Table of Contents

1. HDF5 Query and Indexing

Accessing, selecting and retrieving data from an HDF5 container can be a time-consuming process, particularly so when data is very large. To enable, ease and accelerate this process, we introduce a set of extensions to the HDF5 library that enables efficient query, selection, and indexing of data. This document defines the *H5Q query* and *H5X indexing* interfaces as well as a *view* structure that can be used to store and retrieve query results.

When working on large datasets, finding and selecting the interesting pieces of the data can be a cumbersome process. Currently, the HDF5 library enables the application developer to select, read and write data but does not provide any mechanism to select and retrieve pieces without prior knowledge of its content, or without the developer to provide the exact data coordinates that he is willing to access. To satisfy that need, one must be able to issue queries by specifying a data selection criteria. These queries, applied to the data, can then generate a selection, which in turn contains a set of coordinates satisfying the query. However, generating a selection may actually imply accessing the data. To accelerate and facilitate that process (i.e., so that the actual data no longer needs to be accessed), one can also generate indexes and use these indexes to directly answer the specified query, by finding the coordinates of the matching elements directly from the index.
The functionality of HDF5 has been expanded to enable application developers to create complex and high-performance queries on both metadata and data elements within an HDF5 container and retrieve the results of applying those query operations. Support for these operations can be defined via:

- New query objects[1] and API routines, enabling the construction of query requests for execution on HDF5 containers;
- New index objects and API routines, which allows the creation of indexes on the contents of HDF5 containers, to improve query performance.

1.1. Query Objects

Query objects are the foundation of the data analysis operations in HDF5 and can be built up from simple components in a programmatic way to create complex operations using *Boolean* operations. The current API is presented below:

```
/* Function prototypes */
hid_t  H5Qcreate(H5Q_type_t query_type, H5Q_match_op_t match_op, ...);
herr_t H5Qclose(hid_t query_id);
hid_t  H5Qcombine(hid_t query1_id, H5Q_combine_op_t combine_op, hid_t query2_id);
herr_t H5Qget_type(hid_t query_id, H5Q_type_t *query_type);
herr_t H5Qget_match_op(hid_t query_id, H5Q_match_op_t *match_op);
herr_t H5Qget_components(hid_t query_id, hid_t *subquery1_id, hid_t *subquery2_id);
herr_t H5Qget_combine_op(hid_t query_id, H5Q_combine_op_t *op_type);
herr_t H5Qencode(hid_t query_id, void *buf, size_t *nalloc);
hid_t  H5Qdecode(const void *buf);
```

[1] Query objects are in-memory objects, which, therefore, do not modify the content of the container.

1.2. Query Creation

The core query API is composed of two routines: `H5Qcreate` and `H5Qcombine`. `H5Qcreate` creates new queries, by specifying an aspect of an HDF5 container, such as:

- `H5Q_TYPE_DATA_ELEM` (data element);
- `H5Q_TYPE_LINK_NAME` (link name);
- `H5Q_TYPE_ATTR_NAME` (attribute name);
- `H5Q_TYPE_ATTR_VALUE` (attribute value);

as well as a match operator, such as:

- `H5Q_MATCH_EQUAL (=);`
- `H5Q_MATCH_NOT_EQUAL (≠);`
- `H5Q_MATCH_LESS_THAN (≤);`
- `H5Q_MATCH_GREATER_THAN (≥);`

and a value for the match operator. Created query objects can be serialized and deserialized using the `H5Qencode` and `H5Qdecode` routines[2], and their content can be retrieved using the corresponding accessor routines. `H5Qcombine` combines two query objects into a new query object, using Boolean operators such as:

- `H5Q_COMBINE_AND (∧);`
- `H5Q_COMBINE_OR (∨);`

Queries created with `H5Qcombine` can be used as input to further calls to `H5Qcombine`, creating more complex queries. For example, a single call to `H5Qcreate` could create a query object that would match data elements in any dataset within the container that is equal to the value "`17`". Another call to `H5Qcreate` could create a query object that would match link names equal to "`Pressure`"
Calling H5Qcombine with the ∧ operator and those two query objects would create a new query object that matched elements equal to "`17`" in HDF5 datasets with link names equal to "`Pressure`".
Creating the data analysis extensions to HDF5 using a *programmatic interface* for defining queries avoids defining a text-based query language as a core component of the data analysis interface and is more in keeping with the design and level of abstraction of the HDF5 API.

The HDF5 data model is more complex than traditional database tables, and a simpler query model would likely not be able to express the kinds of queries needed to extract the full set of components of an HDF5 container. A text-based query language (or GUI) could certainly be built on top of the query API defined here to provide a more user-friendly (as opposed to a *developer-friendly*) query syntax like "`Pressure = 17`". However, we regard this as out-of-scope for now.

