Incrementally Mining Temporal Patterns in Interval-based Databases

Yi-Cheng Chen[1], Julia Tzu-Ya Weng[2,3], Jun-Zhe Wang[4], Chien-Li Chou[4], Jiun-Long Huang[4], and Suh-Yin Lee[4]
[1]*Department of Computer Science & information Engineering, Tamkang University, New Taipei City, Taiwan*
[2]*Department of Computer Science & Engineering, Yuan Ze University, Taoyuan, Taiwan*
[3]*Innovation Center for Big Data and Digital Convergence, Yuan Ze University, Taoyuan, Taiwan*
[4]*Department of Computer Science, National Chiao Tung University, Hsinchu, Taiwan*
ycchen@mail.tku.edu.tw julweng@saturn.yzu.edu.tw {jzwang, fallwind, jlhuang, sylee}@cs.nctu.edu.tw

***Abstract*—In several applications, sequence databases generally update incrementally with time. Obviously, it is impractical and inefficient to re-mine sequential patterns from scratch every time a number of new sequences are added into the database. Some recent studies have focused on mining sequential patterns in an incremental manner; however, most of them only considered patterns extracted from time point-based data. In this paper, we proposed an efficient algorithm, *Inc_TPMiner*, to incrementally mine sequential patterns from interval-based data. We also employ some optimization techniques to reduce the search space effectively. The experimental results indicate that Inc_TPMiner is efficient in execution time and possesses scalability. Finally, we show the practicability of incremental mining of interval-based sequential patterns on real datasets.**

Keywords- dynamic representation; incremental mining; interval-based pattern; sequential pattern mining

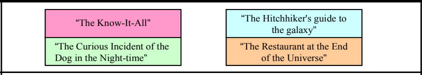
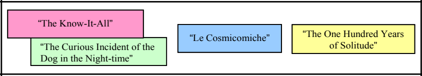
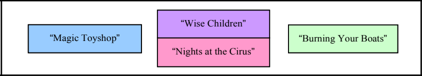
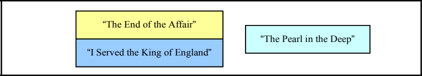
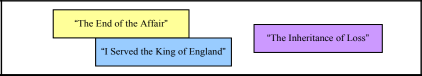
I. Introduction

Sequential pattern mining is an essential data mining technique with broad applications. Several efficient algorithms exhibit excellent performance in discovering sequential patterns from a static database, i.e., mining the entire database from scratch. However, the assumption of having a static database may not hold in some applications. Usually, new data may be added over time. The result discovered from the original database may no longer be valid. Apparently, for each database update, re-mining the databases from scratch is inefficient because it wastes the computational resources and neglects the previous results.

Previous research on incremental mining [2, 4, 7, 8, 11, 17] mainly focused on sequential patterns discovered from time point-based data. However, prior efforts have revealed that mining time interval-based patterns is more practical [3, 5, 6, 9, 10, 12, 13, 15, 16] in reality. In many applications, some events that intrinsically persist for periods of time instead of instantaneous occurrences cannot be treated as "time points." The data is usually a sequence of interval events with starting and finishing times. Examples include library lending, stock fluctuation, patient diseases, and meteorology data, to name a few. Hence, mining interval-based sequential patterns, also referred to as ***temporal patterns***, has more potential to uncover useful information.

Consider an example of pattern mining from Library datasets [3]. Usually, there is duration between the time a reader borrows and returns a book. By extracting the users' lending patterns, we could develop a recommendation system for the library. This information would be more helpful than conventional sequential time point-based pattern. Table 1 illustrates the part of discovered temporal patterns. We use patterns 1 and 2 for discussion. Suppose a reader checks out the books "*The Know-It-All*" and "*The Curious Incident of the Dog in the Night-time*" simultaneously, the library can send him an e-mail to notify that the book "*The Hitchhiker's Guide to the Galaxy*" is still on the shelf. However, if he checks out two books at different times, the library may send him an e-mail about the availability of the books "*Le Cosmicomiche.*" Clearly, the temporal patterns can offer a more expressive result to present correlations among data than conventional sequential patterns.

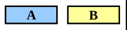
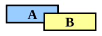
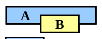
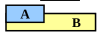
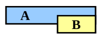
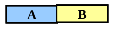
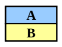
Table 1: The temporal patterns from the Library dataset [3].

PID	temporal patterns	support
1	 "The Know-It-All" "The Curious Incident of the Dog in the Night-time" "The Hitchhiker's guide to the galaxy" "The Restaurant at the End of the Universe"	163 (0.57%)
2	 "The Know-It-All" "The Curious Incident of the Dog in the Night-time" "Le Cosmicomiche" "The One Hundred Years of Solitude"	43 (0.15%)
3	 "Magic Toyshop" "Wise Children" "Nights at the Cirus" "Burning Your Boats"	97 (0.34%)
4	 "The End of the Affair" "I Served the King of England" "The Pearl in the Deep"	92 (0.32%)
5	 "The End of the Affair" "I Served the King of England" "The Inheritance of Loss"	35 (0.12%)

In real applications, interval-based database generally evolves with time, i.e., new data have been inserted and appended. However, incremental mining of temporal patterns from interval-based data is complex and arduous, and requires a different approach from time point-based sequential patterns. To the best of our knowledge, little attention has been paid to this issue, partly because of the complex relationship among event intervals. Since the feature of time intervals differs considerably from that of time points, the pairwise relationships between any two interval events are intrinsically complex. When appending an interval to an event sequence, the complex relations may lead to the generation of a greater number of possible candidates.

