A Filter Index for Complex Queries on Semi-structured Data

Wang Lian Nikos Mamoulis David W.Cheung

Department of Computer Science and Information Systems,
The University of Hong Kong, Pokfulam, Hong Kong.
{*wlian, nikos, dcheung*}*@csis.hku.hk*

Abstract. Answering a query on XML data usually involves breaking it into a number of small components (e.g., edges, paths, twigs, etc.), evaluating them and joining the results. In this paper we propose an alternative technique that uses these components to filter a large part of the database that does not qualify them, before validating the query on the actual data. Our methodology uses a signature index to search fast and prune effectively the search space. The efficiency of the proposed technique is demonstrated by comparison with an existing index, on real data.

1 Introduction

With XML becoming a standard for information exchange, there has been an increasing volume of information in semi-structure format which need to be queried and analyzed efficiently. Conventional query evaluation techniques are not readily applicable due to the loosely defined schema of semi-structured data. As a result, a number of specialized techniques have been developed, for processing queries that conform to XML languages, like XPath [12] and XQuery [13].

XML documents can be modeled as (directed) graphs having as nodes elements or values and as edges the parent/child relationships between them. Typical queries on XML data ask for documents (or parts of them) that contain a *path* or *subgraph* (e.g., twig) expression. An example of such a query is $article/[author = `John\ \ Smith'\ \ AND\ \ year = `2000']$, which asks for all articles of 'John Smith' which were published in year 2000.

The diverse nature of structures and queries renders query evaluation hard. A common solution [11] is to decompose the structural part of the documents into a number of relational tables that capture the relative position of the various *elements* (e.g., journal, author, etc.) into the graph structures. Pattern queries are processed by joining these tables, in order to bring back parent/child relationships. Join processing can become very expensive, especially if the queries involve relative path expressions. This problem was alleviated with the introduction of special encoding schemes for XML graphs [6, 9]. The encoding can be utilized by merge-join like algorithms [1, 2], for structural queries. However these schemes are defined only for the case where the documents are *tree* structures. It is not clear how to apply similar encodings on arbitrary graph structures. Using summarized graphs [4, 10] to evaluate queries usually involves breaking it into small pieces, e.g., paths, evaluating the most selective one(s), and then verifying the remaining query constraints on the candidate results [6]. This process may require traversing parts of a large number of candidate documents, or performing a large number of joins. As a result, for many applications, where the XML data are arbitrary graphs, evaluating queries either on simple relational tables [11] or on exact/summarized graphs [4, 10] can be rather expensive.

We follow a different approach to handle complex queries based on the rationale that we need to utilize as much as possible the selectivity of a query to reduce the search effort. Our methodology is based on breaking the query into a number of small components and use them to filter documents that do not qualify these components. Given an XML database consisting of many documents (i.e., XML graphs), we generate a signature bitmap for each document, indicating the components which are present in the document. These signatures are organized in a hierarchical structure named **SG-tree** which is similar to the R–tree [5]. Each query is also transformed to a signature bitmap indicating the components it contains. After traversing the SG-tree, we can efficiently filter out most of the documents, which do not satisfy the query. Finally, all nonfiltered documents are validated to give the exact answer. The advantages of the proposed SG-tree can be summarized as follows:

- It is simple to implement and its construction/maintenance cost is low, compared to sophisticated graph-based indexes [4, 10, 7]. Moreover, it is a disk-based structure, which can be implemented in limited memory conditions, as opposed to graph-based indexes which do not fit in memory if the data volume is large.
- We propose a technique that encodes value ranges in the indexed components. In this way, the index is useful in filtering queries not only based on structure, but also based on value selections.
- Our techniques can be used to index arbitrary graph structures as opposed to methods restricted to only tree-structured documents [6, 9, 1, 2].

Experiments comparing the SG-tree with the A(k)-index [7] on real data show that the SG-tree is a powerful filter index for high selective queries.

The rest of the paper is organized as follows. Section 2 provides background and reviews previous work on XML query processing. Section 3 describes the proposed indexing techniques and Section 4 the query decomposition method. In Section 5, we evaluate the filter effectiveness and time efficiency of the proposed techniques by real data. Finally, Section 6 concludes with a discussion and directions for future work.

