Discovering Recurring Patterns in Time Series

R. Uday Kiran[†], Haichuan Shang[†], Masashi Toyoda[†] and Masaru Kitsuregawa[†‡]

[†]The University of Tokyo, Tokyo, Japan
[‡]National Institute of Informatics, Tokyo, Japan
{uday_rage, shang, toyoda, kitsure}@tkl.iis.u-tokyo.ac.jp

ABSTRACT

Partial periodic patterns are an important class of regularities that exist in a time series. A key property of these patterns is that they can start, stop, and restart anywhere within a series. We classify partial periodic patterns into two types: (i) regular patterns − patterns exhibiting periodic behavior throughout a series with some exceptions and (ii) recurring patterns − patterns exhibiting periodic behavior only for particular time intervals within a series. Past studies on partial periodic search have been primarily focused on finding regular patterns. One cannot ignore the knowledge pertaining to recurring patterns. This is because they provide useful information pertaining to seasonal or temporal associations between events. Finding recurring patterns is a non-trivial task because of two main reasons. (i) Each recurring pattern is associated with temporal information pertaining to its durations of periodic appearances in a series. Obtaining this information is challenging because the information can vary within and across patterns. (ii) Finding all recurring patterns is a computationally expensive process since they do not satisfy the anti-monotonic property. In this paper, we propose recurring pattern model by addressing the above issues. We also propose Recurring Pattern growth algorithm along with an efficient pruning technique to discover these patterns. Experimental results show that recurring patterns can be useful and that our algorithm is efficient.

Categories and Subject Descriptors

H.2.8 [**Database Management**]: Database Applications - Data Mining.

General Terms

Algorithms

Keywords

Data mining, periodic pattern mining, time series

1. INTRODUCTION

A time series is a collection of events obtained from sequential measurements over time. Periodic pattern mining involves finding all patterns that exhibit either complete or partial cyclic repetitions in a time series. Past studies on partial periodic search have been focused on finding *regular patterns*, i.e., patterns exhibiting either complete or partial cyclic repetitions throughout a series [1, 2, 3, 4, 5, 6, 7, 8, 9]. An example regular pattern of $\{Bat, Ball\}$ states that customers have been purchasing items 'Bat' and '$Ball$' almost every day throughout the year. A useful related type of partial periodic pattern is *recurring patterns*, i.e., patterns exhibiting cyclic repetitions only for particular time intervals within a series. An example recurring pattern of $\{Jackets, Gloves\}$ states that customers have often purchased '$Jackets$' and '$Gloves$' from 10-October-2012 to 26-February-2013 and from 2-November-2013 to 2-March-2014. The purpose of this paper is to discover recurring patterns by addressing mining challenges.

Recurring patterns are ubiquitous in very large datasets. In many real-world applications, they can provide useful information pertaining to seasonal or temporal associations between items. In retail, a user may be interested in determining seasonal purchases for efficient inventory management. Similarly, a social network data analyst may be interested in obtaining temporal information pertaining to bursts of hashtags, such as #earthquakes, #radiation and #floods. Also, an expert in the health-care sector may be interested in finding seasonal diseases in a geographical location. To improve web site design and administration, an administrator may be interested in obtaining temporal information of heavily visited web pages. In the stock market, the set of high stocks indices that rise periodically for a particular time interval may be of special interest to companies and individuals. In a computer network, an administrator may be interested in finding high severity events (e.g. cascading failure) against regular routine events (e.g. data backup).

Unfortunately, finding recurring patterns is a non-trivial task because of the following reasons.

1. Each recurring pattern is associated with temporal information pertaining to its durations of periodic appearances within the data. Obtaining this information is challenging because the information can vary within and across patterns.

2. Most current periodic pattern mining approaches take into account a time series as a symbolic sequence; therefore, they do not take into account the actual temporal

information of events.

3. Recurring patterns do not satisfy the *anti-monotonic property*. That is, all non-empty subsets of a recurring pattern may not be recurring patterns. This increases the search space, which in turn increases the computational cost of finding these patterns. Therefore, developing efficient pruning techniques to reduce the search space is challenging.

4. Since regular patterns exhibit periodic behavior throughout a series with some exceptions, regular pattern mining algorithms do not obtain temporal information pertaining to the durations of periodic appearances of a pattern within the series. As a result, these algorithms cannot be extended for finding recurring patterns.

5. In real-life, recurring patterns involving rare items can be interesting to users. For example, the knowledge pertaining to rare events, such as cascading failures, are more important than regular events for a network administrator. However, finding such patterns is difficult since rare items appear infrequently in the data. Classifying items into frequent or rare is subjective and depends on the user and/or application requirements.

In this paper, we propose a model that addresses all the above-mentioned issues while finding recurring patterns. In particular, our model takes into account time series as a time-based sequence and models it as a transactional database with transactions ordered in respect to a particular timestamp (without loss of generality). Our model consists of three novel measures, *periodic-support*, *periodic-interval* and *recurrence*, to determine the dynamic periodic behavior of recurring patterns. *Periodic-support* determines the number of consecutive cyclic repetitions of a pattern in a subset of data. *Periodic-interval* determines the time interval (or window) pertaining to the periodic appearances of a pattern within a series. *Recurrence* determines the number of interesting *periodic intervals* of a pattern. Finally, we propose a pattern-growth algorithm along with an efficient pruning technique to discover recurring patterns effectively. We call our algorithm recurring pattern-growth (RP-growth). Experimental results show that RP-growth is efficient and recurring patterns can provide useful information in many real-life applications.

The rest of the paper is organized as follows. Section 2 describes related work on mining periodic patterns. Section 3 introduces our model of recurring patterns. Section 4 presents RP-growth. Sections 5 reports on the experimental results. Finally, Section 6 concludes the paper with future research directions.

2. RELATED WORK

Since the introduction of partial periodic patterns [5], the problem of finding these patterns has received a great deal of attention [6, 8, 10, 11, 12]. The model used in all these studies, however, remains the same. That is, it takes into account a time series as a symbolic sequence and finds all patterns using the following two steps:

1. Partition the symbolic sequence into distinct subsets (or period-segments) of a fixed length (or *period*).
2. Discover all partial periodic patterns that satisfy the user-defined *minimum support* (*minSup*), which controls the minimum number of period-segments that a pattern must cover though the sequence.

A major limitation of the above studies is that they do not take into account the actual temporal information of the events within a sequence. To address this issue, Ma and Hellerstein [7] modeled a time series as a time-based sequence and proposed a model to discover a class of partial periodic patterns known as **p-patterns**. In this model, a pattern is considered partial periodic if its number of periodic appearances throughout the sequence satisfies the user-defined *minSup*. It should be noted that the concept of *minSup* is not the same in both frequent pattern mining and partial periodic pattern mining. In frequent pattern mining, *minSup* controls the minimum number of appearances of a pattern throughout the data. However, in partial periodic pattern mining, *minSup* controls the minimum number of periodic appearances (or cyclic repetitions) of a pattern throughout the data. Thus, the partial periodic patterns discovered in all the above studies [6, 8, 10, 11, 12, 7] represent *regular patterns*. Our study, on the other hand, was focused on discovering recurring patterns in a time-based sequence. Moreover, Ma and Hellerstein's model cannot be extended for finding recurring patterns. The reasons are as follows:

1. Their model fails to obtain the temporal information pertaining to the durations of periodic appearances of a pattern within the data.
2. Finding p-patterns with a single *minSup* leads to the dilemma known as the "rare item problem" [13]. If *minSup* is set too high, those patterns that involve rare items will not be found. To find patterns involving both frequent and rare items, *minSup* has to be set very low. However, this can lead to combinatorial explosion producing too many patterns. In particular, many uninteresting aperiodic patterns can be discovered as partial periodic patterns. For example, if we set a low *minSup*, say *minSup* = 5%, then we will be discovering an uninteresting aperiodic pattern that has only 5% of its periodically appearances throughout the data as a partial periodic pattern.

Recently, researchers have been investigating the **complete cyclic behavior** of the frequent patterns in a transactional database to discover a class of user-interest-based patterns known as periodic-frequent patterns [9, 14, 15, 16]. Informally, a frequent pattern satisfying *minSup* is said to be **periodic-frequent** if and only if all its inter-arrival times throughout the database satisfy the user-defined *period* threshold value. Thus, these studies were focused on finding regular patterns in a transactional database. For our study, we investigated the partial cyclic behavior of the patterns to discover recurring patterns; thus, generalizing the current model of periodic-frequent patterns. More importantly, none of the approaches presented in [9, 14, 15, 16] model time series data as a transactional database. Instead, they are based on the implicit assumption that there are transactional databases with a sequentially ordered set of transactions. This paper fills the gap by describing the procedure to model time series data as a transactional database without loss of generality.

