Binning Strategy for Hierarchical Bitmap Indices with Large Scale Domain Hierarchy

Jong Wook Kim

Assistant Professor, Department of Media Software, Sangmyung University, 20-Gil, Hongji-dong, Jongno-gu, Seoul, Korea.

Abstract

As bitmap indices are useful for OLAP queries over low-cardinality data columns, they are frequently used in data warehouses. In many data warehouse applications, the domain of a column tends to be hierarchical, such as categorical data and geographical data. When the domain of a column is hierarchical in nature, the performance of query processing can be improved significantly by leveraging hierarchically organized bitmap indices. However, creating bitmaps for all possible distinct nodes of the large scale hierarchy is not feasible, due to the large amount of space overhead. Thus, in this paper, we introduce the binning strategy, HBIN, which aims to reduce the storage space overheads of hierarchical bitmap indices. The experiment results show that the proposed technique provides significant performance gains over other alternative solutions to this problem.

Keywords: Hierarchical Bitmap Index, Binning, Query Processing

INTRODUCTION

Bitmap indices have been used in data warehouses for their benefit of processing OLAP queries. Bitmap indices are useful for range, set or aggregation queries over low-cardinality data columns [1, 2, 3, 4]. Especially, COUNT queries can benefit significantly from bitmap indices, since they can be answered by scanning indices without accessing records stored in a database table. For this reason, most commercial data warehouse products, such as Teradata [5], are currently leveraging this technology in order to improve the performance of OLAP queries. Although bitmap indices are very efficient for queries over low-cardinality data columns, the same cannot be said for those over high-cardinality data columns. Indeed, creating bitmaps for all possible distinct values of high-cardinality data column is not feasible, due to the large amount of space overhead. Thus, there have been extensive studies on reducing space overhead of bitmap indices for high-cardinality data columns, such as binning strategies [6, 7, 8, 9]. The basic idea of binning strategies is to build a bitmap for each bin, which consists of multiple distinct values, rather than each distinct value. While these strategies are able to significantly reduce space overhead for high cardinality columns, they introduce false positive if query ranges do not exactly fit into binning boundaries. In this case, records stored in a database table must be fetched from the secondary disk into the main memory to remove those that reside outside the query range, which is costly. Since fetching records stored in a database table from the disk to the main memory to remove false positives, which is called as a candidate check, tends to be the dominant factor in processing queries, in the literature, there have been much efforts to find optimal bin boundaries that enable to minimize candidate check costs [7, 8].

In many data warehouse applications, the domain of a column tends to be hierarchical, such as categorical data and geographical data. For example, online shopping sites organize different products in terms of categorical hierarchies, because they are effective when users are browsing with the aim of locating a specific product. As a result, the relational schema generated for these sites would contain attributes whose domains are hierarchical in nature. When the domain of a column is hierarchical, by using hierarchically organized bitmap indices, the performance of queries can be significantly improved [10, 11, 12, 19]. For example, [10, 11] leverage hierarchical bitmap indices to efficiently answer OLAP queries with range or set conditions on hierarchical column domains.

In this paper, we focus on a hierarchically organized bitmap index on an attribute whose values are organized into a large-scale hierarchy. In cases where the disk storage is constrained, it is not feasible to build bitmaps for all distinct nodes in the large-scale hierarchy. To deal with these cases, one possible solution is to apply existing binning strategies to hierarchically organized bitmap indices. However, although existing binning techniques are useful for non-hierarchical bitmap indices, the same cannot be said for hierarchical bitmap indices because they do not utilize the structure information inherent in the given hierarchy, which leads to non-optimal binning boundaries.

Thus, in this paper, we propose the binning strategy (HBIN) which aims to reduce the storage space overheads in creating hierarchical bitmap indices, when the disk storage is constrained. In particular, the main contributions of this paper are as followings: We formally define the cost model for answering queries using a hierarchically organized bitmap index. In particular, our cost model estimates the number of pages to be loaded into the main memory from the secondary disk in order to fully resolve a given query by leveraging bitmap indices. By relying on the cost model developed in this paper, we present the binning algorithm to reduce the storage space overheads of hierarchical bitmap indices in space-constrained environments.
PROBLEM DEFINITION
In this section, we introduce the key terminology we will use to develop and describe our algorithms, and then formally define the problem.
A relation, $R$, in a database consists of a set of attributes, $A = \{A_1, A_2, ..., A_p\}$. Since each bitmap index is constructed for a single attribute, we omit the attribute subscript without loss of generality. That is, $A$ is used to refer $A_i$ in the rest of the paper. Let us further assume that the domain of attribute, $A$, is organized into a hierarchical tree structure, $H$. A set of nodes in the hierarchical tree, $H$, is represented as $N_H = \{n_1, n_2, n_3, ..., n_t\}$. In this paper, we consider workloads consisting of COUNT queries with set operations. For example, given the domain hierarchy shown in Figure 1, the COUNT query, $q_1$, asking for the number of sales for the product, “Laptop”, “Tablet”, “Desktop”, or “Smart TV”, is represented as follows:

$$q_1: \text{SELECT COUNT(*)}$$
$$\text{FROM \ ordertbl}$$
$$\text{WHERE ordertbl.prod_name IN}$$
$$\{'\text{Laptop}', '\text{Tablet}', '\text{Desktop}', '\text{'Smart TV'}'\}$$

Although, we focus on queries with set operations, we also note that the proposed algorithm in this paper is applied to workloads consisting of range queries, which can be equivalently represented as set queries. Let $Q = \{q_1, q_2, ..., q_m\}$ be a query workload consisting $m$ queries having set operations. As the set operation in $q_1$ is applied to the attribute, prod_name, whose possible values correspond to the leaf nodes of the hierarchical tree shown in Figure 1, there may exist queries that are semantically equivalent to $q_1$. Now, we formally define such queries as followings: Given a hierarchical tree, $H$, and a query, $q_i$, let $SQ(q_i, H)$ be a set of queries which are semantically equivalent to $q_i$. For example, given $q_1$, we can list the query in $SQ(q_1, H)$ as followings:

$$q_{1.0}: \text{SELECT COUNT(*)}$$
$$\text{FROM ordertbl}$$
$$\text{WHERE ordertbl.prod_name IN}$$
$$\{'\text{Laptop}', '\text{Tablet}', '\text{Desktop}', '\text{'Smart TV'}'\}$$

$$q_{1.r}: \text{SELECT COUNT(*)}$$
$$\text{FROM ordertbl}$$
$$\text{WHERE ordertbl.prod_name IN}$$
$$\{'\text{Computer}', '\text{Smart TV'}'\}$$

Given a query, $q_{ij} \in SQ(q_1, H)$, let us further assume that $N(q_{ij})$ be a set of nodes in $H$ that are involved in set operations of $q_{ij}$; e.g., $N(q_{1,0}) = \{\text{Laptop}, \text{Tablet}, \text{Desktop}, \text{Smart TV}\}$ and $N(q_{1,1}) = \{\text{Computer}, \text{Smart TV}\}$. In other words, $N(q_{ij})$ denotes a set of nodes whose corresponding bitmaps are needed to fully resolve $q_{ij}$.

The binning problem that this paper addresses is stated as follows: Given

- a query workload, $Q$,
- a disk space budget, $S$, allocated for storing the hierarchical bitmap index and
- an attribute, $A$, in relation, $R$, and the corresponding domain tree, $H$, consisting of $N_H = \{n_1, n_2, n_3, ..., n_t\}$

compute a set of bins, $B_H = \{b_1, b_2, ..., b_m\}$ which minimizes the followings:

$$\sum_{q_i \in Q} \text{cost}(q_i, B_H)$$

subject to

$$\sum_{b_i \in B_H} \text{space}(b_i) \leq S,$$

where

- $b_i \in B_H$ consists of a set of nodes in $N_H$,
- $\text{space}(b_i)$ is the amount of disk space needed to store the bitmap for $b_i$
- $\text{cost}(q_i, B_H)$ denotes the cost for answering $q_i$ when the hierarchical bitmap index that is constructed based on $B_H$ is available.

In other words, in disk storage constraints, the hierarchical bitmap index consisting of a set of bitmaps for each bin of $B_H$ is the one that is most beneficial for the query workload, $Q$.

