CHAPTER 4

CFI MINING IN DATA STREAMS USING SLIDING WINDOWS

Data streams are often continuous, unbounded, high-speed data and its distribution changes with time. In the recent years, data streams play a vital role in many applications. Examples of such applications include financial analysis, network monitoring, sensor networks, telecommunication data management, web usage and others. One important activity in such applications is Mining Frequent Itemsets (MiFIs). Two algorithms are described for MiFIs from data streams in the previous chapter. A main drawback observed in these algorithms is that since the number of FIs is very large, the time taken to generate them and the storage needed are very high. To overcome these drawbacks, an algorithm that mines a compact form of FIs known as Closed Frequent Itemsets (CFI) is proposed in this chapter.

4.1 CLOSED FREQUENT ITEMSET MINING

Effective mining of data streams needs a quick computation of the FIs with only a single scan over the stream. Also applications need to represent the FIs efficiently and in a compact manner to be more effective. The number of FIs grows exponentially when the support threshold is reduced making it essential to represent the FIs in a compressed form. Whenever a frequent k-itemset is generated, all its $2^k$ non-empty subsets, which are also frequent, are generated and stored. Since any subset of a frequent k-itemset is frequent, the subsets need not be stored but can be inferred from the k-itemset itself.

Two smaller alternatives to FI which do not store all the subsets of an FI are available. They are Maximal Frequent Itemsets (MFI) and Closed Frequent Itemsets (CFI). A frequent itemset X is called maximal if there is no other frequent itemset Y such that $X \subseteq Y$. A frequent itemset is closed if none of its supersets have the same support [66]. The relationship between FI, CFI, and MFI is shown in Fig. 4.1. As seen from the Fig. 4.1, $MFI \subseteq CFI \subseteq FI$. Also $|MFI| << |CFI| << |FI|$. The above fact is confirmed when the real world dataset, Connect-4 is analyzed. This dataset is publicly available at www.almaden.ibm.com/cs/quest/demos.html.
It is confirmed that $|\text{MFI}| << |\text{CFI}| << |\text{FI}|$. Table 4.1 shows the number of CFIs, MFIs and FIs extracted from the Connect-4 dataset at different support thresholds. Even though the number of MFIs is very less when compared to the CFIs or FIs, the preferred compressed form is the CFIs rather than the MFIs. Because given an MFI and its support, all its subsets can be generated but their exact support cannot be determined. The only assurance that can be given is that the support of the subsets will be greater than that of the MFI from which it is derived. The CFIs do not suffer from this problem. The subsets of the CFIs and their exact support can be determined [73]. The exact support of a subset is found by finding the CFI with the maximum support which contains that subset. The support of this CFI gives the exact support of the subset [61]. In other words, the complete information about the FIs can be extracted from the CFIs. So CFIs are preferred over MFIs whenever a lossless compressed form of FIs is needed.

Table 4.1 Number of MFIs, CFIs, and FIs

<table>
<thead>
<tr>
<th>Support</th>
<th>#MFI</th>
<th>#CFI</th>
<th>#FI</th>
</tr>
</thead>
<tbody>
<tr>
<td>95%</td>
<td>105</td>
<td>812</td>
<td>2,205</td>
</tr>
<tr>
<td>90%</td>
<td>256</td>
<td>3,486</td>
<td>27,127</td>
</tr>
<tr>
<td>80%</td>
<td>8,756</td>
<td>15,107</td>
<td>53,3975</td>
</tr>
<tr>
<td>70%</td>
<td>1,154</td>
<td>35,875</td>
<td>41,29,839</td>
</tr>
</tbody>
</table>

For example consider a portion of the data stream, $D=\{<T_1, \text{actw}>, <T_2, \text{cdw}>, <T_3, \text{actw}>, <T_4, \text{acdw}>, <T_5, \text{acdtw}>, <T_6, \text{cdt}>\}$. Assume that the specified minimum support threshold is 50%. The frequency threshold in this case is
3. (50% of 6 transactions). The set of Frequent Itemsets (FIs) along with their support count in the descending order of their support count is \{c-6, w-5, cw-5, a-4, d-4, t-4, ac-4, aw-4, cd-4, ct-4, acw-4, at-3, dw-3, tw-3, act-3, atw-3, cdw-3, ctw-3, actw-3\}. The number of FIs is 19. The CFIs in this portion of the data stream are \{c-6, cw-5, cd-4, ct-4, acw-4, cdw-3, actw-3\}. The number of CFIs is only 7. The MFIs in this portion of the data stream are \{cdw-3, actw-3\} and the number of MFIs is only 2. This example helps to illustrate that the number of MFIs is very much less than the number of CFIs which in turn is very much less than the number of FIs.

Suppose that the compressed form, MFI is used to represent the FIs, then all the FIs can be derived from it by finding the subsets of the MFIs. But their support count cannot be determined exactly. In the above example, from the MFIs \{cdw-3, actw-3\} it can be confirmed that ‘w’ is a frequent itemset and its support is at the minimum 3 since ‘w’ is a subset of both MFIs but it cannot be determined from the MFIs that its actual support is 5.

On the other hand, if the CFIs are used to represent the FIs, the support of ‘w’ can be exactly determined. The FI ‘w’ is found to be a subset of the CFIs cw-5, acw-4, cdw-3 and actw-3. The CFI with the maximum support is ‘cw’. So the support of ‘w’ can be exactly determined as the support of ‘cw’ which is 5. This example illustrates the fact that all the FIs can be determined from both the compressed forms, MFIs and CFIs but the exact support of the FIs can be determined only from the CFIs and not from the MFIs. For this reason, CFIs are preferred over MFIs when a lossless compression form of FIs is required [41].

There are many algorithms that mine CFIs over data streams [73]. The TFP algorithm proposed by the authors in 2005 [92] mined top-k frequent Closed Frequent itemsets of length not less than min_l. Chi et al. proposed a new algorithm, MOMENT, (Maintaining Closed Frequent Itemsets by Incremental Updates) [18] to mine and store all closed frequent-itemsets in the year 2006. NewMoment algorithm [49] proposed by Li et al. in the year 2009 is used to mine closed frequent itemsets in data streams with a transaction sensitive sliding window. Hua-Fu Li proposed an interactive single-pass algorithm, called TKC-DS( Top-K frequent Closed Frequent itemsets of Data Streams), in the year 2009 [47]. It is used for mining top-K Closed Frequent itemsets from data streams efficiently within a transaction-sensitive sliding
window. Pauray Tsai proposed the \textit{FCL\_max algorithm} \cite{88} in the year 2010 for mining top-k frequent Closed Frequent itemsets of length not more than max\_l. Tang et al. proposed an algorithm, EMAFCI (Efficient Mining Algorithm for Frequent Closed Frequent itemsets) over data streams \cite{85} in the year 2011. AFPCFI-DS algorithm is an improved FP tree-based algorithm for Closed Frequent Itemset Mining over data streams using sliding window model developed by Dai and Chen in 2012\cite{19}. In the year 2013, Fatemeh Noria et al. \cite{67} have introduced an effective and efficient algorithm, \textit{TMoment algorithm}, for closed frequent itemset mining over data streams. These algorithms were reviewed in the second chapter.

