OXDP & OXiP: the notion of objects for efficient large XML data queries

Norah Saleh Alghamdi*, Wenny Rahayu and Eric Pardede

Department of Computer Science and Computer Engineering,
La Trobe University,
Melbourne, VIC 3083, Australia
Email: nalghamdi@students.latrobe.edu.au
Email: w.rahayu@latrobe.edu.au
Email: E.Pardede@latrobe.edu.au
*Corresponding author

Abstract: Due to the rapid growth of XML representation for information exchange, XML databases have been widely adopted in a variety of applications. This paper presents two layers of optimisation for dealing with large XML databases: (1) OXDP (Object-Based Methodology for XML Data Partitioning) which has been developed to partition XML data efficiently and (2) OXiP (Object-Based XML Indexing for Partitions) which is an indexing and linking mechanism for partitioned data. OXDP provides optimal XML data partitioning based on an object's semantic features which improves XML data query performance. The OXiP method tokenises all rooted label paths and preserves the pathways within each XML object partition. The semantic-based data partition ultimately enhances the notion of a frequently accessed data subset which is an advantageous feature in our proposed methods to decrease the time to answer queries. Experimentally, OXDP and OXiP can achieve an order of magnitude performance improvement for querying XML data.

Keywords: XML database; semantic-based query processing; indexing; partitioning; optimisation; path query.

Reference to this paper should be made as follows: Alghamdi, N.S., Rahayu, W. and Pardede, E. (2012) 'OXDP & OXiP: the notion of objects for efficient large XML data queries', *Int. J. Grid and Utility Computing*, Vol. 3, Nos. 2/3, pp.112–125.

Biographical notes: Norah Saleh Alghamdi completed her Bachelor of Computer Science degree at Taif University, Taif, Saudi Arabia. She completed her Master of Computer Science degree from the Department of Computer Science and Computer Engineering at La Trobe University, Victoria, Australia, where she is currently pursuing her PhD. Her areas of interest are XML databases, query processing and query optimisation.

Wenny Rahayu is an Associate Professor in the Department of Computer Science and Computer Engineering, La Trobe University, Australia. She is currently the Director of Academic Studies and the Head of the Data Engineering and Knowledge Management Laboratory in the department. Her research areas cover a wide range of advanced database topics including XML databases, spatial and temporal databases and data warehousing, and the semantic web and ontology.

Eric Pardede is a Lecturer in the Department of Computer Science and Computer Engineering at La Trobe University, Australia. His current research areas include XML databases, database as a service and data management in social network applications.

1 Introduction

In multi-institutional, dynamic virtual organisations, grid computing plays an important role in solving problems and sharing coordinated resources. Recently, grid technology has become applicable as a computational grid or as a data grid. The concern of increasing the execution time of applications due to excessive computer processing cycles has been addressed by the computational grid (Zhang and Phillips, 2011). However, in the case of utilising a large scale of data, the data grid plays a significant role to sort out data management issues. Since XML has wide-ranging features to support global data representation and exchange over the web, most applications generate their data in XML format (Tusa et al., 2009). Consequently, XML data has expanded significantly even in the area of the data grid. Hence, continuing research on the efficient storage and querying of an enormous amount of XML data is required.

Since RDB fails to deliver all the necessary functions to efficiently store and query XML data, the Enabled XML Database (EXD) or Native XML Database has been proposed. Several query languages have been introduced to query XML data, including XPath (Berglund et al., 2011), Quilt (Chamberlin et al., 2000) and XQuery (Robie et al., 2010).

Partitioning relational data in RDB has been extensively investigated and employed to improve the performance of relational data processing. Many applications have gained benefits from this technique. The underlying concept of data partitioning is now required in the XML data management area. In order to optimise query execution time, this paper presents two layered methodologies for processing large XML documents more efficiently. The Object-Based Methodology for XML Data Partitioning (OXDP) aims to improve query performance through optimal XML data partitions based on the semantics of the objects. This method can be applicable in many different high-performance computing environments. For example, since it introduces a semantic concept in partitioning XML data into objects, this feature makes it appropriate in parallel database environments by allocating each processor with a partition. Due to each partition being disjointed with others, they can be processed in parallel and potentially reduce execution time.

Object-Based Methodology XML Indexing for Partitions (OXiP) exploits the semantics of OXDP in order to efficiently link the data partitions during a query execution of XML data. Although earlier works focused on the development of methodologies to index XML data, none addressed the goal of finding an efficient mechanism for linking partitioned data. It can be argued that in general, a common document-based XML indexing method can be used to index partitioned data whereby each partition is treated as a separate document, and the XML index is used to consolidate and index the data for querying purposes. However, the weakness is that the method does not utilise the knowledge gained from the object partitions for constructing XML data because it only treats them as separate components. In our proposed methodology, the object-based knowledge of OXDP is utilised and leveraged in OXiP-indexing development. OXiP does not merely link partitions together; it leverages the knowledge of the data partition path locations and makes the construction of linking more efficient.

In a previous work on partitioning XML data, we proposed the essential concept of Object-Based Partitioning (OBP) methodology and how XML queries gained improvement in their performance (Alghamdi, 2011). In this publication, we reintroduce the knowledge of objects in combination with XML index to solve a number of limitations in current indexing approaches with respect to leveraging the concept of objects in indexing and querying large XML data.

The key contributions of the work are summarised as follows:

- Although a variety of XML data query processing techniques have been proposed, to the best of our knowledge, none of them takes into consideration the semantic-based query workload which is undertaken in OXDP and OXiP.
- OXDP exploits and supports the hierarchical structure of XML data by introducing the notion of objects during the data partitioning. As a result, queries can access a certain object leading to a *reduction of the cost of traversing entire documents*.
- OXDP *can work independently* to serve specific applications. However, in this paper, we show a second layer of optimisation, OXiP, which is used in conjunction with OXDP to *consistently link objects for further speed*.
- *Significant search space reduction* during the OXiP mechanism by utilising the knowledge of OXDP leads to *high performance in query evaluation*. OXiP devises a pruning technique for irrelevant objects.

The remainder of this paper is organised as follows. A survey of related work is presented in Section 2. We present an overview of the system architecture in Section 3. In Section 4 we describe OXDP in detail. Thereafter, Section 5 describes OXiP. The experiments and evaluation are presented in Section 6 and we conclude our work in Section 7.

2 Related work

In this section, we first review previous research on XML data partitioning and then describe the prior work on indexing XML data. Shredding XML documents is a common method used in storing XML documents within a pure RDB. Shredding is an automatic mechanism to partition XML data into what RDB really understands, such as tables with rows and columns (McGovern et al., 2003). Various approaches have been developed for XML document partitioning for different purposes. Double lazy parser (2LP) is a technique that emulates physical pointers through partitioning XML documents into sub-trees (Farfán et al., 2009). Although navigation and parsing time is reduced, producing several large partitions is somewhat unacceptable since smaller and uniform partition size is more rational. In addition, XML Clustering (XC) takes advantage of XML navigational behaviour to direct its partitioning decision and uses dynamic programming over weight intervals determined by the ‘chunk_size’ parameter (Bordawekar and Shmueli, 2004). However, when the ‘chunk_size’ value is decreased and XC precision is increased, memory usage and runtime would also increase.

Kanne and Moerkotte (2006) proposes a technique that uses the sibling properties of the trees to segregate an XML tree. However, this approach failed to consider the effect of query workload on performance since the partitioning of XML data is purely based on sibling properties.

Of the state-of-the-art path indexing, *DataGuide* is considered a key revolution in XML indices. The main idea of DataGuide is to construct a path summary which indexes each distinct path to evaluate a single path query (Goldman and Widom, 1997). *T-index* is also a path index which selects a path based on specific templates based on the similarity of node pairs (Milo and Suciu, 1999; Cooper et al., 2001). *A(k) index* has been proposed to reduce the size of the XML index. It targets localised structural information about XML data based on bi-similarity on nodes related to paths of length *k* (Kaushik et al., 2002). *D*(*k*) *index* is an improvement of *A*(*k*) index and considers the query load (Chen et al., 2003). Like *D*(*k*) index, *M*(*k*)*-index* and *M**(*k*)*-index* (Hao and Yang, 2004) support dynamic indexing. It can be said that *M*(*k*)-index is a further optimised index of *D*(*k*)-index by not over-refining index nodes for irrelevant index or data nodes. Another workload-aware path index is *APEX* (Chung et al., 2002). It is an adaptive path index constructed by applying data-mining algorithms to mine frequently appearing paths in the query.

XISS is a node XML-indexing approach on a B+-tree designed to support regular path expressions (Li and Moon, 2001). The main notion of XISS is to decompose the query into several simple path expressions. Structural join algorithms then produce an intermediate result for each simple path expression which can be used in the subsequent stage of processing a query.

