3.1 INTRODUCTION

Most of the known rule mining algorithms try to find global patterns in the underlying dataset. Algorithms are designed to find set of items that are frequent in the whole dataset and then find associations that hold among the frequent items. But local patterns are also interesting and they may provide useful information in the decision making process. We propose an algorithm to find local patterns in the underlying dataset.

This study is related mainly to temporal market-basket data but can be extended to other temporal datasets also. Here each transaction contains a set of items bought by a customer at one time together with a time-stamp indicating the time of transaction. Global frequent sets mining algorithms count the support of the items sets throughout the life-span of the whole dataset. In large volume of temporal data, as are used for data mining purposes, information may be found related to products that did not necessarily exist throughout the data-gathering period. So some products may be found which, at the moment of performing the mining, have already discontinued. There may be also new products that were
introduced after the beginning of the gathering. Some of these new products should participate in the associations, but may not be included in any rule because of support restrictions. In [32] the authors address this problem and suggest a new algorithm that computes the support of an item not with respect to the life-span of the whole dataset but with respect to the life-span of the item. The life-span of an item is defined as the time period between the two time-stamps \textit{lastseen} and \textit{firstseen} where \textit{firstseen} is the time-stamp at which the item has first appeared in the transactions and \textit{lastseen} is the time-stamp in which the item was last seen in the transactions.

However there may be items that appear in the transactions for a short time period, but within that period they appear frequently and then they disappear for a long period and appear again. For such items when their supports are calculated with respect to their life-span they may become infrequent. But extracting those sets together with the time slots in which they are frequent is important. The algorithm proposed here is designed to detect such sets.

Another problem is that even if an item set is frequent within its life-span, the density of occurrences of the items in the transactions may vary. Within the life-span there may be time slots when the density of occurrences is high and some time slots where the density is very low. For such items the algorithm proposed calculates their supports within the dense regions individually. The time gaps in which the item has not appeared or has appeared very rarely are not counted. Identifying such dense time slots will certainly provide us knowledge about the dataset and the items concerned. The algorithm [2] proposed, extracts such dense
time slots for the item sets within certain thresholds given as input by the user. 
Associated with those item sets the algorithm produces as output a list of time 
slots or time intervals in which the items are frequent together with their support 
values for each time interval separately. We say that the items are locally frequent 
within the time-intervals in the list.

If due to external influences there are any surprising changes taking place in the 
market, then this algorithm will be able to report the sudden changes. This 
algorithm will be able to extract all frequent sets discovered by the algorithm 
proposed in [32] and some more sets that are frequent according to the way in 
which we are defining locally frequent sets. For frequent sets discovered in [32], 
this method may break down the life-span of the items into dense and sparse 
intervals and report the supports in the dense regions only.

Another advantage of the algorithm is that it is incremental in nature. When new 
transactions are added it is not necessary to process the whole dataset again. 
Obviously the newly added transactions will have time-stamps that are after the 
last time-stamp of the earlier data. Processing of the newly added data will either 
add more time intervals to already detected frequent sets or will detect frequent 
sets with time periods within the duration of newly added data or extend the last 
period associated with an item set. Before describing the algorithm, in section-
3.2 we describe the terms, notations and symbols used in the algorithm. The 
proposed algorithm for finding locally frequent item sets is described in section-
3.3. Section-3.4 reports on the implementation done by us and also describes the 
results obtained. In section-3.5 we describe our proposed algorithm for finding
local association rules after finding the locally frequent sets. In section-3.6, we
discuss about the complexity of the algorithms and finally in section-3.7, a brief
conclusion of this chapter. In describing the algorithms although we have used
pseudo codes but C language constructs are used in many places to make the
algorithms simple.

3.2 TERMS, NOTATIONS AND SYMBOLS USED

Let $T = <t_0, t_1, .........>$ be a sequence of time-stamps over which a linear
ordering $<$ is defined where $t_i < t_j$ means $t_i$ denotes a time which is earlier than $t_j$
Let $I$ denote a finite set of items and the transaction dataset $D$ is a collection of
transactions where each transaction has a part which is a subset of the item set $I$
and the other part is a time-stamp indicating the time in which the transaction had
taken place. We assume that $D$ is ordered in the ascending order of the time-
stamps. For time intervals we always consider closed intervals of the form $[t_1, t_2]$ where $t_1$ and $t_2$ are time-stamps. We say that a transaction is in the time interval
$[t_1, t_2]$ if the time-stamp of the transaction say $t$ is such that $t_1 \leq t \leq t_2$

