A Greedy Approach to Concurrent Processing of Frequent Itemset Queries

Pawel Boinski, Marek Wojciechowski, Maciej Zakrzewicz

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

Abstract. We consider the problem of concurrent execution of multiple frequent itemset queries. If such data mining queries operate on overlapping parts of the database, then their overall I/O cost can be reduced by integrating their dataset scans. The integration requires that data structures of many data mining queries are present in memory at the same time. If the memory size is not sufficient to hold all the data mining queries, then the queries must be scheduled into multiple phases of loading and processing. Since finding the optimal assignment of queries to phases is infeasible for large batches of queries due to the size of the search space, heuristic algorithms have to be applied. In this paper we formulate the problem of assigning the queries to phases as a particular case of hypergraph partitioning. To solve the problem, we propose and experimentally evaluate two greedy optimization algorithms.

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. A specific type of a database query is a Data Mining Query (DMQ) [10], which describes a data mining task. It defines constraints on the data to be mined and constraints on the patterns to be discovered. Existing data mining systems execute DMQs serially and do not try to share any common expressions between different DMQs.

DMQs can be processed in batches, executed during low user activity time. If source datasets of the batched queries overlap, serial execution will result in reading certain parts of the database more times than necessary. If I/O steps of batched DMQs were integrated, 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 Common Counting, focused on frequent itemset discovery queries [1]. It is based on Apriori algorithm [3] 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 Common Counting [17] 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 partition/schedule the DMQs into subsets called phases. The best query scheduling algorithms proposed so far are: *CCAgglomerative* [19] and its extension called *CCAgglomerativeNoise* [6]. In this paper we propose and experimentally evaluate two new greedy optimization algorithms: *CCGreedy* and *CCSemiGreedy*. *CCGreedy* implements a pure greedy strategy, and *CCSemiGreedy* is its extension following a semi-greedy heuristics [9].

2 Related Work

Multiple-query optimization has been extensively studied in the context of database systems (see e.g. [16]), 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 [18].

As an introduction to multiple data mining query optimization, we can regard techniques of reusing results of previous queries to answer a new query [5][7][11] [14]. As we have shown in [15], these methods can be used to optimize processing of batches of data mining queries after appropriate ordering of the queries. However, such an approach is applicable just in a small fraction of cases that Common Counting can successfully handle.

Hypergraph partitioning has been extensively studied particularly in the domain of VLSI design [4]. In data mining context it has been proposed as a clustering technique in [13]. Many formulations of the hypergraph partitioning problem have been considered, differing in partitioning constraints and objectives (see e.g. [4] or [12]). Our formulation differs from typical approaches because we do not have any balance constraint on the sizes of resulting partitions, only a strict upper bound on the on the sum of weights of vertices in a partition, reflecting the memory limit.

3 Background

3.1 Basic Definitions

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

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

3.2 Review of Common Counting

Common Counting is so far the best algorithm for multiple-query optimization in frequent itemset mining. It consists in concurrent execution of a set of frequent itemset queries using the Apriori algorithm and integrating their dataset scans. The algorithm iteratively generates and counts candidates for all the data mining queries, storing candidates generated for each query in a separate hash-tree structure. For each distinct data selection formula, its corresponding database partition is scanned once per iteration, and candidates for all the queries referring to that partition are counted.

Basic Common Counting assumes that memory is unlimited and therefore the candidate hash-trees for all queries can completely fit in memory. If, however, the memory is limited, Common Counting execution must be divided into multiple *phases*, so that in each phase only a subset of queries is processed. In general, many assignments of queries to phases are possible, differing in the reduction of I/O costs. The task of assigning queries to phases in a way minimizing the overall I/O cost is called *query scheduling*.

Since the sizes of candidate hash-trees change between Apriori iterations, the scheduling has to be performed at the beginning of every Apriori iteration. A scheduling algorithm requires that sizes of candidate hash-trees are known in advance. Therefore, in each iteration of Common Counting, we first generate all the candidate hash-trees, measure their sizes, save them to disk, schedule the data mining queries, and then load the hash-trees from disk when they are needed. The exhaustive search for an optimal assignment of queries to Common Counting phases is inapplicable for large batches of queries due to the size of the search space (expressed by a Bell number). Therefore, several scheduling heuristics have been proposed.

