Tutorial on the DAOS API

SC24 Tutorial

Mohamad Chaarawi

Notices and Disclaimers

DAOS User API

- Pools:
  - Connect, disconnect, query
- Containers:
  - Create, destroy, open, close, query, properties, attributes
- Objects:
  - Access APIs based on type

DAOS Usage flow

- Initialize DAOS stack and connect to a Pool
- Create / open a container
- Access an object in the container through the unique oid
  - Open object, update/fetch/list, close object
  - Transaction API available
- Close container & disconnect from pool

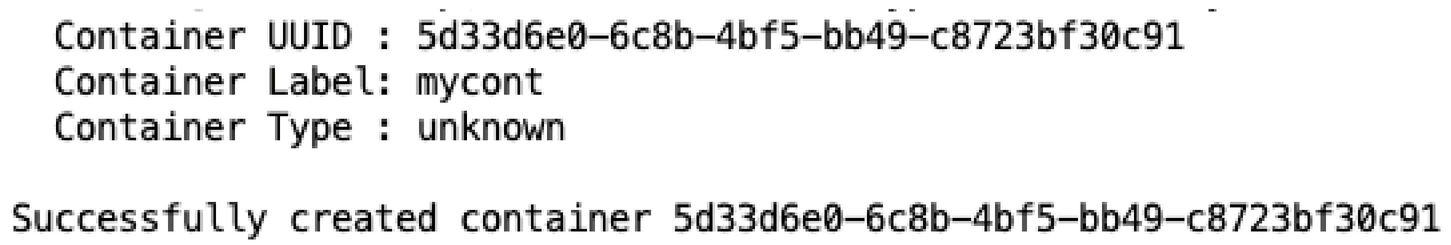
Connect to Pool

- First thing you usually do in your program is initialize DAOS and connect to your pool:
- int daos_init(void);
- int daos_pool_connect(const char *pool, const char *sys, unsigned int flags, daos_handle_t *poh, daos_pool_info_t *info, daos_event_t *ev);
- When finished, disconnect from your pool and finalize daos:
- int daos_pool_disconnect(daos_handle_t poh, daos_event_t *ev);
- int daos_fini(void);
- In an MPI program, consider connecting only from 1 client, and sharing the pool handle (poh) using the pool l2g, g2l functions:
  - daos_pool_local2global
  - daos_pool_global2local

Creating a Container

- Using the daos tool:

```
daos cont create mypool mycont
```



```
   Container UUID : 5d33d6e0-6c8b-4bf5-bb49-c8723bf30c91
   Container Label: mycont
   Container Type : unknown

Successfully created container 5d33d6e0-6c8b-4bf5-bb49-c8723bf30c91
```

- Using the API:

- `int daos_cont_create_with_label (daos_handle_t poh, const char *label, daos_prop_t *cont_prop, uuid_t *uuid,  daos_event_t *ev);`
- `int daos_cont_destroy(daos_handle_t poh, const char *cont, int force, daos_event_t *ev);`

Accessing a Container

- Need to open a container to access object in it:
- `int daos_cont_open(daos_handle_t poh, const char *cont, unsigned int flags, daos_handle_t *coh, daos_cont_info_t *info, daos_event_t *ev);`
- Close container when done:
- `int daos_cont_close(daos_handle_t coh, daos_event_t *ev);`
- In an MPI program, consider opening only from 1 client, and sharing the container handle (coh) using the cont l2g, g2l functions:
  - daos_cont_local2global
  - daos_cont_global2local

Recap Program Flow

```c
#include <daos.h>
int main(int argc, char **argv)
{
        daos_handle_t   poh, coh;

        daos_init();
        daos_pool_connect("mypool", NULL, DAOS_PC_RW, &poh, NULL, NULL);
        daos_cont_create_with_label(poh, "mycont", NULL, NULL, NULL);
        daos_cont_open(poh, "mycont", DAOS_COO_RW, &coh, NULL, NULL);

        /** do things */

        daos_cont_close(coh, NULL);
        daos_pool_disconnect(poh, NULL);
        daos_fini();
        return 0;
}
```

DAOS Object Types

- DAOS Object Types:
  - DAOS KV (put, get, list, remove)
    - 1 string key, 1 opaque value
  - DAOS ARRAY
    - 1D array of fixed sized value
  - DAOS Multi-Level KV (lower level)
- Object ID 128-bit space:
  - Lower 96 bits set by user
    - Unique OID allocator available in API for convenience
  - OID Embeds:
    - Object type
    - Object class (redundancy level and type – Replication, EC, None)

DAOS KV Object

- KV store API that provides:
  - Put, Get, Remove, List
- API:

- `int daos_kv_open(daos_handle_t coh, daos_obj_id_t oid, unsigned int mode, daos_handle_t *oh, daos_event_t *ev);`
- `int daos_kv_put(daos_handle_t oh, daos_handle_t th, uint64_t flags, const char *key, daos_size_t size, const void *buf, daos_event_t *ev);`
- `int daos_kv_get(daos_handle_t oh, daos_handle_t th, uint64_t flags, const char *key, daos_size_t *size, void *buf, daos_event_t *ev);`
- `int daos_kv_remove(daos_handle_t oh, daos_handle_t th, uint64_t flags, const char *key, daos_event_t *ev);`
- `int daos_kv_list(daos_handle_t oh, daos_handle_t th, uint32_t *nr,daos_key_desc_t *kds, d_sg_list_t *sgl, daos_anchor_t *anchor, daos_event_t *ev);`
- `int daos_kv_close(daos_handle_t oh, daos_event_t *ev);`
- `int daos_kv_destroy(daos_handle_t oh, daos_handle_t th, daos_event_t *ev);`

KV Conditional Operations

- By default, KV put/get operations do not check "existence" of key before operations:
  - Put on a key overwrites the value of that key
  - Get of a key does not fail if key does not exist, just returns 0 size.
  - Remove of a key does not fail if key does not exist.
- One can use conditional flags to achieve a different behavior:
- `DAOS_COND_KEY_INSERT`: Insert a key if it doesn't exist, fail if it does.
- `DAOS_COND_KEY_UPDATE`: Update a key if it exists, fail if it doesn't.
- `DAOS_COND_KEY_GET`: Get key value if it exists, fail if it doesn't.
- `DAOS_COND_KEY_REMOVE`: Remove a key if it exists, fail if it doesn't.

KV put/get example

```
/** init, connect, cont_open */

oid.hi = 0;
oid.lo = 1;
daos_obj_generate_oid(coh, &oid, DAOS_OF_KV_FLAT, 0, 0, 0);
daos_kv_open(coh, oid, DAOS_OO_RW, &kv, NULL);


/** set val buffer and size */
daos_kv_put(kv, DAOS_TX_NONE, 0, "key1", val_len1, val_buf1, NULL);
daos_kv_put(kv, DAOS_TX_NONE, 0, "key2", val_len2, val_buf2, NULL);

/** to fetch, can query the size first if not known */
daos_kv_get(kv, DAOS_TX_NONE, 0, "key1", &size, NULL, NULL);
get_buf = malloc (size);
daos_kv_get(kv, DAOS_TX_NONE, 0, "key1", &size, get_buf, NULL);
daos_kv_close(kv, NULL);

/** free buffer, cont_close, disconnect, finalize */
```

KV list example

```
/** enumerate keys in the KV */
daos_anchor_t   anchor = {0};
d_sg_list_t     sgl;
d_iov_t         sg_iov;

/** size of buffer to hold as many keys in memory */
buf = malloc(ENUM_DESC_BUF);
d_iov_set(&sg_iov, buf, ENUM_DESC_BUF);
sgl.sg_nr               = 1;
sgl.sg_nr_out           = 0;
sgl.sg_iovs             = &sg_iov;
```

```
daos_key_desc_t kds[ENUM_DESC_NR];

while (!daos_anchor_is_eof(&anchor)) {
  /** how many keys to attempt to fetch in one call */
  uint32_t       nr = ENUM_DESC_NR;


  memset(buf, 0, ENUM_DESC_BUF);
  daos_kv_list(kv, DAOS_TX_NONE, &nr, kds, &sgl,
               &anchor, NULL);

  if (nr == 0)
    continue;
  /** buf now container nr keys */
  /* kds arrays has length of each key */
}
```

DAOS Array Object

- Array object to manage records:
  - 1 Dimensional
- Array Management API:
- `int daos_array_create(daos_handle_t coh, daos_obj_id_t oid, daos_handle_t th, daos_size_t cell_size, daos_size_t chunk_size, daos_handle_t *oh, daos_event_t *ev);`
- `int daos_array_open(daos_handle_t coh, daos_obj_id_t oid, daos_handle_t th, unsigned int mode, daos_size_t *cell_size, daos_size_t *chunk_size, daos_handle_t *oh, daos_event_t *ev);`
- `int daos_array_close(daos_handle_t oh, daos_event_t *ev);`
- `int daos_array_destroy(daos_handle_t oh, daos_handle_t th, daos_event_t *ev);`

DAOS Array Access API

- Reading & writing record to an Array:
- int daos_array_read(daos_handle_t oh, daos_handle_t th, daos_array_iod_t *iod, d_sg_list_t *sgl, daos_event_t *ev);
- int daos_array_write(daos_handle_t oh, daos_handle_t th, daos_array_iod_t *iod, d_sg_list_t *sgl, daos_event_t *ev);
- int daos_array_get_size(daos_handle_t oh, daos_handle_t th, daos_size_t *size, daos_event_t *ev);
- int daos_array_set_size(daos_handle_t oh, daos_handle_t th, daos_size_t size, daos_event_t *ev);
- int daos_array_get_attr(daos_handle_t oh, daos_size_t *chunk_size, daos_size_t *cell_size);

