IncSpan: Incremental Mining of Sequential Patterns in Large Database *

Hong Cheng
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801
hcheng3@uiuc.edu

Xifeng Yan
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801
xyan@uiuc.edu

Jiawei Han
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801
hanj@cs.uiuc.edu

ABSTRACT

Many real life sequence databases grow incrementally. It is undesirable to mine sequential patterns from scratch each time when a small set of sequences grow, or when some new sequences are added into the database. Incremental algorithm should be developed for sequential pattern mining so that mining can be adapted to incremental database updates. However, it is nontrivial to mine sequential patterns incrementally, especially when the existing sequences grow incrementally because such growth may lead to the generation of many new patterns due to the interactions of the growing subsequences with the original ones. In this study, we develop an efficient algorithm, IncSpan, for incremental mining of sequential patterns, by exploring some interesting properties. Our performance study shows that IncSpan outperforms some previously proposed incremental algorithms as well as a non-incremental one with a wide margin.

Categories and Subject Descriptors

H.2.8 [**Database Management**]: Database Applications—*data mining*

General Terms

Algorithms, Performance, Experimentation

Keywords

incremental mining, buffering pattern, reverse pattern matching, shared projection

*The work was supported in part by U.S. National Science Foundation NSF IIS-02-09199, University of Illinois, and Microsoft Research. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

KDD'04, August 22–25, 2004, Seattle, Washington, USA.

1. INTRODUCTION

Sequential pattern mining is an important and active research topic in data mining [1, 5, 4, 8, 13, 2], with broad applications, such as customer shopping transaction analysis, mining web logs, mining DNA sequences, etc.

There have been quite a few sequential pattern or closed sequential pattern mining algorithms proposed in the previous work, such as [10, 8, 13, 2, 12, 11], that mine frequent subsequences from a large sequence database efficiently. These algorithms work in a *one-time fashion*: mine the entire database and obtain the set of results. However, in many applications, databases are updated incrementally. For example, customer shopping transaction database is growing daily due to the appending of newly purchased items for existing customers for their subsequent purchases and/or insertion of new shopping sequences for new customers. Other examples include Weather sequences and patient treatment sequences which grow incrementally with time. The existing sequential mining algorithms are not suitable for handling this situation because the result mined from the old database is no longer valid on the updated database, and it is intolerably inefficient to mine the updated databases from scratch.

There are two kinds of database updates in applications: (1) inserting new sequences (denoted as INSERT), and (2) appending new itemsets/items to the existing sequences (denoted as APPEND). A real application may contain both.

It is easier to handle the first case: INSERT. An important property of INSERT is that *a frequent sequence in* $DB' = DB \cup \Delta db$ *must be frequent in either* DB *or* Δdb *(or both).* If a sequence is infrequent in both DB and Δdb, it cannot be frequent in DB', as shown in Figure 1. This property is similar to that of frequent patterns, which has been used in incremental frequent pattern mining [3, 9, 14]. Such incremental frequent pattern mining algorithms can be easily extended to handle sequential pattern mining in the case of INSERT.

It is far trickier to handle the second case, APPEND, than the first one. This is because not only the appended items may generate new locally frequent sequences in Δdb, but also that locally infrequent sequences may contribute their occurrence count to the same infrequent sequences in the original database to produce frequent ones. For example, in the appended database in Figure 1, suppose $|DB|$=1000 and $|\Delta db|$=20, min_sup=10%. Suppose a sequence s is in-

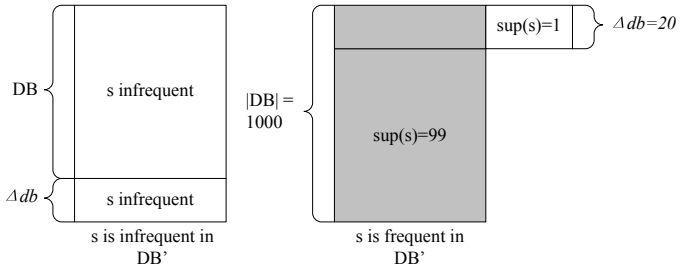


Figure 1: Examples in INSERT and APPEND database

frequent in DB with 99 occurrences ($sup = 9.9\%$). In addition, it is also infrequent in Δdb with only 1 occurrence ($sup = 5\%$). Although s is infrequent in both DB and Δdb, it becomes frequent in DB' with 100 occurrences. This problem complicates the incremental mining since one cannot ignore the infrequent sequences in Δdb, but there are an exponential number of infrequent sequences even in a small Δdb and checking them against the set of infrequent sequences in DB will be very costly.

