Frequent Data Itemset Mining Using VS_Apriori Algorithms

N. Badal
Department of Computer Science & Engineering,
Kamla Nehru Institute of Technology,
Sultanpur (U.P.), India
n_badal@hotmail.com

Shruti Tripathi
Department of Computer Science & Engineering,
Feroze Gandhi Institute of Engineering and Technology,
Raebareli (U.P.), India
shru_tri@yahoo.com

***Abstract*—The organization, management and accessing of information in better manner in various data warehouse applications have been active areas of research for many researchers for more than last two decades. The work presented in this paper is motivated from their work and inspired to reduce complexity involved in data mining from data warehouse.**

A new algorithm named VS_Apriori is introduced as the extension of existing Apriori Algorithm that intelligently mines the frequent data itemset in large scale database.

Experimental results are presented to illustrate the role of Apriori Algorithm, to demonstrate efficient way and to implement the Algorithm for generating frequent data itemset. Experiments are also performed to show high speedups.

Keywords:- **Frequent data itemsets, Apriori**

I. INTRODUCTION

The world around us is full of information. The latest computer systems are allowing us to gather and store that information in the large scale Data Warehouse (DW). However, the ability to process and retrieve that information lags far beyond the abilities as gatherers. Most of the real problems of this time are to extract the meaningful patterns from datasets and Dataset Item from DW.

The work presented in this paper is oriented about intelligent data mining process for depended and demanded Item datasets. These processes contain the extraction of meaningful information from massive datasets. The data sets of item have the relationship between them. Even quite defining what is a meaningful relationship among these item data is non-trivial; but, that said, determining sets of items that co-occur frequently throughout the data is a very good start.

The work is initiated with classic Apriori algorithm for the problem to find the frequent data itemsets. Further, the concept of 'sorting' the consequence is being introduced with classic Apriori to provide more reliable and specific results.

The real problem of processing and retrieving useful data itemset in intelligent manner from the large scale dataset was left unattended. The classical Apriori algorithm is a popular and foundational member of the correlation based intelligent data mining kernels used today for data itemset. However, it is a computationally expensive algorithm in terms of multiple passes. Therefore, it is required to analyze the classical Apriori algorithm for better efficiency in a good space complexity. Further, there is a need of a new algorithm to improve the efficiency of existing one and able to provide the intelligent data mining of itemset.

Section II reviews the literature on frequent data itemset mining techniques. Section III produces a new algorithm VS_Apriori as an extension of classic Apriori algorithm with details of quite thoroughly how the work modifies the original algorithm in order to achieve the better efficiency. Experiments done in support of the proposed algorithm for frequent data itemset mining on sample test dataset is given in Section IV. Finally, in Section V conclusion and future scope of the work is given.

II. BACKGROUND

Before explaining the sorting method, let's first review the problem definition and the previous attempts at addressing the problem, especially including the basic flow of the Apriori algorithm.

The task of frequent data itemset mining was first introduced by Agrawal & Srikant in [11]. A frequent data is the set of one or more items that often occur in a database, and often occurs together in the same basket within the database if it consists of more than one item. Each set of data has a number of items and is called a transaction. The output of Apriori is sets of rules that tell us how often items are contained in sets of data. Here is an example:

Each transaction is a set of items

TABLE I. A Figure Example Of A Dataset In Which {A, B} Is Frequent, Designed To Illustrate The Frequent Data Itemset Mining Problem

Transaction	Data Itemset
t1	a b c
t2	a b d
t3	a b e
t4	a b d

100% of sets with a also contain b

25% of sets with a, b also have c
50% of sets with a, b also have d
25% of sets with a, b also have c

The associations that Apriori finds are called Association rules. An association rule has two parts. The Antecedent is a subset of items found in sets of data. The Consequent is an item that is found in combination with the antecedent. Two terms describe the significance of the association rule. The Confidence is a percentage of data sets that contain the antecedent. The Support is a percentage of the data sets with the antecedent that also contain the consequent.