4 Frequent Itemset Query Scheduling by Hypergraph Partitioning

4.1 Data Sharing Hypergraph

A set of frequent itemset queries $DMQ = \{dmq_1, dmq_2, ..., dmq_n\}$ can be modeled as a weighted hypergraph whose vertices represent queries and hyperedges represent elementary data selection predicates. A hyperedge in the hypergraph corresponds to a database partition and connects the queries whose source datasets *share* that partition.

Formally, a *data sharing hypergraph* for the set of data mining queries $DMQ = \{dmq_1, dmq_2, ..., dmq_n\}$ and its corresponding set of elementary data selection predicates $S=\{s_1, s_2 ,..., s_k\}$ is a hypergraph $DSG=(V,E)$, where $V=DMQ$, $E=S$, and a vertex $dmq_i \in DMQ$ is incident with an hyperedge $s_j \in S$ iff $\sigma_{sj}R \subseteq \sigma_{\Sigma i}R$. Each vertex dmq_i has an associated weight $w(dmq_i)$ representing the amount of memory consumed by data structures of the query dmq_i. Each hyperedge s_j has an associated weight $w(s_j)$ representing the size of the database partition returned by the elementary data selection predicate s_j.

Note that the above definition of a data sharing hypergraph allows hyperedges incident with only one vertex in order to represent database partitions read by only one query.

Example. Given three frequent itemset queries operating on the relation $R_1 = (a_1, a_2)$: dmq_1=(R_1, "a_2", "5<a_1<20", ∅, 3%), dmq_2=(R_1, "a_2", "10<a_1<30", ∅, 5%), dmq_3=(R_1, "a_2", "15<a_1<40", ∅, 4%). The set of elementary data selection predicates for the set of frequent itemset queries DMQ={dmq_1, dmq_2, dmq_3} is S={"5<a_1<10", "10<a_1<15", "15<a_1<20", "20<a_1<30", "30<a_1<40"}. The data sharing hypergraph for DMQ is shown in Fig. 1.

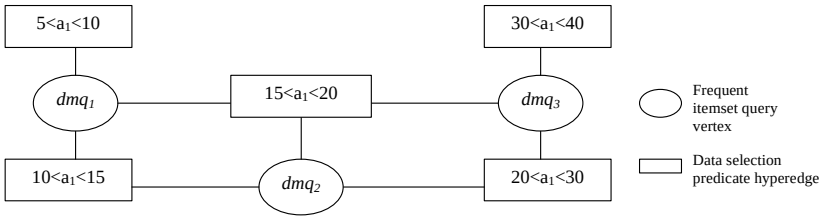

Fig. 1. Example data sharing hypergraph

4.2 Hypergraph Partitioning Problem Formulation

The goal of query scheduling for Common Counting is assigning queries to phases fitting into main memory in a way minimizing the overall I/O cost. Each of the phases returned by the scheduling algorithm is a set of frequent itemset queries for which a data sharing hypergraph can be constructed. Thus, query scheduling for Common Counting can be interpreted as a particular case of hypergraph partitioning.

After partitioning, elementary data selection predicates corresponding to database partitions shared by queries that have been assigned to different phases will be represented as hyperedges in more then one resulting hypergraph. In other words, a hyperedge that is cut by the partitioning will be partitioned into a number of hyperedges connecting subsets of vertices previously connected by the original hyperedge. One of the possible partitionings of the data sharing hypergraph from Fig. 1, representing scheduling into two phases is shown in Fig. 2. Hyperedges that have been cut (partitioned) are presented in bold.

