Exploring Object Stores: RNTuple DAOS Backend

Javier López-Gómez – **CERN**
<javier.lopez.gomez@cern.ch>

SFT group meeting, 7th December 2020

ROOT project,
EP-SFT, CERN

http://root.cern/

Contents

Introduction

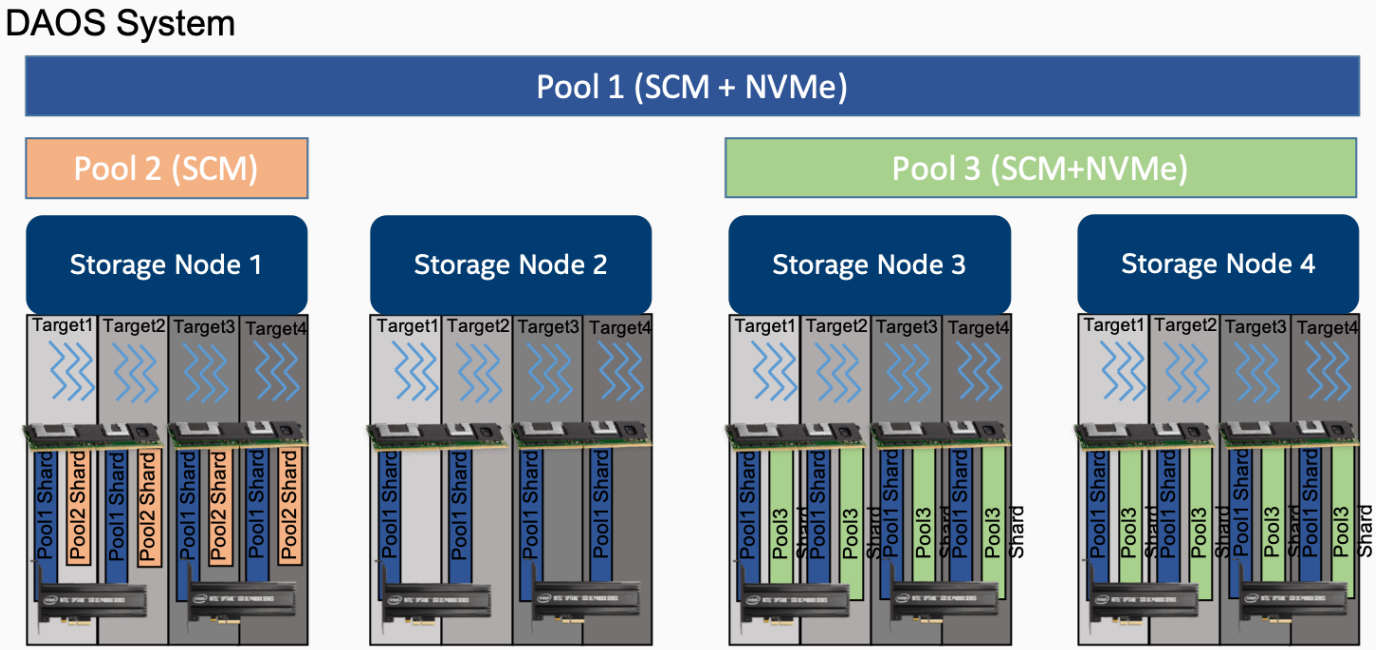
About object storage

- File system: stores metadata and data of file hierarchy (typically on a block device).
- Object storage: manages data as objects ($\sim$ user-defined metadata + data, identified by a UUID). Decouples namespace operations from data read/write.
- Similar to a key–value storage where the key is a UUID, but specifically tuned for high workloads.

Intel DAOS: Overview

- Foundation of the Intel exascale storage stack.
- High bandwidth, high IOPS, low latency, fault-tolerant. Supports distributed transactions.
- Overcomes POSIX I/O limitations, e.g. block layer and I/O scheduling (typically designed for rotating disks).
- Optimal use of Intel Optane DC persistent memory and NVMe SSD (access times in the order of μs).

