Future Generation Computer Systems 37 (2014) 212–231

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Semantic-based Structural and Content indexing for the efficient retrieval of queries over large XML data repositories

Norah Saleh Alghamdi [*], Wenny Rahayu, Eric Pardede

Department of Computer Science and Computer Engineering, La Trobe University, Australia

HIGHLIGHTS

- We exploit the semantics of XML Schema and data in building our index.
- We trim search space using an object-based intersection technique of large data.
- We eliminate irrelevant portions of data by discarding and irrelevant objects.
- We prove the efficiency based on measuring CPU Cost and the scalability.
- We show the high precision and recall of query results.

ARTICLE INFO

Article history:
Received 19 April 2013
Received in revised form
10 October 2013
Accepted 17 February 2014
Available online 11 March 2014

Keywords:
Query optimization
Index
Query processing
Objects
Semantic
XML
Twig query

ABSTRACT

The emergence of XML adoption as semi-structured data representation in multi-disciplinary domains has highlighted the need to support the optimization of complex data retrieval processing. In a Big Data environment, the need to speed up data retrieval processes has further grown significantly. In this paper, we have adopted an optimization approach that takes into consideration the semantics of the dataset in order to deal with the complexity of multi-disciplinary domains in Big Data, in particular when the data is represented as XML documents. Our method particularly addresses a twig XML query (or a branched path query), as it is one of the most costly query tasks due to the complexity of the join operation between multiple paths. Our work focuses on optimizing the structural and the content part of XML queries by presenting a method for indexing and processing XML data based on the concept of objects that is formed from the semantic connectivity between XML data nodes. Our method performs object-based data partitioning, which aims at leveraging the notion of frequently-accessed data subsets and putting these subsets together into adjacent partitions. Then, it evaluates branched queries through two essential components: (i) Structural and Content indexing, which use an object-based connection to construct indices i.e. Schema Index, Data Index and Value Index; and (ii) query processing to produce the final results in optimal time. At the end of this paper, a set of experimental results for the proposed approach on a range of real and synthetic XML data, as well as a comparative study with other related work in the area, are presented to demonstrate the effectiveness of our proposed method in terms of *CPU cost, matching and merging cost, scalability (size and number of branches) and total number of scanned elements*. Our evaluation demonstrates the benefit of the proposed index in terms of performance speed as well as scalability which is critical in a large data repository.

1. Introduction

The powerful ability of XML in describing and presenting data has been recognized as the standard for electronic data interchange in multi-disciplinary domains [1–3]. The metadata in XML documents provides a semantically rich structure which can be leveraged for various information system applications. The metadata also opens up opportunities to improve techniques to access and process XML data.

In this paper, we focus on processing XML queries efficiently by taking into consideration the semantic connectivity of the underlying XML documents. In particular, we focus on XML twig queries with or without value predicates. A twig query is a type of query which accesses XML trees with multiple branches and

[*] Corresponding author. Tel.: +61 421386433.
E-mail addresses: nalghamdi@students.latrobe.edu.au, ninasg11@hotmail.com (N.S. Alghamdi), w.rahayu@latrobe.edu.au (W. Rahayu), e.pardede@latrobe.edu.au (E. Pardede).

http://dx.doi.org/10.1016/j.future.2014.02.010
0167-739X/© 2014 Elsevier B.V. All rights reserved.

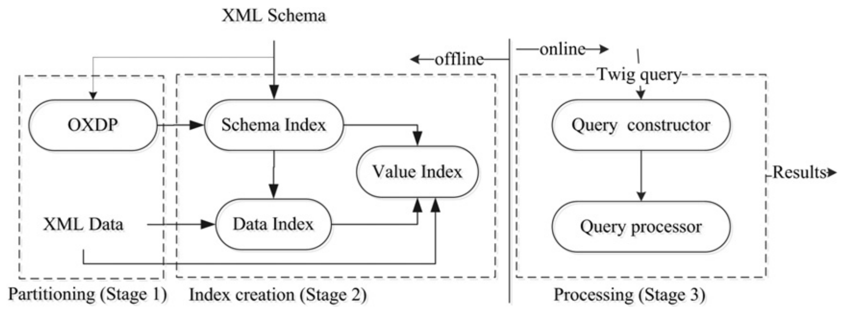


Fig. 1. The system configuration.

this query requires complex processing due to the joins between multiple paths.

A Naïve query processing by scanning the entire XML data to search for a particular path will cause significant performance degradation. Indexing schemes have been developed in recent years to overcome this issue. Different XML data indexing and query processing approaches have been proposed to support twig queries. The previous XML data indices were classified into three categories [4]. The first is path-based indices such as APEX [5] and MDFB [6], which group nodes in data trees based on local similarity and have an adjustable index structure depending on the query workload. These indices need to deal with a huge index size because its index keeps tracks of the forward and backward paths to establish a supportive layer that can help in answering twig queries effectively. The second is node-based indices, such as TwigX-Guide [7], which index the position of each node within the XML tree and then process the nodes by joining them when in some cases; aggressive joins deteriorate the query performance. The third is sequence-based indices such as ViST [8], PRIX [9] and LCS-Trim [10], which evaluate queries based on sequence matching after transforming both XML data and twig queries into sequences. However, the third approach has a drawback, which is the occurrence of false positive caused by sequence matching instead of tree matching. All the above work focus on the structural presentation of XML data (e.g. a sequence of nodes or a tree pattern) rather than the semantic relationship between groups of nodes (i.e. objects). Therefore, exploiting the semantic relationship between XML data nodes to build an index scheme for twig query processing is an ideal solution which has not been proposed in the existing literature as yet.

The work presented in this paper is the ongoing work of a research project on XML query optimization, which consists of three stages (see Fig. 1). The first two stages can be considered as off-line stages and the third is on-line stage. Stage one focuses on the object-based partitioning methodology of XML data. Stage two focuses on the XML data indexing methodology. Stage three focuses on the query processing method over indexed data. This paper presents the second and third stage in more detail and the methodology of the first stage of the Object-based XML Data Partitioning (OXDP) has been presented in [11,12]. Fig. 1 shows an overview of our system configuration.

Our proposed indices in the second stage consist of three main parts. The first part is the Schema Index; the second is the Data Index; and the third is the Value Index. The construction phases of the indices in the current stage are:

Phase 1: We build the Schema Index based on the object partitions identified by the OXDP process. We tokenize each distinct element tag to set up the Schema Index components.

Phase 2: We construct the Data Index components by grouping the XML data within object partitions and we establish keys from the Schema Index to the Data Index.

Phase 3: We then build the Value Index with knowledge of Schema and the Data Index.

Then, in the third stage of our system, we proposed a query processing method to handle indexed data. The query processor can evaluate simple XML paths as well as XML paths with branches and different value predicates.

This paper is organized as follows. Section 2 provides our motivation with a brief example that shows the utilization of the semantics of XML in indexing and processing XML data to improve the query performance. Also, it presents the importance of processing value predicates in XML queries. The related work is reviewed in Section 3. The preliminary knowledge is presented in Section 4. We introduce our proposed indices in Section 5 and discuss the processing method of XML queries in Section 6. Experimental results are provided in Section 7. Finally, we conclude the paper in Section 8.

2. Our motivation and contributions

While most of the existing work does not consider the semantics of XML data during the construction or processing of XML data, our work actually exploits the semantics connectivity between nodes to construct our indices. The incorporation of the semantic features of index construction into the XML query processing approach will lead to an efficient pruning technique. This is because the search space can be trimmed down to a group of data that follows certain semantics.

$Q1 = \text{“}/purchaseOrder/ShipTo[city][state]/name/Fname\text{”}$

For instance, assume we have a purchase order schema as shown in Fig. 2 which describes a purchase order generated by home products ordering and a billing application [13] and we have Q1 as above. Let say that this schema has two object partitions: one includes a ShipTo element with its descendants and another includes a BillTo element with its descendants. Instead of accessing the two partitions to find the answer to Q1, it is ideal to retrieve the answer from a related portion of data. This action is possible if the index is constructed semantically. Therefore, we introduce the concept of objects in constructing and processing XML data. It will accelerate the processing of XML queries by localizing the XML data into a small and relevant portion of data.

In this paper, we tackle the issue of iteration on a large index size to find matched trees, or matched sequences by introducing the Schema Index. The Schema Index usually has a smaller index size compared to an index built only based on the data. This will help to find important keys within a small index size which is the Schema Index built from the schema to find the answer from the Data Index which is usually bigger since it is built from the data.

$$\begin{aligned}Q1_a = \text{“}&/purchaseOrder/ShipTo[city \\ &= \text{‘}Melbourne\text{’} and\ state = \text{‘}VIC\text{’}]/name/Fname\text{”}.\end{aligned}$$

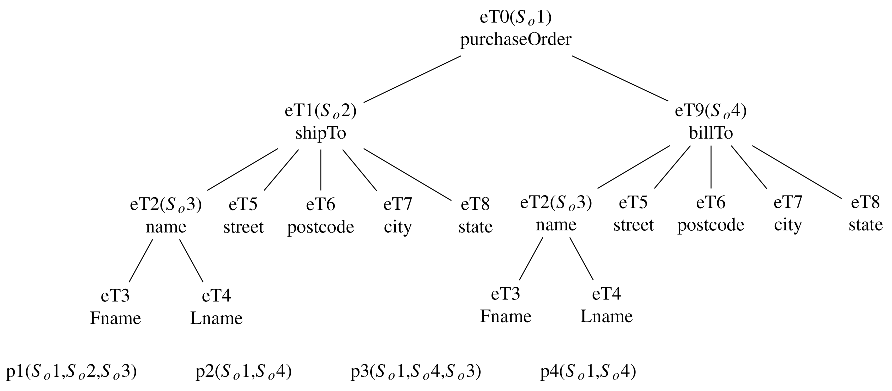


Fig. 2. Purchase order XML Schema.

In addition, we aim to improve the performance of queries with a value predicate. Let us modify $Q1$ to $Q1_a$ to include value predicates as above. This query is a combination of content and structural constraints and it aims to find all customers who sent item by shipment to Melbourne, Victoria. A particular customer will be retrieved using the structural constraints of the query and only a specific part of the customer's information will be finally retrieved utilizing the content constraints, which is also known as a value predicate. It is not practical to query without value predicates for transactional queries over a large collection of data where each collection can contain up to millions or more of kilobytes. Therefore, an efficient method to index content or value is necessary to improve the performance of XML queries with value predicates. However, in order to address queries with value predicates, content constraint alone is not enough. A knowledge of the whole path query is also required since the value predicate will not be standing alone without the existence of the whole path query. The significant characteristics embedded within $Q1_a$ are:

- The underlying semantic structure between a set of interconnected nodes.
- The path connectivity between each node.
- The content constraints of the predicates.

Therefore, it is important to take advantage of these query characteristics in optimizing query performance. Since the nature of XML data and schema structure is rich in semantics, identifying and leveraging the semantic connectivity from the data and schema in query processing are beneficial in improving query performance.

We aim in this paper to achieve the following targets:

Construction design:

- Exploiting the semantics of XML Schema and data in building our index.

Query processing:

- Introducing an Object-based intersection technique, which trims the search space of a large data repository based on the knowledge of structures and content derived from XML Schema.
- Eliminating irrelevant portions of data of our index by discarding unmatched paths and irrelevant objects.

In this paper we run different experiments to support our methodology. We demonstrate by the experiments that our method improves the query performance in terms of CPU cost including the cost needed to match XML query patterns and the cost of joins. We show that our method scales up efficiently when processing large XML data size. We measured the scalability when processing a query with a different number of XML tree branches. An evaluation of the search quality is undertaken to show the high level of precision and recall of the query results generated using our indices.

3. Related work

Broadly, a tree model captures the hierarchical structure of an XML document. XPath or XQuery are determined by path expressions to navigate the documents with various structures. The path expressions of the query could constitute a tree structure, to be called a twig query. Different research efforts have been spent on relevant processing techniques of the path expressions [4,14,15]. Their methods rely on identifying the elements which meet the tag or path requirements, and then to expedite the complex processes, they perform specially-designed encoding schemes, algorithms, or data structures including indices or stacks. The search on XML data can be classified into a structure search and a content search.

3.1. Structure search

Structure Index is used for structure searches to retrieve matches on a query tree whose tags and relationships consist of parent–child (P–C) relationships, ancestor–descendant (A–D) relationships or a mixed type of relationships. This type of index can be classified into a sequence-based index, a path-based index, a node-based index and a semantic-based index.

3.1.1. Sequence-based index

A sequence-based index transforms both XML data and path queries into a certain type of sequence, then performs sequence matching to process the query sequence over the data sequence in a top-down or bottom-up manner. In the top-down index, such as the Virtual Suffix Tree (*ViST*) [8], the process of querying XML paths is a matter of finding subsequence matches of the path query in XML data since each of them are transformed into structure-encoded sequences. Each sequence consists of pairs of an XML node and its prefix path within the XML data tree. In ViST, the structural information of the data is integrated with the content into a sequence. One of the advantages of this method is its ability to handle the query flexibly without join operations. However, it suffers from the existence of false alarms in some query results because a matched sequence is not necessarily a tree match. Also, its top-down transformation affects the query processing negatively since it produces a large number of paths needed to be evaluated during the subsequence matching for non-contiguous tag names [16]. To overcome the limitation of ViST, *PRIX* which is a bottom-up index, has been proposed [9]. PRIX proposed a bottom-up transformation approach where XML data and queries are transformed into a sequence of labels by the Prüfer's transformation method. Thereafter, subsequence matching is done on the sequence-based index. To obtain efficient query results, a series of refinement phases, namely (i) subsequence matching filtering, (ii) connectedness refinement, (iii) structure refinement and (iv) leaf node refinement, needs to be done after the subsequence matching.

3.1.2. Path-based index

Another type of XML index is a path-based index which can be classified into *a path summary-based index* and *a bi-similarity-based index*.

