Date: **September 29, 2013** **Delivered as part of Milestones 5.6 & 5.7**	# ***Design and Implementation of FastForward Features in HDF5*** **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	Description	Authors
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.	Mohamad Chaarawi, Ruth Aydt, Quincey Koziol, The HDF Group

		Change all asynchronous operations to take 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
July 22, 2013	22.0	Reformatted, primarily to add numbered section headings.	Ruth Aydt, The HDF Group
September 29, 2013	23.0	• Change title. • Update text in Generic Changes to HDF5 API Routines. • Move routine descriptions to User's Guide reference man pages, expanding details. • Refer readers to *HDF5 Data in IOD Containers Layout Specification* for details on Storing HDF5 Objects in IOD Containers. • Change from Event Queues and Asynchronous Objects to Event Stacks and remove Asynchronous Objects. • Update discussion of Optimized Append. • Update Transactions section to reflect latest design. • Deliver to DOE as part of Milestones 5.6 & 5.7	Ruth Aydt, Quincey Koziol, The HDF Group

1 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.

2 Definitions

ACG = Arbitrarily Connected Graph

AXE = Asynchronous Execution Engine

BB = Burst Buffer

CN = Compute Node

DAOS = Distributed Application Object Storage

EFF = Exascale FastForward

IOD = I/O Dispatcher

ION = I/O Node

VOL = Virtual Object Layer

3 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 adding support for optimized appends and 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.

4 Specification

4.1 New HDF5 Library Capabilities

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

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

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

4.1.1 Asynchronous I/O and Event Stack 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 stack 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 was to be passed as a parameter in all the newly added asynchronous routines instead of the request object that was used in Quarter 3.

After further consideration in Quarter 5, the Event Queue object was renamed Event Stack, and the test and wait functionality slated for the Asynchronous Object routines was integrated into Event Stack APIs. The Event Stack Object APIs also allow access to more complete function call, completion status, and error information for the asynchronous requests that was previously provided.

Although the Event Stack design was completed in Quarter 5, the code is not yet implemented and the library source and example programs delivered in Quarter 5 continue to use the Event Queue and Asynchronous Object APIs. The Event Stack APIs are included in the User Guide delivered in Quarter 5, and other routines in the User Guide were also updated as if the Event Stack Objects were being used, while in fact they continue to use Event Queues and Asynchronous Objects. In this case, the User Guide is ahead of the code base. Event Stack Objects will be delivered in Quarter 6.

An Event Stack provides an organizing structure for managing and monitoring the status of functions that have been called asynchronously. The association of an event for an asynchronous function call with a given Event Stack has nothing to do with the order in which the function (the event) will execute or complete. The Event Stacks merely organize the IDs that are needed to track the status of the asynchronous functions.

Once an Event Stack is created, its identifier can be passed to other HDF5 APIs that will be run asynchronously. The event associated with an asynchronous call will be pushed onto the Event Stack whose identifier was passed as a parameter to the function. The application can monitor the completion status of individual events via `H5ESwait` and `H5EStest`. The application can also wait or test the status of all the request objects in a given Event Stack. The `H5ESget_event_info` routine provides information on the calling parameters, completion status, and error codes for one or more events in an Event Stack. Event cancellation is also supported.

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 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

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

will no longer generate asynchronous dependencies. If the asynchronous operation fails, the future object ID issued for the operation (and any future object IDs 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, and details on new H5ES* API routines to create and operate on event stack objects.

4.1.2 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.

In addition to checksumming data elements and metadata, which protect against passive or active corruption of data buffers (i.e., corruption of an in-place buffer in memory or while a memory buffer is being moved), additional integrity checks may be performed as information from a CN is sent to an ION to be stored in a container. Each time information is transformed from one representation to another, serializing a metadata data structure in memory into a buffer for storage or changing data element values from one endianness to another, for example, the transformed result can be verified to ensure that the transformation was accurate. However, at this time, we don't see a strong need to implement every possible verification step and have limited our implementation to only verifying data movement from the CN to the ION, across the interconnect fabric. As needed, or as part of a future project, we will implement the full set of verification steps on all data transformations, along with properties for enabling/disabling individual verification steps.

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.

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

Several revisions occurred in this section during Quarter 5. Most significant was the addition of the explicit read context for a given version of a container, and the specification of a read context when a transaction is started. The transaction management model also evolved from leader/followers and peers to one or more leaders and delegates.

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, prefetching 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.

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.

