

Three Cases for Query Decorrelation in XQuery

Norman May, Sven Helmer, and Guido Moerkotte

Universität Mannheim

D7, 27

Mannheim

Germany

`norman|helmer|moer@pi3.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 [26]. Very quickly, it became clear that enormous performance gains are possible by avoiding nested-loops 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 [27]. Retrospectively, we can summarize the problem areas as follows:

- Special cases like empty results lead easily to bugs like the count bug [27]. These have been corrected by different approaches [9, 18, 25, 27, 30, 31].
- If the nested query contains grouping, special rules are needed to pull up grouping operators [5].
- Special care has to be taken for a correct duplicate treatment [21, 28, 35, 37, 38].

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, 18, 25, 30]. After their introduction into SQL and their usage for unnesting, reordering of outer joins became an important topic [3, 17, 36]. A unifying framework for different unnesting strategies for SQL can be found in [31].

With the advent of object-oriented databases and their query languages, unnesting once again drew some attention [7, 8, 12, 40–43]. In contrast to the relational unnesting strategies, which performed unnesting at the (extended) SQL source level, most 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 retain order. Fegaras and Maier describe an alternative approach in which queries are translated into a monoid comprehension calculus representation [12]. The actual unnesting of the queries is done in terms of this calculus. A disadvantage of this approach is the need for another level of representation (additionally to the algebraic representation).

XQuery¹ is a query language that allows the user to specify whether to retain the order of its input documents or not. If the result's order is relevant, the unnesting techniques from the object-oriented context cannot be applied.

Consequently, the area of unnesting nested queries was reopened for XQuery by Fegaras et al. [11] and Paparizos et al. [32]. Fegaras et al. focus on unnesting queries operating on streams. It is unclear to which extent order preservation is considered (e.g. on the algebra level hash joins are used, which are not necessarily known for their order preserving properties). Another publication by Fegaras et al. [10] describes the translation of XML-OQL into OQL, but is not concerned with unnesting. The approach by Paparizos et al. describes the introduction of a grouping operator for a nested query. However, their verbal description of this 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 of their technique. To the best of our knowledge, no other paper discusses unnesting in the ordered context.

Within this paper we introduce several different unnesting strategies and discuss their application to different query types. All these unnesting techniques are described by means of algebraic equivalences which we proved to be correct. We also provide performance figures for every query execution plan demonstrating the significant speed-up gained by unnesting.

Our Unnesting Approach consists of three steps (in this paper we focus on the third step, details on the first two steps can be found in [29]):

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

¹ <http://www.w3.org/TR/xquery>

The remainder of the paper is organized as follows. Section 2 briefly motivates and defines our algebra. In Section 3 we introduce equivalences used for unnesting queries via exemplary XQuery queries taken from the XQuery use-case document². Section 4 concludes the paper.

2 Notation and Algebra

We do not use the TAX-Algebra [24] but our NAL-algebra instead, which 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. Lately, another core for an order-preserving algebra was presented [39]. Although it has been developed for OLAP-style querying, it is (almost) identical to the core of SAL and NAL. This algebra is very similar to XAL, another core algebra for ordered sequences [15]. All NAL operators except for our result constructing operator Ξ can be found in the literature. Nevertheless, we give definitions for all algebraic operators to make the paper self-contained.

The reason for using NAL instead of TAX is that NAL is close to our physical algebra and we believe that a physical algebra for TAX [33] is not as efficient as one for NAL. Our reasons for this belief are that (1) trees are dynamically generated for intermediate results and (2) selection predicates, join predicates and the like all rely on pattern matching on these trees. In our algebra we also allow tree-valued attributes but try to restrict their contents to node handles pointing to nodes in trees stored in the database. Of course our approach can only be justified or falsified by extensively benchmarking Timber [23] and Natix [13].

Our algebra works on sequences of sets of variable bindings, i.e. sequences of 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 using the standard $[\cdot]$ brackets. 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 $\mathcal{F}(e)$.

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 $\mathcal{A}(e_2) \subseteq \mathcal{F}(e_1)$. 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 equate elements with single element sequences. 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 $e[a] = [a : \alpha(e)] \oplus \tau(e)[a]$.

