INTEGRATED CANDIDATE GENERATION IN PROCESSING BATCHES OF FREQUENT ITEMSET QUERIES USING APRIORI

Piotr Jedrzejczak, Marek Wojciechowski
Institute of Computing Science, Poznan University of Technology, ul. Piotrowo 2, 60-965 Poznan, Poland
Marek.Wojciechowski@cs.put.poznan.pl

Keywords: Data mining, frequent itemsets, Apriori algorithm, data mining queries.

Abstract: Frequent itemset mining can be regarded as advanced database querying where a user specifies constraints on the source dataset and patterns to be discovered. Since such frequent itemset queries can be submitted to the data mining system in batches, a natural question arises whether a batch of queries can be processed more efficiently than by executing each query individually. So far, two methods of processing batches of frequent itemset queries have been proposed for the Apriori algorithm: Common Counting, which integrates only the database scans required to process the queries, and Common Candidate Tree, which extends the concept by allowing the queries to also share their main memory structures. In this paper we propose a new method called Common Candidates, which further integrates processing of the queries from a batch by performing integrated candidate generation.

1 INTRODUCTION

Frequent itemset discovery (Agrawal et al., 1993) is a very important data mining problem with numerous practical applications. Its goal is discovery of the most frequently occurring subsets, in a database of sets of items, called transactions.

Despite significant advances in frequent itemset mining, the most widely implemented and used in practice frequent itemset mining algorithm is the classic Apriori algorithm (Agrawal and Srikant, 1994), due to its simplicity and satisfactory performance in real-world scenarios. Apriori iteratively generates candidates (i.e., potentially frequent itemsets) from previously found smaller frequent itemsets and counts their occurrences in the database. To improve the efficiency of testing which candidates are contained in a transaction read from the database, the candidates are stored in a hash tree.

Frequent itemset mining is often regarded as advanced database querying where a user specifies the source dataset, the minimum support threshold, and optionally pattern constraints within a given constraint model (Imielinski and Mannila, 1996). A significant amount of research on efficient processing of frequent itemset queries has been done in recent years, focusing mainly on constraint handling (see e.g. (Pei and Han, 2000) for an overview) and reusing results of previous queries (Baralis and Psaila, 1999) (Meo, 2003).

Recently, a new problem of optimizing processing of sets of frequent itemset queries has been considered, bringing the concept of multiple-query optimization, the problem extensively studied in the area of database systems (see (Sellis, 1988) for an overview), to the domain of frequent itemset mining. The idea was to process the queries concurrently rather than sequentially and exploit the overlapping of queries' source datasets.

Two general approaches have been taken to design methods of processing batches of frequent itemset queries: (1) providing methods independent from a particular frequent pattern mining algorithm, and (2) tailoring dedicated methods for the most prominent frequent pattern mining algorithms with a particular emphasis on Apriori (Wojciechowski and Zakrzewicz, 2002). It has been shown that the latter approach yields more efficient algorithms than the

former, due to better sharing of computations and I/O operations among the queries forming a batch.

The first method of processing batches of frequent itemset queries proposed for Apriori was Common Counting (Wojciechowski and Zakrzewicz, 2002), which consists in concurrent execution of the queries with the integration of scans of parts of the database shared among the queries. Later, Common Counting was improved by additionally sharing the hash tree structures used to store candidates, resulting in the Common Candidate Tree method (Grudzinski and Wojciechowski, 2007). In this paper we present a new algorithm called Common Candidates, which builds on the success of Common Candidate Tree, offering further integration of computations among the queries by performing integrated candidate generation.

2 RELATED WORK

To the best of our knowledge, apart from the problem considered in this paper, multiple-query optimization for frequent pattern queries has been considered only in the context of frequent pattern mining on multiple datasets (Jin et al., 2005). The idea was to reduce the common computations appearing in different complex queries, each of which compared the support of patterns in several disjoint datasets. This is fundamentally different from our problem, where each query refers to only one dataset and the queries' datasets overlap.

