HDF5 Advanced Topics

Elena Pourmal
The HDF Group

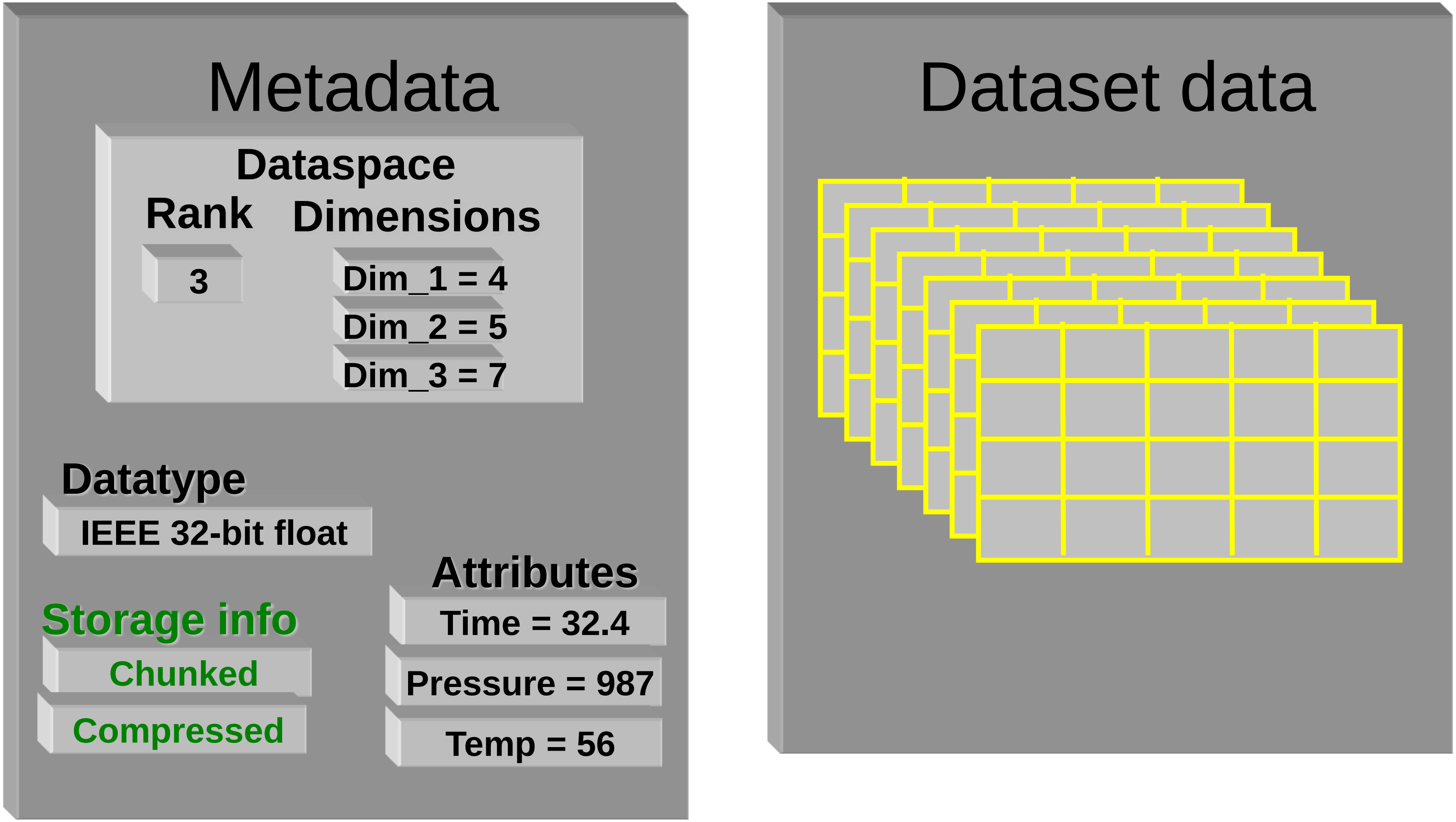
The 13th HDF and HDF-EOS Workshop
November 3-5, 2009

Chunking in HDF5

Goal

- To help you with understanding of how HDF5 chunking works, so you can efficiently store and retrieve data from HDF5

Recall from Intro: HDF5 Dataset



Metadata

Dataspace

Rank: 3

Dimensions:
Dim_1 = 4
Dim_2 = 5
Dim_3 = 7

Datatype: IEEE 32-bit float

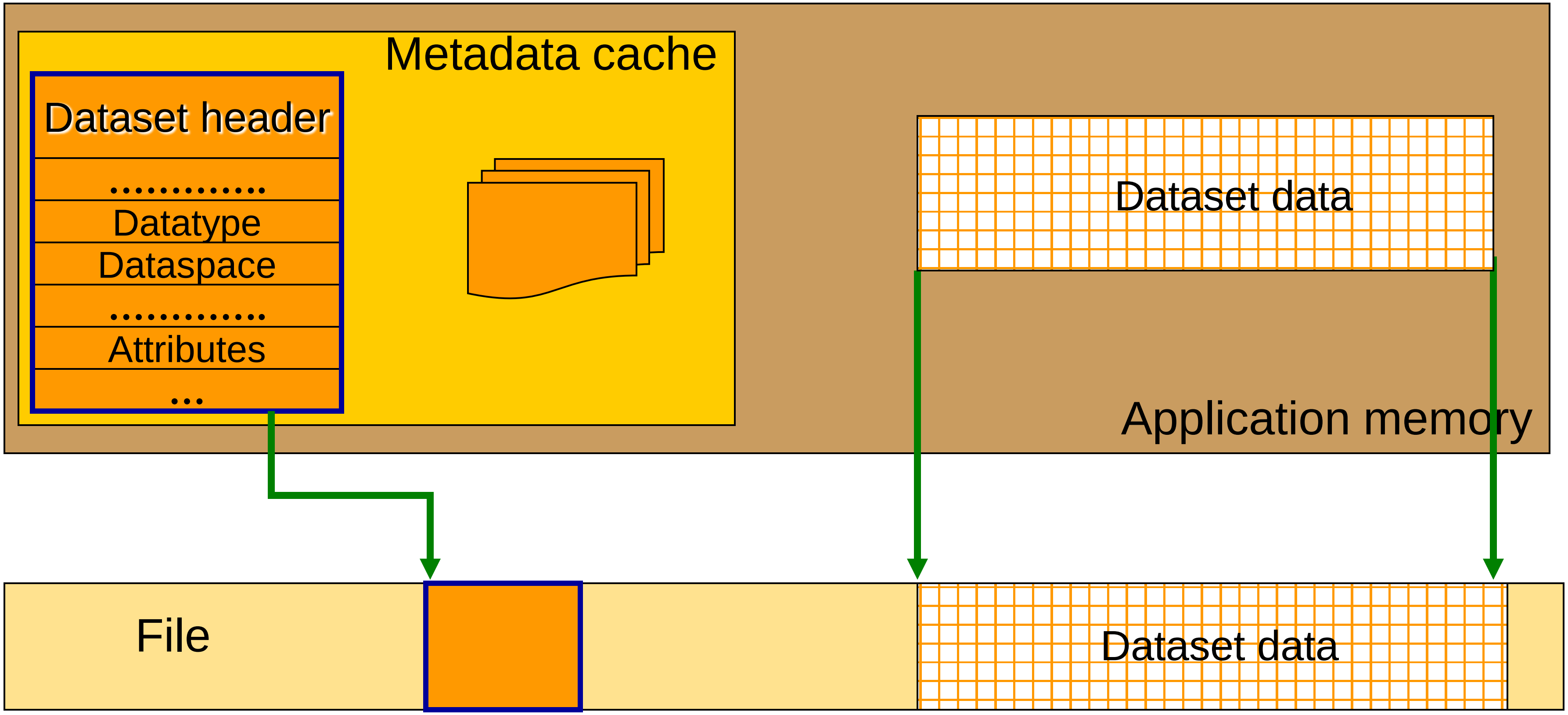
Storage info:
Chunked
Compressed

Attributes:
Time = 32.4
Pressure = 987
Temp = 56

Dataset data

Contiguous storage layout

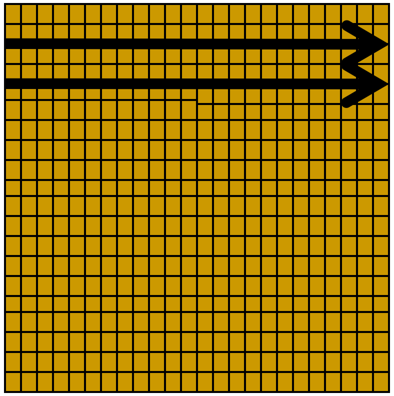
- Metadata header separate from dataset data
- Data stored in one contiguous block in HDF5



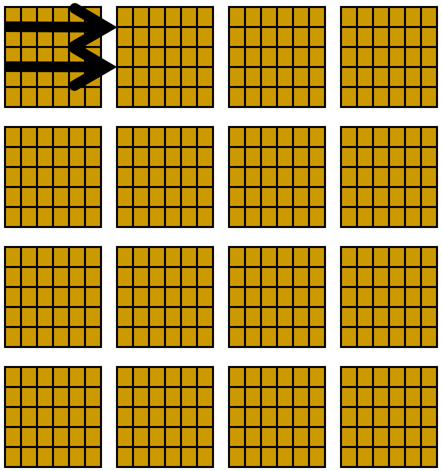
What is HDF5 Chunking?

