WPI-CS-TR-03-20 June 2003

# FS-Miner: An Efficient and Incremental System to Mine Contiguous Frequent Sequences

by

Maged El-Sayed
Elke A. Rundensteiner
Carolina Ruiz

Computer Science
Technical Report
Series

WORCESTER POLYTECHNIC INSTITUTE

Computer Science Department
100 Institute Road, Worcester, Massachusetts 01609-2280

# FS-Miner: An Efficient and Incremental System to Mine Contiguous Frequent Sequences

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

June 8, 2003

**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 appropriate web pages.

Mining solutions in the literature are often based on the use of an Apriori-like candidate generation strategy, which typically requires numerous scans of the potentially huge sequence database. In this paper we instead introduce a more 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 pattern 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. Our FS-Miner system 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 using two different data sets, 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 scalability with respect 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.

**Keywords:** Frequent Patterns, Traversal Patterns, Sequence Mining, Incremental Mining, Prediction, Prefetching, Web Logs.

## 1 Introduction

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. In web usage mining many types of patterns can be discovered including association rule patterns and traversal patterns. These patterns can be classified based on four main features [2]:

- Whether or not the order of page references in a pattern matters.
- Whether or not duplicate page references (backward traversal[1] or page refresh/reload) are allowed.
- Whether patterns must consist of contiguous page references or they can have gaps.
- Whether or not only maximal patterns are considered. A pattern is maximal when it is not part of another pattern.

In this paper 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 [2]. Furthermore, knowledge of frequent backward traversal is useful for improving the design of web pages [2]. 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 [13]. 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 have four properties: the order of page references in patterns is important, duplicate page references are allowed (backward traversals and page refreshes), patterns consist of contiguous page references, and maximal and non-maximal patterns are allowed.

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, 7, 9] that requires multiple expensive scans of the database, one for each level, to determine which of the candidates are in fact frequent. To address this issue, Han et al. [4] 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.

[1]the same page reference can appear more than one time in the sequence in a non contiguous manner.

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. The problem of incrementally mining for association rules has been studied widely [3, 12]. Parthasarathy et al. [8] introduced an interactive and incremental sequence mining approach using a lattice structure. In their approach, the discovery of frequent sequences is done by traversing the lattice and intersecting subsequences of common suffixes to obtain their support. Their performance study has shown that the incremental capability of their system 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 [8].

Our work is similar to [4] in that we also aim to avoid the expensive candidate generation process, particularly in the presence of large number of items (page references). We propose a frequent sequence tree structure (FS-tree) for storing compressed essential information about frequent sequences. Unlike [4], which aims to discover frequent item sets in which order is not important, our work takes order among page references into consideration. We 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 rest of this paper is organized as follows. Section 2 introduces the FS-tree data structure design and the FS-tree construction algorithm. Section 3 describes the FS-mine algorithm for discovering frequent sequences from the FS-tree structure. Section 4 describes the incremental mining algorithm while Section 5 introduces the interactive capabilities of our system. Section 6 discusses related work. Section 7 discusses our experiment results. Lastly, Section 8 provides some conclusions.

## 2 Frequent Sequence Tree

Let $I = \{i_1, i_2, ..., i_m\}$ be a set of unique items, such as page references. A sequence **Seq** = $<p_1p_2...p_n>$ is an ordered collection of items with $p_i \in I$ for $1 \leq \text{i} \leq \text{n}$. 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 **SID** field and the input sequence field **InSeq**. The order of 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 $<aabc>$.

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[2]) 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 **potential frequent** link. And if $Supp^{link}(h)$ does not satisfy $MSuppC^{link}$ and $MSuppC^{seq}$ we call $h$ a**non-frequent link**.

**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 of the root. Each node $n$ in the FS-tree has a* ***node-name*** *field that represents an item from the input database*[3]*. 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.*

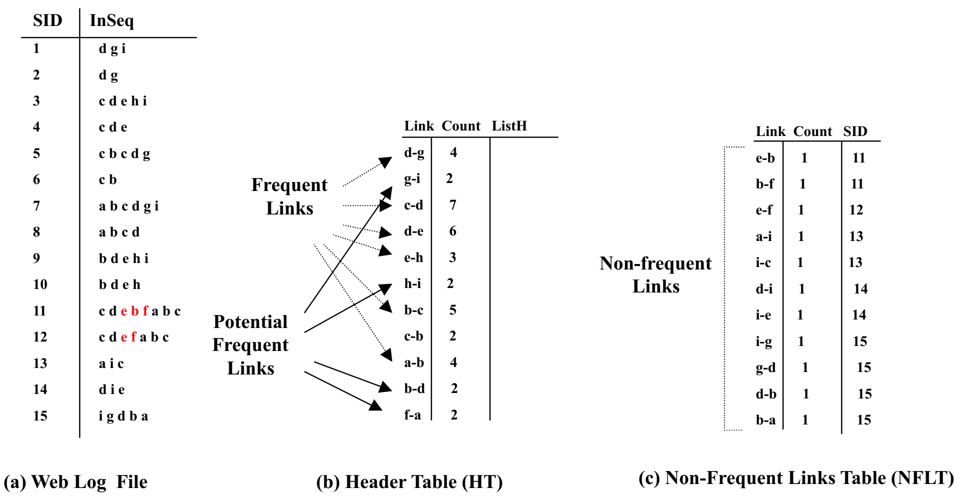
- *A header table $HT$ that stores information about frequent and potential frequent links in the database. Each entry in the header table $HT$ has three fields:* ***Link*** *which stores the name of the link,* ***count*** *stores the count of that link in the database, and* ***listH*** *pointer, which 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* ***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*

