PARASOL: a hybrid approximation approach for scalable frequent itemset mining in streaming data

Yoshitaka Yamamoto¹ · Yasuo Tabei² · Koji Iwanuma³

Received: 13 March 2019 / Revised: 18 November 2019 / Accepted: 20 November 2019 /
Published online: 17 December 2019
© The Author(s) 2019

Abstract

Here, we present a novel algorithm for frequent itemset mining in streaming data (FIM-SD). For the past decade, various FIM-SD methods in one-pass approximation settings that allow to approximate the support of each itemset have been proposed. They can be categorized into two approximation types: *parameter-constrained* (PC) mining and *resource-constrained* (RC) mining. PC methods control the maximum error that can be included in the approximate support based on a pre-defined parameter. In contrast, RC methods limit the maximum memory consumption based on resource constraints. However, the existing PC methods can exponentially increase the memory consumption, while the existing RC methods can rapidly increase the maximum error. In this study, we address this problem by introducing a hybrid approach of PC-RC approximations, called *PARASOL*. For any streaming data, PARASOL ensures to provide a condensed representation, called a Δ -covered set, which is regarded as an extension of the closedness compression; when $\Delta = 0$, the solution corresponds to the ordinary closed itemsets. PARASOL searches for such approximate closed itemsets that can restore the frequent itemsets and their supports while the maximum error is bounded by an integer, Δ . Then, we empirically demonstrate that the proposed algorithm significantly outperforms the state-of-the-art PC and RC methods for FIM-SD.

Keywords Streaming data mining · Online approximation algorithm

✉ Yoshitaka Yamamoto
yyamamoto@inf.shizuoka.ac.jp

Yasuo Tabei
yasuo.tabei@riken.jp

Koji Iwanuma
iwanuma@yamanashi.ac.jp

¹ Shizuoka University, Shizuoka, Japan

² RIKEN Center for Advanced Intelligence Project, Tokyo, Japan

³ University of Yamanashi, Kofu, Japan

1 Introduction

Streaming data analysis is a central issue in many domains such as computer system monitoring (Du and Li 2016), online text analysis (Iwata et al. 2010; Hoffman et al. 2010), financial and economic analyses (Zhao et al. 2013; Zhu and Shasha 2002), and medical and health record data analysis (Ginsberg et al. 2009; Keogh et al. 2001). *Streaming data* is an infinite and continuous sequence of data, and, nowadays, is generated and collected rapidly. The sudden emergence of an intensive bursty event, called *concept drift*, in streaming data can make it difficult to extract meaningful information from such data. Therefore, there is a strong and growing need for powerful methods that are robust against concept drift in large-scale streaming data analysis.

Frequent itemset mining in streaming data (FIM-SD) (Han et al. 2007; Mala and Dhanaseelan 2011; Lee et al. 2014) is the most fundamental and well-used task in streaming data analysis. It is used to find frequently occurring itemsets (in this study, sets of non-negative integers) in streaming data (in this study, a sequence of itemsets). FIM-SD must exhibit two important properties; (i) the *real time property*, which is the ability to process a huge volume of itemsets continuously arriving at high speed and simultaneously (on-the-fly) output the detected frequent itemsets (FIs); and (ii) *memory efficiency*, which is the ability to enumerate FIs while managing an exponential number of candidate FIs with limited memory. Concept drift suddenly introduces huge itemsets that must be processed, which demands a huge amount of memory. This makes it difficult to design scalable and efficient FIM-SD. Thus, the development of a space-efficient FIM-SD that is especially robust against concept drift is an important open challenge that must be solved to enable large-scale streaming data analysis.

One reasonable approach for memory efficiency is to compress the FIs. There are two major classes of solutions: *closed* FIs, which have no frequent supersets with the same support (i.e., occurrence frequency), and *maximal* FIs, which are not contained in any other FI. The maximal FIs can restore the original FIs, though their supports are lost. In contrast, the closed FIs can exactly restore every FI and its support, though memory efficiency is bounded by limitation of lossless compression. For an intermediate solution between the closedness and maximality, various lossy condensed representations have been proposed in Xin et al. (2005), Cheng et al. (2006, 2008), Shin et al. (2014) and Quadrana et al. (2015).

A common approach among them is using an error parameter ϵ ($0 \leq \epsilon < 1$) to control the maximum error that can be included in the approximate supports. In the following, we call this approach *parameter-constrained* (PC) approximation. The value of ϵ is fixed in advance, however it is difficult to keep the initial value when concept drift emerges beyond user's expectation. Let us explain this situation using *MOA-IncMine* (Quadrana et al. 2015), which is the state-of-the-art PC method for FIM-SD. This method incrementally manages approximate closed FIs, called *semi-frequent closed itemsets* (SFCIs), for each segment of streaming transactions. Figure 1 shows the number of SFCIs stored at each segment of 1,000 transactions generated from a part of the Yahoo! Hadoop grid logs dataset.¹ Then, we observe that the SFCIs rapidly increase on the way to process an intermediate segment.

¹It consists of the HDFS audit logs on Hadoop file access that contain attributes like timestamp, IP address of the client, command types (ex., open, delete and so on) (Yahoo! Hadoop grid logs (from <https://webscope.sandbox.yahoo.com>)).

Fig. 1 Time-series on the number of SFCIs

When ϵ is less than or equal to 0.1, the process terminates due to out-of-memory exception (with 2GB JavaVM Heap size). Such sudden and intensive increase of memory consumption is a common drawback of PC mining in streaming data.

There is an alternative approximation approach, called *resource-constrained* (RC) approximation, working under resource (ex. memory and response time) constraint. The *Skip LC-SS* (Yamamoto et al. 2014) is a representative RC method which works in a constant $O(k)$ space and returns approximate FIs, where k is a size constant associated with the available memory resource. By setting a relevant k , the Skip LC-SS can avoid the out-of-memory; this represents a significant advantage. However, it involves another drawback: due to the combinatorial explosion of candidate FIs, the maximum error becomes extremely high in accordance with k , rendering some solutions in the output useless (Yamamoto et al. 2014).

In this study, we tackle the limitation of those PC and RC mining. First, to address the maximum error issue in RC mining, we introduce a lossy condensed representation of the FIs, called Δ -covered set. The notion of Δ -cover is regarded as an extension of closedness compression and allows the original FIs to be compressed while bounding the maximum error by an integer, Δ . Secondly, to address the memory consumption issue in PC mining, we introduce a hybrid approach of PC and RC approximation. As shown in Fig. 1, sudden and intensive memory consumption can be considered a temporary phenomenon. Indeed, normal PC approximation is sufficient except for within a brief period. From the viewpoint of memory efficiency, it is reasonable to dynamically switch between PC and RC approximations. We call this hybrid approach *PARASOL* (an acronym for Parameter- and Resource-constrained Approximation for Soft and Lazy mining).

In this paper, we propose the first PARASOL method to find a Δ -covered set of the FIs. The proposed method is based on two key techniques: *incremental intersection*, which is an incremental way to compute the closed itemsets (Borgelt et al. 2011; Yen et al. 2011), and *minimum entry deletion*, which is a space-saving technique for PC and RC approximation (Manku and Motwani 2002; Metwally et al. 2005; Yamamoto et al. 2014). We firstly show that the output obtained from these existing techniques composes a Δ -covered set of the FIs. Based on this finding, the proposed method controls the memory consumption and maximum error using an error parameter and takes an RC approximation using

a size constant k only in certain scenarios. Consequently, the following characteristics are exhibited:

On-the-fly manner: it can process any transaction that has L items in $O(k)$ space in almost $O(kL)$ time.

Anytime feature: it can monitor the maximum error, Δ , and return the output, T , at anytime.

Quality of output: for every FI α , T contains some superset β of α such that the difference between the frequencies of α and β is at most Δ .

We also propose a post-processing technique, called Δ -compression, which enables to reduce the size of the raw output for more concise solution. Then, we develop a novel data structure, called *weeping tree*, to efficiently carry out the three key operations (i.e., incremental intersection, minimal entry deletion, and Δ -compression). Finally, we empirically show that the proposed method outperforms the existing PC and RC methods when applied to various real datasets.

The remainder of the paper is organized as follows. Section 2 presents a brief review of the related works. Section 3 is a preliminary. Section 4 describes the baseline algorithm used to compute a Δ -covered set, and Section 5 describes PARASOL and Δ -compression. The data structure is discussed in Section 6. The experimental results are summarized in Section 7. Then, we conclude in Section 8.

2 Related works

Here, we clarify our contribution through the comparison with the related works. There are many previous works (Xin et al. 2005; Cheng et al. 2006, 2008; Song et al. 2007; Boley et al. 2009, 2010; Liu et al. 2012; Quadrana et al. 2015; Hu and Imielinski 2017) on PC mining that deal with lossy condensed representations of the FIs. However, most of those previous works, except for Song et al. (2007), Cheng et al. (2008), and Quadrana et al. (2015), are oriented to a transactional database that tolerates multiple scanning and allows us to assume a stable distribution of occurrences. It is infeasible to apply them for SD where concept drift can emerge.

Song et al. (2007) proposed an online PC approximation algorithm called *CLAIM* to compute *relaxed closed itemsets*. Like the notion of a Δ -covered set, their solution (i.e., relaxed closed itemsets) can bound the maximum error in the approximate support of any FI. In this sense, CLAIM has the anytime feature. However, its update process involves exhaustive computation including set enumeration and generate-and-test based search. In contrast, the proposed method always performs the update processing in $O(kL)$ time for any transaction of length L .

Cheng et al. (2008) also proposed an online PC approximation algorithm called *IncMine* to compute *semi-frequent closed itemsets* (SFCIs). IncMine incrementally maintains SCFIs for each segment (i.e., each set of transactions) in two steps: computing the SFCIs in the current segment followed by updating whole set of SFCIs based on the newly computed ones. IncMine utilizes various pruning techniques and efficient data structure to avoid redundant computation in the update process. To our knowledge, IncMine is one of the state-of-the-art PC methods for condensed representation mining. Indeed, IncMine has been implemented in a *massive online analysis* (MOA) platform which enables it to be

widely distributed (Quadrana et al. 2015). However, as shown in Fig. 1, a large number of SFCIs emerges temporarily, causing sudden and intensive memory consumption. In contrast, the proposed method based on PARASOL never issues the out-of-memory. Besides, it is unknown if the obtained SFCIs can restore the support of each FI within a bounded error.

