Heuristic Scheduling of Concurrent Data Mining Queries

Marek Wojciechowski, Maciej Zakrzewicz

Poznan University of Technology
Institute of Computing Science
ul. Piotrowo 3a, 60-965 Poznan, Poland
tel. +48 61 6652378, fax +48 61 8771525
{marek,mzakrz}@cs.put.poznan.pl

Abstract. Execution cost of batched data mining queries can be reduced by integrating their I/O steps. Due to memory limitations, not all data mining queries in a batch can be executed together. In this paper we introduce our heuristic algorithm called *CCFull*, which suboptimally schedules the data mining queries into a number of separate execution phases. The algorithm significantly outperforms the optimal approach while providing a very good accuracy.

1 Introduction

Multiple Query Optimization (MQO) [16] is a database research area that focuses on optimizing sets of queries together by executing their common expressions only once in order to save query execution time. Many exhaustive and heuristic algorithms have been proposed for traditional MQO [15][17]. A specific type of a database query is a Data Mining Query (DMQ) [9], which describes a data mining task. It defines constraints on the data to be mined and constraints on the patterns to be discovered. DMQs are expressed using various declarative data mining query languages [5][7][10][12]. DMQs are submitted for execution to a Knowledge Discovery Management System KDDMS [9], which is a Database Management System (DBMS) extended with data mining functionality. Traditional KDDMSs execute DMQs serially and do not try to share any common expressions between different DMQs.

DMQs are often processed in batches of dozens queries, executed during low user activity time. Such queries may show many similarities about their constraints. If they were executed serially, then it would be likely that many I/O operations were wasted because the same database blocks were possibly required by multiple DMQs. If I/O steps of different DMQs were integrated and performed once, then it would be possible to decrease the overall execution cost and time of the whole batch. One of the methods to process batches of DMQs is Apriori Common Counting (ACC), focused on frequent itemset discovery queries [1]. ACC is based on Apriori algorithm [2] and it integrates the steps of candidate support counting – all candidate hash trees for multiple DMQs are loaded into memory and the database is scanned only once. Basic ACC [18] assumes that all DMQs fit in memory, which is not the common case, at least for initial Apriori iterations. If the memory can hold only a subset of all DMQs, then it is necessary to divide/schedule the DMQs into subsets called phases

[19]. The way such scheduling is done determines the overall cost of batched DMQs execution. To solve the scheduling problem, in [19] we proposed an "initial" heuristic algorithm, called *CCRecursive*.

In this paper we present our new, faster heuristic algorithm *CCFull* for scheduling data mining queries to be executed by ACC. We compare its performance and accuracy with the optimal solution. The structure of the paper is the following. Section 2 describes the related work. In Section 3 we discuss the basic definitions and we formally state the data mining query scheduling problem. Section 4 describes *CCFull* algorithm. Section 5 contains experimental results.

2 Related Work

Multiple-query optimization has been extensively studied in the context of database systems (see e.g. [3] [11][15][16][17]), however very little work has been done on optimizing sets of data mining queries. To the best of our knowledge, apart from the Common Counting method discussed in this paper, the only other multiple data mining query processing scheme is Mine Merge, presented in one of our previous papers [20].

3 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, Σ is a condition involving the attributes of the relation R, Φ 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 Φ.

Problem statement. Given is a set of data mining queries $DMQ = \{dmq_1, dmq_2, ..., dmq_n\}$, where $dmq_i = (\mathcal{R}, a, \Sigma_i, \Phi_i, \beta_i)$, Σ_i has the form "$(l^i_{1min} < a < l^i_{1max}) \vee (l^i_{2min} < a < l^i_{2max}) \vee .. \vee (l^i_{kmin} < a < l^i_{kmax})$", $l^i_* \in dom(a)$ and there exist at least two data mining queries $dmq_i = (\mathcal{R}, a, \Sigma_i, \Phi_i, \beta_i)$ and $dmq_j = (\mathcal{R}, a, \Sigma_j, \Phi_j, \beta_j)$ such that $\sigma_{\Sigma i}\mathcal{R} \cap \cap \sigma_{\Sigma j}\mathcal{R} \neq \emptyset$. The problem of *multiple query optimization* of *DMQ* consists in generating such an algorithm to execute *DMQ* which has the lowest I/O cost.

