Data Mining Query Scheduling for Apriori Common Counting[*]

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. If such data mining queries operate on similar parts of the database, then their overall I/O cost can be reduced by integrating their data retrieval operations. The integration requires that 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. We discuss the problem of data mining query scheduling and propose a heuristic algorithm to efficiently schedule the data mining queries into phases.

Keywords. Data mining, data mining queries

1. Introduction

Data mining is a database research field that 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 is long execution time, 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 sets of items 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 itemset 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

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

database activity time (e.g., night time). It is likely that the batches contain data mining queries that operate on similar parts of the database. If such queries are executed separately, the same parts of the database are retrieved multiple times. We could reduce the overall I/O activity of the batch of data mining queries if we integrated their data retrieval operations on the same portions of the database.

For a system with unlimited memory, the integration of execution of multiple data mining queries consists in *common counting* [13][14] of candidate itemsets for all the queries so that every portion of the database needs to be read only once per iteration. However, if the memory is limited, we are not able to keep all candidate itemsets of all the data mining queries in the memory at the same time. The whole process must then be split into multiple phases of loading and counting the candidates, and therefore the data mining queries must be divided into subsets to be executed in each phase. We refer to the problem of dividing the data mining into subsets as to the *data mining query scheduling*.

In this paper we discuss the problem of data mining query scheduling and we introduce a heuristic algorithm to perform the scheduling for a system with limited memory. The goal of the algorithm is to schedule the data mining queries in such a way that the overall I/O cost for the whole batch is minimized.

1.1. Related Work

The problem of mining association rules was first introduced in [1] and an algorithm called *AIS* was proposed. In [2], 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. It leads to a level-wise procedure. First, all possible 1-itemsets (itemsets containing 1 item) are counted in the database to determine *frequent 1-itemsets*. Then, frequent 1-itemsets are combined to form potentially frequent 2-itemsets, called *candidate 2-itemsets*. Candidate 2-itemsets are counted in the database to determine *frequent 2-itemsets*. The procedure is continued by combining the frequent 2-itemsets to form *candidate 3-itemsets* and so forth. A disadvantage of the algorithm is that it requires K or $K+1$ passes over the database to discover all frequent itemsets, where K is the size of the greatest frequent itemset found.

In [4], 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 [11]. 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* [12] was maintained.

In [10] the issue of interactive mining of association rules was addressed and the concept of *knowledge cache* was introduced. The cache was designed to hold frequent itemsets that were discovered while processing other queries. Several cache management schemas were proposed and their integration with the *Apriori* algorithm was analyzed. An important contribution was an algorithm that used itemsets discovered for higher support thresholds in the discovery process for the same task, but with a lower support threshold.

The notion of data mining queries (or *KDD* queries) was introduced in [6]. 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 [3][5][7][8][9].

2. Basic Definitions and Problem Formulation

Definition. 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.

Definition. Apriori algorithm.

Apriori is an example of a level-wise algorithm for association 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.

```
C1 = {all 1-itemsets from D};
for (k=1; Ck ≠ ∅; k++) do
begin
   count(Ck, D);
   Lk = {c ∈ Ck | c.count ≥ minsup};
   Ck+1 = generate_candidates(Lk);
end;
Answer = ∪k Lk;
```

```
L1 = {frequent 1-itemsets};
for (k = 2; Lk-1 ≠ ∅; k++) do
begin
   Ck = generate_candidates(Lk-1);
   forall tuples t ∈ D do
   begin
      Ct=Ck ∩ subset(t, k);
      forall candidates c ∈ Ct do
         c.count++;
   end;
   Lk = {c ∈ Ck | c.count ≥ minsup}
end;
Answer = ∪k Lk;
```

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

Definition. 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. Multiple data mining query optimization.

Given is a set of data mining queries $DMQ=\{dmq_1, dmq_2, ..., dmq_n\}$, where $dmq_i=(R, a, \Sigma_i, \Phi_i, \beta_i)$, Σ_i is of the form "$(l^i_{1min}<a<l^i_{1max}) \vee (l^i_{2min}<a<l^i_{2max}) \vee .. \vee (l^i_{kmin}<a<l^i_{kmax})$", and there are at least two data mining queries $dmq_i=(R_i, a, \Sigma_i, \Phi_i, \beta_i)$ 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. Data sharing graph.

