Hierarchical Cubes for Range-Sum Queries

Chee-Yong Chan **Yannis E. Ioannidis**[*] [†]

Department of Computer Sciences
University of Wisconsin-Madison
{cychan,yannis}@cs.wisc.edu

Abstract

A range-sum query sums over all selected cells of an OLAP data cube where the selection is specified by ranges of contiguous values for each dimension. An efficient approach to process such queries is to precompute a prefix cube (PC), which is a cube of the same dimensionality and size as the original data cube but with a prefix range-sum stored in each cell. Using a PC, any range-sum query can be evaluated at a cost that is independent of the size of the sub-cube circumscribed by the query. However, a drawback of PCs is that they are very costly to maintain. Recently, a variant of prefix cubes called Relative Prefix Cubes (RPC) has been proposed to alleviate this problem.

In this paper, we propose a new class of cube representations called Hierarchical Cubes, which is based on a design framework defined by two orthogonal dimensions. Our results show that a particular cube design called the Hierarchical Band Cube (HBC) is the overall winner: it not only has a significantly better query-update tradeoff than previous approaches, but it can also be more effectively buffered.

[*]Partially supported by the National Science Foundation under Grant IRI-9157368 (PYI Award) and the members of the Wisconsin database group industrial affiliates program (http://www.cs.wisc.edu/~raghu/dbaffiliates.html).

[†]Author's present address: Department of Informatics, University of Athens, Hellas (Greece).

Proceedings of the 25th VLDB Conference, Edinburgh, Scotland, 1999.

1 Introduction

Aggregation is a common and computation-intensive operation in on-line analytical processing systems (OLAP), where the data is usually modeled as a multidimensional data cube, and queries typically involve aggregations across various cube dimensions. Conceptually, an n-dimensional data cube is derived from a projection of $(n+1)$ attributes from some relation R, where one of these attributes is classified as a *measure attribute* and the remaining n attributes are classified as *dimensional attributes.* Each dimension of the data cube corresponds to a dimensional attribute, and the value in each cube cell is an aggregation of the measure attribute value of all records in R having the same dimensional attribute values.

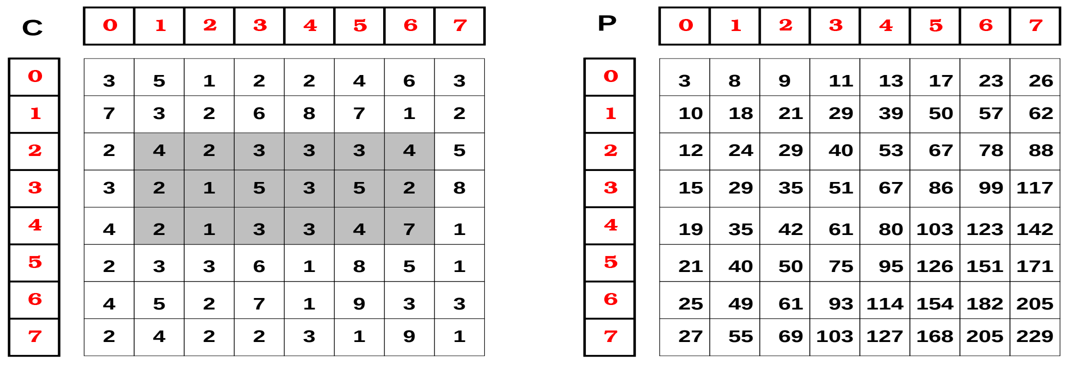
Various forms of precomputation techniques [5, 8, 13] and indexing methods [3, 6, 7, 9, 10, 11, 12] have been proposed to expedite processing of OLAP queries. In this paper, we propose a new precomputation technique for a class of OLAP queries called *range-sum queries.*

A **range-sum query** sums over all selected cells of an OLAP data cube where the selection is specified by ranges of contiguous values for each dimension. An example of a range-sum query over a data cube C with schema $(A_1, A_2, \ldots, A_n, M)$ is as follows:

SELECT	**SUM** $(C.M)$
FROM	C
WHERE	$l_1 \leq C.A_1 \leq h_1$
and	$l_2 \leq C.A_2 \leq h_2$
and	....
and	$l_n \leq C.A_n \leq h_n$

We refer to range-sum queries with $l_i = 0$ for $1 \leq i \leq n$ as **prefix range-sum queries**.

The most direct approach to evaluate a range-sum query is to use the data cube itself, but the disadvantage of this approach is that the number of cells that need to be accessed is proportional to the size of the sub-cube defined by the query. Recently, a more efficient approach to compute range-sum queries was proposed using a **prefix cube** (PC) [5] which costs



C	0	1	2	3	4	5	6	7
0	3	5	1	2	2	4	6	3
1	7	3	2	6	8	7	1	2
2	2	4	2	3	3	3	4	5
3	3	2	1	5	3	5	2	8
4	4	2	1	3	3	4	7	1
5	2	3	3	6	1	8	5	1
6	4	5	2	7	1	9	3	3
7	2	4	2	2	3	1	9	1

(a) Data Cube

P	0	1	2	3	4	5	6	7
0	3	8	9	11	13	17	23	26
1	10	18	21	29	39	50	57	62
2	12	24	29	40	53	67	78	88
3	15	29	35	51	67	86	99	117
4	19	35	42	61	80	103	123	142
5	21	40	50	75	95	126	151	171
6	25	49	61	93	114	154	182	205
7	27	55	69	103	127	168	205	229

(b) Prefix Cube

Figure 1: Example of an 8×8 Data Cube C and its Prefix Cube $\mathcal{P}$.

at most 2^n cell accesses to evaluate each range-sum query, where n is the dimensionality of the data cube. However, maintaining a prefix cube is very expensive because a single cell modification in the data cube can affect a large number of cells in the prefix cube. For applications where the data cubes are dynamic and are of very large size, having both fast query response as well as efficient cube maintenance is critical. More recently, a variant of the prefix cube approach called **relative prefix cube** (RPC) [4] has been proposed to try to balance the query-update tradeoff between the data cube and prefix cube approaches.

In addition to the above approaches, which provide precise answers to range-sum queries, a method that provides approximate answers for high-dimensional, sparse data cubes has recently been proposed as well [14].

In this paper, our focus is on data cube designs that provide precise answers for range-sum queries. We make the following contributions:

- We propose a new class of cube representations called **hierarchical cubes**. This new class of cubes is based on a design framework that is defined by two orthogonal dimensions. By varying the options along each dimension, various cube designs with different query-update tradeoffs can be generated. In particular, we present two new cube designs called *hierarchical rectangle cubes* (HRC) and *hierarchical band cubes* (HBC), which are generalizations of the existing cube designs.

- We demonstrate analytically that both HRC and HBC have significantly better query-update tradeoff than earlier approaches, for both expected-case as well as worst-case performances.

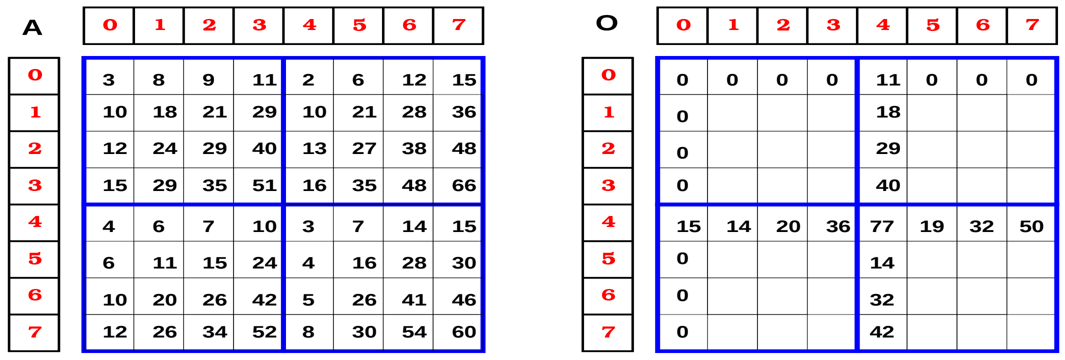
- We also analyze the effect of buffering on the tradeoff among the various classes of precomputed cubes. Our results show that HBC can be more effectively buffered than the other classes of precomputed cubes; by using a moderate amount of main-memory for buffering, HBC can become as query-efficient as the most query-efficient approach without incurring its high update-cost.

