# Evaluation of Common Counting Method for Concurrent Data Mining Queries[*]

Marek Wojciechowski, Maciej Zakrzewicz

Poznan University of Technology
Institute of Computing Science
ul. Piotrowo 3a, 60-965 Poznan, Poland
{marek,mzakrz}@cs.put.poznan.pl

**Abstract.** Data mining queries are often submitted concurrently to the data mining system. The data mining system should take advantage of overlapping of the mined datasets. In this paper we focus on frequent itemset mining and we discuss and experimentally evaluate the implementation of the Common Counting method on top of the Apriori algorithm. The general idea of Common Counting is to reduce the number of times the common parts of the source datasets are scanned during the processing of the set of frequent pattern queries.

## 1 Introduction

Data mining, also referred to as database mining or knowledge discovery in databases (KDD), aims at discovery of useful patterns from large databases or warehouses. Nowadays we are witnessing the evolution of data mining environments from specialized tools to multi-purpose data mining systems offering some level of integration with existing database management systems. Data mining can be seen as advanced querying, where a user specifies the source dataset and the requested pattern constraints, then the system chooses the appropriate data mining algorithm and returns the discovered patterns to the user. One of the most serious problems concerning data mining queries is long response time. Current systems often consume minutes or hours to answer single queries.

In practical applications, data mining queries are often executed during nights, when system activity is low. Sets of queries are scheduled and then automatically evaluated by a data mining system. It is possible that the data mining queries delivered to the system are somehow similar, e.g., their source datasets overlap. Such queries can be executed serially, however, the fact that the datasets overlap may help in parallelizing the query execution.

In [13] we have proposed various methods for batch processing of data mining queries for frequent itemset discovery. In order to improve the overall performance of the batched data mining queries, the methods tried to employ the fact that their source datasets overlapped. One of the methods, called *Mine Merge*, partitioned the set of

[*] This work was partially supported by the grant no. 4T11C01923 from the State Committee for Scientific Research (KBN), Poland.

batched queries into the set of simple isolated queries. Another method, called *Common Counting*, integrated the phases of support counting for candidate itemsets. Both methods resulted in significant database I/O reduction.

In this paper we present our experiences in implementing and evaluating the *Common Counting* method for concurrent data mining queries. We discuss the influence of various parameters on the efficiency of *Common Counting*. Experimental results are given to prove the advantages of the *Common Counting* method in the context of concurrent data mining queries.

## 1.1 Related Work

The problem of mining frequent itemsets and association rules was introduced in [1] and an algorithm called *AIS* was proposed. In [3], two new algorithms were presented, called *Apriori* and *AprioriTid*, that achieved significant improvements over *AIS* and became the core of many new algorithms for mining association rules. *Apriori* and its variants first generate all frequent itemsets (sets of items appearing together in a number of database records meeting the user-specified support threshold) and then use them to generate rules. *Apriori* and its variants rely on the property that an itemset can only be frequent if all of its subsets are frequent.

Since it has been observed that generation of association rules from frequent itemsets is a straightforward task, further research focused on alternative methods for frequent itemset mining. In [14], a different algorithm called *Eclat* has been proposed. *Eclat* generates lists of itemset identifiers to efficiently generate and count frequent itemsets. Recently, a new family of pattern discovery algorithms, called pattern-growth methods (see [7] for a review), has been developed for discovery of frequent itemsets (and other classes of patterns). The methods project databases based on the currently discovered frequent patterns and grow such patterns to longer ones in corresponding projected databases. Pattern-growth methods are supposed to perform better than *Apriori*-like algorithms in case of low minimum support thresholds. Nevertheless, practical studies [10] show that for real datasets *Apriori* and *Eclat* might still be a more efficient solution.

The notion of data mining queries (or *KDD* queries) was introduced in [8]. The need for Knowledge and Data Management Systems (KDDMS) as second generation data mining tools was expressed. The ideas of application programming interfaces and data mining query optimizers were also mentioned. Several data mining query languages that are extensions of *SQL* were proposed [4][6][9][11][12].

