Accessing ESRF Data

ESRF Software Group

Summary

- Introduction to ESRF data formats
    - SPEC file format (aka specfile)
    - ESRF Data Format (aka EDF)
    - HDF5
    - NeXus Convention
- Available libraries
    - specfile
    - EDFFile and FabIO
    - h5py
    - sps (SPEC shared memory)

SPEC file format

Until recently SPEC was the only acquisition sequencer used at the ESRF

- Its default data format is ASCII based
- SPEC provides functions to write other formats like EDF and HDF5

SPEC file layout

A complete SPEC file contains one file header and one or several scan headers

File header is introduced by the characters #F

Scan header starts by the characters #S and is preceded by a newline character : \n

Every line in the file ends by \n (newline character)

WARNING: No effort is made nor it is intended to be made to support \r\n ending

The SPEC file header

```
#F /mntdirect/_data_id03_inhouse/2006/Jul06/balmes/CuZnO/CuZnO_2.spec
#E 1145450191
#D Sat Jul 22 13:05:06 2006
#C sixcvertical  User = opid03
#O0  Delta  Theta  Chi  Phi  mu  Gamma  dummy  Mutrans
#O1  bver  bhor  xt  yt  zax  a2theta  ath  atran
#O2  delsl  gamsl  fs3  fs4  pssvo  pssvg  pssho  psshg
#O3  eh1_ao1  eh1_ao2
```

SPEC supports a space in names -> **Separation between names is two spaces.**

#F -> Original name of the file when generated

#E -> EPOCH

#D -> Date

#C -> Comment line, they can be anywhere in the file

#O... -> Names of all the motors in the SPEC session (in this case named sixcvertical

The SPEC scan header

```
#S 1  ascan  zax -0.73 -0.23  20 1
#D Sat Jul 22 13:11:07 2006
#T 1  (Seconds)
#G0 13 0 1 0.0007850334972 0.0006321396896 0.9999994921 0 0 0 0 0 0 600 0 0 1 143.6
#G1 3.25 3.25 5.207 90 90 120 2.232368448 2.232368448 1.206680489 90 90 60 1 1 2 -1 2 2 26.132 7.41 -88.96 1.11 1.000012861 15.19 26.06 67.355 -88.96 1.11 1.000012861 15.11 0.723353 0.723353
#G3 0.06337923671 0.027529133 1.206191273 -1.43467075 0.7633438883 0.02401568018 -1.709143587 -2.097621783 0.02456954971
#G4 0 0 0 0.723353 0 0 0 0 180 88.96 -1.11 1 0 0 0 0 0 0 0 0 0 -180 -180 -180 -180 -180 -180 0
#Q 0 0 0
#P0 0 0 -88.96 1.11 0 0 0 167
#P1 0 0 -1.5 -2 -0.73 5.25 7 0
#P2 2 2 -30 -60 0.22 0.2 5.12 0.2
#P3 0 0
#N 23
#L zax  H  K  L  Epoch  Seconds  det  attn  dtime  detc  ndetc  delcnt  thcnt  mucnt  gamcnt  tthcnt  inbeam  outbeam  eh1_ai1  eh1_ai1  SRcurr  phd3  mon
```

#S -> Command used

#D -> Data

#T -> Preset acquisition time

#G -> Diffractometer geometry parameters (if any)

#Q -> HKL (if enabled)

#P -> Position of all the motors at the beginning of the scan

#N and #L -> Number and names of all the labels associated to the following data

The SPEC data block

-0.73 0 0 0 8116476 1 246553 15 0.25528 6.1599787e+11 8962185.1 0 0 0 0 0 7.4479203e-08 0 0 0 86.223145 68733 625307
-0.705 0 0 0 8116478 1 246496 15 0.25625 6.1585546e+11 8955683.1 0 0 0 0 0 7.44975e-08 0 0 0 86.219574 68767 624960
-0.68 0 0 0 8116479 1 245717 15 0.253826 6.1390918e+11 8944680.1 0 0 0 0 0 7.4302001e-08 0 0 0 86.216827 68634 623641
-0.655 0 0 0 8116481 1 245946 15 0.25388 6.1448132e+11 8931804 0 0 0 0 0 7.4408902e-08 0 0 0 86.211464 68797 624420
-0.63 0 0 0 8116483 1 245795 15 0.254015 6.1410405e+11 8936842.3 0 0 0 0 0 7.4502097e-08 0 0 0 86.209305 68716 624237
-0.605 0 0 0 8116484 1 243979 15 0.251457 6.0956689e+11 8864880.1 0 0 0 0 0 7.4416498e-08 0 0 0 86.205849 68762 624042
-0.58 0 0 0 8116486 1 241508 15 0.249518 6.0339324e+11 8760573.2 0 0 0 0 0 7.4274503e-08 0 0 0 86.201073 68876 624377
-0.555 0 0 0 8116488 1 235118 15 0.240175 5.8742821e+11 8528655.6 0 0 0 0 0 7.4428797e-08 0 0 0 86.197807 68877 624321
-0.53 0 0 0 8116490 1 211161 15 0.211069 5.2757308e+11 7672228 0 0 0 0 0 7.4130902e-08 0 0 0 86.191887 68764 623696

Immediately follows the line containing the labels (line introduced by #L

Each line corresponds to one acquisition point

Only numbers are allowed (no NaN …)

Number of values must match number of labels

Supported use

The format is simple enough to allow multiple tables inside a single file in an easy way

Several ESRF users and programs save ASCII data in that format omitting the file header

```
\n
#S 685 dscan m34 0 10 10 0.1
#N 4
#L m34  Bal  Blo  Bli
1 1 2 3
2 3 4 5
3 6 7 8
4 7 8 10
\n
#S 686 dscan m34 0 10 10 0.1
#N 3
#L m34  Blo  Bli
1 2 3
3 4 5
6 7 8
7 8 10
9 8 10
\n
#S 687 dscan m43 0 10 10 0.1
#N 3
#L m43  Blo  Bli
1 2 3
3 4 5
6 7 8
7 8 10
9 8 10
```

SPEC MCA Support

Special entries were added to specfile to handle saving of Multichannel Analyzer data

They start with the symbol @

Lines starting by @A in the data block denote the beginning of a set of MCA values

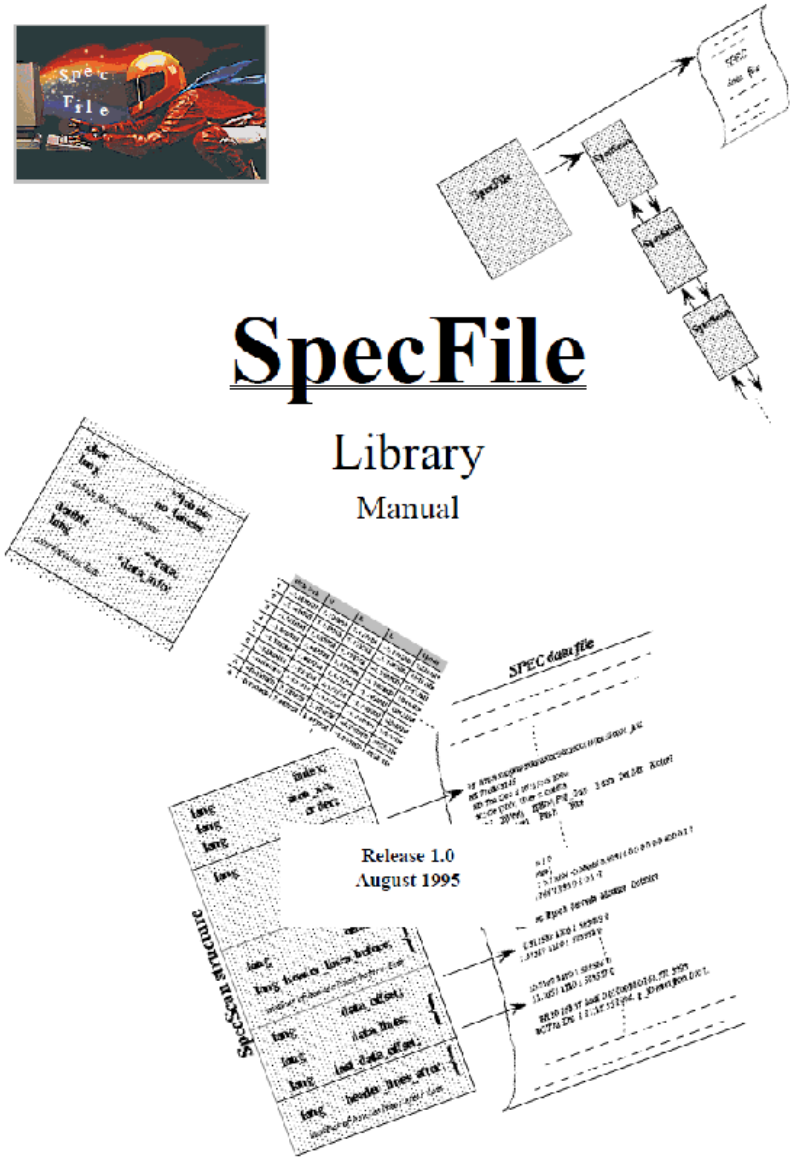
First channel is 0 unless specified by the #@CHANN key in the header

Calibration (in the form a + bx + c x^2 is found, if present, under #@CALIB

```
#F C:/titat/titat-2009/Xaloc-2013/Fluorescencia\ctrf6.mca
#D Mon Sep 01 12:57:10 2014

#S 1 Fluoscan_ctrf6-A.dat 2.1.1.2
#D Mon Sep 01 12:57:10 2014
#@MCA %16C
#@CHANN 1024 0 1023 1
#@CALIB -0.1025283 0.01976545 0
@A 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000
0.0000 0.0000 3.0000 \
12.0000 37.0000 96.0000 265.0000 522.0000 870.0000 1268.0000 1569.0000 1574.0000 1335.0000
889.0000 549.0000 278.0000 107.0000 49.0000 12.0000 \
2.0000 0.0000 0.0000 1.0000 0.0000 0.0000 0.0000 0.0000 0.0000 7.0000 23.0000 26.0000 26.0000
22.0000 30.0000 24.0000 \
29.0000 24.0000 26.0000 33.0000 24.0000 14.0000 17.0000 26.0000 21.0000 14.0000 17.0000 18.0000
21.0000 15.0000 19.0000 12.0000 \
….
1.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 0.0000 1.0000 0.0000 0.0000
1.0000 0.0000
```

The SpecFile library



It is a C library with a Python binding

Mainly developed by Vicente Rey

Maintained by V. Armando Solé

Reading SPEC files in Python

```
from PyMca5.PyMca import specfilewrapper as specfile
sf = specfile.Specfile(filename)
sf.scanno()                 #returns the number of scans
sf.allmotors()              #return the motor names
scan = sf.select('1.1')     #select the scan 1.1 if exists!!!
scan = sf[0]                #select the first scan
dir(scan)                   #list the available methods
scan.alllabels()            #list the labels
scan.data()                 #retrieve all data
scan.nbmca()                #returns the number of mca in the scan
scan.mca(1)                 #retrieve first mca
scan.header('')             #return the complete scan header
scan.header('S')            #return the specified line (#S) of the scan header
```

specfilewrapper provides access to other text based 1D formats using the same API

Exercise

- Use the file XCOM_CrossSections.dat
- That file contains cross sections as specified by the labels in the scan headers:

