we can treat unnesting of `return` and `let` uniformly and can always assume that a `return` clause contains a single variable reference.

`ForOrLetClauses`
`WhereClause?`
`OrderByClause?`
`return` $\texttt{ExprSingle}_1$
	$\rightarrow$	`ForOrLetClauses`
		`let` $\$x1 :=$ $\texttt{ExprSingle}_1$
		`WhereClause?`
		`OrderByClause?`
		`return` $\$x1$
			(6.1)

$\texttt{ForOrLetClauses}_1$
`let` $\$x1 :=$ $\texttt{ExprSingle}_1$
$\texttt{ForOrLetClauses}_2$
`WhereClause?`
`OrderByClause?`
`return` $\$x1$
	$\rightarrow$	$\texttt{ForOrLetClauses}_1$
		`for` $\$x1$ `in` $\texttt{ExprSingle}_1$
		$\texttt{ForOrLetClauses}_2$
		`WhereClause?`
		`OrderByClause?`
		`return` $\$x1$
			(6.2)

Condition: There are no other occurrences of $\$x1$.

Figure 6.2: Return rewrites

Other than normalizing the `return` clause, we can further prepare optimization by converting the new `let` clause into a `for` clause (see Rewrite 6.2). This is possible because on the right-hand side the concatenation semantics of FLWOR blocks reestablishes the same result sequence as on the left-hand side of the rewrite, as long as $\$x1$ is not used anywhere but in the `return` clause.

Turning `let` into `for` expressions allows a significantly larger range of alternatives for plan generation. Evaluation of `for` clauses can be done in an iterative manner, generating the items of the binding sequence one by one, instead of computing and materializing the whole sequence at once. This allows efficient techniques such as pipelining and is the preferred style of implementation in database runtime engines [].

Running Example

Applying the `return` elimination and `let` transformation rewrites (6.1 and 6.2) to the `return` expressions of our example query results in the following:

```
let $auction := doc("auction.xml")
for $x1 in let $euro:= for $o in $auction/site/open_auctions/open_auction
                       for $i in $auction/site/regions/europe/item/@id
                       where $o/itemref/@item eq $i
                       return $o
           for $a in $euro
           for $x2 in for $p in $auction/site/people/person[profile/@income > 5000]
                      for $w in $p/watches/watch
                      for $x3 in <auction id="{$a/@id}">
                                   <increase first="{$a/bidder[1]/increase/text()}"
                                             last="{$a/bidder[last()]/increase/text()}"/>
                                   <watched_by id="{$p/@id}"/>
                                 </auction>
                      where $a/@id = $w/@open_auction
                      return $x3
           where zero-or-one($a/bidder[1]/increase/text()) * 2
                 <= $a/bidder[last()]/increase/text()
           return $x2
return $x1
```

6.2.2 Path Normalization

Path expressions are a crucial performance factor for the evaluation of almost every XQuery query. For efficiently evaluating path expressions, the plan generator makes cost-based decisions on algorithms that should be used to evaluate them. For example, an optimizer decides whether a holistic approach (e.g. [21, 66]) for evaluating multiple path expressions is superior to a fine granular approach that evaluates single steps individually (e.g. [2, 53]) probably with the help of an index. The plan generator requires a canonical form of the path expressions to make such decisions. Besides separating each processing step for plan generation, cutting path expressions involves two other advantages:

- It allows to move location step predicates from the middle of location paths into the `where` clause.
- Common subexpression elimination (see below) can be done at the granularity of steps.

Sometimes it is not possible to cut location path into single steps. For example, if a path occurs in a `let` binding or if duplicates are generated by a path. In these cases, we rely on the algebraic techniques, we have proposed in previous chapters. Our full-fledged XPath approach can be seamlessly integrated in the approaches proposed by [82] in order to evaluate such XPath expressions.

Path Tailoring

`for` $x `in` `StepExpr`/`PathExpr`	→	`for` $x1 `in` `StepExpr` `for` $x2 `in` $x1/`PathExpr`	(6.3)
Condition: `StepExpr` must not produce duplicates.			
`for` $x `in` ddo(`StepExpr`)/`PathExpr`	→	`for` $x1 `in` `StepExpr` `for` $x2 `in` $x1/`PathExpr`	(6.4)
`for` $x `in` `StepExpr`/`PathExpr`	→	`let` $x1 := `StepExpr` `for` $x2 `in` $x1/`PathExpr`	(6.5)
`let` $x := `StepExpr`/`PathExpr`	→	`let` $x1 := `StepExpr` `let` $x2 := $x1/`PathExpr`	(6.6)

Figure 6.3: Path tailoring rewrites

In order to separate each processing step, we first extract all path expressions from the query which are not already binding expressions of `for` or `let`, and bind them to new `let` variables. We keep path expressions in `for` clauses because they need a different treatment in our predicate rewrites below.

Having extracted all path expressions, we cut them up into single location steps (see Fig. 6.3 for rewriting rules). Again to facilitate iterator-based evaluation, we attempt to avoid `let` clauses when possible (6.3 and 6.5) while breaking up path expressions in `for` clauses. Without further refinements, we can only cut those steps that do not produce duplicates (see [59, 60]). Rule 6.4 uses the ddo function (`fs:distinct-doc-order`) which is defined in the XQuery formal semantics [37]. This function can be introduced as for example described in [60]. With this rule, we can tailor and later factorize path which would not have been possible without explicit treatment of the ddo function.

Of course, location steps assigned to a `let` variable remain in a `let` binding (6.6).

Predicate Normalization

The plan generator not only decides on the path evaluation algorithms and the order of joins based on structural predicates, but also on the order of regular, value-based joins and selections. Moving all non-structural predicates into the `where` clause makes such join and selection predicates explicitly available in a uniform manner. This allows a search space of plans that is robust against the syntactical placement of the predicate. Further, a unified `where` also allows predicate processing, including, but not limited to inference of new predicates, and elimination of redundant ones.

$$\begin{array}{l}\texttt{ForOrLetClauses}_1\\ \texttt{for}\ \$x1\ \texttt{in}\ \texttt{StepExpr}[\texttt{Expr}_1]\\ \texttt{ForOrLetClauses}_2\\ \texttt{where}\ \texttt{Expr}_2\\ \texttt{OrderByClause}?\\ \texttt{return}\ \texttt{ExprSingle}_1\end{array} \rightarrow \begin{array}{l}\texttt{ForOrLetClauses}_1\\ \texttt{for}\ \$x1\ \texttt{in}\ \texttt{StepExpr}\\ \texttt{ForOrLetClauses}_2\\ \texttt{where}\ \texttt{fn}:\texttt{boolean}(\$x1/(\texttt{Expr}_1))\ \texttt{and}\ \texttt{Expr}_2\\ \texttt{OrderByClause}?\\ \texttt{return}\ \texttt{ExprSingle}_1\end{array} \quad (6.7)$$

Condition: The value of $\texttt{Expr}_1$ must not depend on the context position or context size.

$$\begin{array}{l}\texttt{ForOrLetClauses}_1\\ \texttt{let}\ \$x1 := \texttt{StepExpr}[\texttt{Expr}_1]\\ \texttt{ForOrLetClauses}_2\\ \texttt{where}\ \texttt{Expr}_2\\ \texttt{OrderByClause}?\\ \texttt{return}\ \texttt{ExprSingle}_1\end{array} \rightarrow \begin{array}{l}\texttt{ForOrLetClauses}_1\\ \texttt{let}\ \$x1 := \texttt{StepExpr}\\ \texttt{ForOrLetClauses}_2\\ \texttt{where}\ \texttt{fn}:\texttt{boolean}(\texttt{Expr}_1)\ \texttt{and}\ \texttt{Expr}_2\\ \texttt{OrderByClause}?\\ \texttt{return}\ \texttt{ExprSingle}_1\end{array} \quad (6.8)$$

Condition: The value of $\texttt{Expr}_1$ must not depend on the focus (context item, context position, or context size).

$$\begin{array}{l}\texttt{ForOrLetClauses}_1\\ \texttt{for}\ \$x1\ \texttt{in}\ \texttt{StepExpr}[\texttt{Expr}_1\ \texttt{and}\ \texttt{Expr}_2]\\ \texttt{ForOrLetClauses}_2\\ \texttt{where}\ \texttt{Expr}_3\\ \texttt{OrderByClause}?\\ \texttt{return}\ \texttt{ExprSingle}_1\end{array} \rightarrow \begin{array}{l}\texttt{ForOrLetClauses}_1\\ \texttt{for}\ \$x1\ \texttt{in}\ \texttt{StepExpr}[\texttt{Expr}_2]\\ \texttt{ForOrLetClauses}_2\\ \texttt{where}\ \texttt{fn}:\texttt{boolean}(\$x1/(\texttt{Expr}_1))\ \texttt{and}\ \texttt{Expr}_3\\ \texttt{OrderByClause}?\\ \texttt{return}\ \texttt{ExprSingle}_1\end{array} \quad (6.9)$$

Condition: The value of $\texttt{Expr}_1$ must not depend on the context position or context size.

$$\begin{array}{l}\texttt{ForOrLetClauses}_1\\ \texttt{for}\ \$x1\ \texttt{in}\ \texttt{StepExpr}[\texttt{Expr}_1\ \texttt{and}\ \texttt{Expr}_2]\\ \texttt{ForOrLetClauses}_2\\ \texttt{where}\ \texttt{Expr}_3\\ \texttt{OrderByClause}?\\ \texttt{return}\ \texttt{ExprSingle}_1\end{array} \rightarrow \begin{array}{l}\texttt{ForOrLetClauses}_1\\ \texttt{for}\ \$x1\ \texttt{at}\ \$y1\ \texttt{in}\ \texttt{StepExpr}[\texttt{Expr}_2]\\ \texttt{ForOrLetClauses}_2\\ \texttt{where}\ \texttt{Expr}'_1\ \texttt{and}\ \texttt{Expr}_3\\ \texttt{OrderByClause}?\\ \texttt{return}\ \texttt{ExprSingle}_1\end{array} \quad (6.10)$$

Conditions: The value of $\texttt{Expr}_1$ depends on the context position, but not the context size. $\texttt{Expr}'_1 := \texttt{Expr}_1[\$fs : \text{position} \leftarrow \$y1]$ and `StepExpr` must not consist of a reverse axis step (see text).

Figure 6.4: Predicate normalization rewrites

In Fig. 6.4, we present rules that get predicate expressions of location steps and move them into the `where` clause of the surrounding FLWOR block. For each extracted predicate expression, we have to set the context to the context defined by

the according step. For example, if we move $\texttt{Expr}_1$ from a location step predicate into a `where` clause (see Rule 6.7), we have to guarantee that all context accesses are performed with respect to $\$x1$, which is why we prepend $\$x1$ to the predicate expression. Similarly, we can get comparison expressions that contain calls to the context position of a location step by creating a positional variable using the `for VarRef at VarRef` syntax and replacing accesses to the context position with the variable (see Rule 6.10). This is not strictly possible in XQuery syntax, but easily implemented in most evaluators, because the context position is modeled as a special variable anyway. Our choice of variable name ($fs:position) follows the XQuery Formal Semantics, which also replaces context position by a special variable. Further, reverse axis steps cannot be handled this way, because the context position numbering is different from the order of the result sequence[2].

Note that for the sake of brevity, we assume that there always is a `where` clause in the outer expression. We treat `where`-less outer FLWORs as if there was a `where true` clause.

Common Path Elimination

$\texttt{let } \$x1 := \texttt{StepExpr}_1$ $\texttt{let } \$x2 := \texttt{StepExpr}_1\texttt{/StepExpr}_2$	$\rightarrow$	$\texttt{let } \$x1 := \texttt{StepExpr}_1$ $\texttt{let } \$x2 := \$x1\texttt{/StepExpr}_2$	(6.11)
$\texttt{for } \$x1 \texttt{ in StepExpr}_1$ $\texttt{for } \$x2 \texttt{ in StepExpr}_1\texttt{/StepExpr}_2$	$\rightarrow$	$\texttt{let } \$x0 := \texttt{StepExpr}_1$ $\texttt{for } \$x1 \texttt{ in } \$x0$ $\texttt{for } \$x2 \texttt{ in } \$x0\texttt{/StepExpr}_2$	(6.12)
$\texttt{let } \$x1 := \texttt{StepExpr}_1$ $\texttt{for } \$x2 \texttt{ in StepExpr}_1\texttt{/StepExpr}_2$	$\rightarrow$	$\texttt{let } \$x1 := \texttt{StepExpr}_1$ $\texttt{for } \$x2 \texttt{ in } \$x1\texttt{/StepExpr}_2$	(6.13)
$\texttt{for } \$x1 \texttt{ in StepExpr}_1$ $\texttt{let } \$x2 \texttt{ in } \$x1\texttt{/StepExpr}_2$	$\rightarrow$	$\texttt{let } \$x0 := \texttt{StepExpr}_1$ $\texttt{for } \$x1 \texttt{ in } \$x0$ $\texttt{let } \$x2 := \$x0\texttt{/StepExpr}_2$	(6.14)

Figure 6.5: Common path elimination

To avoid redundant evaluation, we eliminate common paths, binding them to new `for` or `let` variables as needed. In Fig. 6.5, we present four rules for eliminating common location steps. However, elimination of common subexpressions is a complex process that cannot be sufficiently described using only those rules. We refer to [1] for algorithms on subexpression elimination.

[2] If the rewrite is not done on source level, the internal representation may have a suitable special variable to bind for reverse axis numbering, making our rewrite possible again.

Running Example

In the following, we present the query that is obtained by applying normalization, i.e. path extraction, path tailoring, predicate normalization, and common path elimination, to our example query.

```
let $auction := doc("auction.xml")
let $x32 := $auction/site
for $x1 in let $euro:= for $o in $x32, $x13 in $o/open_auctions
                       for $x14 in $x13/open_auction, $i in $x32,
                       for $x15 in $i/regions, $x16 in $x15/europe,
                       for $x17 in $x16/item, $x18 in $x17/@id
                       let $x4 := $x14/itemref, $x19 := $x4/@item
                       where $x19 eq $x18
                       return $x14
          for $a in $euro
          let $x33 := $a/bidder[1], $x34 := $x33/increase
          let $x35 := $x34/text()
          let $x36 := $a/bidder[last()], $x37 := $x36/increase
          let $x38 := $x37/text(), $x39 := $a/@id
          for $x2 in for $p in $x32, $x20 in $p/people, $x21 in $x20/person
                     for $w in $x21/watches, $x22 in $w/watch
                     let $x8 := $x21/@id
                     let $x10 := $x22/@open_auction
                     let $x13 := $x21/profile, $x27 := $x13/@income
                     for $x3 in <auction id="{$x39}">
                                  <increase first="{$x35}" last="{$x38}"/>
                                  <watched_by id="{$x8}"/>
                                </auction>
                     where $x39 = $x10 and  $x27 > 5000
                     return $x3
          where zero-or-one($x35) * 2 <= $x38
          return $x2
return $x1
```

In this expression, for example, the XPath predicate `profile/@income > 5000` is removed from the location step and added to the `where` clause of the according FLWOR block. Moreover, we replaced the common path expressions from within the element construction and the `where` clauses (e.g. the path selecting the increases of the first and last bid) by single variables. Note that it is not possible to move the positional predicates into the `where` clause, as they occur in a `let` binding. Also note that for presentation purposes, we abbreviated consecutive occurrences of `for` and `let` expressions using commas. In the full representation of this query, `for` and `let` expressions that bind multiple variables are split into separate expressions.

6.3 Merging FLWOR Blocks

After finishing the normalization phase, the query is prepared for the core rules of our toolkit, the `for` and `let` merging rewrites. The ultimate goal of the rewrites presented in this section is to reduce the number of query blocks as much as possible.

Reconsider our normalized example query shown above. This formulation of

the query contains several nested FLWOR expressions. The FLWOR nesting depth in Line 3 is three. The `for`-clause binding $o is nested in a `let` clause which, in turn, is nested in the outer-most `for`-clause binding $x1. Moreover, the query contains a `for` clause defining $x2 whose binding sequence is generated by another `for` clause.

In the following, we introduce rewrite rules that remove such nested expressions. Applying them to our example query eliminates all nested FLWORs.

We start with rewrites that remove FLWORs nested in `for` clauses (see Fig. 6.6), and then proceed to `let` clauses (see Fig. 6.7).

6.3.1 For Rewrites

The semantics of a `for` clause is to iterate over items of the binding sequence, binding the `for` variable to every item from that sequence. The remaining FLWOR expression is evaluated for each such binding, and the individual result sequences are concatenated. We are interested in a `for` clause if its binding sequence is created by a nested FLWOR expression. In some cases, we can lift the inner FLWOR to the outer level. This rewrite opportunity results from the fact that sequences in the XQuery data model are never nested. Hence, it often does not matter on how many levels of implicit concatenation of `return` sequences occurs, because the result is always a flat sequence.

$$
\begin{array}{l}
\texttt{ForOrLetClauses}_1 \\
\texttt{for } \$x1 \texttt{ in } (\texttt{ForOrLetClauses}_2 \\
\qquad \texttt{for } \$x2 \texttt{ in ExprSingle}_1 \\
\qquad \texttt{ForOrLetClauses}_3 \\
\qquad \texttt{where ExprSingle}_2 \\
\qquad \texttt{return } \$x2) \\
\texttt{ForOrLetClauses}_4 \\
\texttt{where ExprSingle}_3 \\
\texttt{return VarRef}_1
\end{array}
\quad \rightarrow \quad
\begin{array}{l}
\texttt{ForOrLetClauses}_1 \\
\texttt{ForOrLetClauses}_2 \\
\texttt{for } \$x1 \texttt{ in ExprSingle}_1 \\
\texttt{ForOrLetClauses}'_3 \\
\texttt{ForOrLetClauses}_4 \\
\texttt{where ExprSingle}_3 \texttt{ and ExprSingle}'_2 \\
\texttt{return VarRef}_1
\end{array}
\tag{6.15}
$$

Conditions: $\texttt{ForOrLetClauses}'_3 := \texttt{ForOrLetClauses}_3[\$x2 \leftarrow \$x1]$ and $\texttt{ExprSingle}'_2 := \texttt{ExprSingle}_2[\$x2 \leftarrow \$x1]$

$$
\begin{array}{l}
\texttt{ForOrLetClauses}_1 \\
\texttt{for } \$x1 \texttt{ in } (\texttt{ForOrLetClauses}_2 \\
\qquad \texttt{let } \$x2 := \texttt{ExprSingle}_1 \\
\qquad \texttt{ForOrLetClauses}_3 \\
\qquad \texttt{where ExprSingle}_2 \\
\qquad \texttt{return } \$x2) \\
\texttt{ForOrLetClauses}_4 \\
\texttt{where ExprSingle}_3 \\
\texttt{return VarRef}_1
\end{array}
\quad \rightarrow \quad
\begin{array}{l}
\texttt{ForOrLetClauses}_1 \\
\texttt{ForOrLetClauses}_2 \\
\texttt{let } \$x2 := \texttt{ExprSingle}_1 \\
\texttt{ForOrLetClauses}_3 \\
\texttt{for } \$x1 \texttt{ in } \$x2 \\
\texttt{ForOrLetClauses}_4 \\
\texttt{where ExprSingle}_3 \texttt{ and ExprSingle}_2 \\
\texttt{return VarRef}_1
\end{array}
\tag{6.16}
$$

Figure 6.6: For rewrites

For example, consider the left-hand side of the first `for` Rewrite 6.15. In this rewrite, the variable $x1 is iteratively bound to each item returned by the inner

FLWOR. The result of the inner FLWOR is generated by the `return` clause. Note that in our case the `return` clause consists only of a variable reference, i.e. variable $x2. To merge the two blocks, we have to guarantee that the outer `for` variable $x1, after merging, is still bound to the same items, i.e. those generated by variable $x2. To this end, we replace the nested FLWOR with the expression responsible for binding $x2. In the rewrite this expression is called $\texttt{ExprSingle}_1$ and bound by a `for` clause. The remaining (optional) clauses are moved into the outer FLWOR block. Specifically, $\texttt{ForOrLetClauses}_2$ and $\texttt{ForOrLetClauses}_3$ are pulled up one level. $\texttt{ExprSingle}_2$ from the inner `where` clause is conjunctively connected to the expression in the outer `where` clause [3]. After relocating the inner expressions, we have to replace free occurrences of the previous inner variable $x2 with $x1.

Similarly, we merge two query blocks if the binding sequence is created by a nested `let` variable (see our Rewrite Rule 6.16). Note that the right-hand side of Rule 6.16 may still contain a FLWOR nested in a `let` clause. This case is unnested by Rule 6.17, which is presented in the next section.

Running Example

On our example query, we can apply Rewrite Rule 6.15 twice. First, to eliminate the inner **for** -clause binding $x2, as this variable is returned to create the binding sequence for $x1. Second, we apply this rule to eliminate the `for` expression binding $x3. This results in the following expression:

```
let $auction := doc("auction.xml")
let $x32 := $auction/site
let $euro:= for $o in $x32, $x13 in $o/open_auctions, $x14 in $x13/open_auction
            for $i in $x32, $x15 in $i/regions, $x16 in $x15/europe
            for $x17 in $x16/item, $x18 in $x17/@id
            let $x4 := $x14/itemref, $x19 := $x4/@item
            where $x19 eq $x18
            return $x14
for $a in $euro
let $x33 := $a/bidder[1], $x34 := $x33/increase, $x35 := $x34/text()
let $x36 := $a/bidder[last()], $x37 := $x36/increase, $x38 := $x37/text()
let $x39 := $a/@id
for $p in $x32, $x20 in $p/people, $x21 in $x20/person
for $w in $x21/watches, $x22 in $w/watch
let $x8 := $x21/@id, $x10 := $x22/@open_auction
let $x13 := $x21/profile, $x27 := $x13/@income
for $x1 in <auction id="{$x39}">
             <increase first="{$x35}"
                       last="{$x38}"/>
             <watched_by id="{$x8}"/>
           </auction>
where zero-or-one($x35) * 2 <= $x38 and $x39 = $x10 and $x27 > 5000
return $x1
```

[3] As before, expressions without `where` are treated as if a `where true` clause was added.

6.3.2 Let Rewrites

`let` clauses require separate rewrites because they bind a variable to the result of its associated expression, i.e. without iterating over this result. Fig. 6.7 presents three rewrite rules to eliminate FLWORs nested in `let` clauses.

```
ForOrLetClauses1                          ForOrLetClauses1
let $x1 := ExprSingle1                    ForOrLetClauses2
ForOrLetClauses2                  →       for $x2 in ExprSingle1                    (6.17)
for $x2 in $x1                            ForOrLetClauses3
ForOrLetClauses3                          where ExprSingle2
where ExprSingle2                         return VarRef
return VarRef
```

Condition: There are no other occurrences of $x1.

```
ForOrLetClauses1                          ForOrLetClauses1
let $x1 := (ForOrLetClauses2              ForOrLetClauses2
            for $x2 in ExprSingle1        for $x2 in ExprSingle1                    (6.18)
            where ExprSingle2       →     where ExprSingle3 and ExprSingle2
            return $x2)                   return $x2
where ExprSingle3
return $x1
```

Condition: There are no other occurrences of $x1.

```
ForOrLetClauses1                          ForOrLetClauses1
let $x1 := (ForOrLetClauses2              ForOrLetClauses2
            let $x2 := ExprSingle1        let $x2 in ExprSingle1                    (6.19)
            where ExprSingle2       →     where ExprSingle3 and ExprSingle2
            return $x2)                   return $x2
where ExprSingle3
return $x1
```

Condition: There are no other occurrences of $x1.

Figure 6.7: Let rewrites

Rewrite Rule 6.17 tackles a frequently used case. There, a `for` iteration is used to enumerate all items contained in a `let` variable. This technique is used in our example query and may result from inlining an XQuery function as explained at the beginning of this section. The rules suggests to eliminate the `let` variable if it is used only once and inline the associated expression (i.e. $\texttt{ExprSingle}_1$). On this result, the rewrites of the previous section (see Fig. 6.6) can be applied and eliminate the nesting.

Fig. 6.7 also contains two rewrites that remove nested `for` (see Rule 6.18) and `let` (see Rule 6.19) expressions, respectively. The outer `let` clause in both rules is immediately followed by the `where` clause. If there was another `for` or `let` clause in between, it wouldn't contain occurrences of $x1$ and, hence, could w.l.o.g. be moved above the `let` clause that is binding $x1$.

