An Efficient Approach for Mining Weighted Sequential Patterns in Dynamic Databases

Sabrina Zaman Ishita, Faria Noor, and Chowdhury Farhan Ahmed(✉)

Department of Computer Science and Engineering,
University of Dhaka, Dhaka, Bangladesh
sz.ishita@gmail.com, faria.noor93@gmail.com, farhan@du.ac.bd

Abstract. In modern time of flowing data where more data accumulate every minute than we can store or make sense of, a fast approach for analyzing incremental or dynamic database has a lot of significance. In a lot of instances, the data are sequential and the ordering of events has interesting meaning itself. Algorithms have been developed to mine sequential patterns efficiently from dynamic databases. However, in real life not all events bear the same urgency or importance, and by treating them as equally important the algorithms will be prone to leaving out rare but high impact events. Our proposed algorithm solves this problem by taking both the weight and frequency of patterns and the dynamic nature of the databases into account. It mines weighted sequential patterns from dynamic databases in efficient manner. Extensive experimental analysis is conducted to evaluate the performance of the proposed algorithm using large datasets. This algorithm is found to outperform previous method for mining weighted sequential patterns when the database is dynamic.

Keywords: Dynamic databases · Weighted sequential pattern
Weighted support · Incremental mining

1 Introduction

Data Mining is the analysis of data, usually in large volumes, for uncovering useful relationships among events or items that make up the data. Frequent pattern mining is an important data mining problem with extensive application; here patterns are mined which occur frequently in a database. Another important domain of data mining is sequential pattern mining where the ordering of items in a sequence is important. Unless weights (value or cost) are assigned to individual items, they are usually treated as equally valuable. However, that is not the case in most real life scenarios. When the weight of items is taken into account in a sequential database, it is known as weighted sequential pattern mining.

As technology and memory devices improve at an exponential rate, their usage grows along too, allowing for the storage of databases to occur at an even higher rate. This calls for the need of incremental mining for dynamic databases whose data are being continuously added. Most organizations that generate and collect data on a daily basis have unlimited growth. When a database update occurs, mining patterns from scratch is costly with respect to time and memory. It is clearly unfeasible. Several approaches have been adopted to mine sequential patterns in incremental database that avoids mining from scratch. This way, considering the dynamic nature of the database, patterns are mined efficiently. However, the weights of the items are not considered in those approaches.

Consider the scenario of a supermarket that sells a range of products. Each item is assigned a weight value according to the profit it generates per unit. In the classic style of market basket analysis, if we have 5 items {“milk”, “perfume”, “gold”, “detergent”, “pen”} from the data of the store, the sale of each unit of gold is likely to generate a much higher profit than the sales of other items. Gold will therefore bear a high weight. In a practical scenario, the frequency of sale of gold is also going to be much less than other lower weight everyday items such as milk or detergent. If a frequent pattern mining approach only considers the frequency without taking into account the weight, it will miss out on important events which will not be realistic or useful. By taking weight into account we are also able to prune out many low weight items that may appear a few times but are not significant, thus decreasing the overall mining time and memory requirement.

Existing algorithms for mining weighted sequential patterns or mining sequential patterns in incremental database give compelling results in their own domain, but have the following drawbacks: existing sequential pattern mining algorithms in incremental database do not consider weights of patterns, though low-occurrence patterns with high-weight are often interesting, hence they are missed out if uniform weight is assigned. Weighted sequential patterns are mined from scratch every time the database is appended, which is not feasible for any repository that grows incrementally. These motivated us to overcome these problems and provide a solution that gives better result compared to state-of-the-art approaches. In our approach we have developed an algorithm to mine weighted sequential patterns in an incremental database that will benefit a wide range of applications, from Market Transaction and Web Log Analysis to Bioinformatics, Clinical and Network applications.

With this work we have addressed an important sub-domain of frequent pattern mining where several categories such as sequential, weighted and incremental mining collide. Our contributions are: (1) the construction of an algorithm, *WIncSpan*, that is capable of mining weighted sequential patterns in dynamic databases continuously over time. (2) Thorough testing on real life datasets to prove the competence of the algorithm for practical use. (3) Marked improvement in results of the proposed method when compared to existing algorithm.

The paper is organized as follows: Sect. 2 talks about the preliminary concepts and discusses some of the state-of-the-art mining techniques which directly influence this study. In Sect. 3, the proposed algorithm is developed and an example is worked out. Comparison of results of the proposed algorithm with existing algorithm is given in Sect. 4. And finally, the summary is provided as conclusions in Sect. 5.