We define the local support of an item set in a time interval $[t_1, t_2]$ as the ratio of
the number of transactions in the time interval $[t_1, t_2]$ containing the item set to
the total number of transactions in $[t_1, t_2]$ for the whole dataset $D$. We use the
notation $Supp_{[t_1, t_2]}(X)$ to denote the support of the item set $X$ in the time interval
$[t_1, t_2]$. Given a threshold $\sigma$ we say that an item set $X$ is frequent in the time
interval $[t_1, t_2]$ if $Supp_{[t_1, t_2]}(X) \geq (\sigma/100)* tc$ where $tc$ denotes the total number of
transactions in $D$ that are in the time interval $[t_1, t_2]$. We say that an association
rule $X \Rightarrow Y$, where $X$ and $Y$ are item sets holds in the time interval $[t_1, t_2]$ if and only if given threshold $\tau$, 
\begin{equation}
\text{Supp}_{[s,d]}(X \cup Y) / \text{Supp}_{[s,d]}(X) \geq \tau / 100.0
\end{equation}
and $X \cup Y$ is frequent in $[t_1, t_2]$. In this case we say that the confidence of the rule is $\tau$.

### 3.3 Algorithm Proposed

#### 3.3.1 Generating Locally Frequent Sets

While constructing locally frequent sets, with each locally frequent set a list of time intervals is maintained in which the set is frequent. Two thresholds $\minthdl$ and $\minthd2$ are used for this and these are given as input by the user. During the execution of the algorithm while making a pass through the dataset, if for a particular item set the time gap between its current time-stamp and the time when it was last seen is less than the value of $\minthdl$ then the current transaction is included in the current time interval under consideration; otherwise a new time interval is started with the current time-stamp as the starting point. The support count of the item set in the previous time interval is checked to see whether it is frequent in that interval or not and if it is then it is added to the list maintained for that set. Also for the locally frequent sets, a minimum period length is given by the user as $\minthd2$ and time intervals of length greater than or equal to this value are only kept. If $\minthd2$ is not used than an item appearing once in the whole dataset will also become locally frequent.
ALGORITHM TO COMPUTE \( L_1 \), THE SET OF ALL LOCALLY FREQUENT ITEM SETS OF SIZE-1

For each item while going through the dataset we always keep a time-stamp called \( \text{lastseen} \) that corresponds to the time when the item was last seen. When an item is found in a transaction and the time-stamp is \( tm \) and the time gap between \( \text{lastseen} \) and \( tm \) is greater than the current minimum threshold given, then a new time interval is started by setting \( \text{start} \) of the new time interval as \( tm \) and \( \text{end} \) of the previous time interval as \( \text{lastseen} \). The previous time interval is added to the list maintained for that item provided that the duration of the interval and the support of the item set in that interval are both greater than the minimum thresholds specified for each. Otherwise \( \text{lastseen} \) is set to \( tm \), the counters maintained for counting transactions are increased appropriately and the process is continued.

Following is the algorithm to compute \( L_1 \), the list of locally frequent sets of size-1. Suppose the number of items in the dataset under consideration is \( n \) and we assume an ordering among the items.

ALGORITHM 3.1

\[
C_1 = \{(i_k, tp[k]) : k = 1, 2, \ldots, n\}
\]

where \( i_k \) is the \( k \)-th item and \( tp[k] \) points to a list of time intervals initially empty

for \( k = 1 \) to \( n \) do

set \( \text{lastseen}[k], \text{icount}[k], \text{ctcount}[k] \) and \( \text{ptcount}[k] \) to zero

for each transaction \( t \) in the database with time stamp \( tm \) do

if \( (t_k) \subseteq t \) then

if \( (\text{lastseen}[k] = 0) \)

\( \text{lastseen}[k] = \text{firstseen}[k] = \text{tm}; \)

\( \text{icount}[k] = \text{ptcount}[k] = \text{ctcount}[k] = 1; \)

else if \( (\text{tm} - \text{lastseen}[k] < \text{minthd}) \)

