Revisiting Source-Level XQuery Normalization

Ki-Hoon LEE††, Member and Young-Ho PARK††, Nonmember

SUMMARY XQuery has become the standard for querying XML. Just like SQL, XQuery allows nested expressions. To optimize XQuery processing, a lot of research has been done on normalization, i.e., transforming nested expressions to equivalent unnested ones. Previous normalization rules are classified into two categories—source-level and algebra-level—depending on whether a construct is specified in the XQuery syntax or as equivalent algebraic expressions. From an implementation point of view, the former is preferable to the latter since it can be implemented in a variety of XQuery engines with different algebras. However, existing source-level rules have several problems: They do not handle quantified expressions, incur duplicated query results, and use many temporary files. In this paper, we propose new source-level normalization rules that solve these problems. Through analysis and experiments, we show that our normalization rules can reduce query execution time from hours to a few seconds and can be adapted to a variety of XQuery engines.

key words: source-level XQuery normalization, query optimization

1. Introduction

XML has emerged as a standard for representing, storing, and exchanging data on the Internet, and XPath and XQuery are widely used for querying XML data [8], [12]–[14], [24]. XQuery encompasses XPath and supports a FLWOR expression that is similar to a select-from-where expression in SQL. A FLWOR expression comprises the for, let, where, order by, and return clauses, and it can include another FLWOR expression, thus allowing nested expressions. XQuery supports existentially and universally quantified expressions. Quantified expressions can be nested in the where clause of a FLWOR expression.

Processing queries that contain nested expressions can be time-consuming since nested expressions are repeatedly executed. Thus, a lot of research effort has been devoted to normalization, i.e., transforming nested expressions to equivalent unnested ones. In his seminal paper, Kim [11] opened the area of unnesting nested queries in the relational context. With the advent of the XQuery language, several normalization rules [2], [4], [12], [15], [16] have also been reported in the XML context.

Previous normalization rules are classified into two categories—source-level and algebra-level—depending on whether a construct is specified in the XQuery syntax or as equivalent algebraic expressions. Examples of the former include the rules proposed by Kim [11], Manolescu et al. [15], Lee et al. [12], and Brantner et al. [2], and examples of the latter include those proposed by Fegaras et al. [5] and May et al. [16].

From an implementation point of view, we contend that the source-level rules are preferable to the algebra-level rules. First, the algebra-level rules would be difficult to adapt to a variety of XML engines with different algebras [2]. Second, there is no uniformly accepted algebra for querying XML data yet [2]. Third, the algebra rules proposed in the literature [5], [16] are too complicated to implement practically in conventional DBMSs [12]. Nevertheless, existing source-level rules have the following problems. First, they do not handle quantified expressions. Second, they incur duplicated query results [1], [12]. Third, unnested queries obtained by the source-level normalization rules use many temporary files to decouple the execution of the inner query from that of the outer query [1], [3], [7], [10]–[12], [17], [21].

In this paper, we propose new normalization rules that solve the above problems. First, we provide normalization rules for nested expressions, thus significantly improving the power of the normalization rules. We support both existentially and universally quantified expressions. Second, our rules do not incur duplicated query results. Third, our rules use fewer temporary files than existing methods. Through analysis and experiments, we show that our normalization rules improve query performance by up to several orders of magnitude and can be adapted to a variety of XQuery engines—Saxon [20], Zorba [27], and Qizx [19]. For example, we reduce the query execution time from 11 hours to 6 seconds for a Type-J query on a 1 GBytes TPC-H database using Zorba. Furthermore, the performance improvement becomes more marked as the database size gets larger. For example, for an existentially quantified query using Qizx, when the database size increases from 100 MBytes to 1 GBytes, the improvement increases from 350.86 to 4,721.15 times. In Sect. 5, we provide an analysis on the computation complexity before and after normalization, and the analysis results agree with the experimental results.

Our normalization rules have the following limitations and assumptions: First, we require the group by clause of W3C XQuery 1.1 [26], which is an extended version of the current XQuery 1.0 Recommendation. Second, for the rules containing element constructors, node comparison opera-
tors [6], [9] such as the is operator are not allowed. Third, we assume unordered mode, which is similar to the SQL rules.

The rest of this paper is organized as follows. Section 2 introduces the XQuery language. Section 3 summarizes prior work on normalization rules for XQuery. Section 4 proposes our normalization rules and proves their correctness. Section 5 presents the rationale for normalizing nested queries. Section 6 presents the results of performance evaluation. Finally, Sect. 7 concludes the paper.