Yang et al. [17] investigated the change in periodic behavior of patterns due to the intervention of random noise and introduced a class of user-interest-based patterns known as *asynchronous periodic patterns*. Although asynchronous-periodic pattern mining is closely related to our work, it cannot be extended for finding recurring patterns. The reason is asynchronous periodic pattern mining models a time series as a symbolic sequence; therefore, it does not take in account the actual temporal information of the events within a sequence.

The problem of finding sequential patterns [18] and frequent episodes [19, 20] has received a great deal of attention. However, it should be noted that *periodicity* is not considered in these studies. Ozden et al. [2] investigated the problem of finding cyclic association rules. However, that study is quite restrictive in finding the patterns that are present at every cycle.

Finding partial periodic patterns [4], motifs [21], and recurring patterns [22] has also been studied in time series; however, the focus was on finding numerical curve patterns rather than symbolic patterns.

Overall, the proposed model of finding recurring patterns is novel and is distinct from current models. In the next section, we introduce our model of recurring patterns.

3. PROPOSED MODEL

In this section, we first describe time series as defined in [7]. Next, we represent these series as a transactional database and introduce measures to find recurring patterns.

DEFINITION 1. *Let I be a set of items (or event types). **An event** is a pair (i, ts), where $i \in I$ is an item and $ts \in R$ is the timestamp of the event. Let $X \subseteq I$ be a **pattern**. **An event sequence** S is an ordered collection of events, i.e., $\{(i_1, ts_1), (i_2, ts_2), \cdots, (i_N, ts_N)\}$, where $i_j \in I$ is an item at the j-th event. The term ts_j represents the occurrence timestamp of the event, and $ts_h \leq ts_j$ for $1 \leq h \leq j \leq N$ [7].*

DEFINITION 2. *A **point sequence** is an ordered collection of occurrence times. Given an event sequence $S = \{(i_1, ts_1), (i_2, ts_2), \cdots, (i_N, ts_N)\}$, there is an implied point sequence, $\widehat{S} = \{ts_1, ts_2, \cdots, ts_N\}$. An event sequence can be viewed as a mixture of multiple point sequences of each item. Let TSD denote the time series data (or a set of events) being mined.*

EXAMPLE 1. *Figure 1 shows a TSD with a set of items $I = \{a, b, c, d, e, f, g\}$. In this figure, an item of each event is labeled above its occurrence timestamp. It should be noted that no item appears at the timestamps of 8 and 13. The item 'a' appears at the timestamps of 1, 2, 3, 4, 7, 11, 12 and 14. Therefore, the event sequence of 'a' is represented as $S^a = \{(a, 1), (a, 2), (a, 3), (a, 4), (a, 7), (a, 11), (a, 12), (a, 14)\}$. The point sequence of 'a' is represented as $\widehat{S^a} = \{1, 2, 3, 4, 7, 11, 12, 14\}$. Similarly, the point sequences of 'b' and 'ab' are represented as $\widehat{S^b} = \widehat{S^{ab}} = \{1, 3, 4, 7, 11, 12, 14\}$.*

The point sequence plays an important role in assessing the periodic behavior of the patterns in a time series. We now describe the temporally ordered transactional database which preserves the point sequence of items in the TSD.

A **transaction**, $tr = (ts, Y)$, is a tuple, where ts represents the timestamp and Y is a pattern. A **transactional database** TDB over I is a set of transactions, $TDB = \{tr_1, \cdots, tr_m\}$, $m = |TDB|$, where $|TDB|$ is the size of the TDB in total number of transactions. For a transaction $tr = (ts, Y)$, such that $X \subseteq Y$, it is said that X occurs in tr and such a timestamp is denoted as ts^X. Let $TS^X = \{ts^X_k, \cdots, ts^X_l\}$, where $1 \leq k \leq l \leq m$, denote an **ordered set of timestamps** at which X has occurred in the TDB. The TS^X in the TDB is the same as the point sequence of X in the TSD. Therefore, we do not miss any information pertaining to the temporal appearances of a pattern in the data.

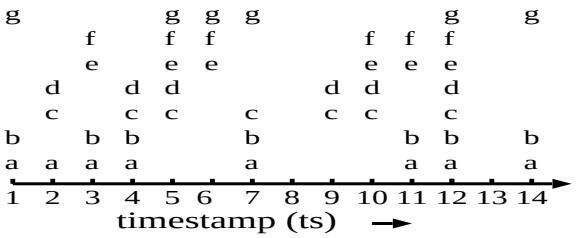


Figure 1: Running example: time-based sequence consisting of items from 'a' to 'g'

EXAMPLE 2. *Table 1 shows the transactional database constructed by grouping the items appearing together at a particular timestamp in Figure 1. Each transaction in this database is uniquely identifiable with a timestamp. All transactions have been ordered with respect to their timestamps. It can be observed that the constructed database does not contain the transactions with timestamps 8 and 13. The reason is that no item appears at these timestamps in Figure 1. In this database, the pattern 'ab' appears at the timestamps of $1, 3, 4, 7, 11, 12,$ and 14. Therefore, $TS^{ab} = \{1, 3, 4, 7, 11, 12, 14\}$.*

Table 1: Transactional database constructed from time-based sequence shown in Figure 1. The term 'ts' is an acronym for timestamp

ts	Items
1	*a, b, g*
2	*a, c, d*
3	*a, b, e, f*
4	*a, b, c, d*
5	*c, d, e, f,g*
6	*e, f, g*
7	*a, b, c, g*
9	*c, d*
10	*c, d, e ,f*
11	*a, b, e, f*
12	*a, b, c, d, e, f, g*
14	*a, b, g*

DEFINITION 3. ***(Support of pattern X)** The number of transactions containing X in the TDB is defined as the support of X and denoted as $Sup(X)$. That is, $Sup(X) = |TS^X|$.*

EXAMPLE 3. *The support of 'ab' in Table 1 is the size of TS^{ab}. Therefore, $Sup(ab) = |\{1, 3, 4, 7, 11, 12, 14\}| = 7$.*

DEFINITION 4. ***(Periodic appearance of pattern X)** Let $ts^X_j, ts^X_k \in TS^X$, $1 \leq j < k \leq m$, denote any two consecutive timestamps in TS^X. The time difference between ts^X_k and ts^X_j is referred to as **an inter-arrival time** of X, and denoted as iat^X. That is, $iat^X = ts^X_k - ts^X_j$. Let $IAT^X = \{iat^X_1, iat^X_2, \cdots, iat^X_k\}$, $k = Sup(X) - 1$, be the*

set of all inter-arrival times of X in TDB. An inter-arrival time of pattern X is said to be ***periodic*** *(or interesting) if it is no more than the user-defined period threshold value. That is, a $iat_i^X \in IAT^X$ is said to be* ***periodic*** *if $iat_i^X \leq per$, where 'per' represents the period.*

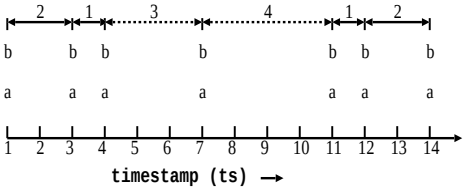


Figure 2: Inter-arrival times of 'ab', IAT^{ab}

Example 4. *The pattern 'ab' has initially appeared at the timestamps of 1 and 3. The difference between these two timestamps gives an inter-arrival time of 'ab.' That is, $iat_1^{ab} = 2\ (= 3 - 1)$. Similarly, other inter-arrival times of 'ab' are $iat_2^{ab} = 1$, $iat_3^{ab} = 3$, $iat_4^{ab} = 4$, $iat_5^{ab} = 1$ and $iat_6^{ab} = 2$. Therefore, the resultant $IAT^{ab} = \{2, 1, 3, 4, 1, 2\}$. If the user-defined $per = 2$, then iat_1^{ab}, iat_2^{ab}, iat_5^{ab} and iat_6^{ab} are considered the periodic occurrences of 'ab' in the data. On the other hand, iat_3^{ab} and iat_4^{ab} are considered the aperiodic occurrences of 'ab' as $iat_3^{ab} \not\leq per$ and $iat_4^{ab} \not\leq per$. Figure 2 shows the set of all inter-arrival times for pattern 'ab', i.e., IAT^{ab}. The thick lines represent the inter-arrival times that satisfy the period, while the dotted lines represent the inter-arrival times that fail to satisfy the period.*

Most current partial periodic pattern mining approaches use *minSup* to assess the periodic interestingness of a pattern [5]. This measure cannot be used for finding recurring patterns because it controls the minimum number of cyclic repetitions a pattern must have in all the data. Therefore, we introduce the following measures to determine the partial periodic behavior of recurring patterns.

