

# Partitioned Blockmap Indexes for Multidimensional Data Access

Kenneth A. Ross and Evangelia A. Sitaridi<sup>\*</sup>  
Columbia University, New York, NY 10025  
{kar, eva}@cs.columbia.edu

Columbia University Technical Report CUCS-006-12

## ABSTRACT

Given recent increases in the size of main memory in modern machines, it is now common to store large data sets in RAM for faster processing. Multidimensional access methods aim to provide efficient access to large data sets when queries apply predicates to some of the data dimensions. We examine multidimensional access methods in the context of an in-memory column store tuned for on-line analytical processing or scientific data analysis. We propose a multidimensional data structure that contains a novel combination of a grid array and several bitmaps. The base data is clustered in an order matching that of the index structure. The bitmaps contain one bit per block of data, motivating the term “blockmap.” The proposed data structures are compact, typically taking less than one bit of space per row of data. Partition boundaries can be chosen in a way that reflects both the query workload and the data distribution, and boundaries are not required to evenly divide the data if there is a bias in the query distribution. We examine the theoretical performance of the data structure and experimentally measure its performance on three modern CPUs and one GPU processor. We demonstrate that efficient multidimensional access can be achieved with minimal space overhead.

## 1. INTRODUCTION

At various levels of the memory hierarchy, modern memory subsystems transfer data in blocked units. Examples include disk blocks from disks or solid-state storage devices, and cache-lines from RAM. In some architectures such as GPUs, the efficiency of coalesced reads (where contiguous data is read in parallel by many threads) also means that access to a block of data can be particularly efficient.

In many scenarios typical of data warehousing and decision support, it is the block transfer bandwidth that is the bottleneck resource. Techniques that reduce the block bandwidth requirements are valuable. For example, a block can be decompressed after the data is in the faster memory, leading to improved bandwidth utilization in exchange for extra computation [20]. Another way to reduce bandwidth is to use a column store, since unneeded columns do not need to be read at all [5].

Bandwidth can also be reduced by the use of suitable indexing techniques. Instead of scanning all records, an index may help the system locate and load only records matching

certain query conditions. The effectiveness of the reduction in bandwidth depends on how well the data is clustered with respect to the indexed conditions.

### 1.1 Prior Work

Two examples of indexing techniques are bitmaps and multidimensional clustering. A bitmap index for an attribute contains one bitmap for each value of the attribute. Alternatively, the attribute may be divided into bins [13], such as by using disjoint ranges of values, with one bitmap for each bin. The length of each bitmap is equal to the number of records in the indexed table. The  $i$ th bit is 1 precisely when the  $i$ th record has a matching attribute value. If binning is used and the query conditions do not exactly match the binning ranges, the underlying attribute must be checked for false positives, which can be a significant portion of the total effort [13]. More sophisticated bitmap schemes are possible [15, 12]. Sparse bitmaps can be compressed into more compact physical representations [17].

Several variants of multidimensional clustering have been proposed [3]. In one variant [11], a set of attributes is chosen in advance, and disjoint ranges for each of those attributes are determined. For example, if there were three attributes, and each attribute was divided into four ranges, there would be  $4^3 = 64$  combinations. The underlying data would be partitioned into 64 pieces based on these combinations. Data is allocated in block units corresponding to many contiguous disk pages to avoid random disk I/O. One dimensional indices on each attribute store block identifiers (rather than record identifiers) for each key. Queries that place conditions on the clustering attributes can potentially be answered using just a few of the data partitions.

A more sophisticated kind of clustering involves the use of space filling curves such as the Z-ordering [3]. Multidimensional data is mapped to a one-dimensional address by interleaving the bits for each attribute range. The address is then used to sort the records, leading to improved locality of reference. A clustered table can be supplemented by an access structure such as a UB-tree [2] that uses the same underlying order [9]. Using the tree structure, only data pages that contain potential matches are read.

Grid files [10] partition the data like multidimensional clustering, with partitions (“buckets”) that are all no bigger than a block. Underutilized neighboring buckets are merged into single blocks. Grid files also provide a multidimensional array of pointers to buckets to enable the rapid location of the corresponding data bucket.

Each of these proposed methods has disadvantages for

---

<sup>\*</sup>Supported by NSF Grants IIS-1049898 and IIS-0915956, and by an equipment gift from Nvidia Corp.

query intensive workloads.

**Bitmap Indices.** In the absence of clustering, bitmaps perform poorly because matching records may be spread uniformly throughout the table. Many blocks (in the worst case, one per matching record) will need to be read. The space required (even with compression) may be high. If many bitmaps need to be created, more space may be needed than the base data [17]. In the presence of clustered data, bitmaps become more compressible [17], but the overhead of decompression must be paid on each access.

**Multidimensional Clustering.** The partitioning of data into disjoint pieces has a potential fragmentation problem if data is partitioned too finely [7]. As a result, the partitioning depth is chosen so that fragmentation is bounded, potentially underutilizing the available partitioning opportunity. If there is significant skew in the data due to attribute correlations or nonuniformly chosen partitioning elements, this kind of partitioning may yield some very large partitions whose potential is not utilized.

**Z-ordering.** Bit interleaving effectively means that the partitioning address space and the partitioning factor in any one dimension must be a power of 2. Since the optimal choices may not be powers of 2, the clustering may be sub-optimal. Addresses need to be explicitly materialized in the index structures based on Z-ordering. Further, structures such as UB-Trees materialize a record pointer for every record. While processing a pointer per record may not be a noticeable overhead in a row-store, it does represent a significant overhead in a column store when only one or two columns are being read.

**Grid Files.** The partitioning elements for grid files are chosen based on the data distribution without regard for the query distribution. As a result, the partitioning may yield suboptimal query performance (see Section 3). Further, correlations between attributes (even correlations in small regions of the data space) can lead to cells with much more data than the mean cell occupancy. In order to get the maximum cell size down to the capacity of a block, a very high degree of partitioning may be necessary, increasing the space requirements for the pointer array.

## 1.2 Solution Overview

On modern architectures with SIMD capabilities, it is relatively efficient to process all records in a block. A large portion of the cost is the initial block loading time, which is independent of how many records are processed for that block. Even so, it only takes a few matching records in a block before record-at-a-time processing becomes slower than a brute-force SIMD approach, as we demonstrate experimentally. Assuming that a clustering scheme does a decent job of concentrating matching records into common blocks, SIMD processing will actually yield a net performance improvement.

Because we plan to process each block in this brute-force fashion, we do not need to record record identifiers (RIDs) in the index structure. Instead, we simply record the blocks that contain at least one matching record. We can therefore index the data much more compactly.

Our proposed solution combines the best elements of prior methods, while avoiding most of their drawbacks. We describe our proposal in stages. For each dimension  $i$  we choose a partitioning factor  $p_i$  so that the range of the dimension is split into  $p_i$  pieces. We would typically choose range end-

points that split the data evenly in that dimension, but alternative choices are possible such as when certain endpoints are common in the query workload [13].  $p_i$  does not need to be a power of 2, and we analyze how to determine good choices for  $p_i$  in Section 3. If  $d$  is the number of dimensions, then  $P = \prod_{i=1}^d p_i$  is the number of  $d$ -dimensional partitions. If  $r$  is the number of rows, and  $b$  is the block capacity we will choose  $P$  to tune the expected number  $D = \frac{r}{bP}$  of data pages per partition. The choice of a suitable value of  $D$  is discussed in Section 4.

Each dimension value can be mapped to a one dimensional partition number, and a row can be mapped to a tuple  $(n_1, \dots, n_d)$ , where  $d$  is the dimensionality of the database. The underlying data is clustered based on  $(n_1, \dots, n_d)$ . The order of rows sharing a partition number vector will be discussed in Section 4.

We define a  $d$ -dimensional array of length  $P$  resembling the multidimensional array of a grid file [10]. Each cell contains the offset of the first record in the data array that maps to the corresponding range. In the event that there are no such blocks, the offset of the first record after the range (according to the physical order) is stored. The end record corresponding to this partition can be inferred as one less than the offset of the next partition. Unlike grid files, partitions can consist of many consecutive blocks. Similar bin-based structures have been used for partitioning data in conjunction with bitmap indexes [18, 4]. However, this past work typically uses much smaller arrays than we propose, and their index structures are much larger.

We also define a collection of bitmaps, typically two or four for each dimension. These bitmaps are used to extend the partitioning precision of blocks within a partition. Unlike previous bitmap indexing techniques, our bitmaps each contain one bit per *block* of data. We shall also attempt to cluster records within a partition so that many of these bits are zero (see Section 4.2). By doing so, we can exploit the additional partitioning potential remaining in large partitions, while using less space than a conventional bitmap.

