IADIS International Journal on Computer Science and Information Systems
Vol. 4, No. 2, pp. 41-58
ISSN: 1646-3692

ADAPTIVE STORAGE MODEL FOR XML IN OBJECT-RELATIONAL DATABASES

Michael Kamel. *Dept. of Computer and Systems Engineering, Faculty of Engineering, Alexandria University, Egypt. Elshatby, Alexandria, Egypt.*

mkamel@alex.edu.eg

Khaled Nagi. *Dept. of Computer and Systems Engineering, Faculty of Engineering, Alexandria University, Egypt. Elshatby, Alexandria, Egypt.*

knagi@alex.edu.eg

Nagwa El-Makky. *Dept. of Computer and Systems Engineering, Faculty of Engineering, Alexandria University, Egypt. Elshatby, Alexandria, Egypt.*

nagwamakky@alex.edu.eg

ABSTRACT

Object relational database management systems (ORDBMS) are becoming more popular in storing and retrieving XML than native XML DBMS. In most ORDBMS, XML is stored as CLOB inside the relation. Efficient XML parsers and indexing techniques are used to retrieve the desired XML nodes. However, less attention is given to XML updates queries. With the upcoming standardization of XML updates queries, the current implementation of the lock granularity imposes a great limitation on the concurrency of parallel transactions. This motivated several experimental ORDBMS to shred the XML nodes across internal relations. This approach has also several drawbacks. In this paper, we propose an adaptive technique for selective shredding. It is based on existing database engines and takes the changes in the workload pattern into consideration. We analyze the performance of our approach and compare it to the CLOB and the complete shredding approaches.

KEYWORDS

XML storage models, data shredding techniques, ORDBMS, performance analysis.

1. INTRODUCTION

XQuery is becoming the standard query language for querying XML data. Currently, most commercial Object Relational Database Management Systems (ORDBMS) treat the whole XML document as a single text attribute in the relation. They all implement row-level or page locking techniques. Due to the non-exclusive (shared) locks of read operations, XQuery operations retrieving data from XML never impose a performance problem with the increase of concurrent users in a database management system. The whole row containing the XML in question is locked for reading in a shared mode and efficient XML parsers are used to retrieve the desired XML nodes.

However, with the upcoming standardization of update queries in the XQuery language (Chamberlin & Robie 2005) the current implementation of the lock granularity imposes a great limitation on the concurrency of parallel transactions. To overcome this problem, several experimental ORDBMS completely shred the XML nodes across internal relations to achieve more concurrency. The main drawbacks of this approach are *the huge space consumption used to store meta-data of the shredded XML nodes and due to fragmentation* and *the degradation of response time of XQueries accessing large XML Sub-trees (not just single nodes) which require an extra overhead to rebuild the XML tree from the shredded nodes*.

The current work presents a new XML storage approach called *Selective XML Shredding* in which the XML document is gradually shredded into smaller XML portions to achieve higher concurrency for XQuery updates. The XML sub-trees that are frequently accessed will be stored into separate XML portions in a separate relation which helps to achieve higher concurrency of access on these sub-trees. This approach tries to save the huge space used by the complete shredding schemes and meanwhile achieve higher concurrency than CLOB based schemes. The selective XML shredding is designed to perform better than complete XML shredding for operations on XML Sub-trees as it saves the overhead required by the complete XML shredding scheme to rebuild the accessed XML Sub-tree from shredded nodes. The approach has another important advantage of being adaptive. The scheme shreds the portions of XML that are heavily accessed, in so called hot spot areas. If the hot spot area changes its location, the scheme gradually consolidates the fragmented XML portions that are no longer heavily used and shreds those portions in the new area.

In our design, we undergo an important constraint, which is simply ignored in the experimental ORDBMS. We do not attempt to change the internal storage management of the database engine. We build the shredding scheme as an isolated layer on top of existing commercial ORDBMS. This makes our approach more ready to use than others.

The rest of the paper is organized as follows. Section 2 presents a background on existing storage models. In Section 3, we present our proposed storage model and briefly describe its implementation in Section 4. Section 5 contains a brief validation and verification of the system. Section 6 describes the simulation model used to analyze our approach and compare it to the standard CLOB based storage model and the complete shredding storage model. The experiment results are presented in Section 7. Section 8 concludes the paper.

2. BACKGROUND

There are several schemes used to store XML data in ORDBMS which can be categorized as follows (Nicola & van der Linden 2005). Schemes used to store XML data natively are not considered in this paper.

2.1 Storing XML as a Single Field

Generally, storing XML as a single field (generally CLOB) allows for fast insertion and retrieval of full documents but suffers from poor search and extraction performance due to XML parsing at query execution time. This can be moderately improved if indexes are built at insert time. While this incurs XML parsing overhead, it may speed up queries that look for documents which match given search conditions. Yet, extraction of document fragments and sub-tree level updates still require expensive XML parsing. In Oracle 10g XML documents can be stored with indexing support as CLOBs or shredded to object-relational tables. Microsoft SQL Server 2005 stores XML documents as byte sequences in CLOB columns as mentioned in (Pat et al. 2004). A primary XML index can be defined to avoid parsing the XML CLOBs at query time (Pat et al. 2004).