Running Example

The result of applying the `let` Rewrite Rule 6.17 and the `for` Rewrite Rule 6.15 to our example is the following query finally consisting of a single query block.

```
let $auction := doc("auction.xml"), $x32 := $auction/site
for $o in $x32, $x13 in $o/open_auctions, $a in $x13/open_auction
for $i in $x32, $x15 in $i/regions, $x16 in $x15/europe
for $x17 in $x16/item, $x18 in $x17/@id
let $x4 := $a/itemref, $x19 := $x4/@item
let $x33 := $a/bidder[1], $x34 := $x33/increase, $x35 := $x34/text()
let $x36 := $a/bidder[last()], $x37 := $x36/increase, $x38 := $x37/text()
let $x39 := $a/@id
for $p in $x32, $x20 in $p/people, $x21 in $x20/person
for $w in $x21/watches, $x22 in $w/watch
let $x8 := $x21/@id, $x10 := $x22/@open_auction
let $x13 := $x21/profile, $x27 := $x13/@income
for $x1 in  <auction id="{$x39}">
              <increase first="{$x35}"
                        last="{$x38}"/>
              <watched_by id="{$x8}"/>
            </auction>
where zero-or-one($x35) * 2 <= $x38 and $x39 = $x10
      and $x27 > 5000 and $x19 eq $x18
return $x1
```

Note how in this form all value-based join and selection predicates are available in a unified `where` clause. This allows a plan generator to decide on index access and join orders.

6.4 Intricacies

In the last section, we presented rewrite rules to reduce the number of FLWOR expressions in a query. However, all of the presented elimination rewrites were limited in terms of FLWORs that do not contain positional variables or `order by` clauses. Merging FLWORs which contain one of these constructs requires more sophisticated rewrite techniques. In case of an `order by` clause, for example, one has to keep track of the order that is given (1) through the explicit sort statement in an outer query block and (2) implicitly in a nested FLWOR expression. In this section, we present new rewrite rules that are similar to the previously presented rules but also fix one of these intricacies.

6.4.1 Positional For Rewrites

First, we consider the case that the outer block contains a positional clause. A `for` expression that has a nested FLWOR and additionally binds a positional variable is called a *positional* `for` expression. Fig. 6.8 presents two rewrites that merge an expression that is nested inside a positional `for` expression. Both have on the left-hand side a positional `for` expression. Each of them binds the positional variable $\$y1$ using the `for VarRef at VarRef` syntax. In order to be able to merge the outer and the inner FLWOR into one, the inner FLWOR must not contain a

`where` or `order by` clause. Otherwise, the numbering of the result sequence of the inner query could not be guaranteed after merging. Our solution for merging such FLWOR expressions is shown on the right-hand side of Rewrite Rules 6.20 and 6.21.

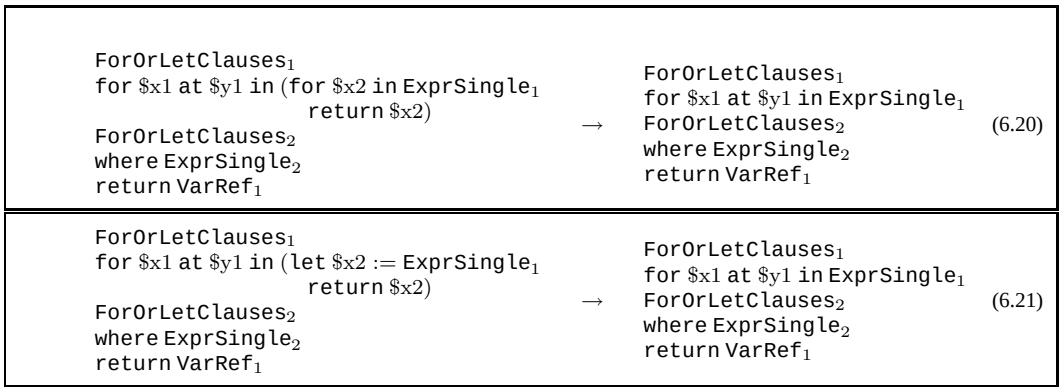


Figure 6.8: Positional for rewrites

The former rule contains a nested **for** clause. In this case, we can simply reduce the nesting level and pull $\texttt{ExprSingle}_1$ from the inner into the outer block. Because there are no other clauses in the inner query block, we do not reorder the result and, hence, keep the numbering. The inner FLWOR block of the latter rule starts with a `let` clause. Here, eliminating the `let` clause is possible for the same reason.

6.4.2 Order-by

A second restriction on the rewrites from the previous section is that neither the outer nor the inner FLWOR can contain an `order by` clause. In this section, we augment our rewrite merging rules with expressions containing an `order by` clause. We present rules for all cases, i.e. either `for` or `let` expressions are nested in a `for` clause or they are nested in a `let` clause, respectively.

If an `order by` clause is present, we have to guarantee that both the explicit order given by the `order by` clause and the implicit order given by the occurrence of `for` or `let` clauses are preserved. For maintaining the hierarchy in the unnested case, we introduce a technique that we call *Canonical Order By*. This technique modifies `for` clauses to bind a positional variable and add an `order by` clause to the corresponding FLWOR expression which has the new positional variables in its `OrderSpecList`. If there are multiple `for` and/or `let` clauses in one FLWOR expression, the order of the positional variables in the `OrderSpecList` is determined by the order in which these clauses occur in the FLWOR block. Doing so, we guarantee that the result is in requested order.

In the following, we show how this technique can be applied to merge FLWOR expressions that contain an `order by` clause in the outer and/or the inner FLWOR. Analogously to the previous section, we start with `for` rewrites and then proceed to `let` merging rules.

For Rewrites

Fig. 6.9 presents four rules to tackle queries whose inner or outer FLWOR contains an `order by` clause.

The left-hand side of each rule in this figure is similar to a rule from Fig. 6.6 but allows for `order by` clauses. The right-hand side of all rewrite rules shows how the outer and inner FLWOR expression can be merged, despite the order of the result is not changed.

At the bottom of each rule, we delineate the expressions that need to bind positional variables. For example, on the right-hand side of Equivalence 6.22 the i-th `for` clause in $\texttt{ForOrLetClauses}_1$ binds the positional variable $\$\mathrm{y}1_i$ using the `for VarRef at VarRef` syntax. In Rules 6.22 and 6.23, we could omit the positional variables $\$\mathrm{y}4_1, \ldots, \$\mathrm{y}4_m$ and instead use `stable order by`. However, this case could be less efficient, as it may require to keep of the sortedness of unnecessary attributes.

In case both the outer and inner FLWOR contain an `order by` clause, we proceed as shown in Fig. 6.6. However, we need to merge the two `order by` clauses such that sorting is done according to the `OrderSpecList` of the outer FLWOR first and then for the `OrderSpecList` of the inner FLWOR block.

Let Rewrites

Similarly to the previous subsection, we also allow our `let` merging rules from Fig. 6.7 to contain `order by` clauses. These rules are presented in Fig. 6.10.

The left-hand side of Rewrite Rule 6.26 corresponds to Rule 6.17 but has an `order by` clause in the outer block. On the right-hand side, we merge the `let` clause binding $\$\mathrm{x}_1$ with the `for` clause that iterates over the sequence resulting from $\$\mathrm{x}_1$. The `order by` clause remains with the outer block.

In Rule 6.27, we eliminate the `let` clause in the outer block and, therefore, pull the inner block one level higher. However, in this case, the `for` clauses of the outer block are augmented to bind new positional variables on which sorting is done. This is necessary to keep the order that is predetermined by the outer block.

Rules 6.28 and 6.29 merge blocks that have a `let` clause in the outer block that contains a nested `for` or `let` expression. Additionally, the outer clause features an `order by` clause. The `order by` clause of the merged representation is augmented to reestablish the order of the inner FLWOR block.

$$
\begin{array}{l}
\texttt{ForOrLetClauses}_1 \\
\texttt{for } \$x1 \texttt{ in } (\texttt{ForOrLetClauses}_2 \\
\qquad \texttt{for } \$x2 \texttt{ in ExprSingle}_1 \\
\qquad \texttt{ForOrLetClauses}_3 \\
\qquad \texttt{where ExprSingle}_2 \\
\qquad \texttt{order by OrderSpecList}_1 \\
\qquad \texttt{return } \$x2) \\
\texttt{ForOrLetClauses}_4 \\
\texttt{where ExprSingle}_4 \\
\texttt{return VarRef}_1
\end{array}
\quad \rightarrow \quad
\begin{array}{l}
\texttt{ForOrLetClauses}_1 \\
\texttt{ForOrLetClauses}_2 \\
\texttt{for } \$x1 \texttt{ at } \$y2 \texttt{ in ExprSingle}_1 \\
\texttt{ForOrLetClauses}'_3 \\
\texttt{ForOrLetClauses}_4 \\
\texttt{where ExprSingle}_4 \texttt{ and ExprSingle}'_2 \\
\texttt{order by } \$y1_1, \ldots, \$y1_n, \$y2, \\
\qquad \texttt{OrderSpecList}'_1, \$y4_1, \ldots, \$y4_m \\
\texttt{return VarRef}_1
\end{array}
\tag{6.22}
$$

On the right-hand side $\texttt{ForOrLetClauses}_1$ bind the positional variables $\$y1_1, \ldots, \$y1_n$ and $\texttt{ForOrLetClauses}_4$ bind $\$y4_1, \ldots, \$y4_m$.
Conditions: $\texttt{ForOrLetClauses}'_3 := \texttt{ForOrLetClauses}_3[\$x2 \leftarrow \$x1]$, $\texttt{ExprSingle}'_2 := \texttt{ExprSingle}_2[\$x2 \leftarrow \$x1]$, and $\texttt{OrderSpecList}'_1 := \texttt{OrderSpecList}_1[\$x2 \leftarrow \$x1]$

$$
\begin{array}{l}
\texttt{ForOrLetClauses}_1 \\
\texttt{for } \$x1 \texttt{ in } (\texttt{ForOrLetClauses}_2 \\
\qquad \texttt{let } \$x2 := \texttt{ExprSingle}_1 \\
\qquad \texttt{ForOrLetClauses}_3 \\
\qquad \texttt{where ExprSingle}_2 \\
\qquad \texttt{order by OrderSpecList}_1 \\
\qquad \texttt{return } \$x2) \\
\texttt{ForOrLetClauses}_4 \\
\texttt{where ExprSingle}_4 \\
\texttt{return VarRef}_1
\end{array}
\quad \rightarrow \quad
\begin{array}{l}
\texttt{ForOrLetClauses}_1 \\
\texttt{ForOrLetClauses}_2 \\
\texttt{let } \$x2 := \texttt{ExprSingle}_1 \\
\texttt{ForOrLetClauses}_3 \\
\texttt{for } \$x1 \texttt{ at } \$y2 \texttt{ in } \$x2 \\
\texttt{ForOrLetClauses}_4 \\
\texttt{where ExprSingle}_4 \texttt{ and ExprSingle}_2 \\
\texttt{order by } \$y1_1, \ldots, \$y1_n, \$y2, \\
\qquad \texttt{OrderSpecList}_1, \$y4_1, \ldots, \$y4_m \\
\texttt{return VarRef}_1
\end{array}
\tag{6.23}
$$

On the right-hand side $\texttt{ForOrLetClauses}_1$ bind the positional variables $\$y1_1, \ldots, \$y1_n$ and $\texttt{ForOrLetClauses}_4$ bind $\$y4_1, \ldots, \$y4_m$.

$$
\begin{array}{l}
\texttt{ForOrLetClauses}_1 \\
\texttt{for } \$x1 \texttt{ in } (\texttt{ForOrLetClauses}_2 \\
\qquad \texttt{for } \$x2 \texttt{ in ExprSingle}_1 \\
\qquad \texttt{ForOrLetClauses}_3 \\
\qquad \texttt{where ExprSingle}_2 \\
\qquad \texttt{return } \$x2) \\
\texttt{ForOrLetClauses}_4 \\
\texttt{where ExprSingle}_4 \\
\texttt{order by OrderSpecList}_1 \\
\texttt{return VarRef}_1
\end{array}
\quad \rightarrow \quad
\begin{array}{l}
\texttt{ForOrLetClauses}_1 \\
\texttt{ForOrLetClauses}_2 \\
\texttt{for } \$x1 \texttt{ at } \$y2 \texttt{ in ExprSingle}_1 \\
\texttt{ForOrLetClauses}'_3 \\
\texttt{ForOrLetClauses}_4 \\
\texttt{where ExprSingle}_4 \texttt{ and ExprSingle}'_2 \\
\texttt{order by OrderSpecList}_1, \$y2_1, \ldots, \$y2_n, \$y2, \\
\qquad \$y3_1, \ldots, \$y3_m \\
\texttt{return VarRef}_1
\end{array}
\tag{6.24}
$$

On the right-hand side $\texttt{ForOrLetClauses}_1$ bind the positional variables $\$y1_1, \ldots, \$y1_n$ and $\texttt{ForOrLetClauses}'_3$ bind $\$y4_3, \ldots, \$y3_m$.
Conditions: $\texttt{ForOrLetClauses}'_3 := \texttt{ForOrLetClauses}_3[\$x2 \leftarrow \$x1]$, $\texttt{ExprSingle}'_2 := \texttt{ExprSingle}_2[\$x2 \leftarrow \$x1]$, and $\texttt{OrderSpecList}'_1 := \texttt{OrderSpecList}_1[\$x2 \leftarrow \$x1]$

$$
\begin{array}{l}
\texttt{ForOrLetClauses}_1 \\
\texttt{for } \$x1 \texttt{ in } (\texttt{ForOrLetClauses}_2 \\
\qquad \texttt{let } \$x2 := \texttt{ExprSingle}_1 \\
\qquad \texttt{ForOrLetClauses}_3 \\
\qquad \texttt{where ExprSingle}_2 \\
\qquad \texttt{return } \$x2) \\
\texttt{ForOrLetClauses}_4 \\
\texttt{where ExprSingle}_4 \\
\texttt{order by OrderSpecList}_1 \\
\texttt{return VarRef}_1
\end{array}
\quad \rightarrow \quad
\begin{array}{l}
\texttt{ForOrLetClauses}_1 \\
\texttt{ForOrLetClauses}_2 \\
\texttt{let } \$x2 := \texttt{ExprSingle}_1 \\
\texttt{ForOrLetClauses}_3 \\
\texttt{for } \$x1 \texttt{ at } \$y2 \texttt{ in } \$x2 \\
\texttt{ForOrLetClauses}_4 \\
\texttt{where ExprSingle}_4 \texttt{ and ExprSingle}_2 \\
\texttt{order by OrderSpecList}_1, \$y2_1, \ldots, \$y2_n, \$y2, \\
\qquad \$y3_1, \ldots, \$y3_m \\
\texttt{return VarRef}_1
\end{array}
\tag{6.25}
$$

On the right-hand side $\texttt{ForOrLetClauses}_1$ bind the positional variables $\$y1_1, \ldots, \$y1_n$ and $\texttt{ForOrLetClauses}_3$ bind $\$y4_3, \ldots, \$y3_m$.

Figure 6.9: Order-by for rewrites

```
ForOrLetClauses1                          ForOrLetClauses1
let $x1 := ExprSingle1                    ForOrLetClauses2
ForOrLetClauses2                          for $x2 in ExprSingle1
for $x2 in $x1                    →       ForOrLetClauses3                    (6.26)
ForOrLetClauses3                          where ExprSingle2
where ExprSingle2                         order by OrderSpecList
order by OrderSpecList                    return VarRef
return VarRef
```

Condition: There are no other occurrences of $x1.

```
ForOrLetClauses1
let $x1 := (ForOrLetClauses2                  ForOrLetClauses1
            for $x2 in ExprSingle1            ForOrLetClauses2
            where ExprSingle2                 for $x2 at $y2 in ExprSingle1
            order by OrderSpecList1     →     where ExprSingle3 and ExprSingle2   (6.27)
            return $x2)                       order by $y11, ..., $y1n, $y2, OrderSpecList1
where ExprSingle3                             return $x2
return $x1
```

On the right-hand side ForOrLetClauses1 bind the positional variables $y11, ..., $y1n.
Condition: There are no other occurrences of $x1.

```
ForOrLetClauses1
let $x1 := (ForOrLetClauses2                  ForOrLetClauses1
            for $x2 in ExprSingle1            ForOrLetClauses2
            where ExprSingle2                 for $x2 at $y2 in ExprSingle1
            return $x2)                 →     where ExprSingle3 and ExprSingle2   (6.28)
where ExprSingle3                             order by OrderSpecList1, $y21, ..., $y2n, $y2
order by OrderSpecList1                       return $x2
return $x1
```

On the right-hand side ForOrLetClauses2 bind the positional variables $y21, ..., $y2n.
Condition: There are no other occurrences of $x1.

```
ForOrLetClauses1
let $x1 := (ForOrLetClauses2                  ForOrLetClauses1
            let $x2 := ExprSingle1            ForOrLetClauses2
            where ExprSingle2                 let $x2 in ExprSingle1
            return $x2)                 →     where ExprSingle3 and ExprSingle2   (6.29)
where ExprSingle3                             order by OrderSpecList1, $y21, ..., $y2n
order by OrderSpecList1                       return $x2
return $x1
```

On the right-hand side ForOrLetClauses2 bind the positional variables $y21, ..., $y2n.
Condition: There are no other occurrences of $x1.

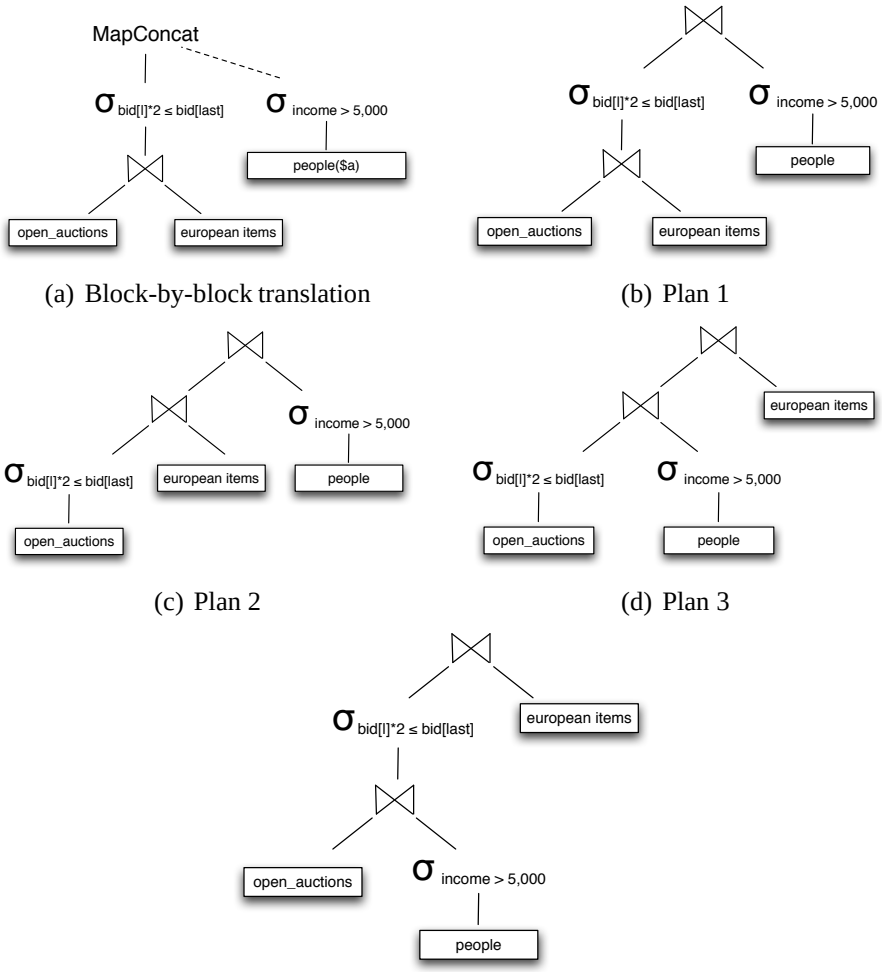
Figure 6.10: Order-by let rewrites

6.5 Evaluation

A goal of this chapter is to show how to rewrite a query into a form that consists of a single query block to give a single run of the plan generator as much uniformly structured information about the query as possible. We now elaborate on the importance of this goal by discussing the optimization of our example query during plan generation. We will see how more efficient plans can be generated only when the query has been reduced to a single block.

In order to demonstrate that better plans are possible, it is not necessary to explore the whole search space available. Hence, we only focus on join ordering. Therefore, we assume that the optimizer has decided on subplans to produce the

sequences for Open Auctions, European Items, and Persons. The subplans may be based on pattern matching algorithms. Further, we assume that the predicate selecting the auctions according to their bids has been converted into a single predicate subplan. This predicate is, however, more expensive to evaluate than a simple value comparison, and its placement in the overall plan does significantly affect performance. Thus, finding an optimal plan includes finding an optimal position for this predicate. We now discuss execution plans for our example query in the form of algebraic expressions on an abstract level (see Fig. 6.11).



(a) Block-by-block translation

(b) Plan 1

(c) Plan 2

(d) Plan 3

(e) Plan 4

Figure 6.11: Alternative execution plans

A straightforward translation of the original, nested, multi-block query looks like Fig. 6.11(a). Here, the FLWOR blocks are translated directly into separate subplans, and no global optimization takes place. For simplicity, we disregard the first line of the example query (the initial `let` clause for the document root). The top-level MapConcat operator represents the main FLWOR expression. Its operand

generates the tuple stream and contains subplans for the European Auctions query block. The subplan connected to MapConcat by the dashed line represents the query block in the `return` clause (the last eight lines of the query). It has a free variable $a in the subplan for the `people` sequence, and, hence, has to be reevaluated for every tuple of the MapConcat operand, as dictated by XQuery FLWOR semantics.

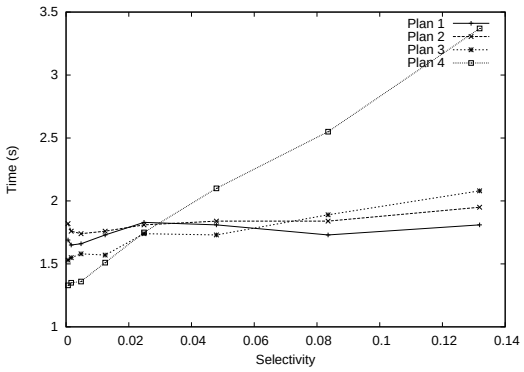


Figure 6.12: Performance results

Fig. 6.11 shows four other execution plans based on the rewritten, single-block form of our example query. These can be enumerated by the plan generator because it has access to all value-based predicates of the query in a single `where` clause, and can detect joins and determine an optimal order for them and the residual selections. We executed all five plans from Fig. 6.11 in our hybrid relational and XML DBMS Natix [40] on an XMark document with scaling factor one.

The experimental setup consisted of a PC with an Intel Pentium D CPU having 3.40GHz and 1GB of main memory, running on openSUSE 10.2 with Linux Kernel 2.6.18 SMP. To investigate the relative performance of the execution plans, we varied the selectivity of the predicate restricting the people by their income between 0.14 and 0. This corresponds to incomes between 60,000$ to 130,000$ instead of 5,000$ in the original query. Fig. 6.12 shows the result of this small performance study (execution time in seconds) for four plans from Fig. 6.11.

The experiment makes obvious why careful global plan generation based on single-block queries is crucial for efficient execution. The results of the nested-loop strategy of the straightforward translation are orders of magnitude slower (well beyond 100s) and have been left out of the graph. The join-based plans made possible by our rewritten single-block query show that an enumeration of alternatives is as important as in relational query processing: Depending on selectivity, the overall best plan varies. The plan according to Fig. 6.11(e) performs best with a very low selectivity, whereas the plan belonging to Fig. 6.11(b) outperforms the others with an increasing selectivity.

6.6 Related Work

Michiels et al. [84] discuss rewrite rules on two levels. Starting from expressions in XQuery Core, they propose to first rewrite them into normal forms (still in XQuery Core) that make the subsequent stages robust against different syntactic formulations of the same query, and to support tree-pattern detection. They also simplify the query by removing unnecessary constructs introduced by Core normalization. Some of these simplification rewrites could be incorporated into our toolkit. The rewritten query is then translated into an algebra that includes a tree-pattern matching operator. These algebraic expressions are then rewritten using algebraic equivalences in order to merge simple path navigation operators into holistic tree pattern matching operators. The rewrite rules on the algebraic level are orthogonal to the ones presented in our toolkit and can be used by a plan generator to create execution plans based on tree-pattern matching.

