Volume 7, Issue 5, May 2017 ISSN: 2277 128X
International Journal of Advanced Research in Computer Science and Software Engineering
Research Paper
Available online at: www.ijarcsse.com

Performance Evaluation of Native XML Database and XML Enabled Database

S. Balamurugan*
Ph.D., Research Scholar
Department of Computer Science and Engineering,
Faculty of Engineering and Technology
Annamalai University, Tamilnadu, India

Dr. A. Ayyasamy
Assistant Professor
Department of Computer Science and Engineering
Faculty of Engineering and Technology
Annamalai University, Tamilnadu, India

DOI: 10.23956/ijarcsse/SV7I5/0109

Abstract— Now-a-days, there is a authentic need for a database system for storing, retrieving and manipulate XML based data to the purpose of exchange data over the web with an efficient manner. XML provides a noteworthy boost to web-based and business-to-business (B2B) applications. Data were normally stored in relational databases and XML was used as a medium to transport data between web-based and business-to-business (B2B) applications. XML quickly became the de facto standard for deploying applications that managed large volumes of data and either wanted to be able to communicate with other businesses or to expose their data on the web. The paper helps to explore and compare between the performance of XML-based database and native XML database. This model was supported by XML-enabled databases (XEDs), which adopted a simple strategy to store XML data: each XML document is decomposed and its data are stored within tables. The fundamental difference between XEDs and NXDs is that the latter adopt the XML data model for storing XML data. Much like hierarchical and object databases, they are able to preserve the hierarchy and ordering of nodes of XML documents in a much more efficient manner than XML-enabled databases, hence the tremendous performance improvement in handling large XML documents. XML Data provides a new approach to data management, which positively impact many organization manage and exchange information.

Keywords- XML-Enabled database, Native XML database, Performance analysis, data model, XML document.

I. INTRODUCTION

XML is the abbreviation for Extensible Markup Language [1]. This is an open and popular standard for marking up text in a way that is both machine and human readable. By "marking up text" we mean that the data in the text files is formatted to include meaningful symbols that communicate to a reader what that data is for. The syntax of XML is similar in style to HTML, the markup language of the World Wide Web (WWW).The data in an XML file can be organized into hierarchies so that the relationships between data elements are visually obvious. The data and the structure are always presented together. XML is intuitive once a few simple syntax rules are understood. Designing a good XML structure is something a non-technical business subject matter expert can do after minimal instruction. Beyond simple XML text, there are related technologies such as XML Schemas (XSD) and XML Transformations (XSL). These technologies complement XML text documents by adding value in the areas of validation and processing. There are three different types of XML databases:

1. ***Native XML Database (NXD): a database that:***
   a. Defines a (logical) model for an XML document [2] as opposed to the data in that document and stores [16] and retrieves documents according to that model. At a minimum, the model must include elements, attributes, PCDATA, and document order. Example: XPath data model.
   b. Has an XML document as its fundamental unit of (logical) storage, just as a relational database has a row in a table as its fundamental unit of (logical) storage.
   c. Is not required to have any particular underlying physical storage model. For example, it can be built on a relational, hierarchical, or object-oriented database, or use a proprietary storage format such as indexed, compressed files.

2. ***XML Enabled Database (XEDB):***

A database that has an added XML mapping layer provided either by the database vendor or a third party. This mapping layer manages the storage and retrieval of XML data. Data that is mapped into the database is mapped into application specific formats and the original XML meta-data and structure may be lost. Data retrieved as XML is NOT guaranteed to have originated in XML form. Data manipulation may occur via either XML specific technologies (e.g. XPath, XSLT and DOM) or other database technologies (e.g. SQL). The fundamental unit of storage in an XML Enabled Database is implementation dependent.

3. Hybrid XML Databases (HXD):

A database is that can be treated as either a Native XML Database or as an XML Enabled Database depending on the requirements of the application.

II. XML-ENABLED DATABASES

The need to support XML data storage, retrieval and update by utilizing existing database systems, along with their well-established technologies, is the main reason that the solution of the XML-enabling layer over the RDBMS [11] was adopted. This extra software layer contains extensions for transferring data between XML documents and the data structures supported by the underlying databases. The storage methods employed by XML-enabled databases [5, 17] for XML data, along with the main features the systems provide for retrieving, converting and updating XML data are the main topics that are discussed in the sections that follow.

2.1 Storing XML in an XED

XML-enabled databases are primarily used in settings where XML is the format for exchanging data between the underlying database and an application or another database, for example when a Web service is providing data stored in a relational database as an XML document. The main characteristic of the XML-enabled database storage methodology is that it uses a data model other than XML, most commonly the relational data model. Individual instances of this data model are mapped to one or more instances of the XML data model, for example a relational schema [9] can be mapped to different XML schemas depending on the structure of the XML document required by the application that will consume it.

