Hewlett Packard
Enterprise

Tutorial on the DAOS API

Kenneth Cain, Mohamad Chaarawi, HPE

May 4, 2025

DAOS User API

- Pools:
  - Connect, disconnect, query
- Containers:
  - Create, destroy, open, close, query, properties, attributes
- Objects:
  - Access APIs based on type (KVS, global array, multi-level KVS)
- POSIX API:
  - DFS + dfuse

DAOS API Usage, Program 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 (not covered in this tutorial)
- Close container & disconnect from pool

Program Flow: Connect to a Pool

- First (typically): initialize DAOS, 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);
```

Note: pool already exists

Administrator used daos management utility “dmg” to create a pool – e.g.,

```
dmg pool create
--size=10TB mypool
```

- MPI program: connect from 1 rank, serialize handle, then share with MPI

```
int daos_pool_local2global(daos_handle_t poh, d_iov_t *glob)
int daos_pool_global2local(daos_handle_t poh, d_iov_t *glob)
```

- Last, disconnect from your pool, finalize DAOS:

```
int daos_pool_disconnect(daos_handle_t poh, daos_event_t *ev);
int daos_fini(void);
```

Program Flow: Connect to a Pool

- First (typically): initialize DAOS, 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);
```

- MPI program: connect from 1 rank, serialize handle, then share with MPI

```
int daos_pool_local2global(daos_handle_t poh, d_iov_t *glob)
int daos_pool_global2local(daos_handle_t poh, d_iov_t *glob)
```

- Last, disconnect from your pool, finalize DAOS:

```
int daos_pool_disconnect(daos_handle_t poh, daos_event_t *ev);
int daos_fini(void);
```

const char *pool: label string

daos_handle_t: opaque handle type
- "poh" - "pool open handle"
- "coh" – "container open handle"
- "oh" – "object open handle"

daos_pool_info_t:
- capacity, free space, (im)balance
- Health, rebuild state
- Also output by `daos_pool_query()`

d_iov_t:
- Refers to a contiguous app buffer

daos_event_t: (not covered here)
- Asynchronous API invoke/test

Program Flow: Create 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
```

User admin tool 'daos'

API: input poh from pool_connect

Container also has a label id string

daos_prop_t: properties

- Label
- Type (POSIX, HDF5, untyped)
- Redundancy Factor (RF)

- 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, /* ev */);
```

Program Flow: Access 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, /* ev */);
```

- MPI program: connect from 1 rank, serialize handle, then share with MPI:

```
int daos_cont_local2global(daos_handle_t poh, d_iov_t *glob)
int daos_cont_global2local(daos_handle_t poh, d_iov_t *glob)
```

- Close container when done:

```
int daos_cont_close(daos_handle_t coh, daos_event_t *ev);
```

daos_cont_info_t:

- Pool UUID
- Container UUID
- # container open handles
- Metadata open/close/modify times
- RF
- …
- Also output by `daos_cont_query()`

Program Flow: Recap

```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);

        /** perform object I/O - presented next */

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

DAOS Object Types and Classes and IDs

- DAOS Object Types:
  - DAOS KV – Each item having 1 string key, 1 opaque value
    - Operations: put, get, list, remove
    - Entire value collocated / atomic on 1 target
  - DAOS ARRAY – 1D array of fixed-size value
    - Operations: read, write, get/set size
    - cell_size: single array value size
    - chunk_size: controls striping of array regions/chunks across targets
  - DAOS Multi-Level KV – lower-level API
    - Operations: update, fetch, list
    - dkey – distribution key
    - akey – attribute key
- Object ID 128-bit space:
  - Lower 96 bits set by user
    - Unique OID allocator available in API for convenience
  - Upper 32 bits set by daos. OID Embeds:
    - Object type
    - Object class (redundancy level and type – Replication, EC, None)

DAOS Object Types and Classes and IDs

- DAOS Object Types:
  - DAOS KV – Each item having 1 string key, 1 opaque value
    - Operations: put, get, list, remove
    - Entire value collocated / atomic on 1 target
  - DAOS ARRAY – 1D array of fixed-size value
    - Operations: read, write, get/set size
    - cell_size: single array value size
    - chunk_size: controls striping of array regions/chunks across targets
  - DAOS Multi-Level KV – lower-level API
    - Operations: update, fetch, list
    - dkey – distribution key
    - akey – attribute key
- Object ID 128-bit space:
  - Lower 96 bits set by user
    - Unique OID allocator available in API for convenience
  - Upper 32 bits set by daos. OID Embeds:
    - Object type
    - Object class (redundancy level and type – Replication, EC, None)

Sample Object Types (`enum daos_otype_t`)

```
/** flat KV (no akey) with hashed dkey */
    DAOS_OT_KV_HASHED,
/** Array, attributes provided by user */
    DAOS_OT_ARRAY_ATTR,
/** multi-level KV with uint64 [ad]keys */
    DAOS_OT_MULTI_UINT64,
```

DAOS Object Types and Classes and IDs

- DAOS Object Types:
  - DAOS KV – Each item having 1 string key, 1 opaque value
    - Operations: put, get, list, remove
    - Entire value collocated / atomic on 1 target
  - DAOS ARRAY – 1D array of fixed-size value
    - Operations: read, write, get/set size
    - cell_size: single array value size
    - chunk_size: number of cells placed together in a storage target
  - DAOS Multi-Level KV – lower-level API
    - Operations: update, fetch, list
    - dkey – distribution key
    - akey – attribute key
- Object ID 128-bit space:
  - Lower 96 bits set by user
    - Unique OID allocator available in API for convenience
  - Upper 32 bits set by daos. OID Embeds:
    - Object type (e.g., KV, Array, Multi-Level KV)
    - Object class (data protection kind and degree – Replication, EC, None)

Sample Object Types (`enum daos_otype_t`)

```
/** flat KV (no akey) with hashed dkey */
    DAOS_OT_KV_HASHED,
/** Array, attributes provided by user */
    DAOS_OT_ARRAY_ATTR,
/** multi-level KV with uint64 [ad]keys */
    DAOS_OT_MULTI_UINT64,
```

Sample Object Classes (`daos_oclass_id_t`)

```
/* Explicit layout, no data protection
 * Examples: OC_S1, OC_S2, …, OC_S32, OC_SX
 * S1 : shards=1, S2 shards=2, SX shards=all tgts
 */

/* Replicated object (OC_RP_), explicit layout:
 * <number of replicas> G<redundancy groups>
 * Ex OC_RP_2G1, 2G2..32 2GX, 3G1..32 3GX, …
 * 2G1 : 2 replicas group=1
 * 3G2 : 3 replicas groups=2, ...
 * 6GX : 6 replicas, spread across all targets
 */

/* Erasure coded (OC_EC_), explicit layout:
 * <data_cells>P<parity_cells>G<redun_groups>
 * Ex: EC_8P2G1, EC_8P2G<2..32>, EC_8P2GX,
 *     EC_16P2G1, EC_16P2G<2..32>, EC_16P2GX,
 * - 8P2G2:   8+2 EC object, groups=2
 * - 16P2GX: 16+2 EC object, all targets in pool
 * - 2P1G1:   2+1 EC object, group=1
 * - 4P2G8:   4+2 EC object, groups=2
 */
```

DAOS Object Types and Classes and IDs

- Object ID 128-bit space (Lower 96 user; upper 32 daos):
  - Object type (e.g., KV, Array, Multi-Level KV)
  - Object class (Replication, EC, None)

```
int daos_obj_generate_oid(
        daos_handle_t coh,
        daos_obj_id_t *oid,
        enum daos_otype_t type,
        daos_oclass_id_t cid,
        daos_oclass_hints_t hints, uint32_t args);

daos_obj_id_t oid;
oid.hi = 0;
oid.lo = 1;
daos_obj_generate_oid(coh, &oid,
DAOS_OF_KV_HASHED, OC_RP_2GX, 0, 0);
```

