COMPASS: Compact array storage with value index

Haoyuan Xing
The Ohio State University
xing.136@osu.edu

Gagan Agrawal
The Ohio State University
agrawal.28@osu.edu

ABSTRACT

Efficient array storage is the backbone of scientific data processing. With an explosion of data, rapidly answering queries on array data is becoming increasingly important. Although most of the array storages today support subsetting of an array based on dimensions efficiently, they fall back to full scan while executing value-based filter operations. This has lead to an interest in approximate query processing, but such methods can have substantial inaccuracies.

This paper presents COMPASS, an array storage system with integrated value index support. Our approach efficiently encodes arrays as bin-based indices and corresponding residuals describing elements in each bin. Our query processing method uses bin-based indices, with residuals decompressed as needed, to ensure that accuracy is not sacrificed. Our evaluation shows that compared with current array storage systems such as SciDB, our method achieves a smaller storage footprint, but most importantly, can perform filter operations an order of magnitude faster on low selectivity queries. Meanwhile, COMPASS maintains comparable performance on high-selectivity queries or dimension-based subsetting operations.

CCS CONCEPTS

• **Information systems → DBMS engine architectures**; *Indexed file organization*;

ACM Reference Format:
Haoyuan Xing and Gagan Agrawal. 2018. COMPASS: Compact array storage with value index. In *SSDBM '18: 30th International Conference on Scientific and Statistical Database Management, July 9–11, 2018, Bozen-Bolzano, Italy.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3221269.3223033

1 INTRODUCTION

Our understanding of the universe advances with our ability to obtain and process data [28]. An unprecedented amount of data is being generated in a variety of scientific and engineering domains, from the images of the deep universe [25] to the tracings of particle collisions [21], and from the sequences of human genes [43] to the simulations of global climate changes [48]. Today, projects such as the Large Synoptic Survey Telescope [25] generates terabytes of data every day, pushing the data amount on our hand to exascale and beyond.

Data in many of these domains are naturally represented as a set of *multi-dimensional arrays*, making the efficient storage and navigation of array data a vital challenge. Hence, various array storages [9, 22, 36, 37, 44] have been introduced. Most of these storages rely on a *chunked* structure, storing arrays in small I/O blocks based on its dimensions, to allow efficient execution of operations such as full scan and dimension-based selection (subsetting).

However, scientists and analysts demand more than dimension-based selection. For example, a climate scientist may want to find the grid-cells with higher temperatures. Similarly, one may want to select all the high-energy particles before further analysis or visualization. Such situations require value-based selection or filtering. With the vast amount of data on hand, and the computational power out-growing the I/O capability of the systems today, it is crucial that the filter operation should conserve I/O as much as possible. In addition, because the same dataset is often being analyzed by different teams, at different times, and for different purposes, it is also important that the query range can be chosen flexibly, without relying on our *a priori* knowledge about possible queries.

Options for value-selections on arrays are limited, though. Many existing array storages [9, 36, 44] only store data based on their dimensions, and fall back to scan everything while executing filtering operation. It is not uncommon for users to manually store the filtered results for different query ranges with the goal of accelerating later analytics. This creates unnecessary data duplication, and requires the query ranges to be known ahead of time. *External indices* are another option – bin-based bitmap indices in particular, are widely used [8, 16, 23, 52, 53]. However, retrieving the original values based on its bitmap indices is prohibitively expensive; hence, the bitmap index is often seen as a lossy compressed representation of the data, and used for executing query approximately without referring the original data [42, 50, 57]. As a result, the types of the queries such indices can answer are limited. Also, these methods often rely on the binning strategy and our assumption about the distribution of the query range to achieve higher accuracies [50]. Moreover, because the exact data could not be recovered, it is not possible to pipe the filtered data for further analyses.

Overall, there are advantages of using external indices as a representation for scientific data, provided that the loss of accuracy can be avoided, the original data can be recovered, and the I/O overhead is acceptable. One way to achieve this is integrating the index into the storage. This paper presents the design of COMPASS (COMPASS Array Storage System), an array storage with support for compact and integrated value indices. In addition to the traditional *plain* datasets, COMPASS also supports storing and querying data in *indexed datasets* – specifically, the values are indexed using a number of bins specified by the user, while also storing necessary information to recover the original dataset. It

SSDBM '18, July 9–11, 2018, Bozen-Bolzano, Italy
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-6505-5/18/07...$15.00
https://doi.org/10.1145/3221269.3223033

enables methods for efficient filtering, while reporting accurate results regardless of the binning strategy being used. In addition, because original data can be recovered, the filtered result can be aggregated, visualized, or used as the input for subsequent analysis step(s). Furthermore, this representation almost always occupies same or less storage size as the original dataset.

The basic idea of our approach is as follows. The data in an array are reorganized according to the bins it belongs. Each bin tracks the positions of the values in it with a compressed *positional index*. A bin also keeps a *residual data* segment, so that the actual values in a bin can be recovered. COMPASS utilizes the range of a bin as extra information to compress the *residual data*, reducing the storage redundancy. Moreover, it implements a chunk-based storage scheme, ensuring its quick subsetting ability.

We experimentally evaluate our implementation using both synthesized and real data. Our results indicate that, compared with the original dataset, our method usually achieves a positive compression ratio; it can perform accurate filtering operations by an order or more of magnitude faster when the selectivity is low, and still outperforms the operations on plain dataset even when the selectivity is relatively high. We observe a similar performance advantage over the popular SciDB array database.

This paper makes the following contribution to the scientific data management community:

- We introduce an efficient array storage technique, integrating value index with a number of user-specified bins into the storage, facilitating fast and accurate filtering operations. In the process, we propose multiple options for efficient encoding of the value index and its complementary data.
- We present COMPASS, an array storage system with support for datasets both with and without value indices. We show an access method suitable for an indexed dataset, and how to implement common array operations on top of it.
- Our extensive evaluation demonstrates data storing in an indexed form can often save the storage footprint, frequently outperforms performing a full scan on ordinary datasets even when selectivity is high, and can be effectively converted back to its ordinary form if necessary.

The remainder of this paper is organized as follows. We lay out necessary background concepts of COMPASS in Section 2. Section 3 discusses the storage organization of an indexed dataset. Section 4 follows by describing the design of the access methods in COMPASS and how queries are executed on top of it. We experimentally evaluate the performance of COMPASS in Section 5, discuss related work in Section 6 and conclude the paper in Section 7.

2 BACKGROUND

This section lays out necessary concepts related to COMPASS. This includes the array data model, how bitmap indices and inverted lists can be used to encode the positions of elements in a dataset, and background information on the floating-point number format and compression.

2.1 Array data model

The central concept of the array-oriented data model is *datasets*. Each dataset is a multi-dimensional indexed array, reflecting a one-to-one map from an n-dimensional vector (dimensions) to one value. We refer to n as the rank of the array, and the n-dimensional vectors as the *coordinates*. Some of the coordinates might have no corresponding values. These positions are represented as *empty values*.

For I/O and subsetting efficiency, a dataset is often stored and processed in the granularity of *chunks*. Most array storages today employ the *regular chunking strategy* [39], which divides a dataset into multiple equal-sized hyper-rectangular chunks according to the coordinates of elements. The elements in a chunk are stored contiguously, with optional compression being applied to save I/O.

2.2 Bitmap indices and inverted lists

Both bitmap indices and inverted lists provide a way to index records in a column. Given N records, a bitmap index uses N bits to indicate whether each of these records satisfies a certain condition. It has been used to accelerate queries in relational databases [14, 15, 34, 53, 54] as well as on scientific datasets [8, 16, 23, 42, 46, 47, 50, 52, 57].

A bitmap is usually stored and processed in its compressed form for efficiency. Popular methods include WAH [55], EWAH [33] and Roaring [31]. Most methods utilize the fact that there are often contiguous chunks of `0`s and `1`s in a bitmap. Hence, instead of storing these *0-fills* and *1-fills* as *literal words*, they can be stored as a special *RLE word*. We call an uncompressed bitmap a *bitvector*.

