Uncharted Territory –
Exploring New Frontiers for HDF5

Quincey Koziol, Principal Engineer | HDF5 User Group Meeting - 2024

Agenda

- Introduction
- Accelerator-Native I/O Pipeline
- Sharded Storage
- Streaming Access

Uncharted Territory

Exploring New Frontiers for HDF5

- A sampling of near-future HDF5 projects
  - Accelerator-native I/O Pipeline
  - Sharded Storage
  - Streamed Access

Uncharted Territory

Exploring New Frontiers for HDF5

- A sampling of near-future HDF5 projects
  - Accelerator-native I/O Pipeline
  - Sharded Storage
  - Streamed Access

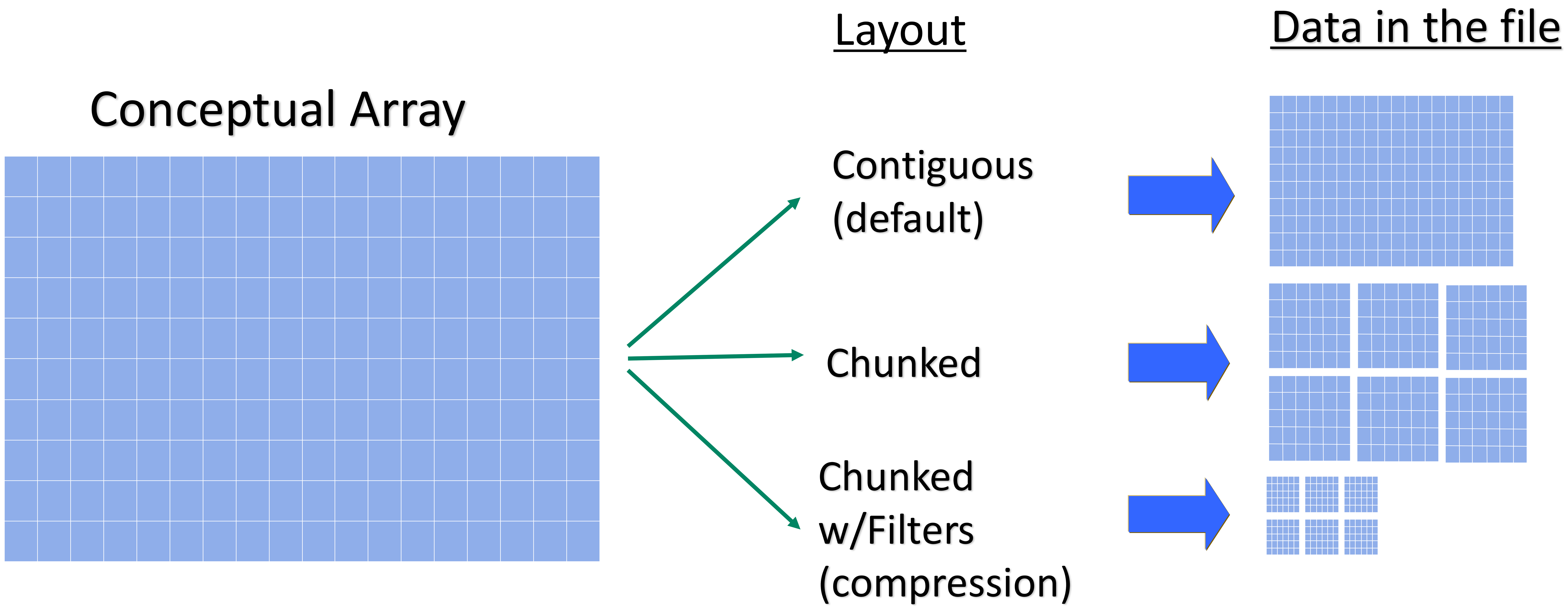
- Also on the horizon
  - Queries, with Indexing
  - Security, Resiliency, and Access-control
  - Data Warehousing
  - DPU Coprocessing
  - Cloud-native Storage
  - Concurrent Multiprocess Access

Accelerator-native I/O Pipeline

“We need another way of doing computing — so that we can continue to scale, so that we can continue to drive down the cost of computing, so that we can continue to consume more and more computing while being sustainable. Accelerated computing is a dramatic speedup over general-purpose computing, in every single industry.”

— Jensen Huang, NVIDIA CEO, GTC 2024

HDF5 Background



CPU-Based I/O Pipeline

CPU-Based I/O Pipeline

CPU-Based I/O Pipeline

CPU-Based I/O Pipeline

CPU-Based I/O Pipeline

NVIDIA

CPU-Based I/O Pipeline

CPU-Based I/O Pipeline

CPU-Based I/O Pipeline

CPU-Based I/O Pipeline

CPU-Based I/O Pipeline

CPU-Based I/O Pipeline

CPU-Based I/O Pipeline

CPU-Based I/O Pipeline

CPU-Based I/O Pipeline

GPU-Based I/O Pipeline

GPU-Based I/O Pipeline

GPU-Based I/O Pipeline

NVIDIA

GPU-Based I/O Pipeline

GPU-Based I/O Pipeline

GPU-Based I/O Pipeline

GPU-Based I/O Pipeline

GPU-Based I/O Pipeline

CPU Implementation

- I/O is performed with POSIX calls
- Dataset chunks are cached in CPU memory
- I/O filters run on CPU and access chunks in CPU memory
- Uncompressed data is transferred to GPU memory

