UNIVERSITY OF CALIFORNIA,
IRVINE

Comparing Shredded and Native XML Data Management Approaches in Relational DBMSs

THESIS

submitted in partial satisfaction of the requirements
for the degree of

MASTER OF SCIENCE

in Computer Science

by

Lin Shao

Thesis Committee:
Professor Michael J. Carey, Chair
Professor Chen Li
Professor Sharad Mehrotra

2010

Table of Contents

List of Figures

List of Tables

Acknowledgments

First, I would like to express my sincere gratitude to my advisor Prof. Michael Carey for his continuous support and encouragement of my study and research in the past two years and for his motivation, enthusiasm, great patience and also enormous knowledge. His guidance helped me in all the time of study, research and writing of this thesis.

Besides my advisor, I would like to thank the rest of my thesis committee, Prof. Chen Li and Prof. Sharad Mehrotra for their comments on the thesis and their help during my study here in UCI.

I would like to thank Matthias Nicola from IBM for his support, great patience, enormous time investment, and valuable suggestions on this project.

I wish to thank my fellow labmate Ling Ling for her contributions on the design and implementation of the query part for our benchmark, for stimulating discussions, and for the sleepless nights we were working together before our conference paper deadline. Also, I want to thank my friends David Zhao and Hanlin Ouyang for encouraging me over those difficulties and for their priceless friendship.

Last but not the least, I would like to thank the software engineers from the companies whose systems I used for this research and who helped me on this project. Without their valuable suggestions, I can’t imagine when this thesis could have been finished.

Abstract of the Thesis

Comparing Shredded and Native XML Data Management Approaches in Relational DBMSs

By

Lin Shao

Master of Science in Computer Science

University of California, Irvine, 2010

Professor Michael J. Carey, Chair

XML database functionality is becoming more mature over time, both in native XML database and relational database products. Exploration of new approaches for storing and managing XML information, as the major functionality of a database with XML-support, has become a serious and ongoing challenge to all database system vendors. Several benchmarks have been established to evaluate different aspects of XML database performance. However, there is still no benchmark to evaluate the performance of XML database functionality as compared to relational database functionality. This thesis will assess the current state of XML and XQuery support in commercial relational database systems. It will also present EXRT

(Experimental XML Readiness Test), which is a new XML benchmark designed to methodically evaluate XML data management tradeoffs, such as the impact of query characteristics on the relative performance of shredded versus native XML. This benchmark has been implemented and comparisons have been made of two commercial relational database systems. Valuable results have been obtained which illustrate the relative performance of different approaches and methods of processing XML (even within the same system) as well as the importance of some of the factors that the performance depends on. The new benchmark, which differs from previous XML micro-benchmarks, tests basic insertion, deletion and update performance, in addition to read-only (query) performance.

1. Introduction

With the rapid expansion in the usage of XML data, a growing need for reliable systems to store and provide efficient access to these data has triggered the development of XML support in relational database systems. This has drawn considerable attention with a view to leveraging their powerful and reliable (i.e. proven) data management services. In order to store an XML document in a traditional relational database, the tree-structured schema of an XML document-type must first be mapped to some sort of equivalent flat relational schema. Instances of XML documents can then be "shredded" and loaded into these mapped tables. Finally, at runtime, XML queries must be translated into SQL and submitted to the Relational Database Management System (RDBMS). The query results must then be translated into XML.

An alternative approach to supporting XML is to truly integrate XML into the database system. Support for native XML data storage has recently become available in most commercial RDBMSs. For example, the IBM DB2 Universal Database has been enhanced with comprehensive native XML support under a commercial name DB2 pureXML [1]. Oracle XML DB provided native XML storage and retrieval technology, and this functionality is also now a feature in the Oracle Database Server as well [2]. MS SQL Server 2008 also provides native

XML support [3]. Finally, MySQL 5.1 and 5.5 have been enhanced with some rudimentary XML functionality [4]. In the native approach, XML documents can be stored as whole documents in tables with user-defined columns of type XML.

Given the current state of XML support, it is interesting to ask which XML database solution offers performance better today for various XML operations: shredded XML storage or native XML storage. A comprehensive evaluation of the functionality of the XML storage systems should include database performance, ease of use, scalability, automatic tuning, etc. In current industrial application scenarios, a web application or SOA application would not know much detail about the storage format behind the service. Given a request with XML content as input, the output results can be published as XML format. A web application does not really know what the storage type is. To begin with some simple XML performance comparisons, a simulation of a client-server system with a single-user application scenario is appropriate. Only the performance of the most common operations will be considered in this thesis.

In this thesis, EXRT (Experimental XML Readiness Test) [5] is developed and implemented, which is a simple micro-benchmark that methodically evaluates the impact of basic XML query characteristics on the choice of shredded and native XML storage types. In the first part of this thesis, background information is briefly reviewed, and comparisons are made

between previous benchmarks and the proposed EXRT benchmark. Next, in the benchmark design section of this thesis, the main idea for the EXRT benchmark and its details will be presented. In the experimental setup section, the experimental setup and test procedure will be elaborated. In the results and analysis section, a set of experimental results based on the XML data management facilities offered by two relational database products will be described. Finally, the lessons section will cover some other valuable findings related to the relative performance of different ways of processing XML (even within the same system) as well as some of the key factors that their performance depends upon.

2. Background

Before introducing the EXRT benchmark, some background information about XML and XML support must be discussed. Afterwards, the objectives and values of the EXRT benchmark will be described.

2.1 Document-Centric XML versus Data-Centric XML

Uses of Extensible Markup Language (XML) can be broadly divided into two categories: (a) document-centric, in which XML is widely used in the large-scale electronic publishing industry, and (b) data-centric, in which XML also plays an increasingly important role in the exchange of a wide variety of data on the web and elsewhere. Nowadays, many web applications and services have connections to databases and use XML to transfer data between databases and web applications.

Document-centric XML applications have different requirements than do data-centric applications. In document-centric XML applications, XML documents are often treated as resources (files and directories) in a content repository which needs to support access control, revisions, versioning, etc. In most cases, document-centric XML documents are relatively large. The number of documents can also be enormous. Document schemas may be complex, fluid, or unknown. A document-centric XML store will keep the elements in order and utilize the

hierarchy based on the original narrative. Figure 2-1 is an example of document-centric XML data.

```
...
<section>
<section.title>Procedure</section.title>
<section.content>
The patient was taken to the operating room where she was placed
in supine position and
<anesthesia>induced under general anesthesia.</anesthesia>
<prep>
<action>A Foley catheter was placed to decompress the
bladder</action> and the abdomen was then prepped and draped
in sterile fashion.
</prep>
...
The fascia was identified and
<action>#2 0 Maxon stay sutures were placed on each side of
the midline.
</action>
<incision>
The fascia was divided using
<instrument>electrocautery</instrument>
and the peritoneum was entered.
</incision>
<observation>The small bowel was identified.</observation>

,,,
```

Figure 2-1: Simple Medical Document --- Document-Centric XML [6]

In data-centric use cases, XML is used as a storage or interchange format for business data that is structured, appears in a regular order, and is likely to be machine-processed rather than being read by a human. Figure 2-2 is an example of a data-centric XML document.

```
<Customer id="1011">
....
<Addresses>
    <Address primary="Yes" type="Temporary">
    .....
    </Address>
    <Address primary="No" type="Temporary">
    .....
    </Address>
    <EmailAddresses>
        <Email primary="Yes">Marjo.Villoldo@evergreen.edu</Email>
        <Email primary="No">Fosca.Palomar@co.in</Email>
    </EmailAddresses>
</Addresses>
<Accounts>
    <Account id="104138966">
    .....
    </Account>
    <Account id="104138967">
    .....
    </Account>
</Accounts>
</Customer>
```

Figure 2-2: Customer Document --- Data-Centric XML [7]

2.2 XQuery and SQL/XML

XQuery [8] and SQL/XML [9] are two different standards to return XML results by querying data. XQuery is XML-centric, while SQL/XML is SQL-centric. Contemporary relational database vendors offer support for both standards.

XQuery is a language that uses XML as the basis for its data model and type system. It was developed in the XML Query Working Group [10], which is a part of the World Wide Web Consortium (W3C). XQuery is based on XML in the same way that SQL is based on the relational model and that object-oriented query languages are based upon the object-oriented model. Although there is no concept of relational data in XQuery, several products and many projects have provided ways to query relational data via an XML view of the database.

SQL/XML operates on the boundary between SQL and XML, while XQuery dwells within a pure XML realm. SQL/XML is designed for SQL programmers, and XQuery is designed for those who prefer a pure XML view of the world. For relational database vendors, SQL/XML support is usually more mature than XQuery support, as commercial database systems begin to support SQL/XML much earlier than XQuery.

The SQL/XML language has a list of basic XML construction functions that allow users to construct new XML elements or attributes using values drawn from relational tables. The SQL/XML standard also has functions to combine smaller XML fragments into larger ones.

The set of available construction XML functions includes: XMLELEMENT, XMLATTRIBUTES, XMLFOREST, XMLCONCAT, XMLNAMESPACES, XMLCOMMENT, XMLDOCUMENT, and XMLAGG.

In order to provide compatibility and interoperability between SQL/XML and XQuery, the following functions are also provided in SQL/XML today: XMLQUERY, XMLTABLE and XMLEXISTS. XMLQUERY queries XML-typed data and returns values of type XML as query results, while the XMLTABLE function accepts XML-typed data as input and generates a relational table as output. Predicates on XML data, such as search conditions, can be expressed with the XMLEXISTS predicate, which typically appears in the WHERE clause of a SQL statement.

2.3 EXRT and Its Objectives

The primary objective in the EXRT benchmark effort is to examine how well given commercial RDBMS can process native XML type information as compared to its established relational storage and query processing techniques. In a “mature” system, one would hope both to work equally well.

In this study, the focus is on basic XML query processing and update capabilities. A good XML query processing scheme should know how to effectively rewrite queries, use indexes

wisely, and do grouping and XML reconstructing efficiently. EXRT seeks to test these properties of a system as well as test basic XML update capabilities.

2.4 Value of the EXRT Benchmark

There are a number of existing benchmarks available to measure XQuery support and XML database processing performance, including XMach-1 [11], XMark [12] [13], XPathMark [14], XOO7 [15], XBench [16], MBench [17], and MemBeR [18] [19]. Some of these benchmarks are predominantly application-oriented, such as XMach-1 and XBench, while others were designed as abstract XML micro-benchmarks, e.g., MBench and MemBeR. Still others, like XMark, XPathMark and X007, can be viewed as a mixture of the two ---- although their data and queries represent an application scenario, they nevertheless also try to exercise “all” relevant aspects of the XQuery and/or XPath languages.

The TPoX [7] benchmark is an application domain benchmark that focuses on XML support and transactional processing ability [20]. TPoX tests the capabilities of database systems by doing multi-user read/write tests with high scalability using very large numbers of small XML documents. TPoX is also the first XML database benchmark to include complex updates based upon the XQuery Update Facility [21].

The benchmark EXRT, proposed in this study, is different from these previous XML

benchmarks in the following ways:

1. To the best of our knowledge, EXRT is the first XML benchmark to compare the performance of SQL/XML, XQuery, and relational support all in one benchmark.

2. Existing XML micro-benchmarks focus on exercising “all” features of the XPath and XQuery languages, while EXRT focuses on the basic capabilities relevant to enterprise (i.e. data-centric) XML applications.

3. Similar to the TPoX benchmark, EXRT attempts to emulate a real application scenario. Unlike TPoX, however, EXRT focuses on the performance of individual operations instead of focusing on multi-user transaction mixes.

4. Results from existing micro-benchmarks have been targeted predominantly at designers of XQuery query processing engines. EXRT aims to provide information that is useful for XML database schema and application designers.

An additional practical contribution of the EXRT benchmark is the extension of the TPoX code base. The TPoX code structure has been enriched to fit the needs of a micro-benchmark and has been modified to handle a more flexible parameter-binding scheme. In addition, code is provided for pre-generating trace files and parameter-value files to run the EXRT benchmark. We plan to release the EXRT code and benchmark statement-templates via the web after the results have been discussed with each vendor and a suitable way has been found to

release the benchmark without violating performance related clauses in their licenses agreements.

3. EXRT Benchmark Design

Most database benchmarks focus on one specific usage scenario, and the EXRT benchmark is no exception. To clearly identify which XML use-cases this EXRT is targeted for. The following options are three XML use-cases, each of which has its own characteristics:

1. Document-centric XML, in which typical query is focused on context-aware, full-text keyword searches. (e.g., see XQuery Full Text Support [22].) There is no known benchmark in this area, and designing such a benchmark would require an effort outside the scope of this study. Therefore, document-centric XML use-cases will not be considered in the EXRT benchmark.

2. Data-centric XML, which is what TPoX primarily targets. The typical database assumption is a large collection of moderately sized XML documents and XML indexes are primarily used to locate and filter these documents.

The EXRT benchmark just focuses on the second of these XML use-cases. The EXRT benchmark's major focus is to compare data-centric XML stored either as XML typed data or as shredded relational data to see the impact for both query and update operations while varying the XML data and query characteristics.

3.1 Benchmark Structure

A good benchmark should always select scalable datasets and load these datasets into suitable tables with well-designed table-schemas. In order to make reasonable comparisons, indexes must also be built upon these table columns. In addition, the output result must also be meaningful according to the application scenario that the benchmark is trying to emulate.

