Performance Evaluation of some Online Association Rule Mining Algorithms for sorted and unsorted Data sets

Pramod S.
Reader, Information Technology,
M.P.Christian College of Engineering,
Bhilai,C.G. INDIA.

O.P.Vyas,
Professor, IIIT-Allahabad,
U.P. INDIA.

ABSTRACT
The association rules and its usage put forwarded lots of hopes in the field of data mining. The researchers in the field are going after the association rule mining techniques to find fastest as well as more precise association rules so that it will indirectly increase the profit of the company if we take it as an example. Here in this paper our effort is to do the performance evaluation of some of the existing association rule mining algorithms. Now a day's, online association rule mining is getting its importance due to the popularity of internet as well as the changing behavior of the customer to depend internet for almost everything. The time required for generating frequent itemsets plays an important role. Some algorithms are designed, considering only the time factor. The implementations has been tested as by used the dataset from Freequent Itemset Mining(FIM) Dataset repository. The work yields a detailed analysis with deep understanding of the algorithms to elucidate the performance with standard datasets. The performance evaluation includes aspects like different support value, size of transaction and different datasets.

Keywords- Data Mining, Frequent Itemset, Online Data Mining.

1. INTRODUCTION
The mining for association rules is a form of data mining introduced in [1]. The two basic parameters of Association Rule Mining (ARM) are: support and confidence. *Support*(*s*) of an association rule is defined as the percentage/fraction of records that contain XUY to the total number of records in the database. The count for each item is increased by one every time the item is encountered in different transaction T in database D during the scanning process. It means the support count does not take the quantity of the item into account. For example in a transaction, a customer buys three bottles of beers but we only increase the support count number of {beer*}* by one, in another word if a transaction contains a item then the supportcount of this item is increased by one. Support(*s*) is calculated by the following formula:

Support(*XY*) = (*Support count of XY)\(Total number of transaction in D).*

Confidence of an association rule is defined as the percentage/fraction of the number of transactions that contain X*U*Y to the total number of records that contain X, where if the percentage exceeds the threshold of confidence an interesting association rule X=>Y can be generated. *Confidence*(*X,Y*) = (*Support*(*XY*)) \ (*Support*(*X*)).

2. REVIEW OF ONLINE ASSOCIATION RULE MINING ALGORITHM

2.1 Continuous Association Rule Mining Algorithm
This is a two scan algorithm[2] and during the first scan, the algorithm continuously constructs the lattice of all potentially large itemsets. After each transaction, it inserts and/or removes some itemsets from the lattice. During the second scan the algorithm find out the precise support of each set in the lattice and continuously removes all small itemsets. Merging of two phases(Phase - I with Phase – II) will give the Continuous Association Rule Mining Algorithm(CARMA).

2.1.1.PHASE-I Algorithm
2.1.1.1Support Lattice & Support Sequence
An itemset v in a transaction sequence denoted $support_i$ (v) as the support of v in the first i transactions. The lattice of itemsets are taken as V such that v we have the three associated integers are count(v), firstTrans(v) and maxMissed(v). A support lattice is a superset of all large itemsets in the first i transactions with respect to the support threshold s.

An arbitrary support threshold can be specified by the user for each transaction processed. We get a sequence of support thresholds σ = (σ1, σ2,…) where σi denotes the support threshold for the i-th transaction. This is called as support sequence. For a support sequence σ and an integer i, denoted by $\lceil\sigma\rceil_i$ the least monotone decreasing sequence which is up to i point wise greater or equal to σ and 0 otherwise. The ⌈ σ ⌉i is the ceiling of σ by $avg_i(\sigma)$ and denoting the running average of σ up to i, i.e. $avg_i(\sigma) = \frac{1}{i}\sum_{j=1}^{i}\sigma j$.

