HCS: Hierarchical Cut Selection for Efficiently Processing Queries on Data Columns using Hierarchical Bitmap Indices *

Parth Nagarkar
School of Computing, Informatics, and Decision Systems Engineering
Arizona State University
Tempe, AZ 85287-8809, USA
nagarkar@asu.edu

K. Selçuk Candan
School of Computing, Informatics, and Decision Systems Engineering
Arizona State University
Tempe, AZ 85287-8809, USA
candan@asu.edu

ABSTRACT

When data are large and query processing workloads consist of data selection and aggregation operations (as in online analytical processing), column-oriented data stores are generally the preferred choice of data organization, because they enable effective data compression, leading to significantly reduced IO. Most column-store architectures leverage bitmap indices, which themselves can be compressed, for answering queries over data columns. Column-domains (e.g., geographical data, categorical data, biological taxonomies, organizational data) are hierarchical in nature, and it may be more advantageous to create *hierarchical bitmap indices*, that can help answer queries over different sub-ranges of the domain. However, given a query workload, it is critical to choose the appropriate subset of bitmap indices from the given hierarchy. Thus, in this paper, we introduce the *cut-selection* problem, which aims to help identify a subset (*cut*) of the nodes of the domain hierarchy, with the appropriate bitmap indices. We discuss *inclusive*, *exclusive*, and *hybrid* strategies for cut-selection and show that the hybrid strategy can be efficiently computed and returns optimal (in terms of IO) results in cases where there are no memory constraints. We also show that when there is a memory availability constraint, the cut-selection problem becomes difficult and, thus, present efficient cut-selection strategies that return close to optimal results, especially in situations where the memory limitations are very strict (i.e., the data and the hierarchy are much larger than the available memory). Experiment results confirm the efficiency and effectiveness of the proposed *cut-selection* algorithms.

1. INTRODUCTION

Range selection queries are frequent in many applications, including online analytical processing (OLAP) scenarios, where an aggregation operation needs to be applied over a certain range of data [1]. When data are large and the query processing workloads consist of such data selection and aggregation operations, column-oriented data stores are generally the preferred choice of data organization, especially because they enable effective data compression, leading to significantly reduced IO [2].

Recently, many databases have leveraged bitmap-indices, which themselves can be compressed, for efficiently answering queries [3], [4]. When column-domains (e.g., geographical data, categorical data, biological taxonomies, organizational data) are hierarchical in nature [5], it is often more advantageous to create *hierarchical bitmap indices* to efficiently answer queries over different sub-ranges of the domain. [5] for example proposes a hierarchically organized bitmap index (HOBI) for answering OLAP queries over data with hierarchical domains.

In this paper, we also focus on hierarchically organized bitmap indices for answering queries over column-oriented data and present efficient algorithms for selecting the subset of bitmap indices to answer queries efficiently over compressed data columns. Before we detail the contributions of this paper in Section 1.2, we first provide an overview of the related work in the area.

1.1 Related Work

Range and Aggregation Queries over Data Columns. As mentioned above, column-oriented data stores are generally the preferred choice of data organization for aggregation queries over single attribute columns, because they enable effective data compression, leading to significantly reduced IO [2]. Range queries are used in data warehouse environments to perform aggregations over a specified range for analysis. Research has been done on creation of specific data structures that can better the performance of range queries. In [6], the authors propose an update to an existing data structure to store the aggregate values of the leaf nodes in the internal nodes. This reduces the lookup at the leaf nodes if all the leaf nodes fall under a range of an internal node in the query. Their drawback is that the proposed approach stores aggregation values even for queries that do not require aggregation, thus degrading their performance. In [7], the authors build upon the work described in [6], and create a more generalized data structure that leverages upper levels to help aggregate queries but does not store the aggregate values for queries that do not require any aggregation. In [1], the authors present algorithms to solve range queries for two types of aggregation operations, sum and max, by using precomputed max over balanced hierarchical tree structures. Caching results of query data has also been looked into, particularly for column store environments. In [8], the authors' main focus is to develop a system that can cache aggregate results as well

*This work is supported by NSF grant #1116394 "RanKloud: Data Partitioning and Resource Allocation Strategies for Scalable Multimedia and Social Media Analysis"

10.5441/002/edbt.2014.26

as be able to handle transactional and analytical workloads in one system. In [9], the authors present a hierarchical structure to efficiently execute range-sum queries. Their focus is on reducing number of cell accesses per query and improving update performance specifically in a data cube. In this paper, we present algorithms that choose specific bitmap indices to be cached in the memory to speed up a given query workload.

Bitmap Indices. Bitmap indices have been used in OLAP queries and data warehouses for their benefit of compression. There has been significant amount of work to improve the performance of bitmap indices as well as keeping the compression rates high [10], [11], [12]. Most of the newer compression algorithms use run-length encoding for compression: it provides a good compression ratio and one can do bitwise operations directly on decompressed bitmaps without actually decompressing them [10]. Recently, researchers have shown that bitmap indices perform well even on high-cardinality attributes [13].

Multi-level Indices. There has also been considerable research done in the area of multi-level indices [14], [15], [16]. In data warehouse applications, bitmap indices are also shown to perform better than traditional database index structures like the B-tree [10], [17]. Our work focuses on which bitmaps to read and cache in the memory from a given hierarchy. We introduce novel algorithms that choose these bitmaps efficiently. As far as we know, most of the existing approaches use what we term as an *inclusive* strategy: they first identify upper level bitmaps for retrieving the data that fully satisfies the given query and the lower level bitmaps to satisfy the boundary bitmaps in the hierarchy that the upper level bitmaps could not be used to give an exact answer [11]. As discussed in the next subsection, however, we generalize the problem into a *cut-selection problem* and introduce *exclusive* and hybrid strategies that complement *inclusive* result construction. The works [5] and [18] focus on building hierarchies on dimensions, specifically in a data warehouse environment, to efficiently execute range queries. [19] deals with the similar problem of choosing the appropriate set of bitmap join indices of one or more attributes using data mining techniques; we on the other hand focus on choosing the appropriate set of bitmap indices for a given domain hierarchy.

1.2 Contributions of this Paper

Since IO is often the main bottleneck in processing OLAP workloads over large data sets, given a query or a workload consisting of multiple queries, the main challenge in leveraging hierarchically organized bitmap indices is to choose the appropriate subset of bitmap indices from the given hierarchy to process the query. [5], for example, proposes a (what we term as an "*inclusive*") strategy which leverages bitmap indices associated to the internal nodes along with the bitmap indices associated to the data leaves to bring together the data elements needed to answer the query.

In this paper, we note that such inclusive strategies can be sub-optimal. In fact, [5] shows that the inclusive strategy is effective mainly for small query ranges. Therefore, in this paper, we introduce a more general *cut-selection* problem, which aims to help identify a subset (referred to as a *cut*) of the nodes of the domain hierarchy, which contain the operations nodes with the appropriate bitmap indices to efficiently answer queries. In particular, we discuss *inclusive*, *exclusive*, and *hybrid* strategies for cut-selection (Section 3.1) and experimentally show that the so-called *exclusive strategy* provides gains when the query ranges are large and that the *hybrid strategy* provides best solutions across all query range sizes, improving over the *inclusive* strategy even when the ranges of interest are relatively small (Section 4.1). We also show that the *hybrid strategy* can be efficiently computed for a single query or a workload of multiple queries and also that it returns optimal (in terms of IO) results in cases where there are no memory constraints (Section 4.2).

However, in cases where the memory is constrained, the cut-selection problem becomes difficult to solve. To deal with these cases, in Section 2.3.4, we present efficient cut-selection strategies that return close to optimal results, especially in situations where the memory limitations are very strict (i.e., the data and the hierarchy are much larger than the available memory).

Experiment results presented in Section 4 confirm the efficiency and effectiveness of the proposed *cut-selection* algorithms.

2. PROBLEM SPECIFICATION

In this section, we first introduce the relevant concepts and notations, provide a cost model, and introduce the *cut-selection* problem for identifying a subset of the nodes of the domain hierarchy, containing the nodes with the bitmap indices to efficiently answer a given query or a query workload.

2.1 Key Concepts, Parameters, and Notations

We first provide an overview of the concepts and parameters necessary to formulate the problem described in this paper and introduce the relevant notations.

2.1.1 Columns and Domain Hierarchies

A database consists of relations, $\mathcal{R} = \{R_1, \ldots, R_{maxr}\}$. Each relation, R_r, consists of a set of attributes, $\mathcal{A}_r = \{A_{r,1}, \ldots, A_{r,maxa_r}\}$, with domains $\mathcal{D}_r = \{D_{r,1}, \ldots, D_{r,maxa_r}\}$ In this paper, without loss of generality, we associate to each attribute, $A_{r,a}$, a corresponding hierarchy, $H_{r,a}$, which consists of a set of nodes, $\mathcal{N}_{r,a} = \{N_{r,a,1}, \ldots, N_{r,a,maxn_{r,a}}\}$. Also, since our goal is to efficiently answer queries over a single data column, unless necessary, we omit explicit references to relation R_r and attribute $A_{r,a}$; hence, when we do not need to refer to a specific relation and attribute, we simply omit the relation and attribute subscripts; e.g., we refer to H instead of $H_{r,a}$.

