Data Min Knowl Disc
DOI 10.1007/s10618-009-0133-6

Fast incremental mining of web sequential patterns with PLWAP tree

C. I. Ezeife · Yi Liu

Received: 1 September 2007 / Accepted: 13 May 2009
Springer Science+Business Media, LLC 2009

Abstract Point and click at web pages generate continuous data sequences, which flow into the web log data, causing the need to update previously mined web sequential patterns. Algorithms for mining web sequential patterns from scratch include WAP, PLWAP and Apriori-based GSP. Reusing old patterns with only recent additional data sequences in an incremental fashion, when updating patterns, would achieve fast response time with reasonable memory space usage. This paper proposes two algorithms, RePL4UP (Revised PLWAP For UPdate), and PL4UP (PLWAP For UPdate), which use the PLWAP tree structure to incrementally update web sequential patterns efficiently without scanning the whole database even when previous small items become frequent. The RePL4UP concisely stores the position codes of small items in the database sequences in its metadata during tree construction. During mining, RePL4UP scans only the new additional database sequences, revises the old PLWAP tree to restore information on previous small items that have become frequent, while it deletes previous frequent items that have become small using the small item position codes. PL4UP initially builds a bigger PLWAP tree that includes all sequences in the database using a tolerance support, t, that is lower than the regular minimum support, s. The position code features of the PLWAP tree are used to efficiently mine these trees to extract current frequent patterns when the database is updated. These

Responsible editor: Eamonn Keogh.

C. I. Ezeife (✉) · Y. Liu
School of Computer Science, University of Windsor, Windsor,
ON N9B 3P4, Canada
e-mail: cezeife@uwindsor.ca
URL: http://www.cs.uwindsor.ca/∼cezeife

Y. Liu
e-mail: woddlab@uwindsor.ca

approaches more quickly update old frequent patterns without the need to re-scan the entire updated database.

Keywords Incremental mining · Sequential mining · Frequent patterns · Data streams · PLWAP tree · Scalability

1 Introduction

The goal of incremental mining of web sequential patterns is to generate current frequent patterns for the updated database (consisting of both old and incremental data) using mostly only the incremental (or newly added) data and previously mined frequent patterns. When data (like web access patterns) are inserted into a database (like web log), some previous frequent patterns may no longer be interesting, while some new interesting patterns could appear in the updated database. Incremental mining of web sequential patterns is beneficial because it may:

1. Scale web sequential mining to large datasets by speeding up processing time.
2. Be effective for mining fast changing and highly dynamic data environment like in stream processing environment requiring fast and real time responses.
3. Be an approach for more efficient utilization of I/O, memory and CPU resources that may be scarce in some applications.
4. Be more effective for detecting similarities and differences in versions of data and their patterns for purposes of predicting and detecting such phenomena as intrusions.

A web access sequential database is a special case of a general sequential database, where every event e_i in a web access sequence is a single event (or item) and not a set of events as is the case in a general sequential database. Events in a web access sequential database could represent, for example, web pages accessed by users stored in web log data, products accessed in an E-Commerce web site. In such web applications, since each click is on a page, single element sets form members of each sequence. While a general sequence looks like $\langle\{a\}, \{a, b, c\}, \{e, f\}\rangle$, a web access sequence contains only sequences like $\langle a, a, b, c, e, f\rangle$. Prominent and good generic sequence mining algorithms include AprioriAll (Agrawal and Srikant 1995), GSP (Srikant and Agrawal 1995), Suffix Tree (Wang 1997), SPADE (Zaki 2000), FreeSpan and PrefixSpan (Pei et al. 2001). The few algorithms designed specifically for single-element set sequences suitable for web navigational sequences include: WAP-tree (Pei et al. 2000), PLWAP-tree (Ezeife and Lu 2005; Ezeife et al. 2005; Lu and Ezeife 2003), and FS-Miner (El-Sayed et al. 2004). Web sequence mining requires an incremental algorithm. The PLWAP-tree (Ezeife and Lu 2005) sequential miner is a a good candidate for incremental web sequential mining because with the use of its position code features, it stores relevant parts of the original database in a comparatively compressed tree structure, and it does not require multiple scans over the entire original data. The PLWAP's position code labels for its nodes can also be used to maximize the reuse of already mined information for incremental maintenance purposes.

1.1 Problem definition

Given (1) an initial web access sequential database, (represented as **DB**), consisting of a number of sequences S_1, S_2, …, S_m, where each sequence S_i is in the form $e_1 e_2 \ldots e_k$, for $e_i \in E$ (set of events) ($1 \leq i \leq numberofevents$), (2) the minimum support threshold, s (throughout the paper s is given as either a percentage $s\%$ of the number of records or simply as the number of records s), (3) the initial mined frequent patterns, FP^{DB}, (4) newly inserted records from the database (called the incremental data **db**), the problem of incremental web usage sequential pattern mining is that of finding all frequent patterns (FP') in the updated database, **U**, (old DB + incremental db), which have supports greater than or equal to updated minimum support count s', using mostly only the new incremental data *db* with previously generated frequent patterns FP^{DB}.

1.2 Background

Most web sequential pattern miners perform their pattern discovery tasks through association rule mining techniques applied on a web log sequential database table or file. Association rule is an implication of the form $X \rightarrow Y$, where X and Y are sets of items (e.g., sets of web pages) and $X \cap Y = \emptyset$, Han and Kamber (2001). The support of this rule is the percentage of transactions that contain the set $X \cup Y$, while its confidence is the percentage of "X" transactions that also contain "Y" itemsets. In traditional association rule mining, all sets of items with support higher than or equal to a specified minimum support are called large or frequent itemsets. The first stage of rule mining consists of generating all combinations of itemsets (frequent itemsets) from the database that have support greater than or equal to the minimum support (specified by the user). During the second stage of rule mining, association rules are generated from each frequent itemset, and only rules with confidence greater than or equal to the minimum confidence are retained. For web sequential pattern mining, input to the mining process is a sequential database (for pre-processed web log sequences (Berendt and Spiliopoulou 2000)) example of which is in Table 1, columns one, (for transaction ID) and two (for the web access sequence record), where items {a, b, c, d, e, f, g, h} represent access site pages.

Typically, web log data and other data stream applications (e.g., credit card database, stock analysis data, cell phone database) contain thousands and millions of records

Table 1 The example database transaction table (DB) with frequent sequences

TID	Web access seq.	Frequent subseq with $s = 50\%$	Frequent subseq with $t = 0.6s = 30\%$
100	abdac	abac	abac
200	aebcace	abcac	aebcace
300	baba	baba	baba
400	afbacfc	abacc	afbacfc
500	abegfh	ab	abef

that continuously grow. Each transaction in the database contains a sequence of database events (or items), corresponding to the sequence of web pages visited by each user id. Just as an itemset X is called an n-itemset if it contains n items, an n-sequence has n items (events) in its sequence. For example, the 3-itemset *abc* is a frequent itemset in the five record database of Table 1 because it has a support of 60%, which is greater than the assumed user given minimum support of 50%. An example of a sequential pattern from this database is "if web page a is accessed, it is often followed by an access to web page b and then web page c". This sequence *abc* has a support of 60% since it is present in three of the five transactions. While traditional association rule mining finds patterns (presence of a set of items) among database transactions, sequential pattern mining finds sequential patterns S (presence of a set of items in a time-ordered sequence) among database transactions, where an item may re-occur in the same sequence. Items in a sequence do not necessarily need to be consecutive. Every item has only one support count from each database transaction access sequence it is present in. In the above example, set of events $E = \{$a,b,c,d,e,f,g,h$\}$ and a sequence S is aebcace. An access sequence $S' = e'_1 e'_2 \ldots e'_l$ is called a subsequence of another access sequence, $S = e_1 e_2 \ldots e_k$, if and only if S is a super-sequence of S', denoted as $S' \subseteq S$, if and for every event e'_j in S', there is an equal event e_q in S, while the order that events occur in S is the order of events in S'. For example, with $S' =$ ab, $S =$ babcd, S' is a subsequence of S and ac is a subsequence of S, although there is b occurring between a and c in S. In S = babcd, while *ba* is the prefix subsequence of *bcd, bcd* is the suffix sequence of *ba*. The candidate 1-item list C_1 showing support of each item and the frequent 1-items F_1 are used during mining.

1.3 Related work

Work on mining sequential patterns can be classified into Apriori-based and Non-Apriori based algorithms. The Apriori based algorithms include AprioriAll (Agrawal and Srikant 1995), GSP (Srikant and Agrawal 1995), the PSP (Masseglia et al. 1999), the G sequence (Spiliopoulou 1999), SPADE (Zaki 2000) and the graph traversal (Nanopoulos and Manolopoulos 2001) algorithms. The Non-Apriori based algorithms include Suffix tree (Wang 1997), Manolopoulos (Nanopoulos and Manolopoulos 2000, 2001), PrefixSpan (Pei et al. 2001). Algorithms specifically for mining web sequential patterns include WAP-tree (Pei et al. 2000), PLWAP-tree (Ezeife and Lu 2005; Ezeife et al. 2005; Lu and Ezeife 2003), and FS-Miner (El-Sayed et al. 2004). The Apriori-based algorithms make multiple passes over data and during each iteration i, compute candidate i-itemsets by joining large $(i-1)$-itemset (L_{i-1}) with itself and pruning candidates having infrequent subsequences, before scanning the database for support, for all candidate itemsets in the list, in order to compute the next frequent itemsets L_i. Unlike the Apriori-based approaches, the non-Apriori techniques use graphs, pattern-growth trees (Han et al. 2004) and database projections to avoid level wise candidate itemset generations. Some hybrid web sequential pattern mining approaches combining Apriori style techniques with other non-Apriori techniques like pattern-growth, have recently emerged and include (Tang and Turkia 2007). The most related web

sequential mining technique, the PLWAP algorithm is summarized before discussing the incremental mining techniques.

1.3.1 The PLWAP algorithm

Example 1.1 Given a sample database shown in Table 1 (columns 1 and 2), a minimum support threshold of 50% or three transactions, use the PLWAP algorithm to mine all frequent patterns

Solution 1-1 The PLWAP algorithm first computes frequent 1-items, which it uses to create frequent sequences (shown in column 3 of Table 1) for each database sequence by deleting all non-frequent items from the sequence. Then, it proceeds to build the PLWAP tree with these frequent sequences before mining the frequent patterns from the PLWAP tree. Details of these three steps in the algorithm are discussed further as follows:

Step 1. The PLWAP algorithm computes frequent 1-items from the database transactions (column 2 of 1) as $F_1 = \{a{:}5, b{:}5, c{:}3\}$, listing each event with its occurrence. It generates frequent sequences from each transaction in the second column of Table 1 by deleting all small events (not in F_1 list) from it, to get column 3 of Table 1.

Step 2. Using the frequent sequences (column 3 of Table 1), it builds the PLWAP tree by inserting each sequence from Root to leaf node, where each node is labeled as (label of the node:count of the node:position code of the node). The count of each node (e.g., *a*) is incremented by one each time an event of this node type is inserted at that node, and a new child of a parent node is created if there is no path from *Root* that corresponds to the sequence being inserted in the tree. Common prefix sequences share paths and counts, and uncommon suffix sequences branch off towards the leaves of the tree. In defining the position code of a node (e.g., the *a node* which is the left child of the *Root* has position code of 1), the PLWAP algorithm applies the rule that the Root of the tree has a null position code, but every other node has a position code that is equivalent to appending '1' to the position code of this node's parent if this node is the leftmost child of its parent, otherwise, its position code is obtained by appending '0' to the position code of its nearest left sibling. After building the tree, a pre-order traversal mechanism (visit root, visit left subtree, visit right subtree) is used to add a pre-order linkage on the tree for all frequent 1-items. Dashed lines starting with each frequent F_1 item, are used to show the pre-order linkages between nodes of this F_1 item type. Thus, to build the PLWAP tree of Fig. 1 for the example database, the frequent subsequence of each transaction at minsuppport of 50% shown on column 3 of Table 1 is inserted one after the other from *Root* to leaf. Starting with the first frequent sequence "abac", node a:1:1 is first inserted as the left child of *Root* with count of 1 and position code of 1. Then, node b:1:11 is the left child of node a:1:1. The complete sequence branch for the first transaction is a:1:1, b:1:11, a:1:111, c:1:1111. The second sequence abcac is inserted as the branch a:2:1, b:2;11, c:1:1110, a:1:11101, c:1:111011. The rest of the transaction frequent sequences are inserted in a

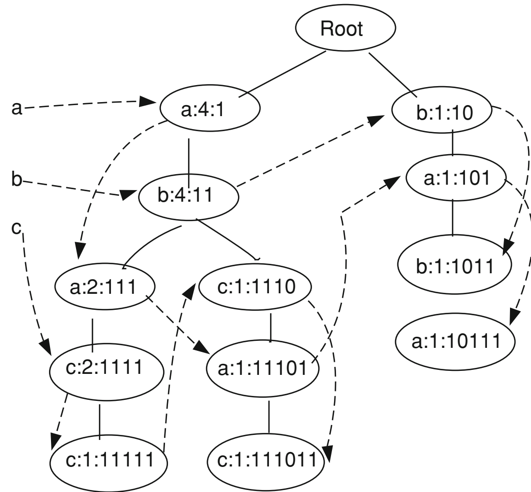

Fig. 1 The PLWAP tree for the example database

similar fashion to obtain the PLWAP tree. Then, using the F_1 header list of {a, b, c}, the tree is traversed pre-order fashion (root, left, right) to link each header event with nodes of its type for look-ahead mining purposes.

