Order-sensitive XML Query Processing Over Relational Sources

by

Brian Murphy

A Thesis

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Master of Science

in

Computer Science

by

May 2003

APPROVED:

Professor Elke Rundensteiner, Major Thesis Advisor

Professor Lee Becker, Thesis Reader

Professor Micha Hofri, Head of Department

Abstract

XML is an emerging standard format for data on the Web as well as in business applications. In order to store and access this information in an efficient manner, database technology must be utilized. A relational database system, the most established and mature technology for query processing and storage, creates a strong foundation for such an XML data management system. However, while relational databases are based on SQL queries, the original user queries are written in XQuery, an XML query language. This XML query language has support for order-sensitive queries as XML is an order-sensitive markup language.

A major problem has been discovered with loading XML in a relational database. That problem is the lack of native SQL support for and management of order handling. While XQuery has order and positional support, SQL does not have the same support. For example, individuals who were viewing XML information about music albums would have a hard time querying for the first three songs of a track list from a relational backend. Mapping XML documents to relational backends also proves hard as the data models (hierarchical elements versus flat tables) are so different.

For these reasons, and other purposes, the Rainbow System is being developed at WPI as a system that bridges XML data and relational data. This thesis in particular deals with the algebra operators that affect order, order sensitive loading and mapping of XML documents, and the pushdown of order handling into SQL-capable query engines. The contributions of the thesis are the order-sensitive rewrite rules, new XML to relational mappings with different order styles, order-sensitive template-driven SQL generation, and a proposed metadata table for order-sensitive information. A system that implements these proposed techniques with XQuery as the XML query language and

Oracle as the backend relational storage system has been developed. Experiments were created to measure execution time based on various factors. First, scalability of the system as backend data set size grows is studied. Second, scalability of the system as results returned from the database grows, and finally, query execution times with different loading types are explored. The experimental results are encouraging. Query execution with the relational backend proves to be much faster than native execution within the Rainbow system. These results confirm the practical utility of our proposed order-sensitive XQuery execution solution over relational data.

Acknowledgements

First, I would like to thank my advisor Prof. Elke Rundensteiner. She pushed and prodded me the whole way through this Thesis. Without her, none of this would be here. Also to my reader, Prof. Lee Becker. I kept him in the dark throughout most of my research and writing phases, but every time I saw him, he had nothing but positive things to say.

Second, I would like to thank my fellow Rainbow team members new and old: Brad Pielech, Mukesh Mulchandani, Steffen Christ, Katica Dimitrova, Luping Ding, Song Wang, Ling Wang, and Maged El Sayed with special thanks to Xin Zhang. Also many thanks to my DSRG office mates, again, old and new: Li Chen, Hong Su, Yali Zhou, and Andreas Koeller.

Finally, I would like to thank my friends and family for all their help and support.

Contents

1 Introduction **1**

1.1 Motivation 1

1.2 State of the Query Processing Systems 2

1.3 Problem Definition 3

1.4 My Approach 3

1.5 Contributions 4

1.6 Thesis Outline 5

2 Related Work **6**

2.1 XML Query Processing 6

2.2 Order Based Queries 8

2.3 Temporal Data Bases 8

3 Background **9**

3.1 XML Schema 9

3.2 XML Data 10

3.3 XQuery Statements 10

3.4 Default XML View 11

4 Rainbow **12**

4.1 System Overview 12
4.2 Order-sensitive Rainbow Modifications 13
4.3 Data Flow of the Order-Sensitive System 14

5 XML Algebra and Rewrite Rules 15

5.1 XAT Data Model 15
5.2 XAT Binding Table 16
5.3 XAT Operators 16
5.3.1 Select 16
5.3.2 Position Function 17
5.3.3 GroupBy 17
5.4 XAT Decorrelation 18
5.5 XAT Rewrites 21
5.5.1 GroupBy/Function Replace 21
5.5.2 Single Step Order Rewrite 22
5.5.3 Multi Step Order Rewrite 22

6 XML Document Loading 24

6.1 Relational Storage 24
6.2 Order Preservation 24
6.2.1 Local Order 25
6.2.2 Global Order 25
6.2.3 Full Path Order 26
6.3 Guidelines for Ordering Methods 27
6.4 Loading 27
6.4.1 Inline Loading 28
6.4.2 Edge Loading 29

6.5 General Loading 31
6.6 Metadata Table 31
6.7 Query Composition 33
6.8 View Queries 33

7 Order-based Methodology 35

7.1 The Order-Sensitive XQuery 35
7.2 Position Operator Replace 36
7.3 Metadata Table Query 36
7.4 Main Memory Position Execution 38

8 Order-based Template Heuristics 39

8.1 Order-Sensitive SQL grammar 40
8.2 Template Completion 41
8.3 General Template Discussion 44

9 Implementation 45

9.1 SQL Generation 47

10 Experiments 51

10.1 System Setup 51
10.2 Experimental Evaluation 52
10.2.1 Single Step Query Experiments 52
10.2.2 Multi Step Query Experiments 55
10.3 Experiments with Varying Selectivity 57
10.3.1 Single Step Query Experiments 57
10.3.2 Multi Step Query Experiments 58
10.4 Experimental Summary 59

11 Conclusions **61**

11.1 Summary 61

11.2 Contributions 62

11.3 Future Work 63

A Loading Queries **67**

A.1 Schema Queries 67

A.1.1 Edge Loading Schema Query 67

A.1.2 Inline Loading Schema Query 68

A.2 Step 0 69

A.2.1 Local Edge Loading Step 0 69

A.2.2 Global Edge Loading Step 0 71

A.2.3 Lexicographical Edge Loading Step 0 73

A.2.4 Local Inline Loading Step 0 75

A.2.5 Global Inline Loading Step 0 77

A.2.6 Lexicographical Inline Loading Step 0 79

A.3 Step 1 81

A.3.1 Edge Loading Step 1 81

A.3.2 Inline Loading Step 1 82

A.4 Step 2 84

A.4.1 Edge Loading Step 2 84

A.4.2 Inline Loading Step 2 84

B View Queries **86**

B.0.3 Edge Loading 86

B.0.4 Inline Loading 86

C Recordlist XML Schema **87**

D Default XML View **88**

List of Figures

3.1 Record List Schema in the Condensed W3C Format [18] 9

3.2 Record List XML Document 10

3.3 Order Sensitive XQuery Q1 10

3.4 XML Document Result of XQuery Q1 10

3.5 Order Sensitive XQuery Q2 11

3.6 Result of XQuery Q2 11

4.1 The Rainbow System Architecture 13

5.1 Order Sensitive XQuery Q1 as XAT Tree before Decorrelation 18

5.2 Order Sensitive XQuery Q2 as XAT Tree before Decorrelation 19

5.3 Decorrelation example with query Q2 20

5.4 Order Sensitive XQuery Q1 as XAT Tree after Decorrelation 20

5.5 Order Sensitive XQuery Q2 as XAT Tree after Decorrelation 21

5.6 Q1 XAT Segment Rewriting 22

5.7 Q2 XAT Segment Rewriting 23

6.1 Local Order Encoding: Sibling Order Traversal 25

6.2 Global Order Encoding: Post-Order Traversal 26

6.3 Full Path Order Encoding 27

6.4 Full Path Order Encoding with Update Sn 27

6.5 Relational Inlining Strategy with Local Order 30
6.6 Relational Edges Strategy with Global Order 30
6.7 View (Extraction) Query for Q1 (Figure 3.3) with Inline Strategy 34

7.1 Metdata Query for Q1 36
7.2 Query Q1 as XAT Tree with Rewrite 37
7.3 Query Q2 as XAT Tree with Rewrite 37

8.1 SQL Grammar for Order-Sensitive Queries 40
8.2 The Template for our Example Filled 42
8.3 Query Q1 from Figure 7.2 after SQL Generation 43
8.4 Query Q2 from Figure 7.3 after SQL Generation 43
8.5 Query Q1 over an Inline Loading with Alphanumeric Order 44
8.6 Query Q1 over an Edge Loading with Alphanumeric Order 44

9.1 Order Sensitive XQuery Q3 48
9.2 XML Document Result of XQuery Q3 48
9.3 Q3 Incorrect SQL Statement for Inline Global Loading 49
9.4 Q3 Correct SQL Statement for Inline Global Loading 49

List of Tables

6.1 Order-Based Metadata Table for the Inlined Recordlist Document 31

8.1 Condensed Order-Based Metadata Table for the Inlined Recordlist Document 41

10.1 Query Q1 Executed with Global Edge Loading 52
10.2 Query Q1 Executed with Local Edge Loading 53
10.3 Query Q1 Executed with Global Inline Loading 54
10.4 Query Q1 Executed with Local Inline Loading 54
10.5 Query Q2 Executed with Global Inline Loading 55
10.6 Query Q2 Executed with Local Inline Loading 56
10.7 Query Q1 with Selectivity Executed with Global Edge Loading 56
10.8 Query Q1 with Selectivity Executed with Global Inline Loading 57
10.9 Query Q2 with Selectivity Executed with Global Edge Loading 58
10.10 Query Q2 with Selectivity Executed with Global Inline Loading 58

Chapter 1

Introduction

1.1 Motivation

XML [1] has become *the* standard markup language for documents on the World Wide Web and within the business community. In many instances, these documents are stored in hard to query file systems. A better solution would be to store them in a relational database [11], which has been proven through time to be highly optimized for queries, secure and mature. These are qualities which are desired by the business community. However, utilizing both technologies in harmony can be a difficult task. Storing or loading XML documents into relational tables is not always a lossless operation. Information regarding order structure, hierarchical and sibling order specifically may be lost in the relational database. This ordering of element information, despite being implicit, is critical to document reconstruction and thus to correct query processing. Queries based on this implicit document order can also be important to the end user. For example, suppose the end users wants to know the second song title from a tracklist. This query would be rather simple in XQuery, but more difficult to produce in SQL. Hence a system needs to be implemented which can effectively manage flexible order-sensitive document loading

as well as efficient order-sensitive query processing.

1.2 State of the Query Processing Systems

Currently systems exist to query over XML documents using XQuery, the now almost standard XML query language. Many [13, 9] are considered native systems in that their data is represented and stored in some proprietary XML specific representation, such as DOM [16]. These systems are useful as a proof of concepts, but in the long run probably won't be efficient for large documents that could be gigabytes in size, with many queries over them. A few of these systems handle the order problem, often by simply keeping the XML data as well as intermediate results in physical XML document order [13]. This however is based on their inefficient storage, which makes their systems a poor choice for the business community. For this reason, many systems [8, 21, 6] are beginning to use relational backends. Once XML documents have been mapped to relational schemas, a technique for order-sensitive query execution is then required. The translation of XML query languages to SQL while preserving order remains a challenging task. Some projects do a direct translation – syntax to syntax [2, 6], while other projects translate from syntax to algebra representations to syntax [8, 10]. Regardless of the translation strategy, the process is complicated when it comes to order-sensitive translation. Currently, only one project [15] attempts to solve this issue, and it is done by a direct syntax to syntax translation. While this method may solve an immediate need, if a new method of order storage was proposed, this group would have to recode their syntax to syntax translator. The Rainbow system was designed in part to solve the order problem and this thesis now focuses on these issues.

1.3 Problem Definition

Consider the following queries. “Return the second song title of every album”, “Return the first 3 band names of all bands from Boston”, “Of all the songs in a discography, return the sixth title.” These queries are almost impossible to create with a set-based query language, such as SQL, which has no concept of ordered sets. XQuery, however, utilizes ordered sets and can execute these queries. Providing a middleware from relational storage to XML representation that will allow us to create such queries in XQuery and execute them in SQL is the goal.

Before this goal can be reached, there are many research issues which must be solved. First, the question is if XML documents can always be generically decomposed into a relational structure, then generically be composed back to the original document without losing any information especially any order-sensitive information? Secondly, how can this order-sensitive information be stored within the relational database? Next, is it possible to push down all order-sensitive execution to the relational backend, or must certain parts be executed by the native system? If this order-sensitive execution can be pushed down, how can it properly be executed over a order-insensitive relational backend? Finally, what is the cost of this pushdown solution?

1.4 My Approach

In this thesis I am proposing to process and execute order-sensitive XQuery queries using the following steps. First XML documents and schemas must be loaded and stored in the relational database such that their order-sensitive information is kept. All required implicit order-sensitive information is made explicit. This explicit information must also be captured. To do this, an order-sensitive metadata table is created that contains this information for each schema and each chosen loading strategy. Order-sensitive user queries

must then be merged with queries that contain knowledge of these loadings in order to compose them into a complete query.

This merged query is then translated to the system-specific algebra tree, XAT (relevant algebra operators are discussed in Chapter 5). This query is massaged using equivalence rules in order to optimize the tree before SQL generation takes place. Algebra trees which contain order-sensitive position information are rewritten with extra metadata information. Rewrite rules exist to create special order-sensitive operators in their place. These special operators alert SQL generation that an order-specific query was issued. This is fully described in Section 5.5.1.

Once the tree is sufficiently optimized, as determined by the system, the operators which can be translated to SQL are translated. As each operator is translated, specific portions of an SQL statement are created. When the newly created order-sensitive operators are reached, special SQL templates are used and filled in with information from the algebra tree as well as information from the metadata table. The finalized SQL statement(s) are then executed over the relational database, with the results returned to our system. Finally Rainbow executes all the remaining non-SQL translatable operators in main memory. The final result is then returned to the user.

1.5 Contributions

Together with my addition to the Rainbow system, Rainbow now will be the first system to process order-based queries with algebra optimizations, not a syntax to syntax translation. Each contribution is listed specifically below:

1. Created different loading queries, based on 3 different order encodings

2. Created a framework to allow general mappings and general encodings

3. Created Rewrite rules for order-sensitive computation pushdown
4. Designed the metadata table for order-sensitive information management
5. Designed a method for general order-sensitive SQL generation via templates
6. Implemented this work as part of the Rainbow system
7. Designed and conducted experiments to test the order-sensitive system

1.6 Thesis Outline

The next chapter details the related work for the thesis. Chapter 3 gives background information on XML and XQuery, while Chapter 4 is a general overview of the Rainbow System. Chapter 5 discusses the XML Algebra Tree of Rainbow as well as Operator Rewrites. The following chapter then deals with order based loading queries and view queries. Chapter 7 describes order-based heuristics, while Chapter 8 describes order-based template heuristics. Chapter 9 discusses our system implementation. Experimental results are shown in Chapter 10. Conclusions and contributions are given in Chapter 11. The rest of the document is dedicated to Appendices.

Chapter 2

Related Work

2.1 XML Query Processing

The Kweelt [13] project is an XML query engine that uses Quilt [3] as the query language. Quilt, the predecessor to XQuery, is an order sensitive language. It can determine, for example, what the globally 3^{rd} element of the overall document is or what the 2^{nd} child is relative to a specific element in the document. Kweelt, however, does not support a relational database as backend, rather it works directly on native XML documents through a file directory structure. This is not a very sophisticated nor efficient document storage method. Also, since order is implicitly preserved throughout their system (it appears by physical sequential order in main-memory data structures), little can be learned from their project in respect to relational backends.

