Chapter 4

Incremental Algorithm for Discovery of Closed Itemsets

This chapter describes a lattice based algorithm for discovery of closed itemsets from a data stream. The algorithm uses a novel data structure, CILattice, for storing the discovered closed itemsets.

Section 4.1 discusses various issues related to frequent closed itemset mining in data streams and section 4.2 presents the work done in this area. Contributions of this chapter are summarized in section 4.1.1. Section 4.3 describes the proposed novel data structure CILattice. Procedures of updation of CILattice are explained in detail in section 4.4. Section 4.5 presents the CLICI algorithm for data streams. The experimental results are described in section 4.6 and conclusion is presented in section 4.7.

4.1 Introduction

Frequent closed itemset (FCI) mining has been well studied in the literature [Han et al., 2007, Yahia et al., 2006]. Several algorithms like Closet [Pei et al., 2000], CHARM [Zaki and Hsiao, 2002], Closet+ [Wang et al., 2003], CHARM – L [Zaki and Hsiao, 2005], FP – Close [Grahne and Zhu, 2003], DCI – Closed [Lucchese et al., 2006] has been proposed to generate FCIs in static datasets. Re-
Recently, more and more applications such as network traffic analysis, web log and click stream mining, trend analysis and fraud detection in telecommunications data, e-business and stock market analysis and sensor networks have emerged that generate a large amount of high speed streaming data [Cheng et al., 2008, Aggarwal, 2007]. Mining of closed itemsets from data streams can capture interesting trends and patterns.

Formally, a data stream is a sequence of incoming transactions, $D_N = \{t_1, t_2, ..., t_i, ..., t_N\}$ where $D_N$ represents the data stream with $N$ transactions arrived so far and $t_i$ represents the $i^{th}$ arrived transaction [Cheng et al., 2008, Aggarwal, 2007]. A window is defined as an excerpt of the stream and is a subsequence between $i^{th}$ and $j^{th}$ arrived transactions, denoted as $W[i, j] = (t_i, ..., t_j), i \leq j$ [Cheng et al., 2008, Aggarwal, 2007].

Processing of large volume of data in a data stream poses the constraints of space for storage and time for computation [Cheng et al., 2008, Aggarwal, 2007]. Thus, results of data mining techniques like clustering, classification and association rule mining on stream are stored in synopsis data structure which provides efficient storage and aids fast processing. As the size of the synopsis increases, both memory and per-transaction processing time increase monotonically. Consequently, it becomes increasingly hard to process a transaction before new transactions come in and restrict memory requirement [Cheng et al., 2008, Aggarwal, 2007].

Different window models have been proposed in data streams for processing incoming transactions. Sliding window model constrains the size of the synopsis by fixing the number of transactions in the window. In this model, when new transactions arrive, old transactions are removed from the window keeping the number of transactions in the window fixed. Thus $W$ is a sliding window if $W = \{t_{r-w+1}, ..., t_r\}$, where $t_{r-w+1}$ and $t_r$ are the oldest and the current transactions respectively, and $w$ is the size of the sliding window. Sliding window model has the advantage of having nearly constant per transaction processing time and limited memory requirement, however the model falls short of monitoring the continuous
variation of data stream [Cheng et al., 2008, Aggarwal, 2007].

Landmark window model considers entire data starting from a particular landmark to the current time [Liu et al., 2009]. In landmark window model, all transactions starting from the set landmark to the current time are considered for processing. Thus $W$ is a landmark window if $W = \{t_i, t_{i+1}, ..., t_{i+r}\}$, where $i \geq 1$ and is known as the landmark, and $t_{i+r}$ is the just arrived transaction. Entire data stream is treated as a special case of landmark window where $i = 1$. Although this model enables monitoring of gradual changes in the data stream, capturing recent data and keeping the size of synopsis under control are challenging tasks.

Damped window model, a variant of landmark window model, considers entire data starting from a particular landmark to the current time [Chang and Lee, 2003] and mines recent data by fading the older transactions with time, thus presenting recent results only.

The problem of mining data streams is more challenging than mining static datasets in view of the following aspects [Cheng et al., 2008, Aggarwal, 2007]

1. Stream is a continuous flow of data and hence data must be processed at a rate faster than its arrival.

2. Each element of stream must be examined only once.

3. Memory usage should be bounded even though the stream is potentially unbounded.

4. Results should be instantly available in real time and error in the results, if any, should be bounded.

5. Since stream evolves with time, the patterns captured should be recent.

Although several algorithms have been proposed for frequent itemset mining in data streams [Cheng et al., 2008], mining frequent closed itemsets (FCI) in data stream throws newer challenges. Since every transaction in a data stream is a closed itemset [Valtchev et al., 2008], it leads to addition of at least one entity in the synopsis.
Several algorithms for mining frequent closed itemsets (FCI) in sliding window model [Chen and Li, 2007, Jiang and Gruenwald, 2006, Li et al., 2009, Chi et al., 2006] have been proposed. Algorithm proposed by [Liu et al., 2009] mines FCIs in landmark window model. Although algorithms for mining frequent itemsets have been proposed for damped window model [Chang and Lee, 2003], not much attention has been paid to mining of closed itemsets in damped window model.

4.1.1 Our Contribution

In this chapter, we propose an algorithm, \textit{CLICI}, for discovery of all closed itemsets in sliding window model and also present a variant which discovers all recent closed itemsets in damped window model of data streams. \textit{CLICI} has an online component that processes the transactions without candidate generation and stores the results in a synopsis. Offline component is invoked on demand and mines the closed itemsets from the synopsis based on dynamically specified support threshold. A novel data structure, \textit{CILattice}, is proposed for storing closed itemsets and their support. The salient features of the proposed algorithm \textit{CLICI} are listed below:

1. The algorithm performs single scan of data.

2. The algorithm processes transactions without any candidate generation and mines closed itemsets in two different window models.

