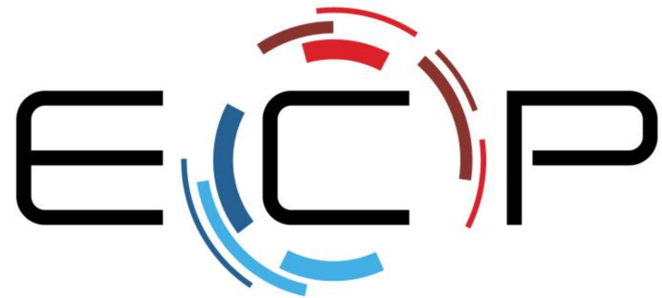


DRAFT

Design Document: Parallel Query and Indexing in HDF5

20 December 2018

DOCUMENT AVAILABILITY

Reports produced after January 1, 1996, are generally available free via US Department of Energy (DOE) SciTech Connect.

Website www.osti.gov

Reports produced before January 1, 1996, may be purchased by members of the public from the following source:

National Technical Information Service
5285 Port Royal Road
Springfield, VA 22161
Telephone 703-605-6000 (1-800-553-6847)
TDD 703-487-4639
Fax 703-605-6900
E-mail info@ntis.gov
Website http://classic.ntis.gov/

Reports are available to DOE employees, DOE contractors, Energy Technology Data Exchange representatives, and International Nuclear Information System representatives from the following source:

Office of Scientific and Technical Information
PO Box 62
Oak Ridge, TN 37831
Telephone 865-576-8401
Fax 865-576-5728
E-mail reports@osti.gov
Website http://www.osti.gov/contact.html

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

DRAFT

Design Document: Query and Indexing in HDF5

Office of Advanced Scientific Computing Research
Office of Science
US Department of Energy

Office of Advanced Simulation and Computing
National Nuclear Security Administration
US Department of Energy

Date

DRAFT v.0.1

ECP Milestone Report
EXASCALE COMPUTING PROJECT (ECP)

APPROVALS

Submitted by:

Jerome Soumagne, Richard Warren, The HDF Group Exascale Computing Project	Date

Concurrence:

Suren Byna Lawarence Berkeley National Laboratory	Date

Approval:

Kathlyn Boudwin, Project Management Director Exascale Computing Project	Date

EXASCALE COMPUTING PROJECT

TITLE OF DOCUMENT
VERSION CHANGE CONTROL TABLE

Version	Creation Date	Description	Approval Date
1.0	July 15, 2016	Original	

TABLE OF CONTENTS

APPROVALS iii

VERSION CHANGE CONTROL TABLE v

LIST OF FIGURES ix

LIST OF TABLES ix

ACRONYMS xi

EXECUTIVE SUMMARY xiii

1. HEADING 1 STYLE 1
   - 1.1 HEADING 2 STYLE ..... **Error! Bookmark not defined.**~~1~~
2. HEADING 1 STYLE 1
   - 2.1 HEADING 2 STYLE ..... **Error! Bookmark not defined.**~~1~~
   - 2.2 HEADING 2 STYLE ..... **Error! Bookmark not defined.**
   - 2.3 HEADING 2 STYLE ..... **Error! Bookmark not defined.**~~5~~
   - 2.4 HEADING 2 STYLE ..... **Error! Bookmark not defined.**~~7~~
     - 2.4.1 Heading 3 Style ..... **Error! Bookmark not defined.**~~7~~
     - 2.4.2 Heading 3 Style ..... **Error! Bookmark not defined.**~~7~~
3. HEADING 1 STYLE ..... **Error! Bookmark not defined.**~~7~~
   - 3.1 HEADING 2 STYLE ..... **Error! Bookmark not defined.**~~7~~
4. HEADING 1 STYLE ..... **Error! Bookmark not defined.**~~7~~
5. HEADING 1 STYLE ..... **Error! Bookmark not defined.**~~7~~
6. ACKNOWLEDGMENTS [HEADING 1 STYLE] ..... 15~~7~~
7. REFERENCES [HEADING 1 STYLE] ..... 16~~7~~

APPENDIX A. TITLE [HEADING 9 STYLE] ..... **Error! Bookmark not defined.**~~A-1~~

LIST OF FIGURES

Figure 1. FIGCAP 1 line. .... **Error! Bookmark not defined.**~~1~~

LIST OF TABLES

Table 1. Table Caption. .... **Error! Bookmark not defined.**

ACRONYMS

INSERT ACRONYMS

ECP EXASCALE COMPUTING PROJECT

EXECUTIVE SUMMARY

Methods to store, move, and access data across complex exascale architectures are essential to improve the productivity of scientists. To assist with accessing data efficiently, we propose to integrate functionality for querying raw data within the HDF5 library, allowing application developers to create queries on data elements within an HDF5 dataset. The interface relies on three main components: queries, *views and indexes*. *Query* objects are the foundation of the data analysis operations and can be built up from simple components in a programmatic way to create complex operations using Boolean operations. *View* objects allow the user to retrieve an organized set of query results. *Index* objects are the core means of accelerating the HDF5 query interface and enable queries to perform efficiently. The indexing interface uses a plugin mechanism that enables flexible and transparent index selection and allows for third-party indexing plug-ins to be separately packaged and loaded.

This milestone report documents the integration plan for the query and index feature of HDF5 raw data. The integration will ensure that the query interface is robust and efficient for workloads relevant to ECP stakeholders. This report includes preliminary results from a prototype that was previously developed and serves as a basis for our integration work. This report also documents how the feature will be expanded and improved for parallel query and indexing of raw data as well as relevant use cases.