Allen's 13 temporal relations [1] are usually adopted to describe the complex relations among intervals, as shown in Fig. 1. However, Allen's temporal logics are binary relations and may be problematic when describing relationships among more than three event intervals. An appropriate representation is crucial for this circumstance. Various representations [3, 5, 6, 9, 10, 13, 16] have been proposed; however, most of them have a restriction on either ambiguity or scalability and do not consider the processing of incremental maintenance.

Table 2: Allen's 13 temporal relations between two intervals.

Temporal Relation	Inversed Relation	Pictorial Example	Endpoint sequence
A before *B*	*B* after *A*		$A^+ A^- B^+ B^-$
A overlaps *B*	*B* overlapped-by *A*		$A^+ B^+ A^- B^-$
A contains *B*	*B* during *A*		$A^+ B^+ B^- A^-$
A starts *B*	*B* started-by *A*		$(A^+ B^+) A^- B^-$
A finished-by *B*	*B* finishes *A*		$A^+ B^+ (A^- B^-)$
A meets *B*	*B* met-by *A*		$A^+ (A^- B^+) B^-$
A equal *B*	*B* equal *A*		$(A^+ B^+) (A^- B^-)$

In this paper, we aim to design algorithms to incrementally mine temporal patterns. The contributions of our work are as follows:

- First, we develop a new representation, *dynamic representation*, to express a pattern nonambiguously. We use the arrangement of endpoints of all intervals to simplify the processing of complex relation among intervals, and consider the time information to facilitate incremental mining.
- Second, based on the dynamic representation, an algorithm, *Inc_TPMiner* (Incremental Temporal Pattern Miner), is proposed to incrementally discover temporal patterns in database. Experimental studies indicated that, in incremental environment, Inc_TPMiner is efficient and outperforms other state-of-the-art algorithms. Our experiments also revealed that the proposed approach is scalable and consumes a smaller memory space.
- Third, Inc_TPMiner employs some pruning strategies to reduce the search space and avoids non-promising database projection. The experimental results reveal that pruning strategies can improvement the runtime performance of Inc_TPMiner efficiently.
- Finally, we applied Inc_TPMiner on real datasets to demonstrate the practicability of incremental maintenance of the temporal patterns.

The rest of the paper is organized as follows. Section 2 provides the related works. Section 3 introduces the preliminary and the dynamic representation. Section 4 describes the Inc_TPMiner algorithm. Section 5 gives the experiments and performance study, and we conclude in Section 6.

II. Related Works

Some prior works have focused on incrementally mining sequential patterns from time point-based data. Masseglia et. al [8] use a sequence lattice included the frequent sequences and the sequences in the negative border for incremental mining. Zhang et al. [17] developed two algorithms, GSP+ and MFS+, for incremental mining of sequential patterns when sequences are inserted into or deleted from the original database. The IncSpan [4] buffers a set of semi-frequent sequences as the candidates in the updated database which can accelerate the mining process. IncSpan+ [11] corrects the incompleteness and weaknesses of IncSpan. The IncSP [7] solved the maintenance problem through effective implicit merging and efficient separate counting. PBIncSpan [2] uses a prefix tree to record all frequent sequences and corresponding projected databases to incrementally mine the sequential patterns.

Several algorithms have been proposed to discover temporal patterns from interval-based data. Kam et al. [6] proposed a priori-like algorithm to discover temporal patterns based on hierarchy representation. Hoppner [8] proposed a nonambiguous representation, relation matrix, which exhaustively lists all binary relationships between event intervals in a pattern. H-DFS [12] discovers frequent arrangements by merging the id-lists iteratively to generate temporal patterns. IEMiner [13] proposes some optimization strategies to reduce the search space and decrease candidates to discover temporal patterns. ARMADA [15] uses a stem-growth method to find frequent temporal patterns from a large database. TPrefixSpan [16] generates all the possible candidates and then discovers frequent events and scans the projected databases recursively to discover all temporal patterns. SIPO [23] used the partial order among semi-intervals and found an abstraction that can represent many examples with similar properties. CTMiner [3] utilizes the coincidence concept to facilitate the mining process of temporal patterns.

All of the aforementioned algorithms only focus on maintaining sequential patterns from time point-based data or mining temporal patterns from time interval-based data. Little effort has been put to maintain discovered temporal pattern. In this paper, we discuss and design an efficient method to incrementally mine temporal patterns from interval-based database.

III. Problem Definition

Definition 1 (interval sequence and temporal database) Let $E = \{e_1, e_2, \ldots, e_k\}$ be the set of event symbols. We say the triplet $(e_i, s_i, f_i) \in E \times N \times N$ is an interval, where $e_i \in E$, $s_i, f_i \in N$ *(natural number set)* and $s_i < f_i$. The s_i and f_i are called the starting time and the finishing time, respectively. An interval sequence q is a series of interval triplets $\langle(e_1, s_1, f_1), \ldots, (e_n, s_n, f_n)\rangle$. The time information of q is the starting time of first interval and the finishing time of last interval in q, i.e., s_1 and f_n. A database $DB = \{r_1, r_2, \ldots, r_m\}$ is called a temporal database where each record r_i is a pair of sequence-id (SID) and interval sequence, i.e., $r_i = \langle SID_i, q_i\rangle$.