From the example of table I, we could find the following association rules

# consequence <- antecedent (confidence, support)
a <- b (100%, 100%) # b <- a is the same
c <- a b (100%, 25%)
d <- a b (100%, 50%)
c <- a b (100%, 25%)

A. *Apriori Algorithm*

Agrawal et al. proposed the classical Apriori algorithm [10] to more efficiently solve it. Classical Apriori algorithm can be divided into three sections. Initial frequent item sets are fed into the system, and candidate generation, candidate pruning, and candidate support is executed in turn.

If some set t contains some subset s, then it also contains all subsets of s. Considering this on a grander scale, if s is known to occur in, say, p transactions, then all subsets of s occur in at least p transactions, since they must occur in those transactions in which s occurs, even if no others. Stating this alternatively gives the Apriori Principle:

$$\text{Supp}(s_i) \geq \text{supp}(s_i \cup s_j), \text{ for all sets } s_i, s_j \quad (1)$$

It is common place to refer to these three happenings as Candidate Generation, Candidate Pruning, and Support Counting, respectively. For detail refer [10]. The Apriori algorithm works in following steps –

1) Candidate Generation Method:

How does one construct candidates of a particular size from a set of frequent data itemsets? Just taking the union of arbitrary sets is not going to produce new sets with exactly k +1 elements. Although there are a number of ways to choose sets to join, only one is used prominently and with much success: the $(k - 1) \times (k - 1)$ method. Consider two data itemsets of size k. Their union will contain precisely k+1 item exactly when they share

$$k + k - (k + 1) = k - 1 \text{ items.} \quad (2)$$

So, all candidates that are generated are done so from frequent data itemsets of size k that share in common $k - 1$ items. But consider the data itemsets of Table I

In so doing, one guarantees that a candidate will be generated only once.

2) Pruning the Search Space:

By only generating candidates from frequent $(k - 1)$-data itemsets as above, one makes use of the Apriori Principle. Candidates can only be generated if two particular subsets are frequent. If either happens to be infrequent, then the support of their union is necessarily infrequent, too—and as such the candidate is not generated.

3) Support Counting:

Finally, one must count the support of those candidate (k+1)-data itemsets that survived the pruning phase in order to determine whether they truly are frequent. Agrawal et al. accomplished this with a hashing scheme. They construct an array in which to store the counts for each candidate and a hash function that maps the candidates onto the array. They then scan the entire dataset and, for each transaction t in it, extract every size k + 1 subset of t and apply the hash function to the subsets. If a subset of t hashes to a candidate, they increment that candidate's support count. After proceeding through the entire dataset, they make one pass through the array and collect those candidates with support counts above the threshold. These are the frequent (k + 1)-data itemsets. It is important to note that the purpose of the hashing i for indexing, not for compression. Each candidate still retains its own support count.

B. *Recent advances*

A concise literature review of related work with mining of frequent data itemset is being presented in this section. The literature review is being partitioned in two sections. The first section is concern with the work related to the progress on frequent data itemset mining. The limitation of existing work is being given in the subsequent section.

Some ideas that are designed to address the issue of scalability are as follows.

1) Tries:

The first of these advances is, as introduced by Brin [14], of storing candidates in a trie. A trie (alternatively known as a prefix tree) is a data structure developed by Fredkin [7] which takes advantage of redundancies in the keys that are placed in the tree.

This approach has the potential to break down on large datasets if the data structure no longer fits in main memory. The depth of the trie is equal to the length of those candidates. To fit all nodes into main memory requires those candidates to overlap quite substantially. When they do not, the effect of the trie's heavily pointer-based makeup is very poor localisation and cache utilisation. Consequently, traversing it causes one to thrash on disk and the efficiency of the structure is quickly consumed by I/O costs.

2) FPGrowth:

Han et al. in [9] introduce a quite novel algorithm to solve the frequent data itemset mining problem. FPGrowth is a highly compact representation of all relevant frequency information in the data set. Every path of FPGrowth represents a frequent itemset and the nodes in the path are stored in decreasing order of the frequency of the corresponding items.