## 1.2 Basic Definitions

**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 support threshold.

**Apriori algorithm**. *Apriori* is an example of a level-wise algorithm for frequent itemset discovery. It makes multiple passes over the input data to determine all frequent itemsets. Let $L_k$ denote the set of frequent itemsets of size $k$ and let $C_k$ denote the set of candidate itemsets of size $k$. Before making the $k$-th pass, *Apriori* generates $C_k$ using $L_{k-1}$. Its candidate generation process ensures that all subsets of size $k$-$1$ of $C_k$ are all members of the set $L_{k-1}$. In the $k$-th pass, it then counts the support for all the itemsets in $C_k$. At the end of the pass all itemsets in $C_k$ with a support greater than or equal to the minimum support form the set of frequent itemsets $L_k$. Figure 1 provides the pseudocode for the general level-wise algorithm, and its *Apriori* implementation. The *subset(t, k)* function gives all the subsets of size $k$ in the set $t$.

This method of pruning the $C_k$ set using $L_{k-1}$ results in a much more efficient support counting phase for *Apriori* when compared to the earlier algorithms. In addition, the usage of a hash-tree data structure for storing the candidates provides a very efficient support-counting process.

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

```
$L_1$ = {frequent 1-itemsets}
for ($k$ = 2; $L_{k-1} \neq \emptyset$; $k$++)
        $C_k$ = generate_candidates($L_{k-1}$);
        forall tuples $t \in D$
                $C_t = C_k \cap$ subset($t$, $k$);
                forall candidates $c \in C_t$
                        $c.count$++;
        $L_k = \{c \in C_k \mid c.count \geq minsup\}$
$Answer = \bigcup_k L_k$;
```

**Fig. 1.** A general level-wise algorithm for association discovery (left) and its Apriori implementation (right)

## 2 Preliminaries and Problem Statement

**Data mining query**. A data mining query is a tuple $DMQ = (R, a, \Sigma, \Phi)$, where $R$ is a relation, $a$ is an attribute of $R$, $\Sigma$ is a condition involving the attributes of the relation $R$, $\Phi$ is a condition involving discovered patterns. The result of the data mining query is a set of patterns discovered in $\pi_a \sigma_\Sigma$ and satisfying $\Phi$.

**Example**. Given the relation $R_1$ shown in Fig. 2, the result of the data mining query $DMQ_1$=($R_1$, *"basket", "id>5 AND id<10", "minsup ≥3"*) is shown in Fig. 3.

```
R1: id  basket
    --------
     1  a,b,c
     4  a,c
     6  d,f,g
     7  f,g,k,m
     8  e,f,g
    15  a,f
```

**Fig. 2.** Example relation $R_1$

```
result of DMQ1:

{f}
{g}
{f,g}
```

**Fig. 3.** $DMQ_1$ query result

**Problem statement**. Given a set $S = \{DMQ_1, DMQ_2, \ldots, DMQ_n\}$ of data mining queries, where $DMQ_i = (R, a, \Sigma_i, \Phi_i)$ and $\forall_i \exists_{j \neq i}\ \sigma_{\Sigma i}(R) \cap \sigma_{\Sigma j}(R) \neq \emptyset$, the goal is to minimize the I/O cost and the CPU cost of executing S.

### 2.1 Motivating example

Consider a relation *Sales(uad, basket, time)* to store purchases made by users of an internet shop. For every visit to a shop, we store the user's internet address (*uad*), the products purchased (*basket*) and the time of the visit (*time*). Since datasets of this kind tend to be very large, there is a need for automated analysis of their contents. One of the data mining methods that proved to be useful for such analysis is associations discovery. Assume a shop manager is interested in finding sets of products that were frequently co-occurring in the users' purchases. The shop manager plans to create two reports: one showing the frequent sets that appeared in more than 350 purchases in January 2002 and one showing the frequent sets that appeared in more than 20 purchases made by customers from France. The two data mining queries shown below are required to construct the response.

