Date: June 30, 2014 Delivered as part of Milestones 8.1, 8.2, 8.5	*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

NOTICE: THIS MANUSCRIPT HAS BEEN AUTHORED BY THE HDF GROUP UNDER THE INTEL SUBCONTRACT WITH LAWRENCE LIVERMORE NATIONAL SECURITY, LLC WHO IS THE OPERATOR AND MANAGER OF LAWRENCE LIVERMORE NATIONAL LABORATORY UNDER CONTRACT NO. DE-AC52-07NA27344 WITH THE U.S. DEPARTMENT OF ENERGY. THE UNITED STATES GOVERNMENT RETAINS AND THE PUBLISHER, BY ACCEPTING THE ARTICLE OF PUBLICATION, ACKNOWLEDGES THAT THE UNITED STATES GOVERNMENT RETAINS A NON-EXCLUSIVE, PAID-UP, IRREVOCABLE, WORLD-WIDE LICENSE TO PUBLISH OR REPRODUCE THE PUBLISHED FORM OF THIS MANUSCRIPT, OR ALLOW OTHERS TO DO SO, FOR UNITED STATES GOVERNMENT PURPOSES. THE VIEWS AND OPINIONS OF AUTHORS EXPRESSED HEREIN DO NOT NECESSARILY REFLECT THOSE OF THE UNITED STATES GOVERNMENT OR LAWRENCE LIVERMORE NATIONAL SECURITY, LLC.

Copyright 2014, The HDF Group. All Rights Reserved.
B599860-SS

Table of Contents

Revision History iii
1 Introduction 1
2 Definitions 1
3 Changes from Solution Architecture 1
4 Specification 2
4.1 New HDF5 Library Capabilities 2
4.1.1 Asynchronous I/O and Event Stack Objects 2
4.1.2 End-to-End Data Integrity 4
4.1.3 Transactions, Container Versions, and Data Movement in the I/O Stack 5
4.1.3.1 Transactions and Writing to HDF5 Files (Containers) 6
4.1.3.1.1 Managing Transactions 7
4.1.3.1.2 Container Versions 8
4.1.3.1.3 Persist and Snapshot 8
4.1.3.1.4 General Discussion 9
4.1.3.1.5 Design Decisions 9
4.1.3.1.6 Support for Legacy Applications 11
4.1.3.2 Reading from HDF5 Files (Containers) 12
4.1.3.3 Burst Buffer Space Management 13
4.1.4 Data Layout Properties 13
4.1.5 Optimized Append/Sequence Operations 13
4.1.6 Map Objects 14
4.1.7 Data Analysis Extensions (Query, View and Index Operations) 14
4.1.7.1 Query Objects 15
4.1.7.2 View Objects 15
4.1.7.3 Index Objects 18
4.1.7.3.1 Index Plugins 18
4.1.7.3.2 Stakeholder Feedback and Possible Future Extensions 19
4.1.7.3.3 Index Plugin Limitations 20
4.1.7.4 Analysis Shipping Operation 20
4.2 Architectural Changes to the HDF5 Library 21
4.3 Storing HDF5 Objects in IOD Containers 22
5 API and Protocol Additions and Changes 23
5.1 Generic changes to HDF5 API routines 23
5.2 Additions to the HDF5 API 24
5.2.1 Dataset Objects – Optimized APIs (designed but not fully implemented) 24
5.2.1.1 H5DOappend_ff 24
5.2.1.2 H5DOget_ff 26
5.2.1.3 H5DOset_ff 27
5.2.1.4 H5DOsequence_ff 28
5.2.1.5 H5Pset_dcpl_append_only 29
*5.2.2 Analysis Shipping API ..... **Error! Bookmark not defined.***
6 Risks & Unknowns 29

Use or disclosure of data contained on this sheet is subject to the restriction on the title page of this document.
Copyright © The HDF Group, 2014. All rights reserved

Appendix A Aspects of the HDF5 Data Model 30

Use or disclosure of data contained on this sheet is subject to the restriction on the title page of this document.
Copyright © The HDF Group, 2014. All rights reserved

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

Use or disclosure of data contained on this sheet is subject to the restriction on the title page of this document.
Copyright © The HDF Group, 2014. All rights reserved

		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
December 16, 2013	24.0	• Update H5Q routines with H5Qapply • Add analysis shipping routine, H5ASexecute • Update analysis shipping description	Mohamad Chaarawi, The HDF Group
December 19, 2013	25.0	• Update analysis shipping section	Jerome Soumagne, The HDF Group
December 23, 2013	26.0	• Final edits, update date / disclaimers / TOC; • Footer date on main pages reconciled without version change 1/4/14.	Quincey Koziol, Ruth Aydt, The HDF Group

Use or disclosure of data contained on this sheet is subject to the restriction on the title page of this document.
Copyright © The HDF Group, 2014. All rights reserved

February 7, 2014	27.1	• Update data markings / copyright year	Ruth Aydt, The HDF Group
February 13, 2014	27.2	• Update query/view/index APIs	Quincey Koziol, The HDF Group
February 25, 2014	27.3	• Revise text related to BB space management, referencing the separate document on this topic.	Ruth Aydt, The HDF Group
March 28, 2014	27.4	• Update index design • Update index/view APIs	Jerome Soumagne, The HDF Group
March 30, 2014	27.5	• Overall edit pass	Ruth Aydt, The HDF Group
March 31, 2014	27.6	• Indexing sections editing pass and added index plugin design appendix • Deliver to DOE as part of Milestones 7.1, 7.2, 7.3	Quincey Koziol, The HDF Group
April 10, 2014	28.0	• Base version for Quarter 8 deliverables	Ruth Aydt, The HDF Group
April 28, 2014	28.1	• Expand query and view APIs for metadata additions, plus new H5Dquery API routine	Quincey Koziol, The HDF Group
May 6, 2014	28.2	• Add note about evict granularity and named snapshots to Open Issues. • Update text related to evict & related APIs	Ruth Aydt, The HDF Group
May 21, 2014	28.3	• Update H5Q API	Jerome Soumagne, The HDF Group
May 22, 2014	28.4	• Update Open Issues with notes about RCacquire. • Add footnote about HDF5 using a transaction on file close.	Ruth Aydt, The HDF Group
June 23, 2014	28.5	• Revise text about optimized appends, for which there is an initial design but no implementation in prototype. Include designed APIs in this document, previously cut from UG. • Change future tense to past tense. • Delete enumerated APIs that were added for the project, referring reader to the User Guide. • Delete Open Issues – these have been covered in the final report.	Ruth Aydt, The HDF Group
June 27, 2014	28.6	• Misc. light edits, mostly in response to Ruth's comments • Included Mohamad's info for data layout properties • Updated query/view/index text • Updated analysis shipping text • Added some additional text to the Risks & Unknowns section	Quincey Koziol, The HDF Group
June 30, 2014	28.7	• Moved Index Plugin Appendix to the UG • Move H5X interface routines to the UG	Quincey Koziol, The HDF Group
June 30, 2014	28.8	• Move View interface routines to the UG • Move some Query interface routines to the	Quincey Koziol, The HDF Group