1.3. Query Execution and Views

Applying a query to an HDF5 container creates an HDF5 *view.* A view is a temporary memory container. It is composed of a group that contains 1-D dataset objects. These objects contain references to the matching elements that were queried. There can be three different types of references, therefore there are up to

[2] Serialization/Deserialization of queries were introduced so that queries can be sent through the network.

three 1-D dataset objects with references into the contents of the HDF5 container that the query was applied to: one with *object* references, one with *region* references and one with *attribute* references. A new view is created by the following `H5Qapply` routine, which applies a query to an HDF5 container, group hierarchy, or individual object and returns the object ID of the newly created group.

```
hid_t H5Qapply(hid_t loc_id, hid_t query_id, unsigned *result, hid_t vcpl_id);
```

The attributes, objects, and/or data regions referenced within a view's datasets can be retrieved by further HDF5 dataset (H5D*) API calls.

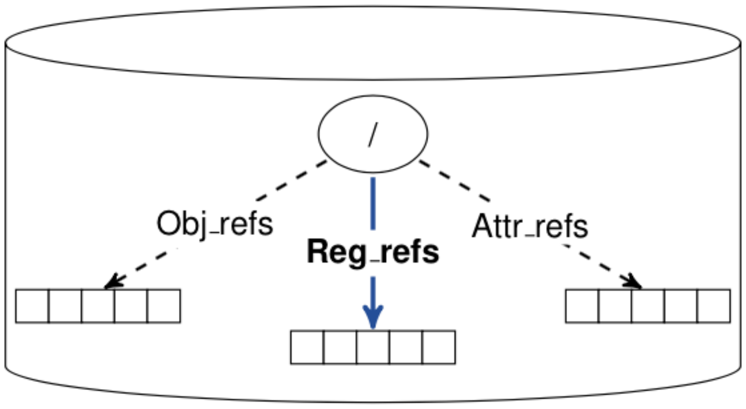


Figure 1: Temporary memory container with datasets containing query results (i.e., HDF5 references)

Although the created view is stored in memory, it can be persisted by calling `H5Ocopy()` to copy the group (stored in a virtual container) to a persistent container. For coherency, a timestamp may also be attached to it so that its states has a meaning compared to the state of the container that the query was applied to (as the container may have been modified in the meantime). It may be useful in the future to define different states to the view (*dead* or a*live*) so that the user knows whether the view is current or not.

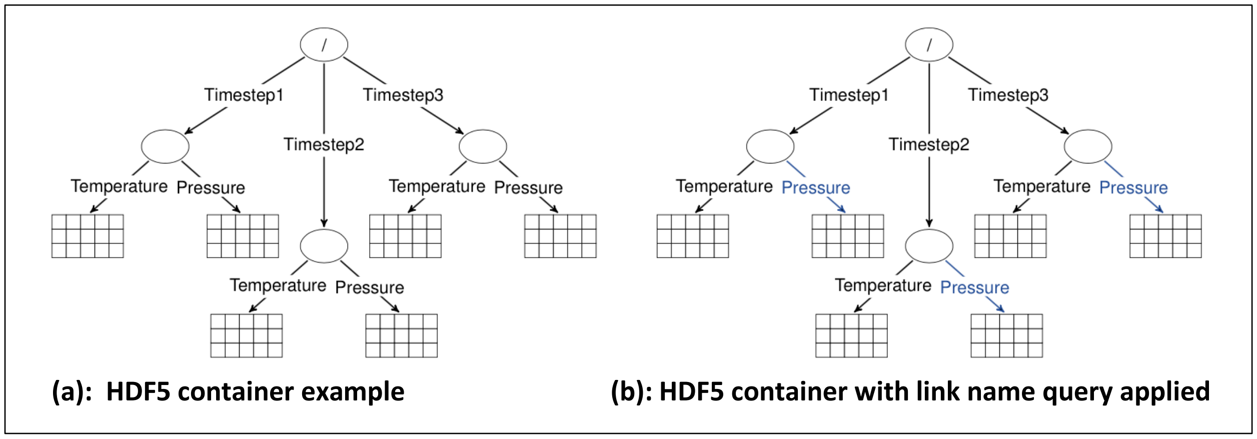
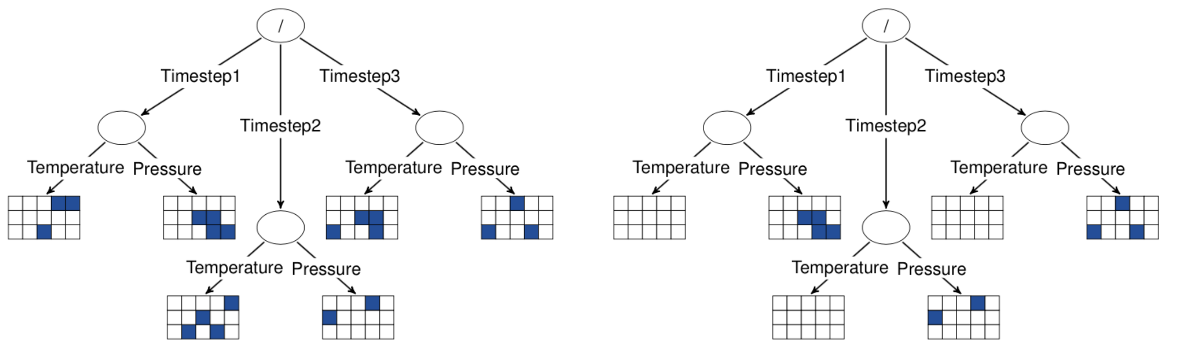

/
Timestep1
Timestep2
Timestep3
Temperature
Pressure
(a): HDF5 container example
(b): HDF5 container with link name query applied

Figure 2: HDF5 view creation example with simple query

For example, starting with the HDF5 container described in Figure 2a, applying a ``link_name=Pressure” query would result in the view shown in Figure 2b, highlighted in blue.