2. Background

In this section, we briefly review the XQuery language and present a few example queries. The basic unit of XQuery is an expression. We introduce the FLWOR expression, comparison expression, and quantified expression.

- A FLWOR expression consists of the for, let, where, order by, and return clauses. The for, where, order by, and return clauses correspond to the from, where, order by, and select clauses in SQL, respectively. The let clause allows us to substitute an expression with a variable.
- A comparison expression compares the results of two expressions (i.e., \( E_1 \) op \( E_2 \)) and returns true or false. \( E_1 \) and \( E_2 \) produce either a single value or a sequence of values. In contrast to SQL, XQuery allows us to compare two sequences. Given two sequences \( E_1 \) and \( E_2 \), \( E_1 \) op \( E_2 \) is true if there exists at least one pair \((x, y)\) such that \( x \ op \ y \) is true where \( x \in E_1 \) and \( y \in E_2 \). We note that, if \( E_1 \) or \( E_2 \) is a sequence, a comparison expression is equivalent to an existentially quantified expression [25].
- Quantified expressions support existential and universal quantification. The former returns true if at least one value satisfies a given condition, and the latter returns true if all values satisfy a given condition. The keywords some and every indicate existentially and universally quantified expressions, respectively.

We present a few example queries to help understand XQuery. Figure 1 shows three XML documents used throughout this paper. Depts.xml represents the information of departments in a company; Projs.xml that of projects being carried out by the departments; Works.xml that of employees involved in the projects.

Query 1 is a path expression that finds for employees named “John.” Here, a bracket represents a predicate. Query 2 is a FLWOR expression that finds the location of the “Research” department. In Query 2, the for clause binds each Dept element to the variable \( $x \); the where clause checks whether \( $x/DName \) is equal to “Research;” the return clause outputs \( $x/DLocs \) that satisfy the condition in the where clause.

```
QUERY 1: Find employees named “John.”

doc("Depts.xml")/Depts/Dept/Emp[ENAME="John"]
```

```
QUERY 2: Find the location of the “Research” department.

for $x in doc("Depts.xml")/Depts/Dept
where $x/DName = "Research"
return $x/DLoc
```

XQuery allows nested expressions. A nesting (surrounding) FLWOR expression is called an outer FLWOR expression, and a nested (surrounded) one an inner FLWOR expression. Query 3 is an example query that contains a nested expression. In Query 3, the inner FLWOR expression returns the DNO’s of the departments having at least one project, and the outer FLWOR expression the names of those departments.

```
QUERY 3: Find the names of the departments that have at least one project.

for $x in doc("Depts.xml")/Depts/Dept
where $x/DNO = (for $y in doc("Projs.xml")/Projs/Proj
return $y/DNO)
return $x/ENAME
```
Query 4 contains an existentially quantified expression. It finds the names of the departments having at least one project. Here, we note that Query 4 and Query 3 have the same meaning because the right operand of the comparison expression in Query 3 is a sequence. Query 5 contains a universally quantified expression. It finds the names of the departments whose projects are all located in “Houston.”

Query 4: Find the names of the departments that have at least one project.
```xml
for $x$ in doc("Depts.xml")/Depts/Dept
where some $y$ in doc("Projs.xml")/Projs/Proj
satisfies $y/"DNO = "x/"DNO
return $x/"Name"
```

Query 5: Find the names of the departments whose projects are all located in “Houston.”
```xml
for $x$ in doc("Depts.xml")/Depts/Dept
where every $y$ in doc("Projs.xml")/Projs/Proj[
"DNO = "x/"DNO
satisfies $y/"PLoc = "Houston"
return $x/"Name"
```

3. Related Work

Normalization rules are classified into the source-level rule and the algebra-level rule, and we focus on the source-level rule in this paper.

Manolescu et al. [15] have firstly proposed source-level normalization rules for XQuery. Their normalization aims at facilitating the translation of XQuery into SQL. We explain some notations used in Manolescu et al. [15]. For each rule, $x$ and $y$ are variables, and $E_n$ is an expression. $E_n(x)$ means that the variable $x$ appears inside the expression $E_n$.

The normalization rule NR 1 [15] unnests nested expressions in the for clause. It first merges the for clauses of the outer and inner FLWOR expressions, and then, the where clauses of those two FLWOR expressions using the and operator. Next, it replaces the return clause of the outer FLWOR expression with that of the inner FLWOR expression.

