SCI-Tree: An Incremental Algorithm for Computing Support Counts of all Closed Intervals from an Interval Dataset

Dwipen Laskar, Naba Jyoti Sarmah, Anjana Kakoti Mahanta

Abstract*: Interval data mining is used to extract unknown patterns, hidden rules, associations etc. associated in interval based data. The extraction of closed interval is important because by mining the set of closed intervals and their support counts, the support counts of any interval can be computed easily. In this work an incremental algorithm for computing closed intervals together with their support counts from interval dataset is proposed. Many methods for mining closed intervals are available. Most of these methods assume a static data set as input and hence the algorithms are non-incremental. Real life data sets are however dynamic by nature. An efficient incremental algorithm called CI-Tree has been already proposed for computing closed intervals present in dynamic interval data. However this method could not compute the support values of the closed intervals. The proposed algorithm called SCI-Tree extracts all closed intervals together with their support values incrementally from the given interval data. Also, all the frequent closed intervals can be computed for any user defined minimum support with a single scan of SCI-Tree without revisiting the dataset. The proposed method has been tested with real life and synthetic datasets and results have been reported.

***Index Terms*: Data mining, Interval data, Closed Interval, Support Count, Minimum Support.**

I. INTRODUCTION

Interval data refers to those data where intervals of distance, time etc. which are associated with events. In many real world data we often encounter intervals. Mining from interval data is comparatively a new field of research in data mining. It is used to extract unknown patterns, rules and relationships hidden in dynamically increasing large data that has intervals associated with them. The concept of closed interval is important because by using the set of closed intervals and their support counts, the support of any given interval can be computed. In this work, we consider a dataset of interval transactions. Each interval transaction has a start and end point. For example, in a temperature recoding system that records the minimum and maximum temperature per day, if we extract closed intervals of temperatures then it is possible to find out the dominant temperature intervals in a particular period. This can help the system to predict temperature. Similarly, in an online e-learning portal by using the recorded user login and logout times, the administrator can extract the closed intervals to design strategies to involve maximum number of users taking part in a discussion.

In this paper we propose an incremental algorithm for finding all closed intervals and their support counts from interval datasets. The proposed algorithm constructs a tree of Closed Interval with Support counts. The tree is named as SCI-Tree that keeps the information of all the closed intervals along with their support counts present in the dataset. Whenever a new interval comes in, the algorithm updates the SCI-Tree to generate the new closed intervals along with support counts without visiting the dataset. All the frequent closed intervals can also be computed for any user given minimum support with a single scan of the SCI-Tree. No separate scan of the database is necessary to do this. In [1] authors have proposed an efficient incremental algorithm for computing all closed intervals. They have used a data-structure called CI-Tree (Closed Interval Tree) for storing the closed intervals. When new intervals come in, CI-Tree is updated to generate the new closed intervals. However this method could not calculate the support values of the closed intervals. It is necessary to visit the interval database separately again to compute the supports of closed intervals.

In Section II, some of the earlier works related to the problem at hand are discussed. In Section III, some basic definitions related to closed interval mining problem are given. In Section IV, an incremental approach of closed interval mining is discussed in details. Section V provides details about SCI-Tree. In section VI, construction of SCI-Tree is explained. Proposed algorithm is discussed in section VII .In Section VIII explains Performance analysis of the proposed algorithm. The experimental results are given in section IX. We conclude and outline the scope of future work in section X.

II. RELATED WORKS

Very limited research works have been found in interval data. J. F. Allen [2] had shown that between two temporal intervals can be related in thirteen possible ways. He had put forward a method for mining knowledge from time-related intervals using these 13 relationships.

Kam and Fu [3] pointed out some limitations in the Allen's method [2] by showing that ambiguity arises with composite patterns depending on the ways of combining the relationships and proposed a

Revised Manuscript Received on August 05, 2019.

Dwipen Laskar, Assistant Professor, Department of Computer Science, Gauhati University, Gauhati, India.
Naba Jyoti Sarmah, Assistant Professor, Department of B.Voc.(IT), Nalbari Commerce College, Nalbari, Assam.
Anjana Kakoti Mahanta, Professor, Department of Computer Science, Gauhati University, Gauhati, India.

new method removal of these ambiguities.

In S. Wu and Y. Chen [4], proposed a algorithm called as *TPrefixSpan* to mine non-ambiguous temporal patterns from interval-based event data. Villafane et al. [5][6] proposed containment relationship in time series data by treating them as interval events. They also put forward an algorithm to discover patterns using these relationships.

Patel et al. [7] put forward a lossless hierarchical representation of temporal interval-based events and extends proposed an algorithm called *IEMiner* for extracting frequent time-interval based patterns from interval data. Based on the discovered time-interval patterns, a classifier called *IEClassifier* was build to classify closely related classes.

Chen et al. [8] put forwarded an end-point representation of event intervals and event sequences to simplify the complex relations between events of interval. Any event *E* is represented by using the endpoint sequence E^+E^-, where E^+ and E^- represents the *start-time* and *finish-time* of the event *E*. A parenthesis is used to represent the endpoints occurred at the same time. E.g. the endpoint sequence, $(X^+Y^+)(Y^-)(X^-)$ represents that event *X* and *Y* starts at the same time but *Y* finishes before *X*. For handling multiple occurrences of events, an occurrence number is used that is attached with the endpoints. Based on these representations he proposed an algorithm called *CEMiner* for mining closed time interval patterns from large interval dataset.

J. Lin [9] first introduced the concept of maximal frequent intervals suitable for intervals in discrete as well as continuous domain. A data structure called I-Tree is used to store the frequencies of the intervals in an interval database. He proposed a Pre-order traversal algorithm to extract the maximal frequent intervals by scanning the I-Tree once. Pasquier et al. [10] was the first to introduce the notion of closed frequent set. They defined the framework of closed itemset lattices and proposed an algorithm called A-Close to discover frequent closed itemsets.

M. Dutta and A. K. Mahanta [11] extended the idea of closed frequent itemsets to interval dataset and proposed a method to compute closed frequent intervals by using the set of maximal frequent intervals present in that interval dataset.

M. Dutta [12] proposed an efficient method of construction of *I-Tree* as designed by Lin [9] and developed an algorithm called *MIntMiner* for mining maximal frequent intervals. A data structure called *IS- Tree* is also proposed to discover closed frequent intervals.

