RFC: HDF5 Virtual Dataset (Draft)

Elena Pourmal

Quincey Koziol

This document introduces Virtual Datasets for HDF5 and summarizes the use cases, requirements, and programming model for them.

# Introduction

There have been several requests to The HDF Group to implement a new feature called “Virtual Datasets” (VDS). While requests have come from different communities, such as the synchrotron community and earth sciences community, they have a common requirement: to view data stored across HDF5 files as a single unified HDF5 dataset on which normal I/O operations can be performed.

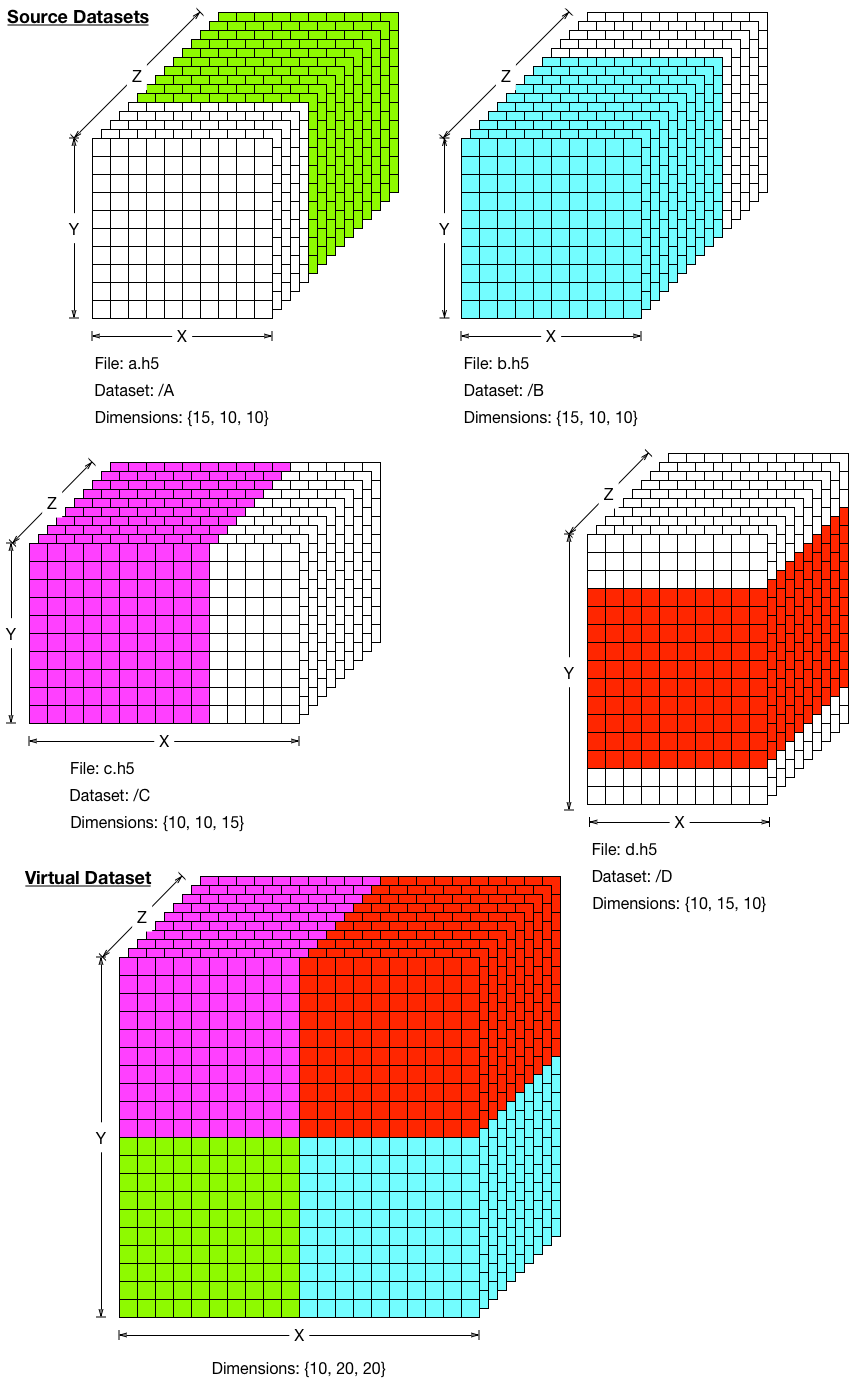
This RFC formalizes the notion of the “Virtual Dataset”, and documents its requirements and use cases.

# HDF5 Virtual Datasets

Our goal is to introduce and implement a new way of constructing and accessing collections of HDF5 datasets in the same way as if it was stored in a single dataset in one HDF5 dataset. An HDF5 Virtual Dataset is an HDF5 dataset with the following properties:

1. Data elements for a virtual dataset are stored in source datasets in one or more source HDF5 files.
2. There is a mapping from a set of elements in each source dataset to a set of elements in the virtual dataset.

For example, the diagram below shows mapping portions of four source datasets to a single virtual dataset:



After a virtual dataset has been created, its data elements can be accessed using typical HDF5 API routines (i.e. H5Dread and H5Dwrite), with the actual data elements being retrieved from the corresponding locations in source datasets.