StreamMining (Jin and Agrawal 2005) is a PC approximation method that seeks the FIs. Based on an error parameter, ϵ , it deletes itemsets that are not promising as they have supports less than or equal to $\epsilon \times i$ for each timestamp i . The accuracy of the solution (i.e., the maximum error in the approximate supports) can be directly controlled by ϵ . However, this involves a combinatorial explosion of candidates itemsets to be managed, causing huge memory consumption. In contrast, the proposed method focuses the solution on a Δ -covered set of the FIs. Then, the memory consumption becomes much smaller than the one of *StreamMining*.

CloStream (Yen et al. 2011) is a non-constrained (NC) online method that computes the closed FIs exactly. The update process is based on the incremental intersection that never requires set enumeration computation. However, the memory consumption increases as the whole stream becomes large. This is due to the limitation of closedness compression. In contrast, the proposed method can control the memory consumption using the space-saving technique based on *minimal-entry deletion* (Manku and Motwani 2002; Metwally et al. 2005; Yamamoto et al. 2014). On the other hand, the proposed method is incomplete for finding the closed itemsets. This is a drawback of the proposed method. Note that there exists the notion of strong closedness, called Δ -closed set (Boley et al. 2009). The Δ -closed set is defined as a characteristic subset of the closed FIs, capturing the *interesting* patterns. Then, the proposed method is complete for finding this Δ -closed set (i.e., any Δ -covered set contains the Δ -closed set), as detailed in the Appendices A and B.

Skip LC-SS (Yamamoto et al. 2014) is an RC method that seeks FIs based on minimal-entry deletion. Given a size constant k , it can process any transaction of length L in $O(k)$ space and $O(kL)$ time, keeping the top- k itemsets with respect to their approximate supports. However, Skip LC-SS must increment the maximum error Δ by one, whenever a transaction t_i that satisfies $2^{|t_i|} > k$ is processed; hence, it suffers from a rapid increase in Δ in accordance with k . By introducing the condensed representation (i.e., Δ -covered set), the proposed method enables to reduce the maximum error, as demonstrated in the experiment later.

Hence, our important contribution lies in proposing the first online PARASOL method that seeks a Δ -covered set of the FIs. It can process any transaction of length L in $O(k)$ space and almost in $O(kL)$ time, unlike to the existing online PC and NC methods (Manku and Motwani 2002; Karp and Shenker 2003; Jin and Agrawal 2005; Chi et al. 2004; Jiang and Gruenwald 2006; Song et al. 2007; Li et al. 2008; Cheng et al. 2008; Yen et al. 2011; Shin et al. 2014; Quadrana et al. 2015). Compared with the existing RC method (Yamamoto et al. 2014), our solution is more accurate in the sense that Δ is much smaller, while the support of every FI can be restored within an error bounded by Δ .

We also highlight the issue on data structure to efficiently perform the key operations of the proposed method. Especially, we focus on a binomial spanning tree, called *weeping tree*. Unlike the previously proposed data structures, such as prefix trees (Borgelt et al. 2011; Shin et al. 2014) and vertical format index (Yen et al. 2011), the weeping tree captures not only each itemset but its occurrence sequence in the transactions. As a result, the weeping tree exhibits several interesting features that are essential to prune redundant update process. To our knowledge, this is the first attempt to demonstrate the applicability of binomial spanning trees in the context of FIM-SD.

3 Notation and terminology

Let $I = \{1, 2, \dots, N\}$ be the universal set of items. Itemset t is a non-empty subset of I , i.e., $t \subset I$. Data stream $\mathcal{S}_n$ is the sequence of itemsets $\langle t_1, t_2, \dots, t_n \rangle$ for which $t_i \subset I$ for $i = 1, 2, \dots, n$ where n denotes the timestamp for which an output is requested by the user. Each t_i is called transaction at timestamp i . The number of items in t_i (i.e., the cardinality) is denoted as $|t_i|$ and is referred to as the *length* of t_i . L denotes the maximum length of the transactions in $\mathcal{S}_n$. For itemset $\alpha \subset I$ and timestamp i , $\text{tran}(\alpha, i)$ denotes the family of itemsets each of which includes α as a subset at timestamp i (i.e., $\text{tran}(\alpha, i) = \{t_j \mid \alpha \subseteq t_j, 1 \leq j \leq i\}$). Support $\text{sup}(\alpha, i)$ of itemset α at timestamp i is defined as $|\text{tran}(\alpha, i)|$. Given a minimum support threshold σ ($0 \leq \sigma \leq 1$), if $\text{sup}(\alpha, n) > \sigma n$, then α is frequent with respect to σ in $\mathcal{S}_n$. $\mathcal{F}_n$ denotes the family of FIs with respect to σ at timestamp n in $\mathcal{S}_n$, i.e., $\mathcal{F}_n = \{\alpha \subset I \mid \text{sup}(\alpha, n) > \sigma n\}$. FI α is *closed* if there is no α 's proper superset whose support is equal to α 's support in $\mathcal{F}_n$. On the other hand, FI α is *maximal* if there is no α 's proper superset in $\mathcal{F}_n$.

Next, we define two key notions to describe a condensed representation of the FIs as follows:

Definition 1 (Δ -cover) Let α and β be two itemsets. If $\alpha \subseteq \beta$ and $\text{sup}(\alpha, i) \leq \text{sup}(\beta, i) + \Delta$ for a non-negative integer Δ , then α is Δ -covered by β at timestamp i ; this state is denoted by $\alpha \preceq_{\Delta}^i \beta$.

Definition 2 (Δ -covered set) Let P and Q be two families of itemsets. If $\forall \alpha \in P \exists \beta \in Q$ such that $\alpha \preceq_{\Delta}^i \beta$ for a non-negative integer Δ , then Q is a Δ -covered set of P at timestamp i .

Example 1 Consider the family, P of four itemsets $\alpha_1, \alpha_2, \alpha_3$, and α_4 , shown in Fig. 2. For $Q_0 = \{\alpha_1, \alpha_3, \alpha_4\}$, $Q_1 = \{\alpha_3, \alpha_4\}$ and $Q_2 = \{\alpha_4\}$. Hence, Q_0, Q_1 and Q_2 are 0-covered, 1-covered and 2-covered sets of P , respectively. Note also that there exists another 1-covered set $Q'_1 = \{\alpha_1, \alpha_4\}$ of P .

A Δ -covered set is a generalization of the closed itemsets. The closed itemsets of the family $\mathcal{F}_n$ of FIs compose a 0-covered set of $\mathcal{F}_n$. Note also that each FI and its support can be recovered from a Δ -covered set within an error bounded by Δ ; in the following, this property is called Δ -deficiency. Let us consider again the 2-covered set Q_2 , in Example 1.

Fig. 2 Example of a Δ -covered set

By anti-monotonicity with respect to support, $\sup(\alpha_4, n) \leq \sup(\alpha_j, n)$ ($1 \leq j \leq 3$) holds for a timestamp n . Since Q_2 is a 2-covered set of P , $\alpha_4 \preceq_2^n \alpha_j$, so $\sup(\alpha_j, n) \leq \sup(\alpha_4, n) + 2$. Since both $3 \leq \sup(\alpha_j, n)$ and $\sup(\alpha_j, n) \leq 5$ hold, the maximum error is bounded by 2.

The objective is to find a Δ -covered set of $\mathcal{F}_n$ for a non-negative integer, Δ , while processing each transaction t_i ($1 \leq i \leq n$) only once. As explained before, PC mining and RC mining approaches for FIM-SD have been established. PC mining controls Δ according to an error parameter, ϵ , such that $\Delta \leq \epsilon \times i$ for each timestamp i ; however, in the worst case, an exponential number of candidate itemsets must be stored with this approach. Indeed, it is not feasible for any PC method to solve this problem in $O(k)$ space for a constant k .

Hence, we aim to design a novel method by integrating PC and RC mining approaches so that Δ is kept as small as possible based on a given value of k , which bounds the memory consumption. Herein, we present a baseline method to find a Δ -covered set of $\mathcal{F}_n$ for a given k in $O(kLn)$ time and $O(k)$ space, where L is the maximum length of transactions and n is the end timestamp.

4 Baseline algorithm for online Δ -covered set mining

The baseline algorithm is built on two key techniques: *incremental intersection* and *minimum entry deletion*, which are described in detail in this section.

Incremental intersection (Borgelt et al. 2011; Yen et al. 2011) is used for computing the closed itemsets. It is based on the following cumulative and incremental features of the closed itemsets. Let $\mathcal{C}_i$ be the family of closed itemsets in $\mathcal{S}_i$.

Theorem 1 (Borgelt et al. 2011; Yen et al. 2011) *Given $\mathcal{C}_{i-1}$ at timestamp $i - 1$ and transaction t_i at timestamp i , $\mathcal{C}_i$ is given as follows:*

$$\mathcal{C}_i = \emptyset \text{ for } i = 0$$

$$\mathcal{C}_i = \mathcal{C}_{i-1} \cup \{t_i\} \cup \{\beta \mid \beta = \alpha \cap t_i, \beta \neq \emptyset, \alpha \in \mathcal{C}_{i-1}\} \text{ for } i \geq 1$$

Proof This recursive relationship has been revealed in the literature (Borgelt et al. 2011; Yen et al. 2011). We also give a brief sketch of the proof in Appendices A and B. $\square$

Theorem 1 ensures that $\mathcal{C}_i$ can be computed from the intersection of each itemset in $\mathcal{C}_{i-1}$ with t_i .

Example 2 Let $I = \{1, 2, 3, 4, 5\}$ and the stream $\mathcal{S}_4^1 = \langle t_1, t_2, t_3, t_4 \rangle$ where $t_i = I - \{i\}$ for each i ($1 \leq i \leq 4$). Figure 3 illustrates how $\mathcal{C}_i$ is incrementally generated until $i = 3$. $\mathcal{C}_1$ consists of the first transaction. $\mathcal{C}_2$ newly contains α_2 and α_3 , which correspond to the second transaction and its intersection with α_1 , respectively. Finally, $\mathcal{C}_3$ and $\mathcal{C}_4$ consist of the seven and fifteen closed itemsets, respectively.