N. Sarmah et al. [1] proposed an incremental approach by defining a new data structure called *CI-Tree* to mine all closed intervals in the interval dataset. When never a new interval comes in to the interval dataset, the *CI-Tree* is updated without scanning the dataset. The major disadvantage in *CI-Tree* is that it cannot store the support count of discovered closed intervals. In our work, we modify the *CI-Tree* to construct a new data structure called *SCI-Tree* (Support Closed Interval Tree) which can store all the closed intervals and their respective support values. It also stores the information to differentiate between the generated closed intervals and input closed intervals. a generated closed interval is a closed interval but not an interval in the input interval dataset. Remaining all closed intervals are input closed intervals.

III. PROBLEM DEFINITION

$TDB = \{r_1, r_2, ..., r_n\}$ is a database consisting of a set of records r_i , where each record r_i is a triplet, $[l, f, r]$. Here, $[l, r]$ is an interval with l and r as the left and right endpoint of it. The symbol f denotes the frequency of the interval. The objective of present work is to discover all closed intervals and their support counts in the interval dataset. It is assumed that the interval domain is discrete and if $[l, r]$, an interval in the interval dataset and *p* is a point in that interval then $l \le p \le r$.

Let $L = \{l_1, l_2, ..., l_{|L|}\}$ denotes the distinct left endpoints and sorted in the ascending order, i.e. $l_1 < l_2 < ..... < l_{|L|}$. Similarly, let $R = \{r_1, r_2, ..., r_{|R|}\}$ denotes the distinct right endpoints and sorted in the ascending order. i.e. $r_1 < r_2 < ..... < r_{|R|}$.

If $I_1 = [l_1, r_1]$ and $I_2 = [l_2, r_2]$ are any two intervals and if $l_2 \le l_1 \le r_1 \le r_2$ then I_1 is said to be contained in interval I_2 and it is represented as $I_1 \subseteq I_2$. If *s* is the total number of intervals that contains I_1 in the interval dataset, then it is called as the support counts of I_1 . An interval I_1 is said to be properly contained in another interval I_2 if and only if $l_2 < l_1 \le r_1 \le r_2$ or $l_2 \le l_1 \le r_1 < r_2$ or $l_2 < l_1 \le r_1 < r_2$, and is denoted as $I_1 \subset I_2$.

Any interval I_1 is called as a closed if the support of any interval properly containing I_1 is less than the support of I_2 i.e. if $I_1 \subset I_2$ and if s_1 and s_2 are support of I_1 and I_2 respectively then $s_2 < s_1$.

A closed interval I is termed as an input closed interval and if $I \in TDB$, otherwise it is denoted as a generated closed interval.

Properties of closed intervals are based on following theorems [1].

Theorem-1: if I_1 and I_2 are any two intervals in the dataset and if $I_1 \subseteq I_2$ with s_1 and s_2 are support of I_1 and I_2 respectively then $s_2 < s_1$.

Theorem-2: for any interval $[l, r]$ in the dataset, all the closed intervals are denoted as $[l, r']$, where $l \le r' \le r$ and $part(r') \le l$.

Theorem-3: If $[l, r]$ is a closed interval then *r* is a right endpoint in the dataset and $part(r) \le l$.

IV. INCREMENTAL METHOD FOR MINING CLOSED INTERVALS

Algorithm proposed by Sarmah et al. [1] extracts all closed intervals in an interval dataset by using the data structure *CI-Tree*. The CI-Tree consists of a header list *H*. Each element in that list is a node *I* which is linked with a distinct left endpoint $l_{|L|} \in L$ in the dataset. Each such node *I*

also contains a sub-list S_I of distinct right endpoints $r_{|R|} \in R$. The elements in H are sorted in ascending order of the left endpoints $l_{|L|} \in L$ and those in S_I are sorted in ascending order of right endpoints $r_{|R|} \in R$. An interval $I = [l, r]$ is a closed interval iff a node I for the left endpoint, l occurs in H and a node for the right endpoint r occurs in S_I. Since our proposed algorithm in based on the work done in [1], for the sake of completeness a short description of the *CI-Tree* together with its construction is elaborated below with diagrams.

The building of CI-Tree has two steps. In the first step, the algorithm uses *Header_ Modify* function to modify the header list and in the second step, it uses *SubList_Modify* function to modify the sub lists. In the updation of header list, H if the left endpoint of the newly inserted interval is present in H then it modifies the right value of the respective header node in case it is less than the right value of newly inserted interval. Otherwise, the new interval is inserted in H maintaining the sorted order. The modification of closed intervals in CI-Tree is based on two conditions [1].

Condition-1: If $I = [I.left, I.right]$ is the newly inserted header node and $H = [H.left, H.right]$ is an existing header node in the CI-Tree then if $H.left \le I.left \le I.right \le H.right$ or $H.left < I.left \le I.right < H.right$ then it adds all those endpoints ep_H in S_H to S_I such that $I.left \le ep_H \le I.right$, where S_H and S_I refers to the sub lists of header node H and I respectively. [*Theorem 2*]

Condition-2: If $I.left < H.left \le I.right < H.right$ then $I.right$ is added to the sub list S_H [*Theorem 3*].

Example

Let *TDB* is transactions of intervals as shown below in *Table I* and construction of CI-Tree is elaborated below with diagrams.

Table I. *Interval Dataset, TDB*

TID	1	2	3	4	5	6	7
I	[6,9]	[1,5]	[2,7]	[1,7]	[2,4]	[8,10]	[7,11]

I=Interval

Now, Initial Header-List, $H = NULL$. The input Interval, $I = [6,9]$, is added to the H as header node $I_H = [6,9]$ (using *Header_Modify*). Now, 9 is added to the sub-list S_H of the newly created header node I_H (*condition 2*).

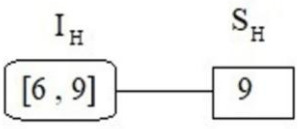

Fig. 1. CI-Tree after the insertion of interval [6, 9]

For the Interval, $I = [1,5]$, H does not contain an entry with $H.left = 1$. So, node I is added to H as $I_H = [1,5]$ maintaining the sorted order. 5 is added to the sub-list S_H of I_H (*condition 2*).

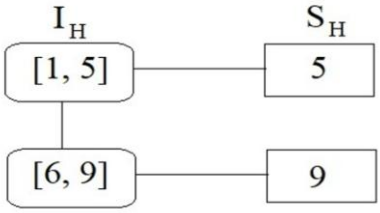

Fig. 2. CI-Tree after the insertion of interval [1, 5]

For the input Interval, $I = [2,7]$, H does not contain an entry with $H.left = 2$. So, node I is added to H as $I_H = [2,7]$ maintaining the sorted order. 7 is added to the sub-list of I_H using *condition-1*. *5* is added to the sub-list of I_H and *7* is added to the sub-list of header node $[6,9]$ (*condition-2*).