3. The algorithm is based on sound mathematical foundation of Formal Concept Analysis and stores the closed itemsets in a lattice based synopsis, \textit{CILattice}. \textit{CILattice} stores only the closed itemsets, unlike concept lattice where set of transactions are stored additionally. Following advantages accrue due to use of lattice based synopsis.

   (a) Support of discovered closed itemsets can be easily derived from the synopsis.
(b) Non-redundant association rules can be directly derived from the synopsis 5.

4. CILattice maintains all closed itemsets, thereby providing a valuable facility for experimentation with varying support threshold without any overhead.

4.2 Related Work

Several algorithms like Moment [Chi et al., 2006], NewMoment [Li et al., 2009], CFI–Stream [Jiang and Gruenwald, 2006], GC–Tree [Chen and Li, 2007], Moment+ [Li and Chen, 2008] have been proposed for mining of frequent closed itemsets (FCI) in sliding window model of data streams. Researchers working in the area of Formal Concept Analysis (FCA) have contributed algorithms like [Godin et al., 1995b], [Nourine and Raynaud, 2002], addintent [van der Merwe et al., 2004] and [Valtchev et al., 2008] for generation of concepts in incremental datasets. Since intent of a concept is same as closed itemset [Stumme et al., 2002, Zaki, 2000], all algorithms for generating concepts can be considered as algorithms for generating closed itemsets in incremental datasets / data stream. In the following section, we analyze various algorithms for mining FCI in data streams.

4.2.1 Closed Itemset Mining in Data Streams

Moment [Chi et al., 2006], CFI–Stream [Jiang and Gruenwald, 2006], NewMoment [Li et al., 2009], GC–Tree [Chen and Li, 2007], Moment+ [Li and Chen, 2008] are some of the recent algorithms which generate closed itemsets in sliding window model of data streams. These algorithms store a set of current transactions in a memory based window. On arrival of a new transaction, these algorithms determine the new closed itemsets generated and modifies the synopsis by storing the newly generated closed itemsets. NewMoment stores bitwise representation of the associated transactions along with 1-itemsets to find the support of discovered itemsets while rest of the algorithms scan the window to find the current support
of the discovered itemsets [Li et al., 2009].

\textit{FP – CDS} algorithm discovers frequent closed itemsets in landmark window model [Liu et al., 2009]. The algorithm works in a batch mode, dividing the landmark window into several 'basic' windows, using them as updating units. For each basic window, subfrequent closed itemsets, closed itemsets with support slightly less than the support threshold, are maintained.

In damped window model, \textit{estDec} algorithm has been proposed for discovery of recent frequent itemsets [Chang and Lee, 2003]. Older transactions are decayed and later pruned to maintain the recency of the discovered itemsets. Itemsets having support greater than the pre-defined support threshold are maintained.

\section*{4.2.2 FCA Based Incremental Algorithms}

Algorithms generating concepts incrementally like [Godin et al., 1995b], [Nourine and Raynaud, 2002], [Valtchev et al., 2008], and [van der Merwe et al., 2004], generate closed itemsets in incremental datasets. The algorithms compute intersection of the extent (set of transactions) of old concepts with the new transaction, thus discovering new concepts and modifying the old ones. However these algorithms lack scalability because in each of these algorithms, a concept stores the associated set of transactions (extent) along with the set of attributes (intent). Data mining applications are interested in the cardinality of the extent (support) rather than the extent itself.

\textit{Galicia} algorithm generates the concepts incrementally in large datasets as it does not store set of transactions in the concept but it does not maintain the lattice [Valtchev et al., 2008]. Hence it cannot be used for generation of non-redundant association rules.

\section*{4.2.3 Comparative Analysis}

The algorithms based on FCA have the advantage that the lattice structure enables the algorithm to generate non-redundant association rules directly from the
lattice [Zaki, 2000]. FCA based algorithms are naturally on-line in the sense that insertion of transactions in the concept lattice doesn’t need a dataset scan. Major disadvantage of FCA based algorithms is the need to store the extent (set of transactions) with each node and the computational expense required for computing the intersection of extents of different nodes.

Conventional closed itemset mining algorithms for data streams do not store extent with each node but they need to scan the window repeatedly. Hence algorithms designed for sliding window model like Moment [Chi et al., 2006], CFI – Stream [Jiang and Gruenwald, 2006], NewMoment [Li et al., 2009], GC – Tree [Chen and Li, 2007], Moment+ [Li and Chen, 2008] are not suitable for landmark or damped window model. Further, data mining algorithms do not facilitate derivation of non-redundant association rules directly from the synopsis.

4.3 CILattice: Data structure for storing Closed Itemsets

We propose a novel lattice based synopsis called CILattice for storing the closed itemsets. CILattice has two components:

1. A lattice \( \mathcal{L} \): \( \mathcal{L} \) is a complete lattice, with topnode \( \top \) and bottom node \( \bot \). A node \( \mathcal{N} \) of \( \mathcal{L} \) comprises of following components:
   (a) a closed itemset \( I_{\mathcal{N}} \)
   (b) frequency \( f_{\mathcal{N}} \)
   (c) links to its parents
   (d) links to its child nodes

2. An accompanying header table \( \text{Itable} \): It is a two-dimensional array storing items and pointers to the nodes corresponding to first occurrence of that item in \( \mathcal{L} \), which aids efficient traversal during search and insert procedure.
A node $N$ is represented as $< I_N, f_N >$.

