Research Project Report

Performance Evaluation of Data Formats and Access Libraries for Raster Data Processing

Sagnik Sarkar
Research in Computer and Systems Engineering
Technische Universität Ilmenau
Email: sagnik.sarkar@tu-ilmenau.de
Matriculation Number: 64446
Supervisor: Dr. Marcus Paradies

March 1, 2024

1 General Terms, Keyword

Zarr, Multidimensional arrays, Performance benchmarking, compression algorithms, chunking strategies, advanced indexing

2 Introduction

This Research Project report encapsulates a detailed overview of the performance evaluation of the Zarr library that is used to store multi dimensional arrays or tensors [2]. The advantage of using Zarr is that it can store the datasets into several chunks which can be user defined and those compressed chunks can be accessed while reading the data. Consequently there are two main advantages to it. Firstly, storing compressed chunks requires less space and also accessing data from dataset does not require the whole dataset to be loaded, instead it requires the particular chunk and only that chunk can be loaded resulting in faster access speeds. In this report I have configured different parameters(eg. Compressors, blocksize, shuffle) while creating the data sets and observed the impact on accessing the data. Also I evaluated the native Zarr queries with different configurations and evaluated their performance. I have used Read Time, Average CPU usage and Disk IO count to evaluate the performance.

3 Background

3.1 Zarr

Zarr is a storage file format that is specified by .zarr extension. It is used to store large dataset using multidimensional arrays. It is useful to store data from Life Science/climate forecasts domain which contains data upto 7 dimensions. Zarr provides a very intelligent and efficient way to store data in memory by the method of chunks. This chunks can be defined by the user at the time of creation or Zarr can by itself define the chunks based on the dataset structure. This chunks are stored individually in memory in a compressed way. And when a particular data is accessed, Zarr loads up the chunk where the data may reside instead of loading up the whole dataset into the memory which improves the access speed as well.

3.2 Compressor

There are multiple compressors available in for the implementation in Zarr. It uses a library called *numcodecs* which contains various compressor classes such as Blosc, LZ4, Zstd, Zlib, LZMA, GZip and various others. For this project, blosc framework [5] is selected for the benchmarking since it is the default compressor provided in the Zarr library. Now different algorithms are used inside blosc for the benchmarking evaluation such as lz4, lz4hc, zlib etc. Now all these algorithms requires certain parameters for creating a blosc compressor object. The parameters include:

3.2.1 cname

This parameter takes string as an argument which specifies the compression algorithm. For example, the compression algorithm includes 'lz4', 'zlib' etc.

3.2.2 clevel

This parameter specifies the compression level. This value ranges from 0 to 9. The more the compression level, better the compressed size but less reliable data.

3.2.3 shuffle

This parameter takes integer as an argument. *No Shuffle* is denoted by 0, *Shuffle* is denoted by 1, *Bit Shuffle* is denoted by 2 and *AutoShuffle* is denoted by -1. Shuffle is used to juggle the data to increase the compression efficiency. But again the reliability is compromised where the original data might be altered after the decompression.

3.2.4 Blocksize

This specifies the requested byte size of the compressed blocks. If 0 is selected then the block size of the compressed blocks are automatically determined.

3.3 Chunking Strategies

Chunking can significantly reduce the access time and increase the efficiency for storing the data in memory [4]. But to attain an optimal chunking strategy is dependent of data and there is no generic optimal strategy for Zarr. Although there has been a literature available which discussed about some optimal chunking strategy using geometrical programming and steep descent optimization method [3]marcus. There is a way to configure the chunks while creating a Zarr array. Now it depends on the data access patterns to set the chunking strategies. The chunking can be done on various dimensions. As per the Zarr documentation, 1 Megabyte uncompressed size seems to provide better performance. So if there is a data of 2 dimensions and there is only need to access the first dimension then only the second dimension can be chunked.

```
z1 = zarr.zeros((10000, 10000), chunks=(100, None), dtype='i4')
z1.chunks #(100, 1000)
```

Similarly, if there is a need to perform the chunking on the second dimension then that can also be done.

```
z1 = zarr.zeros((10000, 10000), chunks=(None, 100), dtype='i4')
z1.chunks #(1000, 100)
```

In cases where both all the dimensions need to be chunked,

```
z1 = zarr.zeros((10000, 10000), chunks=(100, 100), dtype='i4')
z1.chunks #(100, 100)
```

In certain cases where the chunking strategy cannot be determined by the user, it can use the internal chunking algorithm of Zarr to perform the chunking. Although it is worth mentioning that this chunking may not be optimal since the decision of the chunking strategy of Zarr is based on simple heuristics.

```
z1 = zarr.zeros((10000, 10000), chunks=True, dtype='i4')
z1.chunks #(625, 625)
```

Finally, if there is a case where chunking is not required and every time the entire dataset needs to be loaded into memory for use, then the chunking can be disabled as well.

```
z1 = zarr.zeros((10000, 10000), chunks=False, dtype='i4')
z1.chunks #(1000, 1000)
```

It is also possible to change the chunk shape afterwards also termed as re chunking where there is a need to change the chunk shape after creation of the Zarr array.

3.4 Advanced Indexing

There are certain advanced indexing implementations available in the Zarr library such as Orthogonal Indexing, Vindexing, Coordinate Selection and Block Indexing.

3.4.1 Coordinate Selection

This indexing technique access the element in the multi dimensional array by specifying the coordinates. For instance if there is a dataset of 64 elements with shape 4*4*4, the coordinate selection will work like this,

```
import zarr
import numpy as np
a = np.arange(64).reshape(4,4,4)
test_zarr = zarr.array(a, chunks=(2, 2, 2))
test_zarr[:]
```

Now output of this array creation object will look like this:

array([[[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11], [12, 13, 14, 15]],

[[16, 17, 18, 19], [20, 21, 22, 23], [24, 25, 26, 27], [28, 29, 30, 31]],

[[32, 33, 34, 35], [36, 37, 38, 39], [40, 41, 42, 43], [44, 45, 46, 47]],

[[48, 49, 50, 51], [52, 53, 54, 55], [56, 57, 58, 59], [60, 61, 62, 63]]])

Now to select a particular element using the coordinate selection, the command is,

```
test_zarr.get_coordinate_selection(([0, 2], [1, 3], [3, 3]))
#array([ 7, 47])
```

Now from the syntax it is imperative that row that is accessed resides in the 0th row in the 1st dimension, 1st row in the 2nd dimension and 3rd row in the 3rd dimension and the column resides in 2nd column in the 1st dimension, 3rd column in the 2nd dimension and 3rd column in the 3rd dimension which results in the value (7, 47). For using the coordinate selection function it is essential to provide the coordinates of all the dimensions for multidimensional arrays. Coordinate selection can also used to select set the values using the *set_coordinate_selection* function.

3.4.2 vindex

This indexing technique is also similar to Coordinate selection in terms of functionality. It uses square bracket operator unlike the Coordinate selection function.

```
z.vindex[[0, 2], [1, 3]]
```

This indexing is also used for this benchmarking.

3.4.3 Orthognal Selection

This indexing allows for the library to select values from a particular dimension independently. A particular dimension can be chose to access the elements omitting the other dimensions.

```
z = zarr.array(np.arange(15).reshape(3, 5))
z[:]
#array([[ 0,  1,  2,  3,  4],
#       [ 5,  6,  7,  8,  9],
#       [10, 11, 12, 13, 14]])
z.get_orthogonal_selection(([0, 2], slice(None)))  # select first and third rows
#array([[ 0,  1,  2,  3,  4],
 #          [10, 11, 12, 13, 14]])
z.get_orthogonal_selection((slice(None), [1, 3]))  # select second and fourth
                                                   columns
#array([[ 1,  3],
#          [ 6,  8],
#          [11, 13]])
z.get_orthogonal_selection(([0, 2], [1, 3]))

# select rows [0, 2] and columns [1, 4]
#array([[ 1,  3],
#          [11, 13]])
```

For the snippet *z.get_orthogonal_selection(([0, 2], slice(None)))*, only the first dimension is selected. Similarly either the other dimension or all the dimensions can be selected for the multidimensional array.

3.4.4 Block Indexing

The Block indexing retrieves the block of data based on the logical indices along each dimension of an array.

```
import zarr
import numpy as np
z = zarr.array(np.arange(100).reshape(10, 10), chunks=(3, 3))
#Selecting the block with logical index 1
z.get_block_selection(1)
#array([[30, 31, 32, 33, 34, 35, 36, 37, 38, 39],
 #  [40, 41, 42, 43, 44, 45, 46, 47, 48, 49],
  # [50, 51, 52, 53, 54, 55, 56, 57, 58, 59]])
#Selecting the block with logical index 0
z.get_block_selection(0)
#array([[ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9],
 #  [10, 11, 12, 13, 14, 15, 16, 17, 18, 19],
  # [20, 21, 22, 23, 24, 25, 26, 27, 28, 29]])
```

The block selection is accessible until and unless there exists a logical index for the multi dimensional array. For this case the logical index ends at 3. Now, if logical index 4 is tried to be accessed, then it throws a *BoundCheckError*.