Furthermore, virtual datasets can have unlimited dimensions and map to source datasets with unlimited dimensions, allowing a virtual dataset to grow over time, as its underlying source datasets change in size. Extending the idea of source datasets growing over time (and therefore the virtual dataset growing) to the realm of simultaneous writing to and reading from those datasets (i.e. the ‘SWMR’ feature in HDF5) shows that a virtual dataset can be employed to create a single, coherent view of a set of source datasets that are being concurrently written while the virtual dataset is read from, something otherwise not possible with HDF5.

# Use cases

In this section we discuss several use cases for VDS.

## DLS use case “Excalibur”

The EXCALIBUR detector is described in the section 3.1 of [1]. A series of images taken from the detector are stored in six HDF5 files as shown in Figure 1.

At every time step, data for a 2D image is divided into six sub-images *A*, *B*, *C*, *D*, *E* and *F*. The sub-images *A*, *C* and *E* have size k x M, the sub-images *B*, *D* and *F* have size n x M. The size of the full image is (3k +3n) x M.

Time series for each sub-image *A*, *B*, *C*, *D*, *E* and *F* is stored in a 3D dataset in the corresponding HDF5 file; for example, time series for sub-images *A* are stored in a.h5 in the dataset A, time series for sub-images *B* are stored in b.h5 in the datasets B, etc.

Each 3D source dataset may have its own chunking and compression properties. For example, datasets A, C, and E have chunk size (1 x k x M), while datasets B, D, and F have chunk size (1 x n x M).

The slowest changing dimension of each 3D dataset corresponds to the “time” axis while a plane orthogonal to the axis represents the sub-image at the given time step. The sub-images are written in the order they are taken. Independent processes write sub-images and there is no synchronization between the processes. Therefore, at any particular moment the number of the planes in the sub-image datasets may be different.

If an application needs to read a full image at some particular time step, it can read corresponding sub-images and then combine them in an application buffer, but the application (or user) has to know how to find sub-images in the HDF5 files.

A virtual dataset VDS in the file VDS.h5 (see Figure 1) provides a seamless access to the sub-images without explicit knowledge by application of where the actual data is stored. As with sub-image datasets, a VDS represents the series of full images. The slowest changing dimension of the VDS corresponds to the “time” axis, and the planes orthogonal to the axis correspond to the full images. In Figure 1 the highlighted planes in each datasets A, B, C, D, and E are sub-images of the image “stored” in the VDS at time step t2.

If some of the sub-images at a particular time step have not been written yet, for example, highlighted images in b.h5, d.h5, and f.h5, and if the second plane in VDS is read, the corresponding sub-images *B*, *D*, and *F* in the full image will be filled with the fill values or the read may fail (depending on the access properties for the I/O operation).

Excalibur use case

time

Series of images

*A*

*B*

*C*

*D*

*E*

*F*

a.h5

c.h5

e.h5

b.h5

d.h5

f.h5

Virtual Dataset VDS

*A*

*C*

*E*

*B*

*D*

*F*

t1

t2

t3

VDS.h5

111

Image at time t2

*A*

*B*

*C*

*D*

*E*

*F*

*A*

*B*

*C*

*D*

*E*

*F*

*A*

*A*

Dataset A

Dataset C

Dataset E

*C*

*C*

Dataset B

Dataset D

Dataset F

*F*

*F*

*D*

*D*

*E*

*E*

*B*

*B*

M

n

k

n

n

k

k

n

k

n

n

k

k

M

t2

t2

t2

t2

t2

t2

Figure 1: Excalibur use case. The series of each sub-image is stored in a 3D dataset with unlimited dimension corresponding to time. A virtual dataset presents the series of full images comprised of sub-images.

## DLS use case “Dual PCO Edge”

The “Dual PCO” detector is described in the section 3.2 of [1].

Data for a full image consists of 5 sub-images as showed in Figure 2. Each sub-image is stored as in use case 4.1 with one difference: each dataset is written by a selection that contains several consecutive sub-images. To optimize writing of compressed data the slowest changing dimension for a chunk can be set to a number of sub-images desired to be compressed and written with one HDF5 write call.

When written the data comprises of two groups of HDF5 files. Each group contains data for an image taken over the time from one camera. Sub-images are stored in the corresponding datasets as in the use case 4.1

Dual PCO use case

time

Series of sub-images

*A*

*B*

*C*

a.h5

c.h5

b.h5

*A*

*C*

*B*

t1

t2

t3

*D*

*E*

*D*

*E*

*D*

*E*

Series of sub-images

A

B

C

*A*

*B*

*C*

*D*

*E*

d.h5

e.h5

Virtual Dataset VDS

VDS.h5

Image at time t2

Dataset A

Dataset B

Dataset C

Dataset D

Dataset E

*A*

*A*

*B*

*B*

*C*

*C*

*E*

*E*

*D*

*D*

t2

t2

t2

t2

t2

Figure 2: Sub-images are taken from two different cameras. Storage layout is similar to Excalibur use case. Properties of each 3D dataset may be set to have “thicker” chunks for data buffering.

## Dealing with gaps

Both “Excalibur” and “Dual PCO Edge” have to accommodate gaps between sub-images. This can be accomplished by allowing “gaps” in the virtual dataset as shown in Figure 3. When a plane is read from VDS “gaps” are filled with the fill values defined by the virtual dataset, or with private data for the VDS.

