FS-Miner: Efficient and Incremental Mining of Frequent Sequence Patterns in Web logs

Maged El-Sayed, Carolina Ruiz, and Elke A. Rundensteiner
Department of Computer Science, Worcester Polytechnic Institute
Worcester, MA 01609-2280
{maged | ruiz | rundenst}@cs.wpi.edu

ABSTRACT

Mining frequent patterns is an important component of many prediction systems. One common usage in web applications is the mining of users' access behavior for the purpose of predicting and hence pre-fetching the web pages that the user is likely to visit.

In this paper we introduce an efficient strategy for discovering frequent patterns in sequence databases that requires only two scans of the database. The first scan obtains support counts for subsequences of length two. The second scan extracts potentially frequent sequences of any length and represents them as a compressed frequent sequences tree structure (FS-tree). Frequent sequence patterns are then mined from the FS-tree. Incremental and interactive mining functionalities are also facilitated by the FS-tree. As part of this work, we developed the FS-Miner, a system that discovers frequent sequences from web log files. The FS-Miner has the ability to adapt to changes in users' behavior over time, in the form of new input sequences, and to respond incrementally without the need to perform full re-computation. Our system also allows the user to change the input parameters (e.g., minimum support and desired pattern size) interactively without requiring full re-computation in most cases.

We have tested our system comparing it against two other algorithms from the literature. Our experimental results show that our system scales up linearly with the size of the input database. Furthermore, it exhibits excellent adaptability to support threshold decreases. We also show that the incremental update capability of the system provides significant performance advantages over full re-computation even for relatively large update sizes.

Categories and Subject Descriptors

H.2.8 [**Database Applications**]: Data mining

General Terms

Algorithms, Experimentation

Keywords

Web Usage Mining, Frequent Patterns, Traversal Patterns, Sequence Mining, Incremental Mining, Prediction, Prefetching, Web Logs

1. INTRODUCTION

Web Usage Mining. A sequence database stores a collection of sequences, where each sequence is a collection of ordered data items or events. Examples of sequences are DNA sequences, web usage data files or customers' transactions logs. For web applications, where users' requests are satisfied by downloading pages to their local machines, the use of mining techniques to predict access behaviors and hence help with prefetching of the most appropriate pages to the local machine cache can dramatically increase the runtime performance of those applications. These mining techniques analyze web log files composed of listings of page accesses (references) organized typically into sessions. These techniques are part of what is called *Web Usage Mining*, a term first introduced by Cooley et al. [2]. Typically web usage mining techniques rely on a Markov assumption with depth n. This means that it is assumed that the next request page depends only on the last n pages visited. A study conducted in [9] showed that Markov based structures for web usage mining is best suited for prefetching, targeted ads, and personalization. Web usage mining approaches can be classified based on the type of patterns they produce into three categories [6]: association rules, frequent sequences, and frequent generalized sequence. With association rules, the problem of finding web pages visited together is similar to finding association among item sets in transaction databases. Frequent sequences can be thought of as an ordered (by time) list of non-empty item sets, and hence frequent sequence mining can be thought of as association rule mining over temporal data sets. A frequent generalized sequence is a frequent sequence that allows wildcards in order to reflect the user's navigation in a flexible way [5]. [6] evaluated the three approaches and found that the frequent sequence approach gives better results than the association rules and the frequent generalized sequence approaches when we need to find the correct predictions within the first predictions. Frequent sequences are also known as *Traversal Patterns*. According to [3], traversal patterns can be classified based on four main features, (1) whether or not the order of page references in a pattern matters, (2) whether or not duplicate page references (backward traversal and page refresh/reload) are allowed, (3) whether patterns must consist of contiguous page references or they can have gaps, and (4) whether or not only maximal patterns are considered[1].

Mining Cost. In general, discovering frequent patterns in large databases is a costly process in terms of I/O and CPU costs. One major cost associated with the mining process is the generation of potentially frequent items (or sequences), called candidate item sets. Many mining techniques use an Apriori style level-wise candidate generation approach [1, 11, 13] that requires multiple expensive scans of the database, one for each level, to determine which

[1]A pattern is maximal when it is not part of another pattern.

of the candidates are in fact frequent. To address this issue, Han et al. [7] proposed a frequent pattern growth (FP-growth) based mining method that avoids costly repeated database scans and candidate generation. Their work focuses on the discovery of frequent item sets in transactional databases. In that work the order of the items in each record (i.e. in each transaction) is not of consideration. Hence it does not support mining for sequences where order among items is important. We now propose an extension of their technique to tackle the sequence mining case. The mining cost is even more prohibitive for dynamic databases which are subject to updates such as the continuous insertion of new sessions to the web log. In this case the reconstruction of frequent sequences may require re-executing the mining process from the beginning.

Problem Description. In this work we are particularly interested in web usage mining for the purpose of extracting frequent sequence patterns that can be used for pre-fetching and caching. For pre-fetching and caching, knowledge of such ordered contiguous page references is useful for predicting future references [3]. Furthermore, knowledge of frequent backward traversal is useful for improving the design of web pages [3]. In other words we are interested in mining for *traversal patterns*, where *traversal patterns* are defined to be sequences with duplicates as well as consecutive ordering between page references [16]. Our goal is to introduce a technique for discovering such sequence patterns, that is efficient, yet incremental and can adapt to user parameter changes. The patterns extracted by our system follow the Markov assumption discussed above and have four properties: (1) the order of page references in patterns is important, (2) duplicate page references are allowed (backward traversals and page refreshes), (3) patterns consist of contiguous page references, and (4) maximal and non-maximal patterns are allowed.

Contributions. We propose a frequent sequence tree structure (FS-tree) for storing compressed essential information about frequent sequences. We also introduce an algorithm which we call Frequent Sequence mining (*FS-mine*) that analyzes the FS-tree to discover frequent sequences. Our approach is incremental in that it allows updates to the database to be incrementally reflected in the FS-tree and in the discovered frequent sequences, without the need to reload the whole database or to re-execute the whole mining process from scratch. Finally the user can interactively change key system parameters (in particular the minimum support threshold and the maximum pattern size) and the system will remove the patterns that are no longer frequent and will introduce the patterns that are now frequent according to the new parameter values, without the need for scanning and loading the entire database. The results of the experiments that we have conducted using our approach, and compared against two other approach from the literature, show that our system, as well as the other two systems, scales up linearly with the size of the input database. Furthermore, our system shows a much better response time to the decrease in the support level than the other two systems. The incremental update capability of our approach provides significant performance advantages over full re-computation even for relatively large update sizes.