3.1.1 Datasets

In EXRT, it would be ideal to use an XML dataset that can represent an application scenario with meaningful operations on it. In order to compare the performance between XML and relational storage, it is also necessary to control the flexibility and the sizes of the XML elements in this dataset. In addition, the mapping between the schemas of the source XML documents and the relational storage repositories is required to be simplified for both XML and relational cases.

This benchmark will use synthetic data generated by the Toxgene data generator, which is a template-based generator for large, consistent collections of synthetic data-centric XML documents, developed as part of the ToX (the *Toronto XML Server*) project [23].

For EXRT, the "CustAcc" document for TPoX was selected as the dataset, which is one type of document generated by the Toxgene data generator. This was done because, even though

it is a synthetic dataset, it has been modeled after the customer and accounts information aspects of real banking systems. Structurally, these CustAcc documents have several properties that are ideal for the purposes of this study, such as unique constraints on customer id and account id, a unique primary address for each customer, and nested and repeating data with at least one address and one account for each customer. For each customer, there is a CustAcc document which contains all customer information, account information, and holding information for that particular customer. Figure 3-1 shows the XML Schemas that define the XML dataset with well-defined value distributions and referential integrity across documents. In the situation under study, the CustAcc document instances range in size from 4KB to 20KB.

As indicated in Figure 3-1, each CustAcc document contains one customer's detailed profile and account information. Every customer has a unique customer ID number (4 digit number or more), a "Name" element, which contains one "FirstName" element, one "LastName" element, and zero to three "MiddleName" elements. Every customer also has an "Addresses" node which contains zero to three "Address" elements. Each "Address" element contains one to five different "Phone" elements. Besides this profile information, every customer also has an "Accounts" node which contains one to seven different "Account" elements. Each "Account" element has a Unique ID number (10 digits) plus all of the information about the account itself.

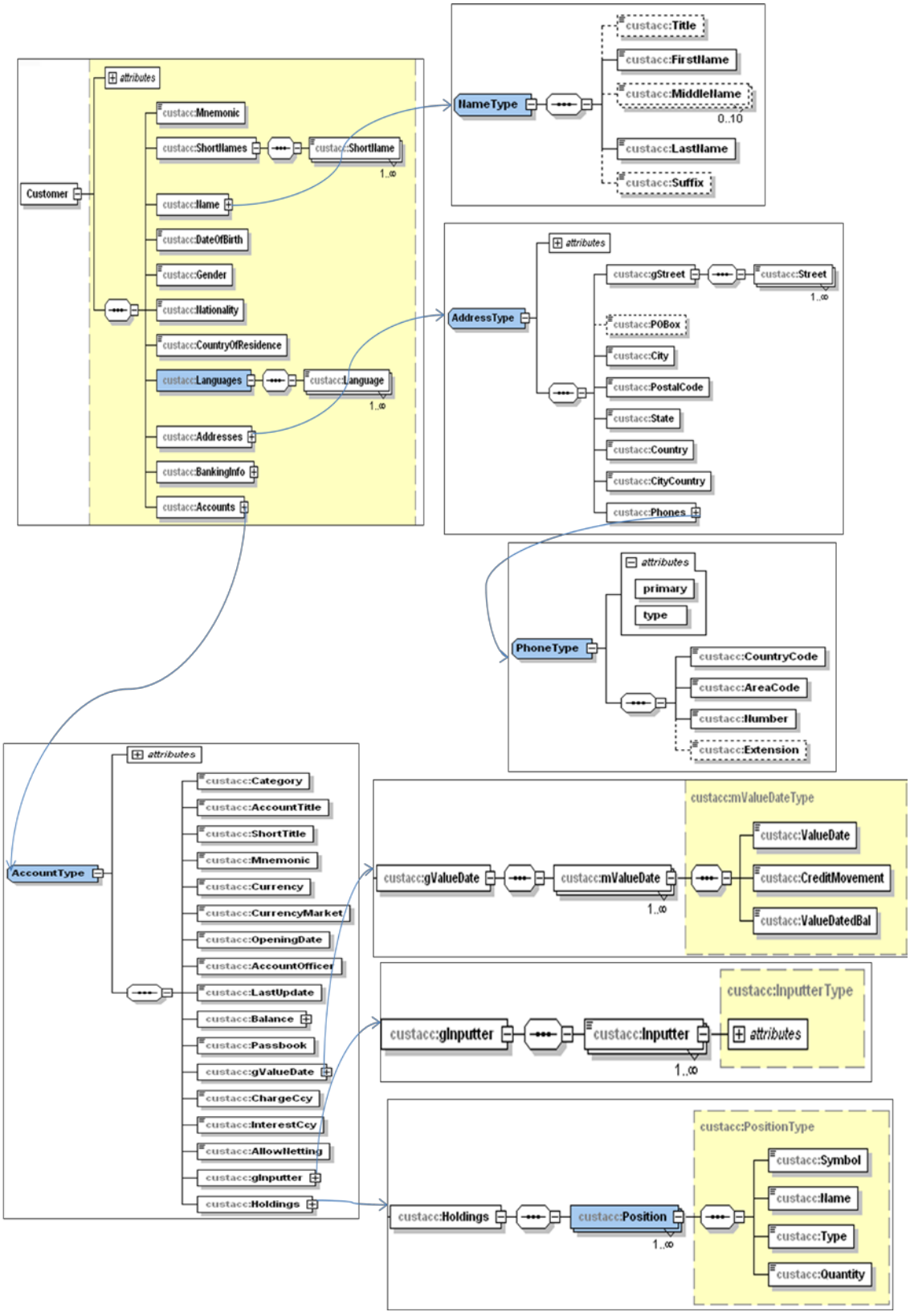


Figure 3-1 Document CustAcc XML Schema

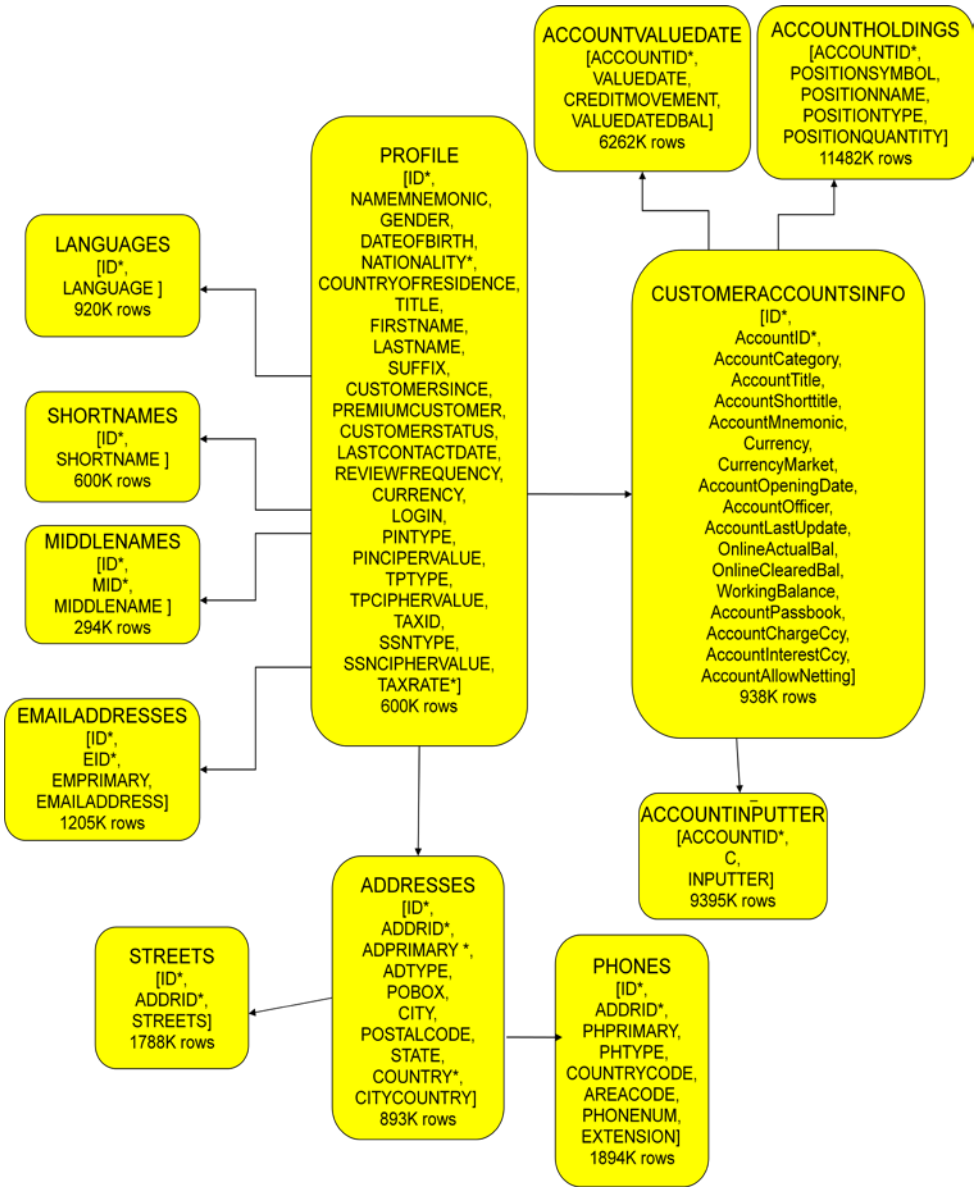
3.1.2 Table Schema

For this study, one option is that XML data is loaded into XML type storage, which could be a table with an XML column where the data instances can live. The other option is to "shred" the XML data into normalized relational tables. In order to normalize the relational tables, if the element in XML appears more than one time, then this element must be put into a separate table. Thus, in all, the relational schema needs a total of twelve tables. Figure 3-2 shows the details of the twelve tables, including their names, columns, indexes, and rows counts. The PROFILE table contains the basic customer profile information. Each row in PROFILE table has a unique primary key -- customer ID. Similarly, the CUSTOMERACCOUNTS table contains the basic account information, and this table has a unique primary key – AccountID.

Based on the XML data-shredding mechanism, the tables LANGUAGES, SHORTNAMES, MIDDLENAMES, EMAILADDRESSES, CUSTOMERACCOUNTS and ADDRESSES all have a foreign key relationship with the PROFILE table. The customer ID column in the PROFILE table is referenced by the customer ID columns in these tables. Likewise, the tables STREETS and PHONES also have foreign key relationship with the ADDRESSES table. Finally, the tables ACCOUNTVALUEDATE, ACCOUNTHOLDINGS and ACCOUNTINPUTER reference the AccountID column in CUSTOMERACCOUNTS table. The

SQL definitions for these relational tables as well as the one for the XML table are included in the appendix.

One important detail regarding the XML data shredding approach needs to be mentioned here. Some of the resulting relational tables did not have any actual primary key constraints on them, so the Update part of the benchmark operation on these tables will become difficult to implement. It was thus necessary to construct primary keys for these tables involved. (Otherwise it would not be possible to identify a single piece of information in these tables.) In an XML document, there is inherent position of information to denote the 1^{st}, 2^{nd} or 3^{rd} child node under a given parent node. Thus, in our relational tables, a new sequence number column has been introduced for aggregated elements under the same parent node. For example, when the Addresses element is shredded into relational tables, the current position number of an Address node under its parent Addresses node will be inserted as an integer value into the ADDRID columns in corresponding address-related tables. Using ADDRID and customer ID as a composite primary key in the ADDRESSES table makes Update operations on this table much easier. Similar things are done to the EMAILADDRESSES table (using EID as the position number for each EMAILADDRESS element) and the MIDDLENAMES table (using MID as the position number for each MIDDLENAME element).



Note: * = indexed column

Figure 3-2 Shredded Relational Table Schemas and Cardinalities

3.1.3 XML and Relational Indexes

In order to get the best performance for each benchmark operation, the EXRT database design must include indexes for each predicate that is used to query an XML table or the

equivalent relational tables. These would depend on the specific operations that the benchmark queries, insert, delete and update statements try to perform. In order to get a meaningful comparable set of performance results for XML versus relational storage, two rules must be followed for index creation:

1. The “same” indexes must be created for both the XML table and the relational tables. (Here, the “same” means that the XML indexes must benefit the XML table in the same way that the relational indexes benefit the relational tables.)

2. Similarly, every database system being tested must have the same indexes for all of the tables.

In Figure 3-2, columns with a * mean that these columns are indexed in order to support EXRT queries and/or updates.

3.1.4 Result Format

Since the intent of EXRT is to compare the performance of shredded data management with that of XML-typed or native XML data management, the same requests and the same returned results should be obtained from both approaches. For this reason, each EXRT query operation reconstructs a new XML document as the benchmark query result format.

For the shredded case, which is all relational data in terms of storage, it is necessary for the SQL/XML queries to construct a new XML document with the help of these SQL/XML publishing functions: XMLAGG(), XMLELEMENT(), and XMLFOREST(). For the case of XML storage and XQuery queries, it is necessary (in most cases) to wrap up the existing XML elements with new XML tags when doing reconstruction to create the XML result for each query.

3.2 Benchmark Metrics

The benchmark for this project includes a range of query and update statements that are designed to model common XML data operations that occur when interacting with XML business data in typical web and SOA applications.

In the EXRT benchmark, two query-related metrics – “width” and “tallness” are used to vary the characteristics of the queries. In the relational case, the “width” of a query represents the number of relational tables accessed in the query. In the XML case, the “width” means the number of different types of aggregated XML elements that are used in the query. The “tallness” measure is used to represent how many reconstructed XML documents are returned by the query or modified by an update. (I.e., the “tallness” is set to be the number of returned results in one query.) Figure 3-3 illustrates the meaning of “width” and “tallness”. Each triangle in this figure represents an XML document that has been divided into four layers. Queries of “width 1” only

fetch the XML content of the top layer. Queries of “width 2” touch the top layer and second layer, and so on. “Tallness 1” means only one XML document is returned or modified.

