SplitFT: Fault Tolerance for Disaggregated Datacenters via Remote Memory Logging

Xuhao Luo, Ramnatthan Alagappan, Aishwarya Ganesan
University of Illinois Urbana-Champaign

4/24/2024 @ EuroSys'24

Storage-Centric Applications on the Cloud

Storage-Centric Applications on the Cloud

Storage-Centric Applications on the Cloud

Storage-Centric Applications on the Cloud

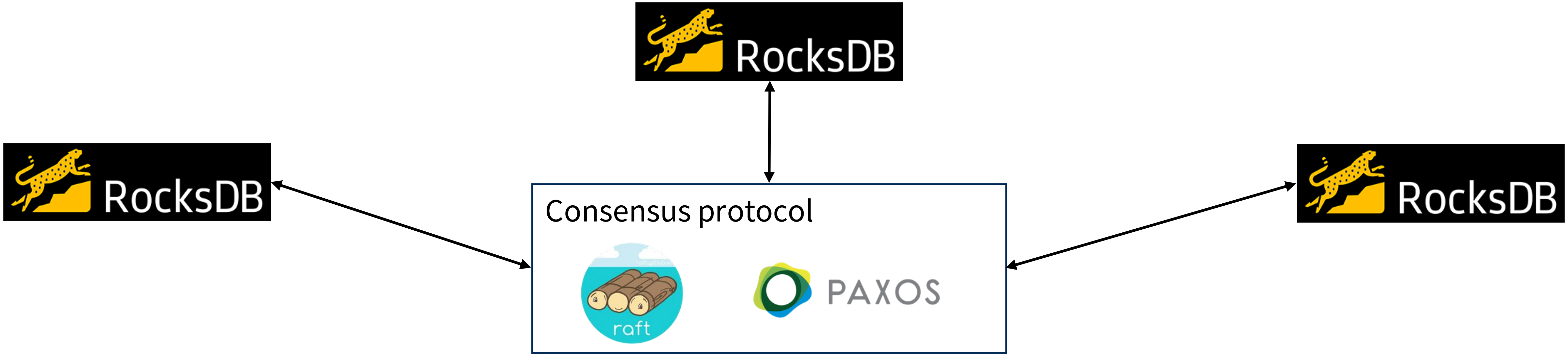
Storage-Centric Applications on the Cloud

Storage-Centric Applications on the Cloud

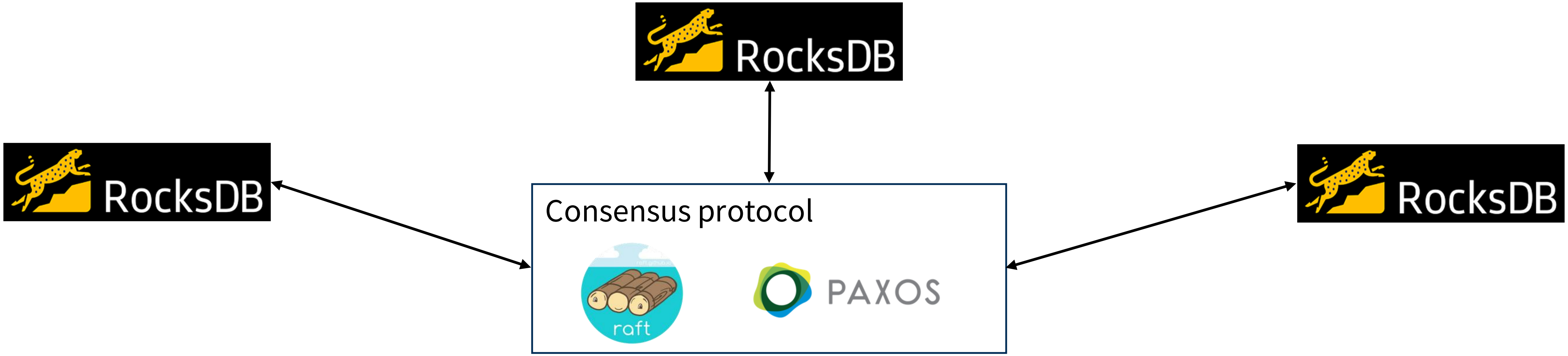
Requirement:

- High availability
- Durability
- Strong consistency

Traditional Way: Application-Level Fault Tolerance

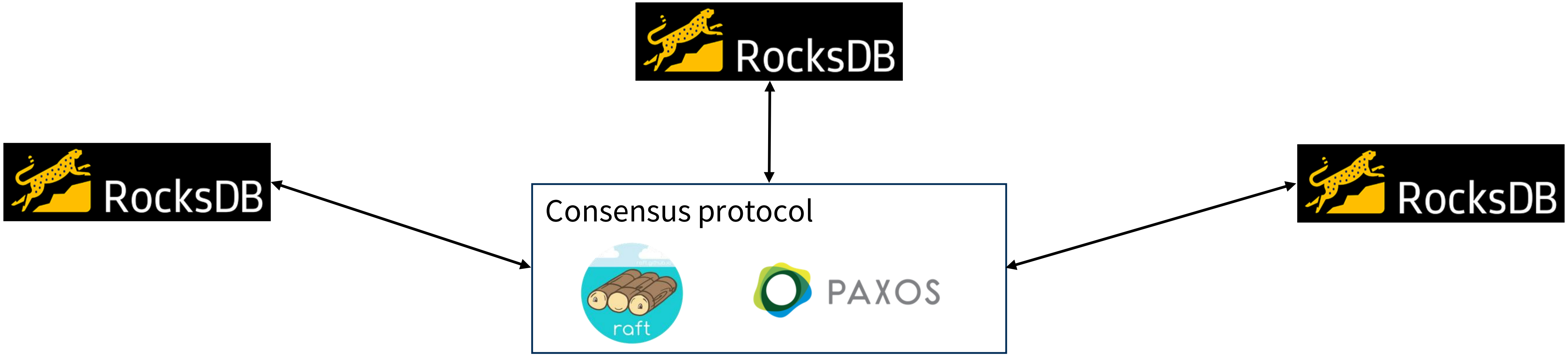


Traditional Way: Application-Level Fault Tolerance



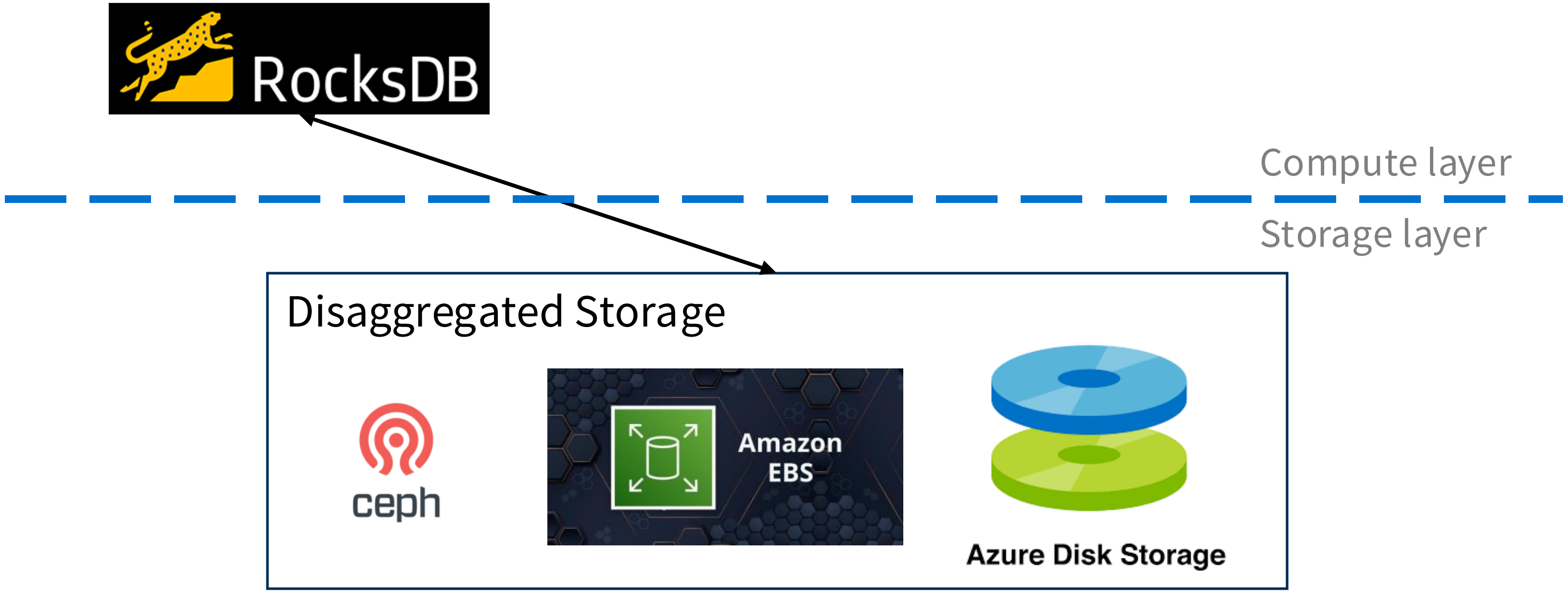
