Three Strategies for Concurrent Processing of Frequent Itemset Queries Using FP-growth*

Marek Wojciechowski, Krzysztof Galecki, Krzysztof Gawronek

Poznan University of Technology
Institute of Computing Science
ul. Piotrowo 2, 60-965 Poznan, Poland
marek@cs.put.poznan.pl

Abstract. Frequent itemset mining is often regarded as advanced querying where a user specifies the source dataset and pattern constraints using a given constraint model. Recently, a new problem of optimizing processing of sets of frequent itemset queries has been considered and two multiple query optimization techniques for frequent itemset queries: Mine Merge and Common Counting have been proposed and tested on the Apriori algorithm. In this paper we discuss and experimentally evaluate three strategies for concurrent processing of frequent itemset queries using FP-growth as a basic frequent itemset mining algorithm. The first strategy is Mine Merge, which does not depend on a particular mining algorithm and can be applied to FP-growth without modifications. The second is an implementation of the general idea of Common Counting for FP-growth. The last is a completely new strategy, motivated by identified shortcomings of the previous two strategies in the context of FP-growth.

1 Introduction

Discovery of frequent itemsets [1] is a very important data mining problem with numerous practical applications. Informally, frequent itemsets are subsets frequently occurring in a collection of sets of items. Frequent itemsets are typically used to generate association rules. However, since generation of rules is a rather straightforward task, the focus of researchers has been mostly on optimizing the frequent itemset discovery step.

Many frequent itemset mining algorithms have been developed. The two most prominent classes of algorithms are Apriori-like (level-wise) and pattern-growth methods. Apriori-like solutions, represented by the classic Apriori algorithm [3], perform a breadth-first search of the pattern space. Apriori starts with discovering frequent itemsets of size 1, and then iteratively generates candidates from previously found smaller frequent itemsets and counts their occurrences in a database scan. The

* Part of this work has been published by the authors as “Concurrent Processing of Frequent Itemset Queries Using FP-Growth Algorithm” in the Proceedings of the 1st ADBIS Workshop on Data Mining and Knowledge Discovery (ADMKD'05), Tallinn, Estonia, 2005.

problems identified with Apriori are: (1) multiple database scans, and (2) huge number of candidates generated for dense datasets and/or low frequency threshold (minimum support).

To address the limitations of Apriori-like methods, a novel mining paradigm has been proposed, called pattern-growth [8], which consists in a depth-first search of the pattern space. Pattern-growth methods also build larger frequent itemsets from smaller ones but instead of candidate generation and testing, they exploit the idea of database projections. Typically, pattern-growth methods start with transforming the original database into some complex data structure, preferably fitting into main memory. A classic example of the pattern-growth family of algorithms is FP-growth [9][10], which transforms a database into an FP-tree stored in main memory using just 2 database scans, and then performs mining on that optimized FP-tree structure.

Frequent itemset mining is often regarded as advanced database querying where a user specifies the source dataset, the minimum support threshold, and optionally pattern constraints within a given constraint model [11]. A significant amount of research on efficient processing of frequent itemset queries has been done in recent years, focusing mainly on constraint handling (see [18] for an overview) and reusing results of previous queries [5][7][15][16].

Recently, a new problem of optimizing processing of sets of frequent itemset queries has been considered, bringing the concept of multiple-query optimization to the domain of frequent itemset mining. The idea was to process the queries concurrently rather than sequentially and exploit the overlapping of queries' source datasets. Sets of frequent itemset queries available for concurrent processing may arise in data mining systems operating in a batch mode or be collected within a given time window in multi-user interactive data mining environments. A motivating example from the domain of market basket analysis could be a set of queries discovering frequent itemsets from the overlapping parts of a database table containing customer transaction data from overlapping time periods.

Two multiple-query optimization techniques for frequent itemset queries have been proposed: Mine Merge [24] and Common Counting [22]. Mine Merge is a general strategy that consists in transforming the original batch of queries into a batch of intermediate queries operating on non-overlapping datasets, and then using the results of the intermediate queries to answer the original queries. Although Mine Merge does not depend on a particular mining algorithm, its efficiency has been evaluated only for Apriori, and it is unclear how it would perform with pattern-growth algorithms like FP-growth. Common Counting has been specifically designed to work with Apriori-like algorithms. The idea of Common Counting is concurrent execution of Apriori for each query, and integration of dataset scans required by Apriori so that the parts of the dataset shared by the queries are read only once per Apriori iteration.

In this paper, we (1) generalize the strategy applied by Common Counting and adapt it to work with FP-growth in the form of the Common Building method; (2) propose a completely new strategy of processing of batches of frequent itemset queries, aiming at integrating the data structures used by the queries, and implement it for FP-growth as the Common FP-tree method; (3) experimentally evaluate the three strategies in the context of FP-growth.

1.1 Related Work

Multiple-query optimization has been extensively studied in the context of database systems (see [21] for an overview). The idea was to identify common subexpressions and construct a global execution plan minimizing the overall processing time by executing the common subexpressions only once for the set of queries [4][12][19]. Data mining queries could also benefit from this general strategy, however, due to their different nature they require novel multiple-query processing methods.

To the best of our knowledge, apart from the Common Counting and Mine Merge methods mentioned above, multiple-query optimization for frequent pattern queries has been considered only in the context of frequent pattern mining on multiple datasets [14]. The idea was to reduce the common computations appearing in different complex queries, each of which compared the support of patterns in several disjoint datasets. This is fundamentally different from our problem, where each query refers to only one dataset and the queries' datasets overlap.

Earlier, the need for multiple-query optimization has been postulated in the somewhat related research area of inductive logic programming, where a technique based on similar ideas as Common Counting has been proposed, consisting in combining similar queries into query packs [6].

As an introduction to multiple-data-mining-query optimization, we can regard techniques of reusing intermediate or final results of previous queries to answer a new query. Methods falling into that category that have been studied in the context of frequent itemset discovery are: incremental mining [7], caching intermediate query results [17], and reusing materialized complete [5][15][16] or condensed [13] results of previous queries provided that syntactic differences between the queries satisfy certain conditions.

1.2 Organization of the Paper

The remainder of the paper is organized as follows. In Sect. 2 we review basic definitions regarding frequent itemset mining and we briefly describe the FP-growth algorithm. Section 3 contains basic definitions regarding frequent itemset queries and presents the previously proposed multiple-query optimization techniques: Mine Merge and Common Counting. In Sect. 4 we present the Common Building method as an adaptation of Common Counting to FP-growth. In Sect. 5 we introduce a new strategy for concurrent processing of frequent itemset queries and its implementation for FP-growth, called Common FP-tree. Section 6 presents experimental results. Section 7 contains conclusions and directions for future work.