[2]Note that this is slightly different from the definition of support ratio in other work [13], which has the same patterns assumptions, that defines this ratio to be the frequency of the sequence to the total number of pages in the database. We think that our ratio is more convenient since it eliminates the effect of sessions with single page reference, in the input web log, on the desired ratio (given that we are interested in patterns of size $\geq 2$).

[3]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 ceratin sequences. We will introduce this structure in more details in the incremental mining section.

*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*[4].



| 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) $HT$ and (c) $NFLT$. Assuming $MSuppC^{link} = 2$ and $MSuppC^{seq} = 3$, Frequent links are those satisfying both support thresholds, Potential Frequent links are those satisfying only $MSuppC^{link}$ and Non-frequent links are those not satisfying any of the two support thresholds.

**FS-tree construction.** Consider the web log file in Figure 1(a). It stores a set of users' sessions where each session has two fields: $SID$ filed that stores the session id and $InSeq$ filed that stores sequence of page references accessed by the user in a certain order. Given such input web log file 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[5], 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. The *insertTree* function inserts the input sequences in the FS-tree starting by the first link in the sequence, frequent links (and potentially frequent links) are stored as edges in tree branches (sharing nodes and edges when possible), until some non-frequent link is encountered, or the input sequence

[4]For optimization, if more than one of these sessions have exactly the same sequence we might store only the ID of one of them along side with their count. For example if the link a-b was non-frequent and if it appeared in three sessions in the database: {5, <*abc*>} {9, <*eabd*>} and {15, <*abc*>} we may store this information in the $NFLT$ as {a-b, 3, {(5:2), (9:1)} } where a-b is the link name, 3 the link count and $\{(5:2),(9:1)\}$ means that a-b appears in a session with SID=5 and in another session that has exactly the same sequence as the one in session 5, and also appeared in a session with ID=9 that had different sequence.

[5]only required for supporting incremental mining

is exhausted. If a non-frequent link is encountered in the inserted sequence we do not insert it, rather, the insertion process is started over again from the root of the tree, with the remaining input subsequence, in a recursive manner. Besides inserting sequences into the **FS-tree** we also maintain the **ListH** linked lists that link different edges in the tree to the header table (HT).

```
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
```

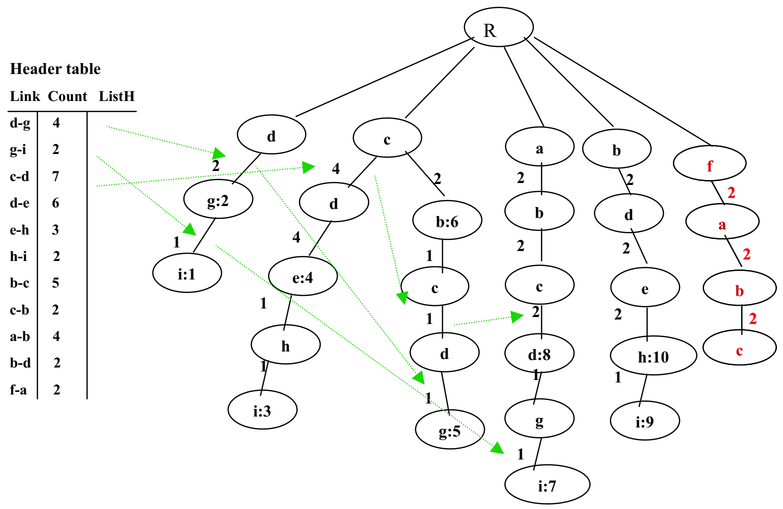
```
Function insertTree (tree root node S, sequence p-P) :
        Updated FS-tree in which all the potentially frequent
        subsequences are inserted .
(1)  If (link p-P    HT) {
(2)      If (S has a child N and N.node-name = p) {
(3)          increment S-N.edge-count by 1
(4)      }Else {
(5)          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[6]. 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.

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

[6]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

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.

## 3 Mining Frequent Sequences from 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 $Supp^{link} \geq MSuppC^{seq} \geq MSuppC^{link}$. These links are stored in $HT$ and are represented in the FS-tree and can be part of frequent sequences.
- *Potential 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.

## 3.1 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. only potentially frequent subsequences of it are to be inserted in the FS-tree.
- If $MSuppC^{link} < MSuppC^{seq}$, the FS-tree is storing more information than required for the current mining task. Hence, the mining algorithm would not care about all sequences encoded in the FS-tree.
- 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 the tree root. The frequency count of that sequence is equal to the count associated with the edge that ends this prefix path. We also can extract certain length of the prefix path based on user maximum pattern size preference. This feature is important for optimizing the mining phase[7].

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.

## 3.2 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 ($HT$):

**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[8]. 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 $(b - d : 3, d - e : 2, e - h : 2)$.

**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

[7]For example if we follow the *ListH* pointer for link $g - i$ from header table in Figure 3 to the second edge and assuming that, at the mining stage, the user is interested in patterns of maximum size of 4, we need to extract only the path prefix (c-d:2, d-g:1, g-i:1) instead of the full path starting from the tree root.

[8]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 potential frequent link (since we do not mine for them).

