Efficient Processing of Streams of Frequent Itemset Queries*

Monika Rokosik and Marek Wojciechowski

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

Abstract. Frequent itemset mining is one of fundamental data mining problems that shares many similarities with traditional database querying. Hence, several query optimization techniques known from database systems have been successfully applied to frequent itemset queries, including reusing results of previous queries and multi-query optimization. In this paper, we consider a new problem of processing of streams of incoming frequent itemset queries, where like in multi-query optimization a number of queries are executed together and share some of their operations, but unlike in previously considered scenarios, new queries are dynamically being added to the currently processed set of queries.

Keywords: data mining, frequent itemsets, data mining queries

1 Introduction

Frequent itemset mining is one of fundamental data mining problems, where the goal is to discover subsets frequently occurring in a collection of sets of items. The problem was introduced in the context of market basket analysis as the initial step in association rule mining [1] but quickly became the main focus of research on frequent pattern discovery. While generating association rules from discovered frequent itemsets is a relatively straightforward task, numerous frequent itemset mining algorithms have been proposed, of which Apriori [3] is the most widely implemented in practice. Apriori starts with the discovery of frequent items and then iteratively finds larger frequent itemsets using a generate-and-test strategy, exploiting the property that all subsets of a frequent itemset must also be frequent. In order to facilitate efficient counting of potentially frequent itemsets (called candidates), Apriori maintains a specialized in-memory data structure called hash tree.

Frequent itemset mining can be regarded as advanced database querying [7], and hence may benefit from optimization strategies that have previously been considered and successfully applied in the context of database management

* This work was partially supported by the Polish National Science Center (NCN), Grant No. 2011/01/B/ST6/05169

systems. A frequent itemset query contains predicates for selection of source data to be mined and a minimum support threshold. Optionally, it may also contain predicates concerning frequent itemsets to be discovered.

First solutions addressing efficient processing of frequent itemset queries focused on incorporating pattern constraints into the mining process, rather than verifying them in the post-processing step, to reduce the query execution time. Various types of patterns constraints were identified and strategies of handling them within existing pattern mining methodologies were proposed [13].

The next step in the area of frequent itemset query optimization was reusing materialized results of previous queries. It was observed that data mining is often an interactive and iterative process where users adjust constraints of their queries, and as a result, a sequence of similar data mining queries may be submitted to the system. Several result reusing schemes were proposed, exploiting various classes of differences between the queries [6][10][12].

Finally, the problem of efficient processing of sets of frequent itemset queries was considered, borrowing general ideas of computation sharing from the area of multi-query optimization in database systems. This time the motivation was to speed up execution of batches of queries that may occur mainly in data mining systems working in a batch mode. Several processing schemes were proposed for concurrent execution of sets of frequent itemset queries, broadly divided into techniques depending on (e.g. [15]) and independent of (e.g., [16]) a particular frequent itemset mining algorithm.

In this paper we shift the focus back on interactive data mining systems. Reusing results of previous queries, which is targeted at such systems, has several important limitations. Firstly, any given query may benefit from the results of only these queries which have completed earlier and whose results have been materialized. Secondly, specific relationships between the source datasets and pattern constraints of the queries must hold for one query to be able to consume the results of another query.

On the other hand, techniques of processing of sets of frequent itemset queries try to exploit any overlapping between the queries' datasets but their application to streams of queries is problematic. It was postulated that an interactive system could group queries from a given time window in order to process them together but clearly such a strategy, while possibly beneficial from the point of view of utilizing the system's resources, may lead to postponing some queries with no actual benefit.

Motivated by the shortcomings of existing solutions, we propose to handle streams of frequent itemset queries in a similar manner to sets of such queries, i.e., trying to benefit from any overlapping among the datasets of the queries currently available to the data mining system, but allowing new queries to join the batch currently being executed without waiting for it to complete. Obviously, within such an approach we are going to look for solutions by adapting existing techniques for sets of frequent itemset queries to handle "dynamic" batches of queries. In this paper, we focus on the adaptation of one of the simplest but at the

same time efficient and easy to implement technique called Common Counting [15], dedicated to Apriori.

