FRACTURE–Mining: Mining Frequently and Concurrently Mutating Structures from Historical XML Documents

Ling Chen   Sourav S. Bhowmick   Liang-Tien Chia

School of Computer Engineering, Nanyang Technological University
639798, Singapore {pg02322722, assourav, asltchia}@ntu.edu.sg

---

Abstract

In the past few years, the fast proliferation of available XML documents has stimulated a great deal of interest in discovering hidden and nontrivial knowledge from XML repositories. However, to the best of our knowledge, none of existing work on XML mining has taken into account the dynamic nature of XML documents as online information. The present article proposes a novel type of frequent pattern, namely, *FRequently And Concurrently muTating substructUREs* (*FRACTURE*), that is mined from the evolution of an XML document. A discovered *FRACTURE* is a set of substructures of an XML document that frequently change together. Knowledge obtained from *FRACTURE* is useful in applications such as XML indexing, XML clustering etc. In order to keep the result patterns concise and explicit, we further formulate the problem of *maximal FRACTURE* mining. Two algorithms, which employ the *level-wise* and *divide-and-conquer* strategies respectively, are designed to mine the set of *FRACTURE*s. The second algorithm, which is more efficient, is also optimized to discover the set of *maximal FRACTURE*s. Experiments involving a wide range of synthetic and real-life datasets verify the efficiency and scalability of the developed algorithms.

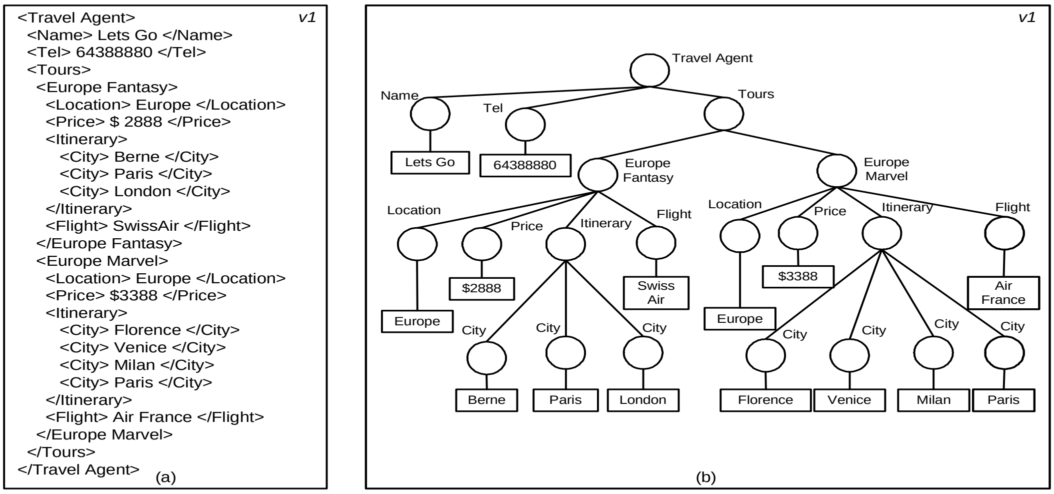
Key words: XML, Frequent Pattern, Structural Delta

---

1 Introduction

Developed under auspices of W3C in 1998, XML is rapidly emerging as the *de facto* standard for data representation and exchange on the Web. The self-describing property empowers XML to represent information without loss of semantics. The semi-structured nature allows XML to model a wide variety of databases. Not surprisingly, industries are indeed enthusiastic about XML, which leads to the fast proliferation of XML data.

Preprint submitted to Elsevier Science   *14 August 2005*



```
<Travel Agent>
  <Name> Lets Go </Name>
  <Tel> 64388880 </Tel>
  <Tours>
    <Europe Fantasy>
      <Location> Europe </Location>
      <Price> $ 2888 </Price>
      <Itinerary>
        <City> Berne </City>
        <City> Paris </City>
        <City> London </City>
      </Itinerary>
      <Flight> SwissAir </Flight>
    </Europe Fantasy>
    <Europe Marvel>
      <Location> Europe </Location>
      <Price> $3388 </Price>
      <Itinerary>
        <City> Florence </City>
        <City> Venice </City>
        <City> Milan </City>
        <City> Paris </City>
      </Itinerary>
      <Flight> Air France </Flight>
    </Europe Marvel>
  </Tours>
</Travel Agent>
```

Fig. 1. XML Document and Tree Representation

With the ever-increasing amount of available XML data, data mining community has been motivated to discover underlying but interesting knowledge from XML. For example, recently, there has been increasing research effort in classifying XML documents [26]; clustering XML data [14] [16]; and mining sequential patterns [15] or frequent patterns [5][20] from XML repositories. Currently, two types of data in XML has been studied to find frequent patterns: XML content and XML structure. The former aims to discover patterns of frequent data values [5]; the latter focuses on discovering patterns of frequent substructures [20].

Besides the self-describing and semi-structured properties, XML has another feature that it is dynamic. As online information, XML may change at any time in any way. Consequently, issues related to detecting changes to XML documents received considerable attention recently [23][8]. Detected changes can be used in XML query systems, search engines etc [23]. Actually, the changes to XML documents can be further studied. Since changes to XML documents do not just occur randomly, there may be interesting and nontrivial knowledge hidden in these changes. Thus, the changes to XML documents can be exploited by data mining techniques to discover novel knowledge. In this paper, we consider a sequence of historical versions of an XML document to discover knowledge from the sequence of corresponding changes. We illustrate various novel knowledge that can be discovered from a sequence of changes to an XML document with the following motivating example.

1.1 Motivation

An XML document can be represented as a tree according to Document Object Model (DOM) specification. For example, consider the XML file in Figure 1 (a) which describes the information of a travel agent. It can be modeled as a tree as shown in Figure 1 (b). The itineraries provided by a travel agent

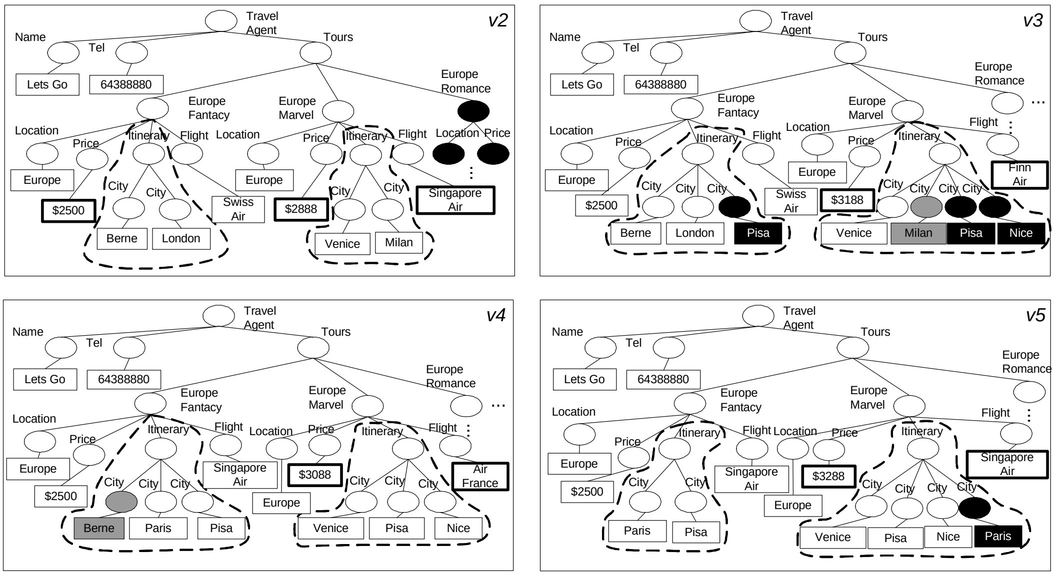


Fig. 2. A Sequence of Historical Versions

might be adjusted from time to time according to factors such as seasons and profits etc. Figure 2 presents another four historical versions of the XML tree along the time sequence. The black nodes and grey nodes in the figure depict the insertion and deletion of elements respectively, while the nodes with thick boundaries depict the modification of element values.

Before discussing the knowledge that can be discovered from the sequence of changes to an XML document, we first investigate the different types of changes to XML. Corresponding to the classification on XML data, changes to XML can be divided into the following two categories.

- **Changes to XML structure (also called *structural deltas*)**. Structural deltas mean the changes in the hierarchical topology composed of nodes and edges in an XML tree, which are usually resulted in by the change operations of insertions and deletions. For example, the inserted node "City" with its value of "Pisa" in the *v3* of Figure 2 belongs to structural deltas.
- **Changes to XML content (also called *content deltas*)**. Content deltas mean the changes in the values of nodes, which are usually caused by the change operations of modifying nodes. For example, the value of the node "Price" under the itinerary of "Europe Marvel" is *\$2888* in *v2* of Figure 2, whereas the value of the node is *\$3188* in *v3*. Hence, the value of the node is changed and the change belongs to content deltas.

Then, novel knowledge, such as sequential patterns and classification rules based on different types of XML deltas, might be discovered from the sequence of historical versions of an XML document. In the following, we enumerate the novel frequent patterns that might be discovered from the sequence of changes to an XML document.

- **Frequent patterns in structural deltas**: A frequent pattern mined from structural deltas is a set of subtrees whose structures frequently mutate together. That is, we consider the structural deltas in terms of *changed subtrees*. Once a node is inserted or deleted, the subtree containing it is changed. Then, we aim to discover which subtrees frequently change together in their structures. For example, consider the sequence of structural deltas in Figure 2. Since the nodes of "City" in the subtrees rooted at the node "Europe Fantasy/Itinerary" and the node "Europe Marvel/Itinerary" (highlighted by the dotted line) are inserted and deleted together frequently, the two subtrees will be discovered as a frequent pattern. Since it is rather common in an XML document that the structure of an object is designed to reflect its semantics, we can glean the knowledge from such a pattern that the two objects represented by the two subtrees may have some underlying association. For example, the two itineraries are associated in the cities on the routes.
- **Frequent patterns in content deltas**: A frequent pattern mined from content deltas is a set of nodes whose values are frequently changed together. For example, from the sequence of content deltas exhibited in Figure 2, we observe that when the value of the node "Flight" of the itinerary "Europe Marvel" was changed, the value of the node "Price" of the itinerary was changed as well. Hence, the two nodes form a frequent pattern mined from the content deltas. The knowledge that can be inferred from such a pattern is that the set of nodes may have some underlying association. For example, the flight may be a factor that influences the price of the itinerary.
- **Frequent patterns in hybrid deltas**: Certainly, frequent patterns can also be mined from changes to XML documents without discriminating content and structural deltas. Thus, a frequent pattern discovered from hybrid deltas is a set of disjointed fragments of an XML document that change together frequently in either data values or structures. For example, we may discover a frequent pattern of the two fragments embedded in the elements of "Europe Fantasy" and "Europe Marvel" respectively from the above example. Knowledge can be obtained from such a pattern that the two routes are related in prices, itineraries or flights.

Hence, considering the dynamic nature of XML documents, we identified a new domain for XML mining. Namely, a sequence of changes to an XML document. Novel and interesting knowledge can be discovered from them, which can be used in a wide range of applications, such as native XML storage and approximate XML change detection etc. The details will be discussed in Section 6.

In this paper, we focus on the problem of mining the set of frequent patterns from XML structural deltas, where each pattern is a set of subtrees with their structures frequently mutating together. We call such a pattern as a *FRACTURE* (*FRequently And Concurrently muTating substructUREs*). A short version of the paper appeared in [7].

FRACTURE mining is a challenging problem as existing techniques of XML frequent pattern mining cannot be applied. This is due to the following two reasons: (1) existing approaches aim to find frequently occurring substructures, whereas we need to search for frequently and concurrently mutating substructures; (2) existing approaches find frequent patterns from the whole collection of XML documents while we only need to consider the sequence of structural deltas between each two historical documents, which should be more space- and time-efficient.

1.2 Roadmap of the Paper

The remainder of the paper is organized as follows. Section 2 gives an overview of our approach and presents the main contributions of the paper. Section 3 formally defines the problem of *FRACTURE* mining and *maximal FRACTURE* mining. Section 4 describes the mining procedure and developed algorithms, *Apriori-FRACTURE* and *FPG-FRACTURE*, and optimizing strategies for *maximal FRACTURE* mining. Section 5 evaluates the performance of the algorithms based on experimental results. We discuss the applications of FRACTUREs in Section 6 and review related work in Section 7. The last section concludes the paper.

2 Overview and Contributions

Given a sequence of historical versions of an XML document, we study the sequence of structural deltas between each pair of successive versions. Changes to the structures of an XML document are considered in terms of changed subtrees. Then, the goal of *FRACTURE mining* is to discover subtrees that frequently change together in their structures.

We treat an XML tree as a collection of subtrees and aim to discover frequent patterns of subtrees from any level of the XML tree. Once a node is inserted or deleted, all subtrees containing the node are changed. Thus, requiring that subtrees should frequently change together to be a *FRACTURE* will result in many *FRACTURE*s containing subtrees with ancestor relationships. For example, in Figure 2, every time a node “City” under the node “Europe Fantasy/Itinerary” is inserted or deleted, both the subtree rooted at the node “Itinerary” and the subtree rooted at the node “Europe Fantasy” are changed. However, discovering the two subtrees as a *FRACTURE* does not make any sense as the knowledge that the two subtrees frequently change together is too trivial. Actually, when a node is inserted or deleted, it has different influence on changing the structures of different subtrees containing it. For example, since the subtree rooted at the node “Europe Fantasy” contains more structures (nodes) than the subtree rooted at the node “Itinerary” does, inserting or deleting a small number of “City” nodes may not be significant changes to the former subtree (the significance of changes will be

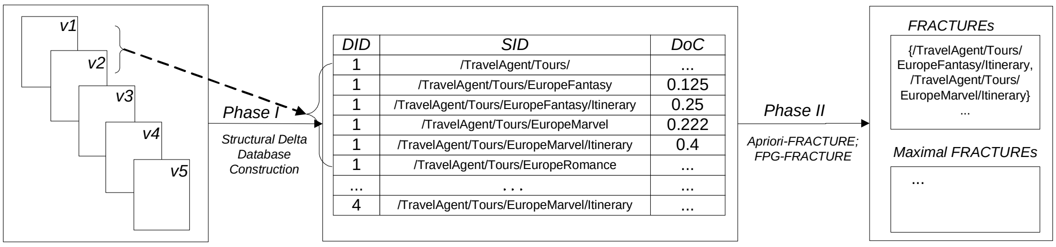


Fig. 3. Overview of the Mining Process

defined in Section 3), whereas it may be significant to the latter. Hence, to make a *FRACTURE* represent nontrivial knowledge, we formulate the notion of *FRACTURE* in consideration of not only the *frequency of change* but also the *degree of change* of a set of subtrees. In other words, we require the set of subtrees of a *FRACTURE* should not only frequently change together but also frequently change significantly when they change together.

