RFC: H5Ocompare Function

Peter Cao
Neil Fortner
Quincey Koziol

This RFC describes a new "H5Ocompare" function that compares two HDF5 objects. A set of rules for comparing two HDF5 files or two HDF5 objects have been specified in the "HDF5 File and Object Comparison Specification[1]". The purpose of this document is to provide details of how two objects should be compared and guidelines of how H5Ocompare should be implemented.

1 Introduction

An HDF5 file appears to the user as a directed graph with three higher-level objects that are exposed by the HDF5 APIs: groups, datasets, and committed datatypes. The simplicity of the HDF5 model provides great flexibility with regard to what can be put in a file. At the same time, it creates challenges in determining how to compare two files or two objects.

A command line tool, h5diff, was developed to compare two HDF5 files or objects and report the differences. The tool is one of the most used tools. However, three major issues cannot be easily resolved with the current implementation of h5diff:

- **Performance:** h5diff has poor performance in some cases. Although the performance has been greatly improved recently, some issues cannot be resolved given the current implementation. For example, uncompressing data to compare dataset values is a major bottleneck for compressed datasets; it takes over 80% of the total time. It would be far better to perform the comparison within the library on the compressed data.
- **Completeness:** h5diff was implemented with insufficient design and definition. When two objects are compared, h5diff does not provide sufficient details of what should be examined. For example, when two datasets are compared, creation properties such as storage layout are not compared. There is no clear definition of what should be compared and what should be not. One of the main goals of this document is to provide a clear and complete set of definitions for the H5Ocompare function on what to compare.
- **Complexity:** The code for h5diff is very complex. Features and options have been added to it in an *ad hoc* way making the code complex and prone to errors. Over the years, a great deal of effort has been spent fixing problems, but it is becoming increasingly difficult to maintain.

[1] https://www.hdfgroup.uiuc.edu/RFC/HDF5/tools/h5diff/h5diff_spec.pdf

The purpose of H5Ocompare is to address the problems above. Other advantages of having an H5Ocompare function include the following:

- Any tool built on H5Ocompare should have less code to maintain since the main work will be done by the function. Applications built on the function should be simple and specific.
- The function can be used in other applications such as HDFView.
- The function can be used in high-level languages such as Java, Fortran, C++, and Python. Users can program their own applications to accomplish their own goals.

We will develop a new tool that is based on H5Ocompare. The new tool is not targeted to replace the current h5diff; it is intended to address the problems of the current h5diff. The new tool will not try to mimic the output format of the current h5diff; however, some useful elements of h5diff output format will be adopted by the new tool.

We will present a number of options regarding how objects are compared in this RFC. Not all options will be implemented in the first stage, but the design of the function should allow options to be added incrementally.

2 Comparing Objects

This section describes how two HDF5 files or objects will be compared by H5Ocompare. Options, such as excluding certain metadata, can be applied to the comparison.

2.1 Files

Each HDF5 file contains a root group and file metadata such as file creation properties. Comparing two files means comparing the root groups and the metadata of the two files. The appropriate way to call the function when comparing two files is to use the "/" as the object name. For example,

```
herr_t retval = H5Ocompare(file1_id, "/", file2_id, "/",  H5P_DEFAULT, cb_info);
```

2.1.1 What is Compared

When two files are compared, the following things will be compared:

- Important (but not all) file creation properties will be checked. We will examine only the ones that are important to users and applications. This includes:
  - super block version number,
  - global freelist version number,
  - symbol table version number,
  - shared object header version number, and
  - the offsets and lengths used within the HDF5 file.
- The root groups of the two files will be compared in the same way as other groups. Group comparison is discussed in the next section.

The result of the comparison is determined by the file metadata and the contents of the root groups of the two files.

2.1.2 Options

The following options can be applied when two files are compared:

- Creation properties are excluded. By default, creation properties will be checked. However, users can choose to ignore the file creation properties for H5Ocompare.
- Options of comparing groups, which are discussed in the next section, will be applied to file comparison.

2.1.3 Special Cases

H5Ocompare should handle the following special cases of file comparison:

A) A file is compared to itself. There should be no difference if a file is compared to itself. H5Ocompare should quickly confirm that it is the same file -- without going through all the objects in the file.

B) Two identical files are compared. There should be no difference if two identical files are compared. This case is the same as (A) except that the two files are two separate physical files with exactly the same contents.

C) Two empty files are compared. A file is empty if it contains only an empty root group, and an empty root group is a group that does not have any members or links. If two empty files are compared, the result varies according to the comparison options. If file creation properties are ignored in the comparison, there should be no difference between two empty files. Otherwise, the result is determined by the file creation properties of the two files.

D) An empty file is compared with a non-empty file. The result varies according the comparison options. By default, an empty file and a non-empty file should be different. If unique objects and file properties are excluded, there will be no difference.

2.2 Groups

When two groups are compared, these items will form the basis of the comparison: group creation properties, attributes, the links under the two groups, and objects that are targets of links with the same names.

2.2.1 What is Compared

The following is a list of the things to be compared for groups:

- Group creation properties, which include
  - Link creation order
  - Link storage layout (compact or dense)
- Attributes attached to the group (details at Attribute section)
- Links under the group (details at Link section)
- Objects contained in the groups and having the same link names

2.2.2 Options

The following options can be used for comparing groups:

- Creation properties are excluded. By default, creation properties will be compared.
- Attributes are excluded. By default, attributes will be compared (see Attribute section below).
- Unique links, i.e. links with different names, are excluded. The comparison results will be determined by the links with common names and objects reached by the links. By default, unique links will be reported as different.
- Comparing only links, but not objects reached by the links. By default, all the objects reachable through the group's links are recursively compared.

