The VLDB Journal (2008) 17:1321–1344
DOI 10.1007/s00778-007-0078-6

REGULAR PAPER

Mining *top-k* frequent patterns in the presence of the memory constraint

Kun-Ta Chuang · Jiun-Long Huang · Ming-Syan Chen

Received: 16 January 2006 / Revised: 11 March 2007 / Accepted: 8 August 2007 / Published online: 7 November 2007

Abstract We explore in this paper a practicably interesting mining task to retrieve *top-k* (*closed*) itemsets in the presence of the memory constraint. Specifically, as opposed to most previous works that concentrate on improving the mining efficiency or on reducing the memory size by best effort, we first attempt to specify the available upper memory size that can be utilized by mining frequent itemsets. To comply with the upper bound of the memory consumption, two efficient algorithms, called *MTK* and *MTK_Close*, are devised for mining frequent itemsets and *closed* itemsets, respectively, *without* specifying the subtle minimum support. Instead, users only need to give a more human-understandable parameter, namely the desired number of frequent (*closed*) itemsets k. In practice, it is quite challenging to constrain the memory consumption while also efficiently retrieving *top-k* itemsets. To effectively achieve this, *MTK* and *MTK_Close* are devised as level-wise search algorithms, where the number of candidates being generated-and-tested in each database scan will be limited. A novel search approach, called *δ-stair search,* is utilized in *MTK* and *MTK_Close* to effectively assign the available memory for testing candidate itemsets with various itemset-lengths, which leads to a small number of required database scans. As demonstrated in the empirical study on real data and synthetic data, instead of only providing the flexibility of striking a compromise between the execution efficiency and the memory consumption, *MTK* and *MTK_Close* can both achieve high efficiency and have a constrained memory bound, showing the prominent advantage to be practical algorithms of mining frequent patterns.

K.-T. Chuang (✉) · M.-S. Chen
Department of Electrical Engineering,
National Taiwan University, Taipei, Taiwan, ROC
e-mail: doug@arbor.ee.ntu.edu.tw

M.-S. Chen
e-mail: mschen@cc.ee.ntu.edu.tw

J.-L. Huang
Department of Computer Science,
National Chiao Tung University, Hsinchu, Taiwan, ROC
e-mail: jlhuang@cs.nctu.edu.tw

1 Introduction

The discovery of frequent relationship among a huge database has been known to be useful in selective marketing, decision analysis, and business management [14]. A popular area of its applications is the market basket analysis, which studies the buying behaviors of customers by searching for sets of items that are frequently purchased together. Specifically, let $\mathcal{I} = \{x_1, x_2, \ldots, x_m\}$ be a set of items. A set $X \subseteq \mathcal{I}$ with $m = |X|$ is called a m-itemset or simply an itemset. Formally, an itemset X refers to a *frequent* itemset or a *large* itemset if the support of X, i.e., the fraction of transactions in the database that contain X, is larger than the *minimum support* threshold, indicating that the presence of itemset X is significant in the database.

However, it is reported that discovering frequent itemsets suffers from two inherent obstacles, namely, (1) the subtle determination of the minimum support [22]; (2) the unbounded memory consumption [11]. Specifically, without specific knowledge, a critical problem "*What is the appropriate minimum support?*" is usually left unsolved to users in previous works. Note that setting the minimum support is quite subtle since a small minimum support may result in an extremely large size of frequent itemsets at the cost of execution efficiency. Oppositely, setting a large minimum support may only generate a few itemsets, which cannot provide enough information for marketing decisions. In

order to obtain a desired result, users in general need to tune the minimum support over a wide range. This is very time-consuming and indeed is a serious problem for the applicability of mining frequent itemsets. Furthermore, another issue which will be faced in practice is the large memory consumption. A large memory, which may not be affordable in most personal computers nowadays, is in general required during the mining process, especially when the minimum support is small or the database size is large. It will result in the serious "out of memory" system crash, making users shy away from executing the frequent itemset mining. Note that users may tolerate to mine frequent itemsets off-line. For example, frequent itemsets can be discovered in every night as long as users are able to make their marketing decisions in the morning. In contrast, the system crash due to the "out of memory" error is repulsive in a commercial mining system.

To remedy the first problem, recent research advances in data mining call for the need to discover *top-k* frequent patterns without the minimum support specification [6,22]. The *top-k* frequent patterns refer to the k most frequent itemsets in the database. As opposed to specify the subtle minimum support, users will only need to give the desired count of frequent itemsets, which is indeed a more human-understandable parameter. For example, to make marketing decisions, users may be interested in less than 10,000 frequent itemsets. Hence they can easily give the number of frequent itemsets k equal to 10,000 and further mine *top-k* frequent itemsets. More specifically, instead of mining *top-k* frequent itemsets, the work in [22] aimed to discover *top-k closed* itemsets whose lengths exceed a specified threshold. Under such specific constraint, the FP-tree can be constructed with several pruning strategies such as omitting transactions whose lengths are less than the specified itemset-length. Moreover, the work in [6] initially constructs a complete FP-tree in memory, and then retrieves *k most frequent l*-itemsets, where l lies in a range specified by users. In addition, a recent work in [1] studied a post-processing approach to determine the k patterns that best approximate all frequent itemsets discovered. However, its concept inherently deviates far from discovering *top-k* frequent patterns, since its objective is to approximately describe the set of frequent itemsets and the minimum support still needs to be specified in advance.

The second problem of mining frequent patterns, i.e., the unbounded memory consumption, has been discussed in the direction of reducing the required memory by means of compressed structures or skillful search approaches [4,10,17,19]. Recently, the issue has also received a great deal of attention in mining data streams [7,15,23,25]. Since a large memory consumption is prohibitive in streaming environments, we have to discover frequent patterns within an estimated memory upper bound at the cost of the resulting precision [3]. For example, the solution in [25] empirically derived its memory upper bound of $O(\frac{1}{s^3})$, where s is the specified minimum support. Formally, same as traditional algorithms such as *Apriori* [2] and *FPGrowth* [12], the applicability of these approaches is valid based on the premise that the required memory can be unconditionally provided by the system. However, it is improbable and the "out of memory" system crash is still likely to happen while the minimum support is small or the data distribution is quite dense. Recent studies in frequent-pattern mining have pointed out that most previous works were optimized for efficiency at the cost of the memory space, and thus their scalability will need further justification [11]. In practice, a desirable research direction is to allow that the available memory upper bound, say 100 or 200 MB, can be specified by system designers. Mining frequent patterns under the specified bound of the memory consumption is referred to as the "*memory-constraint frequent-pattern mining*" in this paper. Despite of its great applicability, how to realize the *memory-constraint frequent-pattern mining* is however not fully explored thus far.

To enable the better feasibility of mining frequent patterns, we examine in this paper the problem of discovering *top-k* frequent patterns, coupled with the need of the *memory-constraint mining*. The goal is desirable but is quite challenging. Note that previous works of mining *top-k* frequent patterns [6,22] need to be executed by initially building a complete FP-tree in memory. It is clear that the memory problem will be worse than the traditional frequent itemset mining since the size of the in-memory FP-tree is solely proportional to the entire database size.[1] Although we can implement the disk-based *FPGrowth* algorithm [13] to ensure the complete FP-tree can be constructed, it has been reported that the disk-based implementation is much inefficient as compared to the memory-based implementation since the I/O swap will drastically degrade the mining performance [10]. Furthermore, previous works of mining *top-k* frequent itemsets only concentrate on mining special itemsets such as closed itemsets [22] or itemsets with the specified long itemset-length [6], because some heuristic strategies to reduce the search space can be applied. For example, as mentioned above, the FP-tree can be constructed by omitting transactions whose lengths are less than the specified itemset-length, which helps to reduce the size of the FP-tree and makes the search more efficient [22]. However, those pruning techniques will be no longer valid in the general model of mining pure *top-k* frequent itemsets. Mining *top-k* frequent itemsets without any constraint of item types or itemset-lengths are referred to as mining pure *top-k* frequent patterns in this paper. The naive extensions of previous solutions to discover pure *top-k* frequent itemsets

[1] One may suggest to apply sampling prior to mine *top-k* frequent itemsets at the cost of resulting precision. However, for obtaining a consistent mining result, the space to store the complete FP-tree is still unbounded since all itemset combinations remain in the tree (note that, what changed after the unbiased sampling is the frequency of itemsets rather than the tree structure).

will not only lead to inefficiency but also face more serious problem of the memory bottleneck. Note that determining the minimum itemset-length incurs another inconvenience to users, which conflicts the purpose to release users from the determination of subtle parameters. In addition, in many real applications such as retail applications, mining pure *top-k* frequent itemsets is equally or more important than mining *top-k* itemsets with long itemset-lengths (users may not be interested in long itemsets since they usually attempt to cross-sell two or three products as opposed to one hundred products [14]).

Actually, as we can imagine, even though the memory space is not affordable in a PC nowadays, the memory issue will become insignificant in a server-level machine in the near future. In addition, the minimum support requirement may be determined by a domain expert without much effort. However, in our consideration, the data mining functionality is not a patent owned by few people. It is worth providing an easily deployed solution to mine association rules everywhere and every time in such a way that the visibility and usability of the mining capability can be broadened to more users with a PC in hand.

As a consequence, we propose in this paper efficient solutions, called *MTK* (standing for the *M*emory-constraint *top-k* frequent-pattern mining) and *MTK_Close*, to discover pure *top-k* frequent patterns and *top-k closed* patterns, respectively, in the presence of the memory constraint. Since our goal is to release users from the burden of setting subtle parameters and to provide better flexibility in various applications, the itemset-length constraint is not imposed on our model. Note that FP-tree based solutions intrinsically cannot be *memory-constraint frequent-pattern mining* approaches since the size of an in-memory FP-tree is proportional to the database size. As such, we devise *MTK* and *MTK_Close* as level-wise based algorithms as analogous to *Apriori* [2], *DHP* [19], and *DIC* [4]. In practice, level-wise based algorithms generate a potentially huge set of candidate itemsets which may not fit in memory. It also leads to a large memory requirement. To remedy this, we devise an efficient search approach in *MTK* and *MTK_Close*, called the *δ-stair search*, to limit the number of candidates which are generated-and-tested in each database scan. Specifically, the *δ-stair search* assigns the available memory to concurrently generate candidates with consecutive itemset-lengths. Using the *δ-stair search* will lead to a small number of database scans which are required to retrieve the set of *top-k* frequent itemsets, to ensure the memory usage can be constrained without comprising the execution efficiency. More importantly, the *MTK* algorithm even requires a smaller number of database scans than traditional approaches with an unbounded memory usage. This is attributed to that the *δ-stair search* can effectively utilize the memory to test candidates which are highly potential to be included in *top-k* frequent itemsets. In addition, the high efficiency also comes from that the *MTK* and *MTK_Close* algorithms are sophisticatedly designed to fully integrate with many skillful techniques proposed in the literature, such as the scan-reduction technique [2,20] and the hash-indexing technique [19] (readers can refer [9] for the detailed survey and comparison of these optimizations). As such, the *MTK* and *MTK_Close* algorithms cannot only comply with the memory constraint but also retrieve *top-k* frequent/closed itemsets with high efficiency.

The contribution of this paper can be summarized as follows: (1) while previous works on mining frequent patterns mostly concentrate on improving the mining efficiency or on reducing the memory size by best effort, we further investigate in this paper the important issue of mining frequent/closed itemsets in the presence of the explicit memory constraint. (2) While previous works on mining *top-k* frequent patterns aimed to discover special *top-k* patterns, we propose the *MTK* algorithm to mine pure *top-k* frequent itemsets, and devise its extension to mine *top-k closed* itemsets, to provide better flexibility of mining frequent patterns for various applications. (3) We complement our analytical and algorithmic results by a thorough empirical study on real data and synthetic data, and show that *MTK* and *MTK_Close* can retrieve *top-k* itemsets and *top-k closed* itemsets with high efficiency even though the memory usage is constrained. The result demonstrates that, instead of only providing the flexibility of striking a compromise between the execution efficiency and the memory consumption, *MTK* and *MTK_Close* can both achieve high efficiency and have a constrained memory bound, showing their prominent advantages to be practical algorithms of mining frequent patterns.

This paper is organized as follows. Section 2 introduces the problem description and gives a baseline approach to discover frequent itemsets with the memory constraint. In Sect. 3, we give the design of the *δ -stair search* to retrieve *top-k* frequent itemsets. The implementation of the *MTK* and *MTK_Close* algorithms are presented in Sect. 4. Section 5 shows the experimental results. Finally, this paper concludes with Sect. 6.

2 Memory-constraint frequent-pattern mining

In Sect. 2.1, we formally specify the problem we study in this paper. In Sect. 2.2, we introduce a baseline approach, referred to as the *Naive* algorithm, to discover frequent patterns in the presence of the memory constraint.

2.1 Problem description

We first introduce the notations used hereafter. For ease of exposition, in the sequel, pure *top-k* frequent itemsets will be simply denoted by *top-k* frequent itemsets, to distinguish

from *top-k closed* itemsets without ambiguity. Suppose that $\sup(X)$ denotes the support [2] of itemset X in the database D. We give several necessary definitions as follows:

Definition 1 (*top-k* frequent itemsets) Given the desired number of frequent itemsets k, an itemset X is a *top-k* frequent itemset in D if there are less than k itemsets[3] whose supports are larger than $\sup(X)$. Let T_k denote the set of all *top-k* frequent itemsets. The minimum support to retrieve T_k will be

$$\sup_{\min}(T_k) = \min\{\sup(X) \mid X \in T_k\}.$$

Definition 2 (closed itemsets) An itemset X is referred to as a closed itemset if there exists no itemset X' that (1) $\sup(X) = \sup(X')$; and (2) $X \subset X'$ [26].

Definition 3 (*top-k* closed itemsets) Given the desired number of closed itemsets k, an itemset X is a *top-k* closed itemset in D if there are less than k closed itemsets whose supports are larger than $\sup(X)$. Let TC_k denote the set of all *top-k* closed itemsets. The minimum support to retrieve TC_k will be

$$\sup_{\min}(TC_k) = \min\{\sup(X) \mid X \in TC_k\}.$$

Furthermore, let an itemset containing j items be referred to as a j -itemset. We then have Definition 4 below:

Definition 4 An itemset, denoted by $X_{j,m}$, is the mth most frequent j-itemset if and only if there are $(m-1)$ j-itemsets whose supports exceed $\sup(X_{j,m})$. In addition, an itemset, denoted by $X^c_{j,m}$, is the mth most closed frequent j-itemset if and only if there are $(m-1)$ closed j-itemsets whose supports exceed $\sup(X^c_{j,m})$.

Example 2.1 As the example shown in Table 1, we illustrate top ten frequent itemsets and top ten closed itemsets in Table 2 to best understand the notation used. As can be seen, the minimum support to retrieve the top ten frequent itemsets is equal to five, i.e., $\sup_{\min}(T_{10}) = 5$, because there are ten itemsets with support larger than or equal to 5. Moreover, the minimum support to retrieve the top ten closed itemsets is equal to four [$\sup_{\min}(TC_{10}) = 4$], where we will retrieve 11 closed itemsets to be independent of the order of items. In addition, itemset $\{A\}$ is not a closed itemset since one of its superset, $\{AF\}$, has the same support. In this example, $\{D\}$ is the fourth most frequent 1-item which is denoted as $X_{1,4}$, because three 1-itemsets $\{A\}$, $\{B\}$ and $\{F\}$ have supports larger than $\sup(X_{1,4})$. In other words, $\sup(X_{1,4}) = 6$ will be the minimum support to retrieve top four 1-items. In addition, $X_{2,1}$ will correspond to either itemsets $\{AF\}$ or $\{BF\}$ because they have the same support and there is no 2-itemset whose support exceeds theirs. Furthermore, the first most closed frequent 3-itemsets, denoted by $X^c_{3,1}$, is $\{ABF\}$, and $\sup(X^c_{3,1}) = 7$. □

Note that closed itemsets have been deemed as the condensed representation of frequent itemsets, because a closed itemset is the itemset that covers all of its sub-itemsets with the same support [22]. In some applications, mining *top-k closed* itemsets will be equally or more applicable than

Table 1 An example transaction database

TID	Items
100	*A B D F*
200	*A B F*
300	*A D F*
400	*B C E D*
500	*B C D E F*
600	*A B F*
700	*A B F*
800	*A B D F*
900	*A B C D F*
1,000	*A B C E F*

Table 2 The illustrative example of *top-k* frequent/closed itemsets

Itemset	Sup.	Itemset	Sup.
Top-ten frequent itemsets $\sup_{\min}(T_{10}) = 5$			
A	8	*A F*	8
B	9	*B D*	5
D	6	*B F*	8
F	9	*D F*	5
A B	7	*A B F*	7
Top-ten closed itemsets $\sup_{\min}(TC_{10}) = 4$			
B	9	*B F*	8
D	6	*D F*	5
F	9	*A B F*	7
A F	8	*A D F*	4
B C	4	*B D F*	4
B D	5		
Examples of the mth most frequent/closed itemsets			
$X_{1,4} = \{D\}$			
$\sup(X_{1,4}) = 6$			
$X_{2,1} = \{AF\}, \{BF\}$			
$\sup(X_{2,1}) = 8$			
$X^c_{3,1} = \{ABF\}$			
$\sup(X^c_{3,1}) = 7$			

[2] Without loss of generality, the support is considered as the *absolute* occurrence frequency in this paper.