#S Atomic_symbol
#N 7
#L PhotonEnergy[keV] Rayleigh(coherent)[cm2/g] Compton(incoherent)[cm2/g] CoherentPlusIncoherent[cm2/g] Photoelectric[cm2/g] PairProduction[cm2/g] TotalCrossSection[cm2/g]

- Without cheating (that means not knowing the associated atomic number) write a function or script that takes as input the atomic symbol of one element and generates a plot having the following characteristics:

Title set to the supplied element
Logarithmic X axis labelled “Energy (keV)”, limits 1.0 keV and 100 keV
Logarithmic Y axis labelled “Cross Section (cm2/g)”
A curve with the coherent scattering cross section in blue
A curve with the Compton scattering cross section in green
A curve with the photoelectric cross section in red
A curve with the total cross section in black

ESRF Data Format (EDF)

We have seen how SPEC file format was “pushed” to support 1D data …

The ESRF data format was originally intended to stock images

It consists of an ASCII header followed by binary data

That structure can be repeated as many times as images are stored in the file

The EDF Header

EDF headers must have a size that is a multiple of 512 bytes

The header starts by the sequence **{\n** and ends by the sequence **}\n**

In between those delimiters, we find strings following the pattern **Key = value ;\n**

In between those delimiters, we find strings following the pattern **Key = value ;\n**

Mandatory keys:

```
HeaderID = EH:000001:000000:000000 ;
Image = 1 ;
ByteOrder = LowByteFirst ;
DataType = FloatValue ;
Dim_1 = 256 ;
Dim_2 = 256 ;
Size = 67108864
```

EDF Data types and numpy

SignedByte	int8
UnsignedByte	uint8
SignedShort	int16
UnsignedShort	uint16
SignedInteger	int32
UnsignedInteger	uint32
SignedLong	int32
UnsignedLong	uint32
Signed64	int64
Unsigned64	uint64
FloatValue	float32
DoubleValue	float64
QuadrupleValue	float128

Additional conventions

BCU macros foresee additional keys for storing positioners and counters in the header

counter_pos = counter_value1 counter_value2 counter_value3 counter_value4 … ;\n
counter_mne = counter_name1 counter_name2 counter_name3 counter_name4 … ;\n
motor_pos = motor_value1 motor_value2 motor_value3 motor_value4 … ;\n
motor_mne = motor_name1 motor_name2 motor_name3 motor_name4 … ;\n

Raster scans can reduce the number of files using multiframe EDFs following a pattern

USER_DEFINED_ROOT_NAME_%04d_%04d_%04d.edf

Where the first index is the SCAN number, the second the detector number and the third the order number in the sequence of files.

Two APIs available

The EdfFile module developed by the BCU.

The FabIO package developed as part of the Fable project (originally ID11 development)

Both approaches go beyond just reading EDF files.

They read several other image formats using the same API used to read EDF files.

EdfFile

- Its most basic implementation consists on a single file reading and writing EDF
- It does not provide converters to other formats

FabIO supports a larger amount of image formats and provide converters

EdfFile usage - Reading

```
from PyMca5.PyMca import EdfFile
edf= EdfFile.EdfFile(filename, access='rb')
edf.GetNumImages()          #get the number of images in file
data = edf.GetData(0)       #reads the first image
info = edf.GetHeader(0)     #reads the first header
dir(edf)                    #list the available methods
```

EdfFile provides read access to image formats like ADSC, MarCCD, PilatusCBF and uncompressed TIFF files.

EdfFile usage - Writing

```
from PyMca5.PyMca import EdfFile
import numpy
data = numpy.arange(10000.)              #we need some data to be written
data.shape =100, 100
info = {}                                #provide some keywords
info ['Title''] = 'Test Image 1'
edf= EdfFile.EdfFile(new_filename,  access='a+')
edf.WriteImage(info, data)               #writes the first image
info ['Title''] = 'Test Image 2'
edf.WriteImage(info, data * 2, Append=1) #writes a second image
```

FabIO

FabIO - Usage

```
import fabio                          # http://fabio.readthedocs.org

edf= fabio.open(filename)
edf.data                              #contains the image
edf.header                            #contains the header

new_edf = fabio.edfimage.edfimage(data=numpy.random.random(10, 10),
                                  header={'origin':'random'})
new_edf.write("new.edf")              #write new edf file

cbf = edf.convert('cbf')              #convert  data
cbf.write(ouput_filename)             #write image in CBF format
```

In addition to ESRF formats, FabIO supports image formats from most manufacturers:
Mar, Rayonix, Bruker, Dectris, ADSC, Rigaku, Oxford, General Electric, …

A complete description is available at:
http://dx.doi.org/10.1107/S0021889813000150

Finding your way

EdfFile and FabIO try to support *all* EDF files:

- They cannot make assumptions about the header size
- They have to parse the header

You *can* be faster if you *know* what you are dealing with:

- skipping the header block
- reading the data block
- interpreting the data block

Unpacking binary data

Using numpy:

```
import numpy

data = numpy.fromstring(string_containing_the_data, dtype)

data.byteswap() if reading machine has different endiannes than writing machine
```

Unpacking binary data

Using struct:

```
import struct
```

struct.unpack(*fmt*, *string*)

Unpack the string (presumably packed by pack(fmt, ...)) according to the given format. The result is a tuple even if it contains exactly one item. The string must contain exactly the amount of data required by the format (len(string) must equal calcsize(fmt)).

Detailed information in

https://docs.python.org/2/library/struct.html
https://docs.python.org/3/library/struct.html

Exercise

Take the file medipix.edf

- Assume header size of 1024 * 3 = 3072 bytes
- Assume an EDF data type UnsignedShort
- Assume a single image in the file
- Assume a shape of 256 rows x 256 columns

- Open the file (hint ‘rb’ mode)
- Skip the header using seek
- Read the data block
- Convert it to a numpy array of the appropriate type
- Compare the result with that obtained using EdfFile or FabIO

Introduction to HDF5

ESRF Software Group

ESRF current situation: SPEC File Format

Advantages

- Simplicity (multiple column ASCII)
- Widespread
- Counters, Motors and MCA in same file

Disadvantages

- Not suited to large datasets (images)

ESRF current situation: ESRF Data Format

Advantages

- Suited to images

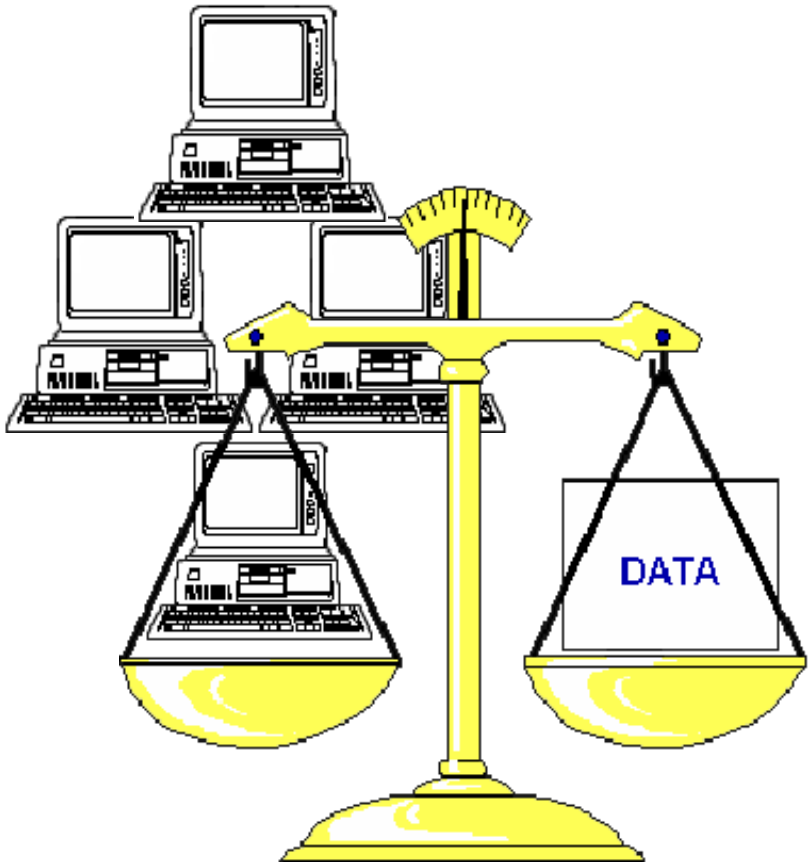
Disadvantages

- Not widespread (basically ESRF)
- Incomplete « official » metadata
- Not efficient for large datasets (parse N headers to read image N)

Faster and larger detectors

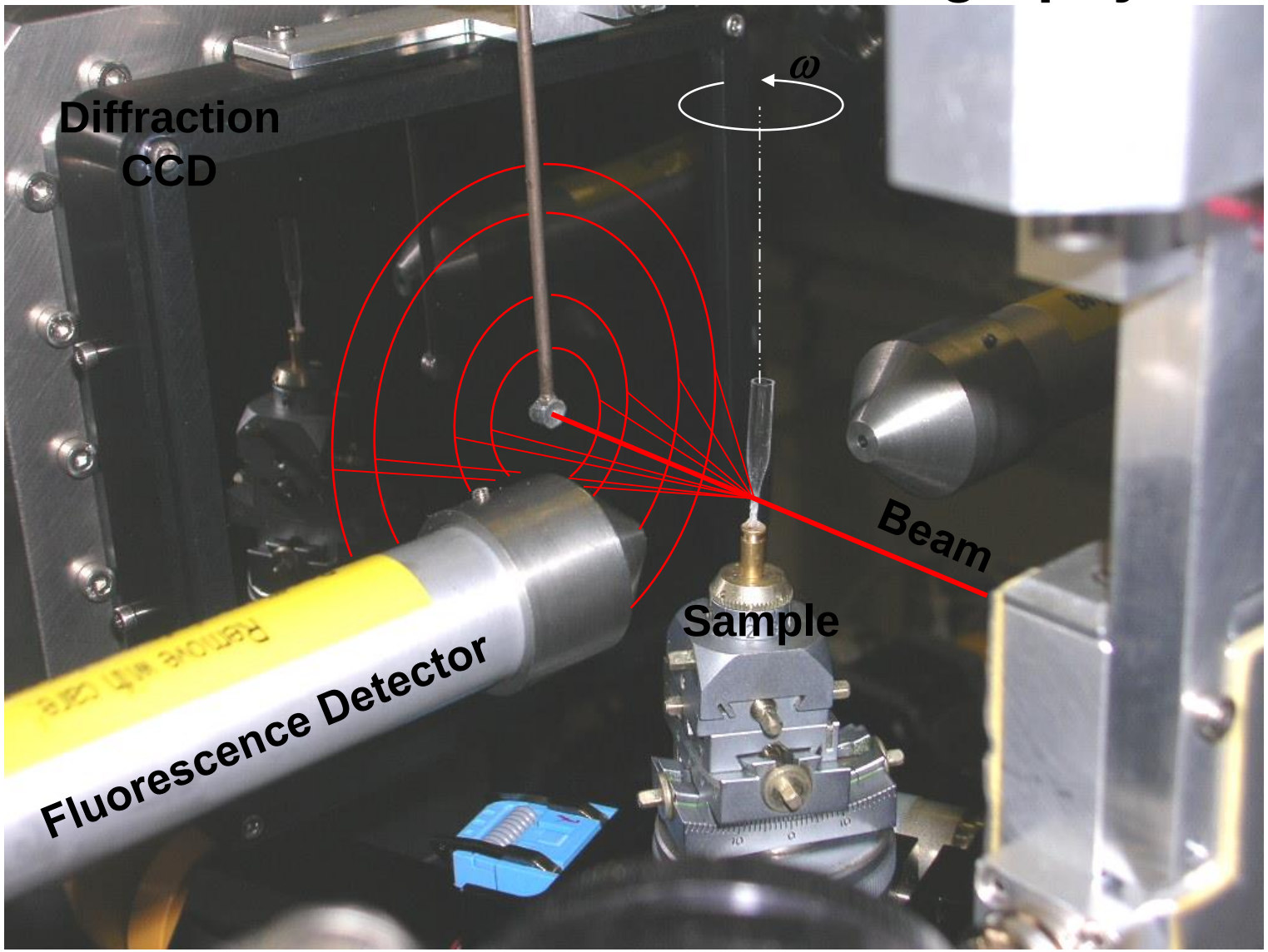
Multi-technique experiments

Huge data sets



Users face each time more difficulties to analyze data!