Another method of indexing the records is storing the record IDs that satisfy the index condition in an *inverted list*. However, storing the IDs literally is not desirable due to its space overhead. Instead, assuming the records IDs are increasing monolithic integer sequences, the delta of adjacent IDs ('gap') can be stored. In the case that the delta can be negative, *zig-zag encoding* could be used to transform signed integers to unsigned integers ($0 \rightarrow 0, -1 \rightarrow 1, 1 \rightarrow 2...$). This converts the problem of storing inverted lists to compressing small integers. One of the most simple, yet effective, methods is *variable bytes* [60], which uses a number of bytes to encode a value, and a certain number of bits to indicate the number of bytes being used. The state-of-the-art methods are block-based ones, such as *PFor* [32, 60], PFor encodes the numbers in relatively large blocks such as 64 or 128, and uses bx bits to encode each number in the block. If a number could not be encoded using bx bits, a special marker is placed in the place, and the number is attached at the back of the block. PFor can achieve a high decoding speed yet preserve a high compression ratio with SIMD vectorization [32].

It is not practical to keep a bitmap or inverted list for every unique floating number in the dataset. Thus, some sort of binning is needed. The simplest binning method is *equi-width binning*, which divides the dataset to bins containing the equal intervals of the value domain. Another method, *equi-depth binning* divides the dataset to bins with equal numbers of elements and generally are more accurate in approximate aggregations, but are much slower to build. Some more complex method [50] assumes *a priori* knowledge of the query frequency, hence are often not suitable in the case which query condition and range can change.

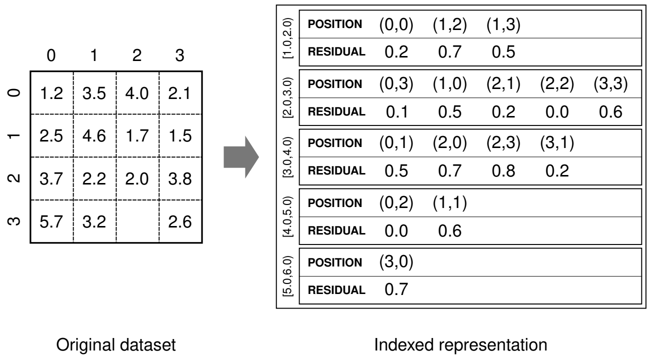


Figure 1: COMPASS reorganizes a dataset into a number of *bins*. A *positional index* tracks the positions of values in each bin. The *residual data* store the extra data necessary for recovering the original dataset.

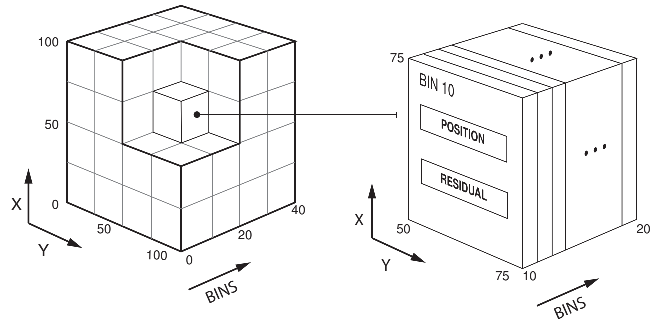


Figure 2: An example of the chunking strategy of COMPASS. The example dataset here has 100×100 elements, with 40 bins in total. A chunk of it contains 10 bins in a rectangular region of 25×25.

2.3 Floating-point number and compression

IEEE 754 [27] is the prevailing standard of floating-point representation. It describes a finite floating number using three components, a sign bit s, an exponent q, and a significand c. These three parts are concatenated in a binary form $s||q||c$, and represent the numerical value $(-1)^s \times c \times 2^q$. We always choose the exponent q to be the smallest exponent allowing the value to be represented exactly to ensure a unique binary representation. Also, the leading bit of the significand is always 1 and is represented implicitly.

The IEEE floating-point format has a nice property of maintaining the *lexical order*. That is, if the binary representations of two floating-point numbers f_a and f_b are viewed as *sign-magnitude* integers s_a and s_b, it does not change its partial order: $f_a < f_b \Leftrightarrow s_a < s_b$. This property shows the floating-point numbers within a small range are likely to contain the same higher bits, which is vital for our storage scheme described in Section 3.

General floating-point streams are hard to compress. Most successful methods use a combination of differentiation and predictors. FPC [10] uses two hash-based predictors, *fcm* and *dfcm* to predict the next value based on the previous values and their deltas. The prediction is then compared with the actual values, and a residual is computed. FPC then uses a modified *variable bytes compression* scheme to compress the residual values. In practice, general compressors, such as the Lempel-Ziv compressor family [58], are often used to compress the chunk content of a dataset.

3 DATA STORAGE

This section discusses the novel value-indexed storage format of COMPASS.

3.1 Indexed representation of an array

COMPASS organizes array data using a bin-based index: values are grouped into a number of *bins* and their positions are stored in the *positional index* of each bin. Because multiple unique values exist in each bin for scientific data, the positional index only provides an approximate picture of the dataset, and extra information is needed to reconstruct the dataset losslessly. COMPASS stores this information as the *residual data* of each bin.

Figure 1 illustrates how a dataset is stored in this representation: the values are divided into five bins based on the range, each bin stores the positions of the values in it, as well as the residual data (here shown as the delta of the value and the lower bound of the bin, we discuss the actual representation in §3.5). Using this representation, a query only needs to retrieve the bins that contains the elements of interest and then reconstructs the elements using the positions and residuals, saving both I/O and processing time. Empty elements are represented implicitly, not stored in any bins, allowing an indexed dataset to handle sparse datasets efficiently.

Furthermore, as in prior work on array storage, COMPASS divides an indexed dataset to small fragments called *chunks*, which are the basic I/O and processing units of the dataset. Both the chunking strategy and the representation of a chunk needs to be adjusted to store an indexed dataset efficiently. We discuss these two important problems later in this section.

3.2 Binning and chunking strategy

Given a value domain, there can be many binning methods. This paper only considers range-based bins, that is, if a dataset has a value-domain of $[a_0, a_m)$, we choose $m-1$ points $a_1, a_2, \ldots, a_{m-1}$ in the domain, thus dividing the data into m bins. The i^{th} bin b_i contains values in $[a_{i-1}, a_i)$ range.

For efficient subsetting operation, a dataset is still divided into n hyper-rectangle *segments* of equal size according to its dimensions using *regular chunking* (§2.1). Combined with m bins, this creates a total of *nm slices*. Consider there can be hundreds of bins, and some bins might contain only a few elements, COMPASS groups these slices together into *chunks* for better I/O efficiency. However, storing all m bins of a segment inside a chunk causes any value-based selection to load all bins from disk, defying the point of indexing. Hence, we can group a few bins together, so that each chunk contains the data of k bins in a segment. Figure 2 demonstrates this chunking strategy.

An *indexed chunk* can be uniquely identified by its boundary coordinates and the bin(s) it contains. A tree-based chunk map stores the address of each chunk. As the number of chunks is usually not very large, COMPASS keeps this map in the main memory.

3.3 Layout of an indexed chunk

An indexed chunk contains three segments: the *residual data* and the *positional index* as described earlier, and the *chunk header* for bookkeeping purposes. One question is the storage layout of the positional indices and residential data inside a chunk. Our design decision comes from the observation that positional indices are not always necessary for query processing. For example, an aggregation operator can select all the elements inside a chunk, and thus does not need the positional indices to proceed. Therefore, layout-wise, the positional indices and residual data of all bins in the chunk are combined into two separate segments. This way, when the positional index is not needed, only the residual data segment is loaded, sparing the extra I/O for loading the positional indices.

3.4 Positional index

For each element in a bin, the positional index marks its dimensional positions within the segment. To achieve this, the multiple-dimensional coordinates of the elements are first serialized to linear offsets. The offsets can then be stored in a compressed form, either as a bitmap, or using a monolithic sequence inverted-list compression scheme such as PForDelta [32, 60]. Both methods have their merits. Bitmaps are generally more efficient in executing approximate queries based on set intersection [50]. However, decoding the offsets from a bitmap requires tracking how many bits have already been visited, therefore brings extra CPU overhead. The storage footprints of the two methods can be affected by the dataset and binning strategy. We investigate it in Section 5.2.

3.5 Residual data

The residual data of a bin store additional data necessary to distinguish different values in the same bin. This can be seen as a compression problem: how to losslessly compress the values in a bin given its range $[lb, ub)$. Simply applying general or scientific data compression algorithms to a bin of values is often sub-optimal, as the compressor is unaware of the bound of the values. Here, we present a general framework of compressing range-limited data.

We compress the data in two steps: *map* and *encode*. The *map* phase removes the redundant information from the input value stream, mapping the data into a stream of integers with the same bit-length, making it easier to compress. The *encode* phase takes the input stream generated by the *map* phase and compresses it into a more condensed representation.