Step 3. Mine the PLWAP tree to generate frequent pattern, FP^{DB} by following the header linkage of the first frequent item, like "a", and obtaining the support of this prefix subsequence "a" as the sum of the counts of all first "a" nodes in the current a:suffix root set (i.e., a:4:1 and a:1:101) on different branches of the tree at this level of the tree being mined. If the sum of the counts of these first *a* nodes on different branches of the tree at the level of the tree under consideration is greater than or equal to the support of 3, we include this sequence in the FP^{DB} frequent pattern list. Next, we shall continue to check down the suffix trees under use to see if new sequences (aa, ab, ac) are also frequent, having already found prefix frequent patterns FP^{DB} as {a}. To check for the support of aa, we obtain the a:suffix root set that are children of the previous a:suffix root set considered, which are the nodes (a:2:111), (a:1:11101), (a:1:10111). The a-header link serves the purpose of tracking and constructing these root sets during mining. Since the sum of these "a" counts is 4 and greater than 3, "aa" is included in the list of frequent sequences. We further look for "a" nodes in suffix tree that are children of the above already used a:suffix roots to check if the sequence "aaa" is also frequent. Since no other "a" nodes are further down in the tree, "aaa" is not frequent. Backing up to the suffix root set with three a:nodes given above, we look lower down in the tree to see if "aab" is frequent. It is not. However,"aac" is frequent because we can find the c:suffix root set as (c:2:1111), (c:1:111011), which totals 3. Now, that we started with checking a, aa, aaa, we need to back up to each root set, to check for prefix growing patterns having other frequent 1-item events like aab, aac, ab, ac, aba, abb, abc, etc. Process continues in a similar fashion recursively. After checking for frequent patterns starting with item "a", it

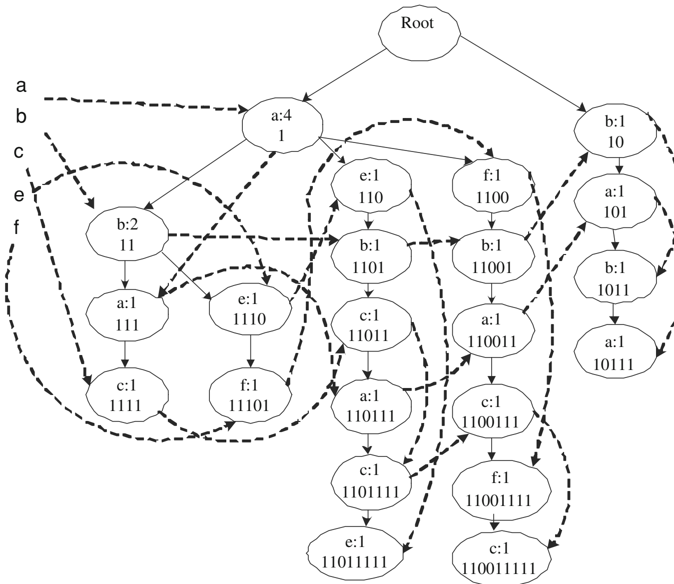

Fig. 2 The PL4UP tree with tolerance support $t\% = 30\%$ for the example database

starts from the Root node of the tree again using "b" header linkage, to look for patterns beginning with "b", "ba", "baa" and so on. Then, using the "c" header linkage, it checks for patterns beginning with "c". After checking all patterns, the list of FP^{DB} mined based on the support of 3 is: FP^{DB} = {a:5, aa:4, aac: 3, ab:5, ac:3, aba:4, abac:3, abc:3, b:5. ba:4, bac:3, bc:3, c:3}. The proposed incremental web sequential mining techniques utilize the position code features of the PLWAP algorithm to perform efficient and effective restoration of the database small events and maintain tree frequent sequences without re-scanning the entire updated database. Note that the PLWAP tree structure is a general tree with any number of branches depending on the transactions and the minimum support. For example, if the same database, DB of Table 1 is mined at a lower minimum support of 30% or 2 transactions, then $F_1 = \{a, : 5, b : 5, c : 3, e : 2, f : 2\}$ and small $S_1 = \{d : 1, g : 1, h : 1\}$. This causes a PLWAP tree with more branches to be built from the frequent sequences abac, aebcace, baba, afbacfc, abef (Fig. 2).

1.3.2 Existing incremental mining techniques

Many of the generic sequential mining algorithms have incremental sequential mining versions proposed and these incremental mining algorithms include: Suffix Tree Approach (Wang 1997; Wang and Tan 1996), ISL (an incremental algorithm based on SPADE) (Parthasarathy et al. 1999), ISE (incremental sequential algorithm based on GSP) (Masseglia et al. 2003), IncSpan (incremental sequential algorithm based on the PrefixSpan) (Cheung et al. 2004; Nguyen et al. 2005), GSP+ and MFS+ (incremental sequential mining algorithms based on GSP and its general form MFS) (Zhang et al.

2002; Kao et al. 2005; Zhang et al. 2002), FS-Miner (an incremental sequential update based on the FS-Tree algorithm) (El-Sayed et al. 2004). Some recent incremental sequential mining algorithms are based on a hybrid of approaches including incremental maintenance of clusters (Liu et al. 2005), recording positions of candidate traversal sequences (Yen and Lee 2006), applying Markov chain model (Ou et al. 2008) and interactive semantic information (Lee and Yen 2008). The incremental sequential mining algorithms (RePL4UP and PL4UP) being proposed are based on the efficient PLWAP web sequential miner. All the incremental frequent sequential pattern mining algorithms listed above, with the exception of IncSpan and FS-Miner, are Apriori-based. The general technique employed by the Apriori-based incremental mining algorithms is consistent with the method used in Cheung et al. (1996, 1997) for their incremental non-sequential FUP algorithm, where the goal of FUP is to first filter out all previous frequent itemsets not likely to be frequent in the updated database by examining only the incremental database, db. Then, it scans the original database for all items it cannot deduce their support because they were small previously. Details of the more related algorithms are provided next.

1.3.2.1 Suffix tree approach With the suffix tree approach for computing sequential pattern (Wang 1997; Wang and Tan 1996), a sequence S is taken as a set of records and not a record. The sequence S is mapped to a suffix tree in breadth (rather than depth) fashion, where a sequence with n events is represented by a tree with n branches at depth 1, each for a suffix of the sequence. The algorithm assumes that the suffix trees are materialized, implying availability of large memory, since representing a number m of sequential records as in our example, would require constructing m suffix trees. When updates occur, the algorithm scans only affected parts of the tree. Mining is done recursively post-order fashion. Our technique has a different and more natural representation for sequences that facilitates mining, and requires less storage and computation time for the one PLWAP tree than this method.

1.3.2.2 ISM algorithm ISM (Parthasarathy et al. 1999) is an Apriori-like algorithm, which has two phases. It separates the sequence into two sets: the Frequent Set (FS), which denotes the set of all frequent sequences in the updated database, and Negative Border (NB), which is the collection of all sequences that are not frequent but have their subsequences frequent. Phase 1 is for updating the supports of elements in NB and FS, Phase 2 is for adding to NB and FS beyond what was done in Phase 1. It still needs to rescan the entire updated database many times if previous small items become large after database update. Unlike the ISM, our proposed incremental techniques do not incur the cost of database transposition, and scanning of the entire old database when small items become large in updated database, and do not engage in level-wise generation of candidates or sequence list computation.

1.3.2.3 GSP+ algorithm The GSP+ (Zhang et al. 2002; Kao et al. 2005) is an incremental version of the GSP algorithm, which updates frequent patterns in the updated database (DB') by minimizing the scanning of the unchanged part of the old database (DB−). It utilizes support counts of old frequent patterns and the concept of maximum bound frequent patterns, which is the longest (with highest number of items) frequent

pattern of a negative border sequence (now no longer frequent). For example, while *aa* is frequent, *aab* is not frequent, making *aab* a negative border sequence and *aa* is its maximum bound frequent pattern. The idea is to apply pruning rules to eliminate candidate sequences not likely to be frequent in the updated database, DB'. GSP+ first scans the updated database as partitions of deleted database, db−, inserted database, db+ and unchanged database, DB− so that it computes updated DB as DB− ∪ db+ to db−. From this relationship, available frequent pattern count, maximum bound frequent patterns, it goes through iterations of computing the entire candidate itemset as done by the GSP algorithm, then, it eliminates a lot of candidate itemsets that cannot be frequent before scanning either the incremental database db+ or for cases of small items being frequent, scanning the old DB−. Unlike the GSP+, our technique avoids scanning the entire original old database even when small items become frequent. In addition, the massive iterative CPU intensive and I/O intensive level-wise candidate generation, pruning and testing are avoided as shown in experimental comparisons section.

1.3.2.4 MFS+ algorithm The MFS+ algorithm (Zhang et al. 2002; Kao et al. 2005), is another GSP-based incremental sequential algorithm. It is based on a more general version of the GSP algorithm called MFS (maximal frequent sequences). The MFS+ is like the GSP+ except that rather than generating candidate i-itemset during each iteration, it estimates the MFSS (maximal frequent sequence set), then, it applies the same types of pruning rules applied in the GSP+ to eliminate MFSSs in the list, which it confirms either frequent or not, without having to scan the unchanged database, DB−. The main advantage of this algorithm over its GSP+ counterpart is that the initial estimate and the use of maximal sequences enable it cut down on the number of iterations drastically and thus, reduce the number of database scans. Our approach is different from this algorithm and is still superior in its technique of not scanning the entire original old database when small items become large, during which time this algorithm would scan the database.

1.3.2.5 IncSpan incremental sequential mining algorithm IncSpan (Cheung et al. 2004) is a system that buffers semi-frequent patterns (having support less than minimum support, but greater than or equal to a specified tolerance support), and performs incremental mining with PrefixSpan sequential mining technique. IncSpan's idea is to reduce the need to scan the old unchanged database DB−, by getting the support counts of newly inserted sequences from the buffered semi-frequent sequences (SFS). The algorithm, like many other incremental algorithms, partitions the updated database, DB' into sets of newly inserted sequences, db+, newly deleted or altered sequences (db−), and unchanged database, DB−. Whenever there is an update, it scans db+ to update support of frequent and semi frequent sequences (FS) and (SFS). Six types of changes can be identified in patterns after a database update as: case 1 are patterns that are frequent in old DB and still frequent in updated DB', case 2 is for patterns that are semi frequent in old DB but frequent in DB', case 3 is for patterns that are semi frequent in both old DB and updated DB', case 4 is for patterns that are new in the updated DB', case 5 is for patterns that are small in old DB but frequent in updated DB' and case 6 is for patterns that are small in old DB but semi frequent in updated DB'. Only for cases 1–3, does this algorithm not scan database, but deduces the support

counts from available frequent patterns. For case 4, it scans incremental database db+ and proceeds with recursive prefix database projections and mining as done by PrefixSpan. For cases 5 and 6, the bigger database, old unchanged DB− has to be scanned and the database projections are done when mining recursively the usual PrefixSpan fashion. Our proposed algorithms, unlike IncSpan, would avoid scanning the entire old database for cases like 5 and 6 above and does not incur the cost of intermediate database projections. However, just as IncSpan buffers its already found frequent and semi-frequent patterns, the proposed RePL4UP would also keep frequent patterns on the compressed frequent database sequences as PLWAP tree and the compressed position codes of the small items in sequences as small code profile. The PL4UP algorithm also buffers near frequent patterns through a compressed PLWAP tree.

1.3.2.6 FS-miner algorithm The FS-Miner (El-Sayed et al. 2004) proposes a method for mining web sequential patterns using a data structure called FS-tree that is similar to the WAP and PLWAP tree, which were proposed before this FS-tree algorithm. The differences between the FS-tree and WAP, PLWAP trees are that the FS-tree, unlike both WAP and PLWAP trees, stores both the frequent and potentially frequent sequences (those with support count less than minimum support count but greater than or equal to a lower tolerance minimum support). While the WAP and the PLWAP algorithms maintain a more concise frequent header linkage tables for only frequent 1-items, the FS-tree maintains a header table (HT) for all links, their counts and link head pointers. This places some demand on storage as this metadata may need more storage than the original database when the size of database increases with long sequences.

FS-Miner (El-Sayed et al. 2004) provides the option of incremental mining of patterns in the same sequential mining algorithm by including two more data structures (Non-frequent links table NFLT, and further adding to each node of the FS-tree a sequence end structure, seqEnd consisting of sequence id and count) in its FS-tree. The algorithm scans the incremental database db to obtain the count of links in db so that it can update counts in HT and NFLT header tables. It then updates the minimum support count and minimum link support thresholds so that it can move link entries between NFLT and HT tables. Then, it inserts all links that moved from the NFLT to HT tables into the FS-tree. This process may entail deleting some re-occurring sequences that had been at the top of the tree as this algorithm's insertions are done from current node. After updating the FS-tree as described above, it mines only updated links by either dropping patterns that no longer meet minimum support or calling the FS-Mine algorithm for the updated links. The maintenance techniques of this algorithm are similar to our approach in some respects, which are that it stores the database concisely in a tree and does not engage in level-wise candidate generation and avoids scanning the entire old database to handle database updates. The differences between our techniques, based on the PLWAP algorithm, and the FS-Miner algorithm are that while FS-Miner stores the entire database as links with a lot of supporting data structures for monitoring counts, it still performs recursive intermediate generation of conditional sequence bases, CFST (conditional FS-tree) trees during mining as done by the WAP algorithm. The proposed incremental RePL4UP, like the PLWAP, mines only updated branches of the revised tree using position codes without the need to construct and store intermediate PLWAP trees.

An initial draft outline of the two techniques for incremental mining using the RePL4UP (Ezeife and Chen 2004a) and PL4UP (Ezeife and Chen 2004b) were presented in conference papers. These do not discuss many of the recent literature discussed in this paper as the systems were not developed then, and these earlier drafts do not include full system implementations, proofs of correctness, algorithmic details and experimentations with other existing systems.

1.4 Contributions

This paper proposes two algorithms namely, the novel RePL4UP (Revised PLWAP FOR UPdated sequential mining) and the more traditional PL4UP (PLWAP FOR UPdated sequential mining). These algorithms apply the PLWAP-tree (Ezeife and Lu 2005; Ezeife et al. 2005; Lu and Ezeife 2003) to the incremental web sequential mining problem. The algorithms eliminate the need to re-scan the old database when new changes arrive, in order to update old patterns. The RePL4UP initially builds a PLWAP tree that is based on regular minimum support percentage, $s\%$, which stores compact position codes of all small items in the database. When new database records are inserted or deleted, only these new incremental records are scanned to revise the original PLWAP tree and to mine new patterns. With the PL4UP algorithm, the approach is to initially build a PLWAP tree that is based on a lower tolerance minimum support percentage, $t\%$, which is lower than application's regular minimum support percentage, $s\%$. The tolerance support, $t\%$, is used to predict and accommodate items likely to become frequent after database update in the tree. These incremental techniques yield better performance and allow for application scalability.

1.5 Outline of the paper

Section 2 presents the proposed RePL4UP algorithm with example mining using this algorithm detailing how to build and mine this version of PLWAP tree. Section 3 presents PL4UP algorithm with examples. Section 4 discusses experimental performance, time complexities and algorithm overhead analysis, while Sect. 5 presents conclusions and future work.