2 Related Work

Apart from the approaches to optimizing execution of frequent itemset queries already mentioned in the introduction, the most related to the problem considered in this paper are works concerning multi-query optimization in data mining and other research domains.

Multiple-query optimization has been extensively studied in the context of database systems (see [14] 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]. In data warehousing, multiple-query optimization has been applied to speed up maintenance of the set of materialized views by exploiting common subexpressions between different view maintenance expressions [11].

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 [9]. 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.

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 Common Counting was proposed, consisting in combining similar queries into query packs [5].

3 Background and Common Counting Technique

3.1 Basic Definitions

Definition 1. Let I be a set of literals called *items*. An *itemset* X is a set of items from I ($X \subseteq I$). The *size* of the itemset is the number of items in it. An itemset of size k is called a k-itemset.

A *transaction* over I is a couple $T = \langle tid, X \rangle$, where *tid* is a transaction identifier and X is an itemset. A database $\mathcal{D}$ over I is a set of transactions over I such that each transaction has a unique identifier.

A transaction $T = \langle tid, X \rangle$ *supports* an itemset Y if $Y \subseteq X$. The *support* of an itemset Y in $\mathcal{D}$ is the number of transactions in $\mathcal{D}$ that support Y. An itemset is called frequent in $\mathcal{D}$ if its support is no less than a user-specified minimum support threshold.

Given a database $\mathcal{D}$ and a minimum support threshold *minsup*, the problem of *frequent itemset mining* consists in discovering all frequent itemsets in $\mathcal{D}$ together with their supports.

Definition 2. A *frequent itemset query* is a tuple $dmq = (\mathcal{R}, a, \Sigma, \Phi, minsup)$, where $\mathcal{R}$ is a database relation, a is a set-valued attribute of $\mathcal{R}$, Σ is a condition involving the attributes of $\mathcal{R}$ called a *data selection predicate*, Φ is a condition involving discovered itemsets called a *pattern constraint*, and *minsup* is the minimum support threshold. The result of dmq is a set of itemsets discovered in $\pi_a \sigma_\Sigma \mathcal{R}$, satisfying Φ, and having support $\geq minsup$ (π and σ denote relational projection and selection operations respectively).

Definition 3. 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 $\mathcal{R}$ such that for each u, v ($u \neq v$) 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 $\sigma_{\Sigma_i}\mathcal{R} = \sigma_{s_a}\mathcal{R} \cup \sigma_{s_b}\mathcal{R} \cup .. \cup \sigma_{s_m}\mathcal{R}$. The set of elementary data selection predicates represents the partitioning of the database determined by overlapping of queries' datasets.

3.2 Common Counting

Common Counting reduces the data retrieval costs for a batch of frequent itemset queries with respect to sequential processing by concurrent execution of a set of frequent itemset queries using Apriori and integration of scans of the shared parts of the database. The method iteratively generates and counts candidates for all frequent itemset queries. The candidates are generated separately for each query using the original procedure from the Apriori algorithm and then stored in separate hash trees. Occurrences of candidates for all the queries are counted during one integrated database scan so that if a database partition is shared by several queries, it is read only once during each candidate counting phase. Common Counting does not incorporate pattern constraints into the actual mining process, leaving them for post-processing.

Common Counting is a simple technique, optimizing only one aspect of frequent itemset query execution, i.e., data retrieval, but it has several desired properties important from the point of view of its practical applications. Firstly, it has a negligible overhead and therefore practically guarantees reduction of the overall processing time if any overlapping between the queries' datasets occurs. Secondly, it can be applied to a large number queries even if their hash trees do not fit together in memory thanks to the possibility of partitioning the set of queries and dividing candidate counting into phases [17]. Finally, it has been shown to work well regardless of the availability of efficient access paths to data partitions determined by query overlapping [8].

