Range Queries in OLAP Data Cubes

Ching-Tien Ho *Rakesh Agrawal* *Nimrod Megiddo* *Ramakrishnan Srikant*

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

Abstract

A range query applies an aggregation operation over all selected cells of an OLAP data cube where the selection is specified by providing ranges of values for numeric dimensions. We present fast algorithms for range queries for two types of aggregation operations: SUM and MAX. These two operations cover techniques required for most popular aggregation operations, such as those supported by SQL.

For range-sum queries, the essential idea is to precompute some auxiliary information (prefix sums) that is used to answer ad hoc queries at run-time. By maintaining auxiliary information which is of the same size as the data cube, all range queries for a given cube can be answered in constant time, irrespective of the size of the sub-cube circumscribed by a query. Alternatively, one can keep auxiliary information which is $1/b^d$ of the size of the d-dimensional data cube. Response to a range query may now require access to some cells of the data cube in addition to the access to the auxiliary information, but the overall time complexity is typically reduced significantly. We also discuss how the precomputed information is incrementally updated by batching updates to the data cube. Finally, we present algorithms for choosing the subset of the data cube dimensions for which the auxiliary information is computed and the blocking factor to use for each such subset.

Our approach to answering range-max queries is based on precomputed max over balanced hierarchical tree structures. We use a branch-and-bound-like procedure to speed up the finding of max in a region. We also show that with a branch-and-bound procedure, the average-case complexity is much smaller than the worst-case complexity.

1 Introduction

On-Line Analytical Processing (OLAP) [Cod93] allows companies to analyze aggregate databases built from their data warehouses. An increasingly popular data model for OLAP applications is the multidimensional database (MDDB) [OLA96] [AGS97], also known as data cube [GBLP96]. To build an MDDB from a data warehouse, certain attributes (typically 5 to 10) are selected. Thus, each data record contains a value for each of these attributes. Some of these attributes are chosen as metrics of interest and are referred to as the *measure attributes*. The remaining attributes, say d of them, are referred to as *dimensions* or the *functional attributes*. The measure attributes of those records with the same functional attributes values are combined (e.g. summed up) into an aggregate value. Thus, an MDDB can be viewed as a d-dimensional array, indexed by the values of the d functional attributes, whose cells contain the values of the measure attributes for the corresponding combination of functional attributes. Consider a data cube from insurance company as an example. Assume the data cube has four functional attributes (dimensions): age, year, state, and (insurance) type. Further assume that the domain of age is 1 to 100, of year is 1987 to 1996, of state is the 50 states in U.S., and of type is {home, auto, health}. The data cube will have $100 \times 10 \times 50 \times 3$ cells, with each cell containing the total revenue (the measure attribute) for the corresponding combination of age, year, state, and type.

Recently, [GBLP96] proposed that the domain of each functional attribute be augmented with an additional value for each aggregation operation, denoted by "all", to store aggregated values of the measure attributes in all of the cells along that functional attribute. In the above example, the data cube will be extended to $101 \times 11 \times 51 \times 4$ if only one aggregation operation, say SUM, is considered. Thus, any sum-query of (age, year, state, type), where each attribute is either a singleton value in its domain or *all*, can be answered by accessing a single cell in the extended data cube. For instance, the total amount of revenue for the auto insurance in the whole US in 1995 is a query specified by (all, 1995, all, auto), which can be answered in one cell access. We call such queries *singleton* queries.

We consider a class of queries over data cubes, which we shall call *range queries*, that apply a given aggregation operation over selected cells where the selection is specified as contiguous ranges in the domains of some of the attributes. In particular, we consider two different types of aggregation operations: one typified by SUM and another by MAX. The corresponding range queries are termed range-sum and range-max queries, respectively.

Such range queries are frequent with respect to numeric attributes with natural semantics in ordering, such as age, time, salary, etc. Consider a range-sum query to the same insurance data cube: find the revenue from customers with an age from 37 to 52, in a year from 1988 to 1996, in all of U.S., and with auto insurance. To answer this query, we can use precomputed values for "all" in the state domain.

However, since the query specifies 16 (but not *all*) different values in the age domain, and 9 (but not *all*) different values in the year domain, one needs to access $16 \times 9 \times 1 \times 1$ cells in the extended data cube and sum them up before returning the answer. 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.

Contribution We present different techniques to speedup range-sum queries and range-max queries, respectively. The main idea for speeding up range-sum queries is to precompute multidimensional prefix-sums of the data cube. By precomputing as many prefix-sums as the number of elements in the original data cube, any range-sum query can be answered by accessing and combining 2^d appropriate prefix-sums, where d is the number of numeric dimensions for which ranges have been specified in the query. This compares to a naive algorithm of a time complexity equal to the volume of the query sub-cube, which is at least 2^d if the range of every dimension in the query is of size at least two. If storage is a premium, then the original data cube can be discarded, and singleton queries can be answered by using precomputed prefix-sums, as any cell of the data cube can be computed with the same time-complexity as a range-sum query. We also propose an alternative technique that trades time for space and stores prefix-sums only at a block level. Any range sum query can now be answered by accessing and combining 2^d prefix-sums as well as some cells of the data cube.

Our approach to answering range-max queries is based on precomputed max over balanced hierarchical tree structures. We use a branch-and-bound[Mit70]-like procedure to speed up the finding of max in a region, based on the following property of max: given two sets of numbers S_1 and S_2 and the precomputed $\max(S_1)$, if there exists a number $i \in S_2$ such that $i \geq \max(S_1)$ then $\max(S_2) = \max(S_2 - S_1)$ (even if $S_2 \not\supset S_1$). We also show that with a branch-and-bound procedure, the average-case complexity is much smaller than the worst-case complexity.

We present algorithms for incrementally updating the precomputed information by batching updates to the data cube. We also discuss issues important for a practical realization of the proposed techniques. We discuss how to choose the subset of the data cube dimensions for which the auxiliary information is computed and the blocking factor to use for each such subset.

Techniques described for range-sum queries can be applied to any binary operator $\oplus$ for which there exists an *inverse* binary operator $\ominus$ such that $a \oplus b \ominus b = a$, for any a and b in the domain. Examples of such $(\oplus, \ominus)$ operators include $(+, -)$, (bitwise-exclusive-or, bitwise-exclusive-or), (exclusive-or, exclusive-or), and (multiplication, division, for a domain excluding zero). We describe the algorithms for range-sums based on the $(+, -)$ operators, as SUM is the most prevalent aggregation operation in OLAP applications. Note that the aggregation operator COUNT and AVERAGE for a range can also be derived using the same algorithm: COUNT is a special case of SUM and AVERAGE can be obtained by keeping the 2-tuple (sum, count). Note also that ROLLING SUM and ROLLING AVERAGE, two other frequently used operations in OLAP, are special cases of range-sum and range-average, respectively. Techniques for MAX straightforwardly apply to MIN operation. Thus, we have covered most popular aggregation operations supported by SQL [IBM95].

Related Work Following the introduction of the data cube model in [GBLP96], there has been considerable research in the database community on 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].

In the field of computational geometry, there is extensive literature on efficient algorithms for handling various types of range queries (see, e.g., [BF79] [Ben80] [CR89] [Cha90] [Meh84] [Vai85] [WL85] [Yao85]). The range queries are typically defined as follows: given m weighted points in an unbounded d-dimensional integer domain, and a query q represented by a d-dimensional rectangle, apply some aggregation operator to all weighted points contained in q. Most of the results share the following properties: First, the space overhead is mostly *non-linear* in m (e.g. $O(m \log^{d-1} m)$). Second, the index domain of each dimension is assumed to be *unbounded*. Third, mostly the *worst-case* space and time trade-offs are considered. Fourth, most of them do not exploit any properties of the aggregation operation (e.g., the existence of an inverse function with respect to the aggregation operator).

As a contrast, we 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. Finally, our techniques for range-sum queries take advantage of the existence of the inverse operation for SUM. 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.

In an accompanying paper [HBA97], we discuss efficient techniques for partial-sum queries where queries are on arbitrary subsets (not necessarily contiguous) of the categorical attributes. We map the partial-sum problem to the covering problem in the theory of error-correcting codes [CLS86], apply some known covering codes to the problem, and devise a new covering code tailored for this application that offers the best space and time trade-off. Although a range-sum query can be viewed as a special case of the partial-sum query, the techniques specialized for range-sum queries proposed in this paper take advantage of the contiguous ranges of selection and have a better performance.

Paper Organization The remainder of the paper is organized as follows. In Section 2, we give a model for both the range-sum and range-max problems. In Section 3, we present the basic algorithm for range-sum queries based on precomputed prefix sums of the data cube. In Section 4, we generalize the basic algorithm to the blocked algorithms. Section 5 presents an incremental algorithm to handle the data cube updates. We present a range-max algorithm based on tree structures with branch-and-bound-like procedure in Section 6, then give an incremental algorithm for updating the precomputed tree structures in Section 7. A natural range-sum algorithm is to use the same tree structure used

for range-max queries, but without the branch-and-bound optimization. In Section 8, we show that a tree-based range-sum algorithm is inferior to our prefix-sum-based approach. We then give an algorithm for choosing the subset of the data cube dimensions for which the prefix sum is computed and the blocking factor to use for each such subset in Section 9. Section 10 deals with sparse data cubes. Section 11 concludes the results. Proofs of the theorems are given in Appendix.

2 The Model

Let $D = \{1, 2, \cdots, d\}$ denote the set of dimensions, where each dimension corresponds to a functional attribute. We will represent the d-dimensional data cube by a d-dimensional array A of size $n_1 \times n_2 \times \cdots \times n_d$, where $n_j \geq 2$, $j \in D$. We assume an array has a starting index 0. For convenience, we will call each array element a *cell*. Also, let $N = \prod_{j=1}^{d} n_j$ be the total size of array A.

We will describe all range queries with respect to array A. In practice, each dimension of A is the *rank domain* of a corresponding attribute of the data cube. Thus, the attributes of the data cube need not be restricted to the continuous integer domain. However, for performance reasons, it is desirable that there exists a simple function mapping the attribute domain to the rank domain. If such function does not exist, then additional storage and time overhead for lookup tables or hash tables may be required for the mapping.