3.5 Benchmarks

3.5.1 Data

Data that is used for the compressor benchmarking contains 1000000000 elements with shape 1000*1000*1000 and contains *Integer* data type elements. For the experiment purpose the fill value is made with elements from 0 to 99999999 for the entire dataset using the *arange* function of *numpy* library in Python. The snippet of the Zarr object creation is

```
compressor = Blosc(cname=value[0].value,clevel=int(value[1].value), shuffle=int(value
                                                    [2].value))
zarr_blosc = zarr.create((1000, 1000, 1000), chunks=True, compressor=compressor,
                                                    dtype='i4')
zarr_blosc = zarr.array(np.arange(1000000000).reshape(1000, 1000, 1000))
```

Now, to create the compressor object, *cname, clevel and blocksize* parameters are extracted from a configuration file where different parameter configurations are listed and this code snippet will receive the values one after another. The next line in the code is creating the Zarr object based on different parameters of the compression algorithm created above. *create* function is used of the Zarr library where an array of 1000*1000*1000 elements array is created and the chunks parameter is set to True which means that the Zarr library will create chunks based on on the data and the decision is based on simple heuristics. The next parameter used is the compressor parameter which is created in the above line of the code snippet.

The benchmarks that are considered for this Research Projects are Read Time, CPU Usage and Disk Read Count and RAM usage percentage. .

3.5.2 Read Time

For the compressor benchmark evaluation, the whole data set is read and the time taken is measured to read the data. Here the Python **time** library is used to compute the access time for reading the data. For advanced indexing techniques, full data is not read instead a particular data or a block of data is read and the access time is measured.

3.5.3 CPU Usage

CPU usage computation is little tricky and ambiguous since there is no direct method to compute the CPU percentage used by a function. But for this evaluation, the start CPU percentage and the end CPU

percentage is determined after executing a function to determine if the CPU percentage increased/decreased. Positive value denotes the increase in CPU percentage and negative value denotes decrease in CPU percentage. This is determined using the *cpu_percent* function of the *psutil* class in Python.

3.5.4 Disk IO Read Count

This is also similar to computing the CPU usage. The disk read count before and after the execution of the function is calculated and the difference is calculated. For this, *disk_io_counter()* function is used of the *psutil* class in Python. This function returns the total read count operations performed on the disk in a particular time. High number of disk reads indicates heavier read workload on the disk and less number of disk reads indicates less workload on the disk. High number of disk reads may impact disk performance.

3.5.5 RAM Usage Percent

This specifies the RAM usage percent after reading the dataset. The RAM usage percentage is dervied from the *psutil* class and virtual.memory().percent provides the RAM usage percent. Multiple iterations are done for the same query to gather the data for each iteration and stored in a list in python using the *append* function. Finally the data that is collected is the mean value which is implemented using the *mean* function of *statistics* class in Python

4 Experimental Setup and Evaluation

From this experiment, the read time, cpu usage percentage and Disk Read counts metrics are used for evaluation. For the compressor algorithm benchmark evaluation, a data of 1000000000 elements is chosen and also the data is filled using *arrange* function in Python for the whole data set. For the compression algorithm benchmarking and analyse the impact of different configuration, mean values are considered of 20 simultaneous execution of the each configuration and data is plotted accordingly. To achieve this, *xrange* is used to perform the iteration. Read time is plotted as a mean of 20 simultaneous executions. The system that is used for this benchmark evaluation has the following configuration:

- OS: *MAC OS Sonoma 14.3.1*
- Chip: *Apple M1 8 cores (4 performance and 4 efficiency)*
- RAM: *16GB LPDDR4 (Hynix)*
- DISK: *SSD(NVMExpress Model APPLE SSD AP1024Q) with TRIM support*

Below are the software configurations used for this benchmark evaluation:

- Python: *3.12.1 x64 bit architecture*
- IDE: *Visual Studio Code 1.77.1(Universal)*
- Numpy: *1.26.3*
- openpyxl: *3.1.2*
- numcodecs: 0.12.1
- matplotlib: 3.8.2
- psutil: *5.9.8*
- zarr: *2.16.1*

Now, initially the uncompressed zarr is created with no chunking and the baseline benchmarks are generated. Below is the python code snippet to create a uncompressed 3D zarr array with no chunking.

```
#Creating an uncompressed zarr array object with no chunking of shape 1000*1000*1000
z = zarr.create((1000, 1000, 1000), chunks=False, compressor=None, dtype='i4')
#Using arange function of numpy library, array values are filled with integers
z = zarr.array(np.arange(1000000000).reshape(1000, 1000, 1000))
```

The mean values generated after 20 iterations of the same configuration to read the full Zarr dataset(z[:]) are below:

- Read Time: 4.4 seconds
- CPU usage: 25.235%
- Disk Reads count: 19.2
- RAM usage: 35.74%

The above values are generated for the uncompressed variant and no chunking. Now for the subsequent experiments as discussed later in this report are for different configurations of compression algorithms and combining chunking strategies with different advanced indexing techniques.

The below graph is the Read time for five compressor algorithms with parameters *clevel=3*, *shuffle=2* and *blocksize=0*. The x axis of the plot represents different compression algorithms and the y axis represents the read time in seconds which indicates the total read time to read the full dataset.

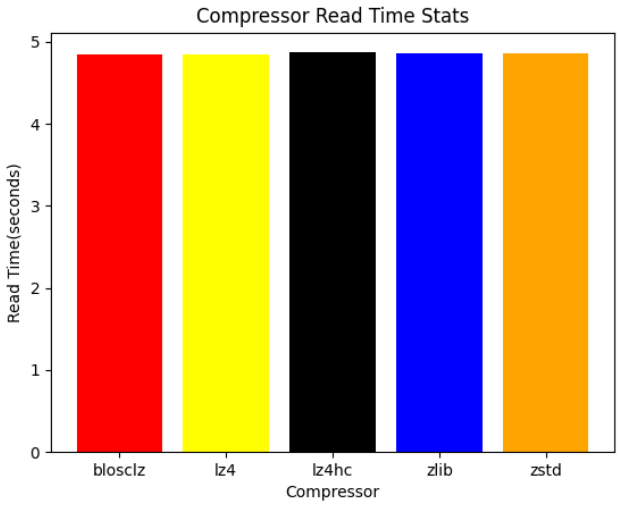


Figure 1: Read Time

In *Figure 1*, different compressor algorithm are tested keeping the default compression parameters(*clevel=3, shuffle=2 and blocksize=0*). For each algorithms(blosclz, lz4, lz4hc, zlib and zstd), read times are measured using the *zarr[:]* syntax to read the full data. To make this data reliable 20 subsequent iterations are extracted and stored in a python list and mean is calculated using the *statistics* library in Python. After analysing the data, it is found that read time of all the algorithms comes around 4.8 seconds to read the full data set of 1000000000 elements. Now, this same experiment has been done on a different dataset where all the values are set to 0 of the dataset. To read that dataset also same read time(*1.3 seconds*) has been measured for all the algorithms. Hence, the effect of different compression algorithms does not produce any effect on the read time of the dataset. Here it is mention worthy that the compression algorithms are tested for above plot based on the default compression parameters. The different compression parameters and their definition is discussed in section 3.2

The below graph is mean CPU percent usage as discussed section 3.5.3 in for the five algorithms while reading all the elements.The x axis of the plot represents different compression algorithms and y axis of the plot represents the average CPU usage in percentage for reading the whole dataset.

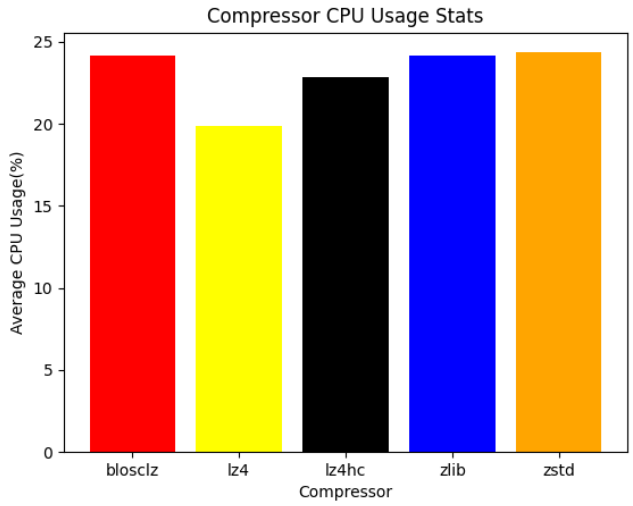


Figure 2: CPU Usage

In *Figure 2*, all the compression algorithms and Average CPU usage percentage is plotted. The compression parameters are set to default configuration with *clevel=3, shuffle=2 and blocksize=0.* From the respective graph, it can be seen that the mean value of CPU Usage percentage is less for the lz4 algorithm and almost similar for other compression algorithms. Now, for low CPU percent usage, it can be inferred that CPU is not utilized at its maximum level and has spare CPU available for other tasks. High CPU usage percentage means that CPU has less resource available and has less shared availability for other processes. Hence to read the same dataset, it can be inferred from the plot that lz4 performs better in CPU utilization than the other algorithms.

The next graph is for the Disk Read Count discussed in 3.5.4 performed to read all the elements in the multi dimensional array of 1000000000 elements. Since there is no way to detect the exact Disk Read Count for a particular function, the Disk Read Count is calculated by subtracting the initial Disk Read Count and the final Disk Read Count. To detect the Disk Read count or no of reads on the disk, *psutil* from Python is used. The *disk_io_counters* function of the *psutil* library in Python returns tuple containing the system wide disk IO statistics such as *no of reads, no of writes, read bytes and write bytes.* For this particular experiment, read count is measured to get the total count of read operations performed on a disk or a storage device for our particular compression configuration. A large no of read operations on a disk indicates high workload on the disk. The x axis of the plot represents the different compression algorithms and y axis of the plot represents the disk read count for each algorithm.

