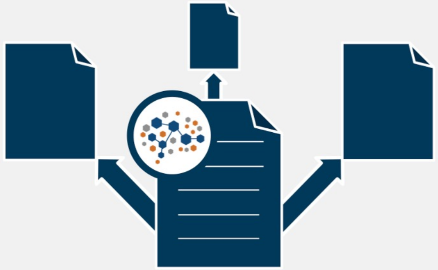
PyNWB: Advanced Data I/O

Oliver Rübel

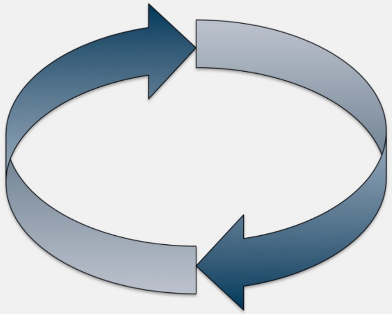
Machine Learning & Analytics Group
Computational Biosciences Group
Scientific Data Division
Lawrence Berkeley National Laboratory

Overview

Modular Data Storage using External Files

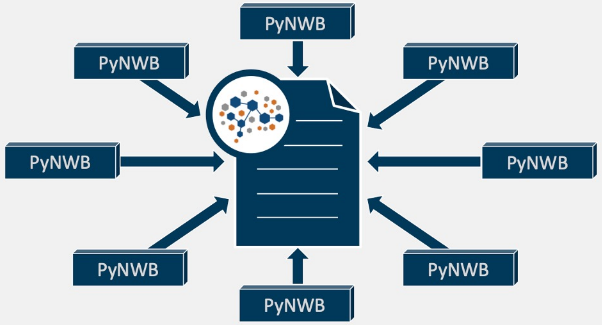


Iterative Data Write



HDF5 Dataset I/O

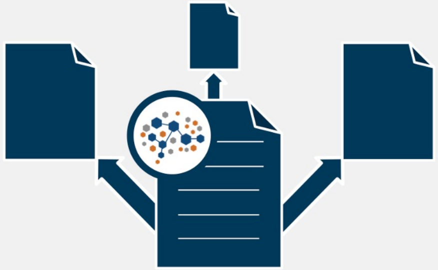
Parallel I/O using MPI



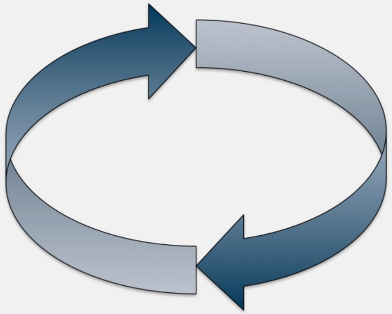
https://pynwb.readthedocs.io/en/stable/tutorials/#advanced-i-o

Next Up

Modular Data Storage using External Files



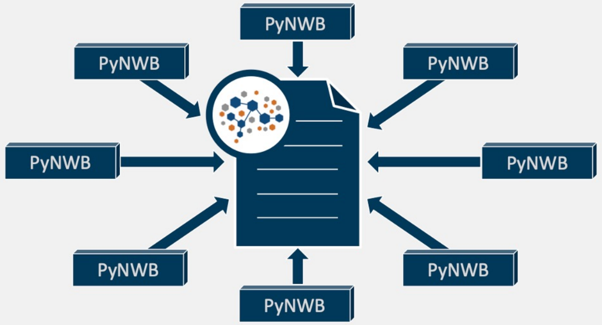
Iterative Data Write



HDF5 Dataset I/O

Defining HDF5 Dataset I/O Settings: Chunking and Compression

Parallel I/O using MPI

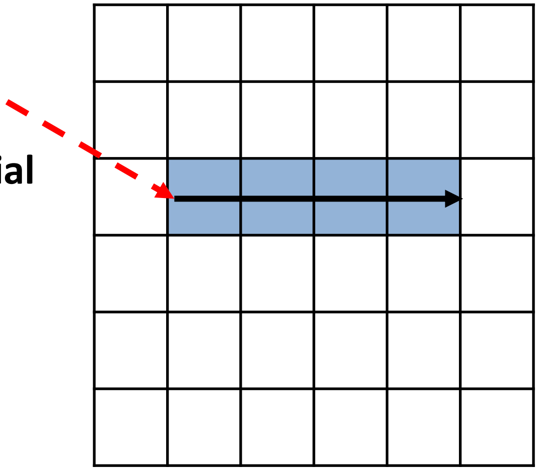


Contiguous vs. chunked storage

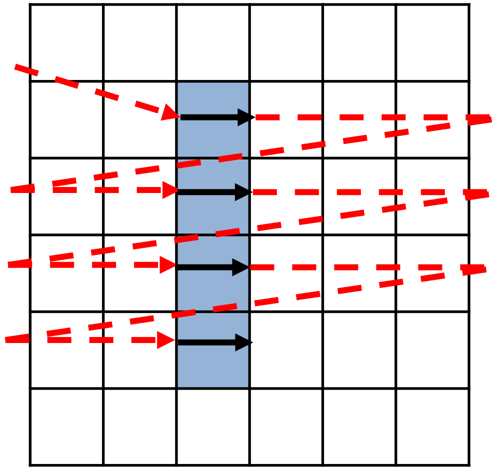
Data is ultimately stored in one-dimensional compute memory

