Date: **July 15, 2013**	**Milestone 4.2, 4.3, 4.4: HDF5 Design Document – Updated for Quarter 4 demo deliverables** **FOR EXTREME-SCALE COMPUTING RESEARCH AND DEVELOPMENT (FAST FORWARD) STORAGE AND I/O**

LLNS Subcontract No.	B599860
Subcontractor Name	Intel Federal LLC
Subcontractor Address	2200 Mission College Blvd. Santa Clara, CA 95052

Table of Contents

Revision History

Date	Revision	History	Author
Feb. 26, 2013	1.0		Quincey Koziol, The HDF Group
Feb. 27, 2013	2.0, 3.0		Quincey Koziol, Ruth Aydt, The HDF Group
Feb. 28, 2013	4.0		Quincey Koziol, Ruth Aydt, Mohamad Chaarawi, Jerome Soumagne, The HDF Group
Mar. 1, 2013	5.0		Quincey Koziol, Ruth Aydt, Mohamad Chaarawi, Jerome Soumagne, The HDF Group
Mar. 1, 2013	6.0		Quincey Koziol, Ruth Aydt, Mohamad Chaarawi, Jerome Soumagne, The HDF Group
Mar. 4, 2013	7.0	Delivered to DOE as part of Milestone 3.1	Quincey Koziol, Ruth Aydt, Mohamad Chaarawi, Jerome Soumagne, The HDF Group
Mar. 22, 2013	8.0	Small additions related to end-to-end data integrity and asynchronous operations based on feedback.	Quincey Koziol, Ruth Aydt, Mohamad Chaarawi, Jerome Soumagne, The HDF Group
May 27, 2013	9.0	Add sections for dynamic data structures (append property and operations, and map object)	Quincey Koziol, The HDF Group
May 30, 2013	10.0, 11.0, 12.0, 13.0, 14.0	Add data analysis extensions (query/view/index objects).	Quincey Koziol, The HDF Group
June 5, 2013	15.0, 16.0	Add section “Transactions, Container Versions, and Data Movement in the I/O Stack” and related man pages, replacing the less fully-developed Transactions section originally delivered as part of Revision 7.0. Highlight headings of sections added or substantially revised since Milestone 3.1. Deliver to DOE as part of Milestone 4.1.	Ruth Aydt, Quincey Koziol, The HDF Group
June 20, 2013	17.0	Add Event Queue objects and operations. Update description of Asynchronous operations to use Event Queues. Change all asynchronous operations to take	Mohamad Chaarawi, Ruth Aydt, Quincey Koziol, The HDF Group

		event queues instead of request pointers. Highlight headings of sections added or substantially revised since Milestone 4.1 and remove highlights that were in V16.0. Deliver to DOE as part of Milestones 4.2, 4.3, and 4.4.	
June 27, 2013	18.0	Clarify text based on feedback received for Version 16. Add Note to Data Analysis Extensions alerting reader that further updates will be made to response to DOE feedback on Version 17. Make available to public as part of Q4 accepted document.	Ruth Aydt, Quincey Koziol, The HDF Group
July 13, 2013	19.0	Clarify text based on DOE stakeholder feedback received for Version 18.	Quincey Koziol, The HDF Group
July 15, 2013	20.0	Revisions to text, based on internal dicussions. Distributed to DOE reviewers.	Quincey Koziol, Ruth Aydt, The HDF Group
July 15, 2013	21.0	Removed change tracking, added previous highlighting (from v18), and posted publicly	Quincey Koziol, The HDF Group

Introduction

This document describes the design of multiple additions to the HDF5 library and API, including asynchronous I/O, end-to-end data integrity, transactions, data layout properties, optimized append operations, a new Map object and data analysis extensions for indexing and querying HDF5 containers. All changes for these capabilities were combined into one document for easier tracking; furthermore, because many of the features affect the same HDF5 API routines, they are easier to understand in combination.

Definitions

ACG = Arbitrarily Connected Graph

AXE = Asynchronous Execution Engine

BB = Burst Buffer

CN = Compute Node

DAOS = Distributed Application Object Storage

IOD = I/O Dispatcher

ION = I/O Node

VOL = Virtual Object Layer

Changes from Solution Architecture

As we've continued discussions with the ACG team, we've determined that their needs don't necessarily include the addition of a pointer or other dynamic datatype to HDF5. Instead, they have indicated that optimized support for append operations (which target the ingest phase of graph creation) and adding a new Map object to HDF5's data model would have a greater utility to them. So, this document reflects that divergence from the Solution Architecture document.

Specification

New HDF5 Library Capabilities

New functionality added to the HDF5 library is listed below, with sections for each capability:

- Asynchronous I/O and Event Queue Objects
- End-to-End Data Integrity
- Transactions, Container Versions, and Data Movement in the I/O Stack

- Data Layout Properties
- Optimized Dataset Append/Sequence Operations
- Map Objects
- Data Analysis Extensions

Asynchronous I/O and Event Queue Objects

Support for asynchronous I/O in HDF5 will be implemented by:

1) Building a description of the asynchronous operation
2) Shipping that description from the CN to the ION for execution
3) Generating a request object and inserting it into an event queue object that the application provides, while the operation completes on the ION

As originally designed, all asynchronous operations returned a request object for every operation that the application used to test/wait on. Completing every request through a call to test or wait was necessary or resource leaks would occur. This, along with tracking all of the request objects became very cumbersome in scenarios with large number of asynchronous operations. To address these issues, in Quarter 4 we added a new type of object to HDF5 called an Event Queue. This object will be passed as a parameter in all the newly added asynchronous routines instead of the request object that was used in Quarter 3.

An Event Queue provides an organizing structure for managing and monitoring the status of functions that have been called asynchronously. The name "Event Queue" is misleading (and will likely be changed), as the association of a request object for an asynchronous function with a given Event Queue has nothing to do with the order in which the function (the event) will happen. The Event Queues merely organize the IDs that are needed to track the status of the asynchronous functions, and except for the ability to "pop" the request object for the last function added to the queue, they are more like bags than queues.

Once an Event Queue is created, its identifier can be passed to other HDF5 APIs that will be run asynchronously. The request object associated with an asynchronous call will be pushed onto the Event Queue whose identifier was passed as a parameter to the function. The application can pop the last request object off of an Event Queue and monitor its completion status individually via H5AOwait and H5AOtest. The application can also wait or test the status of all the request objects in a given Event Queue. Note that when a request object is popped, it is no longer in the Event Queue and will not impact the status of the overall queue. The ability to check the error status of the individual request objects that are in given Event Queue in a user-friendly manner is a design/implementation goal for future quarters.

The application is free to continue with other actions while an asynchronous operation executes. The application may test or wait for an asynchronous operation's completion with calls to HDF5 API routines. All parameters passed to asynchronous operations are copied into the HDF5 library and may be deallocated or reused, except for the buffers containing data elements. The application must not deallocate or modify data element buffers used in asynchronous operations until the asynchronous operation has completed.

In addition, for reads, the data element buffers should not be examined until the asynchronous read operation has completed.

The HDF5 library tracks asynchronous operations to determine dependencies between operations. Dependencies exist between operations when a later operation requires information from an unfinished earlier operation in order to proceed. A simple "progress engine" within the HDF5 library updates the state of asynchronous operations when the library is called from the application. There is *no* use of background threads on CNs, only on the IONs, eliminating the possibility of "jitter" from background operations on CNs interfering with application computation and communication.

Dependencies between operations are captured at the HDF5 IOD VOL client and shipped with every operation to the HDF5 IOD VOL servers on the IONs. At the server, the operations are inserted into the AXE, taking into account the dependencies that they have been shipped with. The AXE makes sure that child operations are scheduled only after their parent operations have completed. While this approach allows completely asynchronous behavior at the client (HDF5 operations return immediately regardless of dependencies between each other), there are still few scenarios that retain the asynchronous behavior that was described in Quarter 3, where the dependent operation may be delayed at the client waiting for the parent operation to complete.

This behavior is a consequence of not using background threads on the CNs and not having a complex progress engine.

To demonstrate the behaviour of different asynchronous execution scenarios we give two examples.

First, consider an application that asynchronously creates an attribute then asynchrously writes data elements to the new attribute. Both calls are asynchronous and return immediately to the application. In the write call, the IOD VOL plugin detects a dependency on the attribute create call and ships the dependency to the server. At the server, the write operation is delayed until the attribute create operation completes.

Next, consider an application that asynchronously opens an attribute then asynchronously writes data elements to the attribute. In this example, the data write operation may be delayed inside the HDF5 library until the attribute open operation completes. The reason for this delay is that the write operation at the client requires the dataspace of the attribute that is being opened before it can ship the write operation to the server. This metadata is available in the first scenario, in the case of attribute create, because the create call provides this metadata about the attribute. In contrast, for the open call the metadata needs to be pulled from the server.

Asynchronous invocations of HDF5 routines that create or open an HDF5 object will return a "future" object ID[1] when they succeed. Future object IDs (currently referred to as placeholder object IDs in the User's Guide) can be used in all HDF5 API calls, with the HDF5 library tracking dependencies created as a result. If the asynchronous operation completes successfully, a future object ID will transparently transition to a normal object ID and will no longer generate asynchronous dependencies. If the asynchronous operation fails, the future object ID issued for the operation (and any future object IDs

[1] For other uses of "future variables", see e.g. http://blog.interlinked.org/programming/rfuture.html

that depend on it) will be invalidated and not be accepted in further HDF5 API calls. If a future object ID is invalidated, all asynchronous operations that depend on it will fail.

See below, in the API and Protocol Additions and Changes section, for details on how existing HDF5 API routines are extended, details on new H5EQ* API routines to create and operate on event queue objects along with new H5AO* API routines to test and wait on asynchronous request objects.

End-to-End Data Integrity

When enabled by the application, end-to-end data integrity is guaranteed by performing a checksum operation on all application data before it leaves a CN. The checksum for the information (both data elements and metadata information, such as object names, etc.) in each HDF5 operation will be passed along with the information to the underlying IOD layer, which will store the checksum in addition to the information. Checksum information is stored in the container for both data elements and the metadata (such as creation properties, the group hierarchy, attributes, etc.)

The HDF5 library will checksum application data before sending it from the CN to the ION for storage in the HDF5 container. In addition, the HDF5 library can optionally perform a checksum of the application data that it must copy into internal buffers within the library; whenever possible, data is written directly from the application's buffers and this copy is avoided. When data is read from the container, the IOD layer will provide a checksum with the data, which will be verified by the HDF5 library before returning the data to the application. If the checksum of the data read doesn't match the checksum from IOD, the HDF5 library will issue an error by default, but will also provide a way for the application to override this behavior and retrieve data even in the presence of checksum errors.

See below, in the API and Protocol Additions and Changes section, for details on new API routines to set properties for controlling the optional checksum behaviors.

Transactions, Container Versions, and Data Movement in the I/O Stack

The application is given almost complete control over managing data movement in the Exascale FastForward I/O stack. The HDF5 library, building on the capabilities of IOD and DAOS, provides the application with the ability to coordinate data movement between the application's memory on the CNs, the BBs on the IONs managed by IOD, and the storage managed by DAOS.

In this section, we introduce and discuss transactions, container versions, container snapshots, evicting data from the BB to DAOS storage, pre-fetching data from DAOS storage into the BB, reading of data from the BB or DAOS storage into the application memory, and replicating or rearranging data on the BBs for optimized performance. HDF5 Transaction Extensions are the Q4 deliverable, but the other topics are inter-related and therefore introduced as well.

Some of the design, especially as it relates to BB memory management and the specification of layout optimization hints, remains under active development. Open issues are noted, and discussions within the architecture team are ongoing. Readers are encouraged to consult the IOD and DAOS design documents for details on those levels of the stack.

A high-level diagram of the components of the Exascale FastForward I/O stack and the data movement that takes place under the control of the application is shown in Figure

1.[2] Although not referenced explicitly in the text that follows, the diagram may provide a useful visual model of the concepts that are introduced and discussed in this section.

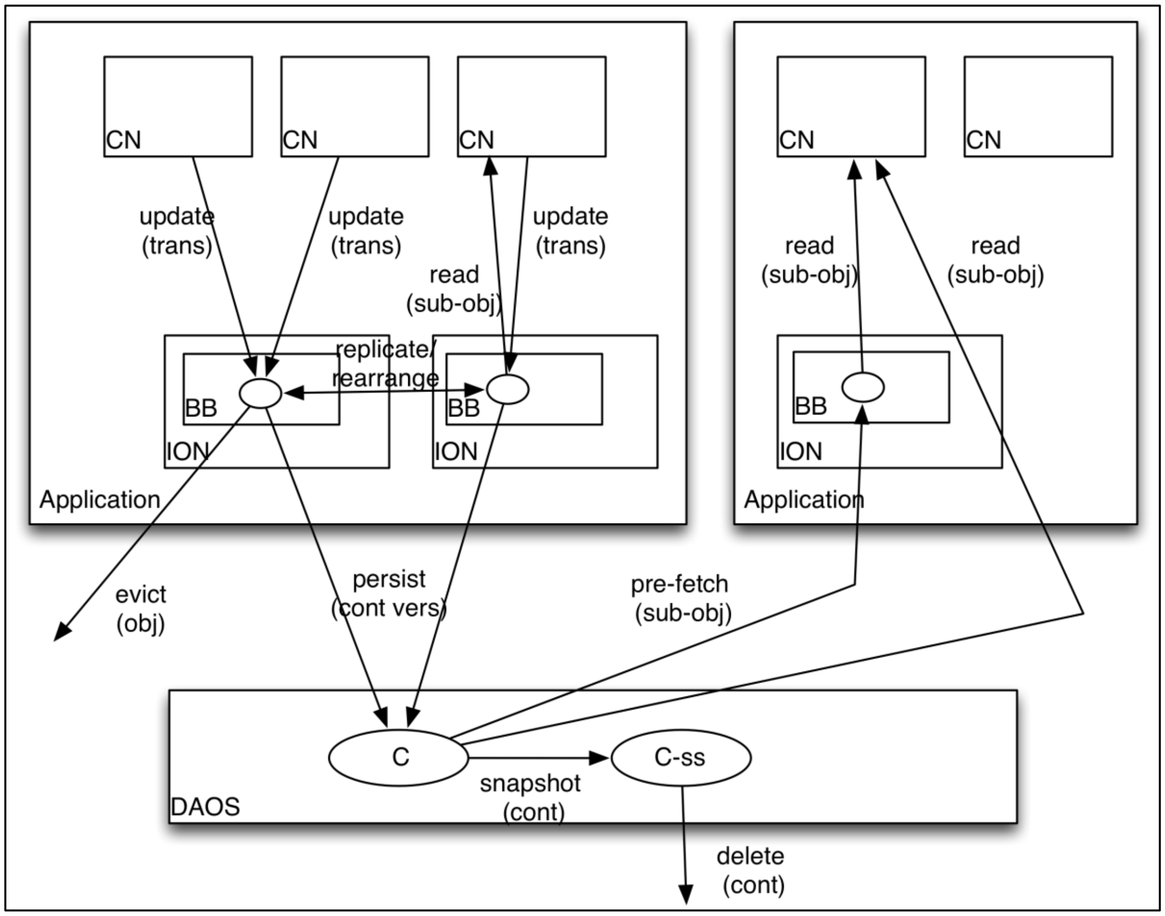


Figure 1: Data movement in the Exascale Fast Forward stack controlled by application requests.