## 2. PROBLEM FORMULATION

We assume an OLAP, geospatial, or scientific data analysis setting. There are multiple attributes (or dimensions) by which the data can be analyzed using ranges of dimension values, and queries are more important/frequent than updates. Updates are recorded in a separate data structure, and merged into the main data structure in batches.

We assume a column store setting as commonly advocated for query intensive workloads [5]. Some kind of columnar organization (including a columnar organization within disk pages of a row-store [1]) is also needed in order to utilize SIMD instructions. The performance requirements of an index are more stringent in column stores than in row stores: Because less base data is consulted, the overhead of an index structure is more significant.

We assume a memory resident working set, where I/O is not a performance issue. The recent enthusiasm for main memory databases is motivated by the availability of inexpensive machines with many tens of gigabytes of RAM.

The data is assumed to be a collection of points in a  $d$ -dimensional space, with one additional measure attribute that may be aggregated. The data is thus represented as a table with  $d + 1$  attributes. Attribute domains are assumed to be floating point numbers, or discrete domains with ar-

bitrary cardinality. Low cardinality domains require us to limit the degree of partitioning in those dimensions.

Queries correspond to conjunctions of one-sided or two-sided range conditions on one or more dimensions.

We assume all attributes have the same number of rows in a block. If the data types of different attributes have different size, then the block size may not match the physical transfer unit size for all columns. For example, if we have a 4-byte attribute and a 2-byte attribute, and a 64-byte cache line, then we might choose a block size of either 16 or 32 rows. With a choice of 16, a block of the 2-byte attribute occupies half a cache line. With a choice of 32, a block of the 4-byte attribute occupies two cache lines.

### 3. CHOOSING PARTITIONING VALUES

A query can constrain any number of attributes, from 1 to  $d$  where  $d$  is the partitioning dimensionality. (Partitioning plays no role in the performance of unconstrained queries that require a full scan.) A query workload will contain some mix of these kinds of queries, and we need to use the workload to choose appropriate partitioning values to optimize overall performance.

Queries limiting just one attribute are likely to have the most impact on performance, because (all else being equal) they need to read the most data. If  $m$  is the effective partitioning factor for all dimensions (assuming symmetry for this analysis), then it will take  $m^2$  3-attribute queries to generate the same I/O as one 1-attribute query. Based on this observation, we optimize based on the query frequencies of queries that limit a single attribute.

#### 3.1 Single Attribute Partitioning

In the case where query ranges are spread uniformly over the data range, partitioning values that uniformly divide the data range are a good choice. Nevertheless, it is not uncommon for queries to have a biased distribution, such as selecting recent data more often than historical data. Example 3.1 shows how simple approaches based on dividing the data into equal-sized pieces can be suboptimal.

**EXAMPLE 3.1.** *Consider a time dimension in which more recent data is more likely to be queried than historical data, but for which the data is distributed uniformly over time. For the sake of concreteness, let 0 denote the present moment, 1 denote the start time of the data, and suppose that time values are uniform over the range [0, 1]. All queries in the workload are one-sided range-queries of the form “Find and process all rows with time  $\leq t$ ,” for various thresholds  $t$ . Suppose that the query distribution has a cumulative probability distribution function of  $t^{1/2}$ . For example, 50% of all queries will touch only the segment of data at or before time 0.25, reflecting a bias towards accessing the most recent data.*

*Suppose that we have a “partitioning budget” that allows us to define one partitioning element  $p \in [0, 1]$ . The data is split into two pieces, one containing data from time  $\leq p$  (one  $p$ th of the data), and the other containing the remaining data. The optimal value of  $p$  is the one that minimizes*

$$p(p^{1/2}) + (1 - p)^{1/2}.$$

*Some elementary calculus shows that the optimal value of  $p$  is 1/3. It thus pays to split the data nonuniformly into partitions whose size is in the ratio 1:2. The expected cost*

*in that case is about 61.5% of a full table scan; a simple even split would cost 64.6% of a table scan, which is 5% less efficient.*

One can actually generalize this example to any distribution function of the form  $t^\alpha$  and obtain a closed-form solution for any number of partitioning values. The solutions are obtained by solving a set of equations of the form

$$(\alpha + 1)p_i^\alpha - \alpha p_{i+1}p_i^{\alpha-1} = p_{i-1}^\alpha$$

with the boundary conditions  $p_0 = 0$  and  $p_n = 1$  for  $n - 1$  partitioners  $\{p_1, \dots, p_{n-1}\}$ . In the optimal solution,

$$p_1 = \frac{\alpha}{\alpha + 1}p_2, p_2 = \frac{\alpha(\alpha + 1)}{(\alpha + 1)^{\alpha+1} - \alpha^\alpha}p_3, \text{ etc.}$$

Given the potential complexities of query distributions, a closed form optimal solution will not always be achievable. We leave to future work the problem of selecting optimal partitioning elements given an arbitrary distribution.

#### 3.2 Other Query-Dependent Factors

There are several other factors that may influence the choice of partitioning elements. We assume that when a system estimates the benefits of partitioning by an attribute for a query workload, it takes the following effects into account.

##### 3.2.1 False Positive Elimination

Specific data boundaries that exactly correspond to query boundaries can reduce the overall cost by removing the need to re-check the condition for data in the corresponding partition [13]. The re-checking cost can be a significant portion of the total effort [13]. For dimensions of low cardinality, one can avoid re-checks altogether if the domain is completely partitioned.

**EXAMPLE 3.2.** *Consider the query*

```
Select sum(B)
From R
Where A>10 and A<=20
```

*Consider a possible partitioning of R by A into 5 pieces with boundaries at 8, 12, 17, and 21. The first and last pieces ( $A \leq 8$  and  $A > 21$ ) are ignored because they contain no matches. For the middle piece, with  $12 < A \leq 17$ , we can just sum the B column without even reading the A column because we know the A value must match. For the other two pieces with  $8 < A \leq 12$  and  $17 < A \leq 21$  we need to read both A and B to verify whether the A value is in the desired range. Had the boundaries included 10 and 20, we could have answered the entire query without consulting the A column.*

##### 3.2.2 The Horizon Effect

**EXAMPLE 3.3.** *Consider a query workload consisting of queries of the form*

```
Select sum(C)
From R
Where A<V1 or A>V2
```

*that aggregates records at the top or bottom ends of the domain. Even if we were to partition A into two subranges, it would not improve query performance as both subranges would need to be scanned. Only with three (or more) subranges can we hope to reduce the data volume required.*

The lesson from Example 3.3 is that it is not sufficient to limit attention to the benefit that comes from adding one partitioning value. If we did that in Example 3.3, we would conclude that it is not worthwhile to further partition on  $A$ . Instead, we must consider the possibility of adding any number  $m$  of partitioning values at once (within our partitioning budget), and computing the amortized benefit over  $m$  values. The greatest amortized benefit should be used when comparing the candidate attribute with other attributes.

### 3.3 Trade-Offs Between Attributes

Once we have analyzed each attribute as above, we will have some estimate of the marginal benefit of increasing the number of partitioning values for each attribute. The marginal cost is the number of extra partitions in multidimensional space, which corresponds to the product of the number of partitions in the other attributes. Each dimension will have a weight proportional to its query frequency. The attribute that is awarded the next partitioning element is the one with the highest weighted benefit to cost ratio.

*EXAMPLE 3.4. Consider a two dimensional dataset with uniformly distributed queries covering narrow ranges. We have a budget of four partitions to allocate to the two dimensions. We choose the dimension (say  $d_1$ ) with the higher weight ( $w_1$ ) to receive the first partitioning value, which divides the space in half for the first dimension. The benefit of giving a second partitioning element to  $d_1$  is  $T/2 - T/3 = T/6$  where  $T$  is the cost of scanning the entire table. The marginal benefit of giving a partitioning element to  $d_2$  is  $T - T/2 = T/2$ . The marginal costs are 1 for  $d_1$  (one extra partition) and 2 for  $d_2$  (two extra partitions). We thus choose to assign the partitioning element to  $d_2$  if  $w_2 * (T/2)/2 > w_1 * (T/6)/1$ , i.e., if  $w_2/4 > w_1/6$ .*

By assigning partitioning elements in this way, the top-level partitioning balances the needs of the various dimensions. This balance happens without requiring that the partitioning degree in any dimension is a power of 2. When we later look at how to cluster the records within a partition, we will be able to assume that further refining each attribute range has roughly equal impact on the overall performance.

