First International Conference
Modelling and Development of Intelligent Systems
Sibiu - Romania, 22-25 October, 2009

Improvements of Apriori Algorithm

Ioan Daniel Hunyadi

Abstract

In this paper, I describe the main techniques used to solve mining frequent itemset problems and give a comprehensive survey of the most influential algorithms that were proposed. The Apriori Algorithm was first algorithm proposed for mining frequent itemsets problem. I will present new improvements on Apriori algorithm which solve these mining problems more efficiently.

1 Introduction

Most of the established companies have accumulated masses of data from their customers for decades. With the e-commerce applications growing rapidly, the companies will have a significant amount of data in months not in years. The scope of Data Mining, also known as Knowledge Discovery in Databases (KDD), is to find trends, patterns, correlations, anomalies in these databases which can help us to make accurate future decisions.

Data mining is not magic. No one can guarantee that the decision will lead to good results. Data Mining only helps experts to understand the data and lead to good decisions. Data Mining is an intersection of the fields Databases, Artificial Intelligence and Machine Learning.

Since their introduction in 1993 by Argawal et al. [1], the frequent itemset and association rule mining problems have received a great deal of attention. Within the past decade, hundreds of research papers have been published presenting new algorithms or improvements on existing algorithms to solve these mining problems more efficiently.

2 Problem Descriptions

Let $I = (i_1, i_2, ..., i_m)$ be a set of transactions. Each i is called an *item*. D is the set of transactions where each *transaction* T is a set of items (itemset) such that $T \subseteq I$. Every transaction has a unique identifier called the TID. An itemset having k items is called a *k-itemset*. Let X and Y be distinct itemsets. The *support* of an itemset X is the ratio of the itemsets containing X to the number of all itemsets. Let us define $|X|$ as the number of itemsets containing X and $|D|$ as the number of all items, $|X,Y|$ as the number of itemsets containing both X and Y. The support of itemset X is defined as follows:

$$support(X) = \frac{|X|}{|D|}$$

The rule $X \Rightarrow Y$ has *support s* if %*s* of the transactions in *D* contain *X* and *Y* together.

$$support(X \Rightarrow Y) = \frac{|X,Y|}{|D|}$$

Support measures how common the itemsets are in the database and *confidence* measures the strength of the rule. A rule is said to have *confidence c* if %*c* of the transactions that contains *X* also contains *Y*.

$$confidence(X \Rightarrow Y) = \frac{support(X,Y)}{support(X)}$$

Given a set of transactions *D* the task of association rule mining is to find rules $X \Rightarrow Y$ such that the support of the rule is greater than a user specified minimum support called *minsupp* and the confidence is greater than a user specified minimum called *minconf*. An itemset is called *frequend* if its support is greater than *minsupp*.

The collection of frequent itemsets in D which have their support grater then *minsupp* is denoted by *F*.

$$F=\{ X \subseteq I \mid support(X) \geq minsupp\}.$$

The task of association rule mining can be divided into two: In the first phase, the frequent itemsets are found using *minsupp*, and in the second phase, the rules are generated using *minconf*.

The collection of frequent and confident association rules with respect to *minsupp* and *minconf* is denoted by *R*.

$$R = \{X \Rightarrow Y \mid X, Y \subseteq I, X \cap Y = \{\}, X \cup Y \in F, confidence(X \Rightarrow Y) \geq minconf\}$$

The algorithms that implement association mining make multiple passes over the data. Most algorithms first find the frequent itemsets and then generate the rules accordingly. They find the large itemsets incrementally increasing itemset sizes and then counting the itemsets to see if they are large or not. Since finding the large itemsets is the hard part, research mostly focused on this topic.

3 Apriori Algorithm

The first algorithm to generate all frequent itemsets and confident association rules was the AIS algorithm by Agrawal et al. [1], which was given together with the introduction of this mining problem. Shortly after that, the algorithm was improved and renamed Apriori by Agrawal et al., by exploiting the monotonicity property of the support of itemsets and the confidence of association rules [3, 15].

For simplicity the items in transactions and itemsets are kept sorted in their lexicographic order unless stated otherwise. The itemset mining phase of the Apriori algorithm is given in Listing 1. I use the notation *X[i],* to represent the ith item in *X*. The *k*-prefix of an itemset *X* is the *k*-itemset {*X[1], . . . ,X[k]*}.

