CHAPTER 5
CFI MINING IN DATA STREAMS USING LANDMARK WINDOW

The extraction of closed frequent itemsets from the data stream using a sliding window model is discussed in the previous chapter. The proposed HATCI algorithm, discussed in the previous chapter is a single-pass algorithm that mines closed frequent itemsets from the current window of transactions whenever a user request arrives. It employs a sliding window model as many applications need to concentrate on only the most recent transactions. But some applications need to maintain not only the recent transactions but all transactions seen from a specified time till the current time. For such applications, sliding window model is not sufficient. Landmark windows that store all transactions from a specified time, called landmark, till the current time, are a better choice for such applications. In a landmark window, whenever a transaction arrives, if its time of arrival is after the specified time, it is added to the window without any further processing.

5.1 CFI MINING

In a landmark window, all transactions that arrive after the specified time are maintained [58] and they cannot be discarded even if the size of the window becomes very large. The size of data stream is unbounded. So the size of the window keeps increasing as time progresses. The set of Frequent Itemsets (FIs) extracted from such large windows is very large. This necessitates the use of Closed Frequent Itemsets (CFIs) which is a lossless compressed form of FIs. The number of CFIs and hence the storage needed for them keeps increasing as the window size increases and more number of transactions are stored in the landmark window. The time needed for generating the CFIs is also on the increase as it takes a prohibitively long time to extract CFIs from the window whenever a user request arrives. Landmark windows are preferred only when the application demands it because of the above said drawbacks. Examining web server log files to determine the user access patterns [82] which may help web site owners to reorganize and redesign the web pages is one
such application which makes landmark windows mandatory in spite of the problems involved in maintaining them.

The proposed work concentrates on making use of Closed Frequent Itemsets (CFIs) for compact representation of the numerous Frequent Itemsets (FIs). It utilizes a tree organization for efficient representation of the list of Closed Itemsets (CIs) that helps to make the search and update of CIs more efficient. These operations are performed whenever a new transaction arrives on the data stream. On the arrival of a user’s request for CFIs, these CI trees are traversed and the CIs with support greater than or equal to the specified support threshold are retrieved and returned to the user.

Data streams can be modeled using landmark windows, sliding windows or damped windows. Landmark-window model focuses on the values between a specific time called landmark and the current time. CLICI (Concept Lattice based Incremental Closed Itemset) algorithm [31] proposed by Anamika et al. is used for mining all recent CIs in a landmark window model of the online data stream in the year 2010. In the year 2009, Liu et al. proposed FP-CDS (FP-tree based Closed itemsets mining from Data Stream) algorithm [58] that uses a landmark window. Also in the year 2009, Yen et al. proposed CloStream (Closed Frequent Itemsets over Data Stream) algorithm [97] to mine CFIs using an incremental 1-scan approach.

The proposed algorithm uses the landmark window model with a single scan over the data stream to build a forest of CI trees. Whenever a new transaction arrives, the trees are updated incrementally. Incremental updates are very essential in handling data streams because reconstructing entire structure from the scratch for every transaction that arrives on the data stream is infeasible and time consuming as the transactions keep arriving indefinitely and at a fast rate. The CIs affected by the new transaction are identified by searching the forest of CI trees and their support is incremented. Any new CIs created as a result of the new transaction are also inserted into the appropriate CI tree in its correct location. The correct location is determined by performing a binary search in the appropriate CI tree. Special techniques like marking the visited nodes, marking the completely traversed trees and pruning the right sub trees from the search space by checking the second item present in the CI are used to make the search and insertion of CIs more efficient.
When a user requests for the current CFIs, the proposed algorithm performs an in-order traversal on all the non-empty CI trees present inside the Forest of Closed Itemset Trees (FOCIT), finds the CIs with support greater than or equal to the user specified support threshold and forms the set of CFIs. After completing the traversal of all trees in the FOCIT, this set of CFIs is returned to the user.

5.2 PRELIMINARY DEFINITIONS

**Definition 1: Landmark Window**

The window that maintains all the transactions from a specified time called as landmark till the current time is called as the landmark window. Let the specified time be denoted as \( t_{sp} \) and the current time be denoted as \( t_{ct} \). Any transaction arriving at time \( t_m \) is valid iff the following is satisfied

\[
t_{sp} \leq t_m \leq t_{ct}.
\]

**Definition 2: Closed Itemset (CI)**

As discussed in Chapter 4, an itemset, \( X \) is said to be a CI iff no superset of \( X \) has the same support as \( X \). For example consider a portion of the data stream \( \text{TDS}= \{ < T_1, (acd) >, < T_2, (bce) >, < T_3, (abce) >, < T_4, (be) > \} \). Here the 3-itemset ‘bce’ has support 2 and it does not have any superset with the same support. So ‘bce’ is a closed itemset. But the 1-itemset ‘b’ whose support is 3 is not a closed itemset since its superset ‘be’ also has the same support 3.

**Definition 3: Closed Frequent Itemset (CFI)**

As discussed in Chapter 4, an itemset, \( X \) is said to be a CFI iff no superset of \( X \) has the same support as \( X \) and \( X \) is frequent. For example consider a portion of the data stream \( \text{TDS}= \{ < T_1, (acd) >, < T_2, (bce) >, < T_3, (abce) >, < T_4, (be) > \} \). Assume that the user-specified support threshold, \( s \) is 60%. The frequency threshold is 2. ( \( |\text{TDS}| \times s = 4 \times 60\% \approx 2 \) ) In the illustrated example, ‘c’ is a CFI. The 1-itemset ‘c’ has support 3 (since it is appearing in the transactions \( T_1, T_2, \) and \( T_3 \) and none of its superset has support 3. Also the support of ‘c’ is greater than the frequency threshold, 2. So ‘c’ is a CFI. On the other hand, the 3-itemset ‘acd’ with support 1 (since it appears only in the transaction \( T_1 \) ) is a closed itemset but not
a closed frequent itemset since its support is less than the frequency threshold, 2. The importance of CFIs and its advantages over MFIs are discussed in Chapter 4.

**Definition 4: Tree**

A tree is a non-linear data structure that consists of a root node and potentially many levels of additional nodes that form a hierarchy. It is a structure consisting of one node called the root and one or more subtrees. A tree can be empty with no nodes called the null or empty tree.