Earlier, the need for multiple-query optimization has been postulated in the area of inductive logic programming, where a technique based on similar ideas as Common Counting has been proposed (Blockeel et al., 2002).

3 BASIC DEFINITIONS

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

Elementary data selection predicates. The set of elementary data selection predicates for a set of frequent itemset queries $DMQ = \{dmq_1, dmq_2, ..., dmq_n\}$ is the smallest set $S=\{s_1, s_2, ..., s_k\}$ of data selection predicates over the relation R such that for each u, v ($u \neq v$) we have $\sigma_{su}R \cap \sigma_{sv}R = \emptyset$ and for each dmq_i there exist integers $a, b, ..., m$ such that $\sigma_{\Sigma i}R = \sigma_{sa}R \cup \sigma_{sb}R \cup .. \cup \sigma_{sm}R$. The set of elementary data selection predicates represents the partitioning of the database determined by overlapping of queries' datasets.

Problem Statement. 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 an algorithm to execute DMQ that minimizes the overall processing time.

4 COMMON CANDIDATES

The only part of Apriori that is still performed separately for each query in Common Candidate Tree (CCT) is the candidate itemset generation. In order to introduce concurrency in that area, we propose a new method: Common Candidates (CCan), which makes it possible to generate candidates for all queries in a batch at once while preserving all the optimizations present in CCT. The pseudo-code for CCan is presented in Figure 1.

CCT used two representations of an itemset: a standard, single-query representation (to store the frequent itemsets and freshly generated candidates) and an extended, multiple-query one (to store the frequent itemsets inside a common hash tree). CCan abandons the former completely and stores both the frequent and candidate itemsets using the extended representation with a bitmap (*fromQuery*[]) used to indicate which queries generated a candidate itemset and then updated to show in which queries that itemset has been verified to be frequent.

The general idea of candidate generation remains identical to that of Apriori and is composed of the join phase and pruning phase. There are, however, some significant differences. Unlike all the previous methods which performed the join phase with only the itemsets from one query at a time, CCan joins all frequent itemsets from all queries simultaneously. To avoid generating candidates that do not apply to any query, only those pairs of itemsets that share at least one query are considered. After a candidate has been generated, its bitmap is calculated during the mandatory pruning phase by performing a logical AND operation on the bitmaps of all of its subsets of

Input: $DMQ = \{dmq_1, dmq_2, \ldots, dmq_n\}$, where $dmq_i = (R, a, \sum_i, \Phi_i, minsup_i)$

(1) $C_1 =$ {all possible 1-itemsets};
(2) **for** $(k = 1; C_k \neq \emptyset; k{+}{+})$ **do begin**
(3) **for each** $s_j \in S$ **do begin**
(4) $CC = \{c \in C_k : \exists i\ c.fromQuery[i] = \text{true} \wedge \sigma_{s_j} R \subseteq \sigma_{\sum_i} R\}$;
(5) **if** $CC \neq \emptyset$ **then** $count(CC, \sigma_{s_j} R)$; **end**
(6) **for each** $c \in C_k$ **do**
(7) **for** $(i = 1; i \leq n; i{+}{+})$ **do**
(8) **if** $c.counters[i] < minsup_i$ **then** $c.fromQuery[i] = false$;
(9) $F_k = \{c \in C_k : \exists i\ c.fromQuery[i] = \text{true}\}$;
(10) $C_{k+1} = generate\_candidates(F_k)$;
(11) **end**
(12) **for** $(i = 1; i \leq n; i{+}{+})$ **do** $Answer^i = \sigma_{\Phi_i} \cup_k \{f \in F_k : f.fromQuery[i] = \text{true}\}$;

Figure 1: Common Candidates.

size 1 less. The resulting bitmap has its bits set only for those queries in which all of the subsets are frequent (queries that the candidate actually applies to), and candidates with an empty bitmap are automatically pruned. As the candidates generated using this method already use the extended itemset representation, they can be stored inside a common hash tree without any merging or conversion.