2 Frequent Itemset Mining and Review of FP-growth

Frequent itemsets. Let $L=\{l_1, l_2, ..., l_m\}$ be a set of literals, called items. Let a non-empty set of items T be called an *itemset*. Let D be a set of variable length itemsets, where each itemset $T \subseteq L$. We say that an itemset T *supports* an item $x \in L$ if x is in T.

We say that an itemset *T* *supports* an itemset $X \subseteq L$ if *T* supports every item in the set *X*. The *support* of the itemset *X* is the percentage of *T* in *D* that support *X*. The problem of mining frequent itemsets in *D* consists in discovering all itemsets whose support is no less than a user-defined minimum support threshold *minsup*.

FP-growth. The initial phase of FP-growth is the construction of a memory structure called FP-tree. FP-tree is a highly compact representation of the original database (in particular for so-called dense datasets), which is assumed to fit into the main memory. FP-tree contains only frequent items, each transaction has a corresponding path in the tree, and transactions having a common prefix share the common starting fragment of their paths. When storing a transaction in an FP-tree, only its frequent items are considered and are sorted according to a fixed order. Typically, frequency descending order is used as it is likely to result in a good compression ratio. The procedure of creating an FP-tree requires two database scans: one to discover frequent items and their counts, and one to build the tree by adding transactions to it one by one. An FP-tree for an example database represented by the first two columns of Table 1 and the minimum support threshold of 50% is presented in Fig. 1 (example from [9]).

Table 1. Example transaction database

TID	Items	Ordered frequent items
100	a;c;d;f;g;i;m;p	f;c;a;m;p
200	a;b;c;f;l;m;o	f;c;a;b;m
300	b;f;h;j;o	f;b
400	b;c;k;s;p	c;b;p
500	a;c;e;f;l;m;n;p	f;c;a;m;p

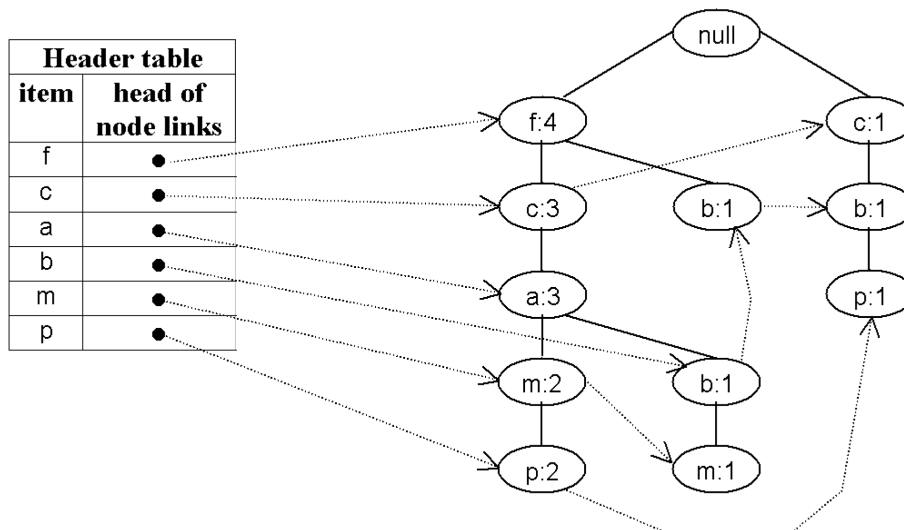

Fig. 1. An FP-tree for an example database

After an FP-tree is built, the actual FP-growth procedure is recursively applied to it, which discovers all frequent itemsets in a depth-first manner by exploring projections (called conditional pattern bases) of the tree with respect to frequent prefixes found so far. The projections are stored in memory in the form of FP-trees (called conditional FP-trees). FP-growth exploits the property that the support of an itemset $X \cup Y$ is equal to the support of Y in the set of transactions containing X (which forms the conditional pattern base of X). Thus, FP-growth builds longer patters from previously found shorter ones. Part of the FP-tree structure is the header table containing pointers to the lists containing all occurrences of given items in a tree, which facilitate the projection operation. It should be noted that after the FP-tree is created, the original database is not scanned anymore, and therefore the whole mining process requires exactly two database scans. The FP-growth algorithm is formally presented in Fig. 2, together with its initial tree-building phase. Our formulation differs slightly from that from [9] because we assume that *minsup* is relative to the total number of transactions. Therefore, in the first scan of the dataset we calculate the minimum required number of occurrences *mincount*, corresponding to *minsup* provided in a query. The *mincount* value is passed as a parameter to the FP-growth procedure. The term “frequent” within the algorithm implicitly refers to this *mincount* threshold.

Input: database *D*, minimum support threshold *minsup*
Output: the complete set of frequent itemsets
Method:

1. scan *D* to calculate *mincount* and discover frequent items and their counts
2. create the root of *FP-tree* labeled as *null*
3. scan *D* and add each transaction to *FP-tree* omitting non-frequent items
4. call *FP-growth*(*FP-tree*, *null*, *mincount*)

procedure *FP-growth*(*FP-tree*, α, *mincount*) {
  if *FP-tree* contains a single path *P*
  then for each combination β of nodes in *P* **do**
    generate frequent itemset $\beta \cup \alpha$
    with $count(\beta \cup \alpha, D)$ = min *count* of nodes in β;
  else for each a_i in header table of *FP-tree* **do** {
    generate frequent itemset $\beta = a_i \cup \alpha$
    with $count(\beta, D) = count(a_i, FP\text{-}tree)$;
    construct β's conditional pattern base and then
    β's conditional $FP\text{-}tree_\beta$;
    if $FP\text{-}tree_\beta \neq \emptyset$ **then** $FP\text{-}growth(FP\text{-}tree_\beta, \beta, mincount)$;
}

Fig. 2. FP-growth algorithm

FP-growth has been found more efficient than Apriori for dense datasets (i.e., containing numerous and/or long frequent itemsets) and for low support thresholds. Moreover, as stated in [18], FP-growth can incorporate more types of pattern constraints than Apriori. In particular, a class of convertible constraints has been identified, representing the constraints that can be handled by FP-growth by properly

ordering the items when storing a transaction in a tree (instead of “default” frequency descending order).