Contiguous Dataset

Read partial row

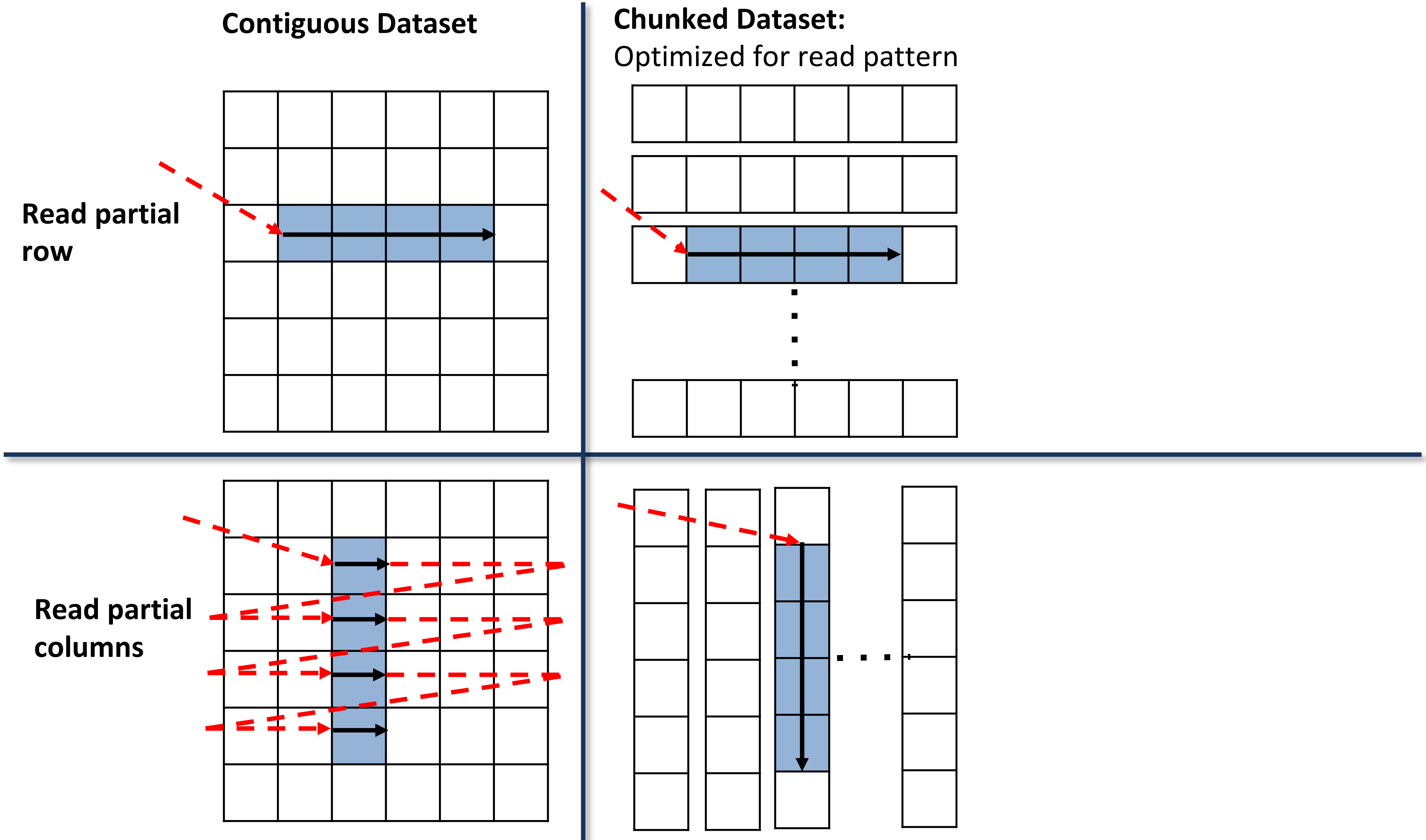


Read partial columns



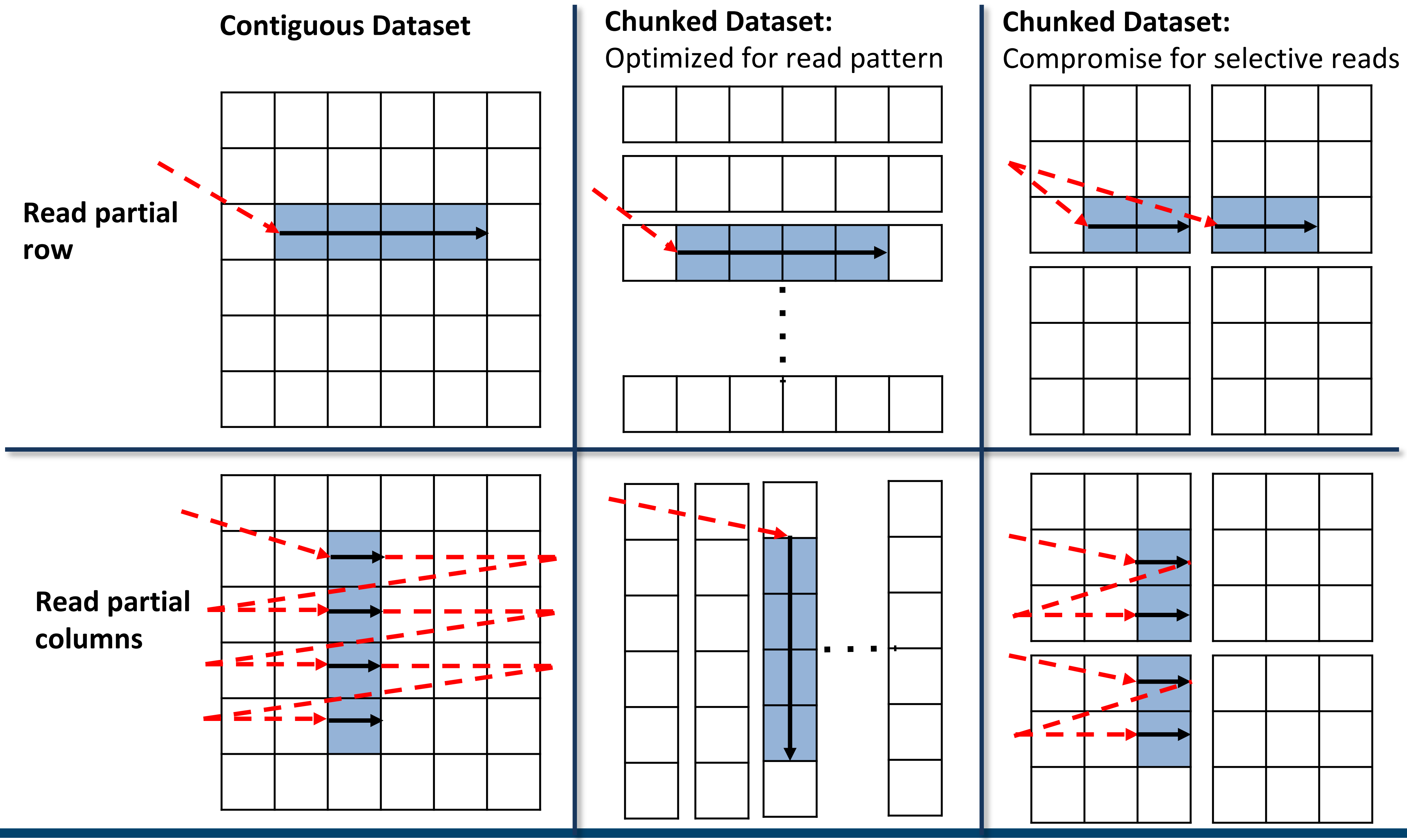
Contiguous vs. chunked storage

Data is ultimately stored in one-dimensional compute memory



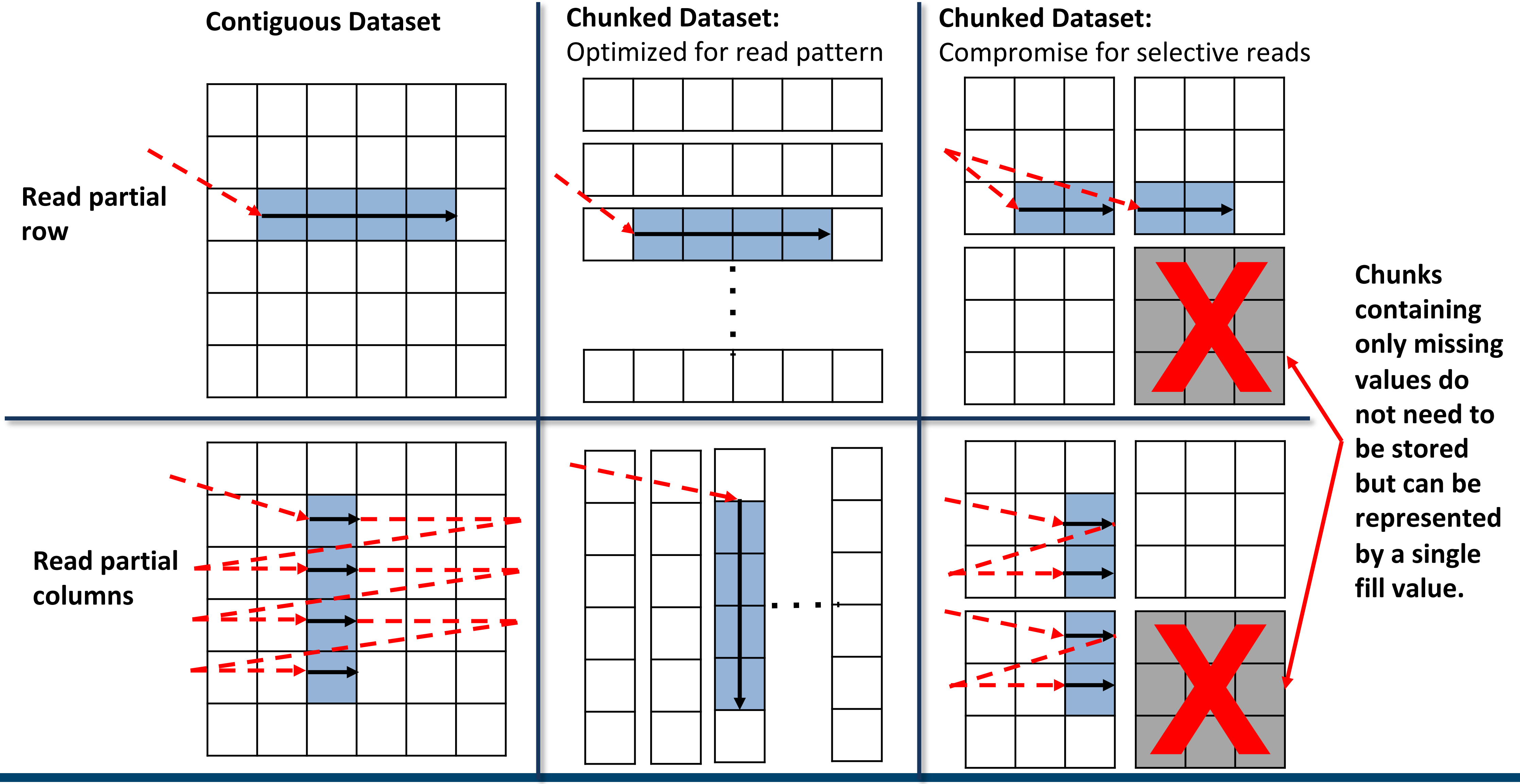
Contiguous vs. chunked storage

Data is ultimately stored in one-dimensional compute memory



Contiguous vs. chunked storage

Data is ultimately stored in one-dimensional compute memory



Chunking data with H5DataIO

1. Create a test NWBFile

```python
from datetime import datetime
from dateutil.tz import tzlocal
from pynwb import NWBFile

nwbfile = NWBFile(
    session_description="demonstrate advanced HDF5 I/O features",
    identifier="NWB123",
    session_start_time=datetime(2017, 4, 3, 11, tzinfo=tzlocal()))
```

2. Wrap data with H5DataIO for chunking and create a TimeSeries with that data

```python
from hdmf.backends.hdf5.h5_utils import H5DataIO

data = np.arange(10000).reshape((1000, 10))
wrapped_data = H5DataIO(