Generally data streams contain large amount of data with old and new information. People are always interested in newer information rather than the older information. This leads to the sliding window concept. Sliding window needs less amount of storage space compared with that of an entire data stream. And also the sliding window shows only the current information of a data stream.

Many applications need to concentrate only on the recent transactions. For example, in web usage mining the log files maintained at the proxy servers or at the client machines need to be examined to build the user profile. Such user profiles help in personalizing web pages and also decide pages to be pre-fetched to reduce the fetching time \cite{29}. This application needs to concentrate only on the recent transactions, i.e. the recent entries in the web log files. It does not need to maintain very old information about the user. Also, the user’s behavior keeps changing over time and his old behaviors can be discarded safely while constructing the user’s current profile. A sliding window model \cite{79} is well suited for such applications.

In this chapter, a single-pass algorithm to generate the Closed Frequent Itemsets (CFIs) from the data streams is proposed which is based on the sliding window model \cite{7}. This algorithm named as HATCI Algorithm, (\textbf{H}ash \textbf{T}able of \textbf{C}losed \textbf{I}temsets) is used for building a table of Closed Itemsets (CI-Table) present in the transactions within the current sliding window. Along with this table, three additional tables are maintained for storing the supersets of the CIs (Superset-Table), subsets of the CIs (Subset-Table) and list of CIs affected by each transaction (Transaction-Table). CI-Table, Superset-Table, and Subset-Table are implemented as hash tables which enable fast searching and retrieval operations \cite{69}. For building
and maintaining these tables, a sliding window model is used, that performs only one scan over the data stream. The size of the Transaction-Table is equal to the size of the sliding window. The size of the other tables depends on the number of CIs extracted so far. Whenever the user requests for the CFIs, the CI-Table is accessed sequentially, and all the CIs in it are checked out. Only the CIs with support count greater than or equal to the user specified support threshold are extracted and returned as CFIs.

4.2 HASH TABLES

Hash tables are direct address tables used to store a collection of ‘n’ elements whose keys are unique integers in the range (1, m), where m >= n. Searching a direct address table is clearly an O(1) operation. For a key, ‘k’ the search operation returns the element if the table contains it. If not it returns a NULL value. The two constraints to be satisfied are: the keys must be unique, and the range of the key must be severely bounded. Hash tables are made up of two parts: an array (the actual table where the data to be searched is stored) and a mapping function, known as a hash function. The hash function is a mapping from the input space to the integer space that defines the indices of the array. In other words, the hash function provides a way for assigning numbers to the input data such that the data can then be stored at the array index corresponding to the assigned number.

A perfect hash function is a function which, when applied to all the members of the set of items to be stored in a hash table, produces a unique set of integers within some suitable range. It is ideal to have a perfect hash function but it may not always be possible. In such cases, one of the numerous collision resolution techniques available may be followed. If the number of collisions (cases where multiple keys map onto the same integer), is sufficiently small, then hash tables work quite well and give O(1) search times.

Assume that a hash table of strings is constructed and its size is 12. The hash function used takes the string as input and returns a hash value. The sum of the ASCII (American Standard Code for Information Interchange) values of the characters mod the size of the table produces the hash value that is then returned by
the hash function. In this scenario, the location of string ‘Spark’ is calculated as shown below.

\[
\text{Sum} = \text{ASCII}(\text{S})+ \text{ASCII}(\text{p})+\text{ASCII}(\text{a})+ \text{ASCII}(\text{r})+\text{ASCII}(\text{k})
\]
\[
= 83+112+97+114+107
\]
\[
= 513
\]

Hash value = Sum % 12 = 513 % 12 = 9

Thus the string ‘Spark’ gets stored at location 9 in the hash table. There is always the chance that two inputs will hash to the same output. For example the strings ‘Cab’ and ‘Note’ hash to the same location 10. This indicates that both elements should be inserted at the same place in the array, and this is impossible. This phenomenon is known as a collision. There are many collision resolution techniques like separate chaining, open addressing, robin hood hashing, and cuckoo hashing.

Separate chaining otherwise called open hashing or closed addressing, requires a slight modification to the data structure. Instead of storing the data elements right into the array, they are stored in linked lists. Each slot in the array then points to one of these linked lists. When an element hashes to a value, it is added to the linked list at that index in the array. Because a linked list has no limit on length, collisions are no longer a problem. If more than one element hashes to the same value, then both are stored in that linked list. It is called “open hashing” because the values are not stored in the array but in separate linked lists. It is also called “closed addressing” because the address is entirely decided by the hash value.

Open addressing otherwise called closed hashing is another strategy where all entry records are stored in the bucket array itself. When a new entry has to be inserted, the buckets are examined, starting with the hashed-to slot and proceeding in some probe sequence, until an unoccupied slot is found. When searching for an entry, the buckets are scanned in the same sequence, until either the target record is found, or an unused array slot is found, which indicates that there is no such key in the table. The name "open addressing” refers to the fact that the location (address) of the item is not determined by its hash value. The name “closed hashing” means that the values are stored in the array itself.
Re-hashing schemes decide the probe sequence. They use a second hashing operation when there is a collision. If there is a further collision, re-hashing is done until an empty "slot" in the table is found. The re-hashing function can either be a new function or a re-application of the original one. As long as the functions are applied to a key in the same order, then a sought key can always be located. Some of the well-known re-hashing schemes that generate different probe sequences include:

- Linear probing, in which the interval between probes is fixed (usually 1)
- Quadratic probing, in which the interval between probes is increased by adding the successive outputs of a quadratic polynomial to the starting value given by the original hash computation
- Double hashing, in which the interval between probes is computed by another hash function

Linear probing is a simple scheme in which the next slot in the table is checked on a collision. It calculates the new address extremely quickly but suffers from the clustering problem. Re-hashes from one location occupy a block of slots in the table which "grows" towards slots to which other keys hash. This exacerbates the collision problem and the number of re-hashes required to resolve the collision can become large.

Quadratic probing is a scheme in which a higher (usually 2nd) order secondary hash function of the hash index already obtained is used to calculate the address. Better behavior is usually obtained with quadratic probing, where the secondary hash function depends on the re-hash index:

\[ \text{address} = h(\text{key}) + c i^2 \]  \hspace{1cm} (4.1)

on the \(i^{th}\) re-hash. Since keys which are mapped to the same value by the primary hash function follow the same sequence of addresses, quadratic probing shows secondary clustering. However, secondary clustering is not nearly as severe as the clustering shown by linear probes.