The very thorough paper by Hidders et al. [60] has a similar aim, but translates a fragment of XQuery directly into tree patterns without an intermediate algebraic phase. In a first phase, the queries are annotated with properties such as result cardinality, ordering, and occurrence of duplicates. These properties are then used to control a rewriting of the query into the Tree Pattern Normal Form (TPNF), which is always possible for the language fragment under consideration. For TPNF, a direct mapping onto tree patterns is then described. Unfortunately, the language fragment does not cover important XQuery constructs, such as value-based predicates. Another problem is that the rewrite rules are based on XQuery Core, which is unsuitable as a plan generator input, for example because the absence of a `where` clause makes it difficult to identify applicable join conditions. However, the property annotations are not only useful for TPNF rewriting and can be used when implementing our rewrite toolkit. Further, the TPNF technique may be used by plan generators to identify parts of the query that can be evaluated using pattern matching.

May et al. [82] have presented unnesting strategies for XQuery. Their approach is based on algebraic equivalences to be applied after translation of XQuery into the NAL algebra of the Natix system. The main focus of that work is unnesting of selection predicates which correspond to `where` clauses on the source level. The paper also discusses unnesting the subscripts of map operators, which on source level corresponds to `let` clauses. However, the rules are exclusively for the conversion of implicit grouping into explicit grouping operators, and not for the general unnesting of `let`. Translated into the source form, the presented rewrite rules are complementary to the rules discussed in this thesis.

6.7 Conclusion

In this chapter, we have proposed a toolkit for rewriting XQuery expressions into expressions with as few query blocks as possible. This form is especially useful

because merging query blocks usually increases the search space of a plan generator and, hence, allows for better query execution strategies.

Our rewrite toolkit basically proceeds in a two-phase manner: First, we normalize the query such that the unnesting rewrite rules can be applied despite syntactical variations considering the `return` clause, XPath expressions, and predicates from the `where` clause.

Second, we apply rewrite rules that merge all variants of nested `for` and `let` clauses. That is, we have given rules that merge inner blocks that are contained in a `for` or `let` binding, respectively. All our rules rely on the fact that sequences in the XQuery data model are never nested. The result is always a flat sequence because the level of implicit concatenations of `return` sequences does not matter.

In a brief experimental study, we have shown that more efficient plans are possible if queries are rewritten using our toolkit.

Although XQuery seems to become the de facto query language for XML, this fact does not supersede our work on the evaluation of XPath. Instead, because XPath is an essential ingredient of XQuery, our algebraic techniques can also be applied within an algebraic framework for XQuery. For example, as we have seen in this chapter, it is not possible to cut XPath expressions that occur in an XQuery `let` binding. In this case, all of our techniques for XPath (i.e. the full-fledged algebraic approach and our unnesting techniques) can directly be applied to such an XPath expression.

Chapter 7

Conclusion & Outlook

7.1 Conclusion

Relational systems have shown how algebraic techniques can be used to efficiently process SQL queries. These systems feature powerful query optimization techniques as well as a fast and scalable implementations of their query execution engines. In this thesis, we have developed a framework for leveraging algebraic techniques to enable the efficient evaluation of XML query languages. Moreover, we have presented novel rewrites to unnest nested SQL queries with disjunctions, a problem that has not yet gained any attention.

7.1.1 Algebraic XPath Evaluation

We have started out with the development of an order-aware tuple-based logical algebra that is capable to evaluate all of XPath. A translation function taking an arbitrary XPath 1.0 expression as argument maps its input onto operators of the new logical algebra. In a first step, we have defined the translation function such that its output does not avoid the exponential runtime behavior of the naïve XPath evaluation. However, in a second step, we refined the translation function in order to remedy this situation. Additionally, we have also described the implementation of our algebra in the runtime system of the native XML database system Natix. Using this system, we performed an evaluation comparing our approach against freely available main-memory engines. As a result, we could observe that our approach can compete with such engines — even though Natix is not purely main-memory based — and that applying optimizations such as pushing duplicate elimination, our approach is able to clearly outperform the others.

7.1.2 Unnesting XPath Expressions

Having established a complete algebraic approach, we have turned to optimizations. Specifically, we have presented algebraic equivalences in order to unnest nested

XPath expressions.

In order to systematically develop new unnesting techniques, we have established a classification of nested XPath expressions. Primarily, this classification distinguishes expressions according to their dependency on the local context and the cardinality of their result. Our equivalences are targeted on unnesting each of the classes. Therefore, we have extended our logical algebra by six operators that are especially useful for unnesting. For example, we have developed the kappa-join operator for efficiently evaluating existentially quantified comparison expressions. As our performance study shows, our unnesting approach clearly outperforms the other XPath evaluators. For example, our main competitor, the approach by Gottlob et al. [46, 47], avoids the exponential runtime in the worst case, but is still several orders of magnitude slower than our unnested approach. The reasons is that they need to materialize intermediate results for different contexts, which is not necessary. We, however, can avoid materialization for different contexts by unnesting nested (independent) XPath expressions.

Existing unnesting techniques that were developed in the relational context are presented as source level rewrites of the query. Rewriting queries on the source level turned out to be error-prone because their validity is difficult to confirm (e.g. [71]). Pursuing unnesting on the algebraic level, however, allowed us to formally ensure the correctness of our rewrites. To this end, we provide proofs for all our unnesting equivalences in the appendix.

7.1.3 Disjunctive Unnesting for XPath

The unnesting techniques we have presented in Chapter 3 are capable of unnesting arbitrary quantified comparison expressions as long as they occur in a conjunction. However, in an XPath predicate, a comparison expression can also occur disjunctively. In this case, all existing unnesting techniques — including ours — fail so far. Hence, we have developed a novel unnesting approach that is able to unnest quantified comparison expressions that occur in a disjunction.

Therefore, we have combined the existing bypass technique (e.g. [27]) with our unnesting approach for nested expressions in conjunctions. This combination allows for the efficient evaluation of nested XPath expressions with disjunctions. For example, we have presented a variant of the kappa-join operator that is injected with the bypass technique. The result is our novel bypass kappa-join which can efficiently evaluate arbitrary existentially quantified comparison expressions that occur in disjunctions. Our experimental study demonstrates the superiority of our approach compared to other existing evaluators.

Furthermore, our approach is not only applicable to unnest nested XPath queries. Instead, combining algebraic unnesting techniques with bypass operators also enables the unnested evaluation of nested XQuery or SQL queries. Our kappa-join, for example, can be used to unnest existentially quantified XQuery expressions [13].

7.1.4 Disjunctive Unnesting for SQL

Unnesting of nested queries with disjunctions has not been studied for any query language so far. In an excursion into the relational world, we have presented an unnesting technique for unnesting such queries. To the best of our knowledge, we are the first to study this problem.

Our unnesting approach is presented by means of algebraic equivalences. Based on the orthogonal classifications by Kim [70] and Muraliskrishna [85], we have presented equivalences to unnest table and scalar subqueries that can be shaped in a single, a tree, or a linear form. The right-hand side of all equivalences features a bypass operator in order to efficiently deal with the disjunction.

Our extensive experimental study has shown, that optimizations for such queries have not abundantly found their ways in commercial DBMSs. This is despite the fact that such techniques seem to become more and more common in practice and emerge, for example, in the latest decision support benchmark of the transaction processing performance council [105]. Our approach, in contrast, outperforms the major commercial systems by orders of magnitude in most cases.

To ensure the correctness of our SQL unnesting equivalences, we provide proofs for all of them in Appendix C. These proofs also confirm the validity of our rewrites on an algebra for bags.

7.1.5 Preparing XQuery for Plan Generation

In Chapter 6, we have been back to XML query optimization. Specifically, we have proposed a rewrite toolkit in order to prepare plan generation for XQuery. Taking a single query block as argument a plan generator creates an optimal plan for the given query block. Our toolkit provides rewrite rules in order to transform an XQuery — possibly consisting of many query blocks — in a form with fewer query blocks. Applying these rewrites, we increase the search space for the plan generator which can then (usually) generate better plans. Using an example query, we were able to demonstrate the effect of fewer query blocks on the execution time of query execution plans that would not have been possible to generate without our rewrites.

7.2 Outlook

In this thesis, we have developed an algebraic approach for the evaluation of XPath and presented optimizations for the evaluation of XPath and XQuery. Moreover, we studied the unnesting of nested SQL queries in the presence of disjunctions. Still, in all of these areas, this is just the beginning of research that needs to be done in order to employ these techniques in industrial-strength database management systems. In

the last section, we discuss outstanding topics that have not yet received any or enough attention.

7.2.1 XML Query Processing

Plan Generation Plan generation is the problem to find for a given data set a query execution plan that is the fastest to execute. Finding such a plan requires the plan generator to compare alternative (equivalent) plans. Therefore, the plan generator attaches costs to single operations of a plan and calculates, given statistical information about the data, the overall cost needed to execute it. These costs are computed using cost functions for physical algebra operators that take the mapping of the data onto storage mediums (disk or main memory) and the physical characteristics of this medium into account. For relational algebras and storage, this problem has been solved successfully. For an XML algebra and storage, no such (published) work currently exists. A major future research task is to make up a complete cost model out of physical algebra operators, statistical information about XML data (e.g. [95, 96, 108]), and functions that describe the characteristics of accessing XML data on disk or in main memory. In some cases, our unnesting techniques may not always result in better execution plans. A cost model would allow us to decide on the benefit of applying our unnesting equivalences, i.e. apply them in a cost-based manner during plan generation. Moreover, a plan generator should determine the most effective way to evaluate XPath location steps or paths among many techniques that have been proposed for this (e.g. [2, 21, 54, 66, 67]). First studies comparing the use of index-based evaluation techniques (i.e. structural joins) against the navigational technique that is used in Natix have already been presented in [79].

Schema-Based Optimization A detailed investigation of schema-based optimization should be a further point of research. Although there have already been some results presented in this area (e.g. [9, 73, 74]), there remains a lot of work to be done. We have already shown some promising rewrites in order to optimize dependent nested XPath expressions (see Sec. 3.3.3). Some of these rewrites are not applicable without knowledge about the schema of the underlying document. Techniques such as magic sets could, for example, be used to optimize dependent path expressions further.

Factorization A byproduct of the previous chapter addressed the factorization of common location steps in an XQuery. This avoids multiple evaluations of the same step. However, this is not sufficient. For instance, factorizing more complex XQuery expressions, such as predicates, node constructions, or functions, would further reduce execution costs. Moreover, in order to ensure optimal plans, factorization has to be incorporated into plan generation algorithms. First research that

focused on plan generation and efficient query execution under factorization has already be done in [87, 88, 89]. A step towards this direction would be to incorporate our unnest map operator in their plan generation algorithms.

Materialized Views Materialized views are well-studied and widely used in order to reduce the runtime of complex SQL queries. In the context of XQuery, it has already been shown how materialized XPath views can be exploited to answer user queries with XPath expressions [4]. However, their algorithms still lack some features. For example, they cannot completely decide whether views can be used if the user query contains disjunctions. Moreover, it is not sufficient to answer XPath expressions with the help of views but their approach should be extended to answer complex XQuery expressions as well.

7.2.2 Unnesting Disjunctive SQL Queries

Our unnesting techniques for nested SQL queries with disjunctions also lack some cases that need to be handled. These include, for example, (1) unnesting queries whose linking *and* correlation predicate occurs in a disjunction, (2) optimizing nested disjunctive queries in the `from` clause, (3) handling all linking operators, i.e. θ `ALL` and θ `SOME/ANY` for $\theta \in \{<, \leq, >, \geq\}$, and finally (4) unnesting queries featuring correlation predicates that refer to attributes defined in a non-adjacent query block (indirect correlation).

Moreover, since our unnesting technique creates DAG-structured algebraic expressions, we rely on effective optimization techniques for generating and executing DAG-structured query plans. A first framework for the evaluation of such plans has already been introduced in [87]. However, the algebraic expressions produced by our techniques are quite demanding and, hence, might trigger further research in this direction. Integrating techniques in order to efficiently evaluate nested queries (e.g. [50]) into a push-based algebra could also be a next step.

Appendix A

Proofs for Unnesting XPath Queries

For the following proofs let lhs denote the left-hand side and rhs the right-hand side of an equivalence. Unlike our equivalences, where e_i denotes an expression, let $e_i(i = 1, \ldots, n)$ be sequences of tuples. That is why we omit the translation function $\mathcal{T}$ in the proofs. Because the result of $\mathcal{T}$ is an algebraic expressions which in turn, when evaluated, returns a sequence of tuples.

A.1 Proof of Equivalence 3.1

$$\sigma_{e_2=\mathfrak{A}_{m:max_{cn}(e_3)}}(e_1) \quad \equiv \quad \Pi_{cn:g'}(Max_{g;m;cn'=}(\Pi_{cn':cn}(e_1) < e_2 >).g)$$

Assumptions:

- $e_1 < e_2 >= e_3$,
- e_2 is single-valued,
- e_3 independent,
- and e_2 dependent on e_1.

In a first step, we substitute the sequence e_3 on the lhs with $e_1 < e_2 >$, which is equivalent according to the assumptions.

$$\sigma_{e_2=\mathfrak{A}_{m:max_{cn}(e_1<e_2>)}}(e_1) \quad \equiv \quad \Pi_{cn:g'}(Max_{g;m;cn'=}(\Pi_{cn':cn}(e_1) < e_2 >).g)$$

Proof by Induction: over the length of the sequence e_1.
Base Case:
$e_1 = \epsilon$:
lhs = ϵ
rhs = ϵ as $\Pi_{cn':cn}(e_1) < e_2 >$ is empty and (by definition of the max operator) if its input sequence is empty.
Inductive Hypothesis:

$$\sigma_{e_2=\mathfrak{A}_{m:max_{cn}(e_1<e_2>)}}(e_1) \quad \equiv \quad \Pi_{cn:g'}(Max_{g;m;cn'=}(\Pi_{cn':cn}(e_1) < e_2 >).g)$$

Inductive Step:
$e_1 \rightarrow e_1 \oplus t$

$$\begin{aligned} & \sigma_{e_2=\mathfrak{A}_{m:max_{cn}((e_1 \oplus t)<e_2>)}}(e_1 \oplus t) \equiv \Pi_{cn:g'}(Max_{g;m;cn'=}(\Pi_{cn':cn}(e_1 \oplus t) <e_2>).g) \\ \Leftrightarrow \quad & \sigma_{e_2=\mathfrak{A}_{m:max_{cn}((e_1 \oplus t)<e_2>)}}(e_1) \oplus \sigma_{e_2=\mathfrak{A}_{m:max_{cn}((e_1 \oplus t)<e_2>)}}(t) \equiv \\ & \Pi_{cn:g'}(Max_{g;m;cn'=}(\Pi_{cn':cn}(e_1) <e_2> \oplus \Pi_{cn':cn}(t) <e_2>).g) \end{aligned}$$

Case 1: The maximum element in e_2 is not in the context of t:
lhs = $\sigma_{e_2=\mathfrak{A}_{m:max_{cn}((e_1)<e_2>)}}(e_1)$ as $\sigma_{e_2=\mathfrak{A}_{m:max_{cn}((e_1 \oplus t)<e_2>)}}(t)$ is empty and we can omit the t in the subscript $e_1 \oplus t$ of the selection because the maximum element is not in the context of t.
rhs = $\Pi_{cn:g'}(Max_{g;m;cn'=}(\Pi_{cn':cn}(e_1) <e_2>).g)$

Case 2: The maximum element in e_2 is in the context of t:

Case 2a: without duplicate maximum values:
lhs = $\sigma_{e_2=\mathfrak{A}_{m:max_{cn}((t)<e_2>)}}(t)$. We can omit e_1 because it does not contain the maximum value.
rhs = $\Pi_{cn:g'}(Max_{g;m;cn'=}(\Pi_{cn':cn}(t) <e_2>).g)$
As the maximum element in e_2 is in the context of t the lhs = t. The rhs follows from the same argument by definition of the max operator and therefore rhs = t.

Case 2b: with duplicate maximum values:
lhs = $\sigma_{e_2=\mathfrak{A}_{m:max_{cn}((e_1)<e_2>)}}(e_1) \oplus \sigma_{e_2=\mathfrak{A}_{m:max_{cn}((e_1)<e_2>)}}(t)$. We can omit the t in the subscript of the selection, because the maximum value already is in e_1.
In this case
$\Pi_{cn:g'}(Max_{g;m;cn'=}(\Pi_{cn':cn}(e_1) <e_2> \oplus \Pi_{cn':cn}(t) <e_2>).g) \equiv$
$\Pi_{cn:g'}(Max_{g;m;cn'=}(\Pi_{cn':cn}(e_1) <e_2>).g)$
$\oplus \Pi_{cn:g'}(Max_{g;m;cn'=}(\Pi_{cn':cn}(t) <e_2>).g)$

and therefore

$$\begin{aligned} rhs \quad = \quad & \Pi_{cn:g'}(Max_{g;m;cn'=}(\Pi_{cn':cn}(e_1) <e_2>).g) \\ & \oplus \Pi_{cn:g'}(Max_{g;m;cn'=}(\Pi_{cn':cn}(t) <e_2>).g) \end{aligned}$$

As we know that

$$\sigma_{e_2=\mathfrak{A}_{m:max_{cn}(e_1<e_2>)}}(e_1) \equiv \Pi_{cn:g'}(Max_{g;m;cn'=}(\Pi_{cn':cn}(e_1) <e_2>).g)$$

we have to proof that

$$\sigma_{e_2=\mathfrak{A}_{m:max_{cn}((e_1)<e_2>)}}(t) \equiv \Pi_{cn:g'}(Max_{g;m;cn'=}(\Pi_{cn':cn}(t) <e_2>).g)$$

The lhs = t because the maximum value in e_2 is in the context of t and there are duplicate maximum values. Hence, t passes the selection.
The same argument holds for the rhs by definition of the max operator and therefore rhs = t.

A.2 Proof of Equivalence 3.2

$$\sigma_{\mathfrak{A}_{x;exists(e_2\theta e_3)}}(e_1) \quad \equiv \quad \Pi^A_{cn:cn'',\overline{A}}((\nu_A(\Pi_{cn'':cn}(e_1)) < e_2 >) \ltimes_{cn\theta cn'} \Pi^D_{cn':cn}(e_3))$$

Assumptions:

- e_3 independent,
- e_2 dependent on e_1, and
- $\theta =$ ' = '.

Proof by Induction: over the length of the sequence e_1.
Base Case: $e_1 = \epsilon$: lhs = rhs = ϵ
Inductive Hypothesis:

$$\sigma_{\mathfrak{A}_{x;exists(e_2=e_3)}}(e_1) \quad \equiv \quad \Pi^A_{cn:cn'',\overline{A}}((\nu_A(\Pi_{cn'':cn}(e_1)) < e_2 >) \ltimes_{cn=cn'} \Pi^D_{cn':cn}(e_3))$$

Inductive Step:
$e_1 \rightarrow e_1 \oplus t$

$$\begin{aligned}
& \sigma_{\mathfrak{A}_{x;exists(e_2=e_3)}}(e_1 \oplus t) \equiv \Pi^A_{cn:cn'',\overline{A}}((\nu_A(\Pi_{cn'':cn}(e_1 \oplus t) < e_2 >) \ltimes_{cn=cn'} \Pi^D_{cn':cn}(e_3)) \\
\Leftrightarrow \quad & \sigma_{\mathfrak{A}_{x;exists(e_2=e_3)}}(e_1) \oplus \sigma_{\mathfrak{A}_{x;exists(e_2=e_3)}}(t) \equiv \\
& \Pi^A_{cn:cn'',\overline{A}}((\nu_A(\Pi_{cn'':cn}(e_1) < e_2 >) \ltimes_{cn=cn'} \Pi^D_{cn':cn}(e_3)) \oplus \\
& \Pi^A_{cn:cn'',\overline{A}}((t \circ [A : max(\Pi_A(\nu_A(e_1))) + 1](t) < e_2 >) \ltimes_{cn=cn'} \Pi^D_{cn':cn}(e_3))
\end{aligned}$$

As we know that

$$\sigma_{\mathfrak{A}_{x;exists(e_2=e_3)}}(e_1) \equiv \Pi^A_{cn:cn'',\overline{A}}((\nu_A(\Pi_{cn'':cn}(e_1)) < e_2 >) \ltimes_{cn=cn'} \Pi^D_{cn':cn}(e_3))$$

we have to proof that

$$\begin{aligned}
\sigma_{\mathfrak{A}_{x;exists(e_2=e_3)}}(t) \quad \equiv \quad & \Pi^A_{cn:cn'',\overline{A}}((\Pi_{cn'':cn} \\
& (\Pi_{cn'':cn}(t \circ [A : max(\Pi_A(\nu_A(e_1))) + 1](t)) < e_2 >)) \\
& \ltimes_{cn=cn'} \Pi^D_{cn':cn}(e_3))
\end{aligned}$$

Case 1: $\exists z \in e_2(t) : \exists y \in e_3 : z.cn = y.cn$
For the lhs, this means that t will pass the selection operator, so

$$\sigma_{\mathfrak{A}_{x;exists(e_2=e_3)}}(t) = t$$

For the rhs, this means that

$$\Pi^A_{cn:cn'',\overline{A}}((\Pi_{cn'':cn}(t \circ [A : max(\Pi_A(\nu_A(e_1))) + 1](t))) < e_2 >) \ltimes_{cn=cn'} \Pi^D_{cn':cn}(e_3))$$

will contain all tuples in

$$(\Pi_{cn'':cn}(t \circ [A : max(\Pi_A(\nu_A(e_1))) + 1](t)) < e_2 >).$$

As cn stems from e_2 and cn' from e_3 which is independent from e_1, these are all tuples from $(\Pi_{cn'':cn}(t \circ [A : max(\Pi_A(\nu_A(e_1))) + 1](t)) < e_2 >)$. As all tuples have the same attribute values for A, the final projection will reduce this to a single tuple t, relabel the attribute cn'', and project A away.

Case 2: $\nexists z \in e_2(t) : \nexists y \in e_3 : z.cn = y.cn$
For the lhs, this means that

$$\sigma_{\mathfrak{A}_{x;exists(e_2=e_3)}}(t) = \epsilon$$

For the rhs, this means that

$$(\Pi_{cn'':cn}(t \circ [A : max(\Pi_A(\nu_A(e_1))) + 1](t)) < e_2 >) \ltimes_{cn=cn'} \Pi^D_{cn':cn}(e_3)$$

will be empty and therefore $rhs = \epsilon$.

The proof for $\theta =$ ' $\neq$ ' is analog to the proof above.

A.3 Proof of Equivalence 3.3

$$\begin{aligned} \sigma_{\mathfrak{A}_{x;exists(not(e_2=e_3))}}(e_1) &\equiv \Pi_{cn:cn'',\overline{A}}(\sigma_{c=0}(\nu_A(e_1) \lbag\!\rtimes_{c;A=A';count} E)) \\ \text{with } E &= (\Pi_{cn'':cn}((\nu_{A'}(e_1)) < e_2 >) \ltimes_{cn=cn'} (\Pi_{cn':cn}(e_3))) \end{aligned}$$

Assumptions:

- e_3 independent,
- e_2 dependent on e_1, and
- $\theta =$ ' $=$ '.