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 its content (i.e., 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 generate a selection, which in turn contains a set of coordinates satisfying the query. However, generating a selection may 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 generate indexes and use these indexes to directly find the coordinates of the matching elements in order to answer the specified query. A previous HDF5 prototype introduced new object types to support these operations:

- HDF5 *query* objects and API routines, enabling the construction of query requests for execution on HDF5 containers;

- HDF5 index objects and API routines, which allow the creation of indexes on the contents of HDF5 containers, to improve query performance

These new HDF5 API functions and data structures that are designed to allow *plug-ins* as a means of incorporating third party libraries such as *fastbit*, to improve performance and versatility. The remainder of this design document focuses on extending these APIs and data structures to allow parallel processing to be leveraged with the goal of improving scalability and overall performance of querying ever larger scientific datasets.

2. TECHNICAL WORK SCOPE, APPROACH, RESULTS

This section outlines in detail, the initial prototype and results.

2.1 DESIGN CONSIDERATIONS

The HDF5 library is designed to incorporate parallelism into the reading and writing of files by utilizing the features of MPI-IO. Along with several restrictions (e.g. a requirement to write HDF5 metadata collectively), these features will apply in our parallel extension of the original Q&I prototype. In addition, the new parallel prototype will need to address the subject of *how* and *where* indexed results are to be stored. There are two alternatives which we consider:

- Storing index results as *anonymous* datasets within the original HDF5 file;

- Storing index results into node specific datasets which are written into separate independent files.

Irrespective of the selected approach, the design of the object reference type needs to be independent of the number of processes used to generate results.

2.1.1 Parallelization of HDF5 Query and Index APIs

The most common approach to parallelize indexing and querying is to divide the original input data into blocks or chunks and then to query/index each block independently. This is the approach used in

FastQuery, DRIAQ (parallel ALACRITY), PIQUE, and is proposed for this project as well. This approach leverages the standard parallel access methods for datasets currently in use within the parallel HDF5 community, i.e. by creating a global *file-view* of a dataset and then having a parallel process/rank specific *memory-view* of the local data to be queried. These views are created using an HDF5 hyperslab description and allow for a broad range of "slicing and dicing" user options, the most common of which is to divide the *last-dimension* of an N-dimensional array size by the number of MPI ranks to produce the number of local data entries to be processed. Data offsets when writing into the global file-space view in this parallel example, are typically computed by multiplying the number of elements-per-rank by the MPI rank of the process (MPI_Comm_rank). This is an important consideration to retain since each parallel process would generally contain reference indices which are relative to its local *memory-view* of the data read from the specified dataset. Handling these partial chunk results can be accomplished in several ways:

- Each reference is modified to incorporate the process local base index. Example: If 100 MPI ranks generate results in which the first element in its *memory-view* of a dataset is a match to the query, then the index returned locally would be zero (0). Assuming 1000000 elements per MPI rank need to be examined, then MPI rank 0 should return index 0, MPI rank 1 would return 1000000, MPI rank 2 would return 2000000, etc, and MPI rank 99 would return 99000000.

- Results are returned as blocks of indices. Under this condition, only the block metadata needs to retain the MPI rank specific local offset. All results within that block can be treated as offsets from that process specific base.

- Chunks or Blocks need not be related to MPI size or MPI rank. They can be of an arbitrary fixed size with an initial base calculated from the original *memory-view* hyperslab definition.

Another consideration is that the design space that we are investigating allows asymmetry in the number of query results that are produced. Unlike the read_data parallel operations in which the total number of elements per MPI rank is initially fixed once a dataset size is determined, the writing of indexed results is unknown until a query has been evaluated. As a consequence, the generation of a coherent set of results requires additional communication between MPI ranks such that all processes, including those ranks not having any positive query results, enter the collective write to produce the same results that a serial application would generate. Lastly, we need to consider the storage of fastbit internal information. Each process in the parallel application is effectively a unique instance of the fastbit Query and Indexing engine. Once a dataset is indexed in parallel, and the internals of fastbit are serialized for storage, we need to determine whether a simple concatenation of these serialized structures is valid for future use. The alternative would be to undertake additional communication such that a single MPI rank does the serialization and storage of all fastbit results. For now, we assume that the concatenations approach will be sufficient. From the HDF5 perspective, the serialized fastbit indexing structures are opaque and cannot be combined without helper functions from the fastbit library itself. This introduces some inefficiencies in the performance of parallel operations when querying is separated from the generation of the fastbit indices. *One of the explicit goals of this project, however, is that the number of query processes may differ from the number of processes used to generate dataset indices.*

The sections which follow will focus further on the actual details of each of these topics.

2.1.2 The PLUGIN interface

This section describes the current PLUGIN interface for HDF5. In addition to the HDF5 specific APIs to specify a query, combine queries, and ultimately to apply queries and record the results as HDF5 reference types, there exists an internal plugin API which allows users to select specific third-party

libraries as query engines for their applications. These plugin APIs are designed to closely match the higher level HDF5 functions invoked by the user. The original software development effort included support for the following third-party libraries:

- FastBit -- A Lawrence Berkeley National Lab project: FastBit is a minimalistic data warehousing engine designed to test ideas on bitmap indexes.
- Alacrity-ADIOS -- A collaborative effort between NC State, ORNL, Sandia National Lab, and Argonne National Lab: Analytics-driven Lossless Data Compression for Rapid In-situ Indexing, Storing, and Querying.
- Dummy -- a minimalistic test engine used to validate the plugin approach. This is sometimes referred to as the "brute force" method since no data indexing is available and the implementation is thus forced to read the entire dataset before a query can be evaluated.