Use or disclosure of data contained on this sheet is subject to the restriction on the title page of this document.
Copyright © The HDF Group, 2014. All rights reserved

		UG	
June 30, 2014	28.9	• Move rest of interface routines to the UG • Revised asynchronous operations text to reflect that cancel operations for event stacks are actually implemented. • Final polishing pass • Delivered to DOE stakeholders as part of Milestone 8.1/8.2/8.5 demonstration output.	Quincey Koziol, The HDF Group

Use or disclosure of data contained on this sheet is subject to the restriction on the title page of this document.
Copyright © The HDF Group, 2014. All rights reserved

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, a new Map object, and data analysis extensions for indexing and querying HDF5 containers and shipping analysis scripts to execute where the data resides. All changes for these capabilities were combined into one design document for easier tracking; furthermore, because many of the features affect the same HDF5 API routines, they are easier to understand in combination.

Please refer to the *User's Guide to FastForward Features in HDF5* for details on H5* API routines that were created or modified to support the features described in this document.

Some technical lessons learned during the project are mentioned in this documention, while the majority are discussed in the final report.

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

In discussions with the ACG team, it was determined the addition of a pointer or other dynamic datatype to HDF5 was not the best approach to meet their needs. Instead, adding support for optimized appends and a new Map object to HDF5's data model would have a greater utility to their use case. This document reflects that divergence from the Solution Architecture document. The Map object has been added, and initial support for optimized appends to blobs was added at the IOD level and designed at the HDF5 level. Unfortunately, as described in the final report, some unexpected challenges arose in conjunction with the implementation of optimized appends and transactional I/O, and that feature was not completed during the prototype. The initial design of optimized appends at the HDF5 level is included in this document for reference and possible future implementation.

Use or disclosure of data contained on this sheet is subject to the restriction on the title page of this document.
Copyright © The HDF Group, 2014. All rights reserved

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 (initial design, not implemented)
- Map Objects
- Query, view and indexing extensions
- Data Analysis Extensions

4.1.1 Asynchronous I/O and Event Stack Objects

Support for asynchronous I/O in HDF5 was 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 was not implemented and delivered until Quarter 6. Although the design for retrieving detailed error information about operations that fail has been designed and documented here, implementation of this more advanced feature wasn't necessary at this stage and has been deferred.

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

Use or disclosure of data contained on this sheet is subject to the restriction on the title page of this document.
Copyright © The HDF Group, 2014. All rights reserved

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 executed 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 (although implementation of this feature has been deferred currently). 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 by 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 Asynchronous eXecution Engine (AXE)[1], 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.

[1] See the *AXE Design Document* for details on AXE.

Use or disclosure of data contained on this sheet is subject to the restriction on the title page of this document.
Copyright © The HDF Group, 2014. All rights reserved

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[2] 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 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 checksums 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

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

Use or disclosure of data contained on this sheet is subject to the restriction on the title page of this document.
Copyright © The HDF Group, 2014. All rights reserved

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, persisting data from the BB to DAOS storage, prefetching data from DAOS storage into the BB, reading data from the BB or DAOS storage into the application memory, and evicting data from the BB.

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.[3] 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.

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

Use or disclosure of data contained on this sheet is subject to the restriction on the title page of this document.
Copyright © The HDF Group, 2014. All rights reserved