The phaseI will constructs the supersets of all large itemsets and maintain a support lattice V while scanning the transaction sequence. At the beginning of the algorithm it initialize V to P{Ø} and set count(Ø)=0, firstTrans(Ø)=0 and maxMissed(Ø)=0. Suppose V is a support lattice up to transaction i -1, read the i-th transaction t_i and to transform V into a support lattice up to i. Let σ_i be the current user-specified support threshold and the lattice can be maintained as given below:

a) Increment:Increment the count(v) for all itemsets v∈V that are contained in t_i maintaining the correctness of all integers stored in V .

b) Insert: Insert a subset v of t_i in V if and only if all subsets w of v are already there in V and are potentially large. i.e. maxSupport(w) > σi. It is based on set of all large itemsets are closed under subsets. This condition also limits the growth of V,

due to the reason that, only minimal supersets of sets in V are added. Thus, the maximal cardinality of all sets in V increases at most by 1 per transaction processed. Inserting v in V, set firstTrans(v) = i and count(v) = 1, since v is contained in the current transaction ti. Since supporti(w) >supporti(v) for all subsets w of v and w⊂ti we get maxMissed(v) < maxMissed(w) + count(w) - 1.

maxMissed(v) defined as
min { max{⌊(i-1)avgi-1(⌈ σ ⌉i-1) ⌋, |v|-1 },
max Missed(ω⊂ v} (1)
In particular we get maxMissed(v) <= i-1,since the empty set is a subset of v, Ø is an element of V and the count of Ø equals i, the current transaction index.
3) Prune: required only small itemsets, i.e. maxSupport(v) < σ_i, are removed from V and if a set is removed then all its supersets are removed as well.

2.1.2PHASE - II Algorithm

Let V be the computed support lattice by by PhaseI. PhaseII uses the last user-specified support threshold σ_n as pruning threshold. At first, PhaseII removes all trivially small itemsets. That is itemsets with maxSupport < σ_n, from V.

PhaseII increments count and decrements maxMissed as by scanning the transaction sequence for each itemset contained in the current transaction, up to the transaction at which the itemset was inserted. Setting maxMissed(v) = 0 for an itemset v may lead maxSupport(w) > maxSupport(v) for some superset w of v. Thus we set maxMissed(w) = count(v)-count(w) for all supersets w of v with maxSupport(w) > maxSupport(v). PhaseII stops as soon as the current transaction index is past firstTrans for all itemsets in the lattice. The resulting lattice contains all large itemsets along with the precise support for each itemset.

2.1.2.1Forward Pruning

We extend the preliminary PhaseII algorithm described above by a forward pruning" technique, which allows us to remove some small singleton set v and all its descendants from V, before we reach firstTrans(v) even if maxSupport(v)>σn. The general idea is the following: The insertion of v in V is guaranteed to take place, since its only subset is the emptyset which always has support 1. By an induction on ft-i we get that if

⌈n.σ⌉ - count(v) + ⌊i.avg ı(⌈σ⌉i) ⌋ >⌊(ft-1)avg ft-1 (⌈σ⌉ft-1) ⌋

(2)

The algorithm is as shown below

Function CARMA(transaction sequence T, support sequence σ):
support lattice;
Support lattice V;
Begin