Proof by Induction: over the length of the sequence e_1.
Base Case: $e_1 = \epsilon$: lhs = ϵ
rhs = ϵ, by definition of binary operators, if their left input is empty.
Inductive Hypothesis:

$$\begin{aligned} \sigma_{\mathfrak{A}_{x;exists(not(e_2=e_3))}}(e_1) &\equiv \Pi_{cn:cn'',\overline{A}}(\sigma_{c=0}(\nu_A(e_1) \lbag\!\rtimes_{c;A=A';count} \\ &\quad (\Pi_{cn'':cn}((\nu_{A'}(e_1)) < e_2 >) \ltimes_{cn=cn'} (\Pi_{cn':cn}(e_3))))) \end{aligned}$$

Inductive Step: $e_1 \rightarrow e_1 \oplus t$

$$\begin{aligned} & \sigma_{\mathfrak{A}_{x;exists(not(e_2=e_3))}}(e_1 \oplus t) \equiv \Pi_{cn:cn'',\overline{A}}(\sigma_{c=0}(\nu_A(e_1 \oplus t) \lbag\!\rtimes_{c;A=A';count} \\ & (\Pi_{cn'':cn}((\nu_{A'}(e_1 \oplus t)) < e_2 >) \ltimes_{cn=cn'} (\Pi_{cn':cn}(e_3))))) \\ \Leftrightarrow\ & \sigma_{\mathfrak{A}_{x;exists(not(e_2=e_3))}}(e_1) \oplus \sigma_{\mathfrak{A}_{x;exists(not(e_2=e_3))}}(t) \equiv \\ & \Pi_{cn:cn'',\overline{A}}(\sigma_{c=0}(\nu_A(e_1)) \oplus \Pi_{cn:cn'',\overline{A}}(\sigma_{c=0}(t \circ [A : max(\Pi_A(\nu_A(e_1))) + 1)) \\ & \quad \lbag\!\rtimes_{c;A=A';count} (\Pi_{cn'':cn}((\nu_{A'}(e_1)) < e_2 >) \ltimes_{cn=cn'} (\Pi_{cn':cn}(e_3))) \oplus \\ & (\Pi_{cn'':cn}((t \circ [A' : max(\Pi_{A'}(\nu_{A'}(e_1))) + 1) < e_2 >) \ltimes_{cn=cn'} (\Pi_{cn':cn}(e_3)))) \end{aligned}$$

For the grouping operator tuples in e_1 will match only other tuples in $((\nu_{A'}(e_1)) < e_2 >)$, while tuples in $(t \circ [A : max(\Pi_A(\nu_A(e_1))) + 1)$ will match only other tuples in $(t \circ [A' : max(\Pi_{A'}(\nu_{A'}(e_1))) + 1] < e_2 >$. So we can divide up the $\lbag\!\rtimes$ operator into the concatenation of two $\lbag\!\rtimes$ operators.

$$\begin{aligned}\Leftrightarrow\quad & \sigma_{\mathfrak{A}_{x;exists(not(e_2=e_3))}}(e_1) \oplus \sigma_{\mathfrak{A}_{x;exists(not(e_2=e_3))}}(t) \equiv \\ & \Pi_{cn:cn'',\overline{A}}(\sigma_{c=0}(\nu_A(e_1))\ltimes\!\!\!\curlywedge_{c;A=A';count}(\Pi_{cn'':cn}((\nu_{A'}(e_1)) < e_2 >)\ltimes_{cn=cn'} \\ & \quad (\Pi_{cn':cn}(e_3)))) \oplus \\ & \quad \Pi_{cn:cn'',\overline{A}}(\sigma_{c=0}(t \circ [A : max(\Pi_A(\nu_A(e_1))) + 1)\ltimes\!\!\!\curlywedge_{c;A=A';count} \\ & (\Pi_{cn'':cn}((t \circ [A' : max(\Pi_{A'}(\nu_{A'}(e_1))) + 1) < e_2 >)\ltimes_{cn=cn'}(\Pi_{cn':cn}(e_3))))\end{aligned}$$

As we know that

$$\begin{aligned}\sigma_{\mathfrak{A}_{x;exists(not(e_2=e_3))}}(e_1) \quad \equiv \quad & \Pi_{cn:cn'',\overline{A}}(\sigma_{c=0}(\nu_A(e_1)\ltimes\!\!\!\curlywedge_{c;A=A';count} \\ & (\Pi_{cn'':cn}((\nu_{A'}(e_1)) < e_2 >)\ltimes_{cn=cn'}(\Pi_{cn':cn}(e_3)))))\end{aligned}$$

we have to proof that

$$\begin{aligned}\sigma_{\mathfrak{A}_{x;exists(not(e_2=e_3))}}(t) \quad \equiv \quad & \Pi_{cn:cn'',\overline{A}}(\sigma_{c=0}(t \circ [A : max(\Pi_A(\nu_A(e_1))) + 1)\ltimes\!\!\!\curlywedge_{c;A=A';count} \\ & (\Pi_{cn'':cn}((t \circ [A' : max(\Pi_{A'}(\nu_{A'}(e_1))) + 1) < e_2 >) \\ & \ltimes_{cn=cn'}(\Pi_{cn':cn}(e_3))))\end{aligned}$$

Case 1: $\exists z \in e_2(t) : \exists y \in e_3 : z.cn = y.cn$
For the lhs, this means that t will not pass the selection operator. Hence,

$$\sigma_{\mathfrak{A}_{x;exists(not(e_2=e_3))}}(t) = \epsilon.$$

For the rhs, this means that t will pass the semi-join operator and the grouping operator $\ltimes\!\!\!\curlywedge$ creates a group with $c > 0$. Thus t will not pass the selection and it follows that rhs = ϵ.

Case 2: $\nexists z \in e_2(t) : \nexists y \in e_3 : z.cn = y.cn$
For the lhs, this means that t will pass the selection and therefore lhs = t.
For the rhs, this means that t will not pass the semi-join operator and the grouping operator will create a group with $c = 0$. Thus t will pass the selection and it follows that rhs = t.

The proof for $\theta =$ ' $\neq$ ' is analog to the proof above.

A.4 Proof of Equivalence 3.4

$$\sigma_{\mathfrak{A}_{x;exists(e_2 \geq e_3)}}(e_1) \quad \equiv \quad \Pi^A_{cn:cn',\overline{A}}(\sigma_{cn \geq \mathfrak{A}_{x;min_{cn}(e_3)}}((\nu_A(\Pi_{cn':cn}(e_1))) < e_2 >))$$

Assumptions:

- e_3 independent,
- e_2 dependent on e_1, and
- $\theta =$ ' $\geq$ '.

Proof by Induction: over the length of the sequence e_1.
Base Case: $e_1 = \epsilon$: lhs = ϵ
rhs = ϵ
Inductive Hypothesis:

$$\sigma_{\mathfrak{A}_{x;exists(e_2 \geq e_3)}}(e_1) \equiv \Pi^A_{cn:cn',\overline{A}}(\sigma_{cn \geq \mathfrak{A}_{x;min_{cn}(e_3)}}((\nu_A(\Pi_{cn':cn}(e_1))) < e_2 >))$$

Inductive Step: $e_1 \rightarrow e_1 \oplus t$

$$\begin{aligned}
& \sigma_{\mathfrak{A}_{x;exists(e_2 \geq e_3)}}(e_1 \oplus t) \equiv \Pi^A_{cn:cn',\overline{A}}(\sigma_{cn \geq \mathfrak{A}_{x;min_{cn}(e_3)}}((\nu_A(\Pi_{cn':cn}(e_1 \oplus t))) < e_2 >)) \\
\Leftrightarrow \quad & \sigma_{\mathfrak{A}_{x;exists(e_2 \geq e_3)}}(e_1) \oplus \sigma_{\mathfrak{A}_{x;exists(e_2 \geq e_3)}}(t) \equiv \\
& \Pi^A_{cn:cn',\overline{A}}(\sigma_{cn \geq \mathfrak{A}_{x;min_{cn}(e_3)}}((\nu_A(\Pi_{cn':cn}(e_1))) < e_2 >)) \oplus \\
& \Pi^A_{cn:cn',\overline{A}}(\sigma_{cn \geq \mathfrak{A}_{x;min_{cn}(e_3)}}(\Pi_{cn':cn}(t \circ [A : max(\Pi_A(\nu_A(e_1))) + 1](t)) < e_2 >))
\end{aligned}$$

As we know that

$$\sigma_{\mathfrak{A}_{x;exists(e_2 \geq e_3)}}(e_1) \equiv \Pi^A_{cn:cn',\overline{A}}(\sigma_{cn \geq \mathfrak{A}_{x;min_{cn}(e_3)}}((\nu_A(\Pi_{cn':cn}(e_1))) < e_2 >))$$

we have to proof that

$$\sigma_{\mathfrak{A}_{x;exists(e_2 \geq e_3)}}(t) \equiv \Pi^A_{cn:cn',\overline{A}}(\sigma_{cn \geq \mathfrak{A}_{x;min_{cn}(e_3)}}(\Pi_{cn':cn}(t \circ [A : max(\Pi_A(\nu_A(e_1))) + 1](t)) < e_2 >))$$

Case 1: $\exists z \in e_2(t) : \exists y \in e_3 : z.cn \geq y.cn$

For the lhs, this means that t will pass the selection operator. Hence,

$$\sigma_{\mathfrak{A}_{y;exists(not(e_2 \geq e_3))}}(t) = t.$$

For the rhs, this means that t will pass the selection operator. As all tuples have the same attribute values for A, the final projection will reduce this to a single tuple t and restore the original attribute cn from sequence e_1.

Case 2: $\nexists z \in e_2(t) : \nexists y \in e_3 : z.cn \geq y.cn$

For the lhs, this means that t will not pass the selection and therefore lhs = ϵ.
For the rhs, this means that t will pass the selection operator and e_3 is independent from sequence e_1. It follows that rhs = ϵ.

The proofs for $\theta \in \{>, \leq, <\}$ are analog to the above proof.

A.5 Proof of Equivalence 3.5

$$\begin{aligned}
\sigma_{\mathfrak{A}_{y;exists(not(e_2 \geq e_3))}}(e_1) &\equiv \Pi_{cn:cn'',\overline{A}}(\sigma_{c=0}(E)) \\
\text{with } E &= \nu_A(e_1) \bowtie \Gamma_{c;A=A';count}(\sigma_{cn \geq \mathfrak{A}_{x;min_{cn}(e_3)}}((\Pi_{cn'':cn}(\nu_{A'}(e_1))) < e_2 >))
\end{aligned}$$

Assumptions:

- e_3 independent,
- e_2 dependent on e_1, and
- $\theta =$ ' $\geq$ '.

Proof by Induction: over the length of the sequence e_1.
Base Case: $e_1 = \epsilon$: lhs = ϵ
rhs = ϵ, by definition of binary operators, if their left input is empty.
Inductive Hypothesis:

$$\sigma_{\mathfrak{A}_{y;exists(not(e_2 \geq e_3))}}(e_1) \equiv$$
$$\Pi_{cn:cn'',\overline{A}}(\sigma_{c=0}(\nu_A(e_1) \ltimes\!\!\!\!\cap_{c;A=A';count}(\sigma_{cn \geq \mathfrak{A}_{x;min_{cn}(e_3)}}((\Pi_{cn'':cn}(\nu_{A'}(e_1))) < e_2 >))))$$

Inductive Step: $e_1 \rightarrow e_1 \oplus t$

$$\begin{aligned}
& \sigma_{\mathfrak{A}_{y;exists(not(e_2 \geq e_3))}}(e_1 \oplus t) \equiv \Pi_{cn:cn'',\overline{A}}(\sigma_{c=0}(\nu_A(e_1 \oplus t) \\
& \ltimes\!\!\!\!\cap_{c;A=A';count}(\sigma_{cn \geq \mathfrak{A}_{x;min_{cn}(e_3)}}((\Pi_{cn'':cn}(\nu_{A'}(e_1 \oplus t))) < e_2 >)))) \\
\Leftrightarrow \quad & \sigma_{\mathfrak{A}_{y;exists(not(e_2 \geq e_3))}}(e_1) \oplus \sigma_{\mathfrak{A}_{y;exists(not(e_2 \geq e_3))}}(t) \equiv \Pi_{cn:cn'',\overline{A}}(\sigma_{c=0}(\nu_A(e_1 \oplus t) \\
& \ltimes\!\!\!\!\cap_{c;A=A';count}(\sigma_{cn \geq \mathfrak{A}_{x;min_{cn}(e_3)}}((\Pi_{cn'':cn}(\nu_{A'}(e_1 \oplus t))) < e_2 >)))) \\
\Leftrightarrow \quad & \sigma_{\mathfrak{A}_{y;exists(not(e_2 \geq e_3))}}(e_1) \oplus \sigma_{\mathfrak{A}_{y;exists(not(e_2 \geq e_3))}}(t) \equiv \\
& \Pi_{cn:cn'',\overline{A}}(\sigma_{c=0}(\nu_A(e_1))) \oplus \Pi_{cn:cn'',\overline{A}}(\sigma_{c=0}(t \circ [A : max(\Pi_A(\nu_A(e_1))) + 1](t))) \\
& \ltimes\!\!\!\!\cap_{c;A=A';count}(\sigma_{cn \geq \mathfrak{A}_{x;min_{cn}(e_3)}}((\Pi_{cn'':cn}(\nu_{A'}(e_1))) < e_2 >)) \oplus \\
& (\sigma_{cn \geq \mathfrak{A}_{x;min_{cn}(e_3)}}((\Pi_{cn'':cn}(t \circ [A' : max(\Pi_{A'}(\nu_{A'}(e_1))) + 1](t))) < e_2 >))
\end{aligned}$$

For the grouping operator tuples in

$$\Pi_{cn:cn'',\overline{A}}(\sigma_{c=0}(\nu_A(e_1)))$$

will match only other tuples in

$$(\sigma_{cn \geq \mathfrak{A}_{x;min_{cn}(e_3)}}((\Pi_{cn'':cn}(\nu_{A'}(e_1))) < e_2 >))$$

while tuples in

$$\Pi_{cn:cn'',\overline{A}}(\sigma_{c=0}(t \circ [A : max(\Pi_A(\nu_A(e_1))) + 1](t)))$$

will match only other tuples in

$$((\sigma_{cn \geq \mathfrak{A}_{x;min_{cn}(e_3)}}((\Pi_{cn'':cn}(t \circ [A' : max(\Pi_{A'}(\nu_{A'}(e_1))) + 1](t))) < e_2 >)))$$

So we can divide up the $\ltimes\!\!\!\!\cap$ operator into the concatenation of two $\ltimes\!\!\!\!\cap$ operators.

$$\begin{aligned}
\Leftrightarrow \quad & \sigma_{\mathfrak{A}_{y;exists(not(e_2 \geq e_3))}}(e_1) \oplus \sigma_{\mathfrak{A}_{y;exists(not(e_2 \geq e_3))}}(t) \equiv \\
& (\Pi_{cn:cn'',\overline{A}}(\sigma_{c=0}(\nu_A(e_1))) \ltimes\!\!\!\!\cap_{c;A=A';count}(\sigma_{cn \geq \mathfrak{A}_{x;min_{cn}(e_3)}} \\
& ((\Pi_{cn'':cn}(\nu_{A'}(e_1))) < e_2 >))) \oplus \\
& (\Pi_{cn:cn'',\overline{A}}(\sigma_{c=0}(t \circ [A : max(\Pi_A(\nu_A(e_1))) + 1](t))) \ltimes\!\!\!\!\cap_{c;A=A';count} \\
& ((\sigma_{cn \geq \mathfrak{A}_{x;min_{cn}(e_3)}}((\Pi_{cn'':cn}(t \circ [A' : max(\Pi_{A'}(\nu_{A'}(e_1))) + 1](t))) < e_2 >))))
\end{aligned}$$

As we know that the hypothesis holds it suffices to proof that

$$\begin{aligned}
\sigma_{\mathfrak{A}_{y;exists(not(e_2 \geq e_3))}}(t) \quad \equiv \quad & (\Pi_{cn:cn'',\overline{A}}(\sigma_{c=0}(t \circ [A : max(\Pi_A(\nu_A(e_1))) + 1](t))) \\
& \ltimes\!\!\!\!\cap_{c;A=A';count}(\sigma_{cn \geq \mathfrak{A}_{x;min_{cn}(e_3)}} \\
& ((\Pi_{cn'':cn}(t \circ [A' : max(\Pi_{A'}(\nu_{A'}(e_1))) + 1](t))) < e_2 >)))
\end{aligned}$$

Case 1: $\exists z \in e_2(t) : \exists w \in e_3 : z.cn \geq w.cn$
For the lhs, this means that t will not pass the selection operator. Hence,

$$\sigma_{\mathfrak{A}_{y;exists(not(e_2 \geq e_3))}}(t) = \epsilon$$

For the rhs, this means that t will pass the selection operator and the grouping operator $\bowtie\!\!\!\!\!\!\Gamma$ creates a group with $c > 0$. Thus t will not pass the final selection and it follows that rhs = ϵ.

Case 2: $\nexists z \in e_2(t) : \nexists w \in e_3 : z.cn \geq w.cn$
For the lhs, this means that t will pass the selection and therefore lhs = t.
For the rhs, this means that t will not pass the selection operator, because even the smallest $z.cn \in e_3$ is bigger than any $w.cn$. The grouping operator will create a group with $c = 0$ and t will pass the selection. The final projection will rename the original cn attribute of the sequence e_1. It follows that rhs = t.

The proofs for $\theta \in \{>, \leq, <\}$ are analog to the above proof.

A.6 Proof of Equivalence 3.6

$$\begin{aligned} \sigma_{\mathfrak{A}_{x;exists(e_2=(\sigma_p(e_3)))}}(e_1) &\equiv \sigma_{g>0}(E) \\ \text{with } E &= \Pi_{cn:cn',\overline{A}}(\Gamma_{g;=A;count\circ\sigma_p}((\nu_A(\Pi_{cn':cn}(e_1))) <e_2>)) \end{aligned}$$

Assumptions:

- e_3 independent,
- e_2 dependent on e_1, and
- $(e_1 <e_2>) = e_3$.

Proof by Induction: over the length of the sequence e_1.
Base Case: $e_1 = \epsilon$: lhs = ϵ
rhs = ϵ, by definition of the grouping operator.
Inductive Hypothesis:

$$\begin{aligned} \sigma_{\mathfrak{A}_{x;exists(e_2=(\sigma_p(e_3)))}}(e_1) \equiv\ & \sigma_{g>0}(\Pi_{cn:cn',\overline{A}}(\Gamma_{g;=A;count\circ\sigma_p}(\\ & (\nu_A(\Pi_{cn':cn}(e_1))) <e_2>))) \end{aligned}$$

Inductive Step: $e_1 \rightarrow e_1 \oplus t$

$$\begin{aligned} & \sigma_{\mathfrak{A}_{x;exists(e_2=(\sigma_p(e_3)))}}(e_1 \oplus t) \equiv \\ & \sigma_{g>0}(\Pi_{cn:cn',\overline{A}}(\Gamma_{g;=A;count\circ\sigma_p}((\nu_A(\Pi_{cn':cn}(e_1 \oplus t))) <e_2>))) \\ \Leftrightarrow\ & \sigma_{\mathfrak{A}_{x;exists(e_2=(\sigma_p(e_3)))}}(e_1) \oplus \sigma_{\mathfrak{A}_{x;exists(e_2=(\sigma_p(e_3)))}}(t) \equiv \\ & \sigma_{g>0}(\Pi_{cn:cn',\overline{A}}(\Gamma_{g;=A;count\circ\sigma_p}((\nu_A(\Pi_{cn':cn}(e_1 \oplus t))) <e_2>))) \\ \Leftrightarrow\ & \sigma_{\mathfrak{A}_{x;exists(e_2=(\sigma_p(e_3)))}}(e_1) \oplus \sigma_{\mathfrak{A}_{x;exists(e_2=(\sigma_p(e_3)))}}(t) \equiv \\ & \sigma_{g>0}(\Pi_{cn:cn',\overline{A}}(\Gamma_{g;=A;count\circ\sigma_p}((\nu_A(\Pi_{cn':cn}(e_1))) \oplus \\ & (\Pi_{cn':cn}(t \circ [A : max(\Pi_A(\nu_A(e_1))) + 1](t))) <e_2>))) \end{aligned}$$

At this point, we should actually replace the unary grouping operator (Γ) by the binary grouping operator ($\bowtie\!\!\!\!\!\Gamma$), as in the definition of the operator. As a shortcut, we skip this step, because the grouping attribute A in tuple t is different from all other attribute A from tuples in e_1. Hence, we can safely emerge the grouping operator into two grouping operators.

$$
\begin{aligned}
\Leftrightarrow \quad & \sigma_{\mathfrak{A}_{x;exists(e_2=(\sigma_p(e_3)))}}(e_1) \oplus \sigma_{\mathfrak{A}_{x;exists(e_2=(\sigma_p(e_3)))}}(t) \equiv \\
& \sigma_{g>0}(\Pi_{cn:cn',\overline{A}}(\Gamma_{g;=A;count\circ\sigma_p}((\nu_A(\Pi_{cn':cn}(e_1))) < e_2 >) \oplus \\
& (\Gamma_{g;=A;count\circ\sigma_p}(t \circ [A : max(\Pi_A(\nu_A(e_1))) + 1](t)) < e_2 >))) \\
\Leftrightarrow \quad & \sigma_{\mathfrak{A}_{x;exists(e_2=(\sigma_p(e_3)))}}(e_1) \oplus \sigma_{\mathfrak{A}_{x;exists(e_2=(\sigma_p(e_3)))}}(t) \equiv \\
& (\sigma_{g>0}(\Pi_{cn:cn',\overline{A}}(\Gamma_{g;=A;count\circ\sigma_p}((\nu_A(\Pi_{cn':cn}(e_1))) < e_2 >)))) \oplus \\
& (\sigma_{g>0}(\Pi_{cn:cn',\overline{A}}(\Gamma_{g;=A;count\circ\sigma_p}(t \circ [A : max(\Pi_A(\nu_A(e_1))) + 1](t)) < e_2 >)))
\end{aligned}
$$

As we know that the hypothesis holds it suffices to proof that

$$
\begin{aligned}
\sigma_{\mathfrak{A}_{x;exists(e_2=(\sigma_p(e_3)))}}(t) \quad \equiv \quad & (\sigma_{g>0}(\Pi_{cn:cn',\overline{A}}(\Gamma_{g;=A;count\circ\sigma_p} \\
& (t \circ [A : max(\Pi_A(\nu_A(e_1))) + 1](t)) < e_2 >)))
\end{aligned}
$$

Case 1: $\exists z \in e_2(t) : \exists y \in e_3 : \sigma_p(e_3) = z.cn$
For the lhs, this means that t will pass the selection operator. Hence,

$$\sigma_{\mathfrak{A}_{x;exists(e_2=(\sigma_p(e_3)))}}(t) = t.$$

For the rhs, this means that t will be in a group whose g value is greater than zero. In this case, t will pass the selection operator and it follows that $rhs = t$.

Case 2: $\nexists z \in e_2(t) : \nexists y \in e_3 : \sigma_p(e_3) = z.cn$
For the lhs, this means that t will not pass the selection and therefore lhs = ϵ.

For the rhs, this means that t will be in a group whose g value is zero. Hence, t will not pass the selection operator and $rhs = \epsilon$.

A.7 Proof of Equivalence 3.7

$$
\sigma_{\mathfrak{A}_{y;exists(e_2=e_3)}}(e_1) \quad \equiv \quad \Pi^A_{cn:cn',\overline{A}}(\sigma_{cn=e_3}((\nu_A(\Pi_{cn':cn}(e_1))) < e_2 >))
$$

Assumptions:

- e_3 is single-valued as well as independent,
- e_2 dependent on e_1, and
- $\theta =$ ' = '.