Intel DAOS: architecture overview

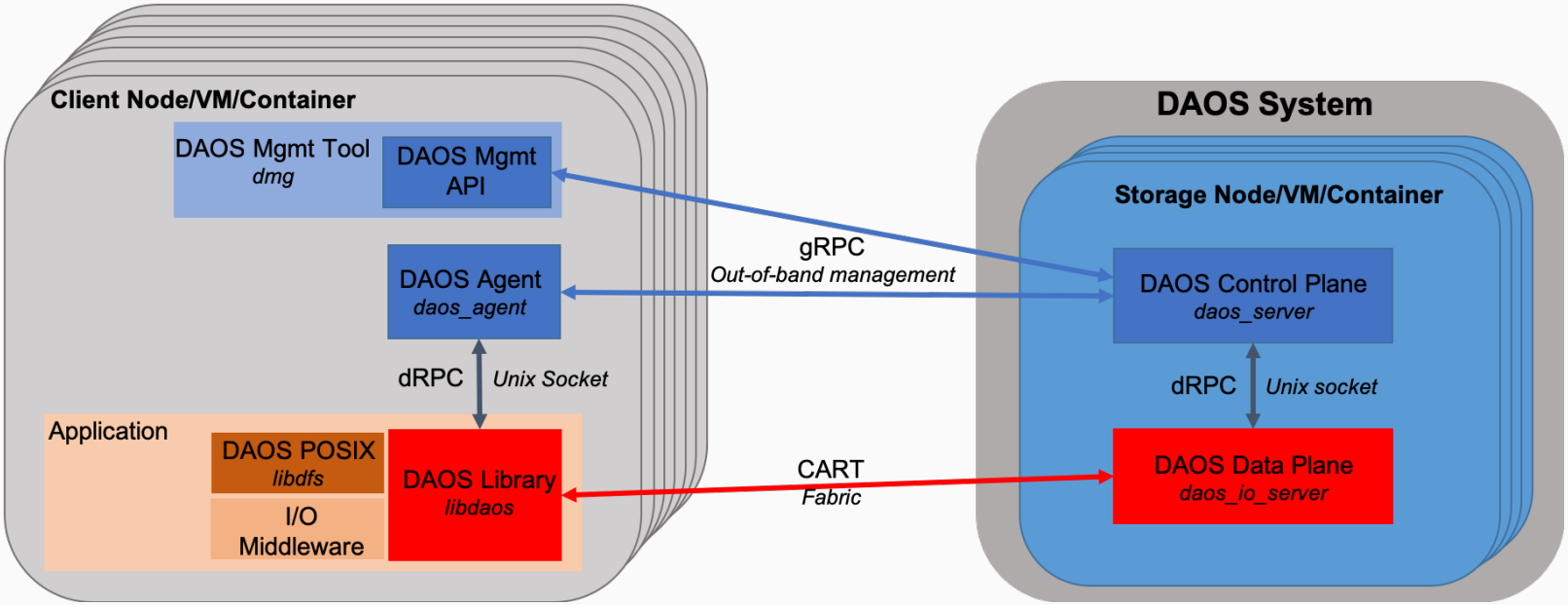


Server: Linux daemon that exports locally-attached NVM storage. Listens on a management interface and 1+ fabric endpoints.