V:= PhaseI(T, σ); V:= PhaseII(V,T, σ);
Return V;
End;
Function PhaseI(transaction sequence(t1,….tn), support sequence σ=(σ1,…. σn):
support lattice; support lattice V;
Begin
V:={Ø}, maxMissed(v):=0, firstTrans(v):=0,count(v)=0.
For i from 1 to n do
Increment: for all vϵ V with v ⊆ ti do count(v)++;
Insert: for all v⊆ti with v∉V do
If ∀ω c v: ω ϵ V and maxSupport(ω)>= σi then V:= V ∪ {v};
maxMissed(v):=min{max{⌊(i-1)avg_{i-1}(⌈σ⌉i-1];|v|-1},
maxMissed(ω)+count(ω)|ωc v};
firstTrans(v):=i; count(v):=1;fi,od;
Prune: remove vϵ V from V only if maxSupport(v)< σi.
If vϵ V is removed, remove all supersets as well; od;return;
V;end;
Function PhaseII(support Lattice V, transaction sequence(t1…….tn), support sequence σ):
support lattice; Integer ft, i=0;
Begin
Initial prune V:= V\{v ϵ V| maxSupport(v) < σ_n
Rescan: while ∃v ϵ V: i< firstTrans(v) doi++;
for all v ϵ V do ft := firstTrans(v);
Adjust: if v ⊆ t_i and ft<I then
count(v)++, maxMissed(v)--;fi;
if ft=i then maxMissed(v):=0;
for all ω ϵ V: v⊂ω and
maxSupport(ω)> maxSupport(v) do
maxSupport(ω) := count(v)-count(ω);od; fi;
prune: if maxSupport(v)< σ_n then V:=V\{v}; fi;
Forward Prune: if |v|=1 and v does not occur in t1…….ti and⌈ n. σ_n⌉-count(v)+ ⌊i.avg_i (⌈σ⌉i⌋ > ⌊(ft-1)avg_{ft} (⌈σ⌉ft-1)⌋ then V:= V\{ωϵ V| v ⊆ ω };fi; return v;

2.2.The Data Stream Combinatorial Approximation Algorithm

2.2.1.DSCA Algorithm

Basically DSCA[3] has two different phases during its execution, one is the transaction processing phase and the other is the count approximation phase. As a rule, it belongs in the phaseI. The latter phase is entered only when invoked by the user, and as the approximation is completed and the request is answered, it returns to the former phase immediately. The DSCA algorithm itself maintains a lexicographic order prefix tree, which will be null at the beginning. The main function of this tree is to keep the count information about I_1 , I_2 , and (part of) I_3 of the data stream for the approximation in the later step, where In denotes the n-itemsets together with their counts. Different from Lossy Counting [4], FDPM[5], and most of the existing data-stream mining algorithms, the DSCA algorithm will not incrementally maintain the potentially large itemsets of the entire data stream.

If the stream data is keep on coming, DSCA just returns to the transaction processing phase and goes on. Unlike other stream mining algorithms, DSCA does not have the ‘‘concept drift” problem existing in the data- stream mining domain. Since DSCA just records the count information about I_1 I_2 , and (part of) I_3 for each incoming transaction, it calculates the approximate counts of itemsets (of the entire data stream) after invoked. We have to identify whether an itemset is frequent or not by checking if its approximate count is above the minimum support threshold. We can approximate[3] the value of m-union term using the Eq. (1) as given below

$$|A_1 \cup A_2 \cup \ldots \cup A_m| = \sum_{|s| \le k|} \alpha_{|s|}^{k,m} \, |\bigcap_{i \epsilon s} Ai| \qquad (1)$$

DSCA Algorithm is as given below:

Input: A transactional data stream S, the minimum support

```
Value(ms)
Output: A list of frequent itemsets(F)
Data structure: A lexicographic order prefix tree L;
Method:
Build an empty lexicographic order prefix tree L;
While transactional data is still streaming in do begin
Clear the contents in F(set F to be empty);
While there is no request from the user do begin
Read the next incoming transaction T from S;
Find all 1-subsets and 2-subsets contained in T and
        record(increase) their counts in the corresponding
        nodes in L;
        if the length of T>2 then begin
        find all 3-subsets contained in T with the Skip-
        andComplete technique, and record their counts in the
        corresponding groups
        end if; end while
Find all large 1-items and 2-items in L and insert them into F
      Calculate the counts of all 3-itemsets by Eq(1) with
    m=3 and k=2, together with the counts bounding
        technique;
        Correct the approximate supports of 3-itemsets with
        the group counts;
        Insert every large 3-itemsets(n>3) by Eq(1)
  With k=3 in both the depth-first manner and the
        lexicographic order;
        Insert every large n-itemset into F;
        Output the frequent itemsets list F; end while.
```

2.3.estDec METHOD

All the itemset that appear in a data stream are not significant for finding frequent itemsets. The itemset that has very less support than a predefined minimum support is not very much required to monitor since it cannot be a frequent itemset in the near future. Therefore it can be delayed until be a frequent itemset in the near future. When the estimated support of a new itemset become large enough, it can called significant itemset and it can be inserted in to the lattice. Here in the estDec method that maintains a triple(cnt, err, MRtid) in every node for its corresponding itemset e. The count of the itemset e is denoted by cnt. The maximum error count of the itemset e is denoted by err. Finally, the transaction identifier of the most recent transaction that contains the itemset e is denoted by MRtid.