Definition 5. ***(Periodic-interval of pattern X)*** *Let $TS_j^X = \{ts_p^X, \cdots, ts_q^X\} \subseteq TS^X$, $p \leq q$, be a set of timestamps such that $\forall ts_k^X \in TS_j^X$, $p \leq k < q$, $ts_{k+1}^X - ts_k^X \leq per$. The TS_j^X is a* ***maximal set*** *if there exists no superset in which an inter-arrival time between the two consecutive timestamps is no more than the period. The range of timestamps in TS_j^X represents a periodic-interval of X and is denoted as pi_j^X. That is, $pi_j^X = [ts_p^X, ts_q^X]$.*

Example 5. *The maximal sets of timestamps in which 'ab' has appeared within the user-defined $per = 2$ are: $TS_1^{ab} = \{1, 3, 4\}$, $TS_2^{ab} = \{7\}$, and $TS_3^{ab} = \{11, 12, 14\}$. Therefore, the corresponding periodic-intervals for 'ab' are $pi_1^{ab} = [1, 4]$, $pi_2^{ab} = [7, 7]$, and $pi_3^{ab} = [11, 14]$.*

The *periodic-interval*, as defined above, obtains information pertaining to the duration (or window) of periodic appearances of a pattern in a database. Most importantly, it can effectively determine the periodic durations that can vary within and across patterns. In very large databases, a pattern may have too many periodic-intervals. An efficient technique to reduce this number is to select only those periodic-intervals in which the number of cyclic repetitions of the corresponding pattern satisfies the user-defined threshold value. Thus, we introduce the following two definitions.

Definition 6. ***(Periodic-support of pattern X)*** *The size of TS_j^X is defined as the* periodic-support *of X, and denoted as ps_j^X. That is, $ps_j^X = |TS_j^X|$.*

Example 6. *The periodic-support of 'ab' in pi_1^{ab} is the size of $|TS_1^{ab}|$. Therefore, $ps_1^{ab} = |TS_1^{ab}| = 3$. Similarly, the periodic-supports of 'ab' in pi_2^{ab} and pi_3^{ab} are 1 and 3, respectively.*

In the real-world applications, some items appear very frequently in the data, while others rarely appear. We have observed that some rare items also exhibit periodic behavior in a portion of the data. The *periodic-support*, as defined above, facilitates the user to discover the knowledge pertaining to those frequent and rare items that have exhibited sufficient number of cyclic repetitions in a portion of database. Each *periodic-interval* of a pattern will have only one *periodic-support* and vice-versa. In other words, there is one-to-one relationship between the *periodic-intervals* and *periodic-supports* of a pattern.

Definition 7. ***(Interesting periodic-interval of pattern X)*** *Let $PI^X = \{pi_1^X, \cdots, pi_k^X\}$ and $PS^X = \{ps_1^X, \cdots, ps_k^X\}$, $1 \leq k$, be the complete set of periodic-intervals and periodic-supports of pattern X in the TDB, respectively. A $pi_k^X \in PI^X$ is said to be an interesting periodic-interval if its corresponding $ps_k^X \in PS^X$ has $ps_k^X \geq minPS$. The $minPS$ represents the user-defined minimum periodic-support.*

Example 7. *If the user-defined $minPS = 3$, then pi_1^{ab} and pi_3^{ab} are considered the interesting periodic-intervals of 'ab'. This is because $ps_1^{ab} \geq minPS$ and $ps_3^{ab} \geq minPS$. The pi_2^{ab} is considered an uninteresting periodic-interval of 'ab' as $ps_2^{ab} \not\geq minPS$.*

Since very large databases are generally composed over a very long time frame, it has been observed that some users may specify a constraint on the minimum number of interesting periodic-intervals. Thus, we introduce the following definitions.

Definition 8. ***(Recurrence of pattern X)*** *The* recurrence count *of a pattern represents its number of interesting periodic-intervals in a database. Let $IPI^X \subseteq PI^X$ be the set of periodic-intervals of X such that for every $pi_k^X \in IPI^X$, its corresponding $ps_k^X \geq minPS$. The* recurrence *of pattern X is denoted as $Rec(X) = |IPI^X|$.*

Example 8. *Continuing with the previous example, $IPI^{ab} = \{[1, 4], [11, 14]\}$. The recurrence of 'ab' is the size of IPI^{ab}. That is, $Rec(ab) = |IPI^{ab}| = 2$.*

Definition 9. ***(Recurring pattern X)*** *Pattern X is a* recurring pattern *if $Rec(X) \geq minRec$, where $minRec$ is the user-specified* minimum recurrence count*. Recurring pattern X is expressed as follows:*

$$X \quad [Sup(X), Rec(X), \{\{pi_k^X : ps_k^X\} | \forall pi_k^X \in IPI^X\}]. \quad (1)$$

Example 9. *If the user-defined $minRec = 2$, then 'ab' is a recurring pattern and is expressed as follows:*

ab $[support = 7, recurrence = 2, \{\{[1, 4] : 3\}, \{[11, 14] : 3\}\}]$.

The above pattern informs that 'ab' has occurred in 7 transactions and its periodic occurrence behavior of once in every two transactions consecutively for at least three times has been observed at two distinct subsets of a database whose timestamps are in the range [1, 4] and [11, 14]. Table 2 shows the complete set of recurring patterns discovered from Table 1.

Table 2: Recurring patterns in Table 1. Terms '*Sup*,' '*Rec*' and '*IPI*' respectively denote *support*, *recurrence*, and interesting periodic-intervals along with their periodic-supports

Pattern	*Sup*	*Rec*	*IPI*
a	8	2	{{[1,4]:4}, {[11,14]:3}}
b	7	2	{{[1,4]:3}, {[11,14]:3}}
d	6	2	{{[2,5]:3}, {[9,12]:3}}
e	6	2	{{[3,6]:3}, {[10,12]:3}}
f	6	2	{{[3,6]:3}, {[10,12]:3}}
ab	7	2	{{[1,4]:3}, {[11,14]:3}}
cd	6	2	{{[2,5]:3}, {[9,12]:3}}
ef	6	2	{{[3,6]:3}, {[10,12]:3}}

DEFINITION 10. *(**Problem Definition:**) Given a time-based sequence (i.e., a TSD), the problem of finding recurring patterns involve discovering all those patterns that satisfy the user-defined per, minPS and minRec constraints.*

The measures, *support*, *period* and *periodic-support*, can also be expressed in percentage of $|TDB|$. However, we use the former definitions for ease of explanation. Table 3 lists the nomenclature of different terms used in our model.

Table 3: Nomenclature of various terms used in our model

Terminology	Notation
The timestamp of a transaction containing X	ts_i^X
The set of all timestamps containing X	TS^X
The *support* of X	$Sup(X)$
An inter-arrival time of X	iat_i^X
The set of all inter-arrival times of X	IAT^X
The user-defined *period*	per
A *periodic-support* of X	ps_i^X
The set of all *periodic-supports* of X	PS^X
A *periodic-interval* of X	pi_i^X
The set of all *periodic-intervals* of X	PI^X
The set of interesting periodic-intervals of X	IPI^X
The *recurrence* of X	$Rec(X)$

The construction of a transactional database from a time series involves grouping the items appearing together at a particular timestamp and storing them in a linked hash table. As this process is simple and straight forward, we do not discuss it in this paper. Instead, we focus on finding the recurring patterns from the constructed database.

4. PROPOSED ALGORITHM

In this section, we first introduce our pruning technique to reduce the computational cost of finding recurring patterns. Next, we present our algorithm to mine the complete set of recurring patterns from the constructed database.

