RFC: Data Analysis Extensions

Jerome Soumagne
Quincey Koziol

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 in this RFC extensions to the library to efficiently query, select and index data.

1 Introduction

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 their 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, when applied to the data, can then generate a selection, which contains the coordinates that satisfy the query—this may imply accessing the data and selecting it, depending on the query condition satisfaction. To accelerate and facilitate this process (i.e., so that the data no longer needs to be directly accessed), one can use indexing techniques, which consist of generating an index and using that index to answer the query selection criteria and find the matching elements.

We define in this RFC the components that can 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 on HDF5 containers can be defined via:

1. New *query* objects[1] and API routines, enabling the construction of query requests for execution on HDF5 containers;
2. New *view* objects[1] and API routines, which [c1]are the result of a query applied to an HDF5 container and consist of a set of references into the container that fulfills the query criteria;
3. New *index* objects and API routines, which allows the creation of indices on the contents of HDF5 containers, to improve query performance.

[c1] *JS*: ~~apply a query to an HDF5 container and return~~

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

2 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:

```
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);
```

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 data elements, link names, attribute names, etc., a match operator, such as $=$, $\neq$, $\leq$, $\geq$, and a value for the match operator. Created query objects can be serialized and deserialized using `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 *AND*($\wedge$) and *OR*($\vee$). 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 are 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 $\wedge$ 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 *developer-friendly*) query syntax like *Pressure* = 17. However, we regard this as out-of-scope for now.

Table 1 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].

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

[3]Query results of *None* type may be implemented with another result type in the future, once experience with the query framework is acquired and a meaningful grammar for those results are defined.

Table 1 Query combinations and associated result type.

Query	Result Type
`H5Q_TYPE_DATA_ELEM`	*Dataset Element*
`H5Q_TYPE_ATTR_VALUE`	*Attribute*
`H5Q_TYPE_ATTR_NAME`	*Object*
`H5Q_TYPE_LINK_NAME`	*Object*
Dataset Element ∧ *Dataset Element*	*Dataset Element*
Dataset Element ∧ *Attribute*	*None*
Dataset Element ∧ *Object*	*Dataset Element*
Attribute ∧ *Attribute*	*Attribute*
Attribute ∧ *Object*	*Attribute*
Object ∧ *Object*	*Object*
Dataset Element ∨ *Dataset Element*	*Dataset Element*
Dataset Element ∨ *Attribute*	*Combination*
Dataset Element ∨ *Object*	*Combination*
Dataset Element ∨ *Combination*	*Combination*
Attribute ∨ *Attribute*	*Attribute*
Attribute ∨ *Object*	*Combination*
Attribute ∨ *Combination*	*Combination*
Object ∨ *Object*	*Object*
Object ∨ *Combination*	*Combination*
Combination ∨ *Combination*	*Combination*
Combination ∧ *Dataset Element*	*None*
Combination ∧ *Attribute*	*None*
Combination ∧ *Object*	*None*
Combination ∧ *Combination*	*None*

3 View Objects

Applying a query to an HDF5 container creates an HDF5 view. HDF5 view objects are run-time, in-memory objects (i.e., not stored in a container [c1]but which can be persisted), that consist of read-only references [c2]to attributes, objects and data regions into the contents of the HDF5 container that the query was applied to. [c3]The view object can be therefore seen as a virtual container. It is defined as an anonymous HDF5 group, which consists of three predefined datasets that contain attribute, object and region references, and stored in-memory, i.e., using the existing core driver. A new view is created by the following routine, which applies a query to an HDF5 container, group hierarchy, or individual object and returns the object ID of the newly created group:

[c1] *JS: Text added.*

[c2] *JS: Text added.*

[c3] *JS: Text added.*

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

[c1]As mentioned, the created view may also need to be persisted and this can be done by calling

[c1] *JS: Text added.*

H5Ocopy() to copy the group (stored in a virtual container) to a persistent container. For coherency, a time stamp may 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 also be useful in that case to define different states to the view (*dead* or *live*) so that the user knows whether the view is current or not.

[c1] Note that currently, attribute references are not available, this feature will be added in order to support views that contain attribute references[c1].