2 Background and Related Work

In our data model we assume a database of XML documents. Each document is represented by a directed, node-labeled graph $G = (V_G, E_G)$, where each (non-terminal) node $v \in V_G$ has a unique *oid* and a label (which corresponds to an

XML element). Nodes with no outcoming edges (terminal nodes) contain values. E_G contains the edges of the graph, which are either parent-child relationships or *IDREF* links between elements. For simplicity, attributes and elements are represented by a single construct (i.e., non-terminal node). Figure 1 shows an example of two XML documents represented by node-labeled graphs. The document on the left is an item of a bibliography database. The graph in this case, is a tree, i.e., there are no links between elements. The document on the right contains information about a part (i.e., mechanical component). A nut can be paired with a bolt, so there are links between elements, represented by dashed lines.

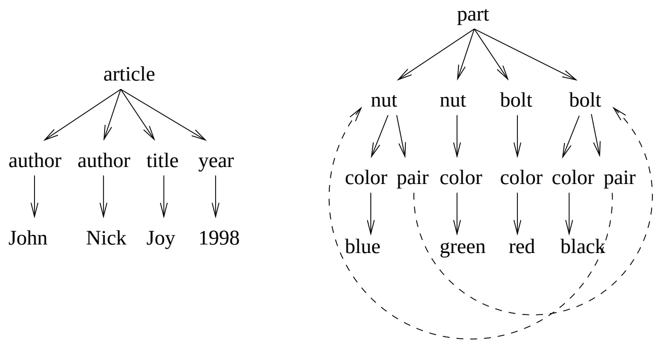


Fig. 1. XML documents represented as directed graphs

Queries are defined by paths/structural patterns between XML elements potentially enriched by selection predicates on attribute values. Following an XQuery-like notation, we use '/' to denote direct parent/child relationships, '//' to denote descendant/ancestor relationships and '$/*/$' to denote descendant/ancestor relationships interleaved by one node. The root is denoted by a label at the beginning of the query with a '/' before it. Branches are expressed using brackets '[]'. E.g., $/a/[b]/c$ denotes a twig having a as root and b, c as left and right son, respectively. Selection predicates on attributes are also enclosed by brackets. An example is '$article/[author = \text{'}John' \;\; AND \;\; year = \text{'}1998']$', asking for all documents with *article* as root label, having at least two attributes *author* and *year*, with values 'John' and 1998, respectively. A query result is therefore a set of document subgraphs which match the query predicates (e.g., the left document of Figure 1).

2.1 Previous work

Many XML database applications decompose the data and store them into relational tables that capture the structural relationships between them. A straightforward solution defines a database schema using the Document Type Definition

(DTD) of the documents [11]. This approach fragments the data into a small number of tables and expensive joins are required to bring back structural information. A more efficient solution is to encode [6, 9] the relative positions of the nodes in the documents using a preorder traversal of the structure and store them into tables clustered by node label. Thus parent/child queries a/b (or ancestor/descendant queries $a//b$, in general) can be evaluated by first selecting the nodes labeled a and b into two lists and then traverse the two lists in a sort-merge join fashion verifying the structural relationships between the elements [1]. A complex query containing path or twig expressions can be evaluated by decomposing it into a set of binary joins and then merging the partial results. Recently [2], stack-based algorithms were proposed to avoid this multi-step process by evaluating synchronously all structural components of the query. Nevertheless these approaches can operate only on tree structures, whereas in general XML documents are graph-structured. Some researchers have proposed the use of graph summaries that capture most of the characteristics of the original graph, serving thus as a structural abstraction of them [4, 10]. Path queries are evaluated on these summaries rather on the original documents. There are several limitations of these indexes. First, they are usually very large (with size comparable to the size of the original documents) and therefore accessing them is costly. This problem is alleviated in [7], where a family of progressively more refined summaries is introduced and one that achieves a good space-time trade-off is chosen. Second, queries are usually evaluated by first finding the nodes with the same label as the first element of the query and then traversing the graph until a path condition fails, or the final element is found (together with a result). This process is rather expensive for relative path queries, since large part of the graph may need to be accessed before the qualifying paths are found. Finally, they are appropriate only for simple path queries and cannot be used directly to evaluate complex ones that contain conjunctions of simple paths.

Summarizing, evaluating generic queries on XML graph structures is still a hard problem. In the next section, we propose a methodology that aims to reduce the size of the problem using indexes that filter fast documents which do not qualify complex queries involving multiple structural components.