**Example 4.1** Fig. 4.1 shows a toy dataset $D_3$ and the corresponding $< \mathcal{L}, \text{Itable} >$.

<table>
<thead>
<tr>
<th>TID</th>
<th>Items</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>abc</td>
</tr>
<tr>
<td>2</td>
<td>ad</td>
</tr>
<tr>
<td>3</td>
<td>be</td>
</tr>
<tr>
<td>4</td>
<td>abde</td>
</tr>
<tr>
<td>5</td>
<td>acd</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>item</th>
<th>$f_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
</tr>
<tr>
<td>d</td>
<td></td>
</tr>
<tr>
<td>e</td>
<td></td>
</tr>
</tbody>
</table>

Figure 4.1: (a) Toy Dataset $D_3$ (b) Itable (c) $\mathcal{L}$

### 4.3.1 Notations and Definitions

Definitions and observations used later in the algorithm are given below:

Let $A(N)$, $D(N)$, $P(N)$ and $C(N)$ denote the set of ancestors, descendants, parents and children respectively of the node $N$ in $\mathcal{L}$.

**Definition 4.1** A node $X$ is ancestor of node $Y$ iff $I_X \subseteq I_Y (I_X \neq I_Y)$.

**Definition 4.2** A node $X$ is descendant of node $Y$ iff $I_X \supset I_Y (I_X \neq I_Y)$.
Table 4.1: Sets of Ancestors, Descendants, Parents and Children of Node $X$

<table>
<thead>
<tr>
<th>Set</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A(X)$</td>
<td>$({a}, 4), ({b}, 3), (\emptyset, 5)$</td>
</tr>
<tr>
<td>$D(X)$</td>
<td>$({abc}, 1), ({abde}, 1), ({abcde}, 0)$</td>
</tr>
<tr>
<td>$P(X)$</td>
<td>$({a}, 4), ({b}, 3)$</td>
</tr>
<tr>
<td>$C(X)$</td>
<td>$({abc}, 1), ({abde}, 1)$</td>
</tr>
</tbody>
</table>

**Definition 4.3** A node $X$ is parent of a node $Y$ if $X \in A(Y)$ and $\not\exists$ any $Z \in A(Y) : X \in A(Z)$ and $Z \neq X$.

**Definition 4.4** A node $X$ is a child of node $Y$ if $X \in D(Y)$ and $\not\exists$ any $Z \in D(Y) : X \in D(Z)$ and $Z \neq X$.

It is obvious from the above definitions that ancestor nodes are generalizations of the descendant nodes. Further, parents of a node are its immediate ancestors (generalizations) and children of a node are its immediate descendants (specializations).

**Observation 4.1** If node $X$ has a child node $Y$ in $\mathcal{L}$ then closed itemset of $Y$ is minimal superset of all descendants of $X$. Similarly closed itemset of $X$ is maximal subset of all ancestors of $Y$.

**Definition 4.5** First Node $F_i$ of an item $i$ is a node $X$ in the $\mathcal{L}$ where $i \in I_X$ and $\not\exists$ a node $Y$ such that $i \in I_Y$ and $Y \in A(X)$.

Naturally there is exactly one $F_i$ for each item $i \in I$ and $\text{Itable}$ stores the pair $(i, F_i)$. For example, $F_a = (\{a\}, 4)$ and $F_d = (\{ad\}, 3)$.

Table 4.1 represents the set of ancestors $A(X)$, descendants $D(X)$, parents $P(X)$ and children $C(X)$ for the node $X(\{ab\}, 2)$ marked in Fig. 4.1.

### 4.4 Updating $CILattice$

$CILattice$ synopsis is updated whenever a new transaction is inserted or an old transaction is removed from the data stream. We describe below the procedures
developed for traversal of the lattice $L$, searching a transaction, inserting a transaction and deleting a node in $CILattice$. Procedure for updation of $ITable$ is also described.

4.4.1 Procedure: Traverse

$Traverse$ visits each node $X$ of the lattice $L$ of $CILattice$. Breadth first traversal mechanism is used for traversing $L$. A queue of unprocessed nodes is maintained. Initially top node is inserted in the queue. Each node in the queue is visited to output the closed itemset and its support. Whenever a node is visited, its child nodes are added in the queue. Traversal stops when all nodes in the queue are visited. The procedure for traversing $CILattice$ is described in Algorithm 4.1.

<table>
<thead>
<tr>
<th>Input: $L$ - lattice of closed itemsets</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output: None</td>
</tr>
<tr>
<td>Process:</td>
</tr>
<tr>
<td>Create an empty queue $Q$</td>
</tr>
<tr>
<td>enQueue top node of $L$</td>
</tr>
<tr>
<td>while $Q$ is not empty do</td>
</tr>
<tr>
<td>deQue a node $X$ from $Q$</td>
</tr>
<tr>
<td>visit($X$) {outputs the closed itemset and its support}</td>
</tr>
<tr>
<td>for all child nodes $Y$ in $C(X)$ do</td>
</tr>
<tr>
<td>enqueue($Y$)</td>
</tr>
<tr>
<td>for end</td>
</tr>
<tr>
<td>end while</td>
</tr>
</tbody>
</table>

**Algorithm 4.1: Procedure Traverse**

4.4.2 Procedure: Search

$Search$ procedure checks whether a transaction $t$ exists as a node in $L$. The brute force approach of searching $L$ for a node containing $t$ would be to start the search
either from $\top$ or $\bot$. This approach is computationally expensive as the lattice has a tendency to grow in size as new transactions arrive. Proposition 4.1 enables an improvement by searching \textit{First Node} corresponding to any one of the items in the transaction and its descendants.

\textbf{Proposition 4.1} Let $t$ be the transaction to be searched in $\mathcal{L}$ and let $i \in t$ be one of the items in $t$ chosen arbitrarily. It is necessary and sufficient to search among $F_i$ and its descendants.

\textbf{Proof 4.1} If $t = I_{F_i}$ for an arbitrarily chosen $i$, $i \in t$, then $t \in \mathcal{L}$.

