Partial-Sum Queries in OLAP Data Cubes Using Covering Codes

Ching-Tien Ho   Jehoshua Bruck*   Rakesh Agrawal

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

California Institute of Technology*
Mail Stop 136-93
Pasadena, CA 91125
bruck@paradise.caltech.edu

Abstract

A partial-sum query obtains the summation over a set of specified cells of a data cube. We establish a connection between the covering problem in the theory of covering codes and the partial-sum problem and use this connection to devise algorithms for the partial-sum problem with efficient space-time trade-offs. For example, using our algorithms, with 44% additional storage, the query response time can be improved by about 12%; by roughly doubling the storage requirement, the query response time can be improved by about 34%.

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], also known as data cube [GBLP96]. To build an MDDB from a data warehouse, certain number of attributes are selected. 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 all records with the same combination of functional attributes 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.

We consider a class of queries, which we shall call *partial-sum queries*, that sum over all selected cells of a data cube, where selection is specified by providing a subset of values for some of the functional attributes. Partial-sum queries are frequent with respect to categorical attributes whose values do not have a natural ordering, although they can arise with respect to numeric attributes as well. For instance, consider an insurance data cube with the functional attributes of state, insurance type, and time period, and the measure attribute of revenue. A partial-sum query may obtain the total revenue from the states of California, Florida, Texas, and Arizona, for life and health insurances, and for 1Qtr94, 1Qtr95, and 1Qtr96. 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.

*Research was supported in part by the NSF Young Investigator Award CCR-9457811 and by the Sloan Research Fellowship.

Partial-Sum Problem The one-dimensional partial-sum problem can be formally stated as follows. (The d-dimensional partial-sum problem will be defined in Section 7.) Let A be an array of size m, indexed from 0 though $m-1$, whose value is known in advance. Let $M = \{0, 1, \cdots, m-1\}$ be the set of index domain of A. Given a subset of A's index domain $I \subset M$ at query time, we are interested in getting partial sum of A, specified by I as:

$$Psum(A, I) = \sum_{i \in I} A[i].$$

We will use two metrics to measure the cost of solving the partial-sum problem: time overhead T and space overhead S. The partial-sum computation requires an access to an element of A followed by an addition of its value to an existing value (the cumulative partial sum). Thus, a time step can be modeled as the average time for accessing one array element and one arithmetic operation. We define T of an algorithm as the maximum number of time steps required by the algorithm (over all possible input I's). We define S as the number of storage cells required for the execution of the partial-sum operation. The storage may be used for the original array A and for precomputed data that will help in achieving better response time. Clearly, a lower bound on S is m since at least the entire array A, or some encoded form of it, has to be stored. Without any precomputation, i.e., $S = m$, the worst-case time complexity is $T = m$ (which occurs when $I = M$). On the other hand, if one precomputes and stores all possible combinations of partial sums ($S = 2^m - 1$), which is clearly infeasible for

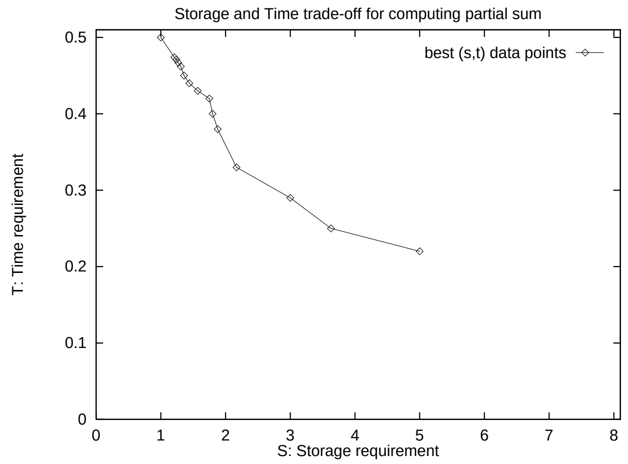

Figure 1: The best (s,t) data points for computing partial sum.

large m, only one data access is needed ($T = 1$). A straightforward observation is that if we precompute only the total sum of A, say $A[*] = \sum_{i=0}^{m-1} A[i]$, then the worst-case time complexity for any partial sum can be reduced from m to $\lceil m/2 \rceil$. This is because a partial sum can also be derived from $A[*] - Psum(A, I')$ where $I' = M - I$. We will consider the normalized measures for time and space. Namely, $s = S/m$ and $t = T/m$. Clearly, using the $A[*]$ we can get $(s,t) \approx (1, 0.5)$.

Contributions The goal of the paper is to derive a suite of (s,t) pairs, better than $(s,t) \approx (1, 0.5)$. In particular, we will focus on finding (s,t) for $t < 0.5$ and s being a small constant (say, less than 5 or so). The best (s,t)-pairs obtained in this paper are summarized in Figure 1. (More detailed (s,t) values are listed in Table 7 later.) For example, the entry $(s,t) = (1.44, 0.44)$ implies that with 44% additional storage, one can improve the query response time by about 12% (i.e., from $t = 0.5$ to $t = 0.44$). Another entry $(s,t) = (2.17, 0.33)$ means that if we roughly double the storage requirement, the query response time can be improved by about 34%.

The main contributions of the paper are as follows. First, we establish the connection between covering codes [GS85] and the partial-sum problem. Second, we apply four known covering codes from [GS85], [CLS86], and [CLLJ97] to the partial-sum problem to obtain algorithms with various space-time trade-offs. Third, we modify the requirements on covering codes to better reflect the partial-sum problem and devise new covering codes with respect to the new requirements. As a result, we further improve many of the (s,t) points and give better space-time trade-offs.

Although we discuss explicitly only the SUM aggregation operation, the techniques presented apply to the other common OLAP aggregation operations of COUNT and AVERAGE — COUNT is a special case of SUM and AVERAGE can be obtained by keeping the 2-tuple (sum, count). In general, these techniques can be applied to any binary operation **op** for which there exists an *inverse* binary operation **iop** such that a **op** b **iop** $b = a$, for any a and b in the domain.

Related work Following the introduction of the data cube model in [GBLP96], there has been considerable research in 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]. However, these works do not directly addresses efficient precomputation techniques for partial-sum queries.

Closest to the work presented in this paper is the accompanying paper [HAMS97], in which we consider *range-sum queries* over data cubes and give fast algorithms for them. A range-sum query obtains the sum over all selected cells of a data cube where the selection is specified by providing *contiguous* ranges of values for numeric dimensions. An example of a range-sum query over an insurance data cube is to find the revenue from customers with an age between 37 and 52, in a year from 1988 to 1996, in all of U.S., and with auto insurance. Although a range-sum query can be viewed as a special case of the partial-sum query (thus the general techniques proposed here can also be applied to the range-sum query), the techniques specialized for range-sum queries take advantage of the contiguous ranges of selection and should be preferred for better performance.