² <http://www.w3.org/TR/xmlquery-use-cases>

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 (\square) 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 not 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 expressions. Hence, a join within a selection predicate containing a nested algebraic expression 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_A^D . 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. When we want to eliminate the set of attributes A , we denote this by $\Pi_{\bar{A}}$. We use Π for renaming attributes using the notation $\Pi_{A':A}$. Here, 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 $e_2(t_1)$.

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

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

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:

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

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

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

where $g \in \mathcal{A}(e_2)$

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

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

where the **binary grouping** operator (sometimes called nest-join [40,41]) 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, $G(x) := f(\sigma_{x|_{A_1\theta A_2}}(e_2))$.

Given a tuple with a sequence-valued attribute, we can unnest it 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 $\perp_{\mathcal{A}(e.g)}$.

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 [19, 22], 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 . An optimized translation is well beyond the scope of the paper.

For **result construction**, we employ a simplified operator Ξ that combines a pair of Groupify-GroupApply operators [14]. 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, besides 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. Similarly for more complex expressions. 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, $\Xi^{<author>";a;"</author>}(e)$ embeds every author name into an `author` element.

In its group detecting form, ${}^{s_1}\Xi_{s_2(g)}^{s_3}$ uses a list of grouping attributes (g) and three sequences of commands. We define

$${}^{s_1}\Xi_{s_2(g)}^{s_3}(e) := \Xi_{(s_1;\Xi_{s_2(g)};s_3)}(I_{g:=A;\Pi_{g'}}e)$$

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 g . 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)—which may only depend on the attribute g' in g —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 g . Introducing the complex Ξ saves a grouping operation that would have to construct a sequence-valued attribute.

Let us illustrate ${}^{s_1}\Xi_{s_2(g)}^{s_3}(e)$ 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 ${}^{s_1}\Xi_{s_2(a)}^{s_3}(e)$ 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>
```

One word on implementation. Standard implementation techniques for algebraic operators [20] do not preserve order. Claussen et al. provide an efficient

implementation for an order-preserving hash join [6]. Currently, we have not implemented it but use a Grace-Hash-Join [16] instead with a subsequent sorting operator to restore order. Further performance enhancements for unnested plans with joins can be expected when using the order-preserving hash join [6]. The technique presented there can be adopted to implement unary and binary grouping. For another implementation of the binary grouping operator see [4]. There, it is called MD-Join. We would also like to point out that the \mathcal{Y} operator generates its output in document order if the translation of XPath expressions described in [22] is used.

On the physical level there are still various optimization techniques left that we have not exploited yet. So, we expect even better performance figures when employing all these techniques.

3 Three Cases for Unnesting

In this section we present several examples of nested queries (based on the XQuery use-case document) for which unnesting techniques result in major performance gains. For space reasons we omit the details of the normalization and translation (see [29] for details). We concentrate on the equivalences used for unnesting our example queries.

```

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

<!DOCTYPE items [
  <!ELEMENT items (item_tuple*)>
  <!ELEMENT item_tuple (itemno, description, offered_by, start_date?, end_date?, reserve_price? )>
  <!ELEMENT itemno (#PCDATA )>
  <!ELEMENT description (#PCDATA )>
  <!ELEMENT offered_by (#PCDATA )>
  <!ELEMENT start_date (#PCDATA )>
  <!ELEMENT end_date (#PCDATA )>
  <!ELEMENT reserve_price (#PCDATA )>
]>

<!DOCTYPE users [
  <!ELEMENT users (user_tuple*)>
  <!ELEMENT user_tuple (userid, name, rating?)>
  <!ELEMENT userid (#PCDATA )>
  <!ELEMENT name (#PCDATA )>
  <!ELEMENT rating (#PCDATA )>
]>

