Improving Quality of Agglomerative Scheduling in Concurrent Processing of Frequent Itemset Queries

Pawel Boinski, Konrad Jozwiak, Marek Wojciechowski, and Maciej Zakrzewicz

Poznan University of Technology, ul. Piotrowo 2, Poznan, Poland

Abstract. Frequent itemset mining is often regarded as advanced querying where a user specifies the source dataset and pattern constraints using a given constraint model. Recently, a new problem of optimizing processing of batches of frequent itemset queries has been considered. The best technique for this problem proposed so far is Common Counting, which consists in concurrent processing of frequent itemset queries and integrating their database scans. Common Counting requires that data structures of several queries are stored in main memory at the same time. Since in practice memory is limited, the crucial problem is scheduling the queries to Common Counting phases so that the I/O cost is optimized. According to our previous studies, the best algorithm for this task, applicable to large batches of queries, is CCAgglomerative. In this paper we present a novel query scheduling method CCAgglomerativeNoise, built around CCAgglomerative, increasing its chances of finding an optimal solution.

1 Introduction

Discovery of frequent itemsets [1] is a very important data mining problem with numerous practical applications. Informally, frequent itemsets are subsets frequently occurring in a collection of sets of items. Frequent itemsets are typically used to generate association rules. However, since generation of rules is a rather straightforward task, the focus of researchers has been mostly on optimizing the frequent itemset discovery step.

Frequent itemset mining (and in general, frequent pattern mining) is often regarded as advanced querying where a user specifies the source dataset, the minimum support threshold, and optionally pattern constraints within a given constraint model [9]. A significant amount of research on efficient processing of frequent itemset queries has been done in recent years, focusing mainly on constraint handling and reusing results of previous queries [4][6][10].

Recently, a new problem of optimizing processing of batches of frequent itemset queries has been considered [14][15]. The problem was motivated by data mining systems working in a batch mode or periodically refreshed data warehouses, but is also relevant in the context of multi-user, interactive data mining environments. It is a particular case of multiple-query optimization

[13], well-studied in database systems. The goal is to find an optimal global execution plan, exploiting similarities between the queries.

One of the methods we proposed to process batches of frequent itemset queries is Common Counting [14] using Apriori [3] as a basic mining algorithm[1]. Common Counting integrates database scans performed by frequent itemset queries. It offers performance gains over sequential processing of the queries thanks to reducing the number of scans of parts of the database shared among the queries. Basic Common Counting assumes that the data structures (candidate hash-trees) of all the queries fit in memory, which may not be the case for large batches of queries, at least in initial Apriori iterations. If the memory can hold only a subset of queries, then it is necessary to schedule (assign) the queries into subsets, called phases. The way such scheduling is done determines the overall cost of batched execution of the queries.

The number of all possible assignments of queries to phases is expressed with the Bell number, which makes the complete algorithm considering all feasible assignments inapplicable for large batches of queries. Therefore, in our previous works we proposed several heuristic algorithms, the best of which was CCAgglomerative [16]. In this paper, we present a novel query scheduling method CCAgglomerativeNoise, built around CCAgglomerative, increasing its chances of finding an optimal solution. CCAgglomerativeNoise achieves its goal by iteratively randomizing the graph model on which CCAgglomerative operates.

2 Related Work

Multiple-query optimization has been extensively studied in the context of database systems (see [13] for an overview). The idea was to identify common subexpressions and construct a global execution plan minimizing the overall processing time by executing the common subexpressions only once for the set of queries. Data mining queries could also benefit from this general strategy, however, due to their different nature they require novel multiple-query processing methods.

To the best of our knowledge, apart from Common Counting, the only multiple-query processing method for data mining queries is Mine Merge [15], which is less predicable and generally offers worse performance than Common Counting. As an introduction to multiple data mining query optimization, we can regard techniques of reusing intermediate [12] or final [4][6][10][11] results of previous queries to answer a new query.

[1] It should be noted that Common Counting can be directly applied to mining other types of frequent patterns using Apriori-like algorithms. Its general idea can also be carried over to other mining paradigms like pattern-growth methods. Nevertheless, Apriori-like methods are best-suited for Common Counting as they require numerous database scans.

The need for multiple-query optimization has also been postulated in a somewhat related research area of inductive logic programming, where a technique based on similar ideas as Common Counting has been proposed, consisting in combining similar queries into query packs [5].