Let $S=\{s_1, s_2, ..., s_k\}$ 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 $\sigma_{si}R \cap \sigma_{sj}R = \emptyset$ and for each i there exist integers $a, b, ..., m$ such that $\sigma_{\Sigma i}R=\sigma_{sa}R \cup \sigma_{sb}R \cup .. \cup \sigma_{sm}R$ (example in Fig. 2). A graph $DSG=(V,E)$ is called a *data sharing graph* for the set of data mining queries DMQ iff $V=DMQ \cup S$, $E=\{(dmq_i,s_j) | dmq_j \in DMQ, s_j \in S, \sigma_{\Sigma i}R \cap \sigma_{sj}R \neq \emptyset\}$.

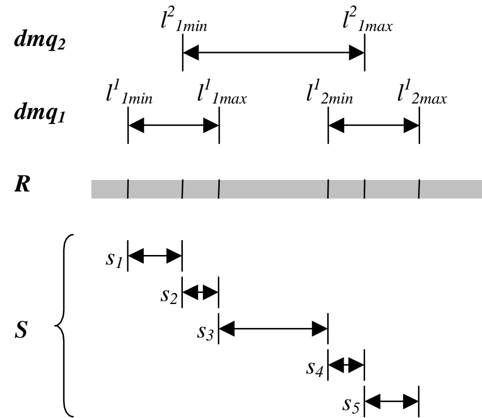

Figure 2. Example set of data mining queries and their elementary data selection predicates.

Example. Given is the relation $R_1=(attr_1, attr_2)$ and three data mining queries: $dmq_1=(R_1,$ "$attr_2$", "$5 < attr1 < 20$", $\varnothing$, 3), $dmq_2=(R_1,$ "$attr_2$", "$10 < attr1 < 30$", $\varnothing$, 5), $dmq_3=(R_1,$ "$attr_2$", "$15 < attr1 <40$", $\varnothing$, 4). The set of elementary data selection predicates is then $S=\{s_1=$"$5<attr_1<10$", $s_2=$"$10<attr_1<15$", $s_3=$"$15<attr_1<20$", $s_4=$"$20<attr_1<30$", $s_5=$ "$30<attr_1<40$"$\}$. The data sharing graph for $\{dmq_1, dmq_2, dmq_3\}$ is shown in Fig. 3.

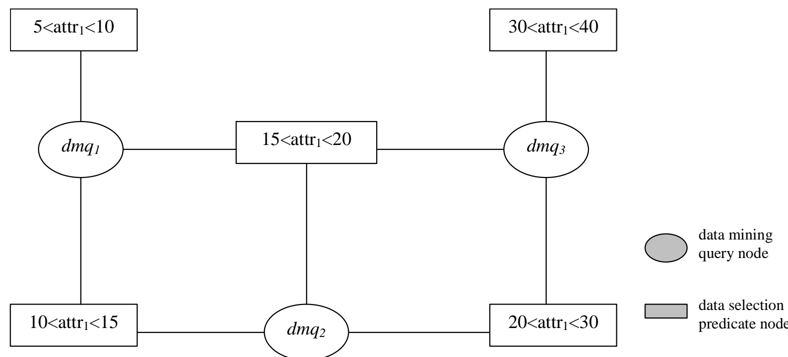

Figure 3. Example data sharing graph.

Definition. Apriori Common Counting.