An incremental intersection never unfolds a transaction to intermediate subsets. We just intersect each stored itemset by the transaction. Then, the update operation never involves the subset enumeration of itemsets. Besides, we do not need to check if the generated one is a closed itemset by Theorem 1. In this sense, incremental intersection can simply but efficiently compute only the closed itemsets. On the other hand, lossless compression based on closedness cannot necessarily control an exponential increase of the closed itemsets; in the worst case, $\Omega(2^L)$ closed itemsets can be generated. In Example 2, there is a total of

Fig. 3 Example of incremental intersection

fifteen closed itemsets from $\mathcal{S}_4^1$, which is almost the same as the 2^L closed itemsets that are attained for $L = 4$.

Hence, it is required to embed a space-saving technique into incremental intersection. In this paper, we introduce the notion of *minimum entry deletion* which has been proposed in RC mining (Metwally et al. 2005; Yamamoto et al. 2014). Given a size constant, k , minimal entry deletion is performed when the number of stored itemsets exceeds k to keep only the top- k itemsets; in other words, the itemset with the lowest approximate support must be deleted iteratively as long as the number of stored itemsets is greater than k . The maximum value in the approximate supports of the deleted itemsets is then maintained as the maximum error, Δ . It is reasonable for space-saving to embed the same scheme into the update process by incremental intersection.

The baseline algorithm manages two types of information for each stored itemset: its approximate support count and maximum error count. They are represented as a tuple entry of the form $\langle \alpha, c, \Delta \rangle$, which represents the stored itemset, the approximate support count, and maximum error count, respectively. In the following, we call c and Δ the *frequency count* and *error count* of α , respectively. Besides, the itemset, frequency count, and error count of an entry, e , are often denoted by α_e , c_e , and Δ_e , respectively.

Example 3 Consider $\mathcal{S}_4^1$ again. Let $k = 3$. The number of $\mathcal{C}_3$ is beyond $k = 3$ at timestamp $i = 3$. As shown in Fig. 4, we remove four closed itemsets in order of increasing frequency count; the highest count of those that are removed is two. The next transaction, t_4 , is stored in the entry $\langle t_4, 3, 2 \rangle$, meaning that the support of t_4 is between one and three (i.e., $1 \leq \text{sup}(t_4, 4) \leq 3$).

In the following, T_i denotes the collection of entries used at timestamp i , $k(i)$ denotes the number of entries in T_i , and $\Delta(i)$ denotes the maximum error at timestamp i . A *minimum entry* in T_i is an entry whose frequency count is the lowest of those in T_i . Then, the baseline algorithm is described in Algorithm 1. The updating process is composed of entry addition and entry deletion for each timestamp. The addition operation is carried out by the function $\text{intersect}(T_{i-1}, t_i)$, which performs the incremental intersection. The deletion operation is realized by the function $\text{delete}(T_i)$, which performs the minimum entry deletion.

Algorithm 1 The baseline algorithm.

Input: a size constant (k), minimum support threshold (σ) and data stream ($\mathcal{S}_n = \langle t_1, t_2, \dots, t_n \rangle$).

Output: a $\Delta(n)$ -covered set of the FIs with respect to σ .

```

1: set  $i$  as 1 ( $i := 1$ )                                ▷  $i$  is the current time
2:  $\Delta(i) := 0$                                           ▷  $\Delta(i)$  is the maximum error at time  $i$ 
3: initialize  $T_0$ 
4: while  $i \leq n$  do read  $t_i$ 
5:    $T_i := \text{intersect}(T_{i-1}, t_i)$ 
6:    $\Delta(i) := \text{delete}(T_i)$ 
7:    $i := i + 1$ 
8: end while
9: for each entry  $e$  in  $T_n$  such that  $c_e > \sigma n$  do
10:   output  $e$                                           ▷ composing a  $\Delta(n)$ -covered set
11: end for

12: function INTERSECT( $T_i, t_{i+1}$ )
13:   initialize  $C$                                       ▷  $C$ : the collection of candidate entries
14:    $e := \text{get}(t_{i+1}, T_i)$                           ▷ getting the entry  $e$  for  $t_{i+1}$  in  $T_i$ 
15:   if  $e$  is null then
16:     add the entry  $\langle t_{i+1}, \Delta(i), \Delta(i) \rangle$  to  $T_i$ 
17:   end if
18:   for each entry  $u$  in  $T_i$  such that  $\alpha_u \cap t_{i+1} \neq \emptyset$  do
19:      $v := \text{get}(\alpha_u \cap t_{i+1}, C)$ 
20:     if  $v$  is null then
21:       add  $\langle \alpha_u \cap t_{i+1}, c_u + 1, \Delta_u \rangle$  to  $C$ 
22:     else if  $c_v < c_u + 1$  then
23:        $c_v := c_u + 1, \Delta_v := \Delta_u$ 
24:     end if
25:   end for
26:   for each entry  $r$  in  $C$  do
27:      $e := \text{get}(\alpha_r, T_i)$                           ▷ getting the entry  $e$  for  $\alpha_r$ 
28:     if  $e$  is null then                                ▷ no entry for  $\alpha_r$ 
29:       add the entry  $r$  to  $T_i$ 
30:     else
31:       replace the entry  $e$  with the entry  $r$  in  $T_i$ 
32:     end if
33:   end for
34:   return  $T_i$                                           ▷ corresponding to the next entry table
35: end function

36: function DELETE( $T_i$ )
37:   while  $k < k(i)$  do
38:      $m := \text{getMin}(T_i)$                                 ▷  $m$  is a minimum entry in  $T_i$ 
39:      $\Delta(i) := c_m$                                     ▷  $c_m$  is the minimum frequency count in  $T_i$ 
40:      $\text{deleteMin}(T_i)$                                   ▷ delete a minimum entry in  $T_i$ 
41:   end while
42:   return  $\Delta(i)$ 
43: end function

```

Fig. 4 Example of minimal entry deletion

Example 4 Consider T_4 again as shown in Fig. 4. T_4 corresponds to the output of Line 5 in Algorithm 1 at $i = 4$ for the input stream S_4^1 and $k = 3$. By the deletion process of Line 6, $\Delta(4)$ becomes three. Now, we assume that the top three entries in T_4 remain as shown in Fig. 5. Next, let the transaction arriving at $i = 5$ be $t_5 = \{1, 3, 5\}$. Then, by the addition process, we first add the entry $\langle t_5, 3, 3 \rangle$ to T_4 , since there is no entry for t_5 in T_4 . After that, the intersection of each stored itemset in T_4 with t_5 is computed and stored in C . In total, C consists of three entries. Then, T_5 is obtained by updating T_4 with C by way of Lines 26–33 in the intersect function. Finally, the minimum entry, $\langle \{2, 5\}, 3, 0 \rangle$, is deleted from T_5 to ensure that $k(5) \leq k$.

Fig. 5 Example of the baseline algorithm

Next, we clarify the quality of the output by the baseline algorithm.

Theorem 2 *Algorithm 1 outputs a $\Delta(n)$ -covered set of the FIs wrt σ , provided that $\Delta(n) \leq \sigma n$.*

This implies that for every FI, α , T_n must contain an entry, e , in which α is $\Delta(n)$ -covered by α_e . To demonstrate this feature, we introduce the notion of a *representative entry*.

Definition 3 Let α be an itemset. Given T_i , T_i^α is used to denote the set of entries in T_i that contains such an itemset β that $\alpha \subseteq \beta$. If T_i^α is not empty, a *representative entry*, r , for α is defined as an entry that has the maximum frequency count of those in T_i^α (i.e., $r = \operatorname{argmax}_{e \in T_i^\alpha} (c_e)$).

Then, for every FI, α , and a representative entry, r , of α , we claim that α is $\Delta(n)$ -covered by α_r . This claim can be explained as shown in Fig. 5: Since $\Delta(5) = 3$, we set $\sigma = 0.6$ to ensure that $\Delta(5) \leq \sigma \times 5$. There exist five FIs $\{1\}, \{3\}, \{5\}, \{1, 5\}, \{3, 5\}$ wrt σ in $\mathcal{S}_5^1$ (that is, $\mathcal{S}_5^1 = \mathcal{S}_4^1 \circ \langle t_5 = \{1, 3, 5\} \rangle$). Let α be the FI $\{3, 5\}$. T_5^α uniquely contains the entry $r = \langle \{1, 3, 5\}, 4, 3 \rangle$. Hence, r is the representative entry of α . Note that $\alpha = \{3, 5\}$ is 1-covered by $\alpha_r = \{1, 3, 5\}$, since $\sup(\{1, 3, 5\}, 5) = 3$ and $\sup(\{3, 5\}, 5) = 4$.

First, we prove the following proposition and two lemmas.

Proposition 1 *Let e be an entry in T_i . Then, e is a representative entry for α_e .*

Proof. Suppose that e is not a representative entry. Then, there exists another entry, $e' \in T_i$, such that $\alpha_e \subseteq \alpha_{e'}$ and $c_e < c_{e'}$. This contradicts the anti-monotonicity in T_i , which is proved by using the mathematical induction, following the below argument. First, since T_1 contains only one entry, T_1 satisfies the anti-monotonicity property. Assume that T_i satisfies this property (i.e., for every two entries, e and e' , if $\alpha_e \subseteq \alpha_{e'}$ then $c_e \geq c_{e'}$ holds). Then, by the cumulative feature of the incremental intersection, the output of the function $\operatorname{intersect}(T_i, t_{i+1})$ also exhibits anti-monotonicity. Moreover, this property is preserved by the deletion process. Therefore, T_{i+1} also exhibits the anti-monotonicity. $\square$

Lemma 1 *Let α be an itemset. Given T_i , if there exists a representative entry r for α in T_i , then it holds that*

$$c_r - \Delta_r \leq \sup(\alpha, i) \leq c_r.$$

Proof We use the mathematical induction. T_1 consists of only one entry, $\langle t_1, 1, 0 \rangle$. Since $k > 0$, this entry is not deleted. Accordingly, this lemma is true in the case of T_1 . Assume that T_i satisfies the claim of the lemma. Let α be an itemset whose representative entry, r_{i+1} , exists in T_{i+1} . Now, we consider two cases for the existence of a representative entry r_i for α in T_i .