As shown Fig.1, the XML documents are used as a data exchange format without the XML document to have any particular identity within the database. For example, suppose XML is used to transfer temperature data from a weather station to a database. After the data from a particular document is stored in the database, the document is discarded. To the opposite direction, when an XML document is requested as a result of a query, it is constructed from the results that are retrieved after querying the underlying database, and once it is consumed by the client that has requested it, it is again discarded. There is no way to ask explicitly for a document by its name, nor does any guarantee exist that the original document that was stored in the database can be reconstructed. Because of this, it is not a good idea to shred a document into an XML-enabled database as a way of storing it. The basic use of XML-enabled systems is for publishing existing relational data (regardless of its source) as XML. The process of extracting data from a database and constructing an XML document (or XML fragments) is known as publishing or composition. The reverse process (extracting data from an XML document/fragment and storing it in the database) is known as redding or decomposition.

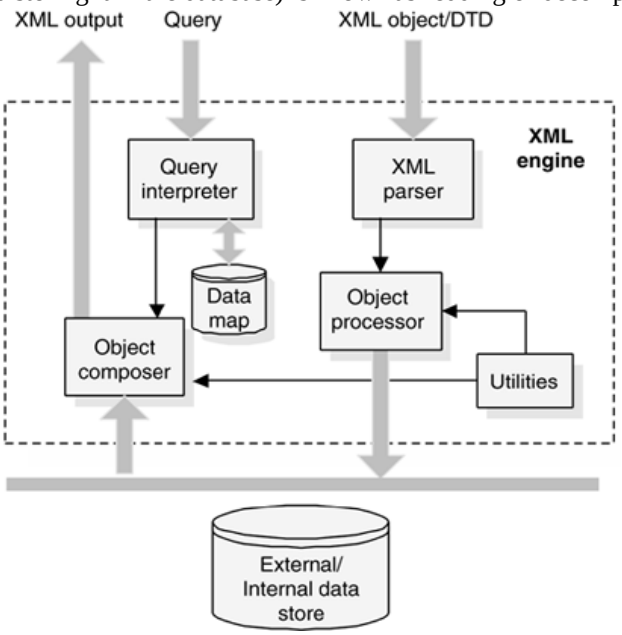

Fig.1. Architecture of Native XML Database XML Engine

Regardless of whether we are shredding or publishing XML documents, there are two important things to note here. First, an XML-enabled database does not contain XML, i.e. the XML document is completely external to the database. Any XML document/fragment is constructed from data already in the database or is used as a source of new data to store in the database. Second, the database schema matches the XML schema, that is, a different XML schema is needed for each database schema.

XML-enabled databases generally include software for performing both publishing and shredding between relational data and XML documents. This extra piece of software can either be integrated into the database or be provided by a third-party vendor outside the database. This software, used by XML-enabled databases, cannot, generally, handle all possible XML documents. Instead, it can handle the subclass of documents that are needed to model the data found in the database.

2.2 Relational-to-XML schema mapping

When using an XML-enabled database, it is necessary to map the database schema to the XML schema (or vice versa). Such mappings are many-to-many. For example, a database schema for sales order information can be mapped to an XML schema for sales order documents, or it can be mapped to an XML schema for reports showing the total sales by region. There are two important kinds of mappings:

(a). Table-based mapping
(b). Object-relational mapping

Both table-based mapping and object-relational mappings define bi-directional mappings, that is, the same mapping can be used to transfer data both to and from the database.

2.3 Table-based mapping:

When using a table-based mapping, the XML document must have the same structure as a relational database. That is, the data is grouped into rows and rows are grouped into \tables". In the following example, the SalesOrders and the Items elements represent the corresponding relational tables containing a list of SalesOrder and Item elements, respectively that in turn represent the rows of each table.

```
<Database>
<SalesOrders>
<SalesOrder>
<Number>123</Number>
<OrderDate>2003-07-28</OrderDate>
<CustomerNumber>456</CustomerNumber>
</SalesOrder>
</SalesOrders>
<Items>
<Item>
<Number>1</Number>
<PartNumber>XY-47</PartNumber>
<Quantity>14</Quantity>
<Price>16.80</Price>
<OrderNo>123</OrderNo>
</Item>
<Item>
<Number>2</Number>
<PartNumber>B-987</PartNumber>
<Quantity>6</Quantity>
<Price>2.34</Price>
<OrderNo>123</OrderNo>
</Item>
</Items>
</Database>
```

