Mining Frequently Changing Substructures from Historical Unordered XML Documents

Q Zhao S S. Bhowmick M Mohania Y Kambayashi

Abstract

Recently, there is an increasing research efforts in XML data mining. These efforts largely assumed that XML documents are static. However, in many real applications, XML data are evolutionary in nature. In this paper, we focus on mining evolution patterns from historical XML documents. Specifically, we propose a novel approach to discover *frequently changing structures* (FCS) from a sequence of historical versions of unordered XML documents. The objective is to extract substructures that change *frequently* and *significantly* by analyzing structural evolution patterns of XML documents. We propose two algorithms based on a set of *evolution metrics* to extract FCS from the historical XML data. We also present a battery of optimization techniques to improve the space efficiency of our algorithms. Note that such structures cannot be extracted accurately and efficiently by repeatedly applying existing frequent substructure mining techniques on a sequence of snapshot data. FCS can be useful in several applications such as monitoring *interesting structures* in a specific domain, FCS-based classifier, indexing XML documents, and evolution-conscious XML query caching. Extensive experiments with both synthetic and real data show that the proposed algorithms are efficient and scalable and can discover FCS accurately.

Keywords: XML, evolutionary features, structural delta, evolution metrics, frequently changing structures, applications, data mining.

I. Introduction

XML has emerged as the leading textual language for representing and exchanging data over the Web. Due to staggering growth of XML data in different domains, mining XML data has increasingly become an interesting and important research problem in the data mining community [2], [13], [15], [28], [23], [26], [30], [17], [31], [19]. Existing works on mining XML data can

Preliminary version of this work was done prior to Kambayashi's sad demise on 6th, Feb, 2004 and was published in [33].

be broadly classified into three categories: *XML association rule mining* [4], [32], *frequent substructure mining* [2], [13], [15], [28], [23], [26], [30], and *XML classification/clustering* [19]. Among these, the *frequent substructure mining* is the most well researched topic. The basic idea is to extract substructures (subtrees or subgraphs), which occur frequently among a set of XML documents or within an individual XML document. For example, suppose there is a collection of XML documents that describe information about university professors. Figures 1(a) and (d) are the tree representations of two XML documents (partial view only). By applying existing state-of-the-art data mining techniques, frequent substructures among them can be discovered. For example, by applying the gSpan [28] mining approach, the structures shown in Figures 1(b) and (c) will be returned as frequent substructure mining results. These frequent substructures have been found useful in several applications such as efficient querying [29] and classification of XML documents [31].

A. Motivation

Our initial investigation revealed that majority of the existing approaches of XML mining focus only on snapshot XML data, while in real life XML data is dynamic in nature. That is, XML data may evolve at any time in different ways. For example, consider document 2 in Figure 1(d). The *publication* and *activity* of a professor may change over time. Figures 1(e), (f), (g) are the tree representations of three versions of document 2. The black and gray circles represent the newly inserted nodes (elements/attributes) and deleted nodes, respectively. The bold circles are nodes whose contents have been updated. It can be observed from the above example that there are primarily two types of changes to XML data: changes to *data content* (leaf nodes) and changes to the *structure* of XML data (internal nodes). Content changes occur when the data values of elements (attributes) are modified over time. Whereas structural changes to an XML document occur due to insertion/deletion of elements (attributes). *In this paper, we focus on the structural evolution of XML data only.* Note that there are many applications where structural representation of data is important, e.g. chemical compounds, biological data, computer network, and web browsing history [28]. Also, in archive-based applications such as the SIGMOD record and DBLP XML documents, content changes are rare compared to structural changes.

The evolutionary nature of structure of XML documents leads to two challenging problems in the context of data mining. The first one is to maintain the previously discovered knowledge. For

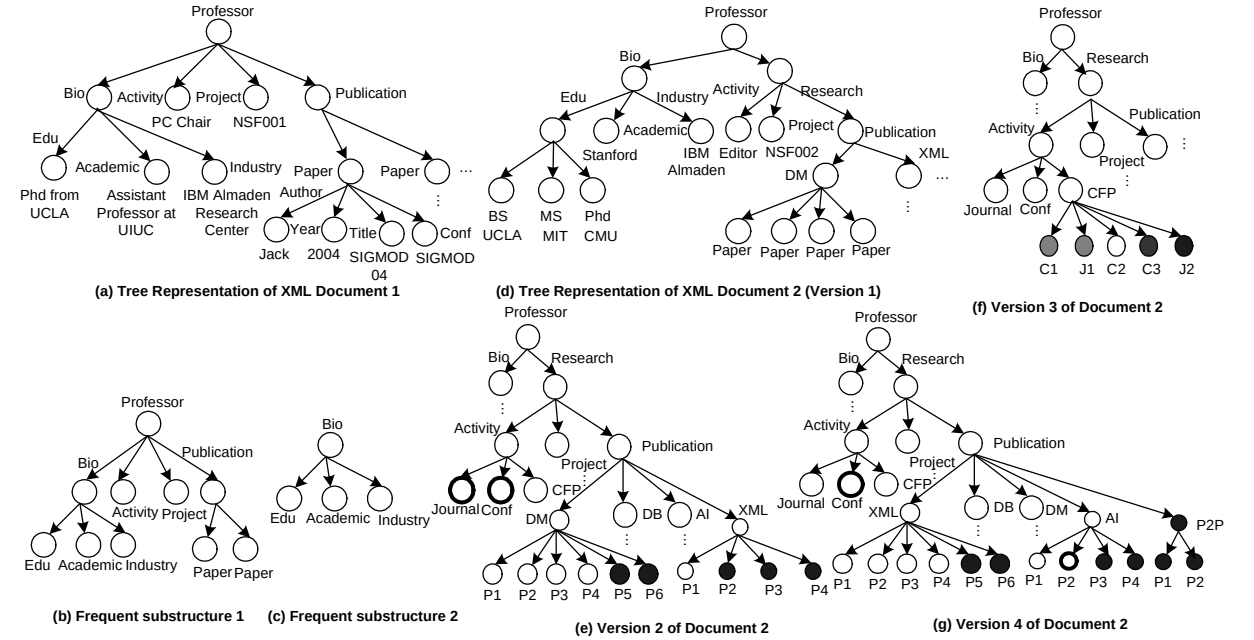


Fig. 1. An Example

instance, in frequent substructure mining, as the data source changes new frequent structures may emerge and some existing ones may not be frequent anymore. The second one is to discover novel knowledge by analyzing the evolutionary characteristics of historical XML data. Such knowledge is difficult or even impossible to discover from snapshot data efficiently due to the absence of evolution-related information. In this paper, we focus on the second issue. That is, *we present techniques to discover a specific type of novel knowledge by mining the evolutionary features of XML data*.

Let us elaborate informally on the types of novel knowledge one may discover by analyzing evolutionary features of XML data. Consider the different versions of XML documents in Figure 1. We may discover the following types of novel knowledge by exploiting the evolution-related information associated with the documents. Note that this list is by no means exhaustive.

- *Frequently Changing Structures (FCS):* Evolution of XML documents over time is generally *heterogeneous* in nature. That is, different parts of the XML documents may evolve in different ways over time. Some parts of the XML document may evolve more *frequently* than other parts. Some parts may change more *significantly* in the history compared to other parts that may only change slightly. We refer to structures that change frequently and significantly in the history as *frequently changing structures*. Here, *frequently* refers to the

large number of times the corresponding parts changed, while *significantly* refers to the large percentage of nodes that have changed in the corresponding subtree. For example, the structure rooted at *XML* changed more frequently, while the structure rooted at *Bio* never changed in the history.

- *Associative Evolutionary Structures:* Similar to the transactional association rule, different parts of the XML data may be associated in terms of their evolutionary features over time. For example, assume that whenever the structure rooted at *Publication* changes frequently and significantly, structure *Activity* also changes frequently and significantly. Then, an association rule $Publication \rightarrow Activity$ (we use the root node to represent a changed subtree) may be extracted with respect to some appropriately specified thresholds. We refer to such structures as *associative evolutionary structures*. Such structures can be useful in applications such as XML search engine, XML clustering, XML query caching, etc. The reader may refer to [5], [6] for further details.

Observe that the core foundation of associative evolutionary structures is also the notion of frequently changing structures. Hence, in this paper, we focus on discovering the frequently changing structures (FCS) from historical XML documents. As we shall see in Sections V and VI, FCS can be useful in several applications such as monitoring *interesting* structures in a specific domain that are important to the users, FCS-based classifier, XML indexing, and evolution-conscious XML query caching.

B. Why Existing Techniques Fail to Discover FCS Efficiently and Accurately?

At first glance it may seem that if we apply existing state-of-the-art XML mining techniques (such as gSpan [28]) repeatedly to a sequence of snapshots of XML data, then it may be possible to extract the frequently changing structures. However, this is not the case as such knowledge cannot be discovered accurately and efficiently by tweaking existing techniques on XML data sequence. Let us elaborate on this by using gSpan [28] algorithm as an example. Suppose there are n versions of XML documents denoted as X_1, X_2, $\cdots$, X_n. For each version, gSpan is applied and the sets of frequent structure mining results are denoted as M_1, M_2, $\cdots$, M_n. By postprocessing the sequence of mining results, we may find two sets of structures, denoted as I and J, where $I = M_1 \cap M_2 \cap \cdots \cap M_n$ is the set of structures that are frequent over all the time points from 1 to n; $J_{p,q} = M_p - M_q$ $(1 \leq q < p \leq n)$ is the set of structure that is frequent at

time point p but not frequent at time point q where $J_{p,q} \in J$. However, such structures may not reflect their evolution patterns and frequencies accurately. For example, structures in I may have changed or may not have changed. Also, it is possible that some of them may have been deleted from one position/document and inserted into another position/document. Similarly, structures in J may have changed or may not have changed. It may be the result of other changes that affect the computation of frequent structure such as such as changes to the total number of transactions, changes to other parts of the documents, etc. Moreover, such mining and postprocessing efforts are computationally expensive and as a result it will render FCS mining impractical. Hence, there is a need to develop novel techniques to discover FCS.

C. Overview of our Approach

Given a sequence of versions of an XML document, the goal of FCS mining is to discover all the substructures that change *frequently* and *significantly* in the history. Specifically, the *significance* and *frequency* of frequently changing structures (FCS) are defined and measured by a set of *evolution metrics*. These metrics are used to measure the structural evolutionary features of the XML documents. Based on such metrics, we propose two algorithms (based on top-down and bottom-up traversals of an XML tree) to extract the frequently changing structures by scanning the XML sequence only twice. Our proposed algorithms consist of two major phases: the *H-DOM construction* phase and the *FCS extraction* phase. In the first phase, given a sequence of historical XML documents, the *H-DOM (Historical Document Object Model) tree* is constructed to efficiently represent history of changes to XML data. The goal of second phase is to extract the frequently changing structures by traversing the H-DOM tree in top-down or bottom-up fashion. We also present a battery of optimization techniques to make the algorithm more scalable by reducing the size of the H-DOM tree under various conditions.

Our experiment results show that the proposed algorithms can successfully extract all the frequently changing structures efficiently. Also, the H-DOM tree is very compact, its size is around 50% of the original size of the XML sequence. Moreover, the proposed space optimization techniques can make the H-DOM tree more compact by around 30% and consequently make the algorithms more scalable.

D. Contributions

The major contributions of this paper can be summarized as follows.

