Strategies for Query Unnesting in XML Databases

NORMAN MAY
University of Mannheim
SVEN HELMER
University of London
and
GUIDO MOERKOTTE
University of Mannheim

Queries formulated in a nested way are very common in XQuery. Unfortunately, their evaluation is usually very inefficient when done in a straightforward fashion. We present a framework for handling nested queries that is based on unnesting the queries after having translated them into an algebra. We not only present a collection of algebraic equivalences, but also supply a strategy on how to use them effectively. The full potential of the approach is demonstrated by applying our rewrites to actual queries and showing that performance gains of several orders of magnitude are possible.

Categories and Subject Descriptors: H.2.4 [**Database Management**]: Systems—*Query processing*

General Terms: Theory, Experimentation, Performance

Additional Key Words and Phrases: Nested queries, query decorrelation, query optimization, XML, XQuery

1. INTRODUCTION

We consider unnesting nested queries in the context of XML databases. Like other current declarative query languages (e.g., SQL, OQL), XQuery allows for nested query blocks. For example, look at the following query which contains a nested existentially quantified query block:

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

This article extends previous work that was published in May et al. [2004b].
Authors' addresses: N. May and G. Moerkotte, University of Mannheim, Chair of Practical Computer Science III, B6, 29, 68131 Mannheim, Germany; email: {norman,moer}@pi3.informatik.uni-mannheim.de; S. Helmer, School of Computer Science and Information Systems, Birkbeck College, Malet Street, London WC1E 7HX, U.K.; email: sven@dcs.bbk.ac.uk.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2006 ACM 0362-5915/06/0900-0968 $5.00

```
return
  <book-with-review>
     { $t1 }
  </book-with-review>
```

The naive evaluation of such queries in a nested-loop fashion is very inefficient. It was observed earlier for other query languages that, by rewriting nested queries in such a way as to eliminate nested subexpressions (i.e., unnesting the query), performance gains of several orders of magnitude were possible. This was due to the change from nested-loop evaluation to an evaluation that used the more efficient join operators. This, together with a pipelined processing via the iterator principle and avoidance of multiple evaluations of the nested expression, led to the gains mentioned before. Our goal for the evaluation of nested queries in XQuery is to reach similar performance gains as demonstrated for other query languages.

There are two main approaches for unnesting queries. One works on the source level of the query language, while the other operates on algebraic expressions (after having translated the query). The first approaches for unnesting SQL worked on the source level, but had the problem of limited expressiveness of SQL. While this has been solved in the meantime, the approach of unnesting algebraic expressions still gains more and more ground on account of several advantages. First, it is easier to show the formal correctness for the rewrite rules (it took quite some time to arrive at correct unnesting rewrites for SQL). In fact we have proven the correctness of all unnesting equivalences we present in this article. Second, the algebraic equivalences for rewriting are more general (i.e., we can apply them for every query language that is translatable into the particular algebra). Third, unnesting equivalences can be used during plan generation and, thus, the unnesting procedure itself becomes cost-based. This is especially important since alternative unnested query variants may differ in costs (it may even be the case that the original nested plan has the lowest costs).

Given these advantages, we decided to look at the problem of unnesting in the context of XQuery at the algebraic level. Let us first look a little more carefully at reuse issues. The advantage of query language independence holds as long as the algebraic operators found in the equivalences can be used to express the queries in the given query language. This requires that the domain underlying the algebra is compatible with that demanded by the query. The domains considered for unnesting so far are the bulk types *set* and *bag*. Both types are not order-preserving. This is why the unnesting equivalences developed thus far remain applicable only to those XQuery queries or query parts that do not require order preservation (e.g., expressions containing clauses like *distinct-value*, *unordered*, or *order by*). For example, the query given above consists of two distinctive parts. The path expression `doc("bib.xml")//book/title` needs to return the book titles in exactly the order they appear in the document. In contrast, the order of the titles returned by the expression `doc("reviews.xml")//entry/title` is irrelevant, since it is nested in an existential quantifier. As we will see, if (intermediate) order is of no

concern, we can simply replace certain operators in our algebraic expressions with a non-order-preserving counterpart without compromising the query result. Since the techniques for evaluating non-order-preserving XQuery queries reduce to the case of sets or bags, we focus on order-preserving queries as a novel challenge.

Let us now give a quick overview of our approach. In a first step, an XQuery query is normalized (i.e., rewritten at the source level, resulting in a canonical form). At this point no unnesting takes place yet. The normalization makes it easier for us to translate the query into our algebra and reduces the number of query patterns that we have to be able to distinguish in order to detect the nested parts of a query. In the next step, we translate the query into our algebra. After that the actual unnesting takes place. We look for certain patterns that represent nested algebraic subexpressions obtained from the translation step. We have identified three major patterns: one for existentially quantified expressions, one for universally quantified expressions, and one for (implicit) grouping. For each of these patterns we give a number of unnesting equivalences, that is, rules on how to replace a nested subexpression with one that is not nested anymore. Each set of equivalences (for the different patterns) is complemented by a strategy that describes when to apply which rule. Even though we normalize the queries, we may not be able to apply unnesting equivalences to an algebraic expression immediately. Consequently, we supply so-called support rewrite rules that help us in bringing an algebraic expression into the appropriate syntactical form.

We go even further by showing the unnesting process in action, that is, we take real-world queries (inspired by the XQuery use cases), normalize and translate them, and go through the unnesting step by step. We took care to also select more complicated queries where the procedure for unnesting is not immediately obvious in order to demonstrate the full power of our framework. Based on this detailed description, an implementation can be derived. But for space reasons, we do not cover the implementation of the unnesting procedure. Last but not least, we ran all queries in our native XML DBMS Natix, showing the huge performance gains possible by unnesting.

In Section 2 we review previous approaches for optimizing nested queries. So far research has concentrated on the unordered case. Instead, in the current article we focus on queries relying on order. We take the approach of unnesting at the algebraic level. Therefore, we present an algebra whose domain consists of (ordered) sequences (Section 3). Then in Section 4, we show how XQuery can be translated into this algebra. Thereby, nested queries will result in nested algebraic expressions. To make sure that this translation results only in a limited number of patterns for nested queries, we normalize queries before translating them into the algebra. In Section 5 we explicitly elaborate on the patterns that result from translating nested normalized queries. Thereafter, we present the algebraic equivalences for unnesting and some helper equivalences that enable their applicability. Examples and experimental results demonstrating the power of unnesting accompany the equivalences. In order not to just overwhelm the reader with an unstructured bag of equivalences, we organize them into a decision tree. This then represents the unnesting strategy to follow when

encountering a nested query. The top level of the decision tree is reflected by the three main sections 6, 7, and 8. Each section is devoted to one of the patterns identified in Section 5. The first section discusses selections with existential quantifiers while the second treats universal quantifiers. The third section handles nesting in the map operator. Each section contains the decision (sub-) tree for its corresponding pattern. Section 9 summarizes the results of this article and outlines future research directions.

2. RELATED WORK

The problem of how to handle nested queries first occurred for SQL. The original technique proposed was to evaluate the inner query block for each tuple of the outer block [Astrahan and Chamberlin 1975]. Although Graefe [2003] showed that this straightforward nested evaluation can be improved by several techniques (which were later extended by Guravannavar et al. [2005]), this approach usually lacks efficiency. This is the case when there are many tuples produced by the outer block. Then the invocation of the subquery demands considerable work. Furthermore, the nested evaluation often hinders subsequent algebraic optimizations.

Kim [1982] was the first to observe that it is possible to rewrite a nested SQL query into an unnested one and thereby significantly improve the evaluation cost. He introduced a classification for nested queries and pointed out that nested queries can be unnested such that the transformed query uses joins or grouping instead of nested queries. However, restrictions required for their validity have been found for some of his rewrites. They mainly concern empty results for the inner query block, NULL values, and duplicate handling.

Several proposals have been made to avoid problems with empty results [Dayal 1987; Ganski and Wong 1987; Kiessling 1984; Muralikrishna 1989, 1992] and duplicates [Klug 1982; Pirahesh et al. 1992; Seshadri et al. 1996b]. The rewrites introduced grouping, outer joins, and semijoins, which increased the expressiveness of SQL and widened the range for additional optimizations [Muralikrishna 1989, 1992; Yan and Larson 1994]. One of the most important constructs needed for correctly unnesting queries turned out to be outer joins [Dayal 1987; Ganski and Wong 1987; Kiessling 1984]. After their introduction into SQL and their usage for unnesting, reordering of outer joins became an important topic [Bhargava et al. 1995; Galindo-Legaria and Rosenthal 1997; Rosenthal and Galindo-Legaria 1990]. A major technique for decorrelating queries are Magic sets [Mumick et al. 1990; Seshadri et al. 1996b]. A unifying framework for different unnesting strategies for SQL can be found in Muralikrishna [1992].

It is important to note that early approaches cited in this section unnest at the query language level or a query representation close to it and not at the algebraic level.

A representation we would consider close to the query language level is any kind of calculus due to its declarative nature. An approach representing unnesting techniques for calculus expressions was proposed by Fegaras [1998] and Fegaras and Maier [2000]. Later, Fegaras tried to adopt his approach to

XQuery [Fegaras et al. 2002]. However, from his exposition it is far from clear whether the unnesting techniques presented there preserve order. Further, the calculus representation demands another transformation to an algebraic representation of the query. Also note that, under his approach, it is not possible to incorporate unnesting into cost-based plan generation.

When object-oriented databases and their query language OQL became popular, it was time to reconsider the treatment of nested queries [Cluet and Moerkotte 1994, 1995; Steenhagen 1995; Steenhagen et al. 1994]. Now a paradigm shift took place: OQL queries were translated into nested algebraic expressions and unnesting was performed at the algebraic level. One of the main advantages of this approach is that the results can be applied directly to any other query language translatable into the underlying algebra. In fact, the unnesting techniques developed in Cluet and Moerkotte [1994, 1995], Steenhagen [1995], and Steenhagen et al. [1994] for OQL have been applied directly to SQL [Galindo-Legaria and Joshi 2001]. Similarly, optimization of XQuery can benefit from these techniques for queries that do not preserve order or when order is explicitly treated in an unordered query processing environment. The latter can be achieved by translating XQuery into SQL [Grust et al. 2004] or into a relational algebra [Pal et al. 2005; Liu et al. 2005], unnesting the query, and adding a final sort. While this technique is feasible, we argued in May et al. [2004b] that the decision to destroy and later repair document order should be based on costs. One contribution of this article is to point out when no sorting is needed after unnesting nested queries in an order-preserving query processor.

Another major advantage of unnesting at the algebraic level is that now unnesting can be integrated into cost-based plan generation [Galindo-Legaria and Joshi 2001]. Before, unnesting would always be applied in a rewrite phase that preceded the actual plan generation. On the one hand, unnesting in the rewrite phase is good for those unnesting techniques which always improve performance, as the plan generator does not have to explore an increased search space. On the other hand, it is bad for those unnesting techniques which only sometimes improve performance. These are better dealt with during the actual plan generation.

The optimization of queries containing quantifiers has been investigated in the relational and object-oriented context. Techniques for unnesting existentially quantified nested query blocks can be found in Cluet and Moerkotte [1994, 1995], Steenhagen [1995], and Steenhagen et al. [1994]. Nakano [1990] proposed a rule set for translating quantified calculus expressions into equivalent unnested algebraic expressions. A survey on how to treat universal quantification can be found in Claussen et al. [1997]. In both cases, order was of no concern.

XQuery lacks an explicit grouping construct—a situation that is likely to be remedied [Borkar and Carey 2004; Beyer et al. 2004, 2005]. Until then, grouping must be formulated implicitly, giving rise to another stereotype of nested queries. But even when explicit grouping arrives in XQuery, nested queries will probably still be used sometimes to express grouping implicitly. Detecting and unnesting implicit grouping is a challenging task; before us, Paparizos et al. [2002] tried to tackle it. In their approach, a tree pattern-based grouping

operator was proposed, and a single case where it can be beneficially used to unnest a nested query was identified. However, the description was at a rather high level and special cases were not taken care of, for example, empty groups. Deutsch et al. [2004] presented an algorithm for detecting grouping on a subset of XQuery. Their algorithm minimizes the number of navigation steps needed to evaluate a query. However, their algorithm does not preserve order semantics as required in XQuery.

In our own previous work [May et al. 2003, 2004b], we have looked at specific patterns to unnest XQuery queries containing quantifiers or implicit grouping. In this article, we extend this work with a classification of nested queries based on three basic algebraic patterns. We embed the unnesting equivalences of our previous work and some new equivalences into an unnesting strategy in the form of one decision tree for each pattern. To point out the power of our unnesting strategy, we apply this stragegy to queries of considerable complexity.

Closely connected to the efficient evaluation of XQuery is that of XPath [Gottlob et al. 2002, 2003; Brantner et al. 2005]. When XPath expressions are translated into our algebra, our unnesting techniques can also be applied to them.

3. NOTATION AND ALGEBRA

In this section we discuss the Natix ALgebra (NAL), which works on sequences of tuples. Readers familiar with our algebra may skip this section and resume with Section 4. In Figure 1, we give a brief overview with the formal definitions of our algebraic operators. It might serve as a reference in the remainder of this article.

3.1 Notation

Our algebra (NAL) extends the SAL-Algebra developed by Beeri and Tzaban [1999]. SAL, in turn, is the order-preserving counterpart of the algebra used in Cluet and Moerkotte [1994, 1995]. Both SAL and NAL work on sequences of tuples and allow for nested tuples, that is, the value of an attribute may be a sequence of tuples.

We denote sequences by $\langle \cdot \rangle$, the empty sequence by ϵ, and sequence concatenation by $\oplus$. For a sequence e, we use $\alpha(e)$ to select its first element and the $\tau(e)$ to retrieve its tail. We identify sequences containing a single item with the item contained.

Tuples are denoted using brackets ($[\cdot]$) and their concatenation by $\circ$. The set of attributes of a tuple t is denoted by $\mathcal{A}(t)$. The projection of a tuple t on a set of attributes A is denoted by $t|_A$.

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

Scan singleton

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

Selection

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

Tid

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

Projection

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

Tid-duplicate elimination

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

Map

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

Product

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

Cross product

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

Join

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

Semijoin

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

Antijoin

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

Left outer join

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

Unnest

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

Unnest map

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

Binary grouping

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

Unary grouping

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

Fig. 1. Natix ALgebra: algebraic operators.

We call an attribute A in an expression e *free* if it occurs in e and is not bound to a value by e. That is, a value for A has to be provided by some other expression, for example, an outer query block. We denote the set of free attributes of an expression e by $\mathcal{F}(e)$. Note that attributes behave the same way as variables: they are bound to a value by some expression and referenced by another one. Henceforth, we will use the terms variable and attribute interchangeably.

For an expression e_1 possibly containing free variables, and a tuple e_2, we denote by $e_1(e_2)$ the result of evaluating e_1 where bindings of free variables are taken from variable bindings provided by e_2. Of course this requires $\mathcal{F}(e_1) \subseteq \mathcal{A}(e_2)$. For a set of attributes, we define the tuple constructor $\bot_A$ such that it returns a tuple with attributes in A initialized to NULL.

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

We denote the identity function by *id* and function concatenation by $\circ$.

3.2 The NAL Algebra

We give the definitions for the order-preserving algebraic operators. For the unordered counterparts, see Cluet and Moerkotte [1995]. The NAL algebra allows for nesting of algebraic expressions. For example, within a selection predicate we allow for the occurrence of a nested algebraic expression. Hence, for example, a join within a selection predicate is possible. This simplifies the translation procedure of nested XQuery expressions into the algebra.