A problem strongly related to our query scheduling is graph partitioning [7]. In fact, the methods that we consider in this paper model the batch of queries as a graph, and thus query scheduling becomes a particular kind of graph partitioning. Nevertheless, classic graph partitioning algorithms are not applicable in our case due to different objectives of partitioning. In the classic formulation of the graph partitioning problem, the goal is to divide the graph into a given number of partitions, in such a way that the sum of weights of vertices is approximately equal in each partition, and the sum of weights of cut edges is minimized. We have a strict constraint on the sum of weights of vertices (reflecting the memory limit), and we do not care about the number of resulting partitions as long as the sum of weights of cut edges is minimized.

The method that we propose in this paper in order to improve the previously proposed algorithm CCAgglomerative is based on the same ideas as the semi-greedy heuristics proposed in [8] to improve on a greedy search strategy. Both techniques execute some basic algorithm several times and exploit randomization. The main difference is that our method randomizes the model on which the basic algorithm operates, not its individual steps as in [8].

3 Background

3.1 Basic Definitions and Problem Statement

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

The set $S = \{s_1, s_2, ..., s_k\}$ of data selection predicates over the relation $\mathcal{R}$ 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 $\sigma_{s_u}\mathcal{R} \cap \sigma_{s_v}\mathcal{R} = \emptyset$ and for each dmq_i there exist integers $a, b, ..., m$ such that $\sigma_{\Sigma_i}\mathcal{R} = \sigma_{s_a}\mathcal{R} \cup \sigma_{s_b}\mathcal{R} \cup .. \cup \sigma_{s_m}\mathcal{R}$.

Given a set of frequent itemset queries $DMQ = \{dmq_1, dmq_2, ..., dmq_n\}$, the problem of *multiple query optimization* of DMQ consists in generating such an algorithm to execute DMQ which has the lowest I/O cost.

3.2 Common Counting

Common Counting is so far the best algorithm for multiple-query optimization in frequent itemset mining. It consists in concurrent executing of a set of frequent itemset queries and integrating their I/O operations. Its implementation using the Apriori algorithm is depicted in Fig. 1.

```
**Input:** $DMQ = \{dmq_1, dmq_2, ..., dmq_n\}$, where $dmq_i = (\mathcal{R}, a, \Sigma_i, \Phi_i, \beta_i)$
**for** ($i$=1; $i \leq n$; $i$++) **do** /* $n$ = number of data mining queries */
  $\mathcal{C}_1^i$ = all possible 1-itemsets /* 1-candidates */
**for** ($k$=1; $\mathcal{C}_k^1 \cup \mathcal{C}_k^2 \cup .. \cup \mathcal{C}_k^n \neq \emptyset$; $k$++) **do begin**
  **for each** $s_j \in S$ **do begin**
    $\mathcal{CC} = \{\mathcal{C}_k^i : \sigma_{s_j}\mathcal{R} \subseteq \sigma_{\Sigma_i}\mathcal{R}\}$ /* select the candidate sets to count now */
    **if** $\mathcal{CC} \neq \emptyset$ **then** $count(\mathcal{CC}, \sigma_{s_j}\mathcal{R})$ **end**
  **for** ($i$=1; $i \leq n$; $i$++) **do begin**
    $\mathcal{F}_k^i = \{C \in \mathcal{C}_k^i : C.count \geq \beta_i\}$ /* identify frequent itemsets */
    $\mathcal{C}_{k+1}^i = generate\_candidates(\mathcal{F}_k^i)$ **end**
**end**
**for** ($i$=1; $i \leq n$; $i$++) **do**
  $Answer_i = \sigma_{\Phi_i} \bigcup_k \mathcal{F}_k^i$ /* generate responses */
```

Fig. 1. Common Counting for Apriori

The algorithm iteratively generates and counts candidates for all the data mining queries. The candidates of size 1 are all possible items. Candidates of size k (k>1) are generated from the frequent itemsets of size k-1, separately for each query. The candidate generation step (represented in the algorithm as the *generate_candidates*() function) works exactly the same way as in the original Apriori algorithm [3]. The candidates generated for each query are stored in a separate hash-tree structure, implemented according to [3].

The candidates for all the queries are counted in an integrated database scan in the following manner. For each distinct data selection formula, its corresponding database partition is scanned, and candidates for all the data mining queries referring to that partition are counted. 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.

The counting operation itself is represented in the algorithm as the *count*() function and works as follows. Sets of items from the given database partition are read one by one, and each of them is independently checked against candidate hash-trees of all relevant queries. Candidates which are found to be contained in the set of items retrieved from the database have their counters incremented.