The path summary-based index is based on summarizing the paths of XML data to construct the index. The *DataGuides* index [17,18] summarizes all the unique paths in XML data starting from the root. There are two DataGuide index types: (i) a minimal DataGuide which has less traversal paths, thus its index size is compact, and (ii) a strong DataGuide where every label path in the XML data is described exactly once in the DataGuides. Despite the fact that DataGuide is a concise and accurate summary index that reduces the portion of scanned data for queries, it lacks the ability to answer branching queries. The Index Fabric [19] is a disk-based extension of the Patricia trie with the same scaling property but in a more balanced and optimized way for disk-based access. Each path in the XML data is encoded using unique designators in the Index Fabric. The designators are special characters which need to be interpreted using designator dictionaries which are used to map between designators and the element tags. The main drawback of this method is that the information on non-leaf nodes i.e. XML elements without data values, is not stored. Therefore, partial matching queries cannot be handled [5]. *ToXin* [20] also collects the values of XML data in a Value Index in addition to summarizing all forward and backward paths of XML graphs in a path index. The Value Index contains values and its corresponding nodes. The path index consists of the index tree corresponding to DataGuide [17] and an instance function for each edge of the tree index. ToXin navigates down, navigates up and filters an XML query to produce a set of nodes that match a set of query nodes and relationships over value predicates. Since this approach uses DataGuide and the edge approach to join paths, it does not keep the hierarchy information to answer complex twig queries. Unlike ToXin, *CTree* [21] does not provide only path summaries at the XML document level but also at the group level. It also provides details of child–parent links at the element-level. In addition, CTree [21] has multiple value indices per each data type of XML data including (List, Number, DTime). All the value indices support a search (value, gid, input parameter) operation where gid indicates a certain group of CTree. By determining gid, irrelevant groups are eliminated in order to evaluate value predicates. Therefore, the I/O cost is low. CTree can only efficiently handle XML documents with regular groups. However, in the case of an XML document containing lots of irregular groups, the index space will rapidly rise, due to the need for element-level links for each element. The RootPath and DataPath index [22] can evaluate XML twig queries with value predicates to be tightly integrated with a relational database query processor. In the RootPath index, the prefixes of the root-to-leaf paths are indexed. It is a concatenation of the leaf value and the reverse of the schema path, and it returns the complete node ID List. In contrast, the *DataPath* index stores all the sub-paths of the root-to-leaf paths. In fact, the DataPath index is bigger than RootPath, due to the duplication of the schema paths and the node ID of its structure. The increased size of the index tends to rise accordingly with the increase in XML documents size. To overcome this shortcoming, [22] explored lossless and lossy compression techniques to reduce the index sizes.

The *bi-similarity-based index* is constructed based on bi-similarity from the root element to the indexed element and consists of two types, namely the forward bi-similarity index as **1**-*index*, $\mathbf{A(k)}$-*index* [23] and $\mathbf{D(k)}$-*index* [24] and the forward and backward bi-similarity index as *F&B-index* [25] and $\mathbf{(F + B)^K}$-*index* [26]. In 1-index, XML data is partitioned into equivalence classes based on its backward bi-similarity from the root element to the indexed element. In 1-index, each root path represents only one element in the graph. However, there is possibility that an element is reachable by multiple paths because the index may contain similar root paths [16]. Another drawback of this method is reported in [24] that the 1-index is considered inefficient when indexed data is large and irregular due to its structural summaries being too big. The $A(k)$-index is based on k-bi-similarity. XML data elements are grouped together based on local similarity structure. As the $A(k)$-index groups XML elements based on the incoming paths of lengths up to k, it is able to efficiently evaluate path queries of length no longer than k by the automaton evaluation method. Otherwise, the validation step is necessary of nodes with false positives against the source data. The validation can be done by a reverse execution of the automaton on the data beginning with each node with a false positive. Unlike the $A(k)$-index which is static structure index, the $D(k)$-index is a dynamic structure index. It is a generalization of the $A(k)$-index since it applies the concept of bi-similarity. However, it has an adaptive structural summary for general graph-structured data. This means, it has the ability to support a given set of frequently used path expressions by adjusting its structure based on the query workload. A smaller k can handle shorter path expressions while a larger k handles longer path expressions. The experimental evaluations report that this adaptive structural summary outperforms other static ones for evaluation path queries. The F&B-index [25] is a "covering index" that is created by inversing XML data edges to obtain structural summaries of in-coming (forward) and out-going (backward) paths. In other words, it partitions XML data elements into equivalence classes based on their F&B-bi-similarity which is opposite to the 1-index, the $A(k)$-index and the $D(k)$-index which partition only on forward bi-similarity. This approach is superior to other approaches due to its ability to support branching path expression queries. However, in practice, the size of the F&B index is huge and may approach the size of the base XML database itself. Therefore, on this large index, there is definitely little query performance gain because it is similar to evaluating a query on the base data. Optimization becomes an essential research issue in the covering index because, for instance, the evaluation of queries consisting of many ancestor–descendant "//" relationships remains expensive to perform with the F&B index. $(F + B)^K$-index is an enhanced version of the F&B index because the size of the index is controlled by determining the value of "k" [26]. Limited classes of branching path queries can be covered if the index has a low value of "k". However, the index size is reduced compared to the F&B index. On the other hand, a wide range of query classes can be covered with a high value of "k". However, the index size in this case is larger.

3.1.3. Node-based index

The third type of Structure Index is the node index which can be further classified into the *numbering-based index* and *multi-dimension-based index*.

The *numbering-based index* relies on labeling XML nodes to represent the position of nodes within a document. Several labeling schemes have been proposed and surveyed in [27]. We reviewed the indices that use the *range-based sub-tree index*, the *regional-based sub-tree index* and the *prefix-based labeling*.

The *range-based sub-tree index* is used to determine the ancestor–descendant relationship in $O(1)$ time between any pair of XML elements. Dietz [28] proposed the first range based encoding scheme based on tree traversal order. The proposition of this method is that for two given nodes x and y of a tree T, x is an ancestor of y if and only if x occurs before y in the pre-order traversal of T and after y in the post-order traversal. However, its drawback is the need to re-compute the post-order and pre-order for each new insertion. Therefore, Li and Moon in XISS [29] proposed a new range-based index to overcome the limitations of Dietz's method. Their method is based on $\langle order, size\rangle$ to enable graceful node insertions. XISS is used $\langle order, size\rangle$ to create an element index, attribute index, and Structure Index. The element and attribute indices are implemented by $B + -tree$ index with the name identifier (nid) as keys. The Structure Index stores a collection of linear arrays, each of which contains fixed-length records for all elements and attributes. The other two components of the system are the name index to store name strings and the value table to store values. To evaluate path queries, XISS decomposes a path query into several simple paths, each of which generates an intermediate result that can be joined together by either EA-Join, EE-Join, or KC-Join to obtain the final result. However, large intermediate results lead to delay in the query performance when producing large final results. Also, another limitation is when all the reserved spaces have been consumed and a new node insertion is needed and, a global reordering is required.

The *regional-based sub-tree index* consists of a pair of (start position, end position) of the substring of XML data counted from the start of the XML document on depth-first traversal. Zhang et al. [30] utilized this sort of encoding to build an index using inverted lists. Two types of indices are used to process containment queries: (i) text index (T-*index*) and (ii) element index (E-*index*). The occurrence of an element or a word is recorded in the inverted list and indexed by its document number, position and depth. E-index consists of $\langle docno, begin : end, level\rangle$ and T-index consists of $\langle docno, wordno, level\rangle$. Despite the effectiveness of this labeling scheme in determining the relationship between nodes, it is unable to handle frequent updates of XML data since it is based on the assumption that the node positions are never changed once they are assigned.

The *prefix-based index* is able to precisely indicate an ancestor of a node on its path. It encodes each node on a certain path from the root down to that node. Therefore, the relationship can be identified between a given node x and its ancestor as y iff label(y) is a prefix of label(x). For instance, 1.4.2 is the parent of the child 1.4.2.8. Dewey labeling is an example of a prefix labeling scheme that was initialized for general information classification. Thereafter, [31] utilized the idea of Dewey labeling in XML data as for each node, there is an associated vector of numbers that represents the ID of the node in a path from the root to that node by including its ancestors coding as a prefix and it also includes the node number within its siblings of the same parent. The level can be determined implicitly by counting the number of codes separated by dots. [32] propose *ORDPATH*, which is similar to the Dewey ID [31], but the ORDPATH label differs from the Dewey ID in that the ORDPATH label uses only odd numbers in its coding and reserves even numbers for further node insertions. Prefix-based labels are much easier to update than range-based labels because only the nodes in the sub-tree rooted at the following sibling need to be updated when a new node is inserted [31]. However, when the depth of a tree increases, the size of the label increases.

In the *multi-dimension-based index* in [33,34], the author mapped every element node onto the two-dimension plane, using its pre-order rank on the x-axis and, its post-order rank on the y-axis. The context node and the four major axes of the XPath steps (descendant, ancestor, following, and preceding) divide the two-dimension space into four document regions, each corresponding to one major axis. Given a context node, the process of calculating one of its axes can be simplified as partitioning nodes on the two-dimension plane and retrieving the nodes falling into the region corresponding to the specific axis in a query. A new index structure, XPath Accelerator, has been proposed to support the multi-dimension approach. Although it could be implemented using a relational database system, it will benefit greatly if the underlying database supports spatial indexing techniques such as R-tree. The resulting set of nodes of the four major axes will be combined to support all path expression axes. Optimizations can be performed to further reduce the size of the document region in a query, though the pre-rank and post-rank in some cases of node insertions have to be re-calculated.

3.1.4. Semantic-based index

The only index that is semantic-based for processing value predicates of twig queries is TwigTable [35]. To evaluate the queries, *TwigTable* stores values in semantic-based relational tables whereas the internal structure of XML documents is stored in inverted lists [35]. In this approach, structural join algorithm is used to maintain the inverted list while the relational database processor maintains the tables. The semantic-based design of the tables results in performance advantages of TwigTable. On the other hand, the limitation of this approach is when a query does not have value predicates, no semantics will be applied and merely a structural join algorithm is performed.

3.2. Content search (keyword search)

When users do not need to know the structure of an XML document, the keyword-based search provides a friendly user environment to query the document. However, the returned results should ensure that they satisfy the structure of the XML document. A keyword search is somehow similar to content retrieval in IR technology. Various approaches have been proposed to identify relevant keyword matches. We divide these approaches into three categories. The first is the *LCA (Lowest Common Ancestor)-based approach*, which connects matches of the keyword and uses different LCA techniques to identify relevant matches. A number of LCA-based approaches have been proposed, including *XSEarch* [36], *MLCA* [37], *CVLCA* [38], and *MaxMatch* [39]. The second is the *statistics-based approach*, which determines relevant matches based on the statistics of the data such as XReal [40]. The third category is *minimal tree/graph-based approach*, which considers keyword matches in sub-trees/subgraphs of the data that contains all or part of the query keywords [41,42].

3.3. The query process approaches

In this section, we review different proposed methods in processing an XML query. The existing approaches are classified into structural joins and holistic joins.

3.3.1. The structural join approach

The *list-based approach* [30] proposed the first structural joins algorithm known as the *MPMGJN* (Multi-Predicate MerGeJoiN) algorithm. In XML data, each mode is encoded with the region labeling scheme i.e. (start, end, level). To evaluate a query as "a/b", structural joins will be done by comparing two lists of nodes, aList and bList consisting of occurrences of the nodes with tags a, b respectively. Two cursors, cur_a and cur_b, pointing to the head of the aList and bList respectively, are created. The core limitation of this approach is accessing a node several times during the matching [4,43].

The *stack based approach* [44] improved MPMGJN by utilizing a single stack to operate binary structural join. To evaluate a query "a/b", the algorithm pushes the node with tag a into a stack and when it finds the node tagged b, it tries to produce the result with the existing node tagged a. The advantage of this approach is that each node in the input list is scanned only once in contrast to MPMGJN. However, both the MPMGJN and the stack-tree approaches decompose queries into multiple binary relationships which might generate large intermediate results.

3.3.2. The holistic joins approach

Unlike the structural joins approach, in the holistic joins approach, the query is evaluated holistically without decomposing it into multiple binary relationships. Therefore, this approach can reduce the intermediary join results. [45] proposed novel approaches called *PathStack* and *TwigStack* to minimize the intermediate results in the memory. In their approach, each individual query node is associated with its own stack. Each node is pushed into a stack with a pointer to the closest ancestor in the parent stack. The structure of multiple stacks and pointers maintain the ancestor–descendant relationship easily. The difference between the two methods is that PathStack is proposed for a linear path query; while TwigStack, which is an extended version of PathStack aims to evaluate a general twig query. The main limitation of their approach is that if relational data is published as XML data, there will be an explosion of intermediate data because the stack-based structure is not supported efficiently if the workload does not involve document recursion [43]. The extension version of the holistic approach has been proposed using pipelining, joining multiple inverted lists at one time based on B+-tree indexes to eliminate intermediate results [46].

Twig$_2$ Stack [47] employed hierarchical stack encoding to evaluate a twig query and a Generalized Tree Pattern (GTP). For each query node, a hierarchical stack HS_q is maintained. HS_q contains a list of stack trees sorted in post-order. A stack tree is an ordered tree whose nodes are a stack of data nodes. Each node in the stack has a list of pointers to the top of the stacked nodes matching the query relationship. Twig2Stack evaluates a query in a bottom-up manner to reduce the double phase of TwigStack to a single phase. [48] argued that the limitation of Twig$_2$Stack is its expensive memory usage, due to it keeping all query leaf matches in memory until the tree is completely processed, as they could be part of a match. In addition, there is a complication in maintaining the ancestor–descendant relationship in a complex hierarchical stack structure.

TJFast [49] evaluates a query on data encoded by extended Dewey labeling which helps in deriving all element names along the path from the root-to-the element. TJFast is a holistic twig join algorithm that only needs to access the leaf node label of a query. Therefore, disk access will be reduced compared to other holistic approaches. However, TJFast possibly outputs many path solutions that do not contribute to any final answer [50].

Several indexing schemes have been proposed in order to improve the performance of XML path queries. However, despite the past efforts, the focus was mostly on utilizing an index to assist in effectively processing the structural part in twig queries. However, these methods do not distinguish between the structural and content search. Leaf nodes with values and internal nodes without values in XML data have different characteristics, thus, processing the content in the same way as processing the structure will lead to expensive structural joins to search for content. Add to this shortcoming, the semantic information of XML data has been ignored in most previous studies for either value or non-value nodes. Therefore, the limitation of the previous approaches is that they scan unrelated portions of data that are semantically related to the query. This work eliminates these shortcomings of the past approaches by proposing a new XML data indexing and query processing method which leverages the semantics of XML data into its indexing construction for better query performance.

4. Preliminary knowledge

In this section, we cover some important knowledge that needs to be clarified before presenting the proposed method. As mentioned before, we deployed our previous work [11] as a preprocessing step before indexing XML data. Thus, in the discussion on object-based XML data partitioning in this section, we introduce the importance of employing the concept of object in our work and we state the definitions of the object and the object partitions. After this, the models of XML data, Schema and queries are defined as preliminary knowledge which needs to be understood before introducing our indexing methods.