By definition 4.2 and 4.5, $i \in I_{F_i}, \forall i \in t$. $D(F_i)$ is a set containing all supersets of $I_{F_i}$. If $t \in \mathcal{L}$ then $t$ is a superset of $I_{F_i}$ and hence $t$ exists as a descendant of $F_i$. Thus $t$ is necessarily a descendant of each of the items contained in $t$.

By definition 4.2 and 4.5, no other node except nodes belonging to $D(F_i)$ contain $i$. Since $i \in t$ so if $t \in \mathcal{L}$ then $t$ is among the descendants of $F_i$. Hence it is sufficient to search among descendants of $F_i$.

Thus, if $t$ contains an item $i$ for which $F_i$ does not exist, then $t \notin \mathcal{L}$. Otherwise it needs to be searched among $F_i$ and its descendants corresponding to any one of the arbitrarily chosen item $i$ in $t$.

If $t$ exists as a closed itemset of a node $X$ in $\mathcal{L}$, Search procedure returns a pointer to the node $X$, otherwise pointer to a generator node (node whose closed itemset is a superset of $t$) of $t$ is returned.

\textbf{4.4.3 Procedure: Insert}

[Valtchev et al., 2008] stated that each incoming transaction $t$ is a closed itemset. We provide a formal proof in Proposition 4.2. In case transaction $t$ already exists in $\mathcal{L}$, then the frequency of the node corresponding to $t$ and its ancestor nodes is incremented by one. Otherwise processing of $t$ results into insertion of one or more nodes in $\mathcal{L}$. Proposition 4.2 establishes this fact.
Input: $\mathcal{L}$ - lattice of closed itemsets
$t$ - transaction to be searched

Output: i) pointer to node if $t$ exists in $\mathcal{L}$
or pointer to generator of $t$ if $t$ does not exist in $\mathcal{L}$
ii) variable status which is true if $t$ exists in $\mathcal{L}$ else is false

Process:

if $F_i == \text{NULL}$ for an item $i \in t$ then
    return NULL pointer and set status to false
end if

create an empty queue $Q$

for all arbitrarily chosen $i \in t$ do
    enQueue($F_i$)
end for

while $Q$ is not empty do
    deQue a node $X$ from $Q$
    Compare $t$ with closed itemset of $X$
    if $t$ is same as closed itemset of $X$ then
        return pointer to $X$ and set status to true
    else
        if $t$ is superset of closed itemset of $X$ then
            enQueue all child nodes of temp in $Q$
        else
            if $t$ is subset of closed itemset of $X$ then
                return pointer to $X$ and set status to false
            end if
        end if
    end if
end while

Algorithm 4.2: Procedure Search
Proposition 4.2  Every transaction \( t \) is a closed itemset.

Proof 4.2  If \( t \) is the new transaction to insert then either \( t \in \mathcal{L} \) or \( t \notin \mathcal{L} \). If \( t \in \mathcal{L} \) then obviously \( t \) is closed. If \( t \notin \mathcal{L} \), then either \( t \) does not have any superset or some supersets of \( t \) exist. If none of the superset of \( t \) exists then trivially \( t \) is closed. If some supersets of \( t \) exist then support of \( t \) becomes one more than the maximum support of its supersets, implying that \( t \) is closed.

Thus, if transaction \( t \) does not already appear as a node in \( \mathcal{L} \) then it is added as a new node \( \mathcal{N} \) in \( \mathcal{L} \), with \( I_\mathcal{N} = t \). Insert Procedure determines the minimal superset of \( I_\mathcal{N} \) to find the child nodes and the maximal subsets of \( I_\mathcal{N} \) to find the parents of \( \mathcal{N} \) (Observation 4.1). Proposition 4.3 states that Node \( \mathcal{N} \) will have exactly one child in \( \mathcal{L} \).

Proposition 4.3  If \( \mathcal{N} \) is the node corresponding to transaction \( t \) that has just been added to lattice \( \mathcal{L} \), then \( |C(\mathcal{N})| = 1 \).

Proof 4.3  We prove this assertion by contradiction. Let, if possible, \( |C(\mathcal{N})| = m > 1 \). Let \( C(\mathcal{N}) = C_1, C_2, \ldots, C_m \) be the set of children of \( \mathcal{N} \) in \( \mathcal{L} \). By definition, \( I_\mathcal{N} = I_{C_1} \cap I_{C_2} \ldots \cap I_{C_m} \). According to theorem 2.1, intersection of extents of nodes in lattice is always an extent of a node in lattice i.e. there must exist a node \( X \) in \( \mathcal{L} \) such that \( I_X = I_{C_1} \cap I_{C_2} \ldots \cap I_{C_m} \). But then \( X \) and \( \mathcal{N} \) are the same nodes. This contradicts the fact that \( \mathcal{N} \) is a new node. Hence the assumption that \( |C(\mathcal{N})| = m > 1 \) is not correct.

Further \( m \) cannot be zero because bottom node \( \bot \) of \( \mathcal{L} \) contains all the items and is always a superset of transaction \( t \). Hence \( m = 1 \).

Insert traverses \( \mathcal{L} \) using \( ITable \) for finding the child node and parent nodes of \( \mathcal{N} \). The use of Proposition 4.3 reduces the effort involved in searching child nodes of \( \mathcal{N} \) as it can have only one child in \( \mathcal{L} \). However, potential number of ancestors can be \( 2^{|I_\mathcal{N}|} \). The exact number of immediate ancestors i.e. parents is unpredictable. This process of searching immediate ancestors i.e. parents of \( \mathcal{N} \) is
speeded up using $F_i$ of $Itable$, $i \in t$. $F_i$ provides the entry point for search in $L$. The following three cases arise:

Case 1: $F_i$ is ancestor of $N$ i.e. $I_{F_i} \subset I_N$.