The estDec[6] method is composed of four phases: parameter updating phase (Phase I), count updating phase (Phase II), delayed- insertion phase (Phase III) and frequent itemset selection phase (Phase IV). When a new transaction Tk is generated in a data stream, the total number of transactions in the current data stream $|D|_k$ is updated in the parameter updating phase.

$$|D|_k = |D|_{k-1} \times d+1$$

In the count updating phase the counts of those itemsets in a monitoring lattice that appear in the new transactions are updated. All the paths of a monitoring lattice that are induced by the items of the transaction are traversed and the previous triple (cnt_{pre}, err_{pre}, $MRtid_{pre}$) of each node in the paths is updated to the current triple (cnt_k, err_k, $MRtid_k$) as follows:

$$cnt_k = cnt_{pre} \times d^{(k-MRtidpre)}+1,$$

$$err_k=err_{pre} \times d^{(k-MRtidpre)}, MRtid_k=k$$

When the updated support, that is $cnt_k / |D|_k$ of an itemset in a monitoring lattice becomes less than a predefined threshold, the itemset is regarded as an insignificant itemset, so that it is pruned from the monitoring lattice as in conventional lattice-based data mining methods [7,8]. Anyhow, if a 1-itemset is pruned from a monitoring lattice, it is impossible to estimate its count later. Therefore, it does not required pruning. The threshold of this operation is defined as a threshold for pruning S_{prn} which should be less than a minimum support S_{min}. After all of these itemsets are updated, the delayed-insertion phase is started in order to find any new itemset that has a high possibility to become a frequent itemset in the near future.

A new itemset is inserted to a monitoring lattice only in the following two cases. At first the new 1-itemset appears in a newly generated transaction. In this case, the itemset is instantly inserted to a monitoring lattice without any estimation process. Consequently, the count cnt of every 1-itemset in a monitoring lattice is not an estimated value but an actual value. The second case is when the estimated support of an n-itemset ($n \geq 2$) that is not in the monitoring lattice is large enough to be monitored. In this phase, among the items of the new transaction, the items whose supports are less than S_{ins} are not considered. While navigating the lattice according to the remaining items of the new transaction, the count of an insignificant itemset that is composed of a significant itemset and one of the remaining items is estimated by its maximum count $C_{max}(e)$. Due to the characteristics of a prefix lattice structure, there is no candidate itemset generation process. There is no candidate itemset generation process because such an itemset is identified systematically while navigating the lattice according to the remaining items in the new transaction.

If any of its (|e|-1)-subsets in $P_{n-1}(e)$ is not currently maintained in the monitoring lattice, the count of the itemset e is not estimated. This is because its $C_{max}(e)$ is always 0 in this case. Subsequently, the estimated support of the itemset can be found by the ratio of its count cnt over the current total number of transactions $|D|_k$. If it is greater than or equal to a predefined threshold, the itemset is inserted to the monitoring lattice. This mechanism is called as a delayed-insertion operation and the pre-defined threshold for this insertion is defined as a threshold for delayed-insertion S_{ins} which should be also less than a minimum support S_{min}. When an itemset *e* is inserted, all of its (*|e|-1*)-subsets should be significant. Due to this reason, it is possible to find the upper bound *Cupper(e)* of its actual count when it is inserted at the k^{th} transaction. In other words, among the *k* transactions generated so far, at least *|e|*-1 transactions that contain the itemset *e* are required to insert all of its subsets to the monitoring lattice in advance.