3 Multiple-Query Optimization for Frequent Itemset Queries

3.1 Basic Definitions and Problem Statement

Frequent itemset query. A frequent itemset query is a tuple $dmq = (R, a, \Sigma, \Phi, minsup)$, where R is a database relation, a is a set-valued attribute of R, Σ is a condition involving the attributes of R called *data selection predicate*, Φ is a condition involving discovered itemsets called *pattern constraint*, and *minsup* is the minimum support threshold. The result of *dmq* is a set of itemsets discovered in $\pi_a\sigma_\Sigma R$, satisfying Φ, and having support $\geq minsup$ (π and σ denote relational projection and selection operations respectively).

Example. Given the database relation $R_1(a_1, a_2)$, where a_2 is a set-valued attribute and a_1 is of integer type. The frequent itemset query $dmq_1 = (R_1, "a_2", "a_1>5", "|itemset|<4", 3\%)$ describes the problem of discovering frequent itemsets in the set-valued attribute a_2 of the relation R_1. The frequent itemsets with support of at least 3% and length less than 4 are discovered in the collection of records having $a_1>5$.

Elementary data selection predicates. The set $S=\{s_1, s_2, ..., s_k\}$ of data selection predicates over the relation R is a set of elementary data selection predicates for a set of frequent itemset queries $DMQ = \{dmq_1, dmq_2, ..., dmq_n\}$ if for all u,v we have $\sigma_{su}R \cap \sigma_{sv}R = \emptyset$ and for each dmq_i there exist integers $a, b, ..., m$ such that $\sigma_{\Sigma i}R = \sigma_{sa}R \cup \sigma_{sb}R \cup .. \cup \sigma_{sm}R$. The set of elementary data selection predicates represents the partitioning of the database determined by overlapping of queries’ datasets.

Example. Given the relation $R_1=(attr_1, attr_2)$ and three data mining queries: $dmq_1=(R_1, "attr_2", "5 <attr_1<20", \emptyset, 3\%)$, $dmq_2=(R_1, "attr_2", "0<attr_1<15", \emptyset, 5\%)$, $dmq_3=(R_1, "attr_2", "5<attr_1<15 \text{ or } 30<attr_1<40", \emptyset, 4\%)$. The set of elementary data selection predicates is then $S=\{s_1="0<attr_1<5", s_2="5<attr_1<15", s_3="15<attr_1<20", s_4="30<attr_1<40"\}$.

Problem Statement. Given a set of frequent itemset queries $DMQ = \{dmq_1, dmq_2, ..., dmq_n\}$, the problem of *multiple-query optimization* of *DMQ* consists in generating an algorithm to execute *DMQ* that minimizes the overall processing time.

In general, it is assumed that after collecting the queries to be concurrently executed using any strategy, duplicated queries are eliminated in a pre-processing step. It is also advisable to combine queries operating on exactly the same dataset (at least the ones that have the same data selection predicate) into one query, whose results can be used to answer the original queries by simple checking of pattern constraints and/or support. Such a new query should have the support threshold equal to the smallest threshold among the queries to be replaced and the pattern constraint in the form of a disjunction of their pattern constraints.

3.2 Mine Merge

Mine Merge employs the property that for a database divided into a set of disjoint partitions, an itemset frequent in a whole database, must also be frequent in at least one partition of it. This important property has been proved in [20] and served as the basis for a frequent itemset mining algorithm called Partition. The difference between Partition and Mine Merge is that Partition uses memory-based partitions, determined by the amount of available main-memory, while Mine Merge operates on disk-based partitions, which are the consequence of overlapping between queries' datasets.

/* Generate intermediate queries $IDMQ = \{idmq_1, idmq_2, ...\}$ */
$IDMQ \leftarrow \varnothing$
for each $s_j \in S$ **do begin**
$Q \leftarrow \{dmq_i \in DMQ \mid \sigma_{sj}R \subseteq \sigma_{\Sigma i}R\}$
$intermediate_minsup \leftarrow min\{minsup_i \mid dmq_i=(R, a, s_i, \Phi_i, minsup_i) \in Q\}$
$intermediate_\Phi \leftarrow \Phi_1 \vee \Phi_2 \vee ... \vee \Phi_{|Q|}, \forall i=1..|Q|, dmq_i=(R, a, s_i, \Phi_i, minsup_i) \in Q$
$IDMQ \leftarrow IDMQ \cup idmq_j=(R, a, s_j, intermediate_\Phi, intermediate_minsup)$
end
/* Execute intermediate queries */
for each $idmq_i \in IDMQ$ **do**
$IF_i \leftarrow execute(idmq_i)$
/* Generate results for original queries $DMQ = \{dmq_1, dmq_2, ...\}$ */
for each $dmq_i \in DMQ$ **do**
$C^i \leftarrow \{c \mid c \in \bigcup_k IF_k, \sigma_{sk}R \subseteq \sigma_{\Sigma i}R, c.support \geq minsup_i, c \text{ satisfies } \Phi_i\}$
for each $s_j \in S$ **do begin**
$CC \leftarrow \{C^i \mid \sigma_{sj}R \subseteq \sigma_{\Sigma i}R\}$; /* select the candidates to count now */
if $CC \neq \varnothing$ **then** $count(CC, \sigma_{sj}R)$;
end
for (i=1; i<=n; i++) **do**
$Answer^i \leftarrow \{c \in C^i \mid c.support \geq minsup_i\}$ /* generate final results */

Fig. 3. Mine Merge method

Mine Merge first generates a set of *intermediate queries*, in which each frequent itemset query is based on a single elementary data selection predicate only. The intermediate queries are derived from those original queries that are sharing a given elementary data selection predicate. The minimum support thresholds and pattern constraints for the intermediate queries are chosen so that their results are guaranteed to include all locally frequent itemsets for all the original queries that refer to the database partition corresponding to a given intermediate query, i.e., (1) the support threshold of an intermediate query is the smallest minimum support threshold value from all the relevant original queries, (2) the pattern constraint of an intermediate query is a disjunction of the pattern constraints of the relevant original queries.

Next, the intermediate queries are executed sequentially using any frequent itemset mining algorithm (Apriori, FP-growth, etc.). The pattern constraints that the chosen

algorithm can incorporate are pushed into the mining process, the remaining ones are verified in a post-processing phase.

The results of intermediate queries are merged to form global candidates for the original queries. For each of the original queries the set of its global candidates is a union of frequent itemsets from all the intermediate queries operating on subsets of its dataset. Since intermediate queries correspond to elementary data selection predicates, these intermediate queries represent a partitioning of the original query's dataset into a set of disjoint partitions. Thus, the set of global candidates is guaranteed to contain all frequent itemsets thanks to the property that in a partitioned dataset a pattern can be frequent only if it is frequent in at least one partition.