Child of $N$ is among the descendants of $F_i$. Parent(s) of $N$ is either the node $F_i$ itself or some of $F_i$’s descendant(s). If closed itemset of any of the child node $X$ of $F_i$ is superset of $I_N$ then $F_i$ is the parent of $N$ and $X$ is the child of $N$. Otherwise we search amongst the descendants of $F_i$ till we find a node $X$ such that $I_X$ is superset of $I_N$ and closed itemset of one of the parent $Y$ of $X$ is subset of $I_N$. In that case, $X$ is the child of $N$ and $Y$ is the parent of $N$.

Case 2: $F_i$ is descendant of $N$ i.e. $I_{F_i} \supset I_N$.

$F_i$ is the child of $N$ and First Node $F_i$ corresponding to $i \in I_N$ is set to $N$. We search among the parents of $F_i$ for finding parents of $N$. If closed itemset of those parents is subset of $I_N$, then those parents are parents of $N$ also. If no such parent exist then top node $\top$ of $L$ is the parent of $N$.

Case 3: $F_i$ is neither ancestor nor descendant of $N$ i.e. $I_{F_i} \nsubseteq I_N$, $I_{F_i} \nsubseteq I_N$ and $I_{F_i} \cap I_N \neq \phi$.

In this case, common parent of $F_i$ and $N$ with closed itemset as $I_{F_i} \cap I_N$ may exist in $L$ or it may not exist.

Case 3.1: $I_{F_i} \cap I_N$ exists as closed itemset of a node $X$ in $L$.

If such a node $X$ exists in $L$ then $X$ is the parent of $N$. Child of $N$ is among the descendants of $F_i$. We check all the descendants of $F_i$ till we get a node whose closed itemset is minimal superset of closed itemset of $N$, that node becomes the child of $N$.

Case 3.2: $I_{F_i} \cap I_N$ does not exist as closed itemset of a node in $L$.  

64
A new node $X$ is created with $I_X = I_{F_i} \cap I_N$ and added to $\mathcal{L}$. $X$ becomes the parent of $\mathcal{N}$ and child of $\mathcal{N}$ is among the descendants of $F_i$. We check all the descendants of $F_i$ till we get a node whose closed itemset is minimal superset of closed itemset of $\mathcal{N}$, that node becomes the child of $\mathcal{N}$. Next step is to find the child and parents of $X$. Parents of $X$ are among ancestors of $F_i$. We check all the ancestors of $F_i$. If we get a node $Y$ such that closed itemset of $Y$ is superset of $X$ and closed itemset of one of the parent $Z$ of $Y$ is subset of $X$, then $Z$ becomes the parent of $X$. If we find an ancestor $Y$ of $F_i$ such that $Y$ is neither ancestor nor descendant of $X$, then a new node $U$ is created with $I_U = I_Y \cap I_X$. Then we repeat the process to find child and parents of node $U$.

The above discussion is formalized in algorithm 4.3.

**Example 4.2** Fig. 4.2 shows the updated CILattice after inserting the transaction $ae$.

**4.4.4 Procedure Delete**

*Delete* removes a node $X$ from $\mathcal{L}$. When a node $X$ is removed from $\mathcal{L}$, it is removed from the parents list of its child nodes and children list of its parent nodes. When $X$ is removed from the parents list of its child nodes, two cases arise:

**Case 1: Child nodes of $X$ have more than one parent** In that case, $X$ is removed from the parents list of child nodes of $X$.

**Case 2: Child nodes of $X$ have only one parent i.e. $X$ itself** In that case, all parents of $X$ become parents of child nodes of $X$.

When $X$ is removed from the children list of its parent nodes, two cases arise:

**Case 1: Parent nodes of $X$ have more than one child** In that case, $X$ is removed from the children list of parent nodes of $X$.

**Case 2: Parent nodes of $X$ have only one child i.e. $X$ itself** In that case, all child nodes of $X$ become the child nodes of parent nodes of $X$. 

65
Input: $\mathcal{L}$ - lattice of closed itemsets, $t$ - transaction to insert
Output: $\mathcal{L}$ - updated lattice
newNode - pointer to node created corresponding to $t$

Process:
create a new node $\mathcal{N}$ with $I_\mathcal{N} = t$

for all item $i \in t$ in $\mathcal{L}$ do
  if $F_i$ is ancestor of $\mathcal{N}$ then
    find child and parents of $\mathcal{N}$ among the descendants of $F_i$
  else
    if $F_i$ is descendant of $\mathcal{N}$ then
      child of $\mathcal{N}$ is $F_i$ and parents of $\mathcal{N}$ are among the ancestors of $F_i$
    else
      if common parent $X$ of $F_i$ and $\mathcal{N}$ exist in $\mathcal{L}$ then
        parent of $\mathcal{N}$ is $X$ and child of $\mathcal{N}$ is among the descendants of $F_i$
      else
        create a new node $X$ where $I_X = I_{F_i} \cap I_{\mathcal{N}}$
parent of $\mathcal{N}$ is $X$ and child of $\mathcal{N}$ is among descendants of $F_i$
        children of $X$ are $F_i$ and some ancestor of $\mathcal{N}$.
        parents of $X$ are among the ancestors of $F_i$.
      end if
    end if
  end if
end for

Update $ITable$ and frequency of $\mathcal{N}$ and its ancestor nodes.
return a pointer to $\mathcal{N}$

Algorithm 4.3: Procedure Insert
The detailed procedure for deleting a node from $\mathcal{L}$ is presented in Algorithm 4.4.

**Example 4.3** Fig. 4.3 shows the updated CILattice after removing the transaction $acd$ in the dataset mentioned in Fig. 4.2 (a).