**5.3 FOCIT ALGORITHM DESCRIPTION**

The proposed algorithm, FOrest of Closed Itemset Trees (FOCIT), is a single-pass incremental algorithm that processes the transactions that arrive on the data stream and constructs a forest of Binary Search Trees (BST) to store the closed itemsets in the transactions seen so far in the data stream. The forest of BSTs is updated incrementally whenever a new transaction arrives.

**5.3.1 Binary Search Tree**

A Binary Search Tree (BST), [6] also known as an ordered binary tree, is a node-based data structure in which each node has no more than two child nodes. Each child must either be a leaf node or the root of another binary search tree. The left subtree contains only nodes with keys less than the parent node; the right subtree contains only nodes with keys greater than the parent node. This property called as Binary Search Property is satisfied at every node in the BST. A BST satisfies the following properties:

- The left subtree of a node contains only nodes with keys less than the node's key
- The right subtree of a node contains only nodes with keys greater than the node's key
- The left and right subtree each must also be a binary search tree
- There must be no duplicate nodes

An ordered collection of zero or more binary search trees forms a forest of binary search trees. The major advantage of binary search trees over a binary tree is that the operations on a BST like search, insert, and delete are more efficient.
when compared to those on a binary tree. The operations performed on a BST in the proposed algorithm are insertion and traversal. Deletion is not needed since landmark window model is used and there is no need for removing any closed itemset from the trees.

**Insertion**

For insertion operation, the correct position in which to insert the key has to be found. For this, the key is compared to the root element. If the key is less than the root the search continues with the left subtree. If the key is greater than the root the search continues with the right subtree. If it is equal, “Failure to insert” error is reported as the BST cannot have duplicate keys. If it is a new key-value, eventually an external node (null node) is reached and the key-value pair is inserted at that position, as its right or left child, depending on the node's key. In other words, the root is examined and the new node is inserted recursively into the left subtree if its key is less than that of the root, or the right subtree if its key is greater than or equal to the root. The procedure for insertion of a value into a BST is shown below in the Fig. 5.1.

<table>
<thead>
<tr>
<th>Procedure Insert( Node *node, int value)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> Node *node, int value</td>
</tr>
<tr>
<td><strong>Output:</strong> Updated Tree</td>
</tr>
</tbody>
</table>

```c
Procedure Insert( Node *node, int value)  
Input: Node *node, int value           
Output: Updated Tree                   
Begin                                    
    if (value < node->key)              
        if (node.leftChild == NULL)    
            node.leftChild = value    
        else                           
            Insert(node.leftChild, value) 
    else                                
        if(node.rightChild == NULL)   
            node.rightChild = value   
        else                          
            Insert(node.rightChild, value) 
End                                        
```

Fig. 5.1 Procedure Insert
The above procedure modifies the tree in place. It uses only constant heap space. This operation requires time proportional to the height of the tree, which is $O(\log n)$ time in the average case.

**Traversal**

A tree traversal is used to visit and process all the nodes in the binary search tree in a systematic manner. There are three types of traversals: in-order, pre-order, and post-order. In-order traversal visits all nodes by recursively traversing the left subtree of the root node, accessing the node itself, then recursively traversing the right subtree of the node, continuing this pattern with each node in the tree as it is recursively accessed. Pre-order traversal visits all nodes by accessing the node first, then recursively traversing the left and right subtrees in that order. Post-order traversal traverses the left and right subtrees recursively and then accesses the node. Among the three traversals, only the in-order traversal results in a key-wise sorted list of the nodes the BST. For this reason, the in-order traversal is chosen to traverse the BST when the user requests the CFIs in the current sliding window and the simplified code to perform this traversal is shown in Fig. 5.2.

<table>
<thead>
<tr>
<th><strong>Procedure Inorder_traversal( Node n)</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>//Input: Node n</td>
</tr>
<tr>
<td>//Output: Nodes visited in order</td>
</tr>
<tr>
<td>Begin</td>
</tr>
<tr>
<td>If (n == nil)</td>
</tr>
<tr>
<td>return</td>
</tr>
<tr>
<td>Inorder_traversal(n.leftChild)</td>
</tr>
<tr>
<td>Access(n.value)</td>
</tr>
<tr>
<td>Inorder_traversal(n.rightChild)</td>
</tr>
<tr>
<td>End</td>
</tr>
</tbody>
</table>

Fig. 5.2 Procedure Inorder_traversal

Any type of tree traversal requires $O(n)$ time, since it must visit every node. This algorithm is also $O(n)$, so it is asymptotically optimal.
Balanced trees like the Adelson-Velskii and Landis tree (AVL tree) and the red-black tree are found to be not better than BST for the proposed work. An AVL tree [6] is a binary search tree which has the following properties:

1. The sub-trees of every node differ in height by at most one
2. Every sub-tree is an AVL tree.

AVL tree takes up more space than the BST, as AVL tree has to remember the balance factor of each node in addition to the values maintained by a BST. The operations on an AVL tree are also slower as the balance factor has to be computed and maintained at each node. Single-rotations or double-rotations may be necessary to maintain the balance of the tree whenever an insert operation takes place.

The only advantage to be gained by using an AVL tree instead of a BST is that, the height of the tree is reduced to a large extent when the values are inserted into the tree in increasing or decreasing orders. This helps in reducing the search time during insertion. Such insertions will create splayed trees in the case of BST whereas AVL trees will be balanced, as after each insertion rotation is performed to rebalance the tree if necessary. But this is the worst case scenario and in the real world applications, worst case input is very rare. In the proposed work, the transactions that represent the user sessions are random and they exhibit the average case behavior and never the worst case behavior. The time complexity of AVL trees and Binary Search Trees are the same in the average case but AVL trees take up more space and the operations are also slower. So AVL trees have not been used in the proposed work.

Red-black trees [6] are an evolution of binary search trees that aim to keep the tree balanced without affecting the complexity of the primitive operations. This is done by coloring each node in the tree with either red or black and preserving a set of properties that guarantee that the deepest path in the tree is not longer than twice the shortest one. A red-black tree is a BST with the following properties:

1. Every node is colored with either red or black
2. All leaf (nil) nodes are colored with black; if a node’s child is missing then we will assume that it has a nil child in that place and this nil child is always colored black
3. Both children of a red node must be black nodes
4. Every path from a node n to a descendent leaf has the same number of black nodes (not counting node n). This number is called as the black height of ‘n’, which is denoted by bh(n)

Red-black trees take up more space than Binary Search Tree because the color of each node has to be maintained in addition to the other values in the node. Also, the insertion operation takes more time since additional operations are needed to determine if any violation of the red-black tree properties occurs after the insertion. If any property is violated then corrective action has to be taken to make sure that all the properties of red-black trees are satisfied. In the proposed work, the majority of the operations performed are insertions and the inserted values are also random. This makes the choice of BST more suitable than the red-black tree. Also only if removal operation is performed often, balancing trees like AVL and red-black are needed, because BST unbalances due to removals leading to increasing heights which leads to increasing search times.