When the database is updated with a combination of INSERT and APPEND, we can treat INSERT as a special case of APPEND – treating the inserted sequences as appended transactions to an empty sequence in the original database. Then this problem is reduced to APPEND. Therefore, we focus on the APPEND case in the following discussion.

In this paper, an efficient algorithm, called IncSpan, is developed, for incremental mining over multiple database increments. Several novel ideas are introduced in the algorithm development: (1) maintaining a set of "*almost frequent*" sequences as the candidates in the updated database, which has several nice properties and leads to efficient techniques, and (2) two optimization techniques, *reverse pattern matching* and *shared projection*, are designed to improve the performance. Reverse pattern matching is used for matching a sequential pattern in a sequence and prune some search space. Shared projection is designed to reduce the number of database projections for some sequences which share a common prefix. Our performance study shows that IncSpan is efficient and scalable.

The remaining of the paper is organized as follows. Section 2 introduces the basic concepts related to incremental sequential pattern mining. Section 3 presents the idea of buffering patterns, several properties of this technique and the associated method. Section 4 formulates the IncSpan algorithm with two optimization techniques. We report and analyze performance study in Section 5, introduce related work in Section 6. We conclude our study in Section 7.

2. PRELIMINARY CONCEPTS

Let $I = \{i_1, i_2, \ldots, i_k\}$ be a set of all items. A subset of I is called an *itemset*. A *sequence* $s = \langle t_1, t_2, \ldots, t_m \rangle$ ($t_i \subseteq I$) is an ordered list. The *size*, $|s|$, of a sequence is the number of itemsets in the sequence. The *length*, $l(s)$, is the total number of items in the sequence, i.e., $l(s) = \sum_{i=1}^{n} |t_i|$. A sequence $\alpha = \langle a_1, a_2, \ldots, a_m \rangle$ is a *sub-sequence* of another sequence $\beta = \langle b_1, b_2, \ldots, b_n \rangle$, denoted as $\alpha \sqsubseteq \beta$ (if $\alpha \neq \beta$, written as $\alpha \sqsubset \beta$), if and only if $\exists i_1, i_2, \ldots, i_m$, such that $1 \leqslant i_1 < i_2 < \ldots < i_m \leqslant n$ and $a_1 \subseteq b_{i_1}, a_2 \subseteq b_{i_2}, \ldots,$ *and* $a_m \subseteq b_{i_m}$.

A sequence database, $D = \{s_1, s_2, \ldots, s_n\}$, is a set of sequences. The *support* of a sequence α in D is the number of sequences in D which contain α, $support(\alpha) = |\{s | s \in D \text{ and } \alpha \sqsubseteq s\}|$. Given a minimum support threshold, *min_sup*, a sequence is **frequent** if its support is no less than *min_sup*; given a factor $\mu \leq 1$, a sequence is **semi-frequent** if its support is less than *min_sup* but no less than $\mu * min\_sup$; a sequence is **infrequent** if its support is less than $\mu * min\_sup$. The set of **frequent sequential pattern**, FS, includes all the frequent sequences; and the set of **semi-frequent sequential pattern** SFS, includes all the semi-frequent sequences.