From the above list, we no longer anticipate continued support for *Alacrity*. The *Dummy* plugin will continue to be a testbed for query enhancements, e.g. allowing multiple query conditions to be evaluated in the context of a single pass through a dataset. A simple example of this optimization would be to evaluate a range check per data element: $10 < x < 20$ as a single condition check rather than forcing two passes through a dataset to first produce the set of matches in which $10 < X$; followed by a second pass which gathers the results of $X < 20$, and then set merge operation combining partial result into a final result.

The following is the basic Data index class structure that is initialized with plugin specific function

```
/* 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); /* TODO pass datatype id */
    herr_t (*remove)(hid_t file_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 (*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);
    hid_t  (*query)(void *idx_handle, hid_t dataspace_id, hid_t query_id,
        hid_t xxpl_id);
    herr_t (*refresh)(void *idx_handle, size_t *metadata_size, void **metadata);
    herr_t (*copy)(hid_t src_file_id, hid_t dest_file_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 (*get_size)(void *idx_handle, hsize_t *idx_size);
} H5X_data_class_t;
```

pointers:

2.1.2.1 Create

The plugin *create* function is invoked as part of the normal processing of H5Dcreate2. The input arguments are:

- The dataset ID that will eventually be returned to the user from H5Dcreate2; Note too, that the plugin specific initialization function is called prior to calling the create function.
- An index property created from H5Pcreate(H5P_INDEX_CREATE);
- A default index access id (H5P_INDEX_ACCESS_DEFAULT) which is currently unused.
- A pointer to a length field for the metadata (output)
- A pointer to the metadata (output).

This API is unchanged for parallel query operations.

2.1.2.2 Remove

Implemented, but not used in current implementation; no change for parallel operations.

2.1.2.3 Open

The plugin *open* function is invoked as part of H5Qapply call and opens an existing index from the file (if one exists). The input arguments are:

- The dataset ID that was used to create the dataset (see Create)
- A default index access id (H5P_INDEX_ACCESS_DEFAULT) which is currently unused.
- A pointer to a length field for the metadata (output)
- A pointer to the metadata (output).

This API will likely require some work as part of the parallel implementation. In particular, there are questions about how and where indexing results will be stored.

Questions:

- How would indexing results be accessed from the various ranks of a parallel implementation?
- Is it convenient that each MPI rank read all of the metadata or should each rank somehow access only the index info pertaining to its current local data?

Returns a pointer to an opaque object created and then used by the plugin implementation to carry interesting information during the entirety of the query/indexing operations. This opaque object has content specific to the parallel implementation.

2.1.2.4 Close

The plugin *close* function is invoked as part of H5Qapply call and closes the index which was opened by the plugin *open* function. The input arguments are:

- Index handle: A pointer the opaque object returned by the plugin *open* call.

This API is unchanged for parallel query operations.

2.1.2.5 Pre_update

The plugin *pre_update* function is invoked as part of writing data to a dataset via H5Dwrite and provides an opportunity to update the plugin specific structure that was returned by the plugin *open* function. The input arguments are:

- Index handle: A pointer the opaque object returned by the plugin *open* call.
- Filespace id: In the parallel implementation, we copy this filespace descriptor into the plugin opaque object since it represents the hyperslab selection used for accessing the dataset on this specific MPI rank.
- A default index access id (H5P_INDEX_ACCESS_DEFAULT) which is currently unused.

As mentioned, the only change in this API in support of a parallel query is the caching of the filespace_id.

2.1.2.6 Post_update

The plugin *post_update* function is invoked as part of writing data to a dataset via H5Dwrite. It does an update of the indexing information by first reading the dataset, merging the old and new data, and then rebuilding the index. The input arguments are:

- Index handle: A pointer the opaque object returned by the plugin *open* call.
- Data pointer: A pointer to the new dataset data.
- Dataspace_id: The dataset descriptor
- a default index access id (H5P_INDEX_ACCESS_DEFAULT) which is currently unused.

The internal call to read data from a dataset before merging the write_data is similar to what one would expect in terms of H5Dread behavior. Unlike the H5Dread however, a parallel query operation would not behave correctly were it not for the Pre-update caching of a filespace_id (provided by the user as part of a parallel write data slice). The internal read operation needs to reuse that hyperslab selection descriptor to access the rank specific data within the dataset.

As mentioned above, subsequent to reading existing data and then merging the newest write_data, the plugin builds or rebuilds an index. The results of the build_index operation are currently written into anonymous datasets. In the parallel implementation, indexing size-per-rank is non-deterministic, but because file write operations must be collective, some additional work needs to be undertaken to:

a) determine the overall size of each new dataset; and
b) set the access property list to utilized parallel IO; and
c) exchange local dataset size information to allow the proper file offsets to be determined.

Otherwise, the API calling conventions are unchanged for parallel query operations.

2.1.2.7 Query

The plugin *query* function is invoked as part of the H5Qapply functionality. As the name implies, it applies the user query to the specified dataset and returns a selection if one exists. The input arguments are as follows:

- Index handle: A pointer the opaque object returned by the plugin *open* call.
- Dataspace_id: A dataset descriptor (unused since each instance of the plugin opaque object retains the id of the dataset used in the *create* step).
- Query_id: The id returned by an H5Qcombine or H5Qcreate function.
- a default index access id (H5P_INDEX_ACCESS_DEFAULT) which is currently unused.

The return value is a selection spaceid.

This API is unchanged for parallel query operations.