NR 1: Unnesting nested expressions in the for clause.
```xml
for $x$ in $E_1$
where $E_2(x)$
return (for $y$ in $E_3(x)$
where $E_4(x, y)$
return $E_5(x, y)$)
```

NR 2: Unnesting nested expressions in the return clause.
```xml
for $x$ in $E_1$
where $E_2(x)$
return $E_3(x, y)$
```

Similar to Manolescu et al., Brantner et al. [2] have proposed source-level normalization rules that unnest FLWOR expressions nested in the for or let clause. However, the rules of Manolescu et al. and those of Brantner et al. are not complete because nested expressions in the where clause and in an aggregate function are not considered.

Lee et al. [12] have proposed a comprehensive set of source-level normalization rules that remove nesting in the for, where, or return clause. They classify nesting types of XQuery using the classification proposed by Kim [11] for SQL: Type-A, Type-N, Type-J, Type-JA, and Type-D. These nesting types are classified depending on the existence of correlation and aggregation in the same manner as in SQL. Definitions 1 and 2 define correlation and aggregation, respectively.

Definition 1: [12] Correlation exists if a variable in the for clause of an outer FLWOR expression appears in the where clause of an inner FLWOR expression.

Definition 2: [12] Aggregation exists if an inner FLWOR expression is used as an argument of aggregate functions.

Based on the definitions above, all the nesting types except Type-D are described as in Table 1.

An XQuery expression of Type-D has two inner FLWOR expressions connected by a comparison operator in the where clause of the outer FLWOR expression. At least one of the inner FLWOR expressions must have correlation. Zero or more inner FLWOR expressions may have aggregation. Thus, Type-D does not belong to any category in Table 1; it falls into either (i) no aggregation and yes correlation or (ii) yes aggregation and yes correlation.

We introduce one normalization rule NR 3 of Lee et al. NR 3 handles Type-JA nesting in the where clause and uses Definitions 3 and 4.

Definition 3: [12] Correlated expressions are two expressions connected by a comparison operator in the where clause of an inner FLWOR expression, where at least one of them contains a variable defined in the for clause of an outer FLWOR expression.

Definition 4: [12] Isolation of an inner FLWOR expression is to remove all of the correlated expressions from the inner FLWOR expression.
NR 3 first isolates the inner FLWOR expression, rendering the correlated expression $E_4(y)$ $op$ $E_5(x)$ removed. Then, it pre-evaluates the aggregated value for each value of $E_4(y)$ since the value of $E_4(y)$ needs to be referenced in the outer FLWOR expression. During this pre-aggregation, we use the group by clause of XQuery 1.1 [26]. Next, NR 3 substitutes the inner FLWOR expression with the pre-evaluated result. Finally, it adds the correlated expression into the where clause of the outer FLWOR expression. This rule extends the NEST-JA rule of Kim [11] so as to accommodate the XQuery language.

NR 3: The normalization rule of Lee et al. [12] for Type-JA nesting.

As discussed in the literature, this rule can incur the count bug [10] and the relations-other-than-equality bug [7]. When these bugs can occur, we exploit existing solutions [3], [7], [17], [21] for correcting the bugs.

The rules of Lee et al. can handle nested FLWOR expressions in the where clause and in an aggregate function. However, they have several problems. They do not handle quantified expressions, incur duplicated query results for Type-N, Type-J, and Type-D nesting, and use many temporary files for Type-D nesting.

4. Normalization Rules for XQuery Expressions

In this section, we propose our source-level normalization rules. Section 4.1 proposes normalization rules for quantified expressions. Section 4.2 for FLWOR expressions.

4.1 Normalization Rules for Quantified Expressions

Normalization rules for existentially quantified expressions consist of two steps. First, we transform existentially quantified expressions (with negation) to FLWOR expressions of Type-A or Type-JA using the transformation rules 1 and 2. Here, TR 1 handles existentially quantified expressions without negation, and TR 2 those with negation. These rules extend the rules originally proposed by Granski and Wong [7] for the SQL language so as to accommodate the XQuery language. Second, we apply the existing normalization rules for Type-A or Type-JA of Lee et al. [12].

**Correctness of TR’s 1 and 2:** Existential quantification means that at least one value satisfies a given condition. That is, if the number of values that satisfy the condition is greater than zero, the result of existential quantification is true. If an expression involves negation, the number of those values should be zero. Thus, by using the count function, existentially quantified expressions (with negation) can be transformed to FLWOR expressions of Type-A or Type-JA.