    data=data,
    chunks=True,  # <---- Enable chunking
    maxshape=(None, 10),  # <---- Make the time dimension unlimited and hence resizable
)
test_ts = TimeSeries(
    name="test_chunked_timeseries",
    data=wrapped_data,  # <----
    unit="SIunit",
    starting_time=0.0,
    rate=10.0,
)
nwbfile.add_acquisition(test_ts)
```

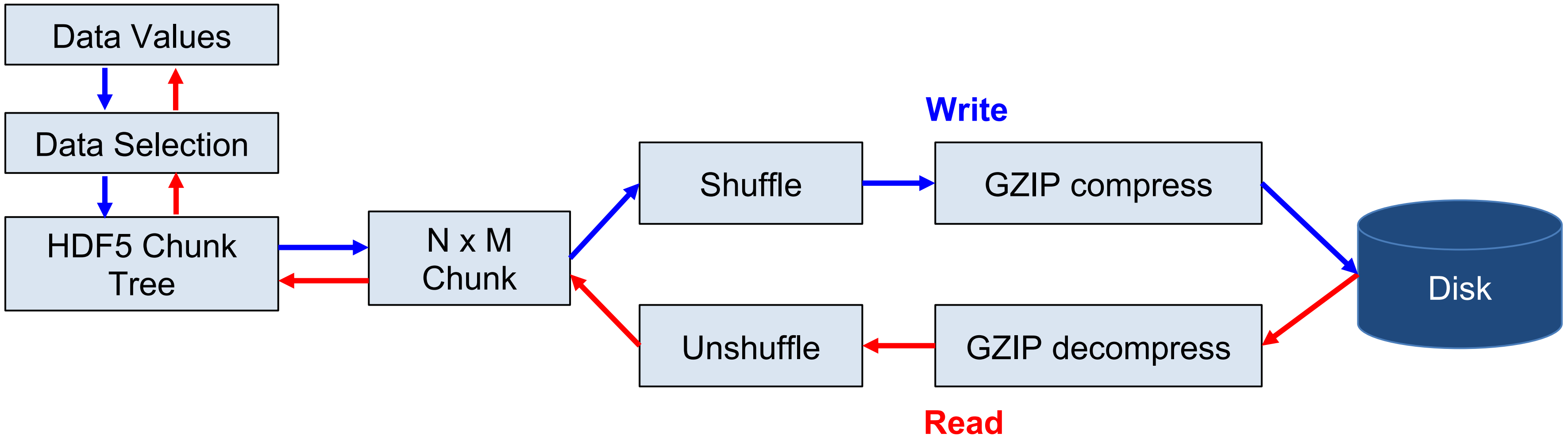
Writing data wrapped with H5DataIO

3. Write/Read the data as usual

```
from pynwb import NWBHDF5IO

with NWBHDF5IO("advanced_io_example.nwb", "w") as io:
    io.write(nwbfile)
```

Chunking enables the use of HDF5 I/O filter pipelines

- I/O filters operate on data chunks
- Each filter is free to do anything it wants to the data in a chunk, e.g., compress it, checksum it, add metadata etc.
- On read, each filter is run in “reverse” to reconstruct the original data



Compressing data with H5DataIO

1. Create a test NWBFile

2. Wrap data with H5DataIO for compression and create a TimeSeries with that data

```
from hdmf.backends.hdf5.h5_utils import H5DataIO

wrapped_data = H5DataIO(
    data=data,
    compression="gzip",  # <---- Use GZip
    compression_opts=4,  # <---- Optional GZip aggression option
)
test_ts = TimeSeries(
    name="test_gzipped_timeseries",
    data=wrapped_data,  # <----
    unit="SIunit",
    starting_time=0.0,
    rate=10.0,
)
nwbfile.add_acquisition(test_ts)
```

3. Write/Read the data as usual

Using dynamically loaded filters with H5DataIO

1. Install common HDF5 filter plugins

```
pip install hdf5plugin
```

2. Wrap data with H5DataIO for compression and create a TimeSeries with that data

```python
import hdf5plugin
from hdmf.backends.hdf5.h5_utils import H5DataIO

from pynwb.file import TimeSeries

wrapped_data = H5DataIO(
    data=data,
    **hdf5plugin.Zstd(clevel=3),  # set the compression and compression_opts parameters
    allow_plugin_filters=True,
)

test_ts = TimeSeries(
    name="test_gzipped_timeseries",
    data=wrapped_data,
    unit="SIunit",
    starting_time=0.0,
    rate=10.0,
)
```