Listing 1. Apriori algorithm – Itemset mining

```
Input: D, minsupp
Output: F
  C1={{i}|i ∈ I};
  k=1;
  while Ck≠{} do{
    //Compute the supports of all candidate itemsets
    forall transactions(tid,D) ∈ D
      forall candidate itemsets X ∈ Ck
        if ( X ⊆ I )
          X.support++;
    //Extract all frequent itemsets
    Fk = {X|X.support ≥ minsupp};
```

```
    //Generate new candidate itemsets
    forall X,Y ∈ Fk, X[i]=Y[i] for 1 ≤ i ≤ k-1, and X[k}<Y[k]{
        I = X ∪ {Y[k]};
        if (∀J ⊂ I, |J| = k, J ∈ Fk)
            Ck+1 = Ck+1 ∪ I;
    }
    k++;
}
```

The algorithm performs a breadth-first search through the search space of all itemsets by iteratively generating candidate itemsets C_{k+1} of size $k+1$, starting with $k = 0$. An itemset is a candidate if all of its subsets are known to be frequent. More specifically, C_1 consists of all items in I, and at a certain level k, all itemsets of size $k+1$ are generated. This is done in two steps. First, in the *join* step, F_k is joined with itself. The union $X \cup Y$ of itemsets $X,Y \in F_k$ is generated if they have the same $(k-1)$ - prefix. In the prune step, $X \cup Y$ is only inserted into C_{k+1} if all of its k-subsets occur in F_k.

To count the supports of all candidate k-itemsets, the database, which retains on secondary storage in the horizontal database layout, is scanned one transaction at a time, and the supports of all candidate itemsets that are included in that transaction are incremented . All itemsets that turn out to be frequent are inserted into F_k.

If the number of candidate $(k+1)$ - itemsets is too large to retain into main memory, the candidate generation procedure stops and the supports of all generated candidates is computed as if nothing happened. But then, in the next iteration, instead of generating candidate itemsets of size $k+2$, the remainder of all candidate $(k+1)$ - itemsets is generated and counted repeatedly until all frequent itemsets of size $k+1$ are generated.

4. Data Structures

The candidate generation and the support counting processes require an efficient data structure in which all candidate itemsets are stored since it is important to efficiently find the itemsets that are contained in a transaction or in another itemset.

4.1. Hash-tree

In order to efficiently find all k-subsets of a potential candidate itemset, all frequent itemsets in F_k are stored in a hash table.

Candidate itemsets are stored in a hash-tree [2]. A node of the hash-tree either contains a list of itemsets (a leaf node) or a hash table (an interior node). In an interior node, each bucket of the hash table points to another node. The root of the hash-tree is defined to be at depth 1. An interior node at depth d points to nodes at depth $d+1$. Itemsets are stored in leaves.

When we add a k-itemset X during the candidate generation process, we start from the root and go down the tree until we reach a leaf. At an interior node at depth d, we decide which branch to follow by applying a hash function to the $X[d]$ item of the itemset, and following the pointer in the corresponding bucket. All nodes are initially created as leaf nodes. When the number of itemsets in a leaf node at depth d exceeds a specified threshold, the leaf node is converted into an interior node, only if $k>d$.

In order to find the candidate-itemsets that are contained in a transaction T, we start from the root node. If we are at a leaf, we find which of the itemsets in the leaf are contained in T and increment their support. If we are at an interior node and we have reached it by hashing the item i, we hash on each item that comes after i in T and recursively apply this procedure to the node in

the corresponding bucket. For the root node, we hash on every item in *T*. An example of hash-tree structure for five items is presented in figure 1.