Map There are several options for performing the *map* operation:
none. Do not do any transformation, the rationale here is the data inside the same bin probably has some similarity, therefore, should be already easy to compress.
prefix removal. Recall that if viewed as *sign-magnitude* integers, IEEE floating-point numbers keep the lexical order of their binary representations. Therefore, if the binary representations of lb and ub have a common prefix, all the values inside the range have the same prefix. The length of the common prefix is determined by the numbers of floating numbers between the bounds. Removing this prefix converts the float values to smaller integers.

Algorithm 1: Unsigned flip.

```
function unsigned_contiguous_map(lb, ub, value):
    return $uflip(value) - uflip(lb)$;
end
function uflip(value):
    if $value \geq 0$ then
        return flip the sign bit of $value$;
    else
        return $\sim value$;
    end
end
```

Algorithm 2: The BFPC data compression.

```
function bfpc(data, lb, ub, table_size, lshift, rshift,
  dlshift, drshift):
    $len \leftarrow$ number of leading zero bits of $lb$ and $ub$;
    $rshift \leftarrow min(rshift - len, 0)$;
    $drshift \leftarrow min(drshift - len, 0)$;
    initialize bit vector $selector$;
    initialize integer array $residual$;
    initialize $fcm$ predictor $fpredictor$ using $table\_size$,
      $lshift$ and $rshift$;
    initialize $dfcm$ predictor $dpredictor$ using $table\_size$,
      $dlshift$ and $drshift$;
    for $i = 1$ to $data.size()$ do
        $fpredict \leftarrow$ predict value from $fpredictor$;
        $dpredict \leftarrow$ predict value from $dpredictor$;
        $xor[0] \leftarrow fpredict$ xor $value[i]$;
        $xor[1] \leftarrow dpredict$ xor $value[i]$;
        $residual[i] \leftarrow min(xor[0], xor[1])$;
        $selector[i] \leftarrow xor[0] \leq xor[1]$ ? 0 : 1;
        Update the predictors using $value[i]$;
    end
    $compressed \leftarrow$ compress $residual$ using PFor;
    return $selector \parallel compressed$;
end
```

unsigned flip. The problem of *prefix removal* is floating point numbers are represented in a sign-magnitude fashion. Therefore, although the values ±0.000001 only have a small difference, their binary representations have no common prefix bits. *unsigned flip*, shown in Algorithm 1, solves this problem by mapping all the floating point numbers to the contiguous range of unsigned integers according to their value. This could be done by flipping the sign bits of the positive numbers, and all bits of the negative numbers. After the flip, the residual is computed by simply subtracting the mapped lower bound from the flipped value.

Encode After the *map* phase, an encoder is used to compress the mapped data into a more compact form. We present four different methods here.

PFor uses the PFor algorithm to compress the mapped data. If the mapped integers are always positive, PFor is invoked directly [32,

60]. If the data is signed, zig-zag encoding (§2.2) is applied before the compressor so that the residuals with smaller absolute values are still stored with fewer bits.
PForDelta first computes the delta between mapped integers. It then applies zig-zag encoding on the data, and use the PFor algorithm to compress the results.
FPC uses the FPC floating number compression algorithm to compress the mapped data directly.
Bounded-Fpc (BFPC) is an improved version of the FPC algorithm. It aims to overcome two shortcomings of FPC while using it to compress bounded values. First, when generating the hash value, FPC extracts a fixed number of most significant bits from the value being compressed. Because a mapped integer likely contains zero bits in higher bits, this results in less information of the value being utilized. Second, FPC uses a modified variant-bytes format to compress the difference between the actual value and the predicted value. This format might store unnecessary zero bits, as well as requires 3 bits for encoding the length of the encoded number.

The BFPC encoder is illustrated in Algorithm 2. It uses the same hash-based predictors as the FPC compressor. However, BFPC adjusts the right shift bits parameter of the two predictors according to the common prefix bits of the upper and lower bound (line 2-4), so that the extract bits do not always include some zero bits. The generated residual is also compressed using the PFor algorithm for a better compression ratio (line 18). A *selector* bit vector tracks which predictor is used (line 15) and is prepended to the compressed residual.

All the mapping methods can be vectorized using SIMD instructions without conditional jumps, and the PFor-based compression algorithms are well-optimized for SIMD execution. We omit the details here.

Combining the different *map* and *encode* methods yields a range of possible residual compressors, but not all the combinations make sense. For example, the *none* mapping method can only work with an encoder which can efficiently compress patterned floating data. Therefore, we only consider the following combinations: *prefix removal / PFor* (prefix-pfor), *unsigned flip / PFor* (uf-pfor), *unsigned filp / PForDelta* (uf-pfordelta), *None/FPC* (fpc), *prefix removal / BFPC* (prefix-pfor) and *unsigned flip / BFPC* (uf-pfor). We evaluate the compression ratio of these compressors is evaluated in §5.2 as well.

4 QUERY PROCESSING

This section discusses how arrays in COMPASS can be queried.

4.1 System architecture

A dataset in COMPASS can be either a *plain* dataset (without an index), or an *indexed* dataset. Both kinds of datasets provide a chunk-based access method API for client programs (see §4.2). COMPASS also provides a set of composable pull-based operators for the users to access or process these datasets easily, implemented using the same chunk-based API.

Figure 3 gives an overview of the query processing using COMPASS. First, the `scan()` operator reads a COMPASS dataset, either plain or indexed, from the storage. COMPASS also has the ability to read data in external formats, such as NetCDF, directly as a plain dataset via ArrayBridge [56]. Value-based or dimension-based selection can then be performed on the scanned dataset, before being further processed by a parent operator (like aggregation) or a client program. Indexed and plain datasets can be converted to each other using the `to-plain()`/`to-indexed()` operator, in case the parent operator does not support the other type of dataset.

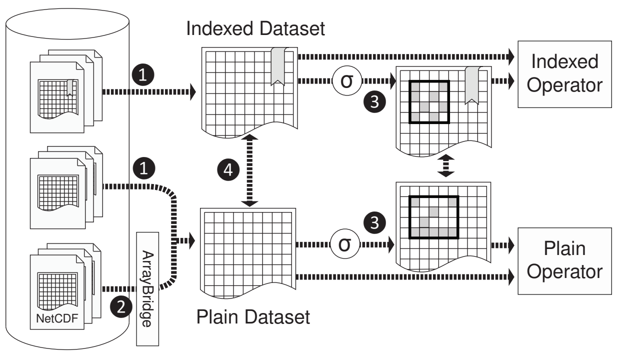


Figure 3: Overview of query processing in COMPASS. Indexed or plain datasets are read from storage via `scan()`(❶) or ArrayBridge (❷). Further selection based on dimension or value condition (❸) can be performed on the dataset before it is further processed. The indexed and plain datasets can be converted to each other (❹).

```
IndexedChunkIterator open(Chunk<T>* chunk);
bool next_bin();
bool next_tile();
size_t tile_size();
size_t* offsets();
T* values();
```

Table 1: The interface of an indexed chunk iterator.

4.2 Access method

This subsection describes the chunk-based access method provided by COMPASS. A client browses through the chunks of a dataset using a `DatasetIterator`, and reads the data inside a chunk using a `ChunkIterator`.

A `DatasetIterator` provides four major APIs for browsing the chunks in a dataset: `open()`, `next()`, `seek()` and `read_chunk()`, as well as additional methods to query the shape or type of chunk, or the bins it contains. Most of the methods have their usual semantics. However, as mentioned in the previous section, if only the values are needed, not retrieving the positions of the elements could save I/O and processing time. Therefore, the `read_chunk()` method accepts an additional parameter, to indicate whether the caller needs the positions.

After a chunk is retrieved, the user uses a chunk iterator to navigate through its content. How to process a plain chunk is well explored in prior works, so here we focus on the interface and implementation of an indexed chunk iterator.

An `IndexedChunkIterator`, as shown in Table 1, supports navigating through the bins of an indexed chunk via the `next_bin()` and `next_tile()` methods. Repeatedly calling `next_bin()` jumps to the next bin in the chunk. For each bin, the values and positions of the elements are returned in smaller fixed-sized blocks referred as *tiles*, in the same order as a plain dataset. Compared with decompressing the entire bin at once, this avoids accessing unnecessary tiles, as well as improves cache locality, hence improves the scanning performance. Calling `next_tile()` jumps to the next tile and decompresses it. The values of the elements in the tile can then be accessed via `values()` method. `offsets()` returns the position information if it is loaded. An `IndexedChunkIterator` only returns non-empty elements, the empty values are naturally ignored due to the storage representation.