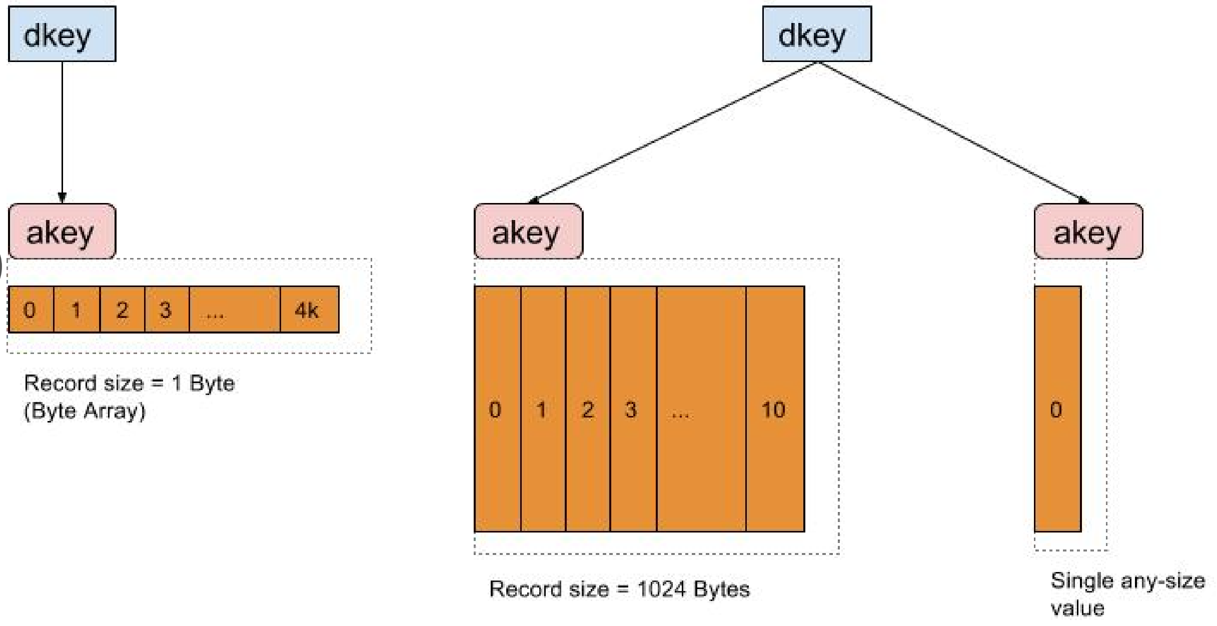
intel.

DAOS Array example

```
/** create array - if array exists just open it */
daos_array_create(coh, oid, DAOS_TX_NONE, 1, 1048576, &array, NULL);


daos_array_iod_t iod;
d_sg_list_t      sgl;
daos_range_t     rg;
d_iov_t          iov;


/** set array location */
iod.arr_nr = 1; /** number of ranges / array iovec */
rg.rg_len = BUFLEN; /** length */
rg.rg_idx = rank * BUFLEN; /** offset */
iod.arr_rgs = &rg;


/** set memory location, each rank writing BUFLEN */
sgl.sg_nr = 1;
d_iov_set(&iov, buf, BUFLEN);
sgl.sg_iovs = &iov;


daos_array_write(array, DAOS_TX_NONE, &iod, &sgl, NULL);
daos_array_read(array, DAOS_TX_NONE, &iod, &sgl, NULL);
daos_array_close(array, NULL);
```

DAOS Multi-Level KV Object

- 2 level keys:
  - Distribution Key - Dkey (collocate all entries under it), holds multiple akeys
  - Attribute Key - Akey (lower level to address records)
  - Both are opaque (support any size / type)
- Value types (under akey):
  - Single value: one blob (traditional value in KV store)
  - Array value:
    - 1 record size per akey
    - Array of records that can be updates via different extents / iovec



- **Intentionally very flexible, rich API; but at the expense of higher complexity for the regular user.**

Object Management API

- int daos_obj_open(daos_handle_t coh, daos_obj_id_t oid, unsigned int mode, daos_handle_t *oh, daos_event_t *ev);
- int daos_obj_close(daos_handle_t oh, daos_event_t *ev);
- int daos_obj_punch(daos_handle_t oh, daos_handle_t th, uint64_t flags, daos_event_t *ev);
- int daos_obj_punch_dkeys(daos_handle_t oh, daos_handle_t th, uint64_t flags, unsigned int nr, daos_key_t *dkeys, daos_event_t *ev);
- int daos_obj_punch_akeys(daos_handle_t oh, daos_handle_t th, uint64_t flags, daos_key_t *dkey, unsigned int nr, daos_key_t *akeys, daos_event_t *ev);