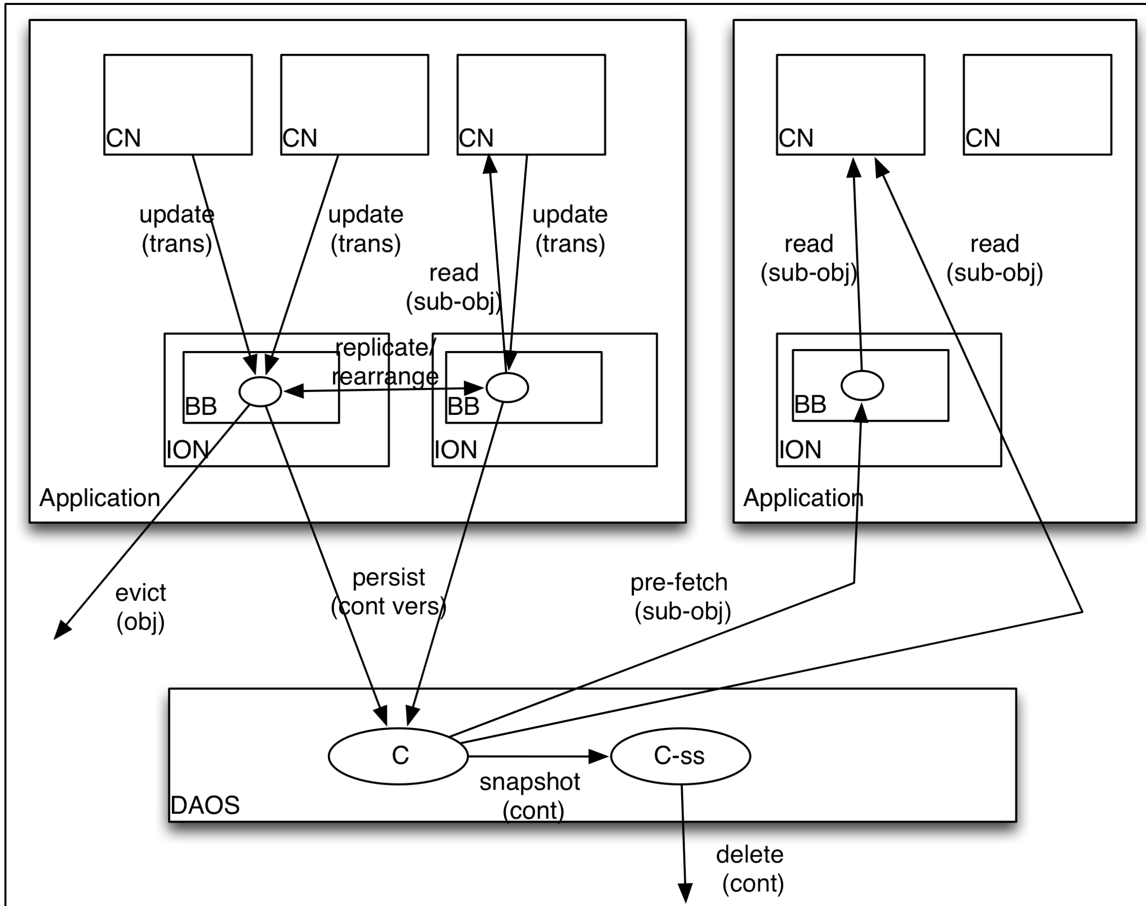


Figure 1: Data movement in the Exascale Fast Forward stack controlled by application requests.

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.

Use or disclosure of data contained on this sheet is subject to the restriction on the title page of this document.
Copyright © The HDF Group, 2014. All rights reserved

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 while a container is open for write.[4][5] 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 that are added to a given transaction can include adding or deleting H5Datasets, H5NamedDataTypes, 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

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

[5] HDF5 commits one additional transaction when the container is closed to save file metadata that helps improve performance when the container is re-opened. This file metadata is "nice but not necessary", so if a crash occurs and the transaction with the metadata has not been committed, the container is still accessible.

Use or disclosure of data contained on this sheet is subject to the restriction on the title page of this document.
Copyright © The HDF Group, 2014. All rights reserved

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 the BB to be copied to DAOS and atomically committed to the persistent storage. 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[6] and copied to DAOS. Data (and metadata) for persisted container versions is not automatically removed from the BB. The application must explicitly *evict* data from the BB – this is discussed in more detail in a later section on Burst Buffer Space Management. Note that

[6] 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.

Use or disclosure of data contained on this sheet is subject to the restriction on the title page of this document.
Copyright © The HDF Group, 2014. All rights reserved

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 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 offered some opportunity to assess the frequencies that can be supported with good performance, but performance testing was limited due to time contraints.

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 allow the application to start, abort, finish, and skip transactions, and to persist and snapshot container versions. Routines were also added to inquire about transaction and container status. Existing API routines were 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.

Use or disclosure of data contained on this sheet is subject to the restriction on the title page of this document.
Copyright © The HDF Group, 2014. All rights reserved

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

Use or disclosure of data contained on this sheet is subject to the restriction on the title page of this document.
Copyright © The HDF Group, 2014. All rights reserved

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

Use or disclosure of data contained on this sheet is subject to the restriction on the title page of this document.
Copyright © The HDF Group, 2014. All rights reserved

“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

Use or disclosure of data contained on this sheet is subject to the restriction on the title page of this document.
Copyright © The HDF Group, 2014. All rights reserved

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.

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.

Please refer to the separate document, *Burst Buffer Space Management – Prototype and Production,* and to the final report, for a discussion of the prefetch and evict capabilities that will be delivered as part of the EFF prototype project. Some additional features that have been identified as potentially beneficial in a production version of the EFF stack are also introduced.

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 in the UG, in the H5Pset_layout* functions.

The new properties control how HDF5 dataset objects (as IOD Arrays) will be stored on DAOS after a call to persist those objects. The layout properties implemented are:

- The physical dimensionality storage of a dataset object. The user can indicate whether a dataset should be physically stored on DAOS in row-major (default) or column-major order.
- The stripe count or the number of DAOS shards used to store the dataset raw data.
- The stripe size or the number of dataset elements to store contiguously on each DAOS shard, before moving to the next one.

Layout properties for storage on the Burst Buffer were not implemented, as they require HDF5 or the user to maintain a tag which is required to access the object with the modified layout.

4.1.5 Optimized Append/Sequence Operations

Optimized dataset append operations were added to the HDF5 design and code in Quarter 4. In Quarter 5 it seemed that they would be very difficult to support efficiently within the read

Use or disclosure of data contained on this sheet is subject to the restriction on the title page of this document.
Copyright © The HDF Group, 2014. All rights reserved

context / transaction model adopted by the EFF project, 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, and IOD later added support for appendable blob objects. In the end, these features, intended to help ACG, were not implemented on the EFF stack, but we retain this section and the design ideas for possible future implementation. The final report discusses the implementation challenges in more detail.

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 planned to extend 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 planned to add simple routines for quickly setting and retrieving single elements in an HDF5 dataset.

As we continued to refine our support for ACG applications, we determined that appending new values to variable-length datatype elements may be a better match for ACG application needs. Therefore, we hoped to 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. In the end, none of the H5DO* APIs were fully integrated into the stack.

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 added a new Map object to the HDF5 data model and API.

Map objects in HDF5 are similar to a typical "map" data structure in computer science. HDF5 maps set/get a value in the object, according to the key provided, with a 1-1 mapping of keys to values. All keys for a given 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.

4.1.7 Data Analysis Extensions (Query, View and Index Operations)

Support for data analysis operations on HDF5 containers were implemented via:

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

Use or disclosure of data contained on this sheet is subject to the restriction on the title page of this document.
Copyright © The HDF Group, 2014. All rights reserved

- 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
- New “analysis shipping” API routines for sending a query along with python scripts to VOL servers to analyze data in a container

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 in HDF5 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”.