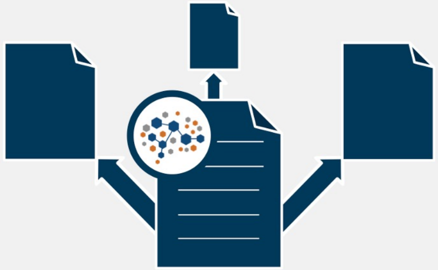
3. Write/Read the data as usual

Tips

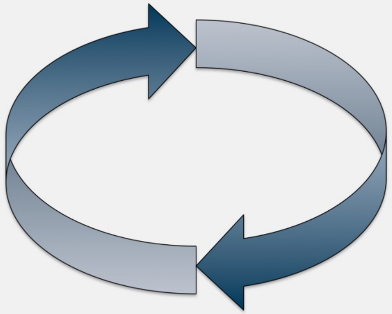
- Caution! Chunking and I/O filters can help to significantly reduce storage and I/O cost, but if used wrong, can also harm performance
  - Don't make many tiny chunks
  - Poor chunk size can actually increase your file size
  - Gzip level 9 is computationally costly for little gain
- **`H5DataIO`** supports other filters:
  - computing checksum using Fletcher-32 algorithm
  - more optimized compression with shuffling
  - Third-party filters, e.g., via `hdf5plugin`

Next Up

Modular Data Storage using External Files

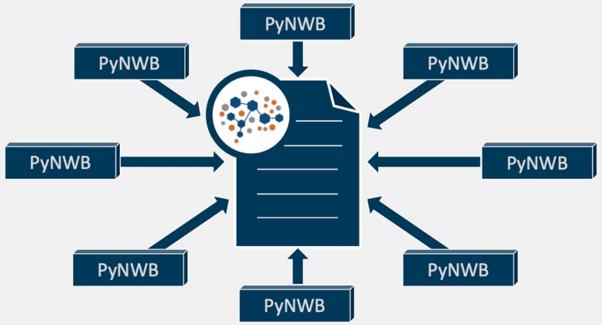


Iterative Data Write



HDF5 Dataset I/O

Parallel I/O using MPI



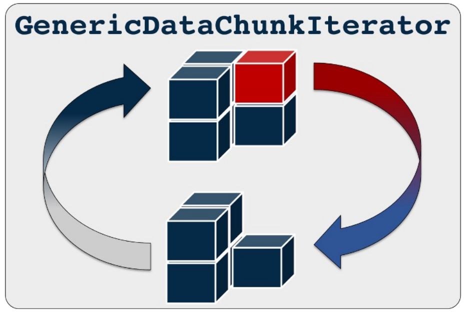
Iterating over data arrays

- `DataChunk`: Data structure to describe a data chunk, i.e., a subset of larger data array:
  - `DataChunk.data`: The subarray data values
  - `DataChunk.selection`: NumPy index tuple describing the location of the chunk. e.g.:
    `numpy.s_[0:10, …] □ (slice(0, 10, None), Ellipsis))`

- `AbstractDataChunkIterator`: Abstract base class for iterating over data arrays one `DataChunk` at a time:
  - `__iter__`: Get the iterator (usually `self`)
  - `__next__`: Get the next `DataChunk`
  - `dtype`, `maxshape`: Data type and maximum size of the array (if known)
  - `recommended_data_shape`: Recommended initial size of the target array
  - `recommended_chunk_shape`: Recommended chunking of the target array

Iterating over data arrays

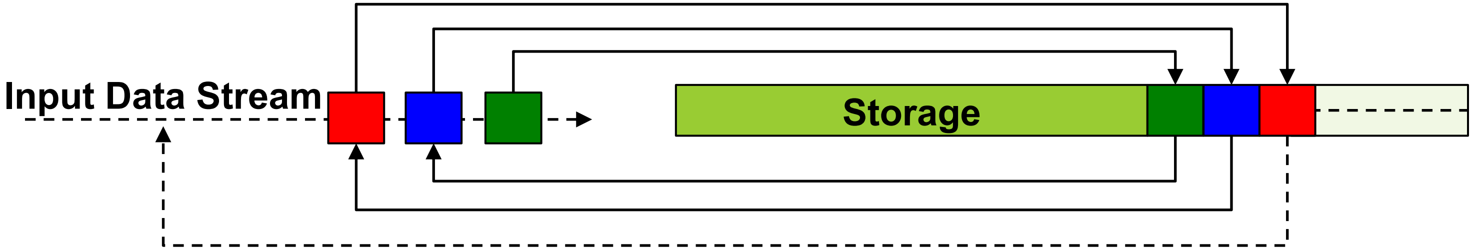
- **`DataChunkIterator`:** Buffered data chunk iterator for wrapping `Iterable` objects that iterates over the first dimension of an array one chunk at a time (e.g., lists, generators etc.)