A straightforward way to perform multiple data mining query optimization is *Apriori Common Counting* algorithm. The algorithm proceeds as follows. In the first step, *Apriori Common Counting* constructs separate candidate 1-itemset hash trees (in memory) for each data mining query. Next, all database partitions corresponding to the elementary selection predicates are scanned and the candidate itemsets for the appropriate data mining queries are counted. This process is repeated for each iteration: for candidate 2-itemsets, candidate 3-itemsets, etc. Notice that if a given elementary selection predicate is shared by multiple data mining queries, then the

specific part of the database needs to be read only once (per iteration). This property helps reduce the overall I/O cost of batched data mining query execution. The idea of *Apriori Common Counting* algorithms is depicted in Fig. 4.

for (i=1; i<=n; i++)   /* *n = number of data mining queries* */
 C_1^i = {all 1-itemsets from $\sigma_{s1 \cup s2 \cup .. \cup sk} R$, $\forall s_j \in S$: $(dmq_i, s_j) \in E$}   /* *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 $CC \neq \emptyset$ **then** *count*(CC, $\sigma_{sj} R$);
  end;
  for (i=1; i<=n; i++) **do begin**
   $L_k^i = \{c \in C_k^i \mid c.count \geq minsup^i\}$;   /* *identify frequent itemsets* */
   C_{k+1}^i = *generate_candidates*(L_k^i);
  end;
end;
for (i=1; i<=n; i++) **do**
  $Answer^i = \bigcup_k L_k^i$;   /* *generate responses* */

Figure 4. Apriori Common Counting.

3. Data Mining Query Scheduling

The basic *Apriori Common Counting* described in the previous section assumes unlimited memory for its operation. However, if the memory is limited, then it is not possible to construct candidate hash trees for all the data mining queries. The whole algorithm must then be split into multiple *phases* and every phase must consist in executing a subset of the data mining queries. The key problem is which data mining queries should be performed in the same phase and which of them can be performed in separate phases. The task of dividing the set of data mining queries into subsets is referred to as *data mining query scheduling*.

There are several aspects to consider when designing a data mining query scheduling algorithm. Firstly, it is obvious that system memory size restricts the number of data mining queries that may be processed in the same phase. Memory requirements for the data mining queries are based on sizes of their candidate hash trees, which in turn depend on data characteristics and the specific iteration of the algorithm (typically, sizes of candidate hash trees systematically reduce for iterations 3, 4, etc.). Since the candidate hash tree sizes change in each iteration, the data mining query scheduling algorithm should be used before generating every new tree, not only at the beginning of the data mining query processing. Another aspect is that the goal of *Apriori Common Counting* is to reduce the overall I/O activity. Therefore, similarities between data mining queries should be taken into account when putting data mining queries into the same phase. Data mining queries that operate on separate portions of the database can be processed in separate phases, while data mining queries that operate on highly overlapping database portions should be executed in the same phase. To measure the “overlapping” between data

mining queries one can rely on a traditional DBMS query optimizer, which estimates predicate costs based on database statistics.

In order to schedule data mining queries, the sizes of their candidate hash trees must be known. There are two options to derive the size. The first option is to calculate the upper bounds on the candidate hash trees and use the upper bounds in the scheduling algorithm. The upper bounds can be evaluated based on the number of frequent itemsets discovered in the previous iteration. A disadvantage of this approach is that the real candidate hash trees are smaller than the estimates, so the scheduling algorithm is likely to miss the optimal solution. The second option is to generate the candidate hash trees first, measure their sizes, save them in temporary files, perform the scheduling and then retrieve the appropriate trees from the files while performing the phases. The main advantage of this approach is that the scheduling algorithm operates on the exact sizes of the trees, and therefore it is able to find the optimal solution. However, the additional I/O cost is introduced because of the need to temporarily store the candidate hash trees on disk. Nevertheless, when dealing with very large databases (in case of which candidate tree sizes are by several orders of magnitude smaller than the database) that extra cost is going to be compensated by reduction of database reads thanks to *Common Counting*.

