UNIVERSITY OF CALIFORNIA,

IRVINE

Comparing Native XML Data Management Approaches in Relational and Native XML Databases

THESIS

submitted in partial satisfaction of the requirements

for the degree of

MASTER OF SCIENCE

in Computer Science

by

Ling Ling

Thesis Committee:

Professor Michael J. Carey, Chair

Professor Chen Li

Professor Sharad Mehrotra

2010

Table of Contents

Page

LIST OF FIGURES

Page

LIST OF TABLES

Page

ACKNOWLEDGEMENTS

First, I would like to thank my advisor, Michael Carey. He shows great enthusiasm to his research in the database area. His attitude actually encouraged me to find my own enthusiasm, by which I made a big decision to start exploring my interests. Also, I wish to thank him for his great patience and nice attitude, which guided me through the first two years in the United States and also in Graduate School.

I would also like to thank my other two advisory committee members, Professor Chen Li and Professor Sharad Mehrotra. I would like to thank them for their time and their feedback on my thesis. Besides my committee, I would like to thank my friends David Lam and Jun Wang. They also give me a lot of help and comments on my thesis.

Related to the EXRT project, I wish to thank my labmate Lin Shao. We discussed issues and tradeoffs together, coded together, debugged together, and stayed in the lab all night together. Also, I would like to thank Matthias Nicola, a Senior Engineer at IBM, for his time and our work together on the EXRT project. Through numerous emails and phone calls, I learned a lot from him, including both technical knowledge and work attitude. I would also like to extend special thanks to our engineering contacts from each of the three database vendors whose products we benchmarked. Their valuable advice and explanations helped us to finish our project.

Last but not least, I would like to acknowledge the support of my family and all of my friends, wherever they are, both here and far away. Thank you for all your support!

ABSTRACT OF THE THESIS

Comparing Native XML Data Management Approaches in Relational and Native XML Databases

by

Ling Ling

Master of Science in Computer Science

University of California, Irvine, 2010

Professor Michael J. Carey, Chair

As XML has become widely used, XML storage and query support has been added to a number of commercial database systems. This has taken two forms, relational shredded storage – when XML data is mapped to rows of tables - and native XML storage. Relational shredded storage has been supported for a number of years, while native XML storage is just starting to be prevalent in the commercial database area. For many use cases, native XML storage is now recommended by database vendors. In this thesis, a new benchmark called EXRT (Experimental XML Readiness Test) is used to examine the performance of XML data management features for native XML storage in three commercial database systems, including two relational database systems and one XML database system. Also, details of the implementation of EXRT, which is a simple micro-benchmark that methodically evaluates the impact of XML

query characteristics on the tradeoffs between various XML storage and query processing schemes, are described. Preliminary results from running benchmarks using EXRT are presented and the lessons learned so far are summarized.

1 Introduction

Extensible Markup Language (XML) [1] is a text-based human readable markup language designed to describe data content rather than its presentation details. It does so by containing markup tags around the content; the tags are self-describing and easy to understand. Figure 1-1 shows an example XML document to illustrate this.

```
<?xml version="1.0" encoding="US-ASCII"?>
<Customer id="1011“ >
        <Mnemonic>VilloldoMarjo</Mnemonic>
        <ShortNames>
                <ShortName>Marjo Villoldo</ShortName>
        </ShortNames>
        <Name>
                <Title>Mrs</Title>
                <FirstName>Marjo</FirstName>
                <MiddleName>Akos</MiddleName>
                <LastName>Villoldo</LastName>
        </Name>
        <DateOfBirth>1956-11-27</DateOfBirth>
        <Gender>Male</Gender>
        <Nationality>Portuguese</Nationality>
        <CountryOfResidence>Portugal</CountryOfResidence>
        <Languages>
                <Language>Portuguese</Language>
        </Languages>
</Customer >
```

Figure 1-1 Sample XML Document

In this way, XML makes information exchange easier among different devices, applications, platforms and systems from different vendors, as systems are able to determine what the data is representing from its markup. Moreover, XML is relatively flexible, and it is relatively trivial for users to add new tags to describe any new content

that might need to be added into an XML document. As XML has become more and more widely adopted over the years, relational database management systems (RDBMs) have at the same time added and improved their support of XML.

At the outset, XML documents in relational systems would be simply mapped into existing relational database structures like “Large Objects” (LOBs), with XML being stored intact as plain unparsed text. Another approach used by RDBMs has been to create appropriate relational schemas and “shred” XML documents into many tables. The limitations of these two approaches are now well known. While inserting and extracting full XML documents is relatively fast in the case of LOB storage, this approach can be relatively slow during query processing and fragment extraction due to the need for XML parsing at query execution time [2, 4]. While the “shredded” approach can provide reasonable performance given a good mapping, mapping an XML schema to an equivalent relational schema is usually a complicated job. Also, the nested and repeating elements in XML documents can quite easily result in an unmanageable number of tables. Furthermore, it is usually very difficult after insertion to change the relational schema due to XML schema changes. Thus, the flexibility of XML is essentially lost with this approach [2, 4].

Consequently, RDBMs vendors began to realize the need to support XML documents natively, where the intrinsic hierarchical structure of an XML document would be maintained. Soon, several RDBMs vendors announced their own native XML storage features, such as Oracle XML DB [4] and IBM pureXML [2, 9]. Microsoft SQL

Server also stores XML document natively as binary XML [5]. Along with these larger (relational) database vendors, there are also quite a few “newer” native XML databases like MarkLogic Server, webMethods Tamino XML server, and EMC Documentum xDB [6]. These databases all support XML native storage and search using XQuery amongst other features. To differentiate from XML-enhanced relational database (RDB) systems in this thesis, we will refer to these “newer” XML-only database systems as XML Database (XDB) systems. Given the variety of available RDB and XDB systems, there are some questions one might ask: How does each of them perform? And which system is better, the native XML storage in XDB or the shredding or native XML approaches in RDBs?

This thesis contains five chapters which build towards answering these questions. Chapter 2 presents a brief survey of some aspects of native XML storage. In the following chapters, I draw upon the benchmark, Experimental XML Readiness Test (EXRT) [21], which I co-designed as a part of this thesis work. Chapter 3 presents an overview of the EXRT benchmark, which is a micro-benchmark created to evaluate XML data management tradeoffs between shredded and/or native XML storage and query evaluation techniques. Chapter 3 describes aspects of the experimental environment, data sources and workload for EXRT. With this in hand, Chapter 4 then shows the results of the EXRT benchmark being applied to the native XML features provided in two RDB systems and one XDB system and discuss their implications. Finally, Chapter 5 contains the conclusion and a few suggestions for potential future work.

2 Native XML Storage

Storing XML documents as plain text in CLOBs or shredding them into relational tables has some fundamental limitations. As the limitations have become better known, native XML storage techniques have also been developed and are becoming more and more mature. In fact, we can see many database vendors beginning to recommend storing XML documents in native storage today. Since native XML storage has been studied in research for nearly a decade, there are many different approaches proposed. In this chapter, I will briefly review these approaches as they relate to the following issues: representation, compression, labeling, path processing and indexing.

2.1 Representation

In this section, two types of storage formats and logical data models are reviewed. One format stores XML documents as ordered labeled trees, a method demonstrated by Natix [7] and DB2 [2, 9]. The other format is to represent the XML tree in some linearly serialized format, e.g., as a SAX events stream, which will be illustrated by Oracle XDB [4].

2.1.1 Ordered Labeled Tree Format

In Natix [7], an XML document is stored as an ordered labeled tree structure. The mapping from an XML document to the tree model is as follows: all elements are mapped into tree nodes. When an element has an attribute, it is wrapped in an attribute container that is represented as an additional child node of its containing

element. This attribute container is always the first child of its element node. Attributes, PCDATA, CDATA nodes and comments are all stored as leaf nodes of element nodes.

When the XML data is large, Natix splits the XML tree structure into several partitions, which are then each stored in a single record. Here, a record is a sequence of bytes stored on a single page, which means its subtree must fit on one page. Also, instead of partitioning the data at random positions within the document, the partitions of a Natix document are each meaningful subtrees of the logical tree model. Hence, structure information is retained in every single page. Natix uses so-called proxy nodes and aggregate nodes to link subtrees in different pages together. Figure 2-1 shows the use of such nodes based on an example from the Natix paper [7]. Physical aggregate nodes h1 and h2 are used to group the nodes in records 2 and 3 (r2, r3) into a tree. Physical proxy nodes p1 and p2 are used to substitute for the nodes in r2 and r3, and they contain the record identifiers of r2 and r3 where the subtree nodes are actually stored.

Because Natix subtree partitions are small enough to fit in on one page, insertion and deletion are usually local operations on a page, which makes updating easier. According to their experiments, XML documents stored in the Natix system consume on average about as much space as they consume when stored as plain files in a file system.

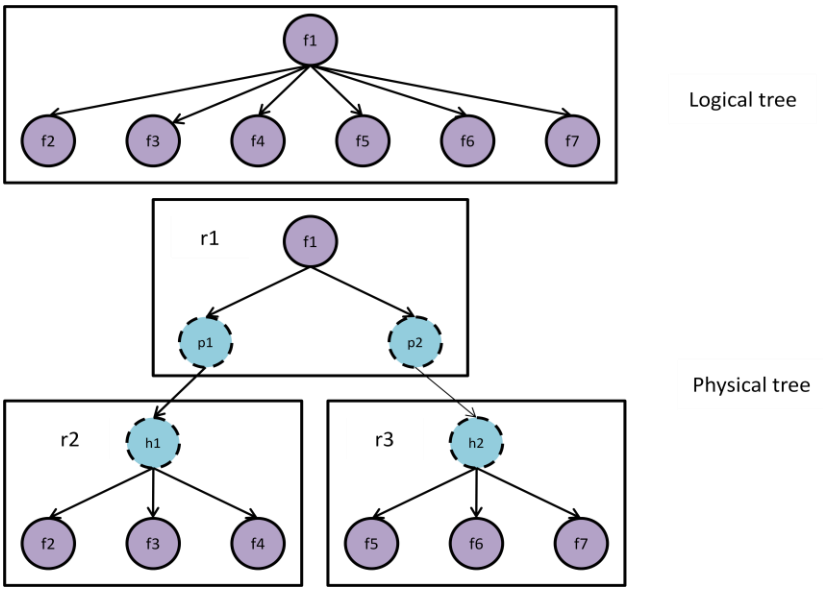

Figure 2-1 One Possibility for Distribution of Logical Nodes onto Records in Natix

On input, IBM DB2 [2, 9] uses a SAX parser to parse XML documents into sequences of parsed SAX events. The parsed SAX events are then converted into an instance of the XQuery Data Model, and are stored in hierarchical, type-annotated XML trees. The physical data format used by DB2 is similar to Natix. One difference is that each node in DB2 contains pointers to both its parent as well as its children to support efficient navigation.

DB2 also splits large data trees into subtrees. Each of the subtrees in DB2 fits in a single page, like what Natix does. The separated subtrees are called regions. Figure 2-2 shows three different regions of a document tree in different colors, the regions R1, R2 and R3. This figure is an example drawn from [2]. The region R1 and R2 are each big enough to occupy one page, while the region R3 is small enough to share a page with region R4 from another document. Instead of the proxy and aggregate nodes used in

Natix, an auxiliary structure called the “regions index” is used to connect the regions (subtrees) in DB2. If the target node is in region R1, then the target can be found without touching the region R2 and R3 by using the regions index. Also, DB2 supports direct access to a node by way of the regions index and its node identifier. Therefore, a lot of physical disk I/Os can be saved by avoiding a full traversal from the root to the target node.

Similar to Natix, since regions in DB2 are stored on single pages and connected by the regions index, only a portion of the document is affected during a document update.

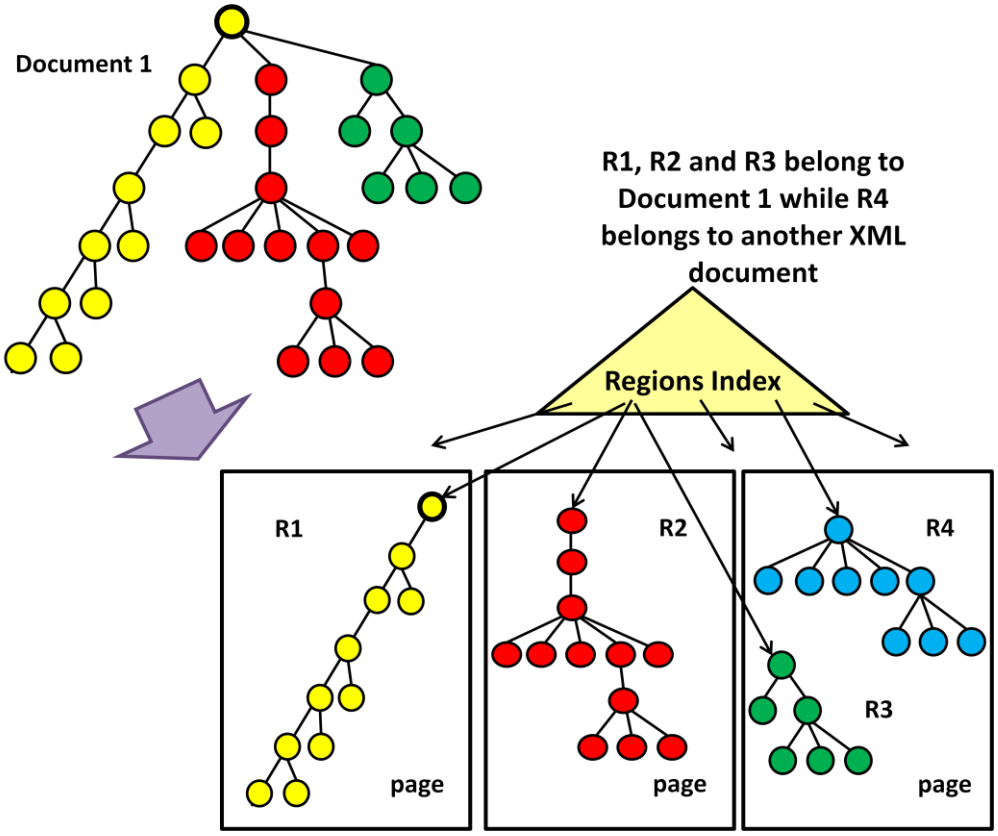

Figure 2-2 Interlinked Document Regions

2.1.2 Serialized Format

Oracle [4] stores XML documents in some linearly serialized format. Like DB2, Oracle also uses SAX parsers to parse XML document into a sequence of SAX events. The SAX events are represented by SAX opcodes. Generally, one opcode corresponds to a single SAX event, such as START_ELEMENT. Oracle persists the SAX opcodes along with encoding of QName, data, schema information and other structural properties. From a high level viewpoint, the Oracle Binary XML format can be viewed as a serialized set of SAX opcodes that are persisted in a logically sequential form [4], not unlike the token stream format used by the BEA XQuery engine [30].

