Approximate-Closed-Itemset Mining for Streaming Data Under Resource Constraint

Yoshitaka Yamamoto  
University of Yamanashi, Japan  
yyamamoto@yamanashi.ac.jp

Yasuo Tabei  
RIKEN Center for Advanced Intelligence Project, Japan  
yasuo.tabei@riken.jp

Koji Iwanuma  
University of Yamanashi, Japan  
iwanuma@yamanashi.ac.jp

ABSTRACT
Here, we present a novel algorithm for frequent itemset mining for streaming data (FIM-SD). For the past decade, various FIM-SD methods in one-pass approximation settings have been developed to approximate the frequency of each itemset. These approaches 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 frequency 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 the notion of a condensed representation, called a \(\Delta\)-covered set, to the RC approximation. This notion is regarded as an extension of the closedness compression and when \(\Delta = 0\), the solution corresponds to an ordinary closed itemset. The algorithm searches for such approximate closed itemsets that can restore the frequent itemsets and their frequencies under resource constraint while the maximum error is bounded by an integer, \(\Delta\). We first propose a one-pass approximation algorithm to find the condensed solution. Then, we improve the basic algorithm by introducing a unified PC-RC approximation approach. Finally, 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

1. INTRODUCTION
Streaming data analysis is a central issue in many domains such as computer system monitoring [25], online text analysis [35, 34], financial and economic analyses [36, 37], and medical and health record data analysis [12, 17]. 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 for streaming data (FIM-SD) [1] is the most fundamental and well-used task in streaming data analysis. It is used to find frequently occurring itemsets in streaming data. It can be categorized into two major classes of solutions: (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.

Two major classes of solutions to compactly represent FIs in FIM-SD have been proposed: closed FIs, which have no frequent supersets with the same support, and maximal FIs, which are not contained in any other FI. Mining closed and maximal itemsets allows the number of solutions to be reduced by orders of magnitude compared with the mining of all FIs, which significantly reduces the memory consumed by FIM-SD. However, since an exponential number of closed or maximal itemsets may still exist, the risk of huge memory consumption by FIM-SD remains.

Recent research on FIM-SD [22, 15, 28, 19, 8, 27, 32, 26] has mainly pursued two directions: parameter constrained (PC) mining and resource constrained (RC) mining. In PC mining [29, 7, 8, 27, 26], an intermediate solution between closed and maximal itemsets, called a semi-frequent closed itemset (SFCI), is computed; a parameter, \(\epsilon\), controls the maximum error that can be included in the restored frequencies. Although PC mining enables a huge memory reduction compared with closed and maximal itemset mining, it still consumes huge amounts of memory when processing concept drift [13, 9]. This is illustrated in Figure 1 which shows examples of the number of SFCIs mined by MOA-IncMine [26], which is the state-of-the-art PC method, in a dataset of a hadoop grid logs [30]. The results show that the
Figure 1: Time-series on the number of SFCIs

number of SFCIs bursts when processing the 12th segment
of transactions (which represents concept drift) in the case
when ε is low (0.12), resulting in huge memory consumption.

On the other hand, RC mining is another approximation
working under resource (i.e., memory) constraint; Skip LCC-
SS is a representative RC mining method [32]. RC mining
works in a constant space based on a user-defined parameter
and returns approximate solutions of FIs; this represents a
significant advantage. However, a major disadvantage of
RC mining is the large error in approximate solutions due
to the strict space constraint, rendering some solutions in
the output useless [32].

Contributions. In this study, we tackle the problem of
approximation error in RC mining by introducing a novel
condensed representation called a ∆-covered set. The no-
tion of ∆-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, ∆.

First, we propose a new RC method to find this condensed
solution. The approach involves two key techniques: incre-
mental intersection, which is an incremental way to com-
pute the closed itemsets [1] [33], and minimum entry de-
letion, which is a space-saving technique for the RC approxi-
imation [24] [32]. Consequently, the proposed method exhibits
the following three characteristics:

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, ∆,
and return the output, T, at anytime.

Quality of output: for every FI α, T contains some su-
perset β of α such that the difference between the fre-
cuencies of α and β is at most ∆.

Next, we present a modification of this baseline method.
As shown in Figure 1, sudden and intensive memory con-
sumption can be considered a temporary phenomenon. In-
deed, normal PC approximation is sufficient except for within
a brief period. From the viewpoint of memory efficiency, it
is reasonable to switch between PC and RC approxima-
tions. Thus, in this study, we call such a “unified” approach
PARASOL. In PARASOL, the memory consumption is gen-
erally controlled using an error parameter and an RC ap-
proximation using a size constant is taken only in certain
scenarios. In addition, we present a post-processing tech-
nique called ∆-compression to reduce the size of the raw
output to yield a more concise ∆-covered set. Moreover, we
investigate a data structure to efficiently carry out the key
operations in the proposed method. 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. Sec-
tion 2 is a preliminary. Section 3 presents a brief review
of the existing methods. Section 4 describes the baseline
algorithm used to compute a ∆-covered set, and Section 5
describes PARASOL and ∆-compression. The utilized data
structure is discussed in Section 6. The experimental results
are summarized in Section 7, and we conclude in Section 8.

2. NOTATION AND TERMINOLOGY

Let I = {1, 2, . . . , N} be the universal set of items. Item-
set t is a non-empty subset of I, i.e., t ⊂ I. Data stream S_n
is the sequence of itemsets (t_1, t_2, . . . , t_n) for which t_i ⊂ I
for i = 1, 2, . . . , n where n denotes the timestamp for which
an output is requested. Each t_i is called trans-
action 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 S_n. For itemset α ⊂ I and timestamp i, tran(α, i) de-
notes the family of itemsets that include α as a subset at
timestamp i (i.e., tran(α, i) = {t_j | α ⊂ t_j, 1 ≤ j ≤ i}).

Support sup(α, i) of itemset α at timestamp i is defined as
|tran(α, i)|. Given a minimum support threshold σ (0 ≤
σ ≤ 1), if sup(α, n) > σn, then α is frequent with respect to
σ in S_n. \( F_n \) denotes the family of FIs with respect to σ at
timestamp n in S_n, i.e., \( F_n = \{ α \subset I | sup(α, n) > σn \} \).