In this paper, when talking about the nodes of a domain hierarchy H, we use the following notations:

- **Parent of a node:** For all N_*, $parent(N_*)$ denotes the parent of N_* in the corresponding hierarchy; if N_* is the root, then $parent(N_*) = \perp$.
- **Descendants of a Node:** The set of descendants of node n in the corresponding hierarchy is denoted as $desc(n)$.
- **Leaves:** L_H denotes the set of leaf nodes of the hierarchy H. Any other node in H that is not a leaf node is called an internal node. The set of internal nodes of H is denoted by I_H. *We assume that only the leaves of a hierarchy occur in the database.*
- **Leaf Descendants of a Node:** Leaf descendants of a node are the set of nodes such that they are leaf nodes as well as descendants of the given node; i.e., for a node n, $leafDesc(n)$ returns a set of nodes such that

$$\forall_{b \in leafDesc(n)} b \in L_H \wedge b \in desc(n).$$

2.1.2 Query Workload

In this paper, we focus on query workloads with range queries on an attribute (i.e., column) of the database relations:

- **Range Specification:** Given an attribute A_a and the start and end points, i and j, we denote the corresponding range specification as, $rs_{a,i,j}$.

  Given two range specifications, $rs_{a,i,j}$ and $rs_{a,k,l}$,

  - if $k > j$, then these two range specifications are **disjoint**,
  - if $(i < k, l) \wedge (j > k) \wedge (j < l)$, then the two range specifications are **intersecting**, and
  - if $(i < k, l) \wedge (j > k, l)$, then the two range specifications are **overlapping**.

- **Range Queries:** Each query q involves fetching one or more sets of column values, such that each set of values belongs to a continuous range over the domain hierarchy of the attribute.

  A query, q, can have multiple range specifications. The set of range specifications for a query q is denoted as RS_q. Without loss of generality, we assume that all range specifications in RS_q are *disjoint*. If a query has two intersecting or overlapping range specifications, $rs_{a,i,j}$ and $rs_{a,k,l}$, then we partition the query into two subqueries, q_1 and q_2, such that range specification for q_1 is $rs_{a,i,j}$ and specification for q_2 is $rs_{a,k,l}$. In Sections 3.2 and 3.3, we discuss algorithms for handling multiple queries.

- **Range Nodes:** Given a range specification, $rs_{a,i,j}$, the set of *leaf nodes* that fall in this range is denoted as, $RN_{a,i,j}$. These nodes are also referred to as *range nodes*.

  Given a query, q, and a node n, $G_{q,n} \in \{0, 1\}$ denotes whether the node n is a range node for query q. More specifically, if node n is a range node for any range specification in RS_q, then $G_{q,n} = 1$ and otherwise, $G_{q,n} = 0$.

  The set of all range nodes for any range specification of query q is denoted as RN_q. If RN_q is empty, the query returns null, whereas if RN_q has the exact same nodes as L_H, then the query returns the entire database content for the attribute on which H is defined.

2.1.3 Hierarchically Organized Bitmap Indices

As described above, the query workload includes queries that fetch ranges of values from columns of relations in the database, before performing further operations on these ranges. When bitmap indices are available, these operations are implemented in terms of bitmap manipulations [11]: for example, intersection of two range queries can be performed as bitwise-AND of two bitmap indices representing the database values in the two ranges. This ensures that those data objects that will be pruned as a result of the query processing are never fetched into memory. In this paper, we assume that indices are organized hierarchically; i.e., every node n in H has a corresponding bitmap B_n denoting which of the leaf nodes of n occur in attribute A of the database.

- **Bitmap Density:** Each bitmap B_n has a bit density, $0 \leq D_{B_n} \leq 1$, denoting ratio of bits set to 1 to the size in the bitmap.

Note that bitmap B_n *may or may not* have been materialized in the form of a bitmap index in the database.

2.2 Cost Model and Query Plans

Especially when the data sets are large, the bitmaps are often stored in a compressed manner and the various bit-wise operations are performed on compressed versions of the bitmap indices, further boosting the query performance [10]. In general, the time taken to read the bitmaps from secondary storage into the memory dominates the overall bitwise manipulation time [11], [20]. The cost of this process is proportional to the size of the bitmap file on the secondary storage; the larger the size of a bitmap file on a secondary storage, the longer it takes to bring the bitmap into the physical memory.

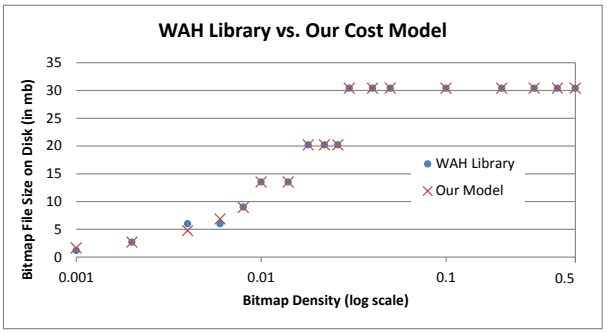


Figure 1: Comparison of our Cost Model and WAH Library Model. $D_{x_1} = 0.01$, $D_{x_2} = 0.015$, $D_{x_3} = 0.03$, and $a = 1043, b = 0.5895$, on a 500 GB SATA Hard Drive with 7200 RPM, and 16 MB Buffer Size.

2.2.1 Read Cost of Compressed Bitmap Indices

Therefore, in this paper, we model the cost of a bitmap operation as proportional to the size of the corresponding (compressed) bitmap file, which in turn determines the time taken to read a bitmap into the memory. Note that in general the query performance of a bitmap index with density greater than 0.5 is equivalent to the performance of a bitmap with density complement to the original [21]. For example, performance of a bitmap with density 0.7 is often equivalent to the performance of a bitmap with density 0.3. This is because a bitmap with density 0.7 can be negated and stored as a bitmap with density 0.3. We also include this behavior in our cost model, $readCost(B_n)$, of reading a bitmap index, B_n, as follows:

$$readCost(B_n) = \begin{cases} 0 & \text{if } D_{B_n} = 0 \vee D_{B_n} = 1 \\ aD_{B_n} + b & \text{if } (0 < D_{B_n} \leq D_{x_1}) \vee (1 - D_{x_1} \leq D_{B_n} < 1) \\ k_1 & \text{if } (D_{x_1} < D_{B_n} \leq D_{x_2}) \vee \\ & (1 - D_{x_2} \leq D_{B_n} < 1 - D_{x_1}) \\ k_2 & \text{if } (D_{x_2} < D_{B_n} \leq D_{x_3}) \vee \\ & (1 - D_{x_3} \leq D_{B_n} < 1 - D_{x_2}) \\ k_3 & \text{otherwise} \end{cases}$$

Here D_{B_n} is the bit density, $0 < D_{x_1} < D_{x_2} < D_{x_3} < 0.5$ are three bit density thresholds, and a, b, k_1, k_2, and k_3 are constants.

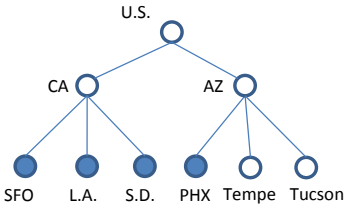
Intuitively, when the bit density of a bitmap is 0 or 1, the size of the bitmap on the disk is very negligible due to the high-level of compression. Hence, we assume the size of these bitmaps as non-existant on the secondary storage. The bit density thresholds, D_{x_1}, D_{x_2}, and D_{x_3}, and the constant values, a, b, k_1, k_2, and k_3, are specific to the implementation of the bitmap library.

Figure 1 shows alignment of our cost model with the read cost of the WAH library for different bit densities.

2.2.2 Inclusive, Exclusive, and Hybrid Query Plans

For a query plan for q, we define the set of nodes that are required

to execute q as its *operation nodes*. Naturally, a given query can be executed in various different ways, each with a different set, ON_q, of *operation nodes*. In particular, in this paper, we consider two distinct types of query plans: *inclusive* and *exclusive* plans.



Consider the 3-level location hierarchy, H, shown above. Here, the leaf nodes (cities in U.S.) are the actual values in the database. The node $U.S.$ is the root node of the hierarchy. Let us consider a query q that has a set of range nodes (shaded nodes in the figure) RN_q =[SFO, $L.A.$, $S.D.$, PHX]. Assume that we have bitmap indices for all the nodes of H. There are at least two different plans of executing q:

- *Inclusive query plans:* The first plan is to combine a subset of the bitmaps of H. In the above example, one inclusive way to do this would be to combine the bitmaps of RN_q.

