HDF5 DAOS VOL Connector Design

Neil Fortner, Jordan Henderson, Jerome Soumagne

This document is an on-going draft design document of the DAOS VOL connector.

It aims at providing a detailed description of the mapping between HDF5 objects and DAOS objects. It also includes the design notes of the new HDF5 Map objects.

Revision History

Version Number	Date	Comments
v1	Apr. 24, 2019	First draft.

Contents

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, named 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 plugin 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. Currently this is only the maximum object id. 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. All object types also have attributes stored under the `/Attribute` dkey. Each attribute stores its datatype under the `T-<attribute name>` akey, its dataspace under the `S-<attribute name>` akey, and its value under the `V-<attribute name>` akey.

1.1.1 Groups

The purpose of HDF5 groups is to store links to other objects, and each of these is stored under a dkey, which equals the link name, with a constant akey ("Link"). Links can be hard, in which case it contains the object id for the target, which can then be directly opened, or soft, in which case the link contains the path name of the target, which must be traversed to open the target.

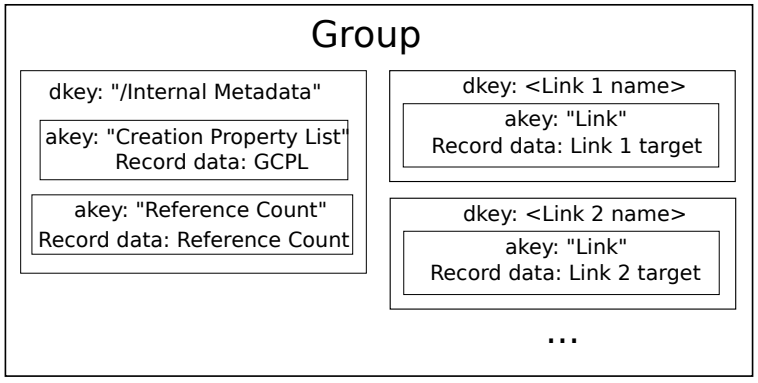

Figure 1 – Group Dkey and link examples.

1.1.2 Datasets

Datasets are used to store the bulk array data in HDF5. In addition to the creation property list, these store the datatype and dataspace under the "/Internal Metadata" dkey and the "Datatype" and "Dataspace" dkeys respectively. 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 chunk is stored in its own dkey. The dkey 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 `/` prefix for internal metadata and attributes prevents conflicts with arbitrarily named links, which cannot contain the `/` character, and the 0 prefix for chunks prevents the binary chunk dkeys from conflicting with the string metadata dkeys, since nonzero length strings cannot begin with 0 (`\0`).

For datasets with fixed length datatypes, the data for the chunk is stored in a single akey (with a value of a 0 byte), 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 chunk. For datasets with variable length datatypes, each element is stored in a separate akey, as a single record, whose size i equal to the total size of the variable length element.

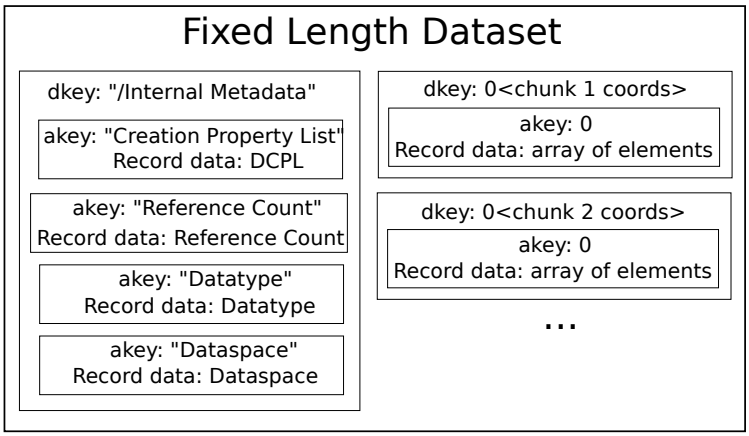

Figure 2 – Fixed Length Dataset Dkey and link examples.

1.1.3 Committed Datatypes

Committed datatypes are an HDF5 datatype that has been stored as an HDF5 object in the group structure. The only object type specific data is the datatype itself, which is stored similarly to that of datasets, under the `/Internal Metadata` dkey and `Datatype` akey.

1.1.4 Map Objects

Map objects are an HDF5 object that contains an arbitrary application-defined key-value store. The map's key and value datatypes are stored under the `/Internal Metadata` dkey, and under the `Key Datatype`

and `Value Datatype` akeys respectively. Values are stored under a dkey which is equal to the binary representation of the key, and under a constant akey (`Map Record`).

1.1.5 Object Ids

DAOS objects are referenced through the DAOS API by a 192 bit object ID. Only the lower 64 bits are currently used by HDF5, the rest of the ID is set to 0 and/or set by DAOS to encode the DAOS object class, which is always the same for each HDF5 object type. The lower 62 bits are simply set in increasing order, starting from 1, which is always the root group (0 is reserved for a global metadata object). The remaining 2 bits 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 plugin to determine the object type and therefore the routines used to access it without needed to query DAOS, reducing the number of server requests needed for metadata operations. The lower 64 bits (including the encoded object type) are considered the "address" for the purposes of the HDF5 API, and is what is returned as `addr` from `H5Oget_info()` and what is accepted for `H5Oopen_by_addr()`.

Object ID allocation is currently not handled correctly when done independently by multiple processes. This is a temporary solution until a DAOS object ID allocator is available. This means that either all objects should be created by the same process, or they should always be created collectively using `H5Pset_all_coll_metadata_ops()`.

2 HDF5 Map Objects

Add notes

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 Approach

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 plugin, meaning that map objects will only work with the DAOS plugin, and with any other VOL plugins that are written to support maps. API The HDF5 Map API will consist of 9 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`.

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 H5Mset

```
herr_t H5Mset(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);
```

`H5Mset()` 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

```
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_types

```
H5Mget_types(hid_t map_id, hid_t *key_type_id, hid_t *val_type_id);
```

`H5Mget_types()` retrieves the key and value datatype ids from the Map specified by `map_id`.

2.1.7 H5Mget_count

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

`H5Mget_count()` retrieves the number of key-value pairs stored in the Map specified by `map_id`.

2.1.8 H5Miterate

```
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.

2.1.9 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.

2.1.10 H5Mclose

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

H5Mclose() closes the Map object handle map_id.

2.1.11 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);

H5Mset(map_id, vls_type_id, &names[0], H5T_NATIVE_UINT64, &IDs[0], H5P_DEFAULT
    );
H5Mset(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

Since DAOS is built on top of key-value stores, implementation of map objects in the DAOS plugin 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 constant metadata will be stored under the `/Internal Metadata` dkey, and under the `Key Datatype`, `Value Datatype`, and `Creation Property List` akeys, respectively. 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.