Case 1: r_i exists. Based on the assumption, we have $c_{r_i} - \Delta_{r_i} \leq \sup(\alpha, i) \leq c_{r_i}$. If $\alpha \not\subseteq t_{i+1}$, then $c_{r_{i+1}} = c_{r_i}$ because $T_i^\alpha \supseteq T_{i+1}^\alpha$ and $T_{i+1}^\alpha \neq \emptyset$ (i.e., the maximum frequency count in T_i^α is preserved in T_{i+1}^α by the minimum entry deletion); thus, since $\sup(\alpha, i+1) = \sup(\alpha, i)$, $c_{r_{i+1}} - \Delta_{r_i} \leq \sup(\alpha, i+1) \leq c_{r_{i+1}}$. Otherwise, since $\alpha \subseteq t_{i+1}$, $c_{r_{i+1}} = c_{r_i} + 1$ holds because T_{i+1} contains the entry for $\alpha_{r_i} \cap t_{i+1}$ which has a frequency count of $c_{r_i} + 1$. Since $\sup(\alpha, i+1) = \sup(\alpha, i) + 1$,

$c_{r_{i+1}} - \Delta_{r_i} \leq \sup(\alpha, i + 1) \leq c_{r_{i+1}}$. Finally, since $\Delta_{r_i} \leq \Delta_{r_{i+1}}$, we have $c_{r_{i+1}} - \Delta_{r_{i+1}} \leq \sup(\alpha, i + 1) \leq c_{r_{i+1}}$.

Case 2: r_i does not exist. This means that α or its supersets never appear or have been deleted before the time i . Hence, $\sup(\alpha, i)$ is at most $\Delta(i)$, because $\Delta(i)$ is the upper bound of the frequencies of the deleted itemsets. Since r_{i+1} exists but r_i does not, r_{i+1} corresponds to $\langle t_{i+1}, 1 + \Delta(i), \Delta(i) \rangle$. Hence, $\alpha \subseteq t_{i+1}$ holds. Accordingly, $1 \leq \sup(\alpha, i + 1) \leq 1 + \Delta(i)$ holds. Since $c_{r_{i+1}} = 1 + \Delta(i)$ and $\Delta_{r_{i+1}} = \Delta(i)$, it holds that $c_{r_{i+1}} - \Delta_{r_{i+1}} \leq \sup(\alpha, i + 1) \leq c_{r_{i+1}}$.

In both cases, T_{i+1} also satisfies the claim of the lemma. □

Lemma 2 For every itemset, α , such that $\sup(\alpha, i) > \Delta(i)$, there exists a representative entry $r \in T_i$ for α .

Proof We also use the mathematical induction. The lemma is obviously true in the case of T_1 since $\Delta(1) = 0$ holds. Assume that T_i satisfies the claim of this lemma; we show the induction step by contradiction. Assume that $\exists \alpha$ such that $\sup(\alpha, i + 1) > \Delta(i + 1)$ but there is no representative entry for α in T_{i+1} . We can consider two cases: whether or not a representative entry, r , has existed in T_{i+1} before the deletion operation (i.e., in the intermediate T_{i+1} at the moment when the incremental intersection has been completed, as depicted in the middle T_{i+1} in Fig. 6).

Case 1: such an r exists in T_{i+1} but was deleted. By Lemma 1, we have $c_r - \Delta_r \leq \sup(\alpha, i + 1) \leq c_r$. Since r was deleted, $c_r \leq \Delta(i + 1)$ should hold. Thus, $\sup(\alpha, i + 1) \leq \Delta(i + 1)$ also holds. However, this is a contradiction.

Case 2: such an r does not exist in T_{i+1} . Here, T_{i+1} captures the entry table before the deletion operation. Thus, $T_i^\alpha \subseteq T_{i+1}^\alpha$ holds. Since there is no representative entry for α in T_{i+1} , it holds that $T_{i+1}^\alpha = \emptyset$ and $T_i^\alpha = \emptyset$. Note that T_{i+1} should contain the entry for t_{i+1} . Hence, $\alpha \not\subseteq t_{i+1}$ because $T_{i+1}^\alpha = \emptyset$. Therefore, $\sup(\alpha, i) =$

Fig. 6 Sketch of updating T_i to T_{i+1} with t_{i+1}

$\sup(\alpha, i + 1)$. Since $\sup(\alpha, i + 1) > \Delta(i + 1)$ and $\Delta(i + 1) \geq \Delta(i)$, it should hold that $\sup(\alpha, i) > \Delta(i)$. Then, by the assumption for T_i , there should exist a representative entry for α in T_i . However, this contradicts the conclusion that $T_i^\alpha = \emptyset$.

In both cases, there is a contradiction. Therefore, for every α such that $\sup(\alpha, i + 1) > \Delta(i + 1)$, there exists a representative entry for α in T_{i+1} . $\square$

Now, we prove Theorem 2 as follows:

Proof of Theorem 2 Suppose that the condition of σ (i.e., $\Delta(n) \leq \sigma n$) is satisfied. Then, by Lemma 2, for every FI α wrt σ , there exists a representative entry, r , in T_n for α . By Lemma 1, $c_r - \Delta_r \leq \sup(\alpha, n) \leq c_r$ holds. By Proposition 1, r is also a representative entry for α_r . Thus, we have $c_r - \Delta_r \leq \sup(\alpha_r, n) \leq c_r$. Hence, $\sup(\alpha, n) \leq \sup(\alpha_r, n) + \Delta_r$ holds. Since $\alpha_r \supseteq \alpha$, α is Δ_r -covered by α_r . Since $\Delta_r \leq \Delta(n)$, we can claim that for every FI α , there exists an entry, r , such that α is $\Delta(n)$ -covered by α_r . Note that $c_r > \sigma n$ holds because $\sup(\alpha, n) > \sigma n$. Hence, r is contained in the output because of Lines 9–11 in Algorithm 1. $\square$

Next, we clarify the complexity of the baseline algorithm. Incremental intersection generates at most $k(i) + 1$ new entries for the intersections of the entries in T_i with t_{i+1} as well as t_{i+1} itself; these new entries are added to C in Lines 16 and 21 of the *intersect* function. Thus, the total number of stored entries is at most $2 \times k(i) + 1$, and $k(i)$ is always bounded by the size constant k . Hence, the complexity of the baseline algorithm can be described as follows:

Theorem 3 For a size constant k and the maximum transaction length L , Algorithm 1 processes every transaction in $O(kL + k \log k)$ time and $O(k)$ space.

Proof First, the number of stored entries is at most $2k + 1$ for every time point. Thus, the space required to store them is $O(k)$. In addition, it takes at most $O(2L)$ time to compute the intersection of a stored itemset with a transaction. Thus, the incremental intersection is performed in $O(kL)$ time. It takes at most $O(\log k)$ time to delete a minimum entry and add a new entry by using a heap structure. Hence, the total time required to update the table is at most $O(kL + k \log k)$. $\square$

Based on Theorems 2 and 3, Algorithm 1 can extract a Δ -covered set of the FIs wrt $\sigma \geq \frac{\Delta}{n}$ for an integer Δ and can process each transaction in $O(kL)$ time and $O(k)$ space for a size constant k ($k < 2^L$). Therefore, Algorithm 1 satisfies the three characteristics (i.e., on-the-fly manner, anytime feature, and quality of output) presented in Section 1.

5 Parasol: a hybrid of PC and RC approximation

We observe in Fig. 1 that a sudden and intensive memory consumption may occur as a temporary phenomenon. In other words, the PC approximation is feasible unless all of the memory is consumed during this short period. Hence, it is reasonable to take a hybrid of PC and RC approximation; the memory consumption is generally controlled by using an error parameter ϵ and an RC approximation using a size constant k is taken only when the number of used entries exceeds k . We call this hybrid approximation approach *PARASOL*.

(an acronym for Parameter- and Resource-constrained Approximation for Soft and Lazy mining).

Our proposed PARASOL method can be realized just by replacing the *delete* function in Algorithm 1 with the following function:

```

1: function PARASOL_DELETE( $T_i$ )
2:    $m := \text{getMin}(T_i)$   $\triangleright m$  is a minimum entry in  $T_i$ 
3:   while  $k < k(i)$  or  $c_m \leq \epsilon \times i$  do
4:      $\Delta(i) := c_m$   $\triangleright c_m$  is the minimum frequency count in  $T_i$ 
5:      $\text{deleteMin}(T_i)$   $\triangleright$  delete a minimum entry in  $T_i$ 
6:      $m := \text{getMin}(T_i)$ 
7:   end while
8:   return  $\Delta(i)$ 
9: end function

```

The main difference appears in the condition of Line 3. Unless sudden and intensive memory consumption occurs (i.e., $k < k(i)$ holds), the proposed PARASOL method performs the minimal-entry deletion in the PC approximation manner. For this ordinary scenario, it always ensures $\Delta(i) \leq \epsilon \times i$. Otherwise, it performs the minimal-entry deletion in the RC approximation manner due to the constant space constraint. For this exceptional scenario, it allows $\Delta(i)$ to exceed $\epsilon \times i$. Since sudden and intensive memory consumption is a temporal phenomenon, the RC approximation is only required for a short period. Therefore, the error ratio, $\frac{\Delta(i)}{i}$ gradually converges to ϵ after this short period. This *self-sustained recovery* of the error ratio is a unique and advantageous characteristic of the proposed PARASOL method, and will be empirically demonstrated in experiment later.

5.1 Furthermore improvement

Note that the solution to our problem setting is not unique: there can exist many Δ -covered sets of $\mathcal{F}$. Then, it is reasonable to extract a concise Δ -covered set. Here, we propose a fast post-processing technique to reduce the original output to a more concise one based on the maximum error, $\Delta(n)$.

First, we explain the intuition using T_5 (the right-most table in Fig. 5) in Example 4. Since $\Delta(5) = 3$, the three itemsets stored in T_5 compose a 3-covered set of the FIs. Interestingly, even if $\{1, 5\}$ was deleted from T_5 , the two remaining itemsets could compose a 3-covered set. Let us consider the entry $r = \langle \{1, 5\}, 4, 0 \rangle$, and let α be an itemset whose representative entry is r . By Lemma 1, the support of α is at most c_r . Since c_r is the same as the frequency count for $\{1, 3, 5\}$, α is also 3-covered by $\{1, 3, 5\}$. Therefore, there is no need to keep r in T_5 for restoring the support of α when $\Delta = 3$.

There is another choice point to delete the entry for $\{5\}$ from T_5 . In exchange for deletion, we need to update the entry for $\{1, 5\}$ by $\langle \{1, 5\}, 5, 1 \rangle$ (i.e., incrementing the frequency and error counts by one). In this way, the support of $\{5\}$ can be restored from the entry for $\{1, 5\}$.