Finally, a database scan is performed to count the global candidate supports and to answer the original queries. The pseudocode of Mine Merge is shown in Fig. 3.

Obviously, efficiency of Mine Merge depends on the presence of efficient access paths to dataset partitions corresponding to elementary data selection predicates. In fact, Mine Merge must exploit ordering and/or indexing of the database relation containing the mined datasets. Otherwise, each of the intermediate queries would perform full relation scans, similarly as in sequential processing of the original queries. This would lead to worse performance of Mine Merge than in case of sequential processing because the number of intermediate queries generated by Mine Merge is greater than the number of original queries. Another problem with Mine Merge is that it introduces an extra database scan to generate final results from the results of the intermediate queries, and therefore requires significant overlapping of queries' datasets in order to outperform sequential processing. Finally, Mine Merge is not appropriate for large batches of queries as the number of intermediate queries in the worst case is 2^n-1, where n is the number of queries (all subsets of the set of queries except the empty set). In such worst-case scenarios gains thanks to I/O reduction may not compensate the increased amount of computations.

Although Mine Merge is independent of the mining algorithm used to execute intermediate queries, one can expect that its performance relative to sequential processing will depend on the chosen mining algorithm. For instance, the efficiency of Mine Merge for Apriori strongly depends on data distribution which has an impact on the number of Apriori iterations required for the intermediate queries. The same should be true for FP-growth, i.e., the size of FP-tree and processing time of the recursive FP-growth procedure for each intermediate query will depend on data distribution. Additionally, one can expect that in case of FP-growth, which requires exactly only two database scans, it will be more difficult for Mine Merge to compensate the cost of its extra database scan with the reduction of I/O thanks to dataset overlapping between the queries than it was for Apriori.

3.3 Common Counting

Common Counting was specifically developed for the Apriori algorithm. It consists in concurrent execution of a set of frequent itemset queries using Apriori and integrating their dataset scans. The method iteratively generates and counts candidates for all the data mining queries, storing candidates generated for each query in memory (in separate hash-tree structures). For each elementary data selection predicate, its

corresponding database partition is scanned once per iteration, and candidates for all the queries referring to that partition are counted.

An advantage of Common Counting over Mine Merge is that it does not introduce any significant computations and I/O operations apart from these performed by Apriori executions. Therefore, Common Counting outperforms sequential processing if any overlapping between queries' datasets occurs and in general is more predictable than Mine Merge. Another positive feature of Common Counting is that, contrary to Mine Merge, in order to outperform sequential processing it does not require efficient access paths to dataset partitions corresponding to elementary data selection predicates. Moreover, if full scans of database relation are necessary to identify datasets for each query, Common Counting is particularly efficient compared to sequential processing as it performs one full scan per Apriori iteration, serving all the queries. (Transactions are read sequentially and each of them is processed by the queries, whose data selection predicates it satisfies.)

One problem with Common Counting is that it needs to maintain data structures (candidate hash-trees) of several queries in main memory at the same time. If the candidates of all the queries do not fit into memory, the counting process is divided into phases, and queries are scheduled into phases so that an overall I/O cost is minimized [23][25].

4 Common Building: Adaptation of Common Counting for FP-growth

Common Counting as formulated for Apriori cannot be applied directly to FP-growth because FP-growth does not perform candidate generation and counting. However, we can exploit the general strategy of Common Counting, which is integration of operations performed by a set of queries during the scan of the common part of the dataset. In case of FP-growth, the database is scanned 2 times (during the FP-tree building phase), and these two scans can be integrated for the collection of queries for which FP-trees are to be built. Thus, our adaptation of Common Counting to FP-growth will consist in concurrent building of FP-trees in main memory for a batch of queries, and therefore will be called Common Building. The Common Building method for FP-growth for two concurrent queries dmq_1 and dmq_2 can be formalized as presented in Fig. 4. Generalization of the procedure for an arbitrary number of queries is straightforward.

Integration of common I/O operations takes place only during the tree-building step, the FP-growth recursive procedure is not affected by the multiple-query processing strategy. D^1 and D^2 denote parts of the database read by dmq_1 and dmq_2 respectively. Similarly, $mincount^1$ and $mincount^2$ are minimum required numbers of occurrences for an itemset to be frequent for dmq_1 and dmq_2 respectively. $FP\text{-}tree^1$ and $FP\text{-}tree^2$ are separate FP-tree structures containing compressed datasets for dmq_1 and dmq_2 as proposed in [9].

It should be noted that Common Building for FP-growth preserves one of the crucial positive features of Apriori Common Counting as it also does not rely on the presence of efficient access paths to dataset partitions corresponding to elementary

data selection predicates for its efficiency, and is even more advantageous if full scans are the only (or the most efficient) choice. If full scans of the database relation are necessary, Common Building will build FP-trees for all the queries using two scans, whereas in case of sequential processing each query would need its own two scans.

1. scan D to calculate $mincount^1$ and $mincount^2$,
   and discover frequent items for dmq_1 and dmq_2
2. create the root of $FP\text{-}tree^1$ labeled as *null*
3. create the root of $FP\text{-}tree^2$ labeled as *null*
4. scan $D^1 - D^2$ and add each transaction to $FP\text{-}tree^1$,
   omitting items not frequent for dmq_1
5. scan $D^1 \cap D^2$ and add each transaction to both $FP\text{-}tree^1$ and $FP\text{-}tree^2$,
   omitting items not frequent for dmq_1 and dmq_2 respectively
6. scan $D^2 - D^1$ and add each transaction to $FP\text{-}tree^2$,
   omitting items not frequent for dmq_2
7. call $FP\text{-}growth(FP\text{-}tree^1, null, mincount^1)$
8. call $FP\text{-}growth(FP\text{-}tree^2, null, mincount^2)$

Fig. 4. Common Building method

Common Building does not explicitly consider pattern constraints, which are an important elements of frequent pattern queries, and had to be considered by Mine Merge (when generating intermediate queries). This is due to the fact that constraints are taken into account by FP-growth when sorting the frequent elements from a transaction before adding it to an FP-tree and within the recursive FP-growth procedure. With Common Building, these operations are performed independently for each query, and therefore the constraints can be handled as described in [18].