$DMQ_A$=(*Sales, "basket", "time between '01-01-02' and '01-31-02'", "minsup > 350"*)
$DMQ_B$=(*Sales, "basket", "uad like '%.fr'", "minsup > 20"*)

If the size of the *Sales* relation is very large, each of the above data mining queries can take a significant amount of time to execute. Part of this time will be spent on reading the *Sales* relation from disk in order to count occurrences of candidate itemsets. Notice that the sets of blocks to be read by the two data mining queries may overlap. If we try to merge the processing of the two data mining queries, we can reduce redundancy resulting from this overlapping. In the remaining of this paper we will use this example to illustrate our method.

## 3 Cost Analysis of Level-Wise Algorithms

In order to analytically compare the complexity of the method considered, first we derive the execution cost functions for a generic level-wise algorithm. We make the following assumptions: (1) the size of the database is much larger than the size of all candidate itemsets, (2) the size of all candidate itemsets can be larger than the memory size, and (3) frequent itemsets fit in memory. The notation we use is given in Table 1.

**Table 1.** Notation used in cost models

| | |
|---|---|
| M | main memory size (blocks) |
| $\|D\|$ | number of itemsets in the database |
| $\|\|D\|\|$ | size of the database (blocks) |
| $\|C_i\|$ | number of candidate itemsets for step *i* |
| $\|\|C_i\|\|$ | size of all candidate itemsets for step *i* (blocks), $\|\|C_i\|\|<<\|\|D\|\|$, $\|\|C_i\|\|<M$ |
| $\|L_i\|$ | number of frequent itemsets for step *i*, $\|L_i\|<\|C_i\|$ |
| $\|\|L_i\|\|$ | size of all frequent itemsets for step *i* (blocks), $\|\|L_i\|\|<M$ |

The cost of performing the general level-wise association discovery algorithm is as follows:

1. **Candidate counting-pruning.** Candidate itemsets must be read from disk in portions equal to the available memory size. For each portion, the database must be scanned to join itemsets from $C_i$ with itemsets from $D$. Next, the candidate itemsets with support greater or equal to *minsup* become frequent itemsets and must be written to disk. The I/O cost of a single iteration *i* is the following:

$$\text{cost}_{I/O} = \|C_i\| + \frac{\|C_i\|}{M}\|D\| + \|L_i\|$$

The dominant part of the CPU cost is join condition verification. For the simplicity, we assume the cost of comparing two itemsets does not depend on their sizes and equals 1. Thus, the CPU cost of a single iteration *i* is the following:

$$\text{cost}_{CPU} = |C_i||D|$$

2. **Candidate generation.** Frequent itemsets from the previous iteration must be read from disk, joined in memory, and saved as new candidate itemsets. The I/O cost of a single iteration *i* is the following:

$$\text{cost}_{I/O} = \|L_i\| + \|C_{i+1}\|$$

The CPU cost of this phase of the algorithm is the following:

$$\text{cost}_{CPU} = |L_i||L_i|$$

Therefore, if $K$ is the number of iterations, the overall cost of the level-wise algorithm is as follows:

$$\text{cost}_{I/O} = \sum_{i=1}^{K} \left( \|C_i\| + \frac{\|C_i\|}{M} \|D\| + 2\|L_i\| + \|C_{i+1}\| \right)$$

$$\text{cost}_{CPU} = \sum_{i=1}^{K} \left( |C_{i+1}\|D| + |L_i|^2 \right)$$

## 4 Common Counting Method

When two or more different DMQs count their candidate itemsets in the same part of the database, only one scan of the common part of the database is required and during that scan candidates generated by all the queries referring to that part of the database are counted. For the sake of simplicity we formally present the *Common Counting* method in the context of *Apriori* algorithm for two concurrent DMQs only (Fig. 4). It is straightforward to extend the technique to support more than two DMQs.

```
$C_1^A$ = {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($L_k^B$);
$Answer^A = \bigcup_k L_k^A$;
$Answer^B = \bigcup_k L_k^B$;
```

**Fig. 4.** Model of the Common Counting method

During the integrated counting, the candidates from all the DMQs 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' set does not fit into memory, it is loaded in parts and the database is scanned multiple times (for the combined candidate set we apply the scheme proposed in the context of the original *Apriori* algorithm in [3]). After the candidate supports are calculated, frequent itemsets are derived in a traditional way.