Fluorescence-Diffraction Tomography



Data format issues

- Multiple data formats
  - Diffraction images in EDF or MarCCD format
  - Fluorescence data in EDF or SPEC file format
  - Scan information in SPEC file format
  - Result of azimuthal integration in Fit2D .chi format

Lesson learned:

Try to avoid inventing a new data format. Use an existing one

Needs

Efficient format to store different data types

Keep together motors, counters, images, mca, …

Compression support

Editable

Widespread support

Efficient and easy access to the data for visualization and analysis

A slide from The HDF Group

What is HDF5?

HDF stands for Hierarchical Data Format

- A file format for managing any kind of data
  - http://www.hdfgroup.org/HDF5/doc/H5.format.html
- Software system to manage data in the format

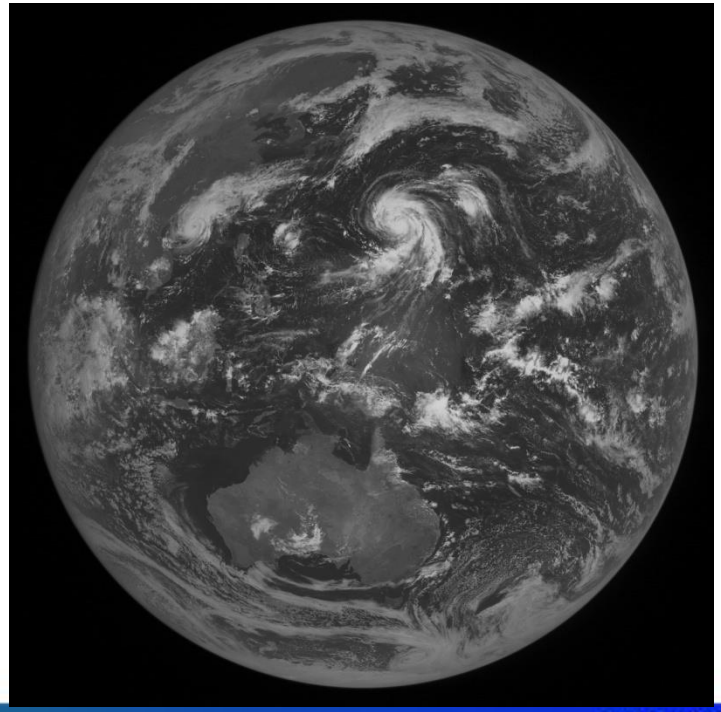
- Designed for high volume or complex data
- Designed for every size and type of system

Who uses HDF5?

- Applications that deal with big or complex data
- Over 200 different types of apps
- 2+million product users world-wide
- Academia, government agencies, industry

NASA EOS remote sense data

- HDF format is the standard file format for storing data from NASA's Earth Observing System (EOS) mission.
- Petabytes of data stored in HDF and HDF5 to support the Global Climate Change Research Program.



Outstanding Features of HDF5

- Can store all kinds of data in a variety of ways
- Runs on most systems
- Lots of tools to access data
- Long term format support (HDF-EOS, CGNS)
- Library and format emphasis on I/O efficiency and different kinds of storage

HDF5
Basic File Structure

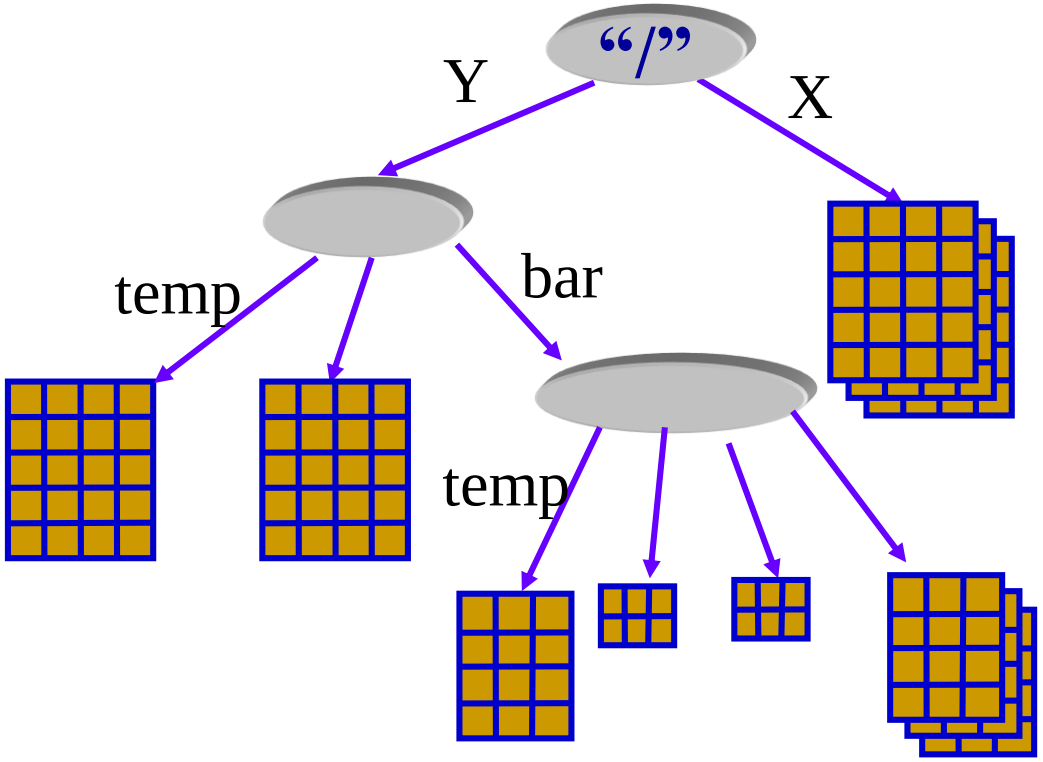
HDF5 model

- Groups – provide structure among objects
- Datasets – where the primary data goes
  - Rich set of datatype options
  - Flexible, efficient storage and I/O
- Attributes, for metadata annotations
- Links – point to other groups or datasets
  - Hard, soft and external flavors