```
FS-Mine Algorithm
Input: FS-tree root R, and minimum sequence support $MSupp_S$
Output: Frequent sequences
Method:
(1)  Frequent sequences set FSS $\leftarrow \phi$
(2)  (2) For (all links $l_i \in$ HT and $l_i$.count $\geq MinSupp_s$) {
(3)    Conditional sequence set CSS $\leftarrow \phi$
(4)    For (all paths $P_j$ in FS-tree reachable from HT.ListH($l_i$)){
(5)        CSS $\leftarrow$ CSS $\cup$ extract $P_j$, remove last link, and adjust $P_j$.count = last link count }
(6)    Conditional FS-tree CFST $\leftarrow \phi$
(7)    Construct CFST
(8)    For (all sequences $Seq_l$ in CFST){
(9)         FSS $\leftarrow$ FSS $\cup$ concatenate ($Seq_l$, $l_i$)   } }
```

Figure 4: FS-Mine Algorithm.

in the path to the count of the removed $h$ (this gives he frequency of the derived path), Then we remove $h$ from the end of each of the derived paths, since it is a common ending for all of them. 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 $(b-d:2, d-e:2)$.

**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.

**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 $MSuppC^{seq}$. We append $h$ to the end of each of the sequences extracted from the tree to obtain the full length frequent sequences for link $h$. 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.

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.

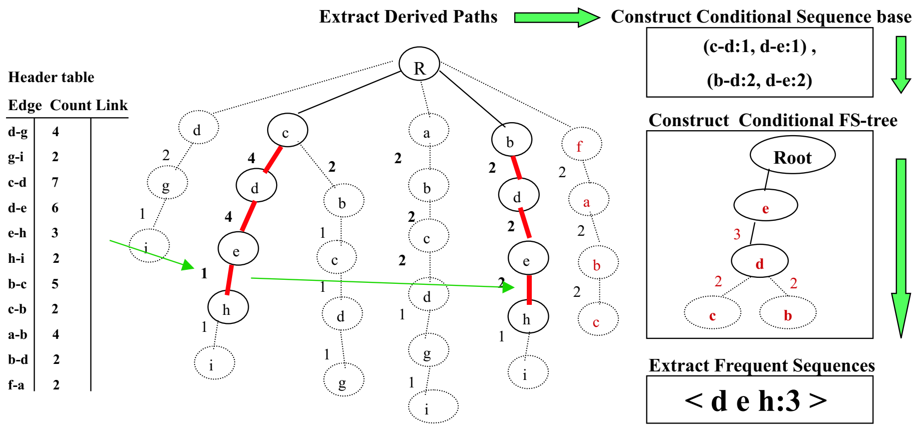


Figure 5: Mining steps for link $e - h$ from the example in Figure 1.

| *Link* | *Derived Paths* | *Conditional Sequence base* | *Conditional FS tree* | *Frequent Sequences generated* |
|---|---|---|---|---|
| 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) | $\phi$ | $\phi$ |
| a-b | (a-b:2), (f-a:2, a-b:2) | (f-a:2) | $\phi$ | $\phi$ |
| b-c | (c-2:2, 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 that satisfy $MSuppC^{seq}$=3.

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

### 4.1 Requirements for Supporting Incremental Mining in the FS-miner

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**

of **seqEnd** is assigned at tree construction time. At the end of input sequence insertion into the tree, we might set **sid** of the node corresponding to the last item in the input sequence to be equivalent to the input sequence id. To assign a new value for **sid** two conditions must be satisfied: if the input sequence is inserted as one piece into the tree without being pruned[9] and if the **sid** does not contain another sequence id already (since we store only one id in this field). For each node with **sid** not equivalent to null we know that the tree branch that starts from the tree root and ends at that node is representing a complete input sequence(s) from the database. 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 [10].

## 4.2 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. Figure 6 shows the incremental FS-tree construction algorithm.