The *Common Counting* method is directly applicable to all level-wise algorithms (e.g., *Apriori*)[1]. The advantage of the method is database scan reduction because the common part of the database must be logically read only once.

Different ways of implementing the candidate counting step are possible. In this paper we propose the following solution. The database is logically divided into disjoint partitions such that each part is a subset of the source dataset for one or more DMQs. In the candidate verification phase the database is scanned sequentially. Before scanning each of the partitions, candidates for all the queries referring to that partition have to be available. If for a given query the candidates have not been generated while processing previous partitions, then they have to be generated. We do not combine the sets of candidates for the collection of queries - for each query a separate candidate hash-tree is built. If the candidates for a given partition do not fit into available memory, they are processed in parts and the database partition is scanned several times (as in the original *Apriori*). After processing a database partition, the candidate hash-trees are swapped to disk only if the system is short of memory. Candidate hash-trees for queries that will not be needed for the highest number of subsequent partitions are first to be swapped.

We see two alternatives to the candidate counting scheme described above based on the idea of combining candidate sets. Firstly, the candidates for all the queries can be combined and stored in one hash-tree with each candidate having several counters (one for each query). Since such a structure will contain candidates that should be counted in some partitions but not considered in others, a list of relevant database partitions should be associated with each candidate itemset. Efficiency of the scheme will likely depend on the number of common candidates between the queries.

Secondly, we can modify the above scheme by combining the sets of candidates of the queries before processing each database partition, considering only the candidates of the relevant queries. This approach minimizes the memory usage but requires several costly operations of building and then destroying hash-trees. Evaluation of the two alternative candidate counting schemes is beyond the scope of this paper and is regarded as a topic for future research.

**Example.** Let us consider the following example of the *Common Counting* method. Using the database selection conditions from the Section 2.1, we construct three separate dataset definitions:

1.
```
select basket
from   sales
where  time between '01-01-02' and '01-31-02'
  and  NOT uad like '%.fr'
```

2.
```
select basket
from   sales
where  time between '01-01-02' and '01-31-02'
  and  uad like '%.fr'
```

[1] It should be noted that Common Counting can also be applied to algorithms that use Apriori-like generate-and-test scheme in one of their phases (e.g., Eclat)

3.
```
select basket
from   sales
where  NOT time between '01-01-02' and '01-31-02'
  and  uad like '%.fr'
```

Next, we scan the first query's result in order to count *DMQ*$^A$ candidate itemsets, then we scan the second query's result in order to count both *DMQ*$^A$ and *DMQ*$^B$ candidate itemsets, finally we scan the third query's result in order to count *DMQ*$^B$ candidate itemsets. Notice that none of the database blocks needed to be read twice, on the condition that candidate itemsets fit in memory.

Let us analyze the cost of the level-wise phase of the *Common Counting* method. Candidate itemsets of *DMQ*$^A$ must be read, joined with $D^A$-$D^B$, counted, and saved to disk. Also, candidate itemsets of *DMQ*$^B$ must be read, joined with $D^B$-$D^A$, counted, and saved to disk. Next, all candidates of *DMQ*$^A$ and *DMQ*$^B$ must be read, joined with $D^A \cap D^B$, counted, and saved to disk. The candidate itemsets with support greater or equal to, respectively, *minsup*$^A$ or *minsup*$^B$, become frequent itemsets and are written to disk. In order to generate new candidate itemsets, all frequent itemsets must be read from disk and new candidate itemsets must be written to disk. Therefore, the I/O cost of this method is the following:

$$\text{cost}_{I/O} = \sum_{i=1}^{\max(K^A,K^B)} \left( \begin{array}{l} 3\|C_i^A\| + \frac{\|C_i^A\|}{M}\|D^A - D^B\| + 3\|C_i^B\| + \frac{\|C_i^B\|}{M}\|D^B - D^A\| \\ + \frac{\|C_i^A\| + \|C_i^B\|}{M}\|D^A \cap D^B\| + 2\|L_i^A\| + 2\|L_i^B\| + \|C_{i+1}^A\| + \|C_{i+1}^B\| \end{array} \right)$$