CPU-Based I/O Pipeline

CPU-Based I/O Pipeline

CPU-Based I/O Pipeline

CPU-Based I/O Pipeline

I/O filters run on CPU and access chunks in CPU memory

CPU-Based I/O Pipeline

Uncompressed data is transferred to GPU memory

Accelerator Implementation

- I/O is performed with GPUDirect Storage (GDS) calls
- Dataset chunks are cached in GPU memory
- I/O filters run as GPU kernels and access chunks in GPU memory
  - Including using the hardware accelerated decompression engine in upcoming NVIDIA GPUs
- ~~Uncompressed data is transferred to GPU memory~~

GPU-Based I/O Pipeline

GPU-Based I/O Pipeline

GPU-Based I/O Pipeline

I/O is performed with GDS calls

GPU-Based I/O Pipeline

GPU-Based I/O Pipeline

How will the HDF5 library need to change?

- Update Virtual File Driver interface
  - Add flag(s) to indicate where a VFD plugin's source / destination buffers can be
- Update HDF5 chunk cache
  - Refactor to allow data to be cached in CPU, GPU, and possibly other memory
- Update HDF5 I/O filter interface
  - Add flag(s) to indicate where filter's source / destination buffers can be

How will applications need to change?

How will applications need to change?

They don't!

How will applications need to change?

They don’t!

Why?

How will applications need to change?

They don’t!

Why?

Library can auto-detect if buffer pointer for read / write is in GPU or CPU memory, then compose optimal I/O pipeline out of components that understand CPU / GPU memory

On The Horizon

Add support for “Data Processing Units” (DPUs)

> “The CPU is for general-purpose computing, the GPU is for accelerated computing, and the DPU, which moves data around the data center, does data processing.”
>
> — *NVIDIA Documentation*

On The Horizon

Add support for "Data Processing Units" (DPUs)

> "The CPU is for general-purpose computing, the GPU is for accelerated computing, and the DPU, which moves data around the data center, does data processing."
>
> — *NVIDIA Documentation*

Offload coordination and data movement operations to DPU

- Have the GPU contact the DPU for I/O requests
- DPU can send data directly back to GPU
- Keep metadata cache on DPU

On The Horizon

Add support for "Data Processing Units" (DPUs)

> "The CPU is for general-purpose computing, the GPU is for accelerated computing, and the DPU, which moves data around the data center, does data processing."
>
> — *NVIDIA Documentation*

Offload coordination and data movement operations to DPU

- Have the GPU contact the DPU for I/O requests
- DPU can send data directly back to GPU
- Keep metadata cache on DPU

Repurpose CPU to only general-purpose operations

- Build indices, track performance, do "weird" stuff

Sharding HDF5 Storage

HDF5 Native File Format Today

HDF5 Native File Format Today

HDF5 was built and optimized for high-speed POSIX I/O

- Self-describing, sequential data layout, in single file*
- Many custom data structures
    - B-trees, heaps, etc.
- Blocks of dataset elements: contiguous & chunked layout

HDF5 Native File Format Today

HDF5 was built and optimized for high-speed POSIX I/O

- Self-describing, sequential data layout, in single file*
- Many custom data structures
  - B-trees, heaps, etc.
- Blocks of dataset elements: contiguous & chunked layout

When running parallel applications with MPI, we compensated

- Require applications to perform collective metadata modification

HDF5 Native File Format Today

HDF5 was built and optimized for high-speed POSIX I/O

- Self-describing, sequential data layout, in single file*
- Many custom data structures
  - B-trees, heaps, etc.
- Blocks of dataset elements: contiguous & chunked layout

When running parallel applications with MPI, we compensated quickly

- Require applications to perform collective metadata modification

When cloud computing with object storage arose, we avoided it for 10 years

HDF5 Native File Format Today

HDF5 was built and optimized for high-speed POSIX I/O

- Self-describing, sequential data layout, in single file*
- Many custom data structures
  - B-trees, heaps, etc.
- Blocks of dataset elements: contiguous & chunked layout