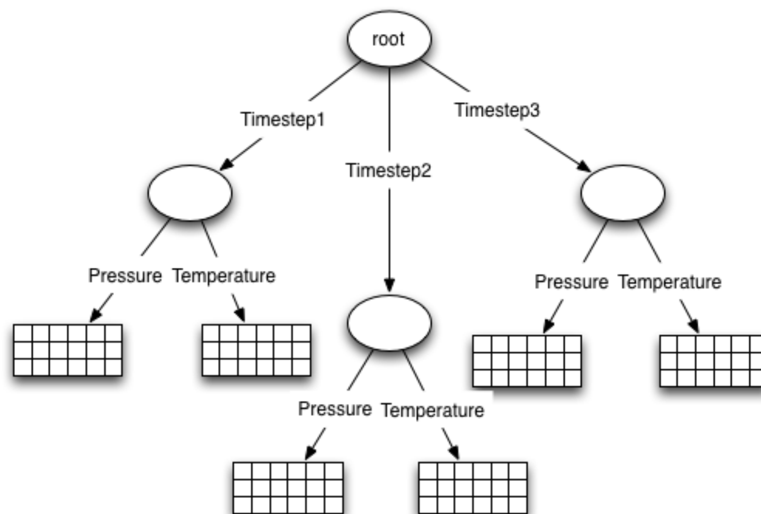
Creating the data analysis extensions to HDF5 using a “programmatic interface” for defining queries avoids defining a text-based query language as a core component of the data analysis interface, and is more in keeping with the design and level of abstraction of the HDF5 API. The HDF5 data model is more 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-based query language (or GUI) could certainly be built on top of the query API defined here to provide a more user-friendly (as opposed to “developer-friendly”) query syntax like “Pressure = 17”. However, we regard this as out-of-scope for the current project.

4.1.7.2 View Objects

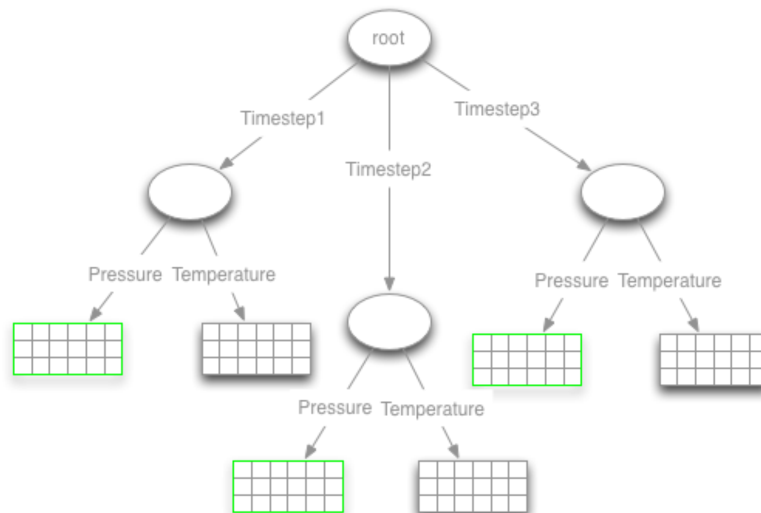
Applying a query to an HDF5 container creates an HDF5 view object. HDF5 view objects are runtime, in-memory objects (i.e., not stored in a container) that consist of read-only references into the contents of the HDF5 container that the query was applied to. View objects are created with H5Vcreate, which 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 within a view can be retrieved by further API calls.

Use or disclosure of data contained on this sheet is subject to the restriction on the title page of this document.
Copyright © The HDF Group, 2014. All rights reserved

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

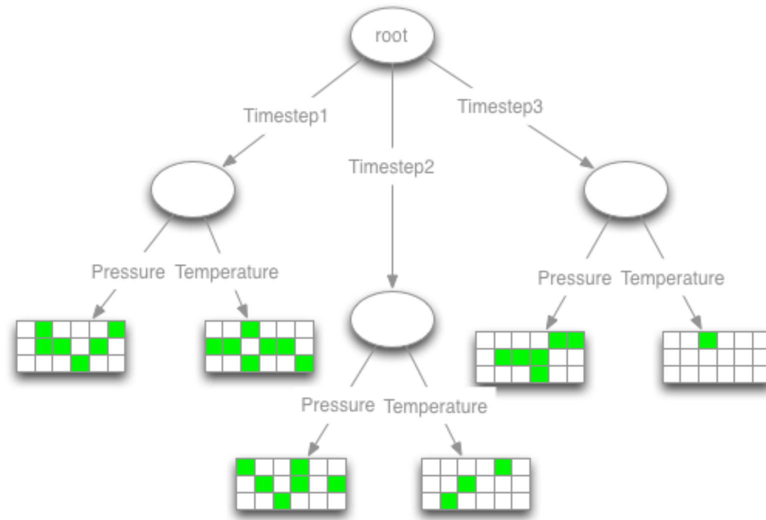


Applying the '<link name> = "Pressure"' query (described above) would result in the view shown below, with the underlying container greyed out and the view highlighted in green:

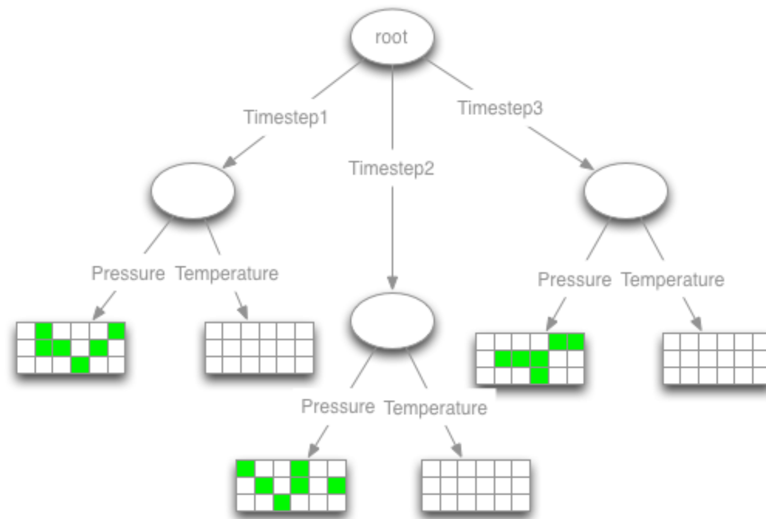


Use or disclosure of data contained on this sheet is subject to the restriction on the title page of this document.
Copyright © The HDF Group, 2014. All rights reserved

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[7] 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

[7] Attribute references were initially targeted for delivery in Q6 of the FastForward project, but were rescheduled to Q8, along with other metadata query operations.

Use or disclosure of data contained on this sheet is subject to the restriction on the title page of this document.
Copyright © The HDF Group, 2014. All rights reserved

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

4.1.7.3 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.