Definition 2 (dynamic representation) Given an event sequence $q = \langle(e_1, s_1, f_1), \ldots, (e_i, s_i, f_i), \ldots, (e_n, s_n, f_n)\rangle$, $T_q = \{ s_1, f_1, \ldots, s_i, f_i, \ldots, s_n, f_n \}$ is a set of all endpoints in q. After sorting T in non-decreasing order, an endpoint sequence $q_e = \langle t_1, t_2, \ldots, t_{2n}\rangle$ can be derived by representing

s_i and f_i as e_i^+ and e_i^-, respectively. We use the parenthesis to form an endpointset to indicate the times of endpoints are the same. The corresponding endpoint sequences of 13 Allen's temporal relations are shown in Table 1. To deal with multiple occurrences of events, we attach **occurrence number** to endpoint to distinguish multiple occurrences of the same event type in an endpoint sequence. The dynamic representation of q includes the corresponding endpoint sequence q_e and time information $[s_1, f_n]$ of q. For example, given an event sequence $\langle(A, 1, 3), (B, 5, 9)\rangle$, its time set is $\{1, 3, 5, 9\}$; hence, the corresponding endpoint sequence is $\langle A^+A^- B^+B^-\rangle$. The dynamic representation of q is $\langle A^+A^- B^+ B^- \rangle$ [1, 9]. Without loss of generality, for the rest of this paper, we suppose all the sequences in a temporal database have been transformed into dynamic representation.

Definition 3 (temporal pattern and pattern tree) Given a temporal database DB, a record $\langle SID, q_e, [s, f]\rangle$ is said to contain an endpoint sequence α, if α is a subsequence of q_e (represented as $\alpha \sqsubseteq q_e$). The support of α in DB is the number of records containing α, i.e., $support\ (\alpha) = |\{ \langle SID, q_e, [s, f]\rangle \in DB) \mid \alpha \sqsubseteq q_e\}|$. Given a positive integer *min_sup* as the support threshold, the set of temporal patterns includes all endpoint sequences whose supports are no less than *min_sup*. A frequent pattern tree (*FPT*) T is a tree that represents the set of temporal patterns in database. A node d in T stores an endpoint corresponding to a temporal pattern that starts from the root node to d. Each node also preserves two information, say ***support_value*** and ***sequence_list***. The support_value represents the support count of the temporal pattern. The sequence_list stores a list of *SID*s to represent the sequences containing this temporal pattern.

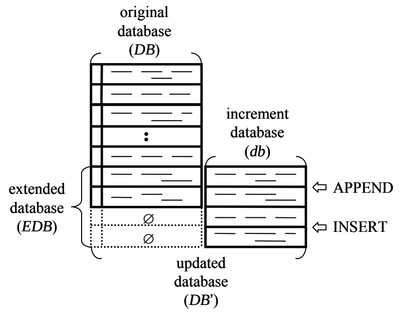


Fig. 1: Concept of incremental update in temporal database.

Actually, two types of incremental updates for interval sequence database are used: 1) inserting new sequences into database, denoted as INSERT; 2) appending new intervals to existing sequences, denoted as APPEND. An application may include all types of updates. When the database is updated with a combination of INSERT and APPEND, we can regard the INSERT as a special case of APPEND, for inserting a new sequence is equivalent to appending a new sequence to an empty sequence, as shown in Fig. 1.

With three interval sequences q, q' and q'', $q'' = q \diamond q'$ means q' is the concatenation of q. q' is called the **appended sequence** of q. q'' is an **updated sequence** of q appended with q'. To facilitate the presentation of this paper, we define increment and update databases. Given a temporal database, DB, truncated and appended with a few event sequences after a period, DB is called **original database**.

Definition 4 (increment and updated database) The increment database db is referred to as the set of newly appended sequences. The *SID*s of the appended sequences in db may already exist in DB. A database DB combining all the event sequences in db is referred to as the updated database DB', as shown in Fig. 1.

As mentioned above, appending an interval sequence is more challenging than conventional sequence. Since an interval has duration, an interval in existing sequence may merge with an interval in appended sequence. Given two intervals I_1 and I_2 with the same event symbol and I_1 is in existing sequence and I_2 is in appended sequence, if the end time of I_1 is the same with the start time of I_2, I_1 and I_2 will merge together. The interval-extension may vary the relation among intervals in the sequence, hence also modify the representation of the interval sequence, as shown in Fig. 2.

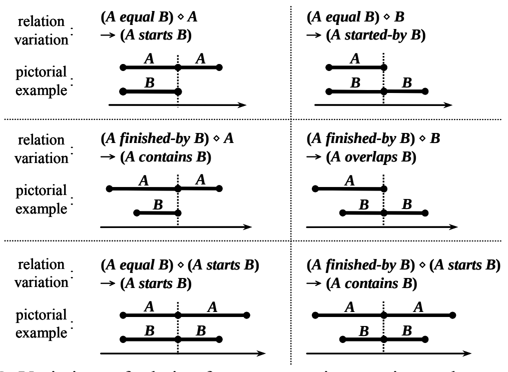

relation variation:
(A equal B) ◇ A → (A starts B)
(A equal B) ◇ B → (A started-by B)
pictorial example:
A
A
B
A
B
B
relation variation:
(A finished-by B) ◇ A → (A contains B)
(A finished-by B) ◇ B → (A overlaps B)
pictorial example:
A
A
B
A
B
B
relation variation:
(A equal B) ◇ (A starts B) → (A starts B)
(A finished-by B) ◇ (A starts B) → (A contains B)
pictorial example:
A
A
B
B
A
A
B
B

Fig. 2: Variations of relation for concatenating two interval sequences.