Similarly, the CPU cost is as follows:

$$\text{cost}_{CPU} = \sum_{i=1}^{K^A}\left(|C_{i+1}^A||D^A - D^B| + |L_i^A|^2\right) + \sum_{i=1}^{K^B}\left(|C_{i+1}^B||D^B - D^A| + |L_i^B|^2\right) +$$
$$\sum_{i=1}^{\max(K^A,K^B)}\left(\left(|C_{i+1}^A| + |C_{i+1}^B|\right)|D^B \cap D^A|\right)$$

## 5 Performance Analysis

In order to evaluate performance of the *Common Counting* method in the context of frequent itemset mining we performed several experiments on synthetic and real datasets. The synthetic datasets were generated by means of the *GEN* generator from the *Quest* project [2]. The real datasets that we have used come from the UCI KDD Archive [5]. Here we report results obtained on two real datasets from UCI KDD

Archive: Movies[2] and MSWeb[3] (Microsoft Anonymous Web Data), and two synthetic datasets (denoted as Gen1 and Gen2). Table 2 presents basic characteristics of these datasets (for both synthetic datasets the remaining *GEN* parameters not listed in Table 2 were set to the following values: the number of patterns was set to 500 and the average pattern length to 3).

**Table 2.** Datasets used in experiments

| | **Gen1** | **Gen2** | **Movies** | **MSWeb** |
|---|---|---|---|---|
| Number of sets in DB | 100000 | 100000 | 8040 | 32710 |
| Avg length of set in DB | 5 | 8 | 5 | 3 |
| Number of different items in DB | 1000 | 10000 | 14561 | 285 |

We implemented *Common Counting* on top of the classic *Apriori* algorithm using the candidate counting scheme with a separate candidate hash-tree for each query. The experiments were conducted on a PC with AMD Duron 1200 MHz processor and 256 MB of main memory. The datasets used in all experiments resided in flat files on a local disk (disk cache was disabled). The amount of available main memory in the reported experiments was big enough to store all the candidates generated by all the queries in a given iteration. (We discuss the impact of the amount of available memory at the end of this section.)

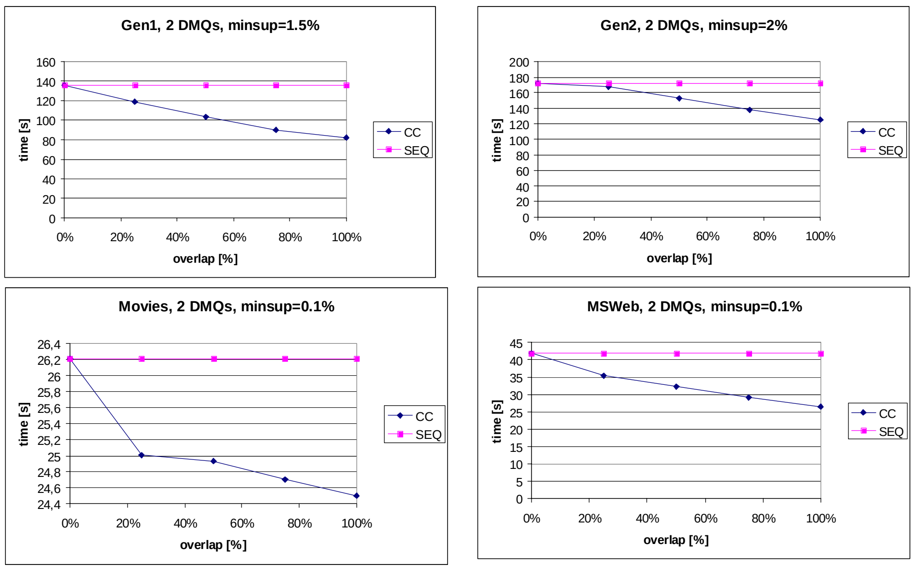

**Fig. 5.** Performance of Common Counting for various levels of dataset overlapping

[2] http://kdd.ics.uci.edu/databases/movies/movies.html
[3] http://kdd.ics.uci.edu/databases/msweb/msweb.html