Object IO API

- `int daos_obj_update(daos_handle_t oh, daos_handle_t th, uint64_t flags, daos_key_t *dkey, unsigned int nr, daos_iod_t *iods, d_sg_list_t *sgls, daos_event_t *ev);`

```
daos_key_t iod_name; /* akey */
daos_iod_type_t iod_type; /* value type (single value or array value) */
daos_size_t iod_size; /* value or record size */
uint32_t iod_nr; /* number of extents (1 for SV) *?
daos_recx_t *iod_recxs; /* array of extents - offset, length pairs */
```

```
uint32_t sg_nr;
uint32_t sg_nr_out;
d_iov_t *sg_iovs;
```

- `int daos_obj_fetch(daos_handle_t oh, daos_handle_t th, uint64_t flags, daos_key_t *dkey, unsigned int nr, daos_iod_t *iods, d_sg_list_t *sgls, daos_iom_t *ioms, daos_event_t *ev);`

Object Enumerate API

```
int daos_obj_list_dkey(daos_handle_t oh, daos_handle_t
th, uint32_t *nr, daos_key_desc_t *kds, d_sg_list_t
*sgl, daos_anchor_t *anchor, daos_event_t *ev);

int daos_obj_list_akey(daos_handle_t oh, daos_handle_t
th, daos_key_t *dkey, uint32_t *nr, daos_key_desc_t
*kds, d_sg_list_t *sgl, daos_anchor_t *anchor,
daos_event_t *ev);
```

DAOS Object Update Example

```
daos_obj_open(coh, oid, DAOS_OO_RW, &oh, NULL);
d_iov_set(&dkey, "dkey1", strlen("dkey1"));

d_iov_set(&sg_iov, buf, BUFLEN);
sgl[0].sg_nr = 1;
sgl[0].sg_iovs = &sg_iov;
sgl[1].sg_nr = 1;
sgl[1].sg_iovs = &sg_iov;

d_iov_set(&iod[0].iod_name, "akey1", strlen("akey1"));
d_iov_set(&iod[1].iod_name, "akey2", strlen("akey2"));

iod[0].iod_nr = 1;
iod[0].iod_size = BUFLEN;
iod[0].iod_recxs = NULL;
iod[0].iod_type = DAOS_IOD_SINGLE;

iod[1].iod_nr = 1;
iod[1].iod_size = 1;
recx.rx_nr = BUFLEN;
recx.rx_idx = 0;
iod[1].iod_recxs = &recx;
iod[1].iod_type = DAOS_IOD_ARRAY;

daos_obj_update(oh, DAOS_TX_NONE, 0, &dkey, 2, &iod, &sgl, NULL);
```

