Partition-Based Approach to Processing Batches of Frequent Itemset Queries

Przemyslaw Grudzinski, Marek Wojciechowski, and Maciej Zakrzewicz

Poznan University of Technology
Institute of Computing Science
ul. Piotrowo 2, 60-965 Poznan, Poland
{marek, mzakrz}@cs.put.poznan.pl

Abstract. We consider the problem of optimizing processing of batches of frequent itemset queries. The problem is a particular case of multiple-query optimization, where the goal is to minimize the total execution time of the set of queries. We propose an algorithm that is a combination of the Mine Merge method, previously proposed for processing of batches of frequent itemset queries, and the Partition algorithm for memory-based frequent itemset mining. The experiments show that the novel approach outperforms the original Mine Merge and sequential processing in majority of cases.

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 [5][7][12][13].

Recently, a new problem of optimizing processing of batches of frequent itemset queries has been considered [19][20]. 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 [18], well-studied in database systems. The goal is to find an optimal global execution plan, exploiting similarities between the queries.

So far, two methods of processing batches of frequent itemset queries have been proposed: Mine Merge [19] and Common Counting [20]. Both methods exploit the overlapping between queries' datasets to reduce the overall processing time. Unfortunately, both methods have serious limitations, which is the motivation for further research on the topic.

H. Legind Larsen et al. (Eds.): FQAS 2006, LNAI 4027, pp. 479 – 488, 2006.
© Springer-Verlag Berlin Heidelberg 2006

Common Counting consists in concurrent executing of a frequent itemset mining algorithm for the queries, and integrating dataset scans performed by the queries. Common Counting was designed to work with Apriori [3], in case of which it needs to maintain candidate hash-trees of several queries in main memory. If not all the hash-trees fit into memory, the queries have to be scheduled into phases, which degrades Common Counting's performance. Application of Common Counting to newer pattern-growth mining algorithms [8] is problematic as these algorithms store a compressed form of the database in main memory, which may be infeasible for more than one query at the same time, even for today's machines.

The idea of Mine Merge is to transform the original batch of overlapping queries into the set of intermediate non-overlapping queries operating on dataset partitions, whose boundaries are defined by the overlapping between the original queries. After executing the intermediate queries, the answers to original queries are generated using the method proposed in [17] for memory-based partitioning. Mine Merge is not bound to a particular mining algorithm and its memory requirements are not greater that those of the basic mining algorithm applied to intermediate queries. The disadvantage of Mine Merge is that it requires significant overlapping between the queries in order to compensate the extra database scan needed to consolidate the results from intermediate queries.

In this paper we propose a novel method for processing batches of frequent itemset queries, called PMM+ (Partition Mine Merge Improved), which combines disk-based partitioning of Mine Merge with memory-based partitioning of the well-known Partition algorithm from [17]. The advantage of the new method is that it requires exactly two scans of the union of source datasets of the queries forming a batch.

The paper is organized as follows. In Section 2 we review related work. Section 3 contains basic definitions regarding frequent itemset queries and reviews the Mine Merge method for processing of batches of frequent itemset queries. The motivations underlying PMM+ and the new method itself are presented in Section 4. In Section 5 we present and discuss results of experiments conducted to evaluate performance of PMM+. Section 6 contains conclusions.

2 Related Work

Multiple-query optimization has been extensively studied in the context of database systems (see [18] 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 [4][10][15]. 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, the only two multiple-query processing methods for data mining queries are Mine Merge [19] and Common Counting [20], mentioned above. Recently, the need for multiple-query optimization has been postulated in the 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 [6].

As an introduction to multiple data mining query optimization, we can regard techniques of reusing intermediate or final results of previous queries to answer a new query. Methods falling into that category that have been studied in the context of frequent itemset discovery are: incremental mining [7], caching intermediate query results [14], and reusing materialized complete [5][12][13] or condensed [11] results of previous queries provided that syntactic differences between the queries satisfy certain conditions.

