ISSN 1392 – 124X INFORMATION TECHNOLOGY AND CONTROL, 2008, Vol.37, No.4

APRIORI MULTIPLE ALGORITHM FOR MINING ASSOCIATION RULES

Predrag Stanišić, Savo Tomović

University of Montenegro, Podgorica, Montenegro

Abstract. One of the most important data mining problems is mining association rules. In this paper we consider discovering association rules from large transaction databases. The problem of discovering association rules can be decomposed into two sub-problems: find large itemsets and generate association rules from large itemsets. The second sub-problem is easier one and the complexity of discovering association rules is determined by complexity of discovering large itemsets. In this paper, we suggest Apriori-based algorithm for discovering large itemsets. Actually, we suggest a new procedure for large itemsets generation which is more efficient than the appropriate procedure of the original Apriori algorithm. For its implementation, we suggest a modified sort-merge-join algorithm, which is more efficient than nested-loop-join algorithm, which is suggested in the original Apriori algorithm. Besides, we propose a way in which Apriori Multiple finishes in just two iterations.

Keywords: data mining, knowledge discovery in databases, association analysis, Apriori algorithm.

1. Introduction

The motivation for discovering association rules has come from the requirement to analyze large amounts of supermarket basket data. A record in such data typically consists of the transaction unique identifier and the items bought in that transaction. Items can be different products which one can buy in supermarkets or on-line shops, or car equipment, or telecommunication companies' services etc.

The aim of association analysis for market basket data is to discover customer habits or patterns (to discover products which one buys together). For example, an association rule may state that "85% of customers who bought milk also bought bread". Discovering all such rules is important for planning marketing campaigns, designing catalogues, managing prices and stocks, customer relationships management etc. For example, a shop may decide to place bread close to milk because they are often bought together, to help shoppers finish their task faster. Or the shop may place them at opposite ends of a row, and place other associated items in between to tempt people to buy those items as well, as shoppers walk from one end of the row to the other.

In association analysis for market basket data it is important if item is part of customer basket or not, so each item has a Boolean variable representing the presence or absence of that item. Then each basket can be represented by a Boolean vector. By association analysis of these vectors we can discover rules, which present customer habits. For example, the fact that shopper who purchases milk also tends to purchase bread at the same time can be represented by association rule $milk \Rightarrow bread$ [*support* = 20%, *confidence* = 85%]. Support and confidence are measures of rule usefulness and certainty. A support of 20% for the previous association rule means that 20% of all transactions under analysis contain milk and bread. A confidence of 85% for the previous association rule means that 85% of the customers who purchased milk also purchased bread. The result of association analysis is strong association rules, which are rules satisfying a minimal support and a minimal confidence threshold. The minimal support and the minimal confidence are input parameters for association analysis.

The problem of association rules mining can be decomposed into two sub- problems [1]:

- Discovering *large* itemsets. Large itemsets have support greater than the minimal support;
- Generating rules. The aim of this step is to derive rules with high confidence (strong rules) from large itemsets. For each large itemset *l* one finds all not empty subsets of *l*; for each $a \subset l \wedge a \neq \varnothing$ one generates the rule $a \Rightarrow l - a$, if $\frac{\sigma(l)}{\sigma(a)} \geq$ *minimal confidence*.

We do not consider the second sub-problem in this paper, because the overall performances of mining association rules are determined by the first step. Efficient algorithms for solving the second sub-problem are exposed in [23].

We will briefly expose basic ideas of the Apriori algorithm from [2], which is one of the most famous for discovering large itemsets and which will be referred to as the original Apriori algorithm in the following text. The original Apriori algorithm generates large itemsets iteratively: in iteration $k \geq 2$, large (k-1)-itemsets, from previous iteration, is used for generating large k-itemsets, where large k-itemset is a large itemset with k items. Large k-itemsets are generated in two steps: first we generate a set of candidate k-itemsets (possibly large k-itemsets) and then we identify those ones with support greater than the minimal support among the set of candidates. The support of candidate itemset is the number of transactions, which contain that itemset, so it is necessary to read all transactions to determine support. The total number of iterations in the original Apriori algorithm is $k_{Max} + 1$, where k_{Max} is the maximum size of a large itemset. In each iteration the whole database is scanned.

From previous paragraph one can conclude that the efficiency of the original Apriori algorithm is mostly determined by the number of generated candidate itemsets and the number of iterations (in other words, the total number of I/O operations). In this paper we suggest the Apriori Multiple algorithm, which solves the problem of discovering large itemsets. It uses the new procedure for candidate generation, which is more efficient than the appropriate procedure from the original Apriori algorithm. The original Apriori algorithm suggests the join procedure $C_k = L_{k-1} \times L_{k-1}$, which generates candidate k-itemset by joining two large (k-1)-itemsets if and only if they have k-2 first items in common. Because of that, each join operation requires at most k-2 equality comparisons. The procedure for candidate generation in our Apriori Multiple algorithm is named $C_k = L_{k-1} \times L_{k-2}$, and candidate k-itemset is generated by joining large (k-1)-itemset and large (k-2)-itemset if and only if they have k-3 first items in common. According to that procedure each join operation requires at most k-3 equality comparisons. In the paper we prove the correctness of the procedure $C_k = L_{k-1} \times L_{k-2}$. For its implementation we suggest a modified sort-merge-join algorithm, which is more efficient than nested-loop-join algorithm which is suggested in the original Apriori algorithm. This is fully expressed when the sets L_{k-1} and L_{k-2} can not be read into main memory. Also, Apriori Multiple in the best case finishes in just two iterations.

The remainder of this paper is organized as follows. Related works are described in section 2. In section 3 basic concepts from association analysis are defined. In sections 4, 5 and 6 the Apriori Multiple algorithm is exposed and we compare steps of this algorithm to the corresponding steps of the original Apriori algorithm. At the end, section 7 contains a comparison of our algorithm with other methods and experiment results.