2.3 Datasets

When two datasets are compared, the datasets' creation properties and attributes, as well as the data values of the two datasets will be compared.

2.3.1 What is compared

Comparing datasets means comparing both the data values and metadata of the datasets, unless an option excluding a specific comparison is given. The following things will be compared:

- Data values (see "Comparing Data Values" below)
- Dataset creation properties, which include:
    - Layout of the raw data
    - Size of chunks for the raw data of a chunked layout dataset
    - Fill value if it is defined
    - Filter(s) applied to the datasets (e.g. compression)
- Attributes attached to the dataset (details at Attribute section)
- Datatype (details at separate section)
- Dataspace (details at separate section)

2.3.2 General Options

Options for comparing datasets are listed below. Options for comparing data values are presented at separate section.

- Creation properties are excluded. By default, creation properties will be compared.
- Attributes are excluded. By default, attributes will be compared (see "Attributes" below).

2.3.3 Comparing Data Values

How the data values should be compared will depend on the datatype and dataspace for the datasets. By default, the values of two datasets will be compared only if their datatype and dataspace are the same. Options will be provided to handle special cases.

2.3.3.1 Comparing Data Values

When the dataspaces and datatypes of the two datasets being compared are the same, their values are compared point by point defined by dataspace. How the two data points are compared depends on their datatype class.

- Two integer (H5T_INTEGER) numbers of any precision can be directly compared.
- Comparing two float (H5T_FLOAT) numbers is a little more complicated. First, an EPSILON is needed for floating-point values. To determine whether two floating-point values, float1 and float2, are different, one cannot use the simple comparison of (float1 == float2). Two floating-point values can be the same while (float1 == float2) may appear to differ because of floating-point precision. In HDF5, a precision limit can be set when comparing floating point values; for details, see the "Default EPSILON Values for Comparing Floating Point Data[2]" RFC. An additional API routine for the compare property list will be needed to set the EPSILON value (we are still working on comparison property and property list functions). Second, Not-a-Number (NaN) values need to be handled. By definition, two NaNs are always equal. A NaN and a regular number are always different. Checking for NaN is very expensive. By default, H5Ocompare will check NaNs. If a user knows there are no NaNs in the datasets, they can skip checking NaN for better performance.
- When two strings (H5T_STRING) are different, i.e. strcmp(str1, str2) is non-zero, the strings will be compared character by characters and the differences will be reported.
- Two data values with type class of H5T_BITFIELD and H5T_OPAQUE will be compared byte by byte.
- Compound (H5T_COMPOUND) data will be compared by field according to the type of the field. For nested compound types, the comparison will recursively run through the nested structures.
- Each element of variable length datatype (H5T_VLEN) is a one-dimensional sequence of data values, of a particular base datatype. If the base datatypes are the same, the lengths of the one-dimensional sequences are compared, and if the lengths are the same, each data value in the sequences will be compared. If the base datatypes are the same, but the lengths are different, data values will be compared up to the shortest one. Extra values of the longer array will be reported as different.
- Values of array datatypes (H5T_ARRAY) will be compared if only their base datatypes and dimensions (rank and dimension sizes) are the same. The data values are compared based on the base datatype.
- Currently we have two types of references (H5T_REFERENCE): object references and dataset region references. When two dataset region references are compared, the data types of the datasets pointed to by the references and the values defined by the region will be compared. When two object references are compared, the two objects pointed to by the references will

[2] https://www.hdfgroup.uiuc.edu/RFC/HDF5/tools/h5diff/RFC_h5diff_default_epsilon.pdf

be compared. Options will be available so that the application can choose to not follow the references; in such a case, objects pointed by the references will not be compared.

2.3.3.2 Options for Comparing Data Values

- Datatypes can be converted if they are compatible (data can be converted by H5Tconvert). By default, datatypes will be compared. Two datasets with different datatypes will be considered as different, and their data values will be not compared. When datatypes are compatible, e.g. float and double, we sometimes want to know if there are any differences in the data values even if the datatypes of the two datasets are different.
- An option is needed to skip checking Not-a-Number (NaN). Checking for NaN is very expensive. By default, H5Ocompare will check NaNs. If we know there is no NaN in the datasets, we skip checking NaN for better performance.
- We also need an option to not compare references when two reference datatypes are compared. By default, the objects pointed to by references will be compared.
- When there are differences in two datasets, we will need an option for what level of detail we want to report. This can improve performance in some cases. For example, we could have an option just to report that the values of the two datasets are different. By skipping the details of the differences, H5Ocompare will take much less time.

2.4 Links

A link is owned by a group and the link's value points to an existing object or a non-existing object (symbolic links only). Each link has a name, a type, and a value.

2.4.1 Things to be compared

Link characteristics to be compared include:

- creation properties (character encoding and link order),
- link name,
- link type (hard, soft, external, or user-defined), and
- link value.

2.4.2 Options

Options for comparing links include the following:

- Creation properties are excluded. By default, creation properties will be checked.
- Link type is excluded. By default, link types will be compared. Links of different types, e.g. one is an external link and the other is a soft link, will be reported as different.
- Objects pointed to by symbolic links (soft links or external links) are excluded. By default, objects pointed by symbolic links will be compared. If this option is given, objects pointed to by links will not be compared.

2.5 Attributes

An attribute is like a small dataset; it has a datatype, a dataspace, and data values. Attributes are compared by name, and comparing two attributes is the same as comparing two datasets except that some dataset creation properties such as storage layout are not applied to attributes. Attribute creation properties such as creation order should be compared.

