Nested Queries and Quantifiers in an Ordered Context

Norman May Sven Helmer Guido Moerkotte
Universität Mannheim
Mannheim, Germany
norman|*helmer*|*moerkotte*@informatik.uni-mannheim.de

Abstract

We present algebraic equivalences that allow to unnest nested algebraic expressions for order-preserving algebraic operators. We illustrate how these equivalences can be applied successfully to unnest nested queries given in the XQuery language. Measurements illustrate the performance gains possible by unnesting.

1. Introduction

With his seminal paper Kim opened the area of unnesting nested queries in the relational context [22]. Very quickly, it became clear that enormous performance gains are possible by avoiding nested-loop evaluation of nested query blocks (as proposed in [1]) by unnesting them. Almost as quickly, the subtleties of unnesting became apparent. The first bugs in the original approach were detected — among them the famous count bug [23]. Retrospectively, we can summarize the problem areas as follows:

- Special cases like empty results lead easily to bugs like the count bug [23]. They have been corrected by different approaches [9, 15, 21, 23, 26].
- If the nested query contains grouping, special rules are needed to pull up grouping operators [4].
- Special care has to be taken for a correct duplicate treatment [18, 28, 30].

The main reason for the problems was that SQL lacked expressiveness, and unnesting took place at the query language level. The most important construct needed for correctly unnesting queries are outer joins [9, 15, 21]. After their introduction into SQL and their usage for unnesting, reordering of outer joins became an important topic [3, 14, 29]. A unifying framework for different unnesting strategies for SQL can be found in [26].

With the advent of object-oriented databases and their query languages, unnesting once again attracted some attention [7, 8, 11, 31, 32, 33]. In contrast to the relational unnesting strategies, which performed unnesting at the (extended) SQL source level, researchers from the object-oriented area preferred to describe unnesting techniques at the algebraic level. They used algebras that allow nesting: algebraic expressions can be found in subscripts of algebraic operators. For example, a predicate of a selection or join operator could again contain algebraic operators. These algebras allow a straightforward representation of nested queries, and unnesting can then take place at the algebraic level. The main advantage of this approach is that unnesting rewrites can be described by algebraic equivalences for which rigorous correctness proofs could be delivered. Further, these equivalence-based unnesting techniques remain valid independently of the query language as long as queries remain expressible in the underlying algebra. For example, they can also successfully be applied to SQL. However, the algebras used for unnesting do not maintain order. Hence, they are only applicable to queries that do not have to preserve order.

XQuery[1] is a query language that allows the user to specify whether to retain the order of its input documents or not. If the `unordered` function is applied to a query, the order of its result is independent of the input order, and the query processor can generate the output in any order. If this is the case, the XQuery expression can be translated into an unordered algebra and the unnesting techniques discovered in the object-oriented context remain applicable. Apart from the `unordered` function, the query processor can determine other cases where the output order is irrelevant. Examples include aggregate functions, the `distinct-values` function, and quantifiers. However, if the result's order is relevant, the unnesting techniques from the object-oriented context cannot be applied.

Quantification is a core feature of XQuery in which the keywords `some` and `every` are used to express existential and universal quantification. Optimization for queries containing quantification has been investigated in the relational and object-oriented context — see [5] for related work.

---

[1] http://www.w3.org/XML/Query

The area of unnesting nested queries was reopened for XQuery by Paparizos et al. [27]. Their approach describes the introduction of a grouping operator for a nested query. However, the verbal description of their transformation is not rigorous and indeed not complete: one important restriction that guarantees correctness is missing. We will come back to this point when discussing our counterpart (Eqv. 7 in Sec. 4) of their technique in Section 5.1. To the best of our knowledge, no other paper discusses unnesting in the ordered context. Based on their previous work, Fegaras et al. [10] focus on unnesting queries operating on streams. So, in principle it should be possible to use their approach for unnesting in an ordered context. However, it is unclear to which extent order preservation is considered (e.g. on the algebraic level hash joins are used, which are not necessarily known for their order-preserving properties).

Within this paper we introduce several different unnesting strategies and discuss their application to different query types. All these techniques are described by means of algebraic equivalences which we proved to be correct. In particular they are order-preserving. They can be applied to algebraic expressions resulting from queries

- with or without aggregate functions,
- with different comparison operators in their correlation predicate, and
- with existential and universal quantifiers.

Providing a complete framework for optimizing XQuery queries is still future work and beyond the scope of this paper. Nevertheless, we provide performance figures for every query execution plan that is presented, demonstrating the significant speed-up gained by unnesting.

Overview of Our Approach Except for quantifiers, our unnesting approach consists of the following three steps:

1. Normalization introduces additional **let** clauses for nested queries (see Sec. 3).
2. **let** clauses are translated into map operations (χ) (see Sec. 2) with nested algebraic expressions representing the nested query (see Sec. 3).
3. Unnesting equivalences (see Fig. 4) pull up expressions nested in a χ operator.

The remainder of the paper is organized as follows. Section 2 briefly motivates and defines our algebra. Section 3 shows how to normalize and translate nested queries into our algebra. Section 4 is the core of the paper, containing the equivalences used for unnesting nested queries. The way they are applied is demonstrated in Section 5. There, we apply different equivalences to queries found in the XQuery use-case document[2]. For every query execution plan, we provide performance figures. Section 6 concludes the paper.

2. Notation and Algebra

Our algebra (NAL) extends the SAL-Algebra [2] developed by Beeri and Tzaban. SAL is the order-preserving counterpart of the algebra used in [7, 8] extended to handle semistructured data. Order-preserving algebras are not only interesting in the context of XQuery, but can also be used in fields dealing with time series, like finance, molecular biology, or network management [24].

SAL and NAL work on sequences of sets of variable bindings, i.e., sequences of unordered tuples where every attribute corresponds to a variable. We allow nested tuples, i.e. the value of an attribute may be a sequence of tuples. Single tuples are constructed by using the standard $[\cdot]$ brackets. The concatenation of tuples and functions is denoted by $\circ$. The set of attributes defined for an expression e is defined as $\mathcal{A}(e)$. The set of free variables of an expression e is defined as $\mathcal{F}(e)$.

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

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

By *id* we denote the identity function. In order to avoid special cases during the translation of XQuery into the algebra, we use the special algebraic operator ($\Box$) that returns a singleton sequence consisting of the empty tuple, i.e., a tuple with no attributes.

We will only define order-preserving algebraic operators. For the unordered counterparts see [8]. Typically, when translating a more complex XQuery into our algebra, a mixture of order-preserving and non-order-preserving operators will occur. In order to keep the paper readable, we only employ the order-preserving operators and use the same notation for them that has been used in [7, 8] and SAL [2].