- We introduce an approach that, to the best of our knowledge, is the first one to discover novel knowledge from the evolution pattern of historical XML documents. Specifically, in this paper we focus on discovering frequently changing structures (FCS).
- We propose a set of metrics to measure the evolutionary features of historical XML structures. Based on the *evolution metrics*, we present a set of algorithms and optimization techniques to discover FCS efficiently.
- We show with illustrative examples that FCS are useful for several real life applications. Specifically, we elaborate in detail on how FCS can be used as the framework for discovering evolutionary characteristics of *interesting FCS* that are of interest to a particular user group in a specific domain.
- We present the results of extensive experiments with both synthetic and real datasets that we have conducted to demonstrate the efficiency and scalability of the proposed algorithms and novelty of the mining results.

E. Paper Organization

The rest of this paper is organized as follows. In Section II, we introduce a model to represent the changes to historical XML documents and metrics used to detect FCS. In Section III, we present our proposed techniques of mining *frequently changing substructures*. Performances of the FCS mining algorithms are evaluated using synthetic and real datasets in Section IV. In Section V, we present some representative applications of FCS. In Section VI, we elaborate on a specific application of FCS. Section VII reviews the related works. Finally, the last section concludes this paper. A shorter version of this paper appeared in [33].

II. Representing Changes to Historical XML Documents

In this section, the problem of how to model the historical XML documents and measure their evolutionary features is discussed. We begin by discussing how an XML document is represented in our approach. Next, we present different types of structural changes that may occur in XML documents and how they are represented. Then, a set of *evolution metrics* are proposed to measure the structural evolutionary nature of the XML documents. Finally, our representation technique to concisely record the information related to the evolution history of XML documents is discussed.

A. Representation of an XML Document

The structure of an XML document can be modeled as a tree according to the Document Object Model (DOM) specification (hereafter called *XML trees*). XML trees can be classified into *ordered* trees, in which both the parent-child relationship and the left-to-right order among siblings are important, and *unordered* trees, in which the parent-child relationship is significant, while the left-to-right order among siblings is not important. In this paper, we focus on the unordered XML documents. An unordered model is more suitable for most database applications [25]. However, our technique can easily be extended to ordered XML as well.

An XML document is denoted as $T = (N, E, r)$, where N is the set of labeled nodes, E is the set of edges, $r \in N$ is the root. Note that we do not distinguish between elements and attributes, both of them are mapped to the set of labeled nodes. Each edge, $e = (x, y)$ is an ordered pair of nodes, where x is the parent of y in the XML tree. The *size* of the structure T, denoted by $|T|$, is the number of nodes in N.

Next, we introduce the notion of *induced subtree* of an unordered XML document. Given two rooted tree representations for two unordered XML documents T and T', T' is the *induced subtree* of T, denoted as $T' \preceq T$, if and only if: (1) $V' \subseteq V$ and $E' \subseteq E$; (2) the labeling of V' and E' is preserved in T'. In the rest of the paper, unless otherwise specified, whenever we refer to a subtree we refer to induced subtree.

B. Types of Structural Changes

Changes to an unordered XML document can be represented as five types of edit operations as follows [25]. The first three are basic operations and the last two are composite operations that can be represented as a list of basic operations.

- *Insert*(*x*(*name*, *value*), *y*): insert a node x, with node name $name$ and node value $value$, as a leaf child node of node y.
- *Delete*(*x*): delete a leaf node x.
- *Update*(*x*, *new_value*): change the value of a leaf node x to *new_value*. Note that only the value can be updated, but not its name.
- *Insert*(T_x,*y*): insert a subtree T_x, which is rooted at x, to node y.
- *Delete*(T_x): delete a subtree T_x, which is rooted at node x.

Based on the above edit operations, an *edit script* is defined as a sequence of edit operations that transform an XML document from one version to another [25]. However, not all the edit operations can change the structure of the XML documents. For example, the *Update* operation will not change the structure of a document. Hence, corresponding to the structural changes, we define the *structural edit script* as a sequence of *basic* edit operations that converts the structure of one version to the structure of another version. Note that it differs from the definition of edit script in two ways. First, a structural edit script does not include any update operation. Second, unlike edit script, it is composed of *basic* edit operations (insertion and deletion of a node). Note that an edit script may contain composite edit operations. To make it easier to locate the edit operation in the tree, an *affiliated node* is defined for each edit operation. For the insertion operation (*Insert*(*x*(*name*, *value*), *y*)), the *affiliated node* is y; for the deletion (*Delete*(*x*)) and update operations (*Update*(*x*, *new_value*)) the *affiliated node* is x.

Given two versions of an XML document, formally, the *structural delta* between them is defined as follows.

DEFINITION *2.1 (**Structural Delta**): Let T_i and T_{i+1} be the tree representations of two versions of an XML document, denoted as X_i and X_{i+1}. Let $t_i \preceq T_i$. The corresponding structure of t_i in T_{i+1} is t_{i+1}, denoted as $t_{i+1} \preceq T_{i+1}$. The **structural delta** for the subtree t_i from T_i to T_{i+1}, denoted as $\triangle_i(t)$, is defined as a structural edit script $\langle o_1, o_2, \cdots, o_m \rangle$ that transform the structure of t_i into t_{i+1}. That is, $\triangle_i(t) = \langle o_1, o_2, \cdots, o_m \rangle$ where o_k is a basic edit operation $\forall\ 0 < k \leq m$. The **size** of the structural delta $\triangle_i(t)$, denoted as $|\triangle_i(t)|$, is m. That is, $|\triangle_i(t)| = m$. Furthermore, the structural delta from X_i to X_{i+1} is denoted as $\triangle_i$.* □

Consider the previous examples in Figure 1. The structural delta from version 2 to version 3 is $\triangle_2$=⟨Delete(C_1), Delete(J_1), Insert(C_3(v_1), CFP), Insert(J_2(v_2), CFP)⟩ and the value of $|\triangle_2|$ is 4 since there are 4 basic edit operations shown as colored circles in Figure 1 (f). In the above definition, the XML structural delta is defined for two consecutive versions of an XML document. To represent the sequence of changes to more than two versions of an XML document, we define the notion of *XML structural delta sequence*.

DEFINITION *2.2 (**Structural Delta Sequence**): Let $\langle T_1, T_2, \cdots, T_n \rangle$ be the sequence of tree representations of n historical versions of an XML document X. Let $t \preceq T_1$. The **structural delta sequence** for the subtree t from T_1 to T_n is $\Psi_t = \langle \triangle_1(t), \triangle_2(t), \cdots, \triangle_{n-1}(t) \rangle$, where $\triangle_i(t)$ is the XML structural delta for t from ith version to (i+1)th version. Also, Ψ_t is contained in Ψ_X,*

denoted as $\Psi_t \vdash \Psi_X$*, where* Ψ_X *is the structural delta sequence of* X *and* $\Psi_X = \langle \triangle_1, \triangle_2, \cdots, \triangle_{n-1} \rangle$. □

Reconsider the examples in Figure 1. For the substructure rooted at node *Activity* (denoted as t_{act}), the corresponding of structural delta is $\langle\ \triangle_1(t_{act}), \triangle_2(t_{act}, \triangle_3(t_{act})\ \rangle$ where $\triangle_1(t_{act})$ and $\triangle_3(t_{act})$ are empty, $\triangle_2(t_{act})$ is the same as in the structural delta example.

C. Evolution Metrics

From the example in Figure 1, we observed that different substructures of the XML document might change in different ways at different frequencies. Hence, in order to extract frequently changing structures, it is important to define metric(s) that can quantify the evolutionary characteristics of a specific XML document in history. Intuitively, the lower the degree of evolution of a subtree in XML document, the less frequently and significantly the subtree changes in the history. In this section, we introduce a set of *evolution metrics* to measure this. Specifically, three evolution metrics, namely *structure dynamic*, *version dynamic*, and *degree of dynamic*, will be discussed. We begin by defining the notion of *consolidate structure* which shall be used subsequently.

DEFINITION *2.3:* **Consolidate Structure:** *Given two structures* $t_i = (N_{t_i}, E_{t_i}, r_{t_i})$ *and* $t_j = (N_{t_j}, E_{t_j}, r_{t_j})$ *where* $r_{t_i} = r_{t_j}$*, the* ***consolidate structure*** *of* t_i *and* t_j*, denoted as* $t_i \uplus t_j$*, where* $t_i \uplus t_j = (N_{t_i \uplus t_j}, E_{t_i \uplus t_j}, r_{t_i})$*,* $N_{t_i \uplus t_j} = N_{t_i} \cup N_{t_j}$ *and* $e = (x, y) \in E_{t_i \uplus t_j}$*, if and only if* x *is the parent of* y *in* E_{t_i} *or* E_{t_j}. □

Consider the structures in Figure 1. For the substructures rooted at node *Bio* in Figure 1(a) and (d), the consolidate structure is the structure rooted at node *Bio* in Figure 1(d). Next, we define the *structure dynamic* metric.

DEFINITION *2.4 (**Structure Dynamic**): Let* T_i *and* T_{i+1} *be the tree representations of two versions of XML documents. Suppose* $t \preceq T_i$*. The* ***structure dynamic*** *of* t *from* T_i *to* T_{i+1}*, denoted as* $N_i(t)$*, is defined as:*

$$N_i(t) \quad = \quad \frac{|\triangle_i(t)|}{|t_i \uplus t_{i+1}|}$$

□

Here $N_i(t)$ is the structural dynamic of t from version i to $i+1$. By using the consolidate structure, the total number of unique nodes in the two versions can be obtained as $|t_i \uplus t_{i+1}|$. It includes not only nodes that are in version $i+1$ but also nodes that have been deleted in version

i. $N_i(t)$ is the percentage of nodes that have changed from X_i to X_{i+1} in t against the total number of nodes in its consolidate structure. For example, consider the two structures shown in Figures 1(d) and 1(e). We calculate the structure dynamic value for the substructure rooted at node *DM* from version 1 to version 2. Based on the definition, $|\triangle_{DM_1}| = 2$, $|DM_1 \uplus DM_2| = 6$. Consequently, $N_1(DM) = 0.33\ (2/6)$. It also can be observed that $N_i(t) \in [0, 1]$. If t is inserted or deleted, then the corresponding value of structure dynamic is 1 since $\triangle_{t_i} = t_i \uplus t_{i+1} = t$. If t did not change from version i to version $i + 1$, then the value of structure dynamic is 0 since $|\triangle_{t_i}|$ is 0. Note that larger the value of structure dynamic of a substructure, more significantly it changed. Next, we introduce the notion of *version dynamic*.

DEFINITION *2.5 (**Version Dynamic**): Let $\langle\ T_1, T_2, \cdots, T_n\ \rangle$ be the sequence of n versions of an XML document. Suppose $t \preceq T_j$ ($1 \leq j \leq n$). The **version dynamic** of t, denoted as $V(t)$, is defined as:*

$$V(t) = \frac{\sum_{i=1}^{n-1} v_i}{n-1} \quad where \quad v_i = \begin{cases} 1, & \textit{if } |\triangle_i(t)| \neq 0; \\ 0, & \textit{if } |\triangle_i(t)| = 0; \end{cases}$$

□

Consider the 4 different versions of the XML document in Figure 1. We calculate the version dynamic value for the substructure rooted at node *XML*. The n value here is 4. For the first delta, $|\triangle_{XML_1}|$ is not 0, so $v_1 = 1$. Similarly, $v_2 = 0$, $v_3 = 1$. Then, $\sum_{i=1}^{3} v_i = 2$. Consequently, the version dynamic of this substructure is $0.67\ (2/3)$. It can be observed that $V(t) \in [0, 1]$. If t changed in every version in the history, then the corresponding value of $\sum_{i=1}^{n} v_i$ is $n-1$, so the version dynamic value is 1. If t did not change in the history at all, then the value of $\sum_{i=1}^{n} v_i$ is 0 and version dynamic value is 0. Also, it implies that larger the value of version dynamic is, more frequently the substructure changed in the history.