Organization of the paper The rest of the paper is organized as follows. In Section 2, we give a brief background on the covering codes that is pertinent to the partial-sum problem. In Section 3, we give main theorems that relate the properties of covering codes to the space and time complexities in solving the partial-sum problem. In Section 4, we apply the known covering codes to the partial-sum problem. In Section 5, we modify the definition of the covering code by assuming all the weight-1 vectors are included as codewords, in order to derive faster algorithms. In Section 6, we further modify the definition of the covering code based on a composition function. This results in further improvement in space and time overheads in solving the partial sum problem. Section 7 discusses partial-sum queries over multi-dimensional cubes. We conclude with a summary in Section 8.

2 Covering Codes

In this section, we briefly review some concepts from the theory of error correcting codes [GS85] that are pertinent to the partial-sum problem.

A *code* is a set of *codewords* where each codeword defines a valid string of digits. For the purposes of this paper, we are only interested in binary codes of fixed length. We will represent a binary vector in a bit string format and use the terms *vector* and *bit string* interchangeably depending on the context. The bit position of a length-m bit string (or vector) is labeled from 0 through $m-1$ from left (the most significant bit) to right (the least significant bit). Also, $\mathcal{R}^*(V)$ denotes any bit-rotation of vector V and "|" denotes concatenation of two bit strings (vectors).

The *Hamming weight* of a length-m binary vector $V = (b_0 b_1 \cdots b_{m-1})$ is $\sum_{i=0}^{m-1} b_i$, i.e., the number of 1-bits in this vector. The *Hamming distance* of two binary vectors V and V', denoted $Hamming(V, V')$, is the Hamming weight of $V \otimes V'$ where "$\otimes$" is the bit-wise exclusive-or operator. For instance, the Hamming weight of the vector $V = (0010110)$

weight	Vector	the closest codeword	dist.
0	(00000)	itself	0
1	$\mathcal{R}^*$(00001)	(00000) or itself	0 or 1
2	(00)|$\mathcal{R}^*$(011)	(00111)	1
	(01)|$\mathcal{R}^*$(001)	(01000)	1
	(10)|$\mathcal{R}^*$(001)	(10000)	1
	(11000)	(01000) or (10000)	1
3	(00111)	itself	0
	(01110)	(11110)	1
	(11100)	(11110) or (11101)	1
	(11001)	(11011) or (11101)	1
	(10011)	(11011)	1
	(01011)	(11011)	1
	(10110)	(11110)	1
	(01101)	(11101)	1
	(11010)	(11011) or (11110)	1
	(10101)	(11101)	1
4	(01111)	(00111)	1
	(11110)	itself	0
	(11101)	itself	0
	(11011)	itself	0
	(10111)	(00111)	1
5	(11111)	any weight-4 codeword	1

Table 1: The $(5,7,1)$-covering code $\{(00000), (00111), (10000), (01000), (11011), (11101), (11110)\}$. $\mathcal{R}^*(V)$ denotes any bit-rotation of vector V and "|" denotes concatenation of two bit strings.

is 3. The Hamming distance between $V = (0010110)$ and $V' = (0010001)$ is 3, which is the Hamming weight of $V \otimes V' = (0000111)$. Throughout the paper, the *weight* of a codeword or a vector always means the *Hamming weight.*

The *covering radius* R of a binary code is the maximal Hamming distance of any vector of the same length from a codeword (a vector in the code). A binary code C is an (m, K, R)-covering code if (1) each codeword is of length m; (2) there are K (legal) codewords in C (out of all 2^m possible combinations in the vector space); and (3) the *covering radius* of the code is R.

Example The code $C = \{(00000), (11111)\}$ is a $(5,2,2)$-covering code because $m = 5$, $K = 2$ and $R = 2$. For this code, $R = 2$ because every binary vector of length 5 is within distance 2 from either (00000) or (11111). As another example, the code $C = \{(00000), (00111), (10000), (01000), (11011), (11101), (11110)\}$ can be verified from Table 1 as a $(5,7,1)$-covering code because all 32 vectors are within distance 1 from one of the 7 codewords.

3 Relating the Covering Radius of Codes to Partial Sums

We are now ready to relate covering codes to the partial-sum problem.

3.1 Using Covering Codes to Solve Partial Sums

Given a length-m covering code C and any m-bit vector V, we use $f_t(m)$ and $f_s(m)$ to denote the time and associated space overheads, respectively, in deriving the index to codeword in C that is closest to V. Note that $f_t(m)$ and $f_s(m)$ may depend on certain property of the code, in addition to the length of the codeword. However, for notational simplicity, we omit the parameter C in f_t and f_s.

For convenience, we define an m-bit *mask* of I as $mask(I) = (b_0 b_1 \cdots b_{m-1})$ where $b_i = 1$ if $i \in I$, and $b_i = 0$ otherwise. Also, if $V = mask(I)$, then the set I will be called the *support* of vector V, denoted $support(V) = I$. (Support and mask are inverse functions). For instance, if $m = 5$, $I = \{0, 1, 3\}$ then $mask(I) = (11010)$. Also, $support((11010)) = \{0, 1, 3\}$.

Lemma 1 *Given an (m, K, R)-covering code with c codewords of Hamming weight 1 or 0 in the code, we can construct an algorithm to derive the partial sum $Psum(A, I)$ in time $T = R + f_t(m) + 1$ and in space $S = m + K - c + f_s(m)$.*

Proof: Denote the K codewords (vectors) by $V_1, V_2, \cdots, V_K$. Let $I_i = support(V_i)$. Without loss of generality, assume that the c codewords with weight 1 or 0 are the first c on the list. (Thus, the partial sum for each of $I_1, I_2, \cdots, I_c$ is already known as they correspond to entries in array A.) We will precompute and store the partial sums for $K - c$ different subsets specified by $I_{c+1}, I_{c+2}, \cdots, I_K$, respectively. This requires a space overhead of $K - c$. Given an index subset parameter I at run time, let $V = mask(I)$. We first find an index i such that V_i is the closest codeword from V. This requires a time overhead of $f_t(m)$ and a space overhead of $f_s(m)$. Then, we access the precomputed $Psum(A, I_i)$ in one step. Since V is at most distance R away from V_i (due to the property of an (m, K, R)-covering code), the partial sum $Psum(A, I)$ can be obtained from $Psum(A, I_i)$ by accessing and adding or subtracting up to R elements of A, which correspond to the 1-bit positions of $V \otimes V_i$. Thus, the time overhead for this modification is at most R. Overall, we have $T = R + f_t(m) + 1$ and $S = m + K - c + f_s(m)$. □

3.2 Reducing Space Overhead

Recall that array A is of size m. The above lemma applies any covering code of length m to the entire array. However, many covering codes have small R and large K relative to m. Applying these covering codes directly to the entire array typically yields an unreasonable space overhead, even though the time is much improved. Furthermore, the space overhead depends on the array size m. In the following theorem, we will partition the array into blocks of size n and apply length-n covering codes to each block.

