Integration of Candidate Hash Trees in Concurrent Processing of Frequent Itemset Queries Using Apriori

Przemyslaw Grudzinski[1], Marek Wojciechowski[2]

[1] Adam Mickiewicz University
Faculty of Mathematics and Computer Science
ul. Umultowska 87, 61-614 Poznan, Poland
[2] Poznan University of Technology
Institute of Computing Science
ul. Piotrowo 2, 60-965 Poznan, Poland
marek@cs.put.poznan.pl

Abstract. In this paper we address the problem of processing of batches of frequent itemset queries using the Apriori algorithm. The best solution of this problem proposed so far is Common Counting, which consists in concurrent execution of the queries using Apriori with the integration of scans of the parts of the database shared among the queries. In this paper we propose a new method – Common Candidate Tree, offering a more tight integration of the concurrently processed queries by sharing memory data structures, i.e., candidate hash trees. The experiments show that Common Candidate Tree outperforms Common Counting in terms of execution time. Moreover, thanks to smaller memory consumption, Common Candidate Tree can be applied to larger batches of queries.

1 Introduction

Discovery of frequent patterns is a very important data mining problem with numerous practical applications. The most prominent class of frequent patterns are frequent itemsets [1], i.e., subsets frequently occurring in a collection of sets of items (called transactions). 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.

Many frequent itemset mining algorithms have been developed. The two most prominent classes of algorithms are determined by a strategy of traversing the pattern search space. Level-wise algorithms, represented by the classic Apriori algorithm [3], follow the breadth-first strategy, whereas pattern-growth methods, among which FP-growth [9] is the best known, perform the depth-first search.

Apriori starts with discovering frequent itemsets of size 1, and then iteratively generates candidates (i.e., potentially frequent itemsets) from previously found smaller frequent itemsets and counts their occurrences in a database scan. To improve efficiency of testing which candidates are contained in a transaction read from the database, the candidates are stored in a hash tree in main memory. The number of

Apriori iterations, and consequently the number of database scans, depends on the size of the largest frequent itemset to be discovered.

FP-growth, similarly to Apriori, also builds larger frequent itemsets from smaller ones but instead of candidate generation and testing, it exploits the idea of database projections. Projections are determined by frequent itemsets found so far, and patterns are grown by discovering items frequent in their projections. To facilitate efficient projections, FP-growth transforms a database into an FP-tree, which is a highly compact data structure, designed to be stored in main memory. Only two database scans are needed to build an FP-tree, and then actual mining is performed on the FP-tree, with no furthers scans of the original database.

FP-growth has been found more efficient than Apriori for low support thresholds and/or dense datasets (i.e., datasets containing numerous and long frequent itemsets). However, in real life, datasets having different characteristics are being analyzed, and there is no single algorithm best in all cases.

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 [10]. A significant amount of research on efficient processing of frequent itemset queries has been done in recent years, focusing mainly on constraint handling (see [17] for an overview) and reusing results of previous queries [5][8][14][15].

Recently, a new problem of optimizing processing of sets of frequent itemset queries has been considered, bringing the concept of multiple-query optimization 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. Sets of frequent itemset queries available for concurrent processing may arise in data mining systems operating in a batch mode or be collected within a given time window in multi-user interactive data mining environments. A motivating example from the domain of market basket analysis could be a set of queries discovering frequent itemsets from the overlapping parts of a database table containing customer transaction data from overlapping time periods.

So far, the best method of processing batches of frequent itemset queries is Common Counting, which consists in concurrent execution of the queries with the integration of scans of parts of the database shared among the queries. Common Counting has been originally designed for Apriori, in case of which dataset scans required to count candidates were integrated [21]. Later, the method was adapted to work with FP-growth, reducing the number of disk blocks read during the phase of building FP-trees for a batch of queries [20].