#### 3.3.1 Absolute vs. Relative Costs

Consider a workload consisting of two queries of the form

```
Q1: Select sum(C)
     From R
     Where A>V1 and A<=V2
```

```
Q2: Select sum(C)
     From R
     Where B>D1 and B<=D2
```

Suppose that the distribution of  $A$  ranges (i.e.,  $V2-V1$ ) is such that  $V2-V1$  spans ten percent of the domain of  $A$  values. Suppose that the distribution of  $B$  ranges (i.e.,  $D2-D1$ ) is such that  $D2-D1$  spans one percent of the domain of  $B$  values. Both  $A$  and  $B$  occur equally often in queries, and both are candidates for partitioning. Suppose we have allocated nine uniformly distributed partitioning values (defining ten partitions) to each of  $A$  and  $B$ . The typical cost for a Q1 query is 20% of a scan because the query range spans two  $A$

partitions. The typical cost for a Q2 query is 10% of a scan because the query range will fall within a single partition. Doubling the number of  $A$  partitions to 20 by adding 10 new partitioning values would reduce the cost for a Q1 query to 15% of a scan, while doing the same for  $B$  would yield a cost for Q2 of 5% percent of a scan. Because we optimize the additive cost of queries in the workload, we care about the absolute change in overall performance. Thus, each of these two options is equally beneficial, even though the relative impact on the cost of Q2 is higher.

### 3.4 Correlated Attributes

When attributes are correlated, partitioning is less effective than with independently distributed attributes. Consider the extreme example where  $d$  dimensions are perfectly correlated. Partitioning by a factor of  $p$  in each dimension would yield  $p^d$  partitions, but only  $p$  of them would be nonempty. Such correlations present significant difficulties for pure partitioning approaches such as multidimensional clustering and grid files.

Methods such as UB-trees that index data based on Z-ordering also have problems for such examples. To get the same effective partitioning as an independent data set, addresses must be  $d$  times as long, since only one bit out of  $d$  generates any significant partitioning. Aside from the overhead of representing such long addresses within the UB-tree, searching for data becomes more difficult as the identification of matching address ranges becomes more complex.

Our proposed solution is robust under extreme skew. While we use a grid array as our top-level structure, its depth is limited. Within a partition, we order the data using an interleaved bit pattern as in Z-ordering, but we do not explicitly materialize the Z-order addresses in the on-line structure. The blockmaps in each dimension compactly represent blocks meeting subrange criteria, effectively partitioning the space. To achieve  $p$ -way partitioning we need just  $2 \log_2 p$  bits per block of input. Even for  $p = 2^{20} \approx 10^6$ , this is just 40 bits per block, a small overhead for 64-byte blocks.

## 4. DETAILED DESIGN

We now elaborate on the design initially outlined in Section 1.2. For the space and time analysis below, we initially assume that the data is independently distributed in each dimension, and that dimensions can be arbitrarily finely partitioned (so that there are no heavy hitters, for example). The time and space analyses simplify expressions by removing integer rounding constraints. In practice, integral solutions will need to be derived from the nonintegral solutions described below.

### 4.1 Traversing the Grid Array

The  $d$ -dimensional grid array  $GA$  contains offsets into the main data file indicating the first record from the matching partition. For example, when  $d = 3$ ,  $GA[1, 2, 3]$  points to the start of the partition containing values within the  $i$ th data subrange in dimension  $i$ . Consider a one-dimensional query on the first dimension, where the query range falls entirely within the second range for dimension 1. Then we need to consult all rows between  $GA[2, 1, 1]$  and  $GA[3, 1, 1]-1$ , a contiguous block of data. For an analogous query on the second dimension, we would need to consult all  $p_1$  segments from  $GA[x, 2, 1]$  to  $GA[x, 3, 1]-1$ , where  $p_i$  is the partitioning factor in the  $i$ th dimension and  $1 \leq x \leq p_1$ . For the third

dimension, we would need to consult all  $p_1 p_2$  segments between  $\text{GA}[\mathbf{x}, \mathbf{y}, 2]$  and  $\text{GA}[\mathbf{x}, \mathbf{y}, 3] - 1$ ,  $1 \leq \mathbf{x} \leq p_1$ ,  $1 \leq \mathbf{y} \leq p_2$ . Multidimensional queries can be handled in a similar fashion, limiting more than one of the array indices. For queries that span multiple subranges, we can perform simpler range checks in each partition because all records in the partition are known to satisfy one (or in the case of wholly enclosed subregions, both) of the conditions.

This grid array structure is not symmetric in the dimensions because it takes more grid array accesses to process the later dimensions. One could make the dimensions more symmetric by using a balanced Gray code order for the grid array, but we leave such efforts to future work.

## 4.2 Row Clustering and Blockmaps

Within a partition, we further refine the partitioning determined by the grid array. We add  $k$  bits of precision to some dimensions, conceptually splitting each dimension range in  $2^k$  pieces. For each row in a partition, the bits from each dimension are interleaved (as in Z-ordering) to form a derived subpartition  $s$ , and records are sorted in order of the  $s$  values. Note that the  $s$  values are not explicitly materialized in the on-line structure.

These extra dimension bits can be used to define bitmap indexes called “blockmaps” for all blocks. There would be a “0-blockmap” and a “1-blockmap” for each dimension. The  $j$ th bit of the  $i$ -blockmap is set if block number  $j$  contains a record that maps to subpartition  $i$  in that dimension. When a block is known to contain only values from one subpartition, the block can be skipped for queries that do not overlap this subpartition. Because blockmaps are defined at the block level, some blocks may be marked in both the 0-blockmap and 1-blockmap.

*EXAMPLE 4.1. Suppose we identify 5 dimensions to refine, and construct 5-bit  $s$  values in dimension order  $d_1, \dots, d_5$ . Records within each partition are sorted in  $s$  order. With a uniform distribution of  $s$  values, we expect the first half (roughly) of the blocks in a partition to belong to the first  $d_1$  subpartition, and the second half to belong to the second  $d_1$  subpartition. One blockmap pair is kept for each dimension, yielding ten blockmaps in total. Each blockmap has length in bits equal to the number of blocks in the data set.*

*Suppose that a partition spans exactly 32 blocks. 32 contiguous bits of each blockmap would be relevant for queries to that partition. There may be at most one block out of the 32 with records from both  $d_1$  subpartitions. In such a case, there will be at most 33 bits (out of 64) that are set to 1 in the two  $d_1$  blockmaps. Similarly, there will be four contiguous regions for  $d_2$ , and at most 3 blocks contain records from both subpartitions on  $d_2$ . Thus there will be at most 35 bits (out of 64) that are set to 1 in the two  $d_2$  blockmaps.*

The blockmap indices can be combined using bitwise **AND** and **OR** operators as for conventional bitmaps. Unlike record-level bitmaps, when the **AND** of two blockmaps contains a 1 bit, the corresponding block may contain no matching records. This situation occurs when different records in the same block match the blockmap-defining conditions. With blockmaps, one cannot use the **NOT** operator to find records satisfying the complement of a condition.

To determine a range within a partition, one combines the bitmaps in a suitable way. For example, suppose we have divided the range for an attribute within a partition into two

using a “high-order bit” blockmap  $H$ , and further divide the subranges into two using a “low order bit” blockmap  $L$ . Recall that we keep track of both the zero and one blockmaps, so we actually have four blockmaps  $H_0, H_1, L_0, L_1$ . A one-sided range query that included just part of the first subregion would be specified as  $H_0 \wedge L_0$  where  $\wedge$  represents a bitwise **AND**. A two-sided range query that included parts of the second and third subregions would be specified as  $(H_0 \wedge L_1) \vee (H_1 \wedge L_0)$  where  $\vee$  represents a bitwise **OR**. It is relatively straightforward to construct these blockmap expressions by induction on the number of bits.

Logic minimization can be used to reduce the complexity of the expression in some cases. As an example, consider a query where the blockmap depth is 1, and the query spans the two subregions for a dimension within the partition. The formula above would give  $H_0 \vee H_1$  which simplifies to 1, and means that it is unnecessary to check the blockmaps. Our implementation will not check any blockmaps for examples like this. For depth greater than 1, we have not implemented a complete logic minimization step, so there may be slightly more blockmap processing done than necessary. In general, logic minimization is believed to be intractable [6].

## 4.3 Space Analysis

The space consumed by the grid array, together with the space consumed by the bitmaps, define the overall space requirements of the index structure. We seek to find the balance between the two structures that minimizes the total space. Assuming that the data can be indexed using 4 byte offsets, we need  $4P$  bytes for the grid array, where  $P$  is the partitioning factor achieved by the grid array.