PRELIMINARY: Hierarchical Bitmap Index
In this section, we introduce the key terminology we will use to develop and describe our algorithms, and then formally define the problem. Let us consider the hierarchy, $H$, shown in Figure 1. Furthermore, we assume that the leaf nodes in the hierarchy, $H$, are the actual values in the attribute, $A$. Then, the corresponding hierarchical bitmap index is built as follows (Figure 2):

- The $i$-th value of a bitmap for a leaf node is set to 1, if the value of the attribute, $A$, of the $i$-th record equals to the leaf node. Otherwise, it is set to 0.
- A bitmap for an internal node is constructed by bitwise ORing the bitmaps for all its child nodes. For example, the bitmap for “Computer” is computed by bitwise ORing the bitmaps for “Laptop”, “Tablet”, and “Desktop”.

In order to illustrate the usage of the hierarchical bitmap index in Figure 2, let us consider the example query, $q_1$, in Subsection 2.1. As bitmaps are constructed for all distinct nodes in the hierarchy, the above query can be answered as following:

- The first plan is to bitwise ORing the bitmaps for leaf nodes, “Laptop”, “Tablet”, “Desktop”, and “Smart TV” that appear in the query, which is case for $q_{1.0}$.
- As all child nodes of “Computer” are involved in the set condition, more promising strategy is to bitwise ORing
the bitmaps for “Computer” and “Smart TV”, which is case for \( q_{1,1} \).

As pointed out in [12], all these plans return the same result. However, the cost of processing the query using these different plans would be different from each other, because the I/O costs of reading the bitmaps, which are required for resolving the query, from the disk to the main memory are different from each other.

\[
\text{cost}(q_i, B_H) = \min_{q_{1,j} \in S(q_i, H)} \left( \sum_{b_t \in S(b_{q_{1,i}})} (\text{cost}\_\text{seek}(b_t) + \text{cost}\_\text{load}(b_t)) \right),
\]

where \( \text{cost}\_\text{seek}(b_t) \) is the cost needed to locate the bitmap associated to the bin, \( b_t \), through \( B^*-\text{tree} \) index and \( \text{cost}\_\text{load}(b_t) \) denotes the cost of reading the bitmap associated to \( b_t \) from the secondary disk to the main memory. We note that since the given query, \( q_i \), can be processed in different ways as explained in Section 3, the query plan having the lowest cost is selected, which is inline with the purpose of query optimizer of DBMS.

The cost of locating the bitmap associated to \( b_t \) through \( B^*-\text{tree} \), \( \text{cost}\_\text{seek}(b_t) \), depends on the number of pages needed to be loaded into the main memory in order to reach a leaf node from a root. As \( B^*-\text{tree} \) is usually implemented such that its node size equals to the disk page size, \( \text{cost}\_\text{seek}(b_t) \) is determined by the height of \( B^*-\text{tree} \) as followings:

\[
\text{cost}\_\text{seek}(b_t) = \log_k (\text{Num}\_\text{of}\_\text{Bitmap}) + 1.
\]

Here, \( k \) is approximately estimated as \( (s_{\text{disk-page}} + s_{\text{node-id}}) / (s_{\text{disk-page}} + s_{\text{pointer}} + s_{\text{node-id}}) \) where \( s_{\text{disk-page}} \) and \( s_{\text{pointer}} \) are respectively the size of one disk page and the size of pointer, and \( s_{\text{node-id}} \) is the size of node ID [13].

Similarly, the cost of reading the bitmap associated to \( b_t \), \( \text{cost}\_\text{load}(b_t) \), is computed as the number of pages loaded into the main memory from the secondary disk as following:

\[
\text{cost}\_\text{load}(b_t) = \left\lfloor \frac{\text{space}(b_t)}{s_{\text{disk-page}}} \right\rfloor,
\]

where as defined in Subsection 2.2, \( \text{space}(b_t) \) is the amount of disk space needed to store the bitmap associated to \( b_t \).

**BINNING ALGORITHM FOR HIERARCHICAL BITMAP INDEX**

**A. Modeling query processing cost without candidate checks**

Since bitmaps are stored in the secondary disk, query processing with bitmap indices is performed in two phases: Firstly, all bitmaps necessary for answering an user query are loaded into the main memory from the secondary disk, and secondly, bitwise logical operations on these bitmaps are performed. Especially, when the number of records in the database is huge, the major bottleneck of using bitmap indices is the I/O costs incurred in the first phase and thus, the query processing cost is determined by the number of pages that should be fetched from the secondary disk to the main memory during the first phase. Therefore, the query cost in this paper is modeled as the expected number of disk pages to be loaded into the main memory during the first phase.