Another project with many of the same benefits as Kweelt is the Timber project [9] from the University of Michigan. This is a native XML query engine, such that the XML data is actually stored in a graph or tree structure on disk. Following W3C standards, the data is stored in the appropriate document order. This allows the query results to be returned correctly even when queries do not have order-specific predicates. Unfortunately,

this project does not rely on a Relational Backend but rather an object storage manager, Shore [14]. While there may be benefits to using Shore, a low-level storage medium, the Rainbow system would have to be redesigned from scratch. We instead aim to exploit relational technologies.

[22] is an XML query engine. This engine creates an algebra, which passes data tables between the nodes within the algebra. The current system, in order to handle order, simply creates a final column that is filled with integers that contain positional information. If however a query only requires one 2^{nd} value, the system would pull too many values out of the relational backend, only to discard a majority of them within a few steps. This causes resources to be greedily consumed, which further causes the system to slow down. This is not an efficient query system.

Many projects [2, 8, 24, 21, 6, 10] do use relational backends. They allow for the translation of a particular XML Query language into SQL for the querying of XML documents. However, these projects, for the time being, have not yet focused on the implicit order of these documents. The future work section of each of these papers lists document order as a task for the future.

The XPERANTO group in [15] is the only work we are aware of thus far that works with XML, SQL and the implicit order of the documents. In their paper, they describe how they directly translate XQuery statements into SQL ”order-based” statements, syntax to syntax. Part of the entire XPERANTO system uses algebra translation, but this portion of their work does not. By not using their algebra, XQGM, the SQL statements appear to be always ”hardcoded.” If an individual created a new mapping strategy, a Database Administrator would have to step in and create some possibly new strategy for this one to one mapping from XQuery to SQL. Rainbow avoids this by being more generic. Rainbow first translates the XQuery into an algebra, optimizes this algebra as much as possible, and then translates a majority of the algebra to SQL.

2.2 Order Based Queries

Attempting to change the nature of SQL in order to extend it towards supporting ordered queries is another possibility. In SRQL [12], adding the concept of position and order to relational databases is solved by extending the SQL operator set and by creating a hierarchical ordering strategy. This hierarchical ordering strategy extends the concept of a relational set to include a set of grouping attributes as well as a set of sequencing attributes. This project deals with ordering elements based on the grouping and sequencing attributes which are determined on the fly. Based on this on demand sequencing, all tuples are then ordered numerically. This ordering technique, however, is not appropriate for XML element ordering, especially when SRQL determines order based on the numeric rank of an element's data (in a manner similar to Temporal Databases, as discussed in the next section), not based on location within the original document.

2.3 Temporal Data Bases

Another possibility could be storing the XML in temporal databases. While temporal databases and temporal query languages focus on tuples with time stamps related to time order, XML documents contain document order. The two order concepts are in two different domains. Attempting to force XML documents to use time domains is too awkward of a concept [19], especially when considering the irrelevant details, such as time granularities (minutes versus hours) or the different possibly calendars. These details bear no relevance on document order nor on how a hierarchical document could be ordered in such a manner. Also temporal databases and temporal query languages are not supported widely in relational databases or relational query languages. For these reasons temporal database concepts are not ideal for XML order-sensitive query processing.

Chapter 3

Background

3.1 XML Schema

For the rest of the document, the XML schema which refers to Musical Records, as shown in Figure 3.1, will be used in examples. This schema is useful for explaining alternate mapping queries in Chapter 6. The full W3C XML Schema is shown in Appendix C.

```
RECORDLIST[
    SHORT_PLAY[
        BAND [string],
        LABEL [string],
        SONG [string]*
    ]*
]
```

Figure 3.1: Record List Schema in the Condensed W3C Format [18]

The schema in Figure 3.1, called the RECORDLIST schema, has a root RECORDLIST, with zero or more SHORT_PLAYs, each with one BAND of type string, one LABEL of type string and two SONGs of type string.

3.2 XML Data

```
<recordlist>
 <short_play>
   <band>Misfits</band>
   <label>blank</label>
   <song>Cough/Cool</song>
   <song>She</song>
 </short_play>
 <short_play>
   <band>Misfits</band>
   <label>Plan 9</label>
   <song>Bullet </song>
   <song>We Are 138</song>
 </short_play>
 <short_play>
   <band>Project X </band>
   <label>Schism</label>
   <song>SXE Revenge</song>
   <song>Shutdown </song>
 </short_play>
</recordlist>
```

Figure 3.2: Record List XML Document

The XML document in Figure 3.2 conforms to the RECORDLIST schema in Figure 3.1.

3.3 XQuery Statements

The XQuery [17] statement Q1 in Figure 3.3 that can be interpreted as, ”Return every second SONG of each SHORT_PLAY in the XML document, r.xml”, can be executed over the RECORDLIST schema to return some simple ordered information. The resulting XML document is shown in Figure 3.4. Since RECORDLIST has only three SHORT_PLAYs with SONGs in the 2^{nd} position, these three SONGs are returned. This query is an example of a query belonging to the *single step class,* where each order predicate only refers to a single step.

The user XQuery statement Q2 in Figure 3.5 is similar to Q1 except it has a small but important variation; Q2 does not group locally. Instead, Q2 “groups” over the root ele-

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

Figure 3.3: Order Sensitive XQuery Q1

```
<RESULT>
    <SONG>She</SONG>
    <SONG>We Are 138</SONG>
    <SONG>Shutdown</SONG>
</RESULT>
```

Figure 3.4: XML Document Result of XQuery Q1

ment as denoted by the parenthesis along the XPath (($\$record/SHORT\_PLAY/SONG$)). Given that $\$record$ is the root, this requires order with respect to the root. Query Q2 can be interpreted as "Return the fourth SONG of ALL the SHORT_PLAYs in the XML document, r.xml." The resulting XML document is shown in Figure 3.6. This query is an example of a query belonging to the *multi-step class*, where each order predicate refers to multiple steps.

```
<RESULT>
FOR $record in document("r.xml")
RETURN
    <SONG>($record/SHORT_PLAY/SONG)[4]/text()</SONG>
</RESULT>
```

Figure 3.5: Order Sensitive XQuery Q2

```
<RESULT>
    <SONG>We Are 138</SONG>
</RESULT>
```

Figure 3.6: Result of XQuery Q2

3.4 Default XML View

The default XML view (DXV) is the direct instantiation of the relational backend schema and data in XML format. A DXV is created by restructuring the original XML document so that it can be mapped in a one to one method to the relational backend. This is accomplished through various methods of mapping, as discussed in Chapter 6. The example shown in Appendix D is created by the Inline method with global ordering.

Chapter 4

Rainbow

4.1 System Overview

The Rainbow system is a complete XML data management system. It can load XML documents into relational databases. XML Queries are executed over the relational database after a translation to the XML Algebra Tree (XAT) and then to SQL. Before such query processing can be enabled, we must translate the XML schema into the relational schema, and secondly load the XML data into the relational schema. More details on this loading will be discussed below. Detailed accounts of each particular loading can be found in [4].

The Rainbow system architecture is shown in Figure 4.1. The system is composed of three major modules and a relational backend. The first being the Loading Manager. This part of the system is responsible for managing the flexible loading of the XML documents with schemas to the relational backend. The second major component is the Extraction Manager. This component is essentially the reverse of the Loading Manager. It extracts the loaded information from the relational database. The final component is the XML Query Engine. This component is responsible for creating system specific algebra trees from the user XQuery statements, optimizing these trees, and executing them in

conjunction with the relational backend. For each of the components, order plays a key role. Data is loaded with order knowledge and extracted with order knowledge. The algebra trees are also optimized with order-specific considerations.

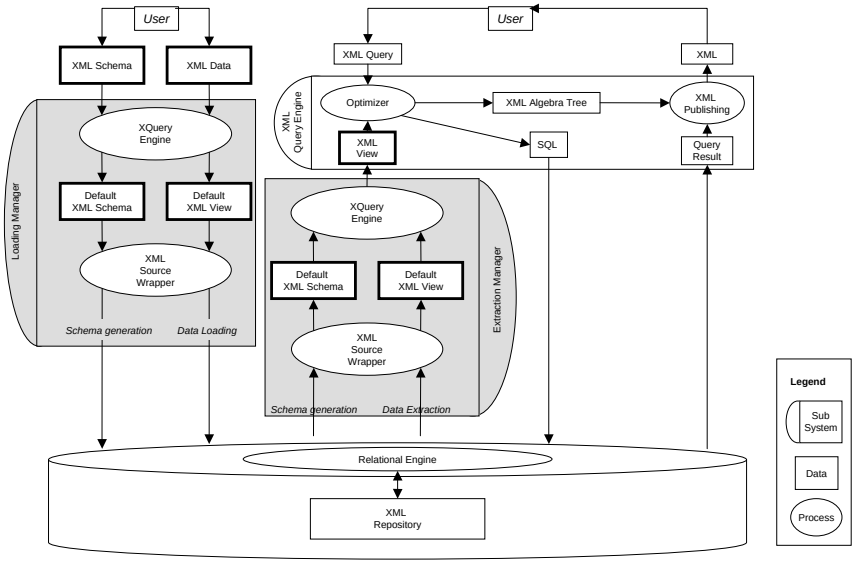

Figure 4.1: The Rainbow System Architecture

4.2 Order-sensitive Rainbow Modifications

For this thesis work, I had to update several modules within the optimizer and select portions of the overall Rainbow framework. First, view queries had to be segmented to become order-sensitive, as shown in Chapter 6.8. Then the optimizer required position-sensitive rewrite rules to be created, as shown in Section 5.5. The SQL module, as part of the optimizer, required order-sensitive SQL generation capabilities as well. To accomplish this, metadata information then needed to be stored within the Relational Engine (Section 6.6).

4.3 Data Flow of the Order-Sensitive System

The first step in the now upgraded Rainbow system is to load the XML document with its schema into the system. This is done by choosing a particular order encoding method (the explicit order-sensitive capture in a numeric or alpha-numeric format) and a data model mapping method (the method in which the XML document will be shredded and stored in the database). Both of these methods are written in XQuery. When executed over the original document and schema, a Default XML View is created. This view can then be directly loaded one-to-one into the relational engine. After the document has been loaded into the RDB with explicit order capture, a metadata table must be created that captures the explicit order information of the loading. With this loading, a particular view query must be created to extract the information from the DXV for the creation of the original document, again with order preservation. At this point, an order-sensitive user query can be issued against the view. The query engine would merge the view query with the user query. This merged query is translated into an algebra tree that can be rewritten. Special rewrite rules can then be applied to the order-sensitive portions of the algebra. When the tree is optimized, it will be translated to SQL statements. Order-sensitive portions of the algebra are translated via special order-sensitive SQL Templates. This newly created SQL statement is executed over the relational engine, which returns the resulting tuples to the Rainbow system. Rainbow then executes any remaining query operators that couldn't be translated to SQL. This final order-sensitive XML result will be returned to the user.

Chapter 5

XML Algebra and Rewrite Rules

The algebra for the Rainbow system is called XML Algebra Tree, or XAT [26]. An XAT is made up of algebra nodes, that are in part based on XPERANTO [2] as well as Niagara [8]. These algebra nodes operate over the data model of the system, the XAT Tables. For a full description of the algebra nodes, please refer to [26]. In this document, we only discuss a subset of algebra nodes, namely those that are most relevant for order-based queries.

5.1 XAT Data Model

The XAT data model consists of an order-sensitive table composed of order-sensitive columns and tuples where ordering among the tuples is essential. This table is called the XAT table. It is based on the W3C's XQuery 1.0 Formal Semantics, where every cell value of a tuple in a given column can consist of:

- an atomic value or
- a node: an XML element, XML document or an XML attribute or
- a collection: an ordered bag of items that contains atomic values or nodes.

Each column has a name. This name is associated with a binding, which is either created explicitly by the user query or is a temporary value created by the system.

5.2 XAT Binding Table

The XAT Binding hash table contains all one-to-one mappings from variables to XML elements, whether they are user created, such as $record, or system created, such as $col3. The bindings are used as keys of the hash, where each key's value is the XML element's path or an alias to another key. For example, in Query Q1 $record maps to "/RECORDLIST/SHORT_PLAY" and $col2 to "/RECORDLIST/SHORT_PLAY/SONG". This hash table is most frequently used for comparisons of variables within the rewrite rule component and within the SQL generation component.

5.3 XAT Operators

The following section briefly describes the operators that will be used for order-sensitive computation pushdown. A complete description of each operator can be found in [26].

5.3.1 Select

The Select operator has the same functionality as the SQL selection operator and is denoted by the $\sigma_c(s)$ symbol set, where c is an the expression and s is an XAT input table. The Select operator will filter all tuples from s based on the condition of the select expression, c.

5.3.2 Position Function

The Position operator is a function operator of type position. The function operator is also used for aggregates like SUM, AVG, COUNT and other functions that are derived from SQL or are strictly XML specific. The position function has no true SQL equivalent. This operator creates a new column in the XAT data table that contains incremental integer values that encode the relative order among the tuples.

5.3.3 GroupBy

The GroupBy operator is very similar to its SQL equivalent and is denoted by the $\gamma_{col[1..n]}(s, sq)$ symbol set, where $col[1..n]$ are the columns within the input table s to group over, and sq is the subquery that will be executed on each group. For all examples in this paper, the subquery for a GroupBy node will be a simple aggregation operator, and in most instances, it will specifically be the Position operator.

When the subquery of a GroupBy node is a Position function, the Position function will create numeric order values for each group. For the Query Q1 example shown in Figure 5.1, there are three SHORT_PLAY groups, with two SONGs each, the Position function in conjunction with the GroupBy node will create the following values $(1, 2, 1, 2, 1, 2)$. For the Query Q2 example as shown in Figure 5.2, the grouping will occur over the root element creating only one group, and the Position function will correspondingly create the following values $(1, 2, 3, 4, 5, 6)$.

When the subquery of a GroupBy node is an Aggregate operator, the aggregate operator turns all grouping tuples into one tuple.

5.4 XAT Decorrelation

When an XAT (XML Algebra Tree) is first constructed, it can contain zero or more FOR nodes with each FOR node having two subqueries. A FOR node is effectively an iterator modeling the semantics of the ”FOR” statement in an XQuery FLWR expression. The purpose of this node is “for each item returned from the inner branch, execute the outer branch.” In order to make the system more efficient, we need to push this computation down such that the inner branch is only executed once.

Figures 5.1 and 5.2 show the XATs before to decorrelation.