An FPGrowth has a header table. The nodes in the header table link to the same nodes in its FPGrowth. Single items and their counts are stored in the header table by decreasing order of their counts.

Next, the trie is mined recursively to extract the frequent data itemsets. By following the linked-list of nodes labelled by the least-frequent item, one retrieves all paths involving that item. Then a new conditional prefix tree can be built by copying and then modifying the original tree. All paths whose leaves are not labelled with the least-frequent item are removed, this least-frequent item is itself removed, duplicate paths are merged, and the trie is resorted based on the new conditional frequencies. This creates a trie with the same structure as the original tree, but conditioned on the presence of the least frequent item. So, the procedure can be repeatedly recursively from here until the trie consists of nothing but a root node denoting the empty set. This yields all frequent data itemsets involving the least-frequent item.

The procedure is then repeated for the second-least-frequent item, third-least frequent item, and so on to extract from the trie all frequent data itemsets.

Limitations of Existing Work:

he primary bottleneck of the classical Apriori algorithm is in incrementing counters for those candidates that are active in a particular transaction. The trie structure helps immensely in this regard because the process of matching a candidate to a transaction simultaneously accomplishes that of loading the appropriate counter because it is stored in the leaf of the trie.

But even this approach is not fast enough. When comparing nearly 1.7 million transactions to 30 million candidates as is done on the Webdocs dataset presented by Lucchese [6], the cost of everything is significantly magnified. However, the story changes when the dataset is quite large because it suffers the same consequences as did the trie of candidates. Even building the trie becomes extremely costly. Buehrer [5] remarked that the dominant percentage of execution time is that of constructing the trie. Consequently, on truly large datasets, the FPGrowth algorithm fails even to initialise.

The disadvantage of FP-Growth is that it needs to work out conditional pattern bases and build conditional FP-tree recursively. It performs badly in data sets of long patterns.

Another general problem with the FPGrowth algorithm is that it lacks the incremental behaviour of Apriori, something that builds fault tolerance into the algorithm.

III. VS_APRIORI ALGORITHM

The base of this method is the classical Apriori algorithm. This method provides novel scalable approaches for each building block.

To start with counting the support of every item in the dataset and sort them in decreasing order of their frequencies. Next, sort each transaction with respect to the frequency order of their items is horizontal sort. This method generates the candidate data itemsets such that they are also horizontally sorted. Furthermore, care to generate the candidate data itemsets in sorted order with respect to each other. This is vertical sort. When data itemsets are both horizontally and vertically sorted, they are fully sorted. Generating sorted candidate data itemsets (for any size k), both horizontally and vertically, is computationally free and maintaining that sort order for all subsequent candidate and frequent data itemsets requires careful implementation, but no cost in execution time. This conceptually simple sorting idea has implications for every subsequent part of the algorithm.

In particular, having transactions, candidates, and frequent data itemsets all adhering to the same sort order has the following advantages:

- Generating candidates can be done very efficiently
- Indices on lists of candidates can be efficiently generated at the same time as are the candidates
- Groups of similar candidates can be compressed together and counted simultaneously
- Candidates can be compared to transactions in linear time
- Better locality of data and cache-consciousness is achieved

Each of these advantages is detailed more thoroughly in the next sections.

A. *Efficiently Generating Candidates*

Let's consider generating candidates of an arbitrarily chosen size, k + 1. It will assume that the frequent k-data itemsets are sorted both horizontally and vertically. A small example if k were four is given in Table II.

TABLE II. Example Set of Frequent 4-Data itemsets

f1	6 5 3 2
f2	6 5 3 1
f3	6 5 3 0
f4	6 5 2 0
f5	6 5 1 0
f6	5 4 3 2
f7	5 4 3 0

As described in Section II.A.1, the $(k - 1) \times (k - 1)$ technique generates candidate (k+1)-data itemsets by taking the union of frequent k-data itemsets. If the first k−1 elements are identical for two distinct frequent k-data itemsets, f_i and f_j, call them near-equal and denote their near-equality by $f_i \approx f_j$. Then, classically, every frequent data itemset f_i is compared to every f_j and the candidate $f_i \cup f_j$ is generated whenever $f_i \approx f_j$. However, even in the small example it must verify this relationship for

$$\binom{7}{2} = 7 * 8 / 2 = 28 \tag{3}$$

pairs of frequent k-data itemsets. This step is too slow because the number of frequent k-data itemsets is so large.