Dividing the dataset into partitions fitting into main memory in order to find locally frequent itemsets using only in-memory operations, and then integrating the partial results to find globally frequent itemsets was first considered in [17], where the Partition algorithm was proposed. The most important contribution of [17] was the proof that given the dataset divided into the set of non-overlapping partitions, an itemset can be globally frequent only if it is locally frequent in at least one of the partitions. Partition used a variation of Apriori for in-memory frequent itemset mining, and its advantage over the original Apriori was that it performed exactly two database scans: one to read the partitions and one to verify which of the locally frequent itemsets are globally frequent.

In [16] another memory-based partitioning algorithm H-Mine was proposed for frequent itemset mining, outperforming Partition thanks to: (1) replacing Apriori for in-memory mining with a newly developed efficient pattern-growth algorithm H-Mine(Mem), and (2) applying some optimizations in the consolidation step.

3 Background

3.1 Basic Definitions and Problem Statement

Frequent Itemset Query. A frequent itemset query is a tuple $dmq = (R, a, \Sigma, \Phi, \beta)$, where R is a relation, a is a set-valued attribute of R, Σ is a condition involving the attributes of R, Φ is a condition involving discovered itemsets, and β 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 $\geq \beta$ (π and σ denote relational projection and selection operations respectively).

Example. Given the database relation $R_1(a_1, a_2)$, where a_2 is a set-valued attribute and a_1 is of integer type. The frequent itemset query $dmq_1 = (R_1, "a_2", "a_1>5", "|itemset|<4", 3\%)$ describes the problem of discovering frequent itemsets in the set-valued attribute a_2 of the relation R_1. The frequent itemsets with support of at least 3% and length less than 4 are discovered in the collection of records having $a_1>5$.

Elementary Data Selection Predicates. The set $S=\{s_1, s_2, ..., s_k\}$ of data selection predicates over the relation 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 $\sigma_{su}R \cap \sigma_{sv}R = \emptyset$ and for each dmq_i there exist integers $a, b, ..., m$ such that $\sigma_{\Sigma i}R = \sigma_{sa}R \cup \sigma_{sb}R \cup .. \cup \sigma_{sm}R$.

Example. Given the relation $R_1=(attr_1, attr_2)$ and three data mining queries: $dmq_1=(R_1, "attr_2", "5 <attr_1<20", \emptyset, 3)$, $dmq_2=(R_1, "attr_2", "0<attr_1<15", \emptyset, 5)$, $dmq_3=(R_1, "attr_2", "5<attr_1<15 \text{ or } 30<attr_1<40", \emptyset, 4)$. The set of elementary data

selection predicates is then $S=\{s_1="0<attr_1<5", s_2="5<attr_1<15", s_3="15<attr_1<20", s_4="30<attr_1<40"\}$.

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 such an algorithm to execute *DMQ* which minimizes the overall processing time.

3.2 Mine Merge

Similarly to Partition, Mine Merge employs the property that for a database divided into a set of disjoint partitions, an itemset which is frequent in a whole database, must also be frequent in at least one partition of it. The difference is that Partition uses memory-based partitions, determined by the amount of available main-memory, while Mine Merge operates on disk-based partitions, which are the consequence of overlapping between queries' datasets.

```
/* Generate intermediate data mining queries IDMQ = {idmq1, idmq2, ...} */
  IDMQ ←∅
  for each sj∈S do begin
    Q ← {dmqi∈DMQ | σsjR⊆σΣiR }
    intermediate_β ← min{βi | dmqi=(R, a, si, Φi, βi)∈Q}
    intermediate_Φ ←Φ1∨Φ2∨...∨Φ|Q|, ∀i=1..|Q|, dmqi=(R, a, si, Φi, βi)∈Q
    IDMQ ← IDMQ ∪ idmqj=(R, a, sj, intermediate_Φ, intermediate_β)
  end
/* Execute intermediate data mining queries */
  for each idmqi ∈ IDMQ do
    IFi ← execute(idmqi)
/* Generate results for original queries DMQ = {dmq1, dmq2, ...} */
  for each dmqi∈ DMQ do
    C^i ← {c|c∈ ∪k IFk , σskR⊆σΣiR, c.count ≥ βi, c satisfies Φi}
  for each sj∈S do begin
    CC ← { C^i | σsjR⊆σΣiR };      /* select the candidates to count now */
    if CC≠∅ then count(CC, σsjR);
  end
  for (i=1; i<=n; i++) do
    Answer^i ← {c ∈ C^i | c.count ≥ βi}   /* generate responses */
```