An XML document can also be partitioned into subtrees in the Oracle storage format via a storage concept known as sections. To connect parent and child sections, a section reference is needed. The section reference provides a link from the parent section to the child section. Each section has a header followed by the actual subtree XML data. The tree data is represented as a set of instructions, each consisting of an opcode followed by its operands. The header contains meta information such as the DocID, the identifier of the document to which this section belongs.

In order to process a node in the XML stream directly without processing the stream from the beginning, a locator is needed. The locator contains information about the location/byte offset of the node in the binary encoded stream along with its QName and associated information [12] and serves as its Node Identifier.

2.2 Compression

Approached naively, native XML storage would not be space efficient since there can be many redundant repeating element names, namespace URLs and namespace prefixes. Hence, a number of techniques have appeared to solve this problem, e.g., mapping the name strings into integers, to achieve better XML storage compression. Natix [7], DB2 [2] and Oracle [4] all use this integer-based mapping scheme to solve this problem.

Natix maps XML tag and attribute names to integers, which are used as an alphabet $\sum_{Tags}$. Non-leaf nodes of the XML tree are then labeled with an integer taken from $\sum_{Tags}$. Leaf-nodes (attributes, PCDATA, CADATA, comments) can be labeled with additional arbitrary long strings from another alphabet (e.g., their content) along with an integer from $\sum_{Tags}$. In addition to the integers used for tag and attribute names, some integer values are reserved to indicate attribute containers, text nodes, processing instructions, comments and entity references [7].

In DB2, all tag names and namespace URIs are mapped into integer values called StringIDs. DB2 uses a mapping table to store the names and their corresponding StringIDs. The size of this table is usually small since it corresponds to the number of unique tags in the database, typically only hundreds or thousands [2]. In addition, there is a special-purpose cache used in DB2 to ensure high performance when accessing this table.

Oracle calls its variant of this compression technique tokenization [4]. Tag names (QNames for elements/attributes), Namespace URLs and Namespace prefixes are all mapped to integers. The token definition (mapping information) in Oracle can either be encoded into the document or stored in a token repository.

Figure 2-3 illustrates how this sort of mapping saves space. All long strings, like student names, are substituted by short integers. One last benefit worth noting with this mapping technique is that it renders node comparisons to being just integer comparisons, which naturally improves the CPU performance of navigation.

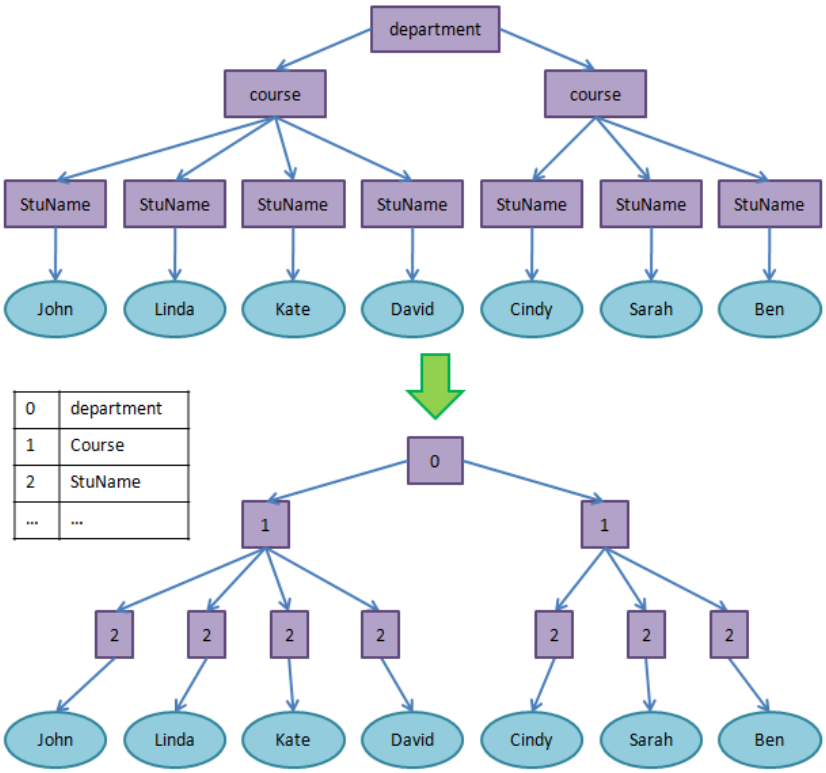

Figure 2-3 Mapping Tag Names to Integers

2.3 Node Labeling

When XML data is represented as an ordered labeled tree, each node requires a unique identifier that can be used for indexing and query evaluation. How to label tree nodes is the problem known as node labeling and is discussed here.

Node labeling techniques have attracted a lot of attention in the XML database research. This is because labeling plays a fundamental role for both storage consumption and efficient support of navigational and declarative query evaluation [11]. Several kinds of labeling schemes exist. One is called the prefix–based labeling scheme, such as Dewey Order [15, 18], ORDPATH [13] and DLN [14]. The XTC system [11, 17] and IBM DB2 both adopted this scheme. Another scheme is called the range-base labeling scheme [16]. Systems like Timber [8] utilize this technique instead.

Knowing this, what are the central ideas of these two schemes?

2.3.1 Prefix-based Labeling

Here, we review the central idea of prefix-based labeling schemes based on the concept of ORDPATH [13].

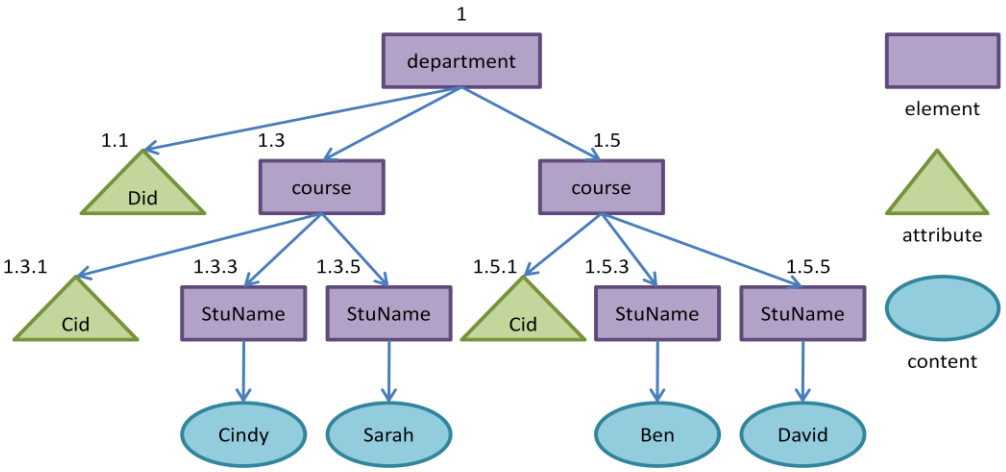

Figure 2-4 Prefix-based Labeling

Figure 2-4 shows the ORDPATH labeling of a XML document. According to ORDPATH, each label has several components (divisions), which are separated by dots. The dots represent the tree edges from the root element to the target element. For example, in the figure, one department has three children, an attribute Did (department ID) labeled 1.1, and two course elements whose labels are 1.3 and 1.5. One dot means there is only one tree edge from the root to these nodes. Here, the last components in the labels indicate a sibling relationship among them. Their numbering shows that Did is the first child, and thereafter are the course elements. Furthermore, from the labels, it is easy to figure out that their parent department element is labeled as “1”, and that the various children of the course element labeled 1.3 have the same prefix as “1.3”.

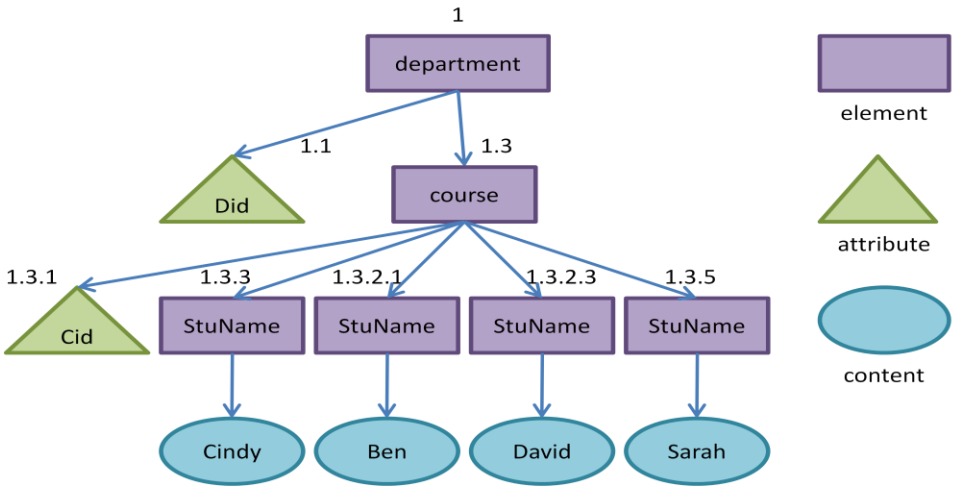

Figure 2-5 Insertion in Prefix-based Labeling Scheme

In Figure 2-4, one can see that only positive and odd integers have been used. Even numbers and negative numbers are reserved for insertion purposes. For example, when new nodes are inserted in Figure 2-5 under the “course” node between the two StuName (Student Name) nodes labeled 1.3.3 and 1.3.5, there is no need to re-order other nodes. These new nodes could be labeled as 1.3.2.1 and 1.3.2.3. Components which contain even numbers are not counted for ancestry. For instance, the new nodes 1.3.2.1 and 1.3.2.3 are the direct children of the “course” node labeled 1.3. The even component only means that this node was inserted after initially labeling the tree. Note that all labels end with odd components enable the successful insertion between any two sibling nodes.

Other prefix-based labeling schemes are similar in this regard, but differ in other aspects like the overflow technique for dynamically inserted nodes, attribute/content node labeling, and encoding mechanism [11].

2.3.2 Range-based Labeling

Figure 2-6 shows the same example document labeled with range-based labels. These labels take the form of (start, end, level). The start and end numbers are fixed by a depth-first traversal algorithm. In the tree traversal, when at first a node is reached, it is given a start number. When it is visited the last time, it is given an end number. As a result, each interval (i.e., start end pair) of a descendant node will be included in its ancestors' intervals. A level number that indicates the node's level in the document tree can be included to help identify the parent/child relationship. The formulas to define the relationship between nodes labeled in this manner are listed below:

Ancestor-descendant relationship: a node (S1, E1, L1) is the ancestor of node (S2, E2, L2) iff S1 < S2 and E1 > E2;

Parent-child relationship: a node (S1, E1, L1) is the parent of node (S2, E2, L2) iff S1 < S2 and E1 > E2 and L1 = L2-1

As an example, the label of the first "course" child of "department" is (4, 15, 2). Its first element child "StuName" is labeled as (7, 10, 3) and the child of "StuName", the content "Cindy", is labeled as (8, 9, 4). Comparing the labels of "course" and "Cindy", (8, 9) is included in (4, 15) which means that "Cindy" is the descendant of this "course". Comparing the labels of "course" and the first "StuName", (7, 10) is also included in (4, 15) and 2 = 3 – 1, which means "StuName" is a direct child of this "course".

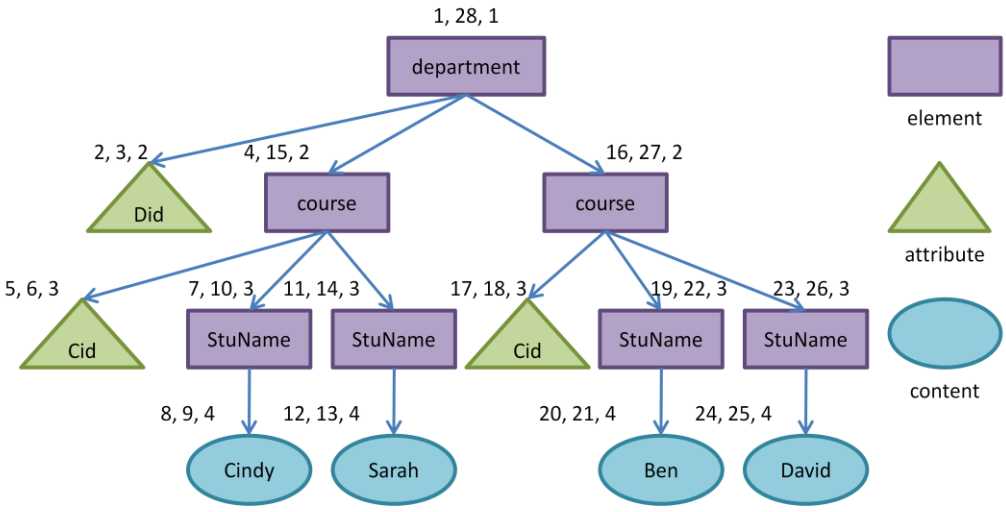

Figure 2-6 Range-based Labeling

2.3.3 Labeling Tradeoffs

Range-based labeling schemes are preferable for static (infrequently updated) XML documents that are deep and complex [16]. This is because their label sizes (only start, end, level) and query performance are not dependent on the document structure. Deep structure is problematic for prefix-based labeling schemes because the lengths of prefix-based labels increase linearly with the depth of the document.

