Order-sensitive XML Query Processing over Relational Sources: An Algebraic Approach

Ling Wang, Song Wang, Brian Murphy and Elke A. Rundensteiner
Computer Science Department, Worcester Polytechnic Institute, Worcester, MA 01609
Email: (lingw|songwang|rudie|rundenst)@cs.wpi.edu

Abstract

The XML data is order-sensitive. The order problem, that is how ordered XML documents and order-sensitive queries over it can be efficiently supported when mapped into the unordered relational data model, has not yet been adequately addressed. In this paper, we present a general approach for supporting order-sensitive XQuery-to-SQL translation that works irrespective of the chosen XML-to-relational data mapping and the selected order-encoding method. Our approach, called XSOT, utilizes an order-aware XML algebra representation. We propose order-sensitive rewriting rules at the algebraic level to eliminate the dependency of the order determining operators on the implicit XML view order. Furthermore, we introduce a series of order-sensitive optimization steps to transform the XML algebra tree for the purpose of efficient SQL translation. Lastly, we utilize a template-based approach using SQL-99 order features to generate SQL statements.

1 Introduction

Recent XML management systems [4, 8, 20] bridge relational databases and XML applications by creating XML views that wrap the relational base. Such system then answer queries against the view by translating them into SQL queries. However, while it is well known that XML [3] is an ordered data model and XQuery [16] is an order-sensitive query language, aspects related to order have been ignored in most of XQuery-to-SQL solutions [4, 8].

According to [11], the XQuery-to-SQL query translation problem can be broadly classified into two scenarios: *XML Publishing* and *XML Storage*. The XML publishing scenario studies the translation of queries over the XML views wrapping an *existing relational database*. Since the relational data model is not order sensitive, any XML result view generated over an (un-ordered) relational database is by default not ordered. Ordered XQuery expressions over such unordered XML views are meaningless. Hence, the order issue has been and for practical purposes can be safely ignored in this scenario.

On the other hand, the XML storage scenario focuses on storing and querying *existing XML data*. This involves three steps. First, a storage structure needs to be designed to load and maintain the XML data. Typically a relational database is used for this purpose. Second, a virtual XML view *identical* to the original XML document is extracted from the relational database. Third, queries against this wrapper view are translated into SQL statements to be executed against the relational database. When order is to be considered, the original XML document order must first be *explicitly* captured in the relational store along with the other document information. Then it can be extracted as *implicit order* into the XML view. This "round-trip" would guarantee that the extracted XML view has an associated document order like any other (regular) XML document. Thereafter an order-sensitive user query can expect to not only get ordered query result from such views, but also to be able to specify order oriented predicates to select the desired portion of the view. To answer such order-sensitive user queries, the XQuery-to-SQL translation needs to consider order in the query evaluation and optimization time.

Motivation. Different loading strategies might be required to store the XML data into the relational database based on application requirements [9]. For example, it has been shown that an inlining loading strategies is preferred when the XML schema is available, while an edge loading might be chosen when the XML schema is not available. Different order encodings have also been found to be useful for different update and query workloads [15]. The experimental study [15] shows that the performances of ordered XML queries and updates varies with the particular order encoding methods (e.g., global vs. local) and the loading strategies (e.g., inline vs. edge) used to build up the relational database from the XML model.

In fact it is possible to use a hybrid of multiple loading and order encoding strategies to load one XML document and schema into the relational database, especially for the purpose of speeding up certain heavily used user queries and updates. For example, in a music database application, most users will query their favorite band's information, while the administrator may frequently insert or delete the songs of all bands. One good design may be to order the bands by global order to facilitate fast retrieval, but encode their songs using Dewey order to optimize performance for this heavy update workload [15]. Moreover, the situation may arise when we want to integrate information from two different XML documents, where one may have been loaded using inline loading with Dewey order encoding versus the second document had been loaded using a different loading and encoding strategy.

It is obvious that hard-coding XQuery processing engine for one fixed order-encoding and loading combination is not practical. Developing an array of many different mapping and encoding specific query optimization and translation algorithms as done in [15] also not manageable in practical. There is clearly a need to develop one *general* XQuery-to-SQL translation approach handling any existing flexible encoding and loading strategies as well as possible future ones. This is exactly the focus of our work.

Utilizing both XQuery and relational technologies in harmony within the context of order handling can be a difficult task for two reasons. One, besides the document order, the user XQuery may impose new order on the XML query result through explicit $OrderBy$ clauses as well as through the structure of the *nested* FLOWR expressions of the user query. The query result reflects in an interrelated manner both the implicit *XML document order* and the order explicitly imposed by the *XQuery* expression [16]. Both order aspects have to be taken care of in the XQuery-to-SQL translation. Two, recent XML-relational systems [4, 8] push as much as possible of the query execution into the SQL engine, while leaving only construction operations (Taggers) for extraction to the middle-ware system. When we consider order in SQL translation, the question arises whether we indeed want to push all order operations down into the relational engine. Due to the fact that the nested SQL syntax destroys the order, translating as much as possible computation into SQL might force the engine to perform extra explicit sorting. As we will show, such redundant sorting may cause the generated SQL statement to become highly inefficient.

XSOT Approach. The XQuery-to-SQL Order-sensitive Translation (**XSOT**) approach presented in this paper is the first general order-handling approach that tackles the above challenges. Being built on top of the the *Rainbow* XML Query Engine [20], XSOT uses an XML algebra tree (XAT) as internal representation of both the view and user queries as well as their composition. A series of order-aware algebra equivalence rules, classified as *order explicit* rules and *SQL-oriented XAT rewrite* rules, are used to optimize the composed XAT for subsequent efficient SQL generation. An SQL translation algorithm then converts the optimized XAT into SQL statements (to be evaluated by the underlying relational engine) and construction operators (to be executed by the *Rainbow* XQuery engine). We have conducted several experiments to verify the feasibility and generality of our XSOT approach.

Our XSOT approach is *general* in the sense that the SQL translation techniques are independent of the loading and encoding strategies used to build the relational database. The reason of this independence is that the *view query* and the *order-code comparison functions* "encapsulate" both the order-encoding and the data loading diversity. Our XSOT framework is thus able to re-apply the techniques from the XML publishing scenario in this XML storage scenario, in particular query composition and optimization. It is *efficient* since most computation intensive operations (such as *OrderBy*) can be pushed down to the relational engine, while only a few operators, which can be evaluated efficiently, remain for the XQuery native engine.

Contributions. (1) We propose a general framework for XQuery-to-SQL order-sensitive translation (XSOT) approach. (2) We extend the XML algebra tree (XAT) in [20] to support the order-sensitive XQuery semantics. (3) We present SQL-oriented order-sensitive XAT optimization techniques. (4) We propose efficient order SQL statements generation and optimization techniques. (5) We have implemented our XSOT approach using the Rainbow XQuery Engine [20]. (6) Experiments are shown to verify the generality and to assess the SQL translation performance in different order-encoding and XML loading scenarios.

Outline. Section 2 introduces our XSOT system framework and necessary background in particular the XQuery algebra representation. Order preserving loading strategies and order sensitive XQuery examples are described in Section 3. Section 4 discusses the optimization techniques for order-sensitive SQL translation. The translation algorithm is presented in Section 5. Section 6 provides experimental studies. Section 7 reviews the related work while Section 8 provides the conclusions.

