Chapter 5

“There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies”.

--C. A. R. Hoare

Alternate Indexing Technique II: Temporal Bitmap

In Chapter 3, two different index structures are discussed, which can be used to index multi-valued attributes, namely, signature based indexes and bitmap indexes. However both of these structures do not take into account the ordering of the attribute values. Since, for temporal patterns, ordering of states is important; both of these techniques cannot be applied directly. While signature based indexing can be used for indexing temporal patterns after some preprocessing has been done to maintain the relative ordering of the state [64], [3]; bitmap indexes cannot be used as they show only presence or absence of a state. Discussed below is the new alternate indexing technique based on bitmaps, but is modified to use for sequential data.

5.1 Temporal Bitmap

The temporal bitmap proposed in this research work is similar to basic bitmap but along with showing presence or absence of states, it also shows relative ordering states. As defined in chapter 3, the temporal pattern has a sequence of states and temporal relationship among the states. In the temporal bitmap indexing, the focus is on the relative ordering of the states in the temporal pattern. When a query is fired, the ordering of the states is matched first from the index and these pattern ids are selected. These are termed as drops as in signature based indexing. After that, false drops verification is done.
where those patterns for which temporal relationship between the states does not match are rejected.

In temporal bitmap, instead of keeping one bit to show presence of absence of a state, as many bits are kept as are required to show the relative ordering of states. For example, if there are 4 possible positions, 4 bits for each possible position are kept. So if a state can appear at 1\textsuperscript{st} position in a pattern, 1\textsuperscript{st} bit is set to 1 for that state in that pattern’s bitmap.

All position bits 0 indicate absence of that particular state in that pattern. Thus, a map is formed for each state depending on the position at which that state occurs in each of the patterns in the database. Figure 5.1 shows a sample database of 10 patterns defined over a set of five possible states \( S=\{A, B, C, D, E\} \) with four possible positions of these states in the patterns. Figure 5.2 shows the temporal map of this sample database. For five states and four possible positions of these states in a pattern, \( 5 \times 4 = 20 \) bits are needed for each pattern. So, if there are 10 patterns, \( 20 \times 10 = 200 \) bits are needed. Generalizing this, if \( D \) is the number of patterns, \( N \) is the number of states possible and \( K \) is the number of positions available, \( D \times N \times K \) number of bits are needed.

![Figure 5.1 Sample Temporal Patterns](image-url)
Important factor here is the number of positions (K) considered. In the example above, K has been taken to be four. This does not, in anyway, restrict the pattern size to be four only. If the size of the pattern is more than K, only first K states are considered to build the index. While matching with the query pattern, only first K states of the query pattern are matched with the corresponding bitmaps of the target patterns. A smaller value of K will lead to more number of false drops while a larger value of K will have a high space overhead. Thus an important requirement, when using temporal bitmaps as indexes, is a thorough understanding of the underlying data which includes

a) knowledge about the approximate number of states possible
b) average size of the generated pattern,

so that an optimum value of K can be decided. Nevertheless, this requirement should not deter the application of temporal bitmaps as indexes, because this information is generally available to the users actively involved in mining activities.

5.2 Retrieval Using Temporal Bitmap

For content based queries as against point queries or range queries this temporal bitmap is very suitable. Retrieval using this bitmap requires as many passes as there are states in the query pattern or K, whichever is smaller. Matching the position of a state in the
temporal bitmap is done by creating a mask for that position. If state is any state in query pattern \( q \), and \( p \) is any pattern in the database \( D \), then

\[
\text{temporal bitmap(state, } p) \land (\text{mask}) = \text{mask}
\]

...\( (1) \)

i.e. the state is present in \( p \) and is at the position indicated by the mask.

When a sub pattern query is fired, for the first state in the query pattern, all those patterns which have matching state in the first position, their pattern ids are selected. In the subsequent iterations, those pattern ids which are selected in the previous iteration are checked to match the next states and their position against those in the query pattern using (1). This continues till either the end of pattern is reached or the available positions are exhausted \( (K) \). Thus, the list of pattern ids gets refined in each iteration and the final list will be evaluated for false drops.

For a super pattern query, i.e. to retrieve all those patterns which are sub patterns of the query pattern, the process is slightly different. All those patterns which match the first state in the query pattern (using (1)), their pattern-ids get selected for further evaluation. Let these patterns be \( p' \). From these, those patterns are retained which satisfy (2) below, to ensure all sub-patterns of the query pattern are selected.

\[
\text{temporal bitmap(state, } p') \land (\text{mask}) = \text{mask} \lor 0
\]

...\( (2) \)

i.e. either the state of \( q \) is present at the position indicated by the mask or it is absent.