\]
Three support counts \textit{icount}, \textit{ctcount} and \textit{ptcount} are maintained with each item. When an item is first seen then these are initialized to 1. For each item while making a pass through the dataset when a transaction containing the item is found then \textit{icount} for that item is increased. To see whether an item is frequent in an interval the total number of transactions in that interval will have to be counted. For this with each item two counts \textit{ptcount} and \textit{ctcount} are kept. The value of \textit{ctcount} increases with each transaction but \textit{ptcount} changes its value only when a transaction containing an item is found within \textit{minthd1} from current value of \textit{lastseen} and then it takes the value of \textit{ctcount}. When an item is not seen for more than \textit{minthd1} time distance from \textit{lastseen} then the value of \textit{ptcount} is used to compute the percentage support count of the item between \textit{firstseen} and \textit{lastseen}. If the count percentage of an item in a time interval is greater than the minimum threshold then only the set is considered as a locally frequent set and the locality is the time interval. When a new interval is started for an item then the three counts again start from 1.
as computed above will contain all locally frequent sets of size-1 and with each set there is associated an ordered list of time intervals in which the set is frequent. Then \textit{A-priori} candidate generation algorithm is used to find candidate frequent sets of size-2. This level-wise frequent set computation goes on till a level becomes empty. In the candidate set generation phase before two item sets are joined to generate candidates for the next level, pair-wise intersection of the time intervals maintained with the two item sets is computed. If this results in an empty-interval list then the sets are not operated with the join operation. This way certain candidate item sets are pruned in the generation phase itself. The same technique is done when it is to be checked whether all subsets of a given set appears in the previous level. When subsets of a set are found then pair-wise intersection of the corresponding time intervals associated with the subsets are done successively. If during the search operation this list turns out to be empty then the corresponding set is pruned.

Using this concept we describe below the modified \textit{A-priori} algorithm for the problem under consideration.

\textbf{ALGORITHM 3.2}

\begin{verbatim}
Modified A-priori
initialize
k = 1;
C_1 = all item sets of size-1
L_1 = \{ frequent item sets of size-1 where with each itemset \{a_i\} a list tp[i] is maintained which gives all time intervals in which the set is frequent \}
L_1 is computed using algorithm 1.1 */
for(k = 2; L_{k-1} \neq \phi; k++) do
{ C_k = apriorigen(L_{k-1}) /* same as the candidate generation method of the A-priori algorithm setting tp[i] to zero for all i*/
prune(C_k);
drop all lists of time intervals maintained with the sets in C_k
\}
\end{verbatim}
Compute $L_k$ from $C_k$.  
\[ L_k \text{ can be computed from } C_k \text{ using the same procedure used for computing } L, \]
\[ k = k + 1 \]
\}

Answer = $\bigcup_{t} L_k$

Prune($C_k$)
\{Let $m$ be the number of sets in $C_k$ and let the sets be $s_1, s_2, \ldots, s_m$. Initialize the pointers $tp[i]$ pointing to the list of time-intervals maintained with each set $s_i$ to null for $i = 1 \text{ to } m$ do 
\{for each $(k-1)$ subset $d$ of $s_i$ do 
\{if $d \subseteq L_{k-1}$ then 
\{$C_k = C_k - \{s, tp[i]\}$, break;\} 
else 
\{if ($tp[i] == null$) then set $tp[i]$ to point to the list of time intervals maintained for $d$ 
else \{ take all possible pair-wise intersection of time intervals one from each list, 
one list maintained with $tp[i]$ and the other maintained with $d$ and take this as the list for $tp[i]$ 
delete all time intervals whose size is less than the value of minthd2 
if $tp[i]$ is empty then 
\{$C_k = C_k - \{s, tp[i]\}$; 
break; \} \} \}\}
\}\}
\}

3.4 IMPLEMENTATION

3.4.1 DATA STRUCTURE USED

The candidate generation and the support counting processes require an efficient data structure in which all candidate item sets are stored since it is important to efficiently find the item sets that are contained in a transaction or in another item set. In general two data structures namely Hash-tree and Trie (or Prefix-tree) are used for this purpose. In our work, we have used the Trie-data structure.

TRIE

All the items under consideration are marked as 1, 2, \ldots, $n$ where $n$ is the total number of items. We also assume that the items in the transactions are ordered in
the same ordering. In a trie, every \( k \)-item set has a node associated with it, as does its \( (k-1) \)-prefix. The empty item set is the root node. All the 1-item sets are attached to the root node as its children. Every other \( k \)-item set is attached to its \( (k-1) \)-prefix. Every node represents an item set. The node stores the last item in the item set it represents, pointer to childlist, pointer to parent, pointer to list of time intervals where the item set is frequent and pointer to its right sibling. The siblings of each node are implemented as linked list. So, every level consists of a collection of lists. In our implementation all the list in a level are again maintained as a list of lists. The level-1 is a single list consisting of all the 1-item sets.

At a certain iteration \( k \), all the candidate \( k \)-item sets are stored at depth \( k \) in the trie. In order to count the supports of item sets for a particular level, the item sets represented by the nodes are found by moving upwards to the root using parent pointers.

