|  |  |
| --- | --- |
| **Date: February 12, 2014** | ***Burst Buffer Space Management – Prototype to Production***  **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 ITS SUBCONTRACT WITH INTEL AND 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 ii

1 Introduction 1

2 Definitions 1

3 Putting data into the Burst Buffer: IOD 1

3.1 Add Updates to a Transaction 2

3.2 Replicate Readable Data 4

4 Reading data: IOD 4

5 Removing data from the Burst Buffer: IOD 4

6 Putting data in the Burst Buffer: HDF5 5

7 Reading data: HDF5 5

8 Removing data from the Burst Buffer: HDF5 5

1 HDF5 Prefetch 6

2 HDF5 Evict 10

3 HDF5 Read 14

4 IOD Prefetch 15

4.1 Dec 2013 version 15

4.2 Needed version 15

5 IOD Evict 17

5.1 Dec 2013 version 17

5.2 Needed versions 17

6 IOD Read 18

6.1 Dec 2013 version 18

6.2 Needed version 19

Revision History

|  |  |  |  |
| --- | --- | --- | --- |
| **Date** | **Revision** | **Description** | **Authors** |
| Feb. 10, 2014 | 1.0 | Initial Draft for discussion with EFF technical partners. | Ruth Aydt, The HDF Group |
| Feb 12, 2014 | 1.1 | Added more content. Reasonably complete through section 5, then mostly outline and cut/paste of material that needs to be revised. Send to EFF technical partners, knowing considerably more work needed. | Ruth Aydt, The HDF Group |

# Introduction

The BB provides two primary functions in the context of the EFF stack. First, as the name suggests, it provides a buffer where bursty data can be written quickly and then drained off to spinning disk in the background at a slower rate between bursts. Second, it provides another level in the cache hierarchy of the system where data that exceeds CN memory can reside for faster access than would be possible if read directly from disk.

This document describes how data is moved in and out of the BB attached to IONs under the control of IOD. In addition to providing an overview of the the current features supported by the IOD API, the document discusses the HDF5 APIs that we expect to implement for the EFF project, and gives an overview of some additional features we believe should be supported as the EFF software stack moves from prototype to production.

# Definitions

ACG = Arbitrarily Connected Graph

BB = Burst Buffer

CN = Compute Node

CV = Container Version

DAOS = Distributed Application Object Storage

EFF = Exascale FastForward

IOD = I/O Dispatcher

ION = I/O Node

RC = Read Context

TID = Transaction ID at IOD level; At HDF5 level, there are transaction numbers (corresponding to uncommitted transactions) container versions (corresponding to committed transactions)

Tagged TID = Transaction ID for a replica at IOD level; Contains information about both the container version and how to find the replica in the BB. Type is same as TID

TN = Transaction Number

VOL = Virtual Object Layer

# Putting data into the Burst Buffer: IOD

There are two primary user-initiated ways for data to be written to the BB.

1. Add IOD object updates to a transaction.
   * IOD objects can be Arrays, KV Stores, or Blobs.
   * Updates can delete data, for example “delete KV entry with key = k”
   * Each update request is stored in log format to the BB, and the data becomes readable when the transaction is committed.
2. Replicate some or all of the readable data in an IOD object.
   * If the source of the replicated data is DOAS, this is a *prefetch*
   * If the source of the replicated data is the BB, this is referred to by IOD as a *multi-format replica* or *semantic resharding*, depending on the type of object*.*
     + The BB->BB replication is not discussed further in this document and is not supported by the prototype HDF5 APIs for EFF.

Other user-initiated operations, such as creating an IOD container or an IOD object, also add data to the Burst Buffer. This document ignores the issue of container and object creation, focusing on the data movement related to object contents.

## Add Updates to a Transaction

The IOD APIs *iod\_array\_write, iod\_array\_write\_list, iod\_kv\_set, iod\_kv\_set\_list, iod\_kv\_unlink\_keys, iod\_blob\_write, iod\_blob\_write\_list, and iod\_blob\_append* add updates to Array, KV Store, and Blob objects. All take a container handle, a TID, one or more object handles, an indication of “where” the data should be written (the Array cell(s), the key(s), the byte range(s) or “at the end”), and – with the exception of *iod\_kv\_unlink\_keys*, the data.

In our discussions we will assume all objects are in a single container and we will primarily focus our examples on Array and KV objects.

Before looking at an example, recall that (1) multiple transactions can be in progress at any given time, (2) transactions can be finished[[1]](#footnote-2) in any order order, and (3) transactions are committed in strict numeric order when all lower-numbered transactions have been committed or aborted.

When transaction #T is committed, the updates in the transaction are applied atomically and a new container version #CV=T becomes readable. When there is an open read context (RC) on a given container version, the container contents as of that version will remain readable until the context is closed. The container contents of highest container version at any point in time are guaranteed to be readable until the container is closed, even if there is no open read context on that version.

In IOD terminology, a TID is used to refer to both the transaction numbers of in-progress transactions and to the container versions of committed transactions.

Table 1 illustrates updates for two IOD objects (Array and KV Store) added to five transactions. For the Array, commas separate cell values, and \_’s are used to indicate no updates to a given cell in a given transaction. For the KV Store, the nomenclature is [key,value], with del[key] meaning a key (and it’s value) is deleted from the store.