2.2 Shredding XML to a Relational Schema

Shredding XML to a relational schema is the process of mapping XML elements into relational data based on the tree representation of the XML document. Shredding XML to relational tables is expensive at insert time due to costly XML parsing and multi-table inserts (Nicola et al. 2003). But once XML is broken into relational scalar values, queries and updates in plain SQL promise higher performance. XML Shredding can be categorized into two main categories:

- *Schema-based XML Storage*: It depends on storing XML in relational systems that make use of a schema for the XML data in order to choose a good relational schema.
- *Schema-oblivious XML Storage*: Its goal is to find a relational schema that works for storing XML documents independent of the presence or absence of a schema.

Our work focuses on XML documents that do *not* necessarily have a schema.

In STORED (Deutsch et al. 1999), given a semi-structured database instance, a special mapping is generated automatically using data mining techniques. STORED is a declarative query language proposed for this purpose. This mapping has two parts: a relational schema and an overflow graph for the data not conforming to the relational schema. STORED can be classified as a schema-oblivious technique since the data inserted is not required to conform to the derived schema.

According to the edge approach, the input XML document is viewed as a graph and each edge of the graph is represented as a tuple in a single table. In a variant known as the attribute approach, the edge table is horizontally partitioned on the tag name yielding a separate table for each element/attribute. Two other alternatives, the Universal table approach and the Normalized Universal approach are proposed but shown to be inferior to the other two.

The binary association approach (Schmidt et al. 2000) is a path-based approach that stores all elements that correspond to a given root-to-leaf path together in a single relation. Parent-child relationships are maintained through parent and child ids. The XRel approach

(Yashikawa et al. 2001) is another path-based approach. The main difference here is that for each element, the path id corresponding to the root-to-leaf path as well as an interval representing the region covered by the element is stored. The latter is similar to interval-based schemes for representing inverted lists proposed in (Li & Moon 2001) and (Zhang et al. 2001).

In (Tatarinov 2002), the focus is on supporting order based queries over XML data. The schema assumed is a modified edge relation where the path id is stored as in (Yashikawa et al. 2001), and an extra field for order is also stored. In (DeHaan et al. 2003), all XML data is stored in a single table containing a tuple for each element, attribute and text node. For an element, the element name and an interval representing the region covered by the element is stored. Analogous information is stored for attributes and text nodes.

There has been extensive work on using inverted lists to evaluate path expression queries by performing containment joins (e.g., Jiang et al. 2003, Li & Moon 2001, Srivastava et al. 2002, Wang et al. 2003, and Zhang et al. 2001). In (Zhang et al. 2001), the performance of containment algorithms in an RDBMS and a native XML system are compared. All other strategies are for native XML systems. In order to adapt these inside a relational engine, it is supposed to add new containment algorithms and novel data structures. The issue of how the relational engine is extended to identify the use of these strategies is open. In particular, the question of how the optimizer maps SQL operations into these strategies needs to be addressed.

In (Teubner 2003), a new database index structure called the XPath accelerator is proposed that supports all XPath axes. The pre-order and post-order ranks of an element are used to map nodes onto a two-dimensional plane. The evaluation of the XPath axis steps then reduces to processing region queries in this pre/post plane. In (Teubner 2003), the focus is on exploiting additional properties of the pre/post plane to speedup XPath query evaluation and the Staircase join operator is proposed for this purpose. The focus of (Grust 2002) and (Teubner 2003) is on efficiently supporting the basic operations in a path expression and is complementary to the XML-to-SQL query translation issue.

3. PROPOSED STORAGE MODEL

In our work, we focus on schema oblivious XML storage for ORDBMS. We do not attempt to change the underlying database storage manager. Our solution is built on top any existing storage manager. This implies, as well, that we do not change any interface to the lock manger residing above the storage manager.

Figure 1 illustrates a general layered architecture of an ORDBMS. We introduce a component that maps XQuery operations to SQL statements: the *XQuery To SQL Translator*. The XML *Storage Mapper* is tightly coupled to the *XQuery To SQL Translator* in order to map the operation on XML nodes to database operations performed on the corresponding database tuples. The *XML Access Monitor* examines the tuples being accessed either through read or write operations. Its job is to identify the hot spots, which are XPathes with lots of read and write operations in order to apply the selective shredding algorithm.

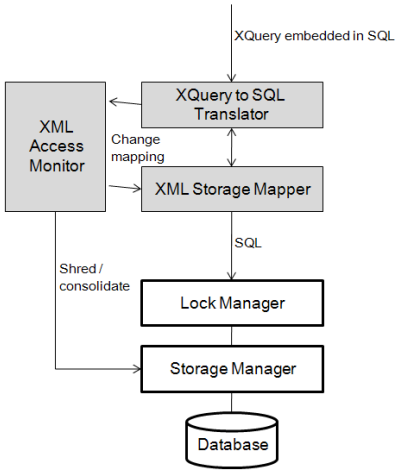


Figure 1. Proposed system components

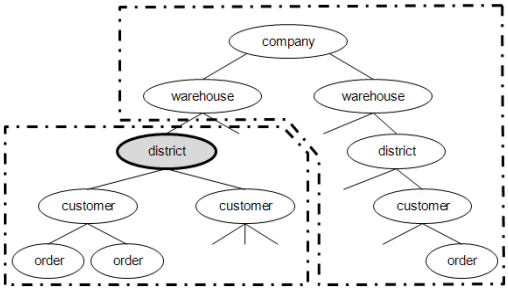
3.1 Selective Shredding

Selective Shredding based storage of XML documents means that the XML document is gradually shredded into smaller XML portions to achieve higher concurrency for XQuery updates. Using a sliding window concept to evaluate the frequency of access, the XML sub-trees that are frequently accessed are gradually stored into separate XML portions. There are two main parameters that control the shredding phase.