EXAMPLE 1. The second column of Table 1 is a sample sequence database D. If $min\_sup = 3$, $FS = \{\langle (a) \rangle : 4, \langle (b) \rangle : 3, \langle (d) \rangle : 4, \langle (b)(d) \rangle : 3\}$. ∎

Seq ID.	Original Part	Appended Part
0	$(a)(h)$	(**c**)
1	(eg)	(**a**)(**bce**)
2	$(a)(b)(d)$	(**ck**)(**l**)
3	$(b)(df)(a)(b)$	ϕ
4	$(a)(d)$	ϕ
5	$(be)(d)$	ϕ

Table 1: A Sample Sequence Database D and the Appended part

Given a sequence $s = \langle t_1, \ldots, t_m \rangle$ and another sequence $s_a = \langle t'_1, \ldots, t'_n \rangle$, $s' = s \diamond s_a$ means s concatenates with s_a. s' is called an **appended sequence** of s, denoted as $s' \sim_a s$. If s_a is empty, $s' = s$, denoted as $s' \cong_a s$. An **appended sequence database** D' of a sequence database D is one that (1) $\forall s'_i \in D'$, $\exists s_j \in D$ such that $s'_i \sim_a s_j$ or $s'_i \cong_a s_j$, and (2) $\forall s_i \in D$, $\exists s'_j \in D'$ such that $s'_j \sim_a s_i$ or $s'_j \cong_a s_i$. We denote $LDB = \{s'_i | s'_i \in D' \text{ and } s'_i \sim_a s_j\}$, i.e., LDB is the set of sequences in D' which are appended with items/itemsets. We denote $ODB = \{s_i | s_i \in D \text{ and } s_i \sim_a s'_j\}$, i.e., ODB is the set of sequences in D which are appended with items/itemsets in D'. We denote the set of frequent sequences in D' as FS'.

EXAMPLE 2. The third column of Table 1 is the appended part of the original database. If $min\_sup = 3$, $FS' = \{\langle (a) \rangle : 5, \langle (b) \rangle : 4, \langle (d) \rangle : 4, \langle (b)(d) \rangle : 3, \langle (c) \rangle : 3, \langle (a)(b) \rangle : 3, \langle (a)(c) \rangle : 3\}$. ∎

A **sequential pattern tree T** is a tree that represents the set of frequent subsequences in a database. Each node p in T has a tag labelled with s or i. s means the node is a starting item in an itemset; i means the node is an intermediate item in an itemset. Each node p has a support value which represents the support of the subsequence starting from the root of T and ending at the node p.

Problem Statement. Given a sequence database D, a *min_sup* threshold, the set of frequent subsequences FS in D, and an appended sequence database D' of D, the problem of **incremental sequential pattern mining** is to mine the set of frequent subsequences FS' in D' based on FS instead of mining on D' from scratch.

3. BUFFER SEMI-FREQUENT PATTERNS

In this section, we present the idea of buffering semi-frequent patterns, study its properties, and design solutions of how to incrementally mine and maintain FS and SFS.

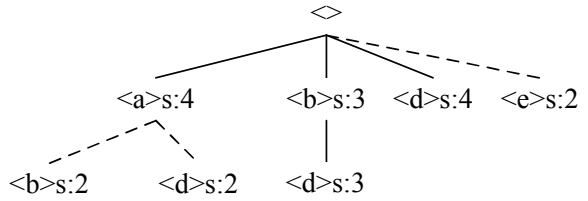


Figure 2: The Sequential Pattern Tree of FS and SFS in D

3.1 Buffering Semi-frequent Patterns

We buffer semi-frequent patterns, which can be considered as a statistics-based approach. The technique is to lower the *min_sup* by a *buffer ratio* $\mu \leq 1$ and keep a set SFS in the original database D. This is because since the sequences in SFS are "*almost frequent*", most of the frequent subsequences in the appended database will either come from SFS or they are already frequent in the original database. With a minor update to the original database, it is expected that only a small fraction of subsequences which were infrequent previously would become frequent. This is based on the assumption that updates to the original database have a uniform probability distribution on items. It is expected that most of the frequent subsequences introduced by the updated part of the database would come from the SFS. The SFS forms a kind of boundary (or "*buffer zone*") between the frequent subsequences and infrequent subsequences.

EXAMPLE 3. Given a database D in Example 1, *min_sup* = 3, $\mu = 0.6$. The sequential pattern tree T representing FS and SFS in D is shown in Figure 2. FS are shown in solid line and SFS in dashed line.

