Available online at www.sciencedirect.com
ScienceDirect
Applied Mathematics and Computation 185 (2007) 769–783
APPLIED MATHEMATICS AND COMPUTATION
www.elsevier.com/locate/amc

RFIMiner: A regression-based algorithm for recently frequent patterns in multiple time granularity data streams

Lifeng Jia *, Zhe Wang, Nan Lu, Xiujuan Xu, Dongbin Zhou, Yan Wang

College of Computer Science, Jilin University, Key Laboratory of Symbol Computation and Knowledge, Engineering of the Ministry of Education, Changchun 130012, China

Abstract

In this paper, we propose an algorithm for computing and maintaining recently frequent patterns which is more stable and smaller than the data stream and dynamically updating them with the incoming transactions. Our study mainly has two contributions. First, a regression-based data stream model is proposed to differentiate new and old transactions. The novel model reflects transactions into many multiple time granularities and can automatically adjust transactional fading rate by defining a fading factor. The factor defines a desired life-time of the information of transactions in the data stream. Second, we develop *RFIMiner*, a single-scan algorithm for mining recently frequent patterns from data streams. Our algorithm employs a special property among suffix-trees, so it is unnecessary to traverse suffix-trees when patterns are discovered. To cater to suffix-trees, we also adopt a new method called *Depth-first and Bottom–up Inside Itemset Growth* to find more recently frequent patterns from known frequent ones. Moreover, it avoids generating redundant computation and candidate patterns as well. We conduct detailed experiments to evaluate the performance of algorithm in several aspects. Results confirm that the new method has an excellent scalability and the performance meets the condition which requires better quality and efficiency of mining recently frequent itemsets in the data stream.

Keywords: Data mining; Data streams; Multiple time granularities; Recently frequent patterns; Suffix-trees

1. Introduction

Frequent pattern (or itemset) mining is a basic but hot research field in data mining problems, and thus has a broad application at mining association rules [1,2], correlation [5], sequential patterns [3], episodes [6], max-patterns [4], multi-dimensional patterns [7] and profit mining [8], and many other important data mining tasks.

The problem of frequent itemset mining can be defined as follows. Given a transactional database, let $I = \{I_1, I_2, \ldots, I_n\}$ be the complete set of items that appear in the database. Each transaction, $T = \{I_k, \ldots, I_m\}$, is an itemset, such that $k < m$ and $T \in I$, and is associated with a unique identifier, called TID. For any itemset P, let $T(P)$ is the corresponding set of TIDs, where $T(P) = \{T.\mathrm{TID} | P \subseteq T\}$. We say that $|T(P)|$ is the support

* Corresponding author.
E-mail addresses: jia_lifeng@hotmail.com (L. Jia), cgzhou@jlu.edu.cn (D. Zhou).

0096-3003/$ - see front matter © 2006 Elsevier Inc. All rights reserved.
doi:10.1016/j.amc.2006.06.115

of P. An itemset P is frequent if $|T(P)| \geqslant min\_sup$, where min_sup is the support threshold defined by users. The objective of frequent itemset mining is to discovery the complete set of frequent itemsets in transactional databases.

The majority of algorithms, including Apriori [2], FP-growth [9], H-mine [10], and OP [11], mine the complete set of frequent itemsets. Then, extended studies concerning closed frequent itemsets [12], maximal frequent itemsets [15], and mining representative itemsets [22] are proposed. Their tasks are to find a succinct presentation that describes the complete set of frequent itemsets accurately or approximately. Subsequently, some novel algorithms, including A-close [12], CLOSET [13], CHARM [14], MAFIA [15], and RPlocal and RPglobal [22] are proposed. All algorithms mentioned above have good performance in the sparse database with a high *min_sup* threshold. However, when the *min_sup* drops low or the size of database increases dynamically, their performances are influenced negatively. Unfortunately, a new kind of dense and large databases, such as network traffic analysis, web click stream mining, power consumption measurement, sensor network data analysis, and dynamic tracing of stock fluctuation, appears in recent emerging applications. They are called streaming data where data takes the form of continuous, potentially infinite data streams, as opposed to finite, statically stored databases. Data stream management systems and continuous stream query processors are under popular investigation and development. Besides querying data streams, another important task is to mine data streams for frequent itemsets.

Actually, the problem concerning mining frequent itemsets in large databases was first proposed by Agrawal et al. [1] in 1993. It has been widely studied and applied since the last decade. In the environment of data streams, mining frequent itemsets, however, becomes a challenging problem, because the information in the streaming data is huge and rapidly changing. Consequently, infrequent items and itemsets can become frequent later on and hence can not be ignored.

In my opinion, the best description of data stream is as follow. A data stream is a continuous, huge, fast changing, rapid, infinite sequence of data elements. According to this definition, we can draw a conclusion that the nature of streaming data makes the algorithm which only requires scanning the whole dataset once be devised to support aggregation queries on demand. In addition, this kind of algorithms usually owns a data structure far smaller than the size of whole dataset. Based on the discussion so far, the single scan requirement of streaming data model conflicts with the objective of frequent itemset mining which is to discovery the complete set of frequent itemsets. To harmonize this conflict, an estimation mechanism [16] is proposed in the Lossy Counting algorithm. Lossy Counting is a stream-based algorithm for mining frequent itemsets and utilizes the well-known Apriori property [2]: *if any length k pattern is not frequent in the database, its length (k + 1) super-patterns can never be frequent*, to discovery frequent itemsets in data streams.

The estimation mechanism is defined as follow. Given a maximum allowable error threshold ε as well as a minimum support threshold θ, the information about the previous results up to the latest block operation is maintained in a data structure called lattice. The lattice contains a set of entries of the form, (e,f,∇), where e is an itemset, f is the frequency of itemset e, and ∇ is the maximum possible error count of the itemset e. For each entry in the lattice, if e is one of the itemsets identified by new transactions, its previous count, f, is incremented by its count in new transactions. Subsequently, if its estimated count, $f + \nabla$, is less than $\varepsilon \cdot N$, such that N is the number of transactions, its entry is pruned from the lattice. On the other hand, when there is no entry in lattice for a new itemset identified by new transactions, a new entry, (e,f,∇), is inserted into the lattice. Its maximum possible error count is set to $\varepsilon \cdot N'$ where N'denotes the number of transactions that were processed up to the latest block operation before.