4 Common Counting Stream

The key to adapting Common Counting to streams of frequent itemset queries is the observation that a Common Counting iteration does not rely on the fact that all the processed queries are at the same Apriori iteration. Hence, we can

actually add a new query to the currently processed batch even if the queries previously added to it already performed one or more Apriori iterations. We formalize this idea as the Common Counting Stream technique that maintains a dynamic batch of queries and integrates their data retrieval phases. Similarly to Common Counting, Common Counting Stream works in iterations but each query has to control its own iteration counter because Common Counting Stream iterations are not aligned with the queries' Apriori iterations. The pseudo-code of Common Counting Stream is presented in Fig. 1.

Input: $DMQ = \{dmq_1, dmq_2, ..., dmq_n\}$,
where $dmq_i = (\mathcal{R}, a, \Sigma_i, \Phi_i, minsup_i)$
(1) **while** *true* **do**
(2) update DMQ
(3) S = set of elementary data selection predicates for DMQ
(4) **for** (i=1; $i \leq n$; i++) **do**
(5) **if** $ki = 1$ **then**
(6) $\mathcal{C}_{ki,i}$ = all possible 1-itemsets
(7) **else**
(8) $\mathcal{C}_{ki,i} = apriori\_gen(\mathcal{F}_{ki-1,i})$
(9) **end if**
(10) **if** $\mathcal{C}_{ki,i} = \emptyset$ **then** $Answer_i = \sigma_{\Phi_i} \bigcup_k \mathcal{F}_{k,i}$
(11) **end for**
(12) **for each** $s_j \in S$ **do**
(13) $\mathcal{CC} = \{\mathcal{C}_{ki,i} : \sigma_{s_j}\mathcal{R} \subseteq \sigma_{\Sigma_i}\mathcal{R}\}$
(14) **if** $\mathcal{CC} \neq \emptyset$ **then** $count(\mathcal{CC}, \sigma_{s_j}\mathcal{R})$
(15) **end for**
(16) **for** (i=1; $i \leq n$; i++) **do**
(17) $\mathcal{F}_{ki,i} = \{C \in \mathcal{C}_{ki,i} : C.counter \geq minsup_i\}$
(18) **end while**

Fig. 1. Common Counting Stream

Common Counting Stream works in an infinite loop (line 1). At the beginning of the loop (line 2) it updates the current batch of queries by adding new queries and removing the ones that completed in the previous iteration, and then updates the set of elementary data selection predicates (line 3). Next, the algorithm generates candidates for each query from the batch (lines 4-11) taking into account that for some queries it may be the first iteration whereas for others a later one. Generation of candidates of size greater than one (represented in the pseudo-code by the *apriori_gen*() function) is performed exactly as in the original Apriori algorithm. Current Apriori iterations are tracked individually for each query and denoted by ki in the algorithm. $\mathcal{C}_{ki,i}$ and $\mathcal{F}_{ki,i}$ denote candidates and frequent itemsets of size ki for the query dmq_i. If for a given query no further candidates can be generated, the query completes and its final results are collected (line 10).

The counting of candidates is performed exactly as in Common Counting (lines 12-17). 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 12-15). 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.

Analogously to Common Counting, in our formulation of Common Counting Stream we assumed that hash trees of all the currently processed queries fit into main memory. However, a practical implementation of Common Counting Stream should apply the same strategy of dividing the counting into phases if the queries' data structures cannot be accommodated together in memory as developed for Common Counting.

5 Experimental Results

In order to evaluate efficiency of the proposed new technique of processing streams of frequent itemset queries we performed a series of experiments on synthetic data on a PC with Intel Core 2 Duo 2.4GHz processor and 3.5GB RAM, running Windows 7 32-bit. The compared algorithms were implemented in Java. The test dataset was prepared using the following procedure. First, we generated a small dataset using the GEN [2] generator with the following settings: number of transactions in the database = 100000, average number of items in a transactions = 8, number of different items = 1000, number of patterns = 500, average pattern length = 4. Then, we multiplied the resulting dataset 10 times, thus producing a dataset containing 1000000 transactions. formed of 10 partitions having exactly the same data distribution. Thanks to the applied procedure, when we later considered only queries selecting a number of identical partitions, we eliminated the possible impact of irregular data distribution on the obtained results. The total size of the prepared test dataset was 71MB. The dataset was stored on a hard disk as a flat file accompanied by an index facilitating selective access to data partitions.

