CHAPTER 3

LOOK AHEAD MATCHING TECHNIQUE

Look ahead matching technique (LAMT) is the first subpart of bitmap preprocessing strategy. It works on the generated bitmap index of query attribute. This technique intends to overcome the fruitless operation problem occur in traditional BI method. This technique considers different parameters for analyzing bitmap vector. These are sequence matching, the sequence of attributes, aggregate functions, the cardinality of bitmap vector, size of bitmap vector, a subpart of bitmap vector, threshold value, and intermediate sequence. Parameters used to divide the vectors into subpart are bitmap vector, size of bitmap vector and the threshold value. Intermediate results are matched with further sequences by using the parameters like sequence matching, the sequence of attribute and the threshold value. Standard sequence is generated using aggregate function and threshold value of iceberg query. In this way by considering all these parameters look ahead matching technique help to reduce fruitless bitwise operation problem occur in previous research.

3.1 Look Ahead Matching Technique for Iceberg Query Evaluation

This section describes the detailed workflow of LAMT. The first step of LAMT is a generation of bitmap index. The bitmap index is created on attribute set of input Iceberg query. This is a very important task as in future it helps to reduce the overhead of database access. It selects only required attribute from huge input data set. During Iceberg query processing only these parameters are considered till the declaration of the query result. Another important feature of this module is it create bitmap vector of selected input which is in the form of 0’s and 1’s which help for fast query processing.
LAMT gets activated immediately after generation of bitmap index. In this module, based on threshold value the analysis of logical operation is done in advance. The query result is compared with threshold value, if it satisfies threshold condition then only remaining part will be evaluated otherwise it will be prune and declare as fruitless operation. Initially, it partitions the vector into subpart, and each subpart is evaluated independently as shown in Figure 3.1. The bitmap vectors to be considered for evaluation are selected based on the sequence of attributes in Iceberg query. Once bitmap vectors get finalized, then it is ready for analysis phase.

Figure 3.1 Flow diagram of Look Ahead Matching Technique
It performs bitwise AND operation of the subpart. Bitwise AND operation is used to generate intermediate result. Intermediate result passes through the threshold condition check. Based on the result of threshold check the further analysis is planned. The result of threshold check is analysed by considering aggregate function used in it. The combination of vector is included in final iceberg query result only when their results satisfy threshold condition and its aggregate function posses antimonotone property. Further bitwise AND operations are skipped. In this way, LAMT helps to reduce the empty bitwise operation problem and improve the efficiency of bitmap preprocessing strategy for iceberg query evaluation.

3.2 Algorithms and Analysis of Look Ahead Matching Technique

This subsection describes the intuition behind this research. The look ahead matching technique comprises of different modules like the generation of bitmap Index, performing bitwise AND operation between different sequences, performing bitwise XOR operation between different sequences and matching of subsequence. Implementation of LAMT is based on following algorithms.

Algorithm 3.1 Look Ahead Matching Technique (LAMT)

Algorithm 3.1 describes the detail procedure which is used for implementation of LAMT. It takes care of analysis of bitmap vector. Based on analysis and threshold value it declared the Iceberg query result in advance. In this way, it helps to reduce the fruitless bitwise operation problem of traditional BI method.

**Input:** (Iceberg query, Input Database, Threshold Condition, Aggregate Function)

**Processing:** (Generating Bitmap Index on query attributes, Performing Bitwise AND, Performing sequence matching, Performing Bitwise XOR, declaring fruitless attribute)
Output: (Set of all combinations which are the part of FINAL ICEBERG QUERY RESULT)

1. Generate the bitmap index on the Iceberg query attribute.
   Generate BITMAP_INDEX (Iceberg Query Attribute, Cardinality of Attribute, Distinct sub attribute)
2. Declare q.list, cardinality.value and sub attribute[ ]
3. for each Iceberg Query Attribute
4.    find (distinct sub attribute list)
5. for( i=0;i<=cardinality.value;i++)
6.    q.list ← first Sub attribute[i]
7.    inc(q. list);
8. Divide each attribute into equal number of subsections.
   L= length of attribute vector, Sv = Vector size
   Input vector P = (L/Vs) parts.
   P={ P1,P2,P3,…….,Pn}
9. For each Part
   Count number of 1’s and their positions.
10. Start with first subpart
11. for (i=0;i<total_parts; i++)
12. for(j= 0;j<total_attribute&& j< total_combinations;j++)
13. Perform Bitwise AND operation (Attribute1 AND Attribute1.Combination1)
14. Continue this till half of the subparts are evaluated.
15. if (half_Result_AND satisfy Threshold Condition) then
16.    perform matching subsequences based on attribute value.
17. else check for the number of 1’s bit in remaining subpart
18. if it satisfies threshold condition then
19.    perform Bitwise AND
20. (CollectResult_sequence_Matching) &&(CombineResult_All_SubParts)
21. if (Final Result satisfy Threshold Condition) then
22.    add this combination in FINAL ICEBERG QUERY RESULT
23. else discard the combination and declare it as fruitless operation and stop further processing.
24. for all the combinations which are the part of FINAL ICEBERG QUERY RESULT
25. Perform XOR operation
26. Generate New Vector from the current one.
   \[ R1 = (X1 \text{ AND } X2) \]
   \[ \text{New } X1 = (X1 - R1) \]
   \[ \text{New } X2 = (X2 - R1) \]
27. for each newly generated vector
28. if( Number of 1’s greater than Threshold) then
29. consider it for further processing
30. else discard this vector from any future processing of this query
31. return(FINAL ICEBERG RESULT)