❌ Significant developing burden

Traditional Way: Application-Level Fault Tolerance

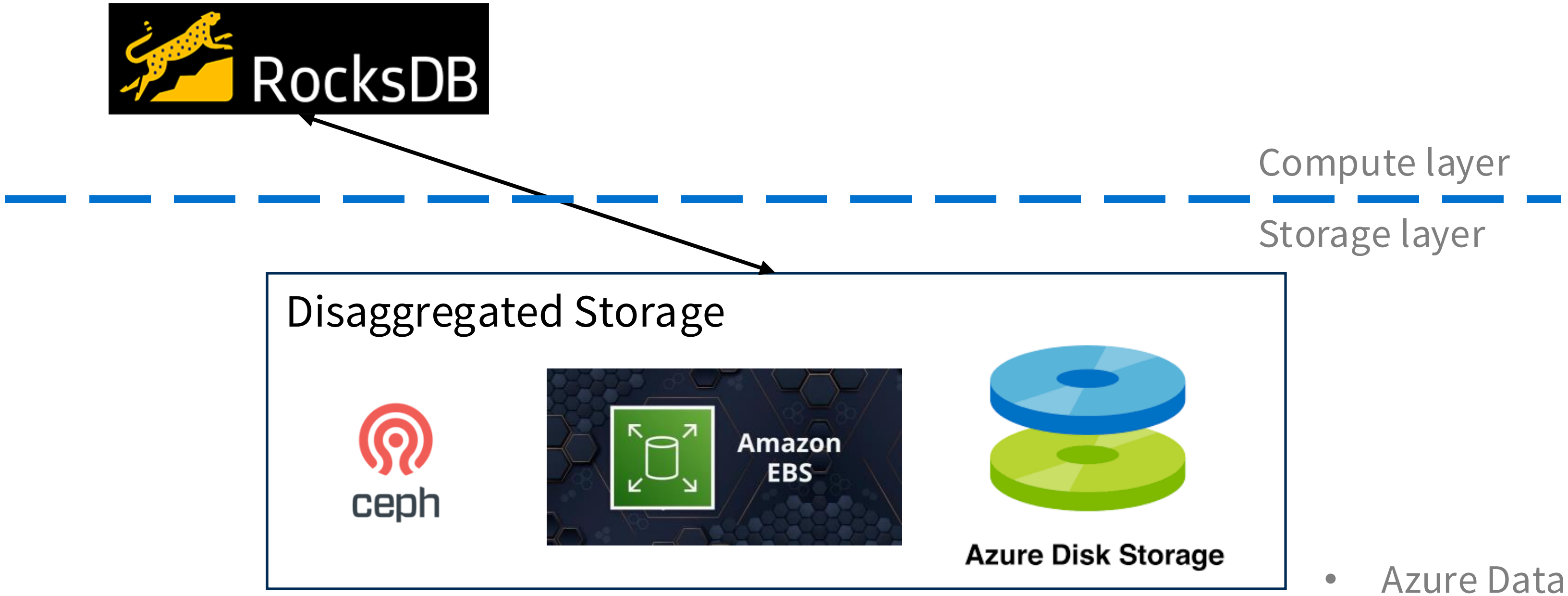


- ✗ Significant developing burden
- ✗ N-times Resource overhead

Alternative Way: Disaggregated Fault Tolerance (DFT)