[3] Note that there may be larger than k itemsets satisfying this definition since itemsets may have the same support. Definition 1 will avoid the situation that the mining result depends on the order of items.

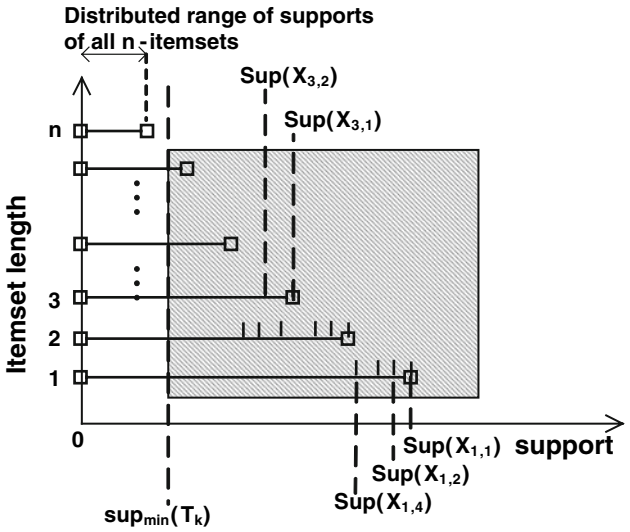


Fig. 1 The illustrative support distribution plot

mining *top-k* itemsets. We consider in this paper the approach which is equally applicable to mine *top-k* frequent itemsets and *top-k closed* itemsets, depending on the application need. In the following, we first describe the support distribution plot, which will be frequently exploited to illustrate our model of retrieving *top-k* frequent itemsets hereafter.

The support distribution plot: The support distribution plot consists of various parallel lines, where the ith line presents the range of supports of all i-itemsets, and each i-itemset can be plotted in the ith line with respect to its support. An illustration of the support distribution plot is shown in Fig. 1, where we can identify the position of itemset $X_{i,m}$, $\forall m$ in the ith line according to its support $\sup(X_{i,m})$. As can be seen, itemsets whose supports lie in the shadow region will comprise *top-k* frequent itemsets. Furthermore, according to the downward closure property [2], the line with respect to i-itemsets will be shorter than the line with respect to j-itemsets, where $i > j$. □

We then describe the concept to retrieve frequent patterns in the presence of the memory constraint. For ease of presentation, we discuss the issue of the memory constraint on the case of frequent itemsets. The discussion of *closed* itemsets is similar and thus we defer the details to Sect. 4. Note that the upper memory consumption of depth-first algorithms such as the *FPGrowth* algorithm [12] is proportional to the database size, which inherently cannot be limited below a user-specified memory size. We resort to level-wise search algorithms to realize the *memory-constraint frequent-pattern mining*. Specifically, it is clear that the memory consumption of level-wise search algorithms is solely proportional to the number of itemsets residing in the memory [24] , including the candidate itemsets and the stored itemsets.[4] Moreover, following Definition 1, mining *top-k* frequent itemsets can be viewed as mining frequent itemsets with the minimum support equal to $\sup_{\min}(T_k)$ if we assume that $\sup_{\min}(T_k)$ can be known in advance. Although *it is infeasible to make such an assumption, it can help to clarify important concepts of the considered model*. As such, Remark 1 below tells that we can limit the memory consumption in level-wise search algorithms by constraining the size of candidates tested in each database scan.

Remark 1 Suppose that the available memory size is specified as M. M can be equivalently transformed to the upper number of itemsets concurrently residing in the memory. Let the corresponding upper number of itemsets in memory be denoted by M_c. As such, the memory consumption will be limited below M if at most M_c candidates will be concurrently generated-and-tested in each database scan.[5]

In essence, the memory size occupied by each i-itemset is proportional to the corresponding itemset-length i. For simplicity, here we assume that all candidate itemsets occupy the same memory without considering its itemset-length. The discussion of this implementation issue will be deferred to Sect. 4. Clearly, Remark 1 states that a level-wise search algorithm is able to limit its memory consumption if we can guarantee the upper number of candidate itemsets being tested in one database scan. For example, suppose that $M_c = 300,000$, meaning that at most 300,000 candidate itemsets can be generated-and-tested in one database scan. Assuming we have 1,000 frequent 1-items, we can only select 775 1-items to generate candidate 2-itemsets since $\binom{775}{2} = 299,925$, which is bounded below M_c. These 299, 925 candidate 2-itemsets will be tested in one database scan, and the remaining $\binom{1000}{2} - \binom{775}{2} = 199,575$ candidate 2-itemsets will be generated-and-tested in the next database scan.

The concept of this approach deviates far from that of previous level-wise search algorithms, where all candidate $(i+1)$-itemsets are generated-and-tested in one database scan after all frequent i-itemsets have been found. Actually, readers may easily point out a straightforward solution to constrain the number of generated candidates: (1) arbitrarily combining frequent i-itemsets to generate their corresponding candidate $(i+1)$-itemsets until the number of candidates reaches the upper number of candidates M_c; (2) test these M_c candidates in one database scan and identify the contained frequent itemsets; (3) return to Step 1 unless no candidate $(i+1)$-itemset can be generated; (4) increase i by 1 and return to Step 1.

[4] We assume that discovered frequent itemsets will be stored in the menory for further use.

[5] It is reasonable to assume that M_c is much larger than the desired number of frequent itemsets k. Therefore, without loss of generality, we simply assume M_c only indicates the upper number of candidate itemsets which will be concurrently generated-and-tested in each database scan.

However, it is clear to see that the procedure of the candidate generation will become difficult since those early generated candidates must be systematically recorded to avoid the duplicate generation. Moreover, this approach may incur extra database scans since the available memory may not be fully utilized in some database scans (for example, the latest scan to test candidate i-itemsets may only occupy a small memory). It conflicts the spirit of previous works to reduce the number of database scans. To realize the *memory-constraint frequent-pattern mining* without comprising the execution efficiency, the number of database scans is required to be as small as possible. We therefore devise a baseline algorithm, called the *Naive* algorithm, to be an efficient *memory-constraint frequent-pattern mining* approach.

2.2 Algorithm Naive: the baseline method to discover frequent patterns in the presence of the memory constraint

In order to efficiently generate candidates in the presence of the memory constraint, we resort to the recent advanced technique presented in [8]. Specifically, the technique in [8] can estimate a tight upper bound of candidate itemsets. In our model, this technique can be further utilized to select the appropriate set of frequent i-itemsets in such a way that we can guarantee that their candidates $(i+1)$-itemsets can be fully generated in the available memory. Formally, given a set of j -itemsets F_j, the upper bound of candidate $(j+i)$-itemsets, generated from F_j can be estimated according to Theorem 1 below:

Theorem 1 *Given N and j, there exists a unique representation, called the j-canonical representation, as the form*

$$N = \binom{m_j}{j} + \binom{m_{j-1}}{j-1} + \cdots + \binom{m_r}{r},$$

where $r \geq 1, m_j \geq m_{j-1} \geq \cdots \geq m_r$, *and* $m_v \geq v$, *for* $v = r, r+1, \ldots, j$. *Therefore, assuming we have N j-itemsets, the tight upper bound of candidate* $(j+i)$*-itemsets generated from these N j-itemsets will be equal to*

$$\widehat{C}_{j,i}(N) = \binom{m_j}{j+i} + \binom{m_{j-1}}{j-1+i} + \cdots + \binom{m_{s+1}}{s+i+1},$$

where $i \geq 1$ *and s is the smallest integer such that* $m_s < s+i$. *If no such an integer exists, s will be equal to* $r-1$ *[8].*

Theorem 1 gives the tight upper bound of candidate $(j+i)$-itemsets which will be generated from a set of N j-itemsets. An illustrative example, which is quoted from [8], is shown below to clarify the concept of Theorem 1.

Example 2.2 Suppose that there are 13 3-itemsets in L_3, which are

{{3,2,1}, {4,2,1}, {4,3,1}, {4,3,2}, {5,2,1}, {5,3,2}, {5,4,1}, {5,4,2}, {5,4,2}, {5,4,3}, {5,3,1}, {6,2,1}, {6,3,2}}.

The *3-canonical representation* of 13 is $\binom{5}{3} + \binom{3}{2} = 13$, and hence the upper bound of candidate 4-itemsets is $\widehat{C}_{3,1}(13) = \binom{5}{4} + \binom{3}{3} = 6$. The upper bound of candidate 5-itemsets is $\widehat{C}_{3,2}(13) = \binom{5}{5} = 1$. This is tight indeed since candidates C_4 generated from L_3 will be $C_4 = \{\{4,3,2,1\}, \{5,3,2,1\}, \{5,4,2,1\}, \{5,4,3,1\}, \{5,4,3,2\}, \{6,3,2,1\}\}$, and C_5 is {5,4,3,2,1}. □

In light of Theorem 1, we devise a naive extension of level-wise search algorithms, called the *Naive* algorithm, to discover *top-k* frequent itemsets with the memory constraint:

Naive algorithm: We illustrate the idea of the *Naive* algorithm in Fig. 2, where Fig. 2a shows the process of database scans in the perspective of the support distribution plot and Fig. 2b shows the perspective of candidates generated in the available memory. Assuming $\sup_{\min}(T_k)$ can be known in advance, we can initially obtain the set of 1-items whose supports exceed $\sup_{\min}(T_k)$ after the first database scan. Suppose that L_i denotes the set of i-itemsets whose supports exceed $\sup_{\min}(T_k)$, and $|L_i|$ denotes the number of itemsets in L_i. We then select the most n_1 frequent items of L_1, i.e., $\{X_{1,1}, X_{1,2}, \ldots, X_{1,n_1}\}$ to generate their candidate 2-itemsets in the second database scan, where

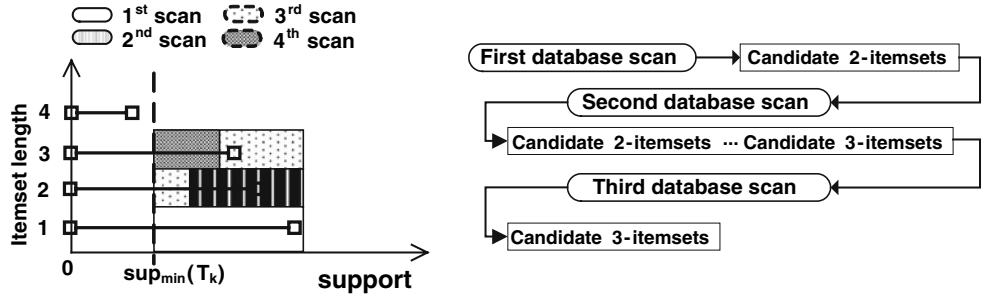
$$n_1 = \max\left\{n \,\middle|\, \widehat{C}_{1,1}(n) \leq M_c\right\}.$$

For example, $n_1 = 775$ if M_c is specified as 300,000 [$\because \widehat{C}_{1,1}(775) = 299,925$]. Therefore only candidate 2-itemsets from the most frequent 775 1-items will be generated in memory and tested in the second database scan. Formally, Lemma 1 tells that all 2-itemsets whose supports exceed $X_{1,775}$ will be retrieved:

Lemma 1 *Given the most n frequent i-itemsets* $\{X_{i,1}, X_{i,2}, \ldots, X_{i,n}\}$, *the set of (i+j)-itemsets whose supports exceed* $sup(X_{i,n})$ *will be included in the candidates generated from* $\{X_{i,1}, X_{i,2}, \ldots, X_{i,n}\}$, *for* $j \geq 1$.

Note that Lemma 1 is a direct result from the downward closure property. In this case, all 2-itemsets whose supports exceed $\sup(X_{1,775})$ will be retrieved after the second database scan. Afterward, if $\widehat{C}_{1,1}(|L_1|) - \widehat{C}_{1,1}(n_1) < M_c$, the remaining candidate 2-itemsets will be generated-and-tested in the third database scan. To better utilize the available memory, partial candidate 3-itemsets from the most frequent n_2 2-itemsets will be also generated and tested in the third database scan, where

$$n_2 = \max\left\{n \,\middle|\, \widehat{C}_{2,1}(n) \leq 2M_c - \widehat{C}_{1,1}(|L_1|)\right\}.$$



(a) The process of database scans in the perspective of the support distribution plot

(b) The process of database scans in the perspective of the candidate generation

Fig. 2 The illustration of mining frequent itemsets under the memory constraint

For example, suppose $|L_1| = 1,000$. We will generate and test $\widehat{C}_{1,1}(1000) - \widehat{C}_{1,1}(775) = 199,575$ candidate 2-itemsets, and at most $2 \times 300,000 - 499,500 = 100,500$ candidate 3-itemsets in the third database scan, where candidate 3-itemsets are generated from most frequent 3,629 2-itemsets since $n_2 = 3,629$ [$\because \widehat{C}_{2,1}(3629) = 100,481$ and $\widehat{C}_{2,1}(3630) = 100,540$]. Accordingly, we retrieve all 3-itemsets whose support exceed $\sup(X_{2,3629})$ after the third scan.

Explicitly, at most M_c candidate itemsets, possibly including candidate itemsets with various lengths, will be generated-and-tested in one database scan until no further candidates will be generated. Following the procedure of traditional level-wise search algorithms except the strategy of the candidate generation, we will retrieve *top-k* frequent itemsets finally. In case M_c is large enough, we may directly generate candidate i-itemsets from L_{i-2} or L_{i-3} as long as the candidate count is below M_c [Theorem 1 is able to determine the tight upper bound of candidate $(i + j)$-itemset generated from frequent i-itemsets, where $j > 1$]. It can be achieved by the technique similar to the scan-reduction technique discussed in [5]. □

Note that the feasibility of the naive algorithm relies on the process to avoid the generation of duplicate candidates. To realize this, recall a standard candidate generation procedure addressed in [16] at first:

$$C'_i = \left\{X \cup X' \middle| X, X' \in L_{i-1},\ \left|X \cap X'\right| = i-2\right\}.$$
$$C_i = \left\{X \in C'_i \middle| X \text{ contains } i \text{ members of } L_{i-1}\right\}.$$

In light of the property that the *Naive* algorithm always tests the set of candidates by joining higher-frequency itemsets, we can effectively avoid the candidate regeneration by rewriting the two-step candidate generation procedure, as shown in Lemma 2 below:

Lemma 2 *Suppose that following the procedure of the Naive algorithm, we have identified all frequent i-itemsets with supports exceeding $\sup(X_{i-1,n1})$ after the wth database scan. Let $F_{i-1,n1} = \{X_{i-1,1}, X_{i-1,2}, \ldots, X_{i-1,n1}\}$, and $F_{i-1,n2} = \{F_{i-1,n1}, F'\}$, where*

$$F' = \{X_{i-1,n1+1}, X_{i-1,n1+2}, \ldots, X_{i-1,n_2}\}$$

and $n_2 \geq n_1$. While we expect to identify all i-itemsets with support exceeding $\sup(X_{i-1,n2})$ in the next scan, the set of candidate i-itemsets C_i that will be generated by the Naive algorithm in the $w+1$th scan is:

$$C''_i = \left\{X \cup X' \middle| X \in F_{i-1,n2}, X' \in F',\ \left|X \cap X'\right| = i-2\right\}.$$
$$C_i = \left\{X \in C''_i \middle| X \text{ contains } i \text{ members of } F_{i-1,n2}\right\}.$$

As such, the Naive algorithm can effectively test necessary candidates without regenerating candidates which have been generated before.