The problem of computing a range-sum query in a d-dimensional data cube can be formulated as follows:

$$Sum(\ell_1 : h_1, \cdots, \ell_d : h_d) = \sum_{i_1=\ell_1}^{h_1} \cdots \sum_{i_d=\ell_d}^{h_d} A[i_1, \cdots, i_d].$$

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

$$\begin{aligned} &Max\_index(\ell_1 : h_1, \ldots, \ell_d : h_d) = (x_1, \ldots, x_d) \\ &\quad \text{where } (\forall i \in D)(\ell_i \leq x_i \leq h_i) \text{ and} \\ &A[x_1, \ldots, x_d] \\ &\quad = max\{A[y_1, \ldots, y_d] \mid (\forall i \in D)(\ell_i \leq y_i \leq h_i)\ \}. \end{aligned}$$

We will use $Region(\ell_1 : h_1, \ell_2 : h_2, \cdots, \ell_d : h_d)$ to denote a d-dimensional space (region) bounded by $\ell_j \leq i_j \leq h_j$ in dimension j for all $j \in D$. We refer to the *volume* of a region as the number of integer points defined within it. That is, the volume of $Region(\ell_1 : h_1, \ell_2 : h_2, \cdots, \ell_d : h_d)$ is $\prod_{j=1}^{d}(h_j - \ell_j + 1)$.

We will refer to the *volume* of a range query as the volume of the region defining the range query. The range parameters ℓ_j and h_j for all $j \in D$ is specified by the user and typically not known in advance, while the array A is given in advance.

When $d = 1$, we will sometimes drop the subscript 1 of n, ℓ and h. For a range-max query, there may be more than one index with the same maximum value in the specified region. In such case, we assume that the algorithm arbitrarily returns one of the indices with the maximum value in the region.

We will first assume a dense data cube and defer the discussion of sparse data cubes to Section 10.

Array A						
Index	0	1	2	3	4	5
0	3	5	1	2	2	3
1	7	3	2	6	8	2
2	2	4	2	3	3	5

Array P						
Index	0	1	2	3	4	5
0	3	8	9	11	13	16
1	10	18	21	29	39	44
2	12	24	29	40	53	63

Figure 1: Example of the original array A (top) and its prefix-sum array P (bottom).

3 The Basic Range-Sum Algorithm

We first propose a simple method, which needs $N = \prod_{i=1}^{d} n_i$ additional cells to store certain precomputed prefix-sums such that any d-dimensional range-sum can be computed in $2^d - 1$ computation steps, based on up to 2^d appropriate precomputed prefix sums.

3.1 Precomputed Prefix-Sum Array

Let P be a d-dimensional array of size $N = n_1 \times n_2 \times \cdots \times n_d$ (which has the same size as A). P will be used to store various precomputed prefix-sums of A. We will precompute, for all $0 \leq x_j < n_j$ and $j \in D$,

$$\begin{aligned} P[x_1, x_2, \cdots, x_d] &= Sum(0 : x_1,\ 0 : x_2,\ \cdots,\ 0 : x_d) \\ &= \sum_{i_1=0}^{x_1} \sum_{i_2=0}^{x_2} \cdots \sum_{i_d=0}^{x_d} A[i_1, i_2, \cdots, i_d]. \quad (1) \end{aligned}$$

For example, when $d = 2$, we precompute, for all $0 \leq x < n_1$ and $0 \leq y < n_2$,

$$P[x, y] = Sum(0 : x,\ 0 : y) = \sum_{i=0}^{x} \sum_{j=0}^{y} A[i, j].$$

Figure 1 shows an example of $A[x, y]$ and its corresponding $P[x, y]$ for $d = 2$, $n_1 = 6$ and $n_2 = 3$.

3.2 The Basic Range-Sum Algorithm

The theorem below provides how any range-sum of A can be computed from up to 2^d appropriate elements of P. The left hand side of Equation 2 specifies a range-sum of A. The right hand side of Equation 2 consists of 2^d additive terms, each is from an element of P with a sign "+" or "−" defined by product of all $s(i)$'s. For notational convenience, let $P[x_1, x_2, \cdots, x_d] = 0$ if $x_j = -1$ for some $j \in D$.

Theorem 1 *For all $j \in D$, let*

$$s(j) = \begin{cases} 1, & \text{if } x_j = h_j, \\ -1, & \text{if } x_j = \ell_j - 1. \end{cases}$$

Then, for all $j \in D$,

$$\begin{aligned} &Sum(\ell_1 : h_1,\ \ell_2 : h_2, \cdots,\ \ell_d : h_d) \\ &\quad = \sum_{\forall x_j \in \{\ell_j - 1, h_j\}} \left\{ \left(\prod_{i=1}^{d} s(i) \right) * P\left[x_1, x_2, \cdots, x_d\right] \right\} \quad (2) \end{aligned}$$

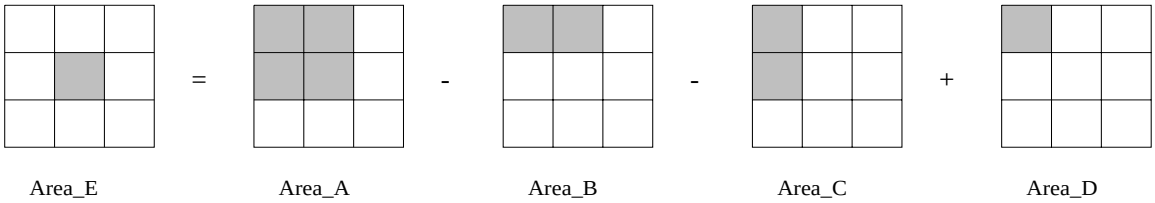

Figure 2: A geometrical explanation for the two-dimensional case: $Sum(Area\_E) = Sum(Area\_A) - Sum(Area\_B) - Sum(Area\_C) + Sum(Area\_D)$.

Example When $d = 2$, the range-sum $Sum(\ell_1 : h_1, \ell_2 : h_2)$ can be obtained in three computation steps as

$$P[h_1, h_2] - P[h_1, \ell_2 - 1] - P[\ell_1 - 1, h_2] + P[\ell_1 - 1, \ell_2 - 1].$$

For instance in Figure 1, the range-sum $Sum(2 : 3, 1 : 2)$ can be derived from

$$P[3, 2] - P[3, 0] - P[1, 2] + P[1, 0] = 40 - 11 - 24 + 8 = 13.$$

Figure 2 gives a geometrical explanation for the two-dimensional case.

When $d = 3$, the range-sum $Sum(\ell_1 : h_1, \ell_2 : h_2, \ell_3 : h_3)$ can be computed in seven steps through

$$P[h_1, h_2, h_3] - P[h_1, h_2, \ell_3 - 1] - P[h_1, \ell_2 - 1, h_3]$$
$$+ P[h_1, \ell_2 - 1, \ell_3 - 1] - P[\ell_1 - 1, h_2, h_3] + P[\ell_1 - 1, h_2, \ell_3 - 1]$$
$$+ P[\ell_1 - 1, \ell_2 - 1, h_3] - P[\ell_1 - 1, \ell_2 - 1, \ell_3 - 1].$$

3.3 Computing the Prefix-Sum Array

Recalling that N is the total size of array P, a naive algorithm to compute P from A may take a time of $O(N^2)$. A better algorithm with a time complexity of $N(2^d - 1)$ steps can be derived based on Theorem 1. We now describe an algorithm with a time complexity of dN steps.

The algorithm has d phases. During the first phase, we perform one-dimensional prefix-sum of A along dimension 1 for all elements of A and store the result in P (denoted P_1). During the i-th phase, for all $2 \leq i \leq d$, we perform one-dimensional prefix-sum of P_{i-1} (the output from previous phase) along dimension i and store the result back to P (denoted P_i). Note that only one copy of P is needed because the same array is reused during each phase of prefix-sum computation. The correctness proof of the algorithm is straightforward because after d phases each $P[y_1, y_2, \cdots, y_d]$ contains the sum of $A[x_1, x_2, \cdots, x_d]$'s for all $0 \leq x_j \leq y_j$, $j \in D$.

On an implementation note, the order of P_i elements visited should follow the natural order in storage as opposed to following the dimension along which the prefix-sum is performed. With such an implementation, each page of P will be paged in at most twice for each phase of the algorithm. Consider Figure 1 as an example and assume arrays are stored in the row-major order. During the first phase, the intermediate array is computed from array A in the order of prefix-sum for first row, second row, etc. During the second phase, the final array P on the right of Figure 1 is computed from the intermediate array. Six prefix sums, one along each column, are required. In order to visit P_1 following the row-major ordering, these six prefix-sum operations are properly interleaved. It is also possible to minimize the number of page-in per phase for each page to one, if array P is properly page-aligned.

Array P						
Index	0	1	2	3	4	5
0	-	-	-	-	-	-
1	-	18	-	29	-	44
2	-	24	-	40	-	63

Figure 3: Example of the blocked prefix-sum array P with $b = 2$.

3.4 Storage Consideration

It is possible to discard array A once P is computed, thus preserving the same total required storage. This is because any element of A can be viewed as a special case of the range-sum: $A[x_1, x_2, \cdots, x_d] = Sum(x_1 : x_1, x_2 : x_2, \cdots, x_d : x_d)$. Thus, whenever $A[x_1, x_2, \cdots, x_d]$ is accessed later, we compute it on the fly from up to 2^d elements of P, based on Theorem 1, as opposed to accessing a single element of A. This offers an interesting space-time trade-off, especially when d is small.

In the next section, we will present a different space-time trade-off by keeping prefix-sums at a coarser grain level and also keeping A.

4 The Blocked Range-Sum Algorithm

4.1 The Blocked Prefix-Sum Array

A simple variation to save space as a trade-off to time is to keep prefix-sums at a coarser-grained (blocked) level. More specifically, we store the prefix-sum only when every index is either one less than some multiple of b or the last index, i.e., $P[i_1, i_2, \cdots, i_d]$, where $(i_j + 1) \bmod b = 0$ or $i_j = n_j - 1$, for all $j \in D$. Thus, for every d-dimensional block of size $b \times b \times \cdots \times b$, only one prefix-sum is precomputed, asymptotically. For example, in the two-dimensional case, only $P[b-1, b-1], P[b-1, 2b-1], \cdots, P[b-1, n_2-1], P[2b-1, b-1], P[2b-1, 2b-1]$, etc., are precomputed, Figure 3.

For clarity, we will refer to this algorithm with $b > 1$ as the *blocked* algorithm and the algorithm of the preceding section with $b = 1$ as the *basic* algorithm. For notational convenience, we still use the same array P and assume that only $P[i_1, i_2, \cdots, i_d]$'s, where "$(i_j - 1) \bmod b = 0$ or $i_j = n_j - 1$ for all $j \in D$, are defined. (In a practical implementation, the sparse array P of size N and density $\approx 1/b^d$ will be packed to a dense array of size about N/b^d.) We discuss the problem of choosing b in Section 9. Note that if the blocked algorithm is used, the original array A cannot be dropped.

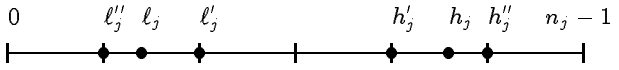

Figure 4: An example for notations $\ell''_j, \ell_j, \ell'_j, h'_j, h_j$ and h''_j.