Our algebra will allow nesting of algebraic expressions. For example, within a selection predicate of a select operator we allow the occurrence of further nested algebraic

---

[2] http://www.w3.org/TR/xmlquery-use-cases

expressions. Hence, a join within a selection predicate is possible. This simplifies the translation procedure of nested XQuery expressions into the algebra. However, nested algebraic expressions force a nested loop evaluation strategy. Thus, the goal of the paper will be to remove nested algebraic expressions. As a result, we perform unnesting of nested queries not at the source level but at the algebraic level. This approach is more versatile and less error-prone.

We define the algebraic operators recursively on their input sequences. 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.

The order-preserving selection operator is defined as

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

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. Sometimes we just want to eliminate some attributes. If we want to eliminate the set of attributes A, we denote this by $\Pi_{\overline{A}}$. We use Π also for renaming attributes. Then we write $\Pi_{A':A}$. The attributes in A are renamed to those in A'. Attributes other than those 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 1.

S_1

A_1
1
2
3

S_2

A_2	B
1	2
1	3
2	4
2	5

$\chi_{a:\sigma_{A_1=A_2}(S_2)}(S_1) =$

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

Figure 1. Example for Map Operator

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

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

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 anti-join as

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

The left outer join, which will play an essential role in unnesting, is defined as $e_1 ⟕_p^{g:e} e_2 :=$

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

where $g \in \mathcal{A}(e_2)$. Our definition deviates slightly from the standard left outer join operator, as we want to use it in conjunction with grouping and (aggregate) functions. Consider the sequences S_1 and S_2 in Figure 2. If we want to join S_1 (via left outer join) to S_2^{count} that is grouped by A_2 with counted values for B, we must be able to handle empty groups (for $A_1 = 3$). e defines the value given to attribute g for values in e_1 that do not find a join partner in e_2 (in this case 0).

For the rest of the paper let $\theta \in \{=, \leq, \geq, <, >, \neq\}$ be a comparison operator on atomic values. The grouping operator which produces a sequence-valued new attribute containing "the group" is defined by using a binary grouping operator.

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

where the binary grouping operator (sometimes called nest-join [31]) is defined as

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

Here, $G(x) := f(\sigma_{x|_{A_1}\theta A_2}(e_2))$ and function f assigns a meaningful value to empty groups. See also Figure 2 for an example. The unary grouping operator processes a single sequence and obviously groups only on those values that are present. The binary grouping operator works on two sequences and uses the left hand one to determine the groups. This will become important for the correctness of the unnesting procedure.

S_1

A_1
1
2
3

S_2

A_2	B
1	2
1	3
2	4
2	5

$\Gamma_{g;=A_2;count}(S_2) = S_2^{count}$

A_2	g
1	2
2	2

$\Gamma_{g;=A_2;id}(S_2) = S_2^g$

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

$S_1 \; \Gamma_{g;A_1=A_2;id}(S_2) = S_{1,2}^g$

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

Figure 2. Examples for Unary and Binary Γ

Given a tuple with a sequence-valued attribute, we can unnest it by using the unnest operator defined as

$$\mu_g(e) := (\alpha(e)|_{\overline{\{g\}}} \times \alpha(e).g) \oplus \mu_g(\tau(e))$$

where $e.g$ retrieves the sequence of tuples of attribute g. In case that g is empty, it returns the tuple $\perp_{\mathcal{A}(e.g)}$. (In our example in Figure 2, $\mu_g(S_2^g) = S_2$.)

We define the unnest map operator as follows:

$$\Upsilon_{a:e_2}(e_1) := \mu_g(\chi_{g:e_2[a]}(e_1))$$

This operator is mainly used for evaluating XPath expressions. Since this is a very complex issue [16, 17, 19], we do not delve into optimizing XPath evaluation but instead take an XPath expression occurring in a query as it is and use it in the place of e_2. Optimized translation of XPath is orthogonal to our unnesting approach and not covered in this paper. The interested reader is referred to [19].

For result construction, we employ a simplified operator Ξ that combines a pair of Groupify-GroupApply operators [13]. It executes a semicolon-separated list of commands and, as a side effect, constructs the query result. The Ξ operator occurs in two different forms. In its simple form, apart from side-effects, Ξ is the identity function, i.e., it returns its input sequence. For simplicity, we assume that the result is constructed as a string on some output stream. Then the simplest command is a string copied to the output stream. If the command is a variable, its string value is copied to the output stream. For more complex expressions, the procedure is similar. If e is an expression that evaluates to a sequence of tuples containing a string-valued attribute a that is successively bound to author names from some bibliography document, $\Xi_{\text{"}<author>\text{"};a;\text{"}</author>\text{"}}(e)$ embeds every author name into an `author` element.

In its group-detecting form, ${}^{s_1}\Xi^{s_3}_{A;s_2}$ uses a list of attributes (A) and three sequences of commands. We define

$${}^{s_1}\Xi^{s_3}_{A;s_2}(e) := \Xi_{(s_1;\Xi_{s_2};s_3)}(\Gamma_{g;=A;id}(e))$$

where Γ has to use an order-preserving duplicate elimination operation in its definition and Ξ_{s_2} processes the sequence-valued attribute created by Γ. Like grouping in general, Ξ can be implemented very efficiently on condition that a group spans consecutive tuples in the input sequence and group boundaries are detected by a change of any of the attribute values in A. Then, for every group the first sequence of statements (s_1) is executed using the first tuple of a group, the second one (s_2) is executed for every tuple within a group, and the third one (s_3) is executed using the last tuple of a group. This condition can be met by a stable(!) sort on A. Introducing the complex Ξ saves a grouping operation that would have to construct a sequence-valued attribute.

Let us illustrate ${}^{s_1}\Xi^{s_3}_{A;s_2}(e)$ by a simple example. Assume that the expression e produces the following sequence of four tuples:

```
[a: "author1", t: "title1"]
[a: "author1", t: "title2"]
[a: "author2", t: "title1"]
[a: "author2", t: "title3"]
```

Then ${}^{s_1}\Xi^{s_3}_{a;s_2}(e)$ with

```
s1 = "<author>";"<name>";a;"</name>"
s2 = "<title>";t;"</title>"
s3 = "</author>"}
```

produces

```
<author>
  <name>author1</name>
    <title>title1</title>
    <title>title2</title>
</author>
<author>
  <name>author2</name>
    <title>title1</title>
    <title>title3</title>
</author>
```

To acquaint the reader with ordered sequences, we state some familiar equivalences that still hold.