Like linear probing, double hashing also uses one hash value as a starting point and then repeatedly steps forward an interval until the desired value is located, an empty location is reached, or the entire table has been searched; but this interval is
decided using a second, independent hash function (hence the name double hashing). Unlike linear probing and quadratic probing, the interval depends on the data, so that even values mapping to the same location have different bucket sequences; this minimizes repeated collisions and the effects of clustering.

A drawback of all these open addressing schemes is that the number of stored entries cannot exceed the number of slots in the bucket array. In fact, even with good hash functions, their performance dramatically degrades when the load factor grows beyond 0.7 or so. Thus a more aggressive resize scheme is needed. Separate linking works correctly with any load factor although performance is likely to be reasonable if it is kept below 2. For many applications, these restrictions mandate the use of dynamic resizing, with its attendant costs.

Open addressing schemes also put more stringent requirements on the hash function: besides distributing the keys more uniformly over the buckets, the function must also minimize the clustering of hash values that are consecutive in the probe order. Using separate chaining, the only concern is that too many objects map to the same hash value; whether they are adjacent or nearby is completely irrelevant.

Open addressing only saves memory if the entries are small (less than four times the size of a pointer) and the load factor is not too small. If the load factor is close to zero (that is, there are far more buckets than stored entries), open addressing is wasteful even if each entry is just two words.

Another alternative open-addressing solution is cuckoo hashing, which ensures constant lookup time in the worst case, and constant amortized time for insertions and deletions. It uses two or more hash functions, which means any key/value pair could be in two or more locations. For lookup, the first hash function is used; if the key/value is not found, then the second hash function is used, and so on. If a collision happens during insertion, then the key is re-hashed with the second hash function to map it to another bucket. If all hash functions are used and there is still a collision, then the key it collided with is removed to make space for the new key, and the old key is re-hashed with one of the other hash functions, which maps it to another bucket. If that location also results in a collision, then the process repeats until there is no collision or the process traverses all the buckets, at which point the
table is resized. By combining multiple hash functions with multiple cells per bucket, very high space utilization can be achieved.

One interesting variation on double-hashing collision resolution is Robin Hood hashing. The idea is that a new key may displace a key already inserted, if its probe count is larger than that of the key at the current position. The net effect of this is that it reduces worst case search times in the table. The criterion for bumping a key does not depend on a direct relationship between the keys.

Java has a built-in class, HashTable that implements a hash table. It maps keys to values and also has several built-in functions to manipulate the hash table. Any non-null object can be used as a key or as a value. HashTable class stores key/value pairs in a table. When using a HashTable, an object that is to be used as a key and the value that should be linked to that key are specified. The key is then hashed, and the resulting hash code is used as the index at which the value is stored within the table. Java’s hashing uses hashCode() method from the key and value objects to compute. Following is the core code from HashTable where the hashCode ‘h’ is computed. Both the key’s and value’s hashCode() method is called. hashCode() guarantees distinct integers by using the internal address of the object.

\[ h += e.key.hashCode() \oplus e.value.hashCode() \]

(4.2)

When there is a collision HashTable class makes use of separate chaining to resolve collisions. HashTable stores elements in buckets. In separate chaining, every bucket will store a reference to a linked list. If one element is stored in bucket 1, it means that in bucket 1 there is a reference to a linked list and in that linked list there is a node with two fields: the key and its corresponding value. If one more element also hashes to the same bucket another node containing the new key and its value is appended to the linked list.

Separate chaining solves the problem of collision in a simple yet powerful manner. Of course, there are some drawbacks. Imagine the worst case scenario where through some fluke of bad luck and bad programming, every data element hashed to the same value. In that case, to do a lookup, a straight linear search on a linked list is needed, which means that the search operation is back to being \( O(n) \). The worst case search time for a hash table is \( O(n) \). However, the
probability of that happening is so small that, while the worst case search time is $O(n)$, both the best and average cases are $O(1)$.

### 4.3 HATCI ALGORITHM

Hash Table of Closed Itemsets (HATCI) algorithm is used for mining Closed Frequent Itemsets (CFI) from the transactions in the current sliding window. HATCI algorithm employs hash tables for maintaining the Closed Itemsets (CIs), the supersets and subsets of the CIs and also for storing the list of CIs generated by each transaction. For building and maintaining these tables, a sliding window model is used and it performs a single scan over the data stream [96]. The size of the transaction table is equal to that of the sliding window. The tables are constructed by scanning the data stream only once. Also whenever a new transaction arrives, all the tables are updated incrementally. When a user request arrives for the CFI, all the CIs in the CI-Table at that instant are retrieved and those with support greater than or equal to the user specified support threshold are returned to the user [90].

#### 4.3.1 Closed Itemset Hash Table

The closed itemsets in the transactions of the current sliding window are stored in a hash table called Closed Itemset Hash Table (CI-Table). Each entry in the hash table is a pair of values, the Closed Itemset (CI) and its support (s). The key that determines the location of the CI is a pair (f, n) where ‘f’ indicates the numeric representation of the first item in the CI and ‘n’ indicates the order of the CIs generated so far that begins with this particular item. This hash table organization ensures that a CI can be directly accessed, when it has to be updated or checked during further processing.

For example, consider that the Closed Itemsets (CI) at a particular instant are \{ (ab, 4), (abc, 3), (bcd, 2), (bc, 4), (cd, 3), (a, 5) \} generated in that order. The CI (ab, 4) indicates that the support of the CI, ‘ab’ is 4. The ‘f’ value for the CI ‘ab’ is 1. Because, the numeric representation assigned to its first item ‘a’ is 1. For itemset ‘bc’, the ‘f’ value is 2 since the numeric representation for its first item ‘b’ is 2. And also, the ‘n’ value of the CI ‘ab’ is 1 and the ‘n’ value of ‘abc’ is 2 since the CI ‘ab’ is generated first and then ‘abc’ is generated.
Table 4.2 CIs and their Keys

<table>
<thead>
<tr>
<th>Closed Itemset</th>
<th>Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>ab</td>
<td>(1,1)</td>
</tr>
<tr>
<td>abc</td>
<td>(1,2)</td>
</tr>
<tr>
<td>bcd</td>
<td>(2,1)</td>
</tr>
<tr>
<td>bc</td>
<td>(2,2)</td>
</tr>
<tr>
<td>cd</td>
<td>(3,1)</td>
</tr>
<tr>
<td>a</td>
<td>(1,3)</td>
</tr>
</tbody>
</table>

The Table 4.2 lists the keys assigned to each CI assuming that they are generated in the listed order and Table 4.3 shows the CI-table maintained as a hash table for this example.

