# Mining Association Rules in Large Database by Implementing Pipelining Technique in Partition Algorithm

**Article** *in* International Journal of Computer Applications · May 2010

DOI: 10.5120/657-924 · Source: DOAJ

Nirmal Chandra Mahanti
Birla Institute of Technology, Mesra
**55** PUBLICATIONS **364** CITATIONS

*International Journal of Computer Applications (0975 – 8887)*
*Volume 2 – No.4, June 2010*

# Mining Association Rules in Large Database by Implementing Pipelining Technique in Partition Algorithm

Kanhaiya Lal
Senior Lecturer
Department of Computer Sc. & Engg.
Birla Institute of Technology
Patna Campus, India

N.C.Mahanti
Professor & Head
Department of Applied Mathematics
Birla Institute of Technology
Mesra Ranchi, India

## ABSTRACT

Mining for association rules between items in a large database of sales transactions has been described as an important database mining problem. In this paper we present an efficient algorithm for mining association rules that is faster than the previously proposed partition algorithms approximately m times where m is the number of stages in pipeline. The algorithm is also ideally suited for parallelization.

## General Terms

Database, Data Mining, Algorithms

## Keywords

Association rules, Partition, pipeline, parallelization

## 1. INTRODUCTION

Association rule mining is to find out association rules that satisfy the predefined minimum support and confidence from a given database. The problem is usually decomposed into two sub problems. One is to find those item sets whose occurrences exceed a predefined threshold in the database; those item sets are called frequent or large item sets. The second problem is to generate association rules from those large item sets with the constraints of minimal confidence. Suppose one of the large item sets is $L_k$, $L_k = \{I_1, I_2, \dots, I_k\}$, association rules with this item sets are generated in the following way: the first rule is $\{I_1, I_2, \dots, I_{k-1}\} \Rightarrow \{I_k\}$, by checking the confidence this rule can be determined as interesting or not. Then other rule are generated by deleting the last items in the antecedent and inserting it to the consequent, further the confidences of the new rules are checked to determine the interestingness of them[7]. Those processes iterated until the antecedent becomes empty. Since the second sub problem is quite straight forward, most of the researches focus on the first sub problem. The first sub-problem can be further divided into two sub-problems: candidate large item sets generation process and frequent item sets generation process. We call those item sets whose support exceed the support threshold as large or frequent item-sets, those item sets that are expected or have the hope to be large or frequent are called candidate item sets [12]. In many cases, the algorithms generate an extremely large number of association rules, often in thousands or even millions. Further, the association rules are sometimes very large. It is nearly impossible for the end users to comprehend or validate such large number of complex association rules, thereby limiting the usefulness of the data mining results. Several strategies have been proposed to reduce the number of association rules, such as generating only "interesting" rules, generating only "non redundant" rules, or generating only those rules satisfying certain other criteria such as coverage, leverage, lift or strength[1].

Increasingly, business organizations are depending on sophisticated decision-making information to maintain their competitiveness in today's demanding and fast changing marketplace. Inferring valuable high-level information based on large volumes of routine business data is becoming critical for making sound business decisions. For example, customer buying patterns and preferences, sales trends etc can be learned from analyzing point-of sales data at supermarkets [13]. This information may be used for retaining market leadership by tuning to the needs of customers Database mining is motivated by such decision support problems and is described as an important area of research. One of the most difficult problems in database mining is the large volume of data that needs to be handled in a medium sized business; it is not uncommon to collect hundreds of megabytes to a few gigabytes of data. Database mining applications often perform long-running, complex data analysis over the entire database. Given the large database sizes, one of the main challenges in database mining is developing fast and efficient algorithms that can handle large volumes of data. Discovering association rules between items over *basket* data was introduced in [4]. Basket data typically consists of items bought by a customer along with the date of transaction, quantity, Price etc. Such data may be collected, for example, at supermarket checkout counters. Association rules identify the set of items that are most often purchased with another set of items. For example, An association rule may state that "95% of customers who bought items A and B also bought C and D."This type of information may be used to decide catalog design, store layout, product placement, target marketing, etc. [4].

## 2. PROBLEM DESCRIPTION

This section is largely based on the description of the problem in [4] and [5]. Formally, the problem can be stated as follows: Let L = { $i_1$, $i_2$…... $i_m$ } be a set of *m* distinct literals called items . D is a set of variable length transactions over L. Each transaction contains a set of items $I_i, i_j \ldots\ldots i_k$ I .A transaction also has an associated unique identifier called TID. An association rule is an implication of the form X => Y, where X, Y ⊂I, and X□Y=□; X is called the antecedent and Y is called the consequent of the rule.