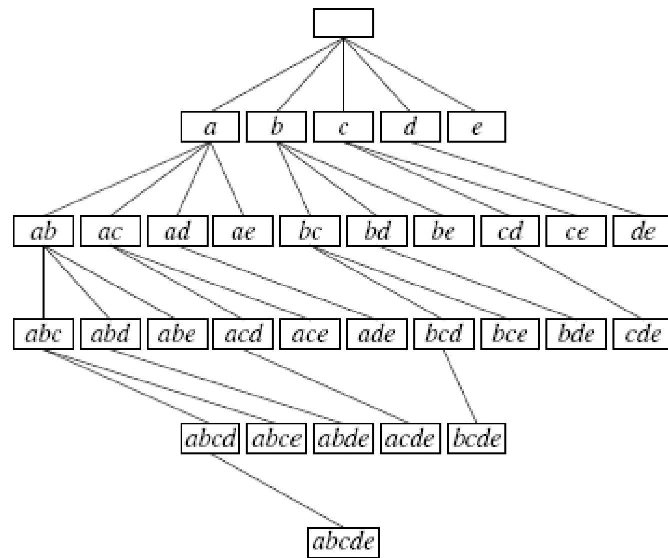

Figure 1. Hash-tree structure

4.2. Trie

Another data structure that is commonly used is a trie (or prefix-tree) [5,7, 8, 6]. In a trie, every *k*-itemset has a node associated with it, as does its ($k-1$) - prefix. The empty itemset is the root node. All the 1-itemsets are attached to the root node, and their branches are labeled by the item they represent. Every other *k*-itemset is attached to its ($k-1$) - prefix. Every node stores the last item in the itemset it represents, its support, and its branches. The branches of a node can be implemented using several data structures such as a hash table, a binary search tree or a vector. An example of prefix-tree structure for five items is presented in figure 2.

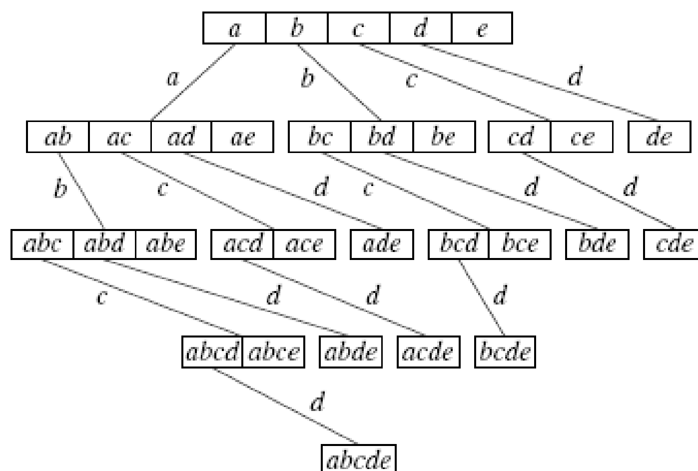

Figure 2. Prefix-tree structure

At a certain iteration *k*, all candidate *k*-itemsets are stored at depth *k* in the trie. In order to find the candidate-itemsets that are contained in a transaction *T*, we start at the root node. To process a transaction for a node of the trie, (1) follow the branch corresponding to the first item in the

transaction and process the remainder of the transaction recursively for that branch, and (2) discard the first item of the transaction and process it recursively for the node itself. This procedure can still be optimized, as is described in [7].

Also the join step of the candidate generation procedure becomes very simple using a trie, since all itemsets of size k with the same $(k-1)$ – prefix are represented by the branches of the same node (that node represents the $(k-1)$ - prefix). Indeed, to generate all candidate itemsets with $(k-1)$ – prefix X, we simply copy all siblings of the node that represents X as branches of that node. Moreover, we can try to minimize the number of such siblings by reordering the items in the database in support ascending order [7, 8, 6]. Using this heuristic, we reduce the number of itemsets that is generated during the join step, and hence, we implicitly reduce the number of times the prune step needs to be performed. Also, to find the node representing a specific k-itemset in the trie, we have to perform k searches within a set of branches. Obviously, the performance of such a search can be improved when these sets are kept as small as possible.

An in depth study on the implementation details of a trie for Apriori can be found in [7].

5. Optimizations

A lot of other algorithms proposed after the introduction of Apriori retain the same general structure, adding several techniques to optimize certain steps within the algorithm. Since the performance of the Apriori algorithm is almost completely dictated by its support counting procedure, most research has focused on that aspect of the Apriori algorithm. The performance of this procedure is mainly dependent on the number of candidate itemsets that occur in each transaction.

5.1. AprioriTid, AprioriHybrid

Together with the proposal of the Apriori algorithm, Agrawal et al. [3, 2] proposed two other algorithms, AprioriTid and AprioriHybrid. The AprioriTid algorithm reduces the time needed for the support counting procedure by replacing every transaction in the database by the set of candidate itemsets that occur in that transaction. This is done repeatedly at every iteration k. The adapted transaction database is denoted by $\overline{C_k}$. The algorithm is given in Listing 2.