When the database D is updated to D', we have to check LDB to update support of every sequence in FS and SFS. There are several possibilities:

1. A pattern which is frequent in D is still frequent in D';
2. A pattern which is semi-frequent in D becomes frequent in D';
3. A pattern which is semi-frequent in D is still semi-frequent in D';
4. Appended database Δdb brings new items.
5. A pattern which is infrequent in D becomes frequent in D';
6. A pattern which is infrequent in D becomes semi-frequent in D';

Case (1)–(3) are trivial cases since we already keep the information. We will consider case (4)–(6) now.

Case (4): Appended database Δdb brings new items. For example, in the database D', $\langle(c)\rangle$ is a new item brought by Δdb. It does not appear in D.

Property: An item which does not appear in D and is brought by Δdb has no information in FS or SFS.

Solution: Scan the database LDB for single items.

For a new item or an originally infrequent item in D, if it becomes frequent or semi-frequent, insert it into FS or SFS. Then use the new frequent item as prefix to construct projected database and discover frequent and semi-frequent sequences recursively. For a frequent or semi-frequent item in D, update its support.

Case (5): A pattern which is infrequent in D becomes frequent in D'. For example, in the database D', $\langle(a)(c)\rangle$ is an example of case (5). It is infrequent in D and becomes frequent in D'. We do not keep $\langle(a)(c)\rangle$ in FS or SFS, but we have the information of its prefix $\langle(a)\rangle$.

Property: If an infrequent sequence p' in D becomes frequent in D', all of its prefix subsequences must also be frequent in D'. Then at least one of its prefix subsequences p is in FS.

Solution: Start from its frequent prefix p in FS and construct p-projected database, we will discover p'.

Formally stated, given a frequent pattern p in D', we want to discover whether there is any pattern p' with p as prefix where p' was infrequent in D but is frequent in D'. A sequence p' which changes from infrequent to frequent must have $\Delta sup(p') > (1-\mu)min\_sup$.

We claim if a frequent pattern p has support in LDB $sup_{LDB}(p) \geq (1-\mu)min\_sup$, it is possible that some subsequences with p as prefix will change from infrequent to frequent. If $sup_{LDB}(p) < (1-\mu)min\_sup$, we can safely prune search with prefix p.

THEOREM 1. *For a frequent pattern p, if its support in LDB $sup_{LDB}(p) < (1-\mu)min\_sup$, then there is no sequence p' having p as prefix changing from infrequent in D to frequent in D'.*

Proof *: p' was infrequent in D, so*

$$sup_D(p') < \mu * min\_sup \quad (1)$$

If $sup_{LDB}(p) < (1-\mu)min\_sup$, then

$$sup_{LDB}(p') \leq sup_{LDB}(p) < (1-\mu)min\_sup$$

Since $sup_{LDB}(p') = sup_{ODB}(p') + \Delta sup(p')$. Then we have

$$\Delta sup_{LDB}(p') \leq sup_{LDB}(p') < (1-\mu)min\_sup. \quad (2)$$

Since $sup_{D'}(p') = sup_D(p') + \Delta sup(p')$, combining (1) and (2), we have $sup_{D'}(p') < min\_sup$. So p' cannot be frequent in D'. ∎

Therefore, if a pattern p has support in LDB $sup_{LDB}(p) < (1-\mu)min\_sup$, we can prune search with prefix p. Otherwise, if $sup_{LDB}(p) \geq (1-\mu)min\_sup$, it is possible that some sequences with p as prefix will change from infrequent to frequent. In this case, we have to project the whole database D' using p as prefix. If $|LDB|$ is small or μ is small, there are very few patterns that have $sup_{LDB}(p) \geq (1-\mu)min\_sup$, making the number of projections small.

In our example, $sup_{LDB}(a) = 3 > (1-0.6)*3$, we have to do the projection with $\langle(a)\rangle$ as prefix. And we discover "$\langle(a)(c)\rangle : 3$" which was infrequent in D. For another example, $sup_{LDB}(d) = 1 < (1-0.6)*3$, there is no sequence with d as prefix which changes from infrequent to frequent, so we can prune the search on it.

Theorem 1 provides an effective bound to decide whether it is necessary to project a database. It is essential to guarantee the result be complete.

