

Evaluation of the Mine Merge Method for Data Mining Query Processing*

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. In this paper we consider concurrent execution of multiple data mining queries in the context of discovery of frequent itemsets. If such data mining queries operate on similar parts of the database, then their overall I/O cost can be reduced by transforming the set of data mining queries into another set of non-overlapping queries, whose results can be used to efficiently answer the original queries. We discuss the problem of multiple data mining query optimization and experimentally evaluate the Mine Merge algorithm to efficiently execute sets of data mining queries.

1 Introduction

Data mining is a database research field which aims at the discovery of trends, patterns and regularities in very large databases. We are currently witnessing the evolution of data mining environments towards their full integration with DBMS functionality. In this context, data mining is considered to be an advanced form of database querying, where users formulate declarative *data mining queries*, which are then optimized and executed by one of data mining algorithms built into the DBMS. One of the most significant issues in data mining query processing are their long execution times, ranging from minutes to hours.

One of the most popular pattern types discovered by data mining queries are *frequent itemsets*. Frequent itemsets describe co-occurrences of individual items in item sets stored in the database. An example of a frequent itemset can be a collection of products that customers typically purchase together during their visits to a supermarket. Such frequent itemsets can be discovered in the database of customer shopping baskets. Frequent itemsets are usually discovered using *level-wise algorithms*, which divide the problem into multiple iterations of database scanning and counting occurrences of candidate itemsets of equal size.

Due to long execution times, data mining queries are often performed in a *batch mode*, where users submit sets of data mining queries to be executed during low database activity time (e.g., night time). It is likely that the batches contain data

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

mining queries that operate on similar parts of the database. If such queries are executed independently, the same parts of the database are retrieved multiple times.

In [16] we introduced the problem of *multiple data mining query optimization*, aimed at reducing the overall I/O activity of the batch of data mining queries. We proposed two multi-query processing algorithms for discovery of frequent itemsets: *Common Counting* and *Mine Merge*, providing theoretical cost analysis for each of the methods. The *Common Counting* algorithm executes all the data mining queries concurrently, performing a common database pass to count candidate itemsets belonging to all the queries. *Common Counting* requires that all candidate itemsets for multiple data mining queries fit in memory together. *Mine Merge* transforms a batch of possibly overlapping queries into a set of queries operating on disjoint partitions of the database, and then merges the results to generate answers to the original queries.

In [17] we discussed implementation details regarding the *Common Counting* method and we presented experimental results proving its efficiency. In this paper we analyze properties of the *Mine Merge* algorithm and experimentally evaluate its performance.

1.1 Related Work

The problem of mining association rules was first introduced in [1] and an algorithm called *AIS* was proposed. In [3], two new algorithms were presented, called *Apriori* and *AprioriTid* that are fundamentally different from the previous ones. The algorithms 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, leading to a level-wise procedure.

Savasere, Omiecinski and Navathe have proposed an algorithm called *Partition* [11], which discovers all frequent itemsets using at most two database passes. *Partition* divides the database into logical, non-overlapping parts, and then it discovers frequent itemsets inside each part. In the last step, the discovered frequent itemsets are merged and their final supports are counted.

In [5], an algorithm called *FUP* (Fast Update Algorithm) was proposed for finding the frequent itemsets in the expanded database using the old frequent itemsets. The major idea of *FUP* algorithm is to reuse the information of the old frequent itemsets and to integrate the support information of the new frequent itemsets in order to reduce the pool of candidate itemsets to be re-examined. Another approach to incremental mining of frequent itemsets was presented in [13]. The algorithm introduced there required only one database pass and was applicable not only for expanded but also for reduced database. Along with the itemsets, a *negative border* [14] was maintained.

The notion of data mining queries (or *KDD* queries) was introduced in [7]. 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][8][9][10].

Multi-query optimization has been extensively studied in the context of database systems (e.g., [12]). A general idea was to exploit the fact that several queries to be answered may share some common data. This general idea remains the same for data mining query processing. However, specialized multi-query processing methods are needed for data mining queries due to their different nature.