**Algorithm 3.2  Performing XOR operation between different sequences.**

In LAMT bitwise AND and XOR operations are used. Algorithm 3.2 describes the detail of bitwise XOR operation. The detailed procedure for bitwise AND is mentioned in Algorithm 3.1 from step 13 to 20. Bitwise XOR operation helps to identify fruitless operations and fruitless vectors in advance. The exclusive or (XOR) operation is commonly used to reduce the bias from the generated bits.

Exclusive OR (XOR) operation is denoted by the symbol \( \otimes \). Bitwise XOR operation result is \( X \otimes Y = 1 \) if only one of \( X \) and \( Y \) is equal to 1 otherwise \( X \otimes Y = 0 \). The properties of XOR operation are:

1. Commutative: \( X \otimes Y = Y \otimes X \)
2. Associative: \( X \otimes (Y \otimes Z) = (X \otimes Y) \otimes Z \)
Bitwise exclusive OR operation helps to detect the possibility of expected output, variance and covariance. Bitmap preprocessing strategy takes benefit of these features of XOR operation.

**Input**: Sequence X and Sequence Y

**Output**: RESULT SEQUENCE = X ⊕ Y

1. Initialize RESULT SEQUENCE = 0;
2. Assuming 32-bit Integer
3. for (int j=31;j>=0;j--)
4. { 
5. find current bits in X and Y
6. bool r1 = X &(1<< j);
7. bool r2 = Y &(1<< j);
8. }
9. if (both X and Y bits are 1) then
10. bit of RESULT SEQUENCE = 0
11. else XOR is same as OR
12. bool XORBIT = (r1 & r2) ? 0 : (r1| r2);
13. Update result
14. RESULT SEQUENCE <<= 1;
15. RESULT SEQUENCE = XORBIT;
16. return RESULT SEQUENCE

Algorithm 3.1 consist of the procedure of bitwise AND operation and algorithm 3.2 perform bitwise XOR operation. Both these algorithms work on the sequences provided by bitmap index. Given sequence ε, is a set of elements n. Each element e of ε is associated with the set B of m properties or attributes namely B1, B2, B3, ..., Bm. The value of attribute Bi of an element n is represented by e. As per the example mentioned in section 2.5, a set of Month corresponds to the element domain set ε. Attribute Category and attribute Amount are two examples of
the properties of Month. Consider that one of the attribute \( B \) can uniquely identify an element \( e \). This attribute is called an identifying attribute. In this example, Month is identifying attribute. Attribute Category and attribute Amount are a non identifying attribute.

Suppose, the domain of attribute \( B_i \) denoted by \( D_i \), which is the set of all possible values in attribute \( B_i \) where \( i = 1, 2, 3, \ldots, m \). In this example, all Months like "June", "July" and "August" form the domain of attribute "Month". The value of domain set represented by \( V \), to be the union of the domains of all attributes. The concept of sequence analysis introduced by Peng et al. (2012) is useful during bitwise operation processing. The bitmap index is a set of sequences. Each sequence is associated with an ordered list of elements.

Therefore, \( V = \bigcup_{i=1}^{n} D_i \)  

(3.1)

The bitmap index is generated using Equation (3.1). Here, \( V \) is the set of all possible values. A value \( v \) is said to be an identifying value if \( v \) is a subset of an identifying attribute. This \( v \) can be used to identify an element \( e \) uniquely. For example, both month and category are identifying values. Logically the selection of amount as identifying attribute is not effective in this example.

Therefore, define \( U \) to be a set of all identifying values. Note that \( U \subseteq V \).

If value \( v \) is an identifying value we define the attribute value set of \( v \) denoted by \( \alpha(v) \), to be the set of all possible attribute values of the element identified by \( v \).

For example, if \( v \) is “June” an identifying value selected from \( V \) then \( \alpha(v)=\{“June”, “Fruit”, “Vegetable”, “Milk”, 100, 200, 300, 400, 500, 600, 700, 800\} \). A sequence is an order list of values where each value is drawn from \( V \). Suppose that there are \( k \) values in the sequence, a sequence is represented in the form of “\( v_1, v_2, \ldots, v_k \)” where \( v_i \in V \) for \( i=1, 2, 3, \ldots, k \). In this representation for any two values \( v_i \) and \( v_j \) where \( i < j \), \( v_i \) appears before \( v_j \).
Consider a query q in the form of “v1,v2,...vk” where vi ∈ V for i ∈ [1,k]. An identifying sequence s ∈ S in the form of “u1,u2,...,ul” where ui ∈ u for i ∈ [1,l]. Query q is said to match s or s is matched by q if there exist k integers namely j1,j2,...jk such that for each i ∈ [1,k], vi matches Uji and 1 ≤ j1 < j2 < ... < jk ≤ l.