Theorem 2 *Given an (n, K, R)-covering code with c codewords of Hamming weight 1 or 0 in the code, we can construct an algorithm to derive the partial sum $Psum(A, I)$ in time $T \approx (R + f_t(n) + 1)\frac{m}{n}$ and in space $S \approx (n + K - c)\frac{m}{n} + f_s(n)$.*

Proof: Assume first that m is a multiple of n. Logically partition the array A into m/n blocks of size n each. Let $x = m/n$. Denote them as $A_0, \cdots, A_{x-1}$. Also partition I into $I_0, \cdots, I_{x-1}$. Then, $Psum(A, I) = \sum_{i=0}^{x-1} Psum(A_i, I_i)$. To derive $Psum(A_i, I_i)$ for each $0 \le i < x$, we apply the algorithm constructed in Lemma 1, which incurs overhead $T_i = R + f_t(n) + 1$ in time and $S_i = n + K - c + f_s(n)$ in space. The space overhead $f_s(n)$ is the same for all i's because the same covering code is applied. Thus, the overall time complexity is $T = \sum_{i=0}^{x-1} T_i = (R + f_t(n) + 1)\frac{m}{n}$ and the overall space overhead is $S = (\sum_{i=0}^{x-1}(S_i - f_s(n))) + f_s(n) = (n + K - c)\frac{m}{n} + f_s(n)$. When m is not a multiple

	Vector	Initial or precomputed value
$[i,0]$	$(10000)^*$	$A[5i]$
$[i,1]$	$(01000)^*$	$A[5i+1]$
$[i,2]$	(00100)	$A[5i+2]$
$[i,3]$	(00010)	$A[5i+3]$
$[i,4]$	(00001)	$A[5i+4]$
$[i,5]$	$(00111)^*$	$A[5i+2]+A[5i+3]+A[5i+4]$
$[i,6]$	$(11011)^*$	$A[5i]+A[5i+1]+A[5i+3]+A[5i+4]$
$[i,7]$	$(11101)^*$	$A[5i]+A[5i+1]+A[5i+2]+A[5i+4]$
$[i,8]$	$(11110)^*$	$A[5i]+A[5i+1]+A[5i+2]+A[5i+3]$

Table 2: The partial-sum look-up table for the i-th block of A based on the $(5,7,1)$-covering code. The codewords of the $(5,7,1)$-covering code are marked with "$*$". Also, (00000) is not needed.

of n, we can extend the array A to a size $m' = \lceil m/n \rceil\, n$ by padding $m'-m$ elements of value 0. This introduces the approximation sign in the complexities of T and S. □

By comparing the time and space complexities of this theorem to that of Lemma 1, it may appear that both time and space complexities are worse in this theorem. Note, however, that R is a function of the vector length (m or n) for a fixed K.

3.3 Implementation Using Look-up Tables

In this subsection, we give a concrete example of implementation based on Theorem 2 and give a general estimate of the time and space overhead ($f_t(n)$ and $f_s(n)$) through the use of look-up tables.

We assume m is a multiple of n. (If not, we can extend the size of A to $\lceil m/n \rceil\, n$ by padding zero elements to A.) First, we will restructure A as a two-dimensional array $A[i,j]$, where i indexes a block, $0 \le i < \lceil m/n \rceil$, and j indexes an element of A within the block, $0 \le j < n$. Thus, the new $A[i,j]$ is the same as the old $A[ni+j]$. Then, for each block i, we precompute the $K-c$ partial sums and store their value in $A[i,j]$ for $n \le j < n+K-c$ in some arbitrary order (though the order is the same for all blocks).

The augmented two-dimensional array A is a *partial-sum look-up table* including the original elements of A (i.e., all n codewords with a Hamming weight 1 for each block) and selected precomputed partial sums for each block of A. Table 2 shows an example of the partial-sum look-up table for the i-th block of A, based on the $(5,7,1)$-covering code described in Table 1. The codewords of the $(5,7,1)$-covering code are marked with "$*$" in the table. Also note that codeword (00000) is not needed in the table because the corresponding partial-sum is 0, which can be omitted. The second column in the table is included for clarity only and is not needed in the look-up table. There are $\lceil m/n \rceil$ such tables, one for each block and each of size $n+K-c$. Thus, a total of size $(n+K-c)\lceil m/n \rceil$ is needed for the partial-sum look-up table.

Second, we will create an *index look-up table* with 2^n-1 entries, indexed from 1 to 2^n-1. For each entry, we store a list of (index, sign)-pairs, denoted $(j_1,s_1),(j_2,s_2),\cdots$, so that the partial sum of the i-th block with vector V can be derived as $\sum(s_x * A[i,j_x])$ for all (j_x,s_x)-pairs defined in the list. Note that the list has at most $R+1$ pairs. Following the same example, Table 3 gives an example of the index look-up table. In the table, an index of "-1" marks the end of the list and a question mark "?" implies a *don't-care* value. As before, the "vector-column" is included here for clarity only and is not needed in the look-up table. Also, it is possible to build the table so that the sign for the first index is always positive (such as the example given) and can be omitted.

Index	Vector	1st index	1st sign	2nd index	2nd sign
1	(00001)	4	+1	−1	?
2	(00010)	3	+1	−1	?
3	(00011)	3	+1	4	+1
4	(00100)	2	+1	−1	?
5	(00101)	2	+1	4	+1
6	(00110)	2	+1	3	+1
7	(00111)	5	+1	−1	?
8	(01000)	1	+1	−1	?
9	(01001)	1	+1	4	+1
10	(01010)	1	+1	3	+1
11	(01011)	6	+1	0	−1
12	(01100)	1	+1	2	+1
13	(01101)	7	+1	0	−1
14	(01110)	8	+1	0	−1
15	(01111)	5	+1	1	+1
16	(10000)	0	+1	−1	?
17	(10001)	0	+1	4	+1
18	(10010)	0	+1	3	+1
19	(10011)	6	+1	1	−1
20	(10100)	0	+1	2	+1
21	(10101)	7	+1	1	−1
22	(10110)	8	+1	1	−1
23	(10111)	5	+1	0	+1
24	(11000)	0	+1	1	+1
25	(11001)	7	+1	2	−1
26	(11010)	8	+1	2	−1
27	(11011)	6	+1	−1	?
28	(11100)	8	+1	3	−1
29	(11101)	7	+1	−1	?
30	(11110)	8	+1	−1	?
31	(11111)	8	+1	4	+1

Table 3: The index look-up table.

As an example, assume the i-th block of I is (00011). We use the value of (00011), which is 3, to index this table. According to the table, the partial sum corresponding to (00011) in the i-block can be derived by $A[i,3] + A[i,4]$. Then, from Table 2, $A[i,3]$ and $A[i,4]$ are pre-stored with values $A[5i+3]$ and $A[5i+4]$, respectively. As another example, assume the i-th block of I is (01011). According to Table 3, the partial sum is $A[i,6] - A[i,0]$, which, according to Table 2, yields $(A[5i]+A[5i+1]+A[5i+3]+A[5i+4]) - A[5i] = A[5i+1] + A[5i+3] + A[5i+4]$.