Common Counting does not address the problem of efficient handling of selection conditions on the discovered patterns Φ, leaving any constraint-based optimizations to the basic frequent itemset mining algorithm. Since the original Apriori does not take pattern constraints into account, in the last step of Common Counting implementation for Apriori, frequent patterns discovered by all the queries are filtered according to their individual pattern selection conditions Φ_i.

3.3 Query Scheduling for Common Counting

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. We refer to the task of assigning queries to phases as to *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.

3.4 The CCAgglomerative Query Scheduling Algorithm

The exhaustive search for an optimal (minimizing I/O costs) 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). According to the previous studies, the best heuristics for query scheduling in Common Counting, both in terms of scheduling time and quality of schedules, is CCAgglomerative. CCAgglomerative represents the batch of queries in the form of a *gain graph* G=(V, E), which contains (1) vertices corresponding to the queries (with hash-tree sizes as weights of vertices) and (2) two-vertex edges whose weights describe gains (in disk blocks read) that can be reached by executing the connected queries in the same phase. A sample gain graph is shown in Fig. 2.

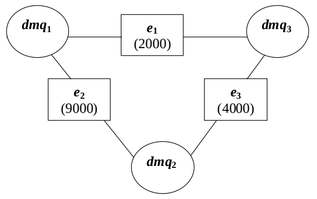

Fig. 2. Sample gain graph

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 hash-trees of all the data mining queries in such a 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 in Fig. 3.

```
Input: Gain graph $G = (V, E)$
begin
  $Phases = \emptyset$
  for each $v$ in $V$ do $Phases = Phases \cup \{\{v\}\}$
  sort $E = \{e_1, e_2, ..., e_k\}$ in desc. order with respect to $e_i.gain$,
    ignoring edges with zero gains
  for each $e_i = (v_1, v_2)$ in $E$ do begin
    $phase_1 = p \in Phases$ such that $v_1 \in p$
    $phase_2 = p \in Phases$ such that $v_2 \in p$
    if $treesize(phase_1 \cup phase_2) \leq MEMSIZE$ then
      $Phases = Phases - \{phase_1\}$
      $Phases = Phases - \{phase_2\}$
      $Phases = Phases \cup \{phase_1 \cup phase_2\}$
    end if
  end
  return $Phases$
end
```

Fig. 3. CCAgglomerative Algorithm

4 CCAgglomerativeNoise: Scheduling on a Randomized Model

Algorithm CCAgglomerative is a heuristics that suffers from the same problem as classic greedy algorithms. Merging phases connected by the heaviest edge in each iteration may not always lead to the optimal assignment of queries to phases. Let us consider an example gain graph representing a batch of queries shown in Fig. 4.

Assume that in a certain iteration of Common Counting the sizes of candidate hash-trees are 20 KB for all four queries, and the amount of available memory is 40KB, which means that no more than two queries can be processed in one phase. In such a case, CCAgglomerative would start with assigning dmq_2 and dmq_3 to the same phase, and then dmq_1 and dmq_4 would be scheduled into separate phases. The reduction in number of disk blocks read, compared to sequential execution, would be 20 blocks. Obviously, the

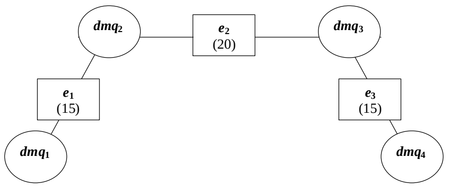

Fig. 4. Example gain graph for which CCAgglomerative misses the optimal solution

optimal solution is to execute dmq_1 and dmq_2 in one phase and dmq_3 and dmq_4 in another, leading to the gain of 30 blocks.

To give the scheduling algorithm a chance of finding an optimal assignment, we propose to randomize the graph by randomly modifying weights of graph edges within a user-specified window (expressed in percents, e.g., $\pm 10\%$), and then execute the unmodified CCAgglomerative algorithm on a modified gain graph. The procedure of randomizing the graph and schedule generation should be repeated a user-specified number of times, each time starting with the original gain graph. We call the extended scheduling algorithm CCAgglomerativeNoise as it introduces some "noise" into the graph model of the batch of queries, before performing actual scheduling. For the noise of $X\%$, in a randomized gain graph the weight $e.gain'$ of each edge e will be a random number from the range $\langle e.gain - X\% * e.gain, e.gain + X\% * e.gain \rangle$, where $e.gain$ is the original weight of the edge e.