Note that all the techniques developed for range-sum queries can be applied to any binary operator for which there exists an inverse operator; other applicable aggregation operators include COUNT, AVERAGE, ROLLING-SUM, and ROLLING-AVERAGE [5].

We conclude this section with some preliminaries. Let C be a n-dimensional data cube of size $D_1 \times D_2 \times \cdots D_n$, where D_i is the cardinality of the i^{th} dimension. For simplicity and without loss of generality, let the domain of the i^{th} dimension be $\{0, 1, \cdots, D_i - 1\}$. We use the generic term **precomputed cube** to refer to a cube belonging to any of the classes of precomputed cubes (i.e., data cube, PC, RPC, HRC, and HBC). We denote a range-sum query by $(l_1 : h_1, l_2 : h_2, \cdots, l_n : h_n)$ and a prefix range-sum query by $(h_1, h_2, \cdots, h_n)$. Given two cells $x = (x_1, x_2, \cdots, x_n)$ and $y = (y_1, y_2, \cdots, y_n)$ in an n-dimensional cube, we say that **x precedes y**, denoted by $x \preceq y$, if and only if $x_i \leq y_i$ for $1 \leq i \leq n$.

For ease of presentation, we use only 2-dimensional cube examples to illustrate the various classes of precomputed cubes, and use the notational convention (x, y) to denote a cube cell in row x and column y.

The rest of this paper is organized as follows. Section 2 presents related work on precomputed cubes for processing range-sum queries. In Section 3, we present a new class of precomputed cubes called *hierarchical cubes*. Section 4 introduces three metrics for comparing the space, update, and query costs of the various classes of precomputed cubes, and presents an analytical comparison of their tradeoffs. In Section 5, we consider the effect of buffering on the query-update tradeoff of precomputed cubes. Finally, we summarize our results in Section 6. Due to space constraints, all analytical results are omitted in this paper but are available elsewhere [2].



A	0	1	2	3	4	5	6	7
0	3	8	9	11	2	6	12	15
1	10	18	21	29	10	21	28	36
2	12	24	29	40	13	27	38	48
3	15	29	35	51	16	35	48	66
4	4	6	7	10	3	7	14	15
5	6	11	15	24	4	16	28	30
6	10	20	26	42	5	26	41	46
7	12	26	34	52	8	30	54	60

(a) Relative-Prefix Array A

O	0	1	2	3	4	5	6	7
0	0	0	0	0	11	0	0	0
1	0				18			
2	0				29			
3	0				40			
4	15	14	20	36	77	19	32	50
5	0				14			
6	0				32			
7	0				42			

(b) Overlay Box O

Figure 2: Example of a Relative Prefix Cube for the Data Cube in Figure 1.

2 Related Work

This section presents two classes of precomputed cubes for processing range-sum queries, namely, prefix cube (PC) and relative prefix cube (RPC).

2.1 Prefix Cubes (PC)

The **prefix cube** of a data cube C, denoted by $\mathcal{P}$, is a cube of the same dimensionality and size as C such that each cell $x = (x_1, x_2, \cdots, x_n)$ in $\mathcal{P}$ stores the result of the prefix range-sum $(x_1, x_2, \cdots, x_n)$; i.e., $\mathcal{P}[x] = \sum_{i_1=0}^{x_1} \sum_{i_2=0}^{x_2} \cdots \sum_{i_n=0}^{x_n} C[i_1, i_2, \cdots, i_n]$. The PC approach exploits the property that any range-sum query can be evaluated in terms of at most 2^n appropriate prefix range-sum queries. Therefore, by precomputing all possible prefix range-sums, the evaluation cost of a range-sum query using a PC is no more than 2^n cell accesses. However, the update cost of the PC is high since every modification of a single data cube cell u affects the set of cells $\{c : u \preceq c\}$ in the PC.

For example, Figure 1 shows an 8×8 data cube and its prefix cube. Evaluating the range-sum query indicated by the shaded region in Figure 1 using the data cube C requires 18 cell accesses. On the other hand, processing the same query using the prefix cube $\mathcal{P}$ is given by $\mathcal{P}[4,6] - \mathcal{P}[4,0] - \mathcal{P}[1,6] + \mathcal{P}[1,0]$ which accesses only 4 cells. The disadvantage, of course, is that updating, for example, cell $C[1,2]$ requires updating cells $\mathcal{P}[i,j]$ for $1 \leq i \leq 7$, $2 \leq j \leq 7$. Details of the prefix cube can be found elsewhere [5].

2.2 Relative Prefix Cubes (RPC)

An approach that has recently been proposed to balance the query-update tradeoff between the data cube and prefix cube is the **relative prefix cube** (RPC) [4]. An RPC consists of two components: (1) a *relative-prefix array* $\mathcal{A}$ and (2) an *overlay box* $\mathcal{O}$. The relative-prefix array $\mathcal{A}$ is similar to a prefix cube $\mathcal{P}$ except that it is partitioned into a number of disjoint sub-cubes of equal size such that each sub-cube is organized as a local prefix sub-cube. By structuring the single, large prefix cube into a collection of smaller prefix sub-cubes, $\mathcal{A}$ limits the effect of an update propagation to a local sub-cube thereby reducing its update-cost.

However, evaluating a prefix range-sum query using only the relative-prefix array $\mathcal{A}$ has a worst-case cost proportional to the number of sub-cubes in $\mathcal{A}$. To improve the worst-case query evaluation cost, the RPC approach also precomputes additional information in a second component called the overlay box. Conceptually, the overlay box $\mathcal{O}$ is a cube of the same dimensionality as $\mathcal{A}$ that is partitioned into a number of sub-boxes (of the same dimensionality as each sub-cube) such that there is one sub-box in $\mathcal{O}$ associated with each sub-cube in $\mathcal{A}$ containing additional precomputed values in some cells.

Figure 2 shows an example of a RPC (for the same data cube in Figure 1) where the relative-prefix array is partitioned into four 4×4 sub-cubes. The cost of evaluating a prefix range-sum query using a RPC is between 1 and $(n + 2)$ cell accesses; specifically, it requires access to one cell in $\mathcal{A}$ and at most $(n+1)$ cells in $\mathcal{O}$. For example, using the RPC in Figure 2, the prefix range-sum query $(5,6)$ is evaluated as $\mathcal{A}[5,6] + \mathcal{O}[4,4] + \mathcal{O}[5,4] + \mathcal{O}[4,6]$.

3 Hierarchical Cubes

In general, we can characterize a precomputed cube X by the way it maps each of its cells to a collection of cells in the data cube, i.e., each cell value in X is a sum over some subset of cell values in the data cube. We refer to each mapped collection of data cube cells as a **mapped region**. For the data cube, the mapped region for each cell v is simply $\{v\}$, while for the prefix cube, the mapped region for each cell v is $\{c : c \preceq v\}$.

In this section, we present a new class of precomputed cube designs called *hierarchical cubes* that require the same space as the data cube but provide better query-update tradeoff than all earlier approaches. This new class is based on a hierarchical organization of the precomputed cube cells, and its design space is characterized by two orthogonal dimensions[1]:

1. **Cube decomposition**
   This dimension organizes the precomputed cube cells into a hierarchical structure.

2. **Cube mapping**
   This dimension defines a mapping from each precomputed cube cell to a particular subset of cells in the data cube by taking into account the hierarchical organization of the precomputed cube cells.

By varying the options along each dimension, various precomputed cube designs with different query-update tradeoffs can be generated.

3.1 Cube Decomposition

The cells in a cube can be organized into a hierarchical structure by decomposing each dimension of the cube. Such decomposition has previously been applied to the design of bitmap indexes [1]. We first introduce the notion of decomposition and then explain how it defines a hierarchical structure on the cube cells.

