HDF5 DAOS VOL Connector Design

Neil Fortner, Jordan Henderson, Jerome Soumagne

This document is the design document for the HDF5 DAOS VOL connector. It aims to provide a detailed description of the manner in which the VOL connector maps between HDF5 objects and DAOS objects. It also includes design notes for the new HDF5 Map objects.

Revision History

Version Number	Date	Comments
v0.1	Apr. 24, 2019	First draft.
v0.2	Oct. 27, 2019	Second draft.
v0.3	Nov. 1, 2019	Third draft.
v1.0	Dec. 16, 2019	Initial release of DAOS VOL connector.

Contents

List of Figures

Acronyms

DAOS Distributed Asynchronous Object Storage. 4, 6, 8, 11, 12, 14–16

HDF5 Hierarchical Data Format, v5. 6–12, 15, 16

VOL Virtual Object Layer. 12

Glossary

akey A DAOS 'attribute key,' which distinguishes individual arrays. Similar to a dkey, an akey has an arbitrary size.. 6–10, 16

connector A VOL connector is a thin layer between HDF5 and the storage. It translates HDF5 VOL storage related calls into actual storage operations.. 6, 11, 12, 15

dkey A DAOS 'distribution key,' which denotes a set of arrays co-located on the same storage targets. A dkey has an arbitrary size.. 6–10, 16

1 VOL Connector Design

1.1 Mapping HDF5 to DAOS

The central paradigm for mapping HDF5 to DAOS is that HDF5 files are stored as DAOS containers, and HDF5 objects (groups, datasets, committed datatypes, and maps) are stored as DAOS objects, both using a 1:1 mapping. All metadata and raw data associated with an HDF5 object is then stored as entries in the key value store associated with that object provided by DAOS.

Since DAOS containers are identified by a UUID, the connector uses a hashing function to generate a UUID from the file name. Each container for an HDF5 file contains at least two objects: the global metadata object and the root group. The global metadata object stores metadata that describes properties that apply to the entire file. The root group is always the start of the HDF5 group structure, and uses the same format as all other groups.

Similar to the native HDF5 file format, the formats for the different object types are kept similar, with only differences as necessary to represent the specific object type. All object types have a creation property list and a reference count (not yet implemented) stored under the `/Internal Metadata` dkey and the `Creation Property List` and `Reference Count` akeys, respectively. The `/` prefix for the `/Internal Metadata` dkey prevents conflicts with arbitrarily named HDF5 links, which cannot contain the `/` character.

1.1.1 Groups

The purpose of HDF5 groups is to store links to other objects. Each link is stored under a dkey equal to the link's name and under the constant akey `Link`. Links can be hard links, in which case the value stored for the link is the object ID for the target object. This object ID can then be used to directly open the target object. Alternatively, links can be soft links, in which case the value stored for the link is the path name of the target object. Once retrieved, this path name must be traversed in order to open the target object.

When link creation order is tracked for a group, additional information is written to that group under the `/Link Creation Order` dkey. Under that dkey, link creation order-related information is stored under the following akeys:

- `Max Link Creation Order` — This akey stores the current maximum link creation order value for the group. The maximum link creation order value starts at 0 and increases as new links are created in the group. This value is mostly used to supply a group's maximum link creation order value when a call to `H5Gget_info(_by_name/_by_idx)` is made.
- `Num Links` — This akey stores the current number of links in the group.
- `<encoded num. links>` — The value for this akey is determined by encoding the current number of links in the group *at creation time of the new link* into a binary buffer; this encoded value represents the creation order index value assigned to the new link being written. This akey is used to store a mapping from link creation order index values to link names, which allows an easy lookup of a link's name by a given creation order index value. One of these akeys will exist for each link currently in the group, as a new akey will be added each time a new link is written to the group.
- `<encoded num. links>0` — The value for this akey is determined similarly to the previous akey, but the akey value is suffixed with a trailing `0` byte in order to distinguish it from the previous akey. This akey is used to store a mapping from link creation order index values to link targets, which allows an easy lookup of a link's target by a given creation order index value. One of these akeys will exist for each link currently in the group, as a new akey will be added each time a new link is written to the group.