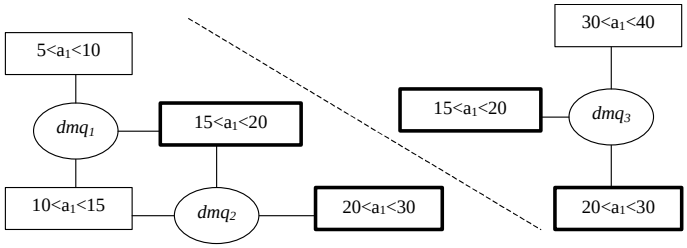

Fig. 2. Example partitioning of the data sharing hypergraph from Fig. 1

In terms of hypergraph partitioning, the goal of query scheduling for Common Counting can be stated as follows:

Problem Statement. Given a data sharing hypergraph for the set of frequent itemsets queries $DSG = (V,E)$ and the amount of available main memory *MEMSIZE*, the goal is to partition the vertices of the hypergraph into k disjoint subsets $V_1, V_2, \ldots, V_k$, and their corresponding data sharing hypergraphs $DSG_1 = (V_1,E_1)$, $DSG_2 = (V_2,E_2)$, …, $DSG_k = (V_k,E_k)$ such that

$$\forall_{x=1..k} \sum_{dmq_i \in V_x} w(dmq_i) \leq MEMSIZE$$

minimizing

$$\sum_{x=1..k} \sum_{s_j \in E_x} w(s_j).$$

In the above formulation, the partitioning constraint has the form of an upper bound on the sum of weights of vertices in each partition, reflecting the amount of available memory, while the partitioning objective is to minimize the total sum of weights of hyperedges across all the partitions, representing the overall I/O cost of the Common Counting iteration. It should be noted that the number of resulting partitions (i.e. Common Counting phases) is not known a priori, and there is no lower bound on the sum of weights of vertices in each partition. Informally, the latter means that we do not require that the resulting partitions are of similar sizes.

According to the classification from [12], the partitioning objective in our problem formulation is equivalent to minimizing the k-1 metric, where the goal is to minimize the size of the hyperedge cut to which each cut hyperedge contributes k-1 times its weight.

Our hypergraph partitioning problem is NP-hard since if we consider only hypergraphs with hyperedges connecting exactly two vertices, its decision version will restrict itself to the classic graph partitioning problem formulation from [8] (proof of NP-completeness by restriction). Taking that into account, for large number of vertices (frequent itemset queries) heuristic approaches have to be applied to solve the problem, resulting in possibly suboptimal solutions.

5 Greedy Approach to Query Scheduling

We propose to solve the hypergraph partitioning problem representing query scheduling for Common Counting by starting with each query in a separate partition and then iteratively merging pairs of partitions, greedily choosing the two partitions whose merging results in greater improvement of the partitioning objective and at the same time does not violate the partitioning constraint. This leads to the *CCGreedy* algorithm presented in Fig. 3. To represent the gain in the partitioning objective for all pairs of partitions the algorithm maintains a *gain graph* $GG=(V, E)$, which is a fully connected graph whose nodes represent partitions and each edge weight represents the gain thanks to merging a pair of partitions connected by the edge. The gain is computed as the difference between the values of partitioning objectives after and before merging a given pair of queries. Limited by space we omit the formal description of initial gain graph generation.

```
CCGreedy(GG=(V,E)):
begin
  while (true) begin
    sort E in desc. order with respect to e_i.gain,  ignore edges with zero gains
    newPartition = ∅
   for each e_i = {v_x, v_y} in E do
    if (treesize(e_i) ≤ MEMSIZE) then
     newPartition = v_x ∪ v_y
     V = V \ {v_x, v_y};       V = V ∪ {newPartition};       E = E \ e_i
     for each v in V do begin
      newEdge = {v, newPartition}, compute newEdge.gain
      E = E ∪ {newEdge}
     end
     break
    end if
   end
   if newPartition = ∅ then break end if
  end
  return V
end
```