Normalization rules for universally quantified expressions consist of three steps. First, we transform universally quantified expressions to existentially quantified expressions with negation using the transformation rules TR’s 3 and 4. Here, TR 3 handles universally quantified expressions without negation, and TR 4 those with negation. These rules extend the rule originally proposed by Whang et al. [23] for the two-dimensional database query language so as to accommodate the XQuery language. Second, we apply the transformation rules TR’s 1 and 2 for existentially quantified expressions. Third, we apply normalization rules for Type-A or Type-JA.

**Correctness of TR’s 3 and 4:** TR’s 3 and 4 are basically identical to the following equations:

- $(\forall x)(P(x)) \equiv (\neg (\exists x)(\neg P(x)))$
- $(\neg \forall x)(P(x)) \equiv (\exists x)(\neg P(x))$

---

TR 1: some

\[ \text{some } x \text{ in } E_1 \text{ satisfies } E_2(x) \]

TR 2: not some

\[ \text{not(some } x \text{ in } E_1 \text{ satisfies } E_2(x)) \]

TR 3: every

\[ \text{every } x \text{ in } E_1 \text{ satisfies } E_2(x) \]

TR 4: not every

\[ \text{not(every } x \text{ in } E_1 \text{ satisfies } E_2(x)) \]

---

4.2 Normalization Rules for FLWOR Expressions

The normalization rules of Lee et al. [12] for Type-N, Type-J, and Type-D may incur duplicated query results, and the rule for Type-D uses many temporary files. We first explain the normalization rules of Lee et al. and their problems. We then propose our rules without the problems.

\[ \text{some } x \text{ in } E_1 \text{ satisfies } E_2(x) \]

\[ \text{not(some } x \text{ in } E_1 \text{ satisfies } E_2(x)) \]

\[ 0 < \text{count(for } x \text{ in } E_1 \text{ where } E_2(x) \text{ return } x) \]

\[ 0 = \text{count(for } x \text{ in } E_1 \text{ where } E_2(x) \text{ return } x) \]

---

\[ \text{some } x \text{ in } E_1 \text{ satisfies not(E_2(x))} \]

\[ \text{not(some } x \text{ in } E_1 \text{ satisfies not(E_2(x))}) \]

\[ \text{TR 1: some} \]

\[ \text{TR 2: not some} \]

\[ \text{TR 3: every} \]

\[ \text{TR 4: not every} \]

\[ \text{TR’s 1 } \sim \text{ 4: Transformation rules for quantified expressions.} \]

---

\[^{1}\text{Transformation rules produce the equivalent nested expression to which normalization rules are easily applicable.}\]
4.2.1 Normalization Rule for Type-N

The normalization rule NR 4 of Lee et al. handles Type-N nesting in the where clause. When the comparison operator \( op \) in NR 4 is not equality (\( = \)), the distinct function cannot eliminate duplicated query results. For example, we obtain Query 7 by applying NR 4 to Query 6. Suppose that there is one department whose DNO is ‘D1’, and there are two projects whose DNOs are ‘D2’ and ‘D3’. Then, the results of Query 6 and Query 7 are different. Query 6 produces just one result since comparison involving a sequence is equivalent to existential quantification. In contrast, Query 7 produces duplicated results since it performs join. When a normalization rule incurs duplicated query results, we say that the rule has a duplication problem.

\[
\text{NR 4: The normalization rule of Lee et al. [12] for Type-N nesting.}
\]

\[
\text{Query 6: An example query showing the duplication problem.}
\]

\[
\text{Result 6: The result of Query 6.}
\]

\[
\text{Query 7: Normalize Query 6 using NR 4.}
\]

\[
\text{Result 7: The result of Query 7.}
\]

Our normalization rule for Type-N nesting without the duplication problem consists of three steps. First, we transform nested expressions of Type-N to existentially quantified expressions using the transformation rule TR 5. Second, we transform the existentially quantified expressions to nested expressions of Type-JA using TR 1. Third, we apply normalization rules for Type-JA. There are normalization rules [7], [21] for Type-JA that do not have the duplication problem.

\[
\text{Correctness of TR 5: As explained in Sect. 2, if at least one of the operands of a comparison expression is a sequence, the comparison expression is equivalent to an existentially quantified expression. For more details, please see Sect. 4.5.2 of the Ref. [25].} \]

4.2.2 Normalization Rule for Type-J

The normalization rule NR 5 of Lee et al. [12] handles Type-J nesting in the where clause. Just like Type-N, we present the transformation rule TR 6 to solve the duplication problem that could occur in operators other than equality. Using the combination of TR 1 and TR 6, we can transform nested expressions of Type-J to those of Type-JA, where the duplication problem does not occur.

\[
\text{NR 5: The normalization rule of Lee et al. [12] for Type-J.}
\]

\[
\text{TR 6: A transformation rule for Type-J.}
\]

Correctness of TR 6: See the correctness of TR 5. \]

4.2.3 Normalization Rule for Type-D

The normalization rule NR 6 of Lee et al. [12] handles Type-D nesting in the where clause. NR 6 uses the following notations.