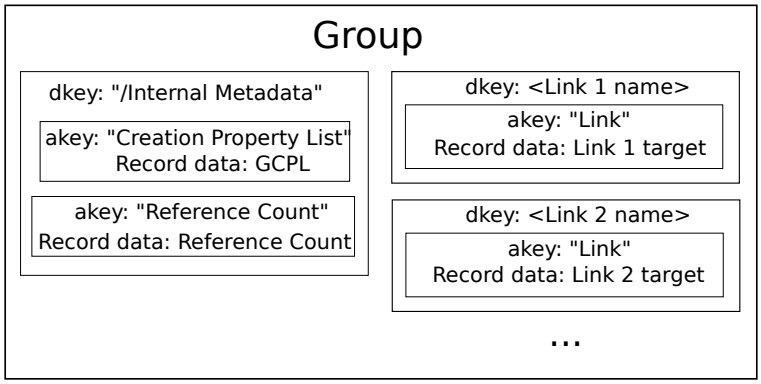

Figure 1 – Group dkey and link examples.

1.1.2 Datasets

Datasets are used to store the bulk array data in HDF5. A dataset's metadata is stored under the `/Internal Metadata` dkey and under the following akeys:

- `Datatype` — This akey stores the dataset's datatype.
- `Dataspace` — This akey stores the dataset's dataspace.
- `Creation Property List` — This akey stores the dataset's Dataset Creation Property List.
- `Fill Value` — This optional akey stores the dataset's fill value in file format.

A dataset's raw data is divided into "chunks" which are regularly spaced blocks of data, where the size of the block is specified by the application. Each data chunk is stored in its own dkey, which is encoded with a leading 0 byte, followed by <number of dimensions> 64-bit little endian unsigned integers which denote the chunk offset within the dataset. The `0` prefix for chunks prevents the binary chunk dkeys from conflicting with the string metadata dkey, since nonzero length strings cannot begin with 0 (`'\0'`).

For datasets with fixed-length datatypes, the data for a chunk is stored in a single akey with a value of a 0 byte. The data is stored as an array of records, where each record corresponds to an element in the dataset. The record size is therefore equal to the size of the datatype and the number of records is equal to the number of elements in the data chunk. For datasets with variable-length datatypes, the fixed-length portion of the data is stored normally, but each variable-length sequence is stored separately in the global metadata object, with a dkey consisting of a randomly generated UUID, and the constant akey `Blob`. The UUID used for the dkey for the variable-length sequence is stored in the fixed-length portion of the dataset. Reference type data is stored in the same manner as variable-length data.

Fill values are not written to the dataset's data array, rather, we take advantage of DAOS' behaviour when reading from extents that have not been written to. In this case, DAOS does not modify the sections of the read buffer that correspond to the missing records. In order to implement fill values, when the user reads from a dataset, we first propagate the fill value to the entire selection in the read buffer (or type conversion buffer if appropriate), then after fetching the data from DAOS, any unwritten elements will contain the fill value.

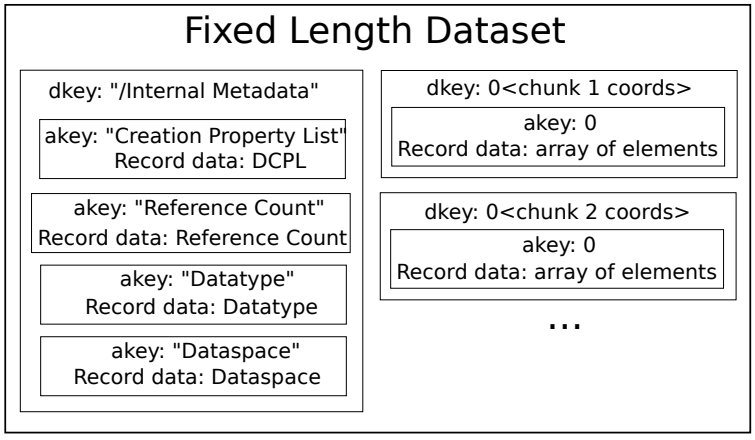

Figure 2 – Fixed length dataset dkey examples.

1.1.3 Committed Datatypes

A committed datatype is an HDF5 datatype that has been stored as an HDF5 object in the group structure. A committed datatype's metadata is stored under the `/Internal Metadata` dkey and under the following akeys:

- `Datatype` — This akey stores the committed datatype's actual datatype.
- `Creation Property List` — This akey stores the committed datatype's Datatype Creation Property List.