Common Building as an adaptation of Common Counting inherits not only its advantages but also its disadvantage, which is the need for maintaining the data structures (FP-trees in case of Common Building) for several queries at the same time in main memory. If fact, this is even a more serious problem for Common Building than it was for original Common Counting for the following two reasons. Firstly, an initial FP-tree serving as a compressed and compact representation of the source dataset is not the only memory structure used by FP-growth. The recursively called FP-growth procedure builds conditional FP-trees, which especially in early calls require significant amounts of main memory. Secondly, if datasets are sparse then the FP-tree structure does not offer significant compression and storing initial FP-trees for several queries simultaneously in main memory may be infeasible. To address the above problem, in the next section we propose a novel, memory-saving strategy for concurrent processing of frequent itemset queries using FP-growth.

5 Common FP-tree: Integration of Queries' FP-trees Into One Data Structure

Common Building builds a separate initial FP-tree for each query. If data distribution is uniform and/or the queries' datasets significantly overlap, FP-trees built by Common Building will have a significant number of paths in common. Motivated by this observation, we propose a new strategy, named Common FP-tree, aiming at integration of FP-trees of several queries into one data structure, and thus reducing memory consumption.

The basic idea is to extend the FP-tree structure so that instead of just one counter, each tree node will contain a vector of counters – one per frequent itemset query. We will call this extended FP-tree *CFP-tree*. CFP-tree must contain all the information needed for answering all the frequent itemset queries whose datasets its represents. In order to guarantee that, when storing a transaction in CFP-tree, items frequent in *any* of the queries referring to this transaction (referred to as *locally frequent*) have to be preserved. However, for each tree node the counter of a given query is incremented only provided that both following conditions are fulfilled: (1) the item represented by the node is frequent for the query *and* (2) the query refers to the transaction being processed. If a new node is introduced to the tree, counters of the queries for which the above two conditions hold are set to 1 and the remaining counters are set to 0.

One remaining implementation detail regarding CFP-tree is the ordering of items. In general, the supports of items can be different for different queries and therefore finding an order-preserving frequency descending order for locally frequent items for all the queries is not possible. As a sensible compromise, we propose to use global frequency descending order when storing a transaction in a CFP-tree. These global supports can be counted in the same database scan as local item supports for the queries (when counting these global supports only parts of the database relevant for at least one query are considered).

Table 2. Example transaction database

TID	Items	Ordered relevant locally frequent items
100	a;c;d;f;g;i;m;p	f;c;a;m;p
200	a;b;c;f;l;m;o	f;c;a;b;m
300	b;f;h;j;o	f;b
400	B;f;k	f;b
500	b;c;k;s;p	c;b;p;s
600	a;c;e;f;l;m;n;p	f;c;a;m;p
700	c;f;m;p;s	f;c;p;s
800	a;c;f;s	f;c;s

To illustrate the structure of CFP-tree let us consider an example database represented by the first two columns of Table 2 (which will be referred to by the queries as relation R_1) and two frequent itemset queries $dmq_1 = (R_1, \text{"Items"}, \text{"}100 \leq TID \leq 600\text{"}, \text{"}\varnothing\text{"}, 40\%)$ and $dmq_2 = (R_1, \text{"Items"}, \text{"}400 \leq TID \leq 800\text{"}, \text{"}\varnothing\text{"}, 50\%)$. The first query refers to the first six transactions, the second – to the last five. Three transactions are

shared by the queries. For both queries an item (and any itemset) is frequent if it is contained in at least three transactions.

In the first scan of the database frequent items for dmq_1 and dmq_2 are discovered and global supports of all the items are registered. The frequent items for dmq_1 are {a, b, c, f, m, p} and for dmq_2: {c, f, p, s}. The global item supports are used to descendingly order the list containing all items frequent for at least one query[1]. In our case: <(f:7), (c:6), (a:4), (b:4), (m:4), (p:4), (s:3)>. This list will be used to sort transactions before storing them in the CFP-tree. The third column of Table 2 shows the form in which each transaction will be inserted into the CFP-tree. For example, from transaction 500, which belongs to the datasets of both queries, items frequent for at least one query are preserved, while for transaction 800, which is referred only by the second query, only its frequent items are preserved.

The resulting CFP-tree for the database from Table 2 and the two example queries is depicted in Fig. 5. Note that, as explained earlier, some of the counters have the value of 0, which means that either a given item is not frequent for a given query or a given path in the tree represents only transactions that do not belong to the source dataset of a given query. For instance, the rightmost branch of the CFP-tree represents only transaction 500. The transaction belongs to the datasets of both considered queries, so items frequent for any of them are preserved and ordered according to descending global supports: <*c*, *b*, *p*, *s*>. However, since *b* and *s* are frequent only for one of the queries, only one of the counters in their nodes on the path is non-zero.

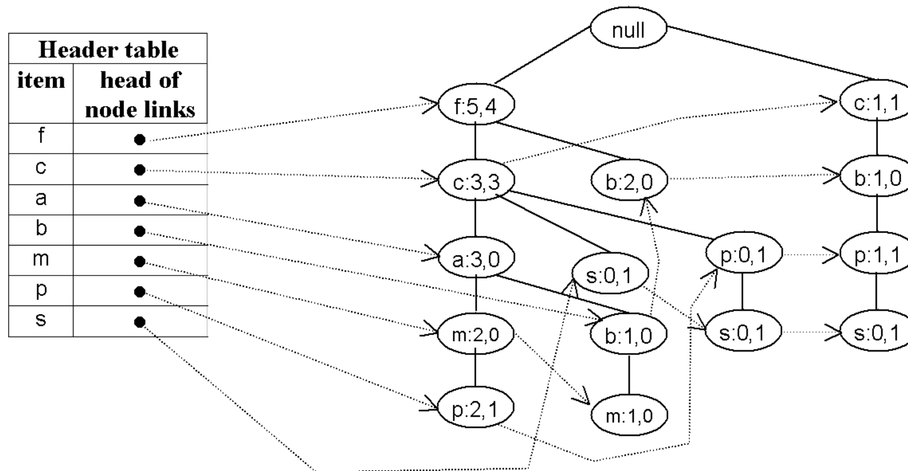

Fig. 5. CFP-tree for an example database and two queries

The Common FP-tree method for two concurrent queries: dmq_1 and dmq_2 is formally presented in Fig. 6. Similarly as with Common Building, generalization of the procedure for an arbitrary number of queries is straightforward.

---

[1] If two or more items have equal global support, they can be ordered arbitrarily. However, this order has to be fixed and used for all the transactions.