The next step is the false drops verification. For each pattern id selected, the corresponding pattern is brought from the database. States after first \( K \) states, if any, and the relationships among all the states are matched with those in the query pattern. If there is a match, the pattern becomes an actual drop otherwise it is a false drop. Since major processing time involves evaluating the selected pattern-ids, the index is efficient if the number of false drops can be minimized.
Below an example is shown to help understand the retrieval using the temporal bitmap as index.

Let a query pattern on the data shown in figure 5.1 be as in figure 5.3.

\[ p^9 = \begin{bmatrix} B \\ D \end{bmatrix} \]

Let the query type be sub pattern query, i.e. retrieve all the patterns which have this pattern as the sub-pattern. From the data, it can be seen that this pattern is a sub pattern of patterns \( p_2 \) and \( p_9 \).

Explained below is the retrieval using temporal bitmap.

First state in the query pattern is B. So the temporal bitmap is checked for state B and all the patterns for which there is B at first positions are selected. (Figure 5.4).

So the patterns selected are \( p_2, p_3, p_5, p_7 \) and \( p_9 \).

Second state in query pattern is D. From the patterns selected above, all those patterns are retained which have D at 2nd position (Figure 5.5).

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>0001</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td>p2</td>
<td>0100</td>
<td>0001</td>
<td>0001</td>
<td>0100</td>
</tr>
<tr>
<td>p3</td>
<td>0000</td>
<td>0100</td>
<td>0100</td>
<td>0010</td>
</tr>
<tr>
<td>p4</td>
<td>0010</td>
<td>0010</td>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td>p5</td>
<td>0000</td>
<td>0000</td>
<td>0010</td>
<td>0000</td>
</tr>
<tr>
<td>p6</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0010</td>
</tr>
<tr>
<td>p7</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td>p8</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td>p9</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td>p10</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
</tr>
</tbody>
</table>

Figure 5.4 State B present in First Position
The patterns selected in this pass are $p_2$, $p_5$ and $p_9$. Since there are only two states in the query pattern, the process stops here and the final list of the patterns is $p_2$, $p_5$ and $p_9$, which will be evaluated for relationships’ match among the states. Out of these, the number of false drops is 1 as $p_5$ does not satisfy the relationship match criteria.

Let another query pattern on the above data be as shown in figure 5.6.

Let the query type be super pattern query. From the data it can be seen that pattern id $p_4$ and $p_{10}$ are required patterns. Explained below is the retrieval using temporal bitmap.
The first state in the pattern is $A$. Select those pattern ids from the map which have $A$ in the first position. The list of pattern ids selected is $p_1$, $p_4$, $p_6$ and $p_{10}$ (Figure 5.7).

State at the second position is $C$. From the list of pattern ids selected in the first step, those are retained which have a 0 in the second position or have $C$ at the second position (Figure 5.8). So from this list, patterns selected are $p_1$, $p_4$ and $p_{10}$. $p_6$ is rejected because the position of state $C$ does not match with position of $C$ in query pattern. $p_1$ is retained as it shows absence of state $C$ and hence could be potential sub-pattern of the query pattern. Next state in the query pattern is $B$.

<table>
<thead>
<tr>
<th>p1</th>
<th>p2</th>
<th>p3</th>
<th>p4</th>
<th>p5</th>
<th>p6</th>
<th>p7</th>
<th>p8</th>
<th>p9</th>
<th>p10</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0001</td>
<td>0000</td>
<td>0000</td>
<td>0001</td>
<td>0100</td>
<td>0001</td>
<td>0010</td>
<td>0100</td>
<td>0000</td>
</tr>
<tr>
<td>B</td>
<td>0100</td>
<td>0001</td>
<td>0001</td>
<td>0100</td>
<td>0001</td>
<td>0000</td>
<td>0001</td>
<td>0000</td>
<td>0010</td>
</tr>
<tr>
<td>C</td>
<td>0000</td>
<td>0100</td>
<td>0100</td>
<td>0000</td>
<td>0100</td>
<td>0000</td>
<td>0100</td>
<td>0000</td>
<td>0010</td>
</tr>
<tr>
<td>D</td>
<td>0010</td>
<td>0010</td>
<td>0000</td>
<td>0000</td>
<td>0010</td>
<td>0000</td>
<td>0100</td>
<td>0001</td>
<td>0010</td>
</tr>
<tr>
<td>E</td>
<td>0000</td>
<td>0000</td>
<td>0010</td>
<td>0000</td>
<td>0000</td>
<td>0010</td>
<td>0000</td>
<td>0000</td>
<td>0000</td>
</tr>
</tbody>
</table>

