Indexing Useful Structural Patterns for XML Query Processing

Wang Lian, Nikos Mamoulis, David Wai-lok Cheung, *Member*, *IEEE Computer Society*, and S.M. Yiu

Abstract—Queries on semistructured data are hard to process due to the complex nature of the data and call for specialized techniques. Existing path-based indexes and query processing algorithms are not efficient for searching complex structures beyond simple paths, even when the queries are high-selective. We introduce the definition of minimal infrequent structures (MIS), which are structures that 1) exist in the data, 2) are not frequent with respect to a support threshold, and 3) all substructures of them are frequent. By indexing the occurrences of MIS, we can efficiently locate the high-selective substructures of a query, improving search performance significantly. An efficient data mining algorithm is proposed, which finds the minimal infrequent structures. Their occurrences in the XML data are then indexed by a lightweight data structure and used as a fast filter step in query evaluation. We validate the efficiency and applicability of our methods through experimentation on both synthetic and real data.

Index Terms—Query processing, XML/XSL/RDF, mining methods and algorithms, document indexing.

1 Introduction

The efficient support of queries on semistructured data is one of the hottest research topics in the database community, as XML is becoming the standard for information exchange over the Internet. As opposed to relational data, which are flattened into tables, the key feature of XML data is their loosely defined structure, which is usually represented by trees or graphs.

Typical queries on semistructured data ask for documents which contain a query structure and are expressed in a language such as XPath. An example *path* query $//conference/author$ retrieves information from documents containing an element *conference*, with a child element *author* in their hierarchical representation. Query $//conference[author\ and\ title]$, as another example, retrieves information from documents containing a small tree (*twig*) with parent element *conference* and two children *author* and *title*.

Several indexes [6], [10], [13], [14], [19], [20] have been proposed to answer path queries, however, they have certain drawbacks: 1) their sizes are usually large and 2) they are designed for path queries only. Thus, in order to answer tree-structured queries, they have to split the query into paths, join the result sets from each path, and finally validate the join results to remove false hits. This can be a time consuming process, especially when the intermediate results from each path are large. For example, consider a database with 25,000 documents extracted randomly from the DBLP XML archive [30]. Assume that a query asks for the titles of all $inproceedings$ papers which have two $crossref$ subelements. If we use a path index to evaluate this query, we have to split it in two paths: $//inproceedings/title$ and $//inproceedings/crossref$, retrieve the partial results and join them to generate results, as illustrated in Fig. 1. However, the selectivity of the two path queries is very low[1] (i.e., $inproceedings/title$ appears in 12,052 documents and $inproceedings/crossref$ in 6,402 documents). On the other hand, the query is high-selective since it has only two answers. It would be convenient to have an index for subtree structures of high selectivity, which can be used to quickly evaluate arbitrary structural queries.

In general, it is impractical to explicitly index the locations of all possible structures found in a database of XML documents. We observe that typical queries retrieve only a small portion of the data, generating, however, a large number of intermediate results, when path indexes are used. Therefore, a good idea would be to index structures which are *infrequent* in the database. Nevertheless, we cannot index all infrequent structures since the number of these structures could be huge.

Notice that all superstructures of an infrequent structure (IS) are also infrequent. Moreover, it would not give us much benefit to index an IS s_i if it has a substructure s_j which is also infrequent because s_j is contained in every query that contains s_i, and it already has high selectivity. Thus, there is a set of interesting structures, which are infrequent and whose substructures are all frequent. We call these *Minimal Infrequent Structures* (MIS). Note that if a query has high selectivity, it has to contain at least one MIS. Thus, by indexing all MIS, we can quickly evaluate queries of high selectivity even if they contain low-selective components. We just need to find the MIS contained in the query and merge-join their *document_id* lists. Our index will not help the evaluation of low-selective queries, but it can serve as a tool to indicate the fact that they are low-selective. Such queries are expensive anyway and left to be handled by conventional techniques. Users may even withdraw submitted queries once they know that they produce a large set of results.

- *W. Lian is with the Faculty of Information Technology, Macao University of Science and Technology, Avenida Wai Long, Taipa, Macao. E-mail: lwang@must.edu.mo.*
- *N. Mamoulis, D.W.-l. Cheung, and S.M Yiu are with the Department of Computer Science, University of Hong Kong, Pokfulam Road, Hong Kong. E-mail: {nikos, dcheung, smyiu}@cs.hku.hk.*

Manuscript received 13 May 2004; revised 17 Dec. 2004; accepted 3 Mar. 2005; published online 18 May 2005.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number TKDE-0143-0504.

[1] Throughout the paper, we use "high-selective" or "very selective" to characterize a query with few results and "low-selective" to characterize a query with many results.

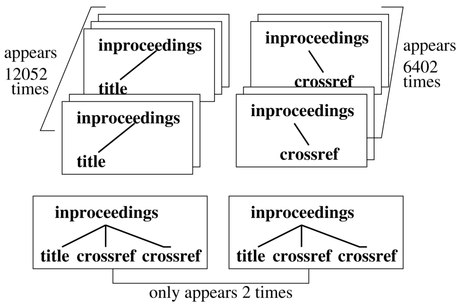


Fig. 1. A case where path indexes fail.

Another reason for indexing MIS is that if a query itself is an MIS, it would be expensive to be processed by a path index, since all substructures (i.e., paths) of it will have low selectivity (otherwise, the whole query structure would not be an MIS). In this case, the path index will have to merge a large part of intermediate results, which join to only few final results.

An additional advantage of the proposed index is that it is lightweight. The number of MIS is usually small enough to be held in memory. We organize them into a specialized inverted file and use their edges to quickly identify the MIS contained in a query. Query processing is restricted to a small range of data which bound the MIS in the query.

To find the set of MIS in a set of documents, we use a data mining algorithm. Our method discovers MIS by mining frequent structures and deriving extensions of them, which are not frequent. The problem of mining frequent tree structures in semistructured data has been studied before and several methods have been proposed [3], [16], [25], [28]. Most of them are based on the classic Apriori technique [2], which generates candidate frequent structures and validates them level by level. However, because these methods are rather slow when applying to find MIS (the number of MIS is rather small, whereas the number of frequent structures is very large, a lot of time is spent on counting the support of frequent structures).

In order to accelerate the discovery of MIS, we apply data mining in three phases. In phase one, we scan the data to derive an edge-based summary (signature) of each XML document. In phase two, we run Apriori on the summaries to quickly generate a set of large candidate frequent structures. In phase three, we run Apriori on the actual frequent structures, but at each level we do not need to count the support of structures that are contained in the frequent structures discovered in the second phase; we already know that they are frequent. To avoid sequential scan in phase two counting, we employ the SG-tree [18], an R-tree-like structure for indexing signatures. Experimental results demonstrate that this three-phase data mining algorithm achieves a significant speed improvement compared to the direct use of the Apriori algorithm.

We also evaluate the effectiveness of the MIS index. Our method shows an order of magnitude speed-up compared to path indexes and relational database approaches, for queries that contain MIS. Finally, we discuss and experimentally evaluate the effectiveness of MIS indexing, when values (not only elements) are considered. Putting everything together, the contribution of this paper is two-fold:

- We introduce the framework of indexing minimal infrequent structures for efficient query processing on XML data. Our method is not a generic index for structural queries, but serves as a fast filter that guides the evaluation of high-selective queries and identifies low-selective ones.
- We propose an efficient three-phase data mining process that discovers frequent structures (and MIS) in XML data. The algorithm is independent and can be used as a generic graph-mining tool that, for instance, can discover frequent compounds in chemical structures [9].

The rest of the paper is organized as follows: Section 2 discusses previous work related to the XML data indexing, mining, and query processing. In Section 3, we formally define the minimal infrequent structures and present the mining algorithm that finds all MIS in a collection of documents. Section 4 describes the in-memory index for MIS and shows how it can be used for query evaluation. In Section 5, we discuss a simple extension to process values in MIS. In Section 6, we evaluate our proposed methods on both synthetic and real data. Finally, Section 7 concludes the paper with a discussion about future work.

2 Related Work

Previous work on structured XML data indexing has mainly focused on paths. DataGuide [19] and the 1-index [20] summarize the path information of every absolute path starting from the root element. These indexes have two major limitations; first, they are suitable only for *absolute* path queries, where the first element in the path query is the root element of the documents. Second, their sizes are usually very large, comparable to the size of the indexed data. The F&B-Index [14] is a similar structure, which summarizes XML data to allow both forward and backward traversal, as opposed to DataGuide and the 1-index, which allow only forward traversal. In [13] another path index, called A(k)-index, which provides accurate results to all path queries of length up to k, is proposed. This parameter trades off between accuracy and space. However, the A(k)-index is still path-based, which means that it cannot efficiently process queries beyond simple paths. The D(k)-index [6] is similar to the A(k)-index, but, in addition, it can be updated according to the change of query load. APEX [8] is yet another path index that can be updated with the change in the query load.