Definition 5 (interval extension) Suppose two interval sequences with dynamic representation, $q = \langle E_1, E_2, \ldots, E_n\rangle$, $[s, f]$ and $q' = \langle E_1', E_2', \ldots, E_m'\rangle$, $[s', f']$, and $E_n = (a_1, \ldots, a_x)$, $E_1' = (b_1, \ldots, b_y)$, where E_i and E_i' are endpointsets and a_j, b_j are endpoints. We say that a_i and b_j are **similar**, denoted as $a_i \approx b_j$, if the event symbol of a_i is identical to the event symbol of b_j. There are two kinds of concatenation for endpoint sequences q and q',

i) Sequence-extension: $q \diamond q' = \langle E_1, E_2, \ldots, E_n, E_1', E_2', \ldots, E_m'\rangle$ if $f \neq s'$.

ii) Endpoint-extension: $q \diamond q' = \langle E_1, E_2, \ldots, E_{n-1}, E_x, E_2', \ldots, E_m'\rangle$ if $f = s'$, where $E_x = (c_1, \ldots, c_x, c_{x+1}, \ldots, c_{x+y})$,

$$c_k = \begin{cases} a_k & \text{if } (1 \le k \le x) \text{ and } (\nexists\, a_k \approx b_i \text{ where } 1 \le i \le y) \\ b_{k\text{-}x} & \text{if } (x \le k \le x+y) \text{ and } (\nexists\, b_{k\text{-}x} \approx a_j \text{ where } 1 \le j \le x) \\ \varnothing & \text{if } (\exists\, a_k \approx b_i \text{ where } 1 \le i \le y) \\ & \qquad \text{or } (\exists\, b_{k\text{-}x} \approx a_j \text{ where } 1 \le j \le x) \end{cases}$$

IV. Inc_TPMiner Algorithm

When a temporal database *DB* is updated to *DB'*, there are three possible cases for the temporal patterns in *DB'*: Case1: A pattern is frequent in *DB'*, and also frequent in *DB*. Case2: A pattern is frequent in *DB'*, and infrequent in *DB* but has a frequent pattern in *DB* as a prefix. Case3: A pattern is frequent in *DB'*, and infrequent in *DB* and has no any frequent patterns in *DB* as a prefix.

Case1 is easy to handle since we have already stored the information of previous mining results into FPT_{DB}. We can obtain the temporal patterns in Case1 by checking and adjusting the support of every pattern in FPT_{DB} in *DB'*. In case2, although we have not preserved any information of infrequent sequences in *DB*, all temporal patterns have at least one prefix subsequence which is frequent in *DB*, i.e., the frequent prefix is stored in FPT_{DB}. Hence, we can utilize the temporal patterns in FPT_{DB} as prefix to recursively discover the temporal patterns in Case 2. Since, in Case 3, the temporal patterns have no information stored in previous mining results, FPT_{DB}, we need to scan *DB'* for all new frequent endpoints, and then use each new frequent endpoint as prefix to construct projected database and recursively mine all temporal patterns in Case 3.

Before introducing Inc_TPMiner, we first give an intuitive approach, *Naïve_Method*, for incremental mining temporal patterns, as shown in Algorithm 1. We will also use it for baseline comparisons to assess the merit of Inc_TPMiner later. It first determines the extended database, *EDB*, and transforms all interval sequences in *DB'* to dynamic representation (line 3). Then it calls ***CPrefixSpan*** on *EDB* and store mined results in a pattern tree, PT_{EDB} (line 4, algorithm 1). Note that, when mining *EDB*, the mined results should include both frequent and infrequent patterns, i.e., the *min_sup* is set as 1. Since even a pattern is infrequent in *EDB*, it still may become frequent in the updated database *DB'*. For each temporal pattern in FTP_{DB}, we update its support count if it also exists in PT_{EDB} and check whether it is still frequent in *DB'* (lines 5-10, algorithm 1). Finally, we verify each remaining pattern in PT_{EDB} in *DB – EDB* to adjust the support and output if it is frequent in *DB'* (lines 13-20, algorithm 1).

CPrefixSpan extends the concept of projected database from [14] and employs two optimization strategies to reduce the search space. Since the starting endpoints and finishing endpoints definitely occur in pairs in a sequence, we only project the frequent finishing endpoints which have the corresponding starting endpoints in their prefixes (lines 3-5, procedure 1). We can prune off non-qualified patterns before constructing projected database. Moreover, when constructing a projected database, some endpoints in postfixes need not be considered. With respect to a prefix $\langle p \rangle$, a finishing endpoint in a projected postfix is called significant, if it has corresponding starting endpoint in $\langle p \rangle$. We construct the projected database $DB_{|\langle p \rangle}$ by collecting significant endpoints only (line 9, procedure 1). All insignificant endpoints are eliminated since they can be ignored in the discovery of temporal patterns.

In order to calculate the support of all patterns which are infrequent in *DB* but frequent in *DB'*, Naïve_Method keeps the information of all possible candidate set, i.e., mining *EDB* with *min_sup* = 1 (line 3, algorithm 1). This awkward approach may consume large memory and many non-promising database projection. To remedy this problem, we design an algorithm, Inc_TPMiner, with two optimization techniques to reduce unnecessary space searches.