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

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

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

We define an auxiliary operator *tid* which numbers the tuples in a sequence by adding an attribute A to each tuple that contains its position within the sequence. We need this operator to identify original tuples of a sequence after they have been connected to other tuples. Numbering tuples is also a convenient means to remember order [May et al. 2004a], to implement position-based functions, or to support positional variables (*at*) in *for* clauses. We define $tid_A(e) := tid_A(e, 1)$ where

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

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

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

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

We also need a special order-preserving duplicate-eliminating projection $\Pi^{tid_B}_A$, which removes multiple occurrences of the same *tid*-value B (if it

R_1
A_1
1
2
3

R_2	
A_2	B
1	2
1	3
2	4
2	5

$\chi_{a:\sigma_{A_1=A_2}(R_2)}(R_1) =$	
A_1	a
1	$\langle [A_2 : 1, B : 2], [A_2 : 1, B : 3] \rangle$
2	$\langle [A_2 : 2, B : 4], [A_2 : 2, B : 5] \rangle$
3	$\langle \, \rangle$

Fig. 2. Example for the map operator.

appears in subsequent tuples):

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

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

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

The *map* operator is defined as follows:

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

It extends a given input tuple $t_1 \in e_1$ by a new attribute a whose value is computed by evaluating $e_2(t_1)$. For an example see Figure 2.

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

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

where

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

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

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

We define the *semijoin* as

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

and the *antijoin* as

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

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

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

R_1

A_1
1
2
3

R_2

A_2	B
1	2
1	3
2	4
2	5

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

A_2	g
1	2
2	2

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

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

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

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

Fig. 3. Examples for unary and binary grouping.

where $g \in \mathcal{A}(e_2)$. Our definition slightly deviates from the standard left outer join, as we want to use it in conjunction with grouping and (aggregate) functions. Consider for example the sequences R_1, R_2, and R_2^{count} in Figure 3. Note that R_2^{count} is derived from R_2 by grouping it on A_2 and then counting the tuples in each group. Now assume that we want to join R_1 (via left outer join) with R_2^{count}. Obviously, tuple 3 of R_1 does not have a join partner. The standard left outer join would add a NULL value for g. In our case, having no join partner corresponds to an empty group and the cardinality of it is well known (0). Hence, we use it as a default value. In general, e defines the value given to attribute g for values in e_1 that do not find a join partner in e_2.

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

As the definitions of the grouping operators are rather involved, we employ the example in Figure 3 again. Unary grouping (cf. R_2^g in Figure 3) groups R_2 on attribute A_2 and adds a new attribute g which is "the group." In the example in Figure 3, attribute g of R_2^g contains a sequence of tuples. These tuples all share the same value on the grouping attribute A_2. For some functions f (in particular aggregate functions), we do not have to keep all the tuples that comprise a group. In our example, the values for the count of each group in R_2^{count} can be computed incrementally.

In contrast to unary grouping, which works on one input sequence, binary grouping takes two input sequences as input (cf. $R_{1,2}^g$ in Figure 3). The left input R_1 defines the groups while the tuples of the right input R_2 are matched to these groups. Again, function f is used to combine the tuples in each group. For the identity function id, this results in a sequence of tuples. Note that the last group does not find matching tuples in R_2. Thus, this group contains an empty sequence. This is important when we access the sequence-valued attribute g. Also note that binary grouping in this example computes the same result as the map operator in Figure 2.

We define unary grouping in terms of binary grouping. Hence, we start with the formal definition of *binary grouping*:

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

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

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

Given a sequence of tuples containing a sequence-valued attribute, the *unnest* operator unnests this attribute by producing a result tuple for every tuple contained in the sequence-valued attribute. The order-preserving counterpart to the well-known unnest operator is defined as

$$\mu_{a:g}(e) := (\alpha(e) \times (\alpha(e).g)_{|a:\mathcal{A}(g)}) \oplus \mu_{a:g}(\tau(e)),$$

where $e.g$ retrieves the sequence of tuples of attribute g and renames them to the attribute names given in a. Only in some rare cases the attribute g is referred to in operators following the unnest. Hence, unnest preserves the sequence-valued attribute g. However, we will mostly ignore its existence. Still, we may use the fact that $\Pi_{A_2:A_3}(\Pi_{A_3 B}(\mu_{\mathcal{A}(g):g}(\Pi_{A_3:A_2}(R^g_2)))) = R_2$ holds for the sequences R_2 and R^g_2 in Figure 3. Hence, the unnest operator can extract the sequence-valued attributes computed by a grouping operation.

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

$$\Upsilon_{a:e_2}(e_1) := \mu_{a:\hat{a}}(\chi_{\hat{a}:e_2}(e_1)).$$

It first materializes a sequence of tuples in a new sequence-valued attribute $\hat{a}$, which is then immediately unnested. As a result, the tuples of e_1 are extended by the attributes in e_2 which are renamed to the set of attribute names in a. Basically, the *unnest map* operator has the same semantics as a *d-join* [Cluet and Moerkotte 1994] or the *Apply operator* [Galindo-Legaria and Joshi 2001].

We mainly use the unnest map operator to evaluate XPath expressions. Therefore, we translate the XPath expressions as presented in Brantner et al. [2005]. We put the resulting algebraic expression in the place of e_2 in the subscript of the operator. In this article, we only use the items of the result sequence and ignore the context position and context size. The unnest map operator binds these items to the variable given in the XQuery expression. Note that our translation of XPath expressions yields sequences of tuples as opposed to sequences of items as defined in XQuery [Draper et al. 2005].

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

Note, that several equivalences known from the unordered context still hold. In Electronic Appendix A.1, we list the ones we use in this article.

Let us comment on the implementations of the more complex algebraic operators. Standard implementation techniques for some algebraic operators [Graefe 1993] do not preserve order. Claussen et al. [1998] provided an efficient implementation for an order-preserving hash join. Currently, we have not implemented it but use a nested-loop-join instead to preserve order. When order is not relevant, we employ the Grace-Hash-Join [Fushimi et al. 1986]. Further performance enhancements for unnested plans with joins can be expected when using the order-preserving hash join or the techniques described in May et al. [2004a]. Implementations of binary grouping have been discussed in Chatziantoniou et al. [2001], Cluet and Moerkotte [1996], and May and Moerkotte [2005]. We would also like to point out that the Υ operator generates its output in document order if the translation of XPath expressions described in Brantner et al. [2005] is used. One proposal to implement result construction can be found in Fiebig and Moerkotte [2001].

4. NORMALIZATION AND TRANSLATION

The first part of this section briefly describes the normalization step that is applied to the original query. It takes place at the source level. Then we sketch the translation from XQuery into our algebra. We illustrate both steps using the example query from the Introduction. More examples follow later in the article.

4.1 Normalization

Prior to the translation into our algebra, we use a normalization step that introduces new variables. This step is called *dependency-based optimization* and is used to eliminate common subexpressions. This kind of optimization, although vital, is simple enough and requires mainly one traversal of the query's syntax tree. Since it has been presented elsewhere [Cluet and Delobel 1992], we will not detail it. We apply the following steps:

(1) We break up complex expressions and introduce new variables for subexpressions.
(2) We factorize common subexpressions.
(3) We move predicates from XPath expressions to the *where* clause whenever possible and turn all predicates into conjunctive normal form.
(4) We replace *for* or *let* clauses that bind multiple variables by sequences of individual *for* or *let* clauses.
(5) We turn implicit computations into explicit ones, for example, general comparisons into quantified expressions, and we insert functions to compute effective Boolean values or atomization.

When we apply these normalization steps to the example query from the introduction, the result is the following:

```
for $t1 in doc("bib.xml")//book/title
let $t1d := fn:data($t1)
let $res := <book-with-review> { $t1 } </book-with-review>
```

```
where some $t2 in doc("reviews.xml")//entry/title
       let $t2d := fn:data($t2)
       let $req := $t1d op:equal $t2d
       let $beq := fn:boolean($req)
      satisfies $beq
return $res
```

Normalization of this query consists of splitting complex subexpressions. We classify nested FLWR expressions, path expressions, function calls, element constructors, and sequence expressions as complex expressions. Only constants or variable references are considered simple expressions.

Let us first consider the element construction in the *return* clause of the example query. In the normalized query, the element constructor is replaced by a reference to the new variable $res, which is bound to the element constructor. Consequently, the *return* clause of a normalized query consists only of a single variable reference.

Next, we examine the quantified expression. To ensure that the value comparison is done on atomic values, atomization is performed on both arguments of the comparison by inserting the built-in function `fn:data`. Additionally, the effective Boolean value of the comparison is computed by function `fn:boolean`. Since it is our goal to remove complex expressions from the range predicate of quantified expressions, we introduce new variables for those functions. Therefore, *let* clauses (and possibly *where* clauses) are required in the extended range expression of the quantified expression. These expressions are evaluated in the scope of the last clause of the quantifier. This syntactic sugar on top of XQuery simplifies the translation into the algebra.

The splitting described above allows us to consider every possible subexpression that can be factorized. In this article, we will not split every complex expression but only when necessary to keep our exposition readable. With that same argument, we will also simplify the normalization process. In particular, we will not insert implicit conversion functions when the meaning of the query is obvious.

The motivation for splitting becomes apparent when considering that (1) a *let* clause will be translated into a map operator and (2) many unnesting equivalences (see Section 8) use a map operator as their starting point.

Note that all of these steps require some attention, since a careless application of this procedure may change the semantics of the query. As the result of the normalization step, the translation into our algebra is simplified.

4.2 Translation

We specify the translation procedure by means of three mutually recursive procedures $\mathcal{T}$ (see Figure 4). For a given query Q, $\mathcal{T}_T(Q)$ translates Q into the algebra.

The binary function $\mathcal{T}(Q, A)$ is responsible for translating a FLWR expression Q into the algebra. The first argument of this function is the (remainder of) the query to be translated, and the second argument is the algebraic expression constructed so far. The result of each translation step is a tree of algebraic

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

$$\mathcal{T}(Q, A) := \begin{cases} \mathcal{T}(\text{ REST}, \Upsilon_{x:\mathcal{T}_T(e)}(A)) & \text{if } Q = \textbf{for } \$x \textbf{ in } e \text{ REST} \\ & \text{or if } Q = \$x \textbf{ in } e \text{ REST} \\ \mathcal{T}(\text{ REST}, \chi_{x:\mathcal{T}(e)}(A)) & \text{if } Q = \textbf{let } \$x := e \text{ REST} \\ \mathcal{T}(\text{ REST}, \sigma_{\mathcal{T}_I(p)}(A)) & \text{if } Q = \textbf{where } \text{p REST} \\ \Pi_e(A) & \text{if } Q = \textbf{return } \$\text{e} \\ A & \text{if } Q \text{ is empty string} \end{cases}$$

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

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

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

Fig. 4. Translation of XQuery FLWR expressions into the algebra.

operators which produce sequences of tuples. For each clause of the FLWR expression, we give the corresponding translation rule. We do not treat the *order by* clause here—it requires a trivial extension of our translation. When the result of an expression is sorted, we do not need to preserve the order during the computation of the expression. Hence, optimization techniques for bags suffice in the presence of *order by*.

For non-FLWR expressions, we use two different unary translation functions. Function $\mathcal{T}_I(Q)$ translates a subexpression Q into a function with an simple return type in the XQuery data model, while function $\mathcal{T}_T(Q)$ returns an algebraic expression which produces sequences of tuples.

Since a FLWR expression can occur within simple expressions and vice versa, these functions are mutually recursive. In the translation rule for the *let* clause and functions we do not specify the subscript of $\mathcal{T}$ because the necessary call depends on the argument types of the function to translate.

Before we discuss details of our translation function, we present the result of the translation of the example query into an algebraic expression:

$$\Pi_{res}(\sigma_{\exists t \in e_1 : beq}(\chi_{res:\mathcal{C}(\text{elem,bwr},t1)}(\chi_{t1d:\text{fn:data}(t1)}(\Upsilon_{t1:doc1//book/title}(\square)))))$$

with

$$\begin{aligned} e_1 &:= \chi_{beq:\text{fn:boolean}(req)}(\chi_{req:t1d=t2d}(\chi_{t2d:\text{fn:data}(t2)}(\Upsilon_{t2:doc2//entry/title}(\square)))), \\ \texttt{doc1} &:= \texttt{doc("bib.xml")}, \\ \texttt{doc2} &:= \texttt{doc("reviews.xml")}, \\ \texttt{bwr} &:= \texttt{"book-with-review"}. \end{aligned}$$

Note, that our data model represents sequences of items as sequences of tuples. Thus, if necessary, the translation function must wrap the items in the

sequence into tuples. This is accomplished by using the tuple constructor [·], which must invent a new attribute name to which the item is bound.

Our translation function treats a node constructor like a regular function call and, hence, maps it to a node construction function. Thus, we do not need any special treatment for node construction.

Since normalization simplifies the *return* clause to one variable reference, we simply project the result tuples to the corresponding attribute. As a result, our queries return sequences of tuples where each tuple contains one item. Consequently, a subsequent component can consume the result tuples and serialize the query result as it is convenient for the user.

5. ALGEBRAIC PATTERNS

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

5.1 Quantified Queries

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

Let us reconsider the example query introduced in Section 1 which uses an (existentially) quantified expression in the *where* clause:

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

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

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

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

result construction):

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

with

$$e_1 := \Upsilon_{t1:doc(\text{“bib.xml”})//book/title}(\Box),$$
$$e_2 := \Upsilon_{t2:doc(\text{“reviews.xml”})//entry/title}(\Box).$$

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

basic patterns for nested quantified queries
$\sigma_{\exists x \in e_2 : p}(e_1)$
$\sigma_{\forall x \in e_2 : p}(e_1)$

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

5.2 Implicit Grouping

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

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

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

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

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

```
let $res := <publisher> { $t, $np } </publisher>
return $res
```

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

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

where

$$e_1 := \Upsilon_{p:\Pi^D(doc(\text{“}bib.xml\text{”})//publisher)}(\Box),$$
$$e_2 := \chi_{t2:b/title}(\chi_{p2:b/publisher}(\Upsilon_{b:doc(\text{“}bib.xml\text{”})//book}(\Box))).$$

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

basic pattern for implicit grouping

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

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

6. EXISTENTIAL QUANTIFIERS

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

6.1 Motivating Example

As a motivating example for queries containing existential quantifiers, let us reconsider the query from Section 5 (where we want to find all books with at least one review):

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

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

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

with

$$e_1 := \Upsilon_{t1:doc(\text{“}bib.xml\text{”})//book/title}(\Box),$$
$$e_2 := \Upsilon_{t2:doc(\text{“}reviews.xml\text{”})//entry/title}(\Box).$$

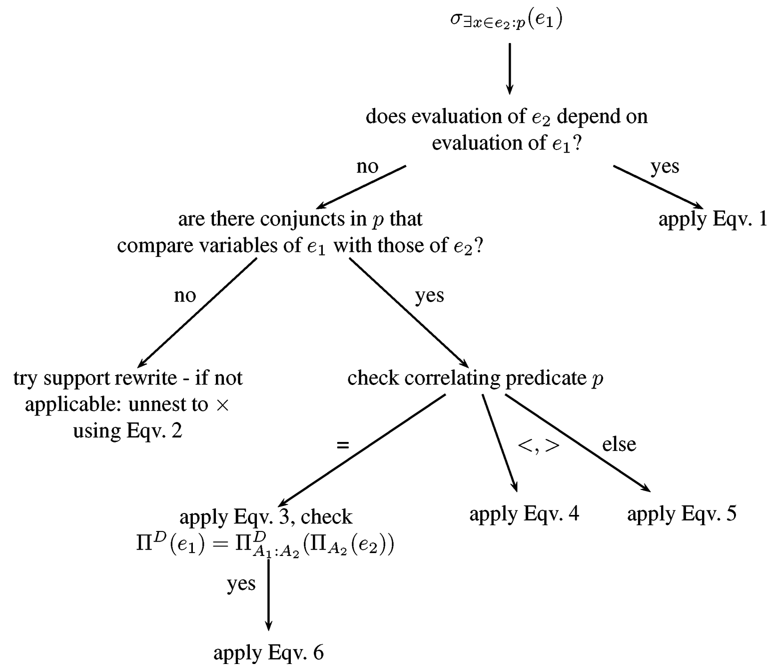

Fig. 5. Decision tree for existentially quantified queries.

