Techniques for Speeding up Range-Max Queries in OLAP Data Cubes

Ching-Tien Ho *Rakesh Agrawal* *Nimrod Megiddo* *Jyh-Jong Tsay**

IBM Almaden Research Center
650 Harry Road, San Jose, CA 95120
{ho,ragrawal,megiddo}@almaden.ibm.com

Abstract

A range-max query obtains the maximum over all selected cells of a data cube where the selection is specified by providing ranges of values for numeric dimensions. Our general approach to speeding up range-max queries is to precompute and store certain key information of the data cube. In [HAMS97], we gave a tree algorithm based on precomputed max over balanced hierarchical tree structures; a branch-and-bound-[Mit70]like procedure was used to prune unnecessary search.

In this paper, we propose three orthogonal techniques with the objective of improving the average response time of the range-max queries. First, rather than keeping only the index of the largest value at each internal node of the tree, we keep the indices of the t largest values with each internal node and use them to decrease the probability of scanning lower level nodes. Second, we further partition each sibling set of internal nodes into smaller groups and sort the precomputed indices within each group according to their indexed values. This speeds up the scanning of internal nodes at the same level and covered by the query region without increasing extra storage overhead. Third, we augment the tree with a precomputed reference array for each level of the tree (except for the leaf level). Elements of a reference array contain references to the next larger value, which are used to speed up the search. We compare our three algorithms with the previous algorithm both analytically and empirically. Based on these comparisons, we then propose and implement a hybrid algorithm, combining the advantages of these orthogonal techniques, that improves the empirically measured range-max query time by as much as 100%. We also give algorithms for incrementally updating the precomputed structures.

1 Introduction

An OLAP data cube [OLA96] can be viewed as a d-dimensional matrix. The dimensions of the matrix correspond to functional attributes and the cells contain the values of the measure attributes for the corresponding combination of functional attributes. For instance, in a 4-dimensional insurance data cube, the dimensions of the matrix may correspond to the functional attributes of year, age,

*On leave from Department of Computer Science and Information Engineering, National Chung Cheng University, Chiayi 621, Taiwan, ROC; Email: tsay@cs.ccu.edu.tw.

state, and insurance type; a cell may contain aggregated value of the revenue in a particular year from customers of a certain age who live in a particular state and have bought insurance policies of a certain type.

A class of queries over OLAP data cubes, called *range queries*, was studied in [HAMS97]. These queries compute aggregation of values from selected cells of a data cube, where the selection is specified by constraining the contiguous range of interest in the domains of some of the functional attributes. Two types of aggregation functions were considered: *max* and *sum*; the corresponding queries were called *range-max* and *range-sum* queries, respectively. Together, these two types of queries cover all aggregate functions supported by SQL. The range-max techniques apply to *min* whereas the range-sum techniques apply to *average* and *count* aggregate functions.

Range queries are frequent with respect to numeric attributes with natural semantics in ordering, such as age, time, salary, etc. For instance, with the insurance data cube mentioned earlier, a range-max query may obtain the maximum revenue from customers in the age group 37 to 52, in a year from 1988 to 1996, in all of U.S., and having auto insurance. In an interactive exploration of data cube, which is the predominant OLAP application area, it is imperative to have a system with fast response time.

In [HAMS97], a tree-based algorithm was used to compute *range-max queries*. This algorithm is based on precomputed max over balanced hierarchical tree structures. A branch-and-bound[Mit70]-like procedure is also used to speed up the search.

Contributions In this paper, we propose three orthogonal techniques for improving the overall response time of the basic tree algorithm for range-max queries. First, rather than keeping only the index of the largest value at each internal node of the tree, we keep the indices of the t largest values with each internal node and use them to further speed up the processing. Second, we further partition each sibling set of internal nodes into smaller groups and sort the precomputed indices within each group according to their indexed values. Without additional storage overhead, this speeds up the scanning of internal nodes at the same level and covered by the query region, Third, we augment the tree with a precomputed reference array for each non-leaf level of the tree. Elements of a reference array contain references to the next larger value, which are used to speed up the search. We present various worst-case and average-case analyses of algorithms based on these new techniques, and confirm these analyses with experimental results. While the first technique yields a comparable algorithm, the algorithms based on the two latter techniques improve the response time over the previous algorithm for most range sizes. Based on these analytical and empirical results, we propose and implement a hybrid algorithm, combining the advantages of the two latter orthogonal techniques, that significantly improves the range-max query time by as much as a factor of two in measured time or six in measured array reference count. Finally, we give algorithms for incrementally updating the precomputed structures.

Related work There is extensive literature in the field of computational geometry on algorithms for handling various types of range queries (see, e.g., [BF79] [Ben80] [CR89] [Cha90] [Meh84] [Vai85] [WL85] [Yao85]). Most of the results share the following properties: First, the space overhead is mostly *non-linear* in m (e.g. $O(m \log^{d-1} m)$), where m is the number of data points. Second, the index domain of each dimension is assumed to be *unbounded*. Third, mostly the *worst-case* space and time trade-offs are considered. We, on the other hand, consider a space overhead which is linear in m. We assume the index domain of each dimension is bounded and we aim at minimizing the *average-case* time complexity. There are also pragmatic differences for typical "data cubes" arising out of the computational geometry domain versus the OLAP domain. A canonical sparsity of the OLAP data cube is about 20% [Col96] and dense sub-clusters typically exist, while the computational geometry data cubes can be much sparser even after placing upper bounds on each index domain.

Besides range-max, techniques for computing range-sum were also proposed in [HAMS97]. These techniques exploit the following special property of the sum operator for their efficiency: for the binary sum operation $\oplus$, there exists an *inverse* binary minus operation $\ominus$ such that $a \oplus b \ominus b = a$, for any a and b in the domain. Since there is no inverse $\ominus$ operation for max that satisfies this property, the algorithms for range-sum in [HAMS97] do not apply to the range-max problem. It is possible to adopt the tree structure used for range-max also for range-sum queries, as a range-sum query may be answered by traversing the tree and adding or subtracting the values at various tree nodes that collectively define the query region. However, unlike range-max, the branch-and-bound optimization cannot be applied to range-sum queries and it was shown in [HAMS97] that the specialized techniques based on prefix-sums perform better for range-sum queries. This observation continues to hold for the techniques proposed in this paper.

More generally, there has been considerable research in database community related to OLAP data cubes. This includes work on modeling [GBLP96] [AGS97] and developing algorithms for computing the data cube [AAD+96], for deciding what subset of a data cube to pre-compute [HRU96] [GHRU97], for estimating the size of multidimensional aggregates [SDNR96], and for indexing pre-computed summaries [SR96] [JS96]. Related work also includes work done in the context of statistical databases [CM89] on indexing pre-computed aggregates [STL89] and incrementally maintaining them [Mic92]. Also relevant is the work on maintenance of materialized views [Lom95] and processing of aggregation queries [CS94] [GHQ95] [YL95].

Organization of the paper The remainder of the paper is organized as follows. In Section 2, we formally define the range-max problem, review the basic tree algorithm as presented in [HAMS97], and present a more detailed analysis of the basic tree algorithm. Our analysis shows that the number of leaf nodes accessed decreases exponentially when the range size increases and gives a new performance bound for small range sizes. In Section 3, we present the *fat-nodes* algorithm,

which is an extension of the basic tree algorithm so that each internal node x will store t indices to the top-t values in the subtree rooted at x. In Section 4, we give the *sorted-nodes* algorithm, an extension of the basic tree algorithm so that adjacent sibling nodes are grouped and sorted in order to speed up the scanning of many consecutive internal nodes. In Section 5, we present the *jump-nodes* algorithm by augmenting the internal nodes of the tree with *reference arrays* to improve the performance. In Section 6, we compare experimental results and the analytical results of the four algorithms. Then, we propose a hybrid algorithm combining different techniques. In Section 7, we give incremental algorithms for updating related precomputed structures. For clarity, we first focus our discussion on one-dimensional trees up to Section 8 and discuss the extension to d-dimensional trees in Section 8. Finally, Section 9 concludes the paper. Proofs of the theorems are given in Appendix.

We only consider dense data cubes in this paper. The sparse data cubes can be handled using an approach similar to one presented in [HAMS97]. The basic idea is to decompose the data cube into dense and non-dense regions and apply the proposed algorithms to the dense clusters. Another related issue is that of selecting the subsets of dimensions for which the precomputed information is stored. A similar problem was considered in [HRU96] [GHRU97]. The decision procedure proposed in [HAMS97] can be directly used for this purpose.

2 Background

2.1 The Model

We are interested in finding Max value and its location in a d-dimensional rectangular region of a d-dimensional array A of order $n_1 \times \cdots \times n_d$. We assume that all arrays have a starting index 0 for each dimension. We will use N to denote the domain (index space) of A, i.e., $N = (0 : n_1 - 1, \ldots, 0 : n_d - 1)$. We denote by $D = \{1, 2, \ldots, d\}$ the set of dimensions.

The *range-max* query problem [HAMS97] can be formulated as getting the Max_index of a region of A defined as follows:

$$Max\_index(\ell_1 : h_1, \ldots, \ell_d : h_d) = (x_1, \ldots, x_d) \text{ where } (\forall i \in D)(\ell_i \leq x_i \leq h_i)$$

$$\text{and } A[x_1, \ldots, x_d] = max\{A[y_1, \ldots, y_d] \mid (\forall i \in D)(\ell_i \leq y_i \leq h_i)\ \}.$$

When $d = 1$, we usually drop the subscript 1 of n, ℓ and h. Unless stated otherwise, we denote by Q the region (or range) of the input query. Thus, when $d = 1$, $Q = (\ell : h)$ consists of all i's such that $\ell \leq i \leq h$.

There may be more than one index with the same maximum value in a specified region. In such case, we assume that the algorithm arbitrarily returns one of the indices with the maximum value in the region.

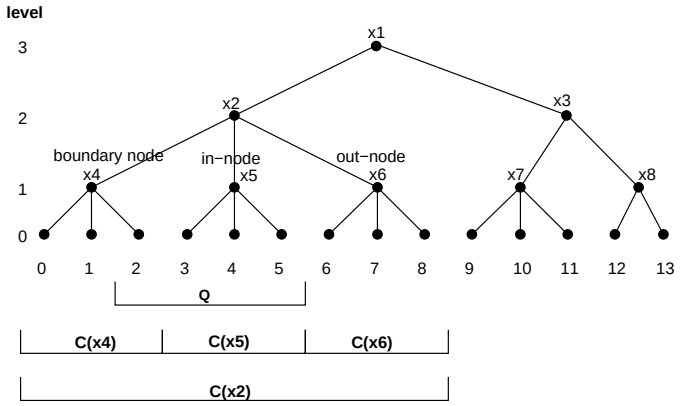

Figure 1: Searching a b-ary tree with $n = 14$ and $b = 3$.

The *range-min* query problem can be similarly formulated as getting the Min_index of a region of A. We only consider the Max_index operation in this paper. The algorithms for the Min_index operation can be similarly derived.

For convenience, we will use Max as a function that takes a region as an argument and returns the maximum value in the region. We use max to represent the traditional function that returns the maximum value of a set or of all input arguments. Throughout the paper, we use the number of tree nodes touched (or the number of indices accessed when there is more than one index or value stored at each tree node) as a measure for the analytical time complexity.

2.2 The Basic Tree Algorithm

We now review the basic tree algorithm for range-max queries as proposed in [HAMS97]. The data structure for storing the precomputed information can be viewed as a generalized quad-tree [Sam89]. Each non-leaf node x "covers" a d-dimensional region (a hyper-cube), denoted $C(x)$, containing all the leaf nodes of the subtree rooted at x. For example, in Figure 1 the region covered by x_2 is $C(x_2) = (0 : 8)$. The index of the maximum value in region $C(x)$ is precomputed and stored at node x. The region covered by a non-leaf node x is partitioned into up to b^d disjoint regions, each covered by one of its children. We thus have a balanced tree structure with roughly the same fanout for each node. The algorithm uses a branch-and-bound[Mit70]-like procedure to speed up the finding of max in a query region using this data structure. For clarity, we only review the one-dimensional case. The construction of the tree and the search algorithm for the d-dimensional case are similar. See [HAMS97] for details.