Since views contain a set of HDF5 references (object, dataset region or attribute references) to components of the underlying container, they retain the context of the original container. For example, the view containing the results in Figure 3 will contain three dataset region references, which can be retrieved from the view object and probed for the dataset and selection containing the elements that match the query with the existing `H5Rdereference` and `H5Rget_region` API calls. Note that selections returned from a region reference retain the underlying dataset's dimensionality and coordinates---they are not *flattened* into a 1-D series of elements. The selection returned from a region reference can also be applied to a different dataset in the container, allowing a query on pressure values to be used to extract temperature values, for example.



(a): Container with data element query applied **(b): HDF5 container with combine query applied**

Figure 3: HDF5 view creation example with a combined query

Table 1 below describes the result types for atomic queries and combining queries of different types. Query results of *None* type are rejected when `H5Qcombine` is called, causing it to return failure[3].

Table 1: Query combinations and associated result reference type.

H5Q_TYPE_DATA_ELEM	Dataset Region
H5Q_TYPE_ATTR_VALUE	Attribute
H5Q_TYPE_ATTR_NAME}	Attribute
H5Q_TYPE_LINK_NAME	Object
Dataset Element $\wedge$ Dataset Element	Dataset Region
Dataset Element $\wedge$ Attribute	Dataset Region
Dataset Element $\wedge$ Object	Dataset Region
Attribute $\wedge$ Attribute	Attribute
Attribute $\wedge$ Object	Object
Object $\wedge$ Object	Object
Dataset Element $\vee$ Dataset Element	Dataset Region
Dataset Element $\vee$ Attribute	Combination
Dataset Element $\vee$ Object	Combination
Dataset Element $\vee$ Combination	Combination

[3] Query results of *None* type may be implemented with another result type in the future, once additional experience with the query framework is acquired.

Attribute $\vee$ Attribute	Attribute
Attribute $\vee$ Object	Combination
Attribute $\vee$ Combination	Combination
Object $\vee$ Object	Object
Object $\vee$ Combination	Combination
Combination $\vee$ Combination	Combination
Combination $\wedge$ Dataset Element	None
Combination $\wedge$ Attribute	None
Combination $\wedge$ Object	None
Combination $\wedge$ Combination	None

1.3.1. Cross Container Queries

In the case where multiple containers (files, locations) need to be queried, a single query operation may be used instead of performing individual queries on each container. Performing queries across multiple containers with a single operation can enable analysis operations to generate an aggregated view of query results more easily, as well as enable a silent execution of the query in parallel, thereby reducing the time to generate a view from multiple containers.

The first extension to the previously described `H5Qapply` call is defined with the following `H5Qapply_multi` routine:

```
hid_t H5Qapply_multi(size_t loc_count, hid_t loc_ids[], hid_t query_id,
  unsigned *result, hid_t vcpl_id);
```

Location IDs specified may be entire containers, groups in a container's hierarchy or individual datasets within a container These locations may reside within the same container, or in multiple containers. Applying cross-container queries creates a view that contains references to external locations (as opposed to the previous `H5Qapply` call where references were internal). Therefore, the extended HDF5 reference object also includes a container identifier so that, when the view result is given back to the user, a subsequent call to `H5Rdereference` can return a reference to an object within an externally referenced container[4].

1.4. Index Objects

Index objects are designed to accelerate the creation of views from query operations. For example, if the previously described ``(`link_name = Pressure`) $\wedge$ (`data_element = 17`)'' query is going to be either frequently executed or executed on a container that contains significantly large datasets, indexes could be created in that container, which would speed up the creation of views when querying for link names and for data element values. Indexes created for accelerating the ``(`link_name = Pressure`)'' or ``(`data\_element}$ = 17`)'' queries would also improve view creation for the more complex query. We distinguish two different types of indexes: *data* indexes and *metadata* indexes. Data indexes apply to dataset elements, which represent the largest volume of data in typical HPC application usage of HDF5, whereas metadata indexes apply to link or attribute name components of the queries.

[4] Please refer to the H5R reference RFC.