Besides the estimation mechanism, many other literatures present some ingenious technologies related with mining frequent itemsets in the data stream. Han et al. developed FP-stream [17], a FP-tree-based algorithm. FP-stream employs a novel titled-time windows technique and mines frequent itemsets at multiple time granularities. Ruoming and Agrawal developed StreamMining [18], a single-scan algorithm that mines frequent itemsets in data streams. StreamMining utilizes some fixed-size sets that decrease the frequencies of candidate itemsets contained by them whenever they are overflowed. Furthermore, because of particular characteristics of data streams, such as infinite and fast changing, new transactions might contain more valuable information than old transactions. So, the importance and practicability of mining recently frequent itemsets attract people's attention. Teng et al. proposed a regression-based algorithm called FTP-DS [19] to mine recently frequent itemsets by sliding windows for the first time. Chang et al. developed an algorithm called estDec [20]

for recently frequent itemsets in the data stream. In estDec, each transaction has a weight that decreases with age. Chang et al. [21] also proposed another single-scan algorithm that incorporates the estimation mechanism and sliding window concept to mine recently frequent itemsets in the data stream.

In this paper, we develop and interpret the following two techniques in order to efficiently mine recently frequent itemsets over data streams.

First, we propose a regression-based streaming data model where all transactions in streams are reflected into many multiple time granularities according to the time-sensitivity. Generally, new transactions are more time-sensitive and contain more valuable information, and thus should be attached more importance than old transactions. In the model, new transactions are usually assigned to a fine and small time granularity, and old ones are assigned to a coarse and big time granularity. When transactions transfers from a fine time granularity to a coarse one, which means transactions are in the progress of turning old and obsolete, their influence is weakened by fading factors to give prominence to the influence of new transactions. That fading factor actually defines a desired life-time of the information of transactions in the data stream.

Second, *RFIMiner*, a suffix-tree-based algorithm for discovering recently frequent itemsets over streams, is developed by us. By using a particular property between suffix-trees, the connectivity of two nodes in a suffix-tree is indirectly but fast confirmed by judging whether corresponding identical nodes synchronously occur in two specific suffix-trees. Frequent itemsets are discovered by the growth from their frequent subitemsets, so candidate itemsets are avoid. If we employ the traditional *bottom–up* or *top–down* strategies to make itemsets grow, a huge amount of redundant computation is generated. To cater to special properties of suffix-trees, a novel itemset growth method is also presented to release time-consumed but useless redundant computation.

The remaining of the paper is organized as follows. In Section 2, Section 2.1 first introduces some well-known strategies for mining recently frequent itemsets in data streams, and then discusses some minor drawbacks of those strategies. Section 2.2 presents the new streaming data model with multiple time granularities, demonstrates its principle, and analyzes and explains how our model covers those drawbacks by automatically adjusting the regression rate of transactions flowing in data stream. Section 3 gives out the idea of suffix-trees, discoveries and proofs an important property between suffix-trees. Then, a novel itemset growth method is proposed to optimize the performance of mining frequent itemsets from suffix-trees. The algorithm called *RFIMiner* is outlined by and large in the Section 4. Section 5 reports the results of our experiments and performance study. Eventually, Section 6 concludes the study in this paper.

2. Streaming data with multiple time granularities

In this section, we first study the previous regression strategies which are used for mining recently frequent itemsets in streaming data. Through carefully analyzing those strategies, we find some predicable but unpleasant situations that influence the result negatively. Then, we propose a novel streaming data model to overcome them.

2.1. Landmark model versus sliding window model

Previous work by Manku and Motwani [16] analyzed the landmark model. In this model, frequent itemsets are measured from the start of the data stream to the current moment and the different importance of old and new transactions are ignored. However, in many real applications, this difference among transactions is vital. For example, some commercial data streams, especially shopping transaction data streams, old and new transactions equally can not be treated as the same useful ones at guiding current business since some old items may have lost their attractions.

Based on the discussion so far, the core problem of mining recently frequent itemsets in the data stream is how to differentiate the valuable information contained by recent and new transactions from the useless information in outdated transactions. The regression strategies adopted by Teng et al. [19] and Chang et al. [20] solve this core problem successfully, especially the sliding window strategy that actually demonstrates the practicability of mining recently frequent itemsets in data streams. These two regression strategies have made a contribution to this problem. However, they still have some minor drawbacks. The FTP-DS algorithm only focuses on the transactions in the sliding window, and ignores transactions outside the sliding window. In

FTP-DS algorithm, the size of sliding window is a key point for this regression strategy. What is the right size for different streams? It is hard to answer. If the size of sliding window is too small, some useful transactions might flow out the window so soon that fail to contribute to the result timely. If the size of sliding window is too big, transactions in it might be of different importance. Whatever situation happens, the result might fail to give out the frequent itemset changes over data streams. Again, the estDec algorithm faces a similar problem as well, because it mines recently frequent itemsets in the data stream where each transaction is allocated with a weight that decreases with age. There also is a predicable and unpleasant situation as follow. Because the influence of an existing transaction is decreased whenever a new one arrives, some valuable transactions might be of uselessness too soon that fail to reflect frequent itemset changes in the data stream either.

2.2. Streaming data with multiple time granularities

Enlightened by the titled-time windows technique [17], a novel streaming data model is proposed to avoid above situations. In this new model, all transactions in the data stream are reflected into many multiple time granularities and their regression rates are adjusted by the model automatically.

Note that the multiple time granularities in our model are actually a kind of pseudo projection. In other words, different time granularities actually are divided by transaction identifiers (TIDs) or block identifiers (BIDs). For implementing this pseudo projection of multiple time granularities in the data stream, we propose a recursive algorithm which dynamically reflects the entire transactional stream into multiple time granularities by utilizing BIDs as the partition. This recursive algorithm called *TG-update* is presented in Section 5.

Before interpreting the streaming data model with multiple time granularities, we should know that BID is a more reasonable partition for time granularity than TID. For example, mining in a shopping transaction stream should give out the information concerning fashion and seasonal products. However, this information may change from time to time, not from one transaction to another. In other words, transactions generated during a period of time should be practically valuable only if they own the same influence on the result. So, BID is clearly a better choice for dividing time granularities.

The principle about how the model functions is described as follow. Given the arrival of a new block of transactions, A, the arrival of A actually causes the adjustment of a block of transactions, B, which is generated just before A. Therefore, B must vacuum its time granularity to store A. At this moment, B needs to judge whether its corresponding intermediate window is empty. Note that every time granularity is assigned with an intermediate window with the same size as the corresponding time granularity. If the intermediate window is empty, B just need to be stored in it to vacuum the time granularity for A; otherwise, supposed that a block of transactions, C, is already in the intermediate window, B incorporates with C and are stored in the next level time granularity with the twice size. Then, both the time granularity and the intermediate window are vacuumed for A. Similarly, the incorporated block of transactions leads to the adjustment of transactions that are already in the next level time granularity. We stipulate that frequencies of itemsets contained by transactions are lessened by a fading factor ϕ, whenever those transactions transfer between two next time granularities. Note that, the transaction's transference between a time granularity and its intermediate window fails to lead to the decrease in frequency. The principle of novel model is now illustrated with a simple example with seven assumed blocks of transactions.