$$\begin{array}{rcl}
\sigma_{p_1}(\sigma_{p_2}(e)) & = & \sigma_{p_2}(\sigma_{p_1}(e)) \\
\sigma_p(e_1 \times e_2) & = & \sigma_p(e_1) \times e_2 \\
\sigma_p(e_1 \times e_2) & = & e_1 \times \sigma_p(e_2) \\
\sigma_{p_1}(e_1 \bowtie_{p_2} e_2) & = & \sigma_{p_1}(e_1) \bowtie_{p_2} e_2 \\
\sigma_{p_1}(e_1 \bowtie_{p_2} e_2) & = & e_1 \bowtie_{p_2} \sigma_{p_1}(e_2) \\
\sigma_{p_1}(e_1 \ltimes_{p_2} e_2) & = & \sigma_{p_1}(e_1) \ltimes_{p_2} e_2 \\
\sigma_{p_1}(e_1 \bowtie^{g:e}_{p_2} e_2) & = & \sigma_{p_1}(e_1) \bowtie^{g:e}_{p_2} e_2 \\
e_1 \times (e_2 \times e_3) & = & (e_1 \times e_2) \times e_3 \\
e_1 \bowtie_{p_1} (e_2 \bowtie_{p_2} e_3) & = & (e_1 \bowtie_{p_1} e_2) \bowtie_{p_2} e_3
\end{array}$$

Of course, in the above equivalences the usual restrictions hold. For example, if we want to push a selection predicate into the left part of a join, it may not reference attributes of the join's right argument. In other words, $\mathcal{F}(p_1) \cap \mathcal{A}(e_2) = \emptyset$ is required. Please note that cross product and join are still associative in the ordered context. However, neither of them is commutative.

For implementation issues concerning the operators of NAL, please consult our technical report [25].

3. 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. Since Section 5 will give many examples of both steps, we do not give any examples in this section.

Prior to translation into the algebra, we use a normalization step that introduces new variables. It 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 [6], we will not detail it. Basically, we break down a query into subexpressions and consider which of them can be factorized. In this paper, we will not split up the query at all possible positions, but only when necessary in order to demonstrate the major points. The motivation for this step becomes apparent when considering that (1) a **let** clause will be translated into a χ operator and (2) most unnesting equivalences (see Fig. 4) use a χ operator as their starting point. Roughly, we apply the following steps:

1. We embed range expressions of quantifiers into new FLWR expressions.
2. We break up complex expressions and introduce new variables for subexpressions.
3. We factorize common subexpressions.
4. We move predicates from XPath expressions to the **where** clause whenever possible.

Note that all of these steps require some attention, since a careless application of this procedure may change the semantics of the query.

For a given query Q, $\mathcal{T}(Q)$ translates Q into the algebra. We specify the translation procedure by means of two recursive procedures $\mathcal{T}$ (see Figure 3). One of them is unary, the other binary. The binary $\mathcal{T}$ procedure is responsible for translating FLWR expressions into the algebra. We do not treat the **order by** clause since we concentrate on the ordered case in this paper. The first argument of the binary $\mathcal{T}$ procedure is the (remainder of) the query to be translated, and the second argument is the algebraic expression constructed so far. For non-FLWR expressions, we use the unary $\mathcal{T}$ operation. Both are mutually recursive, since a FLWR expression can occur within simple expressions and vice versa. The translation is rather straightforward, but two technical remarks are necessary. When translating the **let** clause, we have to introduce additional attributes/variables for the items in the results of the expressions e_i since XQuery expressions do not return sequences of tuples but sequences of items, and the data model of our algebra allows only nested sequences of tuples. Hence, we have to invent new attribute names. However, if the result of some e_i is a singleton, we do not need to do so and will not either.

Additionally, we have to use the function $\mathcal{C}$, which converts the **return** expression into a sequence of expressions. Every expression is either treated as a constant string that is printed by Ξ, or as an evaluable expression if it is escaped by $\{$ and $\}$. On the latter, $\mathcal{C}$ applies $\mathcal{T}$. This treatment of the **return** clause of XQuery is not really advanced, but since our focus is on unnesting nested queries, it suffices for demonstration purposes. The interested reader is referred to [13] for a more detailed description on how to treat result construction for XML query languages.

Besides result construction, another suboptimal spot of our translation is the treatment of path expressions. We are aware of the fact that efficient evaluation algorithms for path expressions exist [16, 17, 19]. But again, since this is orthogonal to the unnesting, we do not describe any optimizing translation procedure. Note the ease of our translation compared to the one described in [20], which also does not elaborate on efficient XPath evaluation.

4. Unnesting Equivalences

Figure 4 contains the equivalences that will allow us to unnest nested algebraic expressions. For readers unfamiliar with the general procedure of unnesting nested queries, we suggest skipping this section during the first reading. We advise to continue with the next section containing the example queries. After having worked through the examples, one can come back to this section to have a look at the rigorous definitions of the equivalences, which are crucial for a *correct* treatment of this subject. Too often, incorrect unnesting procedures have appeared. Thus, before commenting on the equivalences, let us give the conditions that ensure correctness. For the first three equivalences we must have $A_i \subseteq \mathcal{A}(e_i)$ and $\mathcal{F}(e_2) \cap \mathcal{A}(e_1) = \emptyset$. In Eqv. 6, $\mu_g^D(e)$ abbreviates $\Pi_{\overline{g'}}(\mu_{g'}(\chi_{g':\Pi^D(g)}(e)))$. For equivalences 6 and 7, we must have $a_2 \in \mathcal{A}(e_2)$ and $A_2 = \mathcal{A}(a_2)$. In these two equivalences, the function f may not depend on the values of the attributes a_2 and A_2. In other words, it must satisfy that for every sequence s $f(s) = f(\Pi_{\overline{a_2}}(s)) = f(\Pi_{\overline{A_2}}(s))$.

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

$$\mathcal{T}(Q,A) := \begin{cases} \mathcal{T}(\text{REST}, \Upsilon_{x_n:\mathcal{T}(e_n)}(\ldots(\Upsilon_{x_1:\mathcal{T}(e_1)}(A)))) & \text{if } Q \text{ = for } \$x_1 \text{ in } e_1, \ldots, \$x_n \text{ in } e_n \text{ REST} \\ \mathcal{T}(\text{REST}, \chi_{x_n:\mathcal{T}(e_n)[x'_n]}(\ldots(\chi_{x_1:\mathcal{T}(e_1)[x'_1]}(A)))) & \text{if } Q \text{ = let } \$x_1 \text{ := } e_1, \ldots, \$x_n \text{ := } e_n \text{ REST} \\ \mathcal{T}(\text{REST}, \sigma_{\mathcal{T}(p)}(A)) & \text{if } Q \text{ = where p} \\ \Xi_{\mathcal{C}(e)}(A) & \text{if } Q \text{ = return e} \\ A & \text{if } Q \text{ is empty string} \end{cases}$$