A crucial observation is that near-equality is transitive because the equality of individual items is transitive. So, if $f_i \approx f_{i+1}, \ldots, f_{i+m-2} \approx f_{i+m-1}$ then it know that $(\forall j, k) < m, f_{i+j} \approx f_{i+k}$.

Recall also that the frequent k-data itemsets are fully sorted (that is, both horizontally and vertically), so all those that are near-equal appear contiguously. This sorting taken together with the transitivity of near-equality is what this method exploits. Consider the given example.

To begin, set a pointer to the first frequent data itemset, f1 = {6, 5, 3, 2}. Then check if f1= f2, f2= f3, f3= f4 and so on until the near-equality is no longer satisfied. This occurs between f2 and f3 because they differ on their 3rd items. Let m denote the number of data itemsets determined to be near-equal, 3 in this case. Then, because near-equality is transitive, it can take the union of every possible pair of the m = 3 data itemsets to produce the candidates. In this case, the three candidates {{6, 5, 3, 2, 1}, {6, 5, 3, 2, 0}, {6, 5, 3, 1, 0}} and in general $\binom{m}{2}$ candidates will be produced.

Then, to continue, set the pointer to f4 and proceed as before. See that f4 is not near-equal to f5, so have no pairs to merge. The pointer is next set to f5 for which the same can be said. Then set the pointer to f6 and verify that f6=f7. Since there are no more frequent data itemsets, pair f6 and f7 and the candidate generation is complete. The full set of candidates that will generate is {{6, 5, 3, 2, 1}, {6, 5, 3, 1, 0}, {6, 5, 3, 2, 0}, {5, 4, 3, 2, 0}}.

This successfully generates all the candidates with a single pass over the list of frequent k-data itemsets as opposed to the classical nested-loop approach. Strictly speaking, it might seem that the processing of $\binom{m}{2}$ candidates effectively causes extra passes, but it can be shown using the Apriori Principle that m is typically much less than the number of frequent data itemsets. At any rate, circumvent this as described in the next section.

1) Candidate compression

Let us return to the concern of generating $\binom{m}{2}$ candidates from each group of m near-equal frequent k-data itemsets. Since each group of $\binom{m}{2}$ candidates share in common their first k−1 items, it need not repeat the information. As such, this can compress the candidates into a super-candidate.

To illustrate this by reusing the example of Table II. Of those frequent 4-data itemsets, it is discovered that f_0, f_1, and f_2 are near-equal. From them, 0 = {6, 5, 3, 2, 1}, c_1 = {6, 5, 3, 2, 0}, c_2 = {6, 5, 3, 1, 0} would be generated as candidates. But instead consider $c = f_0 \cup f_1 \cup f_2$.

Then, the 2-tuple (k +m−1, c) = (6, {6, 5, 3, 2, 1, 0}) encodes all the information need to know about all the candidates generated from f0, f1, and f2. The first k − 1 items in the set c are common to all $\binom{m}{2}$ candidates; call this 2-tuple a super-candidate.

This new super-candidate still represents all $\binom{m}{2}$ candidates, but takes up much less space in memory and on disk.

The candidates in a super-candidate $c = (c_w, c_s)$ all share the same prefix: the first k − 1 items of c_s. They all have a suffix of size

$$(k + 1) - (k - 1) = 2 \quad (4)$$

By iterating in a nested loop over the last $c_w - k + 1$ items of c_s, produce all possible suffices in sorted order. These, each appended to the prefix, form the $\binom{c_w - k = 1}{2}$ candidates in c.

2) Indexing

There is another nice consequence of generating sorted candidates in a single pass: it can efficiently build an index for retrieving them. In this implementation and in the following example, It build this index on the least frequent item of each candidate (k + 1)-data itemset.