Algorithm 1: *Naïve_Method* (*DB'*, *min_sup*, FPT_{DB})

```
Input: DB': updated temporal database, min_sup: the minimum support, $FPT_{DB}$:
        frequent pattern tree of original DB
Output: $FPT_{DB'}$ : frequent pattern tree of updated database DB'

01: $FPT_{DB'} \leftarrow \varnothing$; $PT_{EDB} \leftarrow \varnothing$;
02: determine EDB;
03: use interval extension to transform DB' into dynamic representation and find
    all frequent endpoints concurrently;
04: $PT_{EDB}$ ← CPrefixSpan (EDB, ⟨ ⟩, $PT_{EDB}$);
05: for each node $\alpha$ in $FPT_{DB}$ do
06:     if $\alpha \in PT_{EDB}$
07:         update support ($\alpha$) and delete node $\alpha$ in $PT_{EDB}$ ;
08:     if support ($\alpha$) ≥ min_sup
09:         insert node $\alpha$ to $FPT_{DB'}$ ;
10:     else
11:         delete node $\alpha$ and all its descendent node in $FPT_{DB}$ ;
12: scan DB − EDB once for updating the support of node in $PT_{EDB}$ ;
13: for each node $\beta$ in $PT_{EDB}$ do
14:     if support($\beta$) ≥ min_sup
15:         insert node $\beta$ to $FPT_{DB'}$ ;
16:     else
17:         delete node $\beta$ and all its descendent node in $PT_{EDB}$ ;
18: for each frequent endpoint $b \notin FPT_{DB'}$ do
19:     CPrefixSpan ( DB', b, min_sup, $FPT_{DB'}$ );
20: Output $FPT_{DB'}$ ;

Procedure 1: CPrefixSpan ($DB_{|\alpha}$ , $\alpha$ , min_sup, $FPT_{DB}$ )
01: scan $DB_{|\alpha}$ once and find all frequent endpoints c;
02: for each frequent endpoint c do
03:     if c is a "finishing endpoint" then
04:         if exist corresponding starting endpoint in $\alpha$ then
05:             append c to $\alpha$ to form $\beta$ ;
06:     if c is a "starting endpoint" then
07:         append c to $\alpha$ to form $\beta$ ;
08: for each $\beta$ do
09:     construct projected database $DB_{|\beta}$ with insignificant postfix elimination;
10:     if $|DB_{|\beta}| \ge$ min_sup then
11:         insert $\beta$ into $FPT_{DB}$ ;
12:     call CPrefixSpan ($DB_{|\beta}$ , $\beta$, min_sup, $FPT_{DB}$ );
```

Definition 6 (search reduction) Given a temporal pattern α in *DB* (node α in FPT_{DB}), when *DB* is updated to *DB'*, *incre_sid* is defined as a set of all SIDs in increment database *db* and $incre\_endpoint_{|\alpha}$ is defined as a set of all event slices in $db_{|\alpha}$. We have two search space reductions,

i) Sequence-reduction: If {α' s sequence list} ∩ *incre_sid* = ∅, then $DB_{|\alpha}$ is identical to $DB'_{|\alpha}$. The support of α and all temporal patterns prefixed with α, i.e., node α and all child nodes of α in FPT_{DB}, are unchanged in *DB'*. Hence there is no temporal pattern which is infrequent in *DB* but becomes frequent in *DB'* with α as prefix. We can stop searching α and all α's child nodes in FPT_{DB}.

ii) Endpoint-reduction: If α' s parent node in in FPT_{DB} does not insert any node as child node when *DB* is updated to *DB'*, and the set of $\{\alpha$ and all α' s sibling nodes$\} \cap$ *incre_ endpoint*$_{|\alpha} = \varnothing$, then the support of α and all temporal patterns prefixed with α, i.e., node α and all child nodes of α in FPT_{DB}, are unchanged in *DB'*. Hence there is no temporal pattern which is infrequent in *DB* but becomes frequent in *DB'* with α as prefix. We can stop searching α and all child nodes of α in FPT_{DB}.

The search space reduction in Definition 6 plays an important role in Inc_TPMiner. When the minimum support goes lower and the maintained patterns turn to be longer, many unnecessary searches can be avoided effectively. As observed in our experiments, the search space reduction can skip more than 60% nodes in FPT_{DB}, especially when minimum support is extremely low. This is also the main reason why Inc_TPMiner not only outperforms other algorithms in runtime performance, but also consumes less memory space. The pseudo code of Inc_TPMiner is shown as in Algorithm 2.

Algorithm 2: *Inc_TPMiner* (*DB', min_sup, FPT$_{DB}$*)

```
Input: DB': updated temporal database, min_sup: the minimum support, $FPT_{DB}$:
        frequent pattern tree of original DB
Output: $FPT_{DB'}$ : frequent pattern tree of updated database DB'

// initial Phase
01: $FPT_{DB'} \leftarrow \varnothing$; determine EDB;
02: use interval_extension to transform DB' into dynamic presentation
    and find all frequent endpoints concurrently;
03: NFS ← new frequent endpoints in DB' ; // frequent endpoints in
    $DB' \notin FPT_{DB}$
// mining phase
04: for each endpoint b in NFS do
05:     insert b into $FPT_{DB'}$ ;
06:     call CPrefixSpan ($DB'_{|b}$ , b , min_sup, $FPT_{DB'}$ );
// extending phase
07: scan DB' once for update the support of node in $FPT_{DB}$ ;
08: for each node $\alpha$ in $FPT_{DB}$ do
09:     $FPT_{DB} \leftarrow$ CPrefixSpan ( DB', $\alpha$, min_sup, $FPT_{DB}$);
10:     for each node $\alpha$ in $FPT_{DB} \geq$ min_supdo
11:         insert $\alpha$ into $FPT_{DB'}$ ;
12:     if search_reduction ($\alpha$, $DB'_{|\alpha}$) = "false" // search reduction
13:         call CPrefixSpan ($DB'_{|\alpha}$ , $\alpha$ , min_sup, $FPT_{DB'}$ );
14: Output $FPT_{DB'}$ ;
```