Steps 1-6 represent the process of building the CFP-tree structure, described earlier in detail. In steps 7 and 8 actual in-memory mining is performed for the two queries sequentially. In the first call to the FP-growth procedure the FP-tree of a given query, "embedded" in the CFP-tree structure has to be "logically extracted". This logical extraction, for brevity represented in the algorithm as a call to *extractFPtree* function, is performed on-line, while traversing the tree, according to the following set of rules:

a) for query dmq_i, only *i*-th counters in tree nodes are considered;
b) when analyzing a path in a tree, nodes whose counters are 0 are ignored, but their descendants are considered;
c) when using the header table for projections, the items infrequent for a given query are omitted;
d) when following the list connecting all the nodes representing the same item (starting from the header table), nodes whose counters are 0 are ignored, but the traversal from such nodes continues.

The above rules are applied only in the first call to FP-growth as conditional FP-trees passed to further recursive calls are classic FP-tree structures.

1. scan D to calculate $mincount^1$ and $mincount^2$, discover frequent items for dmq_1 and dmq_2, and count global support of the locally frequent itemsets
2. create the root of *CFP-tree* labeled as *null*
3. scan $D^1 - D^2$ and add each transaction to *CFP-tree*, omitting items not frequent for dmq_1
5. scan $D^1 \cap D^2$ and add each transaction to *CFP-tree*, omitting items not frequent for both dmq_1 and dmq_2
6. scan $D^2 - D^1$ and add each transaction to *CFP-tree*, omitting items not frequent in for dmq_2
7. call *FP-growth*(*extractFPtree*(dmq_1, *CFP-tree*), *null*, $mincount^1$)
8. call *FP-growth*(*extractFPtree*(dmq_2, *CFP-tree*), *null*, $mincount^2$)

Fig. 6. Common FP-tree method

Similarly to Common Building, Common FP-tree performs exactly two scans of the database for the whole batch of queries, reading parts shared by the queries once per scan. Common FP-tree also does not rely on the presence of efficient access paths to dataset partitions corresponding to selection predicates for its efficiency, and is more advantageous over sequential processing if full scans are required.

As for constraint handling, Common FP-tree has one drawback compared to Common Building. Handling convertible constraints, which require specific ordering of items before storing a transaction in a tree, is possible only for one of the concurrently processed queries, due to the fact that the same fixed order has to be used by all the queries[2]. This problem definitely can be a subject of further study.

[2] Unless, of course, two or more queries would benefit from the same ordering.

6 Experimental Results

In order to evaluate performance of Mine Merge using FP-growth, Common Building, and Common FP-tree we performed several experiments using synthetic datasets generated with GEN [2]. The datasets were stored in flat files on a disk. The transactions forming a dataset were ordered according to the transaction identifier. The dataset selection predicates had a form of range predicates on transaction identifiers. To facilitate access to database partitions determined by overlapping between queries' datasets, the data files were accompanied with simple sequential indexes. The experiments were conducted on a PC with Intel Pentium M 1,6 GHz processor and 1024 MB of main memory, running Microsoft Windows XP.

In the experiments we varied the minimum support threshold and the overlapping between the queries' datasets. Although neither of the methods requires this, in all the experiments all the queries to be concurrently processed used the same support threshold, so as to make the potential influence of the support threshold easier to observe.

In the first series of experiments we used a small dataset (denoted as GEN1) generated using the following parameters: number of transactions = 50000, number of different items = 1000, average number of items in a transaction = 5, number of patterns = 500, average pattern length = 3. The size of this dataset was 2.5 MB. Figure 7 presents the execution times for Mine Merge using FP-growth (MM), Common Building (CB), Common FP-tree (CT), and sequential processing using FP-growth (SP) of two queries for minimum support thresholds of 1% and 2% respectively. The thresholds where experimentally selected so that they resulted in significantly different sizes of FP-trees (on average by the factor of 40). For both values of the support threshold the level of overlapping varied from 0% to 100%.

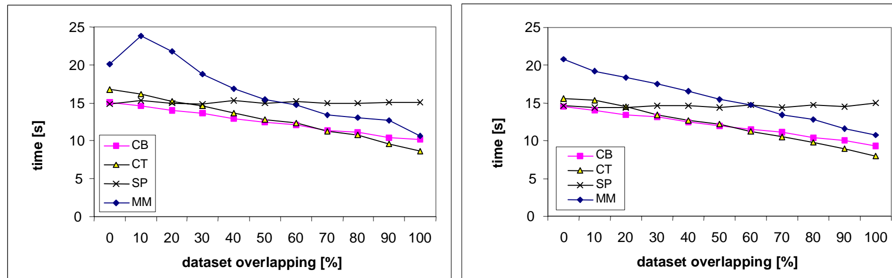

Fig. 7. Execution times on the GEN1 dataset for Mine Merge (MM), Common Building(CB), Common FP-tree (CT), and sequential processing (SP) for 2 overlapping queries with minsup=1% (left) and minsup=2% (right)

The experiments show that Common Building reduces the overall processing time if any overlapping between queries' datasets occurs (the same was true for Apriori as reported in [22]). However, Mine Merge to outperform sequential processing with FP-growth required the overlapping of about 60%, and still was beaten by Common Building and Common FP-tree in each tested case. Execution time of Common FP-tree was shorter than that of Common Building if the overlapping between the

queries' datasets was greater than about 50%. The different support threshold values did not significantly influence the relative performance of the compared methods.

Comparing the above results with the ones reported for concurrent processing of frequent itemset queries using Apriori in [24], we observe that using FP-growth, Mine Merge requires much more significant overlapping between the queries and exhibits worse relative performance to Common Building than to Common Counting in case of Apriori. This can be explained by the fact that FP-growth uses only 2 database scans, typically much fewer then Apriori, and therefore for FP-growth Mine Merge needs more I/O reduction during the integrated scans to compensate the extra scan of database that it performs after collecting results of intermediate queries.