4.1 Basic Idea: Candidate patterns

The space of items in a database gives rise to a subset lattice. An itemset lattice is a conceptualization of search space while finding user-interest-based patterns. The **anti-monotonic property** has been widely used to reduce the search space [23]. Unfortunately, recurring patterns do not satisfy this property. This increases the search space, which in turn increases the computational cost of mining recurring patterns.

EXAMPLE 10. *Consider the patterns 'c' and 'cd' in Table 1. Given the user-defined per = 2, minPS = 3 and minRec = 2, the interesting periodic-intervals of 'c' and 'cd' are $\{[2,12]\}$ and $\{[2,5],[9,12]\}$, respectively. Therefore, the $Rec(c) = |\{[2,12]\}| = 1$ and $Rec(cd) = |\{[2,5],[9,12]\}| = 2$. As the $Rec(c) \not\geq minRec$, 'c' is not a recurring pattern. However, its superset 'cd' is a recurring pattern because $Rec(cd) \geq minRec$. Thus, the recurring patterns do not satisfy the anti-monotonic property. The same can be observed in Table 2.*

We introduce the following pruning technique to reduce the computational cost of finding recurring patterns.

> "Let $E_{rec}(X) = \sum_{i=1}^{|PS^X|} \left\lfloor \frac{ps_i^X}{minPS} \right\rfloor$. If $E_{rec}(X) < minRec$, then neither X nor its supersets will be recurring patterns"

The $E_{rec}(X)$ denotes the upper bound of *recurrence* that a superset of X can have in the database. Thus, we call $E_{rec}(X)$ the **estimated maximum recurrence of a superset of** X. The correctness of our pruning technique is straight forward to prove from Properties 1 and 2, and illustrated in Example 11.

PROPERTY 1. *For the pattern X, $E_{rec}(X) \geq Rec(X)$.*

PROPERTY 2. *If $X \subset Y$, then $TS^X \supseteq TS^Y$ and $E_{rec}(X) \geq E_{rec}(Y)$.*

EXAMPLE 11. *In Table 1, the item 'g' occurs in timestamps of 1, 5, 6, 7, 12 and 14. Therefore, $TS^g = \{1,5,6,7,12,14\}$ and $S(g) = 6$. If per = 2, minPS = 3 and minRec = 2, then $TS_1^g = \{1\}$, $TS_2^g = \{5,6,7\}$, $TS_3^g = \{12,14\}$, $ps_1^g = |TS_1^g| = 1$, $ps_2^g = |TS_2^g| = 3$ and $ps_3^g = |TS_3^g| = 2$. For this item, $E_{rec}(g) = 1 \left(= \sum_{i=1}^{3} \left\lfloor \frac{ps_i^g}{3} \right\rfloor = \left\lfloor \frac{1}{3} \right\rfloor + \left\lfloor \frac{3}{3} \right\rfloor + \left\lfloor \frac{2}{3} \right\rfloor \right)$. That is, any superset of 'g' can at most have recurrence value equal to 1, which is less than the user-defined minRec. Henceforth, pruning 'g' will not result in missing of any recurring pattern.*

Based on our proposed pruning technique, we introduce the following definition.

DEFINITION 11. ***(Candidate pattern X.)** Pattern X is a candidate pattern if $E_{rec}(X) \geq minRec$.*

A candidate pattern containing only one item is called a **candidate item**. The candidate patterns satisfy the *anti-monotonic property* (Property 2). Therefore, we use candidate k-patterns to discover recurring $(k+1)$-patterns.

4.2 RP-growth: Structure, Construction and Mining

Traditional Frequent Pattern-growth algorithm [24] cannot be used for finding recurring patterns. This is because the structure of FP-tree captures only the frequency and disregards the periodic behavior of the patterns in a database. To address this issue, RP-growth introduces an alternative tree structure known as an Recurring Pattern-tree (RP-tree).

Our RP-growth algorithm involves the following two steps: (i) construction of an RP-tree and (ii) recursive mining of the RP-tree to discover the complete set of recurring patterns. Before we describe the above two steps, we introduce the structure of an RP-tree.

4.2.1 Structure of RP-tree

The structure of an RP-tree includes a prefix-tree and a candidate item list, called the RP-list. The RP-list consists of each distinct *item* (i) with *support* (s), *estimated maximum recurrence* (E_{rec}), and a pointer pointing to the first node in the prefix-tree carrying the item.

The prefix-tree in an RP-tree resembles the prefix-tree in FP-tree. However, to obtain both frequency and *periodic* behavior of the patterns, the nodes in an RP-tree explicitly maintain the occurrence information for each transaction by keeping an occurrence timestamp list, called a ***ts*-list**. To achieve memory efficiency, only the last node of every transaction maintains the *ts*-list. Hence, two types of nodes are maintained in a RP-tree: **ordinary node** and **tail-node**. The former is a type of node similar to that used in an FP-tree, whereas the latter represents the last item of any sorted transaction. Therefore, the structure of a *tail*-node is $i[ts_p, ts_q, ..., ts_r]$, $1 \leq p \leq q \leq r \leq m$, where i is the node's item name and ts_i, $i \in [1, m]$, is the timestamp of a transaction containing the items from *root* up to the node i. The conceptual structure of an RP-tree is shown in Figure 3. Like an FP-tree, each node in an RP-tree maintains parent, children, and node traversal pointers. Please note that no node in an RP-tree maintains the support count as in an FP-tree. To facilitate a high degree of compactness, items in the prefix-tree are arranged in support-descending order.

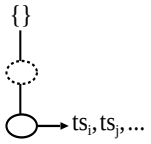


Figure 3: Conceptual structure of prefix-tree in RP-tree. Dotted ellipse represents ordinary node, while other ellipse represents tail-node of sorted transactions with timestamps $ts_i, ts_j \in R$

One can assume that the structure of the prefix-tree in an RP-tree may not be memory efficient since it explicitly maintains timestamps of each transaction. However, it has been argued that such a tree can achieve memory efficiency by keeping transaction information only at the *tail*-nodes and avoiding the support count field at each node [9]. Furthermore, an RP-tree avoids the *complicated combinatorial explosion problem of candidate generation* as in Apriori-like algorithms [23]. Keeping the information pertaining to transactional-identifiers in a tree can also be found in efficient frequent pattern mining [25].

Algorithm 1 RP-List(TDB: Transactional database, I: Set of items, *per*: period, *minPS*: minimum periodic-support, *minRec*: minimum recurrence)

1: Let id_l be a temporary array that records the *timestamp* of the last appearance of each item in the TDB. Let ps be another temporary array that records the periodic-support of an item in a subset of a database. Let $t = \{ts_{cur}, X\}$ denote the current transaction with ts_{cur} and X representing the timestamp of the current transaction and pattern, respectively.
2: **for** each transaction $t \in TDB$ **do**
3: **if** an item i occurs for the first time **then**
4: Add i to the RP-list.
5: Set $s^i = 1$, $e^i_{rec} = 0$, $id^i_l = ts_{cur}$ and $ps^i = 1$.
6: **else**
7: **if** $(ts_{cur} - id^i_l) \leq per$ **then**
8: Set s^i++, ps^i++ and $id^i_l = ts_{cur}$.
9: **else**
10: $e^i_{rec} += \left\lfloor \frac{ps^i}{minPS} \right\rfloor$.
11: Set s^i++, $ps^i = 1$ and $id^i_l = ts_{cur}$. {Beginning of a new subset of a database.}
12: **end if**
13: **end if**
14: **end for**
15: To reflect the correct estimated recurrence value for each item in the RP-list, perform $e^i_{rec} += \left\lfloor \frac{ps^i}{minPS} \right\rfloor$.

Algorithm 2 RP-Tree(TDB, RF-list)

1: Create the root of an RP-tree, T, and label it "*null*".
2: **for** each transaction $t \in TDB$ **do**
3: Set the timestamp of the corresponding transaction as t_{cur}.
4: Select and sort the candidate items in t according to the order of CI. Let the sorted candidate item list in t be $[p|P]$, where p is the first item and P is the remaining list.
5: Call $insert\_tree([p|P], t_{cur}, T)$.
6: **end for**
7: call RP-growth ($Tree$, *null*);

4.2.2 Construction of RP-tree

Since recurring patterns do not satisfy the *anti-monotonic property*, candidate 1-patterns (or items) will play an important role in effective mining of these patterns. The set of candidate items CI in a database for the user-defined *per*, *minPS*, and *minRec* can be discovered by populating the RP-list with a scan on the database. Figure 4 shows the construction of an RP-list using Algorithm 1. Due to page limitation, we only present the key steps in the construction of the RP-list. Please note that the *per*, *minPS*, and *minRec* values have been set to 2, 3 and 2, respectively.