It is not hard to detect the basic pattern for existentially quantified queries in this expression. How do we continue from here? Ideally, we would now hand the algebraic expression to an optimizer that determines an efficient query plan based on a cost model. As full-fledged cost models for algebraic-based optimization of XQuery are not available yet, we rely on a heuristic. This heuristic is presented in the form of a decision tree in the next section.

6.2 Optimization Strategy

Figure 5 shows the decision tree we use for unnesting existentially quantified expressions. Going down the tree from top to bottom, we reach more and more specific rules, which we formally define in Figure 6. At the moment, our heuristic consists of applying the most special rewrite rule possible, as the more special rules tend to improve the performance significantly. (For each rule we enumerate all preconditions that have to be met in order to apply this rule, more details follow in the next section.) Let us have a brief look at the decision tree. First of all, we check for an expression $\sigma_{\exists x \in e_2 : p}(e_1)$ if e_2 can be evaluated independently of e_1. If not, we leave the expression as it is or evaluate it via an efficiently implemented unnest map operator (using Eqv. 1) [Graefe 2003; Brantner et al. 2005]. If yes, we examine the predicate p. If we are not able to correlate the expressions e_1 and e_2 via p, then we unnest the expression with the help of a Cartesian product (using Eqv. 2). If p correlates e_1 and e_2, we use different variants of semijoins or grouping/aggregation to unnest the expression (Eqvs. 3, 4, 5, 6).

$$\sigma_{\exists x \in (e_2):p}(e_1) = \Pi^{tid_{i_1}}_{\mathcal{A}(e_1)}(\sigma_p(\Upsilon_{\mathcal{A}(e_2):e_2}(tid_{i_1}(e_1)))) \quad (1)$$

$$\sigma_{\exists x \in (e_2):p}(e_1) = \Pi^{tid_{i_1}}_{\mathcal{A}(e_1)}(\sigma_p(tid_{i_1}(e_1) \times e_2)) \quad (2)$$

$$\sigma_{\exists x \in (\sigma_{A_1=A_2}(e_2)):p}(e_1) = e_1 \ltimes_{A_1=A_2 \wedge p} e_2 \quad (3)$$

$$\sigma_{\exists x \in (\sigma_{A_1 \theta A_2}(e_2)):p}(e_1) = \sigma_{A_1 \theta aggr_{A_2}(\sigma_p(e_2))}(e_1) \quad (4)$$

$$\sigma_{\exists x \in (\sigma_{A_1 \theta A_2}(e_2)):p}(e_1) = e_1 \ltimes_{A_1=A_3} (\Pi_{A_3:A_1}(e_1 \bowtie_{A_1 \theta A_2 \wedge p} e_2)) \quad (5)$$

$$\Pi^D(e_1) \ltimes_{A_1=A_2}(\sigma_p(e_2)) = \sigma_{c>0}(\Pi_{A_1:A_2}(\Gamma_{c;=A_2;count \circ \sigma_p}(e_2))) \quad (6)$$

Fig. 6. Unnesting equivalences for existentially quantified queries.

For our motivating example this means that we end up at Eqv. 3 (Eqv. 6 is not applicable, as `bib.xml` and `reviews.xml` may not contain the same books). Applying Eqv. 3 to our example yields

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

6.3 Equivalences for Unnesting

After having outlined the general strategy, we now present the concrete equivalences (see Figure 6) and list all prerequisites necessary for applying them.

—*Equivalence* 1:
 —*Preconditions*: e_1 and e_2 cannot be evaluated independently (formally speaking, $\mathcal{F}(e_2) \cap \mathcal{A}(e_1) \neq \emptyset$).
 —*Basic idea*: Combine all tuples in e_1 with all tuples in $e_2(e_1)$ via an unnest map operator and then apply p. We need the `tids` to eliminate duplicates.

—*Equivalence* 2:
 —*Preconditions*: e_1 and e_2 can be evaluated independently ($\mathcal{F}(e_2) \cap \mathcal{A}(e_1) = \emptyset$).
 —*Basic idea*: Combine all tuples in e_1 with all tuples in e_2 via a Cartesian product and then apply p. We need the `tids` to eliminate duplicates. This equivalence has to be used if e_1 and e_2 are not correlated via the predicate p. If e_1 and e_2 are correlated, it should only be used if the other equivalences are not applicable.

—*Equivalence* 3:
 —*Preconditions*: e_1 and e_2 can be evaluated independently, and e_1 and e_2 are correlated with an equality predicate.
 —*Basic idea*: Use a semijoin to evaluate the expression. We expect the evaluation of a semijoin operator to be much more efficient than that of a cross product or the nested version of the expression.

—*Equivalence* 6:
 —*Preconditions*: Eqv. 6 is a special case of Eqv. 3. In addition to the preconditions of Eqv. 3, $\Pi^D(e_1) = \Pi^D_{A_1:A_2}(\Pi_{A_2}(e_2))$ must hold. This is the case, for example, if both expressions, e_1 and e_2, scan the same document.
 —*Basic idea*: We do not have to redundantly evaluate both e_1 and e_2. It is sufficient to just group all tuples in e_2 that satisfy predicate p and count them. In the equivalence we denote this operation by a function

composition. For existential quantification the number of tuples satisfying p for a certain value A_2 has to be greater than 0.

—*Equivalence* 4:

—*Preconditions*: Same as for Eqv. 3, except that e_1 and e_2 are correlated with an inequality predicate. The following table gives the correct assignments for θ, $\neg\theta$ and *aggr*:

θ	*aggr*
$>, \geq$	*min*
$<, \leq$	*max*

—*Basic idea*: If we have an inequality comparison operator ($\theta \in \{<, \leq, \geq, >\}$), we just need to compare the value of A_1 to the minimal or maximal value of A_2. For existential quantification, a tuple of e_1 satisfies the query predicate if A_1 lies in the range $[\min_{A_2}(e_2), \infty)$ or in the range $(-\infty, \max_{A_2}(e_2)]$, respectively. The resulting nested expression can be unnested with equivalences that are introduced in Section 8.

We have to be careful when handling the special case $e_2 = \epsilon$. In this case, the predicate $A_1 \theta aggr$ is evaluated to false. Additionally, we must take care of the semantics of XQuery: In XQuery the sequence of items that functions *min* or *max* get as arguments convert these items to $xs : double$. In contrast, the general comparison does not perform such an implicit type conversion, i.e., $xs : string$ is used for the items. For strings we rely on the collation to order the strings and to compute the minimum or maximum.

—*Equivalence* 5:

—*Preconditions*: Same as for Eqv. 3, except that e_1 and e_2 are correlated with an arbitrary predicate. This is the most general case for correlated expressions.

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

6.4 Support Rewrites

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

For example, take the following expression, in which x and y refer to attributes of the tuples s and t, respectively:

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

None of the equivalences presented in Section 6.3 can be applied to this expression directly. However, if we rewrite it to

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

[1]Note that we cannot use the θ-semijoin proposed by Seshadri et al. [1996a] because it is restricted to an unordered context.

$$\exists x \in e_1 : \exists y \in e_2 : p \;=\; \exists y \in e_2 : \exists x \in e_1 : p \tag{7}$$
$$\exists x \in e_1 : p \wedge q \;=\; \exists x \in \sigma_p(e_1) : q \tag{8}$$
$$\exists x \in \Pi_A(e_1) : p \;=\; \exists x \in e_1 : p \tag{9}$$
$$p \wedge \exists x \in e_1 : q \;=\; \exists x \in e_1 : p \wedge q \tag{10}$$
$$p \vee \exists x \in e_1 : q \;=\; \exists x \in e_1 : p \vee q \tag{11}$$
$$\begin{aligned}\sigma_{\exists x \in e_2 : p \wedge \exists y \in e_3 : q}(e_1) &= \sigma_{\exists x \in e_2 : p}(\sigma_{\exists y \in e_3 : q}(e_1)) \\ &= \sigma_{\exists y \in e_3 : q}(\sigma_{\exists x \in e_2 : p}(e_1))\end{aligned} \tag{12}$$

Fig. 7. Support rewrites for existentially quantified queries.

we can apply equivalence 3 and replace the selection with a semijoin:

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

When rewriting expressions, we follow two general heuristics. First, we try to reduce the number of free variables in the subexpression we want to unnest. This is mainly achieved by splitting and moving predicates [Steenhagen 1995]. As all free variables in a subexpression are bound by the enclosing expression, by moving these free variables we try to decouple the subexpression from the enclosing expression as much as possible. The second heuristic involves minimizing the distance between query blocks that are correlated via predicates. These two strategies simplify the unnesting of subexpressions considerably.

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

Let us now have a look at the rewrite rules (all rules are summarized in Figure 7). This list is in no way exhaustive (we just included rules that are needed in the remainder of this paper) and many of the rules are common knowledge and have already been described elsewhere [Bry 1989; Jarke and Koch 1984; Steenhagen 1995]. Hence, we refer the reader to Electronic Appendix A.2 for a discussion of their applicability.

6.5 Example Queries

We now present more detailed example queries showing the unnesting and support rewrite rules in action. These examples also include measurements on the evaluation times of the different query plans (further experimental results can be found in May et al. [2003] and in Electronic Appendix C).

6.5.1 *Exchanging Quantifiers.* With the following example query, we show how an expression can be rewritten using Eqv. 7 to allow for more efficient unnesting techniques. In the query below we want to determine all users of an auction site who are actively bidding on at least one item:

```
for $u in doc("users.xml")//usertuple
where some $i in doc("items.xml")//itemtuple
      satisfies some $b in doc("bids.xml")//bidtuple
```

```
                  satisfies ($u/userid eq $b/userid and
                             $i/itemno eq $b/itemno)
return $u/name
```

Following the normalization steps introduced in Section 4.1, we move the path expressions in the innermost range predicate into new *let* clauses in the quantified subexpressions.

```
for $u in doc("users.xml")//usertuple
let $un  := $u/name
let $uu  := $u/userid
where some $i in in doc("items.xml")//itemtuple
      let  $ii := $i/itemno
      satisfies some $b in doc("bids.xml")//bidtuple
                let  $bu := $b/userid
                let  $bi := $b/itemno
                satisfies ($uu eq $bu and $ii eq $bi)
return $un
```

Translating the above into our algebra results in the following expression:

$$\Pi_{un}(\chi_{un:u/name}(\sigma_{\exists it\in(e_2):\exists bt\in(e_3):e_4}(e_1))),$$

where

$$e_1 := \chi_{uu:u/userid}(\Upsilon_{u:doc1//usertuple}(\Box))$$
$$e_2 := \chi_{ii:i/itemno}(\Upsilon_{i:doc2//itemtuple}(\Box))$$
$$e_3 := \chi_{bi:b/itemno}(\chi_{bu:b/userid}(\Upsilon_{b:doc3//bidtuple}(\Box)))$$
$$e_4 = uu = bu \wedge ii = bi.$$

and

```
doc1 := doc("users.xml")
doc2 := doc("items.xml")
doc3 := doc("bids.xml")
```

Note that during the translation we exploit the fact that the child nodes of `itemtuple`, `bidtuple`, and `usertuple` occur exactly once. Since predicate e_4 references variables bound in e_1, e_2, and e_3, none of the more efficient unnesting equivalences on the lower right-hand side of the decision tree are applicable immediately. However, using some of the support rewrite rules, we can remedy this situation. First, we are going to present a naive approach to unnesting the above algebraic expression. Then we will show how to optimize it in a more clever way.

6.5.1.1 *Naive Unnesting.* As e_1 and e_2 can be evaluated independently of each other and they are not correlated in any way, we can apply Eqv. 2. After having pushed down the predicate e_4 (see Eqv. 8), we can apply Eqv. 3 connecting e_3 via a semijoin:

$$\begin{aligned}
&\Pi_{un}(\chi_{un:u/name}(\sigma_{\exists it\in(e_2):\exists bt\in(e_3):e_4}(e_1))),\\
\overset{(2)}{=}\ &\Pi_{un}(\chi_{un:u/name}(\Pi^{tid_{p_1}}_{\mathcal{A}(e_1)}(\sigma_{\exists bt\in(e_3):e_4}(tid_{p_1}(e_1)\times e_2)))),\\
\overset{(8)}{=}\ &\Pi_{un}(\chi_{un:u/name}(\Pi^{tid_{p_1}}_{\mathcal{A}(e_1)}(\sigma_{\exists bt\in(\sigma_{e_4}(e_3)):true}(tid_{p_1}(e_1)\times e_2)))),\\
\overset{(3)}{=}\ &\Pi_{un}(\chi_{un:u/name}(\Pi^{tid_{p_1}}_{\mathcal{A}(e_1)}((tid_{p_1}(e_1)\times e_2)\ltimes_{e_4} e_3))).
\end{aligned}$$

6.5.1.2 *Improved Unnesting.* However, we can do better than that and avoid using the Cartesian product. If we first reorder the quantifiers $\exists it \in (e_2) : \exists bt \in (e_3) : e_4$ using Eqv. 7 and then push down the first part of the predicate e_4, we can apply Eqv. 3. After having pushed down the second part of e_4, we can apply Eqv. 3 again, arriving at an expression containing two semijoins:

$$
\begin{aligned}
& \Pi_{un}(\chi_{un:u/name}(\sigma_{\exists\ it\ \in(e_2):\exists\ bt\ \in(e_3):e_4}(e_1))) \\
\stackrel{(7)}{=}\ & \Pi_{un}(\chi_{un:u/name}(\sigma_{\exists\ bt\ \in(e_3):\exists\ it\ \in(e_2):e_4}(e_1))) \\
\stackrel{(8)}{=}\ & \Pi_{un}(\chi_{un:u/name}(\sigma_{\exists\ bt\ \in(e_3):\exists\ it\ \in(\sigma_{ii=bi}(e_2)):uu=bu}(e_1))) \\
\stackrel{(8)}{=}\ & \Pi_{un}(\chi_{un:u/name}(\sigma_{\exists\ bt\ \in\sigma_{\exists\ it\ \in(\sigma_{ii=bi}(e_2))}(e_3):uu=bu}(e_1))) \\
\stackrel{(3)}{=}\ & \Pi_{un}(\chi_{un:u/name}(\sigma_{\exists\ bt\ \in(e_3 \ltimes_{ii=bi} e_2):uu=bu}(e_1))) \\
\stackrel{(8)}{=}\ & \Pi_{un}(\chi_{un:u/name}(\sigma_{\exists\ bt\ \in(\sigma_{uu=bu}(e_3 \ltimes_{ii=bi} e_2)):true}(e_1))) \\
\stackrel{(3)}{=}\ & \Pi_{un}(\chi_{un:u/name}(e_1 \ltimes_{uu=bu}(e_3 \ltimes_{ii=bi}, e_2))).
\end{aligned}
$$

6.5.1.3 *Evaluation.* Before we discuss the experimental results, let us briefly describe the experimental setup. All queries were implemented and evaluated in our native XML database system Natix. They were executed with warm buffer on documents that fit into the database buffer. We only report elapsed times because query execution was CPU-bound.

The data sets we used are based on the XQuery Use Cases "XMP" and "R." "XMP" contains data on books, authors, editors, reviews, and so on, while "R" describes an auction site with users, items, bids, etc. As in this first example query, we will sometimes use the fact that child nodes occur exactly once below their parents. In Electronic Appendix D, we give further details of the experimental setup.

Running the nested, the naively unnested, and the improved unnested versions, we acquired the following averaged running times (in seconds).

Size	100	1000	10,000
Nested	10.42 s	3944.71 s	∞
Naively unnested	0.16 s	8.45 s	860.69 s
Improved unnested	0.08 s	0.12 s	0.56 s

The nested version is clearly the slowest variant (for a document size of 10,000 nodes we aborted the execution after 3 h). While the naively unnested version already improves the performance by several orders of magnitude, we can decrease the evaluation time even further below 1 s for the largest document size by eliminating the Cartesian product.