[c1] *JS*: ~~View objects are created with `H5Vcreate`, which applies a query to an HDF5 container, group hierarchy, or individual object and produces the view object as a result. The attributes, objects, and/or data elements referenced within a view can be retrieved by further API calls.~~

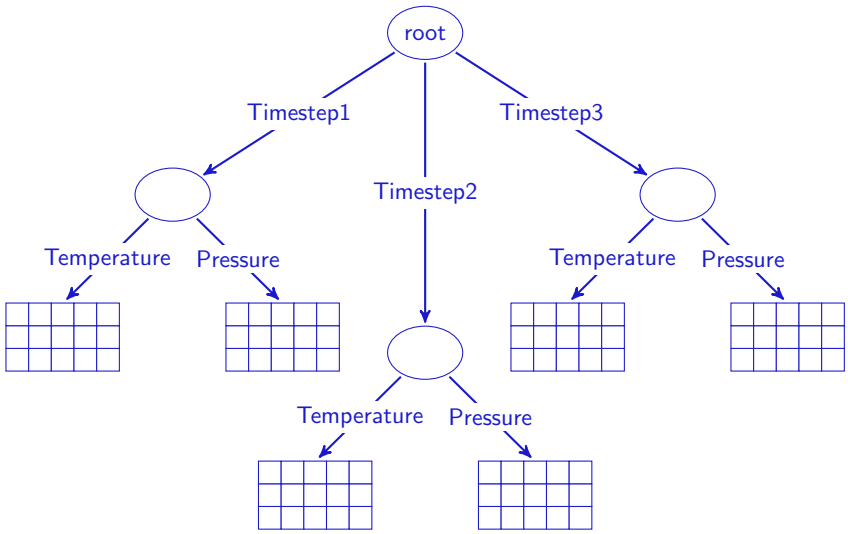

Figure 1 HDF5 container example.

For example, starting with the HDF5 container described in Figure 1, applying the *link_name* = *Pressure* query (described above) would result in the view shown in Figure 2, highlighted in blue.

Alternatively, applying the *data_element* = 17 query (described above) would result in the view shown in Figure 3, highlighted in blue.

Finally, applying the combined (*link_name* = *Pressure*) ∧ (*data_element* = 17) query (described above) would result in the view shown in Figure 4, highlighted in blue.

Views can be thought of as containing a set of HDF5 references (object, dataset region or attribute references) to components of the underlying container, retaining the context of the original container. For example, the view containing the results of the (*link_name* = *Pressure*) ∧ (*data_element* = 17) query 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.

[c1] See RFC 201X-XX-XX.vX.

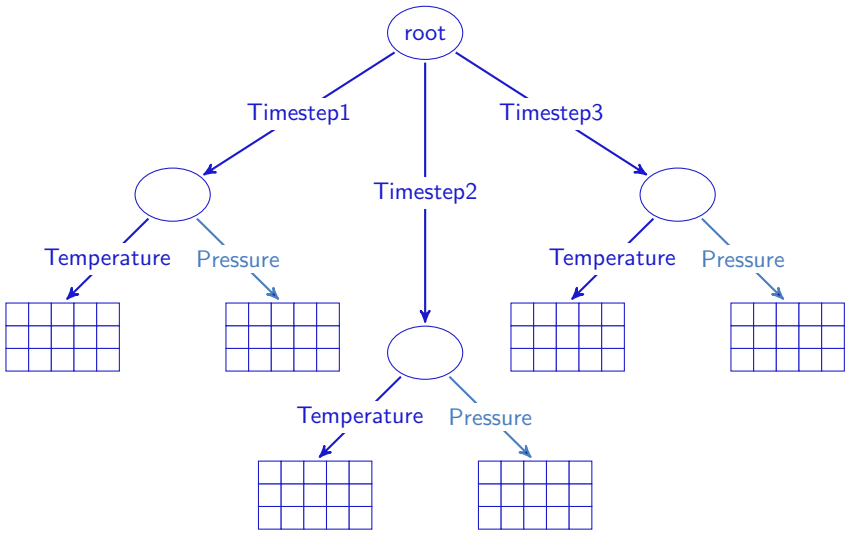

Figure 2 HDF5 container example with query *link_name* = *Pressure* applied.