5.3.2 FOCIT Structure

Forest Of Closed Itemset Trees (FOCIT), is a group of Binary Search Trees (BST) used to store the Closed Itemsets(CIs) in the transactions seen so far in the data stream. Each tree is used to store the closed items that begin with a particular item. For example CIs that begin with item ‘a’ will be in one BST and those that begin with item ‘b’ will be in the next BST and so on. As each transaction arrives, the existing CIs are updated efficiently and new ones are added if necessary. Given a transaction data stream, TDS and its FOCIT, when a new transaction T_n with its itemset X_n arrives, the algorithm updates the forest by considering only the trees in the FOCIT and without bothering about the earlier transactions.

The individual trees in the FOCIT are denoted as CI_i which has all the CIs starting with item, i. This grouping is done to limit the search needed when the CIs have to be updated on the arrival of the new transaction. The closed itemset tree CI_i is a binary search tree with the following node structure

| Left | CI | Support | Right |
Here *Left* and *Right* fields lead to the left and right children of the current node, *CI* has the closed itemset and *Support* field records the number of transactions in which the particular closed itemset is present. The FOCIT algorithm shown in Fig. 5.3 goes through the following three phases:

(i) Building Phase that constructs the intersection tree, ITree for the new transaction
(ii) Update Phase that uses the intersection tree built in the previous phase to update the forest of closed itemset trees, FOCIT.
(iii) Generation Phase that generates the closed frequent itemsets by traversing the FOCIT structure when requested by the user.

Algorithm FOCIT

// Input : A new transaction, T_n
// Output : Updated FOCIT
// Global : n – number of transactions seen so far
//         FOCIT- Forest of CI trees

Begin
1. Repeat while T_n ≠ null
2. If n=1 //First transaction
   Extract the first item, i in the transaction
   Insert T_n as the root node of the binary search tree CI_i with support=1
3. Else
   Call Build_ITree(T_n) algorithm // Phase 1
   Call Update algorithm // Phase 2
4. If a user request for CFI arrives
   Call Generate algorithm //Phase 3
End

Fig. 5.3 Algorithm FOCIT

The transaction can be discarded once the update phase is complete. When this phase is completed, the support of the closed itemsets affected by the new transaction are updated and any new CIs created by this transaction are also added
with support one. There is no further need for the transaction and hence it is
discarded.

**Building Phase**

In this phase, Algorithm Build_ITree shown in Fig. 5.4 is used and the
Closed Itemset(CI) trees are searched to find the CIs that shall be affected by the new
transaction. The CIs that shall be affected are the ones that have at least one common
item shared with the new transaction. The maintenance of the CIs as a forest of BSTs
helps in this identification. Whenever a new transaction arrives, an Intersection Tree
(ITree), maintained also as a binary search tree, is built by processing the CIs that
shall be affected by this new transaction. The structure of a node in this ITree is

<table>
<thead>
<tr>
<th>Left</th>
<th>Intersection</th>
<th>Address</th>
<th>Support</th>
<th>Right</th>
</tr>
</thead>
</table>

where *Intersection* field is used to store the intersection of the new transaction and
the closed itemset in the current node of the tree CI, being processed and *Address*
field indicates whether this intersection is a CI already in the FOCIT. If the *Address*
field is null, it indicates that this intersection is not present already as a CI. Otherwise
it gives the address of the CI which matches.

On extracting an item, i from the new transaction, T_n even without
checking it can be confirmed that all CIs in the tree CI, will be affected by the new
transaction. No checking is needed here as FOCIT has been organized in such a way
that the tree CI, has all CIs that start with the same item, i. This tree is traversed and
the following process is performed at each node. Find the intersection of
the transaction with the CI in that node. Create a node with *Intersection* field set to the
intersection obtained, *Address* field set to null, *Left* and *Right* fields set to null and
*Support* field set to one more than that of the node in the tree CI.

If this intersection is not already present in the ITree, insert it into the
tree after finding the correct position of this node in the ITree. If the intersection is
present already in the ITree, then check its support. If support of new intersection
node found is greater than that already in the ITree, then update the support field of
node in ITree to the new support. Otherwise just skip and proceed. If the intersection
found and the CI in the node being processed is the same, then set the *Address* field
of the node in ITree to the node in FOCIT tree. This helps to avoid searching for the
node again in the following update phase when the nodes in the intersection tree are
inserted into the FOCIT.

**Algorithm Build_ITree(Tₙ)**

| // Input: New Transaction, Tₙ |
| // Output: Intersection Tree, ITree maintained as a Binary Search Tree |
| // Global: FOCIT – Forest of CI trees |

**Begin**

1. Create a new node and set Intersection=Tₙ, Address=null, Left=Right=null, Support=1, and insert it into ITree as root node.
2. For each item, ‘i’ in Tₙ do
3. For each tree CIᵢ where j≤i do
4. Traverse the CIᵢ tree and process each unvisited node, ‘p’ as follows
5. Find intersection I = Tₙ ∩ p.CI
6. Create new node, n and set the values Left= Right=null, Intersection=I, Address=null and Support=p.Support+1
7. If intersection, I not in ITree
8. Insert node, n into the ITree at its correct position
9. If I=p.CI
10. Set n.Address= address of p
11. Else // I already present in ITree as node, q
12. If n.Support > q.Support
13. Assign q.Support =n.Support

**End**

**Fig. 5.4 Algorithm Build_ITree**

The FOCIT trees, CIᵢ with j<i are also searched. Since the closed itemsets in these trees start with j and may contain i in it as the itemsets are in lexicographic order. For example, if the current item extracted from the new transaction is ‘c’, first the tree CIₙ is examined completely and then the trees CIₐ and CIₖ need to be searched since they may have closed itemsets with item, ‘c’ in them like abc, acd, bc, bcd, and many more. The trees CIₐ, CIₖ, and so on need not be searched since there is no possibility of ‘c’ occurring in any of the nodes. All
itemsets in tree CI\textsubscript{d} start with ‘d’ and so the remaining items in the itemsets will definitely be greater than ‘d’ and will not contain ‘c’.