Virtual Dataset VDS with “gaps”

VDS.h5

*A*

*C*

*B*

*D*

*E*

d.h5

e.h5

a.h5

c.h5

b.h5

*A*

*B*

*C*

*D*

*E*

Dataset A

Dataset B

Dataset c

*A*

*B*

*B*

*C*

*E*

*E*

*D*

*D*

Dataset D

Dataset E

*D*

*D*

*E*

*E*

*A*

*A*

Figure 3: Virtual datasets VDS contains “gaps” shown in white.

## DLS use case “Eiger”

The “Eiger” detector is described in the section 3.3 of [1].

Images from every M time steps are written to a 3D dataset into a separate HDF5 file. The slowest changing dimension of the 3D dataset has size M. When written, the data comprises N HDF5 files, each with M consequent images, except the last dataset that may have a fewer images as shown in Figure 4. Chunking and compression parameters for each 3D dataset are the same, for example, chunk size can be chosen to be the same as of the size of the image. The virtual dataset has a slowest changing dimension that represents “time” axis. Each plane orthogonal to the “time” axis represents an image.

*A*

*A*

*A*

f-1.h5

f-2.h5

f-3.h5

f-N.h5

DS N

time

Series of images

Virtual Dataset VDS

VDS.h5

File names are generated by printf

*A*

*A*

*A*

*A*

*A*

*A*

*A*

*A*

*A*

*A*

*A*

Figure 4: Each HDF5 file stores M consequent images.

## DLS use case “Percival Frame Builder”

The “Percival Frame Builder” detector is described in the section 3.4 of [1].

An image taken by the detector is stored as a 2D plane in a 3D dataset in HDF5 file. The images are added to the dataset along the slowest changing dimension, which represents time. Indices of the images added to the same file are monotonically increasing and the value mod(<image index>, <number of the HDF5 files>) is constant. For example, as showed in Figure 5 every 4th image is stored in the dataset A starting with the first image, every 4th images is stored in the dataset B starting with the second image, etc.

time

Series of images

a.h5

b.h5

c.h5

Dataset A

*B*

*C*

*D*

d.h5

Virtual Dataset VDS has images *A*, *B, C and D* interleaved

VDS.h5

Dataset B

Dataset C

Dataset D

*A*

*A*

*B*

*C*

*D*

*A*

*A*

*A*

*C*

*D*

*A*

*B*

*B*

*B*

*B*

*C*

*C*

*D*

*D*

*C*

t1

t2

t3

t4

t3+4k

t1+4k

Figure 5: Images are interleaved in the virtual dataset. Each of 3D datasets contains images taken at the time steps indexed with the fixed stride and a different offset.

## NPP aggregation

Temporal aggregation of the NPP data products is similar to the “Eiger” use case except that data is added along of the slowest 2D image dimension making VDS two-dimensional.

# HDF5 Virtual Dataset Requirements and Constraints

Based on the use cases above and some reasonable extrapolations from them, this section describes requirements and constraints for virtual datasets.

## Requirements

1. There will be no changes to the HDF5 programming model when accessing a VDS.
   1. Dataset create, write, read, sub-set, query and close operations work for VDS.
   2. Virtual datasets will have a new storage type: ‘virtual’ (cmp. with the current storage types: chunked, external, contiguous and compact).
2. Mapping between source dataset elements and VDS elements is described at VDS creation time, using the HDF5 dataspace selection mechanism, by specifying elements in a source dataset and corresponding elements in the VDS.
   1. Current selection mechanism should be extended to allow unlimited hyperslab “count” values to accommodate unbounded selections in source datasets that can be extended/have unlimited dimensions.
   2. There is a query mechanism to discover the mapping for a VDS.
3. A source dataset and VDS’s datatypes must be “convertible”, by the current algorithms available within the HDF5 library (which could include user-registered conversion routines).
4. A source dataset may have a different rank and dimension sizes than the VDS. However, if a source dataset has an unlimited dimension, it must be the slowest-changing dimension and the virtual dataset must be the same rank and have the same dimension as unlimited.
5. A source dataset may be a VDS itself.
6. A VDS may have its own “private” data, i.e., some elements of VDS may not be mapped to source dataset elements and may instead be stored locally to the VDS.
7. A filter can be applied to VDS “private” data (e.g. to compress it), although this will require that the VDS private data be stored in chunked layout.
8. There is a mechanism in the HDF5 library to search for source files at run time (similar to external links).
9. There is a “printf-like” capability to generate source dataset and file names based on some/all of an element’s coordinates in the VDS, enabling a dynamic mapping of source datasets into a VDS at runtime. If a virtual dataset has an unlimited dimension that isn’t the slowest changing dimension, it must use printf-formatting for mapping to source datasets.
10. There is a mechanism within the HDF5 library to validate a mapping when a VDS is created (e.g., selections in VDS mapping cannot overlap one another, including unbounded selections).
11. If there is an element of VDS that does not have a corresponding element in a source dataset or private VDS data, the element is filled with the VDS fill value for I/O operations.
12. There is a mechanism to override a “source” dataset’s fill value with the VDS fill value, when retrieving data from unallocated portions of a source dataset.
13. Source datasets may be written to with the ‘SWMR’ feature, and the virtual dataset may be refreshed to reflect the current size and data of those datasets.