Paper Outline. The rest of this document is organized as follows. Section 2 discusses related work. Section 3 introduces the FS-tree data structure design and the FS-tree construction algorithm. Section 4 describes the FS-mine algorithm for discovering frequent sequences from the FS-tree structure. Section 5 describes the incremental and interactive mining algorithms. Section 6 discusses our experiment results. Lastly, Section 7 provides some conclusions and future work ideas.

2. RELATED WORK

Nanpoulos et al. [10] proposed a method for discovering access patterns from web logs based on a new type of association patterns. They handle the order between page accesses, and allow gaps in sequences. They use a candidate generation algorithm that requires multiple scans of the database. Their pruning strategy assumes that the site structure is known. Srikant and Agrawal [14] presented an algorithm for finding generalized sequential patterns that allows user-specified window-size and user-defined taxonomy over items in the database. This algorithm required multiple scans of the database to generate candidates.

Yang et al. [17] presented an application of web log mining that combines caching and prefetching to improve the performance of internet systems. In this work, association rules are mined from web logs using an algorithm called *Path Model Construction* [15] and then used to improve the GDSF caching replacement algorithm. These association rules assumes order and adjacency information among page references. Han et al. [7] proposed a technique that avoids the costly process of candidate generation by adapting a pattern growth method that uses a highly condensed data structure to compress the database. The proposed technique discovers unordered frequent item sets. However, is does not support the type of sequences we are interested in. Our work is similar to [7] in that it uses a condensed data structure and avoid expensive candidate generation. Yet our approach takes order among input items (page references) into consideration

Parthasarathy et al. [12] introduced a mining technique given incremental updates and user interaction. This technique avoids re-executing the whole mining algorithm on the entire data set. A special data structure called incremental sequence lattice and a vertical layout format for the database are used to store items in the database associated with customer transaction identifiers. Their performance study has shown that the incremental mining is more efficient than re-computing frequent sequence mining process from scratch. However, the limitation of their approach, as they point out, is the resulting high memory utilization as well as the need to keep an intermediate vertical database layout which has the same size as the original database [12]. Similar in spirit to [12], we store in the FS-Tree additional data to reduces the work required at later stages, yet we use very different data structures and algorithms.

Xiao and Dunham [16] proposed an incremental and adaptive algorithm for mining for traversal patterns. This work relies on a generalized suffix tree structure that grows quickly in size, since inserting a sequence into the suffer tree involves inserting all its suffer into the tree. Whenever the size of the tree reaches the size of the available memory during tree construction, pruning and compression techniques are applied to reduce its size in order to be able to continue the insertion process of the remaining sequences from the database. This process of reducing the size of the tree to fit into the available memory is referenced to as adaptive property. Conversely, we do not need to interrupt the FS-Tree construction process to prune or compress the tree as we prune the input sequences before inserting them into the tree and we insert only potentially frequent subsequences. Unlike [16], the adaptive mining here means that the system adapts to changes in user-specific parameters.

3. FS-TREE CONSTRUCTION

Frequent Sequences. Let $I = \{i_1, i_2, ..., i_m\}$ be a set of unique items, such as page references. A sequence **Seq** = $<p_1 p_2 ... p_n>$ is an ordered collection of items with $p_i \in I$ for 1 $\leq$ i $\leq$ n. A database **DB** (for web usage mining typically a web log file) stores a set of records (sessions). Each record has two fields: the record ID field,

SID, and the input sequence field, **InSeq**. The order of the items does matter within such an input sequence. When an item p_{i+1} comes immediately after another item p_i we say that there is a link l_i from p_i to p_{i+1}. We denote that as $l_i = p_i - p_{i+1}$. We may also represent a sequence as **Seq** = $p - P$, where p is the first element in the sequence and P is the remaining subsequence.

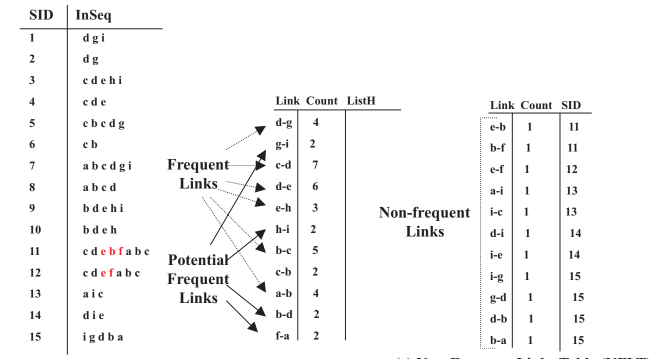
For a link h, the *support count*, $Supp^{link}(h)$, is the number of times this link appears in the database. For example if the link $a-b$ appears in the database five times we say that $Supp^{link}(a - b)$ = 5. For a sequence Seq = $<p_1p_2...p_n>$ we define its size as n which is the number of items in that sequence. Given two sequence S = $<p_1p_2...p_n>$ and R = $<q_1q_2...q_m>$ we say that S is a subsequence of R if there is some i, $1 \leq i \leq m - n + 1$, such that $p_1 = q_i$, $p_2 = q_{i+1}$, ..., $p_n = q_{i+(n-1)}$. For a given input sequence Seq = $<p_1p_2...p_n>$ we consider only subsequences of size $\geq$ 2. For example, if a record in the database has an input sequence $<abcd>$ we extract subsequences $<abcd>$, $<abc>$, $<bcd>$, $<ab>$, $<bc>$, and $<cd>$ from that input sequence. The support count $Supp^{seq}(Seq)$ for a sequence Seq is the number of times the sequence appears in the database either as a the full sequence or as a subsequence of sessions. We allow item duplicates in frequent sequences, which means that the same item can appear more than once in the same sequence. Duplicates can be either backward traversal, e.g. the page b in $<abcb>$, or refresh/reload of the same page, e.g. the page a in $<aabc>$.