The structure is a simple two-dimensional array. Candidates of a particular size k+1 are stored in a sequential file, and this array provides information about offsetting that file. Because of the sort on the candidates, all those that begin with each item i appear contiguously. The exact location in the file of the first such candidate is given by the i^{th} element in the first row of the array. The i^{th} element in the second row of the array indicates how many bytes are consumed by all (k + 1)-candidates that begin with item i.

Consider again the example of Table II. The candidates generated, when stored sequentially as super candidates, appear as below:

6653210565310554320

TABLE III. Sample Index for Candidate 5-Data itemsets

Item	Offset	NumBytes
6	0	52
5	52	24
4	-1	-1
3	-1	-1
2	-1	-1
1	-1	-1
0	-1	-1

The first two super candidates have 6 as their first item and the third, 5. This creates a boundary between the second 0 and

the 5 that succeeds it. The purpose of the indexing structure is to keep track of where in the file that boundary is and offer information that is useful for block-reading along this boundary. Table III indicates how the structure would look if each of these numbers consumed four bytes. Use −1 in an i^{th} position as a sentinel to indicate that no candidates begin with item i.

Note that one could certainly index using the j least frequent items of each candidate, for any fixed $j < k+1$. As j is chosen larger, the index structure is more precise (returns fewer candidates that could not match the transaction) but consumes more memory.

This indicates that the idea of building an index on the candidates is not novel. In fact, this is quite apparent in [3, 11, 12]. However, nature of the indexing structure is very different. In [3, 11, 12], the candidates are compressed into a prefix tree in exactly the same way as transactions are compressed into an FPGrowth in FPGrowth. Consequently, this indexing structure can suffer the same fate as does an FPGrowth when the number of candidates causes the index to grow beyond the limits of memory.

This structure does not suffer from the troubles of [3, 11, 12], as is evident in three immediate ways. First, it is more likely to fit into memory, because it only requires storing three numbers for each item, not the entire set of candidates. Second, it partitions nicely along the same boundaries as the candidates are sorted; so, if the structure is too large to fit in memory, it can be easily divided into components that do. Third, it is incredibly quick to build.

B. *Candidate Pruning*

When Apriori was first proposed in [10], its performance was explained by its effective candidate generation. What makes the candidate generation so effective is its aggressive candidate pruning. This can be omitted entirely while still producing nearly the same set of candidates. Stated alternatively, after this particular method of candidate generation, there is little value in running a candidate pruning step.

Agrawal & Srikant [10] stated that the probability that a candidate is generated is shown to be largely dependent on its best testset — that is, the least frequent of its subsets. Classical Apriori has a very effective candidate generation technique because if any data itemset $c \setminus \{c_i\}$ for $0 \le i \le k$ is infrequent the candidate $c = \{c_0, \ldots, c_k\}$ is pruned from the search space. By the Apriori Principle, the best testset is guaranteed to be included among these. However, if one routinely picks the best testset when first generating the candidate, then the pruning phase is redundant.

In this method, on the other hand, a candidate generated from two particular subsets, $f_k = c \setminus \{c_k\}$ and $f_{k-1} = c \setminus \{c_{k-1}\}$.

If either of these happens to be the best testset, then there is little added value in a candidate pruning phase that checks the other k−2 size k subsets of c. Because of the least-frequent-first sort order, f_0 and f_1 correspond exactly to the subsets missing the most frequent items of all those in c. It observed that usually either f_0 or f_1 is the best testset.

C. *Index-Based Support Counting*

Returning to the example of Table II, it had concluded that three super-candidates would be generated: $\{c_0 = (6, \{6, 5, 3, 2, 1, 0\}), c_1 = (5, \{6, 5, 3, 1, 0\}), c_2 = (5, \{5, 4, 3, 2, 0\})\}$. To compare to a transaction, say $t = \{t_0 = 6, t_1 = 4, t_2 = 3, t_3 = 2, t_4 = 1, t_5 = 0\}$ it first look up t_0 in the index and retrieve the first two super-candidates, c_0 and c_1. Then compare them each to t and update the support counts if they are contained in t. (In this case, they are not.)