2. Related Work

The problem of discovering association rules was first introduced in [1] and an algorithm called *AIS* was proposed for mining association rules. In [20], an algorithm called *SETM* was proposed to solve this problem using relational operations. These were the first algorithms for mining association rules. For last fifteen years many algorithms for rule mining have been proposed. All these algorithms can be classified into two categories: the candidate-generation-and-test approach and the pattern-growth approach.

2.1. Candidate-generation-and-test algorithms

The Apriori algorithm from [2] is the basic candidate-generation-and-test algorithm. It is based on the Apriori principle, which says that the itemset *X'* containing itemset *X* is never large if itemset *X* is not large. Based on this principle, the Apriori algorithm generates a set of candidate large itemsets whose lengths are (k+1) from the large k-itemsets (for $k \geq 1$) and eliminates those candidates, which contain not large subset. Then, for the rest candidates, supports are counted and only those with support over *minsup* threshold are taken to be large (k+1)-itemsets.

Many variations have been proposed that focus on improving the efficiency of the original Apriori algorithm. They are focused on reducing the number of candidates generated or on reducing the number of database scans. We will mention some of them.

In [19], a hash-based technique was used to reduce the size of the candidate k-itemsets, for k>1. This is especially true for k=2. For example, when generating L_1, we need to read all transactions from database in order to determine support for each candidate 1-itemset from C_1. At the same time, we can generate all of the 2-itemsets for each transaction, hash them into different buckets of a hash table and increase corresponding bucket counts. A 2-itemset whose corresponding buckets count in the hash table is less than the minimal support cannot be large and should be removed from the candidate set.

In [2], [16] and [19] there were proposed ways how to reduce the number of transactions scanned in future iterations. The main idea is the following. A transaction that does not contain any large k-itemset can be removed from further considerations because such a transaction cannot contain any large (k+1)-itemset.

In [21], the partitioning technique was proposed. It requires just two database scans to mine large itemsets and consists of two phases. In phase I, the algorithm subdivides the database into *n* no overlapping partitions which can fit into main memory. For all partitions, all large itemsets are found and they are together candidate itemsets for whole database (because any itemset that is large in database must be large in at least one partition). In phase II, the algorithm counts actual support of each candidate from phase I in order to determine large itemsets for whole database.

In [24], the sampling approach is proposed. It requires just one database scan in the best case, and just two database scans in the worst case. The main idea is to randomly choose a sample from database, which can fit into main memory. The algorithm then searches for large itemsets in sample instead of database. These large itemsets called potentially large (PL) itemsets and they are used as part of candidate itemsets for whole database. The rest of candidate itemsets are found by applying function called negative border on set PL. Negative border is defined as minimal set of itemsets which are not in PL, but all their subsets are in PL. The second database scan is necessary only if some of large itemsets is from negative border (if all large itemsets are from PL, the second scan is not done).

In [6], a dynamic itemset counting approach is given. In this approach, database is partitioned into blocks marked by start point and new candidate itemsets can be added at any start point (unlike Apriori). New candidate itemset is added only if all of its subsets are estimated to be large so far.

2.2. Pattern-growth algorithms

Pattern-growth algorithms mine the complete set of large itemsets without candidate generation. The first pattern-growth algorithm was proposed in [17] and was called frequent-pattern growth or simply FP-growth. FP-growth constructs an FP-tree structure and mines large itemsets by traversing the constructed FP-tree.

The FP-tree consists of a prefix-tree of large 1-itemsets and an item header table. Each node in the prefix-tree has three fields: *item-name*, *count* and *node-link*. *Item-name* is the name of the item. *Count* is the number of transactions that consist of the large 1-itemsets on the path from root to this node. *Node-link* is the link to the next same item-name node in the FP-tree. Each entry in the item header table has two fields: *item-name* and *head of node-link*. *Item-name* is, as before, the name of the item. *Head of node-link* is the link to the first same item-name node in the prefix-tree.

FP-growth performs two database scans.

The first scan of the database derives the set of large 1-itemsets and their supports. The set of large 1-itemsets is sorted in the order of descending support count.

The FP-tree is then constructed as follows. First, create the root of the tree, labelled with "null". Scan the database a second time. The items in each transaction are processed according to descending support count and a branch is created for each transaction. In general, when considering the branch to be added for a transaction, the count of each node along a common prefix is incremented by 1, and nodes for the items following the prefix are created and linked accordingly.

The mining of the FP-tree proceeds as follows. Starting from each large 1-itemset construct its conditional pattern base (a "sub database" which consists of the set of prefix paths in the FP-tree co-occurring with the suffix pattern), then construct its (conditional) FP-tree, and perform mining recursively on such a tree. The pattern growth is achieved by the concatenation of the suffix pattern with the large patterns generated from a conditional FP-tree.

The FP-growth algorithm transforms the problem of finding long large itemsets to looking for shorter ones recursively and then concatenating the suffix. It uses the least large items as suffix, offering good selectivity.

Another well-known pattern-growth algorithm is Apriori-TFP [13]. The idea is to copy the input database into a data structure called P-tree, which maintains all the relevant aspects of the input, and then mines this structure.

When the database is large, it is unrealistic to construct a main memory-resident data structure to represent whole database. This is serious limit of pattern-growth approach. Because of this, we focused on candidate-generation-and-test approach, which previously mentioned limit does not exist for.

3. Preliminaries

In this section we define basic concepts of association analysis and give its formal definition.

Definition 1: Let $I=\{I_1,I_2,...,I_m\}$ be a set of items in database of transactions $D=\{t_1,t_2,...,t_n\}$, where $t_i=\{I_{i1},I_{i2},...,I_{ik(i)}\}$ and $I_{ij}\in I, 1\le j\le k(i)$, $1\le i\le n$. An association rule is an implication of the form $X\Rightarrow Y$, where $X\subseteq I$, $Y\subseteq I$ and $X\bigcap Y=\varnothing$ [11].

The result of association analysis should be only "strong" association rules, or in other words those which are "expressive" and "confident". Standard measures of association rules' "strength" are *support* and *confidence*, and both of these are calculated in dependence on the *support* of corresponding itemset. For the rule $X\Rightarrow Y$, the corresponding itemset is $X\cup Y$.