  Another inclusive plan would be to combine the bitmaps of CA and PHX (i.e. CA OR PHX). Note that this strategy is similar to what was reported in the literature [5].

- *Exclusive query plans:* Alternatively, we can remove the bitmaps of the non-range nodes of q from the relevant internal nodes of H. For instance, in this example, we can achieve this by first performing a bitwise-OR operation on the bitmaps of $Tempe$ and $Tucson$ and then doing a bitwise-ANDNOT operation between the bitmap of $U.S$ and the resultant bitmap from the OR operation (i.e. $U.S$ ANDNOT ($Tempe$ OR $Tucson$)).

  Another exclusive plan would be to do the following: CA OR (AZ ANDNOT ($Tempe$ OR $Tucson$).

It is easy to see that all four plans would return the same result; however, these plans have different operation nodes: for the first inclusive query plan, the operation nodes are ON_q = [SFO, $L.A.$, $S.D.$, PHX], whereas for the second inclusive query plan, ON_q = [CA, PHX]. Similarly, for the first exclusive query plan ON_q = [$U.S.$, $Tempe$, $Tucson$], and for the second exclusive query plan ON_q = [CA, AZ, $Tempe$, $Tucson$]. As a result, each execution plan also requires different amount of data being read.

In this paper, we consider inclusive and exclusive strategies for answering range queries using hierarchical bitmaps. We also consider *hybrid* strategies, which combine inclusive and exclusive strategies (that may make inclusive or exclusive decisions at different nodes of the hierarchy) for better performance.

2.3 Cut Selection Problem

As described above, any range query, q, on hierarchy H, can be answered (through inclusive, exclusive, and hybrid strategies) using bitmap indices for the leaves of the hierarchy. We note however that, if we are also given the bitmap indices for a subset of the internal nodes of the hierarchy, we may be able to reduce the overall cost of the query significantly by also leveraging the bitmap indices for these internal nodes. We refer to these subsets as *cuts* of the hierarchy.

2.3.1 Query Processing with Cuts

We define a *cut*, c, as a subset of internal nodes (including the root node) in a hierarchy, H, satisfying the following two conditions:

- *validity:* there is exactly one node on any root-to-leaf branch in a given cut (note that, by this definition, the set containing only the root node of the hierarchy by itself is a cut); and

- *completeness:* the nodes in c collectively cover every possible root-to-leaf branch in the given hierarchy, H.

If a set of internal nodes of H only satisfies the first condition, then we refer to the cut as an *incomplete cut*.

The challenge of course is to select the appropriate cut c of the hierarchy H that will minimize the query processing cost, but will not add significant memory overhead (if the memory is a constraint). We discuss the alternative formulations of the cut-selection problem, next.

2.3.2 Cut Selection Case 1: Single Query without Memory Constraints

The simplest scenario is identifying the cut necessary to execute a single range query. As we explained earlier, the cost for executing a query is proportional to the size of the bitmaps that are read into the memory from the secondary storage. Thus, given a query q and cut c on H, problem

$$cost(c,q) = \underset{ON_q \subseteq (c \cup L_H)}{MIN} \left\{ \sum_{n \in ON_q} readCost(B_n) \right\} \quad (1)$$

denotes the *best execution cost* for query q given the bitmaps for the leaves, and the cut c.

The cut-selection problem for a given query q on hierarchy H can be formulated as *finding a cut c such that $cost(c,q)$ is the smallest among all cuts of the hierarchy H.*

2.3.3 Cut Selection Case 2: Multiple Queries without Memory Constraints

In general, we are not given a single range query, but a set of range queries that need to be executed on the same data set. Therefore, we need to generalize the above formulation to scenarios with multiple range queries. If we are given a set, Q, of queries on hierarchy H, then one way to formulate the cut-selection problem is *to search for a cut c such that $cost(c,Q)$, defined as*

$$cost(c,Q) = \sum_{q \in Q} cost(c,q) \quad (2)$$

is the smallest among all cuts of the hierarchy H.

Note, however, that this formulation treats each query independently and implicitly assumes that each query plan accesses the bitmaps of its operation nodes from the secondary storage; i.e. it pays the cost of reading a bitmap from the secondary storage every time the node is needed for query processing. This will obviously be redundant when the different queries can be processed using the same operation nodes: in such a case, *it would be best to bring the bitmap for the operation nodes to the memory and keep it to process all the relevant queries.*

This, however, changes the problem formulation significantly; in particular, we now need *to search for a cut c such that $cost'(c,Q)$,*

defined as

$$\left(\sum_{n\in c} readCost(B_n)\right) + \left(\sum_{n\in(\cup_{q\in Q} ON_q)/c} readCost(B_n))\right) \quad (3)$$

is the smallest among all cuts of the hierarchy H. Intuitively, the cut is read into the memory once and for each query in Q the remaining operation nodes are brought to the memory as needed. The first term in equation 3 is the cost of reading the bitmaps of the nodes in c from the secondary storage into the memory. Once these bitmaps have been read into the memory, we reuse them for further query processing, i.e. the bitmaps of the cuts need to be read into the memory only once. The second term denotes the cost of reading remaining bitmaps from the secondary storage every time it is needed to execute a query. These remaining bitmaps are also read only once and cached subsequently for further re-use for queries in the workload.

2.3.4 Cut Selection Case 3: Multiple Queries with Memory Constraints

The above formulations do not have any memory availability constraints; i.e., as many bitmaps as needed can be read and cached in memory for the given workload. In general, however, there may be constraints on the amount of data we can cache in memory. Therefore, we next consider scenarios where we have a constraint on the amount of memory that can be used during query processing. Let us assume that we have a memory availability constraint S_{total}. Every bitmap has a size associated to it, S_{B_n}, denoting the memory requirement of the bitmap file of node n in the main memory. Given a query workload Q and S_{total}, *we want to find a (potentially incomplete) cut c that minimizes the following cost*:

$$\left(\sum_{n\in c} readCost(B_n)\right) + \left(\sum_{q\in Q}\sum_{m\in ON_q/c} readCost(B_m)\right) \quad (4)$$

subject to

$$\sum_{n\in c} S_{B_n} \leq S_{total} \quad (5)$$

Note that the bitmaps for c are read into the memory once and for each query in Q the remaining operation nodes are brought to the memory as needed. The major difference from before is that due to the constraint on the size of the nodes that can be maintained in memory, c may be an incomplete cut. Moreover, the operation nodes that are not in the cut cannot be cached in memory for reuse (unless $S_{total} > \sum_{n\in c} S_{B_n}$).

3. CUT SELECTION ALGORITHMS

As described in the previous section, query execution times can be reduced if we are also given the bitmap indices for a subset of the nodes in the domain hierarchy of the column. A key challenge is to select the appropriate subset (or *cut*) of the hierarchy H to minimize the query processing cost, without adding significant memory overhead. In this section, we present algorithms that search for a cut, c, given a query q or a workflow of queries Q. It is important to note that these algorithms do not directly return the operation nodes required to execute q; instead they aim to find a cut, c, such that there exists a set of operation nodes $ON_q \subseteq (c \cup L_H)$ with a small cost. Once a good cut of hierarchy is found, the necessary operation nodes ON_q are identified in post-processing by searching within the cut c.

Algorithm 1 Inclusive Cut Selection Algorithm

```
1: Input: Hierarchy H, Set of internal nodes I_H, Query q
2: Output: Set of nodes c
3: Initialize: Node n = root, c
4: procedure FINDNODEINCLUSIVECUT(n)
5:     Set children = findChildren(n, I_H);
6:     if children is empty then
7:         add n to c;
8:         return nodeInclCost(n, q);
9:     else
10:        costChildren = 0;
11:        for each child m of n do
12:            costChild = findNodeInclusiveCut(m);
13:            if costChild ≠ ∞ then
14:                costChildren = costChildren + costChild;
15:            end if
16:        end for
17:        if costChildren = 0 then
18:            costChildren = ∞;
19:        end if
20:        costCurrNode = nodeInclCost(n, q);
21:        if costCurrNode ≤ costChildren then
22:            remove all descendants of n from c;
23:            add n to c;
24:        end if
25:        return min(costCurrNode, costChildren)
26:    end if
27: end procedure
```

3.1 Case 1: Single Query without Memory Constraints

As described in Section 2.2.2, queries can be processed using inclusive, exclusive, and hybrid strategies. In this subsection, we first provide three algorithms, corresponding to different strategies, for the basic scenario with a single query without memory constraint.