- Time interval *dt*: It is the time between two successive cycles of shredding and consolidation.
- Frequency threshold of XQuery operations on a certain XPath: It is the threshold of number of XQuery operations that access the same XPath during time interval *dt*. If this threshold is exceeded at a certain XPath, selective shredding takes place at this XPath.

When the access on the shredded XML portions decreases, the XML Access Monitor issues a command to *consolidate* these portions back into a greater XML document or portion. The following example explains our scheme.

Let us assume a sample XML document of the TPC-C benchmarking model, illustrated in Figure 2. The node *district* shaded below in the XML document experiences frequent XQuery updates. The XML document will be shredded into two portions at the node being frequently updated; i.e., the *district* node. Each of the two XML portions is stored into a separate XML CLOB field in order to achieve higher concurrency for XQuery updates as illustrated in Figure 2.




```
<district id="A-001">
  <customer id="M-
0023">
   <order id="1">
   ...
   </order>
   <order id="2">
   ...
   </order>
   ...
  </customer>
  <customer id="M-
0024">
  ...
  </customer>
</district>
```

```
<company>
  <warehouse id="A">
 <!--Node Place Holder-->
    <district id="A-002">
     ...
    </district>
    ...
  </warehouse>
  <warehouse id="B">
    <district id="B-001">
      <customer id="K-
0024">
        <order id="1">
        ...
        </order>
        ...
      </customer>
    </district>
     ...
  </warehouse>
  ...
</company>
```

Figure 2. XML after selective shredding

4. IMPLEMENTATION MODEL

In our implementation model, we use the standard edition of MS SQL Server 2005 in order to verify our constraint of being ready to run on existing ORDBMS without changing its internal. We define a user defined data type, *myXMLType*. We implement a simplified XML converter similar to (Yashikawa et al. 2001) and (Tatarinov 2002), which translates XPath expressions in XQuery which are in turn embedded in SQL to plain SQL using stored procedures for insertion, update, deletion and retrieval. The algorithms used in these stored procedures depend on the underlying storage model.

In order to keep our implementation as simple as possible, we assume – without loss of generality – that the lock manager uses the standard Two Phase Locking scheme on row level for the relational data. On the XML level, we assume the Path Lock Propagation scheme (Dekeyser & Hidders 2002) since it is one of the best in time metric. While mapping to the relational model, we make sure that the same logical locks on XML nodes are held by the lock manager using the standard two phase locking scheme.

4.1 CLOB Based Storage

This is the standard approach used by most of the commercial ORDBMS. The whole XML document is stored as a single attribute in the relation. A typical schema looks as in Table 1.

Table 1. CLOB based storage model

PK	Name	XML_Data
4711	ACME	XMLdocument1 as CLOB
4712	Global Inc.	XMLdocument2 as CLOB

4.2 Complete Shredding Based Storage

The data stores in the XML complete shredding based system are:

- XML Relation: This relation is the original relation that is supposed to store the XML data but what actually is done is to store an XML document identifier instead of storing the whole XML document as a CLOB. The identifier for the XML document is used in nodes relation to relate the XML nodes to their original XML document using a foreign key constraint.
- Nodes Relation: This relation is used to store the data of XML nodes resulting from the tree representation of each XML document in XML relation. It is related to XML relation by XML document identifier. This relation is created to achieve concurrency at XML node level instead of being at the whole XML document.

A typical schema looks as in Table 2. The attributes of the *Nodes* relation are listed in Table 3.

Table 2. Complete shredding based storage model

PK	Name	MyXML_Column_ID
4711	ACME	XML00000001
4712	Global Inc.	XML00000002

Table 3. Nodes relation for the complete shredding based storage model

Attribute	Description
PK	Node unique identifier
Type	node type (element or attribute or text or comment, etc.).
Value	node value For nodes of type element, it is NULL. For nodes of type attribute or text or comment, it stores the contents of the node.
XPath	XPath of the XML node (tokenized)
docId	XML document identifier which refers to the identifier of the XML document in the base table.
ParentId	Parent node identifier of the current node (foreign key to PK)

4.3 Selective Shredding Based Storage

The data stores in the XML selective shredding based system are:

- XML Relation: same as in the complete shredding storage model
- XML Portions Relation: This relation is used to store the data of XML portions resulting from the shredding of frequently updated sub-trees in the tree representation of each XML document in the XML relation. It is related to the XML relation by the

XML document identifier. This relation is created to achieve concurrency at XML portion level instead of being at the whole XML document.

The attributes of the XML Portions Relation are listed in Table 4.

Table 4. XML portion relation for the selective shredding based storage model

Attribute	Description
XMLDocID	XML document identifier
PortionID	XML portion identifier
XPathOfRootForXMLPortion	XPath of root for current XML Portion
XMLPortion	XML Portion document
ParentPortionID	Parent XML document for current XML portion

Additionally, the XML access monitor logs the XQuery operations in a volatile log to be used for taking the decision whether to shred or consolidate an XML portion. The log contains the XML document identifier, the operation type, the XPath used in the XQuery, and the timestamp of the XQuery.

5. VALIDATION USING EXAMPLES

The proposed model and its implementation were verified by running several samples runs using both typical and boundary values. In this paper, we show examples using simple values for illustration purposes.

