ESTIMATING HASH-TREE SIZES IN CONCURRENT PROCESSING OF FREQUENT ITEMSET QUERIES

Pawel BOINSKI, **Konrad JOZWIAK**,
Marek WOJCIECHOWSKI, **Maciej ZAKRZEWICZ**

Institute of Computing Science,
Poznan University of Technology, Poznan, Poland
E-mail: Pawel.Boinski@cs.put.poznan.pl
Marek.Wojciechowski@cs.put.poznan.pl
Maciej.Zakrzewicz@cs.put.poznan.pl

Abstract

We consider the problem of optimizing the processing of batches of frequent itemset queries. One of the methods proposed for this task is Apriori Common Counting, which consists in concurrent processing of frequent itemset queries and integrating their database scans. Apriori Common Counting requires that hash-trees 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 execution phases so that the I/O cost is optimized. As the scheduling algorithm has to know the hash-tree sizes of the queries, previous approaches generated all the hash-trees before scheduling and swapped them to disk, which introduced extra I/O cost. In this paper we present a method of calculating an upper bound on the size of a hash tree, and propose to schedule the queries using estimates instead of actual hash-tree sizes.

Keywords: computer science, data mining, frequent itemsets, data mining queries, multi-query optimization

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 Apriori Common Counting [14], using Apriori [3] as a basic mining algorithm. Apriori Common Counting consists in concurrent execution of a set of frequent itemset queries and integration of their I/O operations. 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 Apriori Common Counting assumes that data structures (candidate hash-trees) of all 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 several execution phases.

The query scheduling algorithm used by Apriori Common Counting has to know the hash-tree sizes of all the queries in advance. In previous approaches (e.g., [16]) this problem was solved by generating all the hash-trees before scheduling, swapping them to disk, and reloading when necessary, which introduced extra I/O cost. In this paper, we propose to estimate the sizes of hash-trees, and run the scheduling on these estimates bounds instead of actual hash-tree sizes. To the best of our knowledge, the estimation of hash-tree sizes used by the Apriori algorithm has not been studied before. Therefore, we propose our own method of estimating hash-tree sizes, based on the upper bound on the number of candidates which can be computed from the number of frequent itemsets found in the previous Apriori iteration. We experimentally

evaluate the proposed approach using the best query scheduling algorithm for Apriori Common Counting proposed so far, called CCAgglomerative [16].

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 Apriori 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 Apriori 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. The need for multiple-query optimization has also been postulated in the somewhat related research area of inductive logic programming, where a technique based on similar ideas as Apriori 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 query scheduling algorithm that we apply in this paper models 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 do not have any balance constraint on the sizes of resulting partitions, only a strict upper bound 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.

Candidate number estimation for Apriori-like algorithms has not been an area of intense research yet. However, in [8], an interesting analytical method for finding an upper bound on the number of candidates was presented.

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.

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 parti-

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 */
  C_1^i = all possible 1-itemsets /* 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**
    $CC = \{C_k^i : \sigma_{s_j}\mathcal{R} \subseteq \sigma_{\Sigma_i}\mathcal{R}\}$ /* select the candidate sets to count now */
    if $CC \neq \emptyset$ **then** $count(CC, \sigma_{s_j}\mathcal{R})$ **end**
  for (i=1; $i \leq n$; i++) **do begin**
    $\mathcal{F}_k^i = \{C \in C_k^i : C.count \geq \beta_i\}$ /* identify frequent itemsets */
    $C_{k+1}^i = generate\_candidates(\mathcal{F}_k^i)$ **end**
end
for (i=1; $i \leq n$; i++) **do**
  $Answer_i = \sigma_{\Phi_i} \bigcup_k \mathcal{F}_k^i$ /* generate responses */

Figure 1. Apriori Common Counting

tion 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 Apriori Common Counting

Basic Apriori 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, Apriori 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 Apriori 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 Apriori 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. An initial schedule is created by putting each data

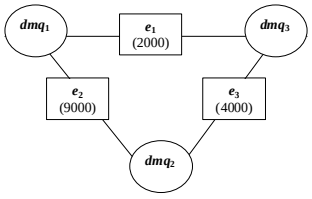

Figure 2. Sample gain graph

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.