- \( FLWOR_n \) denotes a FLWOR expression.
- \( FLWOR^\text{Isolated}_n \) denotes the inner FLWOR expression obtained by isolating \( FLWOR_n \).
- \( r^\text{Correlated}(S_x) \) denotes the correlated expression of \( FLWOR_n \), which includes the variable \( S_x \). If there is no correlated expression, TRUE is returned.

Since the inner FLWOR expressions \( FLWOR_1 \) and \( FLWOR_2 \) do not have correlation between each other, NR 6 isolates them independently into the temporary files \( T_1 \) and \( T_2 \), respectively. Then, it performs join between \( T_1 \) and \( T_2 \). Here, \( S_{t_1}/result/t1 \) and \( S_{t_2}/result/t2 \) are the return values of \( FLWOR_1 \) and \( FLWOR_2 \), while \( S_{t_1}/result/t1 \) and \( S_{t_2}/result/t2 \) are the values used in the correlated expressions. Next, NR 6 binds the result of this join to
the variable \( T_3 \) and remove duplicates. Finally, it replaces \( \text{FLWOR}_1 \ op \ \text{FLWOR}_2 \) in the \( \text{where} \) clause of the outer \( \text{FLWOR} \) expression with the correlated expressions of \( \text{FLWOR}_1 \) and \( \text{FLWOR}_2 \) (i.e., \( E_1^{\text{Correlated}}(S_{t_3}/\text{result}/t_1) \) and \( E_2^{\text{Correlated}}(S_{t_3}/\text{result}/t_2) \)).

\[
\begin{align*}
\text{let } T_1 := & \text{FLWOR}^{\text{Isolated}}_1 \\
\text{let } T_2 := & \text{FLWOR}^{\text{Isolated}}_2 \\
\text{let } T_3 := & \text{distinct(for } S_{t_3} \text{ in } T_1, S_{t_3} \text{ in } T_3 \text{ where } S_{t_3}/\text{result}/t_1 \ op \ S_{t_3}/\text{result}/t_1 \\
& \text{return } \langle \text{result} \rangle: <1 \rangle \langle S_{t_3}/\text{result}/t_2 \rangle <1 \rangle \langle S_{t_3}/\text{result}/t_2 \rangle <1 \rangle \langle \text{result} \rangle \rangle \\
\text{for } S_{x} \text{ in } E_1, \ S_{t_3} \text{ in } T_3 \\
& \text{where } E_1^{\text{Correlated}}(S_{t_3}/\text{result}/t_1) \text{ and } E_2^{\text{Correlated}}(S_{t_3}/\text{result}/t_2) \\
& \text{return } E_3(x) \end{align*}
\]

In the relational context, Kim [11] has compared the cost of processing a nested query using the nested-loop join method and the cost of processing an unnested query using the merge-join method. For the join of two relations, the nested-loop join method requires the inner relation to be retrieved as many times as the number of tuples that satisfy predicates on the outer relation, but the merge-join method

<table>
<thead>
<tr>
<th>FLWOR</th>
<th>Type-A</th>
<th>Type-N</th>
<th>Type-J</th>
<th>Type-JA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type-A</td>
<td>N/A</td>
<td>N/A</td>
<td>TR 9</td>
<td>TR 12</td>
</tr>
<tr>
<td>Type-N</td>
<td>N/A</td>
<td>N/A</td>
<td>TR 7</td>
<td>TR 10</td>
</tr>
<tr>
<td>Type-J</td>
<td>TR 9</td>
<td>TR 7</td>
<td>TR 8</td>
<td>TR 11</td>
</tr>
<tr>
<td>Type-JA</td>
<td>TR 12</td>
<td>TR 10</td>
<td>TR 11</td>
<td>TR 13</td>
</tr>
</tbody>
</table>

**Table 2** All possible cases of Type-D.

To solve those problems, we transform nested expressions of Type-D to those of Type-A, N, J, or JA and apply our normalization rules. After normalization, the number of temporary files used in our rules for Type-D is one or two while that of Lee et al. is three.