Sample Object Types (`enum daos_otype_t`)

```
/** flat KV (no akey) with hashed dkey */
    DAOS_OT_KV_HASHED,
/** Array, attributes provided by user */
    DAOS_OT_ARRAY_ATTR,
/** multi-level KV with uint64 [ad]keys */
    DAOS_OT_MULTI_UINT64,
```

Sample Object Classes (`daos_oclass_id_t`)

```
/* Explicit layout, no data protection
 * Examples: OC_S1, OC_S2, …, OC_S32, OC_SX
 * S1 : shards=1, S2 shards=2, SX shards=all tgts
 */

/* Replicated object (OC_RP_), explicit layout:
 * <number of replicas> G<redundancy groups>
 * Ex OC_RP_2G1, 2G2..32 2GX, 3G1..32 3GX, …
 * 2G1 : 2 replicas group=1
 * 3G2 : 3 replicas groups=2, ...
 * 6GX : 6 replicas, spread across all targets
 */

/* Erasure coded (OC_EC_), explicit layout:
 * <data_cells>P<parity_cells>G<redun_groups>
 * Ex: EC_8P2G1, EC_8P2G<2..32>, EC_8P2GX,
 *     EC_16P2G1, EC_16P2G<2..32>, EC_16P2GX,
 * - 8P2G2:   8+2 EC object, groups=2
 * - 16P2GX: 16+2 EC object, all targets in pool
 * - 2P1G1:   2+1 EC object, group=1
 * - 4P2G8:   4+2 EC object, groups=2
 */
```

DAOS KV Object – Management Operations

- Recall: KV store interface providing access operations: Put, Get, Remove, List

- Management 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_close(daos_handle_t oh, daos_event_t *ev);

int daos_kv_destroy(daos_handle_t oh, daos_handle_t th, /* ev */);
```

KV: string key → opaque/atomic value

API:

- input coh from daos_cont_open()
- Input oid from daos_obj_generate_oid()
- output object handle (oh)

DAOS KV Object – Access Operations

- Access API:

```
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, /* ev */);
```

API: input oh from kv_open()

Put/get/remove values (given string key)

List keys

- Key sizes in daos_key_desc_t *kds
- Key strings in sgl

KV Conditional Operations

- By default, KV put/get operations do not check “existence” of key before operations:
  - Put(key): overwrites the value
  - Get(key): does not fail if key does not exist, just returns 0 size.
  - Remove(key): does not fail if key does not exist.

- One can use conditional flags for different behavior:
- `DAOS_COND_KEY_INSERT`: Insert a key if it doesn't exist (fail if it already exists)
- `DAOS_COND_KEY_UPDATE`: Update a key if it exists, (fail if it does not exist)
- `DAOS_COND_KEY_GET`: Get key value if it exists, (fail if it does not exist).
- `DAOS_COND_KEY_REMOVE`: Remove a key if it exists (fail if it does not exist).

KV Put/Get – Example

```
/** daos_init, daos_pool_connect, daos_cont_open */

oid.hi = 0;
oid.lo = 1;
daos_obj_generate_oid(coh, &oid, DAOS_OF_KV_HASHED, OC_RP_2GX, 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, daos_cont_close, daos_pool_disconnect, daos_fini */
```

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_BYTES);
d_iov_set(&sg_iov, buf, ENUM_DESC_BUF_BYTES);
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_BYTES);
  daos_kv_list(kv, DAOS_TX_NONE, &nr, kds, &sgl,
               &anchor, NULL);

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

DAOS Array Object – Management Operations

- 1-Dimensional Array object to manage records
  - cell_size: single array value size (bytes)
  - chunk_size: number of cells placed together in a storage target –controls striping of array regions across targets

- 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, /* 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 Operations

- 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);
```

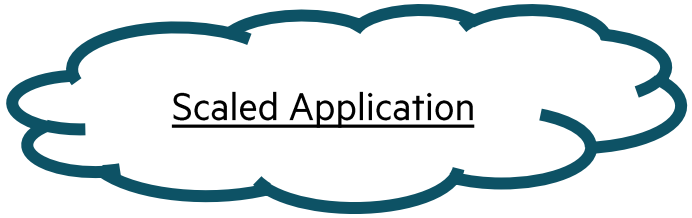
- Misc

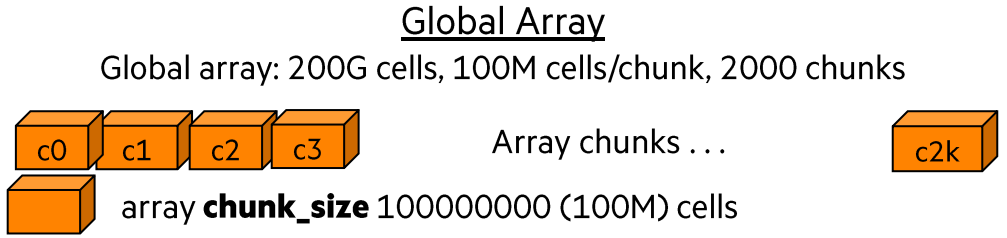
```
int daos_array_get_size(daos_handle_t oh, daos_handle_t th, daos_size_t *size, …);
int daos_array_set_size(daos_handle_t oh, daos_handle_t th, daos_size_t size, …);
int daos_array_get_attr(daos_handle_t oh, daos_size_t *chunk_size,
                        daos_size_t *cell_size);
```

DAOS Array – Example

```
/** create array - 1 byte/cell, NCELLS=100 million cells per chunk */
daos_array_create(coh, oid, DAOS_TX_NONE, 1, 100000000, &array, NULL);
```

