Available online at www.sciencedirect.com
ScienceDirect
Information Systems 32 (2007) 649–669
www.elsevier.com/locate/infosys

Mining association rules in very large clustered domains

Alexandros Nanopoulos*, Apostolos N. Papadopoulos, Yannis Manolopoulos

Department of Informatics, Aristotle University, 54124, Thessaloniki, Greece

Received 15 June 2005; received in revised form 20 March 2006; accepted 16 April 2006
Recommended by N. Koudas

Abstract

Emerging applications introduce the requirement for novel association-rule mining algorithms that will be scalable not only with respect to the number of records (number of rows) but also with respect to the domain's size (number of columns). In this paper, we focus on the cases where the items of a large domain correlate with each other in a way that *small worlds* are formed, that is, the domain is clustered into groups with a large number of intra-group and a small number of inter-group correlations. This property appears in several real-world cases, e.g., in bioinformatics, e-commerce applications, and bibliographic analysis, and can help to significantly prune the search space so as to perform efficient association-rule mining. We develop an algorithm that partitions the domain of items according to their correlations and we describe a mining algorithm that carefully combines partitions to improve the efficiency. Our experiments show the superiority of the proposed method against existing algorithms, and that it overcomes the problems (e.g., increase in CPU cost and possible I/O thrashing) caused by existing algorithms due to the combination of a large domain and a large number of records.

Keywords: Association rules; Clustering; Item domain grouping; Data mining

1. Introduction

The mining of association rules involves the discovery of significant and valid correlations among items that belong to a particular domain [1]. The development of association-rule mining algorithms has attracted remarkable attention during the last years [2], where focus has been placed on efficiency and scalability issues with respect to the number of records[1] [3]. However, emerging applications, e.g., microarray data analysis, e-commerce, citation analysis, introduce additional scalability requirements with respect to the domain's size, which may range from several tenths to hundreds of thousands of items.

For the aforementioned kind of applications, in several cases it can be expected that items will correlate with each other (i.e., create patterns) in a way that *small worlds* [4] are formed. In such cases, the very large domain will be partitioned into smaller groups consisting of items with significant correlations between them, whereas there will be few

*Corresponding author. Tel.: +30 231091924.
E-mail addresses: alex@delab.csd.auth.gr (A. Nanopoulos), apostol@delab.csd.auth.gr (A.N. Papadopoulos), yannis@delab.csd.auth.gr (Y. Manolopoulos).

[1]The type of record depends on the application. For the example of market-basket data, records correspond to transactions. Henceforth, the terms record and transaction are used interchangeably.

0306-4379/$ - see front matter © 2006 Elsevier B.V. All rights reserved.
doi:10.1016/j.is.2006.04.002

random correlations between items of different groups. Small worlds usually appear due to a kind of clustering that is inherent in the examined phenomena. For instance, microarray data contain gene networks that consist of genes which express similar behavior, users of e-commerce sites are grouped into communities showing similar preferences, or, in citation databases, authors are separated into different scientific disciplines. In all these examples, despite the very large domain's size, items do not correlate with each other uniformly and they tend to form groups with a large number of intra-group and a small number of inter-group correlations. This property can help in (i) significantly pruning the search space and (ii) performing efficient association-rule mining.

1.1. Motivation

The issue of domain's cardinality, mainly motivated by biological applications, has been considered recently for the problem of mining closed patterns [5,6], which introduced the concept of *row enumeration*. Nevertheless, to the best of our knowledge, the problem of mining (regular) association rules from databases having both a very large number of items and records, with the ability to prune the search space based on the previously described property, has not been considered so far. Besides, the algorithms in [5,6] are main-memory based, whereas we focus on disk-resident databases.

On the other hand, existing association-rule mining algorithms based on column enumeration can be severely impacted by a large domain. BFS algorithms (Apriori-like) will produce an excessive number of candidates, which drastically increase the CPU and I/O costs (when the candidates do not fit in main memory, they have to be divided into chunks and numerous passes are performed for each chunk [7]). DFS algorithms (e.g., FP-growth [8] and Eclat [9]) use auxiliary data structures that summarize the database (e.g., FP-tree), which become less condensed when the domain size increases, because of the many different items' combinations. This affects the CPU and I/O costs and, more importantly, disk thrashing may occur when the size of the structures exceeds the available main memory. With *database projection* [8] disk thrashing is avoided, but for large domains I/O cost is still high, due to the large amount of projections that have to be examined (one for each frequent item). The aforementioned factors are amplified by the need of using relatively low support thresholds, since in large domains the patterns exist locally, i.e., within each group, and not globally.

In conclusion, domains with very large size pose the following requirements to association-rule mining algorithms:

- To control the increase in CPU cost, which is incurred by the large number of possible items' combinations.
- To avoid disk thrashing caused by the lack of main memory (needed for candidate sets or for auxiliary structures). Also, to control the increase in I/O cost, which is incurred by approaches such as database projection.
- To handle disk-resident data (i.e., databases that cannot be entirely kept in main memory). This yields, in contrast to row enumeration approaches, to the consideration of domains whose size is very large, but relatively not much larger compared to the number of records. For instance, we would like an algorithm able to mine association rules from a database containing million rows and hundreds of thousands of columns.
- To be able, in the above cases, to use relatively low support thresholds.

1.2. Contribution

In this paper, we focus on mining databases with a very large number of records (disk-resident) and with a domain that has a very large number of items whose correlations form a small-world kind of grouping. Initially we propose a partitioning method to detect the groups within the domain. This method acts in an initial phase and detects, very quickly and with low memory consumption, the groups of items. Then, in a second phase, we propose the separate mining of association rules within groups. During this phase, the merging of groups is possible. The separate mining of each partition is performed by (i) focusing each time on the relevant items and (ii) pruning them later on to reduce execution time. The technical contributions of our work are summarized as follows:

- The proposed method has the advantage of scalability to very large domains (up to a million of items are being considered), since the partitioning approach divides, and thus reduces, the complexity of the original problem to the

manageable sub-problem of mining association rules in much smaller sub-domains.

- The scalability with respect to the number of records is also maintained, since no assumption is made for storing the data set in main memory and, moreover, the size of the corresponding data structures is not excessive (as in existing approaches). The problem of thrashing due to memory shortage is avoided.
- The proposed method does not depend on the algorithm used for mining within each partition, a fact which increases its flexibility and its incorporation in existing frameworks. This is analogous to the approach of [10], which proposed the partitioning of the records and used an existing algorithm (AprioriTid) for the mining within each partition. However, the raised issues and the required solution in our case are significantly different than those in [10].
- Extensive experimental results are given, for both synthetic and real data, which show that the proposed approach leads to substantial performance improvements (up to one order of magnitude), in comparison to existing algorithms.
- The discussion of issues related to applications in which the domain may not be partitioned into well-separated groups. We describe some hints on this problem and address interesting points of future work.

The rest of the paper is organized as follows. Section 2 overviews related work in the area. In Section 3 we describe our insight on the presence of groups within domains, which are determined by correlations between items. The method for partitioning the domain is presented in Section 4, whereas the algorithm for mining the partitions is given in Section 5. Performance results are presented in Section 6. Section 7 considers the case of partitions that are not well separated. Finally, Section 8 draws the conclusions and provides directions of future work.

2. Related work

Since its introduction [1], the problem of association-rule mining has been studied thoroughly [11]. Earlier approaches are characterized as Apriori-like, because they constitute improvements over the Apriori algorithm [12]. They explore the search space in a BFS manner, resulting in an enormous cost when patterns are long and/or when the domain size is large. Techniques that reduce the number of candidates [13] may improve the situation in the latter case, but are unable to handle a very large domain (see Section 6.3). Algorithms for mining generalized association rules [14] are also Apriori-like. They do not operate on individual items but they focus on categories at different levels, by assuming that the items are categorized hierarchically. Therefore, a large domain can be replaced by a much smaller number of categories. Nevertheless, this approach requires the existence of a predetermined hierarchy and the knowledge of the items that belong in each category. In contrast, we focus on a kind of grouping that is determined by the in-between correlations of items, which is not predefined and not hierarchical (see Section 3).

A different paradigm consists of algorithms that operate in a DFS manner. This category includes algorithms like Tree Projection [15], Eclat [9], and FP-growth [8], whereas extensions have been developed for mining maximal patterns (e.g., MAFIA [16]). Eclat uses a vertical representation of the database, called *covers*, which allows for efficient support counting through intersections of item's covers. For large domains, this may lead to a significant overhead, since a large number of intersections have to be computed. FP-growth uses a prefix tree, called FP-tree, as a condense representation of the database and performs mining with a recursive procedure over it. Both Eclat and FP-growth require that their representations have to fit entirely in main memory. A study of their main-memory requirements is included in [17]. For large domains, the aforementioned requirement may not always hold. To overcome this problem the technique of database projection has been proposed in [8]. Nevertheless, as will be shown, for very large domains this technique results in high execution times, since projection is performed for each item. A study of the relation between available main memory and association-rule mining is described in [18], but for the different context of dynamic changes in available memory.

All the previous methods are based on column enumeration (also denoted as feature enumeration). For mining closed patterns over databases with much more columns than rows, e.g., gene-expression data, row enumeration has been proposed. CARPENTER [5] is such an algorithm, which finds closed patterns by testing combination of rows. CARPENTER works on the basis that its entire

database representation is held in main memory, which holds for relatively small data sets like microarray data. As the number of rows increases, pure row enumeration may become inefficient. COBBLER [6] proposes the dynamic switching between column and row enumeration and compares favorably against other closed-pattern mining algorithms like CHARM [19] and CLOSET+ [20]. Similar to CARPENTER, COBBLER runs in main memory (it uses conditional pointer lists and performs only one scan of the database to load it into memory [6]). This assumption may be constraining for very large databases, especially in the context of closely coupling data mining with a DBMS [21], where memory is provided by the DBMS and is shared between several mining tasks and OLTP transactions. As described, differently from the aforementioned works, our approach focuses on regular association rules and not closed patterns. Moreover, we are interested in databases that are disk-resident and contain a very large number of columns and a very large number of rows as well (the rows may be more than the columns). Finally, our method prunes the search space by considering groups that are formed by correlations between items. Nevertheless, since closed patterns are a specialization of association rules, in the future we plan to examine the application of the proposed method for the mining of closed patterns.