6.5.2 *Complex Correlation.* In the following example query, we demonstrate how complex correlation predicates between query blocks can be untangled. We retrieve all users who bid on an item (which they do not offer themselves) and where the bid is at least twice as high as the reserve price:

```
for $u in doc("users.xml")//usertuple
where some $i in doc("items.xml")//itemtuple
```

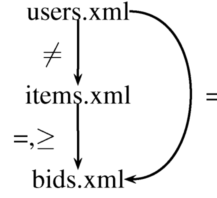

Fig. 8. Dependencies.

```
      satisfies ($i/offeredby ne $u/userid
            and some $b in doc("bids.xml")//bidtuple
                satisfies ($b/userid eq $u/userid
                      and  $b/itemno eq $i/itemno
                      and  ($b/bid cast as xs:double) gt
                           (2.0 * $i/reserveprice)))
return $u/userid
```

Normalizing and translating the XQuery expression into our algebra, we get:

$$\Pi_{ui}(\sigma_{\exists it \in e_2:(io \neq ui \wedge \exists bt \in e_3:e_4)}(e_1)),$$

where

$e_1 := \chi_{ui:u/userid}(\Upsilon_{u:doc1//usertuple}(\Box)),$

$e_2 := \chi_{in:i/itemno}(\chi_{ir:i/reserveprice}(\chi_{io:i/offeredby}(\Upsilon_{i:doc2//itemtuple}(\Box)))),$

$e_3 := \chi_{bn:b/itemno}(\chi_{bb:b/bid}(\chi_{bi:b/userid}(\Upsilon_{b:doc3//bidtuple}(\Box)))),$

$e_4 := bi = ui \wedge bn = in \wedge bb > 2.0 \cdot ir.$

and

```
doc1 := doc("users.xml"),
doc2 := doc("items.xml"),
doc3 := doc("bids.xml"),
```

Although the correlation predicate looks quite complicated, our unnesting techniques are powerful enough to handle even this case. The graph in Figure 8 depicts the complexity of the correlation predicate by showing how the query blocks accessing the different documents (represented as nodes) are connected via the predicates (represented as edges). The edge runs from the query block that binds a variable to the nested query block that uses this binding:

We present two different ways to unnest the above algebraic expression. One involves a direct unnesting via a semijoin, the other an indirect unnesting via a Cartesian product (which is eliminated later on).

6.5.2.1 *Semijoin 1.* This approach is quite straightforward, as we apply Eqv. 5, pull a part of the join predicate into a selection outside the join, and then apply Eqv. 5 again in order to unnest the doubly nested expression:

$$\begin{aligned}
& \Pi_{ui}(\sigma_{\exists it \in e_2:(io \neq ui \wedge \exists bt \in e_3:e_4)}(e_1)) \\
\overset{(5)}{=}\ & \Pi_{ui}(e_1 \ltimes_{\mathcal{A}(e_1)=\mathcal{A}(e_1)'}(\Pi_{\mathcal{A}(e_1)':\mathcal{A}(e_1)}(e_1 \bowtie_{io \neq ui \wedge \exists bt \in e_3:e_4} e_2))) \\
=\ & \Pi_{ui}(e_1 \ltimes_{\mathcal{A}(e_1)=\mathcal{A}(e_1)'}(\Pi_{\mathcal{A}(e_1)':\mathcal{A}(e_1)}(\sigma_{\exists bt \in e_3:e_4}(e_1 \bowtie_{io \neq ui} e_2)))) \\
\overset{(5)}{=}\ & \Pi_{ui}(e_1 \ltimes_{\mathcal{A}(e_1)=\mathcal{A}(e_1)'}(\Pi_{\mathcal{A}(e_1)':\mathcal{A}(e_1)}((e_1 \bowtie_{io \neq ui} e_2) \ltimes_{\mathcal{A}(e_1,e_2)=\mathcal{A}(e_1,e_2)'} \\
& \quad (\Pi_{\mathcal{A}(e_1,e_2)':\mathcal{A}(e_1,e_2)}((e_1 \bowtie_{io \neq ui} e_2) \bowtie_{e_4} e_3))))).
\end{aligned}$$

6.5.2.2 *Semijoin 2.* Although we advised against using Cartesian products, we can use Eqv.2 in a first step, then pull in part of the selection predicate into the Cartesian product to change it into a join, and finally apply Eqv. 5, introducing a semijoin:

$$
\begin{aligned}
& \Pi_{ui}(\sigma_{\exists it \in e_2 : (io \neq ui \wedge \exists bt \in e_3 : e_4)}(e_1)) \\
\overset{(2)}{=}\; & \Pi_{ui}(\Pi^{tid_{p_1}}_{\mathcal{A}(e_1)}((\sigma_{io \neq ui \wedge \exists bt \in (e_3) : e_4}(tid_{p_1}(e_1) \times e_2)))) \\
\overset{(12)}{=}\; & \Pi_{ui}(\Pi^{tid_{p_1}}_{\mathcal{A}(e_1)}(\sigma_{\exists bt \in (e_3) : e_4}(\sigma_{io \neq ui}(tid_{p_1}(e_1) \times e_2)))) \\
\overset{(5)}{=}\; & \Pi_{ui}(\Pi^{tid_{p_1}}_{\mathcal{A}(e_1)}(\Pi_{\mathcal{A}(e_1)' : \mathcal{A}(e_1)}((tid_{p_1}(e_1) \bowtie_{io \neq ui} e_2) \ltimes_{\mathcal{A}(e_1, e_2) = \mathcal{A}(e_1, e_2)'} \\
& \quad (\Pi_{\mathcal{A}(e_1, e_2)' : \mathcal{A}(e_1, e_2)}((tid_{p_1}(e_1) \bowtie_{io \neq ui} e_2) \bowtie_{e_4} e_3))))).
\end{aligned}
$$

The main difference between this expression and the first semijoin variant is the fact that, in the first variant, all θ-joins need *not* be order-preserving (the semijoin with e_1 determines the final order), while here the first θ-join between e_1 and e_2 needs to be order-preserving. In both variants we can optimize the expression $(e_1 \bowtie_{io \neq ui} e_2) \bowtie_{e_4} e_3$ further using standard join ordering techniques (in this way, we get two joins involving equality predicates):

$$(e_1 \bowtie_{io \neq ui} e_2) \bowtie_{e_4} e_3 \;=\; (e_3 \bowtie_{bi = ui} e_1) \bowtie_{bn = in \wedge io \neq ui \wedge bb > 2.0 \cdot ir} e_2.$$

6.5.2.3 *Evaluation.* The following table shows the results from our measurements. As can be seen clearly, both unnested variants outperform the nested version easily. Again, Semijoin 2 is slower because we require the first θ-join to be order-preserving while for Semijoin 1 no such restriction exists for any of the θ-joins.

Size	100	1000	10,000
Nested	56.69 s	3041.22 s	∞
Semijoin 1	0.21 s	0.80 s	81.21 s
Semijoin 2	0.63 s	14.25 s	1176.2 s

6.5.3 *General Comparisons.* In the previous sections, we assumed all comparisons to be value-based. Now we show how we can handle general comparisons with our approach. The main idea is to transform the general comparisons into explicit existentially quantified expressions with value comparisons during normalization. Then, after the translation into the algebra, we use our techniques to unnest these expressions. Following that, we can continue with unnesting the actual nested query as shown before.

Consider the following example query, in which we are looking for books that are sold below the price mentioned in some review (e.g., suggested retail price):

```
for $b in doc("bib.xml")//book
where some $e in doc("reviews.xml")//entry[title = $b/title]
      satisfies $e/price > $b/price
return
  <cheap-book>
     { $b/title, $b/price }
  </cheap-book>
```

During normalization we expand the range expressions of the quantified queries to FLWR expressions. Normalization of the quantified queries ensures that all comparisons become value comparisons[2]:

```
for $b in doc("bib.xml")//book
let $bt  := $b/title
let $bp  := $b/price
let $bs  := ($bt, $bp)
let $res := <cheap-book> { $bs } </cheap-book>
where some $e in doc("reviews.xml")//entry
       let $et := $e/title
       let $ep := $e/price
      where some $ets in $et
            satisfies some $bts in $bt
                      satisfies $ets eq $bts
      satisfies some $eps in $ep
                satisfies some $bps in $bp
                          satisfies $eps gt $bps
return $res
```

Translating this into our algebra yields

$$\Pi_{res}(\sigma_{e_2}(e_0)),$$

where

$$
\begin{aligned}
e_0 &:= \chi_{res:\mathcal{C}(\text{elem},s1,bs)}(\chi_{bs:(bt,bp)}(\chi_{bp:b/price}\\
&\qquad (\chi_{bt:b/title}(\Upsilon_{b:doc1//book}(\Box))))),\\
e_1 &:= \chi_{ep:e/price}(\chi_{et:e/title}\\
&\qquad (\Upsilon_{e:doc2//entry}(\Box))),\\
e_2 &:= \exists et1 \in (\sigma_{e_3}(e_1)) : e_6,\\
e_3 &:= \exists et2 \in e_4 : \exists bt1 \in e_5 : ets = bts,\\
e_4 &:= \Upsilon_{ets:et}(\Box),\\
e_5 &:= \Upsilon_{bts:bt}(\Box),\\
e_6 &:= \exists et3 \in e_7 :\\
&\qquad \exists bt2 \in e_8 : eps > bps,\\
e_7 &:= \Upsilon_{eps:ep}(\Box),\\
e_8 &:= \Upsilon_{bps:bp}(\Box),
\end{aligned}
$$

and

```
doc1 := doc("bib.xml"),
doc2 := doc("reviews.xml"),
s1   := "cheap-book".
```

Dependencies between different expressions (the evaluation of e_5 and e_8 depends on e_0, while that of e_4 and e_7 depends on e_1) do not make our job any easier. That means that in the first step of unnesting the introduced quantified expressions, we are forced to use Eqv. 1. However, we can improve our situation by decoupling the range expression in e_2, $\sigma_{e_3}(e_1)$, from the outer query block. We do this by pushing the independent parts of the predicates in e_2 into the range expression and moving the dependent parts into the range predicate:

$$
\begin{aligned}
e_2 &= \exists et1 \in (\sigma_{e_3}(e_1)) : e_6\\
&\stackrel{(1)}{=} \exists et1 \in (\Pi^{tid_{i_1}}_{\mathcal{A}(e_1)}(\sigma_{\exists bt1 \in e_5 : ets=bts}(\Upsilon_{\mathcal{A}(e_4):e4}(tid_{i_1}(e_1))))) :\\
&\qquad \exists et3 \in e_7 : \exists bt2 \in e_8 : eps > bps
\end{aligned}
$$

[2]Here and in the sequel we omit conversions on the sequences and types for readability. We refer the reader to Draper et al. [2005] for details.

$$\overset{(8)}{=} \exists et1 \in (\sigma_{\exists et3 \in e_7 : \exists bt2 \in e_8 : eps > bps}(\Pi^{tid_{i_1}}_{\mathcal{A}(e_1)}(\sigma_{\exists bt1 \in e_5 : ets = bts}$$
$$(\Upsilon_{\mathcal{A}(e_4):e4}(tid_{i_1}(e_1)))))) : \text{true}$$
$$\overset{(1)}{=} \exists et1 \in (\Pi^{tid_{i_2}}_{\mathcal{A}(e_1)}(\sigma_{\exists bt2 \in e_8 : eps > bps}(\Upsilon_{\mathcal{A}(e_7):e_7}(tid_{i_2}$$
$$(\Pi^{tid_{i_1}}_{\mathcal{A}(e_1)}(\sigma_{\exists bt1 \in e_5 : ets = bts}(\Upsilon_{\mathcal{A}(e_4):e4}(tid_{i_1}(e_1)))))))))$$
$$\overset{(9)}{=} \exists et1 \in (\sigma_{\exists bt2 \in e_8 : eps > bps}(\Upsilon_{\mathcal{A}(e_7):e_7}(\sigma_{\exists bt1 \in e_5 : ets = bts}(\Upsilon_{\mathcal{A}(e_4):e4}(e_1)))))$$
$$\overset{(8)}{=} \exists et1 \in (\Upsilon_{\mathcal{A}(e_7):e_7}(\Upsilon_{\mathcal{A}(e_4):e4}(e_1))) :$$
$$(\exists bt2 \in e_8 : eps > bps) \wedge (\exists bt1 \in e_5 : ets = bts).$$

In the last but one step, we also eliminate the tid operators as they are not needed anymore (as both projections on the tids have been removed). To be able to apply Eqv. 8 twice in the last step, we exchanged the positions of $\Upsilon_{\mathcal{A}(e_7):e_7}$ and $\sigma_{\exists bt2 \in e_5 : ets = bts}$, which poses no problem, as e_7 is not connected to the selection predicate in any way.

After having removed the level of nesting introduced by the general comparisons, we could now continue with the unnesting of the actual query. As we have already shown how to proceed with nested queries containing value comparisons in the previous examples, we leave it out here.

7. UNIVERSAL QUANTIFIERS

We start this section with an example to motivate unnesting queries containing universal quantifiers. Then we introduce a general optimization strategy and present rules for unnesting and rewriting algebraic expressions. The application of these rules to typical query classes follows.

7.1 Motivating Example

As a motivating example for universal quantifiers we present a query in which we want to find all auction items that only have valid bids (all bids are at least as high as the reserve price):

```
for $i in doc("items.xml")//itemtuple
where every $b in doc("bids.xml")//bidtuple
                                        [itemno eq $i/itemno]
      satisfies $b/bid ge $i/reserveprice
return $i/itemno
```

Normalizing and translating this query results in the following algebraic expression:

$$\Pi_{ii}(\sigma_{\forall bt \in \sigma_{bi=ii}(e_2) : bb \geq ir}(e_1)),$$

where

$$e_1 := \chi_{ii:i/itemno}(\chi_{ir:i/reserveprice}(\Upsilon_{i:doc(\text{“}items.xml\text{”})//itemtuple}(\Box))),$$
$$e_2 := \chi_{bb:b/bid}(\chi_{bi:b/itemno}(\Upsilon_{b:doc(\text{“}bids.xml\text{”})//bidtuple}(\Box))).$$

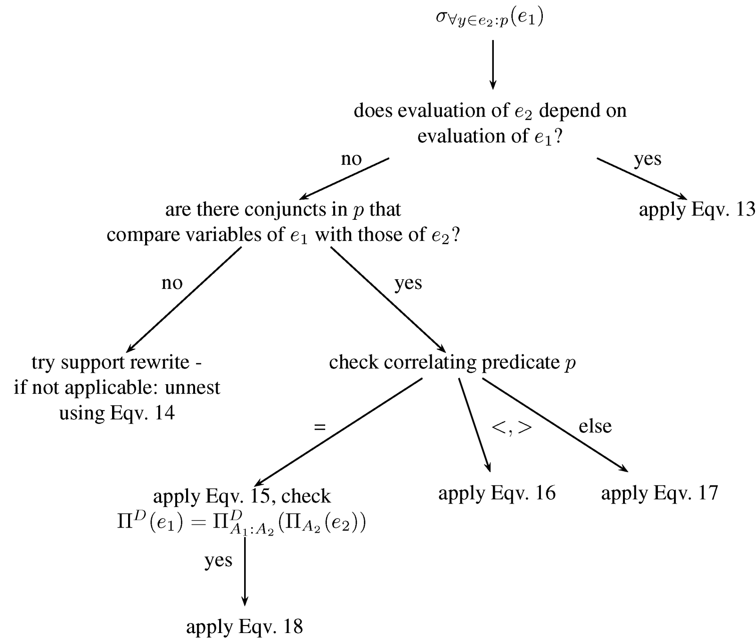

Fig. 9. Decision tree for universally quantified queries.

$$\sigma_{\forall x \in (e_2):p}(e_1) = e_1 \triangleright_{A_1=A_3} \Pi_{A_3:A_1}(\sigma_{\neg p}(\Upsilon_{\mathcal{A}(e_2):e_2}(e_1))) \quad (13)$$

$$\sigma_{\forall x \in (e_2):p}(e_1) = e_1 \triangleright_{\neg p} e_2 \quad (14)$$

$$\sigma_{\forall x \in (\sigma_{A_1=A_2}(e_2)):p}(e_1) = e_1 \triangleright_{A_1=A_2 \wedge \neg p} e_2 \quad (15)$$

$$\sigma_{\forall x \in (\sigma_{A_1 \theta A_2}(e_2)):p}(e_1) = \sigma_{A_1 \neg \theta aggr_{A_2}(\sigma_{\neg p}(e_2))}(e_1) \quad (16)$$

$$\sigma_{\forall x \in (\sigma_{A_1 \theta A_2}(e_2)):p}(e_1) = (e_1) \triangleright_{A_1=A_3} (\Pi_{A_3:A_1}(e_1 \bowtie_{A_1 \theta A_2 \wedge \neg p} e_2)) \quad (17)$$

$$\Pi^D(e_1) \triangleright_{A_1=A_2} (\sigma_p(e_2)) = \sigma_{c=0}(\Pi_{A_1:A_2}(\Gamma_{c;=A_2;count \circ \sigma_p}(e_2))) \quad (18)$$

Fig. 10. Unnesting equivalences for universally quantified queries.

