Performance evaluation of access libraries for raster data processing

Research Project presentation for the degree of M.S in Research in Computer and Systems Engineering.

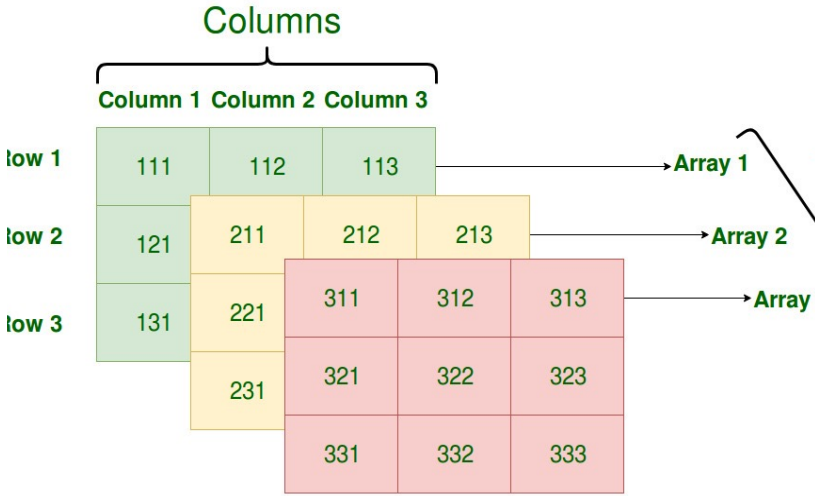
Presented By:	Supervised By:	Department:
M. Sc. Sagnik Sarkar	Dr. Marcus Paradies	Database and Information Systems Group

The SPIRIT of science

TECHNISCHE UNIVERSITÄT ILMENAU

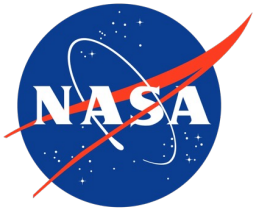
What is Raster Data?

- Multi-dimensional array also termed as Raster Data, Tensors or Gridded data.
- Used in Earth Sciences, Space Sciences, Life Sciences.



Zarr

- Python Library for storing large multi-dimensional arrays.
- Developed by Alistair Miles and his team.
- Datasets can be stored in chunks.
- Adopted by several organizations.



Microsoft

LEAP

NCAR
NATIONAL CENTER FOR ATMOSPHERIC RESEARCH

hhmi | janelia
Research Campus

Benchmarking Parameters

- Read Time
- CPU Usage (psutil.cpu_percent())
- Disk IO Read Count (psutil.disk_io_counters().read_count)
- RAM Usage (psutil.virtual_memory().percent)

Hardware and Software Configurations

Hardware

- OS: MAC OS Sonoma 14.3.1
- Chip: Apple M1 8 cores (4 performance and 4 efficiency)
- RAM: 16GB LPDDR4 (Hynix)
- Disk: SSD (NVMe model Apple SSD AP1024Q) with TRIM support

Software

- Python: 3.12.1 x64 bit architecture
- IDE: VS Code 1.77.1
- Numpy: 1.26.3
- Openpyxl: 3.1.2
- Numcodecs: 0.12.1
- Matplotlib: 3.8.2
- Psutil: 5.9.8
- Zarr: 2.16.1

DataSet

A 3D dataset is created using Zarr library containing 1000000000 elements of shape 1000*1000*1000.