4.2 The Blocked Algorithm

We now present the *blocked* algorithm for computing the range-sum based on the blocked prefix-sum array P and the original array A, given the dimensionality of the data cube d, the block size parameter b, and the range-sum query $Sum(\ell_1 : h_1, \ell_2 : h_2, \cdots, \ell_d : h_d)$.

First, we need to define some notations. For all $j \in D$, let $\ell''_j = b \lfloor \ell_j / b \rfloor$, $\ell'_j = b \lceil \ell_j / b \rceil$, $h'_j = b \lfloor h_j / b \rfloor$, and $h''_j = \min(b \lceil h_j / b \rceil, n_j)$, Figure 4. Clearly, $\ell''_j \leq \ell_j \leq \ell'_j$ and $h'_j \leq h_j \leq h''_j$. However, $\ell_j < h_j$ does not imply $\ell'_j < h'_j$. In the following, we consider two cases separately: case 1 where $\ell'_j < h'_j$ for all $j \in D$, and case 2 where $\ell'_j \geq h'_j$ for some $j \in D$,

Case 1 To compute $Sum(\ell_1 : h_1, \ell_2 : h_2, \cdots, \ell_d : h_d)$, we will decompose its region into 3^d disjoint sub-regions as follows. Let $R_j = \{\ell_j : \ell'_j - 1, \ell'_j : h'_j - 1, h'_j : h_j\}$ be a set of three adjoining ranges. Then, for all $j \in D$,

$$Sum(\ell_1 : h_1, \cdots, \ell_d : h_d) = \sum_{\forall r_j \in R_j} Sum(r_1, \cdots, r_d).$$

Intuitively, we partition the range in each dimension into three adjoining sub-ranges where the middle sub-range is properly aligned with the block structure. Thus, the Cartesian product of these d dimensions will form 3^d disjoint sub-regions. Note that some of the 3^d regions may be *empty* regions.

For example, Figure 5(a) gives a pictorial example of query $Region(50 : 350, 50 : 350)$ (represented by the shaded area). Figure 5(b) shows the $3^2 = 9$ decomposed regions for this query. The decomposed regions are denoted $A1$ through $A9$.

For convenience, we refer to the region $Region(r_1, r_2, \cdots, r_d)$, for which $r_j = \ell'_j : h'_j - 1$ for all $j \in D$, as an *internal* region. All the other $3^d - 1$ regions are referred to as *boundary* regions. For example, $A5$ in Figure 5(b) is an internal region and all other 8 regions are boundary regions.

To compute the total range-sum for a given range-sum query, we first compute the range-sum for the *internal* region. Since the internal region is properly aligned with the block structure, its range-sum can be computed in up to $2^d - 1$ steps based on up to 2^d appropriate elements of the blocked prefix-sum array P only. What remains to be done is to sum up the range-sums for all $3^d - 1$ boundary regions. To get range-sum for a non-empty boundary region, array A is needed and possibly array P as well.

We first define a few terms. For each boundary region $R = Region(r_1, r_2, \cdots, r_d)$, define its *superblock* region as $Region(B_1, B_2, \cdots, B_d)$, where for all $j \in D$

$$B_j = \begin{cases} \ell''_j : \ell'_j - 1, & \text{if } r_j = \ell_j : \ell'_j - 1, \\ r_j, & \text{if } r_j = \ell'_j : h'_j - 1, \\ h'_j : h''_j - 1, & \text{if } r_j = h'_j : h_j. \end{cases}$$

Then, for each boundary region $Region(r_1, r_2, \cdots, r_d)$, define its *complement* region as $Region(B_1, B_2, \cdots, B_d) - Region(r_1, r_2, \cdots, r_d)$. Intuitively, the *superblock* region of a boundary region R is the smallest region that contains R and is aligned with the block structure. Using the example of boundary regions in Figure 5(b), their respective superblock regions are shown in Figure 5(c) and complement regions shown in Figure 5(d).

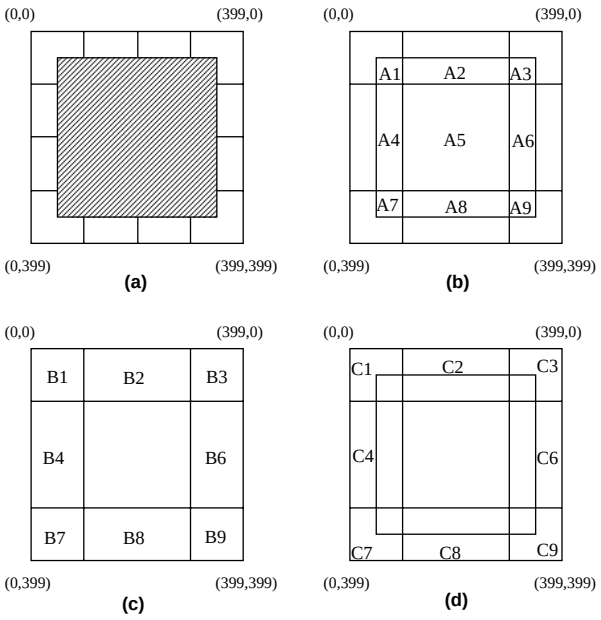

Figure 5: Computing $Sum(50 : 349, 50 : 349)$ where $b = 100$.

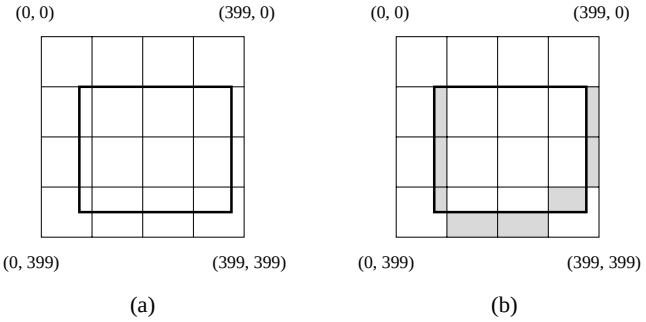

Figure 6: Computing $Sum(75 : 374, 100 : 354)$.

For any boundary region R, there are two possible methods to compute its range-sum. First, one simply sums up all elements of A corresponding to the boundary region R. Second, one can sum up all elements of A corresponding to the *complement* region of R, then subtract the sum from the range-sum for the *superblock* region of R. The range-sum for a superblock region can be computed in $2^d - 1$ steps using P only. To minimize the time complexity, our algorithm will choose the first method when the volume of R is smaller than or equal to "the volume of its complement region plus $2^d - 1$"; and will choose the second method, otherwise. Note that the choice will be made for each boundary region independently. Consider the query $Sum(75 : 374, 100 : 354)$ as shown in Figure 6(a). Different combinations of the first method and the second method will be chosen as denoted by the shaded areas in Figure 6(b).

Case 2 The algorithm description for the case 1 cannot be directly applied to the case 2. A simple modification to the partitioning is as follows. Let $R_j = \{\ell_j : \ell'_j - 1, \ell'_j : h'_j - 1, h'_j : h_j\}$ (as before) if $\ell'_j < h'_j$, and let $R_j = \{\ell_j : h_j\}$ otherwise. Then, for all $j \in D$,

$$Sum(\ell_1 : h_1, \cdots, \ell_d : h_d) = \sum_{\forall r_j \in R_j} Sum(r_1, \cdots, r_d).$$

The definition of *superblock* region is similarly modified as $Region(B_1, B_2, \cdots, B_d)$, where for all $j \in D$

$$B_j = \begin{cases} \ell''_j : \ell'_j - 1, & \text{if } r_j = \ell_j : \ell'_j - 1, \\ r_j, & \text{if } r_j = \ell'_j : h'_j - 1, \\ h'_j : h''_j - 1, & \text{if } r_j = h'_j : h_j, \\ \ell''_j : h''_j - 1, & \text{if } r_j = \ell_j : h_j. \end{cases}$$

With these two modifications, the algorithm description for the case 1 can now be applied to the case 2 as well.

4.3 Computing the Blocked Prefix-Sum Array P

We now describe a two-phase algorithm to compute the blocked prefix-sum array P, given the parameter b. In the first phase, for every d-dimensional block of form $b \times b \times \cdots \times b$, we sum up all elements of A in the block. Thus, the array A is contracted by a factor of b on every dimension during this phase. In the second phase, we apply the algorithm described in Subsection 3.3 to the contracted array of A. It can be seen that the algorithm takes a time no more than $N + dN/b^d = (1 + \epsilon)N$ steps (where $\epsilon = d/b^d$ converges to 0 when b or d increases) and requires no additional buffer space than the blocked prefix-sum array.

5 The Batch-Update Algorithm for Range-Sum Queries

In a typical OLAP environment, updates to data cube are cumulated over a period of time (say, one day) and are performed together as a batch at the end of each period (say, at midnight every day). Thus, it is reasonable to assume a model where $k > 1$ update queries are issued successively before the next read-only query is issued. In this section, we will present an efficient algorithm that batches all the updates together and performs a "combined" update to array P.

5.1 The Batch-Update Algorithm for $b = 1$

Consider first the case when the basic algorithm is used, i.e., $b = 1$. When a query updates an array element $A[x_1, \ldots, x_d]$, all $P[y_1, \cdots, y_d]$'s, where $y_j \geq x_j$ for *all* $j \in D$, need to be updated accordingly. The worst-case complexity of a single update is $O(N)$, which is the total size of array P. This occurs, for instance, when $A[0, \cdots, 0]$ is changed.

For each user update of form (location of an A element, new value), we update the corresponding A element right away and queue an update of a form (location of an A element, *value-to-add*), to be used later for a combined update of P. Here, value-to-add is the new value of the A element subtracting the previous value of it. The inputs to the batch-update algorithm are array P and k queued updates of form (location, value-to-add).

We say an element of P, $P[y_1, \cdots, y_d]$, is *affected* if its value needs to be modified due to any of the k updates, i.e., if there exists an update of $A[x_1, \cdots, x_d]$ where $x_j \leq y_j$ for *all* $j \in D$. We say two elements are in the same *update-class* if they are affected by the same subset of k updates. Clearly, elements in the same update-class have the same combined values-to-add. For example, Figures 7(a) shows the *affected* elements of P, marked by the shaded area, due to an element of A, marked by "1", for a two-dimensional case. Figure 7(b) shows another example. Figure 7(c) shows the combining effect where the affected elements of P can be partitioned into 3 different update-classes, each in a differently shaded region.

