Concurrent Processing of Frequent Itemset Queries Using FP-Growth Algorithm

Marek Wojciechowski, Krzysztof Galecki, Krzysztof Gawronek

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

Abstract. Discovery of frequent itemsets is a very important data mining problem with numerous applications. Frequent itemset mining is often regarded as advanced querying where a user specifies the source dataset and pattern constraints using a given constraint model. A significant amount of research on frequent itemset mining has been done so far, focusing mainly on developing faster complete mining algorithms, efficient constraint handling, and reusing results of previous queries. Recently, a new problem of optimizing processing of batches of frequent itemset queries has been considered and two multiple query optimization techniques for frequent itemset queries: Common Counting and Mine Merge have been proposed. Mine Merge does not depend on a particular mining algorithm, while Common Counting has been specifically designed to work with Apriori. Nevertheless, in previous works the efficiency of Mine Merge was tested only on Apriori, and it is unclear how it would perform with newer pattern-growth algorithms like FP-growth. In this paper we adapt the Common Counting method to work with FP-growth and evaluate efficiency of both methods when FP-growth is used as a basic mining algorithm.

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 and pattern-growth methods. Apriori-like solutions, represented by a 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 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 sets 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], which transforms a database into FP-tree stored in main memory using just 2 database scans, and then performs mining on that optimized FP-tree structure.

Frequent pattern mining is often regarded as advanced querying where a user specifies the source dataset, the minimum support threshold, and optionally pattern constraints within a given constraint model [10]. A significant amount of research on efficient processing of frequent pattern queries has been done in recent years, focusing mainly on constraint handling and reusing results of previous queries in the context of frequent itemsets and sequential patterns.

In this paper we consider the problem of optimizing batches of frequent itemset queries. In previous works, two multiple query optimization techniques for frequent itemset queries: Common Counting [19] and Mine Merge [21] have been proposed. Mine Merge does not depend on a particular mining algorithm, while Common Counting has been specifically designed to work with Apriori. Adaptation of Common Counting to other mining paradigms has not been considered so far. Similarly, although Mine Merge works with any frequent itemset mining algorithm, its efficiency has been evaluated and reported only for Apriori, and it is unclear how it would perform with newer pattern-growth algorithms like FP-growth. In this paper we adapt the Common Counting method to work with FP-growth and evaluate efficiency of both methods when FP-growth is used as a basic mining algorithm.

1.1 Related Work

Multiple-query optimization has been extensively studied in the context of database systems (see [18] 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][11][16]. 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, the only two multiple-query processing methods for data mining queries are Common Counting and Mine Merge, mentioned above. Recently, 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 [15], and reusing materialized complete [5][13][14] or condensed [12] results of previous queries provided that syntactic differences between the queries satisfy certain conditions.

1.2 Organization of the paper

The paper is organized as follows. In Section 2 we review basic definitions regarding frequent itemset mining and we briefly describe the FP-growth algorithm. Section 3 contains basic definitions regarding frequent pattern queries and presents the Common Counting and Mine Merge multiple-query optimization techniques. In Section 4 we discuss the method of incorporating the idea of Common Counting into the pattern-growth mining paradigm, represented by the FP-growth algorithm. Section 5 presents experimental results. Section 6 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 above 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 (a scalable, disk-based version of FP-tree has also been proposed). 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. The procedure of creating an FP-tree requires two database scans: one to discover frequent items and their counts, and second to build the tree by adding transactions to it one by one.