The size of the index look-up table is bounded by $f_s(n) = O(2^n R)$ from above. When R is large, using a link-list implementation for each entry of the look-up table can improve the size $f_s(n) = O(2^n \bar{R})$ where $\bar{R}$ is the average covering radius of the code.

4 Applying Known Covering Codes

In this section, we will apply some known covering codes to the partial-sum problem based on Theorem 2. Different covering codes lead to different look-up tables and hence different space-time trade-offs. We have chosen (n,K,R)-covering codes with combinations of minimum radius R and minimum number of codewords K, given the length of codewords n. Specifically, we consider four classes of codes: two classes for two different generalizations of Hamming code $(7,16,1)$, one class for the generalization of $(5,7,1)$ code, and one class for the generalization of $(6,12,1)$ code. These are the only codes that yielded useful (s,t)-pairs amongst all the codes we examined.

4.1 The $(7+2i, 16, i+1)$-Covering Codes

It was shown in [GS85] that an (n,K,R)-covering code can be generalized to the class of $(n+2i, K, R+i)$-covering codes for all $i \geq 0$, provided that the (n,K,R) code is linear and *normal* (see the definition in [GS85]). Since the $(7,16,1)$ Hamming code is linear and normal, it generalizes to $(7+2i, 16, i+1)$-covering codes, for all $i \geq 0$.

4.2 The $(n+i, 2^iK, R)$-Covering Codes

An (n,K,R)-covering code can also be extended to an $(n+i, 2^iK, R)$-covering code simply by replicating the same set of codewords 2^i times, each in a copy of the 2^n vectors. Thus, $(7,16,1)$ Hamming code also generalizes to $(7+i, 2^{i+4}, 1)$-covering codes for all $i \geq 0$. However, for many $n \geq 9$, better $(n,K,1)$-covering codes than the naive extension from $(7,16,1)$ are known [CLS86] [CLLJ97]. In particular, $(9,62,1)$ is such a code included in [CLLJ97].

4.3 The $(2R+3, 7, R)$-Covering Codes

The 7 codewords for $(5,7,1)$ code are:

$$\{(00000), (00111), (10000), (01000), (11011), (11101), (11110)\}.$$

The $(5,7,1)$ code is a *piecewise constant code* given in [CLS86]. Through an *amalgamated direct sum technique* [GS85][CLS86], one can generalize $(5,7,1)$ to $(2R+3, 7, R)$ with $c = 2$ for all $R > 1$.

4.4 The $(2R+4, 12, R)$-Covering Codes

The 12 codewords for $(6,12,1)$ code are:

$$\{(000100), (000010), (000001), (100111), (010111), (001111),$$

$$(011000), (101000), (110000), (111011), (111101), (111110)\}.$$

The $(6,12,1)$ code is a *piecewise constant code* given in [CLS86]. Through an *amalgamated direct sum technique* [GS85][CLS86], one can generalize $(6,12,1)$ to $(2R+4, 12, R)$ with $c = 3$ for all $R > 1$.

4.5 Results

The results of applying the above codes to the partial-sum problem are summarized in Table 4. The results show a spectrum of space-time trade-offs and one can choose an operating point depending upon the objective.

5 Single-Weight-Extended Covering Codes

In this section, we will modify the property of covering codes to better reflect the partial-sum problem. We will first define a new type of covering codes, which we shall call the *single-weight-extended covering codes*. Then we present a general theorem relating this type of covering codes to the partial-sum problem. Finally, we will devise a class of covering codes of this type.

n	K	R	c	s	t	ref.
m	2	$m/2$	1	$1+1/m$	0.50	
odd n	7	$(n-3)/2$	2	$1+5/n$	$0.5 - \frac{1}{2n}$	§ 4.3
19	7	8	2	1.26	0.474	§ 4.3
17	7	7	2	1.29	0.471	§ 4.3
15	7	6	2	1.33	0.467	§ 4.3
13	7	5	2	1.38	0.462	§ 4.3
11	7	4	2	1.45	0.45	§ 4.3
9	7	3	2	1.56	0.44	§ 4.3
7	7	2	2	1.71	0.43	§ 4.3
14	12	5	3	1.64	0.43	§ 4.4
12	12	4	3	1.75	0.42	§ 4.4
5	7	1	3	1.80	0.40	§ 4.3
8	12	2	3	2.13	0.38	§ 4.4
11	16	3	1	2.36	0.36	§ 4.1
6	12	1	3	2.50	0.33	§ 4.4
7	16	1	1	3.14	0.29	§ 4.1
8	32	1	1	4.88	0.25	§ 4.2
9	62	1	1	7.78	0.22	§ 4.2

Table 4: Best choices of S and T based on existing covering codes.

5.1 Specialized Covering Codes for Partial Sums

In applying existing (n,K,R)-covering codes to the partial-sum problem in the previous section, we chose codes with combinations of minimum radius R and minimum number of codewords K, given the length of codewords n. Minimizing the time for the partial-sum problem is different from minimizing the covering radius R given length n and K codewords of an (n,K,R)-covering code in two ways. First, the all-0 vector $(00\cdots0)$ need not be covered (since the corresponding partial sum is always 0). Second, the n weight-1 vectors can be included in the covering code without space cost since they are present in array A, which may reduce R.

We, therefore, define the *single-weight-extended covering code*. To derive efficient algorithms for partial sums, our new objective is to derive $(n,K',R)^+$-covering codes with combinations of minimum R and K', for various given small n.

Definition 1 A binary code C is an (n,K',R) *single-weight-extended covering code*, denoted $(n,K',R)^+$-covering code, if (1) each codeword is of length n; (2) there are K' codewords in C; and (3) letting $C' = C \cup \{\mathcal{R}^*(00\cdots01)\}$, i.e., C extended with all n weight-1 vectors, the covering radius of the code C' is R.

Since the all-0 vector is always distance one from any weight-1 vector and $R \geq 1$ for all our cases, covering the all-0 vector (to be consistent with the definition of covering codes) does not increase the complexities of K' and R of the code. Clearly, an (n,K,R)-covering code is also an $(n,K-c,R)^+$-covering code. We will use K' throughout this section to denote the number of codewords excluding the all-0 vector and all weight-1 vectors.