FI α is closed if there is no α’s proper superset whose fre-
cuency is equal to α’s frequency in \( F_n \). On the other hand, FI α is
maximal if there is no α’s proper superset in \( F_n \).

2.1 Problem setting

In this paper, to overcome the limitation of the state-of-
the-art PC and RC methods, one key idea is to formalize a
novel FIM-SD problem in which we seek a condensed rep-
resentation of FIs with limited memory. This condensed
representation is defined as follows:

Definition 1 (Δ-cover). Let α and β be two item-
sets. If α ⊆ β and sup(α, i) ≤ sup(β, i) + Δ for a non-
negative integer Δ, then α is Δ-covered by β at timestamp i;
this state is denoted by α ⊆_Δ β.

Definition 2 (Δ-covered set). Let P and Q be two families of itemsets. If ∀α ∈ P ∃β ∈ Q such that α ⊆_Δ β
for a non-negative integer Δ, then Q is a Δ-covered set of
P at timestamp i.

Figure 2: Example of a Δ-covered set.


\[ \alpha \] covered set of \( P \). \( \alpha \)-covered sets of the family \( P \). A \( \alpha \)-covered set of the family \( P \) is a \( \Delta \)-deficient condensed representation. Given an error parameter, \( \epsilon \), the frequency range of the itemsets is divided into \( [\frac{1}{\epsilon}] \) intervals. Then, closed itemsets representing upper or lower bounds for each interval with respect to the inclusion relation were identified. By definition, these closed itemsets composed a \( \Delta \)-covered set. Thus, the notion of \( \Delta \)-covered sets is regarded as a generalization of the relaxed closed itemsets. This study also included the development of an online PC approximation algorithm called CLAIM, which incrementally updates relaxed closed itemsets by computing the \( \delta \)-cluster itemsets for each timestamp. However, CLAIM invokes a relatively complex updating process which makes it slow.

Cheng et al. [7] introduced a condensed representation, called a \( \delta \)-tolerance closed set, and presented a two-pass algorithm to enumerate \( \delta \)-tolerance closed sets using an FP tree. Unlike \( \delta \)-clusters, the maximum error is not controlled by the value \( \delta \).

### 3.2 Related work on online mining methods

Recently, several online algorithms for dealing with lossy condensed representations were proposed; several representative algorithms are summarized in Table 1.

Song et al. [28] presented the notion of relaxed closed itemsets which is a \( \Delta \)-deficient condensed representation. Given an error parameter, \( \epsilon \), the frequency range of the itemsets is divided into \( [\frac{1}{\epsilon}] \) intervals. Then, closed itemsets representing upper or lower bounds for each interval with respect to the inclusion relation were identified. By definition, these closed itemsets composed a \( \Delta \)-covered set. Thus, the notion of \( \Delta \)-covered sets is regarded as a generalization of the relaxed closed itemsets. This study also included the development of an online PC approximation algorithm called CLAIM, which incrementally updates relaxed closed itemsets by computing the \( \delta \)-cluster itemsets for each timestamp. However, CLAIM invokes a relatively complex updating process which makes it slow.

Cheng et al. [8] proposed an incremental method 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. Compared with CLAIM, IncMine utilizes various pruning techniques and efficient data structure. IncMine was implemented in a massive online analysis (MOA) platform which enables it widely distributed [26]. However, the updating process used by IncMine still requires subsets to be enumerated for each SCFI, resulting in a large time delay in some cases. Even worse, it is unknown if the obtained SFCIs are \( \Delta \)-deficient.

StreamMining [15] is a PC approximation method that seeks FIs. Based on an error parameter, \( \epsilon \), it deletes itemsets that are not promising as they have frequencies less than or equal to \( \epsilon \times n \). The accuracy of the solution (i.e., the maximum error in the restored frequencies) can be directly controlled by \( \epsilon \). However, this process involves a combinatorial explosion of the FIs causing huge memory consumption. CloStream [33] is a non-constrained (NC) online method that computes closed FIs exactly. Its solution is oriented to condensed representation of FIs, while memory efficiency is bounded by the limitation of lossless compression based on closedness.

### Table 1: Summary of online itemset mining methods

| Method       | Approach | Solution   | Accuracy | Memory |
|--------------|----------|------------|----------|--------|
| CLAIM [28]   | PC       | Relaxed CFIs | High     | Medium |
| StreamMining [15] | PC       | FIs        | High     | Exhaustive |
| CloStream [33] | NC       | Closed FIs | Exact    | High   |
| Skip LC-SS [32] | RC       | FIs        | Low      | Constant |
| IncMine [8]  | PC       | SFCIs      | High     | Medium |
| PARASOL (this study) | Hybrid | \( \Delta \)-covered set | High     | Constant |

**Example 1.** Consider the family, \( P \) of four itemsets \( \{\alpha_1, \alpha_2, \alpha_3, \alpha_4\} \), shown in Figure 3. 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.

A \( \Delta \)-covered set is a generalization of the closed itemsets. A \( 0 \)-covered set of the family \( F \) of FIs is equal to the closed itemsets of the same family \( F \). Furthermore, each FI and its frequency can be recovered from a \( \Delta \)-covered set with an error that is bounded by \( \Delta \); this property is called \( \Delta \)-deficiency. Let us consider the \( 2 \)-covered set \( Q_2 \), in Example 1. Given anti-monotonicity with respect to frequency, \( sup(\alpha_4, 5) \leq sup(\alpha_3, 5) \) \( (1 \leq i \leq 3) \) holds. Since \( Q_2 \) is a \( 2 \)-covered set of \( P \), \( \epsilon_4 \leq \epsilon_3 \). \( sup(\alpha_4, 5) \leq sup(\alpha_3, 5) + 2 \). Since both \( 3 \leq sup(\alpha_i, 5) \) and \( sup(\alpha_i, 5) \leq 5 \) hold, the error is bounded by 2.

The objective is to find a \( \Delta \)-covered set of \( F_\alpha \) for a non-negative integer, \( \Delta \), 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 \( \Delta \) according to an error parameter, \( \epsilon \), such that \( \Delta \leq \epsilon n \); 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 an RC method by which \( \Delta \) is kept as small as possible based on a given value of \( k \), which bounds the memory consumption. Herein, we present a novel RC method to find a \( \Delta \)-covered set of \( F_\alpha \) for a given \( k \) in \( O(kN\ln n) \) time and \( O(k) \) space, where \( L \) is the maximum length of transactions and \( n \) is the end timestamp.