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 (conditional FP-trees) of the tree with respect to frequent prefixes found so far. 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. 1, together with its initial tree-building phase (the details can be found in [9]).

```
Input: database $D$, minimum support threshold $minsup$
Output: the complete set of frequent patterns
Method:
1.  scan $D$ to discover frequent items and their counts
2.  create the root of $FP\text{-}tree$ labeled as $null$
3.  scan $D$ and add each transaction to $FP\text{-}tree$ (omitting non-frequent items)
4.  call $FP\text{-}growth(FP\text{-}tree, null)$

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

Fig. 1. FP-growth algorithm

3 Multiple Query Optimization for Frequent Itemset Queries

In this section, we review the definitions of a frequent itemset query and an elementary data selection predicate, which are used by the considered multiple-query optimization methods: Common Counting and Mine Merge, and next we describe the methods themselves.

3.1 Frequent itemset query and its predicates

A *frequent itemset query* is a tuple $dmq = (R, a, \Sigma, \Phi, \beta)$, where R is a relation, a is an attribute of R, Σ is a condition involving the attributes of R (called *database predicate*), Φ is a condition involving discovered patterns (called *pattern predicate*), β is the minimum support threshold. The result of the dmq is a set of frequent itemsets (patterns) discovered in $\pi_a \sigma_\Sigma R$, satisfying Φ, and having support $\geq \beta$.

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 pattern 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 $a_1>5$.

3.2 Elementary data selection predicates

The set $S=\{s_1, s_2, ..., s_k\}$ of data selection predicates over the attribute *a* or the relation *R* is a *set of elementary data selection predicates* for a set of frequent itemset queries *DMQ* if for all *i,j* we have $\sigma_{si}R \cap \sigma_{sj}R = \varnothing$ and for each *i* there exist integers *a, b, ..., m* such that $\sigma_{\Sigma i}R=\sigma_{sa}R \cup \sigma_{sb}R \cup .. \cup \sigma_{sm}R$ (example in Fig. 2).

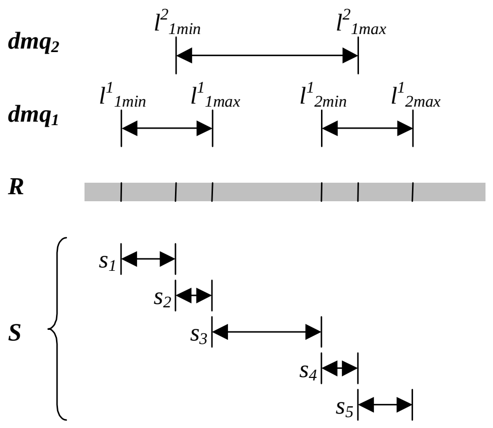


Fig. 2. Example set of frequent itemset queries and their elementary data selection predicates

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

3.3 Common Counting

The Common Counting method was developed for the Apriori algorithm and is based on the observation that when two or more different queries count their candidate itemsets in the same part of the database, only one scan of the common part of the database is required. During that scan, candidates generated by all the queries referring to that part of the database are counted. The Common Counting method in the context of Apriori algorithm for two concurrent queries (denoted as dmq_A and dmq_B) is presented in Fig. 3 (generalization of the algorithm to support more than two frequent itemset queries is straightforward). D^A and D^B denote parts of the database read by dmq_A and dmq_B respectively.

```
$C_1^A$ = {all 1-itemsets from $D^A$}
$C_1^B$ = {all 1-itemsets from $D^B$}
for ($k$=1; $C_k^A \cup C_k^B \neq \emptyset$; $k$++)
        if $C_k^A \neq \emptyset$ count($C_k^A$, $D^A - D^B$);
        if $C_k^B \neq \emptyset$ count($C_k^B$, $D^B - D^A$);
        count($C_k^A \cup C_k^B$, $D^A \cap D^B$);
          $L_k^A = \{c \in C_k^A \mid c.count \geq minsup^A\}$;
          $L_k^B = \{c \in C_k^B \mid c.count \geq minsup^B\}$;
          $C_{k+1}^A$ = generate_candidates($L_k^A$);
          $C_{k+1}^B$ = generate_candidates($L_k^B$);
$Answer(dmq_A) = \bigcup_k L_k^A$;
$Answer(dmq_B) = \bigcup_k L_k^B$;
```

Fig. 3. Apriori Common Counting method

During the integrated counting, the candidates from all the frequent itemset queries are loaded into the main memory. Next, the database is scanned and for each itemset from the database the supported candidate counters for all the relevant queries are incremented. 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 [20][22].

3.4 Mine Merge

Mine Merge employs the property that for a database divided into a set of disjoint partitions, an itemset which is frequent in a whole database, must also be frequent in at least one partition of it [17]. Mine Merge first generates a set of *intermediate data mining queries*, in which each data mining query is based on a single elementary selection predicate only. The intermediate data mining queries are derived from those original data mining queries that are sharing a given elementary selection predicate. Next, the intermediate data mining queries are executed sequentially using any frequent itemset mining algorithm and then their results are merged to form global candidates for the original data mining queries. Finally, a database scan is performed to count the global candidate supports and to answer the original data mining queries. The pseudocode of the Mine Merge algorithm is shown in Fig. 4.