Rationale: Note that the Naive algorithm has a property that it always generates-and-tests the set of candidates from higher-frequency itemsets. According to Lemma 1, the set of candidate i -itemsets generated from $F_{i-1,n1}$ will not be necessary to be examined again because they have been generated in previous scans. Following the first step of the candidate generation in [16], the superset of candidate i-itemsets generated from $F_{i-1,n2}$, is $\left\{X \cup X' \middle| X, X' \in F_{i-1,n2}, \left|X \cap X'\right| = i-2\right\}$, which is equivalent to

$$\left\{X \cup X' \middle| X \in F_{i-1,n2}, X' \in F', \left|X \cap X'\right| = i-2\right\} \cap \left\{X \cup X' \middle| X, X' \in F_{i-1,n1}, \left|X \cap X'\right| = i-2\right\}.$$

Since all validated candidates in

$$\left\{X \cup X' \middle| X, X' \in F_{i-1,n1},\ \left|X \cap X'\right| = i-2\right\}$$

have been generated before, only the set of itemsets in $\left\{X \cup X' \middle| X \in F_{i-1,n2}, X' \in F',\ \left|X \cap X'\right| = i-2\right\}$, i.e., C''_i, is necessary to be examined to generated validated candidates in the $w+1$th scan. Based on the foregoing, the candidate generation procedure in Lemma 2 can effectively

avoid regenerating candidates which have been tested before. □

Lemma 2 can be clearly illustrated by the following example. Suppose that we have tested all candidate 3-itemsets from $F_{i-1,n1} = \{\{1, 2\},\{1, 3\}, \{1, 4\},\{2, 3\},\{2, 4\},\{3, 5\}\}$ in previous scans, i.e., candidates 3-itemsets {1, 2, 3} and {1, 2, 4} have been tested before. While we expect to test all candidates from $F_{i-1,n2} = \{F_{i-1,n1},\{3, 4\},\{1, 5\}\}$, we have

$$C_3'' = \left\{ X \cup X' \,\middle|\, \begin{array}{c} X \in F_{i-1,n2},\, X' \in \{\{3,4\},\, \{1,5\}\}, \\ \left|X \cap X'\right| = i-2 \end{array} \right\} = \{\{1, 3, 4\}, \{2, 3, 4\}, \{3, 4, 5\}\},$$

and $C_3 = \{\{1, 3, 4\},\{2, 3, 4\}\}$. Finally, we generate the necessary candidates without regenerating these candidate 3-itemsets which have been tested before, i.e., {1, 2, 3} and {1, 2, 4}. Since $F_{i-1,n2}$ and F' can be identified without extra overhead, we can effectively avoid the duplicate candidate generation in the *Naive* algorithm.

For interest of space, we omit the formal presentation of this naive approach because this approach is merely devised for the comparison purpose. It is worth mentioning that, the *Naive* algorithm conveys the important concept that the problem of mining *top-k* frequent itemsets can be equivalently viewed as a jigsaw puzzle-like problem as follows:

Remark 2 Consider the view of the support distribution plot such as Fig. 2a. Imagine that following the procedure of the Naive algorithm, we can fill up a right-most region of the support distribution plot, which consists of identified frequent itemsets, after each database scan.[6] For example, the four database scans in Fig. 2a will correspond to four right-most regions in the shadow region. From this point, the problem of efficiently mining *top-k* frequent itemsets can be translated to the problem that "how can the area with the support exceeding $\sup_{\min}(T_k)$ in the support distribution plot be separated into disjoint regions while the number of regions is as small as possible?"

Clearly, Remark 2 gives an important perspective to analyze the problem of mining *top-k* frequent itemsets. This will be used for further devising the efficient solution to mine *top-k* frequent itemsets. Note that the *Naive* approach must be achieved under the assumption that $\sup_{\min}(T_k)$ is known prior to the mining process. However, it is difficult, even impossible to know $\sup_{\min}(T_k)$ in advance. We still need to devise an efficient mining solution to achieve the same goal without such an assumption. Before presenting the details of the feasible approach, we give Remark 3 below to demonstrate that the *Naive* approach is one of the most efficient level-wise search algorithms to retrieve *top-k* frequent itemsets under the memory constraint, if $\sup_{\min}(T_k)$ can be known in advance.

Remark 3 Let C_m denote the set of candidate m-itemsets generated from the set of (m-1)-itemsets whose supports exceed $\sup_{\min}(T_k)$. Without loss of generality, C_m will be the set of candidate m-itemsets which should be generated-and-tested to obtain the set of m-itemsets belonging to *top-k* frequent itemsets by applying level-wise search approaches. Therefore, while also utilizing several advanced pruning techniques of level-wise algorithms such as the hash-pruning technique [5], the Naive algorithm will retrieve the *top-k* frequent itemsets with the minimum number of database scans in the presence of the memory constraint.

In Sect. 5, the *Naive* algorithm will be used to evaluate the efficiency of solutions to mine *top-k* frequent/closed itemsets for comparison purposes.

3 Memory-constraint *top-k* frequent-pattern mining

3.1 Principles to search *top-k* frequent patterns

We present the idea to efficiently retrieve *top-k* frequent itemsets under the memory constraint without the assumption that $\sup_{\min}(T_k)$ is known in advance. At first, necessary properties of *top-k* frequent/closed itemsets are presented.

Lemma 3 *The* mth *most frequent* j*-itemset,* $X_{j,m}$*, is included in candidates generated from the set of* i*-itemsets whose supports are larger than or equal to sup*$(X_{j,m})$*, where* $j > i$.

In essence, Lemma 3 is a direct result from the downward closure property. According to Lemma 3, we also have Lemmas 4, 5 and 6.

Lemma 4 *The set of* $(j+i)$*-itemsets belonging to top-k frequent itemsets will be a subset of candidates generated from the set of* j*-itemsets belonging to top-k frequent itemsets, where* $i \geq 1$.

Rationale: Suppose that C_{j+i} denotes the set of candidate $(j+i)$-itemsets generated from the set of frequent $(j+i-1)$-itemsets, denoted by L_{j+i-1}, whose supports all exceed $\sup_{\min}(T_k)$ in this case. That is, $C_{j+i} = L_{j+i-1} * L_{j+i-1}$. Let C^1_{j+i} be the set of candidate $(j+i)$-itemsets directly generated from C_{j+i-1}, i.e., $C^1_{j+i} = C_{j+i-1} * C_{j+i-1}$. Since $C_{j+i-1} \supseteq L_{j+i-1}$, we have $C_{j+i-1} * C_{j+i-1} \supseteq L_{j+i-1} * L_{j+i-1}$, showing that $C^1_{j+i} \supseteq C_{j+i} \supseteq L_{j+i}$. Recursively we have $C^i_{j+i} \supseteq \cdots \supseteq C^2_{j+i} \supseteq C^1_{j+i} \supseteq C_{j+i}$, where

[6] In practice, some itemsets are generated-and-tested in previous database scans since itemsets with support larger than $\sup_{\min}(T_k)$ will be maintained after each database scan. For simplicity, we convey the concept without considering such a slight difference. It will be a matter of the implementation.

C^i_{j+i} is the candidate $(j+i)$-itemsets directly generated from frequent j-itemsets, thus indicating $C^i_{j+i} \supseteq L_{j+i}$ and leading to Lemma 4. Note that the power of this property has been fully utilized in the *scan-reduction* technique to reduce the number of database scans [2,20]. □

Lemma 5 *Suppose that we make sure that all i-itemsets whose supports exceed $sup(X_{i,v_i})$ have been discovered. It implies that we have already tested all candidate i-itemsets generated from $(i-m)$-itemsets whose supports exceed $sup(X_{i,v_i})$, where $m \geq 1$.*

Rationale: According to Lemma 1, all candidate i-itemsets generated from $(i-1)$-itemsets with support exceeding $\sup(X_{i,v_i})$ must be tested (or be pruned) if we want to obtain the set of frequent i-itemsets whose supports exceed $\sup(X_{i,v_i})$. In addition, the set of candidate i-itemsets generated from the set of $(i-1)$-itemsets whose supports exceed $\sup(X_{i,v_i})$ is a subset of candidate i-itemsets directly generated from frequent $(i-m)$-itemsets whose supports exceed $\sup(X_{i,v_i})$ for $m > 1$. Hence, it is clear that if we have obtained all i-itemsets whose supports exceed $\sup(X_{i,v_i})$, all candidate i-itemsets generated from $(i-1)$-itemsets whose supports exceed $\sup(X_{i,v_i})$ have also been tested, even though in practice candidate i-itemsets are directly generated from frequent $(i-m)$-itemsets for $m > 1$. Recursively, we can derive that all candidate i-itemsets generated from $(i-m)$-itemsets whose supports exceed $\sup(X_{i,v_i})$. □

Lemma 6 *Suppose that after the wth database scan, we have retrieved a set of itemsets $\mathbb{R}_w = \{X_{1,1}, X_{1,2}, \ldots, X_{1,m_1}, X_{2,1}, \ldots, X_{2,m_2}, \ldots, X_{t,m_t}, \ldots\}$, where $t \geq 1$ and $X_{i,1}, \ldots, X_{i,m_i}$ denotes the set of all i-itemsets whose supports exceed $sup(X_{i,m_i})$. Let $sup_k(w)$ be equal to the support of the kth most frequent itemset in $\mathbb{R}_w$ (if $|\mathbb{R}_w| < k$, $\sup_k(w) = 0$). Accordingly, we have $sup_k(w) \leq \sup_k(z)$, for $z > w$, and $sup_k(w) \leq sup_{\min}(T_k)$.*

Rationale: Lemma 6 can be proved by contradiction. Suppose that $\sup_k(w) > \sup_{\min}(T_k)$. Note that there are k itemsets whose supports exceed $\sup_{\min}(T_k)$. Thus we will have less than k itemsets whose supports exceed $\sup_k(w)$, which conflicts the definition of $\sup_k(w)$. As such, $\sup_k(w)$ will be smaller than or equal to $\sup_{\min}(T_k)$. In addition, since $\mathbb{R}_w \subseteq \mathbb{R}_z$ for $z > w$, it is clear that $\sup_k(w) \leq \sup_k(z)$. □

In light of Lemma 6, one may obtain *top-k* frequent itemsets by initially setting the minimum support equal to zero, and then raising the minimum support equal to $\sup_k(w)$ after the wth database scan. Therefore, without the assumption that $\sup_{\min}(T_k)$ is known in advance, the problem to efficiently mine *top-k* frequent itemsets can be viewed as the problem to find $\sup_k(w) = \sup_{\min}(T_k)$, where the number of database scans, w, is as small as possible.

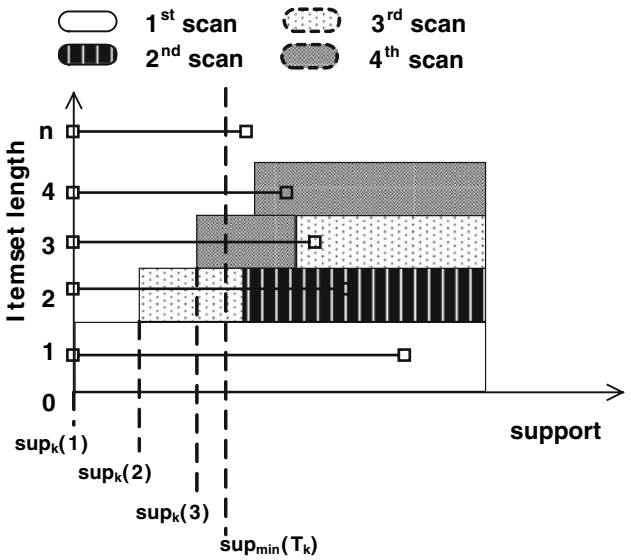


Fig. 3 The illustration of the horizontal first search approach

As a consequence, analogous to the process in the *Naive* algorithm, we can initially discover the set of high-support itemsets in each level, and then progressively discover itemsets with the relatively small support by generating and testing candidates except those candidates tested in previous scans. In other words, we can prioritize to fill up the region of the right part of the support distribution plot (recall that Remark 2 states that mining *top-k* frequent itemsets can be viewed as a jigsaw puzzle-like problem). As such, two alternative approaches below is devised to fill up the right part of the support distribution plot, i.e., to retrieve *top-k* frequent itemsets, without assuming that $\sup_{\min}(T_k)$ is known in advance.

Horizontal first search approach: The first approach is called the *horizontal first search approach*, whose perspective of the support distribution plot is shown in Fig. 3. The basic concept is that, in the wth database scan, we prioritize to discover itemsets in the sibling level whose supports exceed $\sup_k(w-1)$. Specifically, suppose that we have tested candidate $(i+1)$-itemsets generated from i-itemsets whose supports exceed s_1 after the wth database scan. In the $(w+1)$th database scan, we will generate all candidate $(i+1)$-itemsets from i-itemsets whose supports exceed s_2, excluding those candidates generated in previous scans, where $\sup_k(w-1) \leq s_2 < s_1$. Surely, the memory space must be guaranteed to ensure the generated candidates can be maintained in the memory. While we still have the remaining memory, we can concurrently generate partial candidate $(i+2)$-itemsets from those identified most frequent $(i+1)$-itemsets.

Note that the *horizontal first search approach* can fully utilize the merit of level-wise algorithms to effectively prune unnecessary candidates. For example, a candidate 3-itemset $\{A, B, C\}$ will not be generated and tested in the wth database

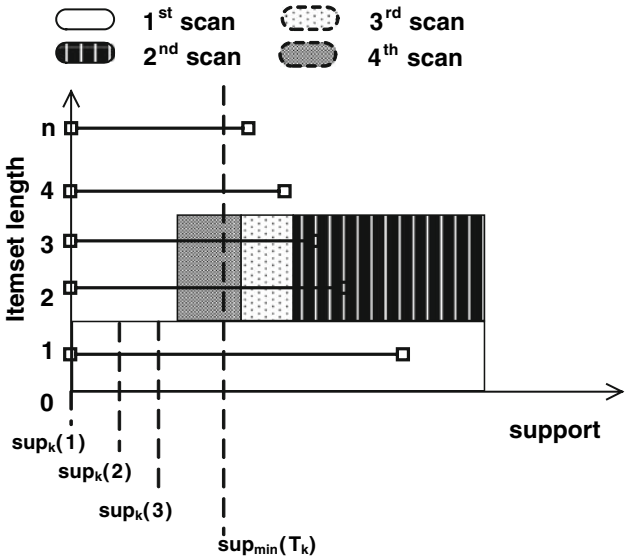


Fig. 4 The illustration of the vertical first search approach

scan if the support of one of $\{A, B\}$, $\{A, C\}$, $\{B, C\}$ is smaller than $\sup_k(w-1)$. However, one drawback of this approach is that we may identify many itemsets which are not included in *top-k* frequent itemsets. The reason results from that a lot of *top-k* itemsets are with long itemset-lengths, and thus $\sup_k(w)$ cannot effectively and quickly approach $\sup_{\min}(T_k)$ when w is small. As shown in Fig. 3, it is clear to see that a lot of itemsets with support smaller than $\sup_{\min}(T_k)$ will be also discovered in the wth database scan when $\sup_k(w-1)$ is not close to $\sup_{\min}(T_k)$. Generating those unnecessary itemsets will lead to extra database scans, thus degrading the execution efficiency. □

Vertical first search approach: The second approach is called the *vertical first search approach*, whose perspective of the support distribution plot is shown in Fig. 4. The basic concept is that, in the wth database scan, we prioritize to discover longer itemsets whose supports exceed $\sup_k(w-1)$. This can be achieved by directly generate candidate i-itemsets from frequent j-itemsets, where j may be smaller than $i-1$. For example, let $n_1 = \max\left\{n \left| \left(\sum_{i=1}^{2} \widehat{C}_{1,i}(n)\right) \le M_c \right.\right\}$. We may select n_1 most frequent 1-items, i.e., $X_{1,1}, X_{1,2}, \ldots, X_{1,n_1}$, and then test whether there are i -itemsets, $2 \le i \le 3$, whose supports exceed $\sup(X_{1,n_1})$. In practice, the *vertical first search approach* is beneficial to efficiently obtain *top-k* frequent itemsets when k is small [or, $\sup_{\min}(T_k)$ is high]. Moreover, another merit of the *vertical first search approach* is that the identified itemsets will mostly belong to *top-k* frequent itemsets as compared to the case in the *horizontal first search approach*. However, the drawback of this approach is that a lot of candidates, which indeed can be pruned by the level-wise search, will also be generated-and-tested. Thus when k is large, this approach will suffer from an extremely large number of database scans. □

3.2 The δ-stair search

Apparently, it is still required to devise a solution to more effectively make $\sup_k(w)$ approach $\sup_{\min}(T_k)$ while also fully utilizing the merit of the level-wise candidate pruning. In other words, we try to integrate the merit of the *horizontal first search approach* and the *vertical first search approach* while diminishing the side-effect of those two approaches. To achieve this, we propose a novel search approach, called the *δ-stair search*, in this paper. The basic concept behind the *δ-stair search* is to equally share M_c candidates to δ different itemset lengths and then to gradually upward or downward search frequent itemsets. Specifically, the *δ-stair search* consists of two distinct steps, namely the *upward δ-stair search step* and the *downward δ-stair search step*. We formally describe the principles of *upward δ-stair search step* and the *downward δ-stair search step*, respectively.