Theorem 3 *Given an $(n,K',R)^+$-covering code, we can construct an algorithm to derive the partial sum $Psum(A,I)$ in time $T \approx (R+f_t(n)+1)\frac{m}{n}$ and in space $S \approx (n+K')\frac{m}{n} + f_s(n)$.*

Proof: Follows from Theorem 2 and Definition 1. □

5.2 The $(2R+3,4,R)^+$-Covering Codes

We now give a construction of a $(2R+3,4,R)^+$-covering code C for all $R \geq 1$ and prove its correctness. The construction is based on a modified version of Figure 5 in [CLS86]. Each codeword has $2R+3$ bits. We will use Y to denote the all-1 vector $(11\cdots1)$ of length $2R-1$ and use Z to denote the all-0 vector $(00\cdots0)$ of length $2R-1$. The four codewords in the $(2R+3,4,R)^+$-covering code are

$$C = \{C_1 = (Z|1111),\ C_2 = (Y|1111),$$
$$C_3 = (Y|1110),\ C_4 = (Y|0001)\}.$$

Theorem 4 *The code C defined above is a $(2R+3,4,R)^+$-covering code.*

Proof: Consider any vector V of length $2R+3$. Partition the vector V into three subvectors, from left to right: V_1 of length $2R-1$, V_2 of length 3, and V_3 of length 1. Let w_1, w_2 and w_3 be the Hamming weight of V_1, V_2 and V_3, respectively. Let W be the set of all length-$(2R+3)$ weight-1 vectors. Recall from Definition 1 that the covering radius of a single-weight-extended covering code is defined with respect to $C \cup W$. Consider the following 3 cases that cover all combinations of V:

Case 1: $w_3 = 0$. If $w_1 + w_2 \geq R+2$ then the Hamming distance of V and $C_3 = (Y|1110)$ is at most $(2R+2)-(R+2) = R$. Otherwise, $w_1 + w_2 \leq R+1$ and there exists a vector in W whose Hamming distance is at most R from V.

Case 2: $w_3 = 1$ and $w_2 \leq 1$. If $w_1 \leq R-1$ then the Hamming distance between V and $(Z|0001) \in W$ is $Hamming(V_1, Z) + w_2 = w_1 + w_2 \leq (R-1)+1 = R$. Otherwise, $w_1 \geq R$ and the Hamming distance between V and $C_4 = (Y|0001)$ is $Hamming(V_1, Y) + w_2 \leq ((2R-1)-R)+1 = R$.

Case 3: $w_3 = 1$ and $w_2 \geq 2$. If $w_1 \leq R-1$ then the Hamming distance between V and $C_1 = (Z|1111)$ is $Hamming(V_1, Z) + (3-w_2) \leq (R-1)+1 = R$. Otherwise, $w_1 \geq R$ and the Hamming distance between V and $C_2 = (Y|1111)$ is $Hamming(V_1, Y) + (3-w_2) \leq ((2R-1)-R)+1 = R$.

□

5.3 Results

Table 5 summarizes the best (s,t)-pairs obtained based on the previous Table 4 and the class of new codes devised in this section.

n	K	R	c	K'	s	t	ref.
m	2	$m/2$	1	-	$1+1/m$	0.50	
odd n	-	$(n-3)/2$	-	4	$1+4/n$	$0.5-\frac{1}{2n}$	§ 5.2
19	-	8	-	4	1.21	0.474	§ 5.2
17	-	7	-	4	1.24	0.471	§ 5.2
15	-	6	-	4	1.27	0.467	§ 5.2
13	-	5	-	4	1.31	0.462	§ 5.2
11	-	4	-	4	1.36	0.45	§ 5.2
9	-	3	-	4	1.44	0.44	§ 5.2
7	-	2	-	4	1.57	0.43	§ 5.2
12	12	4	3	-	1.75	0.42	§ 4.4
5	7	1	3	-	1.80	0.40	§ 4.3
8	12	2	3	-	2.13	0.38	§ 4.4
11	16	3	1	-	2.36	0.36	§ 4.1
6	12	1	3	-	2.50	0.33	§ 4.4
7	16	1	1	-	3.14	0.29	§ 4.1
8	32	1	1	-	4.88	0.25	§ 4.2
9	62	1	1	-	7.78	0.22	§ 4.2

Table 5: Best choices of S and T based on existing and single-weight-extended covering codes.

6 Further Improvements

We now further modify the definition of the covering code by adding a composition function, resulting in a new class of codes, which we shall call *composition-extended covering codes*. The main result (space and time overheads) for the partial-sum problem implied by the new class of covering codes is described in Theorem 6.

6.1 Covering Codes with Composition Function

Let $\ominus$ be the bit-wise *or* operator, $\oplus$ be the bit-wise *and* operator, and $\otimes$ the bit-wise *exclusive-or* operator. Let $\perp$ denote an undefined value.

Definition 2 Define a *composition function* of two binary vectors V and V' as follows:

$$comp(V,V') = V \odot V' = \begin{cases} V \ominus V', & \text{if } V \oplus V' = 0; \\ V \otimes V', & \text{if } V \oplus V' = V \\ & \quad \text{or } V \oplus V' = V'; \\ \perp, & \text{otherwise.} \end{cases}$$

For examples, $comp((001),(011)) = (010)$, $comp((001),(010)) = (011))$ and $comp((011),(110)) = \perp$. The intuition behind this function lies in the following lemma:

Lemma 5 *Let V, V' be two n-bit vectors where $V'' = comp(V,V') \neq \perp$. Also let I, I', and I'' be $support(V)$, $support(V')$, and $support(V'')$, respectively. Then, given $Psum(A,I)$ and $Psum(A,I')$, one can derive $Psum(A,I'')$ in one addition or subtraction operation.*

Proof: By Definition 2, it can be shown that

$$Psum(A,I'') = $$
$$\begin{cases} Psum(A,I) + Psum(A,I'), & \text{if } V \oplus V' = 0; \\ Psum(A,I) - Psum(A,I'), & \text{if } V \oplus V' = V; \\ Psum(A,I') - Psum(A,I), & \text{if } V \oplus V' = V'. \end{cases}$$

□

For consistency, we will let $comp(V,V') = \perp$ if either $V = \perp$ or $V' = \perp$. (All other rules still follow Definition 2.) We assume $\odot$ operator associates from left to right, i.e., $V \odot V' \odot V'' = (V \odot V') \odot V''$. Note that $\odot$ is commutative, but not associative. For instance, $(1100) \odot (1101) \odot (1010) = (1011)$, while $(1100) \odot ((1101) \odot (1010)) = \perp$.

Definition 3 A binary code C is an (n, K'', R) *composition-extended covering code*, denoted $(n, K'', R)^*$-covering code, if (1) each codeword is of length n, (2) there are K'' codewords in C, and (3) every length-n non-codeword vector

weight	Vector	the composition	min distance
1	(0001)	(0111) ⊙ (0110)	2
	(0010)	(0111) ⊙ (0101)	2
	(0100)	(0111) ⊙ (0011)	2
	(1000)	itself	1
2	(0011)	itself	1
	(0110)	itself	1
	(1100)	(1111) ⊙ (0011)	2
	(1001)	(1111) ⊙ (0110)	2
	(0101)	itself	1
	(1010)	(1111) ⊙ (0101)	2
3	(0111)	itself	1
	(1110)	(1000) ⊙ (0110)	2
	(1101)	(1000) ⊙ (0101)	2
	(1011)	(1000) ⊙ (0011)	2
4	(1111)	itself	1

Table 6: The $(4, 6, 1)^*$-covering code.