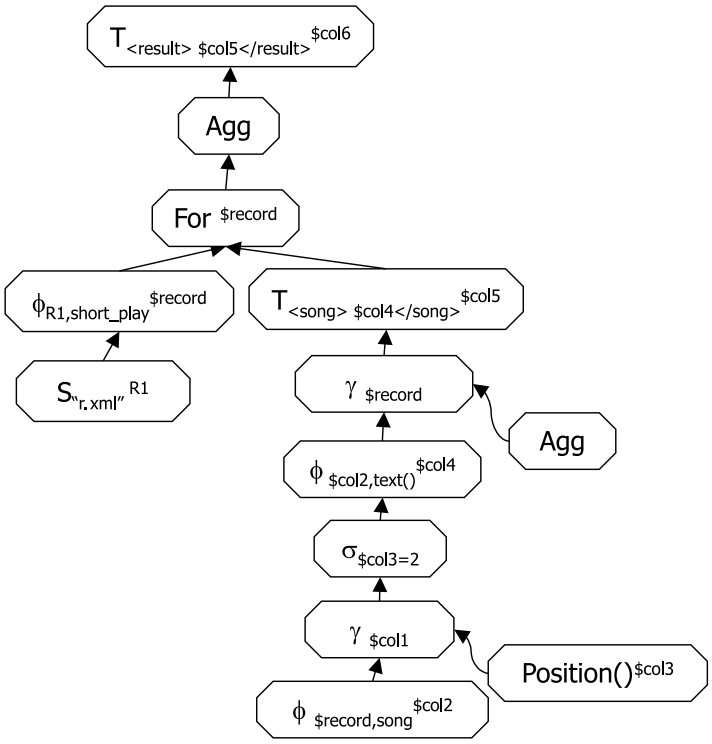

Figure 5.1: Order Sensitive XQuery Q1 as XAT Tree before Decorrelation

For the purpose of efficiency, upon decorrelation, the FOR node is removed and in most cases is replaced with a series of GroupBy nodes and aggregate nodes. In some circumstances the FOR node is replaced with a Left Outer Join node and a series of branches with Cartesian Product nodes. For both cases, instead of executing each FOR

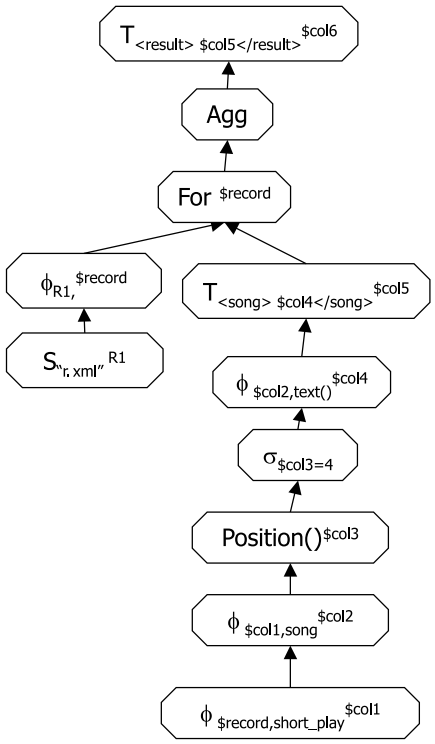

Figure 5.2: Order Sensitive XQuery Q2 as XAT Tree before Decorrelation

branch possibly multiple times, the branches are merged and are executed once, and the results are aggregated or joined. Then only the desired columns are projected out.

When function nodes are decorrelated, they become a function of a GroupBy node, and are grouped on the appropriate previous FOR binding(s). An example for Q2 is shown in Figure 5.3. An example is not shown for Q1, as that tree is already grouped over $record. Upon decorrelation, Query Q1 would attempt to create another GroupBy operator over the already present GroupBy operator. Since both GroupBy operators would be grouping over $record, the outermost GroupBy is redundant and not necessary. In this example, the outermost GroupBy was not generated for this reason.

A complete description of decorrelation can be found in [20]. Figures 5.4 and 5.5 show the XATs after decorrelation.

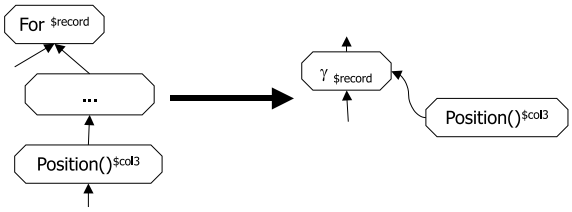

Figure 5.3: Decorrelation example with query Q2

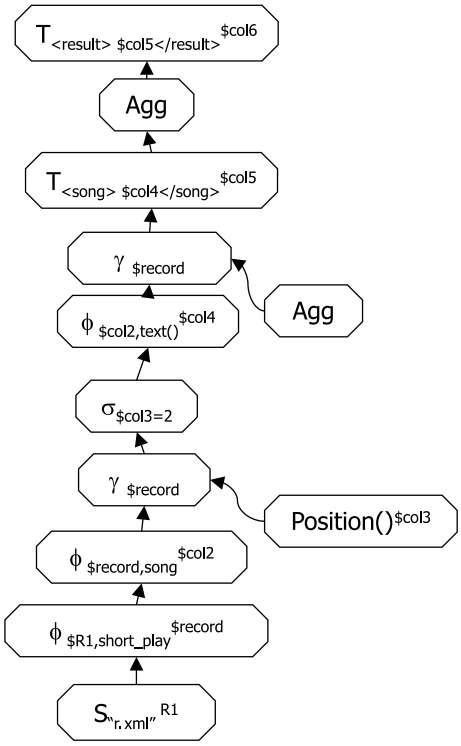

Figure 5.4: Order Sensitive XQuery Q1 as XAT Tree after Decorrelation

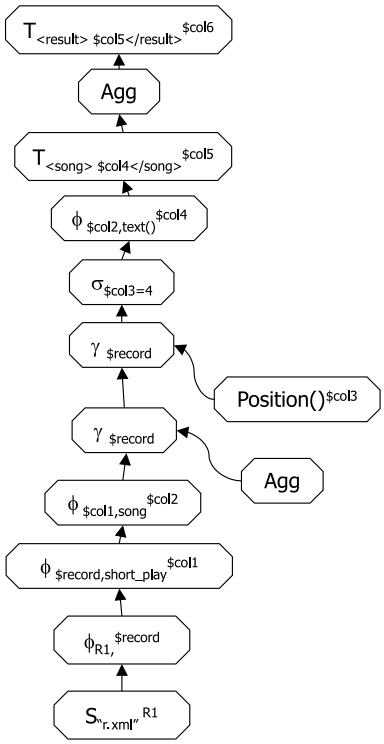

Figure 5.5: Order Sensitive XQuery Q2 as XAT Tree after Decorrelation

5.5 XAT Rewrites

As with all algebras, there are many equivalence rules that can simplify or vice versa make a tree more complex. Order-sensitive pushdown relies on a few key rules. These rules are briefly explained in the following sections. A full report can be found in [23, 25].

5.5.1 GroupBy/Function Replace

The most important rewrite rule for order-sensitive queries is the GroupBy/ Function replacement strategy which is one of the contributions of this work. This rule only applies to Position function nodes or GroupBy nodes with a Position() node subquery. It is only executed during the SQL generation step. That is, if SQL isn't generated then, there is no need for such a replacement as shown later in Chapter 7.4.

There are two cases for replacement. Both rely on where the Position() node is located

in the tree as illustrated below.

5.5.2 Single Step Order Rewrite

The first case is that of a Single Step Order replacement. During the initial tree generation, a Position() Function is created nested within a GroupBy node, as is natural for the algebra generation of a single step query.

This can be seen in Query Q1 by the following contained XPath: $\$record/SONG[2]$. At this point, $\$record$ represents SHORT_PLAY, which becomes the context for the SONGs to be grouped over.

In this case, the immediate child of the GroupBy node is a navigate node, demonstrating a need for the single step order replacement. This will always be the case for Single Step order queries. In this case, the GroupBy node, including the GroupBy subquery, is replaced with the SingleStepOrder node. This rewrite is shown in Figure 5.6.

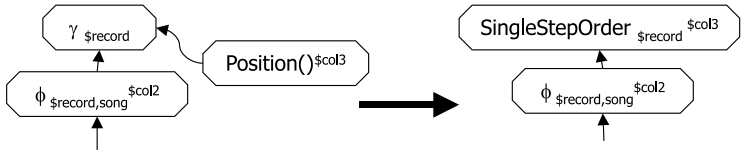

Figure 5.6: Q1 XAT Segment Rewriting

5.5.3 Multi Step Order Rewrite

The second case is for Multi Step Order replacement. In the tree generation step, a GroupBy G1 is created with an Aggregate node subquery. A partial tree is shown in Figure 5.7. This GroupBy G1 groups the entire document based on the XPATH of Query Q2: $(\$record/SHORT\_PLAY/SONG)$. The parent of this GroupBy is a Position() Function node, as shown by the next XPATH of QUERY Q2:
$(\$record/SHORT\_PLAY/SONG)[2]$.

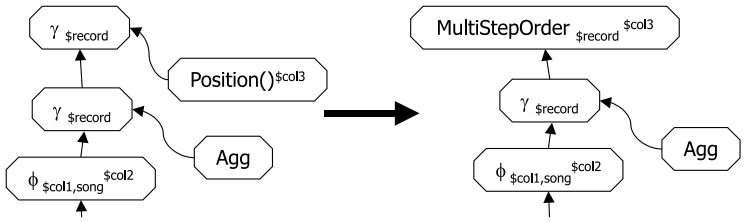

Figure 5.7: Q2 XAT Segment Rewriting

This is one strategy in which a Multi Step query tree can be generated. Then during Decorrelation, the Position() Function will be nested into a GroupBy, G2. This creates two GroupBy nodes in parent–child locations. The GroupBy G2 will be replaced. This rewrite is demonstrated in Figure 5.7.

The other instance where a Multi Step Order replacement is necessary is when there is only a Position() Function (not nested within a GroupBy node) even after decorrelation. In this instance, it is clear that no hierarchical grouping is necessary as is natural for Multi Step queries. Hence the Multi Step Order rewrite is required for this tree.

In both Multi Step Order rewrite cases, the GroupBy or Position() node and the GroupBy subquery, if the node is a GroupBy, is replaced with the MultiStepOrder node.

Parameters to these new functions are taken from the navigate child (in the first case of Multi Step Order, G1's child) as well as information gained from the Metadata table, as will be discussed in Chapter 6.6.

Chapter 6

XML Document Loading

6.1 Relational Storage

There are many strategies for mapping the XML document in Figure 3.2 to relations. The result of two example methods given in Figures 6.5 and 6.6 are shown in Section 6.4.

6.2 Order Preservation

The next three sections describe a sampling of ordering methods. Many more are available, but the three ordering methods chosen show the full range of order encodings. A guideline on general orderings follows the order encodings. In [15], the following order methods are discussed in full detail: local, global, and Dewey order. The full path order described in Section 6.2.3 is similar to the Dewey order but improved upon, as shown below.

[15] proved some interesting facts about order types. Through experiments, they showed that local order is the best method for storage in situations with heavy updates because the need to change many order values will be smaller, especially in instances of

sparse ordering. Sparse ordering would leave gaps in between each order value while still maintaining an ascending order (i.e. 1, 4, 7, ...). Only the order value for the immediate siblings will need to be changed.

Other experiments in [15] showed that global order is the best method for storage when a heavy query load and light update load is expected. It is better than local order in that it doesn't have to group siblings before ordering (where as for the local ordering method, each set of siblings must be ordered through their parents positions. This would recursively continue all the way to the root). Hence this reduces the complexity of the SQL query created. If a medium level of updates is expected, sparse ordering would be most effective.

6.2.1 Local Order

Local order is kept through sibling order. The root will have a value of 1, and its *n* children will have order values from 1 to *n* respectively and so on. The order numbers are not unique globally but rather only unique locally with respect to siblings regardless of element names. An example is shown in Figure 6.1.

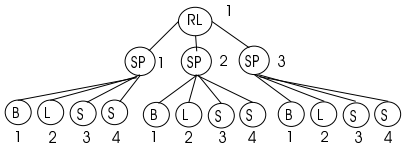

Figure 6.1: Local Order Encoding: Sibling Order Traversal

6.2.2 Global Order

Global order refers to globally unique position values. They are derived by a post order traversal strategy with a value of "1" for the root element and all the other nodes ordered consecutively based on the order visited. The root's left child will have the next value,

and its left child will have the next value, while the bottom most, right most element will have the maximum value. An example is shown in Figure 6.2.

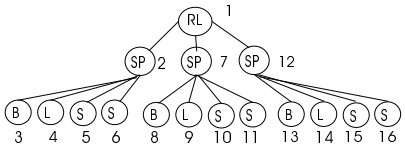

Figure 6.2: Global Order Encoding: Post-Order Traversal

6.2.3 Full Path Order

A full path order is kept by combining the local sibling order at each level of the hierarchy and concatenating these into one path key. This combines the concepts of sibling order as well as unique global order in an effort to maximize query efficiency for both heavy update workloads and heavy query workloads. An example is shown in Figure 6.3. This could be simple integers separated by dots, which is how it is represented by [15]. There are pitfalls with the integer-dot method, including creating special Oracle ”order by” operators. Instead of simply comparing integers, the dotted integers strings would be compared against each other in ASCII order. For this reason, ASCII order wouldn’t be correct as 1.2 would come after 1.10.

Another method would be to create large mostly zero filled integer strings. Instead, a lexicographical method as shown in [5] is utilized. This method uses sparse ordering regardless of the workload and exploits ASCII alphanumeric comparisons. By doing this, it avoids having to create new ”order by” classes in SQL. Lexicographical ordering also provides a strategy for non-cascading updates due to its sibling and sparse ”numbering.”

For example, if a new SONG is added into the second position of the first SHORT_PLAY in Figure 6.3, then the new tree would resemble Figure 6.4, where Sn is the newly inserted SONG.

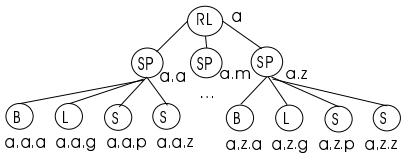

Figure 6.3: Full Path Order Encoding

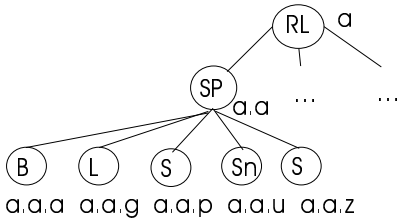

Figure 6.4: Full Path Order Encoding with Update Sn

6.3 Guidelines for Ordering Methods

In order for the Rainbow system to properly execute order based queries, the order value loading must adhere to the following guidelines. First, all order encodings must be lossless, such that all elements which require order values are given order values. Elements which require order values are a) elements with any number of children greater than zero and b) elements with schema modifiers, like * or +. Secondly all order values must be kept in ascending order, such that the natural ascending document order is captured. Finally, the numbering must be either numeric or alphanumeric, such that they can either be completely compared numerically or in ASCII order.

6.4 Loading

The process of loading XML data into a relational database must preserve all order-related aspects of the document. There are a few factors that must be kept in mind. First, the order encoding between the direct children of a parent node must be kept.

Secondly, all same type elements must be stored in the proper order. In the RECORDLIST

document, the SONGs must be stored so that they are in document order. Document order refers to how the SONGs are implicitly ordered within the document. For the first SHORT_PLAY, ”Cough/Cool” must come before ”She”, where in the case of local order, ”Cough/Cool” would get an order value of ”1,” and ”She” would get an order value of ”2.”

Finally, all siblings, similar or not, must also be stored in the proper order. For example, in the RECORDLIST document, the label element must come before any SONG element.

There are many ways to load a document. First, two ways that were used in the experimental section are described below. Secondly, we give a description of how general loadings are handled.

6.4.1 Inline Loading

Inline loading strategy [7] is used to keep the number of tables in the relational backend to a minimum. For this loading, most children (ones that have no sub elements, or fewer than 4 subelements with no subelements of their own) of one element are loaded into the same table. An example of this inline loading is shown in Figure 6.5.