### 4.4.5 Update ITable

$ITable$ is updated whenever there is a change in $F_i$, $i \in I$. Two cases arise here:

1. If $\mathcal{N}$ is added as an ancestor node of $F_i$ in $\mathcal{L}$, First Node $F_i$ corresponding to $i \in I_\mathcal{N}$ is set to $\mathcal{N}$.

2. If a node $\mathcal{N}$ is deleted from $\mathcal{L}$ then First Node $F_i$ corresponding to $i \in I_\mathcal{N}$ is set to its child node $X$ where $i \in I_X$.

Figure 4.2: (a) Updated Dataset $D_4$ (b) Updated ITable and $\mathcal{L}$ after inserting the transaction $ae$. 

The detailed procedure for deleting a node from $\mathcal{L}$ is presented in Algorithm 4.4.

**Example 4.3** Fig. 4.3 shows the updated CILattice after removing the transaction $acd$ in the dataset mentioned in Fig. 4.2 (a).
Input: $\mathcal{L}$ - lattice of closed itemsets, $X$ - Node to delete
Output: $\mathcal{L}$ - updated lattice of closed itemsets

Process:

\begin{verbatim}
for all child nodes $Y$ of $X$ do
    if $X$ is the only parent node of $Y$ then
        for all parent nodes $Z$ of $X$ do
            set $Z$ as parent of $Y$
        end for
    end if
end for

for all parent nodes $Z$ of $X$ do
    if $X$ is the only child node of $Z$ then
        for all child nodes $Y$ of $X$ do
            set $Y$ as child node of $Z$
        end for
    end if
end for

Update Itable.
Update Frequency of $X$ and its ancestor nodes
\end{verbatim}

Algorithm 4.4: Procedure Delete

4.4.6 Update Frequency

Frequency of a node in $\mathcal{L}$ changes in the following cases:

1. Incoming transaction $t$ exists as a closed itemset of node $X$ in $\mathcal{L}$.
   In that case, frequency of node $X$ and all its ancestor nodes is incremented by one.

2. Incoming transaction $t$ doesn’t exist as closed itemset of a node in $\mathcal{L}$.
   A new node $\mathcal{N}$ is created whose frequency is one more than the frequency of its child node. When $\mathcal{N}$ is inserted in $\mathcal{L}$ then frequency of all the ancestor
Figure 4.3: (a) Updated Dataset $D_5$ (b) Updated $Itable$ and $L$ after removing the transaction $acd$

nodes of $N$ is incremented by one.

3. A node $N$ is removed from $L$.

Frequency of $N$ and its ancestor nodes is decremented by one.

4.5 **CLICI - Algorithm for Data Streams**

$CLICI$ discovers closed itemsets in both sliding window as well as damped window model. The algorithm has an online component that processes the transactions of stream without candidate generation and stores the results in a synopsis called $CILattice$. The new transaction is inserted in the lattice, if not already there.
Offline component is invoked on demand and mines the closed itemsets from the synopsis based on dynamically specified support threshold. User can experiment by varying support thresholds with minimal memory and computational expense, since it requires just one traversal of the lattice $L$. Performance of $\text{CLICI}$ is independent of the window model used. Functioning of $\text{CLICI}$ in the two window models is described below:

### 4.5.1 Online Component in Sliding Window Model

Online component of $\text{CLICI}$ stores the closed itemsets in a synopsis, $\text{CILattice}$, corresponding to the transactions in sliding window $W$. New transactions are inserted using the insert procedure till the window reaches to its maximum size, say $w$.

Once the window is full, older transactions are removed when new transactions arrive. Most of the existing sliding window based algorithms eg. $\text{GC} - \text{Tree}$ [Chen and Li, 2007], $\text{CFI} - \text{Stream}$ [Jiang and Gruenwald, 2006] and $\text{Moment}$ [Chi et al., 2006] keep the window in memory for two purposes: i) find the frequency of newly created nodes when a new transaction arrives. ii) To find the oldest transaction to be removed from the window.

$\text{CLICI}$ doesn’t keep the window in memory. The algorithm updates the frequency of newly created nodes using the lattice structure of $L$. Further, $\text{CLICI}$ takes advantage of the fact that each transaction is a closed itemset and thus exist as a node in synopsis. $\text{CLICI}$ maintains an array called nodeArray, of size $w$ where each entry of nodeArray corresponding to index $i$ is a pointer to the node having transaction $t_{r-w+i}$ as a closed itemset. When window reaches to its maximum size, node corresponding to oldest transaction $t_{r-w+1}$ is searched using nodeArray and is removed using delete procedure 4.4 while new transaction is inserted using insert procedure 4.3.

Let us illustrate this with an example. Let us consider the dataset $D_3$ mentioned in Fig. 4.1 with five transactions, $\{t_1, t_2, t_3, t_4, t_5\}$. Let size of sliding window, $w = 5$. Fig 4.4 shows the nodeArray for dataset $D_3$. 


Let us assume that a new transaction $t_6$ has arrived in the stream. Transaction $t_1$ will be removed from the sliding window to keep the window size equal to five. Now the window will have the transactions $\{t_2, t_3, t_4, t_5, t_6\}$. Fig. 4.5 shows the nodeArray when $t_1$ has been removed from the window and $t_6$ has been added in the window.