Data sharing graph. Let $S = \{s_1, s_2, ..., s_k\}$ be a set of *distinct data selection formulas* for *DMQ*, ie. a set of selection formulas on the attribute a of the relation R such that for each i,j we have $\sigma_{si}\mathcal{R} \cap \sigma_{sj}\mathcal{R} = \emptyset$, and for each i there exist integers $a, b, ..., m$, such that $\sigma_{\Sigma i}\mathcal{R} = \sigma_{sa}\mathcal{R} \cup \sigma_{sb}\mathcal{R} \cup ... \cup \sigma_{sm}\mathcal{R}$. We refer to the graph $DSG = (V,E)$ as to a *data sharing graph* for the set of data mining queries *DMQ* if and only if $V = DMQ \cup S$, $E = \{(dmq_i, s_j) \mid dmq_j \in DMQ, s_j \in S, \sigma_{\Sigma i}\mathcal{R} \cap \sigma_{sj}\mathcal{R} \neq \emptyset\}$. Each data selection formula is additionally weighted with the I/O cost of its execution in a database.

Example. Consider the following example of a data sharing graph. Given is a database relation $\mathcal{R}_1 = (attr_1, attr_2)$ and three data mining queries: $dmq_1 = (\mathcal{R}_1$, "$attr_2$", "$5 < attr_1 < 20$", $\emptyset$, 3), $dmq_2 = (R_1$, "$attr_2$", "$10 < attr_1 < 30$", $\emptyset$, 5), $dmq_3 = (\mathcal{R}_1$, "$attr_2$", "$15 < attr_1 < 40$", $\emptyset$, 4). The set of distinct data selection formulas is: $S = \{s_1$ = "$5 < attr_1 < 10$", s_2 = "$10 < attr_1 < 15$", s_3 = "$15 < attr_1 < 20$", s_4 = "$20 < attr_1 < 30$", s_5 = "$30 < attr_1 < 40$"$\}$. The data sharing graph for $\{dmq_1, dmq_2, dmq_3\}$ is shown in Fig. 1. Ovals represent data mining queries and boxes represent distinct selection formulas. The bracketed numbers in formula nodes are their sample I/O costs, eg. to retrieve database records that satisfy the data selection formula "$5 < attr_1 < 10$", 120 database blocks must be read.

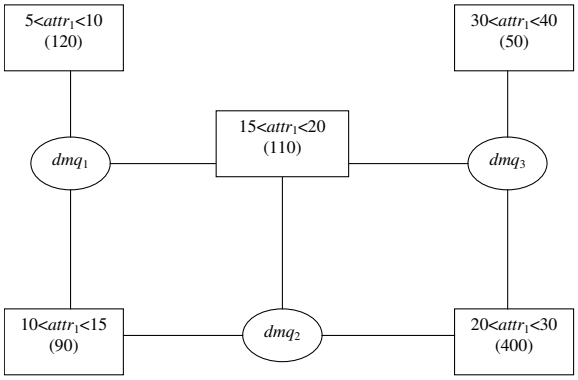

Fig. 1. Sample data sharing graph for a set of data mining queries

Apriori Common Counting (ACC). If the set of data mining queries was executed serially, i.e. one data mining query at a time, then the total execution cost would be the sum of execution costs of data selection formulas for each data mining query separately. ACC executes a set of data mining queries by integrating their I/O operations. It is based on the traditional Apriori approach to discover frequent itemsets. In the first step, for each data mining query we build a separate hash tree for 1-candidates. Next, for each distinct data selection formula we scan its corresponding database partition and we count candidates for all the data mining queries that contain the formula. Such a step is performed for 2-candidates, 3-candidates, etc. Notice that if a given distinct data selection formula is shared by many data mining queries, then its corresponding database partition is read only once. An overview of ACC is shown in Fig. 2.

for (i=1; i<=n; i++) /* n = *number of data mining queries* */
  C_1^i = {all 1-itemsets from $\sigma_{s1 \cup s2 \cup \cdot \cdot \cup sk}\mathcal{R}$, $\forall s_j \in S$: $(dmq_i, s_j) \in E$} /* *generate* 1-*candidates* */
for (k=1; $C_k^1 \cup C_k^2 \cup .. \cup C_k^n \neq \emptyset$; k++) **do begin**
  for each $s_j \in S$ **do begin**
    $CC = \bigcup C_k^l$: $(dmq_l, s_j) \in E$; /* *select the candidates to count now* */

```
    if CC≠ ∅ then count(CC, σ_sj R);
  end
  for (i=1; i<=n; i++) do begin
    F_k^i = {C ∈ C_k^i | C.count ≥ minsup^i};        /* identify frequent itemsets */
    C_{k+1}^i = generate_candidates(F_k^i);
end
   end
for (i=1; i<=n; i++) do
  Answer^i = ∪_k F_k^i;                    /* generate responses */
```