2 The first proposed incremental RePL4UP algorithm

The first algorithm, RePL4UP being proposed for mining frequent sequential patterns incrementally is based on the PLWAP tree structure, which carries with it, some metadata description of virtual positions of all small items in the database sequences as would have been recorded in that version of the PLWAP tree called RePL4UP tree.

2.1 Definitions

This section presents four formal definitions for small code profile, RePL4UP and PL4UP trees.

Definition 2.1 *Small Code Profile* $Scode^{DB}$: is the set of "would have been" position codes of all small 1-items in the database tree. The small code profile for a small item, l written as $Scode^l$ is the position code that this small item would have been assigned if it were large enough to be inserted in the database tree like PLWAP or RePL4UP.

Definition 2.2 *Small Code for a Small Item l,* $Scode^l$: is the position code that corresponds to the first vacant virtual child position code for a suffix item (event or node), which follows an immediate parent prefix event p on the same sequence, already inserted on the RePL4UP tree or has a small code. Thus, the $Scode^l$ of a small item l in a sequence is formed as the code of l's parent node p with binary '1' appended to it, if l is the leftmost child of p, otherwise, $Scode^l$ is formed as the code of l's nearest virtual left sibling (either physical on the tree or virtual on small code profile) with binary '0' appended to it.

For example, given that the transaction has a sequence like $\ldots pl \ldots$, and p is already inserted into the tree with a position code of binary 10, then the small item l should have a small code $Scode^l$ equivalent to the code of an item in sequence following the event p, which is 101 if p does not have any children, but l has small code of 1010 for the second child of p if p has a leftmost child and so on. If parent node p has one physical child on the tree with code 101, and a virtual small child l with code 1010, and another small child $l1$ arrives, $l1$ is assigned a code of 10100. Note that the RePL4UP, like the PLWAP, considers only physical frequent nodes on the tree when assigning position codes to frequent nodes physically on the tree, but considers both physical and virtual small item children codes already assigned, when assigning new small item codes.

Definition 2.3 *A RePL4UP tree*: is a PLWAP tree based on minimum support s, but which carries with it the small code profile of all small 1-items in the database.

An example RePL4UP tree for the sample database of Table 1 mined at a minimum support threshold of 50% or 3 transactions, consists of the PLWAP tree of Fig. 1 and the following small code profile $Scode^{DB}$ representing position codes of all small 1-items in the database sequences. For example, in TID 100 <abdac>, the small 1-item d has a code of 111 because node b in its prefix sequence ab has a code of 11. The first small item e from TID 200 ⟨aebcace⟩ has a code of 110 because its prefix sequence a has already one physical child node b from TID 100 with position code 11, thus, item e should assume the next vacant child position code of 110. Then, following virtual branch 110, the second small item e has the code 11011111. For the five example sequences, the rest of the small code profiles are defined in a similar fashion.

$Scode^d$ = {111} or {7}
$Scode^e$ = {110, 11011111, 11100} or {6, 223, 28}
$Scode^f$ = {1100, 11001111, 1110011} or {12, 207, 115}
$Scode^g$ = {111001} or {57}
$Scode^h$ = {11100111} or {231}

Definition 2.4 *A PL4UP tree*: is a PLWAP tree based on lower tolerance minimum support $t\%$, which is lower than the regular minimum support $s\%$.

2.2 Mining incremental patterns with RePL4UP

Every event e_i in DB (old database) is either a frequent event (belonging to the set F of events) or a small event (belonging to the set S of events). Similarly, every event, e'_i in the updated database (DB + db) belongs to either the set of now frequent events F' or now small events S'. This allows the classification of every item or event e'_i in updated database, U, into one of the six categories of events (frequent-frequent, frequent-small, small-frequent, small-small, new-frequent and new-small items):

1. Frequent-frequent items ($F \rightarrow F'$) are frequent in old database, DB and still frequent in updated database (old + new data), U.
2. Frequent-small items ($F \rightarrow S'$) are frequent in old DB but small in updated database.
3. Small-frequent items ($S \rightarrow F'$) are small in old DB but frequent in updated database.
4. Small-small items ($S \rightarrow S'$) are small in old DB and still small in updated database.
5. New-frequent items ($\emptyset \rightarrow F'$) were not in old DB but are frequent in updated database.
6. New-small items ($\emptyset \rightarrow S'$) were not in old DB and are small in updated database.

The RePL4UP algorithm being proposed, aims at computing new frequent patterns for updated database when new records are inserted or deleted without having to scan the entire updated database. The RePL4UP algorithm assumes that the original frequent patterns of the database, DB have been generated using an original RePL4UP tree with its small code profile $Scode^{DB}$ metadata collected. The RePL4UP algorithm then scans only the incremental database (db). The incremental database (db) is taken as the union of the inserted (db+) and the deleted (db−) records, where counts of deleted records are negative and those of inserted records are positive. The most important update made to the old RePL4UP tree are for two classes of items namely: frequent-small items in category 2, which are, $F \rightarrow S'$, that now need to be deleted from the old tree to get current tree, and the small-frequent items in category 3, which are $S \rightarrow F'$ that need to be inserted into the current tree. The $S \rightarrow F'$ items pose the greatest difficulty in incremental mining because the old PLWAP tree does not carry any information about transactions in the old database that have any of these previous small items, which now need to be included in the current tree. Existing algorithms normally will need to scan the entire old database to get information about these small items. The RePL4UP approach is to take advantage of the position code property of the RePL4UP tree and during initial construction of the RePL4UP tree, store the list of position codes of all small items.

2.2.1 *The RePL4UP algorithm*

The RePL4UP algorithm (presented formally as Algorithm 2.1) RePL4UP for incrementally mining web log sequential patterns, accepts seven input datasets, 1. cardinality of original database $|DB|$, 2. the incremental database db, 3. minimum support percentage (λ), 4. original database tree ($RePL4UP^{DB}$), 5. small code profile of DB

($Scode^{DB}$), 6. old DB mined frequent patterns (FP^{DB}), and 7. old DB candidate lists (candidate 1-items (C_1), frequent 1-items (F_1), small 1-items (S_1)). The RePL4UP algorithm then goes through seven steps to generate updated versions of five of the above input datasets as its output datasets. The five output datasets of the RePL4UP algorithm consists of the updated (1) cardinality of the updated database $|U|$, (2) frequent patterns (FP'), (3) RePL4UP tree for the updated database U ($RePL4UP^U$), (4) updated candidate lists (C'_1, F'_1, S'_1) and (5) updated small code profile ($Scode^U$).

The summary of the seven steps of processing undergone by the RePL4UP algorithm is:

Step 1. The RePL4UP updates all candidate lists (C_1, F_1, S_1) in the most efficient way.

Step 2. From the updated candidate lists, (C'_1, F'_1, S'_1), it classifies items into one of the earlier six item change categories.

Step 3. It then uses the classification of items to revise the old $RePL4UP^{DB}$ so as to delete new small items and insert new large items using the small code profile $Scode^{DB}$.

Step 4. It mines modified branches of the $RePL4UP^{DB}$ tree to get $ReFP^{DB}$ (revised FP).

Step 5. It builds and mines a $RePL4UP^{db}$ for only the incremental db to obtain FP^{db}.

Step 6. It combines all three types of frequent patterns consisting of (i) old FP^{DB} not including the revised FPs from old DB $ReFP^{DB}$, (ii) now the revised FPs from old DB $ReFP^{DB}$, and (iii) FP from incremental database $RePL4UP^{db}$.

Step 7. Using the new F_1^{db} from the incremental db, and the frequent sequences from db transactions, update the $RePL4UP^U$ by inserting each frequent sequence, updating links and $Scode^{DB}$.

The formal algorithm for mining sequential patterns incrementally with RePL4UP is given as Algorithm 2.1 RePL4UP.

Algorithm 2.1 (RePL4UP-Mines Web Log Sequences Incrementally)

Algorithm RePL4UP()

Input: original database, DB or its cardinality, Incremental database, db, minimum support percentage λ ($0 < \lambda \leq 1$), original DB tree $RePL4UP^{DB}$, old frequent pattern, FP^{DB}, old candidate lists (C_1, F_1, S_1), small code profile $Scode^{DB}$.

Output: updated frequent patterns for updated database, U (FP'), updated $RePL4UP^U$ tree. updated candidate lists (C'_1, F'_1, S'_1). updated small code profile $Scode^U$.

Intermediate data: support counts in databases (s, s^{db}, s′), incremental db candidate lists (C_1^{db}, F_1^{db}, S_1^{db}))

begin

(1) Update all candidate lists as follows:

$C'_1 = C_1 \cup C_1^{db}$; s' = λ of ($|DB| + |db|$).

F'_1 = elements in C'_1 with support count $\geq$ s′

S'_1 = elements in C'_1 with support < s′.

$F_1^{db} = C_1^{db} \cap F'_1$; $S_1^{db} = C_1^{db} \cap S'_1$.

(2) Classify items in the updated data, U into one of 6 classes as:
$F_1 \cap F'_1$ are in class $F \to F'$; $F_1 \cap S'_1$ are in class $F \to S'$.
$S_1 \cap F'_1$ are in class $S \to F'$; $S_1 \cap S'_1$ are in class $S \to S'$.
$F'_1 - F_1$ are in class $\emptyset \to F'$; $S'_1 - S_1$ are in class $\emptyset \to S'$.
(3) Modify the old $RePL4UP^{DB}$ tree such that all $F \to S'$ items are deleted (see Algorithm Smallitem_Delete 2.2) from the tree, and all $S \to F'$ are inserted (see Algorithm Largeitem_Insert 2.3) into tree using the $Scode^{DB}$.
(4) Mine only modified branches of $RePL4UP^{DB}$ to obtain frequent patterns $ReFP^{DB}$.
(5) Construct and mine small $RePL4UP^{db}$ to obtain frequent patterns FP^{db}
(6) Combine the three frequent patterns to obtain FP' as:
$FP' = (FP^{DB} - ReFP^{DB}) \cup ReFP^{DB} \cup FP^{db}$
(7) Insert the frequent sequence transactions with ($\emptyset \to F'$ items) from the incremental database into the original $RePL4UP^{DB}$ tree to get $RePL4UP^{U}$; update the links and small code profiles and to include items in class $\emptyset \to F'$.

end // of RePL4UP //

Algorithm 2.2 (Smallitem_Delete - Deleting a Small Item l from Tree)

Algorithm Smallitem_Delete()
Input: RePL4UP tree, smallitem l
Output: RePL4UP tree
begin
(1) Follow l linkage from tree to last l node on tree, record l's position code as l's small code profile, delete last l node, back up and keep deleting l nodes and recording as small code profile until the l header linkage is deleted.
(2) Adjust tree nodes, counts and position such that if a parent node of a node n is deleted, the closest grand parent of n becomes the new parent of n and if two siblings of the new parent have the same node labels, they are merged using the Definition 2.11

end // of Smallitem_Delete //

Algorithm 2.3 (Largeitem_Insert - Inserting a Frequent Item f into the Tree)

Algorithm Large item Insert()
Input: RePL4UP tree, frequent item f,
small item code profile $Scode^{f}$
Output: RePL4UP tree
begin
(1) Following the PLWAP tree position labeling scheme, find the binary position of the small code profile of

frequent item f $Scode^f$ if it exists and insert f.
If position $Scode^f$ does not exist, insert f in
the maximum prefix code of $Scode^f$ found in the tree
as presented in Definition 2.5 for small code restoration.
(2) Adjust tree nodes, counts and position codes if needed.
end // of Large item Insert//

More details of the sequence of seven steps in the RePL4UP algorithm as summarized in the formal Algorithm 2.1 and earlier, are discussed next.

Step 1: In the first step, the RePL4UP algorithm updates all candidate lists (C_1, F_1, S_1) from the previous mining state of original database, DB, to obtain their updated versions (C_1', F_1', S_1') for the updated database U. This algorithm scans only the small incremental database, db, at this stage, to get the incremental db candidate lists (C_1^{db}, F_1^{db}, S_1^{db}), which are used to quickly update the candidate lists of the updated database as: $C_1' = C_1 \cup C_1^{db}$. Now, the updated minimum support (s') is computed as the minimum support percentage of the new cardinality of updated U, which is the sum of the cardinalities of the old DB and the incremental db. This updated s' is now used on updated C_1' to compute the updated frequent 1-item list F_1' as all elements in C_1' with support less than s' form the updated small 1-item list S_1'. The algorithm computes the frequent and small 1-items in the incremental database using the set intersection operation of the candidate 1-item list of the incremental db with the already found frequent 1-item list for the updated database, F_1'. Thus, $F_1^{db} = C_1^{db} \cap F_1'$ and $S_1^{db} = C_1^{db} \cap S_1'$. This approach has the advantage of retaining only those F_1^{db} items as frequent that are also frequent in the updated database, U. Note that the meanings of the set union and intersect operations adopted in these algorithms for updating candidate lists are taken as follows. The union operator as in $C_1' = C_1 \cup C_1^{db}$ operation, returns as its result, all members in either one or both of its input set (e.g., C_1, C_1^{db}) and the count of each event (item) in the result set, is the same as the sum of the counts of this event in the two input sets. For the intersect operation as in $C_1^{db} \cap F_1'$, it would return as its result, all members in both input sets and the count of each event in the result set, is the same as the count of the input event with lower count.

Step 2: Using the updated frequent and small 1-items (F_1',S_1') from the first step above with their original versions from input data (F_1,S_1), the algorithm now classifies all items in updated C_1' into one of six item categories (frequent-frequent, frequent-small, small-frequent, small-small, new-frequent, new-small). This enables it know which items to delete from old tree to get current tree and which items to insert into the current tree. The classifications are handled as follows. Items in the first class $F \rightarrow F'$ (that is frequent in both old DB and updated database U, called frequent-frequent items) are found with the operation $F_1 \cap F_1'$. Items in the second class $F \rightarrow S'$ (frequent-small items) are found with the operation $F_1 \cap S_1'$. Items in the third class, $S \rightarrow F'$ (small-frequent items) are found with $S_1 \cap F_1'$. The fourth class of items (small-small

items) $S \rightarrow S'$ are found with $S_1 \cap S'_1$. The fifth class of items (new-frequent) $\emptyset \rightarrow F'$ are found with operation $F'_1 - F_1$, while the sixth class of items (new-small) $\emptyset \rightarrow S'$ are found with the operation $S'_1 - S_1$.