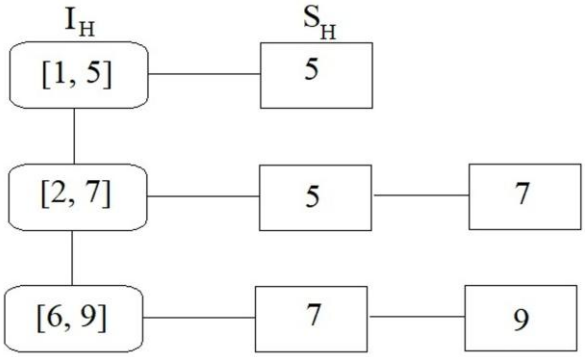

Fig. 3. CI-Tree after the insertion of interval [2, 7]

For the Interval, $I = [1,7]$, H contains a header node $I_H = [1,7]$ with $I_H.left = 1$ and $I_H.right < I.right$ i.e. $5 < 7$. So, $I_H.right$ of the node I_H is updated to 7, i.e. $I_H.right = 7$. Also, 7 is added to the sub-list I_H. Since, $I.right$ is present in the sub-lists of all the following header nodes, so no modification is required in the tree.

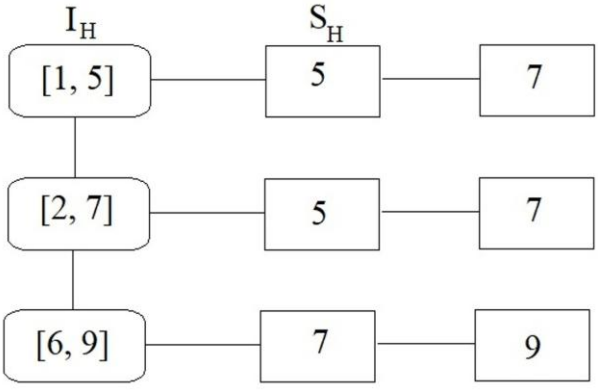

Fig. 4. CI-Tree after the insertion of interval [1, 7]

Following the same process for all the remaining input intervals in *TDB*, $I = \{[2,4], [8,10], [7,11]\}$ as discussed above, the final CI-Tree constructed will be as given below:

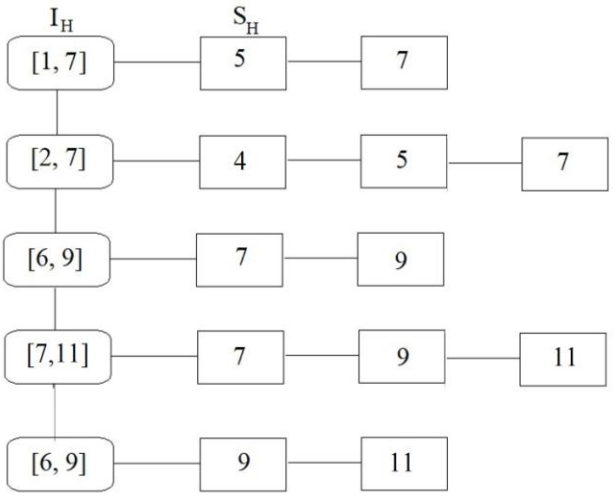

Fig. 5. Final CI-Tree for Dataset in Table I

So, the final CI-Tree, indicated that {[1,7], [1,7], [2,4], [2,5], [2,7], [6,7], [6,9], [7,7], [7,9], [7,11], [6,9], [6,11]} are the closed intervals extracted from the input interval dataset *TDB*.

V. SCI-TREE FOR STORING CLOSED INTERVALS WITH SUPPORT COUNTS

The CI-Tree as described above stores all the closed intervals generated from all input intervals. As input intervals are closed too, the input intervals are stored in the tree as well. There is no distinction kept to identify the input intervals from the generated closed intervals. In this new approach called *SCI-Tree*, the concept of CI-Tree is extended to mine the support counts of all closed interval incrementally by keeping a distinction between generated closed interval and input closed interval.

If $I=[l,r]$, is an input interval, then it generates closed intervals $[l,r']$, where $l \le r' \le r$ and $part(r') \le l$. It also generates closed intervals $[l',r]$,where $l < l' \le r$ and $r \le part(l')$ (*Theorem 2*). The support counts of all closed intervals with right endpoint as r is computed differently from other closed intervals as stated in section-VI. Thus, in SCI-Tree, there is need to differentiate between generated closed intervals and input closed intervals as the support values of an interval is a count of the number of input intervals containing that interval. A *flag* is maintained in an interval to do so.

The proposed SCI-Tree contains two types of lists of nodes-a Header list, H and Sub lists, S_I for each node I in the header list as in CI-Tree. A node I in H contains four fields $(l, r_{mx}, h\_next, s\_next)$, where $l \in L$ and if $\{r_1, r_2, \ldots\ldots, r_{|R|}\} \subseteq R$ is set of distinct right endpoints associated with l in the dataset then r_{mx} is the largest of $\{r_1, r_2, \ldots\ldots, r_{|R|}\}$. For each distinct left endpoint, $l_i \in L$, $i = \{1,2,\ldots\ldots,|L|\}$ in the dataset, there is a corresponding node, l_{l_i} in the header list H with $l_{l_i} = l_i$ and nodes in the header list are sorted in ascending order of their left endpoints l. The h_next and s_next are two pointers, where h_next points the node next of I in the header list and s_next points to the node in the sub list, S_I associated to the header node I.

Each node ep_I in sub-list, S_I of a header node, I contains five field $(l, \text{sup}, freq, flag, s\_next)$, where r is a right endpoint in the dataset, *sup* is the support count of the interval $[l,r]$ with l as left endpoint in I and r as right endpoint in ep_I, *freq* is frequency count of the same interval $[l,r]$ in the input interval dataset, *flag* contains a value, $flag \in \{0,1\}$ to distinguish between generated closed interval and input closed interval and s_next points to the node next to ep_I in the sub list S_I. If a node ep_I in sub-list S_I with right value equal to r is associated with the header node $I=[l, r_{\max}]$ then $[l,r]$ is a closed interval iff $I.r_{\max} \ge ep_I.r$. The general structure of SCI-Tree is as given in Fig- 6.