There are three phases in Inc_TPMiner, initial phase, mining phase and extending phase. Initial phase first uses the interval extension to transform all sequences into dynamic representation (line 2), and scans *db* once to discover all new frequent endpoints in *DB'*. Notice that, if we store previous infrequent endpoints in *DB*, we can find the complete set of new frequent endpoints in *DB'* by just scan *EDB* without rescanning *DB* again (Line 3). Then, in mining phase, we use each new frequent slice as prefix to construct projected database and call *CPrefixSpan* (procedure 1) to discover the temporal patterns (Lines 4-6 algorithm 2). Note that the *search_reduction* technique in Definition 7 can be used in CPrefixSpan when we call it recursively. We can add one line " if *search_reduction* (β, *DB'*$_{|\beta}$) = "false" " before line 12 in procedure 1. We utilize *search_reduction* to check whether growing can stop. If not, we recursively call CPrefixSpan to discover the temporal patterns.

Finally, in extending phase, Inc_TPMiner updates the support of every frequent pattern in *DB*. If a pattern is still frequent in *DB'*, we also use *search_reduction* to check if we can stop growing. If not, *CPrefixSpan* is called to discover the temporal patterns (Lines 12-13, algorithm 2).

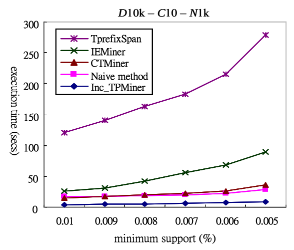
V. EXPERIMENTAL RESULT

To evaluate the performance of Inc_TPMiner, we implement CTMiner [3], TPrefixSpan [15], IEMiner [13] and Naïve method for comparison. All algorithms were implemented in C^{++} language and tested on a computer with Pentium D 3.0 GHz with 2 GB of main memory. The performance study has been conducted on both synthetic and real world datasets. First, we compare the execution time and memory usage using synthetic datasets at extreme low minimum support. Second, we run Inc_TPMiner on different scenario to reflect the influence on performance of updated environments. Third, we conduct an experiment to observe the scalability on execution time of Inc_TPMiner. Finally, we use a real dataset [3] to show the performance and the practicability of incremental mining for temporal patterns.

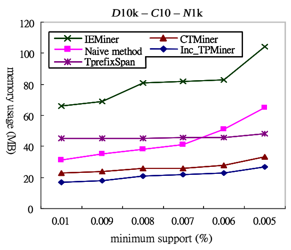
Table 3: Parameters of synthetic data generator.

parameters	description
$\lvert D \rvert$	Number of event sequences
$\lvert C \rvert$	Average size of event sequences
$\lvert S \rvert$	Average size of potentially frequent sequences
N_S	Number of potentially frequent sequences
N	Number of event symbols
R_{inc}	Ratio of the number of sequences in increment database *db* to updated database *DB'*
R_{ext}	Ratio of the number of existed sequences extended to new sequences inserted in increment database *db*
R_{app}	Ratio of the number of intervals of an existed sequence appearing in original database *DB* to increment database *db*

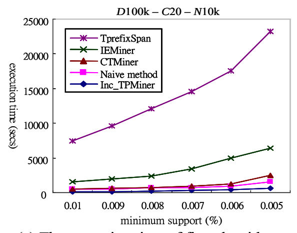
The synthetic datasets are generated using synthetic generation program [3]. Since the original data generation program was designed to generate static database, the generator requires modifications on incremental scenario accordingly. The parameter setting of temporal data generator is shown in Table 3. We partition the updated database *DB'* into the original database *DB* and increment database *db*, as the example in Fig. 1. Different settings of three parameters are used to reflect different updating scenarios. Parameter R_{inc}, called *increment ratio*, decides the size of the increment database *db*. We pick $\lvert D \rvert \times R_{inc}$ sequences randomly into *db* and place remaining $\lvert D \rvert \times (1 - R_{inc})$ sequences into *DB*. Furthermore, we use *extended ratio*, R_{ext}, to divide event sequences in *db* to "old" sequences, which's *sid* have appeared in *DB*, and "new" inserted sequences. Total $\lvert db \rvert \times R_{ext}$ sequences were randomly chosen from *db* as “old” sequence which were to be split further. The splitting of event sequences is to simulate that some intervals are conducted formerly (thus in *DB*), while the remaining intervals are newly appended (thus in *db*). The splitting is controlled by the third parameter R_{app}, the *appended ratio*. If a sequence with total *m* intervals is to split, we placed the leading $m \times (1 - R_{app})$ intervals in *DB* and the remaining $m \times R_{app}$ intervals in db_a.



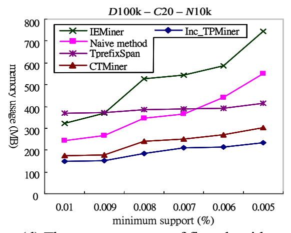
(a) The execution time of five algorithms



(b) The memory usage of five algorithms



(c) The execution time of five algorithms



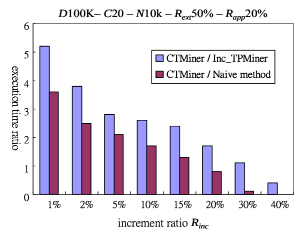
(d) The memory usage of five algorithms

Fig. 3: Execution time and memory usage on synthetic datasets.