2 Preliminary Concepts and Related Work

Let us expand our discussion to better understand the concepts that lie at the heart of mining frequent patterns of different types. Let I be the set of all items $I_1, I_2, \ldots, I_n$. A set of transactions is considered as a transaction database where each transaction is a subset of I. Sequence database is a set of sequences where every sequence is a set of events $<\mathrm{e}_1\ \mathrm{e}_2\ \mathrm{e}_3 \ldots \mathrm{e}_l>$. The order in which events or elements occur is important. Here, event e_1 occurs before event e_2, which occurs before e_3 and so on. Each event $\mathrm{e}_i \subseteq I$.

In a sequence database, support is the count of how frequently a pattern or sequence appears in the database. The support of a pattern P with respect to a sequence database is defined as the number of sequences which contain P. In Table 1, a sequence database is given along with one increment, where in first sequence, there are 2 events: (ab) and (e). For brevity, the brackets are omitted if an event has only one item. Here, (ab) occurs before (e). Given a set of sequences and a user-specified minimum support threshold *min_sup*, sequential pattern mining is regarded as finding all frequent subsequences whose support count is no less than *min_sup*. If $\alpha = <$(ab)b$>$ and $\beta = <$(abc)(be)(de)c$>$, where a, b, c, d, and e are items, then α is a subsequence of β.

Many algorithms, such as GSP [13] and SPADE [16], mine frequent sequential patterns. GSP uses Apriori based approach of candidate generate and test. SPADE uses the same approach as GSP but it maps a sequence database into vertical data format unlike GSP. They also obey the antimonotone or downward-closure property that if a sequence does not fulfill the minimum support requirement then none of its super-sequences will be able to fulfill it as well. FreeSpan [7] takes motivation from FP-Growth Tree and mines sequential patterns. SPAM [2] mines sequential patterns using a bitmap representation. PrefixSpan [12] maintains the antimonotone property and uses a prefix-projected pattern growth method to recursively project corresponding postfix subsequences into projected databases.

The usage and improvement of technology and memory devices grow at an exponential rate which means the databases are also growing dynamically. This calls for the need of incremental mining for dynamic databases whose data is being continuously added, such as in shopping transactions, weather sequences and medical records. The naive solution for mining patterns in dynamic database is to mine the updated database from scratch, but this will be inefficient since the newly appended portion of the database is often much smaller than the whole database. To produce frequent sequential patterns from dynamic database in an efficient way, several algorithms [9–11] were proposed. One of the algorithms for mining sequential patterns from dynamic databases is IncSpan [4]. Here, along with frequent sequences, semi-frequent sequences are also saved to be worked on when new increment is added. For buffering semi-frequent sequences along with frequent sequences, a buffer ratio is used. In our approach, we will use this concept for buffering weighted semi-frequent sequences for further use.

Considering the importance or weights of items, several approaches such as WSpan [15], WIP [14], WSM [5] etc. have been proposed for mining weighted frequent patterns. WSpan mines weighted frequent sequential patterns. Using the weight constraint for mining weighted sequential patterns WSpan [15] uses the prefix projected sequential pattern growth approach. According to WSpan, the weight of a sequence is defined as the average weight of all its items from all the events. For example, using the weight table provided in Table 2, we can calculate the weight of the sequential pattern $P = <abc>$ as $\mathrm{W}(P) = (0.41 + 0.48 + 0.94)/3 = 0.61$.

There exists many work in the field of weighted sequential pattern mining and in the field of incremental mining of sequential patterns separately. But there has been no complete work in the field of mining weighted sequential patterns in incremental databases. A work [8] has attempted to mine weighted sequential patterns in incremental databases, but no complete details and comparative performance analysis were provided there. We are proposing a new algorithm *WIncSpan* which provides a complete work of how weighted sequential patterns can be generated efficiently from dynamic databases and providing detailed experimental results of its performance.

3 The Proposed Approach

In previous section we discussed the preliminary concepts and existing methods of mining frequent sequential patterns separately in weighted and incremental domains. In this chapter we merge those concepts to propose a new method for weighted sequential pattern mining in incremental databases. A sequence database is given in Table 1. Here, from sequences 10 through 50 represent the initial database D and sequences 60 through 80 represent Δdb which is the new appended part of the whole database D'. The corresponding weights of the items of D' is given in Table 2.

Table 1. Appended database D'

	Sequence ID	Sequences
D	10	$<(ab)e>$
	20	$<ab>$
	30	$<a(dc)e>$
	40	$<(ab)d>$
	50	$<b(dc)e>$
Δdb	60	$<(ab)d>$
	70	$<a(dc)(ab)>$
	80	$<a(ab)e>$

Table 2. Weight table for items

Item	Weight
a	0.41
b	0.48
c	0.94
d	0.31
e	0.10

Definition 1 (Minimum Weighted Support: *minw_sup*). As we know, for a given minimum support percentage, the *min_sup* value is calculated as:

*min_sup = number of transactions in database * minimum support percentage.* We are considering the weight of the items as well, we derive a minimum weighted support threshold *minw_sup*:

$$minw\_sup = min\_sup * avgW$$

Here, $avgW$ is the average weight value. This is the average of the total weight or profit that has contributed to the database upto that point. In initial database of D', item a occurs 4 times in total, similarly b occurs 4, c occurs 2, d occurs 3 and e occurs 3 times in total. The $avgW$ is calculated as: $avgW = (4 * 0.41) + (4 * 0.48) + (2 * 0.94) + (3 * 0.31) + (3 * 0.10)/16 = 0.4169$. In initial database D, the $minw\_sup$ for minimum support 60% is therefore calculated as: $minw\_sup = 3 * 0.4169 = 1.25$ (as min_sup = 5 * 60% = 3).