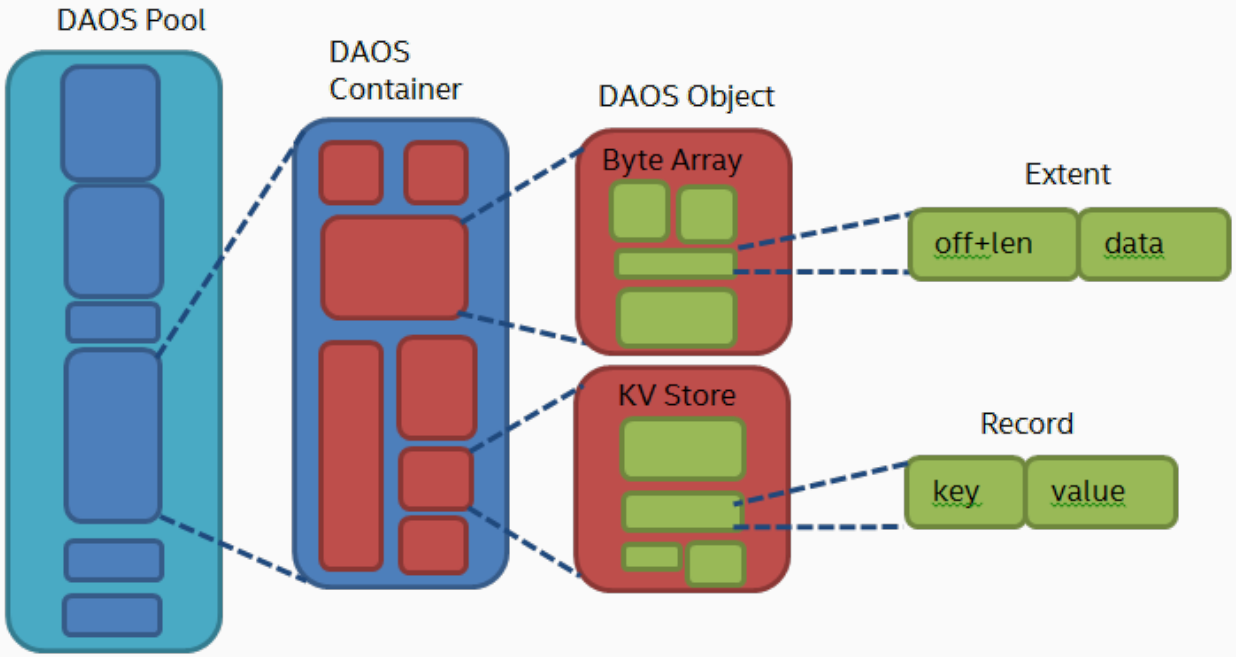
System: a set of DAOS servers connected to the same fabric.

Target: static partition of storage resources (controller, etc.). Avoids contention, as each target has its private storage that can be directly addressed over the fabric independently of the other targets.

Intel DAOS: system architecture



Intel DAOS: pools, containers and objects



- **Object:** a Key–Value store with locality. Can be accessed through 3 different APIs: *multi-level key-array (native)*, *Key–Value*, and *Array*.
  - The key is split into **dkey** (distribution key) and **akey** (attribute key). $dkey_i \mapsto target_k$.
- **Object class:** determines redundancy (replication/erasure code).

Intel DAOS: compatibility layer

Existing software can use DAOS[1,2] through:

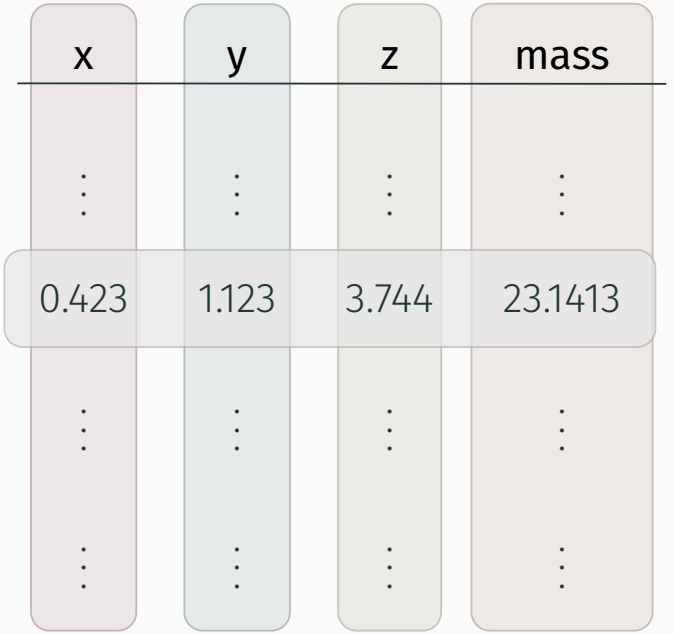
- **POSIX filesystem (libdfs).** Can be used either through `libioil` (I/O call iterception) or `dfuse` (FUSE filesystem).
- **MPI–IO.** Provides DAOS support through a ROMIO driver (MPICH and Intel MPI).
- **HDF5, Apache Spark, ...**

[1] `https://daos-stack.github.io/`
[2] `https://github.com/daos-stack/daos/`

RNTuple 101

TTree and RNTuple

- Row-wise storage is inefficient for HEP data analysis → TTree columnar storage.
- **1+ EB** of HEP data stored in TTree ROOT files.
- TTree there for 25 years. RNTuple is the R&D project to replace TTree for the next 30 years.
- Object storages are first-class.