Also the join step of the candidate generation procedure becomes very simple when a trie is being used. Since all item sets of size-\( k \) with the same \( (k-1) \)-prefix are represented as a linked list (all are children of the same node), to generate all candidate \( k \)-size item sets with \( (k-1) \)-prefix \( X \) we simply copy all right siblings of the node representing \( X \) and add them as child of \( X \). Candidate generation procedure also computes pair-wise intersection of the list of time intervals associated with the two item sets that are joined to get the candidate. If intersection of the lists of time intervals is found to be empty or the length of all intervals in the list is found to be less than \( \text{minthd}_2 \) then the newly added node is
removed. It also includes *A-priori* pruning step, which is required to check whether all the subsets of the candidates are present in the previous level.

To illustrate the structure we consider the items 1, 2, 3, 4.

![Trie Structure](image)

**FIG-3.1 TRIE STRUCTURE**

The trie structure is shown in the above figure. The number in the nodes represents the last item of the item sets represented by the nodes. The items in the item set represented by a node are obtained by moving upwards from the node to the *root* using parent pointers. For example, in the second level we have item sets \{1, 2\}, \{1, 3\}, \{1, 4\}, \{2, 3\}, \{2, 4\}, \{3, 4\}, in the third level we have item sets \{1, 2, 3\}, \{1, 2, 4\}, \{1, 3, 4\} and \{2, 3, 4\} and in the fourth level the item set \{1, 2, 3, 4\}. We have implemented in this way because this facilitates convenient candidate generation.
Programs were written in C++ for implementing the algorithm described in section-3.3. Since singly linked list is the structure used extensively throughout our implementation, we wrote class definition and functions for lists using templates as shown below. Lists were implemented in the circular fashion with the last node pointing to the first node of the list. In this case a pointer to the last node of a list is sufficient to denote a particular linked list.

```cpp
template <class T>

struct Tnode
{
    Tnode *next;
    T *Tptr;
    Tnode() {next = 0; Tptr = 0;}
    Tnode(T *a) {next = a; Tptr = 0;}
    Tnode(const Tnode<T> &nd) {this = &nd;}
    Tnode& operator = (const Tnode<T> &rtd);
};

class list
{
    Tnode<T> *last;
    public:
        void insert(T *a);
        void append(T *a);
        T* get();
        list() {last = 0;}
        void drop();
        void destroy();
    ~list() {destroy();}
    friend class listiter<T>;
};

An iterator was defined for traversing through the list.

class listiter
{
    Tnode<T> *ce,
    list<T> *cs,
    public:
    listiter(list<T> &s) {cs = s; ce = s->last;}
    listiter(list<T> &s, Tnode<T> *t) {cs = s; ce = t;}
    T* operator()()
    {Tnode<T> ret = ce ? (ce = ce->next) : 0;
    
    57
```
if (ce == cs->last) ce = 0;
    return ret ? ret->Tptr : 0;
};

We have implemented the trie-data structure by writing class definitions as shown below.

```cpp
class trie_node
{
    itemtype ino;
    list<inter_count> interlist;
    list<trie_node> childlist;
    trienptr par;
    tempinfo *tp;
};
```

```cpp
class trie
{
    trie_node *root;
    int nlevel;
    list<list<trie_node>> level[maxlevel];
};
```

```cpp
class inter_count
{
    date start, end;
    int count;
};
```

```cpp
struct tempinfo
{
    int icount, ptcount, ctcount;
    date lseen, fseen;
};
```

The class `inter_count` is used to store time intervals stored as two dates representing the boundaries of the interval and an integer member count used to store the support count in that interval for an item set with which it is associated.
Each node of the trie is an object of class trie_node. It holds an item number, which is the last item of the item set represented by the node. The remaining items are the item numbers appearing in the nodes along the unique path from the node to the root of the trie. interlist is the list of time intervals associated with the item set represented by the node in which it is locally frequent. childlist is the list containing all the children of the node. par is a pointer to the parent of the node. With each node in the trie we have some temporary information stored, which are needed for calculating the supports of the corresponding item sets. These are the variables icount, ptcount, ctcount, lastseen and firstseen described in section-3.3. After counting the supports of the item sets in a level this information associated with the nodes of this level are deleted and the corresponding spaces are released. A trie is represented by a pointer to the root of the trie. The variable nlevel stores the number of levels of the trie. All the nodes in a particular level are again linked up as a linked list of lists.

3.4.2 DESCRIPTION OF THE MAIN FUNCTIONS AND PROGRAM

i) addlevel1(): This function adds the first level to the trie. This level represents all 1-item set.

ii) suppcount1(): This function is used to find the support of the first level candidate sets. It goes through the dataset once and calculates the frequencies of all the 1-item sets. It keeps all the 1-size frequent sets and
each item set is associated with a list of time intervals where the item set is frequent.

iii) gencandidate(): This is used for candidate generation. It also includes one type of pruning where the pair-wise intersections of the time intervals of the sets that are joined are computed. It uses the interval list pointers stored with the nodes.

iv) allsubsetintrie(): This function is called from gencandidate to check whether all the subsets of a candidate are present in the previous level.

v) display(): This function displays the trie at any time.

vi) suppcount(): The function is used to calculate the supports of the item sets in a level other than the first level.

vii) cleanlist(): This function is used to remove all nodes which are infrequent.