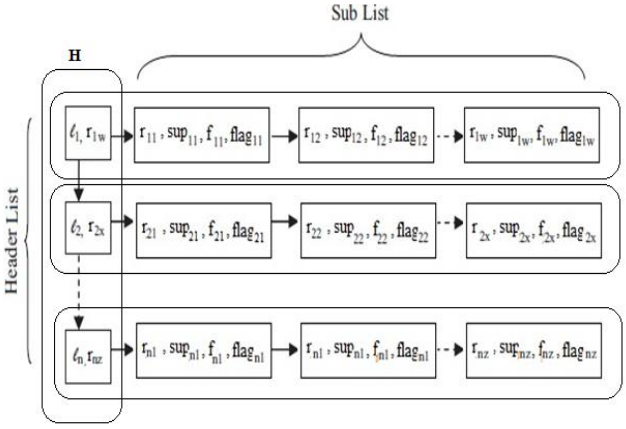

Fig. 6. Illustration of SCI-Tree

In the above SCI-Tree, $[l_1, r_{11}], [l_1, r_{12}], \ldots\ldots\ldots, [l_1, r_{1w}]$, $[l_2, r_{21}]$,....., $[l_n, r_{zz}]$ are closed intervals and sup_{11} , sup_{12} ,……., sup_{1w} , sup_{21} , sup_{22} ,……, sup_{nz} are the supports of the intervals respectively.

VI. CONSTRUCTION OF SCI-TREE

The algorithm for construction of SCI-Tree has two parts. In the first part, it updates the header list using *Header-Update* procedure and in the second part, it updates the sub lists using *SubList-Update* procedure. The sub-list in SCI-Tree maintains closed intervals, their respective frequency, *freq* and support counts, *sup*. For an interval $I=[l,r]$, l is the left endpoint and r is the right endpoint of I. i.e. $l = part(r)$ and $r = part(l)$. In the header list updation, if the left endpoint of the newly inserted interval, I is present in a node h in the header list H, say $h=[l',r']$ and $h.l' = I.l$ and $h.r' < I.r$, then the right value of h is updated to $l.r$. i.e. h is updated as $h=[l',r]$. Otherwise, I is inserted in the correct position of the header list.

In the SCI-tree constructions three counters are used: $I_{contain}$, $I_{contain\_below}$ and $I_{current}$. $I_{contain}$ contains the frequency counts of the newly inserted interval $I=[l,r]$ i.e. the total

number of intervals in interval dataset containing I. $I_{contain\_below}$ contains the frequency counts of the newly generated closed interval $[l', r]$ in the interval dataset, where $l < l'$. i.e. the total number of intervals in interval dataset containing I. The counter $I_{current}$ contains the support counts of the intervals $[l', r]$ in the interval dataset, where $l \le l'$ i.e. total number of intervals in interval dataset containing I. $I_{contain}$, $I_{contain\_below}$ and $I_{current}$ are initialized to *zero* before the sub-list modification starts for each newly inserted input interval.

When a new interval $I = [l, r]$, with frequency, I_{freq} is added to the interval data set *TDB*, the sub list *S* of all the header nodes in the header list are traversed and two types of updates are done in the sub-lists updation:

1) *Generation of new closed intervals together with computing their support counts, and*
2) *Incrementing support counts of all existing closed intervals that are contained in the new input interval* I.

Depending on the nature of updates, the sub lists in the SCI-Tree can be treated as of three types.

Type-1: Sub list S_H ***of the header nodes*** H_H ***having left end point less than*** l ***. i.e.*** $H_H.l < l$

For the new interval $I = [l, r]$, sub list S_H will not require any changes to be made, i.e. no new node will be added to S_H and no change in the support values of the all existing nodes ep_H in S_H. However, an interval, $[H_H.l, ep_H.r]$, for any node ep_H in S_H in the SCI-Tree having non-empty intersection with the input interval i.e. $H_H.l < l \le H_H.r$ and $ep_H.r \le r$, will generate new closed intervals $[l, ep_H r]$ and is therefore added as a node ep_I with $ep_I.r = ep_H.r$ to the sub list S_I of the header node H_I in correct position. The support count of a newly generated interval is initialized as $ep_I.\sup = ep_H.\sup$. Since, $[l, ep_I.r]$ is not an input interval, *flag* and *freq* are set to zero. i.e. $ep_I.flag = ep_i.freq = 0$. But, if a node ep_I with $ep_I.r = ep_H.r$ is already present in S_I i.e. $[l, ep_I.r]$ already exists then only support value is updated for that interval if required. i.e. $ep_I.\sup$ is updated as $ep_H.\sup$ iff $ep_I.\sup < ep_H.\sup$. The frequency counts of $I = [l, r]$ for all input closed intervals in S_H is also computed and is stored in $I_{contain}$ i.e. for all nodes ep_H in S_H, $I_{contain} = \sum ep_H.freq$, where $r \le ep_H.r$ and $ep_H.flag = 1$. The sum of frequencies of the input closed interval is considered because they merely contribute in the support counts of the input closed interval $[l, r]$. At the end of traversal of the sub lists S_H, all the closed intervals in S_I will have the updated support counts before the inclusion of newly inserted input interval $I = [l, r]$ in the interval database. The final support counts of all these close intervals are computed in the next sub list S_I of the header nodes H_I having left end point less than i.e. $H_I.l = l$.

Type-2: Sub list S_I ***of the header nodes*** H_I ***having left end point equal to l. i.e.*** $H_I.l = l$

Since, the support counts of a closed interval $[H_l.l, ep_I.r] \subset I$ (say, ep_I is a node in S_I) is equal to sum of its existing support and the frequency of the new interval I, so, the support counts of all the existing closed interval in the sub list S_I having right endpoint less than r are incremented by adding the frequency (I_{freq}) of the newly inserted interval I. i.e. for all the nodes ep_I in S_I with $ep_I.r < r$, its support $ep_I.\sup$ is set to $ep_I.\sup + I_{freq}$. The modification in S_I for the input closed interval $[H_l.l, ep_I.r] = I$ is performed as explained below.