Therefore, its actual count is maximized when these *|e|-1* transactions are most recently generated. The similar approach is used in [2]. The decayed count of the itemset *e* for the insertion of its subsets by these recent *|e|-1* transactions is represented by a term cnt_for_subsets as follows:

$$cntt\_for\_subsets = \{1-d^{(|e|-1)}\}/(1-d) \quad (4)$$

The maximum possible decayed count of the itemset *e* before the recent *|e|*-1 transactions is denoted by *max_cnt_before_subsets* and it is represented as follows:

$$max\_cnt\_before\_subsets=S_{ins} \times \{|D|_{k-(|e|)-1)}\} \times d^{(|e|-1)} \quad (5)$$

Consequently, $C^{upper}(e)$ can be found as follows:

$$C^{upper}(e) = max\_cnt\_before\_subsets+cnt\_for\_subsets \quad (6)$$

If $C_{max}(e)$ in Eq.(2) is greater than the upper bound Cupper(e), $C^{upper}(e)$ is used as its count cnt. Accordingly, the current triple (cnt_k, err_k, $MRtid_k$) of the itemset e in the corresponding node of the monitoring lattice is updated as follows:

$cnt_k = \min\{C^{max}(e), C^{upper}(e)\}$ (7)

$err_k = E(e) = cnt_k - C^{min}(e)$, $MRtid_k = k$

An itemset pruned at present can be inserted into the monitoring lattice in the future by the delayed-insertion operation if it appears frequently in new transactions.

Consequently, S_{prn} should be less than S_{ins}. As the gap between the two thresholds S_{prn} and S_{ins} is enlarged, the possibility of repeating the insertion and pruning of the same itemset are getting reduced frequently. Furthermore, as the gap between these two thresholds is enlarged, the accuracy of frequent itemsets is improved while the size of a monitoring lattice is increased.

The frequent itemset selection phase is performed only when the mining result of the current data set is required. It produces all current frequent itemsets in a monitoring lattice by the same way as in conventional mining methods [7,8] based on a prefix-tree lattice structure. When this phase is performed in the current data stream Dk, an itemset e is frequent if its current support $\{cnt \times d^{(k-MRtid)}\}/|D|_k$ is greater than a predefined minimum support S_{min}. Furthermore, its current support error $\{err \times d^{(k-MRtid)}\}/|D|_k$ can be found as well.

All the insignificant itemsets in a monitoring lattice can be pruned together by examining the current support of every itemset in the monitoring lattice. This mechanism is called as a force-pruning operation and can be performed periodically or when the current size of a monitoring lattice reaches a pre-defined threshold value.

estDec Algorithm:

Input: A data stream D
Output: A complete set of recent frequent itemsets L_k
d: A given decay rate
ML: A monitoring lattice
ML= ϕ;
For each new transaction in D{
Read current transaction T_k;
//Parameter updating phase
$|D|_k = |D|_{k-1} * d + 1$;
//Count updating phase
For all itemset e s.t. $e \in (2^{Tk} - \{\phi\})$ and $e \in$ ML{
$cnt = cnt * d^{(k-Mrtid)} + 1$; $err = err * d^{(k-Mrtid)}$; $Mrtid = k$;
if $(cnt/|D|_k) < S_{pm}$ and $|e| > 1$ //Pruning
Eliminate e and it's child node from ML;}
//Delayed-insertion phase
$T_k^{!} = ItemFiltering(T_k)$;
For all itemset e s.t. $e \in (2^{Tk!} - \{\phi\})$ and $e \in$ ML{
If $|e| = 1$ {
Insert e into ML;cnt=1;err=0;Mrtid=k
}else{Estimate $C^{max}(e)$ and $C^{min}(e)$;
If $C^{max}(e) > C^{upper}(e)$;
$C^{max}(e) = C^{upper}(e)$
If $(C^{max}(e)/|D|k) >= S_{ins}$ {
Insert e into ML;cnt=Cmax(e);err=Cmax(e)-Cmin(e);Mrtid=k;}}}
//Frequent itemset selection phase
$L_k = \phi$;
For all itemset e∈ML{
$cnt = cnt * d^{(k-Mrtid)}$; $err = err * d^{(k-Mrtid)}$; $Mrtid = k$;
If $(cnt/|D|_k) >= S_{min}$
$L_k = L_k \cup \{e\}$;}}