5.1 Sample Insertion

Consider the following Insert query:

```
Insert into TPC(CompanyID, CompanyName, [TPC XML]) Values(1,
‘‘Buckland Stores’’,
‘<company>
      <warehouse id="A">
            <district id="B-001">
                  <customer id="M-0023" index="M">
                        <name> Michael </name>
                        <order id="1">
                              <item> HB pencil </item>
                              <price> 15 </price>
                              <num> 12 </num>
                              <status> undelivered </status>
                        </order>
                  </customer>
            </district>
      </warehouse>
</company>’)
```

For CLOB-based scheme, the XML document is stored directly as a single field. For Complete Shredding-based scheme, this insertion is performed using the constructor of the User Defined Type “MyXMLDataType”. The XML is validated and the system traverses the

XML tree and stores the nodes in the "Nodes Table" and also maintains the parent-child relationship between XML nodes using foreign key constraints as illustrated in Table 5. It then stores a row in the relational table containing the XML column.

Table 5. Nodes Table after insertion

Id	TagName	TId	Value	HId	Pk1	ParentId
1	company	1	NULL	company	1	NULL
2	warehouse	1	NULL	company/warehouse	1	1
3	id	2	A	company/warehouse/id	1	2
4	district	1	NULL	company/warehouse/district	1	2
5	id	2	B-001	company/warehouse/district/id	1	4
6	customer	1	NULL	company/warehouse/district/customer	1	4
7	id	2	M-0023	company/warehouse/district/customer/id	1	6
8	index	2	M	company/warehouse/district/customer/index	1	6
9	name	1	NULL	company/warehouse/district/customer/name	1	6
10	NULL	4	Michael	company/warehouse/district/customer/name/#text	1	9
11	order	1	NULL	company/warehouse/district/customer/order	1	6
12	id	2	1	company/warehouse/district/customer/order/id	1	11
13	item	1	NULL	company/warehouse/district/customer/order/item	1	11
14	NULL	4	HB pencil	company/warehouse/district/customer/order/item/#text	1	13
15	price	1	NULL	company/warehouse/district/customer/order/price	1	11
16	NULL	4	15	company/warehouse/district/customer/order/price/#text	1	15
17	num	1	NULL	company/warehouse/district/customer/order/num	1	11
18	NULL	4	12	company/warehouse/district/customer/order/num/#text	1	17
19	status	1	NULL	company/warehouse/district/customer/order/status	1	11
20	NULL	4	undelivered	company/warehouse/district/customer/order/status/#text	1	19

For Selective Shredding-based scheme, the system validates the XML document. If the XML is valid, it traverses the XML tree and it initially stores the whole XML document as a single field in the "XML Portions Table". It also stores a row in the relational table containing the XML column. Gradually, the XML document is shredded into smaller portions at the XPaths being frequently accessed. During each shredding process, the XML Portions table is modified by deleting the large XML document and inserting two smaller portions instead as illustrated in Table 6.

Table 6. Portions tables after shredding

XMLDocID	PortionID	XPathOfRootForXMLPortion	XMLPortion	ParentPortionID
1	1	NULL	[XML Portion 1]	1
1	2	company/warehouse/district	[XML Portion 2]	2

5.2 Sample Update XQueries

5.2.1 Insertion

```
let $x := /company/warehouse[@id="A"]/district[@id="B-001"]
do insert $x
<customer id="D-144">
        <name> David </name>
```

```
        <entry_date> 12/02/2002 </entry_date>
 </customer>
```

Steps for mapping XQuery for Selective Shredding Scheme

- XQuery Handler locates the XML Portion that includes the XPath of the node being accessed from the XML Portions table. Let us say the XML Portion being accessed starts with node called “district” as a root node.
- XQuery Handler maps the XPath of the XQuery transaction which is “/Company/Warehouse[@id="A"]/district[@id=”B-001”]” to a corresponding XPath of the XML Portion that includes the node being accessed which is “district[@id=”B-001”]”. The mapped XQuery is

```
let $x := district[@id="B-001"]
do insert $x
<customer id="D-144">
        <name> David </name>
        <entry_date> 12/02/2002 </entry_date>
 </customer>
```

5.2.2 Modification

```
let $x0 := /company/warehouse[@id="B"],
$x1 := $x0/district[@id="D-002"]/customer[@id="C-031"],
$x := $x1/order[@id="5"]/num
do replace value of $x with “10”
```

Steps for mapping XQuery for Selective Shredding Scheme

- XQuery Handler locates the XML Portion that includes the XPath of the node being accessed from the XML Portions table. Let us say the XML Portion being accessed starts with node called “district” as a root node.
- XQuery Handler maps the XPath of the XQuery transaction which is “/Company/Warehouse[@id="B"]/district[@id=”D-002”]” to a corresponding XPath of the XML Portion that includes the node being accessed which is “district[@id=”D-002”]”. The mapped XQuery is:

```
let $x1 := district[@id="D-002"]/customer[@id="C-031"],
$x := $x1/order[@id="5"]/num
do replace value of $x with “10”
```

5.2.3 Deletion

```
let $x0 := /company/warehouse[@id="B"],
$x := $x0/district[@id="D-002"]/customer[@id="C-031"],
$y := $x/order[date="19/02/2002"]
delete $y
```

Steps for mapping XQuery for Selective Shredding Scheme

- XQuery Handler locates the XML Portion that includes the XPath of the node being accessed from the XML Portions table. Let us say the XML Portion being accessed starts with node called “district” as a root node.
- XQuery Handler maps the XPath of the XQuery transaction which is “/Company/Warehouse[@id="B"]/district[@id=”D-002”]” to a corresponding XPath

of the XML Portion that includes the node being accessed which is “district[@id=”D-002”]”. The mapped XQuery is

```
let $x := district[@id="D-002"]/customer[@id="C-031"],
$y := $x/order[date="19/02/2002"]
delete $y
```

6. SIMULATION MODEL

In order to evaluate the performance of our proposed storage model, we build a simulator based on MS SQL Server 2005 as illustrated in Figure 3. We implement a simplified version of all three storage models: *CLOB-based* storage model, *complete XML shredding* model, and the *selective XML shredding* model. The parameters that control the data stored in these data stores include the *number of XML documents*, the *number of levels in each XML document* and *the average number of siblings* for each node. The default is 10,000 documents of depth 3 and average number of siblings 5. In other words, the test database contains about 1,250,000 XML nodes. We use this relatively small number of nodes in order to artificially increase the rate of conflicts and thus stress-test the three approaches.