Given a node \( n_i \in N_H \), let \( \Phi(n_i) \) be the function that returns a bin \( b_t \) \( \in B_H \) to which \( n_i \) belongs. Furthermore, we assume that \( S\Phi(q_i) \) be a set of bins of \( B_H \) such that

\[
S\Phi(q_i) = \{ b_t | b_t = \Phi(n_i), n_i \in N_H \}
\]

Then, to resolve a given query, \( q_i \), we need to read all bitmaps corresponding to \( b_t \in S\Phi(q_i) \) into the main memory from the secondary disk.

In this paper, we assume that the bitmap indices stored in the disk are accessed by \( B^*-\text{tree} \) as is the case for most commercial database products. Then, given a query \( q_i \), the cost model, \( \text{cost}(q_i, B_H) \), of processing the query \( q_i \) by using hierarchically organized bitmap indices is defined as follows:

**Figure 2. A hierarchical bitmap index constructed for the column hierarchy in Figure 1**

**Figure 2. A hierarchical bitmap index constructed for the column hierarchy in Figure 1**

### Table: Bitmap

<table>
<thead>
<tr>
<th>Rid</th>
<th>A's value</th>
<th>Bitmap</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Laptop</td>
<td>1 0 0 1</td>
</tr>
<tr>
<td>2</td>
<td>Tablet</td>
<td>0 1 0 1</td>
</tr>
<tr>
<td>3</td>
<td>Laptop</td>
<td>0 1 0 1</td>
</tr>
<tr>
<td>4</td>
<td>Desktop</td>
<td>0 0 1 1</td>
</tr>
<tr>
<td>5</td>
<td>Smart TV</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>6</td>
<td>LCD TV</td>
<td>0 0 0 1</td>
</tr>
<tr>
<td>7</td>
<td>Smart TV</td>
<td>0 0 0 1</td>
</tr>
<tr>
<td>8</td>
<td>Tablet</td>
<td>0 1 0 1</td>
</tr>
<tr>
<td>9</td>
<td>Plasma TV</td>
<td>0 0 0 0</td>
</tr>
</tbody>
</table>
Figure 3. (a) An initial set of bins, \( B_H = \{b_1, b_2, \ldots, b_8, b_9\} \), obtained from the hierarchy in Figure 1. (b) two bins, \( b_7 \) and \( b_8 \) (which are associated to leaf nodes, “Smart TV” and “Plasma TV”, of the hierarchy \( H \)), are merged into \( b_7 \), and (c) two bins, \( b_7 \) and \( b_9 \), are further merged, which causes \( b_6 \) and \( b_7 \) to be merged to \( b_3 \).

Given a query \( q_i \), the cost model, \( \text{cost}(q_i, B_H) \), of answering \( q_i \) by using hierarchically organized bitmap indices in the presence of the candidate check is defined as follows:

\[
\text{cost}(q_i, B_H) = \min_{q_{ij} \in \text{SQ}(q_i, H)} \left( \sum_{b_i \in B(q_{ij})} \left( \text{cost}_{\text{seek}}(b_i) + \text{cost}_{\text{load}}(b_i) + \text{cost}_{\text{cand}}(b_i) \right) \right),
\]

where \( \text{cost}_{\text{seek}}(b_i) \) and \( \text{cost}_{\text{load}}(b_i) \) are defined as the same way in Subsection 4.1.1. Here, the cost of the candidate check, \( \text{cost}_{\text{cand}}(b_i) \), is defined as followings:

- if \( b_i \subseteq N(q_{ij}) \), \( \text{cost}_{\text{cand}}(b_i) = 0 \).
- Otherwise,

\[
\text{cost}_{\text{cand}}(b_i) = P \times \left(1 - \frac{1}{P}\right)^{v_{b_i}} \approx P \times (1 - e^{-v_{b_i}P}).
\]

Here, \( P \) denotes the number of pages that are needed to store all records in a database table for which we want to create the hierarchical bitmap index and \( v_{b_i} \) is the number of 1-bits in the bitmap associated to \( b_i \) [8]. Note that in cases where \( b_i \) is a subset of \( N(q_{ij}) \), \( \text{cost}_{\text{cand}}(b_i) \) is set to 0 because the candidate check is not needed. On the other hand, in cases where \( b_i \) is not a subset of \( N(q_{ij}) \), original records whose corresponding bit in the bitmap for \( b_i \) equals to 1 should be retrieved from the secondary disk in order to check whether they indeed satisfy query conditions, which leads to the increased cost of query processing leads to the increased cost of query processing.