4.1. Object-based XML data partitioning

We adopt an object-based XML data partitioning technique, called OXDP [11]. In particular, OXDP contains a set of rules that can discover useful semantic information and identify objects within an XML Schema. XML data has rich features and consists of a variety of nested objects in its structure. Hierarchies are the primary characteristics of XML data. Thus, OXDP preserves these while enhancing its efficiency through optimization and semantic improvement.

The XML semantics basically envisioned in our work relate to the XML features that enable us to identify XML data based on the meaning of their tags in addition to the relationships between the tags. Such identification facilitates grouping and partitioning relevant data in order to provide semantically structured data. Since XML Schema or XSD represents XML objects and their semantic structure and provides the ability to define and manipulate XML document context, it has been selected as a feature of our proposed partitioning method. Our semantic model of XML in OXDP, called XTree, is very important particularly for object identification, establishing logical relationships, associations, aggregations, and generalizations before partitioning begins. Definitions 1 and 2 are used to build the concept of objects in our method.

Definition 1 (*Object*)**.** An object of an XML document is defined as a complex element type of XML Schema associated with that document. In other words, an object is a non-leaf element that consists of simple or other complex elements.

Definition 2 (*Object-based Partition (Opart)*)**.** An Object-based Partition is a partition of XML data that consists of a single object or multiple or nested objects.

4.2. Schema and data model

Both XML Schema and data are modeled as large, ordered *node-labeled trees* $T(N, E)$ where each node $n \epsilon N$ corresponds to an XML element and each edge between the nodes $(n_i, n_j) \epsilon E$ is used to identify the containment of node n_j under n_i in T. Each leaf node ln_i of XML data contains a value denoted by $value(ln_i)$.

Fig. 2 shows a purchase order schema which describes a purchase order for a home products ordering and billing application generated by W3C [13]. Fig. 3 shows the XML data tree corresponding to the schema in Fig. 2. Both trees have elements connected by edges but the data has values in each leaf node.

4.3. Query model

The focus of our work is on *XML queries with or without value predicates*, either a simple path or with branches. An XML query Q consists of nodes, labeled edges, and query predicate $Q(N_Q, E_Q, P_Q)$ where each node $q_i \epsilon N_Q$ represents a query tag that adheres to a set of XML document elements. A labeled edge between two nodes $(q_i, q_j) \epsilon E_Q$ indicates a structural constraint, which involves operators "/" and "//" denoting a "P–C" parent–child relationship and an "A–D" ancestor–descendant relationship respectively. A query predicate is held between brackets "[ ]" in query Q including other structural constraints and a filter of content constraints. The filter of content constraints evaluates true based on the corresponding XML document nodes. A list of N-ary tuples is generated to produce a final result of matching Q to the XML document D, where N is the number of query tags and each tuple $(n_1, n_2, \ldots, n_k)$ contains the XML document nodes $n_1, n_2, \ldots, n_k$ which identify the matched results of Q in T.

Query predicate: A query predicate P_Q is a combination of all or some of $(N_Q, E_Q)|(N_Q, E_Q, V_{PQ})|(N_Q, E_Q, V_{PQ}, P_Q)$ where each

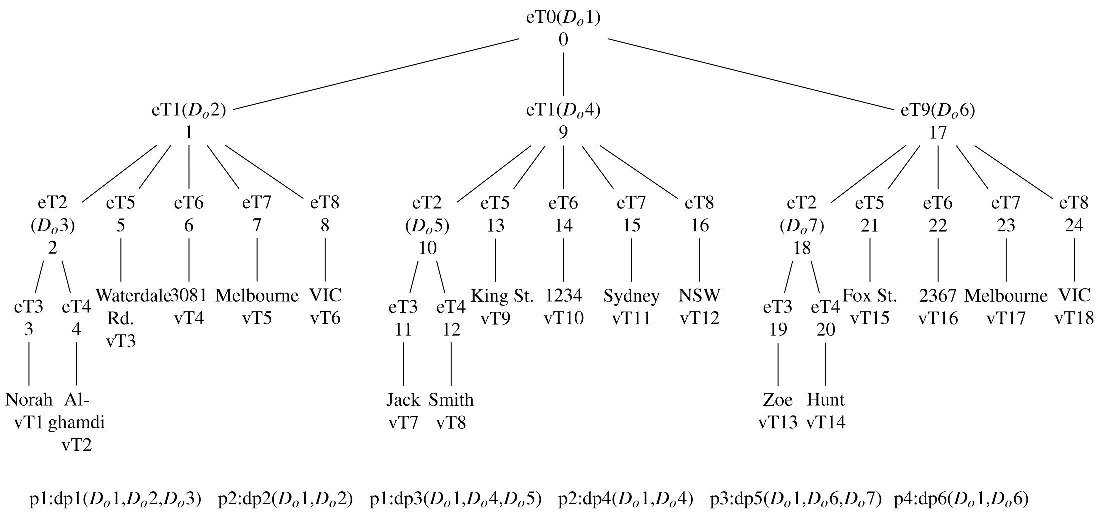


Fig. 3. Purchase order XML data.

$q_i \epsilon N_Q$ is a query tag within the predicate brackets, each $e_i \epsilon E_Q$ is an edge between two query tags, and each $v_i \epsilon V_{PQ}$ is a value that can match the value of leaf node ln_i at the data model and P_Q is another predicate representing a branching point.

Consider a query $Q2 =$ "//*shipTo*[/*state* = '*VIC*']/*name*" where "shipTo", "state", "name", "/", "//" are structural constraints and 'VIC' is a content constraint. "[/*state* = '*VIC*']" in $Q2$ is called a predicate which is a content constraint in this query and can be a combination of content and structural constraints.

5. Theoretical framework object-based Content and Structure XML indexing

This section explains the utilization of the semantics of the structure and the content of XML data and schema during the index construction phase. The Structure Index is introduced first to maintain the structural constraints of XML queries. Thereafter, we represent the Content Index which is proposed to improve the performance of querying constant values within XML data. Our methodology takes into account exploiting the semantic nature of XML data to improve query performance. In order to achieve this goal, we adopt OXDP as the pre-processing phase before constructing the index. In this paper, we utilize such rules in determining XML document's objects and then partitioning the data based on the discovered objects (refer to [11] for more detail).

As can be seen in Fig. 1, the system goes through three important stages: (i) building the knowledge of XML Schema Objects using OXDP as in [11], (ii) index construction where the system leverages the notion of objects to construct its indices, namely the Schema Index, the Data Index and the Value Index; and (iii) query processing to process a user-query either a simple path query or a twig query. The system utilizes the notion of objects to create an XML index in order to trim the search space and gain better performance for XML queries.

In Figs. 2 and 3, shipTo and its descendants are considered Opart1 in our example and billTo and its descendants are considered Opart2. Afterwards, tokenizing all distinct elements of XML Schema as well as tokenizing all distinct value inside XML data take the first place of constructing our indices. In a schema or a data, attributes and their associated values are treated as simple elements with values.

Definition 3 (*Element Token (eT)*)**.** An element token is an identifier that encodes each distinct element's tag name of XML Schema.

Definition 4 (*Value Token (vT)*)**.** A value token is an identifier that indicates each distinct leaf element's value of XML data.

In Fig. 2, each element of the schema is tokenized. For instance, the schema elements: "purchaseOrder", "shipTo", "state" and "postcode" have "eT0", "eT1", "eT8" and "eT6" as their tokens respectively. Element tokens represent all elements in the schema or the data. In contrast, value tokens are created only from the data with values. In Fig. 3, the values "Waterdale Road", "3081" and "VIC" are tokenized to "vT3", "vT4" and "vT6". Both eT and vT are implemented as integers to eliminate the computational overhead caused by the string comparisons.

Preliminary works were published in [51,52], however the work has been extended here with an integration model to prove the scalability of the proposed indices with the increase of the data size and the semantic connectivity of the underlying XML documents. The following subsections describe the Structural indexing and the Content indexing.

5.1. Structural indexing

Structure Indices, composed of the Schema Index and the Data Index, are proposed mainly to cope with the structural part of XML queries in an efficient way. These indices can evaluate arbitrary query structures including "/" or "//" as well as branching queries. As mentioned above Fig. 2 shows the schema tree for the PurchaseOrder schema and it consists of two object partitions as outputs of OXDP. The object partitions are Object1 (purchaseOrder(shipTo(name(Fname, Lname), street, postcode, city, state))) and Object2 (purchaseOrder(billTo(name(Fname, Lname), street, postcode, city, state))).

5.1.1. Schema Index

A *Schema Index* consists of element tokens of the schema associated with the Schema Object and the Path of Schema Object that consists of the ID of the object-based partitions. Table 1 shows the components of the Schema Index of the purchaseOrder schema. The components of each index will be presented as the following definitions.

Definition 5 (*Schema Object* (S_o))**.** A Schema Object is a set of tokens including an element token of a parent element tag, which is a complex element of XML Schema, alongside the element tokens of its children tags. Consider S_o as the set of element tokens S_o ($eT_{parent}, eT_{child(1)}, \ldots, eT_{child(k)}$), where eT_{parent} is the element

Table 1
Schema Index for the PurchaseOrder schema.

Tokenized tags				Schema object	
Tags	Tokens	Tags	Tokens	S_o	Tokens
PurchaseOrder	eT0	Lname	eT4	S_o1	eT0 eT1 eT9
ShipTo	eT1	billTo	eT9	S_o2	eT1 eT2 eT5 eT6 eT7 eT8
State	eT8	Fname	eT3		
Name	eT2	postcode	eT6	S_o3	eT2 eT3 eT4
Street	eT5	city	eT7	S_o4	eT9 eT2 eT5 eT6 eT7 eT8

Path of schema object		
Path	Schema object	Objects
P1	S_o1 S_o2 S_o3	Obj1
P2	S_o1 S_o2	Obj1
P3	S_o1 S_o4 S_o3	Obj2
P4	S_o1 S_o4	Obj2

Table 2
Data Index.

Data Index 1: path of data object		
dPaths	Path of S_o	Path Of D_o
dp1	S_o1 S_o2 S_o3	D_o1 D_o2 D_o3
dp2	S_o1 S_o2	D_o1 D_o2
dp3	S_o1 S_o2 S_o3	D_o1 D_o4 D_o5
dp4	S_o1 S_o2	D_o1 D_o4
dp5	S_o1 S_o4 S_o3	D_o1 D_o6 D_o7
dp6	S_o1 S_o4	D_o1 D_o6

Data Index 2:		
Opart 1		
Data Object	Tokens	XML data position
D_o1	eT0 eT1 eT1 eT1	0 1 9 17
D_o2	eT1 eT2 eT5 eT6 eT7 eT8	1 2 5 6 7 8
D_o3	eT2 eT3 eT4	2 3 4
D_o4	eT1 eT2 eT5 eT6 eT7 eT8	9 10 13 14 15 16
D_o5	eT2 eT3 eT4	10 11 12
Opart 2		
D_o6	eT9 eT2 eT5 eT6 eT7 eT8	17 18 21 22 23 24
D_o7	eT2 eT3 eT4	18 19 20

token of the parent node, $eT_{child(i)}$ is the element token of the parent associated children within the schema and k is the number of the parent's children.

For example, Table 1, S_o1 in the Schema Object table consists of the tag tokens that connect the root "purchaseOrder" with its children ("shipTo", "billTo") represented by tokens eT0, eT1 and eT9. Only distinct S_o is stored in the Schema Object in Table 1. For instance, S_o3, which represents "name" elements as the root with its children, is stored once. It is important to highlight that in the Schema Object, we represent the object as a parent and its direct children tags and not its descendants. For instance, "name" has "Fname" and"Lname" as its children which have not been included in S_o2 and S_o4.

Definition 6. Path of Schema Object (p): The Path of Schema Object is a set of S_o located on the same path from the root to a leaf node of the XML Schema.

In the same table, the Path of Schema Object "p1" is a set of Schema Objects S_o1, S_o2, S_o3 located in the same path from the root "purchaseOrder" to the leaf nodes "Fname" and "Lname" including all elements in between such as "shipTo" and "name".

5.1.2. Data Index

Another index is the *Data Index* which consists of two indices. In the first index (Data Index 1), each Path of Schema Object is associated with all its corresponding Path of Data Object. It can be represented by Definitions 6 and 8. The second index (Data Index 2) consists of all Data Objects "D_o" grouped by each object partition defined in 7. Table 2 shows the Data Index of purchaseOrder XML data.

Definition 7 (*Data Object D_o*)**.** A Data Object is a pair of a set of element tokens associated with a set of their positions inside XML data. Consider $D_o\,(eT_{parent}, eT_{child(1)}, \ldots, eT_{child(k)}, Pos_{parent}, Pos_{child(1)}, \ldots, Pos_{child(k)})$ where eTparent is the element token of the parent node, $eT_{child(i)}$ is the element token of the parent's associated children and k is the number of the parent's children within XML data.

For example, consider "D_o2" in Table 2 as an example of the Data Object for S_o2 (eT1, eT2, eT5, eT6, eT7 and eT8), which corresponds to the twig pattern ("shipTo", "name", "street", "postcode", "city" and "state"). The values 1, 2, 5, 6, 7 and 8 are identifiers of the XML data position of D_o2. Each D_o of the Data Object representing a specific S_o of the Schema Object should contain the same sequence of tokens determined by the S_o.

The positions of XML data are generated during a depth-first traversal of the tree and a number is sequentially assigned at each visit.

Definition 8 (*Path of Data Object (dp)*)**.** A Path of Data Object is a set of D_o located on the same path from the root to a leaf node of the XML data.

For instance, from Tables 1 and 2, "p1" consisting of "S_o1, S_o2, S_o3" in the Schema Index is corresponds to two Paths of Data Object "dp1" and "dp3" containing "D_o1, D_o2, D_o3" and "D_o1, D_o4, D_o5" as a set of Data Objects respectively.

5.1.3. Use cases: answering twig queries using the Structure Index

This section demonstrates how the Structure Index can be utilized to process the structural part of twig queries using our object-based approach. By applying our pruning strategy, the query execution times are found to be more optimized in comparison to earlier work in the area. This optimization technique will be described later.