- **`GenericDataChunkIterator`:** Semi-abstract version of a AbstractDataChunkIterator that automatically handles the selection of buffer regions and resolves communication of compatible chunk regions. User specify chunk and buffer shapes or sizes and the iterator manages how to break the data up for write.
  - Users must define:
    - `_get_data(self, selection)`: Retrieve the data specified by the selection using minimal I/O
    - `_get_maxshape()`: Retrieve the maximum bounds of the data shape using minimal I/O
    - `_get_dtype()`: Retrieve the dtype of the data using minimal I/O
  - 

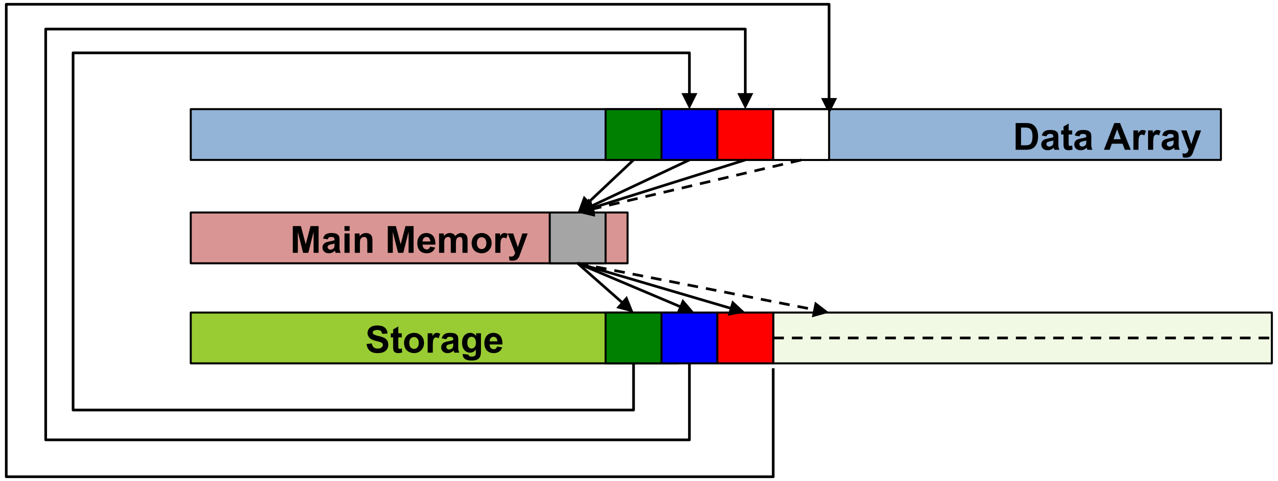
https://hdmf.readthedocs.io/en/stable/tutorials/plot_generic_data_chunk_tutorial.html

Example applications

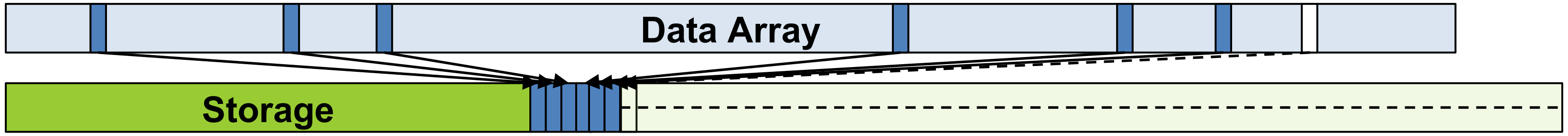
- **Data streaming/generators**:
  - Write data as it is being generated / as it arrives
  - Avoid caching all data in memory
  - The total size of the data is often unknown ahead of time.



- **Converting Large data arrays**
  - Avoid loading the whole array into memory



- **Sparse data arrays:**
  - Reduce I/O (and storage) cost.
  - Avoid writing uninitialized data values.



Iterative write of large binary file

1. Create a generator for our large data array (here a .npy numpy file)

```
def iter_largearray(filename, shape, dtype="float64"):
    """
    Generator reading [chunk_size, :] elements from our array in each iteration.
    """
    for i in range(shape[0]):
        # Open the file and read the next chunk
        newfp = np.memmap(filename, dtype=dtype, mode="r", shape=shape)
        curr_data = newfp[i : (i + 1), ...][0]
        del newfp  # Reopen the file in each iterator to prevent accumulation of data in memor
        yield curr_data
    return
```

2. Wrap the generator in a DataChunkIterator

```
from hdmf.data_utils import DataChunkIterator

data = DataChunkIterator(
    data=iter_largearray(
        filename="basic_sparse_iterwrite_testdata.npy", shape=datashape
    ),
    maxshape=datashape,
    buffer_size=10,
)  # Buffer 10 elements into a chunk, i.e., create chunks of shape (10,10)
```

3. Write the data as usual

Alternatively use `AbstractDataChunkIterator` or `GenericDataChunkIterator`.

NEURODATA WITHOUT BORDERS

User-defined Dataset Write

1. Initially allocate the data as empty

```
from hdmf.backends.hdf5.h5_utils import H5DataIO

# Use H5DataIO to specify how to setup the dataset in the file
dataio = H5DataIO(
    shape=(0, 10),  # Initial shape. If the shape is known then set to full shape
    dtype=np.dtype("float"),  # dtype of the dataset
    maxshape=(None, 10),  # Make the time dimension resizable
    chunks=(131072, 2),  # Use 2MB chunks
    compression="gzip",  # Enable GZip compression
    compression_opts=4,  # GZip aggression
    shuffle=True,  # Enable shuffle filter
    fillvalue=np.nan,  # Use NAN as fillvalue
)
```

2. Write the data as usual but keep the I/O object open so that we can modify the data

```
io = write_test_file(
    filename="basic_alternative_custom_write.nwb", data=dataio, close_io=False
)
```