Definition 2: Let $I=\{I_1,I_2,...,I_m\}$ be a set of items in database of transactions $D=\{t_1,t_2,...,t_n\}$, where $t_i=\{I_{i1},I_{i2},...,I_{ik(i)}\}$ and $I_{ij}\in I, 1\le j\le k(i)$, $1\le i\le n$ (each transaction is a subset of *I*). Arbitrary set $X\subseteq I$ is termed as itemset. The *support* of the itemset *X*, denoted by $\sigma(X)$, is defined by the following formula:

$$\sigma(X) = |\{t_i \mid 1 \le i \le n \wedge X \subseteq t_i \wedge t_i \in D\}|.$$

The support of the itemset *X* is actually the number of transactions that contain *X*. For the transaction t_i, it can be stated that it contains itemset X if $X \subseteq t_i$ [11].

Definition 3: The support of association rule $X \Rightarrow Y$, denoted by $\sigma(X \Rightarrow Y)$, is the ratio of the number of transactions in database which contain $X \cup Y$ to the number of all transactions *n*. More formally:

$$\sigma(X \Rightarrow Y) = \frac{\sigma(X \cup Y)}{n} \qquad [11].$$

Definition 4: The confidence of association rule $X \Rightarrow Y$, denoted by $\alpha(X \Rightarrow Y)$, is the ratio of the number of transactions which contain $X \cup Y$ to the number of transactions containing *X*. More formally:

$$\alpha(X \Rightarrow Y) = \frac{\sigma(X \cup Y)}{\sigma(X)} \qquad [11].$$

Having defined all necessary concepts, we introduce the definition of association analysis problem.

Definition 5: Let $I = \{I_1, I_2, ..., I_m\}$ be a set of items in transactional database $D = \{t_1, t_2, ..., t_n\}$, where $t_i = \{I_{i1}, I_{i2}, ..., I_{ik(i)}\}$ and $I_{ij} \in I, 1 \le j \le k(i)$, $1 \le i \le n$. Association analysis problem consists of discovering all association rules $X \Rightarrow Y$, where $\sigma(X \Rightarrow Y) \ge minsup$ and $\alpha(X \Rightarrow Y) \ge minconf$ and values *minsup* and *minconf* are input parameters of the problem [11].

4. Apriori Multiple Algorithm

Apriori Multiple algorithm generates large itemsets starting with large 1-itemsets (itemsets consisted of just one item). Next, the algorithm iteratively generates large itemsets to the maximum size of large itemsets. Each iteration of the algorithm consists of two phases: *candidate generation* and *support counting*.

In the candidate generation phase potentially large itemsets or candidate itemsets are generated. The Apriori principle is used in this phase. It is based on anti-monotone property of the itemset support and provides elimination or pruning of some candidate itemsets without calculating its support (candidate containing at least one not large subset is pruned immediately, before support counting phase).

The support counting phase consists of calculating support for all previously generated candidates (which are not pruned according to the Apriori principle in the preceding candidate generation phase). In the support counting phase, it is essential to efficient determine if the candidates are contained in particular transaction $t \in D$, in order to increment their support. Because of that, the candidates are organized in hash tree [2]. The candidates, which have enough support, are termed as large itemsets.

The Apriori Multiple algorithm terminates when none of the large itemsets can be generated.

In the Apriori Multiple algorithm we have added new parameter named *multiple_num*, which determines the "length" of iteration. Actually, in the original Apriori algorithm, in the iteration *k*, set L_k (containing all large itemsets with *k* items) is generated, while our Apriori Multiple algorithm in the iteration *k* generates sets $L_{k+i}, 0 \le i \le multiple\_num - 1$.

The Apriori Multiple can use any value for *multiple_num* parameter. If *multiple_num*=0, our Apriori Multiple "becomes" the original Apriori algorithm. If we want to ensure that Apriori Multiple finishes in just two database scans, we need to choose value for *multiple_num* parameter such that k_{Max} < *multiple_num* holds, where k_{Max} is the maximal size of large itemsets. But we do not know k_{Max} value in advance. As a solution, we can apply some statistical methods to estimate k_{Max} value or we can use the following very simple approach. In the first scan, Apriori Multiple generates large 1-itemsets. During this scan Apriori Multiple can determine the length of the longest transaction in the database: t_{Max}. It is clear that $k_{Max} < t_{Max}$, so the algorithm can set *multiple_num* parameter to t_{Max}. Another approach is to set *multiple_num* parameter to average size of transactions. This does not guarantee that the algorithm finishes in two database scans, but it will generate less number of candidates. Also, Apriori Multiple can start with some value for *multiple_num* parameter and change this value in the following iterations. The *multiple_num* parameter can also be defined by user, just like *minsup* threshold. It means that user, according to domain knowledge or some other estimate, can specify the value for *multiple_num* parameter.

In our implementation we set *multiple_num* parameter to the average size of transactions.

In addition, all candidate *k*-itemsets (itemsets containing *k* items) will be signed as C_k, and all large *k*-itemsets as L_k. Pseudocode for our Apriori Multiple algorithm comes next.

```
Apriori Multiple Algorithm
Input: D - transactional database;
    Min_Sup-minimal support;
Output: L - large itemsets in D
Method:
   1. $L_1$ = all_large_1- itemset (D,
      Min_Sup)
   2. multiple_num =
      average_size_of_transactions
   3. FOR (k=2; $L_{k-1} \ne \emptyset$;
      k+=multiple_num) DO
      $C_k$ = apriori_gen($L_{k-1}$, $L_{k-1}$)
```

```
FOR i=1 TO multiple_num-1
   $C_k$ = apriori_gen($C_{k+i-1}$, $C_{k+i-1}$)
END FOR
FOR i=0 TO multiple_num-1
   createCandidateHashtree($C_{k+i}$)
   END FOR
   FOR EACH $t \in D$ DO
      FOR i=0 TO multiple_num-1
         traverseHashtree($C_{k+i}$, t)
      END FOR
   END FOR
   FOR i=0 TO multiple_num-1
      $L_{k+1} = \{c \in C_{k+1} \mid \sigma(c) \geq Min\_Sup\}$
   END FOR
END FOR
4. $L = \cup_k L_{k+1}$
```