Let us consider an example of data mining query scheduling 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 *treesize(dmq,k)* be the *k*-item candidate hash tree size for the data mining query *dmq*. Sample costs and tree sizes (e.g., for the third *Apriori* iteration) are given in the table below. Let us assume the system memory limit of 10MB, meaning that at most two of the data mining queries can fit in at a time (i.e., in one phase).

s_i	$cost(s_i)$
$5<attr_1<10$	5000
$10<attr_1<15$	7000
$15<attr_1<20$	2000
$20<attr_1<30$	2000
$30<attr_1<40$	1000

dmq_i	$treesize(dmq_i,3)$
dmq_1	4M
dmq_2	5M
dmq_3	3M

There exist four different schedules that satisfy the given constraints. The schedules and the total costs of executing the sample set of data mining queries are given below. The *Schedule A* represents a sequential execution of all the data mining queries. One can notice that the optimal solution is the *Schedule B*, which reduces the overall cost by 30%. This schedule has been also depicted in Fig. 5.

Schedule A

phase	*data mining queries*	*trees size*	*data selection predicates*	*phase cost*
1	dmq_1	4M	s_1, s_2, s_3	14,000
2	dmq_2	5M	s_2, s_3, s_4	11,000
3	dmq_3	3M	s_3, s_4, s_5	5,000
total cost				**30,000**

Schedule B

phase	*data mining queries*	*trees size*	*data selection predicates*	*phase cost*
1	dmq_1, dmq_2	9M	s_1, s_2, s_3, s_4	16,000
2	dmq_3	3M	s_3, s_4, s_5	5,000
total cost				**21,000**

Schedule C

phase	*data mining queries*	*trees size*	*data selection predicates*	*phase cost*
1	dmq_1, dmq_3	7M	s_1, s_2, s_3, s_4, s_5	17,000
2	dmq_2	5M	s_2, s_3, s_4	11,000
total cost				**28,000**

Schedule D

phase	*data mining queries*	*trees size*	*data selection predicates*	*phase cost*
1	dmq_2, dmq_3	8M	s_2, s_3, s_4, s_5	12,000
2	dmq_1	4M	s_1, s_2, s_3	14,000
total cost				**26,000**

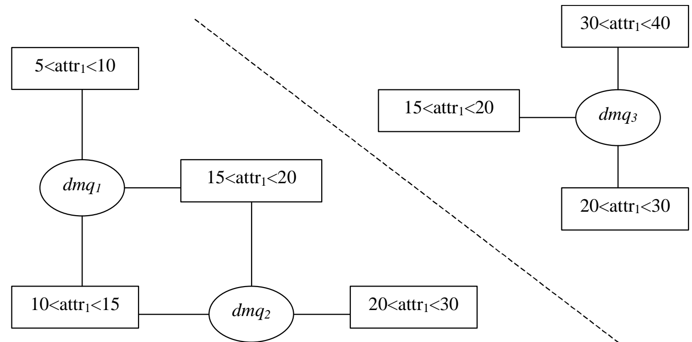

Figure 5. The optimal schedule for the sample set of data mining queries.

The data mining query scheduling problem can be solved using a combinatorial approach, in which all possible (allowable) schedules are generated first, and then their overall costs are calculated. The combinatorial approach can be suitable for a small number of data mining queries in the set, however, for complex problems, involving large numbers of data mining queries, the overhead of the approach would be unacceptable. For a given number of data mining queries, the number of all possible schedules is determined by Bell number – e.g., for 13 queries the number of schedules exceeds 4 millions. Therefore we introduce a heuristic algorithm for finding suboptimal schedules for executing a set of data mining queries.