As the goal of the proposed technique was to reduce the overall processing time of a stream of queries with respect to sequential execution of the queries, the sequential execution was chosen as a primary reference query execution method. As a secondary reference method we decided to include execution of the set of queries using the original Common Counting technique, which can be regarded as the optimal scenario where all the queries to be processed are submitted to the system at once.

In all the experiments we measured total execution times but to provide a better insight into performance gains due to sharing data reading operations between the queries we also counted the total number of transactions retrieved from the database. The problem with relying solely on execution times in assessment

of the compared query processing techniques is that the observed differences in execution times are dependent on the ratio of CPU costs to I/O costs, and the latter depend on the location of the dataset (local disk or remote database server) and its size as well as the possibility of caching the data. In our test bed the whole dataset easily fit into disk cache, while for real-life scenarios that would be unlikely. In fact, we can regard our testing environment as the worst-case scenario to observe reduction of execution times due to sharing data retrieval operations.

In the first series of experiments we tested the effect of the overlapping between the queries' datasets on efficiency of the three compared techniques: sequential execution (seq), Common Counting (cc), and Common Counting Stream (cc-s). We considered the case of two queries, each retrieving half of the dataset. With the way our input database had been generated, for the tested overlapping levels we could formulate the queries so that they always operated on datasets identical in terms of their size and contents, thus eliminating the possible effect of different data distributions on observed results. The minimum support threshold of both the queries was set to 2.1% so that they performed five Apriori iterations. The second query was added after the first one completed its second Apriori iteration.

The results of this first series of experiments are shown in Fig. 2. As expected, both the number of retrieved transactions and the total execution time of Common Counting Stream decrease linearly with the increase of overlapping. However, since the queries were processed by Common Counting Stream together only for three out of their five Apriori iterations, the performance gains are smaller than these of the reference Common Counting method, which was provided with both the queries from the beginning of its operation (this is how Common Counting was designed to operate, of course).

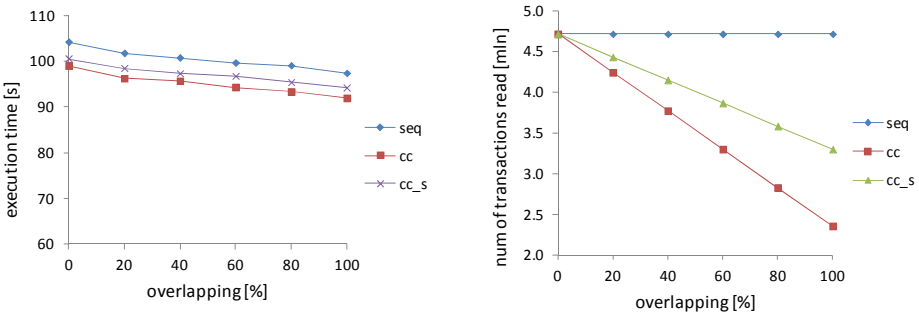

Fig. 2. Execution times (left) and numbers of transactions read (right) for different levels of overlapping between two frequent itemset queries with minsup=2.1%.

The goal of the second series of experiments was to observe the effect of iteration offset between the queries (i.e., the number of Apriori iterations performed by the

first query after which the second query was added). In this series of experiments the queries always shared 60% of their datasets. The experiments were repeated for two minimum support thresholds: 2.1% and 3%. With the increased support threshold both the queries required only four iterations to complete, i.e., one iteration less than for the threshold of 2.1%. The results are presented in Figures 3 and 4.

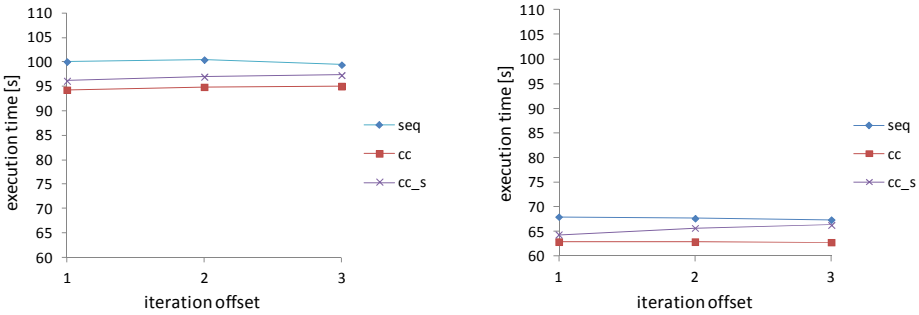

Fig. 3. Execution times for two frequent itemset queries for different iteration offsets between the two queries with minsup=2.1% (left) and minsup=3% (right)