Upward δ-stair search step: Suppose that in the wth database scan, we have concurrently generated-and-tested candidate itemsets whose lengths are between u and $u+\delta-1$. In the $(w+1)$th database scan, the upward δ-stair search will concurrently generate-and-test candidate itemsets whose lengths are between $u+1$ and $u+\delta$. Furthermore, assume before the $(w+1)$th database scan, we have discovered the set of itemsets $\mathbb{R}_w = \{X_{1,1}, X_{1,2}, \ldots, X_{1,v_1}, X_{2,1}, \ldots, X_{2,v_2}, \ldots, X_{t,v_t}, \ldots\}$ in which each itemset has the support exceeding $\sup_k(w)$, where $\{X_{i,1}, \ldots, X_{i,v_i}\}$ denotes the set of i-itemsets whose supports exceed $\sup(X_{i,v_i})$. In the $(w+1)$th database scan, we examine candidate $(j+1)$-itemsets generated from $\{X_{j,1}, X_{j,2}, \ldots, X_{j,n_j}\}$, excluding candidate $(j+1)$-itemsets from $\{X_{j,1}, X_{j,2}, \ldots, X_{j,m_j}\}$, where $u \le j \le u+\delta-1$, and

$$n_j = \max\left\{ n \left| \begin{array}{l} \widehat{C}_{j,1}(n) - \gamma_{j+1} \le \frac{M_c}{\delta}, \\ \sup(X_{j,n}) \ge \sup_k(w) \end{array} \right. \right\},$$

$$m_j = \min\left\{ n \left| \sup(X_{j,n}) \ge \sup(X_{j+1,v_{j+1}}) \right. \right\}.$$

Here γ_i denotes the number of candidate i-itemsets which have been tested before the $(w+1)$th database scan.

As such, we will retrieve all (j+1)-itemsets whose supports exceed $\sup(X_{j,n_j})$ after the $(w+1)$th database scan, where $u \le j \le u+\delta-1$.

Example 2.3 Consider the illustration shown in Fig. 5a. In this case, δ is set to 2, meaning that candidate itemsets of two levels will be concurrently generated-and-tested in each database scan. Specifically, we will obtain the set of 1-items after the first database scan (if the count of distinct 1-items exceeds k, we only select most k frequent 1-items). Afterward, we generate $\frac{M_c}{2}$ candidate 2-itemsets from the set of most h_1 frequent 1-items and generate $\frac{M_c}{2}$ candidate 3-itemsets directly from the set of most frequent h_1' 1-items, where $\widehat{C}_{1,1}(h_1) \le \frac{M_c}{2}$ and $\widehat{C}_{1,2}(h_1') \le \frac{M_c}{2}$. Therefore we will obtain 2-itemsets whose supports exceed $\sup(X_{1,h_1})$ and

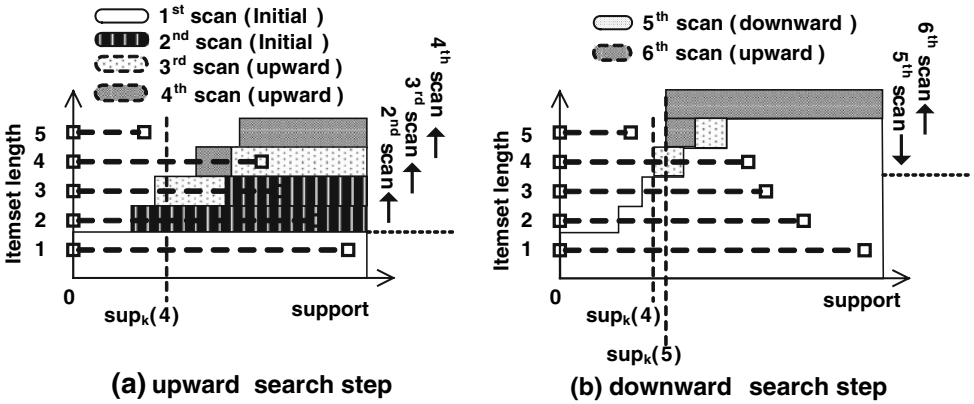


Fig. 5 The illustration of the *δ-stair search* with $\delta = 2$

3-itemsets whose supports exceed $\sup(X_{1,h'_1})$ after the second database scan. For example, suppose that $M_c = 300{,}000$. We select the most frequent 548 1-items to generate candidate 2-itemsets [$h_1 = 548$, because $\widehat{C}_{1,1}(548) = 149{,}878$] and select the most frequent 97 1-items to directly generate candidate 3-itemsets [$h'_1 = 97$, because $\widehat{C}_{1,2}(97) = 147{,}440$]. The first and the second database scans are referred to as *the initial step* in this paper.

Note that after the second database scan, we will retrieve all 2-itemsets, denoted by $\{X_{2,1}, \ldots, X_{2,v_2}\}$, whose supports exceed $\sup(X_{1,h_1})$, and retrieve the set of 3-itemsets, denoted by $\{X_{3,1}, \ldots, X_{3,v_3}\}$, whose support exceed $\sup(X_{1,h'_1})$. In addition, let X_{2,m_2} be the 2-itemset that all 3-itemsets with support exceeding $\sup(X_{2,m_2})$ have been discovered in the second database scan, i.e., $m_2 = \max\left\{n \middle| \sup(X_{2,n}) \le \sup(X_{3,v_3})\right\}$. In the third database scan, we will upwardly search *top-k* frequent itemsets. Note that according to Lemma 4, we have already tested all candidate 3-itemsets generated from $\{X_{2,1}, \ldots, X_{2,m_2}\}$ in the second database scan. As such, in the third database scan, $\frac{M_c}{2}$ candidate 3-itemsets are generated from the set $\{X_{2,1}, X_{2,2}, \ldots, X_{2,n_2}\}$, excluding candidate 3-itemsets from $\{X_{2,1}, X_{2,2}, \ldots, X_{2,m_2}\}$, where $n_2 = \max\left\{n \middle| \widehat{C}_{2,1}(n) \le \frac{M_c}{2}\right\}$. Moreover, since we did not test any 4-itemset in previous database scans, in the third database scan, we will also generate and test $\frac{M_c}{2}$ candidate 4-itemsets, which are concurrently generated from the set $\{X_{3,1}, X_{3,2}, \ldots, X_{3,n_3}\}$, where $n_3 = \max\left\{n \middle| \widehat{C}_{3,1}(n_3) \le \frac{M_c}{2}\right\}$. Finally, the third database scan is executed to test these generated candidates.

The execution of the third database scan is called an *upward δ-stair search step*. Same as the execution of the third database scan, the execution of the fourth database scan is also an *upward δ-stair search step*, as illustrated in Fig. 5a. □

Downward δ-stair search step: Suppose that in the wth database scan, we have concurrently generated-and-tested candidate itemsets whose lengths are between u and $u+\delta-1$. The downward search will generate candidates corresponding to two different cases below:

(a) If the wth database scan is an upward δ-stair search, in the $(w+1)$th database scan, the downward δ-stair search will concurrently generate-and-test candidate itemsets whose lengths are between u and $u+\delta-1$.
(b) If the wth database scan is a downward δ-stair search, in the $(w+1)$th database scan, the downward δ-stair search will concurrently generate-and-test candidate itemsets whose lengths are between $u-1$ and $u+\delta-2$.

Furthermore, assume before the $(w+1)$th database scan, we have discovered the set of itemsets $\mathbb{R}_w = \{X_{1,1}, X_{1,2}, \ldots, X_{1,v_1}, X_{2,1}, \ldots, X_{2,v_2}, \ldots, X_{t,v_t}, \ldots\}$ in which each itemset has the support exceeding $\sup_k(w)$, where $\{X_{i,1}, \ldots, X_{i,v_i}$ denotes the set of i-itemsets whose supports exceed $\sup(X_{i,v_i})$. In the $(w+1)$th database scan, the downward δ-stair search step will generate candidate $(j+1)$-itemsets from $\{X_{j,1}, X_{j,2}, \ldots, X_{j,n_j}\}$, excluding candidate $(j+1)$-itemsets from $\{X_{j,1}, X_{j,2}, \ldots, X_{j,m_j}\}$, where j lies in the range corresponding to case (a) or case (b), and

$$n_j = \max\left\{n \middle| \begin{array}{l} \widehat{C}_{j,1}(n) - \gamma_{j+1} \le \frac{M_c}{\delta}, \\ \sup(X_{j,n}) \ge \sup_k(w) \end{array}\right\},$$

$$m_j = \min\left\{n \middle| \sup(X_{j,n}) \ge \sup(X_{j+1,v_{j+1}})\right\}.$$

Here γ_{j+1} denotes candidate $(j+1)$-itemsets which have been tested before $(w+1)$th database scan.

As such, after the $(w+1)$th database scan, we will retrieve all $(j+1)$-itemsets whose supports exceed $\sup(X_{j,n_j})$ with the itemset-lengths corresponding to case (a) or case (b) above.

Example 2.4 Consider the illustration shown in Fig. 5b. In this example, candidate 5-itemsets generated from $\{X_{4,1}, X_{4,2}, \ldots, X_{4,m_4}\}$ has been tested after the fourth database scan and no 5-itemsets whose supports exceed $\sup(X_{4,m_4})$ were found. As such, the *upward search* will fail to find any 6-itemsets belonging to *top-k* frequent itemsets, thus the fifth

database scan will be turned to the *downward δ-stair search step*.

In the fifth database scan, $\frac{M_c}{2}$ candidate 5-itemsets, excluding candidates generated from $\{X_{4,1}, X_{4,2}, \ldots, X_{4,m_4}\}$, are generated from the set $\{X_{4,1}, X_{4,2}, \ldots, X_{4,n_4}\}$, where

$$n_4 = \max\left\{n \,\middle|\, \widehat{C}_{4,1}(n) - \gamma_5 \leq \frac{M_c}{2}\right\}$$

and γ_5 denotes the number of candidate 5-itemsets which have been tested in previous database scans. Moreover, assume that candidate 4-itemsets generated from $\{X_{3,1}, X_{3,2}, \ldots, X_{3,m_3}\}$ has been tested in previous database scans. We also generate $\frac{M_c}{2}$ candidate 4-itemsets from $\{X_{3,1}, X_{3,2}, \ldots, X_{3,n_3}\}$, excluding candidate 4-itemsets generated from $\{X_{3,1}, X_{3,2}, \ldots, X_{3,m_3}\}$, where $n_4 = \max\left\{n \,\middle|\, \widehat{C}_{3,1}(n) - \gamma_4 \leq \frac{M_c}{2}\right\}$, and γ_4 denotes the number of candidate 4-itemsets which have been tested in previous scans.

Suppose that after the fifth database scan, all 1-items, 2-itemsets, 3-itemsets and 4-itemsets whose supports exceed $\sup_k(5)$ are found. In other words, we did not need to "downward" search itemsets which will not belong to *top-k* frequent itemsets. As such, the sixth database scan will return to be an *upward δ-stair search step*. Finally, the process of mining *top-k* frequent itemsets will end after the sixth database scan since no 5-itemsets and 6-itemsets whose supports exceed $\sup_k(6)$ were found. Accordingly, *top-k* frequent itemsets, i.e., those itemsets whose supports exceed $\sup_k(6)$, are discovered. In this case, $\sup_{\min}(T_k)$ is equal to $\sup_k(6)$. □

We then formally describe when to shift from the *upward δ-stair search step* to the *downward δ-stair search step,* and vice versa.

The upward search step → the downward search step: Suppose that in the wth database scan, which is an *upward δ-stair search step,* we have generated-and-tested candidate itemsets whose lengths are between $i-\delta+1$ and i. Moreover, assume that after the wth *database scan*, candidate i-itemsets generated from $\{X_{i-1,1}, X_{i-1,2}, \ldots, X_{i-1,v_{i-1}}\}$ have been generated and tested, and no i-itemsets with support larger than $\sup(X_{i-1,v_{i-1}})$ were found. In such cases, the $(w+1)$th database scan will turn to be a *downward δ -stair search step* since there will be no $(i+1)$-itemsets whose supports exceed $\sup(X_{i-1,v_{i-1}})$ (see the example of the fourth scan to the fifth scan in Fig. 5). □

The downward search step → the upward search step: Suppose that in the wth *database scan*, which is a *downward δ-stair search step,* we have generated-and-tested itemsets whose lengths are between i and $i+\delta-1$. Moreover, assume that after the wth *database scan*, all candidate i-itemsets generated from $\{X_{i-1,1}, X_{i-1,2}, \ldots, X_{i-1,v_{i-1}}\}$ have been tested, where $v_{i-1} = \max\left\{n \,\middle|\, \sup(X_{i-1,n}) \leq \sup_k(w)\right\}$. In such cases, the $(w+1)$th database scan will turn to be an *upward δ-stair search step*, because no $(i-1)$-itemsets with support below $\sup_k(w)$ will belong to *top-k* frequent itemsets (see the example of the fifth scan to the sixth scan in Fig. 5b). □

Accordingly, the *upward δ-stair search step* and the *downward δ-stair search step* will be adaptively switched until all *top-k* frequent itemsets are found. It is worth mentioning that, the *δ-stair search* approach will efficiently retrieve *top-k* frequent itemsets with the number of database scans close to the optimal one, which is required by the *Naive* algorithm described in Sect. 2 [under the premise that $\sup_{\min}(T_k)$ is known in advance]. This is attributed to that the *δ-stair search* has both advantages of the *horizontal first search approach* and the *vertical first search approach* while diminishing their side-effects.

4 Algorithm *MTK*

To efficiently retrieve *top-k* frequent patterns, we in this section introduce the algorithm, called *MTK* (standing for *M* emory-constraint *top-k* mining), to realize the concept of the *δ-stair search*. In Sect. 4.1, we give the implementation details of *MTK*. The extension of *MTK* to mine *top-k closed* itemsets will be discussed in Sect. 4.2. Section 4.3 gives illustrative examples.

4.1 Implementation of *MTK*

Before presenting the details of the implementation, we give the overview of *MTK* in Fig. 6. In essence, *MTK* applies the hash pruning technique from algorithm *DHP* [19], which can effectively reduce unnecessary candidates by utilizing a *hash table* structure. Specifically, when scanning the database to obtain all 1-items (or most k frequent 1-items), we also examine all 2-itemsets of each transaction, and hash them into the different buckets of the hash table H_2, i.e., increasing the corresponding bucket count. The hash table H_2 can be further used to reduce the amount of candidate 2-itemsets which will be examined in the following database scans if necessary. It is worth mentioning that, as demonstrated in [19], the hash pruning technique will be powerful in early stages, particularly when pruning candidate 2-itemsets. Thus both considering the pruning effect and the other incurred overhead, the hash pruning will only be utilized in generating candidate 2-itemsets.