3.1. Heuristic Scheduling Algorithm: CCRecursive

The algorithm iterates over all the elementary selection predicates, sorted in descending order with respect to their I/O costs. For each elementary selection predicate we identify all the data mining queries that include the predicate. If none of the identified queries has been already scheduled, then we create a new phase and we put all the queries into the new phase. Otherwise, we merge the phases to which the scheduled queries belonged and we assign the other queries to this new phase. If the size of the newly created phase exceeds the memory limit, then the phase is split into smaller ones by recursive execution of the algorithm. At the end of the algorithm, we perform *phase compression*, which consists in merging those phases that do not consume all the available memory. The detailed structure of the algorithm is given in Fig. 6. The auxiliary function *treesize(Q)*, where *Q* is a set of data mining queries, represents total memory size required to hold candidate hash trees for all the data mining queries in *Q*.

```
Phases ← {∅}
sort S = <s_i , s_2 ,..., s_k> in descending order with respect to cost(s_i)
CCRecursive(S, DMQ, Phases):
begin
  ignore in S those predicates that are used by less than two dmqs;
  for each s_i in S do begin
    tmpDMQ ← { dmq_j | dmq_j =(R, a, Σ_j, Φ_j, β_j), s_i ⊆Σ_j , dmq_j ∈ DMQ};
    commonPhases ← {p ∈ Phases | p∩ tmpDMQ  ≠∅};
    if  commonPhases = ∅ then
      newPhase ← tmpDMQ;
    else
      newPhase ← tmpDMQ ∪ U p| p∈ commonPhases;
    end if;
    if treesize(newPhase) ≤MEMSIZE then
              Phases ← Phases \  commonPhases;
              Phases ← Phases ∪ newPhase;
    else
           Phases ← CCRecursive(<s_{i+1}, ... s_k>, newPhase, Phases);
    end if;
  end;
  add phase for each unscheduled query;
  compress Phases containing queries from DMQ;
  return Phases;
end.
```

Figure 6. Heuristic scheduling algorithm: CCRecursive.

4. Experimental Evaluation

To evaluate our heuristic algorithm *CCRecursive* we performed a series of simulations on a PC with *AMD Duron 1200 MHz* processor and 256 MB of main memory. We focused on the isolated problem of scheduling queries into phases fitting in main memory in a given iteration of *Common Counting*. We compared the amount of data read from the database by our heuristic algorithm and the complete "brute-force" algorithm testing all possible assignments of queries to phases.

We simulated actual batches of frequent set discovery tasks by randomly generating a collection of queries. For each query, the database selection predicate and the size of candidate tree was randomly generated. Then the amount of total main memory was also randomly chosen in such a way that the number of queries fitting into it ranged from one query to all the queries.

We performed several series of experiments varying the number of queries. Each of the series consisted of 100 simulations. Figure 7 presents how the accuracy of our heuristic algorithm changes with the number of queries. To assess the accuracy we measured the relative amount of data read from the database by schedules generated by our heuristics compared to the optimal schedules (generated by the complete brute-force scheduling algorithm). For example, in the case of 11 queries, *CCRecursive* generates schedules that read on average about 3.5% more data than the optimal schedules.

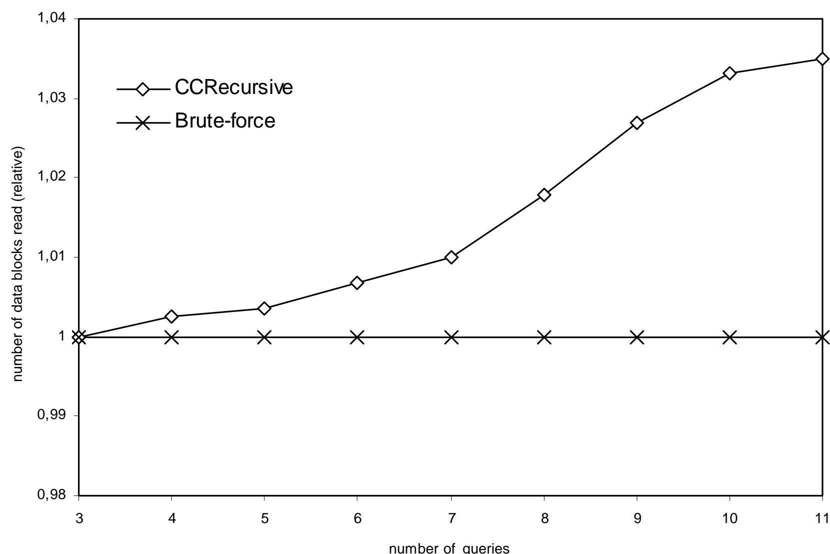

Figure 7. Amounts of data read by CCRecursive schedules and optimal schedules.