LAMT perform aggregation independently by decomposing it in several parts. It contains some sub multi sets to be aggregated. The concept of aggregate function processing suggested by Paulo et al. (2011) is applicable to this research. They proposed that aggregate functions can be performed in a single computation involving all input data set or it may avoid such centralised computation. In case of distributed aggregation, it is necessary to apply multi set aggregation concept. An Aggregation function $Aggf$ takes input as elements from domain $D$ and produces output $O$.

$Aggf : N^D \rightarrow O$ is said to be self-decomposable if it can merge the result of an empty multiset. The formulation of aggregate function in bitmap preprocessing strategy is done using Equation (3.2).

$$Aggf(P \cup Q) = ([Aggf(P)] \circ [Aggf(Q)] \circ ... \circ [Aggf(N)]) \tag{3.2}$$

In Equation (3.2), $\cup$ represents the multi set summation of aggregate function result. The symbol $\circ$ indicates merge operation to be performed between multi set aggregate results. The aggregation result is same for all divided multi sets of the multi set vector. It means that merge operation is commutative and associative. The aggregate functions such as COUNT, SUM, MAX and MIN are self-decomposable.

Computational formulation for single set SUM Function is as per Equation (3.3). Here, $p$ is a set of element and its SUM is represented using $P$.
of all subset is the summation of multi set aggregation result which is calculated using Equation (3.4).

\[ \text{SUM}([p]) = P \]  \hspace{1cm} (3.3)
\[ \text{SUM}(P \cup Q) = \text{SUM}(P) + \text{SUM}(Q) \] \hspace{1cm} (3.4)

Computational formulation for COUNT aggregate function is as per Equation (3.5). COUNT aggregation possesses antimonotone property therefore if anyone subset of COUNT aggregation satisfy threshold condition then whole set comes under that category. Minimum one subset of multi set must satisfy COUNT condition. Multi set COUNT aggregation is computed using Equation (3.6).

\[ \text{COUNT}([p])=1 \] \hspace{1cm} (3.5)
\[ \text{COUNT}(P \cup Q) = \text{COUNT}(P) + \text{COUNT}(Q) \] \hspace{1cm} (3.6)

Computational formulation of MIN function is as per Equation (3.7). It is the MIN result out of the aggregate function result of the multi set. The MIN result of all subset is merged, and from that, the MIN value result is declared.

\[ \text{MIN}(P \cup Q) = \text{MIN}(P) \cap \text{MIN}(Q) \] \hspace{1cm} (3.7)

The operator \( \cap \) indicates the upending into the merge result. The same formulation applies to MAX aggregate function only the individual multi set operation performed is MAX.

An aggregation function \( Aggf:N^D \rightarrow O \) is said to be decomposable for some self-decomposable function \( j \) and normal function \( i \), it is represented as composite function \( f = (i \circ j) \). It means that all self-decomposable functions are a subset of decomposable functions where \( i=ID \), which is the identity function. At the same time for self-decomposable functions, the intermediate results are computed and validate with output \( O \). In case of decomposable functions the intermediate results are kept as a subset to verify. AVERAGE function is
decomposable but not self decomposable. The computation of AVERAGE function done in this research using the following Equations (3.8) to (3.11).

\[
AVERAGE(P) = i(j(P))
\]  
\[j([p]) = (p, 1)
\]  
\[j(P \cup Q) = j(P) + j(Q)
\]  
\[i((\text{sum, count})) = \frac{\text{sum}}{\text{count}}
\]

Here, \(j\) is the self-decomposable function that outputs value of domain pair and (+) is the standard SUM operation. The function \(i\) is the normal function used to find the AVERAGE of parameters results. In this way, the computation of all aggregate functions is performed in bitmap preprocessing strategy. All these computations are required for matching of sub sequences with output domain and to generate intermediate result.

**Algorithm 3.3 Matching of sub sequences based on attribute value.**

The algorithm 3.3 perform matching of subsequence based on attribute value. It consists of MatchCheck(q,s) function which is used to identify the sequence. The intermediate result is checked with the input sequence. This algorithm typically deals with individual identifying sequence and query sequence.