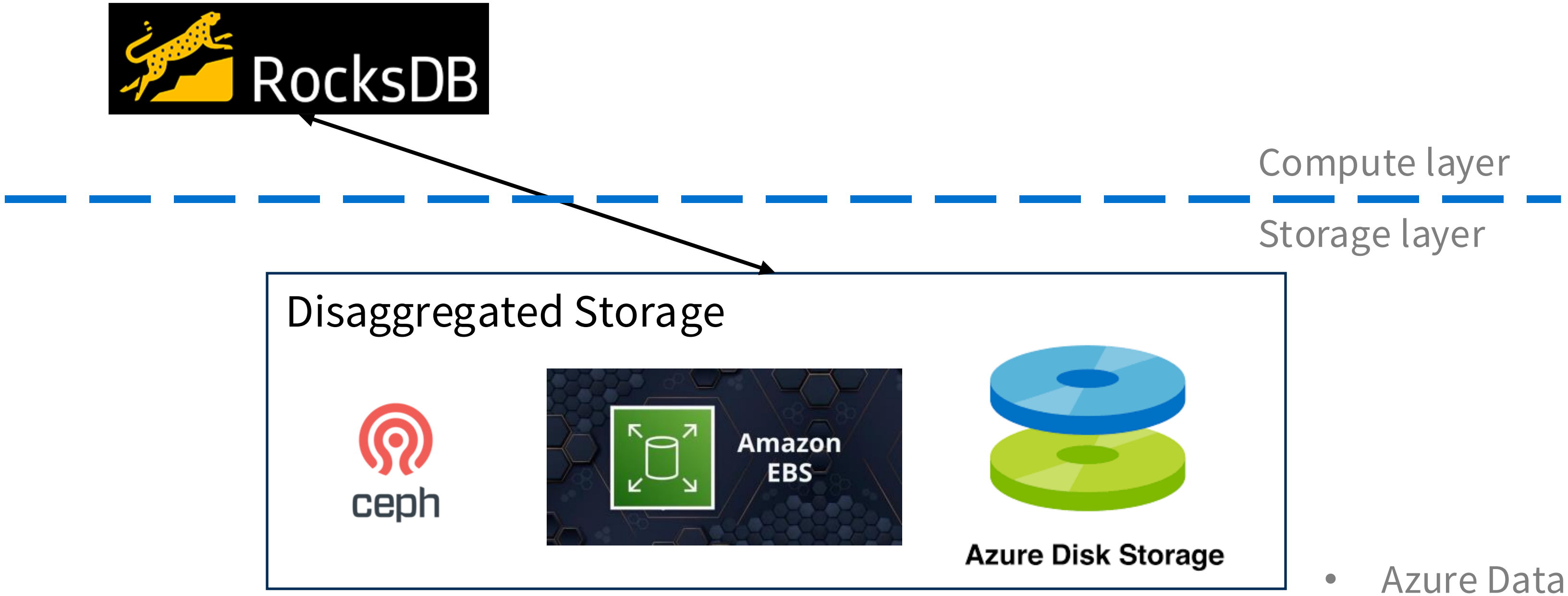


Alternative Way: Disaggregated Fault Tolerance (DFT)