3.1.1 Decomposition Technique

Consider an attribute A with cardinality D (i.e., $domain(A) = \{0, 1, \cdots, D-1\}$). Given a sequence of m positive integers $B =< b_m, b_{m-1}, \cdots, b_2, b_1 >$ (where $b_m = \left\lceil D / \prod_{i=1}^{m-1} b_i \right\rceil$), an integer $v \in Domain(A)$ can be decomposed into a sequence of m **component values** $V =< v_m, v_{m-1}, \cdots, v_2, v_1 >$ as follows:

$$v = v_m \left(\prod_{j=1}^{m-1} b_j \right) + \ldots + v_i \left(\prod_{j=1}^{i-1} b_j \right) + \ldots + v_2 b_1 + v_1. \tag{1}$$

Each component value v_i is a base-b_i digit (i.e., $0 \leq v_i < b_i$). We refer to B as a **base-sequence** and V as the **B-decomposition of v**.

For example, consider $D = 24$ and $v = 22$. Since $22 = 3(6)+4 = 1(3)(4)+2(4)+2$, therefore, $< 3, 4 >$ is the $< 4, 6 >$-decomposition of v, and $< 1, 2, 2 >$ is the $< 2, 3, 4 >$-decomposition of v. Thus, by varying the base-sequence, different decompositions of an attribute value can be obtained.

[1]We note that the design framework defined here for data cubes is similar to that defined for the design space of bitmap indexes [1]. In fact, both design spaces can be abstracted into a more general framework, but a discussion of this is beyond the scope of this paper.

3.1.2 Hierarchical Organization of Cube Cells

We now explain how the cells in an n-dimensional data cube can be organized into a hierarchical structure by applying a (possibly different) base-sequence to each dimensional attribute of the cube. Let m denote the length of the longest base-sequence among the n base-sequences. Base-sequences that are shorter than m are padded with base numbers of value 1 so that all the n base-sequences have the same length of m. Given this, let $\mathcal{B}_i =< b_{i,m}, b_{i,m-1}, \cdots, b_{i,1} >$ denote the base-sequence that is used to decompose the i^{th} dimensional attribute, for $1 \leq i \leq n$; and $p_{i,j}$ denote $\prod_{k=1}^{j} b_{i,k}$ for $1 \leq i \leq n$ and $1 \leq j \leq m$. For notational convenience, let $p_{i,0} = 1$. A data cube that is decomposed using length-m base-sequences has **height** m.

The set of n base-sequences $\{\mathcal{B}_1, \mathcal{B}_2, \cdots, \mathcal{B}_n\}$ recursively partitions the data cube, organizing its cells in a forest as follows:

1. The data cube is first partitioned into $\prod_{i=1}^{n} b_{i,m}$ sub-cubes each of size $\prod_{i=1}^{n} p_{i,m-1}$. The **rank** of each sub-cube is equal to m. The smallest cell in each sub-cube[2] is made a root in the forest at level m.

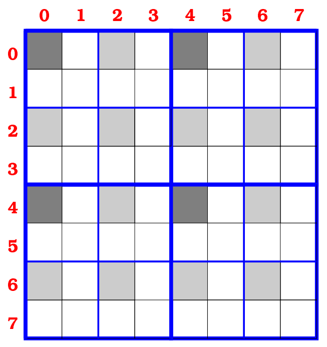
2. Each rank-k sub-cube S $(1 < k \leq m)$ is partitioned into $t = \prod_{i=1}^{n} b_{i,k-1}$ rank-(k-1) sub-cubes each of size $\prod_{i=1}^{n} p_{i,k-2}$. The smallest cell in one of these t sub-cubes is the same as the smallest cell (denoted by c) in the sub-cube S and is already in the forest under construction. The smallest cell in each of the remaining $(t-1)$ sub-cubes becomes a child of c in the forest at level $k-1$.

3. Each rank-1 sub-cube consists of only one cell which is a leaf in the forest at level 1.

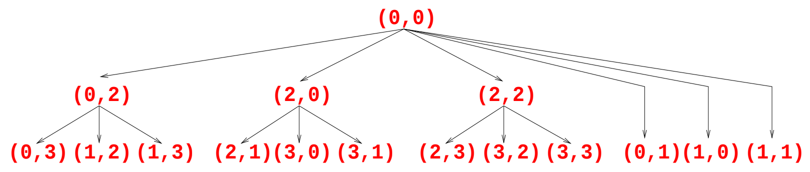
We denote the parent of a cell v by **parent(v)**. If $c = (c_1, c_2, \cdots, c_n)$ is a level-k cell with $k < m$, then $parent(c) = (p_1, p_2, \cdots, p_n)$, where $p_i = c_i - (c_i \ mod \ b_{i,k})$, for $1 \leq i \leq n$. The level of a cell v is denoted by **level(v)**, where $1 \leq level(v) \leq m$.

Figure 3 shows an example of cube decomposition for an 8×8 data cube. In Figure 3(a), the data cube is decomposed with the base-sequence $< 2, 2, 2 >$ for each dimension, which partitions the data cube into four 4×4 rank-3 sub-cubes. Each rank-3 sub-cube is further partitioned into four 2×2 rank-2 sub-cubes

[2]The *smallest cell* in a sub-cube S refers to the cell in S that precedes all other cells in S; the smallest cell is unique due to the rectangular structure of the sub-cube.



(a) B = { <2,2,2>, <2,2,2> }



(b) Hierarchical Organization

Figure 3: Example of Cube Decomposition.

Cell v	**Cube Mappings**	
	Hierarchical Rectangle (HR)	**Hierarchical Band (HB)**
Root cell	$\{v\}$	$\{c : c \preceq v\}$
Non-root cell	$\{c : parent(v) \preceq c \preceq v\}$	$\{c : c \preceq v\} - \{c : c \preceq parent(v)\}$

Table 1: Two Cube Mappings based on Hierarchical Organization of Cube Cells.

each of which consists of 4 cells. Figure 3(b) shows the hierarchical organization for the upper-left rank-3 sub-cube in Figure 3(a). The root cell is $(0,0)$ which is the parent cell of three level-2 cells ($(0,2)$, $(2,0)$, and $(2,2)$) and three level-1 cells ($(0,1)$, $(1,0)$, and $(1,1)$); and each level-2 cell is the parent cell of three level-1 cells (e.g., $(2,2)$ is the parent cell of cells $(2,3)$, $(3,2)$, and $(3,3)$).

Given a cell c in a cube X, let $\lambda(\mathbf{c})$ denote the "largest" descendant cell[3] of c in X; i.e., for any descendant cell c' of c in X, $c' \preceq \lambda(c)$. For example, in Figure 3(a), $\lambda((0,0)) = (3,3)$, $\lambda((4,2)) = (5,3)$, and $\lambda((6,3)) = (6,3)$.

3.2 Cube Mappings

Based on the hierarchical organization of cube cells, we present two cube mappings, **hierarchical rectangle (HR)** mapping and **hierarchical band (HB)** mapping, defined in Table 1. Each mapping is defined in terms of two cases depending on whether the cell being mapped is a root cell or not. Figure 4 illustrates these two cube mappings on an 8×8 cube that is decomposed with the base-sequence $< 2, 4 >$ for each dimension. For HR mapping, the mapped region for a root cell r contains only r itself, and the mapped region for a non-root cell c is the "rectangle" of cells defined by $parent(c)$ and c. For HB mapping, the mapped region for a root cell r contains all the cells preceding r and including r itself, and the mapped region for a non-root cell c is an "angular band" of cells defined by the difference between two regions: (1) the cells preceding c and including c itself, and (2) the cells preceding $parent(c)$ and including $parent(c)$ itself.

[3]A cell x is an **ancestor cell** of a cell y (or equivalently, y is a **descendant cell** of x) if either x is a parent cell of y; or x is a parent cell of some cell z, and z is an ancestor cell of y.

We refer to precomputed cubes based on the HR mapping as **hierarchical rectangle cubes (HRC)**, and to those based on the HB mapping as **hierarchical band cubes (HBC)**. Figure 2(a) shows an example of an HRC decomposed with $< 2, 4 >$ for each dimension, and Figure 5 shows an example of an HBC decomposed with $< 2, 2, 2 >$ for each dimension.