Step 3: This step updates the $RePL4UP^{DB}$ tree. The important class of items are the frequent-small ($F \rightarrow S'$) items, which are first deleted from the $RePL4UP^{DB}$ tree, and the small-frequent ($S \rightarrow F'$) items, which are next inserted into the $RePL4UP^{DB}$ tree being modified. To delete a frequent-small ($F \rightarrow S'$) item, as summarized in formal Algorithm 2.2 Smallitem-Delete, the item link is followed from the item's frequent header linkage entry in the *RePL4UP*^{DB} tree to each of its node label on the tree until its final leaf node. Then, each of this item's nodes is deleted while backtracking till its frequent header linkage entry is deleted. Every deleted node has its position code saved in the updated small code profile for this item. After deleting each small element, the remaining tree nodes, their counts and position codes are adjusted such that if a parent node of a node *n* is deleted, the closest grand parent of *n* becomes the new parent of *n* and if two siblings of the new parent have the same node labels, they are merged. For example, consider the sequence aebcace, where the parent of node "b" is "e", if item "e" was frequent and on the tree but became small after incremental update, then, the item "e" will be deleted from the $RePL4UP^{DB}$ tree causing the new parent of node "b" to be its closest grand parent node, which is "a". This operation will result in a new sequence "abcac" from the previous sequence "aebcace" as theoretically expected. When two sibling nodes (with the same label) merge, the resulting node is assigned the same label as the merged nodes, a count that is the sum of the counts of the merged nodes, and a position code that is consistent with the position code assignment scheme of the PLWAP tree. After deleting all $F \rightarrow S'$ items from the tree, next, all small-frequent ($S \rightarrow F'$) items are inserted into the $RePL4UP^{DB}$ tree using the small code profile ($Scode^{DB}$). As presented in the Algorithm 2.3 Largeitem-Insert, to insert a new frequent item, *f*, the RePL4UP algorithm finds the binary position in the $RePL4UP^{DB}$ tree, that corresponds to each small code in the small code profile set for this frequent item $f(Scode^f)$ and inserts a node for item *f*. If $Scode^f$ does not exist in the tree, *f* is inserted at the position in the tree that corresponds to the maximum prefix code of $Scode^f$. Thus, for every small item *l*, with an $Scode^l$ (a small item code for *l*), there is always a safe small item restoration spot found in the existing $RePL4UP^{DB}$ tree, where this small item *l* should be inserted. This safe restoration spot has the same binary position code as $Scode^l$ or the codes' maximum prefix position found in the tree. For example, for an $Scode^l = 223$ or 11011111 in binary, if the maximum prefix position found in the tree is 1101, this position is where this item *l* is inserted.

Step 4: The algorithm at this stage, after updating the *RePL4UP*^{DB} tree as in third step above, mines only modified branches from the Root of the updated *RePL4UP*^{DB} tree, in order to obtain the revised frequent pattern, *ReFP*^{DB}. In mining the modified branches, any frequent subsequence from this branch already in old frequent pattern list is ignored since including it will inflate its

support count. The ignoring of patterns found in modified branches, which already exist in the database frequent pattern list, can be done during final integration of the three types of patterns by using only those patterns in old FP^{DB} branches that had not been modified to compute remaining unchanged FP^{DB} patterns as $FP^{DB} - ReFP^{DB}$.

Step 5: At this stage, the $RePL4UP^{db}$ tree for only the incremental database, db, is constructed, to obtain the new frequent patterns present in the incremental db, FP^{db}. This $RePL4UP^{db}$ tree is built using the $F_1^{db} = C_1^{db} \cap F_1'$ from step 1. Thus, F_1^{db} is not directly computed from incremental db.

Step 6: This step obtains the updated frequent pattern FP' from the union of the three frequent patterns (i) those from previous mining of original database, not including patterns that have changed after tree revisions ($FP^{DB} - ReFP^{DB}$), (ii) patterns from mining only modified branches of the revised $RePL4UP^{DB}$ tree ($ReFP^{DB}$), and (iii) patterns from mining only the incremental database (FP^{db}). The updated frequent patterns FP' are patterns in $(FP^{DB} - ReFP^{DB}) \cup ReFP^{DB} \cup FP^{db}$ with support count greater than or equal to s'.

Step 7: This step involves ensuring that the updated database tree, $RePL4UP^{DB}$ tree with the small code profiles are updated with the incremental database records for the next round incremental mining and updating. Thus, the algorithm inserts the frequent sequence transactions (with $\emptyset \rightarrow F'$ items) from the incremental database into the updated $RePL4UP^{DB}$ tree, while small code profile of the new small items ($\emptyset \rightarrow S'$) in the incremental db records as well as the small code profiles of all deleted nodes ($F \rightarrow S'$) from the tree are recorded in the small code profiles.

2.2.2 *Discussions on RePL4UP algorithm features*

The correct incremental computation of frequent patterns with the RePL4UP algorithm is based on the two properties, that had been tested experimentally, and logical correctness shown through heuristics.

Property 1 Given a set of database sequences, the RePL4UP algorithm can be used to construct the entire PLWAP tree incrementally starting with the first sequence of the database, and building small code profile as needed.

Property 2 Given a RePL4UP tree with small code profile, when some small items become frequent, the small code profiles can be used to restore the small events in equivalent correct positions in the tree to mine the frequent patterns correctly.

These two facts were analyzed using different datasets on both the RePL4UP and PLWAP algorithms and the results generated the same frequent patterns for various test case scenarios.

Heuristic argument for correctness of properties 1 and 2: Given a database consisting of a set of sequences $S_1, S_2, \ldots, S_n$ with minimum support percent of $s\%$. Each S_i is a sequence of events $e_{i1}e_{i2} \ldots e_{im}$. The RePL4UP algorithm inserts the first event e_{i1} belonging to S_i into the RePL4UP tree if e_{i1} is a frequent event starting from Root. The event e_{i1} is assigned a label e_{i1}, a count 1 if it is the first node with this label,

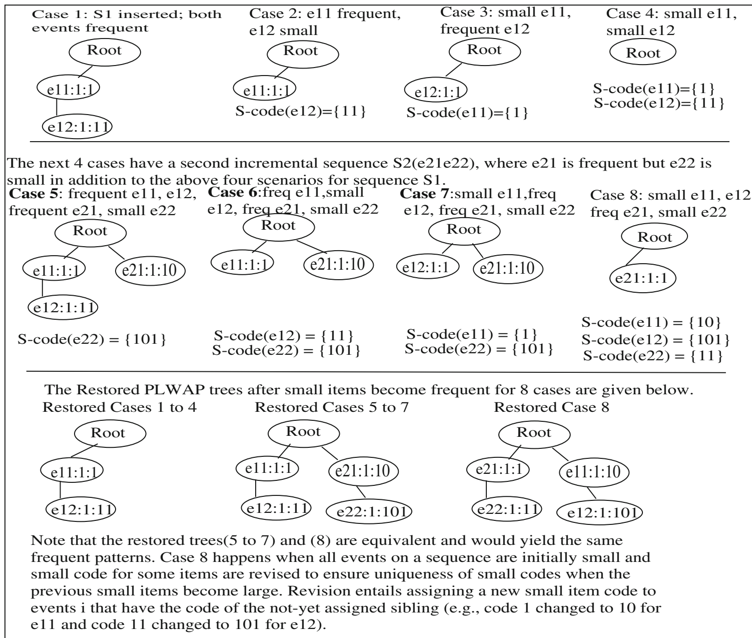

Fig. 3 RePL4UP incremental building of tree and restoration

otherwise the count of this node label is increased by 1. The position code of e_{i1} is binary 1 if it is the leftmost child of Root and binary 10 if it is the second child from left and so on. If e_{i1} is small, it is not inserted into the RePL4UP tree but its "would have been" position code is recorded as its small code profile $Scode^{e_{i1}}$, which is $\{1\}$ for this case. Assume, we continue to work with the frequent e_{i1} track, the next event e_{i2} can be either frequent and inserted with position code 11 or small and not inserted but with small code $Scode^{e_{i2}} = \{11\}$. Figure 3 shows that using the RePL4UP algorithm to insert m (with $m =$ two for simplicity here) sequences covering a number of scenarios incrementally, yields a PLWAP tree. This demonstrates correctness of Property 1.

Property 2 is discussed by showing that for each constructed RePL4UP tree of Fig. 3, the small code profiles are used to restore the small items assuming the small items become large.

Property 3 *Small item l code $Scode^l$ uniqueness*: Given an $Scode^l$ for a small item l, this code is unique and different from any other small item code for any small item except if this small item code l at this specific node position has occurred more than once (that is, has a count $n > 1$), in which case the code is listed n times. For example, an $Scode^l = 223$ or 11011111 is listed only once in the small code profile if this l node has occurred once but is listed n times if it has occurred n times. When a sequence (e.g., S_1) has all of its items small, then, this sequence is not at all represented on the initial RePL4UP tree indicating that a later DB sequence (e.g., S_2) may have occupied a branch (e.g., branch 1 from Root) of the tree that would have been assigned to events of the small sequence (S_1) if they had been frequent. Thus, frequent sequence S_2 event

now becomes the leftmost child of Root with position code 1 because sequence S_1 events were small with small codes as 1, 11, etc. Property 4 shows how uniqueness is preserved for this scenario.

Property 4 Small item l code $Scode^l$ Collision Resolution: To avoid possible conflict or collision between events belonging to different sequences, if all events in a sequence are small, the small codes of the events are revised to be the code of rightmost vacant virtual sibling spot available on the tree, when a new sequence is inserted, in order to ensure uniqueness and avoid two sequences like S_1 (now small) and S_2 (now frequent) both occupying branch 1 of Root. Since S_1 arrived first, it had been assigned small code 1 corresponding to the first branch, but none of its events are attached to the Root yet, because they are not frequent. When S_2 arrives and is attached to Root at branch 1, we revise the previously assigned small codes of S_1 to branch 10 so that when they become frequent, there are no conflicts of events of both sequences S_1 and S_2 both sharing the same branch of 1. This special case is presented in case 8 of Fig. 3 for showing correctness of technique. The only other case that may appear like collision is when a virtual small item code assigned earlier, is similar to a physical frequent event code on the tree. However, as Property 6 discusses when those small items join the physical tree if they become frequent, they are placed in their correct positions on the tree and the position codes of events on the tree are adjusted to be unique and consistent with the PLWAP code assignment scheme.

Property 5 Small item l code $Scode^l$ computation from code of first small item i in a transaction sequence: Given the $Scode^i$ for the first small item i in a transaction sequence, and the offset (number of character positions from item i to item l) the $Scode^l$ is computed quickly from the $Scode^i$ by appending "1" to the binary value of $Scode^i$ number of times corresponding to the "offset". This is equivalent to computing the $Scode^l$ as $Scode^i * 2^{offset} + (2^{offset} - 1)$.

For example, given the transaction sequence "abegfh" where e, f, g, h are small items and the code profile for the first small item in this sequence "e" is 11100 = 28. To compute the small code profile of "g" with offset "1" from "e", we append "1" only once to the small code profile of "e" since the offset of "g" from "e" is 1 to get 111001 or 57. This is also computable from $Scode^i * 2^{offset} + (2^{offset} - 1)$ as $(28 * 2^1 + 2^1 - 1 = 57)$. Similarly, the small code profile for "f" with offset "2" is computed as 1110011 = 115 or $(28 * 2^2 + 2^2 - 1 = 115)$. The small code profile of "h" with offset "3" is computed as 11100111 or 231.

Property 6 Small item l code $Scode^l$ similarity with frequent item code: Given an $Scode^l$ for a small item l, this code is similar to a frequent item f code if (1) small item l is on the same sequence and is a direct parent of the frequent item f in the original database sequence, and (2) if the small item l is a sibling node of the frequent item f. In both cases, the small item is assigned its virtual code first because it arrives first and if it becomes frequent, this code places it in its correct physical position and all codes are revised to be frequent unique.

For example, given a database sequence "abdac" where "d" is the only small item, the frequent item "a" (second one) would have the same physical position code of "111" as the small item "d" which is not in the tree but appears before the second "a" on the database sequence.

2.2.3 *Example mining of the original RePL4UP^{DB} tree*

Example 2.1 Suppose we have a database DB as given in the first two columns of Table 1 with the set of items, I = { a, b, c, d, e, f, g, h} and minimum support count = 50% of DB transactions, build and mine a RePL4UP tree of this database for initial frequent sequential patterns.

Solution 2-1 Mining original database sequences with RePL4UP tree entails first building a PLWAP tree that remembers small code item profiles ($Scode^{DB}$) and then mining this tree to obtain original frequent patterns as shown in the four steps below.

1. Compute all Candidate Lists: frequent 1-items, F_1, small 1-items, S_1 and candidate 1-items, C_1. For Example 2.1 above, these are: C_1 = {a:5, b:5, c:3, d:1, e:2, f:2:, g:1, h:1}.
   With support s = 3, $F_1 = \{a, b, c\}$, $S_1 = \{e, f, d, g, h\}$.
2. Generate frequent sequences for all database records (seq_s as shown in column 3 of Table 1) by deleting all small events.
3. Construct the $RePL4UP^{DB}$ tree with seq_s. The $RePL4UP^{DB}$ tree is built using the sequences in seq_s the same way that the PLWAP tree is built, but also collecting the small code profile $Scode^{DB}$. A RePL4UP tree using the seq_s of the database is shown as (Fig. 1) and its $Scode^{DB}$ given with Definition 2.3. The frequent sequences starting with sequence "abac" are inserted into the tree from Root with each node annotated as node label: node count: node position code. While inserting the frequent items in the sequence, the algorithm checks the original transaction to mark location of small items in the transaction. For example, small item *d* in the original first transaction *abdac* would have had the position code (d:1:111) in the created branch if this *d* were frequent. It will write this position code in the small item code profile for item *d* as $Scode^d$ = {111}. The complete small item code profiles from the second sequence aebcace is for small item *e* as: $Scode^e$ = {110, 11011111}. The third (baba), fourth (afbacfc) and fifth (abegfh) sequences are all inserted into the tree in a similar fashion to create the $RePL4UP^{DB}$ tree shown as Fig. 1.
4. The initial $RePL4UP^{DB}$ is then, mined to generate the initial frequent pattern, FP^{DB} the same way the PLWAP tree is mined. The generated list of FP^{DB} mined based on the support of 3 is: FP^{DB} = {a:5, aa:4, aac: 3, ab:5, ac:3, aba:4, abac:3, abc:3, b:5. ba:4, bac:3, bc:3, c:3}.