Using the Structure Index to handle P–C relationship. In Fig. 2, "$Q1 = billTo[/city][/postcode]$" is used as an example to show how our indexing technique can be utilized to retrieve the answer to XML queries with P–C relationships from XML data. First of all, from the Schema Index, the nodes of twig query Q1 are converted into tokens using the tokenized tags as shown in Table 1. The query nodes are "billTo", "city" and "postcode" and their tokens are eT9, eT6, and eT7, respectively. eT9 is the root of S_o4 at the Schema Index in Table 1. eT6 and eT7 are the children of eT9 in S_o4. As long as eT6 and T7 with their parent eT9 are in the same Schema Object S_o4, the Path of Schema Object that ends with S_o4 is P4 which is the candidate path to assist in finding the data. Table 2 accesses the Data Index using the Path of Schema Object which assists in finding all $D_o(s)$ in the P4. It can be said that Path of Data Object "dp6" is a path of connected Data Objects and it is against the Path of Schema Object P4. The answer in Table 2 will be retrieved from Data Object2 by matching the tokens of Q1 with the tokens of the XML data in Data Object. For instance, in Q1, dp6 consists of S_o1 and S_o4 as a Path of Schema Object and D_o1D_o6 as a Path of Data Object at Data Index 1, each of which consists of tokens and XML data position in Data Index 2. As a result, eT9, eT6, and eT7 have one answer in XML data which is 17, 22 and 23.

Using the Structure Index to handle ancestor–descendant (A–D) relationship. In some cases, there is a need to navigate XML data to answer a query with A–D relationships. Our method is able to handle this case more efficiently. In Fig. 2, the $Q2 =$

Table 3
Schema Object Dictionary.

Token	(Path of Schema Object, Object)
eT1	(P1,Object1)(P2,Object1)
eT2	(P1,Object1)(P2,Object1)(P3,Object2)(P4,Object2)
eT3	(P1,Object1)(P3,Object2)
eT4	(P1,Object1)(P3,Object2)

shipTo[//*Fname*][//*Lname*] is used to demonstrate our technique handling of A–D relationship. eT1, eT3 and eT4 are the tokens of Q2 tags i.e."shipTo", "Fname" and "Lname", respectively. eT1 is the root of S_o2: eT1 eT2 eT5 eT6 eT7 eT8 in Table 1. It can be observed that S_o2 is in P1 and P2 in the Path of Schema Object. We search the descendant Schema Objects S_o3 to find eT3 and eT4. Therefore, the candidate path is P1. It can be seen that Path of the Schema Object connects all Schema Objects which are the twig patterns in the data at the Schema Index and Path of Data Object does the same job in XML data. Similar to use case 1, the Path of Schema Object "P1" is utilized to access the desired XML data positions. In the Data Index 1, "P1", which consists of $S_o1S_o2S_o3$, is equivalent to dp1, dp3, dp5 which consist of $D_o1D_o2D_o3$, $D_o1D_o4D_o5$ and $D_o1D_o6D_o7$ respectively. Thereafter, the data will be retrieved from the Data Index. The produced results should be 1 3 4, 9 11 12, and 17 19 20.

5.1.4. Optimization phase of the Schema Index

In order to speed up index processing further, we aim to avoid a large number of joins in the Schema Index to retrieve the targeted Path of Schema Object; the Schema Object Dictionary (S_oDic) in Definition 9 is proposed to link the tokenized tags, the Schema Object and the Path of Schema Object as an optimized step in processing the Schema Index. S_oDic reduces the number of Tokens, S_o and Path of S_o required for joining process. It is achieved by pruning S_oDic based on the intersection of object IDs.

Definition 9. Schema Object Dictionary (S_oDic) is an optimized dictionary for the Schema indices, consists of a combination of Tokenized Tags, Schema Object and Path of S_o. It links each token with all possible Paths of S_o and the object on which each path ends. S_oDic is represented as "(eT, (p, Opart))" where "eT" is the element token, "p" is the Path of Schema Object and "Opart" is the partition ID where the element exists.

Schema Object Dictionary in Table 3 consists of element tokens of the Schema associated with a set of pairs consisting of the Paths of Schema Object and ID of the object-based partitions. For instance, eT3, which is a token element of Fname, exists in P1 that is in Opart 1 and also exists in Opart 2 in P3. In the case of the element "name", it exists in Object1 at P1 and P2 and in Object2 at P3 and P4. If we would like to answer the path query: "name/Fname", the Object-based intersection process will be introduced in Section 6.2. It will chose candidates P1 and P3 for further processing as explained in more detail in the query processing Section 6.

The components of the Structure Index with the applicable use cases were introduced in the current section. We also showed how our method can process P–C and A–D relationships. Then, the Schema Object Dictionary was proposed as an optimization step to accelerate the search process at the Schema Index. The next section is a description of the Value Index that is used for the content searching of XML data. Note that even in the case when XML Schema is unavailable, our method is still effective, but we need to scan the document once to obtain the distinct structure of the data to build the Schema Index and then scan it again to build the Data Index.

5.2. Content indexing

To index the content of XML data, the Value Index is proposed. The Value Index is built from the Schema Index, the Data Index and XML data. It keeps the semantic connectivity between the nodes of XML data to produce an efficient performance for a query with value predicates. The construction of the Value Index begins with tokenizing all values and storing them according to their schema design. Secondly, the index stores all the value tokens with their corresponding Data Objects according to their data context.

As can be seen in Fig. 4, the Value Index consists of all the object partition identifiers associated with a set of the Path of Schema Object. Each path contains all the element tokens of the leaf nodes only. The value tokens of the corresponding element token are associated with their Data Objects.

Definition 10. Consider a Value Index as $VI = Opart_1, \ldots, Opart_k$ where $Opart_i$ is an object partition identifier and k is the number of object partitions. Consider for each object partition identifier of VI as $Opart_i = Leaf(p_1), \ldots, Leaf(p_m)$ where each $Leaf(p_j)$ is a Path of Schema Object exists in $Opart_i$ and m is the number of the Path of Schema Object. Let $Leaf(p_j) = eT_1, \ldots, eT_n$ is a Path of Schema Object consisting of a set of tokens of leaf elements in the XML data and n is the total number of element tokens. For each eT_i, there is a set of value tokens associated with its corresponding Data Objects $eT_i = \langle vT_1, \{D_{o1}, \ldots, D_{on}\}\rangle, \ldots, \langle vT_y, \{D_{o1}, \ldots, D_{on}\}\rangle$ where y is the number of value tokens and n is the number of Data Objects per each value tokens.

Fig. 4 depicts the Value Index in only "Opart1". The rest of the partitions will have similar representation. "Opart1" has "p1" and "p2" as its Path of Schema Object where each of them has the tokens of the leaf elements "eT3, eT4" and "eT5, eT6, eT7, eT8" respectively. It can be seen that "eT2" in "p2" was not considered since it is an element token of non-leaf nodes. The last level of this index is the association of the value tokens and its associated Data Objects $\langle vT1, D_o3\rangle$ and $\langle vT7, D_o5\rangle$ depicted in the same figure.

The redundancy of the data within an XML document increases the index size in most of the previous works. In our index, this issue has been taken into consideration as shown by Remark 1.

Remark 1. There is a single value token for all matched values located in the same object partition, the same Path of Schema Object and with the same element tokens.

By applying Remark 1, we save memory and reduce the search time. It will be more practical for data that often has a redundancy. For instance, on a small scale, let us say if purchaseOrder XML data has 10 shipTo names with the same shipTo address, it is more precise to ignore the redundant data of the address and record the address only once. On a large scale, this redundancy will negatively affect the performance of a query processed over the index. It is important to highlight that Data Objects associated with each value token will assign the position of that value's parent node within the data as the Definition 7. This feature will improve the efficiency of processing the query by trimming the search space in the Data Index later.

5.2.1. Use case: answering a query using the Value Index

With both the Schema Index and the Data Index, either single-path queries or branching queries can be answered as discussed before. In the following example, we can see the difference between evaluating the query with or without value predicates. For instance, consider $Q3 =$ "/*purchaseOrder*/*shipTo*[/*street*][/*state*] is a query without the value predicate. The element tokens of the query nodes are "eT0,"eT1, "eT5 and "eT8. From the Schema Index,"eT0 is associated with (p1, Opart1), (p2, Opart1), (p3, Opart2),

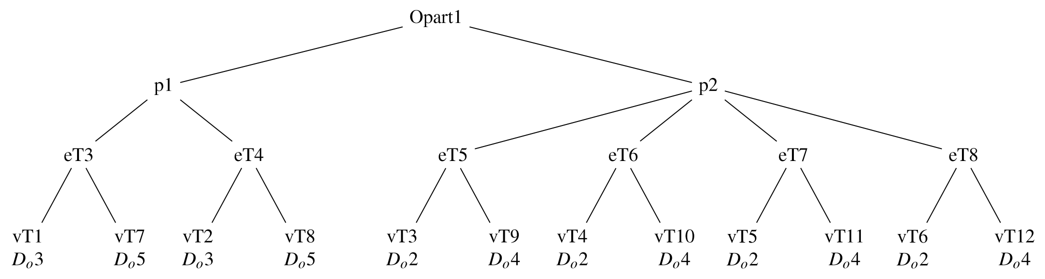


Fig. 4. Value Index.

(p4, Opart2), eT1 is associated with (p1, Opart1), (p2, Opart1) and eT5 and eT8 are associated with (p2, Opart1). To trim the search space, we undertake the object-based intersection on Opart between "eT0 and "eT1, then between the intersected result and each child of "eT1 separately i.e. "eT5 and "eT8. The final result from the Schema Index will be (p2, Opart1). Then, from the Data Index in Table 2, "dp2, "dp4 and "dp6 are retrieved based on the "p2 of the Schema Index in Table 1. Since "dp2, "dp6 and "dp4 consist of a set of Data Objects, the position of the query nodes will be retrieved from the Data Objects, which are located in Opart1, by matching the query element tokens with the element tokens of the Data Objects. The final results of the query would be "0, 1, 5, 8", "0, 9, 13, 16" and "0, 17, 21, 24" as can be seen from Fig. 3 and Table 2.

The Value Index in conjunction with Structure Indices can be used to evaluate arbitrary queries with different value predicates such as a simple value predicate, a single path ending with value predicates or a branched path ending with value predicates. The functionality of the Value Index will be discussed using this part of the previous query of $Q3$ "*shipTo*[/*street* = "*WaterdaleRd*. To evaluate this sort of query, we begin by performing the evaluation of the values before processing the structural part. In doing so, we reduce the total number of scanned elements because the Value Index groups values of an XML document within objects. By applying this semantic-based technique, the search will be trimmed semantically based on each object. In the given query, the Schema Index of "street" is eT5 associated with (p2, Opart1). This means that only Opart1 needs to be accessed. In addition, the index adds the Path of Schema Object as identifiers that assist in decreasing the search space of the values. In our example, the search space will be trimmed to those that have "p2" within the partition "Opart1". Another advantage is that instead of preceding an aggressive string search looking for matching values of the query condition, element tokens will eliminate irrelevant values. Thus, the condition '*Waterdale Rd*' will be mapped to its value token and then the token will be looked at from eT5 without the need to scan all the value tokens within the path "p2". The output of this index is the value token, which is "vT3" in the query, leading us to the right Data Object which is D_o2. The benefit in finding the Data Objects, i.e."D_o2", is that only related Data Objects will visit the Data Index to produce the final results.

In summary, the proposed indices have four features to facilitate the evaluation of twig queries in an optimum execution time. The indices are able to: (i) preserve the details of parent–children elements through the objects; (ii) preserve the details of all objects located in each path of the schema and data as in the Path of Schema Object and the Path of Data Object; (ii) partition and keep links between interconnected data based on object-based semantics and (iv) deploy the concept of object in evaluating XML queries with and without value predicates.

6. Query processing

The index construction takes place before the actual commencement of query processing. This section details the algorithms that process queries based on the concept of objects. It

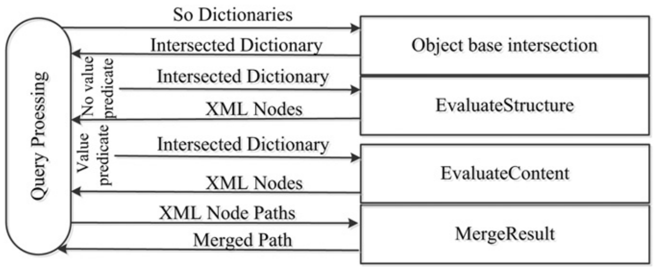


Fig. 5. Overview of the system algorithms.

shows how a twig query is constructed to build the structure of the root and children. Thereafter, it presents query processing algorithms and how they are interconnected with each other.

6.1. Query construction

A method called TreeQueryBuilder is invoked when a twig query is started. TreeQueryBuilder is used to analyze a user-entry query into nodes and identify parent and children nodes. A branched query is constructed into query nodes, each of which consists of a root tag, a root token and its children's tags and tokens. Twig query nodes are presented in our algorithm with the following definition.

Definition 11. Twig query nodes (Tqn) are a set of nodes associated with their token and represented as Tqn ($n_{root}, n_1, \dots, n_k, eT_{root}, eT_1, \dots, eT_m$), where n_{root} is the twig root node tag, each n_i is a child of n_{root}, k is the number of children in this twig and eT_j is the token of the node n_i and m is the number of children tokens.

6.2. Query process algorithms

Our algorithm "ProcessQuery" is a recursive function decomposing a branched path into multiple single paths. An overall view of the main algorithm and its interaction with other functions are shown in Fig. 5 and Algorithm 1. The algorithm starts by applying the intersection based on the objects from the Schema Object Dictionary "S_oDic" of all query nodes "qNode" located on the same query path at line 13. This intersection process will end up with an intersected S_oDic among all the query nodes within the path to help us use the information of the Path of Schema Object "p" and the object "Opart" where the search will be conducted in the next step. Then, at lines 4–8, it evaluates the path ending with a value predicate using the function "*EvaluateContent*" and the path without a value predicate is evaluated by "*EvaluateStructure*". Finally, "*MergeResult*" will be invoked at line 20 to merge the results after finding them by the evaluation algorithms.