Sequence Support. The behavior of our system is governed by two main parameters. The first parameter is *minimum **link** support count*, $MSuppC^{link}$, which is the minimum count that a link should satisfy to be considered potentially frequent. $MSuppC^{link}$ is obtained by multiplying the total number of links in the database by a desired minimum link support threshold ratio $MSuppR^{link}$. $MSuppR^{link}$ is the frequency of the link in the database to the total number of links in the database ($Supp^{link}$/total # of links in the database) which a link has to satisfy in order to be considered potentially frequent. $MSuppR^{link}$ is a system parameter (not set by the user) and is used by the FS-tree construction algorithm to decide what links to include in the FS-tree as will be discussed later. The second parameter $MSuppC^{seq}$, is the *minimum **sequence** support count*, that denotes the minimum number of times that a sequence needs to occur in the database to be considered frequent. $MSuppC^{seq}$ is obtained by multiplying the total number of links in the database by a desired minimum sequence support threshold ratio $MSuppR^{seq}$. This desired ratio is the frequency of the sequence in the database to the total number of links in the database ($Supp^{seq}$/total # of links in the database) which a sequence has to satisfy in order to be considered frequent. $MSuppR^{seq}$ is set by the user and is used by the FS-Mining algorithm during the mining process. $MSuppC^{seq}$ is the main parameter needed for sequence mining in our system. At all times, we assume that $MSuppC^{link} \leq MSuppC^{seq}$. The reason for having $MSuppC^{link}$ is to allow the system to maintain more data about the input database than required for the mining task at hand. This will help in minimizing the amount of processing needed when handling incremental updates to the database, or when the user changes system parameters. This issues will be discussed in more detail in the incremental and interactive mining sections. In short, we consider any sequence Seq that has $Supp^{seq}(Seq) \geq MSuppC^{seq}$ a **frequent sequence** or a **pattern**. We consider any link h that has $Supp^{link}(h) \geq MSuppC^{seq}$ a **frequent link** (also considered a frequent sequence of size 2) . And if $Supp^{link}(h) \geq MSuppC^{link}$ and $Supp^{link}(h) < MSuppC^{seq}$ we call h a **potentially frequent** link. And if $Supp^{link}(h)$ does not satisfy $MSuppC^{link}$ and $MSuppC^{seq}$ we call h a **non-frequent link**.

Frequent Sequence Tree. We now describe our proposed data structure that we use to store potentially frequent sequences to facilitate the mining process.

Definition 1 *A frequent sequence tree is a structure that consists of the following three components:*

- *A tree structure with a special root node R and a set of sequence prefix subtrees as children R. Each node n in the FS-tree has a **node-name** field that represents an item from the input database[2]. Each edge in the tree represents a $link$ relationship between two nodes. Each edge has three fields: **edge-name**, **edge-count**, and **edge-link**. **Edge-name** represents the **from** and **to** nodes that are linked using this edge, **edge-count** represents the number of sequences that share this edge in the particular tree path, where a tree path is the prefix path that starts from the tree root to the current node.*
- *A header table HT that stores information about frequent and potentially frequent links in the database. Each entry in the header table HT has three fields: **Link** which stores the name of the link, **count** stores the count of that link in the database, and **listH** pointer, which is a linked list head pointer that points to the first edge in the tree that has the same **edge-name** as the link name. Note that the **edge-link** field in each edge in the tree is pointing to the next edge in the FS-tree with the same edge-name (or null if there is none).*
- *A non-frequent links table $NFLT$, that stores information about non-frequent links. This table is only required for supporting the incremental feature of the system. The $NFLT$ has three fields: **Link** which stores the name of the link, **count** which stores the count of that link in the database, and **SIDs** which stores the IDs of records in the database that have sequences that include that link.*


Frequent
Links
Potential
Frequent
Links
Non-frequent
Links

SID	InSeq
1	d g i
2	d g
3	c d e h i
4	c d e
5	c b c d g
6	c b
7	a b c d g i
8	a b c d
9	b d e h i
10	b d e h
11	c d e b f a b c
12	c d e f a b c
13	a i c
14	d i e
15	i g d b a

(a) Web Log File

Link	Count	ListH
d-g	4	
g-i	2	
c-d	7	
d-e	6	
e-h	3	
h-i	2	
b-c	5	
c-b	2	
a-b	4	
b-d	2	
f-a	2	

(b) Header Table (HT)

Link	Count	SID
e-b	1	11
b-f	1	11
e-f	1	12
a-i	1	13
i-c	1	13
d-i	1	14
i-e	1	14
i-g	1	15
g-d	1	15
d-b	1	15
b-a	1	15

(c) Non-Frequent Links Table (NFLT)

Figure 1: (a) Web log file example, (b) Header table HT and (c) Non-frequent links table $NFLT$.

Frequent Sequence Tree Construction. Consider the web log file in Figure 1(a). It stores a set of users' sessions where each session has two fields: SID that stores the session id and $InSeq$ that stores sequence of page references accessed by the user in a certain order. Given such input web log file, and assuming $MSuppC^{link}$

[2]For supporting the incremental property of the system, we extent the node by adding a structure that stores a single session ID that ends at this node for certain sequences. We will discuss this structure in more details in the incremental mining Section.

= 2 and $MSuppC^{seq}$ = 3, [3] we construct the FS-tree as follows:

1) We first perform one scan of the input database (log file) to obtain counts for links in the database.

2) We identify those links that have $Supp^{link} \geq MSuppC^{link}$, and we insert them in the header table (HT), along side with their counts, as shown in Figure 1(b). For links that do not satisfy the predefined $MSuppC^{link}$ we insert them in the non-frequent links table ($NFLT$), along side with their counts and the SID of sessions they are obtained from[4], this is shown in Figure 1(c).

3) We create the root of the FS-tree.