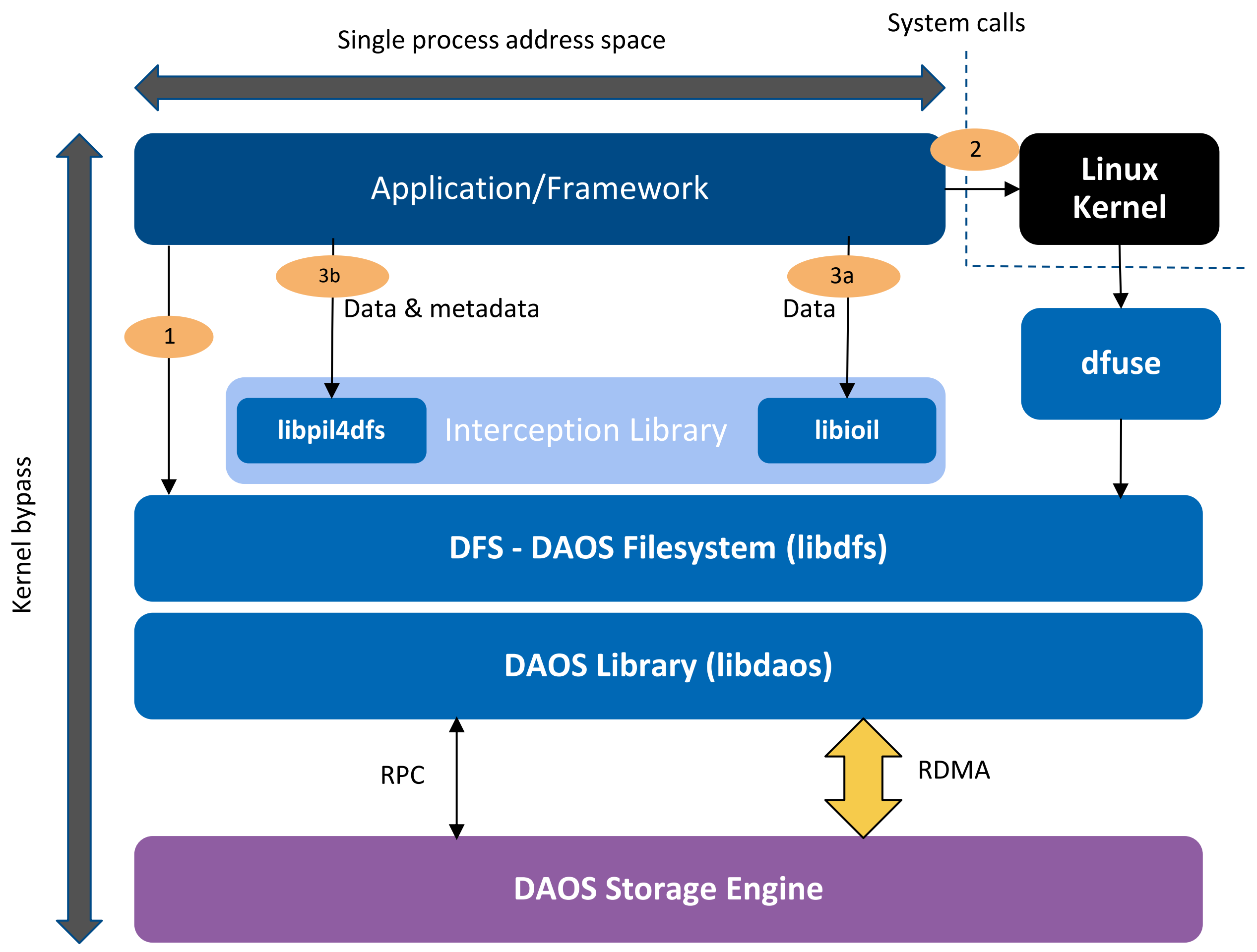
DAOS Object Fetch Example

```
daos_obj_open(coh, oid, DAOS_OO_RW, &oh, NULL);
d_iov_set(&dkey, "dkey1", strlen("dkey1"));

d_iov_set(&sg_iov, buf, BUFLEN);
sgl[0].sg_nr = 1;
sgl[0].sg_iovs = &sg_iov;
sgl[1].sg_nr = 1;
sgl[1].sg_iovs = &sg_iov;

d_iov_set(&iod[0].iod_name, "akey1", strlen("akey1"));
d_iov_set(&iod[1].iod_name, "akey2", strlen("akey2"));

iod[0].iod_nr = 1;
iod[0].iod_size = BUFLEN; /** if size is not known, use DAOS_REC_ANY and NULL sgl */
iod[0].iod_recxs = NULL;
iod[0].iod_type = DAOS_IOD_SINGLE;

iod[1].iod_nr = 1;
iod[1].iod_size = 1; /** if size is not known, use DAOS_REC_ANY and NULL sgl */
recx.rx_nr = BUFLEN;
recx.rx_idx = 0;
iod[1].iod_recxs = &recx;
iod[1].iod_type = DAOS_IOD_ARRAY;

daos_obj_fetch(oh, DAOS_TX_NONE, 0, &dkey, 2, &iod, &sgl, NULL, NULL);
```

More examples

- https://github.com/daos-stack/daos/blob/master/src/tests/simple_obj.c

POSIX Support & Interception



1 Userspace DFS library with API like POSIX
- **Require** application changes
- Low latency & high concurrency
- No caching

2 DFUSE daemon to support POSIX API
- **No** application changes
- VFS mount point & high latency
- Caching by Linux kernel

3 DFUSE + Interception library
- **No** application changes
- 2 flavors using LD_PRELOAD

3a libioil
- (f)read/write interception
- Metadata via dfuse

3b libpil4dfs
- Data & metadata interception
- Aim at delivering same performance as #1 w/o any application change
- Mmap & binary execution via fuse

How to use DFS?

- You should have access to a pool (identified by a string label).
- Create a POSIX container with the daos tool:
  - `daos cont create mypool mycont --type=POSIX`
  - Alternatively, you can programmatically create a container to use directly in your application (if you are using DFS and changing your app).
- Open the DFS mount:
  - `dfs_connect (mypool, mycont, O_RDWR, .. &dfs);`
  - `dfs_disconnect (dfs);`

DFS API

POSIX	DFS
mkdir(), rmdir()	dfs_mkdir(), dfs_rmdir()
open(), close(), access()	dfs_open(), dfs_release(),dfs_lookup()
pwritev(), preadv()	dfs_read/write()
{set,get,list,remove}xattr()	dfs_{set,get,list,remove}xattr
stat(), fstat()	dfs_stat(),ostat()
readdir() …	dfs_readdir() …