Fig. 2. Apriori Common Counting

4 Heuristic Scheduling of Concurrent Data Mining Queries

4.1 Data Mining Query Scheduling

The basic ACC assumes unlimited memory and therefore the candidate hash trees for all DMQs can completely fit in memory. If, however, the memory is limited, then ACC execution must be divided into multiple *phases*, so that in each phase only a subset of DMQs is processed. In such a case, the key question to answer is: which data mining queries from the set should be executed together in one phase and which data mining queries can be executed in different phases? We refer to the task of data mining queries partitioning as to *data mining query scheduling*.

The problem of data mining query scheduling is a combinatorial problem which can be solved by generating all possible schedules and then choosing the best one. Such approach can be easily used for a small number of data mining queries, however, for a realistic case it is infeasible. The number of all possible schedules is determined by the *Bell number*, e.g. for 13 queries we get over four million schedules. Therefore, we propose a heuristic algorithm, called *CCFull*, which quickly finds a suboptimal schedule.

4.2 Algorithm CCFull

In the first step we generate a *gain graph* for the set of data mining queries. The gain graph is a full hypergraph, in which vertices represent the data mining queries while edges are described with weights which represent the amount of I/O cost reduction to be achieved if data mining queries connected with the edge were executed together (in the same phase). If common execution of given data mining queries results in no reduction of I/O cost, the weight of the connecting edge is zero. A sample gain graph and its original data sharing graph are shown in Fig. 3. For example, it can be noticed that common execution of the data mining queries dmq_0, dmq_2, and dmq_3 would reduce the total I/O cost by 16 units (the weight of the connecting hyperedge)

compared with the sequential execution, since for dmq_0 and dmq_2 the cost of redundant I/O operations is 5 units, for dmq_2 and dmq_3 the cost of redundant I/O operations is 8 units, and for dmq_0 and dmq_3 the cost of redundant I/O operations is 3 units. Using the same example, it can be also noticed, that common execution of only the data mining queries dmq_1 and dmq_2 provides no cost reduction (the weight of the connecting hyperedge is zero).