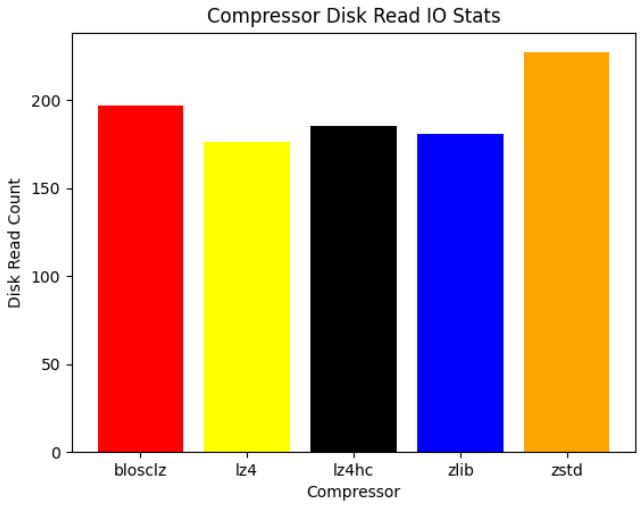


Figure 3: Disk Read Count

From the *Fig. 3*, it can be inferred that zlib and lz4 has the least read count on the disk. To access the whole dataset, *zlib* compression algorithm requires least read operations on the disk unlike *zstd* algorithm which has more read operations on the disk. Hence, to read the same dataset, *zstd* access the disk more number of times than *zlib* or *lz4*. More no of reads may impact disk performance.

The next graph is for depicting the RAM usage percent of different compression algorithm. x axis represents the different compression algorithm and y axis represents the RAM Usage percent for each algorithm.

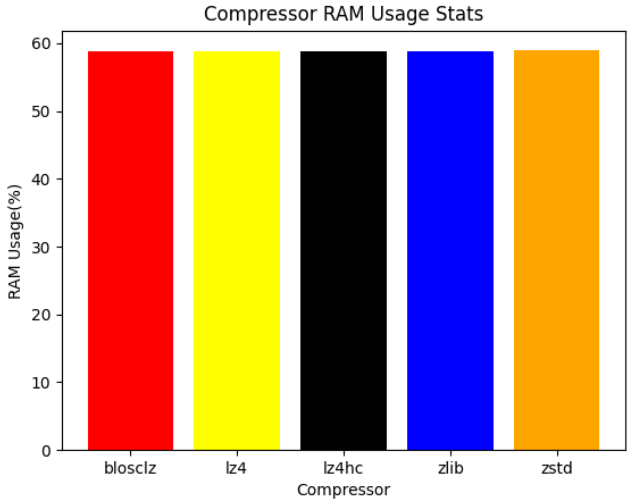


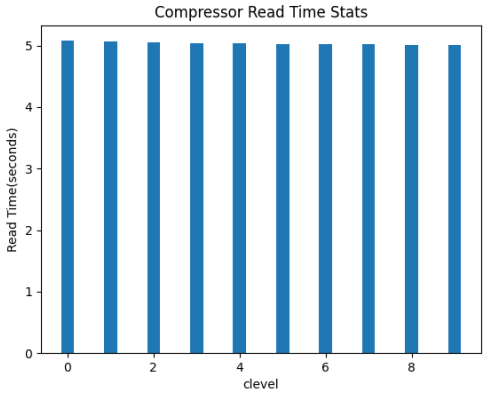
Figure 4: RAM Usage

From *Fig. 4*, it can be seen that all the algorithms has similar RAM percentage usage. The value of

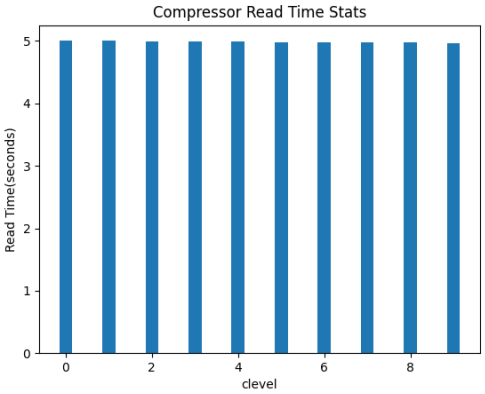
each RAM usage percentage comes around 58% overall. Also for other configuration changes it is evaluated that the compression algorithms does not change much. All the RAM usage percentage remains identical for all the algorithms. The other plots are not shown by changing the clevel, shuffle and blocksize for compression algorithms since for all, the RAM usage percent remains same throughout.

The next following graphs are for the five algorithms where benchmarking is done based on the compression level(*clevel*) of the compressor ranging from 0 - 9. The other parameters *shuffle* and *blocksize* are both set to 0. The graphs represent the impact of the compression level (*clevel*) on the Read Time. To access the read time taken for the Zarr object z to read the full dataset, python *time* library is used. The below code snippet represents how the read time has been measured in python:

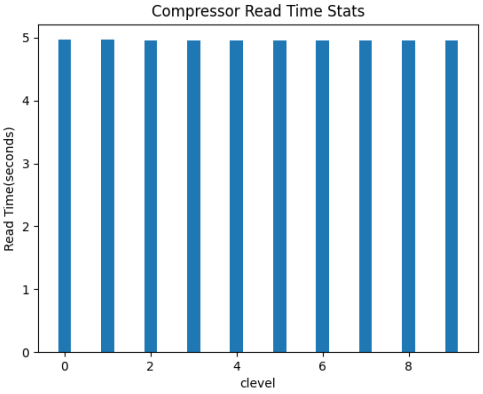
```
#Initially start time is calculated
start_time=time.time()
#Whole data set is read
zarr[:]
#End time is calculated
end_time=time.time()
#Read time is calculated subtracting start_time from end_time
read_time=(end_time - start_time)
#Finally subsequent 20 iterations of the read times are appended into a Python list
arr_read_time.append(read_time)
#Finally the mean value of read_time is calculated using the \textit{mean} function
                                                of the \textit{statistics} library. Also
                                                 the values are rounded to the 3 digit
                                                decimal place.
compressor_sheet.cell(row=index+2, column=5).value = round(mean(arr_read_time), 3)
```



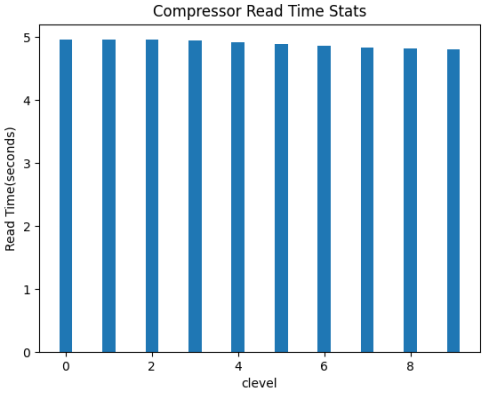
(a) blosclz



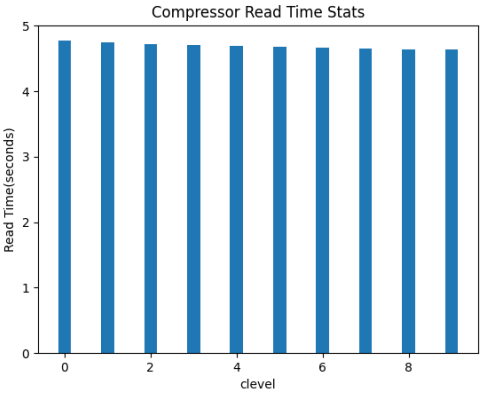
(b) lz4



(c) lz4hc



(d) zlib



(e) zstd

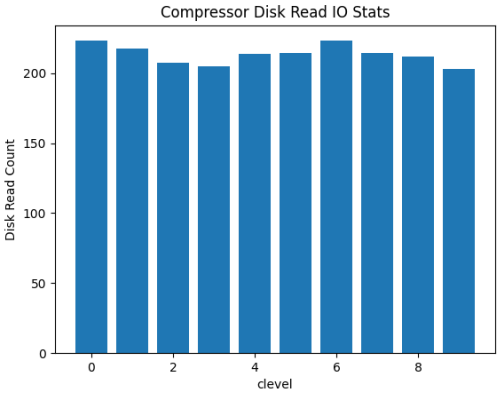
Figure 5: Impact of Compression Level on Read Time

Now in *Figure 4*, x axis denotes different compression level ranging from 0 to 9 and y axis denotes the subsequent access time to read the full dataset. It can be inferred from the plots that compression level as discussed in section 3.2.2 does not has impact on the Read time for the whole dataset. By changing the

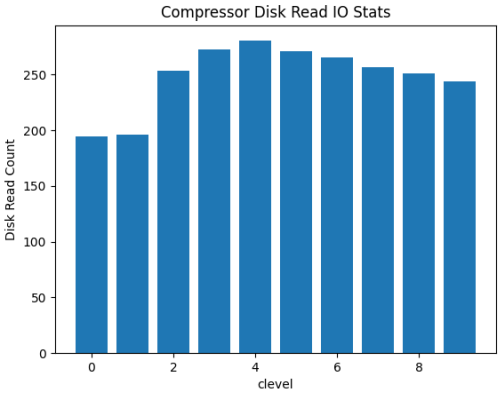
different compression level from 0 to 9, the read times for each algorithms does not vary much. Hence, by varying the compression level, the read times of each algorithm remains almost identical. The mean value comes around 5 seconds for all the compression level

Now the next set of graphs will depict the impact of Compression Level on Disk Read Count for the five algorithms. Disk Read Count is discussed in section 3.5.4. The x axis of the plot represents different compression levels as discussed in 3.2.2 and the y axis represents no of disk reads for each *clevel*. To achieve this in Python, the following code snippet is implemented.