3 Methodology

Since evaluating complex queries directly on the documents (or on decomposed data) can be expensive, it is a good idea to minimize the number of documents that need to be searched for a specific query. Our aim is to develop methods which (i) are lightweight, (ii) have low construction and maintenance costs, (iii) prune efficiently the search space, and (iv) apply for arbitrary (i.e., not only path) queries. On the other hand, we are not interested to create an index that provides exact answers, but one that filters out as many documents as possible.

Our methodology is based on indexing structural components that appear frequently in query patterns. These components can be edges, paths, twigs, relative path expressions, etc. Given a query workload, a data mining algorithm

[3] can be employed to extract a set of frequent structural components. Then an index that finds fast which documents contain these components is built. Queries are preprocessed to extract the components contained in them and the index is used to locate fast the documents that contain these components and filter-out the ones that do not. Finally, the candidate documents are searched using some well-established XML query processing technique to verify the rest of the query predicates and to retrieve the query results.

Most queries involve selections on attribute values in a document. Furthermore, value range selections are expected to be much more selective than edges between elements. For instance, the query $article/author$ is not very selective. On the other hand, the query $//[author = \text{‘}Zebra\text{'}]$ is possibly very selective.

Therefore, it is a good idea to index also frequent query components containing values. There are two limitations here. First, the number of potential values for an attribute is very large and index all of them is too expensive. Second, queries on attribute values not only differ in nature and selectivity, but also show diverse distributions on the attribute domains.

Based on the nature and distribution of the queries and the size of the domains, we split them into classes and use the class values as new tags. For example, consider the year/value pair in the bibliography database, and assume that items published from 1995 to 2002 are queried more frequently than ones before 1995. We can define 10 query ranges: one for each year from 1995 to 2002, one for the range 1990-1994 and one for the years before 1990.

3.1 Indexing document signatures

If a large number of components appear in the query, scanning and merging all the relevant lists or bitmaps could be expensive. We consider an alternative approach that models each document with a bitmap indicating which components are present in the document. We call this bitmap the *signature* of the document.

The query is also transformed to a signature, based on the existence of components in it and compared to the document signatures using bitwise operators. The index is motivated by the following observation: if the signature of a query does not match with a document signature, then the document does not contain an answer to the query, thus it can be safely filtered from consideration. In Section 3.1, we will describe an index that organizes hierarchically the signatures and can accelerate query processing.

Before we show in detail how queries are filtered against signature representations we need to define some operations on bitmaps.

Definition 1. *Let s be a binary signature. The* **length** *of s, denoted by $Len(s)$, is defined by the number of bits in s. The* **area** *of s, denoted by $Area(s)$, is defined by the number of 1's in s.*

Definition 2. *Let s_1, s_2 be two binary signatures, such that $Len(s_1) = Len(s_2)$. Let $\wedge$, $\vee$, and $\otimes$ denote the bitwise operations AND, OR and XOR, respectively, on signatures of the same length. The* **overlap** *$Ovr(s_1, s_2)$ between s_1 and s_2*

is defined by $Area(s_1 \wedge s_2)$*. The* **distance** $Dist(s_1, s_2)$ *between* s_1 *and* s_2 *is defined by* $Area(s_1 \otimes s_2)$*. The* **difference** $Diff(s_1, s_2)$ *of* s_1 *from* s_2 *is defined by* $Area(s_1 \otimes (s_1 \wedge s_2))$*. Finally,* s_1 *is said to* **contain** *or* **cover** s_2 *iff* $s_1 \wedge s_2 = s_2$*.*

In other words, the overlap between two signatures is the number of common 1-bits in them, the (*hamming*) distance is the number of bits with different values in them, and the difference $Diff(s_1, s_2)$ (non-commutative) is the number of 1-bits in s_1 but not in s_2. Finally, s_1 contains (or covers) s_2 if all set bits in s_2 are also set in s_1. For example, $Area(10110) = 3$, $Ovr(10110, 00011) = 1$, $Dist(10110, 00011) = 3$, $Diff(10110, 00011) = 2$, and 10111 covers both 10110 and 00011.