Fig. 1 demonstrates the basic principle of streaming data model with multiple time granularities. We use B_i to denote a certain block of transactions, and assume that the smaller the subscript is, the older the block is. According to the principle described above, with the arrival of B_2, B_1 transfers into the intermediate window ([...] denotes an intermediate window). When B_3 arrives, B_2 incorporates with B_1 into B_{21}. B_{21} transfers from the original time granularity to the next level time granularity. Meanwhile, frequencies of itemsets in B_{21} are

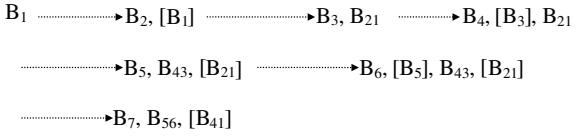

Fig. 1. Multiple time granularity streams.

lessened by the fading factor ϕ. Similarly, with the arrival of B_7, B_{43} incorporates with B_{21} into B_{41}, and B_{41} transfers to the next level time granularity for vacuuming its time granularity for storing B_{65} that is incorporated by B_6 and B_5. Then, frequencies of itemsets contained by B_{41} are lessened. Note that frequencies of itemsets in B_{21} are lessened twice, but those of itemsets in B_{43} are only done so once, because B_{43} is more time-sensitive and valuable than B_{21}. Finally, B_7 is stored in the time granularity which is vacuumed by B_6 and B_5.

How our streaming data model can automatically adjust the fading rate of transactions? To answer this question, a simple definition should be introduced first for better understanding the explanation.

Definition 1. *Regression rate*: $O(I)$ denotes the set of recently frequent itemsets contained by transactions. f_I denotes the frequency of itemset, I, the fading factor is denoted by ϕ. $T_{i,j}$ $(i < j)$ denotes the transference interval between i-level time granularity and j-level time granularity. So, the regression rate of transactions in i-level time granularity, denoted by R, is defined in the follow formula:

$$R = \frac{\sum_{I}^{I \in O(I)} f_I \times \phi}{T_{i,i+1}}.$$

Through analyzing the principle about how the streaming data model functions, we can draw the conclusion that the regression rates of transactions are automatically adjusted by this model to slow down with the lapse of time. According to the principle, transactions always flow from a small and fine time granularity to a wide and coarse one, when valuable transactions are turning old and obsolete. Therefore, new transactions are always in a smaller and finer time granularity than old ones. Again, transferences between two time granularities cause the regression of transactions. A wide time granularity actually protracts the transference interval between two-level time granularities. With the numerator and denominator of R actually getting smaller and bigger respectively, the regression rate is thus slowing down with the lapse of time. So the streaming data model with multiple time granularities guarantees that new and valuable transactions put more influence on recently frequent itemsets than old and obsolete transactions.

3. Mining frequent itemsets by suffix-trees

To mining frequent itemsets, an efficient data structure must be employed to store the summary information concerning transactions in the data stream. Han et al. [9] proposed FP-growth, a frequent itemset mining algorithm. In that method, transactions are compactly stored in FP-trees, because a set of transactions sharing the same subset items may share common prefix paths. However, it must scan the whole database ahead to get the *f_list* that consists of frequent items by the descending order in frequency. Similarly, mining frequent itemsets in the data stream also needs an efficient structure to compactly keep the summary information concerning stream. Tree structure instinctively is a good choice, because it has played key roles in various algorithms for different goals. However, we fail to scan the whole stream to make sure an *f_list* due to some particular characteristics of streaming data, such as continuity and infinite. Therefore, the estimation mechanism [16] combining with the idea of block-processed solves this problem well.

3.1. Suffix-forest

To mine recently frequent itemsets in the data stream, we choose suffix-trees to store the summary information of stream. In this subsection, we respectively study how to store the summary information in suffix-trees and how to utilize a newly-proofed property between suffix-trees to quickly discovery frequent itemsets.

What is a suffix-tree like? How is a suffix-tree constructed? Now, let us answer these two questions together by describing every step of constructing suffix-trees. First, when a new transaction, $\{I_1, I_2, \ldots, I_n\}$, arrives, it is reflected into all of its suffix-sets: $\{\{I_1, I_2, \ldots, I_n\}, \{I_2, I_3, \ldots, I_n\}, \ldots, \{I_{n-1}, I_n\}, \{I_n\}\}$. Second, these suffix-sets are viewed as many individual transactions, and are stored in different suffix-trees. Third, all identical nodes of suffix-tree are connected by a node link team (*NLT* for short). In this paper, we use "suffix-tree (I_i)" and "node (I_k)" to denote a suffix-tree whose root is item I_i and a node containing item I_k, respectively.

Based on the description above, suffix-trees keep some characteristics of FP-tree, such as *Node-Link property*, which facilitates the access of frequent itemset information related to I_i by traversing the suffix-tree once following the node-links of item I_i, but they still own some interesting property which quickens the speed of frequent itemset growth.

Why we adopt suffix-trees to store the summary information concerning transactions in the data stream? We now explain the reason. All itemsets contained by a transaction, $t = \{I_1, I_2, \ldots, I_n\}$, can be divided into n types: (1) itemsets containing I_1; (2) itemsets containing I_2, but no I_1;...; (n) itemsets only containing I_n, but no $I_1, \ldots, I_{n-1}$. Therefore, if our method mines every suffix-tree to only get a special type of frequent i-itemset ($i \geqslant 2$), sets of frequent itemsets contained by t are mined out after the search for the entire suffix-forest is over. For example, for the suffix-tree (I_i), we only mine the itemsets containing I_i, but no $I_1, \ldots, I_{i-1}$, because the itemsets containing $I_1, I_2, \ldots,$ or I_{i-1} will be mined in the suffix-tree (I_1),..., suffix-tree (I_{i-1}).

Property 1. *In a suffix-tree* (I_i)*, its subtree whose root is* $I_k (k > i)$ *must be a subtree of suffix-tree* (I_k)*.*

This property is directly from the suffix-tree construction process. When the algorithm are constructing the suffix-tree (I_i), supposed that a certain branch of subtree whose root is item I_k contains a certain suffix-set, $\{I_i, \ldots, I_k, \ldots, I_n\}$. Meanwhile, the suffix-set, $\{I_k, \ldots, I_n\}$, must appear together with this suffix-set. Our method either constructs a new branch of suffix-tree (I_k) to store the suffix-set or inserts it into an already existed branch of suffix-tree (I_k) by increasing frequencies of nodes in that branch. So the property is proofed.

According to Property 1, we can draw the conclusion as follow. If a certain node (I_i) of suffix-tree (I_m) has a corresponding node (I_i) in the suffix-tree (I_k), this node (I_i) must have an ancestor node (I_k) in the suffix-tree (I_m). In order to recognize corresponding identical nodes in different suffix-trees, we need to code nodes in suffix-trees. The coding method is described as follow. We first suppose that all nodes in the complete suffix-tree are coded by the depth first. Then, all nodes of the actual (complete or incomplete) suffix-tree are endowed with the same serial numbers as those they own in the corresponding complete suffix-tree. According to the method above, there exists a special relationship between the corresponding identical nodes in different suffix-trees: *In the suffix-tree*(I_i)*, the difference of the serial number of node* (I_n) *and that of its ancestor node* (I_m) *is equal to the serial number of its corresponding node* (I_n) *in the suffix-tree* (I_m). Technically, we can use this relationship to identify whether a node in a suffix-tree has a corresponding identical node in another specific suffix-tree.