Other algorithms that are influenced by the characteristics of gene-expression data include FARMER algorithm [22], which mines a particular type of closed patterns from microarray data, and BSC- and FIS-trees [23], which mine regular association rules from gene-expression data. Although [23] shows an improvement over FP-growth, it is limiting because it runs in main memory and only considers data sets with few hundred transactions.

Finally, the concept of finding localized patterns is presented in [24]. This work inspired us to consider that patterns may not always exist globally. Instead, in several applications the data set may contain clusters that provide their own (local) patterns. Nevertheless, [24] is based on the different direction of clustering the transactions, whereas we are interested in clustering the domain. For this reason, [24] does not pay attention to the problems resulting from a large domain. Other methods that partition the database of transactions are described in [10,25].

3. Correlation-based groups of items

According to the notion of association rules, correlation between items is determined by their co-occurrence frequency. Consider a relatively small domain. Provided that patterns exist, the co-occurrence probability between items is not uniform. Items co-occur more frequently along with those items with which they form patterns and less frequently with others. Nevertheless, since the domain is small, different patterns are expected to share many common items. For instance, let two patterns be $p_1 = \{A, C, D\}$ and $p_2 = \{A, B, C\}$, which share items A and C. We stipulate that items A, B, C, and D belong to the same group, albeit p_1 and p_2 do not both contain items B and D. This grouping of items is defined in the sense that, either directly (A and C) or obliquely (B and D), they comprise a set of items among which there exist patterns, i.e., correlations. Accordingly, in small domains very few groups (probably only a single one) will be formed, since items will be shared by many patterns.

Consider the antipodal case, where the domain is very large. Assuming that a sufficient fraction of domain's items participate into patterns, the items that are shared between patterns are expected to be much less compared to the case of a small domain. This is because of the many more different items' combinations. Therefore, the probability of having an item shared by many patterns is now smaller. Although the latter holds at a global level, at a local level it is likely to find such a sharing of items by sub-collections of patterns. Hence, items will tend to form several isolated groups, which will contribute their own patterns, whereas there will be few random correlations between items belonging to different groups. The reasons for the formation of such groups, as already explained, lie on inherent properties of the examined data which produce the small-world effect. As mentioned, the aforementioned behavior can be observed in several real-world cases, e.g., in microarray data (genes are organized into gene networks) or in citation analysis (papers belong to research communities), etc.

It is useful to ponder the following visualization, having the items of a domain comprise the vertex set of a graph. An edge exists between two vertices i and j if the corresponding 2-itemset (i,j) is frequent. Using the generator of [7], we produced two synthetic data sets, with 1000 and 100,000 items, respectively. Both data sets contain 100,000

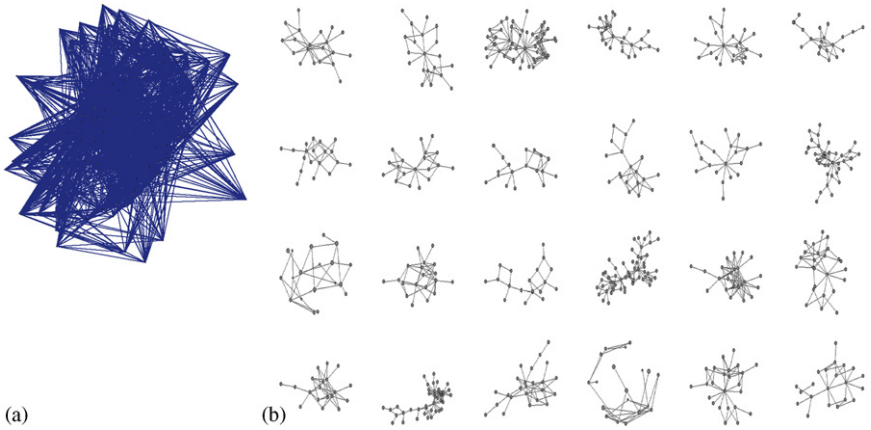

Fig. 1. Resulting graph for 0.1% support threshold: (a) domain size 1000, (b) domain size 100,000.

transactions, the average size of each transaction is 10, the average pattern size is 4, and the support threshold is equal to 0.1% (following the terminology of [7], these data sets are denoted as T10I4D100K). Fig. 1a illustrates the resulting graph for the first data set, whereas, to ease presentation, Fig. 1b a sample of the second one. The former graph consists of a single group plus some few isolated frequent items (which are not depicted for clarity). In contrast, the latter graph consists of several separated groups.

Evidently, the characteristics of the formed groups (e.g., size, shape) depend on the nature of the application and the properties of the data. Regarding the inter-group correlations between items of different groups, if the co-occurrence frequency between these items is insignificant (with respect to the user-defined support threshold), then the items are separable into different groups (as in the example of Fig. 1b). Therefore, the fact that inter-group correlations are few and random can prevent the obstruction of the items' grouping. Our experiments show several cases where the detection of such separate groups is possible. However, in Section 7 we also consider the detection of groups even when, initially, it is not possible to find a clear way to separate the items due to the presence of significant inter-group correlations.

4. Partitioning the domain

By partitioning a large domain we divide the items into disconnected groups, so that it becomes possible to find correlations only between items of the same group. Thus, the original association-rule mining task is divided into each group and becomes manageable. In the rest of this section we describe in detail the proposed method for partitioning the domain. In the following section we describe the mining of the partitions. Table 1 summarizes the most important symbols used throughout the study.

Table 1
Symbols and definitions

Symbol	Definition
n	Number of unique items (domain size)
N	Number of transactions
T_1, T_2	Transactions
S	Average transaction size
i, j	Items
(i, j)	2-itemset and edge
sup	Min. support threshold
G	Graph induced by 2-itemsets
H	Size of hash-table
C, C_i	Components
L_p	Min. partition size
L	List with small partitions
B	List with candidate bridges

4.1. A direct approach

The procedure to partition the domain is based on the determination of graph connected components. A direct approach for this procedure is detailed as follows. Let *sup* be the given support threshold. Based on the Apriori-pruning criterion [12], no itemset can have support higher than *sup* unless all of its sub-sets have also support higher

than *sup*. Therefore, a direct approach to partition the domain of items can be described as with the following operations:

1. Form an undirected graph G. The vertex set of G consists of all items with support higher than *sup* and the edge set of all candidate 2-itemsets.
2. Find the support of each candidate 2-itemset.
3. Keep the edges of G that correspond to a 2-itemset with support higher than *sup* and delete all the others.
4. Find all the connected components of G. Each component corresponds to a group of the domain.

Due to Apriori criterion, it is impossible to have an itemset with support higher than *sup* that contains items from two components of the graph, since there is no common edge (i.e., sub-set of length two) between any two components. One could extend this approach by considering a graph with edges determined by sub-sets of larger lengths (e.g., 3-itemsets, etc). This extension is hardly viable, because it will result to a partitioning phase with prohibitive cost.

Prior work [26,27] has used graphs to represent the supports of the 2-itemsets. However, these works follow a simplistic implementation of the graph. In particular, they first compute the supports of *all* candidate 2-itemsets, before they filter out the infrequent ones and keep the frequent to represent the edges of the graph. This simplistic approach is possible only for domains with small size (up to few thousand items). In contrast, for very large domains, this approach is not feasible, because the graph cannot be maintained in main memory during its building. If we want to overcome the latter problem by simply resorting to virtual memory, we will result with a severe overhead in execution time. The reason is that the graph is built dynamically, by reading records from the database and focusing on the pairs of items within them. In general, the records are stored on disk in no particular order. Consequently, scattered I/O (thrashing) will occur while accessing the edges in order to update their support.

Proposition 1. *The worst-case complexity of the direct approach is* $O(N \cdot S^2 + n^2)$, *where* N *is the number of transactions*, S *the average transaction size*, *and* n *the total number of items*.

Proof. The direct approach requires one pass of all transactions to determine the support of 1-itemsets, requiring a complexity of $O(N \cdot S)$. The graph construction requires a complexity of $O(n^2)$ to determine the edge set of 2-itemsets. The determination of the support of all 2-itemsets requires a complexity of $O(N \cdot S^2)$, since all pairs of items within each transaction should be considered. Next, all pairs of items should be checked to determine which 2-itemsets have a support greater than or equal to *sup*. This step requires a complexity of $O(n^2)$. Finally, determining the connected components of the graph requires a complexity of $O(n^2)$ in the worst case, since all edges must be checked. In conclusion, by considering the previous observations, the complexity of the direct approach is $O(N \cdot S^2 + n^2)$. □

To avoid the aforementioned inadequacies, we observe that we do not need to follow the direct approach to partition the domain into groups. That is, we should not find the support of all candidate 2-itemsets. What we only need to find is the support of exactly those edges that will allow for the detection of groups. This way, the graph will contain much fewer edges, it will fit in main memory, and much smaller processing cost will be entailed. This observation is applied through some steps we propose, which are described in the sequel.

4.2. Proposed partitioning method

In this section, we describe the proposed partitioning method, that has the objective of identifying disjoint components in the graph. Therefore, the proposed method identifies partitions in the case that they are disjoint. We assume that an initial scanning of the database has been performed and the support of all 1-itemsets has been counted (this phase is common in all association-rule mining algorithms). The vertex set of the graph consists of all items with support higher than *sup*.[2] Next, one extra scan is performed, during which the necessary edges are determined in order to detect the groups. Thus, two scans are required in total. In the following we first elaborate on the graph structure that is used by our algorithm, and then we describe the partitioning algorithm.

[2]In the following, the terms vertex and item are used interchangeably.