Transactions and Writing to HDF5 Files (Containers)

The HDF5 library, building on the capabilities of IOD and DAOS, will allow applications to atomically perform multiple update operations on an HDF5 container through the use of ***transactions***. The HDF5 VOL and the IOD VOL plugin handle the translation between the HDF5 transaction APIs called by the application and the IOD transaction APIs.

Exposing a constraint from the DAOS layer that is necessary to insure container consistency, only one application can have a container open for writing at any given time.

The basic sequence of transaction operations an application typically performs on a container that is open for writing is:

1) *start* transaction N
2) perform *updates* to container as part of transaction N
3) *finish* transaction N

[2] Also see Figure 6 in the IOD Design Document.

One or more processes in the application can participate in a transaction, and there may be multiple transactions in progress on a container at any given time.

Managing Transactions

There are two primary modes for managing transactions:

- For tightly-coupled applications, the application can have one process in a process group act as the transaction leader.

  In this mode, the transaction leader is responsible for starting transaction N via a call to *H5TRstart()*, notifying other processes participating in the transaction that they can make updates as part of transaction N, knowing when the participating processes are done making updates, and finishing the transaction via a call to *H5TRfinish().*

- For loosely-coupled applications, the I/O stack, and in particular IOD, can manage a transaction.

  In this mode, each process that will be participating in transaction N calls *H5TRstart()* with the transaction number (N) and the number of ranks that will be participating. Note that all the processes participating in transaction N must be aware of the total number of participants. When a given process finishes the updates it will be making as part of transaction N, it calls *H5TRfinish().* When "number of ranks" processes have called *HRTRfinish()* for transaction N, the transaction is finished.

The application must specify the transaction number (N) when a transaction is started[3], provide that transaction number to all HDF5 operations that occur within the transaction, and eventually finish (or *abort*) the transaction.

When the application creates a container for the first time using H5TRcreate, it knows the first transaction number available to it is 1. When the application opens a previously created container, it can query to find out the next available transaction number and use that to seed the transaction numbers it assigns. As of June 2013, the HDF5 interface for this query has not yet been defined. The HDF5 routine will be a simplified interface that allows access to the more complicated *iod_container_query_tids* function defined in section 4.5.2 of the IOD Design Document or a similar function. For example, the HDF5 API might be "H5Fget_next_unused_transaction". The application is not restricted to using transaction numbers in strictly increasing order, but must never reuse transaction numbers. The section "*Finishing and Committing Transactions"*, below, provides additional information on transaction number requirements.

Container Updates during a Transaction

The container updates that occur within a given transaction can include adding or deleting H5Datasets, H5CommitedDataTypes, H5Groups, H5Links, and H5Attributes. The updates can also change the contents of existing H5Objects. The updates performed

[3] IOD allows the application to ask for the "next transaction number" under some circumstances, but that is currently not supported by HDF5. See the man page for H5TRstart for more details.

by HDF5 operations on H5Objects are reflected in updates to IOD objects. An update to one H5Object can result in updates to multiple IOD objects.

Finishing and Committing Transactions

Transactions can be finished in any order. The application is responsible for finishing a transaction (by calling *H5TRfinish*) when it is done making updates for the transaction. Finished transaction N will be *committed* (become readable) when all lower-numbered transactions are finished, aborted (via *H5TRabort)*, or explicitly skipped (via *H5TRskip*).

Once a transaction number has been used to start a transaction, or has been explicitly skipped, it cannot be reused – even if the transaction is aborted. Transactions that are aborted or explicitly skipped are also considered committed when all lower-numbered transactions are finished, aborted, or explicitly skipped; commitment of an aborted or skipped transaction does not update the container contents.

The application does not explicitly commit a transaction, but it indirectly controls when a transaction is committed through its assignment of transaction numbers in "start transaction" calls and the order in which transactions are finished, aborted, or explicitly skipped. Our current thinking is that the default behavior will be to have asynchronous *H5TRfinish* operations complete when the transaction is committed.

Container Versions

When a transaction is committed, the state of the container is changed atomically. The data for a committed transaction is managed by IOD and, when IONs are present, resides in the Burst Buffers. The *version* of the container after transaction N has been committed is N. A reader of this version of the container will see the results from all committed transactions up through and including N.

Note that container version N may not have resulted from N finished transactions on the container; there is no guarantee that some transactions were not aborted or explicitly skipped. Since aborted and skipped transactions are also committed, they advance the container version even though they do not change the contents of the container.

Terminology Differences across the Stack

There has been considerable discussion within the team about various naming and numbering conventions related to transactions, and there remain some discrepancies in terminology across the various layers of the stack. We mention them here to help the reader as they review and integrate the HDF5, IOD, and DAOS design documents.

At the HDF5 layer, transactions and transaction numbers are used to refer to actions related to atomic updates of the container and the changes associated with those actions, while container versions are used to refer to the state of the container. Therefore, read operations are performed on versions of containers, not on transaction numbers.

For example, in transaction 99 the application sets the element located at location [3,19] to 74 in H5Dataset /G1/A and creates a new H5Group with the path /G2. At the IOD layer, the changes related to this transaction update the Array object associated with /G1/A and the KV object that holds the H5Links for the root group (to point to the new /G2 H5Group), and a new KV object to hold the H5Links for the new group, /G2, is also created. After transaction 99 is committed, the value of A[3,19] in container version 99 will be 74 and /G2 will exist in the container. Values for other elements in H5Dataset A and many other objects in the file are also visible in container version 99 (assuming lower-numbered transactions made updates to other elements in A and added other objects to the container).

IOD does not distinguish between transaction numbers and container versions – it describes things strictly in terms of transactions and transaction ids.   DAOS has "epochs" instead of transactions.

Persisting and Accessing Container Versions

The application can *persist* a container version, N, causing the data (and metadata) for the container contents that are in IOD to be copied to DAOS. When container version N is persisted, the data for all lower-numbered container versions (committed transactions on the container) that have not yet been persisted is also flattened[4] and copied to DAOS. Data (and metadata) for persisted container versions is not automatically removed from IOD. The application must explicitly *evict* data from IOD – this is discussed in more detail in a later section on Burst Buffer Space Management.

After container version N is persisted (assuming no higher-numbered versions have yet been persisted), DAOS holds version N of the container.   DAOS refers to this version as the Highest Committed Epoch (HCE). IOD refers to it as durable.

The Exascale Fast Forward stack does not support unlimited "time travel" to every container version, as versions may be automatically flattened for efficiency when they are persisted. For example, say the HCE on DAOS is 19, the application finishes transactions 20, 21, 22, and those transactions become committed (readable) on IOD.   The application then asks that container version 22 be persisted.   The HCE on DAOS becomes 22, and container versions 19, 20, and 21 are flattened and not individually accessible from DAOS. However, as discussed below, if a read handle is open for a given container version, that version is guaranteed not to be flattened until the read handle is closed. The IOD Design Document covers these concepts in greater detail.

Note that an application is not required to persist any versions of a container. For example, an application that is utilizing the Burst Buffer for out-of-core storage may never persist the data to DAOS.

Making a Snapshot of a Container on DAOS

The application can request a *snapshot* of the highest container version that has been persisted to DAOS.   This makes a permanent entry in the namespace (using a name supplied by the application) that can be used to access that version of the container. The snapshot is created with version ID = [0 or the container version number] [5] and is independent of further changes to the original container. The snapshot container behaves like any other container from this point forward. It can be opened for write and updated via the transaction mechanism (without affecting the contents of the original container), it can be read, and it can be deleted.

[4] Only *valid* data for lower-number container versions is copied. Any data which has been overwritten in later transactions lower than N will *not* be copied.

[5] Is there a preference regarding which number (0 or HCE) is used?

General Discussion

The application has complete control over when container versions are persisted to DAOS and when snapshots are taken. That said, we expect that snapshots will be taken infrequently, persists will encompass multiple committed transactions, and transactions will contain several to many operations.   The prototype implementation will offer the opportunity to assess the frequencies that can be supported with good performance.

Transactions provide the benefit of ensuring logically-consistent container versions. In addition, they provide a mechanism for detecting and recovering from errors, as transactions can be aborted and their updates retried.   In the prototype Exascale FastForward Stack, the DAOS layer is the primary focus of error reporting and recovery. The IOD, VOL, and HDF5 layers will detect and report errors, but will not be designed to recover from them. Ultimately, the application will also need to be involved in the handling of failures that cannot be self-healed by the lower layers.

New HDF5 API routines (see below) will allow the application to start, abort, finish, and skip transactions, and to persist and snapshot container versions. Routines will also be added to inquire about transaction and container status. Existing API routines will be extended to accept transaction numbers, indicating which transaction a given operation is part of.

Design Decisions

Because we allow transactions to be started and finished out of order, and because the application can pipeline multiple transactions, there are situations where operations in later transactions may depend on the actions of earlier transactions that have not yet been committed.   For example, in Transaction 1 the application creates H5Group /A and in Transaction 7 the application creates H5Dataset /A/B. Using asynchronous calls and allowing multiple transactions to be in flight at once, there is no guarantee that the transaction containing /A's creation will have been committed at the time Transaction 7 tries to create /A/B. Even if Transaction 1 was committed, it is possible that one of the operations in Transactions 2-6 could have deleted /A.

Four possible solutions (at least) present themselves for addressing this issue:

A pessimistic (but guaranteed safe) implementation would require that the container be at Version 6 (i.e. transactions 1-6 have committed) before asynchronous operations in Transaction 7 can complete. This allows the HDF5 library to verify the state of the container before completing updates in Transaction 7.

An optimistic implementation would assume the application knows what it is doing, and that it will only update objects that it knows have been created, or that it creates in the same transaction. In the above example, the application should make sure Transaction 1 has committed, and know that it did not delete /A in Transactions 2-6, before trying to create /A/B in Transaction 7.

An implementation could speculatively execute HDF5 operations by maintaining a log of updates within a transaction and replay that log after lower-numbered transactions are committed. This has the benefit of immediate execution and eventual guaranteed correctness, but comes with the drawback of additional complexity and duplicated I/O.

Finally, an implementation could maintain a distributed cache that tracked the state of the container metadata and captured the application's view during all the outstanding transactions. The distributed metadata cache would be used to predict the correctness of operations during a transaction, allowing an application to proceed asynchronously and safely. However, the complexity and expected poor performance of such a cache likely outweigh any correctness benefits it might have.

We have decided to adopt the optimistic approach for this phase of the project. The most complicated dependencies have to do with object creation and metadata management, but we believe that few applications require complex dependencies and can manage simple ones well. The more likely case is that an H5Dataset will be created early in the application, then later multiple ranks will update separate elements in the H5Dataset in independent transactions that are in-flight simultaneously.

Support for Legacy Applications

There is a desire to support legacy library and application code that make HDF5 calls without specifying transactions or asynchronous request pointers. While not optimized for performance, legacy HDF5 API calls will execute correctly, albeit synchronously.

Handling the lack of transaction numbers in legacy API calls is a bit more complicated, because the legacy code could co-exist with new code that does assign and manage transaction numbers. Rather than insisting each legacy HDF5 API call be updated to include a transaction number, we will provide two special API calls that can be used to package legacy operations into a transaction. The exact API signatures are not yet specified, but in general terms the "start_legacy_transaction" will be called with a transaction number that will be assigned to all legacy HDF5 calls executed prior to the "end_legacy_transaction". The application will be responsible for managing other transaction numbers, keeping in mind the transaction number(s) assigned to the legacy operations packaged by the new start/end calls.

exp the legacy application must use the start/end legacy transaction brackets even if never uses the FF APIs. While we could conceivably manage the transactions for the legacy applications, neither of the two possible behaviors "put everything in one transaction" or "put each operation in its own transaction" are very attractive. The first would likely be bad from a fault-tolerance and IOD/DAOS management perspective, and the second would likely be awful from a performance perspective. Since transactions and data migration are so key to the Fast Forward stack, allowing applications to run (poorly) with no attention to these details does not seem wise. As we gain experience with the stack, we may see ways to offer intelligent automation that are currently not obvious to us.