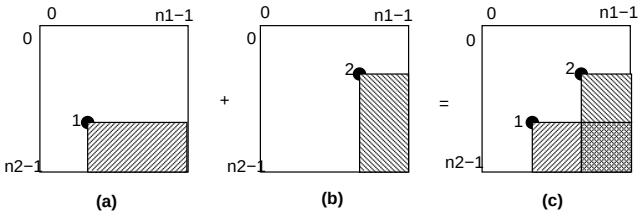

Figure 7: Examples of affected elements, (a) and (b), and the combining effect, (c).

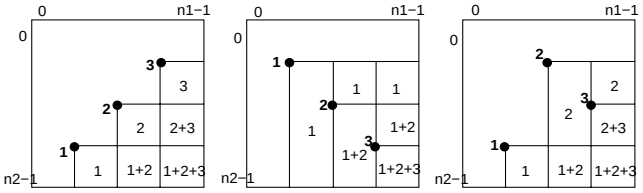

Figure 8: Examples of partitioned regions for $k = 3$ and $d = 2$.

The goal is to group all "affected" elements of array P into minimum number of disjoint regions with the two desirable properties:

Property 1 All elements (of P) in the same region are in the same *update-class*.

Property 1 implies that the same combined values-to-add can be added to all elements of P in the same region. For the purpose of implementation, we add Property 2 below.

Property 2 Each region has a shape of a d-dimensional (rectangular) cube.

Property 2 implies that each region is convex and, hence, can be coded easily. Figure 8 gives some motivating examples for $k = 3$ and $d = 2$, in which all affected elements have been partitioned into 6 regions with the two properties mentioned above.

The Update Algorithm We now give an algorithm for any k and d by recursion on d. For $d = 1$, we sort the indices of k updates in the ascending order and denote the sorted k indices as $u_1, u_2, \cdots, u_k$. Also denote the value-to-add for the i-th update as v_i. We now partition the index space of P, 0 through $n_1 - 1$, into $k + 1$ adjoining regions according to the sorted k indices:

$$Region(0 : u_1 - 1), Region(u_1 : u_2 - 1), \cdots, Region(u_k : n_1 - 1).$$

Refer to these $k+1$ regions as region $0, 1, \cdots, k$, respectively. Clearly, all elements of P in the same region are in the same update-class. Except for region 0, all other regions are affected. Let $n_1 = u_{k+1}$ and $u_0 = 0$. (Note that region i is

empty if $u_i = u_{i+1}$.) We will perform combined updates one region at a time starting from region 1. For all elements of P in region i, $Region(u_i : u_{i+1} - 1)$, where $i \geq 1$, we first combine all necessary updates by computing once the combined values-to-add $V_i = v_1 + v_2 + \cdots + v_i$. (In fact, $V_i = V_{i-1} + v_i$, $i > 1$, is simpler.) Then, add V_i to each $P[y]$ in the range i, i.e., $u_i \leq y \leq u_{i+1} - 1$.

For dimension $d > 1$, we recursively call a number of batch-update algorithms for dimension $d - 1$. First, sort the k update locations according to the index of the first dimension in the ascending order. Denote the k update locations (after sorting by the index of the first dimension) by $(u_1, w_1), \cdots, (u_k, w_k)$, respectively, where each u_i has one index and each w_i contains $d - 1$ indices. Also, denote their corresponding values-to-add by $v_1, \cdots, v_k$, respectively. Next, partition the index space of P into $k + 1$ adjoining regions, according to the sorted k indices $u_1, \cdots, u_k$, as follows:

$$Region(0 : u_1 - 1, R_2, \cdots, R_d), Region(u_1 : u_2 - 1, R_2, \cdots, R_d),$$

$$\cdots, Region(u_k : n_1 - 1, R_2, \cdots, R_d),$$

where $R_i = 0 : n_i - 1$ for all $2 \leq i \leq d$. Each partitioned region is d-dimensional and may contain elements in different update-classes. As before, refer to these $k + 1$ regions as region $0, 1, \cdots, k$, respectively. Also, region 0 is not affected, as before. We will perform combined updates one region at a time starting from region 1. For each region i, $i \geq 1$, we will apply the $(d - 1)$-dimensional batch-update algorithm with inputs as follows: i) there are i updates; ii) the i update locations are $w_1, \cdots w_i$; iii) the i corresponding values-to-add are $v_1, \cdots, v_i$; and iv) the index space of P (now $(d - 1)$-dimensional) is $R_2 \times \cdots \times R_d$ (recall that $R_i = 0 : n_i - 1$ for all $2 \leq i \leq d$).

Then, for every combined values-to-add of $P(y_1, \cdots, y_{d-1})$ generated as output from the $(d-1)$-dimensional algorithm, we apply it, instead, to the value-to-add of $P(x, y_1, \cdots, y_{d-1})$ for all $u_i \leq x \leq u_{i+1} - 1$.

Theorem 2 *The batch-update algorithm will group all affected elements of P into up to $\prod_{j=0}^{d-1} \frac{(n+j)}{d!}$ regions with the two properties described above and perform the k batch-updates correctly.*

5.2 The Batch-Update Algorithm for $b > 1$

We now consider the case when the blocked algorithm is used, i.e., $b > 1$. A simple two-phase algorithm similar to the one in Subsection 4.3 is sufficient. In the first phase, for every d-dimensional block of form $b \times b \times \cdots \times b$, we sum up all values-to-add for all updates of A in the block. Thus, the index space of A has been contracted by a factor of b on every dimension during this phase. In the second phase, we apply the algorithm described in the preceding subsection to the contracted array of A treating each block of A as one element of A, treating the combined values-to-add for each block as a value-to-add for each element of A, and treating the blocked prefix-sum array P as the basic prefix-sum array P.

6 The Range-Max Algorithm

In this section, we present a tree-based algorithm for range-max queries. The data structure we propose 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. We will precompute the index of the maximum value in region $C(x)$ and store it 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.

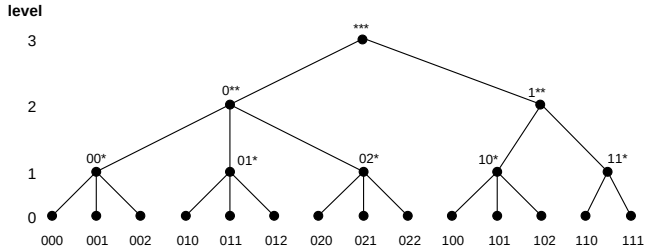

Figure 9: An example of the b-ary tree with $n = 14$ and $b = 3$.

We use a branch-and-bound[Mit70]-like procedure to speed up the finding of max in a query region using this data structure. We will use R to denote the region (or range) of the input query; use Max as a function that takes a region as an argument and returns the maximum value in the region; and use max as the traditional function that returns the maximum value of a set or of all input arguments. For clarity, we first discuss the one-dimensional case, and then generalize to d-dimensions.

6.1 The One-Dimensional Case

6.1.1 Constructing the Tree

We will call the leaves of a tree *level-0 nodes*. A node is said to be at level $i + 1$ if the maximum level of its children is i. We consider trees where 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. When we draw the tree by levels, we place the node with less than b children as the rightmost one within its level. We store the entries of the one-dimensional array A (of size n) as the leaves. Then, we build a b-ary tree in a bottom-up manner and from left to right within each level as follows. We partition A into disjoint ranges, each consisting of b entries, except possibly the last one. For each such range, we 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. We then 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 9 shows an example of $n = 14$ and $b = 3$. Clearly, the root of the tree is at level $\lceil \log_b n \rceil$.

We now describe the addressing scheme of the tree. Let $\delta = \lceil \log_b n \rceil$. We use "||" to denote the concatenation of two strings and use (c^i) to denote the string of i repetitions of the character c. First, all the leaves are encoded as δ-digit base-b strings, starting with (0^δ) on the left, and proceeding to the right. Then, all nodes at level i, $i > 0$, are encoded as δ-digit base-b strings, starting with $(0^{\delta - i}||*^i)$ on left, and proceeding to the right (see Figure 9).

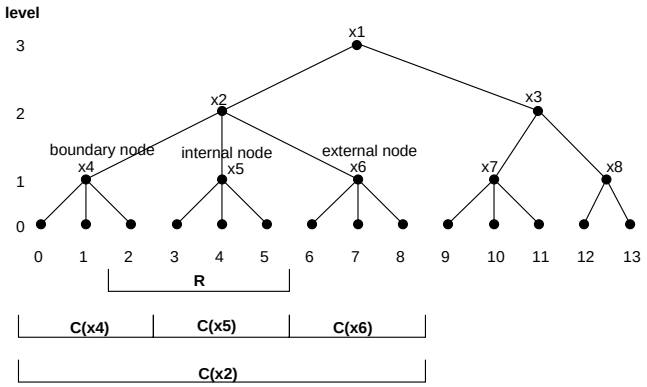

Figure 10: Searching a b-ary tree with $n = 14$ and $b = 3$.

6.1.2 Finding the Lowest-Level Node Covering a Region

Let us denote by $(\ell : h)$ the region of A consisting of all i such that $\ell \leq i \leq h$. The region *covered* by a non-leaf x, denoted $C(x)$, is the region covered by the leaves of the subtree rooted at x. For example, in Figure 10 the region covered by x_2 is $C(x_2) = (0 : 8)$.

Given any range $R = (\ell : h)$, we now derive the lowest-level node covering the region R. This node is used by the algorithm below. Represent both ℓ and h by δ-digit base-b strings. Let $\ell = (u_1 \cdots u_\delta)$, let $h = (v_1 \cdots v_\delta)$, and assume the first w digits of ℓ are the same as that of h (i.e., $u_1 = v_1, \ldots, u_w = v_w$ and $u_{w+1} \neq v_{w+1}$). Then, the lowest-level node $x = (u_1 \cdots u_w || *^{\delta - w})$. Clearly, node x is at level $\delta - w$ and $C(x)$ contains at most $b^{\delta - w}$ nodes. It takes no more than δ comparisons of characters to find w, so this step takes $O(\log n)$ time.

6.1.3 The Search Algorithm

Notation Given a non-leaf node x and a region R, we will categorize each (immediate) child of x, say y, as one of the three types: (1) an *internal* node, if $C(y) \subseteq R$; (2) an *external* node, if $C(y) \cap R = \phi$; and (3) a *boundary* node, otherwise (that is, if $C(y) \cap R \neq \phi$ and $C(y) \not\subseteq R$). Thus, we partition the immediate children into (up to) three sets: (1) $I(x, R)$ contains all the internal nodes, (2) $E(x, R)$ contains all the external nodes, and (3) $B(x, R)$ contains all the boundary nodes. For example, in Figure 10 given a region $R = (2 : 5)$, then $I(x_2, R) = \{x_5\}, E(x_2, R) = \{x_6\}$ and $B(x_2, R) = \{x_4\}$. Note that in the one-dimensional case, for a given x and R, there are at most two boundary nodes, while there can be any number of internal and external nodes.

