Isolating Order Semantics in Order-Sensitive XQuery-to-SQL Translation

Song Wang, Ling Wang and Elke A. Rundensteiner

Worcester Polytechnic Institute, Worcester, MA 01609, USA
(songwang|lingw|rundenst)@cs.wpi.edu

Abstract. Order is essential for XML query processing. Efficient XML processing with order consideration over relational storage is non-trivial, especially for complex nested XQuery expressions. The order semantics may impede efficient query rewriting for nested query blocks. We propose a general order-sensitive XQuery processing approach involving three steps. First an algorithm is proposed for inferencing about and then isolating the order semantics in XQuery expressions specified over virtual XML views. This turns an *ordered* XQuery plan into an *unordered* one decorated with minimized *order context annotations*. Then without loss of semantics, logical optimization via XQuery rewriting can be easily applied to this transformed query plan. As last step, the translation of the optimized logical plan into SQL now correctly incorporates the order context annotations to assure the original order semantics. Our experiments illustrate the performance gains achievable by our order handling strategy.

1 Introduction

Since XQuery semantics are order sensitive, order awareness has been identified as critical for XQuery processors. Order-sensitive XML query processing has been studied for native XQuery engines, such as TIMBER [10], Natix [6] and Rainbow [1, 20]. There has also been considerable work on extending relational query engines to process XQueries over XML documents. See [5] for a survey.

Several aspects of supporting order-sensitive XQuery processing over relational storage have been successfully tackled in the literature. XML document order encoding strategies during XML loading, such as Dewey order [15], ORDPATH [9], and preorder ranks [3], have been proposed. The order-sensitive XPath to SQL translation has been studied for these different order encodings [3, 9, 15]. However, the order semantics in general impede efficient query rewriting for nested query plan blocks. We thus propose a general order-sensitive XQuery processing approach which overcomes this problem. Our solution does not rely on any specific relational order-encoding.

Our work relates to recent work on the order processing and duplication removal of matched pattern trees for the native XQuery engine Timber [10, 11]. The authors propose to use hybrid collections of matched pattern trees to capture the order semantics of XQuery expressions. Although the proposed techniques are sufficient for native XQuery processing, their adoption to an XQuery engine with relational storage faces new challenges, as shown below.

Motivation Example 1: XML nodes matched by XQuery expressions can be heterogeneous, due to the wildcard "*" navigation step in XPath expressions or the *union* operations in the *For* and *Let* clauses. In Figure 1(a), four XML nodes ($b0$ to $b3$) match the given XPath expression for $\$b$. The intermediate result of the translated SQL thus contains four tuples[1]. We observe that every XML node in the matched pattern is represented in a column in the result table shown at the bottom of the figure. No simple sorting on any individual column can achieve the ordering of the *order by* clause, no matter what order encoding is used. Instead, we have to build an extra ordering column to record the order information corresponding to the runtime execution.

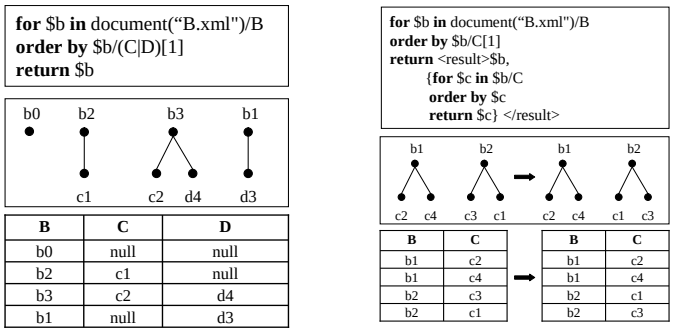


Fig. 1. Motivation Examples: (a) XQuery with Heterogeneous Matched XML Nodes; (b) Sorting in Nested XQuery Expressions

Motivation Example 2: We may need to group and sort the results of the translated SQL queries to achieve correct semantics of nested XQuery expressions. Adding simple sortings into the SQL may not be adequate. As shown in Figure 1(b), the XQuery expressions first sort the "$b" bindings by the first "C" child, and then all "C" children of each "$b" binding. The XML nodes and corresponding tuples on left show the orderings for the outer XQuery expression, while the right ones show the effect of the inner XQuery expression. Obviously we cannot achieve the correct ordering by a simple translated SQL *order by* clause.

The ordering problems shown above are unique to XQuery processing on relational XML views where order can only appear at the top-level of an SQL query. The naive approach to guarantee the correct order processing in the SQL translation is to build an extra order column for each level of the result construction. Such columns are used to capture the runtime order semantics in each of the intermediate results. Those columns can be combined and treated as Dewey order of the result XML. Then a sort at the top-level SQL query can achieve the correct ordering of the tuples. The OLAP amendment *row_number()* with *partition by* and optional *order by* clauses in SQL99 can be used to achieve this [17]. However, such operations tend to be expensive. In fact in many cases they may be redundant, since ordering at all levels of the nesting may not be required.