Object-based intersection. In order to find the semantic connectivity between XML query nodes, the object-based intersection is invoked by the main method. The *Intersection* method, as shown

```
   Input: qNode, c_SI “current Schema Index”, path, depth
   Output: Query nodes position within the data
 1 if ¬qNode.Children then
 2     path.Add(qNode);
 3     i_SI ← Intersect(c_SI,SchemaIndex[qNode]);
       foreach p of i_SI do
 4         if qNode.ValuePredicate then
 5             xNums ← EvaluateContent(p,path,Opart)
 6         else
 7             xNums ← EvaluateStructure(p,path,Opart);
 8         end
 9         res.Add(xNums);
10     end
11     return res;
12 end
13 i_SI ← Intersect(c_SI, SchemaIndex[qNode]);
14 firstOccurrence ← true;
15 path.Add(qNode);
16 foreach c in qNode.Children do
17     temp ← ProcessQuery(c, i_SI, path, depth+1);
       if firstOccurrence then
18         result ← temp; firstOccurrence ← false;
19     else
20         result ← MergeResult(result, temp, depth);
21     end
22 end
23 return result;
```

Algorithm 1: ProcessQuery

```
   Input: $S_o$Dic , $S_o$Dictionary
   Output: iRs
 1 foreach $S_oD$ in $S_oDic$ do
 2     if ! $S_oDictionary.Contains(S_oD.object)$ then
 3         continue;
 4     end
 5     iRs.Add($S_o$D.object );
       temp=$S_o$Dictionary[object];
 6     foreach $S_o$ in $S_oD.PathOfS_o$ do
 7         if $temp.Contains(S_o)$ then
 8             iRs[object].Add($S_o$);
 9         end
10     end
11 end
12 return iRs
```

Algorithm 2: Object-based intersection

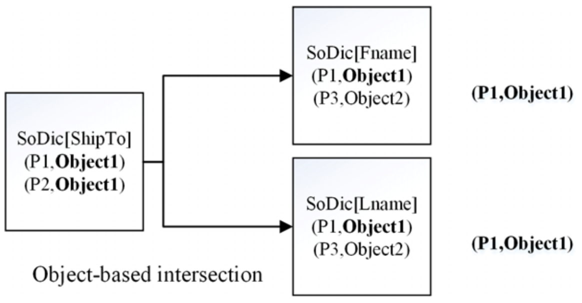


Fig. 6. Object-based intersection.

in Algorithm 2, is done between the query nodes of each root-to-leaf path. The first step of the algorithm is to intersect S_oDic of the query root with S_oDic of the whole data. This scenario handles cases of parent–child (P–C) and ancestor–descendant (A–D) edges inside the twig query as well as the links between objects since the Path of Schema Object keeps the interconnectivity between objects within a partition and among partitions. Fig. 6 is an example which shows how the intersection is performed, based on the concept of objects. The example has a branched path query:*shipTo[/Fname][/Lname]*. Processing this query starts by searching inside S_oDic using the root of the query, which is “shipTo”. The query is split into “shipTo//Fname” and “shipTo//Lname”. Table 3 shows S_oDic of each query node. Each S_oDic of the child/descendant query nodes is intersected with S_oDic of its parent/ancestor node. Fig. 6 shows the intersection between S_oDic[shipTo] and S_oDic[Fname] and the intersection between S_oDic[shipTo] and S_oDic[Lname]. The final result provides the candidate Path of Schema Object which is considered a key to access the Data Index to trim the search space, resulting in an accelerated query process. The search on the Schema Index is much faster than on the Data Index because the Schema Index is built from XML Schema and its size is smaller than the Data Index built from XML data.

Content and structure evaluation. After the intersection process is complete, both the *EvaluateContent* and *EvaluateStructure* search inside the Data Index to find XML node positions within XML data. Query paths ending with a value predicate are evaluated by the function “EvaluateContent” and others without a value predicate are evaluated by “EvaluateStructure”. These two functions are independent of each other, i.e. the whole path can be processed completely by only one of them without the need for the other. The difference between them is that EvaluateContent utilizes the Value Index to find the candidate Data Objects which hold the condition of the value predicate before proceeding the structural search. Thus, we can say that EvaluateStructure can do only the structural search whereas EvaluateContent can do both structural and content search. The main advantage of EvaluateContent is its capability to trim the search space of the scanned elements by searching of query values before the query structure. The details of EvaluateContent will be shown later. After evaluating the content and structure of the query and obtaining the XML nodes positions, the result will be merged through MergeResult which keeps the structural order of the node and produces final results.

EvaluateContent. Algorithm 3 can perform two main functionalities. The first is the content search which starts from line 1 and then embeds the structural search from line 8, continuing with Algorithm 5 at line 13. The content search uses the Value Index to retrieve only the participating Data Objects by filtering the value predicate using the information coming from the Schema Index as “*p*” and “Opart”. After this, a structural search will be done on these XML nodes that exist within the participated Data Objects. At line 8, it goes through the related portion of the first the Data Index that matches “*p*” to check the last D_o of the “dp” if it is matched with D_o coming from the Value Index as line 10. The search space is narrowed down to these Path of Data Object “dp” that end with that D_o of the Value Index. Then, continuing with Algorithm 5, each D_o of Data Index 2 will be visited with regarding to D_o of Data Index 1 as in line 2. It is well known that each D_o consists of the tokens of the nodes associated with their positions. At lines 4–5, if a query node token is matched with a node token of D_o, its position is added to a temporal result collector. Once the leaf node of the query reaches line 18, the temporal result will be added to the list of the final results.

EvaluateStructure. In Algorithm 4, the structural part of a path query is evaluated. At line 1, all the Paths of Data Object are retrieved from the Data Index 1. These paths are matched with the Path of Schema Object from the object-based intersection on the Schema Index. Then, the algorithm manipulates each Path of Data

```
Input: p:"Path Of Schema Object", quP: "query path", Opart:
        "Object partitions"
Output: XML nodes positions
foreach viObject in ValueIndex[OPart] do
    viPath = viObject[p];
    viToken = viPath[quP[quP.Count-1].Token];
    if !viToken[quP[quP.Count-1].vPred] then
    continue;
    $q_v$= quP[quP.Count-1].vPred;
    for $vT_i \in$ viToken[$q_v$] do
        d = viToken[$q_v$][i];
        PathsOfDo = DataIndex1[p];
        foreach dp in PathsOfDo do
            if dp[dp.Count-1] != d then
            continue;
            TmpR = new List (); k = 0;
            for $D_{o_j} \in$ dp do
                Continue Algorithm 5;
            end
            if ($R_t$.Count=quP.Count) Result.Add($R_t$);
        end
    end
end
return Result;
```

Algorithm 3: EvaluateContent

Object to obtain the participating Data Object starting from line 3. Then, using Algorithm 5, which is an extension of Algorithm 4, Data Objects will be retrieved from the participating objects from Data Index 2. Thereafter, each query node token will be checked with Data Objects' tokens to find the position of XML nodes. This function assists in eliminating non-participating paths and objects, thus, the search space is reduced.

```
Input: p:"Path Of Schema Object", quP: "query path", Opart:
        "Object partitions"
Output: XML nodes positions
PathsOfDo = DataIndex1[p];
foreach dp in PathsOfDo do
    for $D_{o_j} \in$ dp do
        Continue Algorithm 5;
    end
    if ($R_t$.Count=quP.Count) Result.Add($R_t$);
end
return Result;
```

Algorithm 4: EvaluateStructure

Linking objects. Our method has the ability to link object partitions should the query requires so. This can be done by the object-based intersection method which assists in preserving correctness when linking the object partitions. Practically, most XML queries require access to a single partition but in this technique, we aim to let our indexing and processing method fulfill different query requirements.

For instance, Fig. 7 and Table 4 show a simple example of XML data partitioned into two nested objects. This example has been chosen to elaborate the linking technique between partitions of nested objects. Our running example, purchaseOrder, does not have this structure type. Object2 of Fig. 7 is nested from Object1.

To do the link between partitions, a track of all siblings with the same parent is kept as ("A", "B"), ("B", "C", "E")("D", "F", "G"), and a track of all twigs in the same path from the root to the leaf also is kept ("A", "B")("A", "B", "D") where D is a nested object of B. If the query "B[/C][//F]" is executed, it can be seen that B and C are in object2 and D is in object3. Also A is located in S_o1 and B is located in S_o1 and S_o2 and D is located in S_o3. The path which connects S_o1 with S_o2 and S_o2 with S_o3, is "P2". Thus, this path connects the object2 with the object3. Our method is more efficient when querying from one object partition. However, this example shows that our proposed idea can work effectively even when the query is evaluated from different object partitions.

```
object=DataIndex2[Opart or viObject];
foreach $d_o \in$ object[$D_{o_j}$] do
    for $eT_i \in d_o$.Tokens & $k <$ quP.Count do
        if $eT_i$=quP[k].Token then
            $R_t$.Add($d_o$.Pos[$eT_i$]);
            if $k > 0$ then
                if quP[k].Tag[0] != '/' then
                    $y_p$=nDepth[$R_t$[$R_t$.Count-1]];
                    $y_c$=nDepth[$R_t$[$R_t$.Count-2]];
                    if $y_p$ != $y_c$+1 then
                        $R_t$.Remove($R_t$.Count-1);
                        continue;
                    end
                end
            end
            k++;
            if k=quP.Count&quP[k-1].leaf then
                if $R_t$.Count=quP.Count then
                    R.Add(new ($R_t$));
                end
                $R_t$.RemoveAt($R_t$.Count-1);
                k- -;
            end
            if k=quP.Count || quP[k].leaf then
            break;;
        end
        if !quP[k].leaf then break;;
    end
end
```

Algorithm 5: Find matched element tokens

In summary, our index leverages the advantage of the semantic workload of queries during the construction of its indices. This characteristic using XML data partitions coupled with an efficient linking technique among partitions to process queries will have a significant impact on the performance of XML queries. From this point of view, 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. Therefore, it is 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. Our method utilizes the object-based intersection in addition to its ability to discard irrelevant objects and the Path of Data Object to reduce query response time.

7. Experimental results

7.1. Experimental environment setup

A prototype system, including all the algorithms of our proposed method, was implemented using C#.Net. All XML indices in this paper were loaded into RAM before running the queries, thus the IO cost of reading the index data is not required. All the

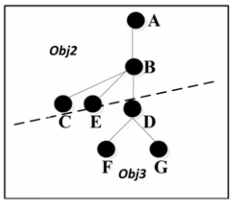


Fig. 7. Linking technique.

Table 4
Illustration of linking technique.

E	eT	S_o	Tokens
A	eT1	S_o1	eT1 eT2
B	eT2	S_o2	eT2 eT3 eT4 eT5
C	eT3	S_o3	eT4 eT6 eT7
D	eT4		
E	eT5	*Paths*	*Path of* S_o
F	eT6	P1	$S_o1\ S_o2$
G	eT7	P2	$S_o1\ S_o2\ S_o3$

Table 5
Different characteristics of each dataset.

Parameter	DBLP	Auction	Sigmod	Yahoo
# Nodes	3 332 131	157	11 527	30 210
Depth max.	6	5	6	5
# Fan-out	22	5	4	5
# Distinct E	36	32	11	32
Opart	8	2	2	2

experiments were conducted on an Intel Core 3.2 GHz P–C with 6.00 G RAM running Windows 7.

In order to study the performance of our Structure Index in processing XML queries, we compare it with the state-of-the-art TwigX-Guide [7]. TwigX-Guide has outperformed other twig querying systems, taking advantage of both the path summary in DataGuide [17] for efficient path queries with parent–child edges and the region encoding in TwigStack [45] in its ability to process twig queries. TwigX-Guide outperformed other well known methods, such TwigStack, TwigStackXB[10] [45], TwigINLAB [53] and TwigStackList [54], in most queries for the comparison done by [7]. It also has comparable results in other queries. TwigX-Guide is not publicly available at this time; it has been implemented based on their algorithm description.

In order to study the improvement in XML query processing with value predicates by our proposed indices, a series of experiments was conducted. Our Value Index performance is compared with the standard value index with a focus on the value predicates. The structural part of the query in both methods is the same. We use our Structure Index including the Schema Index and the Data Index to process the structural part of the query and the content part of the processing is done by both our the Value Index and the standard value index.

In most of the past research, the standard indexing method for values when value predicates exist in XML queries is to index each value with its node position ID. When performing joins, a small amount of node IDs will be returned for further joins. We compare our proposed Value Index with the standard value index.

7.2. Datasets

The datasets used in the experiment are shown in Table 5. These datasets were obtained from the University of Washington's XML repository [55]. The chosen datasets differ in their characteristics. The DBLP dataset has a shallow structure with a recursion in some of its element names. The document trees in the SigmodRecords and Yahoo datasets were used due to their nested structure. Additional information is provided in Table 5, including the maximum depth, the number of elements, and the number of object partitions for each dataset.

7.3. Experimental metrics

To evaluate the performance of our proposed algorithms, two metrics were used. The first metric obtains CPU cost by calculating the average execution time of a query. Secondly, the total number of scanned elements is measured during a joining process. This metric will provide a good reflection of the ability of our algorithms to trim the search space and to skip portions of the data.

7.4. Evaluation criteria

The performance of the Structure Index is compared with TwigX-Guide based on the type of relationships (P–C, A–D, or mixed). The average query processing times for a given twig query is calculated. As mentioned earlier, the query processing time is the time taken by our processing algorithm excluding the off-line stage i.e. building the Schema Index and the Data Index. Added to these criteria, we include simple paths and branch paths with values in the predicates.

7.5. Queries

Table 6 presents the evaluation queries. Each query is coded QXN, where X represents S (SigmodRecords), D (DBLP), or A (Auction Data), and N is the query number within the respective dataset. These queries have different characteristics in terms of the number of access objects, the depth of the path, the type of edge relationship and the twig structure. The experimental queries are divided into two groups: (i) queries without predicates in Table 6(a), and (ii) queries with predicates in Table 6(b).

The queries QS1 to QD6 in Table 6(a) have only P–C edges connecting their nodes and need to access only a single object. They are twig queries with a different number of nodes, their branches varying from 1 to 3. The query answer will be retrieved from a single object. In the queries, QS7–QD11 from the same table only have an A–D relationship between their nodes. Some need to access one object partition and others need to access two object partitions to retrieve the data. The maximum depth of retrieved data is 7 in QS7 and QS8, 5 for QD9 and 4 for QD10 and QD11. The queries from QS12 to QD17 concern the performance of the twig queries with mixed edges which is the hybrid type of relationship connecting between nodes (P–C and A–D). They also consist of a different number of nodes and branches. QS13 needs to access two objects to obtain the final results. However, the rest need to access one object to retrieve the answer of the query.

The queries in Table 6(b) are used to evaluate the Value Index. QD9 and QD11 are simple path queries whereas the rest are branch queries. QD6 and QS5 contain only the P–C relationship, while the rest contain hybrid edges of P–C and A–D. We have a variety of branch numbers in the branched queries. For example, QS1 and QS2 have two branches whilst QD3, QD4, and QD5 have 3, 4, 5 branches respectively. The type of value predicates is also different among the queries, QA9, QS5, and QD8 have path-value predicates while QA10, QS4, and QS5 have path-branch-value predicates and the rest are merely value predicates.

7.6. Performance analysis

7.6.1. Evaluate the Structure Index

We analyze the performance by varying the type of relationship between query nodes.

Table 6
Queries for the experiment.