More implementation details of this algorithm can be found in [4]. Although the AprioriTid algorithm is much faster in later iterations, it performs much slower than Apriori in early iterations. This is mainly due to the additional overhead that is created when $\overline{C_k}$ does not fit into main memory and has to be written to disk. If a transaction does not contain any candidate k-itemsets, then $\overline{C_k}$ will not have an entry for this transaction. Hence, the number of entries in $\overline{C_k}$ may be smaller than the number of transactions in the database, especially at later iterations of the algorithm. Additionally, at later iterations, each entry may be smaller than the corresponding transaction because very few candidates may be contained in the transaction. However, in early iterations, each entry may be larger than its corresponding transaction.

Therefore, another algorithm, AprioriHybrid, has been proposed [3,2] that combines the Apriori and AprioriTid algorithms into a single hybrid. This hybrid algorithm uses Apriori for the initial iterations and switches to AprioriTid when it is expected that the set $\overline{C_k}$ fits into main memory. Since the size of $\overline{C_k}$ is proportional with the number of candidate itemsets, a heuristic is used that estimates the size that $\overline{C_k}$ would have in the current iteration. If this size is small enough and there are fewer candidate patterns in the current iteration than in the previous iteration, the algorithm decides to switch to AprioriTid

Listing 2. AprioriTid algorithm

```
Input: D, minsupp
Output: F
  Compute F_1 of all frequent items;
  C̄_1 =D; (with all items not in F_1 removed)
  k=2;
  while F_k-1≠{} do{
     Compute C_k of all candidate k-itemsets
     C̄_k ={};
     // Compute the supports of all candidate itemsets
     forall transactions(tid,T) ∈ C̄_k {
        C_T = {};
        forall X∈ C_k
          if ( {X[1],...,X[k-1]} ∈ T ∧ {X[1],...,X[k-2],X[k]} ∈ T ){
             C_T = C_T ∪ {X};
             X.support++;
          }
        if (C_T≠{})
          C̄_k = C̄_k ∪ {(tid, C_T)}
     }
     Extract F_k of all frequent k-itemsets;
     k++;
}
```

5.2 Counting candidate 2-itemsets

Shortly after the proposal of the Apriori algorithms described before, Park et al. proposed another optimization, called DHP (Direct Hashing and Pruning) to reduce the number of candidate itemsets [13]. During the kth iteration, when the supports of all candidate k-itemsets are counted by scanning the database, DHP already gathers information about candidate itemsets of size $k + 1$ in such a way that all $(k + 1)$-subsets of each transaction after some pruning are hashed to a hash table. Each bucket in the hash table consists of a counter to represent how many itemsets have been hashed to that bucket so far. Then, if a candidate itemset of size $k+1$ is generated, the hash function is applied on that itemset. If the counter of the corresponding bucket in the hash table is below the minimal support threshold, the generated itemset is not added to the set of candidate itemsets. Also, during the support counting phase of iteration k, every transaction trimmed in the following way. If a transaction contains a frequent itemset of size $k+1$, any item contained in that $k+1$ itemset will appear in at least k of the candidate k-itemsets in C_k. As a result, an item in transaction T can be trimmed if it does not appear in at least k of the candidate k-itemsets in C_k. These techniques result in a significant decrease in the number of candidate itemsets that need to be counted, especially in the second iteration. Nevertheless, creating the hash tables and writing the adapted database to disk at every iteration causes a significant overhead.

Although DHP was reported to have better performance than Apriori and AprioriHybrid, this claim was countered by Ramakrishnan if the following optimization is added to Apriori [14]. Instead of using the hash-tree to store and count all candidate 2-itemsets, a triangular array C is created, in which the support counter of a candidate 2-itemset $\{i, j\}$ is stored at location $C[i][j]$. Using this array, the support counting procedure reduces to a simple two level for-loop over each transaction. A similar technique was later used by Orlando et al. in their DCP and DCI algorithms [11, 12].