Note that a substructure has one value for version dynamic and a sequence of values for structure dynamic. Hence, we proposed another evolution metric called *degree of dynamic*, denoted as DoD, to represent the overall significance of the structural changes in the history. DoD is an extension of structure dynamic by incorporating the version dynamic metric.

DEFINITION *2.6 (**Degree of Dynamic**): Let $\langle T_1, T_2, \cdots,\ T_n\ \rangle$ be the sequence of tree representations of n historical versions of an XML document. Suppose $t \preceq T_j$ ($1 \leq j \leq n$). The **degree of dynamic**, DoD, for t is defined as:*

$$DoD(t,\alpha) = \frac{\sum_{i=1}^{n-1} d_i}{(n-1) * V(t)} \; where \; d_i = \begin{cases} 1, if \; N_i(t) \geq \alpha \\ 0, if \; N_i(t) < \alpha \end{cases}$$

where α is the user-defined threshold for structure dynamic. □

The metric DoD is defined based on the threshold of structure dynamic. It represents the fraction of versions, where the structure dynamic values for the substructure are no less than the predefined threshold α, against the total number of versions in history where the substructure has changed. Consider the examples shown in Figure 1. We can calculate the DoD value for the substructure rooted at node *XML*. From the previous examples, we know that the structure dynamic values of this substructure are $0.75, 0,$ and 0.33. The version dynamic value is 0.67. Suppose the threshold for structure dynamic is set to 0.30, then the value of DoD is 1 (2/2). If the threshold for structure dynamic is set to 0.35, then the corresponding DoD value will be 0.5 (1/2). It is obvious that, $\forall \; \alpha, DoD(t,\alpha) \in [0,1]$. Similar to the structure dynamic, the value of DoD also implies the overall significance of the evolution of the substructure. The larger the value is, more significant are the changes.

D. Representation of XML Document History

As our goal is to discover FCS from a sequence of versions of historical XML documents during a specific time period, it is important to capture the historical information of structural evolution of XML documents concisely to facilitate efficient computation of evolution metrics. As we shall see in the next section, the values of the evolution metrics are used as the foundation for detecting FCS.

Since, the structure of an XML document can be represented as a tree, naively, we can represent the history of XML documents as a sequence of trees. However, this approach is not efficient as there are often substantial overlaps among the different versions of XML trees. Furthermore, in order to compute the evolution metrics we will need to navigate the sequence of trees which is computationally expensive. Hence, we present a concise structure called *H-DOM tree*, to represent the history of evolution of XML data. Intuitively, the H-DOM tree is an extension of the DOM tree with some historical properties so that it can compress the history of changes to XML documents into a single tree. Formally, we define an *H-DOM tree* as follows.

DEFINITION *2.7:* **H-DOM:** *An H-DOM tree is a 4-tuple $H = (N, A, v, r)$, where N is a set of object identifiers; A is a set of labeled, directed arcs (p, l, c) where $p, c \in N$ and l is a*

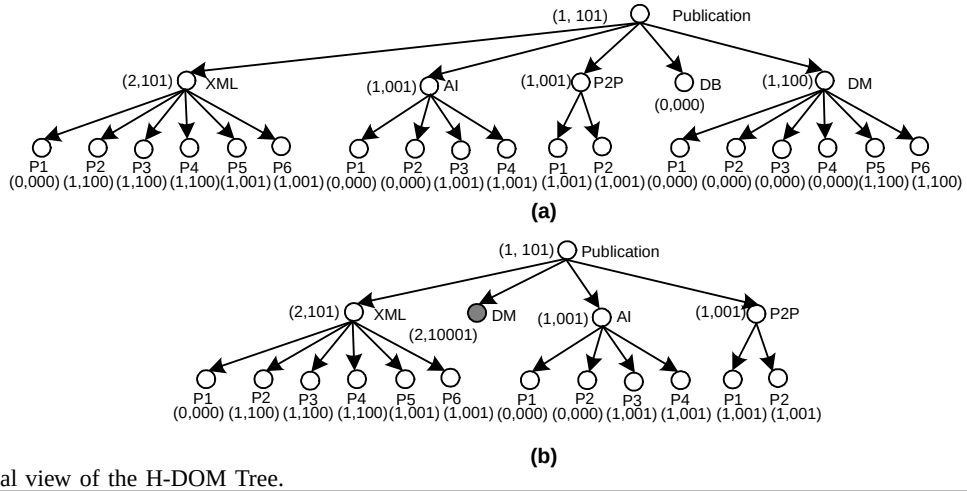


Fig. 2. Partial view of the H-DOM Tree.

string; v *is a function that maps each node* $n \in N$ *to a set of values* (C_n, C_v)*,* C_n *is an integer and* C_v *is a binary string;* r *is a distinguished node in* N *called the root.* □

We now elaborate on the parameters C_n and C_v. The two parameters are introduced to record the evolutionary features of each substructure. C_n is an integer that records the number of versions that a substructure has changed significantly enough (the structure dynamic value is no less than the corresponding threshold). C_v is a binary string that represents the historical changes of a substructure. The length of the string is equal to the number of deltas in the XML sequence. The ith digit of the string denotes the change status of the structure from X_i to X_{i+1}, where the value of *1* means that the particular structure has changed and the value of *0* indicates otherwise. In the H-DOM tree, the C_v value for each structure is lastly updated by using the formula: $C_v(t) = C_v(t_1) \vee C_v(t_2) \vee \cdots \vee C_v(t_j)$, where $t_1, t_2, \cdots, t_j$ are the substructures of t.

For example, Figure 2(a) is part of the H-DOM for the structure sequence in Figure 1. Suppose the threshold for structure dynamic is 0.30, the C_n value of node *XML* is 2, which means that this structure has changed twice in the history with a structure dynamic value no less than 0.30. The C_v value 100 of node p_3 means that this node has changed from X_1 to X_2. The C_v value of the internal nodes and root node are calculated according to the above formula. Using the C_v and C_n values, the set of evolution metrics can be calculated as follows.

- $N_i(t) = \frac{\sum C_v(d_j)[i]}{|t_i \uplus t_{i+1}|}$, where d_j is the list of descendant nodes of t, $C_v(d_j)[i]$ is the ith digit of $C_v(d_j)$.
- $V(t) = \frac{\sum_{i=1}^{n-1} C_v[i]}{n-1}$, where $C_v[i]$ is the ith digit of $C_v(t)$; n is the total number of XML documents.

- $DoD(t) = \frac{C_n}{\sum_{i=1}^{n-1} C_v[i]}$, where $C_v[i]$ is the ith digit of $C_v(t)$; n is the total number of XML documents.

It is worth mentioning that the H-DOM tree structure is inspired by the FP-Tree data structure used in association rule mining [12]. It is designed to preserve and compress the historical structural information of XML versions. The H-DOM tree compresses historical structural data by representing identical nodes only once in the H-DOM tree, while the relevant historical information is preserved using a binary string and an integer. Compared to the FP-Tree, the compactness of H-DOM should be higher since the same nodes may appear more than once with *h-links* in the FP-tree. Moreover, the FCS can be extracted without any candidate generation process by traversing the H-DOM exactly once, while in FP-Tree there is a conditional FP-Tree generation process. Additionally, the H-DOM tree stores the temporal features of the XML structures.

III. FCS MINING

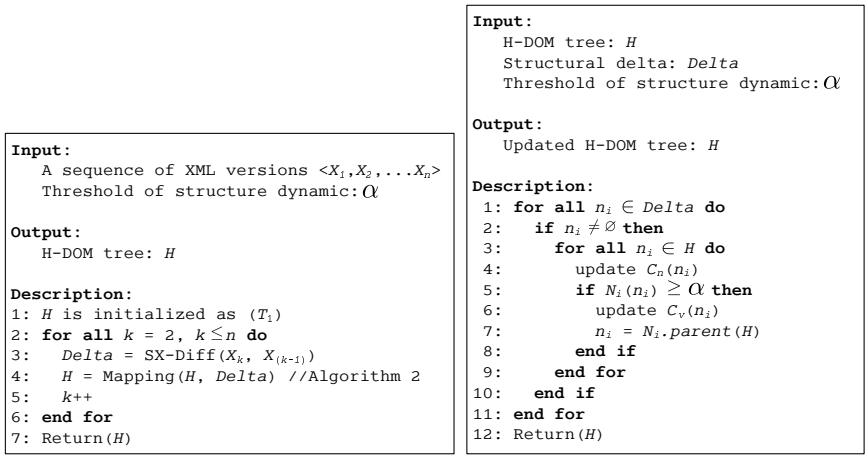
In this section, we present the algorithm for discovering frequently changing structures (FCS). We begin by formally defining FCS.

A. Frequently Changing Structures

The problem of frequently changing structure mining is to discover those structures that changed significantly and frequently in the history. Based on the set of evolution metrics discussed in the preceding section, the frequently changing structure is defined as follows.

DEFINITION *3.1 (**Frequently Changing Structure (FCS)**): Let $\langle T_1, T_2, \cdots, T_n \rangle$ be the tree representations for versions of an XML document X. Let the thresholds of structure dynamic, version dynamic, and degree of dynamic be α, β, γ respectively. A structure $t \preceq T_j$ ($1 \leq j \leq n$) is a **frequently changing structure** in this sequence iff: $\Psi_t \vdash \Psi_X$, $V(t) \geq \beta$, and $DoD(t, \alpha) \geq \gamma$.* □

The FCS is defined based on the predefined thresholds of the evolution metrics. The significance of changes is defined by the structure dynamic and degree of dynamic thresholds, while the frequency of changes is defined by version dynamic threshold. For example, an example of the frequently changing structure in Figure 1 will be the structure rooted at node *XML* as shown in Figure 1(d). This structure may indicate that the corresponding professor is very active in the research area of *XML*.



```
Input:
   A sequence of XML versions <X1,X2,...Xn>
   Threshold of structure dynamic: α

Output:
   H-DOM tree: H

Description:
1: H is initialized as (T1)
2: for all k = 2, k ≤ n do
3:    Delta = SX-Diff(Xk, X(k-1))
4:    H = Mapping(H, Delta) //Algorithm 2
5:    k++
6: end for
7: Return(H)
```

(a) Algorithm 1: H-DOM Construction

```
Input:
   H-DOM tree: H
   Structural delta: Delta
   Threshold of structure dynamic: α

Output:
   Updated H-DOM tree: H

Description:
 1: for all ni ∈ Delta do
 2:    if ni ≠ ∅ then
 3:       for all ni ∈ H do
 4:          update Cn(ni)
 5:          if Ni(ni) ≥ α then
 6:             update Cv(ni)
 7:             ni = Ni.parent(H)
 8:          end if
 9:       end for
10:    end if
11: end for
12: Return(H)
```

(b) Algorithm 2: Mapping

Fig. 3. FCS Mining algorithms.