**Input** : Iceberg Query \(q\) and a set \(S\) of identifying sequences

**Output** : Set of identifying sequences in \(S\) which are matched by \(q\).

1. \(O = \emptyset\);
2. for each \(s \in S\) do
3. \(\text{Check}_\text{for}_\text{Match} = \text{MatchCheck}(q,s)\)
4. Let \(s\) be a sequence in form of “\(u_1,u_2,\ldots,u_l\)”
5. Let \(q\) be a query in the form of “\(v_1,v_2,\ldots,v_k\)”
6. \(j = 1\);
7. for \(i = 0\) to \(k\) do
8. find the smallest integer $r \in [j, l]$ such that $vi \in \alpha(ur)$
9. if there exists such a value $r$ then
10. $j= r+1$;
11. else return false
12. if Check_for_Match = true then
13. $O = O \cup \{s\}$
14. return $O$

In Algorithm 3.3, MatchCheck(q,s) method return a boolean value indicating whether a query q matches identifying sequences. The efficiency of this algorithm depends on how to implement method MatchCheck.

The time complexity of Algorithm 3.3 is $O(lm)$ where $l$ is the maximum length of a sequence and $m$ is the total number of attributes. However, the time complexity of Algorithm 3.1 is $O(nlm)$ where $n$ is the total number of sequences in S. In this way the look ahead matching strategy represented in Algorithm 3.1 work along with all subroutines like dividing the sequence, matching the sequence, performing bitwise AND operation, perform bitwise XOR operation, prune the vector at runtime and declare iceberg query result.

3.3 Demonstration of Look Ahead Matching Technique (LAMT)

This section demonstrates the example considered in section 2.5. This example is demonstrated using the methodology of look ahead matching technique (LAMT). Finally the results are compared with traditional_BI method.

3.3.1 Query with COUNT Aggregate Function

**Query 1:** Select Month, Category, Count (*) from Monthly_Exp_25 group by Month, Category having Count (*)>=4;
LAMT works in following way to evaluate query 1. In case of COUNT aggregate function, it performed bitwise AND operation between all the subset of attributes. As COUNT possess antimonotone property so LAMT work accordingly. It considers this property during bitwise AND as well as during sequence matching. As per query1 ,attribute subset are the combination of Month and Category occurs in database. Query result is the combinations of same Month and Category whose count is greater than or equal to 4. As shown in Table 1.2 of bitmap index, first subset is June from Month and Milk from category. So, LAMT perform bitwise AND operations in following sequence.

1. Divide the attribute subset into number of parts. Here as per sequence the attributes are JUNE and MILK. The length of subset is 25 tuples. Here the Threshold value is 4, so we divide subset in 4 parts each subset will contain approximately 6 tuples. Here numbers of tuples are less therefore we divide it into two subparts only. Partitioning is applicable for dataset with more number of tuples.

2. Assume the subset is divided into two parts. Therefore, Part I contain 13 parameters and part II contain 12 parameters.

3. The detail of first combination (June AND Milk) is represented in Table 3.1. It consist of number of 1’s and their location.

<table>
<thead>
<tr>
<th>Sub _ Part</th>
<th>I</th>
<th>II</th>
</tr>
</thead>
<tbody>
<tr>
<td>Month &amp; Category</td>
<td>5:(1,2,5,12,13)</td>
<td>3: (14,17,18)</td>
</tr>
<tr>
<td>June</td>
<td>7:(1,3,5,10,11,12,13)</td>
<td>5:(18,19,21,23,24)</td>
</tr>
<tr>
<td>Milk</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Now computation of bitwise AND operation between part I of June and Milk is performed.
Table 3.2 R1 result of (June AND Milk) Part I

<table>
<thead>
<tr>
<th></th>
<th>June</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Milk</td>
<td></td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>R1</td>
<td></td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

As shown in Table 3.2, (June AND Milk) combination satisfy the threshold condition specified in Query 1. Therefore, the combination (June AND Milk) is the subset of final Iceberg query 1 result. Here it is observed that within half computation the result is declare and there is no need to perform further bitwise AND operation. In this way LAMT in advance declare the result of query and help to avoid the fruitless operations.

4. As per the algorithm after declaring the query result, next phase is checking the possibility of individual attribute to be a part of final Iceberg query result in future. Threshold constraint is used to check the possibility of vector for further operation otherwise such vectors are pruned at run time. Bitwise XOR operation is performed for vector pruning.

R1 = (June AND Milk)
New June = June OR R1

Table 3.3 New June sequence

|     | June | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| R1  |      | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| New June |      | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Table 3.3 indicates that New June vector does not satisfy threshold condition. So June vector is discarded from bitmap vector. It means there is no need to perform the bitwise AND operation for the combination (June AND Fruit) and (June AND Vegetable). Till this step only one bitwise AND and bitwise OR operation has performed. On the basis of only one bitwise XOR result further two fruitless bitwise AND operations are avoided. However, in case of traditional_BI approach
it performed all AND operation and then declare that (June AND Fruit) and (June AND Vegetable) are not the part of final iceberg Query 1 result.

5. Similarly, check the possibility of Milk attribute to be the part of final query 1 result or not. This is done using bitwise XOR operation.