In general, it is infeasible to identify such itemsets to be deleted from an original $\Delta(n)$ -covered set so that the remaining set keeps the $\Delta(n)$ -deficiency. However, some of them can be simply detected based on the following corollary:

Corollary 1 *For every entry $e \in T_n$, if there exists $r \in T_n$ such that $\alpha_e \subsetneq \alpha_r$ and $c_e \leq c_r - \Delta_r + \Delta(n)$, then the itemsets of $T_n - \{e\}$ compose a $\Delta(n)$ -covered set of the FIs wrt σ , provided that $\Delta(n) \leq \sigma n$.*

Proof Let e be an entry in T_n and α an arbitrary itemset whose representative entry is e . Suppose that there exists an entry r such that $\alpha_e \subsetneq \alpha_r$ and $c_e \leq c_r - \Delta_r + \Delta(n)$. We claim that α is $\Delta(n)$ -covered by α_r . By Lemma 1, it holds that $\sup(\alpha, n) \leq c_e$, followed by $\sup(\alpha, n) \leq c_r - \Delta_r + \Delta(n)$. Since $c_r - \Delta_r \leq \sup(\alpha_r, n)$, we have $\sup(\alpha, n) \leq \sup(\alpha_r, n) + \Delta(n)$. Since $\alpha \subseteq \alpha_e$ and $\alpha_e \subsetneq \alpha_r$, we also have $\alpha \subsetneq \alpha_r$. Hence, α is $\Delta(n)$ -covered by α_r . The claim implies that even if e was removed, the support of α can be recovered using r , while $\Delta(n)$ remains as it is. $\square$

Now, we propose a post-processing technique, called Δ -compression, which incrementally deletes the entry from T_n according to Corollary 1, as described in Algorithm 2. In exchange for deleting e , Δ_r is updated by increasing it by the difference between c_e and c_r , and c_r is updated to c_e in Lines 3 and 4.

Algorithm 2 Δ -compression

```

1: function COMPRESS( $T_n$ )
2:   while  $\exists e \in T_n$  s.t.  $\exists r \in T_n$  s.t.  $\alpha_e \subsetneq \alpha_r$  and  $c_e \leq c_r - \Delta_r + \Delta(n)$  do
3:      $\Delta_r := \Delta_r + c_e - c_r$ 
4:      $c_r := c_e$ 
5:     delete  $e$  from  $T_n$ 
6:   end while
7: end function
  
```

Note that the output produced by Algorithm 2 does not necessarily compose a minimal $\Delta(n)$ -covered set of the FIs. The counter example is derived by using T_5 of Example 4. After applying Algorithm 2, T_5 contains two itemsets $\{1, 3, 5\}$ and $\{5\}$ (alternatively, $\{1, 3, 5\}$ and $\{1, 5\}$). However, only $\{1, 3, 5\}$ is enough to compose a 3-covered set of the FIs (σn is now assumed to be greater than or equal to 3). Thus, the output is not a minimal 3-covered set. A simple implementation of Algorithm 2 is that for each entry e , we search T_n for a corresponding r . This can be achieved in $O(k^2)$ time, since $|k(n)| \leq k$.

6 Data structure

Here, we address the issue of the data structure that is needed to efficiently realize the three key operations of the proposed algorithm: incremental intersection, deletion, and Δ -compression. Of these, incremental intersection incurs the majority of the computational cost. This operation imposes the traverse of every entry, e , to compute its intersection with a transaction, t_i , for each timestamp i . This computation is often redundant: for example, if α_e has no common items with t_i , it is redundant to compute the intersection of α_e with t_i .

Based on this observation, Yen et al. (2011) proposed an indexing data structure, called *cid_list*, corresponding to the vertical format of the stored itemsets: for each item, x , *cid_list*(x) maintains the indexes of the entries corresponding to the itemsets that contain x . Using *cid_list*, we can focus only on the entries whose indexes are contained in $\bigcup_{x \in t_i} \text{cid_list}(x)$ and compute their intersections with t_i . However, the computational cost of updating *cid_list* is relatively high: the entire *cid_list* is dynamically changed by addition and deletion operations. This overhead becomes especially high for *dense* datasets since most itemsets stored in T_i have some of the same items as t_i .

Borgelt et al. (2011) proposed a fast two-pass FIM method, called *ISTA*, based on incremental intersection. In this implementation, the prefix tree (as well as patricia) was

introduced to efficiently maintain T_i and perform the incremental intersection. Although it is reasonable to represent T_i with such a concise data structure, it is not directly applicable in the one-pass approximation setting that is used here. For example, we cannot use the item support as static information, while it is available in the transactional database that allows multi-pass scanning. This information is crucial to constructing a compact trie by sorting items in a pre-processing step. Note that the trie size can be directly affected by the order of (sorted) transactions. Indeed, ISTA constructs the trie with 306 nodes by treating it as a retail problem (No. 3 in Table 1), compared to 244,938 nodes when the pre-processing technique is not applied. Besides, in the context of the SD that emerges concept drift, it is not appropriate to assume a static distribution.

Thus, it is necessary to design a more suitable data structure for our proposed algorithm, which enables to prune redundant computations in incremental intersections, as well as quickly access both the minimum entries and Δ -covered entries required for PARASOL deletion and Δ -compression, respectively.

6.1 Weeping tree

Now, we propose a variation of the *binomial spanning tree* (Johnsson and Ho 1989; Chang 2005), called *weeping tree*, in which a collection of entries, T_n , can be represented in a *binary n -cube* as follows: Let e be an entry in T_n . By Theorem 1, α_e corresponds to the intersection of a certain set S of transactions. This set can be represented as a binary address $(x_1, x_2, \dots, x_n)$ where each x_j is one if S contains t_j and zero otherwise. Every α_e has its own binary address. Thus, T_n can be represented as a set of binary addresses, denoted by $V(n)$, each of which identifies α_e for an entry $e \in T_n$. Each binary address can be described by an integer, $x = \sum_{j=1}^n (x_j \times 2^{n-j})$. Then, $V(n)$ corresponds to a subset of $\{1, \dots, 2^n - 1\}$.

Let x and y be two integers ($1 \leq x, y \leq 2^n - 1$) with the addresses $(x_1, x_2, \dots, x_n)$ and $(y_1, y_2, \dots, y_n)$, respectively. $p(x)$ is the position in x that satisfies two conditions: $x_{p(x)} = 1$ and $x_j = 0$ for each j ($p(x) + 1 \leq j \leq n$). In other words, $p(x)$ is the least significant set bit of x . We say that x covers y if $y_j = x_j$ for each j where $1 \leq j \leq p(x)$. For example, if $x = 12$ and $y = 15$ in 4-cube with addresses (1100) and (1111) then, $p(x) = 2$, $y_1 = x_1$, and $y_2 = x_2$ so x covers y . We assume that zero covers every integer.

Now, we define the binomial spanning tree of $V(n)$, following the notion in the literature (Chang 2005).

Table 1 Characteristics of the used datasets

No	Datasets	n	$ I $	L	L_{ave}
1	earthquake	16,764	1,228	73	2.5
2	weblog	19,465	9,958	106	18.2
3	retail	88,162	16,469	76	10.3
4	accidents	340,183	468	51	33.8
5	chess	3,196	75	37	37
6	connect	67,557	129	43	43
7	kosarak	990,002	41,270	2,498	8.1
8	pumsb	49,046	2,112	74	73.6
9	mushroom	8,124	119	23	23
10	hadoop	3,655,760	1,941,291	1,000	30.6

Definition 4 (Binomial spanning tree) Let $V(n)$ be a subset of $\{1, \dots, 2^n - 1\}$, x be an integer such that $0 \leq x \leq 2^n - 1$, and $C(x)$ be the set of integers in $V(n)$ each of which is covered by x . The binomial spanning tree of $V(n)$ is the tree in which the root node, r , is zero and the other nodes are $V(n)$. The children of each node, x , correspond to the following set:

$$\{y \mid y \in C(x) \text{ and } \nexists y' \in C(x) \text{ s.t. } y \neq y' \text{ and } y \in C(y')\}.$$

The siblings $y^{(1)}, y^{(2)}, \dots, y^{(m)}$ are sorted in the descending order (i.e., $y^{(j)}$ is the left sibling of $y^{(j+1)}$). We say that a non-root node, y , is a *descendant* of node x if $y \in C(x)$, a *precursor* of x if $y \notin C(x)$ and $y > x$, and a *successor* of x if $y < x$. The precursors and successors of x are denoted at $P(x)$ and $S(x)$, respectively.

The *weeping tree* at time n , denoted by $W(n)$, is the binomial spanning tree of $V(n)$ obtained by associating each node, x , with its corresponding entry e (i.e., α_e is the intersection of the transactions indicated by the address x).

Example 5 Consider again the stream $\mathcal{S}_4^1$ in Example 2. Let ϵ and k be 0.2 and 15, respectively. PARASOL uses the 15 entries in $T(4)$ for all closed itemsets (i.e., $V(4) = \{1, 2, \dots, 15\}$). The corresponding weeping tree, $W(4)$, is described in Fig. 7. Each node, x , is associated with its own entry, e . For example, the node $x = 6$ (0110) corresponds to the entry for $\alpha = t_2 \cap t_3$, i.e., $\alpha = \{1, 4, 5\}$. Note also that $C(6) = \{7\}$, $P(6) = \{8, 9, \dots, 15\}$ and $S(6) = \{1, 2, \dots, 5\}$.

One crucial feature of the weeping tree is that it captures inclusion relationships among the stored itemsets in T_n .

Fig. 7 The weeping tree $W(4)$ wrt $\mathcal{S}_4^2$

Proposition 2 *Let two nodes x and y be associated with two entries e_x and e_y . If x covers y , then $\alpha_{e_y} \subseteq \alpha_{e_x}$*

Proof The address of y can be written as $(x_1, \dots, x_{p(x)}, y_{p(x)+1}, \dots, y_n)$, since x covers y . Accordingly, α_{e_y} is written as $(\bigcap_{x_j=1, 1 \leq j \leq p(x)} t_j) \cap I$ where $I = \bigcap_{y_j=1, p(x)+1 \leq j \leq n} t_j$. Since $p(x)$ is the position of the least significant set bit, $\alpha_{e_x} = \bigcap_{x_j=1, 1 \leq j \leq p(x)} t_j$ holds, followed by $\alpha_{e_y} = \alpha_{e_x} \cap I$. $\square$

Proposition 2 has three useful implications. First, it is applicable for pruning the intersection computations. Suppose that during the updating process at time i , an entry, e , is found such that $\alpha_e \subseteq t_i$. Since every descendant of e must be included in t_i , it is not necessary to compute the intersections for these descendants. Proposition 2 also implies that every minimum entry must be located in the shallowest layer in the tree due to the anti-monotonicity of T_i . This feature is useful for the minimum entry deletion (in practice, it is reasonable to use min-heap for the shallowest layer).