`

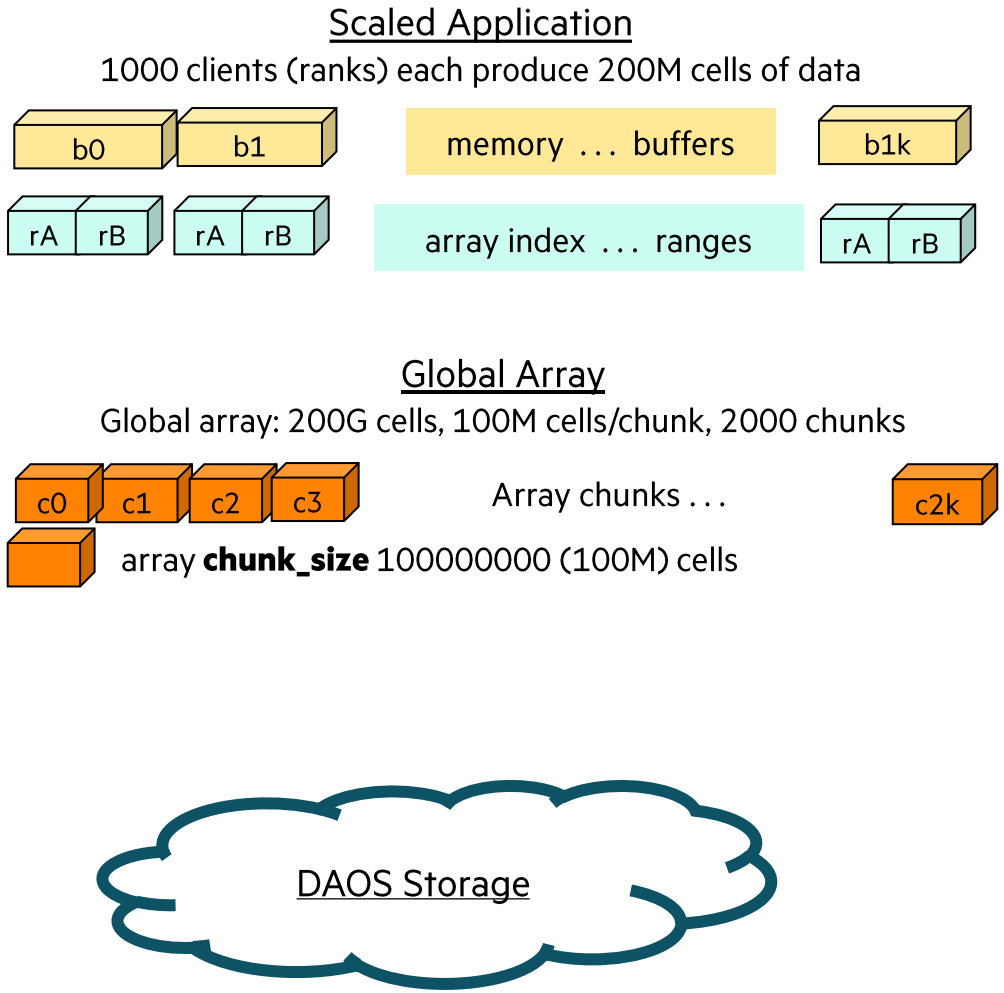




DAOS Array – Example

```
/** create array - 1 byte/cell, NCELLS=100 million cells per chunk */
daos_array_create(coh, oid, DAOS_TX_NONE, 1, 100000000, &array, NULL);

d_sg_list_t      sgl;    /* memory: scatter/gather list of iovecs */
d_iov_t          iov;    /* memory (iovec): 1 buffer (ptr, bytes) */
daos_array_iod_t iod;    /* array IO descriptor - array ranges */
daos_range_t     rgs[2]; /* array ranges(start index, num cells) */

/** set memory location, each rank writing BUFLEN */
sgl.sg_nr = 1;                  /** one memory buffer iovec */
d_iov_set(&iov, buf, BUFLEN); /** one buffer/ptr, BUFLEN=200M bytes */
sgl.sg_iovs = &iov;

/** specify this client's particular array (sub)ranges */
iod.arr_nr    = 2;                /** two array (sub)ranges */
iod.arr_rgs   = rgs;              /** the list of two ranges */
ra_start      = rank * NCELLS*2;  /** array ranges start indices */
rb_start      = ra_start + NCELLS;
rgs[0].rg_idx = ra_start;         /** (and rgs[1] from rb_start) */
rgs[0].rg_len = NCELLS;           /** length (and rgs[1] len=NCELLS) */
```

`

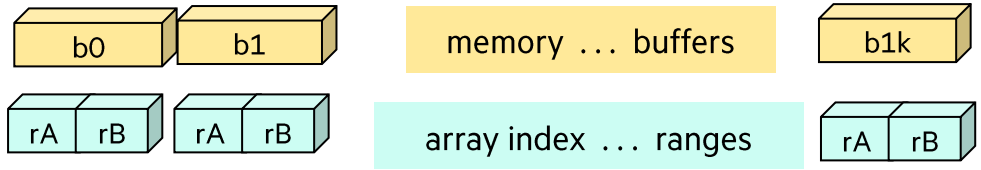


DAOS Array – Example

```
/** create array - 1 byte/cell, NCELLS=100,000,000 cells per chunk */
daos_array_create(coh, oid, DAOS_TX_NONE, 1, NCELLS, &array, NULL);

d_sg_list_t      sgl;    /* memory: scatter/gather list of iovecs */
d_iov_t          iov;    /* memory (iovec): 1 buffer (ptr, bytes) */
daos_array_iod_t iod;    /* array IO descriptor - array ranges */
daos_range_t     rgs[2]; /* array ranges(start index, num cells) */

/** set memory location, each rank writing BUFLEN */
sgl.sg_nr = 1;                  /** one memory buffer iovec */
d_iov_set(&iov, buf, BUFLEN); /** one buffer/ptr, BUFLEN=200M bytes */
sgl.sg_iovs = &iov;

/** specify this client's particular array (sub)ranges */
iod.arr_nr    = 2;                /** two array (sub)ranges */
iod.arr_rgs   = rgs;              /** the list of two ranges */
ra_start      = rank * NCELLS*2;  /** array ranges start indices */
rb_start      = ra_start + NCELLS;
rgs[0].rg_idx = ra_start;         /** (and rgs[1] from rb_start) */
rgs[0].rg_len = NCELLS;           /** length (and rgs[1] len=NCELLS) */

/** write array data to DAOS storage, and read back */
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);
```

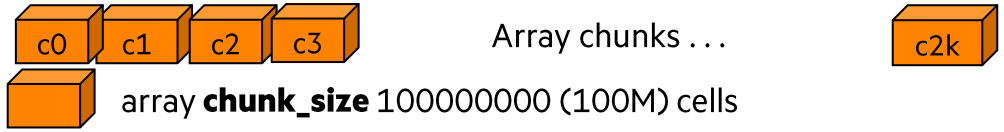
Scaled Application

1000 clients (ranks) each produce 200M cells of data



Global Array

Global array: 200G cells, 100M cells/chunk, 2000 chunks

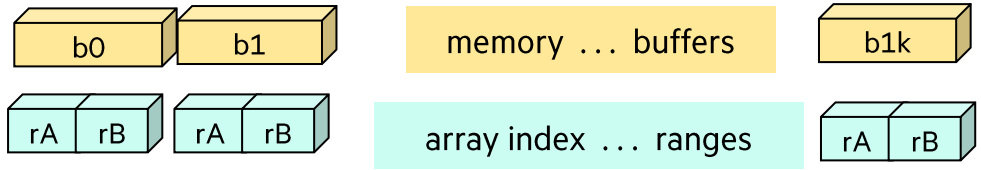


DAOS Array – Example

```
/** create array - 1 byte/cell, NCELLS=100 million cells per chunk */
daos_array_create(coh, oid, DAOS_TX_NONE, 1, 100000000, &array, NULL);

d_sg_list_t      sgl;    /* memory: scatter/gather list of iovecs */
d_iov_t          iov;    /* memory (iovec): 1 buffer (ptr, bytes) */
daos_array_iod_t iod;    /* array IO descriptor - array ranges */
daos_range_t     rgs[2]; /* array ranges(start index, num cells) */

/** set memory location, each rank writing BUFLEN */
sgl.sg_nr = 1;                  /** one memory buffer iovec */
d_iov_set(&iov, buf, BUFLEN); /** one buffer/ptr, BUFLEN=200M bytes */
sgl.sg_iovs = &iov;

/** specify this client's particular array (sub)ranges */
iod.arr_nr    = 2;                /** two array (sub)ranges */
iod.arr_rgs   = rgs;              /** the list of two ranges */
ra_start      = rank * NCELLS*2;  /** array ranges start indices */
rb_start      = ra_start + NCELLS;
rgs[0].rg_idx = ra_start;         /** (and rgs[1] from rb_start) */
rgs[0].rg_len = NCELLS;           /** length (and rgs[1] len=NCELLS) */

/** write array data to DAOS storage, and read back */
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);
```