After defining the *FRACTURE* in this manner, we observed a "subsumption" relationship between some specific pair of subtree sets. That is, in such a specific pair, if a subtree set is a *FRACTURE*, we can infer directly that the other set must be a *FRACTURE* as well. In order to prune redundancy, we further define the notion of *maximal FRACTURE* based on the "subsumption" relationship. A *FRACTURE* is maximal only if it cannot be inferred from some other *FRACTURE*s. Thus, the set of *maximal FRACTURE*s is more concise than the set of *FRACTURE*s, while the complete set of *FRACTURE*s can be inferred from the set of *maximal FRACTURE*s.

The discovery of the set of *FRACTURE*s is performed in two phases, as shown in Figure 3. In the first phase, given an input of a sequence of historical versions of an XML document, we need to detect the sequence of structural deltas and build a *structural delta database*. Each tuple of a *structural delta database* is a triplet, <*DID*, *SID*, *DoC*>, where *DID* is the identifier of a *delta*, which is a comparison of two successive historical versions, *SID* denotes the identifier of a changed subtree and *DoC* records the *degree of change* for the subtree in the two versions. For example, as shown in Figure 3, comparing the first two historical versions results in the first six entries in the *structural delta database* where the *delta* ID is one. (we use the path leading to the root of the subtree to identify a subtree, and the calculation of *degree of change* for a changed subtree will be explained later). The constructed *structural delta database* will be the input of the second phase. We developed two algorithms, *Apriori-FRACTURE* and *FPG-FRACTURE*, to discover the set of *FRACTURE*s from the database. The former is an *apriori*-like algorithm that employs the *level-wise* strategy; while the latter is based on the well-known FP-growth algorithm [10]. Both algorithms can discover the set of *FRACTURE*s completely. Furthermore, we developed several optimization techniques for the algorithm *FPG-FRACTURE* to efficiently discover the set of *maximal FRACTURE*s.

In summary, the main contributions of this paper are as follows.

- We considered the dynamic nature of XML documents to exploit the sequence of changes to an XML document as a new domain for XML mining. We investigated different types of novel knowledge (frequent patterns) that can be mined from this domain.
- We focused on the sequence of structural deltas to formally define the problem of *FRACTURE* mining. To keep the result patterns concise, we further formulate the problem of *maximal FRACTURE* mining.
- We developed two algorithms based on different strategies to mine the set of *FRACTURE*s and optimized the efficient algorithm *FPG-FRACTURE* to discover the set of *maximal FRACTURE*s.
- We implemented all the algorithms and conducted experiments over a wide range of synthetic and real-life datasets to evaluate their efficiency and scalability.

3 Problem Statement

In this section, we first describe some preliminary concepts and basic change operations that result in structural deltas. Then, we define the metrics to measure the *degree of change* and the *frequency of change* for subtree sets. Finally, the *FRACTURE* and *maximal FRACTURE* are defined based on the metrics.

3.1 Preliminary Definitions

An XML document can be represented as a tree according to Document Object Model (DOM) specification. Although DOM specifies that element nodes and text nodes are ordered, XML documents can be treated as unordered trees in many applications [23]. Hence, in this paper, we model the structure of an XML document as an unordered tree $T = (N, E)$, where N is the set of nodes and E is the set of edges. Then substructures in the XML document can be modeled as *subtrees*. A tree $t = (n, e)$ is a subtree of an XML tree T, denoted as $t \prec T$, if and only if $n \subset N$ and for all $(x, y) \in e$, x is a parent of y in T. Then, we treat an XML tree T as a *forest* which is a collection of subtrees $t \prec T$. Furthermore, we call subtree $\hat{t}$ an ancestor of subtree t if the root of $\hat{t}$ is an ancestor of the root of t. Conversely, subtree t is a descendant of subtree $\hat{t}$.

Traditional XML change detection systems [23] [8] usually define three types of basic change operations: insertion, deletion and modification. Since an operation of "modification" only affects the value of a node, it does not cause any structural changes. Hence, we consider the following basic change operations that result in changes to XML structures.

- *Insert(x(name, value), y)*: This operation creates a new node x, with node name "name" and node value "value", as a child node of node y in an XML tree. The set of black nodes in Figure 2 represents this operation.

- *Delete(x)*: This operation is the inverse of the insertion one. It removes node x from an XML tree. The grey nodes in Figure 2 illustrate this operation.

The two basic change operations can be combined to form composite operations such as *Insert* (t_x, y) and *Delete* (t_x), which insert a subtree t_x rooted at node x to node y and delete a subtree t_x respectively. For example, a subtree rooted at node "Europe Romance" was inserted to the node "Tours" in *v2* of Figure 2. In the context of *FRACTURE* mining, only the defined change operations will be taken into account. In other words, we consider the changes caused by the operations defined above as structural deltas of an XML document.

3.2 Metrics

Now we introduce the metrics we defined to measure the *degree of change* and the *frequency of change* for subtree sets.

3.2.1 Degree of Change

As we mentioned above, once a node is inserted or deleted, all subtrees containing it are changed. However, the change operation may have different influence on changing the structures of different subtrees. We quantify the *degree of change* for a subtree between two versions with a distance measure, which is based on the concept of *edit distance* in change detection systems. Edit distance is defined to be the minimum number of change operations required to transform one version to another [23]. Likewise, in the context of *FRACTURE* mining, we define edit distance as the minimum number of basic change operations (*insert* or *delete*) required to transform the structure of one version to the structure of the other. Then, we normalize the distance by the total number of unique nodes of the subtree in two versions so that the same edit distance will have slighter influence on changing the structure of a subtree containing larger number of nodes. The metric of *degree of change* (denoted as *DoC*) for a subtree is formally defined as follows:

Definition 1 [***Degree of Change***] *Let* $<t^i, t^{i+1}>$ *be the ith and the (i+1)th historical versions of a subtree t in an XML tree structure T. Let* $|d(t, i, i+1)|$ *be the edit distance of t from the ith version to the (i+1)th version. Let* $\cup$ *be the operation which unions the nodes in two subtrees. Then,* $|t^i \cup t^{i+1}|$ *is the number of unique nodes of tree t in ith version and (i+1)th version. The degree of change for subtree t from version i to version (i+1) is:*

$$DoC(t,\ i,\ i{+}1) = \frac{|d(t,i,i+1)|}{|t^i \cup t^{i+1}|} \qquad \square$$

If the subtree does not change in the two versions, then its *DoC* will be zero; if the subtree is totally removed or newly inserted, then the *DoC* of the subtree

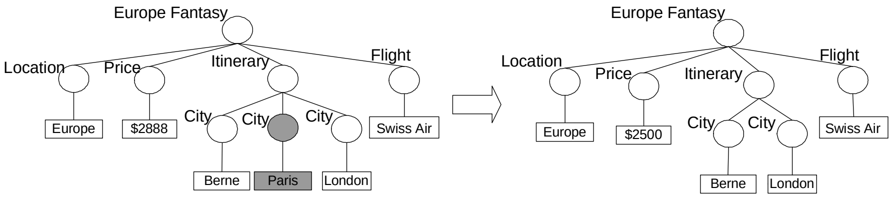


Fig. 4. The first two versions of subtree "Europe Fantasy"

will be one. Obviously, the greater the value of *DoC*, the more significantly the subtree changed.

Example 1 *Consider the first two versions of the subtree rooted at "Europe Fantasy" in the motivating example, which is redrawn in Figure 4. Let* t_1 *be the subtree //Tours/EuropeFantasy. Then, DoC(*t_1*, 1, 2)= 1/8=0.125 as there are 8 unique nodes in the two versions of the subtree and only one node is deleted. Let* t_2 *be the subtree //Tours/EuropeFantasy/Itinerary, DoC(*t_2*, 1, 2)=1/4=0.25. That is, the deletion of a node "City" changed the structure of* t_2 *more significantly than the structure of* t_1*.*

After defining the *DoC* for each subtree in a pair of successive historical versions, a *structural delta database* (denoted as *SDDB*) can be generated from a sequence of historical versions. Each tuple of a *SDDB* is a triplet, <*DID*, *SID*, *DoC*>, where *DID* is the identifier of a *delta*, which is a comparison of two successive historical versions, *SID* denotes the identifier of a changed subtree and *DoC* records the *degree of change* for the subtree in this *delta*. For example, the *SDDB* generated from the five historical versions in Figure 1 and Figure 2 is shown in Table 1. The first entry means that the *degree of change* of the subtree //Tours/EuropeFantasy from version *v1* to version *v2* is 0.13.

3.2.2 Frequency of Change

After measuring how significantly a subtree changed in two versions, we now measure how frequently a subtree changed in a sequence of historical versions. Clearly, for an individual subtree, its *frequency of change* (denoted as *FoC*) can be defined as the fraction of the *deltas* in which the subtree changed. For a set of subtrees, its *FoC* can be similarly defined as the fraction of the *deltas* in which all subtrees in the set changed.

Definition 2 [***Frequency of Change***] *Let* $<T^1, T^2, \ldots T^n>$ *be a sequence of n historical versions of an XML tree structure T. Let* Δ_i *be the set of subtrees that changed between* T^i *and* T^{i+1}*. Then,* $<\Delta_1, \Delta_2, \ldots \Delta_{n-1}>$ *is the sequence of changed subtrees in each pair of successive versions. Let S be a set of subtrees,* $S=\{t_1, t_2, \ldots t_m\}$*, where* $\forall j \in [1, m]$*,* $\exists i \in [1, n-1]$ *s.t.* $t_j \in \Delta_i$*. The FoC of*

Table 1
Structural Delta Database

DID	SID	DoC
1	//Tours/EuropeFantasy	0.13
1	//Tours/EuropeFantasy/Itinerary	0.25
1	//Tours/EuropeMarvel	0.22
1	/Tours/EuropeMarvel/Itinerary	0.4
1	...	...
2	//Tours/EuropeFantasy	0.13
2	//Tours/EuropeFantasy/Itinerary	0.25
2	//Tours/EuropeMarvel	0.22
2	//Tours/EuropeMarvel/Itinerary	0.4
2	...	...
3	//Tours/EuropeMarvel	0.11
3	//Tours/EuropeMarvel/Itinerary	0.2
3	...	...
4	//Tours/EuropeFantasy	0.13
4	//Tours/EuropeFantasy/Itinerary	0.25
4	//Tours/EuropeMarvel	0.11
4	//Tours/EuropeMarvel/Itinerary	0.2
4	...	...

the set S is:

$$FoC(S) = \frac{\sum_{i=1}^{n-1} V_i}{n-1}$$

$$\textit{where } V_i = \prod_{j=1}^{m} V_{j_i} \textit{ and } V_{j_i} = \begin{cases} 1, & \textit{if } DoC(t_j, i, i+1) \neq 0 \\ 0, & \textit{if } DoC(t_j, i, i+1) = 0 \end{cases} \quad 1 \leq j \leq m \qquad \square$$

The value of *FoC* also ranges from zero to one. When subtrees in a set never change together, *FoC* of the set will be zero. When subtrees in a set change together in every delta, *FoC* of the set will be one. The greater the value of *FoC*, the more frequently the set of subtrees changed together.

Example 2 *Consider the SDDB in Table 1. Let S be a set of two subtrees, S={ //Tours/EuropeFantasy/Itinerary, //Tours/EuropeMarvel/Itinerary }, FoC(S) = 0.75 as the two subtrees changed together for three times in four deltas.*

3.2.3 Weight

As we mentioned in the Section 1, in order to make a *FRACTURE* capture nontrivial knowledge, we require that subtrees of a *FRACTURE* should not only frequently change together but also frequently change significantly when they change together. Hence, we define the metric *Weight* to measure how frequently subtrees in a set change significantly when they change together. Basically, *Weight* of a set of subtrees can be defined as the ratio of the number of *deltas* where all subtrees in the set changed significantly (compared with some user-defined minimum *DoC*) to the number of *deltas* where all subtrees in the set changed together.

Definition 3 [***Weight***] *Let* $<T^1, T^2, \ldots T^n>$ *be a sequence of n historical versions of an XML tree structure. Let* $<\Delta_1, \Delta_2, \ldots \Delta_{n-1}>$ *be the sequence of changed subtrees in each pair of successive historical versions. Let S be a set of* *subtrees, S=*$\{t_1,t_2,...t_m\}$*, where* $\forall\ j \in$ *[1, m],* $\exists\ i \in$ *[1, n-1] s.t.* $t_j \in \Delta_i$*. Given a user-defined minimum DoC* α*, we define the Weight of the set of subtrees is:*

$$Weight(S) = \frac{\sum_{i=1}^{n-1} D_i}{(n-1) * FoC(S)}$$

$$where\ D_i = \prod_{j=1}^{m} D_{j_i}\ and\ D_{j_i} = \begin{cases} 1, & if\ DoC(t_j, i, i+1) \geq \alpha \\ 0, & otherwise \end{cases} \qquad 1 \leq j \leq m \qquad \square$$

Therefore, if all subtrees in a set change significantly every time when they change together, then the *Weight* of the set will be one; if subtrees in a set never change significantly when they change together, then the *Weight* of the set will be zero.