### 3. RELATED WORK

Several methods have been proposed for mining condensed representations [29][28][2][8][21]. In this section, we briefly review the state-of-the-art techniques.

#### 3.1 Related work on offline mining methods

Boley et al. [2] proposed an offline algorithm to enumerate \( \Delta \)-closed sets from a transaction database. This method involves the use of a closure operator, in which each fixed point corresponds to a \( \Delta \)-closed set, to compute the frequent \( \Delta \)-closed sets, \( F_\Delta \). The computation time is \( O(N^2n|F_\alpha|) \). Unlike \( \Delta \)-covered sets, \( F_\Delta \) does not satisfy the \( \Delta \)-deficiency requirement while \( F_\Delta \) is included in any \( \Delta \)-covered set; this limitation restricts the potential applications of this method. The appendix details the relationship between \( F_\Delta \) and any \( \Delta \)-covered set.

Xin et al. [29] introduced another relaxed metric, called the \( \delta \)-cluster, which is a \( \Delta \)-deficient condensed representation based on a relative error metric. In that study, NP-hardness was applied to find the minimum \( \delta \)-cluster covering the FIs. In addition, a two-pass algorithm, called \( RP-local \), was introduced to construct a \( \delta \)-cluster based on a greedy strategy. Liu et al. [21] improved the efficiency for computing the \( \delta \)-cluster using a CFP-tree.

Cheng et al. [7] introduced a condensed representation, called a \( \delta \)-tolerance closed set, and presented a two-pass algorithm to enumerate \( \delta \)-tolerance closed sets using an FP tree. Unlike \( \delta \)-clusters, the maximum error is not controlled by the value \( \delta \).
To our knowledge, IncMine \[8\] is the state-of-the-art PC method for condensed representation mining. Unlike IncMine, PARASOL takes RC approximation just in case for processing concept drifts. Then, it can tolerate any bursty transaction without memory overflow. In usual, it takes the PC approximation to keep the accuracy of its solution high, provided that the solution ensures the \(\Delta\)-deficiency property.

Skip LC-SS \[32\] is an RC method that seeks FIs based on a space-saving technique \[24\]. Given a size constant \(k\), it can process any transaction with length \(L\) in \(O(k)\) space and \(O(kL)\) time, keeping the top- \(k\) itemsets with respect to their frequencies. However, Skip LC-SS must increment \(\Delta\) by one, whenever a transaction \(t_i\) that satisfies \(2^{\bar{\beta}^i} > k\) is processed; hence, it suffers from a rapid increase in \(\Delta\) in accordance with \(k\).

To mitigate this issue, PARASOL introduces a lossy condensed representation of FIs and takes RC approximation only for processing concept drifts. Thus, it can tolerate bursty transactions without causing memory overflow. In most cases, however, PARASOL uses the PC approximation to keep the accuracy of the solution high provided that the solution exhibits \(\Delta\)-deficiency.

Compared with the previously developed techniques, the present approach is novel in that it combines PC and RC methods to mine a \(\Delta\)-deficient solution. Although several online algorithms for FIM-SD have been proposed so far, none offer both high accuracy and space efficiency. Unlike the alternative methods, PARASOL avoids sudden and intense memory consumption associated with concept drift.

4. BASELINE ALGORITHM FOR ONLINE \(\Delta\)-COVERED SET MINING

PARASOL mines a \(\Delta\)-deficient condensed solution based on one-pass approximation settings. PARASOL is built on two key techniques: incremental intersection and minimum entry deletion, which are described in detail in this section.

Incremental intersection \[1, 33\] is used for computing the closed itemsets. It is based on the following cumulative and incremental features of the closed itemsets. Let \(C_i\) be the family of closed itemsets in \(S_i\).

\[
\text{Theorem 1.} \quad \text{Given} \ C_i \ \text{at timestamp} \ i \ \text{and transaction} \ t_{i+1} \ \text{at timestamp} \ i+1, \ C_{i+1} \ \text{is defined as follows:} \\
C_i = \emptyset \ \text{for} \ i = 0 \\
C_i = C_{i-1} \cup \{ t_i \} \cup \{ \beta \mid \beta = \alpha \cap t_i, \beta \neq \emptyset, \alpha \in C_{i-1} \} \ \text{for} \ i \geq 1
\]

Theorem 1 ensures that \(C_{i+1}\) can be computed from the intersection of each itemset in \(C_i\) with \(t_{i+1}\).

\[
\text{Example 2.} \quad \text{Let} \ I = \{1, 2, 3, 4, 5\} \ \text{and the stream} \ S_1^2 = \langle t_1, t_2, t_3, t_4 \rangle \ \text{where} \ t_i = I - \{i\} \ \text{for each} \ i (1 \leq i \leq 4). \ \text{Figure 3 illustrates how} \ C_1 \ \text{is incrementally generated until} \ i = 3. \ \text{Example 2.} \quad \text{Given} \ C_4 \ \text{at the first transaction.} \ C_2 \ \text{newly contains} \ a_2 \ \text{and} \ a_3, \ \text{which correspond to the second} \ \text{transaction} \ \text{and its intersection with} \ a_1, \ \text{respectively.} \ \text{Finally,} \ C_3 \ \text{and} \ C_4 \ \text{consist of the seven and fifteen closed itemsets, respectively.}
\]

An incremental intersection never unfolds a transaction to intermediate subsets. In this sense, it can simply but efficiently compute only the closed itemsets. On the other hand, lossless compression based on closedness cannot necessarily control an exponential increase in the number of closed itemsets; in the worst case, \(\Omega(2^L)\) closed itemsets can be generated. In Example 2, there is a total of fifteen closed itemsets from \(S_1^3\), which is almost the same as the \(2^L\) closed itemsets that are attained for \(L = 4\).

Next, we introduce a space-saving technique \[24, 32\] for RC approximation, called minimum entry deletion. Given a size constant, \(k\), RC methods generally sweep the currently stored itemsets and keep only the top- \(k\) itemsets; in other words, the itemset with the lowest frequency is deleted iteratively as long as the number of stored itemsets is greater than \(k\). The maximum frequency of the deleted itemsets is maintained as the maximum error, \(\Delta\).