We can see from the projection condition, $sup_{LDB}(p) \geq (1-\mu)min\_sup$, the smaller μ is, the larger buffer we keep, the fewer database projections the algorithm needs. The choice of μ is heuristic. If μ is too high, then the buffer is small and we have to do a lot of database projections to discover sequences outside of the buffer. If μ is set very low, we will keep many subsequences in the buffer. But mining the buffering patterns using $\mu * min\_sup$ would be much more inefficient than with *min_sup*. We will show this

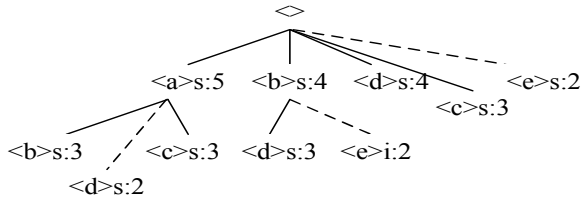


Figure 3: The Sequential Pattern Tree of FS and SFS in D'

tradeoff through experiments in Section 5.

Case (6): A pattern which is infrequent in D becomes semi-frequent in D'. For example, in the database D', $\langle(be)\rangle$ is an example of case (6). It is infrequent in D and becomes semi-frequent in D'.

Property: If an infrequent sequence p' becomes semi-frequent in D', all of its prefix subsequences must be either frequent or semi-frequent. Then at least one of its prefix subsequences, p, is in FS or SFS.

Solution: Start from its prefix p in FS or SFS and construct p-projected database, we will discover p'.

Formally stated, given a pattern p, we want to discover whether there is any pattern p' with p as prefix where p' was infrequent but is semi-frequent in D'.

If the prefix p is in FS or SFS, construct p-projected database and we will discover p' in p-projected database. Therefore, for any pattern p' from infrequent to semi-frequent, if its prefix is in FS or SFS, p' can be discovered.

In our example, for the frequent pattern $\langle(b)\rangle$, we do the projection on $\langle(b)\rangle$ and get a semi-frequent pattern $\langle(be)\rangle : 2$ which was infrequent in D.

We show in Figure 3 the sequential pattern tree T including FS and SFS after the database updates to D'. We can compare it with Figure 2 to see how the database update affects FS and SFS.

4. INCSPAN: DESIGN AND IMPLEMENTATION

In this section, we formulate the IncSpan algorithm which exploits the technique of buffering semi-frequent patterns. We first present the algorithm outline and then introduce two optimization techniques.

4.1 IncSpan: Algorithm Outline

Given an original database D, an appended database D', a threshold *min_sup*, a buffer ratio μ, a set of frequent sequences FS and a set of semi-frequent sequences SFS, we want to discover the set of frequent sequences FS' in D'.

Step 1: Scan LDB for single items, as shown in case (4).

Step 2: Check every pattern in FS and SFS in LDB to adjust the support of those patterns.

Step 2.1: If a pattern becomes frequent, add it to FS'. Then check whether it meets the projection condition. If so, use it as prefix to project database, as shown in case (5).

Step 2.2: If a pattern is semi-frequent, add it to SFS'.

The algorithm is given in Figure 4.

4.2 Reverse Pattern Matching

Reverse pattern matching is a novel optimization technique. It matches a sequential pattern against a sequence from the end towards the front. This is used to check support increase of a sequential pattern in LDB. Since the appended items are always at the end part of the original sequence, reverse pattern matching would be more efficient than projection from the front.

Algorithm. IncSpan(D', *min_sup*, μ, FS, SFS)
Input: An appended database D', *min_sup*, μ, frequent sequences FS in D, semi-frequent sequences SFS in D.
Output: FS' and SFS'.

```
1: FS' = φ, SFS' = φ
2: Scan LDB for single items;
3: Add new frequent item into FS';
4: Add new semi-frequent item into SFS';
5: for each new item i in FS' do
6:       PrefixSpan(i, D'|i, μ * min_sup, FS', SFS');
7: for every pattern p in FS or SFS do
8:     check Δsup(p);
9:     if sup(p) = sup_D(p) + Δsup(p) ≥ min_sup
10:        insert(FS', p);
11:        if sup_LDB(p) ≥ (1 − μ)min_sup
12:          PrefixSpan(p, D'|p, μ * min_sup, FS', SFS');
13:    else
14:        insert(SFS', p);
15: return;
```