```
Incremental FS-tree construction Algorithm
Input: FS-tree root R and ΔDB, and new minimum link support $MSupp_l$
Output: updated FS-tree
Method:
(1)  Scan ΔDB, and collect counts (+ and -) for links
(2)  Update counts of links in HT and NFLT
(3)  Move links,between HT and NFLT, based on $MinSupp_l$
(4)  For (all links $l_l$ moved from HT to NFLT) {
(5)      Start from $l_l$.ListH pointer
(6)      For (every edge p-P in the FS-tree reachable from $l_l$.ListH)
(7)          Cut the edge p-P
(8)      Call insertTree (R, P) }
(9)  For (all links $l_l$ moved from NFLT to HT) {
(10)     For (every sequence ID in $l_l$.SIDs) {
(11)         Obtain input sequence inpSeq with ID from DB
(12)         Call insertTreeInc (R, inpSeq) } }
(13) For (each record $Rec_i$ in ΔDB get $Rec_i$.inpSeq) {
(14)     If ($Rec_i$ is an insert)
(15)         Call insertTree($Rec_i$.InSeq, R)
(16)     Else if ($Rec_i$ is a delete)
(17)         Call deleteTree ($Rec_i$.InpSeq, R)}
```

```
Function insertTreeInc (tree root node S, sequence p-P) :
Updated FS-tree.
(1)  If (link p-P ∈ HT) {
(2)      If (p-P was originally in HT){
(3)          If (S has no child with name = p) {
(4)              Create node N with N.node-name = p
(5)              Create edge S-N with S-N.edge-count = 1}
(6)      }Else if (p-P was originally in NFLT){
(7)              Node ptr = call deleteTree (R, P)
(8)              add ptr as a child to p }
(9)          If (P is non-empty) {call insertTreeInc(N, P)}
(10) }Else if (link p-P ∈ NFLT) {
(11)     If (P is non-empty) {call insertTreeInc(R, P) } }
```