Sequence-based indexing has been proposed where each XML document and a twig pattern of a query are transformed into structure-encoded sequences. Subsequence matching is used to evaluate the query as in *ViST* (Wang et al., 2003) and *PRIX* (Rao and Moon, 2004) which eschew costly join operations by utilising tree structures as the basic unit of a query. Similar to PRIX and ViST, *LCS-TRIM* in Tatikonda et al. (2007) relies on a sequential encoding transformation and matching for both XML data and path queries.

The TwigX-Guide approach processes path queries by extending the existing path summary in DataGuide and region encoding in TwigStack structural algorithms. To process the path query, TwigX-Guide uses the CutMatchMergePath algorithm which adopts the decomposition–matching–merging approach (Haw and Lee, 2009). LTIX combines the Level-based Labelling Scheme LLS and the DataGuide path index. The new idea of LLS is essential to identify the node whose level specifies the most likely result of a query (Mohammad and Martin, 2010).

3 System architecture

The entire view of the system architecture is shown in Figure 1. Our system aims to improve the performance of querying XML data, consisting of two essential components that can work independently. The first component is OXDP which introduces a new notion of partitioning XML data. OXDP preserves XML data hierarchy structures logically by partitioning XML data into objects while enhancing its queries efficiency through optimisation and semantic improvement. The second component, OXiP, can be coupled with OXDP as a partition linking and indexing. It is adopted to increase the efficiency of the queries as an index method, in addition to its ability to link generated partitions.

Figure 1 The architecture of OXDP and OXiP

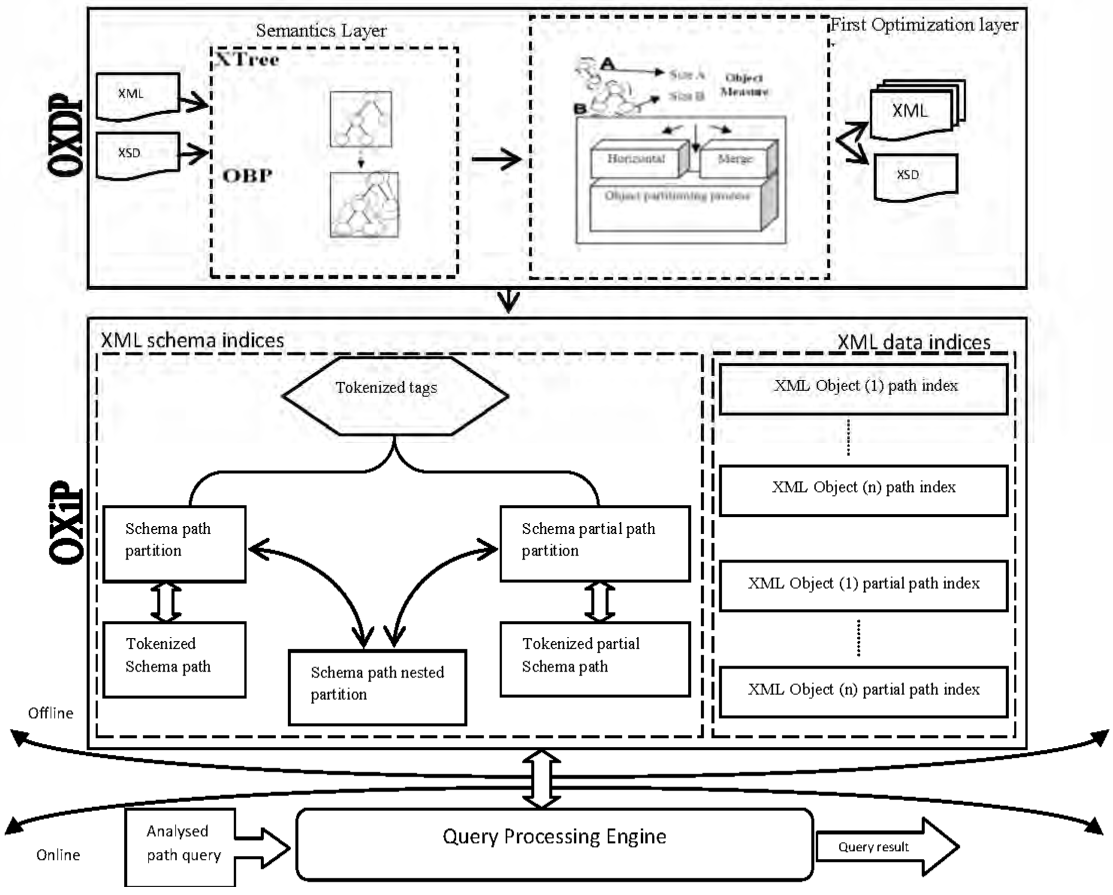
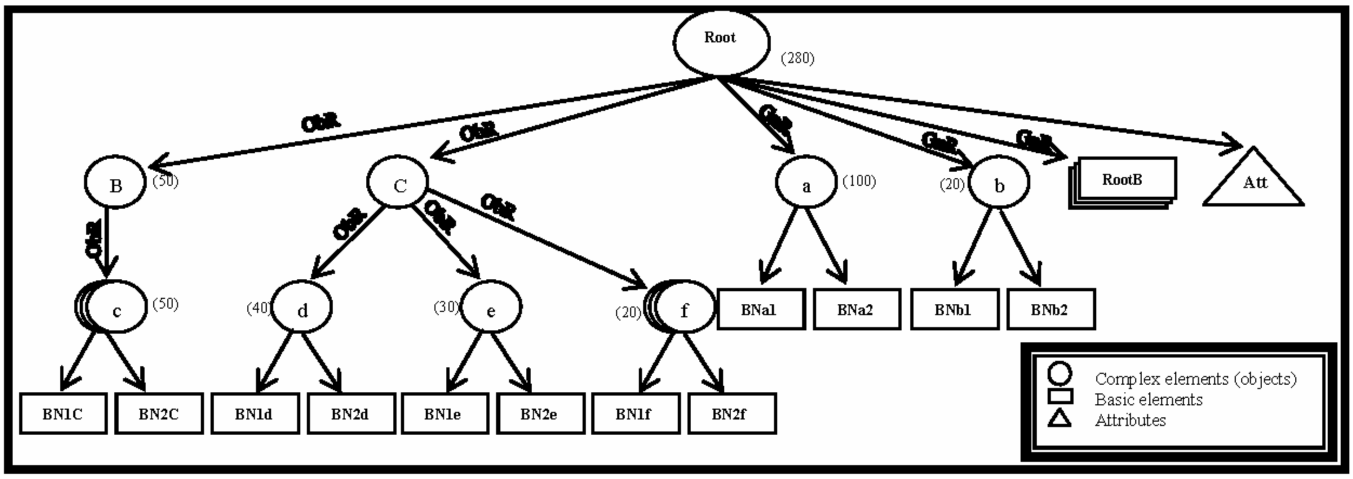


Figure 2 XTree sample



OXDP takes XML Entry, that is, an XML document and its corresponding XML schema (XSD), as input. XSD is used since it represents XML objects and their semantic structure. Hereafter, XML Entry runs through two layers: (1) the semantics layer and (2) the optimisation layer to be partitioned into objects which are to be passed to OXiP as a double optimisation layer to construct indices for joining the obtained objects. It is important to highlight that constructing OXDP as well as OXiP is considered an offline stage before the online phase when the actual query processing commences.

OXiP consists of two main parts: (1) XML schema indices, which are indices and meta-data created from an XML schema based on the knowledge of OXDP and (2) XML data indices, which produce indices for partitioned XML data. OXiP reduces excessive structural joins in query evaluations. The idea beyond OXiP is to utilise an XML document as the main memory structure where queries can be evaluated efficiently. All rooted label paths are tokenised, and their pathways are preserved within each XML object partition with far fewer nodes and edges; thus, the goal of increasing the query-processing efficiency becomes applicable for large amounts of XML data.

Since the main characteristic of OXDP is the semantic workload of queries, OXiP leveraged this advantageous feature during the construction of its indices. Semantic workload characteristics to partition XML data coupling with an efficient linking technique among partitions to processing queries over partitions have a significant impact on the performance of XML queries. From a semantic workload perspective, it is known that not all parts, called objects in this paper, of XML data are equal in ‘access rate’; some objects are more frequently used than others. Consequently, it can be obvious that the ‘access rate’ to some index nodes is highly likely to vary, because of their relativity to the position of the index structure. Therefore, the frequent access to the same object results in benefits in reducing the time to answer queries.

4 OXDP: object-based methodology for XML data partitioning

4.1 OXDP Layer 1: semantics layer

In this layer, XML data is semantically partitioned based on object-oriented (OO) principles. This process takes place before the actual partitioning process by (1) creating XTree from XML Entry and (2) setting and applying the OBP algorithm to identify the objects that are to be partitioned.