Definition 2 (Possible Frequent Sequences). The possible set of frequent sequences is generated to list sequences or patterns in a database that have a chance to grow into patterns that could be frequent later. For a sequence to be possibly frequent, the following condition must be fulfilled:

$$support * maxW \geqslant minw\_sup$$

The notation $maxW$ denotes the weight of the item in the database that has maximum weight. In our example, it would be 0.94 for the item $<c>$. This value is multiplied with the support of the pattern instead of taking the actual weight of the pattern. This is to make sure the anti-monotone property is maintained, since in an incremental database a heavy weighted item may appear later on in the same sequence with less weighted items, thereby lifting the overall support of the pattern. By taking the maximum weight, an early consideration is made to allow growth of patterns later on during prefix projection. The set thus contains all the frequent items, as well as some infrequent items that may grow into frequent patterns later, or be pruned out.

Complete Set of Possible Frequent Sequences. First, the possible length-1 items are mined. For item $<a>$ in D, $support_a * maxW = 4 * 0.94 = 3.76 \geqslant minw\_sup$. The item $<a>$ satisfies the possible frequent sequence condition. Items $<b>$, $<c>$, $<d>$ and $<e>$ are found to satisfy the condition as well and therefore are added to the set of possible frequent length-1 sequences.

Possible Frequent length-1 Sequences: $\{<a>, <b>, <c>, <d>, <e>\}$

Next, the projected database for each frequent length-1 sequence is produced using the frequent length-1 sequence as prefix. The projected databases are mined recursively by identifying the local weighted frequent items at each layer, till there are no more projections. In this way the set of possible frequent sequences is grown, which now includes the sequential patterns grown from the length-1 sequences. At each step of the projection, the items picked will have to satisfy the minimum weighted support condition. For example, for item $<a>$, the projected database contains these sequences: $<(\_b)e>$, $<b>$, $<(dc)e>$, $<(\_b)d>$. And the possible sequential patterns mined from these sequences are: $<a>, <ab>, <(ab)>, <ac>, <ad>, <ae>$. In the similar way, possible sequential patterns are also mined from the projected databases with prefixes $<b>, <c>, <d>$ and $<e>$.

Definition 3 (Weighted Frequent and Semi-frequent Sequences). For static database, at this moment, only the weighted frequent sequences will be saved and others will be pruned out. Considering the dynamic nature of the database, along with weighted frequent sequences, we will keep the weighted semi-frequent sequences too. From the set of possible frequent sequences, set of Frequent Sequences *(FS)* and Semi-Frequent Sequences *(SFS)* can be constructed as follows:

$$\text{Condition for FS: } support(P) * weight(P) \geqslant minw\_sup$$
$$\text{Condition for SFS: } support(P) * weight(P) \geqslant minw\_sup * \mu$$

Here, P is a possible frequent sequence and μ is a buffer ratio. If the support of P multiplied by its actual weight satisfies the minimum weighted support *minw_sup* then it goes to the *FS* list. If not, the support of P times its actual weight is compared with a fraction of *minw_sup* which is derived from multiplying *minw_sup* by a buffer ratio μ. If satisfied, the sequence is listed in *SFS* as a semi-frequent sequence. Otherwise, it is pruned out.

For example, the single length sequence $<a>$ has weighted support $4 * 0.41 = 1.64$. Since 1.64 is greater than the *minw_sup* value 1.25, $<a>$ is added to *FS*. Considering the value of μ as 60%, $minw\_sup * \mu = 1.25 * 60\% = 0.75$. Here, $<bd>$ has support count of 2 and weight of $(0.48 + 0.31)/2 = 0.395$. Its weighted support $2 * 0.395 = 0.79$ is greater than 0.75, so it goes to SFS list. For initial sequence database D, mined frequent sequences are: $<a>, <b>, <c>, <(dc)>$ and semi-frequent sequences are: $<(ab)>, <bd>, <d>$. Other sequences from possible set of frequent sequences are pruned out as infrequent. Interestingly, we see that $<d>$ is a semi-frequent pattern but when we consider it in an event with highly weighted item $<c>$, $<(dc)>$ becomes a frequent pattern. This is possible in our approach as a result of considering the weight of sequential patterns.