Let us briefly outline the most important steps. Generating large 1-itemsets can be done in the same way as in the original Apriori algorithm from [2]. During this step Apriori Multiple determines average size of transactions and sets *multiple_num* to this value. In iteration $k \geq 2$, the set C_k is generated by calling *apriori_gen* function (the first call of *apriori_gen* function in the upper algorithm), but this is not the end of candidate generation phase. The loop in which the candidate itemsets C_{k+1}, $1 \leq multiple\_num - 1$ are generated comes next. It is done in the following way. According to the original Apriori algorithm from [2] candidate itemset C_{k+1} (candidate itemsets containing k+1 items) is formed from the set L_k (large itemsets containing k items) in iteration k+1. However, we want to generate C_{k+1} in iteration k (in order to reduce number of algorithm's iterations), but we do not have the necessary set L_k. As the solution, the first argument of *apriori_gen* function is C_k, which is generated by the first call. The second argument is C_{k-1}, which is known from the previous iteration. For the next call of *apriori_gen* function, the arguments are C_{k+1} and C_k, and in that way C_{k+2} is generated, etc. Candidate generation phase, which is here briefly discussed, is more precisely described in Section 5. The support counting phase comes next. Candidate itemsets C_{k+i}, $0 \leq i \leq multiple\_num - 1$ are organized in hash tree in order to make support counting process efficient. Then, we scan database and calculate support for candidates C_{k+i}, $0 \leq i \leq multiple\_num - 1$ by traversing corresponding hash trees. At the end of support counting phase large itemsets L_{k+i}, $0 \leq i \leq multiple\_num - 1$ are generated from candidate itemsets with large enough support. The support counting phase is described in Section 6.

The algorithm from [2] performs $k_{Max} + 1$ iterations, where k_{Max} is the maximal size of large itemsets, and in each iteration it scans whole database. Our Apriori Multiple algorithm finishes after $1 + \lceil k_{Max} / multiple\_num \rceil$ iterations.

5. Candidate Generation

Both our Apriori Multiple algorithm and the original Apriori algorithm from [2] assume that any itemset *I* is kept sorted according to some relation "<", where for all $x, y \in I$, $x<y$ means that the object *x* is in front of the object *y*. Also, we assume that all transactions in database *D* and all subsets of *I* are kept sorted in lexicographic order according to the relation "<".

For candidate generation we suggest the original method by which $C_k = L_{k-1} \times L_{k-2}$ is calculated, for $k \geq 3$. Candidate *k*-itemset is created from one large (*k*-1)-itemset and one large (*k*-2)-itemset in the following way. Let $X = \{x_1, x_2, .., x_{k-1}\} \in L_{k-1}$ and $Y = \{y_1, y_2, ..., y_{k-2}\} \in L_{k-2}$. Itemsets *X* and *Y* are joined if and only if the following condition is satisfied:

$$x_i = y_i (i = 1,2,...,k-3) \wedge x_{k-1} < y_{k-2}. \quad (1)$$

Let us prove the correctness of this method. It is sufficient to show that $L_k \subseteq C_k$ for all *k*, where C_k is generated by $L_{k-1} \times L_{k-2}$. Let $Z = \{z_1, z_2, ..., z_k\} \in L_k$ be arbitrary chosen and let $k \geq 3$. We will show that $Z \in C_k$. Let us take $X = \{x_1 = z_1, x_2 = z_2, ..., x_{k-2} = z_{k-2}, x_{k-1} = z_{k-1}\} \subseteq Z$ and $Y = \{y_1 = z_1, y_2 = z_2, ..., y_{k-3} = z_{k-3}, y_{k-2} = z_k\} \subseteq Z$. Then, according to the Apriori principle, we have $X \in L_{k-1}$ and $Y \in L_{k-2}$. Now we will check the condition (1) for itemsets *X* and *Y*: $x_1 = y_1 (= z_1) \wedge ... \wedge x_{k-3} = y_{k-3} (= z_{k-3}) \wedge x_{k-1} < y_{k-2} (z_{k-1} < z_k)$. It is directly checked that the previous formula is true, so that $Z = X \times Y \in C_k$. In this way we proved the correctness of the method $C_k = L_{k-1} \times L_{k-2}$ for $k \geq 3$.

In the original Apriori algorithm from [2], the joining procedure $C_k = L_{k-1} \times L_{k-1}$ is suggested. It generates candidate *k*-itemset by joining two large (*k*-1)-itemsets, if and only if they have first (*k*-2) items in common. Because of that, each join operation requires (*k*-2) equality comparisons. If a candidate *k*-itemset is generated by the method $L_{k-1} \times L_{k-2}$ for $k \geq 3$, it is enough (*k*-3) equality comparisons to process.

Generation of $C_k = L_{k-1} \times L_{k-2}$ can be represented by the following SQL query:

```
INSERT INTO $C_k$
SELECT $R_1.item_1, R_1.item_2, ..., R_1.item_{k-3}$,
          $R_2.item_{k-2}, R_1.item_{k-1}$
FROM $L_{k-1}$ AS $R_1$, $L_{k-2}$ AS $R_2$
WHERE $R_1.item_1 = R_2.item_1$ AND
         $R_1.item_2 = R_2.item_2$ AND … AND
         $R_1.item_{k-3} = R_2.item_{k-3}$
        AND $R_1.item_{k-1} < R_2.item_{k-2}$ .
```

In the previous query large (k-1)-itemsets are viewed as a relation with (k-1) attributes: $item_1$, $item_2$, ..., $item_{k-1}$ (attributes are items from large itemset). It is similar to the large (k-2)-itemsets. Function *apriori_gen* from the original Apriori algorithm from [2] processes the upper query according to nested-loop-join algorithm, which is the simplest and the most inefficient join algorithm. For the join $L_{k-1} \times L_{k-2}$ we suggest a modification of the sort-merge-join algorithm (note that L_{k-1} and L_{k-2} are sorted because of the way they are constructed and lexicographic order of itemsets).