Since the number of candidate 2-itemsets is exactly $\binom{|F_1|}{2}$, it is still possible that this number is too large, such that only part of the structure can be generated and multiple scans over the

database need to be performed. A lot of candidate 2-itemsets do not even occur at all in the database, and hence, their support remains 0. Therefore, we propose the following optimization. When all single items are counted, resulting in the set of all frequent items F_1, we do not generate any candidate 2-itemset. Instead, we start scanning the database, and remove from each transaction all items that are not frequent. Then, for each trimmed transaction, we increase the support of all candidate 2-itemsets contained in that transaction. However, if the candidate 2-itemset does not yet exists, we generate the candidate itemset and initialize its support to 1. In this way, only those candidate 2-itemsets that occur at least once in the database are generated.

5.3. Support lower bounding

Apart from the monotonicity property, it is sometimes possible to derive information on the support of an itemset, given the support of all of its subsets. The first algorithm that uses such a technique was proposed by Bayardo in his MaxMiner and Apriori-LB algorithms [6].

In practice, this lower bound can be used in the following way. Every time a candidate ($k + 1$) - itemset is generated by joining two of its subsets of size k, we can easily compute this lower bound for that candidate. Indeed, suppose the candidate itemset $X \cup \{i_1, i_2\}$ is generated by joining $X \cup \{i_1\}$ and $X \cup \{i_2\}$, we simply add up the supports of these two itemsets and subtract the support of X. If this lower bound is higher than the minimal support threshold, then we already know that it is frequent and hence, we can already generate candidate itemsets of larger sizes for which this lower bound can again be computed. Nevertheless, we still need to count the exact supports of all these itemsets, but this can be done all at once during the support counting procedure. Using the efficient support counting mechanism as I described before, this optimization could result in significant performance improvements.

Calders and Goethals presented a generalization of all these techniques resulting in a system of deduction rules that derive tight bounds on the support of candidate itemsets [9]. These deduction rules allow for constructing a minimal representation of all frequent itemsets, but can also be used to efficiently generate the set of all frequent itemsets. Unfortunately, for a given candidate itemset, an exponential number of rules in the length of the itemset need to be evaluated. The rules presented in this section, which are part of the complete set of derivation rules, are shown to result in significant performance improvements, while the other rules only show a marginal improvement.

5.4. Dynamic Itemset Counting

The DIC algorithm, proposed by Brin et al. tries to reduce the number of passes over the database by dividing the database into intervals of a specific size [8]. First, all candidate patterns of size 1 are generated. The supports of the candidate sets are then counted over the first interval of the database. Based on these supports, a new candidate pattern of size 2 is already generated if all of its subsets are already known to be frequent, and its support is counted over the database together with the patterns of size 1. In general, after every interval, candidate patterns are generated and counted. The algorithm stops if no more candidates can be generated and all candidates have been counted over the complete database. Although this method drastically reduces the number of scans through the database, its performance is also heavily dependent on the distribution of the data.

Although the authors claim that the performance improvement of reordering all items in support ascending order is negligible, this is not true for Apriori in general. Indeed, the reordering used in DIC was based on the supports of the 1-itemsets that were computed only in the first interval. Obviously, the success of this heuristic also becomes highly dependent on the distribution of the data.

The CARMA algorithm (Continuous Association Rule Mining Algorithm), proposed by Hidber [10] uses a similar technique, reducing the interval size to 1. More specifically, candidate itemsets are generated from every transaction. After reading a transaction, it increments the supports of all candidate itemsets contained in that transaction and it generates a new candidate itemset contained in that transaction, if all of its subsets are suspected to be relatively frequent with respect to the number of transactions that has already been processed. As a consequence, CARMA generates a lot more candidate itemsets than DIC or Apriori. Additionally, CARMA allows the user to change the minimal support threshold during the execution of the algorithm. After the database has been processed once, CARMA is guaranteed to have generated a superset of all frequent itemsets relative to some threshold which depends on how the user changed the minimal support threshold during its execution. However, when the minimal support threshold was kept fixed during the complete execution of the algorithm, at least all frequent itemsets have been generated. To determinate exact supports of all generated itemsets, a second scan of the database is required.

5.5. Sampling

The sampling algorithm, proposed by Toivonen [16], performs at most two scans through the database by picking a random sample from the database, then finding all relatively frequent patterns in that sample, and then verifying the results with the rest of the database. In the cases where the sampling method does not produce all frequent patterns, the missing patterns can be found by generating all remaining potentially frequent patterns and verifying their supports during a second pass through the database. The probability of such a failure can be kept small by decreasing the minimal support threshold. However, for a reasonably small probability of failure, the threshold must be drastically decreased, which can cause a combinatorial explosion of the number of candidate patterns.