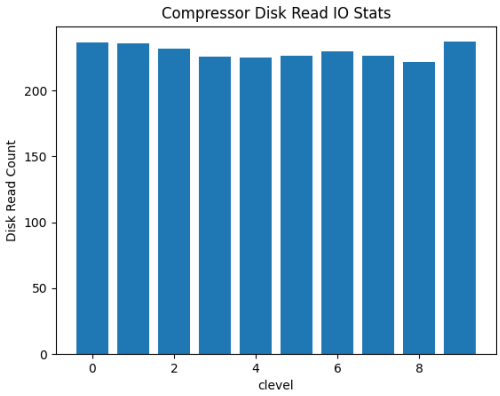
```
#Get the initial disk read count
start_reads = psutil.disk_io_counters().read_count
#Reading the whole dataset
zarr[:]
#Get the final disk read count
end_reads = psutil.disk_io_counters().read_count
#Subtracting the start read count from end read count
reads = end_reads - start_reads
#Appending the read count to a Python list for calculating the mean of 20 iterations
arr_disk_reads.append(reads)
#Mean is calculated and stored in the output sheet for evaluation
compressor_sheet.cell(row=index+2, column=7).value = round(mean(arr_disk_reads), 3)
```



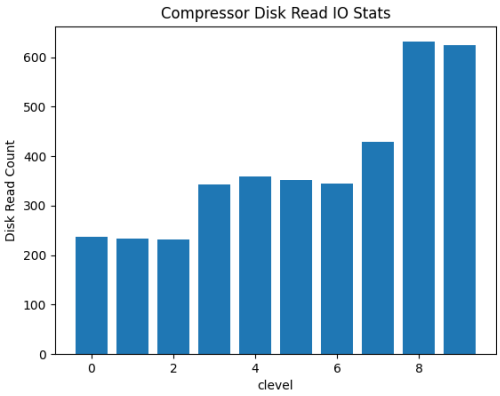
(a) blosclz



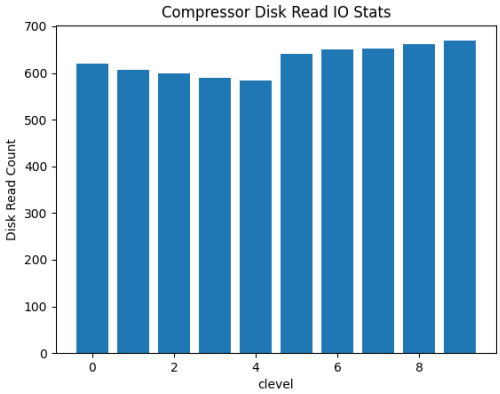
(b) lz4



(c) lz4hc



(d) zlib



(e) zstd

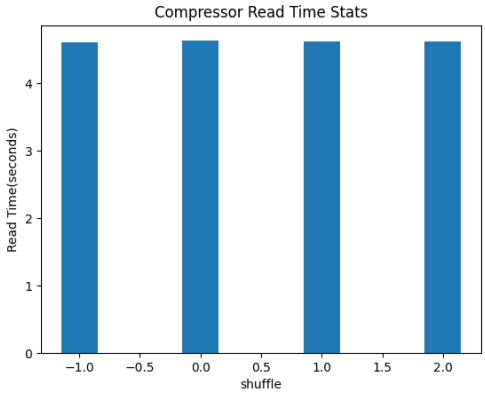
Figure 6: Impact of Compression Level on Disk Read Count

Now, from *Fig. a*, impact of compression level on the disk read count of blosclz algorithm is evaluated. It can be noted that the mean values of read count is almost similar for all the compression levels where compression level 9 has the least read count on disk. Now in *Fig. b*, it can be noted that compression

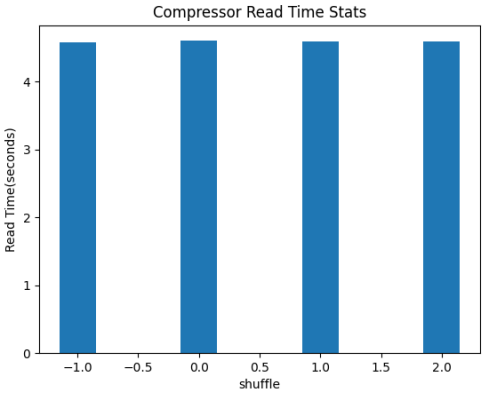
level 0 and 1 has less no of reads on the disk for the same data set for lz4 algorithm. In *Fig. c*, for lz4hc algorithm, the read counts are identical for all compression levels and nothing much can be inferred about the impact. In *Fig. d*, the variation of the impact is much more visible where compression levels 0, 1 and 2 has less read counts for reading the whole dataset rather than other compression levels. Also it can be seen that for zlib compression, the disk read count is directly proportional to compression levels. As compression levels increases, disk read counts also increases simultaneously. In *Fig. e*, it can be seen that as the compression level increases the disk read count decreases till level 4 and increases from level 5 to level 9.

Now the following graphs contains the visual representation of the impact of *shuffle* on Read Time of five compression algorithms. 0(*No Shuffle*), 1(*Shuffle*), 2(*BitShuffle*), -1(*AutoShuffle*). Now the below python code snippet is used to achieve this.

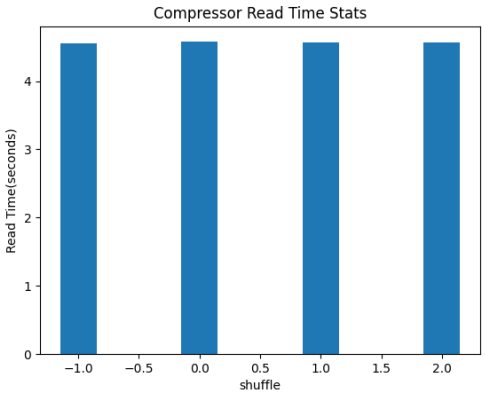
```
#Initializing the compressor parameter and configuring it to pass that in the Zarr
                                                creation parameter. Here the shuffle values(
                                                0, 1, 2, -1) are passed as a parameter from
                                                excel configuration sheet.
compressor = Blosc(cname=value[0].value, clevel=int(value[1].value), shuffle=int(value[2]
                                                .value))
zarr = zarr.create((1000, 1000, 1000), chunks=True, compressor=compressor, dtype='i4')
start_time = time.time()
zarr[:]
end_time = time.time()
read_time = end_time - start_time
#Appending the read time to a python list
arr_read_time.append(read_time)
#Finding the mean of the read time
round(mean(arr_read_time), 3)
```