Dynamic Trie Maintenance. An extended trie is constructed from *FS* and *SFS* patterns from D which is illustrated in Fig. 1. The concept of the extended trie is taken from the work [4]. Each node in the trie will be extended from its parent node as either s-extension or i-extension. If the node is added as different event, then it is s-extension, if it is added in the same event as its parent then it is i-extension. For example, while adding the pattern $<(ab)>$ to the trie, we first go to the branch labeled with $<a>$, increment its support count by the support count of $<(ab)>$, then add a new branch to it labeled with $<b>$ as i-extension. The solid lines represent the *FS* patterns and the dashed lines represent the *SFS* patterns. Each path from root to non-root node represents a pattern along with its support.

When new increments are added, rather than scanning the whole database to check the new support count of a pattern, the dynamic trie becomes handy. This trie will be used dynamically to update the support count of patterns when new increments will be added to the database. Traversing the trie to get the new support count of a pattern is performed a lot faster than scanning the whole database.

Increment to Database. At this point, if an update to the database is made, which is a common nature of most real-life datasets, it is not convenient to run the procedure from scratch. How the appended part of the database will be handled, how new frequent sequences will be generated using the *FS* and *SFS* lists, how the dynamic trie will be helpful, all are explained below.

The Proposed Algorithm. Here, the basic steps of the proposed *WIncSpan* algorithm is illustrated to mine weighted sequential patterns in an incremental database. Further, an incremental scenario is provided to better comprehend the process.

Snapshot of the Proposed Algorithm. The necessary steps for mining weighted sequential frequent patterns in an incremental database are:

1. In the beginning, the initial database is scanned to form the set of possible frequent patterns.
2. The weighted support of each pattern is compared with the minimum weighted support threshold *minw_sup* to pick out the actual frequent patterns, which are stored in a frequent sequential set *FS*.
3. If not satisfied, the weighted support of the pattern is checked against a percentage (buffer ratio) of the *minw_sup* to form the set of semi-frequent set *SFS*. Other patterns are pruned out as infrequent.
4. An extended dynamic trie is constructed using the patterns from *FS* and *SFS* along with their support count.
5. For each increment in the database, the support counts of patterns from the trie are updated.
6. Then the new weighted support of each pattern in *FS* and *SFS* is again compared with the new *minw_sup* and then compared with the percentage of *minw_sup* to check whether it goes to new frequent set FS' or to new semi-frequent set SFS', or it may also become infrequent.
7. FS' and SFS' will serve as *FS* and *SFS* for next increment.
8. At any instance, to get the weighted frequent sequential patterns till that point, the procedure will output the set *FS*.

An Incremental Example Scenario. When an increment to database D occurs, it creates a larger database D' as shown in Table 1. Here, three new transactions have been added which is denoted as Δdb.

The *minw_sup* will get changed due to the changed value of *min_sup* and *avgW*. Taking 60% as the minimum support threshold as before, new absolute value of $min\_sup = 8 * 60\% = 5$ and the new avgW is calculated as 0.422. So, the new *minw_sup* value is now: $5 * 0.422 = 2.11$. The sequences in Δdb are scanned to check for occurrence of the patterns from *FS* and *SFS*, and the support count is updated in the trie. When the support count of patterns in the trie is updated, their weighted support are compared with the new *minw_sup* and *minw_sup* * μ to check if they become frequent or semi-frequent or even infrequent.

After the database update, the newly mined frequent sequences and semi-frequent sequences are listed in Table 3. Patterns not shown in the table are pruned out as infrequent. Although $<(ab)>$ was a semi-frequent pattern in D, it became frequent in D'. On the other hand, the frequent pattern $<(dc)>$ only appears once in Δdb, but it became semi-frequent now. Another pattern, $<bd>$, which was semi-frequent in D only increases one time in support in D'. So, $<bd>$ falls under the category of infrequent patterns.

Table 3. Weighted frequent and semi-frequent sequences in D'

Frequent sequences	Semi-frequent sequences
$<a>, <b>, <c>, <(ab)>$	$<(dc)>, <d>$

After taking the patterns from Δdb into account, the updated FS' and SFS' trie that emerges is illustrated in Fig. 2.

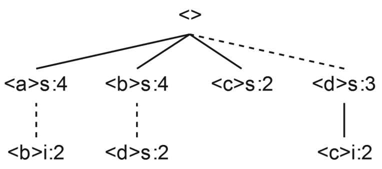

Fig. 1. The sequential pattern trie of *FS* and *SFS* in *D*

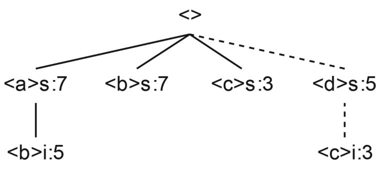

Fig. 2. The updated sequential pattern trie of FS' and SFS' in D'

3.1 The Pseudo-code

To get the weighted sequential patterns from the given databases which are dynamic in nature, we will use the proposed *WIncSpan* algorithm. A sequence database *D*, the minimum weighted support threshold *minw_sup* and the buffer ratio μ are given as input to the algorithm. Algorithm *WIncSpan* will generate the set of weighted frequent sequential patterns at any instance. The pseudo-code is given in Algorithm 1.