3.3 Design Space

Figure 6 shows the design space of precomputed cubes defined by the two dimensions. Along the cube decomposition dimension, we have the various base-sequences to decompose the dimensional attributes of the precomputed cube. Note that each dimensional attribute can be decomposed with a different base-sequence. Along the cube mapping dimension, we have the two mappings, HR and HB. By combining different options along these two dimensions, various data cube designs with different query-update tradeoffs can be obtained.

Interestingly, both data cubes and prefix cubes turn out to be special cases of HRC and HBC, respectively; this occurs when each dimensional attribute A_i of the cube is decomposed with the base-sequence $< D_i >$ so that all the cube cells are root cells (see Table 1 also). Furthermore, the relative-prefix array component of RPC (i.e., RPA) is just a HRC with a height of 2.

3.4 Hierarchical Rectangle Cubes (HRC)

In this section, we present algorithms for constructing, querying, and updating an n-dimensional HRC $\mathcal{H}$ for the case when $\mathcal{H}$ is not a data cube.

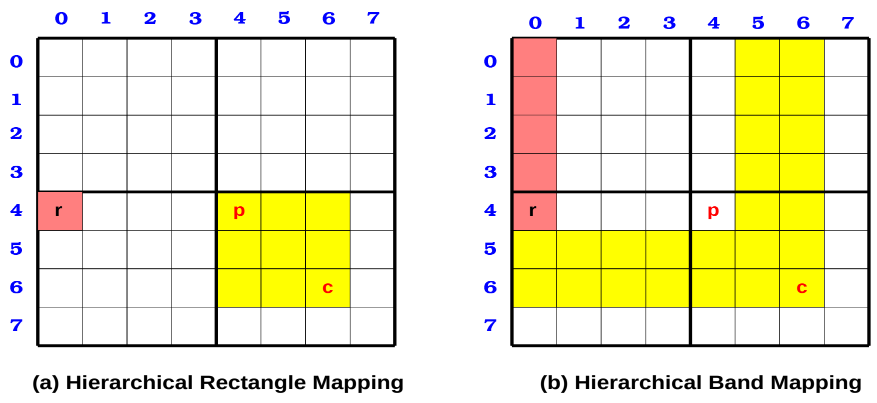
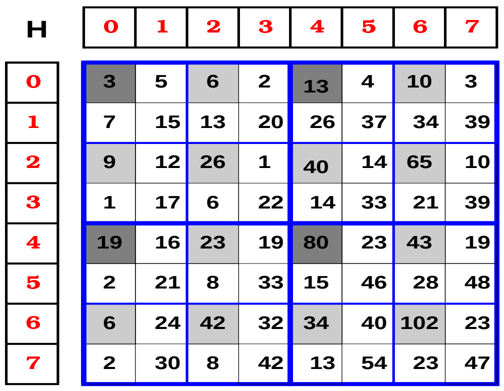

0 1 2 3 4 5 6 7
r
p
c
(a) Hierarchical Rectangle Mapping
(b) Hierarchical Band Mapping

Figure 4: Example of Cube Mappings on an 8×8 cube decomposed with $< 2, 4 >$ for each dimension. Cells r and p are root and $p = parent(c)$.



H	0	1	2	3	4	5	6	7
0	3	5	6	2	13	4	10	3
1	7	15	13	20	26	37	34	39
2	9	12	26	1	40	14	65	10
3	1	17	6	22	14	33	21	39
4	19	16	23	19	80	23	43	19
5	2	21	8	33	15	46	28	48
6	6	24	42	32	34	40	102	23
7	2	30	8	42	13	54	23	47

Figure 5: Example of a HBC for the Data Cube in Figure 1.

3.4.1 Construction Algorithm

The construction algorithm for an HRC $\mathcal{H}$ is similar to that for PC except that the prefix range-sum in each cell of $\mathcal{H}$ is computed locally with respect to the cell's parent rather than with respect to the cell $(0, 0, \cdots, 0)$ [5].

3.4.2 Query Algorithm

Figure 7 shows the algorithms for evaluating a range-sum query using an HRC $\mathcal{H}$. Algorithm **Rewrite-LocalRSQ** rewrites a range-sum query Q into a collection of local range-sum queries, with one local range-sum query for each rank-2 sub-cube in $\mathcal{H}$ that overlaps with Q. Algorithm **RewriteLocalPRSQ** rewrites a range-sum query Q into a collection of local prefix range-sum queries by first invoking Algorithm **RewriteLocalRSQ** to obtain a set of local range-sum queries (step 1), and then further rewriting each local range-sum query into a collection of local prefix range-sum queries (steps 2 to 4); Q^+ (Q^-) contains all the local prefix range-sum queries whose results are to be added (subtracted) for the answer to Q. The details

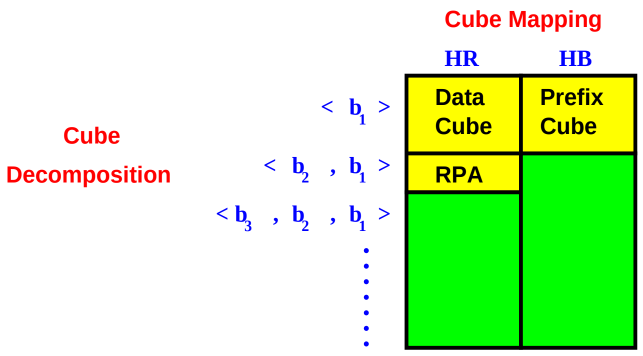


Figure 6: Design Space for Precomputed Cubes.

for step 4 can be found elsewhere [5].

Algorithm **HRC-Query** is the main algorithm to evaluate a range-sum query Q using $\mathcal{H}$. In step 1, it invokes Algorithm **RewriteLocalPRSQ** to rewrite Q into a collection of local prefix range-sum queries $Q^+ \cup Q^-$. For each $q \in Q^+ \cup Q^-$, if q corresponds to a root or leaf cell in $\mathcal{H}$, then q can be evaluated by accessing the cell $\mathcal{H}[q]$. Otherwise, q needs to be evaluated in terms of a collection of local prefix range-sum queries; the additional rewriting for q is performed in steps 2 to 9. Note that if $\mathcal{H}$ has a height of 2, then steps 2 to 9 are omitted.

3.4.3 Update Algorithm

Figure 8 shows the algorithm to maintain an HRC $\mathcal{H}$ in response to a single-cell update in the data cube C. By definition of HRC, the update of a single data cube cell u can affect only the cells in the rank-m sub-cube S in $\mathcal{H}$ that contains u. The root cell in S is identified in step 1, and it needs to be updated only if it corresponds to cell u (steps 2 and 3). Steps 4 to 7 of the algorithm handle updates to the non-root cells in S and are based on the property that a non-root cell c in S is affected by the update of cell u if and only if *parent*$(c) \preceq u \preceq c$.

3.5 Hierarchical Band Cubes (HBC)

In this section, we present algorithms for constructing, querying, and updating an n-dimensional HBC $\mathcal{H}$.

3.5.1 Construction Algorithm

Figure 9 shows a 2-stage algorithm for constructing a HBC from a data cube C. In the first stage (step 1), the prefix cube of C is constructed (an algorithm for this can be found in Section 3.3 in [5]). In the second stage (steps 2 to 5), the HBC is derived from the constructed prefix cube by adjusting the values of the non-root cells. On an implementation note, the order in which the cells are modified should exploit the physical clustering of the cells to minimize I/O.

3.5.2 Query Algorithm

Figure 10 shows the algorithms for evaluating a range-sum query using an HBC. Algorithm **EvaluatePRS-Query** evaluates a prefix range-sum query Q using an HBC $\mathcal{H}$. By the definition of HB mapping (Table 1), the answer is a summation of at most m cell values. Algorithm **HBC-Query** is the main algorithm to evaluate a range-sum query using an HBC. Similarly to PC and HRC, the range-sum query is first rewritten into a collection of prefix range-sum queries $Q^+ \cup Q^-$ (step 2).