4) We then perform a second scan of the database calling the $insertTree$ function (shown in Figure 2) for each input sequence.

```
FS-tree construction Algorithm
Input:  Sequence Database DB and minimum link support MSupp_l
Output: Frequent sequence tree FS-tree of DB
Method:
(1) Scan the DB once to collect counts for all links
(2) Classify links and insert them in HT and NFLT tables
(3) Create a root R for the FS-tree
(4) For (each record in DB get sequence InSeq_i)
        call insertTee (R, InSeq_i)
(5) Return FS-tree
```

```
Function insertTree (tree root node S, sequence p-P) : Updated FS-tree in which all the
        potentially frequent subsequences are inserted .
(1)  If (link p-P    HT) {
(2)      If (S has a child N and N.node-name = p) {
(3)         increment S-N.edge-count by 1
(4)      }Else {
(5)          Create node N with N.node-name = p
(6)          Create edge S-N with S-N.edge-count = 1
(7)          Append edge S-N to HT_{S-N}.ListH   }
(8)    If (P is non-empty) {call insertTree(N, P)}
(9)  }Else if (link p-P    NFLT) {
(10)   If (P is non-empty) {call insertTree(R, P) } }
(11) If P is last page in InSeq_i and InSeq_i was not cut, store Seq.ID in seqEnd.ID
```

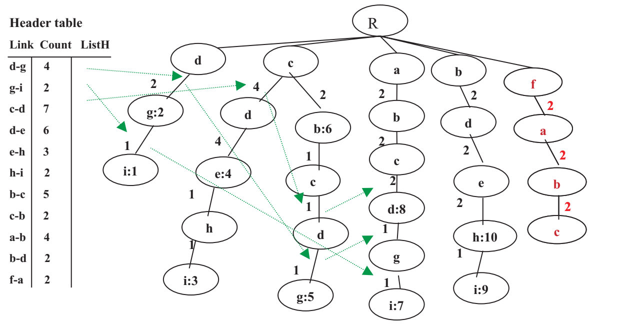
Figure 2: FS-tree construction.

Figure 3 shows the FS-tree constructed for the example in Figure 1[5]. The total number of links in the database is 52, based on first database scan. And assuming that the system defines $MSuppR^{link}$ to be 4% and the user defines $MSuppR^{seq}$ to be 6%, we obtain $MSuppC^{link}$ = 2 and $MSuppC^{seq}$ = 3 accordingly (note that $MSuppC^{link}$ is used in FS-tree construction, while $MSuppC^{seq}$ is used later in FS-tree mining). We create the FS-tree root node R. We then insert sequences into the tree starting from the tree root using the procedure described above. For the sequence $<dgi>$ we start from the root and since the tree is empty so far, we create two new nodes with names d and g. We also create an edge $d-g$ that is assigned **edge-count** = 1. In addition, we link the **ListH** pointer for link $d-g$ in **HT** to the new edge. Lastly, we insert the node i into the **FS-tree** creating a new node and the edge $g-i$ with **edge-count** =1, and link **ListH** pointer for link $g-i$ in **HT** to that edge. When inserting the second input sequence $<dg>$, we share the nodes d and g and the edge $d-g$ and increment the count of that edge to 2.

[3]Frequent links are those satisfying both support thresholds, Potentially Frequent links are those satisfying only $MSuppC^{link}$ and Non-frequent links are those not satisfying any of the two support thresholds.

[4]only required for supporting incremental mining

[5]Note that we only show some of the lines that link the header table to edges in the FS-tree for simplicity



Header table

Link	Count	ListH
d-g	4	
g-i	2	
c-d	7	
d-e	6	
e-h	3	
h-i	2	
b-c	5	
c-b	2	
a-b	4	
b-d	2	
f-a	2	

Figure 3: The FS-tree constructed for the example in Figure 1

Next we insert the sequence $<cdehi>$ by creating new nodes and edges (with counts = 1) for all the items and links in the sequence since there was no possible path sharing. Sequences in sessions with ids 3 to 10 are inserted following the same logic described above. Session 11 ($<cdebfabc>$) is a different from prior sessions, since the sequence in this session has non-frequent links, namely $e-b$ and $b-f$. First, the sub-sequence $<cde>$ is inserted in the tree. Insertion here involves sharing existing nodes and edge and incrementing edges counts. Then we ignore the two non-frequent links $e-b$ and $b-f$. The sub-sequence $<fabc>$ is inserted from the tree root by creating new nodes and edges as described above. For session 12 we insert the sub-sequence $<cde>$ into the tree, then we encounter the non-frequent link e-f, so we skip it and insert the remaining sub-sequence $<fabc>$ starting from the root node of the tree. Sessions 13, 14 and 15 are not inserted, totally or partially, into the FS-tree since all their links are non-frequent. See Figure 3 for the fully constructed FS-tree.

FS-Tree Size. The FS-tree is a compressed form for representing sequences scanned from the input web log file. It is compressed in two manners, first, not all sequences are stored in the tree, only those sequences/subsequence that are potentially frequent are inserted and stored in the FS-Tree. This ensures that any non-potential frequent sequences/subsequences are pruned from the beginning and are not inserted into the tree. Second, insertion into the tree involves sharing of all possible existing nodes and edges. This is even more powerful with the existence of the initial pruning step discussed earlier because it increases the possibilities of sharing tree paths. To give an idea about how small our proposed FS-Tree is we consider the work done in [16], that we have discussed earlier in Section 2. In [16] a suffix tree is constructed and mined for frequent sequences. To construct the suffix tree all possible suffixes of each input sequence is inserted into the tree. This cause the tree to grow in size very quickly. For example if we construct a suffix tree for the sequences shown in Figure 1 we end up with a tree with 95 nodes[6] while our FS-tree requires only 28 nodes, as shown in Figure 3. It is possible to collapse nodes with single child in suffix trees to reduce the number of nodes and edges. The same technique can also be used with the FS-Tree. Collapsing the suffix tree that we have constructed above results in a tree with 50 nodes while collapsing our FS-tree results in a tree with 8 nodes only.

4. MINING THE FS-TREE

Based on $MSuppC^{link}$ and $MSuppC^{seq}$ we classify the links in the database into three types (See Figure 1):

- *Frequent links*: links with support count $Supp^{link} \geq MSuppC^{seq}$

[6]The figure showing this suffix tree is removed from here due to space limitations

$\geq MSuppC^{link}$. These links are stored in HT and are represented in the FS-tree and can be part of frequent sequences.

- *Potentially Frequent links*: links with support count $Supp^{link}$ $\geq MSuppC^{link}$ and $Supp^{link} < MSuppC^{seq}$. These links are stored in the HT and are represented in the FS-tree but they can't be part of frequent sequences (needed for efficient incremental and interactive performance).
- *Non-frequent links*: links with support count $Supp^{link} < MSuppC^{link}$. These links are stored in $NFLT$ and are not represented in the FS-tree (needed for efficient incremental and interactive performance).

Only frequent links may appear in frequent sequences, hence, when mining the FS-tree we consider only links of this type. Before we introduce the FS-mine algorithm, we highlight the properties of the FS-tree.