4.3 Operator implementation

This subsection demonstrates how the access method described in the previous subsection can be used to implement common operations in array processing. We take value-based selection (filter), dimension-based selection (subset), and aggregation as examples.

The `filter()` operator selects the values within a certain range $[lb, rb]$ in the dataset. It first identifies the set of bins that intersects with the queried range, and iterates through all the chunks that contains these bins. For the bins inside the query range entirely, it decompresses and returns the entire bin. Otherwise, it checks the elements and returns the ones inside the query range. The filter operator only loads the positional index if the parent operator requires it to save I/O.

The `subset()` operator returns a hyper-rectangle dimensional area of the array. It first locates and iterates through all the chunks containing the interested area. If the chunk is all covered by the queried area, it returns the chunk directly. Otherwise, it looks through the offsets, and only returns the elements that are inside the query area. As converting the offsets to the coordinates and then check for the query condition is computationally expensive, the subset operator computes the ranges of the offsets being queried using the query condition, and checks if the decompressed ranges matches the computed offset range. A subset operator always loads the position information while reading chunks from its children.

Finally, the `aggregate()` operator returns the aggregates values of all the elements in the input array. The implementation is straightforward. It simply iterates through all the non-empty elements and perform the required aggregation.

4.4 Converting indexed and plain chunks

When performing position-dependent operations on an indexed dataset, such as matrix multiplication or convolution, it might be desirable to convert the dataset to a plain chunk. Similarly, plain chunks need to be converted to indexed chunks in data loading. COMPASS provides the `to-plain()` and `to-indexed()` operators for this need.

The `to-plain()` operator accepts an input dataset as its input and converts it to a plain dataset with the same segment shape. A naive implementation can allocate a plain chunk, iterate through all the overlapped indexed chunks one-by-one, and decompress the values to their correct positions. However, this harms the locality, and adds significant pressure on the memory allocation and paging system. Instead, our implementation constructs and returns the plain chunk in the unit of *tiles*, each containing a fixed number of contiguous elements in the chunk. Internally, `to-plain()` operates similar to a merge join: it reads the next tiles of all overlapped bins, and performs a merge operation as the next tile is requested by the parent operator. To handle empty elements, it uses a *bitvector* to track which elements do not exists in all bins, and returns the bitvector together with the data. Our benchmark shows the tile-based implementation of `to-plain()` improves the native implementation by $\sim 100\%$ in terms of CPU time.

The `to-index()` operator is the inverse operation of `to-plain()`. In addition to the input dataset, it also needs the number of bins, the ranges of each bin, as well as the number of bins in each chunk as parameters. For every plain chunk, the operator uses binary search to place its values to the correct bin, and combines the bins to assemble indexed chunks.

5 EXPERIMENTAL EVALUATION

In this section we experimentally evaluate the COMPASS array storage. We are interested in the following questions:

- How does an indexed dataset compare with a plain dataset in terms of their storage footprints? (§5.2)
- How does the performance of value-based selection (filtering) and dimension-based selection (subsetting) on indexed datasets compare with the performance on plain datasets and a state-of-the-art array database, SciDB? (§5.3 and §5.4)
- Comparing with querying the data approximately on a standalone bitmap index, what kind of trade-offs are we making in terms of accuracy and query time? (§5.5)
- Can the filtered result of an indexed dataset be converted to a plain representation effectively if necessary? (§5.6)
- What would be suitable chunking parameters for an indexed dataset? (§5.7)

5.1 Setup

Configuration. We evaluate our implementation on a platform with 2 14-cores E5-2680 v4 processors and 512 GB of memory, running CentOS 7. The data is stored in a local 2 TB 7200rpm WD hard disk with an `xfs` filesystem. The COMPASS storage engine is implemented in C++. COMPASS does not implement cache management itself and relies on the cache at the OS and storage level. We clear the cache before all experiments were execution times are measured.

We experimentally compare the performance of the same queries on an indexed array with those on the plain datasets. To the best of our knowledge, there is no array storage engine with value indices available. To better illustrate the performance of COMPASS, we also compare it against the popular SciDB array database, version 15.12, whose storage engine has been shown to have a similar I/O performance compared with popular scientific file formats such as HDF5 [56].

Datasets. Both synthetic and real-world datasets are used to evaluate the query performance of COMPASS. The synthesized dataset we used, *uniform*, is a two-dimensional 32-GiB dataset of double numbers with a uniform distribution on its value domain of [1, 101). The two real-world datasets are taken from the NASA Earth Exchange (NEX) Downscaled Climate Projections (NEX-DCP30) [48]. NEX-DCP30 contains the retrospective and perspective climate projection of the conterminous United States with multiple models from year 1950 to 2099. The first dataset, *tasmin*, contains the projected monthly average daily minimum surface temperature from 2066 to 2099 using an RCP model. The dataset can be seen as a 3105×7025×408 single-precision float array, with an uncompressed size of $\sim 34GiB$. The second dataset, *precipitation*, containing the projected precipitation from 1950 to 2005, is a $3105 \times 7025 \times 672$ single-precision float array with an uncompressed size of $\sim 55GiB$. We set the chunk size so that each segment contains approximately 64 MB of data. Unless otherwise specified, we use 100 *equi-width* bins for indexed datasets and standalone bitmap indices, and store 4 bins in an indexed chunk. Our choice of chunking parameters is based on our experiments reported at the end of this section (§5.7).

Table 2: Queries used in our experiments.

Name	Query
filter-sum	`SELECT SUM(value) FROM dataset WHERE` $value \in [lb, ub]$;
filter-subset-sum	`SELECT SUM(value) FROM dataset WHERE` $value \in [lb, ub]$ `AND` $dimension_i \in [ldim_i, udim_i], \forall i \in 1..rank$;
filter-subset-min	`SELECT MIN(value) FROM dataset WHERE` $value \in [lb, ub]$ `AND` $dimension_i \in [ldim_i, udim_i], \forall i \in 1..rank$;
filter-subset-scan	`SELECT value FROM dataset WHERE` $value \in [lb, ub]$ `AND` $dimension_i \in [ldim_i, udim_i], \forall i \in 1..rank$;

Queries. We choose aggregation with range selection conditions as the representative query, as it is one of the most common queries in analytics and its performance is representative for various queries performing an indexed scan. To test the performance of both value-based selection (filter) and dimension-based selection (subset), we use multiple selection conditions. Table 2 summarizes the queries used in our experiments. For value-based queries, the query performance can be related to both how large the queried domain is as a fraction of the value domain (*domain range*), and how many elements are being queried (*data amount*). We report both of the above as percentages of the original dataset.

5.2 Storage footprint

A small storage footprint reduces the I/O time and often results in less overall query response time if I/O is the bottleneck. Here we investigate the storage footprint of the 6 residual bits compression methods proposed in §3.5. We also evaluate whether using the EWAH [33] bitmap compression or the PForDelta [60] inverted list compression is more efficient for storing the positional index. [1]

We test these methods on the various datasets published by Burtscher et al. [10], as well as samples taken from the two real datasets we used, *tasmin* and *precipitation*. Table 3 reports the size of compressed *residual data* and *positional index* in percentages of

[1] For the FPC and BFPC methods, we pick a 32K hash table size, and set the left/right shift lengths for the *fcm* and *dfcm* coders to 4/16 and 2/12 bits respectively.

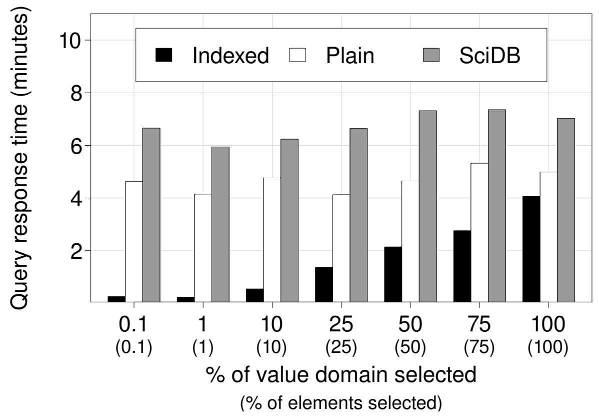


Figure 4: Filtering the *uniform* dataset (~32 GiB).

the original dataset. The methods with the best compression ratio is marked as bold.