R1 = (June AND Milk)
New Milk = Milk OR R1

<table>
<thead>
<tr>
<th>Table 3.4 New Milk sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Milk</td>
</tr>
<tr>
<td>-----</td>
</tr>
<tr>
<td>1 0 1 0 1 0 0 0 0 1 1 1 0 0 0 0 1 1 1 0 1 0 1 1 0</td>
</tr>
<tr>
<td>1 0 0 0 1 0 0 0 0 0 1 1 0 0 0 0 1 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>0 0 1 0 0 0 0 0 0 1 1 0 0 0 0 0 0 1 0 1 0 1 1 0</td>
</tr>
</tbody>
</table>

Table 3.4 declared that, New Milk sequence satisfy the threshold condition so there is possibility that Milk along with another attribute may be the part of final Query 1 result. Hence, Milk attribute is not discarded from bitmap vector.

6. As per bitmap sequence next combination is (July AND Milk). The number of 1’s and their locations for the combination (July AND Milk) is represented in Table 3.5.

<table>
<thead>
<tr>
<th>Table 3.5 (July AND Milk) subpart detail</th>
</tr>
</thead>
<tbody>
<tr>
<td>Month &amp; Category</td>
</tr>
<tr>
<td>------------------</td>
</tr>
<tr>
<td>July</td>
</tr>
<tr>
<td>Milk</td>
</tr>
</tbody>
</table>

Perform bitwise AND between part I and Part II of July and Milk combination.

<table>
<thead>
<tr>
<th>Table 3.6 R4 result of (July AND Milk) Part I and Part II</th>
</tr>
</thead>
<tbody>
<tr>
<td>Part I</td>
</tr>
<tr>
<td>--------</td>
</tr>
<tr>
<td>July</td>
</tr>
<tr>
<td>Milk</td>
</tr>
<tr>
<td>Result(R4)</td>
</tr>
</tbody>
</table>

As per the values from Table 3.6 it has observed that in first part the R4 contain only one time 1’s bit. But in part II total number of 1’s in July and Milk are respectively 4
and 5 which satisfy threshold condition due to this first perform bitwise AND on Part II. Bitwise AND result of Part II contain 2 time 1’s which does not satisfy threshold condition. Therefore it is necessary to perform bitwise AND between remaining subparts. Finally combine the result of part I and II. Here R4 does not satisfy threshold condition therefore it is not in the part of final query 1 result.

7. Next sequence is (July AND Fruit). The contents of combination (July AND Fruit) in terms of number of 1’s and their locations are represented in Table 3.7.

Table 3.7 (July AND Fruit) subpart detail

<table>
<thead>
<tr>
<th>Sub _ Part</th>
<th>Month &amp; Category</th>
<th>I</th>
<th>II</th>
</tr>
</thead>
<tbody>
<tr>
<td>July</td>
<td>5:(3,6,7,8,9)</td>
<td></td>
<td>4: (19,20,21,22)</td>
</tr>
<tr>
<td>Fruit</td>
<td>5:(2,6,7,8,9)</td>
<td></td>
<td>3:(14,17,20)</td>
</tr>
</tbody>
</table>

Compute bitwise AND between Part I and Part II of July and Fruit combination.

Table 3.8 R5 result of (July AND Fruit) Part I

<table>
<thead>
<tr>
<th></th>
<th>July</th>
<th>Fruit</th>
<th>Result(R5)</th>
</tr>
</thead>
<tbody>
<tr>
<td>July</td>
<td>0 0 1 0 1 1 1 0 0 0 0 1 1 1 0 0 0 0 0 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fruit</td>
<td>0 1 0 0 0 1 1 1 0 0 0 0 1 1 1 0 0 0 0 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Result</td>
<td>0 0 0 0 0 1 1 1 0 0 0 0 1 1 1 0 0 0 0 0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

As shown in Table 3.8, part I result of R5 satisfy the threshold condition therefore there is no need to perform the remaining half bitwise AND operation. The combination (July AND Fruit) is included in the final iceberg query1 result.

8. Perform the computation to generate new July and Fruit attribute. As per the algorithm after declaring the result, computation for checking the possibility of individual attribute to be a part of final Iceberg query result has to be performed. The check for vector pruning is performed by comparing threshold value check. The XOR operation is work as below:

\[ R5 = \text{(July AND Fruit)} \]

\[ \text{New July} = \text{July OR R5} \]
Table 3.9 New July sequence

| July   | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 |
| Result(R5) | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| New July | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 |

As shown in Table 3.9, new sequences of July satisfy threshold condition so it is not discarded from the vector.

9. Similarly perform bitwise XOR to generate new Fruit attribute.

R5 = (July AND Fruit)

New Fruit = Fruit OR R5

Table 3.10 New Fruit sequence

| Fruit   | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| Result(R5) | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| New Fruit | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |

As per Table 3.10, new fruit sequence does not satisfy threshold condition so it is discarded from vector list. Fruit attribute discarding help here to declare that (August AND Fruit) operation is fruitless and we skip this operation. In this way it identifies the fruitless operation in advance. This significant contribution of LAMT helps to improve performance of bitmap preprocessing strategy.