Each object update for each transaction is logged to the BB by IOD. We refer to the updates for a given object for a given transaction as the *delta* for that object for that transaction. The transactions are shown in reverse numeric order in Table 1 because deltas in later transactions are effectively “layered” on earlier committed transactions.

|  |  |  |
| --- | --- | --- |
| Tr # | Array Deltas (1D, datatype=int, 5 elements) | KV Store Deltas |
| 5 | 5, 5, 5, 5, 5 | [A, 5] [B, 5] |
| 4 | \_, \_, \_, \_, \_ | [B, 4] |
| 3 | \_, \_, 3, 3, 3 | del[A] |
| 2 | \_, 2, 2, \_, \_ | [B, 2] |
| 1 | 1, 1, 1, 1, 1 | [A, 1] |

Table 1: IOD object updates in five transactions

Table 2 shows the data in the two IOD objects (Array and KV Store) at each container version after all five transactions have been committed. The table also also indicates the deltas (identifed by transaction number) that are needed to read the object data at a given CV.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| CV # | Array Data | Needed Deltas | KV Store Data | Needed Deltas |
| 1 | 1, 1, 1, 1, 1 | 1 | [A, 1] | 1 |
| 2 | 1, 2, 2, 1, 1 | 1, 2 | [A, 1] [B, 2] | 1, 2 |
| 3 | 1, 2, 3, 3, 3 | 1, 2, 3 | [B, 2] | 1, 2, 3 |
| 4 | 1, 2, 3, 3, 3 | 1, 2, 3 | [B, 4] | 3, 4 |
| 5 | 5, 5, 5, 5, 5 | 5 | [A, 5] [B, 5] | 5 |

Table 2: IOD object contents in five container versions and deltas needed to read each version

Container versions can be *persisted* to DAOS, which means the *necessary* data for the objects in the container, as of the version being persisted, is written to DAOS in an atomic DAOS-level transaction. Every CV need not be persisted.

To better understand what we mean by *necessary* data, let’s refer to the information in Tables 1 and 2, and look in detail at two different scenarios and the resulting data transfers from the BB to DAOS:

Scenario 1:  
a) CV 1 is persisted --> the data “1,1,1,1,1” and “[A,1]” is written to DAOS  
b) CV 2 is persisted --> the data “2,2” (2nd & 3rd cells) and “[B,2]” is written atomically to DAOS

Scenario 2:  
a) CV 1 is not persisted  
b) CV 2 is persisted --> the data “1,2,2,1,1” and “[A,1] [B,2]” is written atomically to DAOS

## Replicate Readable Data

The IOD API *iod\_obj\_fetch* is used to make a *replica* of data that is stored on DAOS in the BB. This API takes an object handle, a TID, and parameters that control which data in the object will be replicated and how the replicated data will be laid out in the BB. Depending on the type of IOD object referenced (Array, KV Store, Blob), the user may or may not be able to specify a subset of the object’s data be replicated in the BB -- for KV objects all the data in the KV Store must be included in the replica. [Ruth waiting for response from EMC to confirm what is possible].

When *iod\_obj\_fetch* is called to create a replica, IOD returns a special *Tagged TID* that must be used to read data from the replica, or to evict the replica from the BB. The Tagged TID has the same type as IOD’s TID. But, it contains not only the “number” (container version), but also an identifer that allows the specific replica of the data to be found by IOD.

# Reading data: IOD

The IOD APIs *iod\_array\_read, iod\_array\_read\_list, iod\_kv\_get\_value, iod\_blob\_read, iod\_blob\_read\_list, and iod\_blob\_append* read data from Array, KV Store, and Blob objects. Additional IOD APIs not enumerated here also perform read operations to retrieve information about the Array, about the number of entries in a KV store, and so on. All take a container handle, a TID, one or more object handles, an indication of “where” the data should be read from (the Array cell(s), the key(s), the byte range(s)), and where in memory the read data should be stored.

If a tagged TID is passed in to the IOD call, the referenced replica will be used to read the data. If all of the data requested is not available from the replica, IOD will retrieve the remaining data from DAOS. [Ruth waiting for confirmation of retrieval order... if it’s elsewhere in non-tagged TID will it be read from there? What about if in tagged TID?]

If a (non-tagged) TID is passed in to the IOD call, IOD will retrieve data from the appropriate deltas, if they are still in the BB, and will retrieve the remaining data from DAOS.

When IOD reads data from DAOS, it passes through the memory of the ION on its way to the memory of the CN, but does not get copied into the BB.

# Removing data from the Burst Buffer: IOD

The IOD API *iod\_obj\_purge* removes data from the BB, freeing the space to be used for other data. This call takes an object handle and a TID.

If a tagged TID is passed to the call, the referenced replica will be removed from the BB.

If a (non-tagged) TID is passed, .... rules about when eviction will fail to make sure we meet commitments need to be explained here. See Ruth’s email from 2/11 and scenarios to explain behavior.

IOD leave Burst Buffer space management entirely up to the user, and does not currently offer any way to say “remove data for objects in this container” [useful at file close] or “remove data for all objects at this CV” [useful when CV has been persisted and no expectation that any data will be read, as in checkpoint – believe we will get this in Q8] or “remove all the data for an aborted transaction – regardless of when it arrives in the BB” [useful if the user or the system aborts a transaction. would be nice if this happened automatically].