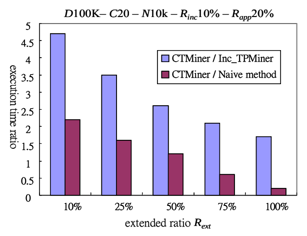
5.1 Execution time and memory usage

In all the following experiments, two parameters are fixed, i.e., the average size of potentially frequent sequences, $|S| = 4$, and the number of potentially frequent sequences, $N_S = 5{,}000$. We set $R_{inc} = 10\%$, $R_{ext} = 50\%$ and $R_{app} = 20\%$ to model common database updating scenario. The first experiment for comparison of five algorithms is on the dataset *D*10k–*C*10–*N*1k with the minimum support thresholds varying from 0.01 % to 0.005 %. Obviously, re-mining from scratch with non-incremental algorithm is less efficient than using incremental maintaining algorithm, as illustrated in Fig. 3(a). When we continue to lower the minimum threshold, the runtime of Inc_TPMiner outperforms the other four algorithms. We can see that when the support is larger than 0.009 %, CTMiner outperforms Naïve method partly because of the generation of a fewer number of frequent patterns for the maintenance. The memory usages of five algorithms are showed as in Fig. 3(b). We can see that Inc_TPMiner consumes less memory than the other four algorithms.

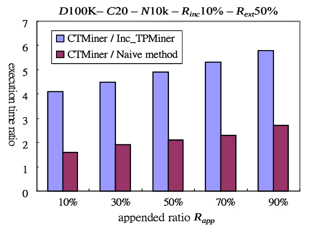
The second experiment is performed on data set *D*100k–*C*20–*N*10k, which contains 100,000 event sequences, average length 40 and 10,000 event intervals with common database updating scenario. The execution time of different algorithms is shown in Fig. 3(c). We can see that when the support is 0.005%, Inc_TPMiner is more than 2.4 times faster than Naïve method. Fig. 3(d) shows the memory usages of five algorithms with different minimum support thresholds. We can see that although Naïve method has better performance on execution time than re-running CTMiner from scratch, it involves larger memory space for execution partly because of storing every possible frequent sequences and doing many non-promising database projection.



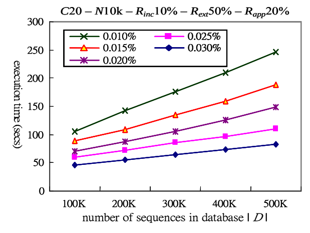
(a) Execution ratio over varying increment ratios



(b) Execution ratio over varying extended ratios



(c) Execution time over varying appended ratios



(d) The performance with different database size

Fig. 4: Execution time on different updating scenario and scalability test.

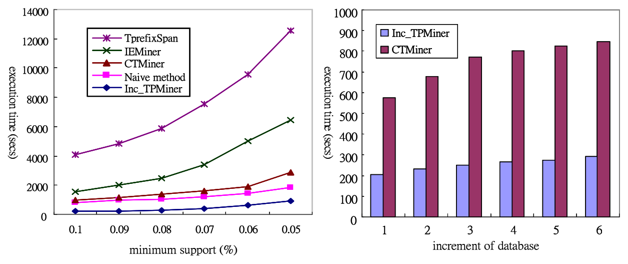
5.2 Different updating scenario and scalability test

In order to reflect the influence of incremental environment on time performance, three parameters, increment ratio, extended ratio and appended ratio, are configured to generate different updating scenarios for comparing the execution times. Generally, incremental mining algorithms gain less at higher increment ratio because larger increment ratio means more sequences appearing in *db* and causes more pattern updates. If most of the frequent sequences in *DB* turn out to be invalid in *DB’*, the information stored by maintenance algorithms in pattern updating might become useless.

Fig. 4(a) is the results of varying increment ratio, R_{inc}, from 1% to 40% on *D*100k – *C*20 – *N*10k. The *min_sup* is fixed at 0.01%. Note that we use the execution time ratio to show the improvement of incremental mining algorithms over CTMiner (i.e., the execution time of incremental maintaining algorithm / the execution time of Inc_TPMiner). As indicated in Fig. 4(a), the smaller the increment database *db* is, the more time Inc_TPMiner could save. Inc_TPMiner is still faster than CTMiner even when R_{inc} reaches 40%. When R_{inc} becomes much larger, say over 40%, Inc_TPMiner is slower than CTMiner. When the size of the increment database becomes larger than the size of the original database, i.e. the database has accumulated dramatic change, re-mining from scratch might be a better choice for the totally new sequence database.

The impact of the extended ratio, R_{ext}, is presented in Fig. 4(b) on *D*100k – *C*20 – *N*10k dataset with *min_sup* = 0.01%. Clearly, Inc_TPMiner updates patterns more efficiently than Naïve method and CTMiner. Higher R_{ext} means that there are more sequences in the original database expended in the increment database. Consequently, the speedup ratio decreases as the R_{ext} increases because more appended sequence need to be processed. We can observe that Inc_TPMiner is efficient even when the R_{ext} is increased to 100%, i.e., all the sequences in the increment database are extended from original database. Fig. 4(c) depicts the performance comparisons of Inc_TPMiner and Naïve method with CTMiner concerning appended ratios, R_{app}, on D100k – C20 – N10k dataset. We can see that Inc_TPMiner is constantly about 5.3 times faster than CTMiner over various R_{app}, ranging from 10% to 90%.