1.4.1. Indexing Interface

This interface is defined for adding third-party indexing plugins, such as Fastbit, ALACRITY, etc.
The interface provides indexing plugins with efficient access to the contents of the container for both the creation and the maintenance of indexes. In addition, the interface allows third-party plugins to create private data structures within the container for storing the contents of the index. The current API, as well as the plugin interface, are presented below:

```
/* Function prototypes */
herr_t  H5Xregister(const H5X_class_t *idx_class);
herr_t  H5Xunregister(unsigned plugin_id);
herr_t  H5Xcreate(hid_t loc_id, unsigned plugin_id, hid_t xcpl_id);
herr_t  H5Xremove(hid_t loc_id, unsigned n /* Index n to be removed */);
herr_t  H5Xget_count(hid_t loc_id, hsize_t *idx_count);
hsize_t H5Xget_size(hid_t loc_id);

/* Index type */
typedef enum {
    H5X_TYPE_DATA,
    H5X_TYPE_METADATA
} H5X_type_t;

/* Index class */
typedef struct {
    unsigned version;      /* Version number of the index plugin class struct */
                           /* (Should always be set to H5X_CLASS_VERSION, which
                            *  may vary between releases of HDF5 library) */
    unsigned id;           /* Index ID (assigned by The HDF Group, for now) */
    const char *idx_name;  /* Index name (for debugging only, currently) */
    H5X_type_t type;       /* Type of data indexed by this plugin */

    /* Callbacks */
    union {                /* Union of callback index structures */
        H5X_data_class_t data_class;
        H5X_metadata_class_t metadata_class;
    } idx_class;
} H5X_class_t;
```

1.4.2. Data Indexing

The data indexing API can work in conjunction with the view creation. When an `H5Qapply` call is made on a given location, an index attached to any dataset queried for element value ranges will be used to speed up the query process and return a dataspace selection to the library for later use.
There are different techniques for creating data element indexes, and the most efficient method will vary depending on the type of data that is to be indexed, its layout, etc. A new interface for the HDF5 library that uses a plugin mechanism is therefore defined.

```
/* Data index class */
typedef struct {
    void *(*create)(hid_t dataset_id, hid_t xcpl_id, hid_t xapl_id,
        size_t *metadata_size, void **metadata);
    herr_t (*remove)(hid_t dataset_id, size_t metadata_size, void *metadata);
    void *(*open)(hid_t dataset_id, hid_t xapl_id, size_t metadata_size,
        void *metadata);
    herr_t (*close)(void *idx_handle);
    herr_t (*copy)(hid_t src_dataset_id, hid_t dest_dataset_id, hid_t xcpl_id,
        hid_t xapl_id, size_t src_metadata_size, void *src_metadata,
        size_t *dest_metadata_size, void **dest_metadata);
    herr_t (*pre_update)(void *idx_handle, hid_t dataspace_id, hid_t xxpl_id);
    herr_t (*post_update)(void *idx_handle, const void *buf, hid_t
        dataspace_id, hid_t xxpl_id);
    herr_t (*query)(void *idx_handle, hid_t query_id, hid_t xxpl_id,
        hid_t *dataspace_id);
    herr_t (*refresh)(void *idx_handle, size_t *metadata_size, void **metadata);
    herr_t (*get_size)(void *idx_handle, hsize_t *idx_size);
} H5X_data_class_t;
```

Index objects are stored in the HDF5 container that they apply to but are not visible in the container's group hierarchy[5]. Instead, index objects are part of the metadata for the indexed dataset. New index objects are created by passing an HDF5 location (group or dataset) to be indexed and the index plugin ID to the `H5Xcreate` call. Alternatively, an index may be created at the same time as a dataset gets created by passing a property to the dataset creation property list. Index information (such as plugin ID and index metadata) is stored at index creation time[6] and when the user later calls H5Dopen, the plugin open callback will retrieve this stored information and make use of the corresponding index plugin for all subsequent operations[7]. Similarly, calling `H5Dclose` will call the plugin index close callback and close the objects used to store the index data.

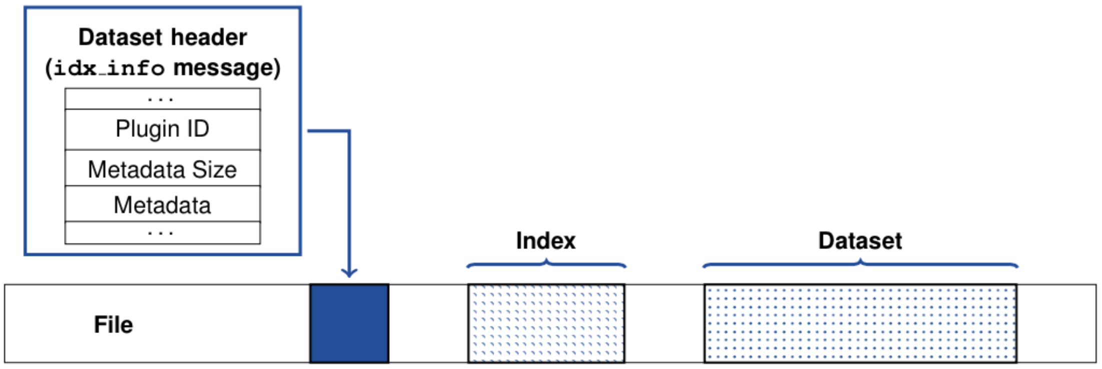


Figure 4: Index information (plugin ID and metadata) is stored along with the object header

When a call to `H5Dwrite` is made, the index plugin `pre_update` and `post_update` callbacks will be triggered, allowing efficient index update by first telling the index plugin the region that is going to be updated with new data, and then realizing the actual index update, after the dataset write has completed.