Fig. 3. *CCGreedy* algorithm

An obvious problem with greedy algorithms like *CCGreedy* is that the locally optimal choice in each operation may not lead to the globally optimal solution. To increase the chances of finding the optimal partitioning we modify *CCGreedy* by applying a semi-greedy strategy to it. The result is the *CCSemiGreedy* algorithm depicted in Fig.4.

```
CCSemiGreedy(GG=(V,E), RCLLen):
begin
 while (true) begin
   sort E in desc. order wrt. e_i.gain,
   ignore edges with zero gains
   newPartition = ∅
  RCL = genRCL(GG, RCLLen)
  if length(RCL) = 0 then break end if
  randomly choose e_i = {v_x, v_y} from RCL
  newPartition = v_x ∪ v_y
  V = V \ {v_x, v_y};  V = V ∪ {newPartition};   E = E \ e_i
  for each v in V do begin
   newEdge = {v, newPartition}
   compute newEdge.gain
   E = E ∪ {newEdge}
  end
 end
 return V
end

function genRCL(GG=(V,E), RCLLen):
begin
 RCL = nil
 for each e_i = {v_x, v_y}  in E do
  if (treesize(e_i) ≤ MEMSIZE) then
   RCL = append(RCL, e_i)
   if length(RCL) = RCLLen then
    break
   end if
  end if
 end
 return RCL
end
```

Fig. 4. *CCSemiGreedy* algorithm

CCSemiGreedy differs from *CCGreedy* in the step of choosing the partitions to merge. *CCSemiGreedy* uses restricted candidate list (*RCL*) which is returned by the function *genRCL*. This procedure iterates over the gain graph and checks if hash trees of all the queries from a given pair of partitions fit together in memory. If this condition is satisfied, the current edge is added to the *RCL*. Generation of the *RCL* is stopped when the list reaches the length of *RCLLen* (set by a user). In *CCSemiGreedy* we check the length of the *RCL*. If it is zero, there is no possible merge, otherwise an edge (for partition merging) is chosen randomly from the *RCL*. Other steps of the *CCSemiGreedy* algorithm are the same as those described for *CCGreedy* algorithm.

In practice, *CCSemiGreedy* should be applied to query scheduling in the following way: Firstly, an initial schedule should be generated with *CCGreedy*. Then, *CCSemiGreedy* should be executed a user-defined number of times. In the end, the best of the generated schedules should be used for Common Counting.

6 Experimental Evaluation

We implemented our algorithms in C# and conducted experiments on a PC with Intel Pentium IV 2.53GHz processor and 512MB of RAM, running Windows XP.

In the first series of experiments, we performed simulations to determine influence of *CCSemiGreedy* parameters (RCL length and number of attempts) on its effectiveness. We simulated batches of data mining queries by randomly generating the database predicate and size of the candidate tree for each query. Size of available memory was randomly generated in such way that at least every single query could fit into memory. Series of simulations consisted of 500 iterations to get average values and were applied to batches of queries ranging from 3 to 50 queries per batch.

Figure 5 presents the influence of chosen RCL length on the number of disk blocks read by *CCSemiGreedy*. The experiments indicate that the length of the RCL should be very small but greater than 2 items. Best results were obtained for 3 to 6 items. For further experiments we have chosen the length of RCL equal to 3.

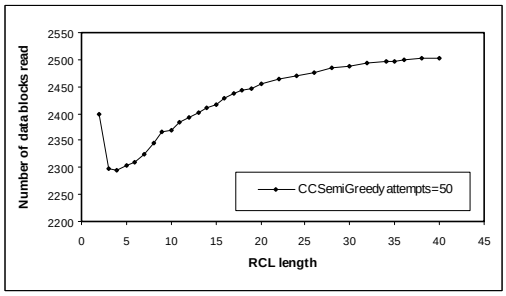

Fig. 5. Influence of the RCL length on the overall accuracy of *CCSemiGreedy*

Figure 6 presents influence of the second parameter of *CCSemiGreedy*, which is the number of attempts to generate schedule. It is obvious that more attempts generally will result in better schedules but at the expense of increasing the scheduling time.

Results indicate that after more than fifty attempts there is no significant improvement in the quality of the schedule.