10. Perform bitwise AND between (July AND Vegetable) combination. Table 3.11 indicate the number of 1’s and their locations for the combination (July AND Vegetable).

Table 3.11 (July AND Vegetable) subpart detail

<table>
<thead>
<tr>
<th>Month &amp; Category</th>
<th>Sub _ Part</th>
<th>I</th>
<th>II</th>
</tr>
</thead>
<tbody>
<tr>
<td>July</td>
<td>5:(3,6,7,8,9)</td>
<td>4: (19,20,21,22)</td>
<td></td>
</tr>
<tr>
<td>Vegetable</td>
<td>1:(4)</td>
<td>4:(15,16,22,25)</td>
<td></td>
</tr>
</tbody>
</table>
Perform bitwise AND between part I and Part II of July and Vegetable combination. Its result is as shown in Table 3.12.

Table 3.12 R6 result of (July AND Vegetable) Part I and Part II

<table>
<thead>
<tr>
<th></th>
<th>Part I</th>
<th></th>
<th>Part II</th>
</tr>
</thead>
<tbody>
<tr>
<td>July</td>
<td>0 0 1 0 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vegetable</td>
<td>0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 1 0 0 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Result(R6)</td>
<td>0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

R6 does not satisfy threshold condition so it is not included in final query 1 result. In this case it is observed that LAMT is unable to find out fruitless AND in advance. This situation occurs because till this operation new vegetable is not generated and tested.

11. Perform (August AND Milk). The number of 1’s and their location is as represented in Table 3.13.

Table 3.13 (August AND Milk) subpart detail

<table>
<thead>
<tr>
<th></th>
<th>Sub _ Part</th>
<th>I</th>
<th>II</th>
</tr>
</thead>
<tbody>
<tr>
<td>Month &amp; Category</td>
<td>August</td>
<td>3:(4,10,11)</td>
<td>5: (15,16,23,24,25)</td>
</tr>
<tr>
<td></td>
<td>Milk</td>
<td>7:(1,3,5,10,11,12,13)</td>
<td>5:(18,19,21,23,24)</td>
</tr>
</tbody>
</table>

Perform bitwise AND between part I and Part II of August AND Milk combination.

Table 3.14 R7 result of (August AND Milk) Part I and Part II

<table>
<thead>
<tr>
<th></th>
<th>Part I</th>
<th></th>
<th>Part II</th>
</tr>
</thead>
<tbody>
<tr>
<td>August</td>
<td>0 0 0 1 0 0 0 0 0 1 1 0 0 0 1 1 0 0 0 0 0 0 1 1 1 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Milk</td>
<td>1 0 1 0 1 0 0 0 0 1 1 1 1 0 0 0 0 1 1 0 1 0 1 1 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Result(R7)</td>
<td>0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 1 1 0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
As shown in Table 3.14, R7 satisfy threshold condition so (August AND Milk) combination is included in final iceberg query 1 result.

12. As (August AND Fruit) combination is already discarded so no need to perform it.

13. Similarly perform bitwise AND between (August AND Vegetable). The details of its subpart is as shown in Table 3.15.

<table>
<thead>
<tr>
<th>Month &amp; Category</th>
<th>Sub Part</th>
<th>I</th>
<th>II</th>
</tr>
</thead>
<tbody>
<tr>
<td>August</td>
<td></td>
<td>3:(4,10,11)</td>
<td>5:(15,16,23,24,25)</td>
</tr>
<tr>
<td>Vegetable</td>
<td></td>
<td>1:(4)</td>
<td>4:(15,16,22,25)</td>
</tr>
</tbody>
</table>

Table 3.16 shows the result of bitwise AND of (August AND Vegetable). Result R9 satisfies the threshold condition therefore it is included in final Iceberg query 1 result.

<table>
<thead>
<tr>
<th>Part I</th>
<th>Part II</th>
</tr>
</thead>
<tbody>
<tr>
<td>August</td>
<td>0 0 0 1 0 0 0 0 0 1 1 0 0 1 1 0 0 0 0 1 1 1</td>
</tr>
<tr>
<td>Vegetable</td>
<td>0 0 0 1 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 1 0 0 1</td>
</tr>
<tr>
<td>Result(R9)</td>
<td>0 0 0 1 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 1</td>
</tr>
</tbody>
</table>

In this example LAMT perform bitwise AND operations 6 times. Out of 6, two times it compute half bitwise AND operation. Means it perform only five bitwise AND operations. In this way LAMT discard the fruitless vectors from the bitmap index and avoid empty bitwise operations. In this way by performing all above steps query 1 is evaluated and its result is declared. In each step of evaluation it is observed that LAMT avoid fruitless operations.
3.3.2 Query with SUM Aggregate Function

**Query 2:** Select Month, Category, SUM (Amount) from Monthly_Exp_25 group by Month, Category having SUM(Amount) > 1000;

The aggregate function used in query 2 is SUM. Evaluation of query 2 requires three attributes like Month, Category and Amount. In evaluation first up all bitwise AND between all subsets has performed. Then computation of bitwise AND between the result of ((Month AND category) AND (Amount)) has evaluated. Here aggregate function is SUM, which posses antimonotone property. The amount of similar combination has to be considered for summation. After each summation the result is checked with threshold condition specified in query 2. In between if it is observed that SUM satisfies threshold condition then further operations are skipped and result is declared.