Figure 4: IncSpan algorithm

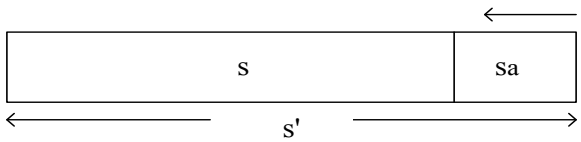


Figure 5: Reverse Pattern Matching

Given an original sequence s, an appended sequence $s' = s \diamond s_a$, and a sequential pattern p, we want to check whether the support of p will be increased by appending s_a to s. There are two possibilities:

1. If the last item of p is not supported by s_a, whether p is supported by s or not, $sup(p)$ is not increased when s grows to s'. Therefore, as long as we do not find the last item of p in s_a, we can prune searching.
2. If the last item of p is supported by s_a, we have to check whether s' supports p. We check this by continuing in the reverse direction. If p is not supported by s', we can prune searching and keep $sup(p)$ unchanged. Otherwise we have to check whether s supports p. If s supports p, keep $sup(p)$ unchanged; otherwise, increase $sup(p)$ by 1.

Figure 5 shows the reverse pattern matching.

4.3 Shared Projection

Shared Projection is another optimization technique we exploit. Suppose we have two sequences $\langle(a)(b)(c)(d)\rangle$ and $\langle(a)(b)(c)(e)\rangle$, and we need to project database using each as prefix. If we make two database projections individually, we do not take advantage of the similarity between the two subsequences. Actually the two projected databases up to subsequence $\langle(a)(b)(c)\rangle$, i.e., $D'|\langle(a)(b)(c)\rangle$ are the same.

From $D'|\langle(a)(b)(c)\rangle$, we do one more step projection for item d and e respectively. Then we can share the projection for $\langle(a)(b)(c)\rangle$.

To use shared projection, when we detect some subsequence that needs projecting database, we do not do the projection immediately. Instead we label it. After finishing checking and labelling all the sequences, we do the projection by traversing the sequential pattern tree. Tree is natural for this task because the same subsequences are represented using shared branches.

5. PERFORMANCE STUDY

A comprehensive performance study has been conducted in our experiments. We use a synthetic data generator provided by IBM. The synthetic dataset generator can be retrieved from an IBM website, http://www.almaden.ibm.com /cs/quest. The details about parameter settings can be referred in [1].

All experiments are done on a PowerEdge 6600 server with Xeon 2.8 , 4G memory. The algorithms are written in C++ and compiled using g++ with -O3 optimization. We compare three algorithms: IncSpan, an incremental mining algorithm ISM [7], and a non-incremental algorithm PrefixSpan[8].

Figure 6 (a) shows the running time of three algorithms when *min_sup* changes on the dataset D10C10T2.5N10, 0.5% of which has been appended with transactions. IncSpan is the fastest, outperforming PrefixSpan by a factor of 5 or more, and outperforming ISM even more. ISM even cannot finish within a time limit when the support is low.

Figure 6 (b) shows how the three algorithms can be affected when we vary the percentage of sequences in the database that have been updated. The dataset we use is D10C10T2.5N10, *min_sup*=1%. The buffer ratio $\mu = 0.8$. The curves show that the time increases as the incremental portion of the database increases. When the incremental part exceeds 5% of the database, PrefixSpan outperforms IncSpan. This is because if the incremental part is not very small, the number of patterns brought by it increases, making a lot overhead for IncSpan to handle. In this case, mining from scratch is better. But IncSpan still outperforms ISM by a wide margin no matter what the parameter is.

Figure 6 (c) shows the memory usage of IncSpan and ISM. The database is D10C10T2.5N10, *min_sup* varies from 0.4% to 1.5%, buffer ratio $\mu = 0.8$. Memory usage of IncSpan increases linearly as *min_sup* decreases while memory used by ISM increases dramatically. This is because the number of sequences in negative border increases sharply as *min_sup* decreases. This figure verifies that negative border is a memory-consuming approach.