When running parallel applications with MPI, we compensated quickly

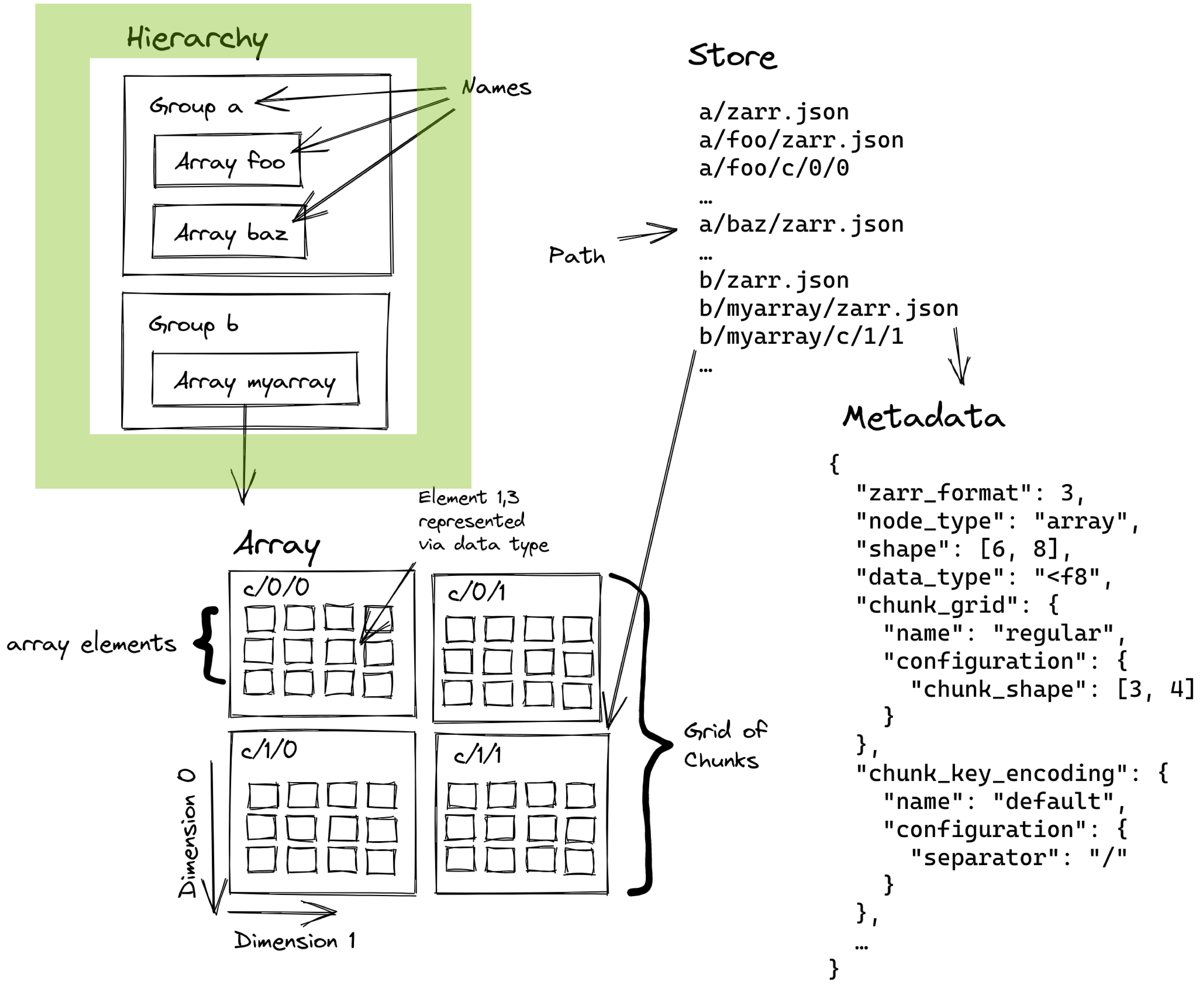
- Require applications to perform collective metadata modification

When cloud computing with object storage arose, we avoided it for 10 years

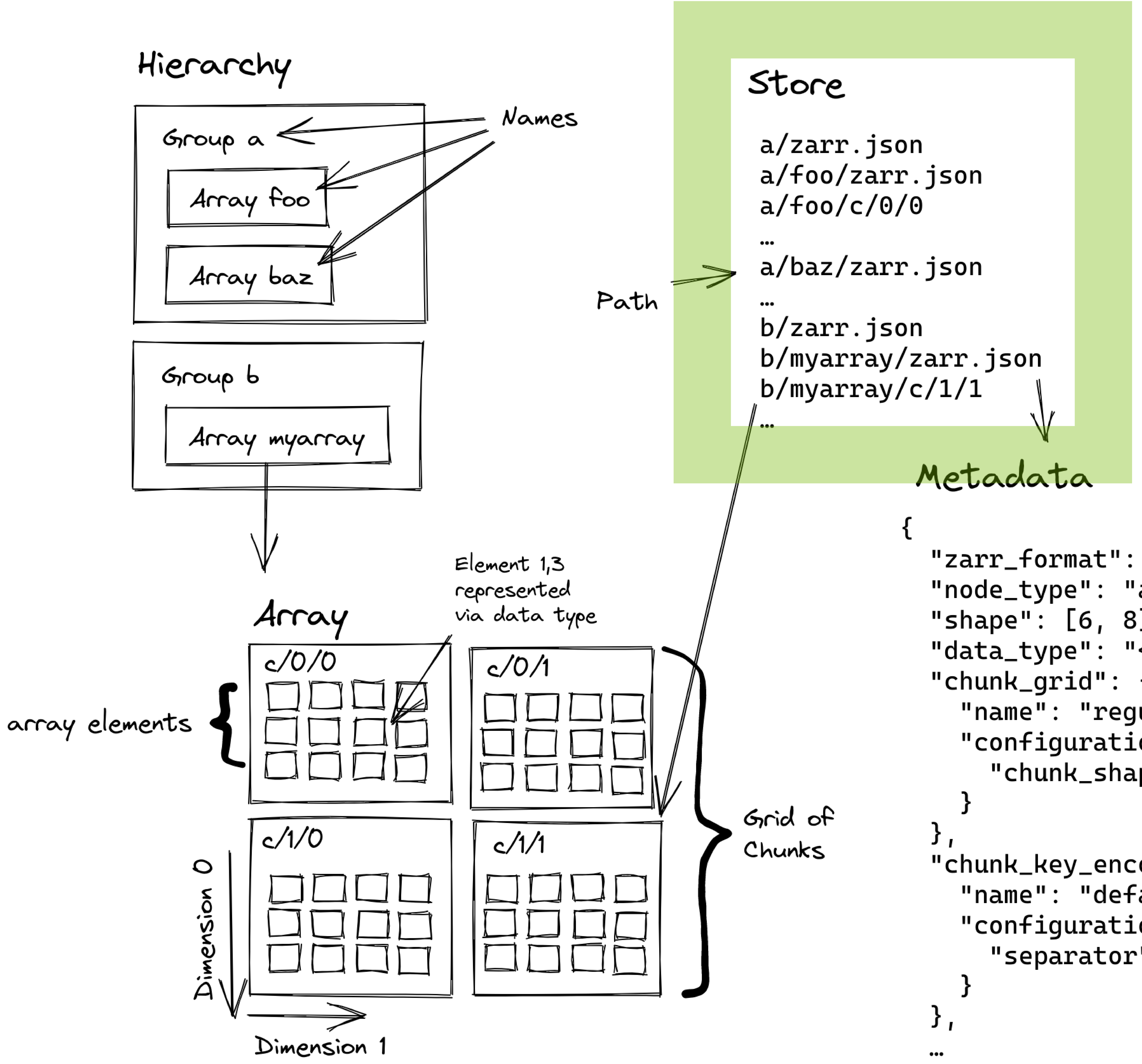
- Then started finding ways to compensate
- Highly Scalable Data Service (HSDS)
  - Cloud-hosted
  - JSON + dataset blocks
  - Requires server / service
- "Cloud-optimized" HDF5 access with read-only S3 VFD
  - Uses native file format, stored as single object

