On Multiple Query Optimization in Data Mining*

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. Traditional multiple query optimization methods focus on identifying common subexpressions in sets of relational queries and on constructing their global execution plans. In this paper we consider the problem of optimizing sets of data mining queries submitted to a Knowledge Discovery Management System. We describe the problem of data mining query scheduling and we introduce a new algorithm called *CCAgglomerative* to schedule data mining queries for frequent itemset discovery.

1 Introduction

Multiple Query Optimization (MQO) [10] is a database research area which focuses on optimizing a set of queries together by executing their common subexpressions once in order to save execution time. The main tasks in MQO are common subexpression identification and global execution plan construction. When common subexpressions have been identified, they can be executed just once and materialized for all the queries, instead of being executed once for each query. A specific type of a query is a Data Mining Query (DMQ) [7], describing a data mining task. It defines constraints on the data to be mined and constraints on the patterns to be discovered. DMQs are submitted for execution to a Knowledge Discovery Management System KDDMS [7], which is a DBMS extended with data mining functions. Traditional KDDMSs execute DMQs serially and do not try to share any common subexpressions.

DMQs are often processed in batches of 10-100 queries. Such queries may show many similarities about data or pattern constraints. If they are executed serially, it is likely that many I/O operations are wasted because the same database blocks may be required by multiple DMQs. If I/O steps of different DMQs were integrated and performed once, then we would be able to decrease the overall execution cost of the whole batch. Traditional MQO methods are not applicable to DMQs. DMQs perform huge database scans, which cannot and should not be materialized. Moreover, DMQs usually have high memory requirements that make it difficult to dynamically materialize intermediate results. One of the methods we proposed to process batches of DMQs is Apriori Common Counting (ACC), focused on frequent itemset discovery

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

queries [1]. ACC is based on Apriori algorithm [2], it integrates the phases of support counting for candidate itemsets – candidate hash trees for multiple DMQs are loaded into memory together and then the database is scanned once. Basic ACC [11] 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 schedule the DMQs into subsets, called phases [12]. The way such scheduling is done determines the overall cost of batched DMQs execution. To solve the scheduling problem, in [12] we proposed an "initial" heuristic algorithm, called *CCRecursive*.

2 Related Work

To the best of our knowledge, apart from the ACC method discussed in this paper, the only other multiple query processing scheme for frequent pattern discovery is Mine Merge, presented in one of our previous papers [13]. In contrast to ACC, Mine Merge is independent of a particular frequent itemset mining algorithm. However, it was proven very sensitive to data distribution and less predictable than ACC. A MQO technique based on similar ideas as ACC has been proposed in the context of inductive logic programming, where similar queries were combined into query packs [4].

Somewhat related to the problem of multiple data mining query optimization is reusing results of previous queries to answer a new query, which can be interpreted as optimizing processing of a sequence of queries independently submitted to the system. Methods falling into that category are: incremental mining [5], caching intermediate query results [9], and reusing materialized results of previous queries provided that syntactic differences between the queries satisfy certain conditions [3] [8].

3 Preliminaries and Problem Statement

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

Problem statement. Given 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*, i.e., 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 $V = DMQ \cup S$, $E = \{(dmq_i, s_j) \mid dmq_j \in DMQ, s_j \in S, \sigma_{\Sigma i}\mathcal{R} \cap \sigma_{sj}\mathcal{R} \neq \emptyset\}$.

Example. Consider the following example of a data sharing graph. Given 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 DMQs and boxes represent distinct selection formulas.