2.1.2.8 Refresh

The plugin *refresh* function is invoked as part of the H5Dwrite operation and is tasked with refreshing the index information once the write operation has been merged with previous indexing information. The input arguments are as follows:

- Index handle: A pointer the opaque object returned by the plugin *open* call.
- Metadata_size pointer: a pointer to a length field for the metadata (output)
- Metadata pointer: a pointer to the metadata (output).

This API is unchanged for parallel query operations.

2.1.2.9 Copy

Unused

2.1.2.10 Get_size

The plugin *get_size* function returns the storage size of the index. The input arguments are as follows:

- Index handle: A pointer the opaque object returned by the plugin *open* call.
- Pointer to an index size information. (output)

This API is unchanged for parallel query operations.

2.1.3 FastBit Plugin support – H5X_fastbit_t

The H5X_fastbit_t structure is an HDF5 specific object which contains HDF5 and Fastbit library context details during the various HDF5 dataset create/write/read operations.   The structure is allocated with the initial reference a dataset, e.g. as a consequence of the user calling H5Dcreate2(). The structure is freed/released once the dataset has been created and closed, e.g. with H5Dclose().   Between these operations, the structure provides the consistent state information which is used for creating fastbit indices and other metadata used by HDF5.

Introducing parallelism forces some additional metadata into the structure. This information needs to be tracked and eventually serialized for storage and subsequent retrieval and use.   Examples of this additional metadata include:

- *Number* of MPI ranks
- *Offsets* and *sizes* used to recover the fastbit indexing information when required. We discussed earlier, that the *divide and conquer* approach of reading datasets in parallel and applying the fastbit indexing engine on a per-rank basis, introduces asymmetric results which are effectively concatenated into anonymous datasets for future access.   Because of that asymmetry, we must maintain each the individual fastbit container size and offset within the anonymous datasets.
- *Layout information* to allow subsequent readers (query processes) to duplicate the memory context for which the indexing metadata (offsets and sizes: see above) is valid.

2.2 DATASET READ AND WRITE – (THE GENERATION AND CONSUMPTION OF DATASET INDICES)

It is important at this point to recall that an explicit goal of the parallel design is to allow the separation of the generation of dataset indices from the consumption of that indexing information when processing queries. This implies that one must envision a scenario in which datasets are written and indexed with

one application; and then in a completely distinct program execution, with a different number of parallel processes (possibly only one), queries are applied to those datasets. This scenario highlights the fact that whatever information we require for querying, we must provide that data as persistent information in the HDF5 file or collection of associated files. The initial parallel prototype will continue to utilize HDF5 anonymous datasets for writing the query plugin specific information. The following describes the collected information which is specific to the *fastbit* plugin. The new parallel context contains:

- Identification of the particular plugin used to generate the indexing information.
- The number of processes and dataset layout information, i.e. how the dataset is partitioned between processors.
- Per-Rank data about the size and starting offset within each of the three (3) anonymous datasets for fastbit: *keys, offsets*, and *bitmaps*.

The importance of this metadata is that on a per-dataset basis, it ensures that the correct memory image is made available for each process when recovering fastbit indexing information and for applying queries.

One critical difference between parallel indexing and sequential is that as a consequence of splitting the actual dataset data into "chunks" with each assigned to a different processor, the resulting fastbit metadata information will not match that of the single processor approach. More concretely, the concatenation of the collection of fastbit indices in parallel will not match the size and content of that produced by the serial process. As a result, any subsequent query processing will need to iterate thru the collection of individual chunked indexing contexts to gather a global query result. This iteration process can occur as a single process or by applying parallelism with a reduction to accumulate the final query results.

Table 1 below, illustrates the metadata sizes for each of the three anonymous datasets which are produced when processing a simple query to locate elements within an 'Energy' dataset of filtered VPIC data which have values greater than 1.9.

Table 1. Fastbit metadata example

	Fastbit->keys (bytes)	Fastbit->offsets (bytes)	Fastbit->bitmaps (bytes)
Serial 36788476 elements	**36**	**19**	**6649850**
Parallel 18394238 elems (rank 0)	34	18	3291452
Parallel 18394238 elems (rank 1)	36	19	3358153

The above table shows a few interesting facts about the current prototype:

1. The upper half of the parallel dataset has the same data elements as the lower half of the dataset, but the metadata entries differ in size.
2. The sum of the parallel metadata sizes does not match those of the serial metadata sizes. This confirms what we should expect, i.e. that a simple concatenation of the various meta buffers is not the equivalent of that produced by a single process indexing with fastbit.

2.2.1 Dataset Querying (reading)

Query processing is a user directed activity which directs the library to employ specific plugins to improve query performance. With regards to our primary focus of utilizing FastBit, we can characterize the query processing into two major categories:

- An HDF5 dataset is specified for query processing and it does NOT currently include indexing metadata; or
- An HDF5 dataset is specified for query processing and it already contains metadata indexing information (see table 1 above).