Each partition will have  $D = \frac{r}{bP}$  blocks of data, where  $r$  is the number of rows and  $b$  is the block size. To partition down to block granularity, we thus need an additional partitioning factor of  $D$  via the blockmaps. We need  $2 \log_2 D$  blockmaps, each of size  $\frac{r}{8b}$  bytes.

We want to find the value of  $P$  such that the expression

$$4P + \frac{r}{4b} \log_2 \frac{r}{bP}$$

is minimized. Note that this expression is independent of the dimensionality of the data. Some elementary calculus shows that this expression is minimized when  $P = \frac{r}{16b \log_e 2}$ . At this value of  $P$ , the number of rows per partition is  $16b \log_e 2$ . The overall space consumed in such a configuration is

$$\frac{r}{4b} \left( \frac{1}{\log_e 2} + \log_2(16 \log_e 2) \right)$$

When  $b = 16$  (corresponding to the number of 4 byte floats in a 64 byte cache line), the number of rows per partition is about 177, or roughly 11 blocks. The total space consumed at  $b = 16$  is about 0.61 bits per row.<sup>1</sup> This is dramatically less than the size of the base data, which would consume  $4d$  bytes per row in uncompressed form.

## 4.4 Time Analysis: Cache Misses

The small size of the grid array and blockmaps means that the access structure is likely to fit in the CPU cache. If so, the only significant source of cache misses will be the base data itself. For example, with an L3 cache of size 12MB, one

<sup>1</sup>In practice, we would store either  $2 \lceil \log_2 11 \rceil = 8$  blockmaps or  $2 \lfloor \log_2 11 \rfloor = 6$  blockmaps, corresponding to either 0.68 or 0.56 bits per row.

could fit the combined index structure for  $12 * 8/0.6 \approx 160$  million rows. The uncompressed base data in such a case would occupy 5.8GB with nine 4-byte attributes. Temporal locality in the query distribution, such as when many consecutive queries ask about the same attribute, could also lead to cache-residence of the active part of the index structure, even when the complete index structure exceeds the cache size.

#### 4.4.1 Optimizing Cache Misses

We now analyze the time cost in the situation where the index structures do not fit in the cache. In such a case, we need to pay attention to the order of the dimensions. With a standard row-major order in the grid array, there will be a significant performance penalty incurred by the last dimension. For narrow queries in that dimension, none of the qualifying partitions are contiguous. The impact of that fact is that accesses in the grid array incur more cache misses than they would for the earlier dimensions. Similarly, blockmap accesses may not fully use all bits in a cache line, and may thus incur more cache misses.

We derive estimates of the number of cache misses for one-dimensional queries, on both the first and last dimensions, as a function of the partition size  $D$  blocks. For queries in the first dimension, where all accesses are contiguous, there are only 2 cache misses in the grid array, to find the first and last base data offsets. There are  $\frac{2}{d} \log_2 D$  blockmaps that may be accessed in each dimension (see Section 4.2). The total length of each blockmap consulted is  $DP^{(d-1)/d} = D^{1/d} \left(\frac{r}{b}\right)^{(d-1)/d}$  bits, for a total of

$$2 + \frac{2D^{1/d}}{Cd} \left(\frac{r}{b}\right)^{(d-1)/d} \log_2 D$$

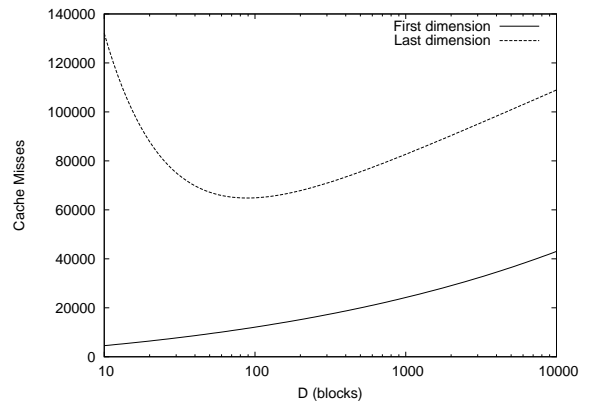
cache lines, where  $C$  is the cache line size in bits.

For the final dimension, we expect to touch every cache line in the grid array. The stride of access is the one-dimensional partitioning factor, which is often smaller than the number of offsets that fit in a cache line. (In the event that there are few dimensions, with a high partitioning factor in each dimension, this analysis is somewhat pessimistic.) Similarly, we expect to touch every cache line in the blockmaps for that dimension, assuming that the number of  $D$ -bit fragments in a cache line exceeds the one-dimensional partitioning factor. The expected number of cache misses in the final dimension is thus

$$\frac{32r}{bDC} + \frac{2r}{bCd} \log_2 D$$

assuming 32-bit offset values.

**EXAMPLE 4.2.** Consider an 8-dimensional dataset with  $2^{28}$  rows. Assume that attribute values and offset values are 4-bytes, and that a cache line is 64 bytes. Figure 1 shows the number of cache misses incurred by the access structure as a function of  $D$ . We consider narrow range queries on the first and eighth dimensions. Note the logarithmic scale on the x-axis. As expected, the misses in the eighth dimension are more significant than those in the first dimension. Even so, the number of misses is small relative to the base data misses. For example, if we were to achieve a single dimensional partitioning factor of 8 ( $8^8 = 2^{24}$ , the number of blocks per column) then scanning one (contiguous) eighth of the rows for two columns of data would yield  $2^{22}$  (about 4



**Figure 1: Access structure misses as a function of  $D$  for the first and eighth dimensions.**

million) cache misses. Further, observe that the number of cache misses is relatively insensitive to the value of  $D$ .

The insensitivity of the access structure misses to  $D$  suggests that a reasonable choice for  $D$  would be the space-optimal value from Section 4.3.

#### 4.4.2 Prefetching

The analysis of Section 4.4.1 ignores an important feature of modern CPU architectures, namely that sequential access to RAM benefits from hardware prefetching. The processing of data and bitmaps in each partition is sequential, meaning that hardware prefetching can hide most or all of the cache misses by overlapping the cache miss latency with other work.

The net effect of hardware prefetching is that the effective cache miss cost of the grid array, becomes a much larger component of the overall cost. The effect is particularly strong for the last dimension because every cache line in the grid array is likely to be touched by a single query. In such a situation, it may be beneficial to choose a different design point in which the grid array fits in the L1 or L2 cache. Queries will be faster because the access time to the grid array will be smaller. The partitions will potentially be larger, and additional bitmaps may be needed to achieve a reasonable partitioning factor. We shall investigate such designs experimentally in Section 5.2.

### 4.5 Skew

As noted above, data skew can arise due to bias in the partitioning values based on the query distribution, or due to correlations between attributes. If there is substantial skew, some partitions may have fewer than their “quota” of blocks, and others may have much more.

Consider the case where a partition has a large number of records corresponding to  $\beta$  blocks of data. As long as we have  $\log_2 \beta$  blockmaps, we can effectively reduce the data from that partition needed to answer queries. This observation leads to a fairly simple design for handling skewed data. After data partitioning, determine the number of blocks  $\beta$  in the largest partition. Ensure that the total number of bits over all dimensions is sufficient to define  $\log_2 \beta$  subregions. One could go even further than this requirement to account for the case that there is skew even within the partition, as discussed in Section 3.4.

The attractive feature of this scheme is that the extra blockmaps needed to handle skew represent a minor overhead. The space overhead is small, typically 2 bits per 16 rows for each blockmap. The time overhead of processing an extra bit is amortized over many rows.

## 4.6 SIMD Processing

The space and time analyses so far have suggested a partition size containing roughly 11 block’s worth of records on average. In the presence of skew, partitions can be much larger. The blockmaps determine which of these blocks contain potentially matching data for the query conditions. Only those matching blocks are processed further. The first and last blocks may contain rows from other partitions, and so those blocks will be processed in a row-at-a-time fashion.<sup>2</sup> If the architecture supports SIMD operators, the remaining blocks are processed using SIMD instructions on every record in the block. Common SIMD instruction sets such as SSE and AVX support vector comparisons (for checking range conditions), vector bitwise operators (for masking out values that do not meet the conditions) and vector addition (for computing a SUM aggregate function).

## 4.7 Disk Blocks

On magnetic or solid state disks, typical block sizes of 4KB correspond to 1024 4-byte values. In the event that an actual I/O is required for a block, that I/O cost would dominate the CPU time needed to process all 1024 values. In practice, though, disk-based databases have in-memory buffer pools where copies of the most frequently accessed pages are kept. For memory-resident disk pages, processing all records in the page may not be time-efficient relative to processing individual records that are known (via a more detailed index) to match.