**---- Ruth notes mostly outline / cut-paste of older information from here on... need to return and revise.**

# Putting data in the Burst Buffer: HDF5

At the HDF5 layer of the stack, the primary objects are H5Groups, H5Datasets, H5Maps, and H5NamedDatatypes.

* refer to Mohamad’s writeup re: how these get mapped to IOD objects. Not one-to-one
* Talk about the write write operations that update the HDF (and IOD) objects
* Talk about prefetch and how HDF5 doesn’t used tagged TID.

To separate the two parts of the tagged TID, the HDF5 layer refers to the CV (the container version that is being replicated) and the replica ID (an indentifier of the particular copy of the data in the BB) rather than combining the two into a single identifier.

Talk about APIs. [what we want and what we’ll have]

# Reading data: HDF5

Talk about APIs. [what we want and what we’ll have]

Talk about how replica ID will be passed in with property list & how only the ‘main’ IOD object for a given HDF5 object will be prefeteched.

Talk about how with VL data, you won’t get the blobs prefetched that actually contain the data.

We won’t demo prefetech of IOD blob objects because won’t prefetch H5NamedDatatypes & no VL prefeteched.

# Removing data from the Burst Buffer: HDF5

Talk about APIs. [what we want and what we’ll have]

-------- **The following copied from Ruth’s earlier doc re: what we needed / wanted in FF & PF. Hasn’t been updated to reflect what we now know we’ll get from IOD and how will we use it. Lots of changes to default behavior & down-scoping of what we can do. Will go through & revise, but this is a starting point.**

# HDF5 Prefetch

|  |  |
| --- | --- |
| *Strictly talking about prefetch routines here – that move data from DAOS to the BB.*  *In future might add “replicate” routines that would take data already in BB and rearrange it in a replica in the BB, but don’t plan to expose that capability at the HDF5 level in FF.* | |
| herr\_t **H5Aprefetch\_ff** ( hid\_t attr\_id,  hid\_t rcntxt\_id hrep\_t \*replica\_id, // OUT *hid\_t* aapl\_id *hid\_t* es\_id ) | Prefetch an H5attribute or a subset of an H5attribute at a given CV.  replica\_id indicates where the prefeteched data can be found.  aapl\_id is an attribute access property list identifier.  Properties allow:   1. “first-index-order” or “last-index-order” layouts 2. subsets (hyperslabs) of the data to be specified 3. placement of the data on the ION that the CN issuing the prefetch is connected to. 4. round-robin layout of the data with a given stripe per ION, with stripe size expressed in terms of # of cells 5. for VL (variable length) datatype, don’t prefetch the IOD Blob object(s) that hold the VL data. Default is to prefetch the blobs. (VL Attribute values not currently supported in FF) 6. ability to direct placement of VL blob data (see H5Tprefetch\_ff for ideas)   Delivery timeline:  PF |
| herr\_t **H5Dprefetch\_ff** ( hid\_t dset\_id,  hid\_t rcntxt\_id, hrep\_t \*replica\_id, // OUT *hid\_t* dapl\_id *hid\_t* es\_id ) | Prefetch an H5dataset or subset of an H5dataset at a given CV.  If the datatype is variable length, default behavior is to also prefetch the IOD Blobs that hold the VL data for the dataset cells being prefetched (based on modifier parameters)  replica\_id indicates where the prefeteched data can be found.  *Requirement that filters down to IOD*: When you pre-fetch a sub-object (say the first row in a dataset), then later pre-fetch another sub-object (say the 2nd row in a dataset), there must be different replica\_ids for these 2 prefetch operations so that they can later be evicted separately.  dapl\_id is a data access property list identifier.  Properties allow:   1. Prefetch attributes along with dataset. (default is attributes aren’t prefetched) 2. “first-index-order” or “last-index-order” layouts 3. subsets (hyperslabs) of the data to be specified 4. placement of the data on the ION that the CN issuing the prefetch is connected to. 5. round-robin layout of the data with a given stripe per ION, with stripe size expressed in terms of # of cells 6. for VL datatype, don’t prefetch the IOD Blob object(s) that hold the VL data. [default is to prefetch the blobs] 7. ability to indicate the metadata associated with the datatype (datatype indicator, byte order, dimensions, etc), should not be prefeteched – just the cell values and Blobs (if VL datatype) 8. ability to direct placement of VL blob data (see H5Tprefetch\_ff for ideas)   Delivery timeline:  Q7: prefetch full objects with default layout (for Demo)  Q8: prefetch subsets (3); ability to put all data that was fetched by the call on “my” ION (4). [to support ACG]  PF: Everything else |
| herr\_t **H5Gprefetch\_ff** ( hid\_t group\_id,  hid\_t rcntxt\_id, hrep\_t\*replica\_id, // OUT *hid\_t* gapl\_id *hid\_t* es\_id ) | Prefetch an H5group at a given CV.  replica\_id indicates where the prefeteched data can be found.  gapl\_id is a group access property list identifier.  Properties allow:   1. prefetch attributes along with group 2. specify one sub-range of keys (link names) to fetch 3. specify multiple sub-ranges of keys (link names) to fetch 4. placement of all the fetched data on the ION that the CN issuing the prefetch is connected to. 5. specify placement of individual sub-ranges of keys (link names). [not default (What is default? Is it round robin, evenly divided?) and not all-on-closest ION] 6. prefetch this group and all its members and all their attributes 7. prefetch this group and all its descendents   Delivery timeline:  Q7: prefetch full objects with default layout (for Demo)  PF: Everything else.  Note that 6&7 may require additional function to support because replica\_ids for the set of objects involved in the prefetech may not have the same values. |
| herr\_t **H5Mprefetch\_ff** ( hid\_t map\_id,  hid\_t rcntxt\_id, hrep\_t\*replica\_id, // OUT *hid\_t* mapl\_id, *hid\_t* es\_id ) | Prefetch an H5map or sub-map at a given CV  replica\_id indicates where the prefeteched data can be found.  mapl\_id is an map access property list identifier.  Properties allow:   1. prefetch attributes along with map 2. specify one sub-range of keys to fetch 3. specify multiple sub-ranges of keys to fetch 4. placement of all the fetched data on the ION that the CN issuing the prefetch is connected to. 5. specify placement of individual sub-ranges of keys [not default (round robin, evenly divided ?) and not all-on-closest ION]   Delivery timeline:  Q7: Prefetch full objects with default layout (for Demo)  Q8: Prefetch one sub-range of keys (2) and place all data on closest ION (4) [to support ACG]  PF: everything else. |
| herr\_t **H5Tprefetch\_ff** ( hid\_t dtype\_id,  hid\_t rcntxt\_id, hrep\_t\*replica\_id, // OUT hid\_t tapl\_id, *hid\_t* es\_id ) | Prefetch an H5namedDatatype at a given CV  replica\_id indicates where the prefeteched data can be found.  tapl\_id is an datatype access property list identifier.  Properties allow:   1. prefetch attributes along with named datatype 2. placement of the data on the ION that the CN issuing the prefetch is connected to. 3. round-robin layout of the data with a given stripe per ION, with stripe size expressed in terms of # of bytes   Q7: prefetch full objects with default layout (for Demo)  PF: Everything else. |
| herr\_t **H5CVprefetch** ( hid\_t rcntxt\_id, *hid\_t* es\_id ) | Prefetch all H5objects in the container (H5file) for the container and container version indicated by rcntxt\_id.  Useful in cases where reading all data in a container that has just been opened for read. Assumes all data fits into BB. And, that no other replicas of the data will be created (so no replica\_id)  Delivery timeline:  PF *Would need support from IOD to do this. Restart may be one important use case, esp if the data could be prefetched by a job scheduler before the job is restarted.* |
|  |  |
|  |  |