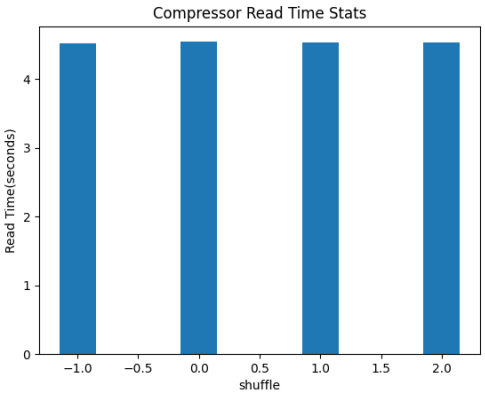
(a) blosclz



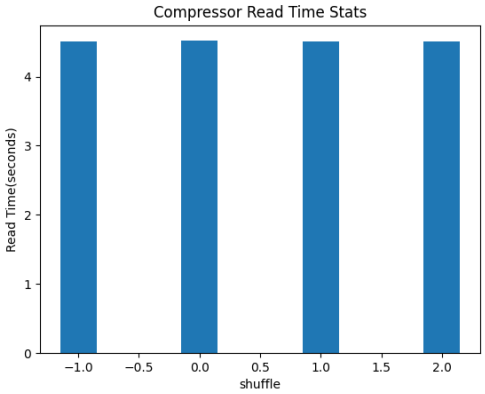
(b) lz4



(c) lz4hc



(d) zlib



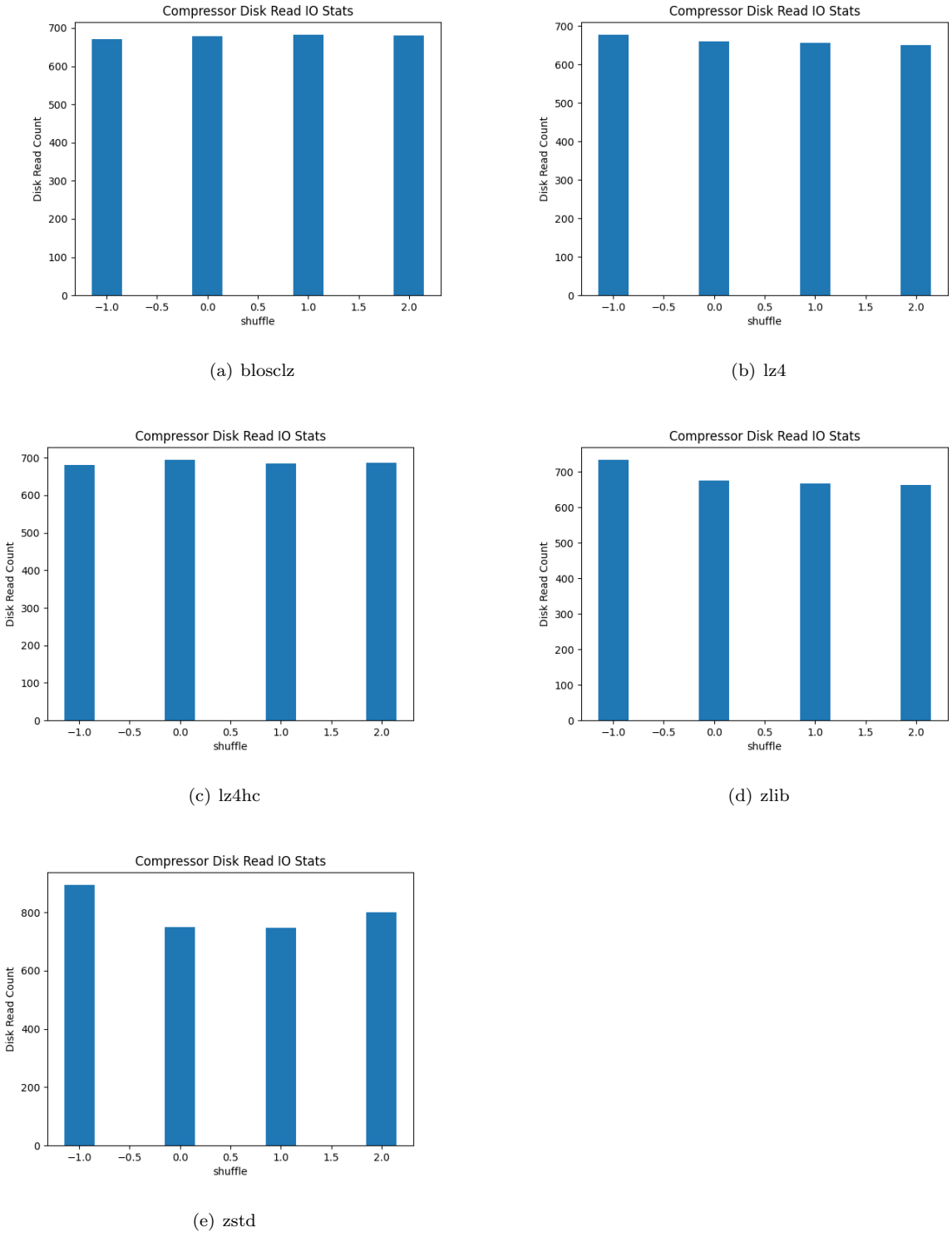
(e) zstd

Figure 7: Impact of shuffling on Read Time

Now in *Fig. 7*, it can be seen that shuffling parameter has no such impact on the Read Time of the data. For all the algorithms, Read time is measured to be around 4.5 seconds to 4.6 seconds.

Now the following graphs contains the visual representation of the impact of *shuffle* on Disk Read

Count of five compression algorithms. x axis of the plots represents the shuffle configurations and y axis is the disk read count for each shuffle configurations. The definition of each shuffle parameters are discussed in section 3.2.3



(a) blosclz

(b) lz4

(c) lz4hc

(d) zlib

(e) zstd

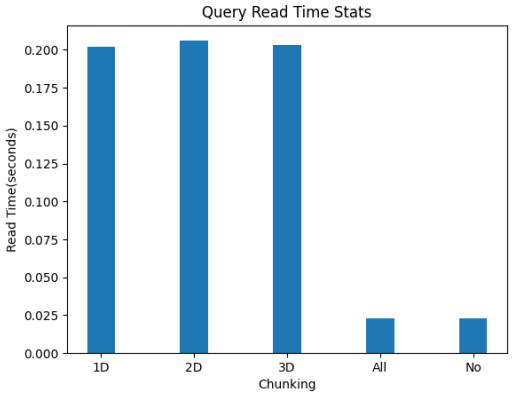
Figure 8: Impact of shuffling on Disk Read Count

Now, from the *Fig. a, b and c*, it can be inferred that shuffle parameters has relatively same impact overall on the disk read count. But for *Fig. d*, it can be noticed that *BitShuffle* has less read count overall than the other shuffle parameters(No Shuffle, Autoshuffle and Default shuffle). Also in *Fig. e* it can be identified that *No shuffle* configurations has less no of reads in the disk than the other configurations. Also it can be seen that the *Autoshuffle(-1)* has more no of read counts on disk than other configurations overall for all the compressor algorithms.

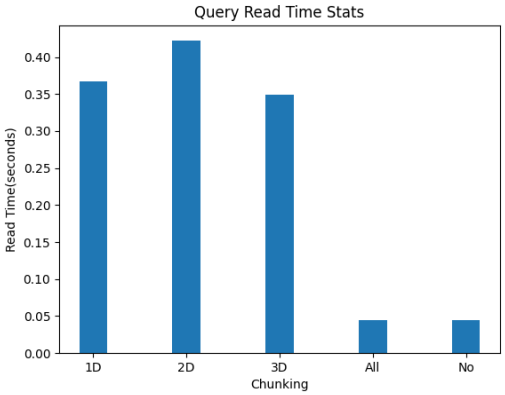
Now the next set of graphs will be focused on the benchmark evaluation of queries along with a set of compression parameters and chunking strategies. So initially the benchmarking is performed on a 3-dimensional data of 1000000000 elements with shape 1000*1000*1000.

```
compressor = Blosc(cname=value[1].value, clevel=int(value[2].value), shuffle=int(
                                                value[3].value))
#Chunking is done on the 1st dimension
if value[5].value == "1D":
    zarr_blosc = zarr.array(np.arange(1000000000, dtype='i4').reshape(1000, 1000,
                                                1000), chunks=(100, None, None),
                                                compressor=compressor)
#Chunking is done on the 2nd dimension
if value[5].value == "2D":
    zarr_blosc = zarr.array(np.arange(1000000000, dtype='i4').reshape(1000, 1000,
                                                1000), chunks=(None, 100, None),
                                                compressor=compressor)
#Chunking is done on the 3rd dimension
if value[5].value == "3D":
    zarr_blosc = zarr.array(np.arange(1000000000, dtype='i4').reshape(1000, 1000,
                                                1000), chunks=(None, None, 100),
                                                compressor=compressor)
#CHunking is done on all dimensions
if value[5].value == "All":
    zarr_blosc = zarr.array(np.arange(1000000000, dtype='i4').reshape(1000, 1000,
                                                1000), chunks=True, compressor=
                                                compressor)
#No chunking is done
if value[5].value == "None":
    zarr_blosc = zarr.array(np.arange(1000000000, dtype='i4').reshape(1000, 1000,
                                                1000), chunks=False, compressor=
                                                compressor)
```

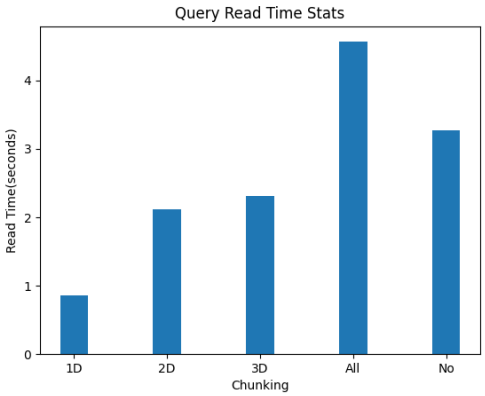
Now the below *Fig. 9* represents the read time values of different chunking strategies(1D, 2D, 3D, All and No). In the below figure, x axis represents the chunking strategies and y axis represents the Read Time values.