The Common Counting method, which optimizes only database scans, definitely does not exploit all optimization possibilities. Further integration of operations performed by concurrently processed frequent itemset queries requires techniques dedicated to particular mining algorithms, or at least families of algorithms. In this paper we propose a new method of processing of batches of frequent itemset queries using the Apriori algorithm, called Common Candidate Tree, which integrates processing of batches of queries more tightly than Common Counting by integrating memory data structures of the queries. Experiments show that Common Candidate Tree is more efficient than Common Counting. Moreover, due to better utilization of main memory, it is also applicable to larger batches of queries.

2 Related Work

Multiple-query optimization has been extensively studied in the context of database systems (see [19] for an overview). The idea was to identify common subexpressions (selections, projections, joins, etc.) and construct a global execution plan minimizing the overall processing time by executing the common subexpressions only once for the set of queries [4][11]. Many heuristic algorithms for multiple-query optimization in database systems were proposed (e.g., [18]). Data mining queries could also benefit from the general strategy of identifying and sharing common computations. However, due to their different nature they require novel multiple-query processing methods.

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 [13]. 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 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 [8], caching intermediate query results [16], and reusing materialized complete [5][14][15] or condensed [12] results of previous queries provided that syntactic differences between the queries satisfy certain conditions.

3 Multiple-Query Optimization for Frequent Itemset Queries

3.1 Basic Definitions and Problem Statement

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).

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 size less than 4 are discovered in the collection of records having $a_1>5$.

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.

Example. Given the relation $R_1=(a_1, a_2)$ and three data mining queries: $dmq_1=(R_1, "a_2", "5 \leq a_1 < 20", \emptyset, 3\%)$, $dmq_2=(R_1, "a_2", "0 \leq a_1 < 15", \emptyset, 5\%)$, $dmq_3=(R_1, "a_2", "5 \leq a_1 < 15 \text{ or } 30 \leq a_1 < 40", \emptyset, 4\%)$. The set of elementary data selection predicates is then $S=\{s_1="0 \leq a_1 < 5", s_2="5 \leq a_1 < 15", s_3="15 \leq a_1 < 20", s_4="30 \leq a_1 < 40"\}$.

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

3.2 Common Counting

Common Counting consists in concurrent execution of a set of frequent itemset queries using Apriori and integrating scans of shared parts of the database. The pseudo-code of Common Counting is presented in Fig. 1. It is assumed that $minsup_i$ thresholds are expressed as absolute numbers of transactions[1].

Input: $DMQ = \{dmq_1, dmq_2, ..., dmq_n\}$, where $dmq_i = (R, a, \Sigma_i, \Phi_i, minsup_i)$
(1) **for** (i=1; i<=n; i++)
(2) C_1^i = {all possible 1-itemsets}
(3) **for** (k=1; $C_k^1 \cup C_k^2 \cup .. \cup C_k^n \neq \emptyset$; k++) **do begin**
(4) **for each** $s_j \in S$ **do begin**
(5) $CC = \{C_k^i : \sigma_{sj}R \subseteq \sigma_{\Sigma i}R\}$
(6) **if** $CC \neq \emptyset$ **then** *count*(CC, $\sigma_{sj}R$); **end**
(7) **for** (i=1; i<=n; i++) **do begin**
(8) $F_k^i = \{C \in C_k^i \mid C.counter \geq minsup_i\}$;
(9) C_{k+1}^i = *generate_candidates*(F_k^i); **end**
(10) **end**
(11) **for** (i=1; i<=n; i++) **do** $Answer^i = \sigma_{\Phi i} \cup_k F_k^i$;

Fig. 1. Common Counting

Common Counting iteratively generates and counts candidates for all frequent itemset queries. In the first iteration, for all the queries, the set of candidates is the set of all possible items (lines 1-2). The candidates of the size[2] k (k>1) are generated from

[1] Typically, users specify the support threshold for the query in percents. In that case the relative threshold is converted to an absolute number of transactions during the first scan of the database performed by the algorithm.