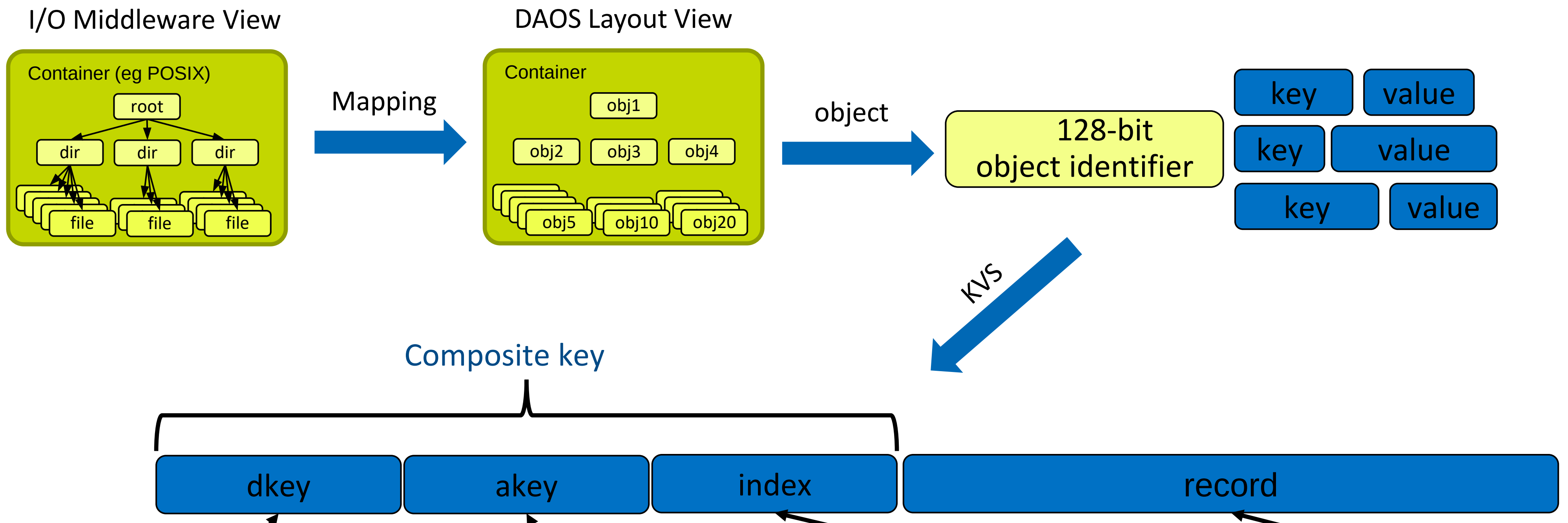
- Mostly 1-1 mapping from POSIX API to DFS API.
- Instead of File & Directory descriptors, use DFS objects.
- All calls need the DFS mount which is usually done once initialization.

DFUSE

- To mount an existing POSIX container with dfuse, run the following command:
  - `dfuse mypool mycont -m /mnt/dfuse`
  - No one can access your container / mountpoint unless access is provided on the pool and container (through ACLs).
- Now you have a parallel file system under /mnt/dfuse on all nodes where that is mounted
  - Access files / directories as any namespace in the container, and applications can run without any modifications (the easy path).
- Interception Libraries:
  - This library works in conjunction with dfuse and allow to interception of POSIX I/O calls and issue the I/O operations directly from the application context through libdaos without any application changes.
  - This provides kernel-bypass for I/O. To use this set the LD_PRELOAD to point to the shared library in the DOAS install dir
    - LD_PRELOAD=/path/to/daos/install/lib/libioil.so or libpil4dfs.so

Hands On

DAOS Data Model: Objects



Distribution key
Variable length
Integer, string, data structure, …
Hashed/distributed across engines

Attribute key
Variable length
Integer, string, data structure, …
Co-located on same engines
All akeys can be retrieved in a single RPC