With FastBit enabled via the "`H5Xcreate(dset, plugin, H5P_DEFAULT)`" function, the user may simply invoke query processing in parallel by creating the query and then by invoking H5Qapply() to create a "view" which represents the collected results from the plugin of choice.

```
/* Create query */
static hid_t
test_query_1(float value1)
{
    hid_t q1 = H5I_BADID;
    hid_t q2 = H5I_BADID;
    hid_t q3 = H5I_BADID;

    /* Create and combine a bunch of queries
     * Query is: (x > value1)
     */
    if ((q1 = H5Qcreate(H5Q_TYPE_DATA_ELEM, H5Q_MATCH_GREATER_THAN, H5T_NATIVE_FLOAT, &value1)) < 0)
FAIL_STACK_ERROR;

    /* Select object */
    if ((q2 = H5Qcreate(H5Q_TYPE_LINK_NAME, H5Q_MATCH_EQUAL, link_name)) < 0) FAIL_STACK_ERROR;
    if ((q3 = H5Qcombine(q2, H5Q_COMBINE_AND, q1)) < 0) FAIL_STACK_ERROR;

    /* H5Q_enable_visualize_query(); */
    return q3;


error:
    H5E_BEGIN_TRY {
        if (q1 != H5I_BADID)
            H5Qclose(q1);
        if (q2 != H5I_BADID)
            H5Qclose(q2);
        if (q3 != H5I_BADID)
            H5Qclose(q3);
    } H5E_END_TRY;
    return -1;
}
```

NOTE: Need to discuss the hyperslab selection mechanism. When writing or reading data, the current approach is to divide the last dimension of an input array by the number of MPI ranks to determine the per-rank data size. The hyperslab offsets are all calculated based on this number. For 2D or 3D arrays,

should we provide alternate mechanisms for reading and writing, e.g. by row or by column; or the possibly of reading by blocks? How do we capture this information (see point 3 above)?

Comment [RW1]:

One should note that a user is only required to open the relevant HDF5 file to pass the query to the library for processing. Under these conditions, if the plugin requires indexing information as in the case of using FastBit and no index information is currently available, then the library is itself responsible for subdividing the applicable datasets into parallel data slices and then invoking the indexing plugin to generate indexing metadata. As we have noted previously, this metadata is subsequently written into the HDF5 file as one or more “anonymous” datasets.

By default, the library will subdivide a given dataset in this situation into N contiguous data slices, where N represents the number of MPI processes. Further subdivision into fixed sized blocks is highly desirable to ensure that any subsequent processing by larger numbers of processors is relatively efficient.

Efficiency for this discussion is merely the question of whether subdivisions of parallel data slices can take place during future query processing without incurring too much extra processing time. We’ve noted that the concatenation of the plugin metadata information is not possible without explicit support by the underlying plugin library. Similarly, any subdivisions of this same metadata are not possible without similar support. The only alternative to subdivision then is one of recalculation. For datasets in the multi-megabyte size, we have observed the times to build an index dataset to be on the order of tens of seconds. This extra processing time can potentially be avoided by utilizing the fixed blocksize approach outlined above. For this reason, we expect to focus the implementation on using fixed blocksize indexing as we move past the initial prototype stage.

When a parallel query is presented to the HDF5 library for processing and the datasets to be queried **have previously been indexed,** then the library is expected to utilize the existing index information rather than forcing a recalculation of the index information. We have discussed already, that utilizing fixed blocksizes is an approach which can remove the necessity for recomputation of indices in many situations. Eventually, however, we will likely find situations in which the number of parallel processes being utilized for querying is larger than the number of indexed blocks. Under this condition, it seems plausible that the best approach would then be to recalculate the indices based on the number of MPI ranks rather than have processes which have no “work” to do.

2.2.2 Dataset Writes (Index creation)

As with the read operations that we discussed in the previous section, *write operations* with plugin indexing will be enabled by associating the desired plugin with the open file or a specific dataset using by setting a H5P_DATASET_CREATE property: `H5Pset_index_plugin(dcpl, plugin).`

As the user creates and then writes data to a dataset, the plugin interfaces for “pre_update”, “create”, and “post_update” are called. The plugin specific libraries are called as required as data is written and indexing information is returned to the plugin for archiving, i.e. saved to HDF5 anonymous datasets.

Writes to a pre-existing dataset will force a re-indexing operation to take place and to overwrite the existing metadata. As a consequence, users might select smaller fixed blocksizes if/when they expect to update small portions of an existing dataset frequently. This approach would “limit the damage” and prevent large, expensive updates of metadata when only a small portion of the overall data is modified.

2.2.3 Views

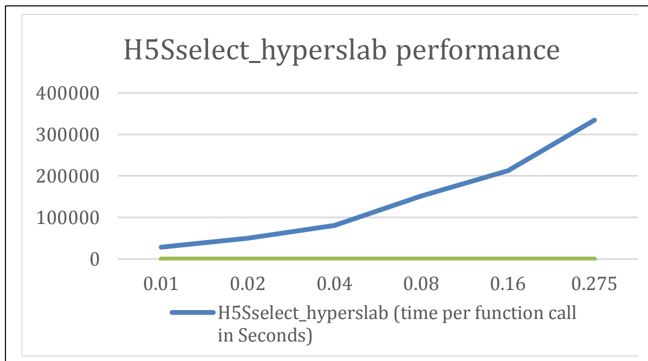
Applying a query to an HDF5 container or dataset creates a *view*, a set of references that match the query. A view is a container stored in memory, hence temporary. It is composed of a group that contains 1-D dataset objects. These objects contain references to the matching elements that were queried. In the case of raw data queries, these references are HDF5 dataset region references which represent a selection of elements within an HDF5 dataset. 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. This should be automatically done via a predetermined watermark for large datasets to prevent memory overflows as results are collected from blocks of indexed data. For coherency, a time stamp 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 also be useful in the future to define different states to the view so that the user knows whether the view is current or not.

A new view is created by a call to the `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. The attributes, objects, and/or data regions referenced within a view's datasets can be retrieved by further HDF5 dataset `(H5D*)` API calls. In some instances and depending on the nature of the query, applying a query may result in a large amount of data being stored in the view. It is therefore also important for the user to be able to limit the size of the results being returned, that limit can be passed through the `vcpl_id` parameter (see below). Similarly, an existing selection on the queried data could be used to narrow down the query search.