2 Basic Definitions and Problem Formulation

Definition 1 (Frequent itemsets). Let $L=\{l_1, l_2, \dots, l_m\}$ 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.

$C_1 = \{\text{all 1-itemsets from } D\}$ for ($k=1; C_k \neq \emptyset; k++$) $\text{count}(C_k, D);$ $L_k = \{c \in C_k \mid c.\text{count} \geq \text{minsup}\};$ $C_{k+1} = \text{generate_candidates}(L_k);$ $\text{Answer} = \bigcup_k L_k;$	$L_1 = \{\text{frequent 1-itemsets}\}$ for ($k = 2; L_{k-1} \neq \emptyset; k++$) $C_k = \text{generate_candidates}(L_{k-1});$ forall tuples $t \in D$ $C_t = C_k \cap \text{subset}(t, k);$ forall candidates $c \in C_t$ $c.\text{count}++;$ $L_k = \{c \in C_k \mid c.\text{count} \geq \text{minsup}\}$ $\text{Answer} = \bigcup_k L_k;$
---	--

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

Definition 2 (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 procedure ensures that all subsets of size $k-1$ of C_k are all members of the set L_{k-1} . This method of pruning the C_k set using L_{k-1} significantly reduces the number of candidates that have to be counted. In the k -th pass, the algorithm counts the supports of all the itemsets in C_k . To facilitate efficient counting procedure, candidates are store in a hash-tree data structure. 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 .

Definition 3 (Data mining query). A data mining query is a tuple $(R, a, \Sigma, \Phi, \beta)$, where R is a database relation, a is an attribute of R , Σ is a selection predicate on R , Φ is a selection predicate on frequent itemsets, β is the minimum support for the frequent itemsets.

Example. Given is the database relation $R_1(attr_1, attr_2)$. The data mining query $dmq_1 = (R_1, "attr_2", "attr_1 > 5", "|itemset| < 4", 3)$ describes the problem of discovering frequent itemsets in the set-valued attribute $attr_2$ of the relation R_1 . The frequent itemsets with support above 3 and length less than 4 are discovered in records having $attr_1 > 5$.

Definition 4 (Multiple data mining query optimization). Given is a set of data mining queries $DMQ = \{dmq_1, dmq_2, \dots, dmq_n\}$, where $dmq_i = (R, a, \Sigma_i, \Phi_i, \beta_i)$, Σ_i is of the form " $(l_{1min}^i < a < l_{1max}^i) \vee (l_{2min}^i < a < l_{2max}^i) \vee \dots \vee (l_{kmin}^i < a < l_{kmax}^i)$ ", and there are at least two data mining queries $dmq_i = (R, a, \Sigma_i, \Phi_i, \beta_i)$ and $dmq_j = (R, a, \Sigma_j, \Phi_j, \beta_j)$ such that $\sigma_{\Sigma_i}R \cap \sigma_{\Sigma_j}R \neq \emptyset$. The problem of multiple data mining query optimization is to generate an algorithm to execute DMQ with the minimal I/O cost.

Definition 5 (Data sharing graph). Let $S = \{s_1, s_2, \dots, s_k\}$ be a set of elementary data selection predicates for DMQ , i.e. selection predicates over the attribute a or the relation R such that for all i, j we have $\sigma_{s_i}R \cap \sigma_{s_j}R = \emptyset$ and for each i there exist integers a, b, \dots, m such that $\sigma_{\Sigma_i}R = \sigma_{s_a}R \cup \sigma_{s_b}R \cup \dots \cup \sigma_{s_m}R$ (example in Fig. 2). A graph $DSG = (V, E)$ is called a data sharing graph for the set of data mining queries DMQ iff $V = DMQ \cup S$, $E = \{(dmq_i, s_j) \mid dmq_i \in DMQ, s_j \in S, \sigma_{\Sigma_i}R \cap \sigma_{s_j}R \neq \emptyset\}$.