2 The XSOT Framework

As shown in Fig. 1, the architecture of XSOT includes two core subsystems, namely an order-preserving XML *Mapping Manager* and an order-sensitive *XQuery Engine*. First, the order-preserving XML *Mapping Manager* maps the original XML documents with their corresponding schemas when available (e.g., Fig. 2) into the relational storage. We embed the XML document order encoding techniques [15] into the loading strategies [6] and lossless-ly load XML document into the relational back-end. Losslessly loading here means that the identical XML document can be extracted back out. Fig. 3(a) shows the result for the local-order inline loading.

Second, the Rainbow XQuery engine is extended to support order-sensitive XQuery-to-SQL translation. A virtual *default XML view* extracted from the relational tables is automatically computed by the system. This default view corresponds to a one-to-one mapping between the hierarchical XML model and the flat relational data model. The database administrator then writes a *view query* over the default XML view. The view can be ordered appropriately based on the explicitly captured XML document order. Fig. 3(b) depicts the view query expressed in the XQuery language. It is used to reconstruct a view identical to the original XML document (Fig. 2) from the inlined relational database. The *order code comparison* function defined by the database administrator is used to compare two order codes. It thus hides the diversity between different order-encoding strategies from users.

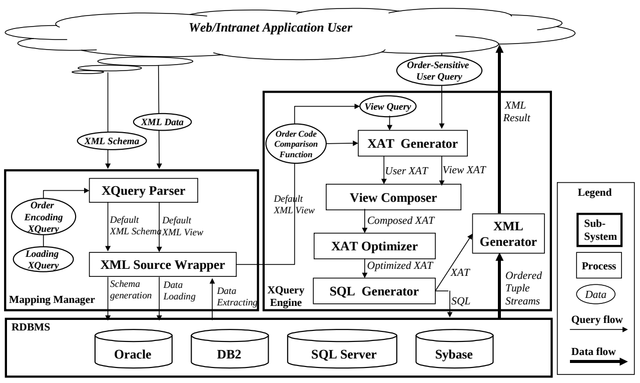


Figure 1. The Architecture of XSOT

```
<xs:schema xmlns:xsd="http://www.w3.org/XMLSchema">
 <xs:element name="RECORDLIST">
  <xs:complexType>
   <xs:element name="PLAY"
               minoccurs="1" maxOccurs="unbounded">
    <xs:complexType>
     <xs:sequence>
      <xs:element name="BAND" type="xs:string"/>
      <xs:element name="SONG" type="xs:string"
               minoccurs="1"/>
     </xs:sequence>
    </xs:complexType>
   </xs:element>
  </xs:complexType>
 </xs:element>
</xs:schema>
```

(a) record.xsd

```
<RECORDLIST>
  <PLAY>
      <BAND> Misfits </BAND>
      <SONG> She </SONG>
  </PLAY>
  <PLAY>
      <BAND> Back Street Boy </BAND>
      <SONG> Bullet </SONG>
      <SONG> We Are 138 </SONG>
  </PLAY>
  <PLAY>
      <BAND> Project X </BAND>
      <SONG> SXE Revenge </SONG>
       <SONG> Shutdown </SONG>
  </PLAY>
</RECORDLIST>
```

(b) record.xml

Figure 2. XML Schema and Document

To access the data, a user formulates a *user query* over the XML view. The Rainbow *XAT generator* represents both the view query and the user query as *XML Algebra Trees* (XAT), named *view XAT* and *user XAT* respectively. Particularly, order specific operators, such as the *OrderBy* operator and the *Position* function operator, are utilized to represent the order operations. *View Composer* then combines the two XATs into one *composed XAT* by replacing all leaf nodes of the user XAT with the view XAT [4, 21].

The *XAT Optimizer* optimizes this composed XAT with order in consideration. This includes a query *decorrelation* step to replace the costly nesting `FOR` operator [18, 19] and several optimization rules. We focus in particular on the addition of the order rewrite rules as described in Section 4.

The *optimized XAT* can be conceptually divided into the top XML construction portion and the bottom computation portion. The *SQL generator* translates the bottom portion into SQL using the order-based SQL template using the algorithms shown in Section 5. The generated order-sensitive SQL queries are sent to the relational engine. The *XML generator* takes the SQL query result and publishes it as the final XML query result to the application user.

3 Order-sensitive XML Algebra Tree

Rainbow XML Algebra Tree. Given that to date no standard XML algebra for query processing purposes has emerged, we will work here with the XML algebra named **XAT** [22] to represent the XQuery expression in the Rainbow query engine [20]. Fig. 5 depicts the correlated XAT representation for the view query (Fig. 3(b)).

The intermediate data model for the XAT algebra is a table model named *XAT table*. An **XAT table** R is an order-sensitive table of tuples t_j (e.g., $t_j \in R$), where the column names represent either a variable binding from the user or view XQuery, e.g., $\$record$ from Figure 4, or an internally generated variable name, e.g., $\$col_1$. Each cell c_{ij} in a tuple can store an XML node or a sequence of nodes.

Typically, an XAT operator takes as input one or more XAT tables and produces an XAT table as output. In general, an XAT operator is denoted as $op_{in}^{out}(s)$, where op is the operator symbol, in represents the input parameters, out the newly produced output column and s the input source(s) for that operator. The XAT operators are classified into two general categories: *XML operators* and *XAT SQL operators*. Here we restrict our discussion to the core subset of the XAT algebra operators [22].

XAT SQL operators correspond to the relational complete subset of the XAT algebra. They include `Select` $\sigma_c(R)$, `CartesianProduct` $\times(R,P)$, `ThetaJoin` $\bowtie_c$ (R,P), `LeftOuterJoin` $\overset{\circ}{\bowtie}_{Lc}(R,P)$, `Distinct` $\delta(R)$, `GroupBy` $\gamma_{col[1..n]}(R, func)$ and `OrderBy` $\tau_{col[1..n]}(R)$, where R and P denote XAT tables. Those operators are equivalent to their relational counterparts, with the additional requirements that the order among the tuples in the input XAT table(s) is reflected in the order among the tuples in the output XAT table. For example, in the output XAT table of `Select`, the relative order between each pair

RECORDLIST

IID	PID	POSITION
1	0	1

PLAY

IID	PID	POSITION	BAND_PCDATA
2	1	1	Misfits
3	1	2	Back Street Boy
4	1	3	Project X

SONG

IID	PID	POSITION	SONG_PCDATA
5	2	1	She
6	3	1	Bullet
7	3	2	We Are 138
8	4	1	SXE Revenge
9	4	2	Shutdown

(a)

```
<RECORDLIST>
FOR $play IN
      document("temp/dxv.xml")/PLAY/ROW
ORDER BY $play/POSITION/text()
RETURN
  <PLAY>
    FOR $song IN
        document("temp/dxv.xml")/SONG/ROW
        [PID/text() = $play/IID/text()]
    ORDER BY $song/POSITION/text()
    RETURN
      <SONG>
        $song/SONG_PCDATA/text()
      </SONG>
  </PLAY>
</RECORDLIST>
```

(b)