Prefix-based labeling schemes are preferable for dynamic XML documents with more frequent updates [16, 17]. This is because, for prefix-based labeling schemes, a re-labeling is needed only when a new node is inserted between two consecutive labels. On the other hand, any insertion in a range-based labeling scheme will result in a re-labeling of other nodes. (This re-labeling problem though can be mitigated somewhat by adopting an encoding approach [16] for range-based labeling schemes.)

2.4 Path Evaluation

After reviewing how XML can be stored natively, the next question to answer is how queries are evaluated. One important part in XML query evaluation is the navigational approach. As suggested by Zhang, et al [4], there are two basic types of navigational approaches: query-driven and data-driven.

In a query-driven navigational approach, the navigational operator translates each location step in the path expression into a transition from one set of XML tree nodes to anther set [4]. Think of an ordered tree structure; a path step from a parent node to its child actually indicates a transition from the parent subtree nodes to the child subtree nodes. Natix adopts this type of approach [7].

On the other hand, in data-driven navigational approaches like Yfilter [19] and XNav [20], the query is translated into an automaton and the data tree is traversed according to the current state of the automaton. While this data-driven approach can be more complex to implement, it needs only one scan of the input data [4]. Both DB2 and Oracle use this type of approach. DB2's navigational operator is similar to XNav [20], whereas Oracle's operator is similar to YFilter [19]. Below, I will give a basic idea about how Oracle evaluates a path expression.

Path queries are similar to regular expressions. Regular expressions allow a user to do pattern matches. For example, if a string matches the pattern "a*bc", the string must contain the character "a" before character "b" and "c", with any characters allowed in between the "a" and "b". A path expression is much like a string pattern.

Take for example a document that matches the path expression /a/b/c: first, it must have an element “a” as the root node, and then “a” must have a child “b” and “b” must have a child “c”.

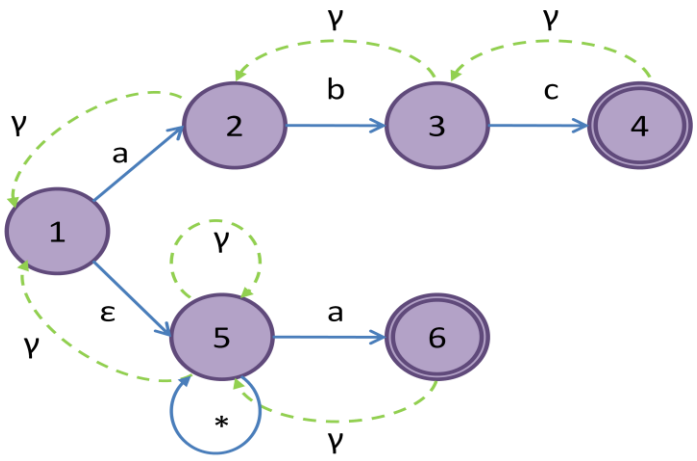

Figure 2-7 NFA Constructed from Paths //a and /a/b/c

Since regular expressions can be evaluated by a finite state machine, Oracle builds a non-deterministic finite automaton (NFA) to process a given set of path expressions. For example, Figure 2-7 shows an NFA constructed from the path expressions //a and /a/b/c. This is an example taken from Zhang et al [4]. When processing a path expression using an NFA, it is possible to combine two or more paths together in a single NFA, as is shown in Figure 2-7. Therefore, it is possible to save some physical disk I/Os by evaluating multiple path expressions from a given XML query in a single pass.

To apply a set of paths, the NFA reads in the XML document a node at a time. After each node is read, the NFA either make a transition to another state or stays put.

This is done node by node until the NFA reaches the final state (state 4 or 6 in Figure 2-7), at which point a match was found. Figure 2-8 diagrams the overall query evaluation methodology [4]. There is a decoder module that decodes the input XML stream and feeds the NFA every event that it reads. Each event read is similar to an XML SAX event, and indicates things like the beginning of the document, the beginning of an element and so on, which decides the state of the NFA. (Many optimizations were made for the input stream decoders in Zhang et al [4].)

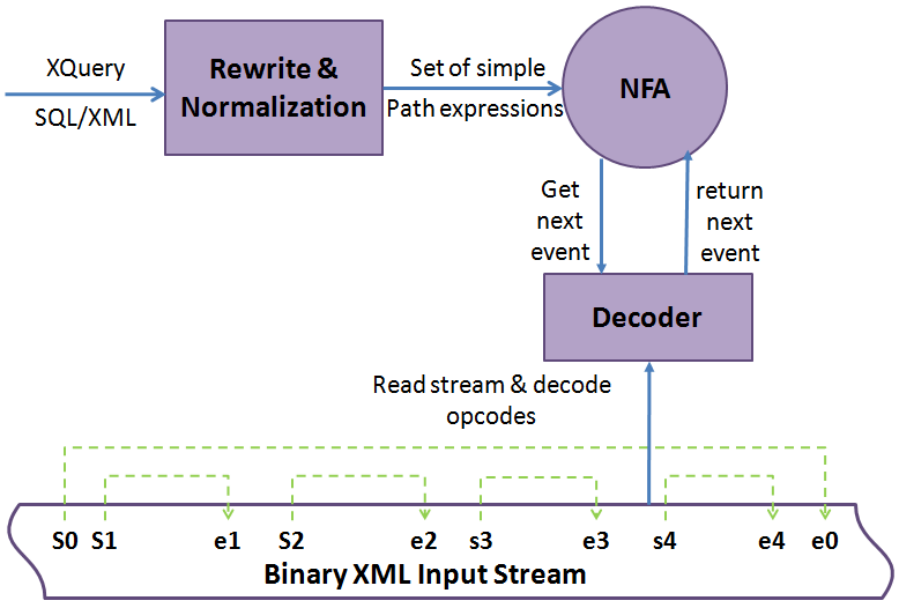

Figure 2-8 Oracle Binary XML Streaming Evaluation Architecture

2.5 Indexing Strategies

Indexing is an important issue for databases. IBM describes three categories of indexes which are used in DB2 for XML documents [9]. The first is the Structural Index, which indexes node names, paths, tag-based expressions or node identifiers. The regions index introduced in section 2.1.1 falls into this category. The second is the Value

Index, also known as path-specific value index, which indexes node values. The third is the Full-Text Index, which indexes tokens (e.g. words) and keeps track of the nodes which contain the token. All in all, each kind of index plays a role in resolving different types of queries.

Internally, an XML index structure in DB2 is implemented using two B+ trees [9]: the path index and the value index. The path index maps the reverse of a path to a PathID. PathID is similar to the concept of StringIDs described in section 2.2. Each unique path is mapped into an integer, PathID, which reduces the size of index entry [2]. It is noted in [9] that the reverse path, namely the path from the leaf to the root, actually serves best for descendant-or-self queries like //book. Such queries will match the prefix of the reverse path, allowing results to be efficiently located in a range scan of the path index. On the other hand, the value index includes the PathID, the value of the node (cast to the declared index type) and a RowID and a NodeID. The RowID indicates the row where the XML document is located in the table, similar to the relational index, and the NodeID represents the label of the target node as a Dewey node identifier [15, 18], another type of prefix-based labeling scheme, and can provide quick access to the target node in the XML store through the regions index without accessing the table [9].

XTC [11, 17] constructs three kinds of indexes using B-tree and B*-tree structures (a variety of B-tree used in the HFS and Reiser4 file systems). One kind of index is called the document index, and it contains key/pointer pairs. The key is the node ID and the pointer points to the first node on each physical page. (The nodes of a document are

stored on several pages in document order in XTC.) With help of the document index, the page on which the target node is located can be found more easily. Another kind of index is called the element index; it consists of the names of all elements and attributes that exist in the document. Each name is followed by a list of node IDs. The third kind of index is called the content index, and it is used for root-to-leaf paths like /bib/book/title; here, path values are also followed by a list of node IDs.

Natix constructs a full-text index using the idea of inverted files [7]. The inverted file is a fundamental concept in information retrieval, and its main purpose is to store a list of document references where each search terms appears. Natix generalizes this idea by storing not only the document references but also some additional context. The indexes in Natix map from search terms to the lists of identifiers that point to the actual lists where the context is stored. The context can include DocumentID, NodeID, offsets and so on.

Last but not least, the native XML database system MarkLogic, like Natix, also uses the idea of inverted files. Not only is each word in the document indexed, but so are the tags from the document's structure information, such as occurrences of a particular element, combinations from a path expression, and so on. The indexes in MarkLogic Server are implemented by hashing words and phrases into their own 'termlists' which each maintain a list of documents that contain a particular word or phrase across the entire domain.

3 Overview of EXRT Benchmark

We developed EXRT [21], a micro-benchmark designed to evaluate the performance and trade-offs between different XML data management systems. The goal of EXRT was to evaluate and drive early commercial native XML storage support like the Wisconsin benchmark [22] did for early relational database systems.

3.1 Related XML Benchmark Work

Most existing XML data management benchmarks have focused on querying large XML documents. These benchmarks perform operations using a multitude of different XQuery features on documents. EXRT adopts a different approach. Instead, EXRT focuses only on the "core" features of XML storage and query processing and on large collections of small (modestly-sized) documents, which are reminiscent of enterprise-oriented use cases like SOA message archiving and querying.

Previous benchmarks can be categorized into two main categories: application-level benchmarks like TPoX [23, 29] and XMach-1 [24], which measure performance based on the scale (number) of documents in the database, and micro-benchmarks like XMark [25], XPathMark [26], and X007 [27], which choose data and queries representing application scenarios. Again, most of these micro-benchmarks exercise relevant features of XQuery only on a single large document. EXRT deviates from these earlier approaches.

Compared to the application-oriented benchmarks, EXRT is a single-user micro benchmark. It measures query processing times and evaluates the impact of various

query characteristics on query times, whereas benchmarks like TPoX and XMach-1 are multi-user and evaluate the overall system performance by measuring the throughput of a complex read/write workload.

Compared to the micro-benchmarks, EXRT uses many small documents and scales up in terms of the total number of documents instead of using a single XML document and scaling in the size of that document. Moreover, EXRT stores XML documents in both native XML storage and relational shredded storage forms and includes test of queries with both XQuery and SQL/XML, while most other benchmarks have only focused on XML and XQuery.

3.2 EXRT Database Design

EXRT is designed based on use cases involving data-oriented XML documents, and it includes sets of queries and updates designed to study performance of enterprise-oriented XML use cases. This section discusses its implementation. Information about the implementation of EXRT can also be found in [10, 21], as I co-implemented EXRT with the author of [10].

3.2.1 TPoX Data Overview

A good data set is a fundamental component of a benchmark. Conveniently, we opted to use the data of the open source benchmark TPoX [23, 29], as it contained a sufficient range of XML features to satisfy EXRT's needs. Along with TPoX data, we also utilized its data generator, Toxgene [28], to generate sets of synthetic data-centric XML documents for EXRT. The data set we generated contains 600,000 XML documents.

TPoX models a financial situation, namely a hypothetical brokerage house. It has three types of documents. The first is a CustAcc document that contains information about a customer and their accounts. The second is an Order document that contains buy/sell orders. The third is a Security document which contains securities information. Typically, a CustAcc document ranges from 4 to 20 KB, an Order document is between 1 to 2 KB, and a Security document is about 3 to 10 KB [29]. The Order documents are modeled based off of a real-world standard XML schema, FIXML, while CustAcc and Security documents are modeled based on experience with real world financial data.

For EXRT, we choose to focus only on the CustAcc data in TPoX. The rationale for this was because we found its structure to be simple enough to map its XML schema to a relational schema. Moreover, CustAcc documents also contained all of the key features that we felt we needed to develop the queries and updates to evaluate performance trade-offs in EXRT.

3.2.2 CustAcc Document

Figure 3-1 shows the schema of the CustAcc document. The information in each CustAcc document includes:

- Name, including title, first name, last name, suffix and when possible, multiple short names and middle names.