2.2.1 Constructing the Tree

The leaves of a tree are called *level-0 nodes*. A node is said to be at level $i + 1$ if the maximum level of its children is i. Each non-leaf node, except possibly for one node per level, has precisely b

children. The number b is called the *fanout* of the tree.

The entries of array A (of size n) are stored as the leaves. Then, a b-ary tree is built in a bottom-up manner and from left to right within each level as follows. Partition A into disjoint ranges, each consisting of b entries, except possibly the last one. For each such range, add a parent node to the b nodes in the range, compute the Max_index of the range, and keep it at the parent node. The last parent node may have less than b children. Recursively apply the same procedure to the $\lceil n/b \rceil$ new parent nodes by viewing them as an array A of size $\lceil n/b \rceil$, until there is only one parent node at the same level which will be the root of the tree. Figure 1 shows an example of $n = 14$ and $b = 3$. Clearly, the root of the tree is at level $\lceil \log_b n \rceil$.

2.2.2 The Search Algorithm

Notation Given a non-leaf node x and a region Q, categorize each (immediate) child of x, say y, as one of the three types: (1) an *in-node*[1], if $C(y) \subseteq Q$; (2) an *out-node*, if $C(y) \cap Q = \phi$; and (3) a *boundary* node, otherwise (that is, if $C(y) \cap Q \neq \phi$ and $C(y) \not\subseteq Q$). Thus, the immediate children are partitioned into (up to) three sets: (1) $I(x, Q)$ contains all the in-nodes; (2) $O(x, Q)$ contains all the out-nodes; and (3) $B(x, Q)$ contains all the boundary nodes. As an example in Figure 1, given a region $Q = (2 : 5)$, then $I(x_2, Q) = \{x_5\}, O(x_2, Q) = \{x_6\}$ and $B(x_2, Q) = \{x_4\}$. Note that in the one-dimensional case, for a given x and Q, there are at most two boundary nodes, while there can be any number (up to b) of in-nodes and out-nodes.

The set of boundary nodes $B(x, Q)$ is further partitioned into two subsets: (1) $B_{in}(x, Q) = \{y | y \in B(x, Q), Max\_index(C(y)) \in Q\}$ and (2) $B_{out}(x, Q) = B(x, Q) - B_{in}(x, Q)$. Use Figure 1 as an example again. Assume $Q = (2 : 5)$. If $Max\_index(C(x_4)) = 2$, then $B_{in}(x_2, Q) = \{x_4\}$. If $Max\_index(C(x_4)) = 0$ or 1, then $B_{out}(x_2, Q) = \{x_4\}$. (Note the label on a leaf node refers to the index, not the value, of array A.)

Algorithm The algorithm for finding *Max_index* of a query region $Q = (\ell : h)$ first finds the lowest-level node x such that $C(x)$, the region covered by x, contains the query region Q. This can be done simply by index calculation without searching the tree because b is fixed. (See [HAMS97].) We treat this as one step in our complexity analysis later. If the precomputed index of the maximum value in $C(x)$ lies in Q, we are done. Otherwise, the algorithm calls a recursive function **get_max_index** given below with arguments x, Q, and ℓ to compute the answer.

Function get_max_index $(x, Q, current\_max\_index)$

(1) **for all** $y \in I(x, Q) \cup B_{in}(x, Q)$ **do**
(2) **if** $(Max(C(y)) > A[current\_max\_index])$

[1] The term *internal node* still refers to the standard meaning as a non-leaf and non-root tree node.

(3) $\quad current\_max\_index = Max\_index(C(y));$

(4) **for all** $z \in B_{out}(x, Q)$ **do**

(5) $\quad$ **if** $(Max(C(z)) > A[current\_max\_index])$

(6) $\quad\quad current\_max\_index =$ **get_max_index**$(z, Q \cap C(z), current\_max\_index);$

(7) **return** $(current\_max\_index);$

The recursive function **get_max_index**$(x, Q, current\_max\_index)$ takes as inputs the root of a subtree containing Q, the region Q, and an index to the maximum value currently known ($current\_max\_index$). Note that $current\_max\_index$ was arbitrarily set to ℓ before calling **get_max_index** the first time. In lines (1) to (3), the Max_index of all regions $C(y)$ for all $y \in I(x, Q) \cup B_{in}(x, Q)$ is found and $current\_max\_index$ is updated whenever necessary. Recall that any node $y \in B_{in}(x, Q)$ has the property that $Max\_index(C(y)) \in Q$ even though $C(y) \not\subseteq Q$. Thus, in lines (4) to (6), the Max_index of all regions $C(z)$ is found for all $z \in B_{out}(x, Q)$ by recursively calling the same function. The condition at line (5) improves running time using a branch-and-bound [Mit70] idea. If a value indexed by a precomputed $Max\_index(C(z))$ is already less than or equal to $A[current\_max\_index]$, then there is no need to find $Max\_index(C(z) \cap Q)$ (because it is still less than or equal to $A[current\_max\_index]$ and would not affect the result).

Complexity The worst-case time complexity of the basic tree search algorithm was shown in [HAMS97] to be $O(b \log_b r)$ for a range of size r, as opposed to $O(b \log_b n)$. There is one caveat here. For the complexity to be valid, we actually have to find the lowest-level node that covers the input region $(Q \cap C(z))$ again after the recursion. Since we aim at minimizing the average running time, we ignore this step in the implementation.

The average-case complexity, on the other hand, was shown as bounded from above by $b+7+1/b$, much less than the worst-case bound and independent of the range size r. Note that the average-case complexity here (and later for subsequent algorithms) refers to the average over random data distribution (i.e., equal probability out of $n!$ permutation of ranks), but maximum over all possible range sizes and locations.

2.3 Further Analysis

The analysis in [HAMS97] gives an average-case upper bound for the total number of elements accessed in both internal and leaf nodes. Notice that accessing elements in internal nodes is in general more expensive than accessing elements in leaf nodes, since the internal nodes keep only indices to elements. To better understand the behavior of the search algorithm, it is necessary to count the number of accessed elements in internal nodes and leaf nodes separately.

In this section, we present an analysis that implies that the number of accessed elements in the leaf nodes decreases exponentially when the range size increases. This implies that when range

size is large, most of the accessed elements are in internal nodes. Since when the range size is small ($\ll b^2$), most of the elements accessed will be in leaf nodes. Our new analysis gives a new performance bound for small-sized ranges. The analysis is justified by the experiments presented in this paper. In addition, the analysis will be used in deriving the performance bounds of the algorithms proposed in this paper, in which we count the number of accesses to internal nodes and leaf nodes separately.

Let $L(b,h)$ denote the average number of elements in the leaf nodes accessed by the search algorithm to locate the maximum for a range of the form $[0,r]$, with $0 \leq r \leq h$. We assume every r is of equal probability.

Lemma 1 *$L(b,h) \leq \frac{b}{6}(\frac{1+\ln b}{b})^{k-1}$ where $k = \lceil \log_b h \rceil - 1$.*

Proof: Let $L(b,h)$ denote the average number of elements in the leaf nodes accessed by the search algorithm to locate the maximum for a range of the form $[0,r-1]$, where $1 \leq r \leq h$ is the range size, and b is the fanout of the tree. We assume every i is of equal probability. We first show that $L(b,h) \leq \frac{b}{6}(\frac{1+\ln b}{b})^{k-1}$ where $k = \lceil \log_b h \rceil - 1$.

We give a sketch of a proof by induction on $k = \lceil \log_b h \rceil - 1$. When $k = 0$, $L(b,h) = \frac{1}{b}\sum_{r=1}^{b} \frac{r(b-r)}{b}$. (When the range size is r starting node 0, the number of accessed leaf nodes is $r * \frac{b-r}{b}$, where $\frac{b-r}{b}$ is the probability that the scanning of r leaf nodes is required. The base case is simply an average of all b combinations of $r = 1$ to b.) By integration, we can show that $L(b,h) \leq \frac{b}{6}$.

Now assume the lemma holds for any $k \leq k'$, $k' \geq 0$. We will show the lemma holds for $k = k' + 1$. Assume $b^{k'+1} < h \leq b^{k'+2}$, and $h' = b^{k'+1}$. Order the the b subtrees of the next level from left to right. The probability to recurse the search on the ith subtree is $1/i$. We thus can derive the following formula:

$$L(b,h) = \frac{L(b,h')}{b}\left(1 + \frac{1}{2} + \ldots + \frac{1}{b}\right) .$$

This implies that $L(b,h) \leq L(b,h')(\frac{1+\ln b}{b})$. By induction, $L(b,h) \leq \frac{b}{6}(\frac{(1+\ln b)}{b})^{(k'-1)+1}$. Therefore, $L(b,h) \leq \frac{b}{6}(\frac{(1+\ln b)}{b})^{(k'+1)-1}$ for $k = \lceil \log_b h \rceil - 1 = (k'+2) - 1 = k' + 1$. This completes the sketch of the proof.

Notice that in the search algorithm, after the process of the first node, the search problem is reduced to two subproblems each on a range of the form $[0, h-1]$. Thus the number of accessed elements in the leaf nodes is about twice of $L(b,h)$ derived in the above lemma. □

Notice that in the search algorithm, after the process of the first node, the search problem is reduced to two subproblems each on a range of the form $[0,h]$. Thus the number of accessed elements in the leaf nodes is about twice of $L(b,h)$ derived in the above lemma.

Lemma 2 *The average number of accessed elements in the leaf nodes is about $\frac{b}{3}(\frac{1+\ln b}{b})^{k-1}$ where $k = \lceil \log_b r \rceil - 1$, where r is the range size.*

The above lemma implies that the average number of elements accessed is bounded by $\frac{b}{3} + 2$ when the range size is small, i.e. much less than b^2.

3 The Fat-Nodes Algorithm

In this section, we present an extension to the basic tree-based algorithm, which we shall call the *fat-nodes* algorithm. The basic idea is that for each internal node x, rather than storing only *one* index to the maximum value in $C(x)$, we will store t indices to the top-t values in $C(x)$.

Assume that t is fixed for all internal nodes and $t \leq b$. Then, it is straightforward to construct the fat-nodes tree in a bottom-up manner similar to constructing the tree in the basic tree-based algorithm. The difference is that choosing a maximum value among b values is now changed to choosing the top-t values among b sorted lists, each of length t (or of length 1 for the bottom level).

3.1 The Search Algorithm

The fat-nodes search algorithm requires a change in the definition and processing of nodes in B_{in} and B_{out}. Let $Max\_index(C(x), i)$, $1 \leq i \leq t$, denote the index to the i-th rank value in $C(x)$. We redefine $B_{in}(x, Q) = \{y | y \in B(x, Q), (\exists i)(1 \leq i \leq t)(Max\_index(C(y), i) \in Q)\}$. As before, $B_{out}(x, Q) = B(x, Q) - B_{in}(x, Q)$ is the complementary set of $B_{in}(x, Q)$. For convenience, for each node $y \in B_{in}(x, Q) \cup I(x, Q)$, we define $Max\_inbound\_index(y, Q) = Max\_index(C(y), i)$, where i is the smallest integer satisfying $Max\_index(C(y), i) \in Q$. Clearly, the value indexed by $Max\_inbound\_index(y, Q)$ is the maximum value in the region $C(y) \cap Q$. Also, for each node $z \in B_{out}(x, Q)$, we define $Min\_outbound\_index(z, Q) = Max\_index(C(z), t)$. The value indexed by $Min\_outbound\_index(z, Q)$ (which is outside Q) will be compared to a candidate of maximum value currently found in Q to decide whether the search of the subtree rooted at z can be omitted or not.

The new **get_max_index** function is given below. Note the changes in lines (2), (3) and (5).