(a) Queries without value predicates

Q	Queries	# Obj	Depth
QS1	SigmodRecord/issue[/volume][/number]	1	3
QA2	root/listing/auction_info/high_bidder [/bidder_name][/bidder_rating]	1	5
QA3	root/listing/auction_info[/current_bid] /time_left	1	4
QD4	/dblp/mastersthesis[/title]/author	1	3
QD5	/dblp/book[/ee]/year	1	3
QD6	//phdthesis[/year][/series][/number]	1	4
QS7	articles[//title][//author]	2	7
QS8	issue[//volume][//author]	2	7
QD9	//inproceedings//title[//i][//sub][//sup]	1	5
QD10	//inproceedings[//month][//url][//ee]	1	4
QD11	//inproceedings[//month][//url][//ee]//title	1	4
QS12	SigmodRecord/issue[/volume][//title]	1	5
QS13	article[/title][//authors]	2	7
QD14	//inproceedings[//title/i][//year]	1	4
QD15	/dblp/article[/journal][//sup]	1	3
QD16	/dblp/incollection[/booktitle][//ee]	1	3
QD17	//article/title[//i][//sub]	1	4
QD18	/dblp/phdthesis[/series][/number][/year]/title	1	3

(b) Queries with value predicates

QXN	Query pattern	# Obj	Depth
QA1	//auction_info[/current_bid = “$620.00”]/time_left	1	5
QA2	/root//auction_info[/location = “LOS ANGELES, CA”]/high_bidder/bidder_name	1	5
QA3	/root/listing[//location = “LOS ANGELES, CA”]/auction_info/time_left	2	4
QA4	//auction_info[/current_bid = “$620.00][/num_items = “1”]/time_left	1	5
QA5	//auction_info[/current_bid = “$610.00”][/num_items = “1”][/started_at = “$100.00”]/time_left	1	5
QA6	//auction_info[/current_bid = “$610.00”][/num_items = “1”][/started_at = “$100.00”] [/num_bids = “16”]/time_left	1	5
QA7	//auction_info[/current_bid = “$610.00][/num_items = “1”][/started_at = “$100.00”] [/num_bids = “16”][/location = “Allentown, PA 18109 ”]/time_left	1	5
QA8	//listing[/seller_info/seller_name = “cubsfantony”]/auction_info/current_bid	1	4
QA9	//listing[/auction_info[/current_bid = “$620.00”][/num_items = “1”]] /bid_history/quantity	1	4
QS1	//issue[/volume = “11”]/number	1	3
QS2	//article[/title = “Architecture of Future Data Base Systems”.]//authors	1	3
QS3	/SigmodRecord[/issue[/volume = “11”][/number = “1”]/articles//title]	2	5
QS4	/SigmodRecord/issue//article[/initPage = “30”][/endPage = “44”]/title	2	5
QS5	/SigmodRecord/issue[/articles/article[/endPage = “44”][/initPage = “30”]]/volume	2	5
QD1	//article[/author = “Frank Manola”]/title	1	3
QD2	//article[/editor = “Paul R. McJones”]/title	1	3
QD3	//article[/editor = “Paul R. McJones”][/volume = “SRC1997-018”]/title	1	3
QD4	//article[/editor = “Paul R. McJones”][/journal = “Digital System Research Center Report”]/year	1	3
QD5	//article[/editor = “Paul R. McJones”][/journal = “Digital System Research Center Report”][/volume = “SRC1997-018”]/year	1	3
QD6	/dblp/article[/author = “Tor Helleseth”]/year	1	3
QD7	/dblp/inproceedings[/author = “Tor Helleseth”]/title//sub	1	4
QD8	/dblp/inproceedings/title[/i = “C”]/sub	1	4
QD9	/dblp/inproceedings/title[/sub = “INF”]	1	4
QD10	/dblp/inproceedings/title[/sub = “INF”]//sub	1	4
QD11	/dblp/inproceedings//i[/sub = “n, n”]	1	5

CPU cost. We compare the average improvement of the CPU time for three different criteria: for only P–C queries, A–D queries and mix of both.

For only *P–C twig queries*, despite the strength of TwigX-Guide in processing and optimizing P–C twig queries, the Structure Index outperforms TwigX-Guide in processing these types of queries, as shown in Fig. 8(a). For instance in QS1, TwigX-Guide decomposes the query into two paths “SigmodRecord/ issue[/volume]” and “SigmodRecord/issue[/number]”. The answer to each path is retrieved directly from the DataGuide and it only needs one join to yield the final result. However, our index is able to trim the search space in the Schema Index as well as in the Data Index. It gains an advantage from its index construction since it preserves the details of all objects located in each path of the schema and data as in Path of So and Path of Do. This feature supports our approach to handle P–C queries more efficiently. Fig. 8(a) shows how our index outperforms TwigX-Guide when querying P–C twig queries in SigmodRecords, Yahoo and DBLP datasets. The average improvement over all the queries with P–C edges is 44.36%.

For only *A–D twig queries*, TwigX-Guide retrieves qualified data from A–D queries based on a region encoding labeling. It incurs a large number of joins that causes a significant delay. The proposed index in this paper has the ability to skip large portions of Schema Objects through object-based intersection as well as discarding irrelevant objects and paths from the Data Object. Thus, it performs significantly better than TwigX-Guide for A–D queries, as shown in Fig. 8(b). The average improvement over all the queries with A–D edges is 80.82%.

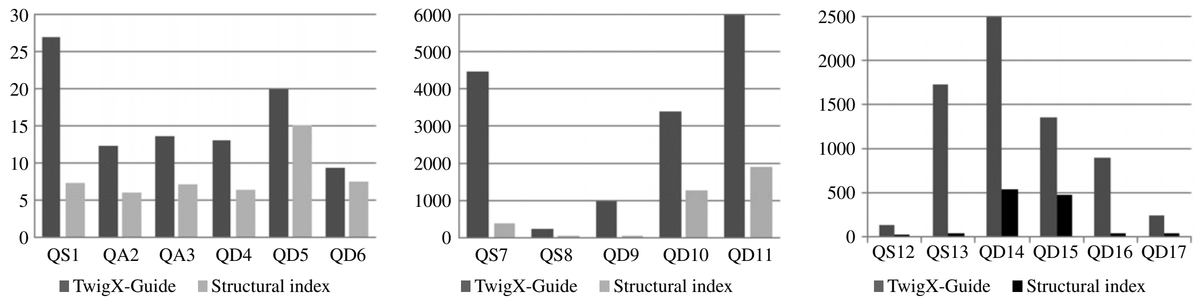
For *mixed edges twig queries*, TwigX-Guide performs better in P–C compared to A–D and performs better in mixed edges compared to A–D, since it uses path matching from DataGuide instead of node matching which reduces the number of joins that need to happen only on A–D. Our index outperforms TwigX-guide in this group of queries as depicted in Fig. 8(c). It performs significantly better than TwigX-Guide for queries with hybrid edges by around 88.87%.

The overall average improvement of the evaluation times for all experimental queries is around 69.68%.

Matching and joining cost. In the previous section, we measured the elapsed time of processing the query inside the CPU. However, this part of the evaluation focuses on two main tasks during the CPU time which are matching query patterns and merging the results. Fig. 9 depicts the percentage cost of matching the query patterns and merging the results in our method in Fig. 9(b) and TwigX-Guide in Fig. 9(a). The figures label each column with the time measurement in milliseconds and the total number of results is shown underneath each query label.

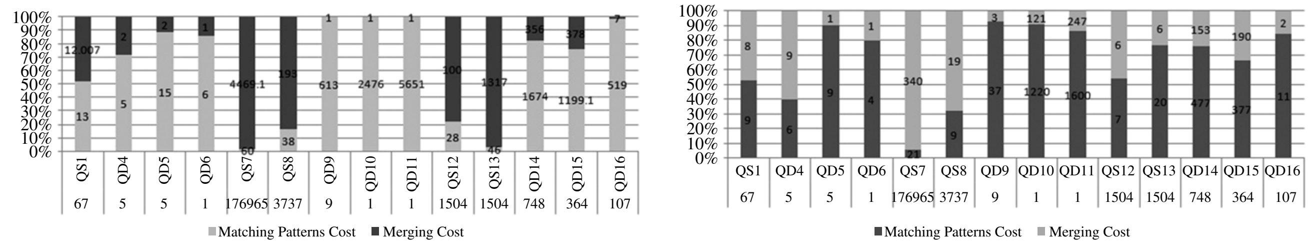
Fig. 9 reveals that our method performs better in matching query patterns in terms of time measurement compared to TwigX-Guide. The cost of matching patterns differs based on the type of relationship between the query nodes. It can be noticed that the matching cost increases when processing queries with A–D relationships as QD9, QD10 and QD11 in both approaches especially, when processing a query over a large data size such as DBLP.

It is obvious from Fig. 9(a) that in most queries, join costs increase dramatically in TwigX-Guide when the number of query results increases, as can be seen in QS7, QS12 and QS13. On the other hand, the cost tends to be 1 millisecond when the result is close to 1 or closer as in QD11 and QD10. However, in our method, it can be seen that the significant increase in the merge cost which occurs in TwigX-Guide no longer exists in our method. We can observe that the join costs are slightly higher in some queries compared to TwigX-Guide as can be seen from the time measurement in QS7, QS12 and QS13 in Fig. 9(b). Ultimately, our method shows the overall superiority in performing the joining process.



(a) PC twig queries. (b) AD twig queries. (c) Mixed edges queries.

Fig. 8. The performance of twig queries in milliseconds.



(a) TwigX-Guide index. (b) Structure Index.

Fig. 9. Query processing cost in ms Table 6(a).

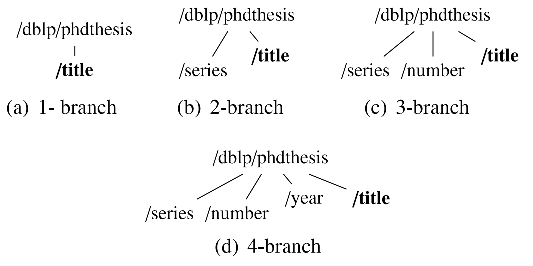


Fig. 10. The query used in the experiment on changing the number of branches.

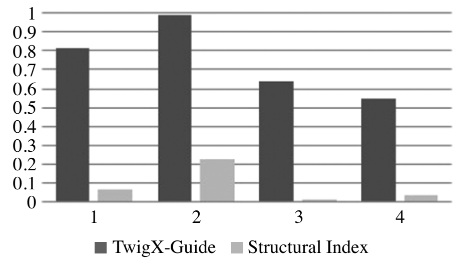


Fig. 11. Changing the number of branches (time is in microseconds).

Changing the number of branches. We select QD18 which is "/dblp/phdthesis [/series] [/number] [/year]/title" from DBLP, with four branches and then varying the number of branches from 1 to 4 as in Fig. 10. The CPU time to process the queries is shown in Fig. 11. It can be seen that the CPU cost of both methods does not increase as the number of branches in the queries increases since the number of output results vary, thus, the number of joined elements varies as well. However, the cost of TwigX-Guide is much more than the cost of the Structure Index. This is because our method reduces the search space to the object partition that includes "phdthesis" instead of searching all data as TwigX-Guide does.

Varying the size of entry documents. We compare the performance of the Structure Index and TwigX-Guide with variable entry document sizes (from 467–2800 KB). We measure the elapsed query processing times and the scale up performance is computed using Eq. (1) as proposed by [56]. QS1, which represents a P–C query, and QS12, which represents a query consisting of P–C and A–D edges, is adopted. According to our experiment results, the execution times are less than those of TwigX-Guide. Figs. 12 and 13 depict that the execution times increase linearly with the document size in both approaches. This observation is consistent with the expected trend. However, it can be seen that the proposed index in this paper scales better than TwigX-Guide as the document sizes increases.

In order to compute the size of scalability performance, Eq. (1) is used where t_{Avg} is the execution time for query processing and $t_{Avg_{base}}$ is the execution time for processing a base case when processing the smallest size of entry documents. As long as the document size is growing, a positive (or negative) value of the scalability indicates that the cost of processing a twig query is increasing (or decreasing). While processing QS1 and QS12, both approaches performed in a linear time while the size of the documents was growing. In spite of this, it can be seen that our index scales better than TwigX-Guide due to the gradual growth of the size scale-up factor.

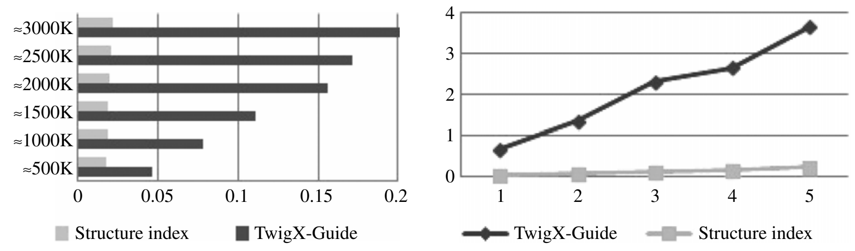
$$Scaleup = \frac{t_{Avg} - t_{Avg_{base}}}{t_{Avg_{base}}} \tag{1}$$

Search quality. Search quality was measured based on terms of accuracy and completeness. Standard precision and recall metrics have been utilized to measure accuracy and completeness. The search accuracy was indicated by the precision measures which determine the fraction of results in the approximate correct answer. On the other hand, search completeness was indicated by the recall measures which determine the fraction of all the correct results found in the approximate answer.

$$Precision = \frac{|DSR \cap SSR|}{|SSR|} \tag{2}$$

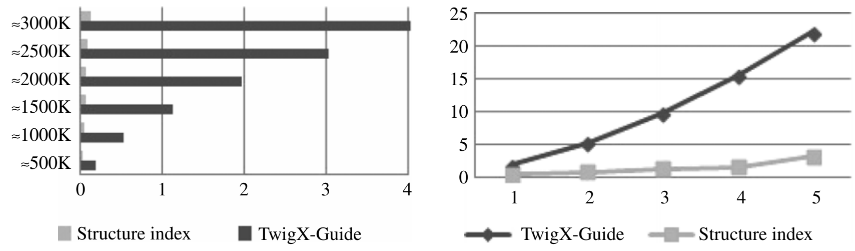
$$Recall = \frac{|DSR \cap SSR|}{|DSR|}. \tag{3}$$

If precision measures the percentage of the output nodes that are desired and recall measures the percentage of the desired



(a) CPU cost. (b) Size scalability.

Fig. 12. Varying the size of entry documents (SQ1 Table 6(a)).



(a) CPU cost. (b) Size scalability.

Fig. 13. Varying the size of entry documents (SQ12 Table 6(a)).

nodes that are output [57], the symbols of Eqs. (2) and (3) will be interpreted as DSR which refers to desired search results and SSR which refers to system search results. The desired / correct search results are the answers returned by SQL/XML Oracle 11g while system search results are the answers generated by our system using the Structure Index.