User-defined Dataset Write

3. Update the dataset and close the I/O

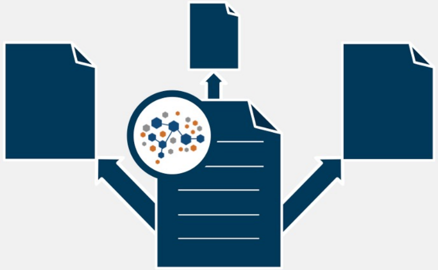
```python
# Allocate space. Only needed if we didn't set the initial shape large enough
dataio.dataset.resize((8, 10))

# Write 1s in timesteps 0-2
dataio.dataset[0:3, :] = 1

# Write 2s in timesteps 3-5
# NOTE: timesteps 6 and 7 are not being initialized
dataio.dataset[3:6, :] = 2
```

4. Close the I/O

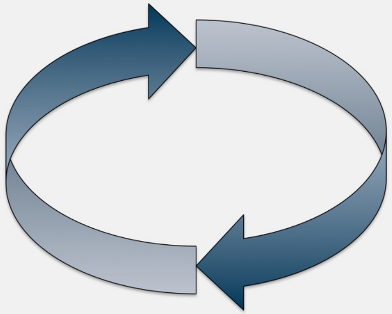
```python
# Close the file
io.close()
```

Next Up

Modular Data Storage using External Files

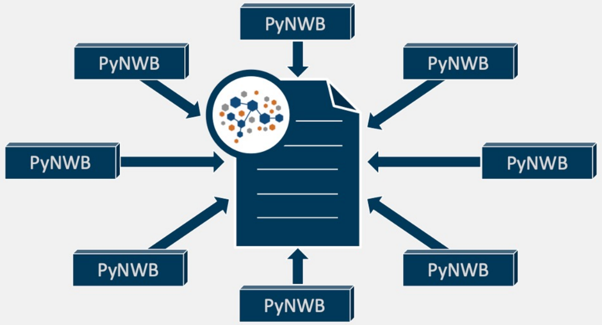


Iterative Data Write



HDF5 Dataset I/O

Parallel I/O using MPI

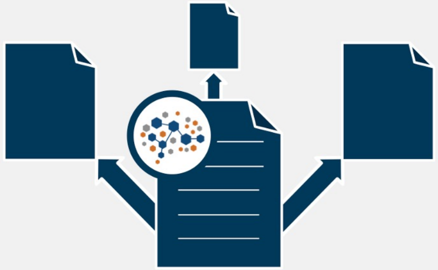


Parallel I/O using MPI

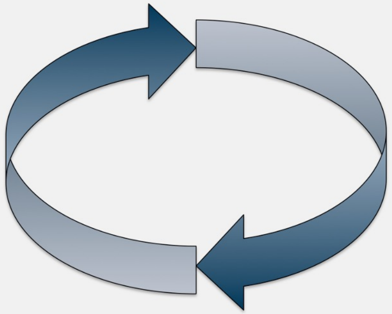
```python
from mpi4py import MPI
import numpy as np
from dateutil import tz
from pynwb import NWBHDF5IO, NWBFile, TimeSeries
from datetime import datetime
from hdmf.backends.hdf5.h5_utils import H5DataIO

start_time = datetime(2018, 4, 25, 2, 30, 3, tzinfo=tz.gettz("US/Pacific"))
fname = "test_parallel_pynwb.nwb"
rank = MPI.COMM_WORLD.rank  # The process ID (integer 0-3 for 4-process run)

# Create file on one rank. Here we only instantiate the dataset we want to
# write in parallel but we do not write any data
if rank == 0:
    nwbfile = NWBFile("aa", "aa", start_time)
    data = H5DataIO(shape=(4,), maxshape=(4,), dtype=np.dtype("int"))

    nwbfile.add_acquisition(
        TimeSeries(name="ts_name", description="desc", data=data, rate=100.0, unit="m")
    )
    with NWBHDF5IO(fname, "w") as io:
        io.write(nwbfile)

# write to dataset in parallel
with NWBHDF5IO(fname, "a", comm=MPI.COMM_WORLD) as io:
    nwbfile = io.read()
    print(rank)
    nwbfile.acquisition["ts_name"].data[rank] = rank

# read from dataset in parallel
with NWBHDF5IO(fname, "r", comm=MPI.COMM_WORLD) as io:
    print(io.read().acquisition["ts_name"].data[rank])
```

Next Up

Modular Data Storage using External Files

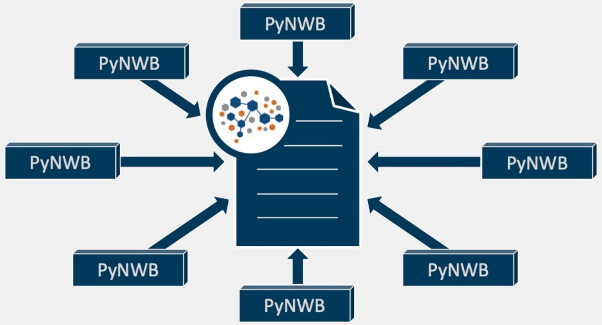


Iterative Data Write



HDF5 Dataset I/O

Parallel I/O using MPI



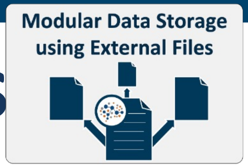
Example use cases

- Data are distributed across files (e.g. by session, electrode)
  - Master file with links to different sources

- Large stimulus data is common to multiple recordings
  - Link to stimulus data across files, avoid duplicating data!