B. FCS Mining Algorithms

We now present the algorithms for discovering FCS. Given a sequence of XML document versions, the FCS mining algorithm consists of two main phases: the *H-DOM tree construction* phase and the *FCS extraction* phase.

The H-DOM Tree Construction Phase: Figure 3(a) describes the process of H-DOM tree construction. Given a sequence of historical XML documents, the H-DOM tree is initialized as the structure of the first version. After that, the algorithm iterates over all the other versions by extracting the structural deltas and mapping them into the H-DOM tree. Given two versions of an XML document, the SX-Diff function is a modification of the X-Diff [25] algorithm that generates only the structural changes. The structural delta is mapped into the H-DOM tree according to mapping rules as described in Figure 3(b). This process iterates until no more XML document is left in the sequence. Finally, the H-DOM tree is returned as the output of this phase.

Figure 3(b) describes the mapping function. Given the H-DOM tree and the structural changes, the objective of this function is to map the deltas into the H-DOM tree and return the updated H-DOM tree. The idea is to update the corresponding values of the nodes in the H-DOM tree for all the nodes in the structural delta. The values of the nodes are updated according to following rules:

1) If the node does not exist in the H-DOM tree, then the node is inserted. The value of C_v is set to $000\cdots1$ where the ith digit of the string is set to 1 and i is the version number

of the structural delta. In addition, the value of N_i is calculated. If $N_i \geq \alpha$, then C_n is set to 1 and the C_n values of its parent nodes are incremented by 1 if N_i is no less than α. Otherwise, C_n is set to 0 and the process terminates.

2) For nodes that exist in the H-DOM, the value of C_v is updated by inserting a 1 at the ith digit of C_v where i is the version number of the structural delta. The value of C_n is also updated based on N_i and α. Similarly, If $N_i \geq \alpha$, then C_n is incremented by 1 and the C_n values of its parent nodes are updated based on the same rule until N_i is less than α. Otherwise, C_n does not change and the process terminates.

The FCS Extraction Phase: The objective of this phase is to extract FCS from the H-DOM tree representation. Specifically, given the H-DOM tree, the values of structure dynamic, version dynamic, and degree of dynamic for each node are calculated and compared against the predefined thresholds. Since for a FCS, both its version dynamic and degree of dynamic should be no less than the thresholds, we first calculate only one of the parameters and determine whether it is necessary to calculate the other parameter. This is because if any of the two parameters does not satisfy the definition, then the substructure is not a FCS. In our algorithm, the version dynamic for a node is checked against the corresponding threshold first. If it is no less than the threshold, then we check its degree of dynamic. Based on the traversal strategy of the H-DOM tree, two variants of the algorithm can be designed for FCS extraction: the *bottom-up* (level by level) approach and the *top-down* (breath first) approach. Before we discuss these two strategies in detail, we present two lemmas that will be used to make the extraction phase more efficient.

LEMMA *3.1: Let $n_i, n_j \in N$ be any two nodes. The substructures rooted at n_i and n_j are denoted as t_{n_i} and t_{n_j} respectively. If n_i is the ancestor of n_j, then $V(t_{n_i}) \geq V(t_{n_j})$.*

PROOF *3.1:* The proof is intuitive. Based on the previous definition, once a node changes, superstructures that include this node are considered as changed. It indicates that the number of versions a superstructure has changed should be no less than its substructures. Consequently, it can be concluded that the version dynamic of a superstructure should be no less than the version dynamic of its substructures, while the total number of versions is the same. ■

LEMMA *3.2: Let t_1 and t_2 be any two structures and $t_2 \preceq t_1$. Given the threshold for DoD as γ, the necessary condition for structure t_1 to be a FCS is that $C_n(t_1) \geq \gamma \times V(t_2) \times (n-1)$.*

PROOF *3.2:* From Lemma 3.1, we can infer that $V(t_1) \geq V(t_2)$. The necessary condition for

```
Input:
   H-DOM tree: H
   Threshold of version dynamic: β
   Threshold of degree of dynamic: γ

Output:
   A set of FCS rooted nodes: F

Description:
 1: for all n_i ≠ ∅ do //Top-down BFS
 2:   if V(n_i) ≥ β then
 3:     if DoD(n_i) ≥ γ then
 4:       F is updated by incorporating n_i
 5:     end if
 6:   else
 7:     prune all descendants of n_i
 8:   end if
 9: end for
10: Return(F)
```

(a) Algorithm 3: FCS Mining (top-down)

```
Input:
   H-DOM tree: H
   Threshold of version dynamic: β
   Threshold of degree of dynamic: γ

Output:
   A set of FCS rooted nodes: F

Description:
 1: for all n_i ≠ ∅ do //bottom-up BFS
 2:   if C_n < γ × V(t_ni) then
 3:     n_i = n_i.next
 4:   else
 5:     if V(n_i) ≥ β then
 6:       if DoD(n_i) ≥ γ then
 7:         F is updated by incorporating n_i
 8:       end if
 9:     end if
10:   end if
11: end for
12: Return(F)
```

(b) Algorithm 4: FCS Mining (bottom-up)

Fig. 4. FCS Mining algorithms (FCS extraction phase).

structure t_1 to be a FCS is that its degree of dynamic is no less than the threshold γ, which is $\gamma \leq \frac{C_n(t_1)}{V(t_1)\times(n-1)}$. Then, $C_n(t_1) \geq \gamma \times V(t_1) \times (n-1)$, while $V(t_1) \geq V(t_2)$, it can be inferred that $C_n(t_1) \geq \gamma \times V(t_2) \times (n-1)$. ■

Based on the above lemmas, we observed that it is not necessary to traverse the entire H-DOM tree. We can skip checking some structures that cannot be FCS. Lemma 3.1 can be used in the top-down traversal strategy. When we reach a node where its version dynamic is less than the threshold, it is not necessary to further traverse down this substructure since the version dynamic of its substructures will definitely be less than the threshold and hence these substructure cannot be FCS. Lemma 3.2 can be used in the bottom-up traversal strategy. In this case, for any node, rather than calculate its version dynamic value, the C_n value of the node is checked against the value of $\gamma \times V(t_i)$, where t_i is any of its substructures. If $C_n < \gamma \times V(t_i)$, then it is not necessary to calculate the version dynamic and degree of dynamic for this structure since it cannot be a FCS. Based on the above lemmas, the top-down and the bottom-up FCS mining algorithms are presented in Figures 4(a) and 4(b).

Algorithm Analysis: We now analyze the time and space complexities of the FCS mining algorithms. In Phase 1, the H-DOM tree is constructed based on the sequence of historical XML documents. In this phase, each XML document is parsed once and only consecutive versions are compared. Let $\langle\ T_1, T_2, \cdots, T_n\ \rangle$ denote the sequence of XML documents and $|T_i|$ denotes the number of nodes in the *i*th document. The complexity of SX-Diff is $O(|T_i| \times |T_{i+1}|) \times max\{\text{deg}(T_i), \text{deg}(T_{i+1})\} \times \log_2(\max\{\text{deg}(T_i), \text{deg}(T_{i+1})\}))$ [25]. The complexity of

the mapping process is $O(|t_i|)$. The SX-Diff and mapping process iterate $k-2$ times in this phase, while the cost of the initialization is $O(|T_1|)$. Since $|t_i| \leq |T_i|$, the dominant of this iteration is the SX-Diff. Hence, the overall complexity of Phase 1 is $O((k-2)\times$ max$\{$ $|T_i| \times |T_{i+1}|)\times$ max$\{$deg(T_i), deg(T_{i+1})$\}$ $\times \log_2$(max$\{$deg(T_i), deg(T_{i+1})$\}$)$\}$), where $i \in [2, k-1]$. In Phase 2, the H-DOM tree is traversed and the parameters for all the potential FCS are calculated and compared against the predefined thresholds. No matter which traversal strategy we choose, the upper bound of this phase is $O(|T|)$, which is the cost of traversing the H-DOM tree, where $|T|$ is the total number of nodes in the H-DOM tree. In practice, the actual cost of this phase is lesser than this as we use Lemmas 3.1 and 3.2 to reduce the traversal space.

In the FCS mining algorithms, the H-DOM tree is processed in memory. The space cost of this algorithm is the size of the H-DOM tree. Based on the algorithm, we observed that the size of the H-DOM tree depends on the amount of overlaps between the consecutive versions. For the same number of XML documents with the same average number of nodes, the more significantly they change, larger is the size of the H-DOM tree. Since only the structural data are stored and each unique node is stored only once, the size of the H-DOM should be no larger than the total size of the sequence of XML documents. However, as the sizes of the XML documents increase or the changes become more significant, or the number of XML documents increases, the size of H-DOM tree will increase accordingly. However, the upper bound of the space requirement is $O(|T_1 \uplus T_2 \uplus \cdots \uplus T_n|)$.

C. Optimization Techniques

We now propose three optimization techniques, the *compression techniques*, the *build and merge strategy*, and the *DTD-based pruning technique*, to make the proposed algorithm more scalable by reducing the size of the H-DOM tree.

Compression Technique: Suppose there are n versions of XML in the sequence. Then, for each node a length n binary string is used to represent the history of changes in the H-DOM tree. Observe that the size of the string can be very large. However, only 2 out of n digits are useful since each node itself in the H-DOM tree could change at most twice (*insertion* and *deletion*). Consequently, rather than using the binary string, we use two integers to represent the changes. Consider the H-DOM tree in Figure 2 as an example. Suppose the node p_2 is deleted in the i+1th version. Then the C_v value of this node will be $1000\cdots01$. Now suppose we only store two

```
Input:
   H-DOM tree: H
   Structural delta: Delta
   Threshold of structure dynamic: α

Output:
   Updated H-DOM tree: H

Description:
1: for all n_i ∈ Delta do
2:   if n_i is deleted according to the DTD and
       V(n_i) < α then
3:     update(n_i,H)
4:     prune all descendants of n_i
5:   else
6:     update(n_i,H)
7:   end if
8: end for
9: Return(H)
```

(a) Algorithm 5: Building and Merging

```
Input:
   H-DOM tree: H
   Threshold of version dynamic: β
   Threshold of degree of dynamic: γ
   User specified concept hierarchy: T

Output:
   A set of nodes where i-FCS rooted : F

Description:
 1: for all n_j=Bottom-upTrav(H)≠null do
 2:   while T_i=Bottom-upTrav(T)≠null
 3:   if S(n_j) ≃ S_i then
 4:     for all s ⊴ S(n_j) do
 5:       if C_n < γ × V(t), {n_j = n_j.next, break}
 6:       if V(n_j) ≥ β & DoD(n_j) ≥ γ, {F = F ∪ n_j}
 7:     end for
 8:     break; end if
 9: end for
10: Return(F)
```

(b) Algorithm 6: i-FCS extraction

Fig. 5. Optimization technique and i-FCS extraction algorithm.

integers 1 and i to represent the changes. Using the proposed algorithm the space requirement is i bits, but using this strategy it only requires 16 bits (for two integers). It is obvious that when $i > 16$, the later strategy is more space efficient. Usually, to get useful knowledge from the changes, the number of versions is greater than eight.