Table 4.3 CI-Table

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
<th>CI</th>
<th>Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1,1)</td>
<td>ab</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>(1,2)</td>
<td>abc</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>(1,3)</td>
<td>a</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>(2,1)</td>
<td>bcd</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>(2,2)</td>
<td>bc</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>(3,1)</td>
<td>cd</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

4.3.2 Superset-Table

The supersets of each closed itemset in the CI-table are listed out in this hash table. This hash table is maintained for quick and easy updates when a transaction is deleted and the support of the CIs affected by this transaction have to be decremented. The value field of this hash table is a list containing the keys of all
the supersets of a CI listed in the CI-Table. Continuing with the same example given in the previous section, the Superset-Table is shown below in the Table 4.4.

Table 4.4 Superset-Table

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1,1)</td>
<td>{(1,2)}</td>
</tr>
<tr>
<td>(1,2)</td>
<td>NULL</td>
</tr>
<tr>
<td>(1,3)</td>
<td>{(1,1), (1,2)}</td>
</tr>
<tr>
<td>(2,1)</td>
<td>NULL</td>
</tr>
<tr>
<td>(2,2)</td>
<td>{(1,2), (2,1)}</td>
</tr>
<tr>
<td>(3,1)</td>
<td>{(2,1)}</td>
</tr>
</tbody>
</table>

The CI-Table and the Superset-Table are of the same size. This table maintains a record of the supersets of each generated Closed Itemset (CI). For example the third entry shows that the CI, ‘a’ with key (1,3) has two supersets, the CIs ‘ab’ and ‘abc’ with keys (1,1) and (1,2) respectively. Such information about all the CIs listed in the CI-Table is maintained in the Superset-Table.

4.3.3 Subset-Table

The subsets of each closed itemset in the CI-table are listed out in this hash table. This hash table is maintained for quick and easy updates when a new transaction is added and the support of the Closed Itemsets (CIs) which are subsets of the new transaction have to be incremented. Also whenever a transaction is deleted, the support of CIs affected by the transaction being deleted and their subsets has to be decremented. Thus whenever transactions are added or deleted a quick access to the subsets of the CIs is essential which is provided by the Subset-Table. The value field of this hash table is a list containing the keys of all the subsets of a CI listed in the CI table. Continuing with the same example given in the previous sections, the Subset-Table is shown in the Table 4.5.
The CI-Table, the Superset-Table and the Subset-Table are of the same size. The Subset-Table maintains a record of the subsets of each generated Closed Itemset (CI). For example the second entry in Table 4.5 shows that the CI, ‘abc’ with key (1,2) has three subsets, the CIs ‘ab’, ‘a’ and ‘bc’ with keys (1,1), (1,3) and (2,2) respectively. Such information about all the CIs listed in the CI-Table is maintained in the Subset-Table.

### 4.3.4 Transaction-Table

Transaction-Table lists the Closed Itemsets (CIs) created by each transaction in the sliding window. Thus the size of the Transaction-Table is decided by the size of the sliding window. The transaction table contains the identifiers of the transactions in the data stream that are stored currently in the sliding window and the Closed Itemsets (CIs) affected by each transaction. The Transaction-Table has two columns T$_{id}$ and CI-Keys and two pointers ‘s’ and ‘e’ pointing to the oldest and the latest transaction in the sliding window respectively. The T$_{id}$ column lists the transaction identifier and the CI-Keys column lists the keys of CIs affected by this transaction. Table 4.6 shows a sample Transaction-Table assuming that the size of the sliding window is four and the current transactions in the window are \{ < T$_1$, cd>, < T$_2$, ab>, < T$_3$, abc>, < T$_4$, abc> \}. ‘s’ points to T$_1$ and ‘e’ points to T$_4$ since T$_1$ is the oldest transaction and T$_4$ is the latest transaction in the current window. The keys of the closed itemsets identified in the data stream are listed in the CI-Keys field of Table 4.6.
Table 4.6 Transaction-Table for \( \{ T_1, T_2, T_3, T_4 \} \)

<table>
<thead>
<tr>
<th>T\text{Id}</th>
<th>CI-Keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>T_1</td>
<td>{ (3,1) }</td>
</tr>
<tr>
<td>T_2</td>
<td>{ (1,1) }</td>
</tr>
<tr>
<td>T_3</td>
<td>{ (1,1), (1,2), (3,2) }</td>
</tr>
<tr>
<td>T_4</td>
<td>{ (1,1), (1,2), (3,2) }</td>
</tr>
</tbody>
</table>

Table 4.7 shows the corresponding CI-Table. The Transaction-Table is maintained as a circular table so that the current transactions in the sliding window are represented in the table between the ‘s’ and ‘e’ pointers. In the above example when transaction \( T_3 \) arrives, the oldest transaction \( T_1 \) has to be removed and in its place the latest transaction \( T_3 \) has to be inserted. For this the Transaction-Table has to be updated circularly as follows: the ‘s’ pointer is advanced so that it points to \( T_2 \) and the new transaction identifier \( T_5 \) with its list of CI-Keys is entered into the table after incrementing ‘e’ circularly. This makes it point to the entry previously occupied by \( T_1 \).

Table 4.7 CI-Table for \( \{ T_1, T_2, T_3, T_4 \} \)

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
<th>Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1,1)</td>
<td>ab</td>
<td>3</td>
</tr>
<tr>
<td>(1,2)</td>
<td>abc</td>
<td>2</td>
</tr>
<tr>
<td>(3,1)</td>
<td>cd</td>
<td>1</td>
</tr>
<tr>
<td>(3,2)</td>
<td>c</td>
<td>3</td>
</tr>
</tbody>
</table>

4.3.5 Phases of HATCI algorithm

The proposed Hash Table of Closed Itemsets (HATCI) algorithm is a single-pass algorithm that uses a sliding window model of data streams. It builds and maintains three hash tables CI-Table, Superset-Table, and Subset-Table and a circular table Transaction-Table. The tables are used to store the Closed Itemsets(CIs) generated by the transactions in the sliding window, its supersets, subsets and CIs affected by each transaction respectively. The algorithm retrieves the
Closed Frequent Itemsets (CFIs) whenever a user request arrives. It works in three phases. They are,

1. Initialization Phase
2. Sliding Phase
3. Generation Phase

The architecture diagram of the HATCI algorithm is shown in Fig. 4.2

![Fig. 4.2 Architecture of HATCI algorithm](image)