# HDF5 Evict

|  |  |
| --- | --- |
| Notes on properties that modify the selection behavior used to determine what is evicted.  The properties are ANDed with other selection criteria (object, container version) and only the data that matches all selection criteria is evicted.  cv\_match\_property:  if == EXACT, only container\_version specified is evicted. **This is the default behavior --> *Note, this is different that what IOD currently does, which is LE***  if == ALL, all container versions are evicted (container\_version is ignored)  if == LE, all container versions less than or equal to container\_version are evicted.  if == LT, all container versions less than container\_version are evicted.  if == GE, all container versions greater than or equal to container\_version are evicted..  if == GT, all container versions greater than container\_version are evicted.  if == NE, all container versions except container\_version are evicted.  replica\_id\_property:  if == ALL\_REPLICAS, evict all replicas. **This is the default behavior**  ***--> Note, this is different than what IOD currently does, which is == LE***  if == TRANSACTION\_DELTAS, evict the ‘logged deltas’ for this object/cv that were written to the transaction with the same number as cv.  *--> Note, this will probably evict just the updates that were made to the object in a given transaction (that has already been committed), and not the “full” view of the object as of the specified CV. Closer to current IOD API, where focus seems to have been on evicts from logged deltas, and not from ‘replicated data’. Need to* ***discuss.***  if == replica\_id, where replica\_is was returned in a prefetch call, evict exact replica stated.  *NOTE: Need to* ***discuss*** *what happens if attempt to evict something that will violate the rules about readability of any version that has an open read handle and readability of last committed version. This may only be an issue when deltas are evicted, not when replicas are evicted.*  *Also, difference between evicting “log entries added to transactions” and evicting replicas – are there different considerations that may require different evict operations? See notes with TRANSACTION\_DELTAS in replica\_id property.*  **Default Behavior: Evict all copies of data for the object and container version specified from the BB. Data in lower-numbered container versions that may be in the BB (the transaction deltas) and contribute values to the CV being evicted will not be evicted with the default settings.** | |
| herr\_t **H5Aevict\_ff** ( hid\_t attr\_id,  uint64\_t container\_version, *hid\_t* aapl\_id *hid\_t* es\_id ) | Evict data associated with given H5attribute and container\_version from the BB.  Note, container\_version means this is a committed transaction.  aapl\_id is an attribute access property list identifier.  Properties allow:   1. cv\_match\_property non-default behaviors 2. replica\_id\_property non-default behaviors 3. For variable-length data types, evict the IOD blobs holding the VL data, but don’t evict the IOD array associated with the attribute (currently no VL for Attributes)   Delivery timeline:  Q8: default options only.  PF: Everything else. |
| herr\_t **H5Devict\_ff** ( hid\_t dset\_id,  uint64\_t container\_version, *hid\_t* dapl\_id *hid\_t* es\_id ) | Evict data associated with given H5dataset and container\_version from the BB.    Note, container\_version means this is a committed transaction.  dapl\_id is an dataset access property list identifier.  Properties allow:   1. cv\_match\_property non-default behaviors 2. replica\_id\_property non-default behaviors 3. evict the dataset and all its attributes 4. For variable-length data types, evict the IOD blobs holding the VL data, but don’t evict the IOD array associated with the attribute   *Requirement that filters down to IOD*: Say you pre-fetch a sub-object (say the first row in a dataset) and get replica\_id\_1, then later pre-fetch another sub-object (say the 2nd row in a dataset) and get replica\_id\_2. If you make this call with replica\_id\_property == replica\_id\_1, only the first row of the dataset is evicted.  Delivery timeline:  Q7: default options (EXACT/ALL\_REPLICAS) (for Demo)  Q8: add support for LE, all replica\_id\_property settings.  PF: everything else |