Example 3 *Suppose the user-defined minimum DoC* α *is 0.25. Let* S_1 *be a set of two subtrees,* S_1 *= { //Tours/EuropeFantasy/Itinerary, //Tours/Europe Marvel/Itinerary }. Then, Weight(*S_1*) = 2/3 = 0.66 because the two subtrees change together in three deltas, while in two deltas both of their DoCs are greater than* α*. Let* S_2 *be another set of two subtrees,* S_2 *= { //Tours/Europe Fantasy, //Tours/EuropeMarvel }, which are ancestors of the two subtrees in* S_1 *respectively. Weight(*S_2*) is zero as the two subtrees never change significantly together.*

3.3 FRACTURE

Based on the above discussion, given a sequence of historical versions of an XML document, FRequently And Concurrently muTating substructUREs (denoted as *FRACTURE*) can be identified by the two metrics, *FoC* and *Weight*, as follows.

Definition 4 [***FRACTURE***] *Let* $<T^1,T^2,...T^n>$ *be a sequence of historical versions of an XML tree structure. Let* $<\Delta_1,\Delta_2,...\Delta_{n-1}>$ *be the sequence of changed subtrees. Let S be a set of subtrees, S=*$\{t_1,t_2,...t_m\}$*, where* $\forall\ j \in$ *[1, m],* $\exists\ i \in$ *[1, n-1] s.t.* $t_j \in \Delta_i$*. Given the user-defined minimum DoC* α*, minimum FoC* β *and minimum Weight* γ*, S is a FRACTURE if it satisfies the two conditions: 1) FoC(S)*$\geq \beta$*, 2) Weight(S)* $\geq \gamma$*.* ☐

According to the definitions of *FoC* and *Weight*, the semantics of a *FRACTURE* can be understood as a set of XML subtrees that not only frequently change together but also frequently change significantly when they change together in a sequence of historical versions of an XML document.

Example 4 *Consider the two subtree set* S_1 *and* S_2 *in Example 3 again. Suppose the user-defined* α *is 0.25, both the user-defined* β *and* γ *are 0.5.* S_1 *is a*

FRACTURE because FoC(S_1)=0.75 $\geq \beta$ and Weight(S_1)=0.66 $\geq \gamma$. Although FoC(S_2)=0.75 $\geq \beta$, S_2 is not a FRACTURE because Weight(S_2)=0 $< \gamma$.

Hence, by requiring the subtrees in a *FRACTURE* not only frequently change together but also frequently change significantly, we can prune the subtree sets carrying trivial knowledge from being *FRACTURE*s. However, we still observed some redundancy existing in some specific pair of subtree sets. That is, in such a pair, if one of the subtree sets is a *FRACTURE*, the other must be a *FRACTURE* as well. The two subtree sets in such a specific pair has a subsumption relationship, which is defined as follows.

Definition 5 [***Subsumption***] *Given two subtree sets S and S', where S' = $S \cup \{t_1, t_2, ..., t_n\}$ and $S \cap \{t_1, t_2, ..., t_n\} = \emptyset$. If $\forall$ i ($1 \leq i \leq n$), $\exists$ $t_j \in S$ s.t. $t_j \prec t_i$, we say S is subsumed by S', or S' subsumes S, denoted as S $\prec$ S'.* □

Example 5 *Consider the motivating example again. Let S be a set of two subtrees, S = { //Tours/EuropeFantasy/Itinerary, //Tours/EuropeMarvel/ Itinerary }, and S' be a set of three subtrees, S' = {//Tours/EuropeFantasy/ Itinerary, //Tours/EuropeMarvel/Itinerary, //Tours/ EuropeFantasy}. Then S is subsumed by S', S $\prec$ S', because subtree //Tours/EuropeFantasy is an ancestor of subtree //Tours/EuropeFantasy/Itinerary.*

Then we have the following lemma between a pair of subtree sets that have the subsumption relationship.

Lemma 1 *Given two subtree sets S and S' s.t. S $\prec$ S'. If S' is a FRACTURE, S is a FRACTURE as well.*

Proof. Let $|\Delta|$ be the total number of *delta*s, $|\Delta_c(S)|$ be the number of *delta*s in which subtrees in S changed and $|\Delta_s(S)|$ be the number of *delta*s in which subtrees in S changed significantly. Suppose the user-defined minimum *FoC* is β and minimum *Weight* is γ. Since $S \prec$ *S'*, $|\Delta_c(S)|$=$|\Delta_c(S')|$, $FoC(S)$ = $\frac{|\Delta_c(S)|}{|\Delta|}$ = $FoC(S')$ = $\frac{|\Delta_c(S')|}{|\Delta|} \geq \beta$. Since $S \prec$ *S'*, then $S \subset$ *S'*, $|\Delta_s(S)| \geq |\Delta_s(S')|$. Hence, $Weight(S) = \frac{|\Delta_s(S)|}{|\Delta_c(S)|} \geq Weight(S') = \frac{|\Delta_s(S')|}{|\Delta_c(S')|} \geq \gamma$. Then S is also a *FRACTURE* and we have the lemma. ▮

According to the Lemma 1, if a subtree set is a *FRACTURE*, we can infer directly that all its subsumed subsets are *FRACTURE*s as well. In other words, a *FRACTURE* can be represented by another *FRACTURE* that subsumes it. Then the notion of *maximal FRACTURE* can be defined as follows.

Definition 6 [***maximal FRACTURE***] *A set of subtrees is a maximal FRACTURE, if it is a FRACTURE, it is not subsumed by any other FRACTURE.* □

Example 6 *Consider the two subtree sets S and S' in Example 5 again. If both S and S' are FRACTURE, S is not a maximal FRACTURE since it is* *subsumed by S'.*

Obviously, the set of *maximal FRACTURE*s is a tightened set of the complete set of *FRACTURE*s, $\{$*maximal FRACTURE*$\} \subseteq \{$*FRACTURE*$\}$, and the complete set of *FRACTURE*s can be inferred from the set of *maximal FRACTURE*s.

3.4 Problem Definition

The problem of *FRACTURE* mining and *maximal FRACTURE* mining can be formally stated as follows: Let $<T^1, T^2, \ldots\ T^n>$ be a sequence of historical versions of an XML tree structure. Let $<\Delta_1,\ \Delta_2,\ \ldots, \Delta_{n-1}>$ be the sequence of structural deltas in terms of changed subtrees. A **Structural Delta DataBase** *SDDB* can be constructed from the sequence of *delta*s, where each tuple <*DID*, *SID*, *DoC*> comprises of a *delta* identifier, a subtree identifier and a *degree of change* for the subtree in the *delta*. Let $S=\{t_1, t_2, \ldots t_m\}$ be the set of changed subtrees such that each $\Delta_i \subseteq S$ $(1 \leq \Delta_i \leq n\text{-}1)$. Given an *SDDB*, a *DoC* threshold α, an *FoC* threshold β and a *Weight* threshold γ, a subtree set $X \subseteq S$ is a ***FRACTURE*** if $FoC(X) \geq \beta$ and $Weight(X) \geq \gamma$. A subtree set $Y \subseteq S$ is a ***maximal FRACTURE*** if it is a *FRACTURE* and it does not subsumed by any other *FRACTURE*. The **problem of *FRACTURE* mining** is to find the set of all *FRACTURE*s and the **problem of *maximal FRACTURE* mining** is to find the set of all *maximal FRACTURE*s.

4 Algorithms

In this section, we present the procedure of mining *FRACTURE*s and *maximal FRACTURE*s. Given a sequence of historical versions of an XML document, two phases are involved in the mining procedure.

- **Phase I: *SDDB* construction**. This phase takes the sequence of historical versions of an XML document as input and generates the *Structural Delta DataBase* (*SDDB*). Since existing change detection systems [23][8] can detect all change operations resulting in structural changes (*insert* and *delete*), changed subtrees in each pair of successive versions can be identified directly. That is, if a node has any inserted or deleted descendants, then it is the root of a changed subtree. The *degree of change* for this subtree can be calculated immediately according to the definition of *DoC*.
- **Phase II: *FRACTURE* and *maximal FRACTURE* mining**. In this phase, we mine the set of *FRACTURE*s or *maximal FRACTURE*s with the input of the constructed *SDDB* and user-defined thresholds of *DoC*, *FoC* and *Weight*.

Since the first phase can be handled in a straightforward way based on the known conditions, subsequent discussion will be focused on the *Phase II* to mine the *FRACTURE*s and *maximal FRACTURE*s.

4.1 FRACTURE Mining

We developed two algorithms, *Apriori-FRACTURE* and *FPG-FRACTURE*, to mine the set of *FRACTURE*s. *Apriori-FRACTURE* is an apriori-like algorithm that searches the set of *FRACTURE*s with the *level-wise* strategy; while *FPG-FRACTURE* is developed from the well known algorithm FP-growth [10] which employs the *divide-and-conquer* strategy.

4.1.1 Apriori-FRACTURE

The basic idea of *Apriori-FRACTURE* is that all nonempty subsets of a subtree set satisfying the threshold of *FoC* satisfy the threshold as well. That is, the property of "downward closure" holds with respect to the metric *FoC*.

Property 1 *Given a structural delta database SDDB and user-defined minimum FoC* β*, if a subtree set S' satisfies the threshold, FoC(S')* $\geq \beta$*, for* $\forall$ *S* $\subseteq$ *S', FoC(S)* $\geq \beta$*.*

Unfortunately, the "downward closure" property does not hold with respect to the metric *Weight*. That is, even if a subtree set satisfies the user-defined threshold for *Weight*, it is not necessary that all of its subsets satisfy the threshold as well (this can be simply obtained from the definition of *Weight*). Hence, only the metric *FoC*, rather than the metric *Weight*, can be utilized to prune candidates.

According to Property 1, candidate *FRACTURE*s can be generated in the similar way as *Apriori* [2]. We call a subtree set containing *k* subtrees as a *k-subtree-set*. Then two *k-subtree-sets* that satisfy the threshold of *FoC* and share a prefix of *k-1* subtrees can be joined to generate a candidate *(k+1)-subtree-set*. For each generated candidate set, we need to check not only its *FoC* but also its *Weight*. If both the *FoC* and the *Weight* of the candidate set satisfy the respective thresholds, then it is a *FRACTURE*. If only the *FoC* of the candidate set satisfies the threshold, then the candidate set will be reserved for generating candidate sets in the next round. If neither the *FoC* nor the *Weight* of the candidate set satisfies the respective thresholds, the candidate set will be discarded. Note that Apriori [2] generates candidate patterns of the next round only from the frequent patterns discovered in this round. By contrast, we need to generate candidate subtree sets not only from the *FRACTURE*s discovered in the round but also the candidate sets satisfying the threshold of *FoC*. The reason is that even if a subtree set does not satisfy the threshold of *Weight*, it is possible that some of its supersets satisfy the threshold.

In order to further prune the search space, we utilize the following lemma which is based on the product of *FoC* and *Weight*.

Lemma 2 *Given a structural delta database SDDB, a user-defined minimum*

FoC β *and a minimum Weight* γ*, if a subtree set S satisfies the condition that FoC(S)* $\times$ *Weight(S)* $< \beta \times \gamma$*, then 1) S is not a FRACTURE; 2) any superset of S is not a FRACTURE as well.*

Proof. The first conclusion is obvious. If S is *FRACTURE*, then $FoC(S) \geq \beta$ and $Weight(S) \geq \gamma$. Thus, *FoC(S)* $\times$ *Weight(S)* $\geq \beta \times \gamma$, which contradicts the condition. We then prove the second conclusion. Let $|\Delta|$ be the total number of *delta*s in *SDDB*. Let $|\Delta_s(S)|$ be the number of *delta*s in which subtrees in S changed significantly. According to the definition of *FoC* and *Weight*, *FoC(S)* $\times$ *Weight(S)*$= \frac{|\Delta_s(S)|}{|\Delta|}$. Let *S'* be a subtree set *s.t.* *S'* $\supseteq S$, $|\Delta_s(S')| \leq |\Delta_s(S)|$. Then *FoC(S')* $\times$ *Weight(S')*$=\frac{|\Delta_G(S')|}{|\Delta|} \leq \frac{|\Delta_G(S)|}{|\Delta|}$ =*FoC(S)* $\times$ *Weight(S)* $\leq \beta \times \gamma$. Thus, *S'* is not a *FRACTURE*. ∎

Therefore, we do not need to generate candidate *(k+1)-subtree-set*s by joining all *k-subtree-set*s that satisfy the threshold of *FoC*. Given a *k-subtree-set S*, it will be used to generate candidate *(k+1)-subtree-sets* only if not only its *FoC* is no less than β, but also the product of $FoC(S) \times Weight(S)$ is no less than $\beta \times \gamma$. The algorithm of *Apriori-FRACTURE* is shown in Figure 5. We scan the *SDDB* for the first time to find the set of individual subtrees, Q_1, which satisfy the threshold of *FoC* and the condition stated in Lemma 2. The function *GenCandidatePatterns* is called to generate candidate *2-subtree-set*s C_2 from Q_1. For each candidate set, we scan the *SDDB* again to compute its *FoC* and *Weight*. Then, we find the *FRACTURE*s and the set of subtree sets, Q_2, which will be used to generate candidate sets in the next round. The algorithm iteratively generates the candidate sets and finds *FRACTURE*s until the set of Q_{k-1} is empty.

Theorem 1 *The algorithm Apriori-FRACTURE discovers the complete set of FRACTUREs.*

The completeness of *Apriori-FRACTURE* follows from the Property 1 and the Lemma 2.

