Control and Cybernetics

vol. **38** (2009) No. 1

Integration of candidate hash trees in concurrent processing of frequent itemset queries using Apriori[*]

by

Przemysław Grudziński[1] **and Marek Wojciechowski**[2]

[1]Adam Mickiewicz University, Faculty of Mathematics and Computer Science
Umultowska 87, 61-614 Poznań, Poland
[2]Poznan University of Technology, Institute of Computing Science
Piotrowo 2, 60-965 Poznań, Poland

Abstract: Frequent itemset mining is often regarded as advanced querying where a user specifies the source dataset and pattern constraints using a given constraint model. In this paper we address the problem of processing 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.

Keywords: data mining, frequent itemset mining, data mining queries.

1. Introduction

Frequent itemset mining (Agrawal, Imielinski and Swami, 1993) is one of the fundamental data mining techniques. Its goal is discovery of all subsets whose number of occurrences in a source collection of sets (called transactions) exceeds a user-specified threshold. Typically, discovered frequent itemsets are used to generate association rules, which provide a deeper insight into associations among items contained in the database. Nevertheless, since generation of rules from frequent itemsets is relatively straightforward (Agrawal, 1994),

[*]Submitted: June 2008; Accepted: October 2008.

researchers focused on the frequent itemset discovery task. The problem of frequent itemset and association rule mining was initially formulated in the context of market-basket analysis, aiming at discovery of items frequently co-occurring in customer transactions. However, the problem quickly found numerous applications in various domains including: medicine, telecommunications, and World Wide Web.

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 (Agrawal and Srikant, 1994), follow the breadth-first strategy, whereas pattern-growth methods, among which FP-growth (Han, Pei and Yin, 2000) 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 further 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 (see Zheng, Kohavi and Mason, 2001).

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

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 query 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 (Wojciechowski and Zakrzewicz, 2003). 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 (Wojciechowski, Gałęcki and Gawronek, 2005).

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.

The paper is organized as follows. Section 2 discusses related work. Section 3 formally presents the frequent itemset mining problem and describes the Apriori algorithm. In Section 4 we review basic definitions regarding frequent itemset queries and we briefly describe the Common Counting method. In Section 5 we introduce Common Candidate Tree - a new method for concurrent processing of frequent itemset queries using Apriori. Section 6 presents experimental results. Section 7 contains conclusions and discusses future work.

2. Related work

Multiple-query optimization has been extensively studied in the context of database systems (see Sellis, 1988, 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 (Alsabbagh and Raghavan, 1994; Jarke, 1985). Many heuristic algorithms for multiple-query optimization in database systems were proposed (e.g., Roy et al., 2000). 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 (Jin, Sinha and Agrawal, 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 query 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 with Common Counting has been proposed, consisting in combining similar queries into query packs (Blockeel et al., 2002).

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 (Cheung et al., 1996), caching intermediate query results (Nag, Deshpande and DeWitt, 1999), and reusing materialized complete (Baralis and Psaila, 1999; Meo, 2003; Morzy, Wojciechowski and Zakrzewicz, 2000) or condensed (Jeudy and Boulicaut, 2002) results of previous queries provided that syntactic differences between the queries satisfy certain conditions.

3. Frequent itemset mining and Apriori algorithm

3.1. Basic definitions and problem statement

Definition 1 *Let $\mathcal{I}$ be a set of literals, called items. An itemset I is a set of items from $\mathcal{I}$ ($I \subseteq \mathcal{I}$). The size of an itemset is the number of items in it. An itemset of size k is called a k-itemset. A transaction over $\mathcal{I}$ is a couple $T = \langle tid, I \rangle$, where tid is a transaction identifier and I is an itemset. A database $\mathcal{D}$ over $\mathcal{I}$ is a set of transactions over $\mathcal{I}$ such that each transaction has a unique identifier.*

Definition 2 *A transaction $T = \langle tid, I \rangle$ supports an itemset X if $X \subseteq I$. The support of an itemset X in $\mathcal{D}$ is the number of transactions in $\mathcal{D}$ that support X. The frequency (also called relative support) of an itemset X in $\mathcal{D}$ is the support of X in $\mathcal{D}$ divided by the total number of transactions in $\mathcal{D}$.*