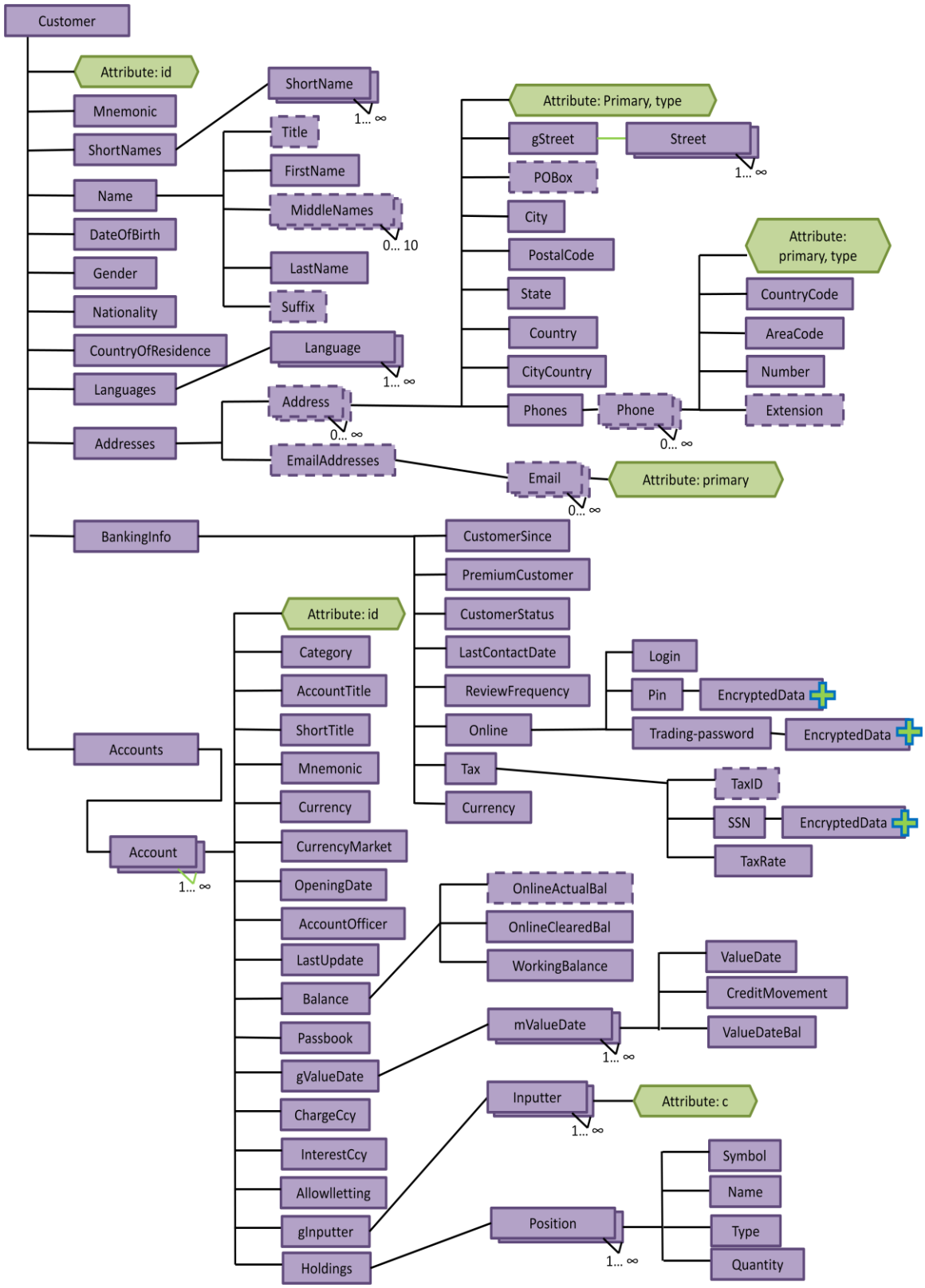

Figure 3-1 XML Document Schema

- Date of birth, gender, nationality, country of residence, languages spoken, bank interaction dates and customer type.

- Addresses, including all details of each address and the corresponding phone numbers.

- Account information, with a variety of account properties like account category, title, currency, balance, opening date, and account officer. It also includes the details of the holdings that belong to an account.

The CustAcc document schema contains both simple and complex data type. The complex data types include the Name Type, Address Type and the Holding Type. Elements can appear one or more times. For instance, most elements like gender only appear once, short name and language elements can appear once or multiple times, and some elements like middle name are optional and can be empty. Besides this, CustAcc documents also contain uniquely-valued nodes, like customer ID and account ID, which can be used for indexes.

3.2.3 Relational Schema for EXRT

EXRT stores XML documents in both native and shredded storage formats. In native storage, each XML document is held either in a column of a row of a table in a relational database or as an XML instance in a collection in an XML database. In the shredded storage case, EXRT has to map the XML schema of a CustAcc document to a reasonable relational schema. Although there are more details in [10] about EXRT's

relational shredded storage, I will briefly introduce its relational schema here as it's important to be aware of its structure in order to understand the way that EXRT characterizes its queries' features. Guided by the principle of "as few as possible", we created twelve tables. As indicated by Shao in Figure 3-2 [10], the referential relationship designated by each arrow represents the element containment hierarchy of the XML schema:

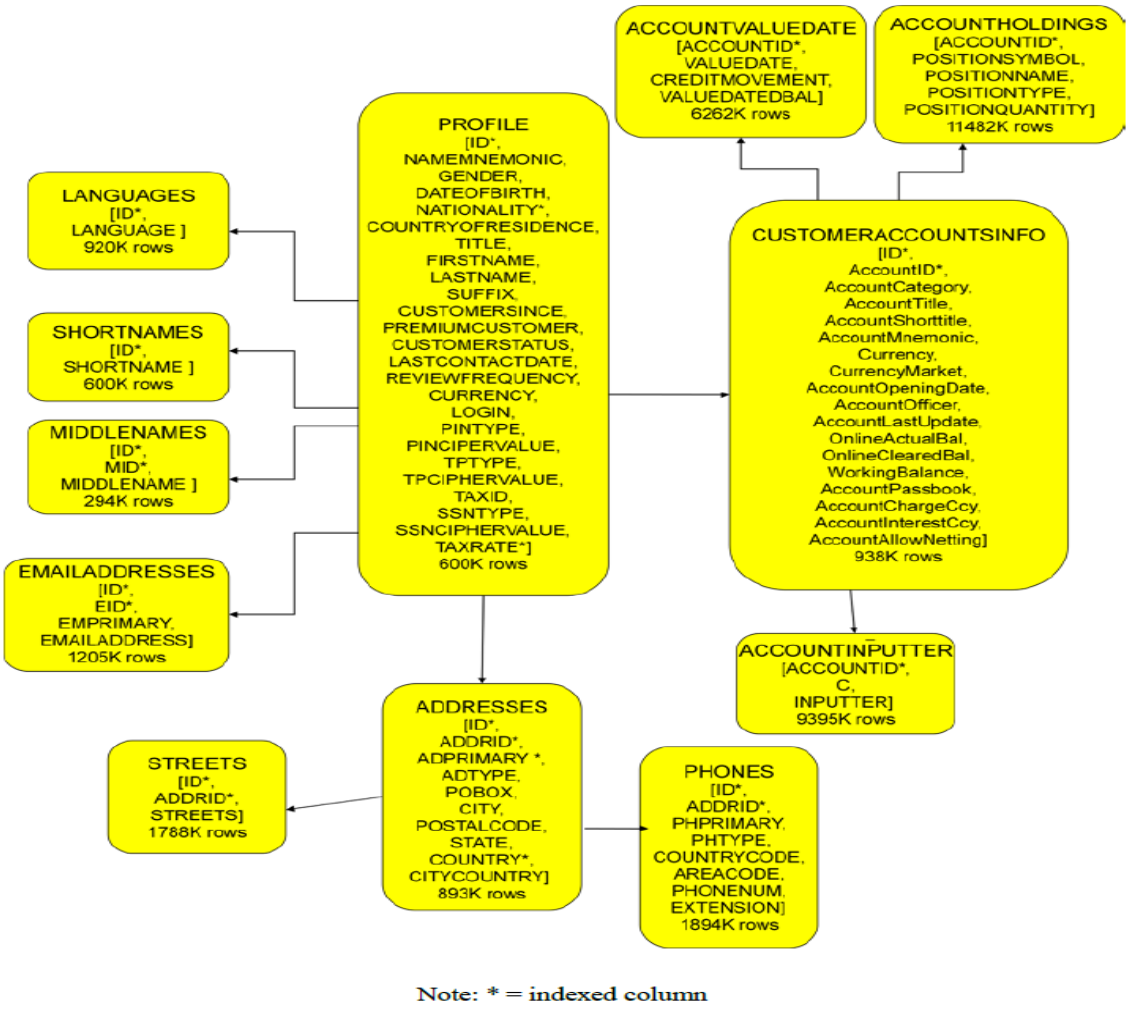

Figure 3-2 Relational Schema

3.3 EXRT Workload

As mentioned earlier, the goal of EXRT is to examine XML storage, query processing performance and the impact of various data features on performance. Therefore, we created a set of queries and updates representative of what might be typical in SOA or web applications.

3.3.1 EXRT Query Features

One query feature that EXRT varies is how many XML documents are selected in a query's results, a concept known as selectivity in the relational query processing area. Here, we utilize the same concept with another name to characterize EXRT queries: query height. In EXRT, we say that when a query's height is h, the query returns h XML documents.

Another EXRT query feature tested concerns how much of the content is extracted from an XML document. For example, when shredding XML documents into twelve tables, we can characterize how much content is extracted by the number of tables touched in the returned results. We term this concept with another name: query width. When the width of an EXRT query is w, it means that the query result needs to be constructed from the content of w tables in EXRT relational storage. This metric effectively measures the fraction of content in each document extracted. Also, note that since the same XML results are returned for each query from both native storage and shredded relational storage, the notion of width can be useful to calculate for native

storage, to indicate how many nested (repeating) elements are extracted from the XML document.

Figure 3-3 illustrates the notions of “height” and “width” for queries over native XML storage, with each triangle representing an XML document:

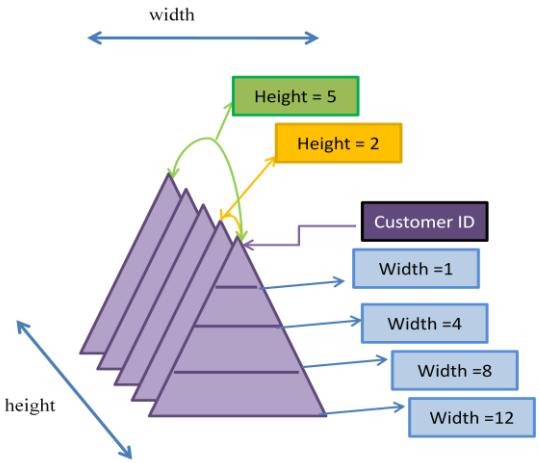

Figure 3-3 Metric: Height and Width

3.3.2 EXRT Queries

Nine queries compose the bulk of the EXRT benchmark. This section discusses each of them, grouped by the differing features that they exercise.

3.3.2.1 Queries 1-4

The first four queries in the EXRT benchmark share a common trait in that they all use an integer Customer ID as a parameter. Together the queries either return a portion of the documents or the entire documents, as illustrated in Figure 3-4 below. The Customer ID is an attribute node in the root element of the document. This is beneficial since it is relatively trivial to navigate down to this attribute. Moreover, it is

also straightforward to create range indexes on this attribute to improve the query performance.

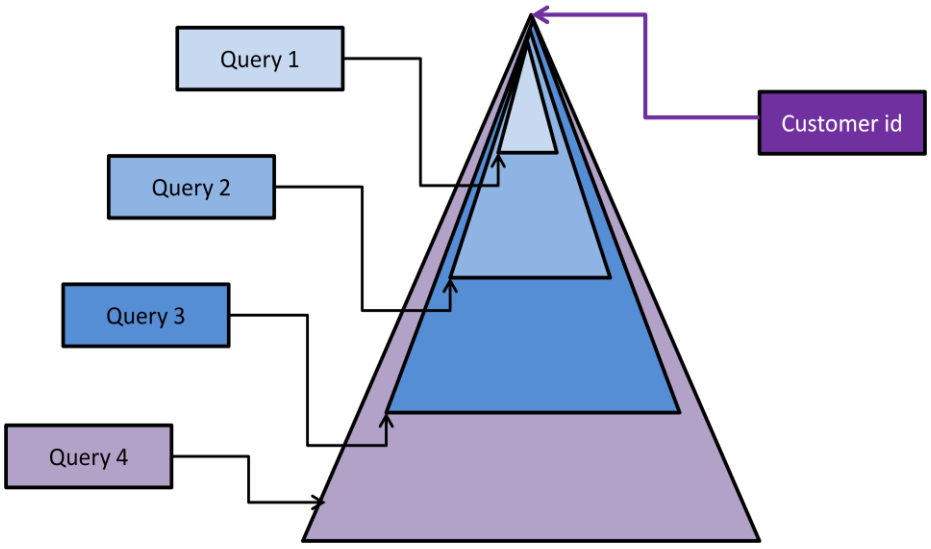

Figure 3-4 Queries 1-4 Vary with Width

A description of Queries 1-4 is given in Table 3-1. Note that these queries increasingly subsume one another. For instance, the returned content of Query 2 includes the results of Query 1, Query 3 includes the content of Query 2, and Query 4 includes the contents of all of them completely. Height and width are varied as well. When a query's height is 1, it means we pass only one customer ID as parameter and test for exact match lookups. Greater heights are achieved by specifying a range of Customer IDs, and range-based ID lookups are then tested instead.

Fetching and assembling of the requested information was tested in native storage. For fetching, increasing numbers of elements are extracted from the base CustAcc documents. For assembling, new XML document are constructed from the

output results. The cost of these queries for native storage is the sum of the cost of document selection, navigation, element extraction, and the construction of new result documents. (For shredded storage, not tested here, their cost involves selection, and result construction [21].)

Op	Description	Width
Q1	Given customer IDs, fetch the customers' *minimal profile* - consisting of their customer IDs, titles, first and last names, and suffixes	1
Q2	Given customer IDs, fetch the customers' *basic profile* - adding middle and short names and languages to the minimal profile	4
Q3	Given customer IDs, fetch the customers' *complete profile* - adding e-mail info, addresses, streets, and phones to the basic profile	8
Q4	Given customer IDs, fetch *all of the customers' information* - including all of the info about their accounts and holdings	12

Table 3-1 Queries 1-4

For the client-side implementation of these queries, parameter binding in JDBC was used to pass parameters to the RDB systems tested. However, there were some cases where it was not possible to pass parameters through JDBC. We used a literal string substitution technique to pass parameters for those cases. For the XDB system,

we used its proprietary Java API, which is similar to JDBC, and the parameter binding technique was always used.

Query 4 has two variants in the case of native storage. The first is simply to return the whole CustAcc document as one document element, and the second is to extract all elements of the document and then construct a separate result document containing those elements. Although extraction and reconstruction might not be necessary, we can use the two variants of Query 4 to understand the effort required for performing construction.