$V \notin C$ can be derived by up to R compositions of $R+1$ codewords, i.e.,

$$V = C_1 \odot C_2 \odot \cdots \odot C_{i+1},$$

for $1 \le i \le R$, $C_i \in C$.

For example, consider a code $C = \{C_1 = (1111), C_2 = (0111), C_3 = (0110), C_4 = (0101), C_5 = (0011), C_6 = (1000)\}$. It can be verified from Table 6 that this code is a $(4, 6, 1)^*$-covering code.

Clearly, an $(n, K', R)^+$-covering code is also an $(n, K' + n, R)^*$-covering code, but not vice versa. We will use K'' throughout this section to denote the total number of codewords. Note that the code may not contain all weight-1 vectors as codewords. However, in our computer search we minimize K'' first given n and R, then maximize the total number of weight-1 vectors among all minimum-K'' solutions. We were able to find a minimum-K'' solution with all n weight-1 vectors included as codewords for all cases listed below.

Given an $(n, K'', R)^*$-composition-extended covering code C and any n-bit vector V, we will redefine $f_t(n)$ and $f_s(n)$ as the time and associated space overheads, respectively, to find the set of codewords $C_1, \cdots, C_{i+1}$ and its precomputed corresponding partial sums such that $V = C_1 \odot C_2 \odot \cdots \odot C_{i+1}$ where $0 \le i \le R$.

Theorem 6 *Given an $(n, K'', R)^*$-covering code, we can construct an algorithm to derive the partial sum $Psum(A, I)$ in time $T \approx (R + f_t(n) + 1)\frac{m}{n}$ and in space $S \approx K''\frac{m}{n} + f_s(n)$.*

Proof: We first show that given an $(m, K'', R)^*$-covering code C, we can construct an algorithm to derive the partial sum $Psum(A, I)$ in time $T = R + f_t(m) + 1$ and in space $S = K'' + f_s(m)$. We will precompute and store the K'' partial sums of A that correspond to the K'' codewords. Given an index subset I at run time, let $V = mask(I)$. By Definition 3, we can assume $V = C_1 \odot C_2 \odot \cdots \odot C_{x+1}$ where $0 \le x \le R$ and $C_x \in C$. Let $I_i = support(C_i)$ for all $1 \le i \le x+1$. By Lemma 5, we can derive $Psum(A, I)$ by combining $Psum(A, I_i)$'s through addition or subtraction for all $1 \le i \le x+1$. This requires an overhead of $f_t(m) + R + 1$ in time and $f_s(m) + K''$ in space. The rest of the proof is similar to that of Theorem 2 by applying the time and space overhead to each block of A of size n. □

6.2 Lower Bounds on K''

Lemma 7 *Let $S_i \in \{+1, -1\}$, $1 \le i \le x$. If $C_1 \odot C_2 \odot \cdots \odot C_x = V \ne \perp$, then there exists a set of S_i's such that $S_1C_1 + S_2C_2 + \cdots + S_xC_x = V$, where the addition is bit-wise.*

Proof: By Definition 3 and the fact that $V \ne \perp$, we have $C_1 \odot C_2 \in \{C_1 + C_2, C_1 - C_2, -C_1 + C_2\}$. By applying the same argument to the sequence $C_1 \odot C_2 \odot \cdots \odot C_x$, the proof follows. □

Lemma 8 *Let π be a permutation function of $\{1, 2, \cdots, x\}$. If $C_1 \odot C_2 \odot \cdots \odot C_x = V \ne \perp$ and $C_{\pi(1)} \odot C_{\pi(2)} \odot \cdots \odot C_{\pi(x)} = V' \ne \perp$, then $V = V'$.*

Proof: Let $S_i \in \{+1, -1\}$ be the sign associated with C_i in order to derive V, Lemma 7. That is, $\sum_{i=1}^{x} S_iC_i = V$. Let S be the *ordered* set $\{S_1, S_2, \cdots, S_x\}$. Assume that $V \ne V'$. Then, there exists a new ordered set $S' = \{S'_1, S'_2, \cdots, S'_x\}$ such that $\sum_{i=1}^{x} S'_iC_i = V'$ and $S' \ne S$ (i.e., $S'_i \ne S_i$ for some $i \in \{1, 2, \cdots, x\}$). The set S' can be derived from the set of S by changing all different (S_i, S'_i)-pairs. Note, however, that every change of sign from S_i to S'_i will result in a "distance-2" or "distance-0" move of all digits in V. More specifically, the j-th digit with value v will be changed to one of $\{v+2, v, v-2\}$, depending on the j-th bit of C_i. Thus, an even-digit (positive, 0, or negative) remains an even-digit due to the changes of signs. Similarly, an odd-digit (positive or negative) remains an odd-digit. For instance, a 0-digit in V will be changed to one in $\{-2, 0, 2\}$ due to one sign change, while a 1-digit will be changed to one in $\{-1, 1, 3\}$. Since 0 is the only valid even digit of any defined vector and 1 is the only valid odd digit of any defined vector, $V = V'$. □

In the above proof, it is possible that $V = V'$ while $S \ne S'$. In this case, there must be some number of codewords which compose to an all-0 vector.

Theorem 9 *Any $(n, K'', R)^*$-covering code must have*

$$\sum_{i=1}^{R+1} \binom{K''}{i} \ge 2^n - 1.$$

Proof: Follows from Lemma 8. □

Corollary 10 *Any $(n, K'', 1)^*$-covering code must have*

$$\frac{K''(K''+1)}{2} \ge 2^n - 1.$$

Corollary 11 *Any $(n, K'', 2)^*$-covering code must have*

$$\frac{K''(K''^2+5)}{6} \ge 2^n - 1.$$

6.3 Some useful composition-extended covering codes

6.3.1 The $(6, 13, 1)^*$-Covering Code

$$C = \{1, 2, 4, 6, 8, 16, 25, 32, 34, 36, 47, 55, 62\}.$$

This code improves from previous $K'' = K - c + n = 15$ (due to $(6, 12, 1)$-covering code in § 4.4) to 13. The number of weight-1 codewords is 6. The lower bound on K'' is 11, by Corollary 10.

6.3.2 The $(7,21,1)^*$-Covering Code

$$C = \{1,2,4,8,16,24,32,33,38,39,64,$$

$$72,80,91,93,94,95,122,123,124,125\}.$$

This code improves from previous $K'' = 22$ (due to $(7,16,1)$ Hamming code in § 4.1) to 21. The number of weight-1 codewords is 7. The lower bound on K'' is 16, by Corollary 10.

6.3.3 The $(8,29,1)^*$-Covering Code

$$C = \{1,2,3,4,8,16,17,18,19,32,64,76,100,108,128,129,$$

$$130,131,144,145,146,159,183,187,191,215,219,243,251\}.$$

This code improves from previous $K'' = 39$ (due to $(8,32,1)$-covering code in § 4.2) to 29. The number of weight-1 codewords is 8. The lower bound on K'' is 23, by Corollary 10.