[5] Plugin developers, note that the HDF5 library's existing anonymous dataset and group creation calls can be used to create objects in HDF5 files that are not visible in the container's group hierarchy.
[6] Adding index information introduces a file format change.
[7] Note that the dataset's index object is opened only when a query call is made on the dataset.

This allows various optimization to be made, depending on the data selection passed and the index plugin used. For example, a plugin could store the region and defer the actual index update until the dataset is closed, hence saving repeated index computation/update calls.

When a call to H5Qapply is made, the index plugin query callback will be invoked to create a selection of elements in the dataset that match the query parameters. Applications can also directly use the internally called H5Dquery routine defined below to directly execute a query on a particular dataset (accelerated by any index defined on the dataset) and retrieve the selection that matches the query.

```
hid_t H5Dquery(hid_t dset_id, hid_t space_id, hid_t query_id, hid_t xapl_id);
```

Because the amount of space taken by the index cannot be directly retrieved by the user (since the datasets storing the indexes are known only by the plugin itself), the get_size callback can query the amount of space that the index takes in the file and users may correspondingly query that information using the H5Xget_size routine.

1.4.3. MetaData Indexing

Metadata indexing accelerates HDF5 metadata query operations, improving the speed of data analysis operations. Currently, metadata query operations in HDF5 are performed by traversing the contents of the container specified in the query, potentially generating a large amount of I/O and taking significant time to create the query results. Adding a mechanism for storing metadata indexes to HDF5 containers can accelerate query operations by avoiding these expensive traversals, as is available for data element queries.

A metadata index is a data structure that tracks a particular aspect of an HDF5 container and can quickly answer queries about that aspect. For example, a metadata index can be created that tracks all the link names for objects in a container, updating the index whenever a link is added, modified or removed. This index can then be applied to queries on link names, avoiding a traversal over all the links for groups in the container.

1.4.4. Metadata Indexing Interface

The interface for metadata indexes in HDF5 containers is modeled after the interface for data element indexes in HDF5, with a plugin architecture that allows new metadata index packages to be developed and added at application runtime, without modifications to the core HDF5 library.

The HDF5 library can then interact with the third-party indexes created in this manner through a set of callback interfaces that present query information to the indexing package and retrieve results from it.
An HDF5 metadata index can be considered as a generic key-value store, in the sense that for each *key* that can, for instance, represent either a link *attribute name* and/or an *attribute value*, is associated an HDF5 object ID. However, this is up to the plugin to define that mapping and its underlying storage organization.

```
/* Metadata index class */
typedef struct {
    void *(*create)(hid_t loc_id, hid_t xcpl_id, hid_t xapl_id,
        size_t *metadata_size, void **metadata);
    herr_t (*remove)(hid_t loc_id, size_t metadata_size, void *metadata);
    void *(*open)(hid_t loc_id, hid_t xapl_id, size_t metadata_size,
        void *metadata);
    herr_t (*close)(void *idx_handle);
    herr_t (*insert_entry)(void *idx_handle, hid_t obj_id, H5Q_type_t key_type,
       const void *key, size_t key_len, hid_t xxpl_id);
    herr_t (*remove_entry)(void *idx_handle, hid_t obj_id, H5Q_type_t key_type,
       const void *key, size_t key_len, hid_t xxpl_id);
    herr_t (*query)(void *idx_handle, hid_t query_id, hid_t xxpl_id,
        size_t *obj_count, hid_t *obj_ids[]);
    herr_t (*get_size)(void *idx_handle, hsize_t *idx_size);
} H5X_metadata_class_t;
```

The `create`, `remove`, open, `close` callbacks are similar to the data index plugins, albeit they operate on a file location and not on a single object. The `insert_entry`, `remove_entry` callbacks respectively insert/remove a referenced object from the index. Information can be directly extracted from the object ID and key passed, depending on the metadata type that the plugin will choose to index. When the `query` callback is invoked, a list of object IDs that match the query is returned. The `get_size` [8] callback returns the size of the index.

1.4.5. Current, Future, and Original Plugins

When originally written, the Query and Indexing supported a broader range of plugins than is currently available for use at present:

```
/* Plugin IDs */
#define H5X_PLUGIN_ERROR        (-1)    /* no plugin              */
#define H5X_PLUGIN_NONE         0       /* reserved indefinitely  */
#define H5X_PLUGIN_DUMMY        1       /* dummy                  */
#define H5X_PLUGIN_FASTBIT      2       /* fastbit                */
#define H5X_PLUGIN_ALACRITY     3       /* alacrity               */

#define H5X_PLUGIN_META_DUMMY   4       /* metadata dummy         */
#define H5X_PLUGIN_META_DB      5       /* metadata db            */
#define H5X_PLUGIN_META_MDHIM   6       /* metadata mdhim         */
```

The original *raw data indexing* plugins that were defined and tested included Fastbit and ALACRITY. The two implementations provided comparable performance and were open sourced software. Of these two, ALACRITY eventually transitioned their software efforts to providing a closer integration with ADIOS. Fastbit is the ONLY currently supported raw data indexing plugin. Internally, HDF5 continues to provide the internal callback functions for ALACRITY which were developed in conjunction with those of Fastbit.