| herr\_t **H5Gevict\_ff** ( hid\_t group\_id,  uint64\_t container\_version, *hid\_t* gapl\_id *hid\_t* es\_id ) | Evict data associated with given H5group and container\_version from the BB.  Note, container\_version means this is a committed transaction.  gapl\_id is a group access property list identifier.  Properties allow:   1. cv\_match\_property non-default behaviors 2. replica\_id\_property non-default behaviors 3. evict the group and all its attributes 4. evict the group and all its members and all their attributes 5. evict the group and all its descendents. 6. Possibly others, such as all the members of this group but not the group itself.   Delivery timeline:  Q7: cv\_flag == EXACT; all options for replica\_id;  PF: Full cv\_flag capabilities, support for gapl\_id; ;tricky w/ group descendents & coordinateion of replica\_ids.  Delivery timeline:  Q7: default options (EXACT/ALL\_REPLICAS) (for Demo)  Q8: add support for LE, all replica\_id\_property settings.  PF: everything else  Note that 4&5 may require additional function to support because replica\_ids for the set of objects involved in the evict may not have the same values. |
| herr\_t **H5Mevict\_ff** ( hid\_t map\_id,  uint64\_t container\_version, *hid\_t* mapl\_id *hid\_t* es\_id ) | Evict data associated with given H5map and container\_version from the BB.  Note, container\_version means this is a committed transaction.  mapl\_id is a data access property list identifier.  Properties allow:   1. cv\_match\_property non-default behaviors 2. replica\_id\_property non-default behaviors 3. evict this map and all its attributes 4. Possibly others,   Q7: default options (EXACT/ALL\_REPLICAS) (for Demo)  Q8: add support for LE, all replica\_id\_property settings.  PF: everything else |
| herr\_t **H5Tevict\_ff** ( hid\_t dtype\_id,  uint64\_t container\_version, hid\_t tapl\_id, *hid\_t* es\_id ) | Evict data associated with the given H5namedDatatype from the BB.  Note, container\_version means this is a committed transaction.  tapl\_id is a datatype access property list identifier.  Properties allow:   1. cv\_match\_property non-default behaviors 2. replica\_id\_property non-default behaviors 3. Possibly others,   Q7: default options (EXACT/ALL\_REPLICAS) (for Demo)  Q8: add support for LE, all replica\_id\_property settings.  PF: everything else |
| herr\_t **H5TRevict** ( hid\_t file\_id, uint64\_t transaction\_num, *hid\_t* es\_id ) | Evict the “logged deltas” for all objects and the given transaction from the BB.  Note, the use of transaction\_num indicates this is a transaction that has not been committed.  There needs to be some way for the updates that have been added to a transaction which is aborted to be cleared out of the BB.  Trying to have the user do this is difficult (impossible?) because with the application or the lower-layers in the I/O stack can abort a transaction. And, due to the asynchronous and multi-process nature, quite likely additional updates will be added to a transaction after it has been aborted.  We really think the IOD layer should evict any existing updates (logged deltas) for a transaction that has been aborted. And, should immediately throw away (never log) the updates that come after a transaction is aborted.  Delivery timeline:  Q8 or PF: **IOD** should really do this without application involvement. If it doesn’t, then need IOD support to manage race conditions and give the application some way to clean things up after aborts occur. |
| herr\_t **H5CVevict** ( hid\_t file\_id, uint64\_t container\_version, enum\_t cv\_match\_property, *hid\_t* es\_id ) | Evict log entries and replicas for multiple objects from the BB.  container\_version and cv\_match\_property control which versions of the objects are evicted from the BB. Note, container\_version means this is a committed transaction.  *Ruth’s design note: Decided not to use rcntxt\_id, because may not have a read costext on every version you want to evict (esp when you’ve been writing to the container)*  Probably used before container close.  On read, might use after one timestamp has been read/processed and ready to move to the next.  Delivery timeline:  PF: Need IOD support |
|  |  |
|  |  |

# HDF5 Read

For the FF project, reading from a specific replica would not be supported. IOD should find a ‘good replica’ (maybe even the best possible), and satisfy the read from there. If all of the data is in the BB, then the read should be satisfied from the BB. If some is in BB and some in DAOS, the hope is that the read would be specified from the ‘closest’ data. [unsure of implications on checksum computation]

In the future, the replica\_id could be passed in as part of the property list in a read operation, and if present, would be given to IOD in the read call. IOD would satisfy as much of the read request as it could from the given replica, then look elsewhere fo rthe data.