2.6 Datatypes

A datatype can describe an atomic type like a fixed- or floating-point type, or more complex types like a C structure (compound datatype), array (array datatype), or C++ vector (variable-length datatype). A datatype is defined by its class and class-specific properties.

2.6.1 What is compared

The following datatype characteristics will be compared:

- Datatype class (e.g. integer (H5T_INTEGER), float (H5T_FLOAT), string (H5T_STRING), etc.)
- Class-specific properties (e.g. size, signed or unsigned, byte order, etc.)
    - H5T_INTEGER – Size (bytes), precision (bits), offset (bits), pad, byte order, signed/unsigned
    - H5T_FLOAT -- Size (bytes), precision (bits), offset (bits), pad, byte order, and field information
    - H5T_STRING – Size (fixed or variable), character set, pad/no pad, pad character
    - H5T_BITFIELD -- Size (bytes), precision (bits), offset (bits), pad, and byte order
    - H5T_OPAQUE -- Size (bytes), tag
    - H5T_COMPOUND – Size (bytes), number of members, member names, member types, member offsets
    - H5T_REFERENCE -- Reference type (object or region)
    - H5T_ENUM -- Number of elements, element names, element values, base datatype
    - H5T_VLEN -- Base datatype
    - H5T_ARRAY -- Number of dimensions, dimension sizes, base datatype
- Attributes (for committed datatypes).

2.6.2 Options

Options for comparing committed datatypes are listed below:

- Attributes are excluded. By default, attributes will be compared (see “Attributes” section above).

2.7 Dataspace

A dataspace describes the rank and the size of each dimension in the data object array.

2.7.1 What is compared

Dataspace characteristics to be compared include:

- rank (the number of dimensions),
- current dimension sizes, and
- maximum dimension sizes.

2.7.2 Options

Options for comparing links include:

- Exclude maximum dimension size – maximum dimension sizes will not be compared. By default, they will be compared.

3 New API function

This section describes the proposed API function (H5Ocompare) and related callback functions and structures. We will continue to work on the details of the compare properties and property functions.

Name:

H5Ocompare

Signature:

*H5O_cmp_status_t H5Ocompare(hid_t loc1_id, const char *name1, hid_t loc2_id, const char *name2, hid_t ocmppl_id, H5O_cmp_cb_t *cb_info)*

Purpose:

Compare two objects in the same or different files.

Description:

H5Ocompare compares the object specified by *name1* from the file or group specified by *loc1_id* to the object specified by *name2* from the file or group specified by *loc2_id* .

Several properties are available to govern the behavior of *H5Ocompare*. These properties are set in the access property lists, *ocmppl_id* with *H5Pset_compare_object*.

The return value is the overall result of the comparison. The type of the return value is defined as:

```
typedef enum H5O_cmp_status_t {
   H5O_STATUS_ERROR  = -1,
   H5O_STATUS_EQUAL,
   H5O_STATUS_UNEQUAL,
   H5O_STATUS_EXIST_ONLY_O1,
   H5O_STATUS_EXIST_ONLY_O2,
   H5O_STATUS_UNCOMPARABLE
} H5O_cmp_status_t;
```

H5Ocompare returns *H5O_STATUS_EQUAL* if all objects are equal, *H5O_STATUS_UNEQUAL* if there exist objects which are unequal, *H5O_STATUS_UNCOMPARABLE* if there exist objects which are not comparable but none that are unequal, and *H5O_STATUS_ERROR* on error. *H5Ocompare* never returns *H5O_STATUS_EXIST_ONLY_O1* or *H5O_STATUS_EXIST_ONLY_O2*; these values are used by the callback functions described below.

Differences in the two objects are reported to callback functions, which are grouped together in a structure that is passed in as a parameter to this routine. The `cb_info` callback structure contains a list of callback functions and a pointer to user's data.

Each callback function is intended to report results for a specific object or data. Examples include:

- file metadata,
- links,
- object metadata,
- raw data differences in datasets,
- raw data and metadata differences of attributes

The *H5O_cmp_cb_t* struct is defined as:

```
typedef struct H5O_cmp_cb_t {
   H5O_cmp_file_md_cb_t    file_md;
   H5O_cmp_link_cb_t       link;
   H5O_cmp_obj_md_cb_t     obj_md;
   H5O_cmp_dset_data_cb_t dset_data;
   H5O_cmp_attr_cb_t       attr;
   void                    *udata;
} H5O_cmp_cb_t;
```

The comparison operation proceeds as follows: first, if the objects are in different files, file metadata is compared for the two files, and any differences are reported via the *file_md* callback. One call is made for each difference found. Next, the metadata of the target objects are compared, and the *obj_md* callback is called once for each difference found. Next, all of that object's attributes are compared by name, and any differences in raw data or metadata are reported via the *attr* callback, once for each difference found. If both objects are datasets, then the raw data is compared (if possible) and a list of differences is reported via the *dset_data* callback. This callback may be made more than once per dataset, depending on the options set and the number of differences. If both objects are groups, the links are compared and any differences are reported via the *link* callback. Finally, if recursion is enabled, the targets of all links that exist in both groups are compared as above, starting with the *obj_md* callback. If objects are different types, they will be reported as incomparable.

All of the callback functions include a *status* parameter with type *H5O_cmp_status_t (*as defined above). This parameter indicates what the result of the comparison is. Since the callback is not made for items that are equal or do not exist in both objects, this parameter will never have the value *H5O_STATUS_EQUAL*. The *status* parameter only needs to report when

an item is different between the two objects, exists in only one of the objects, or if it cannot be compared between the two objects.