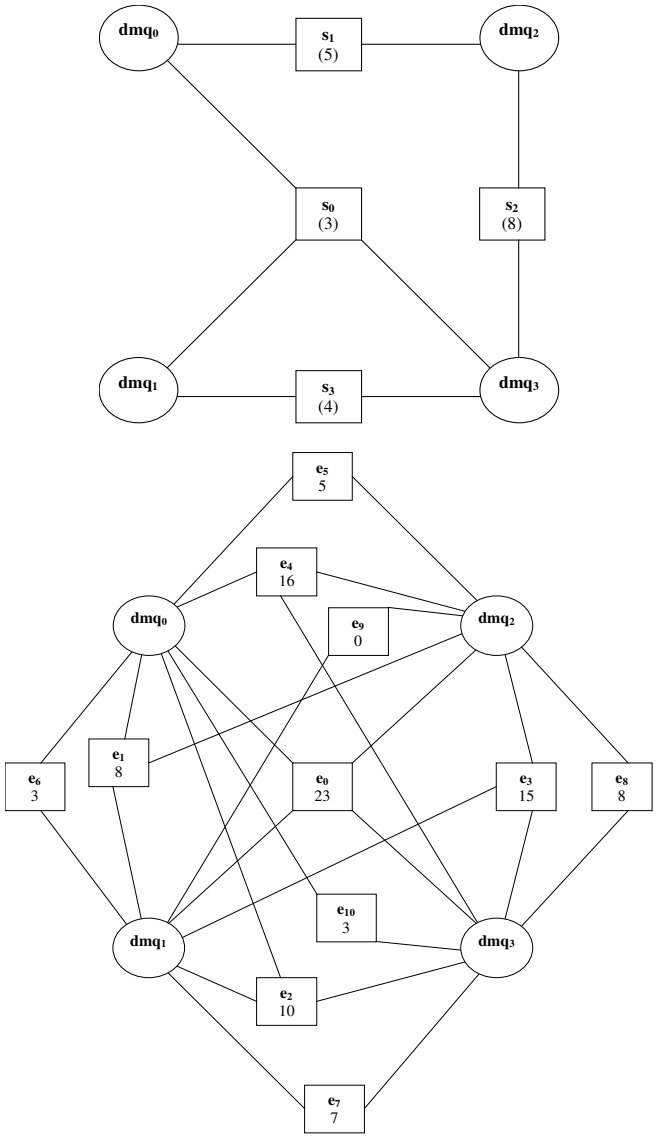

Fig. 3. Sample data sharing graph and corresponding gain graph

The gain graph can be generated using the algorithm *GenerateGainGraph* shown in Fig. 4. The algorithm takes two arguments: the set of all distinct data selection formulas and the set of all data mining queries. First, the algorithm builds a full hypergraph whose nodes are the data mining queries (line 1). Each hyperedge receives the weight of zero, initially (line 3). Then, for each hyperedge e, we create a set P of distinct data selection formulas involved in all data mining queries connected with the hyperedge e (line 4). I/O costs for executing the distinct data selection formulas from P are then summarized and the result is assigned to the hyperedge e weight (line 5 and 6).

```
   GenerateGainGraph(S, DMQ):
     begin
1.   generate a full hypergraph G={V,E}, V=DMQ
2.   for each e ∈ E do begin
3.     e.gain = 0;
4.     P = {s_i ∈ S | ∃ dmq_j ∈ e, dmq_j =(ℛ, a, Σ_j, Φ_j, β_j), s_i ⊆ Σ_j }
5.     for each s ∈ P do begin
6.       e.gain += cost(s)*(|{ dmq_j: dmq_j ∈ e, dmq_j =(ℛ, a, Σ_j, Φ_j, β_j), s_i ⊆ Σ_j }| - 1)
       end
     end
7.   return G
     end
```

Fig. 4. Gain graph generation algorithm

After having created the gain graph, *CCFull* performs the following steps. All hyperedges are sorted in descending order according to their weights. Next, *CCFull* iterates over the hyperedges and checks if data mining queries connected with the current hyperedge have been already scheduled. If none of the data mining queries has been scheduled so far, and if their hash trees fit in memory, then a new phase is generated and the data mining queries are assigned to it. Otherwise, if only some of the data mining queries have been already scheduled to different phases, then *CCFull* tries to combine all those phases together with the unscheduled data mining queries. If such combined phase does not fit in memory, then the current hyperedge is ignored and *CCFull* continues with the next one. The algorithm ends when all hyperedges are processed. The algorithm *CCFull* is shown in Fig. 5.

```
   CCFull(G=(V,E)):
     begin
1.   Phases ← {∅}
2.   sort E = <e_i , e_2 ,..., e_k> in desc. order w.r.to e_i.gain, ignore edges with zero gains
3.   for each e_i in E do begin
4.     tmpV ← {v ∈ V | v ∈ e_i }
5.     if (|{p ∈ Phases | p ∩ tmpV ≠ ∅}| = 0) then
```

```
6.          commonPhases ← ∅
7.          newPhase ← tmpV
            else
8.        commonPhases ← {p ∈ Phases | p∩ tmpV ≠ ∅}
9.        newPhase ← tmpV ∪ ⋃ p| p∈ commonPhases
            end if
10.     if (treesize(newPhase) ≤ MEMSIZE) then
11.       Phases ← Phases - commonPhases
12.       Phases ← Phases ∪ newPhase
          end if
        end
13.     add phase for each unscheduled query
14.     return Phases
      end
```

Fig. 5. *CCFull* algorithm

The detailed steps of the algorithm from Fig. 5 are the following. In line (1) we initialize the set of scheduled phases – we start with the empty set. In line (2) we sort the list E of hyperedges from the gain graph. Hyperedges with weights equal to zero are removed from the list. In line (3) a loop starts, which iterates over the list of hyperedges. In line (4) we select all data mining queries which are connected with the current hyperedge (*tmpV*). In line (5) we test if any of the selected data mining queries belongs to any of the phases scheduled so far. If not, then in line (7) we create a new candidate phase containing all the data mining queries from *tmpV*. Otherwise, in line (9) we create a new candidate phase containing both all the data mining queries from *tmpV* and data mining queries from earlier scheduled phases, to which any of the *tmpV* data mining queries was also scheduled. In line (10) we check if hash trees of all the data mining queries from the new candidate phase fit in memory (*MEMSIZE* is the available memory size). If this condition is satisfied, then in lines (11) and (12) we append the new candidate phase to the current set of scheduled phases *Phases*, possibly replacing some of the existing phases (when multiple phases are combined). In line (13), for each data mining query which has not been scheduled we create a new phase. In step (14) we return the generated phases.