The scan on the first transaction, "$1 : a, b, g$", with $ts_{cur} = 1$ initializes items 'a', 'b', and 'g' in the RP-list and sets their s, e_{rec}, id_l, and ps values to $1, 0, 1$, and 1, respectively (lines 1 to 5 in Algorithm 1). Figure 4(a) shows the RP-list generated after scanning the first transaction. The scan on the second transaction "$2 : a, c, d$" with $ts_{cur} = 2$ initializes the items 'c' and 'd' in the RP-list by setting their s, e_{rec}, id_l,

Algorithm 3 insert_tree($[p|P]$, t_{cur}, T)

1: **while** P is non-empty **do**
2: **if** T has a child N such that $p.itemName \neq N.itemName$ **then**
3: Create a new node N. Let its parent link be linked to T. Let its node-link be linked to nodes with the same itemName via the node-link structure. Remove p from P.
4: **end if**
5: **end while**
6: Add t_{cur} to the leaf node.

Algorithm 4 RP-growth($Tree$, α)

1: **for** each a_i in the header of Tree **do**
2: Generate pattern $\beta = a_i \cup \alpha$. Collect all of the $a_i's$ ts-lists into a temporary array, TS^{β}, and calculate E^{β}_{rec}.
3: **if** $E^{\beta}_{rec} \geq minRec$ **then**
4: Construct β's conditional pattern base then β's conditional RP-tree $Tree_{\beta}$. Call getRecurrence(β, TS^{β}).
5: **if** $Tree_{\beta} \neq \emptyset$ **then**
6: call RP-growth($Tree_{\beta}$, β);
7: **end if**
8: **end if**
9: Remove a_i from the $Tree$ and push the a_i's ts-list to its parent nodes.
10: **end for**

and ps values to $1, 0, 2$, and 1, respectively. In addition, the s, e_{rec}, id_l, and ps values of an already existing item 'a' are updated to $2, 0, 2$, and 2, respectively (lines 7 to 9 in Algorithm 1). Figure 4(b) shows the RP-list generated after scanning the second transaction. Figure 4(c) shows the RP-list constructed after scanning the seventh transaction. It can be observed that the 'e_{rec}' of 'a' and 'b' have been updated from 0 to 1. This is because their $\left\lfloor \frac{ps}{minPS} \right\rfloor = 1$ (line 10 in Algorithm 1). The ps value of 'a' and 'b' is set to 1 because they appeared periodically once again in the database (line 11 in Algorithm 1). Figure 4(d) shows the RP-list constructed after scanning every transaction in the database. The estimated recurrence (e_{rec}) value for all the items in the RP-list is once again computed to reflect the correctness (line 15 in Algorithm 1). Figure 4(e) shows the updated e_{rec} value for all items in the RP-list. Using our pruning technique, 'g' is removed from the RP-list as its $e_{cur} < minRec$. The remaining items are sorted in descending order of their support values (line 16 in Algorithm 1). Figure 4(f) shows the sorted list of candidate items in the RP-list.

After finding candidate items, we conduct another scan on the database and construct the prefix-tree of the RP-tree, as in Algorithms 2 and 3. These procedures are the same as those for constructing an FP-tree [24]. However, the major difference is that no node in an RF-tree maintains the *support* count, as in an FP-tree. The first transaction $\{1 : a, b, g\}$ is scanned and a branch is constructed in the RP-tree with only the candidate items 'b' and 'a.' The *tail* node '$b : 1$' carries the *timestamp* of the transaction. The RP-tree generated after scanning the first transaction is shown in Figure 5(a). A similar process is repeated for the remaining

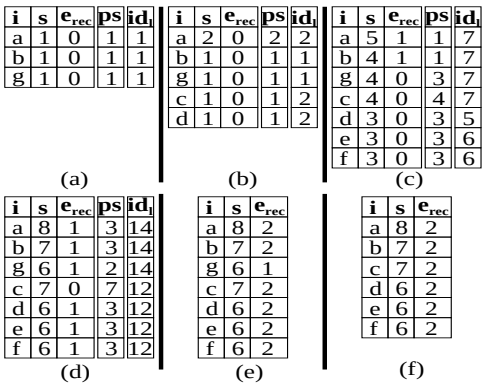


Figure 4: Construction of RP-list: (a) after scanning first transaction, (b) after scanning second transaction, (c) after scanning seventh transaction, (d) after scanning every transaction, (e) after calculating actual 'e_{rec}' values, and (f) sorted list of candidate items

transactions and the tree is updated accordingly. Figure 5(b) shows the RP-tree constructed after scanning the entire database. For simplicity, we do not show the node traversal pointers in trees; however, they are maintained like an FP-tree does.

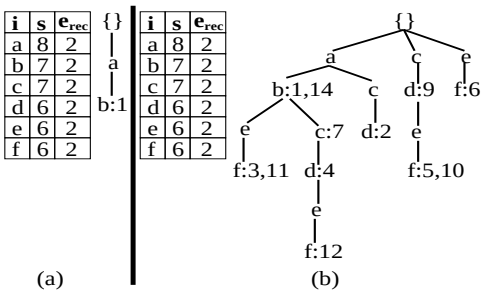


Figure 5: Construction of RP-tree: (a) after scanning first transaction and (b) after scanning entire transactional database

The RP-tree maintains the complete information of all recurring patterns in a database. The correctness is based on Property 3 and shown in Lemmas 1 and 2. For each transaction $t \in theTDB$, $CI(t)$ is the set of all candidate items in t, i.e., $CI(t) = item(t) \cap CI$, and is called the candidate item projection of t.

PROPERTY 3. *An RP-tree maintains a complete set of candidate item projections for each transaction in a database only once.*

LEMMA 1. *Given a TDB and user-defined per, (minPS), and minRec values, the complete set of all recurring item projections of all transactions in the TDB can be derived from the RP-tree.*

PROOF. Based on Property 3, each transaction $t \in TDB$ is mapped to only one path in the tree, and any path from the *root* up to a *tail* node maintains the complete projection for exactly n transactions (where n is the total number of entries in the *ts*-list of the *tail* node). □

LEMMA 2. *The size of the RP-tree (without the root node) on a TDB for user-defined per, minPS, and minRec is bounded by* $\sum_{t \in TDB} |CI(t)|$.

PROOF. According to the RP-tree construction process and Lemma 1, each transaction t contributes at most one path of size $|CI(t)|$ to an RP-tree. Therefore, the total size contribution of all transactions can be $\sum_{t \in TDB} |CI(t)|$ at best. However, since there are usually many common prefix patterns among the transactions, the size of an RP-tree is normally much smaller than $\sum_{t \in TDB} |CI(t)|$. □

Algorithm 5 getRecurrence(X: pattern, TS^X:ts-list of pattern X)

1: Let id_l be a variable that records the timestamp of the last transaction containing X. Let *subDB* be a list of pairs of the form $(startTS, endTS)$, where $startTS$ and $endTS$ respectively represent the starting and ending timestamps of periodic appearances of a pattern in a subset of data. It is used to record the periodic-intervals of a pattern. Let $currentPS$ be a variable to measure the periodic-support of X in a periodic-interval.
2: **for** each timestamp $ts_{cur} \in TS^X$ **do**
3: **if** (ts_{cur} is X's first occurrence) **then**
4: $currentPS = 1$, $startTS = ts_{cur}$;
5: **else**
6: **if** ($ts_{cur} - id_l \leq per$) **then**
7: $currentPS++$;
8: **else**
9: **if** ($currentPS \geq minPS$) **then**
10: $subDB.insert(startTS, id_l)$;
11: **end if**
12: $currentPS = 1$, $startTS = ts_{cur}$;
13: **end if**
14: **end if**
15: $id_l = ts_{cur}$;
16: **end for**
17: // To reflect correct recurrence of X.
18: **if** ($currentPS \geq minPS$) **then**
19: $subDB.insert(startTS, id_l)$;
20: **end if**
21: return (($subDB.size() \geq minRec$)?true:false);

4.2.3 Mining Recurring Patterns

Although an RP-tree and FP-tree arrange items in support-descending order, we cannot directly apply FP-growth mining on an RP-tree. The reasons are as follows: (*i*) an RP-tree does not maintain the support count at each node, and it handles the *ts*-lists at the *tail* nodes and (*ii*) recurring patterns do not satisfy the *anti-monotonic property*. We devised another pattern growth-based bottom-up mining technique to mine the patterns. The basic operations in mining an RP-tree includes: (*i*) counting length-1 candidate items, (*ii*) constructing the prefix tree from each candidate pattern, and (*iii*) constructing the conditional tree from each prefix-tree. The RP-list provides the length-1 candidate items. Before we discuss the prefix-tree construction process, we explore the following important property and lemma of an RP-tree.