The unary $\mathcal{T}$ function for other expressions:

$$\mathcal{T}(Q) := \begin{cases} \exists x \in \mathcal{T}(D)\mathcal{T}(P) & \text{if } Q \text{ = some \$x in D satisfies P} \\ \forall x \in \mathcal{T}(D)\mathcal{T}(P) & \text{if } Q \text{ = every \$x in D satisfies P} \\ \Pi^D(\mathcal{T}(e)) & \text{if } Q \text{ = distinct-values(e)} \\ \mathcal{T}(Q,\Box) & \text{if } Q \text{ is a FLWR expression} \\ f(\mathcal{T}(e_1),\ldots,\mathcal{T}(e_n)) & \text{if } Q = f(e_1,\ldots,e_n) \\ Q & \text{if } Q \text{ is a variable or constant} \end{cases}$$

Figure 3. Translation of XQuery FLWR Expressions into the Algebra

As f will mostly be a projection, aggregate function, or a combination of both, this condition will easily be satisfied. The first five equivalences make use of a new attribute g with $g \not\in \mathcal{A}(e_1) \cup \mathcal{A}(e_2)$. We further assume the attributes occurring in e_1 and e_2 to be different: $A_1 \cap A_2 = \emptyset$. Please note that the conditions given in Eqv. 5 and 7 imply $A_1 = \mathcal{A}(e_1)$. For the last two equivalences, $x' \in \mathcal{A}(e_2)$ must hold. Further, p' results from p by replacing x by x'.

Why are the equivalences useful? Remember from the normalization process that a nested query becomes an expression in the **let** clause, and from the translation process that a **let** is translated into a χ operation. Hence, all unnesting equivalences will be applied from left to right. Whenever there are alternative applications, the most efficient plan should be chosen. Typically, it is the one applying the equivalences with the most restrictive conditions attached. In case the **where** clause contains a quantifier, the translation process results in an expression matching the left-hand side of one of the last two equivalences.

Let us look a little bit closer at the first three equivalences to clarify the matter (due to space constraints we cannot go into details for all equivalences, but it should be no problem to understand the other equivalences by playing through similar examples). Consider Eqv. 3: The left hand side of the equivalence is shown in Figure 1. The naive nested loop evaluation of the expression $\chi_{a:\sigma_{A_1=A_2}(S_2)}(S_1)$ results in three scans over S_2 – the number of items in S_1. The right hand side of the equivalence is depicted in Figure 2. A more efficient evaluation for $S^g_{1,2}$ is possible because S_2 needs to be scanned just once – independent from the number of items. Eqv. 5 is a special case of Eqv. 3 and would be applicable if S_1 were a projection of S_2 on A_2 (i.e., if S_1 just contained the first two tuples). In this case, we would not have to fear losing tuples for empty groups, so we could use the unary grouping operator.

As we are checking for the equality of the attributes A_1 and A_2 in the previous example, we could also apply Eqv. 4. In this case, we first group the tuples in S_2 on A_2 using a unary grouping operator, resulting in the sequence S^g_2 in Figure 2. In a second step, S_1 is joined to S^g_2 via a left outer join (in order to avoid losing empty groups). Here the function f is the identity function, supplying an empty sequence ($\langle\,\rangle$) in the case of an empty input (ϵ). Finally, the attribute A_2 is projected away, resulting in the same sequence as seen on the right hand side of Figure 1.

Related Work: For all equivalences except 6 and 7, counterparts for a traditional algebra on (unordered) sets appeared in the literature (see [8] and the related work discussion there). Equivalences 6 and 7 are new in both the ordered and the unordered context. An equivalent to Eqv. 7 in the ordered context appeared in [27], but without giving the important condition $e_1 = \Pi^D_{A_1:A_2}(\Pi_{A_2}(e_2))$. The counterparts of Eqvs. 8 and 9 appeared, for example, in [5, 9]. Nevertheless, for the ordered context we have to prove the correctness of both, the new equivalences and those with counterparts in the unordered context. The proofs of all these equivalences can be found in [25].

The next two equivalences do not unnest, but save scanning the same document twice, which leads to a faster execution. They are typically applied after unnesting.

$$\Pi^D(e_1) \ltimes_{A_1=A_2} (\sigma_p(e_2)) \quad = \quad \sigma_{c>0}(E) \qquad (1)$$

$$\Pi^D(e_1) \triangleright_{A_1=A_2} (\sigma_p(e_2)) \quad = \quad \sigma_{c=0}(E) \qquad (2)$$

with $E = \Pi_{A_1:A_2}(\Gamma_{c;=A_2;count\circ\sigma_p}(e_2))$. The equivalences hold if $A_i \subseteq \mathcal{A}(e_i)$, $\mathcal{F}(e_2) \cap \mathcal{A}(e_1) = \emptyset$, and $\Pi^D(e_1) = \Pi^D_{A_1:A_2}(\Pi_{A_2}(e_2))$.

$$\chi_{g:f(\sigma_{A_1 \theta A_2}(e_2))}(e_1) = e_1 \Gamma_{g;A_1\theta A_2;f} e_2 \tag{3}$$

$$\chi_{g:f(\sigma_{A_1 = A_2}(e_2))}(e_1) = \Pi_{\overline{A_2}}(e_1 \mathbin{⟕}^{g:f(\epsilon)}_{A_1=A_2} (\Gamma_{g;=A_2;f}(e_2))) \tag{4}$$

$$\chi_{g:f(\sigma_{A_1 \theta A_2}(e_2))}(e_1) = \Pi_{A_1:A_2}(\Gamma_{g;\theta A_2;f}(e_2)) \quad \text{if } e_1 = \Pi^D_{A_1:A_2}(\Pi_{A_2}(e_2)) \tag{5}$$

$$\chi_{g:f(\sigma_{A_1 \in a_2}(e_2))}(e_1) = \Pi_{\overline{A_2}}(e_1 \mathbin{⟕}^{g:f(\epsilon)}_{A_1=A_2} \Gamma_{g;=A_2;f}(\mu^D_{a_2}(e_2))) \tag{6}$$

$$\chi_{g:f(\sigma_{A_1 \in a_2}(e_2))}(e_1) = \Pi_{A_1:A_2}(\Gamma_{g;=A_2;f}(\mu^D_{a_2}(e_2))) \quad \text{if } e_1 = \Pi^D_{A_1:A_2}(\Pi_{A_2}(\mu_{a_2}(e_2))) \tag{7}$$

$$\sigma_{\exists x \in (\Pi_{x'}(\sigma_{A_1=A_2}(e_2)))p}(e_1) = e_1 \ltimes_{A_1=A_2 \wedge p'} e_2 \tag{8}$$

$$\sigma_{\forall x \in (\Pi_{x'}(\sigma_{A_1=A_2}(e_2)))p}(e_1) = e_1 \triangleright_{A_1=A_2 \wedge \neg p'} e_2 \tag{9}$$

Figure 4. Unnesting Equivalences

5. Example Applications

In this section we present example applications for the unnesting equivalences. We based the queries on those in the XQuery use case document and the DTDs therein. The DTD for the first four queries is given in Figure. 5. We rewrote the queries by renaming variables and simplifying them slightly, thereby retaining the essence of the query. The numbers in the subsection headings correspond to the query numbers therein. Due to space restrictions, we do not explain the matter in a highly detailed fashion. For readers trying to attain a deeper understanding, we suggest retracing the transformations by means of exemplary XML documents (e.g. taken from the XQuery use-case document[3]).