```
/* Generate intermediate data mining queries $IDMQ = \{idmq_1, idmq_2, ...\}$ */
    $IDMQ \leftarrow \varnothing$
    for each $s_j \in S$ do begin
      $Q \leftarrow \{dmq_i \in DMQ \mid (dmq_i, s_j) \in E\}$
      $intermediate\_\beta \leftarrow min\{\beta_i \mid dmq_i = (R, a, s_i, \Phi_i, \beta_i) \in Q\}$
      $intermediate\_\Phi \leftarrow \Phi_1 \vee \Phi_2 \vee ... \vee \Phi_{|Q|}, \forall i=1..|Q|, dmq_i = (R, a, s_i, \Phi_i, \beta_i) \in Q$
      $IDMQ \leftarrow IDMQ \cup idmq_j = (R, a, s_j, intermediate\_\Phi, intermediate\_\beta)$
    end
/* Execute intermediate data mining queries */
    for each $idmq_i \in IDMQ$ do
      $IF_i \leftarrow execute(idmq_i)$
/* Generate results for original data mining queries $DMQ = \{dmq_1, dmq_2, ...\}$ */
    for each $dmq_i \in DMQ$ do
      $C^i \leftarrow \{c \in \bigcup_k IF_k, (dmq_i, s_k) \in E, c.count \geq \beta_i\}$
    for each $s_j \in S$ do begin
        $CC \leftarrow \bigcup C^l : (dmq_l, s_j) \in E;$          /* select the candidates to count now */
        if $CC \neq \varnothing$ then $count(CC, \sigma_{sj} R)$;
    end
    for ($i$=1; $i$<=$n$; $i$++) do
      $Answer^i \leftarrow \{c \in C^i \mid c.count \geq \beta_i\}$   /* generate responses */
```

Fig. 4. Mine Merge method

4 Mine Merge and Common Counting for FP-growth

Mine Merge can be applied to FP-growth without modifications, as it is independent of the mining algorithm used to execute intermediate queries. However, its applicability to algorithms other than Apriori has to be evaluated. This is definitely true for FP-growth, representing a pattern-growth family of algorithms, fundamentally different from Apriori. FP-growth requires only exactly two database scans, which makes it difficult for Mine Merge to compensate the cost of its extra database scan with the reduction of I/O thanks to query overlapping.

On the other hand, Common Counting as formulated in [19] for Apriori cannot be applied directly to FP-growth because FP-growth does not perform candidate counting. However, we can exploit the general idea 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, Common Counting in the context of FP-growth consists in concurrent building of FP-trees in main memory for a batch of queries. The Common Counting method in the context of FP-growth for two concurrent queries: dmq_A and dmq_B can be formalized as presented in Fig. 5. Common Counting takes place only during the tree-building step, the FP-growth recursive procedure is not affected by Common Counting.

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

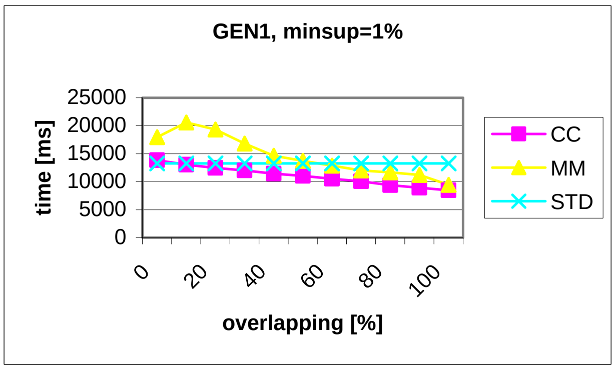
Fig. 5. Common Counting with FP-growth

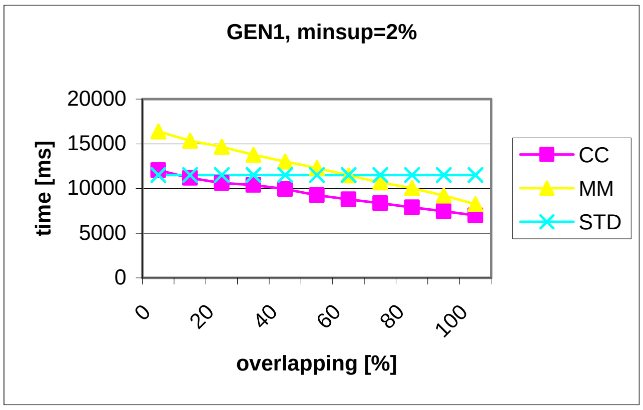