By the original sort-merge-join algorithm [22], it is possible to compute natural joins and equi-joins. Let $r(R)$ and $s(S)$ be the relations and $R \cap S$ denote their common attributes. The algorithm keeps one pointer on the current position in relation $r(R)$ and another one pointer on the current position in relation $s(S)$. As the algorithm proceeds, the pointers move through the relations. It is supposed that the relations are sorted according to joining attributes, so tuples with the same values on the joining attributes are in consecutive order. Thereby, each tuple needs to be read only once, and, as a result, each relation is also read only once. The nested-loop-join algorithm reads each tuple of inner relation for each tuple of outer relation, so that the number of considered tuple pairs is $n_r * n_s$, where n_r is the number of tuples in $r(R)$ and n_s is the number of tuples in $s(S)$.It is clear that sort-merge-join algorithm is more efficient.

The modification of sort-merge-join algorithm we suggest refers to the elimination of restrictions that join must be natural or equi-join. First, we separated the condition (1):

$$x_i = y_i (i = 1,2,...,k-3) \tag{2}$$

$$\text{and } x_{k-1} < y_{k-2}. \tag{3}$$

Joining $L_{k-1} \times L_{k-2} = C_k$ is calculated according to the condition (2), in other words we compute natural join. For this, the described sort-merge-join algorithm is used, and our modification is: before $X = \{x_1, x_2, ..., x_{k-1}\}$ and $Y = \{y_1, y_2, .., y_{k-2}\}$, for which $X \in L_{k-1}$ and $Y \in L_{k-2}$ and $x_i = y_i, 1 \le i \le k-3$ is true, are merged, we check if condition (3) is satisfied.

The pseudocode of *apriori_gen* function comes next.

```
FUNCTION apriori_gen( $L_{k-1}, L_{k-2}$ )
1. i = 1 //pointer for $L_{k-1}$
2. j = 1 //pointer for $L_{k-2}$
3. WHILE ( $i \le L_{k-1}.Count() \wedge j \le L_{k-2}.Count()$ )
     $iset_1 = L_{k-1}[i]$
     i++
     $S_s = \{iset_1\}$
     done = false
     WHILE (NOT done ∧  $i \le L_{k-1}.Count()$ )
         $iset1_1 = L_{k-1}[i]$
       IF ( $iset_1[w] = iset1_1[w], 1 \le w \le k-2$ ) THEN
         $S_s = S_s \cup \{iset1_1\}$
         i++
       ELSE
         done = true
       END IF
     END WHILE
     $iset_2 = L_{k-2}[j]$
     WHILE ( $j <= L_{k-2}.Count() \wedge$
         $iset_2[\bigcup_{w=1}^{k-2} w] < iset_1[\bigcup_{w=1}^{k-2} w]$ )
         $iset_2 = L_{k-2}[j]$
       j++
     END WHILE
     WHILE ( $j <= L_{k-2}.Count() \wedge$
         $\bigwedge_{w=1}^{k-2} iset_1[w] = iset_2[w]$ )
       FOR EACH s IN $S_s$
       IF ( $iset_1[k-1] < iset_2[k-2]$ ) THEN
           $c = \{iset_1[1], ..., iset_1[k-2],$
               $iset_1[k-1], iset_2[k-2]\}$
       IF $contains\_not-large\_subset(c)$
         //pruning according to
         //Apriori principle
         $eliminate(c)$
         ELSE
           $C_k = C_k \cup \{c\}$
         END IF
       END IF
       NEXT
       j++
       $iset_2 = L_{k-2}[j]$
     END WHILE
      END WHILE
END FUNCTION
```

6. Support Counting

Support counting process means counting the support for all candidates which are not eliminated in the candidate generation phase. Those candidates which have greater support than *minsup* parameter are termed as large itemsets. They are used for candidate generation in the next iteration, which we talked about in Section 5.

Candidates from C_k are stored in a hash tree [2]. Nodes in the hash tree can be either leaves or interior nodes. The leaves contain a collection of candidate itemsets. The interior nodes contain a hash table where each row points to another node. The root of the hash tree is on the level 1. The interior node on the level d points to the node on the level d+1. When we add the candidate c_k in the hash tree, we start from the root and go down the tree until we reach a leaf. In the interior node on the level d we decide which branch to follow applying a hash function to the dth item of the candidate c_k. The hash tree for candidate *k*-itemsets has *k* levels.

We need to scan whole database in order to calculate supports of candidate itemsets. How can we use the hash tree to determine which candidates are contained in the transaction *t*? If we reach a leaf of the hash tree, it will be enough to check which candidates from the collection in that leaf are subsets of *t* and to increment their supports. If we reach an interior node by hashing to the *i*th item of the transaction *t*, then we do hashing on the all items of *t* which follow *i*th and recursively apply this procedure to the node from the corresponding bucket. In this way, itemsets from the transaction are compared only to the candidate itemsets from the same bucket (not with all possible ones).

We suggest the structural pattern Composite for implementation of a candidate hash tree. In Figure 1 the class diagram for hash tree implementation is shown.