Everything else is built essentially from these parts

Path to HDF5 object in a file

/ (root)
/X
/Y
/Y/temp
/Y/bar/temp



HDF5 Dataset

Metadata

Dataspace

Rank: 3

Dimensions:
- Dim_1 = 4
- Dim_2 = 5
- Dim_3 = 7

Datatype

IEEE 32-bit float

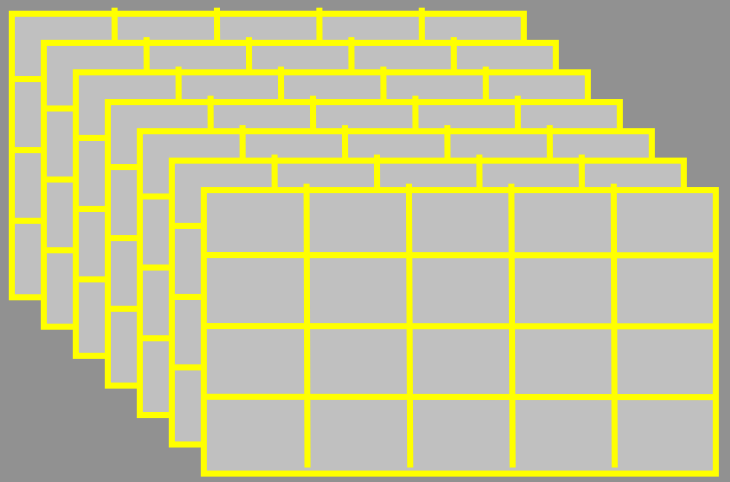
Storage info

- Chunked
- Compressed

Attributes

- Time = 32.4
- Pressure = 987
- Temp = 56

Data

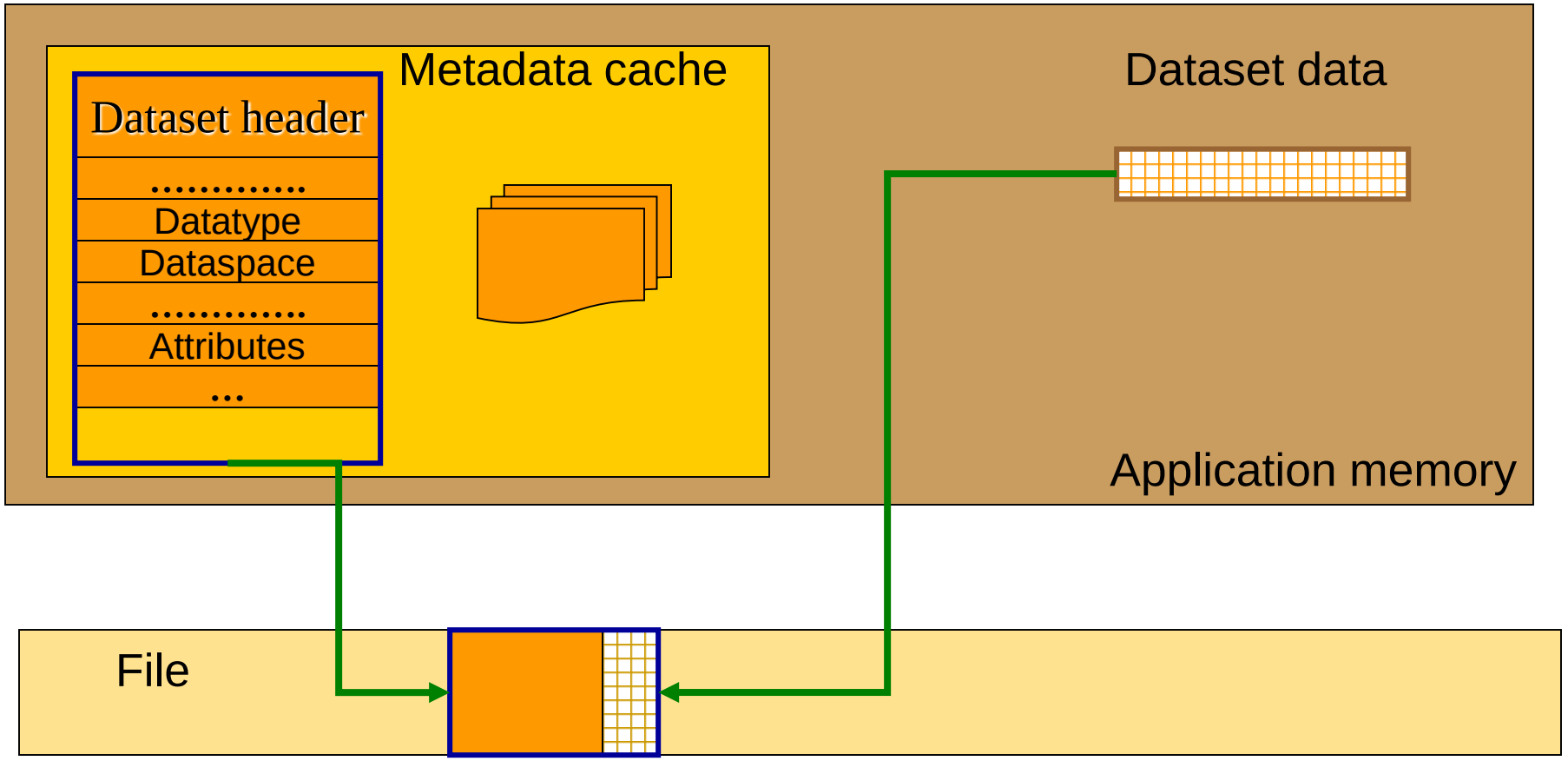


HDF5 dataset storage layouts

- Compact
- Contiguous
- Chunked

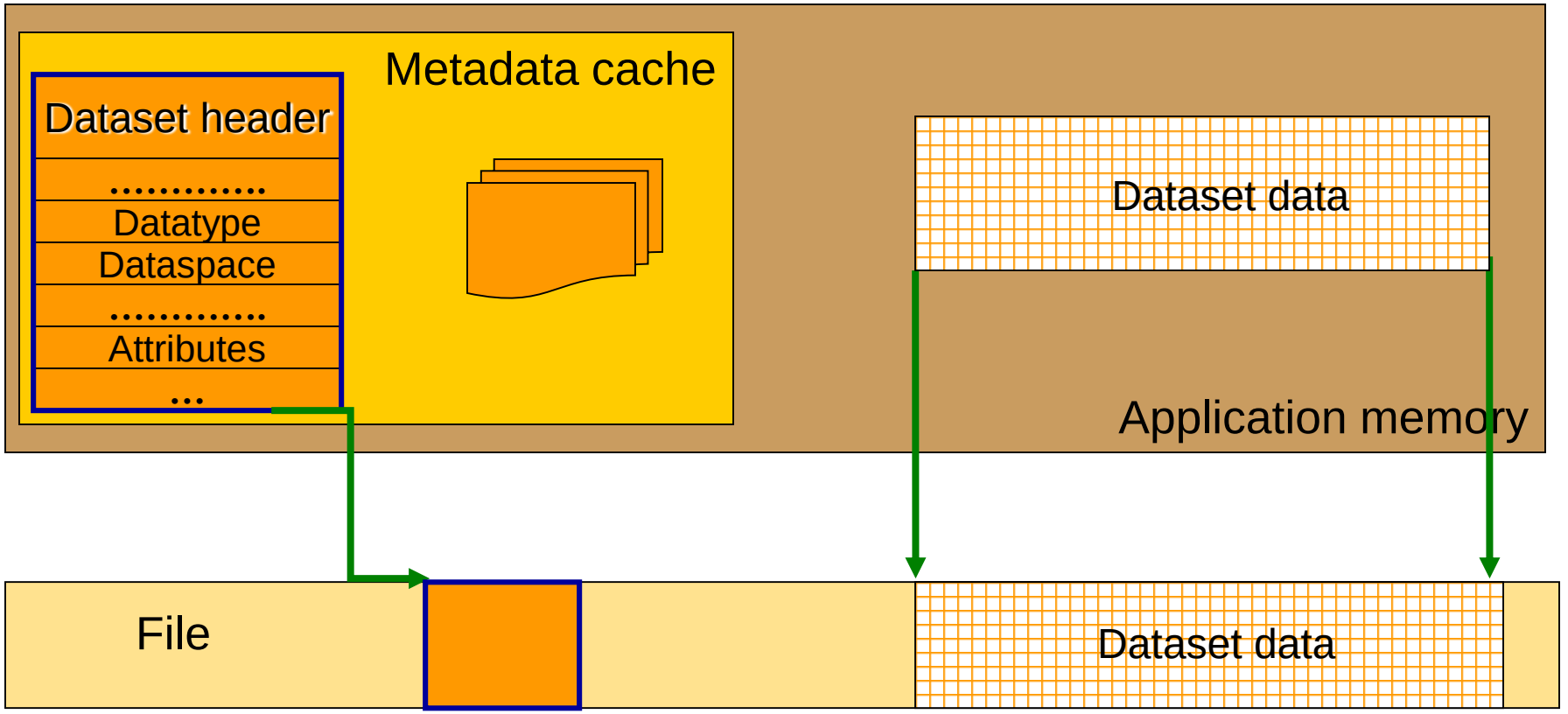
Compact storage layout

- Dataset data and metadata stored together in the object header



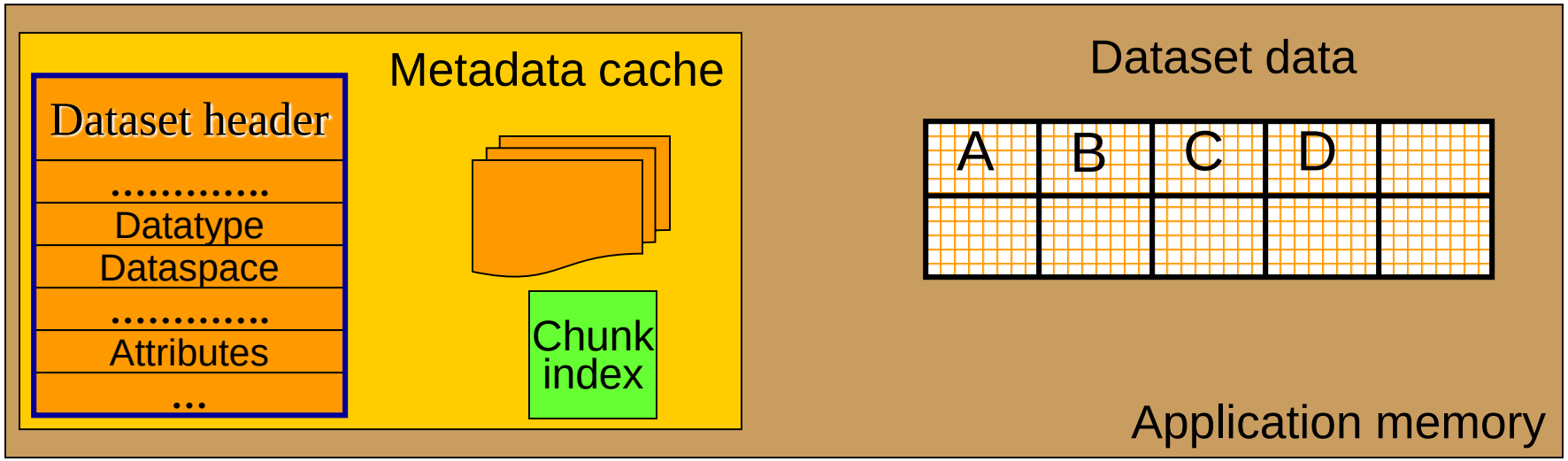
Contiguous storage layout

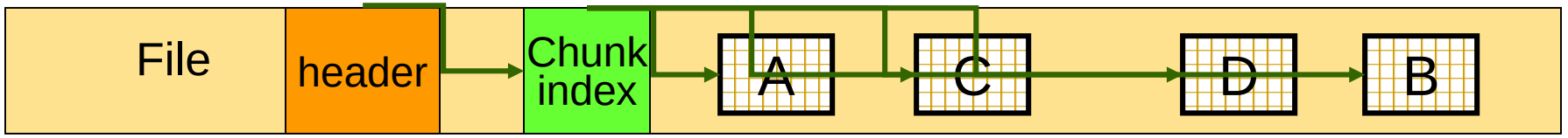
- Metadata header separate from dataset data
- Data stored in one contiguous block in HDF5 file



Chunked storage layout

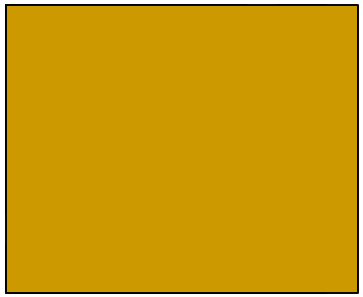
- Dataset data divided into equal sized blocks (chunks)
- Each chunk stored separately as a contiguous block in HDF5 file





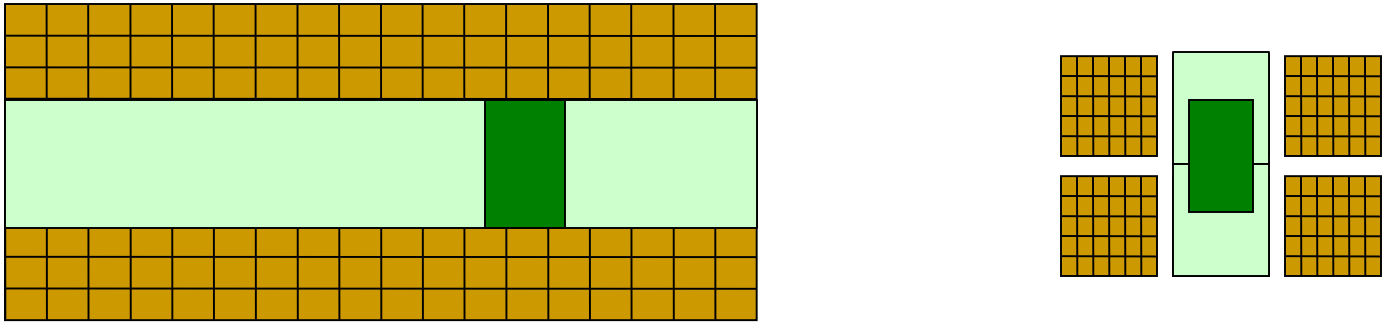
Why HDF5 Chunking?