The pattern for universally quantified expressions can be easily identified in the translated version of the query. The general strategy for unnesting these expressions is given in the following section.

7.2 Optimization Strategy

The strategy for unnesting universally quantified expressions is very similar to that used for existentially quantified expressions. (See Figure 9 for the decision tree and Figure 10 for the equivalences.) Again, we try to apply the most special rewrite rule possible.

For our motivation example, this means that we end up at Eqv. 15. Applying this equivalence to our example yields (note that we have to negate the range predicate in the antijoin):

$$\Pi_{ii}((e_1) \triangleright_{bi=ii \wedge bb<ir} (e_2)).$$

Let us give a word of caution related to pushing conjuncts of p that only refer to e_2 (conjuncts pushed into e_1 can be handled as in the case of existential quantification). If a conjunct pushed into e_2 filters out even a single tuple, then the quantified expression returns an empty answer. During query evaluation, this can be used by first evaluating e_2 and aborting the evaluation immediately after a tuple is filtered out by a pushed conjunct of p.

7.3 Equivalences for Unnesting

Figure 10 lists the equivalences for universal quantification. For each unnesting equivalence in Section 6, we have a universally quantified counterpart. We proceed by discussing the equivalences in more detail:

—*Equivalence* 13:
  - —*Preconditions*: Expression e_1 and e_2 cannot be evaluated independently, that is, $\mathcal{F}(e_2) \cap \mathcal{A}(e_1) \neq \emptyset$.
  - —*Basic idea*: We use an unnest map operator to evaluate the subexpression e_2 depending on the current tuple in e_1. If we find at least one tuple that satisfies the negation of the predicate p, then the corresponding tuple in the outer expression e_1 finds a join partner and will be filtered out by the antijoin.

—*Equivalence* 14:
  - —*Preconditions*: Expression e_1 and e_2 can be evaluated independently, that is, $\mathcal{F}(e_2) \cap \mathcal{A}(e_1) = \emptyset$.
  - —*Basic idea*: At first glance, this equivalence looks quite simple. However, when p does not correlate e_1 and e_2, then the evaluation of this expression has to be done in a nested-loop fashion.

—*Equivalence* 15:
  - —*Preconditions*: The evaluation of e_2 does not depend on e_1, that is, $\mathcal{F}(e_2) \cap \mathcal{A}(e_1) = \emptyset$ and e_1 and e_2 are correlated by an equality predicate.
  - —*Basic idea*: We fall back on an antijoin operator. As e_2 does not depend on e_1, we do not need the unnest map found in Eqv. 13.

—*Equivalence* 18:
  - —*Preconditions*: This equivalence is a special case of Eqv. 15. An additional precondition is $\Pi^D(e_1) = \Pi^D_{A_1:A_2}(\Pi_{A_2}(e_2))$.
  - —*Basic idea*: This equivalence is the counterpart of Eqv. 6 for existential quantification. It avoids to evaluate the same subexpression multiple times if the condition check $\Pi^D(e_1) = \Pi^D_{A_1:A_2}(\Pi_{A_2}(e_2))$ holds. For universal quantification, we need to make sure that no tuple exists that satisfies the predicate p.

—*Equivalence* 16:
  - —*Preconditions*: Same preconditions as for Eqv. 15. Depending on the comparison operator θ in p, we have the following assignments:

θ	$\neg\theta$	*aggr*
$>, \geq$	$\leq, <$	*min*
$<, \leq$	$\geq, >$	*max*

$$\forall x \in e_1 : \forall y \in e_2 : p \;=\; \forall y \in e_2 : \forall x \in e_1 : p \qquad (19)$$

$$\forall x \in e_1 : \neg p \vee q \;=\; \forall x \in (\sigma_p(e_1)) : q \qquad (20)$$

$$p \wedge \forall x \in e_1 : q \;=\; \forall x \in e_1 : p \wedge q \qquad (21)$$

$$\begin{aligned} \sigma_{\forall x \in e_2 : p \wedge \forall y \in e_3 : q}(e_1) &= \sigma_{\forall x \in e_2 : p}(\sigma_{\forall y \in e_3 : q}(e_1)) \\ &= \sigma_{\forall y \in e_3 : q}(\sigma_{\forall x \in e_2 : p}(e_1)) \qquad (22) \end{aligned}$$

Fig. 11. Support rewrites for universally quantified queries.

—*Basic idea*: If the comparison operator $\theta \in \{<, \leq, \geq, >\}$, we just need to compare the value of A_1 to the minimal or maximal value of A_2, respectively. For universal quantification, a tuple of e_1 belongs to the answer set if the value for A_1 does not overlap with the range of values that do not satisfy the predicate p.

Similarly to Eqv. 4, we have to be careful when handling the special case $e_2 = \epsilon$: for universal quantification, it is automatically evaluated to true—for example, the aggregated value can be initialized to ∞ or $-\infty$ depending on *aggr*. In addition, we must be careful with the semantics of the aggregate function *aggr* and the general comparison. The resulting unnested expression can be unnested further with rewrites of Section 8.

—*Equivalence* 17:

—*Preconditions*: Same preconditions as for Eqv. 15. But now predicate p can contain arbitrary Boolean expressions.

—*Basic idea*: The θ-join is delegated to an ordinary join operator, which does not even have to be order-preserving. The outer antijoin preserves the order of the tuples in expression e_1.

7.4 Support Rewrites

Usually, we will have the same problems applying unnesting equivalences to universally quantified expressions as to existentially quantified ones: they may not be immediately applicable. Therefore, we need rules to rewrite universally quantified expressions, bringing them into the right shape. In general, we follow the same two strategies as in Section 6.4, reducing the number of free variables in a subexpression that is to be unnested and minimizing the distance between correlated query blocks.

Let us now take a look at the rewrite rules (summarized in Figure 11). This is not a complete list; more rules can be found in the literature (e.g., Bry [1989]; Jarke and Koch [1984]; Steenhagen [1995]). Again, we refer the reader to Electronic Appendix A.2 for more a detailed discussion of these rewrites.

7.5 Example Queries

As the overall strategies for unnesting universally quantified expressions are very similar to those for unnesting existential quantifiers, we restrict ourselves to two example queries. (Further examples can be found in May et al. [2004b].)

The first one involves general (nonequality) range predicates, as these are more difficult to unnest. The second query uses a combination of existential

and universal quantification. For both queries, we skip the description of the normalization step and proceed directly to the description of the completely translated expression and its unnesting. For completeness, we give the normalized queries in the Electronic Appendix B.

7.5.1 *Nonequality Correlating Predicates.* Our first example query is an extension of the motivating query from the beginning of this section. In addition to checking the reserve price, we also make sure that a bid was placed in the specified period of time.

```
for $i in doc("items.xml")//itemtuple
where every $b in doc("bids.xml")//bidtuple
                                    [itemno eq $i/itemno]
            satisfies ($b/bid ge $i/reserveprice
                  and  $b/bid_date ge $i/startdate
                  and  $b/bid_date le $i/enddate)
return $i/itemno
```

After having normalized and translated this query, we arrive at the following algebraic expression. Again, we exploit the fact that child nodes occur exactly once.

$$\Pi_{ii}(\sigma_{\forall bt \in (\sigma_{bi=ii}(e_2)):bb \geq ir \wedge bd \geq is \wedge bd \leq ie}(e_1)),$$

where

$$e_1 := \chi_{ie:i/enddate}(\chi_{is:i/startdate}(\chi_{ir:i/reserveprice}(\chi_{ii:i/itemno}(\Upsilon_{i:doc1//itemtuple}(\Box))))),$$

$$e_2 = \chi_{bd:b/biddate}(\chi_{bb:b/bid}(\chi_{bi:b/itemno}(\Upsilon_{b:doc2//bidtuple}(\Box)))),$$

and

doc1 := doc("items.xml"),
doc2 := doc("bids.xml").

7.5.1.1 *Antijoin 1.* Only one of the equivalences is immediately applicable: Eqv. 15. Applying this equivalence results in the following expression (note that the predicate $p = bb \geq ir \wedge bd \geq is \wedge bd \leq ie$ is negated for the antijoin):

$$\Pi_{ii}(\sigma_{\forall bt \in (\sigma_{bi=ii}(e_2)):bb \geq ir \wedge bd \geq is \wedge bd \leq ie}(e_1))$$
$$\stackrel{(15)}{=} \Pi_{ii}(e_1 \triangleright_{bi=ii \wedge (bb<ir \vee bd<is \vee bd>ie)} e_2)$$

7.5.1.2 *Antijoin 2.* Applying the support rewrite rule Eqv. 20 allows us to push down the predicate p. After that, we can merge it with the other selection and interpret the resulting predicate as a general θ-comparison, which matches the left hand side of Eqv. 17:

$$\Pi_{ii}(\sigma_{\forall bt \in (\sigma_{bi=ii}(e_2)):(bb \geq ir \wedge bd \geq is \wedge bd \leq ie) \vee false}(e_1))$$
$$\stackrel{(20)}{=} \Pi_{ii}(\sigma_{\forall bt \in (\sigma_{bb<ir \vee bd<is \vee bd>ie}(\sigma_{bi=ii}(e_2))):false}(e_1))$$
$$\stackrel{(17)}{=} \Pi_{ii}(e_3 \triangleright_{ii=ii'} \Pi_{ii':ii}(e_1 \bowtie_{bi=ii \wedge (bb<ir \vee bd<is \vee bd>ie) \wedge true} e_2))$$

7.5.1.3 *Evaluation.* The nested version of the query was implemented using a negated existential quantifier: $\Pi_{ii}(\sigma_{\nexists bt \in \sigma_{bi=ii}(e_2):bb<ir \vee bd<is \vee bd>ie}(e_1))$. This

performs better because, as soon as we find a tuple that satisfies the predicate, we can stop the evaluation of the nested query and return *false*.

In the table below, we present the execution times for the nested and the two unnested variants of the example query. As can be clearly seen, both unnested versions outperform the nested one.

Size	100	1000	10,000
Nested	0.47 s	11.39 s	819.71 s
Antijoin 1	0.21 s	1.01 s	8.54 s
Antijoin 2	0.23 s	1.68 s	23.98 s

7.5.2 *Combining Existential and Universal Quantifiers.* An interesting case that we have not looked at yet is mixing existentially and universally quantified expressions that are correlated with each other and the outer query block.[3] The following query returns the names of all users that bid on every item:

```
for $u in doc("users.xml")//usertuple
where every $i in doc("items.xml")//itemtuple
      satisfies some $b in doc("bids.xml")//bidtuple
                satisfies ($i/itemno eq $b/itemno and
                           $u/userid eq $b/userid)
return $u/name
```

After having normalized and translated this query, we get the following algebraic expression:

$$\Pi_{un}(\sigma_{\forall it \in e_2 : \exists bt \in e_3 : in=bn \wedge ui=bi}(e_1)),$$

where

$$e_1 := \chi_{un:u/name}(\chi_{ui:u/userid}(\Upsilon_{u:doc1//usertuple}(\Box))),$$

$$e_2 := \chi_{in:i/itemno}(\Upsilon_{i:doc2//itemtuple}(\Box)),$$

$$e_3 := \chi_{bi:b/userid}(\chi_{bn:b/itemno}(\Upsilon_{b:doc3//bidtuple}(\Box))),$$

and

```
doc1 := doc("users.xml"),
doc2 := doc("items.xml"),
doc3 := doc("bids.xml").
```

7.5.2.1 *Unnesting.* When we try to unnest the translated query, we observe that we cannot use Eqv. 3 directly because the range predicate of the existential quantifier contains a quantified expression. We cannot apply Eqv. 15 either because the range predicate of the universal quantifier contains free variables that are not bound by the range expression of the universal quantifier.

We remedy this situation by pushing down the range predicates (once for the existential quantifier using Eqv. 8 and once for the universal quantifier using Eqv. 20). After that, we can unnest the inner query block by applying Eqv. 15 and then use Eqv. 13 for the final unnesting step (we use unnesting rules for universal quantifiers twice because by pushing down the existential quantifier

[3]The result of the innermost quantified expression depends on variable bindings passed by the two outer expressions.

we turn it into a universal quantifier):

$$
\begin{aligned}
& \Pi_{un}(\sigma_{\forall it \in e_2 : \exists bt \in e_3 : in=bn \wedge ui=bi}(e_1)) \\
\stackrel{(8)}{=} \; & \Pi_{un}(\sigma_{\forall it \in e_2 : (\exists bt \in \sigma_{in=bn}(e_3) : ui=bi) \vee false}(e_1)) \\
\stackrel{(20)}{=} \; & \Pi_{un}(\sigma_{\forall it \in (\sigma_{\forall bt \in (\sigma_{in=bn}(e_3) : ui \neq bi)}(e_2)) : false}(e_1)) \\
\stackrel{(15)}{=} \; & \Pi_{un}(\sigma_{\forall it \in (e_2 \triangleright_{in=bn \wedge ui=bi} e_3) : false}(e_1)) \\
\stackrel{(13)}{=} \; & \Pi_{un}(e_1 \triangleright_{\mathcal{A}(e_1)=\mathcal{A}(e_1)'} (\Pi_{\mathcal{A}(e_1)':\mathcal{A}(e_1)}(\sigma_{true}(\Upsilon_{\mathcal{A}(e_2):(e_2 \triangleright_{in=bn \wedge ui=bi} e_3)}(e_1))))) \\
= \; & \Pi_{un}(e_1 \triangleright_{\mathcal{A}(e_1)=\mathcal{A}(e_1)'} (\Pi_{\mathcal{A}(e_1)':\mathcal{A}(e_1)}(\Upsilon_{\mathcal{A}(e_2):(e_2 \triangleright_{in=bn \wedge ui=bi} e_3)}(e_1))))
\end{aligned}
$$

7.5.2.2 *Evaluation.* This is one of the rare cases where the unnested version of the query was not faster than the nested one. This underscores the importance of an algebraic approach in which different alternatives can be compared in a cost-based manner.

We compared two different unnested versions of the query. The first version is the expression above after applying Eqv. 15. In the second version, we introduced an unnest map operator. This version is more efficient, as we can stop evaluating the antijoin in the unnest map operator as soon as it produces a tuple (in that case, the current tuple of e_1 will be disqualified by the other antijoin operator). Although it is slightly slower than the nested version, it is still in the same ball park.

The following table summarizes our experimental results for this example query:

Size	100	1000	10,000
Nested	0.50 s	11.12 s	788.14 s
Unnested	0.31 s	18.98 s	2009.24 s
Unnested + unnest map	0.80 s	15.18 s	957.25 s

8. IMPLICIT GROUPING

Unlike SQL or OQL, which feature grouping clauses, XQuery does not have explicit grouping constructs. Grouping in XQuery is done via nested queries; hence we use the term *implicit grouping*. Although some researchers advocate introducing explicit grouping into XQuery [Borkar and Carey 2004; Beyer et al. 2005], this does not mean that the option of implicit grouping will just vanish. Consequently, an optimization approach would have to be able to handle both cases. In the remainder of this section, we present unnesting techniques for expressions containing implicit grouping.