RNTuple architecture

Event iteration

Looping over events for reading/writing

`RNTupleView, RNTupleReader/Writer`

Logical layer / C++ objects

Mapping of C++ types onto columns, e.g.
`std::vector<float>` $\mapsto$ index column and a value column

`RField, RNTupleModel, REntry`

Primitives layer / simple types

"Columns" containing elements of fundamental types (`float`, `int`, ...) grouped into (compressed) pages and clusters

`RColumn, RPage, …`

Storage layer / byte ranges

POSIX files, object stores, ...

`RPageStorage, RCluster, …`

Approximate equivalent of TTree and RNTuple classes:

TTree	$\approx$	RNTupleReader RNTupleWriter
TTreeReader	$\approx$	RNTupleView
TBranch	$\approx$	RField
TBasket	$\approx$	RPage
TTreeCache	$\approx$	RClusterPool

File backend: on-disk format


```
struct Event {
  int fId;
  vector<Particle> fPtcls;
};
struct Particle {
  float fE;
  vector<int> fIds;
};
```

To put it simple...

Anchor: specifies the offset and size of the header and footer sections.

Header: schema information.[2]

Footer: location of pages and clusters.[2]

Pages: little-endian fundamental types (possibly packed, e.g. bit-fields) —typically in the order of tens of KiB.[2]

[2] This element may be compressed or not.

RNTuple DAOS backend

libdaos: writing an object

```
daos_handle_t pool_handle, container_handle, object_handle;
daos_pool_info_t pool_info;
daos_cont_info_t container_info;

// Connect to pool
struct SvcRAII {
   d_rank_list_t *rankList;
   SvcRAII(std::string_view ranks) { rankList = daos_rank_list_parse(ranks.data(), ":"); }
   ~SvcRAII() { d_rank_list_free(rankList); }
} Svc("1");

uuid_t pool_uuid;
uuid_parse("b4f6d9fc-e081-41d4-91ae-41adf800b537", pool_uuid);
if (int err = daos_pool_connect(pool_uuid, nullptr, Svc.rankList, DAOS_PC_RW, &pool_handle, &pool_info
      , nullptr))
   throw std::runtime_error("daos_pool_connect: error: " + std::string(d_errstr(err)));

// Open container
uuid_t container_uuid;
uuid_parse("b4f6d9fc-e081-41d4-91ae-41adf800b537", container_uuid);
if (int err = daos_cont_open(pool_handle, container_uuid, DAOS_COO_RW,
      &container_handle, &container_info, nullptr))
   throw std::runtime_error("daos_cont_open: error: " + std::string(d_errstr(err)));

// Open object
daos_obj_id_t oid{0xcafe4a11deadbeef, 0};
daos_obj_generate_id(&oid, DAOS_OF_DKEY_UINT64 | DAOS_OF_AKEY_UINT64, OC_RP_XSF, 0);
if (int err = daos_obj_open(container_handle, oid, DAOS_OO_RW, &object_handle, nullptr))
   throw std::runtime_error("daos_obj_open: error: " + std::string(d_errstr(err)));
```