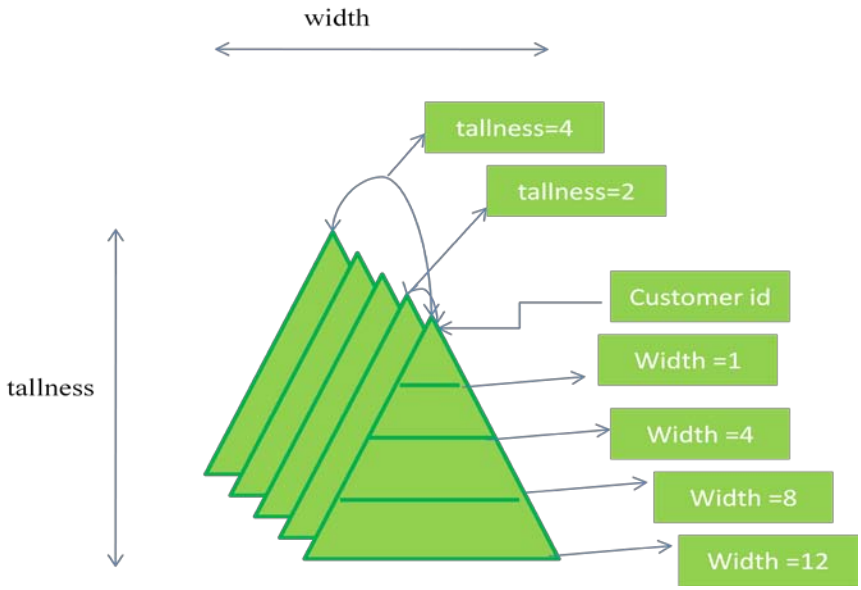


Figure 3-3 Two Benchmark Metrics: “Width” and “Tallness”

3.3 Benchmark Queries and Updates

EXRT consists of a set of query and update operations which cover a range of behaviors that are expected in typical data XML use cases. For the case of native XML storage, the benchmark tests both XQuery and SQL/XML versions of each query (if supported by the system under test) in order to explore their relative performance. For shredded XML storage, only SQL/XML queries are performed over the relational tables. Thus, three different types of queries are compared: (a) XQuery on native XML data, (b) SQL/XML on native XML data, and (c) SQL/XML on shredded relational data. Inserts, deletes and several atomic element updates are

also part of the test suite. These update operations only have two versions to be compared: (a) SQL/XML on XML data, and (b) SQL/XML on relational data here. This is because update operations cannot directly use XQuery to update data in XML columns of a relational database. They must instead use SQL/XML functions (possibly with embedded XQuery Update facility expressions) to modify data in XML columns.

3.3.1 XML Queries

The first four queries in the EXRT benchmark test how width differences affect query performance for both XML part and relational part. These queries will return different parts of a document or the whole document as results; this is controlled by the "width" dimension. Then, for each "width," the query's range predicates will determine the number of elements or rows that will be returned in the result set; this controls the "tallness" dimension. For Query 1-7, these seven EXRT queries all have the following characteristics: they contain either numbered parameter markers denoted by |1, |2, etc. or standard SQL parameter markers denoted by "?" (or ":1"). Based on the benchmark driver description file, the driver replaces these parameter markers with actual literal values. For the XQuery version of each query, a built-in XMLCOLUMN function is used to retrieve a sequence from a column in the currently connected relational database. Then XQuery query uses FLWOR expression to extract the information and construct a

new XML type result. For the SQL/XML version of each query, the SQL/XML predicate XMLEXISTS is used to select XML documents based on one or multiple conditions which are expressed in XQuery notation. The SQL/XML function XMLQUERY is then used to retrieve either full or partial XML documents or to construct new projected result documents that are different from the ones stored in the database. For the relational version of each query, XMLELEMENT and other construction functions are used to construct XML Elements and nested XML elements from relational data. Both XQuery queries and SQL/XML queries take advantage of one or multiple XML indexes to avoid table scans, while relational queries uses relational indexes created on those relational tables. Figure 3-4, Figure 3-5 and Figure 3-6 show the three different versions of Query 1—XQuery on XML storage, SQL/XML on XML storage, and SQL/XML on Relational storage (i.e., SQL/XML on the shredded tables).

XQuery query on native XML storage

```
XQUERY
declare default element namespace "http://tpox-benchmark.com/custacc";
for $cust in 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 } }
```

Note: similar as TPoX, |1, |2 and |3 are the numbered parameter markers which will be replaced by actual literal values supplied from different input sets.

Figure 3-4 XQuery query on native XML for Query 1

SQL/XML query on native XML storage

```
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")
```

Figure 3-5 SQL/XML query on native XML for Query 1

SQL/XML query on Relational storage

```
select
xmldocument(
xmlelement(name "Customer",
   xmlnamespaces( DEFAULT 'http://our-benchmark.com/custacc'),
   xmlattributes(ID as "id"),
   xmlelement(name "Name",
      xmlelement(name "Title", title),
      xmlelement(name "FirstName", firstname),
      xmlelement(name "LastName", lastname),
      xmlelement(name "Suffix", suffix)
   ))) from profile
where profile.id>= cast(? as int) and profile.id < cast(? as int) + cast(? as int)
```

Figure 3-6 SQL/XML query on shredded tables for Query 1

We plan to put the whole set of benchmark queries and update statements on our benchmark website (or provided then by user request) after we discuss our plans with each vendor and figure out a way to do so without violating any software license agreements.

Table 3-1 summarizes the Query 1, 2, 3 and 4 in EXRT. Queries 1 to 4 take customer ID range values and extract varying amounts of information about each customer. These queries test the cost of basic exact-match and range-based ID lookups, as well as testing the cost of retrieving and assembling the requested information. For native XML storage, the query-cost involves selection, XPath navigation, element extraction, and the construction of new result documents (i.e., XML reconstruction). For shredded storage, the cost involves selection, joins, and result construction. The customer information fetched for each customer is varied from basic

information (accessing only one Profile table in the shredded case) up to all of the information (accessing all of the tables in the shredded case).

Table 3-1 Benchmark Operations: Queries 1, 2, 3 & 4 [5]

Op	Description	Width
Q1	For given customer IDs, fetch the customers' minimal profile – consisting of their customer ID, title, first and last names, and suffix.	1
Q2	For given customer IDs, fetch the customers' basic profile – adding middle and short names and languages to the minimal profile.	4
Q3	For given customer IDs, fetch the customers' complete profile – adding e-mail information, addresses, streets, and phones to the basic profile information.	8
Q4	For given customer IDs, fetch all of the customers' information, including all of the information about their accounts and holdings.	12

Included here are two versions of Q4 for XML data. One version is without any node construction, as it just returns a whole customer document. The other version is with node construction, so it reconstructs a new customer document by extracting all nodes and values. The purpose of including these two versions of queries is to determine what the overhead is for XML node construction.

In addition to the XML equivalents of select/project queries, another interesting question is what performance will be like if queries have predicates on nested subdocuments (e.g., on Account id) and return either a whole subdocument or a whole document instead of a partial nested document. In Table 3-2, Queries 5 to 7 are designed to test the performance of navigation,

path indexing, and document fragment extraction for complex XML objects. The widths of these queries are fixed by their information content, but their selectivity varies from 1 to 600.

Table 3-2 Benchmark Operations: Queries 5, 6 &7.

Op	Description	Width
Q5	For given customer IDs, get the complete information for all of their accounts.	5
Q6	For a given account ID, get the complete account information.	4
Q7	Given an account ID, get all of the customer information for the account's owner.	12

Comparing these three queries, it can be seen that Query 5 does distinct integer-value based selection of a parent object and returns its nested child objects of a certain kind. In contrast, Query 6 performs distinct string-value based selection of a nested object and then returns the selected nested object itself. Query 7 also performs distinct string-value based selection of a nested object, but it then returns its parent object.

There are some useful features in the traditional SQL language that have ongoing usefulness in XML use cases, such as SQL's aggregation functions. In Table 3-3, Query 8 and Query 9 are simple summary queries with multiple predicates that test the data aggregation capabilities of a system. Query 8 calculates the average number of accounts of the customers with a given nationality. Query 9 calculates the average balance of accounts of customers whose primary address is located in a given country with a given tax-rate.

Table 3-3 Benchmark Operations: Queries 8 & 9.

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 that country who have a tax rate greater than the specified rate.	3

Overall, for these nine different queries, Queries 1-7 have three different versions: (a) XQuery on XML storage, (b) SQL/XML on XML storage, and (c) SQL/XML on Relational storage. Query 8 and Query 9 have two versions: (a) XQuery on XML storage, and (b) SQL/XML on Relational storage.

3.3.2 Updates

The benchmark used in this study also tests the performance of basic insert, update, and delete operations. These operations include full document insert and delete operations (Operation I and D) as well as node-level changes such as inserting (Operation NI1, NI2, NI3), deleting (Operation ND1, ND2, ND3) or updating (NU1, NU2, NU3) individual pieces of an XML document. Different from the queries, the updates vary just one dimension—"width" as we were not interested in bulk updates for EXRT. Structurally, the customer information modified in those update statements with larger "width" also contains (i.e., is a superset of) the information modified by statements with lower "width." These operations are applied to a randomly selected customer, and they change either the top level customer information or information nested within the customer, such as an address and account. Table 3-4 summarizes the updates tested in EXRT.

Table 3-4 Benchmark Operations: Inserts, Deletes and Updates [5]

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 information about this customer and his accounts.	12
NI1	*Node Insert 1:* Given a customer ID and an XML string with a new Address element, add the new address to the specified customer.	3
NI2	*Node Insert 2:* 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	*Node Insert 3:* Given a customer ID and XML strings containing a new Address element, a new e-mail address, and a new account, add these to the customer.	8
ND1	*Node Delete 1:* 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	*Node Delete 2:* Given a customer ID plus positional indicators for their address and e-mail lists, delete the indicated Address and Email nodes.	4
ND2	*Node Delete 3:* Given a customer ID, an account ID, and positional indicators for their address and e-mail lists, delete the indicated Account, Address, and Email.	8
NU1	*Node Update 1:* For a customer ID, update the customer's last contact date.	1
NU2	*Node Update 2:* 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 status (premium = 'yes'), and update the assigned account officer's name.	2
NU3	*Node Update 3:* 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

4. Experimental Setup

The EXRT benchmark experiments have been run on the latest versions of two commercial relational database systems. These two systems are designated as RDB1 and RDB2. This was done for purposes of anonymity related to the licensing agreements for the systems. Both systems (RDB1 and RDB2) provide both native XML storage and XML query language support. In this chapter, the EXRT benchmark experimental setup and the EXRT test procedures are discussed.

4.1 Equipment Setup and Configuration

Before the other aspects of the study are described, the equipment setup and special configuration need to be clearly specified. Each database system had its own parameter settings that needed to be tuned before the tests could be conducted. The running conditions for the two systems for the benchmark tests are also introduced in this chapter.

4.1.1 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 is Red Hat Enterprise Linux Client release 5.4 (Tikanga),

kernel 2.6.18-164.el5. The database installation files were stored on a single disk. A single volume group (VG) was created that spanned the system's two disks in order to utilize both disks. This striped volume [24] has a size of 100GB per each disk. All data files, index files and logs were stored on that volume.

4.1.2 System Configuration

For both systems, a number of suggestions from their respective vendors were taken into consideration and practice. For example, in both RDB1 and RDB2, XML compression techniques are used to improve performance and reduce the storage size of the 10GB raw XML data (600,000 XML documents). Both database systems were configured to run without any file system caching and with automatic storage management.

Automatic buffer management mechanisms were used so that both systems can make full use of the system memory. (For RDB1, the maximum amount of RAM memory used by a database instance can be accurately controlled, and the database system will automatically expand or shrink the memory areas as needed. For RDB2, one can set the default buffer pool to a suitable size and then ask the system to automatically adjust the size of all buffer pools.)

It is also important, when tuning, to specify the database page sizes. Based on each vendor's suggestions, 8k database page size was used for RDB1. Similarly, as per advice from the

other vendor, a 32k page size was used for RDB2. At the time the study was conducted, the software tested for RDB1 and RDB2 were the latest versions available. (However, RDB1 also provided us with some new patches to improve the system's performance by circumventing certain optimizer issues that EXRT turned up).

4.1.3 System Running Conditions

In order to provide performance information that is useful to developers of XML-based applications, the concept of "cold" and "hot" is introduced, which is similar to what was done in the previous benchmark 007 [25]. In real-world applications, where data volumes are much larger than main memory, systems are neither perfectly "cold" nor entirely "hot". Clarifying these two running-options can reveal the upper-bound and lower-bound of time-costs for each operation, which represent the performance in the best-case and worst-case scenario, respectively.

"Cold" operation means that the data being accessed by queries would not be resident in memory. Each "cold" query would thus have to retrieve all data from the disk instead of accessing data from the data cache or some other buffer cache. (However, the system's other non-data caches, such as caches for query plans or catalog information, will remain in memory to be reused). "Hot" operation means there is always a hit in memory when accessing the requested

data. In the EXRT benchmark tests, the measured data access times, as well as the max and min values for these time costs, are recorded for accurate comparison.