Figure 7 (a) shows how the IncSpan algorithm can be affected by varying buffer ratio μ. Dataset is D10C10T2.5N10, 5% of which is appended with new transactions. We use PrefixSpan as a baseline. As we have discussed before, if we set μ very high, we will have fewer pattern in SFS, then the support update for sequences in SFS on LDB will be more efficient. However, since we keep less information in SFS, we may need to spend more time on projecting databases. In the extreme case $\mu = 1$, SFS becomes empty. On the other hand, if we set the μ very low, we will have a large number of sequences in SFS, which makes the support update stage very slow. Experiment shows, when $\mu = 0.8$, it achieves the best performance.

Figure 7 (b) shows the performance of IncSpan to handle multiple (5 updates in this case) database updates. Each time the database is updated, we run PrefixSpan to mine from scratch. We can see from the figure, as the increments accumulate, the time for incremental mining increases, but increase is very small and the incremental mining still outperforms mining from scratch by a factor of 4 or 5. This experiment shows that IncSpan can really handle multiple database updates without significant performance degrading.

Figure 7 (c) shows the scalability of the three algorithms by varying the size of database. The number of sequences in databases vary from 10,000 to 100,000. 5% of each database is updated. *min_sup*=0.8%. It shows that all three algorithms scale well with the database size.

6. RELATED WORK

In sequential pattern mining, efficient algorithms like GSP [10], SPADE [13], PrefixSpan [8], and SPAM [2] were developed.

Partition [9] and FUP [3] are two algorithms which promote partitioning the database, mining local frequent itemsets, and then consolidating the global frequent itemsets by cross check. This is based on that a frequent itemset must be frequent in at least one local database. If a database is updated with INSERT, we can use this idea to do the incremental mining. Zhang et al. [14] developed two algorithms for incremental mining sequential patterns when sequences are inserted into or deleted from the original database.

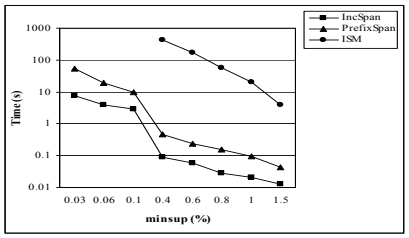
Parthasarathy et al. [7] developed an incremental mining algorithm ISM by maintaining a sequence lattice of an old database. The sequence lattice includes all the frequent sequences and all the sequences in the negative border. However, there are some disadvantages for using negative border: (1) The combined number of sequences in the frequent set and the negative border is huge; (2) The negative border is generated based on the structural relation between sequences. However, these sequences do not necessarily have high support. Therefore, using negative border is very time and memory consuming.

Masseglia et al. [6] developed another incremental mining algorithm ISE using candidate generate-and-test approach. The problem of this algorithm is (1) the candidate set can be very huge, which makes the test-phase very slow; and (2) its level-wise working manner requires multiple scans of the whole database. This is very costly, especially when the sequences are long.

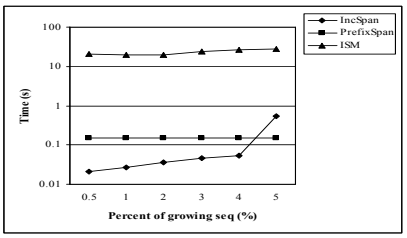
7. CONCLUSIONS

In this paper, we investigated the issues for incremental mining of sequential patterns in large databases and addressed the inefficiency problem of mining the appended database from scratch. We proposed an algorithm IncSpan by exploring several novel techniques to balance efficiency and reusability. IncSpan outperforms the non-incremental method (using PrefixSpan) and a previously proposed incremental mining algorithm ISM by a wide margin. It is a promising algorithm to solve practical problems with many real applications.

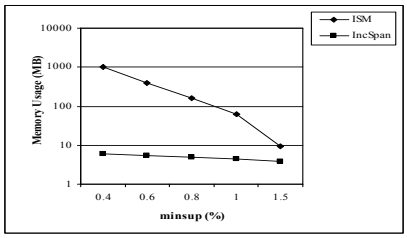
There are many interesting research problems related to IncSpan that should be pursued further. For example, incremental mining of closed sequential patterns, structured