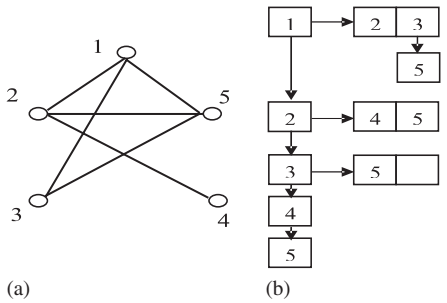

Fig. 2. Example of linked-list representation.

4.2.1. Graph structure

The graph is stored in the form of adjacency lists. For each vertex i we will store a list of the vertices j for which a 2-itemset (i,j) is considered (see Section 4.2.2) during the second database scanning. Along with an edge we also store the support of the corresponding 2-itemset. Initially, all lists are empty and the supports are updated during the scanning. This way we do not statically preallocate space for each possible edge, since not all of them will be considered, and thus a significant amount of space is saved. To reduce the cost of searching vertices within the adjacency lists, the latter are implemented as hash-tables with chains. An example of the graph structure is depicted in Fig. 2a whereas Fig. 2b illustrates its representation with adjacency lists (support values are not shown). For illustration purposes the hash-tables have two positions (in general, a much larger value is used) and a simple hashing scheme is used, where even items are hashed in the first position and odd items in the second. When searching, e.g., for item 4 in the list of 2, we only examine the first position. Collisions cause the formation of chains (e.g., the chain between 3 and 5 in the list of 1). When searching for item 5 in the list of 1, the whole chain (items 3 and 5) is searched. Hash-tables may not be fully utilized (e.g., the list of 3 has one empty position). For selecting the size H of hash-tables, there is a tradeoff between space cost (due to under-utilization when H is large) and reduced searching speed (due to searching in chains when H is small). A good estimate for H is to set it equal to the average number of items in each partition. The reason is that, in order to avoid a large number of collisions, for each item i, we have to allocate a hash-table with size equal to the expected number of items in the partition that i belongs to.[3]

4.2.2. Partitioning algorithm

During the second scan, the necessary edges are determined, in order to detect the groups. The outline of the domain partitioning algorithm is depicted in Fig. 3.

Two vertices belong to the same component, if there is a connecting path formed by edges with supports equal or higher than *sup*. For the edge (i,j), DomainPartitioning examines the following conditions:

(a) If i and j belong to the same component, there is no need to examine pair (i,j) (the corresponding edge is omitted).
(b) If i and j belong to different components of the graph, we distinguish two additional conditions according to the support of edge (i,j):
  (b1) If the support is larger than zero but lower than $sup-1$, it is increased by one. In case it is zero, we allocate a new edge by inserting j in i's adjacency list with support one.
  (b2) If the support is equal to $sup-1$ (about to become frequent), we perform the operations defined by the EarlyMerging process to identify a new component. EarlyMerging is detailed in the following.

The edges that are omitted in the first (a) case are those that do not yield to new components. An example is given in Fig. 4, which depicts two components C and C'. The edges with support equal to *sup* are plotted with solid lines, whereas those with less than *sup* with dotted line. When pair (i,j) is encountered, we omit the allocation of an edge between them, since they both belong to C (there is a path between them formed by edges with solid lines). To facilitate the testing whether two vertices belong to the same component, we represent components with a data structure for disjoint sets. Since the *find* operations are more frequent than the *union* ones (i.e., merging of components), we used a Quick-Find structure instead of Quick-Union [28].

An allocation of an unnecessary edge may not be always avoided. In the example of Fig. 4, for the

[3]Since the average partition size may not be known during the building of the graph, the estimation can be drawn from a sample of the original transactions.

```
Algorithm DomainPartitioning(float sup, Table db)
begin
    initialize the graph structure
    foreach trasaction T ∈ db
        foreach pair of items (i, j) in T
            if i and j belong to the same component
                do nothing
            else if i and j belong to different components
                if support of (i, j) is zero
                    allocate a new edge in the graph structure
                else if support of (i, j) is less than sup − 1
                    increase support of (i, j) by one in the graph structure
                else if support of (i, j) equals sup − 1
                    call EarlyMerging
        end for
    end for
end
```

Fig. 3. The proposed partitioning algorithm.

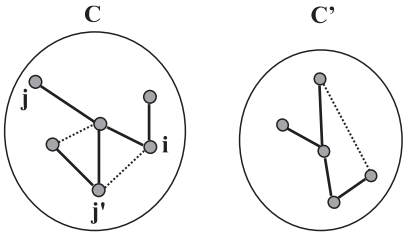

Fig. 4. Example of the examined cases.

pair (i,j') an edge has already been allocated. This happened before some of the solid lines, which connect them, have reached support *sup*. Nevertheless, in this case we avoid the additional CPU cost to increase the edge's support. This is explained as follows. The operation of testing if two items belong to the same component is performed fast through the disjoint-set data structure, not by examining the path between them. In contrast, the increase of the support is performed through the adjacency-list structure. The latter operation is more costly than the former, due to the existence of chains in the hash-tables, whereas the disjoint-set structure is optimized for the *find* operation. Moreover, the testing if the items belong to the same pair has to be performed in any case, because the items may belong to different components (the case when an edge allocation will be required). In conclusion, by omitting unnecessary edges we save processing cost and/or space.

Early merging: Two components C and C' have to be merged when there exist edges of the form (i,j), $i \in C$, $j \in C'$, with support values greater than or equal to *sup*. We can, therefore, merge C and C' upon the first time we find that the support of an edge (i,j), $i \in C$, $j \in C'$, is about to reach the value of *sup*. This corresponds to b2 case in DomainPartitioning algorithm. The merging of C and C' is done by joining their representations in the disjoint-set structure. By merging components as early as possible, we save processing cost, since we avoid any further examination of their in-between vertices. Notice that this way the support of an edge cannot reach a value higher than *sup*.[4] Therefore, we do not consider all edges, whereas for those considered we do not find their actual support. All we find is the necessary information to identify the components. Besides processing cost, we may save space cost as well, if during the merging we prune all other edges, except (i,j), between C and C'. This is because the support of these edges is (currently) lower than *sup* and will not further change. An example is given in Fig. 5, which illustrates two components C and C' that are about to be merged, because the support of edge (i,j) has just become equal to *sup*. Solid and dotted lines have the same meaning as in the example of Fig. 4. Dash-dotted lines between vertices of C and C' represent the edges that have not reached the support threshold and can be pruned. (Actually, dashed lines within C and C'

[4]This is the reason we did not consider another case (say, b3).

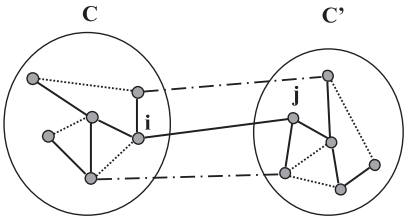

Fig. 5. Example of early merging.

represent such edges, which could have been pruned.) We have to note that the pruning of such edges should be optional, i.e., when main memory does not suffice, since it requires some additional cost.

Proposition 2. *The worst-case complexity of the proposed partitioning scheme is* $O(N \cdot S^2 + n^2)$, *where N is the number of transactions, S the average transaction size, and n the total number of items.*

Proof. The proposed approach requires one pass of all transactions to determine the support of 1-itemsets, requiring a complexity of $O(N \cdot S)$. During the subsequent scan performed by the proposed method, each pair of items should be checked once to determine if the edge should be omitted (Step 1) or two components should be merged (Step 2). This requires a complexity of $O(n^2)$ in the worst case to perform all necessary merges and $O(N \cdot S^2)$ to check every 2-itemset in every transaction. Therefore, the resulting complexity is $O(N \cdot S^2 + n^2)$. □

Although the worst-case complexity given by Propositions 1 and 2 are the same, the proposed partitioning algorithm has the advantage that it does not require the static allocation of the graph structure. Thus, it performs favorably against the direct approach that is presented in Section 4.2.1. This is verified by our experimental results in Section 6.4.

Finally, we have to note that the existence of disconnected partitions depends on the intrinsic mechanism in the data that generates these partitions, e.g., the authors communities, gene expressions, etc. There are several factors that affect the number of disconnected partitions. We will study their impact in Section 6.5. Nevertheless, such factors, like the number of items or the probability distribution they follow, cannot directly determine the number of resulting partitions in a way that can be expressed by a close-formula. The reason is that they affect the number of partitions only through their interaction with the mechanism that generates them and not by themselves. For example, for the same number of items, entirely different number of partitions may be resulted, if the mechanism of their generation changes.[5] In contrast, a useful tool to estimate the number of partitions is sampling. More precisely, we can generate a (uniform) sample with respect to the transactions. As the sample contains a small fraction of the original transactions, we can apply the partitioning procedure over it with only a small time overhead. Therefore, an estimation for the number of clusters can be drawn fast. Sampling has the advantage that it works regardless from the underlying factors, thus it is a general tool and does not require any assumptions about specific distributions. In Section 6.5 we also evaluate the effectiveness of sampling for the aforementioned purpose.

5. Mining the partitions

For the set of items in each partition we have to find the corresponding itemsets with frequency higher than *sup*. A straightforward approach would be to apply an algorithm for frequent-itemset mining (FIM) separately for each partition. When the FIM algorithm is applied on a partition, it performs a scan of the database during which it takes into account within each transaction only the items that belong to this partition.

The FIM algorithm is applied on partitions and does not face the problem of large domain size. For this reason, existing FIM algorithms can be used. Since FP-growth is one of the best FIM algorithms, as long as the number of items is manageable, and for having a fair comparison with it, we henceforth employ it for this task.[6] Nevertheless, the performance of this straightforward approach can be severely impacted by the inconsideration of two facts:

- The length of several partitions may not be adequately large to justify the cost of separate application of an FIM algorithm.

[5]Consider also the analogous case of, say, two-dimensional point data that form clusters. In this case, the number of clusters does not directly depend on the number of points or their distribution within the clusters. For this reason, the number of clusters that will be detected by clustering algorithms cannot be determined a priori.