The table in Figure 6.5 can even be loaded more compactly. If it is known, by the schema, that there are always 2 or less SONGs, these values can be inlined into the main table. The SONG elements can then be relabeled SONG_1 and SONG_2 so that order is preserved. They are labeled in such a way as to preserve the original document order; most importantly to indicate which SONG came first and which SONG came second.

If, for instance, each SONG in the document had subchildren of type lyrics_by, where many individuals had helped to contribute to the lyrics, the SONG element could not be inlined into the base table. This is because the amount of columns necessary would be unknown, and could in fact vary for each SONG.

6.4.2 Edge Loading

The edge loading strategy [7] requires the fewest tables: one. This table will have 4 columns only. The first column is the SOURCE column. This column is equivalent to a Parent ID column. The second column is the POSITION column. This column contains the explicitly created position information. The third column is the NAME column. This column stores all XML tag element value or attribute type values. For our loading with the RECORDLIST document, this column would contain: (RECORDLIST, SHORT_PLAY, BAND, etc). The final column is the TARGET column. This column serves two purposes. First, it contains the data value of any tag element or attribute data value. For instance, in our loading it would contain: (Misfits, Blank, She, etc). The second purpose of this column is to contain a reference between elements and their children. For our example, the first element RECORDLIST would contain the value ”1.0” in the TARGET column. This information could then be used to join with its child’s SOURCE column values, where every SHORT_PLAY SOURCE would equal ”1.0.” The full example of this loading is shown in Figure 6.6.

This loading is useful for many reasons. First, the amount of redundant data is minimized, considering the only redundant data (not including any strings within the XML source which may be the same) is the SOURCE and TARGET join information. This is normally loaded as a small sized float. Since all the data is stored in only one table, when updates are applied, the action is less expensive due to only touching one table. This mapping does have a few drawbacks however. The self joins that could be required to recreate the document can be very expensive. Also the loading is not as intuitive as the previous inline mapping, which was very straight-forward.

```
Table recordlist
IID PID ORDER
 1   0   1

Table short_play
IID PID ORDER   band_PCDATA  label_PCDATA
 2   1   1      Misfits      blank
 3   1   2      Misfits      Plan 9
 4   1   3      Project X    Schism

Table song
IID PID ORDER  song_PCDATA
 5   2   1      Cough/Cool
 6   2   2      She
 7   3   1      Bullet
 8   3   2      We Are 138
 9   4   1      SXE Revenge
 10  4   2      Shutdown
```

Figure 6.5: Relational Inlining Strategy with Local Order

```
Table EDGE
SOURCE  POSITION   NAME        TARGET
 0.0     1.0       RECORDLIST  1.0
 1.0     2.0       SHORT_PLAY  2.0
 2.0     3.0       BAND        Misfits
 2.0     4.0       LABEL       Blank
 2.0     5.0       SONG        She
 2.0     6.0       SONG        Cough/Cool
 1.0     7.0       SHORT_PLAY  7.0
 7.0     8.0       BAND        Misfits
 7.0     9.0       LABEL       Plan 9
 7.0    10.0       SONG        Bullet
 7.0    11.0       SONG        We Are 138
 1.0    12.0       SHORT_PLAY  12.0
12.0    13.0       BAND        Project X
12.0    14.0       LABEL       Schism
12.0    15.0       SONG        SXE Revenge
12.0    16.0       SONG        Shutdown
```

Figure 6.6: Relational Edges Strategy with Global Order

6.5 General Loading

For Rainbow to handle any loading, only one thing is required: the mapping must be able to create a non-recursive default XML view. We made the assumption here that all input XML documents are non-recursive. Any loading with a default XML view that adheres to this can be managed by the Rainbow system.

6.6 Metadata Table

Another part of loading is the creation of a metadata table for storing order information. One issue of this is the loading strategy used while a second orthogonal issue is the strategy selected for retaining order. This information must be captured so that order-based user queries can be executed correctly regardless of how the data is stored in the relational schema. This information will be stored in the metadata table for each XML Schema. The metadata table is created in parallel with the document loading. An example metadata table is shown in Table 6.1.

XML PATH	DXV Order	Order Loading State	Order Context	AorN
RECORDLIST	RECORDLIST/RECORDLIST.ROW/POSITION	fully preserved	RECORDLIST/RECORDLIST.ROW/PID	a
RECORDLIST/SHORT_PLAY	SHORT_PLAY/SHORT_PLAY.ROW/POSITION	fully preserved	SHORT_PLAY/SHORT_PLAY.ROW/PID	a
RECORDLIST/SHORT_PLAY/BAND	SHORT_PLAY/SHORT_PLAY.ROW/POSITION	fully preserved	SHORT_PLAY/SHORT_PLAY.ROW/PID	a
RECORDLIST/SHORT_PLAY/LABEL	SHORT_PLAY/SHORT_PLAY.ROW/POSITION	fully preserved	SHORT_PLAY/SHORT_PLAY.ROW/PID	a
RECORDLIST/SHORT_PLAY/SONG	SONG/SONG.ROW/POSITION	fully preserved	SONG/SONG.ROW/PID	a

Table 6.1: Order-Based Metadata Table for the Inlined Recordlist Document

In Figure 6.1, the first column of the table will store the XML Path of all the XML elements of the original document. For our example, the full XML path is stored for each element: (RECORDLIST, SHORT_PLAY, BAND, etc) as (RECORDLIST, RECORDLIST/SHORT_PLAY, RECORDLIST/SHORT_PLAY/BAND, etc).

The second column will store the default xml view (DXV) element path of the relational column that stores the selected order encoding values.

The third column will store the ordering state in which the document was ordered. This can be one of two values. In the example table above, the value for each element is “fully preserved”. This means that the document will only require one single order encoding value to recreate the document in its entirety. A value of “context required” would be for any other order encodings that cannot fully preserve the document order with one value. This holds for the case of Local Order.

The fourth column will contain all ordering context elements, which in some cases could be more than one item. For the case of multiple elements, the elements will be stored as a string, delimited by the ’:’ character. This item will be especially necessary for the grouping of elements in order to achieve the correct tuple output during the single step order SQL generation phase, described below. This column may be required for certain multi step queries depending on the particular XPATH with multi step information in it. In our multi step example, this path includes the root. Hence there is no need to use this column’s information for grouping.

For example, the fourth column will contain multiple items for a local order loading. This case requires that each element be properly nested by joining on parent IDs and ordering over each hierarchical level of the document.

The final column indicates whether the order values are numeric or alphanumeric. If the order value is numeric, a ”N” value will be present. An ”A” will be present for alphanumeric order values.

The metadata table shown in Figure 6.1 is designed for an inlined table mapping with full path ordering scheme. The inline mapping can be seen by the second columns use of two tables (one for SONGs and one for the other elements), the full path is indicated by the third column, ”fully preserved”, and the final column is set to ”a” for alphanumeric.

At this time, this table is created manually for each XML Schema and loading, though conceptually, it would be possible to automate this step.

6.7 Query Composition

In the system, the user query is not in itself sufficient to produce the result (unless the user has a Database Administrator's knowledge of the system and loading methods used), as all the information is stored in a relational database and not in an XML document. In order to execute a user query properly over the relational backend, the user query and the view query must be merged. View queries are discussed in Chapter 6.8. The user is assuming the query is over the original XML document, but this document is not available. Instead the view query virtually creates this document. There are two possible strategies to accomplish this. First, execute the view query, create a temporary XML document, and then finally execute the user query. A better method which doesn't utilize a two step execution procedure requires the user query to be merged with the view query. With this method, no materialized view of the document needs to be dynamically created which improves the system's performance.

6.8 View Queries

In order to properly execute an ordered-based query over an XML view, that view must be created (or seemingly created) from the relational database information and sorted appropriately based on the explicitly captured XML document order, making the order-sensitive knowledge implicit once again. With this knowledge, the XML document can be reconstructed.

Instead of reconstructing the document, however, the view query is translated into a XAT tree and merged with the user query, as described in Chapter 6.7. After the two have been composed, the tree can be simplified and optimized by the rewrite equivalence rules.

A general view query exists for each type of loading discussed in [4]. However, these queries are currently too complex for the first prototype version of the Rainbow system,

```
<RECORDLIST>
FOR $playMap in DXV()/SHORT_PLAY/SHORT_PLAY.ROW
RETURN
    <SHORT_PLAY pos=$playMap/POSITION/text()>
    FOR $SONGMap in DXV()/SONG/SONG.ROW[PID/text() = $playMap/IID/text()]
    RETURN
        <SONG pos=$SONGMap/POSITION/text()>
            $SONGMap/SONG_PCDATA/text()
        </SONG>
    SORTBY (./@pos ASCENDING)
    </SHORT_PLAY>
SORTBY(./@pos)
</RECORDLIST>
```

Figure 6.7: View (Extraction) Query for Q1 (Figure 3.3) with Inline Strategy

as they require as of now unimplemented functions as well as recursion. For this reason, queries with schema knowledge, called instantiated queries, are used in our current effort.

Instantiated view queries are partially pre-optimized queries with schema knowledge. These queries are generally more compact, and do not require recursive calls. The XQuery shown in Figure 6.7 is an example of an instantiated view query. This query would retrieve every SONG of each SHORT_PLAY from the default xml view (DXV) shown in Appendix D. It would then correctly order and nest the elements, so that the query result resembles the original XML document (without the BAND or LABEL elements which are unnecessary for both Q1 and Q2).

Chapter 7

Order-based Methodology

The main purpose for algebra rewrites is to push the SQL-executable computations down to the SQL engine, while all XML specific nodes remain at the top of the tree. All SQL-executable nodes should be pushed down as far as possible in order to be translated to SQL statements. SQL-executable operators are operators with clear SQL equivalents; XML select becomes SQL where operator, XML Sortby becomes SQL ”Order By” and so forth. This process is further complicated when the order-sensitive query is merged with the view query, creating a larger and more complex query. To make this composed query efficient, all SQL-executable nodes need to pushed down as far as possible. These SQL-executable nodes can then be translated to SQL and executed over the relational backend.

7.1 The Order-Sensitive XQuery

When a tree, as shown in Figure 5.1, is found to have a GroupBy/Position node, this node can be swapped out and replaced with a special order-sensitive node based on the location of the original GroupBy/Position node and its children as discussed in Sections 5.5.2 and

5.5.3.

7.2 Position Operator Replace

After the GroupBy/Position operator has been found and removed, the appropriate order node, be it SingleStepOrder or MultiStepOrder, will be inserted in the place of the original GroupBy/Position as shown in Figures 5.6 and 5.7. The parameters from the original GroupBy/Position operators are not lost. They are kept as the parameters to the SingleStepOrder or MultiStepOrder function. The other parameters to this new node will now be determined, as discussed below.

7.3 Metadata Table Query

An SQL statement will be created to query over the metadata table as part of the Position Operator rewrite. This statement will then query over the metadata order table by selecting the matching XML Path from the immediate child Navigate operator, as this operator is the one in which we want to capture order information. The DXV order element path is then projected from the metadata table, along with the rest of the tuple. The metadata query for Q1 is shown in Figure 7.1.

```
select DXV_ORDER
from RECORDLIST_METADATA
where XML_PATH = “RECORDLIST/SHORT_PLAY/SONG”
```

Figure 7.1: Metdata Query for Q1

The node is now complete with the newly produced parameters and the parameters from the original GroupBy/Position node. The tree is then passed to the SQL generation component as discussed in Chapter 8. A rewritten example of each order-sensitive optimized user tree is shown in Figures 7.2 and 7.3.

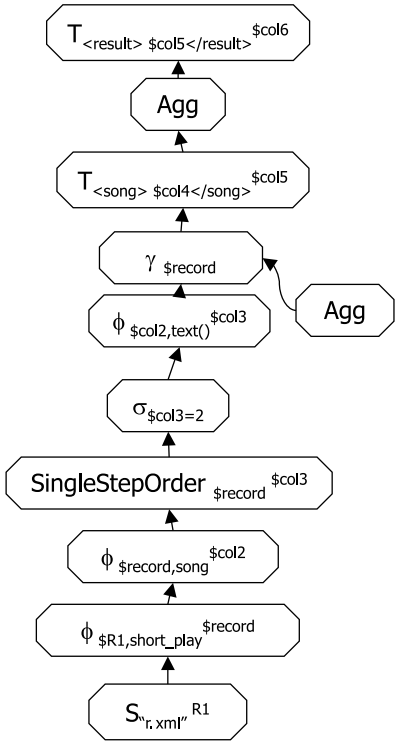

Figure 7.2: Query Q1 as XAT Tree with Rewrite

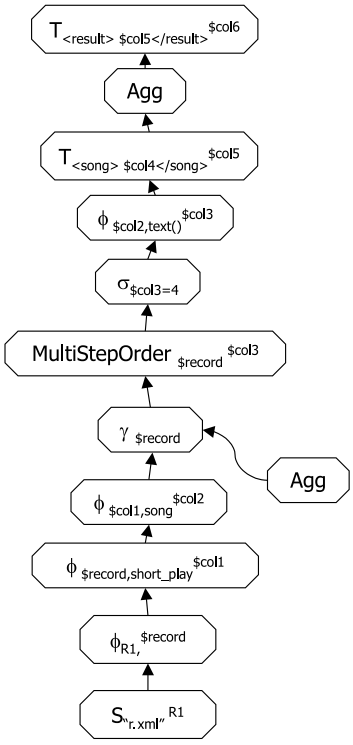

Figure 7.3: Query Q2 as XAT Tree with Rewrite

7.4 Main Memory Position Execution

There may be cases where SQL generation is not desired, particularly if the data is not stored in a relational engine. This can occur when the original XML document is stored locally or in a native XML document repository. For such a case, the GroupBy/Position operator should not be swapped out or pushed down to SQL generation, but instead should be executed using our native XML query processing strategy. This execution is straightforward. The position operator creates a new column in the data model, is grouped appropriately, and then the XAT table cells are filled in with the appropriate ordered values as created by the Position operator.

Chapter 8

Order-based Template Heuristics

SQL generation is an incremental process driven by bottom up tree traversal. As one operator is visited, an appropriate SQL statement fragment is created. The Navigate, Select and other SQL-executable nodes are all translated to SQL. SQL-executable XAT nodes are those which have equivalent SQL operators. For the most part, this is a direct translation of operators to SQL statements. However, when order is of interest, other order-specific pieces must be added to the SQL statement. Finally, when the tree traverser reaches an operator that cannot be executed in SQL, the traverser stops and creates a SQL Statement operator.

To create the order-specific pieces of the SQL query, a template is used. The grammar for the order-sensitive SQL template is shown in Figure 8.1. This grammar adheres to Oracle's "ordered" query lingua. Other templates would need to be created to cover other proprietary DBMS that handle similar order-based statements.

```
TEMPLATE:
SELECT <ELEMENT>,+ row_number() over
(<PARTITION>? <ORDERBY>) $position_function_binding
FROM <TABLE>+

PARTITION:
partition by <ELEMENT>

ORDERBY:
order by <TONUMBER>|<ELEMENT>

TONUMBER:
to_number(<ELEMENT>)

TABLE:
table name | TEMPLATE

ELEMENT:
element name
```

Figure 8.1: SQL Grammar for Order-Sensitive Queries

8.1 Order-Sensitive SQL grammar

The template is based upon 6 rules, which are each fairly simple. The following template is based upon Oracle's SQL rules, but other templates can be created for other DB specific SQL versions. The items that are special to Oracle are the analytical function $row\_number()$ (which creates integer values in the same fashion as the XAT position function), $over$ method (which tells the analytical function what values to work with), and the $partitionby$ phrase (which creates groups or partitions based on the element it is given).