Data structure and some of the functions that were used for performing the necessary operations on calendar dates are

\emph{class date}

\{ \emph{short day, month, year;} \};
i) is_leap_year(): This function is used to check an year in the input data is leap year. If leap year the February will be of 29 days otherwise of 28 days.

ii) countdays: This is used count the number of days associated with a frequent.

iii) operator<=(): operator over loading done for comparing two dates.

Some of the functions written for performing necessary operations on the lists of time intervals are

i) inter_count*intersect(): This function is used to compute the intersection of two time intervals.

ii) list<inter_count>*intersect(): This function is used to compute the intersection of two lists of time intervals.

3.4.3 DESCRIPTION OF THE DATASETS

To compute the local supports of a collection of item sets, we need to access the dataset. Since the datasets tend to be very large, it is not always possible to store them in the main memory.

An important consideration in most algorithms is the representation of the transaction dataset. Conceptually, such a dataset can be represented by a binary two-dimensional matrix in which every row represents an individual transaction and columns represent the items in \( I \). Such a matrix can be implemented in
several ways viz. *horizontal data layout* and *vertical data layout*. But in our case as the dataset temporal transaction dataset, each transaction is attached with time-stamp e.g. calendar-date. So conceptually the dataset consists of numeric as well binary attributes where the time-stamp consists of three columns viz. day, month and year having numeric values within a specified ranges and rest of the attributes are as usual binary. A sample of such dataset is given below:

<table>
<thead>
<tr>
<th>time-stamps</th>
<th>Items</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Day</strong></td>
<td><strong>month</strong></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>15</td>
<td>1</td>
</tr>
<tr>
<td>20</td>
<td>1</td>
</tr>
</tbody>
</table>

**TABLE 3.1: Layout of Dataset D**

3.4.4 EXAMPLES OF DATASETS

For the experiments we performed in this thesis, we used two datasets with different characteristics. We have experimented using one real dataset, and one synthetic dataset generated by the program provided by the *Quest research group* at IBM Almaden. This generator can be downloaded from [http://www.almaden.ibm.com](http://www.almaden.ibm.com). The synthetic dataset T10I4D100K (4.0 MB) is available from FIMI’03 website at [http://fimi.cs.helsinki.fi/testdata.html](http://fimi.cs.helsinki.fi/testdata.html). The retail dataset was donated by Tom Brijs and contains the retail *market basket data* from an anonymous Belgian retail store. As the datasets are non-temporal so they cannot be directly used by our method. We have incorporated the temporal feature in the datasets so that they can be temporal datasets and be handled by our method. A program was developed for this. The program takes as input a starting date and two values for the minimum and maximum number of transactions per
day. A number between these two limits are selected at random and that many consecutive transactions are marked with the same date. So that many transactions have taken place on that day. This process starts from the first transaction to the end by marking the transactions by consecutive dates (assuming that the market remains open on all week days). The datasets used, the number of items, the number of transactions in each dataset and the minimum, maximum and average length of transactions are reported in the table below:

<table>
<thead>
<tr>
<th>Dataset</th>
<th>#Items</th>
<th>#Transactions</th>
<th>Min T</th>
<th>Max T</th>
<th>Avg T</th>
</tr>
</thead>
<tbody>
<tr>
<td>T10I4D100K</td>
<td>942</td>
<td>100000</td>
<td>4</td>
<td>77</td>
<td>39</td>
</tr>
<tr>
<td>Retail dataset</td>
<td>17000</td>
<td>88162</td>
<td>1</td>
<td>52</td>
<td>9</td>
</tr>
</tbody>
</table>

**TABLE 3.2: Dataset Characteristics**

The experiments were conducted at a Pentium-IV Machine with 1.6 GHzs speed, 512 MB RAM, 40 GB hard disc, running PCQ Linux 8.0.

### 3.4.5 RESULTS OBTAINED

In this section we discuss in details our results along with comparative study with results obtained by Ale et al's method. The dataset used for this purpose are the two datasets mentioned above.