- 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



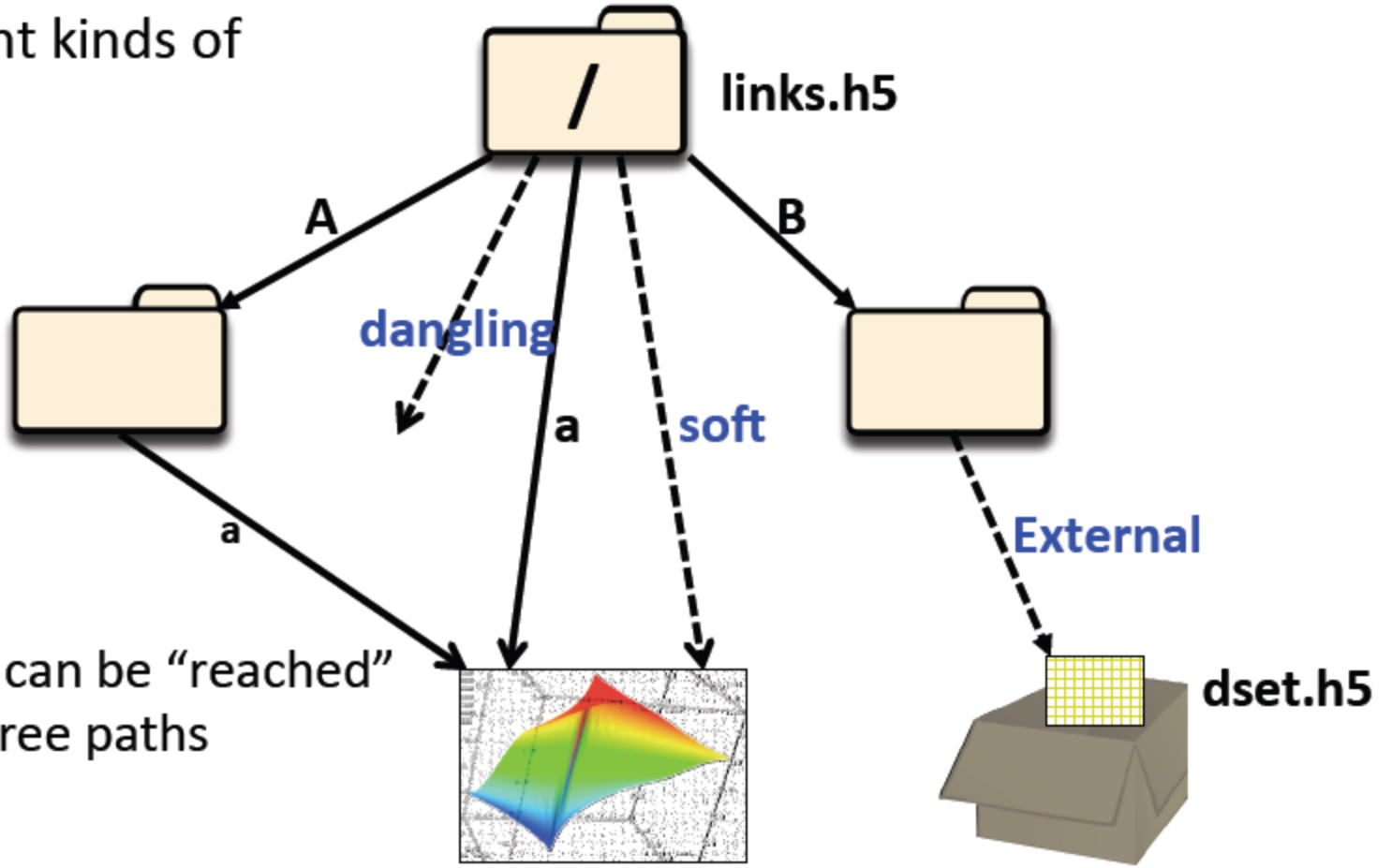
Only two chunks are involved in I/O

HDF5 Attribute

- Attribute – data of the form “name = value”, attached to an object by application
- Operations similar to dataset operations, but …
  - Not extendible
  - No compression or partial I/O
- Can be overwritten, deleted, added during the “life” of a dataset or a group (but not to a link)

HDF5 links

Different kinds of links



Dataset can be "reached" using three paths

/A/a

/a

/soft

Dataset is in a **different** file

HDF5 links

- Name
  - Example: “A”, “B”, “a”, “dangling”, “soft”
  - Unique within a group; “/” are not allowed in names
- Type
  - Hard Link
    - Value is object’s address in a file
    - Created automatically when object is created
    - Can be added to point to existing object
  - Soft Link
    - Value is a string , for example, “/A/a”, but can be anything
    - Use to create aliases

HDF5 links

- Type
  - External Link
    - Value is a pair of strings , for example, (“dset.h5”, “dset”)
    - Use to access data in other HDF5 files
    - HDF5 1.8.7 introduced caching of files opened via external links `H5Pset_elink_file_cache_size`

Discovering HDF5 File Structure

- HDF5 provides C and Fortran 2003 APIs for recursive and non-recursive iterations over the groups and attributes
  - H5Ovisit and H5Literate (H5Giterate)
  - H5Aiterate
- Life is much easier with H5Py (h5_visita.py)

```
import h5py
def print_info(name, obj):
    print name
    for name, value in obj.attrs.iteritems():
        print name+":", value
f = h5py.File('GATMO-SATMS-npp.h5', 'r+')
f.visititems(print_info)
f.close()
```

Accessing HDF5 files

There are two main packages to access HDF5 files: PyTables and h5py

At the ESRF we advocate the use of h5py:

- It is more intuitive for Python users
- It provides a direct binding to HDF5

The documentation is available at:

http://docs.h5py.org/en/latest/

Basic HDF5 access using h5py

```
>>> import numpy
>>> data = numpy.arange(10000.)
>>> data.shape = 100, 100
>>> import h5py
>>> f = h5py.File('myfirstone.h5', access='w')
>>> f['/data'] = data
>>> f.close()
>>> n = h5py.File('myfirstone.h5', access='r')
>>> data = n['/data']              #reference to the file content, no actual reading
>>> data.shape                     #shape of the data
>>> 2 * data[0, 5]                 #read and apply the operation
>>> actualData = data.value        #perform the reading and store in an array
```

Exercise

Create an HDF5 data file and try to create in it:

- An 3D array of 32-bit floats dimensions 100, 512, 1024 filled with elements in increasing order
- A dataset named “title” containing a string (not a string array)
- A dataset named single_int32 consisting on a single 32-bit integer (not an array of integers containing only one element)
- A dataset named single_int16 consisting on a single 16-bit integer

Verify the layout of your file with hdf5view or pymca. You may have surprises.

Introduction to NeXus

http://www.nexusformat.org/

HDF5

Hierarchical data format

Think about an HDF5 file as a hard disk

- You can write whatever you want to it
- It supports links, including external links
- It supports compression
- It is widely supported (Fortran, C, Java, Python, Matlab, IDL, …)
- It can be messy

What is NeXus?

According to the NeXus web pages, NeXus is

- A set of design principles
  To help people understand what is in the files
- A set of data storage objects
  To allow the design of portable analysis software
- A set of subroutines
  To make easy to write and to read NeXus files
- A scientific community
  To provide a forum to discuss ideas about data storage

Seriously, what is NeXus?

The NeXus API was able to read and to write files using HDF4, XML, HDF5, …

Most (all?) synchrotrons have made the choice of using HDF5

The evolution of HDF5 is faster than that of the NeXus API

The HDF5 API is more robust than that of NeXus

NeXus API (NAPI) has been frozen

This leads to consider NeXus **a set of design principles**

What are those principles?

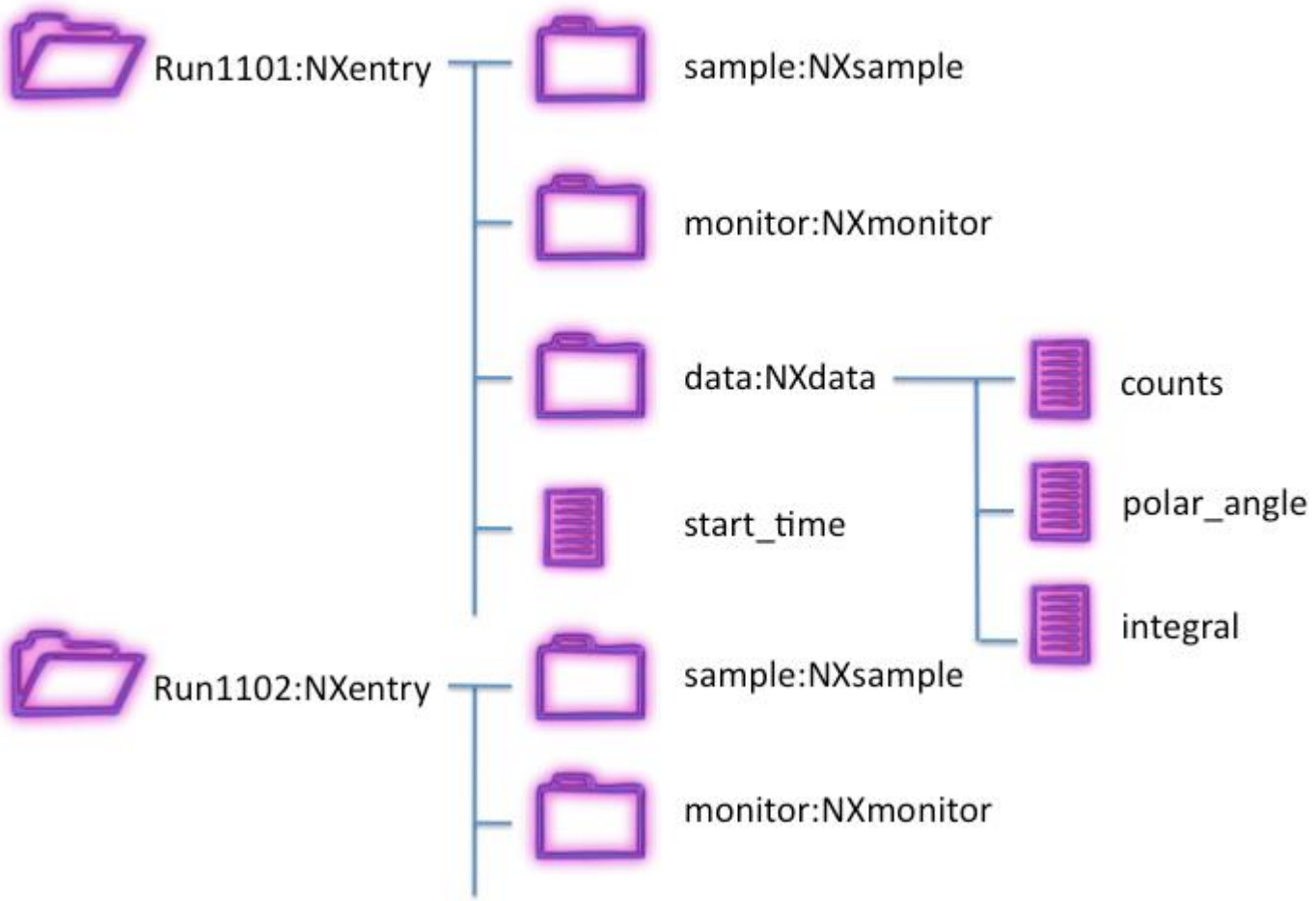
Structure you data in a set of Groups, Fields (=datasets), Attributes and Links

The expected content of a Group is defined by the attribute NX_CLASS

The name of the Groups is not imposed

There are more than 50 “classes” defined:

http://download.nexusformat.org/doc/html/classes/base_classes/index.html


Run1101:NXentry
sample:NXsample
monitor:NXmonitor
data:NXdata
counts
polar_angle
integral
start_time
Run1102:NXentry
sample:NXsample
monitor:NXmonitor

Nexus

NXroot

Top level. One per file.

NXentry

One group per measurement

NXinstrument

Describe the instrument.
Only one per NXentry

NXsample

Define the physical state of the sample during the scan

NXmonitor

Monitor data, i.e., counts, integrals, *etc*.