[6]As also explained in Section 1.2, any other algorithm can be used as well.

- In each transaction, the items that participate during the application of FIM algorithm will no longer be needed for subsequent applications.

In order to overcome these problems, we propose the use of two techniques, denoted as *partition-merging* and *transaction-pruning*, which are described as follows.

Partition-merging can combine several partitions for a single application of the FIM algorithm. Each time an FIM algorithm is applied, a cost is involved (e.g., FP-growth requires one scan of the database, the building of the FP-tree on the items of the partition, and the searching for frequent itemsets in it). It is pointless to pay this cost just to find few frequent itemsets from a small partition. The control of justifiable partition size is achieved through a cut-off value, called L_p. The partitions are examined one by one. Each time a partition with less than L_p items is encountered, its items are appended to a list L (initially empty). The FIM algorithm is applied only in two cases: (i) when for the size $|P|$ of the currently examined partition P it holds that $|P| \geqslant L_p$, or (ii) when $|L| \geqslant L_p$. That is, partitions that are large enough are mined separately; otherwise, when the number of items belonging to small partitions is large enough, then these partitions are mined together. After a combined application of FIM on the items of L, the latter becomes empty.

During each application of FIM algorithm, transactions are read from the data set one by one. In each transaction, only the items that belong to the currently examined partition or partitions (in case of partition-merging) are considered. By using transaction-trimming, we can discard from the transaction all the considered items, since they will contribute nothing to subsequent applications of FIM algorithm. The result is the reduction of I/O cost during these subsequent applications, because smaller transactions will be read. Transaction-trimming is a technique that has been proposed in prior works, e.g., [13], and we adopt it here, because it helps in reducing the overall execution time.[7] Notice that transaction-trimming can easily be incorporated to any FIM algorithm, since it does not change the logic that the latter is based upon.

The resulting mining scheme is given in Fig. 6. The special case where a partition contains only one item is treated separately, since we can immediately report a singleton itemset that has support higher than *sup*. In the end, after having examined all partitions, the list L may not be empty. In this case, regardless of its size, we have to apply the FIM algorithm on its remaining contents. As explained earlier, we choose the FP-growth algorithm for the implementation of FIM. However, we applied an optimization: FP-growth requires two scans of the database *db*, one to find the support of items (singleton itemsets) and resort the items in each transaction of the *db* with respect to their support, and then one to build the FP-tree. In our implementation, the former task is performed only once, during an initialization step (second step of the algorithm MinePartitions), to avoid its repeated execution during the applications of FIM. The reason is that the initial sorting of the database is valid for all partitions and does not have to be repeated.

Example 1. Assume a domain $\mathscr{D} = \{A, B, C, D, E, F, G, H, I, J\}$ and the database depicted in Fig. 7a. Let *sup* be equal to two and L_p equal to three. It is easy to verify that four partitions can be identified: $P_1 = \{A, B\}$, $P_2 = \{C, D\}$, $P_3 = \{E, F, G\}$ (because itemsets (E, F) and (E, G) are frequent), $P_4 = \{H\}$, whereas items I and J are not frequent. MinePartitions commences with P_1, which has size less than L_p, thus items A and B are inserted in list L. P_2 also has size less than L_p, thus items C and D are inserted in L. Now L has size four, which is larger than L_p. FIM is applied for the elements of L. This identifies items A, B, C, D (with supports 2, 2, 2, 3, respectively) and itemsets AB, CD (both with support 2) as frequent. Due to transaction-trimming the database now becomes equal to the one depicted in Fig. 7b.[8] Next,L is emptied and P_3 is considered. Since its size is larger than L_p, FIM is applied. It identifies items E, F, G (with supports 3, 2, 3, respectively) and itemsets EF, EG (both with support 2) as frequent. Now the database becomes equal to the one depicted in Fig. 7c. P_4 contains a single item, whose support is known (from the initial scan before the invocation of MinePartitions). Therefore,

[7] Transaction-trimming needs to write back a modified transaction (not all transactions are modified). This can be done using a double-buffering technique [29], which overlaps the I/O time to write the transaction with the CPU cost of processing the next one (e.g., in FP-growth, the items of a transaction cause the insertion of several paths into the FP-tree, which takes significant fraction of CPU time).

[8] For concreteness, the infrequent items I and J are still depicted. However, they can be easily identified and removed from transactions that are being trimmed.

```
Algorithm MinePartitions(Set Of Partitions P, int L_p, float sup, Table db)
begin
    L = ∅
    initialize FIM
    for each P ∈ P
        if |P| == 1
            report the singleton itemset
        else if |P| < L_p
            L = L ∪ P
            if |L| ≥ L_p
                call FIM(L, sup, db)
                L = ∅
        else
            call FIM(P, sup, db)
    end for
    if L ≠ ∅
        call FIM(L, sup, db)
end
```

Fig. 6. The algorithm that mines the partitions.

T1	ABEF
T2	CDG
T3	ABDI
T4	CDH
T5	EFG
T6	EGHJ

(a)

T1	EF
T2	G
T3	I
T4	H
T5	EFG
T6	EGHJ

(b)

T1	
T2	
T3	I
T4	H
T5	
T6	HJ

(c)

Fig. 7. Example of a database for the application of MinePartitions algorithm.

H (with support 2) is reported as frequent. Since no more partitions exist and L is empty, the algorithm terminates.

Regarding the selection of L_p, we have to consider the following. A very small L_p causes the creation of many small partitions. Therefore, FIM will be applied many times and will require many database scans. In contrast, a very large L_p causes the creation of very few, large partitions. Although FIM will be applied few times, each application will be impeded by the large number of the items that are taken into account (analogously to the problem produced in the case of a single large domain). Thus, L_p represents a tradeoff between the number of times that FIM is applied and the cost for each application. A natural choice for L_p is to set it equal to the average number of items in the partitions, as we want L_p to be tuned according to the sizes of the resulting partitions in the data set. In our experimental results we examine the impact of L_p and its analytical estimation.

By carefully tuning L_p, the number of FIM's invocations is restricted to only a few times. In each one, as explained previously, the database is scanned only once. Moreover, due to the transaction-trimming technique, each scanning is presented with a significantly reduced database. As a result, the I/O overhead is limited and the problems caused by the large domain are overcome (see Section 6). It is worthwhile to contrast this with an approach like FP-growth, that opts for only one extra scan (besides the initial one) and confronts significant difficulties due to the domain's size. This case produces the same problems with a naive selection of a very large L_p. To the other end, an approach like database projection [8] (using the *partition projection* method) for the case of large domains can result in a large number of scans. This causes similar problems as when naively selecting a very low L_p. Therefore, the proposed approach presents a middle ground between the two extremes, i.e., it pays off to perform only some very few additional scans (with reduced database size each time), to resolve the difficulties caused by the large domain.

Finally, another optimization that could be considered is, after the detection of partitions, to cluster the transactions with respect to the items they contain. If transactions contain only items from a single partition (and possibly some infrequent items and/or frequent singletons), then during the invocation of MinePartitions each application of FIM could be performed only to the corresponding cluster of transactions. Thus, a lot of I/O cost

could be saved. This is effective only if the items within transactions are not intermixed from different partitions. This is somehow constraining (e.g., in the case of market-basket data, we buy items in clusters, but in a transaction there may be items from more than one cluster). For this reason we do not exploit this direction. However, if this assumption holds, then the performance of MinePartitions can be further improved.

6. Performance results

6.1. Experimental setup

We examine the performance of the proposed method, henceforth denoted as partition-based mining (PBM). For comparison we consider the FP-growth algorithm (denoted as FP), since it has been reported to have an excellent performance. However, for a more complete conclusion, we also include a comparison with an Apriori-like algorithm and Eclat. As mentioned, for purposes of fair comparison with FP-growth, in MinePartitions algorithm we use FP-growth as an implementation of an FIM algorithm.

The proposed algorithm was implemented in C++, whereas a C++ implementation of FP-growth was obtained from the FIMI site.[9] The latter did not include the technique of database projection. For this reason we developed it on top of the existing code. Although the point when database projection should be invoked is not analyzed thoroughly in [8], the experimental results of the latter paper use the value of 3% for density in order to determine this point. The selection of the point affects the performance of FP-growth, because when FP-tree is small and fits in main memory, database projection may slow down FP-growth. In contrast, when FP-tree is large and does not fit in memory, then database projection overcomes the problem. We run the experiments by selecting each time the best value for this point. The implementation of Eclat was again taken from the FIMI site, whereas the Apriori-like algorithm was implemented in C++.

We used synthetic data sets produced by the generator of [7], which allow us to examine the impact of several parameters. Each synthetic data set is denoted as TxIyDz, where x is the average transaction size, y is the average pattern size, and z is the number of transactions. Also we used two real data sets. The first one includes papers from the DBLP site (dblp.uni-trier.de), where each transaction contains the references included in each paper. There are 7942 transactions (we only used those papers whose references are provided) and 20,826 distinct papers that are cited (this is the size of the items' domain). The second one is obtained from the Machine Learning Database Repository[10] and includes the Casts table from the Movies data set. The transactions contain the actors included in the cast of the listed movies. There are 8766 transactions and 16,608 actors. Although both real data sets are of moderate size (in terms of the number of transactions), they are used because of their relatively high domain and, most importantly, because due to their nature they are expected to contain clusters.

We examine the impact of the following parameters: support threshold, domain's size, number of transactions, and size of transactions. For PBM we examine the impact of L_p, the benefit from employing the proposed dynamic approach for finding the partitions against a simple, static approach, and the impact of hash-table size H. The performance measure is the total execution time, which for PBM includes the time to find the partitions. For all algorithms we do not measure the time to report the results (frequent itemsets), since it is the same for all cases. For the synthetic data, the default data set is the T10I4D100K with domain size equal to 100,000. For PBM, the default value for L_p is equal to 500 and H (the size of hash-tables for the linked-list representation) is set equal to 150.