5 Experimental Results

In order to evaluate performance of Common Counting and Mine Merge using FP-growth as a basic mining algorithm, we performed several experiments using two synthetic datasets generated with GEN [2]. The first dataset (denoted as GEN1) contained 50000 transactions, 1000 different items, and the average number of items in a transaction was 5. In the second dataset (denoted as GEN2) there were 50000 transactions built from 10000 different items, and the average number of items in a transaction was also 5. The experiments were conducted on a PC with AMD Athlon 2200+ 1.8 GHz processor and 256 MB of main memory. The datasets used in all experiments resided in flat files on a local disk.

In the experiments we varied the minimum support threshold and the overlapping between the queries. Figures 6 and 7 present the execution times for Mine Merge (MM), Common Counting (CC), and sequential processing (STD) of two queries using FP-growth, for datasets GEN1 and GEN2 respectively. For both datasets the level of overlapping varied from 0% to 100%, and three values of support threshold have been used. We also experimented with sets of three queries, obtaining consistent results.

The experiments show that Common Counting for FP-growth reduces the overall processing time if any overlapping between queries' datasets occurs (the same was true for Apriori as reported in [19]). However, Mine Merge to outperform sequential processing with FP-growth required the overlapping of at least about 70%, and still was beaten by Common Counting in each tested case. Comparing these results with the ones reported for Apriori in [21], we observe that using FP-growth, Mine Merge requires much more significant overlapping between the queries and exhibits worse relative performance to Common Counting than in case of Apriori. This can be explained by the fact that FP-growth uses only 2 database scans, which is much fewer then in typical scenarios with 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.





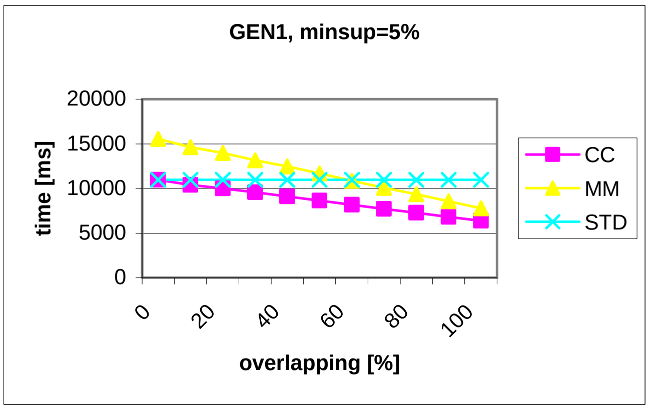
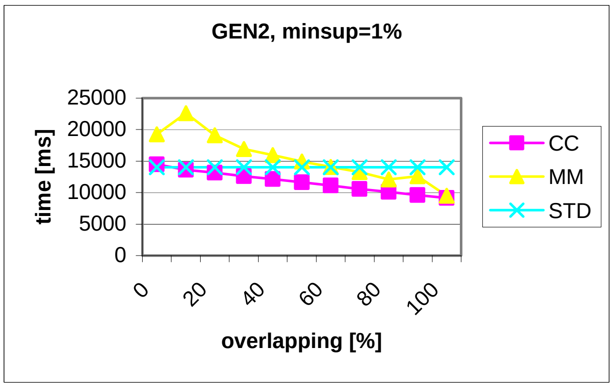
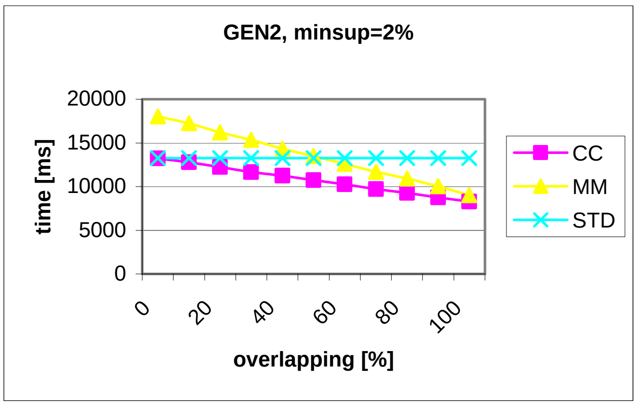


Fig. 6. Execution times for Common Counting, Mine Merge, and sequential processing, using FP-growth for 2 overlapping queries (dataset GEN1)





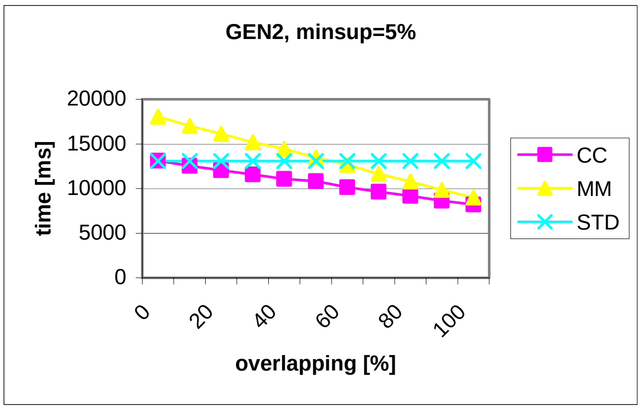


Fig. 7. Execution times for Common Counting, Mine Merge, and sequential processing, using FP-growth for 2 overlapping queries (dataset GEN2)

6 Conclusions

In this paper we addressed the problem of multiple data mining query optimization for frequent itemset queries using Common Counting and Mine Merge methods. The methods reduce the I/O cost for a batch of queries by identifying common execution tasks and executing them only once for the whole batch.