**Initialization Phase**

The algorithm is in the initialization phase till the sliding window becomes full for the first time. The Initialization phase uses the algorithm, AddNewTransaction to add the new transaction to the Transaction-Table. On the arrival of a new transaction, the algorithm automatically checks the sliding window, whether it is full or not. If the sliding window is not full, then the Transaction-Table, the CI-Table, the Superset-Table and the Subset-Table are updated to reflect the addition of the new transaction by using AddNewTransaction algorithm shown in Fig. 4.3.
Algorithm AddNewTransaction

// Global:
// $N$ - size of the sliding window
// CI-Table - hash table of Closed Itemsets (CIs)
// Superset-Table - hash table of supersets of each CI
// Subset-Table - hash table of subsets of each CI
// Transaction-Table - array of CIs created by each transaction in the window

// Input: New transaction - $T$

// Output: Updated CI-Table, Superset-Table, Subset-Table, and Transaction-Table

1. Take the first item of the new transaction, $T$
2. Check the CI-Table to see if this transaction, $T$ is already present as a CI
3. If it is present, then increment its support count and also the support count of all its subsets listed in the Subset-Table and return
4. If it is not present, then add $T$ as a CI in a temporary list, T-list with support count, $SC = 1$
5. For each closed itemset $C$, in the CI-Table do the following
   a. Find $I = T \cap C$ and $SC(I) = SC(C) + 1$
   b. If $I$ not null
      i. Find key, $K$ of intersection, $I$
      ii. If $I \neq T$
          Add $T$ as superset of $I$ and $I$ as subset of $T$
      iii. If $I \neq C$
           Add $C$ as superset of $I$ and $I$ as subset of $C$
      iv. If $I$ not in T-List
          Add $I$ with its $SC$ to the T-List at the determined key position, $K$
          Otherwise, update only the $SC$ of $I$ in T-List
   v. For each superset $S$, of $C$, listed in Superset-Table, do
      Add $I$ as subset of $S$ and $S$ as superset of $I$
6. For each closed itemset, $i$ in T-list, do
   For each remaining closed itemset, $j$ in the T-list do
      If $i \subset j$
      Set $i$ as subset of $j$ and $j$ as superset of $i$
      else if $j \subset i$
      Set $i$ as superset of $j$ and $j$ as subset of $i$
7. Each closed itemset, $C$ in the T-list is inserted into or updated in the CI-Table
8. Each closed itemset, $C$ in the T-list is also added to the list in the Transaction-Table for this new transaction

Fig. 4.3 Algorithm AddNewTransaction

Transaction-Table shows each transaction in the sliding window with its corresponding list of Closed Itemsets. If a new CI is created or an existing CI is affected by a transaction, then these details are updated in the Transaction-Table. Step 5 is used to find the new CIs or existing CIs that will be affected by the new
transaction. Once the sliding window is full, the HATCI algorithm moves into the Sliding phase.

**Sliding Phase**

The RemoveOldestTransaction algorithm shown in Fig. 4.4 followed by the AddNewTransaction algorithm (Fig. 4.3) is used in the Sliding Phase. Whenever a new transaction arrives in the data stream, the sliding window is checked and if it is full, then the HATCI algorithm moves into the sliding phase. In this phase, the oldest transaction is removed from the Transaction-Table and the CI-Table, Superset-Table and Subset-Table are updated to reflect this deletion using the RemoveOldestTransaction algorithm and then the new transaction is added into the Transaction-Table and the other tables are also updated accordingly using AddNewTransaction algorithm.

The oldest transaction affects the Closed Itemsets (CIs) listed in the first entry of the Transaction-Table. The Support Count (SC) of each of the CIs listed in the first entry is then decreased by one. This reduction in SC should be reflected in the subsets of the updated CIs. So the SC of subsets of this CI is also reduced by one. To make this operation efficient, whenever a CI is created, an entry in the Subset-Table is created listing the keys of its subsets. This table is used to access the subsets of the CIs affected by the oldest transaction and reduce their SC also. If the updated SC of any CI, whether listed in the first entry of Transaction-Table or its subset listed in the Subset-Table, is found to be zero after reduction then the CI is marked as invalid.

These updates may result in some supersets having the same SC as the updated CIs. This violates the property of closed itemsets. So the supersets of the updated CIs are also checked. If the SC of any superset of an updated CI is the same as that of the CI, then that CI is marked as invalid. The Superset-Table maintained for each CI helps in performing this task efficiently. After processing all the CIs listed in the first entry of the Transaction-Table, their subsets and supersets, the CIs marked as invalid are deleted from the CI-Table, the Superset-Table and the Subset-Table.
Algorithm RemoveOldestTransaction

// Global:
// N - size of the sliding window
// CI-Table - hash table of Closed Itemsets(CIs)
// Superset-Table - hash table of supersets of each CI
// Subset-Table - hash table of subsets of each CI
// Transaction-Table - array of CIs created by each transaction in the window

// Input: - NIL

// Output: - Updated Transaction-Table, CI-Table, Superset-Table, Subset-Table after removing the oldest transaction

1. For each Closed Itemset(CI) in the first entry in the Transaction-Table (oldest transaction) do
   a. Reduce $SC$ by one. If $SC = 0$ mark CI as invalid
   b. For each of its subset, S in the Subset-Table do
      i. Reduce $SC$ by one. If $SC = 0$ mark S as invalid.

2. For each superset of the updated CIs do
   If $SC$ of superset = $SC$ of the updated CI then mark CI as invalid

Fig. 4.4 Algorithm RemoveOldestTransaction

Generation Phase

The generation phase of the HATCI algorithm shown in Fig. 4.5 is initiated on the arrival of the user’s request for Closed Itemsets (CIs). When the user queries for the Closed Frequent Itemsets (CFIs) at a particular instant, the CI-Table is scanned sequentially, and then it is checked whether the support count of each of the closed itemset is greater than or equal to the support threshold also specified by the user. The CIs in the table which satisfy this condition are extracted and reported as CFIs. This phase employs a sequential scanning of the CI-Table maintained by the algorithm. The user specifies the support threshold percentage which is then multiplied by the size of the sliding window to obtain the support count.
Algorithm Generate-CFI

// Global:

// N - size of the sliding window
// CI-Table - hash table of Closed Itemsets (CIs)
// Superset-Table - hash table of supersets of each CI
// Subset-Table - hash table of subsets of each CI
// Transaction-Table - array of CIs created by each transaction in the window

// Input:

// User request for CFI
// \( S_t \) - Minimum support threshold percentage

// Output:

// CFIs in current window

1. Calculate \( s = S_t \times N / 100 \)
2. For each entry in the CI-Table do
   - If \( \text{Support Count of } CI \geq s \) then
     - \( CFI = CFI \cup CI \)
3. Return CFI

Fig. 4.5 Algorithm Generate-CFI

4.4 ILLUSTRATION OF HATCI ALGORITHM

Consider the transaction data stream, TDS, with five transactions \( \{T_1, T_2, T_3, T_4, T_5\} \), support threshold \( S_t = 50\% \) and window size = 4. The representation of the data stream is, \( TDS = \{T_1, T_2, T_3, T_4, T_5\} \). The data stream with five transactions and their corresponding Itemsets are given in the Table 4.8.