Function get_max_index $(x, Q, current\_max\_index)$

(1) **for all** $y \in I(x, Q) \cup B_{in}(x, Q)$ **do**
(2) **if** $(A[Max\_inbound\_index(y, Q)] > A[current\_max\_index])$
(3) $current\_max\_index = Max\_inbound\_index(y, Q)$;
(4) **for all** $z \in B_{out}(x, Q)$ **do**
(5) **if** $(A[Min\_outbound\_index(z, Q)] > A[current\_max\_index])$
(6) $current\_max\_index =$ **get_max_index**$(z, Q \cap C(z), current\_max\_index)$;
(7) **return** $(current\_max\_index)$;

3.2 Complexity Analysis

The worst-case time complexity of the fat-nodes algorithm is worse than that of the basic tree algorithm with the same fanout b. The storage overhead is a factor of t of the basic tree algorithm. The objective of the fat-nodes algorithm is to reduce the average-case complexity of the basic tree algorithm, after normalizing the storage overhead.

Theorem 3 *The average-case complexity of the fat-nodes algorithm is bounded from above by* $\frac{4t^t}{(t+1)^{t+1}}b + 5t + 4$.

3.3 Remarks

From the recursion F described in the proof of the theorem in Appendix, it is revealed that the fat-nodes algorithm may perform better by checking only the rank-1 index and skipping the remaining $t-1$ indices at a node, say x, even if $Max\_index(C(x), 1) \notin Q$. This depends on the ratio of $Q \cap C(x)$ to $C(x)$. A simple rule is that if the ratio is less than $1/\sqrt{b}$, then we only check the rank-1 index. In fact, even the rank-1 index should be skipped for a slightly better average time in this case for any algorithm.

4 The Sorted-Nodes Algorithm

In this section, we present a different extension to the basic tree algorithm, which we shall call the *sorted-nodes* algorithm.

4.1 Constructing the Tree

The sorted-nodes algorithm starts with a tree that was constructed by the basic tree algorithm. Then, for each sibling set of b internal nodes, we partition them into groups, each with up to c consecutive nodes. Then for each group of c nodes, we sort the c indices according to the descending order of the c indexed values. Table 1 gives an example of this.

Note that the storage overhead is the same as the basic tree algorithm. Within each group, the association of each subtree and its corresponding index is lost in the group, but can be uncovered by scanning the sorted index because the fanout b is fixed. Note that we do not partition the leaf nodes into groups because permuting the original array A may increase the time for other OLAP operations that need to access elements of A.

4.2 The Search Algorithm

First, we say an *in-group* is a group consisting of in-nodes only. We say a *boundary group* is a group consisting of either at least one boundary node, or at least one in-node and one out-node. Clearly, finding the *max_index* among all in-nodes in an in-group can be done by looking into the

range start	0	8	16	24	32	40	48	56
range end	7	15	23	31	39	47	55	63
max_index	3	9	20	28	39	43	48	58
max_value	5.7	4.3	8.2	3.1	4.5	7.7	5.1	7.2
group	group 1				group 2			
sorted max_index	20	3	9	28	43	58	48	39
sorted max_value	8.2	5.7	4.3	3.1	7.7	7.2	5.1	4.5

Table 1: An example of the sorted-nodes algorithm with $b = 8$ and $c = 4$.

first (sorted) index of the group. Thus, a factor of c speedup is achieved when compared to the basic tree algorithm. For each boundary group, we perform the procedure in Figure 2.

```
// Searching a Boundary Group
(1)   while there are more unscanned indices in the group do
(2)       scan the next index, denoted by i;
(3)       if (A[i] ≤ current_max_value)
(4)           break;
(5)       else if (i is in query region Q)    // and (A[i] > current_max_value)
(6)           current_max_value = A[i];
(7)           breek;
(8)       else if (index i refers to a boundary node)    // and index i is outside query region Q
(9)           recurse to find the max_index of the region in the boundary node;
          // else index i refers to an out node, which is skipped
(10)  end while
```

Figure 2: The search algorithm for a boundary group.

The main purpose of this procedure is to get to the first index, along the sorted order, of an in-node or a boundary node whose index is in the region Q (line (5)). The branch-and-bound technique is also applied (lines (3) and (4)). If the condition in line (8) is satisfied, then the index i of the current boundary node must be outside the query region Q and a recursion is required for this boundary node. After the recursion, the while-loop continues until all c indices in the group have been checked or the condition in line (3) or line (5) is satisfied. Compared to the basic tree algorithm, generally less in-nodes, but more out-nodes, need to be visited in the boundary group. These two effects seem to offset each other. However, the multiplicative factor of c gained in scanning the in-groups is a clear advantage of the sorted-nodes algorithm over the basic tree algorithm.

4.3 Complexity Analysis

The Worst-Case Complexity Following the worst-case analysis of the basic tree algorithm, it is straightforward to derive the $O((\frac{b}{c}+c)\log_b r)$ worst-case complexity of the sorted-nodes algorithm for an input range of size r.

The Average-Case Complexity We now show that the asymptotical, in the range size, average-case complexity is in fact bounded from above by $\frac{b}{c} + \frac{7c+7}{2} + \frac{c(c+1)^2}{4b}$, much less than the worst-case bound. The reason for the asymptotical result is because the grouping in the sort-nodes algorithm is not applied to the leaf nodes. The time complexity due to the access of leaf nodes is the same as that of the basic tree algorithm, which decreases exponentially as the range size increases, Lemma 2. When the range size is large, the time complexity is dominated by the number of accesses to internal nodes. Thus, we only need to consider the number of accesses to the internal nodes in the following theorem. Note that the derived average-case bound is not a function of the tree size and range size.

Theorem 4 *The asymptotical average-case complexity of the sorted-nodes algorithm is bounded from above by* $\frac{b}{c} + \frac{7c+7}{2} + \frac{c(c+1)^2}{4b}$.

The Choice of c One should choose the value of c to minimize the average-case bound $\frac{b}{c} + \frac{7c+7}{2} + \frac{c(c+1)^2}{4b}$. By choosing $c \approx 0.52\sqrt{b}$, the average-case bound is minimized to be bounded from above by $3.78\sqrt{b} + 3.5$. The choice of c will be confirmed by our experiments presented in Section 6.

Corollary 5 *The average-case complexity of the sorted-nodes algorithm is bounded from above by* $3.78\sqrt{b} + 3.5$.

4.4 Variations

Instead of fully sorting each group, we can sort the top t maxima in the group and swap them with the first t positions of the group. Complexity analysis shows that it is slightly better than the sorted-nodes algorithm in terms of the number of elements accessed. However, in the real implementation, it may involve more overhead in searching the boundary group due to position swapping. We will include its analytical and empirical results in the final version of the paper.

5 The Jump-Nodes Algorithm

We now present the *jump-nodes* algorithm by augmenting the internal nodes of the tree in the basic tree-based algorithm with *reference arrays* to improve the performance. For ease of exposition, we will first motivate the use of reference arrays by discussing how they alone can be used to answer range-max queries. We will then discuss how they can be used in conjunction with the tree structure.

index	0	1	2	3	4	5	6	7	8	9
A	4	2	8	6	9	4	7	3	6	5
R	2	2	4	4	10	6	10	8	10	10
L	-1	0	-1	2	-1	4	4	6	6	8

Figure 3: An example of array A and its precomputed *next-high* reference arrays, R and L.

5.1 A Stand-Alone Reference Array

We first consider an algorithm based on a stand-alone one-dimensional reference array, without the use of the basic tree structure, for a range-max query. Let $r = h - \ell + 1$ be the range size.

For every array element $A[i]$, we precompute $R[i]$ defined as follows:

$$R[i] = \begin{cases} j, & \text{if } i < j < n,\ A[i] < A[j], \text{ and } (\forall k)(i < k < j \Rightarrow A[i] \geq A[k]) \\ n, & \text{if } (\forall k)(i < k < n \Rightarrow A[i] \geq A[k]) \ . \end{cases}$$

That is, $R[i]$ references index j which is the first that is larger than i also satisfying $A[j] > A[i]$. If index j cannot be found, then we let $R[i] = n$ reference an out-of-bound index. We will call R the *next-high* reference array. Figure 3 shows an example of A and R with $n = 10$. (The array L will be defined and used later.)

Finding Maximum from R The algorithm for finding the max_index in a range $(\ell : h)$ of array A based on precomputed R can be easily expressed as follows:

Function find_max_from_R (A, R, ℓ, h)

(1) $max\_index = \ell$;
(2) **while** $(R[max\_index] \leq h)$
(3) $\quad max\_index = R[max\_index]$;
(4) **return** (max_index);

The variable max_index is initialized to ℓ, the leftmost index of the range (line (1)). Then (lines (2)-(3)), max_index is updated repeatedly by following the *next-high* reference $R[max\_index]$ until $R[max\_index]$ references an index which is larger than h, the rightmost index of the range.

Note that constructing R from A should use a stack-based algorithm, which has a time complexity of at most $2n$ comparisons. If a queue-based algorithm is used, a time complexity of about $n^2/2$ comparisons for the worst case, and $\theta(n \log n)$ comparisons for the average case. We omit the details here.

Complexity Analysis of find_max_from_R The time complexity depends on the number of times line (3) is executed, which further depends on the data distribution. The best case is when $A[\ell]$ has the maximum value within the range, while the worst case is when $A[\ell] < \cdots < A[h]$.

We now analyze the average time complexity of the function **find_max_from_R** assuming all the orders on $A[0], \ldots, A[n-1]$ are equally likely. Let $f(r)$ be the average time complexity of this algorithm for a range of size r. One can derive a recursive formula for $f(r)$ as follows:

$$f(r) = 1 + \frac{1}{r}\{f(1) + f(2) + \cdots + f(r-1)\},$$

with the initial value $f(1) = 1$, assuming the entries of A are distinct. (If some elements are equal then $f(r)$ is even smaller.) To understand the above recursion, let $\ell = 0$ and $h = n-1$ and assume without loss of generality that the n entries of A are the integers $1, \ldots, n$. To derive $f(n)$, consider the value $A[0]$ as a random variable that takes on a value i ($i = 1, \ldots, n$) with probability $1/n$. In case $A[0] = i$, the running time of the algorithm depends only on the order in which the values $i+1, \ldots, n$ occur in the entries $A[1], \ldots, A[n-1]$. Although $A[0] = i$, all these orders are still equally likely. Thus, by induction, the expected running time in this case is $1 + f(n-i)$. Thus, we have $f(n) = 1 + \frac{1}{n}\sum_{i=1}^{n-1} f(i)$. The recursive formula implies

$$f(r) - \left(1 - \frac{1}{r}\right) f(r-1) = \frac{1}{r} + \frac{1}{r} f(r-1)$$

so $f(r) - f(r-1) = 1/r$, and therefore

$$f(r) = \sum_{i=1}^{r} \frac{1}{i} \ .$$

It follows that $f(r)$ is asymptotically equal to $\ln r$. Also, it can be shown that $f(r) \leq 1 + \ln r$ for any positive integer r.

Avoiding Worst-Case Behavior Even though the average running time for the algorithm is about $\ln r$, its worst-case running time is r when A is of the ascending order in this range. If the data distribution is known to be close to the ascending order, we can instead precompute a reference array L, analogous to R, defined as follows:

$$L[i] = \begin{cases} j, & \text{if } 0 \leq j < i,\ A[j] > A[i], \text{ and } (\forall k)(j < k < i \Rightarrow A[k] \leq A[i]) \\ -1, & \text{if } (\forall k)(0 \leq k < i \Rightarrow A[k] \leq A[i]). \end{cases}$$

See Figure 3 for an example. We will also call L the *next-high* reference array. Clearly, for each element the array R references the next larger element of *increasing* index, while the array L references the next larger element of *decreasing* index.

There is still one caveat in choosing R or L array. Even though the data distribution of A is already known at the precompute time, the range for which the max_index query will be performed

is not known until query time. If the array has combination of long ascending segment and long descending segment (for instance, certain stock values over a time), choosing to precompute either R or L (but not both) may still result in the $O(r)$ worst-case performance. To avoid such possibly frequent worst-case behavior, we propose an improvement.