- Data is stored in chunks of predefined size
- Two-dimensional instance may be referred to as data tiling
- HDF5 library always writes/reads the whole chunk

Contiguous

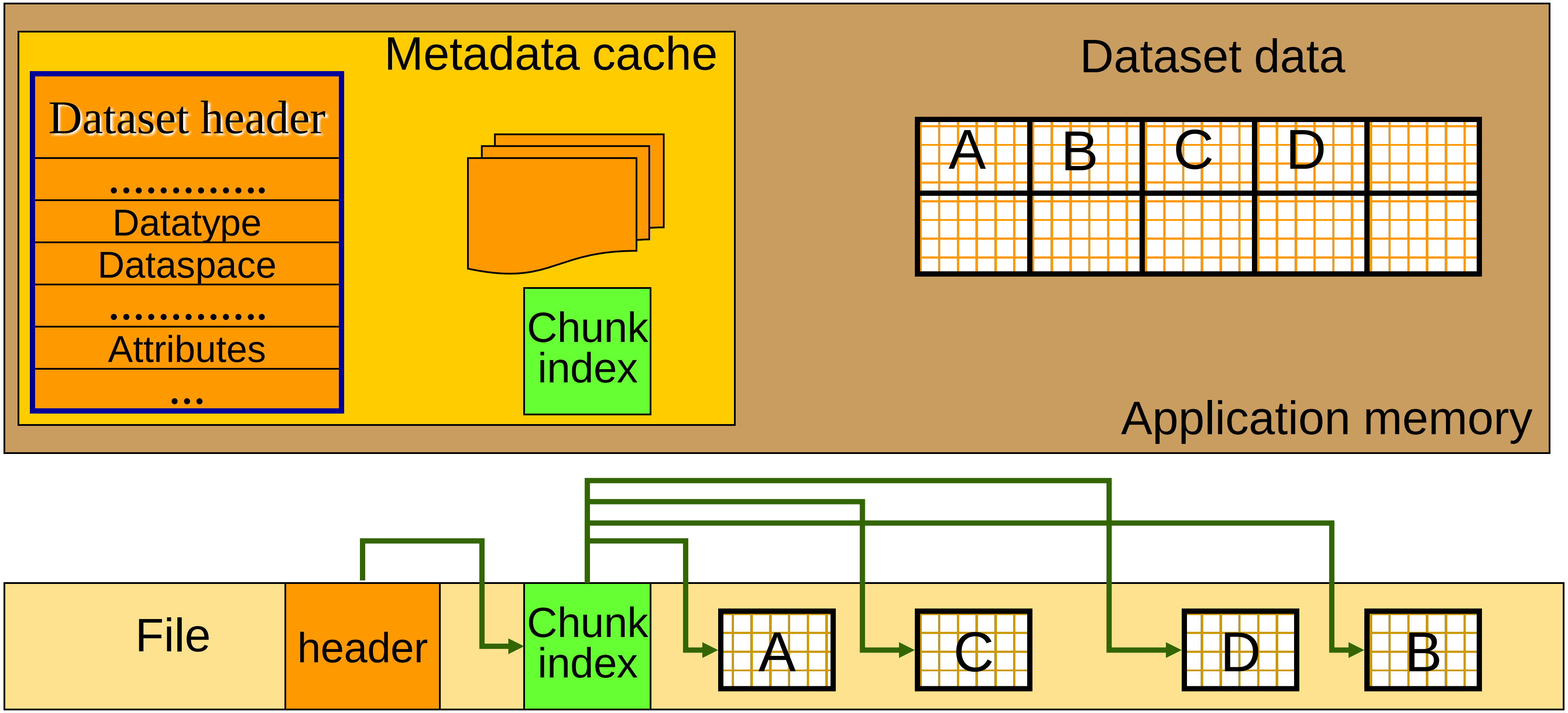


Chunked



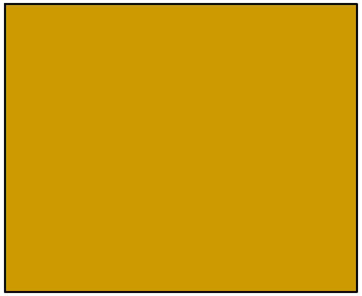
What is HDF5 Chunking?

- Dataset data is divided into equally sized blocks (chunks).
- Each chunk is stored separately as a contiguous block in HDF5 file.



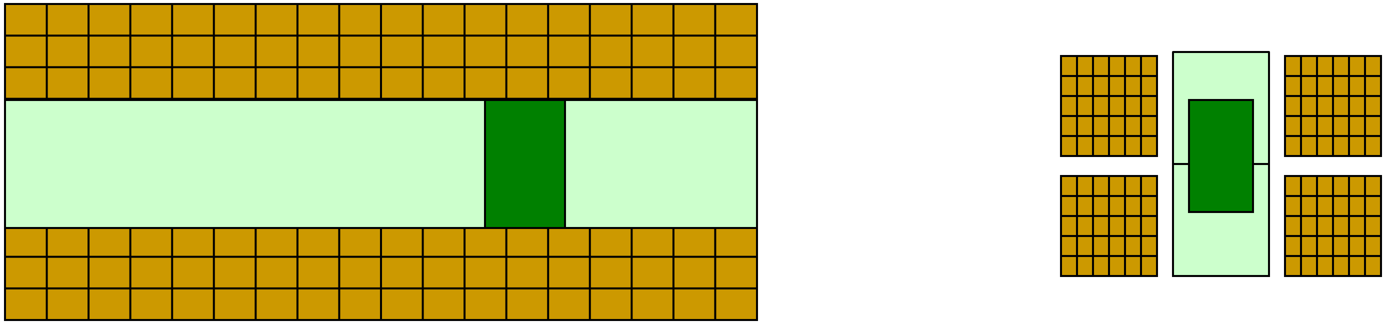
Why HDF5 Chunking?

- Chunking is required for several HDF5 features
  - Enabling compression and other filters like checksum
  - Extendible datasets



Why HDF5 Chunking?

- If used appropriately chunking improves partial I/O for big datasets



Only two chunks are involved in I/O

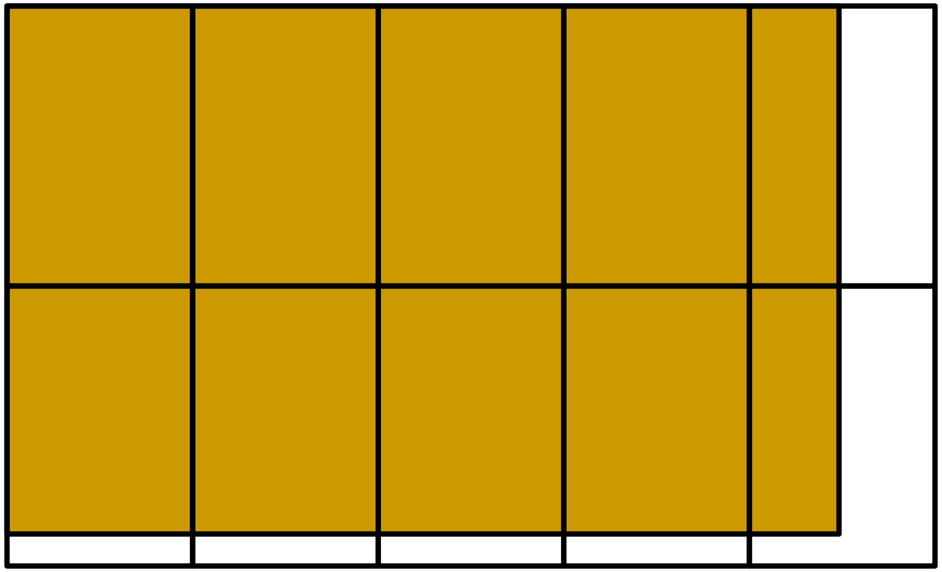
Creating Chunked Dataset

1. Create a dataset creation property list.
2. Set property list to use chunked storage layout.
3. Create dataset with the above property list.

```
dcpl_id   = H5Pcreate(H5P_DATASET_CREATE);
rank = 2;
ch_dims[0] = 100;
ch_dims[1] = 200;
H5Pset_chunk(dcpl_id, rank, ch_dims);
dset_id = H5Dcreate (…, dcpl_id);
H5Pclose(dcpl_id);
```

Creating Chunked Dataset

- Things to remember:
  - Chunk always has the same rank as a dataset
  - Chunk's dimensions do not need to be factors of dataset's dimensions
  - *Caution: May cause* ***more*** *I/O than desired (see white portions of the chunks below)*



Quiz time

- Why shouldn’t I make a chunk with dimension sizes equal to one?
- Can I change chunk size after dataset was created?

Writing or Reading Chunked Dataset

1. Chunking mechanism is transparent to application.
2. Use the same set of operation as for contiguous dataset, for example,

```
H5Dopen(…);
H5Sselect_hyperslab (…);
H5Dread(…);
```

3. Selections do not need to coincide precisely with the chunks boundaries.

HDF5 Chunking and compression

- Chunking is required for compression and other filters
- HDF5 filters modify data during I/O operations
- Filters provided by HDF5:
  - Checksum (H5Pset_fletcher32)
  - Data transformation (in 1.8.*)
  - Shuffling filter (H5Pset_shuffle)
- Compression (also called filters) in HDF5
  - Scale + offset (in 1.8.*) (H5Pset_scaleoffset)
  - N-bit (in 1.8.*) (H5Pset_nbit)
  - GZIP (deflate) (H5Pset_deflate)
  - SZIP (H5Pset_szip)

HDF5 Third-Party Filters

- Compression methods supported by HDF5 User’s community

http://wiki.hdfgroup.org/Community-Support-for-HDF5

  - LZO lossless compression (PyTables)
  - BZIP2 lossless compression (PyTables)
  - BLOSC lossless compression (PyTables)
  - LZF lossless compression H5Py