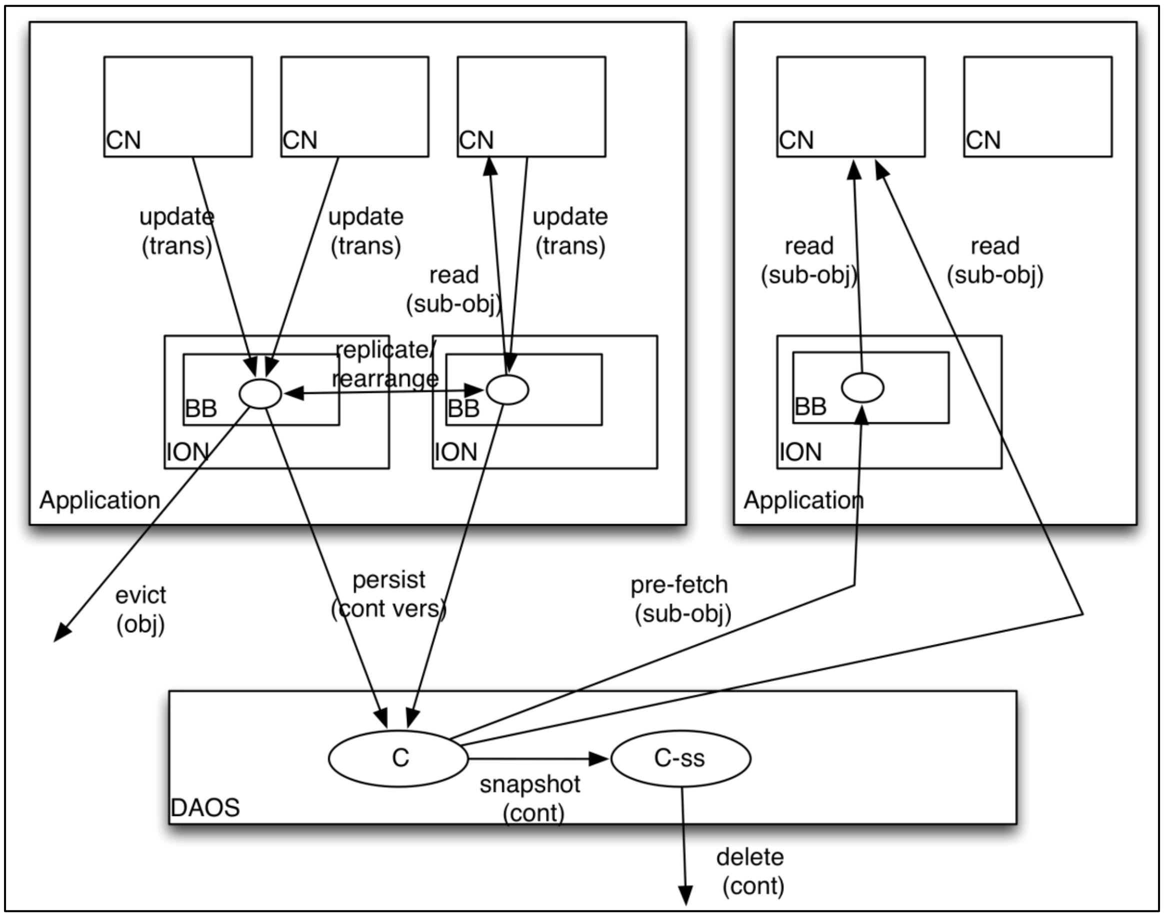


Figure 1: Data movement in the Exascale Fast Forward stack controlled by application requests.

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

4.1.3.1 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***.

A transaction consists of a set of updates to a container. Updates are added to a transaction, not made directly to a container. Updates include additions, deletions, and modifications. When a transaction is committed, the updates in the transaction are applied atomically to the container.

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

1) *start* transaction N
2) add *updates* for container to transaction N
3) *finish* transaction N

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. Transactions can be finished in any order, but they are committed in strict numerical sequence.

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.

4.1.3.1.1 Managing Transactions

Transactions are numbered, and the application is responsible for assigning transaction numbers.[3] Since transactions are committed in strict numerical order, the numbering of transactions affects the order in which updates are applied to the container.

One or more *transaction leaders* can **start** a transaction N by calling *H5TRcreate* and then *H5TRstart()*. If there are multiple transaction leaders for transaction N, each leader that calls *H5TRstart()* for the transaction must also specify the total number of leaders as a parameter to the transaction start call.

If *delegates* (non-leader processes) also participate in the transaction, they also call *H5TRcreate,* but must be notified by one of the transaction leaders that the transaction has been started before they can add updates to transaction N.

Once a transaction has been started, a **dependency** on a lower-numbered *prerequisite* transaction can be registered. This must be done if the dependent transaction would not be able to commit successfully unless the prerequisite transaction committed successfully.