The key grammar rule is the TEMPLATE rule. This rule is the key root rule for all the other elements. Algebra trees will fill out the template in specific ways, depending on the order-sensitive specific operator, whether it is SingleStepOrder or MultiStepOrder and the parameters for this operator.

The key rule to denote this difference is captured by the <PARTITION>? rule. Not all queries require partitioning information, hence the '?' at the end of the rule within the TEMPLATE rule. More specifically, relational tables that are encoded with a "fully

preserved" order schema do not need partitioning with multi step queries, whereas multi step queries with a "context needed" parameter will require nested partitioning. Single step queries are similar, except they always require partitioning. In the instance of a "context needed" parameter, Single step queries would also require nested partitioning.

The $\$position\_function\_binding$ is the binding from the algebra tree. For each of our examples, the $\$position\_function\_binding$ is $\$col3$. This binding is then constrained outside of the template within the WHERE clause of the remaining SQL query. For example $\$col3 = 2$. A complete example is shown in Section 8.2 in Figure 8.2.

8.2 Template Completion

During the SQL generation phase, the template will be filled in. Some sections will be filled in based mostly on the metadata table information. The new order-based query node contains valuable information essential for the filling of the template. We now walk through the example of Q1 for a complete explanation.

XML PATH	DXV Order	Order Loading State	Order Context	AorN
RECORDLIST/SHORT_PLAY/SONG	SONG/SONG.ROW/POSITION	fully preserved	SONG/SONG.ROW/PID	A

Table 8.1: Condensed Order-Based Metadata Table for the Inlined Recordlist Document

The row shown in Figure 8.1 is the one that matches the metadata query for Q1 as shown in Figure 7.1. From the "Order Loading State" column, we can see that the information was loaded with a fully preserved order. This indicates to the template that there is no need to partition over any context to produce the correct order of the result. However, since the query Q1 is a single step query, there is still a need to partition over the Parent IDs in order to get each 2^{nd} SONG. The "AorN" column shows that the order values are alphanumeric, so there is no need to use the TO_NUMBER() function. We also know from the DXV Order column where the SONG order column is. With this raw information, the template construction is nearing completion.

The first step for template completion is to take the raw strings derived from the meta-data table, and put them in an appropriate SQL executable format. This task is simple. Since the DXV stores each of the elements in the following format:

”TABLE_NAME/TABLE_NAME.ROW/ELEMENT”,

the transformation is fairly simple. By creating a reverse lookup within the Binding Hash table, we can use the XML path to return a $binding. Then within the SQL generation portion, there is a local hash structure that maps each binding to its ’RDB equivalent,’ such as $\$col2 -> \text{"}\$song\text{"}.POSITION$, where $\$col2$ is the binding and $\text{"}\$song\text{"}.POSITION$ is the ’RDB equivalent.’ This ’RDB equivalent’ is the desired SQL executable format. Finally, we add each ’RDB equivalent’ to its appropriate location within the template, as shown in Figure 8.2. The double quotes around the variables are required. In Oracle, the $ symbol is a reserved symbol.

```
row_number() over (
    partion by ”$song”.PID
    order by ”$song”.POSITION
) $col4
```

Figure 8.2: The Template for our Example Filled

Once the template is complete, the rest of the SQL generation steps continue. SQL generation will continue to consume nodes and create more SQL fragments. The tagger and aggregate operators however, are not converted into SQL, as they have no SQL equivalents. These two nodes, as well as a few others, must be executed within the Rainbow engine proper.

When the SQL generation step has finished, a new node is created in the tree, a SQL Stmt node. The nodes that were used to create the SQL statement and template will be deleted from the tree and replaced by the SQL Stmt node, which is simply one node that contains an SQL string, prepared for execution. An example of Q1 in the finalized format is shown in Figure 8.3.

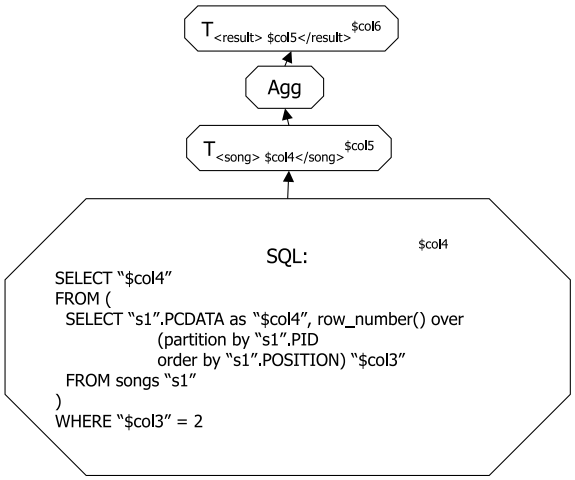

Figure 8.3: Query Q1 from Figure 7.2 after SQL Generation

If the query was instead a multi step query, as in Q2, the template and resulting SQL statement would look differently, as would the tree. In the multi step case, no partitioning would be required (as shown by the algebra tree in Figure 7.3). The query is attempting to find the "global" 2^{nd} SONG so no grouping is required. Hence no partitioning information is required either. The $orderby$ information is still required, as is the row_number and $over$ method. This example is shown in Figure 8.4.

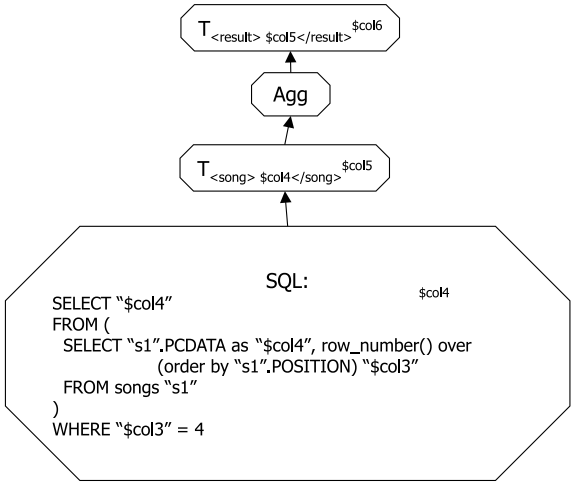

Figure 8.4: Query Q2 from Figure 7.3 after SQL Generation

8.3 General Template Discussion

In the previous section, we want to point out that our procedure does not rely on any hard-coded query statements for specific loadings or specific mappings. The system was designed to be general, to handle a large range of loadings or order encodings that comply with the previously discussed guidelines from Sections 6.5 and 6.3. In fact, there is only a small difference between the Query Q1 when designed for an inlined loading versus an edge loading. The SQL statement for Q1 query over an inline loading is shown in Figure 8.5, while the example with an edge loading is shown in Figure 8.6. The similarity confirms the general nature of our solution.

```
SELECT ”$col2”
FROM ( SELECT ”s1”.PCDATA as ”$col2”, ROW_NUMBER() OVER
      (PARTITION BY ”s1”.PID
      ORDER BY ”s1”.POSITION) ”$col3”
      FROM SONG ”s1”
      )
WHERE ”$col3” = 2
```

Figure 8.5: Query Q1 over an Inline Loading with Alphanumeric Order

```
SELECT ”$col2”
FROM ( SELECT ”s1”.TARGET as ”$col2”, ROW_NUMBER() OVER
      (PARTITION BY ”s1”.SOURCE
      ORDER BY ”s1”.POSITION) ”$col3”
      FROM SONG ”s1”
      WHERE ”s1”.NAME=”song”
      )
WHERE ”$col3” = 2
```

Figure 8.6: Query Q1 over an Edge Loading with Alphanumeric Order

Chapter 9

Implementation

The Rainbow system is implemented with Java JDK 1.2, JDBC and Oracle 8*i*. The major software engineering control behind the system is the visitor pattern. This allows pieces to be removed or replaced without the entire code needing to be updated. Considering there are over 300 files that combined are over 1.2 M in size, this is a very convenient pattern.

Order and order based queries are a subset of the system. To properly process and execute, I had to update many components, though the most order-sensitive work went into the rewrite rules, SQL generation, and various functions, such as SINGLESTEPORDER, MULTISTEPORDER and TRIM. The combined total for all of these methods and classes that I've added is more than 1200 lines of code.

For the rewrite rules, a new class was added, "RewritePositionRules." To this class, three methods were added. Following the visitor pattern with two parameters, a *visit* method was added for each of the singlestep and multistep configurations: GroupBy and Navigate – SingleStep; GroupBy and GroupBy – MultiStep; Position and Navigate – MultiStep.

Each of these methods must then communicate with the metadata table in order to

get the appropriate information for the newly constructed order-sensitive node. This required a JDBC connection to an Oracle 8.1.7 or higher backend, not the 8.1.5 version of Oracle. The 8.1.5 version did not have some of the appropriate functions, in particular row_number().

For SQL generation, a method called getStatement() was added in order to allow for SingleStep and MultiStep template creations. This method reads fragments of queries and translates them to SQL. I updated this code so that it could translate to the appropriate order-sensitive query.

Functions were also added to the Rainbow Engine. First functions for both SingleStepOrder and MultiStepOrder were added. These functions are currently stubs because as discussed before, they should never be executed in native. Later, functions were also added for basic XQuery commands, such as TRIM. Also a function was created to produce the Lexicographical key values used in the full path order type.

In general, many things needed to maintained and updated throughout the Rainbow system. As a senior member of the Rainbow Core team, my main focus was within the Decorrelation component. I also worked on fixing bugs in every sector of the code, from the Binding Hash Table, to Expressions, to the Execution component, where many bugs were fixed. Many weeks and months were spent testing, fixing and testing again.

Decorrelation required the most initial effort. The original design for this component was too incomplete in that it was missing cases, and some of the cases it covered were incorrect. I updated it so that it would effectively work with the new cases discovered, and the previous cases that were incorrect. Later on, the group leader revised all the cases and redid it once again. At this point, the decorrelation component is stable.

The Execution component had many small bugs that needed to be worked out. The most frustrating set of these bugs was in the GroupBy operator. The major problem was tricky, in that it would group the first items properly, but every set of items following that

were not grouped. It turns out that the original designer of the code was comparing every item to the first item instead of incrementing the comparison item. With that fixed, the GroupBy operator worked as intended.

The Expression package had a problem related to the comparison of order elements as described earlier. At a certain point, all values were getting compared in ASCII order, regardless of if they were strings or numbers. This was resolved by changing the cascading class cast exception catching to one of creating new value specific objects, and cascading the catching of number format exceptions. The comparisons now work as designed.

One other problem with the Expression package was the callbacks created in SQL generation. During SQL generation, when an expression was needed in a WHERE clause, the SQL generation package would call back to the Expression package for expression SQL phrasing. Within this callback there was a Binding Hash Table lookup for the particular terminal parts of the expression. However, when the Binding Hash table code updated, it no longer kept the bindings of elements in Expressions. Therefore, when the callback was executed in SQL, the Binding Hash table was called upon, and a null pointer exception was thrown since the terminal did not exist within the Hash. I fixed this by changing the method in which the callback was executed. Now it no longer requires information from the Hash, and it can create the terminal column by itself.

Beyond the Java coding, with the XQuery language, 6 loading queries were designed as well as 2 view queries. The loading queries were based on work done by [4]. More than 10 test queries were created with XQuery.

9.1 SQL Generation

The implementation of SQL generation available initially in Rainbow was rather naive. The generation is an incremental process. As the traverser goes from the bottom source

nodes to the upper tagger nodes, each operator is consumed. The semantic information from each operator is then put into a system of flat structures. The flat structures consist of vectors, one for each piece of required SQL: SELECT, FROM, WHERE, HAVING, ORDERBY, GROUPBY, DISTINCT and FUNCTION. Once a tagger node is reached, each vector is unrolled to create one SQL statement. This process however loses valuable hierarchy information about the query, most importantly when to join tables and when to apply selection operators.

The examples used throughout this paper execute properly and create the appropriate results with the initial naive generation process. This is due to the fact that all the queries want leaf-most elements, and do not query about siblings like BAND and SONG in the same query. If queries were executed that asked for same nested siblings, inappropriate joins would be created.

As an example, consider a new query, Q3, as shown in Figure 9.1. This query is looking for all the SONGs of the 2^{nd} SHORT_PLAY. The result of this query is shown in Figure 9.2.

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

Figure 9.1: Order Sensitive XQuery Q3

```
<RESULT>
    <SONG>Bullet</SONG>
    <SONG>We Are 138</SONG>
</RESULT>
```

Figure 9.2: XML Document Result of XQuery Q3

When this query is executed within the naive SQL generation over an Inlined Global loading, an incorrect SQL statement is created due to this flat structure system. The SQL statement is shown in Figure 9.3.

This statement is incorrect because the SHORT_PLAY and SONG tables are merged too soon. When these tables are merged on PID and IID values and then ordered, each 2^{nd} row_number() will now refer to 2^{nd} SONGs instead of 2^{nd} SHORT_PLAYs. The

```
select songpc, num
from (
       select short_play.iid as shid,
              song.pid as spid,
                   song.POSITION as spos,
                   song.SONG_PCDATA as songpc,
                   row_number() over
                (
                            partition by short_play.pid
                           order by to_number(short_play.position)
                      ) num
           from short_play, song
           where short_play.iid = song.pid
)
where num = 2
ORDER BY spos
```

Figure 9.3: Q3 Incorrect SQL Statement for Inline Global Loading

appropriate SQL statement is shown in Figure 9.4.

```
select song.SONG_PCDATA
from (
        select shid
        from (
                select short_play.iid as shid, row_number() over
                (
                        partition by short_play.pid
                        order by to_number(short_play.position)
                ) num
                from short_play
        )
        where num = 2
), song
where shid = song.pid
ORDER BY song.POSITION
```

Figure 9.4: Q3 Correct SQL Statement for Inline Global Loading

The statement in Figure 9.4 joins properly. It first creates the groups over SHORT_PLAYs, orders appropriately and then finds the 2^{nd} SHORT_PLAY. At this point, the join is created with SONGs, with the appropriate conditions. This SQL statement then produces the correct result. This statement captures the hierarchical information from the tree that was lost in the flat structure query.

In short, if SQL statements were generated in a true incremental fashion with one operator for every created portion of an SQL statement, the final SQL statements would

be correct. The current implementation can be extended such that the flat structures are reused, where common nodes can create SQL statement chunks that can be joined properly together to form the final SQL statement. The flat structures could be used for groups of operators, but when an operator is reached that could cause a different nesting, such as a join, the flat structures would create their piece of SQL and nest it properly with the other SQL statement chunks.

Chapter 10

Experiments

10.1 System Setup

The charts in the experimental section are based on experiments run on a Windows 2000 machine with a 1.2 Gig processor and 512 Megs of RAM. The experiments were run during times of minor process load.