C. Algorithm

As described previously, if a large scale domain hierarchy is used and the disk storage is constrained, building a hierarchically organized bitmap index is not feasible, due to the large amount of space overhead. Hence, in this subsection, we present the binning algorithm which aims to reduce the space overhead of a hierarchical bitmap index.

As shown in Algorithm 1, one possible solution to the problem of this paper is to merge nodes of the given hierarchy in a bottom-up manner, until the amount of disk space needed to store the resulting hierarchical bitmap index is less than the space budget. In this case, as one moves down in the hierarchy, the likelihood of nodes merged together increases. As the candidate check might be required for bitmaps associated to bins which are created by merging different nodes of the given hierarchical tree, this strategy is not useful for workloads consisting of queries, many of which refer to nodes closed to the leaves of the hierarchy.

<table>
<thead>
<tr>
<th>Algorithm 1: Bottom-up node merging method</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Data</strong>: Hierarchical tree ( H ), Space budget ( S ), Query workloads ( Q )</td>
</tr>
<tr>
<td><strong>Result</strong>: A set of bins ( B_H )</td>
</tr>
<tr>
<td><strong>initBins</strong> ( (H, B_H) )</td>
</tr>
<tr>
<td>while ( \text{computeSpace}(B_H) &gt; S ) do</td>
</tr>
<tr>
<td>( \text{SetLeafBins} = \text{findLeafBins}(B_H, H) );</td>
</tr>
<tr>
<td>( \text{cost}_{\text{best}} \leftarrow \infty );</td>
</tr>
<tr>
<td>( B_{H, \text{best}} \leftarrow \emptyset );</td>
</tr>
<tr>
<td>for ( b_i, b_j \in \text{SetLeafBins} ) do</td>
</tr>
<tr>
<td>if ( b_i ) and ( b_j ) have the same parent then</td>
</tr>
<tr>
<td>( B_{H, \text{cur}} \leftarrow B_{H, \text{best}} );</td>
</tr>
<tr>
<td>( \text{updateBins}(B_{H, \text{cur}}, b_i, b_j) );</td>
</tr>
<tr>
<td>( \text{cost}<em>{\text{cur}} \leftarrow \text{getCost}(Q, B</em>{H, \text{cur}}) );</td>
</tr>
<tr>
<td>if ( \text{cost}<em>{\text{cur}} &lt; \text{cost}</em>{\text{best}} ) then</td>
</tr>
<tr>
<td>( \text{cost}<em>{\text{best}} \leftarrow \text{cost}</em>{\text{cur}} );</td>
</tr>
<tr>
<td>( B_{H, \text{best}} \leftarrow B_{H, \text{cur}} );</td>
</tr>
<tr>
<td>end</td>
</tr>
<tr>
<td>end</td>
</tr>
<tr>
<td>end</td>
</tr>
</tbody>
</table>

In hierarchically organized bitmap indices, creating the bitmap corresponding to a non-leaf node of the given hierarchical tree is to enhance query performance by avoiding
to read all bitmaps associated to its descendant leaves from the disk and to perform bitwise operations on them. For example, the query, $q_1$, in Section 2 can be more efficiently answered by leveraging the bitmap associated to the non-leaf node, “Computer” than by using three bitmaps corresponding to “Laptop”, “Tablet”, and “Desktop”. On the other hand, the bitmap corresponding to the non-leaf node, either “TV” or “Electronics”, is not useful for the same query, $q_1$. This implies that the usefulness of the bitmaps corresponding to non-leaf nodes of the hierarchy is dependent on query workloads. Thus, if we know that query workloads follow some known pattern or distribution, we would effectively solve the space overhead problem of hierarchically organized bitmap indices by avoiding to build the bitmaps on those non-leaf nodes that are rarely or never used by query workloads.