[1] For ease of illustration, b_i, c_i and d_i are used for both the XML nodes and the atomized values.

Our approach tackles the above open problem. We propose a general framework to process XQuery expressions on virtual XML views of relational databases. Our approach first isolates the order semantics in the combined query composed of the user XQuery and the XML view query. Then by identifying and reasoning about essential order information, we minimize the usage of the expensive *row_number()* functions. As result we produce a succinct SQL translation of order-sensitive XQuery expressions. Correctness of the translated SQL is achieved by "attaching" back the order semantics identified as "essential".

Contributions. Our contributions include: (1) We propose a general framework for processing order-sensitive XQuery over virtual XML views defined on relational databases. (2) We introduce order propagation techniques for the XQuery algebra [20] to support order isolation in the XQuery plan. (3) We discuss the strategies for SQL translation with ordered semantics. (4) We implement the order propagation and isolation approach in the Rainbow XQuery Engine [20]. (5) We report experimental studies illustrating the tradeoff among different SQL translation strategies.

Outline. The rest of this paper is organized as follows. Section 2 reviews related work, while Section 3 shows preliminaries. Section 4 enhances the XAT algebra with order context annotations. Section 5 describes order propagation and order isolation for order-sensitive XQueries. Different SQL generation strategies are presented in Section 6. Section 7 provides our experimental study and Section 8 concludes this paper.

2 Related Work

XQuery-to-SQL translation can be broadly classified into two scenarios: *XML Publishing* and *XML Storage* [5]. Since the relational data model is unordered, XML Publishing of relational data need not consider order semantics [2, 12]. For XML storage of existing XML data, various order encoding methods have been proposed in [15, 17].

Order inference has been used for relational query optimization in [13, 19, 14, 8, 7] to reason about the physical tuple order of intermediate results during execution. We now use similar ideas for the new purpose of logical order inference in virtual XML views focusing in addition on the hierarchical XML data.

In [4], removing unnecessary duplication elimination and sorting operations in Galax is discussed for processing complex XPath expressions with backwards axes. This is orthogonal to our problem. However the approach is complimentary and could fit into our framework for the purpose of syntax level normalization.

[11] addresses order-sensitive XQuery processing in a pattern tree based native XQuery engine. Instead different optimization opportunities exist for order-sensitive XQuery processing over relational XML views, as illustrated by the motivation examples in Section 1.

3 Preliminaries

3.1 The XQuery Subset

In this paper, we deploy a subset of the XQuery language [16], including nested *FLWOR* expressions and order-sensitive functions (e.g., the position function). With syntax rewriting, such XQuery subset covers a large set of XQuery expressions used in practice. Formal definition of the XQuery subset is in [22].

3.2 A Running Example of Order-Sensitive XQuery Processing

Given a relational database, a view query defines an XML view over the relational database bridged through the *default XML view* [12]. We assume that the relational constraints are:

```
PLAY(IID,NAME): Primary Key(IID), Unique(NAME)
MPLAY(IID,PID,NAME): Primary Key(IID), Unique(NAME)
SONG(IID,PID,NAME): Primary Key(IID), Unique(NAME)
```

Figure 2(b) shows the default XML view query, containing explicit *order by* clauses. Such an XML view is ordered. Figure 2(c) shows the simplified schema tree (with "BAND" omitted) of this view, highlighting the order semantics. Edges are marked as "Unordered" whenever the view query does not impose any *order by* clauses there. We call such an XML view "partially ordered". Figure 2(d) shows one "possible" view result. That is, the order among the play elements can be different from that in Figure 2(d).