We further partition the set of boundary nodes $B(x, R)$ into two subsets: (1) $B_{in}(x, R) = \{y | y \in B(x, R), Max\_index(C(y)) \in R\}$ and (2) $B_{out}(x, R) = B(x, R) - B_{in}(x, R)$. Note that whether a node in $B(x, R)$ is in $B_{in}(x, R)$ or in $B_{out}(x, R)$ depends on the data distribution. Use Figure 10 as an example again. Assume $R = (2 : 5)$. If $Max\_index(C(x_4)) = 2$, then $B_{in}(x_2, R) = \{x_4\}$. If $Max\_index(C(x_4)) = 0$ or 1, then $B_{out}(x_2, R) = \{x_4\}$.

The algorithm of $Max\_index(R)$ is expressed in pseudocode in the following.

Function Max_index (R)

```
(1)   let R = (ℓ : h);
(2)   current_max_index = ℓ;
(3)   let x be the lowest-level node such that R ⊆ C(x);
(4)   if Max_index(C(x)) is in the region R
(5)         return (Max_index(C(x)));
(6)   else
(7)         return (get_max_index(x, R, ℓ));
```

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

```
(1)   for all y ∈ I(x, R) ∪ B_in(x, R) do
(2)         if (Max(C(y)) > A[current_max_index])
(3)               current_max_index = Max_index(C(y));
(4)   for all z ∈ B_out(x, R) do
(5)         if (Max(C(z)) > A[current_max_index])
(6)               current_max_index = get_max_index
                  (z, R ∩ C(z), current_max_index);
(7)   return (current_max_index);
```

Explanation of the Algorithm In line (3) of **Max_index**, we find a node x, which is the lowest-level node containing region R, using the method described before. Note that $Max\_index(C(x))$ is the index of the maximum in region $C(x)$ while we are interested in finding the index of the maximum in region R. Thus, if $Max\_index(C(x))$ falls in region R, then the maximum index is found (lines (4) and (5)). Otherwise, the function calls another recursive function **get_max_index** instead.

The recursive function **get_max_index**$(x, R,$ $current\_max\_index)$ takes as inputs the root of a subtree containing R, the region R, 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), we find the Max_index of all regions $C(y)$ for all $y \in I(x, R) \cup B_{in}(x, R)$ and update $current\_max\_index$ whenever necessary. Recall that any node $y \in B_{in}(x, R)$ has the property that $Max\_index(C(y)) \in R$ even though $C(y) \not\subseteq R$. In lines (4) to (6), we find the Max_index of all regions $C(z)$ for all $z \in B_{out}(x, R)$ by recursively calling the same function. The condition at line (5) is used to improve the running time. The idea here is similar to that used in a branch-and-bound algorithm [Mit70]. If a precomputed $Max(C(z))$ is already less than or equal to $A[current\_max\_index]$, then there is no need to find $Max(C(z) \cap R)$ because it is still less than or equal to $A[current\_max\_index]$ and would not affect the result.

The Worst-Case Complexity To analyze the worst-case complexity, denote by $r = h - \ell + 1$ the size of the input range $(\ell : h)$. Clearly, during a call to the function no more than b nodes are accessed directly, and then a recursive call may be made to function on at most two ranges, each of size not greater than r/b, and subsequent calls give rise to only one such range per call. This implies that the total number of nodes accessed is of order $O(b \log_b r)$.

A worst case scenario is that the user-specified region is covered by all the leaves of a complete b-ary subtree except for the first and the last leaves (i.e., $r + 2$ is a power of b), and the first and the last leaves have values larger than any other leaf.

In line (3) of function **Max_index**, we always find the lowest-level node containing the input region R. This is important in guaranteeing that the complexity of our algorithm is bounded from above by $O(b \log_b r)$. Without this feature,

one may have to search from the root of the tree, resulting in complexity of $O(b \log_b n)$ which can be much larger when $r \ll n$.

The Average-Case Complexity We now show that the average-case complexity is in fact bounded from above by $b + 7 + 1/b$, much less than the worst-case bound.

Theorem 3 *The average-case complexity of the tree algorithm is bounded from above by* $b + 7 + \frac{1}{b}$.

6.2 The d-Dimensional Case

Let $B = b^d$. For the d-dimensional case, we will build a B-ary tree (of order $b \times \cdots \times b$) out of the d-dimensional array A in a bottom-up manner as a natural generalization of the 1-dimensional case. More specifically, we will partition the array A into blocks of size of at most $b \times \cdots \times b$ each. Then, all nodes in the same block are connected to a newly added parent node at level 1. Then, iterate the same procedure for all the new parent nodes at level 1 (by viewing them as the array A of order $\lceil n_1/b \rceil \times \cdots \times \lceil n_d/b \rceil$). Since the values of the n_i's may differ, the tree may degenerate into a lower dimension when it grows higher.

The search algorithm described above for the one-dimensional case can be directly applied to the d-dimensional case. The d-dimensional region is specified by $R = (\ell_1 : h_1, \ldots, \ell_d : h_d)$. The definitions of $I(x, R)$, $E(x, R)$, $B_{in}(x, R)$ and $B_{out}(x, R)$ are exactly the same except that the region R is d-dimensional and x now is a data point in the d-dimensional space. Given any region $R = (\ell_1 : h_1, \ldots, \ell_d : h_d)$, we let $r_i = h_i - \ell_i + 1$, $r_{min} = \min_{i \in D}\{r_i\}$, and $r_{max} = \max_{i \in D}\{r_i\}$. Then, the number of nodes that need to be accessed may be reduced from the volume of the region R, depending mostly on r_{min} and r_{max}. In particular, if r_{min} is large relative to b and r_{min} is close to r_{max}, then the savings in time (compared to a method without precomputing) is generally large. Also, if $r_{min} > 2b - 2$ then there always exists a reduction in the effort of accessing the elements of A. Note that the previous one-dimensional bound on the number of nodes accessed, $O(b \log_b r)$, does not apply here.

7 The Batch-Update Algorithms for Range-Max Data Structures

In this section, we describe a batch-update algorithm that takes a list of update points to array A and modify the precomputed information in the tree data structure in addition to modifying A. The input is a list of update points, each of form ⟨index, value⟩. 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 algorithm.

Let $H = \max_{i \in D} \lceil \log_b n_i \rceil$ be the height of the tree. Recall that the i-th level of the tree can be viewed as a contracted array A_i of size $\lceil n_1/b^i \rceil \times \cdots \times \lceil n_d/b^i \rceil$, $0 \le i \le H$. (Thus, A_0 is the original array A.)

We first give a brief overview of the batch-update algorithm. The algorithm has up to H phases (it may terminate earlier). During phase i, where $0 \le i < H$, the input list is scanned once. For each update point on the list read in, the algorithm will (1) update the corresponding value of A_i; (2) when necessary, update some auxiliary data structures associated with A_{i+1}, create an update point on A_{i+1}, and append it to an output list; and (3) modify the output list to a valid input list for the next phase, if the output list is not empty.

The algorithm uses some auxiliary data structures. For each non-leaf node of the tree, we allocates two integer variables *new_max_index* and *tag*, and one variable *max_value* of the same type as elements of A. (Thus, when considering all nodes at level i together, $i > 0$, *new_max_index*, *tag* and *max_value* are all d-dimensional arrays, each having the same number of entries as A_i.) For the purpose of algorithm description, it is sufficient to describe the processing of all update points in one sibling set of leaf nodes, denoted S, with a parent node x at level 1. This sibling set S is organized as a d-dimensional block, of form $b \times \cdots \times b$, covered by x. Let $y' = Max\_index(C(x))$ be the *max_index* of S and $v' = A[y']$.

An update $\langle y, v \rangle$ is an *increase-update* if the new value is larger than the current value ($v > A[y]$), and is a *decrease-update* otherwise. (We ignore an update that does not change the value.) An increase-update $\langle y, v \rangle$ is *active* if $v > v'$, and is *passive* otherwise. A decrease-update $\langle y, v \rangle$ is *active* if $y = y'$, and is *passive* otherwise. (Thus, there will be at most one active decrease-update, since all updates have different indices.)

We will use *new_max_index* to store the index of the *new* maximum value found so far in S due to updates. We will use the value of *tag*, among $\{-1, 0, 1\}$, to represent three different states: $tag = 0$ means no need to update the parent node; $tag = 1$ means the parent node needs to be updated by *new_max_index*; and $tag = -1$ means the value indexed by *new_max_index* has been reduced and the current correct *max_index* cannot be uncovered without searching the whole S.

We now describe the algorithm for phase 0. First, *tag* is initialized to 0, *new_max_index* to *max_index*, and *max_value* to $A[max\_index]$. For each update point $\langle y, v \rangle$ scanned in, we process according to the following rules:

1. If $A[y] < v$ (an *increase-update*):
   (a) Set $A[y] = v$.
   (b) If $v > A[new\_max\_index]$ (active), then set $tag = 1$ and $new\_max\_index = y$.
   (c) If $v = A[new\_max\_index]$ and $tag = -1$, then set $tag = 1$ and $new\_max\_index = y$.
2. If $A[y] > v$ (a *decrease-update*):
   (a) Set $A[y] = v$.
   (b) If $max\_index = y$ (active) and $tag = 0$, then set $tag = -1$.

Passive updates (increase or decrease) will not change the values of y' and v', and will be ignored. For each active increase-updates $\langle y, v \rangle$, we will compare its new value v (which is larger than v') against $A[new\_max\_index]$. If $v > A[new\_max\_index]$, we will update $new\_max\_index = y$ and set $tag = 1$ (from a previous value of possibly 0, 1 or -1).

We now explain rule 2(b). If there is at least one active increase-update before an active decrease-update on the list ($tag = 1$), then the active decrease-update will be ignored. The only time that the *current* maximum index and value information in S may be lost is when there is no active increase-update before the active decrease-update (i.e., $tag = 0$ before processing the active decrease-update). However, any future active increase-update can recover the *current* maximum index and value information in S. If

V	volume of query
x_i	length of the query in dimension i
S	total surface area of query ($= \sum_{i=1}^{d} 2V/x_i$)

Table 1: Query Statistics.

there is no future active increase-update on the input list, then $tag = -1$ when the end of the list is reached. After all update points have been scanned and processed, we need to search the entire set S for which $tag = -1$ to find new_max_index.

8 Tree Hierarchies for Range-Sum Queries

A natural question to ask is whether the tree structures used for range-max queries are good data structures for range-sum queries. 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 the range-max queries, the branch-and-bound optimization cannot be applied to range-sum queries. In this section, we show that with the same storage overhead, a tree-based range-sum algorithm is inferior to the prefix-sum-based algorithm. We develop cost equations to compare the performance of the prefix-sums technique with this hierarchical-tree technique. We use the number of elements required to answer the query as a proxy for response time. The cost equations below assume the query statistics given in Table 1.