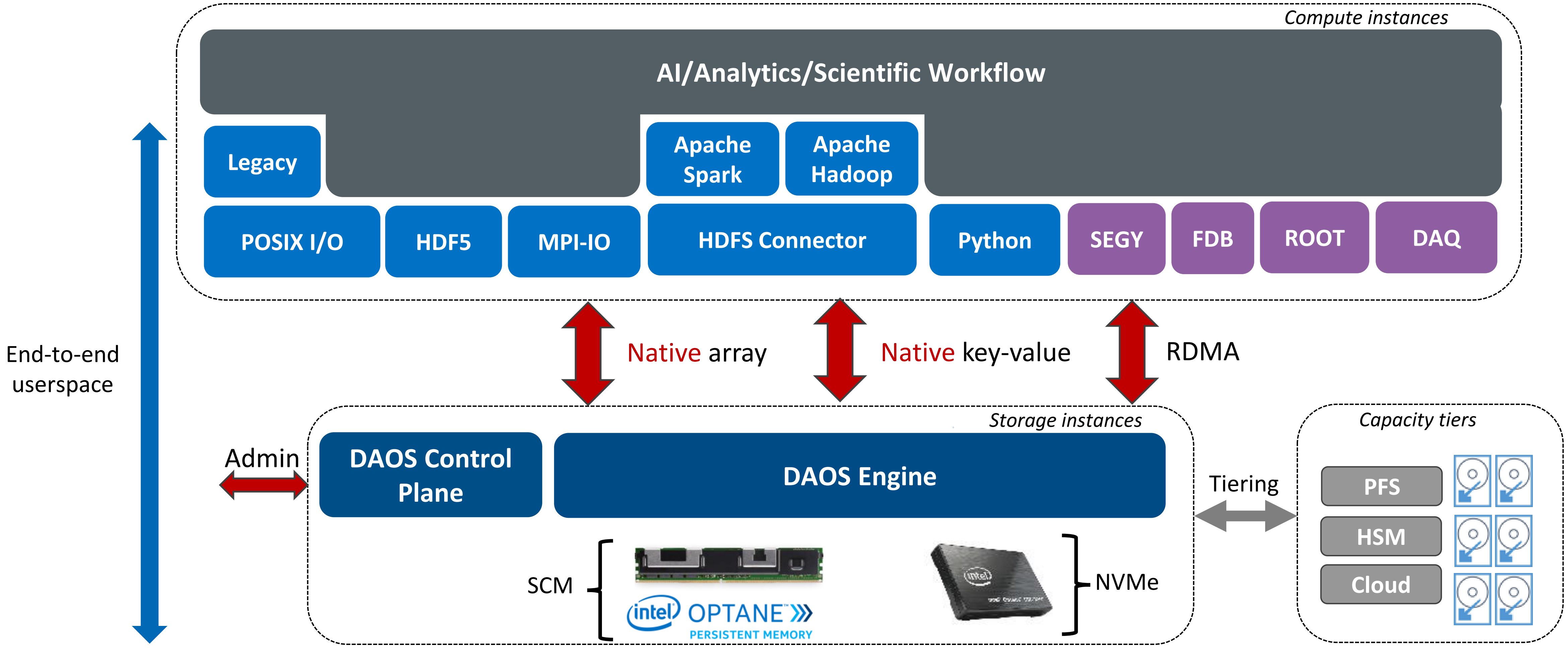
Index
64-bit Integer
Address array of value
Support operation over range of indices (= extent)

Record
Variable length value
From 1 byte to GB's

DAOS Ecosystem

🟦 Generic I/O middleware supported today

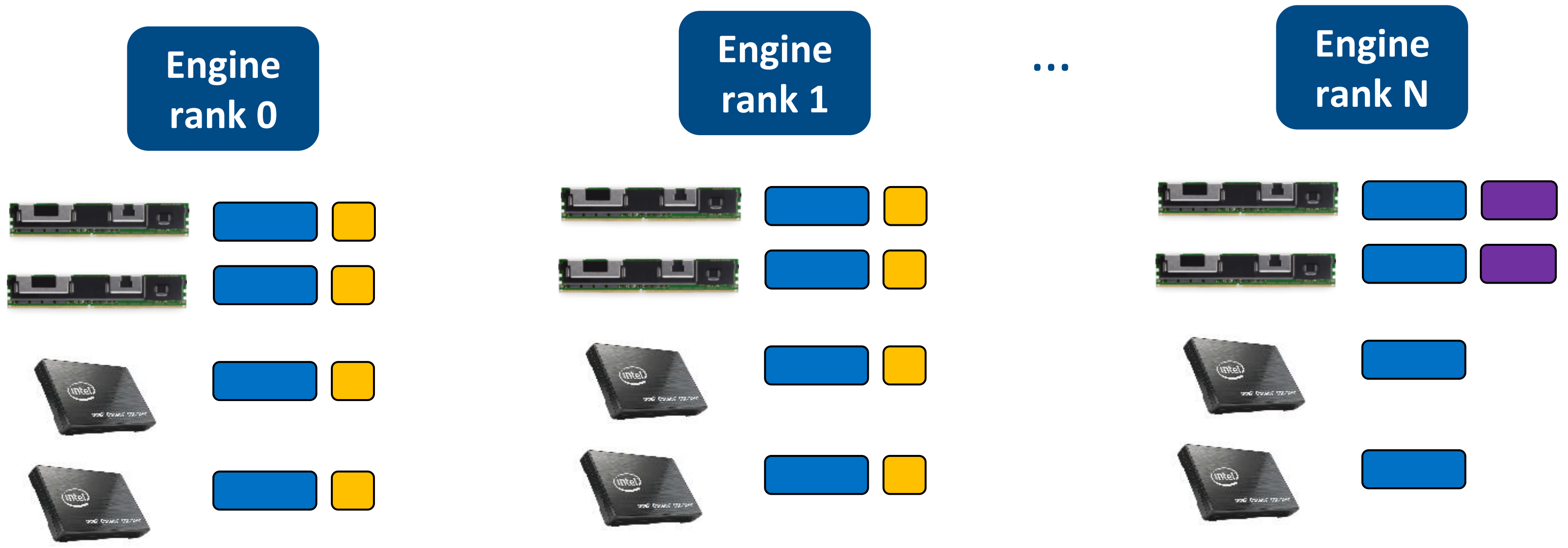
🟪 Domain-specific data models under development in co-design with partners



