The Journal of Systems and Software 85 (2012) 1346–1362

Contents lists available at SciVerse ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Mining frequent patterns from dynamic data streams with data load management☆

Chao-Wei Li, Kuen-Fang Jea*, Ru-Ping Lin, Ssu-Fan Yen, Chih-Wei Hsu

Department of Computer Science and Engineering, National Chung-Hsing University, 250 Kuo-Kuang Road, Taichung 40227, Taiwan, ROC

ARTICLE INFO

Article history:
Received 1 November 2010
Received in revised form 8 August 2011
Accepted 13 January 2012
Available online 24 January 2012

Keywords:
Data mining
Data streams
Frequent patterns
Combinatorial approximation
Overload handling
Load shedding

ABSTRACT

In this paper, we study the practical problem of frequent-itemset discovery in data-stream environments which may suffer from data overload. The main issues include frequent-pattern mining and data-overload handling. Therefore, a mining algorithm together with two dedicated overload-handling mechanisms is proposed. The algorithm extracts basic information from streaming data and keeps the information in its data structure. The mining task is accomplished when requested by calculating the approximate counts of itemsets and then returning the frequent ones. When there exists data overload, one of the two mechanisms is executed to settle the overload by either improving system throughput or shedding data load. From the experimental data, we find that our mining algorithm is efficient and possesses good accuracy. More importantly, it could effectively manage data overload with the overload-handling mechanisms. Our research results may lead to a feasible solution for frequent-pattern mining in dynamic data streams.

1. Introduction

Nowadays many commercial applications have their data presented in the form of continuously transmitted stream, namely *data streams*. In such environments, data is generated at some end nodes or remote sites and received by a local system (to be processed and stored) with continuous transmission. It is usually desirable for decision makers to find out valuable information hidden in the stream. Data-stream mining is just a technique to continuously discover useful information or knowledge from a large amount of running data elements. Like data mining in traditional databases, the subjects of data-stream mining mainly include *frequent itemsets/patterns*, *association rules* (Agrawal and Srikant, 1994), *sequential rules*, *classification*, and *clustering*. Through the data-stream mining process, knowledge contained inside the stream can be discovered in a dynamic way.

Data mining from data streams has three kinds of *time models* (or *temporal spans*) (Zhu and Shasha, 2002). The first one is *landmark window model*, in which the range of mining covers all data elements that have ever been received. The second one is *damped/fading window model*, in which each data element is associated with a variable weight, and recent elements have higher weights than previous ones. The third one is *sliding window model*, in which a fixed-length window which moves with time is given, and the range of mining covers the recent data elements contained within the window. Due to the fact that early received data elements may become out of date and/or insignificant, i.e., the *timeliness* factor, among the three models, the sliding window model is more appropriate for many data-stream applications such as finance, sales, and marketing.

A data-stream mining system may suffer the problem of *data overload*, just like the case of over-demand electricity to a power supply system. A data stream is usually dynamic and its running speed may change with time. When the data transmission rate of the data-stream source exceeds the data processing rate of the mining algorithm of a mining system, e.g., during a peak period, the system is *overloaded with data* and thus unable to handle all incoming data elements properly within a time-unit. Furthermore, an overloaded system may work abnormally or even come into a crash. Accordingly, it is essential for a data-stream mining system to adequately deal with data overload and/or spikes in volume.

In this paper, we propose a load-controllable mining algorithm for discovering frequent patterns in transactional data streams. The mining algorithm works on the basis of *combinatorial approximation* (Jea and Li, 2009). To address the possible case of peak data-load at times, two dedicated overload-handling mechanisms are designed for the algorithm to manage overload situations with their respective means. With the load-controllable ability, a mining system with the proposed algorithm is able to work normally during

☆ This research is supported in part by NSC in Taiwan, ROC under Grant No. NSC 98-2221-E-005-081.

* Corresponding author. Tel.: +886 4 22840497x906; fax: +886 4 22852396.
E-mail address: kfjea@cs.nchu.edu.tw (K.-F. Jea).

0164-1212/$ – see front matter © 2012 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2012.01.024

high-data-load periods. Besides, according to our experimental data, the mining results (after overload handling) still possess reasonable quality in term of accuracy.

The rest of this paper is organized as follows. In Section 2, related work regarding data-stream frequent-pattern mining and data-overload handling is described. Section 3 gives the symbol representation, problem definition, and goal of this research. In Section 4, a mining algorithm together with two overload-handling mechanisms is proposed and explained in detail. Section 5 presents the experimental results with analyses. In Section 6, discussions are given on the overload-handling mechanisms. Finally, Section 7 concludes this work.

2. Related work

Frequent-pattern mining from data streams is initially limited to *singleton items* (e.g., Charikar et al., 2004; Fang et al., 1998). Lossy Counting (Manku and Motwani, 2002) is the first practical algorithm to discover frequent itemsets from transactional data streams. This algorithm works under the landmark window model. In addition to the minimum-support parameter (*ms*), Lossy Counting also employs an error-bound parameter, ε, to maintain those infrequent itemsets having the potential to become frequent in the near future. Lossy Counting processes the stream data batch by batch, and each batch includes bucket(s) of transactions. With the use of parameter ε, when an itemset is newly found, Lossy Counting knows the upper-bound of count that itemset may have before it has been monitored. As a result, the error in itemset's frequency count is limited and controllable. According to the experiments conducted by Manku and Motwani (2002), Lossy Counting can effectively find frequent itemsets over a transactional data stream. This algorithm is a representative of the *ε-deficient mining* methods and has many related extensions. For example, based on the estimation mechanism of Lossy Counting, a sliding window method for finding recently frequent itemsets in a data stream is proposed by Chang and Lee (2004).

A different type of method is to process on stream elements within a limited range and offer no-false mining answers. DSTree (Leung and Khan, 2006) is a representative approach of one such type, which is designed for *exact (stream) mining* of frequent itemsets under the sliding window model. Given a sliding window, DSTree uses its tree structure for capturing the contents of transactions in each batch of data within the current sliding window. More specifically, DSTree enumerates all itemsets (of any length) having ever occurred in transactions within the current window and maintains them fully in its tree structure. The update of tree structure is performed on every batch, while the mining task is delayed until it is needed. According to the experimental results given by Leung and Khan (2006), mining from DSTree achieves 100% accuracy (since all itemsets having ever occurred are stored and monitored). However, because this method needs to enumerate every itemset in each of the transactions, its efficiency is badly affected by the great computation complexity. As a result, it can hardly manage with a data stream consisting of long transactions.

Besides DSTree, there are other methods belonging to the type of exact stream mining. Li and Lee (2009) proposed a bit-sequence based, one-pass algorithm, called MFI-TransSW, to discover frequent itemsets from data-stream elements within a transaction-sensitive sliding window. Every item of each transaction is encoded in a *bit-sequence* representation for the purpose of reducing the time and memory necessary for window sliding; to slide the window efficiently, MFI-TransSW uses the *left bit-shift* technique for all bit-sequences. In addition, Tanbeer et al. (2009) proposed an algorithm called CPS-tree, which is closely related to DSTree, is proposed to discover the recent frequent patterns from a data stream over a sliding window. Instead of maintaining the batch information (of the sliding window) at each node of the tree structure (as DSTree does), CPS-tree maintains it only at the last node of each path to reduce the memory consumption. The authors also introduce the concept of dynamic tree restructuring to produce a compact frequency-descending tree structure during runtime.

Another type of method is to perform the mining task, i.e., discover frequent itemsets, through a support approximation process. One feasible approach is to apply the idea of *Inclusion–Exclusion Principle* in combinatorial mathematics (Liu, 1968), whose general form is shown in Eq. (1), to data-mining domain for *support calculation*, and further apply the theory of *Approximate Inclusion–Exclusion* (Linial and Nisan, 1990) for *approximate-count computation*. This mining approach is called *CA* (*combinatorial approximation*). The DSCA algorithm (Jea and Li, 2009) is a typical example of such an approach. DSCA operates under the landmark window model and monitors all itemsets of lengths 1 and 2 existing in the data stream. It performs the mining task by approximating the counts of longer itemsets (based on the monitored itemsets) and returning the frequent ones as the mining outcome. According to the experimental data given by Jea and Li (2009), the performance of DSCA is quite efficient. Besides DSCA, the SWCA algorithm (Li and Jea, 2011) is also an example of CA-based approach. This method is under the sliding window model and has made an effort to improve the accuracy of support approximation.

$$|A_1 \cup A_2 \cup \ldots \cup A_m| = \sum_i |A_i| - \sum_{i<j} |A_i \cap A_j| + \sum_{i<j<k} |A_i \cap A_j \cap A_k| + \cdots + (-1)^{m+1} |A_1 \cap A_2 \cap \ldots \cap A_m| \quad (1)$$

In the electrical utility industry, *load shedding*, an intentionally engineered electrical power outage, is a last resort measure used by an electric utility company in order to avoid a total blackout of the power system. It is usually in response to a situation where the demand for electricity exceeds the power supply capability of the network. In the data-stream mining domain, many data-stream sources are prone to dramatic spikes in volume (Babcock et al., 2003). Because the peak load during a spike can be orders of magnitude higher than normal loads, fully managing a data-stream mining system to handle the peak load is almost impractical. Therefore, it is important for mining systems which process data streams to be adaptable to unanticipated variations in data transmission rate. An overloaded mining system is unable to manage on its own with all incoming data promptly, so *data-overload handling*, or simply *overload handling*, such as discarding some unprocessed data (i.e., load shedding) becomes necessary for the system to be durable.

The Loadstar system proposed by Chi et al. (2005) introduces load shedding techniques to *classifying* multiple data streams of large volume and high speed. Loadstar employs a metric known as the *quality of decision* (QoD) to measure the level of uncertainty in classification. When sources are in competition with each other for resources, the resources are allocated to the source(s) where uncertainty is high. In order to make optimal classification decisions, Loadstar relies on *feature prediction* to model the dropped and unseen data, and thus can adapt to the changing characteristics in data streams. Experimental results show that Loadstar offers a nice solution to data-stream classification with the presence of system overload.

To our knowledge, there are no representative studies concerning overload-handling schemes/mechanisms for frequent-pattern mining in data streams so far. However, the work conducted by Jea et al. (2010) has made a preliminary attempt. In the paper, a data-stream frequent-itemset mining system is proposed where the mining algorithm is a Lossy Counting based method. The system finds frequent itemsets from the data within a sliding window.

To cope with system-overload situations, a rough mechanism with three different strategies is devised and integrated into the system for shedding the system load. From the experimental data, the system is capable for a certain degree of controlling the workload during peak periods.

In this paper, basing a little on the research results of Jea et al. (2010), we study the problem of frequent-pattern mining in dynamic data streams. To address this problem, an efficient mining algorithm is proposed and mechanisms for overload management are formally designed, both of which will be described in detail later.

3. Problem statement

In this section, we characterize our notations and our definition of data load. We also address the issues of data overload, and describe the goal of this research.

Let $I = \{x_1, x_2, \ldots, x_z\}$ be a set of attributes, and each x_i in I be a single item. An itemset (or a pattern) X is a subset of I and written as $X = x_i x_j \ldots x_m$. The *length* of an itemset is the number of items it contains, and an itemset of length l is called an l-itemset. A transaction T consists of a set of ordered items, and T supports an itemset X if $X \subseteq T$. The *support* of an itemset X in a group of transactions is the number of occurrences of X within the group. An itemset X of length l which is a subset of another itemset Y or a transaction T is called an l-subset of Y or T. A transactional data stream on I is a continuous sequence of elements where each element is a transaction. A threshold of *minimum support* (*ms*) and a size of *sliding window* (*sw*) are two parameters specified by the user. The objective of frequent-pattern mining is to find out from data elements within the current sliding window the itemsets whose numbers of occurrences are equal to or greater than $ms \times sw$ when the user invokes a mining request.

Assume that there is a mining algorithm A running on a mining system and a data-stream source D. Let the processing rate of A on stream elements be z transactions in a time-unit, and the data transmission rate of D be w transactions in the same time-unit. The length of a time-unit depends on the data source and may not be fixed. Because a data stream is continuous and dynamic, its transmission rate varies as time goes by. In contrast with the data transmission rate, the data processing rate of A is relatively static across different time-units so long as D is in a settled data distribution. The *data load* L (of algorithm A's buffer) at a point is defined to be the ratio of transmission rate to processing rate at that point, that is, $L = w/z$.