Cost Analysis for Prefix Sum Let b be the block size. A block size of 1 corresponds to no blocking. The average cost is approximately

$$2^d + SF(b) \tag{3}$$

where

$$F(b) = \begin{cases} b/4 & \text{if } b \text{ is even;} \\ b/4 - 1/(4b) & \text{if } b \text{ is odd.} \end{cases}$$

$SF(b)$ corresponds to the average number of elements in the superblock region that will have to be accessed to answer the query. Since we can take the complement of the query in the superblock region, $F(b)$ is around $b/4$ rather than $b/2$. Note that this formula gives the right cost for the basic algorithm since $F(1) = 0$. For $b > 1$, we can approximate $F(b)$ to $b/4$.

For very small queries where $V < SF(b)$, this formula is somewhat pessimistic since (1) the 2^d part would disappear as there would not be any complete blocks inside the query and (2) the formula assumes that $F(b)$ is typically greater than x_i. (If $F(b) < x_i$, we can add the V points in the query at less cost than adding $SF(b)$ points.)

Cost Analysis for Hierarchical Trees Consider a tree with a fanout of b in each dimension, for a total fanout of b^d. Let the depth of the tree be t. The intuition behind the cost formula is that at the lowest level of the tree, the number of elements that have to be accessed is the same as for a blocked prefix sum with a block size of b (ignoring the 2^d cost). Since subtraction may be used in the tree algorithm (for a fair comparison), $F(b)$ remains about $b/4$, not $b/2$. At each higher level, the cost is reduced by a factor of b^{d-1} since the "surface area" is reduced by that amount. Hence

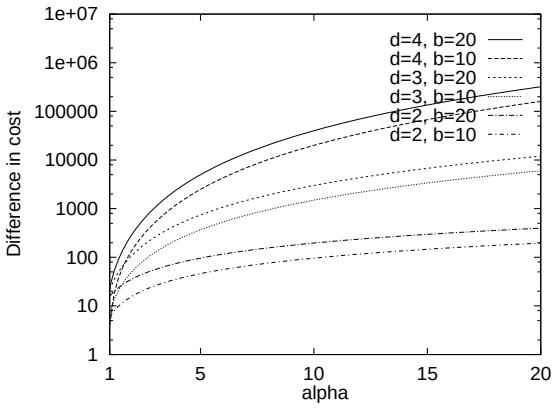

Figure 11: Cost with Hierarchical Tree − Cost with Prefix Sum.

the average cost is approximately

$$F(b) \times \sum_{k=0}^{t-1} \frac{S}{b^{k(d-1)}}.$$

Comparison We assume that the same block size b is used for both prefix sum and the tree. (We are giving space advantage to the tree with this assumption.) Approximating $F(b)$ to $b/4$, the cost for the prefix sum is $2^d + Sb/4$, whereas the cost for the tree is $\sum_{k=0}^{t-1} \frac{Sb/4}{b^{k(d-1)}}$. Hence

Cost with Hierarchical Tree − Cost with Prefix Sum

$$= \sum_{k=1}^{t-1} \frac{Sb/4}{b^{k(d-1)}} - 2^d > \frac{S/4}{b^{d-2}} - 2^d.$$

If we further assume that the queries have a length of αb in each dimension, $S = 2d(\alpha b)^{d-1}$, and the difference in cost becomes $d\alpha^{d-1}b/2 - 2^d$. For queries where αb is significantly greater than the block size the prefix sum is clearly faster. For small queries where αb is less than or roughly equal to the block size, the 2^d factor would usually disappear for the prefix sum since there may not be any complete blocks inside the query. Hence the cost would be comparable for both methods. Figure 11 shows the difference in cost as α changes for different values of b and d.

9 Choosing Dimensions, Cuboids and Block Sizes

In this section, we discuss how to choose the subset of the data cube dimensions for which the auxiliary information is computed and the blocking factor to use for each such subset. While we restrict our discussion to range-sum queries, similar techniques can be applied to range-max queries.

Given a cube on d dimensions, a *cuboid* on k dimensions $\{d_{i_1}, d_{i_2}, ..., d_{i_k}\}$ is defined as a group-by on the dimensions $d_{i_1}, d_{i_2}, ..., d_{i_k}$ [AAD+96]. The cuboid corresponds to the slice of the cube where the remaining $d-k$ dimensions have the value *all*.

1. **Choosing Dimensions.** It may be beneficial to not calculate prefix sums along some attributes of the cube to reduce the average-case time (at the risk of increasing the worst-case time). We discuss the problem of identifying such dimensions in Section 9.1.

2. **Choosing Cuboids.** Having decided to compute the prefix sums along some dimensions, if there is enough space to compute the prefix sum without blocking for the whole cube, we are done. If not, we need to decide for which cuboids the prefix sum should be computed, and with what block sizes. We present a greedy algorithm for this problem in Section 9.2, assuming a "magic formula" that tells us the block size that maximizes the ratio of benefit to space for a given cube. (The benefit is defined as the reduction in the cost of answering queries.)

3. **Choosing Block Size.** We show the derivation of formulae for the block size that maximizes the benefit/space ratio in Section 9.3.

For example, consider a data cube with three dimensions $\langle d_1, d_2, d_3 \rangle$. There are seven possible cuboids (including the cube itself) for which we could compute prefix sums: $\langle d_1, d_2, d_3 \rangle$, $\langle d_1, d_2 \rangle$, $\langle d_1, d_3 \rangle$, $\langle d_2, d_3 \rangle$, $\langle d_1 \rangle$, $\langle d_2 \rangle$, and $\langle d_3 \rangle$. We may first decide that all the queries on dimension d_3 do not involve ranges and hence even for cuboids that include dimension d_3, the prefix sum would only be computed on other dimensions. Next, we may decide to compute a prefix sum on $\langle d_1, d_2, d_3 \rangle$ with a block size of 10 and another prefix sum on $\langle d_1, d_2 \rangle$ with a block size of 1.

If one cuboid has a subset of the dimensions of another cuboid, we call the former a *descendant* of the latter, and the latter an *ancestor* of the former. For example, $\langle d_1, d_3 \rangle$ is a descendant of $\langle d_1, d_2, d_3 \rangle$ and an ancestor of $\langle d_3 \rangle$. Note that if we compute a prefix sum on some cuboid with block size b, there is no benefit to computing a prefix sum on any of its descendants unless the block size is smaller.

We assume that we are given either a query log, or statistics which capture the average query statistics for each cuboid as well as the number of queries (denoted by N_Q). Queries with ranges on dimensions d_1 and d_2 and *all* on dimension d_3 will be assigned to the cuboid $\langle d_1, d_2 \rangle$, and so on. In this section, we use the notation in Table 1 to denote the average rather than the numbers for a single query.

9.1 Choosing Dimensions

First, we assume that the basic algorithm is used (i.e., $b = 1$). Depending on the queries, it may be beneficial with respect to performance to choose a subset of the dimensions upon which P is defined. Consider an example with $d = 3$ dimensions, labeled d_1, d_2, d_3. If all range-sum queries specify only one index with respect to dimension d_3, then the prefix-sum structure along dimension d_3 is not needed. Hence for the cuboid $\langle d_1, d_2, d_3 \rangle$, the prefix sum will only be calculated along dimensions d_1 and d_2. As a result, only $2^2 - 1 = 3$ steps is required for each range-sum query as opposed to $2^3 - 1 = 7$ steps.

We now address the performance issues in choosing a proper subset of data cube dimensions. Let X be the set of d dimensions of the original data cube. Thus, $|X| = d$. We will choose a subset X' of X, where $|X'| = d'$. We say that an attribute $a \in X$ is *active* with respect to a range-sum query q if the selection upon the attribute is a contiguous range in its domain and the range is neither a singleton (i.e., a single value in the domain) nor *all*. An attribute is *passive* with respect to a range-sum query if it is not active. Given a query q, the best way of choosing X' among X is clearly to let X' contain exactly all *active* attributes with respect to q. However, given a set of collected queries from the OLAP log, $Q = \{q_1, q_2, \cdots, q_m\}$, the best way of choosing X' among X that minimizes the overall query time complexity with respect to Q is an optimization problem.

Attribute	1	2	3	4	5
q_1	1	100	1	3	1
q_2	200	1	100	1	1
q_3	500	500	1	1	1
R_j	701	601	102	5	3

Figure 12: Example of the heuristic algorithm in choosing $X' = \{1, 2, 3\}$.

First, we have a few observations (and the details are omitted here). A simple lower bound for the time complexity of any algorithm that gives an optimal solution is $O(md)$. A naive algorithm that gives an optimal solution requires $O(md2^d)$ time: there are 2^d different choices for X' and each choice requires $O(md)$ time to evaluate the cost. We now sketch an $O(m2^d)$-time algorithm that gives an optimal solution. It is possible to arrange the cost evaluations of the 2^d choices of X' in an order such that any two adjacent choices only differ in one attribute. (Such an ordering can be easily derived from the binary-reflected Gray code [RND77].) With such an ordering, the cost evaluation of each choice can be derived from its preceding choice in $O(m)$ steps, with the exception of the first choice whose cost can be derived in $O(md)$ steps.

Finding an algorithm that gives an optimal solution and requires a time less than $O(m2^d)$ is an open problem. In the following, we give a very simple heuristic algorithm of time complexity $O(md)$. Let $X = \{d_1, d_2, \cdots, d_d\}$. With respect to query q_i and attribute d_j, let r_{ij} be the length of its range if the attribute is active, and be 1 if the attribute is passive. (Recall that for a passive attribute, its range is either of length 1 or is *all*). Then, Let $R_j = \sum_{i=1}^{m} r_{ij}$ for all $j \in D$. We will then define

$$X' = \{d_j | R_j \geq 2m\}.$$

Figure 12 shows an example of this.

The intuition behind the heuristic algorithm is as follows. For a given query q_i and given attribute d_j, if $d_j \in X'$ then the time complexity factor for the query q_i contributed by attribute d_j is 2; otherwise it is r_{ij}. Note that this is a multiplicative factor. The heuristic algorithm is derived from a simplified assumption: the time complexity factor contributed by the product of all other attributes are the same for all queries.

9.2 Choosing Cuboids

We define the benefit of a particular solution to be the reduction in the cost of answering all the queries. Given a fixed amount of space in which to store all the prefix sums, the problem is to find the set of prefix sums (and block sizes) that maximize the benefit. Since this problem is NP-complete (reduction from Set-Cover), we use heuristics to get approximate solutions.