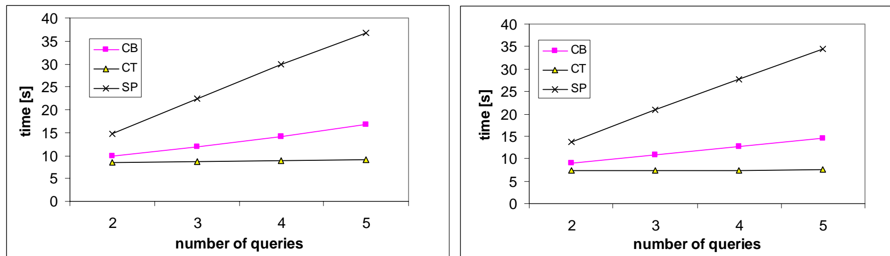

Fig. 8. Execution times on the GEN1 dataset for Common Building(CB), Common FP-tree (CT), and sequential processing (SP) for 2-5 identical queries with minsup=1% (left) and minsup=2% (right)

We also experimented with sets containing more than two queries using Common Building, Common FP-tree, and sequential processing. Mine Merge was excluded from these tests as it was found to be clearly the worst strategy for sets of two queries, and our theoretical analysis (Sect. 3.2) indicated that it is not suitable for large batches of queries. In general, it is hard to compare the performance of our methods for different numbers of queries in a batch because the more queries the more overlapping configurations possible. Therefore, in order to assess the influence of the number of queries on their performance we "benchmarked" the methods on sets of identical queries. Figure 8 shows the execution times for the batches of 2 to 5 queries and support thresholds of 1% and 2%. The results indicate that the greater the number of queries the bigger advantage of Common Building and Common FP-tree over sequential processing. This is due to the fact that the more queries, the greater relative reduction of I/O. The execution time of Common FP-tree stays almost constant with the increase of the number of identical queries as its tree structure stays the same and the time required to handle additional node counters is negligible.

Apart from measuring processing times of the tested methods, we also investigated main memory consumption by the two most efficient methods: Common Building and Common FP-tree[3]. For these two methods, Figure 9 shows the number of tree nodes[4]

[3] Note that Mine Merge does not introduce any specific memory management issues compared to sequential processing as it uses unmodified FP-tree structure and by processing intermediate queries sequentially never needs to maintain FP-trees of more than one query at the same time.

for different levels of overlapping and support thresholds of 1% and 2% respectively. The values for Common Building are sums of the number of nodes for both queries[5]. The experiments show that Common FP-tree requires significantly less memory than Common Building, and as expected memory savings increase with the level of overlapping.

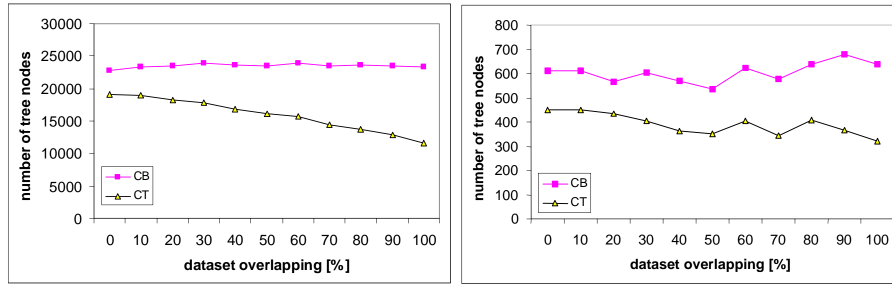

Fig. 9. Number of tree nodes for Common Building (CB) and Common FP-tree (CT) for 2 overlapping queries on GEN1 with minsup=1% (left) and minsup=2% (right)

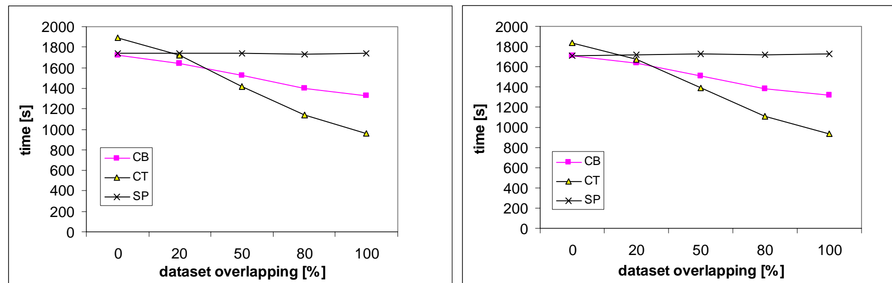

Fig. 10. Execution times on the GEN2 dataset for Common Building(CB), Common FP-tree (CT), and sequential processing (SP) for 2 overlapping queries with minsup=0.9% (left) and minsup=1.05% (right)

In the second series of experiments, aiming at testing scalability of the proposed methods, we used a significantly larger and more dense dataset (denoted as GEN2) generated using the following parameters: number of transactions = 2500000, number of different items = 10000, average number of items in a transaction = 8, number of patterns = 1500, average pattern length = 4. The size of this dataset was 260 MB. Figure 10 presents the execution times for the two most promising methods: Common Building (CB) and Common FP-tree (CT), compared to the execution times of sequential processing using FP-growth (SP) for two queries and minimum support

---

[4] For the case of two queries comparing the numbers of tree nodes provides a satisfactory approximation of the relation between the actual tree sizes as in that case the nodes consist mostly of pointers with one extra integer counter per node in case of Common FP-tree.

[5] The number of nodes measured for Common Building was not constant due to the fact that in our experiments changing the level of overlapping resulted in different parts of the generated dataset being mined and the items were not uniformly distributed.

thresholds of 0.9% and 1.05% respectively. For both values of the support threshold the level of overlapping varied from 0% to 100%. The thresholds where again experimentally selected so that they resulted in significantly different sizes of FP-trees (on average by the factor of 4). The numbers of tree nodes for Common Building (CB) and Common FP-tree (CT) are shown in Fig. 11. The results obtained for the large GEN2 dataset are consistent with the ones on the small GEN1 dataset both in terms of relative execution times and sizes of tree structures.

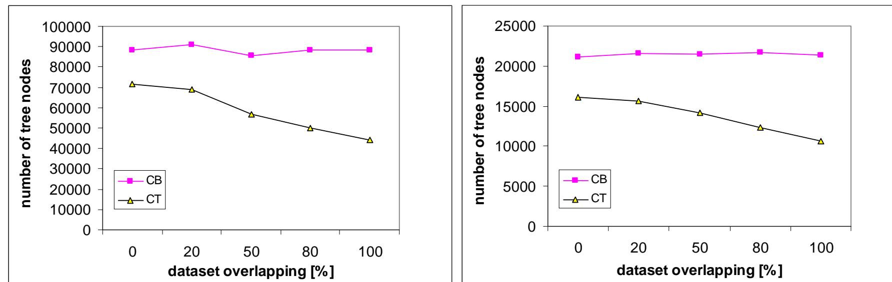

Fig. 11. Number of tree nodes for Common Building (CB) and Common FP-tree (CT) for 2 overlapping queries on GEN2 with minsup=0.9% (left) and minsup=1.05% (right)

7 Conclusions

We have addressed the problem of concurrent processing of frequent itemsets queries. While previous studies analyzed this problem only in the context of the Apriori algorithm, in this paper we focused on FP-growth, which represents a newer, pattern-growth family of data mining algorithms. We considered and experimentally evaluated three multiple-query processing strategies for FP-growth. The first was Mine Merge, originally proposed for Apriori, consisting in transforming the original set of queries into the set of intermediate queries on non-overlapping datasets. The second, inspired by Common Counting for Apriori, was based on integration of dataset scans performed by the queries on shared parts of the database, and was formulated for FP-growth as the Common Building method. The third was a completely new strategy, aiming at integrating memory structures used by the queries, and was implemented in the context of FP-growth as the Common FP-tree method.