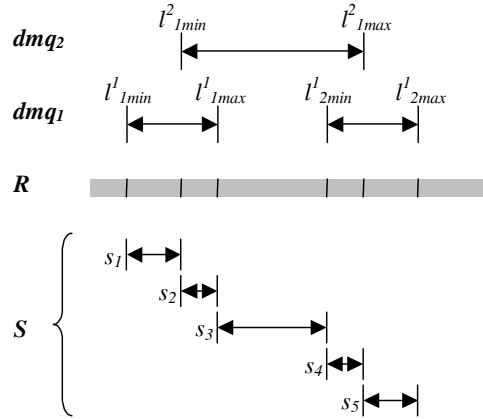


Fig. 2. Example set of data mining queries and their elementary data selection predicates

Example. Given is 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 $S=\{s_1="0 < attr_1 < 5", s_2="5 < attr_1 < 15", s_3="15 < attr_1 < 20", s_4="30 < attr_1 < 40"\}$. The data sharing graph for $\{dmq_1, dmq_2, dmq_3\}$ is shown in Fig. 3.

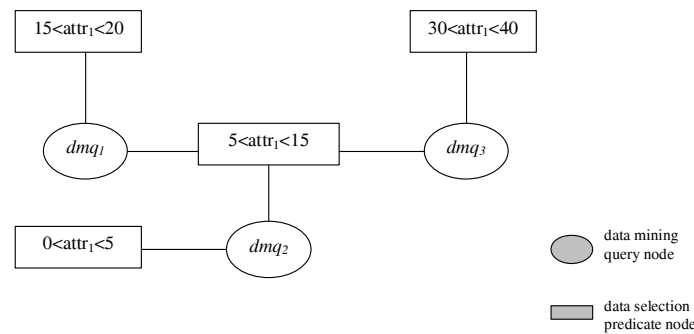


Fig. 3. Example data sharing graph

3 Mine Merge Algorithm

One of the ways to perform multiple data mining query optimization is the *Mine Merge* algorithm. The algorithm 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 [11].

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 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. It is important that not all global candidate itemsets must be counted in that step: if a global candidate itemset belongs to the results of all the appropriate intermediate data mining queries, then its support value can be derived by summing support values it received from the queries. The pseudocode of the *Mine Merge* algorithm is shown in Fig. 4.

```

Generate intermediate data mining queries  $IDMQ = \{idmq_1, idmq_2, \dots\}$ 
 $IDMQ \leftarrow \emptyset$ 
for each  $s_j \in S$  do begin
   $Q \leftarrow \{dmq_i \in DMQ \mid (dmq_i, s_j) \in E\}$ 
   $intermediate\_beta \leftarrow \min\{\beta_i \mid dmq_i = (R, a, s_i, \Phi, \beta_i) \in Q\}$ 
   $intermediate\_Phi \leftarrow \Phi_1 \vee \Phi_2 \vee \dots \vee \Phi_{|Q|}, \forall i=1..|Q|, dmq_i = (R, a, s_i, \Phi_i, \beta_i) \in Q$ 
   $IDMQ \leftarrow IDMQ \cup idmq_j = (R, a, s_j, intermediate\_Phi, intermediate\_beta)$ 
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  $DMQ = \{dmq_1, dmq_2, \dots\}$ 
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
   $CC \leftarrow \bigcup C^j: (dmq_i, s_j) \in E; \quad /* \text{select the candidates to count now} */$ 
  if  $CC \neq \emptyset$  then  $count(CC, \sigma_j R);$ 
end
for  $(i=1; i \leq n; i++)$  do
   $Answer^i \leftarrow \{c \in C^i \mid c.count \geq \beta_i\} \quad /* \text{generate responses} */$ 

```

Fig. 4. Mine Merge algorithm

Let us consider an example of *Mine Merge* algorithm execution, based on our previous set of data mining queries from Fig. 3. Let $cost(s)$ be the I/O cost of retrieving database records that satisfy the data selection predicate s . Let $treewidth(dm, k)$ be the k -item candidate hash tree size for the data mining query dm . Sample costs and tree sizes are given in the table below. For the sake of simplicity assume it takes 5 Apriori iterations to discover frequent itemsets for each intermediate data mining query. Also, assume that each intermediate data mining query discovers 100KB of frequent itemsets, whose I/O cost is 100.