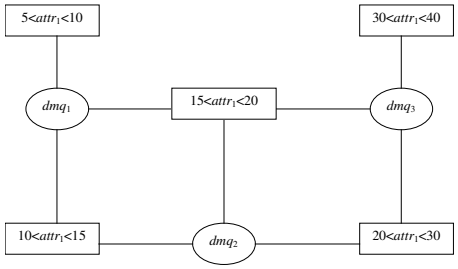

Fig. 1. Sample data sharing graph for a set of data mining queries

Apriori Common Counting (Fig. 2). ACC executes a set of data mining queries by integrating their I/O operations. First, 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.

```
for (i=1; i<=n; i++)                    /* n = number of data mining queries */
  C_1^i = {all 1-itemsets from σ_{s1∪s2∪..∪sk}R, ∀s_j∈S: (dmq_i,s_j)∈E}   /* generate 1-candidates */
for (k=1; C_k^1 ∪ C_k^2 ∪..∪ C_k^n ≠ ∅; k++) do begin
  for each s_j∈S do begin
    CC= ∪C_k^l: (dmq_l,s_j)∈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 Data Mining Query Scheduling

The basic ACC algorithm assumes that memory is unlimited and therefore the candidate hash trees for all DMQs can completely fit in memory. If, however, the memory is limited, ACC execution must be partitioned 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 will refer to the task of data mining queries partitioning as to *data mining query scheduling*.

There are several issues to be addressed when scheduling data mining queries. First of all, it is obvious that the number of data mining queries which can be included in the same phase is restricted by the actual memory size. Memory requirements of individual data mining queries are determined by sizes of their candidate hash trees, which in turn are dependent on underlying data characteristics and on candidate sizes. Since the sizes of candidate hash trees change between Apriori iterations, the scheduling should be performed at the beginning of every iteration, not only before data mining query set execution starts.

Another observation concerns the nature of ACC. Scheduling of DMQs should be based on inter-query similarities. Queries which operate on separate database partitions should be performed in separate phases, while queries which operate on significantly overlapping database partitions could benefit from being executed in the same phase. To measure the level of "overlapping" we can use cost estimation features of existing cost-based query optimizers.

A scheduling algorithm requires that sizes of candidate hash trees are known in advance. They can be estimated in two ways. We can find an upper bound for the number of candidates knowing the number of frequent itemsets from the previous Apriori iteration. Unfortunately, typical upper bounds are far from actual sizes of the candidate hash trees. Another approach is to first generate all the candidate hash trees, measure their sizes, save them to disk, schedule the data mining queries, and then load the required trees from disk. This method introduces the cost of materialization.

5 CCAgglomerative Scheduling Algorithm

The *CCAgglomerative* algorithm first transforms the data sharing graph into a *gain graph*, which contains (1) vertices being the original data mining queries and (2) two-vertex edges whose weights describe gains that can be reached by executing the connected queries in the same phase. Due to the restricted size of this paper we skip the algorithm of gain graph generation. A sample gain graph for the earlier discussed set of data mining queries is shown in Fig. 3. For example, putting the data mining queries dmq_1 and dmq_2 in the same phase will allow us to save 9000 I/O cost units.

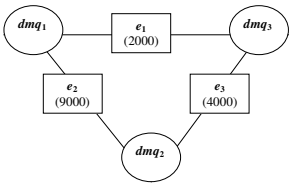

Fig. 3. Sample gain graph

An initial schedule is created by putting each data mining query into a separate phase. Next, the algorithm processes the edges sorted with respect to the decreasing weights. For each edge, the algorithm tries to combine phases containing the connected data mining queries into one phase. If the total size of all the data mining queries in such phase does not exceed the memory size, the original phases are replaced with the new one. Otherwise the algorithm simply ignores the edge and continues. The *CCAgglomerative* algorithm is shown on Fig. 4.