3.3.2.2 Queries 5-7

Queries 5-7 (see Table 3-2) are related in that they all extract and return a sub-document as results instead of performing a construction from extracted XML elements. Query 5 uses an attribute of the root element as the parameter and returns its nested child objects (Accounts). Queries 6 and 7 also use an ID attribute as a parameter, but it is the ID attribute of a nested child object (Account ID). One could consider the Account ID as being in the "middle" of documents, with the path to the attribute being, /Customer/Accounts/Account/@id, as illustrated in Figure 3-5.

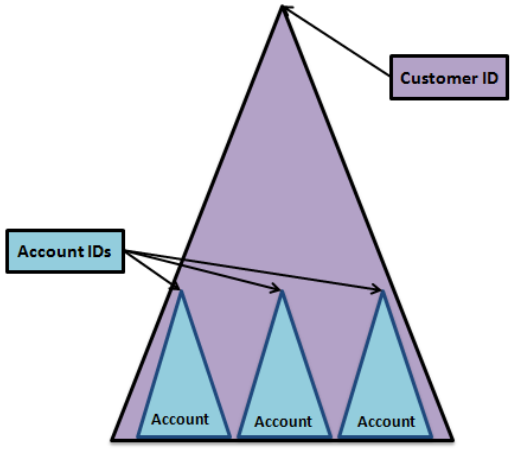

Figure 3-5 Queries 5-7

Op	Description	Width
Q5	Given *customer IDs,* get the complete info for all of their *accounts*	5
Q6	Given *account IDs,* get the complete info for the *accounts*	4
Q7	Given *account IDs,* get all of the info for the *account-owning customers*	12

Table 3-2 Queries 5-7

For Queries 5-7, the width cannot vary since it is fixed by the content of the selected fragment. However, we can still vary the query height by choosing the range of Account IDs. *

* The Account ID is a non-consecutive 10 digit numeric-string. This means we cannot, for example, simply add 600 to the starting point to get the end point if we want the height to be 600. Instead, what was done is that we pre-computed the start and end Account IDs that actually contain 600 accounts in between. These pairs were written to a text file that was subsequently read in as parameters later for Queries 6 and 7.

3.3.2.3 Queries 8, 9

Relational databases can easily handle aggregation queries via the aggregation functions provided by SQL. Something that seemed interesting is how well native XML storage and querying fares with the same sorts of aggregation queries. Queries 8 and 9 represent two simple summary queries with multiple predicates. Some predicates are exact-match predicates, like nationality, and others are range predicates like tax rate.

Op	Description	Width
Q8	Get the average number of accounts for customers of a given nationality	1
Q9	Given a country name and a tax rate, return the average account balance for customers in the specified country who have a tax rate greater than the specified tax rate	3

Table 3-3 Queries 8-9

3.3.3 Updates

The EXRT benchmark also tests basic insert, delete and update operations. The insert (I) and delete (D) operations are described in Table 3-4. The width of each operation is 12, denoting a new document insertion or deletion.

Op	Description	Width
I	Given an XML string containing all of the information for a new customer, insert the new customer into the database	12
D	Given a customer ID, delete all info about this customer and the corresponding accounts	12

Table 3-4 Insert and Delete

Node-level insert (NI) and delete (ND) operations are also described in Table 3-5. Here, the idea is to insert or delete pieces into existing XML documents. Some limiting factors here were the EXRT (TPoX) XML document schema and data generator. Due to these factors, only 0 to 3 addresses were allowed for one customer. We had to do some work to find only those Customer IDs having 1 or 2 addresses, writing them randomly to a text file for use later as a parameter file for the node update operation tests. There, a Customer ID is sequentially read in from this parameter file when these statements are executed. Similarly, there are also constraints for the number of emails and accounts per customer. A single Customer ID parameter file is generated for all node update and delete operations to satisfy the schema requirements.

Op	Description	Width
NI1	Given a customer ID and an XML string with a new Address element, add the new address to the specified customer	3
NI2	Given a customer ID and XML strings with a new Address element and a new e-mail address, add both to the specified customer	4
NI3	Given a customer ID and XML strings with a new Address element, a new e-mail address, and a new account, add them to the customer	8
ND1	Given a customer ID plus an integer positional indication (1, 2, or 3), delete the indicated Address node from the customer's list of addresses	3
ND2	Given a customer ID plus positional indicators for their address and e-mail lists, delete the indicated Address and Email nodes	4
ND3	Given a customer ID, an account ID, and positional indicators for their address and e-mail lists, delete the indicated Account, Address, and Email nodes	8

Table 3-5 Node Insert and Node Delete

For node insert and delete operations, width varies depending on the update content chosen. Unlike queries, the only dimension used is width; updates were only made to one XML customer document. EXRT can be improved in the future by adding more variation of heights for these operations.

Op	Description	Width
NU1	Given a customer ID, update the customer's last contact date	1
NU2	Given a customer ID, a contact date, and the name of a new account officer, update the last contact date, upgrade the customer to premium	2
NU3	Given a customer ID, a contact date, the name of a new account officer, and an XML string with a list of addresses, update the customer's last contact date, upgrade the customer to premium status, update the assigned account officer's, and replace the customer's current list of addresses with the new list	5

Table 3-6 Node Update

Besides full document insert/delete and node-level insert/delete, node-level updates (NU) operations are also tested (see Table 3-6). Put simply, node-level updates replace certain elements with new elements. They change either the top level customer information or information nested within the customer, such as an address and account. The widths of these operations are proportionate to the changed content. Like node inserts and deletes, the information modified in those EXRT update statements with larger width also contains (is a superset of) the information modified by the statements have smaller width. Same as before, a parameter file is also used here; it contains the

random customers who had at least one of such elements to modify. Again, in these tests, the Customer ID is then sequentially read in from this parameter file.

3.3.4 Query Languages

One key difference between existing micro-benchmarks and EXRT is that EXRT tests both XQuery and SQL/XML language that are supported in the current databases. One might think that XQuery is to XML data as SQL is to relational data, but that isn't exactly true. Instead, XQuery is a complete functional programming language that is intended to do much more than just retrieve information from a database. Another query language, SQL/XML, was previously designed for SQL programmers who might not want to learn all of XQuery. For relational vendors, SQL/XML support is more mature than XQuery support because commercial relational databases started to support SQL/XML much earlier than XQuery.

To illustrate the nature of the EXRT queries, Tables 3-7 and 3-8 demonstrate two syntactically different versions of Query 1. The first is an XQuery version for native XML storage, and the second is a SQL/XML version for native XML storage.

```
XQUERY
declare default element namespace “http://tpox-benchmark.com/custacc”;
for $cust in db2-fn:xmlcolumn(‘CUSTACC.CADOC’)/Customer[@id >= |1 and @id <
|2  + |3 ]
return element Profile {
                    attribute CustomerId { $cust/@id },
                    element Name {
                                $cust/Name/Title,
                                $cust/Name/FirstName,
                                $cust/Name/LastName,
                                $cust/Name/Suffix
                                        }
                              }
```

Table 3-7 Sample XQuery Query over XML Native Storage for Query 1

```
SELECT XMLQUERY(‘
      declare default element namespace “http://tpox-benchmark.com/custacc”;
      for $cust in $cadoc/Customer
            return element Profile {
                  attribute CustomerId { $cust/@id },
                  element Name {
                        $cust/Name/Title,
                        $cust/Name/FirstName,
                        $cust/Name/LastName,
                        $cust/Name/Suffix }
            }
      ‘ PASSING CUSTACC.cadoc AS “cadoc”)
FROM CUSTACC
WHERE XMLEXISTS(‘
      declare default element namespace “http://tpox-benchmark.com/custacc”;
      $cadoc/Customer[@id>=$id1 and @id<$id2 + $tallness]’
      PASSING cadoc AS “cadoc”, cast (? As int) as “id1”, cast(? As int) as “id2”,
cast(? As int) as “tallness”)
```

Table 3-8 Sample SQL/XML Query over Native XML Storage for Query 1

3.3.5 Indexes

To improve performance, indexes were created to speed up query execution. For native XML storage in the RDB systems, indexes for Customer ID, Account ID, “primary” attribute, Country of Address, Nationality and TaxRate were created. For the XDB system, however, only three range indexes - for Customer IDs, Account IDs and TaxRate - were needed. This is because the XDB system tested has support for full-text search by indexing each term in a hash table, where the term and the position where that term appears are indexed automatically. In our implementation, the ‘primary’ attribute, Country of Address and Nationality all need to be compared as exact matches. Since these kinds of ‘terms’ are already automatically indexed by XDB, only the other three range indexes, were explicitly needed for Customer ID, Account ID and TaxRate.

3.4 Experimental Setup and Testing Procedure

The EXRT benchmark was run on two commercial RDB systems and one commercial XDB system. For purposes of anonymity related to the licensing agreements of the databases, we refer to the two RDBMs as RDB1 and RDB2 and the XDB system simply as XDB.

3.4.1 Hardware

The EXRT tests were run on a Dell desktop system with a dual-core 3.16 GHz Intel® Core DUO™ E8500 CPU, 4GB of main memory, and a pair of 320 GB 7200 RPM Western Digital disks. The operating system was Red Hat Enterprise Linux Client release 5.4 (Tikanga), kernel 2.6.18-164.el5. The database instances for each system were

created on a striped Linux file system volume in order to utilize both disks. Holding 100GB each, the striped disks were used to store all data files and logs [21].

3.4.2 Configuration

We wanted to compare each system in a fairly comparable way. The simplest way to achieve this would be to let each database use as many of the system's resources as it can to allow for the best performance. To reach this goal, a number of suggestions from the respective vendors were taken into consideration and implemented.

For the RDB systems, we chose the direct I/O option for both relational systems to skip file system caching. Also, we opted for both systems to utilize their automatic buffer management feature, so that each could make use of all available memory on the system. Another important configuration was the database page size. Based on each vendor's suggestions, an 8k database page size was used for RDB1 and a 32k page size was used for RDB2.

In our XDB experiment setup, the direct I/O option was also turned on as well. We let the system choose its own settings and cache sizes which, by default, are automatically set to take use of all available memory. Unneeded XDB full-text search index options were turned off to improve performance.

3.4.3 Testing Procedure Details

We used two test conditions for each query, “cold” and “hot”, a concept that we borrowed from the 007 Benchmark [27]. For the “cold” test condition, query execution time was measured in a clean buffer pool, and randomly selected parameters were bound to queries. Data and index pages were flushed between runs (as discussed more below) to achieve a clean buffer pool. Other auxiliary information such as the query plan cache and catalog information remains in the buffer pool to be reused. For the “hot” test condition, query execution time was measured several times in a row without clearing the buffer pool and the same parameters are reused each time. Here, data pages remain in the buffer pool and the database server can access such data pages without physical disk I/Os.

By providing results for both the “hot” and “cold” testing conditions, we can model the best and worst case for database applications. In real-world applications, systems are neither entirely “cold” nor entirely “hot”. This is because data volumes are usually larger than the main memory, and require a random amount of physical disk I/Os to get the data from disk to buffer pool. Our “hot” and “cold” test results thus provide upper and lower bounds on expected performance in real world.

For RDB1, the data caches, plan caches and catalog information are stored in a shared buffer pool. Under guidance of the vendor, a special database system command was used to clean only the data caches in the buffer pool for the “cold” runs.

RDB2, on the other hand, had the property that after closing all connections to the database server, the buffer pool would be cleared. Since our benchmark implementation only used a single connection to the database server through JDBC, cleaning the buffer pool could be achieved by closing this connection. However, this raised another problem - by cleaning all the information in the buffer pool, it meant also clearing catalog information. Although this might seem to be a problem, it was mitigated due to queries being fully prepared before they are actually executed, and only the actual execution and result fetching stages are timed. During preparation of the query (JDBC prepare statement function), the catalog information is brought into memory and the query plan is computed.

For XDB, the vendor actually added a cache-clearing feature specifically for our tests, and they found it useful enough that they plan to ship it in a future release. With the help of this cache-clearing function before execution of a query, all data caches and index caches can be quickly cleared and the server can be rendered "cold" for our testing purposes.

We tried to bind EXRT's query parameters at the preparation stage instead of doing literal string substitution for the parameters wherever possible. This enables reuse of the query plan generated for a query template (no actual parameters are assigned), which improves the query performance. Although there are benefits to doing this, it was not always possible to use the parameter binding method, as mentioned in

section 3.3.2.1. In those cases, including node updates in RDB1 and XQuery queries in RDB2, literal string substitution was alternatively used.

The time we measure and report includes the query execution time plus the time to fetch query results from server side to client side. To obtain stable results, each query/update is repeated ten times. The first run is discarded to guard against system specific “warm up” behavior from interfering. Also, the fastest and slowest runs of the nine remaining executions are discarded. The final time is the average of the seven remaining time measurements. Figure 3-9 below describes the general testing procedure in pseudo-code form.

```
FOR each query in list
        Get connection;
        Prepare all the queries in list;
        IF in cold condition
                FOR each ith cold run, i < NumOfColdRun  // NumOfColdRun  =10
                        Get start_time;
                        Run this query with random parameter;
                        Read each result in result set to a local variable;
                        Get end_time;
                        IF i != 0 // this is not the first 1 cold run
                                Add (end_time – start_time) to ColdRunTimeSum;
                                Compare (end_time – start_time) with MaxCold and
                MinCold to get max and min cold run time;
                        ENDIF
                        Clear buffer pool to ensure cold condition;
                ENDFOR
        ENDIF
        IF in hot condition
                FOR each ith hot run, i < NumOfHotRun  // NumOfHotRun  = 10
                        Get start_time;
                        Run this query with the parameter used in last cold run;
                        Read each result in result set to a local variable;
                        Get end_time;
                        IF i != 0 // this is not the first 1 hot run
                                Add (end_time – start_time) to HotRunTimeSum;
                                Compare (end_time – start_time) with MaxHot
                        and MinHot to get max and min hot run time;
                        ENDIF
                ENDFOR
        ENDIF
        Close connection;
        Average_ColdRunTimeForThisQuery = (ColdRunTimeSum – MaxCold –
MinCold) / (NumOfColdRun – 3);
        Average_HotRunTimeForThisQuery = (HotRunTimeSun – MaxHot –
MinHot) /( NumOfHotRun – 3);
ENDFOR
```

Figure 3-6 Pseudo Code of Testing Procedure

4 EXRT on XDB vs. Native RDB XML

This chapter presents the EXRT experimental results for XML native storage on the three systems.