PARASOL manages two types of information for each stored itemset: its estimated frequency and maximum error. They are represented as a tuple entry of the form \((\alpha, c, \Delta)\), which represents the stored itemset, the frequency count, and the error count, respectively. The itemset, frequency count, and error count of an entry, \(e\), are often denoted by \(\alpha_e\), \(c_e\), and \(\Delta_e\), respectively.

\[
\text{Example 3.} \quad \text{Consider} \ S_4^2 \ \text{again. Let} \ k = 3. \ \text{The number of} \ C_4 \ \text{is beyond} \ k = 3 \ \text{at timestamp} \ i = 3. \ \text{As shown in Figure 4, we remove four closed itemsets in order of increasing frequency; the highest frequency of those that are removed is two. The next transaction,} \ t_4, \ \text{is stored in the entry} \ (t_4, 3, 2), \ \text{meaning that the true frequency of} \ t_4 \ \text{is between one and three} \ (i.e., 1 \leq \text{sup}(t_4, 4) \leq 3).
\]

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 \(T_i\). 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.

\[
\text{Example 4.} \quad \text{Consider} \ T_4 \ \text{again as shown in Figure 3.} \ \text{Figure 3 corresponds to the output of Line 5 in Algorithm 1 at time} \ i = 4 \ \text{for the input stream} \ S_4^3 \ \text{and} \ k = 3. \ \text{By the deletion process of Line 6,} \ \Delta(4) \ \text{becomes three. Now, we assume that the}
\]
Algorithm 1 The baseline algorithm

Input: a size constant (k), minimum support threshold (σ) and data stream (S_n = (t_1, t_2, ..., t_n)).

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

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

top three entries in T_3 remain as shown in Figure 4. Next, let the transaction arriving at time i = 5 be t_5 = \{1,3,5\}. Then, by the addition process, we first add the entry (t_5,3,3) to T_4, since there is no entry for t_5 in T_4. After that, the intersection of each itemset stored 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 as shown in Lines 15-22 using the intersect function. Finally, one minimum entry, \{(2,5),3,0\}, is deleted from T_5 to ensure that k(5) ≤ k.

Next, we clarify the quality of the output.

Theorem 2. Algorithm 1 outputs a ∆(n)-covered set of the FIs wrt σ, provided that ∆(n) ≤ σn.

This implies that for every FI, α, T_n must contain an entry, e, in which α is ∆(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^n is used to denote the set of entries in T_i that contains such an itemset β that α ⊆ β. If T_i^n 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^n (i.e., r = \text{argmax}_{c_e \in T_i^n} (c_e)).

Then, for every FI, α, a representative entry, r, of α, we claim that α is ∆(n)-covered by α_r. This claim can be explained as shown in Figure 5. Since ∆(5) = 3, we set σ = 0.6 to ensure that ∆(5) ≤ σ × 5. There exist five FIs \{1\}, \{3\}, \{5\}, \{1,5\}, \{3,5\} wrt σ in S^5. Let α be the FI \{3,5\}. T_3^n uniquely contains the entry r = \{(1,3,5),4,3\},

![Figure 4: Example of minimal entry deletion](image)

Hence, r is the representative entry of α. Note that α = \{3,5\} is 1-covered by α_3 = \{1,3,5\}, since sup(\{1,3,5\}) = 3 and sup(\{3,5\}) = 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' ∈ T_i, such that α_e ⊆ α_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 α_e ⊆ α_e' then c_e ≤ c_e' holds). Then, by the cumulative feature of the incremental intersection, the output of the function 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. □

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 − Δr ≤ sup(α, t_i) ≤ c_r.

Proof. We use the mathematical induction. T_i consists of only one entry, \{(t_1,1,0)\}. Since k > 0, this entry is not deleted. Accordingly, this lemma is true in the case of T_1.
induction step by contradiction. Assume that \( \exists o \) such that \( sup(a, i + 1) > \Delta(i + 1) \) but there is no representative entry for \( \alpha \) 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 Figure 6).

Case 1: such an \( r \) exists in \( T_{i+1} \) but was deleted. By Lemma 1 we have \( c_r - \Delta_r \leq sup(a, i + 1) \leq c_r \). Since \( r \) was deleted, \( c_r \leq \Delta(i + 1) \) should hold. Thus, \( sup(a, i + 1) \leq \Delta(i + 1) \) 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, \( T_i^o \subseteq T_{i+1}^n \) holds. Since there is no representative entry for \( \alpha \) in \( T_{i+1} \), it holds that \( T_{i+1}^o = \emptyset \) and \( T_i^o = \emptyset \). Now, assume that \( T_{i+1}^o \) should contain the entry \( r_{i+1} \). Hence, \( \alpha \not\subseteq t_{i+1} \) because \( T_{i+1}^o = \emptyset \). Therefore, \( sup(a, i) = sup(a, i + 1) \). Since \( sup(a, i + 1) > \Delta(i + 1) \) and \( \Delta(i + 1) \geq \Delta(i) \), it should hold that \( sup(a, i) > \Delta(i) \). Then, by the assumption for \( T_{i+1} \), there should exist a representative entry for \( \alpha \) in \( T_{i+1} \). However, this contradicts the conclusion that \( T_{i+1}^o = \emptyset \).

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

Now, we prove Theorem 2 as follows:

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

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 5 and 10 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 7 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 \( O(kL + k \log k) \). □

Based on Theorems 2 and 3, Algorithm 1 can extract a \( \Delta \)-covered set of the FIs wrt \( \sigma \geq \Delta \) for an integer \( \Delta \) and can process each transaction in \( O(kL) \) time and \( O(k) \) space for a size constant \( k (k < 2^k) \). Therefore, Algorithm 1 satisfies the problem setting presented in Section 2.1.
5. IMPROVEMENT OF THE BASELINE ALGORITHM

In this section, we introduce two key improvements to the baseline algorithm.

5.1 Unifying PC and RC approximations

We observe in Figure 5 that a sudden and significant 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 embed a size constant, $k$, into the PC approximation and normally perform the deletion operation using an error parameter, $\epsilon$, but switch to an RC approximation only when the number of used entries exceeds $k$.