Metadata indexing plugins for MDHIM and Berkeley DB were developed and their HDF5 callback structures remain in place. The only active metadata indexing plugin which has undergone testing for HDF5 release and support is for the Berkeley DB library.

[8] Additional info callbacks may also be added depending on the needs.

In the future, more plugins can be added, with or without external dependency, and to satisfy that need, dynamic plugin loading and registration must be supported, allowing external libraries to plug into the current interface.

1.5. Programming: Query and Indexing Function Summaries

Functions which can be used programmatically to provide Query and Indexing of HDF5 files and embedded datasets are listed below.

Table 2: Function Listing - Query

C Function – query API	Purpose
`hid_t H5Qcreate(H5Q_type_t query_type, H5Q_match_op_t match_op, ...)`	Create a new query object of `query_type` type, with `match_op` determining the query's match condition and additional parameters determined by the type of the query. Success returns a new *query_id.*
herr_t H5Qclose(hid_t query_id)	Terminate access to a query object, given by `query_id`.
hid_t H5Qcombine(hid_t query1_id, H5Q_combine_op_t combine_op, hid_t query2_id)	Create a new compound query object by combining two query objects (given by `query1` and `query2`), using the combination operator (`combine_op`). Valid combination operators are: {`H5Q_COMBINE_OR`, `H5Q_COMBINE_AND`}. Success returns a new *query_id*.
herr_t H5Qget_type(hid_t query_id, H5Q_type_t *query_type)	Query the *query object* given by `query_id` for its type information. Information is returned through the `query_type` parameter.
herr_t H5Qget_match_op(hid_t query_id, H5Q_match_op_t *match_op)	Query the *query object* given by `query_id` for its op information. It is an error to perform this call on a compound query object (one created with `H5Qcombine`). Match information is returned through the `match_op` parameter.
herr_t H5Qget_components(hid_t query_id, hid_t *sub_query1_id, hid_t *sub_query2_id)	Query the *compound query object* given by `query_id` for its component queries. It is an error to perform this call on a singleton query object. Component queries are returned through the `sub_query1_id` and `sub_query2_id` parameters.
herr_t H5Qget_combine_op(hid_t query_id, H5Q_combine_op_t *op_type)	Query the *query object* given by `query_id` for its operator type. The possible operator types are `H5Q_SINGLETON`, `H5Q_COMBINE_AND`, `H5Q_COMBINE_OR` `H5Q_SINGLETON` is returned only for query objects created with H5Qcreate.

herr_t H5Qencode(hid_t query_id, void *buf, size_t *nalloc)	Given a *query object* given by `query_id`, serialize the query into `buf`. The encoded size is returned through the `nalloc` parameter.
hid_t H5Qdecode(const void *buf)	Deserialize the given `buf` and return a new query handle. The handle should be closed using `H5Qclose`.

The following functions are principally for new plugin developers. There should be no reason that the general HDF5 user community is required to know or utilize these programming APIs.

Table 3: Function Listing - Index plugins

C Function – Indexing API	**Purpose**
`herr_t H5Xregister(const H5X_class_t *index_class)`	Register an index class.
`herr_t H5Xunregister(unsigned plugin_id)`	Unregister an index class.
`herr_t H5Xcreate(hid_t loc_id, unsigned plugin_id, hid_t xcpl_id)`	Create a new index in a container.
`herr_t H5Xremove(hid_t loc_id, unsigned plugin_id)`	Remove and index from objects in a container.
`herr_t H5Xget_count(hid_t loc_id, hsize_t *idx_count)`	Determine the number of index objects on an object identified by the `loc_id` paramenter. The index count returned via the `idx_count` parameter.
`hsize_t H5Xget_size(hid_t loc_id)`	Returns the amount of storage allocated for an index identified by the `loc_id` parameter.

The above set of indexing functions make calls to the plugin-specific class methods as described by the `H5X_data_class_t` structure or the `H5X_metadata_class_t` structure, depending on the type of HDF5 object being accessed.

2. Query Programming Model.

The programming model used for querying has three phases:

- Building the data and metadata indices;
- Applying a set of queries to create views;
- Extracting the relevant data of interest that are referenced in the constructed views.

The following sections provide the specifics of these phases.

2.1. Building data and metadata index information.

The Raw Data indices get initialized in response to HDF5 dataset creation, e.g. `H5Dcreate2`, etc., which in turn calls the `H5Xcreate` method in the H5X set of APIs. This function, in turn, invokes plugin specific functions to initialize possibly dynamically load or otherwise initialize the requested plugin. The data internal index will be generated in conjunction with the following data write operation, e.g. `H5Dwrite`. An initial invocation of the `idx_class->data_class.pre_update` method flags the intent to update the dataset (and possibly invalidate an existing index), followed by calling `idx_class->data_class.post_update` and providing the actual data buffer to be indexed. Calling the post_update rebuilds the index and allows the internal plugin index metadata to be refreshed.

2.2. Examples of applying a query and creating a view.

In this example, we show how one can create a query, apply that query to a file, and retrieve data from the resulting view.