The above path indexes can provide answers for simple path queries. More complex queries (e.g., tree queries) must be decomposed into a set of paths in order to utilize the index. Optimization of such queries is based on defining a good execution plan, which evaluates the most selective paths first. However, by applying path-based query evaluation, we might not be able to find a high-selective path of the query in many cases, simply because the high-selective parts may be trees. Therefore, path indexes can produce many large intermediate results, which significantly increase the query execution cost.

In this paper, we index only the Minimal Infrequent Structures (MIS) whose substructures are all frequent. To find all MIS, we first find the *maximal* frequent structures. For these, we need to apply a data mining (i.e., counting) algorithm. There are several algorithms for mining frequent structures in graphs [3], [16], [25], [28], based on Apriori [2]. Starting from frequent vertices, the occurrences of more complex structures are counted by adding an edge to the frequent structures of the previous level. The major difference among these algorithms is on the candidate generation and the counting processes.

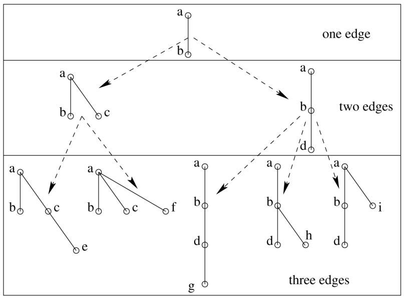


Fig. 2. Candidate generation.

A mining technique that enumerates subtrees in semistructured data efficiently, and a candidate generation procedure that ensures no misses, was proposed in [3]. The tree enumeration works as follows: For each frequent structure s, the next-level candidate subtrees with one more edge, are generated by adding frequent edges to its rightmost path. Thus, we first locate the right most leaf r, traverse back to the root and extend each node visited during the backward traversal. For example, in Fig. 2, edge a/b which is denoted by $(a(b))$, is first extended to $(a(b(d)))$ (assuming that $(b(d))$ is a frequent edge), then $(a(b))$ is extended on a to form $(a(b)(c))$ (assuming $(a(c))$ is also frequent).[2] In the next level, frequent structures with two edges (e.g., $(a(b)(c))$, $(a(b(d)))$) are extended to generate the new candidates.

This technique enumerates the occurrences of trees relatively quickly, but fails to prune candidates early, since the candidate generation is based only on frequent edges. A similar candidate generation technique is applied in TREEMINER [28]. This method also fails to prune candidates early, although the counting efficiency for each candidate is improved with the help of a special encoding schema.

In [16], simple algorithms for canonical labeling and graph isomorphism are used, but they do not scale well and cannot be applied to large graphs. In [25], complicated pruning techniques are incorporated to reduce the size of candidates, however, the method discovers only trees starting from the root, rather than arbitrary frequent trees. Our work is also related to FastXMiner [27], which intelligently arranges frequently asked query patterns in the system cache for enhancing query performance. This method is also Apriori-like, however, it only discovers frequent query patterns rooted at the root of DTD, whereas our mining algorithm discovers patterns rooted at any level of a DTD.

Our data mining method uses the enumeration method in [3] to generate the candidates level-by-level, but we apply more effective pruning techniques to reduce the number of candidates; a generated candidate is pruned if any of its substructures are not in the set of frequent structures generated in previous level. Furthermore, we use a novel approach which employs a tree of document signatures [18] to quickly identify possible large frequent structures of a particular size. The mining algorithm on the exact structures then counts only candidates which are not substructures of the frequent structures already discovered. This greatly reduces the number of candidates that need to be counted and speeds up the mining process significantly. The next section describes the new data mining method in detail.

Even though relational database technology is not well-tuned for semistructured data such as XML, it may still be a practical approach to manage XML documents and support XML queries. Relational database techniques decompose XML documents into tables, and queries are processed by joining tables using the structural relationships between elements and values [4], [12], [15], [23], [24], [29]. In [23], the DTD is used to generate the database schema, whereas four tables: $element$, $path$, $attribute$, and $text$ are used to store decomposed information from documents in [24]. Queries are translated to SQL before evaluation.

In [29], only two tables, $ELEMENTS$ and $TEXT$, are used to store all element and value information, respectively. A simple, preorder-based, numbering scheme is introduced to capture the element-element and element-value containment relationships. This scheme allows the direct validation of ancestor-descendant relationships and facilitates the use of efficient sort-merge join algorithms in query processing. In the same direction, several, progressively more efficient merge-join algorithms were proposed in [4], [12], [15]. In [15], a stack is used to optimize the cost for evaluating an ancestor-descendant relationship. This algorithm was further extended for path and twig queries in [4], whereas [12] employs an index to selectively scan only parts of the tables that match with the partial result of the queries. The efficiency of all these methods still depends on the selectivity of the *paths* that are involved in the query, thus high-selective queries which contain low-selective paths are still expensive to process.

Our work is also related to query optimization methods for XML data management systems. In [1], Markov-based methods are used to estimate the selectivity of path queries. Chen et al. [5] propose techniques that probabilistically count the occurrences of twig patterns in XML files. Polyzotis and Garofalakis [21] are the first to construct statistical synopses of graph-structured XML data, in contrast to previous work, which only handles tree-structured data. Lim et al. [17] collects the query results to produce online statistics, which can be used to estimate the selectivity of path expressions. The above methods focus on summarizing statistics about frequent structures of XML documents to be used in query selectivity estimation, while ignoring infrequent structures to save space. On the other hand, we extract MIS, to speed-up the processing of the high-selective queries containing them. Our work is also related to ViST [26]. This approach builds a complicated index based on a number of B^+-trees. Queries can be answered by only traversing these B^+-trees, avoiding expensive joins on tables required by most other methods. However, the number of B^+-trees that might be traversed for a query are usually large (except for every simple queries). Our method is very efficient, no matter how complex a query is; a simple access to a lightweight MIS-based index brings fast all possible documents that satisfy it.

3 Discovery of MIS

In this section, we formally define the minimal infrequent structures (MIS) which occur in a collection of XML documents. Then, we propose a three-phase methodology that extracts the patterns as well as their occurrences

[2] Note that there is a one-to-one mapping from a tree s to a string $ts(s)$, where parentheses are used to denote the nesting of elements in a structure. We will extensively use this kind of presentation in this paper. For example, the leftmost candidate with three edges in Fig. 2 can be represented as $(a(b)(c(e)))$.

followed by some optimizations on the mining process. Finally, we discuss the complexity of the overall mining process.

3.1 Problem Definition

Let L be the set of labels found in an XML database. A *structure* is a node-labeled tree, where nodes are labels from L. Given two structures, s_1 and s_2, if s_1 can be derived by removing recursively $l \geq 0$ nodes (which are either leaf nodes or root nodes) from s_2, then s_1 is a *substructure* of s_2. In this case, we also say that s_2 *contains* s_1, or that s_2 is a *superstructure* of s_1. Finally, the *size* of a structure s is defined by the number of edges in it. If a structure contains only one element, we assume the size of it is zero. Assuming that $L = \{a, b, c, d, e\}$, two potential structures with respect to L are $s_1 = (a(b)(c(a)))$ and $s_2 = (a(c))$; s_2 is a substructure of s_1 or s_1 contains s_2.

Definition 1. *Given a set D of structures, the* support $sup(s)$ *of a structure s in D is defined as the number of structures in D, which contain s. Given a user input threshold ρ, if $sup(s) \geq \rho \times |D|$, then s is* frequent *in D, otherwise, it is* infrequent. *A structure s is a* Minimal Infrequent Structure *(MIS) if $sup(s) < \rho \times |D|$ and: 1) $size(s) \geq 1$ and for each substructure s_x of s, $sup(s_x) \geq \rho \times |D|$, or 2) $size(s) = 0$.*

Our data mining task is to discover all MIS in a document collection D. Therefore, the set D in Definition 1 can be regarded as a set of documents. Since some MIS could be arbitrarily large and potentially not very useful for query evaluation, we restrict our search to structures up to a maximum number of k edges. Thus, we define our problem formally as follows.

Definition 2 (problem definition). *Given a document set D and two user input parameters ρ and k, find the set S of all MIS with respect to D, such that for each $s \in S$, $size(s) \leq k$.*

The following theorem implies that the set of MIS can be used to answer any high-selective query with up to k edges.

Theorem 1. *Let D be a document set and S be the set of MIS in D with respect to ρ and k. If a query q contains at most k edges and it is an infrequent structure with respect to ρ, then it contains at least one MIS.*

Proof. In the cases where $size(q) = 0$ or all substructures of q are frequent, q itself is an MIS, thus it should be contained in the set of MIS. Now, let us examine the case where q has at least one infrequent proper substructure q'. If q' is MIS, then we have proven our claim. Otherwise, we can find a proper substructure of q' which is infrequent and apply the same test recursively, until we find an MIS (recall that a single node that is infrequent is an MIS of size 0). □

Finally, in order to accelerate the data mining process, we use document abstractions, called *signatures*, which are defined as follows.