1.1.4 Attributes

HDF5 attributes attached to an object (group, dataset, committed datatype or map object) are stored under that object's `/Attribute` dkey (note that the `/` prefix for the `/Attribute` dkey prevents conflicts with arbitrarily named HDF5 links, which cannot contain the `/` character). Under that dkey, an attribute's metadata and raw data is stored under the following akeys:

- `T-<attribute name>` — This akey stores the attribute's datatype.
- `S-<attribute name>` — This akey stores the attribute's dataspace.
- `P-<attribute name>` — This akey stores the attribute's Attribute Creation Property List.
- `V-<attribute name>` — This akey stores the attribute's value.

When attribute creation order is tracked for an object, additional information is written to that object under the same `/Attribute` dkey and under the following akeys:

- `Max Attribute Creation Order` — This akey stores the current maximum attribute creation order value for the object. The maximum attribute creation order value starts at 0 and increases as new attributes are added to the object.
- `Num Attributes` — This akey stores the current number of attributes attached to the object.
- `0<encoded num. attributes>` — The value for this akey is determined by encoding the current number of attributes attached to the object *at creation time of the new attribute* into a binary buffer, prefixed with a leading `0` byte; this encoded value represents the creation order index value assigned to the new attribute being written. This akey is used to store a mapping from attribute creation order index values to attribute names, which allows an easy lookup of an attribute's name by a given creation order index value. One of these akeys will exist for each attribute currently attached to the object, as a new akey will be added each time a new attribute is written to the object.
- `C-<attribute name>` — This akey stores the attribute's creation order value.

1.1.5 Object IDs

DAOS objects are referenced through the DAOS API by a 128 bit object ID. DAOS reserves the upper 32 bits to encode its information, such as the object class. The second 32 bits are used by the application to encode its specific information, and the lower 64 bits are used as a unique object identifier. The lower 64 bits are generated using a DAOS routine to allocate ID ranges for each process, with a separate range of IDs for objects created collectively. There are currently two reserved values for these lower 64 bits, 0 for the global metadata object, and 1 for the root group. The top 2 bits of the application information block are used to encode the HDF5 object type. This removes the need to store the object type in the key-value store for the object itself, and allows the connector to determine the object type and therefore the routines used to access it without needing to query DAOS, reducing the number of server requests needed for metadata operations. The entire 128 object ID is used as the ”token” for the HDF5 token API, which is meant to replace the use of file addresses.

2 HDF5 Map Objects

While the HDF5 data model is a flexible way to store data that is widely used in HPC, some applications require a more general way to index information. While HDF5 effectively uses key-value stores internally for a variety of purposes, it does not expose a generic key-value store to the API. As part of the DAOS project, we will be adding this capability to the HDF5 API, in the form of HDF5 Map objects. These Map objects will contain application-defined key-value stores, to which key-value pairs can be added, and from which values can be retrieved by key.

2.1 Map Object API

To implement map objects, we will add new API routines, and new VOL callbacks, to the HDF5 library. For now, though, we will not be implementing support for maps in the default (native) VOL connector, meaning that map objects will only work with the DAOS connector, and with any other VOL connectors that are written to support maps.

The HDF5 Map API will consist of 11 new HDF5 API functions for managing Map objects, plus closely related functions such as `H5Mcreate_anon()`, `H5Mopen_by_name()`, etc. that are excluded from this list for the sake of brevity.

2.1.1 H5Mcreate

```
hid_t H5Mcreate(hid_t loc_id, const char *name, hid_t keytype, hid_t valtype,
                hid_t lcpl_id, hid_t mcpl_id, hid_t mapl_id);
```

`H5Mcreate()` creates a new Map object in the specified location in the HDF5 file and with the specified name. The datatype for keys and values can be specified separately, and any further options can be specified through the property lists `lcpl_id`, `mcpl_id`, and `mapl_id`.

There are currently some restrictions on the key datatype when used with the DAOS connector. The key datatype may not contain a reference type, and it may not contain a nested variable-length (vlen) type, that is, there cannot be a vlen present inside a compound, array, or another vlen type.