**Algorithm 2: Proposed HBIN algorithm**

```
Algorithm 2: Proposed HBIN algorithm

Data: Hierarchical tree $H$, Space budget $S$, Query workloads $Q$
Result: A set of bins $B_H$

1. $\text{initBins}(H, B_H)$;
2. $\text{Set.UpperMost} \leftarrow \emptyset$;
3. add a bin associated to the root node of $H$ into $\text{Set.UpperMost}$;
4. while (computeSpace($B_H$) $>$ $S$) do
   5. /*Compute the best cost obtained by merging bins*/
      6. $\text{Set.LeafBins} = \text{findLeafBins}(B_H, H)$;
      7. $\text{cost}_{\text{best, bottom-up}} \leftarrow \infty$;
      8. $B_H, \text{best, bottom-up} \leftarrow \emptyset$;
      9. $\text{for } b_i, b_j \in \text{Set.LeafBins} \text{ do}$
         10. if $b_i$ and $b_j$ have the same parent then
               11. $B_H, \text{cur} \leftarrow B_H$;
               12. $\text{updateBins}(B_H, \text{cur}, b_i, b_j)$;
               13. $\text{cost}_{\text{cur}} = \text{getCost}(Q, B_H, \text{cur})$;
               14. if ($\text{cost}_{\text{cur}} < \text{cost}_{\text{best, bottom-up}}$) then
                   15. $\text{cost}_{\text{best, bottom-up}} \leftarrow \text{cost}_{\text{cur}}$;
                   16. $B_H, \text{best, bottom-up} \leftarrow B_H, \text{cur}$;
         17. end
      18. end/*Compute the best cost obtained by removing a non-leaf bin*/
      19. $\text{cost}_{\text{best, top-down}} \leftarrow \infty$;
      20. $B_H, \text{best, top-down} \leftarrow \emptyset$;
      21. $\text{for } b_i \in \text{Set.UpperMost} \text{ do}$
         22. $B_H, \text{cur} \leftarrow B_H$;
         23. remove $b_i$ from $B_H, \text{cur}$;
         24. $\text{cost}_{\text{cur}} = \text{getCost}(Q, B_H, \text{cur})$;
         25. if ($\text{cost}_{\text{cur}} < \text{cost}_{\text{best, top-down}}$) then
             26. $\text{cost}_{\text{best, top-down}} \leftarrow \text{cost}_{\text{cur}}$;
             27. $B_H, \text{best, top-down} \leftarrow B_H, \text{cur}$;
             28. $\text{removedBin} \leftarrow b_i$;
         29. end/*Compare the query costs*/
      30. if ($\text{cost}_{\text{best, bottom-up}} < \text{cost}_{\text{best, top-down}}$) then
          31. $B_H \leftarrow B_H, \text{best, bottom-up}$;
          32. $\text{updateHierarchy}(B_H, H)$;
      33. else
          34. $B_H \leftarrow B_H, \text{best, top-down}$;
          35. remove $\text{removedBin}$ from Set.UpperMost;
          36. add bins associated to child nodes of the node corresponding to $\text{removedBin}$ into Set.UpperMost;
      37. end
```

9293
Based on this intuition, we further refine the bottom-up method in Algorithm 1 in a way that nodes of the given hierarchical tree are merged in a bottom-up manner as the same way of Algorithm 1, and at the same time, non-leaf nodes whose corresponding bitmaps are not very beneficial for query workloads are identified in top-down direction. In this way, we are able to reduce the number of nodes merged in a bottom-up manner, while still benefiting from the bitmaps associated to non-leaf nodes of the hierarchy which are frequently used for resolving queries.

In Algorithm 2, we outline the proposed binning algorithm which leverages the cost model developed in Subsection 4.1. In line 1 of the pseudo-code, the algorithm first initializes a set of bins, \( B_H \), which is the output of the algorithm. Given a domain hierarchical tree \( H \) having a set of nodes \( N_H = \{ n_1, n_2, n_3, \ldots, n_t \} \), the function, \( \text{initBins}(H, B_H) \), initializes \( B_H \) such that each bin, \( b_i \in B_H \), contains one node, \( n_i \in N_H \) and thus, \( |N_H| = |B_H| \). In lines 2-3, \( \text{Set}_{\text{UpperMost}} \), which is a set of bins associated to current uppermost internal nodes of \( H \), is initialized. Note that \( \text{Set}_{\text{UpperMost}} \) is used by the algorithm to identify non-leaf nodes whose corresponding bitmaps are not beneficial for query workloads in top-down direction. Then in lines 4-43, the algorithm iteratively either merges bins in a bottom-up manner or remove one bin from \( B_H \) in a top-down manner, until the amount of disk space required to store the resulting bitmap index is less than the given space budget \( S \):