Proof by Induction: over the length of the sequence e_1.
Base Case: $e_1 = \epsilon$:lhs = ϵ = rhs.
Inductive Hypothesis:

$$
\sigma_{\mathfrak{A}_{y;exists(e_2=e_3)}}(e_1) \equiv \Pi^A_{cn:cn',\overline{A}}(\sigma_{cn=e_3}((\nu_A(\Pi_{cn':cn}(e_1))) < e_2 >))
$$

Inductive Step: $e_1 \rightarrow e_1 \oplus t$

$$
\begin{aligned}
& \sigma_{\mathfrak{A}_{y;exists(e_2=e_3)}}(e_1 \oplus t) \equiv \Pi^A_{cn:cn',\overline{A}}(\sigma_{cn=e_3}((\nu_A(\Pi_{cn':cn}(e_1 \oplus t))) < e_2 >)) \\
\Leftrightarrow \quad & \sigma_{\mathfrak{A}_{y;exists(e_2=e_3)}}(e_1) \oplus \sigma_{\mathfrak{A}_{y;exists(e_2=e_3)}}(t) \equiv \\
& \Pi^A_{cn:cn',\overline{A}}(\sigma_{cn=e_3}((\nu_A(\Pi_{cn':cn}(e_1 \oplus t))) < e_2 >)) \\
\Leftrightarrow \quad & \sigma_{\mathfrak{A}_{y;exists(e_2=e_3)}}(e_1) \oplus \sigma_{\mathfrak{A}_{y;exists(e_2=e_3)}}(t) \equiv \\
& \Pi^A_{cn:cn',\overline{A}}(\sigma_{cn=e_3}((\nu_A(\Pi_{cn':cn}(e_1))) < e_2 >)) \oplus \\
& \Pi^A_{cn:cn',\overline{A}}(\sigma_{cn=e_3}(\Pi_{cn':cn}((t \circ [A : max(\Pi_A(\nu_A(e_1))) + 1](t)))) < e_2 >)
\end{aligned}
$$

As we know that

$$\sigma_{\mathfrak{A}_{y;exists(e_2=e_3)}}(e_1) \equiv \Pi^A_{cn:cn',\overline{A}}(\sigma_{cn=e_3}((\nu_A(\Pi_{cn':cn}(e_1))) < e_2 >))$$

we have to proof that

$$\sigma_{\mathfrak{A}_{y;exists(e_2=e_3)}}(t) \equiv \Pi^A_{cn:cn',\overline{A}}(\sigma_{cn=e_3}(\Pi_{cn':cn}((t \circ [A : max(\Pi_A(\nu_A(e_1))) + 1](t)))) < e_2 >)$$

Case 1: $\exists z \in e_2(t) : z.cn = e_3.cn$
Remember, that e_3 is single-valued.
For the lhs, this means that t will pass the selection operator, so $lhs = t$.
For the rhs, this means that t will pass the selection operator. As all tuples have the same attribute values for A, the final projection will reduce this to a single tuple t and restore the original attribute cn from sequence e_1. It follows, that rhs = t.

Case 2: $\nexists z \in e_2(t) : z.cn = e_3.cn$
For the lhs, this means that t will not pass the selection and therefore lhs = ϵ.
For the rhs, this means that t will also not pass the selection operator and it follows, that rhs = ϵ.

The proofs for $\theta \in \{\neq, \geq, >, \leq, <\}$ are analog to the proof above.

A.8 Proof of Equivalence 3.8

$$\sigma_{\mathfrak{A}_{x;f(e_2)=e_3}}(e_1) \quad \equiv \quad \Pi_{cn:cn'',\overline{A}}(\Gamma_{g;=A;f}(\nu_A(\Pi_{cn'':cn}(e_1)) < e_2 >) \ltimes_{g=cn'} (\Pi^D_{cn':cn}(e_3)))$$

Assumptions:

- e_3 is independent,
- e_2 dependent on e_1,
- f is an aggregation function, e.g. count, and
- $\theta =$ ' = '.

Proof by Induction: over the length of the sequence e_1.
Base Case: $e_1 = \epsilon$: lhs = ϵ
rhs = ϵ, by definition of binary operators if their left input is empty.
Inductive Hypothesis:

$$\sigma_{\mathfrak{A}_{x;f(e_2)=e_3}}(e_1) \equiv \Pi_{cn:cn'',\overline{A}}(\Gamma_{g;=A;f}(\nu_A(\Pi_{cn'':cn}(e_1)) <e_2>)\ltimes_{g=cn'}(\Pi^D_{cn':cn}(e_3)))$$

Inductive Step: $e_1 \rightarrow e_1 \oplus t$

$$\begin{aligned}
& \sigma_{\mathfrak{A}_{x;f(e_2)=e_3}}(e_1 \oplus t) \equiv \Pi_{cn:cn'',\overline{A}}(\Gamma_{g;=A;f}(\nu_A(\Pi_{cn'':cn}(e_1 \oplus t)) <e_2>)\ltimes_{g=cn'} \\
& \quad (\Pi^D_{cn':cn}(e_3))) \\
\Leftrightarrow \quad & \sigma_{\mathfrak{A}_{x;f(e_2)=e_3}}(e_1) \oplus \sigma_{\mathfrak{A}_{x;f(e_2)=e_3}}(t) \equiv \Pi_{cn:cn'',\overline{A}}(\Gamma_{g;=A;f}(\nu_A(\Pi_{cn'':cn}(e_1 \oplus t)) <e_2>) \\
& \ltimes_{g=cn'}(\Pi^D_{cn':cn}(e_3))) \\
\Leftrightarrow \quad & \sigma_{\mathfrak{A}_{x;f(e_2)=e_3}}(e_1) \oplus \sigma_{\mathfrak{A}_{x;f(e_2)=e_3}}(t) \equiv \Pi_{cn:cn'',\overline{A}}((\Gamma_{g;=A;f}(\nu_A(\Pi_{cn'':cn}(e_1)) \\
& \oplus(\Pi_{cn'':cn}(t \circ [A : max(\Pi_A(\nu_A(e_1))) + 1](t))) <e_2>))\ltimes_{g=cn'}(\Pi^D_{cn':cn}(e_3)))
\end{aligned}$$

At this point, we should actually replace the unary grouping operator by the binary grouping operator, as in the definition of the operator. As a shortcut, we skip this step. Because the grouping attribute A in tuple t is different from all other attribute A from tuples in e_1, we can safely emerge the grouping operator into two grouping operators.

$$\begin{aligned}
\Leftrightarrow \quad & \sigma_{\mathfrak{A}_{x;f(e_2)=e_3}}(e_1) \oplus \sigma_{\mathfrak{A}_{x;f(e_2)=e_3}}(t) \equiv \\
& \Pi_{cn:cn'',\overline{A}}((\Gamma_{g;=A;f}(\nu_A(\Pi_{cn'':cn}(e_1)) <e_2>) \oplus \\
& \Gamma_{g;=A;f}(\Pi_{cn'':cn}(t \circ [A : max(\Pi_A(\nu_A(e_1))) + 1](t)) <e_2>))\ltimes_{g=cn'}(\Pi^D_{cn':cn}(e_3))) \\
\Leftrightarrow \quad & \sigma_{\mathfrak{A}_{x;f(e_2)=e_3}}(e_1) \oplus \sigma_{\mathfrak{A}_{x;f(e_2)=e_3}}(t) \equiv \\
& \Pi_{cn:cn'',\overline{A}}(\Gamma g; = A; f(\nu_A(\Pi_{cn'':cn}(e_1)) <e_2>)\ltimes_{g=cn'}(\Pi^D_{cn':cn}(e_3))) \oplus \\
& \Pi_{cn:cn'',\overline{A}}(\Gamma_{g;=A;f}(\Pi_{cn'':cn}(t \circ [A : max(\Pi_A(\nu_A(e_1))) + 1](t)) <e_2>)\ltimes_{g=cn'} \\
& \quad (\Pi^D_{cn':cn}(e_3)))
\end{aligned}$$

As we know that

$$\sigma_{\mathfrak{A}_{x;f(e_2)=e_3}}(e_1) \quad \equiv \quad \Pi_{cn:cn'',\overline{A}}(\Gamma g; = A; f(\nu_A(\Pi_{cn'':cn}(e_1)) <e_2>)\ltimes_{g=cn'}(\Pi^D_{cn':cn}(e_3)))$$

we have to proof that

$$\begin{aligned}
\sigma_{\mathfrak{A}_{x;f(e_2)=e_3}}(t) \quad \equiv \quad & \Pi_{cn:cn'',\overline{A}}(\Gamma_{g;=A;f}(\Pi_{cn'':cn}(t \circ [A : max(\Pi_A(\nu_A(e_1))) + 1](t)) <e_2>) \\
& \ltimes_{g=cn'}(\Pi^D_{cn':cn}(e_3)))
\end{aligned}$$

Case 1: $\exists z \in e_2(t) : \exists y \in e_3 : f(z) = y.cn$

For the lhs, this means that t will pass the selection operator, so $lhs = t$.
For the rhs, this means that the g value, computed by f using the grouping operator will match any cn value that stems from e_3 within the semi-join. So t will pass the semi-join operator. As all tuples have the same attribute values for A, the grouping operator will reduce this to a single tuple. The final projection restore the original attribute cn from sequence e_1. It follows, that rhs = t.

Case 2: $\nexists z \in e_2(t) : \nexists y \in e_3 : f(z) = y.cn$

For the lhs, this means that t will not pass the selection and therefore lhs = ϵ.
For the rhs, this means that g value, computed by f using the grouping operator will not match any cn value that stems e_3 within the semi-join. So t will not pass the semi-join operator. It follows, that rhs = ϵ.

The proofs for $\theta \in \{\neq, \geq, >, \leq, <\}$ are analog to the proof above.

A.9 Proof of Equivalence 3.9

$$\begin{aligned}\sigma_{\text{count}(\sigma_{n=c_1}(\chi_{n:f'(cn)}(\pi)))=\text{count}(\sigma_{n'=c_2}(\chi_{n':f'(cn)}(\pi)))}(e) &\equiv e <\Pi_{\overline{y,y'}}(\sigma_{y'=y}(\chi_{y':2}(\Box)\\ &\quad \rtimes^{y:2}_{true}(\Gamma_{y;=x;\text{count}}\\ &\quad (\Gamma_{x;=n;\text{count}}(E))))) >\\ E &= \sigma_{n=c_1\vee n=c_2}(\chi_{n:f'(cn)}(\pi))\end{aligned}$$

Assumptions: π dependent on e.
Proof by Induction: over the length of the sequence e.
Base Case:
$e = \epsilon$:
lhs = ϵ
rhs = ϵ by definition of the d-join if its left argument is empty.
Inductive Hypothesis:

$$\begin{aligned}\sigma_{\text{count}(\sigma_{n=c_1}(\chi_{n:f'(cn)}(\pi)))=\text{count}(\sigma_{n'=c_2}(\chi_{n':f'(cn)}(\pi)))}(e) &\equiv e <\Pi_{\overline{y,y'}}(\sigma_{y'=y}(\chi_{y':2}(\Box)\\ &\quad \rtimes^{y:2}_{true}(\Gamma_{y;=x;\text{count}}\\ &\quad (\Gamma_{x;=n;\text{count}}(E))))) >\\ E &= \sigma_{n=c_1\vee n=c_2}(\chi_{n:f'(cn)}(\pi))\end{aligned}$$

Inductive Step:
$e \rightarrow e \oplus t$

$$\begin{aligned}\sigma_{\text{count}(\sigma_{n=c_1}(\chi_{n:f'(cn)}(\pi)))=\text{count}(\sigma_{n'=c_2}(\chi_{n':f'(cn)}(\pi)))}(e \oplus t) &\equiv (e \oplus t) <\Pi_{\overline{y,y'}}(\sigma_{y'=y}(\chi_{y':2}(\Box)\\ &\quad \rtimes^{y:2}_{true}(\Gamma_{y;=x;\text{count}}\\ &\quad (\Gamma_{x;=n;\text{count}}(E))))) >\\ E &= \sigma_{n=c_1\vee n=c_2}(\chi_{n:f'(cn)}(\pi))\end{aligned}$$

$$\begin{aligned}\Leftrightarrow \sigma_{\text{count}(\sigma_{n=c_1}(\chi_{n:f'(cn)}(\pi)))=\text{count}(\sigma_{n'=c_2}(\chi_{n':f'(cn)}(\pi)))}(e) \oplus\\ \sigma_{\text{count}(\sigma_{n=c_1}(\chi_{n:f'(cn)}(\pi)))=\text{count}(\sigma_{n'=c_2}(\chi_{n':f'(cn)}(\pi)))}(t) &\equiv (e) <\Pi_{\overline{y,y'}}(\sigma_{y'=y}(\chi_{y':2}(\Box)\\ &\quad \rtimes^{y:2}_{true}(\Gamma_{y;=x;\text{count}}\\ &\quad (\Gamma_{x;=n;\text{count}}(E))))) > \oplus\\ &\quad (t) <\Pi_{\overline{y,y'}}(\sigma_{y'=y}(\chi_{y':2}(\Box)\\ &\quad \rtimes^{y:2}_{true}(\Gamma_{y;=x;\text{count}}\\ &\quad (\Gamma_{x;=n;\text{count}}(E))))) >\\ E &= \sigma_{n=c_1\vee n=c_2}(\chi_{n:f'(cn)}(\pi))\end{aligned}$$

As we know that

$$\begin{aligned}\sigma_{\text{count}(\sigma_{n=c_1}(\chi_{n:f'(cn)}(\pi)))=\text{count}(\sigma_{n'=c_2}(\chi_{n':f'(cn)}(\pi)))}(e) \quad &\equiv (e) \quad < \Pi^{\overline{}}_{\overline{y,y'}}(\sigma_{y'=y}(\chi_{y':2}(\Box) \\ &\quad \exists^{y:2}_{true}(\Gamma_{y;=x;\text{count}} \\ &\quad (\Gamma_{x;=n;\text{count}}(E))))) > \\ E \quad &= \quad \sigma_{n=c_1 \vee n=c_2}(\chi_{n:f'(cn)}(\pi))\end{aligned}$$

we have to proof that

$$\begin{aligned}\sigma_{\text{count}(\sigma_{n=c_1}(\chi_{n:f'(cn)}(\pi)))=\text{count}(\sigma_{n'=c_2}(\chi_{n':f'(cn)}(\pi)))}(t) \quad &\equiv \quad (t) < \Pi^{\overline{}}_{\overline{y,y'}}(\sigma_{y'=y}(\chi_{y':2}(\Box) \\ &\quad \exists^{y:2}_{true}(\Gamma_{y;=x;\text{count}} \\ &\quad (\Gamma_{x;=n;\text{count}}(E))))) > \\ E \quad &= \quad \sigma_{n=c_1 \vee n=c_2}(\chi_{n:f'(cn)}(\pi))\end{aligned}$$

The latter holds because every $t' \in \pi$ that passes the two selections on the lhs also passes the operators of expression E on the rhs. After the selection there exists (among all tuples $t' \in \pi$) two values for the attribute value n, i.e. c_1 and c_2. For each such value, the subsequent grouping operator creates one group. Further, it adds the number of tuples in each group to the attribute x. In a next step, the second grouping operator creates a group for each x value and also counts the tuples per group, storing the result in the attribute y. If there exist an equal number of tuples for both n-values, this grouping operator creates exactly one group having two tuples, i.e. $y = 2$. This one tuple is joined (using a left outer-join) with one tuple that contains an attribute y' with value two. The (one) tuple from the join result that passes the selection operator (comparing $y' = y$) the two count-values are equal because $y = 2$. In this case, t is in the result. In contrast, t is not contained in the result, if the tuple from the join result does not pass the selection, i.e. if the two count-values are not equal. In case the last group is empty (i.e. if there does not exist any result in pi) the outer join handles empty groups. For an empty group, the value 2 is assigned to the attribute y and, hence, the tuple t also qualifies.

Analogously, the same argument holds for the sum function.

A.10 Proof of Equivalence 3.10

$$\begin{aligned}\sigma_E((\pi)[e_2]) \quad &\equiv \quad \Pi^D_{\text{string-value}(cn)}((\pi)[e_2] < e_1 >) \\ \text{with E} \quad &= \quad \text{not}(\mathfrak{A}_{x;\text{exists}}(e_1 \ltimes ((pre :: *)|(anc :: *))[e_2]/e_1))\end{aligned}$$

Assumptions:

- e_1 is single-valued,
- e_1 is context dependent on $(\pi)[e_2]$, and
- e_2 does not contain a call to position or last.

Proof by Induction: over the length of the sequence π.
Base Case:
$\pi = \epsilon$:
lhs = ϵ
rhs = ϵ by definition of the d-join if its left argument is empty.

Inductive Hypothesis:

$$\begin{aligned}\sigma_E((\pi)[e_2]) &\equiv \Pi^D_{\text{string-value}(cn)}((\pi)[e_2] < e_1 >)\\ \text{with E} &= \text{not}(\mathfrak{A}_{x;\text{exists}}(e_1 \ltimes ((pre :: *)|(anc :: *))[e_2]/e_1))\end{aligned}$$

Inductive Step: $\pi \rightarrow \pi \oplus t$
In the following, E denotes $\text{not}(\mathfrak{A}_{x;\text{exists}}(e_1 \ltimes ((pre :: *)|(anc :: *))[e_2]/e_1))$.

$$\begin{aligned}\sigma_E((\pi \oplus t)[e_2]) &\equiv \Pi^D_{\text{string-value}(cn)}((\pi \oplus t)[e_2] < e_1 >)\\ \sigma_E((\pi)[e_2]) \oplus \sigma_E((t)[e_2]) &\equiv \Pi^D_{\text{string-value}(cn)}((\pi)[e_2] < e_1 >) \oplus\\ &\quad \Pi^D_{\text{string-value}(cn)}((t)[e_2] < e_1 >)\end{aligned}$$

The last equation holds, because e_2 does not contain a call to position or last.
As we know that

$$\sigma_E((\pi)[e_2]) \equiv \Pi^D_{\text{string-value}(cn)}((\pi)[e_2] < e_1 >)$$

we have to proof that

$$\sigma_E((t)[e_2]) \equiv \Pi^D_{\text{string-value}(cn)}((t)[e_2] < e_1 >)$$

On the lhs, suppose for the tuple t there exists a tuple in $e_1 \ltimes ((pre :: *)|(anc :: *))[e_2]/e_3$. This means, that there exists a tuple in e_1 (evaluated in the context of t) that is equal to any tuple in e_1 that is evaluated in the context of all preceding or ancestor of t that also satisfies e_2, i.e. $((pre :: *)|(anc :: *))[e_2]/e_1$. Then, the tuple t does not pass the selection because of the not function call. If there does not exist a preceding or ancestor the tuple t contributes to the result. The comparison of the semi-join is done using the string-values of the two input context nodes.

This semantics is exactly resembled by the semantics of our duplicate elimination projection Π_D that keeps the first tuple in a sequence and throws away the remaining ones. The projection also eliminates duplicates based on the string-values.

Appendix B

DTD for the University Schema

```
<!ELEMENT university (employee*|student*|lecture*|exam*)*>

<!ELEMENT employee (name, (professor|research-assistant)?)>
<!ATTLIST employee id ID #REQUIRED>

<!ELEMENT professor (degree| room| teaches*| examines*)*>
<!ELEMENT degree (#PCDATA)>
<!ELEMENT room (#PCDATA)>
<!ELEMENT teaches EMPTY>
<!ATTLIST teaches lecture IDREF #REQUIRED>
<!ELEMENT examines EMPTY>
<!ATTLIST examines lecture IDREF #REQUIRED>

<!ELEMENT research-assistant (research-topic|worksfor)*>
<!ELEMENT research-topic (#PCDATA)>
<!ELEMENT worksfor EMPTY>
<!ATTLIST worksfor professor IDREF #REQUIRED>

<!ELEMENT student (name| semester| examination* | attends)*>
<!ATTLIST student id ID #REQUIRED>
<!ELEMENT attends EMPTY>
<!ATTLIST attends lecture IDREF #REQUIRED>

<!ELEMENT name (#PCDATA)>
<!ELEMENT semester (#PCDATA)>
<!ELEMENT examination EMPTY>
<!ATTLIST examination id IDREF #REQUIRED>

<!ELEMENT lecture (helpers|title|credits|attendies|lecturer?)*>
<!ATTLIST lecture id ID #REQUIRED>
<!ELEMENT title (#PCDATA)>
<!ELEMENT credits (#PCDATA)>
<!ELEMENT lecturer EMPTY>
<!ATTLIST lecturer professor IDREF #REQUIRED>
<!ELEMENT attendies (attendee*)>
<!ELEMENT attendee EMPTY>
```

```
<!ATTLIST attendee student IDREF #REQUIRED>
<!ELEMENT helpers (helper*)>
<!ELEMENT helper EMPTY>
<!ATTLIST helper student IDREF #REQUIRED>

<!ELEMENT exam (grade|belongsto|examiner|examinee)*>
<!ATTLIST exam id ID #REQUIRED>
<!ELEMENT grade (#PCDATA)>
<!ELEMENT belongsto EMPTY>
<!ATTLIST belongsto lecture IDREF #REQUIRED>

<!ELEMENT examiner EMPTY>
<!ATTLIST examiner professor IDREF #REQUIRED>

<!ELEMENT examinee EMPTY>
<!ATTLIST examinee student IDREF #REQUIRED>
```

Appendix C

Proofs for Unnesting SQL Queries

For the following proofs let *lhs* denote the left-hand side and *rhs* the right-hand side of an equivalence.

C.1 Proof of Equivalences 5.2 and 5.6

We proof only Equivalence 5.6 repeated below. The correctness of Equivalence 5.2 follows immediately when we replace the correlation predicate $\mathrm{A}_2 = \mathrm{B}_2$ by the constant TRUE.

$$\begin{aligned} \sigma_{\exists_{\mathrm{A}_1=\mathrm{B}_1}(\sigma_{\mathrm{A}_2=\mathrm{B}_2}(\mathrm{S}))\vee p}(\mathrm{R}) &\equiv e_1 \dot{\cup} e_2 \\ e_1 &:= \sigma_p^+(\mathrm{R}) \\ e_2 &:= (\sigma_p^-(\mathrm{R}))\ltimes_{\mathrm{A}_1=\mathrm{B}_1\wedge\mathrm{A}_2=\mathrm{B}_2}\mathrm{S} \end{aligned}$$

if $\mathcal{F}(p) \subseteq \mathcal{A}(R)$, $\mathcal{A}(R) \cap \mathcal{A}(S) = \emptyset$, $A_1 \in \mathcal{A}(R)$, $B_1 \in \mathcal{A}(S)$, (for Equivalence 5.6 in addition $A_2 \in \mathcal{A}(R), B_2 \in \mathcal{A}(S)$)

$$\begin{aligned} \mathrm{rhs} &= \sigma_p^+(R) \dot{\cup} ((\sigma_p^-(R))\ltimes_{A_1=B_1\wedge A_2=B_2}(S)) \\ &= \{r|r \in R \wedge p\} \dot{\cup} \{r|r \in \{x|x \in R \wedge \neg p\} \wedge \exists s \in S \wedge r.A_1 = s.B_1 \wedge r.A_2 = s.B_2\} \\ &= \{r|r \in R \wedge p\} \dot{\cup} \{r|r \in R \wedge \neg p \wedge \exists s \in S \wedge r.A_1 = s.B_1 \wedge r.A_2 = s.B_2\} \\ &= \{r|r \in R \wedge (p \vee (\neg p \wedge \exists s \in S \wedge r.A_1 = s.B_1 \wedge r.A_2 = s.B_2))\} \\ &= \{r|r \in R \wedge (p \vee (\neg p \wedge \exists s \in \{t|t \in S \wedge r.A_2 = t.B_2\} \wedge r.A_1 = s.B_1))\} \\ &= \sigma_{p\vee\exists_{A_1=B_1}(\sigma_{A_2=B_2}(S))}(R) \\ &= \mathrm{lhs} \end{aligned}$$

Note that predicates p and $\neg p$ partition the tuples of R into two disjoint sets. We explicitly use this property in the bypass selection to avoid the creation of duplicates. Together with the short-circuit evaluation of the disjunction duplicate se-

mantics are preserved. For this reason and because all remaining operators preserve duplicates this equivalence holds for bags.