The advantage of the integrated candidate generation of CCan as compared to CCT is two-fold. Firstly, each candidate is generated only once, no matter how many queries it applies to. Secondly, there is no need to convert the itemsets between the standard and extended representations, as the latter is used in both the generation and count phases.

5 EXPERIMENTAL RESULTS

In order to evaluate the performance of CCan compared to CCT, we conducted a series of experiments on a synthetic dataset generated with GEN (Agrawal et al., 1996) using the following settings: number of transactions = 1000000, average number of items in a transaction = 8, number of different items = 1000, number of patterns = 1500, average pattern length = 4. The dataset was stored in an index-organized table inside an Oracle database to facilitate efficient access to its fragments processed by frequent itemset queries. The experiments were carried out on a Mac with 2.2 GHz Intel Core 2 Duo processor and 4 GB of memory, running Snow Leopard; the database was deployed on a PC with Athlon 64 3800+ processor and 2 GB or memory, running SuSE Linux.

In the experiments we varied the level of overlapping between the queries and the number of queries in a batch. Each query referred to a dataset containing 100000 subsequent transactions from the generated dataset. The support threshold of 0.7%, which resulted in 7-8 Apriori iterations, was used for all queries. In addition to measuring total execution times for both algorithms, we also measured the time spent on candidate generation which is the target of optimizations introduced in CCan.

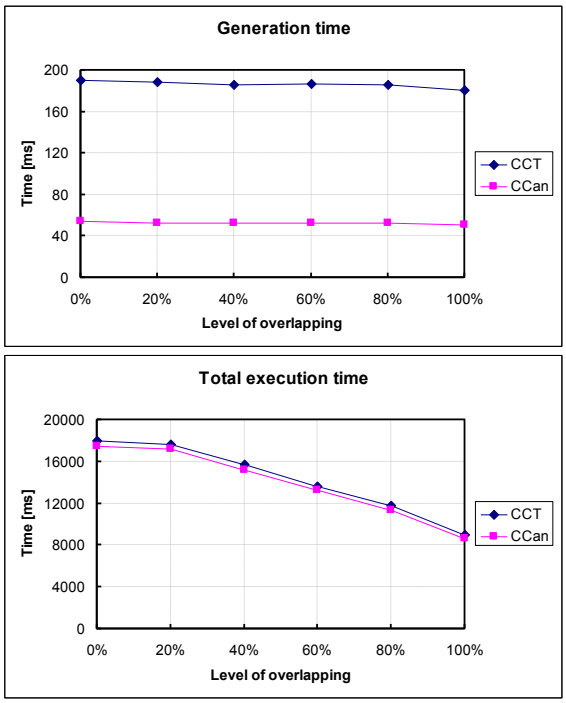

Figure 2: Generation and total execution times for two queries and different levels of overlapping.

The goal of the first experiment was to examine how the level of overlapping between the queries affects the generation and total execution times of

CCan compared to CTT. The batch used in this experiment consisted of two queries. Obtained results are shown in Figure 2.

The generation times of both CCT and CCan remain almost constant regardless of the level of overlapping, with CCan significantly outperforming CCT. The difference in total execution times is less significant, due to the fact that candidate occurrence counting is considerably more time consuming than candidate generation in Apriori-based methods.

Second of the conducted experiments examined how well the algorithms scale with the increasing number of concurrently executed queries. In order to keep the queries equally similar, the level of overlapping between each pair of subsequent queries inside the batch was fixed at 75%. As can be seen in Figure 3, the generation time of CCT grows linearly with the increase of the number of queries in a batch, while CCan remains largely insensitive. Total execution times increase similarly for both methods, with CCan performing slightly better, especially with more queries in a batch.