2.2.4 *Example incremental mining of frequent patterns with RePL4UP*

Example 2.2 Assume that the original database, DB of Table 1 is updated with the records in the incremental database table shown as Table 2, use the RePL4UP algorithm to obtain the updated frequent patterns, FP', given the old frequent patterns, FP^{DB}, the old $RePL4UP^{DB}$ tree from the previous section, the new incremental database (db), and other 1-itemset information from initial mining (candidate 1-items, frequent 1-items, small 1-items, small item code profiles $Scode^{DB}$), the same minimum support percent, $s\%$, which is 50% of the updated database cardinality.

Table 2 The incremental database transaction Table (db)

TID	Web access seq.	Frequent subseq with s using% $C_1^{db} \cap F_1'$	Frequent subseq with t using $C_1^{db} \cap F_t'$
600	bahefg	baef	bahefg
700	aegfh	aef	aegfh

Solution 2-2 The RePL4UP steps for incrementally mining the database using mostly incremental db and old available patterns are:

1. Update all candidate 1-itemset lists and patterns by scanning only the incremental database (db) to compute C_1', F_1', S_1' as: $C_1' = C_1 \cup C_1^{db}$. $F_1^{db} = C_1^{db} \cap F_1'$. $S_1^{db} = C_1^{db} \cap S_1'$. For Example 2.2, these are: C_1 = {a:5, b:5, c:3, d:1, e:2, f:2:, g:1, h:1}, $F_1 = \{a, b, c\}$, $S_1 = \{e, f, d, g, h\}$. C_1^{db} = {a:2, b:1, e:2, f:2, g:2, h:2}. $C_1' = \{a : 7, b : 6, c : 3, d : 1, e : 4, f : 4, g : 3, h : 3\}$. The cardinality of updated database, $|U| = 5 + 2 = 7$, making the support cardinality, s', of updated database, U = 50% of 7 or 4 transactions of U. $F_1' = \{a{:}7, b{:}6, e{:}4, f{:}4\}$. $F_1^{db} = C_1^{db} \cap F_1'$ = {a:2, b:1, e:2, f:2}. $S_1' = \{c{:}3, d{:}1, g{:}3, h{:}3\}$. $S_1^{db} = C_1^{db} \cap S_1'$ = {g:2, h:2}.
2. From the updated data patterns in step 1 above, we want to classify all items in C_1' such that each will belong to one of the earlier identified six classes of (i) Frequent-frequent class = $F \rightarrow F' = F_1 \cap F_1'$ = {a, b} for the example. (ii) Frequent-small class = $F \rightarrow S' = F_1 \cap S_1'$ = {c}. (iii) Small-frequent class = $S \rightarrow F' = S_1 \cap F_1'$ = {e, f}. (iv) Small-small class = $S \rightarrow S' = S_1 \cap S_1'$ = {d, g, h}. (v) New-frequent class = $\emptyset \rightarrow F' = F_1' - C_1 = \emptyset$. (vi) New-small class = $\emptyset \rightarrow S' = S_1' - C_1 = \emptyset$.
3. Modify old $RePL4UP^{DB}$ tree to delete from each branch of the tree, all frequent-small or $F \rightarrow S'$ items constituting the set {c}, and to insert into the tree at the position defined by their small code profile $Scode^{DB}$, all $S \rightarrow F'$ or small-frequent items, which are items in the set {e, f} for the example. For example, we insert the small-frequent items {e, f} in the tree at the positions indicated in the small item code profile, $Scode^{DB}$ collected from the database previously as: $Scode^e$ = {6, 223, 28} or {110, 11011111, 11100}; $Scode^f$ = {12, 207, 115} or {1100, 11001111, 1110011}. Also, during the initial tree construction, some frequent nodes share branches and this will cause some of the computed small codes to not be physically present in the old tree being revised. However, every small item code profile has a unique position in the tree, determined by the maximum matching prefix of its binary position code found in the tree by applying code restoration logic. Thus, if the small code profile is 1110011 and the closest prefix node position in the tree being updated we can find is for the prefix 11100, then, we insert the small-to-frequent item there. The new code profile of the item is its physical position code in the tree. The small code profile list is updated for small-frequent items {e, f} after tree modification as: $Scode^{e'}$ = {110, 1101, 1110} and $Scode^{f'}$ = {1110, 11001, 11101}.

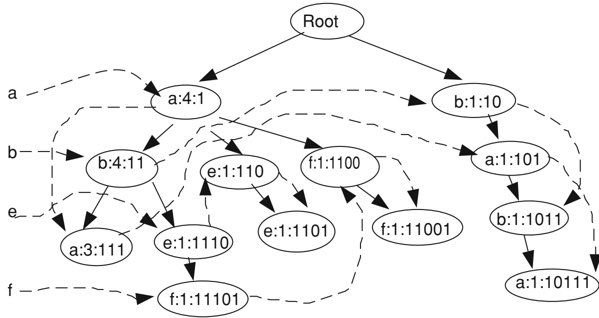

Fig. 4 The RePL4UP after deletions and insertions

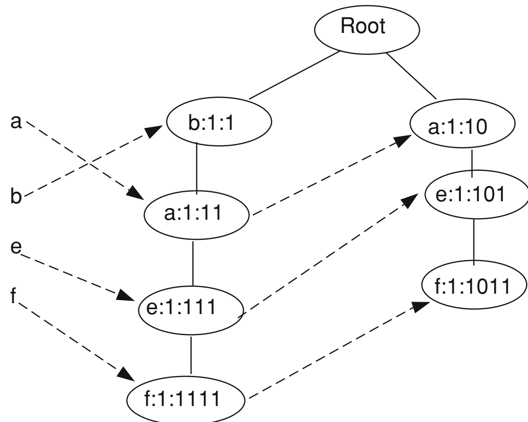

Fig. 5 The RePL4UP tree for incremental database db

4. Mine only the modified branches of *RePL4UP*^{DB} to obtain revised frequent patterns, *ReFP*^{DB}. All branches of the tree that were modified are mined from Root using the PLWAP mining technique with a minimum support count of 1. The modified branch of the *RePL4UP*^{DB} is the left branch of Root in Fig. 4 and mining it gives the frequent sequences called *ReFP*^{DB} as {aee:1, abef:1, aff:1, ae:2, af:2, be:1, bf:1, bef:1, ef:1, ee:1, ff:1, e:2, f:2}.
5. The *RePL4UP*^{db} tree is computed with frequent sequences of small db based on $F_1^{db} = C_1^{db} \cap F_1'$={a, b, e, f} and is shown as Fig. 5.
   Following this process, the useful mined FP^{db} = {b:1, a:2, e:2, f:2, ba:1, be:1, bf:1, bae:1, baf:1, baef:1, ae:2, af:2, ef:2, aef:2, baef:1}.
6. FP' are patterns in $(FP^{DB} - ReFP^{DB}) \cup ReFP^{DB} \cup FP^{db}$ with support count greater than or equal to s' or 4, and FP' = {a:7, aa:4, ab:5, aba:4, b:6, ba:5, ae:4, af:4, e:4, f:4}.
7. Insert the frequent sequence transactions from the incremental database into the original $RePL4UP^{DB}$ tree to keep it updated for next round mining, and updating. For the example, (baef, aef) are inserted into the main revised $RePL4UP^{DB}$ to keep it up-to-date (as shown in Fig. 6), while the small code profile of the new

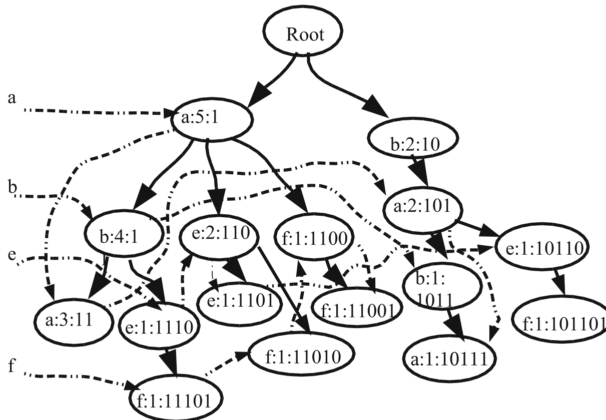

Fig. 6 The final revised RePL4UP tree after all modifications

small items in the new changes as well as the small code profiles of all deleted nodes from the tree are recorded in the small code profiles.

3 The second proposed incremental PL4UP algorithm

This section presents the second algorithm, PL4UP being proposed for mining frequent sequential patterns incrementally based on a PLWAP tree structure. The main idea of PL4UP is to avoid scanning the whole updated database, U (DB+db) when the database changes, by building an initial bigger PLWAP tree that is based on a lower tolerance minimum support percentage $t\%$, which is lower than the regular given minimum support percentage ($s\%$). This database tree that is based on support percentage $t\%$ is called $PL4UP^{DB}$. With the $PL4UP^{DB}$, performing incremental mining entails building and mining a small $PL4UP^{db}$ tree for only these new incremental database (db), which are used for updating existing old patterns. Since regular PLWAP tree for original database, DB based on regular minimum support percentage of $s\%$ stores no information about small items that become frequent when new incremental tuples arrive (that is, items in class $S \rightarrow F'$), PL4UP method is to predict small items likely to become frequent when database is updated so that a lower tolerance support percentage is used to include them in the initial tree. These current small but later "may be" frequent items are called potentially frequent items.

3.1 Use of lower tolerance support $t\%$ for building PL4UP tree

The approach adopted by the proposed PL4UP algorithm is between the two extremes of building a huge tree with all items initially and building a small tree that is based on only frequent items as done by regular sequential mining algorithms. Thus, the success of the PL4UP algorithm depends on its ability to correctly predict potentially frequent (PF) and potentially small items (PS). The PL4UP algorithm uses a formula

to predict the best minimum tolerance support percentage $t\%$, which is smaller than the regular minimum support percentage $s\%$ such that t = factor * s.

3.1.1 How to choose the tolerance minimum support $t\%$ value

In choosing the $t\%$, the objective is to select a $t\%$ that includes the previous small items in the old DB which may become large in the updated database, U. Thus, the value of best recommended $t\%$ is derived as $t\% \geq (1 + CR) * s\% - CR$, where $0 \leq CR \leq 1$, and CR is the rate of change in the database or the ratio of the size of the new changed database ($|db|$) to the size of the original old database ($|DB|$). Thus, change rate (CR) $= \frac{|db|}{|DB|}$. Since t% has to be positive and must be lower than regular minimum support, then (1 + CR) * $s\%$ − CR ≥ 0 and this means that $s\% \geq$ CR/(1 + CR). Evaluating the formula, we arrive at the simple fact that the best recommended $t\%$ value is: $t\% \geq \frac{(|DB|+|db|)*s\%-|db|}{|DB|}$. This is equivalent to: $t\% \geq (\frac{|DB|}{|DB|} + \frac{|db|}{|DB|}) * s\% - \frac{|db|}{|DB|}$ or $t\% \geq (1 + CR) * s\% - CR$, where $0 \leq CR \leq 1$. Also, t% is equal to a factor * s or ($t\% = F * s$), where $0 \leq F \leq 1$. A t% value that exceeds F * s is closer to regular support, $s\%$ and not helpful in achieving the goals of the technique. However, a t% value less than F * s is close to the entire database and is outside the recommended t% limit. The average changes in the database (e.g., what percentage of the database changes every update, hour, day or week) collected over a period of time should be used as the CR in computing the value of t%.

3.2 Incremental mining of FPs with PL4UP using tolerance support

The first process is to build the initial $PL4UP^{DB}$ tree using sequences in a sequential database with support greater than or equal to the tolerance support *t*%. The initial *PL4UP*^{DB} tree is built the same way the PLWAP tree is built (Ezeife and Lu 2005). In both the *PL4UP*^{DB} and *PLWAP*^{DB} trees, there are no records of small items in sequences as done in the *RePL4UP*^{DB} tree using small item code profile. The process of building and mining an original *PL4UP*^{DB} tree and that of incrementally updating and mining it are presented.

3.2.1 The PL4UP algorithm for incremental web frequent patterns

As formally presented in Algorithm 3.1 PL4UP, the eight input datasets to the PL4UP algorithm are: (1) the original database (DB) or its cardinality, (2) the incremental database (db), (3) minimum support percentage percentage (λ), (4) original (*PL4UP*^{DB}) tree, (5) tolerance support percent (θ), (6) old frequent patterns based on tolerance support *t* (*TFP*^{DB}), (7) regular support (*SFP*^{DB}), and (8) five old candidate lists consisting of (i) candidate 1-item list (C_1), (ii)frequent 1-items list based on tolerance support *t* (F_t), (iii) small 1-items (S_1), (iv) potentially frequent 1-items (PF) as items in S_1 with support ≥ *t*, (v) potentially small 1-items (PS) as items in S_1 with support $< t$. The output of the PL4UP algorithm consists of the updated versions of four of the input datasets consisting of updated (1) database U, (2) frequent patterns for the

updated database U (FP'), (3) updated tolerance and regular frequent patterns TFP^U and SFP^U, (4) candidate 1-item lists for the updated database U (C_1', F_t', S_1', PF′, PS′).

The steps in incrementally mining the database with the PL4UP algorithm using mostly changed db and old available patterns are:

1. Update all 1-itemset candidate lists and patterns (F_t', S_1', C_1', PF', PS') using the original DB lists (F_t, S_1, C_1, PF, PS) and the incremental database, db lists. $C_1' = C_1 \cup C_1^{db}$. $F_t^{db} = C_1^{db} \cap F_t'$, The updated regular small 1-items, S_1' are those in C_1' with support less than the updated regular support, $s'\%$, updated potentially frequent (PF') are items in S_1' with support greater than or equal to the updated tolerance support, $t'\%$, and potentially small items (PS') are items in S_1' with support less than the updated tolerance support $t'\%$. Thus, $S_1^{db} = C_1^{db} \cap S_1'$, and $PF^{db} = C_1^{db} \cap PF'$.
2. If no updated potentially small items are now frequent (i.e, $PS' \cap F_1' = \emptyset$), then, continue with tolerance $t\%$ and proceed with steps 3 and 4, otherwise, recompute $t\%$ with the formula on Sect. 3.1 or set to 0% to use the entire database before recomputing input datasets and running from step 1.
3. Construct the small PL4UP tree using only the incremental database, db and with frequent items based on tolerance support, t% computed as $F_t^{db} = C_1^{db} \cap F_t'$. Mine this $PL4UP^{DB}$ to obtain the regular, SFP^{db}, and the tolerance frequent patterns, TFP^{db}.
4. Combine the tolerance frequent patterns from the incremental database, db and the old database, DB, keeping only those patterns that meet the regular support s' in updated SFP' and those that meet tolerance support, $t'\%$ in updated TFP'. Thus, updated SFP' and TFP' are as follows: SFP' are patterns in $TFP \cup TFP^{db}$ with support greater or equal to $s'\%$. Similarly, TFP' are patterns in $TFP \cup TFP^{db}$ with support greater or equal to $t\%$.