We call this unified approximation scheme PARASOL (an acronym for Parameter- and Resource-constrained Approximation for Soft and Lazy mining).

Unlike the existing PC approximation methods, PARASOL can process any bursty transactions in $O(kL)$ time without running out-of-memory. In contrast, PARASOL can process ordinary transactions without filling all $k$ possible entries while ensuring that the error ratio, $\frac{\Delta}{\epsilon}$, is less than or equal to the error parameter, $\epsilon$. This is unlike to the baseline algorithm based on the RC approximation. Given $k$ and $\epsilon$, PARASOL can be realized just by replacing the delete function in Algorithm 1 with the following function:

1: function PARASOL_delete(T_i)
2: $m := getMin(T_i)$ \(\triangleright m \) 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 in $T_i$
5: $deleteMin(T_i)$ \(\triangleright\) delete a minimum entry in $T_i$
6: $m := getMin(T_i)$
7: end while
8: return $\Delta(i)$
9: end function

If there is no deletion for the RC approximation (i.e., $k(i) \leq k$ for every time $i$), PARASOL ensures that $\Delta(i) \leq \epsilon \times i$; otherwise, $\Delta(i)$ can become greater than $\epsilon \times i$. Even in this case, the error ratio, $\frac{\Delta}{\epsilon \times i}$, gradually converges to $\epsilon$ as the RC approximation is only required for a short period. This self-sustained recovery of the error ratio is a unique and advantageous characteristic of PARASOL and will be empirically demonstrated later.

5.2 $\Delta$-compression

Note that the solution to this problem setting is not unique: there can exist many $\Delta$-covered sets of $F$. However, it is useful to extract a concise $\Delta$-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, $\Delta(n)$.

We explain the reasoning using $T_5$ (the right-most table in Figure 5) in Example 4. Since $\Delta(5) = 3$, the three itemsets stored in $T_5$ compose a 3-covered set of the FIs. Note that if $\{5\}$ was deleted, the two remaining itemsets still compose a 3-covered set, since $\{5\}$ is 1-covered by $\{1, 5\}$. Hence, we can delete an entry for any such set that is $\Delta(n)$-covered by another in $T_n$. In exchange for deleting $\{5\}$, we update the entry for $\{1, 5\}$ by $(\{1, 5\}, 5, 1)$ (i.e., incrementing the frequency and error counts by one). In this way, the frequency of $\{5\}$ can be restored from the entry for $\{1, 5\}$.

Note here that the frequency count of entry $e$ is not the exact frequency but rather an estimated one (i.e., $c_e$ is not necessarily equal to $sup(\alpha_e, n)$). Therefore, it is difficult to identify all pairs of itemsets that satisfy the $\Delta(n)$-covering relationship in $T(n)$. However, some of them can be detected based on the following proposition:

**Proposition 2.** Let $e_1$ and $e_2$ be two entries in $T(n)$. Then, $\alpha_{e_1}$ is $\Delta(n)$-covered by $\alpha_{e_2}$ if it holds that $\alpha_{e_1} \subseteq \alpha_{e_2}$ and $c_{e_1} \leq c_{e_2} - \Delta_2 + \Delta(n)$.

**Proof.** By Lemma 1 we have $sup(\alpha_{e_1}, n) \leq e_1$ and $c_{e_2} - \Delta_2 \leq sup(\alpha_{e_2}, n)$. Since $c_{e_1} \leq c_{e_2} - \Delta_2 + \Delta(n)$, it holds that $sup(\alpha_{e_1}, n) \leq sup(\alpha_{e_2}, n) + \Delta(n)$.

Based on Proposition 2, we proposed a post-processing technique, called $\Delta$-compression, that incrementally deletes the entry for a $\Delta(n)$-covered itemset from $T(n)$, as described in Algorithm 2. In exchange for deleting $e_1$, $\Delta_{e_2}$ is updated by increasing it by the difference between $c_{e_1}$ and $c_{e_2}$, and $e_{e_2}$ is updated to $e_{e_1}$ in Lines 4 and 5.

A simple implementation of Algorithm 2 is that for each entry $e_1$, we search $T(n)$ for a corresponding $e_2$. This can be achieved in $O(k^2)$ time, since $|k(n)| \leq k$.

6. DATA STRUCTURE

In this section, 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 $\Delta$-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 time $i$. This computation is often redundant: for example, if $\alpha_e$ has no common items with $t_i$, it is redundant to compute the intersection of $\alpha_e$ with $t_i$.

Based on this observation, Yen et al. [33] 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 $\bigcup_{e \in t_i} cid_list(x)$ 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. [1] 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 frequency 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 2), 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, we consider a novel data structure that is more suitable for our proposed algorithm: it is designed to prune redundant computations in incremental intersections and quickly access the minimum entries and ∆-covered entries as required for PARASOL deletion and ∆-compression.

6.1 Weeping tree

In this paper, we propose a variation of the binomial spanning tree [16], called weeping tree, in which a collection of entries, T(n), can be represented in a binary n-cube as follows: Let ε be an entry in T(n). By Theorem 1, αε corresponds to the intersection of a certain set S of transactions. This set can be represented as a binary address (x1, x2, . . . , xn) where each xj is one if S contains tj and zero otherwise. Every αε 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 αε for each entry ε ∈ T(n). Each binary address address can be described by an integer, x = Σj=1 (xj • 2n−j). Then, V(n) corresponds to a subset of {1, 2, . . . , 2n−1}.

Let x and y be two integers (1 ≤ x, y ≤ 2n−1) with the addresses (x1, x2, . . . , xn) and (y1, y2, . . . , yn), respectively. p(x) is the position in x that satisfies two conditions: xp(x) = 1 and xj = 0 for each j (p(x) + 1 ≤ j ≤ n). In other words, p(x) is the least significant set bit of x. We say that x covers y if yj = xj for each j where 1 ≤ j ≤ p(x). For example, if x = 12 and y = 15 in 4-cube with addresses (1100) and (1111) then, p(x) = 2, y1 = x1, and y2 = x2 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 [6].