Definition 3. *Assume that the total number of distinct edges in D is E, and consider an arbitrary order on them from 1 to E. Let $order(e)$ be the position of edge e in this order. For each $d \in D$, we define an E-length bitmap, $sig(d)$, called* signature; *$sig(d)$ has 1 in position $order(e)$ if and only if e is present in d. Similarly, the signature of a structure s is defined by an E-length bitmap $sig(s)$, which has 1 in position $order(e)$ iff e is present in s.*

The definition above applies not only to documents, but also to structures. Observe that if s_1 is a substructure of s_2, then $sig(s_1) \subseteq sig(s_2)$ (or else $sig(s_1) \wedge \neg sig(s_2) = 0$). Thus, signature can be used as a fast check on whether or not a structure can be contained in a document. On the other hand, it is possible that $sig(s_1) \subseteq sig(s_2)$ and s_1 is not a substructure of s_2. For instance, $s_1 = (a(b(c))(b(d)))$ and $s_2 = (a(b(c)(d)))$ contain the same set of edges $\{a/b, b/c, b/d\}$ (and, therefore, have identical signatures), however, s_1 is not a substructure of s_2 (as s_1 contains two bs and s_2 only one). As a result, we can only use the signature to find an *upper bound* of a structure's support in the database.

3.2 Mining MIS

We solve the problem of mining MIS by a three-phase process. The first *preprocessing* phase is simple; the document collection D is scanned once for two tasks. The first task is to count the frequencies of all elements and edges. After reading the database, all infrequent elements and edges together with all frequent edges become available. All MIS of size zero and one are inserted into M, the set for storing all MIS. The set of frequent edges FE is stored separately, which will be used in the next phases of the algorithm for generating more complex candidate frequent structures. The second task of the preprocessing phase is to compute the signatures of the documents. We do not store the signature of each document, but rather compress this information by increasing a counter for every signature we encounter. By doing so, documents which share the same signature are encoded together. This results in a set SG, of signatures g, which exist in the database, and a counter $sup(g)$ for each of them. For now, we assume that this information is compact enough to fit in memory, since we expect that many documents share the same signature. Later, we discuss how to use the SG-tree, a hierarchical index for signatures, not only to deal with cases where SG does not fit in memory, but also to speed-up counting in the second phase of the mining process.

The second phase of the data mining process, called *signature-based counting*, is described by the pseudocode in Fig. 3. All frequent structures of size k are found and stored in a set F_k (recall that k is a user input parameter which defines the maximum size of MIS that we want to mine). Note that we only mine F_k and not any F_i, $i < k$ in this phase. The motivation is that if we quickly discover some large frequent structures, we can avoid counting the occurrences of all their substructures when running the exact data mining algorithm during the third phase of the MIS discovery process.

The algorithm in Fig. 3 computes the frequent structures in an Apriori-like, level-wise fashion. The supports of the structures, however, are not counted by scanning the documents; instead, the supports of their signatures in SG are used to derive an upper bound for their actual occurrences.

We use the frequent structures of the previous level (F_{prev}) to generate candidate structures for the current level according to the tree enumeration method in [3]. However,

```
/* Input  int $k$, the maximum number of edges in a MIS*/
/* Input  $\rho$, frequent threshold,*/
/* Input  $FE$, frequent edges,*/
/* Input  $D$, the document set*/
/* Input  $SG$, distinct documents' signatures with counters*/
/* Output  $F_k$, all size $k$ frequent structures*/
1).   $F_{prev} = FE$
2).   for $i$=2 to $k$
3).       $candidates = genCandidate(F_{prev}, FE)$
4).       $bitprune(candidates)$
5).       if $candidates == \emptyset$ then break
6).       $F_{prev} = \emptyset$
7).       for each $g$ in $SG$
8).           for each $c$ in $candidates$
9).               if $sig(c) \subseteq g$ then
10).                  $sigsup(c)$+=$sup(g)$
11).                  if $sigsup(c) \geq \rho \times |D|$ then
12).                      $candidates.remove(c)$; insert $c$ into $F_{prev}$
13).  scan documents to count the support of each $c \in F_{prev}$
14).  for each $c$ in $F_{prev}$
15).      if $sup(c) \geq \rho \times |D|$ then insert $c$ into $F_k$
16).  return $F_k$
```

Fig. 3. The mining algorithm—phase 2.

we include an additional optimization. Function $bitprune()$ prunes those candidates for which the signature of some substructure is infrequent with respect to SG. For each remaining candidate c, we calculate its *signature-based* support, $sigsup(c)$, by adding the counters of all signatures in SG that contain $sig(c)$. Lines 11-12 optimize the counting process by avoiding counting the support of candidates which have already exceeded the threshold $\rho \times |D|$. The process continues until all frequent structures of size k (according to the signatures) have been discovered. Finally, we scan the document set once to derive the *actual* support of all these structures (line 13). Although, many of them may turn out to be infrequent in terms of their actual support in the documents, this process does not miss any frequent structures if size k, which are eventually stored in F_k. This is ensured by Theorem 2 below.

Lemma 1. *If a structure s is frequent with respect to D, then its signature, $sig(s)$, is also frequent with respect to SG.*

Theorem 2. *In line 13 of the algorithm in Fig. 3, F_{prev} contains all frequent structures of size k.*

Proof. Whether F_{prev} contains all frequent structures of size k is affected by only two functions: $genCandidate()$ and $bitprune()$. Function $genCandidate()$ enumerates all possible superstructures of the structures from the previous level F_{prev} without misses, as shown in [3]. From Lemma 1, we can derive that if a structure's signature is infrequent, then the structure is also infrequent in D. Since $bitprune()$ only removes those structures having some substructure with an infrequent signature, actual frequent structures will not be missed.□

The final, *structure counting* phase of our methodology finds all MIS in D, as described by the pseudocode in Fig. 4. The key point of this algorithm is in lines 7-9; whenever we detect a candidate which is a substructure of a k-sized frequent structure in F_k, we can skip the counting of its occurrences in D and directly insert it into F_{prev} for the next-level candidate generation. The function $genCandidate()$ is the same as that in Fig. 3.

```
/* Input  int $k$, the maximum number of edges of a structure*/
/* Input  $\rho$, frequent threshold,*/
/* Input  $FE$, frequent edges,*/
/* Input  $F_k$, all size $k$ frequent structures*/
/* Input  $D$, the document set*/
/* Output  $M$, the set of MIS up to size $k$*/
1).   $F_{prev} = FE$; $M = \emptyset$
2).   for $i$=2 to $k$
3).       $candidates = genCandidate(F_{prev}, FE)$
4).       $prune(candidates, F_{prev})$
5).       if $candidates == \emptyset$ then break
6).       $F_{prev} = \emptyset$
7).       for each $c$ in $candidates$
8).           if $c$ is a substructure of an structure in $F_k$ then
9).               $candidates.remove(c)$; insert $c$ into $F_{prev}$
10).      for each document $d$ in $D$
11).          for each $c$ in $candidates$
12).              if $sig(c) \subseteq sig(d)$ then
13).                  if $c$ is in $D$ then $sup(c) = sup(c)$+1
14).                  if $sup(c) \geq \rho \times |D|$ then
15).                      $candidates.remove(c)$; insert $c$ into $F_{prev}$
16).      for each $c$ in $candidates$
17).          if $sup(c) \geq 0$ then insert $c$ into $M$
18).  return $M$
```

Fig. 4. The mining algorithm—phase 3.

Function $prune()$ removes all size $i+1$ candidates which have one or more size i infrequent substructures by checking membership of F_{prev}. For this pruning check, we do not apply a slow subtree matching routine, but use the one-to-one mapping from a tree s to a string $ts(s)$ (as we have mentioned in footnote 2, $ts(s)$ captures all structural information of s) and perform pruning based on $ts(s)$. Given a candidate structure c, we know that the substructures in the previous level can be obtained by removing one of its leaf nodes. This operation can be performed cheaply on $ts(c)$. Leaf nodes can be identified by finding parentheses containing a single element without any nested subelements. By removing such an element, we get the $ts(c')$ of a substructure c' of c. Thus, by scanning the $ts(c)$ once, we can find the strings of all c's substructures at the previous level.

Fig. 5 describes an implementation of function $prune()$ that avoids subtree matching by using strings. It first derives all strings of the frequent structures from the previous level and organizes them in a hash table (lines 1-3). Then, for each candidate c, the strings of all its substructures are quickly generated, using the technique described above (lines 4-6). Finally, the hash function is applied on the string of each substructure of the candidate to determine whether or not it is in F_{prev}. The candidate is pruned as soon as one of its substructure is missing from the hash table (lines 7-12).