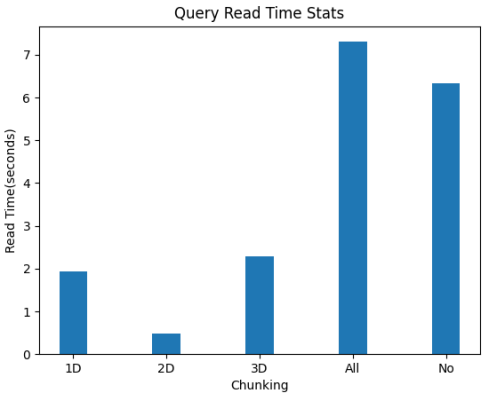
(a) Coordinate Selection



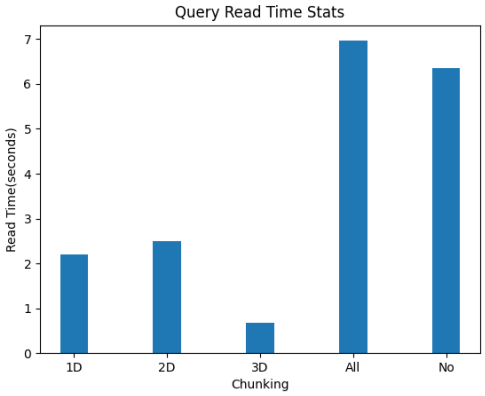
(b) vindex



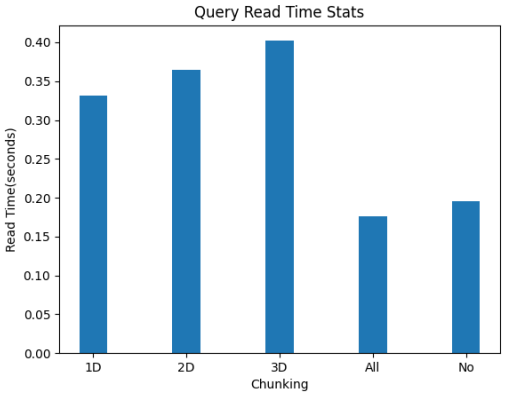
(c) Orthogonal Selection 1



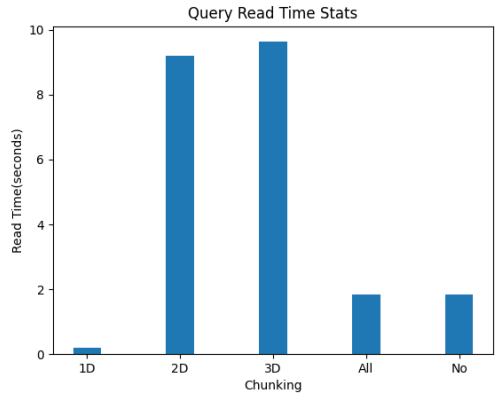
(d) Orthogonal Selection 2



(e) Orthogonal Selection 3



(f) Orthogonal Selection 4



(g) Block Selection

Figure 9: Impact of Chunking Strategies on Read Time of Advanced Indexing Techniques

Now, in *Fig a*, coordinate selection indexing technique is considered. Below is the python snippet for the indexing.

```
# Query to be implemented
if value[0].value == "Coordinate_Selection":
zarr_blosc.get_coordinate_selection(([0, 2], [1, 3], [3, 3]))
```

Now, in coordinate selection to access a particular data, coordinates of all dimensions needs to be passed. Here in this experiment three coordinates needs to be passed since the data is a 3D multidimensional array. Now as per the figure, it can be noticed that if the chunking is done on all the dimensions or No chunking is done then the mean access time for reading the particular data is decreased rather than chunking in any particular dimension.

Now in *Fig. b*, vindex indexing is used. The python snippet for vindexing is:

```
if value[0].value == "vindex":
        zarr_blosc.vindex[[823, 210], [283, 173], [965, 112]]
```

vindex property is similar to coordinate selection where coordinates are passed to access a particular data. Now from the figure, it can be inferred that as similar to Coordinate selection to access data using coodinates, Chunking on all dimensions and NO chunking is preferable since it yields less read time values than chunking on each dimensions.

In *Fig. c*, Orthogonal selection indexing technique is used. Below is the python implementation.

```
if value[0].value == "Orthogonal_Selection1":
zarr_blosc.get_orthogonal_selection(([172, 5], slice(None), slice(None)))
```

Now using Orthogonal selection, any data from any particular dimension can be accessed independently. *Orthogonal_selection1* refers to selection on the 1st dimension of the array. Now from the figure, it can be assessed that chunking on the 1st dimension to select the data from the 1st dimension, the access time is less rather than chunking on other dimension. Also, it can be accessed that chunking all dimensions is also not useful for this indexing technique.

Now, in *Fig. d*, Orthogonal selection is performed on the 2nd dimension on the data. Below is the python code for the implementation.

```
# only reads the 2nd dimensional data
    if value[0].value == "Orthogonal_Selection2":
        zarr_blosc.get_orthogonal_selection((slice(None), [5, 190], slice(None)))
```

Similary, it can be assessed from the figure that chunking on 2nd dimension is much more useful to read the data from the 2nd dimension.

Now, from *Fig. e* it can be seen that to access the data from the 3rd dimension, chunking on the same dimension is much more efficient which aligns with our previous evaluation results for Orthogonal selection technique.

Now in *Fig. f* Orthogonal selection is used where data is accessed from all the dimensions. Below is the python snippet for the implementation.

```
# reads the data from all the dimensions
    if value[0].value == "Orthogonal_Selection4":
        zarr_blosc.get_orthogonal_selection(([126, 754], [900, 25], [786, 112]))
```

Now from the figure, it can be seen that chunking on all dimensions is efficient if data is accessed from all dimensions.

Now from *Fig. g*, block selection is used which allows selections of whole chunks based on their logical indices along each dimension of an array. Below is the python implementation for the same.

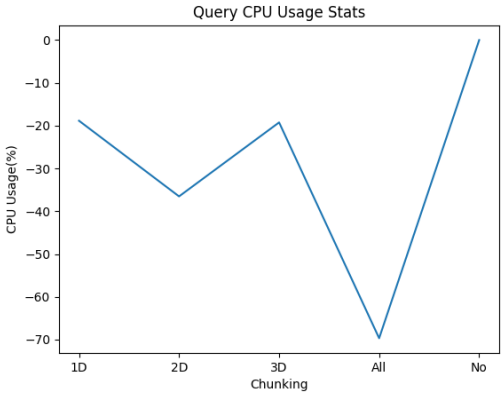
```
if value[0].value == "Block_Selection":
        zarr_blosc.get_block_selection(0)
```

Now from the code it can be seen that, the logical index 0 is selected which is the first subset of chunk. Now similarly from the figure it can be seen that chunking on the first dimension also results in fast access speed than the other dimensions.

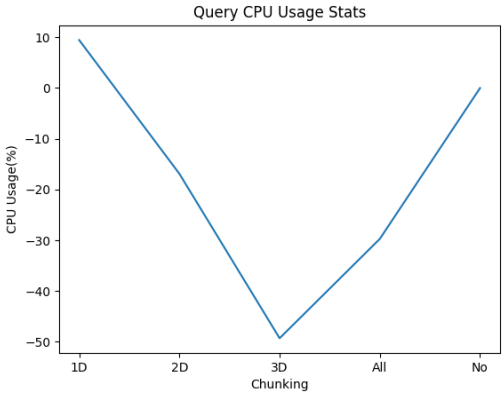
The next figure depicts the CPU usage for executing each query. For each query, CPU utilization is checked by checking the percentage of CPU percent before the execution of the query and after the query. The CPU percentage before the execution of the query is subtracted from the CPU percentage after the execution of the query to get the CPU percentage use curing the execution. Below is the python implementation for the same.

```
import psutil
import time
# Start measuring CPU usage
start_cpu_usage = psutil.cpu_percent()
if value[0].value == "Orthogonal_Selection4":
        zarr_blosc.get_orthogonal_selection(([126, 754], [900, 25], [786, 112]))
end_cpu_usage = psutil.cpu_percent()
average_cpu_usage = (end_cpu_usage - start_cpu_usage)
arr_cpu_usage.append(average_cpu_usage)
query_sheet.cell(row=index+2, column=10).value = round(mean(arr_cpu_usage), 3)
```

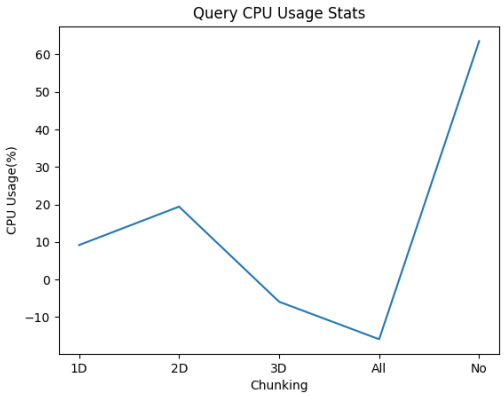
For the CPU usage calculation, 10 iterations are considered and the mean is selected for each query. x axis contains different chunking configurations and y axis contains the different CPU usage in percentage.