Compared to Zarr

Compared to Zarr

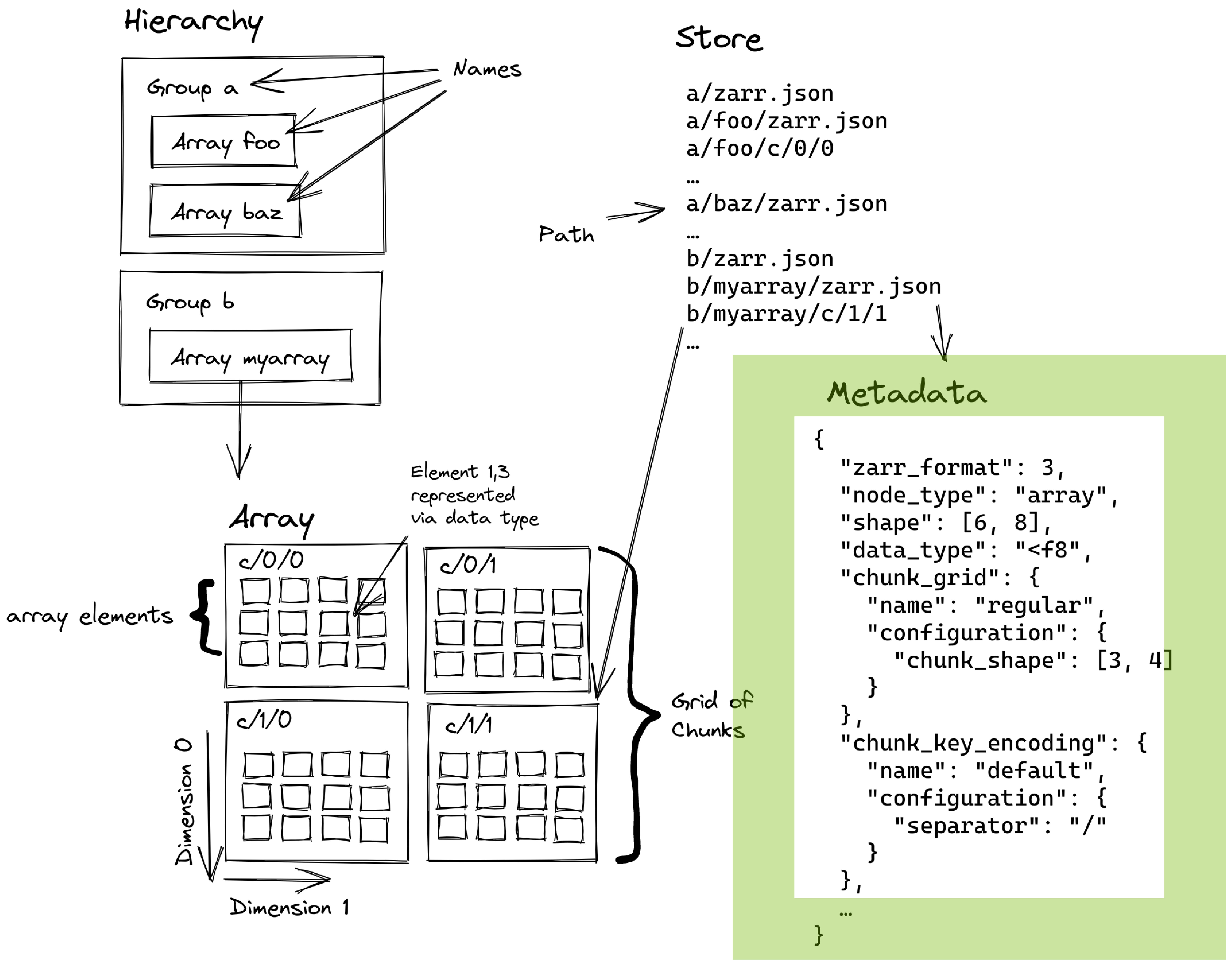


Compared to Zarr

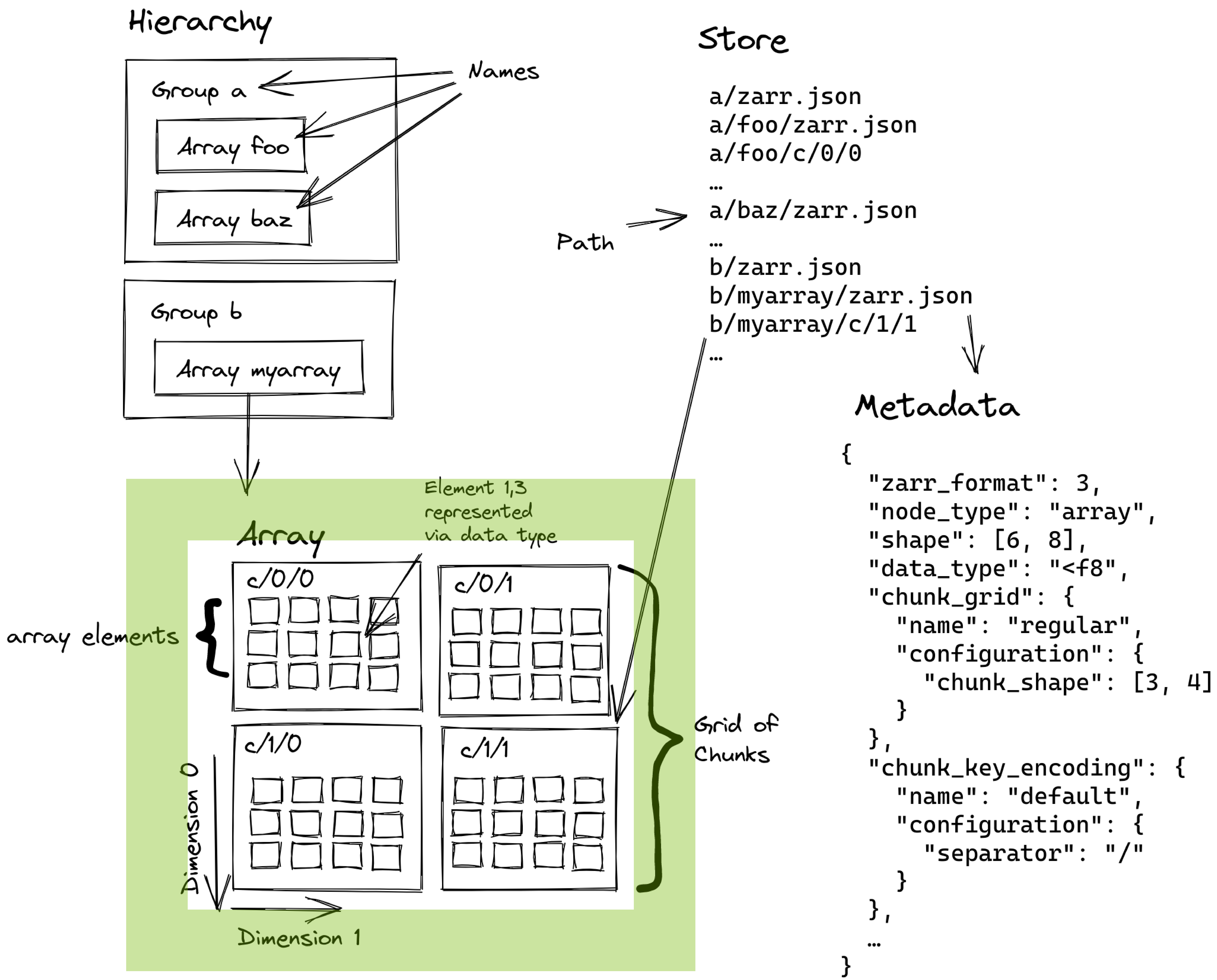



```
{
  "zarr_format": 3,
  "node_type": "array",
  "shape": [6, 8],
  "data_type": "<f8",
  "chunk_grid": {
    "name": "regular",
    "configuration": {
      "chunk_shape": [3, 4]
    }
  },
  "chunk_key_encoding": {
    "name": "default",
    "configuration": {
      "separator": "/"
    }
  },
  …
}
```

Compared to Zarr



Compared to Zarr



Rebasing for the Win

Rebasing for the Win

Credit to Jay Lofstead for the title ☺

Rebasing for the Win

Credit to Jay Lofstead for the title ☺

What would a modern HDF5 file format look like?

- **Targeting: POSIX file systems, object stores, cloud storage**

Rebasing for the Win

Credit to Jay Lofstead for the title ☺

What would a modern HDF5 file format look like?