C.2 Proof of Equivalences 5.3 and 5.7

We proof only Equivalence 5.7 repeated below. The correctness of Equivalence 5.3 follows immediately when we replace the correlation predicate $\mathrm{A}_2 = \mathrm{B}_2$ by the constant TRUE.

$$\begin{aligned}\sigma_{\exists_{\mathrm{A}_1=\mathrm{B}_1}(\sigma_{\mathrm{A}_2=\mathrm{B}_2}(\mathrm{S}))\vee p}(\mathrm{R}) &\equiv e_1 \dot\cup e_2\\ e_1 &:= \mathrm{R}\ltimes^{+}_{\mathrm{A}_1=\mathrm{B}_1\wedge\mathrm{A}_2=\mathrm{B}_2}\mathrm{S}\\ e_2 &:= \sigma_p(\mathrm{R}\ltimes^{-}_{\mathrm{A}_1=\mathrm{B}_1\wedge\mathrm{A}_2=\mathrm{B}_2}\mathrm{S})\end{aligned}$$

if $\mathcal{F}(p) \subseteq \mathcal{A}(R)$, $\mathcal{A}(R) \cap \mathcal{A}(S) = \emptyset$, $A_1 \in \mathcal{A}(R)$, $B_1 \in \mathcal{A}(S)$, (for Equivalence 5.7 in addition $A_2 \in \mathcal{A}(R), B_2 \in \mathcal{A}(S)$)

$$\begin{aligned}\text{rhs} &= (R\ltimes^{+}_{A_1=B_1\wedge A_2=B_2}S) \dot\cup (\sigma_p(R\ltimes^{-}_{A_1=B_1\wedge A_2=B_2}S))\\ &= \{r|r\in R\wedge\exists s\in S\wedge r.A_1=s.B_1\wedge r.A_2=s.B_2\}\dot\cup\\ &\quad \{r|r\in R\wedge p\wedge\neg\exists s\in S\wedge r.A_1=s.B_1\wedge r.A_2=s.B_2\}\\ &= \{r|r\in R\wedge((\exists s\in S\wedge r.A_1=s.B_1\wedge r.A_2=s.B_2)\vee\\ &\qquad (p\wedge\neg\exists s\in S\wedge r.A_1=s.B_1\wedge r.A_2=s.B_2))\}\\ &= \{r|r\in R\wedge(p\vee(\exists s\in S\wedge r.A_1=s.B_1\wedge r.A_2=s.B_2))\}\\ &= \{r|r\in R\wedge(p\vee(\exists s\in\{t|t\in S\wedge r.A_2=t.B_2\}\wedge r.A_1=s.B_1))\}\\ &= \sigma_{p\vee\exists_{A_1=B_1}(\sigma_{A_2=B_2}(S))}(R)\\ &= \text{lhs}\end{aligned}$$

Again, we rely on a disjoint partitioning of R and short-circuit evaluation of the disjunction in the last step. The correctness for multisets follows for the same reasons as for Equivalences 5.2 and 5.6.

C.3 Proof of Equivalences 5.4 and 5.8

We proof only Equivalence 5.8 repeated below. The correctness of Equivalence 5.4 follows immediately when we replace the correlation predicate $\mathrm{A}_2 = \mathrm{B}_2$ by the constant TRUE.

$$\begin{aligned}\sigma_{\nexists_{\mathrm{A_1=B_1}}(\sigma_{\mathrm{A_2=B_2}}(\mathrm{S}))\vee p}(\mathrm{R}) &\equiv e_1 \dot\cup e_2\\ e_1 &:= \sigma_p^+(\mathrm{R})\\ e_2 &:= (\sigma_p^-(\mathrm{R}))\triangleright_{\mathrm{A_1=B_1\wedge A_2=B_2}}\mathrm{S}\end{aligned}$$

if $\mathcal{F}(p) \subseteq \mathcal{A}(R)$, $\mathcal{A}(R) \cap \mathcal{A}(S) = \emptyset$, $A_1 \in \mathcal{A}(R)$, $B_1 \in \mathcal{A}(S)$, (for Equivalence 5.8 in addition $A_2 \in \mathcal{A}(R), B_2 \in \mathcal{A}(S)$)

$$\begin{aligned}\mathrm{rhs} &= \sigma_p^+(R) \dot\cup ((\sigma_p^-(R))\triangleright_{A_1=B_1\wedge A_2=B_2}(S))\\ &= \{r|r\in R\wedge p\} \dot\cup \{r|r\in\{x|x\in R\wedge\neg p\}\wedge\neg\exists s\in S\wedge r.A_1=s.B_1\wedge r.A_2=s.B_2\}\\ &= \{r|r\in R\wedge p\} \dot\cup \{r|r\in R\wedge\neg p\wedge\neg\exists s\in S\wedge r.A_1=s.B_1\wedge r.A_2=s.B_2\}\\ &= \{r|r\in R\wedge(p\vee(\neg p\wedge\neg\exists s\in S\wedge r.A_1=s.B_1\wedge r.A_2=s.B_2))\}\\ &= \{r|r\in R\wedge(p\vee(\neg p\wedge\neg\exists s\in\{t|t\in S\wedge r.A_2=t.B_2\}\wedge r.A_1=s.B_1))\}\\ &= \sigma_{p\vee\nexists_{A_1=B_1}(\sigma_{A_2=B_2}(S))}(R)\\ &= \mathrm{lhs}\end{aligned}$$

Again, we rely on a disjoint partitioning of R and short-circuit evaluation of the disjunction in the last step. The correctness for multisets follows for the same reasons as for Equivalences 5.2 and 5.6.

C.4 Proof of Equivalences 5.5 and 5.9

We proof only Equivalence 5.9 repeated below. The correctness of Equivalence 5.5 follows immediately when we replace the correlation predicate $\mathrm{A_2 = B_2}$ by the constant TRUE.

$$\begin{aligned}\sigma_{\nexists_{\mathrm{A_1=B_1}}(\sigma_{\mathrm{A_2=B_2}}(\mathrm{S}))\vee p}(\mathrm{R}) &\equiv e_1 \dot\cup e_2\\ e_1 &:= \mathrm{R}\triangleright^+_{\mathrm{A_1=B_1\wedge A_2=B_2}}\mathrm{S}\\ e_2 &:= \sigma_p(\mathrm{R}\triangleright^-_{\mathrm{A_1=B_1\wedge A_2=B_2}}\mathrm{S})\end{aligned}$$

if $\mathcal{F}(p) \subseteq \mathcal{A}(R)$, $\mathcal{A}(R) \cap \mathcal{A}(S) = \emptyset$, $A_1 \in \mathcal{A}(R)$, $B_1 \in \mathcal{A}(S)$, (for Equivalence 5.9 in addition $A_2 \in \mathcal{A}(R), B_2 \in \mathcal{A}(S)$)

$$
\begin{aligned}
\text{rhs} &= (R \triangleright^{+}_{A_1=B_1 \wedge A_2=B_2} S) \dot\cup (\sigma_p(R \triangleright^{-}_{A_1=B_1 \wedge A_2=B_2} S)) \\
&= \{r | r \in R \wedge \neg\exists s \in S \wedge r.A_1 = s.B_1 \wedge r.A_2 = s.B_2\} \dot\cup \\
&\quad \{r | r \in R \wedge p \wedge \exists s \in S \wedge r.A_1 = s.B_1 \wedge r.A_2 = s.B_2\} \\
&= \{r | r \in R \wedge ((\neg\exists s \in S \wedge r.A_1 = s.B_1 \wedge r.A_2 = s.B_2) \vee \\
&\qquad (p \wedge \exists s \in S \wedge r.A_1 = s.B_1 \wedge r.A_2 = s.B_2))\} \\
&= \{r | r \in R \wedge (p \vee (\neg\exists s \in \{t | t \in S \wedge r.A_2 = t.B_2\} \wedge r.A_1 = s.B_1))\} \\
&= \sigma_{p \vee \nexists_{A_1=B_1}(\sigma_{A_2=B_2}(S))}(R) \\
&= \text{lhs}
\end{aligned}
$$

Again, we rely on a disjoint partitioning of R and short-circuit evaluation of the disjunction in the last step. The correctness for multisets follows for the same reasons as for Equivalences 5.2 and 5.6.

C.5 Proof of Equivalence 5.10

$$
\begin{aligned}
\sigma_{\exists_{\mathrm{A_1=B_1}}(\sigma_{\mathrm{A_2=B_2} \vee p}(\mathrm{S}))}(\mathrm{R}) &\equiv e_1 \dot\cup e_2 \\
e_1 &:= \mathrm{R} \ltimes^{+}_{\mathrm{A_1=B_1}} e_3 \\
e_2 &:= (\mathrm{R} \ltimes^{-}_{\mathrm{A_1=B_1}} e_3) \ltimes_{\mathrm{A_1=B_1 \wedge A_2=B_2}} (\sigma^{-}_{p}(\mathrm{S})) \\
e_3 &:= \sigma^{+}_{p}(\mathrm{S})
\end{aligned}
$$

if $\mathcal{F}(p) \subseteq \mathcal{A}(S)$, $\mathcal{A}(R) \cap \mathcal{A}(S) = \emptyset$, $A_1, A_2 \in \mathcal{A}(R)$, $B_1, B_2 \in \mathcal{A}(S)$

$$
\begin{array}{rcl}
\text{rhs} & = & (R \ltimes^{+}_{A_1=B_1} (\sigma^{+}_{p}(S))) \,\dot\cup \\
& & ((R \ltimes^{-}_{A_1=B_1} (\sigma^{+}_{p}(S))) \ltimes_{A_1=B_1 \wedge A_2=B_2} (\sigma^{-}_{p}(S))) \\
& = & \{r | r \in R \wedge \exists s \in \{t | t \in S \wedge p\} \wedge r.A_1 = s.B_1\} \,\dot\cup \\
& & \{t | t \in \{r | r \in (R \setminus \{x | x \in R \wedge \exists s \in \{t | t \in S \wedge p\} \wedge x.A_1 = s.B_1\}) \wedge \\
& & \quad \exists s \in \{y | y \in S \wedge \neg p\} \wedge t.A_1 = s.B_1 \wedge t.A_2 = s.B_2\} \\
& = & \{r | r \in R \wedge \exists s \in S \wedge r.A_1 = s.B_1 \wedge p\} \,\dot\cup \\
& & \{t | t \in \{r | r \in R \setminus \{x | x \in R \wedge \exists s \in S \wedge x.A_1 = s.B_1 \wedge p\}\} \wedge \\
& & \quad \exists s \in S \wedge \neg p \wedge t.A_1 = s.B_1 \wedge t.A_2 = s.B_2\} \\
& = & \{r | r \in R \wedge \exists s \in S \wedge r.A_1 = s.B_1 \wedge p\} \,\dot\cup \\
& & \{r | r \in (R - \{x | x \in R \wedge \exists s \in S \wedge x.A_1 = s.B_1 \wedge p\}) \wedge \\
& & \quad \exists s \in S \wedge \neg p \wedge r.A_1 = s.B_1 \wedge r.A_2 = s.B_2\} \\
& = & \{r | ((r \in R \wedge \exists s \in S \wedge r.A_1 = s.B_1 \wedge p) \vee \\
& & \quad (r \in (R - \{x | x \in R \wedge \exists s \in S \wedge x.A_1 = s.B_1 \wedge p\}) \wedge \\
& & \quad \exists s \in S \wedge \neg p \wedge r.A_1 = s.B_1 \wedge r.A_2 = s.B_2))\} \\
& \stackrel{*}{=} & \{r | r \in R \wedge ((\exists s \in S \wedge r.A_1 = s.B_1 \wedge p) \vee \\
& & (\exists s \in S \wedge \neg p \wedge r.A_1 = s.B_1 \wedge r.A_2 = s.B_2))\} \\
& = & \{r | r \in R \wedge \exists s \in S \wedge ((p \wedge r.A_1 = s.B_1) \vee (\neg p \wedge r.A_1 = s.B_1 \wedge r.A_2 = s.B_2))\} \\
& = & \{r | r \in R \wedge \exists s \in S \wedge r.A_1 = s.B_1 \wedge (p \vee (\neg p \wedge r.A_2 = s.B_2))\} \\
& = & \{r | r \in R \wedge \exists s \in \{t | t \in S \wedge (p \vee (\neg p \wedge r.A_2 = t.B_2))\} \wedge r.A_1 = s.B_1\} \\
& = & \sigma_{\exists_{A_1=B_1}(\sigma_{A_2=B_2 \vee p}(S))}(R) \\
& = & \text{lhs}
\end{array}
$$

In the last step we assume short-circuit evaluation of the disjunction. In the step marked * we rely on the more general definition of the bypass semi-join where the negative stream is defined by set difference.

$$
\begin{array}{rcl}
e_1 \ltimes^{+}_{p} e_2 & := & \{x | x \in e_1 \wedge \exists y \in e_2 \wedge p\} \\
e_1 \ltimes^{-}_{p} e_2 & := & e_1 - (e_1 \ltimes^{+}_{p} e_2)
\end{array}
$$

For the correctness of this equivalence for multisets it is important to realize that the bypass semi-join partitions the tuples of R into two disjoint streams. Either stream is correct under multiset semantics. Finally the outermost union merges both streams without changing the number of duplicates.

C.6 Proof of Equivalence 5.11

$$
\begin{array}{rcl}
\sigma_{\not\exists_{\mathrm{A_1=B_1}}(\sigma_{\mathrm{A_2=B_2}\vee p}(\mathrm{S}))}(\mathrm{R}) & \equiv & e_1 \,\dot\cup\, e_2 \\
e_1 & := & \mathrm{R}\triangleright^{+}_{\mathrm{A_1=B_1}} e_3 \\
e_2 & := & (\mathrm{R}\triangleright^{-}_{\mathrm{A_1=B_1}} e_3)\triangleright_{\mathrm{A_1=B_1}\wedge\mathrm{A_2=B_2}}(\sigma^{-}_{p}(\mathrm{S})) \\
e_3 & := & \sigma^{+}_{p}(\mathrm{S})
\end{array}
$$

if $\mathcal{F}(p) \subseteq \mathcal{A}(S)$, $\mathcal{A}(R) \cap \mathcal{A}(S) = \emptyset$, $A_1, A_2 \in \mathcal{A}(R)$, $B_1, B_2 \in \mathcal{A}(S)$

$$
\begin{array}{rcl}
\text{rhs} & = & (R\triangleright^{+}_{A_1=B_1}(\sigma^{+}_{p}(S))) \,\dot\cup \\
 & & ((R\triangleright^{-}_{A_1=B_1}(\sigma^{+}_{p}(S)))\triangleright_{A_1=B_1\wedge A_2=B_2}(\sigma^{-}_{p}(S))) \\
 & = & \{r|r \in R \wedge \neg\exists s \in \{t|t \in S \wedge p\} \wedge r.A_1 = s.B_1\} \,\dot\cup \\
 & & \{t|t \in \{r|r \in (R \setminus \{x|x \in R \wedge \neg\exists s\{t|t \in S \wedge p\} \wedge x.A_1 = s.B_1\}) \wedge \\
 & & \quad \neg\exists s \in \{y|y \in S \wedge \neg p\} \wedge t.A_1 = s.B_1 \wedge t.A_2 = s.B_2\} \\
 & = & \{r|r \in R \wedge \neg\exists s \in S \wedge r.A_1 = s.B_1 \wedge p\} \,\dot\cup \\
 & & \{t|t \in \{r|r \in R \setminus \{x|x \in R \wedge \neg\exists s \in S \wedge x.A_1 = s.B_1 \wedge p\}\} \wedge \\
 & & \quad \neg\exists s \in S \wedge \neg p \wedge t.A_1 = s.B_1 \wedge t.A_2 = s.B_2\} \\
 & = & \{r|r \in R \wedge \neg\exists s \in S \wedge r.A_1 = s.B_1 \wedge p\} \,\dot\cup \\
 & & \{r|r \in (R - \{x|x \in R \wedge \neg\exists s \in S \wedge x.A_1 = s.B_1 \wedge p\}) \wedge \\
 & & \quad \neg\exists s \in S \wedge \neg p \wedge r.A_1 = s.B_1 \wedge r.A_2 = s.B_2\} \\
 & = & \{r|((r \in R \wedge \neg\exists s \in S \wedge r.A_1 = s.B_1 \wedge p) \vee \\
 & & \quad (r \in (R - \{x|x \in R \wedge \neg\exists s \in S \wedge x.A_1 = s.B_1 \wedge p\}) \wedge \\
 & & \quad \neg\exists s \in S \wedge \neg p \wedge r.A_1 = s.B_1 \wedge r.A_2 = s.B_2))\} \\
 & \stackrel{*}{=} & \{r|r \in R \wedge ((\neg\exists s \in S \wedge r.A_1 = s.B_1 \wedge p) \vee \\
 & & (\neg\exists s \in S \wedge \neg p \wedge r.A_1 = s.B_1 \wedge r.A_2 = s.B_2))\} \\
 & = & \{r|r \in R \wedge ((\{s|s \in S \wedge r.A_1 = s.B_1 \wedge p\} = \emptyset) \vee \\
 & & (\{s|s \in S \wedge \neg p \wedge r.A_1 = s.B_1 \wedge r.A_2 = s.B_2\} = \emptyset))\} \\
 & = & \{r|r \in R \wedge (\{s|s \in S \wedge ((r.A_1 = s.B_1 \wedge p) \vee \\
 & & (\neg p \wedge r.A_1 = s.B_1 \wedge r.A_2 = s.B_2))\} = \emptyset)\} \\
 & = & \{r|r \in R \wedge \neg\exists s \in S \wedge ((p \wedge r.A_1 = s.B_1) \vee \\
 & & (\neg p \wedge r.A_1 = s.B_1 \wedge r.A_2 = s.B_2))\} \\
 & = & \{r|r \in R \wedge \neg\exists s \in S \wedge r.A_1 = s.B_1 \wedge (p \vee (\neg p \wedge r.A_2 = s.B_2))\} \\
 & = & \{r|r \in R \wedge \neg\exists s \in \{t|t \in S \wedge (p \vee (\neg p \wedge r.A_2 = t.B_2))\} \wedge r.A_1 = s.B_1\} \\
 & = & \sigma_{\neg\exists_{A_1=B_1}(\sigma_{A_2=B_2\vee p}(S))}(R) \\
 & = & \text{lhs}
\end{array}
$$

In the last step we assume short-circuit evaluation of the disjunction. In the step marked * we rely on the more general definition of the bypass antijoin where the negative stream is defined by set difference.

$$\begin{aligned} e_1 \triangleright_p^+ e_2 &:= \{x | x \in e_1 \wedge \neg \exists y \in e_2 \wedge p\} \\ e_1 \triangleright_p^- e_2 &:= e_1 - (e_1 \triangleright_p^+ e_2) \end{aligned}$$

For the correctness of this equivalence for multisets it is important to realize that the bypass anti-join partitions the tuples of R into two disjoint streams. Either stream is correct under multiset semantics. Finally the outermost union merges both streams without changing the number of duplicates.

C.7 Proof of Equivalences 5.15 and 5.16

$$\sigma_{A_1 \theta_1 f(S)}(R) = \sigma_{A_1 \theta_1 g}(\chi_{g:f(S)}(R)) \tag{C.1}$$

We observe that the equivalence holds if and only if the argument to the selections is equivalent. To see this, we rewrite the lhs as follows:

$$\begin{aligned} \text{lhs} &= \sigma_{A_1 \theta f(S)}(R) \\ &= \{r | r \in R \wedge A_1 \theta f(\{s | s \in S\})\} \\ &= \{t_{|\mathcal{A}(R)} | t \in \{r \circ [g : f(\{s | s \in S\})] | r \in R\} \wedge t.g \theta t.A_1\} \\ &= \sigma_{A_1 \theta g}(\chi_{g:f(S)}(R)) \\ &= \text{rhs} \end{aligned}$$

As a result, all proofs presented next are also valid when the result of the nested scalar query block is returned with the tuples of the outer query block. Hence, these equivalences are also valid for disjunctive correlation of nested scalar queries in the select clause.

C.7.1 Proof of Equivalence 5.15

$$\begin{aligned} \sigma_{p \vee A_1 \theta f(\sigma_{A_2=B_2}(S))}(R) &= e_1 \dot{\cup} e_2 \\ e_1 &:= \sigma_p^+(R) \\ e_2 &:= \Pi_{\mathcal{A}(R)}(\sigma_{g \theta A_1}((\sigma_p^-(R)) \ltimes^{g:f(\emptyset)}_{A_2=B_2} (\Gamma_{g;=B_2;f}(S)))) \end{aligned}$$

if $\mathcal{F}(p) \subseteq \mathcal{A}(R)$, $\mathcal{A}(R) \cap \mathcal{A}(S) = \emptyset$, A_1,$A_2 \in \mathcal{A}(R)$, $B_2 \in \mathcal{A}(S)$, $g \notin \mathcal{A}(R) \cup \mathcal{A}(S)$

We define $R' := \sigma_p^-(R) = \{r | r \in R \wedge \neg p\}$ and
$S' := \Gamma_{g;=B_2;f}(S) = \{s_{|B_2} \circ [g : G] | s \in S \wedge G = f(\{y | y \in S \wedge y.B_2 = s.B_2\})\}$.

$$
\begin{aligned}
\text{rhs} &= \sigma_p^+(R) \dot\cup \Pi_{\mathcal{A}(R)}(\sigma_{g\theta A_1}((\sigma_p^-(R)) \mathbin{\overset{g:f(\emptyset)}{\bowtie}}_{A_2=B_2} (\Gamma_{g;=B_2;f}(S)))) \\
&= \{r | r \in R \wedge p\} \dot\cup \\
&\quad \{r | r \in R' \wedge s \in S' \wedge r.A_2 = s.B_2 \wedge s.g\theta r.A_1\} \cup \\
&\quad \{r | r \in R' \wedge \neg\exists s \in S' : r.A_2 = s.B_2 \wedge \mathcal{A}(z) = \mathcal{A}(S') \wedge g \in \mathcal{A}(S') \wedge \\
&\qquad \forall a \in \mathcal{A}(S) \setminus g : (z.a : \text{NULL}) \wedge z.g : f(\emptyset) \wedge z.g\theta r.A_1\} \\
&= \{r | r \in R \wedge (p \vee (\neg p \wedge ((s \in S' \wedge r.A_2 = s.B_2 \wedge s.g\theta r.A_1) \vee \\
&\qquad (\neg\exists s \in S' : r.A_2 = s.B_2 \wedge \mathcal{A}(z) = \mathcal{A}(S') \wedge g \in \mathcal{A}(S') \wedge \\
&\qquad \forall a \in \mathcal{A}(S) \setminus g : (z.a : \text{NULL}) \wedge z.g : f(\emptyset) \wedge z.g\theta r.A_1))))\} \\
&= \{r | r \in R \wedge (p \vee (\neg p \wedge \\
&\qquad ((s \in \{t_{|B_2} \circ [g : G] | t \in S \wedge G = f(\{y | y \in S \wedge y.B_2 = t.B_2\})\} \\
&\qquad \wedge r.A_2 = s.B_2 \wedge s.g\theta r.A_1) \vee \\
&\qquad (\neg\exists s \in \{t_{|B_2} \circ [g : G] | t \in S \wedge G = f(\{y | y \in S \wedge y.B_2 = t.B_2\})\} : r.A_2 = s.B_2 \wedge \\
&\qquad \mathcal{A}(z) = \mathcal{A}(S') \wedge g \in \mathcal{A}(S') \wedge \forall a \in \mathcal{A}(S) \setminus g : (z.a : \text{NULL}) \wedge z.g : f(\emptyset) \\
&\qquad \wedge z.g\theta r.A_1))))\} \\
&= \{r | r \in R \wedge (p \vee (\neg p \wedge ((s \in S \wedge G = f(\{y | y \in S \wedge y.B_2 = s.B_2\}) \wedge \\
&\qquad r.A_2 = s.B_2 \wedge G\theta r.A_1) \vee (\neg\exists s \in S : r.A_2 = s.B_2 \wedge \mathcal{A}(z) = \mathcal{A}(S) \wedge \\
&\quad \forall a \in \mathcal{A}(S) : (z.a : \text{NULL}) \wedge G = f(\emptyset) \wedge G\theta r.A_1))))\} \\
&= \{r | r \in R \wedge (p \vee (\neg p \wedge G = f(\{y | y \in S \wedge r.A_2 = y.B_2\}) \wedge G\theta r.A_1))\} \\
&= \sigma_{p \vee A_1 \theta f(\sigma_{A_2=B_2}(S))}(R) \\
&= \text{lhs}
\end{aligned}
$$

For the correctness of this equivalence over multisets, we note that the bypass selection partitions the tuples in R into two disjoint sets. The left outer-join finds at most one match with the tuples of expression $\Gamma_{g;=B_2;f}S$ because the grouping operator creates a key on the join attribute B_2. Additionally, the left outer-join returns by definition at least one tuple for each tuple in R and thus handles the case for empty groups. Hence, duplicates in R are handled properly.

The result of the aggregation is computed correctly by the unary grouping operator. It simply precomputes the value of the aggregate function f for each value of attribute B_2 in S, i.e. for each non-empty group.

The proof of Equivalence 5.16 for the special case of equality predicates follows directly from the proof of the following Lemma. Note however, that we require A_2 in R to be a key. Otherwise, as the lemma tells us, attribute A_1 and other attributes of R are not available after binary grouping.