3.1.1 Inclusive Cut Selection (I-CS) Algorithm

The *inclusive cut selection (I-CS)* algorithm associates an *inclusive cost* to all nodes of the hierarchy and selects the cut using these inclusive costs. Given node v of the hierarchy H, let $l(v) = \{m \| (m \in leafDesc(v)) \wedge (G_{q,m} = 1)\}$. Formally, given a query q and a node n on hierarchy H, we define the inclusive cost, $nodeInclCost(n, q)$, of the node in the cut as follows:

$$\begin{cases} \infty & \text{if } \forall_{m\in leafDesc(n)} G_{q,m} = 0 \\ readCost(B_n) & \text{if } \forall_{m\in leafDesc(n)} G_{q,m} = 1 \\ \sum\limits_{\substack{m\in leafDesc(n) \\ \wedge G_{q,m}=1}} readCost(B_m) & \text{otherwise} \end{cases}$$

Note that the inclusive cost is only applicable for the internal nodes of a hierarchy; it is undefined for a leaf node.

In Alg. 1, we present the outline of the proposed algorithm which uses the above definition of inclusive cost to find a cut c that gives the optimal cost to execute a single range query q. Note that since a valid cut does not include any leaf nodes, the algorithm considers only the set of internal nodes, I_H, of hierarchy H.

The *inclusive cut selection algorithm* presented in Alg. 1 is a dynamic programming solution that traverses the nodes in the hierarchy in a bottom-up manner:

- In line 5 of the pseudo-code, the set $children$ is empty for a node on the second-to-last level of the hierarchy H, since the input to the function $findChildren$ is the set of internal nodes I_H. Whenever the set $children$ is empty, we add the current node to the cut c, and return the inclusive cost of the current node.

- The condition on line 13 makes sure that the cost of children of n does not include the cost when a child m has the cost ∞. This will happen when none of the nodes in $leafDesc(m)$ is a range node, i.e. q does not want the contents of m to be included in the result of the query.
- The condition on line 17 will be true if for every child m of n, $nodeInclCost(m) = \infty$. This also means that no node in $leafDesc(n)$ is a range node. In such a case, we want the total cost of all the children of n to be equal to ∞.
- The algorithm then compares the inclusive cost of the parent with the inclusive cost of the set of its children. If the inclusive cost of the parent is cheaper than the combined inclusive cost of its children, then we remove the descendants of n from c and add n to c. Otherwise, we keep the cut as it is, since using the children of n is cheaper than using n.

If the resulting c contains only the root node of the hierarchy, then it means that using the leaves is the cheapest option.

Note that the algorithm is very efficient: each internal node in the hierarchy is considered only once and for each node only its immediate children need to be considered; moreover, the function $nodeInclCost()$, which is called for each node, itself has a bottom-up implementation with $O(1)$ cost per node assuming that node densities for each internal node has been computed ahead of time Consequently, the cost of this algorithm is linear in the size of the hierarchy, H.

3.1.2 Exclusive Cut Selection (E-CS) Algorithm

Above, we considered the inclusive strategy which uses bitwise OR operations among the selected bitmaps to execute the query q. As we see in Section 4.1, this option may be costly when the query ranges are large. Alternatively, we can identify query results using an exclusive strategy: For a given query q, consider a leaf node m such that $G_{q,m} = 0$. That means that this node is not a range node. We call the leaf nodes (like m), which are outside of the query range, the *non-range nodes* and denote them as NS_q. The values of these leaf nodes are part of the actual data that q does not want to be displayed in the result. The exclusive strategy, initially introduced in Section 2.2.2, would first identify the non-range leaf nodes and then use the rest to identify the query results.

Like the inclusive cost, we associate an exclusive cost to all internal nodes of the hierarchy. Consider an internal node n of the hierarchy. If every node in $leafDesc(n)$ is a range node, that means that the q wants the content of n to be included in the result of the query, i.e. $leafDesc(n)$ does not contain any non-range node. In this case, we do not need to remove any node from n, and thus, the exclusive cost of n is the read cost of the node n. Note, that in the same scenario, the inclusive cost of n is also the read cost of n. If, in contrast, none of the leaf descendants of n is a range node, then the query results will not include n and in this case, the *node exclusive cost* of n can be said to be ∞. The main difference is the scenario when only some of $leafDesc(n)$ are non-range nodes. In this case, the exclusive strategy removes the non-range nodes from n, and thus, the exclusive cost of n is the read cost of reading all the non-range nodes under n, in addition to the read cost of n. Based on these, we can formulate the node exclusive cost, $nodeExclCost(n, q)$ as follows:

$$\begin{cases} \infty & \text{if } \forall_{m \in leafDesc(n)} G_{q,m} = 0 \\ readCost(B_n) & \text{if } \forall_{m \in leafDesc(n)} G_{q,m} = 1 \\ readCost(B_n) + \sum_{m \in leafDesc(n) \wedge G_{q,m}=0} readCost(B_m) & \text{otherwise} \end{cases}$$

Given these node exclusive costs (which can again be computed in O(1) time per node using a bottom-up algorithm), an optimal exclusive cut can be find using a linear time algorithm similar to the node inclusive cut algorithm presented in Alg. 1; the main difference being that each internal node in the hierarchy is associated with an exclusive cost, instead of an inclusive cost. In this case, the results would be a cut c such that reading every node in $ON_q \subseteq (c \cup NS_q)$, we can execute the query q optimally using the exclusive strategy. If the output cut c is the root node of the hierarchy, then every node in NS_q has to be removed, i.e. an ANDNOT operation has to be done between the root node and the nodes in NS_q.

3.1.3 Hybrid Cut Selection (H-CS) Algorithm

So far, we have considered inclusive and exclusive strategies independently from each other. However, we could consider both inclusive and exclusive strategies for each node in the hierarchy and associate the better strategy to that node. In other words, we could modify the linear-time, bottom-up algorithm presented in Alg. 1 using the following cost function for each internal node of the hierarchy, H:

$$nodeHybridCost(n, q) = min(\quad nodeInclCost(n, q), \\ nodeExclCost(n, q)).$$

Unlike when searching for the inclusive or exclusive cuts of the hierarchy, during the traversal, we also need to mark each node as an *inclusive-preferred* or *exclusive-preferred* node based on the contributor to the hybrid cost.

Naturally, in this case the resulting cut, c, can be partitioned into two: an *inclusive cut*, c_i (whose nodes are considered in an inclusive way), and an *exclusive cut*, c_e (whose nodes are considered under the exclusive strategy). Those nodes that have a lower inclusive cost are included in c_i, whereas those that have a lower exclusive cost are included in c_e.

- If no $leafDesc(n)$ is in the range, then we call n, an *empty node.* An empty node is not used in any query processing and is ignored.
- If all of $leafDesc(n)$ are in the range, then we call n, a *complete node.* A complete node indicates that all the leaf descendants of the node are needed for query processing. Hence, both the inclusive and the exclusive costs of a complete node are same.
- If only some of the $leafDesc(n)$ are part of the range, then we call n, a *partial node*. Note that the only time n will have potentially different inclusive and exclusive costs is when n is a *partial node*. If a node is a partial node, we find both the inclusive and exclusive costs, and choose the minimum of the two costs. Subsequently, whichever cost is chosen, we label the node accordingly as part of the inclusive or the exclusive cut. This helps us in efficiently finding the operation nodes as described further.

As we mentioned earlier, the algorithms described in this section return a cut c, but not the specific operation nodes that are required to optimally execute the query q. Given a cut c, we need an additional step in order to find the necessary operation nodes. Alg. 2 provides the pseudo-code for finding the operation nodes following execution of the H-CS algorithm. Here, the functions $nodeInclusiveCut(n, q)$ and $nodeExclusiveCut(n, q)$ return the set of operation nodes required to execute the relevant part of

Algorithm 2 Finding the Operation Nodes

```
1: Input: Set of nodes c, Query q
2: Output: Set of operation nodes ON_q
3: Initialize: ON_q
4: procedure FINDOPERATIONNODES(c, q)
5:     for each node n in c do
6:         if n is a complete node then
7:             add n to ON_q;
8:         else if n is a partial node then
9:             inclusiveCost = nodeInclCost(n, q);
10:            exclusiveCost = nodeExclCost(n, q);
11:            if inclusiveCost ≤ exclusiveCost then
12:                add every node from nodeInclusiveCut(n, q) to ON_q;
13:            else
14:                add every node from nodeExclusiveCut(n, q) to ON_q;
15:            end if
16:        end if
17:    end for
18:    return ON_q
19: end procedure
```

Algorithm 3 Hybrid Cut Multiple Query Algorithm

```
1: Input: Hierarchy H, Set of internal nodes I_H, Query Workload Q
2: Output: Set of nodes c
3: Initialize: Node n = root, c
4: procedure FINDHYBRIDCUT(n)
5:     Set children = findChildren(n, I_H);
6:     if children is empty then
7:         add n to c;
8:         return NCNodeCost(n, Q);
9:     else
10:        costChildren = 0;
11:        for each child m of n do
12:            costChildren = costChildren + costChild;
13:        end for
14:        costCurrNode = NCNodeCost(n, Q);
15:        if costCurrNode ≤ costChildren then
16:            remove all descendants of n from c;
17:            add n to c;
18:        end if
19:    end if
20:    return min(costCurrNode, costChildren)
21: end procedure
```

the query q at an internal node n based on inclusive or exclusive strategies, respectively and the algorithm follows the minimal cost strategy to identify the operation nodes for the hybrid execution. We explained our marking strategy earlier in this section. Based on the marking of each node in the cut, we call the respective function to get the corresponding inclusive or exclusive operation nodes. Note that if the cut, c, includes the root of the hierarchy, then either reading the nodes as part of the query range, or removing the non-range nodes from the root is the cheapest option. This decision is again made based on whether the root node was labeled as part of the inclusive or exclusive cut. We do not need to recompute the two individual costs to make that decision.