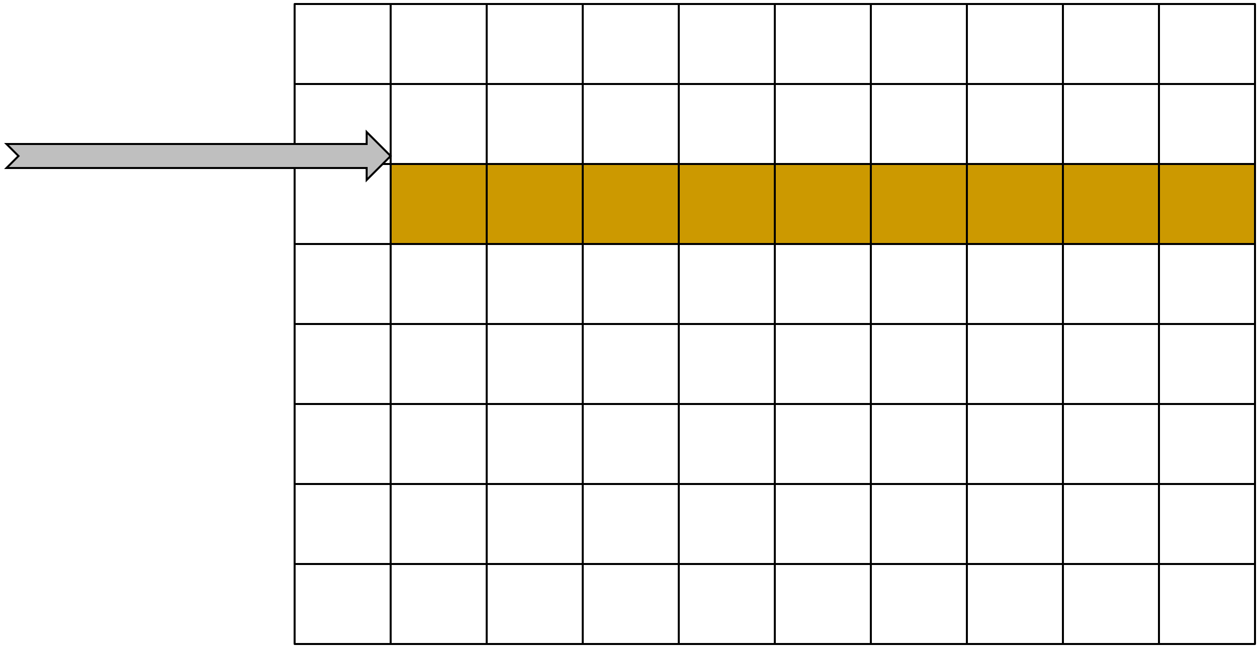
Creating Compressed Dataset

1. Create a dataset creation property list
2. Set property list to use chunked storage layout
3. Set property list to use filters
4. Create dataset with the above property list

```
 crp_id   = H5Pcreate(H5P_DATASET_CREATE);
rank = 2;
ch_dims[0] = 100;
ch_dims[1] = 100;
H5Pset_chunk(crp_id, rank, ch_dims);
H5Pset_deflate(crp_id, 9);
dset_id = H5Dcreate (…, crp_id);
H5Pclose(crp_id);
```

Performance Issues
or
What everyone needs to know about chunking, compression and chunk cache

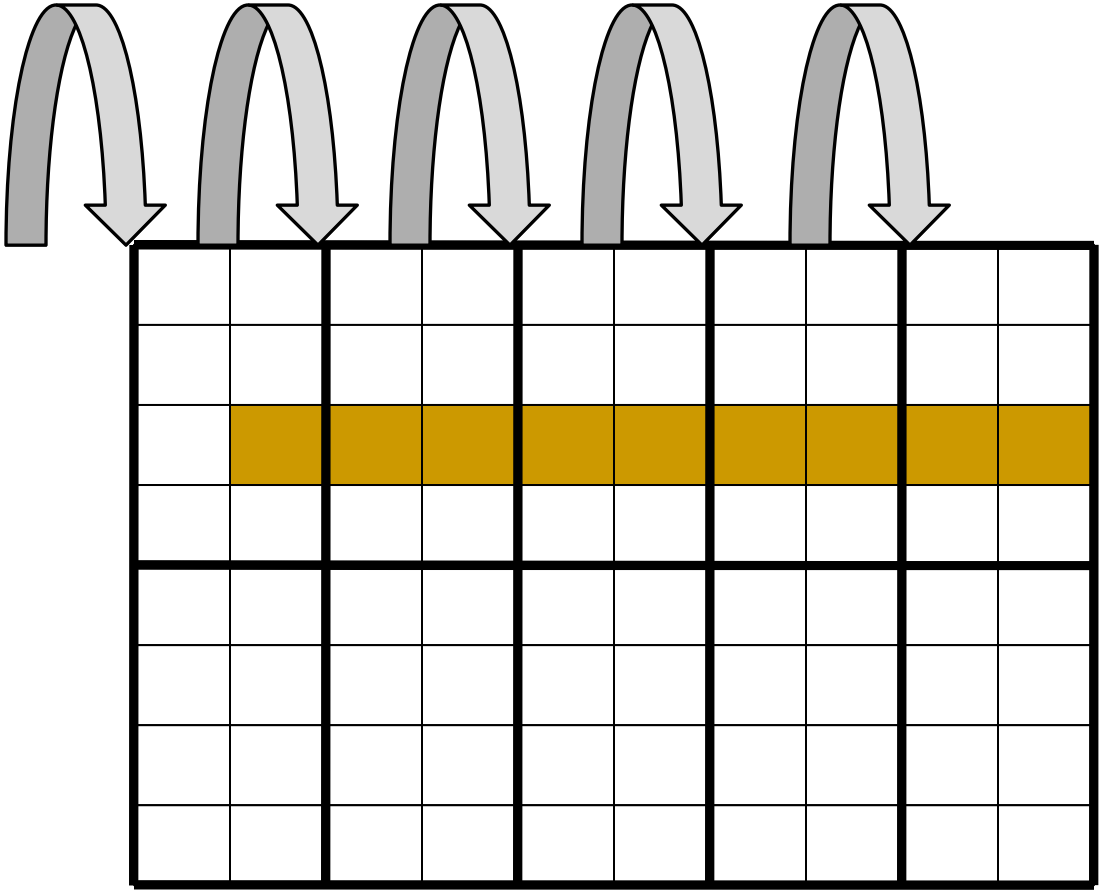
Accessing a row in contiguous dataset



One seek is needed to find the starting location of row of data.

Data is read/written using one disk access.

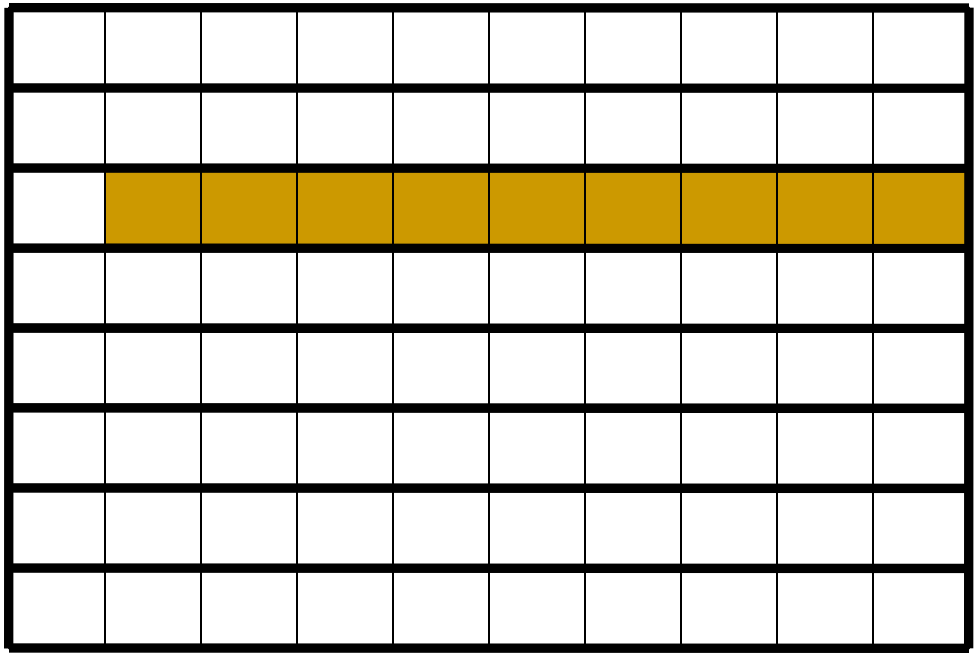
Accessing a row in chunked dataset



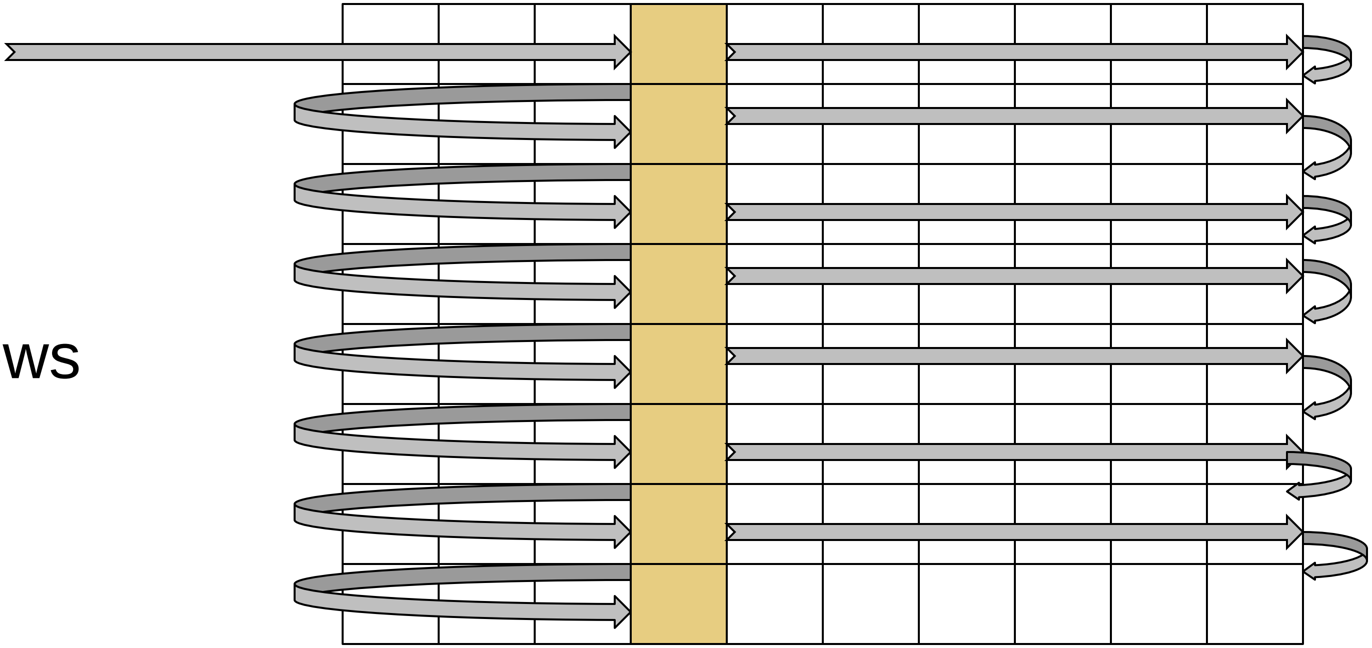
Five seeks is needed to find each chunk. Data is read/written using five disk accesses. Chunking storage is less efficient than contiguous storage.