In general, a set of items (such as the antecedent or the consequent of a rule) is called an item set. The number of items in an item set is called the *length* of an item set. Item sets of some length k are referred to as k -item sets. For an item set X.Y, if Y is an m –item set then Y is called an *m- extension* of X.

Each item set has an associated measure of statistical significance called *support*. For an item set X⊂ I, support(X) = s, if the fraction of transactions in D containing X equals s .A rule has a measure of its strength called the confidence. The confidence of a rule X => Y is computed as the ratio support (X U Y/ support(X)). [10]

The problem of mining association rules is to generate all rules that have support and confidence greater than some user specified minimum support and minimum confidence thresholds, respectively. This problem can be decomposed into the following sub problems:

1. All item sets that have support above the user specified minimum support are generated. These item set are called the large item sets. All others are said to be small.

2 .For each large Item set, all the rules that have minimum confidence are generated as Follows: For a large item set X and any Y ⊂ X, if support(X)/support (X-Y) > minimum confidence, then the rule X-Y => Y is a valid rule.

For example, let T1={A,B,C} T2={A,B,D}, T3={A,D,E} and T4= {A,B,D} be the only transactions in the database. Let the minimum support and minimum confidence is 0.5 and 0.8 respectively. Then the large item sets are the following: { A } , { B } , { D } , { A B } , { A D } and { A B D } The valid rules are B => A and D => A.

The second sub problem, i.e generating rules given all large item sets and their supports, is relatively straightforward however; discovering all large item sets and their supports is a nontrivial problem if the cardinality of the set of items, j Ij and the database, D are large. For example if |L| = m, the number of possible distinct item sets is $2^m$ .The problem is to identify which of these large number of item sets has the minimum support for the given set of transactions. For very small values of m, it is possible to setup $2^m$ counters, one for each distinct item set, and count the support for every item set by scanning the database once. However, for many applications m can be more than 10. Clearly, this approach is impractical It should be noted that only a very small fraction of this exponentially large number of item sets will have minimum support .Hence, it is not necessary to test the support for every item set. Even if practically feasible, testing support for every possible item set results in much waste effort. To reduce the combinatorial search space all algorithms exploit the following property: any subset of a large item set must also be large. For example, if a transaction contains item set ABCD, then it also contains A, AB, BC, ABC, etc

Conversely, all extensions of a small item set are also small. Therefore, if at some stage it is found that item set ADE is small, then none of the item sets which are extensions of ADE, i.e ADEF, ADEFG etc .need be tested for minimum support. All existing algorithms for mining association rules are variants of the following general approach: initially support for all item sets of length 1 (1-itemsets) are tested by scanning the entire database. The item sets that are found to be small are discarded. A set of 2- item sets called *candidate item sets* are generated by extending the large item sets generated in the previous pass by one (1-extensions) and their support is tested by scanning the entire database. Many of these item sets may turn out to be small, and hence discarded. The remaining item sets are extended by 1 and tested for support. This process is repeated until no larger item sets are found .In general; some k$^{th}$ iteration contains the following steps:

1. The set of candidate k-item sets is generated by 1-extensions of the large (k-1)-item sets generated in the previous iteration.

2. Supports for the candidate k item sets are generated by a pass over the database.

3 The item sets that do not have the minimum support are discarded and the remaining item sets are designated large k-item sets.

Therefore, only extensions of those item sets that are found to be large are considered in sub- sequent passes. This process is stopped when in some iteration n, no large item sets are generated. The algorithm m in this case, makes n database scans.[2]

## 3. PREVIOUS WORK

The problem of generating association rules was first introduced in[4] and an algorithm called *AIS* was proposed for mining all association rules .In [6] ,an algorithm called *SETM* was proposed to solve this problem using relational operations in a relational database environment .In[ 4] ,two new algorithms called *Apriori* and *AprioriTid* were proposed. These algorithms achieved significant improvements over the previous algorithms and were specifically applicable to large databases. In [4], the rule generation process was extended to include multiple items in the consequent and an efficient algorithm for generating the rules was also presented.