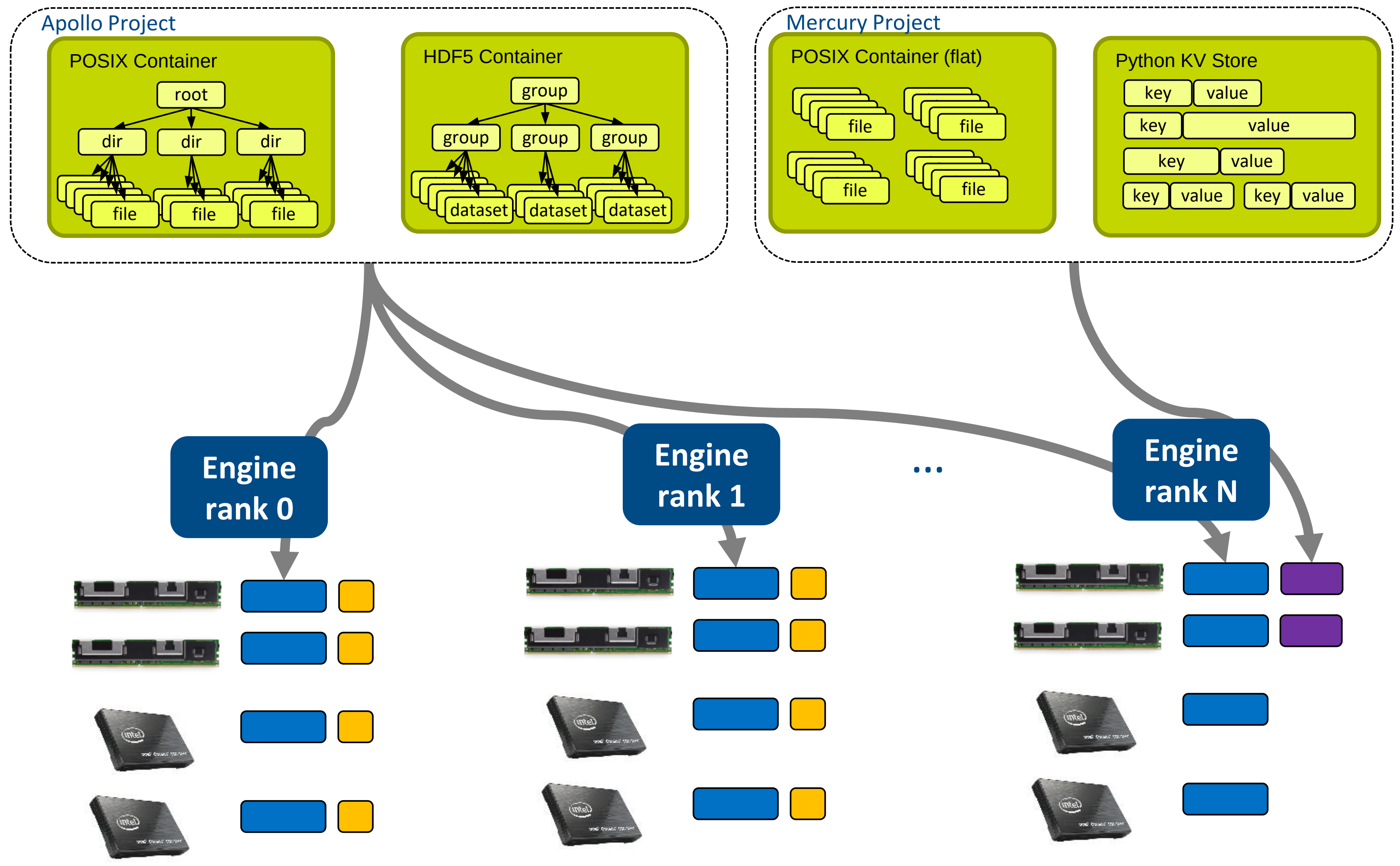
DAOS Data Model: Pools

Example:

Pool 1	🟦	Project Apollo	100PB usable	20TB/s	200M IOPS
Pool 2	🟨	Project Gemini	10PB usable	2TB/s	20M IOPS
Pool 3	🟪	Project Mercury	30TB usable	80GB/s	2M IOPS



DAOS Data Model: Container



Transactions

- Open/close transaction
  - Open returns transaction handle to use in object operations
  - Close just frees the handle (does not commit)
- Commit/abort/restart transaction
- Conflict detection:
  - WR, RW, WW conflicts

```
        daos_tx_open(coh, &th, ...);
restart:
        daos_obj_fetch(..., th, ...);
        daos_obj_update(..., th, ...);
        daos_obj_fetch(..., th, ...);
        daos_obj_update(..., th, ...);
        daos_obj_dkey_punch(..., th, ...);
        rc = daos_tx_commit(th, ...);
        if (rc == -DER_RESTART) {
                daos_tx_restart(th, ...);
                goto restart;
        }
        daos_tx_close(th, ...);
```