Figure 3. (a) Relational Database from Inlining Loading with Local-Order and (b) View Query over Fig.(a)

```
<RESULT>
FOR $record in document("record.xml")
RETURN
   <SONG>
       $record/PLAY/SONG[2]/text()
   </SONG>
</RESULT>
```

```
<RESULT>
    <SONG>
        We are 138
        Shutdown
    </SONG>
</RESULT>
```

Figure 4. An Order-sensitive Query and Query Result Example

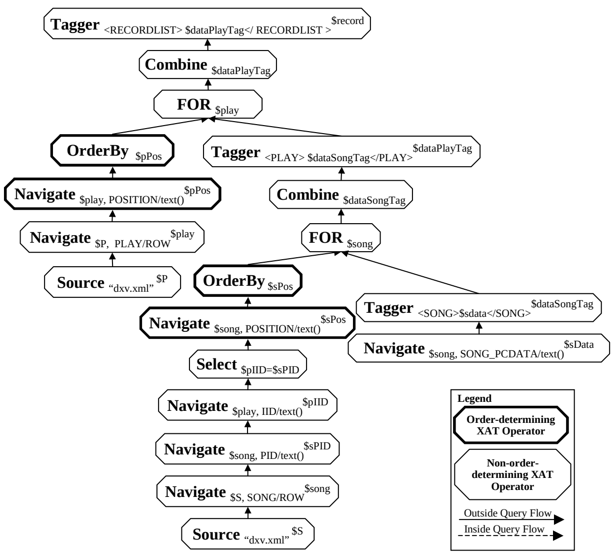

Tagger <RECORDLIST> $dataPlayTag</ RECORDLIST > $record
Combine $dataPlayTag
FOR $play
OrderBy $pPos
Tagger <PLAY> $dataSongTag</PLAY> $dataPlayTag
Navigate $play, POSITION/text() $pPos
Combine $dataSongTag
Navigate $P, PLAY/ROW $play
FOR $song
Source "dxv.xml" $P
OrderBy $sPos
Tagger <SONG>$sdata</SONG> $dataSongTag
Navigate $song, POSITION/text() $sPos
Navigate $song, SONG_PCDATA/text() $sData
Select $pIID=$sPID
Navigate $play, IID/text() $pIID
Navigate $song, PID/text() $sPID
Navigate $S, SONG/ROW $song
Source "dxv.xml" $S
Legend
Order-determining XAT Operator
Non-order-determining XAT Operator
Outside Query Flow
Inside Query Flow

Figure 5. Correlated XAT for View Query in Fig. 3 (b)

of tuples corresponds to the relative order between those two tuples in its input XAT table. The OrderBy operator, however, orders the tuples by the values in the columns given as argument.

The XML operators, used to represent the XML specific operations, are defined below. Source $S_{xmlDoc}^{col'}$ is always a leaf operator in an algebra tree. It takes the XML document $xmlDoc$ as input and outputs an XAT table with a single column col' and a single tuple $tout_1 = (c_{11})$, where the cell c_{11} contains the entire XML document.

Navigate $\phi_{col,path}^{col'}(R)$ unnests the element-subelement relationship. For each tuple tin_j from the input XAT table R, it creates a sequence of m output tuples $tout_j^{(l)}$, where $1 \leq l \leq m$, $m = |tin_j[col]/path|$, $tout_j^{(l)}[col'] = (tin_j[col]/path)[l]$.

Combine $C_{col}(R)$ groups the content of all cells corresponding to col into one sequence (with duplicates). Given the input R with m tuples tin_j, $1 \leq j \leq m$, Combine outputs one tuple $tout = (c)$, where $tout[col] = c = \overset{\circ}{\biguplus}_{j=1}^{m} tin_j[col]$.

Tagger $T_p^{col}(R)$ constructs new XML nodes by applying the tagging pattern p to each input tuple. A pattern p is a template of a valid XML fragment [3] with parameters being column names, e.g., $< result > col < /result >$. For each tuple tin_j from R, it creates one output tuple $tout_j$, where $tout_j[col]$ contains the constructed XML node obtained by evaluating the pattern p for the values in tin_j.

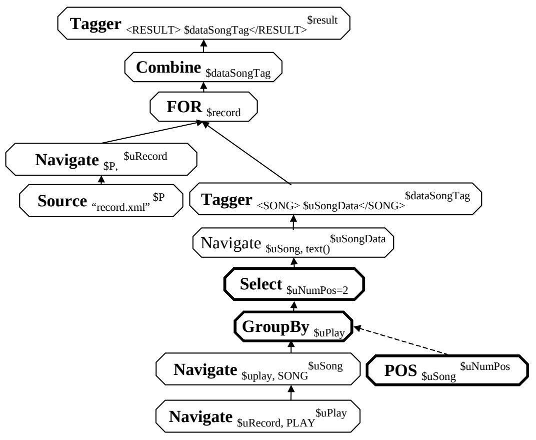
XAT Order Extension. The order-sensitive user XQueries include Position and Range predicates [15]. The basic XML algebra needs to be extended. A new operator named Position function is added for this purpose. The Position $POS_{col}^{col'}(R)$ function appends a new column col' to the input XAT table R to represent the relative positions of its tuples ordered by col. Fig. 6 depicts the correlated XAT for the user query in Fig. 4.

Since the user query XAT is defined over the view instead of the relational database, it needs to be composed with the view query to be evaluated over the relational data. A *Composed XAT* (Figure 7) is thus generated by replacing the Source node in the decorrelated user XAT with the decorrelated view XAT. For keeping correct order semantics of XAT through the decorrelation procedure, a set of order-sensitive decorrelation rules is used [19]. For example, the OrderBy operator in the inner query tree has to be grouped on its context, namely the outer FOR binding. For details please refer to [19].

4 XAT Optimization with Order

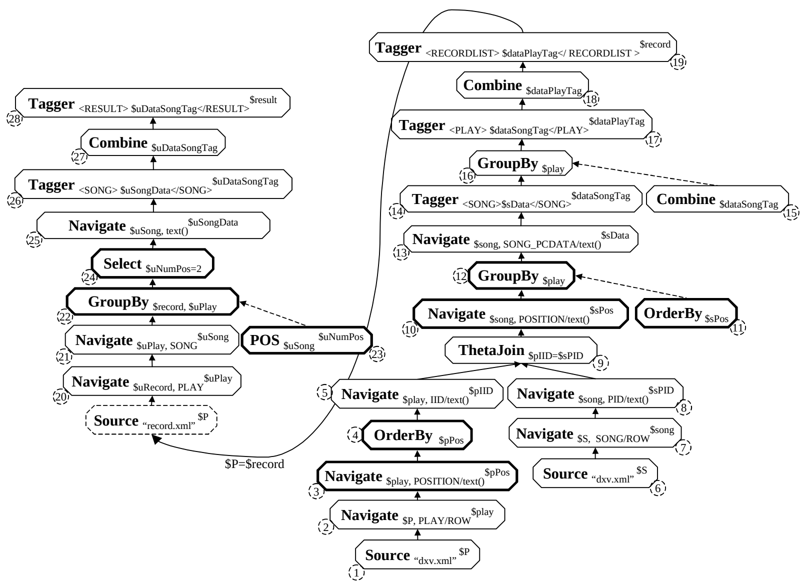
4.1 Order Dictionary

The operators in XAT can be classified as either order-determining or non-order-determining. An *order-determining XAT operator* imposes a new order on the output (e.g., OrderBy). A *non-order-determining XAT operator* just preserves and propagates order through. As identified by the thick-lined nodes in Figure 7, the order-determining XAT in the user XAT query tree includes the Position function and its context node GroupBy.