A macro that is used in most of the callbacks is:

```
#define H5O_CMP_INFO_FIELD(type) \
   struct { \
      type     o1; \
      type     o2; \
   }
```

This macro is simply a way of listing the value of an item of type *type* in both objects using a single short line of code to make the code more readable. If the associated *H5O_cmp_status_t* enum is *H5O_STATUS_EXIST_ONLY_O1*, the value of o2 is undefined. Likewise, if the enum is *H5O_STATUS_EXIST_ONLY_O2*, the value of o1 is undefined.

The prototype of the callback function *file_md* is as follows:

*herr_t (*H5O_cmp_file_md_cb_t)(H5O_cmp_file_md_type_t type, H5O_cmp_status_t status, const H5O_cmp_file_md_info_t *cmp_info, void *udata)*

The parameters of this callback function have the following values or meanings:

type	This is the type of difference being reported. See the definition of *H5O_cmp_file_md_type_t* below.
status	This is the result of the comparison of the information specified by *type*.
cmp_info	This is the value of the information specified in *type* in both files. See the definition of *H5O_cmp_file_md_info_t* below.
udata	This is equal to the *udata* field in the *cb_info* structure passed to *H5Ocompare*. It may be used to share any application-defined data between the application and the callbacks.

The *H5O_cmp_file_md_type_t* type is an enumerated type with the following possible values:

H5O_FILE_MD_SUPERBLOCK_VERSION	Version number of the file superblock
H5O_FILE_MD_SIZEOF_ADDR	Size of addresses stored in the file (*H5Pset_sizes*)
H5O_FILE_MD_SIZEOF_SIZE	Size of lengths stored in the file (*H5Pset_sizes*)
H5O_FILE_MD_ISTORE_K	"K" value of data chunk b-trees (*H5Pset_istore_k*)
H5O_FILE_MD_SYM_IK	"K" value of group b-tree internal nodes (*H5Pset_sym_k*)
H5O_FILE_MD_SYM_LK	"K" value of group b-tree leaf nodes (*H5Pset_sym_k*)
H5O_FILE_MD_USERBLOCK_SIZE	Size of the user block (*H5Pset_userblock*)
H5O_FILE_MD_SHARED_MESG_INDEXES	Number and type of shared message

	indexes (*H5Pset_shared_mesg_nindexes*, *H5Pset_shared_mesg_index*)
H5O_FILE_MD_SHARED_MESG_MAX_LIST	Maximum number of shared messages to store in a list (*H5Pset_shared_mesg_phase_change*)
H5O_FILE_MD_SHARED_MESG_MIN_BTREE	Minimum number of shared messages to store in a b-tree (*H5Pset_shared_mesg_phase_change*)

The *H5O_cmp_file_md_info_t* type is a union whose valid field is determined by the value of the *type* argument. The definition is as follows:

```
typedef union H5O_cmp_file_md_info_t {
   H5O_CMP_INFO_FIELD(unsigned)       superblock_version;
   H5O_CMP_INFO_FIELD(unsigned)       sizeof_addr;
   H5O_CMP_INFO_FIELD(unsigned)       sizeof_size;
   H5O_CMP_INFO_FIELD(unsigned)       istore_k;
   H5O_CMP_INFO_FIELD(unsigned)       sym_ik;
   H5O_CMP_INFO_FIELD(unsigned)       sym_lk;
   H5O_CMP_INFO_FIELD(haddr_t)        userblock_size;
   struct {
      H5O_CMP_INFO_FIELD(unsigned)    nindexes;
      H5O_CMP_INFO_FIELD(H5O_cmp_shared_mesg_index_t *) indexes;
   } shared_mesg_indexes;
   H5O_CMP_INFO_FIELD(unsigned)       shared_mesg_max_list;
   H5O_CMP_INFO_FIELD(unsigned)       shared_mesg_min_btree;
} H5O_cmp_file_md_info_t;
```

Where *H5O_cmp_shared_mesg_index_t* is defined as:

```
typedef struct H5O_cmp_shared_mesg_index_t {
   unsigned    mesg_type_flags;
   unsigned    min_mesg_size;
} H5O_cmp_shared_mesg_index_t;
```

Note that the values stored in the field *shared_mesg_indexes.nindexes* determine the length of the arrays pointed to by *shared_mesg_indexes.indexes*. The *shared_mesg_indexes* field will be reported to be different by *status* if either *nindexes* or one (or more) of the elements in *indexes* is different.

The prototype of the callback function *link* is as follows:

*herr_t (*H5O_cmp_link_cb_t)(char *group_name, char *link_name, H5O_cmp_link_type_t type, H5O_cmp_status_t status, const H5O_cmp_link_info_t *cmp_info, void *udata)*

The parameters of this callback function have the following values or meanings:

group_name This is the name of the group containing the link. This is a path name relative

to the object specified in the call to *H5Ocompare*.

link_name This is the name of the link.

type This is the type of difference being reported. See the definition of *H5O_cmp_link_type_t* below.

status This is the result of the comparison of the information specified by *type*.

cmp_info This is the value of the information specified in *type* for both links. See the definition of *H5O_cmp_link_info_t* below.

udata This is equal to the *udata* field in the *cb_info* struct passed to *H5Ocompare*. It may be used to share any application-defined data between the application and the callbacks.

The *H5O_cmp_link_type_t* type is an enumerated type with the following possible values:

H5O_LINK_EXIST Used to indicate that the link only exists in one group. The object it exists in is indicated by *status*. *cmp_info* will be *NULL*. This will be the only callback made for this link.