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

2.2.4 Hyperslab selection

Apart from the indexing plugins themselves, a major contributor to the current query performance is the time spent in collecting region reference results into a *view*. The existing `H5Sselect_hyperslab()` function adds each dataset coordinate to its local view by concatenation (H5S_SELECT_OR). The original code accomplished this by appending each new element to the end of a singly linked list and was thus an N^2 operation. The newer hyperslab code has improved this by providing a list *tail* pointer to allow list appends in constant time. The table below shows how the current H5Sselect_hyperslab



The table below shows how the current H5Sselect_hyperslab performance relative to the number of data points previously selected. It should be clear that for large selections, the cumulative effect of calling this function to append a new value to the list is considerable. As the prototype moves forward, we will revisit these performance checks to see whether the hyperslab selection code has improved.

2.3 THE LAYOUT CONSIDERATIONS FOR QUERY PROCESSES

A basic premise of introducing parallelism for query/indexing is that for any dataset, the user will subdivide the logical image which is stored on disk, into data-slices of equivalent size on each parallel rank such that the indexing and eventual querying can be accomplished in an accelerated fashion. In the ideal world, applying N processors to a fixed sized problem will produce results in 1/Nth the time. This is the ultimate, though rarely achieved goal of parallel processing.

Applied to Indexing and Querying, the major task when running an indexing scheme which differs in the scale of parallelism applied with that of the query processing, is that each query process needs to replicate the in-memory footprint of each indexing process to be able to utilize the index metadata. There are two extremes in which a solution is obvious:

1. If the number of query processes matches that used for indexing, then each query process will subdivide the logical image of each dataset exactly as was used to generate the indexing metadata. Each query process will then apply the user query to its local data slice exactly once and then communicate the number and indexing information to the rank 0 process for presentation to the user.

2. If the number of query processes is exactly one (1), then the entire dataset will be read by the serial process and the collection of parallel index contexts will be read sequentially; the user query applied within that context; and the results accumulated locally for presentation to the user.

For processing situations which do NOT fit one of these two extremes, we need to examine how each parallel query process is defined in terms of its memory image. The following sections examine a few possible scenarios when the number of parallel query ranks is something other than the two extremes already described.

2.3.1 Layout selection by the Last Dimension

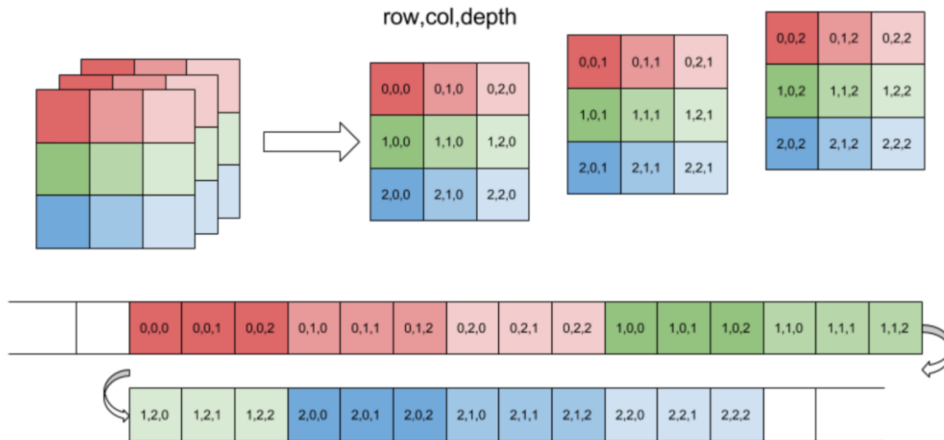
The easiest and perhaps most common scheme for subdividing arrays is to imagine that the contiguous block of memory assigned to a sequential process is broken into N contiguous chunks, where N is the number of parallel application processes assigned to the problem. In this scenario as in others, the total number of indexing ranks and chunk sizes can be extracted from the stored metadata. With this information and the number of parallel ranks used for querying, we can divide $N_{indexing}$ by $N_{querying}$ to determine the number of chunks that each query instance needs to process. If that calculation does not divide equally, then one or more query processes will have additional iterations to process. If $N_{querying}$ for some reason is larger than $N_{indexing}$, then only $N_{indexing}$ number of query process will participate in the querying activity. (*Note: this might require an MPI_Comm_split to be used for the query reductions rather than using a MPI_Comm_dup*).

The number of iterations per query rank is determined before any actual query processing takes place. Similarly, the hyperslab selection criteria are easily determined so that dataset reads can take place as needed.

2.3.2 Layout selection by Row

Selection by row is somewhat similar to selection by subdividing by the last dimension. In the 'C' programming language, arrays in memory are contiguous by rows. Obviously, for 2-D arrays, last dimension and selection by row are equivalent.

For 3-D arrays, a selection by row requires does not collect contiguous blocks of memory, rather it assigns whole rows to individual ranks using a strided access in memory to gather the necessary data. The following diagram shows the logical and memory layouts of a 3x3x3 array. Assignment of any 2D plane to a processor, for example, would require strided access the entire dataset. This is obviously less efficient in terms of performance.



Irrespective of the inefficiency of memory access, the querying process access requirements are to gather its share of the total number of indexing blocks into its memory and iterate over those by applying the

user query against each index block. Calculation of the number of iterations required is exactly that which we described in the introduction of this topic.