In the EXRT experiments, “cold” tests were performed by clearing the buffer pool between each query execution. Different query parameters were randomly selected for each cold query execution. Therefore, the query results were different in each instance, and the data pages retrieved were different in each case. “Hot” tests were performed by running the same query repeatedly with the exact same parameter values, without clearing the buffer pool.

For RDB1, only the data cache was cleaned between each “cold run”. The catalog data cache and the query-plan cache remained in memory. Because data cache, plan cache and catalog cache are all loaded into memory that is shared among all database connections, simply disconnecting and reconnecting a session does not cause RDB1’s data cache, plan cache or catalog cache to be flushed. Thus, based on the suggestions from the RDB1 vendor, a specific database system command was invoked to explicitly clean the data cache. However, for the “hot” runs, the data cache and other cache information was not cleaned after each operation finished. All “hot run” queries were bound with the same parameter as values in their predicates.

In RDB2, it turns out that simply closing all connections between each “cold run” cleans not only the buffer cache, but also the catalog cache and the query plan cache. This could have

posed a problem, making RDB2 “too cold” when being tested for its “cold” performance. However, all of the statements (except the XQuery queries) in a given transaction properties file are prepared in a pre-execution statement preparation stage, this is not timed by EXRT. Catalog data is brought into memory during this stage before execution, and the query plans are also re-computed before execution during this stage. As a result, cleaning the buffer pool by closing all connections for RDB2 does not put RDB2 into an unfair condition relative to RDB1.

4.2 Experimental Procedure

The most difficult part of the EXRT benchmark project was to make sure a fair comparison was being made between the two systems that were tested. Several constraints were placed upon the experiments to ensure this.

First, each database system needed to be permitted to make full use of the overall system resources, disk, memory and other buffer caches. Suitable indexes needed to be built on both XML and relational tables for both systems, and it was necessary to verify that proper indexes were indeed being needed to be used in the query execution plans.

Second, it was necessary to ensure that the performance numbers from each system were not too “cold” as compared with each others. For example, the first “cold run” number after a system restarts is considered too “cold” and should therefore be removed when the average “cold

run” number is calculated. This is because the first “cold run” needs to bring the database catalog information into memory and also compute a new query plan for this query. Ideally, in contrast, for a “hot run”, each query should find the data blocks needed in memory after the first one or two “hot run” executions.

Third, for query plan reuse purposes, each vendor suggested that, when preparing statements, a parameter binding method should be used instead of using the constant value string-replacement method. (String replacement means that the query statement in each “cold run” is different from the statements in the previous “cold runs”, as it has search parameters in it as literals, and therefore the query plan would be different and cannot be reused.)

According to the above constraints, these following decisions were made regarding the EXRT operating procedures:

(a) For RDB1, the session was not disconnected or reconnected, and only the data cache would be cleaned explicitly between each “cold run”. For RDB2, the JDBC connection was singly closed between each “cold run” to clean the data cache.

(b) For both RDB1 and RDB2, all SQL/XML queries and updates were written in standard-compliant SQL/XML notation, taking advantage of parameter markers. However, since RDB2’s pure XQuery API does not support parameter binding, string-replacement methods have

to be used to construct new XQuery statements. (It would have been possible to “warp” RDB2’s queries in SQL/XML for passing parameters, which is what RDB1 required, but it was decided that RDB2’s natural XQuery API should be tested.)

(c) The vendor for RDB1 suggested that some queries should be rewritten to allow the optimizer to produce a (much) more efficient access plan. As a result, the SQL/XML query versions for RDB1 are slightly different from the SQL/XML versions of RDB2 with regarding to the range predicate expression and XPath expression. Thus, RDB1 could have different query performance depending on the query format. In chapter 5, some results are given for the different formats.

It was not the goal for EXRT to demonstrate that one system was significantly better than the other, and the results show that neither system was better than the other in every operation. It was interesting enough to demonstrate both the intra-and inter-system performance trends and tradeoffs under a fair comparison.

4.2.1 Test Execution Methodology

The EXRT benchmark project driver code was running on the same machine which also housed the server. The benchmark driver code was implemented in Java and was used to drive the benchmark queries based on a set of the transaction templates. A transaction template is a query,

insert, update or delete statement in which parameter-markers can be used instead of literal predicate values. Each EXRT transaction must be provided in a separate text file, and a transaction can consist of one or multiple statements (query, insert, update or delete). In this study, only the delete transactions for relational data needed to have multiple SQL statements in one transaction. (This was because delete statements of each transaction were not combined in one stored procedure.) The benchmark driver issues a commit after each transaction finishes, i.e. after the last statement of each transaction was executed (unless a different commit count is specified in the –cc option as the input argument of the driver).

Below is an example of the transaction template for insertion into an XML table:

```
INSERT INTO CUSTACC(column_name) VALUES (?)
```

Another example of the transaction template, for deletion in an XML table, is shown below:

```
DELETE FROM CUSTACC WHERE XMLEXISTS('declare namespace ns= "http://tpox-benchmark.com/custacc";

$cadoc/ns:Customer[@id=$id]' PASSING column name AS "cadoc", cast (? as integer) as "id")
```

The "?" is the input parameter marker which is bound using values read from the trace file specified in the workload description files as input to the driver [26]. The trace file has a list of random customer ID and account ID information, and it is generated before benchmark

execution begins. At run time, the “?” in the transaction template is replaced at binding time by actual values drawn from configurable random value distributions or lists.

4.2.2 Measuring Procedure

The EXRT benchmark measures the query-submission time plus the time to retrieve all results as the execution time cost. This models the response time that a client application is interested in. JDBC query preparation times were not measured. Figure 4-1 show the pseudo code for the EXRT measuring procedure.

```
/*****************************************************************/
FOR each query in list
        Get connection;
    Prepare all the queries in the list.
    IF asked to run query in cold system
        FOR each ith cold run, i < NumOfColdRun    // As an input argument of driver
code, NumOfColdRun in our tests is set to10
            Get start_time;
            Run this query with random parameter;
            Read each result in result set to a local variable;
            Get end_time;
            IF this is not the first 1 cold run
                Add (end_time – start_time) to ColdRunTimeSum;
            ENDIF
            IF this is RDB2
                Close connection;
            IF this is RDB1
                Flush buffer pool to reach cold state through admin connection;
        ENDFOR
    ENDIF
    IF asked to run query in hot system
        FOR each ith hot run, i < NumOfHotRun
            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 this is not the first 1 hot run
                Add (end_time – start_time) to HotRunTimeSum;
            ENDIF
        ENDFOR
    ENDIF
    Close connection
      Average_ColdRunTimeForThisQuery = ( ColdRunTimeSum – MinColdTime –
MaxColdTime) / (NumOfColdRun-3);        // the first execution and of the slowest and
fastest of the remaining executions are discarded. In the tests, NumOfColdRun-3 = 7
        Average_HotRunTimeForThisQuery = (HotRunTimeSun – MinHotTime -
MaxHotTime) / ( NumOfHotRun-3);
        // the first execution and of the slowest and fastest of the remaining executions
are discarded. In the tests, NumOfHotRun-3 = 7
ENDFOR
/*****************************************************************************/
```

Figure 4-1: Pseudo Code for the EXRT Measuring Procedure

For systems that support the separation of query preparation and query execution, the measured response times include only the execution time plus the time to retrieve all query results from the database server. To obtain stable results, in the “cold” condition, each query or update was performed ten times (NumOfColdRun = 10). The measured elapsed times of the first execution and of the slowest and fastest of the remaining nine executions are discarded. The remaining seven (i.e., NumOfColdRun-3 = 7) measurements were then averaged. In the “hot” condition, each query was again performed 10 times, all with identical query parameters. Similar to the “cold run”, the same three outliers were ignored, and only the remaining seven “hot run” (i.e., NumOfHotRun-3 = 7) measurements were averaged.

5. EXRT Results and Analysis

This chapter presents all of the results collected from the EXRT benchmark tests on both systems (RDB1 and RDB2). Some preliminary performance analysis will also be given.

5.1 Document Selection Queries with XML Construction

In what follows, all results will be arranged in the same format – with two charts in a row to represent RDB 1 and RDB 2. The meaning of the legends in the charts is: “XQuery on XML” = XQuery over an XML column/collection; “SQL/XML on XML” = SQL/XML over an XML column; “SQL/XML on Rel” = SQL with XML construction functions over normalized relational tables that contain shredded XML. (In the text of thesis, “XQuery” denotes “XQuery on XML”, “SQL/XML” denotes “SQL/XML on XML”, and “Relational” represents “SQL/XML on Rel”.)

In addition to Queries Q1 - Q4, Query "Q4 (re)" reads a full document by reading all of its elements and then reconstructing it, while "Q4" reads a full document in one piece, which is the preferred and more efficient operation for full document retrieval. Q4 (re) is included to aid in the analysis of Q1 – Q3, where results construction is non-optional.

Figure 5-1 shows the measured average response times of the first four queries (Q1-4) for a selectivity of 1 row for both systems. As expected, the performance under “hot” conditions was

significantly better than under “cold” conditions, for all queries and both database systems. Furthermore, constructing XML data became increasingly more expensive as the size and width of the constructed XML becomes larger in the “hot” case.

In the “cold” case, the XQuery and SQL/XML results were not as much affected by width differences as the Relational query results when moving from Q1 to Q4 (re), which is different from the “hot” situation. This means that the I/O cost for index-lookups and page-fetching dominates the CPU cost for XPath navigation and element reconstruction in XQuery and SQL/XML when the selectivity is small. This was true for both systems (RDB1 and RDB2).

Under both “cold” and “hot” conditions, the cost of the Relational queries, which construct XML from multiple relational tables, became significantly more expensive as the width increased. This was because the number of physical I/Os to access distinct index and data pages increased with the width (i.e. with the number of tables accessed) in the “cold” case. Retrieving a document as a single unit (Q4) is clearly much faster than reconstructing the full document (Q4 (re)). Note that, for the Relational case, there is only Q4 (re) but no Q4. This was because the Relational query must always perform reconstruction. By comparing the SQL/XML (or XQuery) numbers in Q4 with Relational numbers in Q4 (re), it becomes apparent that the queries over

native XML storage can significantly outperform queries over shredded XML storage in both systems when full document results are desired.

Under “hot” conditions, the tests also showed that accessing shredded relational data can outperform native XML access in both relational systems. This is because under “hot” conditions, the physical I/Os to the data pages of the 12 tables are not needed, and the CPU costs are apparently higher for XML data than for relational data.

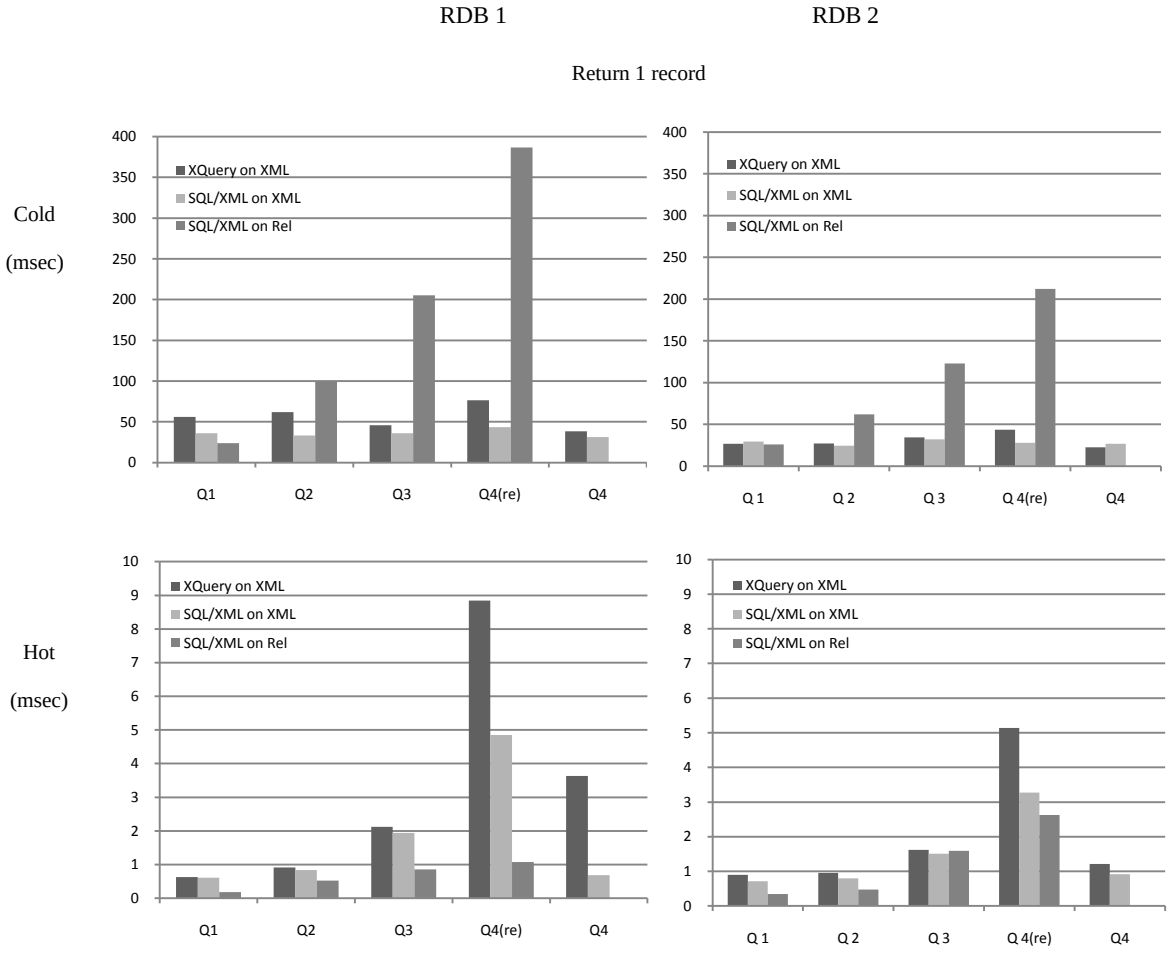


Figure 5-1 Query Response Times in Milliseconds Q1-4 (Result set size = 1).