6.2. Comparison with FP-growth

In the sequel, we demonstrate the performance comparison between PBM and FP for synthetic data sets, by varying several parameters, such as support threshold, number of items, number of transactions, and average transaction size. We also compare the two methods with two real data sets.

6.2.1. Varying the support threshold

We compared PBM and FP for two cases of the data set T10I4D100K. The first has a domain with size 1000 (small domain), whereas the second with 100,000 (large domain). The relative times (normalization against FP) for the small domain, with

[9] URL: http://fimi.cs.helsinki.fi/.

[10] URL: http://www.ics.uci.edu/~mlearn/MLRepository.html.

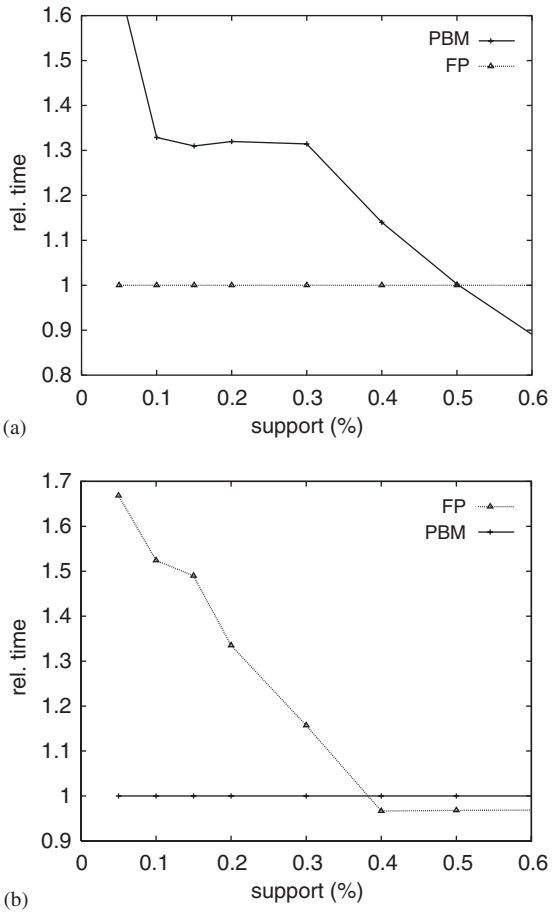

Fig. 8. Relative time vs. support threshold.

respect to support threshold, are depicted in Fig. 8a. Fig. 8b depicts the relative times (normalization against PBM) for the large domain. We use relative times to clearly compare the performance differences between the two cases, since the number of patterns and, thus, absolute execution times differ between them.

As expected, for the small domain, FP outperforms PBM for lower and medium support thresholds. The reason is that the cost of detecting partitions and mining them separately does not payoff for smaller domains, since only a single large partition is detected. FP is quite efficient for this case, because the FP-tree is compact. Notice that as support threshold increases, the performance of both algorithms tends to become comparable, whereas for larger supports PBM performs marginally better. When the support threshold is large, partitions are starting to exist (several items/itemsets become infrequent and the single large partition breaks into smaller ones). This helps PBM in improving its efficiency. Nevertheless, the absolute execution times for large supports are very small and the comparison is of little significance.

For the large domain, PBM clearly outperforms FP for lower and medium support thresholds. In this case there exist several partitions and FP is not as efficient as in the case of the small domain. As support threshold increases, the performance of both algorithms becomes comparable. For large thresholds FP is marginally better, because the number of frequent items (i.e., the effective size of the domain) becomes small. Again, the absolute execution times for large supports are very small and thus the comparison is not significant.

6.2.2. *Varying the number of items*

We now move on to examine varying domain's sizes. First we use the T10I4D100K data set and support threshold is set to 0.1%. As previously, between different domain sizes, absolute execution

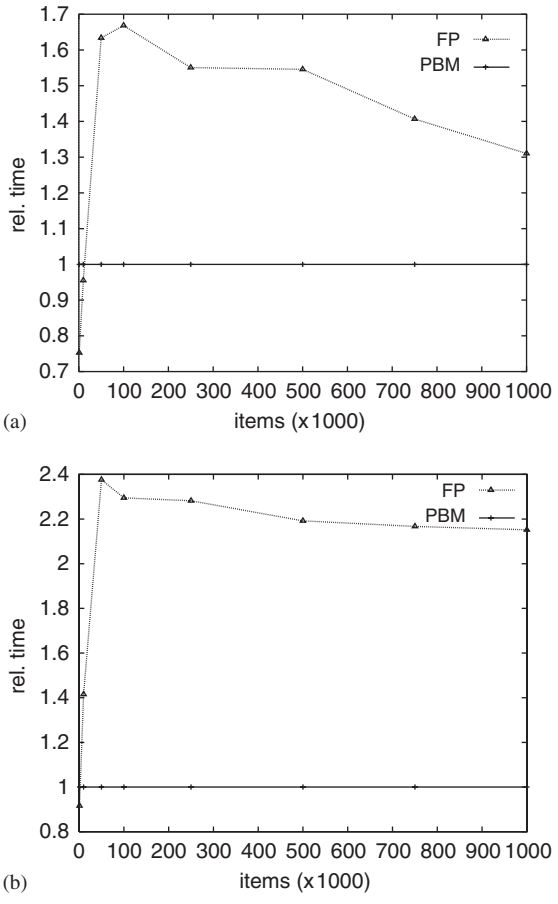

Fig. 9. Relative time vs. number of items.

times vary significantly. For clearer comparison, Fig. 9a illustrates the relative time (normalized by PBM) for varying domain size. For small domains, FP performs better than PBM. However, as the size of the domain increases, PBM becomes better. The performance difference reaches a peak around the value of 100,000 and then it slightly decreases. The reason is that the number of transactions is not large (100K). When the domain becomes too large, as the number of transactions is kept constant, many items become infrequent. This can be more clearly understood by the result depicted in Fig. 9b, where the relative times are given for the T10II4D300K (again the support threshold is 0.1%). Since the number of transactions is larger, the very large domains still have a large number of frequent items. Thus, the performance difference between PBM and FP does not decrease as much as previously. From this it is understood that the performance difference is affected by the combination of domain's size and database's size, and PBM is efficient in the challenging case when both are large.

6.2.3. Varying the number of transactions

Next, we separately test the impact of the database's size by using varying number of transactions. The data set is T10I4Dz with domain size 100,000, and the support threshold is 0.1% (the absolute value varies with varying z value). For FP, the size of the database can cause the FP-tree to not fit in main memory. For this reason we examine two ranges for the number of transactions. The (absolute) execution times for the first range are depicted in Fig. 10a, where FP-tree can always fit in memory and database projection is not used for FP. As shown, in this range both algorithms scale linearly to the database's size. As the latter increases, PBM significantly outperforms FP. The reason is that, as the number of transactions increases, the single FP-tree of FP becomes less condense (more transactions means more different itemsets and thus paths to be inserted). This increases the cost of the mining performed by FP.

The results for the second range of database's sizes are illustrated in Fig. 10b (time axis is logarithmic). In this range, as the number of transactions is increased, the FP-tree starts to not fit in main memory. In the figure, this happens after the size of 1.6×10^6. After this point, we illustrate the execution time of FP both when it plainly resorts to virtual memory and when

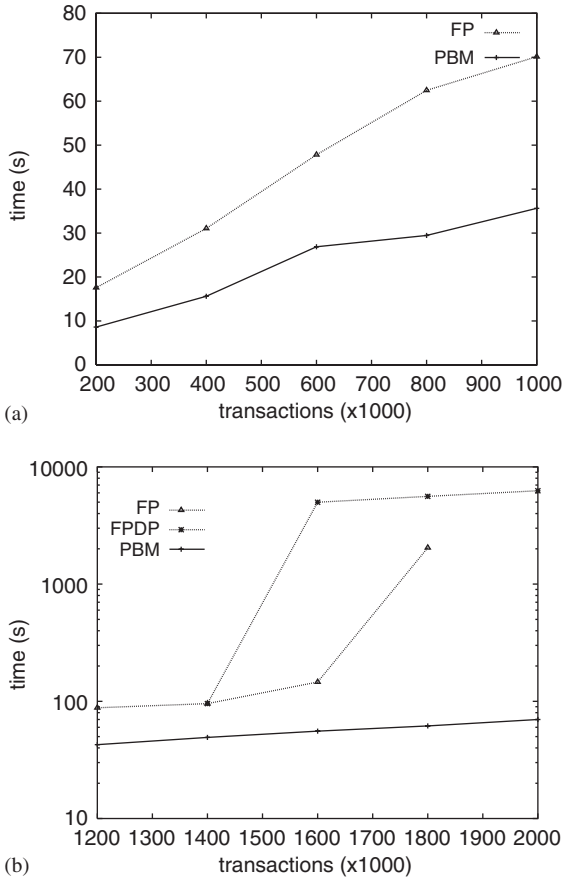

Fig. 10. Execution time vs. number of transactions: (a) range where FP-tree fits in memory, (b) range where FP-tree does not fit in memory (logarithmic vertical axis).

database projection is used (this case is denoted as FPDP). Although database projection avoids the problem of lack of main memory, it presents very high execution times. This is explained by the large domain's size, because according to [8], projection is applied separately for each item. Thus, it has to be applied for a very large number of times and this produces a significant overhead. Regarding the case of FP that resorts to virtual memory, when the size of the FP-tree marginally exceeds available memory, its performance does not degrade very badly. In contrast, for database's size higher than the critical point, thrashing causes an important degradation of FP's performance, and for very large values it fails to finish (due to lack of resources). This result explains the advantage of PBM, which, due to its partitioning technique and optimizations, is able to avoid the problem.