This problem is similar to the problem of deciding which cuboids to materialize considered in [HRU96]. However, the latter problem is somewhat simpler since the choice for each cuboid is whether or not to materialize. In our case, in addition to determining whether or not to compute the prefix sum for a cuboid, we need to determine what the block size should be in each dimension.

```
// Greedy Algorithm
Ans = φ;
while space < limit do begin
   foreach cuboid ∉ Ans do
      Compute block sizes for best ratio of benefit to space.
   Add cuboid that resulted in the best
      benefit/space ratio to Ans
end

// Fine-tuning the output of the Greedy Algorithm
do
   foreach cuboid X ∈ Ans do
      Drop an X from Ans.
      foreach cuboid ∉ V do
         Compute block sizes for best ratio of benefit to space.
      Add cuboid that resulted in the best
         benefit/space ratio to V.
   end
until no improvement.
```

Figure 13: Greedy Algorithm.

We present a greedy algorithm for choosing cuboids and block sizes in Figure 13. The first half of the algorithm is a simple greedy search, similar to the algorithm in [HRU96]. We illustrate the intuition behind the rest of the algorithm with an example. Assume the greedy search first computed prefix sum with block sizes b for the cuboid $\langle d_1, d_2\rangle$. If the greedy search later computes the prefix sum for the cuboid $\langle d_1\rangle$ with block size $b' < b$, some other block sizes b'' may give a better benefit/space ratio for $\langle d_1, d_2\rangle$. In fact some other cuboid, say $\langle d_2, d_3\rangle$ may now have a better benefit/space ratio than $\langle d_1, d_2\rangle$ since the prefix sum for $\langle d_1, d_2\rangle$ no longer provides any benefit to the cuboid $\langle d_1\rangle$.

9.3 Choosing Block Sizes

We refer to the cuboid for which we are currently trying to find the maxima of the benefit/space function as the *current* cuboid. The problem is to find the block size that results in the maximum value of benefit/space. We initially ignore the benefit for the descendant cuboids of computing the prefix sum for the current cuboid.

Let the current cuboid have d dimensions and N cells. We use the cost formulae from Section 8, but split the cases for $b = 1$ (no blocking) and $b > 1$ in order to get a smooth function for the cost. We compute the benefit/space ratio for $b = 1$ as well as for the maxima of the benefit/space function for $b > 1$ and choose whichever has a higher benefit/space ratio. For $b > 1$, we approximate $F(b)$ to $b/4$.

$$\begin{aligned}
\text{Cost without prefix sum} &= N_Q V \\
\text{Cost with prefix sum} &= N_Q(2^d + Sb/4) \\
\text{Benefit} &= N_Q(V - 2^d - Sb/4) \\
\text{Space required} &= N/b^d \\
\text{Benefit/Space} &= N_Q(V - 2^d - Sb/4)\,/\,(N/b^d) \\
&= (N_Q/N) \\
&\quad \times [(V - 2^d)b^d - (S/4)b^{d+1}]
\end{aligned}$$

If $V - 2^d \leq 0$, there is no benefit to computing the prefix sum with or without blocking. If $V - 2^d \leq S/4$, there is no benefit to computing the prefix sum with blocking. Hence we only consider the case where $V - 2^d > S/4$ when trying to find the maxima of the benefit/space function.

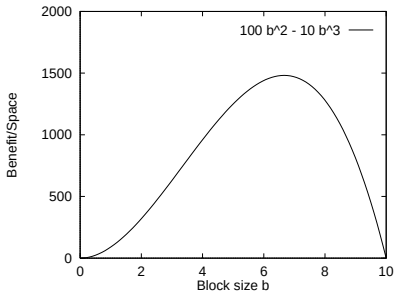

Figure 14: Behavior of benefit/space w.r.t. block size

For example, for $d = 3$, $N_Q/N = 1/100$, $V - 2^d = 1000$ and $S = 400$, the benefit/space graph would look as shown in Figure 14. When $b = 4(V - 2^d)/S$, the benefit becomes 0. This corresponds to block sizes that are so high (relative to query sizes) that there is no advantage to computing the prefix sum.

Differentiating w.r.t. b to find the maxima, we get

$$(N_Q/N) \times [(V - 2^d)db^{d-1} - (S/4)(d+1)b^d] = 0.$$

The solutions are $b = 0$ (corresponding to infinite space) and

$$b = \frac{V - 2^d}{S/4} \times \frac{d}{d+1}.$$

To check that this is a maxima and not a minima, we compute the second differential,

$$(N_Q/N) \times [(V - 2^d)d(d-1)b^{d-2} - (S/4)(d+1)db^{d-1}].$$

The second differential will be negative if

$$b > \frac{V - 2^d}{S/4} \times \frac{d-1}{d+1}.$$

Hence when $b = \frac{V-2^d}{S/4} \times \frac{d}{d+1}$, the second differential is negative and the inflection is a maxima. The value of b obtained from this equation may not be an integer. If so, we should compute the benefit/space ratio for each of the two integers that bound b and choose the larger value.

Incorporating the Effect of Prefix Sums on Ancestor Cuboids If any of the ancestors of the current cuboid have a prefix sum with block size b', there is no benefit to computing a prefix sum for the current cuboid with any block size $b \geq b'$. The benefit function becomes

$$N_Q(2^d + Sb'/4) - N_Q(2^d + Sb/4) = N_Q(S/4)(b' - b)$$

if $b < b'$ and 0 if $b \geq b'$. The maxima of the benefit/space function is $b = b'd/(d+1)$.

Incorporating the Benefit for Descendant Cuboids Computing a prefix sum on a cuboid also benefits its descendant cuboids. For example, a prefix sum $\langle d_1, d_2\rangle$ also helps the cuboids $\langle d_1\rangle$ and $\langle d_2\rangle$. Hence, when computing the benefit function, rather than simply considering the benefit for $\langle d_1, d_2\rangle$, we also need to add the benefits for its descendant cuboids $\langle d_1\rangle$ and $\langle d_2\rangle$ to the benefit for $\langle d_1, d_2\rangle$.

If some descendant cuboid D has a prefix sum with block size b' or an ancestor with a prefix sums with block size b', the benefit function for the current cuboid will be different for $b < b'$ and $b \geq b'$. For $b < b'$, there will be additional benefit for D, while the benefit for D will be 0 for $b \geq b'$. Hence the total benefit function for the current cuboid will be of the form

$$\begin{cases} C_1 - C_2 b & \text{if } b < b', \\ C_3 - C_4 b & \text{if } b \geq b'. \end{cases}$$

where $C1, ..., C_4$ are constants. Thus we differentiate both functions to find the maxima, and choose whichever results in a better benefit/space. In general, if n of the descendant cuboids have prefix sums (either on that cuboid or on an ancestor of the cuboid), we will have $n+1$ different benefit functions for the current cuboid, and we need to find the maxima for each function.

10 Sparse Data Cubes

If the data cube is uniformly sparse, computing a blocked prefix sum with an appropriate block size b and storing it as a dense array solves the problem. We now discuss solutions for cubes which are not uniformly sparse: a special-case solution for range-sum queries with one-dimensional cubes, and a general solution for both range-sum and range-max queries with d-dimensional cubes.

10.1 The One-Dimensional Case for Range-Sum Queries

When $b = 1$, the prefix-sum array P has the same sparse structure as the one-dimensional data cube A. Given a range $(\ell : h)$, we only need to find out the first non-zero $P[\hat{\ell}]$ where $\hat{\ell} \leq \ell$ and the first non-zero $P[\hat{h}]$ where $\hat{h} \leq h$. We can build a B-tree index [Com79] on P to find $\hat{h}$ and $\hat{l}$. A similar solution applies to the case where $b > 1$.

10.2 The d-Dimensional Case for Range-Sum Queries

We first find a set of non-intersecting rectangular dense regions. One approach to finding these regions would be to use clustering (e.g. [JD88].) However, clustering algorithms typically generate centers of clusters and require post-processing to generate rectangular regions. Some clustering algorithms in image analysis (e.g. [BR91] [SB95]) find rectangular dense regions, but are designed for two-dimensional datasets. Hence we use a modified decision-tree classifier [SAM96] to find dense regions (non-empty cells are considered one class and empty cells another). The modification to the classifier is that the number of empty cells in a region are counted by subtracting the number of non-empty cells from the volume of the region. This lets the classifier avoid materializing the full data cube.

Once the dense regions have been found, we compute the prefix sum (or a blocked prefix sum) for each dense region. The boundary of each dense region is added to an R* tree [BKSS90], along with a pointer to the dense region. All points not in a dense region are also added to the R* tree.

Given a range-sum query, we find all dense regions that intersect the query by searching the R* tree. For each such region, we use the prefix sum to get the sum of the elements in the intersection of the dense region and the query. Finally, we add the sum of all points in the R* tree that lie within the query region to get the range sum.

10.3 The d-Dimensional Case for Range-Max Queries

For range-max queries, we can replace the static fixed-fanout tree structure by any other tree structure without affecting the correctness of the algorithm. In the tree, a rectangular region, represented by a tree node, may be split into a number of intersecting regions, each represented by a child node. Thus, the R* tree [BKSS90] is a good data structure in the sparse data cube. Note that in this case where a dynamic tree is used, one needs to traverse starting from the root because the lowest-level node covering the query region cannot be located in constant time.

11 Conclusion

In this paper, we have presented fast algorithms for computing range-sum and range-max queries on a data cube in an OLAP system. The main idea for speeding up range-sum queries is to precompute multidimensional prefix-sums of the data cube. Then, any range-sum query can be answered by accessing 2^d appropriate prefix-sums. The total storage requirement can be kept to be the same as the data cube with a slight increase in time for queries of a singleton cell, because any cell of the data cube can be computed with the same time complexity as a range-sum query.

Alternatively, one can trade time for space by precomputing and storing the multidimensional prefix-sums only at a block level, so that the blocked array P can fit in memory. In this case, any range-sum query can be answered by accessing block-level 2^d prefix-sums as well as some cells of the data cube. The overall query response time is still significantly better than that without any precomputed information or with precomputed tree hierarchies. Indeed, our prototype implementation confirmed the benefit of these techniques, with the advantage increasing as the volume of the circumscribed query sub-cube increases.

For range-max queries, we construct a generalized quad-tree on the data cube and store in each tree node the index of the maximum value in the region covered by that node. We then use a branch-and-bound[Mit70]-like procedure to speed up the queries. We show that with a branch-and-bound procedure, the average-case complexity is much smaller than the worst-case complexity. We also give a simple incremental algorithm to handle the data cube update.

For some datasets, it may be beneficial to not calculate prefix sums along some dimensions of the cube to reduce the average-case time. We presented an algorithm for identifying such dimensions. Given a fixed amount of space, we presented an algorithm for choosing for which cuboids prefix sums should be calculated and with what block sizes in order to minimize the total response time. Similar optimization techniques can be applied to range-max queries.