XTree (see Figure 2) is the first component of the Semantics layer in the OXDP section and is generated from XML Entry.

Definition 1 (XTree): *XTree is a labelled tree consisting of a set of nodes that are linked to each other via labelled edges. Each complex node CN of XML Entry represents an object in XTree. CN is a tag name for non-leaf nodes that are represented by circles in XTree. CN normally has leaf nodes called basic nodes or BN, which are represented by rectangles in XTree.*

Example 1: Figure 2 shows XTree where B and C are CN because they have either BN or other CN or both. It can be seen that d, e and f are BN since they are leaf nodes.

Definition 2 (Object): *An object is defined as an element with complexType or complexContent and it is a non-leaf node in XTree. Objects might consist of other nested objects and basic elements.*

Example 2: In Figure 2, because c is CN, c is an object. It is a nested object of the object B.

XTree in Figure 2 likewise defines the cardinality constraints over the nodes in which a single shape and multiple shapes denote the maximum occurrence of one and more than one, respectively. Dashed lines and solid lines mean minimum occurrence of zero and one, respectively. XTree also represents the object size, which is the summation of the entire node's size value underneath the object, between practices within the graph.

In order to be clear about the objects contained in the document, XTree shows all the objects of XML data regardless of their usability or relevance in the partitioning process. XTree also defines the semantic relationships over the edges between objects.

Definition 3 (semantic relationship between objects): *Semantic relationship between objects is a representation categorised into two relationship-type annotations within XTree: Object relationship (ObR) and generalisation relationship (GnR).*

OBP has three essential criteria: (1) the object relationship type that is either ObR or GnR, (2) the occurrence constraint such as maxOccurs and minOccurs and (3) the type of nodes, which is either basic or complex. This diversity of node type is regarded and treated differently during the OBP process.

Definition 4 (Partitioning a basic node): *Partitioning a basic node that belongs to the parent node N is determined by including it with its parent partition. Let BN be a basic node with N as its parent and OP is the object partition.*

$$\forall BN_1, BN_2, ..., BN_n \in N \Rightarrow \bigcup_{i=1}^{n} BN_i \in OP(N),\ N \in OP$$

Definition 5 (Prior Parent Holder): *Prior Parent Holder (PPH) is an object with complex nodes, which has the priority to hold its parent in its partition. PPH has two essential conditions: (1) it is a complex element and (2) its minimum occurrence is one.*

PPH avoids node repetition and enables XML document reconstruction through a query. If all nodes have these two conditions, one of them will be randomly allocated as PPH and its parent elements will become its partition's root. The rest of the elements will have virtual roots in order to ensure the accuracy of generated XML partitions. Table 1 describes the symbols used in OBP algorithms. The algorithm is based on a top-down approach, similar to the way to traverse XML objects. Basic nodes are partitioned following Definition 4. Complex nodes or objects are initially partitioned based on two OBP algorithms. The first algorithm (see Figure 3a) is used when a parent object does not have basic nodes and has a relationship with complex nodes. For illustration, if we have two complex nodes (X, Y) and they relate to a PR as in Figure 3b, we will obtain two partitions and either X or Y can hold a parent node based on the PPH rules.

Table 1 Symbols used in OBP algorithms

Symbol	*Definition*
PR	Parent root
CN	Complex node
BN	Basic node
[ObR(1-1)|GnR(1-1)]	A CN relates to its PR through one-to-one ObR or GnR relationship
[ObR(1-∞)|GnR(1-∞)]	A CN relates to its PR through one-to-many ObR or GnR relationship
CN(CN_Q)	Complex node which includes other complex nodes
CN(BN_Q)	Complex node which includes basic nodes
CN_1	Number of complex node is 1
CN_Q	Number of complex nodes is Q

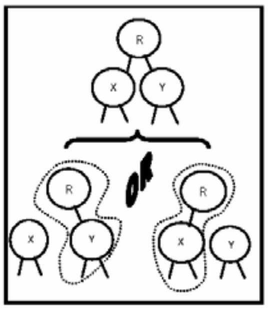
The second algorithm (see Figure 4) is used when the parent node has basic nodes and complex node(s). The main benefit of OBP is that objects, which will be partitioned, can be determined before the actual partitioning process which allows a saving of memory workload for the next layer.

Figure 3 (a) OBPI algorithm and (b) OBPI illustration

```
Algorithm 1: OBP1
Input: XTree
Output: objects // names of objects to be used in objects
partitioning

let COND 1 is not having BN
let COND 2 is only having $CN_Q$
if PR(COND 1, COND 2) then
 Create Q objects
 if PR(COND 1, COND 2) and PR is the root of XML document
then
  each Q object will have PR as its root
 else apply PPH rules
 end if
```

(a)



(b)

Figure 4 OBP2 algorithm

```
Algorithm 2: OBP2 Algorithm
Input: XTree
Output: objects name that ready to be used in
objects partitioning
1.  let [ObR(1-1)|GnR(1-1) ] is COND1
2.  let [ObR (1-∞)|GnR(1-∞)] is COND2
3.  if PR has ($BN_S$ and QCN)
4.   if ALL $CN_Q$(COND1) then
5.    if ALL $CN_Q$($CN_S$) or ALL $CN_Q$($BN_S$) then
6.     Q+1 objects [PR and QCN]
7.    elseif $CN_1$($BN_S$) and $CN_{Q-1}$($CN_S$) then
8.     Q objects [(PR+ $CN_1$($BN_S$)) and $CN_{Q-1}$($CN_S$)]
9.    end if
10. elseif ALL $CN_Q$(COND2) or [$CN_1$(COND1,$CN_S$)and
$CN_{Q-1}$(COND2)]then
11.     Q + 1 objects [PR and QCN]
12.   elseif $CN_1$(COND1, $BN_S$) and $CN_{Q-1}$(COND2)
then
13.     Q objects [(PR+ $CN_1$($BN_S$)) and $CN_{Q-1}$($CN_S$))]
14.   end if
15. end if
```

4.2 OXDP Layer 2: optimisation layer

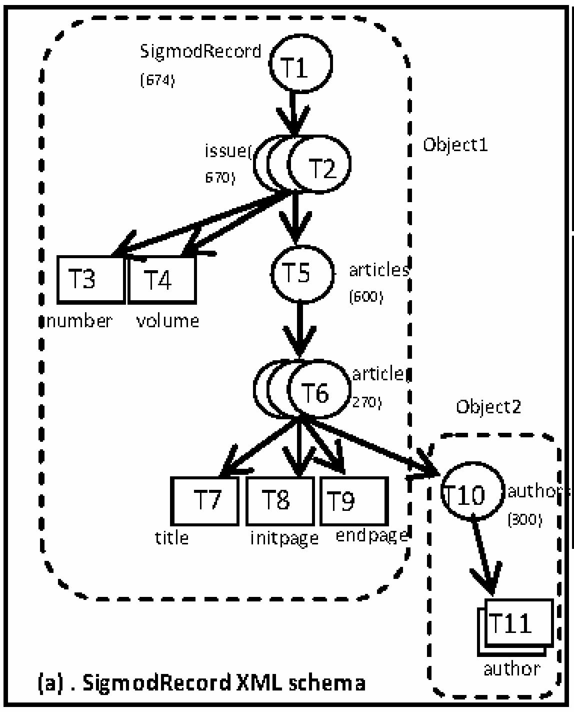
After extracting semantic features of XML data and identifying the objects, we execute the partition of XML documents. The partitioning algorithm, *OXDPartition*, is capable of determining object size. It obtains XML data with its XSD as inputs together with the outputs of the previous semantic layer. Then, it generates XML data partitions and XSD that corresponds to each partition. The main functionality of this algorithm is to check the objects' size. When it finds that the object is optimal, it calls the *partition_object()* that will create partitions for each object. However, when the object size exceeds the maximum limit, the algorithm uses the *partition_objects_horizontally()* method to partition the object horizontally. Alternatively, when the object size is less than the minimum limit, the *merge_objects()* method is used to combine the current object with its parents or its siblings. During the partitioning process, PartitionsID is set up to link some partitions and keep their referential data. The primary advantage of OXDPartition algorithm, aside from its ability to produce optimal partitions while preserving semantic structure, is the ability to check the size of the objects prior to the partitioning process in order to increase the performance by reducing the amount of memory buffers. Alghamdi et al. (2011) provide more detail about this algorithm.

Definition 6 (PartitionsID): *A pair of <linkID, objectID> is called PartitionsID in which linkID is a unique number given to elements to link between them, and objectID is a name of a partition where linked elements are located.*

It can be observed that linked elements share the same *linkID* but differentiate in *objectID* since each of them is located in different partitions. To illustrate OXDP algorithms, SigmodRecord in Figure 5a contains nested objects which are SigmodRecord, issue, articles, article and authors.