In the second database scan, $\frac{M_c}{\delta}$ candidate i-itemsets, $2 \leq i \leq 2+\delta-1$, will be directly generated from 1-items. For the case of candidate 2-itemsets, we need to generate candidate 2-itemsets, which belong to the bucket whose bucket count exceeds $\sup_k(1)$ in H_2 (interested readers can refer to [19] for the details). After the second database scan, we filter out

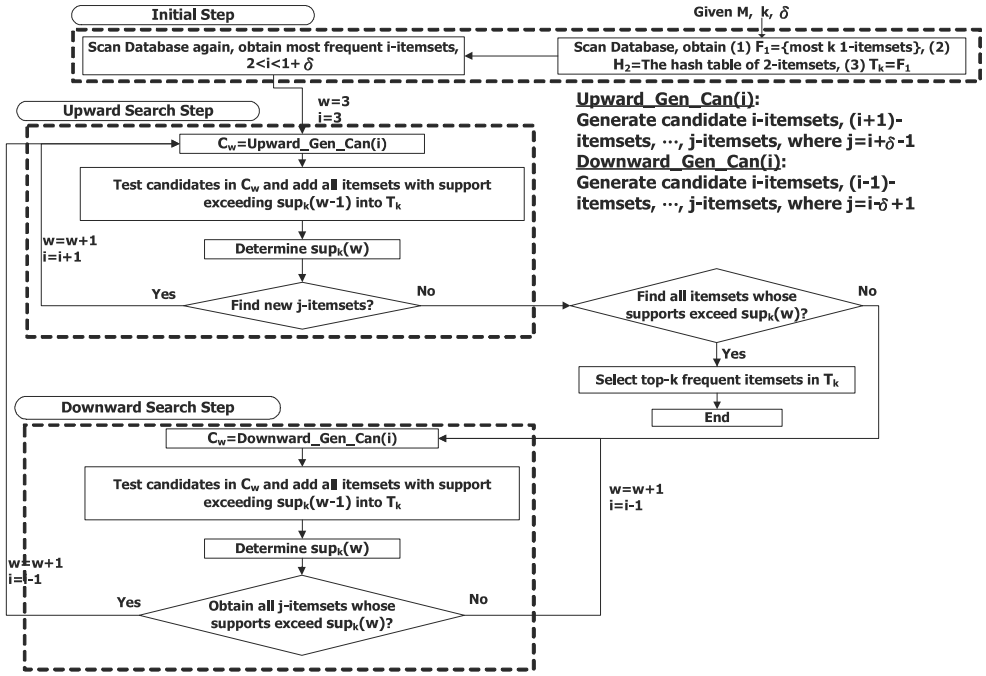


Fig. 6 The flowchart of *MTK*

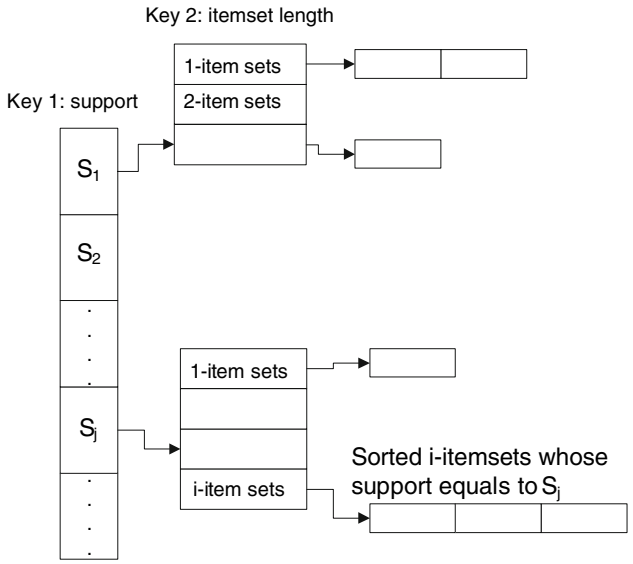


Fig. 7 The two-level sorted array to maintain *top-k* frequent itemsets

itemsets which were discovered in the first and the second database scans and their supports are smaller than $\sup_k(2)$. Afterward, the third database scan will be executed as either an *upward δ-stair search step* or a *downward δ-stair search step*. Then, the *upward δ-stair search step* and the *downward δ-stair search step* will be adaptively switched according to the criterion described in Sect. 3.2, until all *top-k* frequent itemsets are found. In case we need to examine candidate 2-itemsets, H_2 will be utilized again to prune the size of candidates.

The detailed pseudo-codes are outlined in the below. Explicitly, in addition to the hash table H_2 to pruning candidate 2-itemsets, several important global variables will be also maintained, including (1) T_k: the repository to maintain *top-k* frequent itemsets; (2) $size_i$: the pre-estimated memory overhead to store a candidate i-itemsets; (3) $TB[x][y][z]$: a pre-computed sorted array, in which each array unit consists of two variables, namely $TB[x][y][z].bound\_cand$ and $TB[x][y][z]$.itemset_num, to indicate the upper number of candidate $(x+y)$-itemsets generated from x-itemsets, where

$$TB[x][y][z].bound\_cand = \widehat{C}_{x,y}(TB[x][y][z].itemset\_num).$$

Specifically, as shown in Fig. 7, T_k is a dynamic array structure to maintain *top-k* frequent itemsets, sorted by two keys: (1) the support of the itemset; (2) the itemset length. Once we identify a new itemsets whose support exceeds the up-to-date minimum support threshold s, it will be inserted into T_k and itemsets with the smallest support in T_k will be removed if $|T_k| > k$. In addition, the memory to store a candidate is proportional to the itemset length. As such, $size_i$ can be approximated as the memory of necessary units multiplied by a factor Φ, where Φ, depending on the implementation, represents the other overhead such as memory to maintain necessary pointers in a hash-tree [2]. Moreover, to search $n_{\text{opt}} = \max\left\{n \,\middle|\, \widehat{C}_{x,y}(n) \leq m\right\}$, for a given upper number of candidates m, the function to obtain $\widehat{C}_{x,y}(n)$ originally needs to be iteratively executed until n_{opt} is found. We can reduce the execution time to search n_{opt} by

Algorithm: MTK(k, M, δ)

```
/*global variables*/
T_k = ∅;  /*the repository to maintain top-k frequent itemsets, which is sorted by (1) the support of the itemset; (2) the length of the itemset*/
H_2 = ∅;  /*refer to [19] for the details of the hash-pruning technique*/
size_i  /*the pre-specified memory overhead to store a candidate itemset*/
s=0;  /*minimum support*/
tested=∅;  /*a hash table to indicate how many candidate (i+1)-itemsets have been tested in previous scans*/
TB[x][y][z];  /*a pre-computed sorted array to indicate the upper number of candidate (x+y)-itemsets according to Theorem 1*/
len_start=1; len_end=1;  /*the range of itemset-lengths of candidates examined in this database scan*/
examined=∅;  /*a hash table, where examine(i) returns the number of top i-itemsets that all (i+1)-itemsets generated from them have been examined in previous database scans*/
/*Main Program*/
 1. InitStep();  /*The initial step of MTK, including two database scans*/
 2. s = min{sup(x) | x ∈ T_k };  /*The new minimum support threshold*/
 3. IsUpward=true;  /*indicate whether the previous step is an upward step search*/
 4. while (true) {
 5.   if ( IsUpward==true) { /*the previous step is an upward search*/
 6.     if ({c | c ∈ T_k, |c|= len_end } ≠ ∅)  /*remain as the upward search*/
 7.       len_start=len_start+1; len_end=len_end+1;
 8.     else     /*switch to the downward search*/
 9.       len_start=len_start; len_end=len_end; IsUpward=false;
10.   }
11.   else {     /*the previous step is an downward search*/
12.     Let sk = sup(X_{i,m}), where i = len_start and m =examined(len_start)+1;
13.     if (sk < s)     /*switch to the upward search*/
14.       len_start=len_start+1; len_end=len_end+1; IsUpward=true;
15.     else     /*remain as the downward search*/
16.       len_start=len_start-1; len_end=len_end-1;
17.   }
18.   StairCandGen(len_start,len_end,CandSet);
19.   if (|CandSet|>0) {
20.     forall transaction t ∈ D do
21.       countSupport(t,CandSet);
22.     T_k = T_k ∪ {c ∈ CandSet | c.count ≥ s };
23.     T_k = {top k itemsets in T_k};
24.     s = min{sup(x) | x∈T_k }; /*The new minimum support threshold*/
25.   }
26.   else {
27.     Return T_k;
28.     Program End;
29.   }
30. }
```

utilizing $TB[x][y][z]$ via a binary search approach (presented in Procedure binarySearch). For example, we may calculate $\widehat{C}_{2,1}(1000) = 14,235$, $\widehat{C}_{2,1}(2000) = 40,792$, and $\widehat{C}_{2,1}(3000) = 75,851$ in advance. Assuming the available upper number of candidates, m, is 30,000, we can find that n_{opt} lies in the range between 1,000 and 2,000, and then examine n =1,500, 1,750, 1,625 by the binary search approach. Since $\widehat{C}_{2,1}(1625) = 29,666$, which is closest to 30,000, 1,625 will be suggested as n_{opt}. Note that the suggested result may not be the optimal n_{opt}, but they are indeed close to each other. This approach will substantially reduce the time to search n_{opt}.

Detailed subprocedures of the *MTK* algorithm are then outlined. Among them, Procedure StairCandGen, i.e., the major distinguishing innovation of the *MTK* algorithm, is used to generate candidate itemsets with the specified range of itemset-lengths. Note that the generation of candidate 2-itemsets can be pruned by H_2, and sometimes the upper bound of candidates derived in Theorem 1 may deviate from the exact number of generated candidates. Thus we may have the remaining memory after the regular process described in Sect. 3.2 (line 1–line 9 of Procedure StairCandGen), which can be further used to generate candidates in the same database scan. Inspired from the downward closure property, the remaining memory will be shared to generate more candidate itemsets with shorter itemset-length in this database scan since more valid itemsets could be identified (line 10–line 21 of Procedure StairCandGen). Actually, in last few database scans such as the fifth and the sixth database scans in Fig. 5b, a large memory usually remains unoccupied after the regular process. Therefore applying such an approach will reduce the number of database scans.

To gain more efficiency, we also utilize the selective scan technique in the *DHP* algorithm [5] to directly generate candidate i-itemsets from $(i-2)$-itemsets, or $(i-3)$-itemsets if we still have the remaining memory space (line 23–line 33 of Procedure StairCandGen). Specifically, different from the *DHP* algorithm, the selective scan in *MTK* must comply with the memory constraint. Therefore, we will first examine whether candidate i-itemsets generated from discovered high-support i-itemsets plus candidate $(i-1)$-itemsets in current candidate set, i.e., CandSet, can be generated under the memory constraint. It is worth mentioning that, the selective scan will prominently reduce the I/O operations and will lead to the high execution efficiency.

4.2 Extension of *MTK* to mine top-k closed itemsets

In essence, depending on the application need, mining *top-k closed* itemsets may be equally or more important than mining *top-k* itemsets. The reason lies in that many itemsets will be "redundant" among *top-k* frequent itemsets due to the

Procedure: InitStep()
/*the initial step of MTK*/
1. forall transaction $t \in D$ do
2. insert and count 1-items occurrences in a hash tree;
3. forall 2-subsets x of t do
4. $H_2[h_2(x)]$++;
5. $L_1 = \{ c | c.count \geq s,\ c$ exists in the leaf node of the hash tree$\}$;
6. $T_k = \{$top k 1-items in $L_1\}$;
7. $s = min\{sup(x) \mid x \in T_k\}$; /*The new minimum support threshold*/
8. len_start=2; len_end=1+δ; CandSet=$\emptyset$;
9. for (i=len_start; i<len_end;i++) /*Generate candidate i-itemsets from 1-items*/
10. $M_i = M/(\delta \times size_i)$;
11. n_i =binarySearch(M_i, 1, i);
12. $L = \{$top n_i 1-items in $T_k\}$; /* $|L|$ may be smaller than n_i*/
13. $C_i = \emptyset$;
14. if ($i == 2$) /*perform the hash pruning*/
15. forall $c \in L * L$ do
16. if (H2[h2(c)]$\geq s$) then $C_i = C_i \cup \{c\}$;
17. else
18. $C_i = \{$all i-subsets of $L\}$;
19. CandSet=CandSet$\cup C_i$;
20. foreach transaction $t \in D$ do
21. countSupport(t,CandSet);
22. $T_k = T_k \cup \{ c \in$ CandSet $| \ c.count \geq s \}$;
23. $T_k = \{$top k itemsets in $T_k\}$;
24. Let $examined(i)$ be equal to the number of i-itemsets whose all generated candidate (i+1)-itemsets have been tested, len_start$\leq i \leq$len_end;

Procedure: StairCandGen(start,end,CandSet)
/*generate $\delta-$Stair candidate itemsets which is able to stored in the available memory upper bound*/
1. for (i=start; i<end;i++)
2. $M_i = M/(\delta \times size_i)$;
3. n_{i-1} =binarySearch($M_i + tested(i), i-1, 1$);
4. $L_{i-1} = \{$top n_{i-1} $(i-1)$-itemsets in $T_k\}$; /* $|L_{i-1}|$ may be smaller than n_i*/
5. Let $t_s = sup(X_{i-1,m})$, where m =examined(i-1);
6. $nL_{i-1} = \{\eta \mid \eta \in L_{i-1}, sup(\eta) \leq t_s\}$;
7. $examined(i-1) = |L_{i-1}|$;
8. genCandidate(nL_{i-1}, L_{i-1}, C_i);
9. CandSet=CandSet$\cup C_i$;
10. Let M_r be the remaining memory, and $i = \min\{\ell \mid sup(X_{\ell,m}) > s\}$, where $m = examine(\ell - 1)$;
11. while ($M_r != 0$ and $i \leq end$) /*If the memory space remains, prioritize to discover itemsets with the shorter length*/
12. $M_i = M_r/(size_i)$;
13. n_{i-1} =binarySearch($M_i + tested(i), i-1,1$);
14. $L_{i-1} = \{$top n_{i-1} $(i-1)$-itemsets in $T_k\}$; /* $|L_{i-1}|$ may be smaller than n_i*/
15. if ($n_{i-1} \leq$examined(i-1)) then break; /*mean that the remanding memory cannot generate more candidate i-itemsets*/
16. Let $t_s = sup(X_{i-1,m})$, where m =examined(i-1);
17. $nL_{i-1} = \{\eta \mid \eta \in L_{i-1}, sup(\eta) \leq t_s\}$;
18. $examined(i-1) = |L_{i-1}|$;
19. genCandidate(nL_{i-1}, L_{i-1}, C_i);
20. CandSet=CandSet$\cup C_i$;
21. i++;
22. Let $i = start + 1$;
23. while ($M_r != 0$) /*If the memory space remains, directly generate candidates from candidates*/
24. $M_i = M_r/(size_i)$;
25. $nL_{i-1} = \{\eta \mid \eta \in$ CandSet , $|\eta| = i-1\}$
26. $L_{i-1} = nL_{i-1} \cup \{\eta \mid \eta \in T_k$, $|\eta| = i-1\}$;
27. n_{i-1} =binarySearch($M_i + tested(i), i-1,1$);
28. if ($n_{i-1} \geq |L_{i-1}|$) /*mean that we can directly generate candidate i-itemsets from candidate (i-1)-itemsets*/
29. genCandidate(nL_{i-1}, L_{i-1}, C_i);
30. CandSet=CandSet$\cup C_i$;
31. i++;
32. else /*the remaining memory cannot be utilized to generate more candidate i-itemsets*/
33. break;

Procedure: genCandidate(nL_{i-1},L_{i-1},C_i)
/*generate candidate i-itemsets from L_{i-1}, excluding candidates tested in previous database scans*/
1. $C_i = \emptyset$;
2. $F_i' = \{X \cup X' \mid X \in L_{i-1},\ X' \in nL_{i-1},\ |X \cap X'| = i-2\}$
3. $C_i = C_i \cup F_i$;
4. $tested(i) = tested(i) + |C_i|$; /*indicate how many candidate i-itemsets have been tested*/

Procedure: int binarySearch($cand\_num$, item_length, predicted_step)
/*find the size of itemsets to generate at most $cand\_num$ candidates*/
1. while (left$\leq$right) {
2. middle=$\left\lfloor \frac{right+left}{2} \right\rfloor$;
3. if (right==left)
4. return TB[item_length][predicted_step][middle].$itemset\_num$;
5. else if (TB[item_length][predicted_step][middle].$bound\_cand > cand\_num$)
6. right=middle-1;
7. else left=middle+1;
8. }
9. return TB[item_length][predicted_step[right].$itemset\_num$;

Procedure: countSupport(t,CandSet)
/*accumulate the support count of candidates*/
1. forall c such that $c \in$CandSet and $c \in t$ do begin
2. c.count++;
3. end

Algorithm: MTK_Close(k, M, δ)
/*additional global variables*/
$TC_k = \emptyset$; /*the repository to maintain top-k closed frequent itemsets, which is sorted by (1) the support of the itemset; (2) the length of the itemset*/
$nTC_k = \emptyset$; /*contains all itemsets which are not closed itemsets and can be expanded from closed itemsets in TC_k*/
$T_k = \emptyset$; /*T_k is always pointed to be as $TC_k \cup nTC_k$*/
/*Main Program*/
/*lines 1~18 refer to algorithm MTK*/
19. if (|CandSet|>0) {
20. forall transaction $t \in D$ do
21. count_support(t,CandSet);
22. forall $c \in$CandSet do
23. if ($c.count \geq s$)
24. $TC_k = TC_k \cup c$;
25. forall $\{c' \,|\, c' \in TC_k, |c'| = |c| - 1, sup(c') = sup(c)\}$ do
26. if (c' is a subset of c)
27. remove c' from TC_k;
28. $nTC_k = nTC_k \cup c'$;
29. TC_k = {top k itemsets in TC_k};
30. $s = \min\{sup(x) \,|\, x \in TC_k\}$; /*The new minimum support threshold*/
31. remove all itemsets whose supports are smaller than s from nTC_k;
32. Let $h = \min\{\ell \,|\, sup(X_{\ell,m}) \leq s\}$, where $m = examine(\ell - 1)$;
33. remove all itemsets whose lengths are smaller than or equal to h from nTC_k;
34. }
35. else {
36. Return {top k closed itemsets in TC_k};
37. Program End;
38. }
Procedure: InitStep()
/*the modified part in MTK_Close*/
/*lines 1~24 refer to procedure InitStep()*/
25. forall $c \in$CandSet do
26. if ($c.count \geq s$) {
27. $TC_k = TC_k \cup c$;
28. forall $\{c' \,|\, c' \in TC_k, |c'| = |c| - 1, sup(c') = sup(c)\}$ do
29. if (c' is a subset of c)
30. remove c' from TC_k;
31. $nTC_k = nTC_k \cup c'$;
32. }
33. TC_k = {top k itemsets in TC_k};
34. $s = \min\{sup(x) \,|\, x \in TC_k\}$; /*The new minimum support threshold*/
35. remove all itemsets whose supports are smaller than s from nTC_k;
36. Let $examined(i)$ be equal to the number of i-itemsets whose all candidate (i+1)-itemsets have been tested, len_start$\leq i \leq$len_end;

downward closure property (given any itemset in the *top-k* set, all its subsets certainly belong to *top-k* frequent itemsets) [26]. Accordingly, we in this section present the solution, specifically called the *MTK_Close* algorithm, to retrieve *top-k closed* itemsets extended from the *MTK* algorithm.