Example

Consider scheduling of data mining queries from Fig. 3. For the sake of simplicity, assume that hash tree sizes are 10MB for each data mining query and the available memory is 20MB.

Hyperedges of the gain graph are sorted according to their weights (skipping zero-weighted hyperedges): $<e_0, e_4, e_3, e_2, e_1, e_8, e_7, e_5, e_6, e_{10}>$. In the first iteration we select the hyperedge e_0, which is connecting the data mining queries dmq_0, dmq_1, dmq_2 and dmq_3. None of the data mining queries has been scheduled so far, and total size of their hash trees is 40MB, exceeding the available memory. Therefore, the

algorithm ignores the hyperedge and starts another iteration. In the second iteration we select the hyperedge e_4, which is connecting the data mining queries dmq_0, dmq_2 and dmq_3. None of the data mining queries has been scheduled so far, and total size of their hash trees is 30MB, exceeding the available memory again. Therefore, the algorithm ignores the hyperedge and starts another iteration. In a similar way the iterations over the hyperedges e_3, e_2 and e_1 are performed – total sizes of hash trees exceed the available memory. Yet in the sixth iteration the algorithm will behave in a different way. We select the hyperedge e_8, which is connecting the data mining queries dmq_2 and dmq_3. The total size of their hash trees is 20MB, so a new phase is created: $\{dmq_2, dmq_3\}$. In the next iteration we select the hyperedge e_7, which is connecting the data mining queries dmq_1 and dmq_3. Since dmq_3 already belongs to a scheduled phase, we try to replace the existing phase $\{dmq_2, dmq_3\}$ with a new one: $\{dmq_1, dmq_2, dmq_3\}$. We are unsuccessfull because the total size of hash trees for the data mining queries is 30MB, what exceeds the available memory. In the next iteration we select the hyperedge e_5, and again we are unsuccessfull when trying to replace the existing phase $\{dmq_2, dmq_3\}$ with a new phase $\{dmq_0, dmq_2, dmq_3\}$. The next iteration operates on the hyperedge e_6, which is connecting the data mining queries dmq_0 and dmq_1. These data mining queries do not belong to any of the existing phases and total size of their hash trees is 20MB. Therefore, a new phase is created: $\{dmq_0, dmq_1\}$. In the last iteration we select the hyperedge e_{10}, which is connecting the data mining queries dmq_0 and dmq_3. Since both data mining queries have been already scheduled to some phases, the algorithm tries to combine the existing phases $\{dmq_2, dmq_3\}$ and $\{dmq_0, dmq_1\}$. However, the phases are not merged since the total size of hash trees of their data mining queries is 40MB and exceeds the available memory. The algorithm has completed. The constructed scheduling of the four data mining queries is shown in Fig. 6.

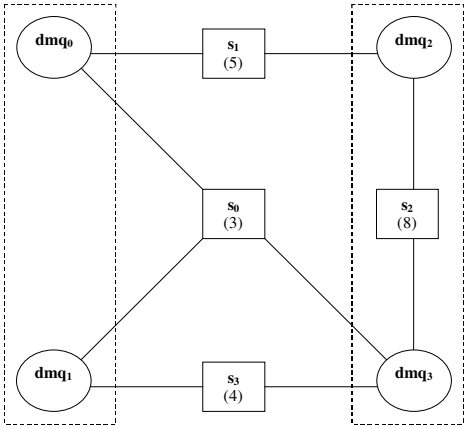

Fig. 6. Data mining queries scheduled with *CCFull*algorithm

5 Experimental Evaluation

In order to evaluate performance and accuracy of the *CCFull* heuristic scheduling algorithm we performed several experiments using the real dataset of MSWeb (Microsoft Anonymous Web Data) dataset from the UCI KDD Archive [8], describing web user visits from February 1998: 285 different URLs, contained in 32710 transactions of three elements each. The experiments were conducted on a PC with AMD Duron 1.2 GHz processor and 256 MB of main memory. The datasets used in all experiments resided in flat files on a local disk. Memory was intentionally restricted to 10kB-50kB. Each experiment was repeated 100 times.

Fig. 7 shows disk I/O costs of schedules generated by the optimal scheduling algorithm, by the *CCFull* algorithm, and by a random algorithm (which randomly builds phases from queries). For example, for the set of 10 data mining queries, the *CCFull* algorithm misses the optimal solution by only 6%.

Fig. 8 illustrates execution times for the optimal scheduling algorithm and for *CCFull*. Notice that the optimal algorithm needs ca. 1000s to schedule 12 data mining queries while *CCFull* executes in 30s.

