Making earth science data more accessible: experience with chunking

Russ Rew, Unidata

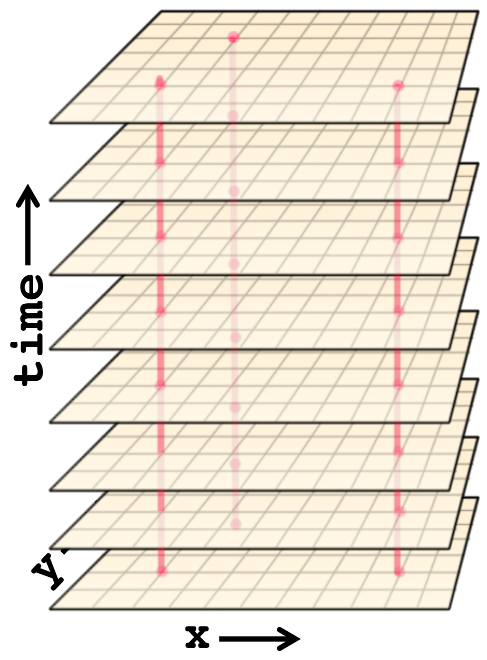
UCAR Software Engineering Assembly

What's the Problem?

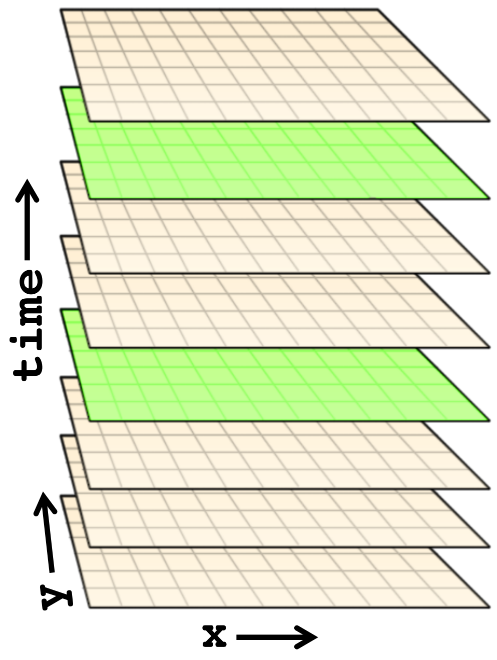
Can large multidimensional datasets be organized for fast *and* flexible access?

***Without multiple versions of the data*, what's the best you can do?**

Time range access



Spatial access



Conventional storage layout	Time varying fastest	Time varying slowest
Access a time series	Fast	Slow
Access a spatial slice	Slow	Fast

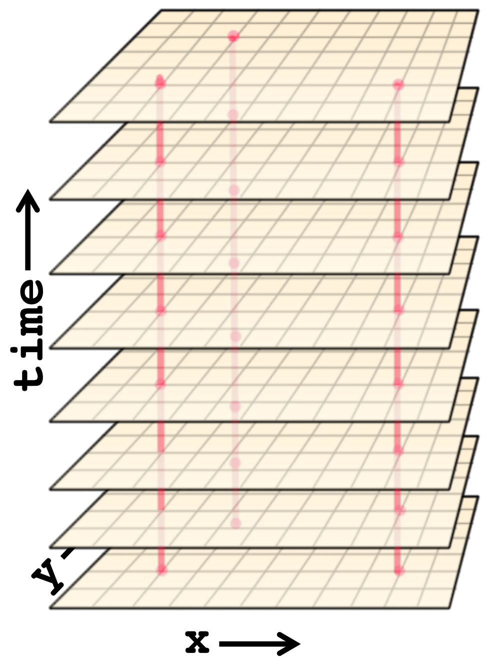
Goal is to solve what looks like a little problem that becomes more serious with …

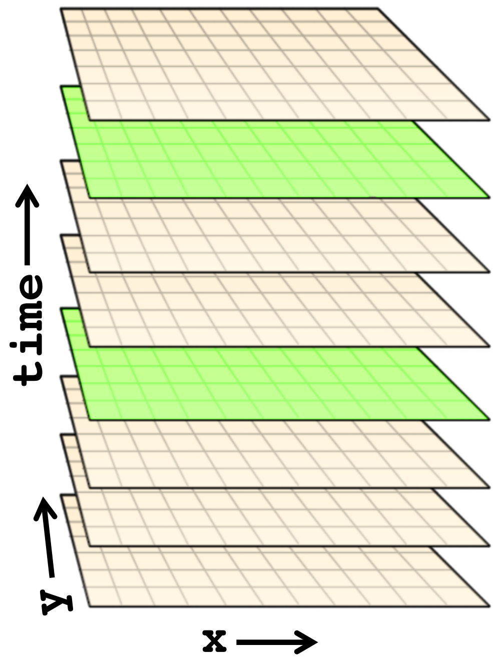
Big
Dat

Real data, conventional storage

`NCEP North American Regional Reanalysis`
`float 200mb_TMP(time=98128, y=277, x=349)`