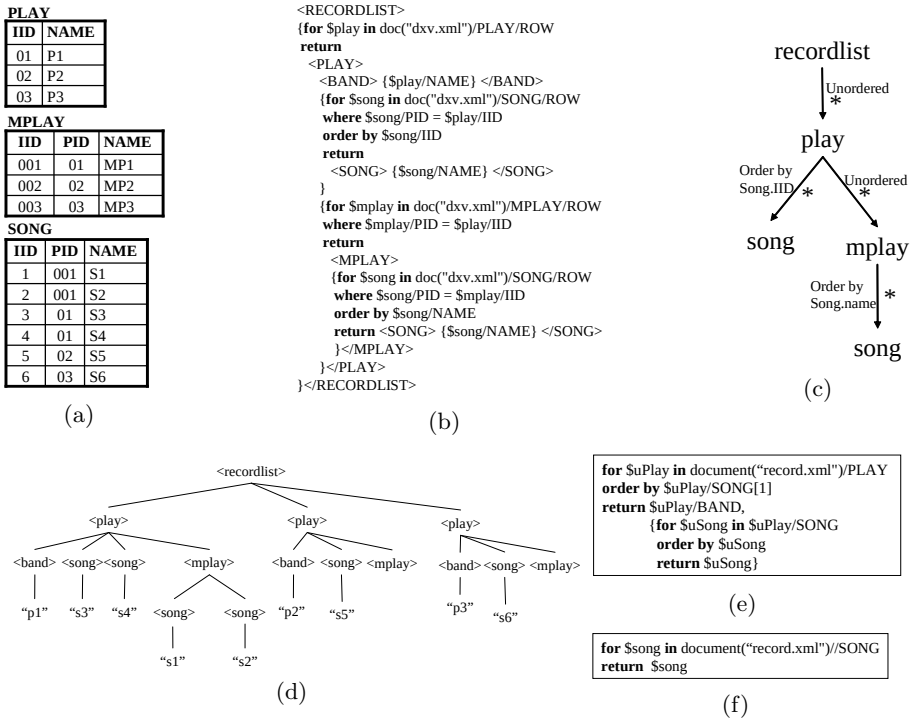


Fig. 2. (a) Relational Tables; (b) Default XML View Query; (c) XML View Schema with Order Information; (d) Order-sensitive XML View; (e) Q1: XQuery with Nested Orderings; (f) Q2: XQuery with Heterogeneous Matched Patterns.

Order-sensitive user XQueries can be launched over the default XML view. Figure 2(e) shows a user query using position function and nested orderings. Figure 2(f) shows another user query with a complex XPath containing "//". Since the XPath expression matches multiple paths in the XML view, this complicates the order of the retrieved SONG elements. We will use these two XQueries to show our approach of order-sensitive SQL translation.

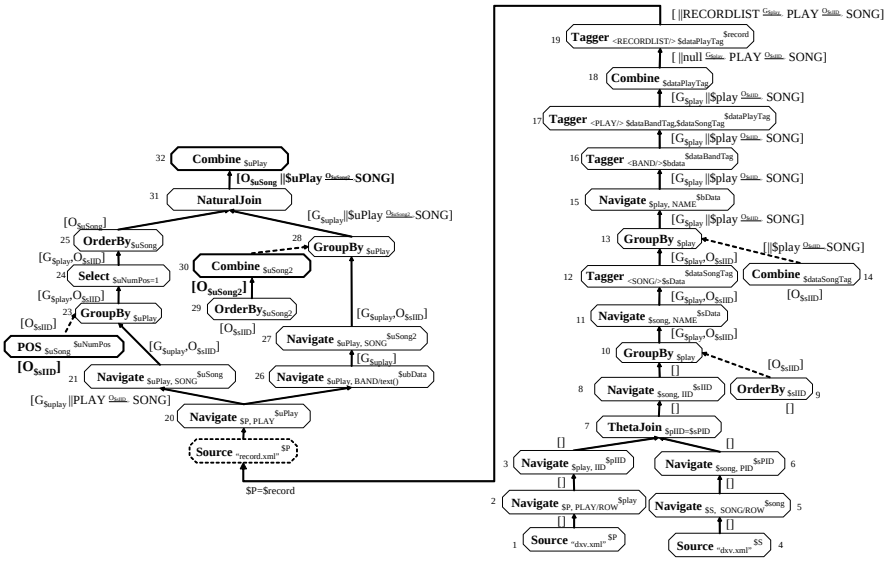


Fig. 3. Composed XAT of View Query and User Query $Q1$

3.3 The XQuery Algebra: XAT

Our approach uses the XAT algebra [20] as internal representation of the view query, user query and their composition. A complete discussion of the XAT algebra is in [22]. The intermediate results of an XAT operator is a sequence of tuples, named *XATTable*. An XAT operator is denoted as $op_{in}^{out}(R)$, where *op* is the operator symbol, *in* represents input parameters, *out* newly produced output column and R the input XATTable(s). Figure 3 shows the composition of the decorrelated XAT trees capturing the user query $Q1$ and the view query.