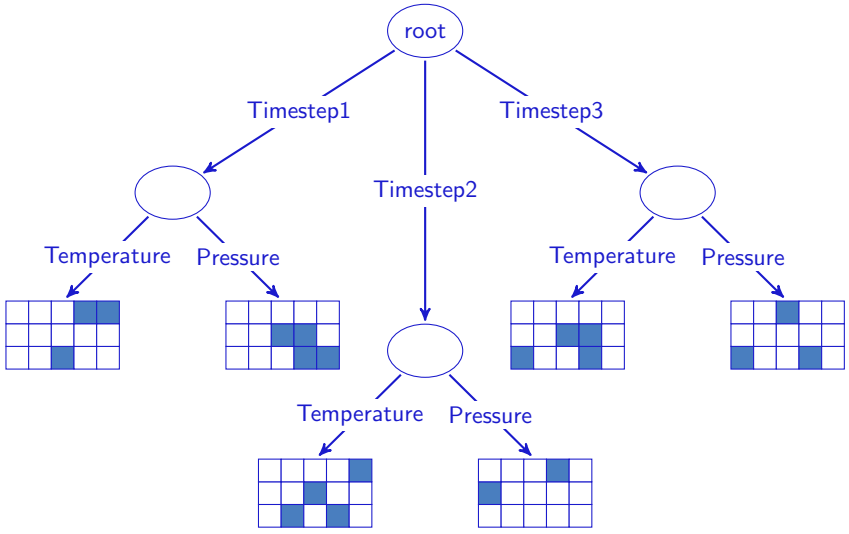

Figure 3 HDF5 container example with query *data_element* = 17 applied.

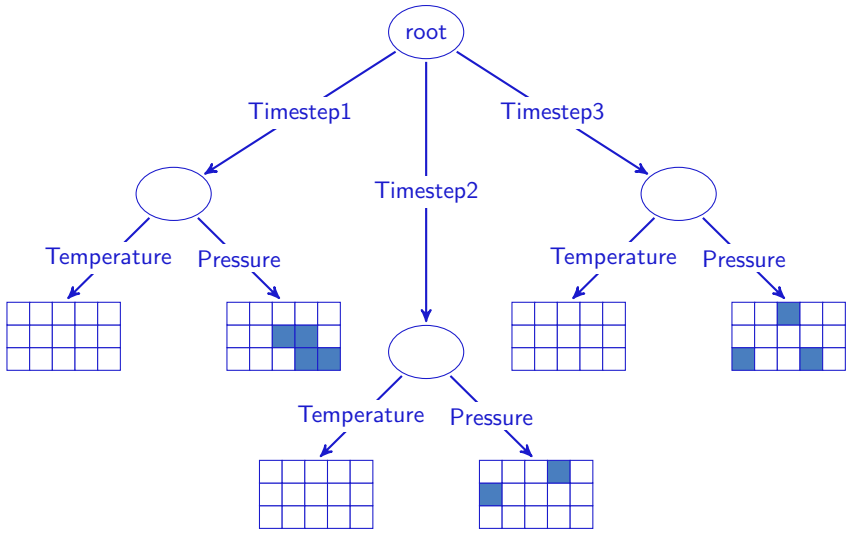

Figure 4 HDF5 container example with query (*link_name* = *Pressure*) ∧ (*data_element* = 17) applied.

4 Index Objects

Index objects are designed to accelerate creation of view objects from frequently occurring query operations. For example, if the (*link_name* = *Pressure*) ∧ (*data_element* = 17) query (previously described) is going to be frequently executed on the container, indices could be created in that container, which would speed up the creation of views when querying for link names and for data element values. Indices created for accelerating the *link_name* = *Pressure* or *data_element* = 17 queries would also improve view creation for the more complex (*link_name* = *Pressure*) ∧ (*data_element* = 17) query.

Although creating indices for metadata components of queries, such as link or attribute names, is possible, we focus on index creation for dataset elements, as they represent the largest volume of data in typical HPC application usage of HDF5. Queries with metadata components execute properly, but are not able to be accelerated with an index currently.

Metadata indexing must be added.

The indexing API can work in conjunction with the view API. When an `H5Vcreate` call is made for a group or dataset, 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 indices, and the most efficient method will vary depending on the type of the data that is to be indexed, its layout, etc. A new interface for the HDF5 library that uses a plugin mechanism is therefore defined.