4 Estimating Hash-Tree Sizes for Query Scheduling

Instead of generating all the hash-trees before scheduling, we propose to estimate the hash-tree sizes and generate schedules based on the estimates, in

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

Figure 3. CCAgglomerative

order to avoid the costly operations of moving the hash-trees between memory and disk. As a starting point for our estimation, we use the upper bound on the number of candidate k+1-itemsets that can be generated from a set L of frequent k-itemsets (denoted as $ub(|C_{k+1}(L)|)$), calculated using the formulas from [8].

In general, the following elements contribute to the overall size of a hash-tree: internal nodes (including the root of a tree), leaves pointing to candidate itemsets, and candidate itemsets themselves. Let x denote the order of a hash-tree, y the number of internal nodes of a hash-tree, υ the size of an internal hash-tree node, ρ the amount of space needed to store a reference to a candidate in a leaf of a hash-tree, and $\varphi(n)$ the amount of space needed to store a candidate of size n. Thus, the size of a hash-tree to store a set of candidate k+1-itemsets is:

$$size(HT_{k+1}) = y * \upsilon + |C_{k+1}(L)| * \rho + |C_{k+1}(L)| * \varphi(k+1) \qquad (1)$$

Values of υ, $\varphi(k+1)$, and ρ can be calculated for a given implementation of a hash-tree structure in a particular programming language on a particular platform. The number of internal nodes y depends on the distribution of items present in candidates itemsets and therefore it cannot be determined from the number of candidates. Nevertheless, for an upper bound on a hash-tree size we

can use the maximal possible number of internal nodes which can be derived from the order of a tree and the number of tree levels. Numbers of nodes on consecutive tree levels starting from the root form a geometric progression with first term equal to 1 and common ratio equal to the order of a tree. The number of hash-tree levels (excluding the leaf level) is equal to the current candidate size. Therefore, for candidate k+1-itemsets the maximum number of internal nodes is:

$$y_{max} = \frac{1 - x^{k+1}}{1 - x} \tag{2}$$

Incorporating Equation (2) and the upper bound on a number of candidates into Equation (1) gives us an upper bound on the hash-tree size for a set of candidate k+1-itemsets:

$$ub(size(HT_{k+1})) = \frac{1 - x^{k+1}}{1 - x} * \upsilon + ub(|C_{k+1}(L)|) * (\rho + \varphi(k+1)) \tag{3}$$

The problem with the above upper bound is that in practice actual hash-tree sizes are going to be much smaller since not all possible branches will be present in a hash-tree due to non-uniform distribution of items in candidates. To address the problem we propose to estimate the size of a hash-tree based on the estimated number of internal nodes if the number of candidates indicates that the full structure of a hash-tree is unlikely to be built. Our estimations are based on observations from a series of experiments.

First of all, we experimentally discovered the threshold value of the number of candidates above which the full hash-tree structure is built as x^{k+2}. Next, again based on observations, we have come up with the following formula to estimate the size of an incomplete hash-tree:

$$est(size(HT_{k+1})) = ub(|C_{k+1}(L)|) * (\frac{4}{3} * \upsilon + \rho + \varphi(k+1)) \tag{4}$$

In the end, as a projected hash-tree size we use:

- $ub(size(HT_{k+1}))$ if $ub(|C_{k+1}(L)|) > x^{k+2}$,
- $min(est(size(HT_{k+1})), ub(size(HT_{k+1})))$ otherwise.

Obviously, relying on the above empirical formula in query scheduling for Apriori Common Counting poses a risk that in rare cases the actual hash-trees of queries assigned to one execution phase will not fit into memory. If such a problem occurs, queries whose hash-trees do not fit into memory should be removed from the current phase, and then rescheduled together with all remaining (assigned to subsequent phases) queries.

5 Experimental Evaluation

To evaluate the impact of using estimates of hash-tree sizes on quality of schedules and overall processing time of Apriori 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 six to sixteen frequent itemset queries. The batches of queries were always generated in such a way that an average overlapping of datasets between pairs of queries in a batch was 40%. The minimum support threshold for all queries in all experiments was set to 0.75%.The average size of a hash-tree built in an Apriori iteration for this support threshold was about 30KB. To introduce the need for query scheduling we intentionally restricted the amount of available main memory to 120KB. In all the experiments we used CCAgglomerative algorithm to generate the schedules.