3.5.3 Update Algorithm

An update of a data cube cell u affects a cell c in an HBC $\mathcal{H}$ if and only if the value in cell c is derived using the value in cell u. The following two properties are true for HBC $\mathcal{H}$:

(P1) The update of u affects c if and only if (1) none of the ancestor cells of c are affected by the update of u, and (2) $u \preceq c$.

(P2) If $u \not\preceq \lambda(c)$, then none of the descendant cells of c (including c) are affected by the update of cell u.

Property (P1) follows from the definition of HB mapping (Table 1), and property (P2) is a corollary of property (P1). Their proofs are given elsewhere [2].

```
Algorithm RewriteLocalRSQ ($\mathcal{H}, \beta, \mathcal{Q}$)
Input: $\mathcal{H}$ is an $n$-dimensional HRC with height $m$.
        $\beta = \{B_1, B_2, \cdots, B_n\}$ is a set of base-
        sequences; each $B_i = < b_{i,m}, \cdots, b_{i,2}, b_{i,1} >$.
        $\mathcal{Q} = (l_1 : h_1, l_2 : h_2, \cdots, l_n : h_n)$ is a
        range-sum query.
Output: $T$ is a set of local range-sum queries
        such that $Q = \sum_{q \in T} q$.
1)  Let $\mathcal{S} = \{S_1, S_2, \cdots, S_k\}$ be the set of all the rank-2
    sub-cubes in $\mathcal{H}$ that overlap with $\mathcal{Q}$.
2)  $T = \{\}$;
3)  for each sub-cube $S_i \in \mathcal{S}$ do
4)      $Q_i = (x_1 : y_1, x_2 : y_2, \cdots, x_n : y_n)$, where
        $x_j = \max\{l_j, a_j\}$, $y_j = \min\{h_j, a_j + b_{j,1} - 1\}$
        and $a = (a_1, a_2, \cdots, a_n)$ is the smallest cell in $S_i$;
5)      $T = T \cup \{Q_i\}$;
6)  return $T$;
```

```
Algorithm RewriteLocalPRSQ ($\mathcal{H}, \beta, \mathcal{Q}$)
Input: $\mathcal{H}$ is an $n$-dimensional HRC with height $m$.
        $\beta = \{B_1, B_2, \cdots, B_n\}$ is a set of base-
        sequences; each $B_i = < b_{i,m}, \cdots, b_{i,2}, b_{i,1} >$.
        $\mathcal{Q} = (l_1 : h_1, l_2 : h_2, \cdots, l_n : h_n)$ is a range-sum
        query.
Output: $Q^+ \cup Q^-$ is a collection of local prefix
        range-sum queries such that
        $Q = \sum_{q \in Q^+} q - \sum_{q \in Q^-} q$.
1)  $T$ = RewriteLocalRSQ($\mathcal{H}$, $\beta$, $\mathcal{Q}$);
2)  $Q^+ = \{\}$; $Q^- = \{\}$;
3)  for each $q \in T$ do
4)      Rewrite $q$ into a combination of local prefix
        range-sum queries, and update $Q^+$ and $Q^-$;
5)  return $(Q^+, Q^-)$;
```

```
Algorithm HRC-Query ($\mathcal{H}, \beta, \mathcal{Q}$)
Input:  $\mathcal{H}$ is an $n$-dimensional HRC with height $m$.
        $\beta = \{B_1, B_2, \cdots, B_n\}$ is a set of base-
        sequences; each $B_i = < b_{i,m}, \cdots, b_{i,2}, b_{i,1} >$.
        $\mathcal{Q} = (l_1 : h_1, l_2 : h_2, \cdots, l_n : h_n)$ is a
        range-sum query.
Output: The answer to $\mathcal{Q}$.
1)  $(Q^+, Q^-)$ = RewriteLocalPRSQ($\mathcal{H}$, $\beta$, $\mathcal{Q}$);
2)  for each $q \in Q^+$ do
3)      if $(1 < level(q) < m)$ then
4)          $p = parent(q)$;
5)          $\mathcal{Q}' = (p_1 : q_1, p_2 : q_2, \cdots, p_n : q_n)$;
6)          $T$ = RewriteLocalRSQ($\mathcal{H}$, $\beta$, $\mathcal{Q}'$);
7)          $Q^- = Q^- \cup T - \{q\}$;
8)  Repeat steps (2) to (7) with $Q^+$ & $Q^-$ interchanged;
9)  $S = Q^+ \cap Q^-$;  $Q^+ = Q^+ - S$;  $Q^- = Q^- - S$;
10) return $\sum_{q \in Q^+} \mathcal{H}[q] - \sum_{q \in Q^-} \mathcal{H}[q]$;
```

Figure 7: Algorithms to Evaluate a Range-Sum Query using an HRC.

Figure 11 shows the algorithm to maintain an HBC in response to a single-cell update in the data cube C; its correctness is based on properties (P1) and (P2). Property (P2) is used in step 1 (step 6) to select root cells (child cells) that are either affected by the update of u, or whose descendant cells are affected by the update of u. Property (P1) is used in step 3 to decide whether or not a cell in $\mathcal{H}$ is affected by the update of u. An implementation of this update algorithm should exploit the physical clustering of the cells in $\mathcal{H}$ to order the sequence of cell updates so as to minimize I/O.

```
Algorithm HRC-Update ($\mathcal{H}, u, \delta$)
Input:  $\mathcal{H}$ is an HRC to be updated.
        $u$ is a cell in the data cube that has been
        modified.
        $\delta$ is the difference between the new and old values
        of cell $u$.
Output: $\mathcal{H}$, an updated HRC.
1) let $r$ be the root cell in $\mathcal{H}$ such that $r \preceq u \preceq \lambda(r)$;
2) if ($r = u$) then
3)     Update cell $r$ with $\delta$;
4) $S = \{c \mid r = parent(c),\ r \preceq u \preceq c\}$;
5) for each $p \in S$ do
6)     Update cell $p$ with $\delta$;
7)     $S = S \cup \{c \mid p = parent(c),\ p \preceq u \preceq c\} - \{p\}$;
8) return $\mathcal{H}$;
```

Figure 8: Algorithm to Update an HRC.

```
Algorithm HBC-Construct ($C, \beta$)
Input:  $C$ is an $n$-dimensional data cube.
        $\beta = \{B_1, B_2, \cdots, B_n\}$ is a set of base-
        sequences; each $B_i = < b_{i,m_i}, \cdots, b_{i,2}, b_{i,1} >$.
Output: $\mathcal{H}$, a hierarchical band cube of $C$.
1) Construct the prefix cube of $C$ and call it $\mathcal{H}$;
2) $L = \max\{m_1, m_2, \cdots, m_n\}$;
3) for i = 1 to L-1 do
4)         for each level-$i$ cell $v \in \mathcal{H}$ do
5)                 $\mathcal{H}[v] = \mathcal{H}[v] - \mathcal{H}[parent(v)]$;
6) return $\mathcal{H}$;
```

Figure 9: Algorithm to Construct an HBC from a Data Cube.

4 Precomputed Cube Comparison

4.1 Performance Metrics

This section presents three performance metrics, namely, space-cost, query-cost, and update-cost, for comparing the tradeoffs among the various classes of precomputed cubes.