As the value of w becomes greater and closer to that of z, the mining system which operates A will get busier in processing the continuously arriving data. When the transmission rate of D becomes higher than the data processing rate of A (i.e., $w > z$ or L is over 100%), the mining system is said to be *data overloaded*. In a data-overloaded situation, the system will receive a batch of w stream transactions within a time-unit whose quantity goes beyond its processing capability of z transactions. Data overload may cause a mining system to perform abnormally. More seriously, it may make higher and higher the system's load, or even paralyze the system. This is an important issue for a data-stream mining algorithm which should not be neglected.

Our goal in this research work is to devise a mining algorithm together with mechanisms for overload management to support frequent-itemset discovery (in the sliding window) in a dynamic data-stream environment. The algorithm has to handle data overload properly with the aid of the overload-handling mechanisms. Mining systems with our proposed algorithm should work normally even at the time when the system is data overloaded up to a certain level, for example, 150%.

4. Frequent-pattern mining algorithm and overload-manageable mechanisms

To achieve the goal stated in Section 3, two main issues need to be addressed; one is to discover frequent itemsets from recent stream elements effectively and efficiently, and the other is to manage with data overload adequately when getting into overloading situations. To address the first issue, a data-stream frequent-pattern mining algorithm is proposed in Section 4.1. For the purpose of data-overload management which concerns the second issue, two different mechanisms are devised and integrated into the mining algorithm. The first mechanism aims at accelerating the process of data handling, which will be explained in Section 4.2. The second mechanism, on the other hand, aims at diminishing the unprocessed data in quantity, which will be detailed in Section 4.3. After that, a summary of the algorithm and the mechanisms will be given in Section 4.4.

4.1. Frequent-pattern mining: dynamic-base combinatorial approximation

To discover frequent patterns in a data-stream environment which is prone to data overload, we remark that an algorithm C of the type *ε-deficient mining* (e.g., Chang and Lee, 2004; Manku and Motwani, 2002) or *exact stream mining* (e.g., Leung and Khan, 2006; Li and Lee, 2009; Tanbeer et al., 2009) may not be an appropriate candidate. There are two reasons for the inappropriateness of C. First, given a batch of incoming stream elements, the workload of C depends on and is inversely proportional to the value of *ms* (if C is an ε-deficient mining approach). A smaller *ms* will force C to process and maintain more data, in terms of both *the amount of itemsets* and *the length of pattern being handled up to*. This fact is a negative factor to data overload because the explosion of incoming data while overloading just brings about even more elements to be processed. Second, the mining result of C has a guarantee that every produced answer has *no error* (if C is an exact mining approach) or *only a limited error* (if C is an ε-deficient mining approach) in its *count*. However, data overload means the incoming data in a period is too much to be fully handled by C. In such a situation, the guarantee of C on its mining result (for example, discovering the complete set of recent frequent patterns) is hardly likely to be satisfied any more.

Due to the characteristic of continuity of the data-stream environment, it is infeasible to multi-scan the entire data, or even the data within a sliding window, for finding the counts of all candidate itemsets (to know which the frequent ones are). To achieve the objective of frequent-pattern discovery in data streams, we adopt an *approximation-based approach* which applies so-called *CA* to count computation. Briefly, if we have the count information of some itemsets, we could calculate the count of a candidate itemset which is grown from such already-known itemsets instead of scanning data for its counts. The obtained count is an approximation to the true count.

Here we give a concise explanation for the principle of CA. Assume there is an itemset X of length m (to be count approximated), and the counts of its first-k-length subsets are known, which compose the *base information* (or simply the *base*) of X. The count of X can be approximated from the counts of its subsets by applying the technique of Approximate Inclusion–Exclusion (Linial and Nisan, 1990). More specifically, the target m-union term can be approximated through the formula shown in Eq. (2), and the m-intersection term which corresponds to the count of m-itemset X can then be calculated. Depending on the relation between k and m, the approximated quantity for the m-union term (and thus the m-intersection term) differs from the exact quantity by at most a factor which is given in Eq. (3). Related derivations and proofs of the equations have been given by Linial and Nisan (1990). According to

Eqs. (2) and (3), if m is fixed while k is variable, choosing a value of k such that $k\sqrt{m}$ will generate approximations which attains a smaller error. Accordingly, mining outcomes with generally higher quality in terms of *accuracy* (e.g., *recall* and/or *precision*) can be obtained.

$$|A_1 \cup A_2 \cup \ldots \cup A_m| \approx \sum\nolimits_{|S|\le k} \alpha^{k,m}_{|S|} \left|\bigcap\nolimits_{i\in S} A_i\right|, \tag{2}$$

$$\frac{\left|\bigcup_{i=1}^{m} A_i\right|}{\sum_{|S|\le k} \alpha^{k,m}_{|S|} \left|\bigcap_{i\in S} A_i\right|} = \begin{cases} 1 + O(e^{-2k/\sqrt{m}}) & \text{if } k \ge \Omega(\sqrt{m}), \\ O\left(\dfrac{m}{k^2}\right) & \text{if } k \le O(\sqrt{m}). \end{cases} \tag{3}$$

Given a batch of (stream) data being received, the first few orders of itemsets are extracted from the batch and then kept as the (global) *base information*. The *size* of base information is the number of orders of itemset being extracted and saved. Base information is used for approximating the counts of *candidate itemsets*, that is, itemsets (whose lengths are over the base size) not belonging to the base information. We present a visual example in Fig. 1 to briefly illustrate both the operation process and essence of the CA-based approach. The base information is of size 2, that is, we capture and keep the itemsets of lengths 2 and under. In the example, $m = 3$, i.e., candidate itemsets of length 3 are to be count approximated, and $k = 2$, i.e., the counts of the first-2-order subsets of the candidate itemset are known. It can be found from the figure that the fast-calculated approximate count of a candidate itemset (e.g., *ace*) is close to its exact count (if the exact count is obtained by actually scanning the batch).

Applying CA to frequent-pattern discovery in data-stream environments is a feasible approach, and there are actual examples including Jea and Li (2009) and Li and Jea (2011) in the existing work. The algorithm proposed by Jea and Li (2009) is under the landmark window model while that proposed by Li and Jea (2011) is under the sliding window model of data streams. Both algorithms maintain the count information of all 1- and 2-itemsets as the *base*, that is, the base of *size* 2. When there is a request for frequent itemsets, they approximate the counts of itemsets whose length is m, $m \ge 3$, through the theory of Approximate Inclusion–Exclusion on the base. In both algorithms the *base size* can be set by at most 3, that is, only itemsets of lengths 3 and under are able to be captured as the base. The main reason is due to the static data structure they have adopted, which needs to reserve space for all possible itemsets belonging to the base and thus limits the base size.

In this research, we propose a different CA-based mining algorithm. The algorithm is under the sliding window model and slides the window in a segment-by-segment manner. More specifically, the sliding window is composed of a number of equal-sized separate *segments* and each segment corresponds to a transaction batch. Each slide of the window adds a new segment (of incoming stream elements) to and removes the oldest segment from the current window. To keep and maintain the extracted *base information*, i.e., itemsets of lengths k and under with their frequency counts, where k is the *size* of base information, we use a dynamically updated *trie* as our data structure for itemset keeping. Entries in the trie are basically of the form ⟨*item_id*, *count_list*, *last_existence*⟩ where *item_id* is the identification of an item, *count_list* is a circular array which records the count-value of the item in each segment (of the current window), and *last_existence* is the ID of the segment where the item has last occurred. In our algorithm, only those itemsets having ever occurred in the transactions within the current window will be allocated entries in the trie, and itemsets which do not exist within the current window anymore will be removed from the trie (by deleting the corresponding entries). As a result, the space utilization of the algorithm is much more efficient than that of the static-allocation approach.

Due to the efficient use of memory space, the data structure with dynamic allocation of entry makes the function of *keeping larger-size base information* with Eqs. (2) and (3) feasible to our algorithm. When a new batch of stream elements is received, the algorithm determines the size k of base information (on the batch) to be kept according to the *average length* of elements in the batch. If in one batch an element averagely has n items, then in general n is the maximal length of candidate itemsets which need to be count approximated. Therefore, $\sqrt{n}$ can be regarded as a basis when we decide the value of k. As long as k is chosen to be equal to or greater than $\sqrt{n}$, we can approximate the counts of itemsets with generally smaller errors as indicated by Eq. (3). We remark that our algorithm need not launch the mining process each time the window slides. More specifically, it performs the mining *at the request of user*, or it just extracts (and then keeps) the base information from each batch upon the batch's arrival. The pseudocode of our algorithm *DBCA* (namely ***d**ynamic-**b**ase **c**ombinatorial **a**pproximation*) is presented below.

Algorithm DBCA.

Input: A transactional data stream (D), the minimum-support value (*ms*), the sliding-window size (*sw*)
Output: A list F of frequent itemsets
Data structure: A lexicographic-ordered prefix trie (P)
Method:

```
1. Build an empty trie P;
2. while data of D is still streaming in do begin
3.   Clear the contents in F;
4.   while there is no request from the user do begin
5.     Receive from D a batch B of transactions;
6.     Calculate n the average length of transactions in B;
7.     Set the base-information size k for a value satisfying k ≥ ⌈√n⌉;
8.     for (i = 1; i ≤ k; i++) do
9.       Extract all i-itemsets whose supports in B satisfy the base threshold β;
10.      Update entries of the found i-itemsets in P;
11.    end for
12.    Remove from P the base information on the oldest segment within the
       current window;
13.  end while
14.  for (i = 1; i ≤ k; i++) do
15.    Find all large i-itemsets in P and insert them as F_i into F;
16.  end for
17.  for (m = k + 1; F_{m−1} ≠ NULL; m++) do
18.    foreach candidate m-itemsets X do
19.      Calculate the counts of X by Eq. (2) based on the base information;
20.      if X's approximate count ≥ sw × ms then
21.        Insert X as a member of F_m into F;
22.      end if
23.    end foreach
24.  end for
25.  Return F as the mining outcome;
26. end while
```

Briefly, for data-stream elements within the current sliding window, our algorithm monitors a portion of itemsets, and approximates the counts for the other portion of (not monitored) itemsets when necessary or requested. More specifically, the algorithm monitors those itemsets of lengths k and under out of all itemsets in each segment with n the average transaction length, where $k\sqrt{n}$. Once the value of k is set, only itemsets having a length of k or under whose supports are above a certain *base threshold* β will be extracted and kept in the data structure as the base information. Therefore the number of orders of itemsets required to be handled by DBCA is fixed and will not be affected by varying values of *ms*. The base threshold β of DBCA is set as, for example, *y occurrences within a batch*, where y is an integer. The minimum setting of β is 1, which means that itemsets having ever occurred in a segment will be extracted and kept. However, a slightly greater value of β such as 3 or 5 is usually preferred, since it may filter out many low-frequency itemsets hardly being used for count approximation (because such itemsets hardly grow into candidate frequent itemsets).

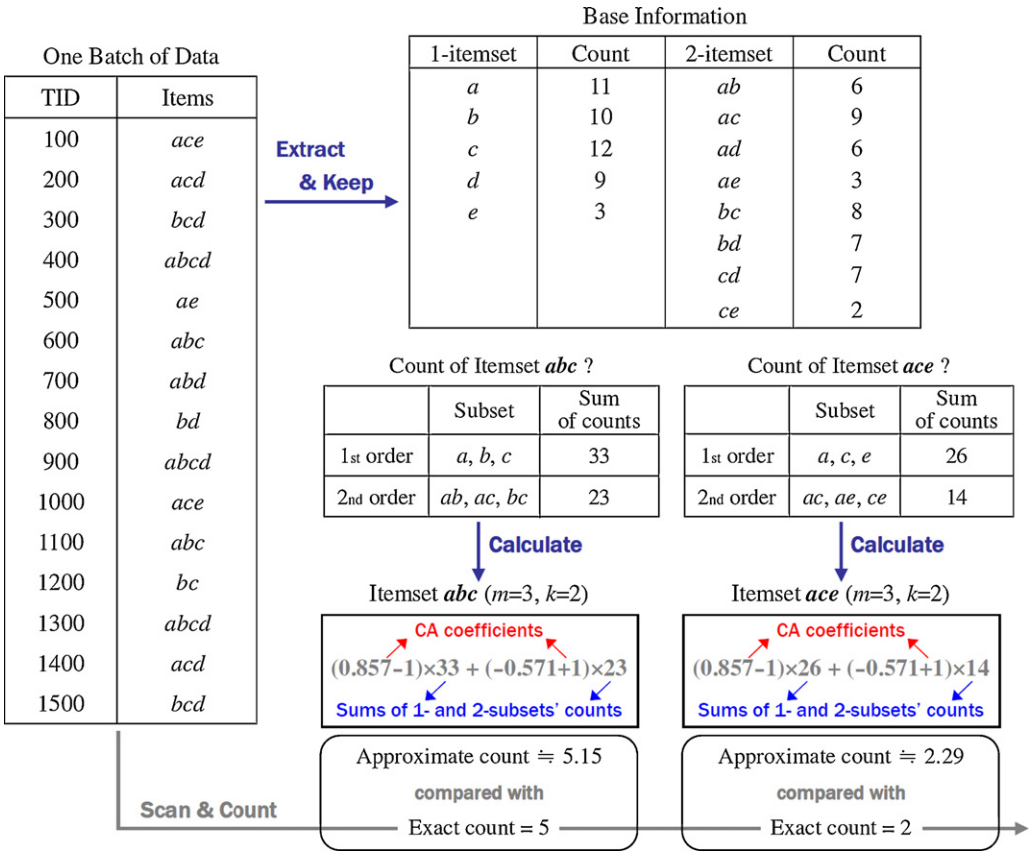


Fig. 1. An illustration of the process and essence of the CA-based approach.