3. IMPLEMENTATION AND COMPARISON

A data stream is a massive unbounded sequence of data elements continuously generated at a rapid rate. Due to this reason, it is impossible to maintain all elements of a data stream, as a result, data stream processing should satisfy the following requirements [1]. First, each data element should be examined at most once to analyze a data stream. Second, memory usage for data stream analysis should be restricted finitely although new data elements are continuously generated in a data stream. Third, newly generated data elements should be processed as fast as possible. Finally, the up-to-date analysis result of a data stream should be instantly available when requested. In order to satisfy these requirements, data stream processing becomes more tedious task.

The three algorithms mentioned in II are implemented in JAVA and the results are plotted. The performance evaluation study showing that the estDec outperforms the other two algorithms in memory usage as well as the total performance of the algorithm in both the cases of sorted and unsorted transaction set. We have tested all the three algorithms with five Data sets and all of them are available in Freequent Itemset Mining Dataset Repository[9]. Transactions of each data set are looked up one by one in sequence to simulate the environment of an online data stream.

3.1 Sorted /Unsorted transaction set

We have found from our studies that the way in which we inputting the transaction will have a role to reduce/increase the memory space as well as the searching performance in the lattice. The transaction sets in online transactions are not in the order based on the item's name or anything. It is always based on the order it entered. We have found that if we make a little modification before it giving to the lattice as by sorting based on the item name it will be easier to keep in the lattice without much redundancy in the items available in the lattice.

In the above described algorithms DSCA is using the sorted transaction items for updating the lattice. Therefore we have not been used the DSCA algorithm unsorted items performance test. The unsorted transaction items are inputed to the other two algorithm implementations and the results are plotted.

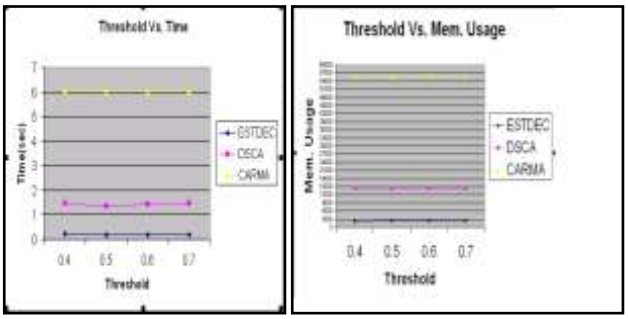

Fig 1: Threshold Vs Time/Memory for T10I4D100K Dataset (Sorted)

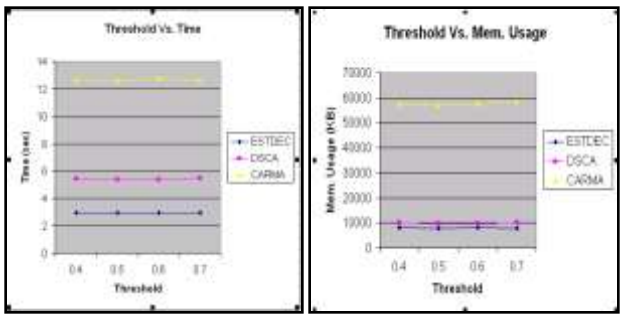

Fig 2:Threshold Vs Time/Memory for T40I10D100K Dataset (Sorted)

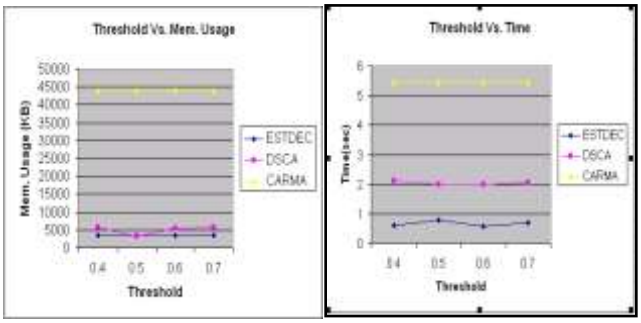

Fig 3: Threshold Vs Memory/Time usage for retail_set Dataset (Sorted)