In the first series of experiments we varied the level of overlapping of source datasets for the case of two queries. Both queries operated on datasets of the same size and containing the same data, the support threshold was also the same for both queries. We changed the level of overlapping from 0% to 100%. Overlapping is expressed as the ratio of the size of the common part of the database to the size of the source dataset of any of the two queries (recall that the experiments were conducted on pairs of identical queries). Figure 5 shows the processing times of *Common Counting* denoted as CC compared to sequential processing of the queries using *Apriori* denoted as SEQ. For all tested datasets *Common Counting* outperformed sequential execution of *Apriori* and the bigger the overlapping of source datasets for the queries the bigger the performance gap between the two methods. However, the performance gains thanks to *Common Counting* depend on the characteristics of the data and the minimum support threshold. *Common Counting* applied to the Movies dataset led to relatively small performance gains since in case of this dataset there was a huge number of candidates generated in the second iteration, and only a few of them turned out to be frequent leading to a small number of iterations (and database scans). Thus, the time spent on candidate verification dominated the time needed to scan the database which is optimized by *Common Counting*.

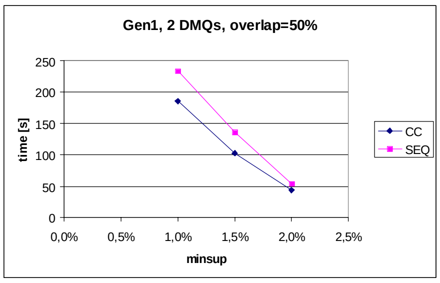

**Fig. 6.** Performance of Common Counting for different minimum support thresholds

In the next experiment we varied the minimum support threshold (same for both queries). Figure 6 presents the impact of the minimum support threshold for the Gen1 dataset, overlapping of 50%, and two queries. Interestingly, the relative performance of *Common Counting* was better for the threshold of 1.5% than for 1% or 2%. This means that neither increasing nor decreasing the support threshold will always work in favor of Common Counting. Such behavior is caused by the fact that the minimum support threshold has an impact on the number of frequent patterns (and indirectly on the number of processed candidates) and the number of iterations (related to the size of the largest frequent pattern). What is guaranteed is that increasing the minimum support threshold will not increase neither the number of candidates not the number of iterations. Similarly, decreasing the minimum support threshold will not decrease neither the number of candidates not the number of iterations. However, changing the support threshold can affect one of the values stronger than the other. Thus, increasing or decreasing the minimum support threshold can change the balance between the time needed to count the candidates (operation not optimized by *Common Counting*) and the time needed to read data for disk (operation optimized by *Common Counting*).

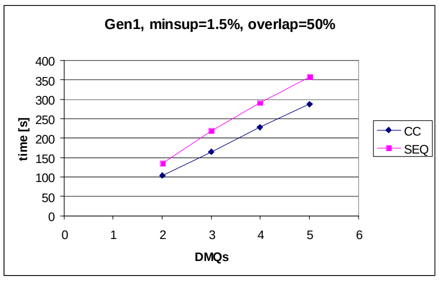

**Fig. 7.** Performance of Common Counting for different number of queries

The next goal of the experiments was testing the efficiency of *Common Counting* for different number of queries processed concurrently. Again, for the sake of simplicity, we performed the tests on collections of identical queries (the same dataset and support threshold) increasing the number of queries from 2 to 5. We considered only one overlapping configuration (from many possible for more than two queries), where there was one partition common to all the queries and the rest of partitions were analyzed each by exactly one query (in such configuration we could express overlapping in the same manner as in the case of two queries). Figure 7 presents the impact of the number of concurrently processed queries for the Gen1 dataset, overlapping of 50%, and minimum support threshold of 1.5%. Again *Common Counting* outperformed *Apriori* in all the cases.