3.2 Case 2: Multiple Queries without Memory Constraint

In this previous section, we have shown that the simple case where there is a single query to be executed can be handled in linear time in the size of the hierarchy. In general, however, we may be given a set of range queries and need to identify a cut of the hierarchy to help process this set of queries efficiently. In this subsection, we present an algorithm to find a cut for multiple queries without any memory constraints. We consider the more realistic case with memory constraints in the next subsection.

Assume we are given a query workload Q that contains more than one query (each with its corresponding range). Since we do not have memory constraints, if a bitmap node in the hierarchy has been read into the memory, it can also be cached to be reused by other queries, without incurring any further read costs.

Remember that in Section 3.1.3 we have discussed how to find a hybrid cut and the corresponding operation nodes given a single query. Let us first assume that we use the algorithms discussed in Section 3.1.3 to find the hybrid costs and the appropriate labeling for each query in the workload, Q, separately. In order to see how important a particular node n is relative to a particular query workload. Let us consider, *Sub-Operation Nodes*, $SN_{n,q}$, which denote the operation nodes required to execute the part of q (in Q) that is under n. Hence, $SN_{n,q}$ will contain nodes that are in $n \cup leafDesc(n)$. In order to decide which nodes to choose in the set $n \cup leafDesc(n)$ given q, we use the same hybrid logic as explained in Algorithm 2.

We associate to each node, n, in the hierarchy a new cost, called *no constraint node cost* ($NCNodeCost(n, Q)$), defined as the cost to perform the query workload such that (a) first the node is read and cached into the memory and (b) the remaining nodes in each query's corresponding $SN_{n,q}$ are read:

$$NCNodeCost(n, Q) =$$

$$(readCost(B_n)) + \left(\sum_{m \in (\cup_{q \in Q} SN_{n,q})/n} readCost(B_m) \right).$$

Intuitively, this cost tells us how important a particular node, n, is relative to the query workload Q: If there are two nodes, n_a and n_b, such that n_a appears in $SN_{n_a,q}$ for more than one query $q \in Q$ and n_b does not appear in any $SN_{n_b,q}$ for any $q \in Q$, then the $NCNodeCost(n_a, Q)$ will be lower than $NCNodeCost(n_b, Q)$. Consequently, we can say that a node that is included in the $SN_{n,q}$ is more important (caching it would impact more queries) and such important nodes have small $NCNodeCost$ values. We use this as the basis of our algorithm, shown in Alg. 3, to find the relevant hybrid cut given multiple queries. This bottom-up traversing algorithm is similar to the Hybrid Cut Algorithm explained in the previous section. The main difference is that we use the cost $NCNodeCost(n, Q)$ for each node, which is derived using the hybrid logic as explained in the previous section.

3.3 Case 3: Multiple Queries with Memory Constraint

In the previous subsection, we introduced a node cost (based on the cost model as described in 2.3.3.) to capture the importance of a node in a multiple query scenario without a memory constraint. In this section, we relax the assumption of unlimited memory availability and consider the more general situation where we have a memory constraint, limiting how many bitmaps we can keep in memory at a time. More specifically, in this section, we present two algorithms, namely *1-Cut Selection Algorithm* and *k-Cut Selection Algorithm*, that find a cut given a query workload and a memory constraint. Note that, as discussed in Section 2.3.4, due to the memory constraint, the resulting cuts may be incomplete.

Let us consider a set of nodes for each query and each n, called *Constraint Operation Nodes*, denoted by $CON_{n,q}$. Here, $CON_{n,q} \subseteq n \cup L_H$. $CON_{n,q}$ chooses the set of nodes from $n \cup L_H$ that are required to execute q in the cheapest possible manner given n and the set of leaf nodes.

$CON_{n,q}$ consists of two sets of nodes. The first set is the set of nodes that includes n and its leaf descendants. We have to decide which nodes to choose in the set $n \cup leafDesc(n)$ given q. In order to make this decision, we use the same hybrid logic as explained in

Algorithm 4 1-Cut Selection Algorithm

1: **Input:** Hierarchy H, Set of internal nodes I_H, Query Workload Q, $S_{available}$
2: **Output:** Set of nodes c
3: **Initialize:** $S_{available} = S_{total}$
4: **procedure** FINDCUTCONSTRAINT(D_H, $S_{available}$)
5: **while** I_H is not empty OR there exists a node n such that $S_{B_n} \leq S_{available}$ **do**
6: choose node n such that n has the lowest $CNodeCost(n, Q)$ among nodes in I_H & $S_{B_n} \leq S_{available}$;
7: add n to c;
8: remove n from I_H;
9: remove ancestors and descendants of n from I_H;
10: update $S_{available} = S_{available} - S_{B_n}$;
11: **end while**
12: **return** c
13: **end procedure**

Algorithm 2. The second set of nodes, consists of the set of leaf nodes that are not descendants of n , i.e. $L_H \cap leafDesc(n)$. In order to execute q, all the query range nodes in this set have to be read, and hence we include them in $CON_{n,q}$.

As we have done in Case 2 (without memory constraints), we introduce a node cost to capture the importance of each internal node in the hierarchy relative to query workload Q. This cost, called *constrained node cost* ($CNodeCost(n, Q)$), reflects the cost of performing the query in such a way that (a) only nodes with low cost, and that can fit into the memory within the given constraint, are read and cached into the memory and (b) the remaining nodes in each query's $CON_{n,q}$ are read from the secondary storage as needed.

$CNodeCost(n, Q) =$

$$(readCost(B_n)) + \left(\sum_{q \in Q} \sum_{m \in CON_{n,q}/n} readCost(B_m) \right)$$

Intuitively, if more queries can reuse a node for further query processing when the node is cached, the lower the *constrained node cost* of the node is relative to the query workload Q.

3.3.1 1-Cut Selection Algorithm

In Alg. 4, we present the pseudo-code of *1-Cut Selection Algorithm*, for Case 3 with multiple queries in the presence of a memory constraint. Here, $S_{available}$ denotes the amount of memory available for adding nodes to a cut and S_{B_n} denotes the size of the bitmap index of node n on the secondary storage. The first time the algorithm is called, we initialize $S_{available}$ to the memory available for the whole process, i.e., S_{total}; in subsequent calls, the amount is reduced as new bitmaps are added to the cut. Note that

- In line 6, we choose a node that has the lowest node cost and the size of the node is lesser than or equal to the remaining memory availability.
- In line 9, we ensure that the returned cut does not contain any two nodes that are on the same root-to-leaf branch.

The stopping condition of the greedy process is reached when the input set of nodes is empty (i.e. a complete cut is found) or when each of the remaining nodes have sizes larger than $S_{available}$. Note that it is possible that in some cases the optimal subset of nodes required to execute the given query workload may all fit in the available memory. Our algorithm adds nodes until all nodes are seen or no nodes can be added further due to memory constraints. In order to avoid adding nodes that are not going to be used in query processing, we introduce a new node label, *unused*, applied while calculating the $CNodeCost(n, Q)$ indicating that the node as *unused* if the node is not used by any query. This is easy to find out if for every q in Q, $P_{n,q}$ does not include n, then the node is an unused node.