Lemma. *In an assumed suffix-tree* (I_1)*, the set of child nodes is denoted by* $O(I_1) = \{I_2, \ldots, I_{i+1}\}$*, and* $f(I_{i+1})$ *and* $c(I_{i+1})$ *denote node* (I_{i+1})*'s frequency and serial number respectively. For each node* (I_{i+1}) *in the suffix-tree* (I_1)*, if it has a corresponding node* (I_{i+1}) *in suffix-tree* (I_k)*, its serial number is stored in the serial number set* C_k $(1 < k < i + 1)$*. Let the total serial number set for nodes* (I_{i+1}) (*T-set for short*)*,* *T-set*(*node* (I_{i+1})) $= C_2 \cap \ldots \cap C_i$*. If* $C \neq \emptyset$*, the frequency of* $(i + 1)$*-itemset* $\{I_1, \ldots, I_{i+1}\} = \sum f(I_{i+1})$*, such that* $c(I_{i+1}) \in C$*; Otherwise, the frequency of* $(i + 1)$*-itemset* $\{I_1, \ldots, I_{i+1}\} = 0$*.*

Proof. To compute the frequency of $(i + 1)$-itemset, $\{I_1, \ldots, I_{i+1}\}$, we need to find all nodes (I_{i+1}) which are the common child nodes of node (I_2),..., node (I_i) in the suffix-tree (I_1). According to Property 1, if a certain node (I_m) of suffix-tree (I_i) has a corresponding node (I_m) in the suffix-tree (I_k), it must have an ancestor node (I_k) in the suffix-tree (I_i). Thus, the set C_k actually stores serial numbers of nodes(I_{i+1}) that have nodes (I_k) as ancestor nodes, If there is the occurrence of nodes (I_{i+1}) which are the common child nodes of node (I_1),..., node (I_i), serial numbers of these nodes (I_{i+1}) must appear in the *T-set*(node (I_{i+1})). Easily understood, the frequency of $(i + 1)$-itemset, $\{I_1, \ldots, I_{i+1}\}$, is equal to the sum of frequencies of these nodes (I_{i+1}), $\sum f(I_{i+1})$, such that $c(I_{i+1}) \in C$. Otherwise, if the *T-set*(node (I_{i+1})) is empty, the frequency of $(i + 1)$-itemset, $\{I_1, \ldots, I_{i+1}\}$, must be equal to zero, because there is no nodes (I_{i+1}) is the common child nodes of node (I_1),..., node (I_i). □

Based on the above statement, in our method, the connectivity of two nodes in a suffix-tree is indirectly but fast confirmed by judging whether corresponding identical nodes synchronously occur in two specific suffix-trees, and thus traversing the suffix-trees is unnecessary during the process of itemset growth. In other words, our method distracts the research attention from the relationship of nodes in a tree to the corresponding identical nodes in different suffix-trees.

Mining frequent itemsets in the data stream requires devised algorithms to be competent in the scalability. A doubt might be proposed based on the description concerning suffix-trees. Actually, suffix-trees might consume more memory to store transactions than their original sizes, because a real transaction is reflected into many pseudo transactions sharing different suffixes. Although their occurrences speed up the itemset growth, they indeed cost more memory space. The estimation mechanism with the block-processed idea solves this problem well. Suffix-trees that store each block of transactions cost more space, but the space is still much smaller then the size of entire streaming data and is timely released after mining frequent itemsets from them. The temporal increase in memory space is tolerable and acceptable.

Example 1. The two columns in Table 1 show the transactional database in our running example. Suppose $min\_sup = 1$ and the *f_list* is the alphabetical.

Now, let us illustrate the concept of suffix-tree with this simple example. First, all transactions are projected into many pseudo transactions sharing different suffixes as follow.

Suffix-sets$(a, c, d) = \{(a, c, d), (c, d), (d)\}$;
Suffix-sets$(a, b, d) = \{(a, b, d), (b, d), (d)\}$;
Suffix-sets$(b, d) = \{(b, d), (d)\}$;
Suffix-sets$(b, c, d) = \{(b, c, d), (c, d), (d)\}$;

The *f_list* is $\{a, b, c, d\}$, and suffix-trees that store the transaction database are shown in the Fig. 2. The corresponding complete suffix-tree is given next to each suffix-tree. Because suffix-tree (b), suffix-tree (c), and suffix-tree (d) are already complete ones, their complete trees are omitted. Every node of suffix-tree contains three

Table 1
The transactional database

TID	Set of items
100	a, c, d
200	a, b, d
300	b, d
400	b, c, d

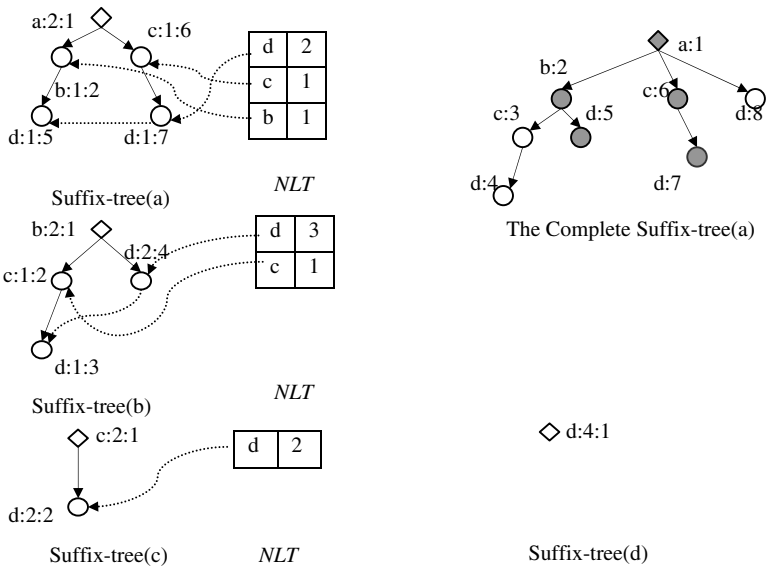

Fig. 2. Suffix-trees storing the information concerning the database in Example 1.

aspects of information, including *node_name*, *frequency*, and *serial number*. Note that corresponding identical nodes in complete and actual suffix-tree (*a*) have the same serial numbers. Therefore, the shaded nodes of complete suffix-tree (*a*) on the right are the real nodes of actual suffix-tree (*a*) on the left.