Reading from HDF5 Files (Containers)

Applications perform reads on HDF5 Files (containers) in the EFF stack in almost the same way they perform them on an HDF5 File stored in the Binary HDF5 format. The difference is that when an H5File is open for read in EFF, not only is the file (container) name specified, but also the container version.

Once an application has obtained a read handle for a container version, it is guaranteed to see the contents of the container at that version until the container is closed (and the read handle released), even if subsequent transactions are committed to the container.

If a container is already opened by other processes that run on the same IOD instance, a new reader can share data in the BB with those processes (even when the reader and the other processes are not accessing the same version of the container. Note that the container versions that are available in the BB on one set of IONs may be different that the container versions that are available directly from DAOS due to flattening that can occur when a container version is persisted.

The application can issue explicit prefetch commands to move data from DAOS to the BB. When the data being read is not already in the BB (as the result of an earlier write or prefetch) it will be read from DAOS. Data that is read from DAOS will go through the IONs to the CNs, but will not be cached in the BBs unless explicitly requested. We are considering adding a hint to the read APIs that direct the data be cached in addition to being read. In addition, we are considering allowing multiple read requests to be tagged with a batch identifier, indicating that all of the data in the batch should be read together.

Pseudocode showing the sequence of HDF5 operations that occur when a container is opened for read follows. The exact API signatures remain to be specified.

```
/* Open highest container version for reading;
 * This results in a read handle being opened on that version so it
 * will not be flattened or evicted. */
file_id = H5Fopen_ff( "myContainer", H5F_ACC_RDONLY, … );


/* If the highest container version is not the one desired,
 * it is still required to get information about what other versions
 * exist. Find out how many there are and the version numbers */
n = H5Fversion_count( file_id );
err = H5Fgetversions( file_id, n, &ver[n] );


/* Now get a read handle for the desired version; this might fail
 * if that version was flattened or evicted between the previous call
 * and this one.  We'll assume it works */
file_id2 = H5Fopen_ff( "myContainer, H5F_ACC_RDONLY, Ver=ver[3], …);


/* Close file_id to release read-handle on that version */
H5Fclose_ff( file_id );

/* Do lots of HDF5 Group / Attribute / Dataset read operations using
 * file_id2 */
…


/* Close file_id2, releasing the read handle on that version */
H5Fclose_ff( file_id2 );
```

Processes that perform reads on an HDF5 file identifier that is open for write (one returned by an *H5Fcreate_ff* or *H5Fopen_ff* with `flags=H5F_ACC_RDWR)` will always see the results of the highest container version. That version may change between subsequent reads because transactions are presumably being committed to the container. Processes that want a stable view of the container data should re-open the container read-only, and use the read handle to access the container.

Burst Buffer Space Management

IOD is responsible for moving data into and out of the BBs when directed to do so by higher-layers in the stack (HDF5, as directed by the application). Because the BB is managed manually, the application must explicitly request eviction and residence, effectively controlling the working set in the BBs.

Moving Data into the Burst Buffers.

As discussed previously, container updates are performed in transactions and result in writes to the BBs. (See the IOD design for more details on how updates are made to various types of IOD objects in the container). All objects in the BB resulting from updates performed in transactions have an associated container and transaction number (for transactions that are not yet committed) or container version (for committed transactions). When a container version is persisted, associated data in the BB is copied to DAOS and the copied data remains in the BB until explicitly evicted.

The application can also request that data be pre-fetched from DAOS into the BBs. The details on how these requests will be made have not yet been fully designed. We anticipate having application processes on the CNs issue pre-fetch requests for a given set of HDF5 Objects or sub-objects (such as a subset of elements in an H5Dataset) using calls that are similar to standard HDF5 read requests in terms of how the objects and sub-objects are specified (for example, through the use of hyperslab selections).

Pre-fetch requests will be made for a specific container version, because the container version will be specified as part of the container open and associated with subsequent operations by virtue of the file handle used to access the HDF5 objects in the container. We are considering allowing multiple pre-fetch requests to be tagged with a batch identifier, indicating that all of the data in the batch should be pre-fetched together. A request from a given CN will be directed to its associated ION, and the data to fill the CNs request will go to the BB on that ION. Hints may also allow further layout optimizations to be specified.

Recall that read requests do not result in updates to the BB. Read requests that can be satisfied by data already in the BB will be; other read requests will move data directly from DAOS to the CN via the ION, but without writing to the BB.

Evicting Data from the Burst Buffers

Putting the application in charge of evicting data from the burst buffers implies that the application must have an interface to manage data in the burst buffer in units that it understands. This is challenging for a number of reasons, perhaps foremost of which is the application deals with HDF5 Objects (and sub-objects), which do not map one-to-one to the IOD objects (and sub-objects) that IOD uses to track BB contents. While the VOL and IOD-VOL plugin can provide some assistance, they are not intimately aware of the “pieces” in the IOD logs that go into making up a particular container version; perhaps this awareness is not required in order for the VOL layer to help, but at this point it remains a concern.

Discussions continue within the EFF team about how to design the evict interface and implementation. Some relevant points and open issues are listed here:

- Objects (and sub-objects) in the BB are associated with a given container and container version. For this reason, an eviction operation should include a [container, version, object] triplet to fully specify the data to be evicted.

- BB data resulting from transactions may be in log-structured format, while BB data resulting from pre-fetches may be in a flattened layout.

- It is impractical to allow eviction of partial-objects from the BB because of implementation details at the IOD level. This means that eviction is not the "mirror image" of pre-fetch, which can specify sub-objects. Note that sub-objects may reside in the BB (for example, as the result of a pre-fetch), but the evict can only be specified on a [container, version, object] granularity.

- IOD's ability to do semantic resharding and multi-format replicas can result in multiple copies of "the same" [container, version, object] data in the BB. How will the application specify which copy to evict?

- Attempts to evict a [container, version, *] that has an open read handle will fail, because of the guarantee that the reader will be able to see a consistent view of the container as long as the read handle is open. If the writer has the designated 'clean up' responsibility, how can it do that if a reader has the objects open? Can evictions be queued until a read handle is released?

- When there are multiple users of the data, can any user evict it (assuming there is no open read handle)? Is there a need to reference-count users?

- How can the application evict objects it believes it is done with, without clearing oft-accessed metadata objects (such as the root group KV store) from the BB?

- What happens to data from aborted transactions? Is it automatically evicted or left around for possibly recovery in which case the application must evict?

- We will likely want to provide APIs that allow "sensible evictions" on a group of objects with a single command. For example, if an H5Group is evicted, all of the children of that group (from the same container version) are evicted.

Layout Optimizations

IOD offers a number of optimizations including semantic resharding and multi-format replicas. The mechanisms for exposing these capabilities to the application via HDF5 APIs have not yet been designed. Open questions include how to specify one of the replicas (versus another) be read or evicted, and how to allow the application to make optimizations without intimate knowledge of the underlying storage architecture.

It may also be beneficial to allow multiple container versions to be flattened on IOD (BB) without having to persist the data to DAOS and pre-fetch it. Out-of-core applications, for example, might benefit from this capability.

We continue to work as a team to address open questions in this area. Note that the initial design of these optimizations was part of the IOD Q4 deliverable (this quarter), so the concepts are still fairly new.

Data Layout Properties

Data layout properties, and other aspects of HDF5, IOD and DAOS software stack behavior, will be controlled by properties in HDF5 property lists (e.g. file creation, object creation, object access, etc.). New properties are set and retrieved by HDF5 API routines described below, in the API and Protocol Additions and Changes section. Existing HDF5 properties will be translated to appropriate actions on the container, e.g. the contiguous and chunked storage properties for datasets in native HDF5 containers will be used by the IOD layer to control analogous storage settings in IOD and DAOS containers. The set of behaviors controlled by properties is still under active development; more properties (and API routines to control them) will be added over the course of the project.

In support of ACG applications' data ingest operations, as well as data gathering applications that record instrument measurements, we have added an optional data layout property to indicate that all write operations to a dataset will be append-only, with no random I/O of elements in the middle of a dataset, and no overwrites of existing elements. This will allow the HDF5 library to store data elements for the dataset in a more optimized fashion.

See below, in the API and Protocol Additions and Changes section, for details on the new H5Pset_write_mode() routine to set this data layout property.

Optimized Dataset Append/Sequence Operations

To support ACG ingest operations and other applications that rapidly append data to an HDF5 dataset (as well as applications that sequence through those datasets in a similar fashion), we are extending the HDF5 API with routines that allow those operations to be performed in an optimized and easy to use manner. In addition, to flesh out the HDF5 API with calls that ACG applications will frequently use, we are adding simple routines for quickly setting and retrieving single elements in an HDF5 dataset.

See below, in the API and Protocol Additions and Changes section, for details on the new H5DO* API routines for optimized sequential reads and writes.

Map Objects

ACG applications have a great deal of data that doesn't correspond well to the current HDF5 data model, showing a need for expanding that model. In particular, ACG data contains many vertices in each graph, each of which has a large amount of name/value pairs that are inefficient to store with HDF5 dataset objects. To address this need, we plan to add a new Map object to the HDF5 data model and API.

Map objects in HDF5 will be similar to a typical "map" data structure in computer science. HDF5 maps will set/get a value in the object, according to the key value provided, with a 1-1 mapping of keys to values. All keys for each map object must be of the same HDF5 datatype, and all values must also be of the same HDF5 datatype (although the key and value datatypes may be different). Like HDF5 datasets, HDF5 maps will be leaf objects in the group hierarchy within a container, and, like other HDF5 objects in the container, can have attributes attached to the map object.

Many extensions beyond a straightforward map data structure were considered, such as support for multiple values for each key (i.e. a "multi-map"), allowing different datatypes for each key and/or value, etc. However, the current capabilities meet the needs for ACG use cases and allow us to explore further extensions to the map object's capabilities incrementally. We expect to add functionality to the map object over the course of the project, or in follow-on projects, as more application needs are exposed.

See below, in the API and Protocol Additions and Changes section, for details on new H5M* API routines to create and operate on map objects.

Data Analysis Extensions (Supporting Query and Index Operations)

Support for data analysis operations on HDF5 containers will be implemented by:

- New "query" object and API routines, enabling the construction of query requests for execution on HDF5 containers
- New "view" object 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

These extensions to the HDF5 API and data model 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.

Query Objects

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. 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 "equal to", "not equal to", "less than", etc. and a value for the match operator. H5Qcombine combines two query objects into a new query object, using Boolean operators such as AND and OR. Queries created with H5Qcombine can be used as input to further calls to H5Qcombine, creating more complex queries.

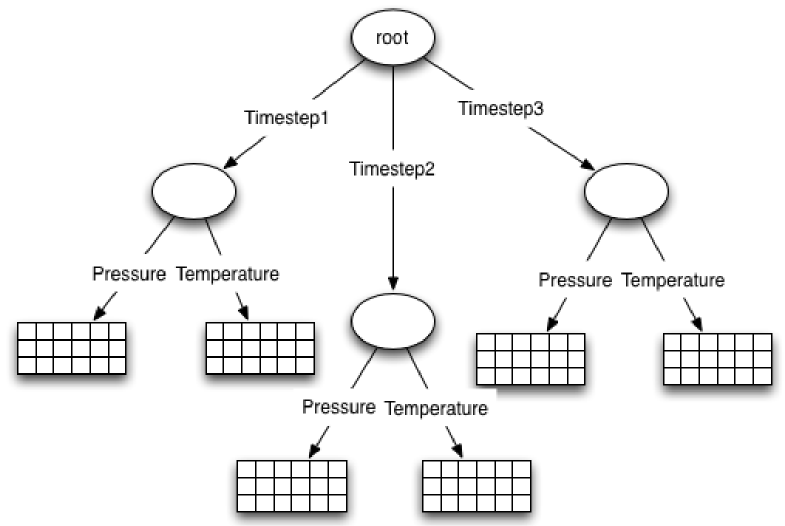
For example, a single call to H5Qcreate could create a query object that would match data elements in any dataset within the container that are equal to the value 17. Another call to H5Qcreate could create a query object that would match link names equal to "Pressure". Calling H5Qcombine with the AND 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 (or GUI) query language 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 the current project.

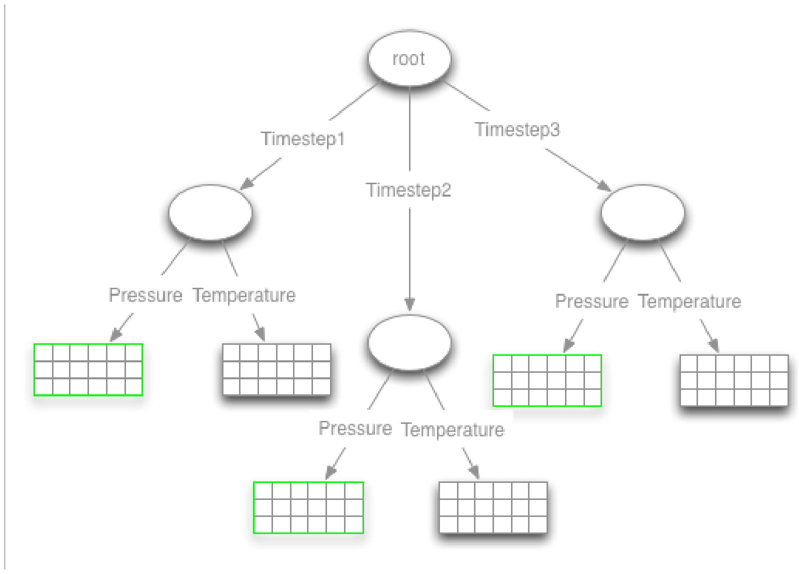
View Objects

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. 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 by a view can be retrieved by further API calls.