Theorem 2 *The complexity of Apriori-FRACTURE is* $O(\Sigma_k\{ k \cdot |Q_{k-1}|^3, m \cdot |\Delta| \cdot |C_k|\})$*, where m is the cost of checking whether all subtrees in* c_k *changed (significantly) in each delta.*

Proof. Consider the function *GenCandidatePatterns*. The complexity of examining each pair of sets in Q_{k-1} is $|Q_{k-1}|^2$. For each generated candidate set, we need to check whether it has k subsets in Q_{k-1}, which is of complexity $k \cdot |Q_{k-1}|$. Hence, the complexity of *GenCandidatePatterns* is $k \cdot |Q_{k-1}|^3$. Since m is the cost of checking whether all subtrees in c_k changed (significantly) in each *delta* (basically, m depends on the length of each *delta* and the number of candidate sets in the *delta*). Then, the complexity of *Apriori-FRACTURE* from line 5 is $m \cdot |\Delta| \cdot |C_k|$. Therefore, the total complexity of *Apriori-FRACTURE* is $O(\Sigma_k\{k \cdot |Q_{k-1}|^3, m \cdot |\Delta| \cdot |C_k|\})$. ∎

(a) Apriori-FRACTURE

```
Input: SDDB $\Delta$, thresholds $\alpha$, $\beta$ and $\gamma$
Output: The set of FRACTUREs $P$
Description:
 1: $Q_1$ = all individual subtrees with $FoC \geq \beta$ && $(FoC \times Weight) \geq (\beta \times \gamma)$
 2: $P_1$ = all individual subtrees with $FoC \geq \beta$ and $Weight \geq \gamma$
 3: for (k=2; $Q_{k-1} \neq \emptyset$; k++) do
 4:   $C_k$ = GenCandidatePatterns($Q_{k-1}$)
 5:   for (i=1; i$\leq|\Delta|$; i++) do
 6:     for each candidate pattern $c_k \in C_k$ do
 7:       if (all subtrees in $c_k$ changed in $\triangle_i$) then
 8:         $c_k.FoC\_count$++
 9:       end if
10:       if (all subtrees in $c_k$ changed significantly in $\triangle_i$)) then
11:         $c_k.Weight\_count$++
12:       end if
13:     end for
14:   end for
15:   $Q_k = \{c_k \in C_k | c_k.FoC\_count \geq (\beta \times |\triangle|)$ && $(c_k.FoC\_count \times c_k.Weight\_count) \geq (\beta \times \gamma)\}$
16:   $P_k = \{c_k \in C_k | c_k.FoC\_count \geq (\beta \times |\triangle|)$ && $(c_k.Weight\_count / c_k.FoC\_count) \geq \gamma\}$
17: end for
18: return $\bigcup_k P_k$
```

(b) GenCandidatePatterns

```
Input: The set of (k-1)-subtree-sets $Q_{k-1}$, $\beta$
Output: The set of k-subtree-sets $C_k$
Description:
 1: for each (k-1)-subtree-set $\{m_1, m_2, \ldots, m_{k-1}\} \in Q_{k-1}$ do
 2:   for each (k-1)-subtree-set $\{n_1, n_2, \ldots, n_{k-1}\} \in Q_{k-1}$ do
 3:     if $(m_1 = n_1) \wedge \ldots \wedge (m_{k-2} = n_{k-2}) \wedge (m_{k-1} < n_{k-1})$ then
 4:       $c_k = (m_1, \ldots, m_{k-2}, m_{k-1}, n_{k-1})$
 5:       if $c_k$ has any subset with $FoC < \beta$ then
 6:         remove $c_k$
 7:       else
 8:         set $c_k.FoC\_count$=0;
            set $c_k.Weight\_count$=0;
            add $c_k$ to $C_k$
 9:       end if
10:     end if
11:   end for
12: end for
```

Fig. 5. Algorithms of *Apriori-FRACTURE* and *GenCandidatePatterns*

The algorithm *Apriori-FRACTURE* can be optimized in several ways. Note that after generating the set of candidate *k-subtree-sets*, the *SDDB* is scanned to calculate the *FoC* and the *Weight* of them. However, if a candidate subtree set does not satisfy the threshold of *FoC*, then we waste resource to compute its *Weight*. Hence, we alternatively count the *FoC* and the *Weight* of candidate sets in separate rounds. In other words, we calculate only the *FoC* of the candidate *k-subtree-set*s in the *k*th round. Only if a candidate *k-subtree-set* satisfies the threshold of *FoC*, its *Weight* will be calculated in the next round, together with the calculating of the *FoC* of the candidate *(k+1)-subtree-sets*. Referring to Figure 5, when line *7* computes the *FoC* of the candidate *k-subtree-set*s, line *10* computes the *Weight* of the candidate *(k-1)-subtree-set*s. Efficiency can be improved by computing the *Weight* for a tightened set of candidate sets. However, Lemma 2 cannot be utilized to prune search space as the *Weight* of the *k-subtree-sets* is unavailable until the (*k+1*)th round. Moreover, an extra scan of database is required to calculate the *Weight* of the candidate sets generated in the last round. The detailed algorithm that integrates this optimizing strategy with *Apriori-FRACTURE* is given in the appendix. We evaluate the performance of this strategy in Section 5.

Due to the fact that when a subtree changes, all of its ancestor subtrees change as well, another optimization can be applied. For example, we can compute the *FoC* for candidate sets containing subtrees rooted at higher-level nodes in the XML tree first. If such a set does not satisfy the threshold of *FoC*, then the sets containing their descendant subtrees cannot satisfy the threshold as well. However, this strategy requires more than one scan of the database to compute the *FoC* for all candidate *k-subtree-set*s. Hence, we do not study the performance of this strategy in our experiments.

4.1.2 FPG-FRACTURE

Apriori-FRACTURE, similar to the algorithm *Apriori* [2], has the bottleneck in generating the candidate sets and scanning database for multiple times. To address the problem, we develop a *divide-and-conquer* algorithm, *FPG-FRACTURE*, which is based on the algorithm FP-growth [10].

Data Structure FP-growth constructs a special data structure *FP-tree* which contains compact information of frequent itemsets. Due to the space constraint, we briefly describe the fundamentals of *FP-tree*, interested readers can refer to the work [10] for the details. First of all, only frequent individual items will have nodes in the *FP-tree*. Transactions sharing common items share prefix paths in the *FP-tree*. Each node registers the number of transactions in which it occurs, together with nodes in its prefix path. Node links are used to indicate the occurrences of an item in different paths. A head table maintains the set of frequent individual items and pointers to their node links.

Obviously, an *FP-tree* can be used here directly to record the information of *FoC* for subtree sets. That is, the information of which subtrees change together in each *delta*. However, it cannot simultaneously maintain the information of *Weight* for subtree sets, such as the information of which subtrees change significantly together in each *delta*. A naive solution may be to construct two *FP-tree*s to record the information of *FoC* and *Weight* respectively and intersect the results mined from the two *FP-tree*s. Clearly, this is not space-economical. Therefore, we study how to record the information of both the *FoC* and the *Weight* of subtree sets in one *FP-tree*.

Consider that each subtree in a *delta* has two states: its *DoC* is either less than the user-defined threshold of *DoC* α or no less than α. We use a pair of identifiers to represent the two states of a subtree. Given a subtree t_i, when its *DoC* is less than α, we use an identifier -t_i to represent it in this *delta*; otherwise, we use the original identifier t_i. For example, given an *SDDB* shown in Figure 6 (a). Suppose the user-defined α is 0.15, it can be transformed into the one shown in Figure 6 (b) (for ease of exposition, we also transformed the schema of the table so that each *delta* has one tuple in the transformed table).

Now if we construct an *FP-tree* from the transformed *SDDB* by creating different nodes for subtrees with different identifiers, the information of both

DID	SID	DoC	DID	SID	DoC	DID	SID	DoC	DID	SID	DoC
1	t_1	0.2	2	t_1	0.3	3	t_1	0.22	4	t_2	0.24
1	t_2	0.05	2	t_2	0.25	3	t_2	0.25	4	t_3	0.22
1	t_3	0.1	2	t_3	0.1	3	t_4	0.3	5	t_1	0.05
1	t_4	0.25	2	t_4	0.3	3	t_5	0.05	5	t_2	0.1
1	t_5	0.3	2	t_5	0.22	4	t_1	0.1	5	t_3	0.35

(a)

DID	SIDs
1	t_1, $-t_2$, $-t_3$, t_4, t_5
2	t_1, t_2, $-t_3$, t_4, t_5
3	t_1, t_2, t_4, $-t_5$
4	$-t_1$, t_2, t_3
5	$-t_1$, $-t_2$, t_3

(b)

Fig. 6. Transforming SDDB

the *FoC* and the *Weight* of subtree sets can be reserved. We call the resulting tree *Signed-FPtree*. The method of constructing a *Signed-FPtree* is similar to the construction of an *FP-tree*. The key difference between the *Signed-FPtree* and the original *FP-tree* is that in a *Signed-FPtree*, node links should connect nodes with identifiers as t_i and -t_i since they actually represent the same changed subtree.

When deciding which subtrees will have nodes in the *Signed-FPtree*, we recall the Property 1 and the Lemma 2. Thus, given the user-defined threshold of *FoC* β and threshold of *Weight* γ, a subtree t_i will be constructed in the *Sighed-FPtree* if $FoC(t_i) \geq \beta$ and $(FoC(t_i) \times Weight(t_i)) \geq (\beta \times \gamma)$. For example, suppose the β is 0.4 and the γ is 0.5. The *Signed-FPtree* constructed from Figure 6 (b) is presented in Figure 7 (a). The algorithm of constructing a *Signed-FPtree* is given in Figure 8 (a). The completeness of the *Signed-FPtree* can be justified by the following theorem.

Theorem 3 *Given an SDDB, a threshold of FoC* β *and a threshold of Weight* γ*, the constructed Signed-FPtree contains the complete information of SDDB in relevance to FRACTURE mining.*

Proof. According to the construction process of the *Signed-FPtree*, each *delta* in the *SDDB* is mapped to one path in the *Signed-FPtree*. The two states of a subtree in each *delta* is reserved by using nodes with different identifiers. Hence, the information of *FRACTURE*s in each *delta* is completely stored in the *Signed-FPtree*. ∎

Mining Algorithm We now explain how to mine the set of *FRACTURE*s from the *Signed-FPtree*. The algorithm of *FPG-FRACTURE* is shown in Figure 8 (b). The critical differences between *FPG-FRACTURE* and the original *FP-growth* are the way we calculate the *FoC* and the *Weight* for subtree sets and the way we construct the *conditional Signed-FPtree*. In the following, we illustrate the algorithm and the differences with an example. Consider the last subtree, t_5, in header table of the *Signed-FPtree* in Figure 7 (Line 10). The *FoC(t_5)* is 3/5=0.6 since the total number of occurrences of t_5 and -t_5 in the *Signed-FPtree* is three. The *Weight(t_5)* is 2/3=0.66 since t_5 occurs twice. Hence, $\{t_5\}$ is a *FRACTURE* (Line 11-14). There are three paths related to t_5: <*t_1:1, -t_2:1, -t_3:1, t_4:1, t_5:1*>, <*t_1:1, t_2:1, -t_3:1, t_4:1, t_5:1*> and <*t_1:1, t_2:1,*

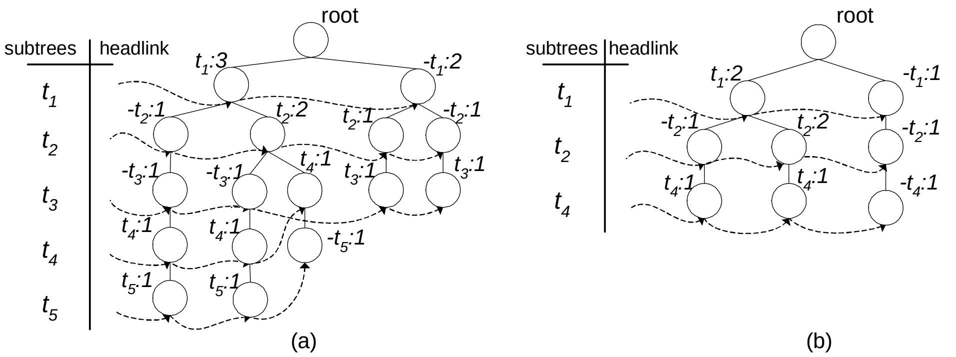


Fig. 7. Signed-FPtree

t_4:1, -t_5:1> (the number after colon indicates the number of *delta*s in which the subtree changed together with t_5). In order to mine *FRACTURE*s related to t_5, we need to construct its *conditional Signed-FPtree*. Since the β is 0.4 and the γ is 0.5, subtree t_3 should not be included in t_5's *conditional Signed-FPtree* as $(FoC(t_3)\times Weight(t_3)) = 0 <(\beta\times\gamma)$. Hence, we construct t_5's *conditional Signed-FPtree* from the following three prefix paths:*<t_1:1, -t_2:1, t_4:1>*, *<t_1:1, t_2:1, t_4:1>* and *<t_1:1, t_2:1, t_4:1>* (Line 15). Note that in the third prefix path, subtree t_5 occurs as -t_5, which means subtree t_5 did not change significantly with the subtrees t_1, t_2 and t_4 in this *delta*. To record this fact, we need to replace the identifiers of the three subtrees with -t_1, -t_2 and -t_4. We shall justify the correctness of this operation in the Lemma 3 in the below. Then the *conditional Signed-FPtree* of t_5 is shown in Figure 7 (b). Mining from it (Line 16-17), we firstly generate the pattern $\{t_4, t_5\}$. Considering both occurrences of t_4 and -t_4 in Figure 7 (b), the *FoC* of $\{t_4, t_5\}$ is 0.6. Considering the occurrences of t_4 only, its *Weight* is 0.66. Then the pattern $\{t_4, t_5\}$ is a *FRACTURE*. Other *FRACTURE*s can be mined similarly.