H5O_LINK_TYPE Type of link

H5O_LINK_CORDER Creation order of link

H5O_LINK_CSET Character set of link (*H5Pset_char_encoding*)

The *H5O_cmp_link_info_t* type is a union whose valid field is determined by the value of the *type* argument. The definition is as follows:

```
typedef union H5O_cmp_link_info_t {
   H5O_CMP_INFO_FIELD(H5L_type_t)   type;
   H5O_CMP_INFO_FIELD(int64_t)      corder;
   H5O_CMP_INFO_FIELD(H5T_cset_t)   cset;
} H5O_cmp_link_info_t;
```

The prototype of the callback function *obj_md* is as follows:

*herr_t (*H5O_cmp_obj_md_cb_t)(char *name, H5O_cmp_obj_md_type_t type, H5O_cmp_status_t status, const H5O_cmp_obj_md_info_t *cmp_info, void *udata)*

The parameters of this callback function have the following values or meanings:

name This is the name of the object. This is a path name relative to the object specified in the call to *H5Ocompare*.

type This is the type of difference being reported. See the definition of *H5O_cmp_obj_md_type_t* below.

status This is the result of the comparison of the information specified by *type*.

cmp_info This is the value of the information specified in *type* in both objects. See the definition of *H5O_cmp_obj_md_info_t* below.

udata This is equal to the *udata* field in the *cb_info* structure passed to *H5Ocompare*. It may be used to share any application-defined data between the application and the callbacks.

The *H5O_cmp_obj_md_type_t* type is an enumerated type with the following possible values:

H5O_OBJ_MD_TYPE	Type of object
H5O_OBJ_MD_RC	Reference count of object
H5O_OBJ_MD_ATIME	Access time (*H5Pset_obj_track_times*)
H5O_OBJ_MD_MTIME	Modification time (*H5Pset_obj_track_times*)
H5O_OBJ_MD_CTIME	Change time (*H5Pset_obj_track_times*)
H5O_OBJ_MD_BTIME	Birth time (*H5Pset_obj_track_times*)
H5O_OBJ_MD_NUM_ATTRS	Number of attributes attached to object
H5O_OBJ_MD_HDR_VERSION	Version of object header format in file
H5O_OBJ_MD_ATTR_CRT_ORDER_FLAGS	Creation order flags for attributes (*H5Pset_attr_creation_order*)
H5O_OBJ_MD_ATTR_MAX_COMPACT	Maximum number of attributes to store in the object header (*H5Pset_attr_phase_change*)
H5O_OBJ_MD_ATTR_MIN_DENSE	Minimum number of attributes to store in dense storage (*H5Pset_attr_phase_change*)
H5O_OBJ_MD_COMMENT	Object comment (*H5Oset_comment*)
H5O_OBJ_MD_DTYPE	Datatype (for datasets and named datatypes). This comparison will not be made if one object is a group.
H5O_OBJ_MD_FILTER_PIPELINE	An object creation property list containing a copy of the object's filter pipeline (for datasets and groups; *H5Pset_filter*, etc.). No other properties are guaranteed to be valid.
H5O_OBJ_MD_SPACE	Dataspace (for datasets). This comparison will not be made if one object is not a dataset.
H5O_OBJ_MD_LAYOUT	Layout type (for datasets; *H5Pset_layout*). This comparison will not be made if one object is not a dataset.
H5O_OBJ_MD_CHUNK	Chunked layout information (for chunked datasets; *H5Pset_chunk*). This comparison will not be made if one object is not a chunked dataset.
H5O_OBJ_MD_EXTERNAL	External layout information (for external datasets; *H5Pset_external*). This comparison will not be made if one object is not an external dataset.
H5O_OBJ_MD_FILL_VALUE	Fill value (for datasets; *H5Pset_fill_value*). This comparison will not be made if one object is not a dataset.
H5O_OBJ_MD_FILL_TIME	Fill time (for datasets; *H5Pset_fill_time*). This comparison will not be made if one

	object is not a dataset.
H5O_OBJ_MD_ALLOC_TIME	Allocation time (for datasets; *H5Pset_alloc_time*). This comparison will not be made if one object is not a dataset.

The *H5O_cmp_obj_md_info_t* type is a union whose valid field is determined by the value of the *type* argument. The definition is as follows:

```
typedef union H5O_cmp_obj_md_info_t {
   H5O_CMP_INFO_FIELD(H5O_type_t)     type;
   H5O_CMP_INFO_FIELD(unsigned)       rc;
   H5O_CMP_INFO_FIELD(time_t)         atime;
   H5O_CMP_INFO_FIELD(time_t)         mtime;
   H5O_CMP_INFO_FIELD(time_t)         ctime;
   H5O_CMP_INFO_FIELD(time_t)         btime;
   H5O_CMP_INFO_FIELD(unsigned)       num_attrs;
   H5O_CMP_INFO_FIELD(unsigned)       hdr_version;
   H5O_CMP_INFO_FIELD(unsigned)       attr_crt_order_flags;
   H5O_CMP_INFO_FIELD(unsigned)       attr_max_compact;
   H5O_CMP_INFO_FIELD(unsigned)       attr_min_dense;
   H5O_CMP_INFO_FIELD(char *)         comment;
   H5O_CMP_INFO_FIELD(hid_t)          dtype;
   H5O_CMP_INFO_FIELD(hid_t)          filter_pipeline;
   H5O_CMP_INFO_FIELD(hid_t)          space;
   H5O_CMP_INFO_FIELD(H5D_layout_t)   layout;
   struct {
      H5O_CMP_INFO_FIELD(int)         ndims;
      H5O_CMP_INFO_FIELD(hsize_t *)   dim;
   } chunk;
   struct {
      H5O_CMP_INFO_FIELD(int)         count;
      H5O_CMP_INFO_FIELD(H5O_cmp_external_t *) external;
   } external;
   struct {
      H5O_CMP_INFO_FIELD(hid_t)       dtype;
      H5O_CMP_INFO_FIELD(void *)      value;
   } fill_value;
   H5O_CMP_INFO_FIELD(H5D_fill_time_t) fill_time;
   H5O_CMP_INFO_FIELD(H5D_alloc_time_t) alloc_time;
} H5O_cmp_obj_md_info_t;
```