Explicitly, following from Definition 3, the basic idea behind *MTK_Close* is to maintain all itemsets with supports exceeding $\sup_{\min}(TC_k)$ and to progressively filter out unnecessary itemsets, i.e., itemsets with support equal to one of the superset. Therefore, as extended from Lemma 3, we can devise the *MTK_Close* algorithm by initially setting the minimum support threshold equal to zero and raising the threshold equal to $\sup_{ck}(w)$ after the wth database scan, where $\sup_{ck}(w)$ denotes the support of the kth most frequent closed itemset we have discovered after the wth database scan.

The main program of the *MTK_Close* algorithm, extended from the *MTK* algorithm, is outlined below. Other subprocedures of the *MTK* algorithm will carry over in the *MTK_Close* algorithm (only a slight modification of procedure InitStep is required). Specifically, as shown in Fig. 7, the structure to maintain *top-k* itemsets, T_k, in the *MTK* algorithm is a two-level sorted array, permitting itemsets to be inserted and removed according to their supports and itemset-lengths. In the *MTK_Close* algorithm, an identical structure, i.e., TC_k, is utilized to maintain the discovered *top-k closed* itemsets. When a new j-itemset X_j with the support exceeding the up-to-date minimum support s is identified, $(j-1)$-itemsets with the same support in TC_k will be examined whether they are subsets of X_j. Accordingly, any $(j-1)$-itemset which is identified as a subset of X_j and has the support equal to $\sup(X_j)$ will be removed from TC_k.

Another difference between *MTK* and *MTK_Close* is that the *MTK_Close* algorithm also maintains a structure, called nTC_k, to store itemsets which are not closed itemsets and can be expanded from *closed* itemsets in TC_k. Note that to precisely generate *top-k closed* itemsets, all candidate itemsets generated from all itemsets whose supports exceed $\sup_{\min}(TC_k)$ need to be tested. Therefore, nTC_k is required to be maintained to complement the set of itemsets which supports exceed $\sup_{\min}(TC_k)$. Importantly, not all itemsets with supports exceeding $\sup_{\min}(TC_k)$ have to be always maintained. Note that all frequent but non-closed i-itemsets will be covered by all closed $(i+1)$-itemsets [21]. When we have discovered all $(i+1)$-itemsets whose supports exceed the up-to-date minimum support s, all i-itemsets in nTC_k can be removed since they will not affect the further discovery of closed itemsets (line 31–line 33 in the *MTK_Close* algorithm). The program of the *MTK_Close* algorithm is then presented. Other supplemental procedures can be found in Sect. 4.1.

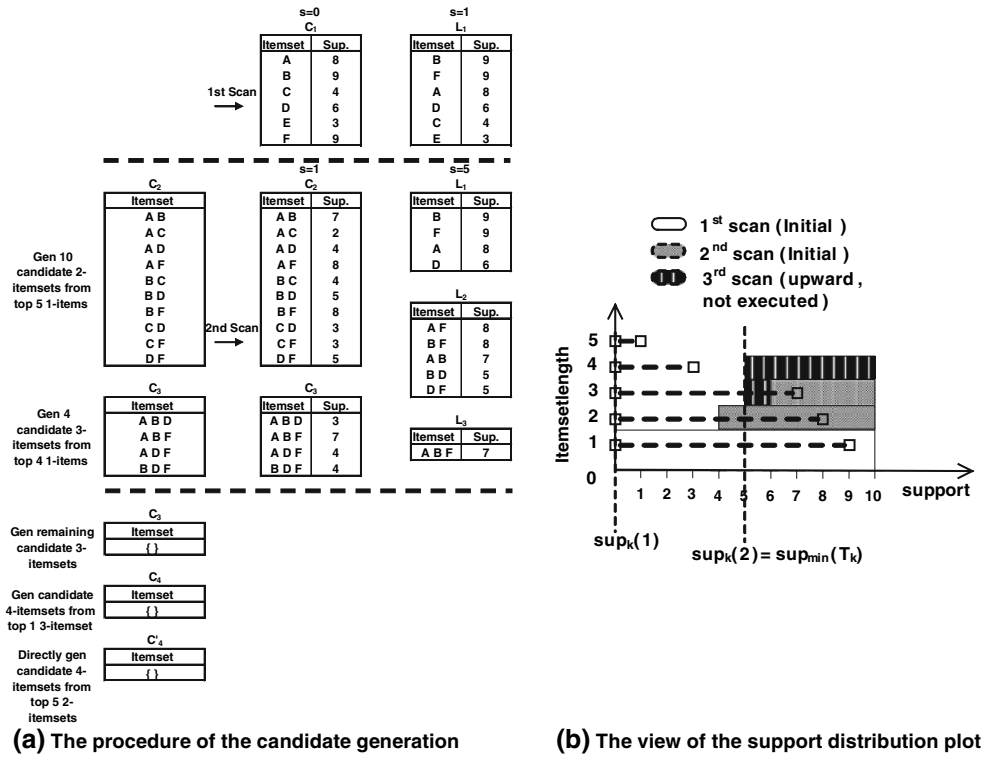


Fig. 8 The illustrative example of discovering *top-k* frequent itemsets

4.3 Illustrative running examples

We give the illustrative examples to exhibit the execution of algorithms *MTK* and *MTK_Close* by applying the dataset in Table 1. For ease of exposition, we assume: (1) the memory occupied to maintain a candidate i-itemset will be equal to i memory units, proportional to the size of the itemset-length; (2) the memory to maintain *top-k* frequent/closed itemsets will not be considered; (3) the hash table, i.e., H_2, will not be applied in the example; (4) $\delta = 2$.

Example of top-k frequent itemsets with the memory constraint: We perform the *MTK* algorithm to retrieve top ten frequent itemsets under the constraint that 40 memory units can be utilized to store candidate itemsets, as shown in Fig. 8, where Fig. 8a shows the process of the candidate generation in each database scan and Fig. 8b shows the corresponding view of the support distribution plot. Without knowing $\sup_{\min}(T_k)$ in advance, the up-to-date minimum support, s, will be equal to zero initially. After performing the first scan, we will obtain all 1-items associated with their supports and s will be raised to one. Afterward, we equally share the 40 memory units to generate candidate 2-itemsets and candidate 3-itemsets, including at most $40/(2 \times 2) = 10$ candidate 2-itemsets and at most $\lfloor 40/(2 \times 3) \rfloor = 6$ candidate 3-itemsets. Since $\widehat{C}_{1,1}(5) = 10$, we will generate candidate 2-itemsets from top five 1-items. Moreover, since $\widehat{C}_{1,2}(4) = 4$ and $\widehat{C}_{1,2}(5) = 10$, we can select top four 1-items to directly generate their candidate 3-itemsets. The second scan will be performed to examine the supports of all those generated candidates. We then insert each itemset into T_k iff the support of the itemset exceeds $s = 1$. Once $|T_k| > 10$, we remove the itemset with the smallest support from T_k. As a result, we obtain the set of high-support itemsets $\{B\}, \{F\}, \{A\}, \{D\}, \{AF\}, \{BF\}, \{AB\}, \{BD\}, \{DF\}$, and $\{ABF\}$ while updating s to five, which is equal to $\sup_k(2)$ shown in Fig. 8b.

We then generate candidates for the next scan. Since the set of 3-itemsets L_3 in T_k is not empty, the third scan will be executed as an upward search, with itemset-lengths between 3 and 4. Explicitly, $|L_3|$ is equal to 1, leading to no candidate 4-itemset can be generated from top one 3-itemset $[\because \widehat{C}_{3,1}(1) = 0]$. In addition, since in the second scan, we have tested candidate 3-itemsets which are directly generated from top four 1-items, i.e., $\{A\}, \{B\}, \{D\}$ and $\{F\}$, we can find that no candidate 3-itemset needs to be generated. Since we still have a remaining memory space, we directly generate candidate 4-itemsets from 2-itemsets. However, according to Theorem 1, there will be no candidate 4-itemsets generating from five discovered 2-itemsets $[\because \widehat{C}_{2,2}(5) = 0]$. As such, no candidate can be generated, and thus the program ends. The final set of top-ten itemsets consists of $\{A\}, \{B\}, \{D\}, \{F\}, \{AB\}, \{AF\}, \{BD\}, \{BF\}, \{DF\}$ and $\{ABF\}$, which are consistent with the result shown in Table 2. Importantly, only two database scans are required by *MTK*.

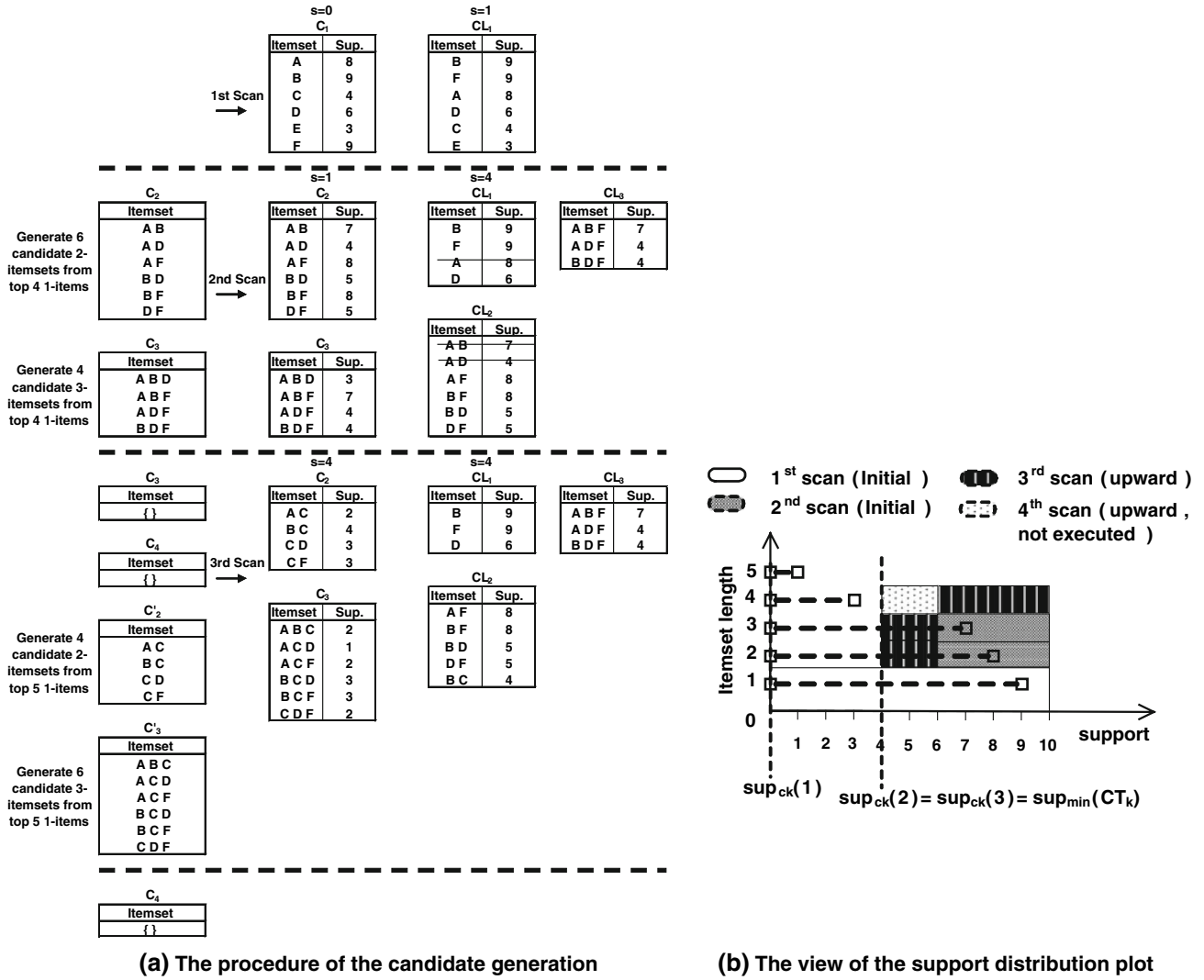


Fig. 9 The illustrative example of discovering *top-k* closed itemsets

Note that a level-wise based algorithm needs at least three scans in this case even though the minimum support can be specified in advance. Hence the efficiency of the *MTK* algorithm and the merit of *δ-stair search* are shown. □

Example of *top-k* closed itemsets with the memory constraint: We perform the *MTK_Close* algorithm to retrieve top ten closed itemsets under the constraint that only 26 memory units can be utilized to store candidate itemsets, as shown in Fig. 9 . Explicitly, after the first database scan, we can generate six candidate 2-itemsets from top four 1-items ($\because \lfloor 26/(2 \times 2) \rfloor = 6$, $\widehat{C}_{1,1}(4) = 6$), and can generate four candidate 3-itemsets from top four 1-items ($\because \lfloor 26/(2 \times 3) \rfloor = 4$, $\widehat{C}_{1,2}(4) = 4$). Thus after the second scan, we identify their supports, and insert the itemsets into TC_k if their supports exceed the up-to-date minimum support s. Importantly, the 1-item $\{A\}$ will be removed from TC_k when we insert 2-itemsets $\{AF\}$ since their supports are the same. Similarly, 2-itemsets $\{AB\}$ and $\{AD\}$ will also removed when we insert 3-itemsets $\{ABF\}$ and $\{ADF\}$, respectively. As a result, after the second database scan, we update s equal to four, and identify closed itemsets $\{B\}, \{F\}, \{D\}, \{Af\}, \{BF\}, \{BD\}, \{DF\}, \{ABF\}, \{ADF\}$ and $\{BDF\}$.

Since the set of 3-itemsets in TC_k is not empty, we perform the third scan as an upward search. Note that we cannot generate any candidate 4-itemset from discovered 3-itemsets [$\because \widehat{C}_{3,1}(3) = 0$]. In addition, no candidate 3-itemset will be generated from discovered 2-itemsets since all 3-itemsets with support exceeding $\sup(X_{2,6})$ were also discovered in the second scan, where $\sup(X_{2,6}) = 4$ and $X_{2,6}$ is $\{AD\}$, which is maintained in nTC_k (see Sect. 4.2). Therefore, the remaining memory space can be utilized to generate candidates with shorter itemset-lengths, i.e., candidate 2-itemsets in this case. We then generate all candidate 2-itemsets from 1-items whose supports exceed s, and find four candidate 2-itemsets (excluding those tested in previous

scans) need to be tested. The remaining memory is also utilized to directly generate candidate 3-itemsets. Note that five 1-items have supports exceeding s, i.e., $\{B\}$, $\{F\}$, $\{A\}$, $\{D\}$, and $\{C\}$, and at most ten candidate 3-itemsets will be generated [$\because \widehat{C}_{1,2}(5) = 10$]. Thus excluding those 3-itemsets tested in previous scans, at most six candidate 3-itemsets will be included, which can be generated without exceeding the memory constraint (four candidate 2-itemsets and six candidate 3-itemsets totally occupy 26 memory units). Finally, we have the set of candidates tested in the third scan. The 2-itemset $\{BC\}$ will be inserted into TC_k after the third scan and the up-to-date threshold s will not be raised because the minimum support in TC_k is still equal to four.