An example Consider the two XML documents (containing bibliography items) shown in Figure 2(a). Assume that the domain of attribute *author* is divided into two classes $J*$ and $N*$ representing author names starting with J and N, respectively. Similarly, the domain of *title* is divided into 2 classes ($J*$, $P*$), and the domains of *publisher* and *year* are considered as categorical with one class per category value.

Figure 2(b) shows the signatures of the two documents and also an *edge directory* (in general *component directory*) that encodes the potential edges of the database. It also shows graph-representations of the signatures (called *s-graphs*). Notice that there is a common edge between the s-graphs ($author/J*$). Also the number of encoded edges, i.e., the number of rows in the edge directory, is defined by the potential edges that can be found in a document. Invalid edges (e.g., ones which do not conform to the DTD), like $article/publisher$) are excluded to save space.

Given a query (e.g., $author/`Nick'$) if a signature (e.g., $sig(D_1)$) contains the corresponding edge (e.g., $author/N*$), the corresponding document (e.g., D_1) may qualify the query. Otherwise, (e.g., document D_2) it can be excluded from consideration.

The signature tree (SG–tree) The signatures of all documents can be stored as $\langle signature, document\_id \rangle$ tuples, sequentially in a *signature file*. Since scanning the whole file for a query can be expensive, we propose an alternative organization of the signatures in a hierarchical structure. So it is a good idea to cluster them based on their similarity. A nice property of the signatures is that we can use the *same* representation, i.e., a bitmap, for documents and document groups. Therefore, in the signature of a group of documents an edge is on iff this edge exists in at least one document of the group. We employ this property to define a simple, but efficient, hierarchical index for signatures.

The SG–tree (or *signature* tree) is a dynamic balanced tree similar to R–tree [5] for signature bitmaps. Each node of the tree corresponds to a disk page (using multipage nodes is a potential implementation) and contains entries of the form $\langle sig, ptr \rangle$. In a leaf node entry, sig is the signature of the document and ptr is a pointer to the file that contains the document (i.e., the $document\_id$). The signature of a directory node entry is the logical OR of all signatures in the node pointed by it and ptr is a pointer to this node. In other words, the signature of each entry *covers* all signatures in the subtree pointed by it. All nodes contain

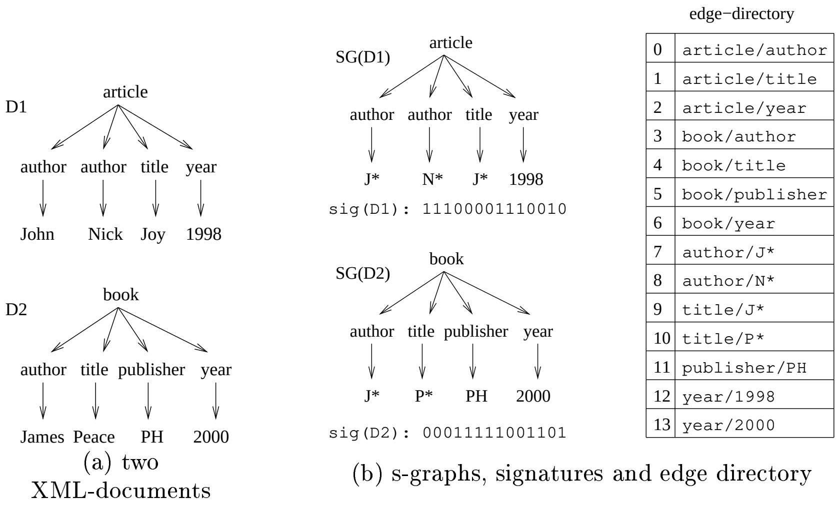


Fig. 2. Example of s-graphs and signature encoding

between c and C entries, where C is the maximum capacity and $c \leq C/2$, except from the root which may contain fewer entries. Figure 3 shows an example of a signature tree. The leaf entries contain signatures of nine documents and pointers to them. In this graphical example the maximum node capacity C is three and the length of the signatures six. in practice, C is in the order of several tens and the length of the signatures in the order of several hundreds.