The *transaction id,* returned by *H5TRcreate(),* is passed to the HDF5 APIs that add **updates** to the transaction. The traditional APIs, such as H5Gcreate, have been modified to accept a transaction ID, and renamed with the "_ff" suffix, such as H5Gcreate_ff. The container updates

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

that are added to a given transaction can include adding or deleting H5Datasets, H5CommitedDataTypes, H5Groups, H5Links, H5Maps, and H5Attributes. The updates can also change the contents of existing H5Objects. The updates performed 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. The updates added to a transaction are not visible in the container until the transaction is committed.

Each delegate must notify a leader when it has finished adding updates to transaction N. Each leader **finishes** the transaction when it and the delegates it is responsible for have completed their updates to the transaction. The transaction finish call is *H5TRfinish().* Transactions can be finished in any order.

Finished transaction N will be **committed** (become readable) when all lower-numbered transactions are committed, aborted, or explicitly skipped.

The application does not explicitly commit a transaction, but it indirectly controls when a transaction is committed through its assignment of transaction numbers in "create transaction / start transaction" calls and the order in which transactions are finished, aborted, or explicitly skipped. An *H5TRfinish* operation completes when the transaction is committed (success) or aborted (failure).

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 discarded. Discarded transactions do not block commits of higher-numbered transactions.

4.1.3.1.2 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. The *H5RC* APIs allow an application acquire a read handle on a particular container version and open an associated *read context* that can be used to access the container version until the context is explicitly closed and the handle released. The application must specify a read context when it creates a transaction, so that metadata reads within the transaction are made from a consistent version of the container.

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.

There has been considerable discussion within the team about various naming and numbering conventions related to transactions and versions, 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.

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.

4.1.3.1.3 Persist and Snapshot

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. 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.

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 may be 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.

The application can request a ***snapshot*** of a readable 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.

4.1.3.1.4 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

[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?

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.

4.1.3.1.5 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, say in Transaction 11 the application creates H5Group /A and in Transaction 17 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 17 tries to create /A/B. Even if Transaction 11 was committed, it is possible that one of the operations in Transactions 12-16 could have deleted /A.

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

1) A pessimistic (but guaranteed safe) implementation would require that the container be at Version 16 (i.e., Transactions 11-16 have committed) before asynchronous operations in Transaction 17 can complete. This allows the HDF5 library to verify the state of the container before completing updates in Transaction 17.

2) 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 11 has committed, and know that it did not delete /A in Transactions 12-16, before trying to create /A/B in Transaction 17.

3) 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.

4) 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 a version of the optimistic approach (option 2) for this phase of the project, and to add support for the expression of some dependencies in the HDF5 API.

In the example above, /A, created in Transaction 11, must exist in the file when Transaction 17 commits and adds /A/B. The application can do one of the following:

- Use a read context between 11 and 16 to create Transaction 17.
  - If /A exists in the read context that is specified, the application can create "/A/B" using the pathname specifier for the new datasets.
  - If any version other than 16 is used as the read context, it is still possible that /A might be deleted Transactions 12-15, causing the commit of Transaction 17 to fail.
- Use a read context < 11 for Transaction 17 and add a dependency to Transaction 17 saying that it depends on Transaction 11.
  - If Transaction 11 is aborted, the I/O stack will abort Transaction 17.
  - If this option is chosen, the application can't create "/A/B" using the pathname specifier, because "/A" can't be read from the container version used as the basis for Transaction 17. When the application creates /A in Transaction 11, the object id returned (not the "/A" path) must be used in the call to create B. If the creation of "/A" occurs on a different process than the creation of "/A/B" then the object id for A must undergo a local->global + share with other process + global->local transformation before the other process can use it to create B.

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.

4.1.3.1.6 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 IDs. While not optimized for performance, legacy HDF5 API calls could avoid the issue of asynchronous request IDs and execute synchronously.

Handling the lack of transaction numbers in legacy API calls is more complicated. Legacy code must operate within the constraints of the operating modes of the FastForward I/O stack and could co-exist with new code that does assign and manage transaction numbers and.

Current HDF5 API calls that modify the file can be divided into two types: operations on container *metadata* and operations on *data elements* of H5Datasets. Operations on container metadata must be executed collectively (i.e., by all MPI processes that opened the container), but operations on data elements may be executed either collectively or independently (i.e., by any MPI process that opened the container, without coordination with other processes). Each of these types of operations (collective metadata, collective data element, and independent data element) must be put into the context of the transactions and versions that must be used to interact with FastForward containers and taken into account when solutions for legacy HDF5 applications are designed.