DEFINITION 4 (BINOMIAL SPANNING TREE). Let V(n) be a subset of {1, 2, . . . , 2n−1}, x be an integer such that 0 ≤ x ≤ 2n−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 | y ∈ C(x) and βy′ ∈ C(x) s.t. y ̸= y′ and y ∈ C(y′)}. The siblings y(1), y(2), . . . , y(m) are sorted in the descending order (i.e., y(1) is the left sibling of y(i+1)). We say that a non-root node, y, is a descendant of node x if y ∈ C(x), a precursor of x if y /∈ 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 ε (i.e., αε is the intersection of the transactions indicated by the address x).

EXAMPLE 5. Consider again the stream S2 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, . . . , 15}). The corresponding weeping tree, W(4), is described in Figure 7. Each node, x, is associated with its own entry, ε. For example, the node x = 6 (0110) corresponds to the entry for α = t2 ∩ t1, i.e., α = {1, 4, 5}. Note also that C(6) = {7}, P(6) = {8, 9, . . . , 15} and S(6) = {1, 2, . . . , 5}.

![Figure 7: The weeping tree W(4) wrt S2 1](image)

One crucial feature of the weeping tree is that it captures inclusion relationships among the stored itemsets in T(n).

PROPOSITION 3. Let two nodes x and y associated with two entries εx and εy. If x covers y, then αεx ⊆ αεy.

Proof. Since x covers y, the address of y can be written as (x1, . . . , xp(x)), yp(x)+1, . . . , yn). Accordingly, αεy is written as ∩tj=1,j≤p(x) tj ∩ I where I = ∩tj=1,tj+1≤p(x) tj. Since p(x) is the position of the least significant bit, αεy = ∩tj=1,tj≤p(x) tj holds, followed by αεy = αεx ∩ I. □

Proposition 3 has three useful implications. First, it is applicable for pruning the intersection computations. Suppose that during the updating process at time i, an entry, ε, is found such that αε ⊆ ti. Since every descendant of ε must be included in ti, it is not necessary to compute the intersections for these descendants. Proposition 3 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 3 is applicable to the pairwise checking involved in ∆-compression. Suppose that we found a parent entry, εp, and its child entry, εc, such that Cεp ⊆ Cεc = ∆εp + ∆(n). Then, αεp must be ∆(n)-covered by αεc, according to Proposition 2. Hence, a quick check can be done to determine if each child is ∆(n)-covered by its parent. Note that a brute-force approach requires O(k2) time for ∆-compression while the quick pairwise checking can be completed in O(k) time. Thus, it is useful as a pre-processing step preceding ∆-compression.
6.2 Weeping tree updating

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

**Example 6.** Consider Example 5 again. Now, let \( \alpha \) 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 \( \alpha \) (see Figure 4) in \( T(3) \). Since \( r \) is given the address (101) at time \( i = 3 \), the address of \( \alpha \) becomes (1011), and the entry for \( \alpha \) is located as a child of node \( r \).

Now, we interpret the meaning of the address assigned to each entry. Let \( e \) be an entry of node \( x \) with the address \((x_1, \ldots, x_n)\). We denote the least and greatest significant bit sets of \( x \) as \( p(x) \) and \( q(x) \), respectively. Then, \( e \) is written as \((\alpha, \Delta(q(x) - 1) + \beta, \Delta(q(x) - 1))\), where \( \alpha = \bigcap_{t_j=1}^{t} x_j \) and \( \Delta(x) \) is the maximum error at time \( q(x) - 1 \) and \( \beta \) is the bit count of \( x \) (see Figure 8). This observation leads to the following proposition:

**Proposition 4.** Consider two nodes, \( x \) and \( y \), of entries \( e_x \) and \( e_y \), respectively, in a weeping tree. If \( \alpha_y \subseteq \alpha_x \) and \( y < x \), then \( y \) is either a descendant or a precursor of \( x \).

**Proof.** We derive a contradiction in the case that \( \alpha_y \subseteq \alpha_x \) 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 write \( v \) as \((v_1, \ldots, v_n)\) and denote by \( \alpha \) the intersection \( \bigcap_{t_j=1}^{t} x_j \). Since \( \alpha_y \subseteq \alpha_x \), we have \( \alpha_v = \alpha_y \). 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 \((u_1, \ldots, u_m)\) such that \( q(u) = q(v) \); moreover, the bit count of the \( m \)-prefix \((u_1, \ldots, u_m)\) is lower than the bit count of the \( m \)-prefix \((v_1, \ldots, v_m)\). 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 \((u_1, \ldots, u_m)\) or \((v_1, \ldots, v_m)\). 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, \((x_1, \ldots, x_m)\) is either equal to \((v_1, \ldots, v_m)\) or has a lower bit count than \((v_1, \ldots, v_m)\). 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 Figure 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 \( t_i \).

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, \( \alpha_x \), \( c_x \) and \( \Delta_x \) 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.

**Algorithm 3** Incremental intersection with weeping tree

```python
1: function update(x, E, W(i))
2:   for each child y of x in order from left to right do
3:       I := ay \cap E
4:       if |I| = |ay| then \( \triangleright \) Case (1)
5:         for each descendant z of y do
6:             increment cz by one
7:         end for
8:       else if I \( \neq \emptyset \) then \( \triangleright \) Case (2)
9:         W(i) := update(y, I, W(i))
10:   end if
11:   if |I| = |E| then \( \triangleright \) 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 (E, c_E + 1, \Delta_E) and
17:   add it to the right-most child of x
18: end if
19: return W(i)
20: end function
```

A few characteristics of the \( update(x, E, W(i)) \) algorithm should be noted:
• Line 4 means that $\alpha_y \subseteq E$. By Proposition 3, the descendants of $y$ are included by $E$. Thus, the frequency count of each node, $\varepsilon \in C(y)$, can be simply incremented without computing the intersection except for $\alpha_y$ 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 with $E$ 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 4 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 $\Delta_x = \Delta(i)$.

**Example 7.** We explain how Algorithm 3 works using $S^4_1 = \{(1, 2, 3, 5), \{1, 2, 4\}, \{2, 3, 4\}, \{1, 2, 5\}\}$. $W(3)$ corresponds to the left tree in Figure 17. The function update($\text{root}, t_4, W(3)$) 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, $e_3$, the intersection $I_3 = \{2\}$ of $e_3$ with $I_2$ is computed. Since $I_3 \neq \emptyset$, update($e_3, I_3, W(3)$) is called. Since $e_3$ does not have any children, the algorithm checks if there exists the entry for $I_3$ then backtracks to the second call (i.e., update($e_1, I_1, W(3)$)). Since there is no sibling of $e_3$, 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($\text{root}, t_{i+1}, W(i)$) 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 $\varepsilon$ 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 Figure 11.