2.4 Object-Relational Mapping:

When using an object-relational mapping, an XML document is viewed as a set of serialized objects and is mapped to the database with an object relational mapping. That is, objects are mapped to tables, properties are mapped to columns and inter-object relationships are mapped to primary key / foreign key relationships. Below is the same document containing sales orders, as above, but encoded using the object-relational mapping.

```
<Database>
<SalesOrder>
<Number>123</Number>
<OrderDate>2003-07-28</OrderDate>
<CustomerNumber>456</CustomerNumber>
<Item>
<Number>1</Number>
<PartNumber>XY-47</PartNumber>
<Quantity>14</Quantity>
<Price>16.80</Price>
</Item>
<Item>
<Number>2</Number>
<PartNumber>B-987</PartNumber>
<Quantity>6</Quantity>
<Price>2.34</Price>
```



```
</Item>
</SalesOrder>
</Database>
```

2.5 Retrieving and Modifying Query Languages

Query languages in XML-enabled databases are used to extract data from the underlying database and transform it. The most widely used query languages for this purpose, *SQL/XML* and *XQuery*. For XML-enabled relational databases, the most widely used query language is SQL/XML, which provides a set of extensions to SQL for creating XML documents and fragments from relational data and is part of the ISO SQL specification of 2003. The main features of SQL/XML are the provision of an XML data type, a set of scalar functions, XMLELEMENT, XMLATTRIBUTES, XMLFOREST, and XMLCONCAT, and an aggregate function, XMLAGG. For example, the following call to the XMLELEMENT function:

```
XMLELEMENT (NAME Customer,
XMLELEMENT (NAME Name, customers.name),
XMLELEMENT (NAME ID, customers.id))
```

constructs the following Customer element for each row in the customers table:

```
<Customer>
<Name>customer name</Name>
<ID>customer id</ID>
</Customer>
```

2.6 Updating XML-enabled databases

The solutions related to updating XML data in XML-enabled databases vary from implementation to implementation, due to the idiosyncrasies of the storage model. The most common practice is either to use a custom extension of the SQL/XML prototype to support updates or use a, custom again, product specific API to perform the modification operations over the stored XML data. In either case, though, the update action does not happen in-place, i.e. directly on the tables where the XML data is stored, but rather the document to be updated is extracted from the database, loaded in memory, updated, then returned to the XML-enabling software layer where it is shredded again and stored back to the database.

III. NATIVE XML DATABASES

An XML database [20] is *native* if it (a) defines a logical model for an XML document and stores and retrieves documents according to that model and (b) has an XML document as its fundamental unit of (logical) storage, while (c) the storage model itself is not constrained.

3.1 XML data models

A data model is an abstraction which incorporates only those properties thought to be relevant to the application at hand. As there are different ways in which one can use XML data, there are more than one XML data models. As shown Fig.2, all XML data models include elements, attributes, text, and document order, but some also include other node types, such as entity references and CDATA sections, while others do not.

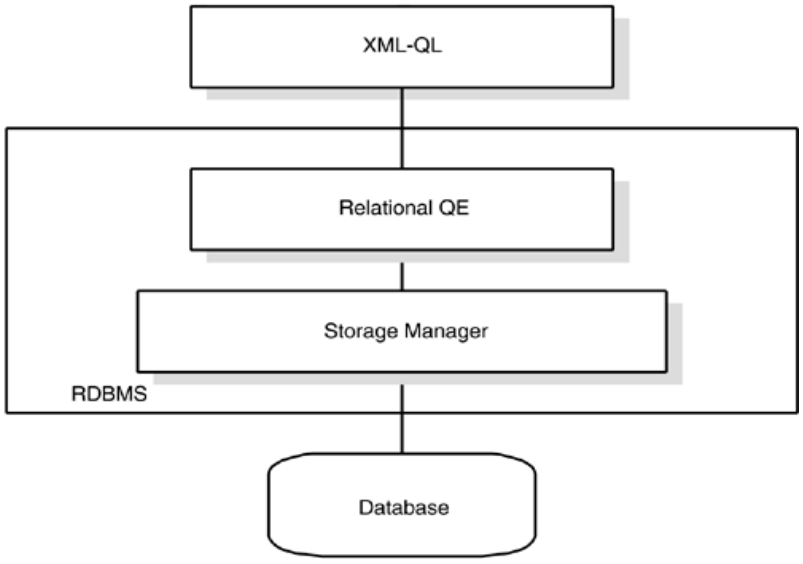

Fig.2.Architecture of the XML-Enabled Database

Therefore, DTD defines its own XML data model, while XML Schema uses the XML InfoSet data model; XPath [16] and XQuery define their own common data model for querying XML data. This excess of data models makes it difficult for applications to combine different features, such as schema validation together with querying. The W3C is therefore in

the process of merging these data models under the XML InfoSet, which is to be replaced by the Post-Schema-Validation Infoset (PSVI). Since many NXDs were created prior to the XML InfoSet and the XPath 2.0 and XQuery data model, they were free to define their own data model. Since the data model of an XML query language [7] defines the minimum amount of information that a native XML database must store, these NXDs need to be upgraded to support XQuery. Fortunately, the vast majority of NXDs currently supports at least XPath 1.0, and it is envisaged that all of them will support XQuery in the near future.