4.1 Indexing Interface and Plugins

This interface is defined for adding third-party indexing plugins, such as FastBit [1], ALACRITY [2], etc. The interface provides indexing plugins with efficient access to the contents of the container for both the creation and the maintenance of indices. 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:

```
herr_t  H5Xregister(const H5X_class_t *idx_class);
herr_t  H5Xunregister(unsigned plugin_id);
herr_t  H5Xcreate(hid_t scope_id, unsigned plugin_id, hid_t xcpl_id);
herr_t  H5Xremove(hid_t scope_id, unsigned n /* Index n to be removed */);
herr_t  H5Xget_count(hid_t scope_id, hsize_t *idx_count);
herr_t  H5Xget_info(hid_t scope_id, unsigned n, H5X_info_t *info);
hsize_t H5Xget_size(hid_t scope_id);

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, described above */
    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_class_t;
```

Index objects are stored in the HDF5 container that they apply to, but are not visible in the container's group hierarchy[c0]. Instead, index objects are part of the metadata for the file itself. New index objects are created by passing an HDF5 container 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

[c0] 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.

and index metadata) is stored at index creation time[c0], 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. Similarly, calling `H5Dclose` will call the plugin index close callback and close the objects used to store the index data.

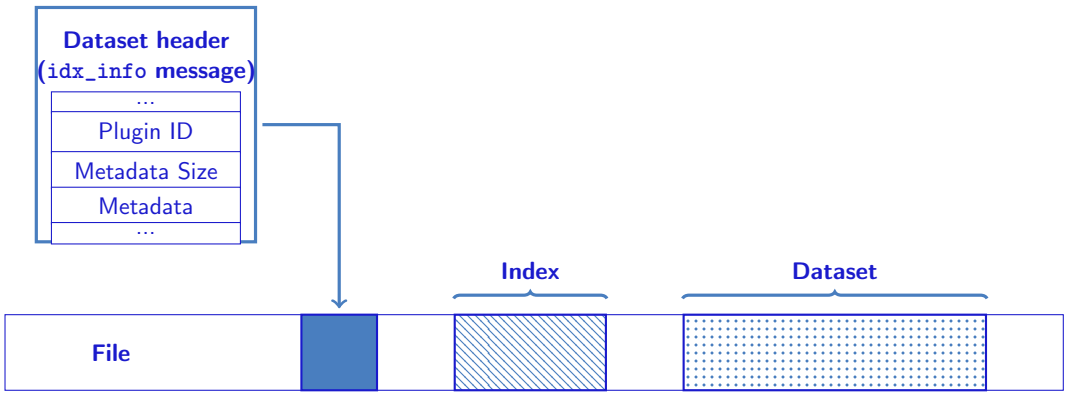

Figure 5 Index information (plugin ID and metadata) is stored along the object header.

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. 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 `H5Vcreate` 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 use the new `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.

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

Because the amount of space taken by the index cannot be directly retrieved by the user (since the datasets storing the indices 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 query that information using the corresponding `H5Xget_size` routine.

4.1.1 Current and future plugins

Implementations for FastBit and ALACRITY index packages are already present, as well as a brute force indexing plugin. Early performance results are presented in Figure 6.

[c0] Adding index information introduces a file format change.