Tagger <RESULT> $dataSongTag</RESULT> $result
Combine $dataSongTag
FOR $record
Navigate $P, $uRecord
Source "record.xml" $P
Tagger <SONG> $uSongData</SONG> $dataSongTag
Navigate $uSong, text() $uSongData
Select $uNumPos=2
GroupBy $uPlay
Navigate $uplay, SONG $uSong
POS $uSong $uNumPos
Navigate $uRecord, PLAY $uPlay

Figure 6. The Generated Correlated XAT for Order Sensitive User XQuery in Fig. 4



Tagger <RESULT> $uDataSongTag</RESULT> $result 28
Combine $uDataSongTag 27
Tagger <SONG> $uSongData</SONG> $uDataSongTag 26
Navigate $uSong, text() $uSongData 25
Select $uNumPos=2 24
GroupBy $record, $uPlay 22
Navigate $uPlay, SONG $uSong 21
POS $uSong $uNumPos 23
Navigate $uRecord, PLAY $uPlay 20
Source "record.xml" $P
$P=$record
Tagger <RECORDLIST> $dataPlayTag</ RECORDLIST > $record 19
Combine $dataPlayTag 18
Tagger <PLAY> $dataSongTag</PLAY> $dataPlayTag 17
GroupBy $play 16
Tagger <SONG>$sData</SONG> $dataSongTag 14
Combine $dataSongTag 15
Navigate $song, SONG_PCDATA/text() $sData 13
GroupBy $play 12
Navigate $song, POSITION/text() $sPos 10
OrderBy $sPos 11
ThetaJoin $pIID=$sPID 9
Navigate $play, IID/text() $pIID 5
Navigate $song, PID/text() $sPID 8
OrderBy $pPos 4
Navigate $S, SONG/ROW $song 7
Navigate $play, POSITION/text() $pPos 3
Source "dxv.xml" $S 6
Navigate $P, PLAY/ROW $play 2
Source "dxv.xml" $P 1

Figure 7. XAT Composing View Query with User Query

While in the view portion, it includes the `OrderBy` and its context nodes (here `GroupBy`).

In a view defined by a FLWOR expression, the order of the view elements is determined by the "OrderBy" clauses. For example, in Fig. 3, the order of the result structure "PLAY" is decided by the clause "OrderBy $play/POSITION/text()". It thus in turn is decided by the order-determining XAT operators `Navigate` (node 3) and `OrderBy` (node 4) in Fig. 7 according to the view query.

However, this connection between the view order and order-determining operators in XAT is not captured by the view XAT itself. Thus a metadata table is created in parallel with the loading procedure to capture this connection, named the *Order Dictionary*.

XML PATH	Order Code
RECORDLIST	NULL
RECORDLIST/PLAY	PLAY/ROW/POSITION
RECORDLIST/PLAY/BAND	PLAY/ROW/POSITION
RECORDLIST/PLAY/SONG	SONG/ROW/POSITION

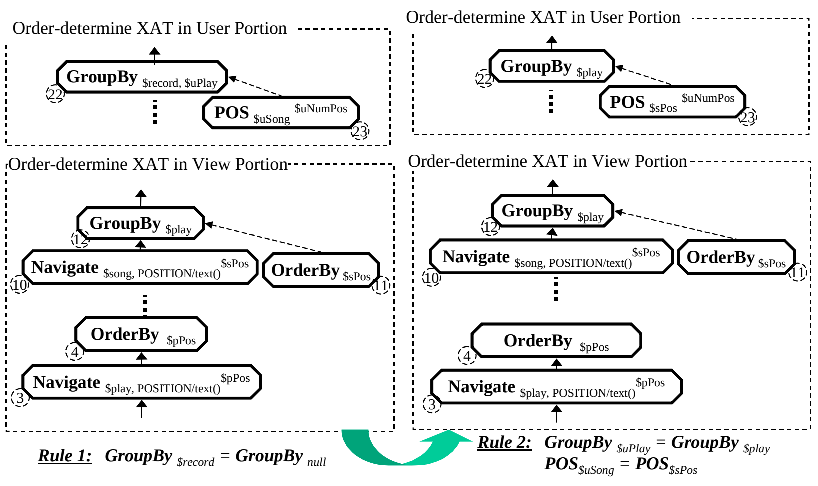
Table 1. Order Dictionary for Relational Database in Fig. 3

The *Order Dictionary* is a generic table that is suitable for any encoding strategies and loading methods. For example, Table 1 represents the Order Dictionary for the relational database in Fig. 3 The *XML Path* column stores the XPath of all XML elements in the view. For example, the full XML path is stored for the element "PLAY" as "RECORDLIST/PLAY". The *Order Code* column captures the order encoding XML path in the *default XML view (DXV)*, which refers to the relational column storing the order encoding values. Note that the first line of the Order Dictionary represents the root of the view. Since it is always a single XML element, we thus denote its order code as "NULL"(no order really matters here).

4.2 Order-sensitive XAT Optimization

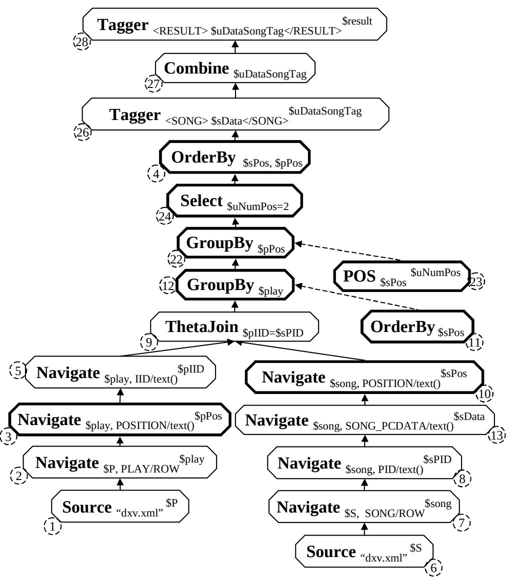
The order-sensitive XAT optimization includes two steps. (1) *Order explicit step*. As we mentioned before, the user order-determining XAT depends on the order of the view, which is *implicit*. Eliminating this dependency will open more opportunities for optimization. In other words, we need to change the `position` and `range` function from filtering on the data column (by its implicit physical position) to filtering on the order code column (by its explicit order value). (2) The *SQL-oriented XAT optimization step* transforms the XAT tree for efficient order-preserving SQL translation. The purpose is to push as many as possible operators down to the bottom of the XAT tree, as long as they have the corresponding relational operations. The rewrite rules used here include the *Computation push-down*, *Order pull-up* and *OrderStep rewrite* rules. Fig. 9 depicts the final optimized XAT for our running example.