Finally, let us compare XQuery and SQL/XML for the two systems. Ideally, the query plans for both cases would be the same, XQuery and SQL/XML should have had the same or similar performance. (This was in fact the case for RDB2.) However, for RDB1, in the earlier tests, XQuery performed considerably worse than SQL/XML over XML. I discussed these "strange" results with the vendor for RDB1, and they offered several explanations and suggestions for improvements. It was rendered that both the SQL/XML query and the pure XQuery over XML data used XML index. However, some optimization techniques used in SQL/XML version were not applied to the XQuery plan over non-schema-based XML data. RDB1 vendor therefore suggested rewriting the all the XQuery statements for RDB1. Suggestions from the RDB1 vendor included saying that the predicates of form [. > X and . < Y] should be used to express the "between" predicates in SQL/XML or XQuery, as this allowed their optimizer to fully optimize the queries. In the end, the vendor for RDB1 supplied a patch release that corrected these issues that the EXRT benchmark queries exposed.

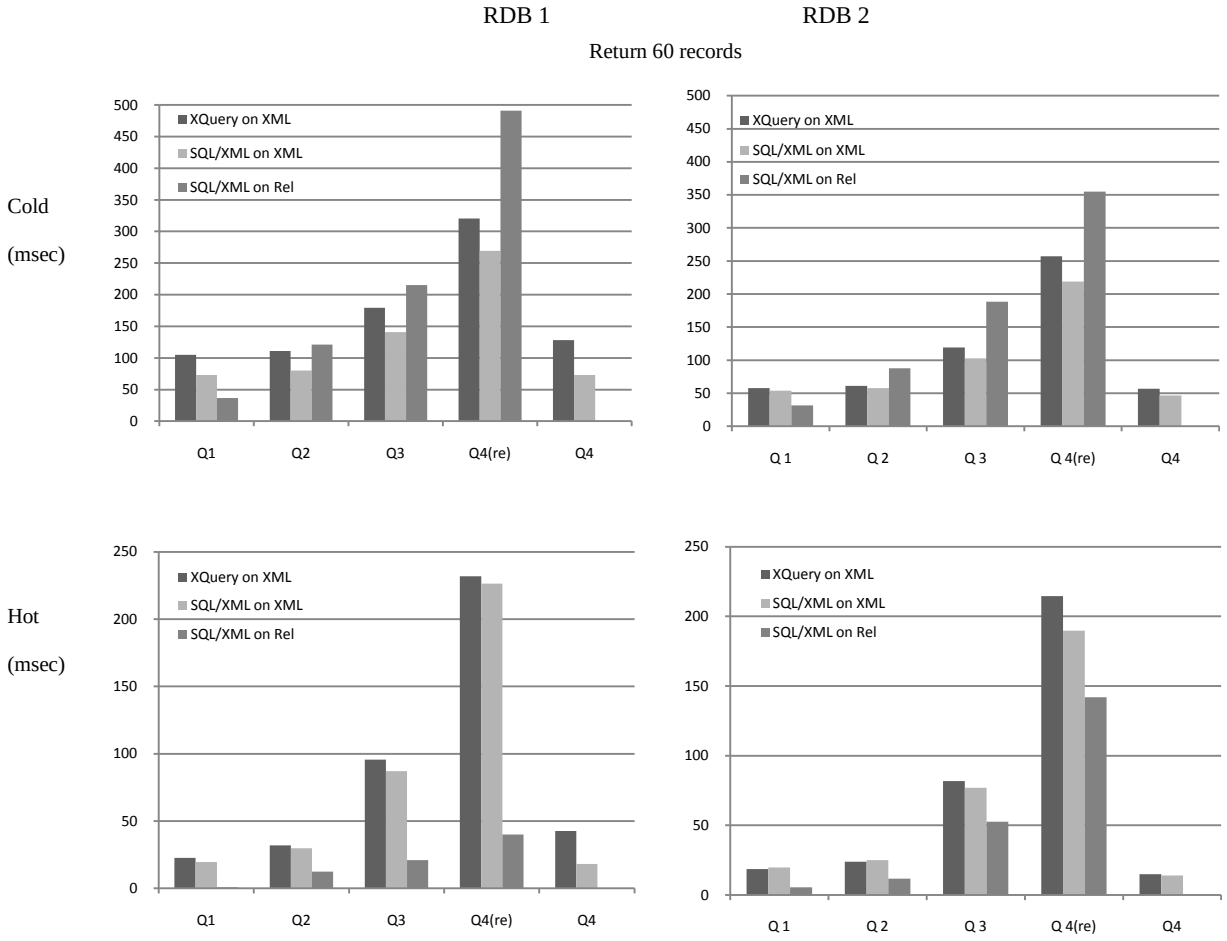


Figure 5-2 Query Response Times in Milliseconds Q1-4 (Result set size = 60).

Figure 5-2 shows the results for the same queries when the query predicates select 60 result documents instead of just a single document. For this larger selectivity, the elapsed time increases as expected. The effect of width on query performance also increases; the width effect became clearer than in the previous "cold run" figure for the SQL/XML query. This occurred because the cost of XPath navigation and element reconstruction increases significantly from a single row to 60 rows. Nevertheless, under "cold" operating conditions, the Relational query performance is still worse than the SQL/XML query performance. However, under "hot"

conditions, the Relational query performance is better than SQL/XML query performance. Retrieving full XML documents without constructing them (Q4) once again displayed better performance than Q4 (re) in both “cold” and “hot” conditions. Because there was no reconstruction in Q4, the performance of Q4 was in fact quite similar to that of Q1, which has the least construction effort in this case.

With the further increasing size of the result set (e.g. 600 rows in Figure 5-3), the difference between “cold” and “hot” performance diminishes. This is because two factors begin to outweigh the cost of physical I/O. One factor is the large CPU cost needed to construct XML for many result rows. The second is the growing overhead of fetching the results from the database server to the client. Unlike the previous cases, SQL/XML and XQuery (Q1, Q2, Q3, Q4 (re)) which construct XML from selected XML data were slower than the Relational query under both “cold” and “hot” conditions.

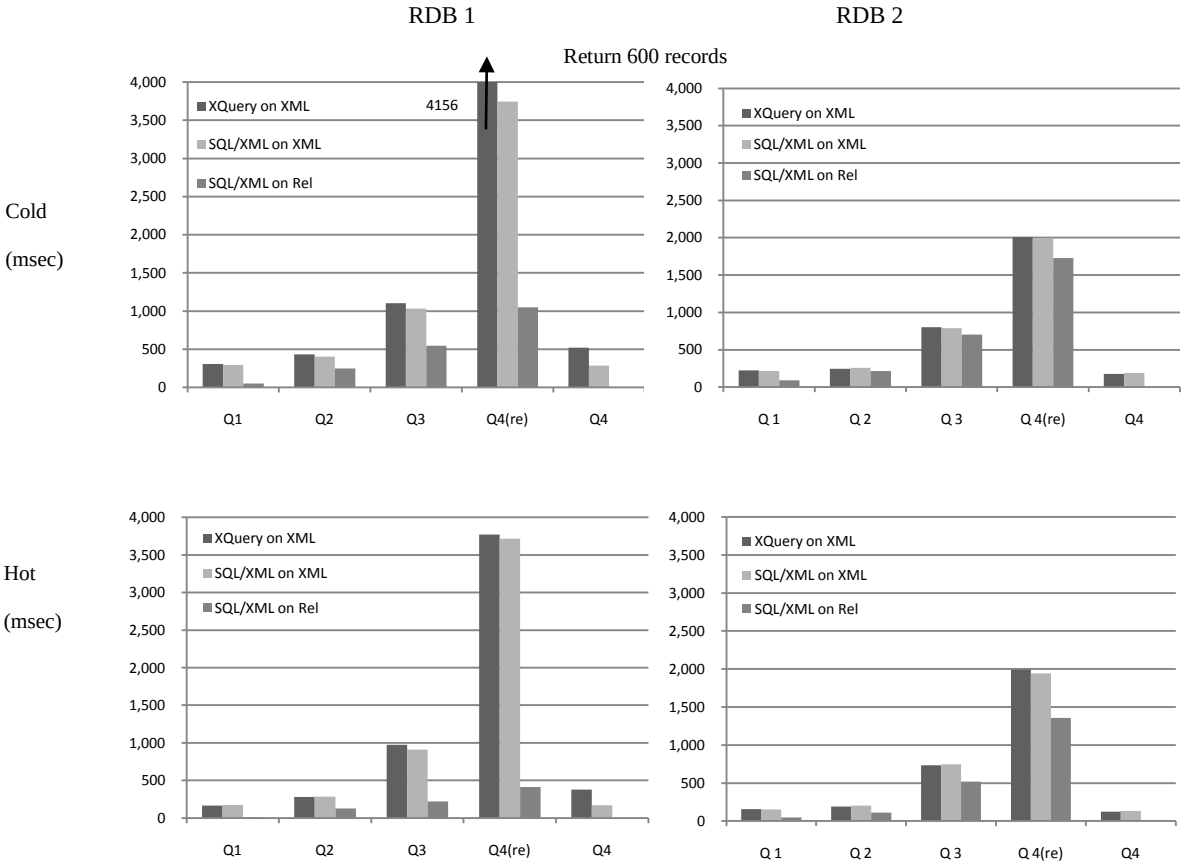


Figure 5-3 Query Response Times in Milliseconds Q1-4 (Result set size = 600).

5.2 Queries that Extract Partial Documents

For Queries 5, 6 and 7, the XQuery and SQL/XML over XML queries return sub-trees (fragments) or full customer documents from stored documents without constructing new XML elements. However, the queries over relational tables still require XML construction, of course. Query 5 extracts "account" fragments from the customer documents that match specific customer IDs. Similarly, Query 6 extracts "account" fragments that themselves match specific account IDs. Query 7 returns the full customer documents that include an account with a specific account ID.

Figure 5-4 shows the response times of these queries Q5 - Q7 when their predicates match only one document. In RDB1, only SQL/XML over XML can outperform the relational for all queries. The performance of XQuery in was still worse than the other two versions for Query Q6. (An XPath-rewritten for this XQuery statement in RDB1 might help improve the XQuery performance. But doing this manual XPath-rewritten might arguably be unfair to RDB2.) In RDB2, both XQuery and SQL/XML over XML columns clearly performed better than the Relational queries where reconstruction of XML objects was required. Reconstructing large XML fragments from relational tables was still expensive in both systems, especially for Q7, which constructs a whole customer document. In the “hot” case, as expected, the performance difference between XQuery and SQL/XML over XML was negligible for RDB2. This was because RDB2’s XQuery and SQL/XML queries were compiled into identical execution plans.

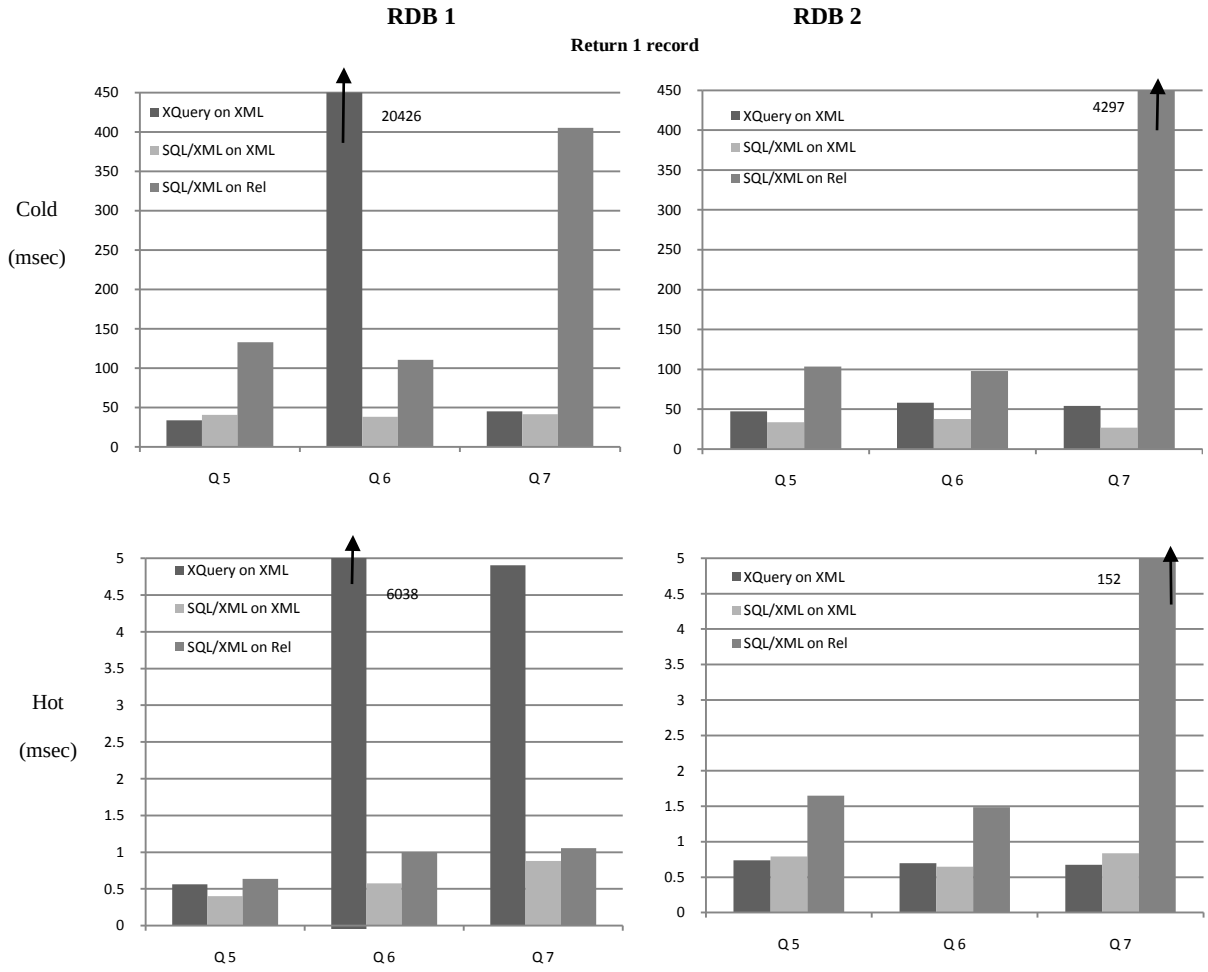


Figure 5-4 Query Response Times in Milliseconds Q5-7 (Result set size = 1).