- **Targeting: POSIX file systems, object stores, cloud storage**

- **What to keep**
  - **Data model concepts: groups, datasets, attributes, links, etc.**
  - **Self-describing format**
  - **Scalable I/O on array data**

Rebasing for the Win

Credit to Jay Lofstead for the title ☺

What would a modern HDF5 file format look like?

- Targeting: POSIX file systems, object stores, cloud storage

- What to keep
  - Data model concepts: groups, datasets, attributes, links, etc.
  - Self-describing format
  - Scalable I/O on array data
- What to leave behind
  - Metadata combined in same file as raw data
  - Custom file data structures

Rebasing for the Win

Credit to Jay Lofstead for the title ☺

What would a modern HDF5 file format look like?

- Targeting: POSIX file systems, object stores, cloud storage

- What to keep
  - Data model concepts: groups, datasets, attributes, links, etc.
  - Self-describing format
  - Scalable I/O on array data
- What to leave behind
  - Metadata combined in same file as raw data
  - Custom file data structures
- What to add
  - Sharded dataset storage
  - Database for metadata

Things to Keep

Data Model Concepts

- Critical to semantic model of HDF5
  - Datasets, Groups, Attributes, Links, Dataspaces, Datatypes

Self-Describing Format

- But make this more robust, so future format variants can always be auto-detected

Scalable Array I/O

- The *sine qua non* of HDF5

Things to Leave Behind

Metadata combined in same file as raw data

- Wow, was this a bad idea!
- At least in the ”small fragments of metadata scattered everywhere” form
  - Page-aligned blocks of metadata probably would have been OK

Custom file data structures

- At least most of them
  - The world did <u>not</u> need 2 more B-tree implementations, 3 kinds of heaps, etc.

Things to Add

Sharded dataset storage

- Learn from and exceed Zarr’s capabilities
- Sharded storage is much more friendly to object stores & cloud storage

Use a database for storing metadata

- Tuned and maintained B-trees, paged I/O, caching, etc.
- Opens up ability to query metadata in standard & scalable ways
- Local & remote possible
  - i.e. SQLite as well as DynamoDB

What will this look like?

What will this look like?

“Bundles”

What will this look like?

"Bundles"

Treat a directory as an HDF5 container

- Easy to detect when opening:
  - If the name you give is a file, open as a native format file
  - If the name you give is a directory, open as a bundle
- Creation property for new bundle containers

What will this look like?

“Bundles”

Treat a directory as an HDF5 container

- Easy to detect when opening:
  - If the name you give is a file, open as a native format file
  - If the name you give is a directory, open as a bundle
- Creation property for new bundle containers

Bootstrap bundle configuration from JSON “superblock” file

- Easy to read and understand
- Self-describing
- Specifies the name & type of the metadata database
  - File name or connection info to reach cloud database

What will this look like?

“Bundles”

Metadata Database File / Connection

- Need schema for tables and record information
  - Maybe:
    - A table per group, storing links for the group
    - A table per object, storing attributes
  - Still unknown:
    - How to store object’s metadata?
      - e.g. dataset dimensions, storage layout, etc?
    - Really want both tables & stores (for sets of key-value pairs) in DB

What will this look like?

"Bundles"

Metadata Database File / Connection

- Need schema for tables and record information
  - Maybe:
    - A table per group, storing links for the group
    - A table per object, storing attributes
  - Still unknown:
    - How to store object's metadata?
      - e.g. dataset dimensions, storage layout, etc?
    - Really want both tables & stores (for sets of key-value pairs) in DB

Dataset storage

- Lots of options that can leverage file system and object storage
  - Can aggregate 1+ datasets into single file
  - Can shard dataset chunks into file-per-chunk
  - Can use sub-filing in a more natural way
  - etc.

How to implement bundles?

How to implement bundles?

Extract [more] components from Native VOL connector

- Anything that will be common across many connectors
- Include in main HDF5 library, or create “I/O core” library
- Leave only the “knowledge” about the file format-specific aspects of the file in the Native VOL connector

How to implement bundles?

Extract [more] components from Native VOL connector

- Anything that will be common across many connectors
- Include in main HDF5 library, or create “I/O core” library
- Leave only the “knowledge” about the file format-specific aspects of the file in the Native VOL connector

Write New ‘Bundle’ VOL Connector

- Uses file system as object store
  - Making it easy to transition bundle to <-> from file system, on-prem, and cloud
- Implements metadata operations on database
  - Probably will abstract this as pluggable interface
    - Support SQLite, DynamoDB, etc.
- Owns the knowledge of where and how datasets are aggregated or sharded
- Calls “I/O core” library for as much functionality as possible

On The Horizon

Add support for concurrent bundle access

- Multiple processes accessing a database is well-known technology
- Sharding datasets enables easier modifications to both structure and contents of datasets

Streaming HDF5 Data

Why is Streaming Performance Bad?

HDF5's current API is very oriented to accessing arbitrary subsets of arrays:

- Set up description of dataspace selection in file
- Perform I/O operation
  - Construct I/O vector information for file and memory
  - Determine and initialize datatype conversion
  - Access file
  - [Type Convert]
- Destroy selection

Lots of overhead when operating in a loop!