PROPERTY 4. *A tail node in an RP-tree maintains the occurrence information for all the nodes in the path (from the tail node to the root) at least in the transactions in its ts-list.*

LEMMA 3. *Let* $Z = \{a_1, a_2, \cdots, a_n\}$ *be a path in an RP-tree where node* a_n *is the tail node carrying the ts-list of the path. If the ts-list is pushed-up to node* a_{n-1}*, then* a_{n-1} *maintains the occurrence information of the path* $Z' = \{a_1, a_2, \cdots, a_{n-1}\}$ *for the same set of transactions in the ts-list without any loss.*

PROOF. Based on Property 4, a_n maintains the occurrence information of path Z' at least in the transactions in its *ts*-list. Therefore, the same *ts*-list at node a_{n-1} maintains the same transaction information for Z' without any loss. □

The procedure to discover recurring patterns from RP-tree is shown in Algorithm 4. The working of this algorithm is as follows. We proceed to construct the prefix tree for each candidate item in the RP-list, starting from the bottom-most item, say i. To construct the prefix-tree for i, the prefix sub-paths of nodes i are accumulated in a tree-structure, PT_i. Since i is the bottom-most item in the RP-list, each node labeled i in the RP-tree must be a tail node. While constructing PT_i, based on Property 4, we map the *ts*-list of every node of i to all items in the respective path explicitly in the temporary array (one for each item). This temporary array facilitates the calculation of *support* and e_{rec} of each item in PT_i (line 2 in Algorithm 4). If an item j in PT_i has $support \geq minSup$ and $e_{rec} \geq minRec$, then we construct its conditional tree and mine it recursively to discover the recurring patterns (lines 3 to 8 in Algorithm 4). Moreover, to enable the construction of the prefix-tree for the next item in the RP-list, based on Lemma 3, the *ts*-lists are pushed-up to the respective parent nodes in the original RP-tree and in PT_i as well. All nodes of i in the original RP-tree and i's entry in the RP-list are deleted thereafter (line 9 in Algorithm 4).

Using Properties 1 and 2, the conditional tree CT_i for PT_i is constructed by removing all those items from PT_i that have $e_{rec} < minRec$. If the deleted node is a *tail* node, its *ts*-list is pushed-up to its parent node. The contents of the temporary array for the bottom item j in the RP-list of CT_i represent TS^{ij} (i.e., the set of all timestamps where items i and j have appeared together in the database). Therefore, using Algorithm 5, the *recurrence* of "ij" is computed and it is determined whether "ij" is a recurring pattern. The same process of creating a prefix-tree and its corresponding conditional tree is repeated for further extensions of "ij". The whole process of mining for each item is repeated until $RP\text{-}list \neq \emptyset$.

Consider item 'f', which is the last item in the RP-list in Figure 4(e). The prefix-tree for 'f', PT_f, is constructed from the RP-tree, as shown in Figure 6(a). There are five items, 'a, b, c, d', and 'e' in PT_f. Only item 'e' satisfies the condition $E_{rec}(e) \geq minRec$. Therefore, the conditional tree CT_f from PT_f is constructed with only one item 'e', as shown in 6(b). The *ts*-list of 'e' in 'CT_f' generates TS^{ef}. The "*recurrence*" of 'ef' is measured using Algorithm 5. Since $Rec(ef) \geq minRec$, 'ef' will be generated as a recurring pattern. A similar process is repeated for the other items in the RP-list. Next, 'f' is pruned from the original RP-tree

Table 4: User-defined *per*, *minPS* and *minRec* values in different databases

	per			*minPS*			*minRec*		
	1	2	3	1	2	3	1	2	3
T10I4D100k	360	720	1440	0.1%	0.2%	0.3%	1	2	3
Shop-14	360 (=6 hr.)	720 (=12 hrs.)	1440 (=24 hrs.)	0.1%	0.2%	0.3%	1	2	3
Twitter	360 (=6 hr.)	720 (=12 hrs.)	1440 (=24 hrs.)	2%	5%	10%	1	2	3

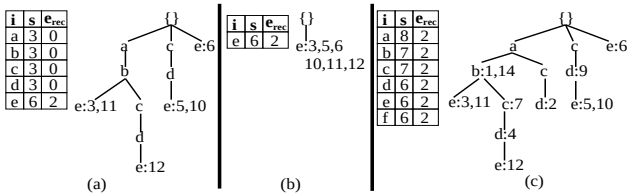


Figure 6: Finding RP-patterns for suffix item 'f' in RP-tree: (a) prefix-tree for item 'f' (i.e., PT_f), (b) conditional tree for item 'f' (i.e., CT_f), and (c) RP-tree after pruning item 'f'

and its *ts*-lists are pushed to its parent nodes, as shown in 6(c). All the above processes are once again repeated until the RP-list $\neq \emptyset$.

The above bottom-up mining technique on a support descending RP-tree is efficient, because it shrinks the search space dramatically as the mining process progresses.

5. EXPERIMENTAL RESULTS

In this section, we first evaluate the performance of RP-growth. Next, we discuss the usefulness of recurring patterns by comparing them against p-patterns and periodic-frequent patterns. It should be noted that our recurring patterns are the generalization of periodic-frequent patterns, as the latter patterns exhibit complete (rather than partial) cyclic repetitions in the entire database. There are only two Apriori-like algorithms, periodic-first and association-first, to discover p-patterns. We use the periodic-first algorithm to discover p-patterns since it is relatively faster than the association-first algorithm. We use the Periodic-Frequent pattern-growth++ algorithm (PF-growth++) [15] to discover periodic-frequent patterns. We do not compare the performance of RP-growth against the periodic-first and PF-growth++ algorithms. The reason is that the other algorithms discover regular patterns; therefore, they use different measures to assess the periodic interestingness of a pattern.

5.1 Experimental setup

The algorithms, RP-growth, periodic-first and PF-growth++, were written in GNU C++ and run with Ubuntu 14.4 on a 2.66 GHz machine with 8 GB of memory. To the best of our knowledge, there are no publicly available time-based sequences. Therefore, we conducted experiments using the following databases.

- **T10I4D100K database.** This database is a synthetic transactional database generated using the procedure given by [23]. This database contains 100,000 transactions and 941 distinct items.
- **Shop-14 database.** A Czech company provided click-stream data of seven online stores in the ECML/PKDD 2005 Discovery challenge. We considered the click stream data of product categories visited by the users in "Shop 14" (www.shop4.cz), and created a transactional database with each transaction representing the set of web pages visited by the people at a particular *minute interval*. The transactional database contains 59,240 transactions (i.e., 41 days of page visits) and 138 distinct items (or product categories).
- **Twitter database.** We created this database by considering the top 1000 English hashtags appearing in 44 million tweets/retweets from 1-May-2013 to 31-August-2013 (i.e., 123 days). The measure, *term frequency-inverse document frequency*, is used to rank the hashtags. The timestamp of each transaction represents a minute starting from 00:00 hours of 1-May-2013 to 24:00 hours of 31-August-2013. The resultant transactional database has 177,120 transactions with 1000 distinct items (or hashtags). More details on the data collection process and the usefulness of this data in finding interesting events has been presented in [26].

To mine p-patterns and recurring patterns, we transformed all the above databases into time-based sequences using the timestamps of each transaction. As this transformation process is a rather simple and straight-forward approach, we do not discuss it for brevity.

Table 4 lists the different *per*, *minPS* and *minRec* values used in above the databases. The periodic interval (i.e., *per* value) in both Shop-14 and Twitter databases varied from six hours to one day. Similarly, the *minRec* in these databases varied from 1 to 3. In this paper, we do not present the results for *minRec* values greater than 3 because very few recurring patterns were getting generated at $minRec > 3$. The *minPS* values in T10I4D100K and Shop-14 databases varied from 0.1% to 0.3%. The reason for choosing low *minPS* values is to discover the patterns involving both frequent and rare items. In the Twitter database, we set *minPS* at 2%, 5% and 10%. The reason for choosing a relatively high *minPS* values as compared with the other two databases is that very low *minPS* values are resulting in a combinatorial explosion producing too many patterns.