Fig. 1. Mine Merge method

Mine Merge first generates a set of *intermediate data mining queries*, in which each data mining query is based on a single elementary selection predicate only. The intermediate data mining queries are derived from those original data mining queries that are sharing a given elementary selection predicate. The minimum support thresholds and selection conditions on itemsets for the intermediate queries are chosen so that their results are guaranteed to include all locally frequent itemsets for all the original queries that refer to the database partition corresponding to a given intermediate query.

Next, the intermediate data mining queries are executed sequentially using any frequent itemset mining algorithm (Apriori, Partition, etc.) and then their results are merged to form global candidates for the original data mining queries. Finally, a database scan is performed to count the global candidate supports and to answer the original data mining queries. The pseudocode of the Mine Merge algorithm is shown in Fig. 1.

4 PMM+: Combining Disk-Based and Memory-Based Dataset Partitioning

The problem of the basic Mine Merge algorithm is that it requires significant overlapping between the queries in order to compensate for the extra database scan needed to consolidate the results from intermediate queries. To avoid this problem we

```
/* Generate intermediate data mining queries IDMQ = {idmq1, idmq2, ...}
 * MEM_SIZE is the memory buffer size for reading database partitions
 */
   IDMQ ←∅
   PDMQ ←∅
   for each sj∈S do begin
     Q ← {dmqi∈DMQ | σsjR⊆σΣiR }
     intermediate_β ← min{βi | dmqi=(R, a, si, Φi, βi)∈Q}
     intermediate_Φ ←Φ1∨Φ2∨... ∨Φ|Q|, ∀i=1..|Q|, dmqi=(R, a, si, Φi, βi)∈Q
     IDMQ ← IDMQ ∪ idmqj=(R, a, sj, intermediate_Φ, intermediate_β)
   end
/* Partition intermediate data mining queries to fit their σsiR in memory */
   for each idmqi ∈ IDMQ do
     PDMQ ← PDMQ ∪ partition(idmqi, MEM_SIZE)
/* Execute partitioned data mining queries */
   for each pdmqi ∈ PDMQ do begin
     read_in_partition(σsiR);
     PFi ← execute(pdmqi);
   end
/* Generate results for original queries DMQ = {dmq1, dmq2, ...} */
   for each dmqi∈ DMQ do
      C^i ← {c|c∈ ∪k PFk , σskR⊆σΣiR, c.count ≥ βi, c satisfies Φi}
   for each sj∈S do begin
      CC ← {C^i | σsjR⊆σΣiR };      /* select the candidates to count now */
      if CC≠∅ then count(CC, σsjR);
   end
   for (i=1; i<=n; i++) do
     Answer^i ← {c ∈ C^i | c.count ≥ βi}   /* generate responses */
```

Fig. 2. PMM+ method

introduce a new method called Partition Mine Merge Improved (PMM+), which additionally partitions the intermediate queries in such a way that the data they operate on can completely fit in memory. Therefore, only a single database scan is needed to execute all the intermediate queries. After the partitioned intermediate queries have been executed, another database scan is performed to generate final results for the original data mining queries. In this way, PMM+ can execute a batch of data mining queries by reading the database only two times.