We verified the effectiveness of the unnesting techniques experimentally. The experiments were carried out on a simple PC running SuSE Linux 8.1 with a 2.4 Ghz Pentium using the Natix query evaluation engine [12] (which was compiled using g++ 3.2). The database cache was configured such that it could hold the queried documents. The XML files were generated by ToXgene using the DTDs in the XQuery use case document. We executed the various evaluation plans on different sizes of input documents. We note the number of elements contained in the input documents for each measurement.

```
<!ELEMENT bib  (book* )>
<!ELEMENT book  (title,  (author+ | editor+ ),
          publisher, price )>
<!ATTLIST book  year CDATA  #REQUIRED >
<!ELEMENT author  (last, first )>
<!ELEMENT editor  (last, first, affiliation )>
<!ELEMENT title  (#PCDATA )>
<!ELEMENT last  (#PCDATA )>
<!ELEMENT first  (#PCDATA )>
<!ELEMENT affiliation  (#PCDATA )>
<!ELEMENT publisher  (#PCDATA )>
<!ELEMENT price  (#PCDATA )>

<!ELEMENT reviews (entry*)>
<!ELEMENT entry   (title, price, review)>
<!ELEMENT price   (#PCDATA)>
<!ELEMENT review  (#PCDATA)>

<!ELEMENT prices (book*)>
<!ELEMENT book (title, source, price)>
<!ELEMENT source (#PCDATA)>
```

Figure 5. DTD for the first four queries

5.1. Query 1.1.9.4 (Grouping)

The first query restructures the input document by grouping books by authors (note that grouping in XQuery is done implicitly).

```
let $d1 := doc("bib.xml")
for $a1 in distinct-values($d1//author)
return
  <author>
        <name> { $a1 } </name>
        {
          let $d2 := doc("bib.xml")
          for $b2 in $d2/book[$a1 = author]
          return $b2/title
        }
  </author>
```

and its normalization yields

```
let $d1 := doc("bib.xml")
for $a1 in distinct-values($d1//author)
let $t1 := let $d2 := doc("bib.xml")
           for $b2 in $d2/book
           let $a2 := $b2/author,
               $t2 := $b2/title
           where $a1 = $a2
           return $t2
return
  <author>
    <name> { $a1 } </name>
```

---

[3] http://www.w3.org/TR/xmlquery-use-cases

```
    { $t1 }
  </author>
```

Normalization of the query first moves the nested FLWR expression outside the **return** clause into a new **let** clause. We prepare the moved **for** clause for the translation into an algebraic expression by introducing new variables. We further moved the predicate at the end of the path expression into the **where** clause to have it translated into a σ. (Note the σ in the subscripts of the χ on the left-hand side of the equivalences in Fig. 4.)

During translation, we have to take care of one important point. There exist different comparison operators in XQuery, and a simple '=' has existential semantics in case either side contains a sequence of expressions. In our case, $a1 is bound to a single value, and $a2 is bound to a sequence. Consequently, we have to translate $a1 = $a2 into $a1 \in a2$. From the DTD we know that every book contains only a single `title` element. Hence, we can save the introduction of an attribute $t2'$ and the invocation of a concatenation operation that is implicitly invoked in XQuery[4]. Hence, we can apply a simple projection on $t2$ to model the `return` clause of the inner query block. The translation then results in

$$\Xi_{s1;a1;s2;t1;s3}(\chi_{t1:\Pi_{t2}(\sigma_{a1 \in a2}(\hat{e}_2))}(\hat{e}_1))$$

where

$$\begin{aligned} \hat{e}_1 &:= \Upsilon_{a1:\Pi^D(d1//author)}(\chi_{d1:doc}(\Box)) \\ \hat{e}_2 &:= \chi_{t2:b2/title}(\chi_{a2:b2/author[a2']}(\\ &\qquad \Upsilon_{b2:d2/book}(\chi_{d2:doc}(\Box)))) \end{aligned}$$

and

```
doc = doc("bib.xml")
s1  = "<author><name>"
s2  = "</name>"
s3  = "</author>"
```

Looking at the left-hand sides of our unnesting equivalences, Eqvs. 6 and 7 are obvious candidates. To verify the conditions mentioned in the text in Sec. 4 is easy for Eqv. 6. In order to meet the conditions of Eqv. 7, we have to project unneeded attributes away. (Although not necessary, we also do so for Eqv. 6.) Hence, we define $e_1 := \Pi_{a_1}(\hat{e}_1)$ and $e_2 := \Pi_{a_2,t_2}(\hat{e}_2)$. Then, the condition $e_1 = \Pi^D_{a_1:a_2}(\Pi_{a2}(e_2))$ of Eqv. 7 obviously holds if there are no `author` elements other than those directly under `book` elements. This is the case for the DTD given in the XQuery use case document. However, it is not true for DBLP's DTD. In fact, exactly this condition escaped the authors of [27]. Still, if we knew from the document that all authors have written a book, the condition would hold.

After having checked the conditions, we can apply both equivalences if the use case document's DTD is satisfied and get the unnested argument expressions for Ξ:

$$\begin{aligned} \Xi_{s1;a1;s2;t1;s3}(\quad & \Pi_{\overline{a2'}}(\\ & e_1 \bowtie^{t1:\epsilon}_{a1=a2'} (\Gamma_{t1;=a2';\Pi_{t2}}(\mu^D_{a2}(e_2))))) \\ \Xi_{s1;a1;s2;t1;s3}(\quad & \Pi_{a1:a2'}(\Gamma_{t1;=a2';\Pi_{t2}}(\mu^D_{a2}(e_2)))) \end{aligned}$$

Note that although the order is destroyed on authors, both expressions produce the titles of each author in document order, as is required by the XQuery semantics for this query.