Definition 3 *An itemset is called frequent in $\mathcal{D}$ if its support is no less than a given minimum support threshold. (Alternatively, if a minimum frequency threshold is provided, an itemset is frequent if its frequency is no less than a given minimum frequency threshold.)*

```
Input: $\mathcal{D}$, $minsup$
(1)   $\mathcal{F}_1$ = {frequent 1-itemsets}
(2)   for ($k$=2; $\mathcal{F}_{k-1} \neq \emptyset$; $k$++) do begin
(3)     $\mathcal{C}_k$ = apriori_gen( $\mathcal{F}_{k-1}$ )
(4)     forall transactions $t \in \mathcal{D}$ do begin
(5)       $\mathcal{C}_t$ = subset($\mathcal{C}_k$, $t$)
(6)       forall candidates $c \in \mathcal{C}_t$ do
(7)         $c$.counter++
(8)     end
(9)     $\mathcal{F}_k$ = { $c \in \mathcal{C}_k | c$.counter $\geq minsup$}
(10)  end
(11)  Answer = $\bigcup_k \mathcal{F}_k$
```

Figure 1. Apriori

Problem *Given a database $\mathcal{D}$ and a minimum support threshold minsup or a minimum frequency threshold $minfreq$, the problem of frequent itemset mining consists in discovering all frequent itemsets in $\mathcal{D}$.*

In general, frequency thresholds are more convenient and informative for end-users than support thresholds. On the other hand, mining algorithms are often formulated for the minimum support threshold, which can be directly compared to the numbers of itemset occurrences in the database. Obviously, $minsup = \lceil minfreq * |D| \rceil$, so conversion between the two thresholds is possible, provided that the total number of transactions in the database is known. Therefore, the conversion can be done after the first scan of the database performed by a mining algorithm.

3.2. Algorithm Apriori

The Apriori algorithm for frequent itemset discovery is presented in Fig. 1. In the formulation of the algorithm $\mathcal{F}_k$ denotes the set of all frequent k-itemsets, and $\mathcal{C}_k$ denotes a set of potentially frequent k-itemsets, called candidates.

Apriori starts with the discovery of frequent 1-itemsets, i.e., frequent items (line 1). For this task, the first scan of the database is performed. Before making the k-th pass (for $k > 1$), the algorithm generates the set of candidates $\mathcal{C}_k$ using $\mathcal{F}_{k-1}$ (line 3). The candidate generation procedure, denoted as *apriori_gen*(), provides efficient pruning of the search space, and is described in Section 3.2.1. In the k-th database pass (lines 4-8), Apriori counts the supports of all the itemsets in $\mathcal{C}_k$. (In practice, the database pass is performed only if the set of generated candidates is not empty.) The key step of this phase of the algorithm is determining which candidates from $\mathcal{C}_k$ are contained in a transaction t, retrieved from the database. This step is denoted in the algorithm as a call to the *subset*() function, and is described in Section 3.2.2. At the end of the pass all itemsets in $\mathcal{C}_k$ with support greater than or equal to the minimum support threshold *minsup* form the set of frequent k-itemsets $\mathcal{F}_k$ (line 9). The algorithm finishes work if there are no frequent itemsets found in a given iteration (condition in line 2) and returns all the frequent itemsets found (line 11).

3.2.1. Candidate generation

The candidate generation procedure consists of two steps: the join step and the prune step. In the join step, each pair of frequent k-1-itemsets that differ only in the last item (according to lexicographical order of the items within itemsets) is joined to form a candidate. This step can be expressed in SQL in the following way:

```
insert into $\mathcal{C}_k$
select p.item$_1$, p.item$_2$, ..., p.item$_{k-1}$, q.item$_{k-1}$
from $\mathcal{F}_{k-1}$ p, $\mathcal{F}_{k-1}$ q
where p.item$_1$ = q.item$_1$, ..., p.item$_{k-2}$ = q.item$_{k-2}$, p.item$_{k-1}$ < q.item$_{k-1}$
```

In the prune step, itemsets having at least one subset that was found infrequent in the previous Apriori iteration are removed from the set of candidates:

```
forall itemsets $c \in \mathcal{C}_k$ do
  forall $k$-1-subsets $s$ of $c$ do
    if $s \notin \mathcal{F}_{k-1}$ then
      $\mathcal{C}_k = \mathcal{C}_k \setminus \{c\}$
```

The candidate generation procedure of Apriori exploits the antimonotonicity property of the support measure, which implies that for a candidate to be frequent all its subsets must also be frequent.

3.2.2. Candidate counting

In order to avoid costly testing of each candidate for inclusion in a transaction retrieved from the database, candidates $\mathcal{C}_k$ are stored in a hash tree. Leaves of a hash tree contain pointers to candidates, while interior nodes at depth d contain hash tables with pointers to nodes at depth d+1. The root of a hash tree is at depth 1.

A candidate is added to the hash tree starting from the root, and traversing the tree until a leaf is reached. At an interior node at depth d, the decision on which branch to follow is based on the result of a chosen hash function applied to the dth item of the candidate (according to the lexicographical order of items within a candidate). Initially, all nodes are created as leaves (starting with the root), and converted into interior nodes when a number of candidates stored in the node exceeds a specified threshold (as long as the depth d of the node is not greater than k).