## Constraints and Assumptions

The following constraints and assumption were identified:

1. There will be no UUIDs to identify source datasets/files (although they could be added, if desired).
2. It is a user’s responsibility to maintain consistency of a VDS. If source files are unavailable, the library will report an error or use fill values when doing I/O on the VDS elements mapped to the elements in the missing source files.
3. Since the source datasets may be out-of-sync with one another’s dimension sizes, the VDS needs to present a coherent view of the data in the source datasets (possibly a restating of assumption 2, above).
4. Source datasets can be assumed not to shrink in size.

## Derived Design Aspects

The following additional complexities were identified during feature design:

1. How should the HDF5 library detect ‘gaps’ in the sequence of files/datasets when printf formatting is used to identify source data for a virtual dataset? (Otherwise, the library could keep looking for the ‘next’ defined file/dataset indefinitely)

* Solution: Add a dataset access property for virtual datasets that specifies the largest ‘gap’ in the sequence of printf formatting to allow, defaulting to something sensible like ‘10’. This will allow applications to control the behavior, but put some upper limit in by default. This new property is described in a section below (link: [H5Pset\_virtual\_printf\_gap](#H5Pset_virtual_printf_gap)).

1. What should the HDF5 library report for the size of a virtual dataset when it has an unlimited dimension(s)? (Since the underlying source datasets are could be different lengths along that dimension)

* Solution: Add a dataset access property for virtual datasets that specifies whether to take the largest source dataset dimension or the smallest, defaulting to the largest. This will allow applications to choose, but default to including all the possible data. This new property is described in a section below (link: [H5Pset\_virtual\_dataspace\_bounds](#H5Pset_virtual_dataspace_bounds)).

1. How should the HDF5 library handle source and virtual datasets located in different directories?

* Solution: The HDF5 library should allow for alternate paths to locate source datasets/files to be specified with both an environment variable and a data access property at dataset open time. This new property is described in a section below (link: [H5Pset\_virtual\_source\_path](#H5Pset_virtual_source_path)).

# Programming model

This section describes the programming model for creating and accessing HDF5 virtual datasets.

The model is similar to the programming model for a regular HDF5 dataset and is summarized below. Please notice an additional step 2.d that is required for creating VDS. The steps 2.a – 2.e are described in detail in the subsections below.

Programming model for a VDS:

1. Create datasets that comprise the VDS (the source datasets) (optional).
2. Create VDS
   1. Define datatype
   2. Define dataspace
   3. Define dataset creation property list
   4. Map elements from the source datasets to the elements of VDS
      1. Iterate over the source datasets:
         1. Select elements in the source dataset (source selection)
         2. Select elements in the virtual dataset (destination selection)
         3. Map destination selection and source selection with dataset creation property list call
      2. End iteration
   5. Call H5Dcreate using properties defined above.
3. Access VDS as a regular HDF5 dataset.
4. Close VDS when finished.

## Create datasets that comprise VDS (optional)

One should follow the current HDF5 data model to create a source dataset if the dataset uses one of the current storage layouts (contiguous, chunked, compact, external) or the programming model for VDS if a source dataset is a virtual dataset.

This step is optional. If a dataset, which elements comprise VDS, doesn’t exist, one can specify the behavior by setting a special property: the fill values may be used when performing I/O on VDS or I/O fails with appropriate error message.

## Create the Virtual Dataset

### Define datatype

A virtual dataset may have any datatype, but it should be “compatible” with the datatypes of the source datasets. Two HDF5 datatypes are called compatible if the HDF5 library can convert an element of one datatype to the element of another datatype; for example, an integer value to a floating point value.

### Define dataspace

The dataspace of a virtual dataset is defined in the same way as a dataspace for a regular dataset. The VDS’s rank and dimension sizes do not depend on the dimensionality of the source datasets. One uses the H5Screate\_simple function to describe the VDS’s dataspace as shown below.

vspace\_id = H5Screate\_simple(vRANK, vdims, vmaxdims);

A VDS may have fixed or unlimited dimensions.

### Define creation property

To define the creation property for a virtual dataset, an application must use a new API routine to indicate that the dataset will be a virtual dataset as shown below:

dcpl\_id = H5Pcreate (H5P\_DATASET\_CREATE);

H5Pset\_virtual(dcpl\_id);

Other creation properties can be applied if VDS will have its “private” data. For example, one can use contiguous, chunked, external or compact storage layout for storing VDS data that is not part of the source datasets. One can also apply a desired compression to the VDS’s “private” data that is different from compression methods used with the source datasets.

### Map elements from the source datasets to the virtual dataset

If the source and virtual datasets have fixed size dimensions, one can use the current HDF5 hyperslab selection mechanisms to select elements for the mapping. As in the case of partial I/O, the number of elements selected in the source dataset for mapping must be the same as the elements selected in the virtual dataset.

The current hyperslab selection mechanism cannot be used if the dimensions of the source or virtual datasets are unlimited (as in several of the use cases described above) or repetition of a more sophisticated pattern is desired.

This section introduces an “unlimited” selection that overcomes a limitation of the current selection mechanism: a hyperslab pattern can be repeated only a finite number of times along each dimension (see the count parameter in the H5Sselect\_hyperslab function).