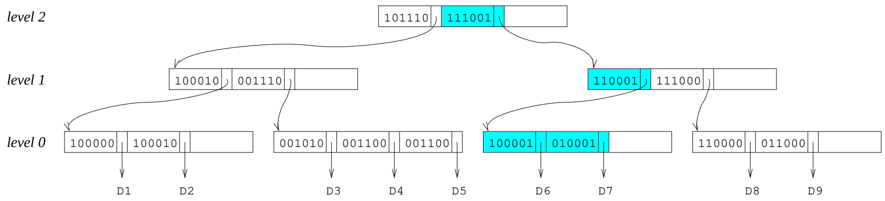


Fig. 3. Example of a signature tree

The tree is designed to answer fast *signature containment queries*, e.g., find all documents containing edges a/b and c/d. The query is transformed to a signature qs and the tree is traversed in a depth-first fashion, following entries whose signature contains qs; if the signature of an entry does not contain qs, no document indexed in the subtree below it can contain a query result. Consider for example the tree of Figure 3 and a query which contains a component encoded by the last (i.e., sixth) signature bit. Since the first entry of the root has 0 in the sixth position of its signature, we know that no document indexed in the subtree under it can participate in the query result. On the other hand, the second entry should be followed and the rightmost node of the next level (i.e., level 1) is visited. Only the first entry of this node contains the component, and it is followed. Finally the query results are found in the third leaf node. The qualifying entries are highlighted in the figure.

Therefore it is crucial for the tree performance that documents with similar signatures are clustered together in the leaf nodes. A good insertion algorithm can achieve this goal. A generic insertion algorithm used for hierarchical access methods like the B–tree and the R–tree usually contains two functions: *choose_subtree* and *split*. When a new entry e needs to be inserted to the tree *choose_subtree* is first called to chooses the most appropriate node n, in order to insert e in it. If n is overflow, then *split* is called to divide the entries of an overflowed node into two groups. Both functions should be tuned to maximize the efficiency of the tree. For the SG–tree, we need to define quality criteria based on which these functions operate. A good tree minimizes the number of nodes that need to be accessed during a signature containment query. Therefore the directory node entries of a good tree should have (i) a small area[1] and (ii) small overlap between them, if they are at the same level.

The algorithm *choose_subtree* shown in Figure 4 distinguishes three cases. In the first case only one entry $e_i \in n$ contains the new entry e and it is directly chosen. In the second case multiple entries contain e. The algorithm chooses the one with the minimum area, since this refines the structure (in analogy to choosing the smaller MBR among a number of ones that contain the new entry in R–trees). Finally, the third case applies when no $e_i \in n$ contains e. The algorithm in this case picks the one which requires the smallest area enlargement to index e under it, or more formally the entry for which $Diff(e, e_i)$ is the minimum. Ties are broken by choosing the entry with the minimum length. We also implemented another version of *choose_subtree* that picks the entry which after extended causes the minimum increase in its overlap with the rest of the entries in the same node. Nevertheless, through experimentation we found that the version of Figure 4 gives as good trees at a much lower insertion cost.

The split algorithm of the SG–tree is also based on that of the R–tree. We first pick the pair of entries in the overflowed node with the maximum distance (i.e., with the maximum number of bits in their exclusive OR). We call these two entries *seeds* and assign them to two groups. The signatures of the groups is the logical OR of the members of the group (in accordance with the MBRs in an R–tree). The rest of the entries are assigned to the group that requires the minimum extension to include them. Ties are broken by (i) choosing the group with the minimum area and (ii) the group with the minimum number of entries. If at some point the cardinality of a group plus the number of remaining entries equals c, the remaining entries are assigned to the group to avoid underflow of the new node. Finally, deletions in the SG–tree are handled as in the R–tree; if a leaf node underflows, it is deleted and its entries are reinserted to the tree.

[1] For simplicity, we extend the function definitions of Section 3.1, to apply on SG-tree node entries, e.g., $Area(e) \equiv Area(e.sig)$.

```
function choose_subtree(Node n, Entry e): Node son {
  if e is contained in exactly one entry e_i ∈ n then
    return e_i;
  else if e is contained in multiple entries e_i ∈ n then
    return e_i with the minimum Area(e_i);
  else /* e is contained in no entry e_i ∈ n */
    pick the entry which requires the minimum area enlargement to contain e;
    break ties by choosing the entry with the minimum area;
}
```

Fig. 4. The *choose_subtree* algorithm

3.2 Which components to index?

In Figure 2, we used the edges as the indexed components of all documents, however other frequently queried components (e.g., relative edge, paths or twigs) can also be considered.