```
Function deleteTree (tree root node S, sequence p-P) :
Updated FS-tree.
Precondition: counts of links in HT and NFLT are already updated
(1)  If (S has child N and S-N = p-P) {
(2)      Decrement S-N.count, and deleted it if count became 0.
(3)      If (P is non-empty){call deleteTree (N, P) }
(4)  }Else{
(5)      If (P is non-empty) call deleteTree (R, P)}
```

Figure 6: Incremental FS-tree construction.

The algorithm first obtains the count of links in $\triangle$**DB** by performing one scan of $\triangle$**DB** (step 1 in

[9] All links in the sequence are frequent

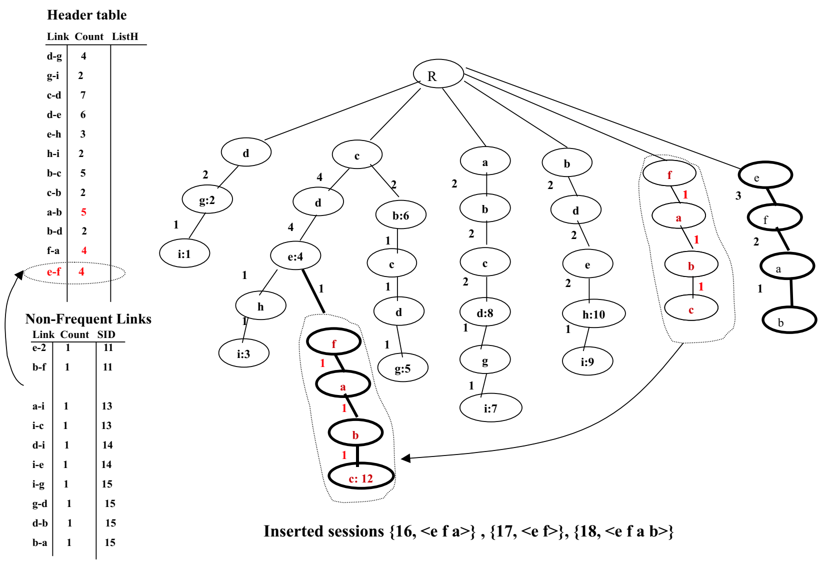
[10] Counts are not shown there for simplicity since they are all equal to 1 for current example.

the algorithm in Figure 6). In step 2, 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. This can be done by following their **ListH** pointer to their edge occurrences in the FS-tree. We remove each edge, then insert the subsequent tree whose root was attached to the removed edge, from the top of the FS-tree, sharing nodes and edges when possible (steps 4 through 8). 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 $insertTreeInc$ (steps 9 through 12). 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[12]. For each of the obtained sequences, the $insertTreeInc$ function traverses the sub-path of it already represented in the FS-tree (staring for the root). When we encounter a link in the inserted sequence that was not frequent before the update and now is frequent (or potentially frequent), we create a new edge and node for it (or share an edge and a node and increment edge's count). 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). This is done by traversing the tree from the top for that subsequence and decrementing the count of any traversed edge. If the count of decremented edge becomes 0, the edge and its subsequent subtree are deleted from the FS-tree. The last phase (steps 14 through 17) inserts (or deletes) input sequences from $\triangle$**DB** into the tree using the $insertTree$ (or the $deleteTree$) function.

**Example 1:** 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 7 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

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

[12]This is needed because if a certain link was non-frequent before the update and became frequent later, during initial tree construction time, the $insertTree$ function has previously broken any input sequence that contained that link at this place and inserted it as subsequences in the FS-tree. But now as that link becomes frequent due to the update, the $insertTreeInc$ will bridge that gap again and put those subsequences together



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 | 5 | |
| b-d | 2 | |
| f-a | 4 | |
| e-f | 4 | |

Non-Frequent Links

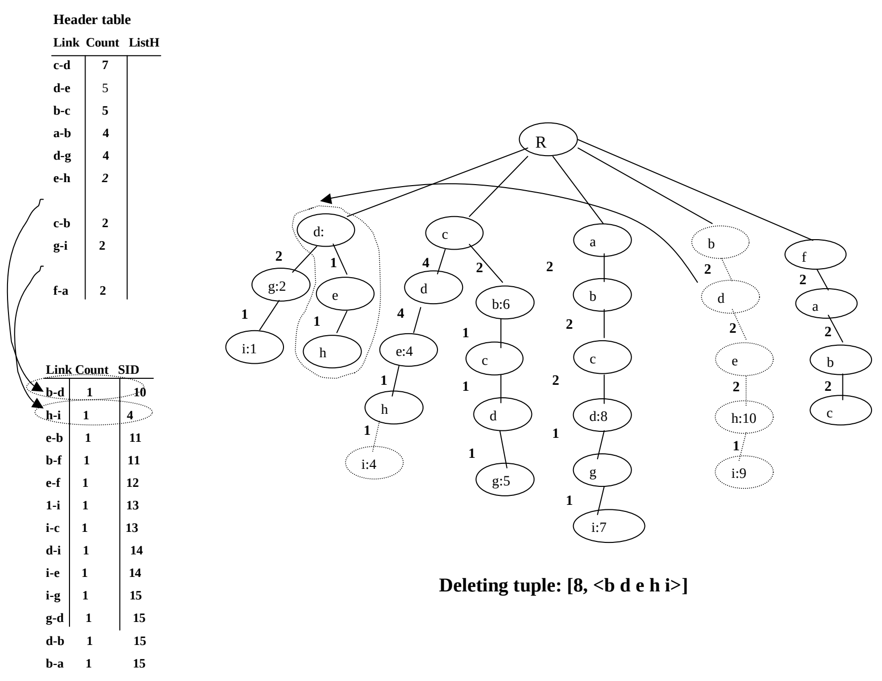
| Link | Count | SID |
|---|---|---|
| e-2 | 1 | 11 |
| b-f | 1 | 11 |
| a-i | 1 | 13 |
| i-c | 1 | 13 |
| d-i | 1 | 14 |
| i-e | 1 | 14 |
| i-g | 1 | 15 |
| g-d | 1 | 15 |
| d-b | 1 | 15 |
| b-a | 1 | 15 |

R
g:2
i:1
e:4
h
i:3
b:6
g:5
d:8
i:7
h:10
i:9
c: 12
Inserted sessions {16, <e f a>} , {17, <e f>}, {18, <e f a b>}

Figure 7: The effect of inserting records to the database in Figure 1.

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.

**Example 2:** As an example for incremental deletes, assume that we delete the tuple {8, <$bdehi$>} from the **DB**. In Figure 8 we note that as a result of deleting that tuple the links $b - d$ and $h - i$ become non-frequent and should not be represented in the FS-tree anymore. The tree pruning step will cause the tree branch ($b - d - e - h - i$) to be cut at $b - d$ and $h - i$ edges, and the part ($b - d - e$) to be inserted at the root of the tree sharing the existing node $d$ and creating nodes $e$ and $h$. also the edge $h - i$ in the tree branch ($c - d - e - h - i$) is pruned. Now the last step is to call the function $deleteTree$ to delete the sequence <$bdehi$>. This will cause the tree branch ($b - d - e$) edges counts to decrement to 1.



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