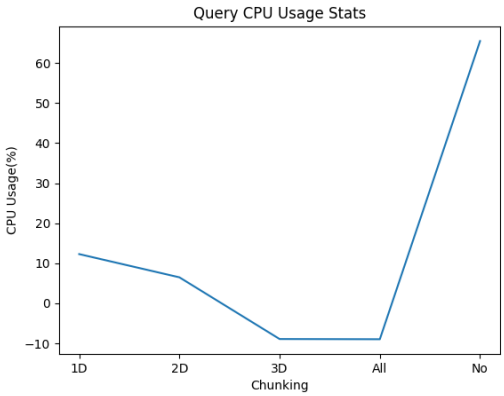
(a) Coordinate Selection



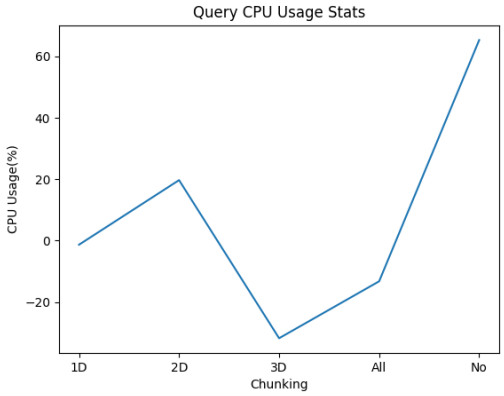
(b) vindex



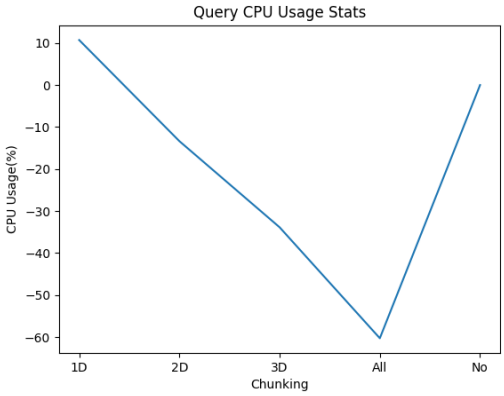
(c) Orthogonal Selection 1



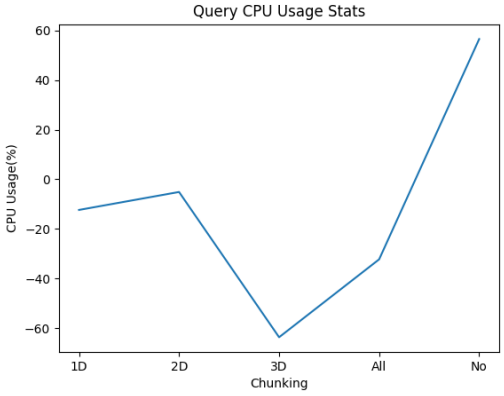
(d) Orthogonal Selection 2



(e) Orthogonal Selection 3



(f) Orthogonal Selection 4



(g) Block Selection

Figure 10: Impact of Chunking Strategies on CPU Usage of Advanced Indexing Techniques

In *Fig. a*, for the coordinate selection query, it can be seen that the CPU usage percentage is less if chunking is done for all the dimensions. Whearas it can be seen that if there is no chunking involved in any dimensions then CPU usage percentage is higher after executing the query.

In *Fig. b*, for the vindex selection strategy it can be seen that if the chunking is done on the 3rd dimension of the array, then the CPU usage percentage is least. If the chunking is done on the 1st dimension then the CPU usage percentage is highest.

In *Fig. c*, it can be seen that for Orthogonal selection of data only on the first dimension, it is seen that for chunking among all dimension, the CPU usage percentage is lowest. If there is no chunking at all on any of the dimensions, then the CPU usage is highest for this selection strategy.

In *Fig. d*, where the Orthogonal selection on the 2nd dimension, it can be seen that the lowest CPU usage is for the chunking on all dimensions and also on the 3rd dimension. But the higher CPU usage is if there is no chunking at all in the array.

In *Fig. e*, for the Orthogonal selection only on the 3rd dimension, it can be seen that the CPU usage is least for if the chunking is performed on the 3rd dimension and highest if there is no chunking at all on any dimension of the array.

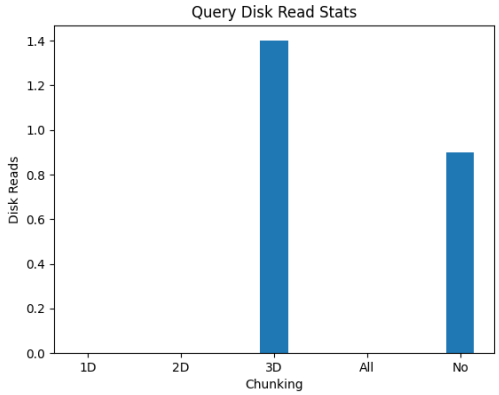
In *Fig. f*, it can be seen that if Orthogonal selection is performed on all the dimensions of the data, then the least CPU usage is for the chuking on all dimensions and highest if chunking is done on the 1st dimension.

In *Fig. g*, for block selection it can be seen that if the chunking is done on only 3rd dimension, the CPU usage is least but highest if there is no chunking at all on any dimensions.

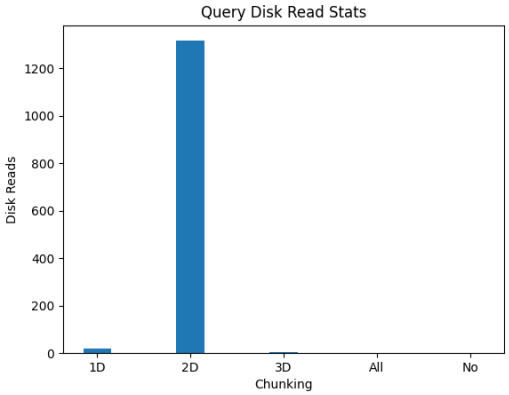
From all the above plots it can be inferred that if there is no chunking at all, then the CPU usage to access the data is highest overall.

Now the next set of graphs depicts the total read count operations on the disk for each query. Below is an example Python code snippet for the implementation.

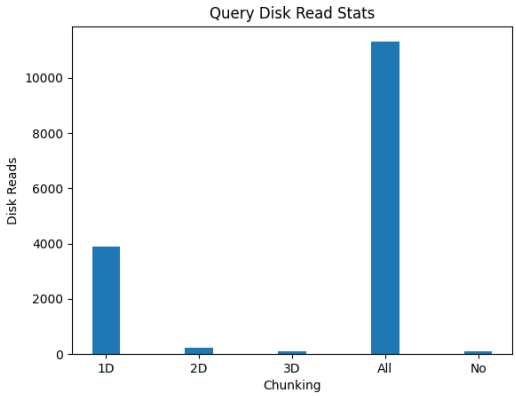
```
import psutil
import time
#Initial Disk I/O usage
start_reads = psutil.disk_io_counters().read_count
if value[0].value == "Orthogonal_Selection4":
        zarr_blosc.get_orthogonal_selection(([126, 754], [900, 25], [786, 112]))
# Get final disk I/O usage
    end_reads = psutil.disk_io_counters().read_count
read_count = (end_reads - start_reads)
arr_read_count.append(read_count)
query_sheet.cell(row=index+2, column=13).value = round(mean(arr_read_count), 3)
```



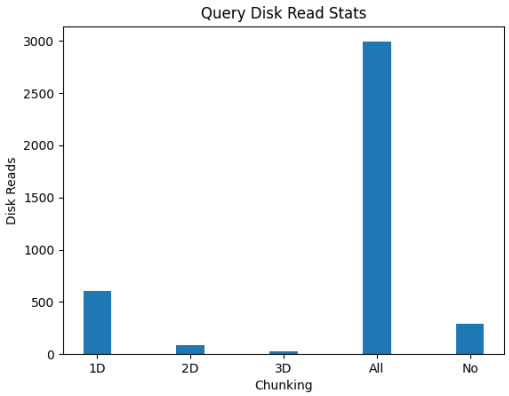
(a) Coordinate Selection



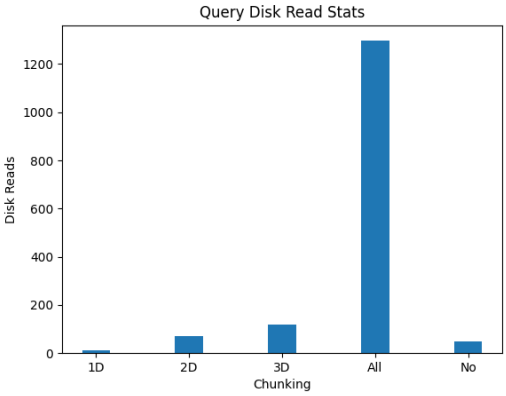
(b) vindex



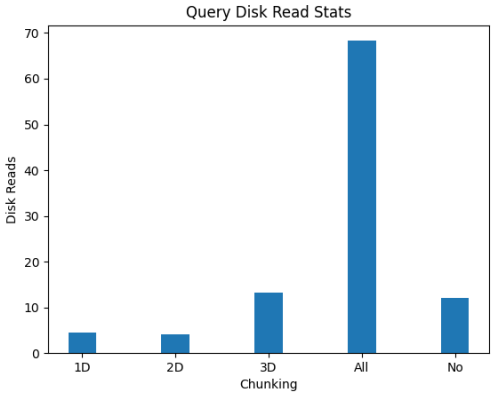
(c) Orthogonal Selection 1



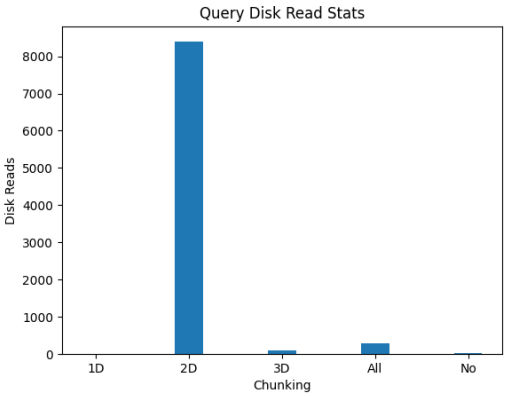
(d) Orthogonal Selection 2



(e) Orthogonal Selection 3



(f) Orthogonal Selection 4



(g) Block Selection

Figure 11: Impact of Chunking Strategies on Disk Read of Advanced Indexing Techniques

From *Fig. a*, it can be inferred that if the chunking is done on the 3rd dimension, then the coordinate selection has highest no of disk reads rather than other chunking on other dimensions.

From *Fig. b*, it can be assessed that if the chunking is done on the 2nd dimension, then for the vindex selection strategy, the disk reads has highest disk read count.

From *Fig. c*, it can be seen that if the chunking is done on all the dimensions, then the disk reads has the highest count for Orthogonal selection performed on the 1st dimension.