[2] The size of an itemset is the number of items in it.

frequent itemsets of size *k*-1, separately for each query (lines 7-9). Generation of candidates (represented in the pseudo-code by the *generate_candidates*() function) is performed exactly as in the original Apriori algorithm [3]. The candidates generated for each query are stored in a separate hash tree implemented as in [3]. The iterative process of candidate generation and counting ends when for all the queries no further candidates can be generated (the condition in line 3).

Occurrences of candidates for all the queries are counted during one integrated database scan in the following manner: For each elementary data selection predicate, the transactions from its corresponding database partition are read one by one. For each transaction the candidates of the queries referring to the database partition being read are considered, and the counters of candidates contained in the transaction are incremented (lines 4-6). The inclusion test is performed by confronting the transaction with hash trees of all the queries referring to the database partition containing the transaction. Candidate counting is represented in the pseudo-code as the *count*() function. It should be noted that if a given elementary data selection predicate is shared by several queries, then during each candidate counting phase its corresponding database partition is read only once.

Common Counting does not handle pattern constraints Φ, but allows to use constraint handling techniques proposed for Apriori, based on modifications of the candidate generation procedure, and then filtering the discovered frequent itemsets in a post-processing phase for those constraints that cannot be handled within Apriori.

4 Common Candidate Tree

Common Counting optimizes scans of the parts of the database shared among the queries, performing other operations of the Apriori algorithm separately for each query. Aiming at the increase of computation sharing between the concurrently processed queries, we introduce a new method: Common Candidate Tree, based on the concept of using one shared hash tree structure to store candidates of all the queries. The proposed solution preserves the integration of scans of shared database regions, and additionally allows to integrate the testing of the inclusion of candidates in a transaction retrieved from the database.

The structure of a hash tree in the Common Candidate Tree method stays unchanged compared to Common Counting and the original Apriori. In order to allow the queries to share one hash tree, it is only necessary to extend the structure of a candidate so that instead of having a single counter, a candidate will be assigned a vector of counters (*counters*[]) – one counter per query. Moreover, each candidate will have a vector of Boolean flags (*fromQuery*[]) to indicate which queries generated a given candidate. The flags will be set during merging the candidate sets generated by the queries into one integrated set of candidates that then will be stored in a common hash tree.

The pseudo-code of Common Candidate Tree is depicted in Fig. 2. The difference between the new method and Common Counting is that in Common Candidate Tree an integrated candidate set is being counted instead of separate candidate sets as in Common Counting (lines 1 and 9). The new approach has two significant advantages.

Firstly, in typical situations, where the queries share many common candidates, Common Candidate Tree should require less memory as it stores each candidate only once, no matter how many queries generated it. Secondly, due to the elimination of duplicated candidates, Common Candidate Tree reduces the number of inclusion tests between candidates and transactions. Candidate generation and selection of frequent itemsets (by comparing candidate support with the minimum support threshold) are still performed separately for each query (lines 6-8). In the phase of counting candidate occurrences, during the scan of a given database partition only these candidates are taken into account that have been generated by at least one of the queries referring to that partition, and if a candidate is included in a transaction only counters for such queries are incremented (lines 3-5).

Input: $DMQ = \{dmq_1, dmq_2, ..., dmq_n\}$, where $dmq_i = (R, a, \Sigma_i, \Phi_i, minsup_i)$
(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) $CC = \{ C \in C_k : \exists i\ C.fromQuery[i] = true \wedge \sigma_{s_j}R \subseteq \sigma_{\Sigma_i}R \}$
(5) **if** $CC \neq \emptyset$ **then** *count*(CC, $\sigma_{s_j}R$); **end**
(6) **for** (i=1; i<=n; i++) **do begin**
(7) $F_k^i = \{C \in C_k \mid C.counters[i] \geq minsup_i \}$;
(8) $C_{k+1}^i = generate\_candidates(F_k^i)$; **end**
(9) $C_{k+1} = C_{k+1}^1 \cup C_{k+1}^2 \cup .. \cup C_{k+1}^n$
(10) **end**
(11) **for** (i=1; i<=n; i++) **do** $Answer^i = \sigma_{\Phi_i} \cup_k F_k^i$;