The transformation needs to be performed depending on the subcategory of Type-D. Table 2 summarizes all possible cases of Type-D classified by the nesting types of two inner FLWOR expressions. The cases where the two inner FLWOR expressions have no correlation (i.e., the combinations of Type-A and Type-N) are not available by definition. Nested expressions of Type-D can be classified into three categories: (1) none of the inner FLWOR expressions has aggregation; (2) only one of the inner FLWOR expressions has aggregation; (3) both of the inner FLWOR expressions have aggregation. We call these categories Type-D-A0, Type-D-A1, and Type-D-A2, respectively.

Normalization rules for Type-D-A0 consist of three steps. First, we transform nested expressions of Type-D-A0 to existentially quantified expressions using the transformation rules TR’s 7 and 8. Second, we apply transformation rules for existentially quantified expressions. Third, we apply normalization rules for Type-JA. After normalization, the number of temporary files used is one.

**Correctness of TR’s 7 and 8:** See the correctness of TR 5.

Normalization rules for Type-D-A1 consist of two steps. First, we transform nested expressions of Type-D-A1 to those of Type-N or Type-J using the transformation rules TR’s 9 ~ 11. In these rules, we pre-evaluate the inner FLWOR expression having aggregation. Second, we apply normalization rules for Type-N or Type-J. After normalization, the number of temporary files used is two.

**Correctness of TR’s 9 ~ 13:** Since the inner FLWOR expressions do not have correlation each other, they can be normalized separately.

**5. Analysis on the Effect of Query Normalization**

A nested query limits the query optimizer’s choices because the nesting imposes a partial join order and only a nested-loop join method can be used [18]. Query normalization transforms a nested query to a logically equivalent unnested query, and the unnested query could then be examined by a query optimizer for alternative plans, including different join methods [7].

In the relational context, Kim [11] has compared the cost of processing a nested query using the nested-loop join method and the cost of processing an unnested query using the merge-join method. For the join of two relations, the nested-loop join method requires the inner relation to be retrieved as many times as the number of tuples that satisfy predicates on the outer relation, but the merge-join method

**TR 7:** Type-N and Type-J

**TR 8:** Type-J and Type-J

\[
\begin{align*}
\text{for } S_{x} \text{ in } E_1 \\
& \text{where (for } S_{y} \text{ in } E_2, \ S_{y} \text{ in } E_4 \text{ where } E_3(S_{y}) \ op \ E_5(S_{x}) \text{ return } E_6(S_{x}) \text{)} \\
& \text{for some } S_{y} \text{ in } E_3, \ S_{z} \text{ in } E_4 \text{ where } E_3(S_{y}) \ op \ E_5(S_{z}) \text{ return } E_6(S_{z}) \\
& \text{for } S_{x} \text{ in } E_1 \\
& \text{where (for } S_{y} \text{ in } E_2, \ S_{x} \text{ in } E_4 \text{ where } E_3(S_{x}) \ op \ E_5(S_{z}) \text{ return } E_6(S_{z}) \text{)} \\
& \text{for some } S_{y} \text{ in } E_3, \ S_{x} \text{ in } E_4 \text{ where } E_3(S_{x}) \ op \ E_5(S_{y}) \text{ return } E_6(S_{y}) \\
& \text{for } S_{x} \text{ in } E_1 \\
& \text{where (for } S_{z} \text{ in } E_2, \ S_{x} \text{ in } E_4 \text{ where } E_3(S_{z}) \ op \ E_5(S_{x}) \text{ return } E_6(S_{x}) \text{)} \\
& \text{for some } S_{z} \text{ in } E_3, \ S_{x} \text{ in } E_4 \text{ where } E_3(S_{x}) \ op \ E_5(S_{z}) \text{ return } E_6(S_{z}) \\
\end{align*}
\]
requires both relations to be sorted in join-column order and scanned only once [11]. He shows the performance improvement attainable using his normalization rules with examples of queries and database conditions.

The same analysis can be applied for XQuery normalization. Table 3 shows the notations used in the analysis.