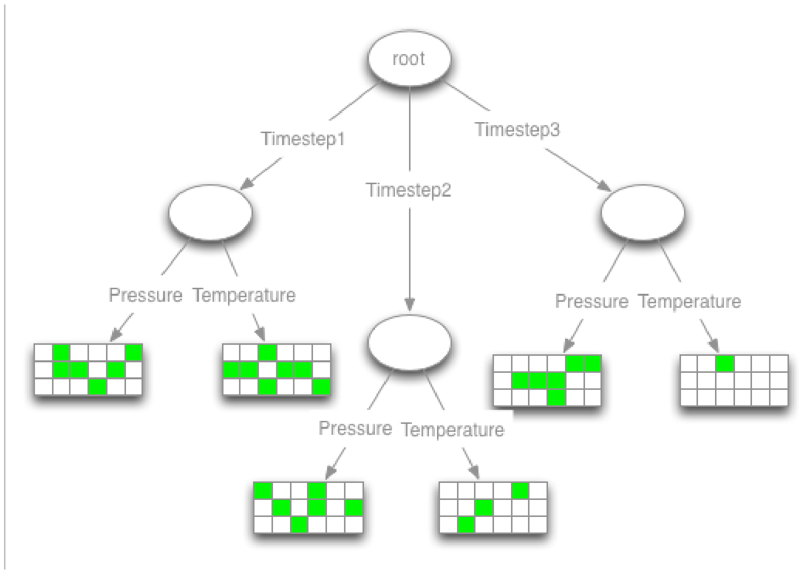
For example, starting with the HDF5 container described in the figure below:



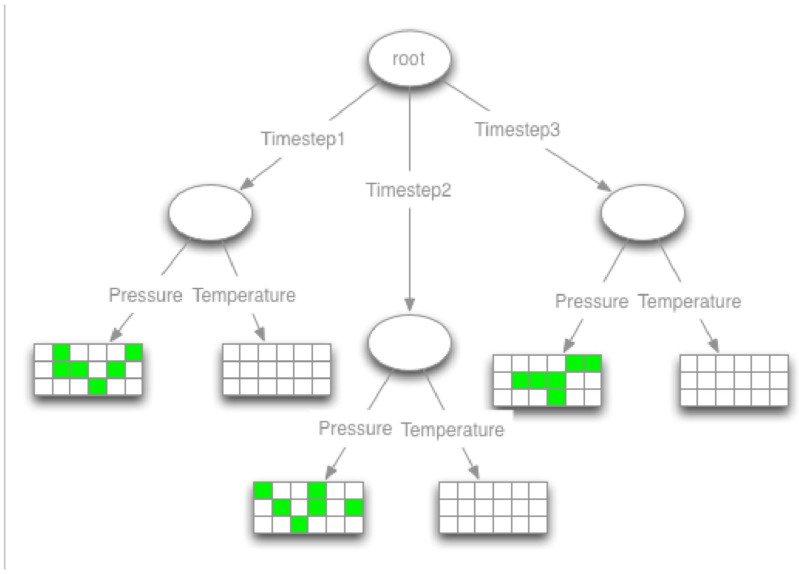
Applying the '<link name> = "Pressure"' query (described above) would result in the view shown below, with the underlying container greyed out and the view highlighted in green:



Alternatively, applying the '<data element> = 17' query (described above) would result in the view shown below, with the underlying container greyed out and the view highlighted in green:



Finally, applying the combined '<link name> = "Pressure" AND <data element> = 17' query (described above) would result in the view shown below, with the underlying container greyed out and the view highlighted in green:



Views can be thought of as containing a set of HDF5 references (object, dataset region or attribute[6] 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" AND <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.

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. Index objects are stored in the HDF5 container that they apply to, but are not visible in the container's group hierarchy. Instead, index objects are part of the metadata for the file itself. New index objects are created by passing a container to be indexed and index type to the H5Xcreate call (see the H5Xcreate API call description below for details).

For example, if the '<link name> = "Pressure" AND <data element> = 17' query (described above) was going to be frequently executed on the container in the figures above, indices could be created in that container which would speed up 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

[6] Attribute references are currently under development for delivery in Q6 of the FastForward project.

improve view creation for the more complex '<link name> = "Pressure" AND <data element> = 17' query.

To allow an application greater control over when the contents of an index are updated, indices must be explicitly updated by an application (with the H5Xupdate API call). HDF5 indices will not reflect modifications of the HDF5 container they apply to unless updated by an application. If an index is not up to date, it will not be used to assist in creating a view from a query (H5Xis_current can be used to query if an index is up to date).

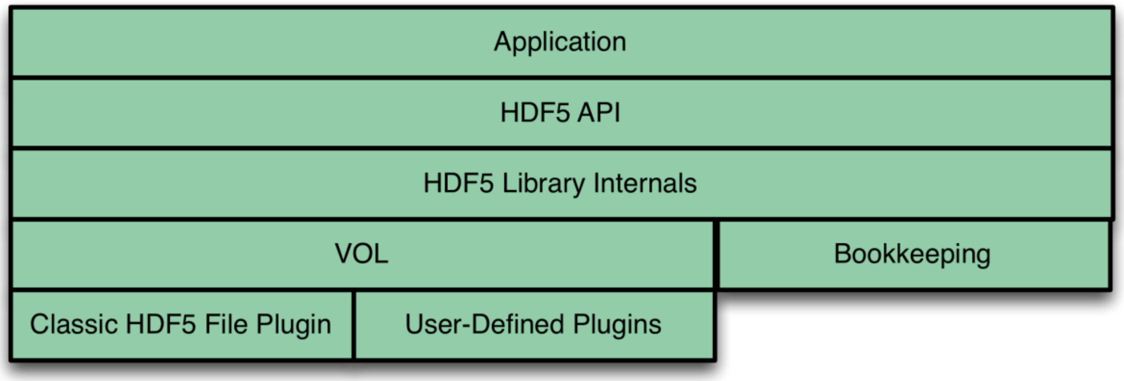
The HDF5 library will expose an interface for third-party indexing plugins, such as interfaces to FastBit[7], etc., which will be defined and demonstrated in quarters 6-8 of the project. This interface will provide indexing plugins with efficient access to creating and maintaining indices on the contents of the container, as well as allowing them to directly create private data structures within the container for storing the contents of the index.

See below, in the API and Protocol Additions and Changes section, for details on the new H5Q*, H5V* and H5X* API routines used for query, view and index operations, respectively.

Architectural Changes to the HDF5 library

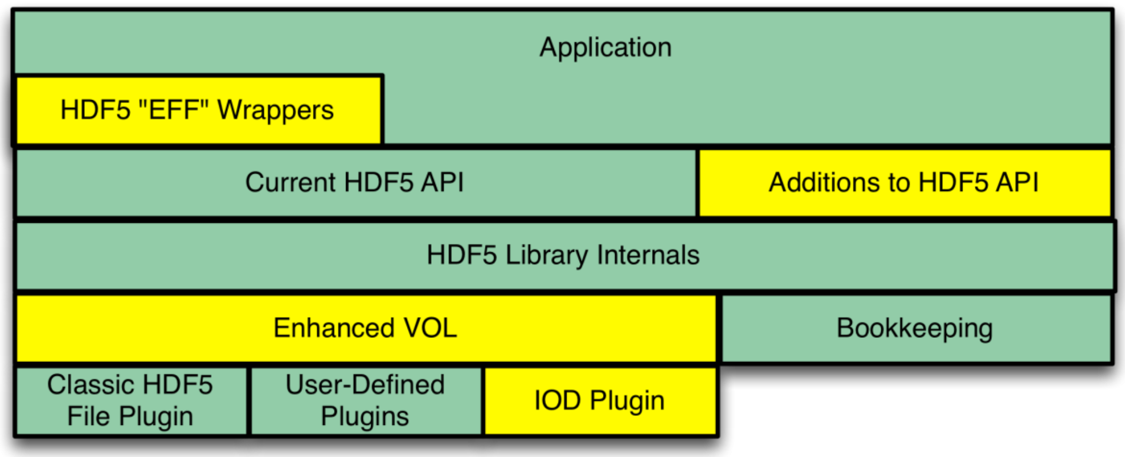
The architecture of the core HDF5 library is largely unaffected by the changes described in this document. The majority of the capabilities added to the HDF5 API are handled by a wrapper layer above the main HDF5 library, and a small number of additions to the main API routines (details of these API changes are described below in the API and Protocols Changes section). Adding transactions requires extending the VOL interface to incorporate some additional callbacks and/or parameters as well. Fortunately, the VOL is already designed to support asynchronous operations (although it is currently not used by any existing plugins), so few changes are required to support that capability.

The following diagram shows an overview of the HDF5 library architecture before the FastForward project capabilities are added:



The following diagram shows an overview of the HDF5 library architecture after the EFF capabilities are added, with the new or enhanced portions highlighted:

[7] https://sdm.lbl.gov/fastbit/



The majority of the implementation work is localized to the EFF wrapper routines and the IOD VOL plugin. In particular, the end-to-end integrity checksums are created and validated in the IOD plugin, and data layout information is translated from HDF5 properties to IOD hints there as well. Transactions and asynchronous operation information is encapsulated in HDF5 properties by the EFF wrapper routines and retrieved, interpreted and returned by the IOD plugin in the same way. Details of the IOD VOL plugin design are located in an accompanying document.

Storing HDF5 Objects in IOD Containers

Objects in the HDF5 data model and operations on them are mapped to IOD objects and operations, as they are handled by the IOD VOL plugin. See section 4.3.2 in the IOD design document for a description of the mapping from HDF5 objects to IOD objects and the accompanying IOD VOL plugin design document for a description of how those mappings are carried out.

API and Protocol Additions and Changes

There are two kinds of changes to the HDF5 library API: generic changes to existing API routines that accommodate new capabilities, such as asynchronous I/O and transactions, and additions to the HDF5 API which add new features. Both of these types of changes to the HDF5 API are described below.

Generic changes to HDF5 API routines

Many HDF5 API routines operate on HDF5 file objects and need to be extended in the same way. Rather than describing each of the modified HDF5 API routines, a generic modification is described below, along with a list of HDF5 API routines that are affected.

[Paragraph added for Milestone 4.2] This section was written before implementation had begun and gives a general overview of the API changes. Readers are encouraged to consult the HDF5 API reference manual pages in the latest version of the *User's Guide to FastForward Features in HDF5* for the most current information.

Existing HDF5 routines that operate on HDF5 file objects are extended by adding two new parameters: a transaction number and a pointer to an asynchronous operation request object. Additionally, HDF5 API routines that are extended in this manner have a suffix appended to the routine name, to distinguish these routines from existing routines. The following pseudo-function prototypes describe the method for these changes to HDF5 API routines:

Current routine:

```
<return type> H5Xexisting_routine(<current parameters>);
```

Extended routine:

```
<return type> H5Xexisting_routine_ff(<current parameters>,

    uint64_t transaction_number, hid_t event_queue_id);
```

In other words, each extended HDF5 API routine has a suffix of "_ff"[8] added to the API's routine name and two new parameters added to its parameter list: a transaction number, which indicates the transaction this operation is part of, and an identifier for the event queue object where the request for this operation is pushed, for testing/waiting on the asynchronous completion of the operation. Passing a NULL pointer for the request object pointer value indicates that an operation should complete synchronously.

As a concrete example, the following prototypes show the change to the group creation API routine for HDF5, H5Gcreate[9]:

Current routine:

```
hid_t H5Gcreate(hid_t loc_id, const char *name, hid_t lcpl_id,

    hid_t gcpl_id, hid_t gapl_id);
```

Extended routine:

```
hid_t H5Gcreate_ff(hid_t loc_id, const char *name, hid_t lcpl_id,

    hid_t gcpl_id, hid_t gapl_id, uint64_t transaction_number,

    hid_t event_queue_id);
```

Note that the error value returned when a routine is invoked asynchronously only indicates the status of the routine up to the point when it is scheduled for later completion. The asynchronous test and wait routines (below) return the error status for the "second half" of the routine's execution.

We anticipate that if the features from the FastForward project are productized in a future public release of HDF5, the "_ff" suffix will be removed and affected API routines will be versioned according to the standard convention for modifying HDF5 API routines[10].

[8] "ff" is short for "FastForward"

[9] http://www.hdfgroup.org/HDF5/doc/RM/RM_H5G.html#Group-Create2

[10] HDF5's API versioning conventions are described here: http://www.hdfgroup.org/HDF5/doc/RM/APICompatMacros.html

A note on the design of the API changes: We considered alternate forms of passing the transaction and request information into and out of the HDF5 API routines, such as using HDF5 properties in one of the property lists passed in to API routines to convey the information. Using HDF5 properties had a number of drawbacks however: (1) several of the API routines did not have property list parameters and so would have to be extended with more parameters anyway, (2) setting the additional information in properties can sometimes obscure the fact that an operation's behavior has been changed, and (3) it is particularly tedious for application developers to retrieve the asynchronous request object from a property list after each API call.

[Paragraph added for Milestone 4.2] The following is a list of all HDF5 API routines that were originally targeted for extension in the manner described above. Please consult the latest version of the *User's Guide to FastForward Features in HDF5* for the most recent information. The list below will likely be removed from this document in Quarter 5.