Quiz time

- How might I improve this situation, if it is common to access my data in this way?



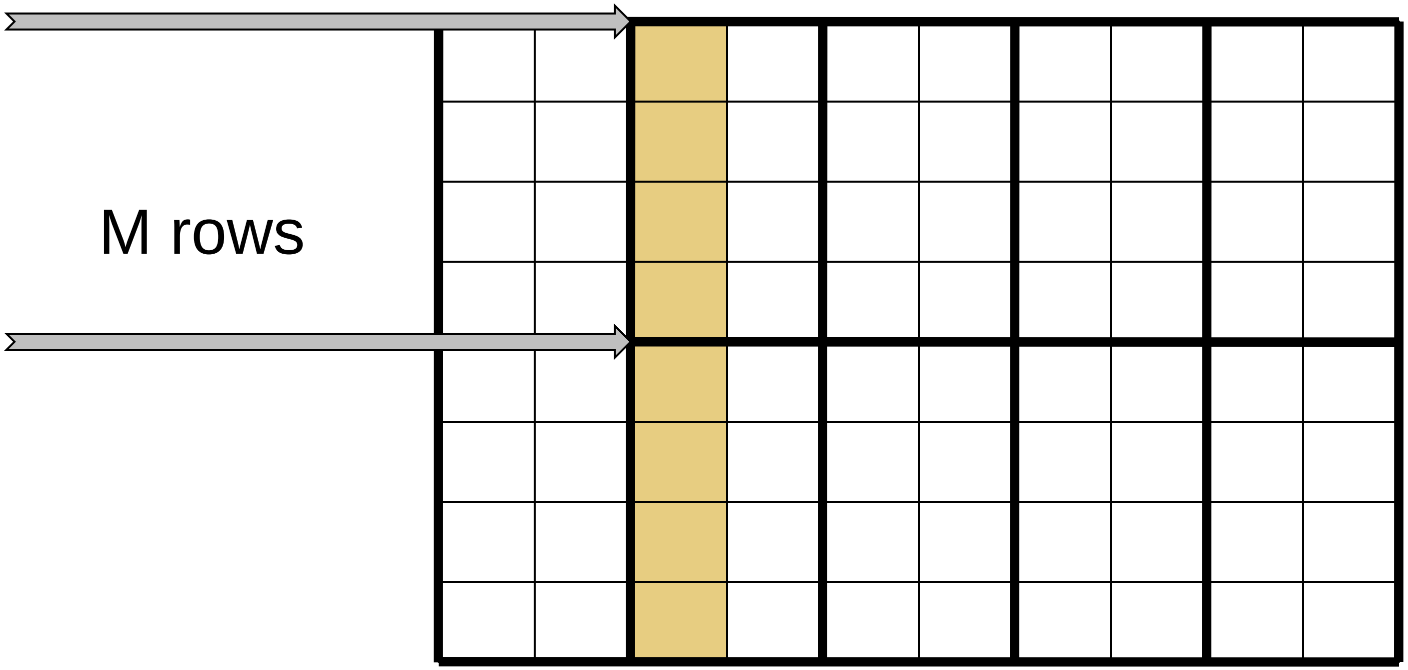
Accessing data in contiguous dataset



M rows

M seeks are needed to find the starting location of the element. Data is read/written using M disk accesses. Performance may be very bad.

Motivation for chunking storage

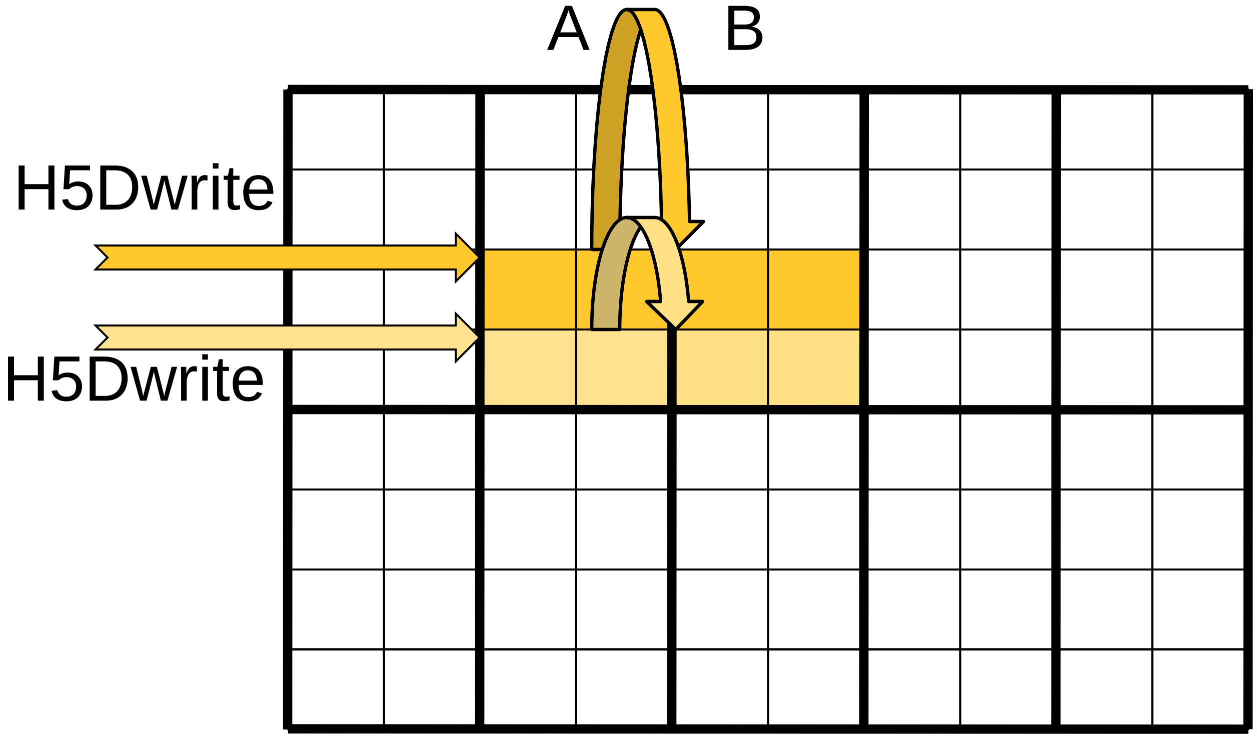


Two seeks are needed to find two chunks. Data is read/written using two disk accesses. For this pattern chunking helps with I/O performance.

Quiz time

- If I know I shall always access a column at a time, what size and shape should I make my chunks?

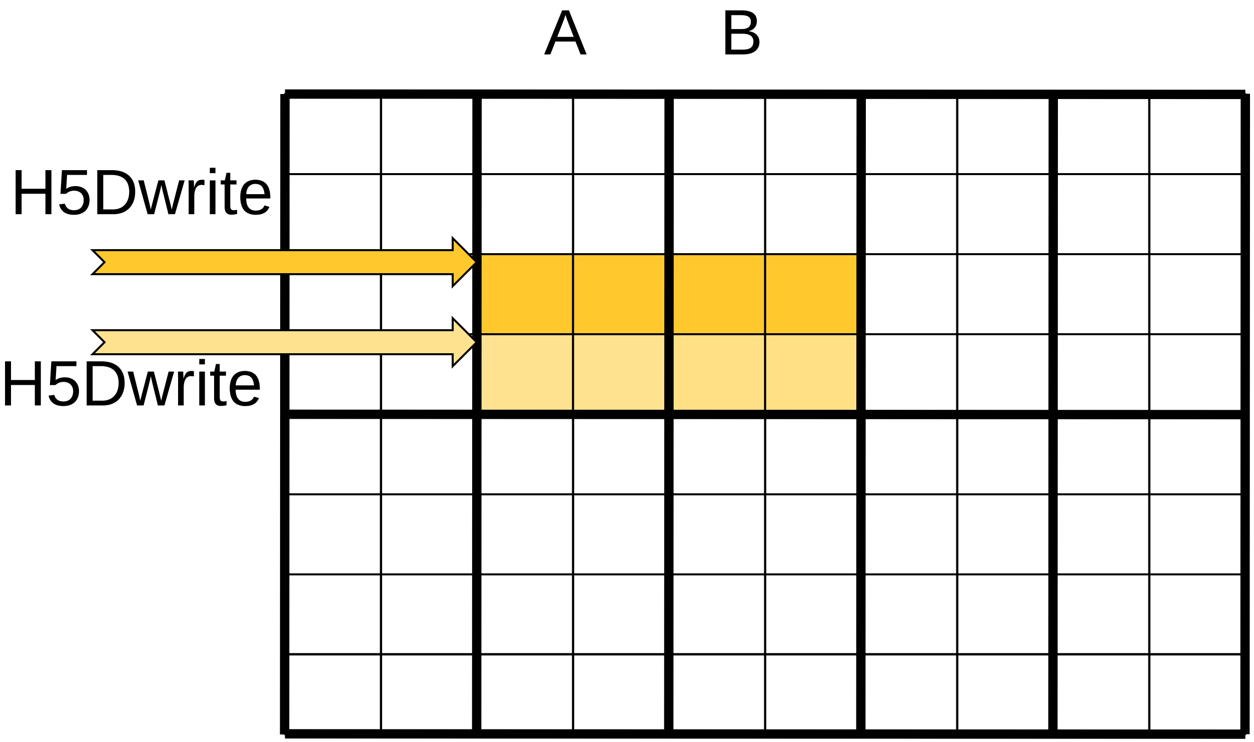
Motivation for chunk cache



Selection shown is written by two H5Dwrite calls (one for each row).