Finally, the weeping tree is used as a pre-processing step to prepare for the $\Delta$-compression (i.e., seeking the tree for such a node that is $\Delta(n)$-covered by the parent). Figure 11 shows the reduced tree obtained by traversing 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 $\Delta$-compression requires $O(k^2)$ time to completely check every pair of nodes. Thus, it is reasonable to carry out the $\Delta$-compression in a two-step procedure; first checking the parent-child $\Delta(n)$-covering to remove the $\Delta(n)$-covered children in a one-time traversal 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 $\Delta$-compression, respectively. They have been implemented with the weeping tree (in C language). Ten real datasets were collected from Nos. 1-2, the FIMI repository (Nos. 3-9), and a new benchmark (No. 10) created from the Yahoo! Hadoop grid logs dataset as shown in Table 2. Note that $n_i$ or $|I|$, $L$,
and $L_{ave}$ are the end time, the number of different types of items, the maximum transaction length, and the average transaction length, respectively.

| 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 | 990,002 | 5.56 s |

Table 2: Characteristics of the used datasets

First, the baseline algorithm was compared with the state-of-the-art RC method, Skip LC-SS [52], 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 [26], when applied to the large-scale dataset (No. 10). Finally, we evaluated the effectiveness of $\Delta$-compression in reducing the output.

Table 3 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, $\Delta(n)/n$, 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 $\sigma$ ($0 < \sigma < 1$), 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 Figure 12(a). 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 12(b) describes the average updating time and the error ratio, $\Delta(n)/n$, 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 $2^k$ 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 Figure 12(c). 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 [31] but the issue of efficient implementation of such a scheme is beyond the scope of this paper.

Next, we compare PARASOL with the baseline algorithm and MOA-IncMine using the hadoop dataset (No. 10). Figures 13 and 14 show the time series changes of the numbers of used entries, $k(i)$, and the error ratio, $\Delta(i)/n$, respectively, while $k = 10,000$ and $\epsilon = 0.015$. The results show that PARASOL violates the set $\epsilon$ value due to the RC approximation that was applied earlier in the time series period but, after this period, the error ratio recovers to $\epsilon$, this autonomous recover of $\epsilon$ is characteristic of PARASOL. Moreover, $k(i)$ became smaller than $k$ due to the PC approximation in PARASOL. Note that PARASOL composes $T(n)$ of 1,626 entries while the baseline algorithm always uses the 10,000 entries. 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 characteristic of existing PC methods. PARASOL avoids this drawback and remains sustainable by switching between PC and RC approximations.

Table 3 compares the performance of MOA-IncMine with that of PARASOL with $\Delta$-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 a 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 $\epsilon = 0$.

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 processes all of these datasets although the error ratio, that MOA-IncMine causes the out-of-memory (abbreviated by OM) and time-out (abbreviated by TO) exceptions in No. 4, 5, 6, and 8 datasets. PARASOL processes all, however the error ratio $(\Delta(n))$, exceeded the initial value of $\epsilon$. In datasets, 1, 3, 7, MOA-IncMine was faster than PARASOL; this is because these datasets were relatively sparse, implying that PARASOL tends to be more powerful for dense datasets, although the speed also depends on the segment size and size constant. The output of PARASOL was generally smaller than that of MOA-IncMine due to $\Delta$-compression as shown by number of used entries, $k(n)$.

Table 5 shows the pruning effect of using the weeping tree structure while $k = 12,000$ and $\epsilon = 0$. The average ratio between traversing nodes and total nodes for each time and the average ratio between nodes that have non-empty intersections and total nodes for each time are shown. The results show that the use of the weeping tree structure is superior for pruning the search space in dense datasets (Nos. 2, 4, 5, 6, and 8) and does not impose any additional maintenance cost compared with the standard indexing technique. The $\#\text{Num}^-$ and $\#\text{Num}^+$ columns show the number of entries after pre-processing and number of entries that were output by the $\Delta$-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 $\Delta$-compression with and without the pre-processing step, respectively.

Table 5: Pruning effects of using the weeping tree

| No | Visiting ratio (%) | Non-empty ratio | $\#\text{Num}^-$ | $\#\text{Num}^+$ | 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          |

8. CONCLUDING REMARKS

In this study, we have proposed a novel solution for FIM-SD that involves seeking a $\Delta$-deficient condensed representation, from which every FI and its frequency can be restored, while bounding the maximum error by an integer, $\Delta$. 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 $\Delta$-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 streaming 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.

Appendix

The $\Delta$-covered set has an interesting relationship with the so-called $\Delta$-closed set $\mathcal{F}$, which is known as a condensed representation for FIs. An itemset $\alpha$ is $\Delta$-closed if there exists no itemset $\beta$ such that $\beta \supseteq \Delta \alpha$. Let $\mathcal{F}_\Delta$ denote the family of $\Delta$-closed FIs. $\mathcal{F}_\Delta$ is uniquely determined but does not have the $\Delta$-deficiency property. Consider again Example 1 and assume that every $\alpha_i \in P$ is frequent by setting $\sigma$ as 0.5. Thus, we have $\mathcal{F}_1 = \{\alpha_4\}$, since $\alpha_i$ (1 $\leq i \leq 3$) is not 1-closed. Accordingly, $\mathcal{F}_1$ is no longer a 1-covered set of $\mathcal{F}$, since the frequency of $\alpha_1$ cannot be restored from $\alpha_4$ within the error range below one.

Proposition 5. Let $\mathcal{F}$ be the family of FIs and $\Delta$ an integer. Then, $\mathcal{F}_\Delta$ is included by any $\Delta$-covered set of $\mathcal{F}$.