2.1.2 H5Mopen

```
hid_t H5Mopen(hid_t loc_id, const char *name, hid_t mapl_id);
```

`H5Mopen()` opens a previously created Map object at the specified location with the specified name. Any further options can be specified through the property list `mapl_id`.

2.1.3 H5Mput

```
herr_t H5Mput(hid_t map_id, hid_t key_mem_type_id, const void *key,
              hid_t val_mem_type_id, const void *value, hid_t dxpl_id);
```

`H5Mput()` adds a key-value pair to the Map specified by `map_id`, or updates the value for the specified key if one was set previously. `key_mem_type_id` and `val_mem_type_id` specify the datatypes for the provided

key and value buffers, and if different from those used to create the Map object, the key and value will be internally converted to the datatypes for the map object. Any further options can be specified through the property list `dxpl_id`.

2.1.4 H5Mget

```
herr_t H5Mget(hid_t map_id, hid_t key_mem_type_id, const void *key,
              hid_t val_mem_type_id, void *value, hid_t dxpl_id);
```

`H5Mget()` retrieves, from the Map specified by `map_id`, the value associated with the provided key. `key_mem_type_id` and `val_mem_type_id` specify the datatypes for the provided key and value buffers. If `key_mem_type_id` is different from that used to create the Map object the key will be internally converted to the datatype for the map object for the query, and if `val_mem_type_id` is different from that used to create the Map object the returned value will be converted to `val_mem_type_id` before the function returns. Any further options can be specified through the property list `dxpl_id`.

2.1.5 H5Mexists

```
herr_t H5Mexists(hid_t map_id, hid_t key_mem_type_id, const void *key,
                 hbool_t *exists, hid_t dxpl_id);
```

`H5Mexists()` checks if the provided key is stored in the Map specified by `map_id`. If `key_mem_type_id` is different from that used to create the Map object the key will be internally converted to the datatype for the map object for the query. Any further options can be specified through the property list `dxpl_id`.

2.1.6 H5Mget_key_type

```
hid_t H5Mget_key_type(hid_t map_id);
```

`H5Mget_key_type()` retrieves the key datatype ID from the Map specified by `map_id`.

2.1.7 H5Mget_val_type

```
hid_t H5Mget_val_type(hid_t map_id);
```

`H5Mget_val_type()` retrieves the value datatype ID from the Map specified by `map_id`.

2.1.8 H5Mget_count

```
herr_t H5Mget_count(hid_t map_id, hsize_t *count, hid_t dxpl_id);
```

`H5Mget_count()` retrieves the number of key-value pairs stored in the Map specified by `map_id`. Any further options can be specified through the property list `dxpl_id`.

2.1.9 H5Miterate

```
herr_t H5Miterate(hid_t map_id, hsize_t *idx, hid_t key_mem_type_id,
                  H5M_iterate_t op, void *op_data, hid_t dxpl_id);
```

`H5Miterate()` iterates over all key-value pairs stored in the Map specified by `map_id`, making the callback specified by op for each. The `idx` parameter is an in/out parameter that may be used to restart a previously interrupted iteration. At the start of iteration `idx` should be set to 0, and to restart iteration at the same location on a subsequent call to `H5Miterate()`, `idx` should be the same value as returned by the previous call. `H5M_iterate_t` is defined as:

```
herr_t (*H5M_iterate_t)(hid_t map_id, const void *key, void *op_data);
```

The key parameter is the buffer for the key for this iteration, converted to the datatype specified by `key_mem_type_id`. The `op_data` parameter is a simple pass through of the value passed to `H5Miterate()`, which can be used to store application-defined data for iteration. A negative return value from this function will cause `H5Miterate()` to issue an error, while a positive return value will cause `H5Miterate()` to stop iterating and return this value without issuing an error. A return value of zero allows iteration to continue.

To implement this function, in order to reduce the number of calls to DAOS that may cause network access, we will fetch more than one key at a time from DAOS. However, since we do not know the size of the keys or the memory usage limitations of the application, it is difficult to know the number of keys we should prefetch in this manner. Currently we plan to add a Map access property to control the number of keys prefetched for iteration, and this function is described below as `H5Pset_map_iterate_hints()`. We could alternatively keep track of the average key size in the file and add a property list setting to control the average memory usage for iteration.