Chunks A and B are accessed twice (one time for each row). If both chunks fit into cache, only two I/O accesses needed to write the shown selections.

Motivation for chunk cache



Question: What happens if there is a space for only one chunk at a time?

HDF5 raw data chunk cache

- Improves performance whenever the same chunks are read or written multiple times.
- Current implementation doesn’t adjust parameters automatically (cache size, size of hash table).
- Chunks are indexed with a simple hash table.
- Hash function = (*cindex* mod *nslots*), where *cindex* is the linear index into a hypothetical array of chunks and *nslots* is the size of hash table.
- Only one of several chunks with the same hash value stays in cache.
- *Nslots* should be a prime number to minimize the number of hash value collisions*.*

HDF5 Chunk Cache APIs

- **H5Pset_chunk_cache** sets raw data chunk cache parameters for a dataset

```
H5Pset_chunk_cache (dapl, rdcc_nslots,
                    rdcc_nbytes, rdcc_w0);
```

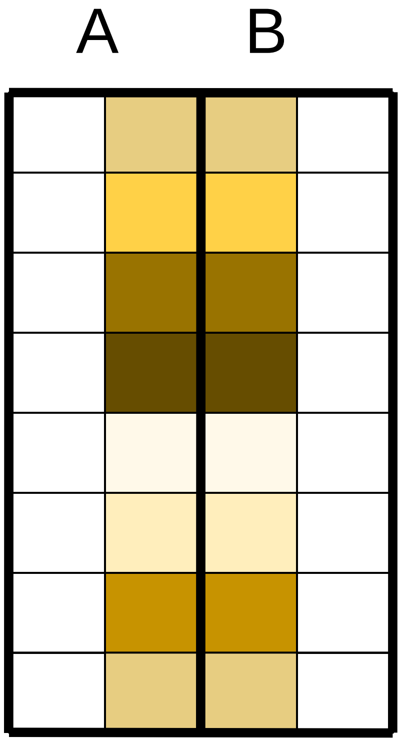
- **H5Pset_cache** sets raw data chunk cache parameters for all datasets in a file

```
H5Pset_cache (fapl, 0, nslots,
              5*1024*1024, rdcc_w0);
```

Hints for Chunk Settings

- Chunk dimension sizes should align as closely as possible with hyperslab dimensions for read/write
- Chunk cache size (**`rdcc_nbytes`**) should be large enough to hold all the chunks in a selection
  - If this is not possible, it may be best to disable chunk caching altogether (set **`rdcc_nbytes`** to 0)
- **`rdcc_slots`** should be a prime number that is at least 10 to 100 times the number of chunks that can fit into **`rdcc_nbytes`**
- **`rdcc_w0`** should be set to 1 if chunks that have been fully read/written will never be read/written again

The Good and The Ugly: Reading a row



M rows
Each row is read by a separate call to H5Dread

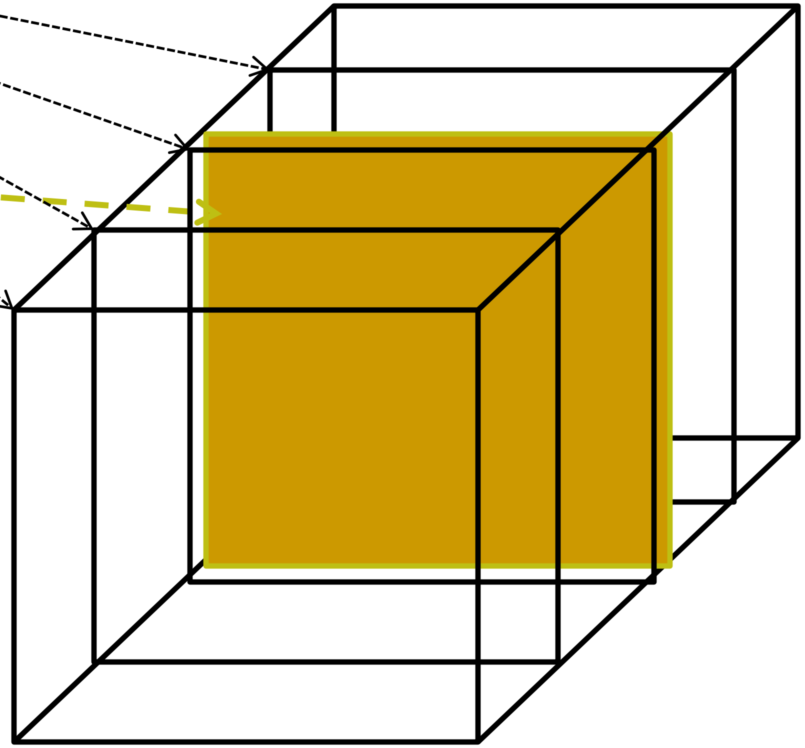
The Good: If both chunks fit into cache, 2 disks accesses are needed to read the data.
The Ugly: If one chunk fits into cache, 2M disks accesses are needed to read the data (compare with M accesses for contiguous storage).

Case study: Writing Chunked Dataset

- 1000x100x100 dataset
  - 4 byte integers
  - Random values 0-99
- 50x100x100 chunks (20 total)
  - Chunk size: 2 MB
- Write the entire dataset using 1x100x100 slices
  - Slices are written sequentially
- Chunk cache size 1MB (default) compared with chunk cache size is 5MB

Test Setup

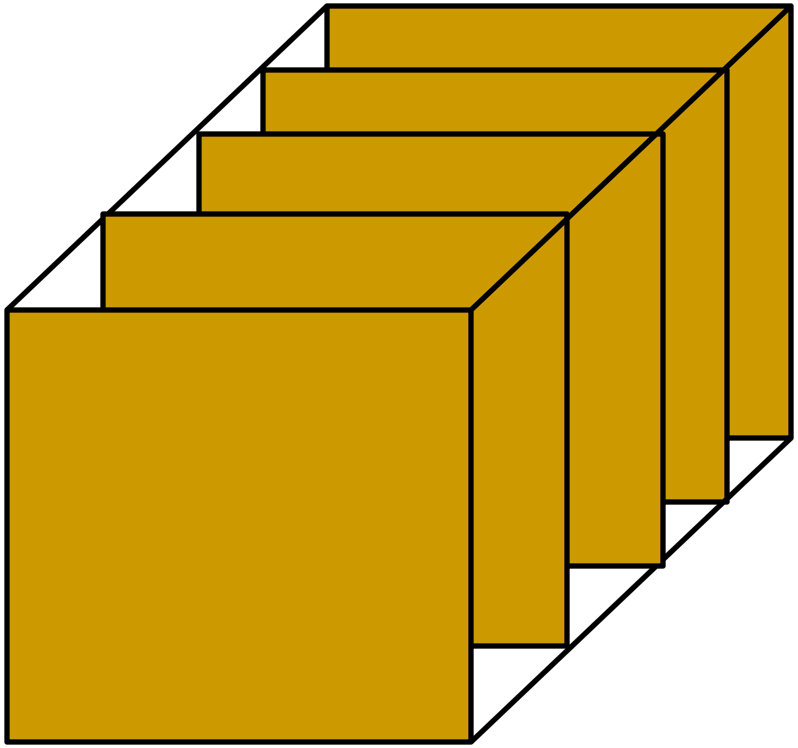
- 20 Chunks
- 1000 slices



- Chunk size ~ 2MB
- Total size ~ 40MB
- Each plane ~ 40KB

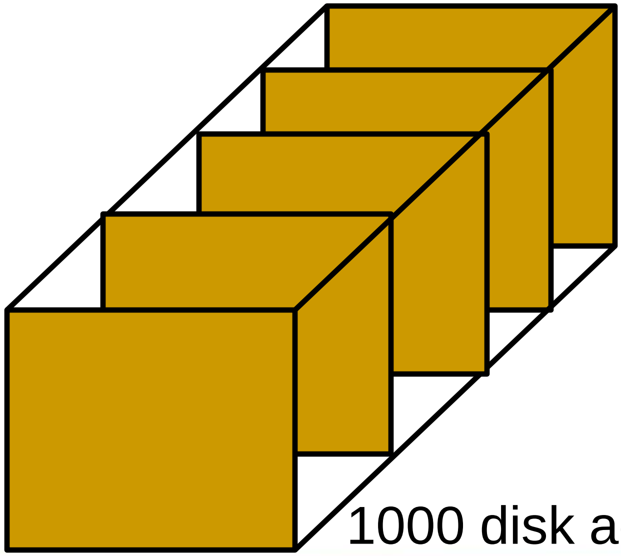
Aside: Writing dataset with contiguous storage

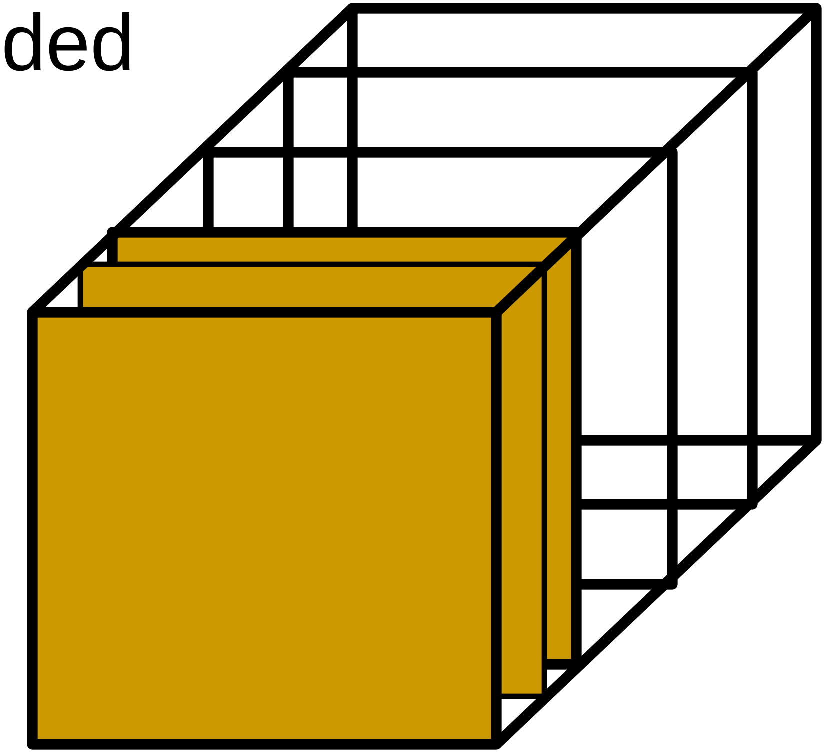
- 1000 disk accesses to write 1000 planes
- Total size written 40MB