The candidates for the next scan is then generated. Originally, the fourth scan will be executed as a downward search since no 4-itemset is identified. However, all 1-items, 2-itemsets, 3-itemsets with support exceeding s were discovered after the third scan. Moreover, only three 3-itemsets are discovered and $\widehat{C}_{3,1}(3) = 0$, indicating that no candidate 4-itemset can be generated. As a result, the program ends with three database scans, and the set of closed itemsets, which is consistent with the result in Table 2, are obtained finally. □

5 Experimental studies

The simulation model of our experimental studies is described in Sect. 5.1. In Sect. 5.2, we conduct empirical studies based on synthetic and real datasets to assess the performance of the *MTK* algorithm. The *MTK_Close* algorithm is evaluated in Sect. 5.3.

5.1 Simulation model

Datasets: Both real and synthetic datasets are evaluated in our studies. Among them, real datasets include seven benchmark datasets from different application domains,[7] and one large dataset from a 3C chain store in Taiwan. Those datasets are summarized in Table 3, where I_s denotes the distinct items in the dataset, $|D|$ denotes the number of transactions, $|S|$ denotes the data size, $T_{\max}$ denotes the maximum itemset-length, and T_{avg} denotes the average itemset-length. For evaluating the scalability of *MTK* and *MTK_Close* algorithms, a set of synthetic data are generated by the IBM dataset generator [2], where the average transaction lengths vary from 5 to 30. Each synthetic dataset consists of 1,000,000 transactions with 5,000 distinct items and with frequent patterns having average length of 4, to simulate sparse and dense retail datasets.

[7] Downloaded from the website, http://fimi.cs.helsinki.fi/data/, of the ICDM workshop on Frequent Itemset Mining, 2003.

Table 3 Parameters of real datasets

Name	I_s	$\|D\|$	$\|S\|$	$T_{\max}$	T_{avg}
3C_Chain	130,108	8,000,000	1.1 GB	87	5.4
Web_docs	5,267,656	1,692,082	1.4 GB	71,427	177.2
Accident	468	340,183	34 MB	51	33.8
Mushroom	119	8,124	1 MB	23	23
Retail	16,470	88,162	4 MB	76	10.3
Kosarak	41,270	990,002	31 MB	2,498	8.1
BMS-POS	1,657	515,596	10 MB	164	6.5
Webview1	497	59,601	1 MB	267	2.5

Simulation environment: All programs of the simulation are coded by C++ and performed on Windows XP in a 1.7 GHz IBM compatible PC with 2 GB of memory. We utilize Intel VTune™ Performance Analyzer to assess the exact memory usage required by each algorithm. We compare the efficiency of *MTK* with several algorithms. The first one is the *FPGrowth* algorithm [12] and the second one is the *DHP* algorithm [19], both with unlimited memory consumption, which will be denoted by "*FPGrowth*(∞)" and "DHP(∞)" in experimental figures, respectively. The third one is the *Naive* algorithm described in Sect. 2, which is denoted by "naive" in experimental figures. Both the hash pruning and the selective scan techniques, which are devised in the *MTK* algorithm, are also applied in the *Naive* algorithm to reduce its required number of database scans. To give the best credit to *DHP, FPGrowth*, and the *Naive* algorithm, $\sup_{\min}(T_k)$ is assigned to these three algorithms in advance in such a way that they will efficiently generate the same set of *top-k* frequent itemsets [note that assigning $\sup_{\min}(T_k)$ prior to the mining process is impractical and is only for the comparison purpose]. In addition, the fourth algorithm is the *BOMO* algorithm [6], which is downloaded from the author's website. *BOMO* is able to retrieve most N frequent i-itemsets, where i can be in a range specified by users. Inherently, *BOMO* is not designed for mining pure *top-k* frequent itemsets (N is in general smaller than 100 in their cases). For fair comparison, we modify the code to collect k most frequent i-itemsets, where i starts from one and will be progressively increased until *top-k* itemsets are retrieved. Same as the *DHP* algorithm, *BOMO* will be compared to other algorithms without the constraint of the memory size, to show its best efficiency of mining *top-k* frequent itemsets.

For examining the performance of the *MTK_Close* algorithm to retrieve *top-k closed* itemsets, we also implement the start-of-the-art algorithm, namely the *TFP* algorithm presented in [22]. Originally, algorithm *TFP* is designed to retrieve *top-k closed* itemsets with the constraint of the minimum itemset-length. In our consideration, determining the minimum itemset-length incurs another inconvenience to

Accidents

	∞	100MB		250MB	
k	DHP(∞)	naive	MTK	naive	MTK
100	4	3	3	3	3
1,000	5	4	4	4	4
10,000	6	6	6	5	5
100,000	7	7	8	7	7

Retail

	∞	100MB		250MB	
k	DHP(∞)	naive	MTK	naive	MTK
100	3	2	2	2	2
1,000	3	3	4	3	3
10,000	4	4	4	3	3
100,000	7	7	8	6	6

Kosarak

	∞	100MB		250MB	
k	DHP(∞)	naive	MTK	naive	MTK
100	4	3	3	3	3
1,000	4	3	4	3	3
10,000	5	6	6	4	4
100,000	9	9	9	7	8

BMS-POS

	∞	100MB		250MB	
k	DHP(∞)	naive	MTK	naive	MTK
100	3	2	2	2	2
1,000	4	3	4	3	3
10,000	5	5	5	4	4
100,000	7	7	8	6	6

Mushroom

	∞	100MB		250MB	
k	DHP(∞)	naive	MTK	naive	MTK
100	4	4	4	3	3
1,000	5	4	4	3	3
10,000	6	6	7	5	5
100,000	8	8	9	7	7

Webview1

	∞	100MB		250MB	
k	DHP(∞)	naive	MTK	naive	MTK
100	3	3	3	2	2
1,000	4	4	4	3	3
10,000	5	5	6	4	4
100,000	6	6	6	5	5

Fig. 10 The number of required database scans in six real datasets ($\delta = 2$)

users, which conflicts the purpose of mining *top-k* itemsets to release users from the determination of subtle parameters. As such, the minimum itemset-length in *TFP* is set as one in our comparison. Note that as validated in [22], algorithm *TFP* outperforms traditional methods of mining closed itemsets such as the CHARM algorithm [26]. Hence we only compare the performance of *MTK_Close* and *TFP* due to their similar purposes.

In our implementation of *MTK* and *MTK_Close*, several pre-computed variables are specified as: (1) the structure TB, i.e., the sorted array to indicate the upper number of candidates, is a three-dimensions integer array with size [50][6][1000] (around 2 MB in our implementation); (2) the available memory consumption is specified as either 100 or 250 MB, which are the reasonable memory sizes permitted for a single mining process in the multi-task mining system; (3) with roughly probing in advance, the factor Φ, which is used to approximate the memory overhead of a candidate itemset, is set equal to 2 so that we have the reasonable mapping between the real specified memory constraint and the upper number of candidates.[8]

5.2 Experiments on mining top-k frequent itemsets

We first investigate the efficiency of the *MTK* algorithm on real datasets. Figure 10 shows the number of database scans required by *DHP*, *Naive* and *MTK* (with $\delta = 2$) in six small real datasets under variant numbers of k. The *BOMO* and *FPGrowth* algorithms are not shown since they always require two scans to build the FP-tree in memory. As can be seen, the number of database scans required by *MTK* is identical to that required by the *Naive* algorithm in most cases, indicating that the efficiency of *MTK* is nearly optimal (as indicated by Remark 3, *Naive* is one of the most efficient level-wise algorithm for mining *top-k* itemsets under the memory constraint). In addition, we can observe that *MTK* and *Naive* sometimes have the number of database scans smaller than that required by the *DHP* algorithm, particularly when the upper available memory is 250 MB. It is because that *MTK* and *Naive* can concurrently generate itemsets of various lengths if the memory space is affordable. The result shows that, even though the memory consumption is constrained in the *MTK* algorithm, it still can be executed without compromising the execution efficiency.

We then investigate the performance of *MTK* with δ varying from 1 to 6. Specifically, we apply these two large real datasets, Webdocs and 3C_Chain, to study in passing the efficiency of *MTK* on data with long transactions (Webdocs) and data with sparse, short transactions (3C_Chain). In Fig. 11, we can see that when $\delta = 2$ or 3, the resulting number of database scans of *MTK* under different sizes of k is equal to that of the *Naive* algorithm (the optimal result of the level-wise *top-k* mining algorithms). Oppositely, when $\delta = 1$, or δ is large, we find that the number of database scans increases a lot. That shows the drawback of the *horizontal first search approach* and the *vertical first search approach* described in Sect. 3.1. As can be seen, $\delta = 2$ or 3 is the best choice of δ since the merit of the *horizontal first search approach* and the *vertical first search approach* are integrated into *MTK* while their drawbacks are diminished.

We also observe the maximum memory consumption and the execution time of different algorithms with various k, where the results are shown in Figs. 12 and 13 by applying the Webdocs and 3C_Chain real datasets. To ensure the execution can be finished within an acceptable execution time, k will be limited between 2,000 and 12,000, which is the reasonable setting of the desired number of frequent itemsets.

Importantly, *BOMO* cannot handle the whole dataset since the complete FP-tree cannot be built in our system (2 GB memory). The Webdocs and 3C_Chain datasets are thus applied to *BOMO* via sampling (reduced to 500 and 700 MB, respectively) so as to have the FP-tree in the main memory at the cost of its resulting precision. As can be seen in Fig. 12, the memory requirement of *DHP* drastically increases as k increases, in both datasets. In addition, the memory required by *FPGrowth* slightly increases as k increases, but the required memory is large as compared to *DHP* when k is small, which confirms the study in [11]. On the other hand, the maximum memory consumption of *MTK* and *BOMO* will be fixed, but the one of *BOMO* is much larger than *MTK* since *BOMO* builds a complete FP-tree initially (note that the result of *BOMO* is based on sampling, indicating that a larger memory is required while the whole dataset is applied). Apparently, the effectiveness of *BOMO* is acquired at the cost of the large memory consumption, which may not be guaranteed below the available memory. In our experiments,

[8] Note that $\Phi = 2$ is a conservative estimation of the mapping, and thus the upper memory occupied by *MTK* and *MTK_Close* is a little bit smaller than the specified 100 or 250 MB.

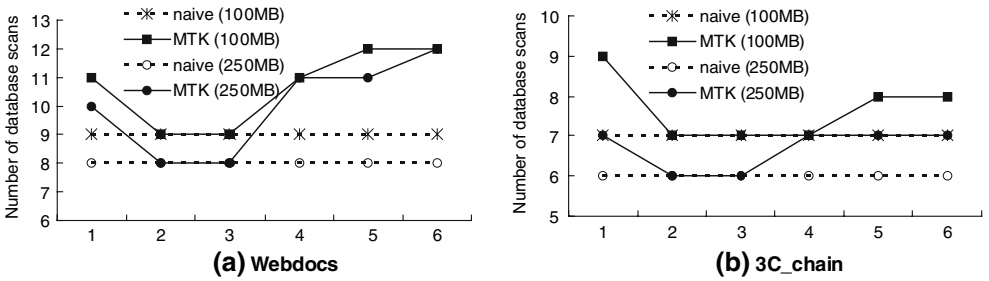


Fig. 11 The number of database scans under varied δ in large real datasets ($k = 10,000$)

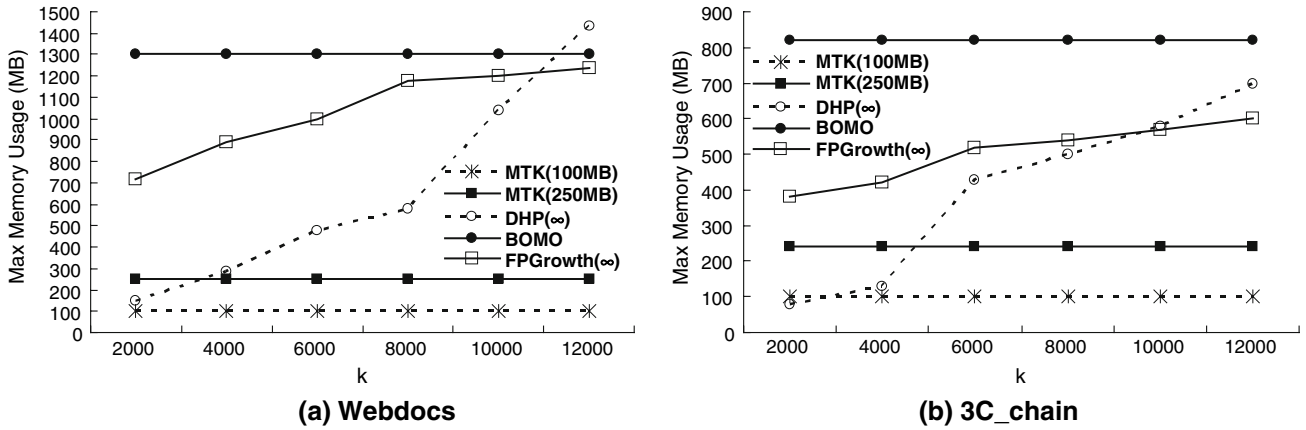
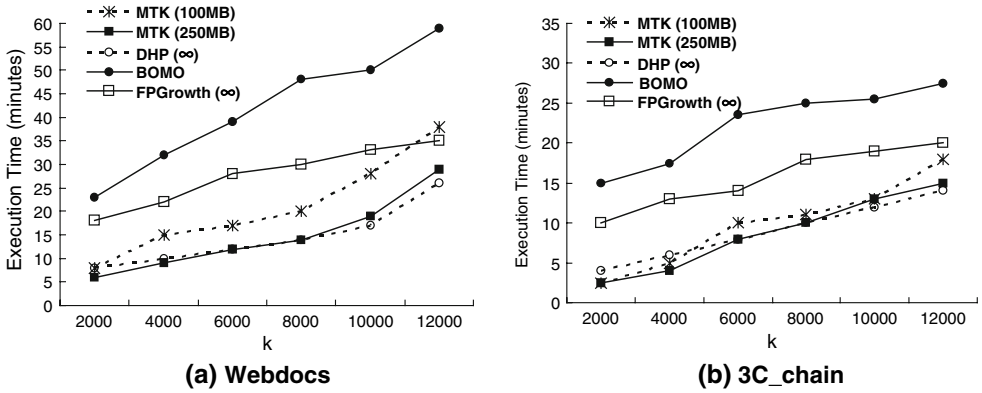


Fig. 12 The memory requirement of different algorithms with various k ($\delta = 2$)

Fig. 13 The execution time of different algorithms with various k ($\delta = 2$)



its maximum memory consumption is around 1.3 GB, which is not affordable for most PCs.