In order to find the candidates that are contained in a transaction t using a hash tree, we start from its root node. If we are at a leaf, we check which itemsets from the leaf are contained in t and increment the counters of candidates that passed the inclusion test. If we are at an interior node and we have reached it by hashing the ith item of the transaction (according to lexicographical order), we hash on each item that comes after the ith item in t and recursively apply this procedure to the node pointed by the result of the hash function. When starting at the root node, we hash on every item in t.

4. Multiple-query optimization for frequent itemset queries

4.1. Basic definitions and problem statement

With the aim of keeping our study on processing batches of frequent itemset queries as general as possible, we are going to use a simple and general query model. The model assumes that the data to be mined are stored in a set-valued attribute of a database relation, accompanied by other attributes used to identify transactions and to allow selection of a subset of the relation as the mined dataset. Pattern constraints are represented as a general logical predicate with no details on their form or nature. Frequent itemsets are selected according to a minimum frequency threshold, which is not only generally more convenient than a minimum support threshold from a user's point of view, but also more appropriate for our model due to the possibility of selecting a subset of the relation for mining as the number of transactions in the dataset to be mined will depend on the selectivity of the specified selection condition.

Definition 4 *A frequent itemset query is a tuple $dmq = (\mathcal{R}, a, \Sigma, \Phi, minfreq)$, 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 data selection predicate, Φ is a condition involving discovered itemsets, called pattern constraint, and $minfreq$ is the minimum frequency threshold. The result of dmq is a set of itemsets discovered in $\pi_a \sigma_\Sigma \mathcal{R}$, satisfying Φ, and having frequency $\geq minfreq$ (π and σ denote relational projection and selection operations, respectively).*

Example 1 *Given the database relation $\mathcal{R}_1(a_1, a_2)$, where a_2 is a set-valued attribute and a_1 is an attribute of integer type. The frequent itemset query $dmq_1 = (\mathcal{R}_1, a_2, a_1 > 5, |itemset| < 4, 3\%)$ describes the problem of discovering frequent itemsets in the set-valued attribute a_2 of the relation $\mathcal{R}_1$. The itemsets with frequency of at least 3% and containing less than 4 items are discovered in the collection of records having $a_1 > 5$.*

Definition 5 *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.*

Example 2 *Given the relation $\mathcal{R}_1(a_1, a_2)$ and three frequent itemset queries: $dmq_1 = (\mathcal{R}_1, a_2, 5 \leq a_1 < 20, \emptyset, 3\%)$, $dmq_2 = (\mathcal{R}_1, a_2, 0 \leq a_1 < 15, \emptyset, 5\%)$, $dmq_3 = (\mathcal{R}_1, a_2, 5 \leq a_1 < 15$ or $30 \leq a_1 < 40, \emptyset, 4\%)$. The set of elementary data selection predicates is then $S = \{0 \leq a_1 < 5,\ 5 \leq a_1 < 15,\ 15 \leq a_1 < 20,\ 30 \leq a_1 < 40\}$.*

Problem *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.*

In general, it is assumed that after collecting the queries to be concurrently executed using any strategy, duplicated queries are eliminated in a preprocessing step. According to previous studies (Morzy, Wojciechowski and Zakrzewicz, 2000), it is also advisable to combine queries operating on exactly the same dataset (at least the ones that have the same data selection predicate) into one query, whose results can be used to answer the original queries by simple checking of pattern constraints and/or frequency. Such a new query should have the frequency threshold equal to the smallest threshold among the queries to be replaced and the pattern constraint in the form of a disjunction of their pattern constraints.

4.2. 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. 2.

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 k (k>1) are generated from frequent itemsets of size k-1, separately for each query (lines 7-11). Generation of candidates (represented in the pseudo-code by the *apriori_gen*() function) is performed exactly as in the original Apriori algorithm. The candidates generated for each query are stored in a separate hash tree.

Minimum frequency thresholds of the queries are converted to their corresponding minimum support thresholds by multiplying them by the numbers of transactions in query source datasets (line 8). The numbers of transactions in query source datasets are determined during the first database scan.