| Link | Count | SID |
|---|---|---|
| b-d | 1 | 10 |
| h-i | 1 | 4 |
| e-b | 1 | 11 |
| b-f | 1 | 11 |
| e-f | 1 | 12 |
| 1-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 |

Figure 8: The effect of deleting records from the database in Figure 1

## 4.3 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 types of links[13]. 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 9[14]. We categories how the incremental mining algorithm deals with these different transaction cases into four categories:

(1) For transaction of type 1: we mine for those links if they are affected [15].

(2) For transactions of type 2 and 4: we mine for these links.

(3) For transactions of type 3 and 5: we delete previously discovered patterns that include these links.

(4) For transactions of type 6, 7, 8 and 9: we do nothing. The incremental FS-mine algorithm is shown in Figure 10. 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 $MSuppC^{seq}$, if changed due to the update). Then for all links in the **HT** if the link satisfies the new $MSuppC^{seq}$ and if it is of transaction type 2, 4 or of type 1 and affected by the update, the algorithm applies the $FS-mine$ algorithm for these links.

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

[14]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.

[15]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

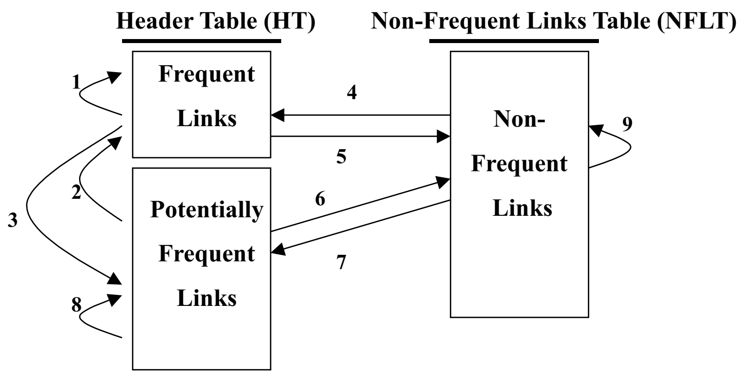


Figure 9: The effect of incremental updates on links in the database

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

Figure 10: Incremental FS-Mine Algorithm.

**Example 3:** 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.

| *Link* | *Derived Paths* | *Conditional Sequence base* | *Conditional FS tree* | *Frequent Sequence generated* |
|---|---|---|---|---|
| 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) | $\phi$ | $\phi$ |

Table 2: Incrementally Mining for link e-h where $MSuppC^{seq}$=3.

**Example 4:** Consider that $\triangle$**DB** denotes a deletion of the record with ID = 8 from the web file in Figure 1. In this case the affected links are: $b-d$, $d-e$, $e-h$, and $h-i$. And since $b-d$, $e-h$ and $h-i$ are no longer supporting the $MSuppC^{seq}$ (assuming 3) we delete any frequent sequences previously discovered that contain any of those links. Namely from the frequent sequences previously generated (and shown in Table 1) we delete the sequence $<deh:3>$. Now we look in the **HT** for those link that satisfy the $MSuppC^{seq}$ and of type 2, 4, or 1 (and affected by the update). Only links $d-e$ and $e-h$ are satisfying this criteria so we apply the mining steps for each of them.

## 5 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 allow 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 increased 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 patters he is interested in discovering. In this case the system does not use the input database, it only uses the FS-tree to extract the frequent sequences for the required size.

Now we give an example for lowering the $MSuppC^{seq}$. The frequent sequences shown in Table 1 were generated based on $MSuppC^{seq} = 3$. Assume that $MSuppC^{link} = 2$ was small enough to satisfy most of the expected changes to the system $MSuppC^{seq}$. And that the user later on sets $MSuppC^{seq}$ to 2. In this case, and since our FS-tree already has all information about links and sequences with minimum frequency of 2, we can directly apply the FS-mine algorithm and obtain the result shown in Table 3 without the need for re-scanning any part of the input database.

## 6 Experimental Results

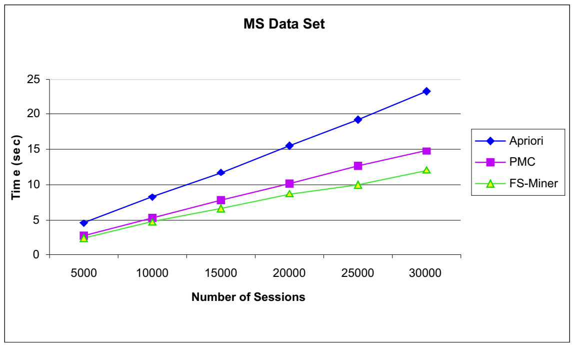
We use two data sets to test our system, the Microsoft Anonymous Web Data Set and the MSNBC Anonymous Web Data Set, both obtained from [5]. Each data set consists of a collection of sessions where each session has a sequence of page references. The Microsoft anonymous data set has 32711 sessions, each session contains from 1 up to 35 page references. The MSNBC data set has 989818 sessions. A session contains

| *Link* | *Derived Paths* | *Conditional Sequence base* | *Conditional FS-tree* | *Frequent Sequence generated* |