3.2. Depth-first and bottom–up inside itemset growth

Any algorithms devised for streaming data should keep pace up with the rapid arrivals of transactions. In our method, suffix-trees are adopted to store transactions and the indirect confirmation of the connectivity in suffix-trees boosts the speed at mining frequent itemsets. Actually, the computation of *T-set*s for various nodes costs the majority of executing time. However, we discovery that no matter what item sequence, such as *top–down* or *bottom–up*, is employed, there is too much redundant computation concerning *T-set* during the itemset growth.

Now, we will introduce a novel itemset growth method which not only avoids generating candidate itemsets, but also optimizes the intermediate computation of *T-set*. Before introducing it, we first define two core operations as follow.

Definition 2. *Insert Itemset Growth* (*IIG for short*): When the frequency of *i*-itemset, $\{I_1,\ldots,I_m,\ldots,I_k\}$, has already been computed, through the operation of insert itemset growth, the frequency of $(i+1)$-itemset, $\{I_1,\ldots,I_p,I_m,\ldots,I_k\}$, is computed, such that I_p is the newly inserted item just next to I_m in the counter item sequence.

Definition 3. *Replace Itemset Growth* (*RIG for short*): When the frequency of *i*-itemset, $\{I_1,\ldots,I_m,\ldots,I_k\}$, has already been computed, through the operation of replace itemset growth, the frequency of *i*-itemset, $\{I_1,\ldots,I_p,\ldots,I_k\}$, is computed, such that I_p is the newly inserted item to replace I_m and is just next to I_m in the counter item sequence.

We stipulate that the priority of *IIG* operation is higher than that of *RIG* operation, and that the newly-inserted item sequence by both *IIG* and *RIG* operations is according to the counter item sequence. Next, we describe the process of *Depth-first and Bottom–up Inside Itemset Growth* with an assumed suffix-tree (I_1) containing $i-1$ kinds of child nodes: node (I_2),…, node (I_i). Supposed that we has already computed frequencies of all 2-itemsets during the construction of suffix-tree (I_1). First (1), the algorithm executes *IIG* operation to 2-itemset $\{I_1,I_i\}$, and then generates a bigger itemset beginning with I_1 and ending with I_i. Our method generates 3-itemset $\{I_1,I_{i-1},I_i\}$, where I_{i-1} is the newly inserted item next to I_i in the counter item sequence. According to Lemma, *T-set*(node (I_i)) contains serial numbers of nodes (I_i) that are common child nodes of both node (I_1) and nodes (I_{i-1}) in the suffix-tree (I_1). Second (2), we continues to make 3-itemset $\{I_1,I_{i-1},I_i\}$ grow to $\{I_1,I_{i-2},I_{i-1},I_i\}$ by *IIG* operation. At this moment, our method only needs to judge whether some of nodes (I_i) contained by *T-set*(node (I_i)) are child nodes of nodes (I_{i-2}) in the suffix-tree (I_1), and then get a subset of *T-set*(node (I_i)) where serial numbers of nodes (I_i) which are common child nodes of nodes (I_1), nodes (I_{i-2}) and nodes (I_{i-1}) in the suffix-tree (I_1) are kept. When the method makes an *n*-itemset, $\{I_1,I_{i-n+2},I_{i-n+3},\ldots,I_i\}$, grow to an infrequent $(n+1)$-itemset, $\{I_1,I_{i-n+1},I_{i-n+2},I_{i-n+3},\ldots,I_i\}$, by *IIG* operation, the method will turn to execute the *RIG* operation to this *n*-itemset for generating another *n*-itemset, $\{I_1,I_{i-n+1},I_{i-n+3},\ldots,I_i\}$. At that moment, our algorithm only continues to compute the subset of *T-set*(node (I_i)) that is already generated for the $(n-1)$-itemset, $\{I_1,I_{i-n+3},\ldots,I_i\}$.

Compared with traditional itemset growth methods, our method is different in three aspects: First (1), because *IIG* operation has a higher priority than *RIG* operation, it always searches for frequent itemsets by depth-first. Second (2), the item sequences by *IIG* and *RIG* operations are the counter item sequence, i.e., our itemset growth process begins with leaf nodes of suffix-tree and ends with the root node. Third (3), the growth actually happens from the inside of itemset, not from the head or the end. Consequently, our novel itemset growth method is named after *Depth-first and Bottom–up Inside Itemset Growth*.

Let us illustrate this novel method with the Example 1. The data structure called lattice is devised to cater to *Depth-first and Bottom–up Inside Itemset Growth*. The result of frequent itemsets in the database of Example 1 is shown in Fig. 3. In the lattice of suffix-tree (*a*), the unit c in the list 2 represents the frequent itemset, $\{a,c,d\}$, because it is generated from the frequent 2-itemset, $\{a,d\}$, by *IIG* operation. Similarly, the unit b below that

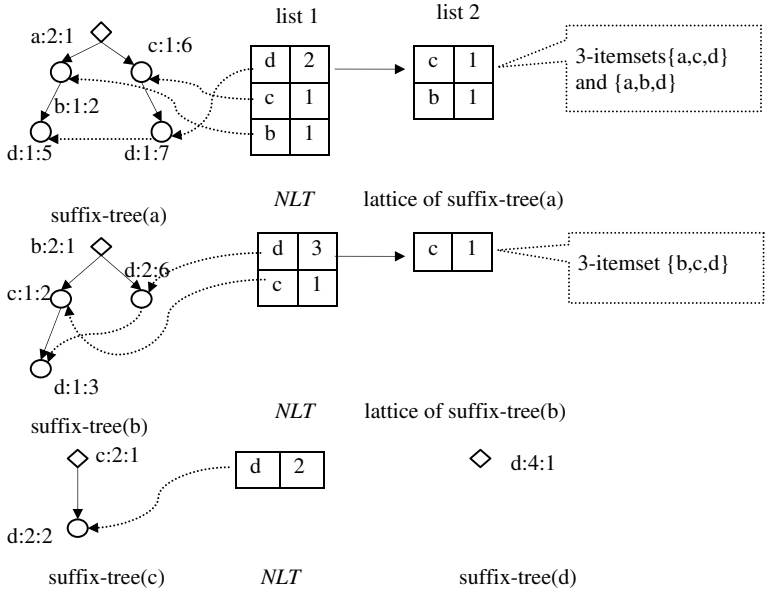

Fig. 3. Lattices for storing all frequent itemsets in the database of Example 1.

unit c represents the frequent itemset, $\{a,b,d\}$, because it is generated from the frequent 3-itemset, $\{a,c,d\}$, by *RIG* operation.