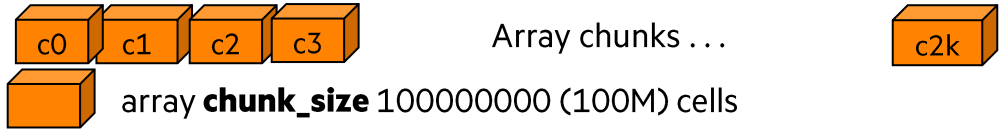
Scaled Application

1000 clients (ranks) each produce 200M cells of data



Global Array

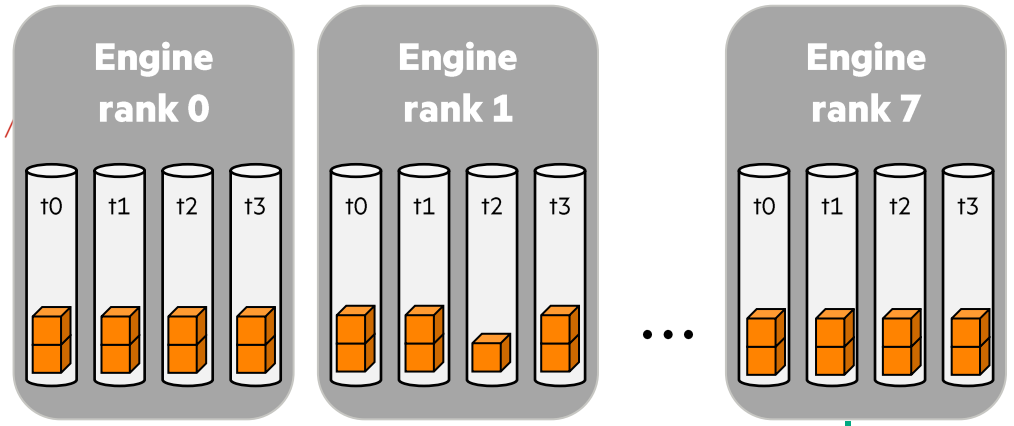
Global array: 200G cells, 100M cells/chunk, 2000 chunks



DAOS Storage

Stored in <num_daos_engines_in_pool> x tgts/daos_server

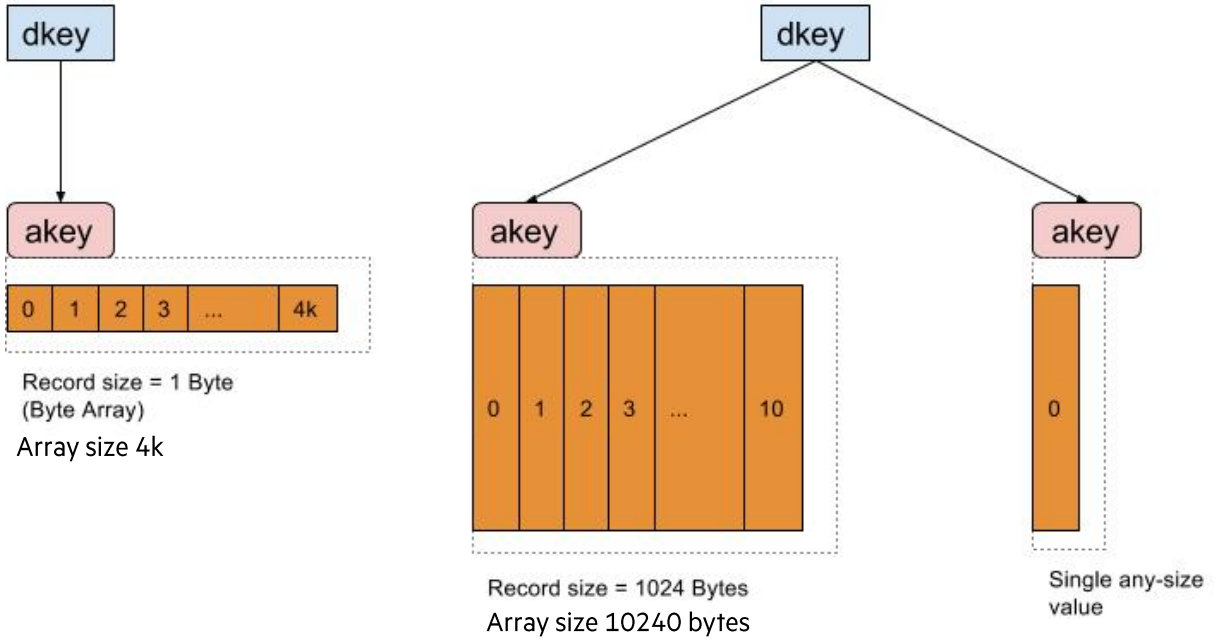
Ex: 8 servers x 4 tgts/server = 32 targets



Multi-Level KV Object

- Two-level key:
  - 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 a single akey):
  - Single value: one blob (traditional value in KV store)
  - Array value:
    - Array of fixed-size cells (records) that can be updated in a fine-grained manner via different range extents
    - This is different than a DAOS (global/distributed) array



- **Intentionally very flexible, rich API**
- **(at the expense of higher complexity for the typical user)**

Multi-Level KV Object – Management Operations

```
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, /* 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, …);
```

API:

- input coh from daos_cont_open()
- Input oid from daos_obj_generate_oid()
- output object handle (oh)

Multi-Level KV Object – Access Operations (Update, Fetch)

```
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;     /* SV: value size, array: record size */
uint32_t iod_nr;          /* SV: 1, array: number of record extents */
daos_recx_t *iod_recxs;   /* SV: NULL, array: (offset, length) pairs */
      uint64_t rx_idx, rx_nr;
```

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

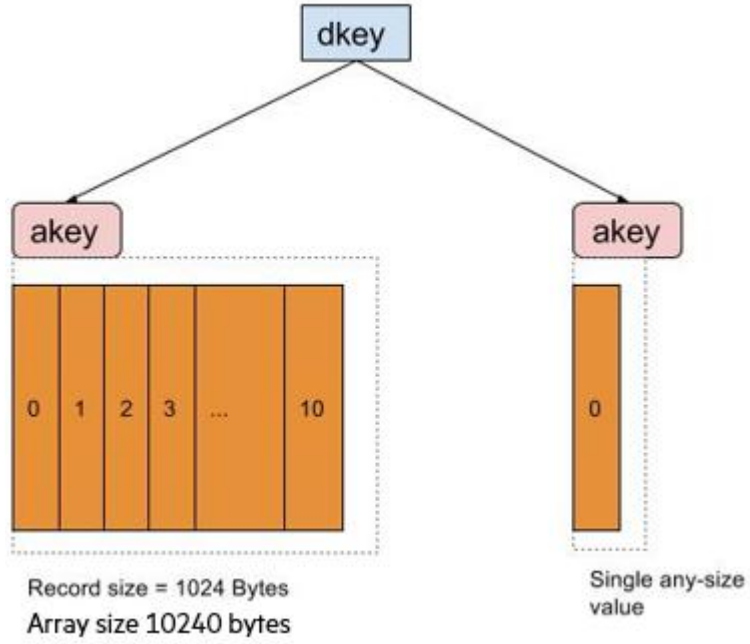
```
int daos_obj_fetch(daos_handle_t oh, daos_handle_t th, uint64_t flags,
      daos_key_t *dkey, unsigned int nr, daos_iod_t *iods,
      d_sg_list_t *sgls, daos_iom_t *ioms, daos_event_t *ev);
```

Multi-Level KV Object – Access Operations (List)

```
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,…);

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);
```