8.1 Motivating Example

As a motivating example, we pick up the query from Section 5 again. In this query we rearrange all books such that they are grouped by their publishers:

```
for $p in distinct-values(doc("bib.xml")//publisher)
return
  <publisher>
```

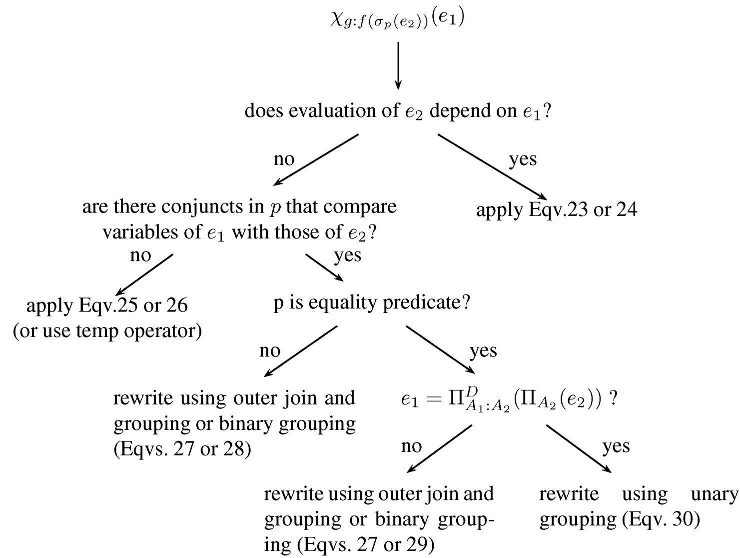

Fig. 12. Decision tree for implicit grouping; Equivalences and decisions refer to the case of value comparisons in predicates.

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

Recalling Section 4, we know that the normalization step for implicit grouping basically consisted of pulling up the *return* clause into a *let* clause and translating this *let* clause into a map operator. After having translated the normalized version of this query, we arrive at the basic pattern for implicit grouping (this time considering the *return* clause):

$$\Pi_{res}(\chi_{res:\mathcal{C}(\text{elem},s1,sq)}(\chi_{sq:(pn,t)}(\chi_{pn:\mathcal{C}(\text{elem},s2,p)}(\chi_{t:\Pi_{t2}(\sigma_{p=p2}(e_2))}(e_1))))),$$

where

$e_1 := \Upsilon_{p:\Pi^D(doc//publisher)}(\Box)$,

$e_2 := \chi_{t2:b/title}(\chi_{p2:b/publisher}(\Upsilon_{b:doc//book}(\Box)))$,

and

```
doc = doc("bib.xml"),
s1  = "publisher",
s2  = "name".
```

We have now arrived at the standard pattern for implicit grouping. Strategies for unnesting this algebraic pattern can be found in the following section.

8.2 Optimization Strategy

The strategy employed for unnesting expressions containing implicit grouping (see Figure 12 for an overview and Figure 13 for the equivalences) is similar to that for quantified expressions. First we check whether e_1 and e_2 can be evaluated independently of each other. If not, we have to rely on an unnest map operator. Otherwise, we take a look at the predicate p. Here, we distinguish the

$$\chi_{g:f(\sigma_p(e_2))}(e_1) = e_1 \Gamma_{g;\mathcal{A}(e_1)=A'_1;f}(\Pi_{A'_1:\mathcal{A}(e_1)}(\sigma_p(\Upsilon_{\mathcal{A}(e_2):e_2}(\Pi^D_{\mathcal{A}(e_1)}(e_1))))) \quad (23)$$

$$\chi_{g:f(\sigma_p(e_2))}(e_1) = \Pi_{\overline{A_3}}(e_1 ⟕^{g:f(\epsilon)}_{\mathcal{A}(e_1)=A_3}(\Pi_{A_3:\mathcal{A}(e_1)}(\Gamma_{g;=\mathcal{A}(e_1);f}(\sigma_p(\Upsilon_{\mathcal{A}(e_2):e_2}(\Pi^D_{\mathcal{A}(e_1)}(e_1))))))) \quad (24)$$

$$\chi_{g:f(\sigma_p(e_2))}(e_1) = e_1 \Gamma_{g;\mathcal{A}(e_1)=A'_1;f}(\Pi_{A'_1:\mathcal{A}(e_1)}(\sigma_p(\Pi^D_{\mathcal{A}(e_1)}(e_1) \times e_2))) \quad (25)$$

$$\chi_{g:f(\sigma_p(e_2))}(e_1) = \Pi_{\overline{A_3}}(e_1 ⟕^{g:f(\epsilon)}_{\mathcal{A}(e_1)=A_3}(\Pi_{A_3:\mathcal{A}(e_1)}(\Gamma_{g;=\mathcal{A}(e_1);f}(\sigma_p(\Pi^D_{\mathcal{A}(e_1)}(e_1) \times e_2))))) \quad (26)$$

$$\chi_{g:f(\sigma_{A_1 \theta A_2}(e_2))}(e_1) = e_1 \Gamma_{g;A_1 \theta A_2;f} e_2 \quad (27)$$

$$\chi_{g:f(\sigma_{A_1 \theta A_2}(e_2))}(e_1) = \Pi_{\overline{A_3}}(e_1 ⟕^{g:f(\epsilon)}_{A_1=A_3}(\Pi_{A_3:A_1}(\Gamma_{g;=A_1;f}(\Pi^D_{A_1}(e_1) ⟕_{A_1 \theta A_2} e_2)))) \quad (28)$$

$$\chi_{g:f(\sigma_{A_1=A_2}(e_2))}(e_1) = \Pi_{\overline{A_2}}(e_1 ⟕^{g:f(\epsilon)}_{A_1=A_2}(\Gamma_{g;=A_2;f}(e_2))) \quad (29)$$

$$\chi_{g:f(\sigma_{A_1=A_2}(e_2))}(e_1) = \Pi_{A_1:A_2}(\Gamma_{g;=A_2;f}(e_2)) \quad (30)$$

Fig. 13. Unnesting equivalences for implicit grouping.

cases that e_1 and e_2

- —are not correlated via p,
- —are correlated via a complex (non-equality) comparison operator,
- —are correlated via an equality predicate.

For the equality predicate, there is room for further optimization if e_1 and e_2 produce identical sequences (save duplicates and additional attributes in e_2).

About our motivational example query we know the following: e_1 and e_2 can be evaluated independently, they are correlated via an equality predicate, and $e_1 = \Pi^D_{p:p2}(\Pi_{p2}(e_2))$. So we would apply Eqv. 30 in this case:

$$\Pi_{res}(\chi_{res:\mathcal{C}(\text{elem},s1,sq)}(\chi_{sq:(pn,t)}(\chi_{pn:\mathcal{C}(\text{elem},s2,p)}(\Pi_{p:p2}(\Gamma_{t;=p2;\Pi_{t2}}(\chi_{t2:b/title}(\chi_{p2:b/publisher}(\Upsilon_{b:doc//book}(\Box)))))))))$$

8.3 Unnesting Equivalences

In Figure 13 the equivalences for unnesting implicit grouping can be found. As for unnesting quantified expressions before, we state the preconditions for applying an equivalence and give a brief description of the underlying idea. For most patterns, we present two alternatives: one alternative that uses an outer join and unary grouping and another alternative that uses binary grouping. The first alternative uses operators that are more generally available in database systems, while the second alternative often results in more efficient plans. We will come back to this in our example queries.

- —*Equivalence* 23:
  - —*Preconditions*: e_1 and e_2 cannot be evaluated independently (formally speaking, $\mathcal{F}(e_2) \cap \mathcal{A}(e_1) \neq \emptyset$).

—*Basic idea*: For each tuple in e_1, we collect the corresponding tuples in e_2 via a binary grouping operator and apply the function f to all tuples in the corresponding group. We generate the tuples of the expression e_2 by combining all tuples t_1 in e_1 with all tuples in $e_2(t_1)$ via an unnest map operator and then apply p.

—*Equivalence* 24:

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

—*Basic idea*: This is a variant of Eqv. 23. Instead of a binary grouping operator, we use a unary one. In order to avoid the “count bug” (i.e., losing a tuple due to an empty group), we use an outer join operator. The main motivation for this variant is the fact that not every DBMS supports a binary grouping operator.

—*Equivalence* 25:

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

—*Basic idea*: This equivalence looks very similar to Eqv. 23 except that e_2 can be evaluated independently of e_1 and, therefore, is connected via a Cartesian product to each tuple in e_1. For each tuple in e_1, the tuples in e_2 are grouped via a binary grouping operator. If the predicate p does not refer to attributes in e_1, we could also compute $f(\sigma_p(e_2))$, store the result temporarily, and attach this result to each tuple in e_1 (as in this case, we have the same group for each tuple in e_1).

—*Equivalence* 26:

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

—*Basic idea*: This is the outer join/unary grouping variant of Eqv. 25.

—*Equivalence* 27:

—*Preconditions*: e_1 and e_2 can be evaluated independently, and e_1 and e_2 are correlated with a predicate containing a θ-comparison.

—*Basic idea*: As we know more about the attributes involved in the predicate, we can group the tuples in e_2 directly without connecting them to tuples in e_1 first. The predicate correlating e_1 and e_2 is now an element of the binary grouping operator.

—*Equivalence* 28:

—*Preconditions*: e_1 and e_2 can be evaluated independently, and e_1 and e_2 are correlated with a predicate containing a θ-comparison.

—*Basic idea*: This is the outer join/unary grouping variant of Eqv. 27. The elegant integration of the correlating predicate into the grouping operator is not possible here, as we use a unary grouping operator. So this looks more like Eqv. 26, replacing the cross product with a θ-join. This technique is also known as *magic set decorrelation* [Seshadri et al. 1996b]. (The θ-join between e_1 and e_2 needs only be order-preserving if the correct computation of f relies on ordered tuples.)

—*Equivalence* 29:

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

$$\Pi_{\overline{g_1}}(\chi_{g_2:f(g_1)}(\chi_{g_1:e_2}(e_1))) = \chi_{g2:f(e_2)}(e_1) \quad (31)$$

$$\Upsilon_{A:\Upsilon_{B:e_2}(\Box)}(e_1) = \Upsilon_{A:e_2}(e_1) \quad (32)$$

$$\Pi^{tid_B}_{A_i}(tid_B(e_1 \times e_2)) = \Pi^{tid_{B_1},tid_{B_2}}_{A_i}(tid_{B_1}(e_1) \times tid_{B_2}(e_2)) \quad (33)$$

$$\Pi^{tid_B}_{A_1}(\Pi^{tid_C}_{A_1}(e_1)) = \Pi^{tid_B}_{A_1}(e_1) \quad (34)$$

Fig. 14. Support rewrites.

—*Basic idea*: In the special case of an equality predicate, the function f is computed for each possible group identified in e_2. The main advantage is that the result of the grouping needs only be evaluated once and can be materialized. The variant using a binary grouping operator is already covered by Eqv. 27.

—*Equivalence* 30:

—*Preconditions*: e_1 and e_2 can be evaluated independently, and e_1 and e_2 are correlated with an equality predicate. Also, $e_1 = \Pi^D_{A_1:A_2}(\Pi_{A_2}(e_2))$, assuming that $A_i = \mathcal{A}(e_i)$.

—*Basic idea*: If we know that there are no empty groups (because e_1 and e_2 contain the same attribute values, save attribute names and duplicates), we do not need to evaluate e_1, but can do a unary grouping on e_2.

8.4 Support Rewrites

As the unnesting equivalences from Section 8.3 expect certain patterns, we may have to rewrite nested algebraic expressions to match these patterns. Figure 14 gives a quick overview of the support rewrite rules for unnesting implicit grouping. The underlying ideas are explained in the following:

—*Equivalence* 31:

—*Preconditions*: None.

—*Basic idea*: This rewrite merges two map operators into one and can be used when the result of one map operator is just consumed by another map operator and does not appear anywhere else afterward. This is useful, as it saves us from constructing the (possibly sequence-valued) attribute g_1.

—*Equivalence* 32:

—*Preconditions*: None.

—*Basic idea*: This rewrite merges two unnest map operators into one. We eliminate an unnecessary step of nesting and then unnesting again.

—*Equivalence* 33:

—*Preconditions*: None.

—*Basic idea*: We break up a tid operator that assigns a unique id to each tuple of a Cartesian product into two tid operators operating on the subexpressions of the product. We can do this because each tuple of the cross product is still identifiable as before. When discarding duplicates, we have to look at both tids. This rewrite allows us to push down operators into the cross product (e.g., selections turning the product into a join).

—*Equivalence* 34:
 —*Preconditions*: The tids are assigned in such a way in e_1 that the attribute B is functionally dependent on C ($C \rightarrow B$).
 —*Basic idea*: In this case, we can get rid of the inner duplicate elimination, as each tuple that is filtered out by $\Pi^{tid_C}_{A_1}$ will also be filtered out by $\Pi^{tid_B}_{A_1}$.

Queries with implicit grouping involving general comparison operators are handled by transforming them into existentially quantified expressions during normalization. It follows that all equivalences (unnesting and support rewrite) found in Section 6 can also be used as support rewrite rules when unnesting implicit grouping expressions containing general comparisons.

8.5 Example Queries

Let us now show how to apply the unnesting equivalences to concrete example queries. Since we have already presented some of the equivalences in an earlier publication [May et al. 2003], we concentrate on the new equivalences.

We are particularly interested in combining the unnesting rules for grouping with those for quantified expressions to demonstrate the full power of our framework. Depending on whether the variables used in our queries are atomic or sequence-valued, we have to employ a value-based or a general comparison operator. We distinguish between the variables in the outer query block and those in the inner (implicit grouping) query block. As both sets of variables can be atomic or sequence-valued, we have four different cases.

Due to space constraints we only discuss two of the more complicated cases here: one where the correlation predicate consists of a general comparison with the outer query block producing a sequence-valued attribute and one where both of the inner and outer query block produce sequence-valued attributes. For the two other cases, we refer the reader to Electronic Appendix C. We will also skip the normalization step and refer the reader to Electronic Appendix B for the details.

8.5.1 *Sequence-Valued Attribute in Outer Expression.* In this section we present an example query in which the sequence-valued attribute is located in the outer query block. For each author, we count the number of books that are cheaper than any book written by that particular author.