## 4.8 Updates

Batch updates can be efficiently handled without re-sorting the existing data as long as the partitioning boundaries are unchanged. When a new batch of updates arrives, it is sorted using the same ordering as the main data set. The existing data can then be merged with the sorted updates in linear time. Updates can be processed in-place if some extra space has been allocated in advance at the end of the data, by merging backwards from the end to the start of the data arrays. The grid array and bitmaps can also be computed in linear time.

A linear scan through even a very large memory-resident data set can be performed in seconds. Thus even this “batch” processing model can be tuned so that the indexed portion of the data is current to within a small time window. At a much larger time granularity, new partitioning elements may be desirable to reflect a change in the data distribution, at which time the base data needs to be repartitioned.

## 5. EXPERIMENTAL EVALUATION

The three CPUs used in this performance study are described in Table 1. The AVX SIMD extensions use 256-bit vectors, while the SSE extensions use 128-bit vectors.<sup>3</sup> The

<sup>2</sup>It may be possible to further optimize processing of these blocks, but we do not pursue such optimizations in this paper.

<sup>3</sup>When profiling both AVX and SSE on the 2630QM ma-

Model	RAM	L1-D/L2/L3 cache size	Speed (GHz)	SIMD
Intel 2630QM	6GB	32K/256K/6M	2.0→2.9	AVX
Intel E5620	48GB	32K/256K/12M	2.4	SSE4.2
Sun T2	32GB	8K/4M/—	1.2	—

Table 1: CPU characteristics.

CPU code is implemented in C++ and compiled using g++ at maximum optimization. Time is measured at microsecond granularity using the `gettimeofday()` system call. Times shown correspond to the *complete* query execution time, not the time per row. For the SIMD versions of the code, explicit calls are made to SSE or AVX intrinsics. All CPU code is single-threaded. While the algorithms are parallelizable, scaling will be limited by the memory bandwidth requirements. (Because the Sun T2 relies on parallelism to achieve high bandwidth, one cannot compare the two processors on the basis of their single-threaded performance.)

All columns are represented as 4-byte floating point numbers. A  $d$ -dimensional data set contains  $d + 1$  attributes, including one “payload” column that is always the attribute being aggregated. For a given workload, the sequence of queries is randomized to avoid caching the data. If we did not do so, multiple consecutive queries on the same dimension could artificially improve the temporal locality of the access pattern.

We also developed a GPU implementation of our suggested indexing scheme on an Nvidia CUDA Tesla C2070 machine, which has 32 cores per Stream Multiprocessor (448 total). The columns for the GPU implementation are represented as 4-byte integers. The Tesla C2070 has 6GB of RAM and 144GB/s nominal RAM bandwidth. It has 48KB of L1 cache per Stream Multiprocessor and a 768KB L2 cache.

## 5.1 SIMD Validation

We claimed that replacing some number of individual record checks with a brute-force SIMD check of all records in a block is a net win. This is a central claim, justifying both an improvement in time performance and space performance (since we only store one bit per block in each blockmap).

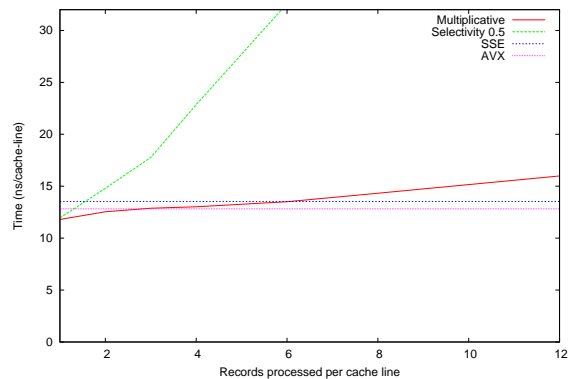


Figure 2: A comparison of aggregation times per cache line for various implementations as a function of the number of records checked per cache line.

Figure 2 shows the performance of several methods on the Intel 2630QM machine. The task involves a comparison, the AVX code was about 7% faster.

ison of one floating point column value with a threshold, and the aggregation of a second floating point column value when the comparison succeeds. The input is a pair of arrays containing 100 million rows. If  $k$  records from a cache line are to be processed, the experiment processes the first  $k$  records in the cache line. The curves labeled “AVX” and “SSE” show the SIMD implementations that process every record in each cache line. The curve labeled “Selectivity 0.5” shows a straightforward implementation based on an `if` test; this implementation suffers from many branch mispredictions at selectivity 0.5. The curve labeled “Multiplicative” avoids mispredictions by using the product of a boolean threshold test with the second column value, and adds the result to the running total. Figure 2 shows that when more than two records qualify in a cache line, the AVX implementation is faster.<sup>4</sup> Even in the worst case where only one record is consulted per cache line, the overhead of the AVX implementation is small.

## 5.2 Initial Experiments

In this section we apply one-dimensional range queries that compute a `SUM` aggregate. Each query restricts one column using a two-sided interval constraint; the interval is randomly located anywhere in the range. The query sums the payload column (for the matching rows) to return a total. As discussed previously, one-dimensional queries are likely to have the greatest performance impact because they consult the most data.

Figure 3 shows performance results for a four dimensional data set containing 11,534,336 rows. 16-way partitioning in each dimension is used in the standard configuration of the grid array because that setting minimizes the size of the access structures. The charts measure the time taken for one-dimensional range queries of various size, on the dimension indicated by the column of the chart. Curves are shown for three implementations: one without blockmap processing, one with one blockmap pair per dimension, and one with two blockmap pairs per dimension. The first two rows show the performance on the Intel 2630QM and Sun T2 platforms without SIMD vector instructions. The remaining rows use SIMD instructions on the Intel 2630QM machine under various settings. The chart labeled “2x underpartitioning” uses a grid array that partitions each dimension into 8 rather than 16. The other overpartitioning and underpartitioning charts are analogous.

Note that with 16-way partitioning one should not expect narrow queries to be 16 (or more) times faster than queries that cover most of the range. Wide queries can process many partitions without even checking the dimension values because the entire partition range is within the query range; only the column being aggregated needs to be consulted.

The first two rows show the performance of the proposed method on the Intel and Sun T2 machines respectively, without the use of SIMD instructions. The blockmaps provide some benefits for the first three dimensions, which are the leading bits in the blockmap representation. Since there are only 11 blocks per partition on average, the blockmaps are not able to reduce the number of blocks consulted for the

<sup>4</sup>The threshold may be even lower, because a practical implementation of the record-at-a-time methods would have additional overheads, including dereferencing some other memory from an index structure to identify the next record to process.

fourth dimension.

The third row shows the same Intel machine running a version of the code with AVX SIMD instructions that are able to process eight data items at once. There appears to be close to a 3–5x speed improvement, in part due to the vectorization, and in part due to the elimination of branch mispredictions [19]. The impact of the blockmaps is now reduced because the per-block cost is lower; the blockmap methods perform about the same as the method that ignores the blockmaps.

For reference, a query that performs a complete scan of the base data, checking range conditions on every record, takes about 6.9ms in the SIMD-enabled implementation on the Intel 2630QM machine. This time corresponds to a very respectable<sup>5</sup> memory bandwidth of about 13GB/sec, benefiting from hardware prefetching during sequential access. Our method is able to achieve a similar bandwidth since most access is sequential, while processing less data.