2.1.10 H5Mdelete

```
herr_t H5Mdelete(hid_t map_id, hid_t key_mem_type_id, const void *key,
                 hid_t dxpl_id);
```

`H5Mdelete()` deletes a key-value pair from the Map specified by `map_id`. `key_mem_type_id` specifies the datatype for the provided `key` buffer, and if different from that used to create the Map object, the key will be internally converted to the datatype for the map object. Any further options can be specified through the property list `dxpl_id`.

2.1.11 H5Pset_map_iterate_hints

```
herr_t H5Pset_map_iterate_hints(hid_t mapl_id, size_t key_prefetch_size,
                                size_t key_alloc_size);
```

`H5Pset_map_iterate_hints()` adjusts the behavior of `H5Miterate()` when prefetching keys for iteration. The `key_prefetch_size` parameter specifies the number of keys to prefetch at a time during iteration. The `key_alloc_size` parameter specifies the initial size of the buffer allocated to hold these prefetched keys, as well as DAOS metadata. If this buffer is too small it will be reallocated to a larger size, though this will result in an additional call to DAOS.

2.1.12 H5Mclose

```
herr_t H5Mclose(hid_t map_id);
```

`H5Mclose()` closes the Map object handle `map_id`.

2.1.13 Example

Below is a short example program for storing ID numbers indexed by name. It creates a map and adds two key-value pairs, then retrieves the value (and integer) using one of the keys (a string).

```
hid_t file_id, fapl_id, map_id, vls_type_id;
const char *names[2] = ["Alice", "Bob"];
uint64_t IDs[2] = [25385486, 34873275];
uint64_t val_out;

<DAOS setup>

vls_type_id = H5Tcopy(H5T_C_S1);
H5Tset_size(vls_type_id, H5T_VARIABLE);

file_id = H5Fcreate("file.h5", H5F_ACC_TRUNC, H5P_DEFAULT, fapl_id);

map_id = H5Mcreate(file_id, "map", vls_type_id, H5T_NATIVE_UINT64,
                   H5P_DEFAULT, H5P_DEFAULT, H5P_DEFAULT);

H5Mput(map_id, vls_type_id, &names[0], H5T_NATIVE_UINT64, &IDs[0],
       H5P_DEFAULT);
H5Mput(map_id, vls_type_id, &names[1], H5T_NATIVE_UINT64, &IDs[1],
       H5P_DEFAULT);

H5Mget(map_id, vls_type_id, &names[0], H5T_NATIVE_UINT64, &val_out,
       H5P_DEFAULT);
if(val_out != IDs[0])
        ERROR;

H5Mclose(map_id);
H5Tclose(vls_type_id);
H5Fclose(file_id);
```

Figure 3 – Diagram of a Map Object in DAOS as created by the example.

2.2 Implementation of Map Objects

Since DAOS is built on top of key-value stores, implementation of map objects in the DAOS connector is fairly straightforward. Like other HDF5 objects, all Map objects will have a certain set of metadata, stored in the same manner as other objects. In this case, the Map objects will need to store serialized forms of the

key datatype, value datatype, and map creation property list (MCPL), as obtained from `H5Tencode()` and `H5Pencode()`. This map object metadata is stored under the `/Internal Metadata` dkey and under the following akeys:

- `Key Datatype` — This akey stores the map object's key datatype.
- `Value Datatype` — This akey stores the map object's value datatype.
- `Creation Property List` — This akey stores the map object's Map Creation Property List.

When setting a key-value pair, we will first convert the key and value to the file datatypes using existing HDF5 facilities, then we will set that pair as a key-value pair in the DAOS object using `daos_obj_update()`, where the key is used for the DAOS dkey field, and the DAOS akey field is set to `MAP_AKEY`. Querying values will likewise use `daos_obj_fetch()` with the same dkey and akey to retrieve the value associated with a key, and HDF5 facilities to perform datatype conversion as needed.

For now, map creation property lists will only contain generic object and link creation properties that apply to all object types. Map access property lists will contain the *map iterate hints* property described above for `H5Pset_map_iterate_hints()`, as well as generic object and link access properties that apply to all object types. This architecture will allow properties specific to map objects to be added at a later time with no change to the existing API functions.