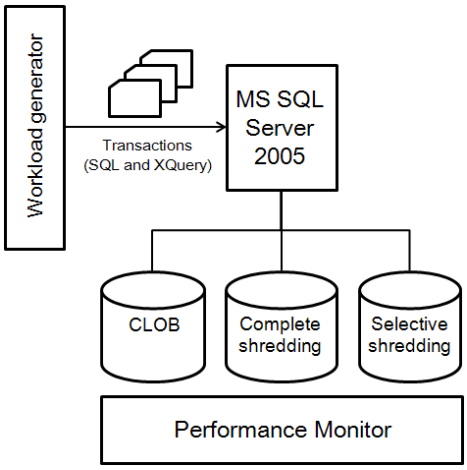


Figure 3. Simulation model

The *workload generator* submits database transactions using XQuery embedded in SQL. Each transaction executes in a separate thread. We launch up to 250 transactions in parallel to simulate 250 *concurrent users*. The *complexity* of a single transaction varies from 1 to 10 database statements. Transactions can be *read-only* containing SELECT and XQuery retrieval operations only or can be *read-write* containing UPDATE and XQuery insertion, modification, or deletion operations. We have two types of *selectivity factors*. The first one is the selectivity of database tuples. Traditionally, it does not exceed 10%. The second factor is inside the XML document itself. It determines the level of the parent node of all nodes accessed by the XQuery statement and accordingly the percentage of its siblings that are being affected by the statement. This percentage can vary from 0% to 100% in real life. In order to test the adaptive nature of each storage model, we artificially create *hot spots* by concentrating

the access to XML nodes to one small subset of the existing nodes. Periodically, we switch to another subset to simulate changes in the hot spot areas over time.

The *performance monitor* measures the overall system *throughput* in terms of committed transactions per second; the average *response time* for all types of transactions; the *space consumption* on the disk, and the percentage of *aborted* transactions, which is a direct indication of the number and severity of conflicts in access to the same document part.

7. EXPERIMENT RESULTS

7.1 Adaptive Nature of the Storage Models

In this set of experiments, we investigate the effect of changing the hot spot access areas of XML nodes over time in the three storage models. The hot spot area is changed periodically and the transient behavior of each storage model is plotted. Figures 4 and 5 show that the system throughput and the response time of the complete shredding and CLOB based storage model are slightly affected by the change in the hot spot area. This is probably due to diverse caching mechanisms. However, the proposed selective shredding has a much better performance as it selectively begins to shred the hot spot. With the shift in the hot spot (the graphs illustrate three shifts in the hot spot) the improvement in the performance measures diminishes till the next consolidation and shredding phase. In all cases, the percentage of aborted transactions remains insignificant. Thus, the transient behavior of the system demonstrates the adaptive nature of the suggested solution.

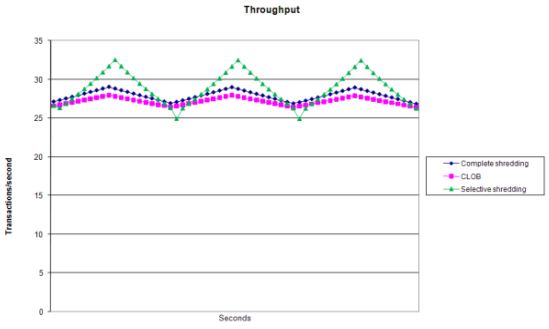


Figure 4. Moving average system throughput

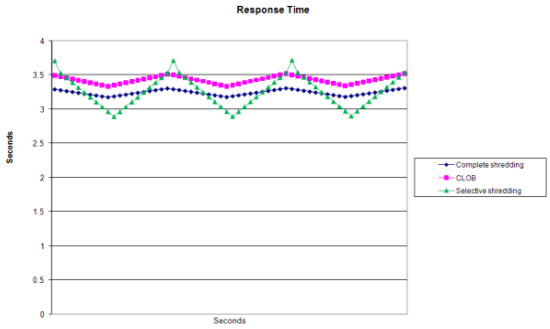


Figure 5. Moving average response time

7.2 Effect of Increasing the Number of Concurrent Users

In this set of experiments, the number of concurrent users submitting transaction is varied from 50 to 250. With this huge number of parallel transactions the possibility of lock conflicts increases dramatically. Moreover, the artificially small database size magnifies the rate of conflicts.

As expected, complete shredding with its fine grained locks performs better than the other storage models. CLOB-based and selective shredding storage model perform similarly. Their system throughput saturates at 40 transactions/second, as illustrated in Figure 6, whereas the complete shredding system seems to scale linearly even at 250 concurrent users achieving a throughput of 80 transactions/second. The same applies to the response illustrated in Figure 7. The response time of CLOB and selective shredding climb to 6 seconds whereas complete shredding remains at 3 seconds. The abort rate of CLOB and selective shredding remains below 12% which is acceptable. Not a single abort is observed in the complete shredding model due to the fine granularity of its locks.