4.1 Queries with XML Construction or Full Document Retrieval

The upcoming charts in this section follow the same format. In each case, there are three charts in a row. From left to right, each represents RDB1, RDB2 and XDB. Two legends are in the charts: “XQuery on XML” and “SQL/XML on XML”. The former denotes XQuery queries over an XML column (RDBs) or collection (XDB), whereas the latter denotes SQL/XML queries which are supported only in the RDB systems.

Recall that there are two kinds of Query 4. One is denoted as Q4 and the other is denoted as Q4 (Re). Q4 (Re) extracts all elements from the XML document and then constructs a new XML result document based on these elements. Q4 reads the XML document as a whole piece, without element extraction and reconstruction. Here, we show the results of both Q4 (Re) and Q4, so that Q4 (Re) can be used as a reference to Q1, Q2 and Q3; Q1, Q2 and Q3 all have to do extraction and reconstruction.

Figure 4-1 shows the response time for Queries 1-4 in milliseconds and with a selectivity of one. Comparing the cold versus hot cases, one finds that, as expected the hot runs are overwhelmingly better than their cold counterparts. Obviously, this is because hot runs execute the same query with the same parameters repeatedly over data that has been cached.

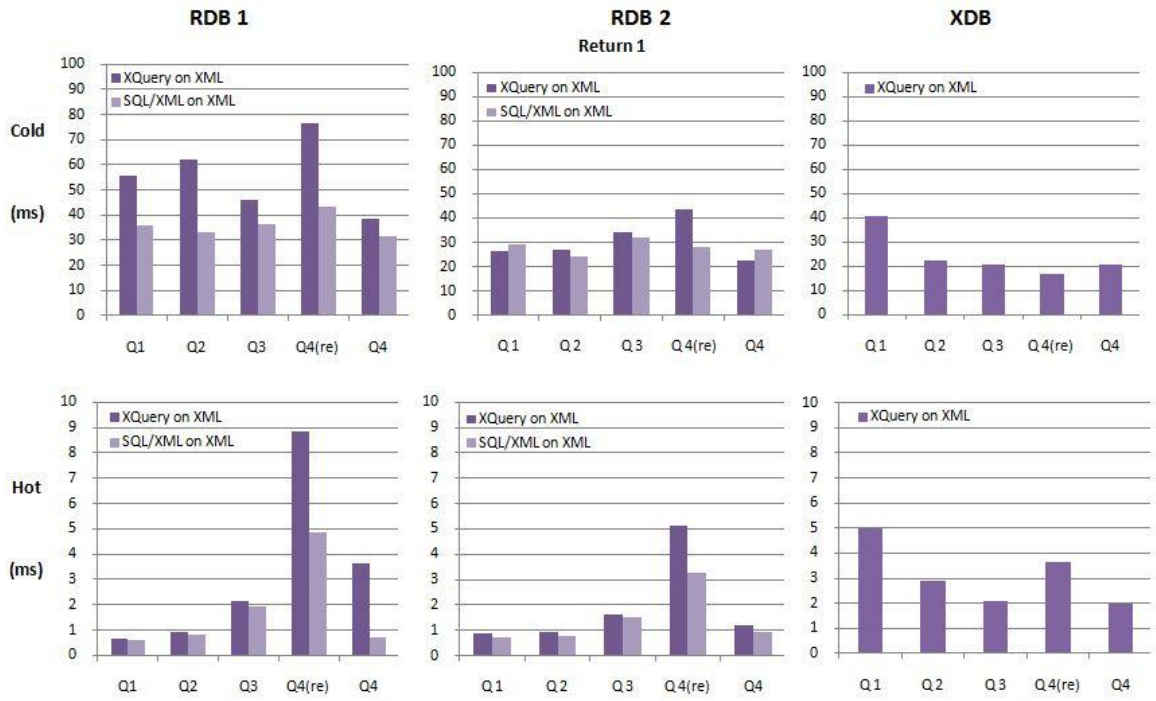

Figure 4-1 Queries 1-4 Response Times in Milliseconds (Result set size = 1)

Also as we expected, the response time increases when the width of the query becomes larger for the 2 RDBs. This is because, when the width becomes larger and larger, more and more XML data needs to be (re)constructed. Two factors contribute to the cost of this reconstruction. The first is the CPU cost for XPath navigation and reconstructing the elements into an XML document. The second is the overhead of feeding results from the server side to the client side. As the width gets larger, the CPU cost and the overhead of shipping results grows, and ultimately the elapsed time gets longer. This trend is much clearer in the hot run results, as the I/O cost also plays an important part in cold run results. Again, the lack of a clear trend in the cold run results of this figure is due to the fact that only one document needs to be reconstructed. The trend will be clearer in later charts, whose result set is larger.

For the two RDBs, the performance of SQL/XML queries is generally better than that of the XQuery queries. This might seem logical because RDB systems have been supporting SQL/XML for much longer than XQuery. Looking more closely, we found that the performance difference between XQuery and SQL/XML is much bigger in RDB1. Ideally, the query plan should be the same for the same query using XQuery or SQL/XML, which we found to be true for RDB2. However, this wasn't true for RDB1 initially. The reason was that an optimization technique utilized in the SQL/XML queries wasn't being applied to XQuery queries over non-schema-based XML data. The workaround was to rewrite the queries in a different manner, with predicates like /Customer[@id > 1002 and @id < 1004] rewritten with the attribute taken out of the predicate, yielding /Customer/@id[. > 1002 and . < 1004].

Figure 4-2 shows the results for the same queries when they return sixty documents. Compared to the results of returning a single document, the execution time increased. That makes sense as the I/O cost, the CPU cost and the overhead of results shipping all increased.

Comparing the difference between every two adjacent queries, one may notice the difference gets larger when result set increased from 1 to 60. This is because more elements needed to be reconstructed. Also, it makes the trend of an increasing cost due to greater width (more elements), more clear, even for XDB. From inspection, one can find a fairly clear trend of increasing from Q1 to Q4 (Re) and then decreasing to Q4 even for the XDB hot run results.

Comparing the RDBs and XDB, one thing that should be observed from our results is that the XDB system appears to be more streamlined (i.e., has lower cost) in terms of the CPU cost for parsing and handling XML data. The hot run results expose the reconstruction effort better than the cold run results. The detailed results are as follows: For the hot run results, from Q1 to Q4 (Re), the elapsed time of RDB1 increased from 22.6 ms to 231.8 ms, a 209.2 ms difference. For RDB2, it increased from 18.5 ms to 214.6 ms, with a 196.1 ms difference. For XDB, it increased from 11.2 ms to 43.9 ms, a 32.7 ms difference. This is not too surprising since the XDB system was built specifically for XML, while XML support was added as more of an afterthought to the two RDB systems; as a result, the CPU path lengths for the RDB systems were not optimized for XML.

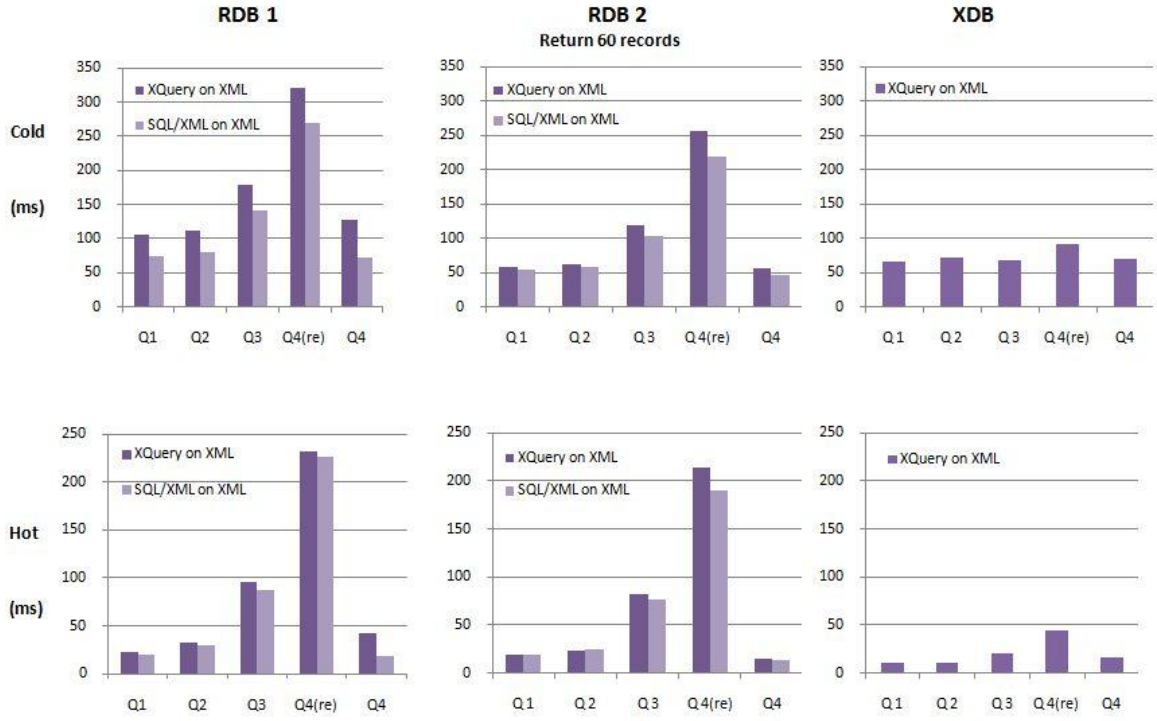

Figure 4-2 Queries 1-4 Response Times in Milliseconds (Result set size = 60)

Comparing Q4 and Q4 (Re), we see that retrieving an XML document as a single unit (Q4) is much faster than reconstructing the document from its individual nodes (Q4 (Re)). Also, we can see that the performance of retrieving a full XML document without reconstruction is slightly better in RDB2 than that in XDB for both the cold and hot cases.

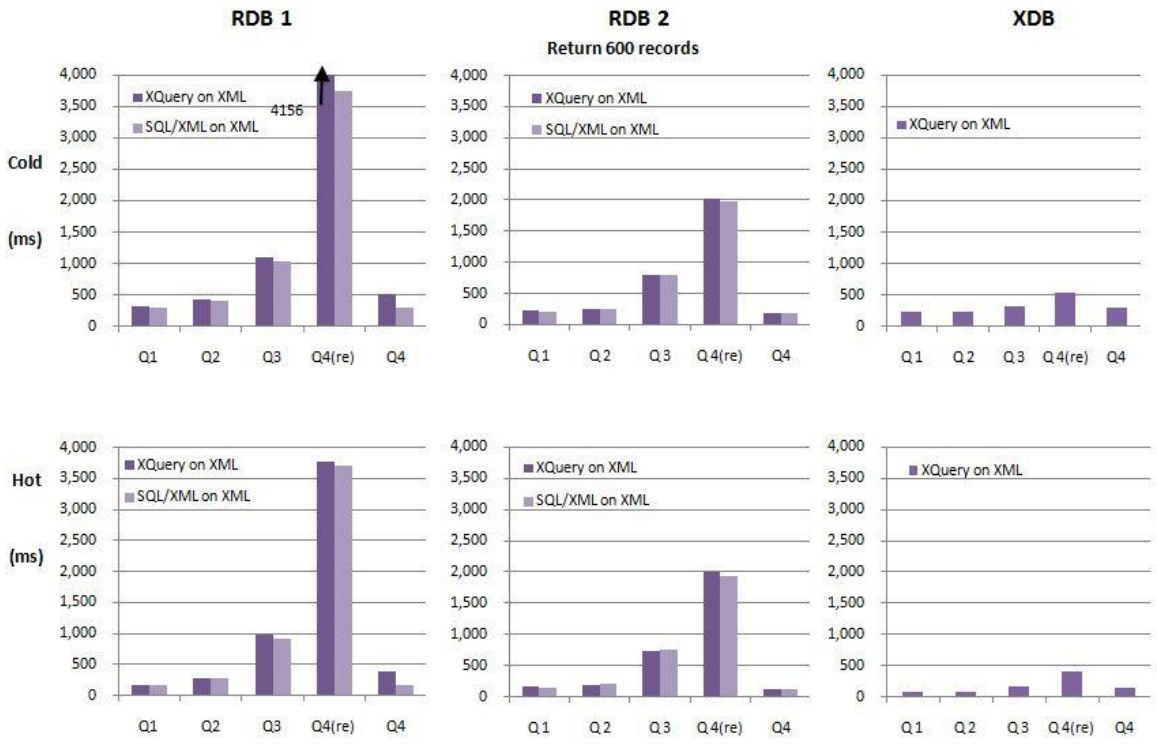

Figure 4-3 Queries 1-4 Response Times in Milliseconds (Result set size = 600)

Figure 4-3 illustrates the same experiments but with ten times the height. Notice the black arrow in the cold run result for RDB1. It indicates the result for Q4 (Re) is 4156ms. (We mark the arrow because the result is slightly off the scale of the chart.)

Comparing the results of the cold runs and hot runs in Figure 4-3, we see that the difference between their performances has diminished. That may be due to

reconstruction being time consuming and becoming the bulk of the work. The CPU cost of reconstructing 600 XML documents and the overhead of shipping 600 XML documents from server to client begin to take a dominant role here. This diminishing trend was even stronger when the result set was further increased to 6000, whose results chart is omitted for brevity.

Like the result charts before, Figure 4-3 demonstrates the same (but clearer) trends as expected. We see this same increasing time when width is increased, and XDB still outperforms the two RDBs on Q3 and Q4 (Re) and RDB2 still performs better than XDB on the full XML document retrieval without reconstruction.