2.3.3 Layout selection by Column

Selection by column will be similar to that of selection by Row but is potentially only an interesting option if the original dataset is written using Fortran. The reason for this being the fact that rather than Row Major memory ordering for arrays, Fortran uses a Column major storage order. The following diagram shows the logical view and its corresponding memory layout:

2.3.4 Layout selection by Block

2.4 RESULTS

The prototype code which implements the parallel indexing and parallel queries and uses the FastBit plugin was used to validate some interesting performance parameters. In particular, we utilized a dataset from NERSC which consists of filtered VPIC data. Several examples were used, but the following graphs show some of the scaling behavior of a simple query against a dataset of 623420550 elements per object:

```
/project/projectdirs/m1248/vpic/converted_particles/T22860/eparticle_T22960_1.5_filter.h5p
```

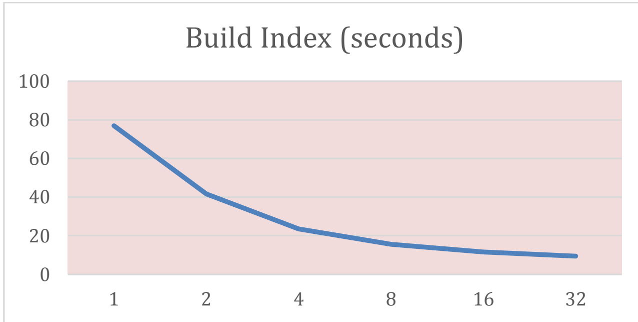
The query that was utilized is shown below (find all elements from the "Energy" dataset whose value exceeds a user-defined value (*in this example, we utilized 1.9 as the comparison value*).

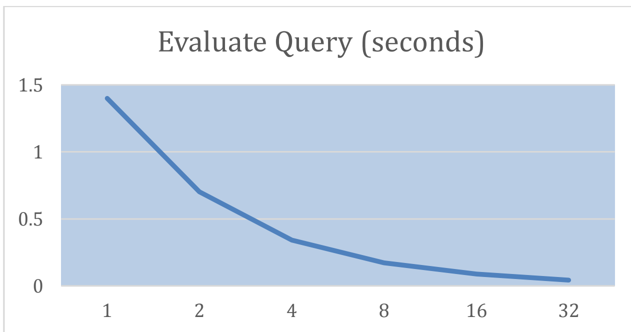
```
/* Create query */
static hid_t
test_query_1(float value1)
{
    hid_t q1 = H5I_BADID;
    hid_t q2 = H5I_BADID;
    hid_t q3 = H5I_BADID;

    /* Create and combine a bunch of queries
     * Query is: (x > value1)
     */
    if ((q1 = H5Qcreate(H5Q_TYPE_DATA_ELEM, H5Q_MATCH_GREATER_THAN, H5T_NATIVE_FLOAT, &value1)) < 0) FAIL_STACK_ERROR;

    /* Select object */
    if ((q2 = H5Qcreate(H5Q_TYPE_LINK_NAME, H5Q_MATCH_EQUAL, link_name)) < 0) FAIL_STACK_ERROR;
    if ((q3 = H5Qcombine(q2, H5Q_COMBINE_AND, q1)) < 0) FAIL_STACK_ERROR;
    return q3;


error:
    H5E_BEGIN_TRY {
        if (q1 != H5I_BADID)
            H5Qclose(q1);
        if (q2 != H5I_BADID)
            H5Qclose(q2);
        if (q3 != H5I_BADID)
            H5Qclose(q3);
    } H5E_END_TRY;
    return -1;
}
```

We identified two(2) specific data points of interest to benchmark:

1. The time to build the plugin (FastBit) index; and
2. The time to evaluate the user query and produce results.

The following graphs capture the scaling performance over a small number of parallel cores. In this we measured the interesting data points on [1, 2, 4, 8, 16, and 32] cores.





2.5 NEXT STEPS

The current implementation is a prototype of many of the ideas presented here. To eventually meet the ECP and HDF5 productization goals, we need to consider several enhancements. One highly desirable feature to incorporate into the project would be the use of Parallel Virtual Datasets (VDS). This would allow the creation of indexing information as well as query view results (as necessary) into new HDF5 files which would leave the original raw data files unmolested. The current approach, as the reader might recall, is to store metadata information into anonymous datasets within the actual data-file. This approach is fine for the short term demonstrations, but might be unacceptable in situations where adding metadata to a file might interfere with the goal of maintaining some analysis provenance (based on file creation/modification time-stamps).

Replace this text [Block Text Style] with your own text. Call out appendices in the order they will appear (Appendix A).

3. ACKNOWLEDGMENTS [HEADING 1 STYLE]

Insert text [Block Text Style]. This research was supported by the Exascale Computing Project (ECP),…

Also acknowledge any other assistance provided by other staff, projects, institutions (e.g., HPC resources), programs, etc.

4. REFERENCES [HEADING 1 STYLE]

Insert references [Block Text Style].