For the reason that the counts of itemsets not monitored can be obtained at mining time through count approximation, rather than by data scan, the performance of DBCA is efficient, which is a property of the CA-based approach. Besides, the *dynamic-base design* of our algorithm possesses two features. First, for different data-stream sources or incoming data batches, it dynamically determines the size of base information which results in a better result of count approximation. Secondly, such dynamic extraction and maintenance of base information permits the so-called *processing acceleration mechanism* for overload handling, which will be described in the next section.

4.2. Overload handling: processing acceleration mechanism

The algorithm DBCA described in the previous section generally has higher performance on processing incoming stream elements than *ε-deficient mining* or *exact stream mining* methods, since it keeps base information and carries out the mining task (mainly count approximation for candidate itemsets) based on the kept information. The base information consists of only the first few (namely k) orders of itemsets out of all orders of itemsets, which is relatively compact as compared with the data needs to be processed and kept by the above types of methods. As a result, given a batch of data-stream elements being received, our algorithm may finish processing the batch within a shorter while, thus lowering the possibility of being data overloaded. However, the transmission rate of a data-stream source is unexpected, at times it may still exceed the data processing rate of our algorithm, e.g., during a peak period. Besides, there may be some constraints on processing time or processing rate imposed by the user (i.e., the *QoS* issue) that DBCA cannot satisfy adequately on its own.

Now we enter the second main issue regarding our goal, namely overload handling. According to Eqs. (2) and (3) of Approximate Inclusion–Exclusion, when the relation between k and m satisfies $k\sqrt{m}$, the m-union and m-intersection terms can be approximated with an error smaller than that concerning $k\sqrt{m}$. So our algorithm follows this rule to dynamically choose k (based on the average length n of stream transactions) as the size of base information, as we have described in Section 4.1. In fact, the manner of setting k's value can lead to a mechanism for managing with data overload.

Given a batch of stream elements with n the average length per element, to approximate the count by Eq. (2), the threshold of k is $\lceil\sqrt{n}\rceil$. Note that in general $\sqrt{n}$ is much smaller than n, especially for large values of n. In addition, a greater value of k (i.e., a larger size of base) generally produces a better approximation while a lower value of k (i.e., a smaller set of itemsets contained in the base) will shorten the time taken to process on the data batch. By taking these two factors together into consideration, we may set the *default value* of k to be $\lceil\sqrt{n}\rceil$ plus a *positive integer* such as 1, that is, $k = \lceil\sqrt{n}\rceil + 1$. In this way, our algorithm extracts and keeps base information of size k in general cases. When the data transmission rate of w becomes higher than the data processing rate of z (i.e., the case of data overload), our algorithm reduces the value of k to $\lceil\sqrt{n}\rceil$ as the base size for newly received data batch(es). We call this mechanism *processing acceleration mechanism*.

We use an example to illustrate such a mechanism. Suppose there is a data stream D in which elements have an average length of 15, i.e., $n = 15$. The threshold of k to enable better approximations is $\lceil\sqrt{n}\rceil = \lceil\sqrt{15}\rceil$, which is 4. Therefore, we take $k = \lceil\sqrt{n}\rceil + 1 = 5$ as the default size of base information. In the cases that DBCA is able to cope with a batch of stream elements received in a time-unit, it extracts itemsets whose length is not over 5 and keeps them as the

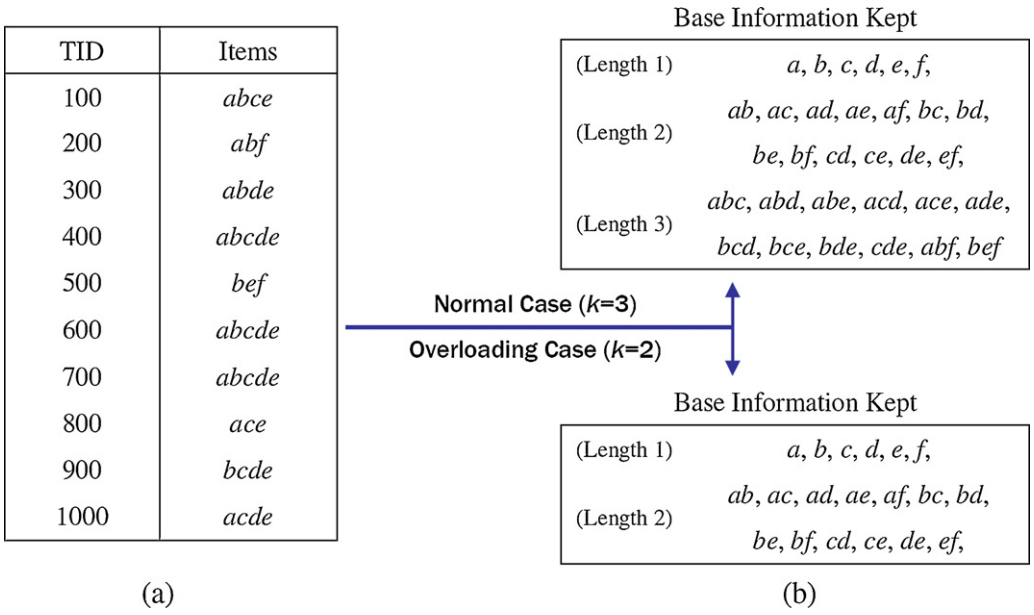


Fig. 2. An illustration of the effect of processing acceleration mechanism.

base. On the contrary, when a data batch received in a time-unit has too many elements to process in time (i.e., $w > z$), DBCA runs the mechanism which resets k to 4, and then extracts only itemsets of lengths 4 and under from the batch. Assuming that there are $|I|$ different attributes existing in the source D, the maximum possible number of itemsets in a batch need to be processed by DBCA before and after running the mechanism will be respectively $\sum_{i=1}^{5} C_i^{|I|}$ and $\sum_{i=1}^{4} C_i^{|I|}$; the latter one is obviously smaller and thus leads to a smaller time cost.

Here we use another smaller but visualized example for a clearer illustration. A batch of transactions is received from a data stream, as detailed in Fig. 2(a). In this batch, each data element averagely consists of 4 items, i.e., $n = 4$. Since $\lceil\sqrt{n}\rceil = 2$, the default value of k is set to $\lceil\sqrt{n}\rceil + 1 = 3$. Assume the base threshold β is set to be 1, that is, 1 occurrence in a batch. The contents of base information extracted and kept by DBCA without and with running the mechanism (i.e., $k = 3$ and $k = 2$) are respectively shown in the upper part and lower part of Fig. 2(b). The difference in number of extracted itemsets between the two cases is apparent. With running the mechanism, DBCA can finish processing the batch (i.e., keeping base information concerning the batch) in a shorter time. In other words, a larger amount of data elements can be handled within the same period. This mechanism brings acceleration into the data processing rate of DBCA.

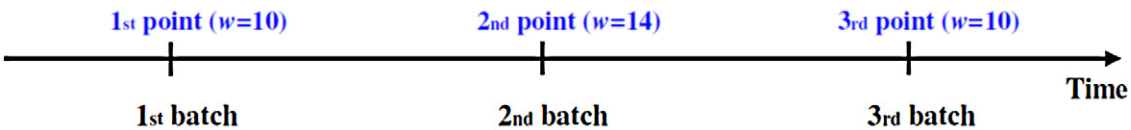
Now we give a more general and formal description to the processing acceleration mechanism. Given a batch of stream elements with n the average length, the *default value* of k is set for $\lceil\sqrt{n}\rceil$ plus *a positive integer* i, i.e., $k = \lceil\sqrt{n}\rceil + i$, $i > 0$. In general situations, the base information to be kept is of size k. In data-overload situations where the increasing data transmission rate exceeds the data processing rate, the value of k is scaled down to $k - j$ by some integer j, where $j > 0$ and $j \leqq i$. Since fewer itemsets need to be extracted as k becomes smaller, the data processing rate (or the *throughput*) on the newly received batch(es) of elements will be raised. This effect just deals with the difficult circumstance of data overload. When the data transmission rate later drops and becomes lower than the (original) data processing rate, the value of k is then set back to $k = \lceil\sqrt{n}\rceil + i$. In the case where the value of i is greater than 1, the value of k may be decreased *gradually*, depending upon the needs of different applications (i.e., QoS) or the required rate to process incoming data in time. In other words, the acceleration on the data processing rate may comprise *multiple stages*.

If the data is highly overloaded and the maximum setting of j, which corresponds to $k = \lceil\sqrt{n}\rceil$, is still unable to raise the data processing rate of DBCA up to the data transmission rate or above, it is possible to make *a further concession* to the base size k. More specifically, we could cut the base size k down to an integer value smaller than $\lceil\sqrt{n}\rceil$, that is, $k \leqq \lfloor\sqrt{n}\rfloor$. In this way, k's value could be decreased to $k - j$ by some integer j, where $j > 0$ and $j < k$. We remark that after all, in a highly overloaded situation, a way which has a less-quality result but is able to process altogether the received data in time will be preferable to that which possesses a better result in quality but can hardly finish the data processing timely. Nevertheless, this means should be regarded as *a last resort*, only when the acceleration on data processing rate under the basis of $k \geqq \lceil\sqrt{n}\rceil$ has failed to manage with data overload.

In summary, the basic idea of processing acceleration mechanism is to set the default value of k with some *reserves* to spare (for possible data-overload situations). When there exists data overload, running this mechanism raises the data processing rate of DBCA and thus lowers the excessive data load L of w/z in terms of increasing the value of z.

4.3. Overload handling: data-load shedding mechanism

When the data transmission rate of data stream D becomes higher than the data processing rate of mining algorithm A, the local end (i.e., mining system) receives stream elements within a time-unit by the amount w beyond its processing capability of z elements. Assume that the average length of stream elements in a batch is n items per transaction. To address the data overload problem, in contrast to the processing acceleration mechanism which raises the data processing rate of the mining algorithm, in this section we describe a mechanism called *data-load shedding* (or *load shedding* for short) which prunes some of the received data. As the unprocessed data is trimmed, the value of w becomes reduced and the excessive load L (of w/z) is thus lessened. The mechanism is *priority-based* which discards members in the overloading data according to their respective priority values. We also propose three practical policies to the mechanism.



1st batch

TID	Items
01	*abce*
02	*abe*
03	*bde*
04	*ab*
05	*be*
06	*abc*
07	*ae*
08	*abd*
09	*ad*
10	*bde*

2nd batch

TID	Items
11	*ade*
12	*abc*
13	*be*
14	*abe*
15	*abd*
16	*bde*
17	*abc*
18	*be*
19	*abcd*
20	*abde*
21	*abce*
22	*ac*
23	*abce*
24	*bd*

3rd batch

TID	Items
25	*abc*
26	*ace*
27	*be*
28	*abe*
29	*bde*
30	*abcde*
31	*ade*
32	*cd*
33	*ab*
34	*abde*

Fig. 3. An example of three batches of transactions received from a data stream *D*.

Because data overload means that the received amount of data (in terms of element) in a while exceeds the processing capacity of a mining system, one straightforward solution to this problem is to cut/delete some of the data elements before being processed by the mining algorithm. Let the *load shedding ratio*, or *LS ratio*, be $(w-z)/w$. At a point, if the LS ratio is equal to or less than 0%, there is no need to shed the data load (since the mining algorithm is capable of processing all data in time). On the other hand, if the LS ratio is greater than 0%, the mechanism is executed to trim the overloading data. Let p be the value of LS ratio at a time. If the LS ratio is over 0% then the buffer is overloaded with data and p percentage (in quantity) of the unprocessed data needs to be cut out through the mechanism.