Figure 5: Example 1

```
#define FILENAME "test.h5"
#define DATASETNAME "Pressure"

int
main(int argc, char *argv[])
{
    hid_t t file, query, subquery1, subquery2, view;
    float subquery2_value = 3.14;
    unsigned apply_result = 0;

    /* Create file */
    file = H5Fcreate(FILENAME, H5F_ACC_TRUNC, H5P_DEFAULT, H5P_DEFAULT);

    /* Create groups / datasets / attributes / etc */
    ...

    /* Close the file */
    H5Fclose(file);

    /* Create a simple query */
    subquery1 = H5Qcreate(H5Q_TYPE_LINK_NAME, H5Q_MATCH_EQUAL, DATASETNAME);
    subquery2 = H5Qcreate(H5Q_TYPE_DATA_ELEM, H5Q_MATCH_LESS_THAN,
        H5T_NATIVE_FLOAT, &subquery2_value);
    query = H5Qcombine(subquery1, H5Q_COMBINE_AND, subquery2);

    /* Open file */
    file = H5Fopen(FILENAME, H5F_ACC_RDONLY, H5P_DEFAULT);

    /* Apply query to file */
    view = H5Qapply(file, query, &apply_result, H5P_DEFAULT)

 /* Result should contain region reference */
    if (result & H5Q_REF_REG) {
        hid_t refs, ref_type, ref_space, space;
        size_t n_refs, ref_size;
        void *ref_buf;

        /* Read region reference from view */
        refs = H5Dopen(view, H5Q_VIEW_REF_REG_NAME, H5P_DEFAULT);
        ref_type = H5Dget_type(refs);
        ref_space = H5Dget_space(refs);
        n_refs = (size_t) H5Sget_select_npoints(ref_space);
        ref_size = H5Tget_size(ref_type);
        ref_buf = malloc(n_refs * ref_size)
        H5Dread(refs, ref_type, H5S_ALL, ref_space, H5P_DEFAULT, ref_buf);
        H5Dclose(refs);
        H5Sclose(ref_space);

        /* Get selection from region reference */
        space = H5Rget_region(file, (href_t *) ref_buf);

        /* Use selection result [...] */

        /* Close selection */
        H5Sclose(space);

        /* Free reference buffer */
        free(ref_buf);
    }

    /* Close view */
    H5Gclose(view);

    /* Close queries */
    H5Qclose(query);
    H5Qclose(subquery1);
```

In a second example, we show how one can make use of the query and indexing APIs to efficiently retrieve a dataspace selection within a dataset.

Figure 6: Example 2

```
#define NTUPLES 256
#define FILENAME "test.h5"
#define DATASETNAME "Pressure"

int
```

```
main(int argc, char *argv[])
{
    float data[NTUPLES], *result;
    hsize_t dims[1] = {NTUPLES};
    hid_t t file, dataspace, dataset, dcpl, query, result_space;
    size_t result_npoints;
    int i;

    /* Initialize data */
    for(i = 0; i < NTUPLES; i++) data[i] = (float) i;

    /* Create file */
    file = H5Fcreate(FILENAME, H5F_ACC_TRUNC, H5P_DEFAULT, H5P_DEFAULT);

    /* Create the data space for the dataset */
    dataspace = H5Screate_simple(1, dims, NULL);

    /* Create and set property list to use FastBit index */
    dcpl = H5Pcreate(H5P_DATASET_CREATE);
    H5Pset_index_plugin(dcpl, H5X_PLUGIN_FASTBIT);

    /* Create dataset */
    dataset = H5Dcreate(file, DATASETNAME, H5T_NATIVE_FLOAT, dataspace,
        H5P_DEFAULT, dcpl, H5P_DEFAULT);

    /* Write dataset */
    H5Dwrite(dataset, H5T_NATIVE_FLOAT, H5S_ALL, H5S_ALL, H5P_DEFAULT, data);

    /* Close the dataset */
    H5Dclose(dataset);

    /* Close dataspace */
    H5Sclose(dataspace);

    /* Close the property */
    H5Pclose(dcpl);

    /* Close the file */
    H5Fclose(file);

    /* Create a simple query */
    query = H5Qcreate(H5Q_TYPE_DATA_ELEM, H5Q_MATCH_EQUAL, H5T_NATIVE_FLOAT,
        &query_value);
    /* Open file */
    file = H5Fopen(FILENAME, H5F_ACC_RDONLY, H5P_DEFAULT);

    /* Open dataset */
    dataset = H5Dopen(file, DATASETNAME, H5P_DEFAULT);

    /* NB. Alternatively the index can be generated on the existing dataset.
     * This, however, requires the file to be open in H5F_ACC_RDWR so that
     * the index can be stored within the file and attached to the dataset.
     * H5Xcreate(dataset, H5X_PLUGIN_FASTBIT, H5P_DEFAULT); */

    /* Use query to select elements in the dataset */
    result_space = H5Dquery(dataset, H5S_ALL, query, H5P_DEFAULT);
    /* Allocate space to read data */
    result_npoints = (size_t) H5Sget_select_npoints(result_space);
    result = malloc(result_npoints * sizeof(float));

    /* Read data using result_space_id */
    H5Dread(dataset, H5T_NATIVE_FLOAT, H5S_ALL, result_space,
        H5P_DEFAULT, result);

    /* Use result [...] */

    /* Free result */
    free(result);

    /* Close the dataset */
    H5Dclose(dataset);

    /* Close dataspace */
    H5Sclose(result_space);

    /* Close query */
    H5Qclose(query);

    /* Close the file */
    H5Fclose(file);
}
```