After a detailed description concerning *Depth-first and Bottom–up Inside Itemset Growth*, we can draw the conclusion that our method is more efficient than some other strategies, such as *top–down search* or *bottom–up search*, in the model of suffix-trees. Let us illustrate the process of *top–down search* in suffix-trees with Example 1. When all frequent 2-itemset, $\{a,b\}$, $\{a,c\}$, $\{a,d\}$, are discovered, frequent itemset, $\{a,b,c\}$, will be generated next. Therefore, the assumed algorithm that incorporates the suffix-tree model and *top–down search* has to turn to computed the *T-set*(node (c)). It has to use extra memory to store the *T-set*(node (d)) and fail to discard it, because *T-set*(node (d)) is still useful when the algorithm computes other itemsets, such as $\{a,b,d\}$ and $\{a,c,d\}$. However, *Depth-first and Bottom–up Inside Itemset Growth* always focuses on a *T-set* until all frequent itemsets related with it are mined. When a *T-set* for a kind of nodes is discarded, no more frequent itemsets related with it will be generated. Similarly, the *bottom–up search* faces the same problem.

4. Algorithm

In this section, we describe the algorithm for constructing and maintaining the streaming data with multiple time granularities and suffix-trees structures in more detail. In particular, we incorporate the property between suffix-trees and *Depth-first and Bottom–up Inside Itemset Growth* method into the high-level description of the algorithm given in previous sections.

4.1. RFIMiner algorithm

In this subsection, we give the description concerning our novel algorithm for mining recently frequent itemsets in the data stream.

The construction of suffix-trees and the update to lattice are bulky, done only when enough incoming transactions have arrived to form a new block B_i. The *RFIMiner* algorithm treats the first block of transactions differently from the rest as an initialization step. Moreover, we propose a recursive subalgorithm that reflects continuously arriving transactions into multiple time granularities to divide the different time granularities and this method is called *TG-update* uses *block identifier*s.

Algorithm 1. ***RFIMiner*** (Mining recently frequent itemsets in the data stream)

INPUT: a block of transactions B; minimal support threshold θ; maximal estimated possible error threshold ε; fading factor ϕ.

OUTPUT: recently frequent itemsets in lattice.

METHOD:

According to the estimation mechanism, blocks of transactions in the data stream are treated as follow:

For each new block of transactions B
  Construct suffix-trees to store the summary information of B
  Call *TG-update* (*BID*, 0);
  For every suffix-tree constructed for storing transactions in B
    Regression Check (each 2-itemsets in *NLT*)
    For each frequent itemset in *NLT* c;
      Depth-first and Bottom–up Inside Itemset Growth (c);
    Delete infrequent itemsets from *NLT* and suffix-trees;

End of *RFIMiner*

Algorithm 2. ***TG-update*** (Incremental adjustment of transactions in the streaming model with multiple time granularities)

INPUT: the *block identifier* of transactions *BID* and the *level identifier* for a certain time granularity *LID*.

OUTPUT: an array of *BID*s to divide the stream into multiple time granularities *TG*

METHOD:

Initialize an array for intermediate window *IW*, an array for time granularity *TG*.

Begin the recursive update process of time granularities as follow.

```
IF (TG [LID] ≠ 0)
{
  IF (IW [LID] = 0)
  {
    IW [LID] ← TG [LID];
    TG [LID] ← BID;
  } ELSE
  {
    IW [LID] ← 0;
    Call TG-update (TG [LID], LID + 1);
    TG [LID] ← BID;
} ELSE
{
  TG [LID] ← BID;
}
```

End of *TG-update*

Algorithm 3. *Depth-first and Bottom–up Inside Itemset Growth*

INPUT: an itemset c.

OUTPUT: the updated lattice structure.

METHOD:

```
d ← IIG(c) || d is the lattice generated from c by IIG operation.
IF (Contained (d))
{
  Regression Check (d);
```

```
    IF (frequent) Update the frequency of d. Call Depth-first and Bottom–up Inside Itemset Growth (d)
} ELSE
{
    Regression Check (d);
    IF (frequent) Call Depth-first and Bottom–up Inside Itemset Growth (d)
}
d ← RIG(d)
While d ≠ NULL
{
    IF (Contained (d))
    {
        Regression Check (d);
        IF (frequent) Update the frequency of d; Call Depth-first and Bottom–up Inside Itemset Growth (d);
    } ELSE
    {
        Regression Check (d);
        IF (frequent) Call Depth-first and Bottom–up Inside Itemset Growth (d);
    }
    d ← RIG(d)
}
```

End of *Depth-first and Bottom–up Inside Itemset Growth*

4.2. Discussion

In this subsection, some discussions are given out for better understanding *RFIMiner*, *TG-update* and *Depth-first and Bottom–up Inside Itemset Growth* algorithms. Some further explanations concerning these three algorithms are omitted, because they have already been presented in previous sections.

4.2.1. The usage of NLTs in suffix-trees

First, we make a discussion concerning the *NLT*s of suffix-trees. Actually *NLT* contains all 2-itemsets in every block of transactions, of course including recently frequent 2-itemsets. Consequently, *RFIMiner* always checks frequencies of 2-itemsets in *NLT*s and makes sure whether they are recently frequent or not. If so, the algorithm makes *IIG* and *RIG* operations on them to generate bigger itemsets. After all recently frequent itemsets are mined, suffix-trees and infrequent units in *NLT*s are deleted and recently frequent units of *NLT*s are kept as lattices.

4.2.2. Infrequent pruning

Although the description about how *Depth-first and Bottom–up Inside Itemset Growth* mines frequent itemsets is presented in previous sections, the objective of *RFIMiner* is to discovery recently frequent itemsets in the data stream. Therefore, we incorporate the *Regression Check* into *Depth-first and Bottom–up Inside Itemset Growth* in our algorithm. *Regression Check* actually judges whether transactions that contain recently frequent itemsets transfer between two time granularities. It also deletes infrequent itemsets due to regression from the lattice, which is called *Infrequent Pruning*. Note that the transference between time granularities actually is the dynamic division of time granularities in streaming data by *TG-update* algorithm.

Consequently, another discussion concerning pruning infrequent itemsets form lattice is made here. Because recently frequent itemsets are mined by Itemset Growth idea, infrequent itemsets can be pruned based on the well-known Apriori property [2]: *if any length k pattern is not frequent in the database, its length (k + 1) super-patterns can never be frequent*. Therefore, a property could be proposed to make the update of lattice efficiently. This property is directly from the previous description about *Depth-first and Bottom–up Inside Itemset Growth* method, and we state it without proof.

Property 2 (infrequent pruning). *If an itemset is infrequent due to regression, all related itemsets generated from it by IIG and RIG operations also could be deleted from the lattice for infrequency.*

5. Experimental evaluation

In this section, we report our performance study. We first give our experimental set-up and then our results in different datasets.

5.1. Experimental set-up

Our algorithm is written in C++ and compiled by Microsoft Visual Studio .NET 2003. All experiments are performed on Dell computer using a 2.8 GHz Intel PC, 512 MB of RAM, and 1350 MB of virtual memory. The operating system in use is the Microsoft Windows XP.