### 4.5.2 Online Component in Damped Window Model

*CLI*CI* maintains all the closed itemsets starting from the set landmark point. Since every transaction arriving in stream is a closed itemset, size of the lattice increases with every non-repeated transaction. Damping the effect of older transactions followed by pruning keeps the size of lattice in control thus maintaining
Input: $D$: data stream
nodeArray: An Array having pointers to nodes in $\mathcal{L}$
Output: $\mathcal{L}$ - updated lattice of closed itemsets

Process:
Let $w$ denote size of the window $W$

while $W$ is not full do
    let $t$ be the transaction of data stream $D$
    newNode = insert($t$) {inserts $t$ in sliding window $W$}
    set nodeArray[i] to pointer of newNode
end while

set deletePointer to 0
for all transaction $t'$ of $D$ do
    decrement support of node $X$ corresponding to nodeArray[deletePointer]
    if support of $X$ becomes same as support of one of its child nodes then
        delete $X$
    end if
    newNode = insert $t'$ in $W$
    set nodeArray[deletePointer] to newNode
    increment deletePointer
    if deletePointer == $w$ then
        set deletePointer to 0
    end if
end for

Algorithm 4.5: Online Component in Sliding Window Model
Chang and Lee define a decay factor $d$ that is applied whenever a new transaction arrives [Chang and Lee, 2003]. Decay factor is defined as $d = b^{-h}$ ($b > 1$, $h \geq 1$, $b^{-1} \leq d < 1$), where decay-base $b$ determines the amount of weight reduction per decay-unit and decay-base-life $h$ is defined by the number of decay-subunits that makes the current weight be $b^{-1}$. When a new transaction $t_k$ arrives at time $k$, number of transactions in the current stream $|D|_k$ is updated as:

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

Every node $N$ stores MRtid, the transaction identifier of the most recent transaction that contains the itemset $I_N$. Let $f_{cur}$ denote the decayed support count of

Figure 4.5: (a) Toy Dataset (b) Itable and $L$ (c) nodeArray
node $N$ in $\mathcal{L}$ where $I_N = t_k$ when the transaction $t$ has arrived and $f_{pre}$ denote the decayed support count of $N$ before the arrival of $t$. $f_{cur}$ is updated as follows:

$$(f_{cur} = f_{pre} * d^{k-(MRtid_{pre})} + 1)/|D_k|, \quad MRtid_k = k.$$  

When decayed support count (age) of the transaction $t$ becomes less than user specified threshold for pruning $S_{prn}$, then the node corresponding to $t$ is removed from $\mathcal{L}$. Note here that pruning threshold $S_{prn}$ is distinctly different from support threshold. While support threshold deals with the frequency of the itemset, $S_{prn}$ takes into account the effect of age of the itemset. For higher values of $S_{prn}$, reduction in the size of the synopsis is more. Since, data characteristics of the stream change with time, guessing the correct value of $S_{prn}$ is a difficult task. An alternative approach, though somewhat naive, would be to fix the size of synopsis based on total available memory and remove decayed nodes from the lattice so as to keep the recent and repetitively occurring itemsets intact.

### 4.5.3 Offline Component

Offline component mines the closed itemsets from the synopsis, $CILattice$, based on dynamically specified support threshold. The process is independent of the underlying window model. Mining closed itemsets require exactly one traversal of the lattice thereby encouraging user to experiment with varying support thresholds. Offline component can be invoked either periodically or on demand. Algorithm 4.7 describes the offline component in data streams.
Input: $D$: data stream
batchSize: Batch size for periodic pruning
$S_{prn}$: Pruning Threshold
Output: $L$ - updated lattice of closed itemsets

Process:
Let $b$ denotes decay-base
Let $h$ denotes decay-units
compute $d = b^{-1/h}$
set curTransNum to 1
for all transactions $t$ of $D$ do
    newNode ($X$) = insert($t$)
    set $k$ to curTransNum
    set support of $X$ to $(\text{Sup}(X) \times d^{k - (\text{MRtid}_{pre})}) + 1$)
    set MRtid of $X$ to curTransNum
    for all ancestor nodes $Y$ of $X$ do
        set support of $Y$ to $(\text{Sup}(Y) \times d^{k - (\text{MRtid}_{pre})}) + 1$)
        set MRtid of $Y$ to curTransNum
    end for
if periodic update is due after a batch then
    for all nodes $Z$ of $L$ do
        if support($Z$) < $S_{prn}$ then
            delete($Z$)
        end if
    end for
end if
increment curTransNum
end for

Algorithm 4.6: Online Component in Damped Window Model
Input: $CILattice$ having $\mathcal{L}$ and ITable

sup\_threshold: User Specified Support Threshold

Output: Set of Closed Itemsets

Process:

Create an empty queue $Q$

enQueue top node of $\mathcal{L}$

while $Q$ is not empty do

    deQue a node $X$ from $Q$

    if $\text{Sup}(X) \geq \text{sup\_threshold}$ then

        visit($X$) \{outputs the closed itemset and its support\}

    end if

    for all child nodes $Y$ in $C(X)$ do

        enqueue($Y$)

    end for

end while

Algorithm 4.7: Offline Component in Data Stream
4.6 Experimental Analysis

In this section, we present the results of applying CLICI algorithm for sliding window model and damped window model. Experiments were conducted using synthetic datasets generated using IBM data generator $^1$ and some real life datasets $^2$. We conducted following experiments to test the scalability and efficiency of CLICI:

1. We evaluated CLICI for scalability using several synthetic and real life datasets.

2. We compared CLICI with an existing sliding window model algorithm, Moment and compared (i) per slide time, (ii) memory required for execution.

All experiments were done on a 2GHz AMD Dual-Core PC with 3 GB main memory, running redhat linux operating system. All algorithms were implemented in C++ and compiled using g++ compiler. In all experiments, the transactions of each dataset were examined one by one in sequence to simulate the environment of an online data stream.

4.6.1 Datasets Used

We generated three different synthetic datasets T1I4D100K, T5I4D100K, T10I4D100K using IBM data generator [Agrawal and Srikant, 1994]. Three numbers of each dataset denote the average transaction length (T), average maximum potential frequent itemset size (I) and the total number of transactions (D) respectively. Number of items in each of the dataset is 1000.