Similarly all itemsets in tree CI\textsubscript{e} start with ‘e’ and so the remaining items in the itemsets will definitely be greater than ‘e’ and will not contain ‘c’. This is true for the remaining trees in the forest. So they need not be searched and the search can stop after examining the trees CI\textsubscript{a}, CI\textsubscript{b} and CI\textsubscript{c}. All nodes in the tree CI\textsubscript{c} must be processed and intersections calculated but all nodes in trees CI\textsubscript{a} and CI\textsubscript{b} need not be searched. If the second item present in the closed itemset in a node is greater than ‘c’ then it is obvious that the right subtree of this node will not have any closed itemset with ‘c’ in it.

Also, when a transaction is taken up for processing, initially all nodes in all trees in the FOCIT are marked unvisited. After that whenever a node is visited and intersection is calculated, it is marked as visited. This helps to avoid calculating intersections repeatedly as each item in the new transaction is processed. For example, if the first item in a transaction is ‘b’ then when processing that item, intersections with nodes in CI\textsubscript{a} and CI\textsubscript{b} would have been calculated. All nodes in CI\textsubscript{b} and some nodes in CI\textsubscript{a} whose closed itemsets have ‘ab’ in them would have been visited and intersections calculated. If the next item in the transaction is ‘d’ then CI\textsubscript{b} need not be visited and also those nodes in CI\textsubscript{a} which have already been visited can be omitted safely. This marking of nodes reduces the repetitive processing of the nodes when each item in the transaction is considered.

**Update Phase**

The building phase comes to an end when all items in the new transaction have been processed. At this point of time, the intersection tree is complete and the algorithm goes into the next phase, Update Phase in which the CI trees are updated using the ITree. This is done using the Update algorithm shown in Fig. 5.5. Here each node, ‘n’ of ITree is processed by performing inorder traversal. If n.Address is null, indicating that this is a new CI, it is inserted into the appropriate CI tree in the FOCIT after finding its correct location in that tree.

The appropriate CI tree is chosen by checking the first item in the closed itemset in node ‘n’. If it is ‘i’ then the node is inserted into the tree CI\textsubscript{i}. The node is
inserted into the tree using the procedure for insertion into binary search tree since the CI trees are maintained as BSTs. If n.Address is not null, then that node is accessed from the CI tree and its Support field is updated to n.Support. When the entire traversal of the intersection tree, inorder traversal technique is used since it helps to visit the nodes in the ITree in lexicographic order. The insertion of new closed itemsets into the FOCIT is done in such a way that binary search property is maintained in the entire tree at every possible node. The correct position of the CI is found by performing binary search on the tree. The root value is compared with the new CI. If new CI is less than the root value, the search continues in the left subtree. If new CI is greater than the root value, then the search continues in the right subtree. This process continues till a null node is reached and the new CI is inserted at that position.

<table>
<thead>
<tr>
<th>Algorithm Update</th>
</tr>
</thead>
<tbody>
<tr>
<td>// Input: Intersection tree, ITree</td>
</tr>
<tr>
<td>// Output: Updated FOCIT – forest of CI trees</td>
</tr>
<tr>
<td>Begin</td>
</tr>
<tr>
<td>1. Process each node, p in the intersection tree as follows</td>
</tr>
<tr>
<td>2. Find the first item, i, in p.Intersection</td>
</tr>
<tr>
<td>3. If (p.Address = null) // new Closed Itemset</td>
</tr>
<tr>
<td>4. Insert node, p into its correct position in CI_i</td>
</tr>
<tr>
<td>5. Else // Already existing in tree CI_i</td>
</tr>
<tr>
<td>6. x=p.Address</td>
</tr>
<tr>
<td>7. Access node, x from CI_i</td>
</tr>
<tr>
<td>8. Update the support field as x.Support=p.Support</td>
</tr>
<tr>
<td>9. Repeat step 1-8 for all nodes in the intersection tree, ITree</td>
</tr>
<tr>
<td>End</td>
</tr>
</tbody>
</table>

Fig. 5.5 Algorithm Update

**Generation Phase**

When user requests for the Closed Frequent Itemsets(CFIs) seen so far in the data stream, the required minimum support (minSup) is calculated as

\[
minSup = s * n
\]

(5.1)
where ‘s’ is user specified support threshold and ‘n’ is the number of transactions seen so far in the data stream. Then as shown in Fig. 5.6 using Generate algorithm all the trees in the FOCIT are traversed, one by one. If the support count of a Closed Itemset(CI) in a node is greater than or equal to minSup, then it is added to the list of CFIs. When all trees have been processed, the list of CFIs may be returned to the user as the CFIs extracted from transactions seen so far in the data stream from the specified landmark time.

For visiting all nodes in the forest of closed itemset trees (FOCIT), again inorder traversal of each tree is preferred as it helps to generate the CFIs in lexicographic order.

![Algorithm Generate](image)

### 5.4 INTERESTING FACTS ABOUT FOCIT

**Property 1:** Adding a new transaction will not change the status of closed itemset already present in the FOCIT.
Proof: Originally, \( \forall J \supset I, \text{support}(J) < \text{support}(I) \), where J and I are closed itemsets in the FOCIT. When a new transaction, \( T_n \) is added, \( \forall J \supset I \), if \( J \subset T \) then automatically the property, \( I \subset T \) also holds. Therefore if J’s support is increased by one by the addition of \( T_n \), so is I ’s support. As a result, even after the addition the property that \( \forall J \supset I, \text{support}(J) < \text{support}(I) \) still holds. So there is no possibility that there may be a change of state of a closed itemset in the FOCIT by the addition of a new transaction.

Property 2: On intersecting a closed itemset in the FOCIT with a new transaction \( T_n \), the resultant intersection, if not empty, is also a closed itemset.

Proof: When a closed itemset, \( C \) in FOCIT is intersected with the new transaction, \( T_n \), the result of the operation, \( C \cap T_n \) is a subset of \( C \) (case i), equal to \( C \) (case ii), or null (case iii), and if intersection is not null then its support is \( \text{support}(C) + 1 \).