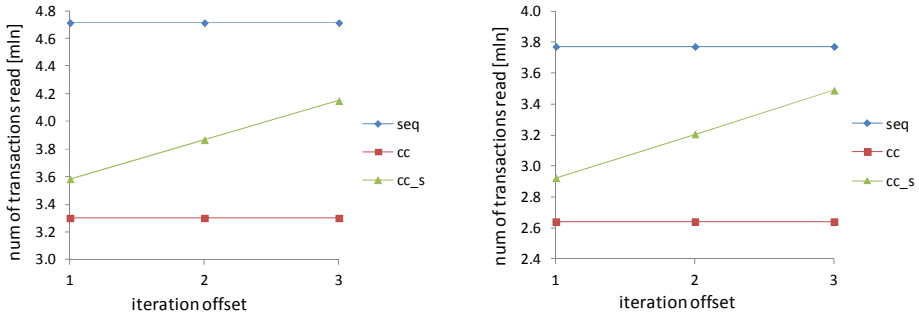

Fig. 4. Numbers of transactions read for two frequent itemset queries for different iteration offsets between the two queries with minsup=2.1% (left) and minsup=3% (right)

Obviously, the iteration offset matters only for Common Counting Stream. In sequential execution any subsequent query is postponed until the previous one completes. On the other hand, Common Counting was applied assuming that both the queries were available from the beginning. The experiments show that the smaller the iteration offset the better for the efficiency of Common Count-

ing Stream. This is certainly not surprising as the smaller this offset the more iterations can have their data scanning phases integrated for the two queries. Since we expressed the iteration offset as the number of Apriori iterations (not the percentage), its impact was more visible for the higher of considered support thresholds, for which the total number of iterations, and consequently the number of iterations where I/O integration took place, was smaller than for the lower support threshold.

In the last series of experiments we tested the three frequent itemset processing schemes on streams of two to five queries in order to evaluate scalability with the number of queries of Common Counting Stream. The streams of queries were prepared using the following set of rules: 1) Each query had the same minimum support threshold (2.1% or 3%) and equal size and contents of the source dataset by referring to 5 consecutive partitions of the database. 2) The first query's dataset started from the first partition, and each subsequent query had its dataset shifted by one partition. 3) Each subsequent query (beginning with the second one) was added after the previous one completed its second iteration.

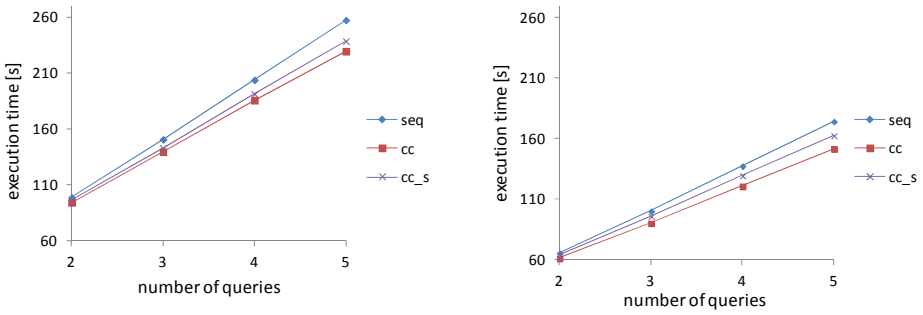

Fig. 5. Execution times for streams of two to five frequent itemset queries with minsup=2.1% (left) and minsup=3% (right)

Execution times and numbers of transactions retrieved from the database for the last series of experiments are shown in Figures 5 and 6, respectively. It can be seen that all the compared methods scale linearly with the number of queries (under the assumption that the queries are identical in terms of their support thresholds and contents of their source datasets, and additionally for Common Counting Stream the time intervals between subsequent queries are uniform). For the smaller of the considered support thresholds performance of Common Counting Stream is relatively closer to that of Common Counting than for the higher threshold for the same reason as in the second series of experiments.