Finally, Proposition 2 is applicable to the pairwise checking involved in Δ -compression. Suppose that we found a parent entry, e_p , and its child entry, e_c such that $C_{e_c} \leq C_{e_p} - \Delta_{e_p} + \Delta(n)$. Then, α_{e_c} must be $\Delta(n)$ -covered by α_{e_p} according to Corollary 1. Hence, a quick check can be done to determine if each child is $\Delta(n)$ -covered by its parent. Note that a brute-force approach requires $O(k^2)$ time for Δ -compression while the quick pairwise checking can be completed in $O(k)$ time. Thus, it is useful as a pre-processing step preceding Δ -compression.

6.2 Weeping tree updating

Here, we explain how to incrementally update $W(i)$ at each time i . Suppose that an itemset, α , is newly stored in $W(i+1)$. An address $(x_1, \dots, x_i, 1)$ is then assigned to α , where $(x_1, \dots, x_i)$ is the address of the node, r , in $W(i)$ that corresponds to a representative entry of α if such an r exists; if there is no such node, r is the root node. Thus, the entry for α is newly located as a child of node r .

Example 6 Consider Example 5 again. Let α be the itemset $\{2, 5\}$ that is newly added in $W(4)$. There exists a representative entry $r = \langle \{2, 4, 5\}, 2, 0 \rangle$, for α (see Fig. 4) in $T(3)$. Since r is given the address (101) at time $i = 3$, the address of α becomes (1011), and the entry for α is located as a child of r .

We can interpret the meaning of the address assigned to each entry. Let e be an entry of node x with the address $(x_1, \dots, x_n)$. We denote the least and greatest significant bit sets of x as $p(x)$ and $q(x)$, respectively. Then, e is written as $\langle \alpha, \Delta(q(x) - 1) + B, \Delta(q(x) - 1) \rangle$, where $\alpha = \bigcap_{x_j=1, q(x) \leq j \leq p(x)} t_j$, $\Delta(q(x) - 1)$ is the maximum error at time $q(x) - 1$ and B is the bit count of x (See Fig. 8). This observation leads to the following proposition:

Proposition 3 *Consider two nodes, x and y , of entries e_x and e_y , respectively, in a weeping tree. If $\alpha_{e_y} \subseteq \alpha_{e_x}$, then y is either a descendant or a precursor of x .*

Proof We derive a contradiction in the case that $\alpha_{e_y} \subseteq \alpha_{e_x}$ and $y < x$. Let v be the address obtained by the bitwise OR operation between x and y . Since $y < x$ and $x \leq v$, $y < v$. We

Fig. 8 Relation between address and entry

write v as $(v_1, \dots, v_n)$ and denote by α_v the intersection $\bigcap_{v_j=1, 1 \leq j \leq n} t_j$. Since $\alpha_{e_y} \subseteq \alpha_{e_x}$, we have $\alpha_v = \alpha_{e_y}$. Thus, node v should not appear in the tree, since its duplicate never occurs in T_n . Without losing generality, this implies that v has been deleted at some time, which is referred to as time m . Accordingly, the tree never contains such a node, u , with an address $(u_1, \dots, u_n)$ such that $q(u) = q(v)$; moreover, the bit count of the m -prefix $(u_1, \dots, u_m)$ is lower than the bit count of the m -prefix $(v_1, \dots, v_m)$. This is because every node with such an address has been deleted at time m , along with v (i.e., u has a lower frequency count than v). Hence, no node can have an address in which the m -prefix matches $(u_1, \dots, u_m)$ or $(v_1, \dots, v_m)$. Next, we consider the address of node x . Since $y < x$ and v is obtained by a bitwise OR operation between x and y , we have $q(x) = q(v)$. In addition, $(x_1, \dots, x_m)$ is either equal to $(v_1, \dots, v_m)$ or has a lower bit count than $(v_1, \dots, v_m)$. Hence, x should not appear in the tree. This is a contradiction. $\square$

For example, consider node 6 for the itemset $\{1, 4, 5\}$ as shown in Fig. 7. There are three nodes 7, 14, 15 that have subsets of this itemset and each of these nodes is either a descendant or a precursor of the node 6.

Proposition 3 is useful for pruning the computation for incremental intersection. Suppose that for some entry C , $t_i \subset \alpha_C$ holds. Thus, the intersection computations for every successor of C with t_i can be skipped as they do not store any subset of t_i (See Fig. 9).

Fig. 9 Reasoning for Proposition 3

The weeping tree can be used to perform the incremental intersection by traversing the weeping tree in a depth-first, left-to-right manner. Algorithm 3 sketches the process for updating a node x in $W(i)$ with an itemset E . Note that E is initially a transaction.

In the algorithm, a node, x , is identified with its associated entry, e_x . Here, α_x , c_x and Δ_x are the itemset, frequency, and error count of e_x , respectively. Given the transaction t_{i+1} and $W(i)$, the next tree, $W(i+1)$, is obtained by calling the function $update(root, t_{i+1}, W(i))$. Note that $W(0)$ is defined as the initial tree consisting of the *root* node.

A few characteristics of the $update(x, E, W(i))$ algorithm should be noted:

- Line 4 means that $\alpha_y \subseteq E$. By Proposition 2, the descendants of y are included by E . Thus, the frequency count of each node, $z \in C(y)$, can be simply incremented without computing the intersection except for α_y itself. This is called **descendant-intersect-skipping** (DIS).
- In Line 9, we continue the updating process. In the recursive call, the intersection, I , is used instead of the original itemset, E . It follows that $\alpha_{y'} \cap E = \alpha_{y'} \cap I$ for each child y' of y since $I = \alpha_y \cap E$. By reducing E to I (i.e., I is a subset of E), the computational cost of the recursive call after Line 9 is reduced. This pruning technique is called **masking**.
- If $I = \emptyset$, every descendant of y has no items in common E so the descendants need not be updated. This is called **descendant-update-skipping** (DUS).
- Line 11 checks if $E \subseteq \alpha_y$ or not. If so, every right sibling of y need not be updated. This follows from the observation in Proposition 3 that the entry for any subset of E to be updated never appears in the successors of y . This is called **successor-update-skipping** (SUS).
- Finally, if there is no entry for E in $W(i)$, the new entry for E is added as the right-most child of x . Note that if x is the root node, we set $c_x = 0$ and $\Delta_x = \Delta(i)$.

Algorithm 3 Incremental intersection with weeping tree.

```

1: function UPDATE( $x, E, W(i)$ )
2:   for each child  $y$  of  $x$  in order from left to right do
3:      $I := \alpha_y \cap E$ 
4:     if  $|I| = |\alpha_y|$  then                                     ▷ Case (1)
5:       for each descendant  $z$  of  $y$  do
6:         increment  $c_z$  by one
7:       end for
8:     else if  $I \neq \emptyset$  then                                   ▷ Case (2)
9:        $W(i) := update(y, I, W(i))$ 
10:    end if
11:    if  $|I| = |E|$  then                                         ▷ Case (3)
12:      break the loop
13:    end if
14:  end for
15:  if there is no entry for  $E$  in  $W(i)$  then
16:    create  $\langle E, c_x + 1, \Delta_x \rangle$  and
17:    add it to the right-most child of  $x$ 
18:  end if
19:  return  $W(i)$ 
20: end function

```

Fig. 10 The weeping trees $W(3)$ and $W(4)$ wrt $\mathcal{S}_4^2$

Example 7 We explain how Algorithm 3 works using

$$\mathcal{S}_4^2 = \{\{1, 2, 3, 5\}, \{1, 2, 4\}, \{2, 3, 4\}, \{1, 2, 5\}\}.$$

$W(3)$ corresponds to the left tree in Fig. 10. The function $update(root, t_4, W(3))$ is called to derive $W(4)$ from $W(3)$. For the left-most child e_1 , the intersection, $I_1 = \{1, 2, 5\}$ of e_1 with t_4 is computed. Since $I_1 \neq \emptyset$, we call $update(e_1, I_1, W(3))$ as shown in Line 9. For the left-most child, e_2 , of e_1 , the intersection $I_2 = \{1, 2\}$ of e_2 with I_1 is computed using I_1 by masking. Since $I_2 = \alpha_{e_2}$, DIS is applied in Lines 5–7. Then, c_{e_5} is simply incremented by one. Moving to the right sibling, e_4 , the intersection $I_4 = \{2\}$ of e_4 with I_2 is computed. Since $I_4 \neq \emptyset$, $update(e_4, I_4, W(3))$ is called. Since e_4 does not have any children, the algorithm checks if there exists the entry for I_4 then backtrack to the second call (i.e., $update(e_1, I_1, W(3))$). Since there is no sibling of e_4 , the algorithm returns to Line 15 and a new entry, e_7 , for I_1 is added as the right most child of e_1 . After backtracking to the first call, SUS is applied in Line 11 since $I_1 = t_4$. Thus, the updating of the two right-most sibling nodes, e_3 and e_6 , is skipped and the algorithm proceeds to Line 15. Since an entry for t_4 exists, the updated tree, $W(4)$, is returned as the output.

In this way, the *update* function realizes the incremental intersection. Note that every node in $W(i)$ is visited at most once, which implies that Algorithm 3 efficiently runs $update(root, t_{i+1}, W(i))$ to return $W(i + 1)$ in $O(kL)$ time.

Next, we show how PARASOL realizes the minimum entry deletion in the weeping tree. As explained before, the minimum entries to be deleted are allocated in the shallowest layer relative to the root. Recall Example 5, in which ϵ was set to 0.25 and PARASOL was used to delete the entries with frequency counts of one at time $i = 4$. These minimum entries can be quickly accessed by applying min-heap to the shallowest layer. The reduced weeping tree is obtained by reconnecting the children of the deleted nodes with the root as shown in Fig. 11.

The weeping tree can be used as a pre-processing step for the Δ -compression in such a way that we find such a child node e that $c_e \leq c_r - \Delta_r + \Delta(n)$ where r is the parent node of e . By Corollary 1, e can be deleted from the tree.