To illustrate a potential usefulness of CCAgglomerativeNoise let us go back to the example gain graph from Fig. 4. For the noise of 20%, in each iteration of CCAgglomerativeNoise modified values of edge weights would be from the following ranges: $e_1.gain' \in \langle 12, 18 \rangle$, $e_2.gain' \in \langle 16, 24 \rangle$, and $e_3.gain' \in \langle 12, 18 \rangle$. So, it is possible that in some iteration of CCAgglomerativeNoise we would have $e_1.gain' > e_2.gain'$ or $e_3.gain' > e_2.gain'$ (e.g., $e_1.gain' = 18$, $e_2.gain' = 16$, and $e_3.gain' = 13$), in which case the basic CCAgglomerative scheduling procedure would find the optimal assignment of queries to Common Counting phases.

We should note that the CCAgglomerativeNoise method should be treated as a means of improving the results of pure CCAgglomerative. In other words, the initial iteration of CCAgglomerativeNoise should always be on the original gain graph. This way it can be guaranteed that CCAgglomerativeNoise will never generate worse schedules than CCAgglomerativeNoise.

5 Experimental Evaluation

To evaluate performance of the improved query scheduling method for Common Counting, we performed a series of experiments using a synthetic dataset generated with GEN [2] as the database. 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 experiments were conducted on a PC with AMD Athlon 1400+ processor and 384 MB of RAM, running Windows XP. The data resided in a local PostgreSQL database, the algorithms were implemented in C#.

We experimented with randomly generated batches of queries, operating on subsets of the test database, containing from 6 to 16 frequent itemset queries. To generate batches of overlapping queries we implemented our own generator, whose one of parameters was average overlapping of datasets between pairs of queries in a batch. Below we report results for the overlapping of 40% but similar relative improvements were observed for other tested levels of overlapping (20%, 60%, and 80%). The minimum support threshold for all queries in all experiments was set to 0.75%, which resulted in reasonable processing times. The average size of a hash-tree built in an Apriori iteration for this support threshold was about 30KB. Therefore, to introduce the need for query scheduling we intentionally restricted the amount of available main memory to 120KB[2].

Figure 5 shows average number of disk blocks read in an Apriori iteration for batches of queries ranging from 6 to 16 queries, and four scheduling algorithms: the optimal one, the random one, CCAgglomerative, and CCAgglomerativeNoise with 5 iterations of randomizing the gain graph with noise of 15% (the optimal algorithm did not finish in a reasonable time for batches larger than 14 queries). The experiments prove that CCAgglomerativeNoise on average generates noticeably better schedules than the original CCAgglomerative method.

The average improvement in the overall processing time of Common Counting execution in case of CCAgglomerativeNoise compared to CCAgglomerative was about 1%. To provide the scale for judging the achieved improvement, we have to mention that the difference in processing time between CCAgglomerativeNoise and the optimal complete scheduling algorithm for the case of 6 queries, where the time needed to generate the optimal schedule was approximately the same as used by CCAgglomerativeNoise, was about 1.3%.

In the experiments we also tested the impact of the amount of noise introduced into the gain graph on the quality of schedules generated by CCAgglomerativeNoise. The best results were achieved for noise between 5% and 15%, depending on the number of queries in a batch. The optimal amount of noise depends also on the distribution of edge weights in the gain graph, which represent the sizes of common parts of the database. In general, too little noise may not be enough to change the schedules generated by CCAgglomerative. On the other hand, too much noise results in the degra-

[2] Obviously, instead of just simulating the physical memory limit we could decrease the support threshold or use a more dense dataset. We opted for limiting the available memory to shorten the time needed to conduct the experiments.

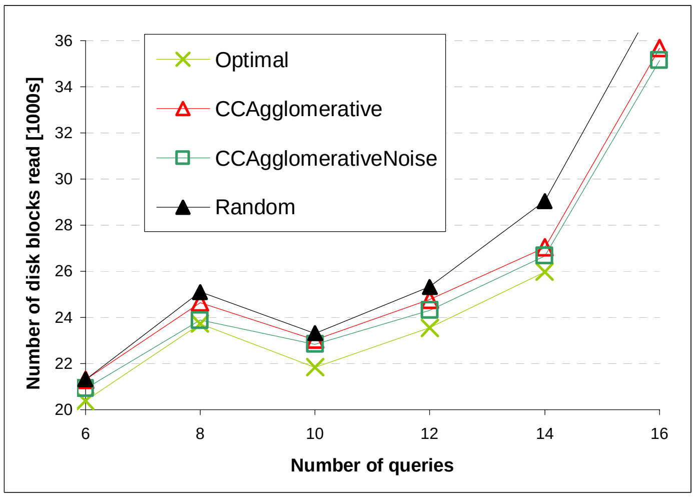

Fig. 5. Number of disk blocks read for different query scheduling algorithms (minsup=0.75%, memory limited to 120 KB, avg dataset overlapping=40%)

dation of generated schedules, as the modified gain graphs become more and more random.