If a node ep_I for a closed interval $[H_l.l, ep_I.r]$ with $ep_I.r = r$ is already present in S_I then only the support of this interval is updated as $ep_I.\sup = ep_I.\sup + I_{freq}$, because its existing support is obtained from the previous sub lists. The computed support is stored in $I_{current}$ as $I_{current} = ep_I.\sup$. The interval $[H_l.l, ep_I.r]$ is marked as an input closed interval if it is previously marked as a generated closed interval. The marking is done by changing the *flag* in ep_I as $ep_I.flag = 1$. The frequency, *freq* of ep_I is updated as $ep_I.freq = ep_I.freq + I_{freq}$ as well. Otherwise, a new node ep_K is added to S_I in correct position for the new input closed interval $[H_l.l, ep_K.r]$ with $ep_K.r = r$. The sum of frequencies of all the input closed intervals $[H_l.l, ep'_I.r]$ which contains $[H_l.l, ep_K.r]$ in S_I is computed and is added to $I_{contain}$ computed in previous list. i.e. for all nodes ep'_I in S_I, $I_{contain} = I_{contain} + \sum ep'_I.freq$, where, $ep'_I.flag = 1$ and $ep'_I.r > r$. The support counts of the newly generated input closed interval $[H_l.l, ep_K.r]$ is computed as $ep_K.\sup = I_{contain} + I_{freq}$. This is because, the support of this interval will be the sum of frequencies of all the input intervals in the interval database including its own frequency. The currently computed support of the closed interval $[H_l.l, ep_K.r]$ is stored in $I_{current}$ as $I_{current} = ep_k.\sup$

Type-3: Sub list S_J ***of the header nodes*** H_J ***having left end point greater than*** l ***. i.e.*** $H_J.l > l$ ***.***

For all the closed intervals in S_J having non empty intersection with I will require changes for support updates and creation of new closed intervals i.e. for all intervals in S_J with $H_J.l \le r$. Since, the support counts of a closed interval $[H_J.l, ep_J.r] \subset I$ (say, ep_J is a node in S_J) is equal to sum of its existing support and the frequency of the new interval I, so, the support counts of all the existing closed interval in the sub list S_J having right endpoint less than r are incremented by adding the frequency (I_{freq}) of the newly inserted interval I .i.e. for all the nodes ep_J in S_J with $ep_J.r < r$,

$ep_J.\text{sup} = ep_J.\text{sup} + I_{freq}$ .The modification in S_J for the interval $[H_J.l, r]$ is only required when $H_J.l \le r \le H_J.r$ and is performed as explained below.

If a node ep_J for a closed interval $[H_J.l, ep_J.r]$ with $ep_J.r = r$ is already present in S_J then only the support of this interval is updated as $ep_J.\text{sup} = ep_J.\text{sup} + I_{freq}$, because its existing support is obtained from the previous sub lists. The computed support count is stored in $I_{current}$ as $I_{current} = ep_J.\text{sup}$. Otherwise, a new node ep_K is added to S_J in correct position for the new input closed interval $[H_J.l, ep_K.r]$ with $ep_K.r = r$.

The sum of frequencies of all the input closed intervals $[H_J.l, ep'_I.r]$ which contains $[H_J.l, ep_K.r]$ in S_J is computed and is stored in $I_{contain\_below}$ .i.e. for all nodes ep'_I in S_J, $I_{contain\_below} = I_{contain\_below} + \sum ep'_I.freq$, where, $ep'_I.flag = 1$ and $ep'_I.r > r$. The support counts of the newly generated closed interval, $[H_J.l, ep_K.r]$ is computed as $ep_K.\text{sup} = I_{contain\_below} + I_{current}$. This is because the support of the closed interval $[H_J.l, ep_K.r]$ will be the sum of frequencies of all the input intervals in the interval database plus its current support. The currently computed support of the closed interval $[H_J.l, ep_K.r]$ is stored in $I_{current}$ as $I_{current} = ep_k.\text{sup}$ and is followed by other sub-lists in SCI-Tree in order.

Example

Let us consider the following interval transaction database, *TDB_2* as shown in *TABLE II* and construction of *SCI-Tree* is elaborated below with diagrams:

Table II. ***Interval Dataset, TDB_2***

TID	1	2	3	4	5	6	7
I	[6,9]	[1,5]	[2,7]	[1,7]	[2,4]	[8,10]	[7,11]
freq	1	1	2	1	1	2	1

I=Interval, freq=frequency

For the better visualization of SCI-Tree, we use following notations:

Table III. ***Notations for SCI-Tree***

Symbols/Notation	Meaning
H	Header List
S_H	Sub-list of Header List *H*
]	Input closed interval (flag=1)
}	Generated Closed interval (flag=0)
Bold integer	Bold text integer represents support counts of the interval with that value. e.g. **5**. i.e. sup=5
Normal underlined integer	Normal text integer represents frequency counts of the interval with that value. e.g. 4 i.e. freq=4

Since, initial Header-List, *H=NIL*, so $I = [6,9]$, is added to the H as header node $I_H = [6,9]$ (using *Header_Update*). Now, $r = 9$ is added to the sub-list S_H of the newly created header node I_H with $flag = 1$, $freq = 1$(i.e. *type-2* sub-list in SCI-Tree as stated in *section-VI*).

The support count of the interval is *sup*=1 and $I_{current} = 1$. Fig. 7 shows the updated SCI-Tree after insertion of $I = [6,9]$

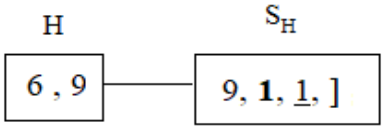

Fig-7: SCI-Tree after the insertion on interval [6, 9].

For, the input Interval, $I = [1,5]$, $I_{freq} = 1$

Initialization: $I_{contain} = 0$, $I_{current} = 0$

Using the *Header_Update* procedure, I is added to the header list H as header node $I_H = [1,5]$ before header node $[6,9]$. Now, $r = 5$ is added to the sub-list S_H of the newly created header node I_H with $flag = 1$, $freq = 1$ (i.e. type-2 sub-list in SCI-Tree as stated in section-VI). The support count of the interval is *=1* and $I_{current} = 1$. Fig. 8 shows the updated SCI-Tree after insertion of $I = [1,5]$

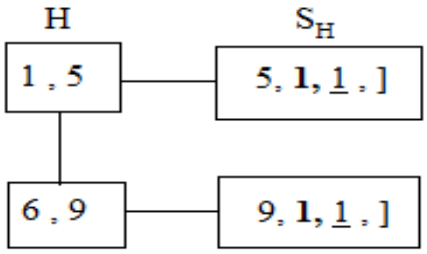

Fig-8: SCI-Tree after the insertion on interval [1, 5].