Lemma 3 *Given a FRACTURE $S=\{t_1, t_2, ..., t_n\}$, where t_n is the last subtree that was discovered in the S, and a (conditional) Signed-FPtree A from which S is discovered, when constructing conditional Signed-FPtree B from A to mine FRACTUREs related to S, any subtree occurs as t_i should be replaced with -t_i on paths where subtree t_n occurs as -t_n. The replacement does not affect the correctness when recursively constructing conditional Signed-FPtrees from B.*

Proof. On the paths in A where t_n occurs as -t_n, any subtree t_i occurring as t_i did not change significantly together with t_n. Hence, replacing t_i with -t_i records the correct information of the *Weight* of $S \cup \{t_i\}$ without affecting the *FoC* of $S \cup \{t_i\}$. Suppose $\{S \cup t_{n+1}\}$ is the *FRACTURE* mined from the *conditional Signed-FPtree B*. Then, for a *conditional Signed-FPtree C* constructed from B, where *FRACTURE*s related to $\{S \cup t_{n+1}\}$ will be mined, the replacement does not affect the correctness of C because of the following reason. If t_i did not change significantly together with S in this *delta*, it also did not change significantly together with $\{S \cup t_{n+1}\}$. Thus, we have the

(a) Signed-FPtree Construction

```
Input: A transformed SDDB $\Delta'$, thresholds $\beta$, $\gamma$
Output: Constructed Signed-FPtree
Description:
scan $\Delta'$ once to find the set of $Q_1$ = all individual subtrees with $FoC \geq \beta$ && $(FoC \times Weight) \geq (\beta \times \gamma)$
Sort subtrees in $Q_1$ in descending order of their FoC as L, the list of potential FRACTUREs.
Create the root of a Signed-FPtree A, and label it as "null".
for each $\Delta_i \in \Delta'$ do
    Select subtrees in L from $\Delta_i$ and sort them according to L. Represent the list of selected and sorted subtrees in the form of $[m|M]$, where m is the first subtree and M is the remaining list. Call INS_TREE([m|M], A)
end for
function INS_TREE([m|M], A)
    if A has a child node n such that n.identifier = m.identifier then
        increment n's count by 1
    else
        create a new node n, initialize its count as 1, its parent node be A and its node-link linking with nodes with identifier as either m.identifier or -m.identifier
    end if
    if M is not empty then
        call INS_TREE(M, n)
    end if
end function
```

(b) FPG-FRACTURE

```
Input: $Signed-FPtree\ A, thresholds\ \beta, \gamma$
Output: P: A set of FRACTUREs
Description:
call FPG-FRACTURE(A, null)
function FPG-FRACTURE(A, a)
    if A contains a single path P then
        for each combination (denoted as b) of the nodes in the path P do
            generate pattern $b \cup a$ with $FoC(b \cup a)$ = minimum FoC of nodes in b and $Weight(b \cup a)$ = minimum Weight of nodes in b
            if $Weight(b \cup a) \geq \gamma$ then
                $P = P \cup (b \cup a)$
            end if
        end for
    else
        for each $t_i$ in the header of tree do
            generate pattern $b = t_i \cup a$ with $FoC(b) = FoC(t_i)$ and $Weight(b) = Weight(t_i)$
            if $Weight(b) \geq \gamma$ then
                $P = P \cup b$
            end if
            construct b's conditional Signed-FPtree $tree_b$
            if $tree_b \neq \emptyset$ then
                FPG-FRACTURE($tree_b$, b)
            end if
        end for
    end if
end function
```

Fig. 8. Algorithms of *Signed-FPtree Construction* and *FPG-FRACTURE*

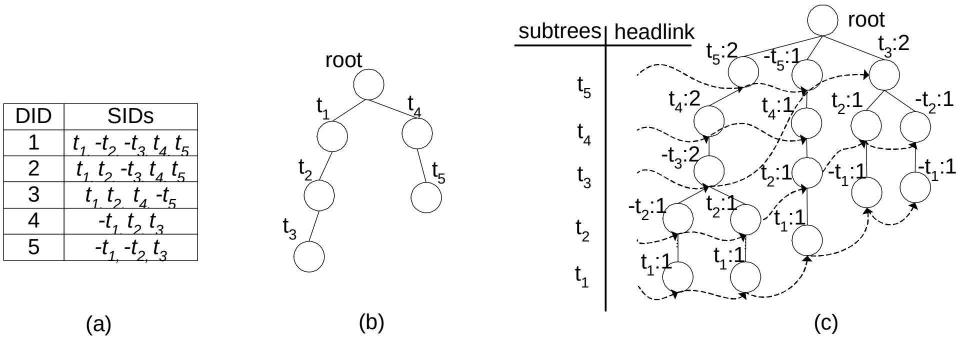
lemma. ∎

Theorem 4 *The algorithm FPG-FRACTURE discovers the set of FRACTURE completely.*

The correctness of *FPG-FRACTURE* comes from the completeness of *Signed-FPtree*, the proved correctness *FP-growth* and the Lemma 3.

4.2 Maximal FRACTURE Mining

In this subsection, we discuss the problem of mining the set of *maximal FRACTURE*s. Obviously, it is unscalable to discover all the *FRACTURE*s first and then prune the non-maximal ones. Hence, we study how to integrate the pruning techniques in the mining process. Note that the *Apriori-FRACTURE* algorithm searches the *FRACTURE*s in the breath-first way, whereas the *FPG-FRACTURE* employs the depth-first manner. Consider that if a *FRACTURE S* is subsumed by a *maximal FRACTURE S'*, then the cardinality of *S'* must



DID	SIDs
1	$t_1, -t_2, -t_3, t_4, t_5$
2	$t_1, t_2, -t_3, t_4, t_5$
3	$t_1, t_2, t_4, -t_5$
4	$-t_1, t_2, t_3$
5	$-t_1, -t_2, t_3$

Fig. 9. Ordering Subtrees in the Head Table of Signed-FPtree

be larger than that of S. In order to efficiently discover the *maximal FRACTURE*s without generating the non-maximal ones, subtree sets with larger cardinality should be checked first. Hence, a depth-first algorithm is more appropriate to be optimized to the discover the *maximal FRACTURE*s. Thus, we focus on modifying the algorithm *FPG-FRACTURE* to discover the *maximal FRACTURE*s.

4.2.1 Optimization of Subtree Ordering

As we discussed above, in order to efficiently discover the *maximal FRACTURE*s, it is ideal to generate a subtree set S' before another subtree set S if $S \prec S'$. Then, if the subtree set S' is a *FRACTURE*, we do not need to examine the subtree set S as it will not be maximal. According to the definition of the subsumption relationship, we have the following optimizing strategy.

Optimization 1 *Given an ancestor relationship between changed subtrees, a subtree set S should be generated earlier than those containing descendant subtrees of subtrees in S.*

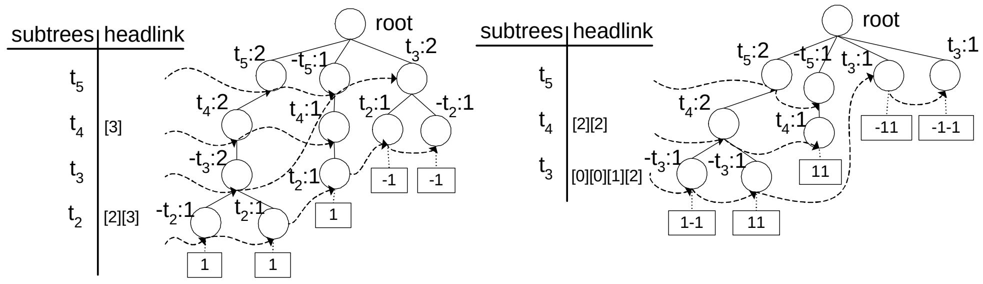
Optimization 1 guides how to order the list of subtrees in the head table of *Signed-FPtree*. For example, consider the transformed *SDDB* in Figure 6 (b), which is redrawn in Figure 9 (a). Suppose the ancestor relationship between the changed subtrees is shown in Figure 9 (b), which means t_1 is an ancestor subtree of t_2, t_2 is an ancestor subtree of t_3 and so on. We can arrange them either in descending order of the number of their ancestor subtrees (i.e. $\{t_3{:}2, t_2{:}1, t_5{:}1, t_1{:}0, t_4{:}0\}$, the number after the colon is the number of ancestor subtrees of this subtree) or in the reverse order of depth-first traversal of the ancestor relationship (i.e. $\{t_5, t_4, t_3, t_2, t_1\}$). With either ordering scheme, a subtree set will be mined earlier than those subsumed by it. For example, with the former ordering scheme, all *FRACTURE*s related to t_4 will be mined before those related to t_5 but t_4, which are probably subsumed by the *FRACTURE*s related to t_4 and then are not maximal. Before deciding which ordering policy should be employed, we examine the following property first.

Property 2 *Given two subtree sets, S and S' s.t. $S \prec S'$, the projected delta sets of S' are same as the projected delta sets of S.*

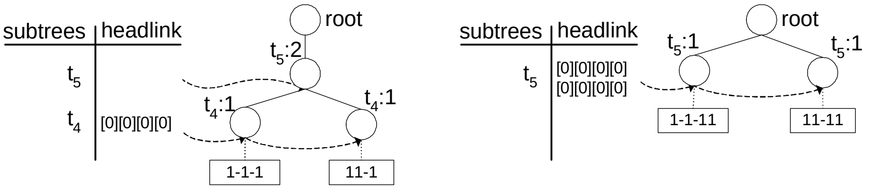
The projected *delta* sets of a subtree set S is the set of *delta*s in which subtrees in S changed. If $S \prec S'$, then $FoC(S)= FoC(S')$. That is, every *delta* containing the subtree set S contains the subtree set S' as well, vice versa.

According to the Property 2, we are presented with the opportunity to mine S and S' from the same *conditional Signed-FPtree*. Furthermore, *FRACTURE*s related to S and S' can also be mined from the same data structure. In order to utilize the same *conditional Signed-FPtree*, *FRACTURE*s related to subtree set S should be examined right after the *FRACTURE*s related to subtree set S'. Then, we need to arrange the individual subtrees in the head table in reverse order of a depth first traversal. For example, given the ancestor relationship shown in Figure 9 (b), the *Signed-FPtree* is shown in Figure 9 (c). Let S be the set $\{t_2\}$ and S' be the set $\{t_1, t_2\}$. We can examine both S and S' from the *conditional Signed-FPtree* constructed for $\{t_1\}$ as $S \prec S'$. Then, we can skip the examination of S if S' is discovered to be a *FRACTURE*. Furthermore, the *conditional Signed-FPtree* constructed for $\{t_1, t_2\}$ can be used to mine not only all *FRACTURE*s related to S' but also all *FRACTURE*s related to S.

As explained in the algorithm *FPG-FRACTURE*, labels of some nodes in *conditional Signed-FPtree* need to be shifted to record the correct information. However, this may incurs the problem that a *conditional Signed-FPtree* cannot be sharable. For example, suppose we examine the *FRACTURE*s related to $\{t_2\}$ from the *conditional Signed-FPtree* constructed for $\{t_1, t_2\}$. Consider the second path from the left in the *Signed-FPtree* shown in Figure 9 (c). Since the subtree t_1 occurs as $-t_1$, the subtree t_3 should be replaced as $-t_3$. Nevertheless, t_3 changed significantly with t_2 in the *delta*. Then, the *Weight* of $\{t_2, t_3\}$ will be computed wrongly from the *conditional Signed-FPtree* constructed for $\{t_1, t_2\}$. Therefore, we propose an alternative technique of shifting node labels. When constructing a *conditional Signed-FPtree* for a subtree set S, we append a tag to each path in the *conditional Signed-FPtree*, which indicates the states of subtrees in S. For example, Figure 10 (a) shows the *conditional Signed-FPtree* for $\{t_1\}$. Each path is appended with a tag: "1" indicates that t_1 changed significantly in this path while "-1" indicates it changed insignificantly in this path. Obviously, the modified *conditional Signed-FPtree* records the complete information of *FoC* and *Weight* of subtree sets without affecting the states of any subtree in the paths. Thus, the modified *conditional Signed-FPtree* is sharable. For example, consider the *conditional Signed-FPtree* constructed for $\{t_1, t_2\}$ which is shown in Figure 10 (b). When mining *FRACTURE*s related to $\{t_1, t_2\}$, we consider every bit in each path's tag. When mining *FRACTURE*s related to $\{t_2\}$ but $\{t_1\}$, we only consider the last bit in the tags.



(a) conditional Signed-Fptree for $\{t_1\}$ (b) conditional Signed-Fptree for $\{t_1, t_2\}$



(c) conditional Signed-Fptree for $\{t_1, t_2, t_3\}$ (d) conditional Signed-Fptree for $\{t_1, t_2, t_3, t_4\}$

Fig. 10. Mining process from t_1 in original *Signed-FPtree*

4.2.2 Optimization of Selectively Examining Subtrees

Based on the *Property 2* and the technique making the *conditional Signed-FPtree* sharable, we do not need to construct the *conditional Signed-FPtree* for a subtree set S to mine *FRACTURE*s related to it if it is subsumed by any subtree set *S'*, since *FRACTURE*s related to S can be mined from *conditional Signed-FPtree* where *FRACTURE*s related to *S'* can be mined. Then we have the following optimization.

Optimization 2 *From the head table of (conditional) Signed-FPtree, only the last subtree and subtrees which are not descendants of the subtree whose patterns are just examined in the previous round need to be examined.*

For example, from the original *Signed-FPtree* where the head table contains the subtrees in the list as $<t_5, t_4, t_3, t_2, t_1>$, we only need to construct *conditional Signed-FPtree* for subtree t_1 and t_4 since $t_2 \prec t_1$, $t_3 \prec t_1$, $t_4 \not\prec t_1$ and $t_5 \prec t_4$. The set $\{t_2\}$ can be examined at the same time when examining the set $\{t_1, t_2\}$. The set $\{t_3\}$ can be examined at the same time when examining the set $\{t_1, t_2, t_3\}$. The sets related to t_2 but t_1 can be mined from the *conditional Signed-FPtree* constructed for $\{t_1, t_2\}$ and the sets related to t_3 but t_1 or t_2 can be mined from the *conditional Signed-FPtree* constructed for $\{t_1, t_2, t_3\}$. Similarly, from the *conditional Signed-FPtree* of $\{t_1\}$ in Figure 10 (a), we only need to mine t_2 and t_4.