The scalability and executing time of our algorithm are evaluated by two synthetic datasets, T8.I4.N1000.D1000k and T5.I3.N1000.D1000k, generated by the IBM Quest Synthetic Data Generator that is available from: www.almaden.ibm.com/software/quest/Resources/datasets/syndata.html/#assoSynData. Some major parameters of IBM Data Generator are shown in Table 2. Default values for other parameters are used (i.e., number of patterns 10 000, correlation coefficient between pattern 0.25, and average confidence in a rule 0.75, etc.).

The data stream is broken into blocks of size 30 K transactions and fed into our program through standard input. The minimum support threshold θ is varied (as to be described below) and the maximal estimated possible error threshold ε is set to $0.1 \cdot \theta$.

5.2. Experimental results

We first performed two sets of experiments that examine the executing time and memory usage by *RFIMiner*. In the first set of experiments, θ is fixed at 0.005 and ε at 0.0005. In the other set of experiments, θ is fixed at 0.0075 and ε at 0.00075. In both sets of experiments, the fading factor ϕ is set to 0.75. Fig. 4 shows that the executing time smoothly and linearly grows and the memory usage changes slightly with the increasing number of transactions. Both of two experimental results indicate *RFIMiner*'s excellent scalability in either synthetic database.

Then, we also evaluate *RFIMiner*'s performance in mining recently frequent itemsets with three different fading factors: 0.75, 0.5, and 0.25. Fig. 5 shows the real-time numbers of recently frequent itemsets in the data stream simulated by the synthetic dataset, T5.I3.N1000.D1000k. In this experiment, the minimum support threshold θ is set to 0.1%, and the maximal estimated possible error threshold ε is set to $0.1 \cdot \theta = 0.01\%$. Results in Fig. 5 present that all three curves succeed in embodying the similar changing in the real-time number of recently frequent itemsets. So no matter what fading factors are selected, *RFIMiner* algorithm succeeds in reflecting the changing trend of frequent itemsets in the data stream. Moreover, we also can draw the conclusion that our algorithm with a big fading factor can monitor recently frequent itemsets in a longer time phase than a small factor.

The unique property among suffix-trees and efficient itemset growth method utilized by *RFIMiner* algorithm is the biggest characteristic different from other well-known algorithms. Consequently, we implement a new version of single-scan algorithm, *RFIMiner-II* that can mine frequent itemsets in datasets without

Table 2
Parameters for IBM data generator

Symbol	Options	Meaning
D	-ntrans	Number of transactions
T	-tlen	Average items per transaction
I	-patlen	Average length of maximal itemsets
N	-nitems	Number of different items

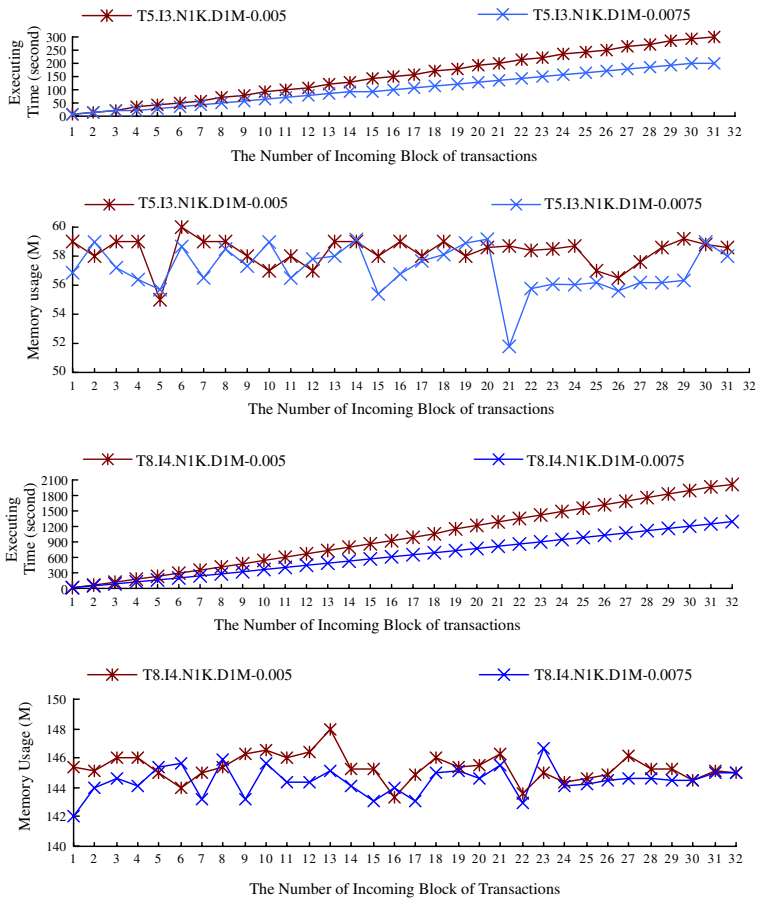

Fig. 4. The performance of *RFIMiner* algorithm in T5.I3.N1000.D1000k and T8.I4.N1000.D1000k.

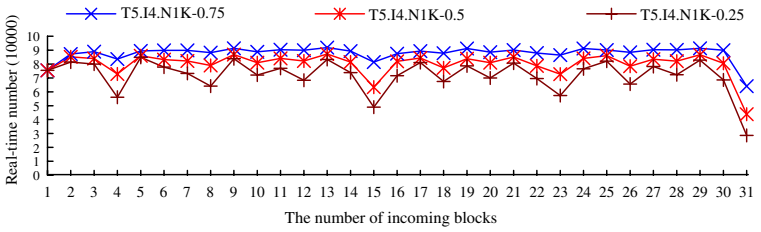

Fig. 5. Real-time number of recently frequent itemsets in different phases of dataset.

regression, to evaluate the practicality of suffix-forest and the efficiency of *Depth-first and Bottom–up Inside Itemset Growth* method. *RFIMiner-II* is compared with multi-scan *Apriori* algorithm and bin-scan *FP-growth* algorithm which are available from: http://www.cs.umb.edu/laur/ARtool/. The comparisons are conducted in the environment of a synthetic dataset, T5.I3.N1000.D.30k. From Fig. 6, we can draw the conclusion that *RFIMiner-II* algorithm outperforms both *Apriori* and *FP-growth* algorithms with factor 10 and 2, respectively.