SUM function computation required all the steps of COUNT operation. Additional bitwise AND operation has performed with the Amount subset (200,800,500,600,700,100,400,300). Following steps are required to evaluate Query 2 using LAMT.

**A.R1: (June AND Milk)**

As mentioned above two times AND operation has performed. To generate R1 same steps are required as described in 2.6.1 for query 1.R1 is the result of (June AND Milk) combination. The SUM of (June AND Milk) combination is calculated using bitwise AND between R1: (June AND Milk) and all the amount subset. The bitwise AND operation between (R1 AND 200) is as indicated in Table 3.17.

**Table 3.17 Result S_R1_200 (R1 AND 200)**

<table>
<thead>
<tr>
<th>Part</th>
<th>I</th>
<th>II</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>01000000</td>
<td>01100000</td>
</tr>
<tr>
<td>200</td>
<td>11000000</td>
<td>10000000</td>
</tr>
<tr>
<td>S_R1_200</td>
<td>10000000</td>
<td>10000000</td>
</tr>
</tbody>
</table>
As shown in table 3.17, S_R1_200 contain two times 1. Therefore, S_R1_200=200+200=400. (June AND Milk) Sum (Amount) =400. Similarly bitwise operations are performed between remaining combinations. Hence, (June AND Milk) Sum(Amount)=400+000+500+000+000+100+000+300=1300

The SUM (Amount) for (June AND Milk) is 1300 which satisfy threshold condition given in query 2. Therefore (June AND Milk) combination is the part of final iceberg query 2 result.

B. R2: (June AND Fruit)
The Sum of R2 is computed by applying same method of R1 computation. In this case it is noticed that up to (R2 AND 400) operation the Sum(Amount) of (June AND Fruit) satisfy threshold condition. Therefore (June AND Fruit) combination is the part of final iceberg query 2 result. So there is no need to perform the further AND operation. In this way this strategy helps to avoid futile AND operation this feature help to improve the performance of iceberg query.

C. R3: (June AND Vegetable)

<table>
<thead>
<tr>
<th>Result(R3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
</tr>
</tbody>
</table>

The combination (June AND Vegetable) contain all zero means this combination does not occurs any time in any transaction. Therefore there is no need to evaluate query for this combination.

D. R4: (July AND Milk)
In this case by performing only three bitwise AND operations that are (R4 AND 200), (R4 AND 800) and (R4 AND 500) the query result satisfy threshold condition. Therefore (July AND Milk) combination is the part of final iceberg query 2 result. So there is no need to perform the further AND operation. In this case LAMT reduces five bitwise AND operations.
E.R5: (July AND Fruit)
In this case also the final result is generated within three bitwise AND operations. Up to (R5 AND 500) operation the Sum (Amount) of (July AND Fruit) satisfy threshold condition. Therefore (July AND Fruit) combination is the part of final iceberg query 2 result. So there is no need to perform the further AND operation. In this case also LAMT reduces five bitwise AND operations .In this way this strategy helps to avoid futile AND operation. This help in bitmap preprocessing strategy to improve the performance of iceberg query.

F.R6: (July AND Vegetable)
Here it is noticed that there is no need to repeat the bitwise operation between R6 and remaining subset of amount because number of 1’s present in R6 is only 1. In 2nd step 1’s value is found which is 800.As per LAMT it analyse the input bitmap vector in advance. During this phase it found that only one time 1’s is present and amount attribute contain maximum 800. So it declares this combination as fruitless. Therefore in case of LAMT without evaluating this operation, it is declared as fruitless.

G.R7: (August AND Milk)
The (August AND Milk) Sum(Amount) is 1000 which does not satisfy threshold condition given in query 2.Therefore (August AND Milk) combination is not the part of final iceberg query 2 result.

H.R8: (August AND Fruit)
Here it is noticed that in given dataset no combination of (August AND Milk) exist. Therefore, it is not possible to perform any operation on such a combination which doesn’t contain any data.

I.R9: (August AND Vegetable)
In this case also up to (R9 AND 700) operation query result satisfies threshold condition. Therefore there is no need to perform remaining operations. Hence,
(August AND Vegetable) combination is included in iceberg query 2 result. Here it is noticed that unnecessary bitwise AND operations are skipped.

3.3.3 Query with AVERAGE Aggregate Function

**Query 3:** Select Month, Category, AVG (Amount) from Monthly_Exp_25 group by Month, Category having AVG(Amount)>600;

The AVERAGE aggregate function is not antimonotone therefore the detail process of query2 evaluation as described in 3.3.2 is followed. In this case no need to perform threshold condition check. After summation of all combinations AVERAGE is calculated and threshold condition check is performed.