Finally, the algorithm of Fig. 4 computes the exact set of MIS, as shown by Theorem 3 below.

Theorem 3. *The algorithm of Fig. 4 finds all MIS up to size k correctly.*

Proof. It is clear that each structure c, inserted in MIS (line 17 of Fig. 4), is infrequent, since 1) it exists in the documents set and 2) its support is smaller than ρ, as indicated by the counting process. Thus, no frequent structures are returned by the algorithm. In addition, c can be inserted in MIS only if it passes the pruning function (line 4). This

```
/* Input $F_{prev}$, frequent structures in previous level*/
/* Input $candidates$, generated from $F_{prev}$*/
for each $s$ in $F_{prev}$
    $ts(s) = get\_treeString(s)$
    insert $ts(s)$ into hash-table $H$
for each $c$ in $candidates$
    $ts(c) = get\_treeString(c)$
    $STS = gen\_Sub\_treeString(ts(c))$
    $found$=true
    for each $str$ in $STS$
        $exists = H.get(str)$
        if not $exists$ then
            $found$=false; break
    if not $found$ then $candidate.remove(c)$
```

Fig. 5. Algorithm $prune()$ for candidates.

means that all substructures of c are frequent. Finally, all MIS are one-edge extensions of frequent structures by definition, therefore, they are generated by line 3 from a frequent structure of the previous level. No candidates (and, therefore, no MIS) are missed by the algorithm according to [3]. Thus, the algorithm computes all MIS correctly. □

3.3 Optimizations

In this section, we describe several optimizations that can improve the mining efficiency.

3.3.1 First Strategy: Use Distinct Children-Sets

During the first scan of the data set, for each element we record every distinct set of children elements found in the database. For example, consider the data set of Fig. 6, consisting of two document trees. Observe that element a has in total three distinct children sets; $((a)(b))$, $((c)(d))$, and $((d)(f))$. The children sets $((a)(b))$, $((c)(d))$ are found in doc1, and the children sets $((a)(b))$, $((d)(f))$ are found in doc2. When an element a having d as child is extended during candidate generation, we consider only f as a new child for it. This technique greatly reduces the number of candidates since generation is based on extending the frequent structures by adding edges to their rightmost path.

3.3.2 Second Strategy: Stop Counting Early

Notice that we are searching for MIS rather than frequent structures, thus we are not interested in the exact support of frequent structures. During the counting process, when the support of a candidate is greater than the threshold, the structure is already frequent. Thus, we do not need to count it anymore; it is immediately removed from $candidates$ and inserted to F_{prev}. This heuristic is implemented in lines 11-12 in Fig. 3 and lines 14-15 in Fig. 4.

3.3.3 Third Strategy: Counting Multiple Levels of Candidates

After candidate pruning in lines 4 and 7-9 of Fig. 4, if the number of remaining ones is small, we can directly use them to generate the next level candidates and count two levels of candidates with a single scan of the documents. This can reduce the I/O cost at the last phases of the data mining process.

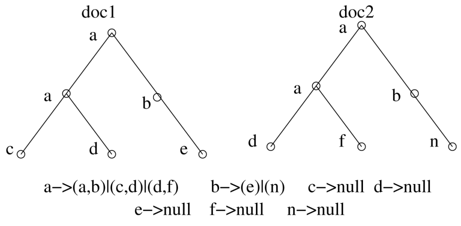


Fig. 6. Summary of children-sets.

3.3.4 Fourth Strategy: Using the SG-Tree in Phase-Two Counting

In Fig. 3, we obtain the supports of candidates by sequentially comparing their signatures to those of all documents. This operation is the bottleneck of the second phase in our mining algorithm. Instead of comparing each candidate with all document signatures, we can employ an index for document signatures, to efficiently select only those that contain a candidate.

The SG-tree (or *signature* tree) [18] is a dynamic balanced tree similar to R-tree for signatures. Each node of the tree corresponds to a disk page and contains entries of the form $\langle sig, ptr\rangle$. In a leaf node entry, sig is the signature of the document and ptr stores the number of documents sharing this signature. 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 *contains* all signatures in the subtree pointed by it. All nodes contain between c and C entries, where C is the maximum capacity and $c \geq C/2$, except from the root which may contain fewer entries. Fig. 7 shows an example of a signature tree, which indexes nine signatures. 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.

The tree can be used to efficiently find all signatures that contain a specific query signature (in fact, [18] have shown that the tree can also answer *similarity* queries). For instance, if $q = 000001$, the shaded entries of the tree in Fig. 7 are the qualifying entries to be followed in order to answer the query. Note that the first entry of the root node does not contain q, thus there could be no signature in the subtree pointed by it that qualifies the query.

In the first phase of our mining algorithm, we construct an SG-tree for the set SG of signatures, using the optimized algorithm of [18] and then use it in the second phase to facilitate counting. Thus, lines 7-12 in Fig. 3 are replaced by a depth-first search in the tree for each candidate c that counts the number of document signatures that contain c. As soon as this number reaches the threshold $\rho \times |D|$, search stops and c are inserted into F_{prev}. Note that we use a slight modification of the original structure of [18]; together with each signature g in a leaf node entry, we store the number of documents $sup(g)$ having this signature (in replacement of the pointer in the original SG-tree).

3.4 Space and Time Complexity

The time and space complexity of the proposed mining algorithm is comparable to that of Apriori since our method

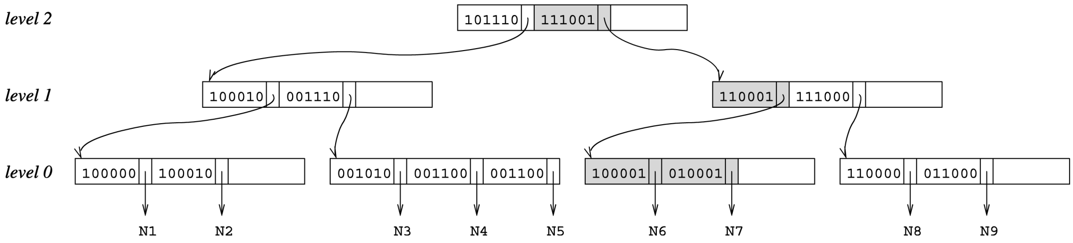


Fig. 7. Example of a signature tree.

is essentially Apriori-based. Like the conventional Apriori, we need (disk) space to accommodate the database and the candidates at each mining level. In addition to conventional Apriori, our method requires space for SG. However, the additional space is insignificant since signatures are very small compared to documents and $|SG|$ is typically much smaller than $|D|$.

The time complexity can be derived by summing up the costs of the three individual phases. Phase 1 requires one database scan to compute the signatures and the time to construct the SG-tree. The cost of this phase is expected to be dominated by the linear database scan since the number of distinct signatures $|SG|$ found is expected to be much smaller than $|D|$ resulting in a cheap $O(|SG| \log |SG|)$ tree construction cost. Phase 2 operates on SG only and performs a (logarithmic) search for each candidate at each level. Thus, its cost, described by $O(\sum_{i=2}^{k} |C_i| log |SG|))$, mainly depends on the number of candidates per level (exponential to $|SG|$ in the worst-case). Finally, Phase 3 applies Apriori on the actual set of document structures, so it is exponential to $|D|$ in the worst case. Nevertheless, in typical cases, our method is expected to be of acceptable cost and much faster compared to conventional Apriori because 1) it uses F_k to avoid counting candidates known to be frequent, 2) it uses $prune()$ to expedite subtree matching, and 3) it employs the optimizations discussed in Section 3.3. The experiments of Section 6 verify this assertion.

4 Query Evaluation

In the previous section, we have shown in detail how MIS is discovered. We now discuss how to use them for fast XML query evaluation. Given a query, we should find all MIS it contains and merge their $document\_id$ lists. The resulting $document\_id$ list contains all the documents that need to be further validated for the remaining components of the query.

Detecting all infrequent elements in a query involves a simple search of the query nodes in the set of MIS structures of size 0. In the following, we will consider MIS with one or more edges. A straightforward method is to scan the set of MIS once, checking whether or not each is in q. However, this process may be expensive if only a few MIS are contained in the query. Observe that if an MIS p is contained in q, the following conditions must be satisfied: 1) each edge e in p is also present in q, 2) the number of distinct edges in p is smaller than or equal to the number of distinct edges in q, and 3) $size(p) \leq size(q)$. According to these three properties, we propose an inverted index for MIS which helps to quickly find those structures that may be contained in the query. The candidate MIS that qualifies them are then matched against the query using a more expensive algorithm (i.e., tree matching).