Moreover, as shown in Fig. 13, *BOMO* cannot efficiently retrieve *top-k* frequent itemsets since *BOMO* retrieves *top-k* frequent itemsets while also finding a lot of itemsets not belonging to *top-k* frequent itemsets, which is similar to the phenomenon of the *horizontal first search approach* in Sect. 3.1. Actually, the major reason lies in that *BOMO* is not specifically devised for mining *top-k* frequent itemsets despite it has the most similar goal. In contrast, *MTK* has the excellent efficiency, which is close to that of *DHP* (even better than that of *DHP* when k is small or the available memory size is large). Importantly, we can see that *MTK* and *DHP* outperform *FPGrowth* when $k \leq 10{,}000$. As also validated in many previous studies [11,14,18,27], the performance of level-wise search algorithms is better than that of depth-first search algorithms when the minimum support is not very small. Our result also confirms the conclusion. In general, users may be interested in less than 10,000 frequent itemsets (it is not prevalent to make marketing decisions according to more than 10,000 itemsets), and $k \leq 10{,}000$ usually corresponds to a reasonable minimum support (not a

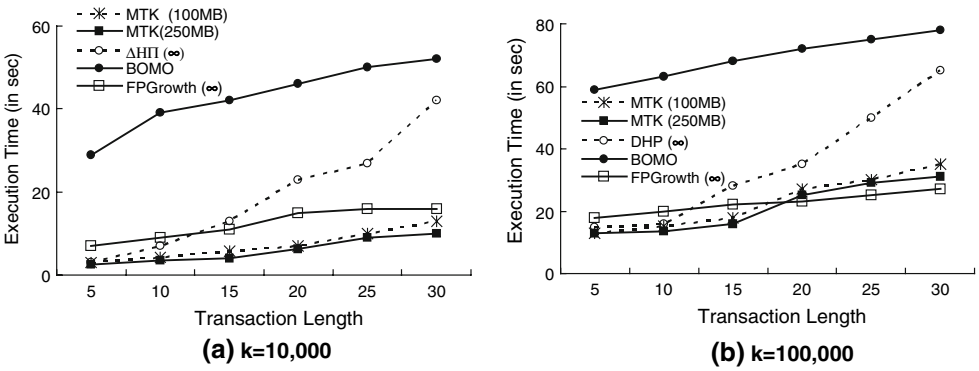


Fig. 14 The execution time under various transaction lengths ($\delta = 2$)

too small one) either in the dense database (Web_docs) or in the spare database (3C_Chain). In such cases, *MTK* can not only achieve the high mining efficiency but also constrain the memory usage. Although when $k > 10,000$, *FPGrowth* results in a better efficiency than *MTK* in the dense database (Web_docs) at the cost of a large memory consumption, *MTK* still has a better feasibility because *MTK* can constrain the memory usage without much comprising the mining efficiency. In addition, *FPGrowth* must have the minimum support with respect to *top-k* itemsets in advance, which is infeasible and only for the comparison purpose. Apparently, both considering the efficiency and the practicability, *MTK* will be the best algorithm to retrieve *top-k* frequent itemsets in the presence of the memory constraint.

Furthermore, it can be seen that, even though only 100 MB memory can be utilized, the *MTK* algorithm still leads to the prominent efficiency as compared to other algorithms. The reason is that the maximum memory consumption in a level-wise algorithm usually appears to generate candidate 2-itemsets and 3-itemsets. Formally, *MTK* may pay for one or two more database scans to generate 2-itemsets and 3-itemsets but may only use one database scan to concurrently generate-and-test 5-itemsets, 6-itemsets and 7-itemsets and so on. It is attributed to the merit of the selective scan technique, showing the prominent applicability of the *MTK* algorithm.

To more precisely demonstrate the above investigation, we study the scalability of the *MTK* algorithm by applying synthetic datasets, where the results are shown in Fig. 14. Interestingly, we can see that the execution time of *MTK* only slightly increases as the average transaction length increases either when $k = 10,000$ or when $k = 100,000$. In contrast, the execution time of *DHP* drastically increases. It is mainly attributed to that *MTK* can fully utilize the available memory. Thus *MTK* requires similar numbers of database scans in various average transaction lengths since candidates with various long itemset-lengths can be generated-and-tested in the same scan. In *MTK*, the execution overhead with respect to the long average transaction length results from the increase of the number of combinations in each transaction, which indeed can be efficiently executed in memory and leads to the better efficiency than *DHP*. Furthermore, the execution time of *FPGrowth* steady increases as the average transaction length increases. In practice, while the database is dense (the average transaction length is long) and k is large ($k = 100,000$), the execution time of *MTK* is larger than that of *FPGrowth*. However, *MTK* still has the competitive efficiency in such cases. It is worth mentioning that *FPGrowth* is executed with unbounded memory and the minimum support to retrieve *top-k* itemsets is given in advance, thus degrading its applicability. As a consequence, the result on synthetic data also validates both the practicability and the efficiency of *MTK*.

5.3 Experiments on mining *top-k* closed itemsets

We here investigate the performance of the *MTK_Close* algorithm, as compared to the state-of-the-art algorithm, *TFP* [22]. Basically, *TFP* constructs a complete FP-tree in memory with the minimum support equal to zero initially. For fair comparison, the minimum itemset-length constraint [22] is not imposed on this experiment. In Fig. 15, we show the performance of these two algorithms with k varying from 100 to 100,000 in six real datasets. It is clear to see that *MTK_Close* outperforms *TFP* when k is small, but *TFP* sometimes outperforms *MTK_Close* when k is large. The reason is that the set of *top-k* itemsets and the set of *top-k closed* itemsets are apparently different to each other only when the minimum support is small [27], corresponding to a large k. Therefore the minimum support to retrieve top 100,000 itemsets may be relatively small and FP-tree based approaches are advantageous in such cases. Nevertheless, k is generally required less than 100,000 in real cases (users will not make their marketing decisions from more than 100,000 frequent itemsets). Therefore *MTK_Close* still has the competitive efficiency in practice.

On the other hand, as shown in Fig. 16, the memory usage required by *TFP* is extremely large while the database is large (same as the experiment on the *BOMO* algorithm, the Webdocs and 3C_Chain datasets are reduced to 500 and 700 MB

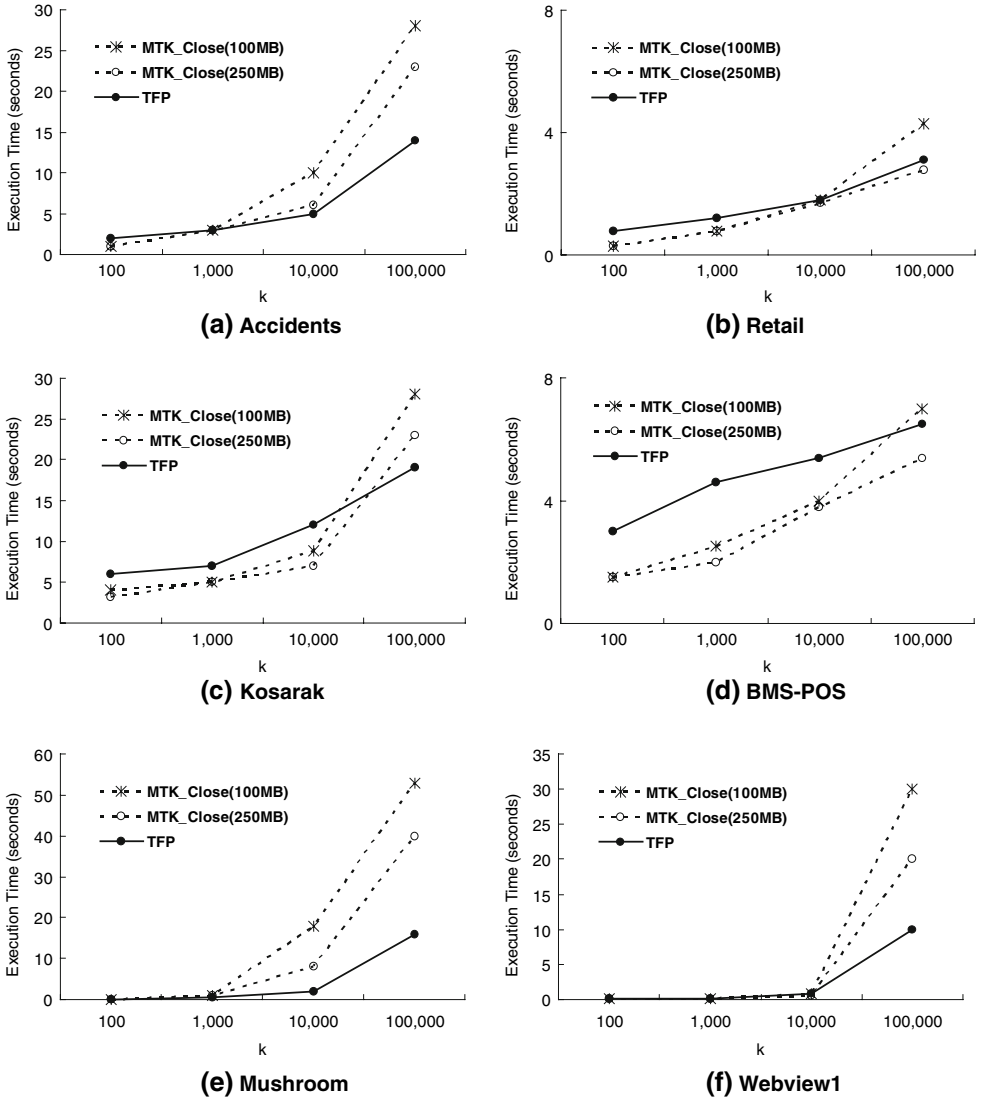


Fig. 15 The performance of mining *top-k* closed itemsets with various k ($\delta = 2$)

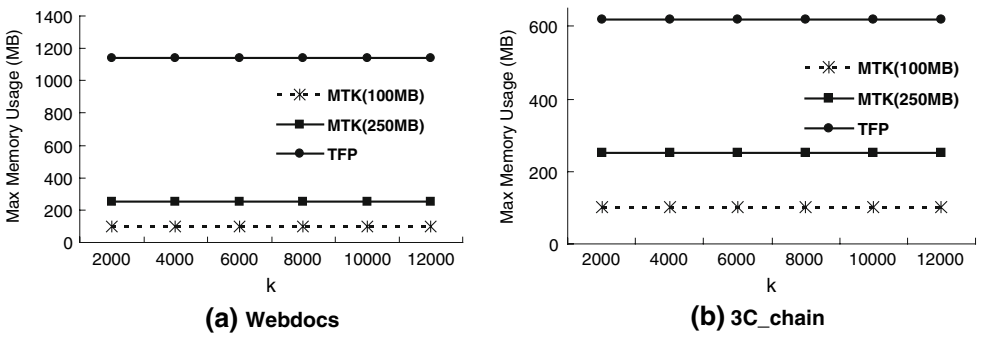


Fig. 16 The upper memory consumption of different *top-k* closed mining algorithms with various k ($\delta = 2$)

by sampling, respectively, to ensure the complete FP-tree can be constructed in memory). Note that *TFP* prunes the FP-tree on the fly after the tree were constructed in memory, and thus its upper memory usage is equal to the size of a complete FP-tree. In contrast, *MTK_Close* can retrieve *top-k closed* itemsets in the available memory. Figure 17 shows the corresponding execution time as k varies from 2,000 to 12,000. As can be seen, *MTK_Close* still has the competitive efficiency, showing its prominent advantages to be a practical algorithm for mining *top-k closed* itemsets.

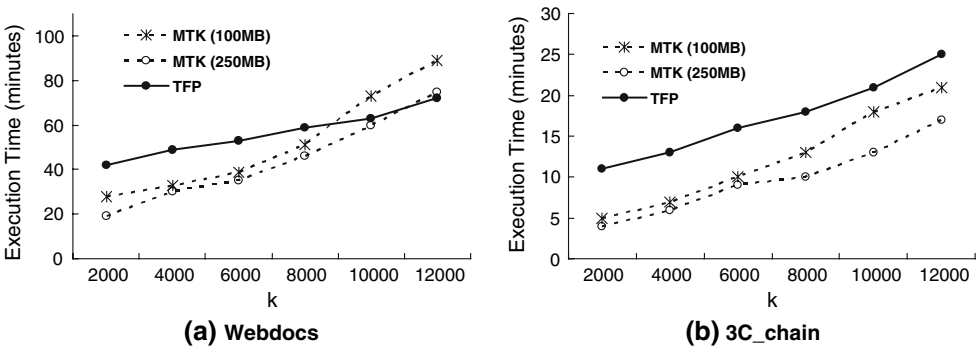


Fig. 17 The execution time of *top-k* closed mining algorithms with various k ($\delta = 2$)

6 Conclusions

In this paper, we have studied a practically important mining problem, namely mining *top-k* frequent/closed itemsets in the presence of the memory constraint. To achieve this, we proposed the *MTK/MTK_Close* algorithms, which are devised as level-wise search algorithms based on an effective approach to constrain the number of candidates that will be generated-and-tested in each database scan. Since the minimum support to retrieve *top-k* frequent itemsets cannot be known in advance, a novel search approach, called the *δ-stair search*, is devised in *MTK* and *MTK_Close* to efficiently retrieve *top-k* frequent/closed itemsets. As demonstrated in the empirical study on real data and synthetic data, instead of only providing the flexibility of striking a compromise between the execution efficiency and the memory consumption, *MTK* can both achieve high efficiency and have a constrained memory bound, showing its prominent advantage to be a practical algorithm of mining frequent patterns.

References

1. Afrati, F., Gionis, A., Mannila, H.: Approximating a collection of frequent sets. In: Proceedings of ACM SIGKDD (2004)
2. Agrawal, R., Srikant, R.: Fast algorithms for mining association rules. In: Proceedings of VLDB (1994)
3. Babcock, B., Babu, S., Datar, M., Motwani, R., Widom, J.: Models and issues in data stream systems. In: Proceedings of PODS (2002)
4. Brin, S., Motwani, R., Ullman, J., Tsur, S.: Dynamic itemset counting and implication rules for market basket data. In: Proceedings of SIGMOD (1997)
5. Chen, M.-S., Park, J.-S., Yu, P.S.: Efficient data mining for path traversal patterns. IEEE Trans. Knowledge Data Eng. **10**(2), 209–221 (1998)
6. Cheung, Y.L., Fu, A.W.: Mining association rules without support threshold: with and without item constraints. In: TKDE (2004)
7. Chi, Y., Wang, H., Yu, P.S., Muntz, R.R.: Moment: maintaining closed frequent itemsets over a stream sliding window. In: Proceedings of ICDM (2004)
8. Geerts, F., Goethals, B., Bussche, J.V.D.: Tight upper bounds on the number of candidate patterns. ACM Trans. Database Syst. **30**(2), 333–363 (2005)
9. Goethals, B.: Survey on frequent pattern mining, online technical report. http://www.adrem.ua.ac.be/bibrem/pubs/fpm_survey.pdf (2003)
10. Goethals, B.: Memory issues in frequent itemset mining. In: Proceedings of SAC (2004)
11. Goethals, B., Zaki, M.J.: Advances in frequent itemset mining implementations: introduction to FIMI03. In: Proceedings of Workshop on Frequent Itemset Mining Implementations (2003)
12. Han, J., Pei, J., Yin, Y.: Mining frequent patterns without candidate generation. In: Proceedings of ACM SIGMOD (2000)
13. Han, J., Pei, J., Yin, Y., Mao, R.: Mining frequent patterns without candidate generation: a frequent-pattern tree approach. In: DMKD (2004)
14. Hipp, J., Guntzer, U., Nakhaeizadeh, G.: Algorithms for association rule mining—a general survey and comparison. In: SIGKDD Explorations (2000)
15. Manku, G.S., Motwani, R.: Approximate frequency counts over streaming data. In: Proceedings of VLDB (2002)
16. Mannila, H., Toivonen, H., Verkamo, A.I.: Efficient algorithms for discovering association rules. In: AAAI Workshop on Knowledge Discovery in Databases (KDD-94) (1994)
17. Orlando, S., Lucchese, C., Palmerini, P., Perego, R., Silvestri, F.: kDCI: a multi-strategy algorithm for mining frequent sets. In: Proceedings of Workshop on Frequent Itemset Mining Implementations (2004)
18. Orlando, S., Palmerini, P., Perego, R., Silvestri, F.: Adaptive and resource-aware mining of frequent sets. In: Proceedings of IEEE ICDM (2002)
19. Park, J.-S., Chen, M.-S., Yu, P.S.: An effective hash based algorithm for mining association rules. In: Proceedings of ACM SIGMOD (1995)
20. Park, J.S., Chen, M.-S., Yu, P.S.: Using a hash-based method with transaction trimming for mining association rules. IEEE Trans. Knowl. Data Eng. **9**(5), (1997)
21. Pasquier, N., Bastide, Y., Taouil, R., Lakhal, L.: Discovering frequent closed itemsets for association rules. In: Proceedings of ICDT (1999)
22. Wang, J., Han, J., Lu, Y., Tzvetkov, P.: TFP: an efficient algorithm for mining top-k frequent closed itemsets. In: TKDE (2005)
23. Wong, R.C.-W., Fu, A.W.: Mining top-k itemsets over a sliding window based on zipfian distribution. In: Proceedings of SIAM SDM (2005)
24. Xiao, Y., Dunham, M.H.: Considering main memory in mining association rules. In: Proceedings of DAWAK (1999)
25. Yu, J.X., Chong, Z., Lu, H., Zhou, A.: False positive or false negative: mining frequent itemsets from high speed transactional data streams. In: Proceedings of VLDB (2004)
26. Zaki, M.J., Hsiao, C.-J.: Charm: an efficient algorithm for closed itemset mining. In: Proceedings of SIAM SDM (2002)
27. Zheng, Z., Kohavi, R., Mason, L.: Real world performance of association rule algorithms. In: Proceedings of SIGKDD (2001)