In general the number of distinct edges is small, while the number of pathes and twigs can be huge, it is impossible to index all of them. In this case, data-mining tools should be used to find out some valuable components for indexing. Our requirement of being a valuable component is intuitive : (a)small size (b)high selectivity. How to find interesting components is out of the range of this paper. Since there are lots of available methods for discovering patterns from trees or graphes, they can easily be adopted for our task.

In our experiments, the indexed components are relative edge denoted by RE. That is all pair of elements with ancestor and descendant relationship. For example, in path */article/author/address*, we can find three REs: *article//author*, *article//address*, *author//address*.

4 Query Decomposition

Given a query and a set of valuable components, the query decomposition is to find out all valuable components that are contained in the query. Because the number of valuable components is rather small, we can use edge-based inverted index to organize them. This inverted index can remove all components that contain one or more edges which are missed in the query. Remaining components should be checked with the query by tree or graph matching to determine whether they are contained in the query. After finding all components in a query, a signature bitmap of this query is constructed and feed into SG-tree to dig out all documents that possibly satisfy the query.

5 Experiments

In order to evaluate the efficiency of SG-tree, we use the DBLP bibliography database [8] in our experiments. The indexed component unit in our tests is the

Relative Edge. The decomposition of all queries are based on all indexed REs. To take care of the values, we used a equi-depth split algorithm to divide the domains of common values (e.g., names of authors) to a number of ranges with an equal number of documents in each range, and generated tag-aliases for them. The total number of RE that appear in the set of 200,000 DBLP documents is 371, i.e., the length of the signatures is about 47 bytes. This definition resulted in 182,224 distinct signatures. Because of the space limitation, we only show two sets of experiments: comparison of SG-tree with A(k)-index and the scalability of SG-tree.

5.1 Query generation

In order to evaluate the efficiency of SG-tree we generate queries from documents. The queries were generated by randomly picking a document and removing some edges from it. The removed edges are randomly replaced either by // or by logical AND. The remaining edges along with their structural/logical relationships formulate the final query. For instance a document $a/[b/c]/d/e$, after removing edges a/b and d/e becomes query $a/d//e \wedge b/c$, which contains the (edge) components b/c and a/d.

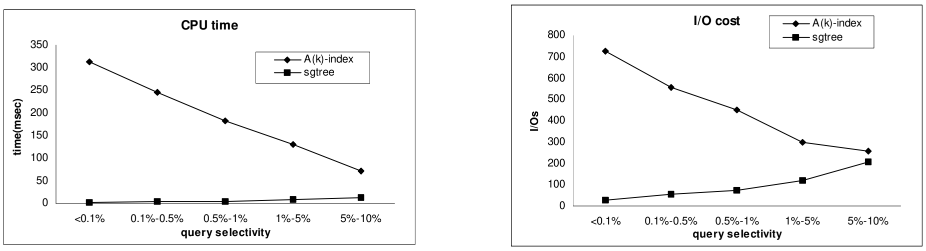
For each document set, we generate five groups of queries with 100 queries in each groups. The selectivity of all queries in group one is smaller than 0.1%; The selectivity of all queries in group two is from 0.1% to 0.5%; The selectivity of all queries in group three is from 0.5% to 1%; The selectivity of all queries in group four is from 1% to 5%; The selectivity of all queries in group five is from 5% to 10%;

Our generation process ensures the generated queries satisfy the selectivity requirement for each group.

5.2 SG-tree V.S. A(3)-index

In this set of experiments we compare SG-tree with A(3)-index from two perspective, CPU time and I/O cost. The dataset contains about 25000 documents. Except the I/O cost, the CPU time for creating A(3)-index is about 12 hours, (It needs several days to create A(3)-index for a large dataset, so we only test it on a rather small dataset) while the SG-tree is created in 1 minute.

