RFC: Data Analysis Extensions

Jerome Soumagne
Quincey Koziol

Accessing 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 the process of querying data, we introduce in this RFC data analysis extensions to query, select and index data.

1 Introduction

Data Analysis Extensions to the HDF5 API and data model 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 to an HDF5 container. Support for data analysis operations on HDF5 containers can be defined via:

- New *query* object[1] and API routines, enabling the construction of query requests for execution on HDF5 containers;
- New *view* object[1] and API routines, which apply a query to an HDF5 container and return a set of references into the container that fulfills the query criteria;
- New *index* object and API routines, which allows the creation of indices on the contents of HDF5 containers, to improve query performance.

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:

```
1  hid_t  H5Qcreate(H5Q_type_t query_type, H5Q_match_op_t match_op, ...);
2  herr_t H5Qclose(hid_t query_id);
3
4  hid_t  H5Qcombine(hid_t query1_id, H5Q_combine_op_t combine_op, hid_t query2_id);
5
6  herr_t H5Qget_type(hid_t query_id, H5Q_type_t *query_type);
7  herr_t H5Qget_match_op(hid_t query_id, H5Q_match_op_t *match_op);
8  herr_t H5Qget_components(hid_t query_id, hid_t *subquery1_id, hid_t *subquery2_id);
9  herr_t H5Qget_combine_op(hid_t query_id, H5Q_combine_op_t *op_type);
10
11 herr_t H5Qencode(hid_t query_id, void *buf, size_t *nalloc);
12 hid_t  H5Qdecode(const void *buf);
```

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

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].

3 View Objects

Applying a query to an HDF5 container creates an HDF5 view object. HDF5 view objects are runtime, in-memory objects (i.e., not stored in a container) that consist of read-only references into the contents of the HDF5 container that the query was applied to. The current API is presented below:

```
hid_t H5Vcreate(hid_t loc_id, hid_t query_id, hid_t vcpl_id);
herr_t H5Vclose(hid_t view_id);
herr_t H5Vget_location(hid_t view_id, hid_t *loc_id);
herr_t H5Vget_query(hid_t view_id, hid_t *query_id);
herr_t H5Vget_counts(hid_t view_id, hsize_t *attr_count, hsize_t *obj_count,
  hsize_t *elem_region_count);
herr_t H5Vget_attrs(hid_t view_id, hsize_t start, hsize_t count, hid_t attr_id[]);
herr_t H5Vget_objs(hid_t view_id, hsize_t start, hsize_t count, hid_t obj_id[]);
herr_t H5Vget_elem_regions(hid_t view_id, hsize_t start, hsize_t count,
  hid_t dataset_id[], hid_t dataspace_id[]);
```

[2] Useful if the query needs to 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 $\wedge$ *Dataset Element*	*Dataset Element*
Dataset Element $\wedge$ *Attribute*	*None*
Dataset Element $\wedge$ *Object*	*Dataset Element*
Attribute $\wedge$ *Attribute*	*Attribute*
Attribute $\wedge$ *Object*	*Attribute*
Object $\wedge$ *Object*	*Object*
Dataset Element $\vee$ *Dataset Element*	*Dataset Element*
Dataset Element $\vee$ *Attribute*	*Combination*
Dataset Element $\vee$ *Object*	*Combination*
Dataset Element $\vee$ *Combination*	*Combination*
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*

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. Note that currently, attribute references are not available, this feature will be added in order to support views that contain attribute references[4].

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, with the underlying container grayed out and the view highlighted in blue.

Alternatively, applying the $data\_element = 17$ query (described above) would result in the view shown in Figure 3, with the underlying container greyed out and the view highlighted in blue.

[4]See RFC XXXX for attribute references.