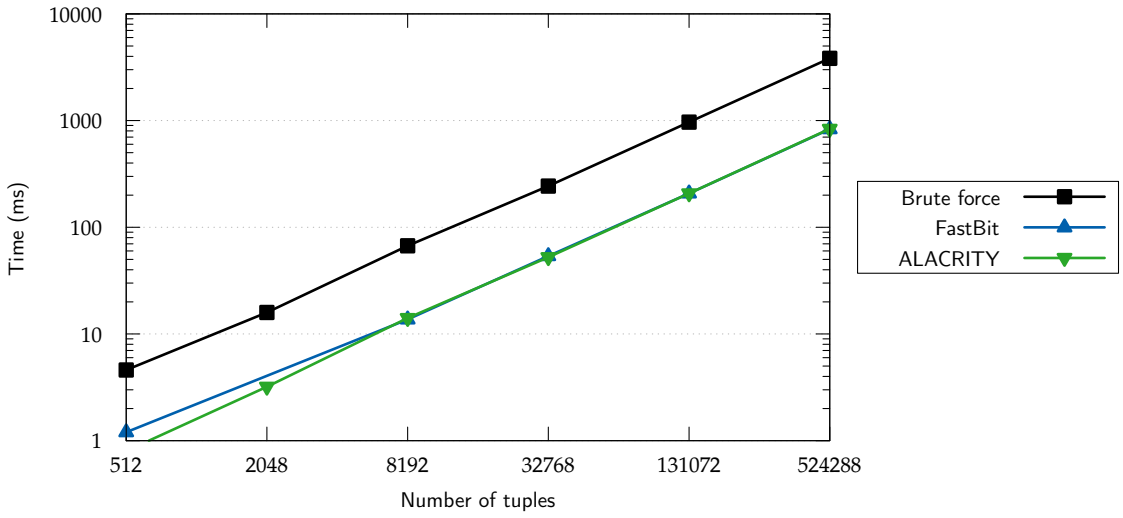

Figure 6 Indexing performance.

In the future more plugins will be added, with or without external dependency (e.g., PyTables indexing, bitmap indexing). To satisfy that need, dynamic plugin loading and registration will be supported, allowing external libraries to plug to the current interface.

4.2 Limitations

There are some existing limitations in the use of indices in the current implementation: FastBit and ALACRITY do not support incremental updates, an index is a shared resource for a dataset. Taken together, these conspire to put limits on application updates to datasets with indices. Additionally, because FastBit and ALACRITY do not allow incremental updates to an index, each modification to an existing index forces the index to be entirely rebuilt. The limitation in FastBit and ALACRITY will need to be addressed in the base packages' implementation, so that incremental updates to their index information can be made.

Questions

Some questions are still open regarding the handling of indices:

1. How to handle index updates when the specified index plugin is not available? (In traditional databases, stored procedures are saved with the data and therefore available at any time, but that is not the case here) We could mark the index as out of date and rebuild the index when the plugin is available again.
2. How to handle index queries when the specified index plugin is not available? We could fallback to another plugin and do a brute force query on the data.

4.3 Support for HDF5 Compound Types

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. Consequently, creating an index from that dataset requires the user to select a particular field to be indexed, which may lead to having multiple indices 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`, the index creation property list, of the `H5Xcreate` call, which passes it down to the plugin `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 (see Figure 5) 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 `H5Qcreate` call. The corresponding index is then used and the query is passed to the `query` callback of the plugin.

Consequently, when removing an index, one may choose to remove the index that corresponds to a particular field. This can be achieved by calling `H5Xget_info`, compare the datatype returned within the info structure, and pass the index number that needs to be removed.

4.4 Support for HDF5 Chunking

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[c0] (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 (see Figure 5). In the case of chunked datasets, multiple metadata that correspond to each index created from each chunck 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 lookup 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 `H5Dquery` call are selected and their index is used to answer that query. The selection returned is then 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.

[c0] The B-tree could also be replaced by a map object.

4.5 Support for Parallel Indexing

An important design choice to support parallel indexing 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:

1. Let the index creation be collective. This however implies having synchronization points, which is the main constraint.
2. Let the index creation be independent. However, creating datasets to store the index data must be done collectively[c0].
3. 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 other that consists of letting the dataset creation be done by a master process, which can then let the other processes write the index data independently. There could however be a memory constraint in this case if the index data has to be kept in memory between these two phases (though building the index twice is not a good solution either).

Changes in the plugin interface include passing the parallel context to the callback (MPI communicator), which can be done by using the index access property list. In the case of chunked datasets, one may also want to operate on several chunks at the same time, in parallel. This can be done by passing a list of IDs that corresponds to each chunk to the callbacks. However, this also makes the plugin interface more heavy.

5 HDF5 and tools

Existing HDF5 tools must be compatible and take into account the existence of indices in the file if there are any. For reference, the following behavior for the tools is given:

- h5copy: copy
- h5dump: report index information
- h5ls: report index information
- h5diff: ignore index information?
- h5repack: copy index information / or generate index
- h5edit: ignore index information?
- h5toh4: ignore index information?
- h5import: ignore index information

Additional tools for indexing data and answering queries will also be added in the future.