In theory, the data elements to be deleted can be selected *randomly*, just making the quantity of remaining data be within the capacity range of the mining algorithm. However, since the discarded part of data will never be processed, the information contained inside it is lost. Therefore, dropping the unprocessed data without any care may bring about *serious effects* on the performance of mining, e.g., the efficiency of data processing or the accuracy of mining outcome. This is the reason why we propose a priority-based approach for the load shedding mechanism. Such an approach may manage data overload while preserving the mining algorithm's performance.

The basic element of a transactional data stream is a transaction, and each element consists of (a set of) attributes. Therefore, the *basic unit* to be discarded at data overload could be either a *transaction* or an *attribute*. Briefly, the priority-based load shedding mechanism sets up a *priority table* for the lookup of *priority value* of each basic unit. Depending upon the policies being adopted, there will be different *functions* to determine the priority value and create the priority table, and the p percentage of units with lowest priority values will be selected for discarding.

In the following subsections, we describe policies for priority-based load shedding. Here we first introduce an example *E*. Assume that from a data stream *D* the first three batches of data elements received are shown in Fig. 3. The minimum support for an itemset to be frequent is 3 occurrences within a batch. Let the value of z be 10, that is, DBCA can process 10 incoming transactions in one time-unit. For the sake of simplicity, let us consider that the data buffer of DBCA has a size of 10 too. At the first point the value of w is 10, which means there are 10 transactions (with TIDs 01–10) received from *D* in a period. Since the LS ratio is $(10-10)/10=0$, there is no need for load shedding and DBCA processes on data in the first batch normally. After that, i.e., at the second point, the value of w increases to 14, there are 14 new transactions (with TIDs 11–24) received during the next period. The LS ratio is now $(14-10)/14=0.286$, so before processing on the second batch, DBCA runs the load shedding mechanism to trim about 28% of unprocessed data in quantity in the batch. By the way, the value of w decreases back to 10 at the third point, which means there are 10 incoming transactions (with TIDs 25–34) in the next period and therefore load shedding is unnecessary for the third batch. Focusing on how load shedding is performed on the second batch, we will employ example *E* consistently in the following subsections for illustration purpose.

4.3.1. *Efficiency-oriented policy*

In this section, we describe a policy which aims at rapid data-load shedding and fast processing on the trimmed data after load shedding. The load shedding mechanism with this policy prunes the quantity of unprocessed data in terms of *transaction* back to the number z according to the LS ratio. The term '*efficiency*' here has two meanings. First, this policy works directly on the current batch and needs no feedback information from previous batches, so its execution of shedding data load is relatively efficient. Secondly, since a transaction with longer length has more subsets (than one of a shorter length) and will take DBCA more time to process, the method of this policy is to prioritize the deletion of transactions with long lengths. Therefore, the remaining data consists of shorter elements and can be processed efficiently by DBCA within a short while.

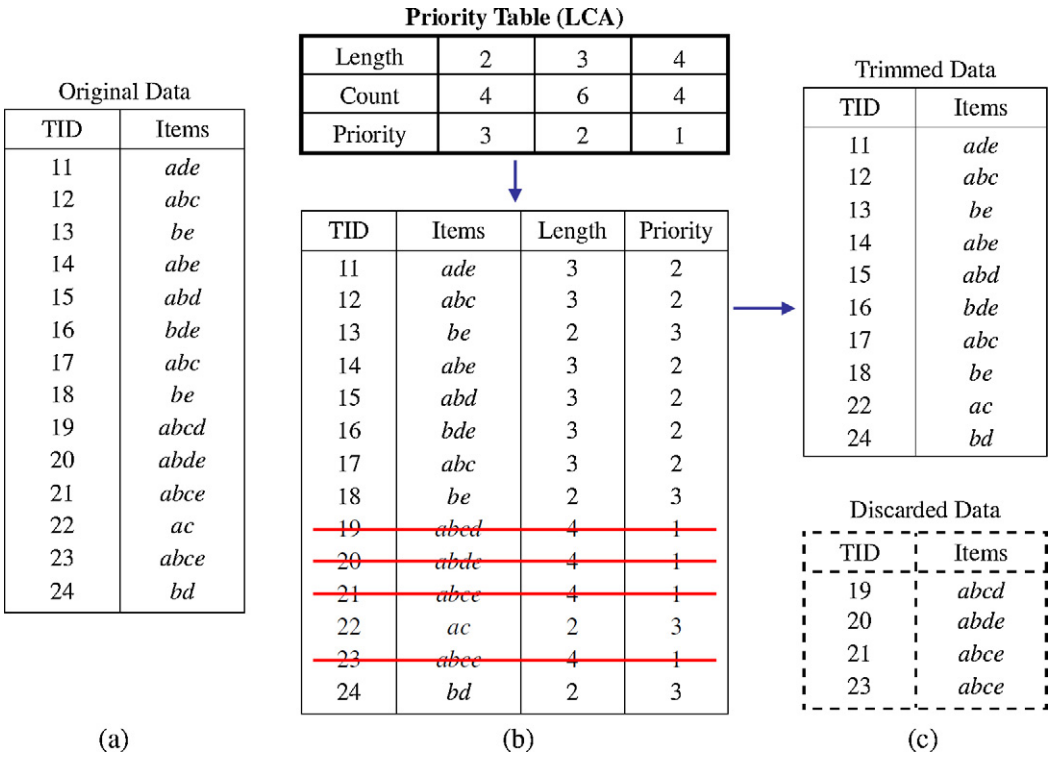


Fig. 4. A demonstration of the efficiency-oriented policy.

In this priority-based policy, the priority function utilizes the *element length* to decide the priority value of each unprocessed transaction. The longer length a transaction has, the lower priority-value it gets. An array called *LCA* (*length's count array*) is used to record the transaction lengths and the corresponding counts as well as priority values. This array is the aforementioned *priority table* to the policy. Among the received and unprocessed w transactions, the $w-z$ transactions with longest lengths are discarded by deleting (from low to high) the transactions having lowest priority values. Therefore, the remaining amount of transactions after discarding becomes z and is able to be handled rightly by DBCA.

Practical demonstration: In example *E*, the second batch of (unprocessed) data consists of transactions with TIDs from 11 to 24, as shown in Fig. 4(a). Since the LS ratio is 0.286, about 28% of transactions in the batch (i.e., 4 transactions) need to be pruned out. The data-load shedding mechanism first calculates the length of each transaction and then creates LCA, as shown in the upper part of Fig. 4(b). To prune the overloading data, the mechanism discards the transactions with lowest priority value(s), starting from low value to high value, until the number of remaining transactions falls below z, as illustrated in Fig. 4(b). In this example, transactions of length 4 are (first) selected to be discarded, and the data load then becomes normal. The execution of load shedding depends on nothing (from the previous batch) but LCA and thus is rather efficient. The contents of the trimmed data and the discarded data after load shedding are presented in Fig. 4(c). It is obvious from the figure that transactions in the remaining data have a shorter length in average.

4.3.2. Complexity-oriented policy

In this section, we describe a different policy which aims at lowering the complexity of data processing after load shedding. The load shedding mechanism with this policy prunes the unprocessed data in terms of *attribute*. If the amount of attributes in a data source is decreased, the possible number of pattern combinations will be correspondingly (or even substantially) diminished. Since an attribute with low frequency has less influence on frequent-pattern generation than an attribute with high frequency, the method of this policy is to prioritize the occurrence-elimination of attributes having low frequencies. Therefore, the remaining data consists of less attributes and the computational complexity on it gets reduced.

In this priority-based policy, the priority function utilizes the *frequency count* in the *previous batch* to decide the priority value of each attribute. That is to say, this policy requires some *feedback information* from the previous batch. The reason is that an attribute whose frequency is low in a batch will possibly have low frequency again in the next batch (i.e., the temporal continuity). The lower count-value an attribute has, the lower priority-value it gets. An array called *ACA* (*attribute's count array*) is used to record the counts and priority values of the attributes, which is the priority table to the policy. Since every transaction has n items in average, the received and unprocessed w transactions have about $w \times n$ items in total. In these transactions, those attributes having the lowest priority values are eliminated from the transactions, so that the trimmed data has its amount in *item* terms below $z \times n$ items and can be handled properly by DBCA with a processing rate of z in one time-unit.

Practical demonstration: In the second (current) batch of example *E*, since the LS ratio is 0.286, about 28% of items out of all items in the batch need to be pruned out. It can be counted from Fig. 5(a) that the number of items within the current batch is 42 (that is, each transaction has 3 items in average) so about 12 items need to be deleted. The load shedding mechanism creates ACA according to data in the previous batch, as shown in Fig. 5(b). From ACA we find that the attribute with lowest priority value is *c*, following by *d*, and so on. Therefore in the original data shown in Fig. 5(a), attributes are discarded by the order of priority indicated by ACA, until the number of remaining items is below 30. In this example the occurrences of attributes *c* and *d* are eliminated. Fig. 5(c) presents the contents of the data after load shedding. Although the amount of transaction remains unchanged, it is obvious from the figure that both the *average length of transactions* and the *number of pattern combinations* are decreased.

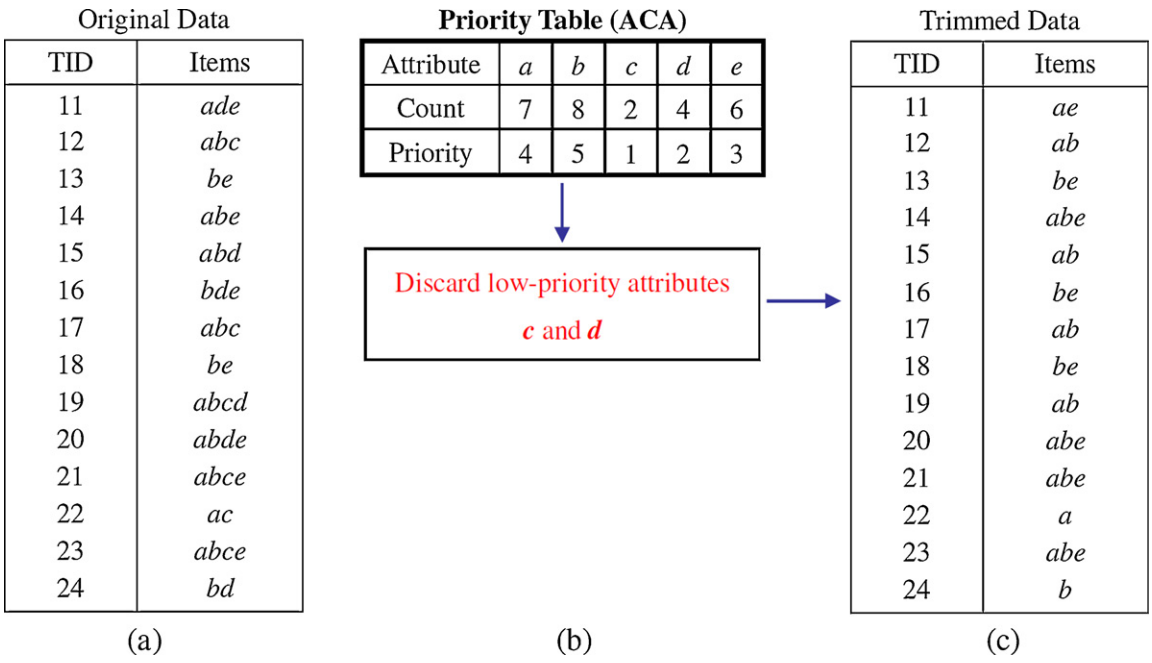


Fig. 5. A demonstration of the complexity-oriented policy.

4.3.3. Accuracy-oriented policy

In this section, we describe another policy which aims at preserving the accuracy of mining after load shedding. The load shedding mechanism with this policy trims the unprocessed data by *deleting some transactions* inside it. This policy relates to the *counts-bounding technique* proposed by Jea and Li (2009). In the counts-bounding technique, briefly, to approximate the count of an itemset X, the counts of the 1-subsets of X are *bounded* by the counts of X's 2-subsets, and counts in the bounded range are more relevant (for the 1-subsets) to X. We remark that DBCA is possible to apply the technique to approximate itemsets' counts. Thus this policy is designed based on such an assumption. Because the function of frequent 2-itemsets is to limit the count ranges of 1-itemsets, a transaction containing more 2-itemsets which are frequent is considered to be more important than a transaction containing less frequent 2-itemsets.

Nevertheless, the frequent 2-itemsets in the current (i.e., unprocessed) batch of data are unknown. Therefore, 2-itemsets which are frequent in the previous (i.e., processed) batch are taken as the *potential frequent 2-itemsets* for the current batch by considering the *temporal continuity*. That is to say, this policy requires some feedback information from the previous batch. If a transaction contains less potential frequent 2-itemsets, it has lower priority-value and will be dropped with a higher possibility. However, counting the exact number of potential frequent 2-itemsets contained in a transaction for all transactions in the current batch is extremely time-consuming. As a result, we use an approach to quickly calculate the *maximum possible number* of potential frequent 2-itemsets (as subsets) in a transaction.