- Azure Database for PostgreSQL
- RocksDB-Cloud
- ChakrDB

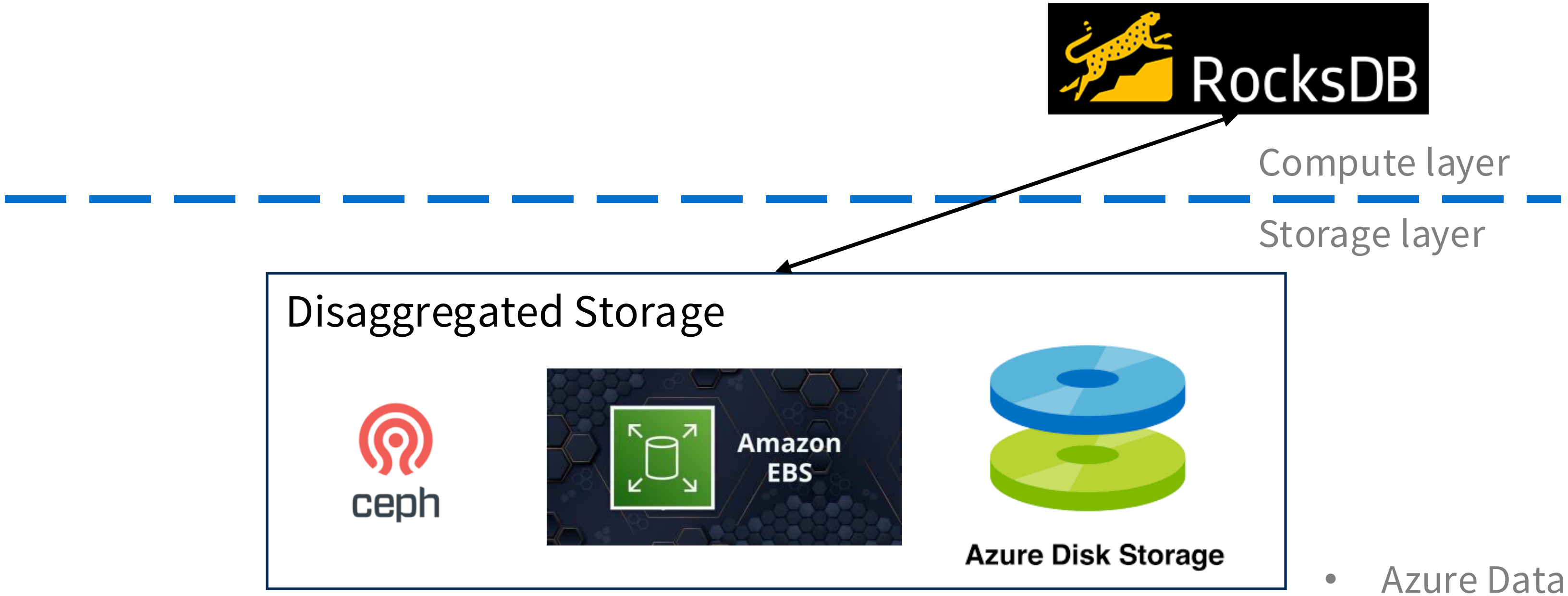
Alternative Way: Disaggregated Fault Tolerance (DFT)