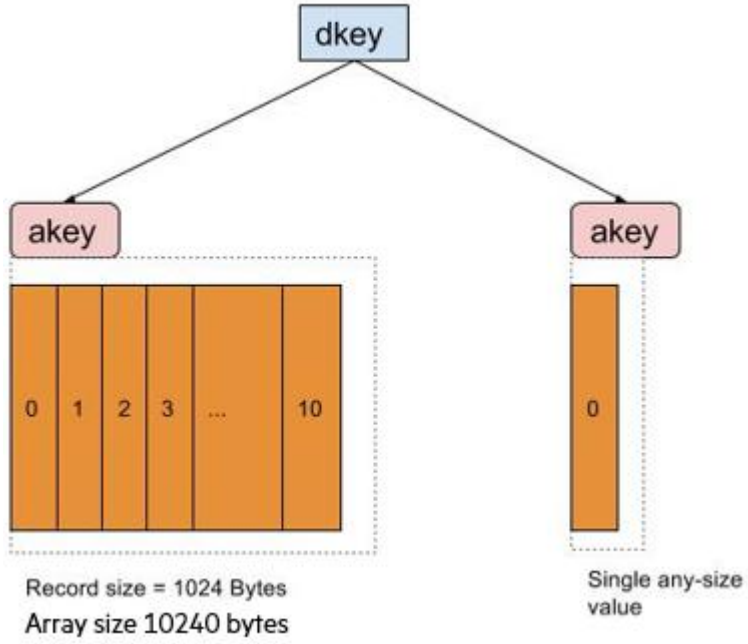
Multi-Level KV Object – Update Example

```
daos_obj_open(coh, oid, DAOS_OO_RW, &oh, NULL);

/* application buffers */
const char *buf2 = "single_value, my string";
d_iov_set(&sg_iovs[0], buf1, BUF1LEN);      /* 10240 byte array val: (dkey1,akey1) */
sgls[0].sg_nr = 1;
sgls[0].sg_iovs = &sg_iovs[0];
d_iov_set(&sg_iovs[1], buf2, strlen(buf2)); /* string val: (dkey1,akey2) */
sgls[1].sg_nr = 1;
sgls[1].sg_iovs = &sg_iovs[1];

/* keys */
d_iov_set(&dkey, "dkey1", strlen("dkey1"));
d_iov_set(&iods[0].iod_name, "akey1", strlen("akey1"));
d_iov_set(&iods[1].iod_name, "akey2", strlen("akey2"));

/* IODs for each akey */
iods[0].iod_type = DAOS_IOD_ARRAY;
iods[0].iod_size = 1;                   /* 1 byte/array cell */
recx.rx_idx = 0;                        /* array index range (0, BUF1LEN) */
recx.rx_nr = BUF1LEN;
iods[0].iod_nr = 1;
iods[0].iod_recxs = &recx;
iods[1].iod_type = DAOS_IOD_SINGLE;
iods[1].iod_size = strlen(buf2);
iods[1].iod_nr = 1;                     /* iod_recxs=NULL for SV */

daos_obj_update(oh, DAOS_TX_NONE, 0, &dkey, 2, &iods[0], &sgls[0], NULL);
```



Multi-Level KV Object – Fetch Example

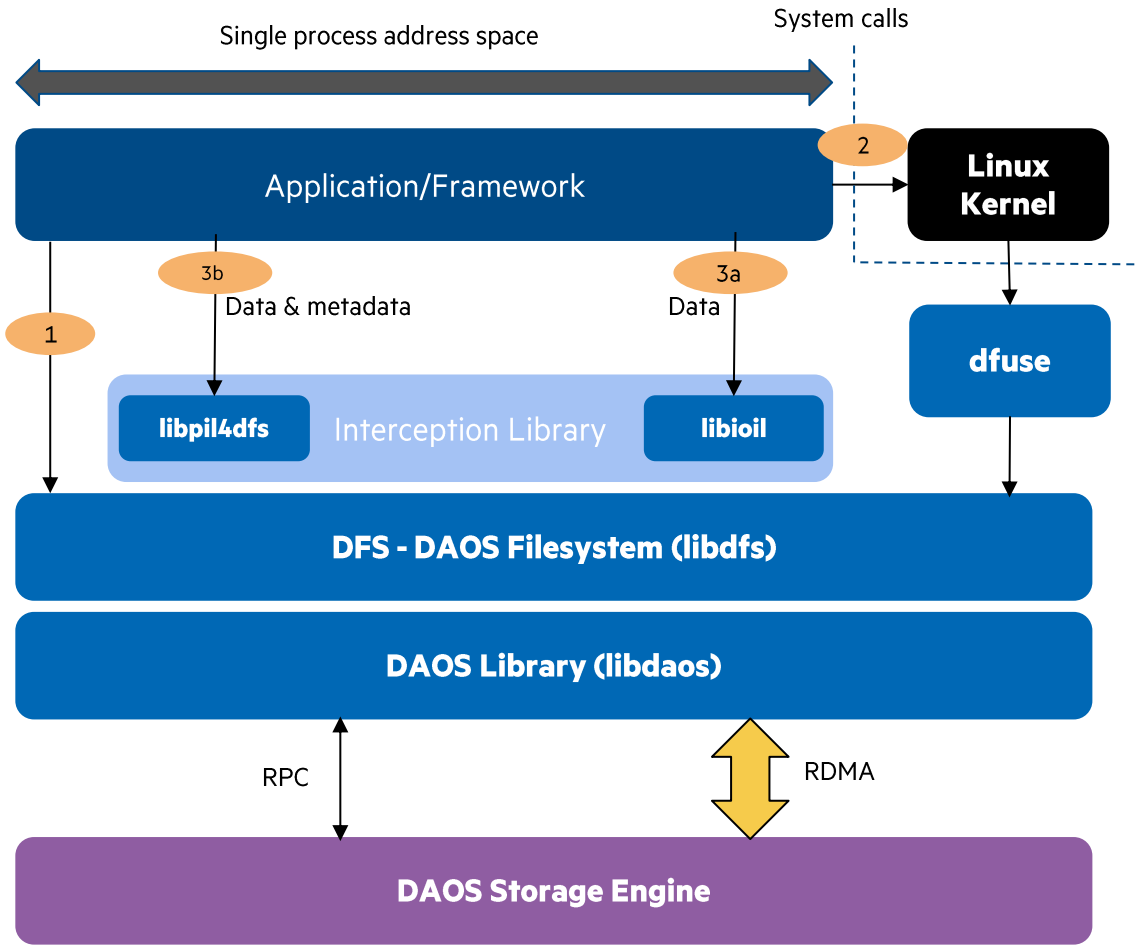
```
daos_obj_open(coh, oid, DAOS_OO_RW, &oh, NULL);

/* application buffers */
char rbuf2[128];
d_iov_set(&sg_iovs[0], rbuf1, BUF1LEN);     /* 10240 byte array val: (dkey1,akey1) */
sgls[0].sg_nr = 1;
sgls[0].sg_iovs = &sg_iovs[0];
d_iov_set(&sg_iovs[1], rbuf2, strlen(buf2)); /* string val: (dkey1,akey2) */
sgls[1].sg_nr = 1;
sgls[1].sg_iovs = &sg_iovs[1];

/* keys */
d_iov_set(&dkey, "dkey1", strlen("dkey1"));
d_iov_set(&iods[0].iod_name, "akey1", strlen("akey1"));
d_iov_set(&iods[1].iod_name, "akey2", strlen("akey2"));

/* IODs for each akey */
/** If iod_size is unknown: specify DAOS_REC_ANY, NULL sgl */
iods[0].iod_type = DAOS_IOD_ARRAY;
iods[0].iod_size = 1;                  /* 1 byte/array cell */
recx.rx_idx = 0;                       /* array index range (0, BUF1LEN) */
recx.rx_nr = BUF1LEN;
iods[0].iod_nr = 1;
iods[0].iod_recxs = &recx;
iods[1].iod_type = DAOS_IOD_SINGLE;
iods[1].iod_size = strlen(buf2);
iods[1].iod_nr = 1;                    /* iod_recxs=NULL for SV */

daos_obj_fetch(oh, DAOS_TX_NONE, 0, &dkey, 2, &iods[0], &sgls[0], 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

- **Requires** application changes
- Low-latency and high-concurrency
- No caching

2

DFUSE daemon to support POSIX API

- **No changes** to application
- VFS mount point and high-latency
- Caching done by Linux kernel

3

DFUSE + Interception Library

- **No changes** to application
- 2 flavors, choose with LD_PRELOAD

3a

- Libioil
  - Intercept data only - (f)read/write
  - Metadata handled by dfuse / kernel

3b

- Libpil4dfs
  - Intercept data and metadata
  - Aim to deliver same performance as DFS (#1)
  - mmap() and 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`
  - Or: use API to create a container to use in your application (if 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 (via 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
  - 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, set LD_PRELOAD to point to the shared library in the DAOS install dir –LD_PRELOAD=/path/to/daos/install/lib/libioil.so or libpil4dfs.so

Best Practices – Container Redundancy Factor (rd_fac) Property

- The number of (not yet rebuilt) concurrent failures container objects are protected against (without loss)
  - A number in the range 0-5

- Production systems recommendation: rd_fac:2 (which is a default value if not specified)
  - `daos cont create –type=POSIX –properties=rd_fac:2 <pool> <container>`

- Note: all objects must use a class with at least this degree of protection. Some legal examples:

	rd_fac:0	rd_fac:1	rd_fac:2
No Protection Classes	OC_S<*>	None	None
Replication Classes	Any	OC_RP_2G<*> OCP_RP_3G<*> …	OC_RP_3G<*> OC_RP_4G<*> …
Erasure Code Classes	Any	OC_EC_8P1G<*> OC_EC_16P1G<*>	OC_EC_8P2G<*> OC_EC_16P2G<*>

Best Practices – POSIX Container Data Protection / Redundancy Type

- Recall: data protection is part of an object's "object class" – None, Replication, or Erasure Code

- Erasure Code:
  - Best for large IO access patterns.
  - Full stripe write: 12%-33% lower performance (vs. no data protection).
  - Partial stripe write: 66% lower performance (vs. no data protection).
  - Read performance should be the same.
  - Not supported for directory objects

- Replication:
  - Best for metadata objects (directories) and small files (<= 16k).
  - Write IOPS: slower (than no data protection) by the number of replicas created.
  - Read IOPS: equal or better (than no data protection) – more shards to serve concurrent requests.

Best Practices – POSIX Container Files, Directories Object Classes