One possible simple solution for legacy HDF5 applications accessing FastForward containers would be to start a new transaction when the legacy application opened the container with write access, using the latest version of the container as the read context for the transaction, and to finish that transaction when the container is closed. However, legacy HDF5 applications expect to be able to read from and interact with changes they make to the container (e.g., reading from newly created objects or reading back data elements written while the file is open), which is not possible when the changes are being written into a transaction, so this is not a completely transparent solution. In addition, if the application is transitioning from using legacy HDF5 APIs to FastForward HDF5 APIs, the transaction started when the file was opened would prevent later transactions (managed explicitly with the FastForward HDF5 APIs) from committing in the container until the application closed the container (and therefore committed the transaction used for legacy operations). Nevertheless, this mode of operation may be useful for applications that understand and can operate within these limitations, and wish to create or modify files in the FastForward I/O environment. We could enable this mode of operation with an HDF5 file access property that was set by the application and used to open the container.

Another, somewhat more flexible, solution to legacy HDF5 applications accessing FastForward containers could be to provide two new 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" could be called with a transaction number that will be assigned to all legacy HDF5 calls executed prior to the "end_legacy_transaction". Note that the legacy application would be required to use the start/end legacy transaction brackets even if it never uses the FastForward APIs, and that the new API calls would need to be collective operations. The application would 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. This solution still has the limitation that an application's legacy HDF5 code would not be able to read data that was created or updated during the current legacy transaction, but application developers may be able to strategicly start and end legacy transactions so that the updates are created in transactions before they need to be read. This solution would also have the advantage that legacy transactions could be committed to the container as desired, allowing explicitly managed transactions from FastForward HDF5 API calls to commit to the container as well.

A third solution to legacy applications accessing FastForward containers would be to put each legacy collective metadata or data element operation into its own transaction and to provide new (collective) API calls that specified a transaction to use for legacy independent data element operations (e.g., "start_ind_data_elem_trans" and "end_ind_data_elem_trans"), since they must be written into a transaction and that transaction number must not collide with the automatically generated transaction numbers for legacy collective operations. Applications that used both legacy HDF5 code and also explicitly managed transaction numbers with FastForward HDF5 API calls would be required to reserve managed transaction number ranges in advance of using them with a new FastForward API call (e.g., "reserve_trans_range"), so that the legacy operations did not use one of those transaction numbers and conflict with a managed transaction. This solution may provide the best backward compability with legacy code, provided it did not use independent data element operations, since each metadata operation would immediately commit in the FastForward container, becoming visible to legacy HDF5 code that wished to access the modified information. Drawbacks to this solution would include the very small transactions created with legacy collective metadata operations, the possibility that the application could attempt to read from independent data element operations before they are committed, the need to reserve explicitly managed transaction numbers and the possibility of transactions that were delayed in committing due to outstanding independent operation transactions or gaps from reserved transactions.

While we could provide one or more of these solutions for legacy HDF5 applications, none are very attractive. The first two would likely be bad from a fault-tolerance and IOD/DAOS container management perspective, and the third would likely be awful from a performance perspective. Since transactions and data migration are key to the Fast Forward stack, allowing applications to run (poorly) with one or more of these proposed solutions does not seem wise, and it is unlikely we will provide a solution for migration of legacy applications in the prototype FastForward project. As we gain experience with the stack, we may see ways to offer intelligent automation that is currently not obvious to us, and we will take advantage of those insights if/when they occur.

4.1.3.2 Reading from HDF5 Files (Containers)

Applications perform reads on a particular version of an HDF5 File (container) in the EFF stack.

Once an application has acquired a read handle and created a read context for a container version, it is guaranteed to see the contents of the container at that version until the context is is closed and the read handle released, even if subsequent transactions are committed to the container. The application must specify a read context when it creates a transaction, so that metadata reads within the transaction are made from a consistent version of the container.

If a container is already open 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.

4.1.3.3 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.

4.1.3.3.1 Moving Data into the Burst Buffers

As discussed previously, container updates are made to 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 prefetched 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 prefetch 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).

Prefetch requests will be made for a specific container version. We are considering allowing multiple prefetch requests to be tagged with a batch identifier, indicating that all of the data in the batch should be prefetched 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.

4.1.3.3.2 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 prefetches 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 prefetch, which can specify sub-objects. Note that sub-objects may reside in the BB (for example, as the result of a prefetch), 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.

4.1.3.3.3 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 prefetch 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.

4.1.4 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.

4.1.5 Optimized Append/Sequence Operations

Optimized dataset append operations were added to the HDF5 code in Quarter 4. In Quarter 5 it seemed that they would be very difficult to support efficiently within the read context / transaction model provided by lower layers of the stack, and they were backed out. At the very *end of Quarter 5, a possible implementation of a modified set of append operations was discussed further. At this point, we are again hopeful that some form of optimized append operations can be added to the EFF stack during the project.*

To support ACG ingest operations and other applications that rapidly append data to HDF5 objects (as well as applications that sequence through objects in a similar fashion), we are extending the HDF5 API with routines that allow append/sequence 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.