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

```
Input: $DMQ = \{dmq_1, dmq_2, ..., dmq_n\}$,
where $dmq_i = (\mathcal{R}, a, \Sigma_i, \Phi_i, minfreq_i)$
(1)  for ($i$=1; $i \leq n$; $i$++) do
(2)    $\mathcal{C}_1^i$ = all possible 1-itemsets
(3)  for ($k$=1; $\mathcal{C}_k^1 \cup \mathcal{C}_k^2 \cup .. \cup \mathcal{C}_k^n \neq \emptyset$; $k$++) do begin
(4)    for each $s_j \in S$ do begin
(5)      $\mathcal{CC} = \{\mathcal{C}_k^i : \sigma_{s_j}\mathcal{R} \subseteq \sigma_{\Sigma_i}\mathcal{R}\}$
(6)      if $\mathcal{CC} \neq \emptyset$ then $count(\mathcal{CC}, \sigma_{s_j}\mathcal{R})$ end
(7)    for ($i$=1; $i \leq n$; $i$++) do begin
(8)      $minsup_i = \lceil minfreq_i * |\sigma_{\Sigma_i}\mathcal{R}| \rceil$
(9)      $\mathcal{F}_k^i = \{C \in \mathcal{C}_k^i : C.counter \geq minsup_i\}$
(10)     $\mathcal{C}_{k+1}^i = apriori\_gen(\mathcal{F}_k^i)$ end
(11) end
(12) for ($i$=1; $i \leq n$; $i$++) do
(13)   $Answer_i = \sigma_{\Phi_i} \bigcup_k \mathcal{F}_k^i$
```

Figure 2. Common Counting for Apriori

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.

The idea of Common Counting and its memory structures are illustrated in Fig. 3 for the set of three queries. Each query creates its own hash tree to store its candidates. If a given itemset is generated as a candidate by more than one query, it appears in more than one hash tree.

Common Counting does not handle pattern constraints Φ, but allows for using 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.

5. Common Candidate Tree

Common Counting optimizes scans of the parts of the database shared among the queries, while 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

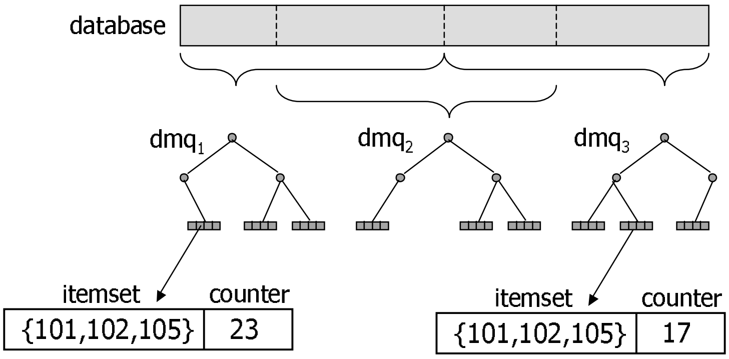

Figure 3. Illustration of Common Counting and its memory structures

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 structure of a candidate in a common hash tree used in the Common Candidate Tree method is illustrated in Fig. 4 for the set of three queries.

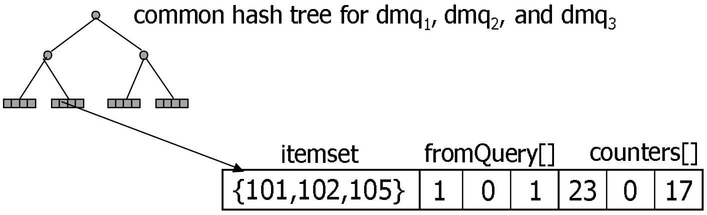

Figure 4. Illustration of a candidate structure used by Common Candidate Tree

The pseudo-code of Common Candidate Tree is depicted in Fig. 5. 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

```
Input: $DMQ = \{dmq_1, dmq_2, ..., dmq_n\}$,
where $dmq_i = (\mathcal{R}, a, \Sigma_i, \Phi_i, minfreq_i)$
(1)   $\mathcal{C}_1$ = all possible 1-itemsets
(2)   for ($k$=1; $\mathcal{C}_k \neq \emptyset$; $k$++) do begin
(3)      for each $s_j \in S$ do begin
(4)         $\mathcal{CC} = \{C \in \mathcal{C}_k : \exists i\ C.fromQuery[i] = true \wedge \sigma_{s_j}\mathcal{R} \subseteq \sigma_{\Sigma_i}\mathcal{R}\}$
(5)         if $\mathcal{CC} \neq \emptyset$ then $count(\mathcal{CC}, \sigma_{s_j}\mathcal{R})$ end
(6)      for ($i$=1; $i \leq n$; $i$++) do begin
(7)         $minsup_i = \lceil minfreq_i * |\sigma_{\Sigma_i}\mathcal{R}|\rceil$
(8)         $\mathcal{F}_k^i = \{C \in \mathcal{C}_k : C.counters[i] \geq minsup_i\}$
(9)         $\mathcal{C}_{k+1}^i = apriori\_gen(\mathcal{F}_k^i)$ end
(10)     $\mathcal{C}_{k+1} = \mathcal{C}_{k+1}^1 \cup \mathcal{C}_{k+1}^2 \cup .. \cup \mathcal{C}_{k+1}^n$
(11)  end
(12)  for ($i$=1; $i \leq n$; $i$++) do
(13)     $Answer_i = \sigma_{\Phi_i} \bigcup_k \mathcal{F}_k^i$
```

Figure 5. Common Candidate Tree

candidate sets as in Common Counting (lines 1 and 10). 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 6-9). 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).