Improved Algorithm Assume both R and L next-high reference arrays have been precomputed. The algorithm has two phases. In phase 1 (lines (1)-(2)) it picks the best starting index among k randomly chosen indices in the range and the two boundary indices. We will address how to choose k later. It can be seen that with a probability of $1-(1-\alpha)^k$, the starting index is among the top α fraction ($\alpha < 1$) in the range. The inclusion of two boundary indices in the initial candidate set further speeds up the algorithm for a data distribution close to ascending or descending in the range. In phase 2 (lines (3)-(12)), the algorithm follows either of the two reference arrays to an index with increasing value of A in the range (the best increase is chosen), until both references of the current index are referencing out-of-range indices.

Function find_max_from_RL (A, R, L, ℓ, h)

```
(1)  pick k random indices from the range (ℓ : h), denote them as i_1, ..., i_k;
(2)  pick max_index from {ℓ, h, i_1, ..., i_k} by comparing {A[ℓ], A[h], A[i_1], ..., A[i_k]};
(3)  while ((R[max_index] ≤ h) or (L[max_index ≥ ℓ)) do
(4)      if ((R[max_index] ≤ h) and (L[max_index] ≥ ℓ))
(5)          if (A[R[max_index] ≥ A[L[max_index]])
(6)              max_index = R[max_index];
(7)          else
(8)              max_index = L[max_index];
(9)      else if (R[max_index] ≤ h)
(10)         max_index = R[max_index];
(11)     else
(12)         max_index = L[max_index];
(13) return (max_index);
```

Complexity Analysis as a Function of k We now consider the average case and worst case of the expected running time of function **find_max_from_RL** on any input, as a function of k. With a random sample of size k, this function has an expected running time no greater than $C_1k + C_2(r/k)$ for some constants C_1 and C_2. By choosing $k = \sqrt{C_2r/C_1}$ (compromising the average-case complexity) we get a worst-case bound of $2\sqrt{C_1C_2r}$, the same order as the average-case bound. On the other hand, if we choose $k = \log r$, we can keep the average-case complexity as $O(\log r)$, but with a worst-case complexity of $O(r/\log r)$.

5.2 Embedding Reference Arrays in the Tree Structure

We are now ready to present the *jump-nodes* algorithm by embedding the reference arrays into the basic tree algorithm.

Recall that the original array A is of size n and the fanout of the tree is b. All level-i nodes of the tree can be logically viewed as a contracted array of size $\lceil n/b^i \rceil$, denoted by A_i. We will add and compute a reference array for each A_i, where $1 \leq i \leq \lceil \log_b n \rceil$. However, we do not have a reference array for the original array A in order to conserve space overhead. Thus, the space overhead due to the reference arrays (assuming only R, but not L, is computed) is the same as that for the tree algorithm which is about $n/(b-1)$, and the total space overhead of the jump-nodes algorithm is about $2n/(b-1)$.

To use the added reference arrays in the algorithm, refer to lines (1) and (2) of the function **get_max_index** in Section 2.2.2. Let $S = I(x,Q) \cup B_{in}(x,Q)$ and assume node x is at level $i \geq 2$. The tree algorithm will scan through all nodes $y \in S$ linearly to find the *current_max_index*. With the precomputed reference array, however, one starts from the leftmost node in S and "jumps" through the region of S until the reference array points to an index past the rightmost node in S. More specifically, one can translate the set of nodes S to a region $(\ell : h)$ in A_{i-1}. Then, apply function **find_max_from_R** given in Section 5.1. Thus, the average scan time for two lines of code has been reduced from $2|S|$ to about $\ln |S|$ for uniformly distributed data. Note the factor of 2 of the former is due to the need to access both the index to A and the value in A, while traversing the reference array does not involve accessing the value in A.

5.3 Complexity Analysis

The worst-case time complexity of the jump-nodes algorithm is the same as that of the basic tree algorithm with the same fanout. The following theorem shows that the asymptotical, in the range size, average-case complexity is of order $O(\ln b)$. Based on the similar argument given in Subsection 4.3, we only need to consider the number of accesses to the internal nodes in the following theorem.

Theorem 6 *The asymptotic average-case complexity of the jump-nodes algorithm is bounded from above by* $5 \ln b + 9$.

6 Algorithms Comparison

6.1 Experimental Results

In this subsection, we present some experimental results to compare the basic tree algorithm with the three proposed new algorithms: the fat-nodes, the sorted-nodes and the jump-nodes algorithms. In the experiment, we take a one-dimensional array A of size 2^{22} (elements) with randomly generated

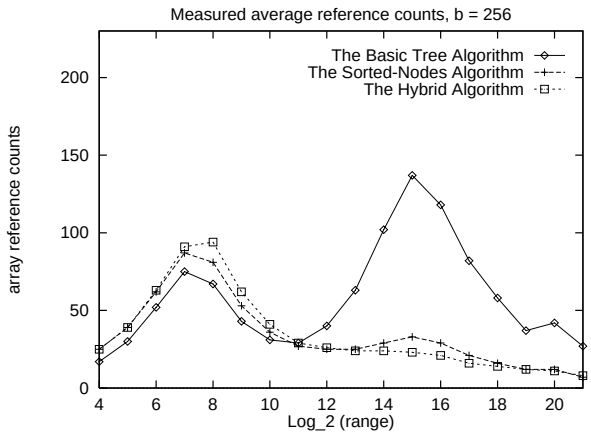

Figure 4: Measured average number of array reference counts as a function of the logarithm of the range size: $n = 2^{22}$ and $b = 256$.

values. For a fair comparison, we let all four algorithms have about the same storage overhead by adjusting the fanout b. Specifically, if we choose b as a fanout of the basic tree algorithm, then we choose tb, b and $2b$ as fanouts of the fat-nodes, sorted-nodes and jump-nodes algorithms, respectively. We only implement $t = 2$ for the fat-nodes algorithm, in such a case it has the same fanout as the jump-nodes algorithm when normalized by storage overhead. For the sorted-nodes algorithm, we choose $c = \sqrt{b}/2$ as suggested by our earlier analysis from Theorem 4.

Figure 4 shows the measured average numbers of array reference counts as a function of the logarithm of the range size, with $b = 1024$ for the basic tree algorithm. (The fanouts of the other algorithms are adjusted accordingly as stated above.) For each data point with a given range size r, $2^4 \leq r \leq n/2$, we take an average of 10000 iterations. For each run, we choose a query region $(\ell : \ell + r - 1)$, where ℓ is randomly chosen from the range $(0 : n/2 - 1)$. Figure 5 shows the measured average run times, corresponding to Figure 4, except that we take an average of 100000 iterations for each data point to get smoother curves. These timings were obtained on a 67-MHz RS/6000 Model-250.

Observe that the array reference counts are pretty consistent with their respective run times. (Some inconsistencies are in part due to different degrees of code optimization.) This indicates that array reference count is a good measure for complexity estimate. Thus, we will base our following discussion on the reference counts.

Figure 6 shows the measured average numbers of array reference counts with $b = 64$ for the basic tree algorithm. Observe that for the basic tree and fat-nodes algorithms, there are roughly two humps when b is chosen around $\sqrt{n}$; and roughly three humps when b is chosen around $\sqrt[3]{n}$. For the jump-nodes and sorted-nodes algorithms, there is only one big hump, which occurs around the first (leftmost) hump of the other algorithms.

For the jump-nodes algorithm, the only hump arises around its fanout value, because we do not

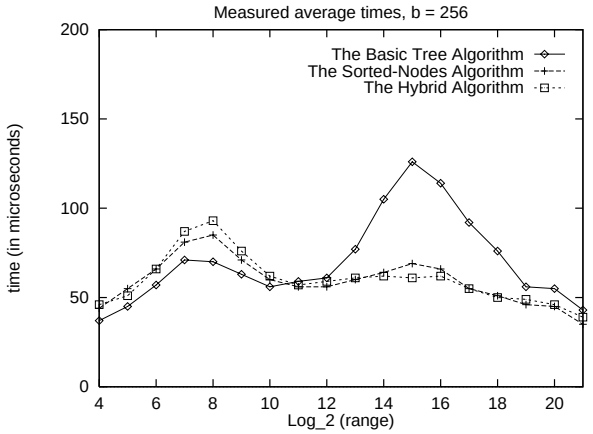

Figure 5: Measured average times of range-max queries corresponding to Figure 4.

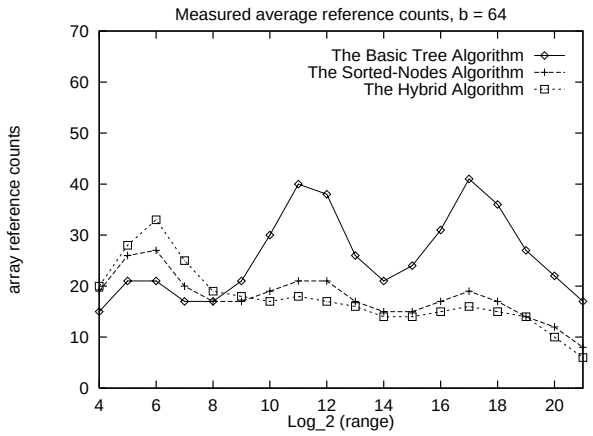

Figure 6: Measured average number of array reference counts as a function of the logarithm of the range size: $n = 2^{22}$ and $b = 64$.

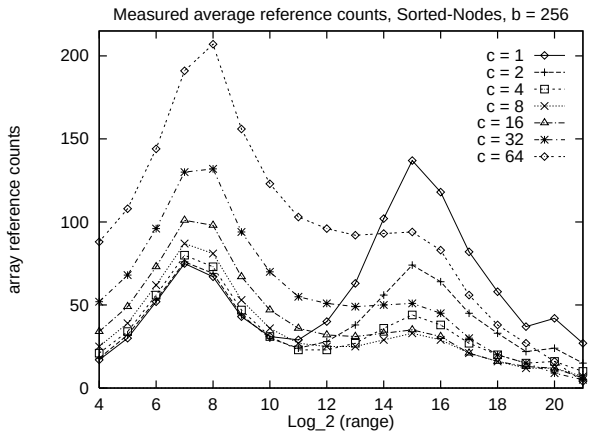

Figure 7: Measured average number of array reference counts for the sorted-nodes algorithm for various values of c: $n = 2^{22}$ and $b = 256$.

build reference array on the leaf nodes. The height of its hump is about twice of other algorithms due to the normalization of fanout from b to $2b$. There is no other humps corresponding to the range size at about b^2 and b^3 due to the time reduction of scanning in-nodes from $O(b)$ to about $O(\ln b)$.

Observe in Figure 4 that the reference counts of the fat-nodes algorithm (with $t = 2$) is generally a right shift of that of the basic tree algorithm. Moreover, the valleys of the fat-nodes algorithm have heights around two-third to half of the values of the corresponding valleys of the basic tree algorithm. On the other hand, the corresponding humps of both algorithms have about the same height, with the exception of the rightmost hump in all three Figures. These phenomena are due to the combined effects of different fanouts ($2b$ vs. b) and different probabilities of visiting the next level of the tree ($(1-\alpha)^2$ versus. $(1-\alpha)$).

To compare the sorted-nodes algorithm against the basic tree algorithm, we show in Figure 7 the effect of choosing different c's. Note that when $c = 1$, the sorted-nodes algorithm degenerates to the basic tree algorithm. By gradually increasing c starting from 1, the first hump also increases slowly, while the other humps decreases first then eventually increases again. A good choice of c experimentally is $c \approx \sqrt{b}/2$. Figure 7 confirms our earlier analysis that choosing $c = \sqrt{b}/2$ (which is 8 in this case) is an overall good choice.