Figure 5-5 shows the elapsed times of the same queries when they returned 60 results. The larger selectivity means that more documents needed to be processed, which for RDB2 increased the performance gap between native XML column performance (XQuery, SQL/XML) and shredded XML performance (Relational). The SQL/XML performance of XML columns in RDB1 was similar to the performance of RDB2. However, the Relational performance of relational tables in RDB1 was better than the Relational performance of RDB2, especially for Q7.

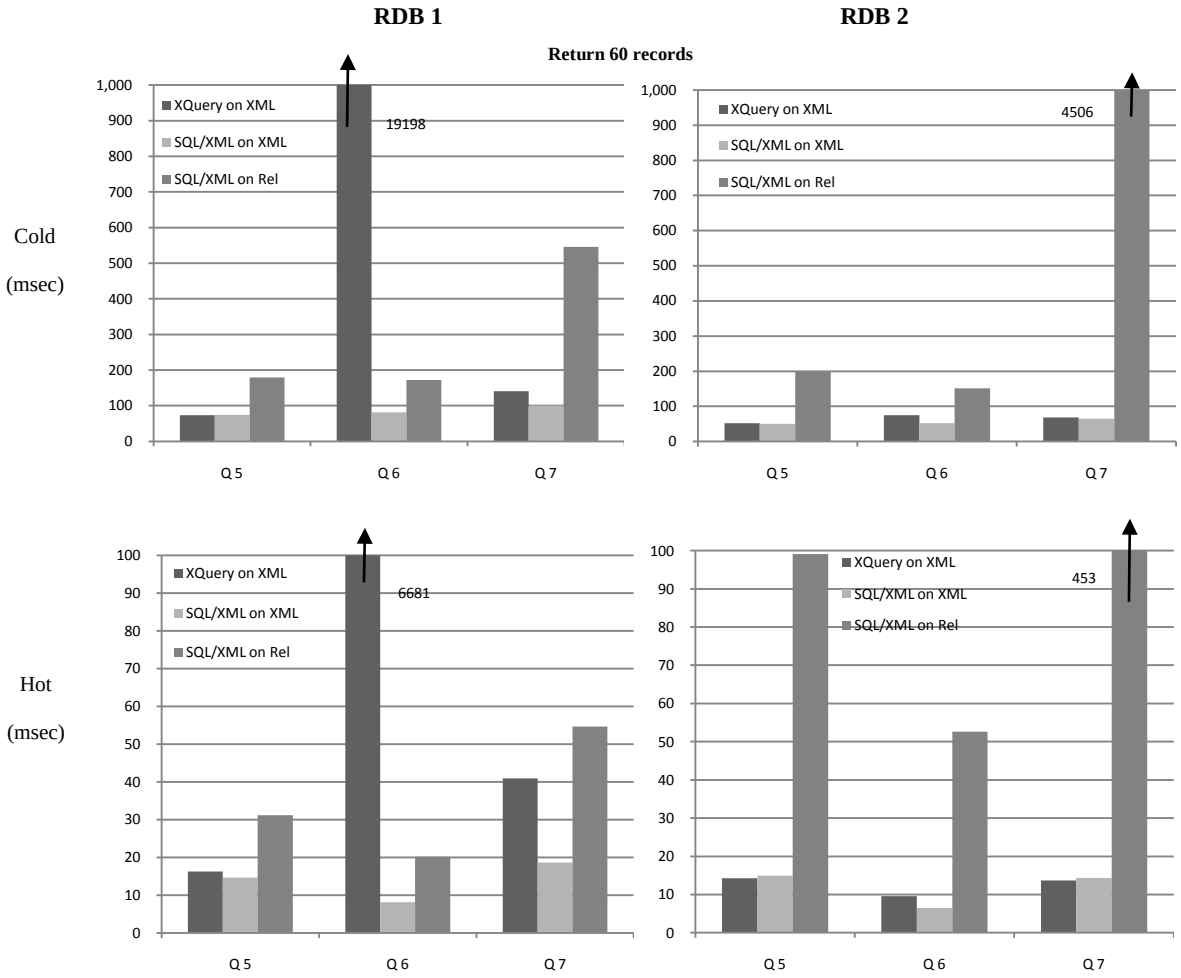


Figure 5-5 Query Response Times in Milliseconds Q5-7 (Result set size = 60).

Figure 5-6 shows the results for the case of 600 result documents. Moving to an even larger result set (600) amplified the difference between native XML and shredded XML storage in both systems.

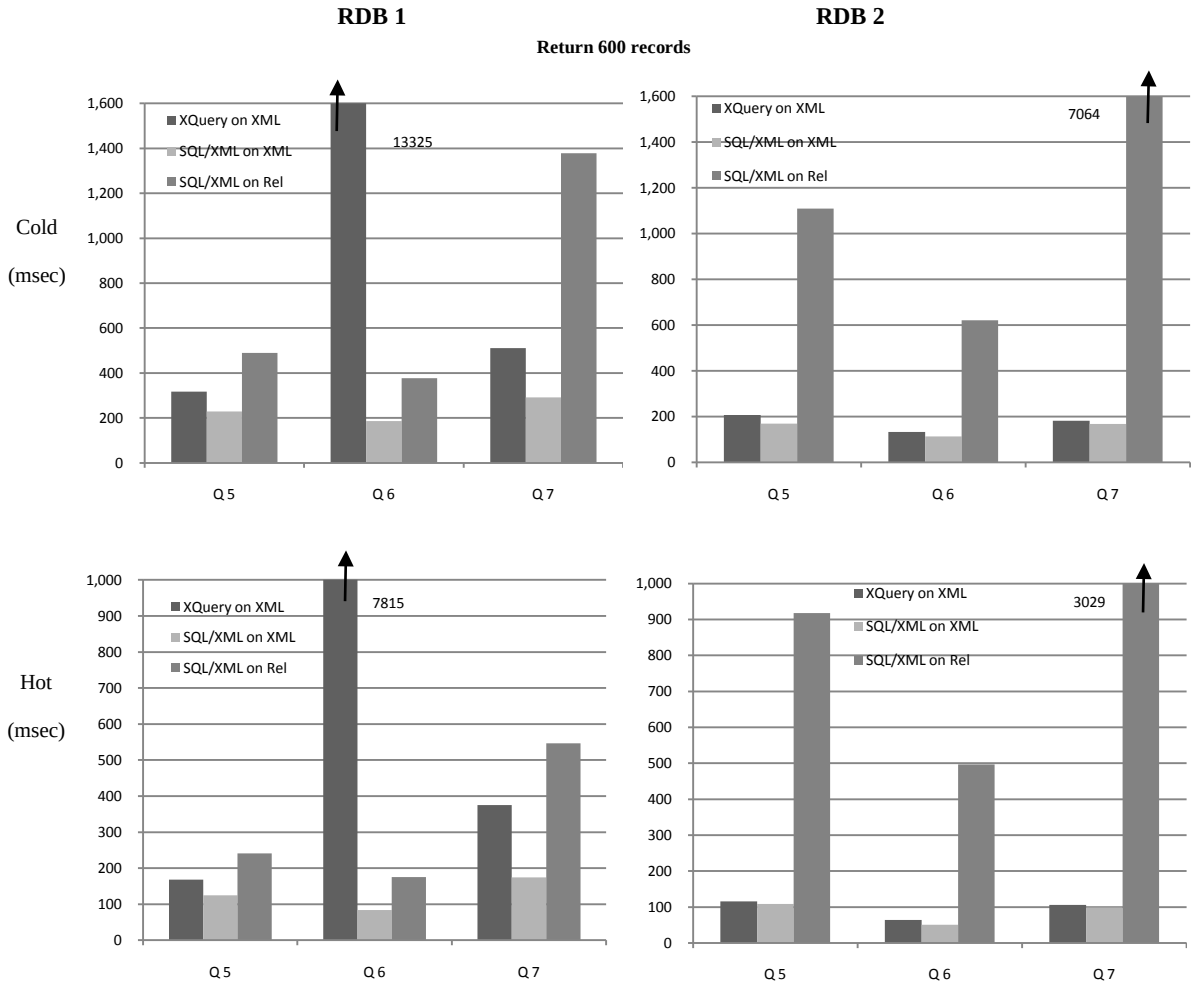


Figure 5-6 Query Response Times in Milliseconds Q5-7 (Result set size = 600).

5.3 Aggregation Queries

As described earlier, Query 8 and Query 9 were basic analytical queries with several predicates for selection and aggregation of a metric of interest. Figure 5-7 shows the results for these queries. Given the predicates in these queries, both queries required access to a large subset of documents that could not be clustered by customer ID. In this situation, random access into the

collection of documents was necessary. Under “cold” conditions, where physical I/O to disk was necessary, RDB2 outperformed RDB1 by almost a factor of 2x~3x in XQuery.

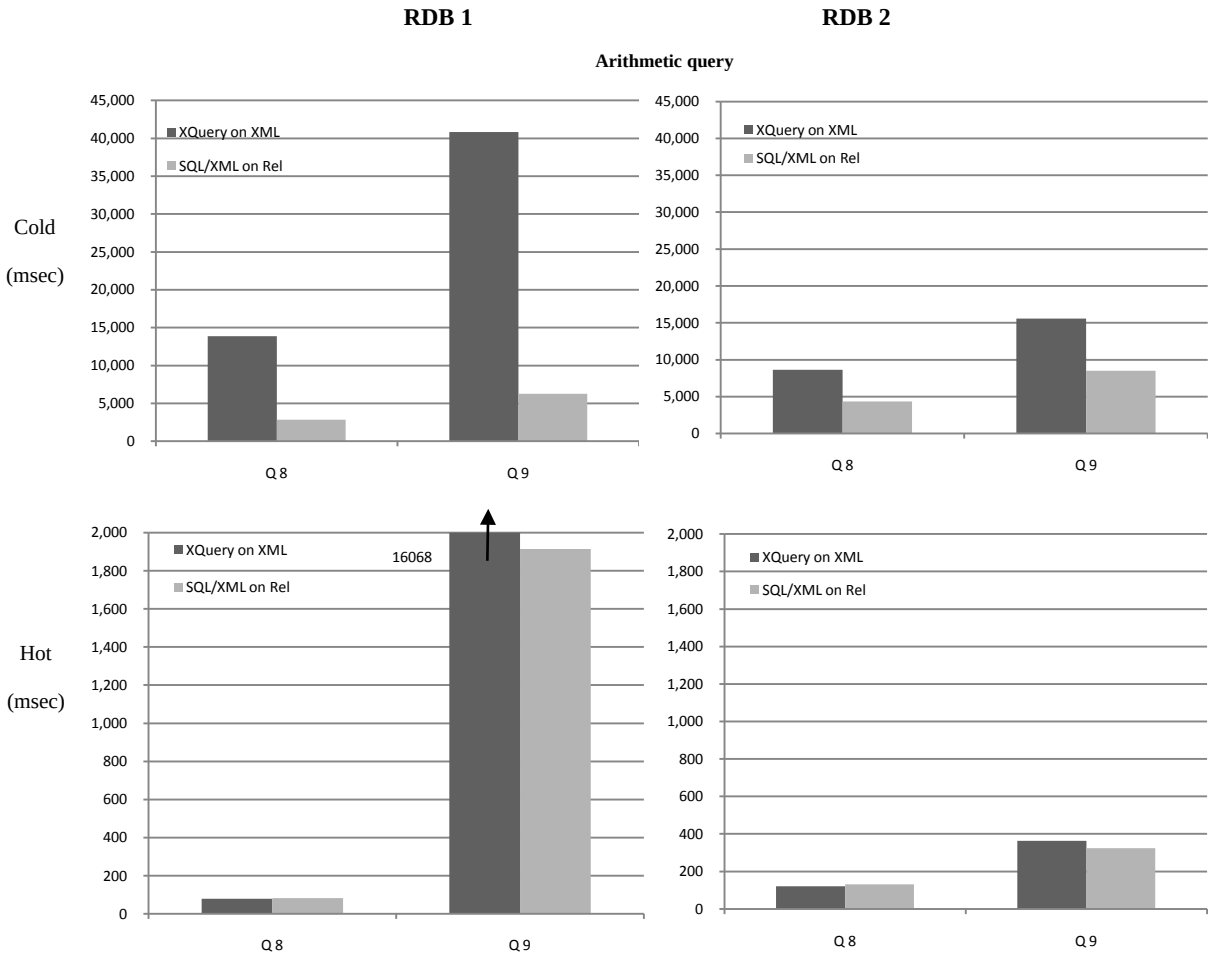


Figure 5-7 Query Response Times in Milliseconds Q8-9– Aggregation queries

5.4 Full Document Inserts and Deletes

Figure 5-8 shows the response times of inserting or deleting all of the information for one customer. These update operations have a similar notion of “cold run” as the query operations, but they do not have “hot run” counterparts. It is not meaningful for an application to execute

repeated insert, delete or update statements with exactly the same parameter values. Therefore, only “cold run” numbers are shown for these operations. In both database systems, these operations were less expensive for the XML column than for the case of shredded XML, as the shredded case required row manipulation for 12 separate tables. For RDB1, delete operations are more expensive than insert operations. However, for RDB2, delete operations were less expensive than insert operations.