Let X denote a precomputed cube. The **space-cost** of X, denoted by $Space(X)$, is the total number of cells (i.e., values) in X. The **expected query-cost** (**worst query-cost**) of X, denoted by $E_{query}(X)$ ($W_{query}(X)$), is the expected (highest) number of cells in X that are accessed to answer a range-sum query. The **expected update-cost** (**worst update-cost**) of X, denoted by $E_{update}(X)$ ($W_{update}(X)$), is the expected (highest) number of cells in X that need to be updated in response to a single cell modification in the data cube. $E_{query}(X)$ ($E_{update}(X)$) assumes a uniform distribution over the collection of all possible range-sum queries (single-cell updates in the data cube).

```
Algorithm EvaluatePRSQuery ($\mathcal{H}, \mathcal{Q}$)
Input:  $\mathcal{H}$ is an HBC.
        $\mathcal{Q}$ is a prefix range-sum query.
Output: $sum$, the answer to $\mathcal{Q}$.
1) $sum = \mathcal{H}[\mathcal{Q}]$;
2) while ($\mathcal{Q}$ is a non-root cell) do
3)         $\mathcal{Q} = parent(\mathcal{Q})$;
4)         $sum = sum + \mathcal{H}[\mathcal{Q}]$;
5) return $sum$;
```

```
Algorithm HBC-Query ($\mathcal{H}, \mathcal{Q}$)
Input:  $\mathcal{H}$ is an HBC.
        $\mathcal{Q}$ is a range-sum query.
Output: $sum$, the answer to $\mathcal{Q}$.
1) $Q^+ = \{\}$; $Q^- = \{\}$;
2) Rewrite $\mathcal{Q}$ into a combination of prefix
   range-sum queries, and update $Q^+$ and $Q^-$;
3) $sum = 0$;
4) for each $q \in Q^+$ do
5)         $sum = sum +$ EvaluatePRSQuery ($\mathcal{H}, q$);
6) for each $q \in Q^-$ do
7)         $sum = sum -$ EvaluatePRSQuery ($\mathcal{H}, q$);
8) return $sum$;
```

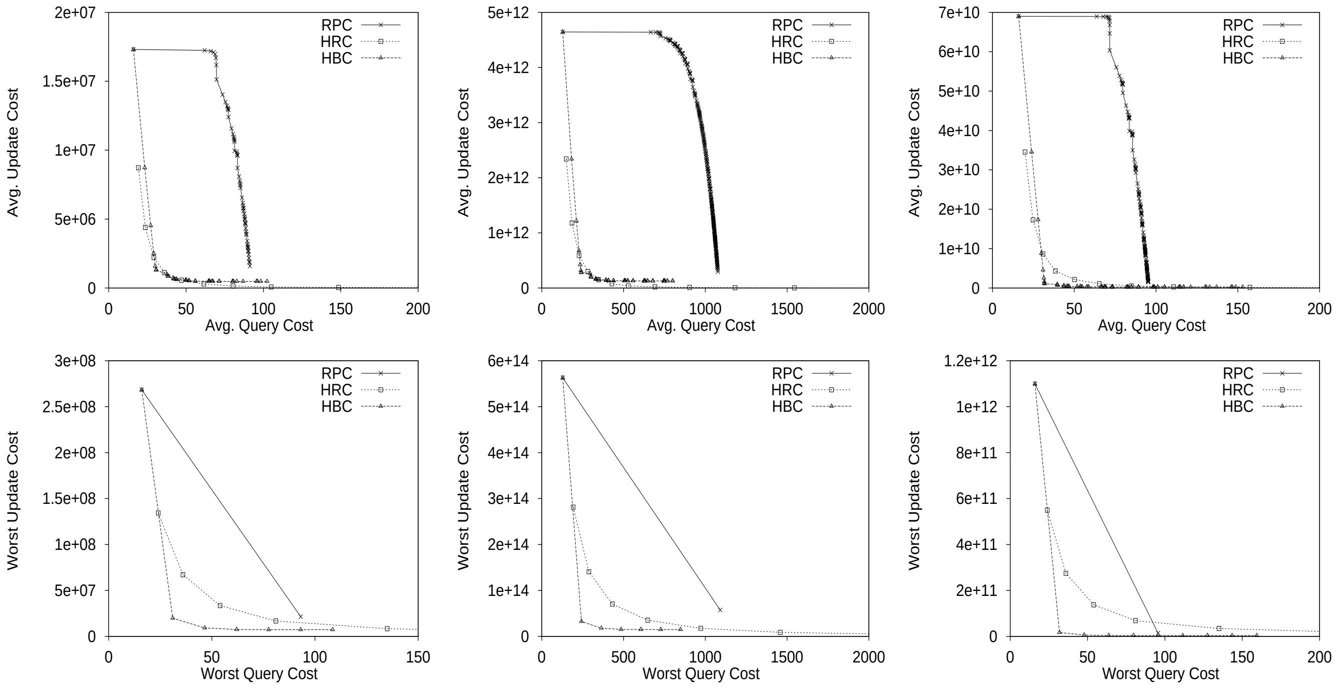
Figure 10: Algorithms to Evaluate a Range-Sum Query using an HBC.

4.2 Comparison of Query-Update Tradeoff

In this section, we compare the performance of RPC, HRC, and HBC based on analytical results for the space-, query-, and update-cost metrics defined in the

```
Algorithm HBC-Update ($\mathcal{H}, u, \delta$)
Input: $\mathcal{H}$ is an HBC to be updated.
       $u$ is a cell in the raw data cube that has been
       modified.
       $\delta$ is the difference between the new and old values
       of cell $u$.
Output: $\mathcal{H}$, an updated HBC.
1) $S = \{c \mid c$ is a root cell in $\mathcal{H},\ u \preceq \lambda(c)\}$;
2) for each $p \in S$ do
3)     if ($u \preceq p$) then
4)         Update cell $p$ with $\delta$;
5)     else
6)         $S = S \cup \{c \mid p = parent(c),\ u \preceq \lambda(c)\}$;
7)     $S = S - \{p\}$;
8) return $\mathcal{H}$;
```

Figure 11: Algorithm to Update an HBC.



(a) $n = 4$, $D = 128$. (b) $n = 7$, $D = 128$. (c) $n = 4$, $D = 1024$.

Figure 12: Comparison of Average- and Worst-Case Query-Update Tradeoff.

previous section. Analytical formulae for these metrics are given elsewhere [2]. In terms of space cost, both HRC and HBC have the same space requirement as the data cube C, while RPC requires more space than these (for the cells in the overlay box).

We focus on comparing just the query-update tradeoff among the various classes of precomputed cubes using *optimal query-update tradeoff graphs* (for both expected-case as well as worst-case) defined as follows. Let S denote the set of all precomputed cubes that belong to the same class, say X, and have the same values for n and D. A cube $s \in S$ has *optimal expected-case query-update tradeoff* if there does not exist another cube $s' \in S$ that satisfies all of the following conditions: (1) $E_{query}(s') \leq E_{query}(s)$, and (2) $E_{update}(s') \leq E_{update}(s)$, and (3) at least one of the above inequalities is strict. The corresponding points in a query-cost/update-cost diagram constitute the *optimal expected-case query-update tradeoff graph* for X. The *optimal worst-case query-update tradeoff cubes* for X and their graphs are defined similarly in terms of $W_{query}()$ and $W_{update}()$. A cube of class X that has optimal tradeoff, whether in the expected case or the worst case, is called **optimal X** for short, when no confusion arises.

We generated precomputed cubes for the various classes (i.e., RPC, HRC, and HBC) by varying the number of cube dimensions ($2 \leq n \leq 7$), and the size of each cube dimension. For simplicity, we consider data cubes with equal sized dimensions; i.e., data cubes with $D_i = D$ (for $1 \leq i \leq n$) with $D = 128, 512, 1024$. For a given n and D, all possible RPCs are generated by enumerating all combinations for the dimensions of the sub-cube/overlay box. For HBCs, we consider only cubes for which each base-sequence $B_i = < b_{i,m}, b_{i,m-1}, \cdots, b_{i,1} >$ satisfies the property that $b_{i,m} \geq b_{i,m-1} \geq \cdots \geq b_{i,1}$ because initial experiments with $D = 128$ showed that only these are optimal.

For HRCs, we consider only HRCs with height equal to 2. The analysis for the more general class of HRCs is more complex and is part of our future work. Note that for every HRC X with a height of k, $k > 2$, there exists an HRC Y with a height of 2 that has the same dimensionality and size for its rank-2 sub-cubes as X. In terms of update-cost, Y is more efficient than X: although the mapped regions for their leaf cells are equivalent, the mapped regions for the non-leaf-and-non-root cells in X are larger, while the mapped regions for each of the non-leaf cells in Y (i.e., root cells) consists of only a single value. In terms of query-cost, the comparison is less clear since neither X nor Y is a clear winner.