Where *H5O_cmp_external_t* is defined as:

```
typedef struct H5O_cmp_external_t {
   char        *name;
   off_t       offset;
   hsize_t     size;
} H5O_cmp_shared_mesg_index_t;
```

Note that the values stored in the field *chunk.ndims* determine the length of the arrays pointed to by *chunk.dim*. The *chunk* field will be reported to be different by *status* if either *ndims* or one (or more) of the elements in *dim* is different.

Likewise, the values stored in the field *external.count* determine the length of the arrays pointed to by *external.external*. The *external* field will be reported to be different by *status* if either *count* or one (or more) of the elements in *external* is different.

The field *fill_value.dtype* contains the datatype IDs for the two datasets, and therefore indicates how to interpret the data stored in *fill_value.value*, as well as the size of the data. The fill values are compared according to the same rules as raw data comparison, and status reports only on the equality of the fill values (not the datatypes). If the datatypes are not compatible, *status* will be *H5O_STATUS_UNCOMPARABLE*.

Note that *dtype* may be reported as different even if the datatypes are compatible.

The prototype of the callback function *dset_data* is as follows:

*herr_t (*H5O_cmp_dset_data_cb_t)(char *name, H5O_cmp_status_t status, const H5O_cmp_dset_data_info_t *cmp_info, void *udata)*

The parameters of this callback function have the following values or meanings:

name	This is the name of the dataset. This is a path name relative to the root object specified in the call to *H5Ocompare*.
status	This is the result of the comparison of the raw data in the datasets. This will be *H5O_STATUS_UNEQUAL* if some of the raw data values are different or *H5O_STATUS_UNCOMPARABLE* if the datatype or dataspace are not compatible. Datatypes are incompatible if one cannot be converted to the other. Dataspaces are incompatible if their ranks or extents differ. Options may be added in the future to strengthen compatibility requirements for datatypes (for example to require all fields in a compound be present in both datatypes) or relax compatibility requirements for dataspaces (for example to allow comparison as long as the total number of elements is the same).
cmp_info	This contains the data differences as well as information needed to interpret these differences. See the definition of *H5O_cmp_dset_data_info_t* below.
udata	This is equal to the *udata* field in the *cb_info* structure passed to *H5Ocompare*. It may be used to share any application-defined data between the application and the callbacks.

The *H5O_cmp_dset_data_info_t* type is a struct whose definition is as follows:

```
typedef struct H5O_cmp_dset_data_info_t {
   H5O_CMP_INFO_FIELD(hid_t)  dtype;
   H5O_CMP_INFO_FIELD(hid_t)  space;
   unsigned                   ndiffs;
   H5O_CMP_INFO_FIELD(H5O_cmp_dset_data_diff_t *) diff;
```

} H5O_cmp_dset_data_info_t;

Where *H5O_cmp_dset_data_diff_t* is defined as:

```
typedef struct H5O_cmp_dset_data_diff_t {
   hsize_t     *offset;
   void        *value;
} H5O_cmp_dset_data_diff_t;
```

dtype contains the datatype IDs for both datasets, and *space* contains the dataspace IDs for both datasets. *ndiffs* contains the number of differences reported by this call, which may be less than the total number of differences. This callback may be called more than once in order to report all the differences in a dataset. Options may be added to adjust the maximum number of differences reported per callback as well as the maximum total number of differences reported per dataset. *ndiffs* indicates the length of the arrays pointed to by *diff*.

diff points to arrays of matched elements that are different between the two datasets. For example, *diff.o1[0]* is matched against *diff.o2[0]*, and its inclusion in this list implies that the data stored in *diff.o1[0].value* is different from that in *diff.o2[0].value*. The *offset* field denotes the logical offset of the element within the dataset. With default options, *diff.o1[x].offset* will always be equal to *diff.o2[x].offset*. The length of *offset* is determined by the number of dimensions in the appropriate *space*.

The prototype of the callback function *attr* is as follows:

*herr_t (*H5O_cmp_attr_cb_t)(char *obj_name, char *attr_name, H5O_cmp_attr_type_t type, H5O_cmp_status_t status, const H5O_cmp_attr_info_t *cmp_info, void *udata)*

The parameters of this callback function have the following values or meanings:

obj_name	This is the name of the object containing the attribute. This is a path name relative to the object specified in the call to *H5Ocompare*.
attr_name	This is the name of the attribute.
type	This is the type of difference being reported. See the definition of *H5O_cmp_attr_type_t* below.
status	This is the result of the comparison of the information specified by *type*.
cmp_info	This is the value of the information specified in *type* for both links. See the definition of *H5O_cmp_attr_info_t* below.
udata	This is equal to the *udata* field in the *cb_info* structure passed to *H5Ocompare*. It may be used to share any application-defined data between the application and the callbacks.