9.5 billion values
38 GB of data
8.5 GB compressed





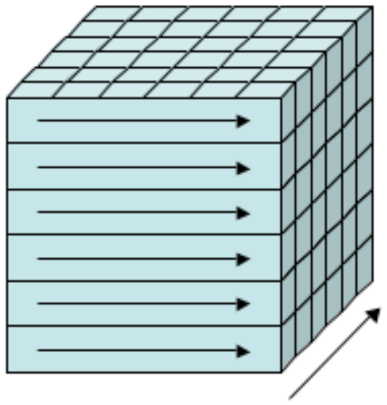
	Time varying fastest	Time varying slowest
Access a time series	0.013 sec	200 sec
Access a spatial slice	180 sec	0.012 sec

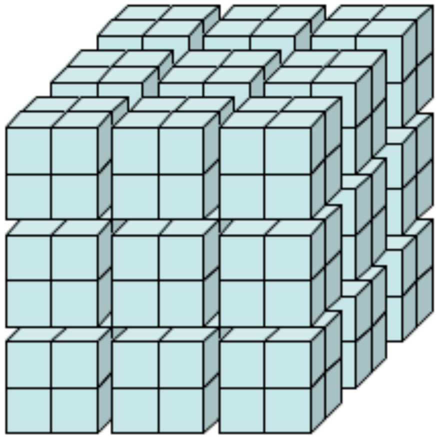
*Single file access time averaged over 100 independent queries, after clearing disk caches. 7200 rpm disk.

Why this example?

- Pretty Big Data: copying on a desktop platform takes minutes, not seconds (13.3 min disk, 4.0 min SSD)
- Multidimensional and gridded: increasingly typical of earth science data
- Three dimensions: usefully generalizable to higher dimensions
- Multiple plausible access patterns: typical of important datasets
- Big dimensions: large performance differences

What’s Chunking?


index order


chunked

Storing data in multidimensional "chunks" along each dimension to provide *balanced* access

Speeds up slow accesses

Slows down fast accesses

Benefits of chunking

- Performance gains for server-side subsetting
- Sparse data: empty chunks are not stored
- Efficient compression: only compress or uncompress chunks that are accessed
- Efficient appending: along multiple dimensions
- Efficient use of cache: for accessing adjacent slices
- Supports unanticipated access patterns

Obstacles to use of chunking

- Rechunking large datasets takes time
  - Either get it right when data created, or
  - Be willing to rechunk later, based on usage
- No optimal chunk sizes and shapes for arbitrary access patterns
- Software to rechunk big datasets is available, but defaults work poorly for some common cases
- Specific guidance for how to choose good chunk shapes for multiple access patterns is lacking

Importance of chunk shapes

Example: `float 200mb_TMP(time=98128, y=277, x=349)`

Storage layout, chunk shapes	Read time series (seconds)	Read horizontal slice (seconds)	Performance bias: (slowest / fastest)
Contiguous, for time series	0.013	180	14,000
Contiguous, for spatial slices	200	0.012	17,000
4MB chunks, 1032 x 29 x 35	3.3	3.3	1.0
1MB chunks, 516 x 20 x 25	3.1	3.2	1.0
8 KB chunks, 46 x 6 x 7	1.3 (*31)	1.2 (*3.2)	1.1 (*9.7)
4 KB chunks, 33 x 5 x 6	1.6 (*38)	1.4 (*3.3)	1.1 (*12)

Average for 256 independent reads.

* 1st read much slower, due to many small chunks?

Chunk shapes

- In 2-D, want chunks to be same shape as data domain to get same number of chunks in each direction of access
- 2-dimensional analog of chunking is too simple for common use case of 1- and (n-1)-D access in an n-dimensional dataset
- In 1-D and (n-1)-D access, need to divide chunks read per access equally between 1-D and (n-1)-D domains
- For 3-D use case example, balancing 1-D and 2-D accesses:
  - Let number of chunks along each dimension be n_{time} , n_y , n_x
  - Let N = total number of chunks = $n_{time}\ n_y\ n_x$
  - *time* by *y* by *x* chunk shape should be integral, near

    $n_{time} / N^{1/2}$ by $c\ n_y / N^{1/4}$ by $1/c\ n_x / N^{1/4}$ *(for any $c > 0$)*
- More detailed guidance in [Unidata's Developer's Blog]()

Computing chunk shapes

Definition: **chunk_shape** (varShape, valSize=4, chunkSize=4096)