3. Best Practices and Special Issues

3.1. Parallelism and Index generation

Applications which are built to utilize the parallel HDF5 library often subdivide the reading or writing of datasets of interest by subdividing the virtual global space into a local memory space. Hyperslab selections for example, utilize a starting offset, stride information, a block count, and the description of a single block in the mapping of a region in global space into the local memory space.
If one considers a global array of 10 million elements and 10 physical processors, then an evenly distributed array would contain 1M elements per processor. Under those initial conditions, the `H5Dwrite` operation will generate a new global dataset on each MPI rank in a coordinated fashion.

An important design choice to support parallel indexing within the `H5X_data_class_t` *plugin class* is to give as much freedom as possible to the indexing plugin developer so that in the case when the indexing library supports parallel indexing, it is still possible to take advantage of it. Three options are available to support parallel indexing from the previous interface:

- Let the index creation be collective. This, however, implies having synchronization points, which is the main constraint.
- Let the index creation be independent. However, creating datasets to store the index data must be done collectively[9].
- Make the index creation in two phases. One that consists of building the index in parallel, independently, and gathering the information (index size, etc.) at the end.

The existing Fastbit plugin implements the 3rd approach.

Upon completion of the dataset creation, each MPI process would proceed to create one or more queries and apply those against their local data. Query application and view creation are accomplished independently.

It is important to note that while the initial conditions will often be well balanced as a result of the dataset creation, the post-application of user queries and view generation can result in an unbalanced distribution of views on each MPI rank. This condition can lead to unexpected performance issues such as some MPI ranks having no data in their local view, while other ranks contain only a few, while still other ranks contain a majority of the results. One potential strategy to address these issues would be to collectively persist the temporary views that reside on the processor as a new global dataset.

3.2. Limitations

3.2.1. No Incremental index Updates

There are some existing limitations in the use of indexes in the current implementation: FastBit does not support incremental updates, an index is a shared resource for a dataset. Taken together, these conspire to

[9] An option could be to use the metadata server VOL plugin but this option is not easily doable yet.

put limits on application updates to datasets with indexes. Additionally, because incremental updates are not supported, each modification to an existing dataset forces the index to be entirely rebuilt. The limitation in FastBit can only be addressed in the base packages' implementation so that incremental updates to their index information can be made.

3.2.2. Support for HDF5 Compound Types (TBD)

In a simple scenario, the HDF5 datatype used for creating the dataset can be defined as a native and simple type. Therefore, building an index on this dataset implies building that index from the entire dataset. However, in more complex scenarios, the dataset may have been created by using a compound datatype, hence defining multiple fields composed of native and simple datatypes within that same dataset. As a consequence, creating an index from that dataset requires the user to select a particular field to be indexed, which may lead to having multiple indexes per dataset depending on the number of fields that it contains. This can be done by passing the `datatype_id` of the field to be indexed to the `xcpl_id` in the index creation property list for the H5X`create` call which then passes
it down to the plugin via the `create` callback. As multiple fields can be defined, the field `datatype_id` must be stored along with the existing metadata, within the `idx_info` message so that the index associated to the field can be retrieved at the time of the query.

When doing a query, the compound type is passed to the plugin via the H5Q`create` call. The corresponding index is then used, and the query is passed to the plugin `query` callback.

When removing an index, one may choose to remove the index that corresponds to a particular field. This can be achieved by calling plugin function H5X`get_info` to compare the datatype returned within the info structure, and pass the index number that needs to be removed.

3.2.3. Support for HDF5 Chunking (TBD)

To support indexing of HDF5 chunks, we make each chunk a local *sub-dataset* of the original dataset. In that sense, handling every chunk can be seen as handling a dataset from the indexing plugin point of view. If the dataset is chunked, at the time of the index creation, we create a B-tree[10] (physically
stored on disk) that maps the coordinates of the chunks to the index plugin metadata. When the `create` callback is called by representing the chunk as a local dataset, i.e., making the dataset layout point to the address of the chunk), metadata information is returned and stored. In the case of contiguous datasets, the index metadata, as well as the index plugin ID, is stored within the dataset header of the index info message.

In the case of chunked datasets, multiple metadata that corresponds to each index created from each chunk may be accessed. Therefore, only the address of the B-tree that contains the metadata pieces for each chunk is stored in that header message and the index metadata itself is stored in the B-tree.

When the dataset is opened and the index reopened, we can locate the index information in the B-tree that corresponds to each chunk and call the `open` callback using the associated metadata. Similarly, when a query is issued and needs to be answered, the chunks that correspond to the selection passed to the H5D`query` call are selected and their index is used to answer that query. The selection returned is then

[10] The B-tree could also be replaced by a map object.

added to a global selection, which is then, in turn, returned to the user. Finally, when `H5Xremove` is called, the `delete` plugin callback is invoked for each chunk, by using the index metadata information stored in the B-tree.