Figure 12 compares both the expected-case and worst-case query-update tradeoff for three cases: (a) $n = 4$ and $D = 128$, (b) $n = 7$ and $D = 128$, and (c) $n = 4$ and $D = 1024$. The top leftmost point in each graph (i.e., the most query-efficient and least update-efficient cube design) corresponds to the prefix cube which is a special case of both RPC and HBC.

In terms of the expected-case query-update performance, both HRC and HBC have comparable tradeoff, and they significantly outperform RPC by up to a factor of 10. In terms of the worst-case query-update performance, both HRC and HBC again have better tradeoff than RPC, with HBC being clearly the winner.

With respect to how query cost is traded off for update cost by each cube design, we have observed the following. For RPCs, the update-cost decreases but the query-cost increases as the number of cells in the overlay box decreases. For HRCs, this happens as the number of rank-2 subcubes increases. For HBCs, things are more complicated. Let $N_i(H)$ denote the total number of level-i cells in a HBC H. For the expected-case, an optimal HBC X with a height of x is more query-efficient but less update-efficient than an optimal HBC Y with a height of y if either (1) $x < y$, or (2) $x = y$, and $N_k(X) < N_k(Y)$ for some $k \geq 1$, and $N_i(X) = N_i(Y)$ for $1 \leq i < k$. For the worst-case, there is exactly one optimal HBC with a height of i for $1 \leq i \leq log_2(D)$ such that the optimal HBC with a height of k is more query-efficient but less update-efficient than the optimal HBC with a height of $(k+1)$ for $1 \leq k < log_2(D)$.

5 Effect of Buffering

5.1 Performance Metrics

One important result from the previous section (Figure 12) is that PC is the most query-efficient but also the least update-efficient precomputed cube. In particular, the query-cost is one cell access per prefix range-sum query evaluation, which implies a worst-case evaluation cost of 2^n cell accesses per range-sum query evaluation, where n is the number of dimensions in the data cube. In this section, we explore the effect of buffering on the performance of the various precomputed cubes; specifically, we examine how each class of precomputed cubes can achieve, with appropriate buffering, the same worst-case query-efficiency as PC but without incurring the high update cost of PC. We compare the *memory-update tradeoff* of the various classes of precomputed cubes, which involves the following two metrics for each class X:

1. Memory: the minimum number of cells in X that need to be buffered so that X achieves the same worst-case query-cost as PC (i.e., access at most 2^n non-buffered cells per range-sum query evaluation). We denote this metric by $MinCell(X)$.

2. Update: the expected number of non-buffered cells in X that need to be updated in response to a single cell modification in the data cube, assuming a uniform distribution over the collection of all $\prod_{i=1}^{n} D_i$ potential single-cell updates. We refer to this metric as the *expected buffered update-cost of* X and denote it by $B_{update}(X)$.

In other words, for each precomputed cube X, we are interested in determining both the cost (in terms of buffer space) as well as the benefit (in terms of the improved update cost) of buffering X for it to become as query-efficient as PC.

5.2 Buffering Strategies

We briefly explain the buffering strategy for each class of precomputed cubes X for the metric $MinCell(X)$. Let Q denote a range-sum query. For an RPC $\mathcal{R}$, each prefix range-sum query evaluation accesses exactly one cell in $\mathcal{A}$ and at most $(n+1)$ cells in $\mathcal{O}$. It follows that for RPC to have the same worst-case query efficiency as PC, at least all the cells in $\mathcal{O}$ need to be buffered.

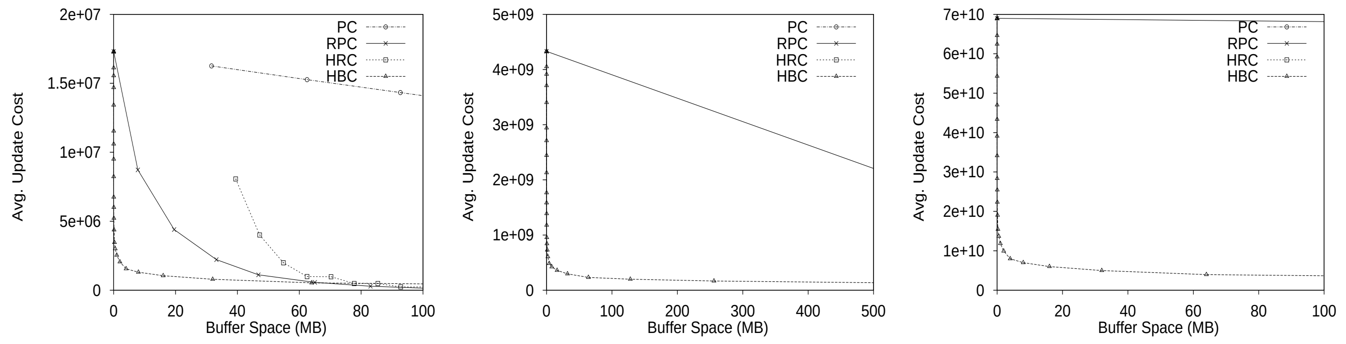
For an HRC $\mathcal{H}$, Q is rewritten into σ local range-sum queries, where σ is the number of rank-2 subcubes that overlapped with Q; each local range-sum query is in turn evaluated in terms of at most 2^n local prefix range-sum queries. Thus, in the worst case, at most $\sigma 2^n$ cells are accessed, of which at most 2^n of them are *non-outer cells*[4]. Therefore, at least all the outer cells in $\mathcal{H}$ need to be buffered for it to be equally query-efficient as PC.

For an HBC $\mathcal{H}$, each prefix range-sum query evaluation accesses exactly one root-level cell and at most one cell for each non-root level. Since HBC has the property that the number of level-i cells decreases with i, buffering all the non-leaf cells incurs the least amount of buffer space for $\mathcal{H}$ to achieve the same worst-case query-efficiency as PC.

In addition to comparing the cost and benefit of buffering each class of precomputed cubes for it to attain the same query-efficiency as PC, we are also interested in examining how buffering benefits PC in terms of reducing its update-cost. Since each cell in a PC is equally likely to be accessed, a good buffering scheme is to buffer those cells that are more likely to be updated so as to reduce the update-cost of PC. A reasonably good buffering strategy for PC is to buffer it in units of *layers.* Consider a prefix cube $\mathcal{P}$ with each dimension of size D. The cells in $\mathcal{P}$ can be partitioned into D disjoint *layers* such that a cell $(c_1, c_2, \cdots, c_n)$ belongs to *layer* k of $\mathcal{P}$, $0 \leq k < D$, if and only if $\max\{c_1, c_2, \cdots, c_n\} = k$. A cell in layer k is generally more likely to be updated than a cell in layer $(k-1)$. We consider the effect of buffering $\mathcal{P}$ in terms of k outermost layers[5], $1 \leq k \leq D$. Let $L_{space}(\mathcal{P}, k)$ denote the total number of cells in the k outermost layers of $\mathcal{P}$, and $L_{update}(\mathcal{P}, k)$ denote the expected number of

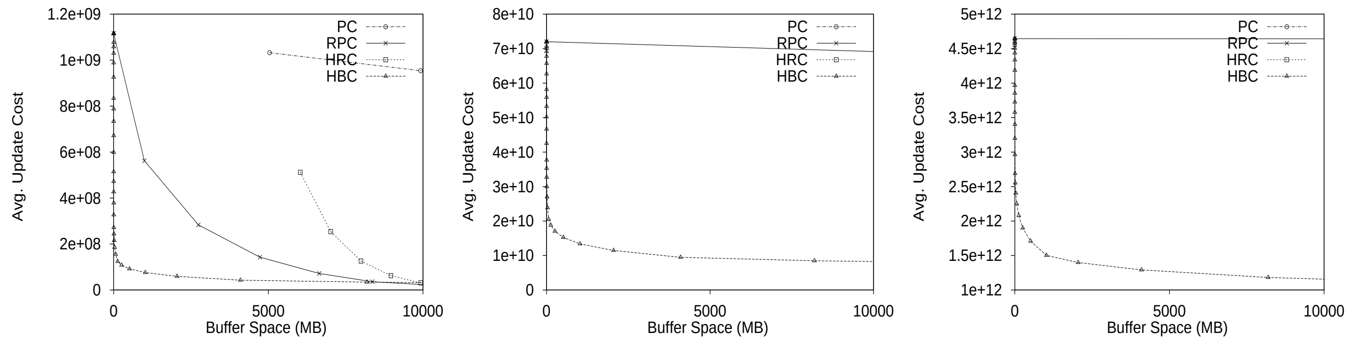
[4]A cell $c = (c_1, c_2, \cdots, c_n)$ in a rank-2 subcube S of an HRC $\mathcal{H}$ is an *outer cell* if $\exists\ 1 \leq j \leq n$, such that $c_j = \max\{s_j | s \in S, s = (s_1, s_2, \cdots, s_n)\}$; otherwise, c is an *non-outer cell.*

[5]Note that the k outermost layers refer to layers $(D-k)$, $(D-k+1)$, $\cdots$, $(D-1)$.