Writing chunked dataset

- Example: Chunk fits into cache
- Chunk is filled in cache and then written to disk
- 20 disk accesses are needed
- Total size written 40MB

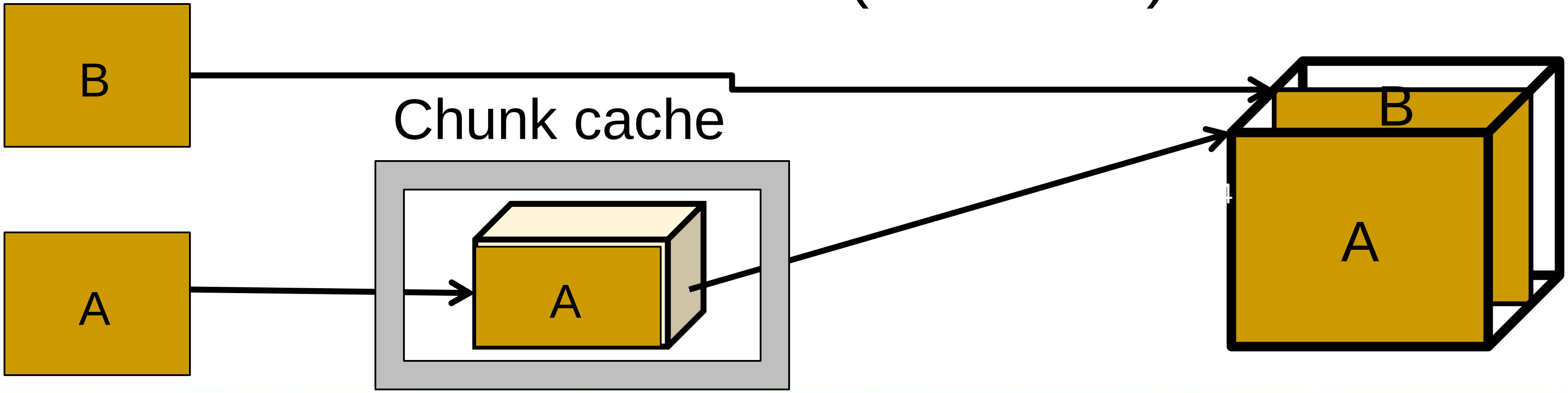




1000 disk access for contiguous

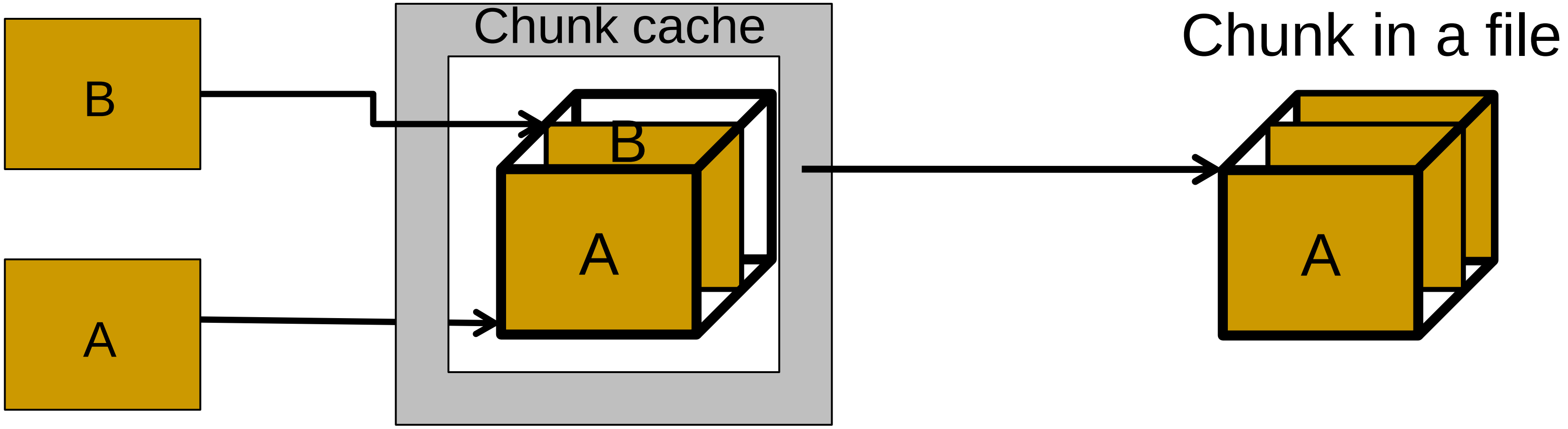
Writing chunked dataset

- Example: Chunk doesn't fit into cache
- *For each chunk (20 total)*
    1. *Fill chunk in memory with the first plane and write it to the file*
    2. *Write 49 new planes to file directly*
- *End For*
- Total disk accesses 20 x(1 + 49)= 1000
- Total data written ~80MB (vs. 40MB)



Writing compressed chunked dataset

- Example: Chunk fits into cache
- *For each chunk (20 total)*
    1. *Fill chunk in memory, compress it and write it to file*
- *End For*
- Total disk accesses 20
- Total data written less than 40MB



Writing compressed chunked dataset

- Example: Chunk doesn’t fit into cache
  - *For each chunk (20 total)*
    - *Fill chunk with the first plane, compress, write to a file*
    - *For each new plane (49 planes)*
      - *Read chunk back*
      - *Fill chunk with the plane*
      - *Compress*
      - *Write chunk to a file*
    - *End For*
  - *End For*
  - Total disk accesses 20 x(1+2x49)= 1980
  - Total data written and read ? (see next slide)
  - Note: HDF5 can probably detect such behavior and increase cache size

Effect of Chunk Cache Size on Write

No compression, chunk size is 2MB

Cache size	I/O operations	Total data written	File size
1 MB (default)	**1002**	**75.54 MB**	38.15 MB
5 MB	22	38.16 MB	38.15 MB

Gzip compression

Cache size	I/O operations	Total data written	File size
1 MB (default)	**1982**	**335.42 MB (322.34 MB read)**	13.08 MB
5 MB	22	**13.08 MB**	13.08 MB

Effect of Chunk Cache Size on Write

- With the 1 MB cache size, a chunk will not fit into the cache
  - All writes to the dataset must be immediately written to disk
  - With compression, the entire chunk must be read and rewritten every time a part of the chunk is written to
    - Data must also be decompressed and recompressed each time
    - Non sequential writes could result in a larger file
- Without compression, the entire chunk must be written when it is first written to the file
- If the selection were not contiguous on disk, it could require as much as 1 I/O disk access for each element

Effect of Chunk Cache Size on Write

- With the 5 MB cache size, the chunk is written only after it is full
  - Drastically reduces the number of I/O operations
  - Reduces the amount of data that must be written (and read)
  - Reduces processing time, especially with the compression filter

Conclusion

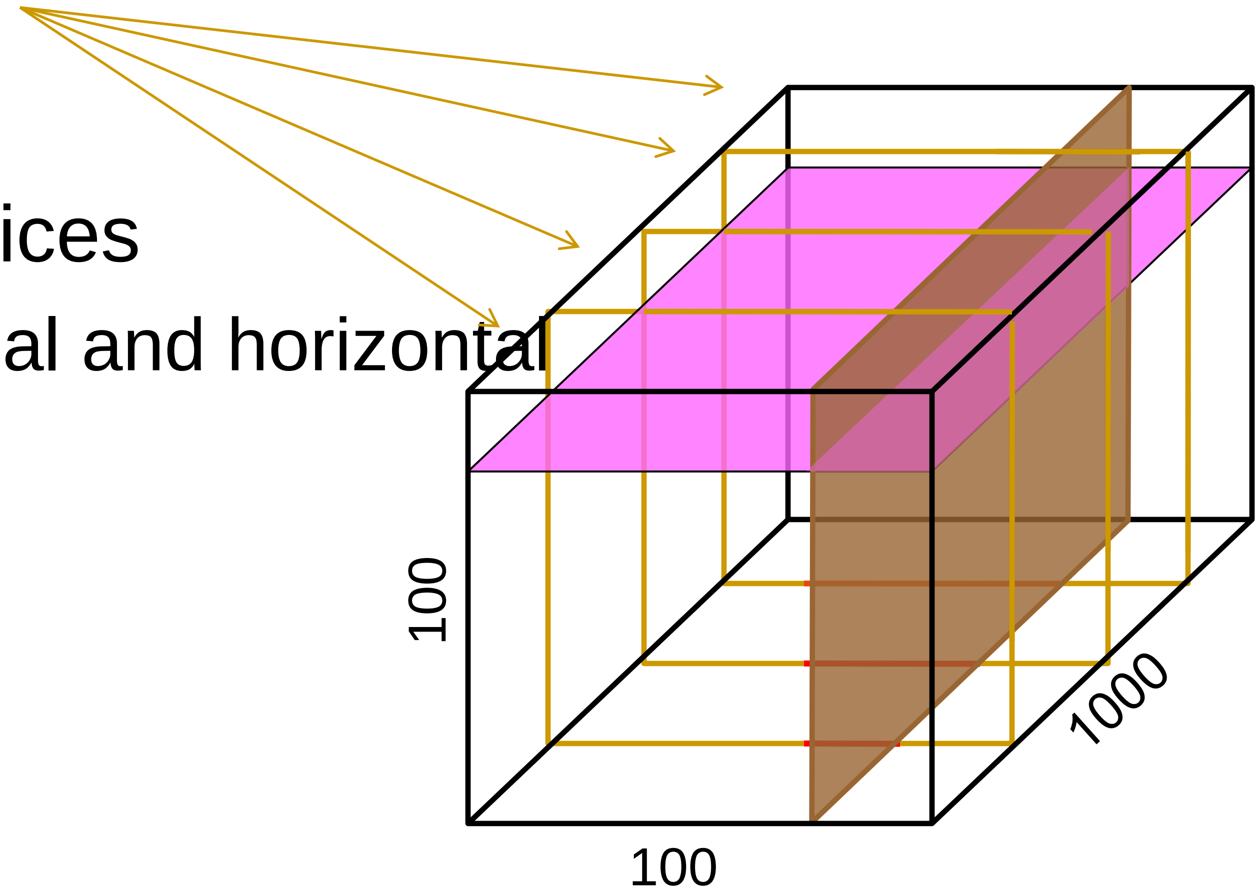
- It is important to make sure that a chunk will fit into the raw data chunk cache
- If you will be writing to multiple chunks at once, you should increase the cache size even more
- Try to design chunk dimensions to minimize the number you will be writing to at once