The algorithms vary mainly in
(a) How the candidate item sets are generated; and

(b) How the supports for the candidate item sets are counted. In [4], the candidate item sets are generated on the fly during the pass over the database. For every transaction, candidate item sets are generated by extending the large item sets from previous pass with the items in the transaction such that the new item sets are contained in that transaction. In [5] candidate item sets are generated using only the large item sets from the previous pass. It is performed by joining the large item set with itself. The resulting set is further pruned to exclude any item set whose subset is not contained in the previous large item sets. This technique produces a much smaller candidate set than the former technique. To count the supports for the candidate item sets, for each transaction the set of all candidate item sets that are contained in that transaction are identified. The counts for these item sets are then incremented by one. Apriori and AprioriTid differ based on the data structures used for generating the supports for candidate item sets.

In Apriori, bitmaps are generated for transactions as well as the candidate item sets. To determine whether a candidate item set is contained in a transaction, the corresponding bitmaps are compared. A hash tree structure is used to restrict the set of candidate item sets compared so that subset testing is optimized. In AprioriTid, after every pass, an encoding of all the large item sets contained in a transaction is used in place of the transaction. In the next pass, candidate item sets are tested for inclusion in a transaction by checking whether the large item sets used to generate the candidate item set are contained in the encoding of the transaction. In Apriori the subset testing is performed for every transaction in each pass. However, in AprioriTid, if a transaction does not contain any large item sets in the current pass, that transaction is not considered in subsequent passes. Consequently, in later passes, the size of the encoding of the transactions can be much smaller than the actual database. However in initial passes the size of the encoding can be larger than the database. A hybrid algorithm is proposed which uses Apriori for initial passes and switches to AprioriTid for later passes [2].

## 3.1 Partition Algorithm

The idea behind Partition algorithm is as follows. Recall that the reason the database needs to be scanned multiple number of times is because the number of possible item sets to be tested for support is exponentially large if it must be done in a single scan of the database .However, suppose we are given a small set of potentially large item sets, say a few thousand item sets Then the support for them can be tested in one scan of the database and the actual large item sets can be discovered. Clearly, this approach will work only if the given set contains all actual large item sets. Partition algorithm accomplishes this in two scans of the database. In one scan it generates a set of all potentially large item sets by scanning the database once this set is a superset of all large item sets, i.e. it may contain false positives. But no false negatives are reported. During the second scan counters for each of these item sets are setup and their actual support is measured in one scan of the database.

The algorithm executes in two phases. In the first phase, the Partition algorithm logically divides the database in to a number of non-overlapping partitions. The partitions are considered one at a time and all large item sets for that partition are generated at the end of phase I, these large item sets are merged to generate a set of all potential large item sets. In phase II the actual support for these item sets are generated and the large item sets are identified The partition sizes are chosen such that each partition can be accommodated in the main memory so that the partitions are read only once in each phase.

We assume the transactions are in the form < TID, $I_j$ $i_{k,\ldots\ldots}i_n$ >.The items in a transaction are assumed to be kept sorted in the lexicographic order. Similar assumption is also made in [5].It is straight-forward to adapt the algorithm to the case where the transactions are kept normalized in <TID, item> form. We also assume that the TIDs are monotonically increasing. This is justified considering the nature of the application. We further assume the database resides on secondary storage and the approximate size of the database in blocks or pages is known in advance the items in an item set are also kept sorted in lexicographic order.

| Notation | Meaning |
|---|---|
| $C^P_K$ | A local candidate k-item set in partition p |
| $L^P_K$ | A local large k-item set in partition p |
| $S^P$ (l) | Support for an item set l within partition p |
| $C^P_K$ | Set of local candidate k-itemsets in partition p |
| $L^P_K$ | Set of local large k-itemsets in partition p |
| $C^G_K$ | Set of global candidate k-item sets |
| $C^G$ | Set of all global candidate item sets |
| $L^G_K$ | Set of global large k-item sets |
| $S^G$ (l) | Support for a global item set l |

[3]

**Table-1. Notations & Meaning**

### 3.1.1 Definition

A partition p $\subseteq$ D of the database refers to any subset of the transactions contained in the database D .Any two different partitions are non-overlapping, i.e.
$P_I$ □$P_j$ =□ , $I \neq j$. We define *local* support for an item set as the fraction of transactions containing that item set in a partition. We define a local large item set as an item set whose local support in a partition is at least the user defined minimum support In other words ,a local large item set is large only in the context of a partition (i.e., consider the entire database as consisting of only that partition). A local candidate item set is an item set that is being tested for minimum support within a given partition. A local large item set may or may not be large in the context of the entire database

We define the *global* support, *global* large item set, and global candidate item set as above except they are in the context of the entire database D. Clearly our goal is to find all global large item sets. We use the notation shown in Table in this paper. Individual item sets are represented by small letters and sets of item sets are represented by capital letters. When there is no ambiguity we omit the partition number when referring to a local item set We use the notation c[1],c2,……..c[k] to represent a k-item set c consisting of items c[1],c2,……c[k].