For example, in future work, the replica\_id would be part of the dxpl\_id property list in the H5Dread call.

herr\_t H5Dread\_ff( hid\_t dset\_id, hid\_t mem\_type\_id, hid\_t mem\_space\_id, hid\_t file\_space\_id, hid\_t dxpl\_id, void \* buf, *hid\_t* rcntxt\_id,   
*hid\_t* es\_id )

Read APIs for accessing data in other types of H5 objects have similar property lists.

# IOD Prefetch

## Dec 2013 version

/\*\*

\* Fetch/pre-stage one object from central storage to BB.

\*

\* Only single rank can call this routine to fetch one object from central

\* storage to BB.

\*

\* \param oh [IN] object handle

\* \param tid [IN] transaction ID

\* \param hints[IN] pointer to hints and can be NULL when no hint

\* \param slab [IN] the hyperslab within this object

\* will be ignored for KV object.

\* \param layout [IN] passed in layout placement on BB,

\* can be NULL if does not change layout on BB.

\* \param tag[IN/OUT] returned tag TID, will be same as passed in

\* \a tid if does not change the \a layout on BB.

\* \param event [IN] completion event pointer list

\*

\* \return zero on success, negative value if error

\*/

iod\_ret\_t

iod\_obj\_fetch(iod\_handle\_t oh, iod\_trans\_id\_t tid, iod\_hint\_list\_t \*hints,

iod\_hyperslab\_t \*slab, iod\_layout\_t \*layout,

iod\_trans\_id\_t \*tag, iod\_event\_t \*event);

## Needed version

/\*\*

\* **Prefetch** one object **or sub-object** from central storage to BB.

\*

\* ~~Only single rank can call this routine to~~ Fetch one object or **sub-object**

**\*** from central storage to BB.

\* **NOTE: want to allow multiple ranks to call this so, for example, you can have data   
 \* replicated on all IONS. Not needed for FF, but for PF.**

\*

\* \param oh [IN] object handle

\* \param **cv** [IN] **container version (must be a committed & persisted)**

\* \param hints[IN] pointer to hints and can be NULL when no hint

\* \param **selection** [IN] **The selection within this object.**

**\* - For Arrays, effectively specifies a set of cells**

**\* - For Blobs, effectively specifies a set of bytes**

**\* - For KVs, effectively specifies a range of keys**

**\* If NULL, “whole object” is fetched.**

**\* NOTE: There should be a different prefetch API for each type of object since   
 \* the selection & layout that makes sense is different for each.**

\* \param layout [IN] The layout placement in BB.

\* - If NULL, IOD can choose layout

\* - Need “all data selected to “this” ION”;

\* - User shouldn’t have to pass byte or cell

\* counts or stripe sizes to get this option.

\* - ION would be the one that issues the call,

\* - ACG needs this capability for **Q8 demos**

\* - Need way to specify stripe size, if data to be

\* striped across BB on multiple IONs.

\* - For array, specify in terms of # of cells

\* - For array, if selection, should be able   
 \* specify order (first-index-major or

\* last-index-major [not sure other-index-major  
 \* needed] Don’t understand EMC-mentioned

\* restrictions w/ extensible arrays, because   
 \* this is read-only data.

\* - For Blob, specify in terms of # of bytes

\* - For KV, not sure exactly how to specify.

\* - could be first N keys on ‘my’ ION,

\* next N on next ion, etc, where user

\* supplies N

\* - could be range specification <X, <y, <z

\*

\* \param **prefetch\_id[IN/OUT] returned identifer that, in combination with an object**

**\* handle and container version, can be used to find**

**\* this exact pre-fetched data in the BB.**

**\* (ask for one or push one in – like object ids.)**

**\* - In PF, be used in a read call, especially when >1**

**\* copy of data in BB.**

**\* - Can be used in an evict call, to evict only this**

**\* copy of the data, but not all the object/cv data**

**\* in the BB.**

\*

\*

\* \param event [IN] completion event pointer list

\*

\* \return zero on success, negative value if error

\*/

iod\_ret\_t

iod\_obj\_fetch(iod\_handle\_t oh, iod\_trans\_id\_t cv, iod\_hint\_list\_t \*hints,

**iod\_selection\_t \*selection, iod\_layout\_t \*layout,**

**iod\_replica\_id\_t \*prefetch\_id,** iod\_event\_t \*event);

*Note: I haven’t looked at the data types for the parameters to see what they provide. Focusing primarily on saying the capabilities we need & trying to highlight where I think there are issues.*

# IOD Evict

## Dec 2013 version

/\*\*

\* Purge a object for an IOD container from BB.

\*

\* Only single rank can call this routine to purge one object from BB.

\*

\* Commonly there are two kinds of usages:

\* 1. used after migrating done and that object of \a tid is un-needed to be

\* kept on BB. IOD won't do auto-purging, user should explicitly call purge()

\* at appropriate time to free BB's storage space. Purging is at object

\* level, IOD will purge all lower TIDs and this TID \a tid.