In terms of residual compression, there is no clear winner, with uf-bfpc and uf-pfordelta being the two methods with the best compression ratios. The good performance of these methods indicates the importance for a residual compression method to effectively encode difference in the residuals. Overall, uf-bfpc seems to be the best choice; even when the uf-bpfc does not achieve the best compression ratio, the difference is usually within one or two percent of the original data size.

Because uf-bfpc captures more information and encodes the residual more efficiently, it also improves the compression ratio of none-fpc by an average of 1.5x, sometimes as much as 5.4x. Which mapping method is more effective depends on the encode method used: unsigned flip works better with the BFPC method, whereas *prefix removal* performs better with the PFor encoder on certain datasets.

As for the index compression, the inverted list performs significantly better than the bitmap in general, except in the datasets with highly concentrated values (msg-bt, msg-lu and msg-sppm). This is because bitmap indices is generally optimized for fast intersection and union operations rather than optimized storage size. Because matching a stored offset with its residual value while using bitmap as index also needs extra tracking, an inverted list is generally more suitable for storing the positional index.

Based on these observations, we compress the residual data using uf-bfpc, and the positional index using PForDelta in subsequent experiments. The last column in Table 3 shows the combined storage footprint of storing the indexed dataset. Even with the additional index, we still achieve better storage footprint compared with the original dataset, illustrating the effectiveness of the indexed storage scheme of COMPASS.

5.3 Filtering

Because scientific datasets are usually not frequently updated, the index creation time is an one-time cost and not our major concern. Hence, we turn our attention to the index scanning performance of COMPASS. This subsection compares the performance of the

Table 3: The storage footprint (% of the uncompressed dataset) of different residual and position compressing algorithms.

Dataset	Residual Compression						Position Compression		
	none-fpc	prefix-pfor	uf-pfor	uf-pfordelta	prefix-bfpc	uf-bfpc	EWAH	PFor-Delta	uf-bfpc + PForDelta
msg-bt	**80.1**	94.4	97.3	84.5	80.9	81.0	**0.0**	0.1	81.1
msg-lu	86.8	100.0	97.1	88.2	84.5	**84.2**	**0.0**	0.1	84.3
msg-sp	80.2	97.6	93.3	90.4	78.9	**78.9**	0.4	**0.3**	79.1
msg-sppm	20.0	76.5	85.1	31.4	15.7	**15.6**	**0.0**	0.1	15.7
msg-sweep3d	47.9	86.4	86.0	76.1	39.7	**39.4**	2.7	**1.2**	40.5
num-brain	86.4	82.3	79.9	**79.0**	81.3	80.2	16.2	**5.5**	85.7
num-comet	85.8	83.3	91.5	**78.8**	81.1	80.9	15.1	**4.8**	85.7
num-control	95.8	98.6	94.2	**90.0**	91.9	91.7	1.5	**0.9**	92.6
num-plasma	10.7	79.6	77.7	76.4	2.0	**2.0**	22.5	**6.1**	8.1
obs-error	50.5	80.2	77.4	55.8	38.4	**37.7**	23.7	**8.7**	46.4
obs-info	56.6	80.4	77.7	75.5	38.9	**37.9**	13.3	**5.0**	42.8
obs-spitzer	99.0	95.8	97.5	**94.4**	96.6	95.0	0.9	**0.5**	95.5
obs-temp	93.0	89.2	**83.1**	84.5	86.1	84.8	15.6	**8.0**	92.8
precipitation	N/A*	44.8	42.8	**40.5**	42.8	42.1	3.6	**2.6**	44.7
tasmin	N/A*	28.3	**26.1**	27.0	28.6	27.8	9.4	**4.7**	32.6

* The FPC method does not support single-precision data.

filter-sum query on indexed datasets against performing the same operation on plain datasets, and evaluating the query using SciDB.

Figure 4 shows the performance of the *filter-sum* query on the *uniform* dataset when the selectivity is varied form 0.1% to 100%. As expected, the query response time of filtering an indexed dataset is proportional to the selectivity, except for when the selectivity is lower than 4%. This is because the storage engine loads the entire chunk even if only one bin in it is accessed. This indicates that the disk I/O of loading the data, rather than the decompressing cost, is the actual bottleneck.

The indexed scan also shows a significant advantage over full scan on a plain dataset, whose response time does not change with selectivity. When selecting 10% of elements, the filter operation on the indexed dataset is ~ 8.7 times faster than on the plain dataset.

Each bin in the *uniform* datasets contains a similar number of the elements. To illustrate the effect of data distribution, we perform the same query on the *tasmin* real dataset, which has a distribution similar to the normal distribution, with values concentrating in the mid-range. Figure 5 and 6 shows the query response time while the query range chosen are relatively dense and sparse, respectively. As shown in the figures, the response time is almost proportional to the data amount being read. Comparison-wise, filtering the indexed dataset outperforms filtering the plain dataset by at least 3x, the advantage of indexed increases even further when fewer elements are selected, with the acceleration ratio reaching as high as 80x. When filtering the *precipitation* dataset (Figure 7), whose most data concentrates in just a few bins, the indexed scan shows similar significant speed up.

Even when all the elements are selected, filtering the indexed array is still faster on all synthetic and real datasets. This is because our storage scheme compresses the datasets efficiently while

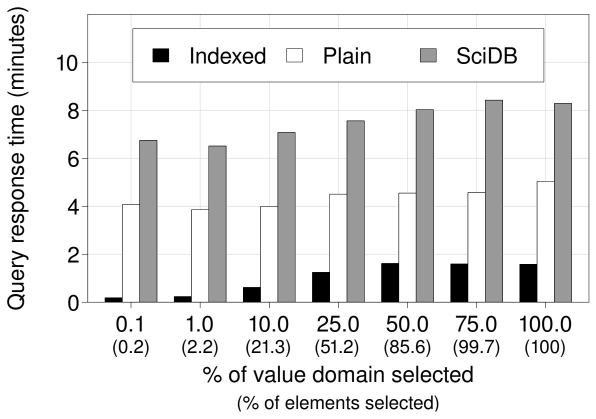


Figure 5: Filtering the *tasmin* dataset (~34 GiB). The query selects a relatively dense domain range.

adding limited decompression overhead: the indexed *tasmin* dataset is almost 70% smaller compared with the plain dataset, but it only takes about 25% extra CPU time to decompress and aggregate the indexed dataset. Our profile shows about half of the CPU time goes to decompressing the residual bins, especially on updating the prediction table of BFPC and calculate the actual values using the table because it could not be vectorized efficiently, indicating further improving decoding efficiency is the key to obtain better in-memory performance.

Finally, both the indexed dataset and the plain datasets of COMPASS outperform SciDB in all scenarios. This is because the `filter()` operator of SciDB is implemented in a general way, without special

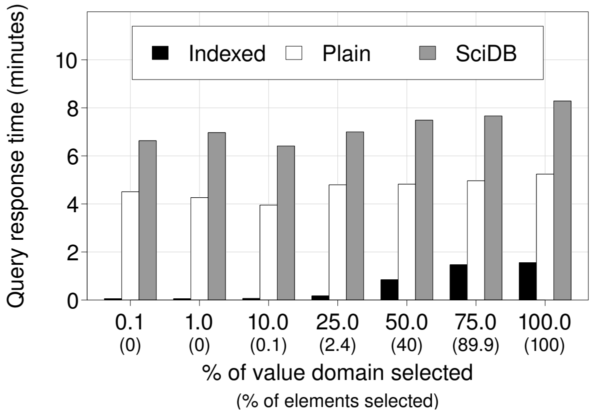


Figure 6: Filtering the *tasmin* dataset (~34 GiB). The query selects a relatively sparse domain range.

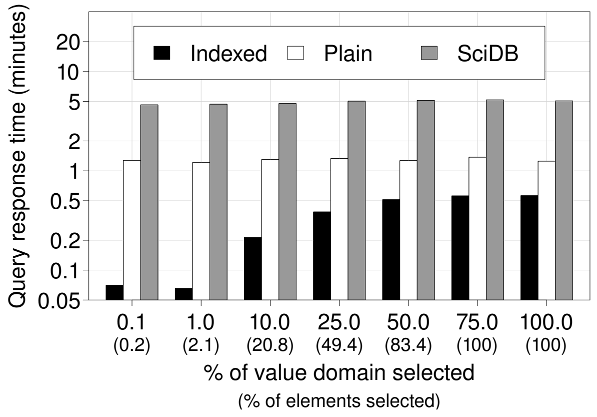


Figure 8: Filtering a subset of the *tasmin* dataset (~34 GiB) and aggregate the result.