libdaos: writing an object (cont.)

```
// Write object
std::string s("foo bar baz");

uint64_t dkey = 0;
uint64_t akey = 0;
daos_key_t distribution_key{};
daos_iod_t iods[1];
d_sg_list_t sgls[1];
d_iov_t iovs[1];

d_iov_set(&distribution_key, &dkey, sizeof(dkey));

d_iov_set(&iods[0].iod_name, &akey, sizeof(akey));
iods[0].iod_nr = 1;
iods[0].iod_size = s.size();
iods[0].iod_recxs = nullptr;
iods[0].iod_type = DAOS_IOD_SINGLE;

d_iov_set(&iovs[0], const_cast<void *>(s.data()), s.size());
sgls[0].sg_nr_out = 0;
sgls[0].sg_nr = 1;
sgls[0].sg_iovs = &iovs;

daos_obj_update(object_handle, DAOS_TX_NONE, DAOS_COND_DKEY_INSERT, distribution_key, 1,
                          iods, sgls, nullptr);

// Close object, container, and pool
daos_obj_close(object_handle, nullptr);
daos_cont_close(container_handle, nullptr);
daos_pool_disconnect(pool_handle, nullptr);
```

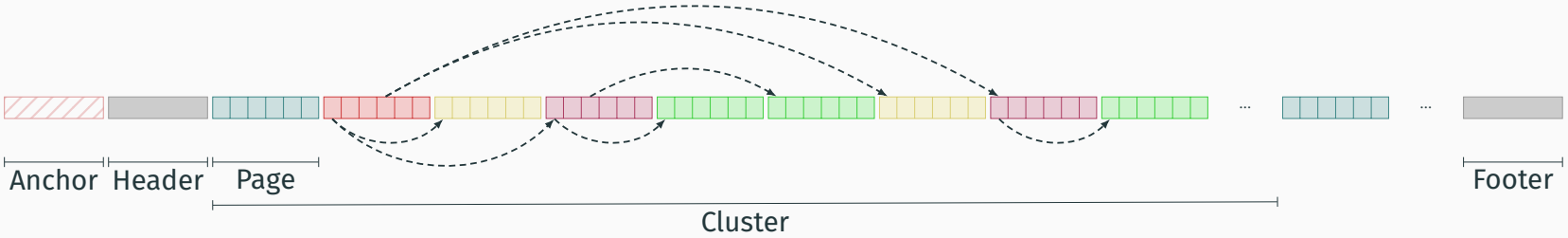
libdaos C++ interface classes

To simplify resource management, we wrote C++ wrappers for part of libdaos functionality.

```
auto pool = std::make_shared<RDaosPool>(
    "e6f8e503-e409-4b08-8eeb-7e4d77cce6bb", "1");
RDaosContainer cont(pool, "b4f6d9fc-e081-41d4-91ae-41adf800b537");

std::string s("foo bar baz");
cont.WriteObject(daos_obj_id_t{0xcafe4a11deadbeef, 0}, s.data(), s.size()
    , /*dkey =*/ 0, /*akey =*/ 0);
```

DAOS backend: mapping things to objects




```
struct Event {
  int fId;
  vector<Particle> fPtcls;
};
struct Particle {
  float fE;
  vector<int> fIds;
};
```

- Each RNTuple page is stored in a separate object. The UUID is sequential starting from `00000000-0000-0000-0000-000000000000`.
- **Header**, **Footer**, and **Anchor** are stored in three different objects with reserved UUIDs.

Usage: RNTuple/file vs. RNTuple/DAOS

From the user's perspective...

```
auto model = RNTupleModel::Create();
auto ntuple = RNTupleReader::Open(std::move(model),
            "DecayTree",
            "./B2HHH~zstd.ntuple");

auto viewH1IsMuon = ntuple->GetView<int>("H1_isMuon");
auto viewH2IsMuon = ntuple->GetView<int>("H2_isMuon");
auto viewH3IsMuon = ntuple->GetView<int>("H3_isMuon");
```

Usage: RNTuple/file vs. RNTuple/DAOS

From the user's perspective...

```
auto model = RNTupleModel::Create();
auto ntuple = RNTupleReader::Open(std::move(model),
            "b4f6d9fc-e081-41d4-91ae-41adf800b537",
            "daos://e6f8e503-e409-4b08-8eeb-7e4d77cce6bb/1");

auto viewH1IsMuon = ntuple->GetView<int>("H1_isMuon");
auto viewH2IsMuon = ntuple->GetView<int>("H2_isMuon");
auto viewH3IsMuon = ntuple->GetView<int>("H3_isMuon");
```

Preliminary evaluation

Test environment

Our evaluation ran on CERN OpenLab DAOS test machines:

- 3 DAOS servers, 1 DAOS head node.
- interconnected by an Omni-Path Edge Switch 100 Series | 24 ports.