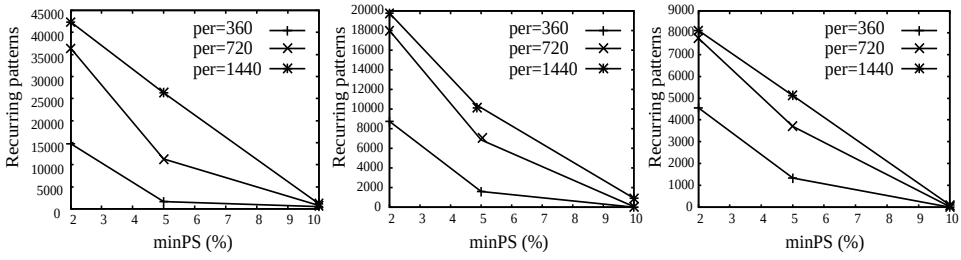
5.2 Generation of recurring patterns

Table 5 lists the numbers of recurring patterns discovered in T10I4D100K, Shop-14 and Twitter databases at different *per*, *minPS* and *minRec* values. The partial results of Table 5 are shown in Figure 7. The following observations can drawn from this figure:

- At a fixed *per* and *minRec*, the increase in *minPS* can decrease the number of recurring patterns. The reason is that many patterns failed to appear periodically for longer time periods.

Table 5: Number of recurring patterns generated at different *per*, *minPS* and *minRec* threshold values

Dataset	*minPS*	Number of recurring patterns								
		minRec = 1			*minRec* = 2			*minRec* = 3		
		per=360	*per*=720	*per*=1440	*per*=360	*per*=720	*per*=1440	*per*=360	*per*=720	*per*=1440
T10I4-D100k	0.1%	428	1254	7193	255	436	1036	194	160	27
	0.2%	339	757	3205	168	103	39	72	0	0
	0.3%	296	622	2148	109	32	2	21	0	0
Shop-14	0.1%	593	1885	4977	447	1339	3198	338	266	9
	0.2%	342	1077	1906	257	750	1470	118	14	0
	0.3%	251	744	933	195	534	760	48	3	0
Twitter	2%	14736	36354	42319	8718	17982	19746	4551	7749	8103
	5%	1655	11268	26341	595	6847	7010	337	3713	5123
	10%	511	714	1190	11	34	912	6	17	98



(a) minRec=1 (b) minRec=2 (c) minRec=3

Figure 7: Recurring patterns discovered in Twitter data

- At a fixed *minPS* and *per*, the increase in *minRec* can decrease the number of recurring patterns. This is because many patterns failed to satisfy the increased *minRec* values.
- At a fixed *minPS*, the increase in *per* can have different impact on the generation of recurring patterns for the values *minRec* = 1 and *minRec* > 1. At *minRec* = 1, increase in *per* can increase the number of recurring patterns. The reason is that the inter-arrival times of the patterns that were considered as aperiodic at low *per* values were considered as periodic with the increase in the *per* value. For *minRec* > 1, increase in *per* can either increase or decrease the number of recurring patterns. The reason for decrease is due to the merging of interesting periodic-intervals discovered at low *per* values.

Table 6 lists some of the recurring patterns discovered from the Twitter database at *per* = 360, *minPS* = 2% and *minRec* = 1. Figures 8 (a) and (b) show the frequencies of the terms present in patterns {yyc, uttarakhand} and {nuclear, hibaku} on a daily basis. It can be observed that "uttarakhand" is a relatively rare term as compared with other terms, and our model has effectively discovered the knowledge pertaining to this term. Another interesting observation from Table 5 is that even at low *minPS* values, our model has generated only a limited number of recurring patterns in each database. This clearly shows that our model can discover the knowledge pertaining to rare terms without producing too many uninteresting patterns. In other words, our model is tolerant of the "rare item problem".

5.3 Performance of RP-growth

Table 7 lists the runtime required using RP-growth to discover recurring patterns in T10I4D100K, Shop-14, and

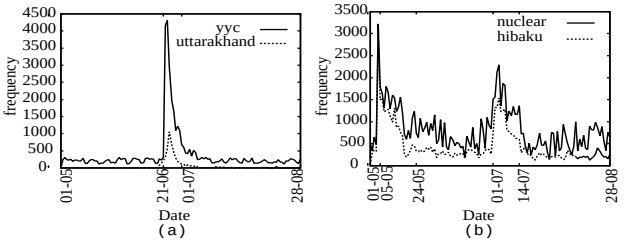


Figure 8: Frequency of hashtags at different days in database. Date is of form 'dd-mm'. Year of this date is 2013

Twitter databases. The runtime involves the time taken to transform the time-based sequence into a transactional database and mining of recurring patterns. Figure 9 shows the runtime required by RP-growth while mining the recurring patterns in Twitter database. The changes in the *per*, *minPS* and *minRec* threshold values shows a similar effect on runtime consumption as in the generation of recurring patterns. The proposed algorithm discovered the complete set of recurring patterns at a reasonable runtime even at low *minPS* thresholds.

5.4 Comparison of p-patterns, recurring patterns and periodic-frequent patterns

We compared p-patterns, recurring patterns and periodic-frequent patterns at different *per* and *minPS* values. For brevity, we present the results discovered when *period* is set to 1 day, i.e., *per* = 1440. The *minSup* and *minPS* values are set to 0.1% and 2% respectively for Shop-14 and Twitter databases. The p-pattern mining requires another parameter known as window length (w). We set $w = 1$ for our experiment.

Table 6: Some of the interesting recurring patterns discovered in Twitter database

S.No	Pattern	Periodic duration	Cause for the events
1	{yyc, uttarakhand}	[2013-06-21 01:08, 2013-07-01 04:27]	On June 20, Uttarakhand, a state in India and Alberta, a province in Canada have witnessed heavy floods.
2	{nuclear, hibaku} (In Japanese, hibaku means radiation.)	[2013-05-06 22:33, 2013-05-24 22:13], [2013-07-01 06:17, 2013-07-14 06:21]	(*i*) A Japanese minister has visited Chernobyl, Ukraine to learn from the recovery from the severe nuclear accident. (*ii*) People were tweeting about detection of Plutonium at a point 12 KM from Fukoshima nuclear reactor.
3	{pakvotes, nayapakistan}	[2013-05-09 16:15, 2013-05-15 14:11]	The general elections were held in Pakisthan. on May 11, 2013.
4	{oklahoma, tornado, prayforoklahoma}	[2013-05-21 11:52, 2013-05-24 21:38]	Oklahoma was struck with a tornado on May 20, 2013.

Table 7: Runtime of RP-growth at different *per*, *minPS* and *minRec* threshold values

Dataset	*minPS*	Runtime of RP-growth								
		minRec = 1			*minRec* = 2			*minRec* = 3		
		per=360	*per*=720	*per*=1440	*per*=360	*per*=720	*per*=1440	*per*=360	*per*=720	*per*=1440
T10I4-D100k	0.1%	14.8	150.9	366.5	3.8	10.7	40.1	3.5	3.9	6.3
	0.2%	7.7	45.9	99.6	3.6	5.4	9.6	2.7	3.1	3.1
	0.3%	3.7	11.6	21.3	3.2	3.4	4.2	2.5	2.4	2.6
Shop-14	0.1%	47.7	55.6	67.3	43.5	47.7	52.3	42.4	45.1	48.2
	0.2%	42.9	46.1	51.3	41.7	43.4	45.0	41.4	42.1	43.8
	0.3%	42.4	44.0	47.3	41.6	42.1	43.6	41.1	41.5	41.7
Twitter	2%	55.1	190.0	290.5	42.9	154.9	248.4	41.3	139.2	226.1
	5%	37.9	134.3	225.6	33.0	105.3	181.9	31.5	96.1	159.7
	10%	32.3	108.3	190.9	30.4	89.2	151.3	29.9	66.9	124.1

Table 8 lists the numbers of periodic-frequent patterns, recurring patterns and p-patterns discovered in the Shop-14 and Twitter databases. The column labeled '*II*' in this table refers to the length of the longest pattern discovered in each of these databases. The following observations can be drawn from this table.