The main difficulties in evaluating this query efficiently are the following. The values that we group on (i.e., the authors) are not found in the correlating predicate (cf. Beyer et al. [2004, 2005] on the grouping problem). In addition to that, the groups are created based on a nonequality predicate.

```
for $a in distinct-values(doc("bib.xml")//book/author)
let $ap := doc("bib.xml")//book[$a = author]/price
return
  <cheaper-books>
    { $a },
    <count>
      { count(doc("bib.xml")//book[price < $ap]) }
    </count>
  </cheaper-books>
```

Normalization introduces several new *let* expressions and shifts the implicit grouping out of the *return* block. The translation step produces the following algebraic expression:

$$\Pi_{res}(\chi_{res:\mathcal{C}(\text{elem},s1,sq)}(\chi_{sq:(a,ce)}(\chi_{ce:\mathcal{C}(\text{elem},s2,ct)}(\chi_{ct:count(lp)}$$
$$(\chi_{lp:\Pi_{pb}(\sigma_{\exists pt\in\Upsilon_{pa:ap}(\Box):pp<pa}(e_3))}(\chi_{ap:\Pi_{abp}(\sigma_{\exists at\in\Upsilon_{aa:aba}(\Box):a=aa}(e_2))}(e_1))))))),$$

where

$$e_1 := \Pi^D(\Upsilon_{a:doc//book/author}(\Box)),$$
$$e_2 := \chi_{abp:ab/price}(\chi_{aba:ab/author}(\Upsilon_{ab:doc//book}(\Box))),$$
$$e_3 := \chi_{pp:pb/price}(\Upsilon_{pb:doc//book}(\Box)),$$

and

```
doc := doc("bib.xml"),
s1  := "cheaper-books",
s2  := "count".
```

8.5.1.1 *Binary Grouping.* Unnesting this algebraic expression involves several steps. First, we unnest the inner existentially quantified expression (applying Eqv. 1 as the range predicate depends on e_2). After that, we eliminate a redundant unnest map operator using the support rewrite rule 32. Then we are ready to apply an equivalence for unnesting grouping on the inner map operator. Eqv. 30 is the most efficient variant in this case, resulting in a unary grouping operator on e_2:

$$\stackrel{(1)}{=} \Pi_{res}(\chi_{res:\mathcal{C}(\text{elem},s1,(a,\mathcal{C}(\text{elem},s2,ct)))}$$
$$(\chi_{ct:count(lp)}(\chi_{lp:\Pi_{pb}(\sigma_{\exists pt\in\Upsilon_{pa:ap}(\Box):pp<pa}(e_3))}$$
$$(\chi_{ap:\Pi_{abp}(\Pi^{tid_B}_{\mathcal{A}(e_2)\cup aa}(\sigma_{aa=a}(\Upsilon_{aa:\Upsilon_{aa:aba}(\Box)}(tid_B(e_2)))))}(e_1)))))$$

$$\stackrel{(32)}{=} \Pi_{res}(\chi_{res:\mathcal{C}(\text{elem},s1,(a,\mathcal{C}(\text{elem},s2,ct)))}(\chi_{ct:count(lp)}(\chi_{lp:\Pi_{pb}(\sigma_{\exists pt\in\Upsilon_{pa:ap}(\Box):pp<pa}(e_3))}$$
$$(\chi_{ap:\Pi_{abp}(\Pi^{tid_B}_{\mathcal{A}(e_2)\cup aa}(\sigma_{aa=a}(\Upsilon_{aa:aba}(tid_B(e_2)))))}(e_1)))))$$

$$\stackrel{(30)}{=} \Pi_{res}(\chi_{res:\mathcal{C}(\text{elem},s1,(a,\mathcal{C}(\text{elem},s2,ct)))}(\chi_{ct:count(lp)}(\chi_{lp:\Pi_{pb}(\sigma_{\exists pt\in\Upsilon_{pa:ap}(\Box):pp<pa}(e_3))}$$
$$\underbrace{(\Pi_{a:aa}(\Gamma_{ap;=aa;\Pi_{abp}\circ\Pi^{tid_B}_{\mathcal{A}(e_2)\cup aa}}\Upsilon_{aa:aba}(tid_B(e_2)))}_{e_4}))))$$

In order to keep things readable, we call the inner, unnested expression e_4 in the following. We continue by merging the two remaining map operators via Eqv. 31, prepare the existentially quantified subexpression for unnesting using Eqv. 8, and then unnest it by applying Eqv. 4. As mentioned earlier, we have to be careful with the semantics of function *max*. In our query it has to use string comparison to compute the maximum:

$$\stackrel{(31)}{=} \Pi_{res}(\chi_{res:\mathcal{C}(\text{elem},s1,(a,\mathcal{C}(\text{elem},s2,ct)))}(\chi_{ct:count(\Pi_{pb}(\sigma_{\exists pt\in\Upsilon_{pa:ap}(\Box):pp<pa}(e_3)))}(e_4)))$$

$$\stackrel{(8)}{=} \Pi_{res}(\chi_{res:\mathcal{C}(\text{elem},s1,(a,\mathcal{C}(\text{elem},s2,ct)))}(\chi_{ct:count(\Pi_{pb}(\sigma_{\exists pt\in\sigma_{pp<pa}(\Upsilon_{pa:ap}(\Box))}(e_3)))}(e_4)))$$

$$\stackrel{(4)}{=} \Pi_{res}(\chi_{res:\mathcal{C}(\text{elem},s1,(a,\mathcal{C}(\text{elem},s2,ct)))}(\chi_{ct:count(\Pi_{pb}(\sigma_{pp<\max_{pa}(\Upsilon_{pa:ap}(\Box))}(e_3)))}(e_4)))$$

Finally, we are now ready to unnest the grouping expression containing the count-function. Here we use the variant based on binary grouping (the outer

join/unary grouping variant will be presented in just a moment). After having unnested the expression, we can transform the unnest map operator into a Cartesian product, as the two involved expressions can be evaluated independently of each other. In a last step, we change the selection and cross product into a join operator:

$$\begin{aligned}
&\overset{(23)}{=} \Pi_{res}(\chi_{res:\mathcal{C}(\text{elem},s1,(a,\mathcal{C}(\text{elem},s2,ct)))} \\
&\quad (e_4 \Gamma_{ct;\mathcal{A}(e_4)=A'_4;count\circ\Pi_{pb}} \Pi_{A'_4:\mathcal{A}(e_4)} \\
&\quad\quad (\sigma_{pp<\max_{pa}(\Upsilon_{pa:ap}(\square))}(\Upsilon_{\mathcal{A}(e_3):e_3}(\Pi^D_{\mathcal{A}(e_4)}(e_4)))))) \\
&= \Pi_{res}(\chi_{res:\mathcal{C}(\text{elem},s1,(a,\mathcal{C}(\text{elem},s2,ct)))} \\
&\quad (e_4 \Gamma_{ct;\mathcal{A}(e_4)=A'_4;count\circ\Pi_{pb}} \Pi_{A'_4:\mathcal{A}(e_4)}(\sigma_{pp<\max_{pa}(\Upsilon_{pa:ap}(\square))}(e_4 \times e_3)))) \\
&= \Pi_{res}(\chi_{res:\mathcal{C}(\text{elem},s1,(a,\mathcal{C}(\text{elem},s2,ct)))} \\
&\quad (e_4 \Gamma_{ct;\mathcal{A}(e_4)=A'_4;count\circ\Pi_{pb}} \Pi_{A'_4:\mathcal{A}(e_4)}(e_4 \bowtie_{pp<\max_{pa}(\Upsilon_{pa:ap}(\square))} e_3)))
\end{aligned}$$

8.5.1.2 *Outer Join.* Instead of unnesting via a binary grouping operator (Eqv. 23), we can also apply a combination of outer join and unary grouping (Eqv. 24):

$$\begin{aligned}
&\overset{(24)}{=} \Pi_{res}(\chi_{res:\mathcal{C}(\text{elem},s1,(a,\mathcal{C}(\text{elem},s2,ct)))}(e_4 ⟕^{ct:0}_{\mathcal{A}(e_4)=A'_4} \\
&\quad (\Pi_{A'_4:\mathcal{A}(e_4)}(\Gamma_{ct;=\mathcal{A}(e_4);count\circ\Pi_{pb}} \\
&\quad\quad (\sigma_{pp<\max_{pa}(\Upsilon_{pa:ap}(\square))}(\Upsilon_{\mathcal{A}(e_3):e_3}(\Pi^D_{\mathcal{A}(e_4)}(e_4)))))))) \\
&= \Pi_{res}(\chi_{res:\mathcal{C}(\text{elem},s1,(a,\mathcal{C}(\text{elem},s2,ct)))}(e_4 ⟕^{ct:0}_{\mathcal{A}(e_4)=A'_4} \\
&\quad (\Pi_{A'_4:\mathcal{A}(e_4)}(\Gamma_{ct;=\mathcal{A}(e_4);count\circ\Pi_{pb}}(e_4 \bowtie_{pp<\max_{pa}(\Upsilon_{pa:ap}(\square))} e_3)))))
\end{aligned}$$

8.5.1.3 *Evaluation.* The following table shows the results for the nested and both unnested versions of the query. Again, the evaluation of the unnested expressions is considerably faster than the evaluation of the nested one (with the binary grouping being [slightly] slower than the outer join).

Size	100	1000	10,000
Nested	1.53 s	132.65 s	∞
Binary grouping	0.15 s	1.04 s	64.93 s
Outer join	0.15 s	0.94 s	58.54 s

8.5.2 *Sequence-Valued Attributes in Both Expressions.* We now come to the most complicated case, in which we allow sequence-valued attributes in both query blocks, the outer and the inner one. As an example query, we take a modified version of the query presented in Section C.2.2. For each book, we determine how many books its authors have edited:

```
for $b in doc("bib.xml")//book
return
  <book-editor>
    { $b }
```

```
    <count> { count(for $c in doc("bib.xml")//book
                    where $b/author = $c/editor
                    return $c)
    } </count>
  </book-editor>
```

Normalization and translation into our algebra results in the following expression:

$$\Pi_{res}(\chi_{res:\mathcal{C}(\text{elem},s1,sq)}(\chi_{sq:(b,ci)}(\chi_{ci:\mathcal{C}(\text{elem},s2,ct)}$$
$$(\chi_{ct:count(cc)}(\chi_{cc:\Pi_c(\sigma_{\exists et\in\Upsilon_{e:ce}(\Box):\exists eat\in\Upsilon_{ea:ba}(\Box):e=ea}(e_2))}(e_1)))))),$$

where

$$e_1 := \chi_{ba:b/author}(\Upsilon_{b:doc//book}(\Box)),$$
$$e_2 := \chi_{ce:c/editor}(\Upsilon_{c:doc//book}(\Box)),$$

and

```
doc := doc("bib.xml"),
s1  := "book-editor",
s2  := "count".
```

8.5.2.1 *Binary Grouping.* In a first step, we merge map operators and then unnest the nested implicit grouping expression:

$$\overset{(31)}{=} \Pi_{res}(\chi_{res:\mathcal{C}(\text{elem},s1,(b,\mathcal{C}(\text{elem},s2,ct)))}$$
$$(\chi_{ct:count(\Pi_c(\sigma_{\exists et\in\Upsilon_{e:ce}(\Box):\exists eat\in\Upsilon_{ea:ba}(\Box):e=ea}(e_2)))}(e_1)))$$
$$\overset{(25)}{=} \Pi_{res}(\chi_{res:\mathcal{C}(\text{elem},s1,(b,\mathcal{C}(\text{elem},s2,ct)))}(e_1\Gamma_{ct;\mathcal{A}(e_1)=A_1';count\circ\Pi_c}(\Pi_{A_1':\mathcal{A}(e_1)}$$
$$(\sigma_{\exists et\in\Upsilon_{e:ce}(\Box):\exists eat\in\Upsilon_{ea:ba}(\Box):e=ea}(\Pi^D_{\mathcal{A}(e_1)}(e_1)\times e_2)))))$$

In a second step, we unnest the existentially quantified expressions introduced by the normalization and eliminate unnecessary unnest map operators:

$$\overset{(1)}{=} \Pi_{res}(\chi_{res:\mathcal{C}(\text{elem},s1,(b,\mathcal{C}(\text{elem},s2,ct)))}(e_1\Gamma_{ct;\mathcal{A}(e_1)=A_1';count\circ\Pi_c}(\Pi_{A_1':\mathcal{A}(e_1)}$$
$$(\Pi^{tid_A}_{\mathcal{A}(e_1)\cup\mathcal{A}(e_2)}(\sigma_{\exists eat\in\Upsilon_{ea:ba}(\Box):e=ea}$$
$$(\Upsilon_{e:\Upsilon_{e:ce}(\Box)}(tid_A(\Pi^D_{\mathcal{A}(e_1)}(e_1)\times e_2))))))))$$
$$\overset{(1)}{=} \Pi_{res}(\chi_{res:\mathcal{C}(\text{elem},s1,(b,\mathcal{C}(\text{elem},s2,ct)))}(e_1\Gamma_{ct;\mathcal{A}(e_1)=A_1';count\circ\Pi_c}(\Pi_{A_1':\mathcal{A}(e_1)}$$
$$(\Pi^{tid_A}_{\mathcal{A}(e_1)\cup\mathcal{A}(e_2)}(\Pi^{tid_B}_{\mathcal{A}(e_1)\cup\mathcal{A}(e_2)}(\sigma_{e=ea}$$
$$(\Upsilon_{ea:\Upsilon_{ea:ba}(\Box)}(tid_B(\Upsilon_{e:\Upsilon_{e:ce}(\Box)}(tid_A(\Pi^D_{\mathcal{A}(e_1)}(e_1)\times e_2)))))))))))$$
$$\overset{(32)}{=} \Pi_{res}(\chi_{res:\mathcal{C}(\text{elem},s1,(b,\mathcal{C}(\text{elem},s2,ct)))}(e_1\Gamma_{ct;\mathcal{A}(e_1)=A_1';count\circ\Pi_c}(\Pi_{A_1':\mathcal{A}(e_1)}$$
$$(\Pi^{tid_A}_{\mathcal{A}(e_1)\cup\mathcal{A}(e_2)}(\Pi^{tid_B}_{\mathcal{A}(e_1)\cup\mathcal{A}(e_2)}(\sigma_{e=ea}$$
$$(\Upsilon_{ea:ba}(tid_B(\Upsilon_{e:ce}(tid_A(\Pi^D_{\mathcal{A}(e_1)}(e_1)\times e_2)))))))))))$$

In a last step, we want to turn the cross-product into a join operator. Before being able to do so, we have to eliminate one of the tid operators and push the other into the cross-product. After having assigned the tid A, we unnest and then assign the tid B. This guarantees that, for every value of B, we have the

same value for A, so $B \rightarrow A$. Therefore, we do not need the duplicate elimination based on attribute B anymore (and can get rid of the operation to assign it):

$$\stackrel{(34)}{=} \Pi_{res}(\chi_{res:\mathcal{C}(\text{elem},s1,(b,\mathcal{C}(\text{elem},s2,ct)))}(e_1\Gamma_{ct;\mathcal{A}(e_1)=A'_1;count\circ\Pi_c}(\Pi_{A'_1:\mathcal{A}(e_1)}(\Pi^{tid_A}_{\mathcal{A}(e_1)\cup\mathcal{A}(e_2)}\sigma_{e=ea}((\Upsilon_{ea:ba}(\Upsilon_{e:ce}(tid_A(\Pi^D_{\mathcal{A}(e_1)}(e_1)\times e_2)))))))))$$

$$\stackrel{(33)}{=} \Pi_{res}(\chi_{res:\mathcal{C}(\text{elem},s1,(b,\mathcal{C}(\text{elem},s2,ct)))}(e_1\Gamma_{ct;\mathcal{A}(e_1)=A'_1;count\circ\Pi_c}(\Pi_{A'_1:\mathcal{A}(e_1)}(\Pi^{tid_{A_1},tid_{A_2}}_{\mathcal{A}(e_1)\cup\mathcal{A}(e_2)}(\sigma_{e=ea}(\Upsilon_{ea:ba}(\Upsilon_{e:ce}(tid_{A_1}(\Pi^D_{\mathcal{A}(e_1)}(e_1))\times tid_{A_2}(e_2)))))))))$$

$$= \Pi_{res}(\chi_{res:\mathcal{C}(\text{elem},s1,(b,\mathcal{C}(\text{elem},s2,ct)))}(e_1\Gamma_{ct;\mathcal{A}(e_1)=A'_1;count\circ\Pi_c}(\Pi_{A'_1:\mathcal{A}(e_1)}(\Pi^{tid_{A_1},tid_{A_2}}_{\mathcal{A}(e_1)\cup\mathcal{A}(e_2)}(\Upsilon_{ea:ba}(tid_{A_1}(\Pi^D_{\mathcal{A}(e_1)}(e_1)))\bowtie_{e=ea}\Upsilon_{e:ce}(tid_{A_2}(e_2))))))))$$

8.5.2.2 *Outer Join.* Instead of applying Eqv. 25 in the second rewrite of the first step above, we could use Eqv. 26 based on the outer join operator. After doing so, we can rewrite the existentially quantified subexpression as shown above:

$$\stackrel{(26)}{=} \Pi_{res}(\chi_{res:\mathcal{C}(\text{elem},s1,(b,\mathcal{C}(\text{elem},s2,ct)))}(e_1⟕^{ct:0}_{\mathcal{A}(e_1)=A'_1}(\Pi_{A'_1:\mathcal{A}(e_1)}(\Gamma_{ct;=\mathcal{A}(e_1);count\circ\Pi_c}(\sigma_{\exists et\in\Upsilon_{e:ce}(\Box):\exists eat\in\Upsilon_{ea:ba}(\Box):e=ea}(\Pi^D_{\mathcal{A}(e_1)}(e_1)\times e_2))))))$$

$$= \Pi_{res}(\chi_{res:\mathcal{C}(\text{elem},s1,(b,\mathcal{C}(\text{elem},s2,ct)))}(e_1⟕^{ct:0}_{\mathcal{A}(e_1)=A'_1}(\Pi_{A'_1:\mathcal{A}(e_1)}(\Gamma_{ct;=\mathcal{A}(e_1);count\circ\Pi_c}(\Pi^{tid_{A_1},tid_{A_2}}_{\mathcal{A}(e_1)\cup\mathcal{A}(e_2)}(\Upsilon_{ea:ba}(tid_{A_1}(\Pi^D_{\mathcal{A}(e_1)}(e_1)))\bowtie_{e=ea}\Upsilon_{e:ce}(tid_{A_2}(e_2))))))))$$

8.5.2.3 *Evaluation.* The following table summarizes the results for the running times of the different versions of the query. This query does not seem to be favorable to unnesting. However, we can exploit the fact that the aggregate function *count* is insensitive to order. Hence, we can employ efficient implementations for the equijoin and the unary grouping operator. This results in substantially more efficient plans with notable advantages for the plan using binary grouping.