6.3.4 The $(9,45,1)^*$-Covering Code

$$C = \{1,2,3,4,8,16,17,18,19,32,36,40,44,64,$$

$$68,96,100,104,128,132,136,140,160,232,236,$$

$$256,257,258,259,272,273,274,287,347,351,$$

$$383,439,443,447,467,471,475,479,499,503\}.$$

This code improves from previous $K'' = 70$ (due to $(9,62,1)$-covering code in § 4.2) to 45. The number of weight-1 codewords is 9. The lower bound on K'' is 32, by Corollary 10.

6.3.5 The $(8,15,2)^*$-Covering Code

$$C = \{1,2,3,4,8,16,32,33,34,64,115,128,191,204,255\}.$$

This code improves from previous $K'' = 17$ (due to $(8,12,2)$-covering code in § 4.4) to 15. The number of weight-1 vectors is 8. The lower bound on K'' is 12, by Corollary 11.

6.4 Results

Table 7 summarizes the best (s,t)-pairs obtained based on the previous Table 5 and the new codes given in this section. Figure 2 shows three sets of data points corresponding to the (s,t)-pairs derived from the existing covering codes, new single-weight-extended covering codes, and new composition-extended covering codes. Figure 1 shows the best (s,t)-pairs combining results from all three types of covering codes, i.e., corresponding to Table 7.

n	R	K''	s	t	ref.
m	$m/2$	-	$1+1/m$	0.50	
odd n	$(n-3)/2$	-	$1+4/n$	$0.5-\frac{1}{2n}$	§ 5.2
19	8	-	1.21	0.474	§ 5.2
17	7	-	1.24	0.471	§ 5.2
15	6	-	1.27	0.467	§ 5.2
13	5	-	1.31	0.462	§ 5.2
11	4	-	1.36	0.45	§ 5.2
9	3	-	1.44	0.44	§ 5.2
7	2	-	1.57	0.43	§ 5.2
12	4	-	1.75	0.42	§ 4.4
5	1	-	1.80	0.40	§ 4.3
8	2	15	1.88	0.38	§ 6
6	1	13	2.17	0.33	§ 6
7	1	21	3.00	0.29	§ 6
8	1	29	3.63	0.25	§ 6
9	1	45	5.00	0.22	§ 6

Table 7: Best obtained choices of S and T based on all techniques.

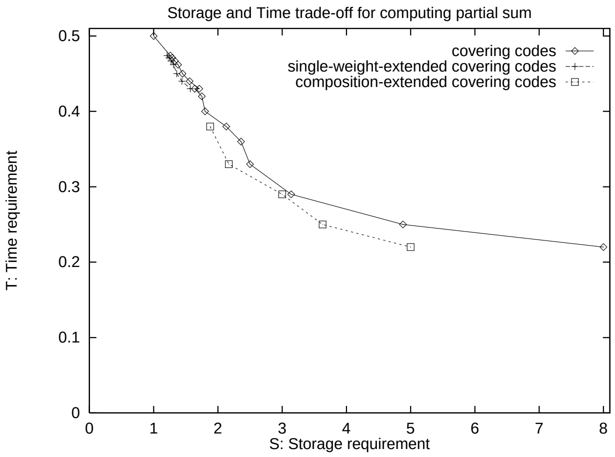

Figure 2: Three types of (s,t) data points for computing partial sum.

7 Partial Sums for Multi-Dimensional Arrays

In this section, we will generalize the one-dimensional partial-sum algorithm to the d-dimensional case. Assume A is a d-dimensional array of form $m_1 \times \cdots \times m_d$ and let $m = \sum_{i=1}^{d} m_i$ be the total size of A. Let M be the index domain of A. Let $D = \{1,\ldots,d\}$ be the set of dimensions. For each $i \in D$, let I_i be an arbitrary subset of $\{0,\ldots,m_i-1\}$ specified by the user at query time. Also let $I = \{(x_1,\ldots,x_d) \mid (\forall i \in D)(x_i \in I_i)\}$. That is, $I = I_1 \times \cdots \times I_d$ and $I \subset M$.

Given A in advance and I during the query time, we are interested in getting partial sum of A, specified by I as:

$$Psum(A,I) = \sum_{\forall(x_1,\ldots,x_d)\in I} A[x_1,\ldots,x_d].$$

7.1 A Motivating Example

Before giving the general d-dimensional algorithm and theorem, we first give a motivating 2-dimensional example. Assume A is a two-dimensional array of form 5×5. Also as-

index	partial sum
$(3,6)$	$A[3,0]+A[3,1]+A[3,3]+A[3,4]$
$(4,6)$	$A[4,0]+A[4,1]+A[4,3]+A[4,4]$
$(3,0)$	$A[3,0]$
$(4,0)$	$A[4,0]$

Table 8: Examples of indexed partial sums in the partial-sum look-up table.

$$\begin{bmatrix} 0&0&0&0&0\\ 0&0&0&1&1\\ 0&0&0&0&0\\ 0&0&0&1&1\\ 0&0&0&1&1 \end{bmatrix} = \begin{bmatrix} 0&0&0&1&0\\ 0&0&0&1&0\\ 0&0&0&0&0\\ 0&0&0&1&0\\ 0&0&0&1&0 \end{bmatrix} + \begin{bmatrix} 0&0&0&0&1\\ 0&0&0&0&1\\ 0&0&0&0&0\\ 0&0&0&0&1\\ 0&0&0&0&1 \end{bmatrix}$$
$$- \begin{bmatrix} 0&0&0&1&0\\ 0&0&0&0&0\\ 0&0&0&0&0\\ 0&0&0&0&0\\ 0&0&0&0&0 \end{bmatrix} - \begin{bmatrix} 0&0&0&0&1\\ 0&0&0&0&0\\ 0&0&0&0&0\\ 0&0&0&0&0\\ 0&0&0&0&0 \end{bmatrix}$$

Figure 3: A pictorial view of $Psum(A,I) = P[3,6]+P[4,6]-P[3,0]-P[4,0]$.

sume that we are applying the $(5,7,1)$-covering code, which is also a $(5,9,1)^+$-single-weight-extended covering code, to each dimension. Denote the 9 codewords by C_0 through C_8, consistent with the order in Table 2. The index look-up table, denoted by X, is still the same as that for the one-dimensional case, Table 3. On the other hand, the partial-sum look-up table will be extended from Table 2 (which has 9 entries) to a two-dimensional table, denoted by P, of 9×9 entries. Then, we will let $P[i,j]$ contain the precomputed partial sum $Psum(A, support(C_i)\times support(C_j))$.