We also study the scalability on the execution time of Inc_TPMiner. Here, the total number of sequences is increased from 100K to 500K, with fixed parameters C = 20, N = 10k, R_{inc} = 10%, R_{ext} = 50% and R_{app} = 20%. Fig. 4(d) shows the results of scalability tests with different *min_sup* varying from 0.03 % to 0.01 %. As can be seen, under different minimum support threshold, Inc_TPMiner is still linearly scalable with different database size.



(a) Performance of algorithms on Library dataset [3] (b) Multiple increments in Library dataset [3]

Fig. 5: Performance test on Library dataset [3].

6.3 Real World Dataset Analysis

In addition to using synthetic datasets, we have also performed an experiment on real dataset to compare the performance and indicate the applicability of temporal pattern mining. The Library dataset [3] consists of a collection lending and returning records of a library for three years. First, we use the records of first two and half years to construct the original database *DB* and use the record of last half year to build the increment database *db*.

Fig. 5(a) shows the performance of execution time with varying minimum support thresholds. As *min_sup* drops to 0.05 %, Inc_TPMiner is almost 2 times faster than Naïve method and more than 2.7 times faster than CTMiner. Finally, we discuss the performance of Inc_TPMiner to process multiple database updates. We still use the records of first two and half years to construct *DB* and divide the records of the rest half years by every one month to build six different *db*. Fig. 5(b) shows the performance of Inc_TPMiner, with *min_sup* = 0.1%, to incrementally maintain multiple database updates. Each time the database is updated, we also run CTMiner to re-mine from scratch for comparison. Clearly, when the increments accumulate, the time for incremental mining also increases, but increase is very small. The incremental mining still outperforms re-mining with CTMiner by a factor of 2.5 or 3.5. Obviously, Inc_TPMiner is efficient for multiple updates of database.

VI. CONCLUSION

Little attention has been paid to the incremental mining of temporal patterns from interval-based data. Since the process of complex relations among intervals may require generating and examining large amount of intermediate subsequences, incrementally mining temporal patterns is a challenging problem. In this paper, we develop a new representation, *dynamic representation*, to simplify the processing of complex relation and facilitate incremental mining. Furthermore, a new algorithm, Inc_TPMiner, is developed to balance the efficiency and reusability with two optimization methods, sequence-reduction and slice-reduction. The experimental results indicate that both execution time and memory usage of Inc_TPMiner outperform previous algorithms designed based on static database. Finally, we apply the algorithm on real dataset to show the efficiency and the practicability of incremental mining of temporal patterns.

REFERENCES

[1] J. Allen, "Maintaining Knowledge about Temporal Intervals," *Communications of ACM*, vol.26, issue 11, pp.832-843, 1983.

[2] Y. Chen, J. Guo, Y. Wang, Y. Xiong and Y. Zhu, "Incremental Mining of Sequential Patterns using Prefix Tree," *PAKDD'07*, pp. 433-440, 2007.

[3] Y. Chen, J. Jiang, W. Peng and S. Lee, "An Efficient Algorithm for Mining Time Interval-based Patterns in Large Databases," *ACM CIKM'10*, pp. 49-58, 2010.

[4] H. Cheng, X. Yan and J. Han, "IncSpan: Incremental Mining of Sequential Patterns in Large Database," *ACM KDD'04*, pp. 527-532, 2004

[5] F. Hoppner, "Finding informative rules in interval sequences," *Intelligent Data Analysis*, vol. 6, no. 3, pp. 237-255, 2002.

[6] P. Kam and W. Fu, "Discovering Temporal Patterns for Interval-based Events," *DaWaK'00*, pp. 317-326, 2000.

[7] M. Lin and S. Lee, "Incremental update on sequential patterns in large databases by implicit merging and efficient counting," *Information Systems*, vol. 29, issue 5, pp. 385-404, 2004.

[8] F. Masseglia, P. Poncelet and M. Teisseire, "Incremental mining of sequential patterns in large databases," *Data & Knowledge Engineering*, vol.46, pp.97–121, 2003

[9] F. Morchen and D. Fradkin, "Robust mining of time intervals with semi-interval partial order patterns," *SIAM SDM'10*, pp.315-326, 2010.

[10] S. Nguyen, X. Sun, M. Orlowska, "Improvements of IncSpan: Incremental Mining of Sequential Patterns in Large Database," *PAKDD'05*, pp. 442-451, 2005.

[11] P. Papapetrou, G. Kollios, S. Sclaroff, and D. Gunopulos, "Discovering frequent arrangements of temporal intervals," *IEEE ICDM'05*, pp. 354-361, 2005.

[12] D. Patel, W. Hsu and M. Lee, "Mining Relationships Among Interval-based Events for Classification," *ACM KDD'08*, pp. 393-404, 2008.

[13] 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," *IEEE ICDE'01*, pp. 215-224, 2001.

[14] E. Winarko and J.F Roddick, "ARMADA-An algorithm for discovering richer relative temporal association rules from interval-based data," *Data & Knowledge Engineering*, vol. 63, issue 1, pp. 76-90, 2007.

[15] S. Wu and Y. Chen, "Mining Nonambiguous Temporal Patterns for Interval-Based Events," *IEEE Transactions on Knowledge and Data Engineering*, vol.19, no. 6, pp. 742-758, 2007.

[16] M. Zhang, B. Kao, D. Cheung, and C. Yip, "Efficient algorithms for incremental updates of frequent sequences," *PAKDD'02*, pp.186-197, 2002.