Case i: Intersection, \( I \) is a subset of \( C \), only if \( T_n \subset C \) or \( T_n \) contains some items present in \( C \). Originally all supersets of \( C \) will have a support less than that of \( C \) (By definition of a closed itemset). Now the new intersection has a support that is one more than that of the closed itemset, \( C \). So, all supersets of \( I \) (\( C \) and its supersets) still have a support less than that of \( I \) which makes it a closed itemset.

Case ii: Intersection, \( I \) is the same as closed itemset, \( C \) if \( T_n = C \) or \( T_n \supset C \). In this case, support(\( C \)) is incremented by 1 to indicate that \( T_n \) is contained in \( C \). No new closed itemset is generated and also \( C \) remains closed as its support is greater than that of its supersets.

Case iii: If there is no common item shared by \( C \) and \( T_n \), then the intersection is null. In this case also the property holds.

5.5 Illustration

To illustrate the FOCIT algorithm assume the transaction data stream, \( TDS= \{T_1, T_2, T_3, T_4, T_5\} \) shown in Table 5.1 and support threshold 50%. Thus the data stream TDS consists of five transactions and \( s=0.5 \) (50%)
Table 5.1 Portion of Data Stream, D containing five transactions

<table>
<thead>
<tr>
<th>$T_{id}$</th>
<th>Itemset</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_1$</td>
<td>ac</td>
</tr>
<tr>
<td>$T_2$</td>
<td>a</td>
</tr>
<tr>
<td>$T_3$</td>
<td>bd</td>
</tr>
<tr>
<td>$T_4$</td>
<td>ad</td>
</tr>
<tr>
<td>$T_5$</td>
<td>acd</td>
</tr>
<tr>
<td>$T_6$</td>
<td>B</td>
</tr>
</tbody>
</table>

When $<T_1, ac>$ arrives the FOCIT is empty as shown in Fig. 5.7 (a) and since the first item in this transaction is ‘a’, the transaction ‘ac’ is inserted as a closed itemset into the tree $CI_a$ and it becomes its root. The intersection tree built in phase 2 is shown in Fig. 5.7 (b) and the updated FOCIT after processing $T_1$ is shown in the Fig. 5.7(c)

When $<T_2, a>$ arrives, phase 1 begins. Initially the new transaction ‘a’ is inserted into the intersection tree as its root with support=1 and NULL pointers for left and right child. First item in $T_2$ is ‘a’. So, $CI_a$ is traversed. On intersecting with the root node ‘ac’ in the FOCIT we obtain ‘a’ itself and a new node is created with support 2. Already there is a node ‘a’ in the intersection tree with the support 1. So new node is not inserted into ITree but support of ‘a’ in the ITree is updated to 2. There are no more nodes in the FOCIT. So phase 1 is completed and phase 2 begins. In phase 2, the only node in ITree is checked and since its Address field is null, the node is inserted into $CI_a$ in its correct position as left child of ‘ac’. The status of
FOCIT before processing $T_2$, the intersection tree (ITree) built while processing $T_2$ and the updated FOCIT after processing $T_2$ are shown in Fig. 5.8 (a), Fig. 5.8 (b) and Fig. 5.8 (c) respectively.

When $<T_3, bd>$ arrives, a new intersection tree is created. Initially the new transaction, ‘bd’ is inserted as the root. Then phase 1 and 2 are repeated. In phase 1, the first item in transaction is extracted. It is ‘b’ and since $\text{CI}_b$ is null no more processing is required. Since the second item in the new transaction is ‘d’, the tree $\text{Cl}_d$ need to be traversed since it may have CIs which contain the item ‘d’ in it.

![Fig. 5.8 FOCIT after $T_2$](image)

All other trees with starting item less than ‘d’ are null. On traversing $\text{Cl}_a$, no intersections are obtained with any node. So phase 1 ends and phase 2 begins. In phase 2, the only node in Intersection tree is a new CI and is inserted into FOCIT tree $\text{Cl}_b$ as its root node. The status of FOCIT before processing $T_2$, the intersection tree (ITree) built while processing $T_2$ and the updated FOCIT after processing $T_3$ are shown in Fig. 5.9 (a), Fig. 5.9 (b) and Fig. 5.9 (c) respectively.

When $<T_4, ad>$ arrives the following changes take place. Initially the new transaction, ‘ad’ is inserted as the root into the ITree with support 1. The left, right and address fields of the root are set to null. Then the first item, ‘a’ is extracted from the new transaction and every node in the $\text{Cl}_a$ tree is processed. On intersecting with node ‘ac’, the intersection obtained is ‘a’ and its support is set to one more than the support of node ‘ac’. Thus the support of new intersection ‘a’ is 2. This node is
not present in the intersection tree and so it is inserted after finding its correct location as left child of ‘ad’. On intersecting with node ‘a’ in the tree CIₐ, the intersection obtained is ‘a’ itself, so its address field is set to address of node, ‘a’ in the CIₐ tree and its support is one more than that of node ‘a’ in CIₐ. Thus the support of ‘a’ is 3. There is already a node ‘a’ in the intersection tree. But its support 2 is less than 3, the new support obtained. So the support of node ‘a’ in the intersection tree is updated to 3 and also its address field which is null is updated to address of ‘a’ in CIₐ tree. Now all nodes in CIₐ are processed.

![Fig. 5.9 FOCIT after T₃](image)

The next item ‘d’ is extracted from the new transaction and the tree CIₐ, CIₐ, CIₐ and CIₐ have to be processed. But CIₐ is processed already. The tree CIₐ and CIₐ is null. So only the nodes in CIₐ have to be processed. Only one node, ‘bd’ is present in CIₐ, and on intersecting the new transaction ‘ad’ with it the intersection obtained is ‘d’ with support of 2 (support of ‘bd’+1). Since this is a new intersection, it is inserted into the intersection tree at its correct location as the right child of ‘ad’.

There are no more items to be processed in the new transaction and phase 1 comes to an end. In the phase 2, in-order traversal of the ITree is performed and the nodes are inserted into the FOCIT in the appropriate tree if the address field is null. On performing in-order traversal the first node processed is node ‘a’ whose address field is not null which indicates that the node is already present in the FOCIT in the CIₐ tree. The node is directly accessed and its support is updated to 3. The next node processed is ‘ad’ whose address field is null. The first item in this node tells that the node belongs to the tree CIₐ and it is inserted into the tree after finding its
correct location as the right child of ‘ac’. The next node processed is node with ‘d’, it belongs in the tree Cl\(_d\) and since the tree is null, the node ‘d’ is inserted as the root node. The FOCIT before arrival of T\(_4\), the intersection tree formed and the updated tree after processing T\(_4\) are shown below in Fig. 5.10(a), Fig. 5.10(b) and Fig. 5.10(c) respectively.