NXdata

The data to be plotted.
One NXdata group per plot

NXuser

Details of a user, i.e., name, affiliation, email address, *etc*

Defines a series of groups (“Directories”)

Advantages

A lot of endless discussions avoided

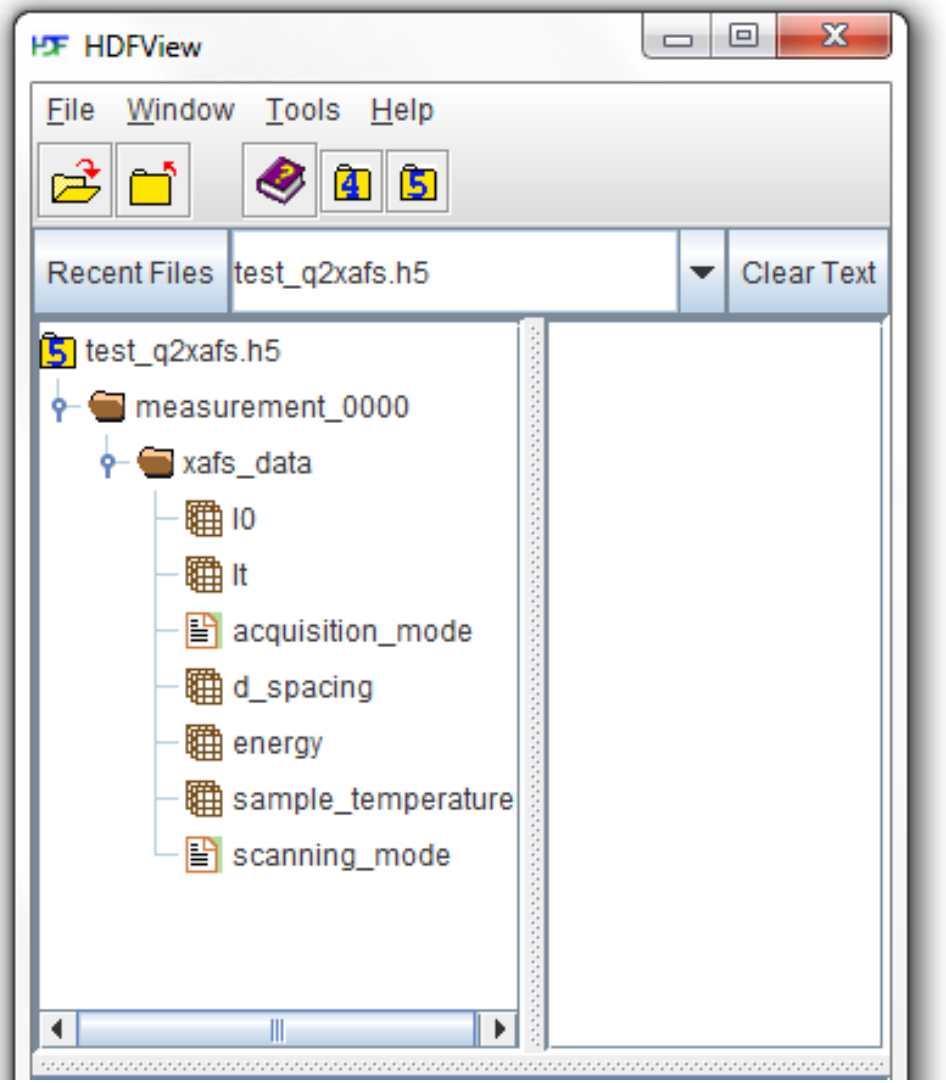
Disadvantages

A lot of endless discussions pending

Originally diffraction minded (SAXS)

No clear path for extension

HDF5 XAS example



DISCLAIMER: The above does not reflect any XAS convention agreed upon.

NeXus equivalent

```
NXroot
  entry_000 (@NXentry)
    title
    start_time
    end_time
    beamline_name (@NXinstrument)
      I0_detector(@NXdetector)
        data
      It_detector(@NXdetector)
        data
      Monochromator(@NXmonochromator)
        energy
        Si111(@NXCrystal)
          d_spacing
    sample(@NXsample)
      temperature
    monitor(@NXmonitor)
      data(link to I0_detector/data)
    xafs_data(@NXsubentry)
```

The NXsubentry would contain the actual definition as:

definition (string set to xafs for instance)
d_spacing (link to d_spacing)
sample_temperature (link to temperature)
I0 (link to I0_detector/data)
It (link to It_detector/data)
acquisition_mode (string)
scanning_mode (string)

Basically what we already had before…

Why not to force the existence of the definition while leaving the rest as optional?

Personal opinion on Nexus

It is an **instrument minded approach**

It can be very convenient for archival

It is **far from a data analysis minded approach**

Imagine having to browse three times three directory levels each time you want to retrieve a motor position, an intensity monitor and a measured spectrum

Nexus definitions can help solving this issue

A dictionary of links to datasets needed to perform a particular analysis

You can keep the freedom of HDF5 and decide up to what level you follow Nexus

NXentry simplifies handling several measurements in one file

NXdata is ideal for data exchange and default measurement plots

I prefer to have all motors, counters, images, … in one group inside an NXentry

You can do whatever you want, whenever you want, through links

In other words, combine the simplicity of SPEC file overcoming its limitations

What do we have in mind?

Access all the measured items in a “measurement” group

measurement

positioners #group containing the equivalent of SPEC motor positions

motor0

motor1

….

scalar_data #group containing the equivalent of SPEC counter values

counter0

counter1

…

mca_device0

mca_data0 #shape (npoints, nchannels) @interpretation=“spectrum”

mca_data0_info # group containing information related to mca_data0, typically a link to the device description in the NXinstrument group)

camera_device0

image_data0 #shape (npoints, nrows, ncolumns) @interpretation=“image”

image_data0_info #group containing information related to image_data0, typically a link to the device description in the NXinstrument group)

Access all the measured items in a “measurement” group

GOALS:

- Provide quick access to everything measured
- Have as little structure as possible (it can be totally flatten)
- The ESRF Metadata Working Group expected to provide the guidelines

Portable Data Analysis

Portable data analysis *cannot* be achieved with previous examples

NeXus files from SOLEIL, DESY or DIAMOND using the same technique are not identical

Portable data analysis is achieved via “Application Definitions” in NXsubentry

Analysis codes just have to explore the HDF5 looking for entries containing the relevant information for the analysis

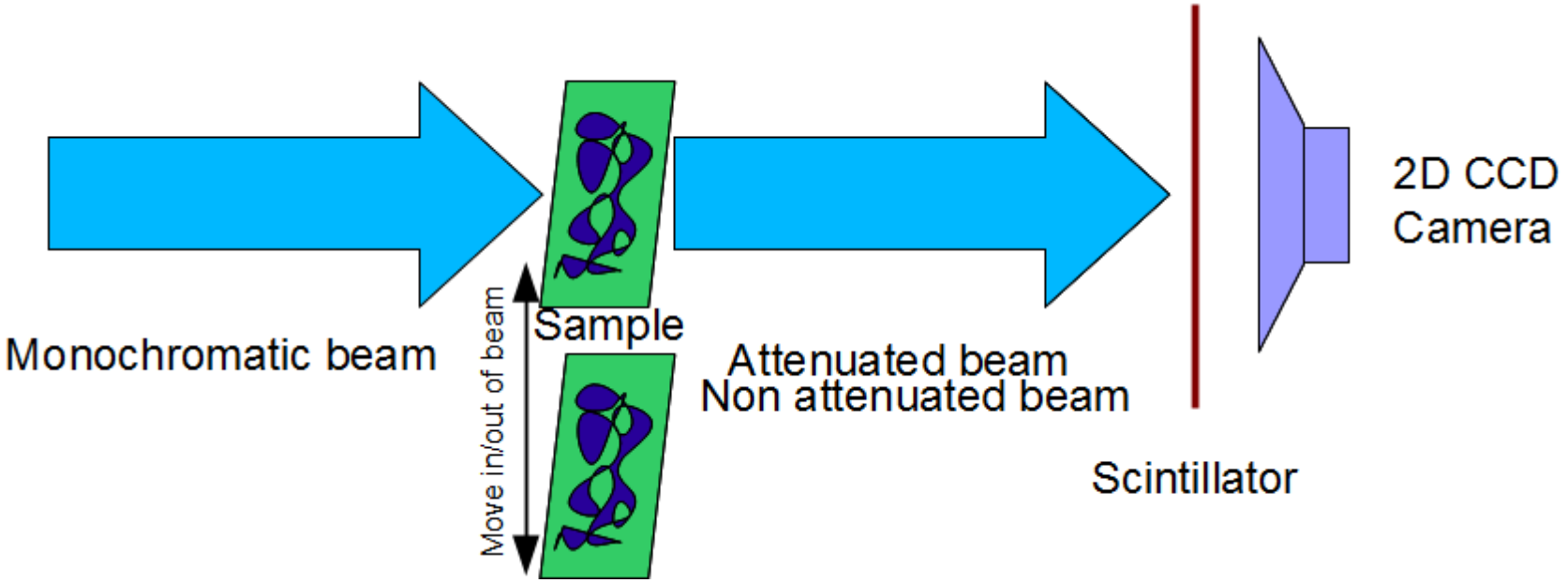
Application definitions will be specified by:

- Technique communities
- Beamlines
- Analysis codes

ID21 Full Field XANES

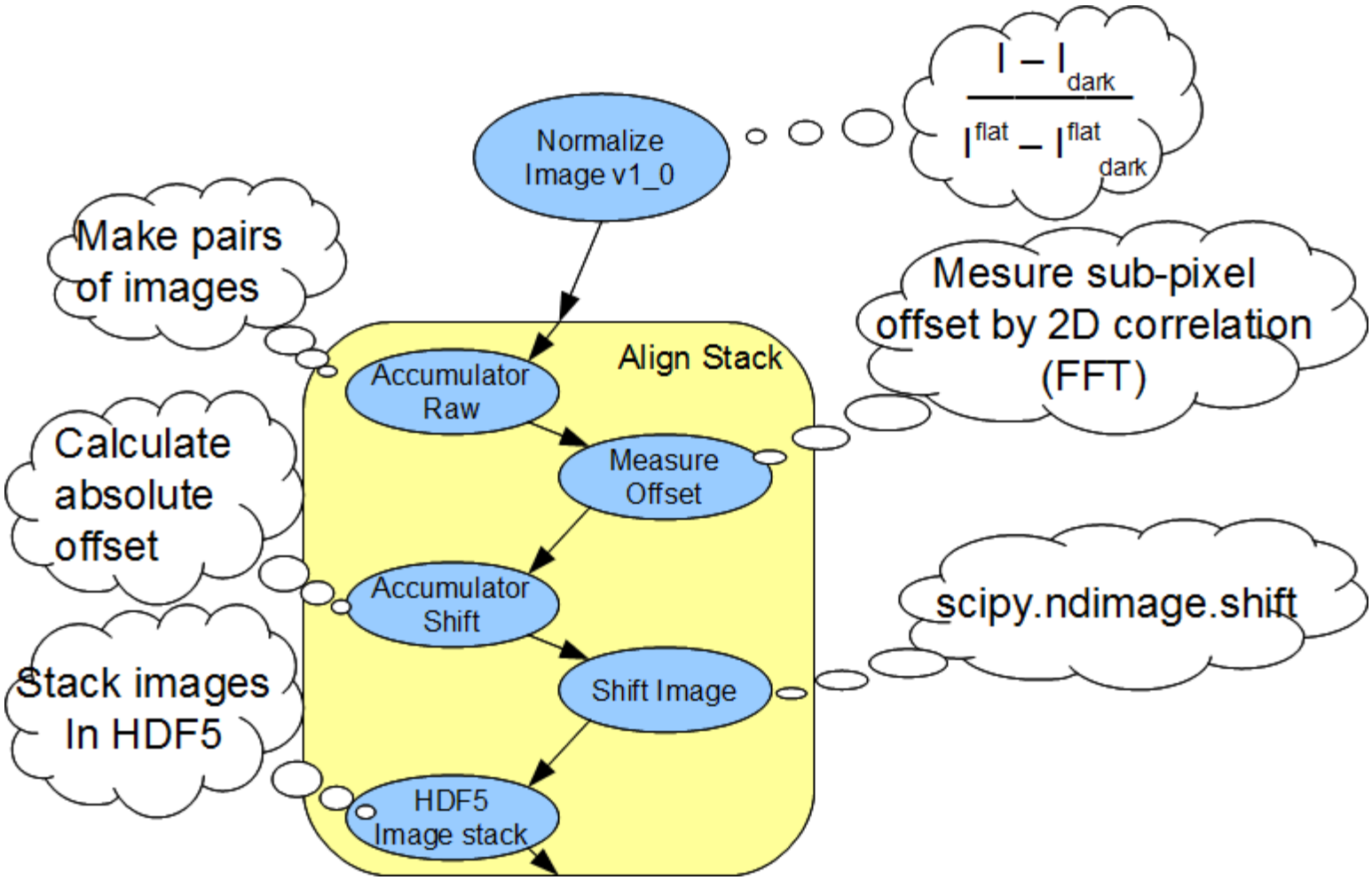
Scan in energy around an absorption edge