Building and Merging Strategy: We observed that for any structure that has been deleted their C_n and C_v values would not change since no change could happen to them again. Thus, whether this deleted structure is a FCS or not can be determined by then. Hence, in this strategy we propose not to keep all substructures in the H-DOM tree. If the structures are not a FCS when they are deleted, only the root nodes are stored in the H-DOM, with the summarized historical information. By using this strategy, the size of the H-DOM tree can be reduced. For example, consider the H-DOM tree in Figure 2. Suppose in the next version the substructures rooted at *DM* and *DB* are deleted. Assume that the substructure *DM* is a FCS while the substructure *DB* is not. Then, rather than storing the entire substructure of *DM* and *DB*, only the root node of *DM* is stored along with summarized historical information. Similarly, the substructure *DB* is merged into its parent node as shown in Figure 2(b). The algorithm based on this strategy is shown in Figure 5(a).

DTD-based Pruning Technique: Our investigation of the history of structural changes to real life XML documents revealed that often some of the nodes in the XML tree never change in the history. Thus, it is not necessary to store such information since it does not play significant role in FCS mining. If we can prune such nodes during the H-DOM construction phase, then

the H-DOM tree will be more compact and the efforts of checking such nodes can be avoided. However, the challenge is to identify such nodes efficiently. If the XML documents do not have a DTD or XML schema then this is an expensive process as we have to analyze the documents in the history to identify such nodes. On the other hand, if the XML documents are valid, then we can extract information related to nodes that never change by analyzing the corresponding DTD or XML schema. Specifically, we can categorize the elements and attributes in the XML documents into two classes based on the DTD. Elements and attributes that can be inserted or deleted belong to class 1, while elements and attributes that cannot be inserted or deleted are in class 2. Using the DTD or XML schema, we can identify elements that can occur more than once or are optional. The nodes representing these elements in the XML documents can be inserted or deleted in the history. Similarly, instances of elements that are defined as "default" and "required" and can occur exactly once cannot be deleted or inserted. Note that this approach does not guarantee identification of *complete* set of nodes that do not change. This is because, although some of the nodes may not evolve in history, they can only be identified by analyzing the actual documents rather than the DTD.

Our DTD-based pruning strategy maps only nodes that belong to class 1, while nodes in class 2 are merged with their parent nodes to save space. For example, suppose elements *BS, MS* and *PhD* are defined as required subelements with exactly one occurrence for element *Edu* in Figure 1(d). Then, rather than store the entire substructure rooted at node *Edu*, it can be represented as a single node *Edu* in the H-DOM tree.

IV. Performance Evaluation of FCS Mining

In this section, we evaluate the FCS mining algorithms with extensive experiments.

A. Experimental Setup and Dataset

We have implemented FCS mining algorithms entirely in Java. We ran experiments on a PC with Intel Pentium 4, 1.7GHz CPU, 256 RAM, 40G hard disk, and Microsoft Windows 2000. For the FCS algorithm, we have implemented two "optimization-unconscious" algorithms, the bottom-up based algorithm (FCS-BASIC-B) and the top-down based algorithm (FCS-BASIC-T). We also create variants of these algorithms by incorporating the proposed optimization techniques. Specifically, FCS-A denotes the algorithm that integrates all the three optimization

Symbol	Description
NoN	Numbers of nodes
NoV	Numbers of versions
PoC	% of changes

(a) Statistic of Datasets

Symbol	Description
α	Threshold of Ni(t)
β	Threshold of V(t)
γ	Threshold of DoD(t, α)

(b) Parameters

Source Data	NoN	NoV	PoC	α	β	γ
SIGMOD1	1124	20	10%	-	0.2	0.4
DBLP1	1143	20	10%	0.2	-	0.4
Synthetic1	1264	20	10%	0.2	0.2	-

(c) Description of Datasets

Source Data	NoN	NoV	PoC	α	β	γ
SIGMOD2	-	30	10%	0.2	0.2	0.4
DBLP2	5743	-	10%	0.2	0.2	0.4
Synthetic2	1264	20	-	0.2	0.2	0.4

(d) Description of Datasets

Data set	NoN	NoV	PoC	α	β	γ
1	10644	20	10%	0.2	0.2	0.4
2	21464	20	10%	0.2	0.2	0.4
3	43196	20	10%	0.2	0.2	0.4
4	87642	20	10%	0.2	0.2	0.4

(e) Description of Datasets

Fig. 6. Symbols, Descriptions, and Datasets.

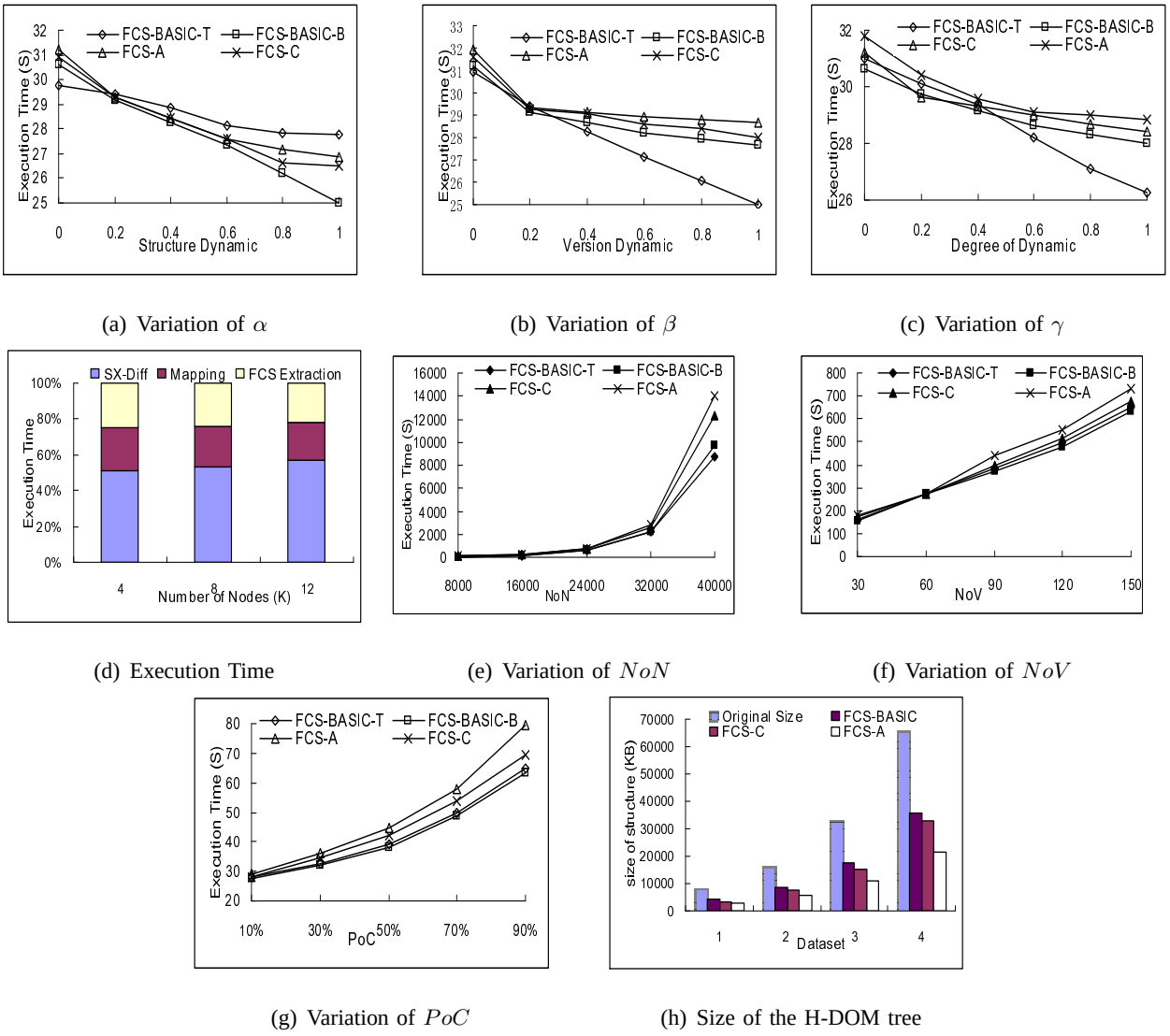
techniques; FCS-C refers to FCS mining algorithm that incorporates the compression and the building and merging techniques. Note that the optimization-based algorithms are implemented using the bottom-up traversal strategy since the metrics can be calculated more efficiently in this way.

We use synthetic XML delta sequences generated from three real and synthetic XML documents. The two real XML documents we use are DBLP and SIGMOD XML downloaded from UW XML repository (http://www.cs.washington.edu/research/xmldatasets), while the synthetic XML is generated using IBM XML Generator (http://www.alphaworks.ibm.com/tech/xmlgenerator). From such XML documents, sequences of XML versions are generated by using our synthetic XML delta generator. We do experiments by using datasets of different characteristics and varying the parameters of each algorithm. For each algorithm, different XML datasets are used to show how the datasets affect the performance. Experiments with the same dataset and all possible variations of the parameters have also been done to show how the parameters can affect the performance. The symbols for characteristics of the datasets and parameters of the algorithms are shown in Tables 6(a) and 6(b) along with their descriptions.

B. Variation of Algorithm Parameters

We evaluate the performance of the four algorithms, FCS-BASIC-T, FCS-BASIC-B, FCS-A, and FCS-C, by varying the thresholds of the three major parameters, α, β, and γ. Table 6(c) shows the characteristics of the datasets and the values of the parameters used in our experiments. Hereafter, we use the symbol "-" to denote the parameters or characteristics of the dataset that will be varied in the experiments. Figure 7(a) shows the performance of the algorithms when α is varied on $SIGMOD1$ XML dataset. Figure 7(b) shows how the algorithms perform when the

threshold β changes. We use the $DBLP1$ XML dataset. Figure 7(c) illustrates how the changes to γ may affect the performance of the algorithms. In this case, we use the $Synthetic1$ XML dataset. From the above figures, following observations can be made.



(a) Variation of α (b) Variation of β (c) Variation of γ

(d) Execution Time (e) Variation of NoN (f) Variation of NoV

(g) Variation of PoC (h) Size of the H-DOM tree

Fig. 7. FCS Experiment Results

The overall observation is that as any of the thresholds increases, the execution time decreases. This is due to the fact that when the threshold increases, the pruning techniques are more efficient and the search space of FCS is reduced. The execution time does not change significantly with the variation of thresholds because the major cost of the algorithms is the cost of SX-Diff, which is independent of the thresholds. As shown in Figure 7(d), the SX-Diff cost is more than 50% of the total cost. From Figure 7(d), we also observed that as the total number of nodes increases

the percentage of SX-Diff cost also increases.

The FCS-BASIC-T algorithm is more stable than others as α changes, but it is more sensitive to the variation of β. This is due to the fact that FCS-BASIC-T uses the heuristic in Lemma 3.1 to prune the H-DOM tree, and other algorithms use the heuristic in Lemma 3.2. Note that Lemma 3.1 is solely based on β and Lemma 3.2 is based on α.

For the same dataset, with the same parameters, we observed that the differences of execution time for the four algorithms are in constant order. It means that although different traversal strategies and optimization techniques are used, the time cost does not change significantly.

C. Characteristics of Datasets

We evaluate the performance of the four algorithms by varying the characteristics of the datasets. Table 6(d) shows the values of the parameters and some of the characteristics of the datasets used in our experiments. Figure 7(e) shows the performance of the algorithms using $SIGMOD2$ by varying the average number of nodes in each XML document from 8,000 to 40,000 (the corresponding size of each XML document is from 3M to 15M). We fix the number of versions to be 30 in the sequence. Figure 7(f) presents the performance of the algorithms using $DBLP2$ by varying the number of versions in the sequence from 30 to 150. The average size of each XML document is 2.3M (5743 nodes) for this experiment. Figure 7(g) evaluates the performance of the algorithms by varying PoC. The $Synthetic2$ dataset is used. From the above figures, several observations can be made.