In the semantics layer, 'issue' is included in 'SigmodRecord' partitions based on OBP1, since its parent does not have any basic node. Since 'articles' has a *1-to-many* relationship with its parent that has basic nodes, 'articles' is excluded along with its children 'article' based on OBP2. The 'authors' is placed in a different partition. Three objects are generated at the completion of the semantics layer: 'SigmodRecord', 'articles' and 'authors'. However, in the optimisation layer, where OXDP identifies the object size, the result becomes two partitions 'SigmodRecord' and 'authors' with PartitionID to link them. At the end of the OXDP process, the optimal partitions with semantic meaning are ready to be queried.

Figure 5 SigmodRecord XML schema sample and its OXiP



(b). Tokenized Tags		
Tags	*Token*	*Type*
Sigmod	*T1*	*complex*
article	*T6*	*complex*
authors	*T10*	*complex*
title	*T7*	*simple*
...	...	...

(c). Schema path partition		
SP	*Tokenized path*	*Partition ID*
SP1	*T1 T2 T5 T6 T7*	*Object1*
...	...	...
(d). Schema partial path partition		
SP12	*T1 T2 T5 T6 T10*	*Nested Object2*
...	...	...

(e). Schema path-Nested partition	
SP	*Tokenized nested partition path*
SP4	*T1 T10 T11*
SP12	*T1 T10*

(f). Tokenized schema path	
Token	*Schema Paths*
T1	*SP1 SP2 SP3 SP4 SP5 SP6 SP7 SP8 SP9 SP10 SP11 SP12*
T6	*SP1 SP2 SP3 SP4 SP11 SP12*
T7	*SP1*
...	...
(g). Tokenized partial schema path	
T1	*SP8*
T10	*SP12*
...	...

5 OXiP: object-based XML data indexing partitions

In this layer, the knowledge of OXDP is utilised. XML data is semantically indexed, based on OO concepts resulting from OXDP. OXiP takes place after OXDP and consists of two components: (1) XML schema indices and (2) XML data indices.

5.1 OXiP terms and definitions

Definition 7 (tokens): *Each node name along a schema path is encrypted with a token.*

Figure 5b represents all distinct node names available in the XML Schema in Figure 5a along with their tokens and their type within the schema.

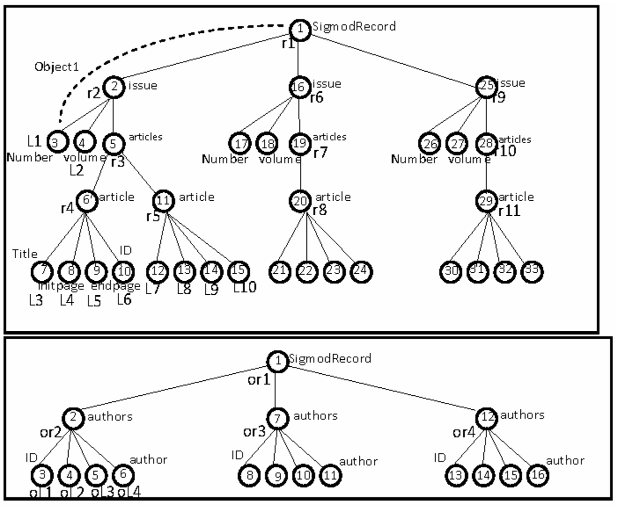
Definition 8 (XML data path): *A path P = {n_1, ..., n_k} is an XML data path, where each n_{i+1} is a child node of n_i for i = 1…k – 1, and is said to be an XML data path of length k if n_1 is a root of the XML data tree and n_k is a leaf node.*

Definition 9 (schema path identifier): *A path SP = {$T1_1$, ..., Tx_k} is a schema path identifier representing the XML data path P, where each Tx_i is a token of n_i for i = 1 … k, and is said to be a schema path identifier for an XML data path of length and where x is a token ID.*

Schema paths are similar to XML data paths starting from the root node. However, schema paths are created from XML schema and XML data paths from XML documents. Each XML data path should have its representation in schema paths.

From SigmodRecord XTree in Figure 5a, a path of the tree is from the root element SigmodRecord until the element title including all other elements within the same path i.e., issue, articles and article is considered a schema path which represents XML data paths of pointers 1,2,5,6,7, 1,16,19,20,21 and 1,25,28,29,30 (see Figure 6).

Figure 6 SigmodRecord XML data partitions generated by OXDP



In Figure 5c, a path T1 T2 T5 T6 T7 is a sequence of tokens for schema path SP1 which represents the previous XML data paths where each T_i is a token of each node in the XML data paths.

Definition 10 (schema path/schema partial path): *Let SP = {$T1_1$, ..., Tx_k} be a schema path identifier and P = {n_1, ..., n_k} be an XML data path, where $T1_1$ of SP is a token of n_1 which is a root of P. SP can either be a schema path when Tx_k of SP is a token of a leaf node n_k or a partial schema path when Tx_k of SP is a token of a non-leaf node n_k.*

SP1 = {T1 T2 T5 T6 T7} shown in Figure 5c identifies a schema path of XTree since T7 is a leaf node. In contrast, SP12 = {T1 T2 T5 T6 T10} of Figure 5d determines a schema partial path since T10 is a non-leaf node.

Definition 11 (query tags): *Consider a query Q = {qt_1 [/ | //] … qt_i[/ |//]; … ; qt_k }. Node names qt_i, for 1 ≤ i ≤ k, are called query tags, where k is the total number of tags in the query.*

For example, a query Q1 = sigmodRecord/issue//initpage is utilised. It can be considered that query tags inside the query are sigmodRecord, issue and initpage.

5.2 XML schema indices

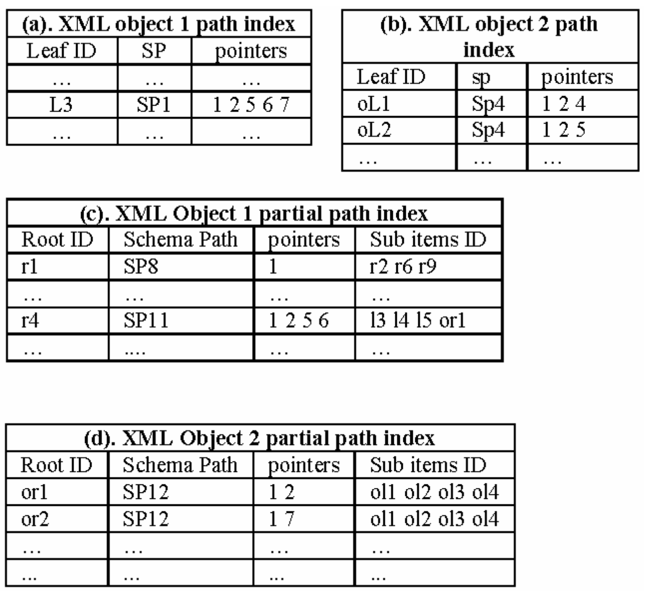
- *Tokenised tags*: This is a meta-data for XML schema used to assign an identical symbol for each element in XML schema that will later represent a node in XML data. It has been considered that these identically symbolled tags are distinct in the meta-data (see Figure 5b). The advantage here is that 'meta-data of schema' most likely does not require frequent changes because it is based on schema and not XML data. The reason beyond this is that in some cases, XML data does not represent all its schema elements and needs, in time, to add some elements which might be optional and do not exist in data. Therefore, meta-data tokenises all elements of XML schema significantly for further fast access. Moreover, tags have been chosen specifically due to the need to decrease the size of the index and avoid redundancies.
- *Schema path partition/schema partial path partition*: Along with each schema path or schema partial path, there are tokenised tags. The index allocates each path and an object partition ID where it exists. In other words, schema path/partial path partition indices identify all distinct paths within the schema and then identify tokens allocated along with each path in a schema besides its supposed partition. *Schema path/partial path partition indices* determine in which partitions the SPs are located.
- *Schema path nested partition*: In regard to OXDP, the generated partitions can be a nested partition. In this case, the schema path nested partition index carefully handles a path of tokenised tags for each SP, which takes a place in a nested partition, to be converted into an equivalent path of tokenised tags reflecting a real path of XML data within the partition.

- *Tokenised schema path index*: This stores each token associated with its SPs where the token exists. It is a reverse index to the schema path partition index since it stores all schema paths that share the same tokenised tag. However, the schema path partition index stores all tokens allocated within an SP. Different from the tokenised schema path which stores all SPs of a token, the tokenised schema partial path stores only the SP of a token that is against the tail node of a partial path. Definitely, this node is a non-leaf in the XML Tree (XTree).

5.3 XML data indices

- *XML objects' path index*: In schema path index, the tail of each SPs is a leaf node. All XML data paths that match the pattern of SP in the schema path index are stored in conjunction with its pointers. The pointer is a unique number assigned to each XML data node by traversing XML data 'in depth first order'. For example SigmodRecord/issue/articles/article/title in Figure 6 has pointers 1, 2, 5, 6 and 7 to determine the position of a node within an XML document. In this path, the schema path is SP1 which matches the schema path of L3 in Figure 7a.