For, the input Interval, $I = [2,7]$, $I_{freq} = 2$

Initialization: $I_{contain} = 0$, $I_{current} = 0$

Using the *Header_Update* procedure, I, is added to the header list H as header node $I_H = [2,7]$ in sorted order. From the header node $[1,5]$, $r = 5$ is added as a node to the sub list S_H of I_H with $flag = 0$, $freq = 0$,*sup=1* (i.e. type-1 sub-list in SCI-Tree as stated in section-VI).

Since, $[2,7]$ is not contained by any of the input intervals in the sublist of header node $[1,5]$, so $I_{contain} = 0$. The support values of all the nodes in the sub-list of I_H with r value less than $r = 7$ is incremented by $I_{freq} = 2$ and also is $I_{contain}$ computed for the interval I in S_H. There is no change in $I_{contain}$.

Now, $r = 7$ is added to the sub-list S_H of the newly created header node I_H with $flag = 1$, $freq = 2$ (i.e. type-2 sub-list in SCI-Tree as stated in section-VI) and there is no change in $I_{contain}$ .The support count of the interval is

$\sup = I_{freq} + I_{contain} = 2 + 0 = 2$ and $I_{current} = 2$.

Now, $r = 7$ is added to the sub-list of the header node [6,9] with $flag = 0$, $freq = 0$ (i.e. type-3 sub-list in SCI-Tree as stated in section-VI) and $I_{contain\_below}$ is computed in this sub list and therefore, $I_{contain\_below} = 1$.

The support count, *sup* of the interval [6,7] is $\sup = I_{current} + I_{contain\_below} = 2 + 1 = 3$ and $I_{current} = 3$ for the header node next to [6,9].

Fig. 9 shows the updated SCI-Tree after insertion of $I = [2,7]$

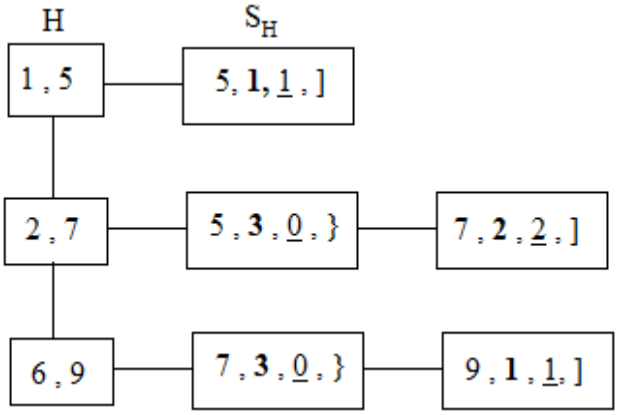

Fig-9: SCI-Tree after the insertion on interval [2, 7].

Following figures (i.e. Fig.10, Fig.11, Fig.12 and Fig. 13) indicates the SCI-Trees obtained after the insertion of input intervals: [1,7], [2,4], [8,10] and [7,11].

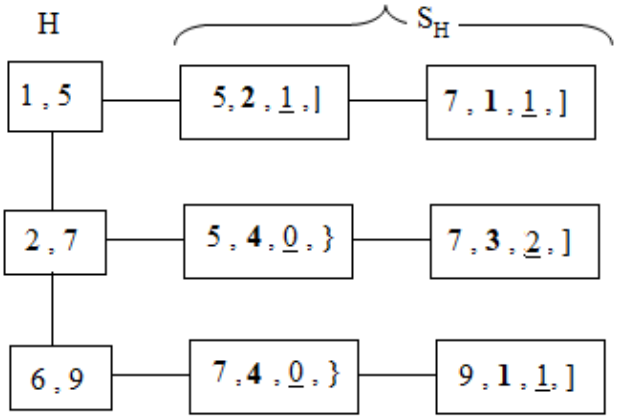

Fig-10: SCI-Tree after the insertion on interval [1, 7].

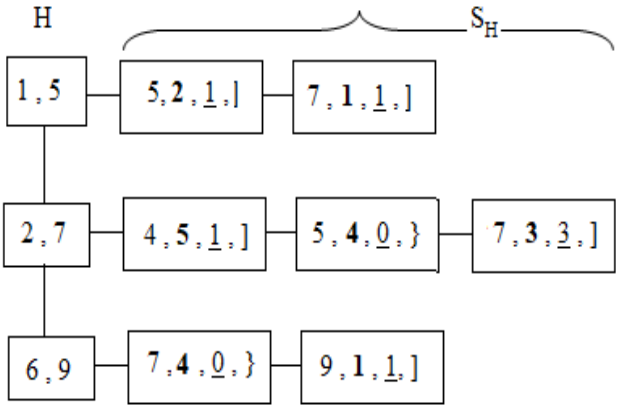

Fig-11: SCI-Tree after the insertion on interval [2, 4].

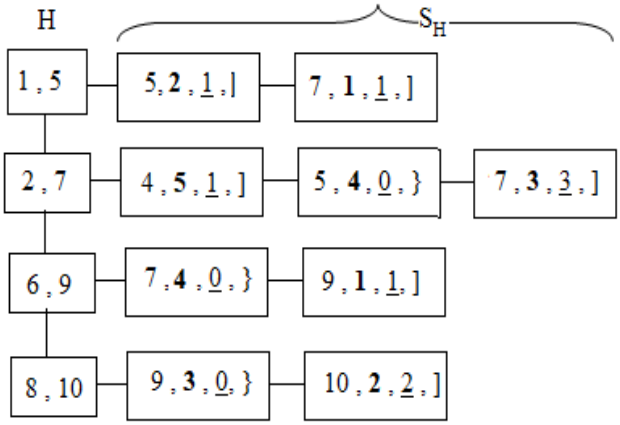

Fig-12: SCI-Tree after the insertion on interval [8, 10].

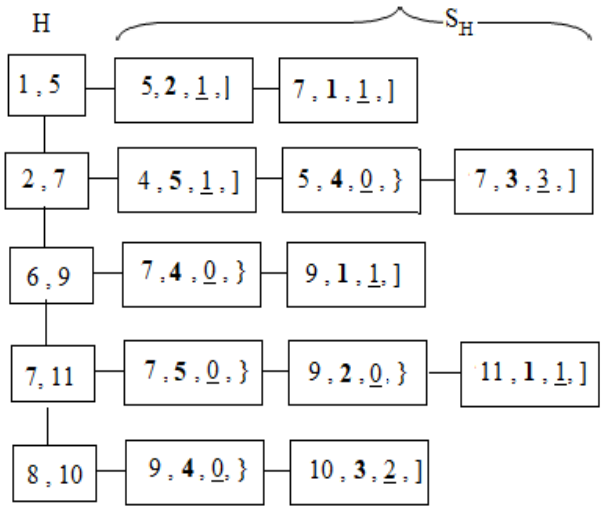

Fig-13: SCI-Tree after the insertion of interval [7, 11]

The output of the final SCI-tree in Fig.13 shows all the closed intervals along with their frequencies and support count. It also keeps the differentiation between generated closed intervals and input closed intervals. A total of *12* closed intervals obtained from the interval dataset *TDB_2*, out of which *5* closed intervals are generated closed intervals. For example, the header node [8,10] has two closed intervals associated with it: one generated closed interval indicated by '**}**' symbol and one input closed interval indicated by '**]**'. The generated closed interval of this node is [8, 9] and its support count is *4*, frequency is *0*. The input closed interval of it is [8,10] with support as *3* and frequency count as *2*.