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 for 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 (Boiński, Wojciechowski and Zakrzewicz, 2006; Wojciechowski and Zakrzewicz, 2005). 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 for using constraint handling techniques proposed for Apriori, since the candidate generation procedure used by Apriori is not modified by Common Candidate Tree.

6. Experimental results

In order to evaluate the performance and memory consumption of Common Candidate Tree we performed a series of experiments on two synthetic datasets generated with GEN (Agrawal et al., 1996). The first dataset, denoted GEN1, was generated using the following GEN settings: number of transactions = 100000, average number of items in a transaction = 5, number of different items = 1000, number of patterns (i.e., frequent itemsets) = 500, average pattern length = 3. The size of the GEN1 dataset was 6 MB. For the second dataset, denoted GEN2, generator parameters were modified to produce a significantly larger dataset, both in terms of the number of transactions and the average transaction size. The following GEN settings were used to generate GEN2: 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 size of the GEN2 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. The datasets were stored in flat files on a local disk.

In the experiments we varied the number of queries in a batch, the minimum frequency threshold, and the level of overlapping between the query datasets. For each query, its source dataset was a contiguous fragment of the generated dataset (containing 50000 subsequent transactions in case of GEN1, and 500000 subsequent transactions in case of GEN2). Although neither of the methods requires this, in all the experiments all the queries to be concurrently processed used the same frequency threshold, so as to make the potential influence of the frequency threshold and the difference in performance between the tested methods easier to observe[1].

In the first series of experiments we tested the effect of the level of overlapping between the query 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 (space occupied

[1] The greater the difference in minimum frequency 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.

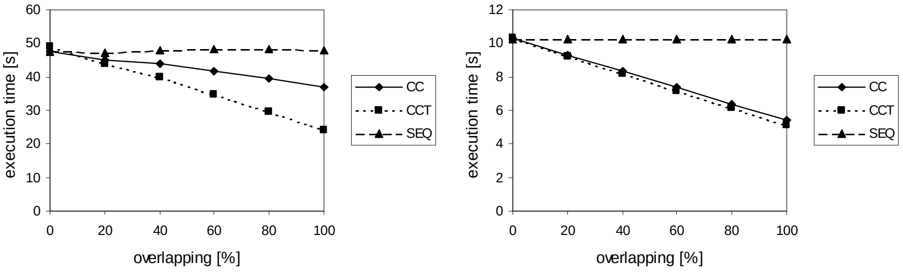

Figure 6. Execution times for two queries and different levels of overlapping with minfreq=1% (left) and minfreq=3% (right) on the GEN1 dataset

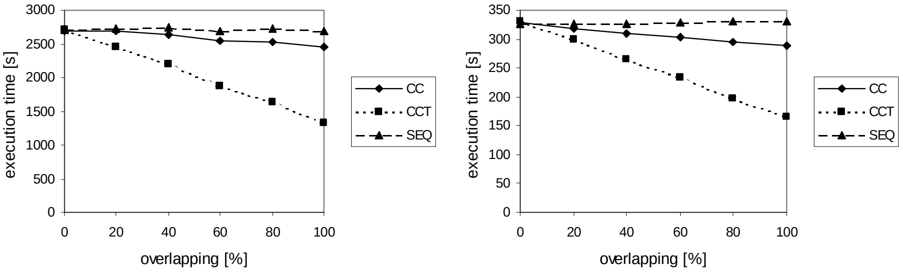

Figure 7. Execution times for two queries and different levels of overlapping with minfreq=0.7% (left) and minfreq=2% (right) on the GEN2 dataset

by tree nodes and candidates). The experiments were performed for the case of two overlapping queries. For each dataset we used two different minimum frequency thresholds. The thresholds were adjusted so that they resulted in significantly different numbers of Apriori iterations: 2 iterations for the higher frequency threshold and 5-8 iterations for the lower one. The respective frequency thresholds were 3% and 1% for GEN1, and 2% and 0.7% for GEN2.