In this priority-based policy, an array called *APA* (*attribute's position array*) is used for fast calculation of the maximum possible number (MPN) of potential frequent 2-subsets for every unprocessed transaction. This array helps to create the priority table to the policy. Since a 2-itemset consists of 2 attributes where the former is the *prefix* and the latter is the *suffix*, given R the set of frequent 2-itemsets in the previous batch, the APA records each attribute the times it occurs as the prefix and the suffix of 2-itemsets in R. Given a transaction T, the corresponding columns which match T's comprising attributes are retrieved from APA, and its MPN of potential frequent 2-subsets is calculated by combining every attribute whose prefix count is nonzero with every other attributes whose suffix count is nonzero.

Here we use an instance transaction *ade* to illustrate how its MPN of potential frequent 2-subsets is calculated. In the first batch of example E, the frequent 2-itemsets are *ab*, *ae*, *bd*, and *be*. So the APA for transactions in the second batch is created according to these itemsets, as shown in Fig. 6. At the beginning, the first, fourth, and fifth columns are retrieved from the APA to form the *local APA* for transaction *ade*, as shown in Fig. 7(a). Next, the attribute with nonzero prefix count, i.e., *a*, is combined with the other attributes whose suffix counts are nonzero. The target of suffix attribute(s) is *d* and *e* which respectively generate potential frequent 2-itemsets *ad* and *ae*. The MPN (of potential frequent 2-subsets) of transaction *ade* is calculated to be 2, as shown in Fig. 7(c). Note that the exact number of potential frequent 2-subsets of *ade* is 1 (i.e., itemset *ae*), while the MPN is an approximated value to the exact value. In this example, instead of checking each of the four potential frequent

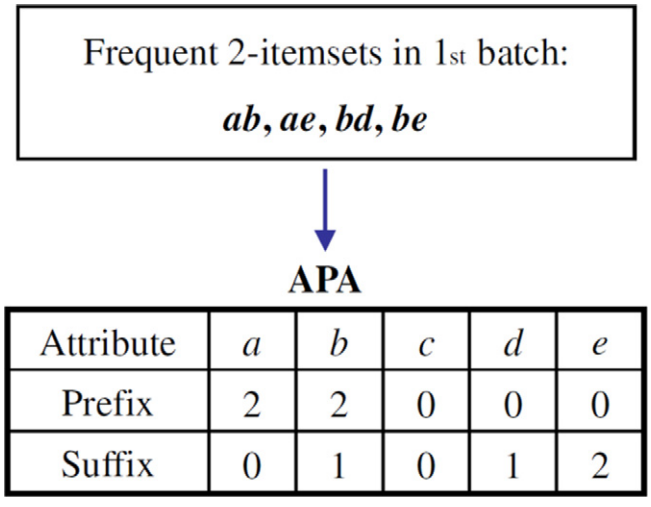


Fig. 6. Creating APA from frequent 2-itemsets in the previous batch.

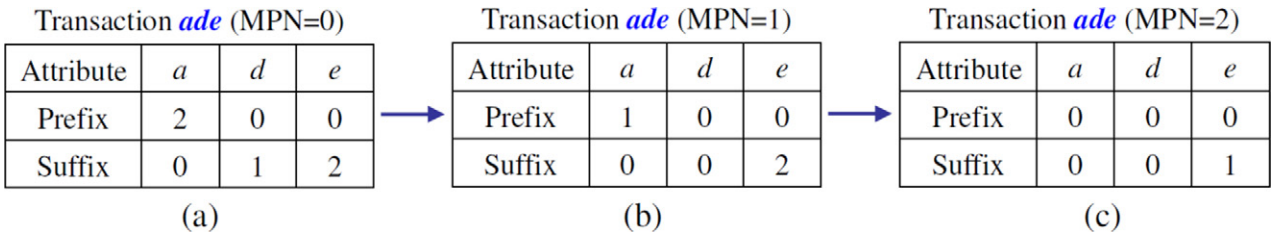


Fig. 7. An illustration of calculating the MPN to a transaction.

2-itemsets for whether it is contained in a transaction or not, we utilize the (local) APA to fast calculate the MPN of potential frequent 2-subsets to the transaction.

To perform the load shedding mechanism with this policy, the APA is first created. The priority function uses APA to calculate the MPN of (potential frequent 2-subsets contained in) each unprocessed transaction and then assigns different priority values to the transactions accordingly. The lower value of MPN a transaction has, the lower value of priority it receives. An array called *MCA* (*MPN's count array*) is used to record the MPNs and their corresponding counts as well as priority values, which is the priority table to the policy. Among the w transactions in the currently received batch, the $w-z$ transactions with least relevance to potential frequent 2-itemsets are discarded by deleting (from low to high) the transactions having lowest priority values. In the case where several transactions have a same priority value, the deletion of transaction is executed in order of TID. Therefore, the remaining amount of transactions after discarding becomes z and can be managed by DBCA.

Practical demonstration: The second data batch in example *E* includes 14 transactions, which is detailed in Fig. 8(a). Since the LS ratio is 0.286, 4 transactions in the batch need to be pruned out. The load shedding mechanism generates APA, calculates the MPN to each transaction, and then creates MCA, as shown in Fig. 8(b). To trim the overloading data, the mechanism discards transactions with lowest priority value(s), starting from low value to high value, until the number of remaining transactions falls below z. In this example, after 4 transactions which have the lowest priority values are deleted from the batch, the data load becomes normal. The contents of the trimmed data and the discarded data after

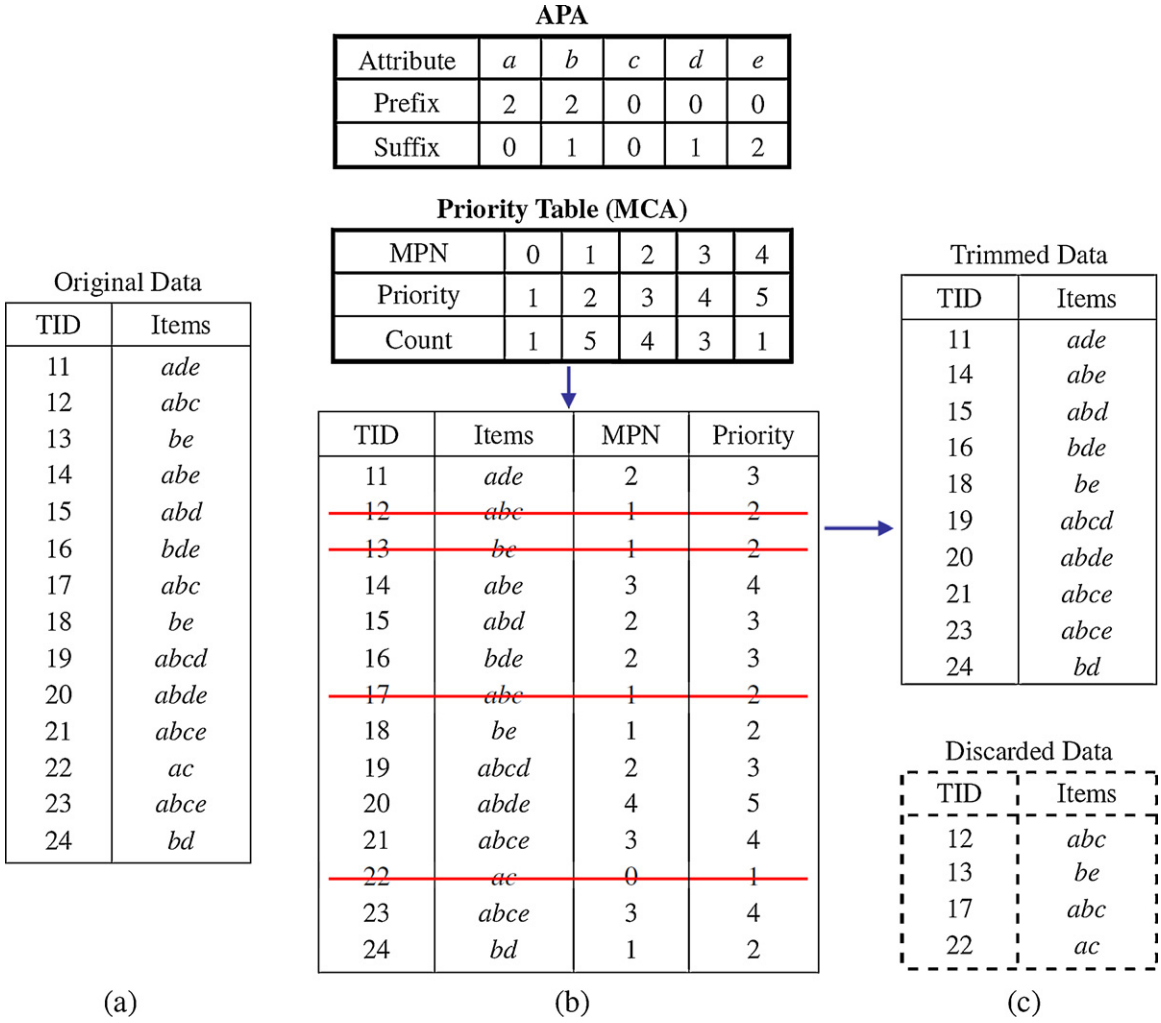


Fig. 8. A demonstration of the accuracy-oriented policy.

Table 1
A summary of the DBCA algorithm and the two overload-handling mechanisms.

Method	Type	Means regarding overload management	Basis
DBCA algorithm	Overload prevention	Keeping data-stream synopsis independent of varying *ms*	Application of CA
Processing acceleration mechanism	Overload treatment	Increasing data processing rate (i.e., throughput)	Keeping base information with a scalable size
Data-load shedding mechanism	Overload treatment	Decreasing data to be processed in quantity	Priority-based discarding of overloading data

load shedding are presented in Fig. 8(c). Although not so obvious from the figure, transactions in the trimmed data are those which have more potential frequent 2-subsets – in other words, they are more contributive to the counts-bounding technique concerning the mining accuracy.

4.4. Summaries

We first give a synopsis to our mining algorithm DBCA, which finds frequent itemsets in a sliding window with segment-based update. The algorithm maintains base information in a dynamic manner according to different sources or batches of data elements in a data stream. The setting of base size k is based on the condition of $k \geqq \lceil\sqrt{n}\rceil$, where n is the average length of elements in one batch. Since DBCA only extracts itemsets of the first k orders from the batch, and the base threshold β is fixed, its efficiency is independent of the value of *ms*. When requested for the mining outcome, DBCA applies Eq. (2) to approximate the counts of candidate itemsets and then returns those frequent ones. Therefore, compared with the static approach which maintains fix-sized base information all along, mining from DBCA generally attains better quality in accuracy terms.

We remark that DBCA is exactly a mining algorithm, but some of its features are plus factors to *data-overload prevention*. Given a batch of stream elements with an average length of n, DBCA only has to extract itemsets of lengths $k\ (= \lceil\sqrt{n}\rceil + i)$ and under. For a large value of n, the resultant value of k is usually much smaller than n, meaning that DBCA has to extract and keep only a small portion of all itemsets occurring in the batch. Because DBCA just maintains small-scale base information, it possesses a high data-processing rate to general data streams. Besides, the workload of DBCA is independent of *ms* due to its fixed base threshold. This feature makes the number of itemsets requiring to be processed not increased as lower values of *ms* are given. As a result, DBCA itself may also be regarded as one method for overload management.

While the DBCA algorithm lessens the risk of being data overloaded with its properties, the two mechanisms described in Sections 4.2 and 4.3 operate when the buffer is overloaded to deal with the overload through their respective means. When the data load L of w/z is over 100%, the processing acceleration mechanism increases the data processing rate of the mining algorithm (which corresponds to raising the value of z), while the data-load shedding mechanism decreases the volume of unprocessed data (which corresponds to reducing the value of w). Table 1 presents a summary of the DBCA algorithm and the two mechanisms regarding data-overload management.

In the mechanisms that we have devised to manage data overload, the data-load shedding mechanism is a priority-based mechanism. In Section 4.3, we have also described three practical policies for the mechanism to adopt. Based on the individual purpose of each policy, the necessary *priority function* and *priority table* are designed accordingly. We finally close this section with a summary of the three priority-based policies shown in Table 2.

5. Experimental results

This section shows empirical evidence about the proposed method. We have conducted three experiments to evaluate our mining algorithm DBCA as well as the two overload-handling mechanisms. All experiments were carried out on a platform of personal computer with an Intel 2.80 GHz dual-core processor and about 2 GB of available memory space. The operating system installed is Windows XP Professional SP3. The programs of the mining algorithm and the overload-management mechanisms are implemented using C++.