Reading Chunked Dataset

- Read the same dataset, again by slices, but the slices cross through all the chunks
- 2 orientations for read plane
  - Plane includes fastest changing dimension
  - Plane does not include fastest changing dimension
- Measure total read operations, and total size read
- Chunk sizes of 50x100x100, and 10x100x100
- 1 MB cache

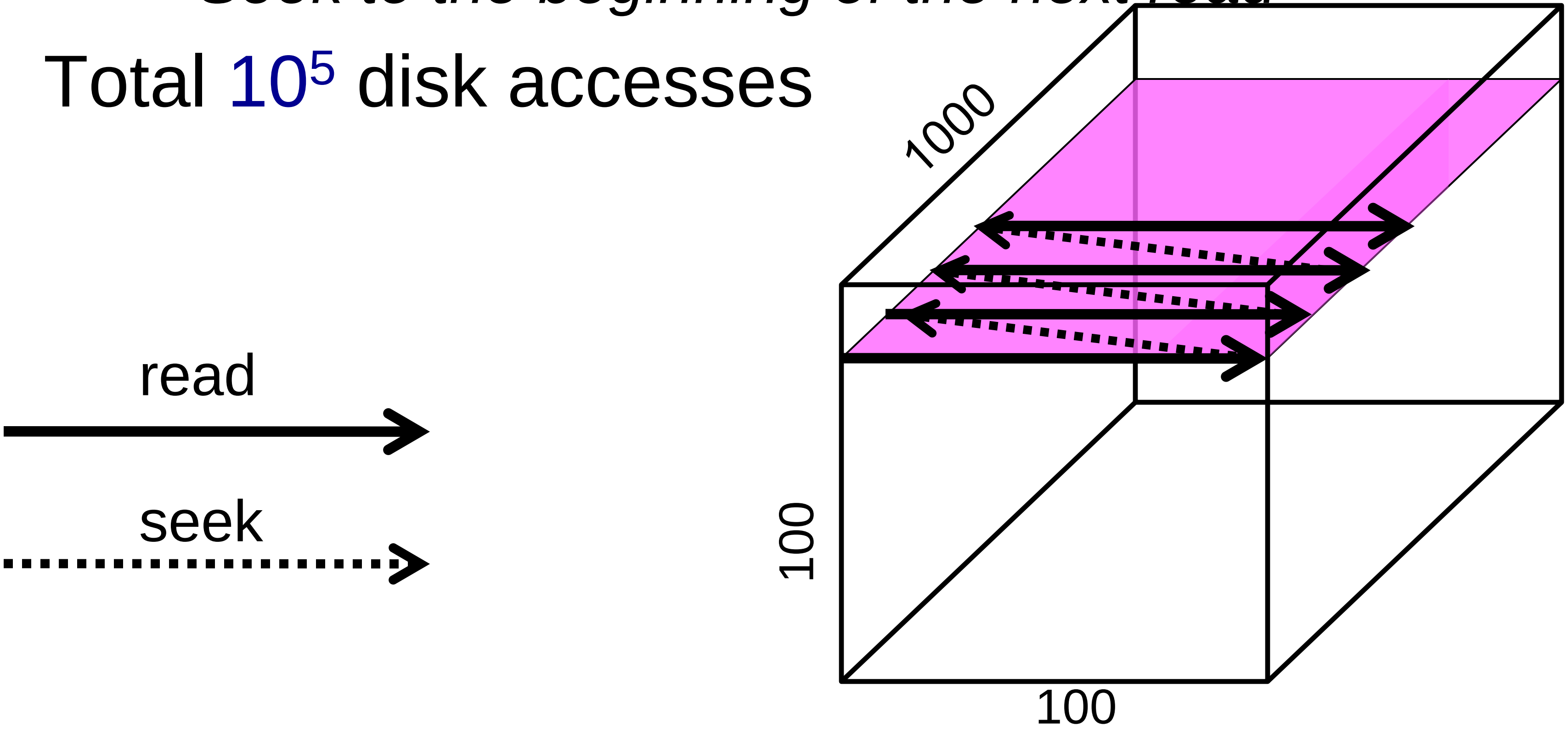
Test Setup

- Chunks
- Read slices
  - Vertical and horizontal



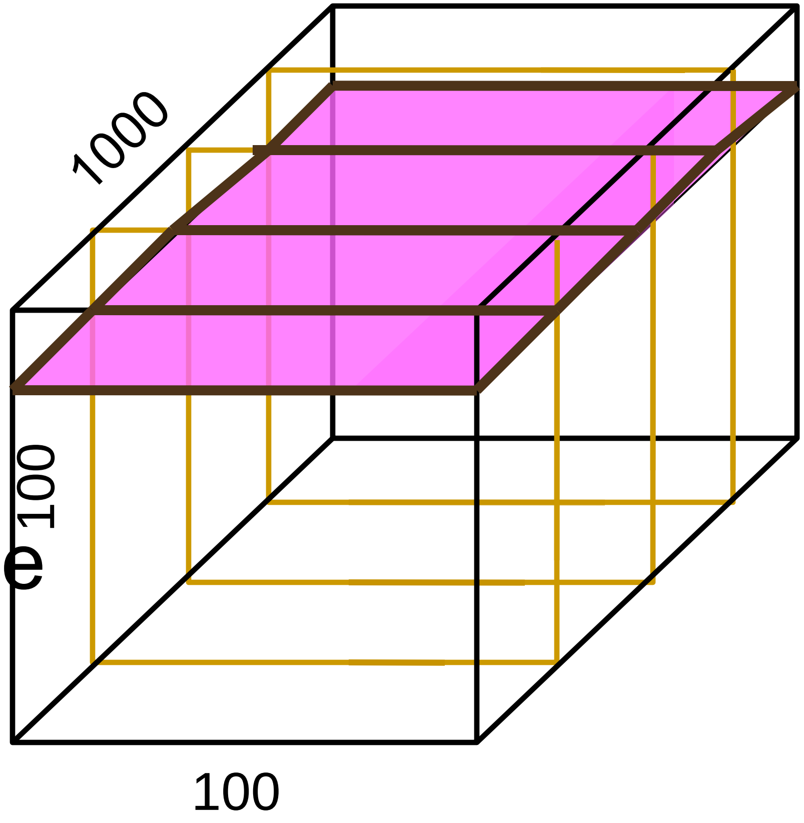
Aside: Reading from contiguous dataset

- *Repeat 100 times for each plane*
  - *Repeat 1000 times*
    - *Read a row*
    - *Seek to the beginning of the next read*
- Total 10^5 disk accesses



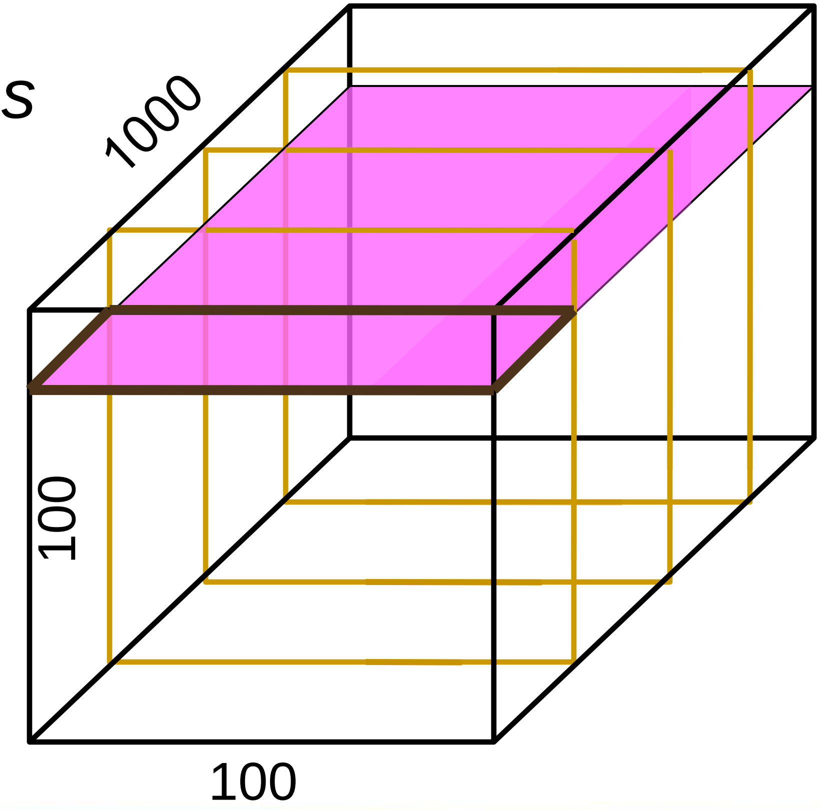
Reading chunked dataset

- No compression; chunk fits into cache
  - *For each plane (100 total)*
    - *For each chunk (20 total)*
      - *Read chunk*
      - *Extract 50 rows*
    - *End For*
  - *End For*
- Total 2000 disk accesses
- Chunk doesn’t fit into cache
  - Data is read directly from the file
  - 10^5 disk accesses