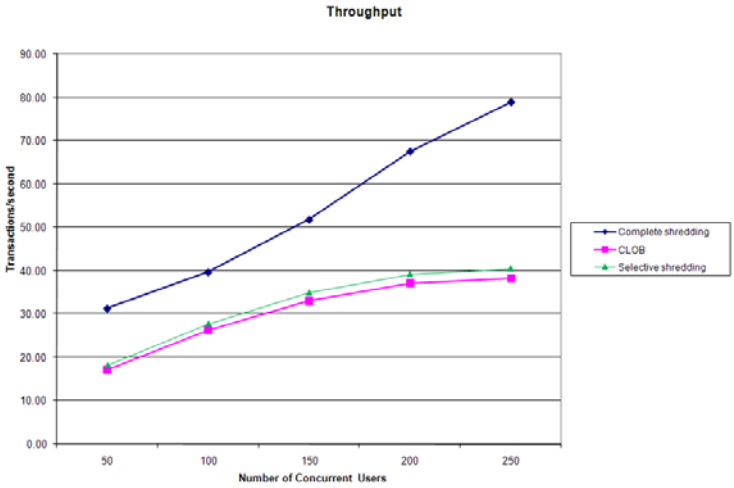


Figure 6. Throughput vs. number of concurrent users

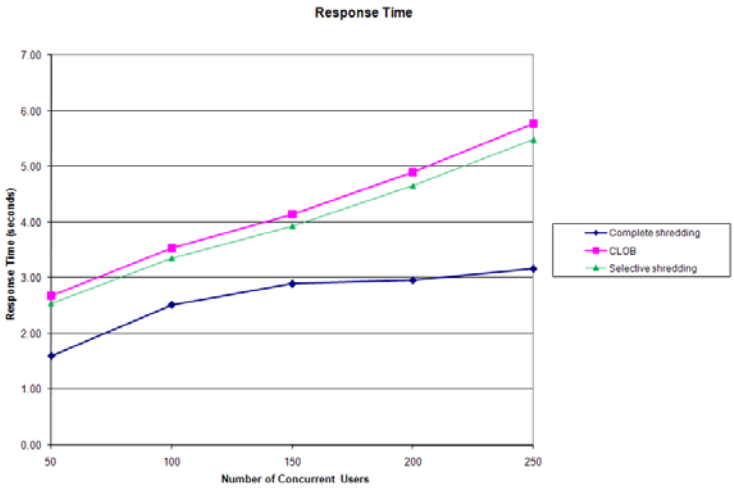


Figure 7. Response time vs. number of concurrent users

7.3 Space Consumption

The great performance of selective complete shredding has certainly its costs on the space consumption. In this set of experiments, we increase the number of XML nodes in the system and observe the space consumption of all three storage models. The increase can be done by either increasing the number of XML documents, their level, or the average number of their siblings. All three factors are applied and all yield similar results. Here, we show only the space consumption as a function of the number of XML documents. The selective shredding and CLOB go side by side with the increase in XML documents; whereas the complete shredding consumes huge among of storage due to fragmentation as illustrated in Figure 8. By increasing the number of XML documents by a factor of 10, the space consumed by the complete shredding increases by a factor of 10 and amounts to 2.5 GB. The selective shredding remains under 1.2 GB and CLOB storage under 750 MB. The throughput and response degrade gracefully in all three storage models. The rate of aborted transactions remains in a save area.

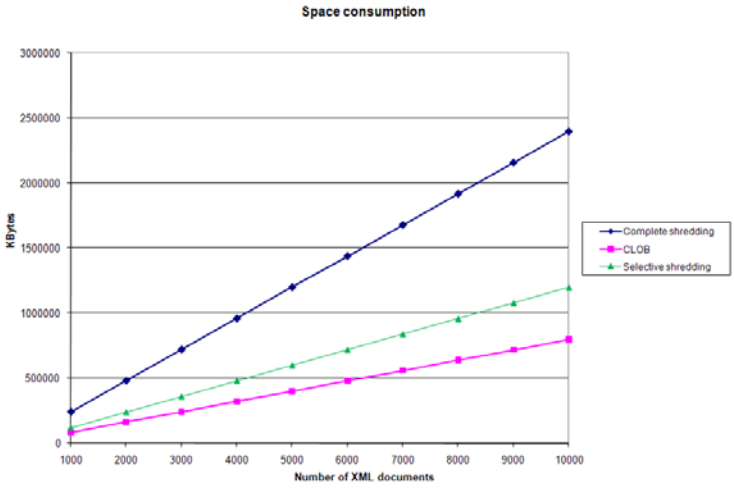


Figure 8. Space consumption vs. number of XML documents

7.4 Analysis of XQuery Update Operations

Figure 9 and 10 show the effect of the different XQuery update operations on the throughput and response time respectively. During XQuery modification and deletion, it is just the value of an XML node(s) that will be affected (updated or deleted). This makes the performance of the complete shredding superior to the other two models, since it is always faster to update or delete the relational data in the nodes tables than to update a CLOB field.

As for the XQuery insertion, it is required for the complete shredding based system to map the inserted XML nodes into relational data as well as to relate the new nodes to their parent nodes in the nodes tables. This is a cost intensive operation. In the CLOB based and selective shredding based systems, a CLOB field is updated with the newly inserted nodes; which is a

much faster operation. The percentage of transaction aborts remains reasonably low for all XQuery update operations in all three storage models.