As the average number of nodes in the XML document increases, the time cost increases. Changes are more significant compared to the changes in Figures 7(f) and 7(g). It is because when the average number of nodes increases, the SX-Diff cost increases as well as the pruning and extraction cost.

As the total number of versions in the XML sequence increases, the execution times of the algorithms increase too. It is obvious that when the total number of versions increases, the number of comparison increases accordingly. Consequently, the cost for detecting the structural changes increases. Compared to the variation of NoN, the variation of NoV do not affect the performance significantly.

As the percentage of changes (PoC) in the XML sequence increases, the execution time also increases. Since our FCS mining is actually dealing with the deltas rather than the original

sequence, as the PoC increases, the size of the delta increases. Consequently, the cost of SX-Diff, the pruning and extraction increases.

D. Compression Efficiency

We evaluate the space efficiency of the algorithms by comparing the compactness of the H-DOM tree. Table 6(e) shows the characteristics of the datasets and parameters used in the experiments. Figure 7(h) shows the size of the H-DOM trees for different datasets and different algorithms. We can observe that compared to the original dataset, the H-DOM trees are very compact. The compression rate of the H-DOM tree is almost 50% in the absence of any optimization techniques. When the optimization techniques are incorporated, the FCS-C, and FCS-A are more compact than the FCS-BASIC. Especially, the H-DOM tree that integrates FCS-A is the most compact of all. The compression rate of FCS-A-based H-DOM tree is around 30% for the benchmark datasets. This fact also explains why the time cost of the FCS-C and FCS-A are relatively more expensive as the saving of space incurs extract cost on calculating the dynamic metrics shown in the results shown in Figures 7(a) to 7(g).

E. Summary

From the above experimental results, we can conclude that our proposed algorithms FCS-BASIC-T and FCS-BASIC-B are efficient and scalable while the three optimization techniques improved the space efficiency substantially. Based on the experimental results, if users want to find out FCS with higher version dynamic, the FCS-BASIC-T is recommended. Otherwise, the FCS-BASIC-B is the best choice, since the three optimization techniques work in a bottom-up manner. The FCS-C algorithm can be applied to any XML documents history, while the FCS-A can only be used for valid XML documents.

V. Applications of FCS

Knowledge of FCS can be useful in several applications, such as monitoring interesting structures in a specific domain, FCS-based classifier, evolution-conscious XML query caching, and XML indexing. We briefly discuss below these representative applications. In the next section, we shall elaborate in detail one of these applications.

Discovering interesting FCS: The FCS mining algorithms, introduced in the preceding section, extracts *complete* collection of FCS in the dataset. However, often users may not be interested

in the complete collection especially when the number of frequently changing structures can be very large. Typically, users in a particular domain may be interested in evolution characteristics of structures that contain specific types of information. For instance, in the e-commerce domain, the material control people may be more interested in the evolution pattern of substructures that contain *products* information; while the marketing people may be more interested in parts of the document that contain *clients* information. We call such structures as *interesting FCS* (i-FCS). In the next section, we shall elaborate on how such structures can be discovered by incorporating user-specified *concept hierarchy* in the FCS mining framework.

FCS-based classifier: Classifying XML documents based on the structures embedded in documents is proposed in [31]. This approach focus on classifying structures based on snapshot data only. Using a FCS-based classifier, we can classify subtrees in XML documents based on their evolution patterns. For example, consider the evolution features of a set of structures in Figure 11. It is possible to detect certain trend-based patterns from the evolution history of these structures. For instance, one can observe that the structural dynamic values of *stamp* and *TV series* have an increasing and decreasing trend, respectively. Also, the *book* substructure shows a periodic change pattern. Hence, we can build a classifier that can classify the collection of FCS based on the nature of their evolution pattern.

FCS-based evolution-conscious XML query caching: Caching XML queries has been recognized as an orthogonal approach to improve the performance of XML query engines [29]. One of the efforts in this direction is to discover the *frequent query patterns* from the historical query log and cache the corresponding query results to reduce the response time for future queries that are the same or similar [29]. The intuition behind this approach is that some query patterns are more popular or important than others and they are expected to be issued more often in the future with higher probabilities compared to others.

Our initial investigation revealed that existing XML caching strategies are solely based on statistics obtained by treating historical XML queries as snapshot data. That is, the frequent XML query patterns are based on only the number of occurrences of the query subtrees in the history. The evolutionary nature of XML queries as well as the underlying XML documents are not taken into consideration.

We believe that the results of FCS mining can be used for developing more effective caching

strategy that takes into account the evolutionary properties of XML structures. Intuitively, our strategy is based on the following principle. If the results of frequent query patterns contain FCS then it should have lower caching priority compared to query patterns that return "FCS-free" results. This is because, although the query patterns are frequent, the query results are expected to change frequently and significantly in the future and hence these patterns should not have higher caching priority. To differentiate the priorities among these two categories of frequent query patterns, we rank the query patterns not only based on the number of occurrences but also based on the evolution patterns of underlying documents.

XML indexing: One of the key issue of XML indexing is to identify the ancestor and descendant relationship quickly. To this end, different numbering schemes have been proposed [18], [14]. Li and Moon proposed a numbering scheme in XISS (XML Indexing and Storage System) [18] that uses an *extended preorder* and a *size*. The *extended preorder* allows additional nodes to be inserted without reordering and the *size* determines the possible number of descendants. More recently, XR-Tree [14] was proposed to index XML data for efficient structural joins. Compared with the XR-tree [14], XISS numbering scheme is more flexible and can deal with dynamic updates of XML data more efficiently. However, Li and Moon did not highlight on how much extra space should be allocated. Allocating too small reserved space will lead to the ineffectiveness in maintaining the numbering scheme, whereas allocating too much extra space will lead to too large numbers being assigned to nodes in a large XML document. Moreover, in the XISS approach, the gaps are equally allocated, while in practice different parts of the document change with different significance. Based on FCS mining results, the numbering scheme can be improved by allocating the gaps in a more intelligent manner. For example, for the parts of structure that change frequently and significantly, larger gaps are allocated while for structures that do not evolve frequently, smaller gaps can be reserved. By using this strategy, the numbering scheme should be more efficient in terms of both index maintenance and space allocation.

VI. Extraction of Interesting FCS

In this section, we elaborate on how FCS mining framework can be used to extract *interesting FCS* (denoted as i-FCS). A set of user-specified *concepts* is used to guide the interesting FCS mining process. These concepts represent the semantic objects in the XML documents. There are two approaches to obtain such concepts. The first approach is to extract interesting concepts from

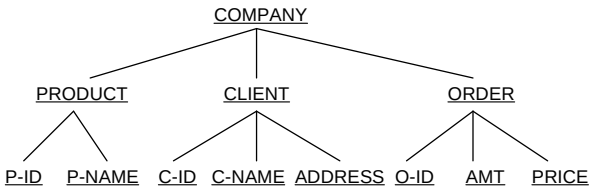


Fig. 8. An Example of Concept Hierarchy

domain-specific ontology. The second approach is to build the concepts based on DTDs/XML Schemas used in this domain to represent the collection of XML documents. We represent interesting concepts in form of a *concept hierarchy* that specifies the relation among them. Nodes in the concept hierarchy can be classified as *primitive* or *nonprimitive*. The *primitive* concepts, which represent the basic elements in a domain, reside in the lowest level in the hierarchy; all *nonprimitive* concepts, which consist of a conglomeration of the primitive concepts, reside in the higher level of the hierarchy. The higher the node's level, the more complex is the concepts it represents. Figure 8 shows an example of concept hierarchy. The leaf nodes such as *P-ID*, and *P-NAME* are primitive concepts; while internal nodes and root node such as *CLIENT* and *COMPANY* are nonprimitive concepts. In our i-FCS mining, we assume that the specified concept hierarchy is provided by users.

Given the user-specified concept hierarchy, thresholds for evolution metrics, and a sequence of versions of XML documents, the goal of i-FCS mining is to discover *interesting FCS* from the document collection. Given a concept C in a specific domain, a structure t in an XML document is an *interesting* structure with respect to concept C, denoted as $t \simeq C$, if t provides the required information of the concept C. Furthermore, if t is a frequently changing structure then it is called *interesting FCS*. We now elaborate on the algorithm for discovering i-FCS.

A. i-FCS Mining Algorithm

The i-FCS mining algorithm is similar to the FCS mining algorithm and consists of two main phases: the *H-DOM tree construction* phase and the *i-FCS extraction* phase. As the H-DOM tree construction phase is similar to the one discussed earlier, we focus on the i-FCS extraction algorithm. The formal algorithm is shown in Figure 5(b).

Given the H-DOM tree, the i-FCS extraction algorithm is to extract all i-FCS based on the user-defined concept hierarchy and evolution constraints. First the substructures are compared

```
<!ELEMENT root (category*)>
<!ELEMENT category (topic*)>
<!ELEMENT topic (item*)>
<!ELEMENT item(listing*)>
<!ELEMENT listing (seller_info,payment_types,shipping_info,
     buyer_protection_info,auction_info,bid_history,item_info)>
<!ELEMENT seller_info (seller_name,seller_rating)>
<!ELEMENT seller_name (#PCDATA)>
<!ELEMENT seller_rating (#PCDATA)>
<!ELEMENT payment_types (#PCDATA)>
<!ELEMENT shipping_info (#PCDATA)>
<!ELEMENT buyer_protection_info (#PCDATA)>
<!ELEMENT auction_info(current_bid,time_left,high_bidder,num_items,
     num_bids,started_at,bid_increment,opened,closed,id_num,notes)>
<!ELEMENT current_bid (#PCDATA)>
<!ELEMENT time_left (#PCDATA)>
<!ELEMENT high_bidder (bidder_name,bidder_rating)>
<!ELEMENT bidder_name (#PCDATA)>
<!ELEMENT bidder_rating (#PCDATA)>
```

```
<!ELEMENT num_items (#PCDATA)>
<!ELEMENT num_bids (#PCDATA)>
<!ELEMENT started_at (#PCDATA)>
<!ELEMENT bid_increment (#PCDATA)>
<!ELEMENT location (#PCDATA)>
<!ELEMENT opened (#PCDATA)>
<!ELEMENT closed (#PCDATA)>
<!ELEMENT id_num (#PCDATA)>
<!ELEMENT notes (#PCDATA)>
<!ELEMENT bid_history (highest_bid_amount,quantity)>
<!ELEMENT highest_bid_amount (#PCDATA)>
<!ELEMENT quantity (#PCDATA)>
<!ELEMENT item_info (color, weight, years,
    brand, description)>
<!ELEMENT color (#PCDATA)>
<!ELEMENT weight (#PCDATA)>
<!ELEMENT years (#PCDATA)>
<!ELEMENT brand (#PCDATA)>
<!ELEMENT description (#PCDATA)>
```

Fig. 9. DTD of the real Yahoo auction data.

with the user-specified concept hierarchy as shown in line 3 in Figure 5(b). If the structures are instances of the concepts in the hierarchy, then the values of the required parameters (version dynamic, and DoD) for each node are calculated and compared against the predefined thresholds as shown in lines 5 and 6. Note that the comparison strategy of the evolution metrics is same as that discussed in FCS mining. Also, we use the bottom-up traversal approach for traversing the H-DOM tree since the set of interesting concepts is represented in a hierarchical manner with primitive concepts in the lower level.