VII. ALGORITHM FOR CONSTRUCTION OF SCI-TREE

Header-Update

Input:

SCI-Tree,

Interval $I = [I.left, I.right]$

frequency of $I = I_{freq}$

Algorithm:

// In the Algorithm

//H → Header List

// e_H → Node in the Header List H

// $e_H.l \rightarrow$ Left data point of the Header Node e_H

// $e_H.r_{\max} \rightarrow$ Right data point of the Header Node e_H

1. if H is empty
2. add a new node I in H
   //create & add node I in H
3. else if (H has a node $e_H = [l, r_{\max}]$ and $e_H.l = I.left$)
4. If $e_H.r_{\max} < I.right$ then
5. $e_H.r_{\max} = I.right$
   // modify the right value of header node e_H as $I.right$
6. else
7. Add node I to H in sorted order
   // Create and add node I in H and place according to ascending order sorted position

SubList-Update
Input:
SCI-Tree,
Interval $I = [I.left, I.right]$
frequency of $I = I_{freq}$

Algorithm:
1. $I_{contain} = I_{current} = 0$; // initialize $I_{contain}$, $I_{current}$ to 0
2. **for** all header nodes H the in the Header List **{**
3. **if** ($H.l < l.left \le H.r$) **{**
4. **for** all endpoints ep_H in the sub-list of H **{**
5. **if** ($I.left \le ep_H.r \le I.right$) **{**
6. **if** (a node ep_I with $ep_I.r = ep_H.r$ is not in S_I) **{**
7. create a new node ep_I and add to S_I maintaining the sorted order;
8. $ep_I.r = ep_H.r$;
9. $ep_I.flag = ep_I.freq = 0$;
10. $ep_I.\sup = ep_H.\sup$;
11. }
12. **else**
13. **if** ($ep_I.\sup < ep_H.\sup$)
14. $ep_I.\sup = ep_H.\sup$;
15. }
16. **if** ($I.right \le ep_H.r$ and $ep_H.flag == 1$)
17. $I_{contain} = I_{contain} + ep_H.freq$;
18. }
19. }
20. **if** ($H.l == I.left$) **{**
21. $flag = 0, flag\_1 = 0$; //used to check existence of a node
22. **for** all endpoints ep_I in the sub-list of S_I {
23. **if** ($ep_I.r == I.right$) {
24. $flag\_1 = 1$;
25. $ep_I.\sup = ep_I.\sup + I_{freq}$;
26. **if** ($ep_I.flag == 0$)
27. $ep_I.flag = 1; ep_I.freq = I_{freq}$;
28. **else**
29. $ep_I.freq = ep_I.freq + I_{freq}$;
30. $I_{current} = ep_I.\sup$;
31. }
32. **if** ($ep_I < I.right$)
33. $ep_I.\sup = ep_I.\sup + I_{freq}$;
34. **if** ($ep_I.r > I.right$ and $ep_I.flag == 1$)
35. $I_{contain} = I_{contain} + ep_I.freq$;
36. }
37. **if** ($flag\_1 == 0$) // $I.rignt$ is not present in S_I {
38. create a new node ep_K and add it to S_I Maintaining sorted order;
39. $ep_K.r = I.right$; $ep_K.flag = 1$;
40. $ep_K.freq = I_{freq}$;
41. $ep_K.\sup = I_{contain} + I_{freq}$;
42. $I_{current} = ep_k.\sup$;
43. }
44. }
45. **if** ($I.left < H.l \le I.right$) {
46. $I_{contain\_below} = 0$; $flag\_2 = 0$;
47. **for** all endpoints ep_H in the sub-list of H {
48. **if** ($ep_H.r == I.right$) {
49. $flag\_2 = 1$; $ep_H.\sup = ep_H.\sup + I_{freq}$;
50. $I_{current} = ep_H.\sup$;
51. }
52. **if** ($ep_H.r > I.right$ and $ep_H.flag == 1$)
53. $I_{contain\_below} = I_{contain\_below} + ep_H.freq$;
54. **if** ($ep_H.r < I.right$)
55. $ep_H.\sup = ep_H.\sup + I_{freq}$;
56. }
57. **if** ($flag\_2 = 0$) // I.rignt is not present in S_H {
58. **if** ($H.l \le I.right \le H.r$) {
59. create a new node ep_K and add it to S_H maintaining sorted order;
60. $ep_K.r = I.right$; $ep_K.flag = 0$;
61. $ep_K.\sup = I_{contain\_below} + I_{current}$;
62. $ep_K.freq = 0$; $I_{current} = ep_K.\sup$;
63. }
64. }
65. }
66. $H = H \rightarrow next$;
67. }

VIII. PERFORMANCE ANALYSIS OF SCI-TEE

In SCI-Tree, Header List, H is sorted in increasing order of the left endpoints l and sub list, S_I of each header node I is also sorted in increasing order of right endpoints r. Linked list implementation is used for both H and the S_I's. So, $O(n)$ steps are required for header list updation for a given input interval.

If an interval $I=[l,r]$ with frequency f newly comes into the existing interval dataset then to modify the SCI-Tree appropriately, two routines *Header-Update* and *Sublist-Update* are needed to be executed. Header-Update looks for a node with left endpoint as l in H. If such a node is not found then it is inserted in H. This therefore takes $O(n)$ steps. After this, in *Sublist-Update*, for each node for l' in H with $l'<l$, S_l is modified in $O(n)$ steps and support counts of each node in S_l is updated simultaneously. For each node for l' in H with $l'>l$, $S_{l'}$ is modified in $O(n)$ steps and their support counts is updated at the same time. Finally, S_l is further modified in $O(n)$ steps with support counts updated all together. Since, there are at most n number of sub-lists and length of each sublist is at most n, so the *Sublist-Update* routine requires $O(n^2)$ steps. Therefore, the worst case time complexity for construction of SCI-tree for one input interval is $((O(n)+O(n^2))=O(n^2)$. If the dataset has n intervals then for inserting n intervals in SCI-Tree, the algorithm will require $n*O(n^2)=O(n^3)$ time in worst case.