HDF5 Attribute Routines:

- H5Acreate
- H5Acreate_by_name
- H5Adelete
- H5Adelete_by_name
- H5Adelete_by_idx
- H5Aexists
- H5Aexists_by_name
- H5Aget_info_by_idx
- H5Aget_info_by_name
- H5Aget_name_by_idx
- H5Aiterate
- H5Aiterate_by_name
- H5Aopen
- H5Aopen_by_idx
- H5Aopen_by_name
- H5Aread
- H5Arename
- H5Arename_by_name
- H5Awrite

HDF5 Dataset Routines:

- H5Dcreate
- H5Dcreate_anon
- H5Dopen
- H5Dread
- H5Dset_extent
- H5Dwrite

HDF5 Group Routines:

- H5Gcreate
- H5Gcreate_anon
- H5Gget_info_by_idx
- H5Gget_info_by_name
- H5Gopen

HDF5 Link Routines:

- H5Lcopy
- H5Lcreate_external
- H5Lcreate_hard
- H5Lcreate_soft
- H5Lcreate_ud
- H5Ldelete
- H5Ldelete_by_idx
- H5Lexists
- H5Lget_info
- H5Lget_info_by_idx
- H5Lget_name_by_idx
- H5Lget_val
- H5Lget_val_by_idx
- H5Literate
- H5Literate_by_name
- H5Lmove
- H5Lvisit
- H5Lvisit_by_name

HDF5 Object Routines:

- H5Ocopy
- H5Odecr_refcount
- H5Oexists_by_name
- H5Oget_comment
- H5Oget_info
- H5Oget_info_by_idx
- H5Oget_info_by_name
- H5Oincr_refcount
- H5Oopen
- H5Oopen_by_idx
- H5Ovisit

- H5Oget_comment_by_name
- H5Olink
- H5Ovisit_by_name

HDF5 Datatype Routines:

- H5Tcommit
- H5Tcommit_anon
- H5Topen

Additions to the HDF5 API

The following routines will be added to the HDF5 API to support the new capabilities in the library.

Event Queue Operations:

H5EQcreate() – Create an event queue object.

```
hid_t H5EQcreate( hid_t fapl_id );
```

H5EQcreate creates an event queue to manage the status of asynchronous requests. Multiple event queues can be created and used concurrently in a program. The identifier for the event queue can be used in all asynchronous operations that push the request associated with the operation into the event queue. All requests that were added to the event queue are bundled and tracked together.

An event queue can be associated with one VOL plugin that the file access property list indicates. It is erroneous to use the same event queue for operations that use different VOL plugins than the one that the fapl_id was set to use.

The return value from H5EQcreate is negative on failure and a positive attribute identifier on success.

H5EQinsert() - Insert an asynchronous request into an event queue.

```
herr_t H5EQinsert(hid_t eventq_id, H5_request_t req )
```

H5EQinsert inserts the asynchronous request, req, into the event queue specified by eventq_id. The request is added in the top position of the event queue.

The return value from H5EQinsert is negative on failure and non-negative on success.

H5EQpop() - Retrieve the top asynchronous request from an event queue.

```
herr_t H5EQpop(hid_t eventq_id, H5_request_t *req )
```

H5EQpop removes the asynchronous request that was added most recently (the top one) from the event queue specified by eventq_id and returns it in req. The user is responsible for the request completion after popping it from the event queue. The user must eventually call H5AOwait() or H5AOtest() on the request returned and ensure completion of the request, otherwise resource leaks will develop.The return value from H5EQpop is negative on failure and non-negative on success.

H5EQwait() - Wait on all asynchronous requests in an event queue.

```
herr_t H5EQwait(hid_t eventq_id, int *num_requests, H5_status_t **status )
```

H5EQwait waits for all asynchronous requests in the event queue specified by eventq_id to complete and returns the number of requests that are in the event queue in the num_requests parameter and the completion status for each request in the queue in the status vector. The order of the requests in the status array corresponds to the order in which they were inserted in the event queue.

Possible status values are:

- H5AO_SUCCEEDED – The operation completed successfully.
- H5AO_FAILED – The operation completed, but was not successful.

Further work will be done on this function in future quarters to refine the way completion status information is delivered.

The return value from H5EQwait is negative on failure and non-negative on success.

H5EQclose() - Close an event queue.

```
herr_t H5EQclose(hid_t eventq_id )
```

H5EQclose closes the event queue specified by eventq_id and releases resources used by it. The event queue identifier, eventq_id, is no longer valid as a result of this call.

The return value from H5EQclose is negative on failure and non-negative on success.

Asynchronous Operations:

Note: The test and wait operations below can be expanded with MPI-like testall/waitall and/or testany/waitany variants as needed.

H5AOtest() – Test if an asynchronous operation has completed:

```
herr_t H5AOtest(H5_request_t *request_ptr, H5_status_t *status_ptr);
```

Calling H5AOtest will determine if an asynchronous operation has completed, and return the operation’s status to the application. Possible values returned for the operation’s status are:

- H5AO_PENDING – The operation has not yet completed
- H5AO_SUCCEEDED – The operation completed successfully
- H5AO_FAILED – The operation has completed, but failed

Once an asynchronous operation has completed (successfully or not), the request object is invalid for future test/wall calls.

An asynchronous operation has completed when the underlying operations have indicated success or failure. That may mean that the data was stored in cache at a lower layer (such as IOD), but from HDF5’s perspective the operation is now out of its control. If the operation is part of a transaction, that transaction’s commit operation must also complete successfully for the operation’s affect to become durable in the container.

The return value from H5AOtest is negative on failure and non-negative on success.

H5AOwait() – Wait for an asynchronous operation to complete:

```
herr_t H5AOwait(H5_request_t *request_ptr, H5_status_t *status_ptr);
```

Calling H5AOwait waits for an asynchronous operation to complete, returning the operation's status to the application. Possible values returned for the operation's status are:[11]

- H5AO_SUCCEEDED – The operation completed successfully
- H5AO_FAILED – The operation has completed, but failed

Once an asynchronous operation has completed (successfully or not), the request object is invalid for future test/wall calls.

An asynchronous operation has completed when the underlying operations have indicated success or failure. That may mean that the data was stored in cache at a lower layer (such as IOD), but from HDF5's perspective the operation is now out of its control. If the operation is part of a transaction, that transaction's commit operation must also complete successfully for the operation's affect to become durable in the container.

The return value from H5AOwait is negative on failure and non-negative on success.

End-to-End Integrity:

H5Pset_dxpl_checksum() – Set a checksum for data buffer in application memory:

```
herr_t H5Pset_dxpl_checksum(hid_t dxpl_id, uint32_t value);
```

H5Pset_dxpl_checksum sets a property in the dxpl_id data access property list specifying value as a user-supplied checksum for data written with a call to H5Dwrite_ff using dxpl_id. When this is set, the HDF5 IOD VOL client will create a checksum for the data and verify that checksum matches the value supplied by the user before sending the data on to the HDF5 VOL IOD server.

Regardless of whether the user supplies a checksum value, the HDF5 IOD VOL client will create a checksum that is compared with the value generated on the HDF5 IOD VOL server. The user must call H5checksum to obtain the value used in the call to H5Pset_dxpl_checksum. This ensures that all levels of the stack are using compatible checksum algorithms. [Optionally, we could have the application give us function pointer to a routine that the HDF5 library can use for verifying the buffer's checksum, but this is slower when the buffer doesn't need to be copied, since the H5checksum routine must be used for passing checksums to IOD]

The return value from H5Pset_dxpl_checksum is negative on failure and non-negative on success.

H5Pset_dxpl_checksum_ptr() – Specifies a memory location to receive checksum:

[11] Note for the future: we should add an intermediate "operation still pending, but data buffer can be re-used" state for asynchronous operations.

```
herr_t H5Pset_dxpl_checksum_ptr(hid_t dxpl_id, uint32_t * chksum_ptr)
```

H5Pset_dxpl_checksum_ptr sets a property in the data transfer property list specifying value as a memory location to receive a checksum for data read with a call to H5Dread_ff using dxpl_id. When this is set, the HDF5 IOD VOL client will put the checksum for the data into the memory location (`*chksum_ptr`) supplied by the user, allowing the user to compare the client's checksum to a value it generates using the H5checksum routine.

Regardless of whether the user supplies a memory location for the checksum, the HDF5 IOD VOL client will create a checksum and compare it with the value generated on the HDF5 IOD VOL server before the operation completes. The user must call H5checksum to obtain the value for its comparison with the received value. This ensures that all levels of the stack are using compatible checksum algorithms.H5checksum() – Perform a checksum on a buffer:[12]

H5checksum() – Generate a checksum.

```
uint32_t H5checksum( const void *buf, hsize_t length, H5_checksum_seed_t
*cseed )
```

H5checksum generates a checksum for the data in buf with size length bytes.

H5checksum will generate the same checksum for identical data regardless of whether the data is stored in a single contiguous buffer or in multiple non-continguous buffers. For non-contiguous buffers, H5checksum must be called once for each buffer to "accumulate" the checksum for the complete data.

The checksum seed structure is defined as follows:

```
typedef struct H5_checksum_seed_t {
    uint32_t a;
    uint32_t b;
    uint32_t c;
          int32_t state;
    size_t total_length;
} H5_checksum_seed_t;
```

If the checksum is for a data in a contiguous buffer, call H5checksum with NULL for cseed parameter.

Otherwise, for checksumming a set of non-contiguous regions that will be passed in a single call to H5Dwrite_ff, cseed captures the internal state of the checksum generation algorithm, allowing a single checksum to be generated for the set of non-contiguous data that is identical to the checksum that would be generated if the same data was checksummed in one contiguous block.

When creating a checksum for a set of non-contiguous buffers, H5checksum should be called on each contiguous portion of the buffer with length set to that portion's corresponding size in bytes. The a, b, c, and state fields in the checksum seed structure

[12] Note: we could improve this by creating a higher-level API routine to compute a checksum on an HDF5 selection within a buffer.

should be initialized to 0 for the first call to H5checksum, and total_length should be set to the total size in bytes of all the data to be checksummed (over all the non-contiguous sections of the data to checksum). Each call to H5checksum updates fields in cseed to capture the current internal state of the parameters used to compute the checksum and returns the updated checksum of the entire data that have been passed in so far to the routine with the same cseed parameter. The return values of the intermediate calls to H5checksum for a non-contiguous buffer can be discarded; only the last returned value is used as the checksum for the non-contiguous data. By using the same cseed structure in subsequent calls to H5checksum for each of the non-contiguous buffers, the overall checksum is computed step-by-step.

The generated checksum can be used with H5Pset_dxpl_checksum to attach a checksum to H5Dwrite_ff transfers and to verify the the checksum returned by H5Dread_ff in the buffer specified by H5Pset_dxpl_checksum_ptr.

H5Pset_edc_check() – *Existing routine* – Enables/disables checksum verification on data element reads.

Transactions, Container Versions, and Data Movement in the I/O Stack:

Name: H5TRstart

Signature:

herr_t `H5TRstart`(*hid_t* `file_id`, *uint64_t* `num_ranks`, *uint64_t* `transaction_num`, *hid_t* `eq_id`)

Purpose:

Starts a new transaction.

Description:

`H5TRstart` starts a new transaction on a container that is open for writing.

After the transaction has been started, objects in the container can be updated using the transaction number, and all updates will appear atomically when the transaction, and all lower-numbered transactions, are finished, aborted, or skipped.

The `file_id` is the file identifier for a container that is open for write. The container could have been created with `H5Fcreate_ff` or with `H5Fopen_ff, flags=H5F_ACC_RDWR`.

`num_ranks` indicates the number of processes participating in the transaction.

If `num_ranks=0`, the application has appointed a transaction leader to signal the start and finish of the transaction, and to coordinate with processes participating in the transaction regarding when they can begin updating and when their updates are done. When this is the case, only one process (the transaction leader) can call `H5TRstart` and `H5TRfinish` for the given `transaction_num`. This mode of operation is typical for tightly-coupled applications.

If `num_ranks>0,` the I/O stack (in particular, IOD) will track the status of the transaction. All `num_ranks` processes participating in the transaction must call `H5TRstart` with the same values for `num_ranks` and `transaction_num`. As each participating process is done with their updates in the transaction, they individually call `H5TRfinish` for this `transaction_num.` When all num_ranks processes have called `H5TRfinish` for this `transaction_num,` the I/O stack detects that the transaction is finished and updates appear atomically when all lower-numbered transactions are finished, aborted, or skipped. This mode of operation is typical for loosely-coupled applications.

EFF Note: As VOL implements this, clarify if when > 1 rank participates in a transaction whether you must have a process group that contains all of the ranks that are participating (or if just some ranks in a process group can participate).

`transaction_num` indicates the transaction that is being started.

EFF Note: iod_trans_start offers the option for leaving the transaction_number unspecified when it is called with num_ranks=0 and having IOD supply the next unused transaction number. This capability is currently not exposed by H5TRstart because it would require a more complicated calling structure to accommodate the returned transaction number in the asynchronous call. In addition, there is ongoing discussion regarding mixing support for application-supplied transaction numbers and IOD-transaction numbers, and race conditions that can occur. The capability may be exposed at a later date, possibly via a separate call that would "request and lock" the next transaction number, that could then be passed into H5TRstart.

The `eq_id` parameter indicates the event queue the request object for this call should be pushed onto when the function is executed asynchronously. The function may be executed synchronously by passing in `H5_EVENT_QUEUE_NULL` for the `eq_id` parameter.

EFF Note: May want to add property list argument for later extensions.

Parameters:

hid_t `file_id`	IN: File identifier for container open for write.
uint64_t `num_ranks`	IN: Number of process ranks participating in the transaction. If =0, only one rank will call H5TRstart for this transaction, and the application will manage the participating processes and call H5TRfinish from one rank. If >0, all ranks participating in this transaction will call H5TRstart with the same num_ranks and all ranks participating will also call H5TRfinish.
uint64_t *`transaction_num`*	IN: Value used to indicate transaction being started.
hid_t `eq_id`	IN: Event queue identifier specifying the queue that will be used to monitor the status of the request object associated with this function call when executed asynchronously. Use `H5_EVENT_QUEUE_NULL` for synchronous execution.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Note that when this routine is executed asynchronously, the return value from the routine only indicates whether the operation has been successfully scheduled for asynchronous execution. The actual success or failure of the asynchronous operation must be checked separately through the event queue.