#### Unlimited selections

To create an unlimited selection one should use the following model:

1. Define a dataspace using the H5Screate\_simple function. Two restrictions are applied:
   1. The rank of the dataspace should be the same as the rank of the dataset to which the unlimited selection will be applied (see the H5Pset\_virtual\_mapping function below).
   2. The maximum dimensions should be set to H5S\_UNLIMITED in the same dimension as desired for unlimited selections:

hsize\_t dims[RANK] = {10, 10, 1};

hsize\_t max\_dims[RANK] = {10, 10, H5S\_UNLIMITED};

space\_id = H5Screate\_simple(RANK, dims, max\_dims);

1. Select an unlimited region in the dataspace as shown:

hsize\_t start[RANK] = {0, 0, 0};

hsize\_t stride[RANK] = {0, 0, 1};

hsize\_t block[RANK] = {10, 10, 1};

hsize\_t count[RANK] = {1, 1, H5S\_UNLIMITED};

H5Sselect\_hyperslab(space\_id, H5S\_SELECT\_SET, start, stride, block,

count);

Setting count[2] to H5S\_UNLIMITED indicates that the selection is unlimited. Although it is technically feasible to allow more complicated operations between selections that have unlimited count values, that capability is out of scope for this project and the ‘op’ parameter to H5Sselect\_hyperslab must always be H5S\_SELECT\_SET when an unlimited count value is used.

#### Select elements in a source and virtual dataset for mapping

The selection mechanism described above should be used to select the elements for mapping in both source and virtual datasets. If the mapping requires multiple source files, the selection in a source dataset may be a fixed-size selection and the printf form of mapping should be used (described below).

#### Mapping elements

After selections are defined on the source and virtual datasets, the mapping is accomplished by calling the H5Pset\_virtual\_mapping function:

status = H5Pset\_virtual\_mapping(dcpl\_id, vspace\_id, src\_file\_name, src\_dset\_name, src\_space\_id);

*Parameters:*

|  |  |  |
| --- | --- | --- |
| dcpl\_id | *-* | The dataset creation property identifier that will be used when creating the virtual dataset. |
| vspace\_id | *-* | The dataspace identifier with the selection within the virtual dataset applied, possibly an unlimited selection. |
| src\_file\_name | *-* | The name of the HDF5 file where the source dataset is located. The file is allowed to not exist (yet). The name can be specified using a C-style printf statement, as described below. |
| src\_dset\_name | *-* | The path to the HDF5 dataset in the file specified by src\_file\_name. The dataset is allowed to not exist (yet). The dataset name can be specified using a C-style printf statement, as described below. |
| src\_space\_id | *-* | The source dataset’s dataspace identifier with a selection applied, possibly an unlimited selection. |

*Returns*: negative on error, and positive on success.

When a selection with unlimited dimensions is used for the source dataset, the selection in the virtual dataset must also be an unlimited selection with the same number of unlimited dimensions. If fixed-size selections are used, the number of elements in the source dataset selection must be the same as the number of elements in the virtual dataset selection.

C-style printf formatting allows a pattern to be specified in the name of a source file or dataset. Strings for the file and dataset names are treated as literals, except for the following substitutions:

|  |  |  |
| --- | --- | --- |
| “%%” | - | Replaced with a single ‘%’ character. |
| “%<*d>b*” | - | Where <d> is the virtual dataset dimension axis (0-based) and ‘b’ indicates that the block count of the selection in that dimension should be used. The full expression (e.g. “%0b”) is replaced with a single numeric value when the mapping is evaluated at VDS access time. See the examples below for usage. |

If the printf form is used for the source file or dataset names, the selection in the source dataset’s dataspace must be fixed-size.

##### Examples of source and virtual dataset mappings

Example code for many source and virtual dataset mappings is available in Appendix I.

#### Create virtual dataset

After all required elements are mapped from the source datasets to the virtual dataset, one can create the dataset as usual:

vdset\_id = H5Dcreate(loc\_id, VDS\_name, datatype, vspace\_id, lcpl\_id,

dcpl\_id, dapl\_id);

If the mapping of source datasets to the virtual dataset’s dataspace does not entirely cover the virtual dataset’s dataspace elements, the virtual dataset will have “private” data elements for those elements. These elements are stored in the virtual dataset’s file as any other dataset would be and are accessed transparently along with the source dataset elements, with H5Dread and H5Dwrite. It is strongly recommended that if a virtual dataset has private data elements, chunked storage should be used for its storage layout, to take advantage of the sparse data storage available with chunked datasets (and chunked storage is required for virtual datasets with unlimited dimensions or that have compression filters applied).

## Performing I/O on a Virtual Dataset

Once a virtual dataset has been created, it may be accessed as with any other HDF5 dataset, using the H5Dread/H5Dwrite API calls. Elements mapped to source datasets are transparently accessed from those datasets and private elements for the virtual dataset are accessed from the virtual dataset’s storage, without further actions from the user application. Currently, there are no new data transfer properties defined for virtual datasets, but they may be defined in the future.

## Closing a Virtual Dataset

Closing a virtual dataset occurs with H5Dclose, as with any other HDF5 dataset. Any cached information about the source datasets is released transparently to the application.

## Opening a Virtual Dataset

Once a virtual dataset has been created, it may be opened as with any other HDF5 dataset, using the H5Dopen API call. The following dataset access properties may be set when opening (or creating) a virtual dataset to control various aspects of library behavior.