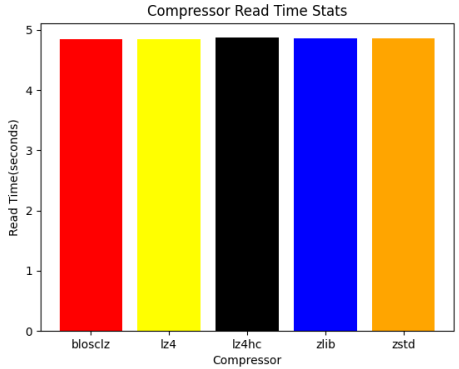
```
compressor = Blosc(cname=value[0].value,clevel=int(value[1].value), shuffle=int(value
                                                    [2].value))
zarr_blosc = zarr.create((1000, 1000, 1000), chunks=True, compressor=compressor,
                                                    dtype='i4')
zarr_blosc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

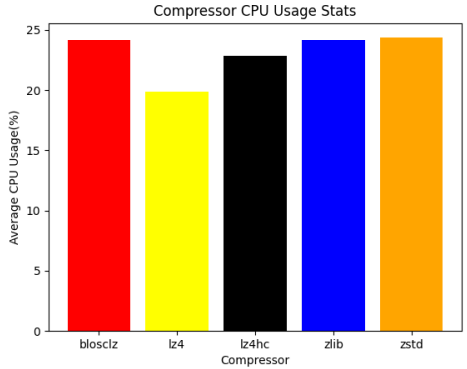
Benchmark Evaluation

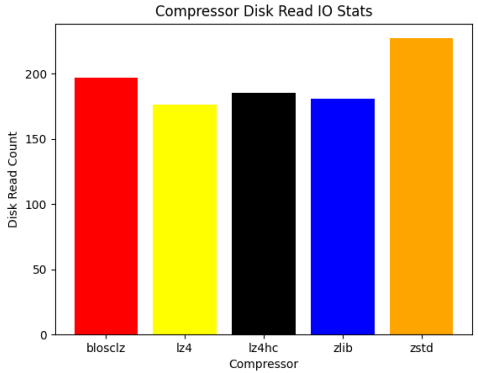
The x-axis contains different compression algorithms and y-axis contains the different benchmark parameters such as Read Time, CPU Usage etc.

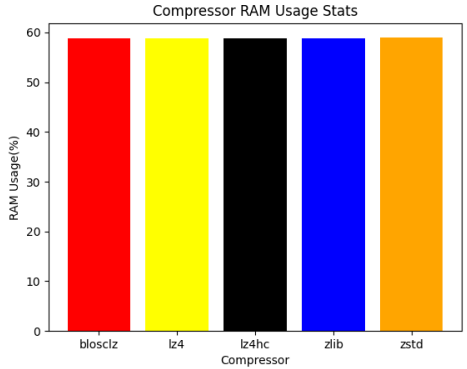
The configuration used for this evaluation are *clevel=3*, *shuffle=2* and *blocksize=0.*

For each compression algorithms the values are the mean value of 20 iterations.







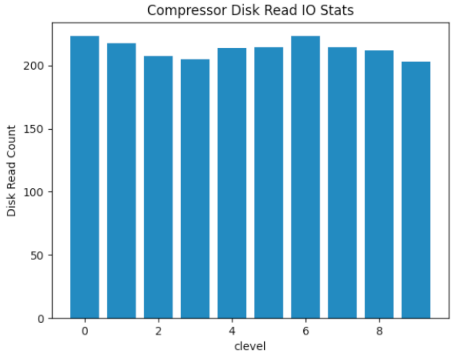


Example Code Snippet

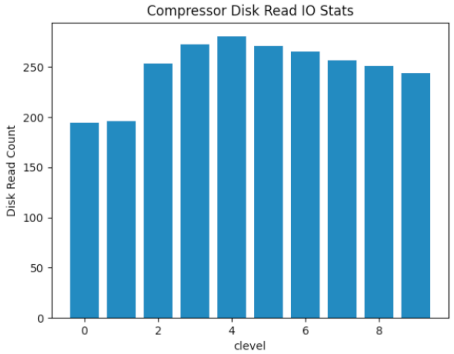
Below is a sample code snippet how the read time is measured while reading the whole data.