Figure 7 OXiP of SigmodRecord XML data



(a). XML object 1 path index

Leaf ID	SP	pointers
…	…	…
L3	SP1	1 2 5 6 7
…	…	…

(b). XML object 2 path index

Leaf ID	sp	pointers
oL1	Sp4	1 2 4
oL2	Sp4	1 2 5
…	…	…

(c). XML Object 1 partial path index

Root ID	Schema Path	pointers	Sub items ID
r1	SP8	1	r2 r6 r9
…	…	…	…
r4	SP11	1 2 5 6	l3 l4 l5 or1
…	....	…	…

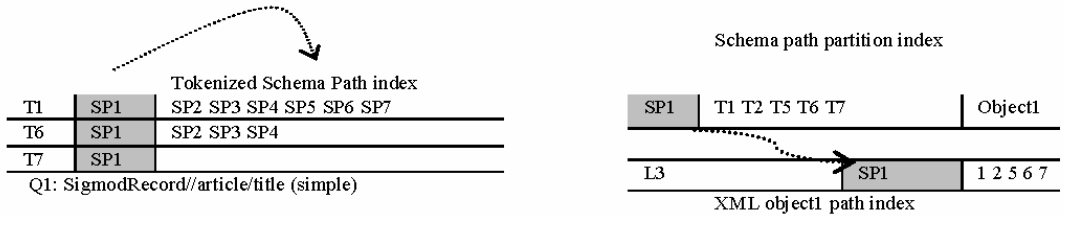
(d). XML Object 2 partial path index

Root ID	Schema Path	pointers	Sub items ID
or1	SP12	1 2	ol1 ol2 ol3 ol4
or2	SP12	1 7	ol1 ol2 ol3 ol4
…	…	…	…
...	...	...	...

- *XML object partial path index*: Since this index supports a path query ending with a non-leaf node, all XML data paths matching a specific SP in the schema partial path index are stored with their pointers along with the SP. For example, a path SigmodRecord/issue in Figure 6 has pointers 1 and 2, in that order, pointing to the positions of these nodes of the path query within the XML document. Since this index is concerned about retrieving an object, not only a leaf node, it keeps track of its children in the sub items' ID. If its child is a leaf node, it will store a leaf ID. However, if its child is a non-leaf and located in a different partition, PartitionsID will be adapted to link partitions and retrieve the sub-item ID. This index is filled in bottom-up traversing XML partitions. For instance, in the leftmost path of object1 in Figure 5a, SigmodRecord/issue/articles/article, matching SP11 and with pointers 1, 2, 5, 6 respectively, has 'L3 L4 L5 or1' as its children (see Figure 7c). To be clear, 'L3 L4 L5' can be retrieved from the same index of object1; however, 'or1' is in object2 where or1 has ol1 ol2 ol3 ol4. The rest of the data can be retrieved in a similar manner.

For the demonstration of the proposed method, in Figure 8, Q1: SigmodRecord//article/title is considered. It has been used over the schema and XML data of the SigmodRecord dataset. This query example is used to illustrate how the OXiP can sufficiently process the query. First the query needs to be analysed to obtain all the tokens of its tags 'SigmodRecord', 'article' and 'title' which are T1, T6, and T7, respectively, as shown in Figure 8. As 'title' is a simple/leaf node type, the paths of schema associated with each token of the query tags are retrieved from the tokenised schema path index (see Figure 5f). All three lists of schema paths associated with each token intersect in SP1, which is the only path that contains all three tokens and is located in object partition 1 with the same partition schema path. It can be seen that the main purpose of the schema path partition index is to identify the exact partition allocation besides its ability to identify the correlation between tags within the schema path. By using this technique, there will be a verification of hierarchical satisfactory of a schema path between the three tags 'SigmodRecord, article and title' and will reduce the exceeding number of joins in past indexing methods. If the SP1 schema path, which is T1 T2 T5 T6 T7, traverses scanning forward, then T1, T6 and T7 can be found in the same order. The tokenised partition path associated with SP1 and the partitioned ID are referred to for further processing. Partition ID assists in reaching the XML data index that is related to specific partitions and this decreases the searching and matching time. For each path of items within XML data associated with each passing schema path identifier, the pointer numbers of XML data nodes that occur in the same query tag positions in the partition path schema entry return a join result.

Figure 8 Q1: processing SigmodRecord//article/title



Tokenized Schema Path index

T1	SP1	SP2 SP3 SP4 SP5 SP6 SP7
T6	SP1	SP2 SP3 SP4
T7	SP1	

Q1: SigmodRecord//article/title (simple)

Schema path partition index

SP1	T1 T2 T5 T6 T7	Object1

L3	SP1	1 2 5 6 7

XML object1 path index

Figure 9 Processing Q2: SigmodRecord//article/authors

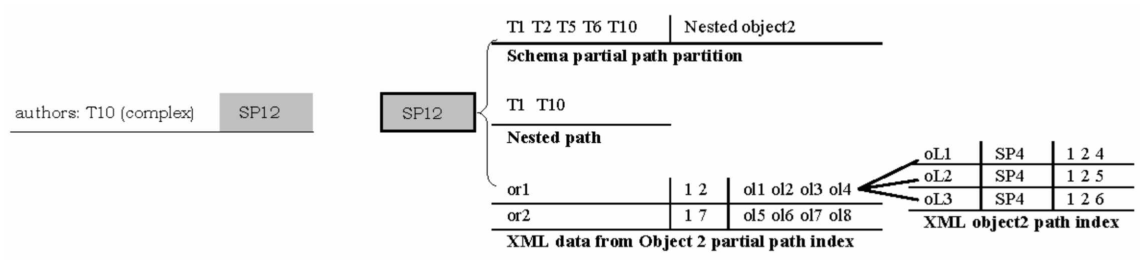


Figure 10 OXiP algorithm and its functions

```
Algorithm 3: Query Processing with OXiP
Input: path query
Output: XML node pointers
1. isPC = true;
2. foreach (qT in qTs)
3.    if (qT == "")then
4.       qEs.Add("");
5.       isPC = false;
6.    else qEs.Add(Tokenise[qT]);
7.    end if
8.   end for
9.       if (tType[qTs[qTs.Length - 1]]= ="simple")then
10.        isSimple = true;
11.       else isSimple = false;
12.       end if
13.       cSP = getCandidatePaths(qEs, isSimple);
14.    if (isPC) then
15.      if (qTs[0] == Root)then
16.       objectID = getObjectsOfPath(cSP);
17.       if (XMLPaths[cSP].ContainsKey(objectID)) then
18.        foreach (XPath in XMLPaths[cSP][objectID])
19.          for (int i = 0; i < XMLPath.Count; i++)
20.              tmp.Append(XPath[i]);
21.          end for
22.          nPointers.Add(tmp);
23.         end for
24.      else isPC = false;
25.      end if
26.    if (!isPC) then
27.      foreach (SP in cSP)
28.       if (!XMLPaths.ContainsKey(SP)) then
29.          continue;
30.       end if
31.       nPos = validateQuery(SP, qEs, isSimple);
32.       if (nPos.Count == 0) then
33.       continue;
34.       end if
35.       foreach (objectsID in XMLPaths[SP])
36.        foreach (XPath in XMLPaths[SP][objectsID])
37.          for (int i = 0; i < nPos.Count; i++)
38.              tmp.Append(XPath[nPos[i]]);
39.          end for
40.          nPointers.Add(tmp);
41.         end for
42.        end for
43.       end for
44.     end if
45.     for (int i=0; i<nPointers.Count; i++)
46.              return nPointers[i];
47.     end for
```

```
Function getCandidatePaths
Input : qEs, isSimple
output: cSP
1. lastElement = qEs[qEs.Count - 1];
2. if (isSimple) then
3.    cSP.Add(TokenizedSchemaPath[lastElement]);
4. else cSP.Add(TokenizedPartialSchemaPath[lastElement]);
5. end if
6. for (int i = 0; i + 1 < qEs.Count; i++)
7.  if (qEs[i].Equals(""))
8.    continue;
9.  cSP = cSP.Intersect(TokenizedSchemaPath[qEs[i]]);
10. end for
11.return cSP;
```

```
Function validateQuery
Input:SP, qEs, isSimple
Output:nPos
1. if (isSimple) then     path = schemaPaths[SP];
2. else path = schemaPartialPaths[SP]; end if
3. j = 0;
4. for (int i = 0; i < path.Count; i++)
5.  if (qEs[j].Equals("")) then j++;
6.  else if(nPos.Count != 0 && nPos[nPos.Count-1]!=i-1)
7.       return new List<int>();
8.  end if
9.  if (path[i].Equals(qEs[j]))
10.      nPos.Add(i);
11.      j++;
12. end if
13. end for
14.if (j == qEs.Count) break;
15.return nPos;
```

In Figure 9, Q2: SigmodRecord//article/authors is used to elaborate how a query is processed for a partial path. The ability of OXiP to link XML data partitions can be seen in this example besides its capability to support OXDP as a double optimisation layer. The query tags have T1, T6, and T12 as shown in Figure 5b. Since T12, which is the tail of the path query, has a complex type, a path of schema associated with T12 is retrieved from the tokenised partial schema path index (refer to Figure 5g). The schema path is shown in Figure 5d to be T1 T2 T5 T6 T10 located in nested object2. This information identifies from which partition the data will be retrieved in order to clarify that this partition is a nested object of another object. For this reason, a new path of tokens will be allocated for SP12 as can be seen in Figure 5e. XML data is retrieved in conjunction between object2 and partial object2.