The inverted index is built as follows: Assume that the number of MIS (with edges) is n. First, all MIS are sorted in ascending order of the number of distinct edges they contain. The order of MIS with the same number of distinct edges is insignificant. After sorting, each MIS gets an id, which indicates its position in the sorted list.

Assume that m is the number of distinct edges found in the set of MIS. The inverted index G is an $m \times n$ binary matrix, where each row is indexed by a distinct edge. Thus, row G_i is a bit-array of size n. If structure j contains edge i, then $G_{ij} = 1$, otherwise, $G_{ij} = 0$. In addition, another integer array I is used, storing in $I[j]$ the id of the last MIS in the sorted list which contains j distinct edges.

The algorithm of Fig. 8 is used to find the MIS contained in a query q. The first three lines are used to detect any infrequent elements in q. Lines 4-12 detect MIS which are potentially contained in q and lines 13-14 are used to verify them. In line 4, we assign the number of distinct edges in q to d, and $I[d]$ is used in line 9 to indicate the column of the matrix beyond which we do not need to search; if the distinct number of edges in an MIS is larger than the distinct number of edges in q, then the MIS cannot be contained in q. Line 11 checks whether or not all distinct edges in an infrequent structure appear in q. Line 12 is required to prune structures which have the same distinct edges as q, but whose total number of edges (including duplicates) is larger than the size of q.

To illustrate the functionality of the algorithm, consider the example of Fig. 9, depicting the matrix G for a database where there are two MIS with one distinct edge and two MIS with two distinct edges. Assuming that $q = (a(a)(b))$, the query is split into edges $(a(a))$ and $(a(b))$. Since q contains two distinct edges, only the MIS with one or two distinct edges are considered ($I_d = 4$, we need to search up to the fourth column of G). After summing the rows corresponding to the query edges $(a(a))$ and $(a(b))$ (lines 6-9), we get 0, 0, 2, 1, thus only MIS $(a(a)(b))$ satisfies the condition in line 11.

We now analyze the complexity of our MIS-based index. As already discussed, the space occupied by the index is $O(nm)$, where n is the number of MIS and m is the number of distinct edges in them. Given a query structure q, the worst-case time required to identify all MIS in q is $O(m_q n)$, where m_q is the number of distinct edges in q. If q contains more than one MIS we should merge the $document\_id$ of the

```
/* Input  $q, G, I$*/
/* Input  $M$ (an array stores all MIS)*/
/* Output  $qual$, all MIS in $q$*/
/* $X$, stores a set of MIS that may be contained in $q$ */
$qual = \emptyset$ and $X = \emptyset$
for each distinct element $ele$ in $q$
    if ($ele$ is an infrequent element) then insert $ele$ into $qual$
$d$=number of distinct edges in $q$
initialize an integer array $SUM$ of size $I[d]$
for each distinct edge $e$ in $q$
    $A = G(e)$ /*get the bit-array of edge $e$*/
    if $A \neq$ null /*some bits are not zero*/
        for $y$=1 to $I[d]$ do $SUM[y] = SUM[y] + A[y]$
for $i$=1 to $I[d]$
    if $SUM[i]$ = number of distinct edges in $M[i]$
        if $size(M[i]) \leq size(q)$ then insert $M[i]$ into $X$
for each structure $x$ in $X$
    if $x$ is in $q$ then insert $x$ into $qual$
return $qual$
```

Fig. 8. Finding MIS in a query.

n_q MIS in q at $O(n_q\rho|D|)$ worst-case cost (recall that an MIS can be contained in at most $\rho|D| - 1$ documents, otherwise, it is frequent). Finally, these documents must be accessed to verify whether they actually contain q at $O(c_q\rho|D|)$, where c_q is the average cost of verifying whether q is included in a document. In general, $O(c_q\rho|D|)$ dominates the overall cost, since expensive tree matching might be required for many documents, whereas using the MIS-based index is cheap and memory-based. As shown in our experiments, the number of documents that contain all MIS in q is usually much smaller than $\rho|D|$, resulting in very efficient search.

5 Processing Values

The reader may notice that so far we have considered only MIS that are structures of labeled nodes, which cannot be used for XML queries with selection predicates on values. Here, we show how we can easily extend our method to handle values. For each element (attribute) we have interest to consider values, we partition these values based on their similarity and interestingness to several groups. Each group is treated as a new element. For example, the years in publications could be partitioned into five groups: *before 1970*, *1970-1979*, *1980-1989*, *1990-1999*, and *2000 or later*. When parsing XML documents, each value is replaced with the element representing the group it falls in. Given a query, all value predicates in it are transformed in the same manner. In this way, our methodology can handle queries with values, as well. A similar segmentation technique for value ranges has been used in [17]. Notice that the set of discovered MIS with values will cover all MIS without values, which is guaranteed by Theorem 4. Therefore, mining MIS with values can only bring performance benefits to query evaluation, as we will show in the experimental section.

Theorem 4. *Given a set D of XML documents and k and ρ, if M is the set of MIS without considering values and MV the set of MIS considering values, then $M \subseteq MV$.*

Proof. The proof is straightforward, according to the definition of MIS. If a structure s is an MIS that does not contain values, then no other MIS can be generated after extending it with nodes that correspond to values. Therefore, s is an MIS in both cases where values are considered or not. □

edge\MIS	(b(e))	(b(n))	(a(a)(b))	(a(b)(d))
(a(d))	0	0	0	1
(b(e))	1	0	0	0
(b(n))	0	1	0	0
(a(a))	0	0	1	0
(a(b))	0	0	1	1
SUM	0	0	2	1

Fig. 9. Example of MIS matching.

Value extension allows us to support queries with predicates on values, however, it may also increase the cost of mining the MIS. Therefore, we recommend picking an interesting subset of all elements with values that are likely to be queried (based on the domain or query statistics) for the extension, rather than consider values for all elements.

6 Experimental Evaluation

In this section, we evaluate the effectiveness and efficiency of our methodology. We used real data from the DBLP archive [30]. In order to evaluate the robustness of our approach under various data and parameter settings, we also used synthetically generated data. We begin by describing the data generator. Then, we validate the efficiency of our data mining algorithm on both synthetic and real data. Based on the MIS discovered in real data, we demonstrate the performance improvements that the MIS index brings to query processing against a representative path index and two state-of-the-art relational database techniques. All experiments are carried out in a computer with 4 Intel Pentium 3 Xeon 700MHZ processors and 4G memory running Solaris 8 Intel Edition.

6.1 Synthetic Data Generation

We generated our synthetic data using the NITF (News Industry Text Format) DTD [31]. Table 1 lists the parameters used at the generation process.

The procedure that generates documents from a DTD is as follows: First, we parse the DTD and build a graph to keep the parent-children relationships and other information like the relationships between children. Then, starting from the root element r of the DTD, for each subelement, if it is accompanied by "*" or "+," we decide how many times it should appear according to a distribution (such as Poisson). If it is accompanied by "?," the element appears or not by tossing a biased coin. If there are choices among several subelements of r, then their appearance in the document follows a random distribution. The process is repeated on the newly generated elements until some termination thresholds, such as a maximum document depth, have been reached.

6.2 Efficiency of the Mining Algorithm

In the first set of experiments, we compare the total running cost (Including the I/O time) of our mining algorithm (denoted by MMIS) compared with the Apriori algorithm (denoted by MFS). Both algorithms are equipped with the first and second optimization strategies. The efficiency of the two techniques is compared with respect to three problem and algorithm parameters: 1) the support threshold, 2) the maximum size k of mined MIS, and 3) the number of documents.

TABLE 1
Input Parameters for Data Generation

Symbol	Interpretation	Value
N	total number of docs	10000–100000
W	distribution of '*'	Poisson
P	distribution of '+'	Poisson
Q	probability of '?' to be 1	a real between 0 and 1
Max	distribution of doc depth	Poisson

The synthetic data used in the experiments were generated by setting the parameters of the distribution functions to: $\mathrm{W} = 3$, $\mathrm{P} = 3$, $\mathrm{Q} = 0.7$, $\mathrm{Max} = 10$. Except experiments in Section 6.2.3, the generated documents were N=10,000 and the real data used were 25,000 documents randomly picked from the DBLP archive.

6.2.1 Varying the Support Threshold

Figs. 10 and 11 show the time spent by MMIS and MFS on the synthetic and real data, respectively. The support threshold varies from 0.005 to 0.1. We set $k = 10$ for the real data set and $k = 15$ for the synthetic one. Both figures show the same trend: As the support threshold increases, the improvement of MMIS over MFS decreases. This is because the number of frequent structures decreases, which degrades the effectiveness of F_k. The improvement in the DBLP data is smaller than the improvement in the synthetic data because these documents are more uniform. However, our three-phase method is still significantly faster than MFS.