Return a good chunk shape for an n-dimensional variable, assuming balanced 1D/(n-1)D access

varShape -- list of variable dimension sizes

chunkSize -- maximum chunksize desired, in bytes (default 4096)

valSize -- size of each data value, in bytes (default 4)

```
>>> chunk_shape( [98128, 277, 349], chunkSize = 2**22 )
[1032, 29, 35]

>>> chunk_shape( [98128, 277, 349], chunkSize = 8192 )
[46, 6, 7]
```

Chunking transparency

- Only creators of a dataset need to be concerned with chunk shapes and sizes
- Like compression, chunking can be specified per variable for netCDF-4 classic model data
- Chunking and compression are ***invisible*** when reading data, except for performance, because implemented in access libraries
- Rechunking and compression supported by **nccopy** or **h5repack** utilities
- Example: rechunk foo.nc to netCDF-4 classic model

```
nccopy –c time/46,y/6,x/7 contig.nc chunked.nc
```

Chunking and compression

- In using netCDF or HDF5 libraries, a chunk is an indivisible unit of disk access, compression, filters, and caching
- In general, larger chunks mean better compression
- Smaller chunks improve access times for compressed data, due to less computation for uncompression
- Including compression introduces caching issues

Chunk size

small chunks	large chunks
faster read access	slower read access
less compression	more compression
slower to create	faster to create
greater chunk overhead	less chunk overhead

- Chunk size should be at least the size of one disk block
- Common disk block sizes are 4KB, 1MB, or 4MB
- Chunk shape may be more important than chunk size for balanced and flexible access in multiple ways
- Many small chunks incur significant 1-time overhead on open
- To re-chunk large datasets, it helps to have lots of memory, SSD

How long does rechunking take?

Example: `float 200mb_TMP(time=98128, y=277, x=349)`

Destination chunks	nccopy: disk, SSD (minutes)	h5repack: disk, SSD (minutes)
4MB chunks, 1032 x 29 x 35	7, 4	99, 38
1MB chunks, 516 x 20 x 25	10, 10	134, 43
8 KB chunks, 46 x 6 x 7	11, 10	? , 46
4 KB chunks, 33 x 5 x 6	12, 14	? , 49

Justifying rechunking

- Rechunking benefits versus cost:
  - Ridiculously slow accesses become 100x faster: minutes to seconds
  - Very fast accesses become 100x slower: msec to seconds
  - 50% of each becomes 100x faster: minutes to seconds
- Consider zopfli zlib-compatible compression …
  - Takes 100x as long to compress as zlib
  - Compresses 5% better than zlib
  - Benefits worth cost for important data: smaller, faster, cheaper access from server

SSD and chunking

- Serial access with SSD can be 4 or 5 times faster than spinning disks
- SSD has much faster latency, typically 75 microsecs compared to 12 ms for a 7200 rpm disk, over 100 times faster
- Using SSD with contiguous layout can make chunking data unnecessary, because direct access is so fast
- However, SSD is still too expensive for servers with large data archives
- But hybrid drives may be a good use of SSD

Timings for SSD access

Example: `float 200mb_TMP(time=98128, y=277, x=349)`

Storage layout, chunk shapes	Read time series (seconds)	Read horizontal slice (seconds)	Performance bias: (slowest / fastest)
Contiguous, for time series	0.00003	0.00004?	1.3
Contiguous, for spatial slices	53?	0.003	?
4 MB chunks, 1032 x 29 x 35	1.2	1.0	1.2
16 KB chunks, 64 x 8 x 8	0.5	0.3	1.5
8 KB chunks, 46 x 6 x 7	0.6	0.2	2.4
4 KB chunks, 33 x 5 x 6	0.6	0.3	2.4

Note: the red timings are suspect, and probably indicate a bug

Summary: Available < Accessible

- Chunking is an under-appreciated tool with multiple benefits
- By rewriting important datasets using appropriate chunking, you can make them more useful
- Proper use of chunking can support multiple common query patterns for large datasets
- Specific guidance for how to choose optimal shapes and sizes of multidimensional chunks is becoming more widely available