```
daos cont create –type=POSIX –dir-oclass=<OC> --file-oclass=<OC>
```

	rd_fac:0	rd_fac:1	rd_fac:2
Defaults - Widely-striped (“X”) objs for: - Large files (GBs), Lean dirs. (<10k ents) - Single-shared access, high BW required	File: SX Dir : S1	File: EC_16P1GX Dir : RP_2G1	File: EC_16P2GX Dir : RP_3G1
Small-stripe (1/2/4/16/32) objs for: - Something in-between huge and tiny files - File per process to large files.	File: S* (1/2/…/32) Dir : S1	File: EC_*P1_G* (G1/2/…/32) Dir : RP_2G1	File: EC_*P2_G* (G1/2/…/32) Dir : RP_3G1
One-stripe objs for: - tiny files, more IOPS required	File/Dir: S1	File/Dir: RP_2G1	File/Dir: RP_3G1

Recall: Sample Object Classes (`daos_oclass_id_t`)

```
/* Explicit layout, no data protection
 * Examples: OC_S1, OC_S2, …, OC_S32, OC_SX
 * S1 : shards=1, S2 shards=2, SX shards=all tgts
 */

/* Replicated object (OC_RP_), explicit layout:
 * <number of replicas> G<redundancy groups>
 * Ex OC_RP_2G1, 2G2..32 2GX, 3G1..32 3GX, …
 * 2G1 : 2 replicas group=1
 * 3G2 : 3 replicas groups=2, ...
 * 6GX : 6 replicas, spread across all targets
 */

/* Erasure coded (OC_EC_), explicit layout:
 * <data_cells>P<parity_cells>G<redun_groups>
 * Ex: EC_8P2G1, EC_8P2G<2..32>, EC_8P2GX,
 *     EC_16P2G1, EC_16P2G<2..32>, EC_16P2GX,
 * - 8P2G2:   8+2 EC object, groups=2
 * - 16P2GX: 16+2 EC object, all targets in pool
 * - 2P1G1:   2+1 EC object, group=1
 * - 4P2G8:   4+2 EC object, groups=2
 */
```

Best Practices – POSIX Container Files, Directories Object Classes – Details

```
daos cont create –type=POSIX –dir-oclass=<OC> --file-oclass=<OC>
```

	rd_fac:0	rd_fac:1	rd_fac:2
Defaults - Widely-striped ("X") objs for: - Large files (GBs), Lean dirs. (<10k ents) - Single-shared access, high BW required Tradeoffs: - If used with file-per-proc, non-scalable pool connect slows pool service. - Slow file stat(), remove, directory listing – RPC to all engines, and query all targets.	File: SX Dir : S1	File: EC_16P1GX Dir : RP_2G1	File: EC_16P2GX Dir : RP_3G1
Small-stripe (1/2/4/16/32) objs for: - Something in-between huge and tiny files - File per process to large files. Tradeoffs: - Faster stat() and directory listing - Limited bandwidth to number of targets - Benchmarking file create/remove/stat could benefit from widely-striped dir objs.	File: S* (1/2/…/32) Dir : S1	File: EC_*P1_G* (G1/2/…/32) Dir : RP_2G1	File: EC_*P2_G* (G1/2/…/32) Dir : RP_3G1
One-stripe objs for: - tiny files, more IOPS required	File/Dir: S1	File/Dir: RP_2G1	File/Dir: RP_3G1

Recall: Sample Object Classes (`daos_oclass_id_t`)

```
/* Explicit layout, no data protection
 * Examples: OC_S1, OC_S2, …, OC_S32, OC_SX
 * S1 : shards=1, S2 shards=2, SX shards=all tgts
 */

/* Replicated object (OC_RP_), explicit layout:
 * <number of replicas> G<redundancy groups>
 * Ex OC_RP_2G1, 2G2..32 2GX, 3G1..32 3GX, …
 * 2G1 : 2 replicas group=1
 * 3G2 : 3 replicas groups=2, ...
 * 6GX : 6 replicas, spread across all targets
 */