All the results reported above were obtained on collections of identical queries (same data, same minimum support threshold). If the queries that are to be performed concurrently differ in the minimum support threshold and/or the source data (which is going to be the case in practical applications), then the performance gains thanks to *Common Counting* are smaller than in case of identical queries if the queries require different number of iterations to complete (performance gains are most significant if all scans of the common part of the database are exploited by all overlapping queries). It should be noted that the number of *Apriori* iterations for a given query is not known until the query completes, and depends on the characteristics of the data and the minimum support threshold.

As we mentioned earlier, in the experiments reported above all the candidates processed in a given iteration fit into main memory, which is realistic for today's computers (in case of reasonable support thresholds) but not required neither by *Apriori* nor by our *Common Counting* method. Recall that if the candidates to be verified in a given partition of database in a given iteration do not fit into main memory, then they have to be processed in parts and the corresponding partition is scanned more than once. Thus, limiting the amount of main memory can degrade performance of original *Apriori* as well as *Common Counting*. However, the relative performance of *Common Counting* compared to sequential *Apriori* may improve, degrade, or stay unchanged with the increase of available memory. The actual performance gains depend on the ratio of the amount of data read from disk by the

*Common Counting* method to the amount of data read from disk by the queries executed sequentially.

## 6 Conclusions

We have presented our experiences in implementing and evaluating the *Common Counting* method for concurrent frequent itemset queries. The *Common Counting* method is specific to the class of algorithms that at least in some phases need to count the occurrences of candidates in the source dataset. The method exploits the fact that the source datasets of queries that are to be processed can overlap, and consists in reducing the number of scans of parts of the database that are common to two or more queries.

We have implemented the *Common Counting* method on top of the classic *Apriori* algorithm and experimentally tested its efficiency with respect to various parameters. Our experiments showed that the method is efficient and usually leads to significant performance gains compared to sequential processing of queries.

In the future we plan to further investigate the *Common Counting* scheme focusing on issues regarding possible strategies of managing candidate sets of queries processed concurrently. We also plan to work on methods of processing of sets of data mining queries that do not depend on a particular mining scheme (such as *Apriori* in case of *Common Counting*).

## 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, Portland, Oregon (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. Ceri S., Meo R., Psaila G.: A New SQL-like Operator for Mining Association Rules. Proc. of the 22nd Int’l Conference on Very Large Data Bases (1996)
5. Hettich S., Bay S. D.: The UCI KDD Archive [http://kdd.ics.uci.edu]. Irvine, CA: University of California, Department of Information and Computer Science (1999)
6. Han J., Fu Y., Wang W., Chiang J., Gong W., Koperski K., Li D., Lu Y., Rajan A., Stefanovic N., Xia B., Zaiane O.R.: DBMiner: A System for Mining Knowledge in Large Relational Databases. Proc. of the 2nd KDD Conference (1996)
7. Han J., Pei J.: Mining Frequent Patterns by Pattern-Growth: Methodology and Implications. SIGKDD Explorations, December 2000 (2000)
8. Imielinski T., Mannila H.: A Database Perspective on Knowledge Discovery. Communications of the ACM, Vol. 39, No. 11 (1996)
9. Imielinski T., Virmani A., Abdulghani A.: Datamine: Application programming interface and query language for data mining. Proc. of the 2nd KDD Conference (1996)
10. Zheng Z., Kohavi R., Mason L.: Real World Performance of Association Rule Algorithms. Proc. of the 7th KDD Conference (2001)

11. Morzy T., Wojciechowski M., Zakrzewicz M.: Data Mining Support in Database Management Systems. Proc. of the 2nd DaWaK Conference (2000)
12. Morzy T., Zakrzewicz M.: SQL-like Language for Database Mining. ADBIS'97 Symposium (1997)
13. Wojciechowski M., Zakrzewicz M.: Methods for Batch Processing of Data Mining Queries, Proc. of the 5th International Baltic Conference on Databases and Information Systems (2002)
14. Zaki M.J.: Scalable Algorithms for Association Mining. IEEE Transactions on Knowledge and Data Engineering, Vol. 12, No. 3 (2000)