The XAT algebra inherits operators from the relational algebra, extended with order semantics (see Section 4.1). For example, the **GroupBy** operator is generated during XAT decorrelation of nested XQuery expressions [18]. The GroupBy operator groups the tuples of the input XATTable by certain column(s), and then performs the embedded functions on each group of tuples. For example, the $GroupBy_{\$uPlay}$ embedded with $POS_{\$uSong}$ (Nodes 22 and 23 in Figure 3) will change the input: $< (p_1, s_3), (p_1, s_4), (p_2, s_5) >$ into output: $< (p_1, s_3, 1), (p_1, s_4, 2), (p_2, s_5, 1) >$. The XAT algebra also introduces new operators to represent XQuery semantics, such as Navigate, Tagger, Combine and POS. The **Navigate** operator extracts destination XML nodes from each entry of XML fragments according to a given XPath expression. Assume the input column $\$uPlay$ of Node 21 in Figure 3 includes two plays p_1 and p_2, which have songs s_3, s_4 and s_5 respectively; the output of the Navigate operator is a sequence of tuples: $< (p_1, s_3), (p_1, s_4), (p_2, s_5) >$. The **Tagger** operator constructs new XML elements by applying a tagging pattern to each tuple in the input XATTable. A tagging pattern (e.g., $<SONG/> \$sData$ in Node 12 of Figure 3) is a template for the XML fragment. The **Combine** operator projects out certain columns from the input XATTable and merges the contents of the remaining columns into one tuple. The **POS** operator captures the semantics of the *position()* function. It assigns a row number to each input tuple.

4 Enhancing XAT with Order Context

4.1 Order Context for XATTable

The *order context* in [13, 19] represents the tuple order of flat relational tables. We extend the *order context* to represent tuple order and XML fragment order in XAT tables. This extension is essential because: (i) hierarchical XML views are defined with multiple level sortings; and (ii) XML view queries define only partial orders in XML views. Figure 2(c) shows a partial order example which captures the order information in a schema graph. The PLAY and MPLAY nodes are not ordered, while SONG nodes are ordered according to different columns.

The order context of an XATTable is composed of two parts, denoted as $[TupleOrder||XMLOrder]$. $TupleOrder$ captures the tuple ordering and grouping properties of an XATTable, while the $XMLOrder$ captures the document order for an XML fragment. Both parts can be optional. $TupleOrder$ is a sequence of order properties: P_i. Each P_i can be either an ordering denoted as $O_{\$c}$ or a grouping denoted as $G_{\$c}$, where $\$c$ is a column of the XATTable (the grouping could be on multiple columns). The tuples in the XATTable are ordered (or grouped) first according to P_1, with ties broken by P_2, and so on. For each P_i, $O_{\$c}$ implies $G_{\$c}$, but not vice versa. The semantics of grouping on multiple columns $G_{\$c_i,\$c_j}$ are not equal to $G_{\$c_i}$ followed by $G_{\$c_j}$. The order context $[G_{\$c_i}, G_{\$c_j}]$ implies the order context $[G_{\$c_i,\$c_j}]$, but not vice versa. $XMLOrder$ is attached to the schema tree of the associated XML fragments. For example, the $XMLOrder$ of the XATTable after the Tagger operator (Node 19) is shown in Figure 4[2].

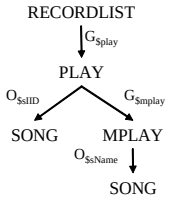


Fig. 4. The $XMLOrder$ in the output XATTable of Node 19.

4.2 Functional Dependencies and Keys

We use the functional dependencies of the base relational tables to propagate the order context through the XAT tree. The constraints of an XATTable can be determined utilizing rules similar to those in [13, 19]. We omit the details here due to space limitation. We use the constraints for the following purposes:

- Minimize the order context by removing redundant orderings (groupings).
- Retrieve trivial orderings and groupings if needed during order propagation. "Trivial" [13] implies that it can be omitted. For example, a key constraint implies a trivial grouping on the key column(s).
- Check the compatibility of order contexts.

[2] For ease of illustration, the XAT tree and order contexts shown in Figure 4 include only part of the XML view query.

5 Order Propagation and Isolation

The identification and isolation of the order semantics of the XAT tree is accomplished in two traversals. First the bottom-up traversal (complete order propagation) computes the order contexts of all intermediate XATTables. Second the top-down traversal (selective order isolation) identifies the operators that indeed require the order context to produce correct results, i.e., the essential ones.

5.1 Order Context Propagation

The order contexts of XATTables originate from explicit sorting in the XML view query. They then are propagated through the operators in the XAT tree to form the ordered XML view and the ordered user XQuery result. We call the procedure of determining the order contexts of the XATTables *Order Propagation*. Figure 3 illustrates the propagation of the order context through the composed XAT tree. The order context is associated with each XATTable and attached to edges between operators. During the propagation, the $XMLOrder$ part of the input order context will always be carried on to the output, except when the corresponding XML fragments are navigated into or are projected out. The propagation of $TupleOrder$ of the XATTable depends not only on the operator semantics but also the constraints in the XATTable.

Select, Project and Tagger. The $TupleOrder$ in the output XATTable of most unary operators, such as Select, Project and Tagger, inherits the $TupleOrder$ of the input XATTable. If one column in the $TupleOrder$ of the input XATTable is projected out, it is also removed from the output order context.