/* Erasure coded (OC_EC_), explicit layout:
 * <data_cells>P<parity_cells>G<redun_groups>
 * Ex: EC_8P2G1, EC_8P2G<2..32>, EC_8P2GX,
 *     EC_16P2G1, EC_16P2G<2..32>, EC_16P2GX,
 * - 8P2G2:   8+2 EC object, groups=2
 * - 16P2GX: 16+2 EC object, all targets in pool
 * - 2P1G1:   2+1 EC object, group=1
 * - 4P2G8:   4+2 EC object, groups=2
 */
```

Best Practices – POSIX Container EC Settings for Performance

- DFS Chunk Size, default 1MiB (`daos container create –chunk_size=`)
  - DAOS splits file data across dkeys in chunk size units

- `ec_cell_sz` container property
  - DAOS splits application buffer into `ec_cell_sz` byte parts (16 parts for EC_16P2, 8 parts for EC_8P2, etc.)

- Full (vs. Partial) Stripe Write – application buffer chunk is an even multiple of `ec_cell_sz` (or not).
  - Full stripe write is more efficient

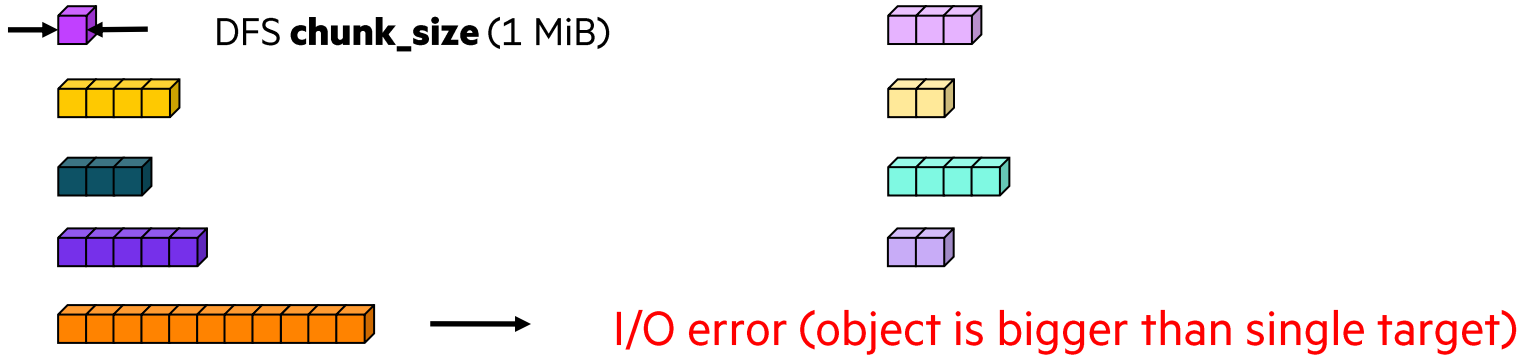
EC Object Class	EC Cell Size	DFS Chunk size	Full or Partial Write?
16P2	128k	1m	Partial: 1m gets divided into 8 128k data parts which < 16 data shards of the object class used
8P2	128k	1m	Full
16P2	128k	2m, 4m, 8m, etc	Full
16P2	256k	2m	Partial
16P2	256k	4m, 8m, etc.	Full

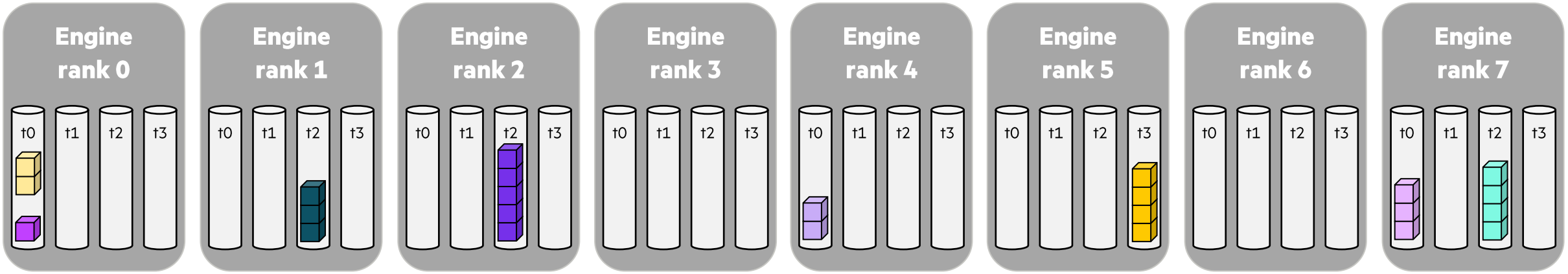
Thank you

kenneth.cain@hpe.com