<!DOCTYPE bids [
  <!ELEMENT bids (bid_tuple*)>
  <!ELEMENT bid_tuple (userid, itemno, bid, bid_date)>
  <!ELEMENT userid (#PCDATA )>
  <!ELEMENT itemno (#PCDATA )>
  <!ELEMENT bid (#PCDATA )>
  <!ELEMENT bid_date (#PCDATA )>
]>

```

Fig. 1. DTDs for the example queries

We verified the effectiveness of the unnesting techniques experimentally. The experiments were carried out on a simple PC with a 2.4 Ghz Pentium using the Natix query evaluation engine [13]. The database cache was configured such that it could hold the queried documents. The XML files were generated with the help of ToXgene using the DTDs from the XQuery use-case document which

are shown in Fig. 1. We executed the various evaluation plans on different sizes of input documents (varying the number of elements).

3.1 Grouping

The first query is a typical example where the nested query is used to express grouping. Frequently, (after normalization) an expression bound in the **let** clause originally occurred in the **return** clause, which is an equivalent way of expressing grouping in XQuery. The normalization step takes care of the uniform treatment of these different formulations.

```

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>
    { $t1 }
  </author>

```

The translation algorithm maps the **for** clause to the Υ operator and the **let** clause to the χ operator. The subscript of these operators is defined by the binding expression of both clauses. (Frequently, these binding expressions contain XPath expressions. However, since we concentrate on unnesting techniques in XQuery in this paper, we rely on efficient translation and evaluation techniques [19, 22]. These techniques can be used orthogonally to the methods presented in this paper.) The **where** clause is translated into a σ operator and the Ξ operator constructs the query result as defined in the **return** clause. We greatly simplify the translation of the **return** clause and refer to [14] for a more advanced treatment of result construction in XQuery. The function `distinct-values` is mapped to the Π^D operator.

For the resulting query we get the following algebraic expression:

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

where

$$\begin{aligned} \hat{e}_1 &:= \Upsilon_{a1:\Pi^D(d1//author)}(\chi_{d1:doc(\square)}) & \text{doc} &= \text{doc}(\text{"bib.xml"}) \\ \hat{e}_2 &:= \chi_{t2:b2/title}(\chi_{a2:b2/author[a2']}(\hat{e}_3)) & \text{s1} &= \text{"<author><name>"} \\ \hat{e}_3 &:= \Upsilon_{b2:d2/book}(\chi_{d2:doc(\square)}) & \text{s2} &= \text{"</name>"} \\ & & \text{s3} &= \text{"</author>"} \end{aligned}$$

During translation we have to ensure the existential semantics of the general comparison in XQuery. 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` element 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.³ Therefore, we can apply a simple projection on $t2$ to model the `return` clause of the inner query block.

As already mentioned, the first example query performs a grouping operation and our unnesting equivalences recognize these semantics. Potentially, this yields more efficient execution plans. We have to be careful, however, because the range of values of the grouping attributes of the outer query block might differ from that of the grouping attributes of the inner query block. Therefore, we advocate the use of a left outer join. We propose the following new unnesting equivalence:

$$\chi_{g:f(\sigma_{A_1 \in a_2}(e_2))}(e_1) = \Pi_{A_2}(e_1 \bowtie_{A_1=A_2}^{g:f(\epsilon)} \Gamma_{g:=A_2;f}(\mu_{a_2}^D(e_2))) \quad (1)$$

which holds, if

$$A_i \subseteq \mathcal{A}(e_i), \mathcal{F}(e_2) \cap \mathcal{A}(e_1) = \emptyset, g \notin \mathcal{A}(e_1) \cup \mathcal{A}(e_2), A_1 \cap A_2 = \emptyset, \\ a_2 \in \mathcal{A}(e_2), A_2 = \mathcal{A}(a_2), \text{ (this implies that } A_1 = \mathcal{A}(e_1))$$

Note that each book can be written by multiple authors. Thus, for the right hand side of the equivalence to be correct we have to unnest these attributes before grouping them by the correlating attributes. This way we explicitly handle the existential semantics of the general comparison. Applying this equivalence to the example query we get:

$$\Xi_{s1;a1;s2;t1;s3}(\Pi_{a2'}(\hat{e}_1 \bowtie_{a1=a2'}^{t1:\epsilon} (\Gamma_{t1:=a2';\Pi_{t2}}(\mu_{a2}^D(\hat{e}_2))))))$$

where \hat{e}_1 , \hat{e}_2 , $s1$, $s2$, and $s3$ are defined as above

There exist alternatives. In our example we know from the DTD that no other `author` elements other than those directly beneath `book` elements can be found in the queried document. Furthermore, if we also know from the document that all authors have written at least one book, we become aware of the fact that the outer and the nested query block actually retrieve their data from the same document. In this case we can apply the following equivalence (in fact, this condition escaped the authors of [34]):

$$\chi_{g:f(\sigma_{A_1 \in a_2}(e_2))}(e_1) = \Pi_{A_1:A_2}(\Gamma_{g:=A_2;f}(\mu_{a_2}^D(e_2))) \quad (2)$$

(Formally speaking, in addition to the preconditions of (1) we have to satisfy $e_1 = \Pi_{A_1:A_2}^D(\Pi_{A_2}(\mu_{a_2}(e_2)))$ to be able to apply this equivalence.)

³ XQuery specifies that the result sequences the `return` clause generates for every tuple binding are concatenated.

Since for the example query this is the case when we define $e'_1 = \Pi_{a_1}(\hat{e}_1)$ and $e'_2 = \Pi_{a_2, t_2}(\hat{e}_2)$, we get:

$$\Xi_{s1;a1;s2;t1;s3}(\Pi_{a1:a2'}(\Gamma_{t1:=a2';\Pi_{t2}}(\mu_{a2}^D(e'_2))))$$

where e'_1 , e'_2 , $s1$, $s2$, and $s3$ are defined as above

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. While the unnesting algorithm published in [11, 12] is able to unnest many more nested expressions, the resulting query does not preserve order (a hash join operator is used). In [34] unnesting is described rather informally making it difficult to apply the technique in a general context. In our approach both the implementation of the algebraic operators and the transformations via equivalences preserve the document order (for proofs see [29]).

After renaming $a1$ to $a2'$ the expression can be enhanced further by using the group detecting Ξ operator as defined in Section 2:

$$s1;a2';s2 \Xi_{t2}^{s3} a2'(\mu_{a2}^D(e'_2))$$

After applying all our rewrites, we need to scan the document just once. (A naive nested-loop evaluation leads to multiple scans of the document.) 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 10 authors per book. This demonstrates the massive performance improvements that are possible by unnesting queries.

Plan	Evaluation Time		
	100	1000	10000
nested	0.40 s	31.65 s	3195 s
outerjoin	0.09 s	0.25 s	2.45 s
grouping	0.10 s	0.27 s	2.07 s
group Ξ	0.08 s	0.17 s	1.37 s

3.2 Grouping and Aggregation

Aggregation is often used in conjunction with grouping. We illustrate further unnesting equivalences using the second example query which introduces an aggregation in addition to grouping.