Join. Suppose OC_L and OC_R denote the order contexts of the left and right input XATTables of a Join operator. Then the $TupleOrder$ of the output order context inherits the $TupleOrder$ in OC_L. The $TupleOrder$ of OC_R is attached to the output order context if the $TupleOrder$ of the OC_L is not empty. Otherwise, the $TupleOrder$ of OC_R is discarded.

Here all ordering and grouping properties in the left input XATTable, even if trivial, need to be included in OC_L for the empty test and order propagation. For example, suppose the left input XATTable has a unique identifier (c_1, c_2) (i.e. key constraint), then G_{c_1,c_2} is trivial since all groups consist of only one tuple. But it is no longer trivial in the Join output since a 1 to m joining between the left and right input tuples may exist.

OrderBy. An OrderBy operator sorting on $c_1, c_2, ...$ will generate a new order context $[O_{c_1}, O_{c_2}, ...]$. The propagation of the order context associated with its input XATTable through the OrderBy operator is determined by the compatibility of the order contexts. For example, $[G_{c_1}, G_{c_2}]$ is not compatible with the explicit sorting on c_2. Thus the explicit sorting overwrites the output order context as $[O_{c_2}]$ only. But $[G_{c_1}, G_{c_2}]$ is compatible with ordering on (c_1) or on (c_1, c_2, c_3) with the output order context then being set to $[O_{c_1}, G_{c_2}]$ and $[O_{c_1}, O_{c_2}, O_{c_3}]$ respectively. In Figure 3 the OrderBy operator (Node 25) sorts by $\$uSong$. The input order context $[G_{\$play}]$ is compatible with this sorting, since $G_{\$play}$ is implied by $O_{\$uSong}$ due to the selection (Node 24).

GroupBy. Similar with OrderBy, the propagation of $TupleOrder$ through the GroupBy is also determined by the compatibility between the input order context and the generated order context. For example, if the input tuples have been sorted on column $\$c_1$ and the grouping is done on column $\$c_2$, where $\$c_2$ is a

key column, then the output order context of the GroupBy operator is $[O_{\$c_1}]$, with $G_{\$c_2}$ being a trivial grouping. For example in Figure 3 the generated order context $G_{\$uPlay}$ of the GroupBy operator (Node 23) is compatible with the input order context $[G_{\$uPlay}, O_{\$sIID}]$, since $\$uPlay$ is a key.

Navigate. The Navigate operator passes the $TupleOrder$ of its input order context to its output order context. If the input $TupleOrder$ including the trivial groupings (if any) is not empty, the extracted order from the XML fragment will be attached to the end of the input $TupleOrder$. Otherwise the output $TupleOrder$ is empty. Different permutations of the same set of Navigates may result in different order contexts. For example, consider two Navigate operators $\$a/b$ and $\$a/c$. If we perform $\$a/b$ before $\$a/c$, then the final order context will be $[O_{\$a}, O_{\$b}, O_{\$c}]$. If we perform the two Navigates in the opposite order, then the output tuple order will be $[O_{\$a}, O_{\$c}, O_{\$b}]$. This illustrates the limitation of handling order using query plan rewriting. Our effort of order isolation in Section 5.2 is thus necessary. The output order context of the Navigate operator includes the $XMLOrder$ extracted from its input order context. For example in Figure 3 the Navigate operator (Node 21) extracts the $XMLOrder$: $(PLAY \xrightarrow{O_{\$sIID}} SONG)$ from the XML fragment. The output order context is $[G_{\$uPlay}, O_{\$sIID}]$.

Combine. The Combine operator forms the $XMLOrder$ in the output order context. In case that Combine is embedded in a GroupBy operator, the formed $XMLOrder$ will use the grouping column(s) as the relative column(s). If Combine is not in a GroupBy operator, *null* will be used for the relative column. For example, in Figure 3, the Combine operator (Node 13) forms: $\$play \xrightarrow{O_{\$sIID}} SONG$.

5.2 Isolating Ordered Semantics in XAT Tree

In the query plan of the user XQuery, if the semantics of an operator are defined based on the tuple order of its input XATTable, we classify this operator as an **order essential operator**. The order context associated with the input XATTable is called an **essential order context**.

In a top-down traversal of the XAT tree representing the user XQuery, we now identify all order-essential operators and bind them with their input order context for possible relocation in future rewriting steps. This denotes the *Order Isolation* phase. In Figure 3, the Combine operator (Node 30), which originates from the user XQuery, is an order-essential operator, since all tuples must be sorted correctly before being "packed" into a collection in the result XML. All operators capturing order sensitive functions are also order-essential operators, e.g., the POS operator (Node 22). An XAT tree, with all OrderBy operators removed and essential order contexts attached to the associated operators, is called an **Order Annotated XAT Tree**.