4.2 Queries that Extract Partial Documents

A key difference between Queries 5-7 and Queries 1-4 is that there isn't reconstruction from many elements to form a full XML document. In Queries 5-7, an element node is extracted, yielding a full sub-document as a result. Query 5 extracts the "accounts" fragments for a given range of Customer IDs. Query 6 extracts the "account" fragments with the specified Account IDs. Query 7 returns the full XML CustAcc documents which include the specified Account IDs.

Here, range indexes were configured for both Customer ID and Account ID. The difference between the two was that Customer ID was indexed as an integer, whereas Account ID was indexed as string, according to the schema.

There are two other differences that should also be pointed out between Query 5 and Query 6. The first regards their index and XPath navigation properties. Query 5

can use the Customer ID index to locate an XML document and then to navigate down to its Accounts element. Query 6 uses the Account ID index to locate the Account element directly. Thus, Query 5 needs extra navigation effort. The second difference is that the physical result size of Query 5 was a little bit larger than Query 6. Here, Query 5 returns the "Accounts" element (plural) whereas Query 6 returns one "Account" element. According to the schema, one Accounts element has 1-7 accounts and there's a 69.2% chance that only 1 account exists in its Accounts element [29].

The difference between Queries 6 and 7 is that Query 6 extracts only "account" fragments while Query 7 returns full XML documents. And as the Account ID index links directly to the Account element, Query 6 needs less effort to locate the target element (Account itself) while Query 7 has to navigate back up to locate its target element, the parent node.

Figure 4-4 shows the response times for Queries 5-7 when the selectivity is one. One thing to notice is that there are some high response times for the XQuery case for RDB1 in the charts. This is because the query plan for the XQuery implementation of Q6 is suboptimal. Another thing to notice is that the response times for the XQuery and SQL/XML hot run test of RDB2 are similar. The difference between them in the hot tests is negligible for RDB2 and less than 0.1 milliseconds. This is what we expected, because identical query execution plans are generated for equivalent XQuery and SQL/XML queries.

In the cold tests, XDB performs a little better than RDBs, whereas in hot tests, the SQL/XML performance of the two RDBs and XQuery in RDB2 outperform XDB.

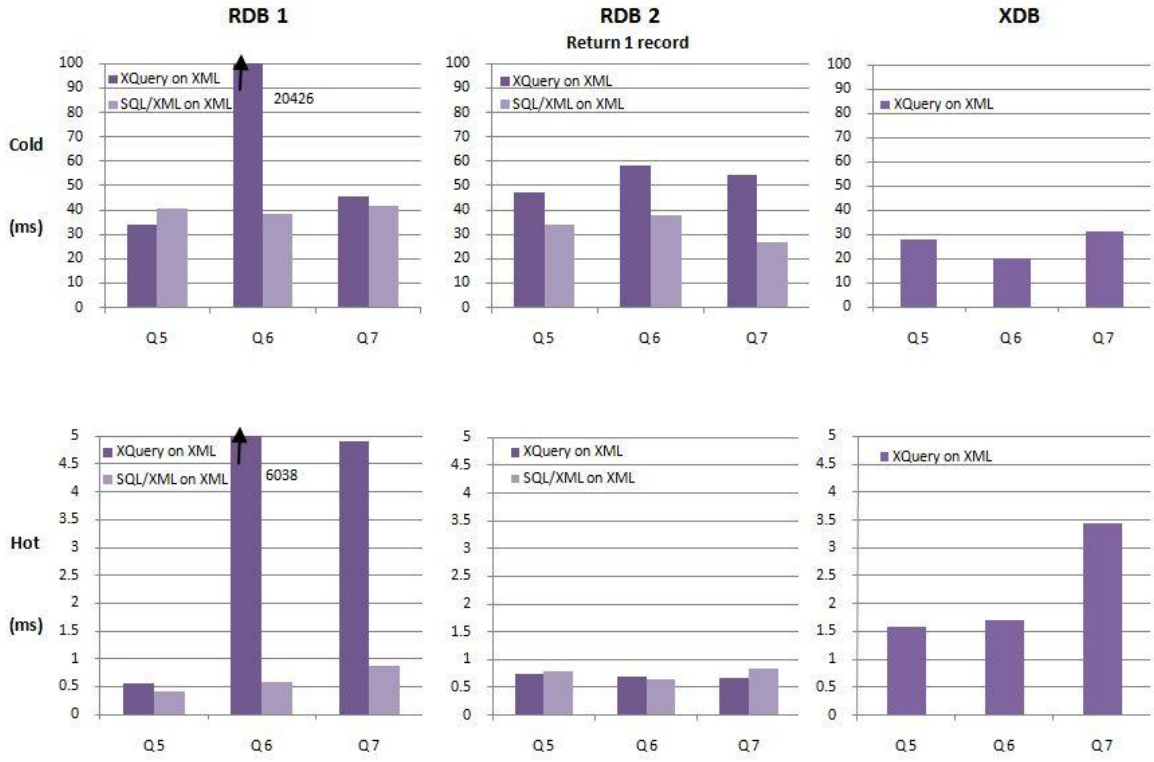

Figure 4-4 Queries 5-7 Response Times in Milliseconds (Result set size = 1)

Figures 4-5 and 4-6 show the response times of Queries 5 through 7 when result sets are larger. The hot run results in Figure 4-5 and all results in Figure 4-6 displays a trend that Query 5 and Query 7 both cost more than Query 6 (excluding the XQuery performance of Q6 in RDB1, which resulted from a suboptimal query plan). As I explained above, Queries 5 and 7 involve more XPath navigation and their result sets are physically larger than that of Q6. Therefore, as the query height increases, Queries 5 and 7 cost more than Query 6.

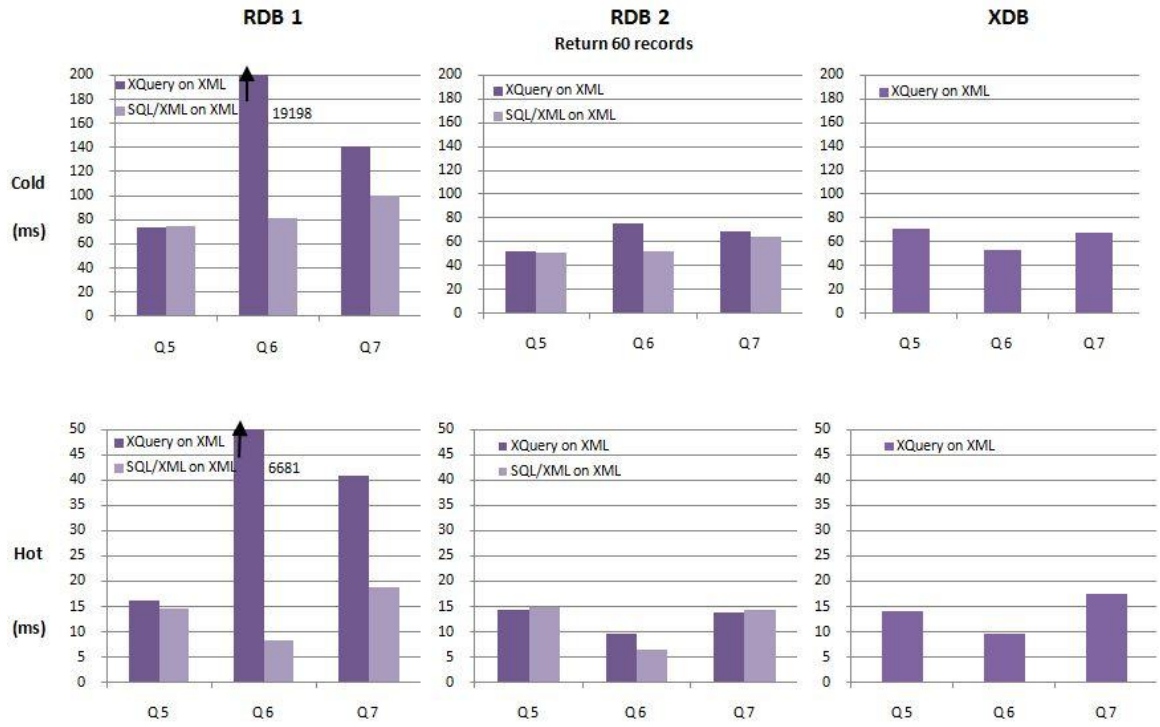

Figure 4-5 Queries 5-7 Response Times in Milliseconds (Result set size = 60)

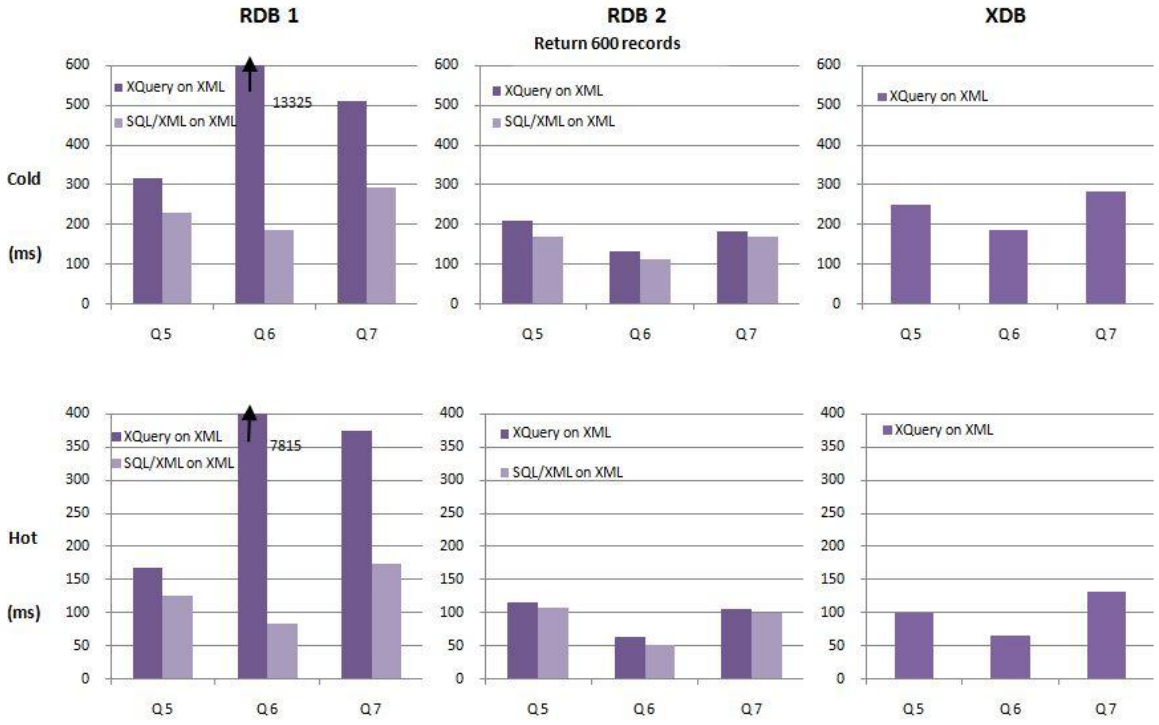

Figure 4-6 Queries 5-7 Response Times in Milliseconds (Result set size = 600)

4.3 Aggregation Queries

Q8 and Q9 are basic analytical queries with several predicates for selection and aggregation of a metric of interest. The predicates include Nationality, Country and TaxRate. Nationality and Country are exact-match predicates, while TaxRate is indexed as a double range index and used with a greater-than condition. Therefore, these queries need to access a large subset of (unclustered) documents. Hence, the performance of random access into the collection of data was critical in these tests.

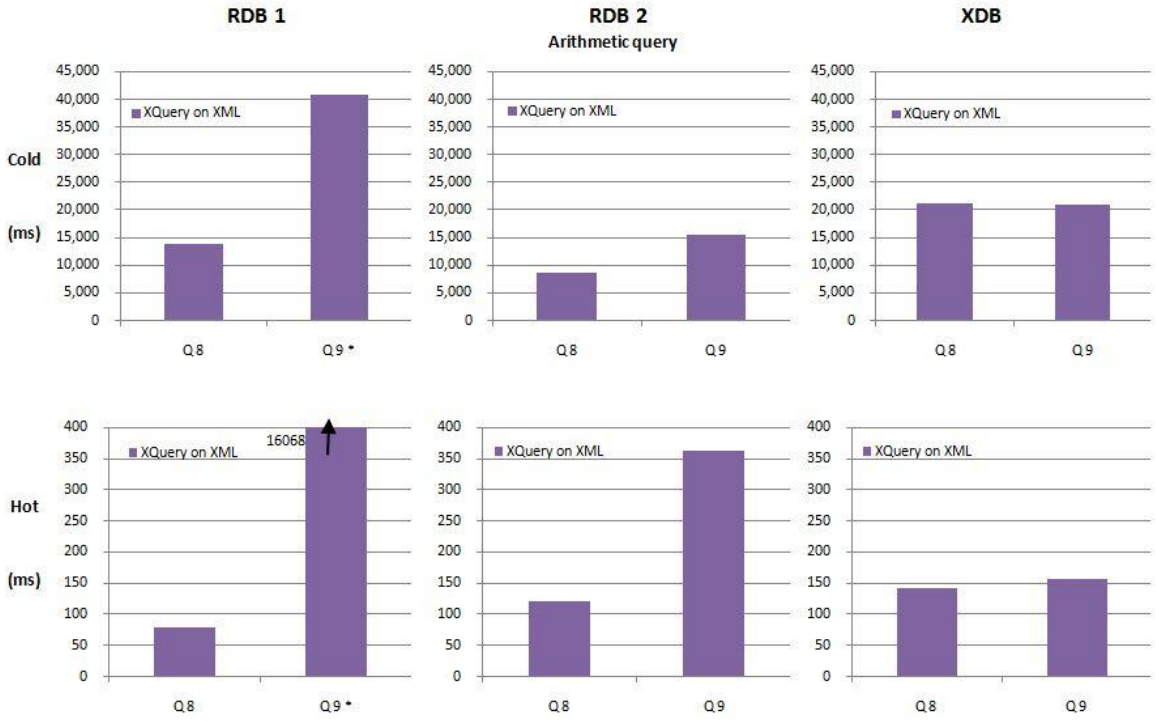

Figure 4-7 Queries 8, 9 Response Times in Milliseconds

For Q8, both RDBs outperform XDB in both cold and hot tests. For Q9, XDB outperformed RDBs in the hot tests, while RDB2 outperformed it in the cold tests. The vendor for the XDB suggested some ways to improve the aggregation query performance by storing some pre-computed auxiliary information in database similar to

materialized views. However, we didn't try this as we strived to make the queries ad hoc and the benchmark fair for all 3 systems.