Step 1: Order Explicit. As a commonly used optimization technique [14, 20], the `Tagger` operators in the view query and the corresponding `Navigate` operators in the user query can always be canceled out as long as no operator "above" the `Navigate` uses the result generated by



Order-determine XAT in User Portion
GroupBy $record, $uPlay
POS $uSong $uNumPos
Order-determine XAT in View Portion
GroupBy $play
Navigate $song, POSITION/text() $sPos
OrderBy $sPos
OrderBy $pPos
Navigate $play, POSITION/text() $pPos
Rule 1: GroupBy $record = GroupBy null
Order-determine XAT in User Portion
GroupBy $play
POS $sPos $uNumPos
Order-determine XAT in View Portion
GroupBy $play
Navigate $song, POSITION/text() $sPos
OrderBy $sPos
OrderBy $pPos
Navigate $play, POSITION/text() $pPos
Rule 2: GroupBy $uPlay = GroupBy $play
POS $uSong = POS $sPos

Figure 8. Optimize XAT in Fig. 7 using Rewrite Rules



Tagger <RESULT> $uDataSongTag</RESULT> $result
Combine $uDataSongTag
Tagger <SONG> $sData</SONG> $uDataSongTag
OrderBy $sPos, $pPos
Select $uNumPos=2
GroupBy $pPos
POS $sPos $uNumPos
GroupBy $play
OrderBy $sPos
ThetaJoin $pIID=$sPID
Navigate $play, IID/text() $pIID
Navigate $song, POSITION/text() $sPos
Navigate $play, POSITION/text() $pPos
Navigate $song, SONG_PCDATA/text() $sData
Navigate $P, PLAY/ROW $play
Navigate $song, PID/text() $sPID
Source "dxv.xml" $P
Navigate $S, SONG/ROW $song
Source "dxv.xml" $S

Figure 9. The XAT after Optimization

the `Tagger`. However, the order-determining operators (e.g., `Position` function) that appear in the user XAT depend on the implicit order of the `Tagger` result. In that case, the `Tagger` cannot be canceled since its result is required later. These non-cancelable `Tagger`s in turn block further optimization such as the computation push-down. The *order explicit* rule given below is utilized to eliminate this dependency on the `Tagger` result, and to create more opportunities for future optimization.

Given an XAT T, let $\overset{\circ}{T}_U$ be the set of the order-determining operators in the user XAT, while $\overset{\circ}{T}_V$ be the set of order-determining operators in the view XAT. Let O denote the Order Dictionary of T (Table 1). Two functions are used for our order explicit rewrite rules. Function *TraceVariableBinding($var, T)* returns the full XPATH p for a given variable binding $\$var$. For example, *TraceVariableBinding($uPlay,T)=“RECORDLIST/PLAY”*. Another function *OrderCode($var, O)* returns the order code c for the given variable binding $\$var$ from the Order Dictionary O. The function first uses *TraceVariableBinding($var,T)* to trace the binding XPath p for a given variable $\$var$. p is then used to look up the OrderCode in the Order Dictionary O. The OrderCode c of $\$var$ is finally returned. For example, *OrderCode($ucol1, O)* = “PLAY/ROW/POSITION”.

Fig. 8 shows the optimization for the composed XAT in Fig. 7 using the two order explicit rewrite rules below.

Rule 1: Partition Elimination Rewrite Rule.

Let GroupBy$_{\$col[1\ldots n]}$(R, func) $\in \overset{\circ}{T}_U$, where func = POS$^{\$col'}_{\$col}$ (R).
(OrderCode($\$col_i$,O)=NULL, $1 \le$ i $\le$ n)
$\Rightarrow$ GroupBy$_{\$col[1\ldots n]}$(R, func) = GroupBy$_{\$col[1\ldots(i-1),(i+1)\ldots n]}$(R, func).

Rule 1 indicates that the partition on a column $\$col$, represented by $GroupBy_{\$col}$ in the order-determining XAT, can be eliminated if the OrderCode of $\$col$ is “NULL”. Namely, there is no order inside $\$col$.

For example, since *TraceVariableBinding($record,T)* = *“RECORDLIST”* while *OrderCode($record,O)* = *NULL*, according to Rule 1, $GroupBy_{\$record,\$uPlay}$ = $GroupBy_{\$uPlay}$. That is, the order partition on $\$record$ does not make any difference, since in our example it only includes a single collection, namely the whole view.

Rule 2: Order Combination Rewrite Rule.

(i) Let $POS^{\$col'}_{\$col}(R) \in \overset{\circ}{T}_U$.($\exists \$var \in V$,
$OrderCode(\$col, O) = TraceVariableBinding(\$var, T))$
$\Rightarrow (POS^{\$col'}_{\$col}(R) = POS^{\$col'}_{\$var}(R))$.

(ii) Let $GroupBy_{\$col}$(R, func) $\in \overset{\circ}{T}_U$.
$\exists \$var \in V$, OrderCode($col,O) = TraceVariableBinding($var,T))
$\Rightarrow$ ($GroupBy_{\$col}$(R, func) = $GroupBy_{\$var}$(R,func)).

Rule 2 describes that the user order-determining XAT can be combined with the view order-determining XAT by replacing the user order column $\$col$ with the view order column $\$var$, as long as the OrderCode of $\$col$ and $\$var$ are equal. As a result, the user order-determining operator does not depend on the implicit order of the view result, but on the explicit order-determining view XAT.

For instance, *OrderCode($uPlay,O)=“PLAY/ROW/ POSITION”*, $TraceVariableBinding(\$play)$= “PLAY/ROW/POSITION”. According to Rule 2, we thus have $GroupBy_{\$uPlay}$(R, func)=$GroupBy_{\$play}$(R, func). That is the order of $\$uPlay$ in the user XAT is decided by the order of $\$play$ in the view XAT. Similarly, we have $POS^{\$uNumPos}_{\$uSong}(R) = POS^{\$uNumPos}_{\$sPos}$(R).

Step 2: SQL-Oriented XAT optimization. After the order explicit step above, the order-determining user XAT no longer depends on the intermediate view result. Traditional SQL-oriented rewrite rules such as *Navigate-Tagger Cancel-Out* and *Computation Push-down* [14, 20, 21] can now be used to optimize the XAT and prepare the XAT for order-sensitive SQL-generation. The optimized XAT is shown in Fig. 9.

OrderBy Pull-up. One rule specific for the order-sensitive SQL translation is called *Order Pull-up*. The SQL-standard (SQL-99) implies order overwrite between nested SQL statements. That is, the sorting of the inner query result is not kept by the outer SQL statement. The `OrderBy` operation should thus appear in the translated nested SQL statements as late as possible to avoid expensive re-orderings. For this purpose we designed the *OrderBy Pull-up* rule to pull the `OrderBy` operator high up the XAT tree.

An `OrderBy` operator τ_{in} can be pulled above an operator op_{in}^{out} as long as op is insensitive to the order of τ_{in}'s result. Typically, the `Position` function is the only operator sensitive to the order. Thus as long as the `Position` function is not sensitive to the result of `OrderBy`, it can be pushed through the XAT as far up as possible.

To do this, we record the ordered information of each intermediate XAT Table according to the semantics of each operator. If the pulling up does not destroy the correct ordered information in the result XML, we call such OrderBy pulling up *safe*. Pullup can then be performed without loss of ordered semantics. More details about OrderBy pulling up in XAT tree can be found in [18].