To compute the search quality (precision and recall) of our method, we performed an intersection between the results for each query presented in Table 6(b) in our method with the results of the same queries run on the SQL/XML oracle engine. Then we divided the intersected results on the SQL/XML results when calculating recall and on the results of our method to calculate precision. Table 7 illustrates the search quality results. From the table, we could see that our method achieves a high search quality of around 98% completeness or precision and 100% recall. We can conclude that pruning the search space using the concept of objects did not affect the search quality.

7.6.2. *Evaluate the Value Index*

The efficiency of the Value Index which is based on objects was studied. As previously mentioned, the system supports a search by content and structure. To achieve this goal, our index provides mechanisms to process the content and structure efficiently. Structure and Content Indexes are combined to answer regular path queries with predicates over values.

We rely on our indices in finding the value predicates before finding and matching the node position. The rationale is that a content search normally results in high selectivity. By performing a content search first, we can reduce the complexity of structural joins. A content search based on a comparison of the specified value predicates works as a filter prior to the structural search.

CPU cost. We compare the time performance of our Value Index with the standard value index. In Fig. 14(a), the experiments run on an Auction dataset. The queries represent a combination of different criteria as mentioned in Section 7.5. Our index is 2 to 4 orders of magnitude more efficient than the standard one in all queries. For instance, while our index takes about 0.1735 ms to retrieve one answer of QA7, the standard index, when querying data of the same size, takes almost 0.2518 ms. The standard index performs well since it uses our Structure Index to search the structural part of the query. However, our method performs better by combining the strength of the object-based Structure Index with the strength of the object-based Value Index. The objects in our Value Index carry semantic meaning and in each value stored based on their paths and tokens within an object to provide fast access to right values that match to the value predicates. This is in contrast to using the standard value which does not carry any semantic meaning, resulting in the consumption of search time to find the right matched values.

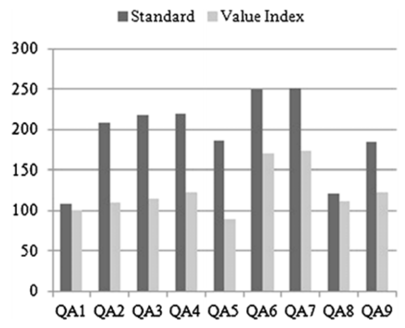
Fig. 15(a) and (b), show the execution time of the queries evaluated over the DBLP dataset. Our experiments reveal that our Value Index outperforms the standard value index. For example, QD6 retrieves 15 results from data of 3 332 131 nodes. Our index needs about 600 ms to produce the results while the standard requires around 900 ms. Since the standard value index is a node-based index, it implies that increasing the total number of nodes increases the size of the data which needs to be scanned and checked. This is because it does not have a specific technique to assist in skipping irrelevant portions of data which exists in ours. Our Value Index trims the search space based on the objects. For instance, in QD6, our method needs to access only the part of the data that is related to the "article" node. However, the standard method searches based on the node and needs to access many nodes which do not participate in the final results. The results of Fig. 14(b) support the earlier experiment outcomes. Our index took less time to evaluate the queries over SigmodRecords dataset. We can notice that QS5 has a significant performance because all the relationships between the query nodes are P–C while the other queries are a hybrid of P–C and A–D. We can observe a large difference in the performance because of the type of relationships. Our method gains more benefits in improving the query performance from the P–C relationship than the standard does.

The total number of scanned elements. The main purpose of this experiment is to indicate the capability of our index to avoid scanning irrelevant portions of the data. Fig. 16 is the total number of visited elements during the evaluation of the Auction data. The total number in our index is reduced by 68% compared to the standard index.

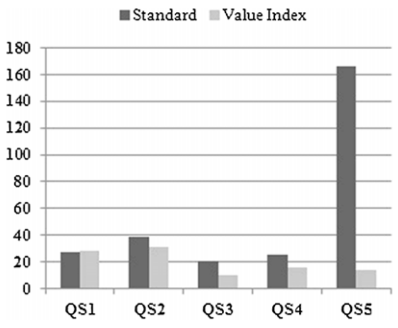
As the same outcome, Fig. 17(a) and (b) show that the total number of visited elements decreased by 59.6% and 77.7% for DBLP

Table 7
Search quality.

Queries	QS1	QD4	QD5	QD6	QS8	QD9	QD10	
Structure Index	67	5	5	1	3737	9	1	Recall: 100%
SQL/XML	67	5	5	1	3737	4	1	
Queries	QD11	QS12	QS13	QD14	QD15	QD16	QD17	
Structure Index	1	1504	1504	748	364	107	344	Precision: 98%
SQL/XML	1	1504	1504	748	364	107	344	



(a) Auction data.



(b) SigmodRecords data.

Fig. 14. The CPU time for datasets (time is in microseconds).

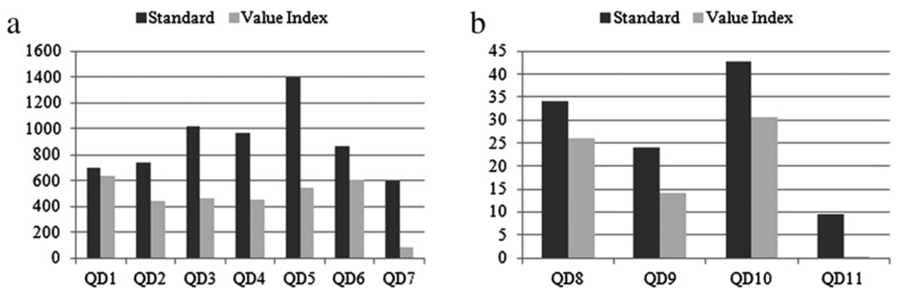


Fig. 15. The elapsed time for DBLP dataset (in milliseconds).

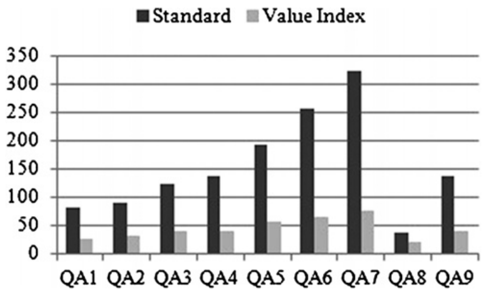
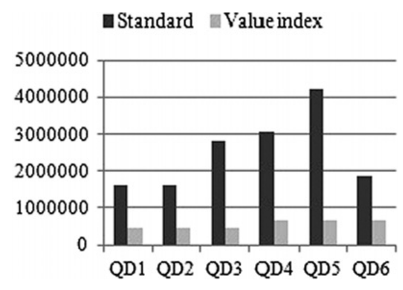


Fig. 16. The total number of scanned elements for Auction data set.

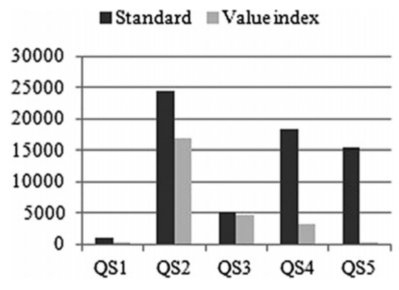
and SigmodRecords respectively. This is evidence of the efficiency of exploiting the semantics of XML data in constructing the Value Index. Since all the query nodes have a semantic connectivity between them that have a similar representation with the data, our Value Index utilizes this significant which results in the reduction of the search space.

The total number of scanned elements is also affected by the type of relationships. In QS5, the average reduction is 98.5% of the total number scanned by each method. This leads us to conclude that the high selectivity caused by a P–C relationship reduces the number of elements to check in order to produce the result.

Changing the number of branches. We select QD5 from DBLP, with four branches and then vary the number of branches from 2 to 4 as shown in Fig. 18. The CPU time to process the queries is shown in Fig. 19(a) whereas the total number of checked elements is shown in Fig. 19(b). It can be seen that the CPU cost of both methods increases as the number of branches in the queries increases.



(a) DBLP dataset.



(b) SigmodRecords dataset.

Fig. 17. The total number of scanned elements.

Table 8
Search quality.

Queries	QA1	QA2	QA3	QA4	QA5	QA6	QA7	QA8	QA9	QS1	QS2	QS3	Search Quality
Value Index	1	1	1	1	1	1	1	1	1	3	1	4	Recall
SQL/XML	2	1	1	2	1	1	1	1	2	3	1	4	97%
Queries	QS4	QD1	QD2	QD3	QD4	QD5	QD6	QD7	QD8	QD9	QD10	QD11	Search Quality
Value Index	1	22	1	1	1	1	15	1	2	1	1	1	Precision
SQL/XML	1	22	1	1	1	1	15	1	2	1	1	1	100%

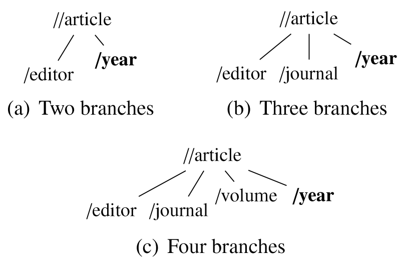


Fig. 18. The query (QD5) used in the experiment on changing the number of branches.

However, the cost of the standard value index is much more than the cost of the Value Index. This is because that the standard index needs to scan more elements than ours, as shown in Fig. 19.

Varying the size of entry documents. We compare the performance of the Value Index and the standard index with different entry document sizes varying from 500 KB to 3000 KB. We measure the CPU Cost to process a query as well as calculating the scale up performance using Eq. (1) as proposed by [56]. We process QS1 with the value predicate in Table 6(b) in different document sizes. According to our experiment results, the execution times are less than those of the standard index. Fig. 20 depicts that the execution times increase linearly with the document sizes in both approaches. However, it can be seen that the Value Index in this paper scales better than Standard index as document size increases. In the first two experiments shown in Fig. 20(b), size scalability has gradual growth. Thereafter in the experiments, it remains steady in both approaches. Also, by adopting the size scale-up factor, it can be indicated that the Value Index scales better than the standard index.

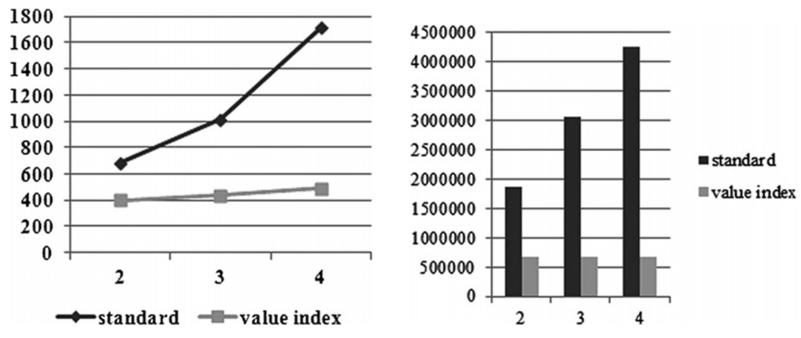
Search quality.

Search quality is measured by precision as in Eq. (2) and recall as in Eq. (3). We captured the desired/correct search results using SQL/XML Oracle 11g as the search quality of the Structure Index whereas the system search results are the answers generated by our system using the Value Index. We used the queries in Table 6(a) to perform this part of the experiment.

Table 8 displays the search results and the search quality of queries over the Value Index. The table reveals that our method obtains an average of 97% recall and 100% precision.

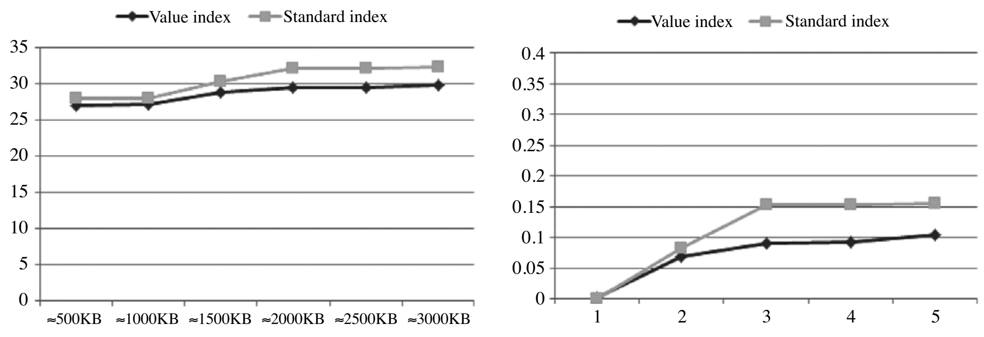
8. Conclusion and future research

This paper proposed the Structure Index to handle the structural part of XML queries and the Content Index to handle the content part. The indices utilized the semantics of XML Schema and XML data in their construction. In addition, this paper introduced the query processing algorithms on the proposed indices.



(a) The elapsed time in milliseconds. (b) The total number of scanned elements.

Fig. 19. Varying the number of branches.



(a) CPU cost. (b) Size scalability.

Fig. 20. Varying the size of entry documents (QS1 Table 6(b)).

Our method can evaluate branching queries, which is considered the most complex part of query processing over XML data due to the complexity of traversing data to find matching results, more efficiently. It utilizes the advantage of XML data being self-described to enhance the notion of a frequently-accessed data subset by employing an object-based data partition. The Schema Index is manipulated using the object-based intersection process to trim the search space and avoid unnecessary joins. One advantage of the Schema Index is instead of iteration on a large index to find the matching trees, the Schema Index usually has a smaller index size compared to an index built only based on the data. Irrelevant data scanning is discarded in the Data Index by eliminating irrelevant objects and paths. The search space is trimmed when evaluating the value predicates because processing the query over the Value Index before drilling down to the Data Index.

Our experiment shows that the proposed method outperforms TwigX-Guide when we compare its performance on different path edge types i.e. (P–C, A–D, or mixed). The calculated average query processing times for a given query indicates that our technique is a viable solution for processing branching queries. The performance evaluation proves the benefits gained from applying the semantic-based indices in trimming the search space and avoiding unnecessary data scanning. Different query patterns with different predicates have been evaluated. The evaluation results on different XML datasets indicate that our proposed method improves a performance by applying the semantic concepts in its Content Index over the non-semantic index.

There is some existing research on processing twig queries with Boolean predicates. [58,59] proposed an algorithm for path and twig query respectively using NOT predicates. Recently, [60,61] proposed algorithms to evaluate twig queries with AND, OR and NOT. However, there are still several gaps which need to be covered. Therefore, in our future research, we need to investigate how to maintain the indices to be able to serve other types of predicates. In fact, Boolean predicates are an important part of the query. Since they have not been widely investigated, in our next work, we would like to focus on this part of the query especially when all the Boolean operators, i.e. AND, OR, NOT, come in a single query.