For example, if the '<link name> = "Pressure" AND <data element> = 17' query (described above) is going to be frequently executed on the container, indices could be created in that container which would speed up the creation of views when querying for link names and for data element values. Indices created for accelerating the '<link name> = "Pressure"' or '<data element> = 17' queries would also improve view creation for the more complex '<link name> = "Pressure" AND <data element> = 17' query.

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

The indexing API works in conjunction with the view API. When an H5Vcreate call is made for a group or dataset, an index attached to any dataset queried for element value ranges will be used to speed up the query process and returns a dataspace selection to the library for later use, for example by analysis shipping tasks.

There are different techniques for creating data element indices, and the most efficient method will vary depending on the type of the data that is to be indexed, its layout, etc. We therefore define a new interface for the HDF5 library that uses a plugin mechanism.

4.1.7.3.1 Index Plugins

A new HDF5 interface has been defined for adding third-party indexing plugins, such as FastBit[8], Alacrity[9], etc., and will be demonstrated in Quarters 7 and 8 of the project. The interface provides indexing plugins with efficient access to the contents of the container for both the creation and the maintenance of indices. In addition, the interface allows third-party plugins to create private data structures within the container for storing the contents of the index.

Index objects are stored in the HDF5 container that they apply to, but are not visible in the container's group hierarchy[10]. Instead, index objects are part of the metadata for the file itself.

---

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

[9] ALACRITY: Analytics-Driven Lossless Data Compression for Rapid In-Situ Indexing, Storing, and Querying, http://link.springer.com/chapter/10.1007%2F978-3-642-41221-9_4

[10] Plugin developers, note that the HDF5 library's existing anonymous dataset and group creation calls can be used to create objects in HDF5 files that are not visible in the container's group hierarchy.

Use or disclosure of data contained on this sheet is subject to the restriction on the title page of this document.
Copyright © The HDF Group, 2014. All rights reserved

New index objects are created by passing an H5Container to be indexed and index plugin ID to the H5Xcreate call (see the H5Xcreate API call description in the User Guide for details).

When using the IOD VOL plugin in the EFF context, index information (such as plugin id and index metadata) is stored/retrieved in the object's metadata KV object. This extra information is stored at index creation time, and when the user later calls H5Dopen, the plugin *open* callback will retrieve this stored information and make use of the corresponding index plugin for all subsequent operations. Similarly, calling H5Dclose will call the plugin index *close* callback and close the objects used to store the index data.

When a call to H5Dwrite is made, the index plugin *pre_update* and *post_update* callbacks will be triggered, allowing efficient index update by first telling the index plugin the region that is going to be updated with new data, and then realizing the actual index update, after the dataset write has completed. This allows various optimizations to be made, depending on the data selection passed and the index plugin used. For example, a plugin could store the region and defer the actual index update until the dataset is closed, hence saving repeated index computation/update calls.

When a call to H5Vcreate is made, the index plugin *query* callback will be invoked to create a selection of elements in the dataset that match the query parameters. Applications can also use the new H5Dquery routine to directly execute a query on a dataset (accelerated by any index defined on the dataset), retrieving the selection within that dataset that matches the query.

See the User's Guide to FastForward Features in HDF5 for details on the index plugin interface and operation.

4.1.7.3.2 Stakeholder Feedback and Possible Future Extensions

During the Milestone 7.1 Index Plugin API Demonstration, the DOE stakeholders raised a number of good questions that are beyond the scope of what will be implemented in the prototype project, but that are worth considering for future work. They are captured here, with some preliminary thoughts on how they might be addressed.

- How could an index be built in parallel for a dataset that already exists?
    - Possibly the analysis shipping framework could help with this. We could prefetch the dataset onto the IONs, run a parallel index build operation on the data in the burst buffer (using the analysis shipping framework), then persist the data back to DAOS.
- How to handle index updates when the specified index plugin is not available? (In traditional databases, stored procedures are saved with the data and therefore available at any time, but that is not the case here)
    - We can possibly track the elements of the dataset that are updated when the plugin is not available, then add a new "refresh" callback to the plugin callbacks, that would get invoked with the selection of the elements that were modified and need to be indexed once the plugin is available again.
- How to handle index queries when the specified index plugin is not available?
    - The HDF5 library will ship with a default "brute force" index plugin, which will be invoked when there is no index for a dataset, or when the index plugin is not available.

Use or disclosure of data contained on this sheet is subject to the restriction on the title page of this document.
Copyright © The HDF Group, 2014. All rights reserved

- Another possible future optimization would be to build the indices on the server rather than on the clients. This was deferred due to the complexity of re-entering the server code from index plugins, but is possible now that a co-resident version of the FastForward I/O stack is available.

4.1.7.3.3 Index Plugin Limitations

Several things limit use of indices in the EFF I/O stack: FastBit and Alacrity don't support incremental updates, an index is a shared resource for a dataset, and the strong transactional model in the EFF I/O stack. Taken together, these conspire to put limits on application updates to datasets with indices. The strong transactional model restricts updates to an index in a transaction from being visible to other processes (or even to the process which made the changes, later in the transaction). This limits updates on existing index values (and the dataset they are attached to) to single writes by one CN within each transaction. Additionally, because FastBit and Alacrity don't allow incremental updates to an index, each modification to an existing index forces the index to be entirely rebuilt.