The testing data includes both *synthetic* and *real-life* datasets. Synthetic datasets are created by IBM's synthetic-data generator (http://www.almaden.ibm.com/cs/projects/iis/hdb/Projects/data_mining/datasets/syndata.html), and each dataset is generated with the parameters **T***s*.**I***t*.**D***u*.**A***v*, where s, t, u, and v respectively represent the average length of transactions, the average length of potentially frequent itemsets, the amount of transactions, and the amount of attributes in the dataset. Every adopted synthetic dataset consists of 500 thousands of transactions. The real-life dataset used in the experimental evaluation is **BMS-POS**, which can be acquired from the well-known website namely *FIMI*. This dataset contains a several-year quantity of point-of-sale data from a large electronics retailer. Table 3 summarizes the datasets utilized in our experiments. We employ these datasets to simulate the data source of frequent-pattern mining, i.e., transactional data streams.

The sliding window in the experiments is set for the size of 100k of transactions, and it consists of 10 equal-sized segments, namely 10k of transactions per segment. As described in Section 4.1, the window slides segment by segment (batch by batch). The slide of the window, i.e., adding a new segment and deleting the oldest segment, progresses every time a transaction batch whose size equals the segment size is newly received. The mining outcome is output each time when the sliding window has covered a range which separates from that of the previous mining. The base threshold β of DBCA in the experiments is set to be 5 occurrences in a batch.

Table 2
A summary of the three policies to the data-load shedding mechanism.

Policy	Aim/purpose	Basic unit of discarding	Feedback information	Priority function	Data structure (priority table)
Efficiency-oriented	Fast load shedding and increasing the throughput	Transaction	Not required	Transaction length based	LCA (length's count array)
Complexity-oriented	Decreasing the computational complexity	Attribute	Required	Attribute frequency based	ACA (attribute's count array)
Accuracy-oriented	Preserving the accuracy of mining	Transaction	Required	MPN of potential frequent 2-subsets based	APA and MCA (MPN's count array)

Table 3
A summary of datasets used in the experiments.

Dataset	Type	Avg. transaction length	Max. transaction length	Number of transactions	Number of attributes
T10.I4	Synthetic	10	30	500k	1500
T15.I6	Synthetic	15	42	500k	1500
T20.I8	Synthetic	20	50	500k	1500
BMS-POS	Real-life	6.5	164	515k	1657

We adopt both metrics of *mining performance* and *mining accuracy* to evaluate a mining algorithm, where performance is measured in terms of *run-time* and accuracy is measured in terms of *precision* and *recall*. For the sake of conciseness, we use the *F-measure* which is a weighted average of the precision and recall for presenting the accuracy. Besides, *memory consumption* is also observed and evaluated.

5.1. Experiment I: DBCA with processing acceleration

In the first experiment, we evaluate the performance of (normal) DBCA together with that of DBCA with processing acceleration mechanism. We also include Lossy Counting (Manku and Motwani, 2002) and DSTree (Leung and Khan, 2006) in this experiment for the purpose of comparison. Both methods have been implemented in C++, and Lossy Counting is modified to the sliding-window-model version (to agree with the other methods). The setting of sliding window to the two methods is identical with that to DBCA (as described). The minimum support (*ms*) in the experiment is in the values ranging from 0.1% to 1.0%; the parameter ε of Lossy Counting is set to be $0.1 \times ms$, which is suggested by the authors. Given a testing dataset, a mining algorithm is run on the dataset several times with different values of *ms*.

We first present experimental result on the *T*10.*I*4 dataset. Since the average length n of transactions in this dataset is 10, it can be calculated that $\lceil\sqrt{n}\rceil = 4$. The default size k of base information is therefore set to be 5 for DBCA. For the need of processing acceleration, we test the performance of DBCA on the base size k which is decreased to 4 and 3, that is, the *first stage* and *second stage* of acceleration, respectively. The empirical result is shown in Fig. 9.

In Fig. 9(a) of mining performance several facts can be observed. First, the run-time of DBCA (and also DSTree) is stable and has nearly nothing to do with different values of *ms*, just as we have explained. In contrast, Lossy Counting runs as fast as DBCA at larger values of *ms* but its run-time gets longer as smaller *ms* are given. This fact is due to the increasing number of handled itemsets resulting from the *ms*-dependent parameter ε. Second, the processing acceleration mechanism is useful for reducing the processing time of DBCA on the same data. When the mechanism starts the second stage of acceleration (i.e., decreases k to 3), the run-time of DBCA is about 75% to that for the default base size. In other words, the processing rate (or throughput) is raised by about 33%. Third, as the value of *ms* becomes smaller, DBCA, especially with the processing acceleration mechanism, outperforms the other methods. If there exists data overload or a limit on run-time (given by the user), DBCA is most possible to properly manage with the situation. On the other hand, from Fig. 9(b) it is found that the mining accuracy of DBCA, especially with the processing acceleration mechanism, is slightly lower than that of the other methods at low *ms* values. However, its average score of *F*-measure is greater than 95%, which represents pretty high accuracy.

The experimental results on the other datasets are shown in Fig. 10. In these experiments, if the average transaction length of a dataset is n, the default base size of DBCA is set to $k = \lceil\sqrt{n}\rceil + 2$, and $k = \lceil\sqrt{n}\rceil + 1$ and $k = \lceil\sqrt{n}\rceil$ are respectively the *first* and *second stages* of reduction in base size when the processing acceleration mechanism is executed. In Fig. 10 the facts similar to those we have stated for Fig. 9 (in the previous paragraph) can also be observed. Note that the DSTree method, although possesses exact mining results, its performance is much poor than that of DBCA, especially when running on a data source whose average transaction length is long. Moreover, its memory consumption is extremely large; on the *BMS-POS* dataset, DSTree fails to finish the mining operation because the available memory space runs out soon after it starts processing the dataset. Also note that Lossy Counting performs badly under low *ms* settings, which is due to the parameter ε, because both the number of itemsets and that of lengths of itemset to be processed have greatly increased. In contrast, DBCA has to process only the first few orders of itemsets with some base threshold β under any *ms* setting. We remark that the effect of the processing acceleration mechanism is quite evident. Depending on the feature of data source which DBCA runs on, the run-time may get a reduction of 15–40% with the first-stage acceleration, or a reduction of 30–60% with the second-stage acceleration. The mining accuracy, on the other hand, degrades gradually (at a reasonable cost) as reduced values of base size are set. It is verified from this experiment that DBCA has stable and efficient performance of stream-data processing. Moreover, the processing acceleration mechanism can actually improve the throughput of DBCA and thus address the data overload, at a price of slightly diminished mining accuracy.

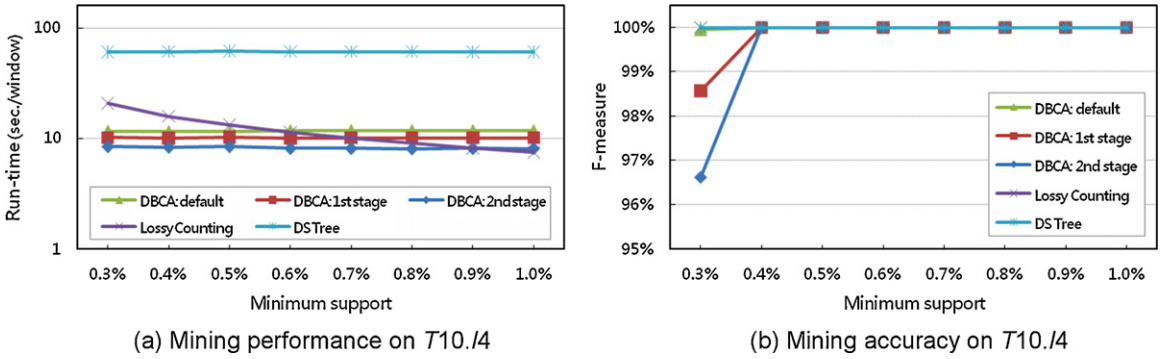


Fig. 9. Experimental result on the *T*10.*I*4 dataset: (a) mining performance on *T*10.*I*4; (b) mining accuracy on *T*10.*I*4.

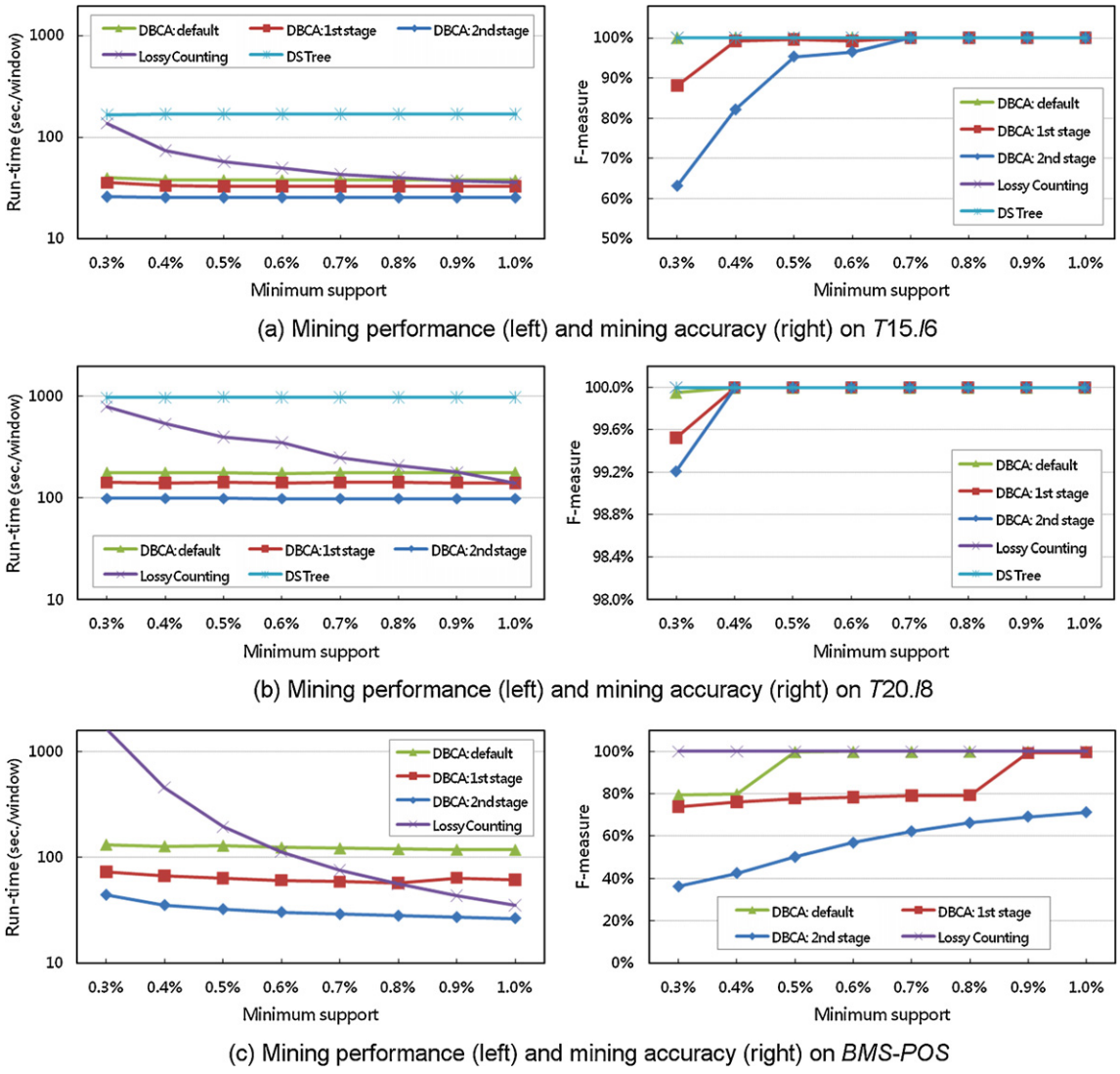


Fig. 10. Experimental results on the *T*15.*I*6, *T*20.*I*8, and *BMS-POS* datasets.

5.2. Experiment II: DBCA with data-load shedding

In the second experiment, we evaluate the performance of DBCA with the load shedding mechanism during data overloading situations. The three proposed policies for the mechanism are separately tested and then compared together. Both the normal DBCA (that is, without load-shedding operation) and Lossy Counting (which has no load-shedding function) are also tested as the *control group* for confirming the effects of load shedding.