Properties of the FS-trees. The FS-tree has the following properties that are important to the FS-mine algorithm:

- Any input sequence that has non-frequent link(s) is pruned before being inserted into the FS-tree.
- If $MSuppC^{link} < MSuppC^{seq}$, the FS-tree is storing more information than required for the current mining task.
- We can obtain all possible subsequences that end with a given frequent link h by following the *ListH* pointer of h from the header table to correct FS-tree branches.
- In order to extract a sequence that ends with a certain link h from an FS-tree branch, we only need to examine the branch prefix path that ends with that link (h) backward up to (maximum) the tree root.

Now we describe in detail the mining steps that we use to extract frequent sequences from the FS-tree. We assume the FS-tree shown in Figure 3, and $MSuppC^{link} = 2$ and $MSuppC^{seq} = 3$ as our running example.

FS-tree Mining Steps. Figure 4 lists the FS-Mine Algorithm. The algorithm has four main steps that are performed for only frequent links (potentially frequent links are excluded) in the header table (HT):

1) Extracting derived paths. For link h in HT with $Supp^{link}(h)$ $\geq MSuppC^{seq}$ we extract its derived paths by following the *ListH* pointer of h from HT to edges in the FS-tree. For each path in the FS-tree that contains h we extract its path prefix that ends at this edge and go maximum up to the tree root[7]. We call these paths *derived paths* of link h. For example, from Figure 3, if we follow the $ListH$ pointer for the link $e-h$ from the header table we can extract two derived paths: $(c-d:4, d-e:4, e-h:1)$ and $(b-d:3, d-e:2, e-h:2)$.

2) Constructing conditional sequence base. Given the set of derived paths of link h extracted in previous step we construct the *conditional sequence base* for h by setting the frequency count of each link in the path to the count of the h link (this gives the frequency of the full derived path). We also remove h from the end of each of the derived paths For example, given the two derived paths extracted above for link $e-h$, the conditional base for that link consists of: $(c-d:1, d-e:1)$ and $(b-d:2, d-e:2)$.

3) Constructing conditional FS-tree. Given the conditional base for h, we create a tree and insert each of the paths from the conditional base of h into it in a backward manner. We create necessary nodes and edges or share them when possible (incrementing edges counts). We call this tree the *conditional FS-tree* for link h. For example, given the conditional base for link $e-h$ the constructed conditional FS-tree is shown in Figure 5.

[7]Note the backward prefix extraction might terminate before the tree root and return a smaller prefix path in two cases: (1) reaching the limit determined by the user as the maximum pattern length he is interested in discovering or (2) encountering a potentially frequent link (since we do not mine for them).

4) Extracting frequent sequences. Given a *conditional FS-tree* of a link h, we perform a depth first traversal for that tree and return only sequences satisfying $MSuppC^{seq}$. By traversing the conditional FS-tree of link $e-h$ only the sequence $<de>$ satisfies the $MSuppC^{seq}$, so we extract it. We then append the link $e-h$ to the end of it to get the full size frequent sequence: $<deh:3>$ where 3 represents the support (count) of that sequence.

FS-Mine Algorithm

Input: FS-tree root **R,** and minimum sequence support $\mathbf{MSupp_S}$

Output: Frequent sequences

Method:

```
(1)  Frequent sequences set FSS
(2)  For (all links $\mathbf{l_i}$   HT and $\mathbf{l_i}$.count   $\mathbf{MinSupp_s}$) {
(3)    Conditional sequence set CSS
(4)    For (all paths $\mathbf{P_j}$ in FS-tree reachable from HT.ListH($\mathbf{l_i}$)){
(5)        CSS   CSS   extract $\mathbf{P_j}$, remove last link, and adjust $\mathbf{P_j}$.count = last link count }
(6)    Conditional FS-tree CFST
(7)    Construct CFST
(8)    For (all sequences $\mathbf{Seq_l}$ in CFST){
(9)        FSS   FSS   concatenate ($\mathbf{Seq_l}$, $\mathbf{l_i}$)   } }
```

Figure 4: FS-Mine Algorithm.

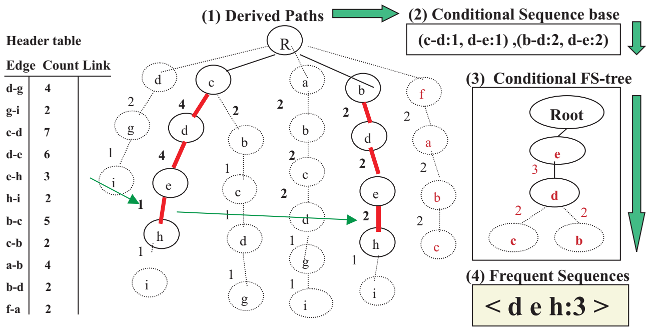


Figure 5: Mining steps for link $e-h$ from the example in Figure 1.

We perform the same steps for the other frequent links in HT, namely $d-g$ $a-b$, $b-c$, $d-e$, and $c-d$. The detailed mining steps for these links are shown in Table 1. The last column in that table gives the final result for the mining process. The generated frequent sequences are: $<deh:3>$, $<abc:4>$, $<cde:4>$, and $<bcd:3>$ in addition to the frequent links themselves: ($<eh:3>$, $<dg:4>$, $<ab:4>$, $<bc:5>$, $<de:6>$, and $<cd:7>$) as they are considered frequent sequences of size 2.

5. INCREMENTAL MINING

In the presence of incremental updates $\triangle$**DB** to the sequence database, our goal is to propagate these updates into the generated frequent sequences with minimum cost. In particular, we aim to develop an incremental maintenance strategy that avoids the need for expensive scans of the complete sequence database and the complete recomputation of frequent sequences. In this section, we discuss requirements for supporting Incremental feature of the FS-miner. We then address how to maintain the FS-tree incrementally

Link	*Derived Paths*	*Conditional Sequence bases*	*Conditional FS trees*	*Frequent Sequences*
e-h	(c-d:4, d-e:4, e-h:1) , (b-d:3, d-e:2, e-h:2)	(c-d:1, d-e:1) , (b-d:2, d-e:2)	(d-e:3)	$<deh : 3>$
d-g	(d-g:2), (c-b:2, b-3:1,c-d:1,d-g:1), (a-b:2,b-c:2 ,c-d:2,d-g:1)	(c-b:1,b-c:1,c-d:1), (a-b:1, b-c:1 ,c-d:1)	ϕ	ϕ
a-b	(a-b:2), (f-a:2, a-b:2)	(f-a:2)	ϕ	ϕ
b-c	(c-b:2, b-c:1), (a-b:2,b-c:2), (f-a:2, a-b:2,b-c:2)	(c-b:1), (a-b:2), (f-a:2, a-b:2)	(a-b:4)	$<abc : 4>$
d-e	(c-d:4, d-e:4), (b-d:3, d-e:2)	(c-d:4),(b-d:2)	(c-d:4)	$<cde : 4>$
c-d	(c-d:4), (c-b:2, b-c:1,c-d:1), (a-b:2,b-c:2 ,c-d:2)	(c-b:1, b-c:1), (a-b:2,b-c:2)	(b-c :3)	$<bcd : 3>$

Table 1: Mining for all sequences satisfying $MSuppC^{seq}$=3.

without reconstructing it from scratch and how to mine incrementally for frequent sequences.