Table 2 shows for $k = 15$, $\rho = 0.01$, the percentage of candidates in synthetic data, which we avoid counting at several levels of the mining algorithm because of the frequent structures F_k, found by the second phase of the mining process. Clearly, the advantage of having computed the frequent *k*-sized structures, before applying exact mining, provides a large benefit.

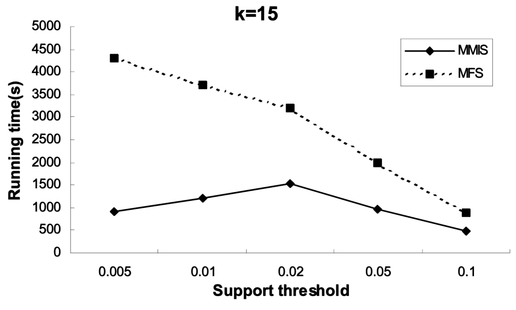


Fig. 10. Varying ρ (synth. data).

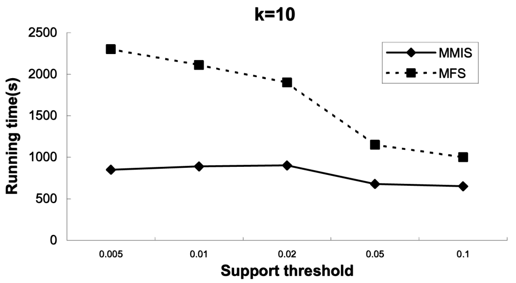


Fig. 11. Varying ρ (real data).

TABLE 2
Effectiveness of the Second Phase

# edges in candidates	8	9	10	11	12	13
% candidates not counted	84	85	90	90	95	97

6.2.2 Varying k

Figs. 12 and 13 show the time spent by MMIS and MFS on synthetic and real data, respectively, for various values of k and $\rho = 0.01$. Observe that as k increases, the speedup of MMIS over MFS increases. The reason for this is that when k goes beyond the peak point of candidates, the number of frequent structures in F_k is smaller while they can still prune most of the frequent candidates without counting.

Tables 3 and 4 show the number of MIS and nonpath MIS discovered for various k on the synthetic and real data set, respectively. The numbers unveil the applicability and usefulness of our approach. First, the total number of MIS in both cases is small enough to be easily accommodated and indexed in memory. Second, the majority of MIS found are not simple paths, but more complex tree structures. This indicates that for queries containing these MIS, path indexes will perform poorly (since all paths contained in the MIS are frequent), while our MIS index will manage to locate fast the high-selective part of the queries. Finally, observe that the number of MIS do not grow significantly with k, thus the mining process is fast even for very large k (e.g., even if we mine *all* MIS of any size). This can also be verified by Figs. 12 and 13.

6.2.3 Varying the Number of Documents

Figs. 14 and 15 show the time spent by MMIS and MFS on synthetic and real document sets of various cardinalities. For this experiment, $k = 10$ for real data and $k = 15$ for synthetic data, while $\rho = 0.01$ in both cases.

In both cases, the speedup of MMIS over MFS is maintained with the increase of problem size, showing that

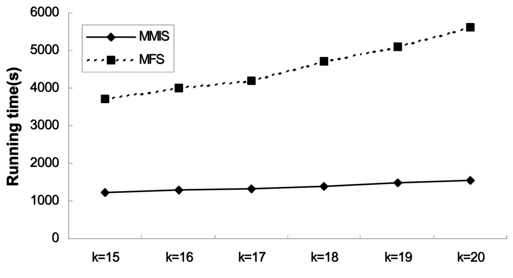


Fig. 12. Varying k (synth. data).

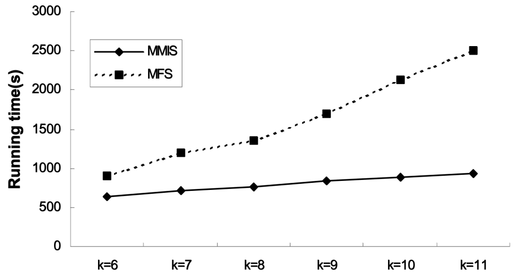


Fig. 13. Varying k (real data).

TABLE 3
MIS in Synthetic Data

k	15	16	17	18	19	20
No. of MIS	321	325	340	342	342	348
No. of non-path MIS	303	307	322	324	324	330

TABLE 4
MIS in Real Data

k	6	7	8	9	10	11
No. of MIS	69	73	75	78	78	81
No. of non-path MIS	54	58	60	63	63	66

MMIS scales well. Observe that for the DBLP data, the speedup actually increases with the problem size. This is due to the fact that DBLP documents have uniform structure and the number of distinct signatures does not increase much by adding more documents.

6.2.4 Mining MIS with Values and the Effect of Using the SG-Tree

In the next set of experiments, we evaluate the effectiveness and efficiency of the MMIS technique in discovering MIS with values on real data. We also show the effect of the optimization method that uses the SG-tree to count fast the supports of candidates in the second phase of MMIS (discussed in Section 3.3). We chose a subset of elements $\{author, editor, title, booktitle, journal\}$ and define new elements that correspond to ranges of their values. One can use statistical methods (e.g., access frequency) to determine the number of value ranges for each attribute and the distribution of values in them (e.g., see [17]). In this paper, in order to demonstrate the applicability of our technique and for the ease of exposition, instead of discovering an optimal value partitioning, we assume that 1) the number of ranges for each element with attribute value is 10 and 2) the number of values in each range is the same (uniform ranges).

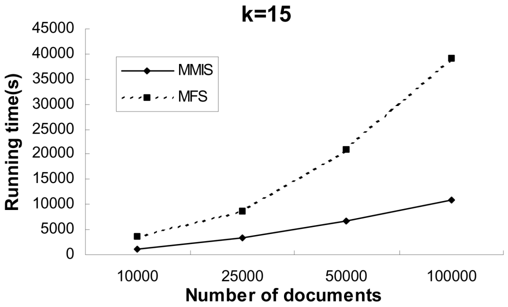


Fig. 14. Scalability (synth. data).

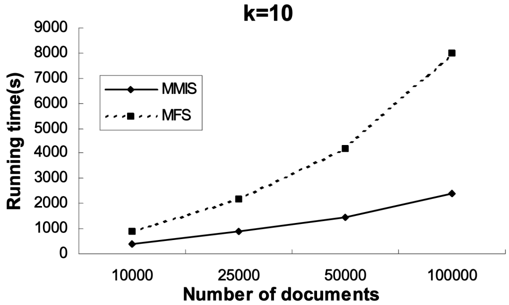


Fig. 15. Scalability (real data).

TABLE 5
Running Time of Different Algorithms on Different Support Threshold

k=11, D=25000, ρ=	0.005	0.01	0.02	0.05	0.1
MFS	18697	12001	10388	5273	4223
MMIS	7534	4423	4111	3803	2701
MMIS-SG–tree	5634	3525	3501	3652	2670

TABLE 6
Running Time of Different Algorithms on Different Number of Documents

k=11, ρ=0.01 D=	12500	25000	50000	100000
MFS	7012	12001	25145	54091
MMIS	2613	4423	8013	14253
MMIS-SG–tree	2100	3525	6876	12187

In the experiments, we compare the total running time (including the I/O time) of two versions of our mining technique 1) MMIS and 2) MMIS-SG-tree (which is MMIS equipped with SG-tree in the second phase) with MFS. In all experiments, the first and second optimization strategies discussed in Section 3.3 are applied.

First, we compare the time spent by the three methods for different values of ρ, after setting $k = 11$ and $D = 2,5000$. As Table 5 shows, MMIS performs very well for small ρ (≤ 0.02), but it loses its large performance gain for large values of this parameter. The result is consistent to that of Fig. 11, for the same reason; the number of frequent structures decreases with ρ and degrades the pruning effectiveness of F_k. For small values of ρ, the SG-tree provides significant performance gain in mining, while the impact of using the tree at search degrades as ρ increases. There are two reasons for this: 1) the number of candidates is reduced with ρ, thus fewer queries are applied on it and 2) the SG-tree is only efficient for high-selective signature containment; smaller values of ρ apply more queries with longer candidates, which are high-selective (many bits are on in them).

Next, we show the time spent by the three methods for different values of D, after fixing $k = 11$ and $\rho = 0.01$. In Table 6, the speedup of MMIS over MFS is maintained with the increase of problem size, showing that MMIS scales well. It also demonstrates that, for small ρ, the advantage of using the SG-tree is maintained.

Table 7 shows for $k = 11$, $\rho = 0.01$, and $D = 25,000$, the percentage of frequent candidates which are skipped from counting at the third phase of the mining process, (classified by length) as subsets of the frequent structures F_k found by the second phase. Clearly, the advantage of having computed the frequent k-sized structures, before applying

TABLE 7
Effectiveness of F_k

size. in candidates	7	8	9	10
% frequent candidates not counted	91	94.6	97.3	99.3

TABLE 8
Number of MIS

size of MIS	3	4	5	6	7	8	9	10	11
No. of MIS up to this size	107	209	463	605	678	725	752	766	771
No. of non-path MIS up to this size	63	165	419	561	634	681	708	722	727

exact mining, provides a large benefit also when values are considered in the MIS.