6.2 Analyses

In this subsection, we compare the analytical complexities of the four algorithms and relate them to the experimental figures of the previous section. First, we summarize the recursive formula, $F(h)$, on the expected number of indices accessed for each algorithm, taking a range $(0 : h-1)$. Note that the figures we obtained in the last section is an average (over many ranges of the same size) of average (over random data distributions). However, the function $h(r)$ derived before is a

maximum (over many ranges of the same size) of average. The function $F(h)$, which considers only one instance of the average, has a better resemblance to the figures.

The basic tree algorithm:

$$F(h) \leq 1 + \left(1 - \frac{d_k}{b}\right)(d_k + 1) + \frac{1}{d_k + 1} F(h - d_k b^k) \ .$$

The fat-nodes algorithm:

$$F(h) \leq \min\left(\frac{b}{d_k}, t\right) + \left(1 - \frac{d_k}{b}\right)^t (d_k + 1) + \frac{1}{d_k + 1} F(h - d_k b^k) \ .$$

The sorted-nodes algorithm:

$$F(h) \leq \frac{c+1}{2} + \left(1 - \frac{d_k}{b}\right)\left(\left\lfloor \frac{d_k}{c} \right\rfloor + \frac{c+1}{2}\right) + \frac{1}{d_k + 1} F(h - d_k b^k) \ .$$

The jump-nodes algorithm:

$$F(h) \leq 1 + \left(1 - \frac{d_k}{b}\right)(\ln d_k + 1) + \frac{1}{d_k + 1} F(h - d_k b^k) \ .$$

In Figure 8, we make the plots of $F(h)$, as a function of $\log_2 h$, for all four algorithms and a hybrid algorithm described later. In fact, we further tighten the recursion for various boundary conditions in the plot to get tighter upper bounds. In order to be more consistent with Figure 4, we modify the recursion for the plot such that the reference count for accessing an index stored in an internal node followed by accessing the indexed element stored in a leaf node is counted as two, while accessing the leaf node only is counted as one.

We choose $b = 256$ for the basic tree algorithm and related b for other algorithms so that they all have about the same storage overhead. Note that this figure has the same parameter setup as Figure 4 in n and b. Note that Figure 8 is an evaluation of the analytical function F which gives an upper bounds on the average numbers of array indexing, while Figure 4 is the measured average numbers (of 10000 iterations) of reference counts.

6.3 Hybrid Algorithm

Let $\alpha = d_k/b$. Table 2 summaries the key coefficients in the recursions of the four algorithms and a hybrid algorithm discussed later. The storage overhead is the multiplicative factor to the size of A not including A. The "check in-bound complexity" shows the average number of array indexing in deciding if the precomputed index (or indices) at a given tree node is in the range. The recurse probability gives (an upper bound of) the probability that a recursion is required for the boundary node. (There is at most one boundary node in the recursion due to the definition of F). The fat-nodes algorithm has an advantage here. The last column shows the complexity of scanning the in-nodes.

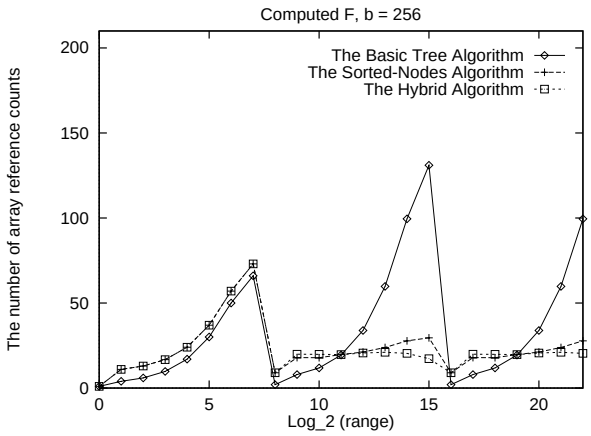

Figure 8: Comparison of $F(h)$ of the five algorithms with $n = 2^{22}$; $b = 256$ for the basic tree algorithm and related b for other algorithms so that the storage overhead is about the same.

algorithm	storage overhead	check in-bound complexity	recurse probability	scanning in-nodes complexity
basic tree	$1/(b-1)$	1	$1-\alpha$	$b/2$
fat-nodes	$t/(b-1)$	$\min(1/\alpha, t)$	$(1-\alpha)^t$	$b/2$
sorted-nodes	$1/(b-1)$	$(c+1)/2$	$1-\alpha$	$b/2c$
jump-nodes	$2/(b-1)$	1	$1-\alpha$	$\approx \ln b$
hybrid	$(c+1)/c(b-1)$	$(c+1)/2$	$1-\alpha$	$\approx \ln b - \ln c$

Table 2: Comparison of the five algorithms.

We now give some suggestions on how these three algorithms should be employed and combined. First, the choice of fanout b depends on how much extra storage is available. For instance, choosing $b = 128$ for the basic tree algorithm means only less than a 1% storage overhead is needed. When the derived $\log_2 b$ (from the given extra storage available) is small relative to $\log_2 n$, there will be many humps (in a time-log(range) plot) and the jump-nodes algorithm is an overall good choice. For the sorted-nodes algorithm, we recommend choosing $c \approx \sqrt{b}/2$ for an overall good performance.

The sorted-nodes algorithm and the jump-nodes algorithm can be combined into a hybrid algorithm. In the structure of the sorted-nodes algorithm, consider a virtual array V formed by the index stored at the first node of each group in a sibling set. (Recall the index of the first node is an index to the maximum value among all nodes in the group.) A reference array will be constructed on V to create short-cuts among groups while scanning in-groups. Thus, the average complexity required for scanning the in-nodes is further reduced from $b/2c$ to $\approx \ln(b/2c) \approx \ln b/2$, when c is chosen as $\sqrt{b}/2$. On the other hand, the storage overhead is only slightly increased from $\frac{1}{b-1}$ to $\frac{1}{b-1}\frac{c+1}{c} \approx \frac{1}{b-\sqrt{b}}$, when c is chosen as $\sqrt{b}/2$. The recursive function F of the hybrid algorithm can

be derived as follows:

$$F(h) \leq \frac{c+1}{2} + \left(1 - \frac{d_k}{b}\right)\left(\ln\left\lfloor\frac{d_k}{c}\right\rfloor + \frac{c+3}{2}\right) + \frac{1}{d_k+1}F(h - d_k b^k) \ .$$

We also implemented the hybrid algorithm and included its reference counts, timings and function F in Figures 4 to 6 and 8, respectively. From our experiments and approximated analysis of the recursion of the hybrid algorithm, the best choice of c of the hybrid algorithm seems to decrease a little bit from the (pure) sorted-nodes algorithm and roughly within the range $\sqrt{b}/4$ to $\sqrt{b}/2$. We choose $c = \sqrt{b}/2 = 8$ for these figures.

In order to normalize the storage overhead, we choose the fanout of the hybrid algorithm as $\frac{c+1}{c}b = \frac{9}{8} \cdot 256 = 288$ in Figures 4 and 5. However, we choose the fanout of the hybrid algorithm as 256 in Figure 8, because the plot of function F is very sensitive to whether the fanout divides the range size h or not (while Figures 4 and 5 are not sensitive to the same thing due to their averages of many randomly chosen ranges).

It is possible to also combine the fat-nodes algorithm into the hybrid algorithm. However, we recommend choosing $t = 1$ for lower levels to conserve storage and a larger t as the level increases. The intuition is that a precomputed index at a higher-level node not only has a higher probability of being visited but also has a higher time reduction once the index is found in the range.

7 The Batch-Update Algorithms

In a typical OLAP environment, updates to data cube are cumulated over a period of time and are performed together as a batch at the end of each period. Thus, it is reasonable to assume a model where a number of updates are issued successively before the next read-only query is issued. In this section, we describe related batch-update algorithms that take a list of update points to array A and modify the precomputed information accordingly in addition to modifying A. The input is a list of update points, each of form $\langle$index, value$\rangle$. For clarity, we assume all update points have different indices (locations) and all indices of update points are in the index domain of A. Both restrictions can be alleviated with minor modifications to the algorithms.

The batch-update algorithm for the basic tree algorithm was given in [HAMS97] in a bottom-up manner. The batch-update algorithm for the fat-nodes algorithm can be easily modified from that of the basic tree algorithm. We omit the details here. In the following, we first describe batch-update for the sorted-nodes algorithm, then for the jump-nodes algorithm.

7.1 Updating for the Sorted-Nodes Algorithm

The batch-update algorithm of the sorted-nodes algorithm is based on that of the basic tree algorithm interleaved with an algorithm to solve the following reduced subproblem.

Let $Q_1, \cdots, Q_c$ be c adjacent ranges. Let x_i be the index of the maximum value in Q_i. Note that due to the fixed fanout of the tree structure, one can derive i from x_i with simple index calculation taking the level of the node and the fanout of the tree. Let C be an array of x_i's sorted descendingly according to the values of $A[x_i]$. For each range Q_i, if there is a change in the new value of x_i or $A[x_i]$, we add the new value of x_i, denoted y_i, into a list U. Thus, U can be denoted as a list of indices: $y_{\delta_1}, \cdots, y_{\delta_u}$ where $u \leq c$. For the reduced subproblem, we assume given C and U, we need to come up with a new array C' which is equivalent to updating x_{δ_i} in C by y_{δ_i} for each $1 \leq i \leq u$, then sorting the updated array according to the (new) indexed values. A straightfoward solution may require a sorting time of c elements.

We now describe an algorithm for the reduced subproblem which only requires a sorting time of u elements and a scanning time of $c + u$ elements. First, we create a c-bit mask, where the i-th bit is one if and only if y_i is in U (i.e., the maximum value or the maximum index of Q_i has been changed). Then, we sort the list U descendingly according to their indexed value. Then, we merge-sort the list C and the update list U with the exception that an entry x_i is ignored if the i-th bit of the mask is one.

7.2 Updating the Reference Arrays

It is sufficient to describe the batch-update algorithm of a (one-dimensional) reference array, given the original array and the input list of update points. For each increase-update $\langle y, v \rangle$, one can derive the new $R[y]$, denoted $R'[y]$ and the new $L[y]$, denoted $L'[y]$. Due to this increase-update $\langle y, v \rangle$, the region $(L'[y] + 1 : L[y])$ of R may become invalid. Similarly, the region $(R[y] : R'[y] - 1)$ of L may become invalid. Figure 9 gives an example of affected R and L for an increase-update $\langle y, v \rangle$, where the values are represented by the vertical bars. For each decrease-update $\langle y, v \rangle$, the region $(L[y] + 1 : L'[y])$ of R may become invalid. Similarly, the region $(R'[y] : R[y] - 1)$ of L may become invalid. At the end of the list scanning, we will take union of all affected regions of R and L, respectively, and run the function **compute_R** given in Section 5.1 and a similar function **compute_L** over these regions.

8 Multi-dimensional Trees

In this section, we discuss how our new techniques can be applied to a d-dimensional tree structure which was built on a d-dimensional array A. Applying the fat-nodes algorithm into a d-dimensional tree is straightforward. We first discuss extensions for the jump-nodes algorithm, then for the sorted-nodes algorithm.

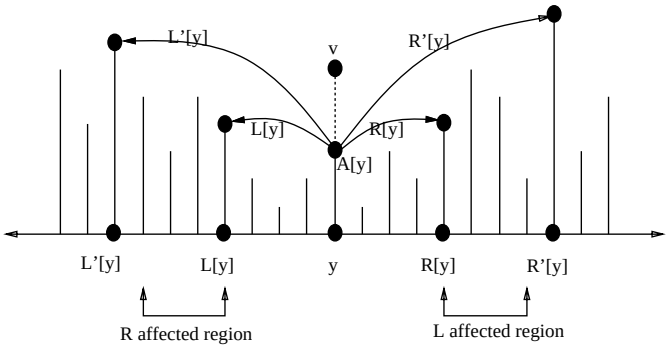

Figure 9: An example of affected R and L in for increase-update $\langle y, v \rangle$.