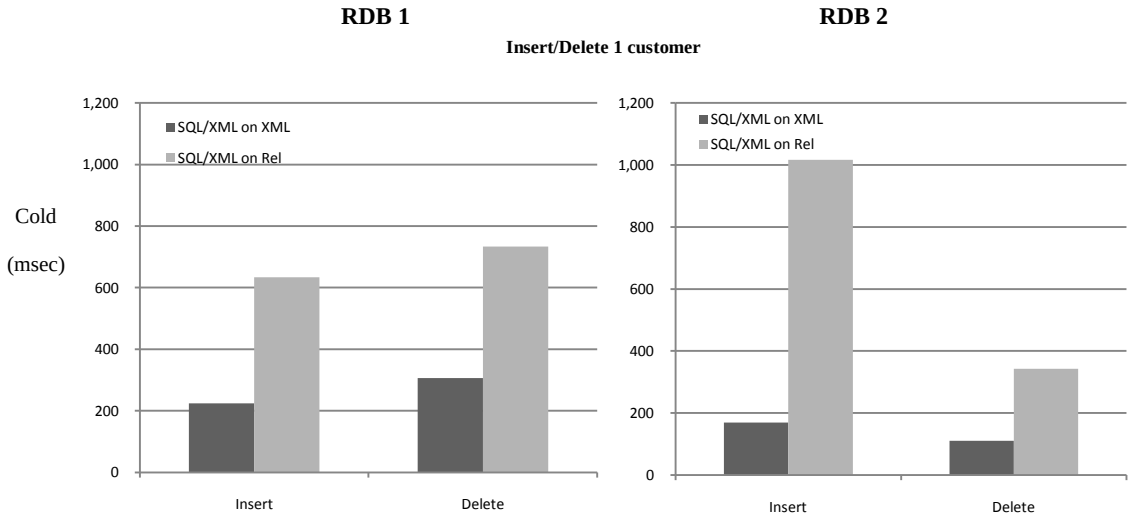


Figure 5-8 Inserting/Deleting a New Customer (Milliseconds)

5.5 Sub-Document Insert, Delete, and Update Operations

The response time for inserting, deleting, and updating XML element nodes within an enclosing XML document are shown in Figure 5-9.

For RDB1, node-insert/delete performance over XML data was worse than shredded relational performance at small width, but the opposite was true at large width. For RDB1, node-update performance on XML data was consistently worse than on shredded relational data at three different widths.

For RDB2, node insert performance over XML data was always better than over relational data (and the larger the width, more advantage there was). Node-update performance for XML data was also better than for relational data. However, delete performance for XML data was close to delete performance for relational data at small widths. Even so, it was still better than relational part at large width. As the width became larger, the relational node-insert/delete/update cost increased more rapidly than the XML node-insert/delete cost.

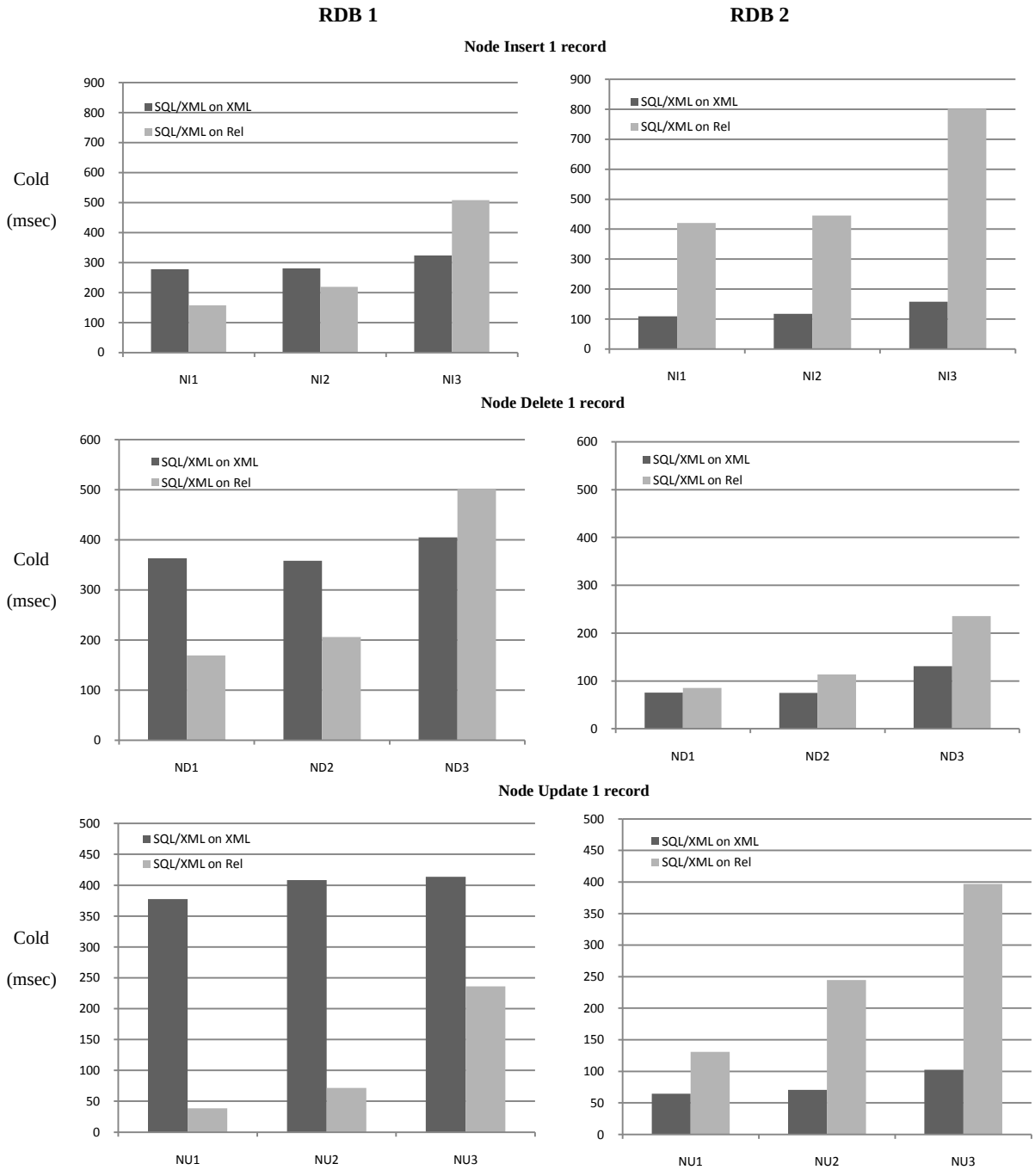


Figure 5-9 Inserting/Deleting/Updating Nodes in an Existing XML Document (Milliseconds)

6. Lessons

When designing, implementing and conducting experiments on this EXRT project, a number of difficulties and uncertain issues require careful thinking and make further investigations necessary. Some of the highlights of these experiences are summarized here.

The biggest challenge throughout was to design and implement a benchmark driver that could drive the benchmark queries to produce meaningful and reproducible results for each system. This was difficult because the results could be affected by many different factors, including the running environment, “cold” or “hot” running conditions, random parameters, JDBC usage method and driver type, etc. Working through these factors required quite a bit of repeated trial and error.

Another major challenge was to make sure the results collected demonstrated the optimal performance of each operation under the desired circumstances. For example, the same XQuery could be written as one of several different statements involving different XPath expressions. Since they might have different query plans, these different statements might have different performance. It was therefore necessary to verify each system’s optimization processes and make certain they were using the best plans based on the information collected by their query

processors. To overcome all of these difficulties, it was necessary to discuss with each vendor about our approaches.

Another minor implementation question encountered during the table-schema design is how to compute the element-position information for the relational tables for the partial update operations. Two different approaches were considered.

(a) Count the total number of same aggregated elements inside the parent node. Based on the total count N, it is possible to assign each element a “position number” ranging from 1 to N and store this “position number” in a column in the corresponding relational table. For example, in the XML document below, there are two different email addresses in the address list. The first one will be assigned position number 1 in the “EID” column of the relational table EmailAddresses, and the second one will be assigned number 2.

```
<Customer id="1011">
    <Addresses>
        <EmailAddresses>
            <Email primary="Yes">Marjo.Villoldo@evergreen.edu</Email>
            <Email primary="No">Fosca.Palomar@co.in</Email>
        </EmailAddresses>
    </Addresses>
</Customer>
```

(b) XQuery “for $email at $eid_no in $ca/Customer/Addresses/EmailAddreses/Email return <PO>...</PO>” is used to step through each email address in the address list and construct

a new XML node for each one. (The “at $eid_no” modifier has the same effect as <xsl:value-of select="position()">.) Then XPath clauses can then be used to split this new XML node into different virtual-table columns in XMLTable() function.

In approach (b), there might be some additional XML-construction effort when using XQuery to return newly constructed XML. However, experimental results revealed that the cost of the second approach was essentially the same as the cost of the first approach. In the tests, for both RDB1 and RDB2, the second approach was used to get the performance numbers for relational insert operation.

Several other important lessons were learned as well as a result of the EXRT benchmarking project:

JDBC driver lessons: The choice of JDBC driver might offer improvement for end-to-end query performance, especially in the “hot” choice. If so, the driver that can give the best performance should be used. In addition, queries with parameter-markers using JDBC’s parameter-binding methods can achieve better results than literal queries with string-replaced values. This suggests that as long as parameter-binding is possible, parameter-markers should be used for query templates. However, not every SQL/XML query or XQuery can support parameter-markers. For example, in RDB1, some XML update functions did not support the

CAST () function, required to convert the data-type of a value represented by a parameter-marker to another type.

Query-rewriting lessons: There were various manual query rewriting approaches available to improve performance in RDB1. For example, some XQuery statements should be re-written as different statements with different XPath expressions or range-predicate expressions, as was previously mentioned. It is an open philosophical question how to deal with these choices in obtaining results for a benchmark like EXRT.

7. Conclusions and Future Work

In this thesis, two different commercially available relational storage schemes for XML data were compared: shredded relational storage and native XML storage (i.e., an XML column). Their corresponding query languages, SQL/XML and XQuery were also compared. In addition, a detailed description of the design of the EXRT benchmark was given. EXRT is a micro-benchmark for evaluating XML data management tradeoff, such as the impact of query characteristics on the performance of shredded versus native XML.

Using the EXRT behchmark, this thesis assessed the robustness of the XML storage and query support of current relational database systems. Specifically, EXRT was used to examine the XML support capabilities of two commercial relational database systems. Significant differences in the performance of these two systems were observed and reported. EXRT also revealed some current deficiencies related to the query optimization of XQuery and SQL/XML queries on one of the systems; patches were provided by the vendor, and the issues the EXRT exposed are to be corrected in the next release of that system.

As expected, no system can outperform the other system in all operations. Considering the issue of shredded XML performance versus natively stored XML, the results were

query-dependent. The tradeoff was partially dependent on the width of the result that those queries are interested in.

Looking ahead, based on the current benchmark used is this study, one possible direction would be to explore the performance of a larger set of commercial relational systems. Also, as the XML support of these systems becomes more mature, it will be necessary to go “deeper” and extend EXRT to include some other operations which might test more characteristics of data-centric XML documents.

The ERXT effort has only focused on data-centric XML documents. It might therefore be interesting for the database community to evaluate the performance of document-centric XML use cases, such as large-scale document management and automated publishing. Additional evaluation methods and analysis effort might be required due to the great difference of characteristics of these two (data versus content) XML use cases.

References

1. Matthias Nicola, Bert van der Linden, Native XML Support in DB2 Universal Database,Proceedings of the 31st VLDB Conference,Trondheim, Norway, 2005.

2. ORACLE XML DB, An Oracle Technical White Paper, January 2004. [Online] http://download-uk.oracle.com/technology/tech/xml/xmldb/current/twp.pdf.

3. What's New for XML in SQL Server 2008? White Paper Published: August 2008. [Online] http://download.microsoft.com/download/a/c/d/acd8e043-d69b-4f09-bc9e-4168b65aaa71/WhatsNewSQL2008XML.doc.