Recall that in the algorithm *FPG-FRACTURE*, we need to maintain a *Weight* count for each subtree in the head table of the (*conditional*) *Signed-FPtree.* The *Weight* count records the number of times the subtree changed significantly with subtrees in the set that the data structure is constructed for. Now, since we examine subtree sets related to both subtree set *S'* and subtree set S

s.t. $S \prec S'$ from the same data structure, we need to maintain more than one *Weight* count for some subtrees in the head table. For example, consider the *conditional Signed-FPtree* for $\{t_1\}$ as in Figure 10 (a), we need to maintain two *Weight* counts for t_2, one records the *Weight* count for the set $\{t_1,\ t_2\}$ and the other records the *Weight* count for the set $\{t_2\}$. Similarly, for the *conditional Signed-FPtree* constructed for $\{t_1,\ t_2\}$ as in Figure 10 (b), consider the last subtree t_3. Besides examining the set $\{t_1,\ t_2,\ t_3\}$, we examine the other three sets subsumed by it, $\{t_1,\ t_3\}$, $\{t_2,\ t_3\}$ and $\{t_3\}$. Hence, there are four *Weight* counts should be maintained for subtree t_3.

As an induction, the number of *Weight* counts that each subtree in the head table of a *conditional Signed-FPtree* should maintain can be calculated as follows.

Definition 7 [***Number of Weight Counts***] *Let* S_i=<t_1, t_2, ..., t_n> *be a list of subtrees s.t.* $\forall\ j \in[1,\ n\text{-}1]$, $t_j \prec t_{j+1}$. *Let* $FT(S_i)$ *be the first subtree and* $LT(S_i)$ *be the last subtree in* S_i. *Let* $P = S_1 \cup S_2 \cup \ldots \cup S_m$, *where* $\forall$ *i, j* $(1 \le i,\ j \le m,\ i \neq j)$, $S_i \cap S_j = \emptyset$ *and* $\forall$ $k (1 \le k \le m\text{-}1)$, $LT(S_k) \not\prec FT(S_{k+1})$. *Suppose subtree sets related to P are mined from current conditional Signed-FPtree. The number of Weight counts we need to maintain for subtree t in the head table is as follows.*

$$Number\ of\ Weight\ Counts = \begin{cases} \prod_{i=1}^{m-1} 2^{|S_i|-1} \cdot 2^{|S_m|} & if\ \ t \prec LT(S_m) \\ \prod_{i=1}^{m} 2^{|S_i|-1} & if\ \ t \not\prec LT(S_m) \end{cases} \qquad \square$$

For example, consider the *conditional Signed-FPtree* in Figure 10 (c). $P = S_1 = <t_1,\ t_2,\ t_3>$. Since $t_4 \not\prec t_3$, the number of *Weight* count we need to maintain for t_4 is 2^{3-1}=4. The four *Weight* counts record the *Weight* information for subtree sets $\{t_1,\ t_2,\ t_3,\ t_4\}$, $\{t_1,\ t_3,\ t_4\}$, $\{t_2,\ t_3,\ t_4\}$ and $\{t_3,\ t_4\}$ respectively. Consider the *conditional Signed-FPtree* in Figure 10 (d). $P = S_1 \cup S_2 = <t_1,\ t_2,\ t_3> \cup <t_4>$. Since $t_5 \prec t_4$, we need to maintain $2^{3-1} \cdot 2^1$=8 *Weight* counts for subtree t_5.

Note that, the checking for *maximal FRACTURE*s can be performed after examining each subtree in the head table of a (conditional) *Signed-FPtree*. For example, after examining the subtree t_3 in the head table of the *conditional Signed-FPtree* in Figure 10 (b), we generate four subtree sets: $\{t_1,\ t_2,\ t_3\}$, $\{t_1,\ t_3\}$, $\{t_2,\ t_3\}$ and $\{t_3\}$. Then, we only need to find *maximal FRACTURE*s from the four subtree sets rather than compare each subtree set with all previously discovered *FRACTURE*s and incoming *FRACTURE*s to verify whether it is maximal.

4.2.3 Optimization of Mining Signed-FPtree of Single Path

When the data structure contains a single path, we have the following optimization strategy.

Table 2
Parameters List

$\|\Delta\|$	Number of *delta*s	10000
S	Average size of each *delta*	20
I	Average size of subtree sets potentially satisfying minimum *FoC*	6
P	Number of subtree sets potentially satisfying minimum *FoC*	2000
W	Mean value of the fraction of subtrees satisfying minimum *Weight* in a *delta*	0.75
N	Number of changed subtrees	1000
L	Average depth of each ancestor relationship	5
F	Average fanout of each ancestor relationship	5

Optimization 3 *If the (conditional) Signed-FPtree contains a single path, maximal FRACTUREs can be generated directly from the subtrees in the head table which are 1) with their node identifier as* t_i *rather than* -t_i *in the path and 2) either last subtree or not descendant of the subtree mined in the previous round.*

For a (*conditional*) *Signed-FPtree* with a single path, if a subtree occurs with the identifier -t_i, every set related to this subtree will have its *Weight* be zero. Thus, we have the first condition. The second condition is similar to Optimization 2.

According to the *FPG-FRACTURE* algorithm in Figure 8 (b), the three optimization techniques can be employed in Line 15, Line 11 and Line 2 respectively. The detailed algorithm is given in the appendix.

5 Experimental Results

In this section, we first evaluate the performance of the developed algorithms for mining *FRACTURE*s and *maximal FRACTURE*s by conducting experiments over the synthetically generated XML structural deltas in Section 4.1. Then, we examine the novel knowledge that can be discovered by *FRACTURE*s with experiments on real-life datasets in Section 4.2. Algorithms are implemented in Java language. All experiments are conducted on a Pentium IV 2.8GHz PC with 512 MB memory. The operating system is Windows 2000 professional.

5.1 Experiments on Synthetic Datasets

We first describe the process of generating synthetic structural deltas of XML documents. Then, we study the performance of the algorithms for mining *FRACTURE*s and *maximal FRACTURE*s respectively.

5.1.1 Datasets

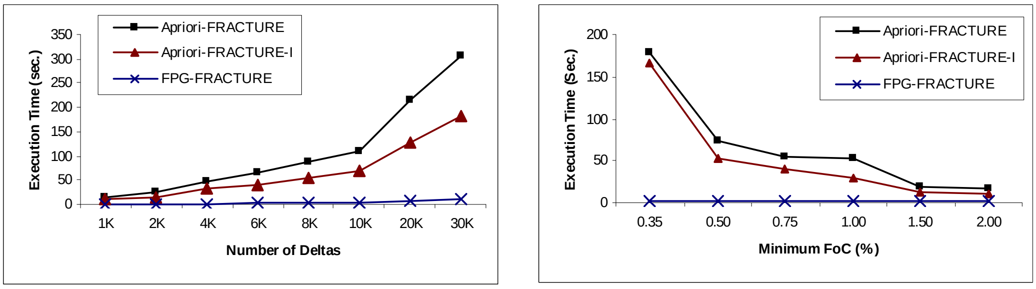
In order to evaluate the algorithms of mining *FRACTURE*s and *maximal FRACTURE*s, an *SDDB* is required. We implemented a structural delta generator by extending the one that is used to generate transaction datasets in [2]. Parameters of the synthetic structural delta generating process are shown in Table 2, with default values in the third column. Four steps are involved in the process of generating synthetic structural deltas.

- Organizing all N subtrees into ancestor relationships with the given average depth L and average fanout F.
- Generating subtree sets which potentially satisfy minimum *FoC* β.
- Picking subtrees from these patterns, together with all their ancestor subtrees, to form every *delta*.
- Assign *DoC* to subtrees in each *delta*. In each *delta*, the number of subtrees whose *DoC* is no less than the minimum *DoC* is picked from a Poisson distribution with a specified mean value W. In all experiments, we set the minimum *DoC* α as 0.15.

5.1.2 Methodology & Results for Algorithms of FRACTURE mining

In evaluating the algorithms of mining *FRACTURE*s, we carried out four experiments for algorithms: *Apriori-FRACTURE*, *Apriori-FRACTURE-I* (optimized *Apriori-FRACTURE*) and *FPG-FRACTURE*.

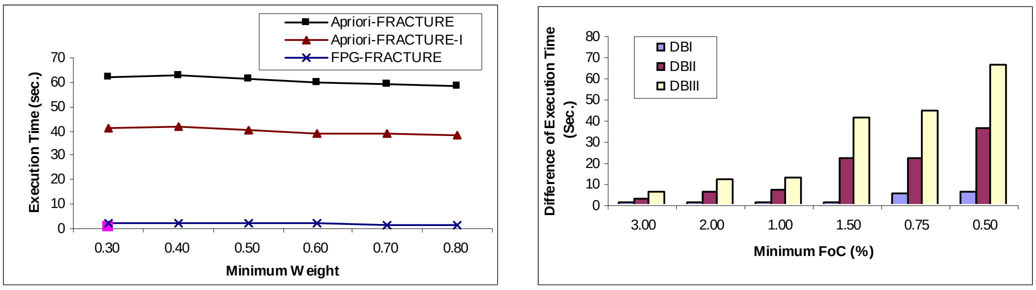
- Scalability Study: We test the scale-up features of all the three algorithms against the number of *delta*s, which is varied from 1K to 30K. The user-defined thresholds for *FoC* β and *Weight* γ are set as 0.75% and 60% respectively. Figure 11 (a) shows the results of the experiment. The performance of *Apriori-FRACTURE* degrades quickly when the number of *delta*s increases while the algorithm *FPG-FRACTURE* scales well with the increasing of the number of *delta*s. The scalability of the optimized *Apriori-FRACTURE-I* is better than the algorithm *Apriori-FRACTURE*.
- Efficiency Study I: We compare the execution time of each algorithm to discover *FRACTURE*s by varying the minimum *FoC* β from 0.35% to 2%. The user-defined γ is set as 60%. The results are shown in Figure 11 (b). Again, the efficiency of the algorithm *FPG-FRACTURE* outperforms the algorithms *Apriori-FRACTURE* and *Apriori-FRACTURE-I*. And the optimized *Apriori-FRACTURE-I* is more efficient than the *Apriori-FRACTURE* although it scans database one more time than the *Apriori-FRACTURE* does.
- Efficiency Study II: We measure the execution time of each algorithm to discover *FRACTURE*s by varying the γ from 30% to 80%. The β is set as 0.75%. As shown in Figure 12 (a), the variation of the minimum *Weight* does not affect the performance of the algorithms. This is true because none of the algorithms utilize this constraint to prune search space.
- Optimizing Strategy for *Apriori-FRACTURE*: We measure the effectiveness of the optimizing strategy for *Apriori-FRACTURE* by comparing the gap between the execution time of the *Apriori-FRACTURE* and the *Apriori-FRACTURE-I* against both the number of *delta*s and minimum *FoC* β.



(a) Variation of number of deltas

(b) Variation of minimum FoC

Fig. 11. Experiment Results I



(a) Variation of minimum Weight

(b) Enhancement of optimized Apriori-FRACTURE

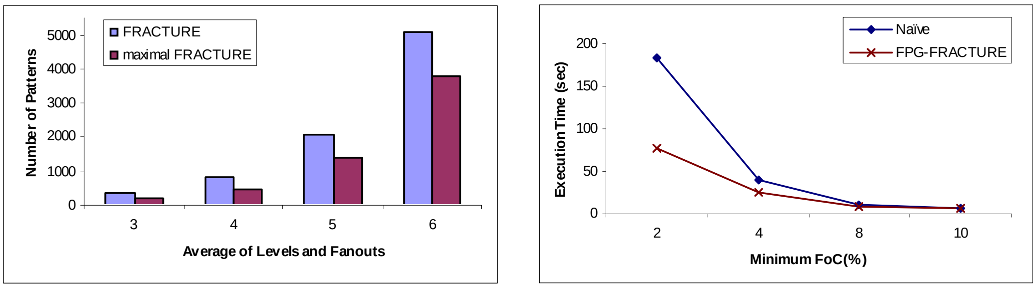
Fig. 12. Experiment Results II

Since *Apriori-FRACTURE-I* tries to gain efficiency by counting *Weight* for a tightened set of candidate sets, both the size of database and the β may affect its performance. Three sets of data are used: DBI(1K), DBII(5K) and DBIII(10K). The β ranges from 0.5% to 3%. As shown in Figure 12 (b), when the size of the dataset turns to be larger and the β turns to be smaller, where the *Apriori-FRACTURE* algorithm cannot perform well, the gap between the *Apriori-FRACTURE-I* and *Apriori-FRACTURE* increases.

5.1.3 Methodology & Results for Algorithms of maximal FRACTURE Mining

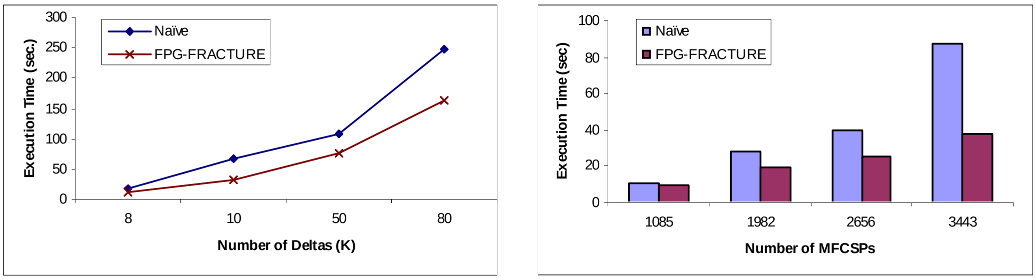
In this section, we first carried out experiments to show how the set of *maximal FRACTURE*s is more concise than the complete set of *FRACTURE*s. Subsequently, we evaluated the performance of the modified algorithm *FPG-FRACTURE* by comparing it with a naive algorithm for *maximal FRACTURE* mining. Basically, the naive algorithm finds the complete set of *FRACTURE*s first and then prune the non-maximal ones. Since the naive algorithm is really unscalable, we also optimized it slightly so that a new *FRACTURE* is only need to be compared with previously discovered *FRACTURE*s to verify whether it is maximal or not.

- Conciseness of *maximal FRACTURE*s: Firstly, we contrast the size of the set of *maximal FRACTURE*s with the size of the complete set of *FRACTURE*s by adjusting the average depth and fanout of the ancestor rela-



(a) Variation of Avg. Depth and Fanout (b) Variation of Minimum FoC

Fig. 13. Experiment Results III



(a) Variation of Number of Deltas (b) Variation of Number of *Maximal FRACTURE*

Fig. 14. Experiment Results IV

tionships. As shown in Figure 13 (a), the set of *maximal FRACTURE*s is apparently more tighter then the set of *FRACTURE*s. When the average depth and fanout of ancestor relationships are larger, more *FRACTURE*s might be subsumed by their supersets. Hence, the compression ratio turns to be greater.