Algorithm 5 k-Cut Selection Algorithm

1: **Input:** Hierarchy H, Set of internal nodes I_H, Query Workload Q, $S_{available}$, $cutList$
2: **Output:** Set of nodes c
3: **Initialize:** $\forall_{c \in cutList} S_{c_i,available} = S_{total}$.
4: **procedure** FINDKCUTCONSTRAINT(H)
5: **while** each node n in I_H is seen OR there exists a node n such that $S_{B_n} \leq S_{c_i,available}$ for $i \leq k$ **do**
6: choose node n such that n has the lowest $CNodeCost(n, Q)$ among nodes in H;
7: mark n as seen;
8: **for** each cut c in $cutList$ **do**
9: **if** $S_{B_n} \leq S_{c_i,available}$ **then**
10: **if** there is no conflict in c for node n **then**
11: **if** n has not been added to any empty cut **then**
12: add n to c;
13: update $S_{c_i,available} = S_{c_i,available} - S_{B_n}$;
14: **end if**
15: **else**
16: copy each node in c to the next available empty cut;
17: replace the conflicting node with node n;
18: **end if**
19: **end if**
20: **end for**
21: Sort the $cutList$ based on the lowest cost for each cut;
22: **end while**
23: **return** the cut c in cutList that has the lowest total cost;
24: **end procedure**

It is important to note that the above algorithm does not necessarily return a cut that has the optimal cost. As we see in Section 4.3, the sub-optimality of the algorithm is most apparent in situations where we have plenty (yet still insufficient amount of) memory and, consequently, the cost-sensitive greedy algorithm over-prunes the solution space (though it still provides cuts that are significantly more efficient than a naïve execution plan). In situations where the memory constraints are tight, however, the algorithm returns very close to optimal or optimal cuts, proving the effectiveness of the cost model and the proposed approach.

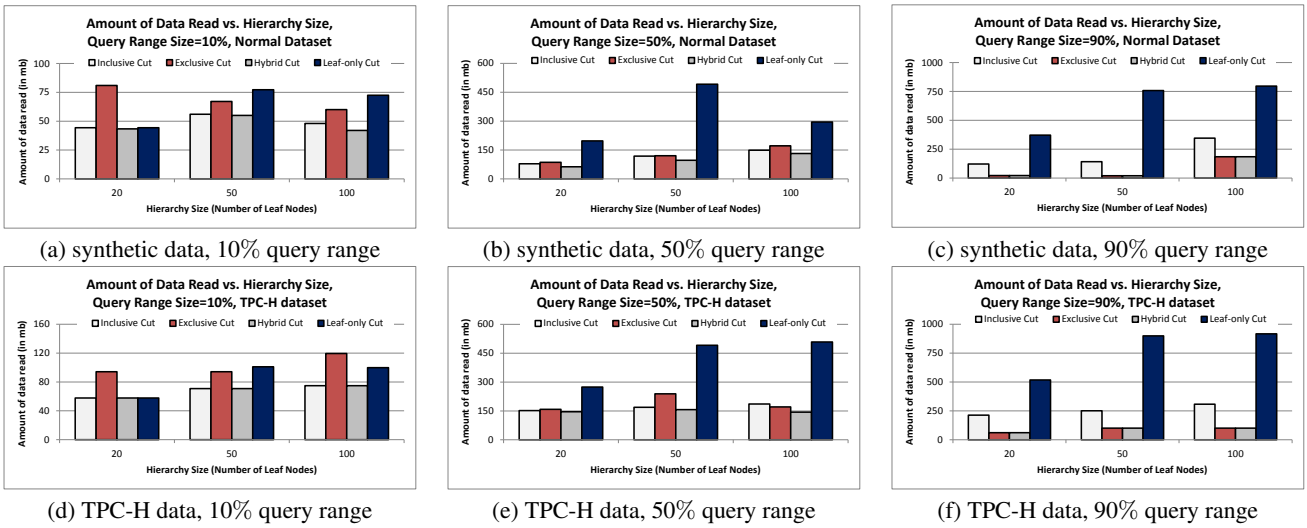
3.3.2 k-Cut Selection Algorithm

In this subsection, we note that the key weakness of the above algorithm is that it considers only a single cut of the hierarchy: When we choose to include a node in the cut, we remove all the ancestors and descendants of the node from further consideration; however, it is possible that a node can have the lowest cost, but two or more of its ancestors or descendants *combined* can lead to a better execution plan. A node n may be chosen before its ancestor m, because cost(n) is lesser than cost(m). But, it is also possible that choosing m could be a better choice than choosing n if m can be used to execute a larger portion of the range nodes of the query.

Therefore, in Alg. 5, we present a variation of the algorithm, called the *k-Cut Selection Algorithm*. In this variation, the algorithm considers k different cuts. When a node, n, is added to a cut, the algorithm does not eliminate its ancestors and descendants from further consideration; instead, it simply does not add these ancestors and descendants to the same cut as n to follow the rules of validity as described in Section 2.3.1. These ancestors and descendants however may be added to the other k-1 cuts.

In Algorithm 5, the i^{th} cut has a corresponding memory requirement, $S_{c_i,available}$.

- In the algorithm, line 11 ensures that a node is not added more than once to an empty cut. This prevents two cuts containing identical nodes.
- Lines 16 and 17 are part of the replacement procedure. Ac-



(a) synthetic data, 10% query range (b) synthetic data, 50% query range (c) synthetic data, 90% query range

(d) TPC-H data, 10% query range (e) TPC-H data, 50% query range (f) TPC-H data, 90% query range

Figure 2: Case 1, single query without memory constraints: effects of varying hierarchy and range sizes on the amount of data read by the three different cut-selection algorithms

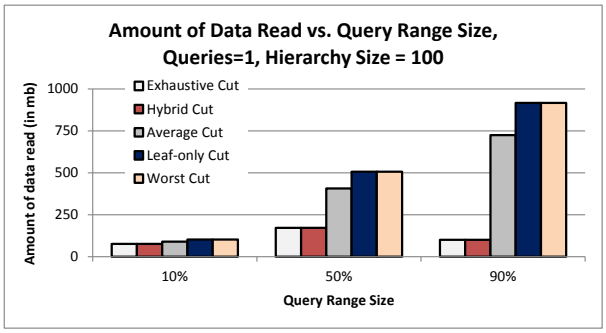


Figure 3: Case 1, single query without memory constraints: comparing the proposed cut algorithm to (exhaustively found) optimal, average, and worst cuts (TPC-H data)

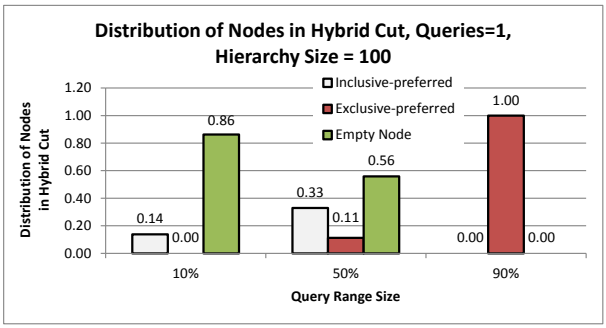


Figure 4: Case 1, single query without memory constraints: percentages of nodes in each strategy (TPC-H data)

cording to Section 2.3.1, a cut cannot have two nodes on the same root-to-leaf branch. Hence, n cannot be added to the existing cut if there is such a *conflict*. In these lines, when we detect a conflict, we add the nodes of a cut to an empty cut and replace the conflicting node with the current node. This lets us construct multiple conflicting cuts that are individually conflict-free. Note that if after replacing the conflicting node with the current node, the size of the cut exceeds the size of available memory, then we ignore this node and the corresponding conflicting cut.

- In Line 21, we sort the $cutList$ in ascending order based on the overall cost of each discovered cut. We do this in order to give more preference to the cuts with a lower cost during the next iteration.

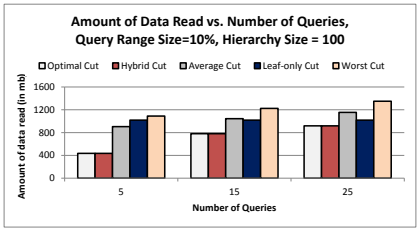
3.3.3 Auto Selection of k

As we see in the next section, in practice it is sufficient to consider fairly small number of cuts to significantly improve the effectiveness of the proposed greedy algorithm (returning very close to optimal cuts), without increasing the cost of the optimization step significantly. However, in cases where it is difficult for the user to set the value of k ahead of the time, we propose a δ *auto-stop* condition: after finding the i'th cut, we evaluate if $cost_{i-1} - cost_i < \delta$, for a user provided per-iteration cost gain value, δ. The algorithm auto-stops when the condition is satisfied (i.e., when the cost gain of the iteration drops below the predetermined gain). In Section 4.3, the auto-stop condition is effective, even when we simply set $\delta = 0$; i.e., we stop when the cost of the new cut has the same cost as the previous cut (note that, for any two integers $l, m > 1$, and $l > m$, the cost of l-greedy cut will always be equal to or lesser than the cost of m-greedy cut; this is because whatever cut that is returned by the m-greedy cut algorithm will always be enumerated and considered by the l-greedy cut algorithm).