Figure 5.7 State A present in First Position

<table>
<thead>
<tr>
<th>p1</th>
<th>p4</th>
<th>p6</th>
<th>p10</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0001</td>
<td>0001</td>
<td>0001</td>
</tr>
<tr>
<td>B</td>
<td>0100</td>
<td>0100</td>
<td>0000</td>
</tr>
<tr>
<td>C</td>
<td>0000</td>
<td>0010</td>
<td>0100</td>
</tr>
<tr>
<td>D</td>
<td>0010</td>
<td>0000</td>
<td>0000</td>
</tr>
<tr>
<td>E</td>
<td>0000</td>
<td>0000</td>
<td>0010</td>
</tr>
</tbody>
</table>

Figure 5.8 State C present in Second Position

Following the same logic as above, from $p_1$, $p_4$ and $p_{10}$, all the patterns are retained. $p_1$ is retained as it shows $B$ at the third position. Last state in the query pattern is $D$, $p_{10}$.
\( p_1 \) is rejected as it has \( D \) at second position and both \( p_4 \) and \( p_{10} \) are retained. This becomes the final list for false drops verification. During false drops verification it is found that both patterns indeed are sub-patterns of query pattern. Hence the number of false drops in this case is zero.

The Temporal bitmap offers the following advantages, as compared to signature based indexes

1. Most of the operations can be at bit level, which makes the processing faster
2. No pre processing on the patterns is required to maintain the ordering of the states

### 5.3 Temporal Bitmap as Index

#### 5.3.1 Creating Temporal Bitmap

Creating a temporal bitmap for a database of temporal patterns is relatively easy. For each pattern in the database, the states of the pattern are read sequentially. The corresponding bit position is set to 1 in the temporal bitmap for that pattern for each of the states present in the pattern. If there are more states present in the pattern than the available positions (\( K \)) in the bitmap, only first \( K \) states of the pattern are considered and their position is indicated in the temporal bitmap. The remaining states will not be shown in the bitmap. Thus, temporal bitmap acts as an imprecise filter when used for retrieval. For each query pattern, first \( K \) states of the query pattern will be matched against \( K \) available positions in the bitmap index created for the database. If these states and their positions match, the corresponding pattern id becomes a drop. These drops will then be verified in the second step called false-drop verification. The selected pattern id (drop) becomes a false drop if the remaining states or the relationship among the states of that pattern id do not match with the query pattern.

Algorithm 5.3.1 lists the pseudo algorithm for building the temporal bitmap for a database of temporal patterns.
Algorithm 5.3.1. Constructing the Index

**Input:** Database $D$ of temporal patterns, Available positions $K$, Number of States possible $N$

**Output:** Temporal bitmap file

**Method:**
1. for each pattern $p$ with pattern-id $pid$ in database $D$ do
2.   take first $K$ states of the pattern $pid$
3.   for each state $state \in S$ at position $j$, set $j$th bit to 1 in the bitmap $(state, pid)$
4. end for
5. end for
6. Save the map in a file
7. Return the file pointer

5.3.2 Answering Sub Pattern Queries

The requirement for sub pattern queries is to retrieve all those patterns $p$ for which the given query pattern $q$ is the sub pattern. First $K$ states of $q$ are taken and first state from there is matched against first state for each patterns $p \in$ database $D$ using (1). Those patterns are selected for which this state and its position match. Next state of the $q$ is matched with these selected patterns. This process continues till the first $K$ states of $q$ and their positions are matched. The final list of pattern-ids becomes the list of drops. These drops are then verified for actual match.

Given a temporal pattern database $D$ and a query pattern $q$, FindSubPattern(), listed in algorithm 5.3.2, is used for evaluating sub patterns.

5.3.3 Answering Super Pattern Queries

The requirement for the super-pattern queries is to retrieve all those patterns $p$ which are sub patterns of the given query pattern $q$. In this case, all patterns $p \in$ database $D$, are selected for which the first state of $q$ matches. From this list, those pattern ids are
progressively removed, for which the next states of $q$ do not match. The next states of $q$ may be absent in $p$, thus making it a potential sub-pattern of $q$.