8.1 Embedding the One-dimensional Reference Arrays

When the tree constructed is d-dimensional, we can embed the (one-dimensional) reference arrays into the tree as follows. As before, we will embed the reference arrays only to all non-leaf nodes A_i, $i > 0$. For each A_i, $i > 0$, which is a d-dimensional contracted array, we choose one dimension, say dimension j, as the single dimension defined in the reference array. Along all other dimensions, we construct as many independent reference arrays as necessary. Specifically, each reference array is of size $\lceil n_j/b^i \rceil$ and there are $\prod_{\forall x \in D, x \neq j} \lceil n_x/b^i \rceil$ independent reference arrays embedded in A_i.

These one-dimensional reference arrays (each along dimension j) in the d-dimensional tree can be used to replace lines (1) and (2) of the function **get_max_index** in Section 2.2.2 and generally result in a speedup in time. Specifically, let $S = I(x, Q) \cup B_{in}(x, Q)$ and assume node x is at level $i \geq 2$. Note that S is a d-dimensional region and can be decomposed into many independent one-dimensional region, denoted $\{S_x\}$, each along dimension j. With the precomputed reference array, one starts from the leftmost node in each S_x and "jumps" through the region of S_x until the reference array points to an index passed the rightmost node in S_x. The same process is repeated for all $\{S_x\}$. Let r_j be the length of region S in dimension j. The time complexity of original lines (1) and (2) (for the tree algorithm) is $2|S|$, while the time complexity for the new lines (1) and (2) (with the use of reference arrays along dimension j) is about $|S|(\ln r_j)/r_j$.

The choice of the single dimension j among all d dimensions can be decided either based on the knowledge of the database administrator or from a collected query log. One simple heuristic is to choose the dimension j which is mostly likely to have the longest length along dimension j in the query region. Since the storage overhead, $2/(b-1)$, can be adjusted by choosing a proper b, one can apply this technique to different dimensions separately, namely having more than one tree. In fact, one can choose different b's for different trees based on a collected query log.

8.2 Multi-dimensional Reference Arrays

A natural question to ask at this stage is whether the reference array techniques generalize to higher dimensions and whether a multi-dimensional reference array alone can be an effective data structure for range-max queries over multi-dimensional data cubes. We give in Appendix a generalization of the reference arrays to multi-dimensions. Such a structure for a d-dimensional array A requires 2^d different references for each element of A, which is a large storage penalty except for small values of d.

This storage penalty can be mitigated by using multi-dimensional arrays in conjunction with the tree structure. All level-i nodes of the tree can be logically viewed as a contracted array of form $\lceil n_1/b^i \rceil \times \cdots \times \lceil n_d/b^i \rceil$, denoted by A_i. Assume that the storage overhead is a small fraction, denoted β, of the size of A. Recall that the fanout is $B = b^d$ and the tree algorithm has a storage overhead of $\beta \approx 1/(b^d - 1)$. If we apply the reference arrays to contracted arrays A_i for all $i \geq 1$, then the space overhead is about $(2^d + 1)\beta$. By choosing a new block parameter b' which is about double of the original block parameter (specifically $b' = \sqrt[d]{2^d + 1}b$), we can keep the space overhead β the same as for the tree algorithm. Alternatively, one can apply the reference arrays only to contracted arrays A_i for all $i \geq 2$ (i.e., ignoring levels 0 and 1). In the case, the additional storage overhead due to the reference arrays is negligible compared to the storage overhead due to the tree structure. The choice depends on the distribution of the sizes of range-max queries.

8.3 The Sorted-Nodes Algorithm

We can clearly embed the one-dimensional grouping of the sorted-nodes algorithm into a d-dimensional tree based on the same technique discussed in Subsection 8.1. Here, we focus on the creation of a d-dimensional structure for the sorted-nodes algorithm.

In the basic tree structure, each sibling set has b^d nodes forming a d-dimensional $b \times \cdots \times b$ cube. For each sibling set, we partition them into groups, each of size $c \times \cdots \times c$. Then, sort all the indices in each group descendingly according to their indexed values as in the one-dimensional case. That is, we only keep a one-dimensional sorted list of size c^d for each group even though the group is a d-dimensional structure. The rest of the algorithm is the same as that of the one-dimensional case. Note that one can extend the hybrid algorithm into a d-dimensional structure by adding the reference array structure, one-dimensional or d-dimensional, among all leaders of the groups in the same sibling set.

9 Summary

In this paper, we propose three different techniques for improving the overall response time of the previous basic tree algorithm [HAMS97]. First, the fat-nodes algorithm keeps the indices of the t largest values with each internal node and uses them to reduce the probability of scanning

lower-level nodes. The sorted-nodes algorithm partitions each sibling set of internal nodes into smaller groups of size $\sqrt{b}/2$ and sorts the precomputed indices within each group according to their indexed values. This speeds up the scanning of in-nodes at the same level from $O(b)$ to $O(\sqrt{b})$ without incurring extra storage overhead. Third, the jump-nodes algorithm augments the tree with a precomputed reference array for each non-leaf level of the tree. This further speeds up the scanning of in-nodes to $O(\ln b)$.

Based on our implementation and theoretical analyses of the three new algorithms, we derived a hybrid algorithm that combines the advantages of sorted-nodes and the jump-nodes algorithms into one. We show through analysis and implementation that the hybrid algorithm improves the overall time complexity of the basic tree algorithm significantly. For range sizes that are larger than $b\sqrt{b}$, the hybrid algorithm improves the basic tree algorithm by as much as a factor of two in measured time and a factor of six in array reference count.

References

[AAD+96] S. Agarwal, R. Agrawal, P.M. Deshpande, A. Gupta, J.F. Naughton, R. Ramakrishnan, and S. Sarawagi. On the computation of multidimensional aggregates. In *Proc. of the 22nd Int'l Conference on Very Large Databases*, pages 506–521, Mumbai (Bombay), India, September 1996.

[AGS97] Rakesh Agrawal, Ashish Gupta, and Sunita Sarawagi. Modeling multidimensional databases. In *Proc. of the 13th Int'l Conference on Data Engineering*, Birmingham, U.K., April 1997.

[Ben80] J.L. Bentley. Multidimensional divide and conquer. *Comm. ACM*, 23(4):214–229, 1980.

[BF79] J. L. Bentley and J. H. Friedman. Data structures for range searching. *Computing Surveys*, 11(4), 1979.

[Cha90] Bernard Chazelle. Lower bounds for orthogonal range searching: Ii. the arithmetic model. *J. ACM*, 37(3):439–463, July 1990.

[CM89] M.C. Chen and L.P. McNamee. The data model and access method of summary data management. *IEEE Transactions on Knowledge and Data Engineering*, 1(4):519–29, 1989.

[Col96] George Colliat. OLAP, relational, and multidimensional database systems. *SIGMOD RECORD*, September 1996.

[CR89] Bernard Chazelle and Burton Rosenberg. Computing partial sums in multidimensional arrays. In *Proc. of the ACM Symp. on Computational Geometry*, pages 131–139, 1989.

[CS94] S. Chaudhuri and K. Shim. Including group-by in query optimization. In *Proc. of the 20th Int'l Conference on Very Large Databases*, pages 354–366, Santiago, Chile, September 1994.

[GBLP96] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tabs and sub-totals. In *Proc. of the 12th Int'l Conference on Data Engineering*, pages 152–159, 1996.

[GHQ95] A. Gupta, V. Harinarayan, and D. Quass. Aggregate-query processing in data warehousing environments. In *Proceedings of the Eighth International Conference on Very Large Databases (VLDB)*, pages 358–369, Zurich, Switzerland, September 1995.

[GHRU97] Himanshu Gupta, Venky Harinarayan, Anand Rajaraman, and Jeffrey D. Ullman. Index selection for OLAP. In *Proc. of the 13th Int'l Conference on Data Engineering*, Birmingham, U.K., April 1997.

[HAMS97] Ching-Tien Ho, Rakesh Agrawal, Nimrod Megiddo, and Ramakrishnan Srikant. Range queries in OLAP data cubes. In *Proc. of the ACM SIGMOD Conference on Management of Data*, Tucson, Arizona, May 1997.

[HRU96] V. Harinarayan, A. Rajaraman, and J.D. Ullman. Implementing data cubes efficiently. In *Proc. of the ACM SIGMOD Conference on Management of Data*, June 1996.

[JS96] T. Johnson and D. Shasha. Hierarchically split cube forests for decision support: description and tuned design, 1996. Working Paper.

[Lom95] D. Lomet, editor. *Special Issue on Materialized Views and Data Warehousing.* IEEE Data Engineering Bulletin, 18(2), June 1995.

[Meh84] Kurt Mehlhorn. *Data Structure and Algorithm 3: Multi-dimensional Searching and Computational Geometry.* Springer-Verlag, 1984.

[Mic92] Z. Michalewicz. *Statistical and Scientific Databases.* Ellis Horwood, 1992.

[Mit70] L. Mitten. Branch and bound methods: General formulation and properties. *Operations Research*, 18:24–34, 1970.

[OLA96] The OLAP Council. *MD-API the OLAP Application Program Interface Version 0.5 Specification*, September 1996.

[Sam89] H. Samet. *The Design and Analysis of Spatial Data Structures.* Addison-Wesley, 1989.

[SDNR96] A. Shukla, P.M. Deshpande, J.F. Naughton, and K. Ramasamy. Storage estimation for multidimensional aggregates in the presence of hierarchies. In *Proc. of the 22nd Int'l Conference on Very Large Databases*, pages 522–531, Mumbai (Bombay), India, September 1996.

[SR96] B. Salzberg and A. Reuter. Indexing for aggregation, 1996. Working Paper.

[STL89] J. Srivastava, J.S.E. Tan, and V.Y. Lum. TBSAM: An access method for efficient processing of statistical queries. *IEEE Transactions on Knowledge and Data Engineering*, 1(4), 1989.

[Vai85] P.M. Vaidya. Space-time tradeoffs for orthogonal range queries. In *Proc. 17th Annual ACM Symp. on Theory of Comput.*, pages 169–174, 1985.

[WL85] D.E. Willard and G.S. Lueker. Adding range restriction capability to dynamic data structures. *J. ACM*, 32(3):597–617, 1985.

[Yao85] Andrew Yao. On the complexity of maintaining partial sums. *SIAM J. Computing*, 14(2):277–288, May 1985.

[YL95] W. P. Yan and P. Larson. Eager aggregation and lazy aggregation. In *Proceedings of the Eighth International Conference on Very Large Databases (VLDB)*, pages 345–357, Zurich, Switzerland, September 1995.

A Appendix — Proofs of Theorems

Theorem 3 The average-case complexity of the fat-nodes algorithm is bounded from above by $\frac{4t^t}{(t+1)^{t+1}}b + 5t + 4$.