Due to space limit, we only present the analysis for a Type-JA nesting. For other nesting types, please refer to the analysis of Kim [11]. Processing a nested query using the nested-loop join method costs up to Formula (1)\(^\dagger\). In contrast, processing an unnested query using the merge-join method costs up to Formula (2). In Formula (2), the first two terms are the cost of generating a temporary tuple stream \(R_t\) by evaluating a query with \texttt{group by} clause, which needs to sort \(R_j\). The next term is the cost of sorting \(R_i\), and the last two terms are the cost of merging \(R_i\) and \(R_t\). We note that we do not need to sort \(R_t\) to merge-join since \(R_t\) is already sorted in the order of the grouping key, which is the join key. We note that the cost formulas may be modified according to the implementation of the underlying XQuery engine.

\[ \begin{align*}
    P_i + N_j P_j \\
    2P_j \cdot \log_{B-1} P_j + R_i + 2P_j \cdot \log_{B-1} P_i + P_i + P_j
\end{align*} \]

\textbf{Example 1:} Let \(P_i = P_j = 128\), \(P_i = 16\), \(B = 3\), and \(N_i = \) \(\dagger\)For the case where there is an index on the join key, please refer to the analysis of Kim [11].
1000. The nested-loop join method of processing $Q_{\text{nested}}$ costs $128 + 1000 \cdot 128 = 128,128$ page fetches. If a two-way merge-sort technique is used, the merge-join method of processing $Q_{\text{unnested}}$ costs $2 \cdot 128 \cdot 7 + 16 + 2 \cdot 128 \cdot 7 + 128 + 16 = 3744$ page fetches. Thus, query normalization improves query performance by 34.22 times. □

According to Formulas (1) and (2), the performance improvement becomes more marked as the database size gets larger. For example, Example 2 uses a ten times larger database than Example 1, and the improvement increases from 34.22 to 235.15 times.

Example 2: Let $P_1 = P_j = 1280$, $P_1 = 160$, $B = 3$, and $N_i = 10,000$. The nested-loop join method of processing $Q_{\text{nested}}$ costs $1280 + 10,000 \cdot 1280 = 1280,1280$ page fetches. If a two-way merge-sort technique is used, the merge-join method of processing $Q_{\text{unnested}}$ costs about $2 \cdot 1280 \cdot 10.32 + 160 + 2 \cdot 1280 \cdot 10.32 + 1280 + 16 = 54,438.40$ page fetches. Thus, query normalization improves query performance by 235.15 times. □

We note that the nested-loop join method will tend to be more efficient than the merge-join method if $N_i$ is very small (around $2 \log_B P_j$), so that the nested-loop join method will not require $R_j$ to be retrieved as many times as it is required to sort $R_j$. However, this represents a very small subset of the set of all possible query and database characteristics [11]. We need a sophisticated query optimizer that can decide whether normalization is beneficial, but it is beyond the scope of this paper.

6. Performance Evaluation

6.1 Experimental Setup

To verify performance improvements of query normalization, we measure query execution times for a nested query and an unnested query for Type-A, Type-N, Type-J, Type-JA, Type-D, and quantified expressions. We also measure the time for normalizing a nested query. All results are averaged over 10 runs.

We perform experiments using XQuery engines that fulfill the following criteria: (i) supporting the group by clause of XQuery 1.1; (ii) free public availability. While there might be more XQuery engines fulfilling these criteria, we limit our evaluation to those in Table 4.

The first two are stand-alone (file-based) XQuery engines, the last one is an XML/XQuery database system.

We use the TPC-H benchmark database [22] of 100 MBytes and 1 GBytes. The data are represented as the XML syntax. Figure 2 shows part of the TPC-H database schema. The database contains information of parts and suppliers. We execute test queries against this schema. For the list of queries, please refer to Appendix.

Our experimentation platform is a 2.9 GHz AMD Athlon II X2 245 (2 MB L2 cache) dual-core processor with 4 GBytes of main memory and a Samsung HD501LJ disk (500 GBytes, 7200 RPM, 16 MB Cache, and SATA 3.0 Gbps interface). The operating system is Microsoft Windows 7 64 bit. We implemented all the normalization rules using C++.

6.2 Experimental Results

Table 5 shows the performance comparison for nested FLWOR and quantified expressions. $T_{\text{nested}}$ and $T_{\text{unnested}}$ are the query execution times for a nested query and an unnested query, respectively. N/A means that the engine fails to execute the nested or unnested query. For example, the unnested query for Type-J requires two grouping keys, but Saxon currently supports only one grouping key.

The results show that the unnested query often outperforms the nested one by up to several orders of magnitude. Furthermore, the performance improvement becomes more marked as the database size gets larger. For example, for the Type-J query using Zorba, when the database size increases from 100 MBytes to 1 GBytes, the improvement increases from 1,043.64 to 7,172.39 times. The main reason for performance improvement is that repeated execution of nested expressions can be avoided by normalization. These results agree with the analysis results in Sect. 5 and indeed demonstrate that our normalization rules are effective for a variety of XQuery engines.