We first highlight the additional information we need to maintain to support incremental mining:

1) The Non-Frequent Links Table **NFLT**, described earlier in Definition 1.

2) We extend the FS-tree node by adding to it a new structure called **seqEnd**. This structure has two fields: **sid** and **count**. **sid** stores a record id of a sequence (from the database), or null. The value of **sid** in **seqEnd** is assigned at tree construction time. When we insert an input sequence into the FS-tree we might set **sid** of the node inserted into the tree to be equivalent to the input sequence id. To assign a new value for **sid** two conditions must be satisfied: (1) the input sequence is inserted as one piece into the tree without being pruned[8] and (2) the **sid** is not already set to another sequence id (since we store only one id in this field). The second field, **count**, stores a count that indicates how many complete (unpruned) input sequences share the same tree branch that ends at this node. Figure 3 shows nodes in the tree with **sid** set to session ids from the database [9].

5.1 Maintaining the FS-tree Incrementally

The FS-miner supports both database inserts and deletes. Our incremental FS-tree construction algorithm takes as input the FS-tree representing the database state before the update and $\triangle$**DB**. Then it inserts (or deletes) sequences from the tree. In some cases, the FS-tree construction algorithm performs partial restructuring of the tree, that is, some branches might be pruned or moved from one place to another in the FS-tree. We now give an overview of how the algorithm works[10].

The algorithm first obtains the count of links in $\triangle$**DB** by performing one scan of $\triangle$**DB**. Then link counts in HT and $NFLT$ are incremented or decremented. $MSuppC^{seq}$ and $MSuppC^{link}$ values are updated if applicable. Link entries in $NFLT$ that now become frequent (or potentially frequent) are moved to HT. Links that were originally in HT and moved to $NFLT$, because they are no longer satisfying $MSuppC^{seq}$ and $MSuppC^{link}$ should no longer be presented in the FS-tree, so we prune edges that represent them from the FS-tree. For links that were originally in $NFLT$ and moved to HT, we obtain input sequences in the order which they appear from the original database[11]. We insert them into the FS-tree using the function $insertTreeInc$. The main difference between this function and the normal $insertTree$ function described earlier is that $insertTreeInc$ aims to compose sequences that were previously decomposed by the $insertTree$ at the initial tree construction phase. After this point, we insert the remaining subsequence starting from the current node. At the same time we call the $deleteTree$ function that deletes the same remaining subsequence from the top of the FS-tree (as it had previously been inserted there).

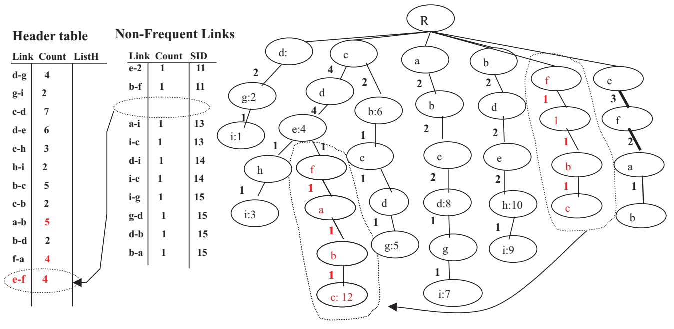


Figure 6: The effect of inserting the records $\{16, < efa >\}$, $\{17, < ef >\}$, $\{18, < efab >\}$ to the database in Figure 1.

Example: As an example for incremental inserts, assume that the following tuples where inserted into the log file in our running example in Figure 1: $\{16, < efa >\}$, $\{17, < ef >\}$, $\{18, < efab >\}$. Figure 6 shows the effect of inserting the new input sequences. First, we scan the new records to obtain counts of links in the inserted session and we update counts of links $a-b$ and $f-a$ in HT and link $e-f$ in $NFLT$. Assuming the $MSuppC^{link}$ and $MSuppC^{seq}$ maintain the same values (2 and 3 respectively), link $a - b$ maintains the same status (frequent), links $f - a$ and $e - f$ becomes frequent thus are moved to table HT. The next step is to prune the tree by removing edges for any link transitioned from frequent to non-frequent. In this example we do not have any. Next we restructure the tree for links that were not frequent and became frequent (link $e - f$ in our example). We obtain from the **SIDs** field of link $e - f$ entry in $NFLT$ sequence id = 12 as the only sequence where the link appears in original database. We retrieve this sequence ($<cdefabc>$) from the original database and insert it into the FS-tree using the $insertTreeInc$ function. This function will first traverse the tree branch that corresponds to the subsequence represented in the tree from before ($<cde>$) and create a new edge for it when it encounters the link $e-f$. Insertion will then continue for the remaining subsequence ($<fabc>$) following this point. At the same time it calls the $deleteTree$ function for the subsequence $<fabc>$ to delete it from the root of the FS-tree. The last step in the incremental FS-tree constructions is to insert all the input sequences from $\triangle$**DB** in the FS-tree using the $insertTree$ function, resulting in the tree shown in Figure 6. For an example illustrating incremental deletes we refer the reader to [4].

5.2 Mining the FS-tree Incrementally

After refreshing the FS-tree, the incremental mining is invoked for certain links in HT, namely those affected by the update. We first need to understand the effect of database updates on different

[8] All links in the sequence are frequent.

[9] Counts are not shown there for simplicity since they are all equal to 1 for current example.