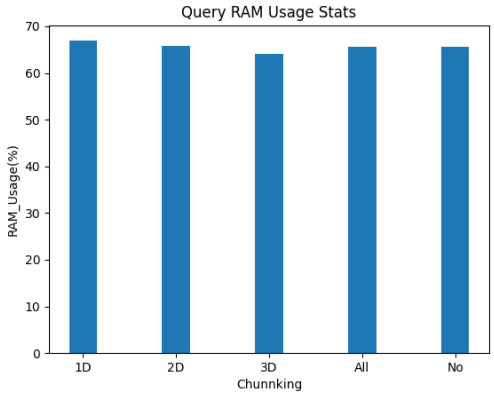
From *Fig. d*, it can be noticed that for the chunking strategy with all dimensions, orthogonal selection query on the second dimension has the highest number of disk reads.

Similarly for all the other orthogonal selection, it can inferred from the *Fig. e, Fig. f* that for chunking on all dimensions has the highest no of disk reads.

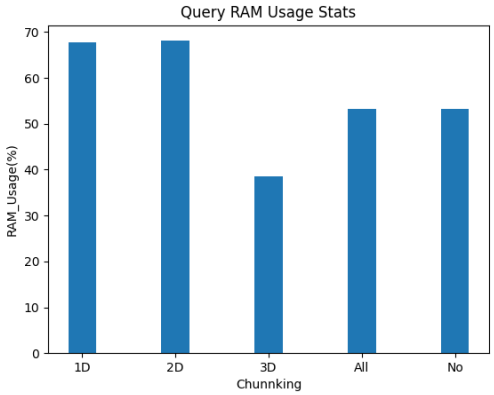
For *Fig. g*, it can be inferred that chunking on the 2nd dimension has the highest no of disk reads for the block selection indexing.

Now the next set of graphs is plotted for evaluating the RAM usage(%) for different chunking strategies on different queries. x axis of the graphs determines the RAM Usage and y axis of the graphs determines the chunking strategies for each of the queries. A sample Python implementation is provided below.

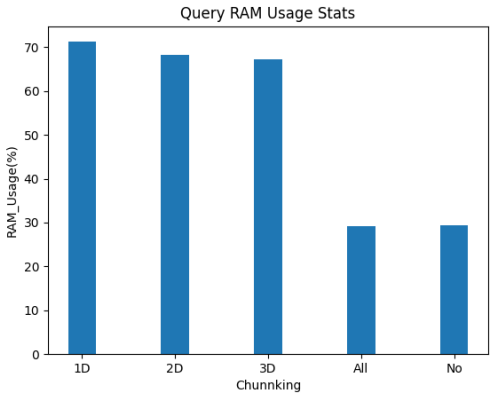
```
# Query to be implemented
    if value[0].value == "Coordinate_Selection":
        zarr_blosc.get_coordinate_selection(([0, 2], [1, 3], [3, 3]))
    # Evaluating the RAM usage percent after the execution of the query
    mem = psutil.virtual_memory()
    # 10 iteration values are stored in a Python list
    arr_mem_percent.append(mem.percent)
    # Extracting the mean value and storing in the output sheet
    query_sheet.cell(row=index+2, column=14).value = round(mean(arr_mem_percent), 3)
```



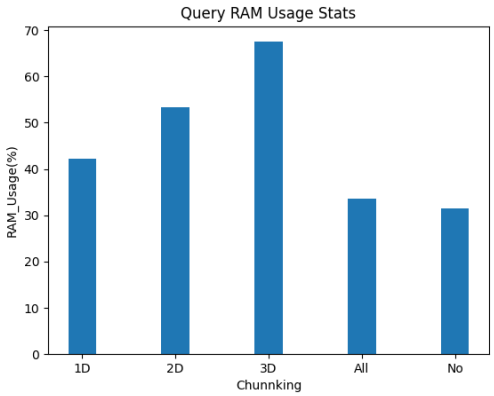
(a) Coordinate Selection



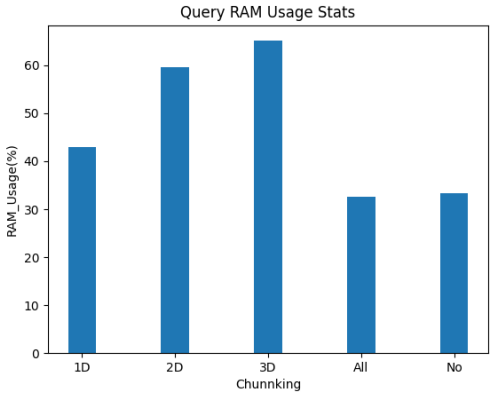
(b) vindex



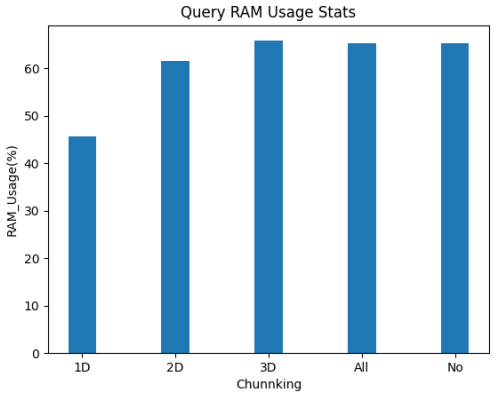
(c) Orthogonal Selection 1



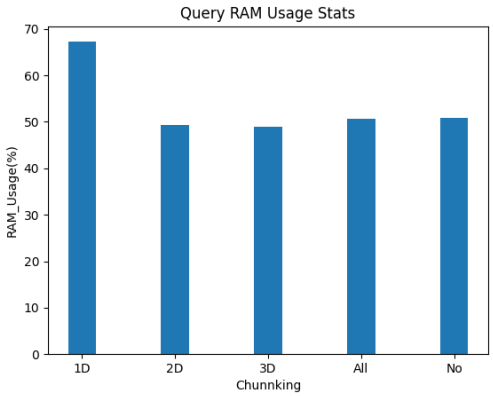
(d) Orthogonal Selection 2



(e) Orthogonal Selection 3



(f) Orthogonal Selection 4



(g) Block Selection

Figure 12: Impact of Chunking Strategies on RAM usage of Advanced Indexing Techniques

In *Fig. a*, for the coordinate selection, all the chunking strategies on all dimensions are almost identical.

In *Fig. b*, for the vindex selection, RAM usage percentage is lowest for the chunking on all dimensions and highest RAM usage is for chunking on the 1st and 2nd dimensions.

In *Fig. c*, for the Orthogonal selection only on the 1st dimension, RAM usage percent is less both when chunking is done for all the dimensions and for No chunking. But if the chunking is done only on 1st, 2nd and 3rd dimensions, then the RAM usage percent is higher.

In *Fig. d*, for the Orthogonal selection on the 2nd dimension, the RAM usage is higher for the chunking on the 3rd dimension and lowest if no chunking is done at all or if chunking is done all the dimensions.

In *Fig. e*, for Orthogonal selection on the 3rd dimension, RAM usage percent is less when there is chunking on all dimensions and when no chunking is there at all. RAM usage is higher if chunking is done on the 3rd dimension.

In *Fig. f*, for Orthogonal selection on all the dimension, the RAM usage is higher for chunking on the 3rd dimension, chunking on all the dimensions and if no chunking is present. The RAM usage is lowest if the chunking is done on the 1st dimension.

In *Fig. g*, for the block selection query, the RAM usage is higher if the chunking is done in the 1st dimension and identical for all other chunking strategies.

5 Conclusion

All the benchmark evaluation Python programs can be found on the codebase link here. There is already an experimented conducted on the Read Time evaluation of various file formats such as netcdf, hdf5 and zarr [1]. But it is more of a file format evaluation rather than access library evaluation. In this research Project report impact of different configurations on the Zarr library is discussed. According to the analysis, it is imperative that Read Time does not vary that much for different configurations of compression algorithms but CPU Usage and Disk Read Count may vary for different configurations. For different combination of chunking strategy and compression algorithm, the Read Time values varies for different advanced indexing techniques. The future scope of this Research Project can be extended to test all the permutations of Chunking strategies with all compression algorithms and varying the different compression algorithms parameters. Also different compression classes can be tested apart from the default *blosc* algorithm.

References

[1] Sriniket Ambatipudi and Suren Byna. A comparison of hdf5, zarr, and netcdf4 in performing common i/o operations. *arXiv preprint arXiv:2207.09503*, 2022.

[2] Peter Baumann, Dimitar Misev, Vlad Merticariu, and Bang Pham Huu. Array databases: concepts, standards, implementations. *Journal of Big Data*, 8(1):1–61, 2021.

[3] Ekow J Otoo, Doron Rotem, and Sridhar Seshadri. Optimal chunking of large multidimensional arrays for data warehousing. In *Proceedings of the ACM tenth international workshop on Data warehousing and OLAP*, pages 25–32, 2007.

[4] Doron Rotem, Ekow J Otoo, and Sridhar Seshadri. Chunking of large multidimensional arrays. 2007.

[5] Max Zeyen, James Ahrens, Hans Hagen, Katrin Heitmann, and Salman Habib. Cosmological particle data compression in practice. In *Proceedings of the In Situ Infrastructures on Enabling Extreme-Scale Analysis and Visualization*, pages 12–16. 2017.