The *H5O_cmp_attr_type_t* type is an enumerated type with the following possible values:

H5O_ATTR_EXIST	Used to indicate that the attribute only exists in one object. The object it exists in is indicated by *status*. *cmp_info* will be *NULL*.

H5O_ATTR_DTYPE	Datatype of attribute
H5O_ATTR_SPACE	Dataspace of attribute
H5O_ATTR_DATA	Attribute raw data

The *H5O_cmp_attr_info_t* type is a union whose valid field is determined by the value of the *type* argument. The definition is as follows:

```
typedef union H5O_cmp_attr_info_t {
   H5O_CMP_INFO_FIELD(hid_t)          dtype;
   H5O_CMP_INFO_FIELD(hid_t)          space;
   H5O_cmp_dset_data_info_t           data;
} H5O_cmp_attr_info_t;
```

The *data* field uses the same format as described above for the *cmp_info* parameter for the *dset_data* callback.

Parameters:

hid_t `loc1_id`	IN: Location identifier of the first object to be compared
const char *`name1`	IN: Name of the first object to be compared
hid_t `loc2_id`	IN: Location identifier of the second object to be compared
const char *`name2`	IN: Name of the second object to be compared
hid_t `ocmppl_id`	IN: Object compare property list identifier
*H5O_cmp_cb_t *cb_info*	IN/OUT: A callback structure that contains a list of callback functions and a pointer to user's data for reporting the results of comparison.

Returns:

Returns *H5O_STATUS_EQUAL* if all objects are equal, *H5O_STATUS_UNEQUAL* if some objects are unequal, *H5O_STATUS_UNCOMPARABLE* if some objects are not comparable but none are unequal, or *H5O_STATUS_ERROR* (*-1*) on error.

4 Examples

This section presents a few examples to help you understand how the H5Ocompare() function can be used for different purposes.

4.1 Example 1: Check File Metadata

In this example, we will compare the differences in the file metadata for two files and print the results to the standard output stream, stdout.

4.1.1 Step 1: implement the callback function for reporting results, *H5O_cmp_file_md_cb_t*,

```
herr_t  compare_file_superblock_cb (H5O_cmp_file_md_type_t type, H5O_cmp_status_t status, const
H5O_cmp_file_md_info_t *cmp_info, UNUSED void *udata)
{
   if (status == H5O_STATUS_UNEQUAL && cmp_info)
   {
      printf("\n");

      switch (type) {
      case H5O_FILE_MD_SUPERBLOCK_VERSION:
         printf("Superblock version (file1, file2): \t%d\t\%d\n",
               cmp_info->superblock_version.o1, cmp_info->superblock_version.o2);
         break;
      case H5O_FILE_MD_SIZEOF_ADDR:
         printf("Size of addresses(file1, file2):: \t%d\t%d\n",
               cmp_info->sizeof_addr.o1, cmp_info->sizeof_addr.o2);
         break;
      case H5O_FILE_MD_SIZEOF_SIZE:
         printf("Size of lengths(file1, file2): \t%d\t%d\n",
               cmp_info->sizeof_size.o1, cmp_info->sizeof_size.o2);
         break;
      case H5O_FILE_MD_ISTORE_K:
         printf("\"K\" value of data chunk b-trees in file one: \t%d\t%d\n",
              cmp_info->istore_k.o1, cmp_info->istore_ke.o2);
         break;
      case H5O_FILE_MD_SYM_IK:
         printf(""\"K\" value of group b-tree internal nodes (file1, file2): \t%d\t%d\n",
              cmp_info->sym_ik.o1, cmp_info->sym_ik.o2);
         break;
      case H5O_FILE_MD_SYM_LK:
         printf("\"K\" value of group b-tree leaf nodes (file1, file2): \t%d\t%d\n",
              cmp_info->sym_ik.o1, cmp_info->sym_ik.o2);
         break;
      case H5O_FILE_MD_USERBLOCK_SIZE:
         printf("Size of the user block(file1, file2): \t%d\t%d\n",
              cmp_info->userblock_size.o1, cmp_info->userblock_size.o2);
         break;
```

```
    case H5O_FILE_MD_SHARED_MESG_INDEXES:
      printf("Number of shared messages(file1, file2): \t%d\t%d\n",
           cmp_info->shared_mesg_indexes.nindexes.o1, cmp_info->shared_mesg_indexes.nindexes.o2);

      printf("Shared message indexes in file1 (mesg type, mesg size): ");
      for (i=0; i<cmp_info->shared_mesg_indexes.nindexes.o1; i++)
        prinft("(%d, %d), ", cmp_info->shared_mesg_indexes.indexes[i].o1.mesg_type_flags,
             cmp_info->shared_mesg_indexes.indexes[i].o1.min_mesg_size);

      printf("Shared message indexes in file2 (mesg type, mesg size): ");
      for (i=0; i<cmp_info->shared_mesg_indexes.nindexes.o2; i++)
        prinft("(%d, %d), ", cmp_info->shared_mesg_indexes.indexes[i].o2.mesg_type_flags,
             cmp_info->shared_mesg_indexes.indexes[i].o2.min_mesg_size);

      break;
    case H5O_FILE_MD_SHARED_MESG_MAX_LIST:
      printf("Maximum number of shared messages to store in a list(file1, file2): \t%d\t%d\n",
           cmp_info->shared_mesg_max_list.o1, cmp_info->shared_mesg_max_list.o2);
      break;
    case H5O_FILE_MD_SHARED_MESG_MIN_BTREE:
      printf("Minimum number of shared messages to store in a b-tree(file1, file2): \t%d\t%d\n",
           cmp_info->shared_mesg_min_btree.o1, cmp_info->shared_mesg_min_btree.o2);
      break;
    }
  }

  return SUCCEED;
}
```