We experimented on two real life datasets, BMS-Web-View-1 and BMS-POS [Zheng et al., 2001]. BMS-Web-View-1 contains a few months of clickstream data from an e-commerce web site [Zheng et al., 2001]. Each transaction in the dataset is a web session consisting of all the product detail pages viewed in that session.

---

$^1$downloaded from http://www.almaden.ibm.com/software/quest/Resources/

$^2$downloaded from http://fimi.ua.ac.be/data/
<table>
<thead>
<tr>
<th>Dataset</th>
<th>Trans</th>
<th>Items</th>
<th>Av Trans Len</th>
<th>Max Trans Len</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 BMS-Web-View-1</td>
<td>59602</td>
<td>497</td>
<td>2.5</td>
<td>267</td>
</tr>
<tr>
<td>2 BMS-Pos</td>
<td>515,597</td>
<td>1657</td>
<td>6.5</td>
<td>164</td>
</tr>
</tbody>
</table>

Table 4.2: Characteristics of Real-life Datasets

BMS-POS contains several years of point of sale data from a large electronics retailer [Zheng et al., 2001]. Each item in the dataset represents a product category and each transaction is a customer’s purchase transaction consisting of all the product categories purchased at one time. Table 4.2 describes the characteristics of the two datasets in terms of number of transactions, number of distinct items, the average transaction size and the maximum transaction size.

4.6.2 Damped Window Model

Following experiments were conducted to demonstrate the scalability of CLICI:

Closed itemsets were generated for the synthetic datasets having different transaction lengths. Here, the landmark was set to the beginning of the stream and no pruning was applied. As shown in Fig. 4.6 (a), the number of closed itemsets increases steeply with increase in average transaction length leading to increase in the size of the lattice. This gives an idea of the magnitude of the problem. Fig. 4.6 (b) shows per-transaction processing time on the same datasets. Since lattice size increases monotonically, the corresponding per-transaction-processing time increases accordingly, as expected.

4.6.2.1 Experiments on Synthetic data

In this experiment, closed itemsets were generated for T10I4D100K dataset and different pruning thresholds were applied to prune the decayed transactions. Fig. 4.7 (a) shows the effect of pruning threshold $S_{prn}$ on per-transaction-processing time in the dataset T10I4D100K, as stream progresses. Higher values of $S_{prn}$ lead to substantial reduction in the size of synopsis as expected. Hence per-transaction-processing time is nearly constant. As value of $S_{prn}$ decreases, size of the synopsis...
Figure 4.6: (a) Rate of growth of generated closed itemsets with increase in transaction lengths (without pruning) (b) corresponding increase in per-transaction-processing time

Figure 4.7: Per-transaction-processing time corresponding to (a) different values of $S_{prn}$ (b) fixed size of synopsis in T10I4D100K dataset
increases, leading to increase in rate of growth of per-transaction-processing time. However if size of synopsis is fixed, per transaction processing time is nearly constant. Fig 4.7 (b) shows the results for different sizes of synopsis. This establishes the scalability of the algorithm.

4.6.2.2 Experiments on Real Life datasets

Fig. 4.8 (a) and Fig. 4.9 (a) shows graph depicting per-transaction-processing time when CLICI runs on BMS-Web-View-1 and BMS-POS datasets respectively. Different values of pruning threshold $S_{prn}$ are tested and it is observed that higher values of $S_{prn}$ leads to reduced size of the synopsis and hence reduced per-transaction-processing time. It is also observed that rate of growth of per-transaction-processing time increases with decrease in pruning threshold. Fig. 4.8 (b) and Fig. 4.9 (b) present results on different sizes of the synopsis, 20K, 40K, 60K, 80K, 100K closed itemsets. As depicted in the graphs, per-transaction-processing time remains nearly constant on a particular size of the synopsis and increases with increase in size of synopsis.
4.6.3 Sliding Window Model

We run CLICI in the sliding window model and compared the performance with Moment [Chi et al., 2006], existing sliding window based algorithm.

Moment is among the first algorithms for mining frequent closed itemsets in data streams. Since Moment is designed to handle only the frequent closed itemsets, the memory gets quickly exhausted if the support threshold is set to 0. Since CLICI discovers all closed itemsets, comparing performance of CLICI and Moment on the basis of transactions is not feasible. Hence we compare average slide time of CLICI and Moment. We run both the algorithms on T10I4D100K to generate a particular number of closed itemsets and then slide the window 100 times to observe average slide time and memory usage. We conducted same experiment on BMS-Web-View-1 and BMS-POS datasets. We note that in all the three datasets, CLICI needs much less memory as compared to Moment (Fig. 4.10).

CLICI algorithm generates all closed itemsets and maintains them in a lattice which facilitates generation of non-redundant association rules. Moment generates
only the frequent closed itemsets and the synopsis doesn’t help in generating non-redundant association rules. Even then $CLICI$ runs slightly faster than $Moment$ (Fig. 4.10). The credit goes to the fast retrieval mechanism using $ITable$. It was also observed that on an average $Moment$ traverses the in-core window in memory ten times in a single slide making it slow while $CLICI$ algorithm scans the transactions only once.

4.7 Conclusions

In this chapter, we have described a novel data structure, $CILattice$ for storing closed itemsets. Next, we described $CLICI$ algorithm to mine all recent closed itemsets in damped window model and all closed itemsets in sliding window model of data stream. The algorithm is based on Formal Concept Analysis, a well established discipline in applied mathematics. The proposed algorithm maintains a lattice of closed itemsets in the stream and delivers frequent closed itemsets to the user on demand, based on the dynamically specified support threshold. Since
Figure 4.11: Comparison of CLICI with Moment (a) Memory (b) Per slide time on BMS-Web-View-1 Dataset

Figure 4.12: Comparison of CLICI with Moment (a) Memory (b) Per slide time on BMS_POS Dataset
the synopsis is independent of the support threshold, user is encouraged to explore and experiment.