Fig. 2. Common Candidate Tree

As for importance of Common Candidate Tree as a new method of processing batches of frequents itemset queries, it should be stressed again that possible performance improvement due to tighter integration of computations is not its only advantage over Common Counting. A serious problem with Common Counting, limiting its applicability to large batches of queries, is the necessity of having hash trees of many queries present in main memory at the same time. This problem was previously solved by dividing the original set of queries into disjoint subsets and running Common Counting separately for each of the query subsets [7][22]. Common Candidate Tree uses a single hash tree, having unmodified structure of internal nodes, and only extends the structure of candidates with extra counters and flags, which should increase its applicability with no need for dividing the query set .

Similarly to Common Counting, Common Candidate Tree does not handle pattern constraints Φ, but allows to use constraint handling techniques proposed for Apriori, since the candidate generation procedure used by Apriori is not modified by Common Candidate Tree.

5 Experimental Results

In order to evaluate performance and memory consumption of Common Candidate Tree we performed a series of experiments on a synthetic dataset generated with GEN

[2]. Limited by space, we only report the results obtained on one dataset[3], generated using the following GEN 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 a flat file on a local disk. The size of this dataset was 97 MB. In experiments we compared Common Candidate Tree with Common Counting, which is the best method so far, and sequential execution as the natural reference point for multiple-query processing and optimization techniques. The experiments were conducted on a PC with Athlon 1700+ processor and 512 MB of main memory, running Microsoft Windows XP.

In the experiments we varied the number of queries in a batch, the minimum support threshold, and the level of overlapping between the queries' datasets. For each query, its source dataset was a list of 500000 subsequent transactions from the generated dataset. Although neither of the methods requires this, in all the experiments all the queries to be concurrently processed used the same support threshold, so as to make the potential influence of the support threshold and the difference in performance between the tested methods easier to observe[4].

In the first series of experiments we tested the effect of the level of overlapping between the queries' datasets on execution times of Common Counting (CC) and Common Candidate Tree (CCT), compared to sequential processing (SEQ). At the same time, in order to compare main memory consumption of the CC and CCT methods we measured the size of hash trees (tree nodes + candidates). The experiments were performed for the case of two overlapping queries and two minimum support thresholds: 2% and 0.7%. The thresholds were adjusted so that they resulted in significantly different numbers of Apriori iterations (2 iterations for 2% and 7-8 iterations for 0.7%).

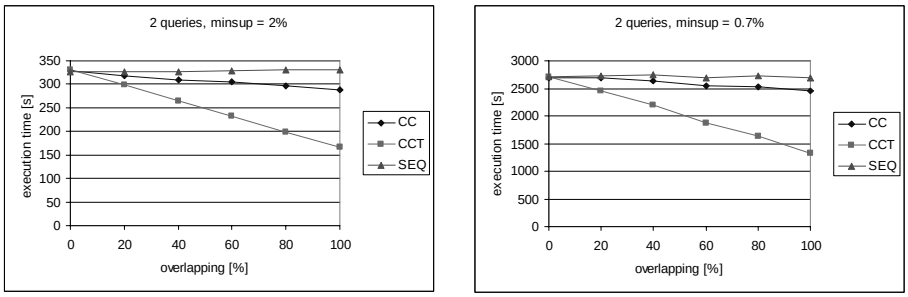

Fig. 3. Execution times for two queries and different levels of overlapping with minsup=2% (left) and minsup=0.7% (right)

[3] We repeated all the experiments on a smaller dataset having different characteristics obtaining similar results as for relative performance and memory consumption of the tested methods.

[4] The greater the difference in minimum support thresholds among the queries forming a batch, the greater the difference in number of Apriori iterations among the queries can be expected. Both Common Counting and Common Candidate Tree reduce the processing time of only those iterations in which at least 2 queries are still being processed.