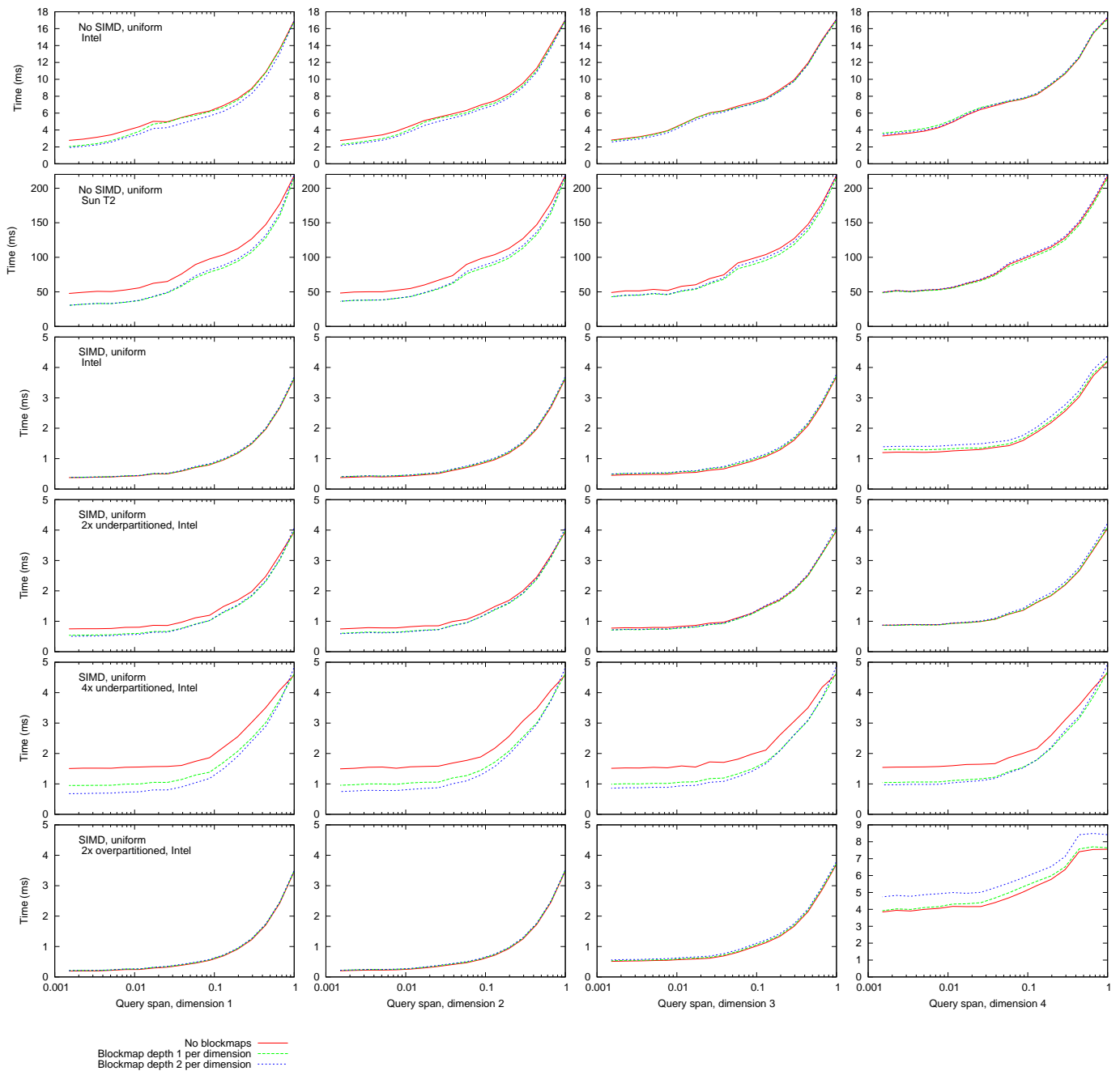
The fourth through sixth rows of Figure 3 show the performance when the grid array uses different one-dimensional partitioning factors, 8, 4, and 32 respectively. The overall performance is robust to slightly smaller partitioning values that lead to larger partitions, as long as the blockmaps are used to make up for the smaller partitioning factor in the grid array. (The fourth dimension benefits from the extra partitioning for reasons explained below.) However, 4x underpartitioning is worse than 2x underpartitioning in all dimensions. Overpartitioning improves performance for the first two dimensions, but dramatically slows performance for the last dimension.

The fourth dimension illustrates the observation made in Section 4.4.2 that L1 and/or L2 misses on the grid array may be a noticeable performance overhead. In the base configuration with 16-way partitioning in the grid array, the grid array occupies 256K bytes, exactly the size of the L2 cache. Thus, even in that configuration we expect to encounter many L1 cache misses, as well as a significant number of L2 misses since other structures will compete for L2 cache space. In the 32-way partitioning case, the grid array is 16 times larger. Aside from the higher cache miss rate, much more data will be read from the grid array to locate more partitions each containing 16 times less data. The “underpartitioning” configurations give the best performance for the fourth dimension because the grid array fits entirely in the L1 cache. It may thus be beneficial to increase space consumption slightly by underpartitioning in order to improve query performance.

## 5.3 Scalability

Figure 4 shows the performance of our algorithms on the Intel E5620 machine for a larger dataset containing 500 million rows. There are six dimensions, each with a partitioning factor of 11. The total data size is 14GB, and the size of each partition is 282 rows on average. When compared with the performance of the same machine on a smaller example ( $11^2 = 121$  times fewer rows and 2 fewer dimensions), we obtain a scale-up factor of 0.63 to 0.68 for a range of queries on all dimensions. This is a reasonably good scale-up factor; linear scaling is unlikely due to the nonresidence of the index structures in the L3 cache for the larger data set.

<sup>5</sup>The rated maximum bandwidth for the machine is 21GB/sec.



**Figure 3: Time performance in milliseconds of various configurations for a four dimensional data set containing 11,534,336 rows. Times shown are the total query processing times and not processing times per result row. A row of graphs corresponds to the four dimensions in a single configuration.**

## 5.4 Skewed Queries

We now consider an example that mirrors the biased-time query distribution from Section 3.1. We construct a 5-dimensional data set with 17.7 million rows and a partitioning factor of 10 in each dimension. The first dimension is the time dimension that is biased in the queries. In particular, time queries are one-sided queries starting at the current time (time 0) and extending into the past a randomly chosen amount, as described in Section 3.1. We consider two values of  $\alpha$  that describe the concentration of the distribution in

the recent past:  $\alpha = 0.5$  and  $\alpha = 0.25$ .

Figure 5 shows the results for queries on the time dimension executed on the Intel 2630QM machine. Approximately 2,000 queries are posed (interleaved at random with the same number of queries on each of other dimensions) and the time taken for each query is shown on the  $y$  axis. The  $x$  axis shows the right endpoint of the query. The lower (green) horizontal line shows the average time of all queries in the workload, while the higher (blue) line shows what the average time would have been had the partition boundaries been chosen uniformly. The partition boundaries in each case can

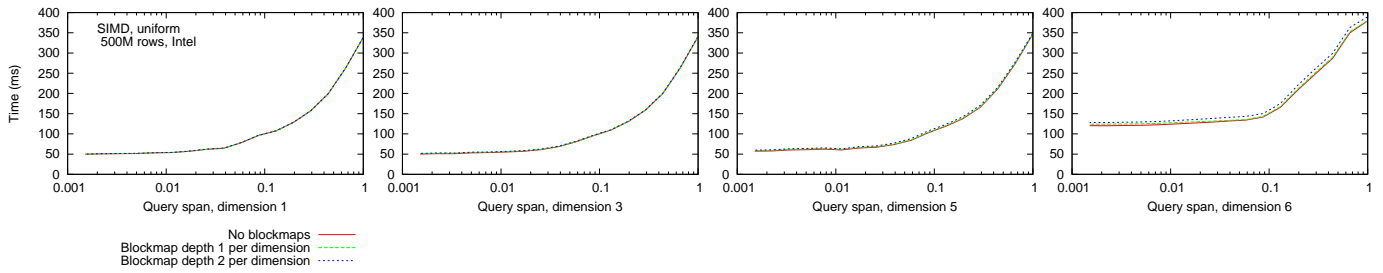


Figure 4: Performance of queries on a 6-dimensional data set containing 500 million rows.

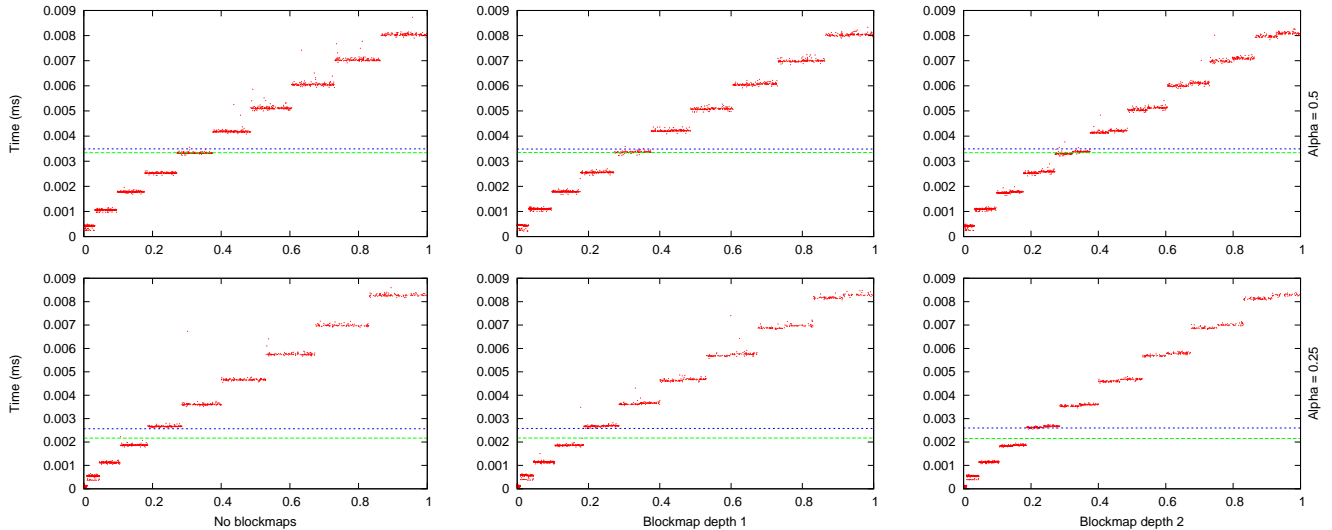


Figure 5: Performance of queries on the first dimension of a 5-dimensional data set. Queries are one-sided, with a biased distribution of right endpoints (shown on the x-axis) to favor data near 0. The degree of query bias is determined by the parameter  $\alpha$ . The underlying data is uniformly distributed in all dimensions.

be inferred from the locations of the step-like discontinuities in the time taken.