For convenience, we will view each entry of X as a set of (sign, index) pairs. Assume given $I_1 = \{3,4\}$ and $I_2 = \{1,3,4\}$ at query time. We use $mask(I_1)$, which is $(00011) = 3$, as an index to the index look-up table X and obtain $X[mask(I_1)] = \{(+1,3),(+1,4)\}$. Also, we use $mask(I_2)$, which is $(01011) = 11$, as an index to the same index look-up table X and obtain $X[mask(I_2)] = \{(+1,6),(-1,0)\}$. We will show later that $Psum(A,I)$ can be computed as follows.

$$Psum(A,I) = \sum_{\forall\ (s_i,x_i)\in X[mask(I_i)]} \{(\prod_{i\in D} s_i) P[x_1,\ldots,x_d]\}.$$

Following this, we have $Psum(A,I) = P[3,6]+P[4,6]-P[3,0]-P[4,0]$ for our example. Intuitively, the final partial sum $Psum(A,I)$ is derived from combination of additions and subtractions of all "relevant entries" in P, where the "relevant entries" are Cartesian products of different entries indexed by $X[mask(I_i)]$. Table 8 shows the precomputed partial sums corresponding to the 4 terms on the right hand side of the formula. Figure 3 gives a pictorial view corresponding to the formula. In the figure, 1 means a selected value.

7.2 The Main Theorem

We are now ready to prove a lemma for the general case of the above example.

Lemma 12 *Let B be a d-dimensional array of form $n\times\cdots\times n$, and let $Psum(B,I)$ be the partial-sum query. Then, given an $(n,K'',R)^*$-covering code, we can construct an algorithm to derive $Psum(B,I)$ for any I in time $T=(R+1)^d + f_t(n)d$ and in space $S = K''^d + f_s(n)$.*

Proof: Denote the set of K'' codewords by $C = \{C_0, C_1, \ldots, C_{K''-1}\}$. Let $J_i = support(C_i)$. We first construct a d-dimensional partial-sum look-up table, of form $K''\times\cdots\times K''$. An entry indexed by $(x_1,\cdots,x_d)$ in the table will contain precomputed result for $Psum(B,J)$ where $J = J_{x_1}\times\cdots\times J_{x_d}$. Given I at query time, let $I = I_1\times\cdots\times I_d$. Note that in the one-dimensional domain, each I_i can be derived by combining up to $R+1$ existing partial sums. Through an inductive proof, one can show that I can be derived by combining up to $(R+1)^d$ existing partial sums from the partial-sum look-up table. For each dimension, a time overhead of $f_t(n)$ is needed to derive the index of that dimension to the partial-sum look-up table. Thus, the overall time is $T = (R+1)^d + f_t(n)d$. For the space overhead, the partial-sum look-up table is of size K''^d and the index look-up table is of size $f_s(n)$. Since we apply the same covering code to all d dimensions, there is only one index look-up table needed. Thus, the overall space overhead is $S = K''^d + f_s(n)$. □

As in the one-dimensional case, we will now partition array A into blocks of form $n\times\cdots\times n$ and apply covering codes to each block (using the above lemma) in order to derive better space overheads. The proof of the following theorem is straightforward:

Theorem 13 *Given an $(n,K'',R)^*$-covering code, we can construct an algorithm to derive the d-dimensional partial sum $Psum(A,I)$ in time $T\approx(\frac{R+1}{n})^d m + df_t(n)\frac{m}{n^d}$ and in space $S\approx(\frac{K''}{n})^d m + f_s(n)$.*

The above theorem assumes that the same covering code is applied to *all* dimensions of each block and, thus, each block is of form $n\times\cdots\times n$. In general, one can apply different covering codes to different dimensions and obtain a wider range of space-time trade-offs. In this case, the length of each side of the block will be tailored to the length of each covering code applied.

Corollary 14 *Given an $(n,K'',R)^*$-covering code, we can construct an algorithm to derive the d-dimensional partial sum $Psum(A,I)$ in time $T\approx(\frac{R+1}{n})^\alpha(\frac{1}{2})^{d-\alpha}m + df_t(n)\frac{m}{n^\alpha}$ and in space $S\approx(\frac{K''}{n})^\alpha(1+\frac{1}{m})^{d-\alpha}m + f_s(n)$.*

Proof: Apply an $(n,K'',R)^*$-composition-extended covering code to α dimensions and the $(m_i, m_i+1, \lceil m_i/2\rceil)^+$-single-weight-extended covering code to the remaining $d-\alpha$ dimensions. The proof completes by noticing that the latter code has $(s,t)\approx(1,0.5)$. □

7.3 Results

Figure 4 shows various (s,t) data points for computing two-dimensional partial sum based on combination of one-dimensional (s,t) data points from Table 7. The best (s,t) data points are joined together by a curve. Note the leftmost (s,t) data point has been changed from $(1,0.5)$ in Figure 1 to $(1,0.25)$ in this figure.

8 Summary

Partial-sum queries obtain the summation over specified cells of a data cube. In this paper, we established the connection between the covering problem [GS85] in the theory of

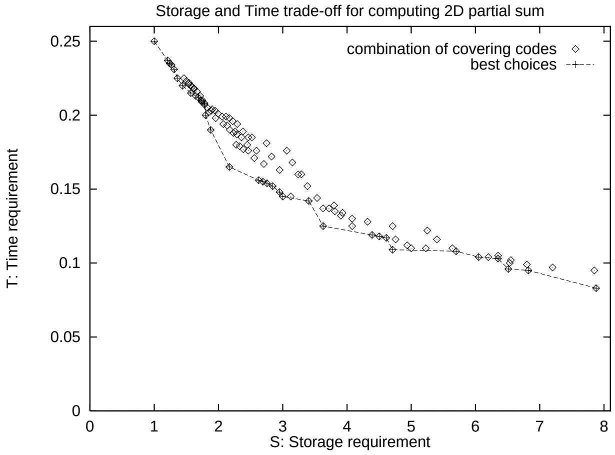

Figure 4: The best (s,t) data points for computing two-dimensional partial sum.

covering codes and the partial-sum problem. We use this connection to apply four known covering codes from [GS85], [CLS86], and [CLLJ97] to the partial-sum problem to obtain algorithms with various space-time trade-offs. We then modified the requirements on covering codes to better reflect the partial-sum problem and devise new covering codes with respect to the new requirements. As a result, we develop new algorithms with better space-time trade-offs. For example, using these algorithms, with 44% additional storage, the query response time can be improved by about 12%; by roughly doubling the storage requirement, the query response time can be improved by about 34%.

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.

[CLLJ97] G.D. Cohen, S. Litsyn, A.C. Lobstein, and H.F. Mattson Jr. Covering radius 1985–1994. *to appear in Journal of Applicable Algebra in Engineering, Communication and Computing, special issue*, 1997.

[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.

[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.

[GS85] R.L. Graham and N.J.A. Sloane. On the covering radius of codes. *IEEE Trans. Information Theory*, IT-31(3):385–401, May 1985.

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

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

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

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

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

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

[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.

[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.