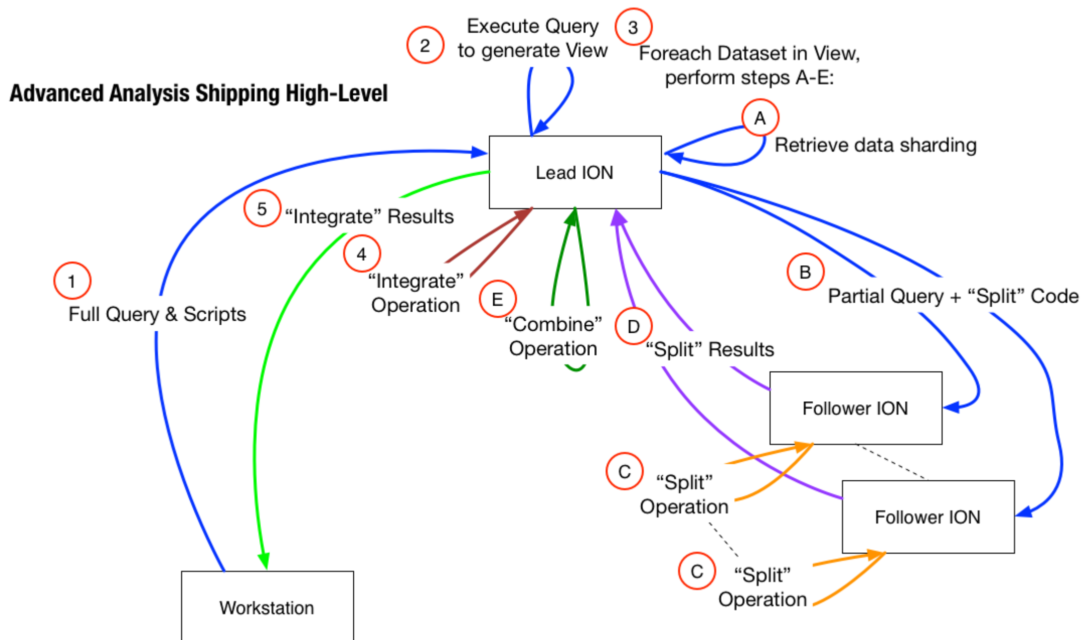
The limitation due to the transactional model could be mitigated by relaxing the model to allow reads from an uncommitted transaction, or by caching index information on the IONs, although the latter implementation would likely be unprotected from faults. The limitation in FastBit and Alacrity will need to be addressed in the base packages' implementation, so that they can make incremental updates to their index information.

4.1.7.4 Analysis Shipping Operation

Analysis shipping is the capstone of the data analysis framework and provides the user with a means to ship an analysis request directly to the IONs or storage server nodes, thereby performing analysis operations as close as possible to the data, and minimizing data movement costs. To operate on the data, the query object, described in the earlier section, contains selection criteria for selecting the data in the object that is stored in a specified container and the user provides Python scripts that perform operations on the selected data.

An analysis request is composed of three stages of operations: a *split* operation, which is executed on each dataset in parallel and locally to the data selected by the query to generate intermediate results, a *combine* operation, which gathers data generated by the *split* operation and optionally performs an operation on the intermediate results and an *integrate* operation which gathers the results of all the combine operations optionally performs a final operation over all of them. The server that receives the analysis request will distribute a *split* operation to all the nodes that own a piece of the data, depending on the layout of the object in the underlying storage, Those nodes then read the object's data and apply the query to generate a subset of the data that satisfies the query conditions. The *split* operation is then applied on the data queried. The results from all the *split* operations are gathered back to the node that received the operation. The original server then applies the *combine* operation on the gathered data to obtain an intermediate analysis result for that dataset, and the integrate operation is applied to all the intermediate results to generate a final result for all the datasets.

Use or disclosure of data contained on this sheet is subject to the restriction on the title page of this document.
Copyright © The HDF Group, 2014. All rights reserved



To easily interface with a user-defined analysis code on the client, the *split, combine* and *integrate* operations are defined in the form of Python scripts that are converted into strings, shipped and executed on the data. It is worth noting that the analysis shipping server must encapsulate / deencapsulate data into / from NumPy arrays before / after the Python scripts are executed.

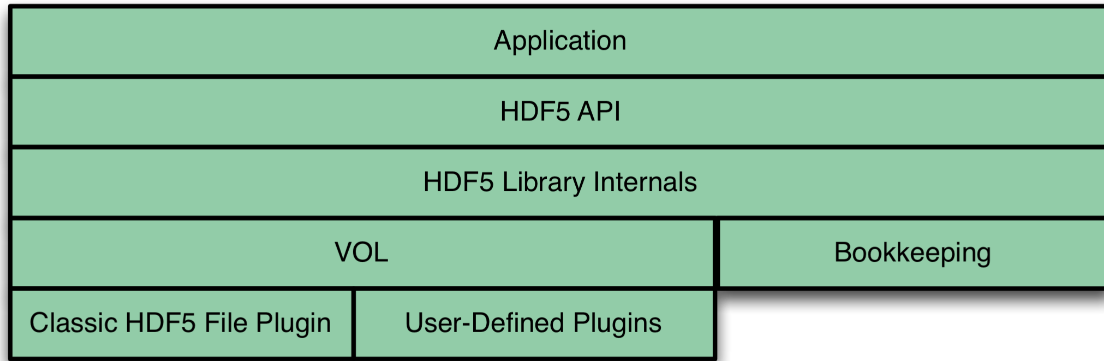
In Q6, the H5AS and H5Q routines apply to raw data elements of HDF5 dataset objects only, and the H5V and H5X routines have not yet been implemented. The H5V and H5X routines have been fully implemented as of Q8, along with updating the H5ASinvoke routine used to initiate analysis shipping operations to handle queries that apply to both metadata and raw data elements.

4.2 Architectural Changes to the HDF5 Library

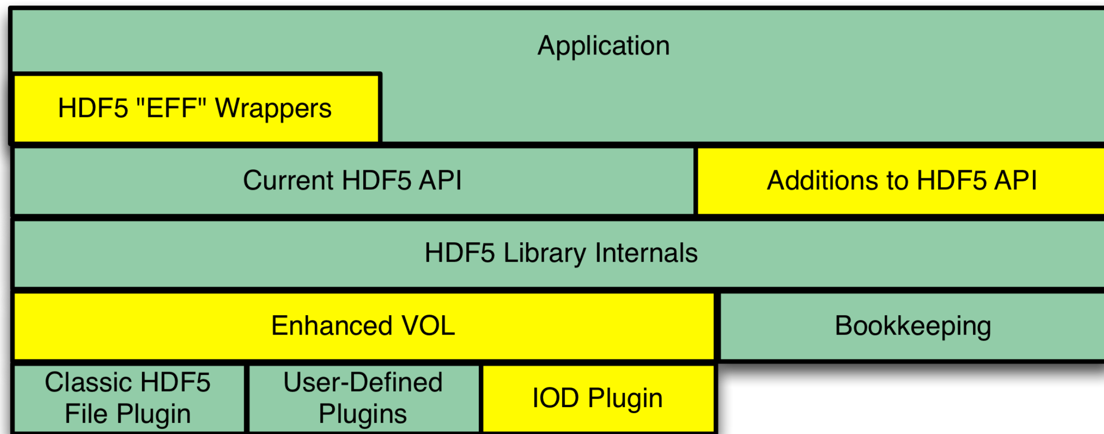
The architecture of the core HDF5 library was largely unaffected by the changes described in this document. The majority of the capabilities added to the HDF5 API were handled by a wrapper layer above the main HDF5 library, and a modest number of additions to the main API routines Adding transactions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

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

Use or disclosure of data contained on this sheet is subject to the restriction on the title page of this document.
Copyright © The HDF Group, 2014. All rights reserved



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



The majority of the implementation work was 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 the *HDF5 Data in IOD Containers Layout Specification* 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.