Table 8 presents the number of MIS and nonpath MIS discovered by size. The numbers unveil the applicability and usefulness of our approach. First, the total number of MIS is still small enough to be easily accommodated and indexed in memory. Second, the majority of MIS found are not simple paths, but more complex tree structures. This indicates that for queries containing these MIS, path indexes will perform poorly (since all paths contained in the MIS are frequent), while our MIS index will manage to locate fast the high-selective part of the queries.

6.3 Applicability of MIS to Query Processing

In this section, we demonstrate the query performance improvements achieved by our MIS-based index over a representative path index and two merge-join algorithms. We compare the effectiveness of our MIS index with a path index, specifically the A(k)-index [13]. We use the A(7)-index, which indexes exactly all paths in the DBLP database (the lengths of all paths in DBLP are less than 7). This means that for any simple *path* query, the A(7)-index can provide accurate answer to "how many documents contain it," and "which are they."

As a comparison measure, we count the number of documents that have to be accessed in order to validate the query, after the indexes have filtered the documents that may not possibly contain it. Filtering is performed in a different way by the indexes as already explained. The MIS index uses the inverted index directly to find the $document_id$s that contain the infrequent part of the query. The path index decomposes the query into paths and then finds the $document_id$s that contain all paths, by joining the occurrence lists of the paths. Notice that we measure the cost in terms of the accessed documents since 1) the query validation cost on the documents is similar for both methods and 2) DBLP documents have similar size, thus *which* documents are filtered is not important.

As we have already seen (Table 4), the majority of MIS are not simple paths. In the first experiment, we apply all nonpath MIS (ordered by selectivity) as queries on the DBLP data set and compare the MIS index with the A(k)-index. Fig. 16 shows the number documents from the DBLP data set that pass the checking, where $k = 11$, ρ is 0.01, and $D = 25,000$. Observe that if the MIS index is used, we need to validate several times fewer documents than when using the A(k)-index. Notice that there are several cases in this figure, where the A(k)-index needs to check thousands of the documents, while MIS points to only few documents.

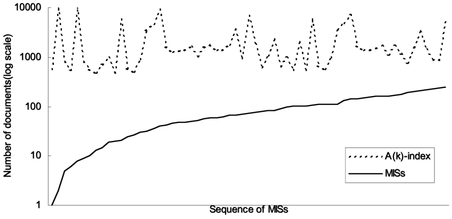


Fig. 16. Effectiveness of indexes (MIS queries).

We also evaluated queries which are not MIS themselves, but they are randomly generated superstructures of nonpath MIS without values. The performance gap between the two methods (see Fig. 17) decreases slightly in this case because the queries are actually superstructures of MIS, containing additional paths, that slightly increase the selectivity. This comes at a benefit of the A(k)-index which has more paths to merge. There are several valleys in the curve, which are caused by the presence of multiple MIS in a single query.

In general, our MIS index is much more effective than the A(k)-index in handling complex queries of high selectivity and also much faster because it is lightweight. An accurate A(k)-index has size comparable to the database, which affects its performance. On the other hand, in our experiments, the MIS index occupied less than 30Kb and the algorithm of Fig. 8 ran in just a few milliseconds, for each query. Finally, the construction cost of the MIS index is much lower than that of path indexes since only MIS are considered, and an efficient data mining process is applied for them, whereas the path indexes require many document traversals (or else many joins between the elements) in order to be built.

Next, we evaluate the effectiveness of our MIS index by comparing it with StackTree [15] and TwigStack [4], two merge-join algorithms that apply on XML data decompositions to relational tables. Again, we used the real data set of 25,000 documents from DBLP. All elements in these documents were extracted, encoded, and stored to relational tables which are managed by Oracle 8i Enterprise Edition release 8.1.5.[3]

For this experiment, we randomly generated six groups of tree structure queries of selectivity smaller than 0.01. Each group contains 10 queries. Group i contains queries having $i + 1$ edges, i.e., group 1 contains queries of two edges, group 2 queries of three edges, etc.

Table 9 shows the time due to I/O operations for the queries of each group on the average, by each of the three methods. StackTree and TwigStack have the same performance in all cases, which corresponds to the time to access all required records from the database sequentially. Our MIS-based index has significantly lower cost since it accesses and validates the queries on only those documents which contain the corresponding MISs. This set of documents corresponds to a small fraction of the database; at most, 250 documents are accessed by a query in the worst case. On the other hand, the other two methods retrieve tens of thousands or even more than one hundred of thousands of records corresponding to occurrences of low-selective

[3] In [15] and [4], all elements are stored in a single table, whereas we construct one table for each distinct element tag; this policy significantly reduces the computations on the server side.

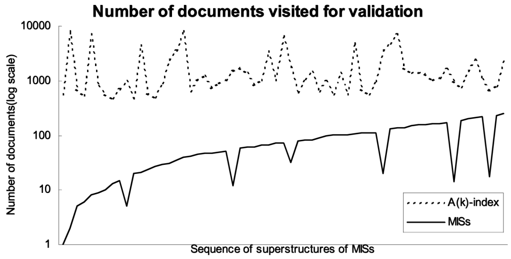


Fig. 17. Effectiveness of indexes (high-selective queries).

TABLE 10
MIS-Based Index versus the Relational Approach (CPU Time)

Time (in seconds)	StackTree	TwigStack	MIS index
Group 1	5	4.2	0.034
Group 2	5	4.2	0.044
Group 3	5.9	4.3	0.051
Group 4	6	4.7	0.057
Group 5	6.9	5	0.068
Group 6	20	9.8	0.085

element tags such as *author*, *title*, etc. Table 10 shows the computational time spent by the methods in query evaluation. Again, our MIS-based index is significantly faster compared to the two marge-join algorithms.

In summary, we have demonstrated the efficiency of our two-step methodology compared to exact query evaluation approaches on decomposed XML data to relational tables. These merge-join approaches suffer from the same problem as path indexes; they may need to access a large amount of data even for high-selective queries if these contain low-selective constructs.

Note that this study is not meant to devaluate indexing and query processing techniques for generic queries; both path indexes and merge-join algorithms can be used to evaluate exactly any XML query (low or high-selective). On the other hand, our index serves as an auxiliary, lightweight mechanism that operates on top of other storage schemes (and/or indexes) and facilitates fast evaluation of high-selective queries, no matter whether they comprise high-selective path components or not.

7 Conclusion and Future Work

In this paper, we introduced the concept of minimal infrequent structures (MIS), which are infrequent structures in XML data, whose substructures are all frequent. Indexing the MIS of a document collection comes at several benefits compared to using path indexes. First, the proposed method indexes not only paths, but also arbitrary useful structures of high selectivity. Second, it is space-efficient, requiring only a limited amount of storage. Third, since the MIS are infrequent, the index prunes large amounts of data fast. Fourth, due to its lightweight nature, it can be used independently, or in combination with other indexing and query processing techniques. Finally, not only does it provide a fast means of query evaluation, but it can also be used to spot queries of low selectivity.

In order to efficiently find all MIS, we developed a data mining algorithm, which can be several times faster than a previous Apriori-based approach. In addition, our algorithm is independent to the problem of indexing MIS since it can be seamlessly used for other data mining applications (e.g., discovery of frequent graphs).

TABLE 9
MIS-Based Index versus the Relational Approach (I/O Cost)

Time (in seconds)	StackTree	TwigStack	MIS index
Group 1	6.7	6.7	1.8
Group 2	6.8	6.8	1.8
Group 3	8.9	8.9	1.8
Group 4	9	9	1.8
Group 5	11	11	1.8
Group 6	19.5	19.5	1.8

In the current work, we have focused on the applicability of our techniques in databases that contain a large number of XML trees (i.e., documents). However, our methodology could be adapted for arbitrarily structured queries (e.g., graph-structured queries with wildcards or relative path expressions) by changing the definitions of the primary structural components (e.g., to consider relative path expressions like $a//b$ instead of plain edges), and the graph matching algorithms. Toward this direction, we plan to combine the encoding schema in [28] with our three-phase algorithm to extend the applicability of our approach.

The idea of extracting and indexing MIS can also be applied to databases of large XML documents, by defining the occurrences of a structure s in all documents as $sup(s)$, instead of counting the number of documents that contain s. In this case, we should redefine the signatures to be based not on the whole document, but on document parts up to a maximum size. We plan to investigate these adaptations in the future.