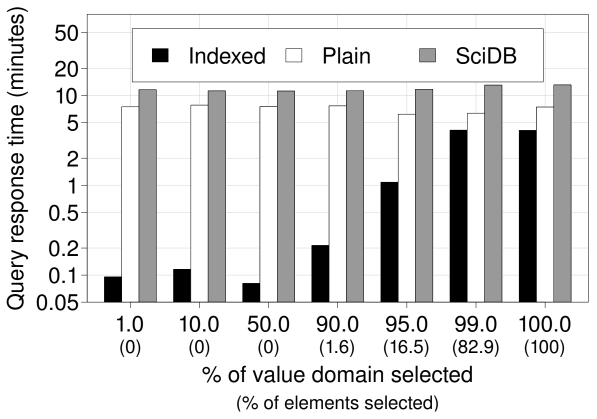


Figure 7: Filtering the *precipitation* dataset (~55 GiB).

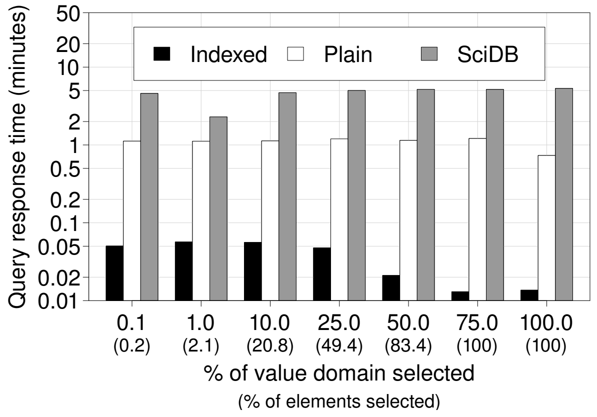


Figure 9: Filtering a subset of the *tasmin* dataset (~34 GiB) and find its minimum value.

optimization for the range filter query used here. Hence, it brings a significant CPU overhead.

5.4 Subsetting

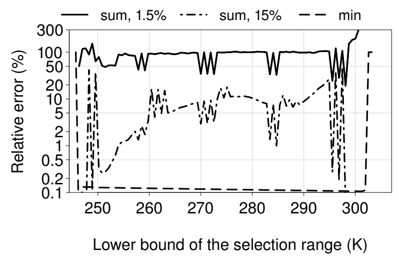
The next question is whether the subset operation can still be performed on an indexed dataset as efficiently as on a plain dataset. We evaluate this by performing the *filter-subset-sum* query on the dataset. We select a region occupying 1/8 volume of the tasmin dataset, and vary the selectivity from 0.1% to 100%.

As the result in Figure 8 shows, with the additional subsetting, querying indexed datasets still retains its performance advantage over both plain datasets and SciDB arrays. This is because an indexed dataset still uses *regular chunking* to accelerate dimension-based selection, and the sub-chunk selection in the boundary chunks can be performed relatively fast as well.

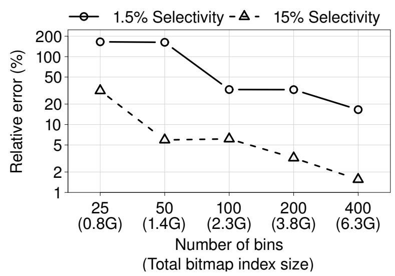
Different aggregation operators can have different performance pattern. For evaluation, we tested the performance of the *filter-subset-min* query with the same parameters. The result is shown in Figure 9. Performing the query on the indexed dataset is ~ 20 times or more faster than performing it on the plain dataset, as the minimum operator only needs to read the bins and chunks that might contain smaller values than the current minimum value. The performance of the indexed method improves when the query range becoming larger. This is because the *tasmin* data distributes similar to the normal distribution, so when the query range increases, the minimal bin and chunk contain less data.

5.5 Comparison with approximate aggregation method

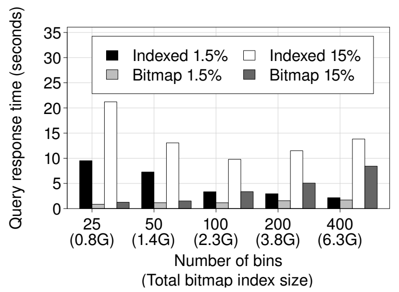
Approximate methods improve performance by sacrificing certain level of accuracy. A promising approximate method for aggregations involves using bitmaps as a standalone representation [50]. The bitmap compression algorithm used is EWAH [33], a widely-used variant of the famous WAH [55] algorithm with a focus on decompression speed. We investigate the accuracy and response time of the query *filter-subset-sum* and *filter-subset-min* under different settings in Figure 10. We use the *tasmin* dataset, and still set the subset area to 12.5%.



(a) Accuracy of the bitmap method when selectivity and the query range vary.



(b) Accuracy of the bitmap method when selectivity and the number of bins vary.



(c) Response time of the *filter-subset-sum* query, when the number of bins varies.

Figure 10: Comparing indexed dataset with bitmap approximate aggregation, using the *filter-subset-sum* and *filter-subset-min* queries.

Figure 10a shows the relative error of performing *filter-subset-sum* and *filter-subset-min* approximately using bitmaps. The width of the query range is set to 1.5% and 15% of the domain range, while its lower bound is moved through the entire domain range. Although the min aggregation can be performed quite accurately on an equal-width binned bitmap, executing the sum query approximately on a 1.5% domain range almost always results in a relative error of more than 50%. The approximate method still results in an average relative error of 8.6% even if the selectivity increases to 15%. Although certain binning methods might improve the accuracy, these methods usually result in a significant increase of the index construction time, and might require *a priori* knowledge regarding the distribution of the query range. On the contrary, filtering an indexed dataset returns the accurate results regardless of the binning strategy, showing the robustness of our index method.

Another way to increase the accuracy of bitmap-based methods is increasing the number of bins. As shown in Figure 10b, the accuracy at both selection levels significantly improves. However, the size of the bitmap also increases. At 400 bins, the compressed bitmap occupies ~ 60% space of the indexed dataset. This decreases the speedup of the approximate method, shown in Figure 10c.

On the contrary, adding more bins to an indexed dataset does not change the storage footprint by much since more fine-grained binning also improves the residual compression ratio. The query response time of the indexed dataset while the number of bins increases are determined by two factors: adding bins reduces the amount of unnecessary data processed, but also increases the overhead of processing the extra bins. In this case, at the low selectivity level, fine-grained binning greatly reduces the unnecessary data processed and improves performance. However, at the 15% selectivity level, the saved I/O was shadowed by the overhead of added bins while using more than 100 bins, increasing the query response time.

Comparison-wise, the performance of using an 100-bins indexed dataset is comparable with using a 400-bins bitmap index in this case, and always returns accurate results, making it an attractive alternative for the approximate query method.

5.6 Converting indexed chunk to plain chunk

Sometimes it can be necessary to convert an indexed dataset to its plain form via the `to_plain()` operator, so that some operations

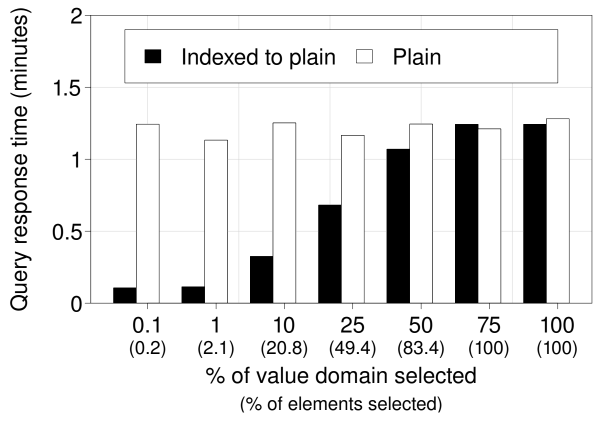


Figure 11: Comparison of the response time for the *filter-subset-scan*query on a 12.5% subset area of the *tasmin* dataset on a plain dataset, and on an indexed dataset, but use `to_plain()` operator to convert the result to a plain form.

can be performed more efficiently. This subsection evaluates the performance of the `to_plain()` operator. Figure 11 compares the response time of performing the *filter-subset-scan* operation on a plain dataset, and on an indexed dataset, but using the `to_plain()` operator to convert it to a position-based representation.