Figure 3 presents execution times of the compared methods for different levels of overlapping for the case of two queries. The execution times of CC and CCT decrease linearly with the increase of the level of overlapping, with CCT significantly outperforming CC. Relative differences between the tested methods are similar for both minimum support thresholds used.

Figure 4 shows average sums of hash tree sizes for CC and CCT (computed as the sum of sizes of hash trees of all the queries from all iterations divided by the number of iterations). For the case of two queries CCT reduced average memory consumption by 33% do 40% compared to CC. Differences in memory consumption of both algorithms for different levels of overlapping, observed for the support threshold of 0.7%, were due to different characteristics of different regions of the generated dataset.

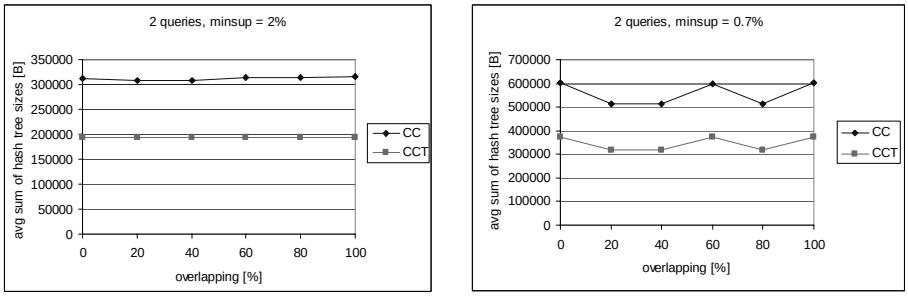

Fig. 4. Average sums of hash tree sizes for two queries and different levels of overlapping with minsup=2% (left) and minsup=0.7% (right)

The goal of the second series of experiments was to evaluate scalability of CC and CCT with respect to the number of concurrently executed queries. In general, it is hard to compare the performance of the considered methods for different numbers of queries in a batch because the more queries the more overlapping configurations possible. Therefore, in order to assess the influence of the number of queries on their performance we "benchmarked" the methods on sets of identical queries (the level of overlapping was always 100%).

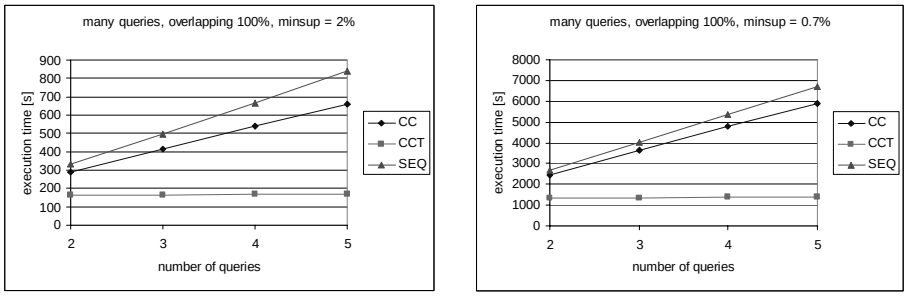

Fig. 5. Execution times for 2-5 identical queries with minsup=2% (left) and minsup=0.7% (right)

Figure 5 shows how the execution time of a batch of queries increases with the number of queries forming it. The execution time of CCT increases insignificantly with the increase of the number of queries, whereas the execution time of CC grows almost as rapidly as in the case of sequential execution of the queries.

Figure 6 presents average sums of sizes of hash trees built by CC and CCT for batches of 2 to 5 queries. With the increase of the number of queries the amount of main memory consumed by CCT grows significantly slower than in case of CC. The above experiment clearly indicates that CCT is applicable for larger batches of queries than CC, even taking into account the fact that the experiment favored CCT (since the queries forming a batch were identical, addition of another query resulted in the addition of another hash tree for CC, and only in the increase of the size of the vectors assigned to candidates in case of CCT).