3.2 Fundamental unit of storage

A fundamental unit of storage [5] is the smallest grouping of data that logically file together in storage. From a relational database perspective, the fundamental unit of storage is a tuple. From a native XML database perspective, the fundamental unit of storage is a document. Since any document fragment headed by a single element is potentially an XML document, the choice of what constitutes an XML document is purely a design decision.

3.3 Text-Based Native XML Databases

Text-based native XML databases [18] store XML as text. This may be a file within the database itself, a file in the file system outside the database, or a proprietary text format. Note here that the first case implies that relational databases storing XML documents within CLOB (Character Large OBject) fields are considered native.

All text-based NXDs make use of indices, giving them a big performance advantage when retrieving entire documents or document fragments, as all it takes is a single index-lookup and a single read operation to retrieve the document. This is contrary to XML-enabled databases and some model-based NXDs which require a large number of joins in order to recreate an XML document that has been shredded and inserted in the database. This makes the comparison between text-based NXDs and relational databases analogous to the comparison between hierarchical and relational databases, in that text-based NXDs will outperform relational databases when returning the document or document fragments in the form in which the document is stored; returning data in any other form, such as inverting the document hierarchy, will lead to performance problems.

3.4 Model-Based Native XML Databases

Model-based NXDs have an internal object model and use this to store their XML documents. The way this model is stored varies: one way is to use a relational or object-oriented database; other databases use proprietary storage formats, optimized for their chosen data model. Model-based NXDs built on other databases will have performance similar to those databases, since they rely on these systems to retrieve data; the data model used, however, can make a notable difference in performance. Model-based NXDs using a proprietary storage format usually have physical pointers between nodes and therefore are expected to have similar performance to text-based NXDs. Clearly, the output format is significant here, as text-based NXDs will probably be faster in outputting in text format, whereas a model-based NXD using the DOM model will be faster in returning a document in DOM format.

3.5 Features

Document Collections Most NXDs organize documents within *collections* (possibly nested), which are similar to tables in a relational database or directories in a file system. This feature allows querying and manipulation of the documents within a collection as a set. An NXD [20] supporting collections *may not* require a schema to be associated with a collection; although this provides a high degree of exibility in application development, it raises the risk of low data integrity. Query Languages At first, most NXDs supported either a proprietary query language or programmatic access through DOM. With the advent of XPath, most NXDs built support for it. However, XPath was not designed to be a query language (XPath 1.0 is defined as \a language for addressing parts of an XML document, designed to be used by both XSLT and XPointer)and therefore has several limitations when used as one, including lack of grouping, sorting, cross document joins, and support for data types.

XQuery [3] has just become a W3C Recommendation. Many NXDs have been building support for it based on the XQuery Working Drafts and it is envisaged that XQuery will become the de facto XML query language. Note that this will probably not include object-oriented NXDs; as an example, Ozone5 supports an ODMG6 interface. Note, however, that the existence of standalone or embeddable XQuery engines means that the lack of XQuery support in an NXD does not prohibit its use when the application requires XQuery support. It is worth noting here that almost all NXDs support, in some form or another, the creation of indices on the data stored in collections, to improve the query execution speed. The implementation of these indices varies widely between each NXD product, and is mainly based on the storage implementation. The three possible types of indices are:

1. Value indices index text and attribute values.
2. Path indices index the location of elements and attributes.
3. Full-text indices index the individual tokens in text and attribute values. Most NXDs support both value and path indices, while some NXDs support full-text indices.

Updates and Deletes Up to now, there is no standard for updating XML data. Because of this, most NXDs provide only simple replace/delete operations on documents. Updates on node level are possible only by retrieving a certain document, modifying it using DOM/SAX, then returning it to the database. There are some NXDs that support proprietary update languages, while the hope for a standardized update language for XML rests within the XML. DB XUpdate language and extensions to XQuery, which builds on the work of both have been implemented in a number of

NXDs. Transactions, Locking, and Concurrency All NXDs support transactions, but the level at which locking is implemented differs. Most often, locking occurs at the document level, and this could lead to problems in multi-user concurrency. The problem with node-level locking is that, in order to lock a node *n*, the parent of *n*, *pn* needs to be locked too, otherwise another transaction would be able to delete *pn* and therefore delete *n*. This goes all the way to the root of the document, making it impossible to update other parts of the document that are not in the path from the root to *n*.