Algorithm 3.1 (PL4UP-Mines Web Log Sequences Incrementally with tolerance MinSupport)

Algorithm PL4UP-Tree()
Input: original database, DB or its cardinality, Incremental database, db, minimum support percent λ ($0 < \lambda \leq 1$), original $PL4UP^{DB}$ tree, tolerance support θ ($0 < \theta \leq 1$),old frequent patterns based on t and s, TFP^{DB}, and SFP^{DB} old candidate lists (C_1, F_t, S_1, PF, PS).
Output: updated frequent patterns for updated database, U (TFP' and SFP'), updated candidate lists (C_1', F_t', S_1', PF′, PS′).
Intermediate data: incremental db candidate lists (C_1^{db}, F_t^{db}, S_1^{db}, PF^{db}, PS^{db})), tolerance and regular minimum support counts (t′, s′)
begin

(1) Update all candidate lists as follows:

$C_1' = C_1 \cup C_1^{db}$; s′ = λ of $|DB| + |db|$; t′ = θ of $|DB| + |db|$
F_t' = element in C_1' with support ≥ t′
S_1' = element in C_1' with support < s′;
PF' = elements in S_1' with support ≥ t′
PS' = elements in S_1' with support < t′

F_t^{db}=$C_1^{db} \cap F_t'$
S_1^{db}=$C_1^{db} \cap S_1'$; PF^{db}=$C_1^{db} \cap PF'$.
PS^{db}=$C_1^{db} \cap PS'$
(2) If no updated potentially small item is also now frequent
i.e., if $PS' \cap F_1' = \emptyset$
then
begin /*proceed with current tolerance $t\%$ tree */
(2.11) Construct the small $PL4UP_t^{db}$ on tolerance support, t, using F_t^{db} and mine to obtain the frequent patterns, SFP^{db} and TFP^{db}
(2.12) Combine the two tolerance frequent patterns to obtain TFP' and SFP' as:
$TFP' = TFP_t^{DB} \cup TFP^{db}$
$SFP' = \text{patterns in TFP' with support} \geq \lambda$
end
else /* recompute tolerance $t\%$ or set to 0 */
begin
(2.21) Recompute tolerance $t\%$ as discussed in Sect. 3.1.
or use the entire database by setting $t\%$ to 0.
(2.22) Recompute input dataset with new $t\%$ and go to step 1.
end
end // of PL4UP //

When there are small items with occurrence count lower than the minimum tolerance support percent, t%, that become large in the updated database, it might be better to reconstruct the original PL4UP tree using the entire updated database or newly computed tolerance support $t\%$.

3.2.2 *Example application of the PL4UP algorithm*

Example application of the PL4UP algorithm on datable of Table 1 with regular support $s\% = 50\%$ and tolerance support $t\% = 0.6s\% = 30\%$ would first construct a PL4UP tree (Fig. 2 with seq_t in column 4 of Table 1). Mining the tree PLWAP fashion produces tolerance frequent patterns, TFP as: {a:5, aa:4, aac: 3, ab:5, aba:4, abac:3, abc:3, abcc:2, abe:2, abf:2, ac:3, acc:2, ae:2, af:2, b:5. ba:4, bac:3, bab:1, bc:3, bcc:2, be:2, bf:2, c:3, cc:2, e:2, f:2}. The actual needed frequent pattern, SFP, now are the ones based on the regular support count of 3. These are all patterns in TFP with support 3 or more. The SFP = {a:5, aa:4, aac: 3, ab:5, ac:3, aba:4, abac:3, abc:3, b:5. ba:4, bac:3, bc:3, c:3}. When the incremental database of Table 2 occurs, following the PL4UP algorithm of Algorithm 3.1, generated updated TFP' = {a:7, aa:4, aac:3, ab:4, aba:4, abac:3, abc:3, ac:3, ae:4, af:4, b:6, ba:5, bac:3, bc:3, be:3, bf:3, c:3, e:4, f:4} and SFP' = {a:7, aa:4, ab:5, aba:4, ae:4, af:4, b:6, ba:5, e:4, f:4}.

4 Performance analysis and experimental evaluation

4.1 Experimental analysis

A performance comparison of the two proposed incremental web sequential mining algorithms, the novel RePL4UP, and more traditional PL4UP, with the non-incremental PLWAP algorithm (Ezeife and Lu 2005), and three other incremental sequential mining algorithms GSP+, MFS+ (Kao et al. 2005; Zhang et al. 2002), IncSpan (Cheung et al. 2004) was conducted and the results of the experiments are presented in this section. All these six algorithms were implemented and run on both synthetic and real web log datasets. The synthetic datasets were generated using the source code found under http://www.almaden.ibm.com/software/quest/Resources/index.shtml of the IBM's project QUEST.

The synthetic data is in the format (Tid, Number of elements in sequence, the sequence). Thus, a sample record in the input file for transaction id 128 is: 128 9 2 6 9 13 6 21 22 29 25. The real dataset was generated from the web log data of a Computer Science department server by first applying a web log cleaner (source codes and sample test datasets attached to recent papers are downloadable from our web site http://www.cs.uwindsor.ca/~cezeife/). Our WebLogCleaner program accepts any web log and generates four tables representing the warehouse fact table with schema (timeID, userID, protocolID, pathID, sessionID, nbyte), for the integrated records of the web access log, and its dimension tables PathDimTbl(for pathID), ProtocolDimTbl (for protocolID), UserDimTbl (for userID). This fact table is further processed to produce a transaction table in the format of (Tid, Number of elements in sequence, the sequence), which our sequential mining algorithms mine. The processed real datasets have between 10 thousand and 80 thousand records and show the same result patterns as the synthetic datasets. While there are a few records with very long sequences (e.g., 218 and 165 items in a sequence), a large percentage of the real dataset has only one element that had been repeated in the dataset. An example record with id 168 from this Real web log data file is: 168 12 17 15 16 17 16 17 18 711 712 17 713 714. The frequent patterns generated for the same dataset by all algorithms are the same indicating correct implementations of the techniques. To accommodate a sixth algorithm (IncSpan), all six algorithms were run on a single user Windows XP Media Center Edition Service Pack 2 Personal Computer environment with processor speed of 1.60GHz Intel core and with 1GB random access memory. Five of the algorithms were at first compared on multiuser UNIX SUN microsystem with a total of 16384 MB memory and 8×1200 MHz processor speed where the proposed techniques maintain better performances consistent with results obtained in the Windows platform now being reported next.

4.1.1 Experimental set up and parameters

While experiments 1–5, and 8 compare execution times of the programs, experiments 6, and 7 compare their memory usage needs collected through the Windows Ctrl/Shift/Esc (which opens up Task Manager), then going to process tab and getting the programs' memory usages as the programs run. The same was accomplished on

Unix with (top -U account) command as the programs run. Experiment 5 further demonstrates scalability of the proposed techniques by running the algorithms on large dataset of 2.5 million rows. Five of the six algorithms compared (RePL4UP, PL4UP, GSP+, MFS+ and PLWAP) were implemented by us in C++ and compiled with "g++ filename" before execution with a.out. The sixth algorithm, IncSpan is also a C++ program, which we downloaded its available executable code from the Illimine project (http://illimine.cs.uiuc.edu/) and run as described later.

4.1.2 Dataset descriptions

For the synthetic data, the average size of transactions (length of sequences) (T) is 10, $|T| = 10$, average length of maximal pattern (that is, average number of items in the longest frequent sequence) (S) is 5, or $|S| = 5$, number of items or events (N) (the total number of attributes) is 30, $N = 30$ for all but experiment 8 (which tests behavior with many small items when $N = 100$). N is kept at 30 to increase number of frequent patterns generated. A dataset described as T10.S5.N30.U200 K has 200,000 rows, 30 attributes, average length of longest pattern as five and average length of transaction sequence as ten. Three other parameters updated database U, incremental database, db and minimum support $s\%$ affect performance of the algorithms, thus, the experiments have been run to keep two of the variables fixed, while the other varies.

4.1.3 Dataset generation and preparations

To run the programs, we first generate what will become the updated database U using the synthetic data generator. Then, we use a C++ split program written by us, to obtain the original database (DB) and the incremental database (db) by splitting the generated U into $n\%$ of U as DB and $(100 - n)\%$ of U as db. The commands used for generating the datasets we used are summarized next. The synthetic data generator parameters are ncust (for number of rows), nitems (for number of attributes), seq.npats (for number of sequential patterns), seq.patlen (for average length of maximal pattern), fname (for filename). To generate dataset T10.S5.N30.U200 K on our Unix server we used the command:

gen seq -ncust 25.9 -nitems 0.03 seq.patlen 5 -fname 200 K
To generate data T10.S5.N30.U2.5 M, we used the command:
gen seq -ncust 323.82 -nitems 0.03 seq.patlen 5 -fname 2_5 K
To generate data T10.S5.N100.U200 K, we used the command:
gen seq -ncust 25.82 -nitems 0.1 -seq.npats 50 seq.patlen 5 -fname 6_200 K

Thus, with the DB part of U in a data file called Original.data, and the db part in a file called Incremental.data, we can run executable Repl4up.o or pl4up.o, GSPplus.o, MFSplus.o to get the frequent patterns. For the IncSpan program, we converted the generated DB (data before update) and U (data after update) to the format for IncSpan data using a conversion program we wrote called change2incSpan.cpp. Thus, to create the two needed data files 'before' and 'after', we would first issue the following commands:change2incSpan Original.data before

Table 3 Expt 1: Execution times for dataset at different supports

Algorithms	CPU time (in secs) at supports of							
	0.05%	0.1%	0.5%	1%	5%	10%	15%	20%
FPs	2320	1249	265	134	20	9	6	3
F_1's	30	30	30	28	19	9	6	3
PLWAP	14.282	9.047	3.843	3.125	2.438	2.025	2.015	2.016
PL4UP	2	1.094	0.5	0.36	0.11	0.234	0.235	0.234
RePL4UP	2.21	1.957	0.813	0.422	0.265	0.35	0.375	0.351
IncSpan	2.53	2.312	2.281	2.13	1.5	1.59	1.343	1.39
GSP+	737.096	390.033	108.43	68.672	21.219	5.922	4.437	2.421
MFS+	235.709	187.096	85.346	51.656	17.297	5.938	3.547	2.407

change2incSpan Updated.data after
Next, we would run the IncSpan program on the 'before' and 'after' data with appropriate parameters as: increspan 2 before after 0.01 30 0.8 0.1
This runs IncSpan on a support of 0.01, large item found in sequence is 30 (meaning 30 attributes), and buffer ratio of 0.8 and percentage of sequences in U that is modified (similar to db percentage) is 0.1.

Experiment 1 Execution times for datasets at different supports: This experiment monitors performance of PLWAP, RePL4UP, PL4UP, GSP+, MFS+ and IncSpan algorithms on updated database size of $|U| = 200{,}000$ records, and incremental database db size of 10% of U, at changing support thresholds of between 0.05% = 0.005 of records and 20% = 0.20 of records. The dataset description is given as T10.S5.N30.U200 K. The experimental results are shown in Table 3 and Fig. 7(a).

Note that the first two rows of the tables show the number of frequent patterns (FPs) and number of frequent 1-items. The figures show only the most relevant comparisons. For example, since the GSP+ and MFS+ execution times are much higher than the proposed techniques, including them in all figures does not allow meaningful comparisons with the IncSpan algorithm. Incremental updates are most useful at lower minimum support thresholds when more patterns are frequent and when the incremental db is as small as 10% of updated database, U. At lower support threshold, it can be seen that the two proposed incremental algorithms (RePL4UP and PL4UP) are nearly ten times faster than the non-incremental algorithm, PLWAP, between twice and 13 times faster than IncSpan incremental sequential miner, over 50 times faster than the MFS+ and 300 times faster than the GSP+ incremental sequential miners for the same support. This is because the proposed techniques avoid scanning original databases but use direct access of efficient compressed versions of the small item and potentially frequent database in the form of small code profile and potential frequent small items.

Experiment 2 Execution time for updated databases with different sizes: We use different updated database (U) sizes that vary from forty thousand to two hundred

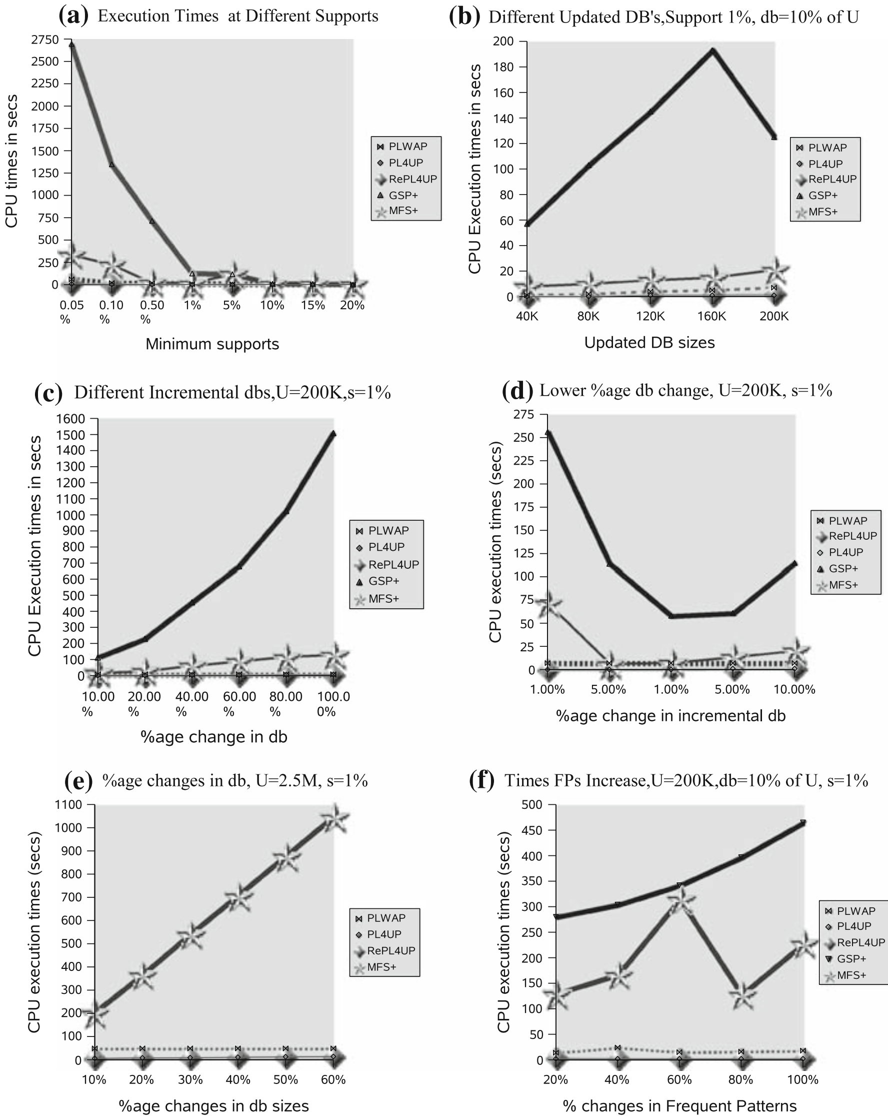

Fig. 7 Execution times trend with different supports/updated database sizes

thousand to compare the six algorithms. The minimum support of 1% is used and the result of the experiments is shown in Table 4 and Fig. 7(b) when the size of inserted and deleted records (incremental database db) in each updated database (U) is 10% of U. The five datasets used for the experiment are T10.S5.N30.U40 K, T10.S5.N30.U80 K, T10.S5.N30.U120 K, T10.S5.N30.U160 K, and T10.S5.N30.U 200 K. From the results of Table 4 and Fig. 7(b), we observe that: at a fixed minimum support of 1%=0.01 of records and fixed incremental database (db) size of 10% of entire database (U), as the updated database size increases, the execution time