Acknowledgment

The first author would like to express her gratitude to the Taif University in Saudi Arabia, for supporting her with a scholarship.

References

[1] Y. Liu, D. Yang, S. Tang, T. Wang, J. Gao, Validating key constraints over XML document using XPath and structure checking, Future Gener. Comput. Syst. 21 (4) (2005) 583–595.
[2] B. Huang, S. Yi, W.T. Chan, Spatio-temporal information integration in XML, Future Gener. Comput. Syst. 20 (7) (2004) 1157–1170.
[3] G. Aloisio, G. Milillo, R. Williams, An XML architecture for high-performance web-based analysis of remote-sensing archives, Future Gener. Comput. Syst. 16 (1) (1999) 91–100.
[4] S. Haw, C. Lee, Data storage practices and query processing in XML databases: a survey, Knowl.-Based Syst. (2011). Elsevier B.V.
[5] J. Chung, C.W. Min, K. Shim, APEX: an adaptive path index for XML data, in: Proceedings of ACM SIGMOD, 2002, pp. 121–132.
[6] J.Y. Han, Z.P. Liang, G. Qian, A multiple-depth structural index for branching query, Inf. Softw. Technol. 48 (9) (2006) 928–936.
[7] S. Haw, C. Lee, Extending path summary and region encoding for efficient structural query processing in native XML databases, J. Syst. Softw. (2009). Elsevier Inc.
[8] H. Wang, S. Park, W. Fan, P. Yu, ViST: a dynamic index method for querying XML data by tree structures, in: ACM SIGMOD international conference on Management of data, 2003, pp. 110–121.
[9] P. Rao, B. Moon, PRIX: indexing and querying XML using prufer sequences, in: Proceedings of ICDE, IEEE, 2004, pp. 288–300.
[10] S. Tatikonda, S. Parthasarathy, M. Goyder, LCS–Trim: dynamic programming meets XML indexing and querying, in: VLDB Endowment, ACM, 2007, pp. 63–74.
[11] N.S. Alghamdi, W. Rahayu, E. Pardede, Object-based methodology for XML data partitioning (OXDP), in: Proceedings of the 2011 IEEE International Conference on Advanced Information Networking and Applications, AINA '11, IEEE Computer Society, Washington, DC, USA, 2011, pp. 307–315. http://dx.doi.org/10.1109/AINA.2011.41.
[12] N.S. Alghamdi, W. Rahayu, E. Pardede, OXDP & OXiP: the notion of objects for efficient large XML data queries, Int. J. Grid Util. Comput. 3 (2/3) (2012) 112–125. http://dx.doi.org/10.1504/IJGUC.2012.047762.
[13] D. Fallside, P. Walmsley, Xml schema part 0: primer second edition, W3C recommendation.
[14] G. Gou, R. Chirkova, Efficiently querying large XML data repositories: a survey, IEEE Trans. Knowl. Data Eng. 19 (10) (2007) 1381–1403.
[15] K. Wong, J. Yu, N. Tang, Answering XML queries using path-based indexes: a survey, World Wide Web 9 (3) (2006) 277–299.
[16] S. Mohammed, P. Martin, Index structures for XML databases, in: L. Changqing, L. Tok (Eds.), Advanced Applications and Structures in XML Processing: Label Streams, Semantics Utilization and Data Query Technologies, IGI Global, 2010, pp. 98–124.
[17] R. Goldman, J. Widom, Dataguides: enabling query formulation and optimization in semistructured databases, in: M. Jarke, M.J. Carey, K.R. Dittrich, F.H. Lochovsky, P. Loucopoulos, M.A. Jeusfeld (Eds.), Proceedings of 23rd International Conference on Very Large Data Bases, VLDB'97, August 25–29, 1997, Athens, Greece, Morgan Kaufmann, 1997, pp. 436–445.
[18] R. Goldman, J. Widom, Approximate dataguides, in: Proceedings of the Workshop on Query Processing for Semistructured Data and Non-Standard Data Formats, 1999, pp. 436–445.
[19] B.F. Cooper, N. Sample, M.J. Franklin, G.R. Hjaltason, M. Shadmon, A fast index for semistructured data, in: VLDB, Morgan Kaufmann, 2001, pp. 341–350.
[20] F. Rizzolo, A. Mendelzon, Indexing XML data with toxin, in: Proc. of WebDB 2001, 2001, pp. 49–54.
[21] Q. Zou, S. Liu, Ctree: a compact tree for indexing XML data, in: Proceedings of the 6th annual ACM international workshop on web information and data management, Press, 2004, pp. 39–46.
[22] Z. Chen, J. Gehrke, F. Korn, N. Koudas, J. Shanmugasundaram, D. Srivastava, Index structures for matching XML twigs using relational query processors, Data Knowl. Eng. 60 (2) (2007) 283–302. http://dx.doi.org/10.1016/j.datak.2006.03.003.
[23] T. Milo, D. Suciu, Index structures for path expressions, in: C. Beeri, P. Buneman (Eds.), Database Theory, Proceedings 7th International Conference, ICDT '99, Jerusalem, Israel, January 10–12, 1999, in: Lecture Notes in Computer Science, vol. 1540, Springer, 1999, pp. 277–295.
[24] Q. Chen, A. Lim, K.W. Ong, $D(k)$-index: an adaptive structural summary for graph-structured data, in: Proceedings of the 2003 ACM SIGMOD international conference on Management of data, SIGMOD '03, ACM, New York, NY, USA, 2003, pp. 134–144. http://dx.doi.org/10.1145/872757.872776. URL http://doi.acm.org/10.1145/872757.872776.
[25] S. Abiteboul, P. Buneman, D. Suciu, Data on the Web: From Relations to Semistructured Data and XML, Morgan Kaufmann Pub., 2000.
[26] R. Kaushik, P. Bohannon, J. Naughton, H. Korth, Covering indexes for branching path queries, in: Proceedings of the 2002 ACM SIGMOD international conference on Management of data, ACM, 2002, pp. 133–144.
[27] S. Haw, C. Lee, Node labeling schemes in XML query optimization: a survey and trends, IETE Tech. Rev. 26 (2009).
[28] P. Dietz, Maintaining order in a linked list, in: Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing, ACM, 1982, pp. 122–127.
[29] Q. Li, B. Moon, et al. Indexing and querying XML data for regular path expressions, in: Proceedings of the International Conference on Very Large Data Bases, 2001, pp. 361–370.
[30] C. Zhang, J. Naughton, D. DeWitt, Q. Luo, G. Lohman, On supporting containment queries in relational database management systems, in: ACM SIGMOD Record, Vol. 30, ACM, 2001, pp. 425–436.
[31] I. Tatarinov, S.D. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita, C. Zhang, Storing and querying ordered XML using a relational database system, in: Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, SIGMOD '02, ACM, New York, NY, USA, 2002, pp. 204–215. http://dx.doi.org/10.1145/564691.564715. URL http://doi.acm.org/10.1145/564691.564715.
[32] P. O'Neil, E. O'Neil, S. Pal, I. Cseri, G. Schaller, N. Westbury, Ordpaths: insert-friendly XML node labels, in: Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data, SIGMOD '04, ACM, New York, NY, USA, 2004, pp. 903–908. http://dx.doi.org/10.1145/1007568.1007686. URL http://doi.acm.org/10.1145/1007568.1007686.
[33] T. Grust, M. Van Keulen, Tree awareness for relational DBMS kernels: Staircase join, Group Commun. Charges. Technol. Bus. Models (2003) 231–245.
[34] T. Grust, M.V. Keulen, J. Teubner, Accelerating XPath evaluation in any RDBMS, ACM Trans. Database Syst. 29 (1) (2004) 91–131. http://dx.doi.org/10.1145/974750.974754.
[35] H. Wu, T. Ling, B. Chen, L. Xu, Twigtable: using semantics in XML twig pattern query processing, J. Data Semant. XV (2011) 102–129.
[36] S. Mamou, Y. Sagiv, XSEarch: a semantic search engine for XML, in: Proceedings of the Twenty-ninth International Conference on Very Large Databases, Berlin, Germany, 9–12 September, 2003, Morgan Kaufmann Pub., 2003, p. 45.
[37] Y. Li, C. Yu, H. Jagadish, Schema-free XQuery, in: Proceedings of the Thirtieth International Conference on Very Large Data Bases, Vol. 30, VLDB Endowment, 2004, pp. 72–83.

[38] G. Li, J. Feng, J. Wang, L. Zhou, Effective keyword search for valuable lcas over XML documents, in: Proceedings of the Sixteenth ACM Conference on Conference on Information and Knowledge Management, ACM, 2007, pp. 31–40.
[39] Z. Liu, Y. Cher, Reasoning and identifying relevant matches for XML keyword search, Proc. VLDB Endowment 1 (1) (2008) 921–932.
[40] Z. Bao, T. Ling, B. Chen, J. Lu, Effective XML keyword search with relevance oriented ranking, in: IEEE 25th International Conference on Data Engineering 2009, ICDE'09, IEEE, 2009, pp. 517–528.
[41] K. Golenberg, B. Kimelfeld, Y. Sagiv, Keyword proximity search in complex data graphs, in: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data, ACM, 2008, pp. 927–940.
[42] B. Ding, J. Xu Yu, S. Wang, L. Qin, X. Zhang, X. Lin, Finding top-*k* min-cost connected trees in databases, in: IEEE 23rd International Conference on Data Engineering, 2007, ICDE 2007, IEEE, 2007, pp. 836–845.
[43] J. Tatemura, XML stream processing: stack-based algorithms, in: L. Changqing, L. Tok (Eds.), Advanced Applications and Structures in XML Processing: Label Streams, Semantics Utilization and Data Query Technologies, IGI Global, 2010, pp. 184–226.
[44] S. Al-Khalifa, H. Jagadish, N. Koudas, J. Patel, D. Srivastava, Y. Wu, Structural joins: a primitive for efficient XML query pattern matching, in: Proceedings. 18th International Conference on Data Engineering, 2002, IEEE, 2002, pp. 141–152.
[45] N. Bruno, N. Koudas, D. Srivastava, Holistic twig joins: optimal XML pattern matching, in: Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, SIGMOD '02, ACM, New York, NY, USA, 2002, pp. 310–321. http://dx.doi.org/10.1145/564691.564727. URL http://doi.acm.org/10.1145/564691.564727.
[46] H. Jiang, W. Wang, H. Lu, J.X. Yu, Holistic twig joins on indexed XML documents, in: Proceedings of the 29th International Conference on Very Large Data Bases, Vol. 29, VLDB '03, VLDB Endowment, 2003, pp. 273–284. URL http://dl.acm.org/citation.cfm?id=1315451.1315476.
[47] S. Chen, H.-G. Li, J. Tatemura, W.-P. Hsiung, D. Agrawal, K.S. Candan, Twig2stack: bottom-up processing of generalized-tree-pattern queries over XML documents, in: Proceedings of the 32nd International Conference on Very Large Data Bases, VLDB '06, VLDB Endowment, 2006, pp. 283–294. URL http://dl.acm.org/citation.cfm?id=1182635.1164153.
[48] N. Grimsmo, T.A. Bjørklund, Towards unifying advances in twig join algorithms, in: Proceedings of the Twenty-First Australasian Conference on Database Technologies, Vol. 104, ADC '10, Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 2010, pp. 57–66. URL http://dl.acm.org/citation.cfm?id=1862242.1862252.
[49] J. Lu, T.W. Ling, C.-Y. Chan, T. Chen, From region encoding to extended dewey: on efficient processing of XML twig pattern matching, in: Proceedings of the 31st International Conference on Very Large Data Bases, VLDB '05, VLDB Endowment, 2005, pp. 193–204. URL http://dl.acm.org/citation.cfm?id=1083592.1083618.
[50] J. Lu, T.W. Ling, Z. Bao, C. Wang, Extended XML tree pattern matching: theories and algorithms, IEEE Trans. Knowl. Data Eng. 23 (3) (2011) 402–416.
[51] N.S. Alghamdi, W. Rahayu, E. Pardede, Semantic-based construction of content and structure XML index, in: The 24th Australasian Database Conference (ADC), ADC'13, Adelaide, Australia, 2013.
[52] N.S. Alghamdi, W. Rahayu, E. Pardede, Object-based semantic partitioning for XML twig query optimization, in: Proceedings of the 2013 IEEE International Conference on Advanced Information Networking and Applications, AINA '13, IEEE Computer Society, Barcelona, Spain, 2013.
[53] S. Haw, C. Lee, Stack-based pattern matching algorithm for XML query processing, J. Digit. Inf. Manage. 5 (3) (2007) 167.
[54] J. Lu, T. Chen, T.W. Ling, Efficient processing of XML twig patterns with parent child edges: a look-ahead approach, in: Proceedings of the Thirteenth ACM International Conference on Information and Knowledge Management, ACM, 2004, pp. 533–542.
[55] University of Washington XML Repository, 2002. URL http://www.cs.washington.edu/research/xmldatasets.
[56] C.-P. Chou, K.-F. Jea, H.-H. Liao, A syntactic approach to twig-query matching on XML streams, J. Syst. Softw. 84 (6) (2011) 993–1007.
[57] Z. Liu, Y. Chen, Identifying meaningful return information for XML keyword search, in: Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data, ACM, 2007, pp. 329–340.
[58] E. Jiao, T. Ling, C.-Y. Chan, Pathstack¬: a holistic path join algorithm for path query with not-predicates on XML data, in: Database Systems for Advanced Applications, Springer, 2005, pp. 113–124.
[59] T. Yu, T. Ling, J. Lu, Twigstacklist¬: a holistic twig join algorithm for twig query with not-predicates on XML data, in: Database Systems for Advanced Applications, Springer, 2006, pp. 249–263.
[60] D. Che, T.W. Ling, W.-C. Hou, Holistic boolean-twig pattern matching for efficient XML query processing, IEEE Trans. Knowl. Data Eng. (2012).
[61] D. Ding, D. Che, W.-C. Hou, A direct approach to holistic boolean-twig pattern evaluation, in: Database and Expert Systems Applications, Springer, 2012, pp. 342–356.

Norah Alghamdi completed a Bachelor of Computer Science at Taif University, Taif, Saudi Arabia. She completed a Master of Computer Science from the Department of Computer Science and Computer Engineering at La Trobe University, Victoria, Australia where she is currently undertaking a Ph.D. Her areas of interest are XML databases, query processing, and query optimization.

Wenny Rahayu is an Associate Professor in the Department of Computer Science and Computer Engineering, La Trobe University, Australia. 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. She is currently the Head of the Department of Computer Science and Computer Engineering at La Trobe University.

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.