A partial solution to this problem has been proposed by where the locked node is annotated with a query that defines the path from the locked node to the target node that is the node to be updated. There are a few commercial NXDs that already support node-level locking and, in the future, most NXDs will probably offer node-level locking. Application Programming Interfaces (APIs) All NXDs provide at least one API in a programming language such as Java/C++/C# etc. These provide an interface for connecting to the database, executing queries, retrieving results and performing other tasks. At the moment, there is only one vendor-neutral XML API [6], XML: DB API from XML: DB.org. It is programming language-neutral, supports XPath, is currently being extended to support XQuery, and is supported by most NXDs and some XEDs. A second XML API, relating to XQuery in the same way that JDBC relates to SQL, is XQuery API for Java (XQJ).The most native XML databases [13] also offer the ability to execute queries and return results over HTTP.

3.6 Round-Tripping

A significant issue in XML databases [14] is the round-tripping problem, that is, the ability of an XML database to store a document and then retrieve it without changing it. There are various levels of round-tripping: other databases are able to offer only basic round-tripping, that is for elements, attributes, text and hierarchy, and are therefore better suited for data-centric applications, while others extend their round-tripping support for processing instructions, comments and even physical structure (whitespace) | these are better suited for document-centric applications and applications that are required by law to keep exact copies of documents. As a general rule, model-based NXDs offer round-tripping support at the level of their model, while text-based NXDs can offer the total round-tripping support.

Normalisation & Referential Integrity Normalisation is one of the cornerstones of relational theory and has as a goal to avoid unnecessary redundancy in a database; as a consequence, this also eliminates the risk of inconsistent data.

Unlike relational data, there are many cases where normalisation for XML data is a non-issue, as there is no data redundancy | and this is usually for document-centric XML documents [4]. For an overview on XML Normal Form (XNF), closely related to normalisation is referential integrity, which refers to the validity of pointers to related data; this is a necessary part of maintaining a consistent database state and complements normalisation. In a relational setting, referential integrity refers to checking that a primary key tuple referenced by a foreign key actually exists. In an NXD, referential integrity refers to checking that pointers in XML documents refer to valid documents or document fragments. There are several referencing mechanisms in XML. XML Schema supports the notion of a primary and foreign key through the use of key and keyref, while DTD uses ID and IDREF; a third mechanism is XLink, while other proprietary mechanisms exist which are able to reference data from relational databases. Support for referential integrity is offered by all NXDs, by performing a validation check against the schema whenever a document is inserted in the database. If the database does not support node-level updates, then this is sufficient; otherwise the database needs a mechanism for checking referential integrity violations due to updates.

IV. DATABASE DESIGN

A relational database [10] is a powerful data storage and retrieval technology where data is stored as rows in tables and the database has one or more tables. Each row of a table has the same columns as every other row in that table. Data is related between tables using the concept of "foreign keys" so that data in a row of one table can be associated with one or more rows of another table. Data in a relational database is readable by executing SQL queries in a management tool to extract and present the data in any number of ways. The extraction requires an understanding of the database structure, including the foreign key relationships. As shown Fig.3, designing a good non-trivial relational database requires significant training and/or significant experience with relational database design techniques.

Relational Schema for the Book Sales and Reports System

Relation Customer (Customer_no, Customer_name, Sex, Postal_code,
Telephone, Email)
Relation Customer_address (Customer_no, Address_type, Address,
City, Sate, Country, Is_default)
Relation Invoice (Invoice_no, Customer_no, Quantity,
Invoice_amount, Invoice_date, Shipment_type, Shipment_date)
Relation Invoice_item (Invoice_no, Item_no, quantity, Unit_price,
Invoice_price, Discount)
Relation Item (Item_no, Item_name, Catalog_type, Author, Publisher,
Item_price)
Relation Category (Catalog_type, Catelog_description)
Relation Shipment (Shipment_type, Shipment_description)
Relation Monthly_sales (Year, Month, Quantity, Total)
Relation Customer_sales (Year, Month, Customer_no, Quantity, Total)
Relation Item_sales (Year, Month, Item_no, Quantity, Total)

The ER diagram of the relational schema:

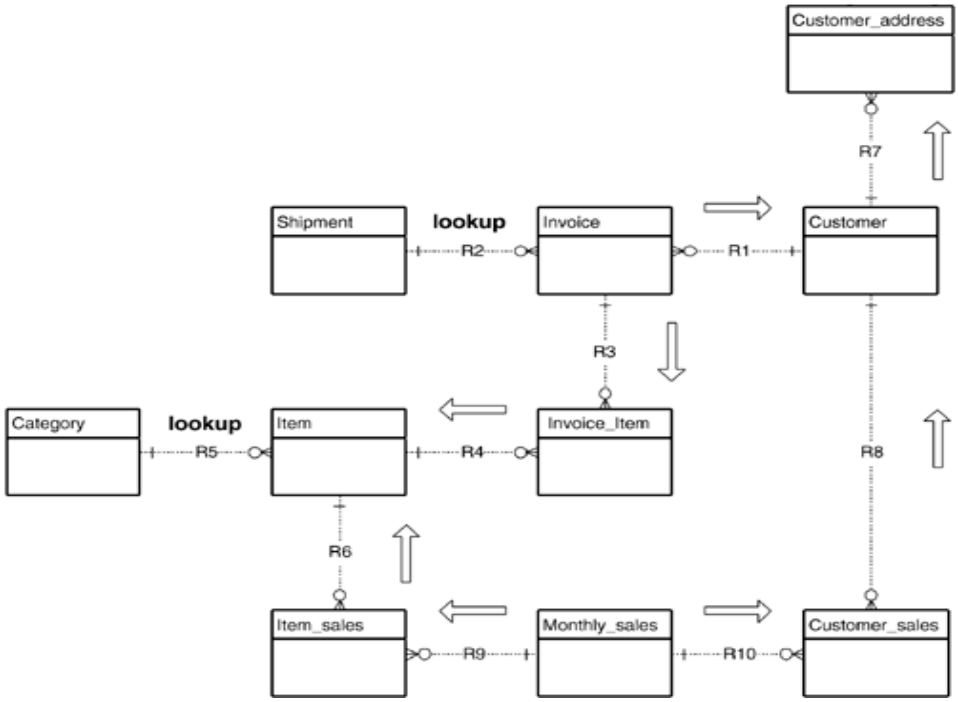

Fig. 3.ER diagram for Book Sales and Reports System

DTD corresponding to the Relational Schema follows:

```
<! ELEMENT        Sales (Invoice*, Customer*, Item*,
Monthly_sales*)>
<! ATTLIST        Sales
  Status         (New|Updated|History) #required>
<! ELEMENT        Invoice (Invoice_item*)>
<! ATTLIST        Invoice
  Quantity       CDATA    #REQUIRED
  Invoice_amount  CDATA    #REQUIRED
  Invoice_date    CDATA    #REQUIRED
  Shipment_type   (Post|DHL|UPS|FedEx|Ship) #IMPLIED
  Shipment_date   CDATA    #IMPLIED
  Customer_idref  IDREF    #REQUIRED>
<! ELEMENT        Customer (Customer_address*)>
<! ATTLIST        Customer
  Customer_id     ID       #REQUIRED
  Customer_name   CDATA    #REQUIRED
  Sex             CDATA    #IMPLIED
  Postal_code     CDATA    #IMPLIED
  Telephone       CDATA    #IMPLIED
  Email           CDATA    #IMPLIED>
<! ELEMENT        Customer_address EMPTY>
<! ATTLIST        Customer_address
  Address_type    (Home|Office) #REQUIRED
  Address         NMTOKENS #REQUIRED
  City            CDATA    #IMPLIED
  State           CDATA    #IMPLIED
  Country         CDATA    #IMPLIED
  Is_default      (Y|N) "Y">
<! ELEMENT        Invoice Item EMPTY>
<! ATTLIST        Invoice Item
  Quantity        CDATA    #REQUIRED
  Unit_price      CDATA    #REQUIRED
  Invoice_price   CDATA    #REQUIRED
  Discount        CDATA    #REQUIRED
  Item_idref      IDREF    #REQUIRED>
<! ELEMENT        Item EMPTY>
```



```
<! ATTLIST        Item
 Item_id        ID      #REQUIRED
 Item_name      CDATA   #REQUIRED
 Category_type  (Art|Comp|Fict|Food|Sci|Sport|Trav) #REQUIRED
 Author         CDATA   #IMPLIED
 Publisher      CDATA   #IMPLIED
 Item_price     CDATA   #REQUIRED>
<! ELEMENT       Monthly_sales (Item_sales*, Customer_sales*)>
<! ATTLIST       Monthly_sales
 Year           CDATA   #REQUIRED
 Month          CDATA   #REQUIRED
 Quantity       CDATA   #REQUIRED
 Total          CDATA   #REQUIRED>
<! ELEMENT       Item_sales EMPTY>
<! ATTLIST       Item_sales
 Quantity       CDATA   #REQUIRED
 Total          CDATA   #REQUIRED
 Item_idref     IDREF   #REQUIRED>
<! ELEMENT       Customer_sales EMPTY>
<! ATTLIST       Customer_sales
 Quantity       CDATA   #REQUIRED
 Total          CDATA   #REQUIRED
 Customer_idref IDREF   #REQUIRED>
```