Figs. 6 and 7 present execution times of the compared methods for different levels of overlapping for the case of two queries on GEN1 and GEN2 respectively. The execution times of CC and CCT decrease linearly with the increase of the level of overlapping, with CCT significantly outperforming CC in all cases with the exception of mining a small dataset with a high frequency threshold, where the performance gains of CCT over CC were not impressive. It should be noted that relative performance of CCT with respect to sequential processing was similar for all the four cases. On the other hand, performance of CC degraded with the increase of the dataset size and the decrease of the frequency threshold. Only for the higher of tested frequency thresholds on the small dataset CC

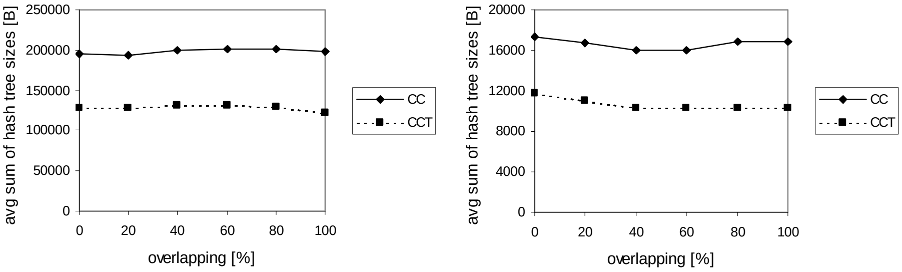

Figure 8. Average sums of hash tree sizes for two queries and different levels of overlapping with minfreq=1% (left) and minfreq=3% (right) on the GEN1 dataset

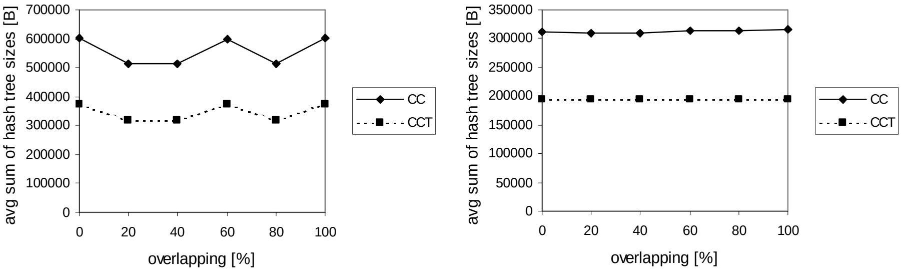

Figure 9. Average sums of hash tree sizes for two queries and different levels of overlapping with minfreq=0.7% (left) and minfreq=2% (right) on the GEN2 dataset

could compete with CCT, because due to a small number of candidates and transactions, penalty in performance for traversing two separate hash trees and possibly testing some candidates twice against the same transaction was not big in that case.

Figs. 8 and 9 show 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 GEN1 and GEN2, respectively. For the case of two queries CCT reduced average memory consumption by 32% to 39% compared to CC. Differences in memory consumption of both algorithms for different levels of overlapping observed for the frequency threshold of 0.7% on GEN2 were due to different characteristics of different regions of the dataset (both algorithms were similarly affected).

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 performance of the considered methods for different

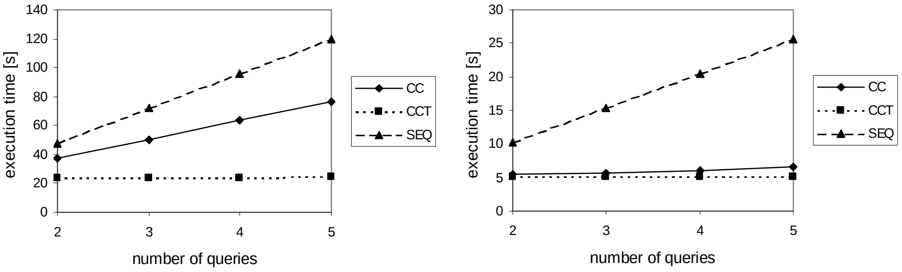

Figure 10. Execution times for 2-5 identical queries with minfreq=1% (left) and minfreq=3% (right) on the GEN1 dataset

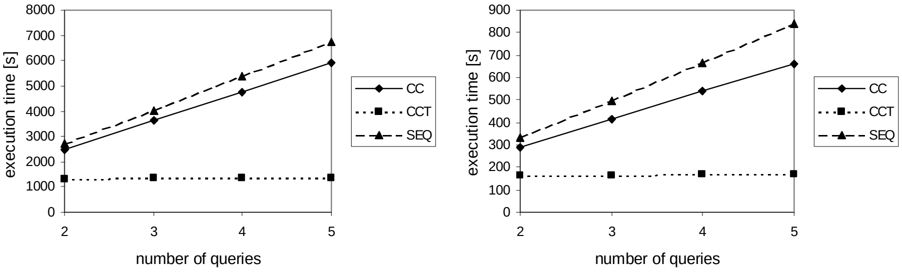

Figure 11. Execution times for 2-5 identical queries with minfreq=0.7% (left) and minfreq=2% (right) on the GEN2 dataset

numbers of queries in a batch, because the more queries, the more overlapping configurations are 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%).