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

9. REFERENCES

[1] C. Borgelt, X. Yang, R. N. Cadenas, P. C. Saez and A. P. Montano: Finding closed item sets by intersecting transactions, Proc. of the 14th Int. Conf. on Extending Database Technology (EDBT), 367–376 (2011).
[2] M. Boley, T. Horváth and S. Wrobel: Efficient discovery of interesting patterns based on strong closedness, Proc. of SIAM Int. Conf. on Data Mining (SDM), 1002–1013 (2009).
[3] M. Boley, T. Gärtner and H. Grosskreutz: Formal concept sampling for counting and threshold-free local pattern mining, Proc. of SIAM Int. Conf. on Data Mining (SDM), 177–188 (2010).
[4] V. Bräveman et al.: BPTree: an $l_2$ heavy hitters algorithm using constant memory, Proc. on PODS, 1–15 (2016)
[5] G. Cormode and M. Hadjieleftheriou: Methods for finding frequent items in data streams, Int. J. on VLDB, 19(1), 3–20 (2010).
[6] Y.-K. Chang: Simple and fast IP lookups using binomial spanning tree, J. of Computer Communications, 28, 529–539 (2005).
[7] J. Cheng, Y. Ke, W. Ng: $\delta$-tolerance closed frequent itemsets, In ICDM, 139–148 (2006).
[8] J. Cheng, Y. Ke, W. Ng: Maintaining frequent closed itemsets over a sliding window, J. of Intell. Inf. Syst., 31, 191–215 (2008).
[9] Frequent Itemset Mining Dataset Repository (from http://fimi.ua.ac.be/data/).
[10] C. Giannella, J. Han, J. Pei and X. Yan: Mining frequent patterns in data streams at multiple time granularities, Next Generation Data Mining, 191–212 (2003).
[11] J. Han, H. Cheng, D. Xin and X. Yan: Frequent pattern mining: current status and future directions, J. of Data Mining and Knowledge Discovery, 15, 55–86 (2007).
[12] J. Ginsberg, M. Mohabbi and R. Patel: Detecting influenza epidemics using search engine query data, Nature, 457:1012-1014 (2009).
[13] Q. Hu and T. Imielinski: ALPINE: Progressive itemset mining with definite guarantees, Proc. of the 2017 SIAM Int. Conf. on Data Mining, 63–71 (2017).
[14] N. Jiang and L. Gruenwald: CFI-Stream: mining closed frequent itemsets in data streams, Proc. of the 12th ACM SIGKDD, 592–597 (2006).
[15] R. Jin and G. Agrawal: An algorithm for in-core frequent itemset mining on streaming data, In ICDM, 210–217 (2005).
[16] S. L. Johnsson and C.-T. Ho: Optimal broadcasting and personalized communication in hypercubes, IEEE Trans. on Computers, 38(9), 1249–1268 (1989).
[17] E. J. Keogh, S. Chu, D. Hart and M. J. Pazzani: An online algorithm for segmenting time series, In ICDM, 289-296 (2001).
[18] R. M. Karp and S. Shenker: A simple algorithm for finding frequent elements in streams and bags, ACM Trans. on Database Systems, 28(1), 51–55 (2003).
[19] H.-F. Li, M.-K. Shan and S.-Y. Lee: DSM-FI: an efficient algorithm for mining frequent itemsets in data streams, J. of Knowl. and Info. Sys., 17, 79–97 (2008).
[20] V. E. Lee, R. Jin and G. Agrawal: Frequent pattern mining in data streams, Book chapter of Frequent Pattern Mining, 199–223 (2014).
[21] G. Liu, H. Zhang, L. Wong: Finding minimum representative pattern sets, Proc. of the 18th ACM SIGKDD, 51–59 (2012).
[22] G. S. Manku and R. Motwani: Approximate frequent counts over data streams, In VLDB, 346–357 (2002).
[23] A. Mala and F. R. Dhanaseelan: Data stream mining algorithms: a review of issues and existing approaches, Int. J. of Comp. and Eng., 3(7), 2726–2732 (2011).
[24] A. Metwally, D. Agrawal and A. E. Abbadi: Efficient computation of frequent and top-k elements in data streams, Proc. of the 10th Int. Conf. on Database Theory (ICDT), 398–412 (2005).
[25] M. Du and F. Li: Spell: streaming parsing of system event logs, Proc. of the 16th Int. Conf. on Data Mining, 859–864 (2016).
[26] M. Quadrana, A. Bifet and R. Gavalda: An efficient
closed frequent itemset miner for the MOA stream mining system, *J. of AI Communications*, 28, 143–158 (2015).

[27] S. J. Shin, D. S. Lee and W. S. Lee: CP-tree: an adaptive synopsis structure for compressing frequent itemsets over online data streams, *J. of Information Sciences*, 278, 559–576 (2014).

[28] G. Song, D. Yang, B. Cui, B. Zheng, Y. Liu and K. Xie: CLAIM: an efficient method for related frequent closed itemsets mining over stream data, DASFAA2007, LNCS4443, 664-675 (2007).

[29] D. Xin, J. Han, X. Yan, H. Cheng: Mining compressed frequent-pattern sets, *In VLDB*, 709–720 (2005).

[30] Yahoo! Hadoop grid logs (from https://webscope.sandbox.yahoo.com).

[31] K. Yamamoto, M. Ikebe, T. Asai and M. Motomura: FPGA-based stream processing for frequent itemset mining with incremental multiple hashes, *Circuits and Systems*, 3299-3309 (2016).

[32] Y. Yamamoto, K. Iwanuma and S. Fukuda: Resource-oriented approximation for frequent itemset mining from bursty data streams, *Proc. of the Int. Conf. on Management of Data (SIGMOD)*, 165-179 (2014).

[33] S.-J. Yen, C.-W. Wu, Y.-S. Lee, V. S. Tseng and C.-H. Hsieh: A fast algorithm for mining frequent closed itemsets over stream sliding window, *Proc. of the 2011 IEEE Int. Conf. on Fuzzy Systems*, 996-1002 (2011).

[34] M. J. Hoffman, D. M. Blei and F. R. Bach: Online learning for latent dirichlet allocation, *In NIPS*, 856-864 (2010).

[35] T. Iwata, T. Yamada, Y. Sakurai and N. Ueda: Online multiscale dynamic topic models, *In KDD*, 663-672 (2010).

[36] Y. Zhao, N. Sundaresan, Z. Shen and P. S. Yu: Anatomy of a web-scale resale market: a data mining approach, *In WWW*, 1533-1544 (2013).

[37] Y. Zhu and D. Shasha: Statistical monitoring of thousands of data streams in real time, *In VLDB*, 358-369 (2002).