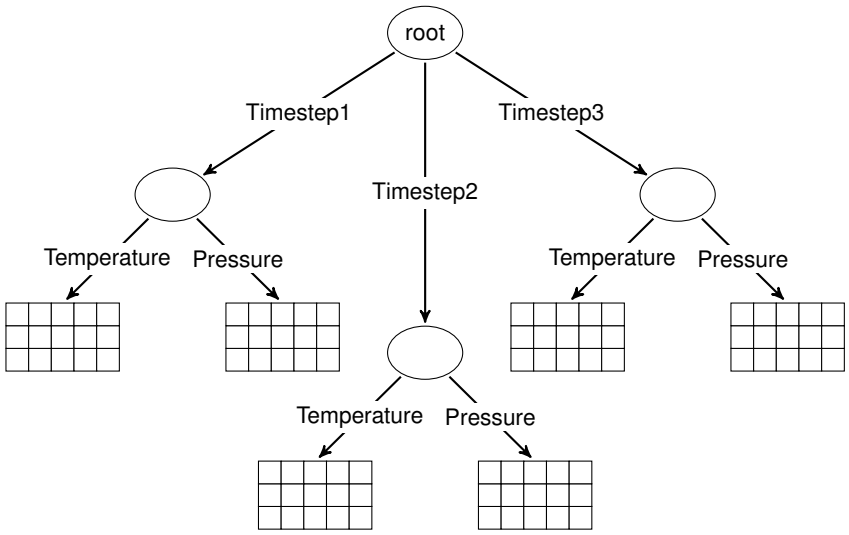

Figure 1 HDF5 container example.

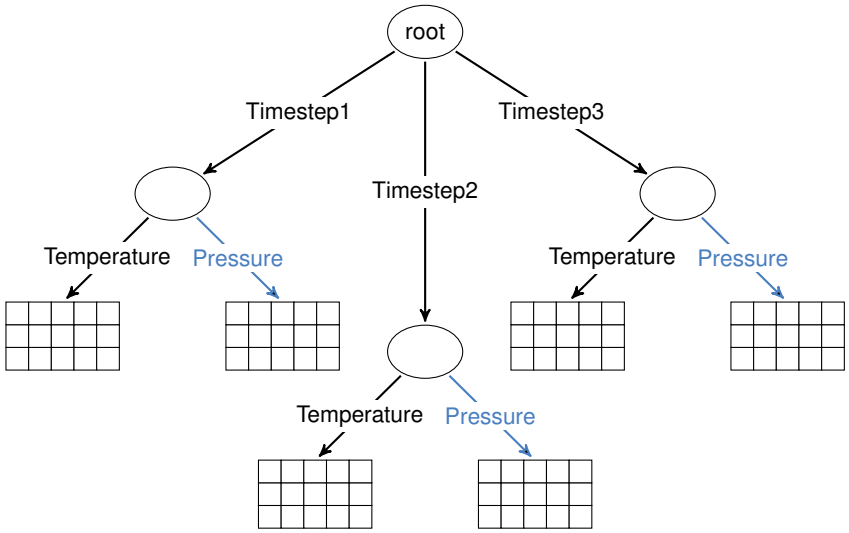

Figure 2 HDF5 container example with query $link\_name = Pressure$ applied.

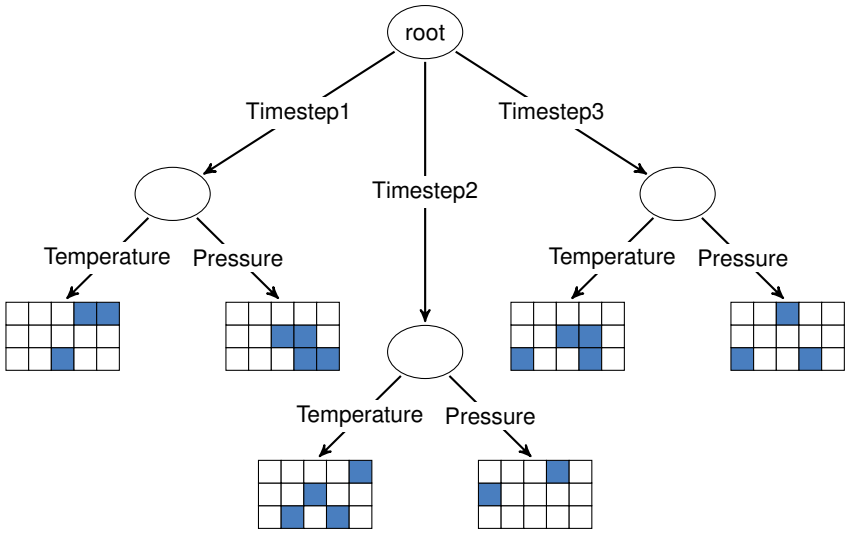

Figure 3 HDF5 container example with query $data\_element = 17$ applied.