Figs. 10 and 11 show how the execution time of a batch of queries increases with the number of queries forming it, for GEN1 and GEN2, respectively. The execution time of CCT increases insignificantly with the increase of the number of queries, whereas the execution time of CC on the large GEN2 dataset grows almost as rapidly as in the case of sequential execution of queries. On the small GEN1 dataset the gap between CCT and CC is smaller, especially for the higher frequency threshold, similarly as observed in the first series of experiments.

Figs. 12 and 13 present average sums of sizes of hash trees built by CC and CCT for batches of 2 to 5 queries on GEN1 and GEN2, respectively. With the increase of the number of queries the volume of main memory consumed by CCT grows significantly slower than in case of CC. The experiment clearly indicates that CCT is applicable for larger batches of queries than CC, even taking into

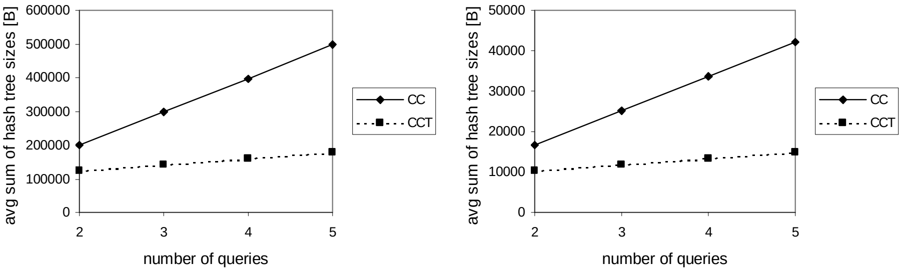

Figure 12. Average sums of hash tree sizes for 2-5 identical queries with minfreq=1% (left) and minfreq=3% (right) on the GEN1 dataset

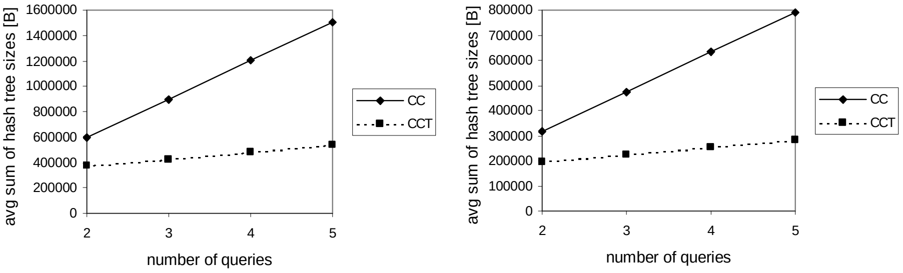

Figure 13. Average sums of hash tree sizes for 2-5 identical queries with minfreq=0.7% (left) and minfreq=2% (right) on the GEN2 dataset

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).

7. 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.

Concurrently to the work reported in this paper, we developed a method analogous to Common Candidate Tree, designed for FP-growth, aiming at the integration of FP-trees of the concurrently executed queries (Wojciechowski, Gałęcki and Gawronek, 2007). 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

Agrawal, R., Imielinski, T., and Swami, A. (1993) Mining Association Rules Between Sets of Items in Large Databases. In: P. Buneman and S. Jajodia, eds., *Proceedings of the 1993 ACM SIGMOD Int'l Conf. on Management of Data.* ACM Press, New York, 207–216.

Agrawal, R., Mehta, M., Shafer, J., Srikant, R., Arning, A. and Bollinger, T. (1996) The Quest Data Mining System. In: E. Simoudis, J. Han and U.M. Fayyad, eds., *Proc. of the 2nd Int'l Conf. on Knowledge Discovery in Databases and Data Mining.* AAAI Press, Portland, Oregon, 244–249.

Agrawal, R. and Srikant, R. (1994) Fast Algorithms for Mining Association Rules. In: J.B. Bocca, M. Jarke and C. Zaniolo, eds., *Proc. of the 20th Int'l Conf. on Very Large Data Bases.* Morgan Kaufmann, 487–499.

Alsabbagh, J.R. and Raghavan, V.V. (1994) Analysis of common subexpression exploitation models in multiple-query processing. *Proceedings of the 10th International Conference on Data Engineering.* IEEE Computer Society, 488–497.

Baralis, E. and Psaila, G. (1999) Incremental Refinement of Mining Queries. In: M.K. Mohania and A.M. Tjoa, eds., *Data Warehousing and Knowledge Discovery. Proceedings of the 1st DaWaK Conference.* **LNCS 1676**, Springer, 173–182.

Blockeel, H., Dehaspe, L., Demoen, B., Janssens, G., Ramon, J. and Vandecasteele, H. (2002) Improving the Efficiency of Inductive Logic Programming Through the Use of Query Packs. *Journal of Artificial Intelligence Research* **16**, 135–166.