Table 4.8 Data stream with 5 transactions

<table>
<thead>
<tr>
<th>Transactions</th>
<th>Items</th>
</tr>
</thead>
<tbody>
<tr>
<td>T_1</td>
<td>cd</td>
</tr>
<tr>
<td>T_2</td>
<td>ab</td>
</tr>
<tr>
<td>T_3</td>
<td>abc</td>
</tr>
<tr>
<td>T_4</td>
<td>abc</td>
</tr>
<tr>
<td>T_5</td>
<td>bc</td>
</tr>
</tbody>
</table>
The size of the sliding window is 4, so the two windows over this portion of the data stream are \( W_1 \) and \( W_2 \) which are as follows:

\[
W_1 = \{T_1, T_2, T_3, T_4\}
\]

\[
W_2 = \{T_2, T_3, T_4, T_5\}
\]

The status of the Transaction-Table, the CI-Table, Superset-Table and Subset-Table after the arrival of each of these transactions are illustrated below.

**After transaction \( <T_1, cd> \)**

\( T_1 \) is the first transaction. So its itemset, ‘cd’ is a Closed Itemset (CI) and its key is determined as (3,1) since starting item is ‘c’ which corresponds to 3 and this is the first CI that starts with ‘c’. This CI is stored in the temporary list, T-List. Before arrival of \( T_1 \) the CI-Table is empty. So no more processing is done and now the T-List is used to update the four HATCI tables. Fig. 4.6 shows the state of the four tables of HATCI algorithm after processing the transaction \( T_1 \).

The Transaction-Table shown in Fig. 4.6(a) contains only one entry, because only the first transaction in the data stream has been processed and the window now contains only one transaction, \( T_1 \). Both ‘s’, ‘e’ pointers point \( T_1 \) at this stage. The CIs generated or affected by this transaction is only ‘cd’, the transaction itemset itself and hence its key (3,1) is listed in the CI-Keys field of the Transaction Table. The state of the CI-Table is shown in Fig. 4.6(b) and it lists the only CI, ‘cd’ with its key (3,1) and its support is 1. The Superset-Table and the Subset-Table are shown in Fig. 4.6(c) and (d) respectively and they show that the CI, ‘cd’ whose key is (3,1) does not have any supersets or subsets at this stage.

![Fig. 4.6 State of HATCI tables after \( <T_1, cd> \)](image-url)
After transaction \(<T_2, ab>\)

The processing of the transaction \(T_2\) and the updates performed on the four HATCI tables are discussed here. The itemset ‘ab’ in transaction, \(T_2\) is not present in the existing CI-Table. So it is put in the T-List as a CI with Support Count(SC) equal to 1. Now \(T_2\) is intersected with the only CI, ‘cd’ present in the existing CI-Table. The intersection obtained is NULL and so no action is performed. No more CIs are present in the CI-Table. The processing is over and now the T-List is used to update the four tables in the HATCI algorithm. The only CI, ‘ab’ present in the T-List is inserted into the CI-Table and also the entry for \(T_2\) in the Transaction-Table is updated to reflect the fact that the only CI that is affected by the transaction, \(T_2\) is ‘ab’. Its key is \((1,1)\) because the starting item is ‘a’ (‘a’ is represented by 1) and it is the first CI that starts with ‘a’ which has been generated so far. Fig. 4.7 shows the state of the four HATCI tables after using T-List to update the tables.

The Transaction-Table in Fig. 4.7 (a) shows that, at present there are only two transactions in the window, \(T_1\) and \(T_2\). Now, ‘s’ points to \(T_1\) and ‘e’ points to \(T_2\). They affect only the CIs, ‘cd’ with key \((3,1)\) and ‘ab’ with key \((1,1)\). The CIs generated so far ‘cd’ and ‘ab’ are listed in the CI-Table shown in Fig. 4.7 (b) along with their support one. The Superset-Table and the Subset-Table shown in Fig. 4.7 (c) and (d) show that both the CIs, ‘cd’ and ‘ab’ with keys \((3,1)\) and \((1,1)\) respectively do not have any supersons or subsets in the existing CI-Table.

![Fig. 4.7 State of HATCI tables after \(<T_2, ab>\)](image-url)
After transaction \(<T_3, abc>\)

First the itemset ‘abc’, in the transaction \(T_3\), is checked against the CIs in the existing CI-Table shown in Fig. 4.7 (b). No match is found. So it is inserted into the temporary T-List with one as its Support Count (SC). Now intersection with each CI in the CI-table is performed. On intersecting with the first CI ‘cd’, the intersection obtained is ‘c’ and its key is (3,2) as the starting item is ‘c’ (represented as 3) and this is the second CI that starts with ‘c’. The T-List at this point contains only ‘abc’. So, ‘c’ is inserted as a CI into the T-List with SC=2. Since ‘c’ is found by intersecting \(T_2\) with ‘cd’ whose support count is 1, the SC(‘c’) = SC(‘cd’) + 1 = 2. Immediately the existing Subset-Table in Fig. 4.7 (d) is checked to find whether ‘cd’ has any subsets. If any subsets are found, their SC should be incremented by 1. But since ‘cd’ does not have any subset so far, no updates are performed.

Also ‘c’ is not equal to the new transaction ‘abc’. So, ‘c’ is added as subset of ‘abc’ and ‘abc’ is added as superset of ‘c’. ‘c’ is not equal to the CI, ‘cd’ which is being processed at present. So ‘c’ is added as subset of ‘cd’ and ‘cd’ is added as superset of ‘c’. The CI, ‘cd’ does not have any supersets and thus no more processing is needed with ‘cd’.

The next CI in the CI-Table is ‘ab’. On intersecting this CI with the new transaction, ‘abc’, the intersection obtained is ‘ab’ itself. Add ‘ab’ to the T-List with SC=2 (support count of the CI ‘ab’ + 1). No more processing is needed as ‘ab’ does not have any supersets or subsets. The intersection found, ‘ab’ is not the same as the transaction, ‘abc’. So ‘ab’ is added as subset of ‘abc’ and ‘abc’ is added as superset of ‘ab’.