s_i	$cost(s_i)$	C^i	$treewidth(*, i)$
$0 < attr_1 < 5$	2,000	C^1	2M
$5 < attr_1 < 15$	40,000	C^2	20M
$15 < attr_1 < 20$	3,000	C^3	10M
$30 < attr_1 < 40$	6,000	C^4	4M
		C^5	1M

The intermediate data mining queries generated by *Mine Merge* (Fig. 5) are the following: $idmq_1 = (R_1, "attr_2", "0 < attr_1 < 5", \emptyset, 5)$, $idmq_2 = (R_1, "attr_2", "5 < attr_1 < 15", \emptyset, 3)$, $idmq_3 = (R_1, "attr_2", "15 < attr_1 < 20", \emptyset, 3)$, $idmq_4 = (R_1, "attr_2", "30 < attr_1 < 40", \emptyset, 4)$.

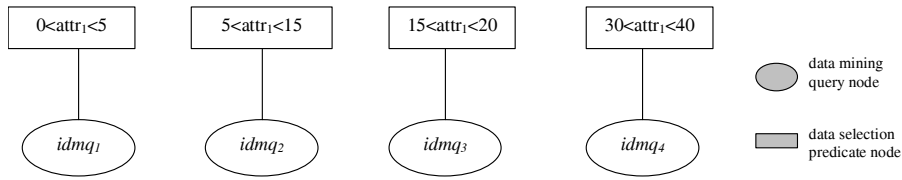


Fig. 5. Intermediate data mining queries

Below we give the I/O and CPU costs for both independent execution and *Mine Merge* execution of the example data mining queries.

Independent execution

operation	I/O cost
execute dmq_1	$(40,000 + 3,000) \times 5$
execute dmq_2	$(40,000 + 2,000) \times 5$
execute dmq_3	$(40,000 + 6,000) \times 5$
total	655,000

Mine Merge execution

operation	I/O cost
execute $idmq_1$	$2,000 \times 5 + 100$
execute $idmq_2$	$40,000 \times 5 + 100$
execute $idmq_3$	$3,000 \times 5 + 100$
execute $idmq_4$	$6,000 \times 5 + 100$
count global candidates	$2,000 + 40,000 + 3,000 + 6,000 + 4 \times 100$
total	306,800

It can be easily noticed that the *Mine Merge* execution can reduce the I/O cost more than twice compared to the independent execution of the three sample data mining queries. At the same time, the CPU cost has been increased since the *Mine Merge* algorithm processes about 50% more itemsets in memory. Therefore, a batch of data mining queries can significantly benefit from *Mine Merge* in disk-bound systems.

The assumption of the equal number of Apriori iterations for all the data mining queries in our example may not hold in practice. Notice that the intermediate data mining queries are likely to discover longer frequent itemsets compared to the final results of the original data mining queries. Such behavior may result from a non-uniform data distribution in database partitions defined by the elementary data selection predicates. Therefore, the real I/O cost of executing the intermediate data mining queries can be higher than we assumed in our example.

4 Experimental Evaluation

In order to evaluate performance of the *Mine Merge* method in the context of frequent itemset mining we performed several experiments on synthetic data, generated by means of the *GEN* generator from the *Quest* project [2]. We experimented with overlapping queries operating on portions of the dataset containing 100000 transactions, generated using the following parameter values: number of different items = 1000, average number of items in a transaction = 8, number of patterns = 500, average pattern length = 4. The experiments were conducted on a PC with AMD Duron 1200 MHz processor and 256 MB of main memory. The dataset used in all experiments resided in a flat file on a local disk. To simulate a realistic environment, where datasets analyzed are significantly larger than the amount of available main memory and therefore do not persist in the system's cache between database scans, we explicitly disabled the operating system's disk cache.