Algorithm 1. WIncSpan: Weighted Sequential Pattern Mining in Dynamic Database

Input: A sequence database D, the minimum weighted support threshold *minw_sup* and the buffer ratio μ

Output: The set of weighted frequent sequential patterns FS.

Method:

```
Begin
 1. Let $WSP$ be the set of Possible Weighted Frequent Sequential Patterns, $FS$ be the
    set of Frequent Patterns and $SFS$ be the set of Semi-Frequent Patterns.
    Now,
    $WSP \longleftarrow \{\}$, $FS \longleftarrow \{\}$, $SFS \longleftarrow \{\}$
 2. $WSP$ = Call the modified WSpan($WSP$, $D$, $minw\_sup$)
 3. for each pattern $P$ in $WSP$ do
 4.    if $sup(P) * weight(P) \geqslant minw\_sup$ then
 5.       insert (FS, $P$)
 6.    else if $sup(P) * weight(P) \geqslant minw\_sup * \mu$ then
 7.       insert (SFS, $P$)
 8.    end if
 9. end for
10. for each new increment $\Delta db$ in $D$ do
11.    $FS$, $SFS$ = Call WIncSpan($FS$, $SFS$, $\Delta db$, $minw\_sup$, $\mu$)
12.    output $FS$
13. end for
End
```

Procedure: WIncSpan(FS, SFS, Δdb, *minw_sup*, μ)

Parameters: FS: Frequent Sequences upto now; SFS: Semi-Frequent Sequences upto now; Δdb: incremented portion of D; *minw_sup*: minimum weighted support threshold; μ: buffer ratio.

```
 1. Let $FS'$ and $SFS'$ be the set of new frequent and semi-frequent patterns respec-
    tively.
 2. Initialize $FS' \longleftarrow \{\}$, $SFS' \longleftarrow \{\}$
 3. for each pattern $P$ in $FS$ or $SFS$ do
 4.    check $\Delta sup(P)$
 5.    $sup(P) = sup_D(P) + \Delta sup(P)$
 6.    if $sup(P) * weight(P) \geqslant minw\_sup$ then
 7.       insert($FS'$, P)
 8.    else if $sup(P) * weight(P) \geqslant minw\_sup * \mu$ then
 9.       insert ($SFS'$, P)
10.    end if
11. end for
12. return $FS'$, $SFS'$
```

In the algorithm, the main method creates the possible set of weighted sequential patterns by calling the modified WSpan as per above discussion. And from that set, the set of frequent sequential patterns *FS* and the set of semi-frequent sequential patterns *SFS* are created according to pre-defined conditions. Each increment is handled in this method. For each of the increment, Procedure: WIncSpan is called with necessary parameters. It creates the set of new frequent and semi-frequent sequential patterns FS' and SFS' respectively. Here, the support count of each pattern of *FS* and *SFS* is updated and then the pattern goes to either FS' or SFS' based on two conditions provided. It returns the complete FS' and SFS' which are to be used for further increments.

At any instance, we can check the *FS* list to get the weighted frequent sequential patterns till that point.

4 Performance Evaluation

In this section, we present the overall performance of our proposed algorithm *WIncSpan* over several datasets. The performance of our algorithm *WIncSpan* is compared with WSpan [15]. Various real-life datasets such as SIGN, BIBLE, Kosarak etc. were used in our experiment. These datasets were in spmf [6] format. Some datasets were collected directly from their site, some were collected from the site Frequent Itemset Mining Dataset repository [3] and then converted to spmf format. Both of the implementations of *WIncSpan* and WSpan were performed in Windows environment (Windows 10), on a core-i5 intel processor which operates at 3.2 GHz with 8 GB of memory.

Using real values of weights of items might be cumbersome in calculations. We used normalized values instead. To produce normalized weights, normal distribution is used with a suitable mean deviation and standard deviation. Thus the actual weights are adjusted to fit the common scale. In real life, items with high weights or costs appear less in number. So do the items with very low weights. On the other hand items with medium range of weights appear the most in number. To keep this realistic nature of items, we are using normal distribution for weight generation.

Here, we are providing the experimental results of the *WIncSpan* algorithm under various performance metrics. Except for the scalability test, for other performance metrics, we have taken an initial dataset to apply *WIncSpan* and WSpan, then we have added increments in the dataset in two consecutive phases. To calculate the overall performance of both of the algorithms, we measured their performances in three phases.