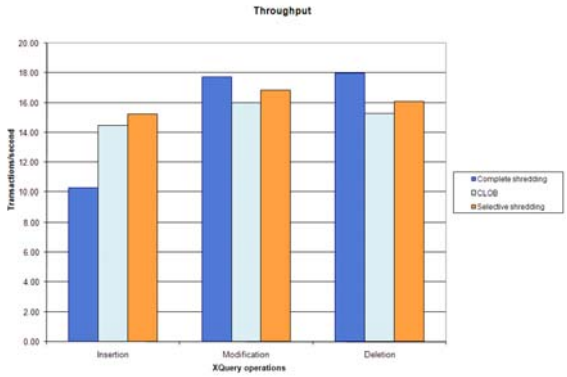


Figure 9. Throughput of XQuery update operations

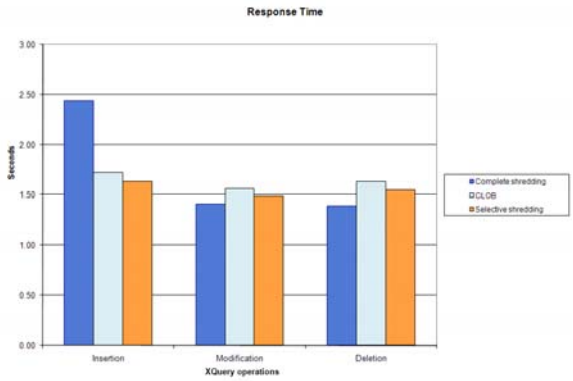


Figure 10. Response time of XQuery operations

7.5 Analysis of XQuery Retrieval Operations

Since retrieval operations usually constitute 80% of the total number of operations, we analyze its performance under different types of queries. In the relational part governing which XML documents are in question for the XQuery operation, we assume a selectivity factor of 10%; which is normal for typical relational database retrieval queries. Inside the XML documents, the selectivity factor of the XML nodes varies heavily. We examine the whole spectrum from 0% to 100%. Moreover, the performance of the system depends on the depth of the XPath. In this paper, we show the throughput and response time in case that the XPath matches level 2 and 3 of the whole XML document. In each case, the percentage of select XML nodes is varied from 0% to 100%.

In Figures 11 and 12, we illustrate the throughout and response time for XPath accessing nodes at level 2 respectively. Here, it is clear that the CLOB-based storage model outperforms the complete shredding storage based model. This is due to the fact that the chosen sub-trees are near to the root of the original document and are relatively deep. This makes their reconstruction from deeply fragmented nodes a cost intensive operation. The selective shredding storage based model comes slightly after CLOB.

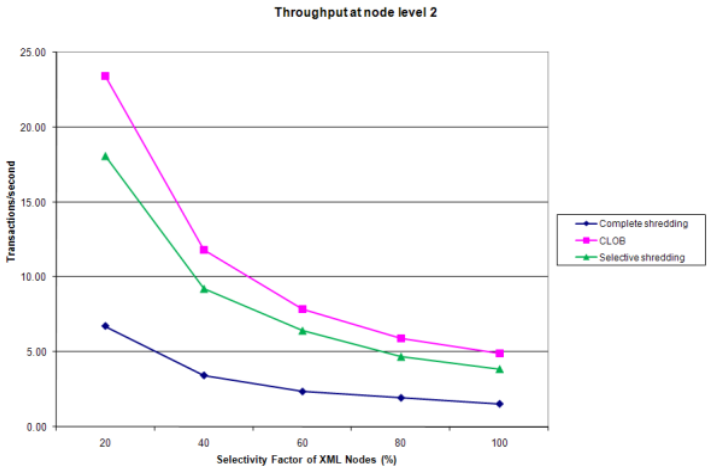


Figure 11. Throughput of XQuery retrieval operations vs. the selectivity factor of XML nodes at level 2

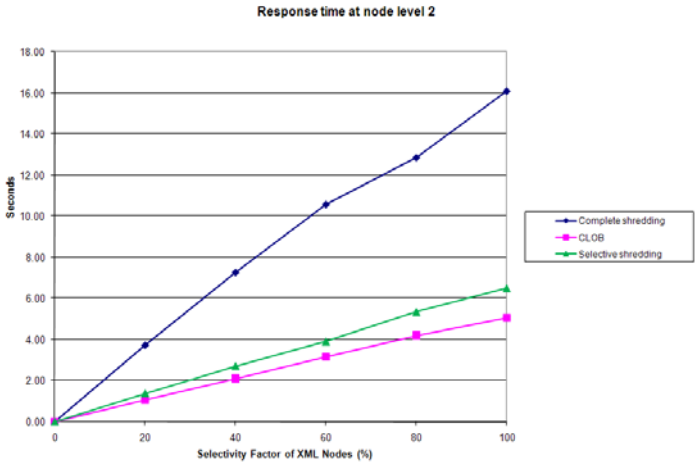


Figure 12. Response time of XQuery retrieval operations vs. the selectivity factor of XML nodes at level 2

In Figures 13 and 14, we illustrate the throughout and response time for XPath accessing nodes at level 3 respectively. Here, we get the opposite results. The complete shredding storage based model outperforms the CLOB based storage model. This is due to the fact that the chosen sub-trees are near to the leaf nodes of the original document and are hence not expensive to reconstruct from fragmented single nodes. Here also, the performance of our selective shredding storage based model lies between both standard approaches.