The experiments show that Common Building reduces the overall processing time compared to sequential processing if any overlapping between queries' datasets occurs (the same was true for Apriori Common Counting). On the other hand, Mine Merge to be successful with FP-growth requires much more significant overlapping between the queries than in case of Apriori. Finally, the novel strategy, applied by Common FP-tree, outperformed Common Building if queries' datasets overlapped by more than 30% to 50% depending on the nature of the dataset, and in all cases had smaller memory requirements, which makes it an optimal solution for highly overlapping queries and environments with limited memory. For queries that do not overlap significantly, Common Building is more appropriate.

For each of the proposed methods we analyzed the influence of the presence of efficient access paths to queries' source datasets and briefly discussed the possibility of integrating pattern constraints into the mining process. Handling pattern constraints within Mine Merge and Common Building is trivial but their incorporation into Common FP-tree leaves some open questions for future research.

Another direction for further research, which we are currently investigating, is concurrent processing of frequent itemset queries using Apriori by integrating candidate hash-trees of the queries, resulting in a method analogous to Common FP-tree for FP-growth. Finally, we also plan to investigate further possibilities of computation sharing between the concurrently processed queries, going beyond sharing disk accesses and memory data structures.

References

1. Agrawal R., Imielinski T., Swami A: Mining Association Rules Between Sets of Items in Large Databases. Proc. of the 1993 ACM SIGMOD Conf. on Management of Data (1993)
2. Agrawal R., Mehta M., Shafer J., Srikant R., Arning A., Bollinger T.: The Quest Data Mining System. Proc. of the 2nd Int'l Conference on Knowledge Discovery in Databases and Data Mining (1996)
3. Agrawal R., Srikant R.: Fast Algorithms for Mining Association Rules. Proc. of the 20th Int'l Conf. on Very Large Data Bases (1994)
4. Alsabbagh J.R., Raghavan V.V.: Analysis of common subexpression exploitation models in multiple-query processing. Proc. of the 10th ICDE Conference (1994)
5. Baralis E., Psaila G.: Incremental Refinement of Mining Queries. Proceedings of the 1st DaWaK Conference (1999)
6. Blockeel H., Dehaspe L., Demoen B., Janssens G., Ramon J., Vandecasteele H.: Improving the Efficiency of Inductive Logic Programming Through the Use of Query Packs, Journal of Artificial Intelligence Research, Vol. 16 (2002)
7. Cheung D.W., Han J., Ng V., Wong C.Y.: Maintenance of Discovered Association Rules in Large Databases: An Incremental Updating Technique. Proc. of the 12th ICDE (1996)
8. Han J., Pei J.: Mining Frequent Patterns by Pattern-Growth: Methodology and Implications. SIGKDD Explorations, December 2000 (2000)
9. Han J., Pei J., Yin Y.: Mining frequent patterns without candidate generation. Proc. of the 2000 ACM SIGMOD Conf. on Management of Data (2000)
10. Han J., Pei J., Yin Y., Mao R.: Mining Frequent Patterns without Candidate Generation: A Frequent-pattern Tree Approach. Data Mining and Knowledge Discovery: An International Journal, Vol. 8, Issue 1 (2004)
11. Imielinski T., Mannila H.: A Database Perspective on Knowledge Discovery. Communications of the ACM, Vol. 39, No. 11 (1996)
12. Jarke M.: Common subexpression isolation in multiple query optimization. Query Processing in Database Systems, Kim W., Reiner D.S. (Eds.), Springer (1985)
13. Jeudy B., Boulicaut J-F.: Using condensed representations for interactive association rule mining. Proceedings of the 6th European Conference on Principles and Practice of Knowledge Discovery in Databases (2002)
14. Jin R., Sinha K., Agrawal G.: Simultaneous Optimization of Complex Mining Tasks with a Knowledgeable Cache. Proc. of the 11th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (2005)
15. Meo R.: Optimization of a Language for Data Mining. Proc. of the ACM Symposium on Applied Computing - Data Mining Track (2003)

16. Morzy T., Wojciechowski M., Zakrzewicz M.: Materialized Data Mining Views. Proceedings of the 4th PKDD Conference (2000)
17. Nag B., Deshpande P.M., DeWitt D.J.: Using a Knowledge Cache for Interactive Discovery of Association Rules. Proc. of the 5th KDD Conference (1999)
18. Pei J., Han J.: Can We Push More Constraints into Frequent Pattern Mining?. Proceedings of the 6th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (2000)
19. Roy P., Seshadri S., Sundarshan S., Bhobe S.: Efficient and Extensible Algorithms for Multi Query Optimization. ACM SIGMOD Intl. Conference on Management of Data (2000)
20. Savasere A., Omiecinski E., Navathe S.: An Efficient Algorithm for Mining Association Rules in Large Databases. Proc. 21th Int'l Conf. Very Large Data Bases (1995)
21. Sellis T.: Multiple-query optimization. ACM Transactions on Database Systems, Vol. 13, No. 1 (1988)
22. Wojciechowski M., Zakrzewicz M.: Evaluation of Common Counting Method for Concurrent Data Mining Queries. Proc. of the 7th ADBIS Conference (2003)
23. Wojciechowski M., Zakrzewicz M.: Data Mining Query Scheduling for Apriori Common Counting. Proc. of the Sixth International Baltic Conference on Databases and Information Systems (2004)
24. Wojciechowski M., Zakrzewicz M.: Evaluation of the Mine Merge Method for Data Mining Query Processing. Proc. of the 8th ADBIS Conference (2004)
25. Wojciechowski M., Zakrzewicz M.: On Multiple Query Optimization in Data Mining. Proc. of the 9th Pacific-Asia Conference on Knowledge Discovery and Data Mining (2005)