Map Objects in the DAOS HDF5 VOL Plugin

Neil Fortner

3/29/2019

# Introduction

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.

# 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.

## 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.

## 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.

## 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.

## 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.

## 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.

## 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.

## 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.

## H5Miterate

H5Miterate(hid\_t map\_id, hsize\_t \*idx, hid\_t key\_mem\_type\_id, hid\_t val\_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, const void \*value, void \*op\_data)

The key and value parameters are the buffers for the key for this iteration and its associated value, converted to the datatypes specified by key\_mem\_type\_id and val\_mem\_type\_id, respectively. 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.

## 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.

## H5Mclose

herr\_t H5Mclose(hid\_t map\_id);

H5Mclose closes the Map object handle map\_id.

# 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);

# 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.

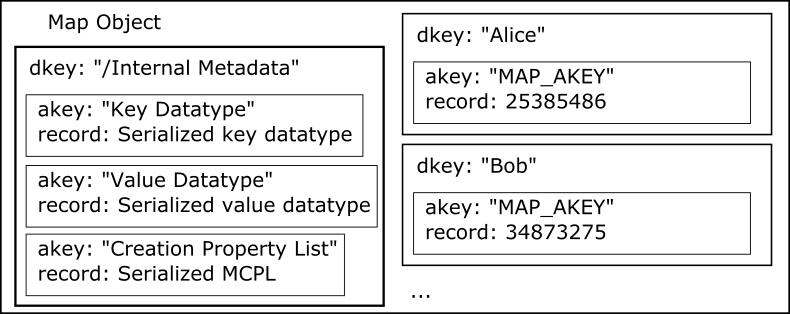


Figure - Diagram of a Map Object in DAOS as created by the above example