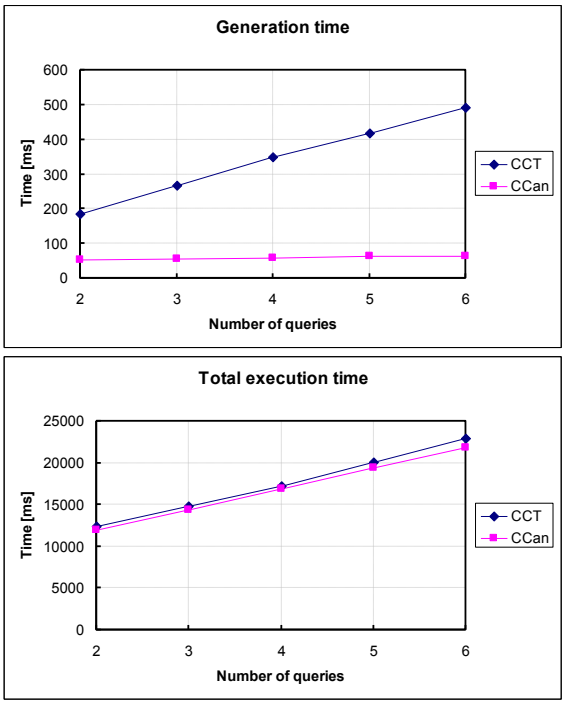

Figure 3: Generation and total execution times for different numbers of similar queries.

6 CONCLUSIONS

In this paper we addressed the problem of efficient processing of batches of frequent itemset queries in the context of the Apriori algorithm. We proposed a new algorithm, called Common Candidates, built upon Common Candidate Tree, offering further integration of computations performed for a batch of queries thanks to the integrated candidate generation procedure.

The conducted experiments showed that the new method results in significant reduction of the total time spent on candidate generation. The impact of the integrated candidate generation procedure on the overall execution time is less spectacular but still noticeable.

In the future we plan to investigate the possible impact of several optimizations applied to Apriori by its practical implementations on our batch processing algorithms.

REFERENCES

Agrawal, R., Imielinski, T., Swami, A., 1993. Mining Association Rules Between Sets of Items in Large Databases, In *Proc. of the 1993 ACM SIGMOD Conf.*

Agrawal, R., Mehta, M., Shafer, J., Srikant, R., Arning, A., Bollinger, T., 1996. The Quest Data Mining System, In *Proc. of the 2nd KDD Conference*.

Agrawal, R., Srikant, R., 1994. Fast Algorithms for Mining Association Rules, In *Proc. of the 20th VLDB Conference*.

Baralis, E., Psaila, G.,1999. Incremental Refinement of Mining Queries, In *Proceedings of the 1st DaWaK Conference.*

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.

Grudzinski, P., Wojciechowski, M., 2007. Integration of Candidate Hash Trees in Concurrent Processing of Frequent Itemset Queries Using Apriori, In *Proc. of the 3rd ADMKD Workshop*.

Imielinski, T., Mannila, H., 1996. A Database Perspective on Knowledge Discovery, *Communications of the ACM*, Vol. 39.

Jin, R., Sinha, K., Agrawal, G., 2005. Simultaneous Optimization of Complex Mining Tasks with a Knowledgeable Cache, In *Proc. of the 11th KDD Conference*.

Meo, R., 2003. Optimization of a Language for Data Mining, In *Proc. of the ACM SAC Conference*.

Pei, J., Han, J., 2000. Can We Push More Constraints into Frequent Pattern Mining?, In *Proc. of the 6th KDD Conference*.

Sellis, T., 1988. Multiple-query optimization, *ACM Transactions on Database Systems*, Vol. 13.

Wojciechowski, M., Zakrzewicz, M., 2002. Methods for Batch Processing of Data Mining Queries, In *Proc. of the 5th DB&IS Conference*.