The latter expression can be simplified by renaming $a1$ to $a2'$:

$$\Xi_{s1;a2';s2;t1;s3}(\Gamma_{t1;=a2';\Pi_{t2}}(\mu^D_{a2}(e_2)))$$

The simplified expression can be enhanced further by using the group-detecting Ξ operator:

$${}_{s1;a2';s2}\Xi^{s3}_{a2';t2}(\mu^D_{a2}(e_2))$$

In the table below, we summarize the evaluation times for the first query. The document `bib.xml` contained either 100, 1000, or 10000 books and authors. To investigate the effect of different group sizes, we varied the number of authors per book between 2 and 10.

Plan	Authors per Book	Evaluation Time (books)		
		100	1000	10000
nested	2	0.15 s	7.04 s	788 s
	5	0.25 s	17.06 s	1678 s
	10	0.40 s	31.65 s	3195 s
outer join	2	0.08 s	0.12 s	0.57 s
	5	0.09 s	0.17 s	1.17 s
	10	0.09 s	0.25 s	2.45 s
grouping	2	0.08 s	0.11 s	0.39 s
	5	0.09 s	0.16 s	0.87 s
	10	0.10 s	0.27 s	2.07 s
group Ξ	2	0.07 s	0.09 s	0.33 s
	5	0.07 s	0.13 s	0.73 s
	10	0.08 s	0.17 s	1.37 s

While the query plan using the outer join needs to scan the input document twice and the last two plans just once, the nested plan needs to scan the document $|author| + 1$ times where $|author|$ is the number of author elements in the input document. The measurements demonstrate the massive performance improvements as an immediate consequence. However, all evaluation plans scale approximately linear for the size of each group.

To give some performance numbers on a reasonably sized document, we also ran the query against the DBLP database comprising about 140 MB. This XML document contains publications including books, articles, theses and so on. Each publication may have child nodes which are authors. When we evaluate the example query against this

---

[4] XQuery specifies that the result sequences the `return` clause generates for every tuple binding are concatenated.

document, we may not apply Eqv. 7 because there are authors that have not published a book. Thus, we have to stay with the more general plan using the outer join. This plan takes 13.95 seconds to evaluate. This is in stark contrast to the execution time of the nested plan taking 182h42m, which is a little more than a week! Due to this high execution time, we limit ourselves to smaller documents for the rest of the paper.

5.2. Query 1.1.9.10 (Aggregation)

Aggregation is often used in conjunction with grouping. The second query extends the first query with an aggregation.

```
let $d1 := doc("prices.xml")
for $t1 in distinct-values($d1//book/title)
let $p1 := let $d2 := doc("prices.xml")
           for $p2 in $d2//book[title = $t1]
                      /price
           return decimal($p2)
return
  <minprice title="{ $t1 }">
     <price> { min( $p1 ) } </price>
  </minprice>
```

We first normalize the query. In general, we have to be very careful when rewriting a path expression. Breaking up the XPath expression in the query is only possible because we know from the DTD that every `book` element has exactly one `price` child element.

```
let $d1 := doc("prices.xml")
for $t1 in distinct-values($d1//book/title)
let $m1 := min(
            let $d2 := doc("prices.xml")
            for $b2 in $d2//book
            let $t2 := $b2/title
            let $p2 := $b2/price
            let $c2 := decimal($p2)
            where $t1 = $t2
            return $c2)
return
  <minprice title="{ $t1 }">
     <price> { $m1 } </price>
  </minprice>
```

Knowing that every `book` element has exactly one `title` child element[5], translation yields

$$\Xi_{s1,t1,s2;m1;s3}(\chi_{m1:min(\Pi_{c2}(\sigma_{t1=t2}(\hat{e}_2)))}(\hat{e}_1))$$

where

$$\begin{aligned}\hat{e}_1 &= \Upsilon_{t1:\Pi^D(d1//book/title)}(\chi_{d1:doc}(\Box))\\ \hat{e}_2 &= \chi_{c2:decimal(p2)}(\chi_{p2:b2/price}(\chi_{t2:b2/title}(\\ &\qquad \Upsilon_{b2:d2//book}(\chi_{d2:doc}(\Box)))))\end{aligned}$$

and

```
doc = doc("prices.xml")
s1 = "<minprice title=\""
s2 = "\"><price>"
s3 = "</price></minprice>"
```

Let us again project unneeded attributes away and define $e_1 := \Pi_{t1}(\hat{e}_1)$ and $e_2 := \Pi_{t2,c2}(\hat{e}_2)$. Since only `title` elements under `book` elements are considered, not only are Eqvs. 3 and 4 applicable but the restriction $e_1 = \Pi^D_{t1:t2}(\Pi_{t2}(e_2))$ holds and Eqv. 5 can be used. Since the latter results in the most efficient plan, we neglect the other possibilities for space reasons. Applying Eqv. 5 leaves us with

$$\Xi_{s1,t1,s2;m1;s3}(\Pi_{t1:t2}(\Gamma_{m1;=t2;min\circ\Pi_{c2}}(e_2)))$$

Below, we compare the evaluation times for the two plans. Again we observe impressive performance gains for the same reasons as previously explained.

	Evaluation Time (books)		
Plan	100	1000	10000
nested	0.09 s	1.81 s	173.51 s
grouping	0.07 s	0.08 s	0.19 s

5.3. Query 1.1.9.5 (Existential Quantifier)

The third example query uses a nested existentially quantified expression in the **where** clause. Note that although order preservation within the quantifier is not needed, the following query retrieves `title` elements in document order.

```
let $d1 := document("bib.xml")
for $t1 in $d1//book/title
where some $t2 in
         document("reviews.xml")//entry/title
      satisfies $t1 = $t2
return
  <book-with-review>
     { $t1 }
  </book-with-review>
```

Normalized, this query reads

```
let $d1 := document("bib.xml")
for $t1 in $d1//book/title
where some $t2 in (
         let $d3 := document("reviews.xml")
         for $t3 in $d3//entry/title
         return $t3 )
      satisfies $t1 = $t2
return
  <book-with-review>
     { $t1 }
  </book-with-review>
```

---

[5] Otherwise, translation must use ‘$\in$’ instead of ‘$=$’.

We can move the correlation predicate into the range expression and translate the normalized query into

$$\Xi_{s1;t1;s2}(\sigma_{\exists t2 \in e_3}\ \text{true}(e_1))$$

where

$$\begin{array}{lcl} e_1 & := & \Upsilon_{t1:d1//book/title}(\chi_{d1:doc1}(\Box)) \\ e_2 & := & \Upsilon_{t3:d3//entry/title}(\chi_{d3:doc3}(\Box)) \\ e_3 & := & \Pi_{t3}(\sigma_{t1=t3}(e_2)) \end{array}$$

and

```
doc1 = document("bib.xml")
doc3 = document("reviews.xml")
s1   = "<book-with-review>"
s2   = "</book-with-review>"
```

We use Eqv. 8 to get

$$\Xi_{s1;t1;s2}(e_1 \ltimes_{t1=t3} e_2).$$

The performance of these two evaluation plans is compared in the following table. As in the previous examples, the unnested query plan scales better for larger input documents.