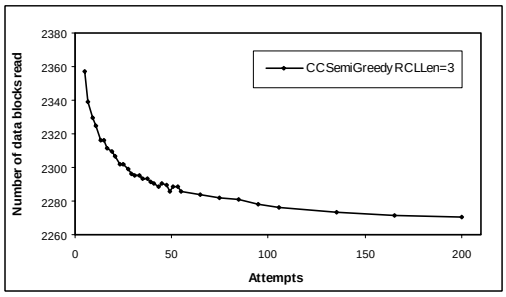

Fig. 6. Influence of the number of attempts on the overall accuracy of *CCSemiGreedy*

In the second series of experiments we compared *CCGreedy* and *CCSemiGreedy* scheduling algorithms with previously proposed *CCAgglomerative* and *CCAgglomerativeNoise* in terms of effectiveness (quality of generated schedules) and efficiency (scheduling times). These experiments were performed on a synthetic dataset generated with GEN [2]. The dataset had the following characteristics: number of transactions = 500000, average number of items in a transaction = 4, number of different items = 10000, number of patterns = 1000. The data resided in a local *PostgreSQL* database. We randomly generated batches of 5 to 30 queries, operating on subsets of the test database.

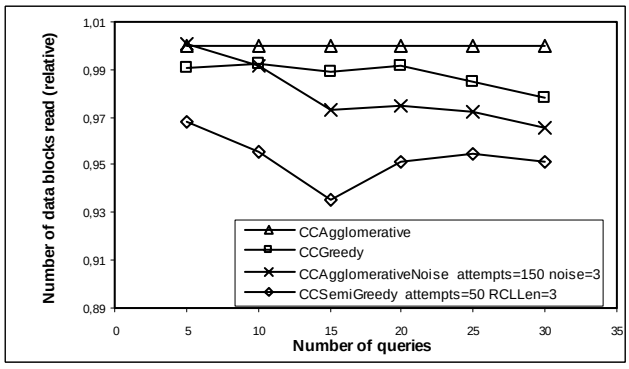

Fig. 7. Amounts of data read by different schedules

Figure 7 presents how the accuracy of the scheduling algorithms changes with the number of queries. To improve readability of the chart, we present relative amount of data blocks read by schedules generated by *CCGreedy*, *CCSemiGreedy* and *CCAgglomerativeNoise* wrt. *CCAgglomerative*. *CCAgglomerativeNoise* iteratively tries to improve the schedule generated by *CCAgglomerative* in a similar way as *CCSemiGreedy* extends *CCGreedy*. We used the optimal value (3%) of the noise parameter of *CCAgglomerativeNoise*, determined in similar simulations to those carried for *CCSemiGreedy*. The number of attempts (150) for *CCAgglomerativeNoise*

was chosen in such way that both *CCSemiGreedy* and *CCAgglomerativeNoise* algorithms had equal time to generate the schedule.

Experiments were performed for three values of the main memory limit (90, 120 and 150kB) and for four levels of the average overlapping of datasets read by queries in the set (20%, 40%, 60%, 80%). Due to limited space we present results that are averages taken over all the conducted experiments. Results show that the most effective schedules are generated by *CCSemiGreedy* and are about 5% better than those generated by *CCAgglomerative*. For *CCAgglomerativeNoise* and *CCGreedy* the measured improvement over *CCAgglomerative* was 2% and 1% respectively.

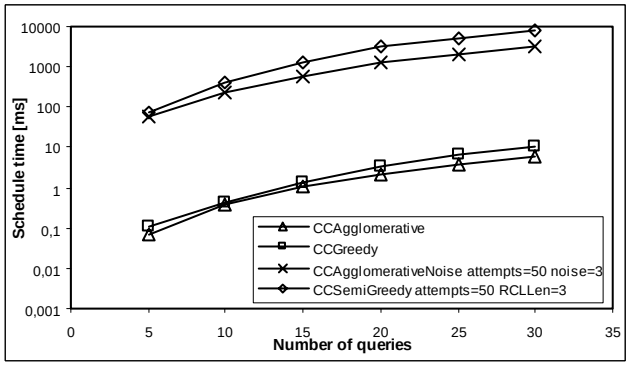

Fig. 8. Scheduling times (logarithmic scale)