Lemma:

$$\Pi_{A_2 \cup g}(R \mathbin{\overset{f(\emptyset)}{\bowtie}}_{A_2=B_2} (\Gamma_{g;=B_2;f}(S))) \equiv (R) \mathbin{\overset{\Gamma}{\bowtie}}_{g;A_2=B_2;f}(S) \tag{C.2}$$

Proof:

$$
\begin{aligned}
\text{lhs} &= \{r_{|A_2} \circ s_{|g} | r \in R \wedge s \in \{t \circ [g:G] | t \in S \wedge \\
&\quad G = f(\{y | y \in S \wedge t.B_2 = y.B_2\})\} \wedge r.A_2 = s.B_2\} \cup \\
&\quad \{r_{|A_2} \circ z_{|g} | r \in R \wedge \neg\exists y \in S : r.A_2 = y.B_2 \wedge \mathcal{A}(z) = \mathcal{A}(S) \wedge g \in \mathcal{A}(S) \wedge \\
&\quad \forall a \in (\mathcal{A}(S) \setminus g) : (z.a : \text{NULL} \wedge z.g : f(\emptyset))\} \\
&= \{r_{|A_2} \circ [g:G] | r \in R \wedge s \in S \wedge r.A_2 = s.B_2 \wedge G = f(\{y | y \in S \wedge s.B_2 = y.B_2\})\} \cup \\
&\quad \{r_{|A_2} \circ [g:G] | r \in R \wedge (\neg\exists s \in S : r.A_2 = s.B_2) \wedge G = f(\emptyset)\} \\
&= \{r_{|A_2} \circ [g:G] | r \in R \wedge ((s \in S \wedge r.A_2 = s.B_2 \wedge G = f(\{y | y \in S \wedge s.B_2 = y.B_2\})) \vee \\
&\quad\quad (G = f(\emptyset) \wedge \neg\exists s \in S : r.A_2 = s.B_2))\} \\
&= \{r_{|A_2} \circ [g:G] | r \in R \wedge G = f(\{y | y \in S \wedge r.A_2 = y.B_2\})\} \\
&= (R) \ltimes\!\!\!\urcorner_{g;A_2=B_2;f}(S) \\
&= \text{rhs}
\end{aligned}
$$

C.7.2 Proof of Equivalence 5.16

$$
\begin{aligned}
\Pi_{A_1,A_2}(\sigma_{p \vee A_1\theta_1 f(\sigma_{A_2\theta_2 B_2}(S))}(R)) &= \Pi_{A_1,A_2}(e_1 \dot\cup e_2) \\
e_1 &:= \sigma_p^+(R) \\
e_2 &:= \sigma_{g\theta_1 A_1}(\sigma_p^-(R) \ltimes\!\!\!\urcorner_{g;A_2\theta_2 B_2;f}(S))
\end{aligned}
$$

if $\mathcal{F}(p) \subseteq \mathcal{A}(R)$, $\mathcal{A}(R) \cap \mathcal{A}(S) = \emptyset$, A_1,$A_2 \in \mathcal{A}(R)$, $B_2 \in \mathcal{A}(S)$, $g \notin \mathcal{A}(R) \cup \mathcal{A}(S)$ and the functional dependency $A_2 \rightarrow A_1$ holds.

$$
\begin{aligned}
\text{rhs} &= \Pi_{A_1,A_2}(\sigma_p^+(R) \dot\cup (\Pi_{\mathcal{A}(R)}(\sigma_{g\theta_1 A_1}(\sigma_p^-(R) \ltimes\!\!\!\urcorner_{g;A_2\theta_2 B_2;f}(S))))) \\
&= \{r_{|A_1,A_2} | r \in R \wedge p\} \dot\cup \{t_{|A_1,A_2} | t \in \{r \circ [g:G] | r \in R \wedge G\theta_1 A_1 \wedge \neg p \wedge \\
&\quad\quad G = f(\{y | y \in S \wedge r.A_2\theta_2 y.B_2\})\}\} \\
&= \{r_{|A_1,A_2} | r \in R \wedge (p \vee (\neg p \wedge G\theta_1 A_1 \wedge G = f(\{y | y \in S \wedge r.A_2\theta_2 y.B_2\})))\} \\
&= \{r_{|A_1,A_2} | r \in R \wedge (p \vee (\neg p \wedge G\theta_1 A_1 G = f(\{y | y \in S \wedge r.A_2\theta_2 y.B_2\})))\} \\
&= \Pi_{A_1,A_2}(\sigma_{p \vee A_1\theta_1 f(\sigma_{A_2\theta_2 B_2}(S))}(R)) \\
&= \text{lhs}
\end{aligned}
$$

For the correctness of this equivalence over multisets, we note that the bypass selection partitions the tuples in R into two disjoint sets. The returns exactly one tuple per input tuple of $\sigma_p^-(R)$ extended with the result of the aggregate function f. Thereby empty groups are initialized and handled properly. The final union merges the disjoint subsets of the bypass selection establishing the correct final result.

C.8 Proof of Equivalences 5.17 and 5.18

C.8.1 Proof of Equivalence 5.17

$$\begin{aligned}\sigma_{\mathrm{p}\vee\mathrm{A}_1\theta(\mathrm{f}(\sigma_{\mathrm{A}_2=\mathrm{B}_2}(\mathrm{S})))}(\mathrm{R}) &\equiv \Pi_{\mathcal{A}(R)}(e_1 \dot\cup e_2)\\ e_1 &:= \sigma^{+}_{\mathrm{g}\theta\mathrm{A}_1}((\mathrm{R})\ltimes^{\mathrm{g:f}(\emptyset)}_{\mathrm{A}_2=\mathrm{B}_2}(\Gamma_{\mathrm{g};=\mathrm{B}_2;\mathrm{f}}(\mathrm{S})))\\ e_2 &:= \sigma_p(\sigma^{-}_{\mathrm{g}\theta\mathrm{A}_1}((\mathrm{R})\ltimes^{\mathrm{g:f}(\emptyset)}_{\mathrm{A}_2=\mathrm{B}_2}(\Gamma_{\mathrm{g};=\mathrm{B}_2;\mathrm{f}}(\mathrm{S}))))\end{aligned}$$

if $\mathcal{F}(p) \subseteq \mathcal{A}(R)$, $\mathcal{A}(R) \cap \mathcal{A}(S) = \emptyset$, A_1,$A_2 \in \mathcal{A}(R)$, $B_2 \in \mathcal{A}(S)$, $g \notin \mathcal{A}(R) \cup \mathcal{A}(S)$

First, we simplify the following subexpression X that occurs both in e_1 and e_2:

$$\begin{aligned}X &= \Pi_{\mathcal{A}(R)\cup g}((\mathrm{R})\ltimes^{\mathrm{g:f}(\emptyset)}_{\mathrm{A}_2=\mathrm{B}_2}(\Gamma_{\mathrm{g};=\mathrm{B}_2;\mathrm{f}}(\mathrm{S})))\\ &= \{r\circ s_{|g}|r\in R\wedge s\in\{t\circ[g:G]|t\in S\wedge\\ &\quad G=f(\{y|y\in S\wedge t.B_2=y.B_2\})\}\wedge r.A_2=s.B_2\}\cup\\ &\quad \{r\circ z_{|g}|r\in R\wedge\neg\exists y\in S: r.A_2=y.B_2\wedge\mathcal{A}(z)=\mathcal{A}(S)\wedge g\in\mathcal{A}(S)\wedge\\ &\quad \forall a\in(\mathcal{A}(S)\setminus g):(z.a:\mathrm{NULL}\wedge z.g:f(\emptyset))\}\\ &= \{r\circ[g:G]|r\in R\wedge s\in S\wedge r.A_2=s.B_2\wedge G=f(\{y|y\in S\wedge s.B_2=y.B_2\})\}\cup\\ &\quad \{r\circ[g:G]|r\in R\wedge\neg\exists s\in S: r.A_2=s.B_2\wedge G=f(\emptyset)\}\\ &= \{r\circ[g:G]|r\in R\wedge((s\in S\wedge r.A_2=s.B_2\wedge G=f(\{y|y\in S\wedge s.B_2=y.B_2\}))\vee\\ &\quad (G=f(\emptyset)\wedge\neg\exists s\in S: r.A_2=s.B_2))\}\\ &= \{r\circ[g:G]|r\in R\wedge G=f(\{y|y\in S\wedge r.A_2=y.B_2\})\}\end{aligned}$$

Now we use X to complete the proof:

$$\begin{aligned}\mathrm{rhs} &= \Pi_{\mathcal{A}(R)}((\sigma^{+}_{g\theta A_1}(X))\dot\cup(\sigma_p(\sigma^{-}_{g\theta A_1}(X))))\\ &= \{x_{|\mathcal{A}(R)}|x\in X\wedge x.g\theta x.A_1\}\dot\cup\{x_{|\mathcal{A}(R)}|x\in X\wedge\neg(x.g\theta x.A_1)\wedge p\}\\ &= \{x_{|\mathcal{A}(R)}|x\in X\wedge((x.g\theta x.A_1)\vee(\neg(x.g\theta x.A_1)\wedge p))\}\\ &= \{x_{|\mathcal{A}(R)}|x\in\{r\circ[g:G]|r\in R\wedge G=f(\{y|y\in S\wedge r.A_2=y.B_2\})\}\\ &\quad \wedge((x.g\theta x.A_1)\vee(\neg(x.g\theta x.A_1)\wedge p))\}\\ &= \{r|r\in R\wedge G=f(\{y|y\in S\wedge r.A_2=y.B_2\})\wedge((G\theta r.A_1)\vee(\neg(G\theta r.A_1)\wedge p))\}\\ &= \sigma_{p\vee A_1\theta f(\sigma_{A_2=B_2}(S))}(R)\\ &= \mathrm{lhs}\end{aligned}$$

The correctness of this equivalence over multisets follows by the same argumentation as for Equivalence 5.15.

C.8.2 Proof of Equivalence 5.18

The proof of Equivalence 5.18 for θ_2 as equality predicate follows from Lemma C.2. Again the A_2 must be a key of R.

For the general case we have:

$$\begin{aligned}\Pi_{A_1,A_2}(\sigma_{\mathrm{p}\vee\mathrm{A}_1\theta_1(\mathrm{f}(\sigma_{\mathrm{A}_2\theta_2\mathrm{B}_2}(\mathrm{S})))}(\mathrm{R})) &\equiv \Pi_{A_1,A_2}(e_1 \dot\cup e_2)\\ e_1 &:= \sigma^{+}_{\mathrm{g}\theta_1\mathrm{A}_1}((\mathrm{R})\ltimes\!\!\!\urcorner_{g;\mathrm{A}_2\theta_2\mathrm{B}_2;f}(\mathrm{S}))\\ e_2 &:= \sigma_p(\sigma^{-}_{\mathrm{g}\theta_1\mathrm{A}_1}((\mathrm{R})\ltimes\!\!\!\urcorner_{g;\mathrm{A}_2\theta_2\mathrm{B}_2;f}(\mathrm{S})))\end{aligned}$$

if $\mathcal{F}(p) \subseteq \mathcal{A}(R)$, $\mathcal{A}(R) \cap \mathcal{A}(S) = \emptyset$, A_1,$A_2 \in \mathcal{A}(R)$, $B_2 \in \mathcal{A}(S)$, $g \notin \mathcal{A}(R) \cup \mathcal{A}(S)$ and the functional dependency $A_2 \rightarrow A_1$ holds.
To simplify the following expression we define X as:

$$\begin{aligned}X &= (\mathrm{R})\ltimes\!\!\!\urcorner_{g;\mathrm{A}_2\theta_2\mathrm{B}_2;f}(\mathrm{S})\\ &= \{r_{|A_1,A_2} \circ [g:G] | r \in R \wedge G = f(\{s | s \in S \wedge r.A_2\theta_2 s.B_2\})\}\end{aligned}$$

$$\begin{aligned}\mathrm{rhs} &= \Pi_{A_1,A_2}(\sigma^{+}_{\mathrm{g}\theta_1\mathrm{A}_1}((\mathrm{R})\ltimes\!\!\!\urcorner_{g;\mathrm{A}_2\theta_2\mathrm{B}_2;f}(\mathrm{S})) \dot\cup \sigma_p(\sigma^{-}_{\mathrm{g}\theta_1\mathrm{A}_1}((\mathrm{R})\ltimes\!\!\!\urcorner_{g;\mathrm{A}_2\theta_2\mathrm{B}_2;f}(\mathrm{S}))))\\ &= \{t_{A_1,A_2} | t \in (\{x | x \in X \wedge x.g\theta_1 x.A_1\} \dot\cup \{y | y \in X \wedge \neg(y.g\theta_1 y.A_1) \wedge p\})\}\\ &= \{t_{A_1,A_2} | t \in X \wedge x.g\theta_1 x.A_1 \vee p\}\\ &= \{t_{A_1,A_2} | t \in \{r_{|A_1,A_2} \circ [g:G] | r \in R \wedge G = f(\{s | s \in S \wedge r.A_2\theta_2 s.B_2\})\} \wedge\\ &\quad (G\theta_1 x.A_1 \vee p)\}\\ &= \{r_{A_1,A_2} | r \in R \wedge G = f(\{s | s \in S \wedge r.A_2\theta_2 s.B_2\}) \wedge (G\theta_1 r.A_1 \vee p)\}\\ &= \Pi_{A_1,A_2}(\sigma_{p\vee A_1\theta_1 f(\sigma_{A_2\theta_2 B_2}(S))}(R))\\ &= \mathrm{lhs}\end{aligned}$$

The correctness of this equivalence for multisets follows by the same arguments as for Equivalence 5.16.

C.9 Proof of Equivalences 5.19 and 5.20

C.9.1 Proof of Equivalence 5.19

$$\begin{aligned}\sigma_{A_1\theta f(\sigma_{A_2=B_2\vee p}(S))}(R) &= \Pi_{\mathcal{A}(R)}(\sigma_{A_1\theta g}(\chi_{g:f_O(g_1,e_2)}(e_1)))\\ e_1 &:= R\rtimes^{g_1:f_I(\emptyset)}_{A_2=B_2}(\Gamma_{g_1;=B_2;f_I}(\sigma^{-}_p(S)))\\ e_2 &:= f_I(\sigma^{+}_p(S))\end{aligned}$$

if $\mathcal{F}(p) \subseteq \mathcal{A}(S)$, $\mathcal{A}(R) \cap \mathcal{A}(S) = \emptyset$, $A_i \in \mathcal{A}(R)$, $i = 1,2$, $B_2 \in \mathcal{A}(S)$, $g, g_1 \notin \mathcal{A}(R) \cup \mathcal{A}(S)$, f is decomposable [30], i.e. there are sets X, Y, and Z, with $X = Y \dot\cup Z$ and $Y \cap Z = \emptyset$.

The scalar aggregate function $f : X \to \mathcal{N}$ is *decomposable* if there exist functions

$$\begin{aligned} f_I : X &\to \mathcal{N}' \\ f_O : \mathcal{N}', \mathcal{N}' &\to \mathcal{N} \end{aligned}$$

with $f(X) = f_O(f_I(Y), f_I(Z))$.

To simplify the expressions we define X.

$$\begin{aligned} X &= \Pi_{\mathcal{A}(R)\cup g_1}(R \mathbin{⟕}^{g_1:f_I(\emptyset)}_{A_2=B_2}(\Gamma_{g_1;=B_2;f_I}(\sigma^-_p(S)))) \\ &= \{r \circ s_{|g_1} | r \in R \wedge s \in \{t \circ [g_1 : G] | t \in S \wedge \neg p \wedge \\ &\quad G = f_I(\{y | y \in S \wedge \neg p \wedge t.B_2 = y.B_2\})\} \wedge r.A_2 = s.B_2\} \cup \\ &\quad \{r \circ z_{|g_1} | r \in R \wedge \neg\exists y \in S : r.A_2 = y.B_2 \wedge \neg p \wedge \mathcal{A}(z) = \mathcal{A}(S) \\ &\quad \wedge g_1 \in \mathcal{A}(S) \wedge \forall a \in (\mathcal{A}(S) \setminus g_1) : (z.a : \mathrm{NULL} \wedge z.g_1 : f_I(\emptyset))\} \\ &= \{r \circ [g_1 : G] | r \in R \wedge G = f_I(\{y | y \in S \wedge \neg p \wedge r.A_2 = y.B_2\})\} \cup \\ &\quad \{r \circ z_{|g_1} | r \in R \wedge \neg\exists y \in S : r.A_2 = y.B_2 \wedge \neg p \wedge \mathcal{A}(z) = \mathcal{A}(S) \wedge \\ &\quad g_1 \in \mathcal{A}(S) \wedge \forall a \in (\mathcal{A}(S) \setminus g_1) : (z.a : \mathrm{NULL} \wedge z.g_1 : f_I(\emptyset))\} \\ &= \{r \circ [g_1 : G] | r \in R \wedge G = f_I(\{y | y \in S \wedge \neg p \wedge r.A_2 = y.B_2\})\} \cup \\ &\quad \{r \circ [g_1 : G] | r \in R \wedge \neg\exists y \in S : r.A_2 = y.B_2 \wedge \neg p \wedge G = f_I(\emptyset))\} \\ &= \{r \circ [g_1 : G] | r \in R \wedge G = f_I(\{y | y \in S \wedge \neg p \wedge r.A_2 = y.B_2\})\} \end{aligned}$$

Using Equivalence C.1, this equivalence also holds without the final selection on both sides. Hence, we remove can it in the remainder of the proof.

$$\begin{aligned} \mathrm{rhs} &= \Pi_{\mathcal{A}(R)\cup g}(\chi_{g:f_O(g_1,f_I(\sigma^+_p(S)))}(X)) \\ &= \{x \circ [g : f_O(g_1, f_I(\{y | y \in S \wedge p\}))]_{|\mathcal{A}(R)\cup g} | x \in X\} \\ &= \{x \circ [g : G]_{|\mathcal{A}(R)\cup g} | x \in \{r \circ [g_1 : G] | r \in R \wedge \\ &\quad G = f_I(\{y | y \in S \wedge \neg p \wedge r.A_2 = y.B_2\})\} \wedge \\ &\quad G = f_O(g_1, f_I(\{y | y \in S \wedge p\}))\} \\ &= \{r \circ [g : G] | r \in R \wedge \\ &\quad G = f_O(f_I(\{y | y \in S \wedge \neg p \wedge r.A_2 = y.B_2\}), f_I(\{y | y \in S \wedge p\}))\} \\ &= \{r \circ [g : f(\{y | y \in S \wedge (r.A_2 = y.B_2 \vee p)\})] | r \in R\} \\ &= \chi_{g:f(\sigma_{A_2=B_2\vee p}(S))}(R) \\ &= \mathrm{lhs} \end{aligned}$$

To establish the correctness for multisets we observe that no tuple of R is duplicated. Hence, we only have to be careful that at most one tuple of S matches with every tuple of R.

In the first part of the proof tuples of $\sigma^-_p(S)$ are partitioned by the join predicate $A_2 = B_2$. Matching tuples in $\sigma^-_p(S)$ are combined to exactly one value — the

result of the aggregation function f_I. This value produces exactly one output tuple for matching tuple in R. Tuples in R that do not find any matching tuple in $\sigma_p^-(S)$ are padded with the value of the aggregation function for an empty input.

In the second part no new tuples are produced or filtered. Thus, the correct number of result tuples are produced. When we investigate the correctness of the final aggregate value we note that the tuples of S are partitioned by predicate p (resp. $\neg p$). The latter were handled properly in the first part of the proof. The former are aggregated independently in the subscript of the map operator χ. When both partitions are combined according to the definition of decomposable aggregate functions the correct aggregate value is computed.

C.9.2 Proof of Equivalence 5.20

$$\begin{aligned}\Pi_{A_1,A_2}(\sigma_{A_1\theta_1 f(\sigma_{A_2\theta_2 B_2\vee p}(S))}(R)) &= \Pi_{A_1,A_2}(\sigma_{A_1\theta_1 g}(\chi_{g:f_O(g_1,e_2)}(e_1)))\\ e_1 &:= (R)\bowtie\!\!\!\!\!\!\Gamma_{g_1;A_2\theta_2 B_2;f_I}(\sigma_p^-(S))\\ e_2 &:= f_I(\sigma_p^+(S))\end{aligned}$$

if $\mathcal{F}(p) \subseteq \mathcal{A}(S)$, $\mathcal{A}(R) \cap \mathcal{A}(S) = \emptyset$, $A_i \in \mathcal{A}(R)$, $i = 1, 2$, $B_2 \in \mathcal{A}(S)$, $g, g_1 \notin \mathcal{A}(R) \cup \mathcal{A}(S)$, the functional dependency $A_2 \rightarrow A_1$ holds, and f is decomposable as discussed in the proof for Equivalence 5.19.

Using Equivalence C.1, this equivalence also holds without the final selection on both sides. Hence, we remove can it in the remainder of the proof. In addition, we ignore the final projection as it is only needed to establish the same schema of the result tuples.

$$\begin{aligned}\text{rhs} &= \chi_{g:f_O(g_1,f_I(\sigma_p^+(S)))}((R)\bowtie\!\!\!\!\!\!\Gamma_{g_1;A_2\theta_2 B_2;f_I}(\sigma_p^-(S)))\\ &= \{x \circ [g : f_O(x.g_1, f_I(\{y|y \in S \wedge p\}))]|x \in \{r \circ [g_1 : G]|r \in R \wedge\\ &\qquad G = f_I(\{y|y \in S \wedge \neg p \wedge r.A_2\theta_2 y.B_2\})\}\\ &= \{r \circ [g : f_O(f_I(\{y|y \in S \wedge \neg p \wedge r.A_2\theta_2 y.B_2\}), f_I(\{y|y \in S \wedge p\}))]|r \in R\}\\ &= \{r \circ [g : f(\{y|y \in S \wedge (r.A_2\theta_2 y.B_2 \vee p)\})]|r \in R\}\\ &= \chi_{g:f(\sigma_{A_2\theta_2 B_2\vee p}(S))}(R)\\ &= \text{lhs}\end{aligned}$$

To establish the correctness for multisets we observe that no tuple of R is duplicated. The binary grouping takes care that for every tuple of R a single group exists.

The bypass selection partitions the tuples of S into two disjoint sets. The binary grouping matches all tuples of $\sigma_p^-(S)$ to the groups established from R. Tuples of R that do not find a join partner are properly initialized by the binary grouping. Hence exactly the same tuples of R – extended with the result of aggregation – are returned by the binary grouping operator.

In the map operator does not produce or filter any tuples. Thus, the correct number of result tuples are produced. When we investigate the correctness of the final aggregate value we note that the tuples of S are partitioned by predicate p (resp. $\neg p$). The latter were handled properly by the binary grouping operator. The former are aggregated independently in the subscript of the map operator χ. When both partitions are combined according to the definition of decomposable aggregate functions the correct aggregate value is computed.

C.10 Proof of Equivalence 5.21

$$\begin{aligned}
\sigma_{A_1\theta_1 \mathrm{f}(\sigma_{A_2\theta_2B_2\vee p}(S))}(R) &= \Pi_{\mathcal{A}(R)}(\sigma_{A_1\theta_1 g}((R')\bowtie\!\!\!\!\!\!\Gamma_{g;t1=t1';f}(\rho_{t1'\leftarrow t1}(e_1 \dot\cup e_2))))\\
R' &:= \nu_{t1}(R)\\
e_1 &:= R'\bowtie^{+}_{A_2\theta_2B_2} S\\
e_2 &:= \sigma_p(R'\bowtie^{-}_{A_2\theta_2B_2} S)
\end{aligned}$$

if $\mathcal{F}(p) \subseteq \mathcal{A}(R) \cup \mathcal{A}(S)$, $\mathcal{A}(R) \cap \mathcal{A}(S) = \emptyset$, $A_i \in \mathcal{A}(R)$, $i = 1, 2$, $B_2 \in \mathcal{A}(S)$, $g \notin \mathcal{A}(R) \cup \mathcal{A}(S)$

Let us point out that we assume the id $t1$ returned by the numbering operator is computed deterministically. For an ordered set, this might be the position of a tuple within the set, it might be a tuple identifier as it is commonly used to store tuples in relational databases, or simply the key attribute of a relation. We may only use that for $t \in R'$ and $s \in R'$ it holds that if $t.t1 = s.t1 \Rightarrow t = s$. Note that $t1$ is a key for the tuple it is generated for.