|---|---|---|---|---|
| f-a | (f-a:2) | $\phi$ | $\phi$ | $\phi$ |
| h-i | (c-d:d,d-e:4 ,e-h:1,h-i:1),<br>(b-d:3,d-e:3 ,e-h:2,h-i:1) | (c-d:1,d-e:1 ,e-h:1)<br>(b-d:1,d-e:1 ,e-h:1) | (d-e:2,e-h:2) | $<dehi:2>$ |
| g-i | (d-g:2,g-i:1)<br>(a-b:2,b-c:2 ,c-d:2,d-g:1,g-i:1) | (d-g:1),<br>(a-b:1,b-3:1 ,c-d:1,d-g:1) | (d-g:2) | $<d-g-i:2>$ |
| c-b | (c-b:2) | $\phi$ | $\phi$ | $\phi$ |
| b-d | (b-d:2) | $\phi$ | $\phi$ | $\phi$ |
| e-h | (c-d:4, d-e:4, e-h:1) ,<br>(b-d:3, d-e:2, e-h:2) | (c-d:1, d-e:1) ,<br>(b-d:2, d-e:2) | (d-e:3),<br>(b-d:2, d-e:2) | $<deh:3>$<br>$<bdeh:2>$ |
| d-g | (d-g:2), (c-b:2, b-c:1,c-d:1,d-g:1),<br>(a-b:2,b-c:2 ,c-d:2,d-g:1) | (c-b:1, b-c:1,c-d:1),<br>(a-b:1,b-c:1 ,c-d:1) | (b-c:2 ,c-d:2) | $<bcdg:2>$ |
| a-b | (a-b:2), (f-a:2, a-b:2) | (f-a:2) | (f-a:2) | f-a-b :2 |
| b-c | (c-b:2, b-c:1), (a-b:2,b-c:2),<br>(f-a:2, a-b:2,b-c:2) | (c-b:1), (a-b:2),<br>(f-a:2, a-b:2) | (a-b:4)<br>(f-a:2, a-b:2) | $<abc:4>$<br>$<fabc:2>$ |
| d-e | (c-d:4, d-e:4),<br>(b-d:3, d-e:2) | (c-d:4),<br>(b-d:2) | (c-d:4)<br>(b-d:2) | $<cde:4>$<br>$<bde:2>$ |
| c-d | (c-d:4),<br>(c-b:2, b-c:1,c-d:1),<br>(a-b:2,b-c:2 ,c-d:2) | (c-b:1, b-c:1),<br>(a-b:2,b-c:2) | (b-c :3)<br>(a-b:2, b-c:2) | $<bcd:3>$, $<abcd:2>$ |

Table 3: Mining for $MSuppC^{seq}$=2.

from 1 up to up to several thousands of page references [16]. The main difference between the two data sets of interest to us 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: a variation of Apriori algorithm [1] for sequence data [17] and the *PathModelConstruction* algorithm [11]. We have implemented the three systems in Java in a Windows environment. We ran the experiments on a PC with a 733 MHz Pentium processor and 512 MB of RAM.



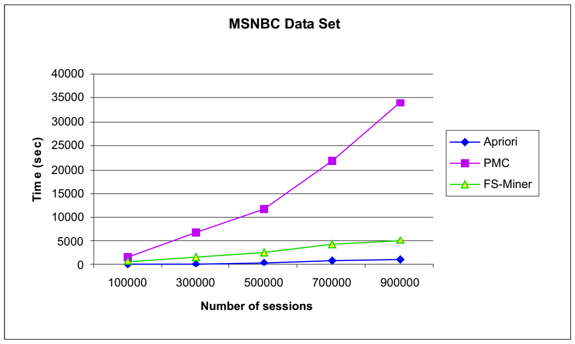
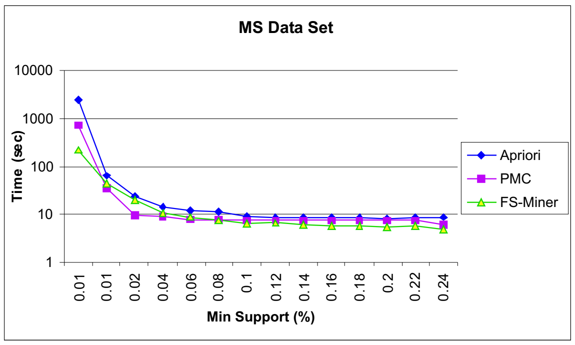


Figure 11: Scalability with number of input sessions

We have conducted three different experiments using both data sets. We first tested the scalability of our approach with respect to changes in input database size. Figure 11 shows that our system, and the other two systems, scale linearly in 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 set) while Apriori tends to perform

[16]We 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 experimental time

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



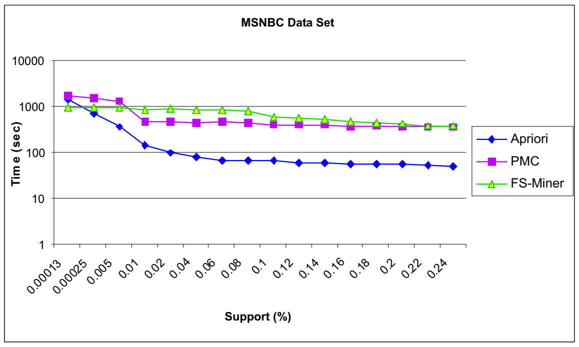
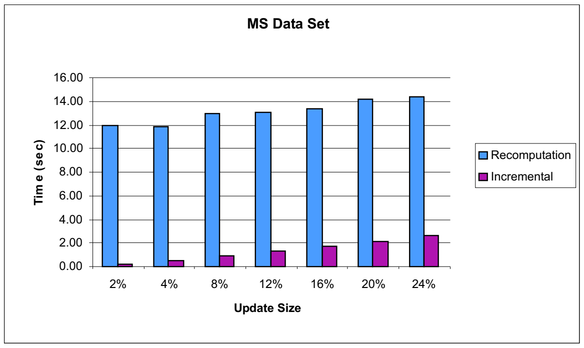


Figure 12: Scalability with support threshold



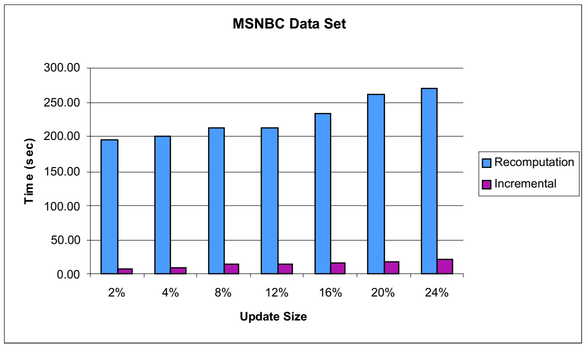


Figure 13: Scalability with support threshold

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 in hand, our system is also maintaining as a byproduct the FS-tree that can later be used for incremental and interactive operations.

We also test the scalability of the system with respect to a decrease of the support threshold level. Figure 12 shows that our system scales better with a decrease of support level. In fact our system shows a very smooth response time to the decrease in the support level unlike the other two systems that experience a dramatic increase in cost when they hit lower support values. This implies that even if we choose to utilize a low $MSuppC^{clink}$, 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 13 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.

## 7 Related Work

Nanpoulos et al. [6] 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 [10] 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. [14] 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* [11] and then used to improve the GDSF caching replacement algorithm. These association rules assumes order and adjacency information among page references. The left hand side of the association rule is a substring of length $n$ (called $n$-gram substring), and is obtained by scanning through all substrings ranging between 1 and $n$ in each user session and pruning substrings that do not satisfy a pre-defined minimum support. Like us, they assume contiguous page references in sequence patterns.