Figure 8 presents the execution times (times needed to generate schedules) of *CCRecursive* and the brute-force algorithm. Although *CCRecursive* still scales exponentially with the number of queries, its execution time increases less rapidly

than in case of the brute-force solution. For instance, the brute-force algorithm consumes more than 1000 s already for 12 queries, while *CCRecursive* exceeds that threshold in case of 22 queries (the chart presents the times for up to 15 queries).

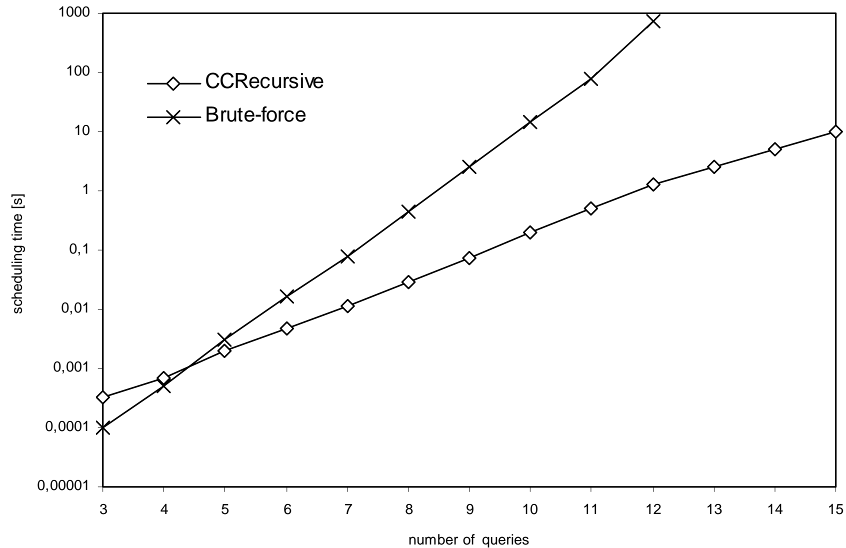

Figure 8. Execution times (logarithmic scale) of CCRecursive and the brute-force scheduling algorithm.

The results of conducted experiments show that *CCRecursive* significantly outperforms the brute-force solution (with the exception of cases with 3 and 4 queries when execution times of both algorithms are negligible), which makes it applicable for larger batches of data mining queries. We believe that the accuracy of our heuristics (shown in Fig. 7) is acceptable. However, it should be noted that the actual trade-off between extra disk accesses (introduced by the heuristics) and reduction in the scheduling time cannot be assessed without knowing the database size and hardware parameters.

5. Concluding Remarks

In this paper we addressed the problem of *common counting* of candidate itemsets for multiple data mining queries. We have formally defined the problem of *data mining query scheduling*, which consists in splitting the set of data mining queries into subsets (phases) such that the candidate hash trees can fit in limited memory and the overall I/O cost is minimized.

Since the number of possible schedules growth rapidly with the number of queries, we proposed a heuristic scheduling algorithm, called *CCRecursive*. The experiments show that our heuristics generates schedules that are close to optimal and is more efficient than the brute-force solution and thus applicable for much greater number of 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] 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.

[4] 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.

[5] 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.

[6] Imielinski T., Mannila H. A Database Perspective on Knowledge Discovery. Communications of the ACM, Vol. 39, No. 11, 1996.

[7] Imielinski T., Virmani A., Abdulghani A. Datamine: Application programming interface and query language for data mining. Proc. of the 2nd KDD Conference, 1996.

[8] Morzy T., Wojciechowski M., Zakrzewicz M. Data Mining Support in Database Management Systems. Proc. of the 2nd DaWaK Conference, 2000.

[9] Morzy T., Zakrzewicz M. SQL-like Language for Database Mining. ADBIS'97 Symposium, 1997.

[10] 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.

[11] 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.

[12] Toivonen H. Sampling Large Databases for Association Rules. Proc. of the 22nd Int'l Conference on Very Large Data Bases, 1996.

[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] Wojciechowski M., Zakrzewicz M. Evaluation of Common Counting Method for Concurrent Data Mining Queries. Proc. of the 7th ADBIS Conference, 2003.