Intuitively the order-essential operators determine the only positions in the query tree where the order context has to be enforced. By enforcing the essential order contexts, the ordered semantics of XQuery are captured. The XAT tree can be optimized now ignoring all OrderBy operators during the subsequent rewriting phase. After applying order-insensitive XQuery rewriting rules, the correct order semantics is restored by inserting explicit sorts below each order essential operator. The optimized annotated XAT tree is shown in Figure 5.

Details of the XAT rewriting are beyond the scope of this paper and can be found in [21].

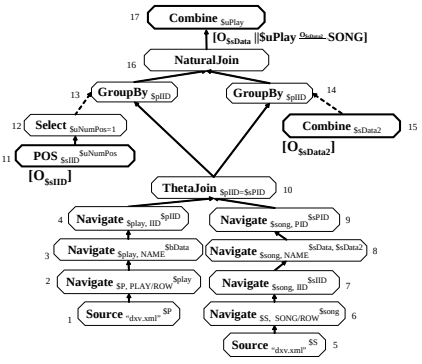


Fig. 5. Optimized XAT Tree

```
WITH Q1 (bData, pIID, sData, sData2, sIID) AS
(SELECT P.NAME, P.IID, S.NAME, S.NAME, S.IID
 FROM PLAY P, SONG S WHERE P.IID = S.PID)

WITH Q2 (bData, pIID, sData, uNumPos) AS
(SELECT Q1.bData, Q1.pIID, Q1.sData, row_number()
 OVER (PARTITION BY Q1.pIID ORDER BY Q1.sIID)
 uNumPos FROM Q1)

WITH Q3 (bData, pIID, sData) AS
(SELECT Q2.bData, Q2.pIID, Q2.sData FROM Q2
 WHERE Q2.uNumPos = 1)

WITH Q4 (bData, pIID, sData, sData2) AS
(SELECT Q1.bData, Q1.pIID, Q3.sData, Q1.sData2 FROM Q1,Q3
 WHERE Q1.pIID=Q3.pIID ORDER BY Q3.sData, Q2.sData2)

SELECT * FROM Q4
```

Fig. 6. SQL Translation for Q_1

6 Order-aware SQL Translation

XML-to-SQL translation based on XML algebra usually assumes some XML middle-ware above the relational engine. For this a simple middle-ware having limited processor and memory resources is commonly desired [12]. We follow the same trend here and limit the computation in the middle-ware to be achievable in a *single* pass over the SQL results. Many operators of the XAT algebra can be achieved in the middle-ware, such as Tagger, Combine, Select, Position and their combinations. The OrderBy and GroupBy operators can clearly not be evaluated by such one-pass middle-ware, unless the input has been sorted by the SQL engine correspondingly.

6.1 SQL Translation for Incompatible Result Orderings

In Figure 5, we can see that the ordering of the left and right branches of the query plan are incompatible with each other. That is, the left branch requires the intermediate result being sorted first on $\$pIID$ then on $\$sIID$, while the right branch requires first on $\$pIID$ then on $\$sData2$. For such cases, an additional order column, which can be the $\$sData2$ column after the selection, must be used for the top-level sorting in SQL. The ordering compatibility checking rules of intermediate results are the same as the rules for order context checking.

The SQL generation is conducted in a bottom up fashion along the XAT tree. Each time we try to include the parent operator in the current SQL block. Nested SQL statements will be generated otherwise. For the example Q_1, the generated SQL is shown in Figure 6. We use the *with* clause in SQL99 for clarity.

6.2 SQL Translation for Multiple Path Matching

We use the example XQuery $Q2$ for illustration of the order-sensitive SQL translation for complex XPath expressions. $Q2$ retrieves all SONG elements with an XPath matching multiple paths in the XML view. Then each path can be independently translated into an SQL query. Instead of simply combining the results

of the SQL queries, order-sensitive query translation needs to sort the XML elements from different paths correctly according to the ordered semantics of the XML view. The ordered semantics of the XML view include the following three categories of parent-child orders in the schema tree:

- **Sorting Order:** If the child node N_c is sorted under the parent node N_p, we call the parent-child order sorting order, denoted as $S(N_p, N_c) = \$col$, with $\$col$ as the sorting column(s).
- **Grouping Order:** If the child node N_c is grouped under the parent node N_p, we call the parent-child order grouping order, denoted as $G(N_p, N_c) = \$col$, with $\$col$ as the grouping column(s).
- **Edge Order:** The order among the sibling nodes N_c below a parent node N_p in the schema tree is called the edge order, denoted as $E(N_p, N_c) = i$, $i \in \mathbb{N}$, which means N_c is the i^{th} child of N_p.