Han et al. [4] 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. This work also used a divide-and-conquer method to decompose the mining task into a set of smaller tasks that reduce the search space. The proposed technique discovers un-ordered frequent item sets. However, is does not support the type of sequences we are interested in.

Parthasarathy et al. [8] 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. Sequence supports are obtained by performing intersection between different nodes in the lattice and obtaining count supports from the intermediate vertical database. Due to the size of the intermediate vertical database and lattice that together typically exceeds memory limits, this process is broken into smaller processes by forming suffix-based equivalence classes. Each class is brought to the memory and processed independently. Similar in spirit to [8], we store in the FS-Tree additional data as that reduces the work required at later stages although we use very different data structures and algorithms to achieve that.

Xiao and Dunham [13] proposed an incremental and adaptive algorithm for mining for traversal patterns.

This work relies on a generalized suffix tree structure where all sequences in the database and their suffixes are inserted into it. This tree grows quickly in size. Whenever the size of the tree reaches the size of the available memory during tree construction, time 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 [13], the adaptive mining here means that the system is adaptive to changes in user-specific parameters.

## 8 Conclusion

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. The FS-miner calculates the incremental effect of these updates directly from the updated FS-tree. Our approach also allows interaction with the user in the form of changes to the system minimum support, and in most cases we can satisfy these requests without having to use the original database. 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 relative to the size of the original database.

**Acknowledgments.** Many people deserve thanks for making the UCI KDD archive a success. Foremost among them are the donors and creators of the databases.

## References

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Intl. Conference on Very Large Data Bases (VLDB)*, pages 487–499, 1994.

[2] M. H. Dunham. *Data Mining: Introductory and Advanced Topics.* Prentice Hall, 2003.

[3] R. Feldman, Y. Aumann, A. Amir, and H. Mannila. Efficient algorithms for discovering frequent sets in incremental databases. In *Proc. of SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery (DMKD)*, pages 59–66, 1997.

[4] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *SIGMOD Conf.*, pages 1–12, May 2000.

[5] 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.

[6] 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.

[7] R. Ng, L. Lakshmanan, J. Han, and Pang. Exploratory mining and pruning optimization of constrained association rules. In *SIGMOD Conf.*, pages 13–24, 1998.

[8] S. Parthasarathy, M. J. Zaki, M. Ogihara, and S. Dwarkadas. Incremental and interactive sequence mining. In *Intl. Conference on Information and Knowledge Management (CIKM)*, pages 251–258, 1999.

[9] 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.

[10] R. Srikant and R. Agrawal. Mining sequential patterns: Generalizations and performance improvements. In *Intl. Conf. on Extending Database Technology (EDBT)*, pages 3–17, 1996.

[11] Z. Su, Q. Yang, Y. Lu, and H. Zhang. Whatnext: A prediction system for web request using n-gram sequence models. In *Intl. Conf. on Web Information Systems Engineering (WISE)*, pages 214–221, 2000.

[12] S. Thomas, S. Bodagala, K. Alsabti, and S. Ranka. An efficient algorithm for the incremental updation of association rules in large databases. In *Intl. Conf. on Knowledge Discovery and Data Mining (KDD)*, pages 263–266, 1997.

[13] Y. Xiao and M. H. Dunham. Efficient Mining of Traversal Patterns. *Data and Knowledge Engineering*, 2(39):191 – 214, 2001.

[14] Q. Yang, H. H. Zhang, and I. T. Y. Li. Mining web logs for prediction models in WWW caching and prefetching. In *Intl. Conf. on Knowledge Discovery and Data Mining (KDD)*, pages 473–478, 2001.