6. Conclusions

A lot of people have implemented and compared several algorithms that try to solve the frequent itemset mining problem as efficiently as possible. Unfortunately, only a very small selection of researchers put the source codes of their algorithms publicly available such that fair empirical evaluations and comparisons of their algorithms become very difficult.

Different implementations of the same algorithms could still result in significantly different performance results. Different compilers and different machine architectures sometimes showed different behavior for the same algorithms. Also, different kinds of data sets on which the algorithms were tested showed remarkable differences in the performance of such algorithms.

In this paper I presented some algorithms which made a significant contribution to improve the efficiency of frequent itemset mining. Also, I propose two implementations for Apriori and AprioriTid algorithms.

References

[1] R. Agrawal, T. Imielinski, and A.N. Swami, Mining association rules between sets of items in large databases, *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, volume 22(2) of SIGMOD Record, pages 207–216. ACM Press, 1993.

[2] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A.I. Verkamo. Fast discovery of association rules, *Advances in Knowledge Discovery and Data Mining*, pages 307–328. MIT Press, 1996.

[3] R. Agrawal and R. Srikant, Fast algorithms for mining association rules, *Proceedings 20th International Conference on Very Large Data Bases*, pages 487–499. Morgan Kaufmann, 1994.

[4] R. Agrawal and R. Srikant, *Fast algorithms for mining association rules*. IBM Research Report RJ9839, IBM Almaden Research Center, San Jose, California, June 1994.
[5] A. Amir, R. Feldman, and R. Kashi. *A new and versatile method for association generation. Information Systems*, 2:333–347, 1997.
[6] R.J. Bayardo, Efficiently mining long patterns from databases, *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, volume 27(2) of SIGMOD Record, pages 85–93. ACM Press, 1998.
[7] C. Borgelt and R. Kruse, Induction of association rules: Apriori implementation, *Proceedings of the 15th Conference on Computational Statistics*, pages 395–400,
[8] S. Brin, R. Motwani, J.D. Ullman, and S. Tsur, Dynamic itemset counting and implication rules for market basket data*, Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, volume 26(2) of SIGMOD Record, pages 255–264. ACM Press, 1997.
[9] T. Calders and B. Goethals, Mining all non-derivable frequent itemsets, Proceedings *of the 6th European Conference on Principles of Data Mining and Knowledge Discovery*, volume 2431 of Lecture Notes in Computer Science, pages 74–85. Springer, 2002.
[10]C. Hidber, Online association rule mining, Proceedings *of the ACM SIGMOD International Conference on Management of Data*, volume 28(2) of SIGMOD Record, pages 145–156. ACM Press, 1999.
[11]S. Orlando, P. Palmerini, and R. Perego, Enhancing the apriori algorithm for frequent set counting, *Proceedings of the Third International Conference on Data Warehousing and Knowledge Discovery*, volume 2114 of Lecture Notes in Computer Science, pages 71–82. Springer, 2001.
[12]S. Orlando, P. Palmerini, R. Perego, and F. Silvestri, Adaptive and resource-aware mining of frequent sets, *Proceedings of the IEEE International Conference on Data Mining. IEEE Computer Society*,2002.
[13]J.S. Park, M.S. Chen, and P.S. Yu, An effective hash based algorithm for mining association rules, *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, volume 24(2) of SIGMOD Record, pages 175–186. ACM Press, 1995.
[14]R. Srikant, *Fast algorithms for mining association rules and sequential patterns,* PhD thesis, University of Wisconsin, Madison, 1996.
[15]R. Srikant and R. Agrawal, *Mining generalized association rules*, 1999, pages 407–419.
[16]H. Toivonen, Sampling large databases for association rules, *Proceedings 22nd International Conference on Very Large Data Bases*, pages 134–145. Morgan Kaufmann, 1996.
[17]HAN Feng, ZHANG Shu-mao, DU Ying-shuang, The analysis and improvement of Apriori algorithm, *Journal of Communication and Computer*, ISSN1548-7709, USA, Sep. 2008, Volume 5, No.9 (Serial No.46), pages 12-18

Ioan Daniel Hunyadi
“Lucian Blaga” University of Sibiu
Informatics Departament
Dr. Ratiu No. 5-7, Sibiu
Romania
daniel.hunyadi@ulbsibiu.ro