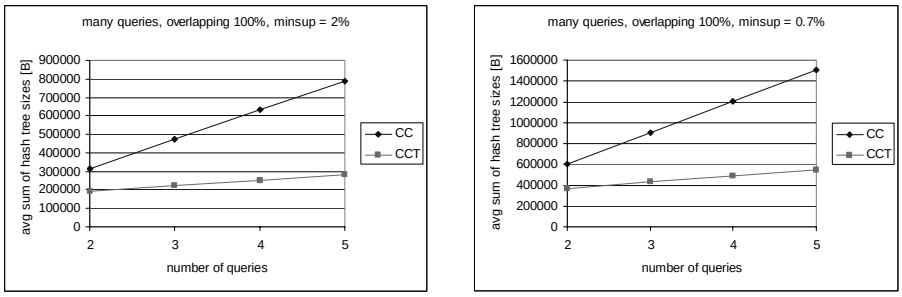

Fig. 6. Average sums of hash tree sizes for 2-5 identical queries with minsup=2% (left) and minsup=0.7% (right)

6 Conclusions

In the paper we proposed a new method of concurrent execution of the set of frequent itemset queries using the Apriori algorithm. The new method is called Common Candidate Tree because it utilizes a common hash tree structure for all the concurrently executed queries. The experiments show that in comparison with the previously proposed Common Counting method, Common Candidate Tree is much more efficient, scales better with respect to the number of queries, and consumes a smaller amount of main memory.

Currently we are working on a method analogous to the one presented in this paper, designed for FP-growth, aiming at the integration of FP-trees of the concurrently executed queries. In the future we plan to investigate further possibilities of computation sharing between the concurrently processed queries, going beyond sharing disk accesses and memory data structures.

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 KDD Conference (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. Boinski P., Wojciechowski M., Zakrzewicz M.: A Greedy Approach to Concurrent Processing of Frequent Itemset Queries. Proc. of the 8th DaWaK Conference (2006)
8. 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)
9. Han J., Pei J., Yin Y.: Mining frequent patterns without candidate generation. Proc. of the 2000 ACM SIGMOD Conf. on Management of Data (2000)
10. Imielinski T., Mannila H.: A Database Perspective on Knowledge Discovery. Communications of the ACM, Vol. 39, No. 11 (1996)
11. Jarke M.: Common subexpression isolation in multiple query optimization. Query Processing in Database Systems, Kim W., Reiner D.S. (Eds.), Springer (1985)
12. Jeudy B., Boulicaut J-F.: Using condensed representations for interactive association rule mining. Proceedings of the 6th PKDD Conference (2002)
13. Jin R., Sinha K., Agrawal G.: Simultaneous Optimization of Complex Mining Tasks with a Knowledgeable Cache. Proc. of the 11th KDD Conference (2005)
14. Meo R.: Optimization of a Language for Data Mining. Proc. of the ACM Symposium on Applied Computing - Data Mining Track (2003)
15. Morzy T., Wojciechowski M., Zakrzewicz M.: Materialized Data Mining Views. Proceedings of the 4th PKDD Conference (2000)
16. 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)
17. Pei J., Han J.: Can We Push More Constraints into Frequent Pattern Mining? Proceedings of the 6th KDD Conference (2000)
18. 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)
19. Sellis T.: Multiple-query optimization. ACM Transactions on Database Systems, Vol. 13, No. 1 (1988)
20. Wojciechowski M., Galecki K., Gawronek K.: Concurrent Processing of Frequent Itemset Queries Using FP-Growth Algorithm. Proceedings of the 1st ADMKD Workshop (2005)
21. Wojciechowski M., Zakrzewicz M.: Evaluation of Common Counting Method for Concurrent Data Mining Queries. Proc. of the 7th ADBIS Conference (2003)
22. 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)
23. Zheng Z., Kohavi R., Mason L.: Real world performance of association rule algorithms. Proc. of the 7th KDD Conference (2001)