Next, proceed to t_1, looking it up in the index. It discovers that there are no candidates that begin with 4, so move along to t_2. However, since

$$w - i = 6 - 2 = 4 < k \tag{5}$$

There cannot possibly be any more candidates contained in t, so the method is done.

1) *Counting with compressed candidates*

Candidates can be compressed. This affords appreciable performance gains. All the candidates compressed into a super-candidate $c = (c_w, c_s)$ share their first k−1 elements. So, for a transaction t, if the first k−1 items of c_s are not strictly a subset of t, then it can immediately jump over $\binom{c_w - k + 1}{2}$ candidates. None could possibly be contained in t.

Suppose instead that the first k−1 items of c_s are strictly a subset of a transaction t. How does it increment the support counts of exactly those candidates in c which are contained in t (no more, no fewer)? Illustrate this by example. Let $t = \{6, 5, 4, 3, 2, 0\}$ be the transaction and, as before, $c = (c_w, c_s) = (6, \{6,5,3,2,1,0\})$ be the super-candidate and k + 1 = 5 be the size of the candidates. Lay out a linear array, A, of

$$\binom{c_w - k + 1}{2} = \binom{3}{2} = 2 \tag{6}$$

integers in which it keeps track of each candidate's support count.

Some items of c_s are also in t. Each has an index in c_s and keeps all such indices above $k-1$. This gives us $c' = \{3,5\}$ (corresponding to the items 3 and 0). Then subtract these indices from $c_w = 6$, producing $c'' = \{3,1\}$

Finally, increment the support counts for each of the $\binom{|c''|}{2}$ candidates contained in t.

To do so for elements i and j in c″ (with $i > j$, it increment

$$A\left[\binom{c_w - k + 1}{2} - 1 - x\right] \tag{7}$$

$$x = \binom{i}{2} + j - i$$

where

In this example, the only choices for i and j are i = 3 and j = 1, so

$$x = \binom{3}{2} + 1 - 3 = 1 \tag{8}$$

and only increment

$$A\left[\binom{3}{2} - 1 - x\right] = A[3 - 1 - 1] = A[1] \tag{9}$$

Reflecting on the super-candidate, it represented the candidates $c_0 = \{6,5,3,2,1\}, c_1 = \{6,5,3,2,0\}, c_2 = \{6,5,3,1,0\}$. Of these three, only c_1 is contained in t. The only integer incremented was $A[1]$. The mapping would increment $A[0]$ for c_0 and $A[2]$ for c_2.

This is how it consistently index the arrays, but certainly any mapping from

$$\{(i, j) : 0 < j < i \le c_w - k + 1\} \tag{10}$$

on to the interval $\left[0, \binom{c_w - k + 1}{2}\right]$ if applied consistently will work. In fact, one need not even map to such a tight interval if space is not a concern. It chose the mapping

$$\binom{cw - k + 1}{2} - 1 - \left(\binom{i}{2} + j - i\right) \tag{11}$$

because it has the nice property that order is maintained.

The VS_Apriori algorithm

Step 1- INPUT: A dataset D and a support threshold s
Step 2- F is set of frequent data itemsets
C is set of candidates
C ← U
Step 3- Scan database to count support of each item in C
Step 4- Add frequent items to F
Step 5- Sort F least-frequent-first (LFF) by support (using quicksort)
Step 6- Output F
Step 7- for all f ∈ F, sorted LFF do
for all g ∈ F, supp(g) ≥ supp(f), sorted LFF do
Add {f, g} to C
end for
Update index for item f
end for
Step 8- while |C| > 0 do
Step 9- {Count support}
for all t ∈ D do
for all i ∈ t do
RelevantCans ← using index, compressed cans from file that start with i
for all CompressedCans ∈ RelevantCans do
if First k − 2 elements of CompressedCans are in t then
Use compressed candidate support counting technique to update appropriate support counts
end if
end for
end for
end for
Add frequent candidates to F
Output F
Clear C
Step 10- {Generate candidates}
Start ← 0
for 1 ≤ i ≤ |F| do
if i == |F| OR fi is not near-equal to f_i−1 then
Create super candidate from fstart to f_i−1 and update index as necessary
Start ← i
end if
end for
Step 11- {Candidate pruning—not needed!}
//Clear F
//Reset hash
end while
Step 12- OUTPUT: All sets that appear in at least s transactions of D