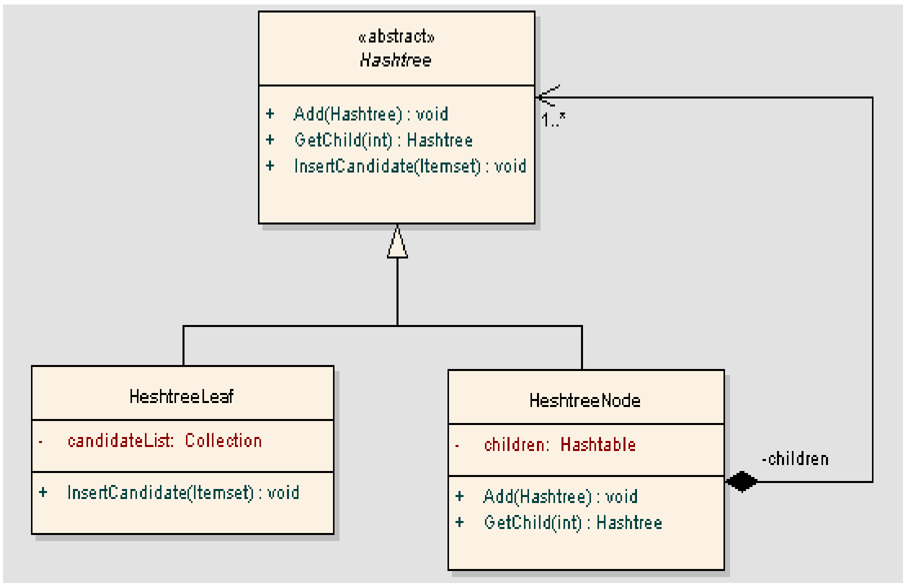
«abstract»
Hashtree
+ Add(Hashtree) : void
+ GetChild(int) : Hashtree
+ InsertCandidate(Itemset) : void
1..*
HeshtreeLeaf
- candidateList: Collection
+ InsertCandidate(Itemset) : void
HeshtreeNode
- children: Hashtable
+ Add(Hashtree) : void
+ GetChild(int) : Hashtree
-children

Figure 1. Composite pattern for hash tree implementation

7. Experimental Results

7.1. Comparison with the Original Apriori Algorithm

We implemented the original Apriori algorithm from [2] to the best of our knowledge based on the published reports. Also, *run time* used here means the total execution time, i.e., the period between input and output instead of CPU time measured in the experiments in some literature. We used programming language VB from Microsoft .NET framework. Experiments are performed on a PC with a CPU Intel(R) Core(TM)2 clock rate of 2.66GHz and with 2GB of RAM.

In experiments dataset which can be found at *www.cs.uregina.ca* is used. It contains 10000 binary transactions. The average length of transactions is 8.

In the first experiment, we compared the methods for candidate generation. Figure 2 shows that the method $L_{k-1} \times L_{k-2}$ generates less number of candidates than the method $L_{k-1} \times L_{k-1}$ for the greater support.

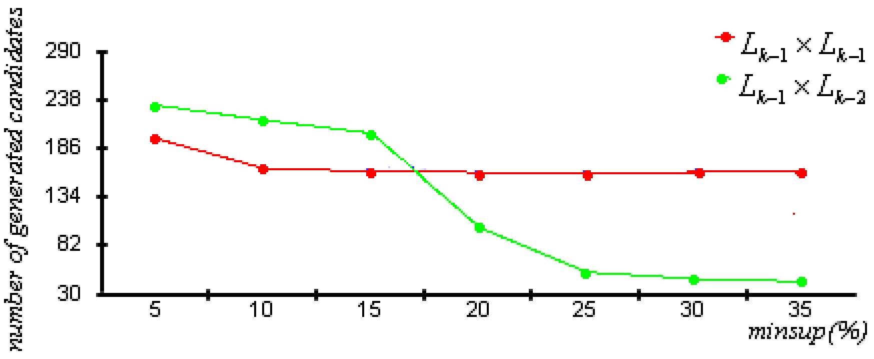

Figure 2. Number of generated candidates

In Figure 3, the results of comparing execution times for procedures for candidate generation are shown.

In Figure 4, the results of comparing the number of I/O operations in Apriori Multiple and in the original Apriori algorithm from [2] are shown. The parameter *multiple_num* from Apriori Multiple algorithm is set to 2. This experiment confirms the expectations that Apriori Multiple requires less I/O operations.

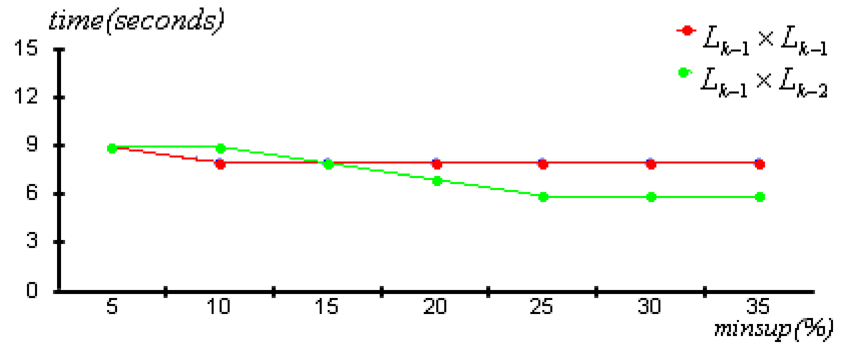

Figure 3. Execution times for different candidate generation methods

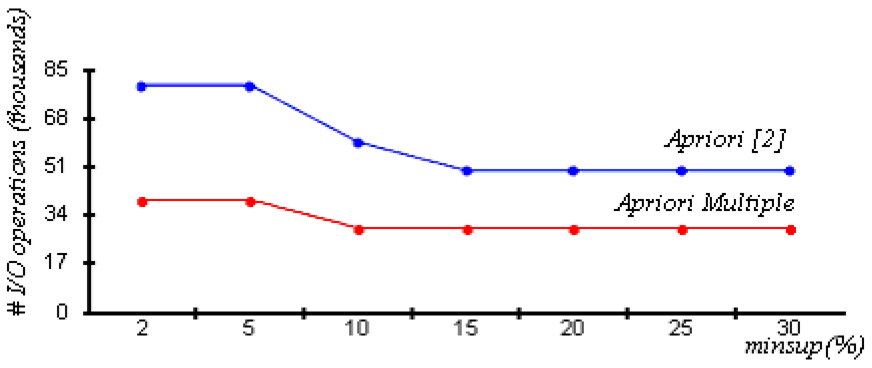

Figure 4. Number of I/O operations for Apriori and Apriori Multiple algorithm

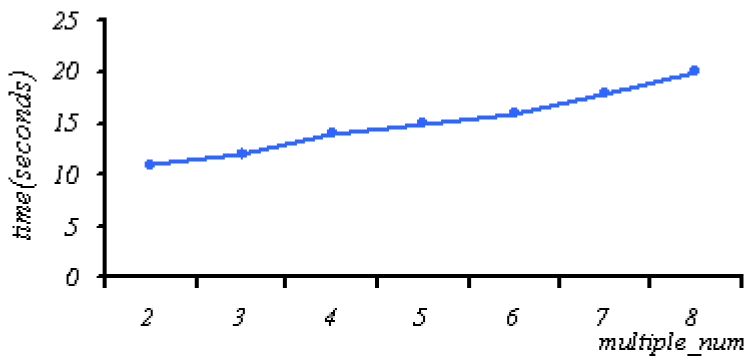

Figure 5. Dependence of execution time for Apriori Multiple algorithm on multiple_num parameter