The contributions of this paper are: (1) adaptation of Common Counting, originally designed for Apriori, to work with FP-growth, and (2) experimental evaluation of Common Counting and Mine Merge using FP-growth as a basic mining algorithm. We have shown that the general idea of Common Counting, which is integration of operations performed by a set of queries during the scan of the common part of the dataset, can be carried over to FP-growth, however the implementation details are different for Apriori and FP-growth.

The experiments show that Common Counting for FP-growth reduces the overall processing time if any overlapping between queries' datasets occurs (the same was true for Apriori). 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.

Since the size of memory structures used by FP-growth may limit the possibility of applying Common Counting in practice, and efficiency of Mine Merge for FP-growth is not satisfactory, there is definitely a need for novel multiple-query optimization techniques for pattern-growth methods, which will be the subject of our future research. Another interesting direction of future research may be multiple data mining query optimization in case of condensed representations of frequent itemsets.

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. Imielinski T., Mannila H.: A Database Perspective on Knowledge Discovery. Communications of the ACM, Vol. 39, No. 11 (1996)
11. Jarke M.: Common subexpression isolation in multiple query optimization. Query Processing in Database Systems, Kim W., Reiner D.S. (Eds.), Springer (1985)
12. 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)
13. Meo R.: Optimization of a Language for Data Mining. Proc. of the ACM Symposium on Applied Computing - Data Mining Track (2003)
14. Morzy T., Wojciechowski M., Zakrzewicz M.: Materialized Data Mining Views. Proceedings of the 4th PKDD Conference (2000)
15. 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)
16. 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)
17. 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)
18. Sellis T.: Multiple-query optimization. ACM Transactions on Database Systems, Vol. 13, No. 1 (1988)
19. Wojciechowski M., Zakrzewicz M.: Evaluation of Common Counting Method for Concurrent Data Mining Queries. Proc. of the 7th ADBIS Conference (2003)
20. 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)
21. Wojciechowski M., Zakrzewicz M.: Evaluation of the Mine Merge Method for Data Mining Query Processing. Proc. of the 8th ADBIS Conference (2004)
22. 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)