Fig. 11 The reduced weeping tree of $W(4)$

Figure 11 shows the reduced tree obtained by searching the weeping tree $W(4)$ in a bottom up manner from the left-most leaf to the root; this process results in the removal of four nodes (7, 11, 13, and 15). Note that Δ -compression requires $O(k^2)$ time to completely check every pair of nodes. Thus, it is reasonable to carry out the Δ -compression in a two-step procedure; first checking each pair of parent-child nodes in a one-time scan and subsequently performing the brute-force search of the remaining nodes.

7 Experiment

Now, we empirically evaluate the performances of the baseline algorithm, PARASOL, and Δ -compression, respectively. They have been implemented with the weeping tree (in C language²). Ten real datasets were collected from Yamamoto et al. (2014) (Nos. 1-2), the FIMI repository³ (Nos. 3-9), and a new benchmark (No. 10) created from the Yahoo! Hadoop grid logs dataset⁴ as shown in Table 1. Note that n , $|I|$, L , and L_{ave} are the end time, the number of different types of items, the maximum transaction length, and the average transaction length, respectively.

First, the baseline algorithm was compared with the state-of-the-art RC method, Skip LC-SS (Yamamoto et al. 2014), on the existing real datasets (Nos. 1-9). Then, the scalability of the baseline algorithm was demonstrated on the synthesized datasets generated by the IBM market basket generator. Next, the performance of PARASOL was compared with those of the baseline algorithm and the state-of-the-art PC method, MOA-IncMine (Quadrana et al. 2015), when applied to the large-scale dataset (No. 10). Finally, we evaluated

²available from <https://github.com/yoshi-3/jiis2019.git>

³Frequent Itemset Mining Dataset Repository (from <http://fimi.ua.ac.be/data/>)

⁴Yahoo! Hadoop grid logs (from <https://webscope.sandbox.yahoo.com/>).

Table 2 Skip LC-SS versus the baseline algorithm

No	Skip LC-SS			Baseline algorithm		
	$\Delta(n)$	$\Delta(n)/n$	Time	$\Delta(n)$	$\Delta(n)/n$	Time
1	157	0.0094	0.15	20	0.0012	0.23
2	11,792	0.61	10	2,953	0.15	0.29
3	23,987	0.27	3.8	156	0.0018	1.20
4	340,182	1	16	179,683	0.53	1.40
5	3,196	1	20	2,434	0.76	4.50
6	67,556	1	18	62,736	0.93	2.10
7	113,143	0.11	1.8	2,681	0.0027	0.69
8	49,046	1	18	42,513	0.87	2.70
9	8,124	1	15	492	0.061	1.80

the effectiveness of Δ -compression in reducing the output. Table 2 describes the overall improvement that has been realized by the baseline algorithm, compared with Skip LC-SS, for each dataset in terms of the maximum error, $\Delta(n)$, the error ratio, $\frac{\Delta(n)}{n}$, and the average updating time (msec). In this experiment, the size constant, k , was fixed as 12,000. The results show that the baseline algorithm drastically reduced the error. Notably, in datasets 4, 5, 6, 8, and 9, the error ratio with Skip LC-SS reached one, meaning that the FIs could not be found for any σ ($0 < \sigma < 1$), while the baseline provided a $\Delta(n)$ -covered set of the FIs for which $\Delta(n) \leq \sigma n$. Moreover, the updating time of the baseline algorithm was lower than the one of Skip LC-SS.

Skip LC-SS was compared with the baseline algorithm with a range of size constants for the dataset 7, on which Skip LC-SS performed the best in terms of the error ratio. The results are summarized in Fig. 12a.

The data shows that the baseline algorithm (bold lines) performed better than Skip LC-SS in terms of maximum error and execution time for all values of k . Moreover, the execution time decreased linearly and $\Delta(n)$ increased as k was decreased.

Next, the scalability of the baseline algorithm was evaluated by varying the maximum transaction length, L , as it is applied to the synthesis data. This dataset was generated by the IBM market basket generator; its stream size, n , was 10,000 and the number of types of items, $|I|$, was about 24,000. Figure 12b describes the average updating time and the error ratio, $\frac{\Delta(n)}{n}$, as functions of L . As the time complexity for the updating process is $O(kL)$, the execution time linearly increased as L was increased. In an ordinary problem setting, we need to focus the solution space on 2^L itemsets for L , resulting in a rapid increase in the error count. In contrast, the proposed solution can tolerate such large transactions that always contain 5,000 items. In addition, the change in the error ratio as a function of k in the range of 1,000,000 to 7,000,000 was evaluated with the dataset 6, for which the error ratio achieved by the baseline algorithm was the largest. The results are shown in Fig. 12c. Then, the error ratio linearly decreased while the execution time increased as k was increased. Note that a simple open MP parallelization was used for this computation. Such a parallelization scheme can be used in RC approximation with a fixed memory resource (Yamamoto et al. 2016) but the issue of efficient implementation of such a scheme is beyond the scope of this paper.

Fig. 12 Performance of the baseline algorithm

Figures 13 and 14 show the time series changes of the number of used entries, $k(i)$, and the error ratio, $\frac{\Delta(i)}{i}$, respectively, which are obtained by applying PARASOL to the hadoop dataset (No. 10) with respect to $k = 10,000$ and $\epsilon = 0.015$. The results show that PARASOL violates the original ϵ value due to the RC approximation that was applied earlier in the time series period. However, after this period, the error ratio recovers to ϵ . This autonomous recover of ϵ is an interesting characteristic of PARASOL. Note that PARASOL finally derives T_n of 1,626 entries while the baseline algorithm always uses the 10,000 entries. Another characteristic of PARASOL lies in this observation that T_n becomes smaller

Fig. 13 Time series of the stored entries, k

Fig. 14 Time series of the error ratio, Δ

than k due to the PC approximation. In contrast, MOA-IncMine generates 770,995 SFCIs for some intermediate segment when $\epsilon = 0.1$ and a segment size of $w = 1,000$; thus, it could not finish updating the SFCIs (this was true for other values of w , as well). The drastic decrease in performance due to streaming transactions (or segments) is a common drawback among the existing PC methods. PARASOL can avoid it and remains sustainable by switching between PC and RC approximations.

Table 3 compares the performance of MOA-IncMine with that of PARASOL with Δ -compression when applied to the other datasets in terms of execution time and number of entries in the output. Given $\epsilon = 0.1$, MOA-IncMine was run with 1 GB Java heap size for six hours (the segment size was fixed as the default (1,000)). PARASOL was run with $k = 100,000$ and $\epsilon = 0.1$.

The results show that MOA-IncMine results in out-of-memory (OM) and time-out (TO) exceptions when applied to datasets 4, 5, 6, and 8. On the other hand, PARASOL successfully processed all of these datasets, however the error ratio ($\frac{\Delta(n)}{n}$), exceeded the initial value of ϵ . In datasets, 1, 3, 7, MOA-IncMine was faster than PARASOL. These datasets were relatively sparse, implying that MOA-IncMine may tend to be more powerful for sparse datasets, although the speed also depends on the segment size (as well as the size constant k for PARASOL). The output of PARASOL was generally smaller than that of MOA-IncMine due to Δ -compression as shown by number of used entries, $k(n)$.

Table 3 MOA-IncMine versus PARASOL

No	MOA-IncMine		PARASOL with Δ -compression		
	Time	#Num	$\Delta(n)/n$	Time	#Num ($k(n)$)
1	0.16 s	19	0.1	1.7 s	20 (23)
2	2 m, 46 s	16,270	0.1	12.92 s	182 (42,204)
3	0.68 s	58	0.1	3.02 s	30 (50)
4	OM	–	0.374	114 m, 7 s	4,194 (100k)
5	OM	–	0.611	9 m, 25 s	2,780 (100k)
6	TO	–	0.761	44 m, 32 s	3,833 (100k)
7	5.56 s	14	0.1	29.21 s	10 (15)
8	OM	–	0.779	29 m, 43 s	7,437 (100k)
9	2 m, 47 s	3,430	0.1	8.49 s	938 (6,290)

Table 4 Pruning effects of using the weeping tree

No	Visiting ratio (%)	Non- empty ratio	#Num ⁻	#Num ⁺	Time ⁻ (sec)	Time ⁺ (sec)
1	38.4	5.0	5,129	3,895	2.23	0.89
2	4.8	13.3	345	138	1.05	0.00
3	68.6	22.6	9,367	6,503	4.39	3.36
4	19.9	52.0	2,010	626	1.49	0.07
5	41.8	55.1	1,468	446	1.63	0.05
6	42.9	62.9	3,418	535	1.48	0.16
7	32.6	15.0	5,589	3,258	2.17	0.74
8	44.4	57.9	6,004	1,267	2.26	0.7
9	36.4	12.5	4,417	2,251	3.33	0.75

Table 4 shows the pruning effect of using the weeping tree structure (parameters were set as $k = 12,000$ and $\epsilon = 0$). The average ratio between traversing nodes and total nodes for each timestamp (i.e., Visiting ratio) and the average ratio between nodes that have non-empty intersections and total nodes for each timestamp (i.e., Non-empty ratio) are shown. The standard indexing technique (i.e., *cid_list*) forces us to compute all the non-empty intersections. Interestingly, the weeping tree succeeds in reducing this search space by the indexing technique in datasets 2, 4, 5, 6, and 8. Those datasets are relatively dense. Note that the execution time with the weeping tree was faster than *cid_list* for every dataset. This is due to the additional cost for maintaining the indexing data structure for each timestamp. The #Num⁻ and #Num⁺ columns show the number of entries after pre-processing and number of entries that were output by the Δ -compression, respectively. For example in dataset 6, the ordinal 12,000 entries were reduced to 3,418 by pre-processing and 535 of these entries remained in the output after the second filtering step. The Time⁻ and Time⁺ columns show the execution time of the Δ -compression with and without the pre-processing step, respectively. It is reasonable for Δ -compression to apply the pre-processing with the weeping tree.

8 Conclusion and future work

In this study, we have proposed a novel solution for FIM-SD that involves seeking a Δ -deficient condensed representation, from which every FI and its support can be restored, while bounding the maximum error by an integer, Δ . While the existing FIM-SD methods are limited to strictly PC and RC methods, we have mitigated the drawbacks of each approach by introducing a unified PC and RC-approximation scheme called PARASOL. We furthermore introduce a post-filtering technique called Δ -compression and a novel data structure called the weeping tree. Experimental trials on ten datasets show that the proposed technique outperforms the existing FIM-SD methods. Moreover, the proposed algorithm is scalable in terms of the transactional length; in other words, it can tolerate any bursty transaction without running into an OM exception. This feature is favorable for analyzing large volumes of streaming transactions that consistently contain many items. Such large stream-

ing transactions are often encountered in surveillance domains of sensor networks and cloud servers. In addition, along with the recent success achieved with deep learning, it is now necessary to analyze the cognitive correlations among mid-level objects for explanatory domains. We believe that integrating modern online learning with the solution presented here will give rise to a new methodology for streaming data analysis. From a technical standpoint, it will be important to introduce novel parallel-processing techniques to further improve the scalability of this technique. In addition, it will be fruitful to study how such techniques can be efficiently embedded into the proposed weeping tree structure.