#### Choose dataspace size reported for virtual datasets

Virtual datasets can rely on multiple source datasets that may have different dimension sizes, if they a have unlimited dimensions (see e.g. examples 3-7 in Appendix I). By default, the dimension sizes of the virtual dataset reported with the H5Dget\_space API routine will be the maximum of all the underlying source datasets’ dimensions. If the application would prefer to have H5Dget\_space report the minimum of all underlying source datasets’ dimensions, the following property can be set on a dataset access property list:

status = H5Pset\_virtual\_dataspace\_bounds(dapl\_id, bounds\_option);

*Parameters:*

|  |  |  |
| --- | --- | --- |
| dapl\_id | *-* | The dataset access property identifier for accessing the VDS |
| bounds\_option | *-* | This parameter can be either H5D\_VDS\_MAX or H5D\_VDS\_MIN, to choose the maximum of all underlying source datasets’ dimensions or the minimum, respectively, when a virtual dataset’s dimensions are queried with H5Dget\_space. |

*Returns*: negative on error, and positive on success.

#### Choose gap size for printf-formatted source dataset/file names

Virtual datasets can rely on source datasets which names are determined by a printf-formatted string when the dataset is created. However, when those datasets or files need to be accessed later, some method must be used to determine that there are no more datasets or files in the sequence generated by the formatting. By default, the library will search for up to 10 more matches to the printf-formatting pattern after the last match of the dataset/file pattern, but the number of matches can be controlled with the following property set on a dataset access property list:

status = H5Pset\_virtual\_printf\_gap(dapl\_id, gap\_size);

*Parameters:*

|  |  |  |
| --- | --- | --- |
| dapl\_id | *-* | The dataset access property identifier for accessing the VDS |
| gap\_size | *-* | This parameter indicates the number of printf-formatted datasets/files which can be missing before the library stops searching for more datasets/files. A value of 0 indicates that no gaps are tolerated. The default value is a gap size of 10. |

*Returns*: negative on error, and positive on success.

#### Choose path to search for locating source datasets/files

Virtual datasets can rely on source datasets whose containing files may not be located in the same directory as the virtual dataset. A path to locate files for source datasets may be specified in one of two ways: an environment variable (“HDF5\_VDS\_SOURCE\_PATH”) or with the following property set on a dataset access property list:

status = H5Pset\_virtual\_source\_path(dapl\_id, source\_path);

*Parameters:*

|  |  |  |
| --- | --- | --- |
| dapl\_id | *-* | The dataset access property identifier for accessing the VDS |
| source\_path | *-* | This parameter is a string containing a colon-separated list of paths to search for files containing source datasets. The default value is the null string (“”). The paths specified by the “HDF5\_VDS\_SOURCE\_PATH” environment variable are searched before the paths specified by this property. |

*Returns*: negative on error, and positive on success.

# Use Case Implementation

This section will provide pseudocode examples how the programming model introduced in section 5 can be applied to create VDS for the use cases 3.1, 3.3, 3.4 and 3.5.

## Excalibur use case

For this use, case source datasets do not exist at the time we create the VDS. The source HDF5 files and datasets will be written by independent processes and will be available for applications that access the VDS.

We will start with the Programming model step 2 - defining creation properties and dataspace for the VDS, then we will proceed with selecting elements in each of the source datasets and the corresponding elements in the VDS, and mapping before creating the VDS.

/\* Set creation property to be used in mapping definition \*/

vdcpl\_ id = H5Pcreate (H5P\_DATASET\_CREATE);

H5Pset\_virtual(vdcpl\_id);

/\* Create a dataspace for virtual dataset VDS \*/

N = 3 \* (k + n); /\* See Figure in section 3.1 for the dimensions of full image \*/

vdims[] = {1, N, M};

vmaxdims = {H5S\_UNLIMITED, N, M};

vspace\_id = H5Screate\_simple(3, vdims, vmaxdims);

/\* Start the loop over the source datasets to define mapping \*/

for each DATASET in (A, B, C, D, E, F) {

/\* First, describe dataspace for a source DATASET \*/

src\_dims[] = {1, *i*, M}; /\* *i* is k for A, C, E and n for B, D, F \*/

src\_maxdims[] = {H5S\_UNLIMITED, *i*, M}; /\* *i* is k for A, C, E and n for B, D, F \*/

src\_dataspace\_id = H5Screate\_simple(3, src\_dims, src\_maxdims);

/\* Define unlimited selection for the source dataset \*/

start = {0, 0, 0};

stride = {1, 1, 1};

count = {H5S\_UNLIMITED, 1, 1};

block = {1, *i*, M}; /\* *i* is k for A, C, E and n for B, D, F \*/

H5Sselect\_hyperslab(src\_dataspace\_id, H5S\_SELECT\_SET, start, stride, count,

block);

/\* Select elements in VDS that will be mapped with the elements in DATASET \*/

vstart = {0, *p*, 0}; /\*The value of *p* is

0 for A,

n for B,

n+k for C,

n+2k for D,

2n+2k for E,

3k+2n for F \*/

vstride = {1, 1, 1};

vblock

H5Sselect\_template(vspace\_id, H5S\_SELECT\_SET, vstart, vstride,

vcount, vblock);

/\* Map selections \*/

H5Pset\_virtual\_mapping(vdcpl\_id, vspace\_id, FILE, DATASET, src\_dataspace\_id);

} /\* End loop over the source datasets \*/

/\* Create virtual dataset using regular HDF5 call \*/

VDS\_id = H5Dcreate(…, “VDS”,…, vspace\_id, …, vdcpl\_id,…);

## Generalized Excalibur use case (dealing with gaps)

The only difference between 5.1 and 5.2-3 is the selection in the VDS dataspace.

vstart = {0, *nSUB-IMAGE*, *mSUB-IMAGE*};

Values *nSUB-IMAGE* and *mSUB-IMAGE* indicate the offset (in number of elements to skip) of sub-images in the full image. For example, *nD* and *mD* are offsets for sub-image *D* as shownin Figure 10 below.