System specifications	
CPU	Intel(R) Xeon(R) Platinum 8260 CPU @ 2.40GHz
CPU per node	24 cores/socket, 2 sockets, 2 threads/core (HT enabled)
Core frequency	Base: 1.0 GHz Range: 1.0GHz - 3.9GHz
Numa nodes	node0: 0-23,48-71 node1: 24-47,72-95
System Memory	12x 32GB DDR4 rank DIMMs
Optane DCPMM	12x 128GB DDR4 rank DIMMs
Optane FW version	01.02.00.5395
BIOS	version: SE5C620.86B.02.01.0011.032620200659 date: 03/26/2020
Storage	4x 1 TB NVMe INTEL SSDPE2KX010T8
HFI	1x Intel Corporation Omni-Path HFI Silicon 100 Series.
HFI Firmware	Termal Management Module: `10.9.0.0.208`; Driver: `1.9.2.0.0`

Figure 1: Server nodes HW (`olcsl-*`)

System specifications	
CPU	Intel(R) Xeon(R) Platinum 8160 CPU @ 2.10GHz
CPU per node	24 cores/socket, 2 sockets, 2 threads/core (HT enabled)
Core frequency	Base: 1.0 GHz Range: 1.0GHz - 3.9GHz
Numa nodes	node0: 0-23,48-71 node1: 24-47,72-95
System Memory	12x 16GB DDR4 rank DIMMs
BIOS	version: SE5C620.86B.02.01.0011.032620200659 date: 03/26/2020
HFI	1x Intel Corporation Omni-Path HFI Silicon 100 Series.
HFI Firmware	Termal Management Module: `10.9.0.0.208`; Driver: `1.9.2.0.0`

Figure 2: Client node HW (`olsky-03`)

`dfuse` simple benchmark (ofi+sockets)

⚠ These results are preliminary and might not be reliable.

	Block size					
	4K	8K	16K	512K	1M	4M
Seq. Write	**7.62**	14.42	27.44	189.21	205.10	**225.62**
Seq. Read	2.62	5.04	9.21	116.86	147.7 9	188.90
Random Write	7.30	**14.67**	**27.63**	**199.68**	**209.17**	211.40
Random Read	2.16	4.20	7.92	120.91	162.7 0	211.12

Table 1: `dfuse` read/write benchmark (**in MiB/s**)

- Far from the 34.2 Gbits/sec (4.275 GiB/s) achieved by `iperf`.
- Path lookup not bad; around **700+** `open()`/`creat()` calls/s.

RNTuple (ofi+sockets)

⚠ These results are preliminary and might not be reliable.