The different blockmap settings all yield similar performance. For  $\alpha = 0.5$ , the overall savings relative to uniform boundaries is about 5%. For  $\alpha = 0.25$ , the savings is 17%.

There are some interesting additional features in Figure 5. With a blockmap depth of 1 or 2, there is a slight performance improvement for the left half of a range where the bitmap allows some blocks to be skipped. (The effect is subtle for depth 1.) Further, the impact of the blockmaps is more apparent for the larger ranges towards the right of the chart, where there is more data per partition. For some of the smaller queries, there seem to be two common levels of query performance. The lower level is achieved when the data needed is cache resident. Since there is a one in five chance that consecutive queries are on the same dimension, queries have some chance of benefiting from data loaded into the cache by the previous query.

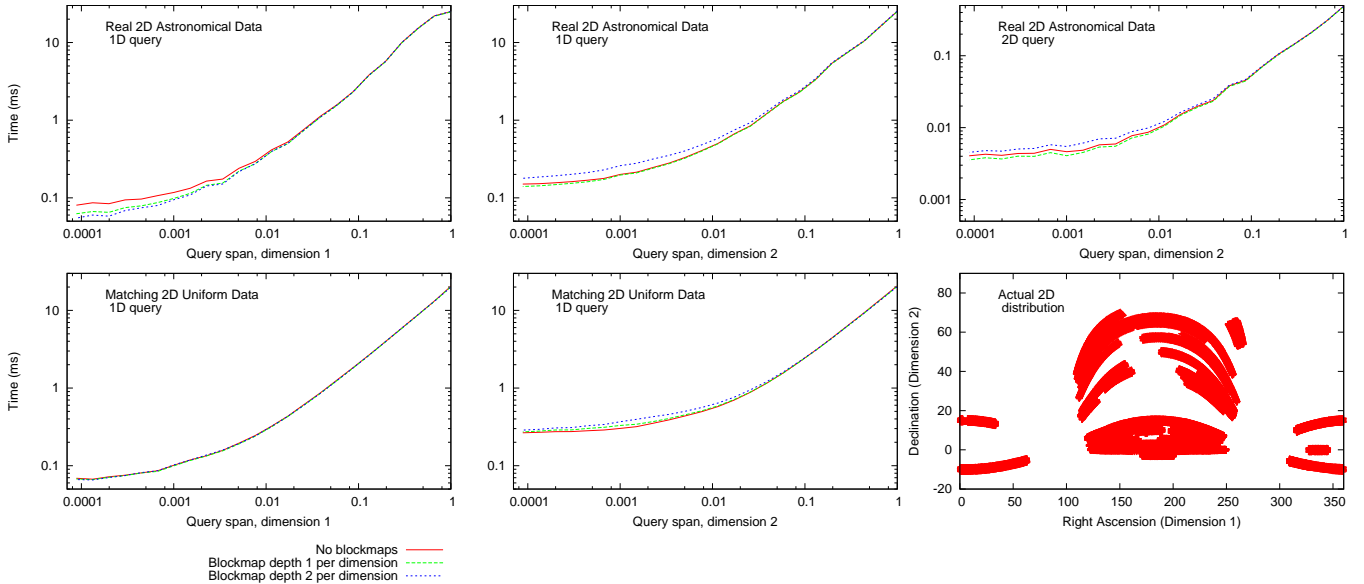
## 5.5 Correlated Real-World Data

Figure 6 shows the performance of our methods on real-world correlated data from the Sloan Digital Sky Survey<sup>6</sup> using the Intel 2630QM machine. We downloaded the first 69,715,000 rows from the PhotoPrimary table, corresponding to a subset of telescope “runs” across the sky. The angu-

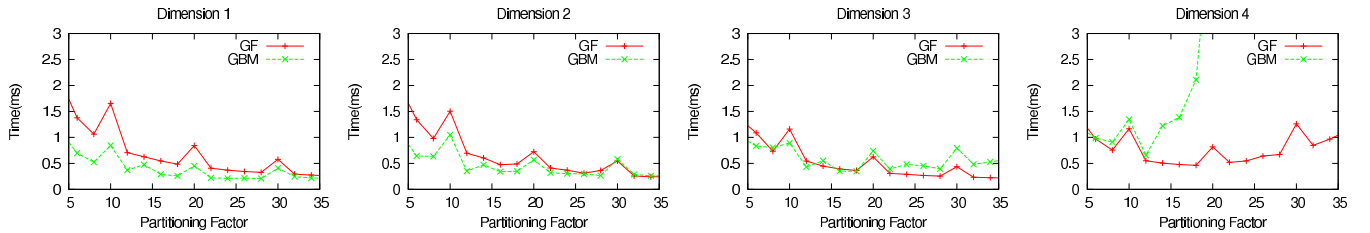
lar coordinates are used as two dimensions, which are correlated in our dataset due to the linear nature of runs; see the bottom right graph in Figure 6. To achieve an optimal partition size, each dimension was partitioned into 624 segments based on the quantiles of the one-dimensional distributions. A matching uniform dataset with the same number of records and partitions was also used for comparison.

For narrow queries, the skew induced by correlation makes the blockmaps more valuable, because they contain extra information that can prevent the processing of some blocks. Blockmaps of depth one perform well in both dimensions. Blockmaps of depth 2 do even better for the first dimension, but worse for the second dimension where the extra work for manipulating more bitmaps does not sufficiently pay off in terms of reduced work. The top right chart shows a two dimensional range query where the first dimension is limited by a predicate with approximately 2% selectivity chosen randomly from the range in such a way that the endpoints exactly match partition boundaries. The span of the second dimension is shown on the x axis. The pattern is similar to that of the one dimensional queries on the second dimension. Note that the time quantities are approaching the resolution of our time measurements (1 microsecond); the resolution is difficult to improve while interleaving different kinds of queries.

<sup>6</sup>[www.sdss.org](http://www.sdss.org)



**Figure 6:** Performance of queries on a two dimensional dataset containing 69,715,000 rows from the Sloan Digital Sky Survey. The data represents coordinate angle pairs of right ascension and declination. Data is obtained in runs, corresponding to sweeps through the sky, and the data used is from a subset of such runs. As shown in the bottom right graph, the resulting data is correlated. The top charts show the performance of queries on the correlated data, while the charts below are for uniform data of matching size. The x-axis span represents the fraction of the data range determined by the smallest and largest actual record values in each dimension.



**Figure 7:** Performance for a single-attribute query, for all dimensions of a 4-dimensional table with 128M rows and varying partitioning factor.

## 5.6 GPU Experiments

Figure 7 shows the performance of our multidimensional index for a 4-dimensional table with 128M rows on the Nvidia Tesla C2070 GPU. The query has a single-dimensional filter with selectivity 0.01, computing a COUNT aggregate. For our GPU implementation the size of a block is 32 records, equal to the footprint of a thread-warp in the CUDA architecture. We assign 256 threads to each thread block, and create 80 thread blocks that are spread across the Stream-Multiprocessors.

We compare the performance of the grid-blockmap of depth 1 (labeled “GBM”) to the performance of a simple grid without blockmaps (labeled “GF”). The estimated optimal partitioning factor per dimension is around 26. To get the same partitioning factor from a grid-only structure the space required would be more than three times higher. Due to the low selectivity, we typically have to consult only one subrange. However, if the condition happens to span two subranges we observe spikes in time performance.

We observe better performance for the first two dimensions using our structure. For the third dimension we benefit

only marginally from using blockmaps when under-partitioning, while performance starts degrading for partitioning factor higher than 16. For the fourth dimension we observe that we get no benefit from using blockmaps and that the performance degradation is dramatic for even moderate partitioning factors. The reason is similar as for our CPU implementation: an increased index processing cost and a higher cache miss rate. One should therefore not use the blockmap on the fourth dimension.