For some cases like Type-A, $T_{\text{nested}}$ is almost the same as (or slightly smaller than) $T_{\text{unnested}}$, but the difference is at most one percent of the query execution time. We think that this is because the underlying XQuery engines already optimize the queries internally.

For all the queries, the normalization time takes only less than one millisecond, which is ignorable compared to the query execution time.

7. Conclusions

In this paper, we proposed new source-level normalization rules. First, we provided normalization rules for quantified expressions, thus significantly improving the power of the
normalization rules. We support both existentially and universally quantified expressions. Second, we proposed normalization rules for Type-N, Type-J, and Type-D that do not incur duplicated query results. Third, we proposed normalization rules for Type-D that use fewer temporary files than existing methods. Through analysis and experiments, we showed that our normalization rules reduce query execution time by up to several orders of magnitude for a variety of XQuery engines, and the performance improvement becomes more marked as the database size gets larger.

Acknowledgements

This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology (2010-0015977).

References


### Appendix: The Queries Used in the Experiments

#### Type-A

<table>
<thead>
<tr>
<th>Query</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>for $p$ in /TPCH/Parts/Part where $p$/size = max($p$/1 in /TPCH/Parts/Part where $p$/1/retailPrice &gt; 1000 return $p$/1/size) return $p$/partKey</td>
<td></td>
</tr>
</tbody>
</table>

#### Type-N

<table>
<thead>
<tr>
<th>Query</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>for $s$ in /TPCH/Suppliers/Supplier where $s$/suppKey = ($s$/suppCost &gt; 990 return $s$/suppKey) return $s$/suppKey</td>
<td></td>
</tr>
</tbody>
</table>

#### Type-J

<table>
<thead>
<tr>
<th>Query</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>for $s$ in /TPCH/Suppliers/Supplier where $s$/accetbal = ($s$/suppCost &gt; 990 return $s$/suppKey) return $s$/suppKey</td>
<td></td>
</tr>
</tbody>
</table>

#### Type-JA

<table>
<thead>
<tr>
<th>Query</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>for $p$ in /TPCH/Parts/Part where $p$/retailPrice = max($p$/1 in /TPCH/PartsSupps/PartSupp where $p$/1/availQty &lt; 2 and $p$/partKey = $p$/partKey return $p$/supplyCost) return $p$/partKey</td>
<td></td>
</tr>
</tbody>
</table>

#### Type-D

<table>
<thead>
<tr>
<th>Query</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>for $s$ in /TPCH/PartSupps/PartSupp where ($s$/suppKey = $s$/suppKey return $s$/accetbal = ($s$/suppCost &gt; 990 return $s$/suppKey) return $s$/partKey</td>
<td></td>
</tr>
</tbody>
</table>

#### Existential Quantification

<table>
<thead>
<tr>
<th>Query</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>for $s$ in /TPCH/Suppliers/Supplier where some $s$/suppKey in /TPCH/PartSupps/PartSupp[supppKey = $s$/suppKey] satisfies $s$/availQty &lt; 2 return $s$/suppKey</td>
<td></td>
</tr>
</tbody>
</table>

#### Universal Quantification

<table>
<thead>
<tr>
<th>Query</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>for $s$ in /TPCH/Suppliers/Supplier where every $s$/suppKey in /TPCH/PartSupps/PartSupp[supppKey = $s$/suppKey] satisfies $s$/availQty &lt; 9999 return $s$/suppKey</td>
<td></td>
</tr>
</tbody>
</table>

Ki-Hoon Lee received B.S. (2000), M.S. (2002), and Ph.D. (2009) degrees in Computer Science from Korea Advanced Institute of Science and Technology (KAIST). He is currently a research engineer in Korea Telecom (KT). His research interests include XML and web databases, IR and search engines, query optimization, object-relational database systems, and spatial databases and GIS.

Young-Ho Park received his Ph.D. degree in Department of Computer Science from the Korea Advanced Institute of Science and Technology (KAIST) in 2005. He had worked for the Electronics and Telecommunication Research Institute (ETRI) as a senior research staff at the ISDN Administration and Maintenance Division for TDX-10 ISDN, the Real-Time DBMS Division and the Real-Time Operating System Division from 1993-1999. And, he had worked for the Advanced Information Technology Research Center (AITrc), Korea Advanced Institute of Science and Technology (KAIST) as a Post Doctor at the from 2005-2006 after receiving Ph.D. degree. Currently, he is an Associate Professor of the Multimedia Science at SooMyung Women’s University, Seoul, Korea. His research interests include Database Management Systems, Information Retrieval, XML, and Mobile Platform.