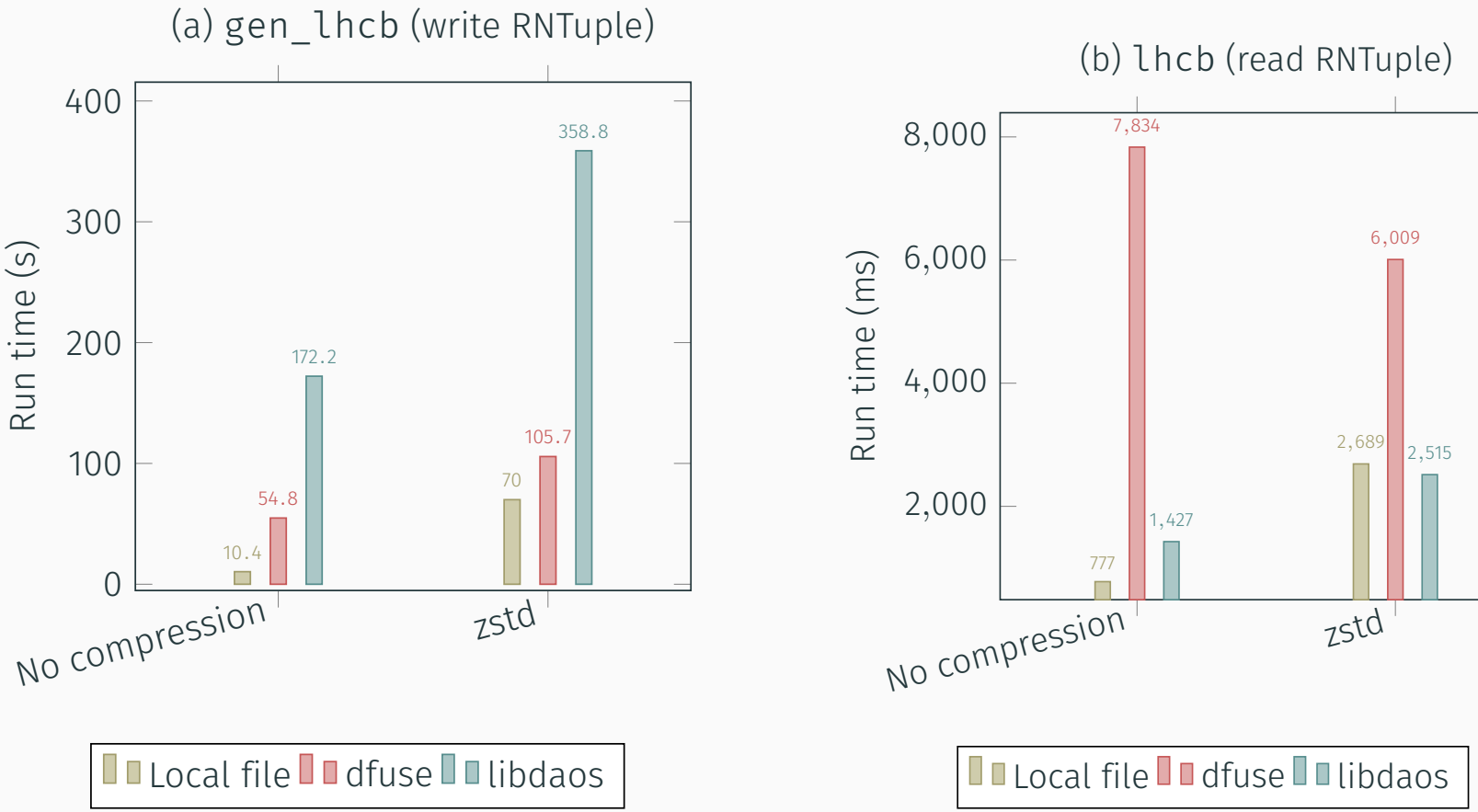


Figure 3: RNTuple benchmark on LHCb data (ofi+sockets).[3][4]

[3] Input data size: 1.5 GiB (uncompressed) / 1007 MiB (zstd).
[4] `https://github.com/jblomer/iotools`

RNTuple (ofi+psm2)

⚠ These results are preliminary and might not be reliable.