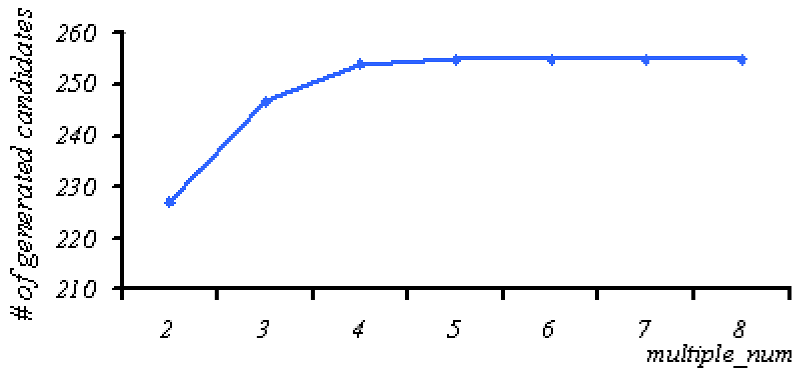

Figure 6. Dependence of the number of generated candidates for Apriori Multiple algorithm on multiple_num parameter

7.2. Comparison with Pattern-Growth Algorithms

Our Apriori Multiple algorithm is a candidate-generation-and-test algorithm for mining frequent itemsets from database of transactions. Recently, much more papers have been written concerning another approach: pattern-growth approach. We will expose some limits of the pattern-growth approach which does not exist in the candidate-and-test approach, and because of that we decided to concentrate on improvements of classic candidate-generation-and-test algorithm: the Apriori algorithm.

The first pattern-growth algorithm was proposed in [17] and was called frequent-pattern growth or simply FP-growth. FP-growth constructs main-memory-resident FP-tree structure to represent database and mines large itemsets by traversing the constructed FP-tree. Many FP-tree based algorithms have been proposed during the last few years. These algorithms differ in tree-based structures they use for database projection into main memory. Some of proposed tree-based structures are: CFP tree [18], COFI tree [12], T-tree [8, 9], P-tree [9, 10], PP-tree [4] etc., and they assume no limitation on main memory capacity. These structures differentiate slightly and all pattern-growth algorithms have common limits.

FP-growth based algorithms work well for sparse datasets, but if transactions contain many distinct items it leads to large and bushy tree structure (reduction ratio is not high). Also, tree depth is maximal number of large 1-items in transactions and for large databases (in which we are most interested) this number can be greater than 100, so with sufficiently large datasets it will not be possible to construct the tree within primary memory. However, results presented for pattern-growth algorithms demonstrate its effectiveness in cases when the tree is memory resident, but the linkage between nodes of the tree makes it difficult to effect a comparable implementation when this is not the case. Apriori-based algorithm can mine database of any size and they show linear scalability with the number of transactions.

The performance of pattern-growth algorithms will be affected if it is impossible to proceed entirely within primary memory. Many partitioning strategies have been proposed [3], [4], [5] and [8] to deal with these cases. The main idea behind these approaches is to subdivide database into segments and then separately process each segment. It means that algorithm creates tree representation for each segment and stores each tree in secondary memory. Then each pass of algorithm requires each of previously created trees to be read in turn from secondary memory as Apriori-based algorithms do with transactions. Also, in some cases it is not possible to compute the support for a set by considering only the sub-tree in which it is located, so procedures for traversing trees are very expensive and complicated. These are especially expensive if the depth of tree is large, because of the greater depth of recursion required. Numerous experiments have shown that 80% of CPU time was used for traversing trees [14].

To sum up, pattern-growth algorithms require two database scans with assumption that database can be mapped into memory resident tree structure, while our Apriori Multiple algorithm requires two database scans and without limits on the database size.

8. Conclusion

In this paper the procedure of discovering association rules in large transaction databases is exposed. Each transaction contains a unique identifier and the items bought in that transaction. Also, each transaction is sorted in lexicographic order. The aim of association analysis is to find association rules which satisfy the minimal support and the minimal confidence thresholds. This task is solved in two steps: discovering large itemsets and generating rules from large itemsets. The overall performance of mining association rules is determined by the first step. For solving the first step we suggested Apriori Multiple algorithm, which is a modification of the well known Apriori algorithm from [2]. Our Apriori Multiple algorithm uses the new procedure for candidate generation. We proved its correctness. It is more efficient than the appropriate one from the original Apriori algorithm. Also, our Apriori Multiple algorithm is able to finish in just two iterations. These considerations are confirmed by experimental results which are part of this paper.

We plan to extend this paper to:

- Apply concepts from association rule mining to classification problems, especially signal/background classification in HEP (High Energy Physics);
- In the previous considerations we ignored quantity of items sold or the price paid to purchase them, which can be really important for some practical applications. Discovering such rules requires additional modifications of the Apriori Multiple algorithm and the Apriori algorithm from [2];
- Mining multilevel association rules from transaction databases (these rules involve items at different levels of abstraction);
- Mining multidimensional association rules from relational databases and data warehouses (these rules involve more than one dimension or predicate, e.g. rules relating what a customer shopper buy as well as shopper's occupation).

References

[1] **R. Agrawal, T. Imielinski, A. Swami.** Mining Association Rules between Sets of Items in Large Databases. *Proceedings of the* 1993 *ACM SIGMOD International Conference on Management of Data, Washington, DC, USA*, 1993, 207-216.