[3]

## 4. PIPELINING TECHNIQUE

Pipelining is a general technique for increasing processor throughput without requiring large amount of extra hardware. It increases overall throughput of an of an instruction set processor .A pipeline' s performance can be measured by its throughput in terms of millions of instruction executed per second or MIPS. Another popular measure of performance is the number of clock cycles per instruction or CPI. These quantities are related by the equation

CPI=f/MIPS …… (1)

Where f is the pipeline's clock frequency in MHz and the values of CPI and MIPS are average figures that can be determined experimentally by processing suits of representative programs. The maximum value of CPI for a single pipeline is 1, making the pipeline‘s maximum possible throughput equal to f.

This throughput is attained only when the pipeline is supplied with a continuous stream of instructions that keeps all the stages busy.

Another general measure of pipeline's performance is to speedup S (m) defined by

S (m) =T (1)/T (m). …. (2)

Where T (m) is the execution time for some target workload on an m-stage pipeline and T (1) is the execution time for the same workload on a similar, non pipelined processor.

It is reasonable to assume that T (1) <=tm (m), in which case S (m) <=m.A pipeline's efficiency and speedup are related as follows:

S (m) =m E (m) ……….
(3)

[3]

### 4.1 OPTIMIZING m:-

Equation (3) suggest that an easy way to improve a pipeline's performance is to increase the number of stages m. this assumes that the pipeline's processing task can be subdivided into a useful ways and that the cost of doing so is acceptable. Each new stage $S_i$ introduces some new hardware cost and delay due to buffer register $R_i$ and associated control logic. In particular, we will determine the pipeline's performance /cost ratio PCR defined as

PCR=f/K ………… (4)

Where f is the pipeline's clock frequency and K is its hardware cost.

Suppose the pipeline P has m stages and implements a particular set of operations (instructions) SI. Let a be the delay of an efficient, non pipelined processor that also implements SI.

It is reasonable to assume that each stage $S_i$ of P has delay a/m-that is ,m times less than the corresponding non- pipelined processor – plus some extra delay b due to $S_I$'s buffer register $R_I$ . Hence if $T_c$=1/f is P's clock period, we can write

$T_c$ =a/m+b ….
(5)

The pipeline's hardware cost can be estimated by

K=cm+d
…..(6)

Where c is the buffer –register cost per stage and d is the cost of the pipeline's (combinational) data-processing logic.

From (4), (5) and (6), we have

$PCR^{-1}=T_cK$= (a/m+b) (cm+d). …
(7)

So

PCR =m/[$bcm^2$+(ac+bd)m+ad] …(8)

To maximize PCR with respect to the number of stages m , we differentiate with respect to m and equate the result to zero.

Using standard differentiation by parts formula

d/dm(u/v)=1/v(du/dx)-u/$v^2$(dv/dx)

we obtain

d/dm (PCR)=1/v-m(2bcm+ac+bd)/$v^2$ …(9)

where u=m and v=$bcm^2$+(ac+bd)m+ad

on equating (9) with zero, we get

v=m(2acm+ad+bc).

Substituting for v and solving for m yields the value $m_{opt}$ of m that maximizes PCR, namely,

$m_{opt}=\sqrt{(ab/bc)}$ …(10)

the optimum number of stages is the integer closest to $m_{opt}$ .[2]

## 5. OUR ALGORITHM FOR GENERATING LOCAL LARGE ITEMSET

In our algorithm we partition the database according to partition algorithm. After partitioning the database we use the concept of pipeline technique. We sequentially put the partitions in an array in reverse order i.e $p_n$, $p_{n-1}$, $p_{n-2}$....$p_3$, $p_2$, $p_1$. (Fig.1)

Let us take m stage pipeline

In each ith pipeline stage, we generate item set of i length, compares the support of each item set with min. support. Those item set whose support is less than min. support are pruned and the rest are passed to i+1th stage. If support of all item set in ith stage is less than min. support than all the item sets in that stage of pipeline are considered as the local maximum for that partition of the database ($L^p{}_k$).