![Fig. 5.10 FOCIT after T\(_4\)](image)

Similar processing is done when transaction <T\(_5\), acd> arrives. Initially itemset ‘acd’ is inserted as the root of the intersection tree. First item in the new transaction T\(_5\) is ‘a’. So, tree Cl\(_a\) is traversed fully and the intersections obtained with the three nodes are ‘ac’ with support 2, ‘a’ with support 4 and ‘ad’ with support 2 respectively. These intersections are inserted into the intersection tree in the same order in their correct locations. When the next item ‘c’ in the transaction T\(_5\) is processed the candidate trees to be traversed are Cl\(_a\), Cl\(_b\), Cl\(_c\), and Cl\(_d\). Tree Cl\(_a\) has already been traversed fully and tree Cl\(_c\) is empty. So, the trees to be traversed are Cl\(_b\) and Cl\(_d\). On intersecting ‘acd’ with node ‘bd’ in the tree Cl\(_b\), ‘d’ is obtained and its support is set to 2 (support of ‘bd’ +1). It is inserted into the intersection tree. Next on intersecting ‘acd’ with node ‘d’ in the tree Cl\(_d\), again ‘d’ is obtained and its support is 3 (support of ‘d’ +1). ‘d’ is already in the intersection tree but its support is less than the new support 3. So the support of node ‘d’ in the intersection tree is updated to 3. The process is complete and the intersection tree is used to update the FOCIT. The status of FOCIT before processing T\(_5\), the intersection tree (ITree) built while processing T\(_5\) and the updated FOCIT after processing T\(_5\) are shown in figures Fig. 5.11 (a), Fig. 5.11 (b) and Fig. 5.11 (c) respectively.
When transaction \(<T_6, b>\) arrives, similar processing is done. Initially node ‘b’ is inserted as root node in the intersection tree with support 1. The only item present in this transaction is ‘b’. So the trees CI_a and CI_b have to be traversed. On traversing tree CI_a it is found that the right subtree of the node need not be traversed since the second item in the root node ‘ac’ is ‘c’ which is lexicographically greater than ‘b’ and so the nodes in right subtree will not contain ‘b’. On intersecting with the nodes ‘ac’ and ‘a’ null values are obtained. On intersecting with node ‘bd’ in tree CI_b, ‘b’ is obtained and its support is 2 (support of ‘bd’ +1). This node is already present in the intersection tree. So no new insertion is necessary but the support is updated to 2. The processing is now complete. The initial FOCIT before \(T_6\), the ITree built on processing \(T_6\) and the updated FOCIT after transaction \(<T_6, b>\) are shown in Fig. 5.12 (a), (b) & (c) respectively.

When a user request arrives, the third phase begins and the minimum support, \(minSup\) is calculated as follows:

\[
minSup = s * |D| = 0.5 * 6 = 3.
\]  
\(5.2\)

Now by traversing the trees CI_a, CI_b and CI_d, the only non-empty trees in the FOCIT structure, it is found that ‘a’, and ‘d’ with supports 4 and 3 respectively are the only Closed Frequent Itemsets (CFIs) at this instant since each of their support is greater than or equal to the required minimum support threshold, 3. This list is then returned to the user as the CFIs seen so far in the data stream shown in the illustration.
5.6 MAKING THE SEARCH MORE EFFICIENT

Searching for Closed Itemsets (CIs) having at least one common item with the new transaction is made more efficient in the following ways: If a transaction has an item, ‘i’ then intersection with all nodes in tree CI\(_i\) is performed without any checking. The CIs are clustered in the FOCIT in such a way that all CIs that start with the same item are grouped into a single binary search tree. This means that the tree CI\(_i\) has only closed itemsets that start with item, ‘i’. Search is thus made efficient.

Item, ‘i’ may be in the CI trees that do not start with ‘i’ but start with an item less than ‘i’. So, the CI\(_j\) trees with \(j < i\), that contain closed itemsets that start with ‘j’, should also be searched. This search is made efficient by looking at the second item in each CI. If it is greater than ‘i’, the search need not continue in the right half. CI trees having CIs starting with item, \(k > i\), need not be searched for possible intersections because definitely they will not have ‘i’ in it, since lexicographic order is followed in the closed itemsets.

During the search for intersections, if it is found that the intersection matches with the CI already present in the tree, then its address is stored in the intersection tree. This is used in the update phase and helps to avoid searching the tree again to see if this CI is already in the tree or if it is a new CI. During the search in the CI trees, whenever nodes get visited, they are marked. This helps in avoiding revisiting when seeing other items in the new transaction.
5.7 MAKING THE ALGORITHM MORE EFFICIENT

Only the Closed Itemsets (CIs) are maintained in FOCIT. The number of CIs is very less when compared to the number of frequent itemsets but all information regarding frequent itemsets and their support can be retrieved from the CIs when needed. This reduces the memory requirement to a large extent. Each CI tree is processed if the transaction contains an item, ‘i’ or an item that comes after ‘i’. Only one tree CI needs to be in main memory at a time. Each time phase 1 begins, a new intersection tree is created and its lifetime is till the end of phase 2 and its scope is limited to phases 1 & 2.

Searching for CIs having at least one common item with the new transaction is made more efficient by using the following techniques.

All nodes in the tree CI definitely will have at least one common item with a transaction that contains item, ‘i’. Since tree, CI stores all the closed itemsets that start with item, ‘i’. So, no search is needed here and intersection with all nodes in this tree is calculated.

Item, ‘i’ may be in the CI trees that do not start with ‘i’ but with item j<i. So, the CI trees starting with an item, j<i, should also be searched. This search is made efficient by looking at the second item in each CI. If it is greater than ‘i’, the search need not continue in the right half, since the CI tree is maintained as a Binary Search Tree which maintains the property that CI at parent node is less than CI at right child and greater than CI at left child. For example if the current item being considered for intersections is ‘b’, then CI tree has to be searched since it has closed items starting with ‘a’ but may have item, ‘b’ in some of those closed itemsets.