For example, compare the composed XAT in Fig. 7, and the optimized XAT in Fig. 9 the `Position` function (Node 23) is affected by the order of the column $\$sPos$ within the partition over the column $\$pPos$. It is not affected by the order of the column $\$pPos$. The `OrderBy` operator (Node 4) in Figure 7 can be pulled all the way up to the position shown in Fig. 9, since all the operators between the two positions are insensitive to the result of ordering on the column $\$pPos$. However, the `OrderBy` operator (Node 11) with its context `GroupBy` (Node 12) cannot be pushed through Node 22.

OrderStep Rewrite. To make SQL translation straightforward, the XAT operators which map to the SQL order-template are merged into one special operator, named `OrderStep`. An `OrderStep` $OS_{pcol[1...m],ocol[1...n]}^{col'}$ operator takes two input parameters: $pcol[1...m]$ is the set of partition columns, while $\$ocol[1...n]$ is the set of order columns. The output parameter is a column col' numbering the ordered output by some explicit ordering number.

In OrderStep rewrite, `OrderStep` operators are created by merging the `Position` function with all its order-related nodes. This includes its context `GroupBy` node, the `OrderBy` operator generating the ordered tuples used by the `Position` function, and the context node of this `OrderBy`. In Fig. 10, the `OrderStep` operator (Node 29) merges Nodes 11, 12, 22 and 23 from Fig. 9. It partitions on the column $\$pPos$, orders on the column $\$sPos$ and outputs the explicit ordering in column $\$uNumPos$. In the next section, we will discuss how the `OrderStep` operator is mapped to the SQL order template.

5 Order-based SQL Translation

Generating Order-sensitive SQL. SQL generation is done in an incremental bottom-up tree traversal process. As an operator is visited, an appropriate SQL statement fragment is created. The order-determining operator is translated into SQL order clauses by applying the SQL order template introduced below. For the algorithm of generating SQL, please refer to [17].

To translate the order-determining operators into SQL fragments, an order-based template (Fig. 13) has been designed. Although the grammar adheres to Oracle's "ordered" query lingua, the templates can be easily adapted for other DBMS specific SQL versions.

One feature specific in SQL-99 is the analytical function $row\_number()$. It creates integer values in the same fashion as the XAT `Position` function. The *over* method tells the analytical function what values to work with. The *PARTITIONBY* phrase creates groups or partitions as context on which the *ORDERBY* clauses is ordering. Not all queries require the partitioning clauses, indicated by the '?' at the end of the partition rule. More specifically, the partition clauses appear only if `POS` requires a `GroupBy` context operator. $\$pos\_func\_binding$ is the binding from the algebra tree. In our example, the *$pos_func_binding* is $\$sPos$. This binding is then constrained by the *WHERE* clause. For example, $\$uNumPos = 2$.

TEMPLATE:
  SELECT row_number() over
  (<PARTITION>?<ORDERBY>) $pos_func_binding
  FROM <TABLE> +
PARTITION: partition by <ELEMENT>

ORDERBY:
order by <TONUMBER> | <ELEMENT>
ELEMENT: element name
TONUMBER: to_number(<ELEMENT>)
TABLE: table name | TEMPLATE

Figure 13. Grammar of Order Template

The order template is filled when the `OrderStep` operator is encountered. Let $OS_{[pcol_1,...,pcol_n],[ocol_1,...,ocol_n]}^{col'}$. The corresponding SQL order fragment is: *"row_number() over (PARTITION BY $pcol_1, ..., pcol_n$ ORDER by $ocol_1, ..., ocol_n$) col'"*. For example, the Node 29 (`OrderStep`) in Figure 10 is translated into the clauses *"row_number() over (PARTITION BY pPos ORDER by sPos) uNumPos"*.

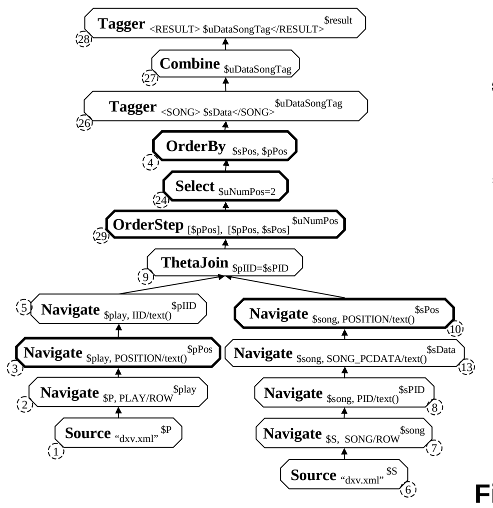
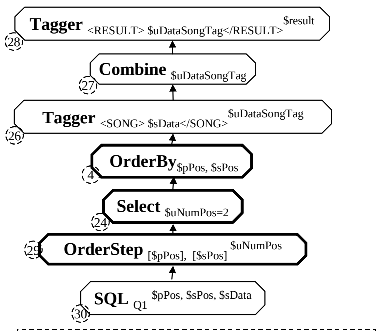
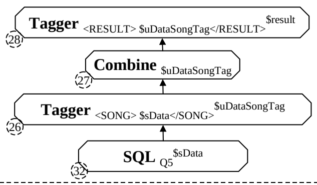

Tagger <RESULT> $uDataSongTag</RESULT> $result 28
Combine $uDataSongTag 27
Tagger <SONG> $sData</SONG> $uDataSongTag 26
OrderBy $sPos, $pPos 4
Select $uNumPos=2 24
OrderStep [$pPos], [$pPos, $sPos] $uNumPos 29
ThetaJoin $pIID=$sPID 9
Navigate $play, IID/text() $pIID 5
Navigate $song, POSITION/text() $sPos 10
Navigate $play, POSITION/text() $pPos 3
Navigate $song, SONG_PCDATA/text() $sData 13
Navigate $P, PLAY/ROW $play 2
Navigate $song, PID/text() $sPID 8
Source "dxv.xml" $P 1
Navigate $S, SONG/ROW $song 7
Source "dxv.xml" $S 6

Figure 10. The XAT after OrderStep Rewrite


Tagger <RESULT> $uDataSongTag</RESULT> $result 28
Combine $uDataSongTag 27
Tagger <SONG> $sData</SONG> $uDataSongTag 26
OrderBy $pPos, $sPos 4
Select $uNumPos=2 24
OrderStep [$pPos], [$sPos] $uNumPos 29
SQL Q1 $pPos, $sPos, $sData 30

```
Q1 =  SELECT P.POSITION AS  pPos, S.POSITION AS sPos,
              S.SONG_PCDATA AS sData
        FROM PLAY P, SONG S
        WHERE P.IID = S.PID
```

Figure 11. Translated subquery for XAT in 10


Tagger <RESULT> $uDataSongTag</RESULT> $result 28
Combine $uDataSongTag 27
Tagger <SONG> $sData</SONG> $uDataSongTag 26
SQL Q5 $sData 32

```
Q5    =
        SELECT Q2.sData
        FROM
           (SELECT Q1.pPos, Q1.sPos, Q1.sData,
                      row_number()  OVER  (PARTITION BY Q1.pPos
                       ORDER BY Q1.sPos)  uNumPos
           FROM
              (SELECT P.POSITION AS  pPos, S.POSITION AS sPos,
                    S.SONG_PCDATA AS sData
              FROM PLAY P, SONG S
              WHERE P.IID = S.PID
              ) Q1
           ) Q2
        WHERE Q2.uNumPos = 2
        ORDER BY Q2.pPos, Q2.sPos
```

Figure 12. Translate SQL for XAT in 11

Note that our SQL generation algorithm does not rely on any specific loading or order-encoding. Unlike existing work in [15], it is not hard-coded but rather a general solution. In fact, there is only a rather small difference when the SQL statements are generated for the edge loading. For the interested readers, in [17], the XQuery-to-SQL translation procedure over an edge loaded relational database will be described. The similarity of the generated SQL statements confirms the generic nature of our solution.