(a) varying *min_sup*

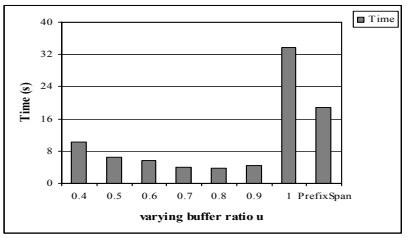


(b) varying percentage of updated sequences

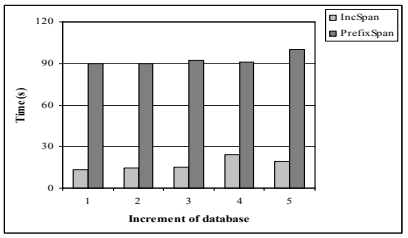


(c) Memory Usage under varied *min_sup*

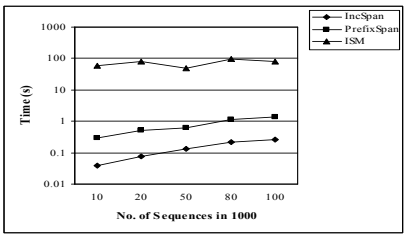
Figure 6: Performance study



(a) varying buffer ratio μ



(b) multiple increments of database



(c) varying # of sequences (in 1000) in DB

Figure 7: Performance study

patterns in databases and/or data streams are interesting problems for future research.

8. REFERENCES

[1] R. Agrawal and R. Srikant. Mining sequential patterns. In *Proc. 1995 Int. Conf. Data Engineering (ICDE'95)*, pages 3–14, March 1995.

[2] J. Ayres, J. E. Gehrke, T. Yiu, and J. Flannick. Sequential pattern mining using bitmaps. In *Proc. 2002 ACM SIGKDD Int. Conf. Knowledge Discovery in Databases (KDD'02)*, July 2002.

[3] D. Cheung, J. Han, V. Ng, and C. Wong. Maintenance of discovered association rules in large databases: An incremental update technique. In *Proc. of the 12th Int. Conf. on Data Engineering (ICDE'96)*, March 1996.

[4] M. Garofalakis, R. Rastogi, and K. Shim. SPIRIT: Sequential pattern mining with regular expression constraints. In *Proc. 1999 Int. Conf. Very Large Data Bases (VLDB'99)*, pages 223–234, Sept 1999.

[5] H. Mannila, H. Toivonen, and A. I. Verkamo. Discovering frequent episodes in sequences. In *Proc. 1995 Int. Conf. Knowledge Discovery and Data Mining (KDD'95)*, pages 210–215, Aug 1995.

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

[7] S. Parthasarathy, M. Zaki, M. Ogihara, and S. Dwarkadas. Incremental and interactive sequence mining. In *Proc. of the 8th Int. Conf. on Information and Knowledge Management (CIKM'99)*, Nov 1999.

[8] J. Pei, J. Han, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, and M.-C. Hsu. PrefixSpan: Mining sequential patterns efficiently by prefix-projected pattern growth. In *Proc. 2001 Int. Conf. Data Engineering (ICDE'01)*, pages 215–224, April 2001.

[9] A. Savasere, E. Omiecinski, and S. Navathe. An efficient algorithm for mining association rules in large databases. In *Proc. 1995 Int. Conf. Very Large Data Bases (VLDB'95)*, Sept 1995.

[10] R. Srikant and R. Agrawal. Mining sequential patterns: Generalizations and performance improvements. In *Proc. of the 5th Int. Conf. on Extending Database Technology (EDBT'96)*, Mar 1996.

[11] J. Wang and J. Han. Bide: Efficient mining of frequent closed sequences. In *Proc. of 2004 Int. Conf. on Data Engineering (ICDE'04)*, March 2004.

[12] X. Yan, J. Han, and R. Afshar. CloSpan: Mining closed sequential patterns in large datasets. In *Proc. 2003 SIAM Int.Conf. on Data Mining (SDM'03)*, May 2003.

[13] M. Zaki. SPADE: An efficient algorithm for mining frequent sequences. *Machine Learning*, 40:31–60, 2001.

[14] M. Zhang, B. Kao, D. Cheung, and C. Yip. Efficient algorithms for incremental updates of frequent sequences. In *Proc. of Pacific-Asia Conf. on Knowledge Discovery and Data Mining (PAKDD'02)*, May 2002.