The pseudocode of the Partition Mine Merge Improved algorithm is shown in Fig. 2. The actual partitioning of the intermediate data mining queries (*partition()* function) can be performed either statically, with help of database query optimizer, or dynamically, while reading the database. Partitioning of the intermediate data mining queries must guarantee that their source data - $\sigma_{si}R$ – can completely fit in memory. Only then a query will be able to discover all the frequent itemsets by using fast in-memory scans.

Partition Mine Merge Improved introduces some overhead caused by discovering locally-frequent itemsets, which are then eliminated in the final scan phase. We expect this overhead to be slightly bigger compared to the basic Mine Merge because the intermediate data mining queries operate on smaller database fragments.

5 Experimental Results

To evaluate performance of the improved batch processing method for frequent itemset queries, we performed a series of experiments using a synthetic dataset generated with GEN [2] as the database. The dataset contained 100000 transactions built from 1000 different items. All the tested batches of queries operated on this dataset. We varied the number of queries in a batch, the level of overlapping between queries' datasets, the minimum support thresholds, and the variance of dataset sizes of the queries forming a batch. The experiments were conducted on a PC with AMD Athlon 1800+ processor and 256 MB of RAM, running Windows XP. The data resided in a flat file on disk, the algorithms were implemented in C++.

We implemented PMM+ with Apriori for in-memory frequent itemset mining. In all the tests, we compared its execution time with: (1) SEQA – sequential processing using Apriori, (2) SEQP – sequential processing using Partition, (3) AMM – Mine Merge using Apriori, and (4) PMM – Mine Merge using Partition. For the methods involving in-memory mining (SEQP, PMM, and PMM+) the amount of main memory reserved for that purpose was always 10000 of the average transaction size[1].

To evaluate performance of PMM+ in various circumstances, we used 14 different batches of queries, differing in the number of queries (two or three), dataset sizes, dataset overlapping, and minimum support thresholds. The execution times for the 14 test query batches of PMM+ and four reference methods are presented in Table 1.

[1] For the ease of implementation, we actually expressed the memory limit as 10000 transactions.

Table 1. Execution times of five methods of processing batches of frequent itemset queries for 14 test query batches

Case	Queries			Execution times [in seconds]				
	from	to	minsup	SEQA	SEQP	AMM	PMM	PMM+
1	1	70000	0,01	53	25	36	19	13
	20001	100000	0,01					
	30001	60000	0,01					
2a	1	70000	0,02	24	14	28	18	12
	50001	100000	0,01					
2b	1	70000	0,015	22	13	24	17	11
	50001	100000	0,015					
3a	1	70000	0,01	44	21	36	20	14
	20001	100000	0,01					
3b	1	70000	0,02	20	16	20	16	10
	20001	100000	0,02					
4a	1	70000	0,02	15	12	20	16	10
	60001	100000	0,02					
4b	1	70000	0,05	8	10	13	14	9
	60001	100000	0,05					
5	1	70000	0,01	26	16	29	18	12
	30001	100000	0,05					
6a	1	100000	0,01	35	16	36	20	14
	10001	30000	0,01					
6b	1	100000	0,05	13	12	19	16	10
	10001	30000	0,01					
7a	1	100000	0,10	10	9	16	14	10
	10001	20000	0,01					
7b	1	100000	0,05	11	10	16	15	11
	10001	20000	0,01					
7c	1	100000	0,03	16	11	21	16	11
	10001	20000	0,01					
7d	1	100000	0,01	32	15	36	19	14
	10000	20000	0,01					

The experiments show that PMM+ is the best from all the tested methods in majority of cases. In particular, PMM+ outperformed PMM and AMM for all tested batches. The average execution time of PMM+ was by 48% shorter than the average execution time of AMM (the original Mine Merge method) and by 32% shorter than the average execution time of PMM.