Consider the CI tree shown in Fig. 5.13. The search for intersections with item ‘b’ has to continue in subtrees T1 and T2 but need not continue in the subtrees T3 and T4 because according to BST property, those subtrees will not have any closed itemset with ‘b’ in it. T3 will have only closed itemsets which are lexically greater than ‘ac’ and these itemsets do not include ‘b’ in them. T4 will have closed itemsets that are greater than ‘ad’ and this does not include an itemset with ‘b’ in it. This is so because the items within an itemset are maintained in lexical order. So if ‘b’ occurs in an itemset it will definitely be before ‘c’ and not after it.
The trees having CIs starting with item, k>i, need not be searched for possible intersections because they will not have ‘i’ in it, since lexicographic order is followed. For example if the item being considered for intersections is ‘d’ then only trees CI_a, CI_b, CF_c, CI_d may have closed itemsets with ‘d’ in it and the remaining trees CI_e, CI_f, CF_g, and so on need not be searched because they will have only closed itemsets that start with ‘e’, ‘f’, and ‘g’ respectively and these will not have ‘d’ in it since lexicographically becomes before ‘e’, ‘f’, or ‘g’.

During the intersection tree construction process, if the intersection found matches with the Closed Itemset (CI) already present in one of the trees in the FOCIT, then its address is stored in the intersection tree. This information is used in the update phase to avoid searching the trees in the FOCIT repeatedly to find the correct location. This search is needed to see if the current CI is already in the tree or if it is a new CI. If the node in the intersection tree has a non-null address, it indicates that the corresponding CI is already in the CI tree without the need for a search in the tree.

During the search in the CI trees, whenever nodes get visited, they are marked. This helps in avoiding revisiting them when subsequent items in the new transaction are processed. While processing the current item, i the tree CI_i is fully-traversed and all trees CI_j with j<i, are partially-traversed. Fully-traversed trees are specially marked to indicate that they need not be traversed when considering subsequent items in the transaction and for the other trees which are partially traversed the nodes visited are marked. Any future traversals of fully-traversed trees can thus be avoided since already all nodes are visited and the intersections
calculated. The partially-traversed trees need to be re-traversed but only nodes not yet visited need to be considered. For this, the marking of individual nodes is essential and proves to be time saving.

5.8 PERFORMANCE EVALUATION

In this section, the experimental evaluation of the proposed algorithm, FOCIT is described. The programs were implemented in Java NetBeans version 6.8 and executed on a system with the following configuration: Intel® Core™2 Duo CPU, E7500 @2.93 GHz, 1.98 GB RAM and 250 GB Hard Disk running on Windows XP. For testing closed frequent itemset mining over the data streams, the Mushroom dataset and Chess dataset downloaded from http://fimi.ua.ac.be/data website were used. They are real world datasets used to evaluate many algorithms in the history of data stream mining. Table 5.2 lists the characteristics of these datasets [66]. To simulate data streams, the transactions were stored in a file and they were read one by one and used to update the forest of CI trees.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>#Items</th>
<th>Avg. Length</th>
<th>#Transactions</th>
<th>Type</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mushroom</td>
<td>117</td>
<td>23</td>
<td>8124</td>
<td>Dense</td>
<td>557 KB</td>
</tr>
<tr>
<td>Chess</td>
<td>74</td>
<td>37</td>
<td>3196</td>
<td>Dense</td>
<td>415 KB</td>
</tr>
<tr>
<td>Retail</td>
<td>1000</td>
<td>14</td>
<td>88165</td>
<td>Sparse</td>
<td>2.18 MB</td>
</tr>
</tbody>
</table>

The proposed algorithm, FOCIT was compared with CloStream algorithm [97] and CL-Stream (Concept Lattice-Stream) algorithm [84] as both generate CFIs from data streams. Fig. 5.14 shows the processing time of the three algorithms for every 100 transactions (100 transaction updates). It very clearly shows that the performance of the proposed algorithm, FOCIT is definitely better than both CloStream and CL-Stream algorithms. The main reason is that the organization of the closed itemsets as binary search trees in the proposed algorithm helps in reducing the search time during the updates. Various search techniques are also employed to further reduce the search time as explained in the previous section.
But CL-Stream maintains a closed itemset list, Clist, and for each new transaction examines every CI in this list, computes intersection and keeps track of the smallest CI that produced this intersection. A separate Clist is also maintained by CL-Stream algorithm which is maintained for each item. It lists the CIs in which this item is present. All this takes more time than the process involved in the proposed algorithm. In CloStream algorithm [97] a table is used to maintain the information about closed itemsets. In addition a Cid list is maintained to keep track of the super closed itemsets of every CI. Maintenance of this list in addition to the other processes involved makes CloStream more time consuming than the proposed FOCIT algorithm.

![Fig. 5.14 Execution Time](image)

Fig. 5.14 Execution Time

Fig. 5.15 shows the time needed for updating the forest of CI trees measured for every 100 transactions arriving on the stream. Mushroom dataset is used for data stream simulation in these measurements. The times taken by the algorithms do not show much variation for the first few hundreds of transactions.

But as more transactions keep on arriving, there is a marked difference in the time measurements for the proposed FOCIT algorithm and the CL-Stream and CloStream algorithms. This is because CL-Stream algorithm needs to maintain a special list to identify the CIs that contain an item and CloStream needs to maintain a list containing the supersets of CIs. The size of these lists and their maintenance cost keeps increasing as time progresses but the proposed algorithm does not maintain
any such list. So as time progresses, the time needed for updating the CIs is lower for the proposed FOCIT algorithm and higher for CL-Stream and CloStream algorithms.

Fig. 5.15 Time for Updating CIs

Fig. 5.16 shows the variation in execution time for different support thresholds on Chess dataset which is a dense dataset with 74 distinct items. It is expected that the number of CIs will decrease as the support is increased and so the time for generating these CIs will also decrease. This behavior is exhibited by all three algorithms but the rate of decrease is higher for the proposed FOCIT algorithm than the other algorithms. This clearly shows that the proposed algorithm, FOCIT performs better than the other two for all threshold values and is definitely better than the other two algorithms taken for comparison purpose.

Fig. 5.16 Execution Time Vs Threshold – Chess Dataset

The same measurements are repeated using the Mushroom dataset which is also dense but the number of unique items is 117. Fig. 5.17 shows that the
proposed algorithm, FOCIT outperforms the other two algorithms for this dataset also. This shows that even when the number of unique items in a data stream increase, the performance of the proposed FOCIT algorithm is better than the algorithms taken for comparison purpose.