Given a temporal pattern database $D$ and a query pattern $q$, FindSuperPattern(), listed in algorithm 5.3.3, is used for evaluating super pattern queries.

### 5.3.4 Answering Equality Queries

The processing for Equality query is similar to Sub pattern query and Algorithm 5.3.4 lists FindEqualPatterns().
Algorithm 5.3.2. Pseudocode of **FindSubPatterns** ()

**Input:** Database \( D \) of temporal patterns, Query Pattern \( q \), Available positions \( K \), Temporal Bitmap for \( D \)

**Output:** Answerset, Falsedrops

**Method:**

1. select the first \( K \) states of \( q \) and form the list \( L \)
2. let the first state in the list \( L \) be \( \text{state} \).
3. create the mask for position 1.
4. check the temporal bitmap of \( \text{state} \)
   
   for each pattern with id \( \text{pid} \) in the bitmap of \( \text{state} \) do
   
   if \((\text{mask} \land \text{bitmap} (\text{state}, \text{pid}))\) is equal to \( \text{mask} \) then
   
   add \( \text{pid} \) to the list \( P \)
   
   end if

   end for

5. for each state \( u \in \) ordered list \( \{L - \text{state}\} \) do
   
   let the position of state \( u \) be \( j \).
   
   create the mask for position \( j \)
   
   for each pattern with id \( \text{pid} \in \) list \( P \) do
   
   if \((\text{mask} \land \text{bitmap} (\text{pid}, u)))\) is equal to \( \text{mask} \) then
   
   retain \( \text{pid} \) in the list \( P \)
   
   else
   
   \( P = P - \{\text{pid}\} \)
   
   end if

   end for

end for

6. evaluate list \( P \) for false drops

   for each pattern with id \( \text{pid} \in \) list \( P \) do
   
   retrieve \( p \) from \( D \)
   
   if \( p \subseteq q \) then
   
   add \( p \) into AnswerSet
   
   else
   
   increment Falsedrops
   
   end if

end for

7. return Answerset and Falsedrops
Algorithm 5.3.3: Pseudocode of FindSuperPatterns()

**Input:** Database D of temporal patterns, Query Pattern q, Available positions K, Temporal Bitmap for D

**Output:** Answer set, Falsedrops

**Method:**
1. select the first K states of q and form the list L
2. let the first state in the list L be state.
3. create the mask for position 1.
4. check the temporal bitmap of state
   - for each pattern id pno in the bitmap of state do
     - if (mask ∧ (bitmap of pattern id pno)) is equal to mask then
       - add pno to the list P
     - end if
   - end for
5. for each state u ∈ list \{L − state\} do
   - let the position of state u be j.
   - create the mask for position j
   - for each pattern id pno ∈ list P do
     - if (mask ∧ (bitmap of pattern id pno and state u)) is equal to mask OR ((bitmap of pattern id pno and state u) is equal to 0) then
       - retain pno in the list P
     - else
       - P = P − {pno}
     - end if
   - end for
   end for
6. Evaluate list P for false drops
   - for each pattern id pno ∈ list P do
     - retrieve p from D
     - if p ≥ q then
       - add p into AnswerSet
     - else
       - increment number of false drops
     - end if
   - end for
7. return Answer set and Falsedrops
**Algorithm 5.3.4.** : Pseudocode of **FindEqualPatterns()**

**Input:** Database D of temporal patterns, Query Pattern q, Available positions K, Temporal Bitmap for D

**Output:** Answerset, Falsedrops

**Method:**
1. select the first K states of q and form the list L
2. let the first state in the list L be state.
3. create the mask for position 1.
4. check the temporal bitmap of state
   for each pattern id pno in the bitmap of state do
      if (mask ∧ (bitmap of pattern id pno)) is equal to mask then
         add pno to the list P
      end if
   end for
5. for each state u ∈ list {L – state} do
   let the position of state u be j.
   create the mask for position j
   for each pattern id pno ∈ list P do
      if (mask ∧ (bitmap of pattern id pno and state u)) is equal to mask then
         retain pno in the list P
      else
         P = P – {pno}
      end if
   end for
6. Evaluate list P for false drops
   for each pattern id pno ∈ list P do
      retrieve p from D
      if p ≡ q then
         add p into AnswerSet
      else
         increment number of false drops
      end if
   end for
7. return Answerset and Falsedrops