The only cases in which PMM+ lost to sequential processing (by 10% in the worst case) were 4b, 7a, and 7b. In case 4b the overlapping between the queries was very small and the support threshold very high (5%). As a result, there were only 2 Apriori iterations for each query, and sequential Apriori finished first. In cases 7a and 7b there were significant differences in dataset sizes and support thresholds of the queries. As a consequence, one of the queries completed very quickly, not leaving

much space for I/O cost reduction with such a small part of the database shared between the queries.

In the next series of experiments we thoroughly evaluated the impact of the overlapping between the queries and the minimum support threshold on the performance of PMM+. This time we tested only batches of two queries operating on the datasets of equal size of 50000 transactions. We changed the overlapping between the queries (expressed as the percentage of transactions shared by the queries) from 0% to 100%, repeating the experiments for the minimum support thresholds of 1%, 3%, and 5%. Figures 3, 4, and 5 present execution times measured for PMM+ and four reference algorithms.

The experiments show that the processing time of PMM+ reduces linearly as the overlapping between queries' datasets increases. More importantly, the usability of PMM+ (i.e., its advantage over sequential processing) has improved significantly

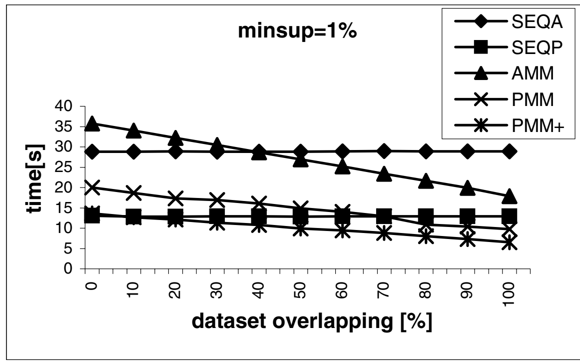

Fig. 3. Execution times for different levels of overlapping between two queries (minsup = 1%)

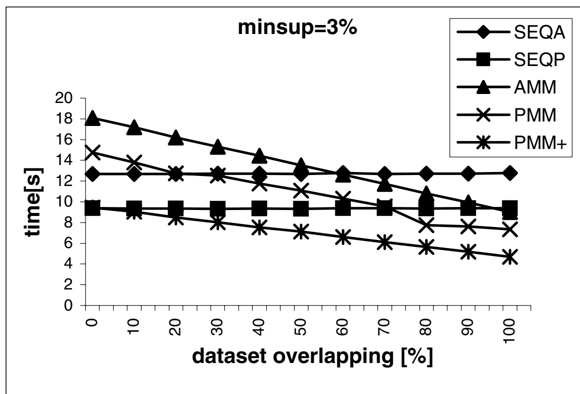

Fig. 4. Execution times for different levels of overlapping between two queries (minsup = 3%)

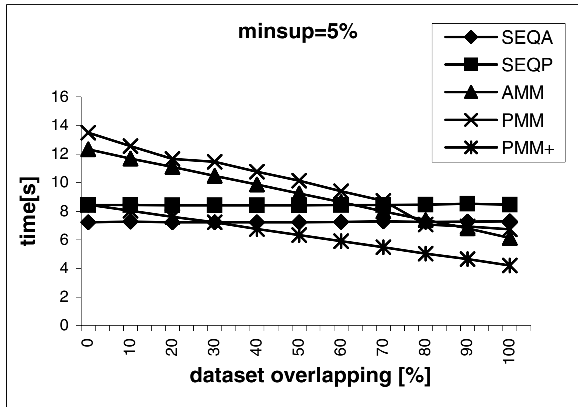

Fig. 5. Execution times for different levels of overlapping between two queries (minsup = 5%)

compared to the original Mine Merge. To outperform SEQA, AMM required at least 50% of overlapping for the supports of 1% and 3%, and 80% for the support of 5%. PMM+ was the most efficient of the tested methods if any overlapping between the queries occurred for the supports of 1% and 3%. PMM+ lost to SEQA only for the support of 5% and dataset overlapping less than 30%. This can be explained by the fact that for the support of 5% Apriori needed only 2 iterations to execute each of the queries. Nevertheless, even for such a high minimum support threshold, PMM+ was the most efficient method already starting with the dataset overlapping of 30%.