- **Bottom-up bin merging:**
  - In line 5, \( \text{Set}_{\text{leafBins}} \) is set to a set of bins which are associated to the leaf nodes in the current hierarchy \( H \). The algorithm should recompute \( \text{Set}_{\text{leafBins}} \) at each iteration, because as shown in Figure 3, the hierarchy \( H \) may be updated at the end of each iteration, due to the process of merging bins.
  - The condition on line 10 checks whether the corresponding nodes of \( b_i \) and \( b_j \) in \( \text{Set}_{\text{leafBins}} \) have the same parent on the current hierarchy \( H \). The purpose of this check is to disallow to merge bins that belong to different branch of the hierarchy.
  - In line 12 of the pseudo-code, we update a set of bins, \( B_{H,\text{cur}} \) (which is created by a deep copy of \( B_H \)) by merging two leaf bins \( b_i \) and \( b_j \). If as shown in Figure 3-(c), the parent of the nodes associated to \( b_i \) and \( b_j \) has only two child nodes (i.e., \( b_i \) and \( b_j \)), we can simply remove \( b_i \) and \( b_j \) from \( B_H \), because in that case, the bitmap obtained by merging \( b_i \) and \( b_j \) is equivalent to one associated to the parent node in the hierarchy. Otherwise, as shown in Figure3-(b), the algorithm removes \( b_i \) from \( B_H \) and updates \( b_i \) by unioning with \( b_j \) (where \( i < j \)).
  - In line 13, by leveraging the cost model developed in Subsection 4.1, the algorithm computes the query cost with the hierarchical bitmap index based on \( B_{H,\text{cur}} \). If the current query cost (\( \text{cost}_{\text{cur}} \)) is cheaper than the so far best one (\( \text{cost}_{\text{best,bottom-up}} \)), the algorithm updates the best cost and the corresponding set of bins, \( B_{H,\text{best,bottom-up}} \), respectively in lines 14-17.

- **Top-down bin elimination:** In lines 21-33, the algorithm iteratively computes the query costs obtained by removing one bin in \( \text{Set}_{\text{UpperMost}} \) from \( B_H \), and then picks the best one (\( \text{cost}_{\text{best,top-down}} \)).

- **Query cost comparison:** In lines 35-42, the algorithm compares the best query cost (\( \text{cost}_{\text{best,bottom-up}} \)) obtained by the bottom-up bin merging with the best query cost (\( \text{cost}_{\text{best,top-down}} \)) obtained by the top-down bin elimination. If the bottom-up bin merging is cheaper than the top-down bin elimination, then we update the hierarchy \( H \) so that it reflects the merge of bins in \( B_H \) in lines 36-37. Otherwise, in lines 39-41, \( \text{Set}_{\text{UpperMost}} \), which is a set of bins associated to current uppermost internal nodes of \( H \), is updated.

**EXPERIMENT**

In this section, we describe the experiments we carried out to evaluate the efficiency of the binning approach.

**A. Experimental Setup**

In order to evaluate the proposed approach in a controlled manner, we systematically generated a large number of data, queries and hierarchies with varying parameters. Table 1 provides an overview of the systematic data, hierarchy and query sets used in these experiments. We have created a workload of 200 queries with different leaf node coverages. For example, 10% leaf node coverage indicates that 10% of leaf nodes in a given hierarchy are referenced in the set operations of each query. In order to simulate a scenario in which query workloads follow some known patterns or distributions, we generated each query in a way that the probability of each leaf node of the hierarchy being used in the query workload follows Zipf’s law. Among 200 queries, the first 150 queries are used as a training data set to compute a set of bins, and the remaining 50 queries are used to evaluate the efficiency of the bitmap index which is generated based on the resulting set of bins. In the experiments, we report results obtained by using following alternatives:

- **case HBIN:** This is the proposed binning scheme in this paper (Algorithm 2).
- **case BU:** This is the bottom-up bin merging method explained in Algorithm 1.
- **case RBU:** Like BU, this is a bottom-up bin merging approach. However, RBU randomly merges bins in a bottom-up manner without leveraging the query workload.
- **case NON-HBIN:** This is a non-hierarchical bitmap index scheme. NON-HBIN first enumerates all leaves of a given hierarchy from left to right, and then computes a set of bins using the method developed in [8].