\* User can only call purge with the \a tid that ever passed in by

\* iod\_trans\_persist.

\* 2. used for removing the replicated or pre-fetched object. For this usage,

\* the passed in "tid" parameter should be the replica or pre-fetched object

\* tag.

\*

\* \param oh [IN] object handle

\* \param tid [IN] transaction ID

\* \param hints[IN] pointer to hints and can be NULL when no hint

\* \param event [IN] pointer to completion event

\*

\* \return zero on success, negative value if error

\*/

iod\_ret\_t

iod\_obj\_purge(iod\_handle\_t oh, iod\_trans\_id\_t tid, iod\_hint\_list\_t \*hints,

iod\_event\_t \*event);

## Needed versions

**/\*\***

**\* Purge all data associated with an uncommitted transaction from the BB.**

**\***

**\* May be called by multiple ranks. Used when a transaction has been aborted to get**

**\* all of the discarded updates from the BB. May need to be called multiple times,**

**\* because updates can be added to the transaction after it has been aborted.**

**\***

**\* (If IOD can handle this cleanup of discarded updates w/o user help, then this**

**\* function not needed) *Want IOD to do this. IOD knows when aborted so in***

***\* discarded state just toss the entries. Also see notes in H5TRevict above.***

**\***

**\***

**\* \param coh [IN] container handle [not needed if findable from tid]**

**\* \param tid [IN] transaction ID**

**\* \param hints[IN] pointer to hints and can be NULL when no hint**

**\* \param event [IN] pointer to completion event**

**\***

**\* \return zero on success, negative value if error**

**\*/**

**iod\_ret\_t**

**iod\_trans\_purge(iod\_handle\_t coh, iod\_trans\_id\_t tid, iod\_hint\_list\_t \*hints,**

**iod\_event\_t \*event);**

/\*\*

\* Purge **specified object data from the BB.**

\*

\* Only single rank can call this routine to purge one object from BB.

\* **[why do we need this restriction? very hard (impossible?) to guarantee]**

\*

\*

\*

\* \param oh [IN] object handle

\* \param **cv [IN] container version – only committed objects can be purged with**

**\* this function.**

**\* \param replica\_id [IN] –**

**\*** if == TRANSACTION\_DELTAS **only data in initial transaction update logs is purged.**

**\*** if == ALL\_REPLICAS, evict all copies of the data for the object/CV from the BB. DEFAULT.

\* if == something else, **only data from identified prefetch (or replica) is purged.**

\* \param hints[IN] pointer to hints and can be NULL when no hint

\* \param event [IN] pointer to completion event

\*

\* \return zero on success, negative value if error

\* **[ please explain what errors could occur ]**

**\* Should be an error to try and purge non-committed data;**

**\* Also to purge data that is needed to satisfy reads of CVs**

**\* with open Read Handles or the largest CV in the container.**

**if prefetch\_id is sub\_object, only purge data for that sub-objec,t not other subobjects at same CV.**

\*/

iod\_ret\_t

iod\_obj\_purge(iod\_handle\_t oh, iod\_trans\_id\_t **cv**, **iod\_replica\_id\_t** **prefetch\_id** iod\_hint\_list\_t \*hints,

iod\_event\_t \*event);

# IOD Read

## Dec 2013 version

/\*\*

\* Read from one IOD array object.

\*

\* \param oh [IN] object handle

\* \param tid [IN] transaction ID

\* \param hints[IN] pointer to hints and can be NULL when no hint

\* \param mem\_desc [IN] pointer to memory buffers descriptor

\* \param io\_desc[IN] pointer to I/O descriptor

\* \param cs[IN/OUT] returned checksum for the read

\* \param event [IN] pointer to completion event

\*

\* \return zero on success, negative value if error

\*

\* notes: The io\_desc(hyperslab selection) cannot exceed array's

\* space; and total number of bytes of mem\_desc and io\_desc

\* must exactly equal. Or -EINVAL will be returned.

\* This is somehow different with blob read.

\*/

iod\_ret\_t

iod\_array\_read(iod\_handle\_t oh, iod\_trans\_id\_t tid,

iod\_hint\_list\_t \*hints, iod\_mem\_desc\_t \*mem\_desc,

iod\_array\_iodesc\_t \*io\_desc, iod\_checksum\_t \*cs,

iod\_event\_t \*event);

## Needed version

iod\_ret\_t

iod\_array\_read(iod\_handle\_t oh, iod\_trans\_id\_t cv,   
 iod\_replica\_id\_t replica\_id,

iod\_hint\_list\_t \*hints, iod\_mem\_desc\_t \*mem\_desc,

iod\_array\_iodesc\_t \*io\_desc, iod\_checksum\_t \*cs,

iod\_event\_t \*event);

* needs to take object ID, CV, replica\_id. (or put replica\_id in hints)
* If replica\_id is not specified (or is enum “BEST\_AVAILABLE”) then read from ‘best cached copy’, falling back to other cached copies to fill in holes, or finally to DAOS.
* If replica\_id is specified, go to that replica, then fall back to others to fill in holes. This could be PF.
* EG: 1x10 array; no replica\_id specified:

prefetch cells 0-5 onto ION 0; prefetch cell 10 onto ION 1 (all CV 100).

read CV 100, cells 4-10, w/ replica\_id = BEST\_AVAILABLE.