Figure 1. VS_Apriori Algorithm

IV. EXPERIMENTAL RESULTS

The new VS_Apriori algorithm is being implemented in this section. It details the results of the experiments on a well-known benchmark test dataset. It is compared against two state-of-the-art implementations that were all designed with the same dataset in mind. Then the new VS_Apriori is being compared to the classic Apriori algorithm with the help of test dataset.

The proposed algorithm may be test to demonstrate and comparing the work on Dual-core Intel Xeon Processor, 2.33GHz/1333MHz, 4Mb L2 machine.

The 1.5 GB of Webdocs data by Lucchese et. al [6] used for study, being the largest dataset commonly used throughout publications on this problem. The data in the Webdocs set comes from a real domain and so is meaningful. Constructing a

random dataset will not necessarily portray the true performance characteristics of the algorithms.

The correctness of methods implementation's output is compared to the output of other algorithms. Since they were all developed for the FIMI workshop and all agree on their output, nonetheless, boundary condition checking was a prominent component during development.

In order to verify the performance of the Classical algorithm and VS_Apriori algorithm Figure 2 and Figure 3 shows the comparison between classical Apriori algorithm and the VS_Apriori algorithm. In Figure 2 comparison is between Run time vs. Support threshold with transaction considered 25000.

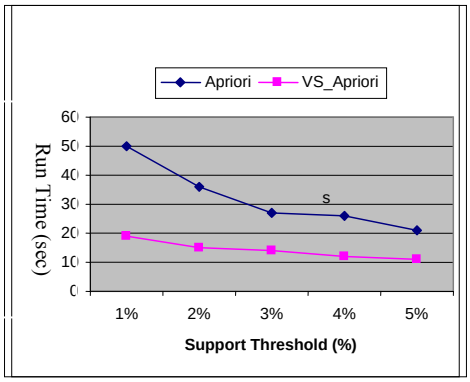

Figure 2. Comparision between Apriori & VS_Apriori (Run Time vs. Support Threshold)

In figure 3 comparisons is between Execution time vs. Number of Transaction with considering threshold 2%.

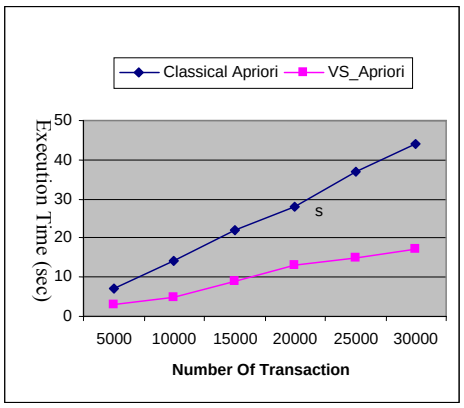

Figure 3. Comparision between Apriori & VS_Apriori (Execution Time vs. Number of Transaction)

This comparison study concludes that VS_Apriori algorithm works much faster then classical Apriori algorithm and also that VS_Apriori algorithm scale well when the support threshold decreases.

V. CONCLUSION

The new VS_Apriori algorithm discussed in this paper improves the efficiency of existing Apriori algorithm for intelligent mining of data itemset. This algorithm offers a reliable technique for accessing frequent data itemset. It helps in managing transaction in controlled manner. It also helps to manage various services, like monitoring, planning and execution of transaction for frequent data itemset mining in intelligent manner.