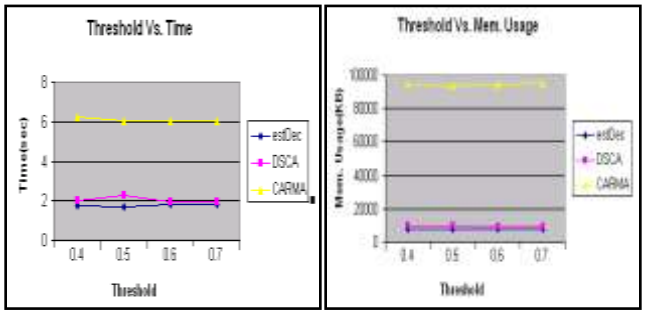

Fig4: Threshold Vs Time/Mem.Usage for accident Dataset (Sorted)

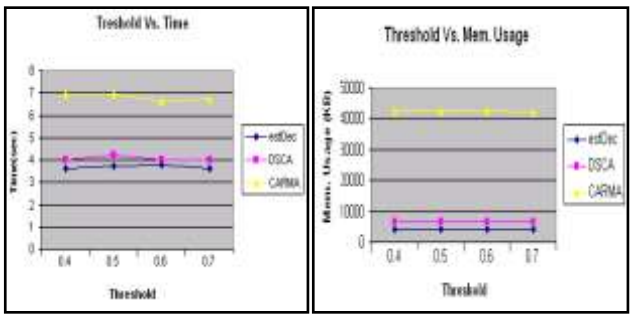

Fig5: Threshold Vs Time/Mem.Usage for KOSARAK Dataset (Sorted)

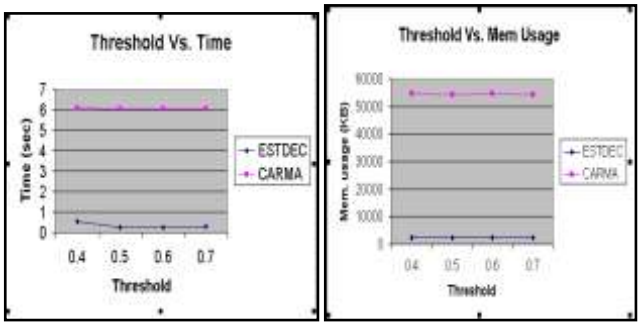

Fig6: Threshold Vs Time/Mem.Usage for T10I4D100K(Unsorted)

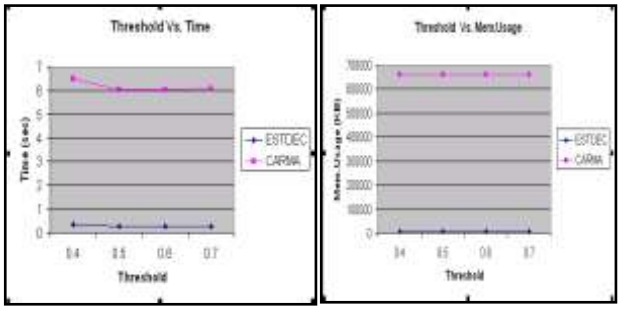

Fig7: Threshold Vs Time/Mem.Usage for T40I10D100K (Unsorted)

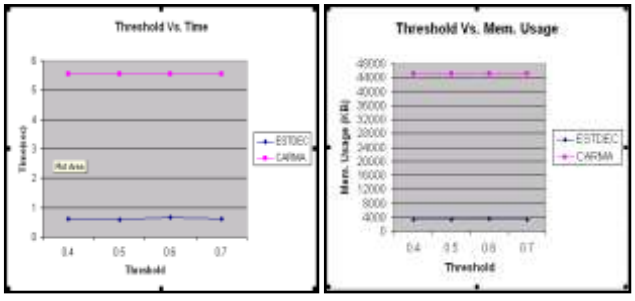

Fig8: Threshold Vs Time/Mem.Usage for retail_set(Unsorted)