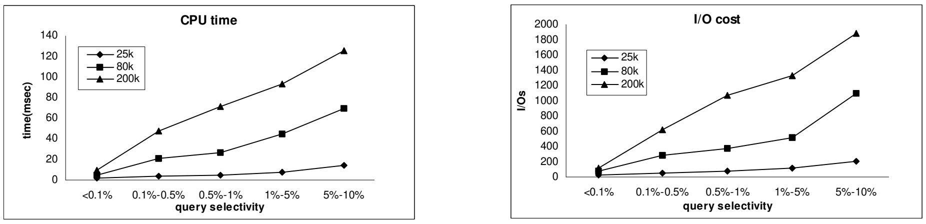
Figure 5 shows the average performance of the A(3)-index and the SG-tree varying the query selectivity. As expected, the performance of the SG-tree is much better than that of A(3)-index in both CPU and I/O cost, especailly for high and medium selective queries(<5%) Just like R-tree, the performance of SG-tree degrades as the query selectivity decreases. On the other hand, the I/O cost and CPU cost of the A(3)-index decreases. The reason is: the lower the selectivity of a query, the higher probability that query contains less pathes, therefore the total number of disk access may be reduced for get the partial result for all pathes in a query. The fewer the paths in a query, the less the CPU time required to merge the results of different paths to answer a query. Hence, it may not be a good idea of using SG-tree for queries with low selectivity(>10%).



(a) CPU time (b) I/O cost

Fig. 5. Average query performance of A(3)-index and SG-tree

Figure 6 shows the performance of SG-tree as the number of documents increases. This experiment clearly shows that SG-tree performs well on large datasets. Generally, SG-tree is more efficiently for high selective queries on a large dataset, The R-tree like structure makes it easy to update and organise a large number of signatures. Highly selective queries traverse only few nodes of SG-tree and all operations are bitwise during the traversal, which leads to low CPU and I/O cost.



(a) CPU time (b) I/O cost

Fig. 6. Scalability of SG-tree

6 Conclusions

In this paper we attempt to support XML queries from a different angle. Our goal is to define a lightweight index that reduces the size of the problem by filtering from consideration documents which may not possibly contain the query. To do this, we extract some components from the XML documents, and create bit-string representations of the documents based on the extracted components. The signatures are indexed using the SG-tree, an R-tree like structure. The incoming query is also transformed into a bit-signature according to the extraced components, which is used as a key to travse the SG-tree to retrieve all document signatures that contain it. This process may filter out a large percentage of documents that do not satisfy the query. The query is finally evaluated on the candidates using techniques from previous work.

Our methodology is similar to the 1-index [10] and $A(k)$-index [7] in what we use structural summaries to index the data. However, our proposed index is much faster and it occupies less space. The creation and maintenance costs are also lower. In the future, we will study its application on more complex structures beyond edges.

References

1. S. Al-Khalifa, H. V. Jagadish, N. Koudas, J. M. Patel, D. Srivastava, and Y. Wu. Structural joins: A primitive for efficient xml query pattern matching. In *International Conference on Data Engineering*. IEEE Computer Society, 2002.
2. N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: Optimal xml pattern matching. In *SIGMOD Conference*. ACM Press, 2002.
3. C.-W. Chung, J.-K. Min, and K. Shim. Apex: An adaptive path index for xml data. In *SIGMOD Conference*. ACM Press, 2002.
4. R. Goldman and J. Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *VLDB Conference*, pages 436–445. Morgan Kaufmann, 1997.
5. A. Guttman. R-trees: A dynamic index structure for spatial searching. In B. Yormark, editor, *SIGMOD Conference*, pages 47–57. ACM Press, 1984.
6. R. Kaushik, P. Bohannon, J. F. Naughton, and H. F. Korth. Covering indexes for branching path queries. In *SIGMOD Conference*. ACM Press, 2002.
7. R. Kaushik, P. Shenoy, P. Bohannon, and E. Gudes. Exploiting local similarity for efficient indexing of paths in graph structured data. In *International Conference on Data Engineering*. IEEE Computer Society, 2002.
8. M. Ley. Dblp computer science bibliography database. http://www.informatik.uni-trier.de/ ley/db/.
9. Q. Li and B. Moon. Indexing and querying xml data for regular path expressions. In *VLDB Conference*, pages 361–370. Morgan Kaufmann, 2001.
10. T. Milo and D. Suciu. Index structures for path expressions. In C. Beeri and P. Buneman, editors, *International Conference on Database Theory*, volume 1540 of *Lecture Notes in Computer Science*, pages 277–295. Springer, 1999.
11. J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D. J. DeWitt, and J. F. Naughton. Relational databases for querying xml documents: Limitations and opportunities. In *VLDB Conference*, pages 302–314, 1999.
12. W3C. Xml path language (xpath). http://www.w3.org/TR/xpath.
13. W3C. Xml query language (xquery). http://www.w3.org/TR/xquery.