- Keep original file with raw data read-only and use separate file for data processing with links to raw data

Example use case: Storing data across multiple files



- Data are distributed across files (e.g. by session, electrode)
  - Master file with links to different sources

- We will use **`TimeSeries`** as an example, but the same approach works for other **`NWBContainer`** objects as well

Linking to whole Containers

1. Get the `BuildManager`

```
from pynwb import get_manager

manager = get_manager()
```

Linking to whole Containers

1. Get the BuildManager

```
from pynwb import get_manager

manager = get_manager()
```

2. Pass into NWBHDF5IO the same BuildManager and get the TimeSeries objects you want to link to

```
io1 = NWBHDF5IO(filename1, 'r', manager=manager)
nwbfile1 = io1.read()
timeseries_1 = nwbfile1.get_acquisition('test_timeseries1')

io2 = NWBHDF5IO(filename2, 'r', manager=manager)
nwbfile2 = io2.read()
timeseries_2 = nwbfile2.get_acquisition('test_timeseries2')
```

Linking to whole Containers

1. Get the `BuildManager`

```
from pynwb import get_manager

manager = get_manager()
```

2. Pass into `NWBHDF5IO` the same `BuildManager` and get the `TimeSeries` objects you want to link to

```
io1 = NWBHDF5IO(filename1, 'r', manager=manager)
nwbfile1 = io1.read()
timeseries_1 = nwbfile1.get_acquisition('test_timeseries1')

io2 = NWBHDF5IO(filename2, 'r', manager=manager)
nwbfile2 = io2.read()
timeseries_2 = nwbfile2.get_acquisition('test_timeseries2')
```

3. Add both `TimeSeries` to another `NWBFile`

```
nwbfile3 = NWBFile(...)
nwbfile3.add_acquisition(timeseries_1)
nwbfile3.add_acquisition(timeseries_2)
```

Linking to whole Containers

1. Get the BuildManager

```
from pynwb import get_manager

manager = get_manager()
```

2. Pass into NWBHDF5IO the same BuildManager and get the TimeSeries objects you want to link to

```
io1 = NWBHDF5IO(filename1, 'r', manager=manager)
nwbfile1 = io1.read()
timeseries_1 = nwbfile1.get_acquisition('test_timeseries1')

io2 = NWBHDF5IO(filename2, 'r', manager=manager)
nwbfile2 = io2.read()
timeseries_2 = nwbfile2.get_acquisition('test_timeseries2')
```

3. Add both TimeSeries to another NWBFile

```
nwbfile3 = NWBFile(...)
nwbfile3.add_acquisition(timeseries_1)
nwbfile3.add_acquisition(timeseries_2)
```

4. Write the data with the BuildManager

```
io3 = NWBHDF5IO(filename3, 'w', manager=manager)
io3.write(nwbfile3)
io3.close()
```

Linking to select datasets

1. Create the new `NWBFile`

```
nwbfile4 = NWBFile(...)
```

Linking to select datasets

1. Create the new `NWBFile`

```
nwbfile4 = NWBFile(...)
```

2. Get data from the `TimeSeries` you want to link to

```
io1 = NWBHDF5IO(filename1, 'r')
nwbfile1 = io1.read()
timeseries_1 = nwbfile1.get_acquisition('test_timeseries1')
timeseries_1_data = timeseries_1.data
```

Linking to select datasets

1. Create the new `NWBFile`

```
nwbfile4 = NWBFile(...)
```

2. Get data from the `TimeSeries` you want to link to

```
io1 = NWBHDF5IO(filename1, 'r')
nwbfile1 = io1.read()
timeseries_1 = nwbfile1.get_acquisition('test_timeseries1')
timeseries_1_data = timeseries_1.data
```

3. Add the data to a new `TimeSeries`

```
test_ts4 = TimeSeries(name='test_timeseries4',
                      data=timeseries_1_data,   # <-------
                      unit='SIunit',
                      timestamps=timestamps)
nwbfile4.add_acquisition(test_ts4)
```

Linking to select datasets

1. Create the new `NWBFile`

```
nwbfile4 = NWBFile(...)
```

2. Get data from the `TimeSeries` you want to link to

```
io1 = NWBHDF5IO(filename1, 'r')
nwbfile1 = io1.read()
timeseries_1 = nwbfile1.get_acquisition('test_timeseries1')
timeseries_1_data = timeseries_1.data
```

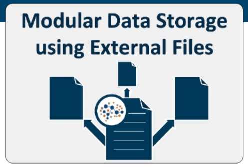
3. Add the data to a new `TimeSeries`

```
test_ts4 = TimeSeries(name='test_timeseries4',
                      data=timeseries_1_data,   # <-------
                      unit='SIunit',
                      timestamps=timestamps)
nwbfile4.add_acquisition(test_ts4)
```

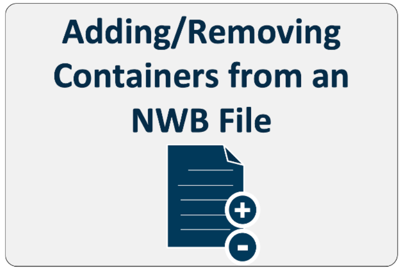
4. Write the data and specify default link behavior:
True = link (default), False = copy

```
io4 = NWBHDF5IO(filename4, 'w')
io4.write(nwbfile4, link_data=True)
io4.close()
```

Tips



- Caution! External links can become stale/break
  - – If linked files are modified, e.g. renamed, moved, access permissions changed, or data deleted within file
- When sharing data, you may want to resolve external links to merge data into a single file:
  - Option 1: **h5copy –f ext –i <input.nwb> -o <output.nwb>**
    - https://portal.hdfgroup.org/display/HDF5/h5copy
  - Option 2: **h5py.File.copy(…)**
  - Option 3: **NWBHDF5IO.export**



  - Option 4: Zip/tar all linked files together, which will maintain relative paths

Questions?