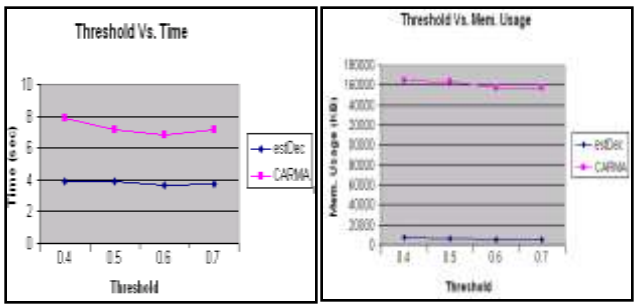

Fig9: Threshold Vs Time/Mem.Usage for KOSARAK(Unsorted)

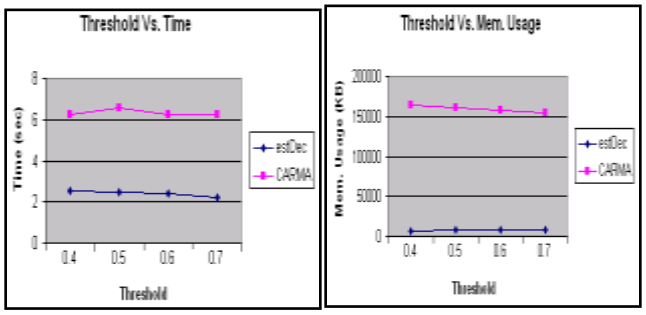

Fig10: Threshold Vs Time/Mem.Usage for accident(Unsorted)

4. CONCLUSION

In this paper, we have gone through the important three online algorithms namely CARMA, DSCA and estDec. The approach of this three online Data Mining algorithms are different and we have implemented all the three algorithms in java. In our study we have found that the estDec algorithm is outperforming the other two algorithms. There are lot more studies required to know about how it perform in different real time situations. We have made a situation of online data stream and one of the problem that noticed in the estDec as well as the other two algorithms, handling of two task simultaneously (for the updation of lattice as well as the query answering) will reduce the performance. To improve the performance priority has to set.

5. REFERENCES

[1] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In Proc. of the ACM SIGMOD Conference on Management of Data, pages 207-216,Washington, D.C., May 1993.

[2] C. Hidber. Online association rule mining. In *Proc. of the* ACM SIGMOD Int'l Conference on Management of Data, pages 145-156, Philadelphia, PA, May 1999.

[3] J.F.Jea and C.W. Li , Discovering Frequent Itemsets over Transactional Data Streams through an efficient and stable approximate approach. Elsevier Journel 2009.

[4] Manku, G. S., & Motwani, R. (2002). Approximate frequency counts over data streams. In Proceedings of the 28th international conference on VLDB (pp. 346–357).

[5] Yu, J. X., Chong, Z., Lu, H., Zhang, Z., & Zhou, A. (2006). A false negative approach to mining frequent itemsets from high speed transactional data streams. Information Sciences, 176, 1986–2015.

[6] J.H. Chang, W.S. Lee, Finding recent frequent itemsets adaptively over online data streams, in: Proceeding of the 9th ACM SIGKDD, 2003, pp. 487–492.

[7] Agrawal, R., and Shafer, J. 1997. Parallel mining of association rules. IEEE Transactions on Knowledge and Data Engineering 8(6). Record, pages 255-264, New York, May 13th-15th 1997. ACM Press.

[8] J. H. Chang and W. S. Lee. Finding Recent Frequent Itemsets Adaptively over Online Data Streams. In Proc. of KDD, 2003.

[9] B. Goethals and M. Zaki. FIMI '03, Frequent Itemset Mining Implementations. In Proc.of the ICDM 2003 Workshop on Frequent Itemset Mining Implementations, 2003.