Use or disclosure of data contained on this sheet is subject to the restriction on the title page of this document.
Copyright © The HDF Group, 2014. All rights reserved

5 API and Protocol Additions and Changes

There were 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.

The reference manual pages for the modified and new routines can be found in the *User's Guide to FastForward Features in HDF5*. The reference manual pages for the designed, but not fully implemented DO (optimized dataset) operations have been retained in this design document.

5.1 Generic changes to HDF5 API routines

Many HDF5 API routines operate on HDF5 file objects and needed to be extended in similar ways. The generic modifications are described in this section. HDF5 API routines that were extended in this manner have the suffix "_ff"[11] added to the API routine's name.

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

1. A *read context id (rcxt_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 (trans_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 (estack_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 trans_id,
                                             hid_t estack_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>,
```

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

Use or disclosure of data contained on this sheet is subject to the restriction on the title page of this document.
Copyright © The HDF Group, 2014. All rights reserved

```
                                               hid_t rcxt_id,
                                               hid_t estack_id);
```

As a concrete example, the following prototypes show the changes to H5Gcreate, the group creation API routine for HDF5[12] — 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_ff(hid_t loc_id, const char *name, hid_t lcpl_id,
                hid_t gcpl_id, hid_t gapl_id,
                hid_t trans_id, hid_t estack_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[13] 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

Please refer to the *User's Guide to FastForward Features in HDF5* to see the reference manual pages for the HDF5 APIs added for the EFF project. The proposed, but not fully implemented, H5DO* routines are documented here.

5.2.1 Dataset Objects – Optimized APIs (designed but not fully implemented)

These descriptions reflect the function of the HDF5 routines delivered in Q4. Those routines were never connected to the lower layers of the EFF stack, and therefore remain only "stubs".

5.2.1.1 H5DOappend_ff

Name: H5DOappend_ff

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

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

Use or disclosure of data contained on this sheet is subject to the restriction on the title page of this document.
Copyright © The HDF Group, 2014. All rights reserved

Signature:

herr_t `H5DOappend_ff`(*hid_t* `dataset_id`, *hid_t* `dxpl_id`, *unsigned* `axis`, *size_t* `extension`, *hid_t* `memtype_id`, *const void* `*buffer`, *hid_t* `trans_id`, *hid_t* `es_id`)

Purpose:

Perform an optimized append operation on a dataset, possibly asynchronously.

Description:

The `H5DOappend_ff` routine extends the dataset specified by `dataset_id` `extension` number of elements along the dimension specified by `axis`, and writes the data values of `memtype_id` datatype located in `buffer` into the newly created dataset elements. Data transfer properties are defined by the argument `dxpl_id`.

`axis` must be a dimension that was declared unlimited when the dataset was created.

`trans_id` indicates the transaction this operation is part of.
EFF Note: This has no effect in the current version.

The `es_id` parameter indicates the event stack the event 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_STACK_NULL` for the `es_id` parameter.

The `H5DOappend_ff` routine combines calls to `H5Dset_extent`, `H5Sselect_hyperslab` and `H5Dwrite_ff` into a single, convenient routine that simplifies application development for the common case of appending elements to an existing dataset. It also 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 `H5DOappend` routine is identical in functionality, but does not allow for asynchronous operation or inclusion in a transaction.

Parameters:

hid_t `dataset_id`	IN: Identifier of the dataset where data will be appended.
hid_t `dxpl_id`	IN: Data transfer property list identifier.
unsigned `axis`	IN: Number specifying the dimension that is to be extended; must correspond to a dimension that was declared unlimited when the dataset was created.
size_t `extension`	IN: Number of elements that will be added along the specified dimension.
hid_t `memtype_id`	IN: Identifier of the memory datatype
const void `*buffer`	IN: Buffer with data to be written (appended) to the dataset
hid_t `trans_id`	IN: Transaction ID that this operation is a part of.
hid_t `es_id`	IN: Event stack identifier specifying the event stack that will be used to monitor the status of the event associated with this function call when

Use or disclosure of data contained on this sheet is subject to the restriction on the title page of this document.
Copyright © The HDF Group, 2014. All rights reserved

executed asynchronously. Use `H5_EVENT_STACK_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.

History:

Added in Quarter 4.

Quarter 5: Changed from transaction to transaction id and from event queue to event stack.

5.2.1.2 H5DOget_ff

Name: H5DOget_ff

Signature:

herr_t `H5DOget_ff(` *hid_t* `dataset_id`, *hid_t* `dxpl_id`, *const hsize_t* `coord[]`, *hid_t* `memtype_id`, *void* `*buffer`, *hid_t* `rcxt_id`, *hid_t* `es_id`)

Purpose:

Read a single element from a dataset, possibly asynchronously.

Description:

The `H5DOget_ff` routine reads a single element at offset `coord` from the dataset specified by `dataset_id` to `buffer` of `memtype_id` datatype. Data transfer properties are defined by the argument `dxpl_id`.

`rcxt_id` indicates the version of the container this operation reads from.
EFF Note: This has no effect in the current version.

The `es_id` parameter indicates the event stack the event 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_STACK_NULL` for the `es_id` parameter.

The `H5DOget_ff` routine combines calls to `H5Dselect_hyperslab` and `H5Dread_ff` into a single, convenient routine that simplifies application development for the common case of reading a single element from a dataset.

The `H5DOget` routine is identical in functionality, but does not allow for asynchronous operation or inclusion in a transaction.

Parameters:

hid_t `dataset_id`	IN: Identifier of the dataset to read from.
hid_t `dxpl_id`	IN: Data transfer property list identifier.
const hsize_t `coord[]`	IN: Vector whose values specify the coordinates of the elemen to be read. Vector length must equal number of dimensions in the dataset.
hid_t `memtype_id`	IN: Identifier of the memory datatype

Use or disclosure of data contained on this sheet is subject to the restriction on the title page of this document.
Copyright © The HDF Group, 2014. All rights reserved

void *buffer	OUT: Buffer to receive data read from the dataset
hid_t rcxt_id	IN: Read context ID, indicating version of container this operation reads from.