```
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 from the DTD that every book element has exactly one `title` child element⁴, the translation yields

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

where

$$\begin{aligned} \hat{e}_1 &= \mathcal{I}_{t1:\Pi^D}(d1//book/title)(\chi_{d1:doc}(\square)) & \text{doc} &= \text{doc}(\text{"prices.xml"}) \\ \hat{e}_2 &= \chi_{c2:decimal}(p2)(\chi_{p2:b2/price}(\hat{e}_3)) & \text{and} & \text{s1} = "<minprice title=\"\" \\ \hat{e}_3 &= \chi_{t2:b2/title}(\mathcal{I}_{b2:d2//book}(\chi_{d2:doc}(\square))) & & \text{s2} = "\"><price>" \\ & & & \text{s3} = "</price></minprice>" \end{aligned}$$

To unnest this expression we propose the following new equivalence:

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

which holds, if

$$\begin{aligned} A_i &\subseteq \mathcal{A}(e_i), \mathcal{F}(e_2) \cap \mathcal{A}(e_1) = \emptyset, g \notin \mathcal{A}(e_1) \cup \mathcal{A}(e_2), A_1 \cap A_2 = \emptyset, \\ e_1 &= \Pi_{A_1:A_2}^D(\Pi_{A_2}(e_2)) \end{aligned}$$

The equivalence is similar to the equivalence used for the first example query. In the equivalence no unnesting needs to be applied before grouping because attribute A_2 is an atomic value. We have equivalences for more general cases where the last restriction is not fulfilled or where the correlation needs not be equality [29].