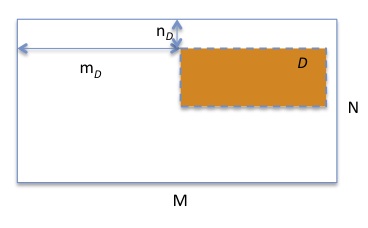


Figure 13: Offsets for sub-image D in the full image

The programming model can easily accommodate gaps shown in Figure 4.

This use case is the same as the base Excalibur use case.

## Eiger use case

The Eiger use case is different from the first two use cases discussed. There is no template selection in the source datasets. The whole dataspace for each dataset in a files f-*i*.h5 will be selected. The loop over the source datasets corresponds to the call to H5Pset\_virtual\_mapping using “printf” formatting for file names.

/\* Set creation property to be used in mapping definition \*/

vdcpl\_ id = H5Pcreate (H5P\_DATASET\_CREATE);

H5Pset\_virtual(vdcpl\_id);

/\* Create a dataspace for virtual dataset VDS \*/

vdims[] = {M, N, K};

vmaxdims = {H5S\_UNLIMITED, N, K};

vspace\_id = H5Screate\_simple(3, vdims, vmaxdims);

/\* No loop over the source datasets to define mapping; we can use the

same source dataspace as shown on the next two lines. The dataspace contains M

images, and the size of each image is NxK \*/

dims[] = {M, N, K};

src\_id = H5Screate\_simple(3, dims, NULL);

/\* Select elements in VDS that will be mapped with the elements in DATASET \*/

vstart = {0, 0, 0};

vcount = {H5S\_UNLIMITED, 1, 1};

vblock = {M, N, K};

H5Sselect\_hyperslab(vspace\_id, H5S\_SELECT\_SET, vstart, vstride, vcount, vblock);

/\* Map selections \*/

H5Pset\_virtual\_mapping(vdcpl\_id, vspace\_id, “f-%0b.h5”, “/A”, src\_id);

/\* Finally create virtual dataset using regular HDF5 call \*/

VDS\_id = H5Dcreate(…, “VDS”,…, vspace\_id, …, vdcpl\_id,…);

Notice that files “f-%0b.h5” and datasets in them may not exist.

## Percival Frame Builder use case

This example will use offset and stride when selecting templates in the VDS dataspace. The rest of the example follows the example discussed in 6.1.

/\* Set creation property to be used in mapping definition \*/

vdcpl\_ id = H5Pcreate (H5P\_DATASET\_CREATE);

H5Pset\_virtual(vdcpl\_id);

/\* Second, create a dataspace for virtual dataset VDS \*/

vdims[] = {1, N, M};

vmaxdims = {H5S\_UNLIMITED, N, M};

vspace\_id = H5Screate\_simple(3, vdims, vmaxdims);

/\* Selection for all source datasets will be the same \*/

dims [] = {1, N, M};

maxdims [] = {H5S\_UNLIMITED, N, M};

src\_id = H5Screate\_simple(3, dims\_DATASET, maxdims);

/\* Select elements in the source dataset \*/

start = {0, 0, 0};

count = {H5S\_UNLIMITED, 1, 1};

block = {1, N, M};

H5Sselect\_hyperslab(src\_id, H5S\_SELECT\_SET, start, stride, count, block);

/\* Start the loop over the source datasets to define mapping \*/

for each DATASET in (A, B, C, D,) {

/\* Select elements in VDS that will be mapped with the elements in DATASET \*/

vstart = {*p*, 0, 0}; /\*The value of *p* is

0 for A,

1 for B,

2 for C,

3 for D \*/

vcount = {H5S\_UNLIMITED, 1, 1};

vblock = {1, N, M};

/\* Select elements in VDS \*/

H5Sselect\_hyperslab(vspace\_id, H5S\_SELECT\_SET, vstart, vstride, vcount, vblock);

/\* Map selections \*/

H5Pset\_virtual\_mapping(vdcpl\_id, vspace\_id, FILE, DATASET, src\_id);

}

/\* Create virtual dataset using regular HDF5 call \*/

VDS\_id = H5Dcreate(…, “VDS”,…, vspace\_id, …, vdcpl\_id,…);

# Implementation Details

This section describes the changes to major components of the HDF5 software package needed for implementing the virtual dataset capability.