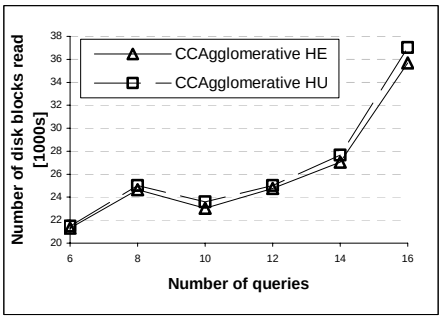

Figure 4. Number of disk blocks read

Figure 4 shows average numbers of disk blocks read by Apriori Common Counting due to generated schedules (HE denotes exact sizes of hash-trees and HU - estimates based on upper bounds on candidate numbers). As expected, since actual hash-tree sizes are typically smaller than their estimates, the schedules generated using exact sizes resulted in a noticeably smaller number of disk blocks read (on average by 3% to 4%, depending on the number of queries). However, as shown in Fig. 5, using estimates reduced the overall pro-

cessing time of Apriori Common Counting by about 10%, thanks to avoiding the costly operations of moving the hash-trees between memory and disk.

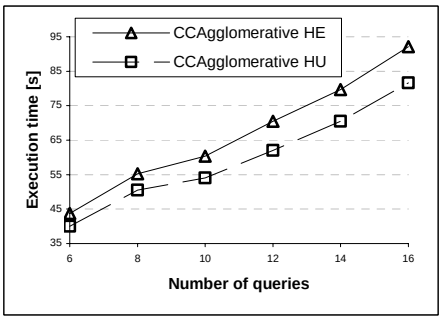

Figure 5. Execution time of data mining query scheduling algorithms

6 Conclusions

The paper addressed the problem of optimizing sets of multiple frequent itemset queries using Apriori Common Counting. Apriori Common Counting exploits dataset overlapping between the queries by processing a set of queries concurrently and integrating their disk operations. To be successful, Apriori Common Counting must keep candidate hash-trees of several queries in memory at the same time. Since in practice memory is limited, the queries have to be scheduled into execution phases. A query scheduling algorithm must be given the hash-tree sizes for all the queries. In previous approaches, the hash-trees were generated before scheduling, swapped to disk, and reloaded when needed. This method allowed the scheduling algorithm to operate on actual hash-tree sizes but introduced the hash-tree materialization step, costly in terms of time and disk space.

In this paper we have presented a method of estimating hash-tree sizes. We proposed to use these estimates instead of exact hash-tree sizes for query scheduling in Apriori Common Counting, to avoid the costly hash-tree swapping and reloading operations. The experiments show that the novel approach significantly reduces the overall processing time of Apriori Common Counting, despite the fact that the quality of generated schedules is noticeably worse.

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 International Conference on Management of Data, Washington, D.C., pp. 207-216.

[2] Agrawal R., Mehta M., Shafer J., Srikant R., Arning A., Bollinger T., 1996, *The Quest Data Mining System*, Proceedings of the Second International Conference on Knowledge Discovery and Data Mining, Portland, Oregon, pp. 244-249.

[3] Agrawal R., Srikant R., 1994, *Fast algorithms for mining association rules*, Proceedings of 20th International Conference on Very Large Data Bases, Santiago de Chile, Chile, pp. 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, pp. 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, Vol. 16 pp. 135-166.

[6] Cheung D.W., Han J., Ng V., Wong C.Y., 1996, *Maintenance of discovered association rules in large databases: An incremental updating technique*, Proceedings of the Twelfth International Conference on Data Engineering, New Orleans, Louisiana, pp. 106-114.

[7] Garey, M., Johnson, D., Stockmeyer, L., 1976, *Some simplified NP-complete graph problems*, Theoretical Computer Science, Vol. 1, No. 3, pp. 237-267.

[8] Geerts F., Goethals B., Van den Bussche J., 2001, *A tight upper bound on the number of candidate patterns*, Proceedings of the 2001 IEEE International Conference on Data Mining, San Jose, California, pp. 155-162.

[9] Imielinski T., Mannila H., 1996, *A database perspective on knowledge discovery*, Communications of the ACM, Vol. 39, No. 11, pp. 58-64.

[10] Meo R., 2003, *Optimization of a language for data mining*, Proceedings of the 2003 ACM Symposium on Applied Computing, Melbourne, Florida, pp. 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, pp. 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 Fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Diego, California, pp. 244-253.

[13] Sellis T., 1988, *Multiple-query optimization*, ACM Transactions on Database Systems, Vol. 13, No. 1, pp. 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, pp. 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, pp. 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, pp. 696-701.