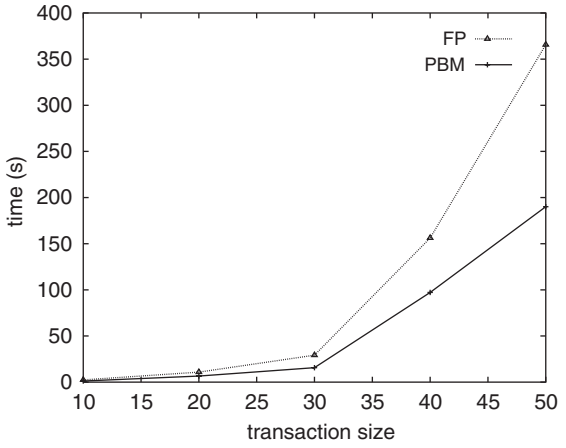

Fig. 11. Execution time vs. average transaction size.

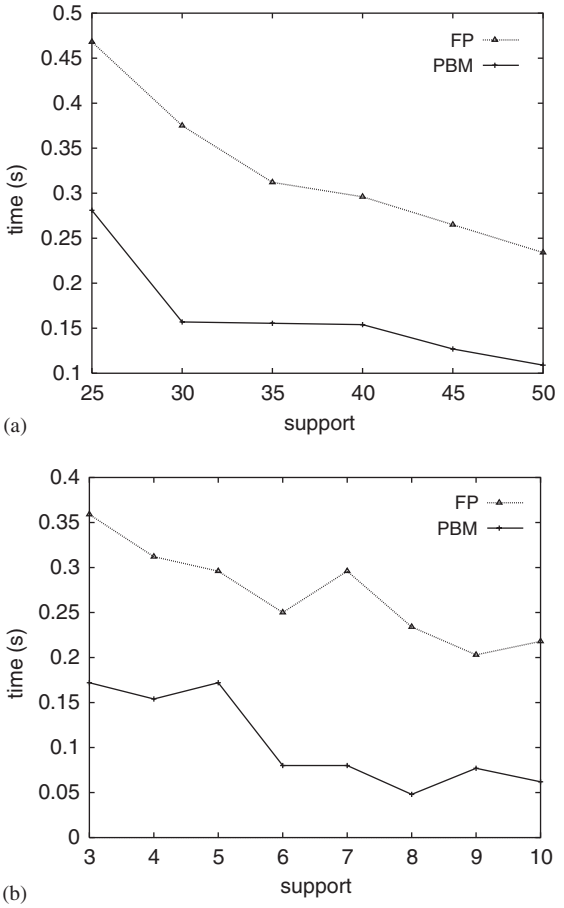

Fig. 12. Execution time vs. support (absolute) for real data: (a) DBLP database, (b) Movies database.

6.2.4. Varying the transaction size

Another parameter that we investigate is the size of transactions. We use the TxIyD100K data set, where y is set as half of x.[11] The support threshold is set to 0.1%. The results are given in Fig. 11. With increasing transactions' size, the difference in execution times also increases. Larger transactions produce more different items' combinations to be inserted in the FP-tree (however, main-memory shortage does not occur). Hence, FP is impacted. The same holds for PBM (recall that FIM is implemented by FP). However, the impact is less on PBM, because each application of FIM takes into account in each transaction only the items that belong to the considered partition. Therefore, partitioning manages to limit the increase in the cost produced by larger transactions' sizes.

6.2.5. Comparison for real data sets

Finally, we compare PBM and FP using the two real data sets. The results for the DBLP data set with respect to the support threshold (absolute) are illustrated in Fig. 12a, whereas the results for the Movies database in Fig. 12b. L_p is set to 40 in both cases. These data sets are of moderate size in terms of the number of transactions. However, their domains' sizes are relatively large and, due to their nature, they contain several partitions. Therefore, due to the existence of partitions in both cases, PBM compares favorably against FP. In particular, the DBLP database, for support equal to 25, contains four partitions: one with size 170, two with size 40, and one with size 19. Analogously, the Movies database, for support equal to 4, contains five partitions: one with size 169, three with size 40, and one with size 19. With increasing support threshold, the number of frequent items reduces, thus the number of partitions reduces too and the execution time of PBM and FP becomes smaller.

6.3. Comparison with other algorithms

In order to draw a more complete conclusion about PBM, we include a comparison with an Apriori-like algorithm and Eclat. The Apriori-like algorithm uses the hashing technique proposed in [13], which tries to reduce the number of candidates

[11] If y remained static, the larger transactions would contain many "random" items (i.e., not belonging to a pattern), a case that would not be characteristic for the measurement of performance.

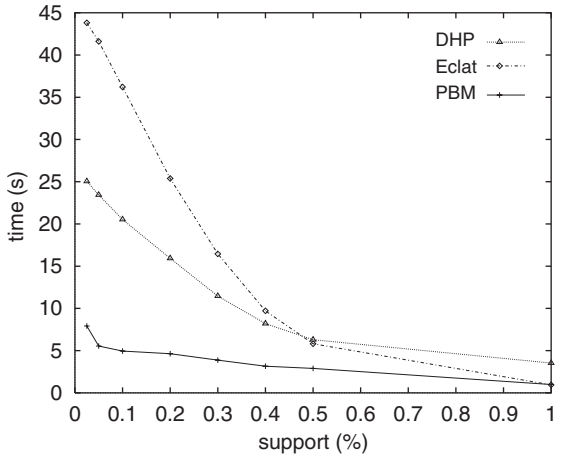

Fig. 13. Comparison with other algorithms: time vs. support.

(applied for the second phase). We use the T10I4D100K data set with domain's size equal to 100,000. The result for varying support are depicted in Fig. 13. Eclat, for small and medium support thresholds, is severely impacted by the large domain's size, for the reasons that have been explained in Section 2. The execution time of the Apriori-like algorithm is also high. The reason is that, despite the use of the hashing technique, it produces a very large number of candidates. As support threshold increases, the execution time of all algorithms converges to a point. In all cases, PBM presents the best performance.

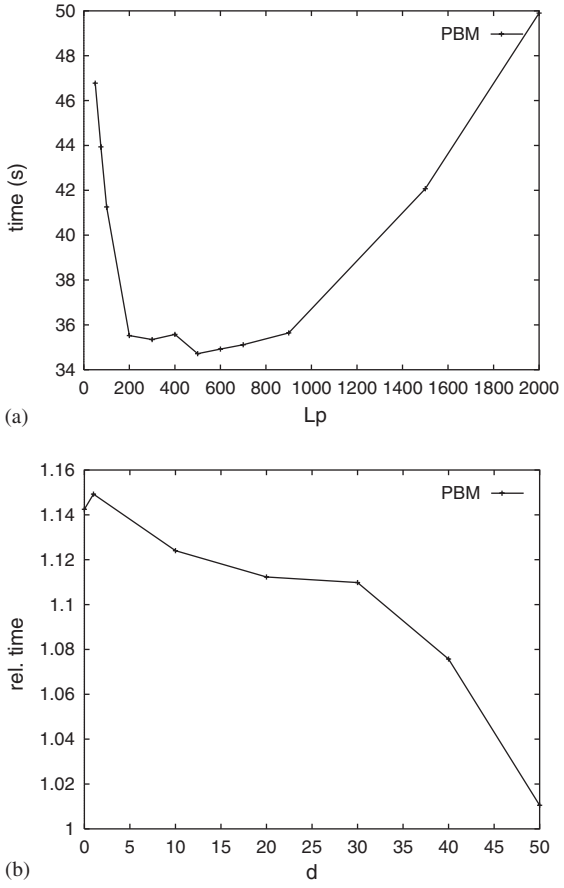

Fig. 14. (a) Execution time w.r.t. L_p, (b) relative execution time w.r.t. d.

6.4. Sensitivity of PBM

To understand the characteristics of PBM in more detail, we measure the impact of L_p, how helpful is the dynamic graph building, and the impact of parameter H.

First, we focus on L_p. We use the T10I4D500K data set, with domain size equal to 100,000. The execution times for varying L_p values are depicted in Fig. 14a (support set to 0.1%). As has been described, very low and very high values of L_p result in high execution times. The former causes a problem because of the unnecessarily large number of invocations of the FIM algorithm. The latter produces very few large partitions. Therefore, the corresponding large sub-domains impact the execution time.

We now evaluate the quality of our analytical estimation to set L_p equal to the average partition size. We want to avoid the impact of singleton and very small partitions on the calculation of the average partition size. Thus, we do not consider in the calculation of the average the partitions with size less than a threshold d. We measure the execution time of PBM, when L_p is analytically tuned in the aforementioned way, vs. d. Fig. 14b presents the latter time relatively to the execution time that results when we choose manually the best L_p value (taken from the previous experiment). When d is very small, the resulting L_p value is affected and, thus, the execution time is relatively high. As d increases, L_p quickly improves and the execution time reduces. When d is set around 50 (that is, when we not consider partitions that are smaller than about the 10% of the average partition size, which is 480), the execution time of PBM is very close to the case when L_p is manually tuned to the optimum value.

Next, we test the quality of the analytical prediction for H, which is the size of hash-tables

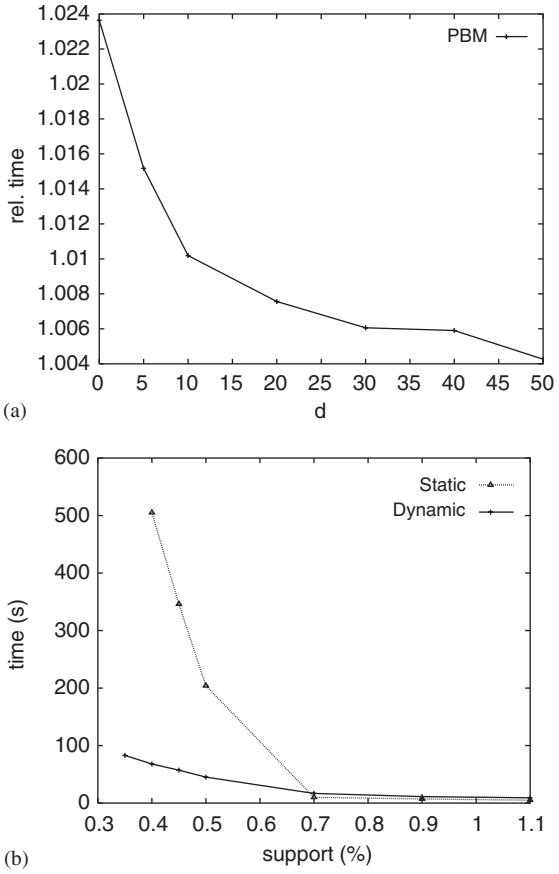

Fig. 15. (a) Evaluation of the analytical prediction for H, (b) comparison between static and dynamic partitioning.

that are used in the adjacency-list representation of PBM. We use the synthetic data set T10I4D500K with domain size 100,000 and support threshold equal to 0.1%. For the analytical prediction of H, we use the same threshold d as in the case of L_p. Similarly, we measure the relative execution time against the case where the best H value is selected manually. The results are given in Fig. 15a. For small values of d, the execution time of the analytically tuned PBM is relatively high, because a small H value is produced and long chains tend to be formed. As d increases, the relative time reduces. For d around 50, H takes a good value and the resulting execution time is very close to that of the best case.