5.4 Query processing with OXiP

An OXiP query-processing method has been proposed to take full advantage of OXiP characteristics. *Algorithm 3*, in Figure 10, has been used to process path queries more efficiently. The essential goal of this algorithm is to match query tags based on tokens. Table 2 shows the algorithm symbols.

Table 2 OXiP algorithm symbols

Symbols	*Descriptions*
qTs	Query tags
qEs	Query elements
tType	Query tag type
cSP	Candidate schema path
nPointer	XML node position
isSimple	Flag indicating type of a query tag
isPC	Flag determining type of relationship between query tags
nPos	Token positions in query matching with token positions in SP

The input of the algorithm is a sequential query. The query is analysed into tags and relationship types which are either ancestor–descendant A–C or parent–child P–C relationship. Let us consider a sequential query Q = qt1/qt2//qt3. The query tags are qt1, qt2, qt3, and the relationships between tags are ancestor–descendant and parent–child. From the tokenised tag index, each extracted tag is mapped into its token represented in Q = {T1, T2, T4}. This part is depicted in Algorithm 3, lines 2–8. Hereafter, getCandidatePath is invoked with tag type and elements of query as its arguments and returns schema path identifiers cSP.

In *getCandidatePaths*, there are two streams: (1) If qt3 in the query Q is a leaf node, the tokenised schema path index will be used to find the schema path identifier that contains all query tokens. For example, let us say *β is a* tokenised schema path and *β = {SP(T1), SP(T2), SP(T4)}*, where *SP(T1)* is a set of schema path identifiers containing the token *T1* and so forth. (2) If qt3 is non-leaf, the tokenised partial schema path index will be used to retrieve SP where the token of qt3 exists. This function will return the intersection values of schema path identifiers.

At line 14 of Algorithm 3, if Q has P–C path only, one path cSP will be retrieved, cSP will be the intersection between *SPs* in *β* and *cSP = SP*, where *SP* is a schema path identifier that consists of all query tags. However, at line 26 of Algorithm 3, if Q has A–C, then *cSP = {SP1 ... SPm}*, where each *SPi* will be a schema path identifier that consists of all query tags and $1 \leq i \leq m$, *m* is the number of intersected schema path identifier. This intersection of paths is used to eliminate processing false matches. In Algorithm 3, relationship classification is identified by a flag called isPC. Once the flag has true value, this means all relationships between query tags are P–C.

However, if it is false, an A–C relationship between query tags should exist. In case all the paths are P–C, lines 14–25 in the algorithm are invoked, else lines 26–44 in the algorithm will be executed. objectID will be retrieved from the schema path partition, and it will assist to prune the search space by eliminating irrelevant parts of XML data. Then the matching and joining process will take its place. The output of this algorithm is the positions of the nodes of interest in the tokenised schema path, which are used to determine the result from the pointers of the XML items in lines 45–47.

validateQuery at line 31 is invoked to validate the query tokens with schema path tokens. ST = {t_1 ... t_ℓ} where ℓ is the length of schema path, for each schema path identifier entry QT = {t_1 … t_j} where *j* is the number of tags in the query QT. By matching ST with QT, the tokens' positions can be identified to retrieve the position of XML data in the algorithm.

6 Experiments and evaluation

OXDP and OXiP have been developed using MS Visual C#, and the database used is an EXD using its Oracle XMLType facility. XSD needs to be traversed using SOM 'Schema Object Model' to determine all object partitions including nested objects. The evaluation was done in an Intel® Core™2 Duo CPU with 2.1.00 GHz processing power. It has 2048 MB of high-speed memory and runs the 32-bit version of Window Vista™.

For evaluation, we conduct an experiment by running different queries. This experiment is divided into two stages: (1) the evaluation of OXDP and (2) evaluation of OXiP.

6.1 Evaluation of OXDP

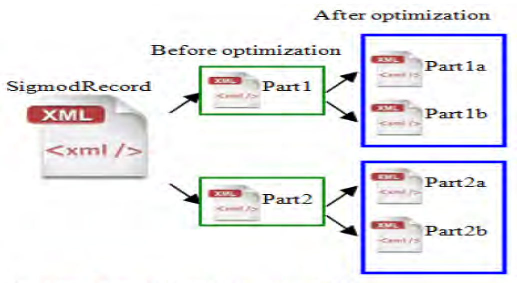
This evaluation has been conducted using different XML datasets: SIGMODRecord and Yahoo (UW, 2002). XMLQuery, XMLTable and XPath were used in EXD. Query performance is measured by the number of accessed data blocks (NADBs). The queries were categorised into three groups: (1) Query Group 1 which varies the number of partitions accessed for a query; (2) Query Group 2 which varies the size of the file and (3) Query Group 3 which alters the depth of the accessed path during the queries.

In Query Group 1 (see Table 3), the query cost was determined for partitioned and non-partitioned XML data in addition to querying optimised and non-optimised partitioned data. Horizontal partitions are stored in a table of XMLType to reduce the join cost. These query scenarios are shown in Figure 11. Figure 12 shows that partitioned XML data outperforms non-partitioned XML data in terms of query performance. In addition, horizontal partitioning methods improved query performance too.

Table 3 QGroup 1 for SigmodRecord

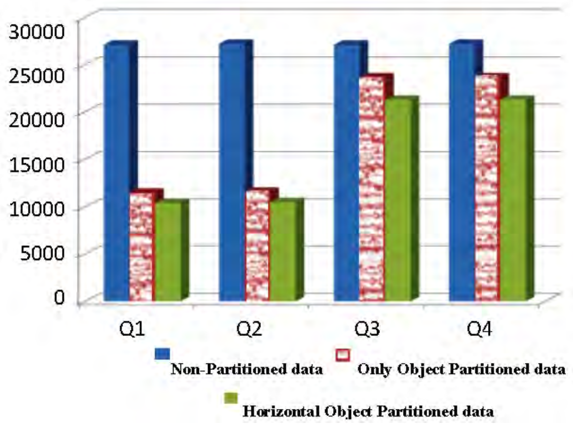
Q#	*Queries*
Q1	SigmodRecord/issues/articles/article [title=" Architecture of Future Data Base Systems."]/ initPage |endPage
Q2	SigmodRecord/issues[volume="1"andnumber="2"] /articles/article[initPage="3"]/title
Q3	SigmodRecord/issue/articles/article/authors [author="Stephen Knowles']/ ../article/title
Q4	SigmodRecord/issue/articles/article [title="Architecture of Future Data Base Systems."]/ authors/ author

Figure 11 A visual scenario of QGroup 1 for SigmodRecord (see online version for colour)



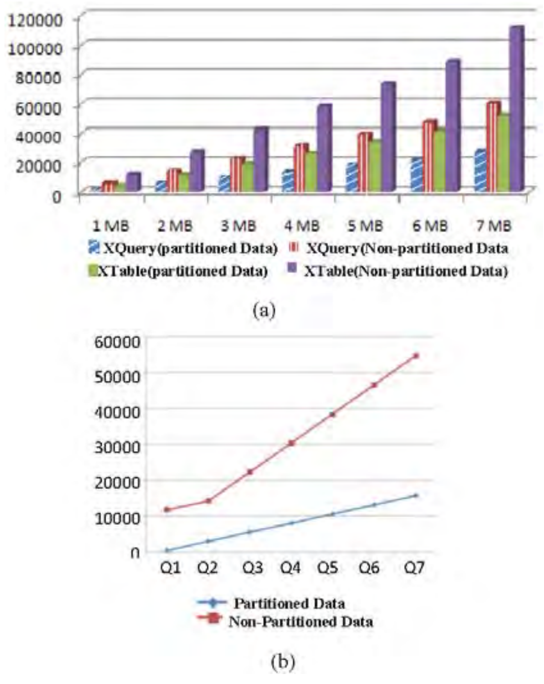
Q1: (SigmodRecord) X (part1) X (part1a, part1b)
Q2: (SigmodRecord) X (part1) X (part1a, part1b)
Q3: (SigmodRecord) X (part1,part2) X (part1a, part1b, part2a, part2b)
Q4: (SigmodRecord) X (part1, part2) X (part1a, part1b, part2a, part2b)

Figure 12 NADB in SigmodRecord XMLdata (see online version for colour)



In Query Group 2, we test the query performance for varying data size. We performed the same queries using XMLQuery and XMLTable methods. In Figure 13, the partitioned data needs to access less data blocks to obtain the answers of the same queries performed on the non-partitioned data. OXDP demonstrated that it is possible to manipulate XML data with different sizes and to reduce query cost.