Acknowledgments The authors would like to thank the anonymous reviewers for giving us useful and constructive comments. This work was supported by JSPS KAKENHI (Grant Number 17K00301 and 19K12096) and also supported by JST PRESTO.

Open Access This article is distributed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits unrestricted use, distribution, and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made.

Appendix A: Relation between Δ -closed set and Δ -covered set

The Δ -covered set has an interesting relationship with the so-called Δ -closed set (Boley et al. 2009), which is known as a condensed representation for FIs. An itemset α is Δ -closed if there exists no itemset β such that $\beta \succeq_{\Delta}^n \alpha$. Let $\mathcal{F}_{\Delta}$ denote the family of Δ -closed FIs. $\mathcal{F}_{\Delta}$ is uniquely determined but does not have the Δ -deficiency property. Consider again Example 1 and assume that every $\alpha_i \in P$ is frequent at some timestamp. Then, we have $\mathcal{F}_1 = \{\alpha_4\}$ (i.e., $\Delta = 1$), since α_i ($1 \leq i \leq 3$) is not 1-closed. $\mathcal{F}_1$ is unique but no longer 1-covered set of P . Indeed, the support of α_1 cannot be restored only by α_4 within the error range below one.

In contrast, there exist two 1-covered sets $Q_1 = \{\alpha_3, \alpha_4\}$ and $Q'_1 = \{\alpha_2, \alpha_4\}$. Note that $\mathcal{F}_1$ corresponds to the intersection of Q_1 and Q'_1 . This characteristic is described as follows:

Proposition 4 Let $\mathcal{F}$ be the family of FIs and Δ an integer. Then, $\mathcal{F}_{\Delta}$ is included by any Δ -covered set of $\mathcal{F}$.

Proof Assume that there exist a Δ -closed FI α such that $\alpha \notin T$ for some Δ -covered set T of $\mathcal{F}$. Since α is a FI but is not included in T , there should exist a $\beta \in T$ such that $\beta \succeq_{\Delta}^n \alpha$. However, this contradicts that α is Δ -closed. $\square$

Appendix B: Proof of Theorem 1

Let α be an itemset and S_i a stream at timestamp i . We define the function $f(\alpha, i) = \bigcap_{t \in \text{tran}(\alpha, i)} t$. Then, it is well-known that α is a closed itemset at timestamp i , that is, $\alpha \in \mathcal{C}_i$ if and only if $\alpha = f(\alpha, i)$ holds. Now, we prove that $\mathcal{C}_i \subseteq \mathcal{C}_{i-1} \cup \{t_i\} \cup \{\beta \mid \beta = \alpha \cap t_i, \beta \neq \emptyset, \alpha \in \mathcal{C}_{i-1}\}$ (*). Assume that $\alpha \in \mathcal{C}_i$. If $\text{tran}(\alpha, i-1) = \emptyset$, α should be equal to t_i , since $\alpha = \bigcap_{t \in \text{tran}(\alpha, i)} t$ holds. Else if $\text{tran}(\alpha, i-1) \neq \emptyset$ and $\alpha \subseteq t_i$, we have $\alpha = t_i \cap \beta$ where $\beta = f(\alpha, i-1)$. Otherwise, α should be equal to β . Since

$\beta \in \mathcal{C}_{i-1}$, $\alpha \in \mathcal{C}_{i-1} \cup \{t_i\} \cup \{\beta \mid \beta = \alpha \cap t_i, \beta \neq \emptyset, \alpha \in \mathcal{C}_{i-1}\}$ holds. In turn, the reverse inclusion relation of (*) is proven in the same argument. Note that Theorem 1 is regarded as an instance of Theorem 2 when the minimal-entry deletion does not happen.

References

- Borgelt, C., Yang, X., Cadenas, R.N., Saez, P.C., Montano, A.P. (2011). Finding closed item sets by intersecting transactions. In *Proc. of the 14th int. conf. on extending database technology (EDBT)* (pp. 367–376).
- Boley, M., Horváth, T., Wrobel, S. (2009). Efficient discovery of interesting patterns based on strong closedness. In *Proc. of SIAM int. conf. on data mining (SDM)* (pp. 1002–1013).
- Boley, M., Gärtner, T., Grosskreuz, H. (2010). Formal concept sampling for counting and threshold-free local pattern mining. In *Proc. of SIAM int. conf. on data mining (SDM)* (pp. 177–188).
- Chi, Y., Wang, H., Yu, P.S., Muntz, R.R. (2004). Moment: maintaining closed frequent itemsets over a stream sliding window. In *Proc. of the 4th IEEE int. conf. on data mining (ICDM)* (pp. 59–66).
- Chang, Y.-K. (2005). Simple and fast IP lookups using binomial spanning tree. *Journal of Computer Communications*, 28, 529–539.
- Cheng, J., Ke, Y., Ng, W. (2006). δ -tolerance closed frequent itemsets. In *ICDM* (pp. 139–148).
- Cheng, J., Ke, Y., Ng, W. (2008). Maintaining frequent closed itemsets over a sliding window. *Journal of Intelligence and Information Systems*, 31, 191–215.
- Han, J., Cheng, H., Xin, D., Yan, X. (2007). Frequent pattern mining: current status and future directions. *Journal of Data Mining and Knowledge Discovery*, 15, 55–86.
- Ginsberg, J., Mohebbi, M., Patel, R. (2009). Detecting influenza epidemics using search engine query data. *Nature*, 457, 1012–1014.
- Hu, Q., & Imielinski, T. (2017). ALPINE: progressive itemset mining with definite guarantees. In *Proc. of the 2017 SIAM int. conf. on data mining* (pp. 63–71).
- Jiang, N., & Gruenwald, L. (2006). CFI-Stream: mining closed frequent itemsets in data streams. In *Proc. of the 12th ACM SIGKDD* (pp. 592–597).
- Jin, R., & Agrawal, G. (2005). An algorithm for in-core frequent itemset mining on streaming data. In *ICDM* (pp. 210–217).
- Johnsson, S.L., & Ho, C.-T. (1989). Optimal broadcasting and personalized communication in hypercubes. *IEEE Transactions on Computers*, 38(9), 1249–1268.
- Keogh, E.J., Chu, S., Hart, D., Pazzani, M.J. (2001). An online algorithm for segmenting time series. In *ICDM* (pp. 289–296).
- Karp, R.M., & Shenker, S. (2003). A simple algorithm for finding frequent elements in streams and bags. *ACM Transactions on Database Systems*, 28(1), 51–55.
- Li, H.-F., Shan, M.-K., Lee, S.-Y. (2008). DSM-FI: an efficient algorithm for mining frequent itemsets in data streams. *Journal of Knowledge and Information Systems*, 17, 79–97.
- Lee, V.E., Jin, R., Agrawal, G. (2014). Frequent pattern mining in data streams. *Book chapter of frequent pattern mining* (pp. 199–223).
- Liu, G., Zhang, H., Wong, L. (2012). Finding minimum representative pattern sets. In *Proc. of the 18th ACM SIGKDD* (pp. 51–59).
- Manku, G.S., & Motwani, R. (2002). Approximate frequent counts over data streams. In *VLDB* (pp. 346–357).
- Mala, A., & Dhanaseelan, F.R. (2011). Data stream mining algorithms: a review of issues and existing approaches. *International Journal of Computer and Electrical Engineering*, 3(7), 2726–2732.
- Metwally, A., Agrawal, D., Abbadi, A.E. (2005). Efficient computation of frequent and top-k elements in data streams. In *Proc. of the 10th int. conf. on database theory (ICDT)* (pp. 398–412).
- Du, M., & Li, F. (2016). Spell: streaming parsing of system event logs. In *Proc. of the 16th int. conf. on data mining* (pp. 859–864).
- Quadrana, M., Bifet, A., Gavaldà, R. (2015). An efficient closed frequent itemset miner for the MOA stream mining system. *Journal of AI Communications*, 28, 143–158.
- Shin, S.J., Lee, D.S., Lee, W.S. (2014). CP-tree: an adaptive synopsis structure for compressing frequent itemsets over online data streams. *Journal of Information Sciences*, 278, 559–576.

- Song, G., Yang, D., Cui, B., Zheng, B., Liu, Y., Xie, K. (2007). CLAIM: an efficient method for related frequent closed itemsets mining over stream data. *DASFAA2007, LNCS4443*, 664–675.
- Xin, D., Han, J., Yan, X., Cheng, H. (2005). Mining compressed frequent-pattern sets. In *VLDB* (pp. 709–720).
- Yamamoto, Y., Iwanuma, K., Fukuda, S. (2014). Resource-oriented approximation for frequent itemset mining from bursty data streams. In *Proc. of the Int. Conf. on Management of Data (SIGMOD)* (pp. 165–179).
- Yamamoto, K., Ikebe, M., Asai, T., Motomura, M. (2016). FPGA-based stream processing for frequent itemset mining with incremental multiple hashes. *Circuits and Systems*, 3299–3309.
- Yen, S.-J., Wu, C.-W., Lee, Y.-S., Tseng, V.S., Hsieh, C.-H. (2011). A fast algorithm for mining frequent closed itemsets over stream sliding window. In *Proc. of the 2011 IEEE int. conf. on fuzzy systems* (pp. 996–1002).
- Hoffman, M.J., Blei, D.M., Bach, F.R. (2010). Online learning for latent dirichlet allocation. In *NIPS* (pp. 856–864).
- Iwata, T., Yamada, T., Sakurai, Y., Ueda, N. (2010). Online multiscale dynamic topic models. In *KDD* (pp. 663–672).
- Zhao, Y., Sundaresan, N., Shen, Z., Yu, P.S. (2013). Anatomy of a web-scale resale market: a data mining approach. In *WWW* (pp. 1533–1544).
- Zhu, Y., & Shasha, D. (2002). Statistical monitoring of thousands of data streams in real time. In *VLDB* (pp. 358–369).

Publisher's note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.