Boiński, P., Wojciechowski, M. and Zakrzewicz, M. (2006) A Greedy Approach to Concurrent Processing of Frequent Itemset Queries. In: A.M. Tjoa and J. Trujillo, eds., *Data Warehousing and Knowledge Discovery. Proc. of the 8th DaWaK Conference.* **LNCS 4081**, Springer, 292–301.

Cheung, D.W., Han, J., Ng, V. and Wong, C.Y. (1996) Maintenance of Discovered Association Rules in Large Databases: An Incremental Updating Technique. In: S.Y.W. Su, ed., *Proceedings of the 12th ICDE Conference.* IEEE Computer Society, 106–114.

Han, J., Pei, J. and Yin, Y. (2000) Mining frequent patterns without candidate generation. In: W. Chen, J.F. Naughton and P.A. Bernstein, eds., *Proceedings of the 2000 ACM SIGMOD Int'l Conference on Management of Data.* ACM Press, 1–12.

Imielinski, T. and Mannila, H. (1996) A Database Perspective on Knowledge Discovery. *Communications of the ACM* **39** (11), 58–64.

Jarke, M. (1985) Common subexpression isolation in multiple query optimization. In: W. Kim, D.S. Reiner and D.S. Batory, eds., *Query Processing in Database Systems*. Springer, 191–205.

Jeudy, B. and Boulicaut, J-F. (2002) Using condensed representations for interactive association rule mining. In: T. Elomaa, H. Mannila and H. Toivonen, *Principles of Data Mining and Knowledge Discovery. 6th European Conference*. **LNCS 2431**, Springer, 225–236.

Jin, R., Sinha, K. and Agrawal, G. (2005) Simultaneous Optimization of Complex Mining Tasks with a Knowledgeable Cache. In: R. Grossman, R.J. Bayardo and K.P. Bennett, eds., *Proceedings of the 11th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM Press, 600–605.

Meo, R. (2003) Optimization of a Language for Data Mining. *Proceedings of the ACM Symposium on Applied Computing*. ACM Press, 437–444.

Morzy, T., Wojciechowski, M. and Zakrzewicz, M. (2000) Materialized Data Mining Views. In: D.A. Zighed, H.J. Komorowski and J.M. Zytkow, eds., *Principles of Data Mining and Knowledge Discovery. 4th European Conference*. **LNCS 1910**, Springer, 65–74.

Nag, B., Deshpande, P.M. and DeWitt, D.J. (1999) Using a Knowledge Cache for Interactive Discovery of Association Rules. *Proceedings of the 5th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM Press, 244–253.

Pei, J. and Han, J. (2000) Can We Push More Constraints into Frequent Pattern Mining? *Proceedings of the 6th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 350–354.

Roy, P., Seshadri, S., Sundarshan, S. and Bhobe, S. (2000) Efficient and Extensible Algorithms for Multi Query Optimization. In: W. Chen, J.F. Naughton, P.A. Bernstein, eds., *Proceedings of the 2000 ACM SIGMOD Int'l Conference on Management of Data*. ACM, 249–260.

Sellis, T. (1988) Multiple-query optimization. *ACM Transactions on Database Systems* **13**, 1, 23–52.

Wojciechowski, M., Gałęcki, K. and Gawronek, K. (2005) Concurrent Processing of Frequent Itemset Queries Using FP-Growth Algorithm. In: T. Morzy, M. Morzy and M. Wojciechowski, eds., *Proceedings of the 1st ADBIS Workshop on Data Mining and Knowledge Discovery*. Publishing House of Poznan University of Technology, 35–46.

Wojciechowski, M., Gałęcki, K. and Gawronek, K. (2007) Three Strategies for Concurrent Processing of Frequent Itemset Queries Using FP-growth. In: S. Dzeroski, J. Struyf, eds., *Knowledge Discovery in Inductive Databases, 5th International Workshop, KDID 2006*. **LNCS 4747**, Springer, 240–258.

Wojciechowski, M. and Zakrzewicz, M. (2003) Evaluation of Common Counting Method for Concurrent Data Mining Queries. In: L.A. Kalinichenko et al., eds., *Advances in Databases and Information Systems. 7th East European Conference, ADBIS 2003.* **LNCS 2798**, Springer, 76–87.

Wojciechowski, M. and Zakrzewicz, M. (2005) On Multiple Query Optimization in Data Mining. In: T.B. Ho, D.W. Cheung and H. Liu, eds., *Advances in Knowledge Discovery and Data Mining. 9th Pacific-Asia Conference, PAKDD 2005.* **LNCS 3518**, Springer, 696–701.

Zheng, Z., Kohavi, R. and Mason, L. (2001) Real world performance of association rule algorithms. *Proceedings of the 7th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining.* ACM, 401–406.