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, the restriction $e'_1 = \Pi_{t1:t2}^D(\Pi_{t2}(e'_2))$ holds and Eqv. 3 can be applied resulting in

$$\Xi_{s1,t1,s2;m1;s3}(\Pi_{t1:t2}(\Gamma_{m1:=t2;mino\Pi_{c2}}(e'_2)))$$

Below, we compare the evaluation times for the two plans with varying numbers of books.

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

⁴ Otherwise, translation must use ‘ \in ’ instead of ‘ $=$ ’.

Again we observe massively improved execution times after unnesting the query because the unnested query plan needs to scan the source document only once. Assuming a naive execution strategy the nested query plan scans the document $|title| + 1$ times. Thus the benefit of unnesting increases with the number of title elements in the document.

3.3 Aggregation in the Where Clause

In our last example query nesting occurs in a predicate in the **where** clause that depends on an aggregate function, in this case **count**. Our normalization heuristics moves the nested query into a **let** clause and the result of the nested query is applied in the **where** clause. Thus the normalized query is:

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

We do not use a result construction operator on the inner query block because we do not return XML fragments but merely tuples containing a count. Hence, a projection is sufficient.

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

where

$$\begin{aligned} \hat{e}_1 &:= \mathcal{T}_{i1:\Pi^D(d1//itemno)}(\chi_{d1:doc}(\square)) & \text{and } s1 &= "<popular_item>" \\ \hat{e}_2 &:= \mathcal{T}_{i2:d2//bidtuple/itemno}(\chi_{d2:doc}(\square)) & s2 &= "</popular_item>" \end{aligned}$$

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 $e'_1 = \Pi_{i1:i2}^D(\Pi_{i2}(e'_2))$ holds and we can apply Eqv. 3:

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

The evaluation times for each plan are given in the table below. The number of bids and items is varied between 100 and 10,000. The number of items equals 0.2 times the number of bids. Again we observe, that the unnested evaluation plan scales better than the nested plan.

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

4 Conclusion

In the core of the paper we presented equivalences that allow to unnest nested algebraic expressions. Eqvs. (1) and (2) are new in both the ordered and the unordered context. An equivalent of Eqv. (2) in the ordered context appeared in [32] but without giving the important condition $e_1 = \Pi_{A_1:A_2}^D(\Pi_{A_2}(e_2))$. The proofs of the equivalences are more complex in the ordered context and can be found in [29].

We demonstrated each of the equivalences by means of an example. Thereby, we showed their applicability to queries with and without aggregate functions. The experiments conducted in this paper include the first extensive performance numbers on the effectiveness of unnesting techniques for XML queries. We observed enormous performance improvements verifying the benefits of applying the rewrites. Besides our measurements only [32] hint on some performance numbers for their unnesting algorithm.

The equivalences assume that the nested queries do not construct XML fragments. In many cases this restriction can be lifted using rewrite before applying the unnesting equivalences presented here. However, including these rewrites is beyond the scope of this paper.

Acknowledgment. We thank Simone Seeger for her help in preparing the manuscript and C.-C. Kanne for his help in carrying out the experiments.