Finally, applying the combined $(link\_name = Pressure) \wedge (data\_element = 17)$ query (described above) would result in the view shown in Figure 4, with the underlying container greyed out and the view highlighted in blue.

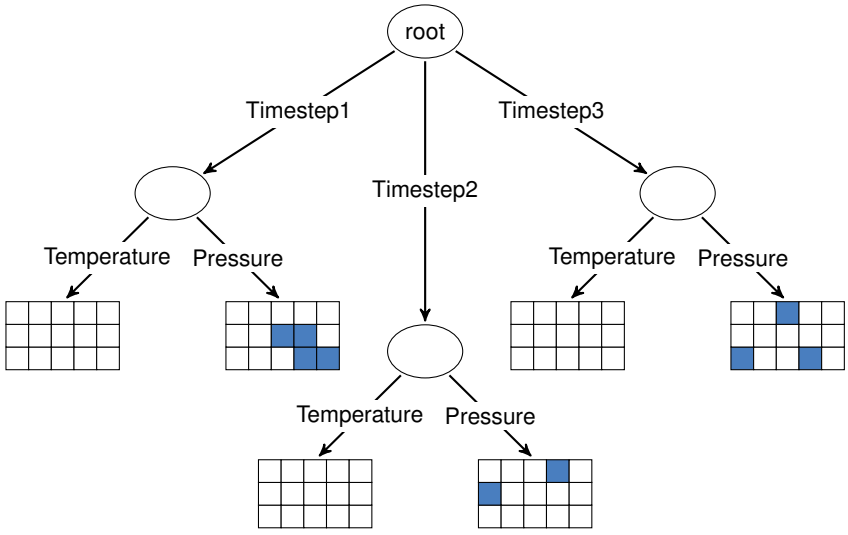

Figure 4 HDF5 container example with query $(link\_name = Pressure) \wedge (data\_element = 17)$ applied.

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 $(link\_name = Pressure) \wedge$

($data\_element = 17$) 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.

4 Index Objects

The final component of the data analysis extensions to HDF5 is the index object. Index objects are designed to accelerate creation of view objects from frequently occurring query operations. For example, if the ($link\_name = Pressure$) $\wedge$ ($data\_element = 17$) query (described above) 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 ($link\_name = Pressure$) $\wedge$ ($data\_element = 17$) 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.

Must write about metadata indexing

The indexing API works 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. We therefore define a new interface for the HDF5 library that uses a plugin mechanism.

4.1 Indexing Interface and Plugins

A new HDF5 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 file_id, unsigned plugin_id, hid_t scope_id,
        hid_t xcpl_id);
herr_t H5Xremove(hid_t file_id, unsigned plugin_id, hid_t scope_id);
herr_t H5Xget_count(hid_t scope_id, hsize_t *idx_count);
```