mohamad.chaarawi@hpe.com

Data Distribution and Data Protection

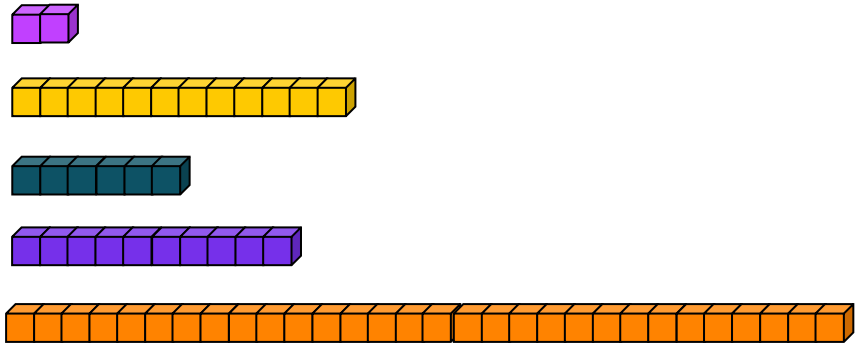
Sharding, object class S1

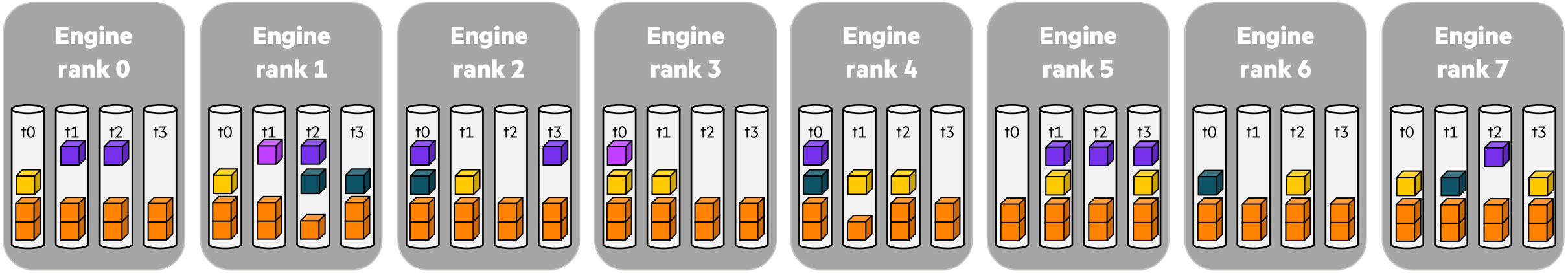




Data Distribution and Data Protection

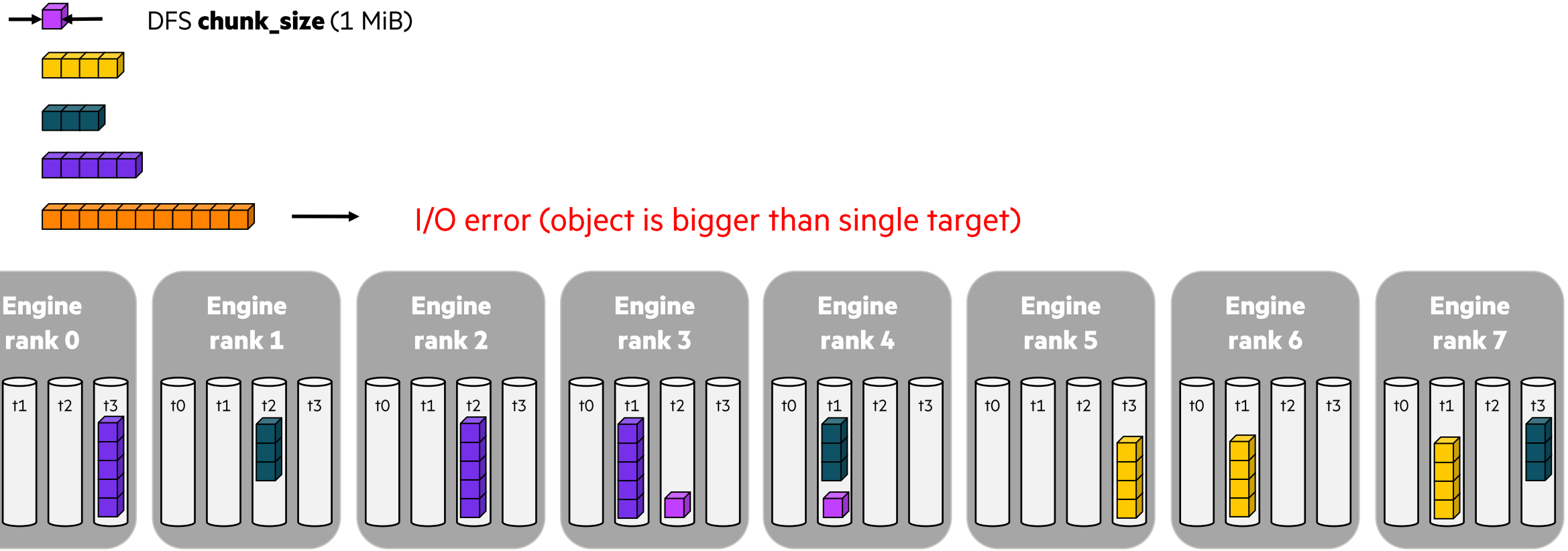
Sharding, object class SX





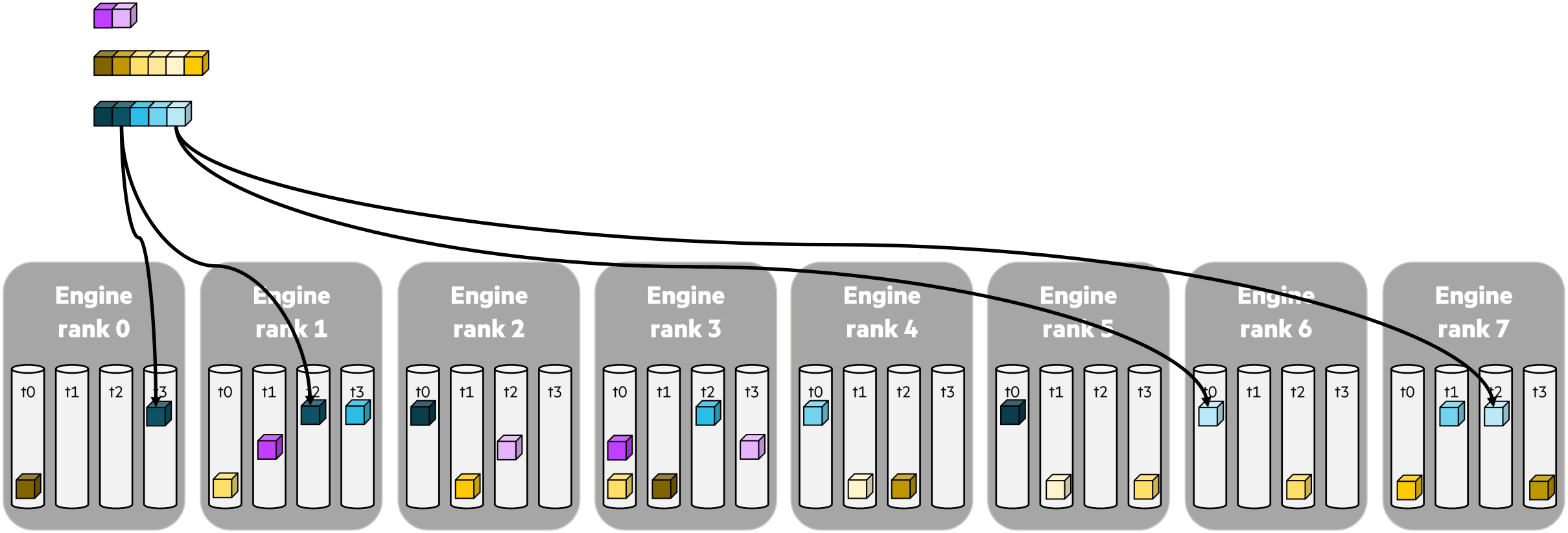
Data Distribution and Data Protection

Replication, object class RP_3G1 (rf_lvl=engine)



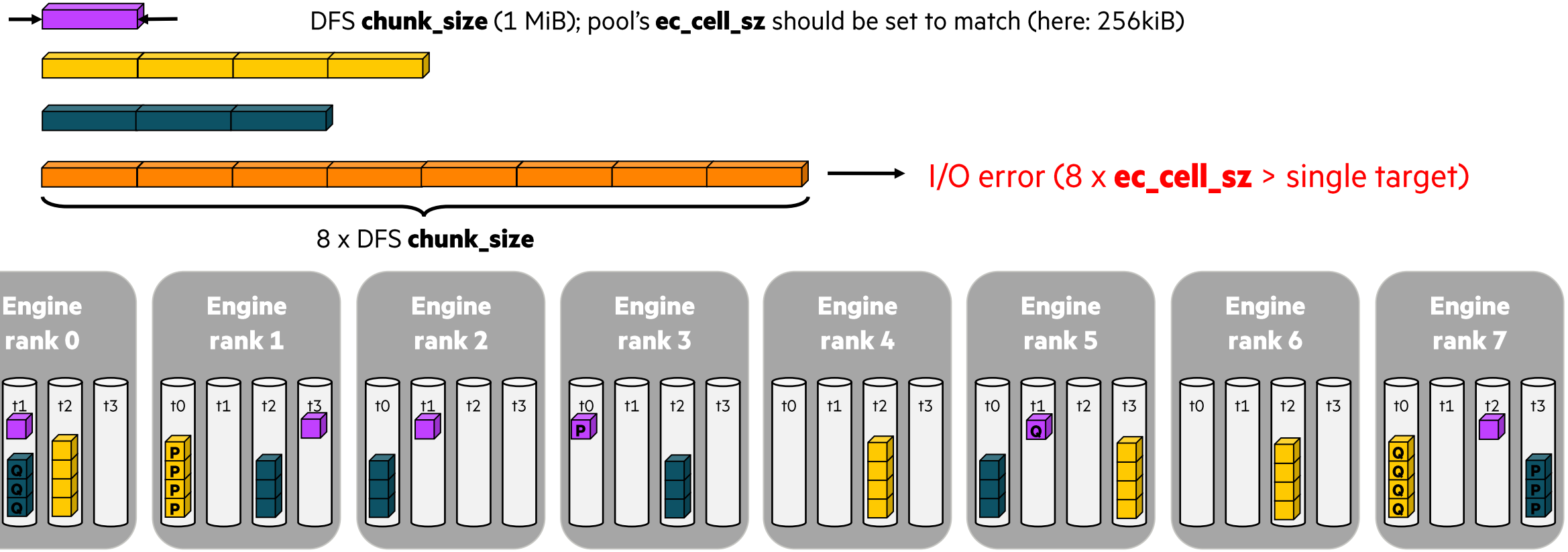
Data Distribution and Data Protection

Replication, object class RP_2GX (rf_lvl=engine)



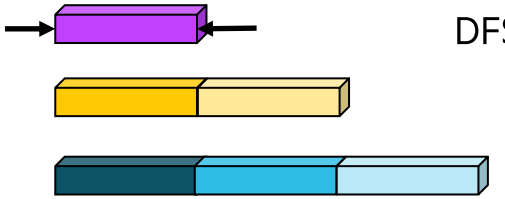
Data Distribution and Data Protection

Erasure Coding, object class EC_4P2G1 (rf_lvl=engine)

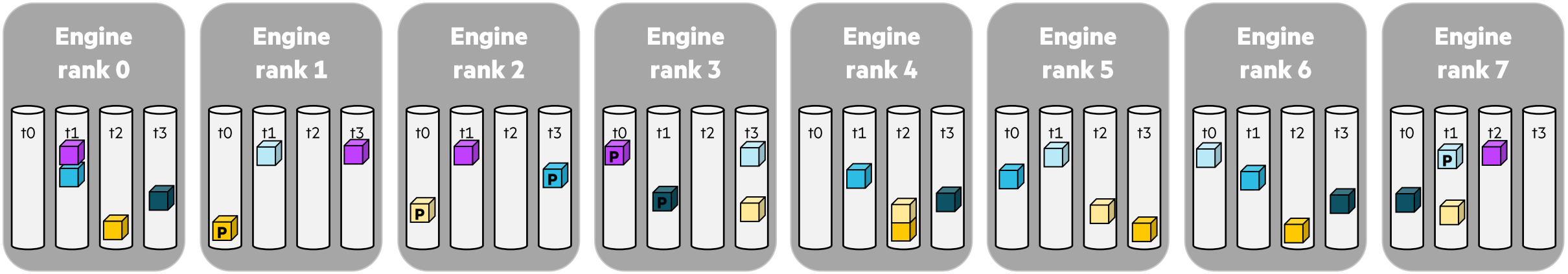


Data Distribution and Data Protection

Erasure Coding, object class EC_4P1GX (rf_lvl=engine)



DFS **chunk_size** (1 MiB); pool's **ec_cell_sz** should be set to match (here: 256kiB)



DAOS Object - Old 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);
```

Multi-Level KV Object – Old 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);
sgls[0].sg_nr = 1;
sgls[0].sg_iovs = &sg_iov;
sgls[1].sg_nr = 1;
sgls[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, &iods[0], &sgls[0], NULL, NULL);
```