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. D. Chatziantoniou, M. Akinde, T. Johnson, and S. Kim. The MD-Join: An Operator for Complex OLAP. In *Proc. IEEE Conference on Data Engineering*, pages 524–533, 2001.
5. 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.
6. J. Claussen, A. Kemper, and D. Kossmann. Order-preserving hash joins: Sorting (almost) for free. Technical Report MIP-9810, University of Passau, 1998.
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 and Ramez Elmasri. Query engines for Web-accessible XML data. In Peter M. G. Apers, Paolo Atzeni, Stefano Ceri, Stefano Paraboschi, Kotagiri Ramamohanarao, and Richard T. Snodgrass, editors, *Proceedings of the Twenty-seventh International Conference on Very Large Data Bases: Roma, Italy, 11–14th September, 2001*, pages 251–260, Los Altos, CA 94022, USA, 2001. Morgan Kaufmann Publishers.
11. 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.
12. Leonidas Fegaras and David Maier. Optimizing object queries using an effective calculus. *ACM Transactions on Database Systems*, 25(4):457–516, 2000.
13. 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.
14. T. Fiebig and G. Moerkotte. Algebraic XML construction and its optimization in Natix. *World Wide Web Journal*, 4(3):167–187, 2002.
15. F. Frasincar, G.-J. Houben, and C. Pau. XAL: An algebra for XML query optimization. In *Australasian Database Conference (ADC)*, 2002.
16. S. Fushimi, M. Kitsuregawa, and H. Tanaka. An overview of the systems software of a parallel relational database machine: GRACE. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 209–219, 1986.
17. C. Galindo-Legaria and A. Rosenthal. Outerjoin simplification and reordering for query optimization. *ACM Trans. on Database Systems*, 22(1):43–73, Marc 1997.
18. 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.
19. G. Gottlob, C. Koch, and R. Pichler. Xpath query evaluation: Improving time and space efficiency. In *Proc. IEEE Conference on Data Engineering*, page to appear, 2003.
20. G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2), June 1993.
21. W. Hasan and H. Pirahesh. Query rewrite optimization in starburst. Research Report RJ6367, IBM, 1988.
22. 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.
23. H. V. Jagadish, S. Al-Khalifa, A. Chapman, L.V.S. Lakshmanan, A. Nierman, S. Pappas, J. Patel, D. Srivastava, N. Wiwatwattana, Y. Wu, and C. Yu. TIMBER: A Native XML Database. *VLDB Journal*, 2003. to appear.
24. 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.
25. W. Kiessling. SQL-like and Quel-like correlation queries with aggregates revisited. ERL/UCB Memo 84/75, University of Berkeley, 1984.
26. W. Kim. On optimizing an SQL-like nested query. *ACM Trans. on Database Systems*, 7(3):443–469, Sep 82.

27. A. Klug. Equivalence of relational algebra and relational calculus query languages having aggregate functions. *Journal of the ACM*, 29(3):699–717, 1982.
28. C. Leung, H. Pirahesh, and P. Seshadri. Query rewrite optimization rules in IBM DB2 universal database. Research Report RJ 10103 (91919), IBM Almaden Research Division, January 1998.
29. N. May, S. Helmer, and G. Moerkotte. Nested queries and quantifiers in an ordered context. Technical Report TR-03-002, Lehrstuhl für Praktische Informatik III, Universität Mannheim, 2003.
30. M. Muralikrishna. Optimization and dataflow algorithms for nested tree queries. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, 1989.
31. M. Muralikrishna. Improved unnesting algorithms for join aggregate SQL queries. In *Proc. Int. Conf. on Very Large Data Bases (VLDB)*, pages 91–102, 1992.
32. 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.
33. S. Paparizos, S. Al-Khalifa, H. V. Jagadish, A. Niermann, and Y. Wu. A physical algebra for XML. Technical report, University of Michigan, 2002.
34. Stelios Paparizos, Shurug Al-Khalifa, H. V. Jagadish, Laks Lakshmanan, Andrew Nierman, Divesh Srivastava, and Yuqing Wu. Grouping in XML. *Lecture Notes in Computer Science*, 2490:128–??, 2002.
35. 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.
36. 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.
37. P. Seshadri, H. Pirahesh, and C. Leung. Decorrelation of complex queries. Research Report RJ 9846 (86093), IBM Almaden Research Division, June 1994.
38. P. Seshadri, H. Pirahesh, and T. Leung. Complex query decorrelation. In *Proc. IEEE Conference on Data Engineering*, pages 450–458, 1996.
39. G. Slivinskas, C. Jensen, and R. Snodgrass. Bringing order to query optimization. *SIGMOD Record*, 13(2):5–14, 2002.
40. H. Steenhagen. *Optimization of Object Query Languages*. PhD thesis, University of Twente, 1995.
41. 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.
42. 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.
43. 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.