Name: H5TRfinish

Signature:

herr_t `H5TRfinish(` *hid_t* `file_id,` *uint64_t* `transaction_num,` *hid_t* `eq_id )`

Purpose:

Finish a transaction that was started with `H5TRstart`.

Description:

`H5TRfinish` signals that no more updates will be made as part of a given transaction on a given container.

The `file_id` is the file identifier for a container that is open for write.

`transaction_num` indicates the transaction that is being finished.

If the transaction was started by a single process who called `H5TRstart` with `num_ranks=0`, then a single process must finish the transaction with a call to `H5TRfinish`.

If the transaction was started by one or more processes who called `H5TRstart` with `num_ranks>0`, then each of those processes must call `H5TRfinish`. The transaction is finished when all of the processes have called `H5TRfinish`.

After the transaction is finished and all lower-numbered transactions are finished, aborted, or explicitly skipped, the transaction is *committed* and all updates that were made as part of the transaction will become readable atomically.

The `eq_id` parameter indicates the event queue the request object for this call should be pushed onto when the function is executed asynchronously. The function may be executed synchronously by passing in `H5_EVENT_QUEUE_NULL` for the `eq_id` parameter.

EFF Note: Currently thinking that the default behavior for H5TRfinish will be to complete when the transaction is committed (not just finished). Will add property list argument to override default behavior (and allow completion on finish rather than commit). Property list argument will also allow for later extensions. Alternatively, may have H5TRcommit call that will finish & return on commit.

Parameters:

hid_t `file_id`	IN: File identifier for container open for write.
uint64_t `transaction_num`	IN: Value used to indicate transaction being finished.
hid_t `eq_id`	IN: Event queue identifier specifying the queue that will be used to monitor the status of the request object associated with this function call

when executed asynchronously. Use `H5_EVENT_QUEUE_NULL` for synchronous execution.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Note that when this routine is executed asynchronously, the return value from the routine only indicates whether the operation has been successfully scheduled for asynchronous execution. The actual success or failure of the asynchronous operation must be checked separately through the event queue.

Name: H5TRabort

Signature:

herr_t `H5TRabort(` *hid_t* `file_id,` *uint64_t* `transaction_num,` *unsigned* `flags,` *hid_t* `eq_id )`

Purpose:

Abort one or more transactions that were started with `H5TRstart`.

Description:

`H5TRabort` signals that all updates made as part of one or more transactions on a given container should be discarded.

The `file_id` is the file identifier for a container that is open for write.

`transaction_num` indicates the lowest numbered transaction that is being aborted.

The `flags` parameter specifies whether the single transaction specified by `transaction_num` should be aborted (`flags=ABORT_SINGLE`) or whether all transactions that have been started whose numbers are greater than or equal to `transaction_num` should be aborted (`flags=ABORT_ALL`). If unspecified, the default value is ABORT_ALL.

Unlike `H5TRfinish`, `H5TRabort` can always be called by a single process. `H5TRabort` will cause all of the transaction updates to be discarded, even if other processes continue updating and eventually call `H5TRfinish`.

Care should be taken when aborting a single transaction without also aborting higher-numbered transactions that have been started, as the higher-numbered transactions could depend on updates that were made in the single transaction being aborted. Typically, a single transaction should be aborted only when it involves updates to H5Dataset elements that are known to be under the sole control of the aborting process.

EFF Note: When running on an I/O stack that does not have burst buffers (IOD runs directly on DAOS), ABORT_SINGLE is not supported.

The `eq_id` parameter indicates the event queue the request object for this call should be pushed onto when the function is executed asynchronously. The function may be executed synchronously by passing in `H5_EVENT_QUEUE_NULL` for the `eq_id` parameter.

EFF Note: May want to add property list argument for later extensions.

Parameters:

hid_t `file_id`	IN: File identifier for container open for write.
uint64_t `transaction_num`	IN: Value used to indicate transaction being aborted.

unsigned `flags`	IN: Abort type
	`ABORT_SINGLE`
	Abort the single transaction identified by `transaction_num`
	`ABORT_ALL`
	Abort the `transaction_num` transaction, and all higher-numbered transactions that have been started on the same container.
	
	`ABORT_ALL` is the default.

hid_t `eq_id`	IN: Event queue identifier specifying the queue that will be used to monitor the status of the request object associated with this function call when executed asynchronously. Use `H5_EVENT_QUEUE_NULL` for synchronous execution.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Note that when this routine is executed asynchronously, the return value from the routine only indicates whether the operation has been successfully scheduled for asynchronous execution. The actual success or failure of the asynchronous operation must be checked separately through the event queue.

Name: H5TRskip

Signature:

herr_t `H5TRskip(` *hid_t* `file_id`, *uint64_t* `transaction_num,` *hid_t* `eq_id )`

Purpose:

Explicitly skip a transaction number for a given container.

Description:

`H5TRskip` signals that the application will not be using the identified transaction number for a given container.

The `file_id` is the file identifier for a container that is open for write.

`transaction_num` indicates the transaction number that will be skipped.

The `eq_id` parameter indicates the event queue the request object for this call should be pushed onto when the function is executed asynchronously. The function may be executed synchronously by passing in `H5_EVENT_QUEUE_NULL` for the `eq_id` parameter.

`H5TRskip` should always be called by a single process. It is an error to call both `H5TRskip` and `H5TRstart` with the same `file_id` and `transaction_num` parameters.

EFF Note: IOD currently doesn't offer a skip but it can be achieved by calling iod_trans_start immediately followed by iod_trans_finish.

If the application will not use one or more transaction numbers, they should be explicitly skipped so that higher-numbered transactions can be finished and committed.

EFF Note: May want to include a property list argument for future expansion (or consistency).

EFF Note: May want to offer option of skipping multiple transaction numbers w/ single call.

Parameters:

hid_t `file_id`	IN: File identifier for container open for write.
uint64_t `transaction_num`	IN: Value used to indicate transaction being skipped.
hid_t `eq_id`	IN: Event queue identifier specifying the queue that will be used to monitor the status of the request object associated with this function call when executed asynchronously. Use `H5_EVENT_QUEUE_NULL` for synchronous execution.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Note that when this routine is executed asynchronously, the return value from the routine only indicates whether the operation has been successfully scheduled for asynchronous execution. The actual success or failure of the asynchronous operation must be checked separately through the event queue.

Name: H5Fpersist_ff

Signature:

herr_t `H5Fpersist_ff(` *hid_t* `file_id,` *uint64_t* `container_version,` *hid_t* `eq_id )`

Purpose:

Copy data from IOD to DAOS, bringing the container contents on DAOS to the specified version.

Description:

`H5Fpersist_ff` requests that IOD update the specified container on DAOS to the specified version.

The `file_id` is the file identifier for a container that is open for write.

`container_version` indicates the version ID of the container that is to be persisted, and must correspond to a committed transaction in IOD. All updates to the container between the last persisted version and the version currently being persisted will be copied to DAOS as part of this persist request.

The `eq_id` parameter indicates the event queue the request object for this call should be pushed onto when the function is executed asynchronously. The function may be executed synchronously by passing in `H5_EVENT_QUEUE_NULL` for the `eq_id` parameter.

EFF Note: May want to include a property list argument for future expansion (or consistency). For example, might be used to request that the objects that have been persisted be evicted from the BB or that a snapshot be taken (and the name of the snapshot).

Parameters:

hid_t `file_id`	IN: File identifier for container open for write.
uint64_t `container_version`	IN: Value indicating the version to be persisted.
hid_t `eq_id`	IN: Event queue identifier specifying the queue that will be used to monitor the status of the request object associated with this function call when executed asynchronously. Use `H5_EVENT_QUEUE_NULL` for synchronous execution.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Note that when this routine is executed asynchronously, the return value from the routine only indicates whether the operation has been successfully scheduled for asynchronous execution. The actual success or failure of the asynchronous operation must be checked separately through the event queue.

Name: H5Fsnapshot_ff

Signature:

herr_t `H5Fsnapshot_ff(` *hid_t* `file_id,` *uint64_t* `container_version,` *const char** `name,` *hid_t* `eq_id )`

Purpose:

Make a snapshot of a container on DAOS.

Description:

`H5Fsnapshot_ff` requests that DAOS make a copy of the specified container and version on DAOS, and give it the indicated container name.

The `file_id` is the file identifier for a container that is open for write.

`container_version` indicates the version ID for which the snapshot is to be made. If this does not correspond to DAOS' HCE for the container, the snapshot request will fail.

EFF Note: We considered integrating the snapshot functionality into the H5Fpersist_ff call, but decided against it as there may be times when the application does not know it wants a snapshot when the data is persisted. Perhaps only later does it discover that a given version is "interesting" and worthy of a snapshot. By de-coupling the two, the application can request a snapshot of a given version at any point prior to doing the next persist.

`name` is the name given to the newly created snapshot.

The `eq_id` parameter indicates the event queue the request object for this call should be pushed onto when the function is executed asynchronously. The function may be executed synchronously by passing in `H5_EVENT_QUEUE_NULL` for the `eq_id` parameter.

EFF Note: May want to include a property list argument for future expansion (or consistency).

Parameters:

hid_t `file_id`	IN: File identifier for container open for write.
uint64_t `container_version`	IN: Version of container for which snapshot should be taken.
*const char** `name`	IN: Name of the snapshot.
hid_t `eq_id`	IN: Event queue identifier specifying the queue that will be used to monitor the status of the request object associated with this function call when executed asynchronously. Use `H5_EVENT_QUEUE_NULL` for synchronous execution.

Returns:

Returns a non-negative value if successful; otherwise returns a negative value.

Note that when this routine is executed asynchronously, the return value from the routine only indicates whether the operation has been successfully scheduled for asynchronous execution. The actual success or failure of the asynchronous operation must be checked separately through the event queue.

Data Layout Properties:

H5Pset_layout() – *Existing routine* – Choose chunked or contiguous layout for dataset storage. This property will be translated to an IOD hint when the dataset is created in the IOD/DAOS container.

H5Pset_write_mode() – Indicate special properties of write operations to an object:

```
herr_t H5Pset_write_mode(hid_t ocpl, H5P_write_mode_t mode);
```

Calling H5Pset_write_mode will indicate special properties of writing data to an object. Possible values returned for the mode are:

- H5P_APPEND_ONLY – Write operations will only append data to the object

Currently this call is only supported for dataset objects, but could be expanded to other objects in the future.

The return value from H5AOtest is negative on failure and non-negative on success.

Library Instructure:

EFF_init() – Initialize the Exascale FastForward storage stack:

```
int EFF_init(MPI_Comm comm, MPI_Info info, const char *fs_driver, const
char *fs_info);
```

Must be called by an application before any HDF5/IOD/DAOS API calls are made. The MPI communicator and info objects are used to set aside the IONs from the CNs and set up communication channels between each CN and an ION. The fs_driver and fs_info parameters choose the network driver to use for function shipper communications and pass configuration information to that driver, respectively.

The return value from EFF_init is negative on failure and non-negative on success.

File Objects/Properties:

H5Pset_fapl_vol_iod() – Use the IOD VOL plugin for container operations:

```
herr_t H5Pset_fapl_vol_iod(hid_t fapl_id, MPI_Comm comm, MPI_Info info);
```

Calling H5Pset_fapl_vol_iod will cause the HDF5 library to use the IOD VOL plugin for accessing the HDF5 container object (as opposed to the native HDF5 file format, or another storage/access mechanism). The communicator and info parameters are used to set up communication channels for collective operations on the HDF5 container.

Calling this routine is *mandatory* to use the HDF5 API capabilities described in this document.

The return value from H5Pset_fapl_vol_iod is negative on failure and non-negative on success.

H5Pset_eff_snapshot() – Set snapshot to use when opening a container:

```
herr_t H5Pset_eff_snapshot(hid_t fapl_id, uint64_t snapshot_value);
```

Calling H5Pset_eff_snapshot will set a container snapshot value in the file access property list, to use when opening the HDF5 container, instead of the default action of accessing the latest consistent version of the container.

The return value from H5Pset_eff_snapshot is negative on failure and non-negative on success.

Dataset Objects:

See the HDF5 API reference man pages in the *User's Guide to Fast Forward Features in HDF5 (Revision 2.0)* for the definitive versions of these routines.

H5DOappend() – Perform an optimized append operation on a dataset:

```
herr_t H5DOappend(hid_t dataset_id, hid_t dxpl_id, unsigned axis, size_t
extension, hid_t memtype, const void *buffer);

herr_t H5DOappend_ff(hid_t dataset_id, hid_t dxpl_id, unsigned axis, size_t
extension, hid_t memtype, const void *buffer, uint64_t transaction_number,
hid_t event_queue_id);
```

The H5DOappend routine extends a dataset `extension` number of elements along the dimension specified by `axis` and writes elements of `memtype` datatype in `buffer` to the new elements. The H5DOappend_ff routine is identical in functionality, but allows for asynchronous operation and inclusion in a transaction.

This routine combines calling H5Dset_extent, H5Sselect_hyperslab and H5Dwrite into a single, convenient routine that simplifies application development for the common case of appending elements to an existing dataset and improves performance of the overall set of operations.

When the dataset has more than one dimension, appending to one axis will write a contiguous hyperslab over the other axes. For example, if a 3-D dataset currently has dimensions (3, 5, 8), extending the 0th axis (currently of size 3) by 3 will append 3*5*8 = 120 elements (which must be pointed to by the `buffer` parameter) to the dataset, making its final dimensions (6, 5, 8).

If a dataset has more than one axis with an unlimited dimension, any of those axes may be appended to, although only along one axis per call to H5DOappend.

The return value from H5DOappend is negative on failure and non-negative on success.