Spot size 1 mm
Resolution 500 nm



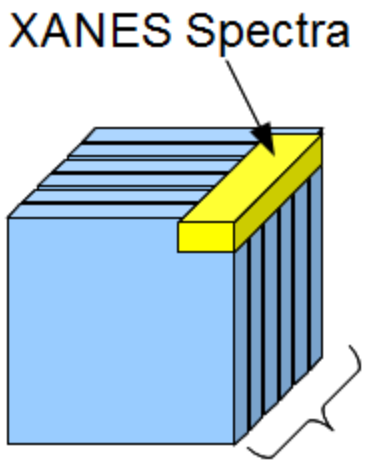
Align sample to correct submicron sample position changes

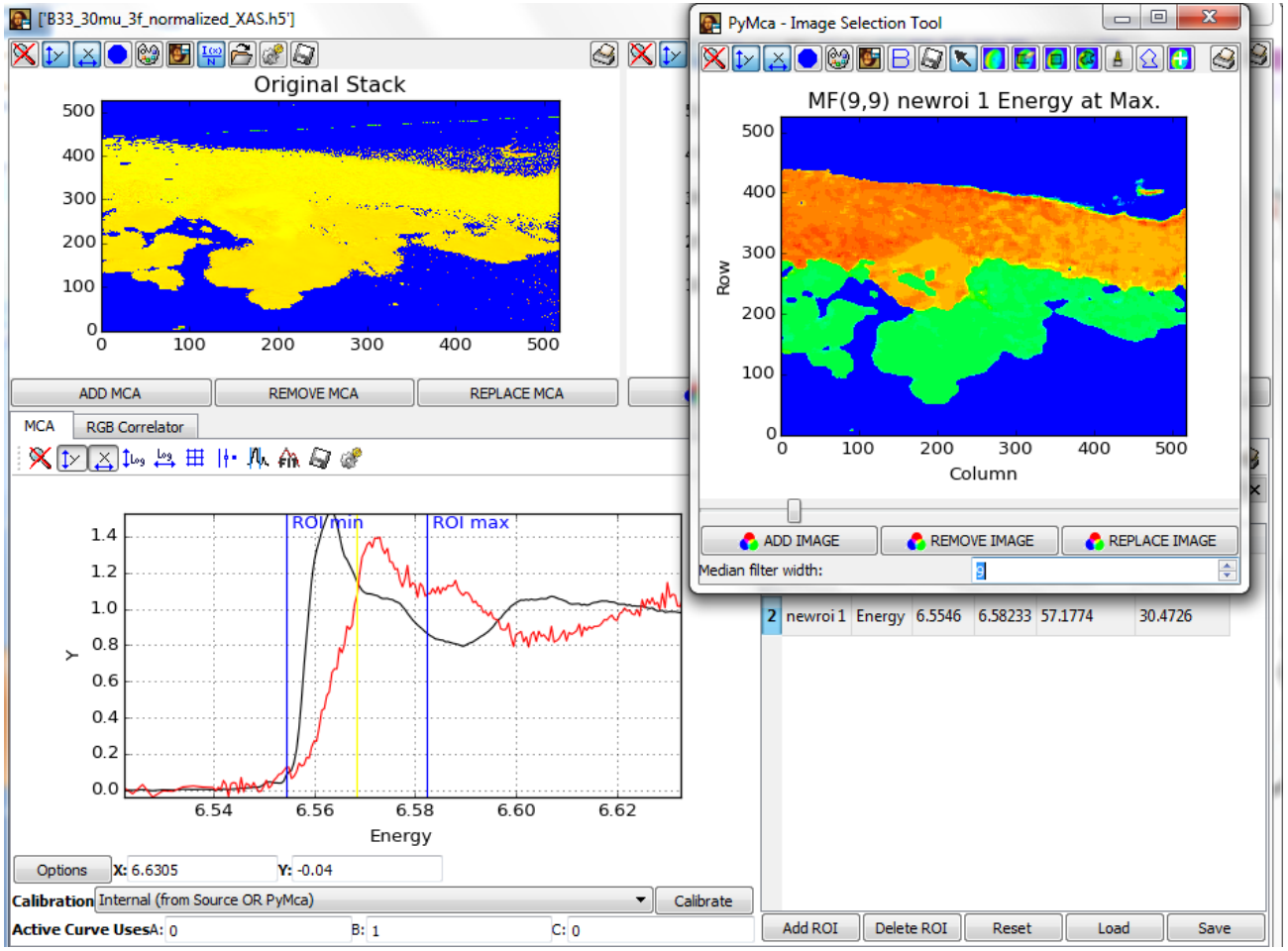
Workflow for ID21 Full Field



Multithreaded treatment therefore chunking needed

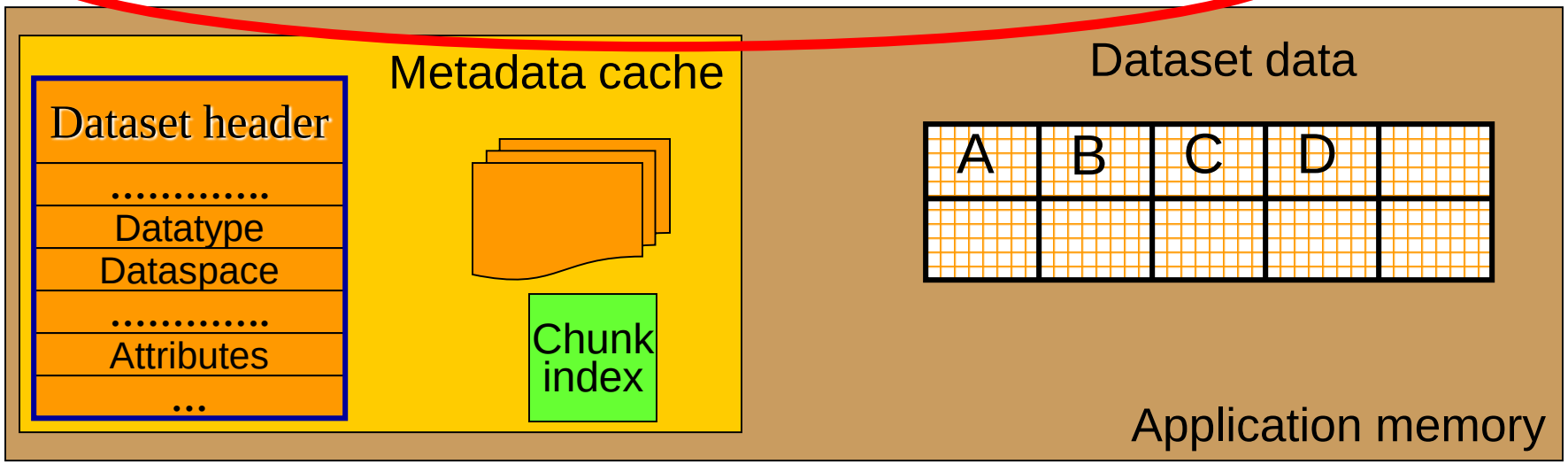
ID21 Analysis

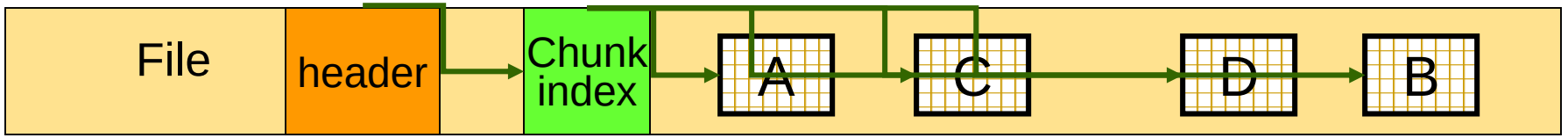




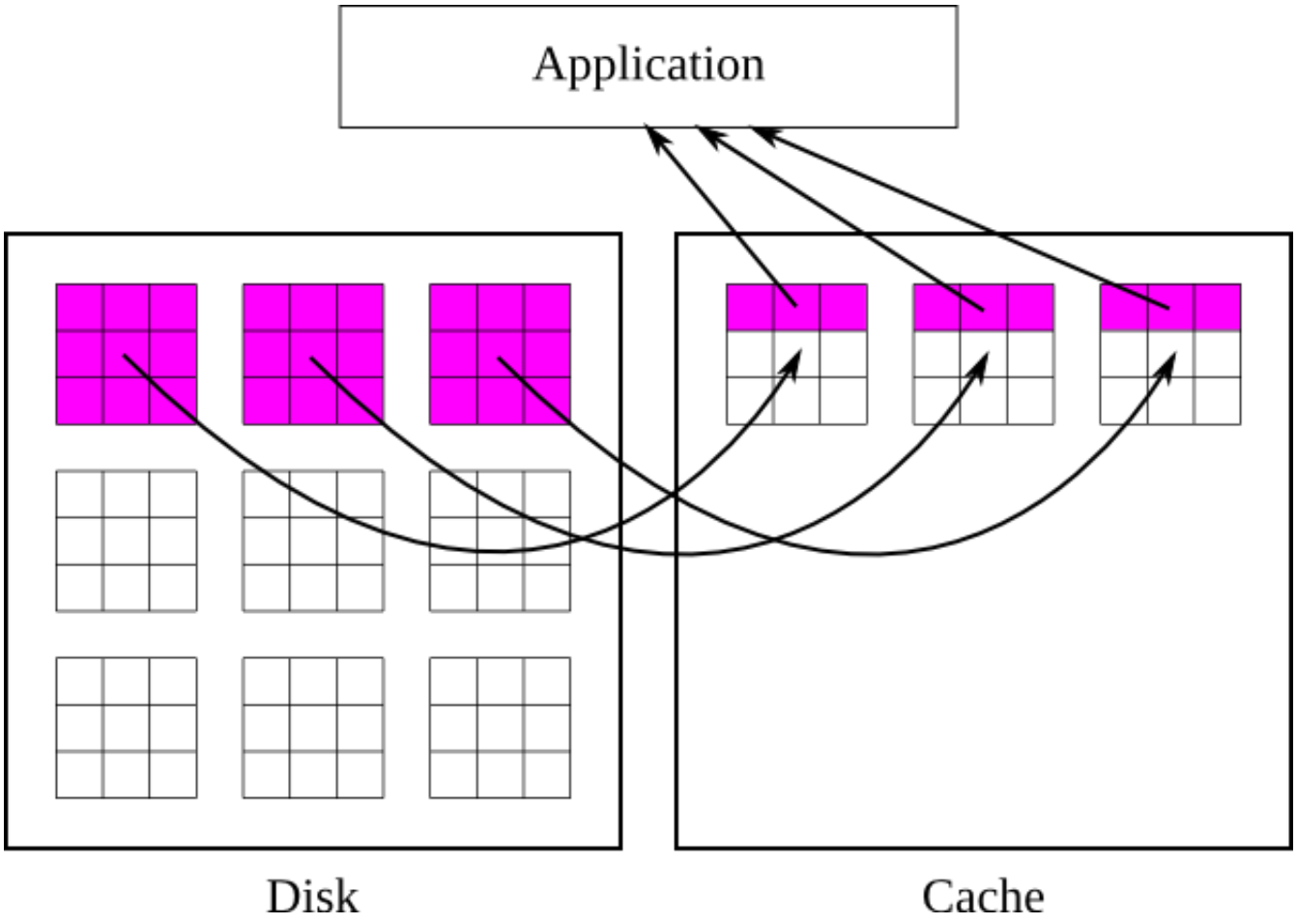
HDF5 Chunked Storage Layout

- Dataset data divided into equal sized blocks (chunks)
- Each chunk stored separately as a contiguous block in HDF5 file
- HDF5 always writes/reads the whole chunk