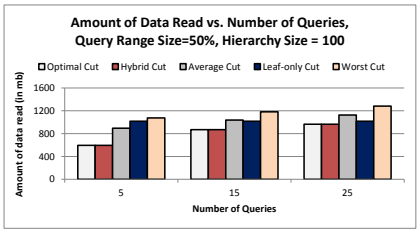
4. EVALUATIONS

In order to evaluate the cut-selection algorithms presented in this paper, we considered two datasets: (a) a synthetically generated dataset (with normal value distribution) and (b) the TPC-H dataset [22], each with 150 million records. In particular, in the TPC-H dataset, we focused on the *account balance* attribute whose values demonstrate a near-uniform distribution, with spikes in the occurrences for some values.

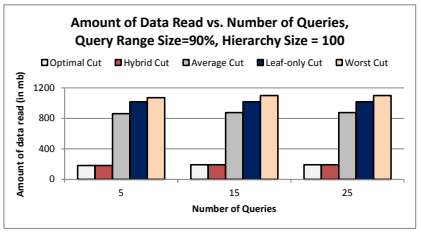
In this section, we have two main evaluation criteria: (1) query execution IO cost and (2) optimization time. We compared the



(a) 10% query range

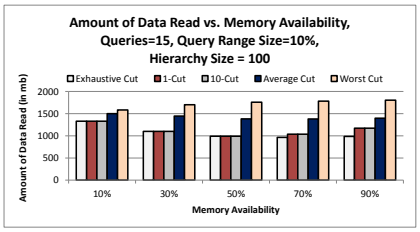


(b) 50% query range

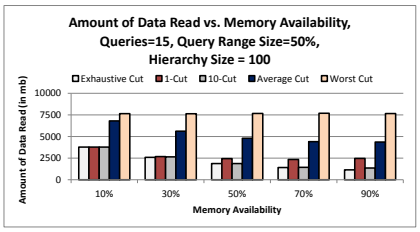


(c) 90% query range

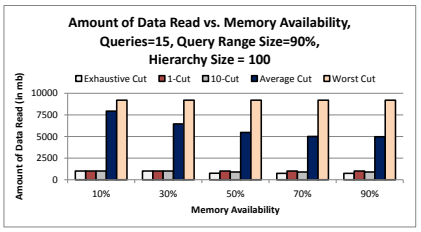
Figure 5: Case 2, multiple queries without memory constraints (TPC-H data)



(a) 10% query range



(b) 50% query range



(c) 90% query range

Figure 6: Case 3, multiple queries with varying memory availability (TPC-H data)

results of our cut-selection algorithms against (a) leaf-only query execution, (b) random cut-selection, and (c) exhaustive cut-search strategies.

For both of the above data sets, we considered (balanced) attribute hierarchies of different depth and internal-node fanout: these were generated for different numbers of leaf nodes and maximum possible fanouts of the internal nodes of the hierarchy. Since finding the optimal cut using an exhaustive strategy *for comparison purposes* is prohibitively expensive, we initially considered small hierarchies, with 20, 50, and 100 leaf nodes and heights of 4, 5, and 4 respectively (the root of the hierarchy being considered at height 1).

In Section 4.4, we consider hierarchies of larger sizes and higher number of queries to study the scalability of the cut-selection algorithms against the hierarchy size.

Bitmap indices were generated for the nodes of these hierarchies using the Java library, WAH bitset [23] as explained in [10]. The parameters of the read cost model presented in Section 2.2.1, and shown in Figure 1 were computed based on these bitmap indices.

We have also created query workloads with different target range sizes. For example, for a hierarchy of 100 leaf nodes, 10% query range size indicates that each range query covers 10 consecutive leaf nodes.

We ran the experiments on a quad-core Intel®Core™i5-2400 CPU @ 3.10GHz machine with 8.00GB RAM. All codes are implemented and run using Java v1.7.

4.1 Case 1: Single Query without Memory Constraints

We first evaluate the cut-selection algorithm for the single query without memory constraints scenario. All reported costs are averages of the costs for 10 different runs.

Figures 2(a) through (f) compares the three different cut-selection algorithms (I-CS, E-CS, and H-CS) presented in Section 3.1 for different data sets and varying hierarchy and range query sizes. As we see in these charts, the inclusive strategy is efficient when the query ranges are small; this is consistent with the observation in [5]. The exclusive strategy, however, is more efficient than the inclusive strategy when the query ranges are larger. Most importantly, in all cases, the hybrid strategy (H-CS) returns the best cuts.

In Figure 3, we compare the hybrid (H-CS) strategy against (exhaustively found) optimal and average cuts. The figure also shows the performance of the worst cut. As expected, the H-CS strategy returns optimal cuts. On the average, randomly selecting a cut performs quite poorly (almost as bad as selecting the worst possible cut), especially as the query range sizes increase. This highlights the importance of utilizing an effective (hybrid) cut-selection algorithm for answering queries.

In Figure 4, we show the percentages of nodes that are labeled inclusive-preferred or exclusive-preferred in a hybrid cut, as explained in Section 3.1.3, for different query ranges. As defined in Section 3.1.3, empty nodes are nodes that are not used in query processing. When the query range size is small, most of the query processing can be done using the leaf nodes. Hence, we see in the figure that most of the nodes in the cut are empty nodes. As expected, when the query range is small, the inclusive strategy dominates and when the range is large, the exclusive strategy dominates. For ranges that are neither small nor large, the hybrid algorithm leverages a mix of inclusive and exclusive strategies.

4.2 Case 2: Multiple Queries without Memory Constraints

In this section, we evaluate the hybrid cut selection algorithm (Alg. 3) for query workloads with multiple queries. For our evaluations, we considered query workloads of different sizes (and with different ranges). All reported costs are averages of the costs for 10 different runs.

Figure 5 shows the impact of using the proposed hybrid cut selection algorithm for different numbers of queries. As we see in this figure, as expected, the hybrid cut selection algorithm returns the optimal cut. The impact of the proposed cut selection algorithm is especially strong when the query includes large ranges as when there are large overlaps among the queries, the query evaluation algorithm has more opportunities for reusing cached nodes, and the proposed hybrid cut strategy is able to leverage these opportunities

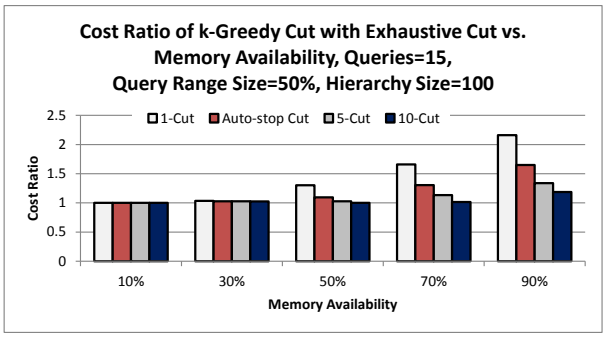


Figure 7: Case 3, multiple queries with varying memory availability (TPC-H data): impact of different k

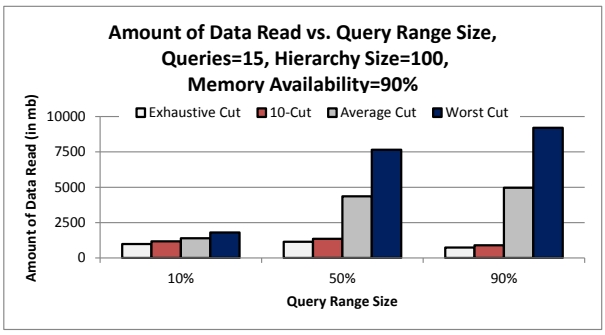


Figure 8: Case 3, effect of different query range sizes (TPC-H data, 90% memory availability)

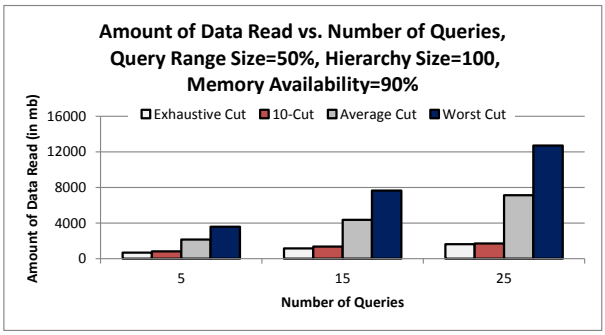


Figure 9: Case 3, effect of different number of queries (TPC-H data, 90% memory availability)

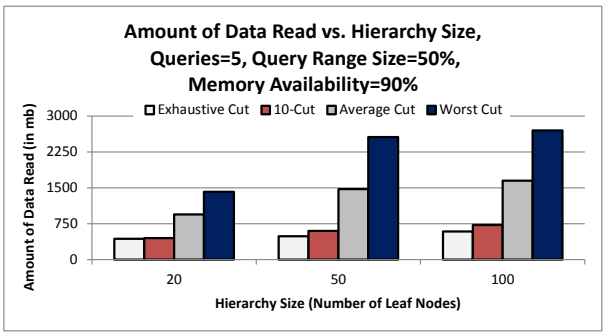


Figure 10: Case 3; effect of different hierarchy sizes (TPC-H data, 90% memory availability)

most effectively.