```
8
9  typedef struct {
10     unsigned version;      /* Version number of the index plugin class struct */
11                            /* (Should always be set to H5X_CLASS_VERSION, which
12                             *  may vary between releases of HDF5 library) */
13     unsigned id;           /* Index ID (assigned by The HDF Group, for now) */
14     const char *idx_name; /* Index name (for debugging only, currently) */
15     H5X_type_t type;       /* Type of data indexed by this plugin */
16
17     /* Callbacks, described above */
18     void *(*create)(hid_t file_id, hid_t dataset_id, hid_t xcpl_id,
19         hid_t xapl_id, size_t *metadata_size, void **metadata);
20     herr_t (*remove)(hid_t file_id, hid_t dataset_id, size_t metadata_size,
21         void *metadata);
22     herr_t (*copy)(hid_t file1_id, hid_t file2_id, hid_t dataset1_id,
23         hid_t dataset2_id, hid_t xcpl_id, hid_t xapl_id,
24         size_t metadata_size, void *metadata);
25     void *(*open)(hid_t file_id, hid_t dataset_id, hid_t xapl_id,
26         size_t metadata_size, void *metadata);
27     herr_t (*close)(void *idx_handle);
28     herr_t (*pre_update)(void *idx_handle, hid_t dataspace_id, hid_t xxpl_id);
29     herr_t (*post_update)(void *idx_handle, const void *buf, hid_t dataspace_id,
30             hid_t xxpl_id);
31     herr_t (*query)(void *idx_handle, hid_t query_id, hid_t xxpl_id,
32         hid_t *dataspace_id);
33     herr_t (*refresh)(void *idx_handle, size_t *metadata_size, void **metadata);
34     herr_t (*get_size)(void *idx_handle, size_t *idx_size);
35 } 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[5]. Instead, index objects are part of the metadata for the file itself. New index objects are created by passing an H5 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 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. Similarly, calling `H5Dclose` will call the plugin index close callback and close the objects used to store the index data.

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

[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] This therefore introduces a file format change

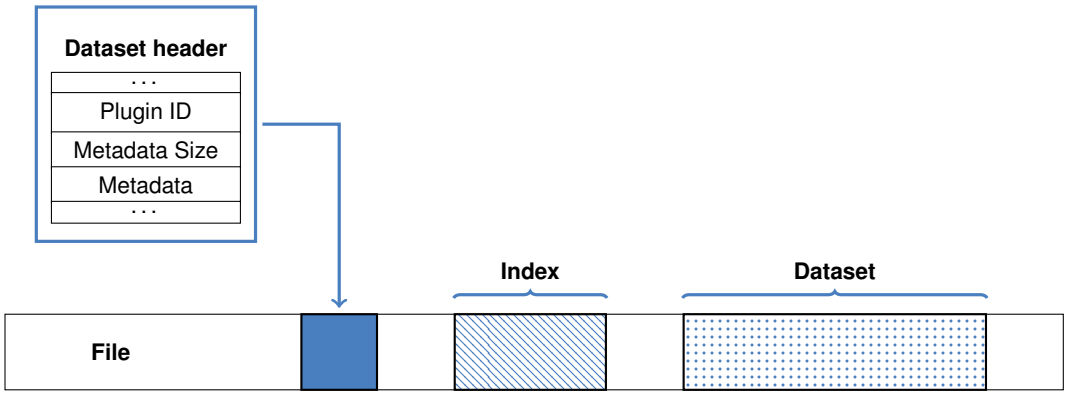

Figure 5 Index information (plugin ID and metadata) is stored along the object header.

new `H5Dquery` routine to directly execute a query on a dataset (accelerated by any index defined on the dataset), retrieving the selection within that dataset that matches the query.

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.

Add something about parallel queries / multi dataset serial / multi-query paragraph

4.1.1 Current and future plugins

Current 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.

In the future more plugins will be added, with and without external dependency (e.g., pytable 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 don't 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 they can make incremental updates to their index information.

Questions

Some questions are still open regarding the handling on indices:

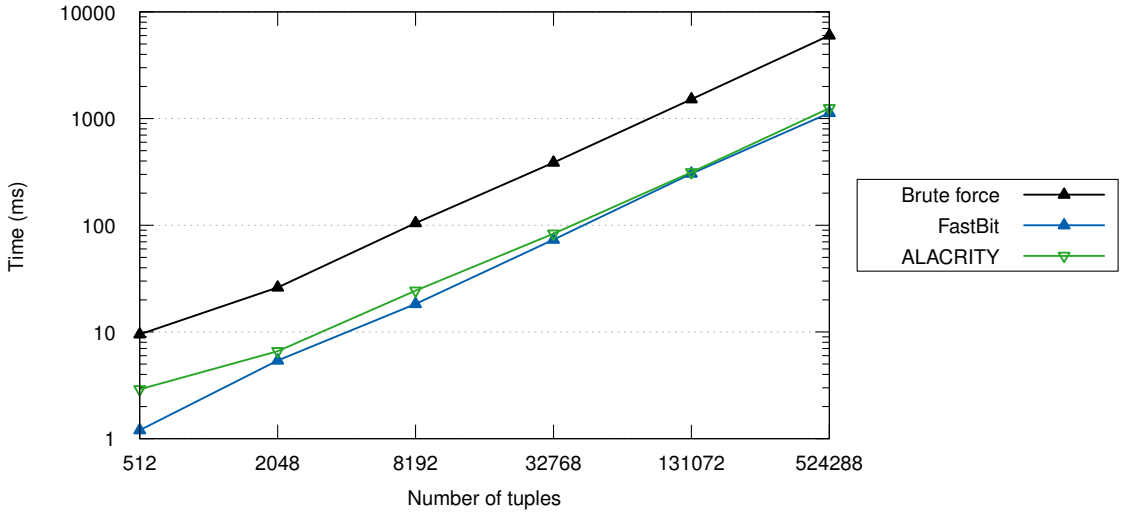

Figure 6 Indexing performance.

- How could an index be built and queried in parallel for a dataset that already exists?
- 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.
- 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.

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.

6 Usage Example

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

```
1  #define NTUPLES 256
2
3  int
4  main(int argc, char *argv[])
5  {
6      float data[NTUPLES];
7      hsize_t dims[1] = {NTUPLES};
8      hid_t t file_id, dataspace_id, dataset_id;
9      hid_t query_id, result_space_id;
10     size_t result_npoints;
11     float *result;
12     int i;
13
14     /* Initialize data. */
15     for(i = 0; i < NTUPLES; i++) data[i] = (float) i;
16
17     /* Create file. */
18     file_id = H5Fcreate(file_name, H5F_ACC_TRUNC, H5P_DEFAULT, H5P_DEFAULT);
19
20     /* Create the data space for the dataset. */
21     dataspace_id = H5Screate_simple(rank, dims, NULL);
22
23     /* Create dataset. */
24     dataset_id = H5Dcreate(file_id, "Pressure", H5T_NATIVE_FLOAT, dataspace_id,
25         H5P_DEFAULT, H5P_DEFAULT, H5P_DEFAULT);
26
27     /* Write dataset. */
28     H5Dwrite(dataset_id, H5T_NATIVE_FLOAT, H5S_ALL, H5S_ALL, H5P_DEFAULT, data);
29
30     /* Close the dataset. */
31     H5Dclose(dataset_id);
32
33     /* Close dataspace. */
34     H5Sclose(dataspace_id);
35
36     /* Open dataset. */
37     dataset_id = H5Dopen(file_id, "Pressure", H5P_DEFAULT);
38
39     /* Create index using FastBit. */
40     H5Xcreate(file_id, H5X_PLUGIN_FASTBIT, dataset_id, H5P_DEFAULT);
41
42     /* Close the dataset. */
43     H5Dclose(dataset_id);
44
45     /* Create a simple query */
```

```
    query_id = H5Qcreate(H5Q_TYPE_DATA_ELEM, H5Q_MATCH_EQUAL, H5T_NATIVE_FLOAT,
        &query_value);

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

    /* Use query to select elements in the dataset. */
    result_space_id = H5Dquery(dataset_id, query_id);

    /* Allocate space to read data. */
    result_npoints = (size_t) H5Sget_select_npoints(result_space_id);
    result = malloc(result_npoints * sizeof(float));

    /* Read data using result_space_id. */
    H5Dread(dataset_id, H5T_NATIVE_FLOAT, H5S_ALL, result_space_id,
        H5P_DEFAULT, result);

    /* Use result. */

    /* Free result. */
    free(result);

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

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

    /* Close query. */
    H5Qclose(query_id);

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

7 Conclusion

Revision History

July 17, 2014: Version 1 circulated for comment within The HDF Group.

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. KÃijng, 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.