	Evaluation Time (books/reviews)		
Plan	100	1000	10000
nested	0.10 s	1.83 s	175.80 s
semijoin	0.08 s	0.09 s	0.20 s

5.4. Query with Universal Quantifier

Besides existential quantification, XQuery supports universal quantification. The following example returns the authors whose books were all published after 1993.

```
let $d1 := doc("bib.xml")
for $a1 in distinct-values($d1//author)
where every $b2 in doc("bib.xml")//
                   book[author = $a1]
      satisfies $b2/@year > 1993
return
  <new-author>
     { $a1 }
  <new-author>
```

Normalization is a little more complex here, as some more rewrites are necessary. First, for the range expression of the quantifier (`doc("bib.xml")//book[author = $a1]`) we introduce a new query block (FLWR expression). Then we unnest the `authors` of the correlation predicate. Finally, since the `year` attribute is the only information about books needed in the `satisfies` part of the quantifier, we change the range variable. As these rewrites have been discussed in depth (see [5]), we do not detail on them here. They result in

```
let $d1 :=  doc("bib.xml")
for $a1 in distinct-values($d1//author)
where every $y2 in (
        let $d3 := doc("bib.xml")
        for $b3 in $d3//book
        let $y3 := $b3/@year
        for $a3 in $b3/author
        where $a1 = $a3
        return $y3 )
      satisfies $y2 > 1993
return
  <new-author>
     { $a1 }
  <new-author>
```

The nested query plan is derived by application of the translation rules.

$$\Xi_{s1;a1;s2}(\sigma_{\forall y2 \in e2\ y2>1993}(e_1))$$

where

$$\begin{array}{lcl} e_1 & = & \Upsilon_{a1:\Pi^D(d1//author)}(\chi_{d1:doc}(\Box)) \\ e_2 & = & \Pi_{y3}(\sigma_{a1=a3}(e_3)) \\ e_3 & = & \Upsilon_{a3:b3/author}(\chi_{y3:b3/@year}(\Upsilon_{b3:d3//book}(\\ & & \quad \chi_{d3:doc}(\Box)))) \end{array}$$

and

```
doc = doc("bib.xml")
s1  = "<new-author>"
s2  = "</new-author>"
```

Eqv. 9 is applicable and yields

$$\Xi_{s1;a1;s2}(e_1 \triangleright_{a1=a3 \wedge y3 \leq 1993} e_3)$$

Of course we can push the second part of the join predicate into its second operand. This yields

$$\Xi_{s1;a1;s2}(e_1 \triangleright_{a1=a3} \sigma_{y3 \leq 1993}(e_3))$$

Since we know from the DTD that `author` elements occur only under `book` elements, $\Pi^D_{a1}(e_1) = \Pi^D_{a_1:a_3}(\Pi_{a_3}(e_3))$ holds and thus, we can apply Eqv. 2, which yields:

$$\Xi_{s1;a1;s2}(\sigma_{c=0}(\Gamma_{c;=aa;count \circ \sigma_{y3 \leq 1993}}(e_3)))$$

A comparison of the evaluation times of the discussed plans is given in the table below. The unnested query plans scale better than the nested plan because they need to scan the input document once or twice. In contrast to that the nested query needs to scan it for each author element in the input document.

Plan	Evaluation Time (books)		
	100	1000	10000
nested	0.12 s	4.86 s	507.85 s
anti-semijoin	0.07 s	0.08 s	0.24 s
grouping	0.07 s	0.08 s	0.23 s

5.5. Query 1.4.4.14

In our last example query nesting occurs in a predicate in the **where** clause depending on an aggregate function, `count` in this case. This is similar to a having-clause in SQL: after grouping bids by `itemno`, they are selected by the result of the aggregation.

```
let $d1 := document("bids.xml")
for $i1 in distinct-values($d1//itemno)
where
  count($d1//bidtuple[itemno = $i1]) >= 3
return
  <popular-item>
     { $i1 }
  </popular-item>
```

During normalization we extract the left argument of the general comparison, turn it into a **let** clause, and move the XPath predicate into a **where** clause.

```
let $d1 := document("bids.xml")
for $i1 in distinct-values($d1//itemno)
let $c1 := count(
           let $d2 := document("bids.xml")
           for $i2 = $d2//bidtuple/itemno
           where $i1 = $i2
           return $i2)
where $c1 >= 3
return
  <popular_item>
     { $i1 }
  </popular_item>
```

Now the translation into our algebra is easy. As the result, tuples from the inner query are counted, we do not need to introduce a Ξ operator, an attribute $i2'$, or project down to $i2$.

$$\Xi_{s1,i1,s2}(\sigma_{c1>=3}(\chi_{c1:count(\sigma_{i1=i2}(\hat{e}_2))}(\hat{e}_1)))$$

where

$$\begin{array}{rcl}\hat{e}_1 & := & \Upsilon_{i1:\Pi^D(d1//itemno)}(\chi_{d1:doc}(\Box)) \\ \hat{e}_2 & := & \Upsilon_{i2:d2//bidtuple/itemno}(\chi_{d2:doc}(\Box))\end{array}$$

and

```
doc = document("bids.xml")
s1  = "<popular_item>"
s2  = "</popular_item>"
```

We would like to apply Eqv. 5 for unnesting the above expression. In order to do that, we have to check that the prerequisites hold. Projecting away unnecessary attributes, we define $e_1 := \Pi_{i1}(\hat{e}_1)$ and $e_2 := \Pi_{i2}(\hat{e}_2)$. Looking at the DTD of bids.xml, we see that `itemno` elements appear only directly beneath `bidtuple` elements. Thus, the condition $e_1 = \Pi^D_{i1:i2}(\Pi_{i2}(e_2))$ holds and we can apply Eqv. 5:

$$\Xi_{s1,i1,s2}(\sigma_{c1>=3}(\Pi_{i1:i2}(\Gamma_{c1;=i2;count}(e2))))$$

The evaluation times for each plan are given in the table below. The number of bids and items is varied. On average, each item has five bids. Again, the measurements verify the effectiveness of the unnesting techniques.

Plan	Evaluation Time (bids)		
	100	1000	10000
nested	0.06 s	0.53 s	48.1 s
grouping	0.06 s	0.07 s	0.10 s

6. Conclusion and Outlook

In the core of the paper, we presented equivalences that allow to unnest nested algebraic expressions. Some of these equivalences are counterparts of existing equivalences valid for algebras whose operators do not preserve order. For others, no counterpart has been published so far. We showed the correctness of all equivalences in [25], as having a counterpart in the unordered context does not automatically imply correctness for the ordered case.