6 Conclusions

In this paper we considered the problem of optimizing batches of frequent itemset queries. We presented a novel batch processing technique, improving the previously proposed Mine Merge method. The new technique, called PMM+, combines disk-based dataset partitioning of Mine Merge with memory-based partitioning of the Partition frequent itemset mining algorithm. PMM+ minimizes I/O costs by performing exactly two scans over the union of datasets of frequent itemset queries forming a batch. The experiments show that PMM+ always performs better than the original Mine Merge and outperforms sequential processing in majority of cases.

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., Mehta M., Shafer J., Srikant R., Arning A., Bollinger T.: The Quest Data Mining System. Proc. of the 2nd Int'l Conference on Knowledge Discovery in Databases and Data Mining (1996)
3. Agrawal R., Srikant R.: Fast Algorithms for Mining Association Rules. Proc. of the 20th Int'l Conf. on Very Large Data Bases (1994)

4. Alsabbagh J.R., Raghavan V.V.: Analysis of common subexpression exploitation models in multiple-query processing. Proc. of the 10th ICDE Conference (1994)
5. Baralis E., Psaila G.: Incremental Refinement of Mining Queries. Proceedings of the 1st DaWaK Conference (1999)
6. Blockeel H., Dehaspe L., Demoen B., Janssens G., Ramon J., Vandecasteele H.: Improving the Efficiency of Inductive Logic Programming Through the Use of Query Packs, Journal of Artificial Intelligence Research, Vol. 16 (2002)
7. 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)
8. Han J., Pei J., Yin Y.: Mining frequent patterns without candidate generation. Proc. of the 2000 ACM SIGMOD Conf. on Management of Data (2000)
9. Imielinski T., Mannila H.: A Database Perspective on Knowledge Discovery. Communications of the ACM, Vol. 39, No. 11 (1996)
10. Jarke M.: Common subexpression isolation in multiple query optimization. Query Processing in Database Systems, Kim W., Reiner D.S. (Eds.), Springer (1985)
11. Jeudy B., Boulicaut J-F.: Using condensed representations for interactive association rule mining. Proceedings of the 6th European Conference on Principles and Practice of Knowledge Discovery in Databases (2002)
12. Meo R.: Optimization of a Language for Data Mining. Proc. of the ACM Symposium on Applied Computing - Data Mining Track (2003)
13. Morzy T., Wojciechowski M., Zakrzewicz M.: Materialized Data Mining Views. Proceedings of the 4th PKDD Conference (2000)
14. 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)
15. Roy P., Seshadri S., Sundarshan S., Bhobe S.: Efficient and Extensible Algorithms for Multi Query Optimization. ACM SIGMOD Intl. Conference on Management of Data (2000)
16. Pei J., Han J., Lu H., Nishio S., Tang S., Yang D.: H-Mine: Hyper-Structure Mining of Frequent Patterns in Large Databases (2001)
17. Savasere A., Omiecinski E., Navathe S.: An Efficient Algorithm for Mining Association Rules in Large Databases. Proc. 21th Int'l Conf. Very Large Data Bases (1995)
18. Sellis T.: Multiple-query optimization. ACM Transactions on Database Systems, Vol. 13, No. 1 (1988)
19. Wojciechowski M., Zakrzewicz M.: Evaluation of the Mine Merge Method for Data Mining Query Processing. Proc. of the 8th ADBIS Conference (2004)
20. Wojciechowski M., Zakrzewicz M.: On Multiple Query Optimization in Data Mining. Proc. of the 9th Pacific-Asia Conference on Knowledge Discovery and Data Mining (2005)