First, the total numbers of periodic-frequent patterns discovered in both databases were relatively less than the number of recurring patterns and p-patterns. The reason is that the strict constraint that a frequent pattern has to exhibit complete cyclic repetitions throughout the data has failed to identify many interesting partial periodically appearing patterns. Moreover, this strict constraint also resulted in finding the very short periodic-frequent patterns (see columns labeled '*II*' in Table 8). This is because longer patterns generally fail to exhibit complete cyclic repetitions throughout the data. Setting a high *period* threshold value can enable a user to discover long periodic-frequent patterns. However, this high *period* can result in discovering sporadically appearing patterns as periodic-frequent patterns. Thus, it is necessary to relax this strict constraint without changing the *period* threshold value. As our model enables a user to relax this strict constraint, recurring patterns have been found more interesting than the periodic-frequent patterns for a given *per* threshold.

Second, the total number of p-patterns discovered in both databases were much higher than the recurring patterns and periodic-frequent patterns. The reason is that the usage of a low *minSup* has facilitated Ma and Hellerstein's model [7] to discover not only all our recurring patterns as p-patterns, but also resulted in a combinatorial explosion of frequently appearing items producing too many p-patterns. Most importantly, many of the p-patterns discovered at low *minSup* were uninteresting to the users. The reason is that frequent items were combining with one another in all possible ways and generating p-patterns by satisfying a low *minSup* value. On the contrary, our model has reduced the combinatorial explosion of frequent items by assessing their interestingness with respect to their number of consecutive periodic appearances in a portion of data.

Table 8: Number of patterns discovered in Shop-14 and Twitter databases. Terms '*I*' and '*II*' represent *total number of patterns* and *maximum length of pattern* found in each database, respectively.

	Shop-14		Twitter	
	I	*II*	*I*	*II*
PF patterns	22	3	466	2
Recurring patterns	4,977	9	42,319	7
p-patterns	156,7001	12	442,076	16

6. CONCLUSIONS AND FUTURE WORK

We introduced a new class of partial periodic patterns known as *recurring patterns* and discussed the usefulness of these patterns in various real-world applications. We also proposed a model for discovering such patterns. The patterns discovered with the proposed model do not satisfy the anti-monotonic property. Therefore, we proposed a novel pruning technique to reduce the computational cost of finding these patterns. We also proposed a pattern-growth algorithm to discover the recurring patterns effectively. Experimental results suggest that the model is tolerant to the "rare item problem" and that the algorithm is efficient. The usefulness of the recurring patterns was discussed by comparing them against the periodic-frequent and p-patterns.

In our current study, we did not considered noisy data and the phase-shifts of the items within the data. For fu-

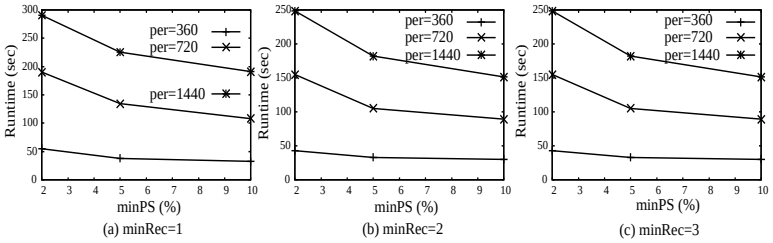


Figure 9: Runtime of RP-growth

ture work, we will develop methods for handling these two scenarios. Another interesting future work will be extending our model to improve the performance of an association rule-based recommender system.

7. ACKNOWLEDGMENTS

This work was supported by the Research and Development on Real World Big Data Integration and Analysis program of the Ministry of Education, Culture, Sports, Science, and Technology, JAPAN.

8. REFERENCES

[1] C. M. Antunes and A. L. Oliveira, "Temporal data mining: An overview," in *Workshop on Temporal Data Mining, KDD*, 2001.

[2] B. Özden, S. Ramaswamy, and A. Silberschatz, "Cyclic association rules," in *ICDE*, 1998, pp. 412–421.

[3] Z. Li, B. Ding, J. Han, R. Kays, and P. Nye, "Mining periodic behaviors for moving objects," in *KDD '10*, 2010, pp. 1099–1108.

[4] P. Esling and C. Agon, "Time-series data mining," *ACM Computing Surveys*, vol. 45, no. 1, pp. 12:1–12:34, Dec. 2012.

[5] J. Han, W. Gong, and Y. Yin, "Mining segment-wise periodic patterns in time-related databases." in *KDD*, 1998, pp. 214–218.

[6] J. Han, G. Dong, and Y. Yin, "Efficient mining of partial periodic patterns in time series database," in *ICDE*, 1999, pp. 106–115.

[7] S. Ma and J. Hellerstein, "Mining partially periodic event patterns with unknown periods," in *ICDE*, 2001, pp. 205–214.

[8] R. Yang, W. Wang, and P. Yu, "Infominer+: mining partial periodic patterns with gap penalties," in *ICDM*, 2002, pp. 725–728.

[9] S. K. Tanbeer, C. F. Ahmed, B. S. Jeong, and Y. K. Lee, "Discovering periodic-frequent patterns in transactional databases," in *PAKDD*, 2009, pp. 242–253.

[10] C. Berberidis, I. Vlahavas, W. Aref, M. Atallah, and A. Elmagarmid, "On the discovery of weak periodicities in large time series," in *PKDD*, 2002, pp. 51–61.

[11] H. Cao, D. Cheung, and N. Mamoulis, "Discovering partial periodic patterns in discrete data sequences," in *Advances in Knowledge Discovery and Data Mining*, 2004, vol. 3056, pp. 653–658.

[12] W. G. Aref, M. G. Elfeky, and A. K. Elmagarmid, "Incremental, online, and merge mining of partial periodic patterns in time-series databases," *IEEE TKDE*, vol. 16, no. 3, pp. 332–342, Mar. 2004.

[13] B. Liu, W. Hsu, and Y. Ma, "Mining association rules with multiple minimum supports," in *KDD*, 1999, pp. 337–341.

[14] R. U. Kiran and P. K. Reddy, "Towards efficient mining of periodic-frequent patterns in transactional databases," in *DEXA (2)*, 2010, pp. 194–208.

[15] R. U. Kiran and M. Kitsuregawa, "Novel techniques to reduce search space in periodic-frequent pattern mining," in *DASFAA (2)*, 2014, pp. 377–391.

[16] M. M. Rashid, M. R. Karim, B. S. Jeong, and H. J. Choi, "Efficient mining regularly frequent patterns in transactional databases," in *DASFAA (1)*, 2012, pp. 258–271.

[17] J. Yang, W. Wang, and P. Yu, "Mining asynchronous periodic patterns in time series data," *IEEE TKDE*, vol. 15, no. 3, pp. 613–628, May 2003.

[18] C. H. Mooney and J. F. Roddick, "Sequential pattern mining – approaches and algorithms," *ACM Comput. Surv.*, vol. 45, no. 2, pp. 19:1–19:39, Mar. 2013.

[19] H. Mannila, "Methods and problems in data mining," in *ICDT*, 1997, pp. 41–55.

[20] S. Laxman, P. S. Sastry, and K. P. Unnikrishnan, "A fast algorithm for finding frequent episodes in event streams," in *KDD*, 2007, pp. 410–419.

[21] T. Oates, "Peruse: An unsupervised algorithm for finding recurrig patterns in time series," in *ICDM*, 2002, pp. 330–337.

[22] Y. Mohammad and T. Nishida, "Approximately recurring motif discovery using shift density estimation," in *Recent Trends in Applied Artificial Intelligence*, 2013, pp. 141–150.

[23] R. Agrawal, T. Imieliński, and A. Swami, "Mining association rules between sets of items in large databases," in *SIGMOD*, 1993, pp. 207–216.

[24] J. Han, J. Pei, Y. Yin, and R. Mao, "Mining frequent patterns without candidate generation: A frequent-pattern tree approach," *Data Min. Knowl. Discov.*, vol. 8, no. 1, pp. 53–87, Jan. 2004.

[25] M. J. Zaki and C.-J. Hsiao, "Efficient algorithms for mining closed itemsets and their lattice structure," *IEEE Trans. on Knowl. and Data Eng.*, vol. 17, no. 4, pp. 462–478, Apr. 2005.

[26] G. Rattanaritnont, M. Toyoda, and M. Kitsuregawa, "Analyzing patterns of information cascades based on users' influence and posting behaviors," in *TempWeb '12*, 2012, pp. 1–8.