Figure 13 NADB in (a) SigmodRecord (b) yahoo XML data (see online version for colour)



In Query Group 3, we check the cost effectiveness in retrieving some elements presented in various paths which can be accessed in the partitioned XML data via different paths. The visual representation for these queries is shown in Figure 14. The variation of the path can be seen in Table 4. The results of these queries show that the OXDP methodology also exhibits better performance than the results of non-partitioned datasets as shown in Figure 15.

Figure 14 A Visual Scenario of QGroup 3 for (a) SigmodRecord and (b) Yahoo (see online version for colour)

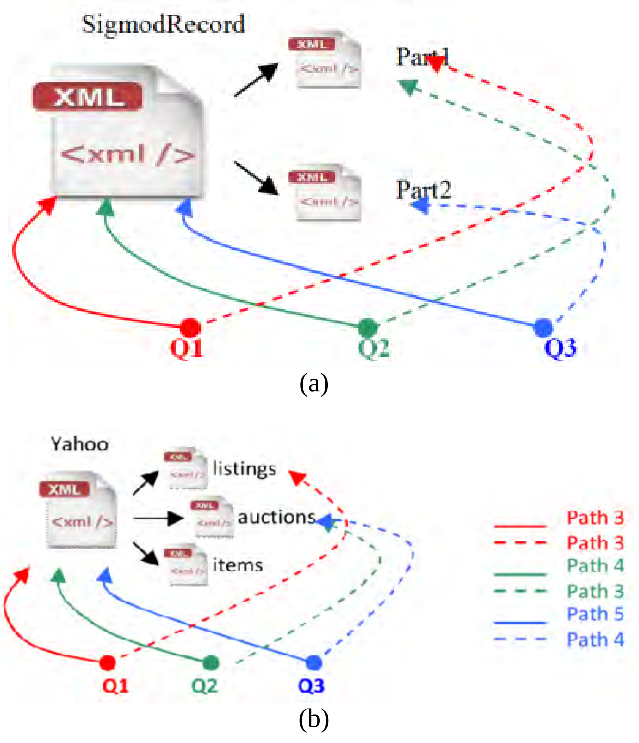
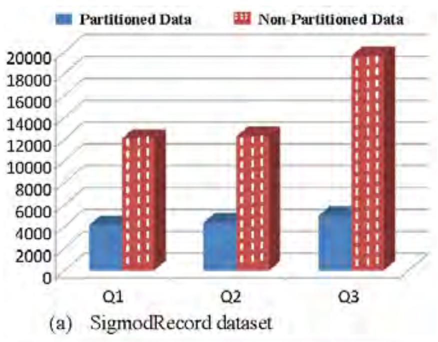
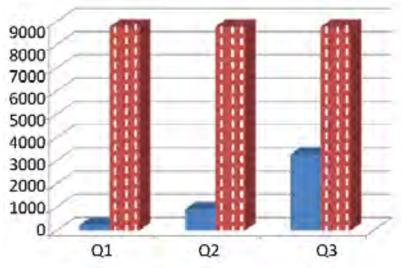


Figure 15 NADB for QGroup3 (see online version for colour)



(a) SigmodRecord dataset



(b) Yahoo dataset

Table 4 Experiments for QGroup 3

Q#	*Depth P*	*Depth Non-P*	*Elements*	*Dataset*
Q1	3	3	//volume	SigmodRecord
Q2	5	5	//title	SigmodRecord
Q3	3	6	//author	SigmodRecord
Q1	3	3	//payment_types	Yahoo
Q2	3	4	//location	Yahoo
Q3	4	5	//bidder_name	Yahoo

6.2 Evaluation of OXiP

Evaluation of OXiP is carried out on three datasets. DBLP and SigmodRecord are real datasets obtained from the UW repository (UW, 2002). Standard is a synthetic dataset obtained from XMark benchmark project (XMark, 2004). Experimental queries are shown in Table 5 with their classification while Table 6 shows the characteristics of the chosen datasets. We compared our result with that of state-of-the-art TwigX-Guide since it has shown comprehensive performance advantages over many other approaches (Haw, 2009). We implemented TwigX-Guide algorithm using C0023 because it is not publicly available at this time.

For queries containing only P–C edges as shown in Table 4, we found similar comparative performance between OXiPs and TwigX-Guide. When both of them query only P–C edges, they use few joins as path matching. Thus, in this case, the performance is optimised. In OXiP, P–C edges are stored as a path of tokens in the schema path partition index. Hence, the desired SP can be used with the objectID to retrieve the required XML data by accessing a small portion of the data and avoid the non-participating part. In TwigX-Guide, P–C edges could be obtained from the Dataguide index table and, thus, the number of required joins is few.

Table 5 Experiments for OXiP

Q#	*P–C*	*A–C*	*Blended*	*Datasets*	*Query*
1	√			DBLP	/dblp/inproceedings/ booktitle
2		√			/dblp//author
3			√		/dblp/inproceedings//i
4	√				/dblp/mastersthesis/title
5		√			/dblp//title
6			√		/dblp/www//title
7	√			XMark	/site/closed_auctions/ closed_auction/price
8		√			//description//keyword
9			√		/site/regions//item/location
10	√				/site/open_auctions/ open_auction
11		√			//regions//item
12			√		/site//africa/item/ description//keyword
13	√			Sigmod Record	/SigmodRecord/issue/ volume
14			√		/SigmodRecord//article/ title
15		√			/SigmodRecord//articles// author
16			√		/SigmodRecord/issue// author
17		√			/SigmodRecord//issue// title
18		√			/SigmodRecord//articles
19		√			//articles//authors

Table 6 Characteristics of the datasets

	DBLP	*XMark*	*SigmodRecord*
Size (MB)	131	114	470
Max. depth	6	12	6
Objects#	8	11	2

The main feature of OXiP utilises the notion of objects in processing the query. Therefore, it has the ability to eliminate irrelevant objects. This feature has a significant impact in improving the performance of querying either A–D edges only or mixed with P–C edges. OXiP performs significantly better compared to TwigX-Guide when the query contains only A–D edges. TwigX-Guide requires a greater number of joins. It can be observed that OXiPs greatly outperform in Q2, Q5 in DBLP (see Figure 16) and Q8 in XMark (see Figure 17) because the candidate SP identifier coupled with the determined object assists in reducing the matching process. As the number of matchings reduces, a fewer numbers of disks is accessed and therefore the execution time is optimised.

Figure 16 OXiP evaluation with DBLP (see online version for colour)

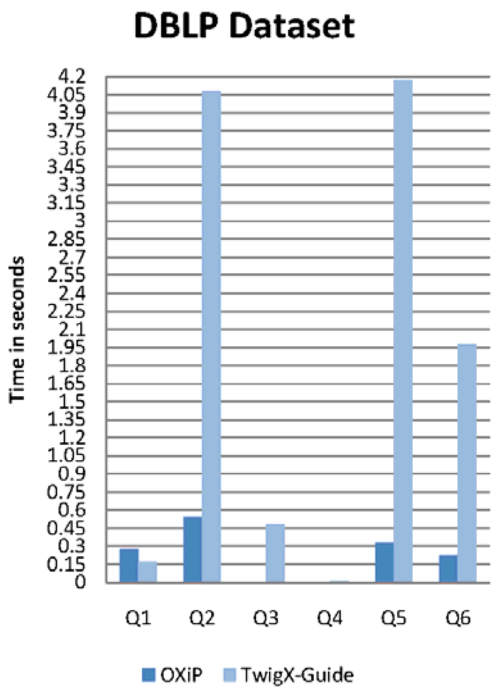


Figure 17 The evaluation of OXiP with XMark (see online version for colour)