Performance Analysis w.r.t Runtime. We measured the runtime of *WIncSpan* and WSpan in three phases. The graphical representations of runtime with varying min_sup threshold for BMS2, BIBLE and Kosarak datasets are shown in Figs. 3, 4 and 5 respectively. Like sparse dataset as Kosarak, the runtime performance was also observed on dense dataset as SIGN. Figure 6 shows the graphical representation.

In the figures, we can see that the time required to run *WIncSpan* is less than the time required to run WSpan. And their differences in time becomes larger when the minimum support threshold is lowered. To understand how the

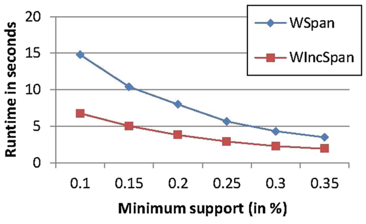

Fig. 3. Runtime for varying min_sup in BMS2 dataset.

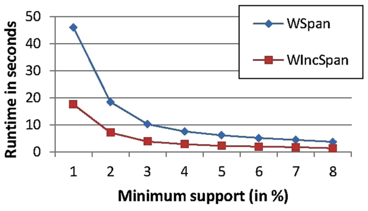

Fig. 4. Runtime for varying min_sup in BIBLE dataset.

runtime calculation is done more clearly, Table 4 shows the runtime in each phase for both *WIncSpan* and WSpan in Kosarak dataset. The total runtime is calculated which is used in the graph. It is clear that *WIncSpan* outperforms WSpan with respect to runtime. With the dynamic increment to the dataset, it is desirable that we generate patterns as fast as we can. *WIncSpan* fulfills this desire, and it runs a magnitude faster than WSpan.

Table 4. Runtime performance of WSpan and WIncSpan with varying min_sup in Kosarak dataset

min_sup (in %)	Runtime in initial database		Runtime after 1st increment		Runtime after 2nd increment		Runtime total (in seconds)	
	WSpan	WIncSpan	WSpan	WIncSpan	WSpan	WIncSpan	WSpan	WIncSpan
0.22%	81.3	81.3	66.8	1.66	65.5	1.53	213.6	84.49
0.26%	22.8	22.8	38.4	1.56	51.1	0.82	108.3	25.18
0.3%	14.5	14.5	20.3	0.75	26.8	0.63	61.6	15.88
0.34%	4.63	4.63	8.12	0.56	15.1	0.48	27.85	5.67
0.38%	2.11	2.11	3.11	0.48	5.11	0.54	10.33	3.13

Performance Analysis w.r.t Number of Patterns. The comparative performance analysis of *WIncSpan* and WSpan with respect to number of patterns for Kosarak and SIGN datasets are given in Figs. 7 and 8 respectively. In these graphs, we can see that the number of patterns generated by WSpan is more than the number of patterns generated by *WIncSpan*. As the minimum threshold is lowered, this difference gets bigger. However, the advantage of *WIncSpan* over WSpan is that it can generate these patterns way faster than WSpan as we saw in the previous section.

Performance Analysis with Varying Buffer Ratio. The lower the buffer ratio is, the higher the buffer size, which can accommodate more semi-frequent patterns. We have measured the number of patterns by varying the buffer ratio.

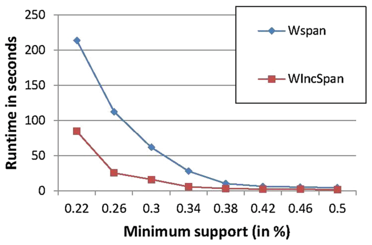

Fig. 5. Runtime for varying min_sup in Kosarak dataset.

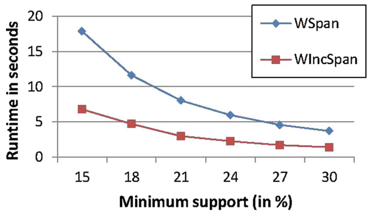

Fig. 6. Runtime for varying min_sup in SIGN dataset.

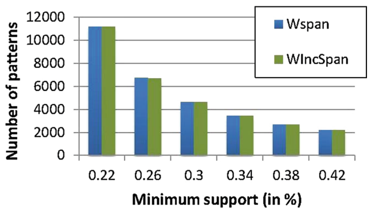

Fig. 7. Number of patterns for varying min_sup in Kosarak dataset.

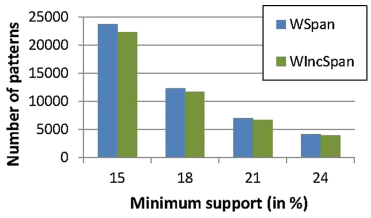

Fig. 8. Number of patterns for varying min_sup in SIGN dataset.