IOD Action: cells 4-5 found on BB0; cell 10 found on BB1; cells 6-9 found on DAOS.

**....**

**Leftover text from first version of this document. May integrate into discussion of evict/purge and why the rules are as they are.**

In the scenario given, for example, say there is an open read handle on CV 1. CVs 1 and 5 are persisted to DAOS. At this point, all of the deltas (1-5) for both objects could be deleted from the BB without breaking the commitments related to readablity of a CV with an open read handle (CV 1) and the readability of the latest CV (CV 5) because the container data for the two CVs can be retrieved from DAOS.

In a slightly different example, say there are open read handles on CV 1 and CV 4. CVs 1 and 5 are persisted to DAOS. In this case, deltas 1 and 5 for both objects, and delta 2 for the KV could be deleted from the BB without breaking the commitments. However, with the current IOD API it is not possible to evict (purge) a higher-numbered delta for a given object without also evicting all of the lower-numbered deltas for the object, so only delta 1 could be purged for the Array, and deltas 1 and 2 for the KV.

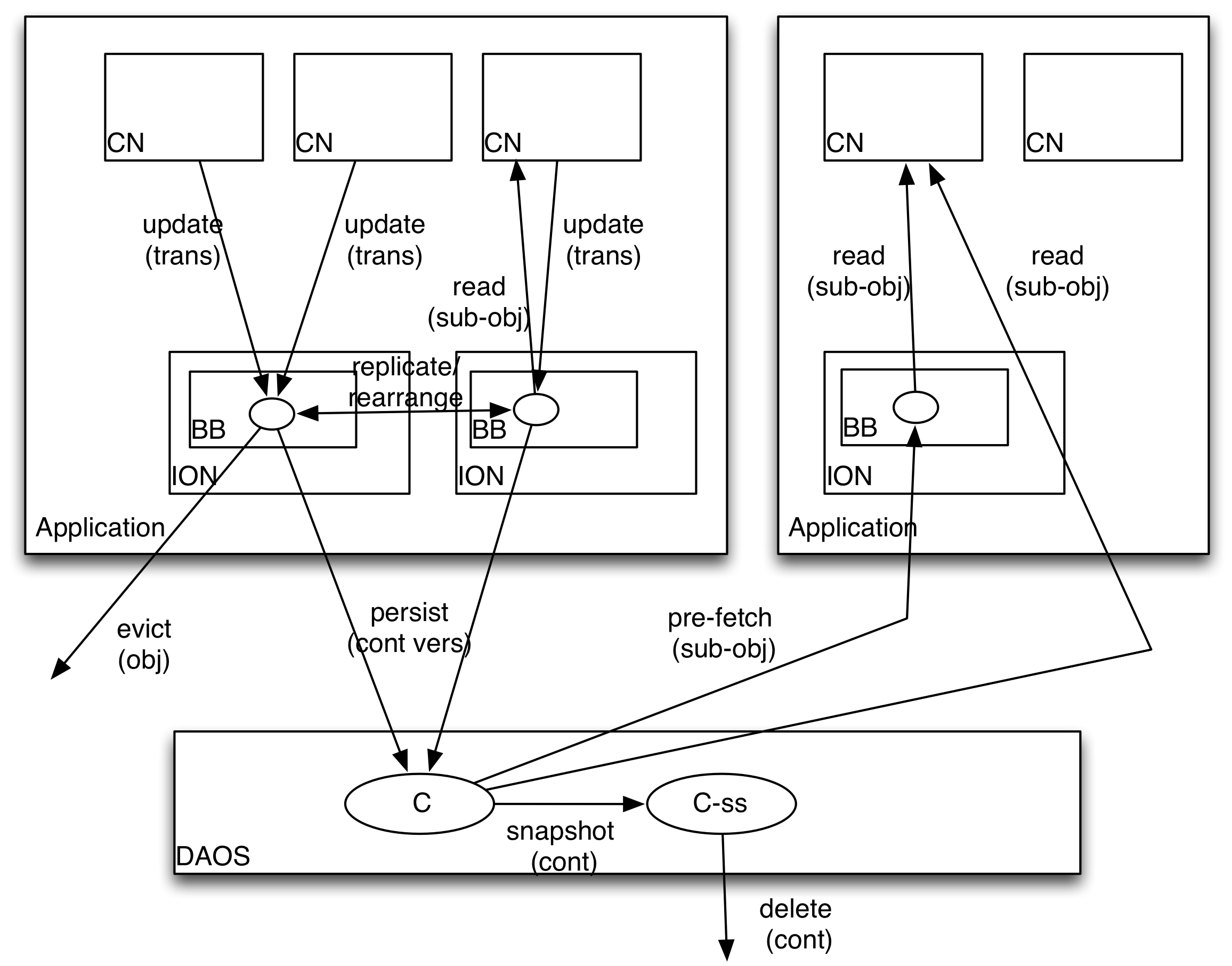


Figure 1: Data movement in the Exascale Fast Forward stack controlled by application requests.

1. A transaction is finished when no more updates will be added to it. [↑](#footnote-ref-2)