hid_t es_id	IN: Event stack identifier specifying the event stack that will be used to monitor the status of the event associated with this function call when executed asynchronously. Use H5_EVENT_STACK_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.

History:

Added in Quarter 4.

Quarter 5: Added read context id and changed from event queue to event stack.

5.2.1.3 H5DOset_ff

Name: H5DOset_ff

Signature:

herr_t H5DOset_ff(*hid_t* dataset_id, *hid_t* dxpl_id, *const hsize_t* coord[], *hid_t* memtype_id, *const void* *buffer, *hid_t* trans_id, *hid_t* es_id)

Purpose:

Write a single element to a dataset, possibly asynchronously.

Description:

The H5DOset_ff routine writes a single element at offset coord to the dataset specified by dataset_id from buffer of memtype_id datatype. Data transfer properties are defined by the argument dxpl_id.

trans_id indicates the transaction this operation is part of.
EFF Note: This has no effect in the current version.

The es_id parameter indicates the event stack the event 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_STACK_NULL for the es_id parameter.

The H5DOset routine combines calls to H5Dselect_hyperslab and H5Dwrite_ff into a single, convenient routine that simplifies application development for the common case of writing a single element to a dataset.

The H5DOset routine is identical in functionality, but does not allow for asynchronous operation or inclusion in a transaction.

Parameters:

hid_t dataset_id	IN: Identifier of the dataset to write.
hid_t dxpl_id	IN: Data transfer property list identifier.

Use or disclosure of data contained on this sheet is subject to the restriction on the title page of this document.
Copyright © The HDF Group, 2014. All rights reserved

const hsize_t coord[]	IN: Vector whose values specify the coordinates of the elemen to be written. Vector length must equal number of dimensions in the dataset.
hid_t memtype_id	IN: Identifier of the memory datatype
const void *buffer	IN: Buffer holding data to be written to the dataset
hid_t trans_id	IN: Transaction ID that this operation is a part of.
hid_t es_id	IN: Event stack identifier specifying the event stack that will be used to monitor the status of the event associated with this function call when executed asynchronously. Use H5_EVENT_STACK_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.

History:

Added in Quarter 4.

Quarter 5: Changed from transaction to transaction id and from event queue to event stack.

5.2.1.4 H5DOsequence_ff

Name: H5DOsequence_ff

Signature:

herr_t H5DOsequence_ff(*hid_t* dataset_id, *hid_t* dxpl_id, *unsigned* axis, *size_t* start, *size_t* sequence, *hid_t* memtype_id, *void* *buffer, *hid_t* rcxt_id, *hid_t* es_id)

Purpose:

Perform an optimized stream-oriented read operation on a dataset, possibly asynchronously.

Description:

The H5DOsequence_ff routine reads sequence elements from the dataset specified by dataset_id along the dimension specified by axis starting at offset start into buffer of memtype_id datatype. Data transfer properties are defined by the argument dxpl_id.

rcxt_id indicates the version of the container this operation reads from.
EFF Note: This has no effect in the current version.

The es_id parameter indicates the event stack the event 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_STACK_NULL for the es_id parameter.

The H5DOsequence_ff routine combines calls to H5Sselect_hyperslab and H5Dread_ff into a single, convenient routine that simplifies application development for the common case of appending elements to an existing dataset.

Use or disclosure of data contained on this sheet is subject to the restriction on the title page of this document.
Copyright © The HDF Group, 2014. All rights reserved

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 `H5DOsequence` routine is identical in functionality, but does not allow for asynchronous operation or inclusion in a transaction.

Parameters:

hid_t `dataset_id`	IN: Identifier of the dataset that will be read.
hid_t `dxpl_id`	IN: Data transfer property list identifier.
unsigned `axis`	IN: Number specifying the dimension that is to be sequenced through.
size_t `start`	IN: Number specifying the starting offset for the read
size_t `sequence`	IN: Number of elements that will be read along the specified dimension.
hid_t `memtype_id`	IN: Identifier of the memory datatype
void `*buffer`	OUT: Buffer for data read (sequenced) from the dataset
hid_t `rcxt_id`	IN: Read context ID, indicating version of container this operation reads from.
hid_t `eq_id`	IN: Event stack identifier specifying the event stack that will be used to monitor the status of the event associated with this function call when executed asynchronously. Use `H5_EVENT_STACK_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.

History:

Added in Quarter 4.

Quarter 5: Added read context id and changed from event queue to event stack.

5.2.1.5 H5Pset_dcpl_append_only

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

6 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

Use or disclosure of data contained on this sheet is subject to the restriction on the title page of this document.
Copyright © The HDF Group, 2014. All rights reserved

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.

Additional specific risks, unknowns and deferred features are outlined below:

- Deferred features, such as support for canceling asynchronous operations, variable-length datatypes or iterating over links in a group, may take more effort than anticipated to implement when desired. These features should be investigated more thoroughly and the implementation costs estimated as accurately as possible before further efforts to implement them are pursued.
- New features that have been added to HDF5 should be tested by porting more DOE applications to the EFF I/O stack. Iterating on the design of the extensions as part of this project before putting them into production will help reduce the future maintenance costs, both at the HDF5 level and for application developers.
- Although routines to append data to HDF5 datasets were defined, they were not implemented. These additions should be evaluated for their value and discarded if they are not valuable to applications.
- Index operations have a number of known limitations, stemming from the index packages chosen and the transactional model for the EFF I/O stack. Care evaluation of how to proceed with this valuable feature should be made.

The reader is also referred to the EFF I/O project final report for a more wholistic perspective, taking all layers of the EFF I/O stack into account, along with many more ideas for future work.

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[14]
Attribute	Name of attribute	Y	Y	Attribute reference[15]

[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.

Use or disclosure of data contained on this sheet is subject to the restriction on the title page of this document.
Copyright © The HDF Group, 2014. All rights reserved

Name	on an object			
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[13]
Map Value	Value of map entry	N	N	N/A

Use or disclosure of data contained on this sheet is subject to the restriction on the title page of this document.
Copyright © The HDF Group, 2014. All rights reserved