- Azure Database for PostgreSQL
- RocksDB-Cloud
- ChakrDB

✔ Transparent fault tolerance

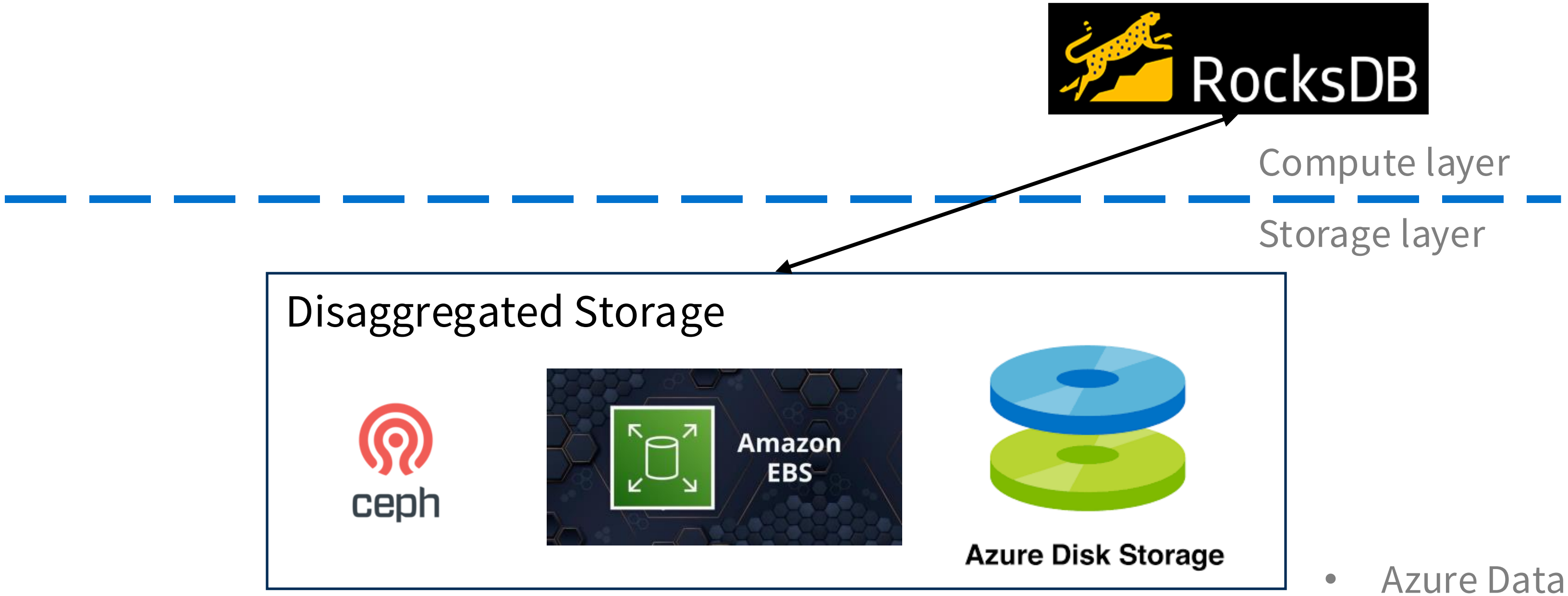
Alternative Way: Disaggregated Fault Tolerance (DFT)



- Azure Database for PostgreSQL
- RocksDB-Cloud
- ChakrDB

✔ Transparent fault tolerance

Alternative Way: Disaggregated Fault Tolerance (DFT)

