Unification of Temporary Storage in the NodeKernel Architecture

Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, Ana Klimovic, Adrian Schuepbach, Bernard Metzler
Temporary/Intermediate Data

Map-reduce job
Temporary/Intermediate Data

Map-reduce job
Temporary/Intermediate Data

Input data → Broadcast → Map → Shuffle → Reduce → Intermediate data → ML training → Output data

HDFS, S3 → Map-reduce job → HDFS, S3 → ML training → HDFS, S3
### Temporary/Intermediate Data

**Input data**
- HDFS, S3

**ML pre-processing** (Spark job)
- Original data
- Zero-centered data
- Normalized data

**Map-reduce job**

**Output data**
- HDFS, S3

**normalized images**

**ML training** (Tensorflow job)

**ML training**
Temporary/Intermediate Data

- 97 out of 100 TPC-DS queries includes at least “order by”, “group by” or “join”
- 50% of all jobs at FB have at least 1 shuffle [Riffle/Eurosys’18]
- ML Workflow [Apache Airflow, MLFlow, etc.]

Temporary data is an important class of data for data processing workloads
Shortcomings of Temporary Data Storage

- **Inefficient:**
  - Difficult to leverage modern networking and storage hardware (e.g., 100 Gb/s Ethernet, NVMe Flash, etc.)
Shortcomings of Temporary Data Storage

• Inefficient:
  - Difficult to leverage modern networking and storage hardware (e.g., 100 Gb/s Ethernet, NVMe Flash, etc.)
Shortcomings of Temporary Data Storage

• Inefficient:
  – Difficult to leverage modern networking and storage hardware (e.g., 100 Gb/s Ethernet, NVMe Flash, etc.)

• Inflexible:
  – Temporary data management hard-wired with data processing framework
  – Difficult to change deployment (e.g., disaggregation, tiered storage, etc.)
Instead of this...

Can’t implement every operation for all the different hardware and deployment options.
...better do this

Implement hardware support once and support different operations and frameworks.
How should the temporary data store look?

Can we use an existing storage platform, e.g., KV store, FS, etc.?
Temporary data distribution

Wide range of data sets (per task)
Temporary data distribution

Wide range of data sets (per task)

KV-Store regime: RAMCloud, ccKVS, etc.
Temporary data distribution

Wide range of data sets (per task)
Temporary data distribution

Wide range of data sets (per task)

KV-Store regime: RAMCloud, ccKVS, etc.

Filesystem regime: Octopus, Gassyfs, etc.
Temporary Data Storage Requirements

• Perform well for wide range of data sizes
  – A few KB to many GBs per storage object
Temporary Data Storage Requirements

- Perform well for wide range of data sizes
  - A few KB to many GBs per storage object
- Support large data volumes
  - Can’t keep all data in memory all the time
Temporary Data Storage Requirements

• Perform well for wide range of data sizes
  – A few KB to many GBs per storage object

• Support large data volumes
  – Can’t keep all data in memory all the time

• Provide convenient abstractions for storing temporary
  – Key-value, File, what else?
Temporary Data Storage Requirements

- Perform well for wide range of data sizes
  - A few KB to many GBs per storage object
- Support large data volumes
  - Can’t keep all data in memory all the time
- Provide convenient abstractions for storing temporary
  - Key-value, File, what else?
- Scalability
Temporary Data Storage Requirements

• Perform well for wide range of data sizes
  – A few KB to many GBs per storage object

• Support large data volumes
  – Can’t keep all data in memory all the time

• Provide convenient abstractions for storing temporary
  – Key-value, File, what else?

• Scalability

• Fault-tolerance, Durability
  – Temporary data is short-lived, can we use coarse grained recovery?
NodeKernel

- Distributed storage architecture for temporary data
- Fusion of filesystem and key-value semantics
- Designed for high-performance hardware
NodeKernel: Data Model

CreateNode()  
LookupNode()  
RemoveNode()  
RenameNode()  

Diagram:

NodeKernel: Data Model

CreateNode()  
LookupNode()  
RemoveNode()  
RenameNode()
NodeKernel: Data Model

CreateNode()
LookupNode()
RemoveNode()
RenameNode()

Node {
    AppendData()
    UpdateData()
    ReadData()
}

Node { param1, param2, config, shuffle, part1, part2, map1, map2, cache, key1, key2 }
NodeKernel: Node Types

Directory: Node

- Enumerate()

File: Node

- Read()
- Append()
NodeKernel: Node Types

Table : Node {
    Put()
    Get()
}

KeyValue : Node {
    Append()
    Read();
}
NodeKernel: Node Types

Bag : Node {
    readSubtree()
}
File : Node {
    Read()
    Append()
}
NodeKernel: System Architecture

Datacenter Network

Application View

foo bar

Storage Class 1 Storage Class 2 Storage Class N

Clients

e.g., all storage servers with NVMe Flash

28
NodeKernel: System Architecture

tree structure, tiering policy, block mapping

Metadata Plane

Storage Class 1

Storage Class 2

Storage Class N

e.g., all storage servers with NVMe Flash

Clients

foo

bar

Datacenter Network

Network

Application View

/
NodeKernel: System Architecture

- **Application View**: foo / bar
- **Datacenter Network**: Metadata Plane, Storage Class 1, Storage Class 2, Storage Class N
- **hierarchical Tiering**: fill higher performing tiers first
- **Clients**: e.g., all storage servers with NVMe Flash