About Push-down Strategies. For the relational engines that do not support the $row\_number()$ function, the SQL generation can be stopped before the OrderStep, causing the other operators to be conducted in the middle-ware. Since the middle-ware operations can keep the ordered semantics of the input XAT Tables, the re-sorting can be avoided. Thus even if the query engine supports the SQL-99, not pushing the OrderStep into the relational engine may be preferred. Since the middle-ware is limited in memory and other resources, we limit the computing power of the middle-ware to the extent that such computation can be achieved by a single pass over the SQL query results. Based on this criterion, the two push down strategies are achievable by the middle-ware.

Without considering order, in general we try to push as much as possible computation into the relational engine to alleviate the workload and complexity of the middle-ware. At the same time, better performance is expected by pushing more into the relational engine. It is a quite different scenario when the order is considered. The SQL engine is not defined for order except the explicit sorting clause. We have to perform multiple sortings, which may even be repeated, during the SQL translation. For example, in the subquery $Q2$ in Figure 12, a sorting on $sPos$ is performed for the partition clause, while in the $Q5$, repeated sorting on $sPos$ is performed again. For some cases, such repeated sorting will degrade the performance.

There are multiple choices for pushing computation to the relational engines considering order. None of them can always outperform others. There are some tradeoffs between them like the selectivity of the selection operator, network traffic cost, sorting cost and others. Based on these statistics, a better push down strategy can be chosen. We perform a comparison of these SQL translation strategies also in the experimental session.

Discussion: Further Optimization of SQL. To highlight the order related aspects of our approach, we choose to generate a single nested SQL statement. We note, however, that the *SilkRoute* [8] cost-based optimization for SQL translation could be fairly easily embedded into our XSOT framework and thus optimizing the generated SQL statement.

The SQL statements generated by our translation algorithm can be further optimized if some order encoding knowledge is assumed. For example, if the global order encoding is used, then the SQL statement in Fig. 12 can be optimized by ordering only on the column $\$sPos$, instead of ordering on both $\$pPos$ and $\$sPos$. The reason is that the global order encoding of $\$sPos$ includes the order information of the $\$pPos$.

Assuming the SQL translation component is aware of the underlying relational schema and its constraints, the generated SQL can be further optimized. For example, the schema-specific SQL optimization [10] could be plugged into our XSOT framework.

6 Experimental Study

The experiments were run on a UNIX machine with two PIII450M CPU processors and 512 Megs of RAM The XML data generated complies with the schema of Fig. 2(a) and includes 10000 PLAYs/file. The underlying relational database was loaded with the Inline and the Edge shredding and two different order encodings (global and local). The test queries are shown in Figure 14.

On Pushing the OrderStep into SQL. Different selectivities of the Select operator (Node 24 in Figure 11) will affect the network cost between the middle-ware and the database server. Different SQLs then will have different performance. In this experiment, we vary the condition of the Select operator to achieve various selectivities of the Select operator. The result is shown in Fig. 15.

In Fig. 15, deep pushing refers to the pushing of the OrderStep into the relational engine, and shallow pushing refers to not pushing the OrderStep into the relational engine. When the selectivity of the Select operator is low, these two pushing strategies perform similarly. When the selectivity is high (over 10%), the shallow pushing outperforms the deep pushing, since the cost of the repeated sorting in the deep pushing will be significant. This is an interesting observation, indicating that pushing as much as possible computation into relational engine may not always be preferable, when order processing is considered.

On the Loading and Encoding Strategies. Various loading strategies, such as *inline* and *edge*, are used in various scenarios to create various relational database structures. Edge is usually used when the schema is not available, while Inline is used in the schema-aware case. Even if the same order encoding strategy is used, the translated SQLs over these two databases differ as shown in Fig. 16. The SQL queries over the edge loaded database is typically more expensive than those over the inline loaded database. The edge loaded relational database requires self-Joins over one huge table, which is rather time consuming.

Different example queries depict preference for different order encoding strategies as shown by Fig. 16. Q1 and Q4 perform better in the inline loaded database using local encoding rather than using global encoding. The reason is they both require the several SONGs from each PLAY, which implicitly favors a local order. Two other queries (Q2 and Q5) require several SONGs from all PLAYs, which is easy for the globally ordered inline database. While Q3 and Q6 return all the SONGs belonging to particular PLAYs, which do not differ between different encodings.

SQL Execution and XML Construction. The output of the SQL operator (Node 32) is an *ordered* tuple stream. This ordered tuple stream serves as input for the remaining construction operators (`Tagger` and `Combine`). Total time refers to the time from the execution start until the result output, including SQL execution and construction. The comparison of the SQL execution time and the total time for all queries under the edge loading is shown in Figures 17 and 18. The SQL execution takes 70% of the total execution in our example cases.

7 Related Work

Order as a key issue specific to the XML data model has not yet been addressed by any of those research projects [2, 4, 7, 8, 12] nor by any of the commercial systems [1, 5, 13].

[15] is one of the earliest works assessing the issue of order in the XML-to-SQL context. Three *order encoding methods* are utilized to encode XML order. Algorithms of translating ordered XPath expressions into SQL, one specific to each encoding and loading method, are proposed respectively. The performance of the ordered-encoding methods on a workload of ordered XML queries is also presented. However, each proposed algorithm is *dependent* on and *specific* to the loading and encoding strategy used to build the relational database to begin with. That is, (1) the knowledge of loading and encoding is required by the translation algorithm, and (2) different loading and encoding strategies require different translation algorithms. In addition, not only the translation strategies proposed but also the performance studies described concentrate on the correctness of XPATH translation and evaluation. The complexity of handling order-sensitive XQuery statements, nested or not nested, is not addressed.

Compared with their work, our XSOT approach is *independent* as it does not require any knowledge of the utilized loading strategy nor order encoding method. In other words, the gap between different loading and encoding strategies is naturally covered by the *view query* embedding this knowledge and the *order code comparison function* for each specific order encoding. It is also *generic* since the algebraic representation is used to represent the input XQuery. One single uniform translation algorithm serves for all combinations of existing encoding and loading strategies, even for the possibly new ones to be introduced in the future.