B. Performance Evaluation

In this section, we evaluate the performance of i-FCS mining algorithm for extracting interesting FCS. We use real datasets extracted from Yahoo! auction site (http://auctions.yahoo.com/) for our performance study. A portion of the DTD of the XML view of the data is shown in Figure 9. The maximum depth of the XML document is *7* and the average depth is *5.37*. To obtain different versions of the auction data, we issue queries related to all the five categories (music, tv, movies, books, and collectibles) of products periodically and each result set is then transformed into an XML document. For instance, we issue the five queries every hour and convert the results into an XML document. In the following experiments, we assume that users are interested in the following two concepts: *product category* and *product*. The dataset consists of 120 versions of auction XML data and is shown in Figure 10 (a).

Efficiency and Scalability: First, we evaluate the efficiency and scalability of the i-FCS algorithm and compare it with the FCS algorithm (FCS-BASIC-T).

Figure 10(b) shows how the running time changes by varying the total number of nodes in the XML documents. There are two ways of increasing the total number of nodes. One way is to increase the number of versions (NoV) in the XML sequence, another way is to increase the average number of nodes (NoN) in each version. To increase the number of versions, we need to crawl the data more frequently, while to increase the size of each version, we need to crawl the data less frequently. Datasets D_2 and D_3 are used in this set of experiments. We set $\alpha = \beta = \gamma = 0.2$. Both results show good scalability with the total number of nodes, while the running time is more sensitive to the number of versions in the XML sequence than the average number of nodes in each version. The reason is that the change detection process is the most expensive phase of the algorithm as shown in Figure 10(c). Moreover, the i-FCS mining algorithm is around *2* times faster than the FCS algorithm for the benchmark dataset as only a portion of the H-DOM tree is processed.

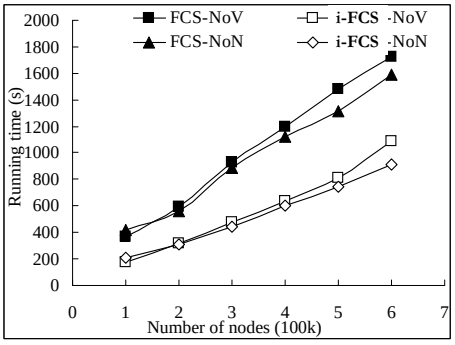
Figure 10(c) shows the cost of each phase in the i-FCS mining algorithm. The dataset D_1 is used and $\alpha = \beta = \gamma = 0.2$. Similar to the FCS mining results, it can be observed that the change detection phase takes a share of up to 40% of the cost. Compared to the cost of SX-Diff in FCS algorithm in Figure 7(d), the percentage of cost for detecting changes decreases in the i-FCS algorithm. This is because only changes to the user-specified interesting structures are detected.

Effects of Parameters: Figure 10(d) shows how the running time changes by varying the thresholds of evolution metrics. We use the D_1 dataset. In the three experiments, we vary one of the thresholds and fixed the thresholds for the other two to 0.2. It can be observed that the running time does not change significantly when the thresholds of evolution metrics are varied. This is due to the fact that the most expensive process, SX-Diff, is independent of the thresholds.

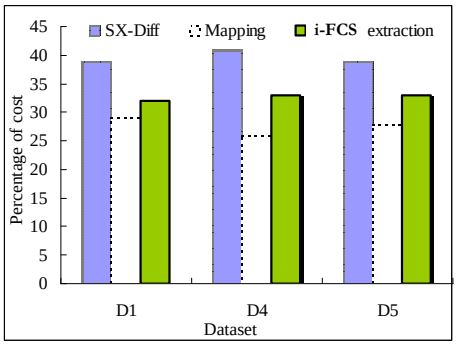
Figure 10(e) shows how the depth of the concept structure that users are interested in affect the performance of the i-FCS algorithm. In this set of experiments, dataset D_1 is used. Fixing the thresholds of the evolution metrics to 0.2, the targeted concepts are varied from level 2 (denoted as case 1), level 3 (denoted as case 2), level 2 and 3 (denoted as case 3), level 4 (denoted as case 4), to level 3 and level 4 (denoted as case 5). It can be observed that the maximum level of the concepts directly affects the performance of the i-FCS algorithm. The smaller the minimum level, the more efficient is the i-FCS algorithm. For concepts that have the same maximum

Dataset	Num. of versions	Avg size
D1	120	2500
D2	120	varying
D3	varying	2500
D4	120	5000
D5	240	2500

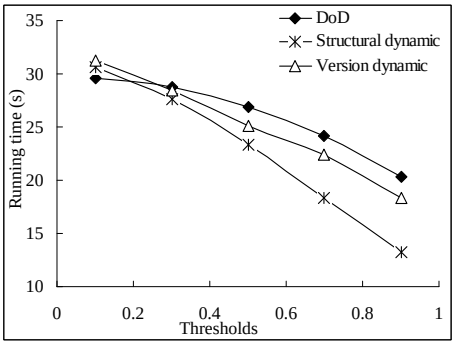
(a) Dataset



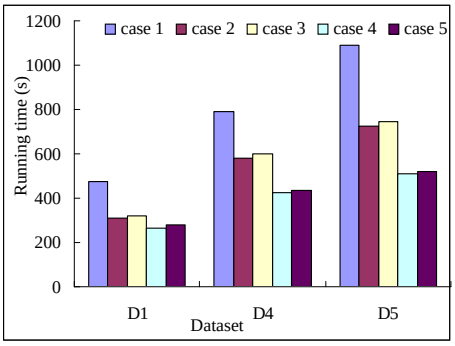
(b) Running time 1



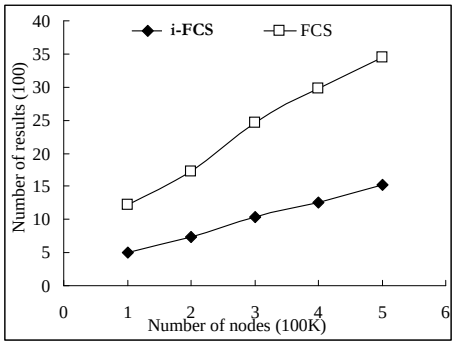
(c) Cost comparison



(d) Running time 2



(e) Running time 3

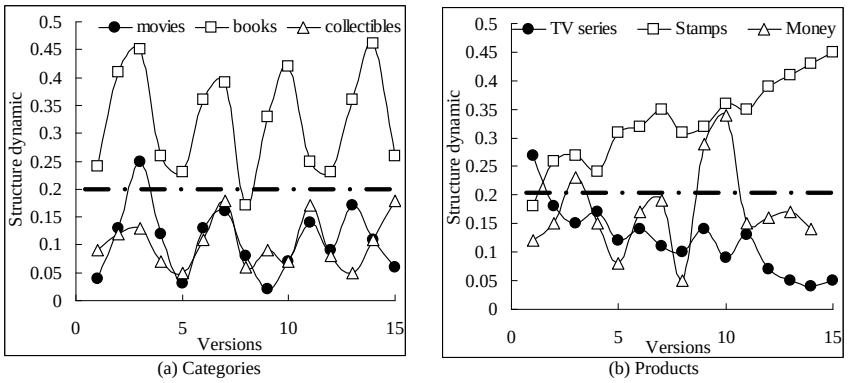


(f) Num. of results

Fig. 10. Evaluation of i-FCS.

level, the performance is quite similar as shown in this figure. This is because the maximum level of the targeted concepts determines the maximum level the i-FCS algorithm explores for both change detection and computation of values of evolution metrics.

Figure 10(f) shows the number of structures in the mining results using the above thresholds for the evolution metrics. It shows that the number of structures in i-FCS mining result is reduced by almost 40% compared to that of FCS mining result. This is because the search space for discovering FCS is reduced considerably as i-FCS only process substructures that users are interested in.



(a) Categories (b) Products

Fig. 11. Evolutionary features of various categories.

Analysis of Mining Results: In the above experiments, we have successfully extracted interesting concepts such as *frequently changing products* and *frequently changing product categories*. We now show that these concepts are indeed FCS by analyzing some of the representative interesting FCS in the actual data. Along with this, we also show the evolution patterns of some other concepts that occurred in the auction datasets. These structures were not considered as FCS as they did not satisfy the evolution metrics.

In Figures 11(a) and (b), we present the *structure dynamic* values of three product categories and three specific topics under these categories. Note that the datasets used in Figures 11(a) and (b) are crawled every *6* hours and every *2* hours, respectively. Suppose that $\alpha = \beta = \gamma = 0.2$. Then, *books* and *comics* are two interesting FCS. Note that the dotted lines in Figures 11(a) and (b) are the thresholds for structure dynamic. A larger structure dynamic value indicates that more elements are inserted and deleted under the corresponding subtree. For instance, the results in Figure 11(a) indicate that the *books* is one of the very popular categories where people keep bidding frequently and new products are inserted constantly. Figure 11(b) shows the historical structure dynamic values for a set of products. It can be observed that some products became less popular with time (such as *TV series*), some products became more popular (such as *stamps*), and other products changed in various ways (such as *money*). Observed that *stamps* is a FCS when the threshold values are set to *0.2*.

VII. Related Work

A. XML Data Mining

As XML has emerged as the leading textual language for representing and exchanging data over the Web, the data mining community has been motivated to discover knowledge from collections of XML documents. For example, there have been increasing research efforts in mining frequent patterns [2], [13], [15], [28], [23], [26], [30] or sequential patterns [17] from XML repositories, classifying [31] and clustering [19] XML documents. We review some of these works here.

Most existing work focus on discovering the frequent substructures from a collection of semi-structured data such as XML documents. Wang and Liu [26] developed an Apriori-like algorithm to mine frequent substructures based on the "downward closure" property. They first found the frequent *1-tree-expressions* that are frequent individual *label path*s. Discovered frequent *1-tree-expression*s are joined to generate candidate *2-tree-expression*s. The process is executed iteratively till no candidate *k-tree-expression*s is generated. AGM [13] is an Apriori-based algorithm for mining frequent substructures. But the results of AGM is restricted to only the *induced* substructures. FSG [15] is also an Apriori-based algorithm for mining all *connected* frequent subgraphs. Experiments results in [15] show that FSG is considerably faster than AGM. However, both AGM and FSG do not scale to very large database. gSpan [28] is an algorithm for extracting frequent subgraphs without candidate generation. It employs the depth-first search strategy over the graph database. Like AGM, gSpan needs elaborate computations to deal with structures with non-canonical forms. Asai et al. [2] developed another algorithm, FREQT, to discover all frequent tree patterns from large semi-structured data. They modeled the semi-structured data as *labeled ordered tree* and discover frequent trees level by level. At each level, only the rightmost branch is extended to discover frequent trees of the next level. Thus, efficiency can be achieved without generating duplicate candidate frequent trees.

TreeMinerH and TreeMinerV [30] are two algorithms for mining frequent trees in a forest. TreeMinerH is an Apriori-like algorithm based on a horizontal database format. In order to efficiently generate candidate trees and count their frequency, a smart *string encoding* is proposed to represent the trees. In contrast, TreeMinerV uses vertical *scope-list* to represent a tree. Frequent trees are searched in depth-first way and the frequency of generated candidate trees are counted by joining *scope-lists*. TreeFinder [23] is an algorithm to find frequent trees that

are *approximately* rather than *exactly* embedded in a collection of tree-structured data modeling XML documents. Each labeled tree is described in *relaxed relational description* which maintains ancestor-descendant relationship of nodes. Input trees are clustered if their atoms of *relaxed relational description* occur together frequently enough. Then maximal common trees are found in each cluster by using algorithm of *least general generalization*. Recently, there is another line of work that employs the pattern-growth strategy to discover frequent subtrees [24], [27].