- Especially when performing single-element appends
- More overhead for extending a dataset dimension in the loop
- *Even* worse when accessing variable-length elements

Code Comparison – Today's API

```
hid_t dataset_id;
hid_t dset_space_id, mem_space_id;
hid_t dxpl_id;
hid_t datatype_id;
hsize_t dset_dims[1];
hsize_t hyperslab_count[1];
...

// Set up datatype for the dataset and buffer
datatype_id = ...;

// Create 1-D dataset with 0 elements
dset_dims[0] = 0;
dataset_id = H5Dcreate2(...);
```

How to Improve Performance?

Extract all common setup & teardown from loop

- Dimension and number of elements to append
- Datatype of memory buffer

Significant reduction in overhead:

```
<initialize streaming>

loop:
      <append>

<shutdown streaming>
```

← Describe everything

- Create stream context

← Copy data to internal buffer

- Periodically [convert] and flush

← Finish last [convert] and flush

- Destroy stream context

New Streaming API Routines

```
hid_t H5Dcursor_create(hid_t dataset_id, H5D_cursor_mode_t
      mode, unsigned axis, size_t extension, hid_t mem_type_id,
      hid_t dxpl_id)

herr_t H5Dcursor_write(hid_t cursor_id, const void *buf)

herr_t H5Dcursor_read(hid_t cursor_id, void *buf)

herr_t H5Dcursor_close(hid_t cursor_id)
```

Code Comparison – Streaming API

```
hid_t dataset_id;
hid_t cursor_id;
hid_t dset_space_id;
hid_t dxpl_id;
hid_t datatype_id;
hsize_t dset_dims[1];
...

// Set up datatype for the dataset and buffer
datatype_id = ...;

// Create 1-D dataset with 0 elements
dset_dims[0] = 0;
dataset_id = H5Dcreate2(...);
```

Optimizing Streamed Storage

HDF5 “extensible array” (E-array) file data structure

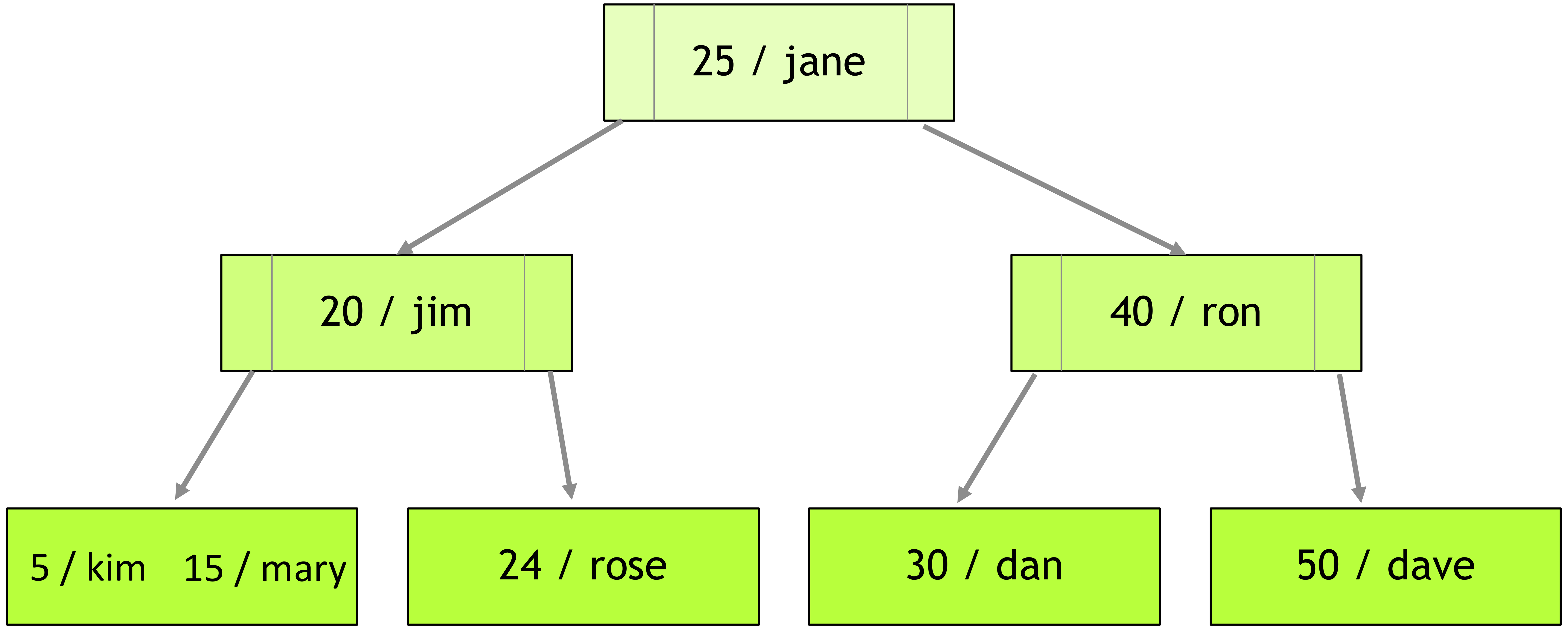
- Provides constant time lookup, i.e. O(1)
- Also provides constant time append, i.e. O(1)
- Used for indexing chunks of datasets with one unlimited dimension
  - Each element in E-array is ”chunk record” that points at a chunk of fixed-size elements in the file
    - If no compression, each chunk is same size, even when storing variable-sized datatype elements
  - Variable-length data is stored in a separate heap file data structure, referenced by fixed-size heap IDs in the elements within the chunks