Finally, we compare the dynamic component detection that PBM uses, against a simplistic, static approach. The latter uses an adjacency matrix to represent the array (we discard unnecessary rows that correspond to non-frequent items), finds the support of every candidate 2-itemset, keeps only the frequent ones as edges, and detects components with a depth-first traversal of the graph. We only measure the time required to detect components, since the mining of the partitions is not affected. To gain better insight, we use the T40I15D200K data set (domain size 100,000), which is challenging for the task of partition detection, since it contains many and large transactions. The results on execution time with respect to the support threshold are given in Fig. 15b. As support threshold decreases, the static approach clearly looses out, due to the large resulting graph and the cost of finding the support of many edges. For very low values of support threshold, the static approach fails to terminate, due to lack of memory to store the graph.

6.5. Estimating the number of disjoint partitions

In this section, we examine the effectiveness of sampling as a tool to estimate the number of disjoint partitions (henceforth denoted as clusters). We use the T10I4D500K data set. The support threshold is set to 0.1%. We measure the (absolute) difference between the number of clusters detected in the original data set and in the sample. This difference is denoted as error. Fig. 16a depicts the error, given as a percentage, with respect to the size of the sample. As shown, for samples larger than 25%, the error in estimation is zero. For sample sizes between 10% and 1%, the error is about 2%. Even when the sample becomes smaller (e.g., 0.7%), the error is not higher than 15%. This indicates that sampling is a useful tool to estimate the number of clustered partitions, as it yields precise estimations and does not require knowledge of any data set's factors.

To clarify further our discussion, we additionally examine factors that are derived from characteristics of the data set and affect the number of clusters. First we examine the impact of the number of items. We use the T10I4D100K data set. The number of patterns[12] is set to 1000 and the support threshold to 0.1%. The number of clusters for varying number of items is given in Fig. 16b (the horizontal axis is plotted in log scale). When the number of items is small, all of them correlate with each other. Thus very few disjoint clusters are produced. As the number of items increases, the

[12]The patterns in the generator of [7] act as seeds to generate the synthetic transactions.

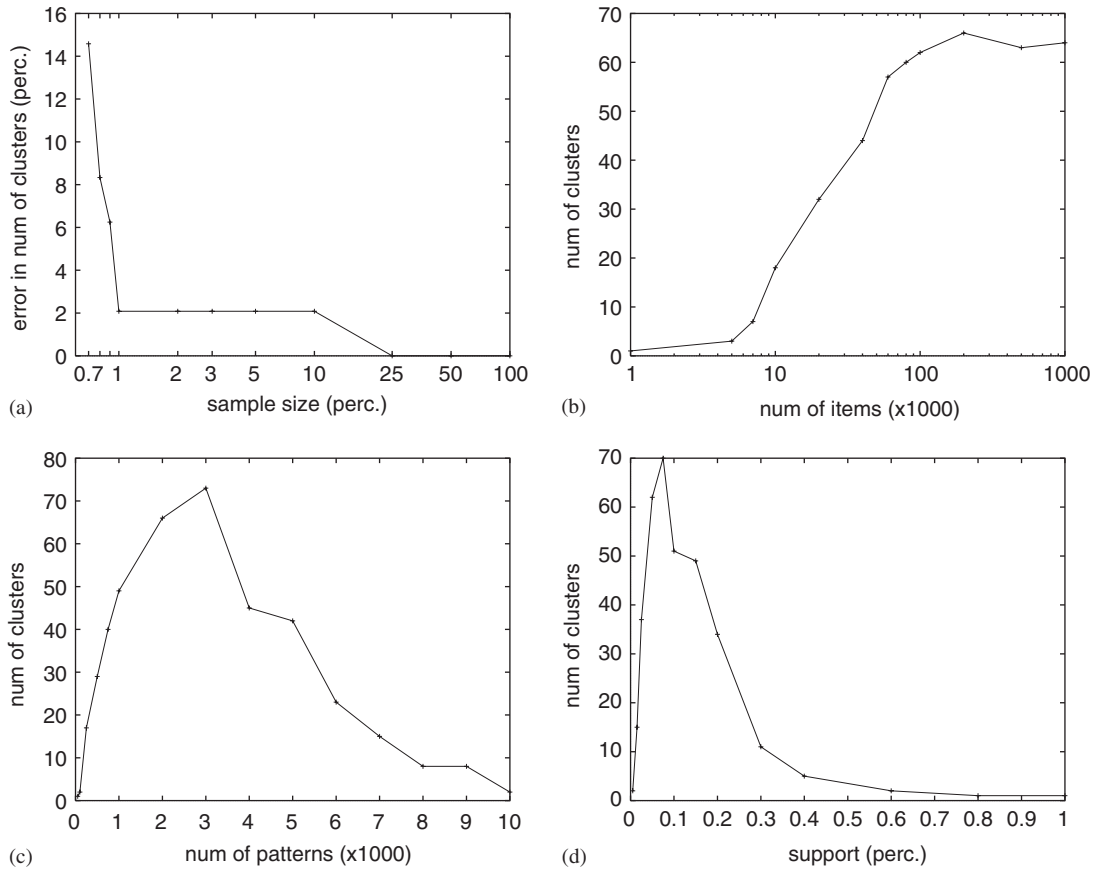

Fig. 16. (a) Error w.r.t. sample size, (b) number of clusters w.r.t. number of items, (c) number of clusters w.r.t. number of patterns, (d) number of clusters w.r.t. support threshold.

number of such clusters increases. After a point, the latter number remains about constant, as the additional items do not produce more clusters, because the number of patterns in the data set is kept fixed.

Next, we examine the impact of the number of patterns in the data set. Fig. 16c illustrates the number of clusters with respect to the number of patterns. We use the same data set as in the previous experiment, where we set the number of items equal to 100,000. When the number of patterns is small, very few clusters are produced, as they correspond to the items that these few patterns contain. As the number of patterns increases, the number of clusters also increases. However, after a point, the latter number reduces. This is explained by the fact that when there are many patterns, the support of their corresponding itemsets reduces (because the number of transactions is fixed).

Finally we examine the impact of the support threshold. Fig. 16d gives the number of clusters with respect to the latter threshold. When support threshold is small, all items correlate with all others, thus the number of disjoint clusters is small. As support threshold increases, the number of clusters increases too. However, when support threshold increases after a point, then the number of clusters starts to reduce. The reason for this fact is that, as support threshold increases, fewer items manage to become large, thus the number of clusters is reduced, as there are less items to form them.

In summary, there are several factors, whose interaction with the underlying mechanism that generates the clusters affects the resulting number of disjoint clusters. The aforementioned results helped to produce qualitative conclusions about the impact of the examined factors. Moreover, our results demonstrated that sampling is a precise tool to

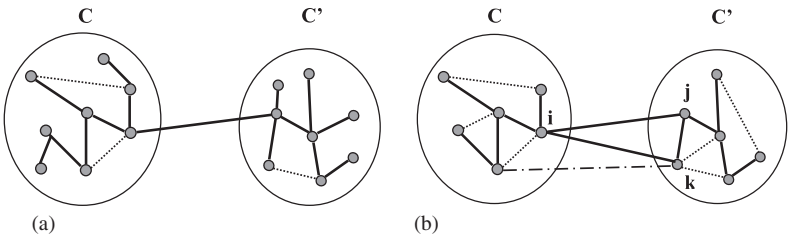

Fig. 17. (a) Example of a bridge between two components, (b) example of the additional searching for an item $k \in C'$ that is connected to item $i \in C$.

estimate the number of clusters without requiring knowledge about any such factors.

7. Non-well-separated partitions

In all cases examined so far, due to the nature of data and the large size of the domains, we could detect clearly separated partitions. But one may ask (a) if it is possible to have data with partitions that are not clearly separated and (b) how the partitions can be detected in this case. The most typical situation to consider for this case is the existence of "noise" in the form of edges that operate as "bridges", connecting items from different partitions, forcing their merging, and thus rendering their separate mining impossible. See Fig. 17a for an illustration of a bridge.

Edges that act as bridges can be identified by revisiting the EarlyMerging procedure (see Section 4.2.2) and performing a test before deciding to merge two components. For two components C and C', when we find that the support of an edge (i,j), $i \in C$, $j \in C'$, has reached the value of *sup*, we perform an additional searching procedure. We search if there exists any edge with support equal or larger[13] than *sup* between $i \in C$ and a $k \in C'$, where $k \neq j$ (see Fig. 17b for an example). If there is no such edge, we equivalently examine if there exists any edge with support equal or higher than *sup* between $j \in C'$ and an $l \in C$, where $l \neq i$. If an edge of any of the two aforementioned types (connected to i or connected to j) exists, then we merge C and C'. Otherwise, we can temporarily defer the merging of C and C'. The reason is that, up to now (due to the property of antimonotonicity), there cannot be any frequent itemset with items from both C and C', besides the 2-itemset (i,j). We utilize a list B that maintains such edges (i,j) as candidate bridges. It is possible that an edge in B may turn out not to be a bridge, if the corresponding components are merged in a following step. Nevertheless, it is not worth deleting them from B. After the end of the entire partitioning process, a traversal of B can discard the edges that are not bridges (by examining for each edge (i,j) in B if i and j belong to different components). We have to report the actual bridges as frequent 2-itemsets too, so that these patterns will not be missed from the output.