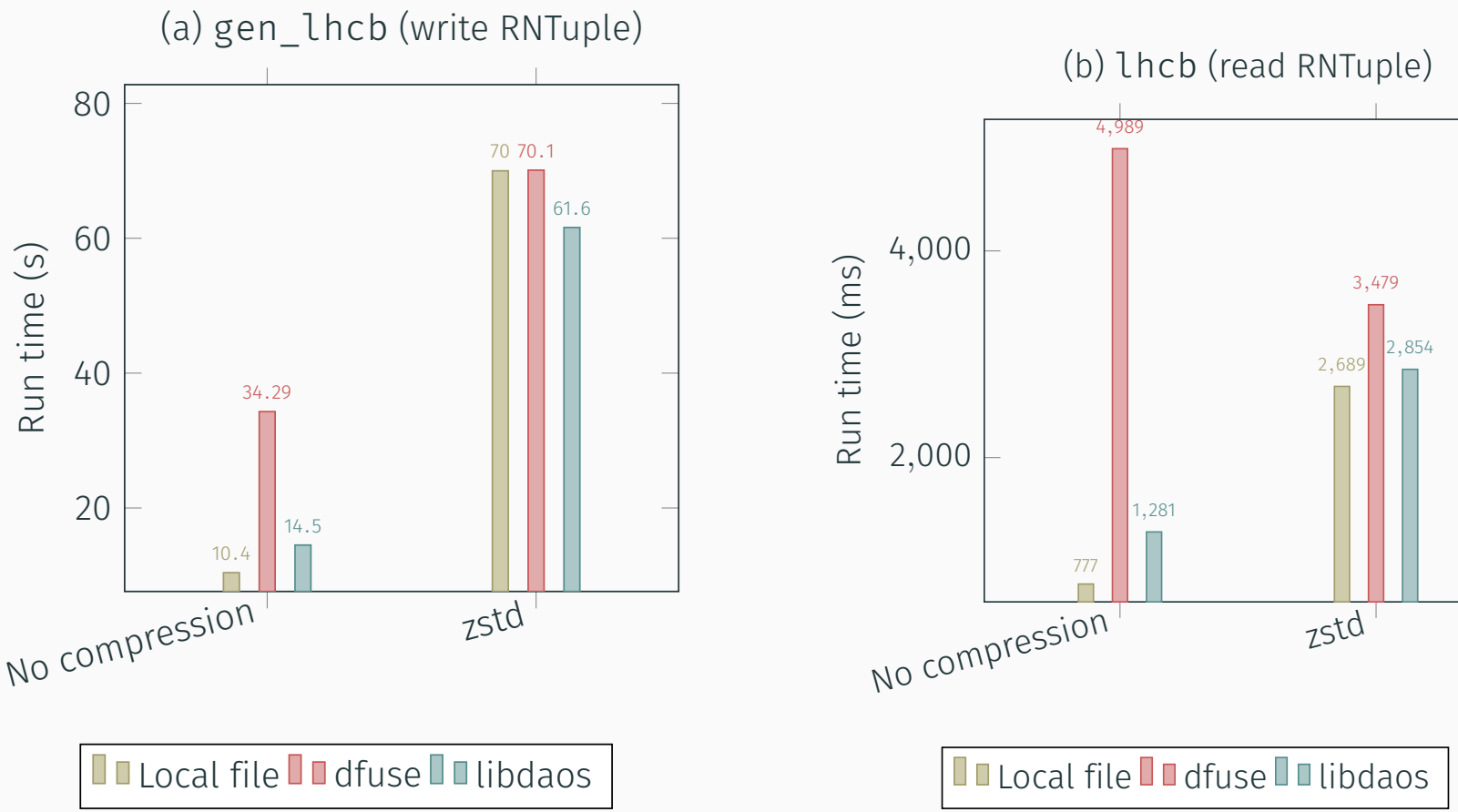


Figure 4: RNTuple benchmark on LHCb data (ofi+PSM2).[5][6]

[5] Input data size: 1.5 GiB (uncompressed) / 1007 MiB (zstd).
[6] `https://github.com/jblomer/iotools`

Closing words

Closing words

- 1+ EB of HEP data in ROOT files (TTree). RNTuples replaces TTree columnar storage for the next 30 years.
- RNTuple architecture decouples storage from serialization/representation. Object stores are first-class.
- First prototype implementation of an Intel DAOS backend. Currently "1 Page == 1 Object" + constant dkey. Needs more benchmarking!
- ROOT PR #6825 (draft): `https://github.com/root-project/root/pull/6825`

Next Questions:

1. How to maximize throughput (bulk reading/writing of pages)?
2. How to distribute pages appropriately, e.g. put together pages corresponding to the same data member?
3. Data movement: how to quickly move large amounts of data from HEP storage to a DAOS data center?

Exploring Object Stores: RNTuple DAOS Backend

Javier López-Gómez – **CERN**
<javier.lopez.gomez@cern.ch>

SFT group meeting, 7th December 2020

ROOT project,
EP-SFT, CERN

http://root.cern/