[c0] An option could be to use the metadata server VOL plugin but this option is not easily doable yet.

6 Usage Example

In the following example, we show how one can make use of the query and indexing interface to retrieve a dataspace selection within a dataset. For simplicity's sake, we first create a dataset within the file, then open it to create and attach a new index, in order to finally query data from it. Note that for convenience, calls to directly read data that corresponds to the result of the index query may be moved to the high-level API in the future.

```
#define NTUPLES 256

int
main(int argc, char *argv[])
{
    float data[NTUPLES];
    hsize_t dims[1] = {NTUPLES};
    hid_t t file_id, dataspace_id, dataset_id;
    hid_t query_id, result_space_id;
    size_t result_npoints;
    float *result;
    int i;

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

    /* Create file. */
    file_id = H5Fcreate(file_name, H5F_ACC_TRUNC, H5P_DEFAULT, H5P_DEFAULT
        );

    /* Create the data space for the dataset. */
    dataspace_id = H5Screate_simple(rank, dims, NULL);

    /* Create dataset. */
    dataset_id = H5Dcreate(file_id, "Pressure", H5T_NATIVE_FLOAT,
        dataspace_id,
         H5P_DEFAULT, H5P_DEFAULT, H5P_DEFAULT);

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

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

    /* Close dataspace. */
    H5Sclose(dataspace_id);

    /* Open dataset. */
    dataset_id = H5Dopen(file_id, "Pressure", H5P_DEFAULT);

    /* Create index using FastBit. */
    H5Xcreate(file_id, H5X_PLUGIN_FASTBIT, dataset_id, H5P_DEFAULT);

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

    /* Create a simple query */
```

```
46      query_id = H5Qcreate(H5Q_TYPE_DATA_ELEM, H5Q_MATCH_EQUAL,
            H5T_NATIVE_FLOAT,
47          &query_value);
48
49      /* Open dataset. */
50      dataset_id = H5Dopen(file_id, "Pressure", H5P_DEFAULT);
51
52      /* Use query to select elements in the dataset. */
53      result_space_id = H5Dquery(dataset_id, query_id);
54
55      /* Allocate space to read data. */
56      result_npoints = (size_t) H5Sget_select_npoints(result_space_id);
57      result = malloc(result_npoints * sizeof(float));
58
59      /* Read data using result_space_id. */
60      H5Dread(dataset_id, H5T_NATIVE_FLOAT, H5S_ALL, result_space_id,
61          H5P_DEFAULT, result);
62
63      /* Use result. */
64
65      /* Free result. */
66      free(result);
67
68      /* Close the dataset. */
69      H5Dclose(dataset_id);
70
71      /* Close dataspace. */
72      H5Sclose(result_space_id);
73
74      /* Close query. */
75      H5Qclose(query_id);
76
77      /* Close the file. */
78      H5Fclose(file_id);
79  }
```

7 Conclusion

Document in progress.

Revision History

Jul. 17, 2014:	Version 1 circulated for comment within The HDF Group.
Aug. 20, 2014:	Version 2 circulated for comment within The HDF Group.
Nov. 14, 2014:	Version 3 circulated for comment within The HDF Group.
Feb. 25, 2015:	Version 4 edits and typo fixing.
Jul. 24, 2015:	Version 5 with edits to view object.

References

[1] K. Wu, "FastBit: an efficient indexing technology for accelerating data-intensive science," *Journal of Physics: Conference Series*, vol. 16, no. 1, p. 556, 2005.

[2] J. Jenkins, I. Arkatkar, S. Lakshminarasimhan, I. Boyuka, DavidA., E. Schendel, N. Shah, S. Ethier, C.-S. Chang, J. Chen, H. Kolla, S. Klasky, R. Ross, and N. Samatova, "ALACRITY: Analytics-Driven Lossless Data Compression for Rapid In-Situ Indexing, Storing, and Querying," in *Transactions on Large-Scale Data- and Knowledge-Centered Systems X* (A. Hameurlain, J. Küng, R. Wagner, S. Liddle, K.-D. Schewe, and X. Zhou, eds.), vol. 8220 of *Lecture Notes in Computer Science*, pp. 95–114, Springer Berlin Heidelberg, 2013.