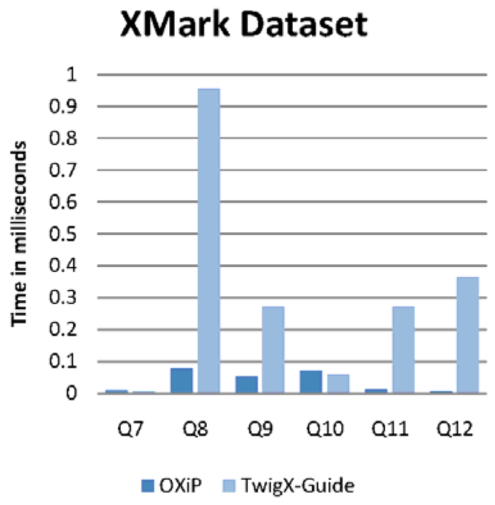
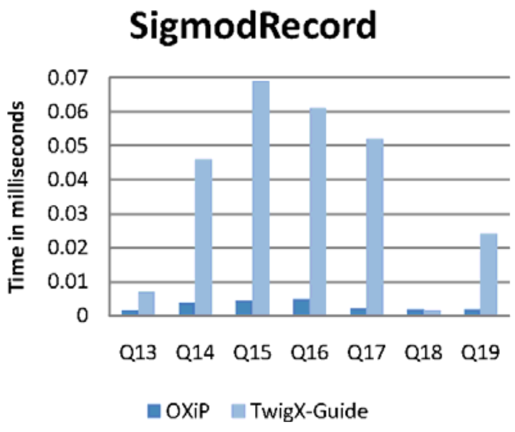
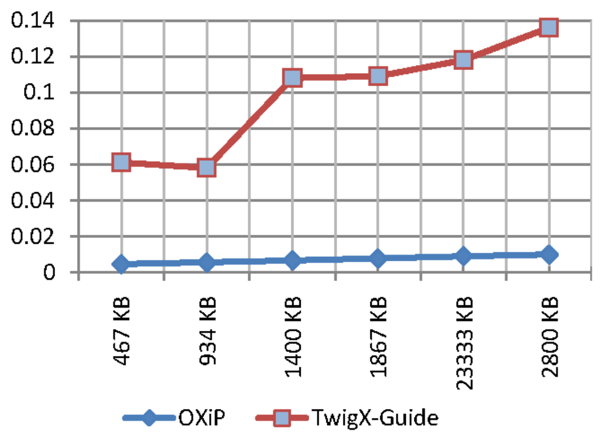


Figure 18 OXiP evaluation with SigmodRecord (see online version for colour)



When path queries contain blended edges, the results show that OXiP outperforms TwigX-Guide. Figures 16, 17 and 18 show the improvement of the performance test results caused from OXiP. It can be seen in Figure 16 that OXiP and TwigX-Guide achieved an optimum result which is quite less than 0.15 ms when we run Q4. Figure 19 indicates that OXiP is more scalable in processing different-scale datasets than TwigX-Guide when we used Q14 from Table 5. On average, OXiP outperforms TwigX-Guide by around 75.73% in queries of DBLP, by 53.46% in queries of XMark and 74.98% in queries of SigmodRecord.

Figure 19 The evaluation of OXiP scalability (see online version for colour)



7 Conclusion and future research

In this paper, we introduced two contributions which can work independently: (1) a new method to partition XML data based on semantics by performing object-based partitioning (OXDP) and (2) OXiP which provides its ability to link partitions as well as processing path queries. We demonstrated that OXDP allows elements that are semantically close to each other to be stored together within the same partition. We have also developed an optimisation algorithm which allows the size of the partitioned XML data to be equally distributed to avoid high costs in query performance. OXDP has demonstrated a method to improve the query performance in EXD environments and yields better performance gains. OXiP is an extended layer of the system cooperating with OXDP. It aims to further improve the query performance by exploiting the semantics of OXDP and adopting an index technique.

For further research and investigation, we focus on extending our methods to deal with branching queries and work out a query-processing method to deal with this kind of query along with OXDP.

Acknowledgements

Norah Saleh Alghamdi would like to express her gratitude to the Taif University in Saudi Arabia, for supporting her with a scholarship.

References

Alghamdi, N., Rahayu, W. and Pardede, E. (2011) ‘Object-based methodology for XML data partitioning (OXDP)’, *Proceedings of the 25th International Conference on Advanced Information Networking and Applications* (*AINA 2011*), Singapore, March 22–25, pp.307–315.

Berglund, A., Boag, S., Chamberlin, D., Fern´andez, M.F., Kay, M., Robie, J. and Sim´eon, J. (2011) *XML Path Language* (*XPath*) *2.0* (*Second Edition*), World Wide Web Consortium, W3C Recommendation.

Bordawekar, R. and Shmueli, O. (2004) ‘Flexible workload-aware clustering of XML documents’, *Proceedings of Second International XML Database Symposium*, *XSym 2004*, 29–30 August, Springer, Toronto, Canada.

Chamberlin, D., Robie, J. and Florescu, D. (2000) ‘An XML query language for heterogeneous data sources’, *Proceedings of the International Workshop on the Web and Databases* (*WebDB’ 2000*), Springer-Verlag, Dallas, TX.

Chen, Q., Lim, A., Ong K. W (2003) ‘*D*(*k*)-Index: an adaptive structural summary for graph-structured data’, *Proceedings of the 2003 ACM SIGMOD*, New York, NY, USA.

Chung, C-W., Min, J-K. and Shim, K. (2002) ‘APEX: an adaptive path index for XML data’, *Proceedings of ACM SIGMOD International Conference on Management of Data*, Madison, WI, USA, pp.121–132.

Cooper, B., Sample, N., Franklin, M.J., Hjaltason, G.R. and Shadmon, M. (2001) ‘A fast index for semistructured data’, *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB),* San Francisco, CA, USA.

Farfán, F., Hristidisa, V. and Rangaswami, R. (2009) ‘2LP: a double-lazy XML parser’, *Information Systems*, Vol. 34, No. 1, pp.145–163.

Goldman, R. and Widom, J. (1997) ‘Dataguides: enabling query formulation and optimization in semistructured databases’, *Proceedings of the 23 International Conference on VLDB*, Athens, Greece, pp.436–445.

Hao, H. and Yang, J. (2004) ‘Multiresolution indexing of XML for frequent queries’, IEEE Computer Society, *Proceedings ICDE ’04*, Boston, Massachusetts, USA.

Haw, S. and Lee, C. (2009) ‘Extending path summary and region encoding for efficient structural query processing in native XML databases’, *Journal of Systems and Software*, Vol. 83, No. 6, pp.1025–1035.

Kanne, C. and Moerkotte, G. (2006) ‘A linear time algorithm for optimal tree sibling partitioning and approximation algorithms in Natix’, *Proceedings of the 32nd International Conference on VLDB Endowment*, Seoul, Korea.

Kaushik, R., Shenoy, P., Bohannon, P. and Gudes, E. (2002) ‘Exploiting local similarity for indexing paths in graph-structured data’, *Proceedings of the 18th International Conference on Data Engineering,* Washington, USA.

Li, Q. and Moon, B. (2001) ‘Indexing and querying XML data for regular path expressions’, *Proceedings of the 27th International Conference on VLDB*, September 11–14, Morgan Kaufmann, Roma, Italy, pp.361–370.

McGovern, J., Bothner, P., Cagle, K., Linn, J. and Nagarajan, V. (2003) *XQuery Kick Start*, SAMS Publishing, Indianapolis, IN, USA.

Milo, T. and Suciu, D. (1999) ‘Index structures for path expression’, *Proceedings of International Conference on Database Theory* (*ICDT*), January, Springer, London, UK.

Mohammad, S. and Martin, P. (2010) ‘LTIX: a compact level-based tree to index XML databases’, *Proceedings of the 14th International Database Engineering & Applications Symposium*, NY, USA, pp.21–25.

Rao, P. and Moon, B. (2004) ‘PRIX: indexing and querying XML using Prüfer sequences’, *Proceedings of the 20th International Conference on Data Engineering, ICDE 2004*, March 30–April 2, Boston, MA, USA.

Robie, J., Chamberlin, D., Dyck, M. and Snelson, J. (2010) ‘XQuery 3.0: an XML query language W3C working draft’.

Tatikonda, S., Parthasarathy, S. and Goyder, M. (2007) ‘LCS-TRIM: dynamic programming meets xml indexing and querying’, *Proceedings of VLDB ’07*, Vienna, Austria.

Tusa, F., Villari, M. and Puliafito, A. (2009) ‘Design and implementation of an XML-based grid file storage system with security features’, *WETICE ’09*, Groningen, the Netherlands, pp.183–188.

UW (2002) *University of Washington XML Repository*. Available online at: http://www.cs.washington.edu/research/xmldatasets

Wang, H., Park, S., Fan, W. and Yu, P.S. (2003) ‘ViST: a dynamic index method for querying XML data by tree structures’, *Proceedings of the 2003 ACM SIGMOD*, San Diego, California, USA.

XMark (2004) *XMark – an XML benchmark project*. Available online at: http://www.xml-benchmark.org

Zhang, J. and Phillips, C. (2011) ‘Job-scheduling via resource availability prediction for volunteer computational grids’, *International Journal of Grid and Utility Computing*, Vol. 2, No. 1, pp.25–32.