The graphical representation of the results in BIBLE dataset is shown in Fig. 9. Here, we can see that by increasing the buffer ratio the number of patterns tends to decrease. Because smaller number of semi-frequent patterns are generated and they can contribute less to frequent patterns in the next phase. We can also see that the number of patterns generated by WSpan is constant for several buffer ratio because WSpan does not buffer semi-frequent patterns, it generates patterns from scratch in every phase.

Performance Analysis w.r.t Memory. Figure 10 shows memory consumption by both *WIncSpan* and WSpan with varying min_sup in Kosarak dataset. For every dataset, it showed that memory consumed by *WIncSpan* is lower than memory consumed by WSpan. This is because *WIncSpan* scans the new appended part of the database and works on the dynamic trie. Whereas WSpan creates projected database for each pattern and generates new patterns from it. This requires a lot more memory compared with *WIncSpan*.

Performance Analysis with Varying Standard Deviation. To generate weights for items, we have used normal distribution with a fixed mean deviation of 0.5 and varying standard deviation (0.15 in most of the cases). For varying standard deviation, the number of items versus weight ranges curves are shown in Fig. 11. The range (mean deviation ± standard deviation) holds the most amount of items which is the characteristic of real-life items. In real life, items with medium values occur frequently whereas items with higher or too lower values occur infrequently.

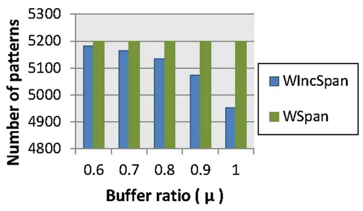

Fig. 9. Number of patterns for varying buffer ratio (μ) in BIBLE dataset.

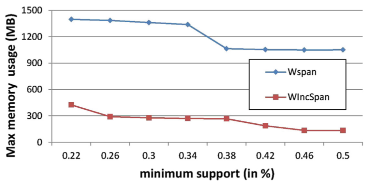

Fig. 10. Memory usage for varying min_sup in Kosarak dataset.

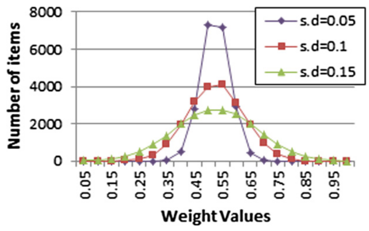

Fig. 11. Weight values by normal distribution with different standard deviations in Kosarak dataset.

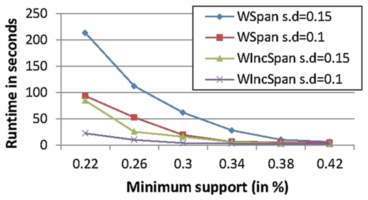

Fig. 12. Runtime evaluation with different standard deviations in Kosarak dataset.

Figures 12 and 13 show performance of *WIncSpan* and WSpan with respect to runtime and number of patterns respectively with different standard deviations. *WIncSpan* outperforms WSpan in case of runtime. The number of patterns in each case does not differ a lot from each other. So, it is clear that *WIncSpan* can work better than WSpan with varying weight ranges too.

Scalability Test. To test whether *WIncSpan* is scalable or not, we have run it on different datasets with several increments. Figure 14 shows the scalability performance analysis of *WIncSpan* and WSpan in Kosarak dataset when the minimum support threshold is 0.3%. After running on an initial set of the database, five consecutive increments were added and the runtime performance was measured in each step.

Here, we can see that both WSpan and *WIncSpan* take same amount of time in initial set of database. As the database grows dynamically, WSpan takes more time than *WIncSpan*. *WIncSpan* tends to consume less time from second increment as it uses the dynamic trie and new appended part of the database only. From second increment to the last increment, consumed time by *WIncSpan* does not vary that much from each other. So, we can see that *WIncSpan* is scalable along with its runtime and memory efficiency.

The above discussion implies that *WIncSpan* can be applied in real life applications where the database tends to grow dynamically and the values (weights) of the items are important. *WIncSpan* outperforms WSpan in all the cases. In case of number of patterns, *WIncSpan* may provide less amount of patterns than

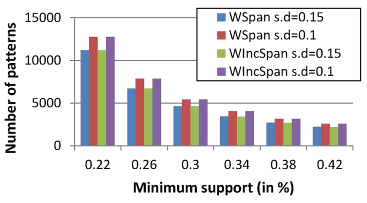

Fig. 13. Number of patterns for varying standard deviations in Kosarak dataset.

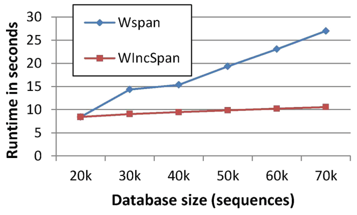

Fig. 14. Scalability test in Kosarak dataset (min_sup = 0.3%)

WSpan, but this behaviour can be acceptable considering the remarkable less amount of time it consumes. In real life, items with lower and higher values are not equally important. So, *WIncSpan* can be applied in place of IncSpan also where the value of the item is important.