4.3 Case 3: Multiple Queries under Memory Constraints

In this section, we evaluate the effectiveness of the proposed k-hybrid cut algorithm (Alg. 5, described in Section 3.3.2), for multiple queries, but under memory constraints. We report the memory availability in terms of the percentage of the memory needed to store the bitmap indices corresponding to the maximum cut of the given hierarchy. The presented results are averages of 10 different runs.

Once again, we compare the proposed cut selection algorithm against solutions found through exhaustive enumeration, average solutions representing randomly selected cuts, and also the worst solution. Remember, that under memory limitations, we need to consider also the incomplete cuts of the input hierarchies.

Note that the number of incomplete cuts that an exhaustive algorithm would need to consider grows very fast:

Num. of leaves	Height	Incomplete cuts
20	4	154
50	5	296,381
100	4	1,185,922

However, since the number of incomplete cuts grow even faster than the number of complete cuts, enumerating all incomplete cuts for the exhaustive algorithm (which we use to locate the optimal cut for comparison purposes), becomes prohibitive beyond hierarchies with 100 leaf nodes.

Figure 6 shows that, in this case, the proposed hybrid cut selection algorithms are not optimal; however, they return cuts that are very close to optimal. In fact, especially when the memory availability is very restricted (which is the expected situation in most realistic deployments), even the 1-Cut algorithm is able to return optimal or very close to optimal answers. As the available memory increases, the optimal cost decreases as there are more caching opportunities, but 1-Cut strategy may not be able to leverage this effectively, especially for larger query ranges. However, we see that the multi-cut strategy (10-Cut in this figure) performs quite close to optimal. Figure 7, which plots the ratio of the cost of the solutions found by the multi-cut strategy (for different values of k) to the cost of the optimal cut found through an exhaustive search, confirms this observation: note the figure also shows that the *auto-stop* strategy described in Section 3.3.3 is effective in reducing the cost, without having to fix the value k ahead of time.

Figures 8 through 10 further confirm that the proposed multi-cut strategy is robust against changes in the size of the query ranges, number of queries, and hierarchy sizes.

4.4 Cut-Selection Time

Up to now, we considered query processing cost using cuts. We now focus on the time needed to select cuts for hierarchies of different sizes. In Figures 11 and 12, we see the cut selection time as a function of the size of the hierarchy (number of leaf nodes; i.e., the size of the domain) and the number of queries, respectively. Please note that, in these figures, we do not compare our algorithm with exhaustively found cuts, and hence are able to consider larger hierarchy sizes and higher number of queries. The figures confirm that the time taken to find the cut increases linearly with size of the attribute domain and the number of queries.

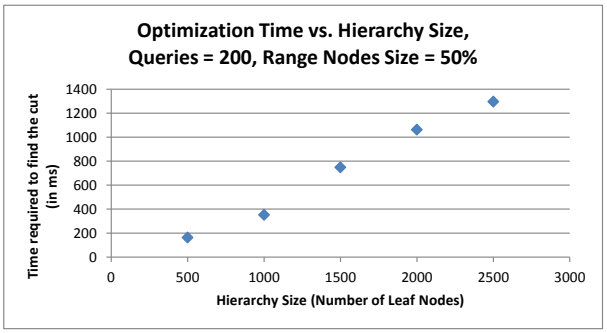


Figure 11: Effect of different hierarchy sizes on time taken to find the hybrid cut

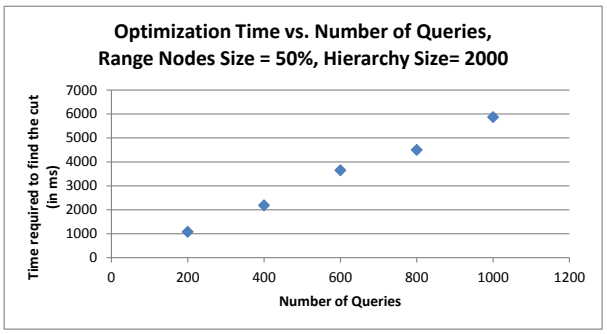


Figure 12: Effect of different number of queries on time taken to find the hybrid cut

5. CONCLUSION

Column-stores use compressed bitmap-indices for answering queries over data columns. When the data domain is hierarchical, organizing the bitmap indices hierarchically can help more efficiently answer queries over different sub-ranges of the attribute domain. In this paper, we showed that existing *inclusive* strategies for leveraging hierarchically organized bitmap indices can be sub-optimal in terms of their IO costs unless the query ranges are small. We also showed that an *exclusive (cut-selection) strategy* provides gains when the query ranges are large and that and that a *hybrid (cut-selection) strategy* can provide best solutions, improving over both strategies even when the ranges of interest are relatively small. In this paper, we also presented algorithms for implementing the *hybrid strategy* efficiently for a single query or a workload of multiple queries, in scenarios with and without memory limitations. In particular, we showed that when the memory is constrained, selecting the right subset of bitmap indices becomes difficult; but, we also showed that, even in this case, there exists efficient cut-selection strategies that return close to optimal results, especially in situations where the memory limitations are very strict. Experiment results confirmed that the *cut-selection* algorithms presented in this paper are efficient, scalable, and highly-effective.

6. REFERENCES

[1] C. Ho, R. Agrawal, N. Megiddo, and R. Srikant. Range queries in olap data cubes. SIGMOD '97, pages 73–88, 1997.

[2] D. Abadi, S. Madden, and M. Ferreira. Integrating compression and execution in column-oriented database systems. SIGMOD '06, pages 671–682, 2006.

[3] Oracle database 10g, 2013.

[4] Luciddb - home, 2013.

[5] J. Chmiel, T. Morzy, and R. Wrembel. Time-hobi: indexing dimension hierarchies by means of hierarchically organized bitmaps. DOLAP '10, 2010.

[6] S. Hong, B. Song, and S. Lee. Efficient execution of range-aggregate queries in data warehouse environments. In *Conceptual Modeling ER'01*. 2001.

[7] Y. Feng and A. Makinouchi. Ag-tree: a novel structure for range queries in data warehouse environments. DASFAA'06, pages 498–512, 2006.

[8] S. Muller and H. Plattner. Aggregates caching in columnar in-memory databases. VLDB '13, 2013.

[9] T. Lauer, D. Mai, and P. Hagedorn. Efficient range-sum queries along dimensional hierarchies in data cubes. DBKDA '09, pages 7–12, 2009.

[10] K. Wu, E. Otoo, and A. Shoshani. On the performance of bitmap indices for high cardinality attributes. VLDB '04, pages 24–35, 2004.

[11] D. Rotem, K. Stockinger, and K. Wu. Optimizing candidate check costs for bitmap indices. CIKM '05, 2005.

[12] O. Kaser, D. Lemire, and K. Aouiche. Histogram-aware sorting for enhanced word-aligned compression in bitmap indexes. DOLAP '08, 2008.

[13] K. Wu, K. Stockinger, and A. Shoshani. Breaking the curse of cardinality on bitmap indexes. In *Scientific and Statistical Database Management*, pages 348–365. 2008.

[14] R.R. Sinha, S. Mitra, and M. Winslett. Bitmap indexes for large scientific data sets: a case study. 2006.

[15] R. Sinha and M. Winslett. Multi-resolution bitmap indexes for scientific data. *ACM Trans. Database Syst.*, August 2007.

[16] M. Morzy, T. Morzy, A. Nanopoulos, and Y. Manolopoulos. Hierarchical bitmap index: An efficient and scalable indexing technique for set-valued attributes. In *Advances in Databases and Information Systems*, pages 236–252. 2003.

[17] M. Zaker, S. Phon-amnuaisuk, and S. Haw. An adequate design for large data warehouse systems: Bitmap index versus b-tree index, 2008.

[18] J. Chmiel, T. Morzy, and R. Wrembel. Hobi: Hierarchically organized bitmap index for indexing dimensional data. In *DaWaK*, pages 87–98, 2009.

[19] L. Bellatreche, R. Missaoui, H. Necir, and H. Drias. Selection and pruning algorithms for bitmap index selection problem using data mining. DaWaK'07, pages 221–230, 2007.

[20] F. Deliège and T. Pedersen. Position list word aligned hybrid: optimizing space and performance for compressed bitmaps. EDBT '10, pages 228–239, 2010.

[21] K. Wu, E. J. Otoo, and A. Shoshani. An efficient compression scheme for bitmap indices. Technical report, ACM Transactions on Database Systems, 2004.

[22] Transaction Processing Performance Council. Tpc-h benchmark specification, 2013.

[23] Compressedbitset - wah compressed bitset for java, November 2007.