[10] We have removed the figure showing algorithm itself due to space limitations, the reader is refereed to [4] for that algorithm

[11] Recall that for each we maintained a list of sequence IDs in which the link appeared in the database.

types of links[12]. We can classify the possible change in the type of a link due to database updates into 9 different transaction types as shown in Figure 7[13]. We categorize how the incremental mining algorithm deals with these different transaction cases into four transaction categories:

(1) Type 1: we mine for those links if they are affected [14].
(2) Type 2 and 4: we mine for these links.
(3) Type 3 and 5: we delete previously discovered patterns that include these links.
(4) Type 6, 7, 8 and 9: we do nothing.

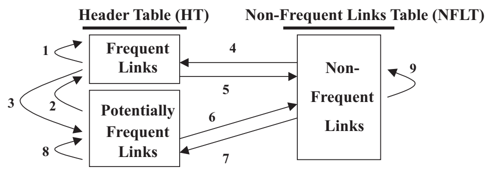


Figure 7: The effect of incremental updates on links in the database

The incremental FS-mine algorithm is shown in Figure 8. The mining algorithm starts by dropping any sequence in the previously discovered frequent sequences that is either of transaction type 3 or 5 (no longer satisfying the new $MSuppC^{seq}$). Then for all links in the **HT** if the link satisfies the new $MSuppC^{seq}$ and if it is of transaction type 2 or 4, or of type 1 and is affected by the update, the algorithm applies the *FS-mine* algorithm for these links.

Incremental FS-Mine Algorithm

Input: FS-tree root **R,** set of frequent sequences **fSeq** and **set of** affected links **affLinks**
Output: new frequent sequences based on **affLinks**
Precondition: all counts for links are updated in HT and NFLT
Method:

```
(1) For (all frequent sequences fSeq_i )
(2)     If (fSeq_i count < Min-Supp_s or fSeq_i has link with count < Min-Supp_s){
(3)         delete fSeq_i}
(4) For (all links l_i    HT where l_i.count >= Min-Supp_s)
(5)         if (l_i moved from NFLT or l_i count was < Min-Supp_s or l_i    affLinks ) {
(6)             call FS-Mine (R) for l_i}
```

Figure 8: Incremental FS-Mine Algorithm.

Example: Consider that $\triangle$**DB** denotes an insertion of $\{16, <efa>\}$, $\{17, <ef>\}$, $\{18, <efab>\}$ described in example 3. link $a - b$ is affected by the update and maintained the same frequent status after the update. Link $f - a$ status is changed from *potentially frequent* to *frequent* due to the update. Link $e - f$ status is changed from non-frequent to frequent due to the update. These three links are the only ones affected by the update, hence we need to mine for these three links. Table 2 shows the steps in mining for these links and the resulting generated frequent sequences.

[12]The three different types of links we discussed earlier (frequent, potentially frequent and non-frequent).

[13]The starting point of the arrow refers to where the link used to be before the database updates and the ending point of the arrow refers to where the link ends up as a result of the database update.

[14]By affected we mean if the link was in $\triangle$DB, or if the link was in one of the subsequences that were deleted from the FS-tree in the tree restructuring process described earlier.

5.3 Interactive Mining

We want to allow the user to make changes to the minimum support value and get a response in a small amount of time. To achieve this goal we need to minimize the need to access the database and to re-execute the mining algorithm. We can support this goal in our system by setting the $MSuppC^{link}$ to a small enough value that is less than any value of $MSuppC^{seq}$ that the user is likely to use. The rational here is that since $MSuppC^{link}$ is responsible for determining the potentially frequent links and hence allowing them to be represented in the FS-tree. This ensures that if the user lowered the $MSuppC^{seq}$ to a value that is $\geq MSuppC^{link}$ we will have enough information in the FS-tree to calculate the new frequent sequences without the need to reference the original database. This is done by applying the FS-mine algorithm for the subset of links in **HT** that is satisfying the new $MSuppC^{seq}$. On the other hand, if the user increases the $MSuppC^{seq}$, we directly provide him/her with the subset of frequent sequences previously discovered that satisfies the new $MSuppC^{seq}$ without the need for any further computation. Our system also allows the user to vary the size of the frequent patterns he/she is interested in discovering. Also in this case the system does not use the input database, it only uses the FS-tree to extract the frequent sequences of the required sizes. We refer the user to [4] for an example of interactive mining.

6. EXPERIMENTAL EVALUATION

We use two data sets to test our system, the Microsoft Anonymous Web Data Set and the MSNBC Anonymous Web Data Set, both obtained from [8]. Each data set consists of a collection of sessions where each session has a sequence of page references. The Microsoft anonymous data set has 32711 sessions, a session contains from 1 up to 35 page references. The MSNBC data set has 989818 sessions. A session contains from 1 up to several thousands of page references [15]. One other important difference between the two data sets is the number of distinct pages. The Microsoft data set has 294 distinct pages, while the MSNBC data set has only 17 distinct pages (as each one of these pages is in fact encodes a category of pages). We compare the performance of our algorithm against two other algorithms from the literature: the $PathModelConstruction$ algorithm [15], and a variation of Apriori algorithm [1] for sequence data [16]. We have implemented the three systems in Java and have run the experiments on a PC with a 733 MHz Pentium processor and 512 MB of RAM.

Figure 9 shows that our system, and the other two systems, scale linearly to the database size. Our system tends to outperform the other two systems with data sets that have a large number of distinct items (such as the MS data set) while Apriori tends to perform slightly better in the case of data sets with a very small distinct items (such as the MSNBC set). This is because the candidate generation cost in this case is small. Note that part of the cost of our system is due to maintaining the extra data needed for incremental and interactive tasks. So while the other two systems are only performing the mining task at hand, our system is also maintaining as a byproduct the FS-tree that can later be used for incremental and interactive operations. We also tested the scalability of the system with respect to decreases of the support threshold level. Figure 10 shows that our system scales better with decreases of support level. In fact our system shows a very smooth response time to decreases

[15]We have preprocessed the MSNBC data sets to keep a maximum of 500 page references for each session to smooth the effect of very large sessions on experiments time.

[16]Optimized using hashing techniques and modified to provide the same sequential patterns we use.