Proof: We consider a range of the form $[\ell, h-1]$ and denote by $r = h - \ell$ the size of the range. Let us first analyze the case $\ell = 0$. Suppose $h = \sum_{i=0}^{k} d_i b^i$ where the d_i's are natural numbers smaller than b. Denote by $F(h)$ the expected number of indices checked during the processing of the interval $[0, h-1]$. Suppose, without loss of generality, that $d_k \geq 1$. First, if $h = b^k$, then $F(h) = 1$ since the maximum over the interval has been precomputed. Next, suppose $b^k < h < b^{k+1}$. Consider the ranges $R_i = [(i-1)b^k, ib^k - 1]$, $i = 1, 2, \ldots, b$. Let $\alpha = d_k/b$. The probability that the maximum over $X = [0, b^{k+1} - 1]$ falls in the range $Y = [0, d_k b^k - 1] = R_1 \cup \cdots \cup R_{d_k}$ is $d_k/b = \alpha$. If it does, then only one index access is needed. Otherwise, we consider the overall probability that the second maximum over X falls in the range Y while the maximum over X does not fall in range Y. This probability is $(1-\alpha)\alpha$ and, thus, its associated expected number of index access is $2(1-\alpha)\alpha$. In general, the expected number of index accesses associated with the probability that none of the top $i-1$ maxima over X falls in the range Y while the i-th maximum falls in the range Y, where $1 \leq i \leq t$, is $i(1-\alpha)^{i-1}\alpha$. Thus, the expected number of indices accessed is

$$
\begin{aligned}
F(h) \leq\ & [\alpha + 2(1-\alpha)\alpha + 3(1-\alpha)^2\alpha + \cdots + t(1-\alpha)^{t-1}\alpha] \\
& + (1-\alpha)^t(t + d_k + 1) + \frac{1}{d_k+1}F(h - d_k b^k) \\
=\ & \left[\frac{1-(1-\alpha)^t}{\alpha} - t(1-\alpha)^t\right] + t(1-\alpha)^t + (1-\alpha)^t(d_k+1) + \frac{1}{d_k+1}F(h - d_k b^k) \\
=\ & \frac{1-(1-\alpha)^t}{\alpha} + (1-\alpha)^t(d_k+1) + \frac{1}{d_k+1}F(h - d_k b^k) \\
=\ & \left[\frac{1-(1-\alpha)^t}{\alpha} + (1-\alpha)^t\right] + \alpha(1-\alpha)^t b + \frac{1}{d_k+1}F(h - d_k b^k)\ . \text{ (by substituting } d_k = \alpha b)
\end{aligned}
$$

We now maximize the three terms separately, as a function of α or d_k. Note that $\alpha = d_k/b$ and $0 < d_k < b$, thus $0 < \alpha < 1$. The first term simplies to $[1-(1-\alpha)^{t+1}]/\alpha$, which is maximized to $t+1$ when $\alpha \to 0$. The second term $\alpha(1-\alpha)^t b$ is maximized when $\alpha = \frac{1}{t+1}$, yielding a value of $\frac{t^t}{(t+1)^{t+1}}b$. The third term is maximized to $\frac{1}{2}F(h - d_k b^k)$. Thus, the above equation continues as

$$
\begin{aligned}
F(h) &\leq (t+1) + \frac{t^t}{(t+1)^{t+1}}b + \frac{1}{2}F(h - d_k b^k) \\
&\leq 2(t+1) + \frac{2t^t}{(t+1)^{t+1}}b\ .
\end{aligned}
$$

Next, for a general interval $[\ell, h-1]$, after the first level, if the maximum has not been found, then the problem is reduced to at most two problems over ranges of the form $[\ell', n-1]$ and $[0, h'-1]$, to which our upper bound applies.

Suppose the smallest complete subtree that covers the given range is of size b^{k+1}. Denote by x the number of subtrees of size b^k contained in the range. Obviously, $0 \leq x \leq rb^{-k}$, so $r \geq xb^k$. Then the first phase accesses up to t indices at the root, and with probability $(1-\frac{r}{b^{k+1}})^t \leq (1-\frac{x}{b})^t$ we will have to access the roots of x internal subtrees in addition to, on the average, at most $\frac{4}{t+1} + \frac{4t^t}{(t+1)^{t+1}}b$ more nodes in the two boundary problems.

Thus, the expected number of accesses for an interval of length r is bounded from above by

$$h(r) \leq t + \left(1 - \frac{x}{b}\right)\left(x + 4(t+1) + \frac{4t^t}{(t+1)^{t+1}}b\right) \leq \frac{4t^t}{(t+1)^{t+1}}b + 5t + 4,$$

where the function attains its maximum over $[0, b-1]$ at $x = 0$. This proves our claim. □

Theorem 4 The asymptotical average-case complexity of the sorted-nodes algorithm is bounded from above by $\frac{b}{c} + \frac{7c+7}{2} + \frac{c(c+1)^2}{4b}$.

Proof: We consider a range of the form $[\ell, h-1]$ and denote by $r = h - \ell$ the size of the range. Let us first analyze the case $\ell = 0$. Suppose $h = \sum_{i=0}^{k} d_i b^i$ where the d_i's are natural numbers smaller than b. Denote by $F(h)$ the expected number of indices checked during the processing of the interval $[0, h-1]$. Suppose, without loss of generality, that $d_k \geq 1$. First, if $h = b^k$, then $F(h) = \frac{c+1}{2}$ since the maximum over the interval can be computed by examining one block of c maximums. Next, suppose $b^k < h < b^{k+1}$. Consider the ranges $R_i = [(i-1)b^k, ib^k - 1]$, $i = 1, 2, \ldots, b$. The probability that the maximum over $[0, b^{k+1} - 1]$ falls in the range $[0, d_k b^k - 1] = R_1 \cup \cdots \cup R_{d_k}$ is d_k/b. If it does, then we are done. Otherwise, we first access the maxima over the ranges $R_1, \ldots, R_{d_k+1}$. Notice that the maximum can be computed by examining $\left\lfloor \frac{d_k}{c} \right\rfloor + \frac{c+1}{2}$ indices in average. If the maximum falls in the ranges $R_1 \cup \ldots \cup R_{d_k}$, then we are done; if not, then we solve the problem, recursively, over the range $[d_k b^k, h-1]$, as a subrange of R_{d_k+1}.[2] It is easy to see that in the latter case all the orders over the data values at points in R_{d_k+1} remain equally probable, so we can use the same function F for describing the average-case complexity. Next, we analyze the probabilities of the events that determine the expected number of indices accessed.

Let E_j denote the event in which the maximum over $[0, b^{k+1} - 1]$ does not fall in $R_1 \cup \cdots \cup R_j$. Thus $\Pr(E_j) = 1 - j/b$. Denote by F_j the event in which the maximum over the R_{j+1} is less than the maximum over $R_1 \cup \cdots \cup R_j$. By definition, $E_j \cap F_j = E_{j+1} \cap F_j$. Thus,

$$\Pr(E_j \cap F_j) = \Pr(E_{j+1} \cap F_j) = \Pr(E_{j+1}) \Pr(F_j | E_{j+1}) = \left(1 - \frac{j+1}{b}\right)\frac{j}{j+1}\ .$$

Denote by $\bar{F}_j$ the complement of F_j. It follows that

$$\Pr(E_j \cap \bar{F}_j) = \Pr(E_j) - \Pr(E_j \cap F_j) = 1 - \frac{j}{b} - \left(1 - \frac{j+1}{b}\right)\frac{j}{j+1} = \frac{1}{j+1}\ .$$

[2] In fact, if the maximum over R_{d_k+1} falls in the query range we are done, but we ignore this event in the probabilistic analysis.

We can now estimate $F(h)$ as follows.

$$\begin{aligned} F(h) &\leq \frac{c+1}{2} + \Pr(E_{d_k}) \left(\left\lfloor \frac{d_k}{c} \right\rfloor + \frac{c+1}{2} \right) + \Pr(E_{d_k} \cap \bar{F}_{d_k}) F(h - d_k b^k) \\ &= \frac{c+1}{2} + \left(1 - \frac{d_k}{b}\right) \left(\left\lfloor \frac{d_k}{c} \right\rfloor + \frac{c+1}{2} \right) + \frac{1}{d_k + 1} F(h - d_k b^k) \ . \end{aligned}$$

To establish that the expected time is bounded by a constant, note that

$$\begin{aligned} F(h) &< \max_x \left\{ \frac{c+1}{2} + \left(1 - \frac{x}{b}\right) \left(\frac{x}{c} + \frac{c+1}{2} \right) \right\} + \frac{1}{2} F(h - d_k b^k) \\ &= \frac{b}{4c} + \frac{3c+3}{4} + \frac{c(c+1)^2}{16b} + \frac{1}{2} F(h - d_k b^k) \end{aligned}$$

(where $x = \frac{2b - c(c+1)}{4}$ gives the maximum) and this inequality implies, by induction, that for all h,

$$F(h) < \frac{b}{2c} + \frac{3c+3}{2} + \frac{c(c+1)^2}{8b} \ .$$

Next, for a general interval $[\ell, h-1]$, after the first level, if the maximum has not been found, then the problem is reduced to at most two problems over ranges of the form $[\ell', n-1]$ and $[0, h'-1]$, to which our upper bound applies.

Suppose the smallest complete subtree that covers the given range is of size b^{k+1}. Denote by x the number of subtrees of size b^k contained in the range. Obviously, $0 \leq x \leq r b^{-k}$, so $r \geq x b^k$. Then the first phase accesses, on the average, $\frac{c+1}{2}$ indices to find the maximum at the starting node, and with probability $1 - \frac{r}{b^{k+1}}$ we will have to access, on the average, the roots of $\lfloor \frac{x}{c} \rfloor + 2 \cdot \frac{c+1}{2}$ internal subtrees in addition to, on the average, at most $\frac{b}{c} + 3c + 3 + \frac{c(c+1)^2}{4b}$ more nodes in the two boundary problems.

Thus, the expected number of accesses for an interval of length r is bounded from above by

$$\begin{aligned} h(r) &= \frac{c+1}{2} + \left(1 - \frac{r}{b^{k+1}}\right) \left(\left\lfloor \frac{x}{c} \right\rfloor + \frac{b}{c} + 3c + 3 + \frac{c(c+1)^2}{4b} \right) \\ &\leq \frac{c+1}{2} + \left(1 - \frac{x}{b}\right) \left(\frac{x}{c} + \frac{b}{c} + 3c + 3 + \frac{c(c+1)^2}{4b} \right) \ . \end{aligned}$$

The function on the right-hand side attains its maximum over $[0, b-1]$ at $x = 0$, where its value is $\frac{b}{c} + \frac{7c+7}{2} + \frac{c(c+1)^2}{4b}$. This proves our claim. □

Theorem 6 The asymptotical average-case complexity of the jump-nodes algorithm is bounded from above by $5 \ln b + 9$.

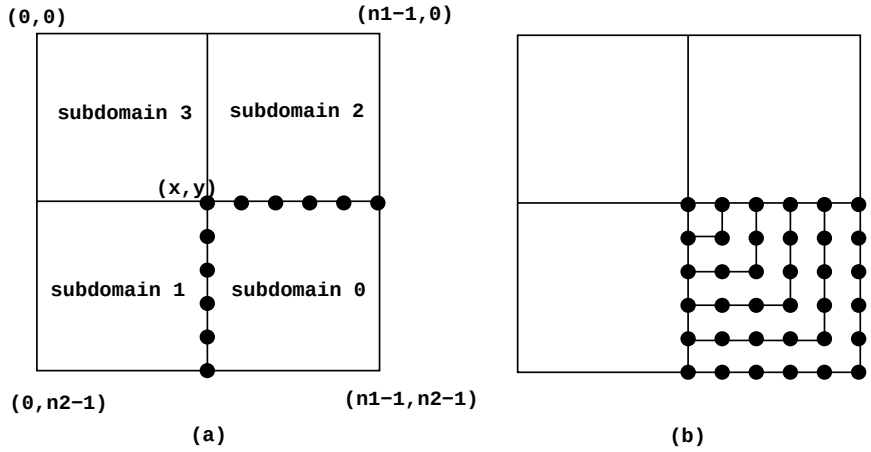

Figure 10: (a) The four subdomains. (b) The waves in subdomain 0.

Proof: Refer to the proof of Theorem 4 for the definitions of $F(h)$ and $h(r)$. The function $F(h)$ and $h(r)$ for the jump-nodes algorithms can be derived as follows:

$$\begin{aligned} F(h) &\leq 1 + \left(1 - \frac{d_k}{b}\right)(\ln d_k + 1) + \frac{1}{d_k + 1} F(h - d_k b^k) \\ &\leq \ln b + 2 + \frac{1}{2} F((h - d_k b^k) \\ &\leq 2 \ln b + 4. \\ h(r) &\leq 1 + (1 - \frac{x}{b})(\ln x + 4 \ln b + 8) \text{ (where } 1 \leq x \leq b) \\ &\leq 5 \ln b + 9. \end{aligned}$$

□

B Appendix — Multi-dimensional Reference Arrays

For clarity, we first generalize the reference arrays to the two-dimensional (rectangular) region. Then give the generalization to the d-dimensional region.