According to the results in Fig. 6, there might be a question concerning the executing time of *RFIMiner-II* algorithm. *Why it hardly changes with the increasing of the minimum support threshold?* After an intensive thinking, we give out the answer as follow. The executing time actually is divided into two parts, including time used for constructing suffix-trees and time for mining frequent itemsets. However, compared with the time for suffix-trees, the latter is rather small. Although the time for mining frequent itemsets decreases with

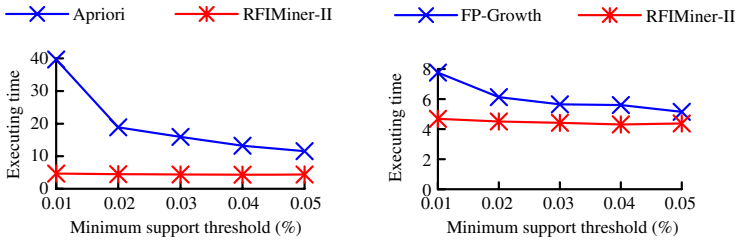

Fig. 6. Comparison of *RFIMiner-II* with *Apriori* and *FP-Growth* algorithms.

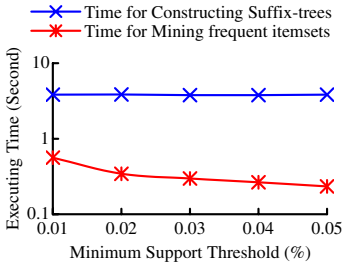

Fig. 7. Division of executing time of *RFIMiner-II* algorithm in T5.I3.N1000.D30k.

the increasing of minimum support threshold, the time for constructing suffix-trees is almost immutable and thus the entire executing time changes slightly with θ increasing. The analysis is testified by the experiment concerning the division of executing time of *RFIMiner-II* algorithm, and the result is shown in Fig. 7.

6. Conclusions

In this paper, we propose an approach to mine recently frequent itemsets in the data stream. First, we propose a new streaming data model where finer granularities are endowed to new transactions and coarser granularities are given to old transactions. We also stipulate a regression rule for lessen the influence of obsolete transactions on recently frequent itemsets. Moreover, suffix-trees proposed by us are employed by *RFIMiner* algorithm. By utilizing a particular property between suffix-trees, the connectivity of two nodes in a suffix-tree is indirectly but fast confirmed by judging whether corresponding identical nodes synchronously occur in two specific suffix-trees. To optimize *RFIMiner*'s performance, a new itemset growth method is presented as well. Experiment results show that *RFIMiner* algorithm is scalable to very long data streams and has an excellent ability to mine recently frequent itemsets in data streams.

Acknowledgements

This work was supported by the Natural Science Foundation of China (Grant No. 60433020) and the Key Science-Technology Project of the National Education Ministry of China (Grant No. 02090).

References

[1] R. Agrawal, T. Imielinske, A. Swami, Mining association rules between Sets of items in large databases, in: Proceedings of the ACM SIGMOD International Conference Management of Data, Washington, DC (1993) pp. 207–216.
[2] R. Agrawal, R. Srikant, Fast algorithm for mining association rules, in: Proceedings of 21st International Conference on Very Large Databases, Santiago, Chile (1994) pp. 487–499.
[3] R. Agrawal, R. Srikant, Mining sequential patterns, in: Proceedings of the International Conference on Data Engineering, Taipei, Taiwan (1995) pp. 3–14.
[4] R. Bayardo, Efficiently mining long patterns from databases, in: Proceedings of the ACM SIGMOD International Conference Management of Data, Seattle, WA (1998) pp. 85–93.

[5] S. Brin, et al., Beyond market basket: Generalizing association rules to correlations, in: Proceedings of the ACM SIGMOD International Conference Management of Data, Tucson, Arizona (1997) pp. 265–276.
[6] H. Mannila et al., Discovery of frequent episodes in event sequences, Data Mining and Knowledge Discovery 1 (1997) 259–289.
[7] B. Lent, et al., Clustering association rules, in: Proceedings of the International Conference on Data Engineering, Birmingham, England (1997) pp. 220–231.
[8] K. Wang, S. Zhou, J. Han, Profit Mining: From Patterns to Actions, in: Proceedings of International. Conference on Extending Data Base Technology, Prague, Czech (2002) pp. 70–87.
[9] J. Han, J. Pei, Y. Yin, Mining frequent patterns without candidate generation, in: Proceedings of the ACM SIGMOD International Conference Management of Data, Dallas, TX (2000) pp. 1–12.
[10] J. Pei, J. Han, H. Lu, et al., H-Mine: Hyper-structure mining of frequent patterns in large databases, in: Proceedings of the IEEE International Conference on Data Mining, San Jose, CA (2001) pp. 175–186.
[11] J. Liu, Y. Pan, K. Wang, J. Han, Mining freuquent itemsets by opportunistic projection, in: Proceedings of SIGKDD'02. KDD (2002) pp. 229–238.
[12] N. Pasquier, Y. Bastide, R. Taouil, L. Lakhal, Discovering frequent closed itemsets for association rules, in: Proceedings of International Conference on Database Theory, Jerusalem, Israel (1999) pp. 398–416.
[13] J. Pei, J. Han, R. Mao, CLOSET: An efficient algorithm for mining frequent closed itemsets, in: Proceedings of the ACM SIGMOD International Workshop Data Mining and Knowledge Discovery, Dallas, TX (2000) pp. 11–20.
[14] M. Zaki, C. Hsiao, CHARM: An efficient algorithm for closed itemset mining, in: Proceedings of SIAM Conference on Data Mining, Arlington, VA (2002) pp. 457–473.
[15] D. Burdick, M. Calimlim, J. Gehrke, MAFIA: A maximal frequent itemset algorithm for transactional databases, in: Proceedings of the International Conference on Data Engineering, Heidelberg, Germany (2001) pp. 443–452.
[16] G.S. Manku, R. Motwani, Approximate Frequency Counts Over Data Streams, in: Proceedings of the International Conference on Very Large Data Bases, Hong Kong, China (2002) pp. 346–357.
[17] C. Giannella, J. Han, J. Pei, X. Yan, P.S. Yu, Mining frequent patterns in data streams at multiple time granularities, Next Generation Data Mining Chapter 3 (2002) 191–211.
[18] R. Jin, G. Agrawal. An Algorithm for In-Core Frequent Itemset Mining on Streaming Data, [online] Available from: http://www.cse.ohio-state.edu/agrawal/ (2004).
[19] W.G. Teng, M.S. Chen, P.S. Yu, A Regression-Based Temporal Pattern Mining Scheme for Data Streams, in: Proceedings of the 29th VLDB Conference (2003) pp. 93–104.
[20] J. Chang, W. Lee, Finding Recent Frequent Itemsets Adaptively over Online Data Streams, in: Proceedings of the ACM International Conference on Knowledge Discovery and Data Mining, Washington, DC (2003) pp. 487–492.
[21] J. Chang, W. Lee, A sliding window method for finding recently frequent itemsets over online data streams, Journal of Information Science and Engineering (2004) 753–762.
[22] D. Xin, J.W. Han, X.F. Yan, H. Cheng, Mining Compressed Frequent Pattern Sets, in: Proceedings of the International Conference on Very Large Data Bases, Trondheim, Norway, (2005).