4.4 Insert, Delete and Update

Figure 4-8 shows the elapsed time for inserting and deleting one full XML document on a cold system. Note that hot tests are not included here, since inserting and deleting the same document repeatedly wouldn't make sense. Consequently, the results for insert, delete and update only included operations on one document and these are the preliminary results for the insert, delete and update operations on XML documents.

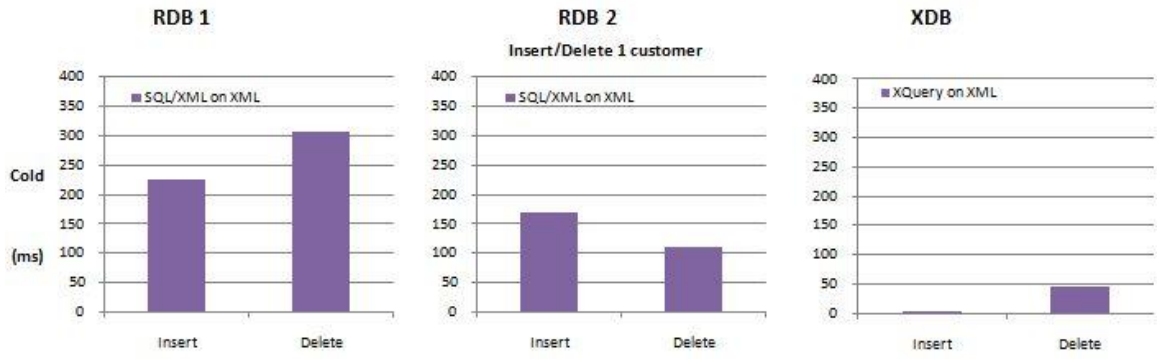

Figure 4-8 Response Times of Inserting/Deleting a Single Customer Document (Milliseconds)

In XDB, we found the costs of both insert and delete to be significantly lower than that of the RDBs. This is because the insert is initially performed in memory, and the only I/O is a single log write in these measurements, something reflective of what one sees in the figure (4ms, a reasonable time for 1-2 I/Os). The eventual I/O to update the document on disk occurs later, outside of EXRT's measured time. It would be

beneficial perhaps in future work to refine these tests to better account for all of the implied I/O costs.

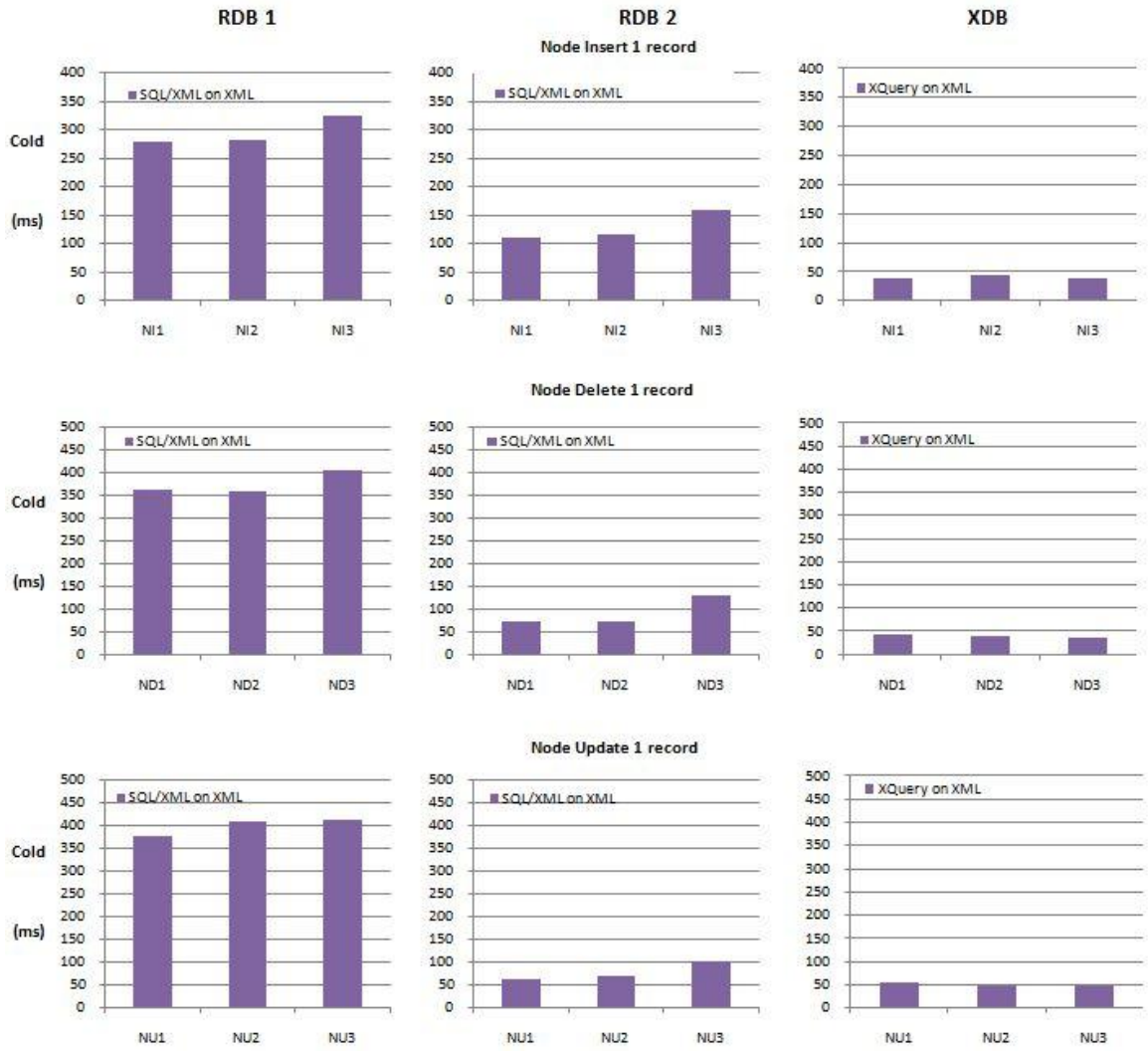

Figure 4-9 Response Times of Inserting/Deleting/Updating Nodes in One Existing Customer Document (Milliseconds)

Figure 4-9 shows the elapsed time for inserting, deleting and updating element nodes within one XML document. These insert, delete and update operations are each a transaction which explicitly calls “commit” at the end of modification. Each insert,

delete and update transaction was run ten times and the average cost was calculated seven times, the same as for the queries.

From the figure, one finds that the cost tended to increase from NI1 to NI3 for both RDB 1 and RDB 2. The width of NI1 is 3; NI2 is 4 and NI3 is 8. It seems that the cost increases when the width increases for RDBs. One possible reason is related to the XML parsing time for the input XML fragment, which is used to update original XML document in database. As the width increases, the bigger XML fragment needs to be parsed. However, the width does not seem to affect the performance of node insert, node delete and node update for XDB.

5 Conclusion and Future Work

This thesis began with a survey of native XML storage technology. The survey covered two typical storage formats and labeling schemes; it also touched on the main techniques used in compression, path evaluation, and indexing for XML data. The thesis then turned its attention to the design of an XML benchmark, called the EXRT benchmark, with a focus on its use in the evaluation of native XML storage alternatives. EXRT (Experimental XML Readiness Test) is a micro-benchmark that has been designed to evaluate XML data management tradeoffs, particularly the impact of query characteristics on the performance of XML storage methods and query evaluation techniques.

Using EXRT, the performance of XML native storage was compared for two relational database systems and one XML database system. No system was found to perform the best in all tests. Both similar and different performance behaviors were observed in the three systems. Further, two different query languages, XQuery and SQL/XML, were compared between the two relational database systems. Interestingly, our benchmark's readiness test discovered some issues related to one vendor's XQuery optimization.

For future work based on EXRT, one step would be to improve the way of testing the insert, delete and update operations to better evaluate the systems' true update performance over XML storage, described in sections 3.3.3 and 4.4. Also, since EXRT's data is focused on data-oriented XML documents, another avenue for future work could

be in testing XML data management performance over content-oriented XML data, like that found in large-scale document management and automated publishing. Finally, since only two relational databases and a single XML database were tested in our work, it could be useful to have more systems undertake our tests to find potentially interesting results.

References

[1] Bray, T., Paoli J., Sperberg-McQueen, C. M., Maler, E., Yergeau, F.: Extensible Markup Language (XML) 1.0 (Fifth Edition), November 2008, http://www.w3.org/TR/xml.

[2] Nicola, M., Linden, B.: "Native XML Support in DB2 Universal Database", *31st International Conference on Very Large Databases,* VLDB 2005.

[3] Nicola, M., John, J.: "XML Parsing, A Threat to Database Performance", *International Conference on Information and Knowledge Management* (CIKM), 2003.

[4] Zhang, N., et al.: "Binary XML Storage and Query Processing in Oracle 11g", *35th International Conference on Very Large Data Bases*, VLDB 2009

[5] Rys, M.: "XML and relational database management systems: Inside Microsoft SQL Server 2005", *SIGMOD* 2005

[6] Bourret, R.: "XML Database Products", http://www.rpbourret.com/xml/XMLDatabaseProds.htm

[7] Fiebig, T., et al.: "Anatomy of a Native XML Base Management System", *28th International Conference on Very Large Databases,* VLDB 2002

[8] Jagadish, H.V. et al.: "TIMBER: A Native XML Database", *28th International Conference on Very Large Databases,* VLDB 2002

[9] Beyer, K., et al.: "System RX: One Part Relational, One Part XML", *SIGMOD* 2005

[10] Shao, L.: "Comparing Shredded and Native XML Data Management Approaches in Relational DBMSs", MS Thesis, Computer Science Department, University of California, Irvine, 2010

[11] Härder, T., Schmidt, K.: "Usage-driven storage structures for native XML databases", 12th *International Database Engineering & Applications Symposium,* IDEAS 2008

[12] Liu, ZH., et al.: "Towards a Physical XML independent XQuery/SQL/XML Engine", 34th *International Conference on Very Large Databases,* VLDB 2008

[13] O'Neil, P., et al.: "ORDPATHs: Insert-Friendly XML Node Labels", *SIGMOD* 2004

[14] Böhme, T., Rahm, E.: “Supporting Efficient Streaming and Insertion of XML Data in RDBMS”, *Proc. 3rd Int. Workshop Data Integration Over The Web (DIWEB),* 2004, pp. 70-81

[15] Dewey, M.: Dewey Decimal Classification System. http://frank.mtsu.edu/~vvesper/dewey2.htm

[16] Xu, L., Ling, T.W., Bao, Z., Wu, H.: "Efficient Label Encoding for Range-Based Dynamic XML Labeling Schemes", in Proc. DASFAA (1), 2010, pp.262-276.

[17] Härder, T., Mathis, C., Schmidt, K.: "Comparison of Complete and Elementless Native Storage of XML Documents", in Proc. IDEAS, 2007, pp.102-113.

[18] Tatarinov, I., et al.: “Storing and Querying Ordered XML Using a Relational Database System”, SIGMOD 2002

[19] Y. Diao, et al.: "Path sharing and predicate evaluation for high-performance XML filtering", *ACM Trans. Database Syst.*, 2003, pp.467-516.

[20] Josifovski, V., Fontoura, M., Barta, A.: “Querying XML Streams”, VLDB 2005

[21] Carey, M.J., Ling, L., Nicola M., Shao, L.: “EXRT: Towards a Simple Benchmark for XML Readiness Testing”, *Second TPC Technology Conference on Performance Evaluation & Benchmarking (TPC-TC)*, 2010

[22] DeWitt, D.: “The Wisconsin Benchmark: Past, Present, and Future”, in *The Benchmark Handbook for Database and Transaction Systems (2nd Ed.)*, Morgan Kaufman, 1993.

[23] Nicola, M., Kogan, I., Schiefer, B.: “An XML Transaction Processing Benchmark”, ACM SIGMOD 2007

[24] Böhme, T., Rahm, E.: “Multi-User Evaluation of XML Data Management Systems with XMach-1”, *Lecture Notes in Computer Science (LNCS),* Vol. 2590, 2003, pp. 148-159.

[25] Schmidt, A., Waas, F., Kersten, M., Carey, M.J., Manolescu, I., Busse, R.: “XMark: A Benchmark for XML Data Management”, *International Conference on Very Large Data Bases (VLDB)*, pp 974-985, August 2002.

[26] Franceschet, M.: “XPathMark - An XPath benchmark for XMark Generated Data”, *XML Database Symposium (XSYM)*, 2005.

[27] Carey, M., DeWitt, D., Naughton, J.: “The 007 Benchmark”, *SIGMOD Conference*, 1993.

[28] ToXgene - the ToX XML Data Generator， http://www.cs.toronto.edu/tox/toxgene/.

[29] XML Database Benchmark: Transaction Processing over XML (TPoX), http://tpox.sourceforge.net/

[30] Florescu D., Hillery, C., Kossmann, D. Lucas, P., Riccardi, F., Westmann, T., Carey, M.J., Sundararajan, A., Agrawal, G.: “The BEA streaming XQuery processor”, *30th International Conference on Very Large Databases,* VLDB 2004.