Notice that the structure of adjacency lists that maintains the graph is not directed, thus for an edge (i,j), $i<j$, we only store j in i's adjacency list. However, the previously described searching procedure may have to examine the adjacency lists of both i and j. For this reason, when the support of an edge (i,j) reaches the threshold and (i,j) is detected as candidate bridge, we additionally insert i into j's adjacency list. With a small increase in space cost, we save a lot of time during the searching.

During the first steps of the domain partitioning algorithm, a lot of sub-partitions will be connected by candidate bridges. Many of them will turn out not to be actual ones, since several sub-partitions will eventually merge. By deferring merging at initial steps, unnecessary cost incurs due to the previously described searching procedure. We avoid this shortcoming by deferring a merging only when $|C| + |C'| \geqslant L_p$, that is, when the components have an adequate size.

To test the effectiveness of the aforementioned procedure, we examined a synthetically generated data set. The reason is that the two real data sets we examined in Section 6 have clearly separated partitions and do not include any bridges. Moreover, with synthetic data we have the advantage of controlling the number of bridges. For the examined synthetic data set, we artificially generated 10

[13]As will be explained, some mergings may be deferred. Therefore, in contrast to the case when we merge as early as possible, there may exist edges with support higher than *sup*.

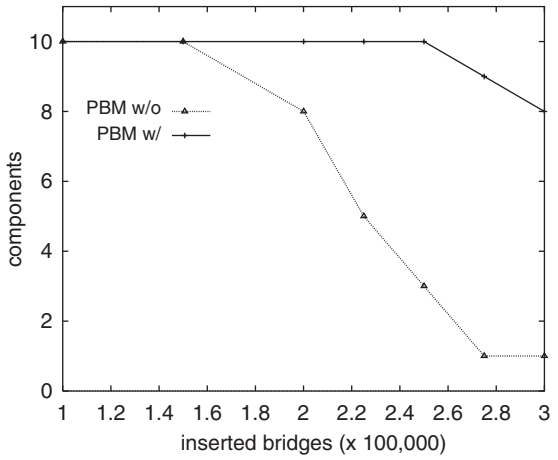

Fig. 18. The number of detected partitions vs. the number of inserted bridges.

partitions that do not share any items. Each partition contained 1000 items and from the items of each partition we generated 10,000 transactions (average transaction size 10 and average pattern-size 4). Next, we inserted transactions containing bridges, i.e., each transaction contained one 2-itemset between items belonging to different partitions. To control the effect of bridges, only a small number of items from each partition could participate in a bridge (we examined 50), otherwise the bridges could not reach an adequate support. Fig. 18 depicts the number of partitions detected with and without performing the previously described procedure (denoted as PBM w/ and PBM w/o, resp.), with respect to the number of added bridges. As shown, PBM w/ can significantly reduce the possibility of incorrect merging.

8. Conclusion

The mining of association rules from domains with very large number of items is becoming an exigency, due to emerging applications such as bioinformatics, e-commerce, and bibliographic analysis. Recently, this fact prompted researchers to develop methods that address the asymmetry between the number of rows and the number of columns in a data set. In this paper, for very large domains, we give insight about the presence of groups determined by correlations between the items.

Based on the aforementioned observation, we considered the (classic) problem of mining all association rules when both the number of rows and the number of columns are very large, thus no assumptions are made that the process can unconditionally be performed in main memory. To our knowledge, our results are the first to consider scalability to databases with hundreds of thousands of items and millions of transactions.

We developed an algorithm that partitions the domain of items according to their correlations. For the partitioning algorithm we provide several optimizations, which attain fast execution and low space cost. Additionally, we described a mining algorithm that carefully combines partitions to improve the efficiency of the mining procedure.

Our experiments examined the impact of several parameters through the use of synthetic data and two characteristic real data sets. The results clearly show the superiority of the proposed method against existing algorithms. In summary, the proposed method overcomes the problems caused by the combination of a large domain and a large number of transactions. These problems are the significant increase in CPU cost and possible I/O thrashing, which impact existing algorithms.

There are some interesting points of future work that can be examined. The first one is to diverge from the classic problem of mining all association rules with respect to minimum support and confidence, and to consider the mining of local pattern within each group of the domain. This way there will be no need to use relatively low support thresholds. Instead, we could discover strong patterns within each group. Another direction we could follow is to consider data sets where items are organized into groups with in-between connections that present scale-free properties. That is, some groups act like hubs and make difficult the separation of the others. Towards this direction, we have started implementing an algorithm that finds partitions with items shared with other partitions. The goal is to include in a partition as few shared items as possible, by categorizing items according to their degree of connectivity.

References

[1] R. Aggrawal, T. Imielinski, A. Swami, Mining association rules between sets of items in very large databases, in: Proceedings of the ACM SIGMOD Conference, 1993, pp. 207–216.

[2] J. Hipp, U. Guntzer, G. Nakhaeizadeh, Algorithms for association rules mining—a general survey and comparison, SIGKDD Explor. 2 (1) (2000) 58–64.

[3] P. Bradley, J. Gehrke, R. Ramakrishnan, R. Srikant, Scaling mining algorithms to large databases, Commun. ACM 45 (8) (2002) 38–43.
[4] D. Watts, S. Strogatz, Collective dynamics of small-world networks, Nature 363 (1998) 202–204.
[5] F. Pan, G. Cong, A.K.H. Tung, Carpenter: finding closed patterns in long biological datasets, in: Proceedings of the SIGKDD Conference, 2003.
[6] F. Pan, A.K.H. Tung, G. Cong, X. Xu, Cobbler: combining column and row enumeration for closed pattern discovery, in: Proceedings of the SSDBM Symposium, 2004, pp. 21–30.
[7] R. Agrawal, R. Srikant, Fast algorithms mining association rules in large databases, Technical Report RJ 9839, IBM Almaden Research Center, San Jose, CA, 1994.
[8] J. Han, J. Pei, Y. Yin, R. Mao, Mining frequent patterns without candidate generation: a frequent-pattern tree approach, Data Min. Knowl. Discovery 8 (2004) 53–87.
[9] M.J. Zaki, Scalable algorithms for association mining, IEEE Trans. Knowl. Data Eng. 12 (3) (2000) 372–390.
[10] A. Savasere, E. Omiecinski, S. Navathe, An efficient algorithm for mining association rules in large databases, in: Proceedings of the VLDB Conference, 1995, pp. 432–444.
[11] B. Goethals, M. Zaki, Advances in frequent itemset mining implementations: introduction to FIMI03, in: Proceedings of the FIMI Workshop, 2003.
[12] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, A.I. Verkamo, Fast discovery of association rules, in: Advances in Knowledge Discovery and Data Mining, 1996, pp. 307–328.
[13] J.S. Park, M.-S. Chen, P. Yu, Using a hash-based method with transaction trimming for mining association rules, IEEE Trans. Knowl. Data Eng. 9 (5) (1997) 813–825.
[14] R. Srikant, R. Aggrawal, Mining generalized association rules, in: Proceedings of the VLDB Conference, 1995, pp. 407–419.
[15] R. Agarwal, C. Aggarwal, V.V. Prasad, A tree projection algorithm for generation of frequent itemsets, J. Parallel Distrib. Comput. 61 (2001) 350–371.
[16] D. Burdick, M. Calimlim, J. Gehrke, Mafia: a maximal frequent itemset algorithm for transactional databases, in: Proceedings of the ICDE Conference, 2001, pp. 443–452.
[17] B. Goethals, Memory issues in frequent itemset mining, in: Proceedings of the SAC Symposium, 2004, pp. 530–534.
[18] A. Nanopoulos, Y. Manolopoulos, Memory-adaptive association rules mining, Inf. Syst. 29 (5) (2004) 365–384.
[19] M. Zaki, C. Hsiao, Charm: an efficient algorithm for closed association rule mining, in: Proceedings of the SDM Conference, 2002.
[20] J. Wang, J. Han, J. Pei, Closet+: searching for the best strategies for mining frequent closed itemsets, in: Proceedings of the KDD Conference, 2003, pp. 236–245.
[21] R. Agrawal, S. Sarawagi, S. Thomas, Integrating association rule mining with databases: alternatives and implications, in: Proceedings of the SIGMOD Conference, 1998, pp. 343–354.
[22] G. Cong, A.K.H. Tung, X. Xu, F. Pan, J. Yang, Farmer: finding interesting rule groups in microarray datasets, in: Proceedings of the SIGMOD Conference, 2004, pp. 143–154.
[23] X.-R. Jiang, L. Gruenwald, Microarray gene expression data association rules mining based on BSC-tree and FIS-tree, Data Knowl. Eng. 53 (1) (2005) 3–29.
[24] C. Aggarwal, C. Procopiuc, P. Yu, Finding localized associations in market basket data, IEEE Trans. Knowl. Data Eng. 14 (1) (2002) 51–62.
[25] Y.-J. Tsay, Y.-W. Chang-Chien, An efficient cluster and decomposition algorithm for mining association rules, Inf. Sci. 160 (2004) 161–171.
[26] G. Lee, K.L. Lee, A.L. Chen, Efficient graph-based algorithms for discovering and maintaining association rules in large databases, Knowl. Inf. Syst. 3 (3) (2001) 338–355.
[27] S.-J. Yen, A.L. Chen, A graph-based approach for discovering various types of association rules, IEEE Trans. Knowl. Data Eng. 13 (5) (2001) 839–845.
[28] T.H. Cormen, C.E. Leiserson, R.L. Rivest, Introduction to Algorithms, The MIT Press, Cambridge, MA, 1990.
[29] B. Salzberg, Merging sorted runs using large main memory, Acta Inf. 27 (3) (1989) 195–215.