The general conclusion is that Common Counting Stream is an efficient technique of processing streams of frequent itemset queries. Even in our test environment where the database resided on a local disk and its size was small enough

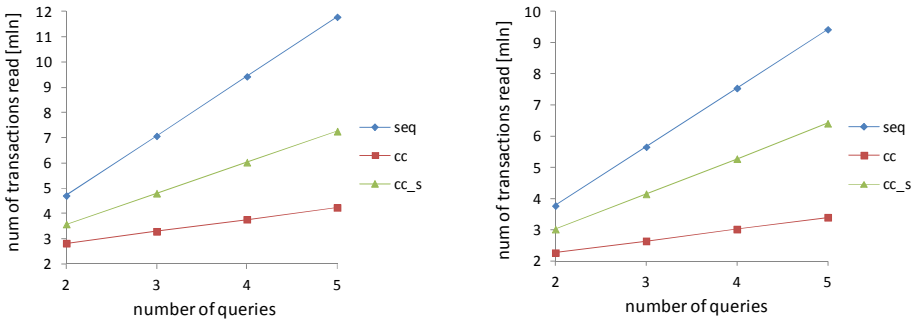

Fig. 6. Numbers of transactions read for streams of two to five frequent itemset queries with minsup=2.1% (left) and minsup=3% (right)

to fit into disk cache, Common Counting Stream noticeably outperformed sequential execution in all the conducted experiments. Moreover, detailed analysis of numbers of data transactions processed by Common Counting Stream is an indication of even more significant benefits in terms of overall processing time in production data mining systems where the source data is often remote and/or too big to fit into disk cache.

On the other hand, in all our experiments Common Counting Stream took longer to complete than Common Counting. Nevertheless, such a behavior was expected as the execution of a stream of frequent itemset queries cannot be easier than the execution of a set of the same queries. In fact, we can regard Common Counting as a specific case of Common Counting Stream where all the queries are available from the beginning, and hence the chances of sharing data retrieval operations between the queries are maximized.

Several parameters influence efficiency of Common Counting Stream. Similarly to Common Counting (and other techniques of processing sets of frequent itemset queries), Common Counting Stream's performance gains with respect to sequential processing are proportional to the level of overlapping between the queries' dataset. Other important factors contributing to the efficiency of Common Counting Stream are the minimum support threshold and time intervals between the queries (which translates to iteration offset for Common Counting Stream). In general, the lower the support threshold and the smaller the interval between query submissions, the better Common Counting Stream performs due to sharing a greater fraction of Apriori iterations between the queries.

6 Conclusions and Future Work

In this paper we addressed interactive data mining systems supporting frequent itemset discovery by means of frequent itemset queries. We claimed that existing solutions were not fully adequate for streams of queries occurring in such systems. As desired features of a processing scheme for streams of frequent itemset queries,

we listed the ability to exploit any overlapping among the queries' datasets and the possibility to add new queries to the currently processed batch of queries when some of the previous queries are still being processed.

We postulated that a natural direction in search for such processing schemes should be adaptation of existing techniques for sets of frequent itemset queries to the scenario where new queries are continually added. In this paper we presented an extension of Common Counting, a method falling into the aforementioned category, dedicated to the most popular frequent itemset mining algorithm, Apriori. The resulting technique, Common Counting Stream, applies the same data retrieval optimization method as Common Counting but maintains a dynamic batch of queries which can be at different Apriori iterations. Our experimental analysis showed that the new technique noticeably outperforms sequential execution, and may significantly reduce the I/O costs depending on the characteristics of the query stream and of the queries themselves.

A natural direction of future research is adaptation of other techniques of processing batches of frequent itemset queries to handle a stream of incoming queries dynamically added to the processed batch.