If even in the last stage of pipeline support of some item sets have support larger than min. support then those item sets are considered as local large k-item set in that partition($L_k^p$).

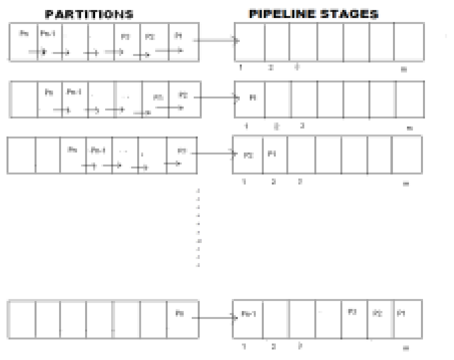


Fig.1 – Partitioning database & pipeline
[8]

***5.1 ALGORITHM:***

```
For( i = 1; i < n ; i++ )
{
Read in partition $p_i$ ($p_i$ belongs to p);
$L^i$=gen_i_length_itemsets($p_i$);
f(x);//shifting partitions
f(y);//shifting contents in pipeline stages to next stage
}
```

<u>Procedure for gen_i_length_itemsets($p_i$)</u>

```
$L_1^p$={ large 1- itemsets along with their tid lists }
For( k = 1 ; k ≤ m ; k++ )
{
For all itemsets $l_1 \in L_{k-1}^p$ do begin
For all itemsets $l_2 \in L_{k-1}^p$ do begin
If ( $l_1[1]=l_2[1] \wedge l_1[2]=l_2[2] \wedge \ldots l_1[k-1]<l_2[k-1]$ ) then
{
c= $l_1[1].l_1[2].l_1[3].l_1[4] \ldots l_1[k-1].l_2[k-1]$;
}
If c cannot be pruned then
c.tidlist=$l_1$.tidlist ^$l_2$.tidlist
if( |c.tidlist|/|p|>=min. Sup.) then
$L_k^p = L_k^p \cup \{ c \}$
End
End
} return $\cup_k L_k^p$
```

<u>For shifting partitions right (f(x))</u>

```
{int array[n]
for (i=n-1;i>=0;i--)
{
array [i]=array[i-1];
}}
```

<u>For shifting contents in pipeline stages to next stage (f(y))</u>

```
{int array[m]
for (i=m-1;i>=0;i--)
{
array [i]=array[i-1];
}}
```

X={pipeline, partition}

F(pipeline);(Now shifting algorithm for pipeline)

F (partition);(shifting of partition to pipeline)

**5.1.1 Generation of Local Large Item sets:-**

The **procedure gen_large_itemsets** takes a partition and generates all large itemsets (of all lengths) for that partition. The procedure is shown in Figure 2. Lines 3-8 show the candidate generation process. The prune step is performed as follows:

```
**Prune** (c: k-itemset)
**Forall** (k-1) subsets s of c **do**
**If** s doesnot belongs to $L_{k-1}$ **then**
**Return** “c can be pruned"
```

The prune step eliminates extensions of (k-1)-itemsets which are not found to be large, from being considered for counting support. For example, if $L_p^3$ is found to be { { 1 2 3 } , { 1 2 4 } , { 1 3 4 } ,{ 1 3 5 } , { 2 3 4 } } the candidate generation initially generates the itemsets { 1 2 3 4 } and { 1 3 4 5 }. However , itemset { 1 3 4 5 } is pruned since { 1 4 5 } is not in $L_p$ . This technique is same as the one described in [4] except in our case, as each candidate itemset is generated, its count is determined immediately.

The counts for the candidate itemsets are generated as follows. Associated with every itemset, we define a structure called as tidlist .A tidlist for itemset l contains the TIDs of all transactions that contain the itemset l within a given partition. The TIDs in a tid list are kept in sorted order. We represent the tidlist for an itemset l by l tid list .Clearly, the cardinality of the tid list of an

itemset divided by the total number of transactions in a partition gives the support for that itemset in that partition. [9, 11]
Initially, the tid lists for 1-itemsets are generated directly by reading the partition. The tid list for a candidate k-itemset is generated by joining the tid lists of the two (k-1)-itemsets that were used to generate the candidate k itemset. For example, in the above case the tid list for the candidate itemset { 1 2 3 4 } is generated by joining the tid lists of itemsets { 1 2 3 } and { 1 2 4 }.