## HDF5 Library

The core of the virtual dataset implementation will be in the implementation of the VDS dataset layout type. This consists of adding callback functions for the VDS H5D\_layout\_t struct and other auxiliary functions. First, we will have to create a structure for the VDS storage type, which will allow the library to track persistent information about the VDS storage for the dataset that is not contained in the layout info message or that should be cached for efficiency. We will also need to create initialization routines to initialize this structure when a dataset is opened, as well as a routine to initialize the layout structure when the dataset is created.

We will have to create a layout callback to inform the library if space has been allocated for the dataset yet. We will need to decide whether to examine the source datasets to see if they have been allocated, or simply return yes, at least until the private data feature is implemented (if ever). We will need to write a function to initialize structures used during an I/O operation. This routine may need to make decisions about how to proceed with I/O and cache the results in the I/O structure, or it may defer those decisions to the read/write callbacks. The translation between selections in the VDS and selections in the source datasets is one major task that could be in either place. This will use HDF5 hyperslab routines in order to transform the selections as necessary. The other major task that could be performed in either place is the resolution of file and dataset names for each source dataset in the selection. This includes resolving the “printf” style names to strings that can be used to open the source dataset directly.

The H5Dread and H5Dwrite routines, used to perform actual I/O, will also need to determine if the I/O is a collective MPI operation, and if so, change it to independent, while adding a barrier to make is behave like a collective operation. The read routine will need to, when a selected part of the VDS does not have an associated source dataset, propagate the fill value to the relevant places in the application's read buffer. We will need to write a flush routine, which will instruct all source datasets to flush, as well as a routine to release any resources used during I/O.

We will also need to make changes to the existing HDF5 hyperslab routines to support the new “unlimited” selections needed for the VDS feature. Hyperslabs with unlimited selections may not be combined with any other selections. This will simplify the process of creating a hyperslab, as the library need only record the initial description of the hyperslab (start/stride/count/block). Other hyperslab functions, such as iterator functions, will need to be updated to handle unlimited selections by clipping the selection to the extent of the dataspace. We will also need to update the serialized form of a dataspace to account for unlimited selections.

The dataset layout object header message will need to be updated to be able to describe virtual datasets. This includes both the in-memory structure and the serialized format written to the file. These will need to include the list of selections and paths to source datasets needed to construct the VDS. This work will use the updated hyperslab routines for serializing selections described above. We will also need to update region references to be able to handle unlimited selections, though this work should be minimal, as it should just use the same hyperslab routines.

Finally, the new API functions will need to be written and others will need to be updated to reflect the changes specified in the requirements document. The API functions themselves will mostly check the parameters for validity and translate them into a form the internal library handles before calling internal functions. We will also need to add internal functions to handle the VDS layout on dataset creation property lists, as well as add the options to control the behavior of H5Dget\_storage\_size and update the search path for source files to dataset access property lists. We will need to update H5Dget\_space to handle virtual datasets by checking to see which source datasets have been allocated and what sizes they are. Finally, we will need to update H5Ocopy to property copy virtual datasets.

## HDF5 Library Testing

The bulk of the regression testing will test the virtual dataset feature directly. These tests will start from simple cases, such as a single non-unlimited source dataset, and build to complex configurations, while testing that corner cases, such as a source dataset with a null dataspace, are handled correctly. These tests will also make sure that invalid inputs, such as overlapping source datasets, are rejected appropriately.

We will also need to write similar tests for the hyperslab selection code to test that unlimited selections are handled correctly by all the internal selection routines, including the routines to serialize and deserialize the selection. We will also add a test to verify that region references to unlimited selections are handled appropriately, as well as a test to make sure that virtual datasets can be accessed in SWMR (single writer, multiple reader) mode correctly.

In addition to the internal tests, we will also write an acceptance test suite to make sure we correctly handle the customer's use cases. We will work closely with the customer to develop these tests to ensure that they closely reflect, and the customer will sign off on the test suite before the project is complete. In addition to the acceptance tests, we will also write a performance benchmark to ensure that the virtual dataset feature's performance in the customer's use cases adequately meets their needs. We will also write a performance regression test to ensure that the performance of the VDS feature does not degrade in the future.

## HDF5 Command-Line Tools & Testing

As the design of the changes to the command-line tools for HDF5 (e.g. h5dump, h5repack, etc.) to support virtual datasets has not yet been performed, implementation details are not yet available. As the design for the command-line tools is finalized, this section will be updated with information about the implementation and testing details.

## HDF5 Language Wrappers & Testing

As with the section above, the language wrappers have not yet been designed, and this section will be updated at that time.

# Appendix I – Examples of Source to Virtual Dataset Mapping

7

8

9

10

11

12

13

# References

1. DLS RFC: HDF5 Parallel Compressed Writer Requirements and Use Cases”, Doc No: TDI-CTRL-REQ-034, Issue: 0.2, April 5, 2013

# Revision History

|  |  |
| --- | --- |
| *April 24, 2013:* | Version 1 prepared for the brain storming meeting by Elena |
|  | Version 2: accepted Quincey’s changes; sent to DLS |
| August 20, 2013 | Version 3: drawing and programming model were added |
| September 1, 2013 | Version 4: revised programming model |
| September 3, 2013 | Version 5: created section 7; sent to DLS |
| November 21, 2014 | Version 6: Many revisions to simplify programming model, added examples of how to use mapping function call; sent to DLS |
| December 4, 2014 | Version 7: Moved programming examples to appendix, more cleanups and clarifications. |
|  |  |