- tree structure, tiering policy, block mapping

- e.g., all storage servers with NVMe Flash
NodeKernel: System Architecture

Application View

hash partitioned storage namespace

tree structure, tiering policy, block mapping

hierarchical Tiering: fill higher performing tiers first

e.g., all storage servers with NVMe Flash

foo

bar

/Metadata Plane

Storage Class 1

Storage Class 2

Storage Class N
NodeKernel: System Architecture

- **Metadata Plane**: Tree structure, tiering policy, block mapping
- **Storage Classes**: Class 1, Class 2, Class N
- **Clients**: Hierarchical tiering: fill higher performing tiers first
- **Applications**: View, hash partitioned storage namespace
Example: Key-Value PUT
Example: Key Value PUT

Separating metadata from data adds **flexibility**
Example: KeyValue PUT

Separating metadata from data adds **flexibility** but requires **low-latency metadata operations**
Apache Crail

• **Implementation** of the NodeKernel architecture
• Low-latency RDMA-based RPC between client and metadata servers
• Two storage classes:
  - Flash accessed via NVM-over-Fabrics
  - DRAM accessed via RDMA
• Open source: [crail.apache.org](http://crail.apache.org)
Evaluation

• 16 node cluster, machine hardware:
  - 100 Gb/s RDMA RoCE
  - 256 GB DRAM
  - Intel Optane NVMe SSD

• Evaluation questions:
  - **Any size**: how well is Crail performing for different object sizes?
  - **Modern hardware**: are we able to accelerate workloads?
  - **Flexibility**: what benefits we get by decoupling data processing and temporary data storage?
  - **Abstractions**: Are KeyValue, File and Bag abstractions helpful?
Small and Large Data Sets

Crail serves small and large data sets close to the hardware limit (latency RDMA: 3us, latency Optane 15us, bandwidth RDMA: 100 Gb/s)
Spark Shuffle

compute cluster

map tasks

local files

network transfers

reduce tasks
Spark Shuffle using Crail::Bag

1. append
2. sequential read
reduce tasks
compute cluster
map tasks
Crail File
Crail Bag
map tasks
Crail File
Crail Bag
map tasks
Crail File
Crail Bag
map tasks
Crail File
Crail Bag
Spark GroupBy (80M keys, 4K)

Spark shuffling via Crail on a single core is 2x faster than vanilla Spark on 8 cores per executor (8 executors)
Flexible Deployment

Data processing job

HDFS, S3

Crail

HDFS, S3

configuration

Disaggregation, Flash/DRAM ratio, RDMA, TCP, NVM-of-Fabrics, etc.
Flexible deployment: Crail permits trading performance for cost
Conclusions

• Sharing temporary efficiently in data processing workloads is challenging
  – Inefficient in deployments with modern hardware
  – Inflexible: difficult to use storage tiering, disaggregation, etc.

• NodeKernel: distributed storage architecture for temporary data storage
  – Fusion of Filesystem and Key-Value semantics in single storage namespace

• Apache Crail: Implementation of NodeKernel for RDMA and NVMf
  – Accelerates temporary data storage on modern hardware
  – Enable flexible deployment: storage tiering, disaggregation, etc.
Open Source

• Crail:
  https://github.com/apache/incubator-crail

• Crail shuffler:
  https://github.com/zrlio/crail-spark-io

• YCSB benchmark:
  https://github.com/brianfrankcooper/YCSB
  (includes Crail)
Backup
YCSB Benchmark: GET Latency

1KB KV pairs: ~12us (DRAM) and 30us (NVMe)
100KB KV pairs: ~30us (DRAM) and 40us (NVMe)
## Persistence & Fault Tolerance

### Data plane
- No replication
- Graceful handling of faulty or crashed storage servers
  (signaled at client during read/write ops)

### Meta data plane
- Persist metadata state using operation logging
- Shutdown and replay log to re-create state

### Pluggable log device
- Current log is on local FS
- Could be a distributed log: maintain a hot standby metadata server
Crail Data Plane

```java
CrailStore crail = CrailStore.newInstance();
Future<

C

Future<

ByteBuffer buffer = crail.allocateBuffer();
Future<

get();
```
Crail Data Plane

```
CrailStore crail = CrailStore.newInstance();
Future<Named> fut = crail.create("/a.dat", CrailType.File);
    //...do work
CrailFile file = fut.get().asFile();
CrailOutputStream stream = file.getOutputStream();
ByteBuffer buffer = crail.allocateBuffer();
Future<CrailResult> ret = stream.write(buf);
    //...do work
ret.get();
```
Crail Data Plane

zero copy data movement

asynchronous API

```java
CrailStore crail = CrailStore.newInstance();
Future<Node> fut = crail.create("/a.dat", CrailType.File);
//...do work
CrailFile file = fut.get().asFile();
CrailOutputStream stream = file.getDirectOutputStream();
ByteBuffer buffer = crail.allocateBuffer();
Future<CrailResult> ret = stream.write(buf); //...do work
ret.get();
```
Crail Data Plane

zero copy data movement
direct data placement
asynchronous API
Crail Data Plane

- Zero copy data movement
- Transfer only data requested
- Direct data placement
- Asynchronous API

```
CrailStore crail = CrailStore.newInstance();
Future<Node> fut = crail.create("/a.dat", CrailType.File);
//...do work
CrailFile file = fut.get().asFile();
CrailOutputStream stream = file.getDirectOutputStream();
ByteBuffer buffer = crail.allocateBuffer();
Future<CrailResult> ret = stream.write(buf);
//...do work
ret.get();
```
Crail Data Plane

zero copy data movement
transfer only data requested
direct data placement

direct data placement

synchronous call
asynchronous API

1. CrailStore crail = CrailStore.newInstance();
2. Future<Node> fut = crail.create("/a.dat", CrailType.File);
3. //...do work
4. CrailFile file = fut.get().asFile();
5. CrailOutputStream stream = file.getDirectOutputStream();
6. ByteBuffer buffer = crail.allocateBuffer();
7. Future<CrailResult> ret = stream.write(buf);
8. //...do work
9. ret.get();