With Retail dataset and thresholds $\text{minthd1} = 15$ days, $\text{minthd2} = 60$ days and $\text{min_sup sigma} = 5\%$, we obtained the following results:
1) By Ale et al's method

<table>
<thead>
<tr>
<th>Levels</th>
<th>Frequent item sets</th>
<th>Time intervals</th>
<th>Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Level-1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>[2-1-2000, 22-3-2003]</td>
<td>17%</td>
</tr>
<tr>
<td></td>
<td>38</td>
<td>[2-1-2000, 22-3-2003]</td>
<td>17.7%</td>
</tr>
<tr>
<td></td>
<td>39</td>
<td>[2-1-2000, 22-3-2003]</td>
<td>57.5%</td>
</tr>
<tr>
<td></td>
<td>41</td>
<td>[2-1-2000, 22-3-2003]</td>
<td>16.9%</td>
</tr>
<tr>
<td></td>
<td>48</td>
<td>[2-1-2000, 22-3-2003]</td>
<td>47.8%</td>
</tr>
<tr>
<td></td>
<td>65</td>
<td>[2-1-2000, 22-3-2003]</td>
<td>5%</td>
</tr>
<tr>
<td>Level-2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>32, 39</td>
<td>[2-1-2000, 22-3-2003]</td>
<td>9.6%</td>
</tr>
<tr>
<td></td>
<td>32, 48</td>
<td>[2-1-2000, 22-3-2003]</td>
<td>9%</td>
</tr>
<tr>
<td></td>
<td>38, 39</td>
<td>[2-1-2000, 22-3-2003]</td>
<td>11.7%</td>
</tr>
<tr>
<td></td>
<td>38, 48</td>
<td>[2-1-2000, 22-3-2003]</td>
<td>9%</td>
</tr>
<tr>
<td></td>
<td>39, 41</td>
<td>[2-1-2000, 22-3-2003]</td>
<td>12.9%</td>
</tr>
<tr>
<td></td>
<td>39, 48</td>
<td>[2-1-2000, 22-3-2003]</td>
<td>33%</td>
</tr>
<tr>
<td></td>
<td>41, 48</td>
<td>[2-1-2000, 22-3-2003]</td>
<td>10%</td>
</tr>
<tr>
<td>Level-3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>32, 39, 48</td>
<td>[2-1-2000, 22-3-2003]</td>
<td>6%</td>
</tr>
<tr>
<td></td>
<td>38, 39, 48</td>
<td>[2-1-2000, 22-3-2003]</td>
<td>6.9%</td>
</tr>
<tr>
<td></td>
<td>39, 41, 48</td>
<td>[2-1-2000, 22-3-2003]</td>
<td>8.6%</td>
</tr>
</tbody>
</table>

**TABLE 3.3: Results obtained by Ale et al's method**

2) By our method

<table>
<thead>
<tr>
<th>Levels</th>
<th>Frequent item sets</th>
<th>Time intervals</th>
<th>Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Level-1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>[2-1-2000, 22-3-2003]</td>
<td>17%</td>
</tr>
<tr>
<td></td>
<td>38</td>
<td>[2-1-2000, 22-3-2003]</td>
<td>17.7%</td>
</tr>
<tr>
<td></td>
<td>39</td>
<td>[2-1-2000, 22-3-2003]</td>
<td>57.5%</td>
</tr>
<tr>
<td></td>
<td>41</td>
<td>[2-1-2000, 22-3-2003]</td>
<td>27.6%</td>
</tr>
<tr>
<td></td>
<td>48</td>
<td>[2-1-2000, 22-3-2003]</td>
<td>47.8%</td>
</tr>
<tr>
<td></td>
<td>65</td>
<td>[2-1-2000, 22-3-2003]</td>
<td>5%</td>
</tr>
<tr>
<td></td>
<td>12935</td>
<td>[10-2-2002, 24-4-2002]</td>
<td>7%</td>
</tr>
<tr>
<td>Level-2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>32, 39</td>
<td>[2-1-2000, 22-3-2003]</td>
<td>9.6%</td>
</tr>
<tr>
<td></td>
<td>32, 48</td>
<td>[2-1-2000, 29-5-2001]</td>
<td>5.9%</td>
</tr>
<tr>
<td></td>
<td>38, 39</td>
<td>[2-1-2000, 22-3-2003]</td>
<td>6%</td>
</tr>
<tr>
<td></td>
<td>38, 41</td>
<td>[2-1-2000, 22-3-2003]</td>
<td>11.7%</td>
</tr>
<tr>
<td></td>
<td>38, 48</td>
<td>[2-1-2000, 22-3-2003]</td>
<td>9%</td>
</tr>
<tr>
<td></td>
<td>39, 48</td>
<td>[2-1-2000, 29-5-2001]</td>
<td>13.5%</td>
</tr>
<tr>
<td></td>
<td>39, 12935</td>
<td>[10-2-2002, 24-4-2002]</td>
<td>5%</td>
</tr>
<tr>
<td></td>
<td>41, 48</td>
<td>[2-1-2000, 22-3-2003]</td>
<td>17.7%</td>
</tr>
<tr>
<td>Level-3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>32, 39, 48</td>
<td>[2-1-2000, 22-3-2003]</td>
<td>6%</td>
</tr>
<tr>
<td></td>
<td>38, 39, 41</td>
<td>[2-1-2000, 25-5-2001]</td>
<td>5.8%</td>
</tr>
<tr>
<td></td>
<td>38, 39, 48</td>
<td>[2-1-2000, 22-3-2003]</td>
<td>6.9%</td>
</tr>
</tbody>
</table>