Table 4 Expt 2: Execution times at different updated database sizes on support 1% = 0.01

Algorithms (times in secs)	Different main updated database sizes				
	40 K	80 K	120 K	160 K	200 K
FPs	132	132	134	134	134
F_1's	29	29	29	29	28
PLWAP	0.828	1.391	1.922	2.532	3.078
PL4UP	0.062	0.109	0.157	0.219	0.25
RePL4UP	0.172	0.25	0.297	0.36	0.438
IncSpan	0.344	0.859	0.906	1.546	2.13
GSP+	23.672	29.344	39.719	52.782	68.672
MFS+	13.5	24.031	31.468	42.953	51.656

for mining the updated U increases a lot faster with the non-incremental algorithm, PLWAP. The proposed two incremental algorithms, RePL4UP and PL4UP outperform the three existing incremental algorithms MFS+ and GSP+ by over 50 times, and IncSpan by between twice to 10 times. The numbers of frequent patterns and frequent 1-items are also listed as 134 FPs and 28 items in the F_1 list for when the size of U is 200K and db is 10% of U and the reasons for proposed algorithms' performance are the same as presented in Experiment 1.

Experiment 3 Execution times at different higher incremental db sizes: This experiment is used to compare the effect on execution times of the proposed algorithms with the PLWAP, GSP+ and MFS+ and IncSpan at fixed updated database U of 200 K records (T10.S5.N30.U200 K), fixed support of 1% = 0.01 of records, but changing sizes of the incremental database (db) of 10, 20, 40, 60, 80 and 100% of the entire updated database (U) size. Table 5 and Fig. 7(c) show the results of this experiment.

From this experiment, it can be seen that although the benefit of incremental algorithms are higher when the incremental database is much smaller than the updated database (e.g., when the percentage of the incremental db to the updated U is lower than 10%), the incremental algorithms RePL4UP and PL4UP consistently run faster than the non-incremental algorithm, PLWAP, and the other incremental algorithms GSP+, MFS+ and IncSpan when the incremental db is less than 60% of U. When the incremental db is over 60% of U, the IncSpan algorithm begins to run faster than the proposed algorithms. This is because the overhead of building, repairing the old tree and updating the structures for next round mining with the small code as well as managing a relatively sizeable incremental db tree, has surpassed the gain obtained from avoiding the scanning of projected parts of original DB several times as done by IncSpan. However, even when db is 100% of U incremental RePL4UP and PL4UP perform better than the PLWAP, meaning that the incremental algorithm may be used to build and mine the entire database faster than the non-incremental PLWAP algorithm and the lower the minimum support and higher the number of found frequent patterns the higher the benefits of RePL4UP and PL4UP. For example, for the same

Table 5 Expt 3: Execution times at different incremental higher db sizes on support 1%

Algorithms (times in secs)	Different incremental database sizes (Higher db sizes)					
	20 K 10%	40 K 20%	80 K 40%	120 K 60%	160 K 80%	200 K 100%
FPs	134	134	134	134	134	134
F_1's	28	28	28	28	28	28
PLWAP	3.062	3.062	3.062	3.062	3.062	3.062
PL4UP	0.265	0.469	0.938	1.312	2.313	2.39
RePL4UP	0.438	0.75	1.55	2.047	2.781	2.454
IncSpan	2.125	2.468	1.579	1.686	1.735	1.532
GSP+	68.672	104.8	195.111	273.545	358.231	443.37
MFS+	51.656	74.705	120	167.64	217.64	288.484

Table 6 Expt 4: Execution times at different incremental lower db sizes on support 1%

Algorithms (times in secs)	Different incremental database sizes				
	200 .1%	1 K .5%	2 K 1	10 K 5%	20 K 10%
FPs	134	134	134	134	134
F_1's	28	28	28	28	28
PLWAP	3.063	3.063	3.063	3.063	3.063
PL4UP	0.15	0.031	0.047	0.157	0.265
RePL4UP	0.062	0.125	0.125	0.312	0.5
IncSpan	2.268	2.702	1.655	1.718	2.047
GSP+	30.724	15.555	20.344	34.827	68.672
MFS+	25.581	12.593	10.689	25.862	51.656

set up, when the minimum support is 0.1% = .001 frequency, while the PLWAP time is 3.062sec, the RePL4UP takes 2.454 secs to rebuild and mine, and PL4UP took 2.39 sec when incremental db = 100% of updated DB.

Experiment 4 Execution times at different lower incremental db sizes: This experiment is used to compare the effect on execution times of the proposed algorithms RePL4UP, PL4UP and the PLWAP, IncSpan, GSP+, MFS+ at fixed updated database U of 200 K records described as T10.S5.N30.U200 K, fixed support of 1% = 0.01 frequency, but changing sizes of the incremental database (db) of 0.1, 0.5, 1, 5, and 10% of the entire updated database (U) size. Table 6 and Fig. 7(d) show the results of this experiment.

From this experiment, it can be seen that while the non-incremental algorithm PLWAP runs at a constant time for this size of updated database, as the size of incremental database db changes, the running times of the incremental algorithms RePL4UP

and PL4UP do not increase but remain between 0 and 1 seconds for this low incremental database sizes. RePL4UP and PL4UP continue to clearly outperform the other incremental sequential mining algorithms IncSpan by magnitude as high as 36 times better, while they are even better than the MFS+ and GSP+ by as high as 400 times at lower minimum supports. Good performance here is because at lower incremental db, the small code profile is not too big and the incremental db itself does not turn into a huge db with higher overhead cost, causing the proposed algorithms to perform very well.

Experiment 5 Scalability: CPU times at different lower incremental db sizes: This experiment tests for scalability by using a much larger database size of U of 2.5 million records with description as T10.S5.N30.U25000 K. The effect on execution times of the proposed algorithms RePL4UP, PL4UP and the PLWAP, IncSpan, GSP+, MFS+ at this very large fixed updated database U of 2.5 million records, fixed support of 1% = 0.01 frequency, but changing sizes of the incremental database (db) of 0.1, 0.5, 1, 5, and 10% of the entire updated database (U) size are compared. Table 7 and Fig. 7(e) show the results of this experiment.

From this experiment, it can be seen that while the non-incremental algorithm PLWAP runs at a constant time for this size of large updated database, as the size of incremental database db changes, the running times of the incremental algorithms RePL4UP and PL4UP increase only marginally in comparison to the other algorithms (IncSpan) when the incremental database goes over 60% of the original DB because of the increased overhead of managing both sizeable small codes and incremental db.

For the storage space demands (as reported on Table 9 experiments on storage space demands), it can be seen that while the maximum memory needed by the RePL4UP for this size of data is 19MB, the memory demands for the PL4UP is 16MB. On the contrary, the IncSpan algorithm demands 50MB of memory for this large dataset. The GSP based incremental algorithms demand about 5MB and 3MB of memory for the same dataset. Thus, while the memory demands depend to some extent on implementations as arrays are used in the RePL4UP and PL4UP current implementations to hold old frequent patterns, the hard disk size of the 2.5 million records is 105MB and making the about 17MB memory demand for processing such data for fast speed very reasonable. Moreover, the RePL4UP and PL4UP have much better reasonable memory demands than the IncSpan algorithm for very large datasets and about the same demands for regular sized datasets.

Experiment 6 Storage space usages at changing incremental database sizes: This experiment is used to compare the memory space demands of the proposed RePL4UP, PL4UP algorithms and the PLWAP, IncSpan, GSP+, MFS+ at fixed updated database U of 200 K records (T10.S5.N30.U200 K), fixed support of 1% = 0.01 frequency, but changing sizes of the incremental database (db) of 10, 20, 40, 60, 80 and 100% of the entire updated database (U) size. The maximum memory needed by each program is recorded by running the Windows Task manager (Ctrl/Shift/Esc) as the programs are running. It can be seen that the programs demand about the same amount of memory for different changes in incremental database. Both the RePL4UP and PL4UP programs demand slightly more memory of around 5MB than the non-incremental

Table 7 Expt 5 on Scalability: CPU times at different incremental db sizes on support 1%

Algorithms (times in secs)	Different incremental database sizes					
	250 K 10%	500 K 20%	750 K 30%	1 M 40%	1.25 M 50%	1.5 M 60%
FPs	136	136	136	136	136	136
F_1's	29	29	29	29	29	29
PLWAP	32.672	32.672	32.672	32.672	32.672	32.672
PL4UP	2.859	4.719	7.515	9.407	12.703	13.062
RePL4UP	3.891	6.375	9.687	11.235	14.698	14.922
IncSpan	13.031	14.862	11.687	12.532	14.85	12.366
GSP+	643.291	1253.16	1652.56	1963	2571.95	3191.566
MFS+	258.481	517.073	652.882	884.25	1035.366	1394.1

Table 8 Expt 6: Memory demands for $U = 200$ K at support 1%

Algorithms	Different incremental database sizes					
	20 K 10%	40 K 20%	80 K 40%	120 K 60%	160 K 80%	200 K 100%
PLWAP	4.42	4.42	4.42	4.42	4.42	4.42
PL4UP	4.9	5.05	5.15	5.2	5.03	4.6
RePL4UP	5.45	5.66	5.72	5.86	5.69	5.46
IncSpan	3.96	3.96	3.52	3.96	3.96	3.96
GSP+	2.9	2.9	2.9	2.9	2.9	2.9
MFS+	3.88	3.88	3.88	3.88	3.88	3.88

PLWAP algorithm requiring about 4MB of memory. While for this regular size of data, the IncSpan requires about 4 MB of memory, the GSP+ requires about 3MB and MFS+ requires about 4 MB. Thus, for this regular size of data, there is no significant difference in the memory demands of all these programs. Table 8 shows the results of this experiment.

While the RePL4UP and PL4UP store the compressed data in the form of PLWAP tree for speed, the GSP based algorithms as well as the IncSpan algorithm only scan the database several times for support when those cannot be deduced and also do not store any data in arrays making for slightly lower memory demands for the algorithms for regular sized dataset. The RePL4UP also keeps only small code profiles while the PL4UP algorithm stores a bigger PLWAP tree. Thus, these with the storage of patterns in arrays account for the slightly higher storage needs, which are still very reasonable for the speed gain and considering that when the dataset size increases, the memory demands of IncSpan very much surpasses that of RePL4UP and PL4UP (as shown in Table 9) which still outperforms it a lot in speed.

Experiment 7 Storage space usages as size of updated database increases: This experiment compares the memory space demands of the proposed RePL4UP, PL4UP

Table 9 Expt 7: Memory demands for increasing *U* database sizes at support 1%

Algorithms	Increasing updated database sizes					
	40 K	80 K	120 K	160 K	200 K	250000 K
PLWAP	2.72	3.27	3.684	4.08	4.42	14.42
PL4UP	2.87	3.5	3.985	4.45	4.9	16.78
RePL4UP	3.18	3.89	4.387	4.98	5.45	18.7
IncSpan	0.781	1.574	2.361	3.16	3.96	49.947
GSP+	2.95	2.98	2.98	2.98	2.98	2.99
MFS+	3.88	3.88	3.88	3.88	3.88	3.88

algorithms and the PLWAP, IncSpan, GSP+, MFS+ as the updated database sizes increase at fixed support of 1% = 0.01 frequency. Thus, the U sizes compared are T10.S5.N30.U40 K, T10.S5.N30.U80 K, T10.S5.N30.U120 K, T10.S5.N30.U160 K, T10.S5.N30.U200 K, T10.S5.N30.U250000 K. The incremental database (db) is 10% of the entire updated database (U) size. It can be seen that the IncSpan program is the most affected by the increase in size of data as its memory demand increases from 4MB for 200 K records to 50MB for 2.5 million records. The PL4UP and RePL4UP remain at around 17MB for 2.5 million records indicating the scalability of the proposed approaches even for memory demands. Table 9 shows the results of this experiment.

Experiment 8 Many small items: execution times at support 3%: This experiment monitors execution times when we drastically increase the number of small 1-items in the database. This, we did by increasing the number of attributes to 100 (not 30 as in the earlier experiments) and also using a parameter in the data generation program to force fewer frequent patterns to be found in the dataset. The gen command used to generate each of the datasets of experiments 1–8 are described earlier in sect. 4. The dataset for this experiment is described as T10.S5.N100.U200 K and has 200 thousand records with 100 attributes. To collect adequate data for comparisons, we ran the experiment on the same U but for changing incremental database sizes of 10, 20, 30, 40, 50, 60, 80% of U. The results of this experiment are shown in Table 10.