Link	*Derived Paths*	*Conditional Sequence bases*	*Conditional FS trees*	*Frequent Sequences*
a-b	(c-d:4, d-e:4, e-f:1, f-a:1, a-b:1), (a-b:2), (f-a:1, a-b:1),(e-f:3, f-a:2, a-b:1)	(c-d:1, d-e:1, e-f:1, f-a:1) (f-a:1),(e-f:1, f-a:1)	(f-a:3)	$<fab:3>$
f-a	(c-d:4, d-e:4, e-f:1, f-a:1),(f-a:1), (e-f:3, f-a :2)	(c-d:1, d-e:1, e-f:1), (e-f:2)	(e-f:3)	$<efa:3>$
e-f	(c-d:4, d-e:4, e-f:1), (e-f:3)	(c-d:1, d-e:1)	ϕ	ϕ

Table 2: Incrementally mine for links a-b, f-a, and e-h where $MSuppC^{seq}$=3.

of the support level unlike the other two systems that experience a dramatic increase in cost when they hit lower support levels. This implies that even if we choose to utilize a low $MSuppC^{link}$, to better support the incremental and interactive tasks of the system at later stages, our system does not experience a significant overhead. The third experiment compares the performance of the incremental mining versus recomputation. Figure 11 shows that even with an incremental update size of up to one quarter of the size of the original database size, the FS-Miner's incremental feature provides significant time savings over full recomputation.

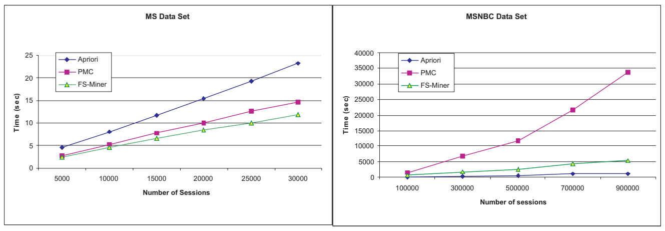


Figure 9: Scalability with number of input sessions

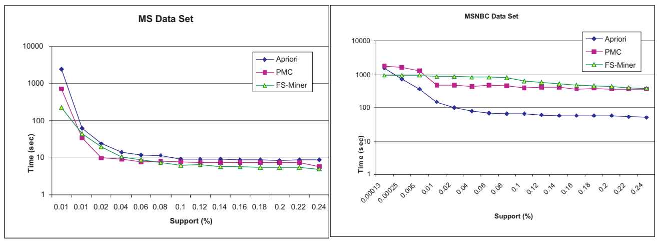


Figure 10: Scalability with support threshold

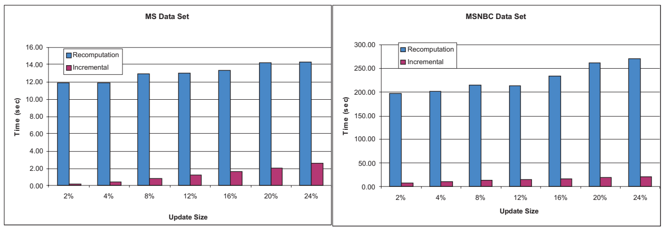


Figure 11: Incremental mining

7. CONCLUSIONS

In this paper we have proposed the FS-Miner, an incremental sequence mining system. The FS-Miner constructs a compressed data structure (FS-tree) that stores potentially frequent sequences and uses that structure to discover frequent sequences. This technique requires only two scans for the input database. Our approach allows for incremental discovery of frequent sequences when the input database is updated, eliminating the need for full recomputation. Our approach also allows interactive response to changes to the system minimum support. Our experiments show that the performance of our system scales linearly to increases in the input database size. It shows an excellent time performance when handling data sets with large number of distinct items. The FS-miner also shows great scalability with the decrease of the minimum support threshold when typically other mining algorithms tend to exhibit dramatic increases in response time. Finally the incremental functionality of our system shows a significant performance gain over recomputation even with large update sizes.

8. REFERENCES

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *VLDB*, pages 487–499, 1994.
[2] R. Cooley, J. Srivastava, and B. Mobasher. Web mining: Information and pattern discovery on the world wide web. In *CTAI*, pages 558–567, 1997.
[3] M. H. Dunham. *Data Mining: Introductory and Advanced Topics*. Prentice Hall, 2003.
[4] M. EL-Sayed, E. A. Rundensteiner, and C. Ruiz. FS-Miner: An Efficient and Incremental System to Mine Contiguous Frequent Sequences. Technical Report WPI-TR-03-20, Department of Computer Science. WPI, June 2003.
[5] W. Gaul and L. Schmidt-Thieme. Mining web navigation path fragments. In *WEBKDD*, 2000.
[6] M. Gery and H. Haddad. Evaluation of web usage mining approaches for user's next request prediction. In *WIDM*, pages 74–81, 2003.
[7] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *SIGMOD*, pages 1–12, May 2000.
[8] Hettich, S. and Bay, S. D. The UCI KDD Archive. Irvine, CA: University of California, Department of Information and Computer Science. http://kdd.ics.uci.edu, 1999.
[9] S. Jespersen, T. B. Pedersen, and J. Thorhauge. Evaluating the Markov assumption for web usage mining. In *WIDM*, pages 82–89, 2003.
[10] A. Nanopoulos, D. Katsaros, and Y. Manolopoulos. Effective prediction of web-user accesses: A data mining approach. In *WEBKDD Workshop, San Francisco, CA*, Aug. 2001.
[11] R. Ng, L. Lakshmanan, J. Han, and Pang. Exploratory mining and pruning optimization of constrained association rules. In *SIGMOD Conf.*, pages 13–24, 1998.
[12] S. Parthasarathy, M. J. Zaki, M. Ogihara, and S. Dwarkadas. Incremental and interactive sequence mining. In *CIKM*, pages 251–258, 1999.
[13] S. Sarawagi, S. Thomas, and R. Agrawal. Integrating association rule mining with relational database systems: alternatives and implications. In *SIGMOD Conf.*, pages 343–354, 1998.
[14] R. Srikant and R. Agrawal. Mining sequential patterns: Generalizations and performance improvements. In *EDBT*, pages 3–17, 1996.
[15] Z. Su, Q. Yang, Y. Lu, and H. Zhang. Whatnext: A prediction system for web request using n-gram sequence models. In *WISE*, pages 214–221, 2000.
[16] Y. Xiao and M. H. Dunham. Efficient Mining of Traversal Patterns. *DKE*, 2(39):191 – 214, 2001.
[17] Q. Yang, H. H. Zhang, and I. T. Y. Li. Mining web logs for prediction models in WWW caching and prefetching. In *KDD*, pages 473–478, 2001.