**TABLE 3.4: Results obtained by our method**

64
If we closely observe the tables 3.3 and 3.4, we notice the following interesting differences:

i) Besides all the frequent item sets extracted by Ale et al's method, our method extracts some extra frequent item sets e.g. \{12935\}, \{32, 41\}, \{39, 12935\}, \{48, 12935\}, \{38, 39, 41\}. Thus our method gives much more frequent sets than Ale et al's method.

ii) Again the item set \{41\} is associated with the time interval [2-1-2000, 22-3-2003] with support 16.9 (Ale et al's method), however by our method we get the same item set associated with two different time intervals [2-1-2000, 29-5-2001] and [31-8-2002, 22-3-2003] with support 27.6% and 28.8% respectively that means there is a large time gap from 29-5-2001 to 31-8-2002 when no transaction contains the item set \{41\} again supports with respect to the individual time interval are much high. Similar observations are recorded for the item sets \{32, 41\}, \{38, 41\}, \{39, 41\}, \{41, 48\}, \{38, 39, 41\} and \{39, 41, 48\}.

Again with the synthetic dataset T10I4D100K and minthdl = 5 days, minthdl = 30 days and min_sup sigma= 4% we obtained the following results

1) By Ale et al's method
If we closely observe the tables 3.5 and 3.6, we notice the following interesting differences:

The item set \( \{1\} \) is not extracted by Ale et al’s method and so it is infrequent if its whole life-span is considered together. However by our method it has come as 8 times frequent within its life-span, so it is locally frequent 8 times within its life-span and rest of the frequent item sets extracted by both the methods are same.
3.5 GENERATING LOCAL ASSOCIATION RULES

For finding an association rule of the form \( A \Rightarrow X-A \) where \( X \) and \( A \) are item sets that holds in a time interval \([t, t']\) we need to know the supports of \( X \) and \( A \) in \([t, t']\). But the way in which supports of item sets are calculated in the algorithm proposed above the supports of \( X \) and any of its subsets \( A \) may not be available for the same time interval \([t, t']\). Suppose \( X \) is locally frequent in \([t, t']\) then its support in \([t, t']\) is known from the algorithm. \( A \) being a subset of \( X \) will also be locally frequent in \([t, t']\) but \( A \) may be locally frequent in a larger interval that contains \([t, t']\) properly. Then the local support of \( A \) will be known for the larger interval only. Thus to know the support of an item set and all its subsets in the same time interval we need to make several passes through the dataset keeping several counters for each item set for each of the intervals in which it is locally frequent. This really will be an expensive operation. In this situation we propose to calculate association rule in the following way. Suppose a set \( X \) is locally frequent in \( t_X = [t_1, t_2] \) and \( A \subseteq X \). Then \( A \) definitely will be locally frequent in some interval \( t_A = [t_1', t_2'] \) where \( t_X \) is included in \( t_A \). The algorithm proposed in section-3.3 for finding locally frequent item sets will give in its output support of \( X \) in \( t_X \) and that of \( A \) in \( t_A \). In the way in which time intervals are extracted for an item set it is clear that in the time periods from \([t_1', t_1] \) and \([t_2, t_2']\), \( X \) does not occur frequently i.e. \( X \) occurs rarely. So in calculating confidence of the rule \( A \Rightarrow X-A \) in \([t_1', t_2']\) if we consider the ratio \( \frac{\text{Supp}_{[t_1', t_2]}X}{\text{Supp}_{[t_1', t_2]}A} \) instead of \( \frac{\text{Supp}_{[t_1', t_2']}X}{\text{Supp}_{[t_1', t_2']}A} \) and lower down the minimum confidence value a bit we
hope not to miss any of the rules. This will hold for any set-subset pair i.e. for each frequent time period of a set there will be a corresponding frequent time period for the subset which includes the former time period.