As we have continued to refine our support for ACG applications, we have determined that appending new values to variable-length datatype elements may be a better match for ACG application needs. Therefore, we may add to or revise the H5DO* API routines initially delivered in Quarter 4 and described below to focus them on appending values to variable-length datatype elements stored in datasets instead of appending elements to the datasets themselves.

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

4.1.6 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 them.

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 the *User's Guide to FastForward Features in HDF5* for details on new H5M* API routines to create and operate on map objects.

4.1.7 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.

4.1.7.1 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.

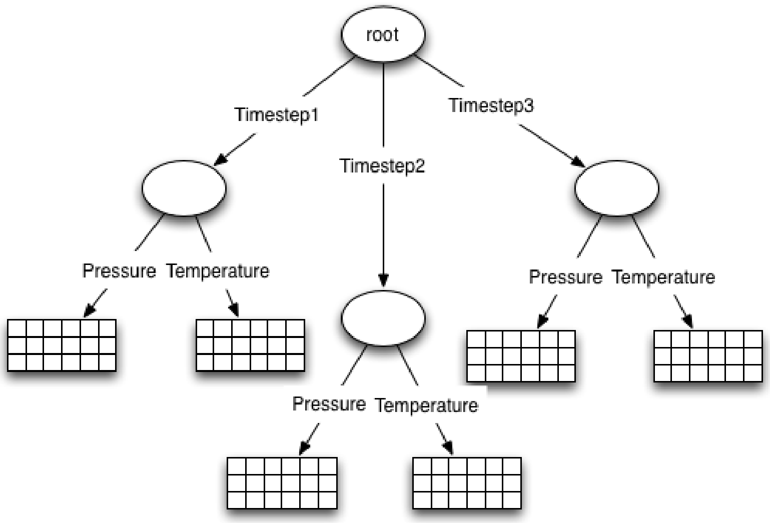
For example, a single call to H5Qcreate could create a query object that would match data elements in any dataset within the container that are equal to the value 17. Another call to H5Qcreate could create a query object that would match link names equal to "Pressure". Calling H5Qcombine with the AND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

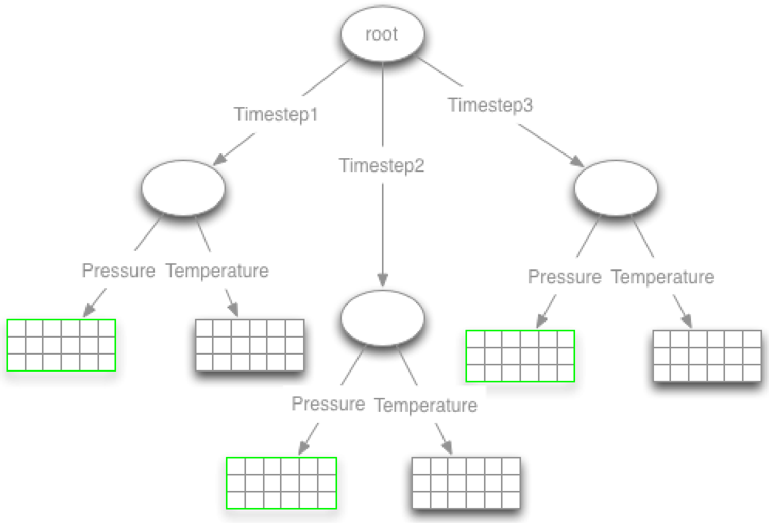
4.1.7.2 View Objects

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.

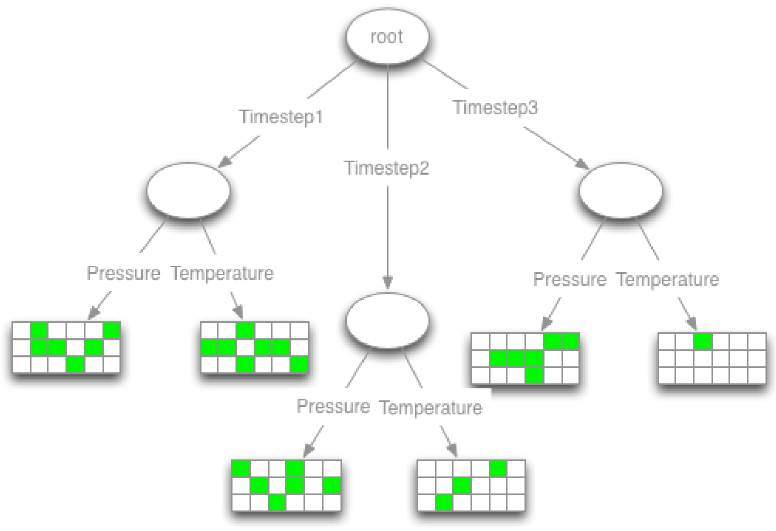
For example, starting with the HDF5 container described in the figure below:



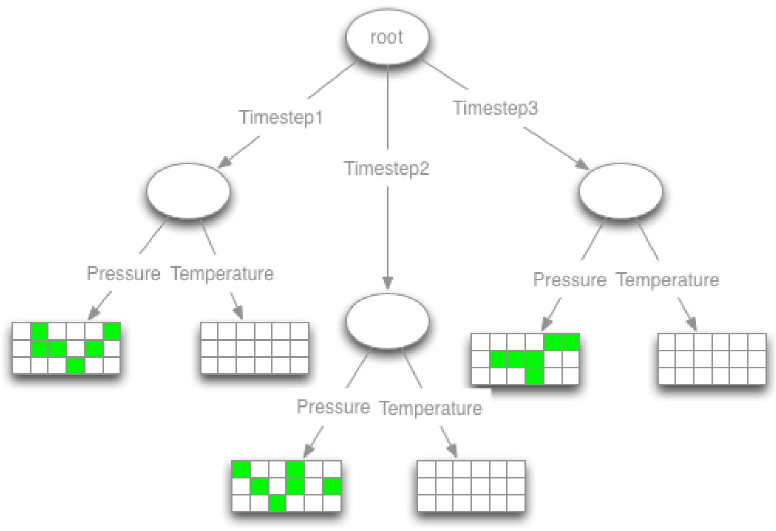
Applying the ‘<link name> = “Pressure”’ query (described above) would result in the view shown below, with the underlying container greyed out and the view highlighted in green:



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

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

elements that match the query with the existing H5Rdereference and H5Rget_region API calls. Note that selections returned from a region reference retain the underlying dataset's dimensionality and coordinates – they are not "flattened" into a 1-D series of elements. The selection returned from a region reference can also be applied to a different dataset in the container, allowing a query on pressure values to be used to extract temperature values, for example.

4.1.7.3 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 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.

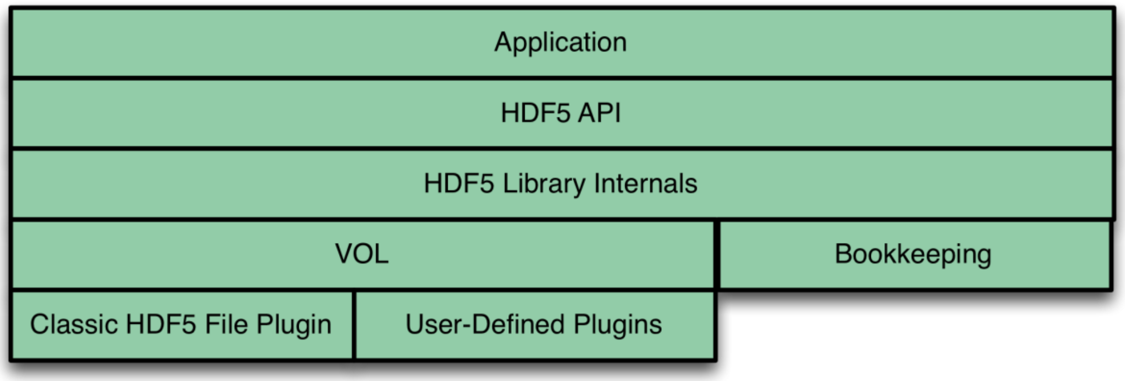
See below, in the API and Protocol Additions and Changes section, for details on the new H5Q*, H5V* and H5X* API routines used for query, view and index operations, respectively.