Optimizing Streamed Storage

HDF5 “extensible array” (E-array) file data structure

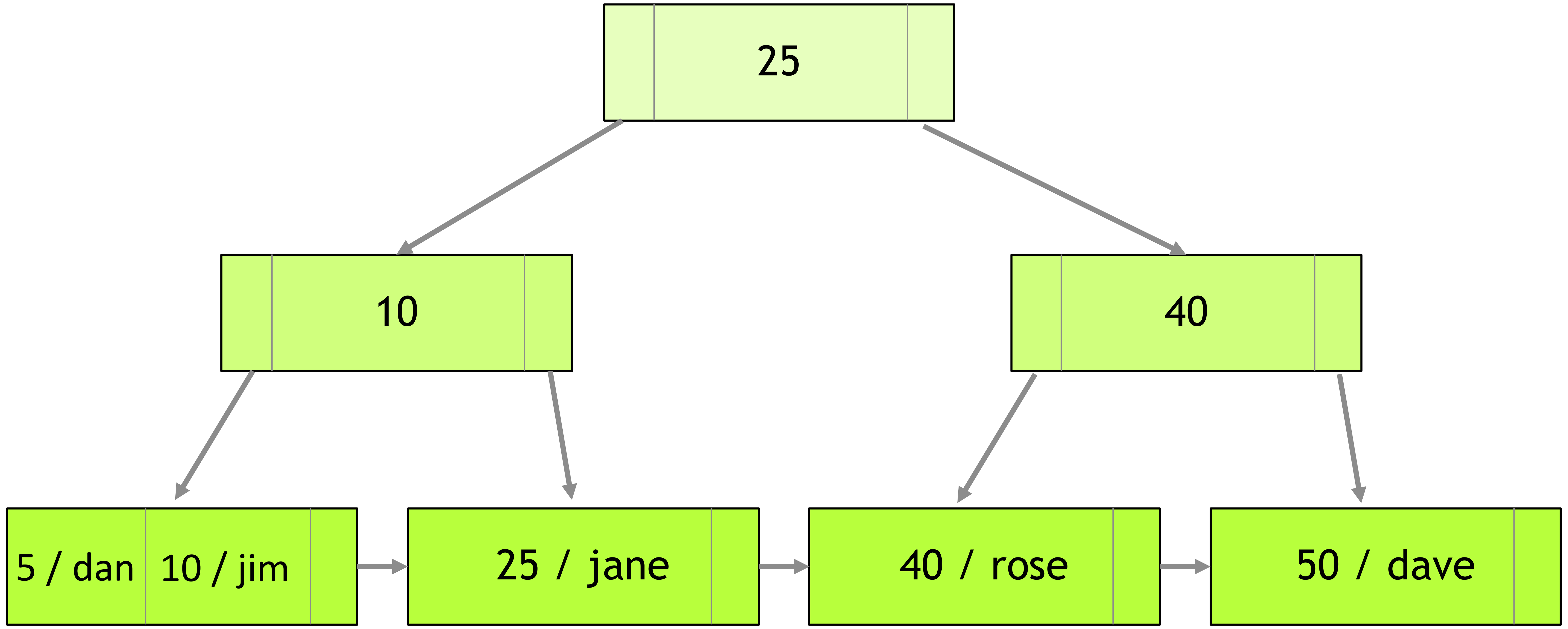
Optimizing Streamed Storage

B-tree



Optimizing Streamed Storage

B+-tree



Optimizing Streamed Storage

E+-array

Optimizing Streamed Storage

E+-array


424
Super Block #0
Super Block #1
Super Block #2
16/16
32/32
32/32
32/32
64/64
Metadata
Fixed-Length Data
Variable-Length Data
Next
Metadata
Fixed-Length Data
Variable-Length Data
Leaf Node #0
Leaf Node #1
NVIDIA

424
Super Block #0
Super Block #1
Super Block #2
16/16
32/32
32/32
32/32
64/64
Metadata
Fixed-Length Data
Variable-Length Data
Next
Metadata
Fixed-Length Data
Variable-Length Data
Leaf Node #0
Leaf Node #1


424
Super Block #0
Super Block #1
Super Block #2
16/16
32/32
32/32
32/32
64/64
Metadata
Fixed-Length Data
Variable-Length Data
Next
Metadata
Fixed-Length Data
Variable-Length Data
Leaf Node #0
Leaf Node #1

Optimizing Streamed Storage

Benefits of E+-arrays with extended chunk format for streaming

- Still provides constant time lookup, i.e. O(1)
- Continues to provide constant time append, i.e. O(1)
- Now also provides zero index accesses when streaming through elements
  - Can ignore index after first lookup when reading
  - Can lazily create index when writing
    - Or even <u>never</u> create the index, if streaming reads will be only accesses in the future
- For variable-length datatypes, eliminates <u>all</u> extra I/O accesses to retrieve variable-length info from heap
  - All VL info is contained within chunk, and brought into memory in one I/O operation, with fixed-size components of datatype

NVIDIA