H5DOsequence() – Perform an optimized stream-oriented read operation on a dataset:

```
herr_t H5DOsequence(hid_t dataset_id, hid_t dxpl_id, unsigned axis, hsize_t
start, size_t sequence, hid_t memtype, void *buffer);
```

```
herr_t H5DOsequence_ff(hid_t dataset_id, hid_t dxpl_id, unsigned axis,
hsize_t start, size_t sequence, hid_t memtype, void *buffer, uint64_t
transaction_number, hid_t event_queue_id);
```

The H5DOsequence routine reads a sequence of `sequence` number of elements along the dimension specified by `axis`, starting at offset `start`, from a dataset into `buffer` of `memtype` datatype. The H5DOsequence_ff routine is identical in functionality, but allows for asynchronous operation and inclusion in a transaction.

This routine combines calling H5Sselect_hyperslab and H5Dread into a single, convenient routine that simplifies application development for the common case of sequenced reads from an existing dataset.

When the dataset has more than one dimension, sequencing along one axis will read a contiguous hyperslab over the other axes. For example, if a 3-D dataset currently has dimensions (6, 5, 8), a sequenced read of size 3 along the 0th axis, starting at offset 0 will read 3*5*8 = 120 elements from the dataset into the `buffer`.

The return value from H5DOsequence is negative on failure and non-negative on success.

H5DOset() – Write a single element to a dataset:

```
herr_t H5DOset(hid_t dataset_id, hid_t dxpl_id, const hsize_t coord[],hid_t
memtype, const void *buffer);
```

```
herr_t H5DOset_ff(hid_t dataset_id, hid_t dxpl_id, const hsize_t
coord[],hid_t memtype, const void *buffer, uint64_t transaction_number,
hid_t event_queue_id);
```

The H5DOset routine writes a single element at offset `coord`, to a dataset from `buffer` of `memtype` datatype. The H5DOset_ff routine is identical in functionality, but allows for asynchronous operation and inclusion in a transaction.

This routine combines calling H5Sselect_hyperslab and H5Dwrite into a single, convenient routine that simplifies application development for the common case of writing a single element from a dataset.

The return value from H5DOset is negative on failure and non-negative on success.

H5DOget() – Read a single element from a dataset:

```
herr_t H5DOget(hid_t dataset_id, hid_t dxpl_id, const hsize_t coord[],hid_t
memtype, void *buffer);
```

```
herr_t H5DOget_ff(hid_t dataset_id, hid_t dxpl_id, const hsize_t
coord[],hid_t memtype, void *buffer, uint64_t transaction_number, hid_t
event_queue_id);
```

The H5DOget routine reads a single element at offset `coord`, from a dataset to `buffer` of `memtype` datatype. The H5DOget_ff routine is identical in functionality, but allows for asynchronous operation and inclusion in a transaction.

This routine combines calling H5Sselect_hyperslab and H5Dread into a single, convenient routine that simplifies application development for the common case of retrieving a single element from a dataset.

The return value from H5DOget is negative on failure and non-negative on success.

H5Pset_dcpl_append_only () – Set a property to indicate whether access to Dataset is in an append only fashion (default is FALSE):

```
herr_t H5Pset_dcpl_append_only(hid_t dcpl_id, hbool_t flag);
```

The H5Pset_dcpl_append_only routine sets a property on the data creation property list, that is used in future operations to create a dataset (H5Dcreate_ff), to indicate that future access to a dataset will be in an append only manner, with no random I/O of elements in the middle of a dataset, and no overwrites of existing elements. This will allow the HDF5 library to store data elements for the dataset in a more optimized fashion.

The return value from H5Pset_dcpl_append_only is negative on failure and non-negative on success.

Group Objects:

None yet

Named Datatype Objects:

None yet

Attribute Objects:

None yet

Link Objects:

None yet

Map Objects:

H5Mcreate() – Create a new map object:

```
hid_t H5Mcreate(hid_t loc_id, const char *name, hid_t keytype, hid_t
valtype, hid_t lcpl_id, hid_t mcpl_id, hid_t mapl_id);
```

```
hid_t H5Mcreate_ff(hid_t loc_id, const char *name, hid_t keytype, hid_t
valtype, hid_t lcpl_id, hid_t mcpl_id, hid_t mapl_id, uint64_t
transaction_number, hid_t event_queue_id);
```

The H5Mcreate routine creates a new map object named `name` at the location given by `loc_id`. Map creation and access property lists (`mcpl_id` and `mapl_id`) modify the new map object's behavior. All keys for the map are of `keytype` datatype and all values for the map are of `valtype` datatype. The H5Mcreate_ff routine is identical in functionality, but allows for asynchronous operation and inclusion in a transaction.

Map IDs returned from this routine must be released with H5Mclose.

The return value from H5Mcreate is negative on failure and a non-negative map object ID on success.

H5Mopen() – Open an existing map object:

```
hid_t H5Mopen(hid_t loc_id, const char *name, hid_t mapl_id);
```

```
hid_t H5Mopen_ff(hid_t loc_id, const char *name, hid_t mapl_id, uint64_t
transaction_number, hid_t event_queue_id);
```

The H5Mcreate routine opens an existing map object named `name` at the location given by `loc_id`. The map access property list (`mapl_id`) modifies the map object's behavior. The H5Mopen_ff routine is identical in functionality, but allows for asynchronous operation and inclusion in a transaction.

Map IDs returned from this routine must be released with H5Mclose.

The return value from H5Mopen is negative on failure and a non-negative map object ID on success.

H5Mset() – Insert or overwrite a key/value pair in a map object:

```
herr_t H5Mset(hid_t map_id, hid_t key_mem_type_id, const void *key, hid_t
val_mem_type_id, const void *value, hid_t dxpl_id);
```

```
herr_t H5Mset_ff(hid_t map_id, hid_t key_mem_type_id, const void *key,
hid_t val_mem_type_id, const void *value, hid_t dxpl_id, uint64_t
transaction_number, hid_t event_queue_id);
```

The H5Mset routine inserts or sets a key/value pair in a map object, given by `map_id`. The key (pointed to by `key`) is of type `key_mem_type_id` in memory and the value (pointed to by `value`) is of type `value_mem_type_id` in memory. The data transfer property list (`dxpl_id`) may modify the operation's behavior. The H5Mset_ff routine is identical in functionality, but allows for asynchronous operation and inclusion in a transaction.

The return value from H5Mset is negative on failure and non-negative on success.

H5Mget() – Retrieves a value from a map object:

```
herr_t H5Mget(hid_t map_id, hid_t key_mem_type_id, const void *key, hid_t
val_mem_type_id, void *value, hid_t dxpl_id);
```

```
herr_t H5Mget_ff(hid_t map_id, hid_t key_mem_type_id, const void *key,
hid_t val_mem_type_id, void *value, hid_t dxpl_id, uint64_t
transaction_number, hid_t event_queue_id);
```

The H5Mget routine retrieves a value from a map object, given by `map_id`. The key value used to retrieve the value (pointed to by `key`) is of type `key_mem_type_id` in memory and the value (pointed to by `value`) is of type `value_mem_type_id` in memory. The data transfer property list (`dxpl_id`) may modify the operation's behavior. The H5Mget_ff routine is identical in functionality, but allows for asynchronous operation and inclusion in a transaction.

The return value from H5Mget is negative on failure and non-negative on success.

H5Mget_types() – Retrieves the datatypes for keys and values of a map object:

```
herr_t H5Mget_types(hid_t map_id, hid_t *key_type_id, hid_t *val_type_id);
```

```
herr_t H5Mget_types(hid_t map_id, hid_t *key_type_id, hid_t *val_type_id,
uint64_t transaction_number, hid_t event_queue_id);
```

The H5Mget_types routine retrieves the datatypes for the keys and values of a map, given by `map_id`. The key datatype is returned in `key_type_id` and the value datatype is returned in `value_type_id`. The H5Mget_types_ff routine is identical in functionality, but allows for asynchronous operation and inclusion in a transaction.

Either (or both) of the datatype ID pointers may be NULL, if that datatype information is not desired.

Any datatype IDs returned from this routine must be released with H5Tclose.

The return value from H5Mget_types is negative on failure and non-negative on success.

H5Mget_count() – Retrieves the number of key/value pairs in a map object:

```
herr_t H5Mget_count(hid_t map_id, hsize_t *count);
```

```
herr_t H5Mget_count_ff(hid_t map_id, hsize_t *count, uint64_t
transaction_number, hid_t event_queue_id);
```

The H5Mget_count routine retrieves the number of key/value pairs in a map, given by `map_id`. The H5Mget_count_ff routine is identical in functionality, but allows for asynchronous operation and inclusion in a transaction.

The return value from H5Mget_count is negative on failure and non-negative on success.

H5Mexists() – Check if a key exists in a map object:

```
herr_t H5Mexists(hid_t map_id, hid_t key_mem_type_id, const void *key,
hbool_t *exists);
```

```
herr_t H5Mexists_ff(hid_t map_id, hid_t key_mem_type_id, const void *key,
hbool_t *exists, uint64_t transaction_number, hid_t event_queue_id);
```

The H5Mexists routine checks if a key exists in a map, given by `map_id`. The key value used (pointed to by `key`) is of type `key_mem_type_id` in memory and the status of the key in the map is returned in the `exists` pointer's value. The H5Mexists_ff routine is identical in functionality, but allows for asynchronous operation and inclusion in a transaction.

The return value from H5Mexists is negative on failure and non-negative on success.

H5Miterate() – Iterate over the key/value pairs in a map object:

```
herr_t H5Miterate(hid_t map_id, hid_t key_mem_type_id, hid_t
value_mem_type_id, H5M_iterate_func_t callback_func, void *context);
```

The H5Miterate routine iterates over the key/value pairs in a map, given by `map_id`. The user-defined callback routine, given by `callback_func`, defined below, will be invoked for each key/value pair in the map:

```
typedef int (*H5M_iterate_func_t)(const void *key, const void *value, void
*context);
```

Keys and values presented to the callback routine will be in `key_mem_type_id` and `value_mem_type_id` format, respectively. Additional information may be given to the callback routine with the `context` parameter, which is passed unmodified from the call to H5Miterate to the application's callback. The iteration callback routine should obey the same rules as other HDF5 iteration callbacks: return H5_ITER_ERROR for an error condition (which will stop iteration), H5_ITER_CONT for success (with continued iteration) and H5_ITER_STOP for success (but stop iteration).

As with other "iteration" routines in the HDF5 API, there is no asynchronous analog for this routine, as there is no way to have user callback routines get invoked asynchronously.

The return value from H5Miterate is negative on failure and non-negative on success.

H5Mdelete() – Delete a key/value pair in a map object:

```
herr_t H5Mdelete(hid_t map_id, hid_t key_mem_type_id, const void *key);
```

```
herr_t H5Mdelete_ff(hid_t map_id, hid_t key_mem_type_id, const void *key,
uint64_t transaction_number, hid_t event_queue_id);
```

The H5Mdelete routine removes a key/value pair from a map, given by `map_id`. The key value used (pointed to by `key`) is of type `key_mem_type_id` in. The H5Mdelete_ff routine is identical in functionality, but allows for asynchronous operation and inclusion in a transaction.

The return value from H5Mdelete is negative on failure and non-negative on success.

H5Mclose() – Close a map object:

```
herr_t H5Mclose(hid_t map_id);

herr_t H5Mclose_ff(hid_t map_id, uint64_t transaction_number, hid_t
event_queue_id);
```

The H5Mclose routine terminates access to a map, given by `map_id`. The H5Mclose_ff routine is identical in functionality, but allows for asynchronous operation and inclusion in a transaction.

The return value from H5Mclose is negative on failure and non-negative on success.

Query Objects:

H5Qcreate() – Create a new query object:

```
hid_t H5Qcreate(H5Q_query_type_t query_type, H5Q_match_op_t match_op, ...);
```

The H5Qcreate routine creates 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. The following table describes the possible query types, match conditions and varargs parameters for the H5Qcreate parameters:

Query Type (`H5Q_query_type_t`)	**Match Conditions** (`H5Q_match_op_t`)	**Varargs parameters**
H5Q_TYPE_DATA_ELEMENT (selects data elements)	H5Q_MATCH_EQUAL H5Q_MATCH_NOT_EQUAL H5Q_MATCH_LESS_THAN H5Q_MATCH_GREATER_THAN	hid_t val_datatype_id, const void *val (gives the element value for the match condition)
H5Q_TYPE_ATTR_NAME (selects attributes)	H5Q_MATCH_EQUAL H5Q_MATCH_NOT_EQUAL	const char *name (gives the string for the match condition)
H5Q_TYPE_LINK_NAME (selects objects)	H5Q_MATCH_EQUAL H5Q_MATCH_NOT_EQUAL	const char *name (gives the string for the match condition)

Examples of possible query creation calls are:

Query to select data elements equal to 17:

```
int x=17;

hid_t q1=H5Qcreate(H5Q_TYPE_DATA_ELEMENT, H5Q_MATCH_EQUAL, H5T_NATIVE_INT,
&x);
```

Query to select objects with link names equal to "Pressure":

```
hid_t q2=H5Qcreate(H5Q_TYPE_LINK_NAME, H5Q_MATCH_EQUAL, "Pressure");
```

Many more query types are possible, including types that select attribute values or types that select datasets based on their datatype or dataspace (such as datasets with an integer datatype or with three dimensions), but the types above represent a starting point and more can always be added over time. The same could be said for the match conditions, with additions of regular expressions for attribute or link names, etc. possible in the future.

There is no asynchronous form of this operation, or transaction ID parameter, as query objects don't persist in HDF5 containers.

Query IDs returned from this routine must be released with H5Qclose.

The return value from H5Qcreate is negative on failure and a non-negative query object ID on success.