In the XML schema tree in Figure 2(c), sorting order is: $S(PLAY, SONG) = \$sIID$; grouping order is $G(PLAY, MPLAY) = \$mplay$; and edge order is $E(PLAY, SONG) = 1$. These orders determine the ordering of the XML elements retrieved by the XPath expressions. In $Q2$ there are two paths in the XML view schema matching the XPath expression: "//SONG". We can construct a new order column for the SONG elements using concatenation of the parent-child orders along the path in a root-to-leaf direction. In case that both edge order and sorting order (or grouping order) exist for a parent-child pair, the edge order is concatenated prior to the sorting order. Sorting by the constructed order column after the union operations can achieve the correct ordering of the SONG elements. We show the translated SQL for $Q2$ in Figure 7(a)[3].

Alternative Computation Separation Strategies.

The union operator can be achieved using sorted merge on common columns in the middle-ware by one scan of the pre-sorted result sets. More precisely, one union operator is attached above every parent node that has multiple children (matching paths) in the XML view schema tree. Thus according to different allocations of the union operations in or out of the relational engine, alternative computation pushdown strategies are achievable. Figure 7(b) shows one alternative SQL translation. Q_{Left} and Q_{Right} provide the two sorted inputs for the sorted merge union in the middle-ware.

The SQL is generated as follows: 1) separate the schema tree into upper part and lower part, all the union operators of the upper part are done in the middle-ware; 2) for each path, the parent-child orders below the lowest union operator done in the middle-ware are used to construct the new order column using concatenations; 3) sort by the parent-child orders top-down along each path and the order column constructed (if any).

Pushing more union operators into the relational engine will have a smaller number of SQL queries but suffer from sorting on order columns constructed at runtime. Performing sorted merge union operators in the middle-ware will require a large number of cheap SQL queries, since the sorting can be done utilizing indices. This results in a performance tradeoff. The search space for identifying optimal strategies is linear in the number of possible union operators.

[3] We assume that the length of the strings for the IIDs are the same for one table.

```
SELECT NAME
FROM ( SELECT S.NAME,  P.IID||'.'||'1'||'.'||S.IID as song_order
          FROM PLAY P, SONG S
          WHERE P.IID = S.PID
          UNION ALL
          SELECT S.NAME,  P.IID||'.'||'2'||'.'||M.IID||'.'||S.NAME as song_order
          FROM PLAY P, MPLAY M, SONG S
          WHERE P.IID = M.PID AND M.IID = S.PID
       ) Q1
ORDER BY Q1.song_order
```

(a)

```
QLeft:
SELECT S.NAME
FROM PLAY P, SONG S
WHERE P.IID = S.PID
ORDER BY P.IID, S.IID
QRight:
SELECT NAME
FROM ( SELECT S.NAME, P.IID, M.IID||'.'||S.NAME as song_order
          FROM PLAY P, MPLAY M, SONG S
          WHERE P.IID = M.PID AND M.IID = S.PID
       ) Q1
ORDER BY Q1.IID, Q1.song_order
```

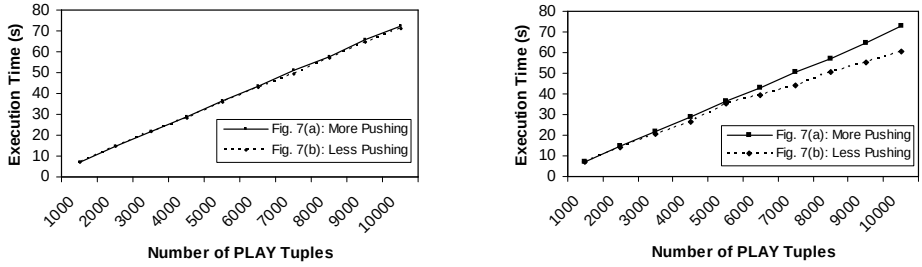
(b)

Fig. 7. Translated SQL for Merging Multiple Matched Paths.

7 Experimental Study

We have implemented the order-sensitive XQuery processing over XML views in the RainbowCore [20] system. We have conducted the performance comparisons among the different order-sensitive SQL translation strategies (Section 6). The experiments are done on a Linux machine with two P3 1GHz CPUs and 1GB memory running Oracle 9.

We compare the performance of the computation separation strategies for the union operators attached to the schema tree formed by multiple matching paths. We compare the execution costs based on the two SQL translations depicted in Figure 7(a) and 7(b). The dataset used includes 1000 to 10000 PLAYs having on average 50 SONGs and 10 MPLAYs per PLAY. Each MPLAY has on average 50 SONGs. The performance comparison of SQL translation in Figure 7(a) versus Figure 7(b) is shown in Figure 8(a) (without any index) and Figure 8(b) (with an index on the primary key).