![Fig. 5.17 Execution Time Vs Threshold – Mushroom Dataset](image)

On repeating the experiment with Retail dataset which is a sparse dataset and has 1000 unique items, it was found that the performance of the proposed FOCIT algorithm is marginally less than the CloStream but still better than the CL-Stream algorithm. The results obtained are illustrated in Fig. 5.18. These evaluations show that the proposed algorithm, FOCIT is most effective on dense datasets and can also handle sparse datasets effectively irrespective of the number of unique items in the data streams.

![Fig. 5.18 Execution Time Vs Threshold – Retail Dataset](image)
5.9 CASE STUDY

With the explosive growth of Internet commerce, the Internet has evolved into a gold mine that contains or dynamically generates information that is beneficial to e-businesses. It is a revolution that the Internet has grown from a simple search tool to a gold mine. Companies doing e-businesses have to implement Web mining systems to understand their customers' profiles, and to identify their own strength and weaknesses of their e-marketing efforts on the web. This may aid them to achieve continuous improvements. Internet is a gold mine, but only for those companies who realize the importance of Web mining and adopt a Web mining strategy. Today a company's web site is the most direct link it has to its current and potential customers. The companies can study the various activities of their clients through web analysis, and find the patterns in the clients' behavior. The web analysis yields rich results that when coupled with the information stored in company data warehouses, offer great opportunities for business improvement in the near future.

Web Usage Mining is an important part of web analysis and it can be described as the discovery and analysis of user access patterns, through the mining of log files and associated data from a particular Web site [102]. With the explosion of E-commerce, the way companies are doing businesses has changed. E-commerce, mainly characterized by electronic transactions through Internet, has provided a cost efficient and effective way of doing business. The growth of some e-businesses is astonishing. The fact that it has made Amazon.com the “on-line Wal-Mart” proves this.

Unfortunately, many companies consider web just as a place where their electronic transactions take place and do not give importance to its impact on their business activities. They do not realize that as millions of visitors interact daily with Web sites around the world, massive amounts of data are being generated. And they also do not realize that this information could be very precious to the company in the fields of understanding customer behavior, improving customer services and relationship, launching target marketing campaigns, measuring the success of marketing efforts, and so on. Web server log files are a huge repository of such data which help to trace the visitors’ on-line behaviors. For example, after some basic traffic analysis, the log files can help to identify the search engines frequently used...
by the visitors to the website and the pages that are the most and least popular in a particular website.

When path analysis is used on the site as a whole, the extracted information can offer valuable insights about navigational problems. Examples of information that can be discovered through path analysis are:

- 78% of clients placed orders through `/company/products/order.asp` by starting at `/company` and proceeding through `/company/whatsnew.html`, and `/company/products/sample.html`

- 60% of clients left the site after at the most four page references

With the first rule it can be inferred that 78% of visitors decided to make a purchase after seeing the sample of the products. Such information motivates the management in making decisions to concentrate more on the layout and information included in the ‘sample.html’ file. The second rule indicates an attrition rate for the site. Since many users do not browse further than four pages into the site, it shall be beneficial to the organization to ensure that most important information (product sample, for example) is contained within four pages of the common site entry points. Consequently, tracking online behavior of clients is no longer a minor issue but a major concern for the entire organization.

Sequential patterns discovery deals with finding the inter-transaction patterns such that the presence of a set of items is followed by another item in the time-stamp ordered transaction set. Web log files can record a set of transactions in time sequence. If the web-based companies can discover the sequential patterns of the visitors, the companies can predict users’ visit patterns and target market on a group of users. For example sequential patterns like 50% of clients who bought items in `/pcworld/computers/`, also placed an order online in `/pcworld/accessories/` within 15 days in an online shopping website may help the management to go for targeted marketing on such clients.

Web mining can be extended further by combining other corporate information with Web traffic data. This allows accounting, customer profile, inventory, and demographic information to be correlated with Web browsing. This information may be used to find out how many people visiting a company’s web site
purchased something, which advertising campaigns resulted in the most purchases and whether the company’s Web visitors fit a certain profile.

Practical applications of Web mining technology are abundant. Web mining can provide companies managerial insight into visitor profiles, which help top management take strategic actions accordingly. Also, the company can obtain some subjective measurements through Web Mining on the effectiveness of their marketing campaign or marketing research, which will help the business to improve and align their marketing strategies timely. For example, the company may have a list of goals including increasing average page views per session, increasing average profit per checkout, decreasing products returned, increasing number of referred customers, increasing brand awareness, increasing retention rate (such as number of visitors who have returned within 30 days), reducing clicks-to-close (average page views to accomplish a purchase or obtain desired information), increasing conversion rate (checkouts per visit).

The company can identify the strength and weakness of its web marketing campaign through Web Mining, and then make strategic adjustments, obtain the feedback through Web Mining again to see the improvement. This procedure is an on-going continuous process. The company normally wants to study the behavior of its clients from the time when an important web marketing campaign was launched till the current time. The client information keeps flowing in at a fast rate and landmark window models are best suited to study such client behavior. The launch time of the campaign may be set as the landmark and the clients’ behavior after the launch and till the current time can be collected and maintained in the landmark window for further analysis. Since the volume of incoming data is very vast, the sequential patterns, identified by analyzing the window content, are also very large in number. This leads to the necessity of using an alternate compressed representation of the patterns identified. The Closed Frequent Itemsets representation of the frequent itemsets used in the proposed algorithm, FOCIT and the landmark window model employed in this algorithm are well suited for such analysis required by the companies that concentrate on e-businesses.

In a landmark window all transactions that arrive after the specified time, called the landmark, have to be maintained and they cannot be discarded even if the
number of transactions in the window becomes very high. Size of the data stream is unbounded. So the size of the window keeps increasing as time progresses. In the proposed FOCIT algorithm, the storage needed for the Closed Itemsets (CIs) keeps increasing even though the transactions themselves need not be stored. First the CIs affected by the new transaction are identified and updated in the Forest of Closed Itemset Trees (FOCIT). Then any new CIs created are inserted into the appropriate CI trees in the forest. After these two steps the new transaction may be discarded safely. The time needed for these processes increases as time progresses and more transactions keep arriving. Also it takes a prohibitively long time to generate the Closed Frequent Itemsets from the FOCIT whenever a user request arrives. Because of the above said drawbacks, landmark windows are preferred only when the application absolutely demands it.