The performance is tested at different data-load degrees, namely from 80% (i.e., normal and is about to be fully loaded) up to 150% (i.e., highly overloaded) with 10% intervals. To test on different data loads, the setting of sliding-window size and segment size in this experiment differs slightly from that in the previous experiment. The segment size is *variable* and is dependent on data load. More specifically, the segment size has a basic quantity of 10k and is of $L \times 10$k under a data load of L. For example, for data loads of 80%, 100%, and 120%, the corresponding segment sizes are 8k, 10k, and 12k, respectively. Thus, the size of the sliding window (which contains 10 segments) varies accordingly. For a data load of over 100%, i.e., overload, the load shedding mechanism, if any, will first shed the load to be 100% (or under). To agree with the definition of data load given in Section 3, the value of z of DBCA is also set by 10k. Therefore, a data load of L means that a batch of $L \times 10$k transactions is received in a time-unit, i.e., $w = L \times 10$k.

The value of *ms* in this experiment is set by 0.3%. For each testing dataset with an average transaction length of n, the default base size of DBCA on the dataset is consistently set for $k = \lceil \sqrt{n} \rceil + 1$; this value is nothing to do with the load shedding mechanism. For each incoming batch (of stream elements), if the mining system is overloaded, in the first place the load shedding mechanism is started to prune the overloading data, then the mining algorithm processes on the trimmed data. As a result, the run-time consists of two parts, namely *load shedding operation* and *data processing operation* (namely base-information keeping, for DBCA). We remark that if the load shedding mechanism is effective, the total run-time spent on load shedding (which trims the overloading data) and data processing (which processes the abridged data) will be shorter than that consumed by processing directly on the full incoming data. Besides, the run-time at 100% of data load can be regarded as an index of efficiency of a load-shedding policy (at an over-100% data load).

In this experiment, the *T*10.*I*4 dataset is excluded due to its shorter average length of transactions. We show the experimental result on the *T*15.*I*6 dataset first in Fig. 11. The dashed line in the performance figure represents a *time constraint* for reference. For

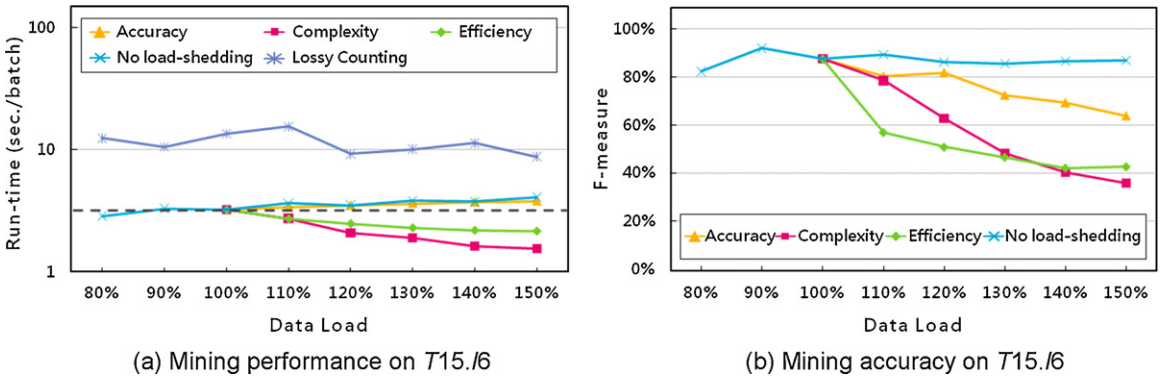


Fig. 11. Experimental result on the *T*15.*I*6 dataset: (a) mining performance on *T*15.*I*6; (b) mining accuracy on *T*15.*I*6.

a data load under 100% load shedding is unnecessary and thus the mechanism does not operate. It is found from Fig. 11(a) that the Lossy Counting performs quite worse than the proposed method across different data loads. Lossy Counting needs to process and maintain a rather larger amount of itemsets, and it has no load-shedding function, after all, so the run-time is much longer. Due to this fact, in the accuracy figure, we exclude it and only show and compare the data of DBCA with the load shedding mechanism.

Regarding DBCA, as can be observed from Fig. 11(a), if the load shedding mechanism is disabled, its run-time will gradually grow longer as the data load becomes heavier. In contrast, if the load shedding mechanism is enabled, the run-time of DBCA is shortened, regardless of the policy being adopted. Both the complexity- and efficiency-oriented policies lead to the run-time which is not only shorter than that of the no load-shedding approach, but even adequately satisfying the time-constraint line. The complexity-oriented policy eliminates the existence of low-frequency attributes, so the number of itemset combinations (in the trimmed data) needing to be extracted is greatly reduced. In the case of the efficiency-oriented policy, transactions with longer lengths are discarded first, so the trimmed data includes shorter transactions and thus can be fast processed. The accuracy-oriented policy, although not so evident when compared with the above two policies, still outperforms the no load-shedding approach in terms of run-time. Moreover, the mining accuracy of this policy is the highest across all overload degrees among the three policies, which can be found from Fig. 11(b).

The experimental results on the other datasets are shown in Fig. 12 where the dashed line in the performance figure represents a reference time-constraint. Briefly, the facts which are noticed from the result on *T*15.*I*6 are similarly observed. First, Lossy Counting totally fails to satisfy the time constraint at every data load (for the reason that we have explained earlier). Secondly, the three load-shedding policies are all effective against data overload, that is to say, they could reduce the run-time of directly processing on an overloading transaction batch. Thirdly, both the complexity- and

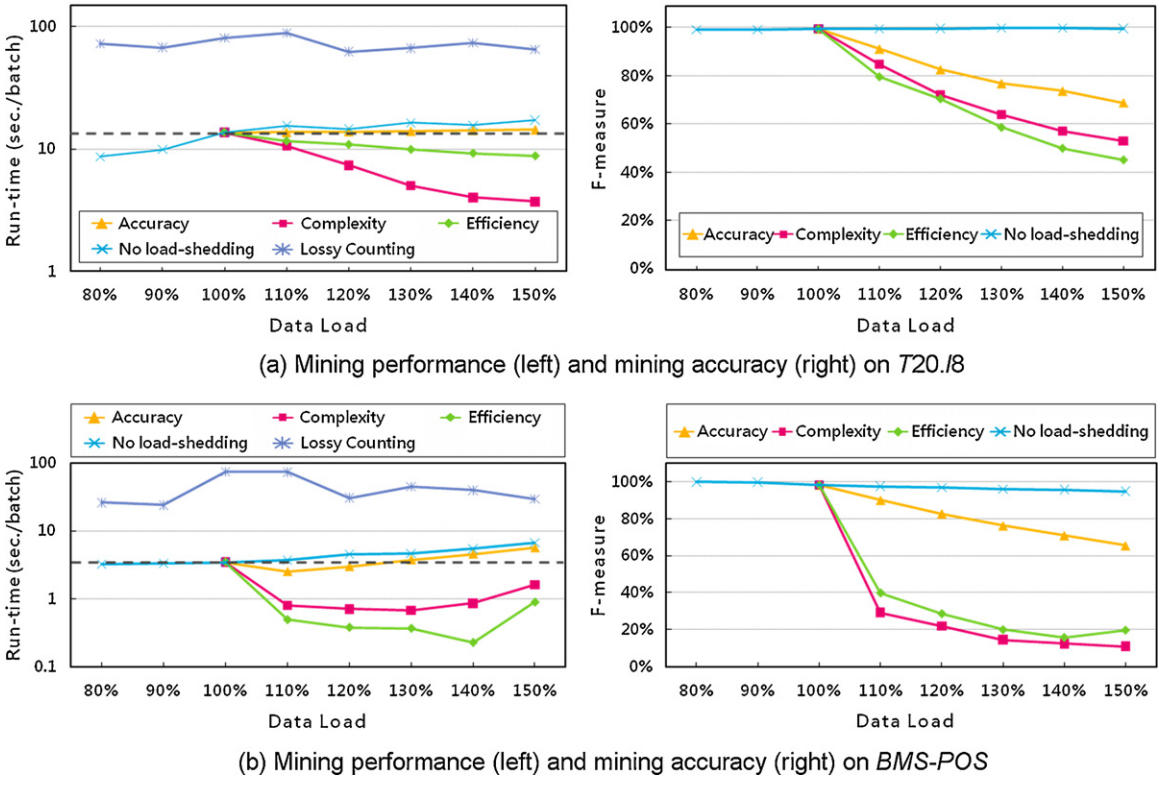


Fig. 12. Experimental results on the *T*20.*I*8 and *BMS-POS* datasets.

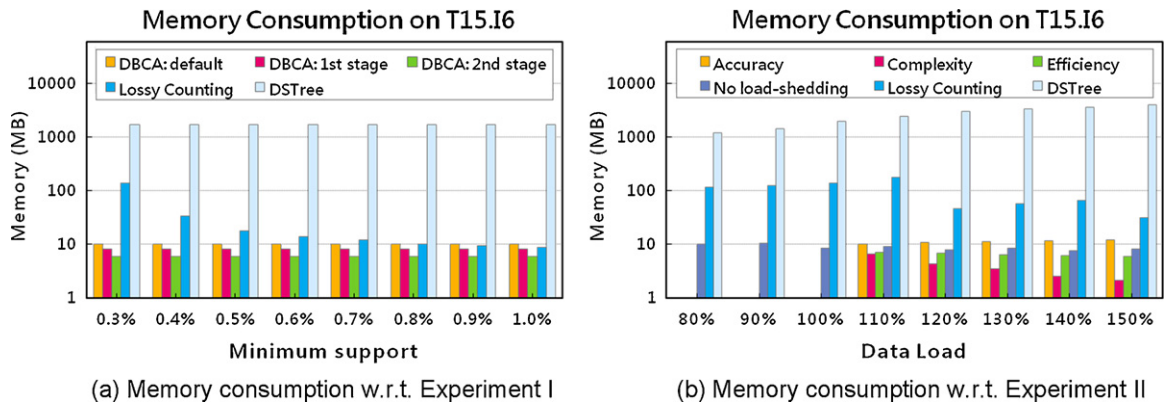


Fig. 13. Experimental results of memory consumption: (a) memory consumption on Experiment I; (b) memory consumption on Experiment II.

efficiency-oriented policies could substantially shorten the run-time of DBCA as the data load becomes heavier, since the workload of itemset enumeration on the trimmed data is highly lightened. Fourthly, the accuracy-oriented policy best preserves the mining accuracy when the mining system is data overloaded, since its design is on the basis of the accuracy-improving technique namely *counts bounding*. Across the tested datasets, the proposed method can work at least up to 150% of data overload. The performance of DBCA with the load shedding mechanism under data overload is well verified by this experiment.

5.3. Experiment III: memory-space consumption and load-shedding time

In addition to the running performance and mining accuracy, we have made an experiment on the memory consumption of DBCA because space efficiency is also an important factor in data-stream mining. The experiment includes two parts; the first part is about DBCA with processing acceleration and the second part is about DBCA with data-load shedding. Fig. 13 shows the experimental result on the *T*15.*I*6 dataset. Here we use *T*15.*I*6 as the representative dataset; on the other datasets listed in Table 3 similar results are obtained. First of all, it is observed that DBCA (regardless of the overload-handling mechanisms) consumes comparatively stable and less memory space. DSTree stores all itemsets across all possible orders; Lossy Counting keeps itemsets of numerous orders (depending upon *ms* and ε); DBCA monitors itemsets of only the first k orders (according to β). Because the memory consumption mostly depends on the number of itemsets being maintained (in an algorithm's data structure), DBCA possesses good space efficiency. Furthermore, both of the overload-handling mechanisms may even lower the memory consumption of DBCA. As for the processing acceleration mechanism, because fewer itemsets are kept with a reduced base-information size, the memory usage is decreased. Regarding the load shedding mechanism, especially with the complexity-oriented policy, because fewer pattern combinations occur in the trimmed data, the space consumption is decreased.

Finally, we give an evaluation of the efficiency of the load-shedding policies. The time for load shedding on the *T*20.*I*8 dataset is detailed in Table 4. Among the three proposed policies, the efficiency-oriented policy, as its name indicates, carries out the load shedding operation most efficiently. The main reasons include that this policy needs no feedback information (from previous batches) and its priority function is relatively simple. Nevertheless, the greatly reduced number of pattern combinations caused by *attribute elimination* brings a very short data processing time, which is the reason for that the complexity-oriented policy may result in an even faster run-time on some datasets like *T*20.*I*8 (as shown in Fig. 12(a)), although both the policies are very effective means for run-time reduction. Besides, it is noted that the accuracy-oriented policy possesses comparable high-efficiency in load shedding to the efficiency-oriented policy, which is convincing evidence that the fast calculation of *MPN* (of potential frequent 2-subsets) is effective. Objectively speaking, the three policies are all capable of trimming the overloading data rapidly because overall, the load-shedding time accounts for less than 10% of the total run-time.