**Procedure gen_final_counts** ( $C^G$: global candidate set, $p^r$ database partition)

```
1) forall 1-itemsets do
2)     generate the tid list
3)     For ( k=2 ; $C_k^G \neq$ □ ; k++) do begin
4)     Forall k –itemset c belongs to $C_K^G$ do begin
5)     Templist = C[1].tidlist □ C[2].tidlist□………c[k].tidlist
6)     C.count = count + | templist |
7)     End
8)     End
```

[2]

#### 5.1.1.1 Generation of Final Large Itemsets :-

The global candidate set is generated as the union of all local large itemsets from all partitions. In phase II of the algorithm, global large itemsets are determined from the global candidate set; this phase also takes n (number of partitions) iterations. Initially, a counter is setup for each candidate itemsets and initialized to zero. Next, for each partition tidlists for all 1-itemsets are generated. The support for a candidate itemset in that partition is generated by intersecting the tidlists of all 1-subsets of that itemset the cumulative count gives the global support for the itemsets

#### 5.1.1.2 Correctness:-

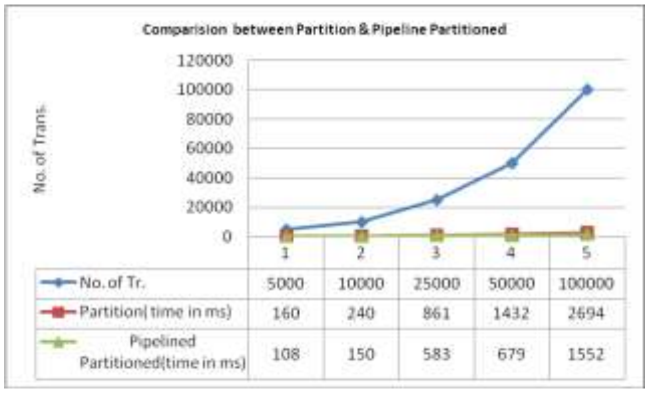
As shown earlier, steps 5-6 generate the support for an itemset in the given partition. Since the partitions are non-overlapping, accumulative count overall partitions gives the support for an item set in the entire database.[2]

#### 5.1.1.3 Discovering Rules

Once the large itemsets and their supports are determined , the rules can be discovered in a straight forward manner as follows : if l is a large itemset , then for every subset a of l , the ratio support ( l ) / support ( a ) is computed . If the ratio is at least equal to the user specified minimum confidence, them the rule a => (l - a) is output. [2]

## 6. Experimental Result

We implemented the algorithms in JAVA on PIII(x-86 Intel) processor (single processor), frequency 700 MHz, Memory 64 MB to check the result clearly and found the efficiency of pipelined partitioned is much-2 better than partition algorithm. The test was conducted for 10000, 20000, 50000, 100000 and 200000 transactions with 15 items & 0.36 min. support.



| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| No. of Tr. | 5000 | 10000 | 25000 | 50000 | 100000 |
| Partition( time in ms) | 160 | 240 | 861 | 1432 | 2694 |
| Pipelined Partitioned(time in ms) | 108 | 150 | 583 | 679 | 1552 |

**Fig.2 – Comparative Result**

## 7. CONCLUSIONS

If total number of itemset is k, and there are n equal partitions of itemsets i.e. k/n itemsets per partition, then in i'th pipeline stage (where itemsets of i length are generated) the number of combinations of itemsets generated is ${}^{k/n-a1-a2-\ldots-a(i-1)}c_i$ where $a_i$ are those itemsets whose support is less than the given minimum support and are pruned.

Number of combinations generated in Apriori algorithm is $2^k$ where k is the number of itemsets.

Since from binomial theorem,

$$(1+x)^n = {}^nc_0x+{}^nc_1x^2+{}^nc_2x^3+\ldots\ldots+{}^nc_{n-1}x^{n-1}+{}^nc_nx^n$$

Thus putting x=1,

$${}^nc_0+{}^nc_1+{}^nc_2+\ldots\ldots+{}^nc_{n-1}+{}^nc_n = 2^n$$

also we know that,

$${}^nc_k > {}^{n-a}c_k \text{ for any values of } a\neq 0$$

so,

$${}^{k/n}c_1+{}^{k/n-a1}c_2+{}^{k/n-a1-a2}c+\ldots\ldots+{}^{k/n-a1-a2\ldots\ldots-a(m-1)}c_m < {}^nc_0+{}^nc_1+{}^nc_2+\ldots\ldots+{}^nc_{n-1}+{}^nc_n \leq 2^k$$

i.e. ${}^{k/n}c_1+{}^{k/n-a1}c_2+{}^{k/n-a1-a2}c+\ldots\ldots+{}^{k/n-a1-a2\ldots\ldots-a(m-1)}c_m \leq 2^k$

In first stage of pipeline,

No of combinations=${}^{k/n}c_1$

Let $a_1$ itemsets have support less than minimum support in first stage i.e. they are pruned.