- Efficiency Study: We compare the execution time of the naive algorithm and the optimized *FPG-FRACTURE*. As shown in Figure 12 (a), the threshold γ does not affect the efficiency of the mining algorithms, we conducted this experiment by varying the threshold β from 2% to 10%. As shown in Figure 13 (b), when the threshold is smaller, the optimized *FPG-FRACTURE* is more efficient. This is because when the threshold is smaller, more *FRACTURE*s will be generated. Hence, the naive algorithm needs to check more *FRACTURE*s to verify whether they are maximal or not.
- Scalability Study I: We test the scale-up features of the two algorithms against the number of *delta*s, which is varied from 8K to 80K. Figure 14 (a) shows that the optimized *FPG-FRACTURE* has the better scalability than the naive one. Moreover, when the number of *delta*s is larger, the gap between the two algorithms is greater.
- Scalability Study II: We also observe the scalability of the two algorithms with respect to the number of discovered *maximal FRACTURE*s. As presented in Figure 14 (b), when mining the same number of *maximal FRACTURE*s, the optimized *FPG-FRACTURE* is faster than the naive algorithm. Furthermore, when the size of the set of *maximal FRACTURE*s increases, the optimized *FPG-FRACTURE* scales even better.

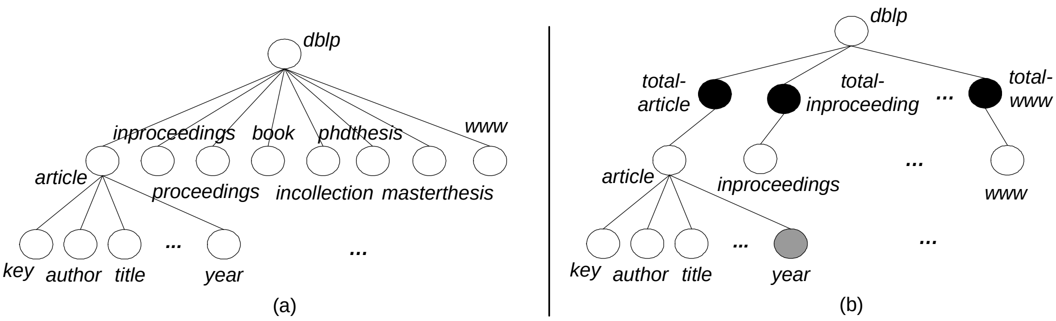


Fig. 15. DBLP DTD Structure

5.2 Experiments on Real-life Datasets

In this section, we conduct experiments on real-life data sets. Note that, since the performance of the approaches has been evaluated with the experiments on synthetic data, we focus on examining the knowledge that can be discovered by *FRACTURE*s with the experiments on real-life data sets. Two sets of real-life data, DBLP data and Web access log data, are used in these experiments. In the following, we describe the two sets of data and experiments respectively.

5.2.1 Methodology & Results on DBLP Data

The DBLP data is the bibliographic information on major computer science journals and proceedings provided by the DBLP server [17]. The basic DTD structure of the document *dblp.xml* is shown in Figure 15 (a). It can be observed that *dblp.xml* has eight distinct elements under the root: *article*, *inproceedings*, *proceedings*, *book*, *incollection*, *phdthesis*, *mastersthesis*, and *www*. With the evolution of *dblp.xml*, new instances of these elements will be added incrementally. Hence, it is hopeful to discover some *structural associations* from these elements. For example, new instances of *inproceedings* and *proceedings* may be frequently added together. In order to discover *FRACTURE*s indicating such *structural associations*, we reorganized the *dblp.xml* file in the following two steps: (1) all instances of each child element of the root are organized under a newly inserted element (black nodes in Figure 15 (b)) corresponding to the original element. The resulted DTD structure is shown in Figure 15 (b); (2) historical versions of *dblp.xml* are generated according to the element *year* (gray node in Figure 15 (b)) of the instances. For example, a resulted historical version *dblp1970.xml* contains all instances whose element *year* has a value less than or equal to "1970".

We totally generated 30 historical versions of *dblp.xml* from year 1971 to year 2000. Experiments are conducted not only on the total 30 versions but also on every 10 versions. For experiments on the whole 30 versions, we set the *minimum DoC* α, *minimum FoC* β and *minimum Weight* γ as 0.15, 0.2 and 0.4 respectively. For experiments on every 10 versions, the three thresholds are set as 0.15, 0.3 and 0.5 respectively. The results are shown in Table 3. It can be observed that discovered *FRACTURE*s contain not only individual subtrees but

Table 3
Results on DBLP Data

1971-2000	1971-1980
dblp/total-proceedings/ dblp/total-article/ dblp/total-phdthesis/ dblp/total-inproceedings/ {dblp/total-inproceedings/, dblp/total-article/}	dblp/total-proceedings/ dblp/total-inproceedings/ dblp/total-article/ {dblp/total-proceedings/, dblp/total-inproceedings/} {dblp/total-inproceedings/, dblp/total-article/}
1981-1990	1991-2000
dblp/total-proceedings dblp/total-phdthesis {dblp/total-proceedings, dblp/total-phdthesis}	dblp/total-proceedings dblp/total-phdthesis dblp/total-masterthesis dblp/total-www {dblp/total-masterthesis, dblp/total-www}

also pairs of subtrees. For example, from the *FRACTURE*s discovered from the whole 30 versions, we noticed that the subtrees *dblp/total-inproceedings* and *dblp/total-article* frequently and concurrently change together. We may infer from the *FRACTURE* that from year 1971 to 2000, new instances of *inproceedings* (conference papers) and *articles* (journal papers) are frequently added together.

Although the *FRACTUREs* discovered from the *dblp.xml* indicate *structural associations*, the *semantical associations* are not obvious. For example, it is hard to explain the *semantical association* in the *FRACTURE* {*dblp/total-masterthesis*, *dblp/total-www*} discovered from year 1991 to year 2000. Furthermore, since the depth of *dblp.xml* is small, we can only discover *FRACTUREs* from the child elements of the root. To overcome these two deficiencies, we conducted experiments on another set of real-life dataset in the next subsection.

5.2.2 Methodology & Results on Web Log Data

Recently, there are proposals [12] [11] on designing Log Markup Language (LOGML), which is XML 1.0 application, to describe the log reports of web servers. The main motivation is that although it is easy to extract simple information from web logs, it is quite challenging to mine complex structural information. In our experiments, we also represent the access logs of a web user in a single day as an XML document since both of them can be modeled as a tree structure. For example, given the access logs of a web user in Figure 16 (a), we represent it as an XML document in Figure 16 (b). Thus, the structural information of the access logs of web users can be captured by the structure of the XML documents. Then, given a sequence of historical versions of XML documents representing a web user's historical access logs, we can mine *FRACTURE*s from them to discover associated interests of the web user or associated substructures of the web site.

```
[18/Sep/2004:13:09:40 ] "GET /Elearning/images/login/dart.gif HTTP/1.1"
[18/Sep/2004:13:09:55 ] "GET /Elearning/css/search_library.css HTTP/1.1"
[18/Sep/2004:13:10:01 ] "GET /Elearning/Lesson/control.html HTTP/1.1"
[18/Sep/2004:13:10:01 ] "GET /Elearning/Lesson/lesson_frameset.html
[18/Sep/2004:13:10:04 ] "GET /Elearning/Lesson/Main_Frame.swf HTTP/1.1"
```

(a)

```
{Elearning/images/drseries, Elearning/Learning_Objects/x186}
{Elearning/imgaes/drseries, Elearning/css}
{Elearning/images/drseries, Elearning/js/lesson_builder.js}
```

(c)

```
<Elearning>
   <images>
      <login>
         <dart.gif/>
      </login>
   </image>
   <css>
      <search_library.css/>
   </css>
   <Lesson>
       <control.html/>
       <lesson_framset.html/>
       <Main_Frame.swf/>
   </Lesson>
</Elearning>
```

(b)

Fig. 16. Web Log and XML

We observed the historical access logs of a particular web user visiting an E-learning Web site [1] during September 2004. Then, we generated XML documents for her access logs in each day. Since the user may not access the Web site every day, there are totally 23 historical versions of XML documents are collected. The maximal depth of the generated XML documents is 5, we are then allowed to discover *FRACTURE*s from the subtrees rooted at different levels of the XML documents.

We conducted the experiments by varying the thresholds, *minimum DoC* α, *minimum FoC* β and *minimum Weight* γ, to find the set of meaningful *FRACTURE*s (Setting loose thresholds gets too many *FRACTURE*s while setting strict thresholds gets *FRACTURE*s containing only individual subtrees). Figure 16 (c) shows the *FRACTURE*s when setting the α, β and γ as 0.5, 0.6 and 0.7 respectively (Due to constraint space, only *FRACTURE*s containing more than one subtrees are shown). Users with certain knowledge of the Web site can infer the *semantical associations* from the results. For example, the first *FRACTURE* may indicate that the web user frequently visited the learning objects under the node “x186”, which use the different images under the node “drseries”.

6 Applications

Discovered *FRACTURE*s can be used in a wide range of applications. We enumerate some of them in this section.

Native XML Storage. Native XML storage usually views an XML document as a tree and partitions the XML tree into distinct records containing disjoint connected subtrees, such as *Natix* [13]. These distinct records are then stored in disk pages. *Natix* did not employ any particular strategy to partition an XML tree or store the records. Actually, the knowledge inferred from *FRACTURE*s can be used as a guide so that when XML document changes, the updating process can be more efficient in locating changed records.

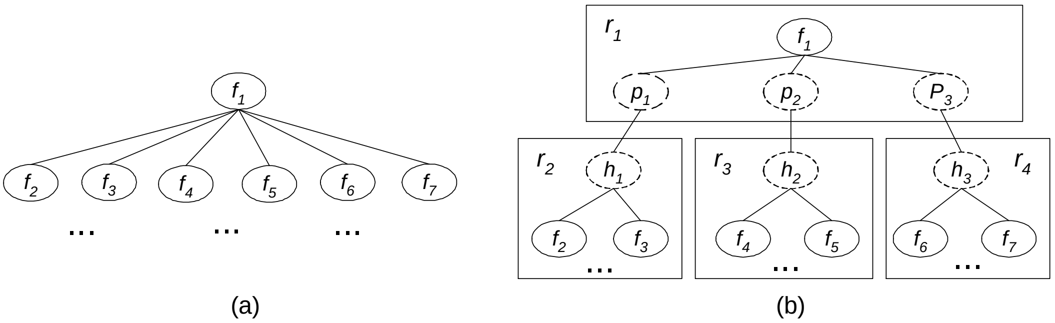


Fig. 17. Distribution of XML tree onto records

For example, given an XML tree as shown in Figure 17 (a). One possibility of *Natix* for partitioning the logical tree into four physical trees, r_1, r_2, r_3, r_4, which will be stored in disk pages is shown in Figure 17 (b). Nodes marked by dashed ovals are added to link the physical trees together. Now, suppose two subtrees rooted at nodes f_2 and f_7 are discovered as a *FRACTURE*. Based on the knowledge inferred from the *FRACTURE* that the two subtrees frequently change together, we can partition the XML tree so that the two subtrees reside in the same physical tree if they fit in a disk page (Otherwise, we can partition them into different physical trees and store them in adjacent disk pages). In subsequent versions of the XML document, the two subtrees very likely change together again as they are discovered as an *FRACTURE*. When updating the records containing the two subtrees, with the partition as in Figure 17 (b), we need to search the locations of two records r_2 and r$_4$. Nevertheless, with the strategy based on *FRACTURE*s, we only need to search the location for one record as the two subtrees are in the same disk page (or adjacent disk pages).

Approximate XML Change Detection. Given a dynamic XML document, when users are not interested in the exact changes to the document [1], *FRACTURE*s can be used to facilitate the approximate XML change detection. X-DIFF [23] is one of the XML change detection algorithms that detect changes most accurately. It detects changes to XML documents in the top-down fashion. For example, Figure 18 (a) shows two versions, t_1 and t_2, of an XML tree. After comparing the signatures of the two nodes labelled as a, we may know that the subtree rooted at node a is changed. Suppose the subtree rooted at node a and the subtree rooted at node b has been discovered as a *FRACTURE*. Then, the approximate change detection algorithm may expediently compare the signatures of the nodes labelled as b and skip comparing the subtrees rooted at nodes c and d. Certainly, there is a tradeoff between efficiency and accuracy.

As *FRACTURE*s are discovered from a sequence of historical versions of a (XML) tree, they can be useful in not only XML-related applications but also

---

[1] For example, users may want to know the rough changes before inquiring the exact changes.

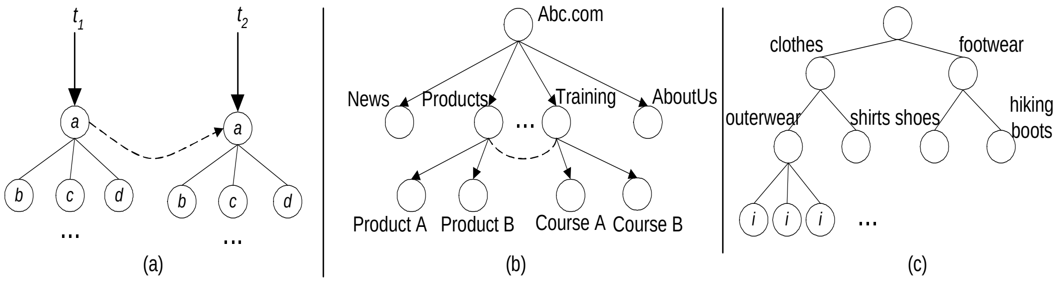


Fig. 18. Distribution of XML tree onto records

other applications where data has hierarchical structures.