6. Discussions on the overload-handling mechanisms

In this section we discuss several issues mainly concerning the two proposed mechanisms for overload management, namely processing acceleration and data-load shedding.

6.1. Features of the mechanisms

The DBCA algorithm extracts base information from data streams in a dynamic way. More specifically, it keeps base information on a data stream D with the size concerning the average length n of transactions in D.

With the processing acceleration mechanism, data overload is addressed by shortening the processing time (or increasing the processing rate) on the same input data. In general cases the threshold-value namely $\lceil\sqrt{n}\rceil$ is much smaller than the value of n. Therefore a base size k which is set to be slightly larger than $\lceil\sqrt{n}\rceil$ is feasible. As a result, the default value of base size is set with some releasable space inside. Such a setting of k permits the *downgrading function* of the base scale, from the default level to a lower level, for data processing acceleration against data overload. Depending on the default value of base size, the mechanism is possible to launch *multiple stages* of acceleration. In our opinion, a default base size which enables a 2 or 3 stages of processing acceleration, i.e., $k = \lceil\sqrt{n}\rceil + 2$ or $k = \lceil\sqrt{n}\rceil + 3$, is suitable for general data streams, as have been observed from the experimental results.

On the other hand, with the load shedding mechanism, data overload is addressed by pruning the excess of incoming data and dealing only with the trimmed data, not by processing on the full amount of incoming data. Depending on how overloading data can be trimmed, there may be various policies on load shedding, and we have described three such policies. The proposed policies, although possess different properties, have all been verified by the experiment to be effective.

We remark that for the load shedding mechanism, according to different aims, there are still possible policies to the mechanism other than the described three ones. In addition, while the processing acceleration mechanism (Section 4.2) is a *closed* approach

Table 4
Load-shedding time of the load-shedding policies on the *T*20.*I*8 dataset.

Policy	Data load					Average time	% in run-time
	110%	120%	130%	140%	150%		
Efficiency-oriented	0.016 s	0.016 s	0.016 s	0.016 s	0.016 s	0.016 s	0.02%
Complexity-oriented	0.896 s	1.656 s	2.427 s	3.130 s	3.844 s	2.391 s	5.21%
Accuracy-oriented	0.011 s	0.014 s	0.021 s	0.025 s	0.028 s	0.020 s	0.01%

which is limited to the DBCA algorithm only, the load shedding mechanism (Section 4.3) is possibly thrown *open* to the other mining algorithms. As long as the priority function and priority table are properly designed following some policy, the mechanism can actually handle data overload while achieving the effect(s) defined by the aim of the policy.

6.2. Choices among the load-shedding policies

There are three policies on load shedding proposed in the paper; here we give some guidelines on the selection of the policies. The efficiency- and complexity-oriented policies possess significant effects on reducing the data load and run-time. If the user requirement is to rapidly reduce the excessive load, or there is a *time constraint* on data processing, then either of the two policies is recommended. In contrast, the accuracy-oriented policy possesses a soft effect on run-time reduction, yet it maintains the mining accuracy (on the trimmed data) at a relatively high and acceptable level. If the user requirement is to preserve the mining quality, this policy is suggested to both shed the data load and make the mining accuracy reasonable.

As for the issue of the policy that best suits a mining system, the prime choice is dependent on the environment where the mining system is located. The reason is that different applications have different priorities of quality/performance requirements. We further give some suggestions for choosing an appropriate policy (in terms of application environment). In an application where the quality of mining outcome is highly desirable, the accuracy-oriented policy is most suitable. On the other hand, in an environment where hardware resources are *limited*, for example, shared by several data-stream analysis systems, the complexity-oriented policy is preferred. In addition, if in a data stream longer elements are more likely to represent patterns of *each individual case*, then the efficiency-oriented policy would be the fittest.

6.3. Other related issues

The DBCA algorithm has two mechanisms for overload handling – either is applicable during data overload. It is possible for DBCA to move from one mechanism to the other mechanism in specific situations, as a matter of fact. Here we give a discussion about the occasion for changing from processing acceleration to load shedding. The processing acceleration mechanism raises the throughput of DBCA by accessing fewer itemsets in length terms. If elements in a data stream are averagely shorter in length, the effect of processing acceleration would become less obvious. In such a case, the overload management should switch to the load shedding mechanism for possibly better performance.

Our discussions so far are given around the main point of *efficiency*, because DBCA is designed for data-stream mining with overload management, where efficiency should play a leading role. At the last of this section, we discuss the issue regarding taking *mining quality* the first priority. As a specific case, we consider how the proposed method may be adjusted in order to achieve a user-defined accuracy rate as well as possible. The proposed method calculates the approximate counts of itemsets and then uses the counts to determine frequent patterns. On the other hand, the accuracy rate namely *F*-measure is measured mainly by the number of *false answers* in the mining outcome. The smaller the error in approximate counts is, the smaller the number of false answers becomes, thus the higher the accuracy rate would be. Therefore, to fulfill the requirement, techniques for more exact approximations to the frequency count are desired. A main challenge is that if we could exclude the irrelevant portion such as the noise of the base information during the approximating process. To meet the challenge, further improvement on the *counts-bounding technique* (i.e., limiting itemsets' counts by their supersets' counts) proposed by Jea and Li (2009) might be a feasible means.

7. Conclusions

In real-life data streams, the data transmission rate usually varies with time; at times it may beyond the data processing capability of a mining algorithm/system, which leads to the serious situation of *data overload*. A key contribution of this paper is to provide users with a feasible solution to frequent-pattern discovery in dynamic data streams which are prone to data overload. More specifically, a count approximation-based mining algorithm, *DBCA*, together with two dedicated mechanisms for overload management, is described and proposed. The DBCA algorithm extracts synopsis from received stream elements as base information, and carries out the mining task when requested through an approximating process. One remarkable feature of DBCA is that it can maintain dynamical-sized base information according to different (batches of) received data. The two mechanisms, namely *processing acceleration* and *data-load shedding*, are prepared for possible data-overload situations. By running one of the mechanisms when the buffer is overloaded with data, DBCA could cope with the overload by means of either increasing its throughput or decrease the data volume.

Different from the existing related work concerning frequent-pattern mining in data streams, our research pays attention to the issue of *data-overload handling* as well as that of *frequent-itemset discovery*. The processing acceleration mechanism raises the data processing rate of DBCA through *scaling* the size of base information. Depending on the required data-processing rate of a data source or an application, the mechanism could improve the throughput of DBCA by tuning the value of base size (i.e., k) progressively lower. On the contrary, the data-load shedding mechanism trims overloading data by discarding some elements in the data. This mechanism is *priority-based* and it discards elements whose priority-values are low according to the priority function employed. We have also described three practical policies with respective aims for the mechanism to adopt. From the experimental data, we found that DBCA has stable and high performance. More importantly, both mechanisms possess a successful effect on overload handling.

To the best of our knowledge, except the preliminary try had by Jea et al. (2010), this paper is the first one which gives a formal and detailed study of overload management in data-stream frequent-pattern mining. As a further step, our future work may include devising other possible mechanism(s) for data-overload management. We remark that overload handling such as load shedding is necessary for data-stream mining systems but only an *emergency measure* after all. If a system is always overloaded with

incoming data, then a more powerful algorithm or machine may be preferable to any overload-handling approach. Nevertheless, so long as data overload is not the common case, that is, it may take place *sometimes* but *not all the time*, a mining system with our load-controllable method could operates normally under different data-load degrees. In other words, the system will possess the nice feature of *durability*.

References

Agrawal, R., Srikant, R., 1994. Fast algorithms for mining association rules. In: Proceedings of the 20th Conference on Very Large Data Bases, pp. 487–499.

Babcock, B., Datar, M., Motwani, R., 2003. Load shedding techniques for data stream systems. In: Proceedings of the 2003 Workshop on Management and Processing of Data Streams, a short paper without page number.

Chang, J.H., Lee, W.S., 2004. A sliding window method for finding recently frequent itemsets over online data streams. Journal of Information Science and Engineering 20 (4), 753–762.

Charikar, M., Chen, K., Farach-Colton, M., 2004. Finding frequent items in data streams. Theoretical Computer Science 312 (1), 3–15.

Chi, Y., Yu, P.S., Wang, H., Muntz, R.R., 2005. Loadstar: a load shedding scheme for classifying data streams. In: Proceedings of the 5th SIAM Conference on Data Mining, pp. 346–357.

Fang, M., Shivakumar, N., Garcia-Molina, H., Motwani, R., Ullman, J.D., 1998. Computing iceberg queries efficiently. In: Proceedings of the 24th Conference on Very Large Data Bases, pp. 299–310.

Frequent Itemset Mining Implementations Repository (FIMI). http://fimi.cs.helsinki.fi/.

Jea, K.-F., Li, C.-W., 2009. Discovering frequent itemsets over transactional data streams through an efficient and stable approximate approach. Expert Systems with Applications 36 (10), 12323–12331.

Jea, K.-F., Li, C.-W., Hsu, C.-W., Lin, R.-P., Yen, S.-F., 2010. A load-controllable mining system for frequent-pattern discovery in dynamic data streams. In: Proceedings of the 9th Conference on Machine Learning and Cybernetics, pp. 2466–2471.

Leung, C.K.-S., Khan, Q.I., 2006. DSTree: a tree structure for the mining of frequent sets from data streams. In: Proceedings of the 6th Conference on Data Mining, pp. 928–932.

Linial, N., Nisan, N., 1990. Approximate Inclusion–Exclusion. Combinatorica 10 (4), 349–365.

Li, C.-W., Jea, K.-F., 2011. An adaptive approximation method to discover frequent itemsets over sliding-window-based data streams. Expert Systems with Applications 38 (10), 13386–13404.

Li, H.-F., Lee, S.-Y., 2009. Mining frequent itemsets over data streams using efficient window sliding techniques. Expert Systems with Applications 36 (2), 1466–1477.

Liu, C.L., 1968. Introduction to Combinatorial Mathematics. McGraw-Hill, New York.

Manku, G.S., Motwani, R., 2002. Approximate frequency counts over data streams. In: Proceedings of the 28th Conference on Very Large Data Bases, pp. 346–357.

Tanbeer, S.K., Ahmed, C.F., Jeong, B.-S., Lee, Y.-K., 2009. Sliding window-based frequent pattern mining over data streams. Information Sciences 179 (22), 3843–3865.

Zhu, Y., Shasha, D., 2002. StatStream: statistical monitoring of thousands of data streams in real time. In: Proceedings of the 28th Conference on Very Large Data Bases, pp. 358–369.

Chao-Wei Li received the B.S. degree from the Department of Information Engineering and Computer Science, Feng Chia University, Taiwan, in 2007. He is currently pursuing his Ph.D. degree at the Department of Computer Science and Engineering, National Chung-Hsing University, Taiwan, and under the supervision of Prof. Kuen-Fang Jea. His research interests include data mining, data stream system, and data stream mining.

Kuen-Fang Jea received his Ph.D. degree in Computer Science and Engineering from the University of Michigan at Ann Arbor in 1989. He is currently a professor in the Department of Computer Science and Engineering, National Chung-Hsing University (NCHU), Taiwan. Before joining NCHU, he was with Bellcore (Bell Communication Research) in USA. His major research interests include XML database system, database performance, data mining, data stream system and cloud computing. Dr. Jea is a member of Eta Kappa Nu and the IEEE Computer Society. He is also on the editorial board of Int. J. of Knowledge-Based Organizations (IJKBO).

Ru-Ping Lin received the B.S. degree from the Department of Information & Computer Engineering, Chung Yuan Christian University, Taiwan, in 2009. Under the supervision of Prof. Kuen-Fang Jea, she received the M.S. degree from the Department of Computer Science and Engineering, National Chung-Hsing University, Taiwan, in 2011. Her research interests include data mining and data stream system.

Ssu-Fan Yen received the B.S. degree from the Department of Computer and Information Science, National Taichung University, Taiwan, in 2009. Under the supervision of Prof. Kuen-Fang Jea, she received the M.S. degree from the Department of Computer Science and Engineering, National Chung-Hsing University, Taiwan, in 2011. Her research interests include data mining and data stream system.

Chih-Wei Hsu received the B.Math degree from the Department of Applied Mathematics, National Chung-Hsing University (NCHU), Taiwan, in 2006, and the M.S. degree from the Department of Computer Science and Engineering at NCHU in 2008. He is currently pursuing his Ph.D. degree at NCHU, and under the supervision of Prof. Kuen-Fang Jea. His research interests include data mining and cloud computing.