4. MySQL 5.5 Reference Manual, Copyright © 1997, 2010, Oracle and/or its affiliates. [Online] http://dev.mysql.com/doc/refman/5.5/en/.

5. M. Carey, Ling Ling, Matthias Nicola, and Lin Shao,EXRT: Towards a Simple Benchmark for XML Readiness Testing,submitted for publication, July 2010.

6. Mary Holstege,Big, Fast XQuery: Enabling Content Applications. IEEE Data Eng. Bull.31(4):41-48(2008).

7. TPoX. [Online] http://tpox.sourceforge.net/.

8. XQuery 1.0: An XML Query Language. [Online] http://www.w3.org/TR/xquery/.

9. ISO (2003). “Information Technology — Database languages — SQL”. Part 14: XML-Related Specifications (SQL/XML).ISO/IEC 9075-14:2003 International Standards.

10. XML Query Working Group. [Online] www.w3.org/XML/Query/.

11. T. Böhme, E. Rahm: XMach-1: “A Benchmark for XMLData Management”, Proceedings of German database conference BTW2001, pp 264-273, March 2001.

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

13. A. Schmidt, F. Waas, S. Manegold, M. L. Kersten: „A Look Back on the XML Benchmark Project", Intelligent Search on XML, Vol. 2818 of LNCS/LNAI, pp 263-278, 2003.

14. M. Franceschet: "XPathMark - An XPath benchmark for XMark generated data", International XML Database Symposium(XSYM), pp.129-143, 2005.

15. S. Bressan, G. Dobbie, Z. Lacroix, M. L. Lee, Y. G. Li, U. Nambiar and B. Wadhwa: "XOO7: Applying OO7 Benchmark to XML Query Processing Tools", International Conference on Information and Knowledge Management (CIKM), November 2001.

16. B. Yao, M. T. Özsu, and J. Keenleyside: "XBench - A Family of Benchmarks for XML DBMSs", EEXTT 2002 and DiWeb 2002, LNCS Vol. 2590, pp. 162-164.

17. K. Runapongsa, J. M. Patel, H. V. Jagadish, Y. Chen, and S. Al-Khalifa: "The Michigan Benchmark: Towards XML Query Performance Diagnostics", Proceedings of the 29th VLDB Conference, 2003.

18. L. Afanasiev, I. Manolescu and P. Michiels: "MemBeR: A Micro-benchmark Repository for XQuery", XML Symposium(XSym) 2005.

19. I. Manolescu, C. Miachon and P. Michiels: "Towards micro-benchmarking XQuery", Experimental Evaluation of Data Management Systems (EXPDB), 2006.

20. Matthias Nicola, Irina Kogan, Berni Schiefer, An XML transaction processing benchmark,Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data .

21. XQuery Update Facility 1.0,W3C Candidate Recommendation 09 June 2009.

22. XQuery and XPath Full Text 1.0, W3C Candidate Recommendation 28 January 2010. .

23. the ToX XML Data Generator (ToXgene). [Online] http://www.cs.toronto.edu/tox/toxgene/.

24. The Linux Logical Volume Manager . [Online] http://www.redhat.com/magazine/009jul05/features/lvm2/.

25. M.Carey, D.DeWitt and J.Naughton : The 007 Benchmark. In Proc. 1993 ACM SIGMOD Int'l. Conf. on Management of Data, Washington, D.C., 1993.

26. TPoX Benchmark: Workload Driver Overview and Usage, Version 2.0, June, 2009. [Online] http://tpox.svn.sourceforge.net/viewvc/tpox/TPoX20/documentation/WorkloadDriverUsage_v2.0.pdf.

Appendix

DDL

For native XML testing, we loaded the XML data into the XML table created by the DDL statement below:

```
CREATE TABLE CUSTACC(CADOC XML);
```

For the relational storage testing, we shredded the XML data into 12 relation tables. They are: PROFILE, MIDDLENAMES, SHORTNAMES, LANGUAGES, ADDRESSES, STREETS, PHONES, EMAILADDRESSES, CUSTOMERACCOUNTSINFO, ACCOUNTVALUEDATE, ACCOUNTINPUTTER, and ACCOUNTHOLDINGS. We use the DDL statements below to create these 12 tables.

```
CREATE TABLE PROFILE (
    ID INTEGER NOT NULL PRIMARY KEY,
    NAMEMNEMONIC VARCHAR(40) NOT NULL,
    GENDER VARCHAR(6) NOT NULL,
    DATEOFBIRTH DATE NOT NULL,
    NATIONALITY VARCHAR(40) NOT NULL,
    COUNTRYOFRESIDENCE VARCHAR(40) NOT NULL,
    TITLE VARCHAR(15),
    FIRSTNAME VARCHAR(40) NOT NULL,
    LASTNAME    VARCHAR(40) NOT NULL,
    SUFFIX VARCHAR(5),
    CUSTOMERSINCE DATE NOT NULL,
    PREMIUMCUSTOMER VARCHAR(3) NOT NULL,
    CUSTOMERSTATUS VARCHAR(8),
    LASTCONTACTDATE DATE NOT NULL,
    REVIEWFREQUENCY VARCHAR(13) NOT NULL,CURRENCY CHAR(3) NOT NULL,
    LOGIN VARCHAR(30) NOT NULL,
    PINTYPE VARCHAR(60) NOT NULL,
```

```
    PINCIPERVALUE VARCHAR(20) NOT NULL,
    TPTYPE VARCHAR(60) NOT NULL,
    TPCIPHERVALUE VARCHAR(20) NOT NULL,
    TAXID VARCHAR(20),SSNTYPE VARCHAR(60),
    SSNCIPHERVALUE VARCHAR(20),
    TAXRATE VARCHAR(5) NOT NULL);

CREATE TABLE MIDDLENAMES(
    ID INTEGER NOT NULL,
    MIDDLENAME VARCHAR(40) NOT NULL,
    FOREIGN KEY (ID) REFERENCES PROFILE(ID) ON DELETE CASCADE ON UPDATE RESTRICT);

CREATE TABLE SHORTNAMES(
    ID INTEGER NOT NULL,
    SHORTNAME VARCHAR(40) NOT NULL,
    PRIMARY KEY (ID),
    FOREIGN KEY (ID) REFERENCES PROFILE(ID) ON DELETE CASCADE ON UPDATE RESTRICT);

CREATE TABLE LANGUAGES(
    ID INTEGER NOT NULL,
    LANGUAGE VARCHAR(30) NOT NULL,
    FOREIGN KEY (ID) REFERENCES PROFILE(ID) ON DELETE CASCADE ON UPDATE RESTRICT);

CREATE TABLE ADDRESSES(
    ID INTEGER NOT NULL ,
    ADPRIMARY VARCHAR(3) NOT NULL,
    ADTYPE VARCHAR(9) NOT NULL,
    POBOX VARCHAR(10),
    CITY VARCHAR(30) NOT NULL,
    POSTALCODE VARCHAR(10) NOT NULL,
    STATE VARCHAR(30) NOT NULL,
    COUNTRY VARCHAR(40) NOT NULL,
    CITYCOUNTRY VARCHAR(80) NOT NULL,
    PRIMARY KEY(ID, POSTALCODE),
    FOREIGN KEY (ID) REFERENCES PROFILE(ID) ON DELETE CASCADE ON UPDATE RESTRICT );
```

```
CREATE TABLE STREETS(
    ID INTEGER NOT NULL,
    POSTALCODE VARCHAR(10) NOT NULL,
    STREETS VARCHAR(100),
    FOREIGN KEY (ID) REFERENCES PROFILE(ID) ON DELETE CASCADE ON UPDATE RESTRICT,FOREIGN KEY (ID, POSTALCODE) REFERENCES ADDRESSES (ID, POSTALCODE) ON DELETE CASCADE ON UPDATE RESTRICT);

CREATE TABLE PHONES(
    ID INTEGER NOT NULL,
    POSTALCODE VARCHAR(10) NOT NULL,
    PHPRIMARY VARCHAR(3) NOT NULL,
    PHTYPE VARCHAR(9) NOT NULL,
    COUNTRYCODE VARCHAR(3),
    AREACODE VARCHAR(3),
    PHONENUM VARCHAR(7),
    EXTENSION VARCHAR(4),
    FOREIGN KEY (ID) REFERENCES PROFILE(ID) ON DELETE CASCADE ON UPDATE RESTRICT);

CREATE TABLE EMAILADDRESSES(
    ID INTEGER NOT NULL,
    EMPRIMARY VARCHAR(3) NOT NULL,
    EMAILADDRESS VARCHAR(60),
    FOREIGN KEY (ID) REFERENCES PROFILE(ID) ON DELETE CASCADE ON UPDATE RESTRICT);

CREATE TABLE CUSTOMERACCOUNTSINFO(
    ID INTEGER NOT NULL,
    ACCOUNTID CHAR(10) NOT NULL PRIMARY KEY,
    ACCOUNTCATEGORY INTEGER NOT NULL,
    ACCOUNTTITLE VARCHAR(80) NOT NULL,
    ACCOUNTSHORTTITLE VARCHAR(40) NOT NULL,
    ACCOUNTMNEMONIC VARCHAR(40) NOT NULL,
    CURRENCY CHAR(3) NOT NULL,
    CURRENCYMARKET INTEGER NOT NULL,
    ACCOUNTOPENINGDATE DATE NOT NULL,
    ACCOUNTOFFICER VARCHAR(40) NOT NULL,
```

```
    ACCOUNTLASTUPDATE VARCHAR(20),
    ONLINEACTUALBAL BIGINT NOT NULL,
    ONLINECLEAREDBAL BIGINT NOT NULL,
    WORKINGBALANCE BIGINT NOT NULL,
    ACCOUNTPASSBOOK VARCHAR(3) NOT NULL,
    ACCOUNTCHARGECCY CHAR(3) NOT NULL,
    ACCOUNTINTERESTCCY CHAR(3) NOT NULL,
    ACCOUNTALLOWNETTING VARCHAR(3) NOT NULL,
    FOREIGN KEY (ID) REFERENCES PROFILE(ID) ON DELETE CASCADE ON UPDATE RESTRICT);

CREATE TABLE ACCOUNTVALUEDATE(
    ACCOUNTID CHAR(10) NOT NULL,
    VALUEDATE DATE NOT NULL,
    CREDITMOVEMENT DOUBLE NOT NULL,
    VALUEDATEDBAL BIGINT NOT NULL,
    FOREIGN KEY (ACCOUNTID) REFERENCES CUSTOMERACCOUNTSINFO(ACCOUNTID) ON DELETE
CASCADE ON UPDATE RESTRICT );

CREATE TABLE ACCOUNTINPUTTER(
    ACCOUNTID CHAR(10) NOT NULL,
    C CHAR(1),
    INPUTTER VARCHAR(40) NOT NULL,
    FOREIGN KEY (ACCOUNTID) REFERENCES CUSTOMERACCOUNTSINFO(ACCOUNTID) ON DELETE
CASCADE ON UPDATE RESTRICT );

CREATE TABLE ACCOUNTHOLDINGS(
    ACCOUNTID CHAR(10) NOT NULL,
    POSITIONSYMBOL VARCHAR(10) NOT NULL,
    POSITIONNAME VARCHAR(80) NOT NULL,
    POSITIONTYPE VARCHAR(20) NOT NULL,
    POSITIONQUANTITY DOUBLE NOT NULL,
    FOREIGN KEY (ACCOUNTID) REFERENCES CUSTOMERACCOUNTSINFO(ACCOUNTID) ON DELETE
CASCADE ON UPDATE RESTRICT );
```

Index

For query efficiency, we created XML indexes on these XML element nodes in the XML column of the XML table:

Customer/@id;

Customer/Accounts/Account/@id;

Customer/Addresses/Address/Country;

Customer/Nationality;

Customer/ BankingInfo/Tax/TaxRate;

Similarly, we created relational indexes on the following columns of the relational tables (not including their primary keys, which were all indexed as well).

ACCOUNTINPUTTER (ACCOUNTID);

ACCOUNTVALUEDATE (ACCOUNTID);

ACCOUNTHOLDINGS (ACCOUNTID);

CUSTOMERACCOUNTSINFO (ID);

CUSTOMERACCOUNTSINFO (ID, ACCOUNTID);

ADDRESSES (ID);

ADDRESSES (COUNTRY);

EMAILADDRESSES (ID);

LANGUAGES (ID);

MIDDLENAMES (ID);

PHONES (ID);

PHONES (ID, ADDRID);

STREETS (ID);

STREETS (ID, ADDRID);

PROFILE (NATIONALITY);

PROFILE (TAXRATE);

Testing environment

The EXRT benchmark was running on 64bit Redhat Enterprise Linux 5. Here is a concise description of the system that was used.

PROCESSOR	Intel(R) Core(TM)2 Duo CPU E8500 @ 3.16GHz Cache size : 6144 KB CPU cores : 2 Address sizes : 36 bits physical, 48 bits virtual
MEMORY	MemTotal: 4GB Type: DDR2 Type Detail: Synchronous Speed: 800 MHz (1.2 ns) Manufacturer: 7F7F7F0B00000000 SwapTotal: 8385920 kB
DISK	Disk /dev/sda: 320.0 GB, Manufacturer: Western Digital, RPM:7200 Disk /dev/sdb: 320.0 GB, Manufacturer: Western Digital, RPM:7200
DATA LAYOUT	Database systems are installed on disk /dev/sda; Database instances are created on a striped volume which uses two disks, each of which has a 100GB partition of the volume
OS	Red Hat Enterprise Linux Client release 5.4 (Tikanga)