There are no more CIs in the CI-Table. The T-List at this point has the CIs ‘abc’ with SC=1, ‘c’ with SC=2 and ‘ab’ with SC=2. All elements in the T-List are inserted into the CI-Table if not present already. If present, the SC is updated. Here ‘abc’ and ‘c’ are new CIs and they are inserted into the CI-Table. ‘ab’ is already present as a CI so its SC is updated to 2. Fig. 4.8 shows the updated HATCI tables after processing the transaction, \(T_3\) with ‘s’,”e’ pointing to \(T_1\) and \(T_3\) respectively indicating that \(T_1\) is the oldest transaction and \(T_3\) is the most recent transaction seen so far in the data stream.
After transaction $T_4$, abc

The itemset \textit{abc} in the transaction $T_4$ is already present as a Closed Itemset (CI) in the CI-Table shown in Fig. 4.8(b). So, its Support Count (SC) is incremented by 1 and the updated SC=2. Also \textit{ab} with key (1,1) and \textit{c} with key (3,2) are subsets of \textit{abc} as seen from the Subset-Table shown in Fig. 4.8(d). So their SC is also incremented by 1 and the updated SC=3 for both \textit{ab} and \textit{c}. No more processing is needed. An entry is made in the Transaction-Table for $T_4$ and the CI keys listed are the same as $T_3$. The ‘e’ pointer is updated to show that at present the most recent transaction encountered is $T_4$. The updated HATCI tables after processing transaction $T_4$ are shown in Fig. 4.9. As seen from the figure, the sliding window is now full (since the window size is set to 4 for this example) and the arrival of the next transaction in the data stream causes phase II, sliding phase of the algorithm to be activated.
Fig. 4.9 Updated HATCI tables after \(T_d, abc\)

**After transaction \(T_5, bc\)**

On arrival of \(T_5\), it is found that the sliding window is full as the Transaction-Table already contains four transactions as seen in Fig. 4.9 (a). The RemoveOldestTransaction algorithm is invoked to remove the oldest transaction, \(T_1, cd\). The first entry in the Transaction-Table is \(T_1\) with \((3,1)\) listed as the only CI-Key. So, the Support Count (SC) of the CI ‘cd’ which corresponds to the key \((3,1)\) is reduced by one. It is marked as invalid since the updated SC=0. The Subset Table indicates that the CI ‘c’ with key \((3,2)\) is a subset of ‘cd’. So, SC of ‘c’ is also reduced by one and its updated SC=2. Now from the Superset-Table it is seen that ‘c’ has a superset ‘abc’ also with SC=2. ‘c’ is no longer a CI and it is marked as invalid. All entries related to ‘cd’ and ‘c’ are removed from the four HATCI tables. The ‘s’ pointer is updated so that it points to the transaction, \(T_2\). The removal part is now over and the updated tables are shown in Fig. 4.10 after removing \(T_1, cd\) but before adding \(T_5, bc\).
Now the new transaction $<T_5, bc>$ is added into the window using the AddNewTransaction algorithm. Itemset ‘bc’ is not present in the existing CI-Table shown in Fig. 4.10 (b). So, it is added into the temporary T-List with SC=1 and key (2,1) since the first item is ‘b’ (represented as 2) and this is the first CI that starts with ‘b’. Itemset ‘bc’ is intersected with the two CIs ‘ab’ and ‘abc’ in the CI-Table. First intersection found is ‘b’ on intersecting with the CI ‘ab’. Its key is (2,2) since the first item is ‘b’ (represented as 2) and this is the second CI that starts with ‘b’. Its SC is calculated as 4 (SC of ‘ab’+1).

The intersection ‘b’ is not equal to the transaction ‘bc’. So, ‘b’ is added as subset of ‘bc’ and ‘bc’ is added as superset of ‘b’. Also ‘b’ is not equal to ‘ab’, the current CI being processed. So, ‘b’ is added as subset of ‘ab’ and ‘ab’ is added as superset of ‘b’. The CI, ‘ab’ being processed has a superset ‘abc’ as seen from the Superset-Table shown in Fig. 4.10 (c). So, ‘b’ is added as subset of ‘abc’ and ‘abc’ is added as superset of ‘b’.

Next intersection is performed with the CI ‘abc’. The intersection obtained is ‘bc’ and its SC=3 (SC of ‘abc’+1). Intersection ‘bc’ is not equal to CI ‘abc’. So, ‘bc’ is added as subset of ‘abc’ and ‘abc’ is added as superset of ‘bc’. Intersection ‘bc’ is already present in the T-List and so only its SC is updated as
three. The CI ‘abc’ does not have any superset and the process is complete. Now the itemsets ‘b’ with SC=4 and ‘bc’ with SC=3 are inserted into the CI-Table. An entry for T5 is stored in the place released after deleting T1 by circularly incrementing the ‘e’ pointer and the keys (2,1) and (2,2) are added as the CI-Keys of T5. The updated HATCI tables after inserting T5 are shown in Fig. 4.11.

![Fig. 4.11 Updated HATCI tables after \langle T_5, bc \rangle](image)

**4.5. PERFORMANCE EVALUATION**

The proposed Closed Frequent Itemset Retrieval system is implemented in JAVA 1.6. Two datasets are used for evaluating the performance of the retrieval system. The two datasets used are Mushroom and Chess datasets.

**4.5.1. Dataset Description**

*Mushroom dataset:* This dataset includes descriptions of hypothetical samples corresponding to 23 species of gilled mushrooms in the Agaricus and Lepiota Family. Each species is identified as definitely edible, definitely poisonous, or of unknown edibility and not recommended. This latter class was combined with the poisonous one. The Guide clearly states that there is no simple rule for determining the edibility of a mushroom; no rule like "leaflets three, let it be" for Poisonous Oak and Ivy.
**Chess dataset:** This dataset contains domain theories; in addition to that, it contains some useful prolog routines. One routine takes a generic chess board description and a domain theory name, and produces a prolog state description suitable for use with the given domain theory.

### 4.5.2 Evaluation Measures

Two measures are used for analyzing the performance of the proposed system. They are given below.

1. Number of Retrievals
2. Computational Time

Number of Retrievals evaluates the total number of Closed Frequent Itemsets (CFIs) that are retrieved from the data stream. Normally, in a data stream, the number of CFIs is very much less than that of the Frequent Itemsets. The Computational Time indicates the total time that is taken for retrieving the CFIs. The time for retrieving should be low for a good system.

### 4.5.3 Experimental Results

The experimental results of the proposed Closed Frequent Itemset Retrieval system are presented in this section. The computational time for retrieval and the number of retrievals of Closed Frequent Itemsets using the proposed method are presented in this section. The following figures Fig. 4.12 and Fig. 4.13 show the graphs comparing the number of Frequent Itemsets with the number of Closed Frequent Itemsets using both datasets Chess and Mushroom.