So with VS_Apriori algorithm, the frequent data itemset mining can be done with lower support threshold then Classical Apriori algorithm without compromising the scalability. In VS_Apriori frequent data itemset mining takes less memory space and thus works much faster. In this way A new algorithmic approach, named, VS_Apriori algorithm provides intelligence in mining of frequent data itemset from Large Scale Data Warehouse

The VS_Apriori algorithm has the potential for use in numerous applications involving Intelligent Data Mining. In future, VS_Apriori algorithm may be incorporated in WAP based applications of transactions for frequent Data Itemset Mining. A Real Time Apriori using mobile communication may be implemented for On Line Real Time Transaction.

The work may be extended to investigate the dynamic strategies for VS_Apriori algorithm. Additional areas of future works related to balancing of transaction for frequent Data Itemset. This technology may further be extended for different purpose in multi-machine environment with features of parallelization. These exists a scope for distributed transaction in VS_Apriori algorithm.

REFERENCES

[1] Amol Ghoting, Gregory Buehrer, Srinivasan Parthasarathy, Daehyun Kim, Anthony Nguyen, Yen-Kuang Chen, and Pradeep Dubey (2005), "Cache conscious frequent pattern mining on a modern processor". In *ACM Transaction*, VLDB-2005, pp 577–588.

[2] Bodon,(2003). "A fast apriori implementation," in Proc. of *The IEEE ICDM Workshop on Frequent Data itemset Mining Implementations (FIMI'03), Melbourne, Florida*, Vol. 90, pp 90-118. November.

[3] Borgelt and R. Kruse,(2002) "Induction of association rules: Apriori implementation," in Proc. of the 15th conf. on *computational statistics*, pp. 395–400.

[4] Borgelt, (2004), "Recursion pruning for the Apriori algorithm," in Proc. *FIMI, ser.* CEUR Workshop., Vol. 126, pp. 120-137.

[5] Buehrer, S. Parthasarathy, and A. Ghoting, (2006) "Out-of-core frequent pattern mining on a commodity pc," in KDD '06: Proc. of the 12th ACM SIGKDD Int'l. conf. on *Knowledge discovery and data mining*. New York, NY, USA, pp. 86–95.

[6] Claudio Lucchese, Salvatore Orlando, Raffaele Perego, and Fabrizio Silvestri. "Webdocs: a real-life huge transactional dataset". In Jr. et al.12

[7] Edward Fredkin. (1960),"Trie memory". Commun. ACM, Vol 3, No. 9, pp. 490–499, 1960.

[8] Grahne and J. Zhu, (2003), "Efficiently using prefix-trees in mining frequent data itemsets," in *FIMI, ser*. CEUR Workshop Proc., Vol. 126.

[9] J. Han, J. Pei, and Y. Yin,(2000), "Mining frequent patterns without candidate generation," in SIGMOD *Conference*, pp. 1–12.

[10] R. Agrawal and R. Srikant.(1994). "Fast Algorithm for Mining Association Rules in Large Databases". In Proc. of the 20th Int'l Conference on *Very Large Databases*, Santiago, Chile, pp. 487–499, August 29-September 1.

[11] R. Agrawal, T. Imielinski, and A. N. Swami, (1993) "Mining association rules between sets of items in large databases," in Proc. of ACM SIGMOD Int'l Conference on *Management of Data*, Washington, pp. 207–216, May 26-28.

[12] Robert D. Blumofe.(1995),"An Efficient Multithreaded Runtime System". In Proc. of the Fifth ACM SIGPLAN Symposium on

Principles and Practice of Parallel Programming, Santa Barbara, California, pp 207-216, July.

[13] Roberto J. Bayardo Jr., Bart Goethals, and Mohammed Javeed Zaki,(2004) editors. FIMI '04, Proc. of the IEEE ICDM Workshop on *Frequent Data itemset Mining Implementations*, Brighton, UK, 2004, Vol. 126, pp. 25-37, November.

[14] S. Brin, R. Motwani, J. D. Ullman, and S. Tsur,(1997), "Dynamic data itemset counting and implication rules for market basket data," SIGMOD Rec., Vol. 26, no. 2, pp. 255–264.