In second stage of pipeline,

No of combinations=${}^{k/n-a1}c_2$

Let $a_2$ itemsets have support less than minimum support in second stage i.e. they are also pruned.

For third stage,

No of combinations= ${}^{k/n-a1-a2}c_3$

Similarly some of itemsets are eliminated at every stage.

At mth stage,

No of combination=${}^{k/n-a1-a2\ldots\ldots-a(m-1)}c_m$

Total number of combination is the summation of all stages i.e.

$^{k/n}c_1 + {}^{k/n-a1}c_2 + {}^{k/n-a1-a2}c + \ldots\ldots + {}^{k/n-a1-a2\ldots\ldots-am}c_m \leq 2^k$

which is obviously true.

This shows that proposed algorithm is better than apriori algorithm.

If we consider $t_1$ as the time taken by partition algorithm to generate local large itemset and $t_2$ to generate global large itemset i.e. total time taken $=t_1+t_2$

Since the proposed algorithm uses m-stage pipeline to generate local large itemset therefore time taken is $t_1/m+t_2$.

Hence the proposed algorithm performs better than the partition algorithm.

## 8. REFERENCES

[1] Sotiris Kotsiantis, Dimitris Kanellopoulos "Association Rules Mining: A Recent Overview", GESTS International Transactions on Computer Science and Engineering, Vol.32 (1), 2006, pp. 71-72.

[2] Ashok savasere, Edward Omiecinski, Shamkant Navathe-"An Efficcient Algorithm for Mining Association Rules in Large Databases", Technical Report No. GIT-CC-95-04.

[3] John P. Hayes,"Computer Architecture and Organizaton", 3/e, McGraw-HILL INTERNATIONAL EDITIONS, Computer Science Series, pp- 275-292 (1998).

[4] R.Agrawal,T.Imielinski,and A.Swami. "Mining association rules between sets of items in large databases", In Proceedings of the ACM SIGMOD International Conference on management of data, Washington, DC May 26-8 1993.

[5] R.Agrawal and R.Srikant. "Fast algorithms for mining association rules", In Proceedings of the 20th VLDB Conference Santiago, Chile, 1994.

[6] M.Houtsma and A. Swami. "Mining sequential Patterns-Set-oriented mining of association rules", Technical Report RJ 9567, IBM October1993.

[7] Han J, Kamber M. "Data Mining: Concepts and Techniques". 2/e San Francisco: CA. Morgan Kaufmann Publishers, an imprint of Elsevier. Chapter 5, (2006).

[8] Ying-Hsiang Wen, Jen-Wei Huang and Ming-Syan Chen," Hardware-Enhanced Association rule Mining with Hashing and Pipelining", IEEE Transactions on Knowledge and Data Engineering, Vol.(20), No.6, pp784-794, June 2008.

[9] Qihua Lan, Defu Zhang, Bo Wu," A New Algorithm For frequent Itemsets Mining Based On Apriori And FP-Tree", Global Congress on Intelligent Systems, IEEE, pp-360-363, 2009.

[10] Liu Hong-min," Study and Implementation of Association Rule Algorithm in Data Mining", International Conference on Signal Processing Systems, IEEE, pp 821-824, 2009.

[11] Fudailah Duemong, Ladda Preechaveerakul and Sirirut Vanichayobon," FIAST: A Novel Algorithm for Mining Frequent Itemsets", International Conference on Future Computer and Communication, IEEE, pp – 140-143, 2009.

[12] Pang-Ning Tan, Michael Steinbach & Vipin Kumar, "Introduction to data Mining", Pearson education, Inc. & Dorling Kindersley pub. Inc., 4th Impression, 2009.

[13] Jun Gao," An New Algorithm of Association Rule Mining", International Conference on Computational Intelligence and Security, IEEE, pp 117-120, 2008.