Figure 8 presents scheduling times for the considered algorithms. This time for *CCSemiGreedy* and *CCAgglomerativeNoise* numbers of attempts were fixed at the same level (50). Execution times of *CCAgglomerative* and *CCGreedy* are negligible, with *CCGreedy* requiring at most twice as much time as *CCAgglomerative*. Execution times of *CCSemiGreedy* are up to three times longer than those of *CCAgglomerativeNoise* and the gap increases with the number of queries.

The results of our experiments show that *CCGreedy* is more effective than *CCAgglomerative*, and properly parameterized *CCSemiGreedy* generates better schedules than *CCAgglomerativeNoise*, which makes it the best scheduling algorithm for Common Counting. The execution times of the new algorithms are longer but in typical situations the increase in scheduling time will be dominated by the reduction of the time spent on disk operations thanks to better schedules.

7 Summary

In this paper we considered the problem of concurrent execution of frequent itemset queries. We introduced two new heuristic query scheduling algorithms for the Common Counting method: *CCGreedy* and *CCSemiGreedy*. Our experiments show that the new algorithms offer a significant improvement in accuracy over the existing solutions while providing acceptable scheduling times.

CCGreedy and *CCSemiGreedy* assume that the set of data mining queries to execute 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 our approach to allow for dynamic scheduling of 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., Mehta M., Shafer J., Srikant R., Arning A., Bollinger T.: The Quest Data Mining System. Proc. of the 2nd KDD Conference (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. Alpert C.J., Kahng A.B.: Recent Directions in Netlist Partitioning: A Survey. Integration: The VLSI Journal 19 (1995)
5. Baralis E., Psaila G.: Incremental Refinement of Mining Queries. Proceedings of the 1st DaWaK Conference (1999)
6. Boinski P., Jozwiak K., Wojciechowski M., Zakrzewicz M.: Improving Quality of Agglomerative Scheduling in Concurrent Processing of Frequent Itemset Queries. Proc. of the International IIS: IIPWM'06 Conference (2006)
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. Garey M.R., Johnson D.S.: Computers and Intractability. A Guide to the Theory of NP-Completeness. WH Freeman and Company (1979)
9. Hart J.P., Shogan A.W.: Semi-greedy Heuristics: An Empirical Study. Operations Research Letters, Vol. 6 (1987)
10. Imielinski T., Mannila H.: A Database Perspective on Knowledge Discovery. Communications of the ACM, Vol. 39, No. 11 (1996)
11. Jeudy B., Boulicaut J-F.: Using Condensed Representations for Interactive Association Rule Mining. Proceedings of the 6th PKDD Conference (2002)
12. Karypis G.: Multilevel Hypergraph Partitioning. In: Cong J., Shinnerl J. (eds.): Multilevel Optimization Methods for VLSI, Kluwer Academic Publishers (2002)
13. Karypis G., Han E., Kumar V.: Chameleon: A Hierarchical Clustering Algorithm Using Dynamic Modeling. IEEE Computer, Vol. 32, No. 8 (1999)
14. Meo R.: Optimization of a Language for Data Mining. Proc. of the ACM Symposium on Applied Computing - Data Mining Track (2003)
15. Morzy M., Wojciechowski M., Zakrzewicz M.: Optimizing a Sequence of Frequent Pattern Queries. Proc. of the 7th DaWaK Conference (2005)
16. Sellis T.: Multiple Query Optimization. ACM Transactions on Database Systems, Vol. 13, No. 1 (1988)
17. Wojciechowski M., Zakrzewicz M.: Evaluation of Common Counting Method for Concurrent Data Mining Queries. Proc. of the 7th ADBIS Conference (2003)
18. Wojciechowski M., Zakrzewicz M.: Evaluation of the Mine Merge Method for Data Mining Query Processing. Proc. of the 8th ADBIS Conference (2004)
19. 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)