Converting the indexed dataset to a plain dataset takes extra computation time because of the overhead in reorganizing array cells, so it is preferable to access an indexed dataset using the bin-based access method we mentioned in §4.2. However, the additional cost is limited because our tile-based algorithm avoid expensive memory operation that harms cache locality. This overhead is offset by the reduction in disk I/O amount. Overall, the indexed scan accelerates the query at 10% selectivity by 3.9x. Its performance scales well with the number of elements selected, and is on par with performing the query on a plain dataset directly.

5.7 Chunking parameters

Finally, we investigate the choices of chunking parameters with a given binning strategy. We pick *filter-subset-sum* as a typical query with both value and dimension selections, selecting 1% or 50% of

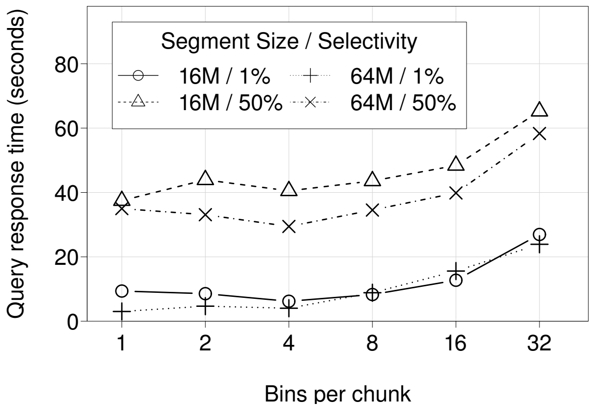


Figure 12: Response time of a filter operation on a 25% subset area of the *uniform* dataset, with different chunk size and number of bins per chunk.

the domain range, and 12.5% of the dimensional space. We use 100 *equi-width* bins and a segment size of either ∼ 16 MiB or ∼ 64 MiB, and varys the number of bins per chunk from 1 to 32.

Figure 12 shows the query response time. As we can see, the best chunking strategy depends on the segment size *and* selectivity. In general, storing too many bins in a chunk increases the unnecessary I/O and harms the performance. However, the response time does not benefit much from too fine-grained chunk sizes either, because the overhead of meta-data processing and seek time overhead shadows the reduced data I/O. In our setting, it seems that this query performs reasonably well when the chunk size is set in an order of megabytes. Based on these observations, we set the segment size to be 64 MiBs, and using 4 bins per chunk in our other experiments.

6 RELATED WORK

Array storage. HDF5 [22] and NetCDF [37] are two dominant array file formats. Both formats focus on portability and usability, and do not provide any support for value-based subsetting. There has been a series of effort to design more feature-rich array storages [39]. Radasman [51] implements array storage as an extension of relational databases. ArrayStore [44] propose a two-layer chunking strategy to balance the CPU and I/O requirements of an array storage. SciDB [9] implements a share-nothing storage engine with features such as replication, and versioning support. TileDB [36] uses a fixed-size tiling strategy and decomposes a dataset to multiple overlapped fragments, allowing efficient writing and updates on sparse arrays. Our work utilizes the *regular chunking* strategy widely used by these systems and considers how value-based index can be implemented in such systems with limited extra cost.

Many array storages compress the chunks before they are written to the disk to reduce data size. A lot of these systems use general-purpose lossless compressors [18, 41, 58]. One challenge is writing compressed scientific files in parallel [4–6]. General floating-point number compressors are hard to design. FPC [10] uses a combination of two predictors to generate residual and uses variant bytes to compress the residual. The ISOBAR preconditioner [40] identifies the compressible bytes of a floating-point number stream and compress them only to improve the overall compression ratio.

ALACRITY and DIRAQ [26, 30] store in-situ simulation data with precision-based index and *fused encoding*. In these systems, the data is reorganized into bins based on its k higher-order bytes, and an inverted index tracks the positions of the values in each bin. The lower bytes of the values in each bin are then compressed using ISOBAR and stored separately. However, the number of bins and their ranges in these systems are determined by the amount of unique high-order bytes in the input, limiting the acceleration due to the index, and reducing the flexibly of its index. These systems also do not consider how to incorporate such fused encoding into a multi-dimensional array storage, while preserving the ability of performing efficient subsetting operations. Our work, COMPASS shows how the idea of fused data and index encoding can be used in an array processing system with arbitrary binning support, presents an efficient bit-based residual compression scheme, and the design of a flexible storage engine capable of processing various queries on both plain and indexed dataset.

Bitmap index and approximate query processing. Efficient compression of a bitmap index reduces the storage footprint as well as query time. Popular methods such as BBC [2], WAH [55] and its variants [17, 20, 24, 29, 33] utilizes Run-Length Encoding (RLE) to encode continuous *0-fill* or *1-fill* efficiently. Roaring bitmap [13, 31] partitions a long bitmap into smaller chunks and stores each chunk either as a bitvector or as a sorted integer array of element indices, based on the density of set bits in the chunk. UCS [12] and Upbit [3] uses muiltple bitmap vectors to allow updates on bitmap indices.

Bitmap indices and its variants have been shown to be efficient in OLAP-style relational workloads, especially when the cardinality of the column is low [14, 15, 34, 53, 54]. FastBit [16, 52], HDF5-FastQuery [23], and SDS/Q [8] support accelerating scientific queries on existing file formats using external bitmap indices. Bitmaps can also be used to answer approximate queries on scientific dataset without referring to the original data[42, 46, 50, 57]. The accuracy of such queries often relies on our *a priori* knowledge about the data and query distribution, and the binning strategy used. By merging the data and indices, COMPASS reduces the storage redunancy of an external bitmap indices, improves I/O efficiency, and provides accurate results regardless of the queries and the binning strategy.

Inverted lists and data compression. Inverted lists are a widely used data structure in information retrieval systems, and are usually stored in a compression form. The compression methods can be roughly divided to three categories: bit-level compression such as Rice coding[38], Gamma coding [11] and elias-fano coding [35]; byte-oriented compression such as variable bytes and its variants [19, 45]; and word-oriented encoders such as the Simple [1] family and various PFor-based compressors [32, 60].

Although inverted lists and bitmap index serve a similar purpose, the latter is widely used in the information retrieval community, while the former are widely used by the database discipline. Until recently, not a lot of the discussion has been made on the similarity of these two data structures. Bjørklund *et al.* discuss the suitability of these two structures in decision support systems [7].

Zou *et al.* [59] propose to use bitmap indices to accelerate intersection operation of two posting lists. Recently, Wang *et al.* provide a very detailed comparison between the two structures [49].

7 CONCLUSION AND FUTURE WORK

We have presented COMPASS, an array storage system with integrated value index support. COMPASS reorganizes the elements into a number of user-defined bins, and efficiently encodes the bin-based indices and their corresponding values, generating an indexed array representation that adds little storage overhead.

This paper also presented a chunk-based access method suitable for processing indexed array data, and the corresponding query processing strategies. An indexed array storage performs accurate filtering operations irrespective of the binning strategy chosen by a user, and preserves the efficiency of the subsetting operations in traditional array storages. The filtering operation on an indexed dataset scales with the number of elements selected, and outperforms the plain representation consistently, even when the selectivity level is relatively high. The filtered result of an indexed dataset can also be converted back to its plain representation efficiently if necessary.

Interesting future research questions include how to implement more complex operators on top of the indexed access method, how an indexed dataset can be dynamically updated, and how the binning strategies and parameters can be automatically selected.