References

1. Agrawal, R., Imielinski, T., Swami, A.N.: Mining association rules between sets of items in large databases. In: Buneman, P., Jajodia, S. (eds.) Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data. pp. 207–216. ACM Press (1993)
2. Agrawal, R., Mehta, M., Shafer, J.C., Srikant, R., Arning, A., Bollinger, T.: The quest data mining system. In: Simoudis, E., Han, J., Fayyad, U.M. (eds.) Proceedings of the 2nd International Conference on Knowledge Discovery and Data Mining. pp. 244–249. AAAI Press (1996)
3. Agrawal, R., Srikant, R.: Fast algorithms for mining association rules in large databases. In: Bocca, J.B., Jarke, M., Zaniolo, C. (eds.) Proceedings of the 20th International Conference on Very Large Data Bases. pp. 487–499. Morgan Kaufmann (1994)
4. Alsabbagh, J.R., Raghavan, V.V.: Analysis of common subexpression exploitation models in multiple-query processing. In: Proceedings of the Tenth International Conference on Data Engineering. pp. 488–497. IEEE Computer Society (1994)
5. 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 16, 135–166 (2002)
6. Cheung, D.W.L., Han, J., Ng, V.T.Y., Wong, C.Y.: Maintenance of discovered association rules in large databases: An incremental updating technique. In: Su, S.Y.W. (ed.) Proceedings of the Twelfth International Conference on Data Engineering. pp. 106–114. IEEE Computer Society (1996)
7. Imielinski, T., Mannila, H.: A database perspective on knowledge discovery. Communications of the ACM 39(11), 58–64 (1996)
8. Jedrzejczak, P., Wojciechowski, M.: Data access paths in processing of sets of frequent itemset queries. In: Kryszkiewicz, M., Rybinski, H., Skowron, A., Ras, Z.W. (eds.) Foundations of Intelligent Systems - 19th International Symposium,

ISMIS 2011, Warsaw, Poland, June 28-30, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6804, pp. 376–385. Springer (2011)
9. Jin, R., Sinha, K., Agrawal, G.: Simultaneous optimization of complex mining tasks with a knowledgeable cache. In: Grossman, R., Bayardo, R.J., Bennett, K.P. (eds.) Proceedings of the 11th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. pp. 600–605. ACM (2005)
10. Meo, R.: Optimization of a language for data mining. In: Proceedings of the 2003 ACM Symposium on Applied Computing. pp. 437–444. ACM (2003)
11. Mistry, H., Roy, P., Sudarshan, S., Ramamritham, K.: Materialized view selection and maintenance using multi-query optimization. In: Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data. pp. 307–318 (2001)
12. Morzy, T., Wojciechowski, M., Zakrzewicz, M.: Materialized data mining views. In: Zighed, D.A., Komorowski, H.J., Zytkow, J.M. (eds.) Principles of Data Mining and Knowledge Discovery, 4th European Conference, PKDD 2000, Lyon, France, September 13-16, 2000, Proceedings. Lecture Notes in Computer Science, vol. 1910, pp. 65–74. Springer (2000)
13. Pei, J., Han, J.: Can we push more constraints into frequent pattern mining? In: Proceedings of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. pp. 350–354 (2000)
14. Sellis, T.K.: Multiple-query optimization. ACM Transactions on Database Systems 13(1), 23–52 (1988)
15. Wojciechowski, M., Zakrzewicz, M.: Evaluation of common counting method for concurrent data mining queries. In: Kalinichenko, L.A., Manthey, R., Thalheim, B., Wloka, U. (eds.) ADBIS. Lecture Notes in Computer Science, vol. 2798, pp. 76–87. Springer (2003)
16. Wojciechowski, M., Zakrzewicz, M.: Evaluation of the mine-merge method for data mining query processing. In: Proceedings of the 8th East European Conference on Advances in Databases and Information Systems (2004)
17. Wojciechowski, M., Zakrzewicz, M., Boinski, P.: Integration of dataset scans in processing sets of frequent itemset queries. In: Holmes, D., Jain, L. (eds.) Data Mining: Foundations and Intelligent Paradigms, Volume 1: Clustering, Association and Classification. pp. 223–266. Springer (2012)