More Information

HDF5 white paper on chunking

  www.hdfgroup.org/HDF5/doc/Advanced/Chunking/

Documentation of nccopy, h5repack

  www.unidata.ucar.edu/netcdf/docs/nccopy-man-1.html

  www.hdfgroup.org/HDF5/doc/RM/Tools.html - Tools-Repack

Good paper on chunking details

  www.escholarship.org/uc/item/35201092

Unidata Developer's Blog

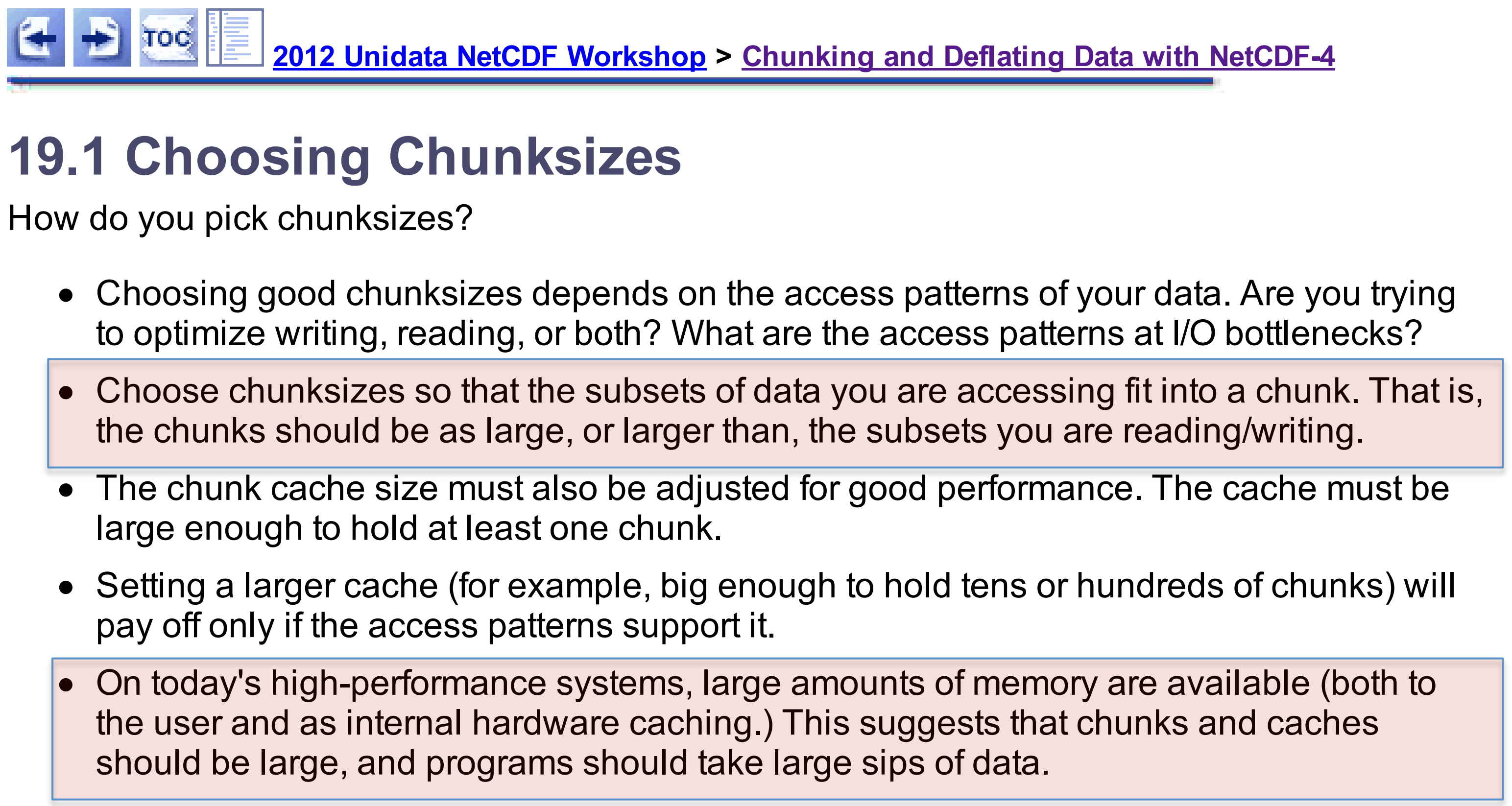
  www.unidata.ucar.edu/blogs/developer/en/tags/chunking

Thank you!

Benchmark details

- Disk cache in memory cleared before each run
- Reported average clock time to read at least 100 time ranges and spatial slices
- There were no common chunks among the time ranges or spatial slices, to avoid benefits of caching
- There was still some speedup from first read to later reads, due to disk caches not in OS memory
- Used local 7200 rpm disk for most tests (44 MB/sec)
- SSD was about 8x faster in sample comparison runs

Questionable chunking advice example



2012 Unidata NetCDF Workshop > Chunking and Deflating Data with NetCDF-4

19.1 Choosing Chunksizes

How do you pick chunksizes?

- Choosing good chunksizes depends on the access patterns of your data. Are you trying to optimize writing, reading, or both? What are the access patterns at I/O bottlenecks?
- Choose chunksizes so that the subsets of data you are accessing fit into a chunk. That is, the chunks should be as large, or larger than, the subsets you are reading/writing.
- The chunk cache size must also be adjusted for good performance. The cache must be large enough to hold at least one chunk.
- Setting a larger cache (for example, big enough to hold tens or hundreds of chunks) will pay off only if the access patterns support it.
- On today's high-performance systems, large amounts of memory are available (both to the user and as internal hardware caching.) This suggests that chunks and caches should be large, and programs should take large sips of data.