ECP
EXASCALE
COMPUTING
PROJECT

APPENDIX A. INTERFACES

A1 - QUERY API

```
/* Function prototypes */

/*-------------------------------------------------------------------------
 * Function:    H5Qcreate
 *
 * Purpose: 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.
 *
 * Return:  Success:    The ID for a new query.
 *          Failure:    FAIL
 *
 *-------------------------------------------------------------------------
 */
hid_t  H5Qcreate(H5Q_type_t query_type, H5Q_match_op_t match_op, ...);



/*-------------------------------------------------------------------------
 * Function:    H5Qclose
 *
 * Purpose: The H5Qclose routine terminates access to a query object,
 * given by query_id.
 *
 * Return:  Non-negative on success/Negative on failure
 *
 *-------------------------------------------------------------------------
hid_t  H5Qcombine(hid_t query1_id, H5Q_combine_op_t combine_op, hid_t query2_id);




/*-------------------------------------------------------------------------
 * Function:    H5Qget_type
 *
 * Purpose: The H5Qget_type routine queries a query object,
 * given by query_id, for its type information, originally provided to
 * H5Qcreate or created after a call to H5Qcombine. Information is returned
 * through the query_type parameter. See H5Qcreate for a table listing the
 * complete set of values that may be returned for query_type.
 *
 * Return:  Non-negative on success/Negative on failure
 *
 *-------------------------------------------------------------------------
 */

herr_t H5Qget_type(hid_t query_id, H5Q_type_t *query_type);
```

```
/*-------------------------------------------------------------------------
 * Function:    H5Qget_match_op
 *
 * Purpose: The H5Qget_match_op routine queries a singleton query object,
 * given by query_id, for its op information, originally provided to
 * H5Qcreate. Match information is returned through the
 * match_op parameter. See H5Qcreate for a table listing the complete set of
 * values that may be returned for match_op.
 * It is an error to perform this call on a compound query object (one which
 * was created with H5Qcombine).
 *
 * Return:  Non-negative on success/Negative on failure
 *
 *-------------------------------------------------------------------------
 */

herr_t H5Qget_match_op(hid_t query_id, H5Q_match_op_t *match_op);
```

```
/*-------------------------------------------------------------------------
 * Function:    H5Qget_components
 *
 * Purpose: The H5Qget_components routine queries a compound query object,
 * given by query_id, for its component queries.  The component queries are
 * returned in sub_query1_id and sub_query2_id, both of which must be closed
 * with H5Qclose.
 * It is an error to apply H5Qget_components to a singleton query object (one
 * which was created with H5Qcreate).
 *
 * Return:  Non-negative on success/Negative on failure
 *
 *-------------------------------------------------------------------------
 */

herr_t H5Qget_components(hid_t query_id, hid_t *subquery1_id, hid_t *subquery2_id);
```

```
/*-------------------------------------------------------------------------
 * Function:    H5Qget_combine_op
 *
 * Purpose: The H5Qget_combine_op routine queries a query object, given by
 * query_id, for its operator type. The possible operator types returned are:
 *       - H5Q_SINGLETON
 *       - H5Q_COMBINE_AND
 *       - H5Q_COMBINE_OR
 * H5Q_COMBINE_AND and H5Q_COMBINE_OR are only returned for query objects
 * produced with H5Qcombine and H5Q_SINGLETON is returned for query objects
 * produced with H5Qcreate.
 *
 * Return:  Non-negative on success/Negative on failure
 *
 *-------------------------------------------------------------------------
 */

herr_t H5Qget_combine_op(hid_t query_id, H5Q_combine_op_t *op_type);
```

```
/*-------------------------------------------------------------------------
 * Function:    H5Qencode
 *
 * Purpose: Given a query ID, serialize the query into buf.
 *
 * Return:  Non-negative on success/Negative on failure
 *
 *-------------------------------------------------------------------------
 */

herr_t H5Qencode(hid_t query_id, void *buf, size_t *nalloc);
```

```
/*-------------------------------------------------------------------------
 * Function:    H5Q_encode
 *
 * Purpose: Private function for H5Qencode.
 *
 * Return:  Non-negative on success/Negative on failure
 *
 *-------------------------------------------------------------------------
 */

hid_t  H5Qdecode(const void *buf);
```

EXASCALE COMPUTING PROJECT

A2 - INDEX API

```
/* Function prototypes */

/*-------------------------------------------------------------------------
 * Function:    H5Xregister
 *
 * Purpose: This function registers new index classes.
 *
 * Return:  Non-negative on success/Negative on failure
 *
 *-------------------------------------------------------------------------
 */
herr_t  H5Xregister(const H5X_class_t *idx_class);



/*-------------------------------------------------------------------------
 * Function:    H5Xunregister
 *
 * Purpose: This function unregisters an index class.
 *
 * Return:  Non-negative on success/Negative on failure
 *
 *-------------------------------------------------------------------------
 */

herr_t  H5Xunregister(unsigned plugin_id);



/*-------------------------------------------------------------------------
 * Function:    H5Xcreate
 *
 * Purpose: Create a new index in a container.
 *
 * Return:  Non-negative on success/Negative on failure
 *
 *-------------------------------------------------------------------------
 */

herr_t  H5Xcreate(hid_t loc_id, unsigned plugin_id, hid_t xcpl_id);



/*-------------------------------------------------------------------------
 * Function:    H5Xremove
 *
 * Purpose: Remove an index from objects in a container.
 *
 * Return:  Non-negative on success/Negative on failure
 *
 *-------------------------------------------------------------------------
 */

herr_t  H5Xremove(hid_t loc_id, unsigned n /* Index n to be removed */);
```

```
/*-------------------------------------------------------------------------
 * Function:    H5Xget_count
 *
 * Purpose: Determine the number of index objects on an object.
 *
 * Return:  Non-negative on success/Negative on failure
 *
 *-------------------------------------------------------------------------
 */

herr_t  H5Xget_count(hid_t loc_id, hsize_t *idx_count);
```

```
/*-------------------------------------------------------------------------
 * Function:    H5Xget_size
 *
 * Purpose: Returns the amount of storage allocated for an index.
 *
 * Return:  Greater than or equal to zero on success/Zero on failure
 *
 *-------------------------------------------------------------------------
 */

hsize_t H5Xget_size(hid_t loc_id);
```

A2 - INDEX SUPPORT

```
/* 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;
```

```
/* 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;
```

Insert text [Block Text Style].