The root element, <Sales>, is related to the meaning of the document. The corresponding content elements that follow are<Invoice>, <Customer>, <Item>, and <Monthly_sales>

V. PERFORMANCE ANALYSIS

After analyzing the above points, we suggests that the XML database [15] has better performance than the XML-enabled database for handling XML documents with larger data sizes. Although the XML-enabled database has better performance for small document sizes (number of records <= 1,000), it cannot handle large-sized documents as efficiently due to conversion overhead. In contrast, the native XML database engine directly accesses XML data without conversion.

Although the native XML database provides high in handling XML documents, it does have some disadvantages. From calibrating the database size, both data and index size consumed by the native XML database is much larger than in the XML-enabled database.

Both databases have steady performance in single data operations: insert, delete, and update. From more than hundred records to fifty thousand records, the result is more or less the same. The native XML database has better performance [8] in searching by index. For mass record operations, the native XML database shows advantages in handling large quantities of XML data. Both the results of bulk loading and delete operations are very similar to the results in reconstruction. The XML-enabled database has better performance in small database sizes. The native XML database provides better scalability as the database grows. For mass updates, the native XML database still has advantages, but the difference is not as obvious as in the previous case. As the XML-enabled database needs one SQL statement to perform mass updates, the native XML database achieves this indirectly. We have tried to discover any API of the native XML database that provides mass update functionality but without success. It seems that update functionality is a weakness for this native XML database. Instead, we have to retrieve a single document, change it by another API, and then return the results to the database or display them through XSL. This consumes quite a lot of running time.

The native XML database produced better results in the reporting section, which implies that the native XML database X-Query has performance gains from query optimization [12]. As shown Fig.4 the XML-enabled database starts better but becomes worse as data size grows. Both have advantages and disadvantages in developing applications for using the native XML database APIs, the compatibility is high for applications written in Java. Since the XML-enabled database is a proprietary product, the application can run only in certain operating system environments. When compared to the XML-enabled database, Java is a relatively low-level language. More coding is required, and hence the debugging time and also the maintenance cost are increased if the application becomes complicated. Since the native XML database is a new product, the technology related to this product is not well known to most developers. It takes time for most IT developers to get accustomed to this new product.

Both the XML-enabled database and the native XML database provide good graphical user interfaces. The native XML database uses a Web-based application, which acts as the centralized database administration software, while the XML-enabled database is a Windows-based application. In order to communicate with the database, both the XML-enabled database and the native XML database use the same approach by sending HTTP requests that use a URL syntax. The XML-enabled database needs the annotated schema as a middle tier to map the XML objects into the corresponding database object. The XML-enabled database is developed so that programmers do not have to learn from scratch. However, since the native XML database is a new product, programmers need time to learn new concepts and functions before using it.

Since the native XML database is developed using Java and is Web based, its portability and accessibility outperform the XML-enabled database [19]. This accommodates the Internet trend as information should be accessed anywhere. Portable database features are welcome if the performance is acceptable. The native nature and technology of a native XML database handle XML efficiently as scalability increases. The APIs (e.g., JavaScript, ActiveX, VBScript, etc.) increase its compatibility as well. Despite its poor performance in handling large XML documents, the XML-enabled database does have an advantage in being relatively fast to develop, which is welcome to business requests. Hence, we do not have a preference in choosing between the two products. But if one wants to develop a system that has very complex structure, nesting, and so on, we recommend the native XML database [21].