REFERENCES

[1] Vo Ngoc Anh and Alistair Moffat. 2010. Index compression using 64-bit words. *Software - Practice and Experience* (2010).
[2] Gennady Antoshenkov. 1995. Byte-aligned bitmap compression. In *DCC'95*.
[3] Manos Athanassoulis et al. 2016. Upbit: Scalable in-memory updatable bitmap indexing. In *SIGMOD '16*.
[4] Tekin Bicer and Gagan Agrawal. 2013. A Compression Framework for Multidimensional Scientific Datasets. In *IPDPS Workshop*.
[5] Tekin Bicer, Jian Yin, and Gagan Agrawal. 2014. Improving I/O Throughput of Scientific Applications using Transparent Parallel Compression. In *CCGrid'14*.
[6] Tekin Bicer, Jian Yin, David Chiu, et al. 2013. Integrating online compression to accelerate large-scale data analytics applications. In *IPDPS '13*.
[7] Truls A Bjørklund et al. 2009. Inverted indexes vs. bitmap indexes in decision support systems. In *CIKM '09*.
[8] Spyros Blanas, Kesheng Wu, Surendra Byna, et al. 2014. Parallel data analysis directly on scientific file formats. In *SIGMOD '14*.
[9] Paul G Brown. 2010. Overview of SciDB: large scale array storage, processing and analysis. In *SIGMOD*.
[10] Martin Burtscher and Paruj Ratanaworabhan. 2009. FPC: A high-speed compressor for double-precision floating-point data. *IEEE Trans. Comput.* 58, 1 (2009).
[11] Stefan Büttcher, Charles L A Clarke, and Gordon V Cormack. 2016. *Information retrieval: Implementing and evaluating search engines.* Mit Press.
[12] Guadalupe Canahuate et al. 2007. Update conscious bitmap indices. In *SSDBM*.
[13] Samy Chambi, Daniel Lemire, et al. 2016. Better bitmap performance with Roaring bitmaps. *Software: practice and experience* 46, 5 (2016).
[14] Chee-Yong Chan and Yannis E Ioannidis. 1998. Bitmap index design and evaluation. In *SIGMOD Record'98*. ACM.
[15] Chee-Yong Chan and Yannis E Ioannidis. 1999. An efficient bitmap encoding scheme for selection queries. In *ACM SIGMOD Record*, Vol. 28. ACM, 215–226.
[16] Jerry Chou, Mark Howison, et al. 2011. Parallel index and query for large scale data analysis. In *SC '11*.
[17] Alessandro Colantonio and Roberto Di Pietro. 2010. Concise: Compressed 'n' Composable Integer Set. *Inform. Process. Lett.* 110, 16 (2010).
[18] Yann Collet. 2016. Zstandard - Real-time data compression algorithm. (2016).
[19] Jeffrey Dean. 2009. Challenges in building large-scale information retrieval systems. In *WSDM '09*.
[20] François Deliège and Torben Bach Pedersen. 2010. Position list word aligned hybrid. In *EDBT '10*.
[21] M Duhrssen et al. 2004. Extracting Higgs boson couplings from CERN LHC data. *Physical Review D* 70, 11 (2004).
[22] Mike Folk, Gerd Heber, and Quincey Koziol. 2011. An overview of the HDF5 technology suite and its applications. *EDBT '11*.
[23] Luke Gosink et al. 2006. HDF5-FastQuery: Accelerating complex queries on HDF datasets using fast bitmap indices. In *SSDBM '06*.
[24] Gheorghi Guzun, Guadalupe Canahuate, David Chiu, and Jason Sawin. 2014. A tunable compression framework for bitmap indices. In *ICDE '14*.
[25] Zeljko Ivezic et al. 2008. LSST: from science drivers to reference design and anticipated data products. *arXiv preprint arXiv:0805.2366* (2008).
[26] John Jenkins et al. 2013. ALACRITY: Analytics-driven lossless data compression for rapid in-situ indexing, storing, and querying. In *Transactions on Large-Scale Data-and Knowledge-Centered Systems X.*
[27] William Kahan. 1996. IEEE standard 754 for binary floating-point arithmetic. *Lecture Notes on the Status of IEEE* 754, 94720-1776 (1996).
[28] Johannes Kepler. 1865. Epitome astronomiae copernicanae [1628], in. *Kepler, J., Opera omnia* (1865).
[29] Sangchul Kim, Junhee Lee, Srinivasa Rao Satti, and Bongki Moon. 2016. SBH: Super byte-aligned hybrid bitmap compression. *Information Systems* 62 (2016).
[30] Sriram Lakshminarasimhan et al. 2014. DIRAQ: scalable in situ data-and resource-aware indexing for optimized query performance. *Cluster computing* 17, 4 (2014).
[31] Daniel Lemire et al. 2017. Roaring Bitmaps: Implementation of an Optimized Software Library. *arXiv preprint arXiv:1709.07821* (2017).
[32] Daniel Lemire and Leonid Boytsov. 2015. Decoding billions of integers per second through vectorization. *Software: Practice and Experience* 45, 1 (2015).
[33] Daniel Lemire, Owen Kaser, and Kamel Aouiche. 2010. Sorting improves word-aligned bitmap indexes. *Data & Knowledge Engineering* 69, 1 (2010).
[34] Patrick O'Neil and Dallan Quass. 1997. Improved query performance with variant indexes. *SIGMOD Record* 26, 2 (1997).
[35] Giuseppe Ottaviano et al. 2014. Partitioned elias-fano indexes. In *SIGIR '14*.
[36] Stavros Papadopoulos, Kushal Datta, et al. 2016. The TileDB array data storage manager. *VLDB '16*.
[37] Russ Rew and Glenn Davis. 1990. Data Management: NetCDF: an Interface for Scientific Data Access. *IEEE Computer Graphics and Applications* 10, 4 (1990).
[38] Robert Rice et al. 1971. Adaptive Variable-Length Coding for Efficient Compression of Spacecraft Television Data. *IEEE Trans. Commun. Technol.* 19, 6 (1971).
[39] Florin Rusu and Yu Cheng. 2013. A Survey on Array Storage, Query Languages, and Systems. (2013). arXiv:1302.0103v2
[40] Eric R. Schendel et al. 2012. ISOBAR preconditioner for effective and high-throughput lossless data compression. In *ICDE '12*.
[41] Julian Seward. 1998. The bzip2 compressor. (1998).
[42] Sameh Shohdy, Yu Su, and Gagan Agrawal. 2015. Load balancing and accelerating parallel spatial join operations using bitmap indexing. In *HiPC '15*.
[43] Nayanah Siva. 2008. 1000 Genomes project. (2008).
[44] Emad Soroush, Magdalena Balazinska, and Daniel Wang. 2011. ArrayStore: a storage manager for complex parallel array processing. In *SIGMOD '11*.
[45] Alexander A. Stepanov, Anil R. Gangolli, et al. 2011. SIMD-based decoding of posting lists. In *CIKM '11*.
[46] Yu Su, Gagan Agrawal, et al. 2014. Effective and efficient data sampling using bitmap indices. *Cluster computing* 17, 4 (2014).
[47] Yu Su, Gagan Agrawal, and Jonathan Woodring. 2012. Indexing and parallel query processing support for visualizing climate datasets. In *ICPP '12*.
[48] Bridget Thrasher et al. 2013. Downscaled Climate Projections Suitable for Resource Management. *Eos* 94, 37 (2013).
[49] Jianguo Wang, Chunbin Lin, et al. 2017. An Experimental Study of Bitmap Compression vs. Inverted List Compression. In *SIGMOD '17*.
[50] Yi Wang, Yu Su, and Gagan Agrawal. 2015. A novel approach for approximate aggregations over arrays. In *SSDBM '15*.
[51] Norbert Widmann and Peter Baumann. 1998. Efficient Execution of Operations in a DBMS for Multidimensional Arrays. In *SSDBM '98*. 155–165.
[52] Kesheng Wu et al. 2009. FastBit: interactively searching massive data. *Journal of Physics: Conference Series* 180, 1 (2009).
[53] Kesheng Wu, Ekow Otoo, and Arie Shoshani. 2004. On the performance of bitmap indices for high cardinality attributes. In *VLDB '04*.
[54] Kesheng Wu, Ekow J Otoo, and Arie Shoshani. 2006. Optimizing bitmap indices with efficient compression. *TODS* 31, 1 (2006).
[55] Kesheng Wu, Ekow J Otoo, Arie Shoshani, and Henrik Nordberg. 2001. *Notes on design and implementation of compressed bit vectors.* Technical Report. LBNL/PUB-3161, Lawrence Berkeley National Laboratory, Berkeley, CA.
[56] Haoyuan Xing et al. 2018. ArrayBridge: Interweaving declarative array processing in SciDB with imperative HDF5-based programs. *ICDE '18*.
[57] Gangyi Zhu, Yi Wang, and Gagan Agrawal. 2015. SciCSM: novel contrast set mining over scientific datasets using bitmap indices. *SSDBM '15* (2015).
[58] Jacob Ziv and Abraham Lempel. 1977. A Universal Algorithm for Sequential Data Compression. *IEEE Transactions on Information Theory* 23, 3 (1977).
[59] Xiaocheng Zou et al. 2014. Fast Set Intersection through Run-Time Bitmap Construction over PForDelta-Compressed Indexes. In *Euro-Par 2014*.
[60] Marcin Zukowski, Sandor Heman, Niels Nes, and Peter Boncz. 2006. Super-scalar RAM-CPU cache compression. In *ICDE '06*.