The data loaded was generated by a random XML data generation script. Thirteen documents were created that all comply to the schema shown in Appendix C. The first document has 100 SHORT_PLAY elements, with 600 total SONG elements. The second document has 200 SHORT_PLAY elements, where the first 100 elements are the same as the first document. This second document has 1200 SONG elements, where the first 600 are also the same as the first document. This generation of documents continued to the thirteenth document which has 1300 SHORT_PLAY elements, and 7800 SONG elements. Of these 7800 song elements, there are 650 distinct song titles. The selectivity of the song distribution is random.

Two loading methods were used, the inline loading and the edge loading. For each loading, two different order encodings were used: global and local. This gives four total

test cases.

The queries executed for the experiments are Q1 and Q2, as shown in Figures 3.3 and 3.5 respectively. Each query was executed over the described above loadings ten times. The result of these ten tests were averaged for the final result.

The Y axis of each chart is time in ms. For the first set of charts the X axis displays how many SHORT_PLAYs were loaded. The final four charts vary the selectivity of the amount of SONGs returned by the query. This was done by creating queries almost identical to Q1 and Q2, but instead of using the binary operator $=$, the operator $<=$ was used. For instance, for Q1, the first bar in the chart represents all SONGS of position $<=$ 1, or 1300 SONGs.

10.2 Experimental Evaluation

10.2.1 Single Step Query Experiments

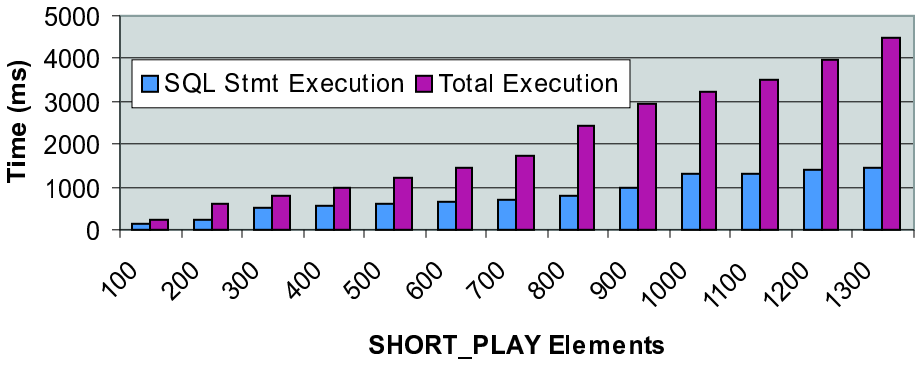

Table 10.1: Query Q1 Executed with Global Edge Loading

Table 10.2: Query Q1 Executed with Local Edge Loading

Tables 10.1 and 10.2 show comparisons between SQL statement execution and total execution times measured in ms for Query Q1 as executed over Global and Local Edge loaded tables respectively. The X axis displays the amount of SHORT_PLAY elements loaded. For 100 SHORT_PLAY elements, a majority of the total execution time is the SQL statement executing, which includes the Oracle connection. By the 13^{th} test, however, the SQL statement execution is almost less than a third of the total execution for both tables, despite the fact that there are only three other XAT operators to execute, as shown in the algebra tree of Figure 8.3. At the most for Local and Global loading, the time is only 4000 ms, with the local loading outperforming global.

Tables 10.3 and 10.4 show comparisons between SQL statement execution and total execution measured in ms for Query Q1 as executed over Global and Local Inline loaded tables. These tables show similar results as the Q1 Edge tables for SQL statement execution times. The inline tables however have overall lower times for all tests. This is due to the different shredding. Instead of creating self joins over one gigantic table, this method joins several smaller tables together. While Table 10.1 shows the 13^{th} test running

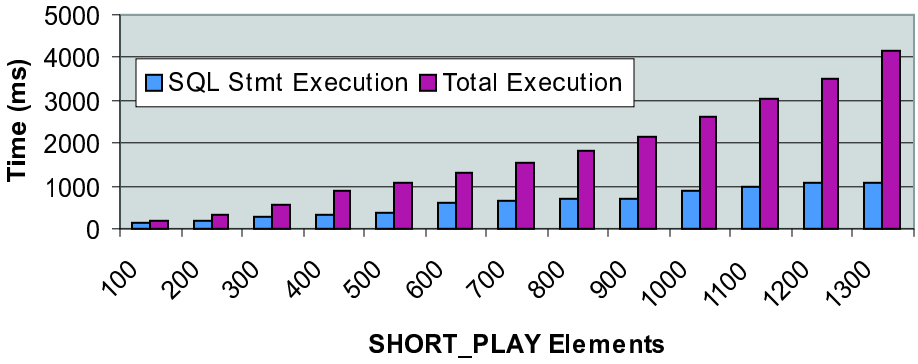

Table 10.3: Query Q1 Executed with Global Inline Loading

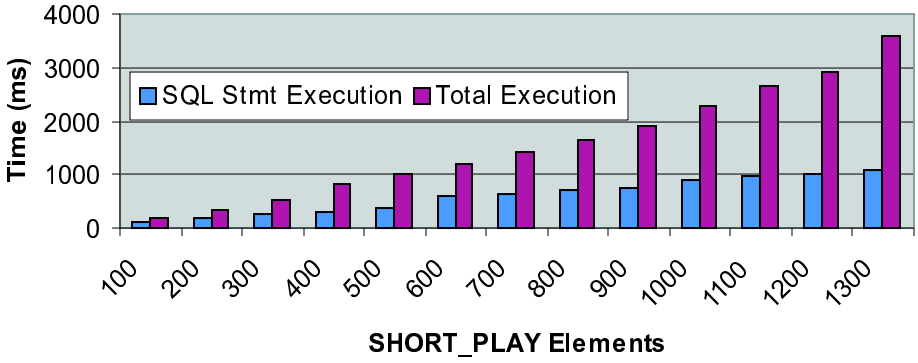

Table 10.4: Query Q1 Executed with Local Inline Loading

over 1300 ms, Table 10.3 shows a smaller time of 1050 ms, which is almost a negligible difference but still important. The local loadings are similar with times of 1600 ms for the Edge table, and 1050 ms for the Inline table, which is a greater difference and shows the potential of the inline loading better. The overall execution times are still 3-4 times greater than the SQL statement execution times, for only three XAT operators.

10.2.2 Multi Step Query Experiments

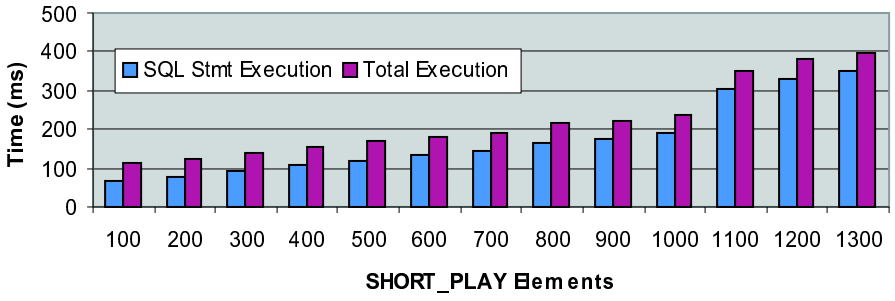

Table 10.5: Query Q2 Executed with Global Inline Loading

Tables 10.5 and 10.6 also compare SQL statement execution and overall execution as measured in ms. These tables show a different picture than the Single Step charts. This is due to the fact that the Q2 query only returns 1 tuple, regardless of the amount of data in the base tables. The SQL statement execution times increase to almost 350 ms at their highest points.

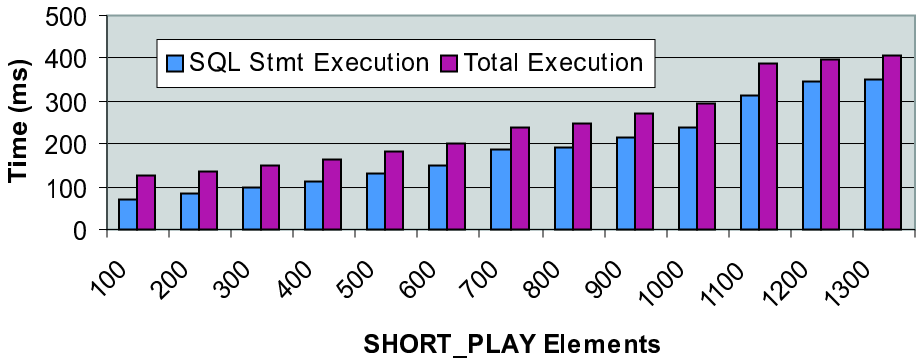

Table 10.6: Query Q2 Executed with Local Inline Loading

Table 10.7: Query Q1 with Selectivity Executed with Global Edge Loading

Table 10.8: Query Q1 with Selectivity Executed with Global Inline Loading

10.3 Experiments with Varying Selectivity

10.3.1 Single Step Query Experiments

Tables 10.7 and 10.8 have different X-axis values than the previous tables. These tables show selectivity of the data returned by the query changing over time (on the X-axis). These queries were executed over a static database with a fixed data size of 1300 SHORT_PLAYs. Tables 10.7 and 10.8 show Query Q1 as executed over Global Edge and Inline loaded tables. These queries no longer use the binary expression $=$, but rather use $<=$. For this reason, the tables appear to be skewed, in that the slopes do not follow the trend of the previous tables. These slopes are appropriate though. For the first bar, there are 1300 SONGs returned, for the second bar there are 2600 SONGs returned and so forth. By the last bar, 7800 SONGs are returned. The computation pushed into the SQL engine scales well with an increase in data searched for while the effect on the native XML execution is more noticeable due to the time required to tag all the returned SONG elements. For the selectivity cases, the inline method is still cheaper in terms of execution

time.

10.3.2 Multi Step Query Experiments

Table 10.9: Query Q2 with Selectivity Executed with Global Edge Loading

Table 10.10: Query Q2 with Selectivity Executed with Global Inline Loading

Tables 10.9 and 10.10 again show another picture, despite having the static 1300 SHORT_PLAYs in the base tables. The X axis of these charts is still selectivity of SONG

elements returned from the base data. The only difference between these Q2 queries and the previous examples are the use of the $<=$ predicate instead of the $=$ predicate. While the previous charts have experienced increasing slopes, these charts show jagged positive and negative slopes. This can be explained by the semantics of the Q2 query in terms of selectivity. The first bar of each table shows every SONG with position $<= 1$, that is only the first SONG. The second bar shows every SONG with position $<= 2$, or just the first two SONGs, while the last only returns 6 SONGs. The time difference between tagging 1 to 6 elements is very minimal which explains the jagged line. Due to the small amount of tuples returned the overall times are almost all less than 400. The inline method again shows smaller time values than the edge method.

10.4 Experimental Summary

As described in the introduction, the execution of XAT operators in main memory is expensive in the context of the Rainbow system. This is clearly shown by each of the experimental figures as the number of SONGs increases. The SQL statement is executed followed by three XAT operators. As the amount of tuples returned increases, the SQL statement grows slowly. However, the cost of executing the three XAT operators begins to grow quite quickly. For instance, in Table 10.3, the time to execute the three XAT operators is less than 70 ms with 100 SHORT_PLAYs, but upon executing over 1300 SHORT_PLAYs, the time to execute the three XAT operators is over 3000 ms. If SQL wasn't created, main memory execution would require executing over 25 such XAT operators. At the rate of Table 10.3, this time would be 25,000 ms, over 5 times the length of execution with an SQL statement.

This point is more clearly shown by Table 10.8. At the highest time line, the 3 XAT operators are tagging 7800 SONG elements. This takes nearly 50,000 ms. If there was

no SQL statement, there would be more than 20 other XAT operators to execute in main memory as well, which at that rate would total almost 300,000 ms. Asking a user to wait more than 300 seconds for a query result is unreasonable.

Chapter 11

Conclusions

11.1 Summary

XML documents are order-sensitive. Queries over these documents are like-wise order sensitive with order predicates and properly ordered results. Efficient storage methods for these documents are also desired. Many systems have chosen to utilize relational databases as this storage strategy. However, there is still a missing link between the XML queries and the relational storage, especially where order is concerned. For this reason, Rainbow was extended to handle the order-sensitive issue. This was accomplished by the following: order-specific loadings that captured implicit order-sensitive information making it explicit; metadata tables that manage the implicit to explicit mapping capture; rewrite rules for order-specific queries; SQL generation techniques for order-sensitive statement creation with the use of templates; and integration and implementation within the Rainbow system. The experimental studies show the correctness of this approach, as well as show a more efficient execution compared to native XML query execution in Rainbow.

11.2 Contributions

My contributions of this thesis are as followed:

1. The system provides guidelines for handling any general loading queries;
2. Six order-sensitive loading queries have been created to test the guidelines;
3. Rewrite rules now exist for XQuery statements that force maximal computation pushdown of order-sensitive operators;
4. A general template grammar for order-sensitive SQL query statements has been created;
5. A metadata table concept has been designed to capture implicit order-sensitive information. It was created in a manner that it is extendable and general to enable the addition of ”new” order encodings or ”new” loading strategies in the future;
6. SQL generation in Rainbow now includes order-sensitive SQL statement creation through the use of templates;
7. This framework was implemented and integrated within the Rainbow system;
8. An experimental study was conducted to evaluate the proposed order-based XML query processing system.

Most importantly, the system was designed in a general manner to handle any new mappings or orderings that may arise. The scenario for each is quite simple. If a new order method is created, any current mapping would require a new order pre-processing step as part of the full loading query. This step is shown in each Step Zero of Appendix A. This simply means that a new Step Zero XQuery function would need to be written. No Java code would need to be added to the system. If a new mapping method was to

be utilized, then a new loading XQuery statement would need to be created – one for each current order types. Again, no code would need to be added to the current system to handle this. Information about this mapping would also need to be created within its own metadata table.

11.3 Future Work

Currently, Rainbow only handles numeric order predicates. In the future, it will be necessary to implement LAST(), FIRST() and other related order functions, as well as their translations to SQL.

Also, the metadata table is created manually for each document and loading in our system. Part of the future work would be to automate this process. This process could be done with a mixture of XQuery and Java in parallel to the loading process.

Finally, the work described in Section 9.1 must be implemented such that SQL generation is purely incremental and aware of the hierarchy of the algebra tree.

Bibliography

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

[2] 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.

[3] D. Chamberlin and J. Robie and D. Florescu. Quilt: An XML Query Language for Heterogeneous Data Sources. In *ACM SIGMOD Associated Workshop on the Web and Databases (WebDB 2000), Dallas, Texas*, pages 53–62, May 2000

[4] S. Christ, X. Zhang and E. A. Rundensteiner. X-Cube: Flexible XML Mapping by XQuery. Technical report, Worcester Polytechnic Institute, 2001.

[5] K. Deschler. Xml navigation indexing. Master's thesis, Worcester Polytechnic Institute, 2001.

[6] 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.

[7] D. Florescu, D. Kossman. Storing and Querying XML Data using an RDBMS. *IEEE Data Engineering Bulletin*, 22(3), 1999.