![Graph](image.png)

**Fig. 4.12 Number of FIs Vs CFIs – Chess dataset**
From the figures Fig. 4.12 and Fig. 4.13, it is clear that the number of closed frequent itemsets is very much less than that of the frequent itemsets. This justifies the fact that closed frequent itemsets is a compact representation of the frequent itemsets and reduces the storage needed. This fact is proved using both the datasets Chess and Mushroom. When the support threshold (%) is a low value, more number of Closed Frequent Itemsets is retrieved.

![Graph](image)

Fig. 4.13 Number of FIs Vs CFIs - Mushroom dataset

**Number of retrievals of Closed Frequent Itemsets**

For getting the number of retrievals of Closed Frequent Itemsets, the proposed HATCI algorithm is compared with the MOMENT algorithm which is a benchmark algorithm for closed frequent itemset retrieval using sliding window model. The figure for comparing the number of closed frequent itemsets retrieved by the MOMENT algorithm and proposed HATCI algorithm is given in Fig. 4.14 and Fig. 4.15 using the Chess dataset and Mushroom dataset respectively.

Fig. 4.14 and Fig. 4.15 illustrate the results of comparing the number Closed Itemset retrieved using HATCI algorithm to the existing implementation of the MOMENT. The x-axis specifies the window size, while the y-axis specifies a value to show the number of CFIs retrieved. HATCI algorithm retrieves more number of CFIs when compared to MOMENT algorithm. Thus it is proved beyond doubt that the HATCI algorithm provides more accurate Closed Frequent Itemset retrieval. Accuracy here refers to the fact that HATCI retrieves some of the CFIs that are not retrieved by the MOMENT algorithm.
From Fig. 4.14 and Fig. 4.15, it is proved that the proposed HATCI algorithm has the ability to retrieve more number of Closed Frequent Itemsets when compared with the MOMENT algorithm by using both the Chess and Mushroom datasets. The number of CFI retrievals varies with the window size. If the Window size is small, then low number of Closed Frequent Itemsets is retrieved; as window size increases more number of Closed Frequent Itemsets is retrieved.

**Computational Time for the Closed Frequent Itemsets Retrieval**

For getting the retrieval execution time, the MOMENT algorithm and the proposed HATCI algorithm are compared using the two datasets Chess and
Mushroom. The graph for comparing the computation time of MOMENT algorithm and proposed HATCI algorithm is given in Fig. 4.16 and Fig. 4.17.

![Graph for comparing computation time of MOMENT and HATCI algorithms](image)

**Fig. 4.16 Computational time of MOMENT and HATCI Algorithms**

- Chess dataset

From Fig. 4.16 and Fig. 4.17, it is proved that the proposed HATCI Algorithm requires less execution time when compared with the MOMENT algorithm by using both datasets Chess and Mushroom. It is understood from these figures that the computational time changes according to the window size. When the window size is increased, the Computational time for retrieving Closed Frequent Itemsets will also increase. From this section, it can be proved that the proposed HATCI algorithm can perform better than the existing algorithms by making comparisons with the benchmark algorithm, MOMENT.

![Graph for comparing computation time of MOMENT and HATCI algorithms](image)

**Fig. 4.17 Computational time of MOMENT and HATCI Algorithms**

- Mushroom dataset
The Closed Itemsets (CI) in the current sliding window are generated and stored in the CI-Table along with three other supporting tables, namely the Superset-Table, the Subset-Table and the Transaction-Table. For generating and maintaining these tables, a sliding window model was used. The Closed Frequent Itemsets (CFIs) are extracted and returned to the user whenever the request arrives. When the user requests for the CFIs, the CI-Table is scanned sequentially, and the CIs with support count greater than or equal to the user specified Support threshold are extracted and returned. The experiments conducted proved beyond doubt that the proposed HATCI algorithm is better than the MOMENT algorithm in aspects of computational time and the number of CFIs retrieved.

4.6 CASE STUDY

In web applications such as search engines, pages that are produced and served to users are often cached in anticipation of identical requests in the near future. Successful caching can reduce a search engine’s hardware requirements, as well as reduce the latency experienced by searchers. Exploring pages on the Web often takes a long time because of the time it takes for the browser to retrieve data from a server. In general, a cache is often used to enhance performance of data access. A cache is a small but fast-to-access place in memory compared to the rest of memory which is often large but takes a long time to access. A cache stores data which hopefully will be accessed soon in the future. The limited cache size and its high cost make it essential to select a suitable set of pages to be cached. It is a fact that pre-fetching and caching all documents with access probabilities greater than a given threshold value does not always lead to the minimum access delay. This is only true for the case when clients have very high access speeds. For the other cases, the access delay improves with the increase in the number of pre-fetched documents until a certain point, after which the trend is reversed.

If a large number of pre-fetch requests are made, then the network will become congested and Web accesses will become very slow. Thus pre-fetching preformed to reduce latency experienced by searchers may lead to increased latency when number of pre-fetch requests becomes high. Therefore if pre-fetching is to be effective, the number of pre-fetches made has to be limited and the number of unnecessary requests must be kept at a minimum. Clearly the problem is not with
pre-fetching, it's with deciding which pages to pre-fetch. In the previous chapter, use of Frequent Itemset (FI) mining algorithms to determine the frequently visited collection of web pages [29] was discussed. The frequently visited web pages could then be cached to reduce the latency experienced by users. The number of FIs identified is very large at low thresholds. Also if a k-itemset is frequent, all its $2^k - 1$ subsets are also frequent. So pre-fetching all the FIs thus identified is not practical. Most of the FIs have the same web pages listed repeatedly. Sending requests for pre-fetching these pages repeatedly increases the number of pre-fetch requests but reduces the effectiveness of pre-fetching. The effectiveness is reduced because the same pages are requested repeatedly and also the increase in number of pre-fetch requests leads to increased network congestion. In order to overcome this problem, Closed Frequent Itemsets (CFIs) may be identified from the web logs and since they are very less in number compared to the FIs but have all information about the FIs, the number of requests for pre-fetching web pages will also be less resulting in improved latency experienced by users.

When association rules [24] are formed from the FIs as suggested in the previous chapter, many redundant rules are generated. This is also because of the repeated appearance of the same web page in many FIs. Such redundant rules may be reduced by identifying the compressed form of FIs. The rules formed from the CFIs do not suffer from this redundancy problem. Thus generation of CFIs from web logs helps to bring down the number of pages to be cached and can serve many users. The CFIs help to find the number of frequently visited web pages without repetitions and they are made available to all users in the limited cache memory available.

4.7 LIMITATIONS OF HATCI

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 employed in the proposed HATCI algorithm 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. In the next chapter, a CFI
mining algorithm that employs a landmark window model to mine the CFIs from a data stream is proposed.

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. Thus landmark windows are preferred only when the application demands it, mainly due to the above said drawbacks. Examining web server log files to determine the user access patterns 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.