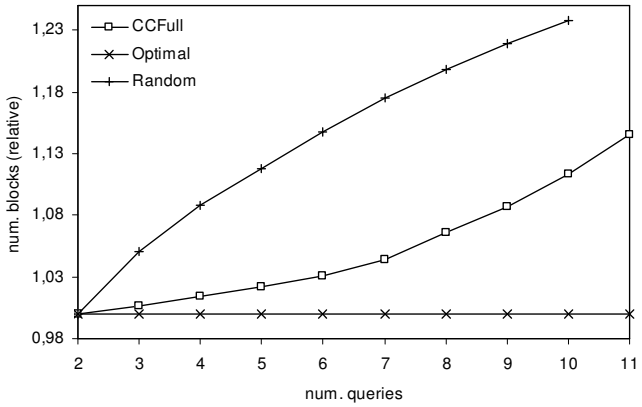

Fig. 7. Accuracy of data mining query scheduling algorithms.

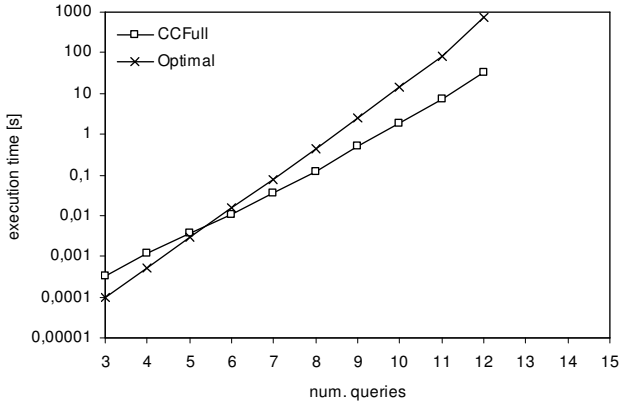

Fig. 8. Execution time of data mining query scheduling algorithms.

7 Conclusions and Future Work

In this paper we have introduced a new heuristic algorithm to schedule data mining queries for Apriori Common Counting. The algorithm offers a significant reduction of execution time over the optimal algorithm while providing a very good accuracy.

CCFull assumes that the set of data mining queries is static. However, in a real system, new queries may arrive while the other queries are being executed. In the future we plan to extend the algorithm to allow for dynamic scheduling of the arriving queries.

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., Srikant R.: Fast Algorithms for Mining Association Rules. Proc. of the 20th Int'l Conf. on Very Large Data Bases (1994)
3. Alsabbagh J.R., Raghavan V.V.: Analysis of common subexpression exploitation models in multiple-query processing. Proc. of the 10th ICDE Conference (1994)
4. Baralis E., Psaila G.: Incremental Refinement of Mining Queries. Proceedings of the 1st DaWaK Conference (1999)
5. 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)
6. 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)

7. 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)
8. 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)
9. Imielinski T., Mannila H.: A Database Perspective on Knowledge Discovery. Communications of the ACM, Vol. 39, No. 11 (1996)
10. Imielinski T., Virmani A., Abdulghani A.: Datamine: Application programming interface and query language for data mining. Proc. of the 2nd KDD Conference (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. Morzy T., Wojciechowski M., Zakrzewicz M.: Data Mining Support in Database Management Systems. Proc. of the 2nd DaWaK Conference (2000)
13. Morzy T., Wojciechowski M., Zakrzewicz M.: Materialized Data Mining Views. Proceedings of the 4th PKDD Conference (2000)
14. 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)
15. 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)
16. Sellis T.: Multiple query optimization. ACM Transactions on Database Systems, Vol. 13, No. 1 (1988)
17. Sellis T., Ghosh S.: On the Multi-Query Optimization Problem. IEEE Transactions on Knowledge and Data Engineering, Vol. 2, No. 2 (1990)
18. Wojciechowski M., Zakrzewicz M.: Evaluation of Common Counting Method for Concurrent Data Mining Queries. Proc. of the 7th ADBIS Conference (2003)
19. Wojciechowski M., Zakrzewicz M.: Data Mining Query Scheduling for Apriori Common Counting. Proc. of the 6th Int'l Baltic Conf. on Databases and Information Systems (2004)
20. Wojciechowski M., Zakrzewicz M.: Evaluation of the Mine Merge Method for Data Mining Query Processing. Proc. of the 8th ADBIS Conference (2004)
21. Zheng Z., Kohavi R., Mason L.: Real World Performance of Association Rule Algorithms. Proc. of the 7th KDD Conference (2001)