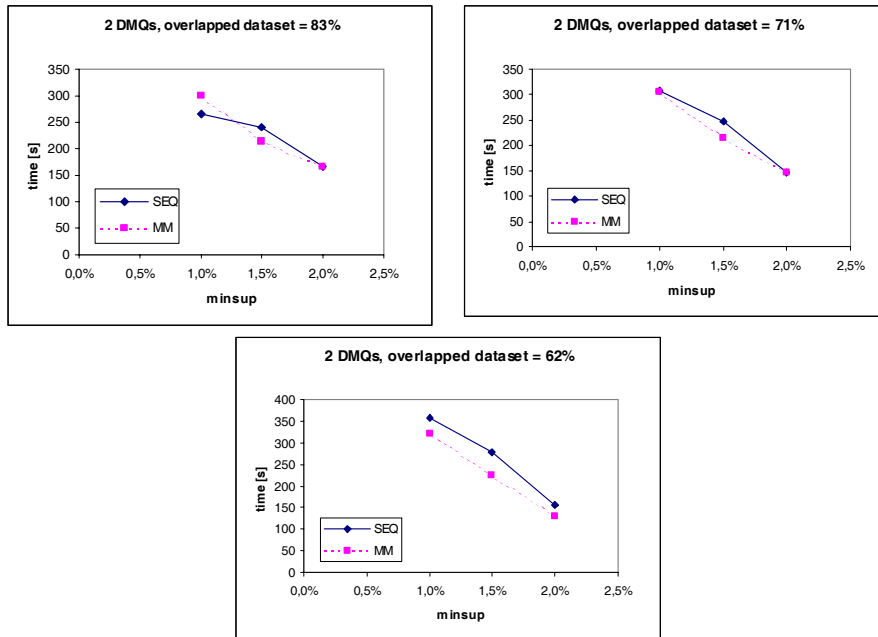


Fig. 6. Performance of Mine Merge for various support thresholds

In the first series of experiments we tested the impact of the minimum support threshold on the performance of *Mine Merge*. We considered pairs of queries with different level of overlapping. We express the overlapping as the ratio of data “covered” by the set of queries to the sum of queries’ sizes (relative size of the overlapped dataset). The advantage of the proposed measure of overlapping is that it works for any number of queries. Figure 6 shows the results for three levels of overlapping and support threshold varying from 1% to 2% (the same threshold for both queries). The execution time of *Mine Merge* (MM) is compared to the execution

time of sequential processing of the queries (SEQ). The experiments show that the impact of minimum support is not deterministic. The reason for this is that changes in the support threshold can change the balance between I/O cost (reduced by *Mine Merge*) and computation cost (reduced or increased by *Mine Merge*, depending on a particular data distribution).

We also analyzed cases when support thresholds for overlapping queries were different (e.g., thresholds of 1.25% and 1.75% instead of 1.5% for both queries). We observed that using different thresholds for the queries degrades the relative performance of *Mine Merge* compared to using the average of the two thresholds for both queries. The actual results depended very strongly on data distribution and support thresholds but still in all cases that we have tested, *Mine Merge* outperformed sequential execution. Different support thresholds for overlapping queries degrade the performance because they lead to lower average support threshold in the dataset partitions.

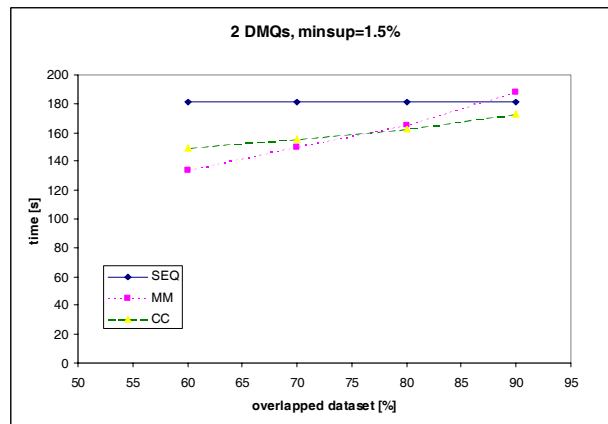


Fig. 7. Performance of Mine Merge for various levels of overlapping (the case of 2 queries)

In the next series of experiments we focused on the impact of query overlapping on the efficiency of the *Mine Merge* method. We tested query sets containing two (Fig. 7) and three (Fig. 8) queries, varying the relative size of the overlapped dataset from 90% to 60%. The minimum support threshold for all queries in all cases was set to 1.5%. For the case of the two queries we compared *Mine Merge* not only to sequential execution but also to the *Common Counting* (CC) method from [17]. (*Common Counting* reduces the I/O costs compared to sequential processing but has no influence on memory computations and therefore is not sensitive to support thresholds and number of queries in the query set.)