(a) Without Any Index

(b) With Index on Primary Key

Fig. 8. Cost Comparison of SQL Translation for Multiple Matching Paths.

When no index is used, the two strategies perform similarly. When an index is present, pushing less union operators into the relational engine outperforms the alternative. The performance difference is increasing with the growth in the size of the relational tables. This experiment demonstrates that pushing less union operators into the relational engine is better than other strategies when indices are deployed.

8 Conclusions

The order semantics of XQuery are crucial for many application domains. We propose a generic approach for inference and isolation of the order semantics

in virtual XML views. Our approach turns the order-sensitive XQuery plans into unordered plans by utilizing order context annotations. Alternatives for SQL translation with order context are also discussed. Performance differences among them are illustrated through an experimental study.

References

1. M. El-Sayed, K. Dimitrova, and E. A. Rundensteiner. Efficiently Supporting Order in XML Query Processing. In *WIDM*, pages 147 – 154, 2003.
2. M. F. Fernandez, A. Morishima, and D. S. et al. Publishing Relational Data in XML: the SilkRoute Approach. *IEEE Data Eng. Bulletin*, 24(2):12–19, 2001.
3. T. Grust, M. van Keulen, and J. Teubner. Staircase join: Teach a relational dbms to watch its (axis) steps. In *VLDB*, pages 524–525, 2003.
4. J. Hidders and P. Michiels. Avoiding Unnecessary Ordering Operations in XPath. In *DBPL*, pages 54–70, 2003.
5. R. Krishnamurthy, R. Kaushik, and J. F. Naughton. XML-SQL Query Translation Literature: The State of the Art and Open Problems. In *Xsym*, pages 1–18, 2003.
6. N. May, S. Helmer, and G. Moerkotte. Nested queries and quantifiers in an ordered context. In *ICDE*, pages 239–250, 2004.
7. T. Neumann and G. Moerkotte. A combined framework for grouping and order optimization. In *VLDB*, pages 960–971, 2004.
8. T. Neumann and G. Moerkotte. An efficient framework for order optimization. In *ICDE*, page 461, 2004.
9. P. O'Neil, E. O'Neil, S. Pal, I. Cseri, G. Schaller, and N. Westbury. ORDPATHs: insert-friendly XML node labels. In *SIGMOD*, pages 903–908, 2004.
10. S. Paparizos, S. Al-Khalifa, A. Chapman, and H. V. J. et al. TIMBER: A Native System for Querying XML. In *SIGMOD*, page 672, 2003.
11. S. Paparizos and H. V. Jagadish. Pattern tree algebras: sets or sequences? In *VLDB*, pages 349–360, 2005.
12. J. Shanmugasundaram, J. Kiernan, E. J. Shekita, C. Fan, and J. Funderburk. Querying XML Views of Relational Data. In *VLDB*, pages 261–270, 2001.
13. D. E. Simmen, E. J. Shekita, and T. Malkemus. Fundamental Techniques for Order Optimization. In *SIGMOD*, pages 57–67, 1996.
14. G. Slivinskas, C. S. Jensen, and R. T. Snodgrass. Bringing order to query optimization. *SIGMOD Record*, 31(2):5–14, 2002.
15. I. Tatarinov, S. Viglas, and K. S. B. et al. Storing and querying ordered XML using a relational database system. In *SIGMOD*, pages 204–215, 2002.
16. W3C. XQuery 1.0: An XML Query Language. http://www.w3.org/TR/xquery/.
17. L. Wang, S. Wang, and E. Rundensteiner. Order-sensitive XML Query Processing over Relational Sources: An Algebraic Approach. In *IDEAS*, pages 175–184, 2005.
18. S. Wang, E. A. Rundensteiner, and M. Mani. Optimization of Nested XQuery Expressions with Orderby Clauses. In *ICDE Workshops:XSDM*, page 1277, 2005.
19. X. Wang and M. Cherniack. Avoiding Sorting and Grouping In Processing Queries. In *VLDB*, pages 826–837, 2003.
20. X. Zhang and K. D. et al. Rainbow: Multi-XQuery Optimization Using Materialized XML Views. In *SIGMOD*, page 671, 2003.
21. X. Zhang, B. Pielech, and E. A. Rundensteiner. Honey, I Shrunk the XQuery! — An XML Algebra Optimization Approach. In *WIDM*, pages 15–22, 2002.
22. X. Zhang and E. A. Rundensteiner. XAT: XML Algebra for the Rainbow System. Technical Report WPI-CS-TR-02-24, Worcester Polytechnic Institute, July 2002.