4.1.2 Step2: call H5Compare() to check the file metadata

```
#include "hdf5.h"

/* usage: PROG_NAME src_file src_obj dst_file */
int main(int argc, char *argv[]) {
   hid_t file1=-1, file2=-1;
   H5O_cmp_cb_t cb_info;

   if (argc < 3) {
      puts("USAGE: PROG_NAME file1 file2\n");
      return 1;
   }

   file1 = H5Fopen(argv[1], H5F_ACC_RDONLY, H5P_DEFAULT);
   file2 = H5Fopen(argv[2], H5F_ACC_RDONLY, H5P_DEFAULT);;

   memset(&cb_info, 0, sizeof(H5O_cmp_cb_t);
   file_md.file_md = compare_file_superblock_cb;
   H5Ocompare(file1, ".", file2, ".", H5P_DEFAULT, &cb_info)

   H5Fclose(file1);
   H5Fclose(file2);

   return 0;
}
```

4.2 Example 2: Check Data Values of Two Datasets

This example shows how to compare the values of two datasets. Instead of printing the results, the callback function returns the differences of the dataset values to the caller.

4.2.1 Step 1: implement the callback function

```
typedef struct compare_dset_cb_info_t {
   size_t     buf_size[2];   /* buf_size[0]=original buf size, buf_size[1]=actual buf size */
   hsize_t    *offset_dset1;  /* OUT */
   void       *value_dset1;   /* OUT */
   hsize_t    *offset_dset2;  /* OUT */
   void       *value_dset2;   /* OUT */
} compare_dset_cb_info_t;


herr_t compare_dset_data_cb (char *name, H5O_cmp_status_t status,
      const H5O_cmp_dset_data_info_t *cmp_info, void *udata)
{
   compare_dset_cb_info_t *info = (compare_dset_cb_info_t *)udata;

   if (status == H5O_STATUS_UNEQUAL && cmp_info)
   {
      info->buf_size[1] = cmp_info->ndiffs;

      if (info->buf_size[0]<cmp_info->ndiffs) {
         offset_dset1 = (hsize_t*) realloc (offset_dset1, cmp_info->ndiffs * sizeof(hsize_t));
         offset_dset2 = (hsize_t*) realloc (offset_dset2, cmp_info->ndiffs * sizeof(hsize_t));
         value_dset1  = realloc (value_dset1, cmp_info->ndiffs * H5Tget_size(cmp_info->dtype.o1));
         value_dset2  = realloc (value_dset2, cmp_info->ndiffs * H5Tget_size(cmp_info->dtype.o2));
      }

      memcpy(offset_dset1, info->diff->o1.offset, cmp_info->ndiffs * sizeof(hsize_t));
      memcpy(offset_dset2, info->diff->o2.offset, cmp_info->ndiffs * sizeof(hsize_t));
      memcpy(value_dset1,  info->diff->o1.value,  cmp_info->ndiffs * H5Tget_size(cmp_info->dtype.o1));
      memcpy(value_dset2,  info->diff->o2.value,  cmp_info->ndiffs * H5Tget_size(cmp_info->dtype.o2));
   }

   return SUCCEED;
};
```

4.2.2 Step2: call the H5Ocompare() to get the differences of the values

```
#include "hdf5.h"

/* usage: PROG_NAME src_file dst_file src_obj dst_obj */
int main(int argc, char *argv[]) {
   hid_t file1=-1, file2=-1, did=-1;
   H5O_cmp_cb_t cb_info;
   compare_dset_cb_info_t udata;

   if (argc < 5) {
      puts("USAGE: PROG_NAME file1 file2 dset1 dset2\n");
      return -1;
   }

   file1 = H5Fopen(argv[1], H5F_ACC_RDONLY, H5P_DEFAULT);
   file2 = H5Fopen(argv[2], H5F_ACC_RDONLY, H5P_DEFAULT);

   /* set up the user data that is passed to the callback */
   udata.buf_size[0] = 10;
   offset_dset1 = (hsize_t *)calloc(udata.buf_size[0], sizeof(hsize_t));
   offset_dset2 = (hsize_t *)calloc(udata.buf_size[0], sizeof(hsize_t));

   did = H5Dopen2(file1, argv[3], H5P_DEFAULT);
   tid = H5Dget_type(did);
   value_dset1 = calloc(udata.buf_size[0], H5Tget_size(tid)));
   H5Tclose(tid);
   H5Dclose(did);

   did = H5Dopen2(file1, argv[4], H5P_DEFAULT);
   tid = H5Dget_type(did);
   value_dset2 = calloc(udata.buf_size[0], H5Tget_size(tid)));
   H5Tclose(tid);
   H5Dclose(did);

   memset(&cb_info, 0, sizeof(H5O_cmp_cb_t);
   cb_info.dset_data = compare_dset_data_cb;
   cb_info.udata = &udata;

   /* call H5Ocompare() to check the data values */
   H5Ocompare(fil1, argv[3], file2, argv[4], H5P_DEFAULT, &cb_info)

   H5Fclose(file1);
   H5Fclose(file2);

   if (udata.buf_size[1] == 0)
      puts("No difference in data values.");
   else {
     /* do something with the results */
   }
....
   return 0;
}
```

Revision History

January 12, 2011:	Version 1 circulated for comment within The HDF Group.
January 20, 2011:	Version 2 revised with Quincey's and Neil's feedback.
February 4, 2011:	Version 3 added more details on how to compare objects.
March 16, 2011:	Version 4 added details for H5Ocompare() function and examples.