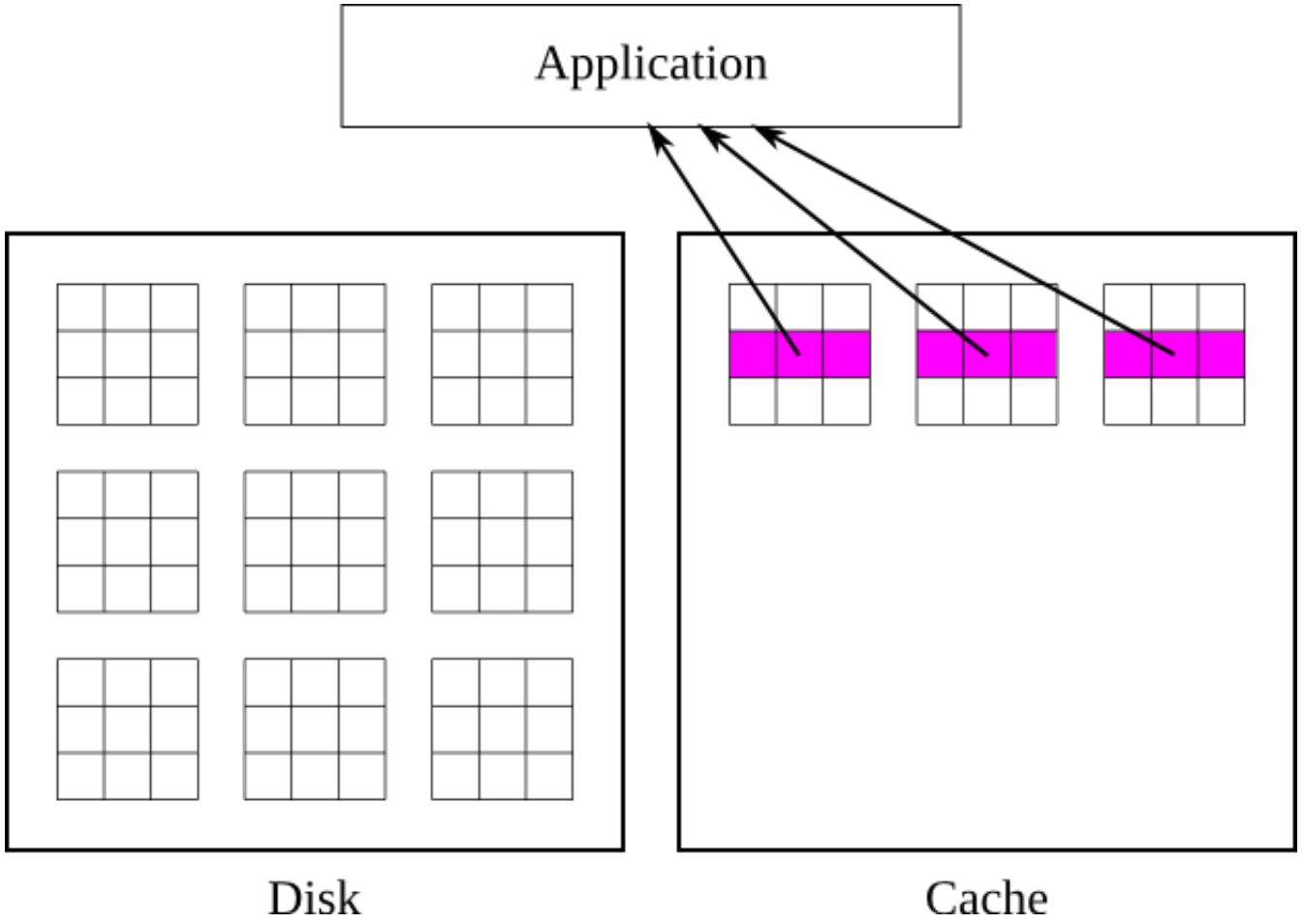


Using the chunk cache (I)



Reading from disk with chunk cache enabled

Using the chunk cache (II)

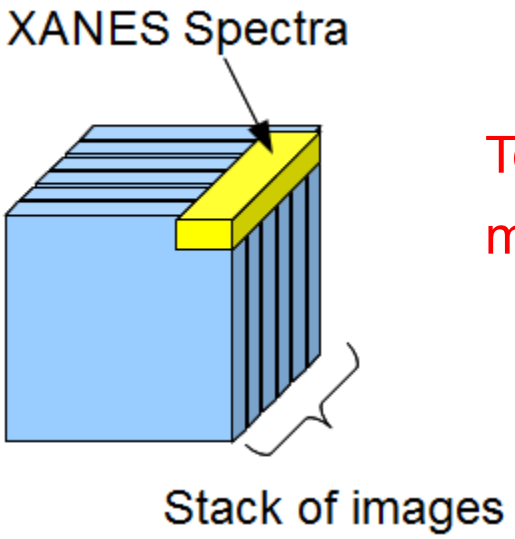


Next readout is taken directly from the cache.

This only works if the chunk cache size was large enough! (default 1MByte)

How does this affected the ID21 case?

- Dataset extended with images of 16 MBytes (n_points, 2048, 2048)
- We were using a chunk size of (1, 2048, 2048)
- The dataset had a size of 5 GBytes
- We had to use dynamic reading (32 bit systems …)



To get the spectrum associated to one pixel of our map we had to read 5 Gbytes ….

WARNING

- If NOT used appropriately (for instance not combined with the appropriate chunk cache size) can have very negative effects

MUST READ

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

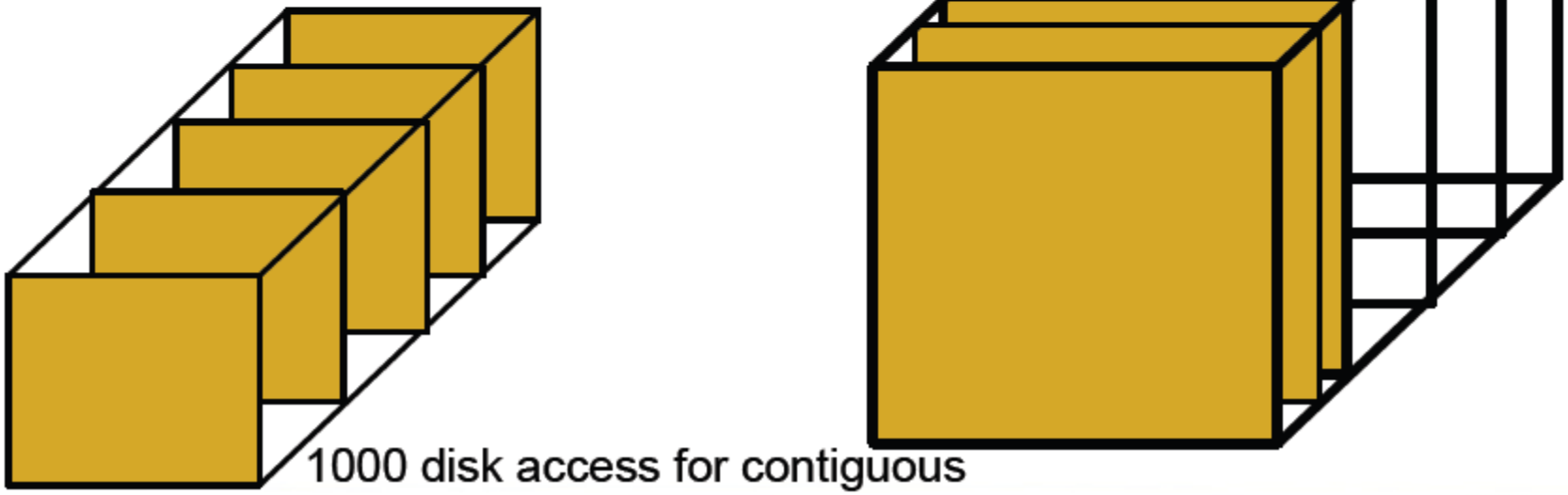
www.hdfgroup.org/HDF5/doc/Advanced/Chunking/Chunking_Tutorial_EOS13_2009.pdf

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

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 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)

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

More detailed information

New Groups can also be created using either create_group

>> g = create_group(name)

h5py group documentation

New datasets are created using either create_dataset or require_dataset

>> d = create_dataset(name, shape, dtype, chunks, compression)

h5py dataset documentation

Exercise

Generate 100 EDF files of 1024 rows and 2048 columns. The first file is filled with zeros, the second with ones, and so on.

Read those files back to a single HDF5 file following the layout (no compression):

mydatacontainer
  mydata[100, nrows, ncolumns] # add attribute @interpretation=“image”

Create a second file following the layout (with gzip compression):

mydatacontainer
  mydata[100, nrows, ncolumns] # add attribute @interpretation=“image”

The control perspective

Advantages

No need to write sequentially

Disadvantages

A new API to learn (complex at low level)

Have to provide customization options (chunking) to allow tailoring to a particular analysis

The Beamline Scientist perspective

Advantages

All the information together and easily accessible

Possibility to use applications of other facilities

Disadvantages

To get used to other tools (or adapt existing tools!)

The Facility User perspective

Advantages

High level APIs available (Python, MATLAB, IDL, …)

Much easier to read than ASCII and EDF files

Ideal exchange format for large datasets

Disadvantages

Data format conversion tools needed during some time for several techniques.

Conclusion

Scientists are those most reluctant to accept a new data format but the ones having the harder time will be the acquisition software people.

HDF5 for data analysis is simply excellent

Tools for exploring HDF5 files and provide data diagnosis at the beamlines already exist both at the ESRF and at other facilities

NeXus offers little to HDF5 itself but some ideas are good and worth to be kept. Labs are giving up the NeXus API in favor of the direct use of the HDF5 API thus reducing NeXus to a convention.

The data have to be written keeping in mind how they will be analyzed