It can be seen that the number of frequent 1-items F_1 is 31, while the number of small 1-items S_1 is 67 in all cases since the same U is used but split into original DB and incremental db of different percentages. Thus, with the number of small items being as large as 67% of all candidate 1-items, the proposed PL4UP algorithm always outperforms all the other incremental algorithms by far. The RePL4UP algorithm also outperforms all, including IncSpan up until when the percentage of incremental database is higher than the unchanged database. This is because the sizes of both the incremental db tree and the small items are bigger resulting in higher maintenance overhead. Thus, when there are many small items and the incremental database is bigger than the unchanged database, while the performance of the traditional PL4UP remains better, that of the RePL4UP degrades.

Table 10 Expt 8: Execution times on DB with more small items, 100 attrs

Algorithms (times in secs)	% of U that is incremental db (Higher dbs)						
	10%	20%	30%	40%	50%	60%	80%
FP's	36	36	36	36	36	36	36
F_1's	31	31	31	31	31	31	31
S_1's	67	67	67	67	67	67	67
PLWAP	3.83	3.83	3.83	3.83	3.83	3.83	3.83
PL4UP	0.157	0.344	0.625	0.891	0.667	1.093	1.094
RePL4UP	0.297	0.672	1.266	1.469	2.735	3.515	5.828
IncSpan	1.578	2.139	1.672	1.655	1.782	1.672	1.735
GSP+	52.047	113/906	151/375	196.140	240.953	300.312	402.546
MFS+	45.875	73.406	106.063	143.719	157.156	177.485	215.697

4.2 Time complexities of RePL4UP and PL4UP algorithms

Let P be the total number of frequent patterns to be extracted from old DB tree, and p the total frequent patterns extracted from incremental db tree. Also, F is the number of frequent 1-items in original DB, while f is the number of frequent 1-items in incremental db and number of frequent 1-items in updated database, U, is F'. If N is the number of sequences in DB and L is the length of the longest frequent sequence in DB. Similarly, small n is the number of sequences in the small incremental database db, while l is the length of the longest sequence in the incremental database. The time complexities of the three algorithms PLWAP (Ezeife and Lu 2005) and the two proposed algorithms RePL4UP and PL4UP are discussed in three parts to represent: (1) Number of database scans done by each algorithm, (2) Construction and updating of the tree and (3) Mining the tree.

1. Number of Database scans as the incremental database size increases: Although all the algorithms scan the database twice, the two proposed algorithms do not scan the old DB and thus their time growth is only affected by the size of the incremental database and not by the size of the old database. PLWAP has O(N+n)(L+l) while both RePL4UP and PL4UP have O(nl).
2. Construction or updating of the PLWAP tree: Since each frequent sequence in the entire updated database consisting of DB + db, is used in constructing the tree, time complexity is O(N+n)(L+l), while for RePL4UP (revising the old tree takes O(Bl) for B number of branches of the tree) and PL4UP using only incremental db needs O(nl).
3. Mining the updated tree: Since for each frequent 1-item, a continuous prefix frequent pattern is mined during each iteration, with the PLWAP, it is O(F'T), for RePL4UP, it is O(Ft) where t is the number of modified nodes of the tree, and for PL4UP, it is O(ft) for t number of nodes in the incremental tree.

Table 11 Overhead of RePL4UP and PL4UP algorithms at minsup 5%=0.05

Data set	DBsize in # of events	PLWAP in % of DB	Scode in % of DB	RePL4UP in % of DB	PL4UP in % of DB	Build time(s) ReP+ Scode	Build times(s) PLWAP	Build times(s) PL4UP
1 K	5767	26%	2%	28%	28%	0 s	0 s	1 s
100 K	593214	5	1.2	6.2	5.8	4	4	4
200 K	1181353	3.1	1.2	4.3	4.3	7	6	7
400 K	2268304	2	1.2	3.2	3.3	13	13	14
600 K	3451710	1.6	1.2	2.8	2.8	22	20	21
800 K	4494023	1.2	1.2	2.4	2.5	28	26	30
1 M	5683824	1.1	1.1	2.2	2.3	38	33	37

4.3 Overhead of the proposed RePL4UP and PL4UP algorithms

In comparison with the non-incremental PLWAP algorithm, the incremental techniques RePL4UP carries the overhead of computing and storing position codes of small items in all database sequences in the form of small code profile as the PLWAP tree is being built, while the PL4UP algorithm stores a bigger PLWAP tree that stores some non-frequent events. The needed additional storages by the proposed algorithms are still less than accessing the entire database as the small code profile data are accessed directly to obtain the codes of a small item used to repair the RePL4UP tree and not sequentially as a database is accessed. All three algorithms scan the original database twice to build the compressed tree, which is now scanned once recursively to mine the frequent patterns. Thus, we measure the size of storage needed by counting the number of events or nodes each algorithm stores. The more the number of events stored and accessed during mining, the higher the execution time of the algorithm as well. On Unix Solaris system, we ran a number of experiments on datasets of sizes 1000 to 100,000 records and up to 1 million records at different minimum support thresholds of 10, 5, 2 and 1%, where the datasets have number of attributes $N = 100$, average length of sequences $T = 10$ and average number of items in longest frequent patterns $S = 5$. To compare the overheads of the PLWAP, RePL4UP and PL4UP algorithms, for each database size, we collected the sizes of the original database, compressed PLWAP, RePL4UP, PL4UP trees, small code profile, time to build these compressed database structures. We represent the size of these structures as the number of events or nodes stored by the structures. A summary of result of our runs at minimum support of $5\% = 0.5$ is presented as Table 11.

It can be seen that in both the RePL4UP and the PL4UP algorithms, the size of the compressed tree and small code is about 3% of the size of the original database and the additional cost of building the structures in comparison to the cost of just building the PLWAP tree is just about 4 s even for the 1 million records. At this minimum support threshold, the small code database size is around 1% of the original DB, but it may be larger than the tree itself depending on the minimum support and the nature of the

sequences in the DB. With the same datasets, at a support of 1%, there are no small codes because all 1-items are frequent.

5 Conclusions and future work

Two incremental web sequential mining algorithms, RePL4UP and the PL4UP, are proposed in this paper. The RePL4UP algorithm revises an existing PLWAP tree by utilizing the metadata of old database transactions as well as old mined frequent patterns in order to incrementally update web log sequential patterns. One major contribution of work is the technique for efficiently using position codes of small items in database sequences to restore information about previous small items that were not stored in the tree, when the database is updated and these items become frequent, without re-scanning the entire old database. The second algorithm, the PL4UP adopts a lower minimum tolerance support middle line approach of t%, determined with average rate of change of data in the system. The goal of PL4UP is to predict items that are likely to become frequent when the database is updated so that their information will be stored in the tolerance PLWAP tree. Thus, frequent patterns can be updated without scanning the entire old database. The proposed RePL4UP and PL4UP algorithms inherit the advantages of PLWAP, and no huge candidate itemsets need to be generated. They also fully utilize old existing information like the old patterns and tree for mining the updated database. Experiments show that the proposed algorithms outperform three prominent incremental mining algorithms, GSP+, MFS+ and IncSpan and in particular, when the incremental database is less than 50% of the updated database. Even when there are too many small items in the database, the proposed algorithms outperform the other three when the incremental database is smaller than the unchanged database. While there is no significant difference in execution times of the RePL4UP and PL4UP algorithms, the RePL4UP is more sophisticated in its approach and would always find all patterns while the competitiveness of the results by the PL4UP depends on how good the predicted tolerance support t% is. The PL4UP has the advantage of conceptual simplicity and has found complete patterns in our experiments. Most importantly, the two proposed incremental web sequential algorithms always perform better than the non-incremental PLWAP algorithm even for mining the entire database except when there are too many small items (over 50% of candidate items being small). Note that the non-incremental PLWAP algorithm had been shown in Ezeife and Lu (2005) to be more efficient than other non-incremental algorithms like the WAP-tree and GSP algorithms.

Future work should investigate combining these two algorithms for limited memory data stream applications and encoding the entire tree using the position codes of the items. A more efficient implementation of PLWAP's position code management scheme even for extremely long sequences would allow the incremental RePL4UP and PL4UP algorithms handle all datasets including those with too many small items more efficiently. As these techniques are efficient and work for only single element set sequences like those for web log sequences, it might be possible to extend the approach to general multi-element sequences to take advantage of its speed. These techniques can apply well to distributed, object-oriented, and parallel mining that may involve continuous time series data, and to web content and text mining.

Acknowledgments This research was supported by the Natural Science and Engineering Research Council (NSERC) of Canada under an operating grant (OGP-0194134) and a University of Windsor grant.

References

Agrawal R, Srikant R (1995) Mining sequential patterns. In: Proceedings of the 11th Int'l conference on data engineering, Taipei, pp 3–14

Berendt B, Spiliopoulou M (2000) Analyzing navigation behavior in web sites integrating multiple information systems. VLDB Journal, Special Issue on Databases and the Web 9(1):56–75

Cheung H, Yan X, Han J (2004) IncSpan: incremental mining of sequential patterns. In: Proceedings of the ACM SIGKDD international conference on knowledge discovery and data mining, Seattle, pp 527–532

Cheung DW, Han J, Ng VT, Wong CY (1996) Maintenance of discovered association rules in large database: an incremental updating technique. In: Proceedings of the 12th international conference on data Engineering, New Orleans

Cheung D, Kao B, Lee J (1997) Discovering user access patterns on the world wide web. In: Proceedings of the 1st Pacific-Asia conference on knowledge discovery and data mining (PAKDD'97)

El-Sayed M, Carolina R, Elke AR (2004) FS-miner: efficient and incremental mining of frequent sequence patterns in web logs. In: Proceedings of the 6th ACM international workshop on web information and data management, Washington DC, pp 128–135

Ezeife CI, Chen M (2004a) Mining web sequential patterns incrementally with revised PLWAP tree. In: Proceedings of the fifth international conference on web-age information management (WAIM 2004) Dalian, published in LNCS by Springer, pp 539–548

Ezeife CI, Chen M (2004b) Incremental mining of web sequential patterns using PLWAP tree on tolerance minsupport. In: Proceedings of the IEEE 8th international database engineering and applications symposium (IDEAS04), Coimbra, pp 465–479

Ezeife CI, Lu Y (2005) Mining web log sequential patterns with position coded pre-order linked WAP-tree. Int J Data Mining Knowl Discov, Kluwer Acad Publ 10:5–38

Ezeife CI, Lu Yi, Liu Yi (2005) PLWAP sequential mining: open source code proceedings of the open source data mining workshop on frequent pattern mining implementations, in conjunction with ACM SIGKDD, Chicago, August 21–24, pp 26–29

Han J, Kamber M (2001) Data mining: concepts and techniques Morgan Kaufmann

Han J, Pei J, Yin Y, Mao R (2004) Mining frequent patterns without candidate generation: a frequent-pattern tree approach. Int J Data Mining Knowl Discov, Kluwer Acad Publ 8(1):53–87

Kao B, Zhang M, Yi C-L, Cheung DW (2005) Efficient algorithms for mining and incremental update of maximal frequent sequences. Int J Data Mining Knowl Discov, Springer Sci Publ 10:87–116

Lee Y-S, Yen S-J (2008) Incremental and interactive mining of web traversal patterns. Inform Sci 178(2):287–306

Liu J-W, Yu S-J, Le J-J (2003) Online mining dynamic web news patterns using machine learn methods. FSKD Conference, Springer Lecture Notes in AI 3614, pp 462–465

Lu Yi, Ezeife CI (2003) Position coded pre-order linked WAP-tree for web log sequential pattern mining. In: Proceedings of the 7th Pacific-Asia conference on knowledge discovery and data mining (PAKDD 2003), Seoul, Korea

Masseglia F, Poncelet P, Cicchetti R (1999) An efficient algorithm for web usage mining. Netw Inform Syst J 2(5–6):571–603

Masseglia F, Poncelet P, Teisseire M (2003) Incremental mining of sequential patterns in large databases. Data Knowl Eng 46(1):97–121

Nanopoulos A, Manolopoulos Y (2000) Finding generalized path patterns for web log data mining. Data Knowl Eng 37(3):243–266

Nanopoulos A, Manolopoulos Y (2001) Mining patterns from graph traversals. Data Knowl Eng 37(3):243–266

Nguyen S, Sun X, Orlowska M (2005) Improvements of incSpan: incremental mining of sequential patterns in large database. In: Proceedings 2000 Pacific-Asia conference on knowledge discovery and data mining (PAKDD'05), pp 442–451

Ou J-C, Lee C-H, Chen M-S (2008) Incremental web log mining with dynamic threshold. VLDBJ 17:827–847

Parthasarathy S, Zaki MJ, Ogihara M, Dwarkadas S (1999) Incremental and interactive sequence mining. In: Proceedings of the 8th international conference on information and knowledge management (CIKM99), Kansas City, pp 251– 258
Pei J, Han J, Mortazavi-asl B, Zhu H (2000) Mining access patterns efficiently from web logs. In: proceedings 2000 Pacific-Asia conference on knowledge discovery and data mining (PAKDD'00), Kyoto, pp 396–407
Pei J, Han J, Mortazavi-Asl B, Pinto H (2001) PrefixSpan: mining sequential patterns efficiently by prefix-projected pattern growth. In: The proceedings of the 2001 international conference on data engineering (ICDE '01), pp 215–224
Srikant R, Agrawal R (1995) Mining generalized association rules. In: Proceedings of the 21st int'l conference on very large databases (VLDB), Zurich
Spiliopoulou M (1999) The laborious way from data mining to web mining. J Comput Syst Sci Eng, Special Issue Semant Web 14:113–126
Tang P, Turkia M (2007) Mining frequent web access patterns with partial enumerations. 45th ACM Annual Southeast Regional Conference, 23–24 March 2007, Winston-Salem, N.Carolina, pp 226–231
Wang K (1997) Discovering patterns from large and dynamic sequential data. J Intell Inform Syst 9(1):33–56
Wang K, Tan J (1996) Incremental discovery of sequential patterns. In: Proceedings of the ACM workshop on research issues on data mining and knowledge discovery, Montreal
Yen S-J, Lee Y-S (2006) An incremental data mining algorithm for discovering web access patterns. Int J Bus Intell Data Mining 1(3):288–303
Zake MJ (2000) SPADE: an efficient algorithm for mining frequent sequences. Mach Learn 42:31–60
Zhang M, Kao B, Cheung D, Yip C-L (2002) Efficient algorithms for incremental update of frequent sequences. In: Proceedings of the sixth Pacific-Asia conference on knowledge discovery and data mining (PAKDD), pp 186–197
Zhang M, Kao B, Yip C-L (2002) A comparison study on algorithms for incremental update of frequent sequences. In: Proceedings of the IEEE international conference on data mining ICDM, pp 554–561