Furthermore, we demonstrated each of the equivalences by means of an example. Thereby, we showed their applicability to queries with and without aggregate functions and with or without quantifiers. Further, we experimentally compared the performance of the nested algebraic expressions with the unnested algebraic expressions. In doing so, enormous performance improvements could be observed. These measurements include the first extensive experiments on the effectiveness of unnesting techniques for XML queries. Besides our measurements only [27] hint on some performance numbers for their unnesting algorithm.

A general question that remains unanswered is whether an operator-based representation is suited for evaluating XQuery efficiently. However, in order to answer this question, we first have to fathom what can be achieved with this representation. Therefore, in the near future we plan to check out how far this approach can be taken.

Acknowledgment. We thank Simone Seeger for her help in preparing the manuscript and C.-C. Kanne for his help in carrying out the experiments. We also thank the anonymous referees for giving helpful hints on improving this paper.

References

[1] M. M. Astrahan and D. D. Chamberlin. Implementation of a structured English query language. *Communications of the ACM*, 18(10):580–588, 1975.

[2] C. Beeri and Y. Tzaban. SAL: An algebra for semistructured data and XML. In *ACM SIGMOD Workshop on the Web and Databases (WebDB)*, 1999.

[3] G. Bhargava, P. Goel, and B. Iyer. Hypergraph based reorderings of outer join queries with complex predicates. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 304–315, 1995.

[4] S. Chaudhuri and K. Shim. Optimizing queries with aggregate views. In *Proc. of the Int. Conf. on Extending Database Technology (EDBT)*, pages 167–182, 1996.

[5] J. Claussen, A. Kemper, G. Moerkotte, and K. Peithner. Optimizing queries with universal quantification in object-oriented and object-relational databases. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 286–295, 1997.

[6] S. Cluet and C. Delobel. A general framework for the optimization of object-oriented queries. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 383–392, 1992.

[7] S. Cluet and G. Moerkotte. Nested queries in object bases. In *Proc. Int. Workshop on Database Programming Languages*, 1993.

[8] S. Cluet and G. Moerkotte. Classification and optimization of nested queries in object bases. Technical Report 95-6, RWTH Aachen, 1995.

[9] U. Dayal. Of nests and trees: A unified approach to processing queries that contain nested subqueries, aggregates, and quantifiers. In *VLDB*, pages 197–208, 1987.

[10] Leonidas Fegaras, David Levine, Sujoe Bose, and Vamsi Chaluvadi. Query processing of streamed XML data. In *Proceedings of the 2002 ACM CIKM International Conference on Information and Knowledge Management, McLean, VA, USA, November 4-9, 2002*, pages 126–133. ACM, 2002.

[11] Leonidas Fegaras and David Maier. Optimizing object queries using an effective calculus. *ACM Transactions on Database Systems*, 25(4):457–516, 2000.

[12] T. Fiebig, S. Helmer, C.-C. Kanne, G. Moerkotte, J. Neumann, R. Schiele, and T. Westmann. Anatomy of a Native XML Base Management System. *VLDB Journal*, 11(4):292–314, 2002.

[13] T. Fiebig and G. Moerkotte. Algebraic XML construction and its optimization in Natix. *World Wide Web Journal*, 4(3):167–187, 2002.

[14] C. Galindo-Legaria and A. Rosenthal. Outerjoin simplification and reordering for query optimization. *ACM Transactions on Database Systems*, 22(1):43–73, March 1997.

[15] R. Ganski and H. Wong. Optimization of nested SQL queries revisited. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 23–33, 1987.

[16] G. Gottlob, C. Koch, and R. Pichler. Efficient algorithms for processing XPath queries. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 95–106, 2002.

[17] G. Gottlob, C. Koch, and R. Pichler. XPath query evaluation: Improving time and space efficiency. In *Proc. of the 19th Int. Conf. on Data Engineering*, pages 379–390, 2003.

[18] W. Hasan and H. Pirahesh. Query rewrite optimization in starburst. Research Report RJ6367, IBM, 1988.

[19] S. Helmer, C.-C. Kanne, and G. Moerkotte. Optimized translation of xpath expressions into algebraic expressions parameterized by programs containing navigational primitives. In *Proc. Int. Conf. on Web Information Systems Engineering (WISE)*, 2002. 215-224.

[20] H. V. Jagadish, L. V. S. Lakshmanan, D. Srivastava, and K. Thompson. TAX: A tree algebra for XML. In *Proc. Int. Workshop on Database Programming Languages*, pages 149–164, 2001.

[21] W. Kiessling. SQL-like and Quel-like correlation queries with aggregates revisited. ERL/UCB Memo 84/75, University of Berkeley, 1984.

[22] W. Kim. On optimizing an SQL-like nested query. *ACM Transactions on Database Systems*, 7(3):443–434, September 1982.

[23] A. Klug. Equivalence of relational algebra and relational calculus query languages having aggregate functions. *Journal of the ACM*, 29(3):699–717, 1982.

[24] A. Lerner and D. Shasha. AQuery: Query language for ordered data, optimization techniques, and experiments. In *Proc. of the 29th VLDB Conference*, pages 345–356, Berlin, Germany, 2003.

[25] N. May, S. Helmer, and G. Moerkotte. Nested queries and quantifiers in an ordered context. Technical report, University of Mannheim, 2003.

[26] M. Muralikrishna. Improved unnesting algorithms for join aggregate SQL queries. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 91–102, 1992.

[27] S. Paparizos, S. Al-Khalifa, H. V. Jagadish, L. V. S. Lakshmanan, A. Nierman, D. Srivastava, and Y. Wu. Grouping in XML. In *EDBT Workshops*, pages 128–147, 2002.

[28] H. Pirahesh, J. Hellerstein, and W. Hasan. Extensible/rule-based query rewrite optimization in Starburst. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 39–48, 1992.

[29] A. Rosenthal and C. Galindo-Legaria. Query graphs, implementing trees, and freely-reorderable outerjoins. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 291–299, 1990.

[30] P. Seshadri, H. Pirahesh, and T. Leung. Complex query decorrelation. In *Proc. IEEE Conference on Data Engineering*, pages 450–458, 1996.

[31] H. Steenhagen, P. Apers, and H. Blanken. Optimization of nested queries in a complex object model. In *Proc. of the Int. Conf. on Extending Database Technology (EDBT)*, pages 337–350, 1994.

[32] H. Steenhagen, P. Apers, H. Blanken, and R. de By. From nested-loop to join queries in oodb. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 618–629, 1994.

[33] H. Steenhagen, R. de By, and H. Blanken. Translating OSQL queries into efficient set expressions. In *Proc. of the Int. Conf. on Extending Database Technology (EDBT)*, pages 183–197, 1996.