In order to evaluate the efficiency of the proposed algorithm, we measured the average number of accessed disk pages that are necessary for answering each query.
Table 1. Data, query and hierarchy sets used in the experiments

<table>
<thead>
<tr>
<th>Data</th>
<th># of records</th>
<th>5M</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>record size</td>
<td>200 bytes</td>
</tr>
<tr>
<td></td>
<td>page (block) size</td>
<td>16384 bytes</td>
</tr>
<tr>
<td>Hierarchy</td>
<td>height</td>
<td>5, 6</td>
</tr>
<tr>
<td></td>
<td>fanout</td>
<td>3</td>
</tr>
<tr>
<td>Query</td>
<td># of queries</td>
<td>200</td>
</tr>
<tr>
<td></td>
<td>leaf node coverage</td>
<td>10%, 20%</td>
</tr>
</tbody>
</table>

B. Experimental Results

Figure 4 shows the average number of accessed disk pages per each query on various parameters. In this experiment, we set the number of records in the database to 5M. Key observations based on these charts, the proposed HBIN outperforms other alternatives on various parameters. In particular, among the three alternatives (HBIN, BU, and NON-HBI) that leverage the query workload in computing a set of bins, based on which the corresponding bitmaps are generated, HBIN scheme proposed in this paper shows the best performance. The bottom-up bin merging scheme, BU, is even worse than NON-HBI which is a non-hierarchical bitmap index scheme. This is because the BU scheme generates all the bitmaps associated to non-leaf nodes of the given hierarchy, even though they are not useful for the query workload, which leads to the increased number of bins merged together, and thus the increased candidate check costs per each query. On the other hand, the proposed HBIN scheme enables to minimize the number of bins merged in a bottom-up manner, because it does not build bitmaps associated to non-leaf nodes that are less frequently used by the query workload, which results in the reduced candidate check costs, comparing with the BU scheme.

Figure 4 also shows that as the height of the hierarchy increases, the average number of accessed disk pages per each query decreases. The reason for this is that if the height of the hierarchy increases, then the number of nodes increases, which in turn leads to a decrease in the average number of 1-bits in each bitmap. Because the costs of candidate check are proportional to the number of 1-bits in each bitmap, the lower number of 1-bits in each bitmap means less expensive candidate check costs.

RELATED WORK

There have been a number of proposals for overcoming space overhead of bitmap indices for high-cardinality data columns. The most well-known method for reducing space overhead of bitmap indices is binning strategy. Given a set of user queries, [6, 7, 8] introduced the dynamic programming based algorithm that aims to efficiently compute the bin boundaries. The bin boundaries in [6, 7, 8] are determined in a way that the number of candidate check incurred by binning method is minimized. [9] presented a dynamic bucket expansion and contraction approach to identify bin boundaries for high cardinality attributes with skew. In this work, we also aim to...
compute bin boundaries by leveraging query workloads. Yet, there are fundamental differences between the above approaches and the proposed HBIN in this paper. The binning strategies in [6, 7, 8, 9] aim to identify bin boundaries for the non-hierarchical attributes, while the proposed HBIN focuses on the attributes whose values are organized as a hierarchy.

Various bitmap compression techniques have been studies in the literature. The most popular two methods of bitmap compression techniques include Byte-Aligned Bitmap Compression (BBC) [14, 15] and Word-Aligned Hybrid (WAH) [4, 16]. The BBC approach divides a given sequence into chunks of fixed-size bytes and then groups the chunks into runs which are composed of a fill followed by a tail of literal bytes. The WAH code is similar with BBC. However, in WAH, compressed data are stored in words, instead of bytes. Generally speaking, BBC encoding-based method shows a better performance than WAH-based scheme in query processing times. Beside BBC and WAH schemes, the Run Length Huffman (RLH) which is based on the Huffman encoding and run-length encoding is developed in [17]. There are also a number of empirical researches that compare the performance of various compression methods [1, 18].

CONCLUSION

In this paper, we proposed a binning strategy (HBIN) for hierarchical bitmap indices, when the large scale hierarchy is used. Given the column domain hierarchy, our proposed binning approach merges nodes of the hierarchy in a bottom-up manner, and at the same time identifies non-leaf nodes whose corresponding bitmaps are not beneficial for the query workload in a top-down manner. Therefore, with the proposed algorithm in this paper, we are able to reduce the number of nodes merged together, while still benefiting from the bitmaps associated to non-leaf nodes of the hierarchy which are frequently used in resolving queries. Experiment results show that the proposed technique provides significant performance gains over other alternative solutions.

REFERENCES