8 Conclusions

In this paper, we propose an algebraic approach for order-sensitive XQuery processing over relational databases. Based on an XML algebra tree (XAT) to represent order-sensitive XQuery expressions, a series of order-related optimization steps for XQuery to SQL translation are proposed. Our approach now serves as a general solution for order-sensitive XQuery to SQL translation, which

Q1
```
<RESULT>
FOR $record in document("record.xml")
RETURN
   <SONG> $record/PLAY/SONG[5]/text() </SONG>
</RESULT>
```

Q2
```
<RESULT>
FOR $record in document("record.xml")
RETURN
    <SONG>
      ($record/PLAY/SONG)[900]/text()
    </SONG>
</RESULT>
```

Q3
```
<RESULT>
FOR $play in document("record.xml")/PLAY[800]
RETURN
  <SONG>$play/SONG/text()</SONG>
</RESULT>
```

Q4
```
<RESULT>
FOR $record in document("record.xml")
RETURN
   <SONG>$record/PLAY/SONG[Position()=2 to 5]/text()</SONG>
</RESULT>
```

Q5
```
<RESULT>
FOR $record in document("record.xml")
RETURN
   <SONG>
      ($record/PLAY/SONG)[Position()=100 to 800]/text()
   </SONG>
</RESULT>
```

Q6
```
<RESULT>
FOR $play in document("record.xml")/PLAY[ Position()=100 to 800]
RETURN
  <SONG>$play/SONG/text()</SONG>
</RESULT>
```

Figure 14. The Test Queries used in Experiment

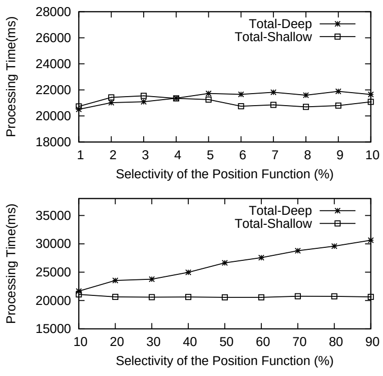


Figure 15. Different Push Strategies for Various Selectivity

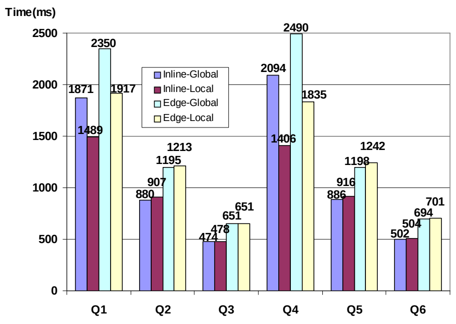


Figure 16. SQL Execution Time when Global vs. Local Encoding used

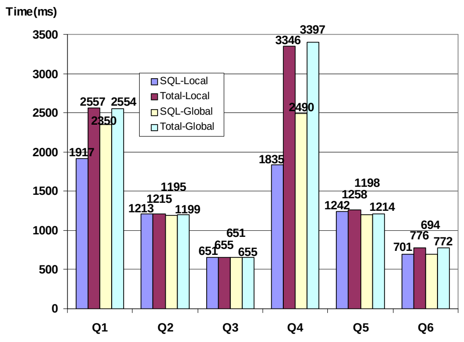


Figure 17. SQL vs. Total Time using Edge Loading

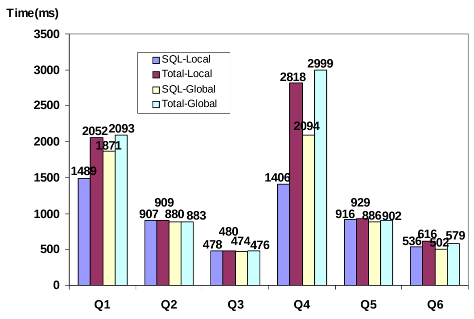


Figure 18. SQL vs. Total Time using Inline Loading

is irrespective of the data loading and order encoding strategies used in building the underlying relational database.

References

[1] S. Banerjee, V. Krishnamurthy, M. Krishnaprasad, and R. Murthy. Oracle8i - The XML Enabled Data Management System. In *ICDE*, pages 561–568, 2000.

[2] P. Bohannon, S. Ganguly, H. F. Korth, P. Narayan, and P. Shenoy. Optimizing View Queries in ROLEX to Support Navigable Result Trees. In *VLDB*, 2002.

[3] E. T. Bray, J. Paoli, and C. Sperberg-McQueen. Extensible Markup Language (XML), 1997. http://www.w3.org/TR/PR-xml-971208.

[4] M. J. Carey, J. Kiernan, J.Shanmugasundaram, E. J. Shekita, and S. N. Subramanian. XPERANTO: Middleware for Publishing Object-Relational Data as XML Documents. In *The VLDB Journal*, pages 646–648, 2000.

[5] J. M. Cheng and J. Xu. XML and DB2. In *ICDE*, pages 569–573, 2000.

[6] S. Christ and E. A. Rundensteiner. X-Cube: A fleXible XML Mapping System Powered by XQuery. Technical Report WPI-CS-TR-02-18, Computer Science Department, WPI, 2002.

[7] A. Deutsch and V. Tannen. MARS: A System for Publishing XML from Mixed and Redundant Storage. In *VLDB*, 2003.

[8] M. F. Fernandez, A. Morishima, D. Suciu, and W. C. Tan. Publishing Relational Data in XML: the SilkRoute Approach. *IEEE Data Engineering Bulletin*, 24(2):12–19, 2001.

[9] J. Shanmugasundaram et al. Relational Databases for Querying XML Documents: Limitations and Opportunities. In *VLDB*, pages 302–314, September 1999.

[10] R. Krishnamurthy, R. Kaushik, and J. F. Naughton. Optimizing Fixed-Schema XML to SQL Query Translation. In *VLDB*, 2002.

[11] R. Krishnamurthy, R. Kaushik, and J. F. Naughton. XML-to-SQL Query Translation Literature: The State of the Art and Open Problems. In *XSym (VLDB Workshop)*, 2003.

[12] I. Manolescu, D. Floresce, and D. Kossmann. Answering XML Queries over Heterogeneous Data Sources. In *VLDB*, 2001.

[13] M. Rys. Bringing the Internet to Your Database: Using SQL Server 2000 and XML to Build Loosely-Coupled Systems. In *VLDB*, pages 465–472, 2001.

[14] J. Shanmugasundaram, J. Kiernan, E. J. Shekita, C. Fan, and J. Funderburk. Querying XML Views of Relational Data. In *VLDB*, 2001.

[15] I. Tatarinov, S. D. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita, and C. Zhang. Storing and Querying Ordered XML Using a Relational Database System. In *SIGMOD*, 2002.

[16] W3C. XQuery 1.0: An XML Query Language. http://www.w3.org/TR/xquery/, May 2003.

[17] L. Wang, S. Wang, B. Murphy, and E. A. Rundensteiner. An Algebraic Approach for Order-sensitive XML Query Processing over Relational Sources. Technical report, Computer Science Department, WPI, 2004 in progress.

[18] S. Wang, E. A. Rundensteiner, and M. Mani. Optimization of Nested XQuery Expressions with Orderby Clauses. In *ICDE Workshop of XML Schema and Data Management 2005*, 2005.

[19] S. Wang, X. Zhang, E. A. Rundensteiner, and M. Mani. An Algebraic Approach towards Complex XQuery Unnesting. Technical Report WPI-CS-TR-05-01, Computer Science Department, WPI, 2005.

[20] X. Zhang, K. Dimitrova, L. Wang, M. EL-Sayed, B. Murphy, L. Ding, and E. A. Rundensteiner. RainbowII: Multi-XQuery Optimization Using Materialized XML Views. In *Demo Session Proceedings of 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. Rundensteiner. XAT: XML Algebra for Rainbow System. Technical Report WPI-CS-TR-02-24, Computer Science Department, WPI, July 2002.