[2] **R. Agrawal, R. Srikant.** Fast Algorithms for Mining Association Rules. *IBM Almaden Research Center, San Jose CA* 95120, 1994.

[3] **S. Ahmed, F. Coenen, P. Leng.** Strategies for Partitioning Data in Association Rule Mining. *Coenen, F. P., Preece, A., Macintosh, A. L. (Eds.), Research and Development in Intelligent Systems* XX, *Springer, London*, 2003, 127-140.

[4] **S. Ahmed, F. Coenen, P. Leng.** A Tree Partitioning Method for Memory Management in Association Rule Mining. *Kambayashi, Y., Mohania, M.K., Woll, W. (Eds.) Data Warehousing and Knowledge Discovery, In Proc. DaWaK* 2004 *conference Lecture Notes in Computer Science*, 2004, *Vol*.3181, 331-340.

[5] **S. Ahmed, F. Coenen, P. Leng.** Tree-based Partitioning of Data for Association Rule Mining. *Knowledge and Information Systems*, *Vol*.10, *No*.3, 2006, 315-331.

[6] **S. Brin, R. Motwani, J.D. Ullman, S. Tsur.** Dynamic Itemset Counting and Implication Rules for Market Basket data. *Proceedings of the* 1997 *ACM SIGMOD International Conference on Management of Data, May* 13-15, 1997, *Tucson, AZ, USA*, 1997, 255-264.

[7] **F. Coenen, G. Goulbourne, P.H. Leng.** Computing Association Rules Using Partial Totals. *In de Raedt, L., Siebes, A. (Eds.), Principles of Data Mining and Knowledge Discovery, Proc PKDD, Spring Verlag Lecture Notes in Computer Science*, 2001, *Vol*.2168, 54-66.

[8] **F. Coenen, P. Leng, S. Ahmed.** T-Trees, Vertical Partitioning and Distributed Association Rule Mining. *Proceedings of the Third IEEE International Conference on Data Mining* (*ICDM*-2003), 2003, 513-516.

[9] **F. Coenen, P. Leng, S. Ahmed.** Data Structure for Association Rule Mining. T-*trees and P-trees. IEEE Transactions on Knowledge and Data Engineering*, *Vol*.16, *No*.6, 2004, 774-778.

[10] **F. Coenen, G. Goulbourne, P. Leng.** Tree Structures for Mining Association Rules. *Data Mining and Knowledge Discovery*, *Vol*.8, *No*.1, 2004, 25-51.

[11] **M.H. Dunham.** Data Mining Introductory and Advanced Topics. *Prentice Hall, New Jersey* 2003.

[12] **M. El-Hajj, O.R. Zaiane.** Non-Recursive Generation of Frequent K-itemsets from Frequent Pattern Tree Representations. *In: Kambayashi Y., Mohania M.K., Woll W. (Eds.) Data Warehousing and Knowledge Discovery, Proceedings of the* 5th *International Conference* (*DaWaK* 2003), *Lecture Notes in Computer Science*, *Vol*.2737, 2003, 371-380.

[13] **G. Goulbourne, F. Coenen, P. Leng.** Algorithms for Computing Association Rules Using a Partial-Support Tree. *Knowledge-Based Systems, Vol*.13, 2000, 141-149.

[14] **G. Grahne, J. Zhu.** Efficiently Using Prefix-trees in Mining Frequent Itemsets. *Proc. of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations*, 2003.

[15] **J. Han, M. Kamber.** Data Mining: Concepts and Techniques. *Morgan Kaufmann Publishers, San Francisco*, 2001.

[16] **J. Han, Y. Fu.** Discovery of multiple-level association rules from large databases. *Proceedings of* 21*th International Conference on Very Large Data Bases* (*VLDB*'95), *September* 11-15, 1995, *Zurich, Switzerland, Morgan Kaufmann*, 1995, 420-431.

[17] **J. Han, J. Pei, Y. Yu.** Mining Frequent Patterns without Candidate Generation. *Proceedings of the* 2000 *ACM SIGMOD International Conference on Management of Data, Dalas, Texas, USA*, 2000, 1-12.

[18] **G. Liu, H. Lu, W. Lou, J. Yu.** On Computing, Storing and Querying Frequent Patterns. *.Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, August* 24-27, 2003, *Washington, DC, USA, ACM*, 2003, 607-612.

[19] **J. S. Park, M.-S. Chen, P.S. Yu.** An Effective Hash-Based Algorithm for Mining Association Rules. *Proceedings of the* 1995 *ACM SIGMOD International Conference on Management of Data, San Jose, CA, USA*, 1995, 175-186.

[20] **G. Piatetsky-Shapiro, W.J. Frawley.** Knowledge Discovery in Databases. *MIT Press*, 1991.

[21] **A. Savasere, E. Omiecinski, S. Navathe.** An Efficient Algorithm for Mining Association Rules in Large Databases. *Proceedings of* 21*th International Conference on Very Large Data Bases* (*VLDB*'95), *September* 11-15, 1995, *Zurich, Switzerland, Morgan Kaufmann*, 1995, 432-444.

[22] **A. Silberschatz, H.F. Korth, S. Sudarshan.** Database System Concepts. *Mc Graw Hill, New York*, 2006.

[23] **P. Tan, M. Steinbach, V. Kumar.** Introduction to Data Mining. *Addison Wesley, Boston*, 2006.

[24] **H. Toivonen.** Sampling Large Databases for Association Rules. *Proceedings of* 22*th International Conference on Very Large Data Bases* (*VLDB*'96), *September* 3-6, 1996, *Mumbai, India, Morgan Kaufmann*, 1996, 134-145.

[25] **M.J. Zaki, S. Parthasarathy, M. Ogihara, W. Li.** New algorithms for Fast Discovery of Association Rules. *Proceedings of the Third International Conference on Knowledge Discovery and Data Mining* (*KDD*-97), *August* 14-17, 1997, *Newport Beach, USA, AAAI Press*, 1997, 283-286.

Received July 2008.