B.1 The Two-Dimensional Case

Given a two-dimensional domain of $N = (0 : n_1 - 1, 0 : n_2 - 1)$ and a data point (x, y) in the domain N, we define *subdomains* 0, 1, 2 and 3 with respect to the data point and the domain N as the domains $(x : n_1 - 1, y : n_2 - 1)$, $(0 : x, y : n_2 - 1)$, $(x : n_1 - 1, 0 : y)$ and $(0 : x, 0 : y)$, respectively, Figure 10(a). Note that we purposely define the subdomain to also cover the two boundary lines adjoining its two adjacent subdomains. For instance, the points shown in the figure are on the two boundary lines of the subdomain 0.

For clarity, we use the following abbreviated notation. We denote the region

$$(\underline{l_1} : \overline{h_1}, \underline{l_2} : \overline{h_2}) = (\min(l_1, 0) : \max(h_1, n_1 - 1), \min(l_2, 0) : \max(h_2, n_2 - 1)).$$

That is, an "underline" and "overline" in the first (resp. second) argument guarantee that the low and high indices are bounded from below by 0 and from above by $n_1 - 1$ (resp. $n_2 - 1$), respectively. Given an array A of domain N, we define, for each data point $(x, y) \in N$,

$$P_0(x,y) = \begin{cases} (x', y'), & \text{if } A[x', y'] = Max(x : \overline{x + z}, y : \overline{y + z}) > A[x, y] \\ & \quad \text{and } A[x, y] = Max(x : \overline{x + z - 1}, y : \overline{y + z - 1}) \\ & \quad \text{where } z = \max(|x' - x|, |y' - y|). \\ \perp, & \text{if } A[x, y] = Max(x : n_1 - 1, y : n_2 - 1). \end{cases}$$

In other words, if we search a larger value than $A[x, y]$ following the order of the right-angled waves in the subdomain 0 originated from point (x, y), Figure 10(b), then (x', y') is on the first wave that we find a larger value in it, and $A(x', y')$ has the largest value in the wave. (If there is more than one point with the largest value in the same wave, then we arbitrarily choose one.) Note that $P_0(x, y) = \perp$ (a symbol for an undefined value) means no larger value than $A[x, y]$ is found in the subdomain. Figure 11 shows an example of $A(x, y)$ and $P_0(x, y)$, for $0 \le x \le 3$ and $0 \le y \le 5$.

$A(x,y)$	$x=0$	$x=1$	$x=2$	$x=3$
$y=0$	3	2	6	5
$y=1$	4	9	7	6
$y=2$	6	7	8	9
$y=3$	8	5	3	2
$y=4$	3	4	8	3
$y=5$	5	3	4	7

$P_0(x,y)$	$x=0$	$x=1$	$x=2$	$x=3$
$y=0$	(1,1)	(1,1)	(2,1)	(3,1)
$y=1$	(1,1)	$\perp$	(3,2)	(3,2)
$y=2$	(0,3)	(2,2)	(3,2)	$\perp$
$y=3$	$\perp$	(2,4)	(2,4)	(3,4)
$y=4$	(0,5)	(2,4)	$\perp$	(3,5)
$y=5$	(3,5)	(2,5)	(3,5)	$\perp$

Figure 11: Example of $A(x, y)$, on the left, and $P_0(x, y)$, on the right.

Similarly, we define $P_1(x, y)$, $P_2(x, y)$ and $P_3(x, y)$ as follows. Here, $z = \max(|x' - x|, |y' - y|)$ as before.

$$P_1(x,y) = \begin{cases} (x', y'), & \text{if } A[x', y'] = Max(\underline{x - z} : x, y : \overline{y + z}) > A[x, y] \\ & \quad \text{and } A[x, y] = Max(\underline{x - z + 1} : x, y : \overline{y + z - 1}), \\ \perp, & \text{if } A[x, y] = Max(0 : x, y : n_2 - 1). \end{cases}$$

$$P_2(x,y) = \begin{cases} (x', y'), & \text{if } A[x', y'] = Max(x : \overline{x + z}, \underline{y - z} : y) > A[x, y] \\ & \quad \text{and } A[x, y] = Max(x : \overline{x + z - 1}, \underline{y - z + 1} : y), \\ \perp, & \text{if } A[x, y] = Max(x : n_1 - 1, 0 : y). \end{cases}$$

$$P_3(x,y) = \begin{cases} (x', y'), & \text{if } A[x', y'] = Max(\underline{x - z} : x, \underline{y - z} : y) > A[x, y] \\ & \quad \text{and } A[x, y] = Max(\underline{x - z + 1} : x, \underline{y - z + 1} : y), \\ \perp, & \text{if } A[x, y] = Max(0 : x, 0 : y). \end{cases}$$

As before, we will call P_0 through P_3 the *next-high* reference arrays, a generalization of R and L arrays for the 1-dimensional case.

The algorithm takes as inputs a two-dimensional integer domain N, a two-dimensional array A of domain N, the precomputed four next-high reference arrays of the same domain N, and a two-dimensional integer region $Q \subset N$. It returns a location $(x, y) \in Q$ such that $A[x, y]$ is the maximum in region Q. The algorithm consists of three phases. We will use the terms location and (data) point interchangeably.

- Phase 1: The objective of this phase is to pick a good starting point (location).

  First, pick k random points in Q, for an appropriately chosen k. Then, pick the point with the maximum value among the k random points and the four corner points.

- Phase 2: This is the main phase that generally iterates many times. The objective of this phase is to monotonically move towards a "better" point in the region (i.e., with larger values) following the best choice among the four next-high references.

  The first iteration starts with the point (x, y) chosen in phase 1. Let $C = \{P_0(x, y), P_1(x, y), P_2(x, y), P_3(x, y)\} \cap Q$. In general, iteration i starts with a point (x, y) computed in the iteration $i - 1$ and does the following:

  - it either generates the output point (x', y') (as input for iteration $i+1$), where $(x', y') \in C$ and $A[x', y'] \geq A[x'', y'']$ for all other $(x'', y'') \in C$,
  - or generates the output $\perp$ if C is empty (i.e., all four next-high references are outside region Q), then the iterations are stopped and the input point (x, y) is passed to phase 3.

- Phase 3: This is the clean-up phase which decides whether there are any left-over subregions (of Q) not covered in phase 2. For each uncovered subregion Q', we recursively call this algorithm with $Q = Q'$. In the worst case, there are 4 left-over subregions to be recursed on. Figure 12(a) shows the beginning of phase 3 with point (x, y) and $P_0(x, y) = (x', y')$ in the subdomain 0. Figure 12(b) shows that the Max_index in the shaded region is (x, y), from the Definition of P_0. The Max_index of the left-over shaded region in Figure 12(c) is still unknown, which needs to be recursed into the algorithm as a new region. Finally, the output Max_index of region Q is then chosen among data points (x, y) and all output data points from the recursion.

B.2 The d-Dimensional Case

Recall that N is the d-dimensional domain $(0 : n_1 - 1, \ldots, 0 : n_d - 1)$ of A. Given a d-dimensional domain N and a data point $X = (x_1, x_2, \ldots, x_d)$ in the domain N, define the i-th *subdomain*, for

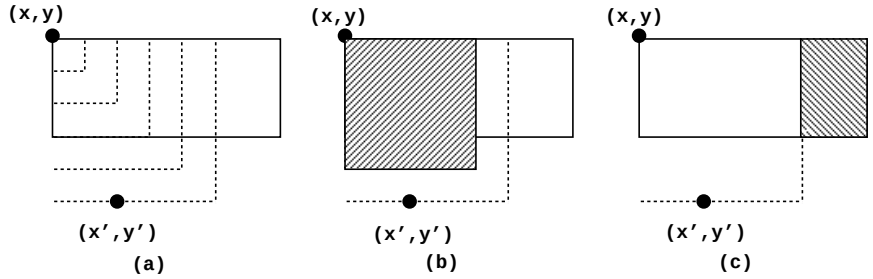

Figure 12: (a) The beginning of phase 3 with points (x, y) and (x', y'). (b) The Max_index of the shaded region is (x, y). (c) The Max_index of the left-over shaded region is still unknown which needs to be recursed into the algorithm as a new region.

all $0 \leq i < 2^d$, with respect to the data point X and the domain N, as the domain $(r_1, \ldots, r_d)$, where $r_j = x_j : n_j - 1$ if the j-th bit of i is 0, and $r_j = 0 : x_j$ if the j-th bit of i is 1.

In the following, let $z = \max(|x_1 - x'_1|, \ldots, |x_d - x'_d|)$. As before, we use an "underline" and "overline" in the j-th argument to guarantee that the low and high indices are bounded from below by 0 and from above by $n_j - 1$, respectively. Also let $r'_j = x_j : \overline{x_j + z}$ if the j-th bit of i is 0 and $r'_j = \underline{x_j - z} : x_j$ if the j-th bit of i is 1. Let $r''_j = x_j : \overline{x_j + z - 1}$ if the j-th bit of i is 0 and $r''_j = \underline{x_j - z + 1} : x_j$ if the j-th bit of i is 1. Clearly, r_j, r'_j and r''_j are all functions of i. The parameter i is omitted for clarity. We will use r_j, r'_j and r''_j to specify three different ranges in the j-th dimension for each given i. Recall that Max takes a region as an argument and returns the maximum value of A in that region.

Given the array A, for each data point $(x_1, \ldots, x_d) \in N$ and each $0 \leq i < 2^d$ we define

$$P_i(x_1, \ldots, x_d) = \begin{cases} (x'_1, \ldots, x'_d), & \text{if } A[x'_1, \ldots, x'_d] = Max(r'_1, \ldots, r'_d) > A[x_1, \ldots, x_d] \\ & \quad \text{and } A[x_1, \ldots, x_d] = Max(r''_1, \ldots, r''_d). \\ \perp, & \text{if } A[x_1, \ldots, x_d] = Max(r_1, \ldots, r_d). \end{cases}$$

Intuitively, if we search a larger value than $A[X]$ following the order of a d-dimensional right-angled waves in the subdomain i originated from point X, then $(x'_1, \ldots, x'_d)$ is on the first wave that we find a larger value in it, and $A[x'_1, \ldots, x'_d]$ has the largest value in the wave. Also, $P_i(X) = \perp$ means no larger value is found in the subdomain i.

The algorithm for the d-dimensional case can be derived by generalizing the two-dimensional algorithm as follows. In the first phase, the four corner points are replaced by 2^d corner points. In the second phase, we define $C = \{P_i(x_1, \ldots, x_d) | 0 \leq i < 2^d\} \cap Q$. In the third phase, deriving the left-over d-dimensional subregions is more complicated. The left-over subregion in each of the 2^d subdomains may not be convex. However, it can be partitioned into up to $d - 1$ convex regions. The specific partitioning is described next.

Let $(x_1, \ldots, x_d)$ be the data point passed to phase 3. Let Q be the input region to the algorithm. Let $P_0(x_1, \ldots, x_d) = (x'_1, \ldots, x'_d) = X'$. Let $(x_1 : y_1, \ldots, x_d : y_d)$ be the intersection of region

Q and subdomain 0, i.e., $Y = (y_1, \ldots, y_d)$ is the corner node of Q in subdomain 0. Recall $z = \max(|x'_1 - x_1|, \ldots, |x'_d - x_d|)$. The left-over subregion in subdomain 0 is partitioned into the following convex regions:

$$(x'_1 : y_1, x_2 : y_2, \ldots, x_d : y_d), (x_1 : x'_1 - 1, x'_2 : y_2, x_3 : y_3, \ldots, x_d, y_d),$$

$$\cdots, (x_1 : x'_1 - 1, \ldots, x_{d-1} : x'_{d-1} - 1, x'_d : y_d).$$

Here, any region containing $\ell_j : h_j$ with $\ell_j > h_j$ is an empty region. Note that among these d regions there is at least one empty one. This is because X' lies outside region Q while Y is the corner node of Q, thus there exists a j, $1 \leq j \leq d$, such that $x'_j > y_j$. The left-over subregions for other subdomains can be similarly partitioned. The only change is that for subdomain i and for each j such that the j-th bit of i is 0, the two values specifying the range in the j-th dimension are reversed.