We use X as a shortcut:

$$\begin{aligned}
X &:= \rho_{t'_1\leftarrow t_1}(e_1 \dot\cup e_2)\\
&= \rho_{t'_1\leftarrow t_1}((R'\bowtie^{+}_{A_2\theta_2B_2} S) \dot\cup (\sigma_p(R'\bowtie^{-}_{A_2\theta_2B_2} S)))\\
&= \{r \circ s_{t'_1\leftarrow t_1} | r \in R' \wedge s \in S \wedge r.A_2\theta_2 s.B_2\} \dot\cup\\
&\quad \{y_{t'_1\leftarrow t_1} | y \in \{r \circ s | r \in R' \wedge s \in S \wedge \neg(r.A_2\theta_2 s.B_2)\} \wedge p\}\\
&= \{r \circ s_{t'_1\leftarrow t_1} | r \in R' \wedge s \in S \wedge r.A_2\theta_2 s.B_2\} \dot\cup\\
&\quad \{r \circ s_{t'_1\leftarrow t_1} | r \in R' \wedge s \in S \wedge \neg(r.A_2\theta_2 s.B_2) \wedge p\}\\
&= \{r \circ s_{t'_1\leftarrow t_1} | r \in R' \wedge s \in S \wedge (r.A_2\theta_2 s.B_2 \vee (\neg(r.A_2\theta_2 s.B_2) \wedge p))\}
\end{aligned}$$

Thanks to Equivalence C.1 we can safely remove the final selection on both sides. We also remove the final projection on the rhs because it is only required for syntactic reasons.

$$
\begin{array}{rcl}
\text{rhs} & = & (R')\ltimes\!\!\!\top_{g;t1=t1';f}(X) \\
& = & \{t \circ [g:G] | t \in R' \wedge G = f(\{y | y \in X \wedge t.t_1 = y.t'_1\})\} \\
& = & \{t \circ [g:G] | t \in R' \wedge G = f(\{y | y \in \{[s \circ r]_{t'_1 \leftarrow t_1} | r \in R' \wedge s \in S \wedge \\
& & \quad (r.A_2\theta_2 s.B_2 \vee (\neg(r.A_2\theta_2 s.B_2) \wedge p))\} \wedge t.t_1 = y.t'_1\})\} \\
& \stackrel{*}{=} & \{t \circ [g:G] | t \in R \wedge G = f(\{s | s \in S \wedge (t.A_2\theta_2 s.B_2 \vee (\neg(t.A_2\theta_2 s.B_2) \wedge p))\})\} \\
& \stackrel{**}{=} & \{t \circ [g:G] | t \in R \wedge G = f(\{s | s \in S \wedge (t.A_2\theta_2 s.B_2 \vee p)\})]\} \\
& = & \{t \circ [g:f(\{s | s \in S \wedge (t.A_2\theta_2 s.B_2 \vee p)\})] | t \in R\} \\
& = & \chi_{g:f(\sigma_{A_2\theta_2 B_2 \vee p}(S))}(R) \\
& = & \text{lhs}
\end{array}
$$

In the step marked * we use the bijectivity of the numbering operator mentioned in the beginning of the proof. Hence we resume with R instead of R'. In the step marked ** we make use of short circuit evaluation of $\vee$.

Note that employing the numbering operator makes this equivalence applicable for either sets or multisets.

Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[2] Shurug Al-Khalifa, H. V. Jagadish, Jignesh M. Patel, Yuqing Wu, Nick Koudas, and Divesh Srivastava. Structural joins: A primitive for efficient XML query pattern matching. In *ICDE* [61], pages 141–.

[3] Sihem Amer-Yahia, Zohra Bellahsene, Ela Hunt, Rainer Unland, and Jeffrey Xu Yu, editors. *Database and XML Technologies, 4th International XML Database Symposium, XSym 2006, Seoul, Korea, September 10-11, 2006, Proceedings*, volume 4156 of *Lecture Notes in Computer Science*. Springer, 2006.

[4] Andrey Balmin, Fatma Özcan, Kevin S. Beyer, Roberta Cochrane, and Hamid Pirahesh. A framework for using materialized xpath views in xml query processing. In Nascimento et al. [86], pages 60–71.

[5] Ziv Bar-Yossef, Marcus Fontoura, and Vanja Josifovski. On the memory requirements of XPath evaluation over XML streams. *J. Comput. Syst. Sci.*, 73(3):391–441, 2007.

[6] Denilson Barbosa, Alberto O. Mendelzon, John Keenleyside, and Kelly A. Lyons. ToXgene: a template-based data generator for XML. In Franklin et al. [42], page 616.

[7] Catriel Beeri and Yariv Tzaban. SAL: An algebra for semistructured data and XML. In *WebDB (Informal Proceedings)*, pages 37–42, 1999.

[8] Kevin S. Beyer, Roberta Cochrane, Vanja Josifovski, Jim Kleewein, George Lapis, Guy M. Lohman, Robert Lyle, Fatma Özcan, Hamid Pirahesh, Norman Seemann, Tuong C. Truong, Bert Van der Linden, Brian Vickery, and Chun Zhang. System RX: One part relational, one part XML. In Özcan [91], pages 347–358.

[9] Klemens Böhm, Karl Aberer, M. Tamer Özsu, and Kathrin Gayer. Query optimization for structured documents based on knowledge on the document type definition. In *ADL*, pages 196–205, 1998.

[10] Peter A. Boncz, Thorsten Grust, Stefan Manegold, Jan Rittinger, and Jens Teubner. Pathfinder: Relational XQuery Over Multi-Gigabyte XML Inputs In Interactive Time. Technical Report INS-E0503, CWI, March 2005. MonetDB 4.8.0, Pathfinder 0.8.0.

[11] Matthias Brantner. Algebraische Auswertung von XPath in Natix. Master's thesis, University of Mannheim, Mannheim, Germany, March 2004. (in German).

[12] Matthias Brantner, Sven Helmer, Carl-Christian Kanne, and Guido Moerkotte. Full-fledged algebraic XPath processing in Natix. In *ICDE* [62], pages 705–716.

[13] Matthias Brantner, Sven Helmer, Carl-Christian Kanne, and Guido Moerkotte. Kappa-Join: Efficient execution of existential quantification in XML query languages. In Amer-Yahia et al. [3], pages 1–15.

[14] Matthias Brantner, Sven Helmer, Carl-Christian Kanne, and Guido Moerkotte. Kappa-Join: Efficient execution of existential quantification in XML query languages. Technical report, University of Mannheim, 2006. http://madoc.bib.uni-mannheim.de/madoc/volltexte/2006/1227/.

[15] Matthias Brantner, Carl-Christian Kanne, and Guido Moerkotte. Let a single FLWOR bloom. Technical Report TR-2007-007, University of Mannheim, July 2007.

[16] Matthias Brantner, Carl-Christian Kanne, and Guido Moerkotte. Let a single FLWOR bloom (to improve XQuery plan generation). In *XSym*, 2007.

[17] Matthias Brantner, Carl-Christian Kanne, Guido Moerkotte, and Sven Helmer. Algebraic optimization of nested XPath expressions. In Liu et al. [76], page 128.

[18] Matthias Brantner, Norman May, and Guido Moerkotte. Unnesting SQL queries in the presence of disjunction. Technical report, University of Mannheim, March 2006. http://db.informatik.uni-mannheim.de/publications/TR-06-013.pdf.

[19] Matthias Brantner, Norman May, and Guido Moerkotte. Unnesting scalar sql queries in the presence of disjunction. In *ICDE* [63], pages 46–55.

[20] Tim Bray, Jean Paoli, C.M. Sperber-McQueen, Eve Maler, Francois Yergeau, and John Cowan. Extensible markup language (xml) 1.1. Technical report, World Wide Web Consortium, August 2006. W3C Recommendation.

[21] Nicolas Bruno, Nick Koudas, and Divesh Srivastava. Holistic twig joins: optimal XML pattern matching. In Franklin et al. [42], pages 310–321.

[22] François Bry. Towards an efficient evaluation of general queries: Quantifier and disjunction processing revisited. In James Clifford, Bruce G. Lindsay, and David Maier, editors, *SIGMOD Conference*, pages 193–204. ACM Press, 1989.

[23] Bin Cao and Antonio Badia. A nested relational approach to processing SQL subqueries. In Özcan [91], pages 191–202.

[24] Akmal B. Chaudhri, Rainer Unland, Chabane Djeraba, and Wolfgang Lindner, editors. *XML-Based Data Management and Multimedia Engineering - EDBT 2002 Workshops, EDBT 2002 Workshops XMLDM, MDDE, and YRWS, Prague, Czech Republic, March 24-28, 2002, Revised Papers*, volume 2490 of *Lecture Notes in Computer Science*. Springer, 2002.

[25] James Clark and Steve DeRose. XML path language (XPath) version 1.0. Technical report, World Wide Web Consortium (W3C) Recommendation, 1999.

[26] Jens Claußen, Alfons Kemper, Guido Moerkotte, and Klaus Peithner. Optimizing queries with universal quantification in object-oriented and object-relational databases. In Matthias Jarke, Michael J. Carey, Klaus R. Dittrich, Frederick H. Lochovsky, Pericles Loucopoulos, and Manfred A. Jeusfeld, editors, *VLDB*, pages 286–295. Morgan Kaufmann, 1997.

[27] Jens Claußen, Alfons Kemper, Guido Moerkotte, Klaus Peithner, and Michael Steinbrunn. Optimization and evaluation of disjunctive queries. *IEEE Trans. Knowl. Data Eng.*, 12(2):238–260, 2000.

[28] Sophie Cluet and Guido Moerkotte. Classification and optimization of nested queries in object bases. In Nicole Bidoit, editor, *BDA*. INRIA, 1994.

[29] Sophie Cluet and Guido Moerkotte. Classification and optimization of nested queries in object bas es. Technical Report 95-6, RWTH Aachen, 1995.

[30] Sophie Cluet and Guido Moerkotte. Efficient evaluation of aggregates on bulk types. In Paolo Atzeni and Val Tannen, editors, *DBPL*, Electronic Workshops in Computing, page 8. Springer, 1995.

[31] C. J. Date. The outer join. In *ICOD*, pages 76–106, 1983.

[32] 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, *VLDB*, pages 197–208. Morgan Kaufmann, 1987.

[33] Alin Deutsch, Yannis Papakonstantinou, and Yu Xu. The next logical framework for xquery. In Nascimento et al. [86], pages 168–179.

[34] Yanlei Diao, Peter M. Fischer, Michael J. Franklin, and Raymond To. YFilter: Efficient and scalable filtering of XML documents. In *ICDE* [61], pages 341–.

[35] Yanlei Diao, Daniela Florescu, Donald Kossmann, Michael J. Carey, and Michael J. Franklin. Implementing memoization in a streaming XQuery processor. In Zohra Bellahsene, Tova Milo, Michael Rys, Dan Suciu, and Rainer Unland, editors, *XSym*, volume 3186 of *Lecture Notes in Computer Science*, pages 35–50. Springer, 2004.

[36] Mostafa Elhemali, César A. Galindo-Legaria, Torsten Grabs, and Milind Joshi. Execution strategies for sql subqueries. In Chee Yong Chan, Beng Chin Ooi, and Aoying Zhou, editors, *SIGMOD Conference*, pages 993–1004. ACM, 2007.

[37] Draper et al. XQuery 1.0 and XPath 2.0 formal semantics. Technical report, World Wide Web Consortium, January 2007. W3C Recommendation.

[38] Scott Boag et al. XQuery 1.0: An XML query language. Technical report, World Wide Web Consortium, January 2007. W3C Recommendation.

[39] Leonidas Fegaras. Query unnesting in object-oriented databases. In Laura M. Haas and Ashutosh Tiwary, editors, *SIGMOD Conference*, pages 49–60. ACM Press, 1998.

[40] 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 J.*, 11(4):292–314, 2002.

[41] Daniela Florescu and Donald Kossmann. Storing and querying XML data using an RDMBS. *IEEE Data Eng. Bull.*, 22(3):27–34, 1999.

[42] Michael J. Franklin, Bongki Moon, and Anastassia Ailamaki, editors. *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin, June 3-6, 2002*. ACM, 2002.

[43] César A. Galindo-Legaria and Milind Joshi. Orthogonal optimization of subqueries and aggregation. In *SIGMOD Conference*, pages 571–581, 2001.

[44] Richard A. Ganski and Harry K. T. Wong. Optimization of nested sql queries revisited. In Umeshwar Dayal and Irving L. Traiger, editors, *SIGMOD Conference*, pages 23–33. ACM Press, 1987.

[45] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database Systems: The Complete Book*. Prentice Hall, 2002. 0-13-098043-9.

[46] Georg Gottlob, Christoph Koch, and Reinhard Pichler. Efficient algorithms for processing XPath queries. In *VLDB*, pages 95–106. Morgan Kaufmann, 2002.

[47] Georg Gottlob, Christoph Koch, and Reinhard Pichler. XPath query evaluation: Improving time and space efficiency. In *ICDE*, pages 379–390, 2003.

[48] Georg Gottlob, Christoph Koch, and Reinhard Pichler. Efficient algorithms for processing XPath queries. *ACM Trans. Database Syst.*, 30(2):444–491, 2005.

[49] Goetz Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2):73–170, 1993.

[50] Goetz Graefe. Executing nested queries. In Gerhard Weikum, Harald Schöning, and Erhard Rahm, editors, *BTW*, volume 26 of *LNI*, pages 58–77. GI, 2003.

[51] Todd J. Green, Ashish Gupta, Gerome Miklau, Makoto Onizuka, and Dan Suciu. Processing XML streams with deterministic automata and stream indexes. *ACM Trans. Database Syst.*, 29(4):752–788, 2004.

[52] Torsten Grust, Sherif Sakr, and Jens Teubner. XQuery on SQL hosts. In Nascimento et al. [86], pages 252–263.

[53] Torsten Grust and Maurice van Keulen. Tree awareness for relational DBMS kernels: Staircase Join. In Henk M. Blanken, Torsten Grabs, Hans-Jörg Schek, Ralf Schenkel, and Gerhard Weikum, editors, *Intelligent Search on XML Data*, volume 2818 of *Lecture Notes in Computer Science*, pages 231–245. Springer, 2003.

[54] Torsten Grust, Maurice van Keulen, and Jens Teubner. Staircase join: Teach a relational dbms to watch its (axis) steps. In *VLDB*, pages 524–525, 2003.

[55] Torsten Grust, Maurice van Keulen, and Jens Teubner. Accelerating XPath evaluation in any RDBMS. *ACM Trans. Database Syst.*, 29:91–131, 2004.

[56] Ravindra Guravannavar, H. S. Ramanujam, and S. Sudarshan. Optimizing nested queries with parameter sort orders. In Klemens Böhm, Christian S. Jensen, Laura M. Haas, Martin L. Kersten, Per-Åke Larson, and Beng Chin Ooi, editors, *VLDB*, pages 481–492. ACM, 2005.

[57] Joseph M. Hellerstein and Jeffrey F. Naughton. Query execution techniques for caching expensive methods. In H. V. Jagadish and Inderpal Singh Mumick, editors, *SIGMOD Conference*, pages 423–434. ACM Press, 1996.

[58] Sven Helmer, Carl-Christian Kanne, and Guido Moerkotte. Optimized translation of XPath into algebraic expressions parameterized by programs containing navigational primitives. In Tok Wang Ling, Umeshwar Dayal, Elisa Bertino, Wee Keong Ng, and Angela Goh, editors, *WISE*, pages 215–224. IEEE Computer Society, 2002.

[59] Jan Hidders and Philippe Michiels. Avoiding unnecessary ordering operations in XPath. In Georg Lausen and Dan Suciu, editors, *DBPL*, volume 2921 of *Lecture Notes in Computer Science*, pages 54–70. Springer, 2003.

[60] Jan Hidders, Philippe Michiels, Jérôme Siméon, and Roel Vercammen. How to recognise different kinds of tree patterns from quite a long way away. In *PLAN-X*, pages 14–24, 2007.

[61] IEEE Computer Society. *Proceedings of the 18th International Conference on Data Engineering, 26 February - 1 March 2002, San Jose, CA*. IEEE Computer Society, 2002.

[62] IEEE Computer Society. *Proceedings of the 21st International Conference on Data Engineering, ICDE 2005, 5-8 April 2005, Tokyo, Japan*. IEEE Computer Society, 2005.

[63] IEEE Computer Society. *Proceedings of the 23nd International Conference on Data Engineering, ICDE 2007, April 15-20, 2007, The Marmara Hotel, Istanbul, Turkey*. IEEE Computer Society, 2007.

[64] H. V. Jagadish, Laks V. S. Lakshmanan, Divesh Srivastava, and Keith Thompson. Tax: A tree algebra for XML. In Giorgio Ghelli and Gösta Grahne, editors, *DBPL*, volume 2397 of *Lecture Notes in Computer Science*, pages 149–164. Springer, 2001.

[65] Matthias Jarke and Jürgen Koch. Query optimization in database systems. *ACM Comput. Surv.*, 16(2):111–152, 1984.

[66] Vanja Josifovski, Marcus Fontoura, and Attila Barta. Querying XML streams. *VLDB J.*, 14(2):197–210, 2005.

[67] Carl-Christian Kanne, Matthias Brantner, and Guido Moerkotte. Cost-sensitive reordering of navigational primitives. In Özcan [91], pages 742–753.

[68] Alfons Kemper, Guido Moerkotte, Klaus Peithner, and Michael Steinbrunn. Optimizing disjunctive queries with expensive predicates. In Richard T. Snodgrass and Marianne Winslett, editors, *SIGMOD Conference*, pages 336–347. ACM Press, 1994.

[69] Wolfgang Kiessling. SQL-like and Quel-like correlation queries with aggregates revisited. ERL/UCB Memo 84/75, University of Berkeley, 1984.

[70] Won Kim. On optimizing an SQL-like nested query. *ACM Trans. Database Syst.*, 7(3):443–469, 1982.

[71] Anthony C. Klug. Equivalence of relational algebra and relational calculus query languages having aggregate functions. *J. ACM*, 29(3):699–717, 1982.

[72] Christoph Koch. XMLTaskForce XPath evaluator, 2004. Released 2004-09-30.

[73] April Kwong and Michael Gertz. Schema-based optimization of XPath expressions. Technical report, University of California Davis, 2002.

[74] Michael Y. Levin and Benjamin C. Pierce. Type-based optimization for regular patterns. In Gavin M. Bierman and Christoph Koch, editors, *DBPL*, volume 3774 of *Lecture Notes in Computer Science*, pages 184–198. Springer, 2005.

[75] Michael Ley. DBLP XML records. http://dblp.uni-trier.de/xml/.

[76] 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.

[77] Zhen Hua Liu, Muralidhar Krishnaprasad, and Vikas Arora. Native XQuery processing in Oracle XMLDB. In Özcan [91], pages 828–833.

[78] Christian Mathis. Integrating structural joins into a tuple-based XPath algebra. In Alfons Kemper, Harald Schöning, Thomas Rose, Matthias Jarke, Thomas Seidl, Christoph Quix, and Christoph Brochhaus, editors, *BTW*, volume 103 of *LNI*, pages 242–261. GI, 2007.

[79] Norman May, Matthias Brantner, Alexander Böhm, Carl-Christian Kanne, and Guido Moerkotte. Index vs. navigation in XPath evaluation. In Amer-Yahia et al. [3], pages 16–30.

[80] 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, 2004.

[81] Norman May, Sven Helmer, and Guido Moerkotte. Nested queries and quantifiers in an ordered context. In *ICDE*, pages 239–250. IEEE Computer Society, IEEE Computer Society, 2004.

[82] Norman May, Sven Helmer, and Guido Moerkotte. Strategies for query unnesting in XML databases. *ACM Trans. Database Syst.*, 31(3):968–1013, 2006.

[83] 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, *XSym*, volume 3671 of *Lecture Notes in Computer Science*, pages 162–176. Springer, 2005.

[84] Philippe Michiels, George A. Mihaila, and Jérôme Siméon. Put a tree pattern in your algebra. In *ICDE* [63], pages 246–255.

[85] M. Muralikrishna. Improved unnesting algorithms for join aggregate sql queries. In Li-Yan Yuan, editor, *VLDB*, pages 91–102. Morgan Kaufmann, 1992.

[86] 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.

[87] Thomas Neumann. *Efficient Generation and Execution of DAG-Structured Query Graphs*. PhD thesis, University of Mannheim, Germany, 2005.

[88] Thomas Neumann, Sven Helmer, and Guido Moerkotte. On the optimal ordering of maps and selections under factorization. In *ICDE* [62], pages 490–501.

[89] Thomas Neumann, Sven Helmer, and Guido Moerkotte. On the optimal ordering of maps, selections, and joins under factorization. In *BNCOD*, pages 115–126, 2006.

[90] Dan Olteanu, Holger Meuss, Tim Furche, and François Bry. XPath: Looking forward. In Chaudhri et al. [24], pages 109–127.

[91] Fatma Özcan, editor. *Proceedings of the ACM SIGMOD International Conference on Management of Data, Baltimore, Maryland, USA, June 14-16, 2005*. ACM, 2005.

[92] Shankar Pal, Istvan Cseri, Gideon Schaller, Oliver Seeliger, Leo Giakoumakis, and Vasili Vasili Zolotov. Indexing XML data stored in a relational database. In Nascimento et al. [86], pages 1134–1145.

[93] Stelios Paparizos, Shurug Al-Khalifa, H. V. Jagadish, Laks V. S. Lakshmanan, Andrew Nierman, Divesh Srivastava, and Yuqing Wu. Grouping in XML. In Chaudhri et al. [24], pages 128–147.

[94] Hamid Pirahesh, Joseph M. Hellerstein, and Waqar Hasan. Extensible/rule based query rewrite optimization in starburst. In Michael Stonebraker, editor, *SIGMOD Conference*, pages 39–48. ACM Press, 1992.

[95] Neoklis Polyzotis and Minos N. Garofalakis. Xcluster synopses for structured xml content. In Liu et al. [76], page 63.

[96] Neoklis Polyzotis and Minos N. Garofalakis. Xsketch synopses for xml data graphs. *ACM Trans. Database Syst.*, 31(3):1014–1063, 2006.

[97] P. Roy. Optimization of DAG-structured query evaluation plans. Master's thesis, Indian Institute of Technology, Bombay, 1998.

[98] Carlo Sartiani and Antonio Albano. Yet another query algebra for XML data. In Mario A. Nascimento, M. Tamer Özsu, and Osmar R. Zaïane, editors, *IDEAS*, pages 106–115. IEEE Computer Society, 2002.

[99] Praveen Seshadri, Hamid Pirahesh, and T. Y. Cliff Leung. Complex query decorrelation. In Stanley Y. W. Su, editor, *ICDE*, pages 450–458. IEEE Computer Society, 1996.

[100] James R. Slagle. An efficient algorithm for finding certain minimum-cost procedures for making binary decisions. *J. ACM*, 11(3):253–264, 1964.

[101] Hennie J. Steenhagen. *Optimization of Object Query Languages.* PhD thesis, Department of Computer Science, University of Twente, 1995.

[102] Hennie J. Steenhagen, Peter M. G. Apers, Henk M. Blanken, and Rolf A. de By. From nested-loop to join queries in OODB. In Jorge B. Bocca, Matthias Jarke, and Carlo Zaniolo, editors, *VLDB*, pages 618–629. Morgan Kaufmann, 1994.

[103] Jens Teuber. *Pathfinder: XQuery Compilation Techniques for Relational Database Targets*. PhD thesis, Technische Universität München, Germany, October 2006.

[104] TPC. TPC benchmark H (decision support). Standard Specification Version 2.3.0, Transaction Processing Performance Council, 2006. http://www.tpc.org/tpch/.

[105] TPC. TPC benchmark DS (decision support). Preliminary Draft Version V 32, Transaction Processing Performance Council, 2007. http://www.tpc.org/tpcds/.

[106] Ning Zhang, Shishir Agrawal, and M. Tamer Özsu. Blossomtree: Evaluating XPaths in FLWOR expressions. In *ICDE* [63], pages 388–389.

[107] Ning Zhang and M. Tamer Özsu. Optimizing correlated path queries in XML languages. Technical report, University of Waterloo, 2002. TR-CS-2002-36.

[108] Ning Zhang, M. Tamer Özsu, Ashraf Aboulnaga, and Ihab F. Ilyas. XSEED: Accurate and fast cardinality estimation for XPath queries. In Liu et al. [76], page 61.