Reading chunked dataset

- Compression
- *Cache size doesn’t matter in this case*
- *For each plane (100 total)*
  - *For each chunk (20 total)*
    - *Read chunk, uncompress*
    - *Extract 50 rows*
  - *End*
- *End*
- Total 2000 disk accesses



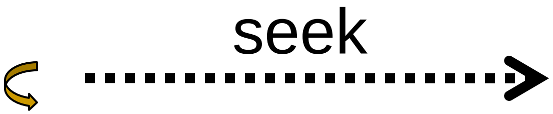
Results

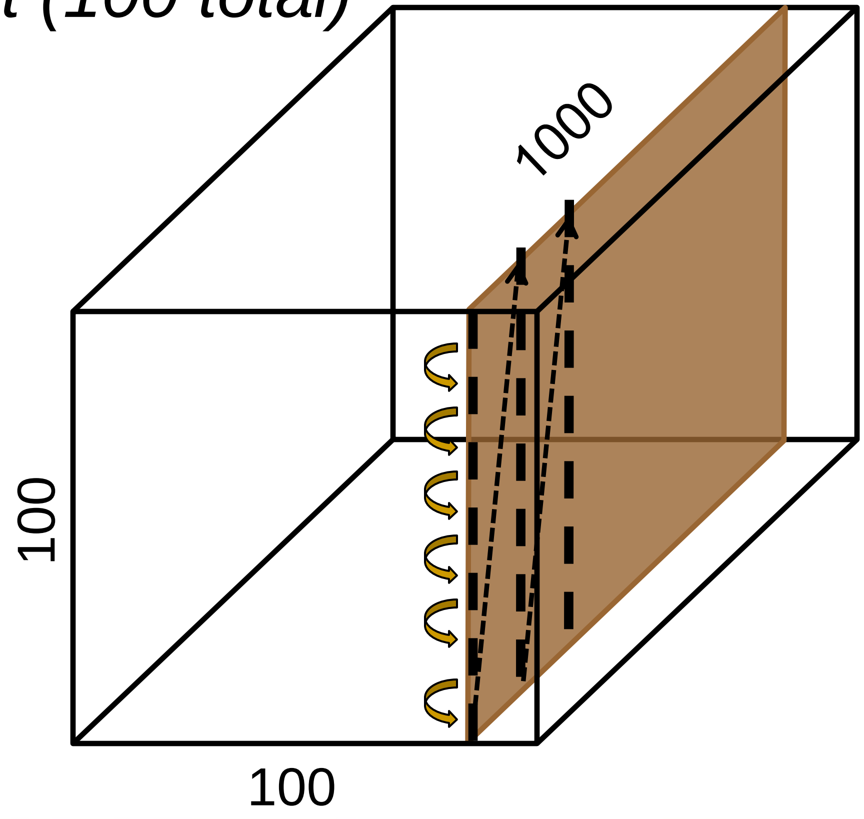
- Read slice includes fastest changing dimension

Chunk size	Compression	I/O operations	Total data read
50	Yes	2010	1307 MB
10	Yes	10012	1308 MB
50	No	100010	38 MB
10	No	10012	3814 MB

Aside: Reading from contiguous dataset

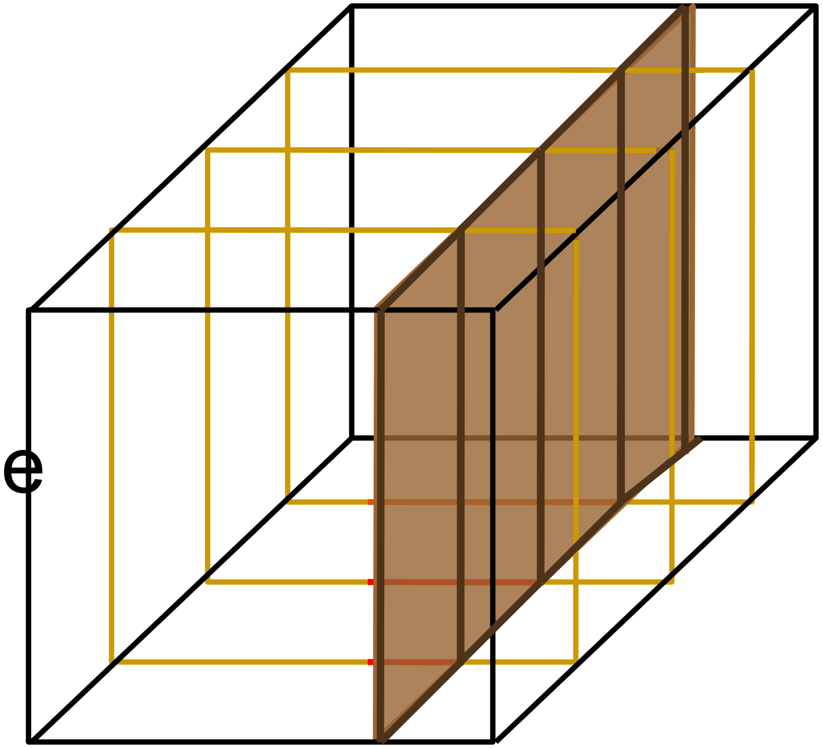
- *Repeat for each plane (100 total)*
  - *Repeat for each column (1000 total)*
    - *Repeat for each element (100 total)*
      - *Read element*
      - *Seek to the next one*
- Total 10^7 disk accesses





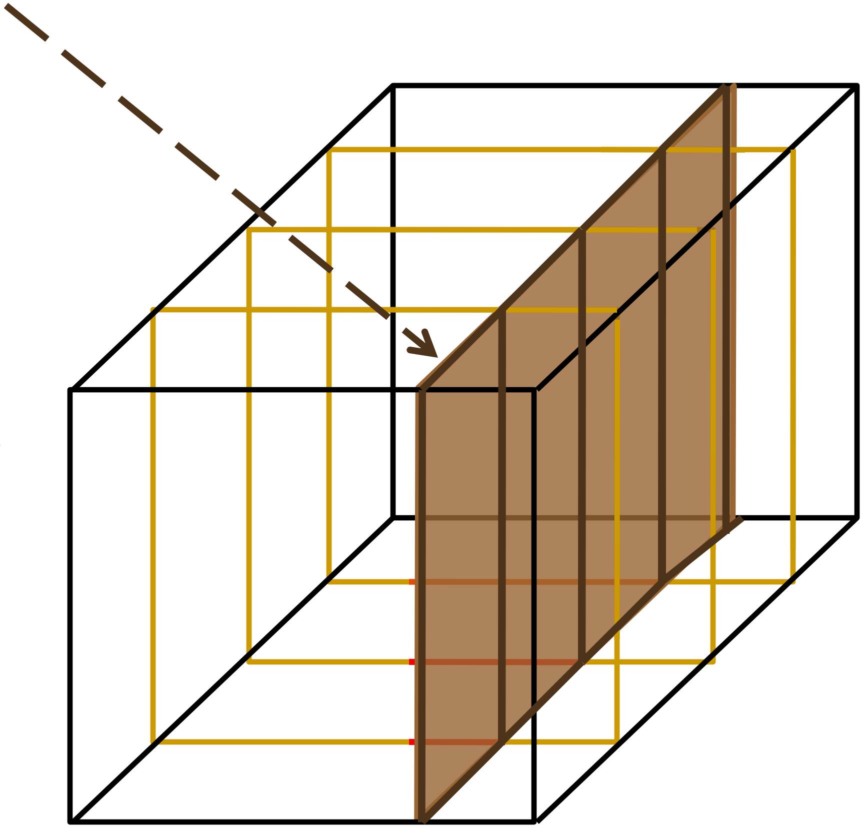
Reading chunked dataset

- No compression; chunk fits into cache
  - *For each plane (100 total)*
    - *For each chunk (20 total)*
      - *Read chunk, uncompress*
      - *Extract 50 columns*
    - *End*
  - *End*
- Total 2000 disk accesses
- Chunk doesn't fit into cache
    - Data is read directly from the file
    - 10^7 disk operations



Reading chunked dataset

- Compression; *cache size doesn't matter*
  - *For each plane (100 total)*
    - *For each chunk (20 total)*
      - *Read chunk, uncompress*
      - *Extract 50 columns*
    - *End*
  - *End*
- Total 2000 disk accesses



Results (continued)

- Read slice does not include fastest changing dimension

Chunk size	Compression	I/O operations	Total data read
50	Yes	**2010**	**1307 MB**
10	Yes	10012	1308 MB
50	No	**10000010**	**38 MB**
10	No	10012	3814 MB

Effect of Cache Size on Read

- When compression is enabled, the library must always read entire chunk once for each call to **`H5Dread`** (unless it is in cache)
- When compression is disabled, the library's behavior depends on the cache size relative to the chunk size.
    - If the chunk fits in cache, the library reads entire chunk once for each call to **`H5Dread`**
    - If the chunk does not fit in cache, the library reads only the data that is selected
        - More read operations, especially if the read plane does not include the fastest changing dimension
        - Less total data read

Conclusion

- On read cache size does not matter when compression is enabled.
- Without compression, the cache must be large enough to hold all of the chunks to get good preformance.
- The optimum cache size depends on the exact shape of the data, as well as the hardware, as well as access pattern.

Thank You!

Acknowledgements

This work was supported by cooperative agreement number NNX08AO77A from the National Aeronautics and Space Administration (NASA).

Any opinions, findings, conclusions, or recommendations expressed in this material are those of the author[s] and do not necessarily reflect the views of the National Aeronautics and Space Administration.

Questions/comments?