5 Conclusions

A new algorithm *WIncSpan*, for mining weighted sequential patterns in large incremental databases, is proposed in this study. It overcomes the limitations of previously existing algorithms. By buffering semi-frequent sequences and maintaining dynamic trie, our approach works efficiently in mining when the database grows dynamically. The actual benefits of the proposed approach is found in its experimental results, where the *WIncSpan* algorithm has been found to outperform WSpan. It is found to be more time and memory efficient.

This work will be highly applicable in mining weighted sequential patterns in databases where constantly new updates are available, and where the individual items can be attributed with weight values to distinguish between them. Areas of application therefore include mining Market Transactions, Weather Forecast, improving Health-Care and Health Insurance and many others. It can also be used in Fraud Detection by assigning high weight values to previously found fraud patterns.

The work presented here can be extended to include more research problems to be solved for efficient solutions. Incremental mining can be done on closed sequential patterns with weights. It can also be extended for mining sliding window based weighted sequential patterns over datastreams [1].

References

1. Ahmed, C.F., Tanbeer, S.K., Jeong, B.S., Lee, Y.K.: An efficient algorithm for sliding window-based weighted frequent pattern mining over data streams. IEICE Trans. Inf. Syst. **92**(7), 1369–1381 (2009)
2. Ayres, J., Flannick, J., Gehrke, J., Yiu, T.: Sequential pattern mining using a bitmap representation. In: Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD 2002, pp. 429–435. ACM, New York (2002)

3. Bart, G., Mohammed, J.Z.: Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations, Fimi 2004, Brighton, UK, 1 November 2004. FIMI (2005). http://fimi.ua.ac.be/
4. Cheng, H., Yan, X., Han, J.: IncSpan: incremental mining of sequential patterns in large database. In: Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 527–532. ACM (2004)
5. Cui, W., An, H.: Discovering interesting sequential pattern in large sequence database. In: Asia-Pacific Conference on Computational Intelligence and Industrial Applications, PACIIA 2009, vol. 2, pp. 270–273. IEEE (2009)
6. Fournier-Viger, P., et al.: The SPMF open-source data mining library version 2. In: Berendt, B., et al. (eds.) ECML PKDD 2016. LNCS, vol. 9853, pp. 36–40. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46131-1_8
7. Han, J., Pei, J., Mortazavi-Asl, B., Chen, Q., Dayal, U., Hsu, M.C.: FreeSpan: frequent pattern-projected sequential pattern mining. In: Proceedings of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 355–359. ACM (2000)
8. Kollu, A., Kotakonda, V.K.: Incremental mining of sequential patterns using weights. IOSR J. Comput. Eng. **5**, 70–73 (2013)
9. Lin, J.C.W., Hong, T.P., Gan, W., Chen, H.Y., Li, S.T.: Incrementally updating the discovered sequential patterns based on pre-large concept. Intell. Data Anal. **19**(5), 1071–1089 (2015)
10. Masseglia, F., Poncelet, P., Teisseire, M.: Incremental mining of sequential patterns in large databases. Data Knowl. Eng. **46**(1), 97–121 (2003)
11. Nguyen, S.N., Sun, X., Orlowska, M.E.: Improvements of IncSpan: incremental mining of sequential patterns in large database. In: Ho, T.B., Cheung, D., Liu, H. (eds.) PAKDD 2005. LNCS, vol. 3518, pp. 442–451. Springer, Heidelberg (2005). https://doi.org/10.1007/11430919_52
12. Pei, J., Han, J., Mortazavi-Asl, B., Wang, J., Pinto, H., Chen, Q., Dayal, U., Hsu, M.C.: Mining sequential patterns by pattern-growth: the PrefixSpan approach. IEEE Trans. Knowl. Data Eng. **16**(11), 1424–1440 (2004)
13. Srikant, R., Agrawal, R.: Mining sequential patterns: generalizations and performance improvements. In: Apers, P., Bouzeghoub, M., Gardarin, G. (eds.) EDBT 1996. LNCS, vol. 1057, pp. 3–17. Springer, Heidelberg (1996). https://doi.org/10.1007/BFb0014140
14. Yun, U.: Efficient mining of weighted interesting patterns with a strong weight and/or support affinity. Inf. Sci. **177**(17), 3477–3499 (2007)
15. Yun, U.: A new framework for detecting weighted sequential patterns in large sequence databases. Knowl.-Based Syst. **21**(2), 110–122 (2008)
16. Zaki, M.J.: SPADE: an efficient algorithm for mining frequent sequences. Mach. Learn. **42**, 31–60 (2001)