In a real OLAP environment, users typically search for trends, patterns, or unusual data behavior by issuing queries interactively. Thus, users may be satisfied with an approximate answer for a query if the response time can be greatly reduced. As an off-shoot of the range-sum block algorithm, one can implement the range-sum algorithm so that an upper bound and a lower bound on the range-sum are returned to users first, followed by a real sum when the final computation is completed. This is because each bound can be derived in at most $2^d - 1$ computation steps. The same approximation approach can be applied to the range-max queries using the tree algorithm.

Appendix: Proofs of Theorems

Theorem 1 For all $j \in D$, let

$$s(j) = \begin{cases} 1, & \text{if } x_j = h_j, \\ -1, & \text{if } x_j = \ell_j. \end{cases}$$

Then, for all $j \in D$,

$$Sum(\ell_1 : h_1,\ \ell_2 : h_2, \cdots,\ \ell_d : h_d) = \sum_{\forall x_j \in \{\ell_j - 1, h_j\}} \left\{ \left(\prod_{i=1}^{d} s(i) \right) * P\left[x_1, x_2, \cdots, x_d\right] \right\} .$$

Proof: We assume that d, n_j, $j \in D$, and A are all given as input. Also, assume $P[x_1, x_2, \cdots, x_d]$'s as defined in Equation 1 have been precomputed for all $1 \leq x_j \leq n_j$ and $j \in D$. We will prove the theorem by proving the following equation instead, for all $t \in D$, by induction:

$$Sum(\ell_1 : h_1,\ \cdots,\ \ell_t : h_t,\ 0 : x_{t+1},\ \cdots,\ 0 : x_d) = \sum_{\forall x_j \in \{\ell_j, h_j\}, 1 \leq j \leq t} \left\{ \left(\prod_{i=1}^{t} s(i) \right) * P[x_1, \cdots, x_d] \right\} \quad (4)$$

(This theorem is then a special case of this equation by letting $t = d$.) Note that the range of index j is $1 \leq j \leq t$, that is there are only 2^t terms of $P[x_1, x_2, \cdots, x_d]$ (as opposed to 2^d such terms).

For the basis $t = 1$, it can be shown from the definition of P. Assume, for the sake of induction hypothesis, that Equation 4 holds for $t = k$ for some k where $1 \leq k < d$. That is, we have

$$Sum(\ell_1 : h_1,\ \cdots,\ \ell_k : h_k,\ 0 : x_{k+1},\ \cdots,\ 0 : x_d) = \sum_{\forall x_j \in \{\ell_j, h_j\}, 1 \leq j \leq k} \left\{ \left(\prod_{i=1}^{k} s(i) \right) * P[x_1, \cdots, x_d] \right\} \quad (5)$$

We wish to show that Equation 4 still holds for $t = k + 1$.

By letting $x_{k+1} = \ell_{k+1}$ in Equation 5, we have

$$Sum(\ell_1 : h_1, \cdots, \ell_k : h_k, 0 : \ell_{k+1}, 0 : x_{k+2}, \cdots, 0 : x_d) \quad (6)$$

$$= \sum_{\forall x_j \in \{\ell_j, h_j\}, 1 \leq j \leq k} \left\{ \left(\prod_{i=1}^{k} s(i) \right) * P[x_1, \cdots, x_k, \ell_{k+1}, x_{k+2}, \cdots, x_d] \right\} . \quad (7)$$

Similarly, by letting $x_{k+1} = h_{k+1}$ in Equation 5, we have

$$Sum(\ell_1 : h_1, \cdots, \ell_k : h_k, 0 : h_{k+1}, 0 : x_{k+2}, \cdots, 0 : x_d) \quad (8)$$

$$= \sum_{\forall x_j \in \{\ell_j, h_j\}, 1 \leq j \leq k} \left\{ \left(\prod_{i=1}^{k} s(i) \right) * P[x_1, \cdots, x_k, h_{k+1}, x_{k+2}, \cdots, x_d] \right\} . \quad (9)$$

For convenience, denote the terms in Equations 6 through 9 by T_1, T_2, T_3 and T_4, respectively. Notice that

$$Sum(\ell_1 : h_1, \cdots, \ell_k : h_k, \ell_{k+1} : h_{k+1}, 0 : x_{k+2}, \cdots, 0 : x_d) = T_3 - T1 = T_4 - T_2 = \sum_{\forall x_j \in \{\ell_j, h_j\}, 1 \leq j \leq k+1} \left\{ \left(\prod_{i=1}^{k+1} s(i) \right) * P[x_1, x_2, \cdots, x_d] \right\} .$$

That is, we have shown Equation 4 holds for $t = k+1$. This completes the proof of the theorem. □

Theorem 2 The batch-update algorithm will group all affected elements of P into up to $\prod_{j=0}^{d-1} \frac{(n+j)}{d!}$ regions with the two properties described above and perform the k batch-updates correctly.

Proof: We first prove, by induction on d, that the algorithm is correct. For $d = 1$, the correctness can be observed trivially. Assume, for the sake of induction hypothesis, that the algorithm is correct for $d - 1$ dimensions. We wish to show that the batch-update algorithm for d dimensions is still correct. Consider any two elements that are in the same region (partitioned from the d-dimensional index space of P) and only differ in the first index. Observe that they are in the same update-class, because there is no update locations between these two elements. This means that for region i we can first concentrate on the update-class for the $(d-1)$-dimensional index space of P with the first index being u_i, then apply the same update along dimension 1 within this region. To solve the update-class for the $(d-1)$-dimensional index space of P with the first index being u_i, one simply projects, along dimension 1, all locations that have the first index smaller than or equal to u_i. Thus, locations $(u_1, w_1), (u_2, w_2), \cdots, (u_i, w_i)$ are projected into locations $w_1, w_2, \cdots, w_i$ and a correct batch-update algorithm for $d-1$ dimensions is applied. This completes the proof of the correctness part.

We now derive the number of partitioned regions covering all affected elements of P at the lowest level of the recursion. (At this level, all elements in the same region are in the same update-class.) Let $NR(k, d)$ be the maximum number of lowest-level regions partitioned from the affected elements of P by the algorithm for k updates and d dimensions. Then, the following recursion is easily derived from the recursive description of the algorithm:

$$NR(k, d) = \sum_{i=1}^{k} NR(i, d-1)$$

and $NR(k, 1) = k$. Through some exercise, one can derive

$$NR(k, d) = \frac{\prod_{i=0}^{d-1} (k+i)}{d!} .$$

As motivating examples for the derived close form for $NR(k, d)$ above, we have

$$NR(k, 2) = \sum_{i=1}^{k} NR(i, 1) = 1 + 2 + \cdots + k = \frac{k(k+1)}{2} .$$

$$NR(k, 3) = \sum_{i=1}^{k} NR(i, 2) = \frac{1 \cdot 2}{2} + \frac{2 \cdot 3}{2} + \cdots + \frac{k(k+1)}{2} = \frac{k(k+1)(k+2)}{6} .$$

□

Theorem 3 The average-case complexity of the tree algorithm is bounded from above by $b + 7 + \frac{1}{b}$.

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 $R_i = [(i-1)b^k, ib^k - 1]$, $i = 1, 2, \ldots, b$. 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 $R_1, \ldots, R_{d_k}$ as well as the one over the range R_{d_k+1}. If the maximum over the latter is no larger than the maximum over $R_1 \cup \cdots \cup R_{d_k}$, 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}.[1] 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 is no larger than the maximum over $R_1 \cup \cdots \cup R_{j-1}$. By definition, $E_j \cap F_j = E_{j+1} \cap F_j$. Thus,

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

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

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

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

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

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

$$\begin{aligned}F(h) &< \max_x \left\{1 + \left(1 - \frac{x}{b}\right)(x+1)\right\} + \frac{1}{2} F(h - d_k b^k) \\ &= \frac{1}{4} b + 1.5 + \frac{1}{4b} + \frac{1}{2} F(h - d_k b^k)\end{aligned}$$

(where $x = (b-1)/2$ gives the maximum) and this inequality implies, by induction, that for all h,

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

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 one index at the root, and with probability $1 - \frac{r}{b^{k+1}}$ we will have to access the roots of x internal subtrees in addition to, on the average, at most $b + 6 + \frac{1}{b}$ more nodes in the two boundary problems.

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

$$\begin{aligned}h(r) &= 1 + \left(1 - \frac{r}{b^{k+1}}\right)\left(x + b + 6 + \frac{1}{b}\right) \\ &\leq 1 + \left(1 - \frac{x}{b}\right)\left(x + b + 6 + \frac{1}{b}\right) \ .\end{aligned}$$

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

[1] 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.

Acknowledgment We wish to thank Prabhakar Raghavan and Sunita Sarawagi for their helpful comments.

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.

[BKSS90] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R^*-tree: an efficient and robust access method for points and rectangles. In *Proc. of the ACM SIGMOD Conference on Management of Data*, pages 322–331, Atlantic City, NJ, May 1990.

[BR91] M. Berger and I. Regoutsos. An algorithm for point clustering and grid generation. *IEEE Transactions on Systems, Man and Cybernetics*, 21(5):1278–86, 1991.

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

[CLS86] G.D. Cohen, A.C. Lobstein, and N.J.A. Sloane. Further results on the covering radius of codes. *IEEE Trans. Information Theory*, IT-32(5):680–694, September 1986.

[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.

[Cod93] E. F. Codd. Providing OLAP (on-line analytical processing) to user-analysts: An IT mandate. Technical report, E. F. Codd and Associates, 1993.

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

[Com79] D. Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–138, June 1979.

[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 subtotals. 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.

[HBA97] Ching-Tien Ho, Jehoshua Bruck, and Rakesh Agrawal. Partial-sum queries in OLAP data cubes using covering codes. In *Proc. of the 16th ACM Symposium on Principles of Database Systems*, 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.

[IBM95] IBM. *DB2 SQL Reference for Common Servers Version 2*, 1995.

[JD88] A. K. Jain and R. C. Dubes. *Algorithms for Clustering Data*. Prentice Hall, 1988.

[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.

[RND77] E.M. Reingold, J. Nievergelt, and N. Deo. *Combinatorial Algorithms*. Prentice-Hall, Englewood Cliffs. NJ, 1977.

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

[SAM96] John Shafer, Rakesh Agrawal, and Manish Mehta. SPRINT: A scalable parallel classifier for data mining. In *Proc. of the 22nd Int'l Conference on Very Large Databases*, Bombay, India, September 1996.

[SB95] P. Schroeter and J. Bigun. Hierarchical image segmentation by multi-dimensional clustering and orientation-adaptive boundary refinement. *Pattern Recognition*, 25(5):695–709, May 1995.

[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.