```
#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)
```

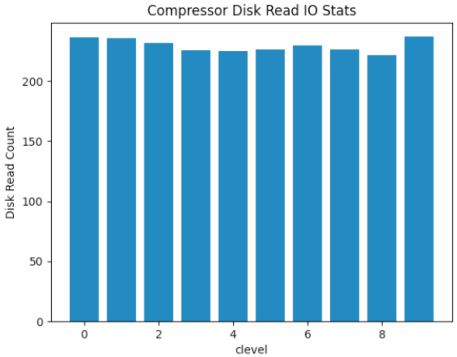
Impact of Compression Level on Disk Read Count



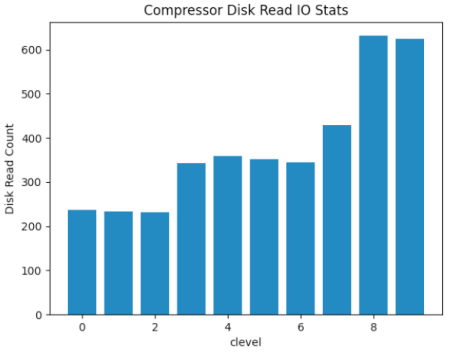
(a) blosclz



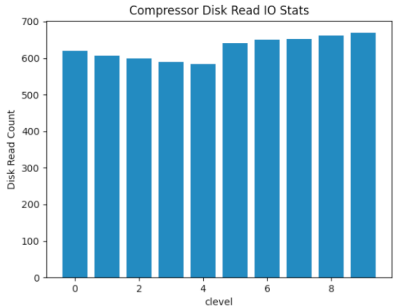
(b) lz4



(c) lz4hc



(d) zlib



(e) zstd

Chunking Strategies in Zarr

```
import zarr
import numpy as np
a = np.arange(100).reshape(10,10)
z = zarr.array(a, chunks=(5, 5))
z.info
```

Type	zarr.core.Array
Data type	int64
Shape	(10, 10)
Chunk shape	(5, 5)
Order	C
Read-only	False
Compressor	Blosc(cname='lz4', clevel=5, shuffle=SHUFFLE, blocksize=0)
Store type	zarr.storage.KVStore
No. bytes	800
No. bytes stored	593
Storage ratio	1.3
Chunks initialized	4/4

```
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)
```

Advanced Indexing in Zarr

```
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]]])

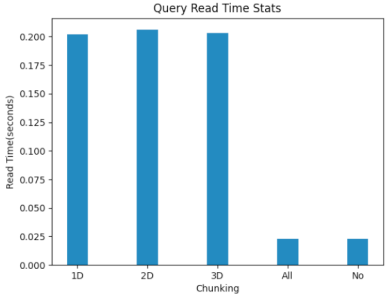
```
test_zarr.get_coordinate_selection(([0, 2], [1, 3], [3, 3]))
#array([ 7, 47])
```

```
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]])
```