Size	100	1000	10,000
Nested	0.84 s	67.96 s	∞
Binary grouping	0.14 s	0.84 s	9.57 s
Outer join	0.14 s	1.04 s	35.99 s

9. CONCLUSION AND OUTLOOK

Performance still plays a very important role in today's DBMS, and native XML DBMSs are no exception. While XQuery—the query language that has been accepted as the standard for querying XML—possesses powerful constructs, the evaluation of queries is still suboptimal. One reason for this is that query optimizers for XQuery are still quite rudimentary compared to, for example, state-of-the-art optimizers for SQL. This is mainly due to the order-preserving nature of XQuery, which prohibits the straightforward application of known techniques for query optimization. Despite the restriction of having to preserve order, it is still possible to achieve tremendous performance gains by rewriting queries. In this article, we emphasize the importance of unnesting nested queries for

optimizing XQuery. Our proposed framework constitutes an important step in building an optimizer for an XML database system.

XQuery optimizers still suffer from the lack of an appropriate cost model. At the moment, we rely on basic heuristics for optimizing queries (which may or may not improve the evaluation time of a query). Our ultimate goal is a cost-based optimizer working on the algebraic level of a query. Consequently, we plan to develop a cost model for the operators contained in NAL, our algebra. Another important building block is the optimization of XPath expressions. Recently, we have shown that this can also be done based on a very similar algebraic approach. This means that our work on XPath can be seamlessly integrated into our framework for optimizing XQuery.

10. ELECTRONIC APPENDIX

Let us briefly summarize the content of the Electronic Appendix of this article, which is available in the ACM Digital Library. It contains details we could not cover here. In Appendix A, we discuss the applicability of several well-known algebraic equivalences in the ordered context and give a detailed explanation of support rewrites for quantified queries. Appendix B covers the result of normalization for several example queries. In Appendix C, we present additional example queries and show how to apply several unnesting equivalences or support rewrites we did not discuss in the main article. The setup of our experiments is explained in Appendix D. In addition, the proofs to all unnesting equivalences presented in this article can be found in Appendix E.

ACKNOWLEDGMENTS

We thank Niki Trigoni and Simone Seeger for their comments on the manuscript. We also thank the anonymous reviewers for their very helpful suggestions for improving this article.

REFERENCES

ASTRAHAN, M. M. AND CHAMBERLIN, D. D. 1975. Implementation of a structured English query language. *Commun. ACM 18*, 10, 580–588.

BEERI, C. AND TZABAN, Y. 1999. SAL: An algebra for semistructured data and XML. In *Informal Proceedings of the International Workshop on the Web and Databases* (WebDB '99), Philadelphia, PA. 37–42.

BEYER, K., CHAMBERLIN, D., COLBY, L., ÖZCAN, F., PIRAHESH, H., AND XU, Y. 2005. Extending XQuery for analytics. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*. ACM Press, New York, NY, 503–514.

BEYER, K. S., COCHRANE, R., COLBY, L. S., OZCAN, F., AND PIRAHESH, H. 2004. XQuery for analytics: Challenges and requirements. In *Informal Proceedings of the <XIME-P/>*. Paris, France, 3–8.

BHARGAVA, G., GOEL, P., AND IYER, B. 1995. Hypergraph based reorderings of outer join queries with complex predicates. In *SIGMOD '95: Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*. ACM Press, New York, NY, 304–315.

BORKAR, V. AND CAREY, M. 2004. Extending XQuery for grouping, duplicate elimination, and outer joins. In *Proceedings of XML 2004*.

BRANTNER, M., KANNE, C.-C., HELMER, S., AND MOERKOTTE, G. 2005. Full-fledged algebraic XPath processing in Natix. In *Proceedings of the 21st International Conference on Data Engineering* (ICDE '05, Tokyo, Japan). IEEE Computer Society Press, Los Alamitos, CA, 705–716.

BRY, F. 1989. Towards an efficient evaluation of general queries: quantifier and disjunction processing revisited. In *SIGMOD '89: Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*. ACM Press, New York, NY, 193–204.

CHATZIANTONIOU, D., AKINDE, M., JOHNSON, T., AND KIM, S. 2001. The MD-Join: An Operator for Complex OLAP. In *ICDE '01: Proceedings of the 17th International Conference on Data Engineering*. IEEE Computer Society Press, Los Alamitos, CA, 524–533.

CLAUSSEN, J., KEMPER, A., AND KOSSMANN, D. 1998. Order-preserving hash joins: Sorting (almost) for free. Technical rep. MIP-9810. University of Passau, Passau, Germany.

CLAUSSEN, J., KEMPER, A., MOERKOTTE, G., AND PEITHNER, K. 1997. Optimizing queries with universal quantification in object-oriented and object-relational databases. In *Proceedings of the International Conference on Very Large Data Bases* (VLDB). Morgan Kaufmann, San Francisco, CA, 286–295.

CLUET, S. AND DELOBEL, C. 1992. A general framework for the optimization of object-oriented queries. In *SIGMOD '92: Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*. ACM Press, New York, NY, 39–48.

CLUET, S. AND MOERKOTTE, G. 1994. Nested queries in object bases. In *DBLP-4: Proceedings of the Fourth International Workshop on Database Programming Languages—Object Models and Languages*. Springer-Verlag, London, U.K, 226–242.

CLUET, S. AND MOERKOTTE, G. 1995. Classification and optimization of nested queries in object bases. Technical rep. 95-6. RWTH Aachen, Aachen, Germany.

CLUET, S. AND MOERKOTTE, G. 1996. Efficient evaluation of aggregates on bulk types. In *DBLP-5: Proceedings of the Fifth International Workshop on Database Programming Languages*. Springer-Verlag, London, U.K., 8.

DAYAL, U. 1987. Of nests and trees: A unified approach to processing queries that contain nested subqueries, aggregates, and quantifiers. In *Proceedings of the International Conference on Very Large Data Bases* (VLDB). Morgan Kaufmann, San Francisco, CA, 197–208.

DEUTSCH, A., PAPAKONSTANTINOU, Y., AND XU, Y. 2004. The NEXT logical framework for XQuery. In *Proceedings of the International Conference on Very Large Data Bases* (VLDB). Morgan Kaufmann, San Francisco, CA 94022, 168–179.

DRAPER, D., FRANKHAUSER, P., FERNANDEZ, M., MALHOTRA, A., ROSE, K., RYS, M., SIMEON, J., AND WADLER, P. 2005. XQuery 1.0 and XPath 2.0 formal semantics. W3C Candidate Recommendation. Nov. Go online to `www.w3c.org`.

FEGARAS, L. 1998. Query unnesting in object-oriented databases. In *SIGMOD '98: Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*. ACM Press, New York, NY, 49–60.

FEGARAS, L., LEVINE, D., BOSE, S., AND CHALUVADI, V. 2002. Query processing of streamed XML data. In *CIKM '02: Proceedings of the Eleventh International Conference on Information and Knowledge Management*. ACM Press, New York, NY, 126–133.

FEGARAS, L. AND MAIER, D. 2000. Optimizing object queries using an effective calculus. *ACM Trans. Database Syst. 25*, 4, 457–516.

FIEBIG, T. AND MOERKOTTE, G. 2001. Algebraic XML construction and its optimization in Natix. *WWW J. 4*, 3, 167–187.

FUSHIMI, S., KITSUREGAWA, M., AND TANAKA, H. 1986. An overview of the systems software of a parallel relational database machine: GRACE. In *Proceedings of the International Conference on Very Large Data Bases* (VLDB). Morgan Kaufmann, San Francisco, CA, 209–219.

GALINDO-LEGARIA, C. AND JOSHI, M. 2001. Orthogonal optimization of subqueries and aggregation. In *SIGMOD '01: Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*. ACM Press, New York, NY, 571–581.

GALINDO-LEGARIA, C. AND ROSENTHAL, A. 1997. Outerjoin simplification and reordering for query optimization. *ACM Trans. Database Syst. 22*, 1 (Mar.), 43–73.

GANSKI, R. A. AND WONG, H. K. T. 1987. Optimization of nested SQL queries revisited. In *SIGMOD '87: Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*. ACM Press, New York, NY, 23–33.

GOTTLOB, G., KOCH, C., AND PICHLER, R. 2002. Efficient algorithms for processing XPath queries. In *Proceedings of the International Conference on Very Large Data Bases* (VLDB). VLDB Endowment, Hong Kong, China, 95–106.

GOTTLOB, G., KOCH, C., AND PICHLER, R. 2003. XPath query evaluation: Improving time and space efficiency. In *Proceedings of the IEEE Conference on Data Engineering*. IEEE Computer Society Press, Los Alamitos, CA, 379–390.

GRAEFE, G. 1993. Query evaluation techniques for large databases. *ACM Comput. Surv. 25*, 2 (Jun.), 73–170.

GRAEFE, G. 2003. Executing nested queries. In *Proceedings of BTW 2003*. GI, Leipzig, Germany, 58–77.

GRUST, T., SAKR, S., AND TEUBNER, J. 2004. XQuery on SQL hosts. In *Proceedings of the International Conference on Very Large Data Bases* (VLDB). 252–263.

GURAVANNAVAR, R., RAMANUJAM, H. S., AND SUDARSHAN, S. 2005. Optimizing nested queries with parameter sort orders. In *Proceedings of the International Conference on Very Large Data Bases* (VLDB). VLDB Endowment, Trondheim, Norway, 481–492.

JARKE, M. AND KOCH, J. 1984. Query optimization in database systems. *ACM Comput. Surv. 16*, 2 (Jun.), 111–152.

KIESSLING, W. 1984. SQL-like and Quel-like correlation queries with aggregates revisited. ERL/UCB Memo 84/75, University of California, Berkeley, Berkeley, CA.

KIM, W. 1982. On optimizing an SQL-like nested query. *ACM Trans. Database Syst. 7*, 3 (Sept.), 443–469.

KLUG, A. 1982. Equivalence of relational algebra and relational calculus query languages having aggregate functions. *J. ACM 29*, 3, 699–717.

LIU, Z. H., KRISHNAPRASAD, M., AND ARORA, V. 2005. Native XQuery processing in Oracle XMLDB. In *Proceedings of the ACM SIGMOD Conference on Management of Data*. ACM Press, New York, NY, 828–833.

MAY, N., HELMER, S., KANNE, C.-C., AND MOERKOTTE, G. 2004a. XQuery processing in Natix with an emphasis on join ordering. In *Informal Proceedings of the First International Workshop on XQuery Implementation, Experience and Perspectives* (*XIME-P '04*, Paris, France). 49–54.

MAY, N., HELMER, S., AND MOERKOTTE, G. 2003. Nested queries and quantifiers in an ordered context. Tech. rep. TR-03-002. Department for Mathematics and Computer Science, University of Mannheim, Mannheim, Germany.

MAY, N., HELMER, S., AND MOERKOTTE, G. 2004b. Nested queries and quantifiers in an ordered context. In *ICDE '04: Proceedings of the 20th International Conference on Data Engineering*. IEEE Computer Society Press, Los Alamitos, CA, 239.

MAY, N. AND MOERKOTTE, G. 2005. Main memory implementations for binary grouping. In *Proceedings of the 3rd International XML Database Symposium* (XSym 2005), Trondheim, Norway. Springer-Verlag, London, U.K., 162–176.

MUMICK, I. S., FINKELSTEIN, S. J., PIRAHESH, H., AND RAMAKRISHNAN, R. 1990. Magic is relevant. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*. ACM Press, New York, NY, 247–258.

MURALIKRISHNA, M. 1989. Optimization and dataflow algorithms for nested tree queries. In *Proceedings of the International Conference on Very Large Data Bases* (VLDB). Morgan Kaufmann, San Francisco, CA, 77–85.

MURALIKRISHNA, M. 1992. Improved unnesting algorithms for join aggregate SQL queries. In *Proceedings of the International Conference on Very Large Data Bases* (VLDB). Morgan Kaufmann, San Francisco, CA, 91–102.

NAKANO, R. 1990. Translation with optimization from relational calculus to relational algebra having aggregate functions. *ACM Trans. Database Syst. 15*, 4, 571–581.

PAL, S., CSERI, I., SEELIGER, O., RYS, M., SCHALLER, G., YU, W., TOMIC, D., BARAS, A., BERG, B., CHURIN, D., AND KOGAN, E. 2005. XQuery implementation in a relational database system. In *Proceedings of the International Conference on Very Large Data Bases* (VLDB). VLDB Endowment, Trondheim, Norway, 1175–1186.

PAPARIZOS, S., AL-KHALIFA, S., JAGADISH, H. V., LAKSHMANAN, L. V. S., NIERMAN, A., SRIVASTAVA, D., AND WU, Y. 2002. Grouping in XML. In *EDBT '02: Proceedings of the Worshops XMLDM, MDDE, and YRWS on XML-Based Data Management and Multimedia Engineering-Revised Papers*. Springer-Verlag, London, U.K., 128–147.

PIRAHESH, H., HELLERSTEIN, J. M., AND HASAN, W. 1992. Extensible/rule based query rewrite optimization in Starburst. In *SIGMOD '92: Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*. ACM Press, New York, NY, 39–48.

ROSENTHAL, A. AND GALINDO-LEGARIA, C. 1990. Query graphs, implementing trees, and freely-reorderable outerjoins. In *SIGMOD '90: Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*. ACM Press, New York, NY, 291–299.

SESHADRI, P., HELLERSTEIN, J. M., PIRAHESH, H., LEUNG, T. Y. C., RAMAKIRSHNAN, R., SRIVASTAVA, D., STUCKEY, P. J., AND SUDARSHAN, S. 1996a. Cost-based optimization for magic: Algebra and implementation. In *SIGMOD '96: Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*. ACM Press, New York, NY, 435–446.

SESHADRI, P., PIRAHESH, H., AND LEUNG, T. Y. C. 1996b. Complex query decorrelation. In *ICDE'96: Proceedings of the Twelfth International Conference on Data Engineering*. IEEE Computer Society Press, Los Alamitos, CA, 450–458.

STEENHAGEN, H. J. 1995. Optimization of object query languages. Ph.D. dissertation. University of Twente, Enschede, The Netherlands.

STEENHAGEN, H. J., APERS, P. M. G., BLANKEN, H. M., AND DE BY, R. A. 1994. From nested-loop to join queries in OODB. In *Proceedings of the International Conference on Very Large Data Bases* (VLDB). Morgan Kaufmann, San Francisco, CA, 618–629.

YAN, W. P. AND LARSON, P.-Å. 1994. Performing group-by before join. In *Proceedings of the Tenth International Conference on Data Engineering*. IEEE Computer Society Press, Los Alamitos, CA, 89–100.

Received November 2005; revised April 2006; accepted May 2006

Copyright of ACM Transactions on Database Systems is the property of Association for Computing Machinery and its content may not be copied or emailed to multiple sites or posted to a listserv without the copyright holder's express written permission. However, users may print, download, or email articles for individual use.