Web Crawling. Pages of a particular web site can be organized as a tree according to the paths in their URL addresses. Figure 18 (b) shows an example hierarchy of pages in a web site at www.abc.com. As a web site might be updated frequently (i.e, some pages may be inserted while some pages may be deleted), *FRACTURE*s can be mined from a sequence of historical versions of the web structure likewise. Discovered *FRACTURE*s can be used by a web crawler in designing intelligent crawling strategies. Consider the example in Figure 18. Suppose the two subtrees rooted at nodes "Products" and "Training" are discovered as a *FRACTURE*. It can be inferred that pages in the two subtrees frequently change together (i.e., when some new products are released, some new training courses are added as well). Thus, a corresponding crawling strategy can be designed that once the crawler detects that pages under the "Products" change significantly, it will automatically create a new copy of pages under the "Training" because these pages very likely change as well according to the discovered *FRACTURE*.

Market Basket Analysis. As pointed out in [18], in most cases, there exists taxonomies over transaction items. For example, Figure 18 (c) shows an example taxonomy, which indicates that outwear *is-a* clothes and shoes *is-a* footwear etc. Once an item is purchased, it corresponds to the insertion of a leaf node to the node representing its category (i.e., the nodes labelled as i in Figure 18 (c)). Such a hierarchy can be updated every certain time period according to the transactions in the period. Thus, *FRACTURE*s can be mined from the sequence of its historical versions. Knowledge gathered from the discovered *FRACTURE*s can be used for market basket analysis. For example, if the subtree rooted at node "clothes" and the subtree rooted at node "shoes" are discovered as a *FRACTURE*, then it can be inferred that when the sales of items of clothes increases, the sales of items of shoes frequently increases as well. Thus, once the merchant sees a rise in the sales of clothes, he may indent more shoes if they are low in stock. Note that, traditional frequent patterns [18] fails to discover such association if the *support* of clothes or shoes does not satisfy some pre-defined threshold. While *FRACTURE*s can discover it only if the increase of *support* of clothes and shoes is significant enough.

7 Related Works

Our proposed *FRACTURE* mining system is largely influenced by several recent technologies by two major research communities in data mining. On one hand, the XML mining community has largely focused on mining frequent substructures from a collection of static XML document collection. On the other hand, the association rule mining community has paid considerable attention to designing efficient and scalable algorithms for finding frequent patterns. In this section, we compare our approach with these approaches and highlight the novelty of our work.

7.1 XML Structure Mining

Since XML documents are typically viewed as semi-structured data, they do not have rigid structure. Major work on XML structure mining focuses on discovering frequent substructures from a collection of XML documents [22] [3] [25] [20]. Wang and Liu [22] 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. Asai et al. [3] 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 obtained without generating duplicate candidate frequent trees. TreeMinerH and TreeMinerV [25] are two algorithms for mining frequent trees in a forest. As the name of the algorithm indicates, 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 [20] 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 labelled 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 algorithm to discover frequent subtrees [21] [24].

The principal character that distinguishes our study from existing XML structure mining is that we aim to discover frequent patterns in terms of changed subtrees. Specifically, we treat a comparison of two versions of an XML structure as a "transaction" and changed subtrees in the two versions as "items", whereas existing XML structure mining treats the structure of each XML document as a "transaction" and the edges, nodes or paths of each structure as "items". In addition, existing work on XML structure mining considers only snapshot structure of an XML document, whereas we consider the dynamic nature of the structures in an XML document.

7.2 Frequent Pattern Mining

There has been increasing research efforts in frequent pattern mining by the data mining community. Frequent pattern mining can be considered as a critical subproblem of the association rule mining problem. Basically, the state-of-art approaches of frequent pattern mining consists of two lines of works, for which the *Apriori* [2] algorithm and the *FP-Growth* [10] algorithm are the representatives respectively. A frequent pattern is a set of items that frequently occur together. In our research, a *FRACTURE* is a set of trees that frequently change significantly together. Thus, the notion of *FRACTURE* is similar to frequent pattern as far as the frequency of co-occurrence of "items" is concerned. However, the critical difference between our study and classical frequent pattern mining problem is that in our research, a frequent pattern is defined based on *not only* the frequency of the pattern but also the weight of the pattern. Furthermore, in classical frequent pattern mining, items are independent from each other. However, "items" have some inherent *relationship* in our study. That is, when a subtree changes, all its ancestor subtrees change as well. This feature makes our problem similar to the *generalized association rule mining* [18]. Hence, our study shares the common redundancy problem with generalized association rule mining in finding patterns of "items" with ancestor relationships. We filter the redundant patterns by capturing not only the *FoC* of a *FRACTURE* but also the *weight* of a *FRACTURE*. In addition, since each subtree is associated with the *DoC* to indicate its change degree in a *delta*, our study has some connection with the *weighted association rule mining* [19]. However, items are associated with fixed *weight* in *weighted association rule mining* whereas in our approach subtrees may have different *DoC* in different *deltas*.

7.3 Maximal Frequent Pattern Mining

Maximal frequent pattern mining is an interesting problem as it discovers a concise set of frequent patterns. MaxMiner [4] applies a breath-first strategy to mine maximal patterns. It employs the "look ahead" technique to discover longer frequent patterns first so that the shorter non-maximal frequent patterns can be skipped. Mafia [6] and GenMax [9] are two algorithms using the depth-first strategy to mine maximal patterns and incorporating a series of optimizing strategies.

Since our definition of the *maximal FRACTURE* is fundamentally different from the classical definition of maximal frequent pattern mining, our algorithms for searching the set of maximal patterns are also different from existing approaches. Essentially, we capture the ancestor relationship between changed subtrees to optimize the mining algorithm.

8 Conclusions and Future Work

This paper proposed a novel problem of frequent pattern mining called *FRACTURE* mining, which is based on changes to XML structures. Discovered *FRACTURE*s imply that some subtrees in an XML structure frequently change together. Knowledge obtained from *FRACTURE*s can be useful in applications such as XML indexing, XML clustering etc. In order to make the result patterns concise, we further defined the problem of *maximal FRACTURE* mining. Two different algorithms, *Apriori-FRACTURE* and *FPG-FRACTURE*, were designed to mine the set of *FRACTURE*s. We then modified the algorithm *FPG-FRACTURE* to handle the problem of *maximal FRACTURE* mining. Experiment results demonstrated that both algorithms can discover the complete set of *FRACTURE*s with certain efficiency and scalability, the optimizing strategies work effectively in improving the performance of the algorithms, and the modified algorithm can discover the set of *maximal FRACTURE*s efficiently. As future work, we are interested in investigating the problem of mining frequent patterns from XML content deltas and hybrid deltas.

References

[1] *http://elearn.litespeed.com.sg.*

[2] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *Proceedings of 20th International Conference on Very Large Databases*, pages 487–499, 1994.

[3] T. Asai, K. Abe, S. Kawasoe, H. Arimura, H. Sakamoto, and S. Arikawa. Efficient substructure discovery from large semi-structured data. In *Proceedings of the 2nd SIAM International Conference on Data Mining*, pages 158–174, 2002.

[4] R. J. Bayardo. Efficiently mining long patterns from databases. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data, Seattle, Washington, USA*, 1998.

[5] D. Braga, A. Campi, S. Ceri, M. Klemettinen, and P. L. Lanzi. A tool for extracting xml association rules from xml documents. In *Proceedings of the 14th IEEE International Conference on Tools with Artificial Intelligence, Washington, DC, USA*, 2002.

[6] D. Burdick, M. Calimlim, and J. Gehrke. Mafia: A maximal frequent

itemset algorithm for transactional databases. In *Proceedings of 17th Internationl Conference of Data Engineering,Heidelberg, Germany*, 2001.

[7] L. Chen, S. S. Bhowmick, and L. T. Chia. Mining association rules from structural deltas of historical xml documents. In *Proceedings of the 8th Pacific-Asia Conference on Knowledge Discovery and Data Mining 2004, Sydney, Australia*, 2004.

[8] G. Cobena, S. Abiteboul, and A. Marian. Detecting changes in xml documents. In *Proceedings of 18th International Conference on Data Engineering, San Jose, California, USA*, 2002.

[9] K. Gouda and M.J. Zaki. Efficiently mining maximal frequent itemsets. In *Proceedings of the 1st International Conference on Data Mining 2001,San Jose, California*, 2001.

[10] J. W. Han, J. Pei, and Y. W. Yin. Mining frequent patterns without candidate generation. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, 2000.

[11] M. S. Krishnamoorthy J. R. Punin and M. J. Zaki. Logml-log markup language for web usage mining. In *WEBKDD Workshop 2001: Mining Log Data Across All Customer TouchPoints (with SIGKDD01)*, 2001.

[12] M. S. Krishnamoorthy J. R. Punin and M. J. Zaki. Logml-xml language for web usage mining. In *Proceedings of the 10th International World Wide Web Conference*, 2001.

[13] C. Kanne and G. Moerkotte. Efficient storage of xml data. In *Proceedings of the 16th International Conference on Data Engineering*, 2000.

[14] M. L. Lee, L. H. Yang, W. Hsu, and X. Yang. Xclust: Clustering xml schemas for effective integration. In *Proceedings of the 11th ACM International Conference on Information and Knowledge Management, McLean, VA*, 2002.

[15] H. P. Leung, F. L. Chung, and S. C. Chan. A new sequential mining approach to xml document similarity computation. In *Proceedings of the 7th Pacific-Asia Conference on Knowledge Discovery and Data Mining, Seoul, Korea*, 2003.

[16] W. Lian, D. W. Cheung, N. Mamoulis, and S. M. Yiu. An efficient and scalable algorithm for clustering xml documents by structure. In *IEEE Transactions on Knolwedge and Data Engineering, vol.16, no.1*, 2004.

[17] UW XML Repository. Dblp computer science bibliography, *http://www.cs.washington.edu/research/xmldatasets/www/repository.html*.

[18] R. Srikant and R. Agrawal. Mining generalized association rules. In *Proceedings of the 21th International Conference on Very Large DatabBases*, pages 407–419, 1995.

[19] F. Tao, F. Murtagh, and M. Farid. Weighted association rule mining using weighted support and significance framework. In *Proceedings of the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2003.

[20] A. Termier, M.C. Rousset, and M. Sebag. Treefinder: A first step towards xml data mining. In *Proceedings of the 2nd IEEE International*

Conference on Data Mining, pages 450–457, 2002.

[21] C. Wang, M. Hong, J. Pei, H. Zhou, W. Wang, and B. Shi. Efficient pattern-growth methods for frequent tree pattern mining. In *Proceedings of the 8th Pacific-Asia Knowledge Discovery and Data Mining*, 2004.

[22] K. Wang and H. Liu. Discovering structural association of semistructured data. In *IEEE Transaction of Knowledge and Data Engineering, vol.12*, pages 353–371, 2000.

[23] Y. Wang, D. J. DeWitt, and J. Y. Cai. X-diff: An effective change detection algorithm for xml documents. In *Proceedings of the 19th International Conference on Data Engineering, India*, 2003.

[24] Y. Xiao, J. F. Yao, Z. Li, and M. H. Dunham. Efficient data mining for maximal frequent subtree. In *Proceedings of the 3rd International Conference on Data Mining*, page 379, 2003.

[25] M. J. Zaki. Efficiently mining frequent trees in a forest. In *Proceedings of the 8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 71–80, 2002.

[26] M. J. Zaki and C. C. Aggarwal. Xrules: An effective structural classifier for xml data. In *Proceedings of the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining , Washington, DC, USA*, 2003.

APPENDIX

The optimized version of *Apriori-FRACTURE* for *FRACTURE* mining and the optimized version of *FPG-FRACTURE* for *maximal FRACTURE* mining are described in the Figure A.1 (a) and (b) respectively.

(a) Optimized Apriori-FRACTURE

```
Input: $SDDB\ \Delta$, thresholds $\alpha$, $\beta$ and $\gamma$
Output: The set of FRACTUREs $P$
Description:
Q$_1$ = all individual subtrees with $FoC \geq \beta$
for (k=2; Q$_{k-1} \neq \emptyset$; k++) do
    C$_k$ = GenCandidatePatterns($Q_{k-1}$)
    for (i=1; i$\leq|\Delta|$; i++) do
        for each candidate pattern $c_k \in$ C$_k$ do
            if (all subtrees in $c_k$ changed in $\triangle_i$) then
                $c_k$.FoC_count++
            end if
        end for
        for each subtree set $q_{k-1} \in$ Q$_{k-1}$ do
            if (all subtrees in $q_{k-1}$ changed significantly in $\triangle_i$)) then
                $q_{k-1}$.Weight_count++
            end if
        end for
    end for
    P$_{k-1}$ = {$q_{k-1} \in$ Q$_{k-1}$ | ($q_{k-1}$.Weight_count / $q_{k-1}$.FoC_count) $\geq \gamma$}
    Q$_k$ = {$c_k \in$ C$_k$ | $c_k$.FoC_count$\geq(\beta \times |\triangle|)$}
end for
return $\bigcup_{k-1}$P$_{k-1}$
```

(b) Optimized FPG-FRACTURE

```
Input:
    Signed-FPtree, thresholds $\beta$,$\gamma$
Output:
    $P$: A set of maximal FRACTUREs
Description:
    call OFPG_FRACTURE(Signed-FPtree, null)
function OFPG_FRACTURE(tree, a)
    if tree contains a single path P then
        generate set $b$ based on Optimization 3
        if Weight($b \cup a$) $\geq \gamma$ && $b \cup a$ is maximal then
            $P = P \cup (b \cup a)$
        end if
    else
        for each $a_i$ in the header of tree do
            generate set $b = a_i \cup a$ and its subsumed sets based on Optimization 2
            for each subsumed set $c$ do
                if Weight($c$) $\geq \gamma$ && $c$ is maximal then
                    $P = P \cup c$
                end if
            end for
            construct $b$'s conditional Signed-FPtree tree$_b$ based on the ordering in Optimization 1
            if tree$_b \neq \emptyset$ then
                OFPG_FRACTURE(tree$_b$, $b$)
            end if
        end for
    end if
end function
```

Fig. A.1. *Optimized Apriori-FRACTURE* and *Optimized FPG-FRACTURE*