Another interesting direction for future work is the incremental maintenance of the set of MIS. A preliminary idea toward solving this problem is to change the mining algorithm of Fig. 3 to compute the exact counts of frequent structures of size k (instead of stopping as soon as the minimum support has been reached). Then, given a set ΔD of new XML documents, we apply the first and second phases of our algorithm for ΔD, to count the frequencies of all frequent structures of size k there. Having the exact count of frequent structures of size k in the existing and new document sets, we can then directly use the algorithm of [7] to compute the exact count of all frequent structures of size k in the updated document set $D + \Delta D$ and simply apply the third phase of our algorithm to update the MIS set.

Acknowledgments

This work has been partially supported by grants HKU 7380/02E and HKU 7149/03E from Hong Kong RGC.

References

[1] A. Aboulnaga, A. Alameldeen, and J. Naughton, "Estimating the Selectivity of XML Path Expressions for Internet Scale Applications," *Proc. Very Large Data Bases Conf.*, 2001.

[2] R. Agrawal and R. Srikant, "Fast Algorithms for Mining Association Rules, *Proc. Very Large Data Bases Conf.*, 1994.

[3] T. Asai, K. Abe, S. Kawasoe, H. Arimura, and H. Sakamoto, "Efficient Substructure Discovery from Large Semi-Structured Data," *Proc. Ann. SIAM Symp. Data Mining*, 2002.

[4] N. Bruno, N. Koudas, and D. Srivastava, "Holistic Twig Joins: Optimal XML Pattern Matching," *Proc. ACM SIGMOD Conf.,* 2002.

[5] Z. Chen, H. Jagadish, F. Korn, N. Koudas, S. Muthukrishnan, R. Ng, and D. Srivastava, "Counting Twig Matches in a Tree," *Proc. Int'l Conf. Data Eng.,* 2001.

[6] Q. Chen, A. Lim, and K.W. Ong, "D(k)-Index: An Adaptive Structural Summary for Graph-Structured Data," *Proc. ACM SIGMOD Conf.,* 2003.

[7] D.W. Cheung, J. Han, V. Ng, and C.Y. Wong, "Maintenance of Discovered Association Rules in Large Databases: An Incremental Updating Techniques," *Proc. Int'l Conf. Data Eng.,* 1996.

[8] C.W. Chung, J.K. Min, and K. Shim, "APEX: An Adaptive Path Index for XML Data," *Proc. ACM SIGMOD Conf.,* 2002.

[9] L. Dehaspe, H. Toivonen, and R.D. King, "Finding Frequent Substructures in Chemical Compounds," *Proc. Knowledge Discovery and Data Mining (KDD) Conf.,* 1998.

[10] R. Goldman and J. Widom, "Approximate DataGuides," *Proc. Workshop Query Processing for Semistructured Data and Non-Standard Data Formats,* 2000.

[11] A. Guttman, "R-Trees: A Dynamic Index Structure for Spatial Searching," *Proc. ACM SIGMOD Conf.,* 1984.

[12] H. Jiang, W. Wang, H. Lu, and J.X. Yu, "Holistic Twig Joins on Indexed XML Documents," *Proc. Very Large Data Bases Conf.,* 2003.

[13] R. Kaushik, P. Shenoy, P. Bohannon, and E. Gudes, "Exploiting Local Similarity for Indexing Paths in Graph-Structured Data," *Proc. Int'l Conf. Data Eng.,* 2002.

[14] R. Kaushik, P. Bohannon, J.F. Naughton, and H.F. Korth, "Covering Indexes for Branching Path Queries," *Proc. ACM SIGMOD Conf.,* 2002.

[15] S. Al-Khalifa, H.V. Jagadish, N. Koudas, J.M. Patel, D. Srivastava, and Y.Q. Wu, "Structural Joins: A Primitive for Efficient XML Query Pattern Matching," *Proc. Int'l Conf. Data Eng.,* 2002.

[16] M. Kuramochi and G. Karypis, "Frequent Subgraph Discovery," *Proc. Int'l Conf. Data Mining (ICDM),* 2001.

[17] L. Lim, M. Wang, S. Padmanabhan, J. Vitter, and R. Parr, "XPathLearner: An On-Line Self-Tuning Markov Histogram for XML Path Selectivity Estimation," *Proc. Very Large Data Bases Conf.,* 2002.

[18] N. Mamoulis, D.W. Cheung, and W. Lian, "Similarity Search in Sets and Categorical Data Using the Signature Tree," *Proc. Int'l Conf. Data Eng.,* 2003.

[19] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom, "Lore: A Database Management System for Semistructured Data," *SIGMOD Record,* vol. 26, no. 3, pp. 54-66, 1997.

[20] T. Milo and D. Suciu, "Index Structures for Path Expressions," *Proc. Int'l Conf. Database Theory,* 1999.

[21] N. Polyzotis and M. Garofalakis, "Statistical Synopses for Graph-Structured XML Databases," *Proc. ACM SIGMOD Conf.,* 2002.

[22] S.M. Selkow, "The Tree-to-Tree Editing Problem," *Information Processing Letters,* vol. 6, no. 6, pp. 184-186, 1977.

[23] J. Shanmugasundaram, K. Tufte, G. He, C. Zhang, D. DeWitt, and J. Naughton, "Relational Databases for Querying XML Documents: Limitations and Opportunities," *Proc. Very Large Data Bases Conf.,* 1999.

[24] T. Shimura, M. Yoshikawa, and S. Uemura, "Storage and Retrieval of XML Documents Using Object-Relational Databases," *Proc. Int'l Conf. Database and Expert Systems Applications (DEXA),* 1999.

[25] K. Wang and H. Liu, "Discovering Structural Association of Semistructured Data," *IEEE Trans. Knowledge and Data Eng.,* vol. 12, no. 3, pp. 353-371, 2000.

[26] H. Wang, S. Park, W. Fan, and P.S. Yu, "Vist: Virtual Suffix Tree for XML Indexing," *Proc. ACM SIGMOD Conf.,* 2003.

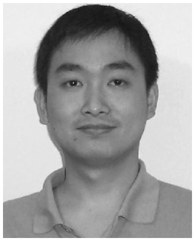
[27] L.H. Yang, M.L. Lee, and W. Hsu, "Efficient Mining of XML Query Patterns for Caching," *Proc. Very Large Data Bases Conf.,* 2003.

[28] M.J. Zaki, "Efficiently Mining Frequent Trees in a Forest," *Proc. SIGKDD Conf.,* 2002.

[29] C. Zhang, J. Naughton, D. Dewitt, Q. Luo, and G. Lohman, "On Supporting Containment Queries in Relational Database Management Systems," *Proc. ACM SIGMOD Conf.,* 2001.

[30] DBLP XML Records, http://www.acm.org/sigmod/dblp/db/index.html, Feb. 2001.

[31] International Press Telecommunications Council, News Industry Text Format (NITF), http://www.nift.org, 2000.



Wang Lian received the BEng degree in computer science from Wuhan University, Wuhan, China, in 1996, and the MPhil and PhD degrees in computer science from The University of Hong Kong in 2000 and 2004, respectively. He is currently an assistant professor at the Faculty of Information Technology, Macau University of Science and Technology. His research interests include semistructured data management and query processing, data mining, data warehousing, and information dissemination.

Nikos Mamoulis received a diploma in computer engineering and informatics in 1995 from the University of Patras, Greece, and the PhD degree in computer science in 2000 from the Hong Kong University of Science and Technology. Since September 2001, he has been an assistant professor in the Department of Computer Science, University of Hong Kong. In the past, he has worked as a research and development engineer at the Computer Technology Institute, Patras, Greece, and as a postdoctoral researcher at the Centrum voor Wiskunde en Informatica (CWI), the Netherlands. His research interests include spatial, spatio-temporal, multimedia, object-oriented, and semistructured databases, constraint satisfaction problems.

David Wai-lok Cheung received the BSc degree in mathematics from the Chinese University of Hong Kong and the MSc and PhD degrees in computer science from Simon Fraser University, Canada, in 1985 and 1989, respectively. From 1989 to 1993, he was a member of scientific staff at Bell Northern Research, Canada. Since 1994, he has been a faculty member of the Department of Computer Science and Information Systems at The University of Hong Kong. He is also the director of the Center for E-Commerce Infrastructure Development. His research interests include data mining, data warehouse, XML technology for e-commerce, and bioinformatics. Dr. Cheung is the program committee chairman of the Fifth Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD 2001). He is the program chairman of the Hong Kong International Computer Conference 2003. Dr. Cheung is a member of the ACM and the IEEE Computer Society.

S.M. Yiu received the BSc degree in computer science from the Chinese University of Hong Kong, the MS degree in computer and information science from Temple University, and the PhD degree in computer science from The University of Hong Kong. He is now a teaching consultant in the Department of Computer Science at the University of Hong Kong. His current research interests include computational biology, data mining, and computer security.