(a) $D = 128$. (b) $D = 512$. (c) $D = 1024$.

Figure 13: Comparison of Average-Case Memory-Update Tradeoff, $n = 4$.



(a) $n = 5$. (b) $n = 6$. (c) $n = 7$.

Figure 14: Comparison of Average-Case Memory-Update Tradeoff, $D = 128$.

non-buffered cells in $\mathcal{P}$ that need to be updated in response to a single cell update in the data cube when the k outermost layers of $\mathcal{P}$ are buffered. Analytical results for $B_{update}(X)$, $MinCell(X)$, $L_{space}(\mathcal{P}, k)$, and $L_{update}(\mathcal{P}, k)$ are given elsewhere [2].

5.3 Comparison of Memory-Update Tradeoff

In this section, we compare the various classes of precomputed cubes using *optimal expected-case memory-update tradeoff graphs*, which are defined similarly as the optimal expected-case query-update tradeoff graphs in Section 4.2, but in terms of $MinCell()$ and $B_{update}()$. For a PC $\mathcal{P}$, the optimal expected-case memory-update tradeoff graph is defined in terms of $L_{space}(\mathcal{P}, k)$ and $L_{update}(\mathcal{P}, k)$ with k being varied from 1 to D. Both $MinCell(X)$ and $L_{space}(\mathcal{P}, k)$ are expressed in units of MB of buffer space by assuming that each cell takes 4 bytes of memory.

As in Section 4.2, precomputed cubes are generated by varying the number of cube dimensions ($2 \leq n \leq 7$) and the size of each cube dimension ($D = 128, 512, 1024$). We consider all possible RPCs and the subclass of HRCs with height 2. Recall from the previous section that the buffering strategy for HBC buffers all the non-leaf cells. Since the number of leaf cells in an HBC and the update cost for them are dependent on the size of the rank-2 subcube but independent of the total number of levels in the HBC, it suffices to consider only the subclass of HBCs with height 2 as well for the memory-update tradeoff comparison in this section.

Figure 13 compares the optimal expected-case memory-update tradeoff for $n = 4$ and $D \in \{128, 512, 1024\}$; the graphs for $D = 128$ and $5 \leq n \leq 7$ are shown in Figure 14. They indicate that HBC has a significantly better memory-update tradeoff than the other classes of precomputed cubes, particularly for large values of n or D. For example, when $n = 4$ and $D = 128$ (Figure 13(a)), the expected update-cost of PC (without buffering) is about 17×10^6. Using a buffer space of 8 KB, HBC reduces this cost by a factor of 2.5 to 8×10^6; by increasing the buffer space to 1 MB, the expected update cost is improved by a factor of almost 7 to 2.5×10^6. Thus, with a moderate amount of buffer space, HBC can become as query-efficient as PC but with a significant improvement over its update-cost.

For RPCs, as the size of sub-cube decreases, the number of overlay box cells increases, and so the buffer space requirement increases and the expected update cost decreases. For HRCs, as the size of the rank-2 sub-cube decreases, the number of outer cells increases, and so the buffer space requirement increases and the expected update cost decreases. For HBCs, as the size of the rank-2 subcube decreases, the number of leaf cells decreases, and so the buffer space requirement increases and the expected update cost decreases.

Note that some of the graphs for PC and HRPC did not appear in Figures 13 and 14 because their points correspond to large values of buffer space that are beyond the range of values shown on the x-axis.

6 Conclusions

Aggregation computation is an important operation in OLAP systems, and several precomputation and indexing techniques have been developed to expedite processing of such OLAP queries. In this paper, we consider precomputation techniques for processing range-sum queries, and propose a new class of precomputed cubes that is based on a design framework defined by two orthogonal dimensions. In particular, we present two new designs in this class of alternatives: HRC and HBC. Our results show that HBC is the overall winner. HBC has not only significantly better query-update tradeoff than previous cube designs (for both expected- and worst-case performances), but it also can be more effectively buffered; in particular, by using a moderate amount of buffer space, HBC can achieve the same query-efficiency as the most query-efficient solution, but with a significantly reduced update-cost.

As part of our future work, we plan to verify our analytical results experimentally, and also to characterize the optimal tradeoff points (e.g., the graph knee) for HBC.

References

[1] C.Y. Chan and Y.E. Ioannidis. Bitmap Index Design and Evaluation. In *Proceedings of the Intl. ACM SIGMOD Conference*, pages 355–366, Seattle, Washington, June 1998.

[2] C.Y. Chan and Y.E. Ioannidis. Hierarchical Cubes for Range-Sum Queries. Computer Sciences Dept., University of Wisconsin-Madison, February 1999. http://www.cs.wisc.edu/~cychan/hc.ps.

[3] R. Earle. Method and Apparatus for Storing and Retrieving Multi-Dimensional Data in Computer Memory. U.S. Patent No: 05359725, October 1994.

[4] S. Geffner, D. Agrawal, A. Abbadi, and T. Smith. Relative Prefix Sums: An Efficient Approach for Querying Dynamic OLAP Data Cubes. In *Proceedings of the 15th Intl. Conference on Data Engineering*, pages 328–335, Sydney, Australia, March 1999.

[5] C.T. Ho, R. Agrawal, R. Megiddo, and R. Srikant. Range Queries in OLAP Data Cubes. In *Proceedings of the Intl. ACM SIGMOD Conference*, pages 73–88, Tucson, Arizona, May 1997.

[6] T. Johnson and D. Shasha. Some Approaches to Index Design for Cube Forest. *Bulletin of the Technical Committee on Data Engineering*, 20(1):27–35, March 1997.

[7] G. Moerkotte. Small Materialized Aggregates: A Light Weight Index Structure for Data Warehousing. In *Proceedings of the Intl. Conference on Very Large Data Bases*, pages 488–499, New York City, New York, August 1998.

[8] I.S. Mumick, D. Quass, and B.S. Mumick. Maintenance of Data Cubes and Summary Tables in a Warehouse. In *Proceedings of the Intl. ACM SIGMOD Conference*, pages 100–111, Tucson, Arizona, May 1997.

[9] P. O'Neil and D. Quass. Improved Query Performance with Variant Indexes. In *Proceedings of the Intl. ACM SIGMOD Conference*, pages 38–49, Tucson, Arizona, May 1997.

[10] N. Roussopoulos, Y. Kotidis, and M. Roussopoulos. Cubetree: Organization of and Bulk Incremental Updates on Cube Data. In *Proceedings of the Intl. ACM SIGMOD Conference*, pages 89–99, Tucson, Arizona, May 1997.

[11] B. Salzberg and A. Reuter. Indexing for Aggregation. In *High Performance Transaction Processing Systems Workshop*, Asilomar, California, September 1995.

[12] S. Sarawagi. Indexing OLAP Data. *Bulletin of the Technical Committee on Data Engineering*, 20(1):36–43, March 1997.

[13] J.R. Smith, V. Castelli, A. Jhingran, and C.-S. Li. Dynamic Assembly of Views in Data Cubes. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 274–283, Seattle, Washington, June 1998.

[14] J.S. Vitter and M. Wang. Approximate Computation of Multidimensional Aggregates of Sparse Data Using Wavelets. In *Proceedings of the Intl. ACM SIGMOD Conference*, pages 193–204, Philadelphia, Pennsylvania, June 1999.