6 Conclusions

The paper addressed the problem of optimizing processing of batches of frequent itemset queries by using the Common Counting scheme. Common Counting exploits dataset overlapping between the queries by processing a set of queries concurrently (keeping their data structures in main memory at the same time) and integrating their disk operations. Since in practice the amount of available main memory is limited, the queries have to be assigned (scheduled) into execution phases. The best algorithm proposed for this task so far was CCAgglomerative.

In this paper, we have presented and experimentally evaluated a novel method, called CCAgglomerativeNoise, built around CCAgglomerative that increases its chances of finding the optimal solution. In the future, we plan to investigate the possibilities of improving schedules generated by CCAgglomerative by applying some of the classic metaheuristics.

References

1. Agrawal, R., Imielinski, T., Swami, A. (1993) Mining Association Rules Between Sets of Items in Large Databases. Proceedings of the 1993 ACM SIGMOD Conference on Management of Data, Washington, D. C., 207–216

2. Agrawal, R., Mehta, M., Shafer, J., Srikant, R., Arning, A., Bollinger, T. (1996) The Quest Data Mining System. Proceedings of the 2nd International Conference on Knowledge Discovery in Databases and Data Mining, Portland, Oregon, 244–249
3. Agrawal, R., Srikant, R. (1994) Fast Algorithms for Mining Association Rules. Proceedings of the 20th International Conference on Very Large Data Bases, Santiago de Chile, Chile, 487–499
4. Baralis, E., Psaila, G. (1999) Incremental Refinement of Mining Queries. Proceedings of the 1st International Conference on Data Warehousing and Knowledge Discovery, Florence, Italy, 173–182
5. Blockeel, H., Dehaspe, L., Demoen, B., Janssens, G., Ramon, J., Vandecasteele, H. (2002) Improving the Efficiency of Inductive Logic Programming Through the Use of Query Packs. Journal of Artificial Intelligence Research **16**, 135–166
6. Cheung, D. W.-L., Han, J., Ng, V., Wong, C. Y. (1996) Maintenance of discovered association rules in large databases: An incremental updating technique. Proceedings of the 12th International Conference on Data Engineering, New Orleans, Louisiana, USA, 106–114
7. Garey, M., Johnson, D., Stockmeyer, L. (1976) Some simplified NP-complete graph problems. Theoretical Computer Science **1**(3), 237–267
8. Hart, J.P., Shogan, A.W. (1987) Semi-greedy heuristics: An empirical study. Operations Research Letters **6**, 107-114
9. Imielinski, T., Mannila, H. (1996) A Database Perspective on Knowledge Discovery. Communications of the ACM **39**(11), 58–64
10. Meo, R. (2003) Optimization of a Language for Data Mining. Proceedings of the ACM Symposium on Applied Computing - Data Mining Track, Melbourne, Florida, USA, 437–444
11. Morzy, M., Wojciechowski, M., Zakrzewicz, M. (2005) Optimizing a Sequence of Frequent Pattern Queries. Proceedings of the 7th International Conference on Data Warehousing and Knowledge Discovery, Copenhagen, Denmark, 448–457
12. Nag, B., Deshpande, P. M., DeWitt, D. J. (1999) Using a Knowledge Cache for Interactive Discovery of Association Rules. Proceedings of the 5th International Conference on Knowledge Discovery and Data Mining, San Diego, California, 244–253
13. Sellis, T. (1988) Multiple-query optimization. ACM Transactions on Database Systems **13**(1), 23–52
14. Wojciechowski, M., Zakrzewicz, M. (2003) Evaluation of Common Counting Method for Concurrent Data Mining Queries. Proceedings of 7th East European Conference on Advances in Databases and Information Systems, Dresden, Germany, 76–87
15. Wojciechowski, M., Zakrzewicz, M. (2004) Evaluation of the Mine Merge Method for Data Mining Query Processing. Proceedings of the 8th East European Conference on Advances in Databases and Information Systems, Budapest, Hungary, 78–88
16. Wojciechowski, M., Zakrzewicz, M. (2005) On Multiple Query Optimization in Data Mining. Proceedings of the 9th Pacific-Asia Conference on Knowledge Discovery and Data Mining, Hanoi, Vietnam, 696–701