4.2 Architectural Changes to the HDF5 library

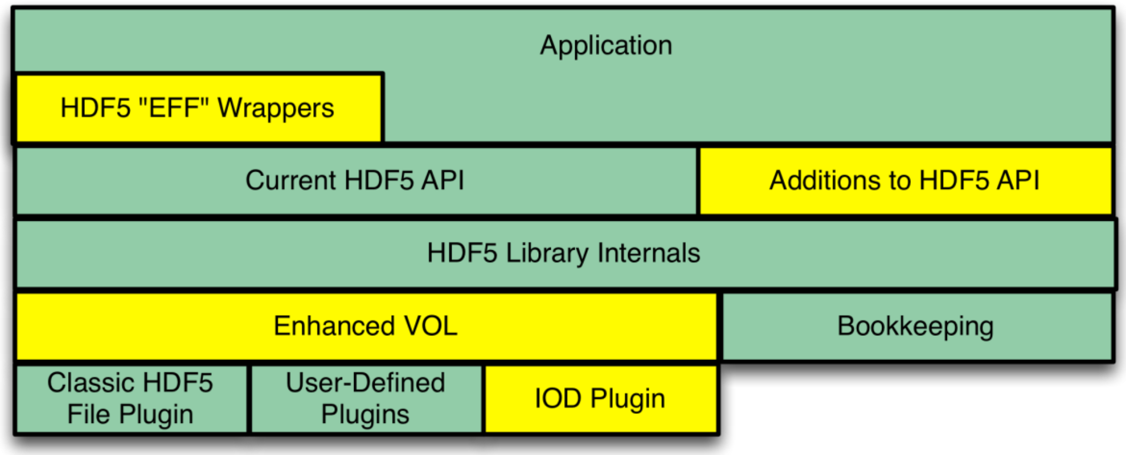
The architecture of the core HDF5 library is largely unaffected by the changes described in this document. The majority of the capabilities added to the HDF5 API are handled by a wrapper layer above the main HDF5 library, and a small number of additions to the main API routines (details of these API 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.

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

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:



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.

4.3 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. Please refer to the document "*HDF5 Data in IOD Containers Layout Specification*" for details.

5 API and Protocol Additions and Changes

There are two kinds of changes to the HDF5 library API for the Exascale FastForward project: (1) generic changes to existing API routines to accommodate new capabilities, such as asynchronous I/O and transactions, and (2) additions to the HDF5 API that support new features. Both of these types of changes to the HDF5 API are described below. The reference manual pages for all of the modified and many of the new routines can be found in the *User's Guide to FastForward Features in HDF5*.

5.1 Generic changes to HDF5 API routines

Many HDF5 API routines operate on HDF5 file objects and need to be extended in similar ways. The generic modifications are described in this section. HDF5 API routines that are extended in this manner have the suffix "_ff"[8] added to the API routine's name.

Existing HDF5 routines that operate on HDF5 file objects are extended by adding one or more parameters:

1. A *read context id* for routines that read from the HDF5 file.
    - The read context id indicates what version of the container (the HDF5 file) will be read.
2. A *transaction id* for routines that update the HDF5 file.
    - The transaction id indicates both the transaction number and the read context for the update operation.
3. An *event stack id* for routines that can execute asynchronously.
    - The event stack identifier indicates where the event associated with the asynchronous operation is pushed. The event stack provides a mechanism for checking the operation's completion status at a later time.
    - Passing H5_EVENT_STACK_NULL for the event stack identifier indicates that the operation should be executed synchronously.

The following pseudo-function prototypes demonstrate the method for these changes to HDF5 API routines:

Current routine that performs update:

```
<return type> H5Xexisting_update_routine(<current parameters>);
```

Extended routine that performs update:

```
<return type> H5Xexisting_update_routine_ff(<current parameters>,
                                            hid_t transaction_id,
                                            hid_t event_stack_id);
```

Current routine that performs read:

```
<return type> H5Xexisting_read_routine(<current parameters>);
```

Extended routine that performs read:

```
<return type> H5Xexisting_read_routine_ff(<current parameters>,
                                          hid_t read_context_id,
                                          hid_t event_stack_id);
```

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

As a concrete example, the following prototypes show the changes to H5Gcreate, the group creation API routine for HDF5[9] — a routine that performs updates:

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(hid_t loc_id, const char *name, hid_t lcpl_id,
                hid_t gcpl_id, hid_t gapl_id,
                hid_t transaction_id, hid_t event_stack_id );
```

Note that the error value returned when a routine is executed asynchronously only indicates the status of the routine up to the point when it is scheduled for later completion. The new asynchronous test and wait routines (H5EStest, H5ESwait, H5EStest_all, and H5ESwait_all) return the error status for the "second half" of the routine's execution.

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

A note on the design of the API changes: We considered alternate forms of passing the transaction, read context, and event stack information in to and out of the HDF5 API routines, such as using HDF5 properties in one of the property lists passed to API. However, using HDF5 properties had some drawbacks. In particular, several of the API routines did not have property list parameters and therefore would have to be extended with more parameters anyway, and setting the additional information in properties can sometimes obscure the fact that an operation's behavior has been changed.

5.2 Additions to the HDF5 API

This section presents the new routines that are being added to the HDF5 API to support FastForward capabilities in the library. Many of the new routines now have reference manual entries in the *User's Guide to FastForward Features in HDF5*. For those routines, only the routine names and brief descriptions are retained in this document.

5.2.1 Event Stack Operations

H5EScreate() – Create a new (empty) event stack object.

H5ESwait_all() – Wait for all events in an event stack to complete.

H5EStest_all() – Test to see if all events in an event stack have completed.

H5EScancel_all() – Cancel all events in an event stack that have not yet completed.

[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

H5ESwait – Wait for a particular event in an event stack to complete.

H5EStest – Test to see if a particular event in an event stack has completed.

H5EScancel – Cancel a particular event in an event stack.

H5ESget_count() – Retrieve the number of events in an event stack.

H5ESget_event_info() – Retrieve information about the events in an event stack.

H5ESclear() – Clear all events from an event stack, provided no events are in progress.

H5ESclose() – Close an event stack, provided no events are in progress.

5.2.2 End-to-End Integrity

H5Pset_dxpl_checksum() – Specifies a user-supplied checksum for a write data transfer.

H5Pset_dxpl_checksum_ptr() – Specifies a memory location to receive the checksum from a read data transfer.

H5Pset_dxpl_inject_corruption() – Specifies that data should be corrupted prior to transfer. [for testing purposes]

H5Pset_metadata_integrity_scope() – Specifies the scope of checksum generation and verification for metadata transfer. Per-container.

H5Pset_rawdata_integrity_scope() – Specifies the scope of checksum generation and verification for raw data transfer. Per-transfer.

H5checksum() – Generate a checksum.

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

H5TRcreate() – Create a transaction associated with a specified container, read context, and number.

H5TRstart() – Start a created transaction.

H5TRset_dependency() – Register the dependency of a transaction on a lower-numbered transaction.

H5TRfinish() - Finish a transaction that was started with H5TRstart.

H5TRclose() – Close the specified transaction.

H5TRskip() – Skip one or more transaction numbers for a given container.

H5TRabort() – Abort a transaction that was started with H5TRstart.

H5RCacquire() – Acquire a read handle for a container at a given version and create a read context associated with the containerversion.

H5RCacquire_wait() – Acquire a read handle for a container at a given version and create a read context associated with the container version, waiting if the specified version is not yet committed. [Designed in Q5 in response to user feedback, but not yet implemented]

H5RCcreate() – Create a read context associated with a container and version.

H5RCget_version() – Retrieve the container version associated with a read context.

H5RCpersist() – Copy data for a container from IOD to DAOS.

H5RCsnapshot() – Make a snapshot of a container on DAOS.

H5RCclose() – Close a read context.

H5RCrelease() – Close a read context and release a read handle for the associated container version.

H5Oevict() – Evict the data for an object from the burst buffer

H5Oget_token() – Get a token for a just-created object to share with other processes so they can open the object in the same transaction.

H5Oopen_by_token() – Open an object created by another process in the same transaction

5.2.4 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 indicates special properties of writing data to an object. Possible values 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 will be expanded to map objects and possibly to other objects in the future.

5.2.5 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.

5.2.6 File Objects/Properties

H5Pset_fapl_iod() – Use the IOD VOL plugin for container operations.

5.2.7 Dataset Objects – Optimized APIs

These descriptions reflect the function of the routines delivered in Q4. They have not been updated to reflect the latest thinking as of the end of Q5, and will be updated further in future quarters.

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

H5DOappend_ff() – Perform an optimized append operation on a dataset, possibly asynchronously.

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

H5DOsequence_ff() – Perform an optimized stream-oriented read operation on a dataset, possibly asynchronously.

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

H5DOset_ff() – Write a single element to a dataset, possibly asynchronously.

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

H5DOget_ff() – Read a single element from a dataset, possibly asynchronously.

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

5.2.8 Map Objects

H5Mcreate_ff() – Create a new map object

H5Mopen() – Open an existing map object

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

H5Mget() – Retrieve a value from a map object

H5Mget_types() – Retrieve the datatypes forthe keys and values of a map object

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

H5Mexists_ff() – Check if a key exists in a map object

H5Miterate_ff() – Iterate over the key/value pairs in a map object

H5Mdelete_ff() – Delete a key/value pair in a map object

H5Mclose_ff() – Close a map object

5.2.9 Query Objects

<u>H5Qcreate()</u> – 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:

<u>Query Type</u> (`H5Q_query_type_t`)	**<u>Match Conditions</u>** (`H5Q_match_op_t`)	**<u>Varargs parameters</u>**
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.

5.2.10 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_ff.

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.

5.2.11 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[11]:

- 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

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:

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

```
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.

6 Open Issues

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 updated design or implementation in future quarters.

7 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 A 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[12]
Attribute Name	Name of attribute on an object	Y	Y	Attribute reference[13]
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

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

[13] Under development.