3.3.4 Query with MIN Aggregate Function

**Query 4:** Select Month, Category, MIN (Amount) from Monthly_Exp_25 group by Month, Category having MIN (Amount)>200;

MIN aggregate function posses antimonotone property. Therefore, if any subset satisfies threshold condition, then that combination gets included in final query 4 result. In this query, each Month and Category combination has to be evaluated with amount subset. MIN(Amount) in query 4 is a threshold value. As per query 2 evaluation, the computation of query 4 has performed.

3.3.5 Query with MAX Aggregate Function

**Query 5:** Select Month, Category, MAX (Amount) from Monthly_Exp_25 group by Month, Category having MAX (Amount)>500;

In this case, Category and Month combination has to go for each combination with amount subset. This query 5 provides MAX (Amount) which is a threshold value. Query 2 evaluation is applicable in this case only instead of SUM here the MAX condition to be checked. As MAX possess antimonotone property, so if any sub part satisfies threshold condition then the query result get declared.
Table 3.18 shows the summary of bitwise AND required by LAMT against Traditional_BI Method.

<table>
<thead>
<tr>
<th>Query and Function</th>
<th>Bitwise AND operation required</th>
<th>Bitwise AND operation required</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Traditional_BI Method</td>
<td>Look Ahead Matching Technique (LAMT)</td>
</tr>
<tr>
<td>1.COUNT</td>
<td>9</td>
<td>5</td>
</tr>
<tr>
<td>2.SUM</td>
<td>81</td>
<td>42</td>
</tr>
<tr>
<td>3.AVERAGE</td>
<td>81</td>
<td>65</td>
</tr>
<tr>
<td>4.MIN</td>
<td>81</td>
<td>27</td>
</tr>
<tr>
<td>5.MAX</td>
<td>81</td>
<td>40</td>
</tr>
<tr>
<td>Total--------------</td>
<td>333</td>
<td>179</td>
</tr>
</tbody>
</table>

It is observed that look ahead matching technique required 53% less bitwise AND operations compared to Traditional_BI method. This reduction in number of bitwise operations also reduces the overhead of bitmap access. In this way, LAMT overcome the limitations of Traditional_BI method and contributed for improving performance of bitmap preprocessing strategy. This is reflected in the implementation results of bitmap preprocessing strategy for iceberg query evaluation.

### 3.4 Experimental Results

In this section, the experimentation results of the proposed look ahead matching technique is presented. Here this technique is shown independently but in bitmap preprocessing strategy it work in coordination with tracking pointer technique which is described in chapter 4.

In this section, the comparison between the traditional_BI based method and proposed look ahead matching technique is represented. The detail process of LAMT is described in section 3.3.1 to 3.3.5. The impact of LAMT is observed in its practical implementations also. The performance of the iceberg query is measured
regarding time and number of iterations required to evaluate the query. In this section, the data size of 5000 tuples is considered for the comparison. In chapter 6 the detail comparison of all the contributions of this research with traditional_BI method is described.

Figure 3.2, 3.3 and 3.4 illustrate the comparative analysis of LAMT with Traditional_Bitmap Index method. In all these figures the threshold value of query get varies and according to that the time required to process the query is measured. In Iceberg query threshold value is the condition specified in query. Threshold value is depends upon the data set, aggregate function and constraint check of iceberg query. In demonstration discussed in section 3.3.1 to 3.3.5 it has observed that the number of bitwise operations as well as fruitless operations get reduced in case of LAMT. As operations are reduced, so number of iterations in implementation gets reduced and it resulted in reduction of time to evaluate iceberg query.

Figure 3.2 Comparisons for COUNT aggregate function

Figure 3.2 depicts the time analysis for COUNT aggregate function. It is observed that time required in case of LAMT goes on decrease and it remain almost constant even though threshold value increases.
Figure 3.3 Comparisons for SUM aggregate function

Figure 3.3 indicates the performance of LAMT for SUM aggregate function. It has observed that time required get reduced even though threshold value increases. Compare to COUNT aggregate function threshold value is having closed impact on SUM functions.

Figure 3.4 Comparisons for AVERAGE aggregate function

The extension to SUM function by considering whole sequence for summation the AVERAGE results is generated. Therefore compare to SUM
function, the time required to evaluate AVG function does not decrease immensely. This is represented in Figure 3.4.

Similarly, the results for other aggregate functions are also noted. The performance analysis by considering parameters like data set size, aggregate functions, Threshold value, time and iterations are described detail in chapter 6.

3.5 Summary

This chapter discusses look ahead matching technique (LAMT) used in bitmap preprocessing strategy for iceberg query evaluation. The objective of LAMT is to minimize the fruitless operations and pruning of the vector. The detailed workflow of LAMT, its algorithm and analysis is described in this chapter. The demonstration is given by considering the example specified in 2.5. During each and every step, it has clearly shown how the LAMT avoid fruitless operations. The results are compared with the traditional method of iceberg query evaluation. Finally, the comparision with implemented module is also described.