```
CCAgglomerative(G=(V,E), E contains 2-node edges only):
begin
  Phases ← ∅
  for each v in V do Phases ← Phases ∪ {{v}}
  sort E = {e_i , e_2 ,..., e_k} in desc. order with respect to e_i.gain, ignore edges with zero gains
  for each e_i = (v_1, v_2) in E do begin
     phase_1 ← p ∈ Phases such that v_1 ∈ p
     phase_2 ← p ∈ Phases such that v_2 ∈ p
     if treesize(phase_1 ∪ phase_2) ≤ MEMSIZE then
       Phases ← Phases – {phase_1}
       Phases ← Phases – {phase_2}
       Phases ← Phases ∪ {phase_1 ∪ phase_2}
     end if
  end
  return Phases
end
```

Fig. 4. *CCAgglomerative* Algorithm

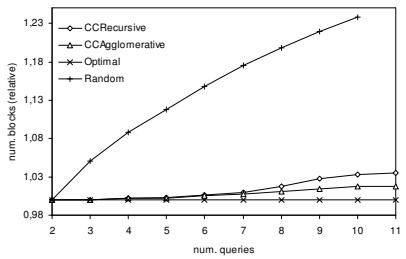

Fig. 5. Accuracy of data mining query scheduling algorithms

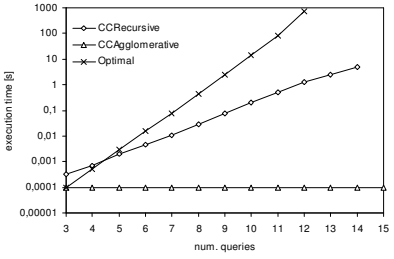

Fig. 6. Execution time of data mining query scheduling algorithms

6 Experimental Evaluation

We performed several experiments using the MSWeb dataset from the UCI KDD Archive [6]. The experiments were conducted on a PC AMD Duron 1.2 GHz with 256 MB of RAM. The datasets resided in flat files on a local disk. Memory was intentionally restricted to 10kB-50kB. Each experiment was repeated 100 times.

Fig.5 shows disk I/O costs of schedules generated by the optimal scheduling algorithm, by the *CCAgglomerative* algorithm, by the *CCRecursive* algorithm, and by a random algorithm (which randomly builds phases from queries). *CCAgglomerative* has outperformed the other heuristic approach and achieved a very good accuracy. For

example, for the set of 10 data mining queries, the *CCAgglomerative* algorithm misses the optimal solution by only 1.5%. Fig. 6 presents execution times for the optimal scheduling algorithm, *CCRecursive*, and *CCAgglomerative* (the execution time for *CCAgglomerative* includes the time required to build the gain graph). Notice that the optimal algorithm needed ca. 1000s to schedule 12 data mining queries, *CCRecursive* showed exponential execution time, while *CCAgglomerative* (polynomial wrt. the number of queries) still needed just about 0.0001s even for 15 queries.

7 Conclusions

The paper addressed the problem of optimizing sets of multiple data mining queries. We showed that in order to apply Apriori Common Counting in a restricted memory system, it is required to schedule data mining queries into separate phases. The way such scheduling is performed influences the overall cost of executing the set of data mining queries. We presented the new heuristic scheduling algorithm, called *CCAgglomerative* which significantly outperforms the other existing approach, *CCRecursive*, yet it provides a very good accuracy.

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. Baralis E., Psaila G.: Incremental Refinement of Mining Queries. Proceedings of the 1st DaWaK Conference (1999)
4. H. Blockeel, L. Dehaspe, B. Demoen, G. Janssens, J. Ramon, H. Vandecasteele: Improving the Efficiency of Inductive Logic Programming Through the Use of Query Packs, Journal of Artificial Intelligence Research, Vol. 16 (2002)
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. Hettich S., Bay S. D.: The UCI KDD Archive [http://kdd.ics.uci.edu]. Irvine, CA: (1999)
7. Imielinski T., Mannila H.: A Database Perspective on Knowledge Discovery. Communications of the ACM, Vol. 39, No. 11 (1996)
8. Morzy T., Wojciechowski M., Zakrzewicz M.: Materialized Data Mining Views. Proceedings of the 4th PKDD Conference (2000)
9. 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)
10. Sellis T.: Multiple query optimization. ACM Transactions on Database Systems, Vol. 13, No. 1 (1988)
11. Wojciechowski M., Zakrzewicz M.: Evaluation of Common Counting Method for Concurrent Data Mining Queries. Proc. of the 7th ADBIS Conference (2003)
12. 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)
13. Wojciechowski M., Zakrzewicz M.: Evaluation of the Mine Merge Method for Data Mining Query Processing. Proc. of the 8th ADBIS Conference (2004)