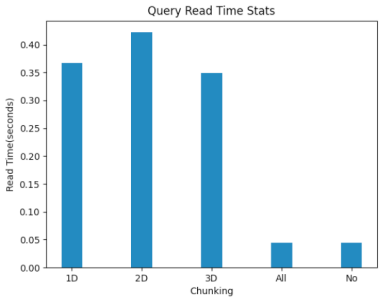
```
z = zarr.array(np.arange(15).reshape(3, 5))
z[:]
#array([[ 0,  1,  2,  3,  4],
#    [ 5,  6,  7,  8,  9],
#    [10, 11, 12, 13, 14]])
z.get_orthogonal_selection(([0, 2], slice(None)))  # select first and third rows
#array([[ 0,  1,  2,  3,  4],
 #       [10, 11, 12, 13, 14]])
z.get_orthogonal_selection((slice(None),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

Impact of Chunking Strategies on Advanced Indexing

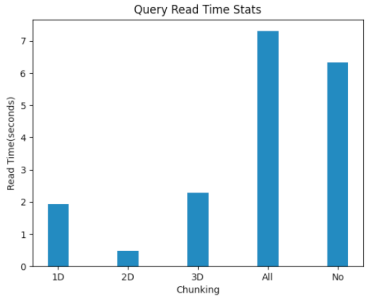


(a) Coordinate Selection



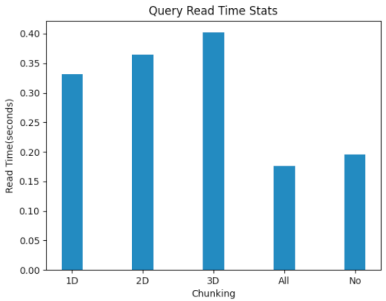
(b) vindex

(c) Orthogonal Selection 1

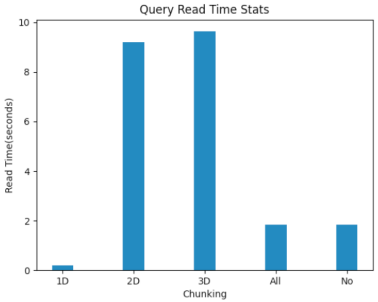


(d) Orthogonal Selection 2

(e) Orthogonal Selection 3

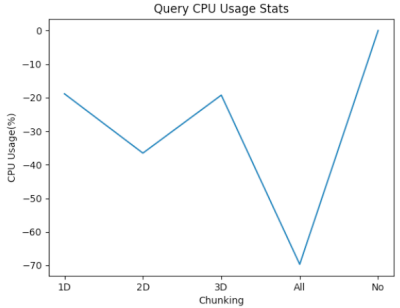


(f) Orthogonal Selection 4

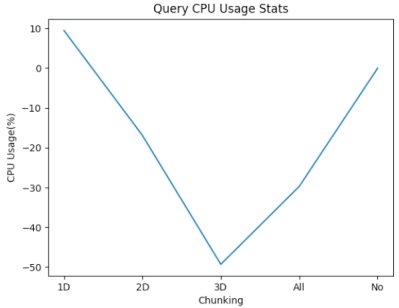


(g) Block Selection

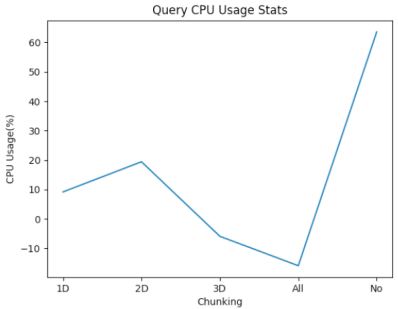
Impact of Chunking Strategies on CPU Usage



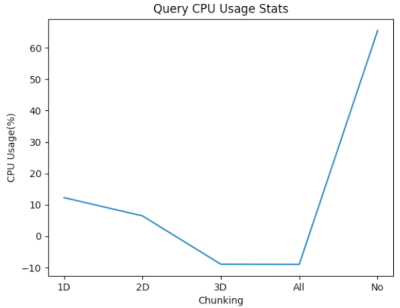
(a) Coordinate Selection



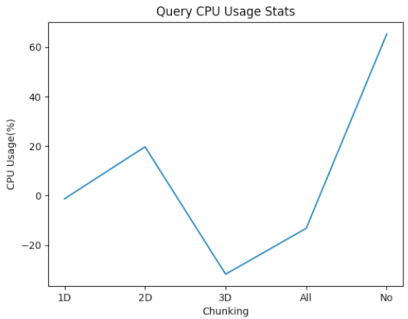
(b) vindex



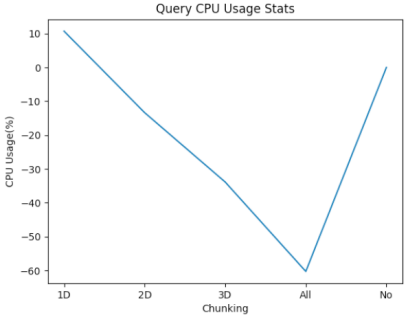
(c) Orthogonal Selection 1



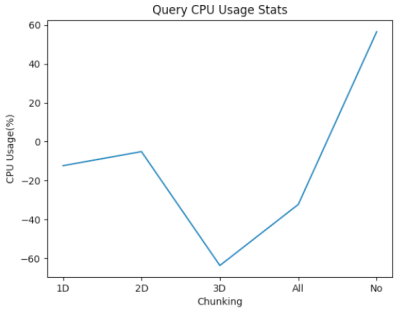
(d) Orthogonal Selection 2



(e) Orthogonal Selection 3



(f) Orthogonal Selection 4



(g) Block Selection

Conclusion

- It is imperative from the experiments that compression configuration parameters has some amount of impact on CPU Usage, disk reads and RAM usage. Mostly Read Time values were similar with the chosen configurations.
- Different chunking strategies and native access functions in Zarr library has different impact on Read Time, CPU and RAM usage and disk reads.
- This Research Project is done on 3-dimensional data. Further extension of this work can be done on more dimensions and evaluation can be done. Also compression ratio can be added as a parameter.

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.
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.

Thank You!

Vorname.Nachname@tu-ilmenau.de | www.tu-ilmenau.de

Bildnachweis: Folie 1: Chris Liebold, Folie 2,5,6,9: Michael Reichel, Folie 10: helibild