The experiments show that if queries' datasets only slightly overlap, *Mine Merge* can even be slower than sequential execution. However, if the queries overlap significantly, *Mine Merge* outperforms both sequential execution and *Common Counting*. As for the impact of the number of queries on *Mine Merge*, greater number of queries leads to more partitions for each query (many overlapping configurations possible) and smaller partitions (more sensitive to changes in data distribution within

the dataset). As a result, the more queries the more significant their overlapping has to be for *Mine Merge* to work efficiently.

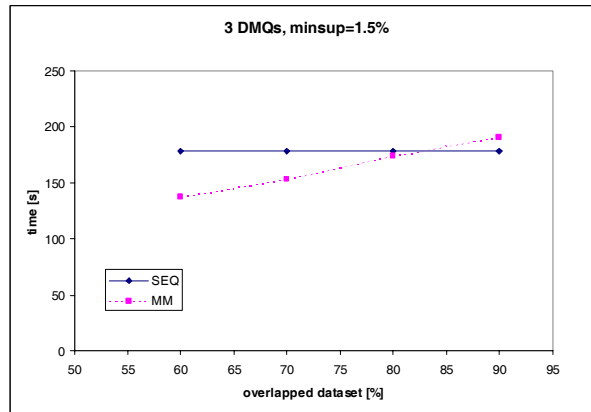


Fig. 8. Performance of Mine Merge for various levels of overlapping (the case of 3 queries)

5 Concluding Remarks

We addressed the problem of multiple data mining query optimization, which consists in sharing some of the execution tasks of multiple data mining queries so that the overall I/O cost is minimized. In this paper we discussed the *Mine Merge* method, which transforms a batch of possibly overlapping queries into a set of queries operating on disjoint partitions of the database, and then merges the results to generate answers to the original queries. We have experimentally evaluated the *Mine Merge* algorithm performance and compared it with the *Common Counting* method previously proposed in the literature.

The experiments show that *Mine Merge* is particularly effective if queries overlap significantly. The advantage of *Common Counting* is that it theoretically guarantees performance gains over sequential processing, which is not the case for *Mine Merge*. However, *Common Counting* requires more memory as it executes several *Apriori* instances concurrently. *Mine Merge* mines dataset partitions one by one, which makes it an attractive solution in practical applications when memory is limited.

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. 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)
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. Imielinski T., Mannila H.: A Database Perspective on Knowledge Discovery. Communications of the ACM, Vol. 39, No. 11 (1996)
8. Imielinski T., Virmani A., Abdulghani A.: Datamine: Application programming interface and query language for data mining. Proc. of the 2nd KDD Conference (1996)
9. Morzy T., Wojciechowski M., Zakrzewicz M.: Data Mining Support in Database Management Systems. Proc. of the 2nd DaWaK Conference (2000)
10. Morzy T., Zakrzewicz M.: SQL-like Language for Database Mining. ADBIS'97 Symposium (1997)
11. 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)
12. Sellis T.K.: Multiple-Query Optimization. ACM Transactions on Database Systems, Vol. 13, No. 1 (1988)
13. Thomas S., Bodagala S., Alsabti K., Ranka S.: An Efficient Algorithm for the Incremental Updation of Association Rules in Large Databases. Proc. of the 3rd KDD Conference (1997)
14. Toivonen H.: Sampling Large Databases for Association Rules. Proc. of the 22nd Int'l Conference on Very Large Data Bases (1996)
15. Wojciechowski M., Zakrzewicz M.: Itemset Materializing for Fast Mining of Association Rules. Proc. of the 2nd ADBIS Conference (1998)
16. 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)
17. Wojciechowski M., Zakrzewicz M.: Evaluation of Common Counting Method for Concurrent Data Mining Queries. Proc. of the 7th ADBIS Conference (2003)