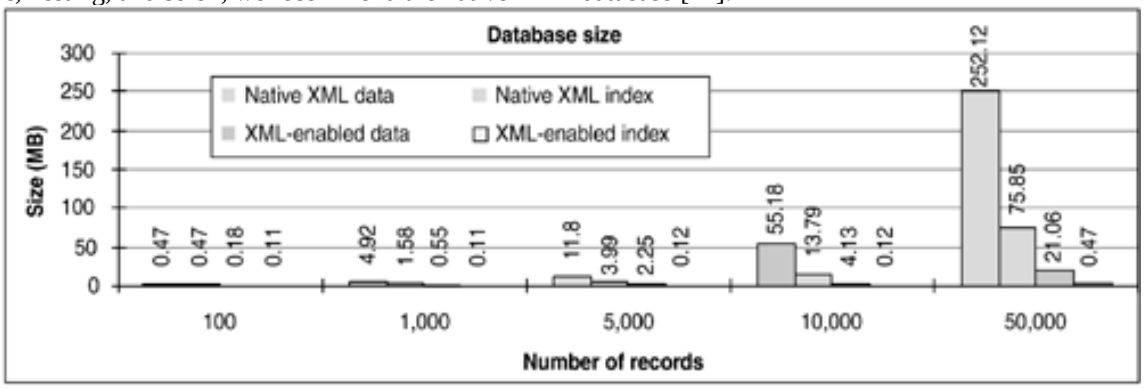

Fig. 4: XML-enabled database v/s Native XML database

VI. CONCLUSIONS

We conclude that the native XML database needs more disk space to store both data and indexes than the XML-enabled database. The growth is almost exponential. The result is more serious as the number of records increases. For 50,000 records, the native XML database has an indexing size over 150 times that of the XML-enabled database. The XML-enabled database controls the sizing much better than the native XML database. From 100 to 10,000 records, the indexing size is approximately 0.11 MB. The larger indexing in the native XML database can be explained by the fact that more comprehensive indexing support is provided, such as full-text searching. Storing XML in the native XML database is not any less space-efficient than decomposing the same data and storing it in a relational database. The only reason for this large size is that the native XML database must store tag names, both elements and attributes, in order to retain the native XML features of the document source. The native XML database has better query optimization than the XML-enabled database for large data sizes. However, the XML-enabled database does dominate for small data sizes.

REFERENCES

[1] http://my.safaribooksonline.com/book/databases/xml/0201844524/contributors/art01lev1sec21.

[2] Bruno, N., Koudas, N., Srivastava, D.: Holistic twig joins: optimal XML pattern matching. *In: SIGMOD'02, ACM Press*, New York, 2002.

[3] Grinev, M., Pleshachkov, P.: Rewriting-based Optimization for XQuery Transformational Queries. *In: IDEAS'05, IEEE Computer Society Press*, Los Alamitos, 2005.

[4] Jagadish, H.V., Lakshmanan, L.V.S., Scannapieco, M., Srivastava, D., Wiwatwattana, N.: Colorful XML: One Hierarchy Isn't Enough. *In: SIGMOD'04, ACM Press*, New York, 2004.

[5] Lapis, G.: XML and Relational Storage-Are they mutually exclusive. *IBM Corporation*, 2005.

[6] Oracle.: Introduction to Oracle XML DB. Oracle XML DB Developer's Guide, 10g Release 2. (2006b), Available at http://download-west.oracle.com/docs/cd/B19306_01/appdev.102/b14259/xdb01int.htm#i1047170

[7] Paparizos, S., Wu, Y., Lakhsmanan, L.V.S., Jagadish, H.V.: Tree Logical Classes for Efficient Evaluation of XQuery. *In: SIGMOD'04, ACM Press,* New York, 2004.

[8] Srivastava, P.G.: Performance Evaluation Tools on Oracle, Department of Computer Science and Computer Engineering, La Trobe University, 2002.

[9] Wang, G., Liu, M., Yu, J.X., Sun, B., Yu, G., Lv, J., Lu, H.: Effective schema-based XML query optimization techniques. *In: IDEAS'03, IEEE Computer Society Press,* Los Alamitos, 2003.

[10] Wang, L., Wang, S., Murphy, B., Rundensteiner, E.: Order-sensitive XML Query Processing over Relational Sources: An Algebraic Approach. *In: IDEAS'05, IEEE Computer Society Press,* Los Alamitos, 2005.

[11] Zhang, C., Naughton, J., Dewitt, D., Luo, Q., Lohman, G.: OHMAN. On Supporting Containment Queries in Relational Database Management Systems. *In: SIGMOD'01, ACM Press,* New York, 2001.

[12] Zhang, X., Pielech, B., Rundesnteiner, E.: Honey, I Shrunk the XQuery! — An XML Algebra Optimization Approach. *In: WIDM'02, ACM Press,* New York, 2002.

[13] XML Databases - The Business Case, Charles Foster, June 2008 - Talks about the current state of Databases and data persistence, how the current Relational Database model is starting to crack at the seams and gives an insight into a strong alternative for today's requirements.

[14] An XML-based Database of Molecular Pathways (2005-06-02) Speed / Performance comparisons of exist, X-Hive, Sedna and Qizx/open

[15] XML Native Database Systems: Review of Sedna, Ozone, NeoCoreXMS 2006
[16] XML Data Stores: Emerging Practices
[17] Bhargava, P.; Rajamani, H.; Thaker, S.; Agarwal, A., *XML Enabled Relational Databases*, Texas, The University of Texas at Austin, 2005.
[18] Initiative for XML Databases
[19] XML and Databases, Ronald Bourret, September 2005
[20] The State of Native XML Databases, Elliotte Rusty Harold, August 13, 2007
[21] Comparing XML database approaches