Classification of XML documents has also been addressed by some recent research works [31]. In [31], Zaki proposed an algorithm to construct *structural rules* in order to classify XML documents. The basic idea is to relate the presence of a particular kind of structural pattern in an XML document to its likelihood of belonging to a particular class.

The critical difference between our proposed frequently changing structure mining and existing works on XML data mining is that we address the dynamic nature of XML data. Existing works on XML data mining extract knowledge from the snapshot version of XML documents, whereas we extract knowledge from a sequence of historical structural deltas of an XML document. Furthermore, techniques for frequent substructure mining focus on designing algorithms to extract structures that *occur frequently* in the snapshot data collections. Whereas the goal of FCS mining is to extract structures that *change frequently* from the sequence of historical XML versions.

B. Mining Change Patterns and Trends

There are several techniques proposed recently for maintaining and updating previously discovered knowledge. They focus on two major issues. One is to actualize the knowledge discovered by detecting changes in the data such as the DEMON framework proposed by Ganti et al [9]. Another is to detect interesting changes in the KDD mining results such as the FOCUS framework proposed by Ganti et al [8], PAM proposed by Baron et al [3], and the fundamental rule change detection tools proposed by Liu et al [20]. Our effort differs from these approaches in the following ways. First, these techniques are proposed either for updating the mining results or detecting the changes to the mining results with respect to the changes to the data sources. Unlike our approach, they do not focus on discovering novel patterns from the evolutionary features of data.

Emerging pattern [7] was proposed to capture significant changes and differences between datasets. Basically, emerging patterns are defined as itemsets whose supports increase signif-

icantly from one dataset to another dataset. Thus, when applied to timestamped databases, emerging patterns can capture emerging trends in business or demographic data. Our study is different from emerging pattern in that we consider the changes in a sequence of snapshots of the data while emerging pattern considers only two snapshots. That is, emerging pattern focuses on local changes while our work addresses the global changes. Consequently, emerging pattern only needs to measure the degree of change while our work needs to measure both the degree of change and the frequency of changes. The knowledge discovered by our work and emerging patterns is different as well. For example, emerging patterns capture useful contrasts between two snapshots while frequently changing structures capture the evolutionary characteristics of tree structured XML data.

Temporal Text Mining (TTM) is also concerned with discovering temporal patterns in text information collected over time. Recently, a particular TTM task − discovering and summarizing the evolutionary patterns of themes in a text stream − was proposed by Mei and Zhai [22]. The evolutionary theme patterns (ETP) discovery problem aims to discover the evolution of themes, i.e. the happening of the Asian tsunami disaster, the statistics of victims and damage, the aids from the world and the lessons from the tsunami. ETP refers to patterns of objects (themes) evolving from one status (subtopic) to another status. In contrast, we focus on structural evolution of hierarchical structured data.

In our previous works [34], [35], we proposed novel approaches for mining evolution of web usage data. In [35], we propose the first approach to detect events from the click-through data, which is the log data of web search engines. In [34], we present an algorithm called WAM-Miner to discover *Web Access Motifs* (WAMs) from web usage data. WAMs are web access patterns that never change or *do not change significantly* most of the time (if not always) in terms of their support values during a specific time period. Compared to this work, in this paper we focus on discovering XML structures that change frequently in the history.

Our research is also related to works on regularities in time series. These previous works include partial periodic patterns [10], [11], sequential patterns in single-variable numerical time series [1], frequent episode [21], and the work [16] which studied the problem of efficiently mining the phrases whose frequency history curves match a given shape (trend) in time-stamped text databases. The basic difference between our work and the above works is that most of the above works consider sets of items or sequences of items while our work focus on more complex

tree structured sequences.

VIII. Conclusions and Future Work

This work is motivated by the fact that existing XML data mining strategies focus on discovering knowledge based on statistical measures obtained from the static characteristics of XML data. They do not consider the evolutionary features of the historical XML data. In this paper, we proposed techniques to discover a novel type of frequent pattern named frequently changing substructures (FCS) by analyzing the structural evolution patterns of historical XML documents. Frequently changing substructures are substructures in XML documents that evolve frequently and significantly during a specific time period. We proposed a set of evolution metrics to measure the evolutionary features of XML structures. Based on these proposed metrics, we presented two algorithms that extract FCS from the historical collection of XML documents. These FCS can be used to build interesting FCS monitoring framework in a specific domain. They can also be used in several other applications such as FCS-based classifier, indexing XML documents, XML query caching, etc. Experimental results showed that the proposed algorithms are efficient and scalable. Furthermore, both the algorithms can accurately identify the FCS.

As part of future work, we intend to build some of the other applications of FCS such as FCS-based classifier and XML query caching. We also wish to explore novel types of useful patterns that may be discovered by integrating content and structural evolution of historical XML documents.

References

[1] R. Agrawal and R. Srikant. Mining Sequential Patterns. *In Proc. of ICDE*, pp. 3–14, 1995.

[2] T. Asai, K. Abe, S. Kawasoe, H. Arimura, H. Sakamoto, and S. Arikawa. Efficient Substructure Discovery from Large Semi-structured Data. *In Proc. of SIAM DM*, pp. 158–174, 2002.

[3] S. Baron, M. Spiliopoulou, and O. Gnther. Efficient Monitoring Patterns in Data Mining Environments. *In Proc. of ADBIS*, pp. 253–265, 2003.

[4] D. Braga, A. Campi, S. Ceri, M. Klemettinen, and P. L. Lanzi. Mining Association Rules from XML Data. *In Proc. of DAWAK*, pp. 21–30, 2000.

[5] L. Chen, S. S. Bhowmick, and L.-T. Chia. FRACTURE-Mining: Mining Frequently and Concurrently Mutating Structures from Historical XML Documents. *To appear in Data and Knowledge Engineering Journal (DKE)*, Elsevier Science, 2006.

[6] L. Chen, S. S. Bhowmick, and L.-T. Chia. Mining Positive and Negative Association Rules from XML Query Patterns for Caching. *In Proc. of DASFAA*, 2005.

[7] G. Dong and J. Li. Efficient Mining of Emerging Patterns: Discovering Trends and Differences. *KDD*, 43–52, 1999.

[8] V. Ganti, J. Gehrke, and R. Ramakrishnan. A Framework for Measuring Changes in Data Characteristics. *PODS*, 1999.

[9] V. Ganti and R. Ramakrishnan. Mining and Monitoring Evolving Data. *Handbook of massive datasets*, 593–642, 2002.

[10] J. Han, G. Dong, and Y. Yin. Mining Segment-wise Periodic Patterns in Time-Related Databases. *KDD*, 1998.

[11] J. Han, G. Dong, and Y. Yin. Efficient Mining of Partial Periodic Patterns in Time Series Database. *In Proc. of ICDE.*, pp. 106–115, 1999.

[12] J. Han, J. Pei, and Y. Yin. Mining Frequent Patterns Without Candidate Generation. *In Proc. of SIGMOD*, pp. 1–12, 2000.

[13] A. Inokuchi, T. Washio, and H. Motoda. An Apriori Based Algorithm for Mining Frequent Substructures from Graph Data. *In Proc. of PKDD*, pp. 13–23, 2000.

[14] H. Jiang and H. Lu. XR-Tree: Indexing XML Data for Efficient Structural Joins. *In Proc. of ICDE*, 2003.

[15] M. Kuramochi and G. Karypis. Frequent Subgraph Discovery. *In Proc. of ICDM*, pp. 313–320, 2001.

[16] B. Lent, R. Agrawal, and R. Srikant. Discovering Trends in Text Databases. *In Proc. of KDD*, pp. 227–230, 1997.

[17] H. P. Leung, F. L. Chung, and S. C. Chan. A New Sequential Mining Approach to XML Document Similarity Computation. *In Proc. of PAKDD*, 2003.

[18] Q. Li and B. Moon. Indexing and Querying XML Data for Regular Path Expressions. *The VLDB Journal*, 361–370, 2001.

[19] W. Lian, D. W. Cheung, N. Mamoulis, and S. M. Yiu. An Efficient and Scalable Algorithm for Clustering XML Documents by Structure. *IEEE TKDE*, vol.16, no.1, 2004.

[20] B. Liu, W. Hsu, and Y. Ma. Discovering the Set of Fundamental Rule Changes. In *In Proc. of KDD*, pp. 335–340, 2001.

[21] H. Mannila, H. Toivonen, and A.I. Verkamo. Discovering Frequent Episodes in Sequences. *KDD*, pp. 210–215, 1995.

[22] Q. Mei and C. Zhai. Discovering Evolutionary Theme Patterns from Text: An Exploration of Temporal Text Mining. *In Proc. of KDD*, pp. 98–207, 2005.

[23] A. Termier, M-C. Rousset, and M. Sebag. TreeFinder: a First Step towards XML Data Mining. *In Proc. of ICDM*, 2002.

[24] C. Wang, M. Hong, J. Pei, H. Zhou, W. Wang, and B. Shi. Efficient Pattern-Growth Methods for Frequent Tree Pattern Mining. *In Proc. of PAKDD*, 2004.

[25] Y. Wang, D. J. DeWitt, and J.-Y. Cai. X-Diff: An Effective Change Detection Algorithm for XML Documents. *In Proc. of ICDE*, pp. 519–530, 2003.

[26] K. Wang and H. Liu. Discovering Structural Association of Semistructured Data. *IEEE TKDE*, vol.12, 2000.

[27] Y. Xiao, J. F. Yao, Z. Li, and M. H. Dunham. Efficient Data Mining for Maximal Frequent Subtree. *ICDM*, 2003.

[28] X. Yan and J. Han. gSpan: Graph-based Substructure Pattern Mining. *In Proc. of ICDM*, pp. 721–724, 2002.

[29] L. Yang, M. Lee, W. Hsu, and X. Guo. 2PX-Miner: an Efficient two Pass Mining of Frequent XML Query Patterns. *In Proc. of ACM SIGKDD*, pp. 731–736, 2004.

[30] M. J. Zaki. Efficiently Mining Frequent Trees in a Forest. *In Proc. SIGKDD*, pp. 71–80, 2002.

[31] M. J. Zaki and C. C. Aggarwal. XRULES: An Effective Structural Classifier for XML Data. *In Proc. of SIGKDD*, pp. 316–325, 2003.

[32] S. Zhang, J. Zhang, H. Liu, and W. Wang. XAR-Miner: Efficient Association Rules for XML Data. *WWW*, 2005.

[33] Q. Zhao, S. S. Bhowmick, M. Mohania, and Y. Kambayashi. Discovering Frequently Changing Structures from Historical Structural Deltas of Unordered XML. *In Proc. of ACM CIKM*, 2004.

[34] Q. Zhao, S. S. Bhowmick, and L. Gruenwald. WAM-Miner: In the Search of Web Access Motifs from Historical Web Log Data. *In Proc. of ACM CIKM*, 2005.

[35] Q. Zhao, T.-Y Liu, S. S. Bhowmick, and W.-Y Ma. Event Detection from Evolution of Click-Through Data. *KDD*, 2006.