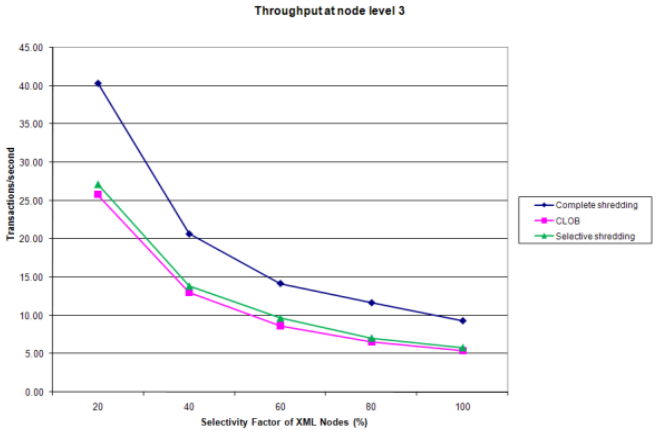


Figure 13. Throughput of XQuery retrieval operations vs. the selectivity factor of XML nodes at level 3

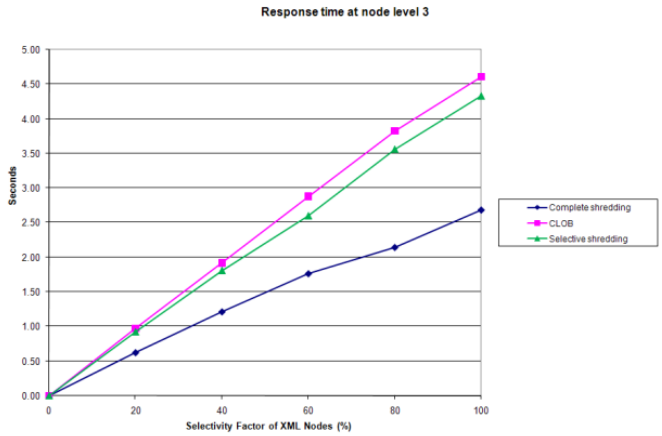


Figure 14. Response time of XQuery retrieval operations vs. the selectivity factor of XML nodes at level 3

8. CONCLUSION

In this paper, we present the *Selective XML Shredding* storage scheme for XML in ORDBMS. It is a mix of CLOB based and XML Shredding based storage models. The main objective of this scheme is to increase concurrency of XQuery operations on XML documents by splitting them into smaller XML portions and store these portions in a separate relation. Our approach is built as a layer on top existing DBMS systems.

We build a prototype of the existing storage schemes and compare the throughput, response time, space consumption and the ratio of aborted transaction to our scheme. The simulation results show that the XML Shredding based system has higher throughput than the

CLOB based system when the number of concurrent users performing XQuery updates increases in the system but the main drawback is the extra storage used to store XML nodes. The depth of the XML nodes being accessed in XPath has the main effect on differentiating the competitive three approaches. Selective Shredding based system is the best on intermediate depths as it is a hybrid approach of the CLOB based and Complete Shredding. Being adaptive against the change in the workload pattern, our approach promises the best compromise between the existing approaches.

REFERENCES

Chamberlin, D and Robie J. 2005. W3C XQuery Update Facility Requirements, W3C Working Draft, *http://www.w3.org/TR/2005/WD-xquery-update-requirements-20050603/.*

DeHaan, D.et al. 2003. A Comprehensive XQuery to SQL Translation using Dynamic Interval Encoding. *Proc. of SIGMOD.*

Dekeyser, S. and Hidders, J. 2002. Path Locks for XML Document Collaboration. *Proc. of WISE.*

Deutsch, A. et al. 1999. Storing semi structured data with STORED. *Proc. of SIGMOD.*

Grust, T. 2002. Accelerating XPath location steps. *Proc. of SIGMOD.*

Jiang, H. et al. 2003. XR-Tree: Indexing XML Data for Efficient Structural Joins. *Proc. of ICDE.*

Li, Q. and Moon, B. 2001. Indexing and querying XML data for regular path expressions. *Proc. of VLDB.*

Nicola, M. et al. 2003. XML Parsing, A Threat to Database Performance. *Proceedings of CIKM.*

Nicola, M. and van der Linden, B. 2005. Native XML Support in DB2 Universal Database. *IBM Silicon Valley Lab.*

Pat et al. 2004. Indexing XML Data Stored in a Relational Database. *Pro. of VLDB.*

Schmidt, A. et al. 2000. Efficient Relational Storage and Retrieval of XML Documents. *Proc. of webDB.*

Srivastava, D. et al. 2002. Structural Joins: A Primitive For Efficient XML Query Pattern Matching. *Proc. of ICDE.*

Tatarinov, I. et al. 2002. Storing and querying ordered XML using a relational database system. *Proc. of SIGMOD.*

Teubner, J. 2003. Staircase Join: Teach a Relational DBMS to Watch its (Axis) Steps. *Proc. of VLDB.*

Wang, W. et al. 2003. PBiTree Coding and Efficient Processing of Containment Joins. *Proc. of ICDE.*

Yoshikawa, M. et al. 2001. XRel: a path-based approach to storage and retrieval of XML documents using relational databases. *ACM Transactions on Internet Technology (TOIT),* 1(1):110-141.

Zhang, C. et al. 2001. On Supporting Containment Queries in Relational Database Management Systems. *Proc. of SIGMOD.*