H5Qcombine() – Combine query objects to create a new query object:

```
hid_t H5Qcombine(hid_t query1, H5Q_combine_op_t combine_op, hid_t query2);
```

The H5Qcombine routine creates a new query object by combining two query objects (given by `query1` and `query2`), using the combination operator `combine_op`. Valid combination operators are: H5Q_COMBINE_AND and H5Q_COMBINE_OR (although more operators can be created in the future).

An example of a query combination to select data elements equal to 17 in datasets with link names equal to "Pressure" is:

```
int x=17;

hid_t q1=H5Qcreate(H5Q_TYPE_DATA_ELEMENT, H5Q_MATCH_EQUAL, H5T_NATIVE_INT,
&x);

hid_t q2=H5Qcreate(H5Q_TYPE_LINK_NAME, H5Q_MATCH_EQUAL, "Pressure");

hid_t q3=H5Qcombine(q1, H5Q_COMBINE_AND, q2);
```

Query IDs returned from this routine must be released with H5Qclose.

The return value from H5Qcombine is negative on failure and a non-negative query object ID on success.

H5Qclose() – Close a query object:

```
herr_t H5Qclose(hid_t query_id);
```

The H5Qclose terminates access to a query object, given by `query_id`.

The return value from H5Qclose is negative on failure and non-negative on success.

View Objects:

H5Vcreate() – Create a new view object:

```
hid_t H5Vcreate(hid_t container_id, hid_t query_id);
```

```
hid_t H5Vcreate_ff(hid_t container_id, hid_t query_id, hid_t
event_queue_id);
```

The H5Vcreate routine creates a new view object on the container, or portion of container, given by `container_id`, using the query given by `query_id` to determine what components of the container are included in the view. The H5Vcreate_ff routine is identical in functionality, but allows for asynchronous operation (a transaction ID is not included as views are not stored in containers).

The container ID can be an HDF5 File ID (indicating that the entire container is used to construct the view), an HDF5 group ID (indicating that just the group and objects recursively linked to from it are used to construct the view), or an HDF5 dataset ID (indicating that just the dataset and its elements are used to construct the view). Some combinations of container and query IDs may result in a view with nothing selected (such as passing a query on link names when using a dataset ID for a container ID, etc.).

View IDs returned from this routine must be released with H5Vclose.

The return value from H5Vcreate is negative on failure and a non-negative view object ID on success.

H5Vget_container() – Retrieve the container for a view object:

```
herr_t H5Vget_container(hid_t view_id, hid_t *container_id);
```

The H5Vget_container routine returns the container for a view object, given by `view_id`, in the `container_id` parameter.

Container IDs returned from this routine can be queried for their ID type with H5Iget_type and must be released with H5Fclose/H5Gclose/H5Dclose.

The return value from H5Vget_container is negative on failure and non-negative on success.

H5Vget_query() – Retrieve the query for a view object:

```
herr_t H5Vget_query(hid_t view_id, hid_t *query_id);
```

The H5Vget_query routine returns a copy of the query used to create a view object, given by `view_id`, in the `query_id` parameter.

Query IDs returned from this routine must be released with H5Qclose.

The return value from H5Vget_query is negative on failure and non-negative on success.

H5Vget_counts() – Retrieve aspects of a view object:

```
herr_t H5Vget_counts(hid_t view_id, hsize_t *attr_count, hsize_t
*obj_count, hsize_t *elem_region_count);
```

The H5Vget_counts routine retrieves various aspects of a view object, given by `view_id`. The number of attributes, objects and dataset element regions in the view is returned in the `attr_count`, `obj_count` and `elem_region_count` parameters, respectively.

The return value from H5Vget_counts is negative on failure and non-negative on success.

H5Vget_attrs() – Retrieve attributes referenced by a view object:

```
herr_t H5Vget_attrs(hid_t view_id, hsize_t start, hsize_t count, hid_t
attr_id[]);
```

The H5Vget_attrs routine retrieves attributes referenced by a view object, given by `view_id`. Attributes referenced by the view are uniquely enumerated internally to the view object, and the `count` attributes returned from this routine begin at offset `start` in that enumeration and are placed in the array of IDs given by `attr_id`.

Attribute IDs returned in `attr_id` must be released with H5Aclose.

The return value from H5Vget_attrs is negative on failure and non-negative on success.

H5Vget_objs() – Retrieve HDF5 objects referenced by a view object:

```
herr_t H5Vget_objs(hid_t view_id, hsize_t start, hsize_t count, hid_t
obj_id[]);
```

The H5Vget_objs routine retrieves objects referenced by a view object, given by `view_id`. Objects referenced by the view are uniquely enumerated internally to the view object, and the `count` objects returned from this routine begin at offset `start` in that enumeration and are placed in the array of IDs given by `obj_id`.

Object IDs returned in `obj_id` must be released with H5Oclose.

The return value from H5Vget_objs is negative on failure and non-negative on success.

H5Vget_elem_regions() – Retrieve data element regions referenced by a view object:

```
herr_t H5Vget_elem_regions(hid_t view_id, hsize_t start, hsize_t count,
hid_t dataset_id[], hid_t dataspace_id[]);
```

The H5Vget_elem_regions routine retrieves dataset and dataspace (with selection) pairs referenced by a view object, given by `view_id`. Data element regions referenced by the view are uniquely enumerated internally to the view object, and the `count` regions returned from this routine begin at offset `start` in that enumeration and are placed in the array of IDs given by `dataset_id and dataspace_id`. Both `dataset_id` and `dataspace_id` must be large enough to hold at least `count` IDs.

Each dataspace ID returned from this routine corresponds to the dataset ID at the same offset as the dataspace ID. Each dataspace returned by this routine has a selection defined, which corresponds to the elements from the dataset that are included in the view.

Dataset and dataspace IDs returned in `dataset_id` and `dataspace_id` must be released with H5Dclose and H5Sclose, respectively.

The return value from H5Vget_elem_regions is negative on failure and non-negative on success.

H5Vclose() – Close a view object:

```
herr_t H5Vclose(hid_t view_id);
```

The H5Vclose terminates access to a view object, given by `view_id`.

The return value from H5Vclose is negative on failure and non-negative on success.

Index Objects:

H5Xcreate() – Create a new index object in a container:

```
hid_t H5Xcreate(hid_t container_id, H5X_type_t itype, hid_t scope_id);
```

```
hid_t H5Xcreate_ff(hid_t container_id, H5X_type_t itype, hid_t scope_id,
uint64_t transaction_number, hid_t event_queue_id);
```

The H5Xcreate routine creates a new index object of type `itype` (from the list of index types below) in a container, given by `container_id`, over a set of objects in the container, given by `scope_id`. The H5Xcreate_ff routine is identical in functionality, but allows for asynchronous operation and inclusion in a transaction.

An index may be one of the following types[13]:

- H5X_TYPE_LINK_NAME – Indexes names of links to objects
- H5X_TYPE_ATTR_NAME – Indexes names of attributes
- H5X_TYPE_DATA_ELEMENT – Indexes elements of datasets

The set of objects that an index applies to is determined by the `scope_id` passed to H5Xupdate. Three types of scope are currently supported, determined by the type of ID passed in for the `scope_id`:

- H5File ID – Creates indices that include information about the contents of the whole container
- H5Group ID – Creates indices that include information about a group and all its descendants
- H5Dataset ID – Creates indices that include information about a dataset

[13] See *Appendix I – Aspects of the HDF5 Data Model* for a list of current and future index types.

Note that some combinations, such as creating a link name index on a dataset, are invalid and will fail with an error.

Indices created in a container are not populated with information until H5Xupdate is called.

Index IDs returned from this routine must be released with H5Xclose.

The return value from H5Xcreate is negative on failure and a non-negative index object ID on success.

H5Xopen() – Open an index object in a container:

```
hid_t H5Xopen(hid_t container_id, hsize_t offset);
```

```
hid_t H5Xopen_ff(hid_t container_id, hsize_t offset, hid_t event_queue_id);
```

The H5Xopen routine opens an existing index object in a container, given by `container_id`, using the offset given by `offset` to determine which index within the container to open. The H5Xopen_ff routine is identical in functionality, but allows for asynchronous operation.

Index IDs returned from this routine must be released with H5Xclose.

The return value from H5Xopen is negative on failure and a non-negative index object ID on success.

H5Xget_count() – Determine the number of index objects in a container:

```
herr_t H5Xget_count(hid_t container_id, hsize_t *index_count);
```

```
herr_t H5Xget_count_ff(hid_t container_id, hsize_t *index_count, hid_t
event_queue_id);
```

The H5Xget_count routine returns the number of index objects in a container, given by `container_id`, in the `index_count` parameter. The H5Xget_count_ff routine is identical in functionality, but allows for asynchronous operation.

The return value from H5Xget_count is negative on failure and non-negative on success.

H5Xget_type() – Retrieve the type of an index object:

```
herr_t H5Xget_type(hid_t index_id, H5X_type_t *itype);
```

```
herr_t H5Xget_type_ff(hid_t index_id, H5X_type_t *itype, hid_t
event_queue_id);
```

The H5Xget_type routine returns the type of an index, given by `index_id`, in the `itype` parameter. The H5Xget_type_ff routine is identical in functionality, but allows for asynchronous operation.

Possible index type values are:

- H5X_TYPE_LINK_NAME – Index tracks names of links to objects

- H5X_TYPE_ATTR_NAME – Index tracks names of attributes
- H5X_TYPE_DATA_ELEMENT – Index tracks elements of datasets

The return value from H5Xget_count is negative on failure and non-negative on success.

H5Xget_scope() – Retrieve the scope of an index object:

```
herr_t H5Xget_scope(hid_t index_id, hid_t *scope_id);

herr_t H5Xget_scope_ff(hid_t index_id, hid_t *scope_id, hid_t
event_queue_id);
```

The H5Xget_scope routine returns the scope of an index, given by `index_id`, in the `scope_id` parameter. The H5Xget_scope_ff routine is identical in functionality, but allows for asynchronous operation.

The ID returned in the scope_id parameter is one of three types:

- H5File ID – Indicates that the scope of the index is over the entire container
- H5Group ID – Indicates that the scope of the index is over a group and its descendants
- H5Dataset ID – Indicates that the scope of the index is a particular dataset

The ID returned is valid HDF5 object ID and can be queried for its type with H5Iget_type. The ID returned is valid for file, group or dataset operations, and must be closed with the corresponding close API call (H5Fclose, H5Gclose or H5Dclose).

The return value from H5Xget_count is negative on failure and non-negative on success.

H5Xis_current() – Checks if an index is current:

```
htri_t H5Xis_current(hid_t index_id);
```

The H5Xis_current routine queries whether the index will be used in assisting the creation of view objects.

The return value from H5Xis_current is negative on failure and non-negative (TRUE/FALSE) on success.

H5Xupdate() – Update an index object:

```
herr_t H5Xupdate(hid_t index_id);

herr_t H5Xupdate_ff(hid_t index_id, uint64_t transaction_number, hid_t
event_queue_id);
```

The H5Xupdate routine updates the information tracked by an index object, given by `index_id`, for use in future queries on the container. The H5Xupdate_ff routine is identical in functionality, but allows for asynchronous operation and inclusion in a transaction.

Index objects track the version of the container they were last updated with, and if the version of the last update does not match the current version of the container, they may be ignored when queries are executed on the container to create view objects. H5Xupdate should be called as the only operation within a transaction, to guarantee that they reflect the current state of the container.

The return value from H5Xupdate is negative on failure and non-negative on success.

H5Xclose() – Close an index object:

```
herr_t H5Xclose(hid_t index_id);
```

The H5Xclose terminates access to a index object, given by `index_id`.

The return value from H5Xclose is negative on failure and non-negative on success.

Open Issues

Some of the existing HDF5 routines that are extended above don't need a transaction ID (e.g. routines which only read information from the container, like H5Lexists) and so might need to be modified differently (they would be "generically" modified to take an event queue identifier, but not a transaction number parameter).

During our internal design discussions, we have considered having a mechanism for tagging objects in some way so that they are prefetched/persisted/removed together. It also seems more likely that an application would want to prefetch/persist/remove objects at the IOD layer instead of transactions. We are considering use cases for these behaviors and may include them in the full design for transactions, next quarter.

Risks & Unknowns

As the changes to the HDF5 library are dependent on capabilities added to multiple lower layers of the software stack (the function shipper, IOD and DAOS layers), it is likely that changes at those layers will ripple up through the HDF5 API and cause additional work at this layer. On the other hand, we can always mitigate the effect of changes at lower levels by abstracting those capabilities and implementing support within the HDF5 library for features missing or different below it.

Conversely, the demands of the applications that use the HDF5 API may pull the features and interface in unexpected directions as well, in order to provide the necessary capabilities for the application to efficiently and effectively store its data. These two forces must be balanced over the course of the project, hopefully producing a high quality storage stack that is useful to applications at the exascale.

Appendix I – Aspects of the HDF5 Data Model

The following table describes aspects of the HDF5 data model, which are possible candidates for indices, queries, and inclusion in views.

Aspect	Details	Indexable	Queryable	View Object

				Reference
Link Name	Name of link to an object	Y	Y	Object reference[14]
Attribute Name	Name of attribute on an object	Y	Y	Attribute reference[15]
Map Key	Key of map entry	N	N	N/A
Datatype	Datatype of attribute or dataset	N	N	N/A
Dataspace	Dataspace of attribute or dataset	N	N	N/A
Dataset Element	Value of element in a dataset	Y	Y	Region reference
Attribute Value	Value of attribute	N	N	Attribute reference[15]
Map Value	Value of map entry	N	N	N/A

[14] An object reference isn't precisely the same as a link name reference, but they are functionally identical in most application usage.

[15] Under development.