[8] L. Galanis, E. Viglas, D. J. DeWitt, J. F. Naughton, and D. Maier. Following the paths of xml data: An algebraic framework for xml query evaluation. http://www.cs.wisc.edu/niagara/papers/algebra.pdf, 2001.

[9] H. V.Jagadish, S. Al-Khalifa, A. Chapman, L. Lakshmanan, A. Nierman, S. Paparizos, J. M. Patel, D. Srivastava, N. Wiwatwattana, Y. Wu, and C. Yu. Timber: A native xml database, 2002. *VLDB Journal*, 11(4), 2002.

[10] I. Manolescu, D. Florescu, D. Kossmann, F. Xhumari, and D. Olteanu. Agora: Living with xml and relational. In A. E. Abbadi, M. L. Brodie, S. Chakravarthy, U. Dayal, N. Kamel, G. Schlageter, and K.-Y. Whang, editors, *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10-14, 2000, Cairo, Egypt*, pages 623–626. Morgan Kaufmann, 2000.

[11] R. Ramakrishnan. *Database Management Systems*. WCB/McGraw-Hill, 2002

[12] R. Ramakrishnan, D. Donjerkovic, A. Ranganathan, K. S. Beyer, and M. Krishnaprasad. Srql: Sorted relational query language. In M. Rafanelli and M. Jarke, editors, *10th International Conference on Scientific and Statistical Database Management, Proceedings, Capri, Italy, July 1-3, 1998*, pages 84–95. IEEE Computer Society, 1998.

[13] A. Sahuguet. Kweelt: More than just "yet another framework to query xml!". In *Demo Session Proceedings of SIGMOD'01*, page 602, 2001.

[14] Shore Team. Shore: Combining the best features of oodbms and file systems. In M. J. Carey and D. A. Schneider, editors, *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, May 22-25, 1995*, page 486. ACM Press, 1995.

[15] I. Tatarinov, S. D. Viglas, K. Beyer, J. Shanmugasundaram, E. J. Shekita, and C. Zhang Storing and Querying Ordered XML Using a Relational Database System. *ACM SIGMOD*, pages 204-215, 2002.

[16] W3C. Document Object Model (DOM). http://www.w3.org/TR/REC-DOM-Level-1/, 1998.

[17] W3C. XQuery: A Query Language for XML. http://www.w3.org/TR/xquery/, February 2001.

[18] W3C. XQuery 1.0 and XPath 2.0 Data Model. http://www.w3.org/TR/query-datamodel, December 2001.

[19] C. Zainolo, S. Ceri, C. Faloustos, R. Snodgrass, V. S. Subrahmanian and R. Zicari. *Advanced Database Systems*. Morgan Kaufman, 1997.

[20] X. Zhang. XML Query Decorrelation. Technical report, Worcester Polytechnic Institute, 2002. to appear.

[21] X. Zhang, G. Mitchell, W.-C. Lee, and E. A. Rundensteiner. Clock: Synchronizing Internal Relational Storage with External XML Documents. In *RIDE-DM*, pages 111–118, April 2001.

[22] X. Zhang, M. Mulchandani, S. Christ, B. Murphy, and B. Pielech. Rainbow: Mapping-Driven XQuery Processing System. In *Demo Session Proceedings of SIGMOD'02*, 2002.

[23] X. Zhang, B. Pielech, and E. A. Rundensteiner. Honey, I Shrunk the XQuery! — An XML Algebra Optimization Approach. In *WIDM*, pages 15–22, Nov. 2002.

[24] X. Zhang, B. Pielech, L. Ding, B. Murphy, L. Wang, K. Dimitrova, M. El Sayed and E. A. Rundensteiner. RainbowII: Multi-XQuery Optimization Using Materialized XML Views. In *Demo Session Proceedings of SIGMOD'03*, 2003.

[25] X. Zhang, B. Pielech, and E. A. Rundensteiner. XAT Optimization. Technical Report WPI-CS-TR-02-25, Worcester Polytechnic Institute, 2002.

[26] 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.

Appendix A

Loading Queries

A.1 Schema Queries

A.1.1 Edge Loading Schema Query

```
<xsd:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
    <xsd:element name="DB">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element name="EDGES">
                    <xsd:complexType>
                        <xsd:sequence>
                            <xsd:element name="EDGES.ROW" minOccurs="1" maxOccurs="unbounded">
                                <xsd:complexType>
                                    <xsd:sequence>
                                        <xsd:element name="SOURCE" type="xsd:double"/>,
                                        <xsd:element name="POSITION" type="xsd:double"/>,
                                        <xsd:element name="NAME" type="xsd:string"/>,
                                        <xsd:element name="TARGET" type="xsd:string"/>
                                    </xsd:sequence>
                                </xsd:complexType>
                            </xsd:element>
                        </xsd:sequence>
                    </xsd:complexType>
                </xsd:element>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>
</xsd:schema>
```

A.1.2 Inline Loading Schema Query

```
<xsd:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
    <xsd:element name="DB">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element name="EDGES">
                    <xsd:complexType>
                        <xsd:sequence>
                            <xsd:element name="EDGES.ROW" minOccurs="1" maxOccurs="unbounded">
                                <xsd:complexType>
                                    <xsd:sequence>
                                        <xsd:element name="SOURCE" type="xsd:double"/>,
                                        <xsd:element name="POSITION" type="xsd:double"/>,
                                        <xsd:element name="NAME" type="xsd:string"/>,
                                        <xsd:element name="TARGET" type="xsd:string"/>
                                    </xsd:sequence>
                                </xsd:complexType>
                            </xsd:element>
                        </xsd:sequence>
                    </xsd:complexType>
                </xsd:element>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>
</xsd:schema>
```

A.2 Step 0

A.2.1 Local Edge Loading Step 0

```
FUNCTION Q1($root, $pid, $porder1){
    LET $maintag := gettag($root),
        $iid := getiid(),
        $porder2 := 0,
        $pos := 0
    RETURN
    {
        <$maintag type="ELEM" iid=$iid pid=$pid porder=$porder1>
            FOR $attribute IN $root/@*
            LET $atttag := gettag($attribute),
                $attiid := getiid(),
                $pcdataiid := getiid()
            RETURN
            {
                <$atttag type="ATT" iid=$attiid pid=$iid porder="0.0">
                    <CDATA type="CDATA" iid=$pcdataiid pid=$attiid porder="0.0">
                        $attribute
                    </CDATA>
                </$atttag>
            },
            FOR $elem IN $root/*
            LET $pos := $pos + 1,
                $pcd := $root/text()[position()=$pos]
            RETURN
            {
                IF (TRIM($pcd)="")
                THEN
                {
                    ""
                }
                ELSE
                {
                    LET $pcdataiid := getiid(),
                        $porder2 := $porder2 + 1
                    RETURN
                    {
                        <PCDATA type="PCDATA" iid=$pcdataiid pid=$iid porder=$porder2>
                            TRIM($pcd)
                        </PCDATA>
                    }
                },
                LET $porder2 := $porder2 + 1
                RETURN
                {
                    Q1($elem, $iid, $porder2)
                }
            },
            LET $pos := $pos + 1,
                $pcd := $root/text()[position()=$pos]
            RETURN
            {
                IF (TRIM($pcd)="")
                THEN
                {
                    ""
                }
                ELSE
                {
                    LET $pcdataiid := getiid(),
                        $porder2 := $porder2 + 1
                    RETURN
```

```
                    {
                        <PCDATA type="PCDATA" iid=$pcdataiid pid=$iid porder=$porder2>
                            TRIM($pcd)
                        </PCDATA>
                    }
                }
            }
        </$maintag>
    }
}

Q1(document("temp/source.xml"), 0, 1)
```

A.2.2 Global Edge Loading Step 0

```
FUNCTION Q1($root, $pid){
    LET $maintag := gettag($root),
        $iid := getiid(),
        $pos := 0
    RETURN
    {
        <$maintag type="ELEM" iid=$iid pid=$pid porder=$iid>
            FOR $attribute IN $root/@*
            LET $atttag := gettag($attribute),
                $attiid := getiid(),
                $pcdataiid := getiid()
            RETURN
            {
                <$atttag type="ATT" iid=$attiid pid=$iid porder="0.0">
                    <CDATA type="CDATA" iid=$pcdataiid pid=$attiid porder="0.0">
                        $attribute
                    </CDATA>
                </$atttag>
            },
            FOR $elem IN $root/*
            LET $pos := $pos + 1,
                $pcd := $root/text()[position()=$pos]
            RETURN
            {
                IF (TRIM($pcd)="")
                THEN
                {
                    ""
                }
                ELSE
                {
                    LET $pcdataiid := getiid()
                    RETURN
                    {
                        <PCDATA type="PCDATA" iid=$pcdataiid pid=$iid porder=$pcdataiid>
                            TRIM($pcd)
                        </PCDATA>
                    }
                },
                RETURN
                {
                    Q1($elem, $iid)
                }
            },
            LET $pos := $pos + 1,
                $pcd := $root/text()[position()=$pos]
            RETURN
            {
                IF (TRIM($pcd)="")
                THEN
                {
                    ""
                }
                ELSE
                {
                    LET $pcdataiid := getiid()
                    RETURN
                    {
                        <PCDATA type="PCDATA" iid=$pcdataiid pid=$iid porder=$pcdataiid>
                            TRIM($pcd)
                        </PCDATA>
                    }
                }
```

```
            }
        </$maintag>
    }
}

Q1(document("temp/source.xml"), 0)
```

A.2.3 Lexicographical Edge Loading Step 0

```
FUNCTION Q1($root, $pid, $paorder){
    LET $maintag := gettag($root),
        $iid := getiid(),
        $pos := 0,
        $porder1 := "aa",
        $fake := LEXICOGRAPHICALORDER("reset"),
        $corder := "aa"
    RETURN
    {
        <$maintag type="ELEM" iid=$iid pid=$pid porder=$paorder>
            FOR $attribute IN $root/@*
            LET $atttag := gettag($attribute),
                $attiid := getiid(),
                $pcdataiid := getiid()
            RETURN
            {
                <$atttag type="ATT" iid=$attiid pid=$iid porder="0.0">
                    <CDATA type="CDATA" iid=$pcdataiid pid=$attiid porder="0.0">
                        $attribute
                    </CDATA>
                </$atttag>
            },
            FOR $elem IN $root/*
            LET $pos := $pos + 1,
                $pcd := $root/text()[position()=$pos],
                $porder1 := LEXICOGRAPHICALORDER(count($root/../*)),
                $corder := CONCAT($paorder, ".", $porder1)
            RETURN
            {
                IF (TRIM($pcd)="")
                THEN
                {
                    ""
                }
                ELSE
                {
                    LET $pcdataiid := getiid()
                    RETURN
                    {
                        <PCDATA type="PCDATA" iid=$pcdataiid pid=$iid porder=$corder>
                            TRIM($pcd)
                        </PCDATA>
                    }
                },
                RETURN
                {
                    Q1($elem, $iid,  $corder)
                }
            },
            LET $pos := $pos + 1,
                $pcd := $root/text()[position()=$pos],
                $porder1 := LEXICOGRAPHICALORDER(count($root/../*)),
                $corder := CONCAT($paorder, ".", $porder1)
            RETURN
            {
                IF (TRIM($pcd)="")
                THEN
                {
                    ""
                }
                ELSE
                {
                    LET $pcdataiid := getiid()
```

```
                    RETURN
                    {
                        <PCDATA type="PCDATA" iid=$pcdataiid pid=$iid porder=$corder>
                            TRIM($pcd)
                        </PCDATA>
                    }
                }
            }
        </$maintag>
    }
}

Q1(document("temp/source.xml"), 0, "aa")
```

A.2.4 Local Inline Loading Step 0

```
FUNCTION Q1($root, $pid, $porder1){
        LET $maintag := gettag($root),
            $iid := getiid(),
            $porder2 := 0,
            $pos := 0
        RETURN
        {
                <$maintag type="ELEM" iid=$iid pid=$pid porder=$porder1>
                        FOR $attribute IN $root/@*
                        LET $atttag := gettag($attribute),
                            $attiid := getiid(),
                            $pcdataiid := getiid()
                        RETURN
                        {
                                <$atttag type="ATT" iid=$attiid pid=$iid porder="0.0">
                                        <CDATA type="CDATA" iid=$pcdataiid pid=$attiid porder="0.0">
                                                $attribute
                                        </CDATA>
                                </$atttag>
                        },
                        FOR $elem IN $root/*
                        LET $pos := $pos + 1,
                            $pcd := $root/text()[position()=$pos]
                        RETURN
                        {
                                IF (TRIM($pcd)="")
                                THEN
                                {
                                        ""
                                }
                                ELSE
                                {
                                        LET $pcdataiid := getiid(),
                                            $porder2 := $porder2 + 1
                                        RETURN
                                        {
                                           <PCDATA type="PCDATA" iid=$pcdataiid pid=$iid porder=$porder2>
                                                 TRIM($pcd)
                                           </PCDATA>
                                        }
                                },
                                LET $porder2 := $porder2 + 1
                                RETURN
                                {
                                        Q1($elem, $iid, $porder2)
                                }
                        },
                        LET $pos := $pos + 1,
                            $pcd := $root/text()[position()=$pos]
                        RETURN
                        {
                                IF (TRIM($pcd)="")
                                THEN
                                {
                                        ""
                                }
                                ELSE
                                {
                                        LET $pcdataiid := getiid(),
                                            $porder2 := $porder2 + 1
                                        RETURN
                                        {
                                           <PCDATA type="PCDATA" iid=$pcdataiid pid=$iid porder=$porder2>
```

```
                                        TRIM($pcd)
                                    </PCDATA>
                                }
                            }
                        }
                    </$maintag>
            }
}

Q1(document("temp/source.xml"), 0, 1)
```

A.2.5 Global Inline Loading Step 0

```
FUNCTION Q1($root, $pid){
    LET $maintag := gettag($root),
        $iid := getiid(),
        $pos := 0
    RETURN
    {
        <$maintag type="ELEM" iid=$iid pid=$pid porder=$iid>
            FOR $attribute IN $root/@*
            LET $atttag := gettag($attribute),
                $attiid := getiid(),
                $pcdataiid := getiid()
            RETURN
            {
                <$atttag type="ATT" iid=$attiid pid=$iid porder="0.0">
                    <CDATA type="CDATA" iid=$pcdataiid pid=$attiid porder="0.0">
                        $attribute
                    </CDATA>
                </$atttag>
            },
            FOR $elem IN $root/*
            LET $pos := $pos + 1,
                $pcd := $root/text()[position()=$pos]
            RETURN
            {
                IF (TRIM($pcd)="")
                THEN
                {
                    ""
                }
                ELSE
                {
                    LET $pcdataiid := getiid()
                    RETURN
                    {
                        <PCDATA type="PCDATA" iid=$pcdataiid pid=$iid porder=$pcdataiid>
                            TRIM($pcd)
                        </PCDATA>
                    }
                },
                LET $porder1 := getiid()
                RETURN
                {
                    Q1($elem, $iid)
                }
            },
            LET $pos := $pos + 1,
                $pcd := $root/text()[position()=$pos]
            RETURN
            {
                IF (TRIM($pcd)="")
                THEN
                {
                    ""
                }
                ELSE
                {
                    LET $pcdataiid := getiid()
                    RETURN
                    {
                        <PCDATA type="PCDATA" iid=$pcdataiid pid=$iid porder=$pcdataiid>
                            TRIM($pcd)
                        </PCDATA>
                    }
```

```
                }
            }
        </$maintag>
    }
}

Q1(document("temp/source.xml"), 0)
```