The above procedure has to be carried out for all locally frequent item sets for all its frequent time periods to compute association rules that hold locally. Such a rule $A \Rightarrow X-A$ can be interpreted as that within a dense region of $A$ in $[t_1', t_2']$ there is a dense region of $X$ in $[t_1, t_2]$ where $[t_1, t_2]$ is a subinterval of $[t_1', t_2']$. We call such rules as local association rules holding in a time interval. We give below the algorithm for finding local association rules.

**ALGORITHM FOR FINDING LOCAL ASSOCIATION RULES**

**ALGORITHM 3.3**

$S$ is a set and $s$ is a subset of $S$

$\text{lists} \leftarrow \text{list of time intervals maintained with } S$

$\text{listS} \leftarrow \text{list of time intervals maintained with } s$

while $((pS = \text{listS.get()}! = \text{null})$

{$tS = pS.ti()$

$\text{suppS} = \text{support of the interval } tS$

while $((ps = \text{lists.get()}! = \text{null})$

{$ts = ps.ti()$

if $ts \geq tS$ break

}$supps \leftarrow \text{support of } s \text{ in the interval } ts$

if $(\text{supps} / \text{supps} \geq \text{minconf})$ then output

$s \Rightarrow S - s \text{ is an association rule holding in } ts$

/* this procedure will require one pass through each of the lists listS and lists */

The function $ti()$ returns the time interval associated with the corresponding node in the time intervals lists. The function $get()$ is the member function of class lists described in section-3.4.1. The algorithm is repeated to find the local association rules of the type $s \Rightarrow S - s$ for every possible subsets $s$ of a locally frequent item set $S$ starting from largest possible subset of $S$. Suppose that the size of $S$ is $n$ then
first of all, the algorithm is applied to find the local association rules from all possible $(n-1)$-size subsets of $S$ to all possible singleton set of $S$ and then from all possible $(n-2)$-size subsets of $S$ to all possible subset of $S$ of size-2 and so on. If in a particular level a rule from a particular subset of $S$ is not confident then the rules from all the subsets of that particular subset of $S$ will not be confident. This way the procedure is optimized. And the above procedure is repeated for every locally frequent item sets obtained using the algorithm-3.3.

3.6 ESTIMATES FOR THE AMOUNT OF WORK DONE FOR THE ABOVE ALGORITHMS

The problem of mining association rule for market basket problem is known to be $NP$-complete [15]. In [40], some results about the computational complexity of mining frequent item sets under combined constraints on the number of items and on the frequency threshold is given. In association rule mining the major step is to find the frequent sets. Several implementations of mining frequent item sets are available [7], [16] and [17]. For implementation of our proposed algorithm for mining locally frequent item sets, we have written our codes using a trie-based approach. We do not claim our code to be very efficient because there are scopes for improvement. Our aim is to show that the proposed algorithm extracts more rules than those extracted by other known methods. In addition to the usual (non-temporal) frequent item set mining process, the propose algorithm does some more work in keeping the track of the time-stamps and in maintaining and manipulating the lists of time intervals associated with the item sets. For counting supports for locally frequent item sets the algorithm will have to do just a few
additional operations. For each item set the value of lastseen can be accessed in O(1) time. The algorithm will have to find the time gap between lastseen and the current date and then either start a new time interval or increase the support value for the current time interval. This process will only take up a constant amount of time since for the lists always a pointer to the last node is kept.

In candidate generation in addition to the usual process this algorithm will have to compute pair-wise intersection of the two time-interval lists associated with the two item sets that are taking part in the join operation. This will take O(l+l') time where l and l' are the lengths of the time-intervals lists. But this process will also prune candidates while these are being generated. Similarly in the pruning step also pair-wise intersection of the time-interval lists are to be carried out which requires one pass through the time-interval lists maintained with each subset of the item set under consideration. But this process will also prune more sets than the usual process.

In finding local association rules, the proposed method makes a pass through the time-interval lists associated with the set-subset pair of frequent item sets and computes the ratio of the support values.

### 3.7 CONCLUSION

An algorithm for finding frequent sets that are frequent in certain time periods but may not be frequent for the whole life-span of the dataset, called locally frequent sets in the first part of this chapter is given. The technique used is similar to the A-priori algorithm.
In the level-wise generation of locally frequent sets, for each locally frequent set we keep a list of all time intervals in which it is frequent. For generating candidates for the next level, pair-wise intersections of the intervals in two lists are taken. From these locally frequent item sets interesting association rules may follow. In extracting the local association rules we need to know support of all subsets of a locally frequent item set in the interval of frequent item set. For this we need a number passes through the dataset, which increases I/O cost. So we loosely define the association rules and give an algorithm to find such rules in the last part of this chapter. Finally, we report the implementation details here and details of the results obtained are reported in tabular form.