## 6. EXTENSIONS

In this section we describe some potential extensions to the proposed method.

### 6.1 Row Clustering

There is some flexibility in the choice of ordering for the rows within a partition. In Section 4.2 we suggested using Z-ordering. The order in which the bits are concatenated will influence the effective partitioning in the various dimensions. A Gray code order could potentially reduce the number of transitions, leading to more zero bits in the bitmaps. Never-

theless, one must be careful in such a design because certain Gray codes could *increase* the number of one bits. Consider for example a balanced Gray code that equally distributes the bit transitions among attributes. If transitions occur on average more than once per block of records, then all blockmaps will degenerate to being all one bits.

The ordering in consecutive partitions may also be chosen to reduce the number of transitions. Since many queries will need data from several consecutive partitions, it may pay to reduce transitions at partition boundaries by reversing the order in even-numbered partitions. For example, with two bits in each of two dimensions, there would be subpartitions corresponding to  $s$  values of 0, 1, 2, 3. In Gray code order, the  $s$  values would be 0, 1, 3, 2. Consecutive partitions would have subpartitions corresponding to

0, 1, 3, 2, 0, 1, 3, 2, 0, 1, 3, 2, 0, . . .

By reversing the order of even numbered partitions we would get

0, 1, 3, 2, 2, 3, 1, 0, 0, 1, 3, 2, 2, . . . ,

with even fewer transitions across partitions.

One could choose different clustering strategies in different regions of the data to reflect different distributions of data and/or queries. One could track the query count to each region for different types of queries. For example, suppose that data from 2011 is queried most often with a range query on *date*, but that data from 2001 is queried most often with a range query on *sale-amount*. Data partitions for 2011 could choose to use bits from *date* as high order bits for row clustering, while data partitions for 2001 could instead use high order bits from *sale-amount*.

## 6.2 Compression

The notion that a grid array is an effective way of compressing a bitmap dates back to the early work on Grid Files [10]. By using the grid array, we avoid representing the high-order bits of a full bitmap (or blockmap) index. Alternatively, we avoid the decompression overhead of processing a compressed bitmap representation. There may still be some potential to further compress the blockmap bits.<sup>7</sup> However, given the negligible size of these blockmaps relative to the base data, compressing them is unlikely to noticeably impact space consumption. Similarly, while the multidimensional grid array could be delta-encoded to save space, such compression is unlikely to have a major space impact. Even so, compression might be desirable to ensure that components of the index structure fit in a suitable CPU cache level (Section 4.4).

A more important source of space savings would come from compressing the base data itself. Column stores exhibit better compression properties than row stores because data of the same type is stored contiguously [8]. Our clustering of the data could further enhance compressibility. The ideas proposed here could be applied to compressed data representations, with the needed data decompressed on the fly. If decompression is sufficiently fast, performance can improve because of a reduction in the volume of data transferred [20, 14]. Note that parameters such as  $b$  (the number of data values per block) would increase since blocks contain more

<sup>7</sup>For example, it can never happen that both the 0-blockmap and the 1-blockmap contain a zero bit in the same position.

values. It may pay to choose a compression scheme that is sensitive to the proposed access structure. For example, the units of data compression could be chosen to match the partitions defined by the grid array.

## 6.3 Replication

Column stores such as Vertica replicate the base data set, using different sort orders to facilitate different access patterns [16]. Such an organization would be very time-efficient at answering single-dimension queries, because some replica will be sorted by that dimension. Multidimensional queries may not be handled as efficiently. The disadvantage of the replication scheme is that it is not space efficient, requiring  $d$  copies of a  $d$ -dimensional data set. Multidimensional access methods such as the one proposed here try to balance access by multiple attributes while keeping just one copy of the data. If range queries of a certain minimum span are all that is required, then full sorting of the data is not necessary to avoid accessing most of the unneeded data. In such a situation, the space cost of replication may not be worth the benefits.

It may be possible to replicate our proposed structure with multiple, complementary partitioning schemes in such a way that fewer than  $d$  copies are needed to get the desired level of performance. For example, one could have two structures, each partitioning the data set by  $d/2$  of the dimensions. The effective partitioning in each copy would be improved, helping all one dimensional queries, and improving higher dimensional queries for which all restricted attributes are among the  $d/2$  attributes in one of the structures. Many other partitioning schemes are possible. A comprehensive analysis of the time/space trade-offs for such schemes is left to future work.

## 7. CONCLUSIONS AND FUTURE WORK

We have proposed a multidimensional indexing scheme that is both time and space efficient. Fast performance is obtained by combining a grid array with a SIMD-enabled block-oriented scan of each partition. Blockmaps are used to allow the system to ignore blocks that are known not to match. Because blockmaps record only one bit per block, the access structure is extremely compact, typically less than one bit per record. The blockmaps also provide a degree of robustness to data skew.

Some of the ideas developed here could also be applied to other indexing schemes. For example, the query dependent partitions of Section 3.1 could be used to help select boundaries for other index structures.

## 8. REFERENCES

- [1] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and Marios Skounakis. Weaving relations for cache performance. In *VLDB*, pages 169–180, 2001.
- [2] Rudolf Bayer. The universal B-tree for multidimensional indexing: general concepts. In *WWCA*, pages 198–209, 1997.
- [3] Volker Gaede and Oliver Günther. Multidimensional access methods. *ACM Comput. Surv.*, 30(2):170–231, 1998.
- [4] Luke J. Gosink, Kesheng Wu, E. Wes Bethel, John D. Owens, and Kenneth I. Joy. Data parallel bin-based indexing for answering queries on multi-core architectures. In *SSDBM*, pages 110–129, 2009.

- [5] Stavros Harizopoulos, Velen Liang, Daniel J. Abadi, and Samuel Madden. Performance tradeoffs in read-optimized databases. In *VLDB*, pages 487–498, 2006.
- [6] Valentine Kabanets and Jin-Yi Cai. Circuit minimization problem. In *Proceedings of the thirty-second annual ACM symposium on Theory of computing*, STOC '00, pages 73–79, 2000.
- [7] Sam Lightstone and Bishwaranjan Bhattacharjee. Automating the design of multi-dimensional clustering tables in relational databases. In *VLDB*, pages 1170–1181, 2004.
- [8] Roger MacNicol and Blaine French. Sybase iq multiplex - designed for analytics. In *VLDB*, pages 1227–1230, 2004.
- [9] Volker Markl, Frank Ramsak, and Rudolf Bayer. Improving olap performance by multidimensional hierarchical clustering. In *IDEAS*, pages 165–177, 1999.
- [10] Jürg Nievergelt, Hans Hinterberger, and Kenneth C. Sevcik. The grid file: An adaptable, symmetric multikey file structure. *ACM Trans. Database Syst.*, 9(1):38–71, 1984.
- [11] Sriram Padmanabhan, Bishwaranjan Bhattacharjee, Timothy Malkemus, Leslie Cranston, and Matthew Huras. Multi-dimensional clustering: A new data layout scheme in DB2. In *SIGMOD Conference*, pages 637–641, 2003.
- [12] Rasmus Pagh and Srinivasa Rao Satti. Secondary indexing in one dimension: beyond B-trees and bitmap indexes. In *PODS*, pages 177–186, 2009.
- [13] Doron Rotem, Kurt Stockinger, and Kesheng Wu. Optimizing candidate check costs for bitmap indices. In *CIKM*, pages 648–655, 2005.
- [14] Benjamin Schlegel, Rainer Gemulla, and Wolfgang Lehner. Fast integer compression using SIMD instructions. In *DaMoN*, pages 34–40, 2010.
- [15] Rishi Rakesh Sinha and Marianne Winslett. Multi-resolution bitmap indexes for scientific data. *ACM Trans. Database Syst.*, 32(3):16, 2007.
- [16] Vertica Inc. The Vertica analytic database technical overview white paper, 2010.
- [17] Kesheng Wu, Ekow J. Otoo, and Arie Shoshani. Optimizing bitmap indices with efficient compression. *ACM Trans. Database Syst.*, 31(1):1–38, 2006.
- [18] Kesheng Wu, Kurt Stockinger, and Arie Shoshani. Breaking the curse of cardinality on bitmap indexes. In *SSDBM*, pages 348–365, 2008.
- [19] Jingren Zhou and Kenneth A. Ross. Implementing database operations using SIMD instructions. In *SIGMOD Conference*, pages 145–156, 2002.
- [20] Marcin Zukowski, Sándor Héman, Niels Nes, and Peter A. Boncz. Super-scalar RAM-CPU cache compression. In *ICDE*, pages 59–70, 2006.