A.2.6 Lexicographical Inline Loading Step 0

```
FUNCTION Q0($root, $pid, $paorder){
    LET $maintag := gettag($root),
        $iid := getiid(),
        $pos := 0,
        $porder1 := "aa",
        $fake := LEXICOGRAPHICALORDER("reset"),
        $corder := "aa"
    RETURN
    {
        <$maintag type="ELEM" iid=$iid pid=$pid porder=$paorder>
            FOR $attribute IN $root/@*
            LET $atttag := gettag($attribute),
                $attiid := getiid(),
                $pcdataiid := getiid()
            RETURN
            {
                <$atttag type="ATT" iid=$attiid pid=$iid porder="0.0">
                    <CDATA type="CDATA" iid=$pcdataiid pid=$attiid porder="0.0">
                        $attribute
                    </CDATA>
                </$atttag>
            },
            FOR $elem IN $root/*
            LET $pos := $pos + 1,
                $pcd := $root/text()[position()=$pos],
                $porder1 := LEXICOGRAPHICALORDER(count($root/../*)),
                $corder := CONCAT($paorder, ".", $porder1)
            RETURN
            {
                IF (TRIM($pcd)="")
                THEN
                {
                    ""
                }
                ELSE
                {
                    LET $pcdataiid := getiid()
                    RETURN
                    {
                        <PCDATA type="PCDATA" iid=$pcdataiid pid=$iid porder=$corder>
                            TRIM($pcd)
                        </PCDATA>
                    }
                },
                RETURN
                {
                    Q0($elem, $iid, $corder)
                }
            },
            LET $pos := $pos + 1,
                $pcd := $root/text()[position()=$pos],
                $porder1 := LEXICOGRAPHICALORDER(count($root/../*)),
                $corder := CONCAT($paorder, ".", $porder1)
            RETURN
            {
                IF (TRIM($pcd)="")
                THEN
                {
                    ""
                }
                ELSE
                {
                    LET $pcdataiid := getiid()
```

```
                    RETURN
                    {
                        <PCDATA type="PCDATA" iid=$pcdataiid pid=$iid porder=$corder>
                            TRIM($pcd)
                        </PCDATA>
                    }
                }
            }
        </$maintag>
    }
}

Q0(document("temp/source.xml"), 0, "aa")
```

A.3 Step 1

This step is the same for each loading regardless of ordering

A.3.1 Edge Loading Step 1

```
FUNCTION Q1($root)
{
    LET $tag := gettag($root)
    RETURN
    {
        IF ($root/PCDATA)
        THEN
        {
            <EDGES.ROW>
                <SOURCE>$root/@pid</SOURCE>,
                <POSITION>$root/@eorder</POSITION>,
                <NAME>$tag</NAME>,
                <TARGET>$root/PCDATA/text()</TARGET>
            </EDGES.ROW>
        }
        ELSE
        {
            IF ($root/CDATA)
            THEN
            {
                <EDGES.ROW>
                    <SOURCE>$root/@pid</SOURCE>,
                    <POSITION>$root/@eorder</POSITION>,
                    <NAME>$tag</NAME>,
                    <TARGET>$root/CDATA/text()</TARGET>
                </EDGES.ROW>
            }
            ELSE
            {
                <EDGES.ROW>
                    <SOURCE>$root/@pid</SOURCE>,
                    <POSITION>$root/@eorder</POSITION>,
                    <NAME>$tag</NAME>,
                    <TARGET>$root/@iid</TARGET>
                </EDGES.ROW>,
                FOR $child IN $root/*[./@type="ELEM" OR ./@type="ATT"]
                RETURN
                {
                    Q1($child)
                }
            }
        }
    }
}

<DB xmlns:xsi="http://www.w3.org/2000/10/XMLSchema-instance" xsi:noNamespaceSchemaLocation="schema.xsd">
    <EDGES>
        Q1({document("temp/result0.xml")})
    </EDGES>
</DB>
```

A.3.2 Inline Loading Step 1

```
FUNCTION Inlinable($root)
{
    LET $elements := document("temp/source.xsd")//xs:element [./@name=$root AND NOT ./@maxOccurs!="1"]
    RETURN
    {
        IF ( COUNT($elements)=0 )
        THEN
        {
            FALSE
        }
        ELSE
        {
            LET $choices := document("temp/source.xsd")//xs:choice [./xs:element/@name=$root]
                            UNION document("temp/source.xsd")/xs:element [./@name=$root]
            RETURN
            {
                IF ( COUNT($choices)=0 )
                THEN
                {
                    TRUE
                }
                ELSE
                {
                    FALSE
                }
            }
        }
    }
}

FUNCTION PCDATAInline($root)
{
    LET $choices := document("temp/source.xsd")//xs:element[./@name=$root
                    AND NOT ./@ref]/xs:complexType/xs:choice
    RETURN
    {
        IF ( COUNT($choices)=0 )
        THEN
        {
            TRUE
        }
        ELSE
        {
            FALSE
        }
    }
}

FUNCTION Q1($root, $parentinlinable, $PCDATAInline)
{
    LET $tag := gettag($root)
    RETURN
    {
        IF ($root/@type="ELEM")
        THEN
        {
            <$tag INLINABLE=Inlinable($tag) $root/@*>
                FOR $child IN $root/*
                      RETURN
                      {
                          Q1($child, Inlinable($tag), PCDATAInline($tag))
                      }
                  </$tag>
```

```
        }
        ELSE
        {
            IF ($root/@type="ATT")
            THEN
            {
                LET $tag := CONCAT($tag, "_ATT")
                RETURN
                {
                    <$tag INLINABLE="TRUE" $root/@*>
                        $root/*[1]/text()
                          </$tag>
                      }
                  }
                  ELSE
                  {
                      IF ($parentinlinable="TRUE" and $PCDATAInline="TRUE")
                      THEN
                      {
                          $root/text()
                      }
                      ELSE
                      {
                          <$tag INLINABLE=$PCDATAInline $root/@*>
                        $root/text()
                          </$tag>

                      }
                  }
        }
    }
}

<DB>
    Q1({document("temp/result0.xml")}, "FALSE", "TRUE")
</DB>
```

A.4 Step 2

This step is the same for each loading regardless of ordering

A.4.1 Edge Loading Step 2

There is no second step for the Edge mapping.

A.4.2 Inline Loading Step 2

```
FUNCTION Q2($root, $path)
{
    RETURN
    {
        LET $tag := CONCAT($path, "_", gettag($root)),
            $elements := $root/*[(./@type="ELEM" OR ./@type="PCDATA") AND ./@INLINABLE="TRUE"]
        RETURN
        {
            IF (COUNT($elements) = 0)
            THEN
            {
                IF (TRIM($root/text())="")
                THEN
                {
                    LET $iidtag := CONCAT($tag, "_IID")
                    RETURN
                    {
                        <$iidtag>$root/@iid</$iidtag>
                    }
                }
                ELSE
                {
                    <$tag>$root/text()</$tag>
                }
            }
            ELSE
            {
                FOR $elem IN $elements
                RETURN
                {
                    Q2($elem, $tag)
                }
            }
        },
        FOR $attchild IN $root/*[./@type="ATT"]
        LET $atttag := CONCAT($path, "_", gettag($root), "_", gettag($attchild))
        RETURN
        {
            <$atttag>$attchild/text()</$atttag>
        }
    }
}

<DB xmlns:xsi="http://www.w3.org/2000/10/XMLSchema-instance" xsi:noNamespaceSchemaLocation="schema.xsd">
    FOR $tag IN DISTINCT (FOR $xxx IN document("temp/result1.xml")//*[./@INLINABLE="FALSE"]
                  RETURN{<T>gettag($xxx)</T>})
    LET $tables := document("temp/result1.xml")//*[name(.)=TRIM($tag/text()) AND ./@INLINABLE="FALSE"],
        $tag := $tag/text()
    RETURN
    {
        <$tag>
            FOR $table IN $tables
            LET $tabletag := CONCAT($tag, ".ROW")
            RETURN
```

```
            {
                <$tabletag>
                    <IID>$table/@iid</IID>,
                    <PID>$table/@pid</PID>,
                    <POSITION>$table/@porder</POSITION>,
                    IF ($tag="PCDATA")
                    THEN
                    {
                        <VALUE>$table/text()</VALUE>
                    }
                    ELSE
                    {
                        FOR $attchild IN $table/*[./@type="ATT"]
                        LET $atttag := CONCAT($tag, "_", gettag($attchild))
                        RETURN
                        {
                            <$atttag>$attchild/text()</$atttag>
                        },
                        FOR $elem IN $table/*[(./@type="ELEM" OR ./@type="PCDATA") AND ./@INLINABLE="TRUE"]
                        RETURN
                        {
                            Q2($elem, $tag)
                        }
                    }
                </$tabletag>
            }
        </$tag>
    }
</DB>
```

Appendix B

View Queries

B.0.3 Edge Loading

```
<RECORDLIST>
FOR $playMap in document("temp/dxvfe.xml")/EDGES/EDGES.ROW[
       NAME/text()="short_play"]
RETURN
<SHORT_PLAY pos=$playMap/POSITION/text()>
 FOR $songMap in document("temp/dxvfe.xml")/EDGES/EDGES.ROW[
       SOURCE = $playMap/TARGET and NAME/text()="song"]
 RETURN
       <SONG pos=$songMap/POSITION/text()>$songMap/TARGET/text()</SONG>
 SORTBY (./@pos ASCENDING)
</SHORT_PLAY>
SORTBY(./@pos)
</RECORDLIST>
```

B.0.4 Inline Loading

```
<RECORDLIST>
FOR $playMap in document("temp/dxvli.xml")/SHORT_PLAY/SHORT_PLAY.ROW
RETURN
<SHORT_PLAY pos=$playMap/POSITION/text()>
 FOR $songMap in document("temp/dxvli.xml")/SONG/SONG.ROW[
        PID/text() = $playMap/IID/text()]
 RETURN
        <SONG pos=$songMap/POSITION/text()>
              $songMap/SONG_PCDATA/text()
        </SONG>
 SORTBY (./@pos ASCENDING)
</SHORT_PLAY>
SORTBY(./@pos)
</RECORDLIST>
```

Appendix C

Recordlist XML Schema

```
<?xml version="1.0"?>
  <xs:schema xmlns:xsd="http://www.w3.org/2000/10/XMLSchema"
    elementFormDefault="qualified">
    <xs:element name="RECORDLIST">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="SHORT_PLAY" maxOccurs="unbounded">
                    <xs:complexType>
                        <xs:sequence>
                            <xs:element name="BAND" type="xs:string"/>
                            <xs:element name="LABEL" type="xs:string"/>
                            <xs:element name="SONG" maxOccurs="6"
                                type="xs:string"/>
                        </xs:sequence>
                    </xs:complexType>
                </xs:element>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
  </xs:schema>
```

Appendix D

Default XML View

Recordlist DXV by Inline Global Strategy

```
<?xml version="1.0"?>
<DB xmlns:xsi="http://www.w3.org/2000/10/XMLSchema-instance" xsi:noNamespaceSchemaLocation="schema.xsd">
  <band>
    <band.ROW>
      <IID>
        33.0
      </IID>
      <PID>
        31.0
      </PID>
      <POSITION>
        33.0
      </POSITION>
      <band_PCDATA>
        Project X
      </band_PCDATA>
    </band.ROW>
    <band.ROW>
      <IID>
        19.0
      </IID>
      <PID>
        17.0
      </PID>
      <POSITION>
        19.0
      </POSITION>
      <band_PCDATA>
        Misfits
      </band_PCDATA>
    </band.ROW>
    <band.ROW>
      <IID>
        5.0
      </IID>
      <PID>
        3.0
      </PID>
      <POSITION>
        5.0
      </POSITION>
      <band_PCDATA>
        Misfits
      </band_PCDATA>
```

```
  </band.ROW>
</band>
<label>
  <label.ROW>
    <IID>
      36.0
    </IID>
    <PID>
      31.0
    </PID>
    <POSITION>
      36.0
    </POSITION>
    <label_PCDATA>
      Schism
    </label_PCDATA>
  </label.ROW>
  <label.ROW>
    <IID>
      22.0
    </IID>
    <PID>
      17.0
    </PID>
    <POSITION>
      22.0
    </POSITION>
    <label_PCDATA>
      Plan 9
    </label_PCDATA>
  </label.ROW>
  <label.ROW>
    <IID>
      8.0
    </IID>
    <PID>
      3.0
    </PID>
    <POSITION>
      8.0
    </POSITION>
    <label_PCDATA>
      blank
    </label_PCDATA>
  </label.ROW>
</label>
<song>
  <song.ROW>
    <IID>
      39.0
    </IID>
    <PID>
      31.0
    </PID>
    <POSITION>
      39.0
    </POSITION>
    <song_PCDATA>
      SXE Revenge
    </song_PCDATA>
  </song.ROW>
  <song.ROW>
    <IID>
      42.0
    </IID>
```

```
  <PID>
    31.0
  </PID>
  <POSITION>
    42.0
  </POSITION>
  <song_PCDATA>
    Shutdown
  </song_PCDATA>
</song.ROW>
<song.ROW>
  <IID>
    25.0
  </IID>
  <PID>
    17.0
  </PID>
  <POSITION>
    25.0
  </POSITION>
  <song_PCDATA>
    Bullet
  </song_PCDATA>
</song.ROW>
<song.ROW>
  <IID>
    28.0
  </IID>
  <PID>
    17.0
  </PID>
  <POSITION>
    28.0
  </POSITION>
  <song_PCDATA>
    We Are 138
  </song_PCDATA>
</song.ROW>
<song.ROW>
  <IID>
    11.0
  </IID>
  <PID>
    3.0
  </PID>
  <POSITION>
    11.0
  </POSITION>
  <song_PCDATA>
    Cough Cool
  </song_PCDATA>
</song.ROW>
<song.ROW>
  <IID>
    14.0
  </IID>
  <PID>
    3.0
  </PID>
  <POSITION>
    14.0
  </POSITION>
  <song_PCDATA>
    She
  </song_PCDATA>
```

```
    </song.ROW>
  </song>
  <short_play>
    <short_play.ROW>
      <IID>
        3.0
      </IID>
      <PID>
        1.0
      </PID>
      <POSITION>
        3.0
      </POSITION>
    </short_play.ROW>
    <short_play.ROW>
      <IID>
        17.0
      </IID>
      <PID>
        1.0
      </PID>
      <POSITION>
        17.0
      </POSITION>
    </short_play.ROW>
    <short_play.ROW>
      <IID>
        31.0
      </IID>
      <PID>
        1.0
      </PID>
      <POSITION>
        31.0
      </POSITION>
    </short_play.ROW>
  </short_play>
  <recordlist>
    <recordlist.ROW>
      <IID>
        1.0
      </IID>
      <PID>
        0.0
      </PID>
      <POSITION>
        1.0
      </POSITION>
    </recordlist.ROW>
  </recordlist>
</DB>
<!-- end of document -->
```