IX. RESULT AND DISCUSSION

The proposed algorithm has been implemented by developing programs in C++. We used a Linux PC having Intel Core i3 processor with 1GB RAM. To test the algorithm, it has been applied to the dataset that contains a total of 10031 records collected from Bodhidroom", [1][2]. The records contain the information about the login and logout time of the users of the web portal from 31/3/2009 to 14/10/2011. Adding all the intervals in SCI-Tree incrementally, we have found 11780 closed intervals with support counts in this dataset. The same dataset was used in [1] for finding the set of closed intervals where the same number of closed intervals (*shown in Table IV*) was obtained, verifying the correctness of our method and its implementation.

Table IV. ***Results of CI-Tree and SCI-Tree as applied on dataset obtained from "Bodhindrom"***

No of Intervals	No of Closed Intervals with *Support Count* in CI-Tree	No of Closed Intervals with *Support Count* in proposed SCI-Tree
10031	11780	11780

The proposed method is also verified by using a synthetic datasets used in [12], where a sequence of *n* intervals with left endpoints are generated a data generator. The left endpoints are distributed uniformly between l and a given $l_{\max}$.

The Poisson distribution with a given mean is used to distribute the lengths of these intervals. The synthetic datasets are generated with $\max=1000$ and $mean=200$.

We implemented both CI-Tree [1] and SCI-Tree for the generated synthetic datasets and the results are found as follows-

Table V. ***Results of CI-Tree and SCI-Tree as applied on Synthetic dataset***

No of Intervals	No of Closed Intervals with *Support Count* in CI-Tree	No of Closed Intervals with *Support Count* in proposed SCI-Tree
1000	70351	70351
2000	136474	136474
3000	167414	167414
4000	181588	181588
5000	190504	190504

X. CONCLUSION AND FUTURE WORK

In this paper, we proposed an incremental algorithm for computing the support counts of all closed intervals in a interval transaction database. The extraction of all closed intervals and their support counts is important because by using the set of closed intervals and their respective support counts, it is possible to extract the support of any interval in an interval dataset. The existing CI-Tree algorithm can mine all the closed intervals from an interval dataset but it cannot compute the support counts. So, the proposed method is an enhancement of existing incremental CI-Tree. In addition, if any user defined threshold of minimum support is given as input, all the frequent closed intervals can be extracted by a scan of the SCI-Tree formed. The tree also keeps a *flag* to differentiate the generated and input closed intervals. The frequency of input intervals is also maintained along with the support counts. The effectiveness of the proposed algorithm is verified by testing on both real-life datasets and synthetic data sets. Our proposed algorithm can be more useful in mining meaningful information from real time interval datasets by using its associated information such as support counts, frequency etc. . Future works include extending the proposed algorithm to mine multi-dimensional closed intervals, discovery of periodicities in temporal events etc. Attempts can also be made to improve the performance of the algorithm by using other suitable data structures.

REFERENCES

1. N. Sarmah, A. K. Mahanta, "An Incremental Approach for mining all Closed Intervals from an Interval Database", IEEE International Advance Computing Conference (IACC), pp. 529-532, Feb 2014.
2. J. F. Alen, "Maintaining Knowledge about Temporal I ntervals", Communication of the ACM, Vol.26, Nov 1983.
3. Po-shan Kam, Ada Wai-chee Fu, "Discovering Temporal Patterns for Interval-based Events", Proceedings of the second International Conference on Data Warehousing and Knowledge Discovery, p. 317-326, 2000.
4. S. Wu and Y. Chen, "Mining nonambiguous temporal patterns for interval-based events," IEEE Transactions on Knowledge and Data Engineering, vol. 19, no. 6, pp. 742-758, June 2007.
5. R. Villafane, K. Hua, D. Tran, B. Maulik, "Knowledge Discovery from Series of Interval Events," Journal of Intelligent Information Systems, vol. 15, no. 1, pp. 71-89, 2000.
6. R. Villafane, K. Hua, D. Tran, B. Maulik, "Mining Interval Time-series," Data Warehousing and Knowledge Discovery, pp. 318-330, 1999.
7. D. Patel, W. Hsu, M. Lee, "Mining Relationships Among Interval-based Events for Classification," Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data, pp. 393-404, 2008.
8. Yi-Cheng Chen, Wen-Chih Peng, Suh-Yin Lee, "CEMiner – An Efficient Algorithm for Mining Closed Patterns from Interval-based Datasets", 11th IEEE International Conference on Data Mining, 2011.

9. J. Lin, "Mining Maximal Frequent Intervals", Proceedings of 2003 ACM symposium on Applied Computing, p.426-431, ACM, New York, 2003.
10. N. Pasquier, Y. Bastide, R. Taouil, L. Lakhal, "Discovering Frequent Closed Itemsets for Association Rules" In Proc. of 7th International Conference on Database Theory (ICDT' 99), pages 398-416, Jerusalem Israel, Jan 1999.
11. A. K. Mahanta, M. Dutta, "Mining Closed Frequent Intervals from Interval Data", International Journal of Applied Science and Advance Technology, Vol.1, pp.1-3, January-June 2012.
12. M. Dutta, "Development of Efficient Algorithms for Some Problems in Interval Data Mining," Ph.D thesis, Gauhati University, 2012.

AUTHORS PROFILE

Dwipen Laskar, M.Tech in Information Technology and Assistant Professor at department of Computer Science at Gauhati University, Guwahati, Assam. He has more than 12 years of teaching experiences and a member in IARA, IACSIT and IAENG. His research area covers data mining and image processing.

Naba Jyoti Sarmah, Ph.D. from Gauhati University, is an Assistant Professor at department of B.Voc.(IT), Nalbari Commerce College, Nalbari, Assam. His current area of research includes interval data mining, temporal data mining etc.

Anjana Kakoti Mahanta, Ph.D. is a Professor in department of Computer Science, Gauhati University. She has more than 30 years of teaching experience. Her current area of research is Algorithms and Data Mining. In 2007, under a bilateral exchange program of Indian National Science Academy (INSA) and Polish Academy of Sciences, Poland, she visited the University of Warsaw for three months and had done research work in collaboration with faculty members in the department of Computer Science of the University.