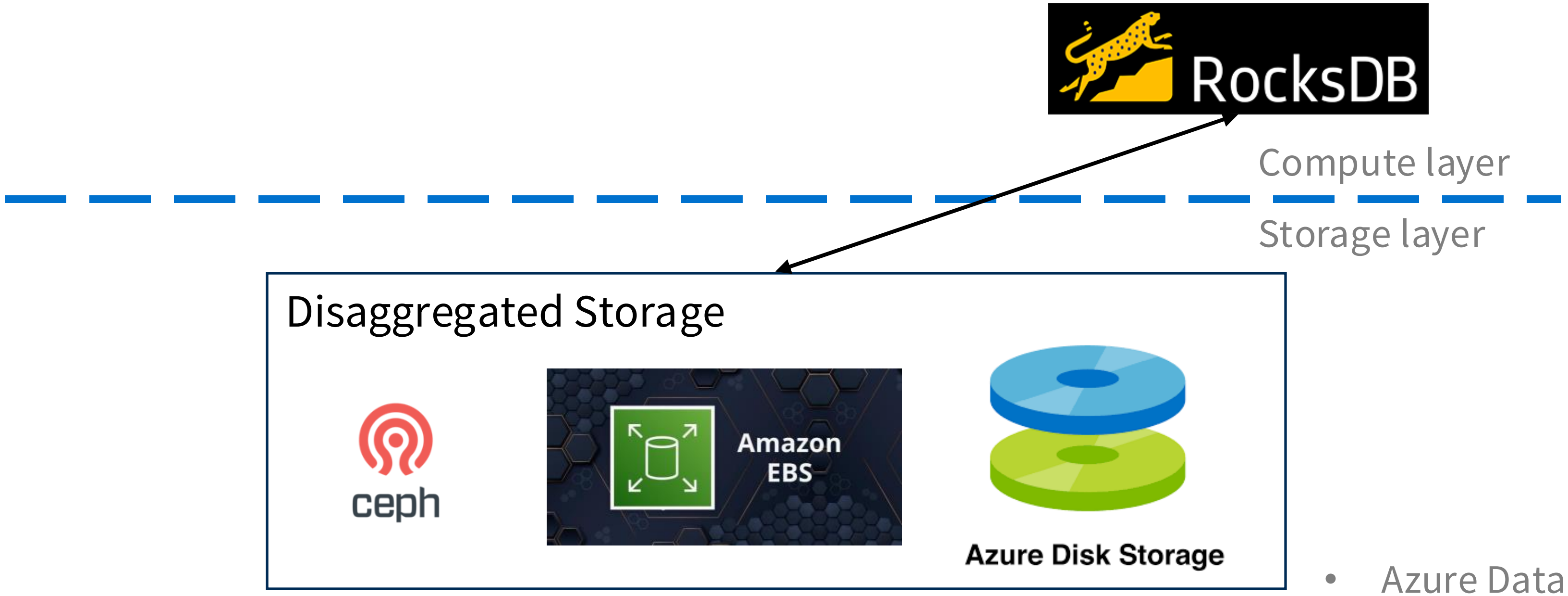


- Azure Database for PostgreSQL
- RocksDB-Cloud
- ChakrDB

✔ Transparent fault tolerance

✔ Low resource consumption

Alternative Way: Disaggregated Fault Tolerance (DFT)



- Azure Database for PostgreSQL
- RocksDB-Cloud
- ChakrDB

✓ Transparent fault tolerance

✓ Low resource consumption

✗ Tradeoff between *performance* and *strong guarantees* (FT, Durability)

Strong Guarantee or Performance in DFT

Update | Update | Update

Update | Update | Update

Disaggregated Storage

ceph

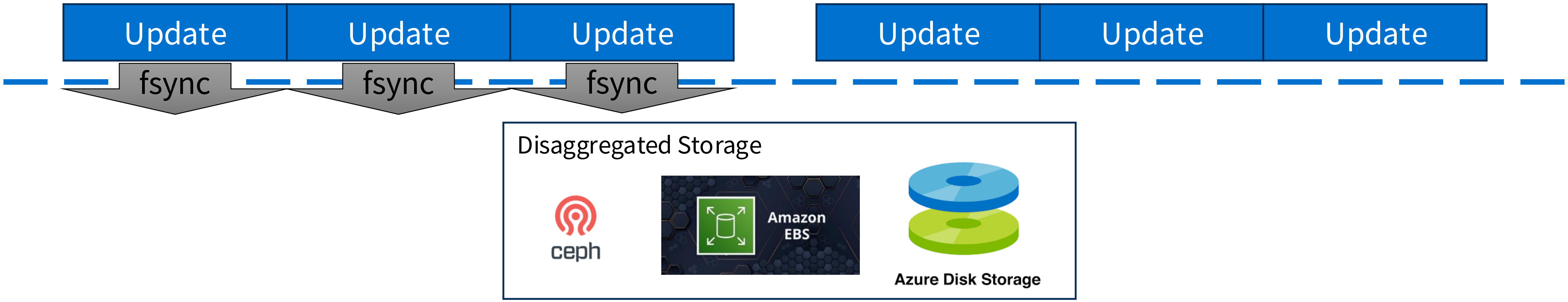
Amazon EBS

Azure Disk Storage

Synchronous flush:

Asynchronous flush:

Strong Guarantee or Performance in DFT

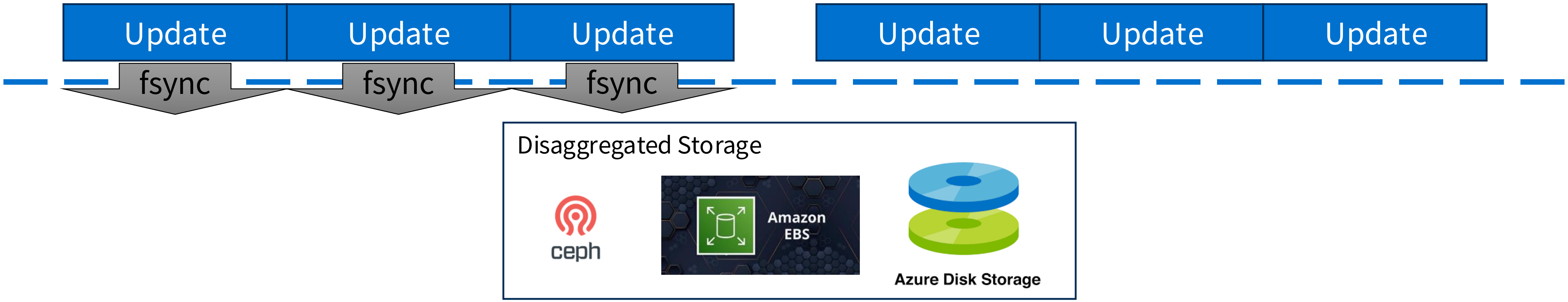


Synchronous flush:

- strong durability

Asynchronous flush:

Strong Guarantee or Performance in DFT

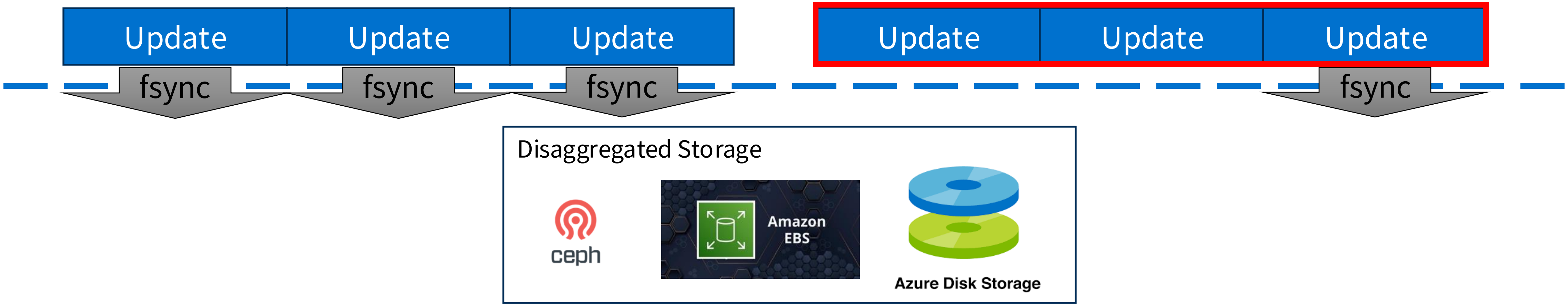


Synchronous flush:

- strong durability
- poor performance

Asynchronous flush:

Strong Guarantee or Performance in DFT



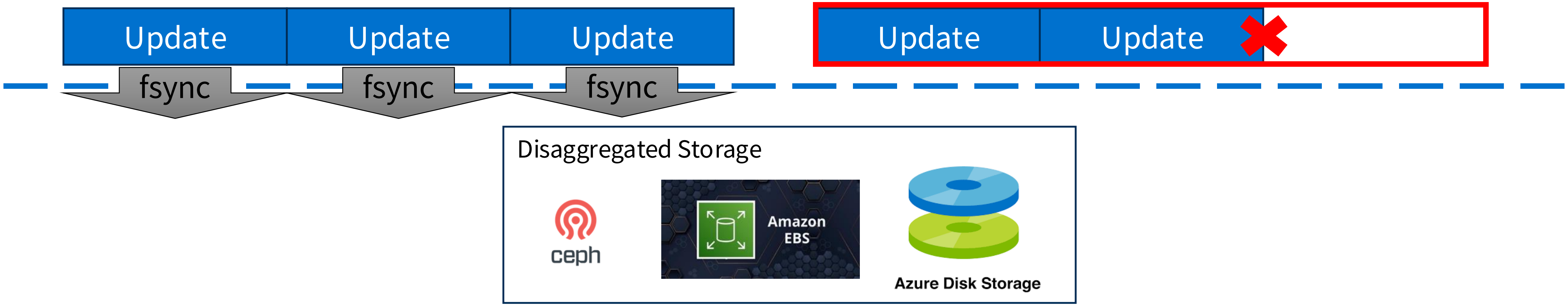
Synchronous flush:

- strong durability
- poor performance

Asynchronous flush:

- good performance

Strong Guarantee or Performance in DFT



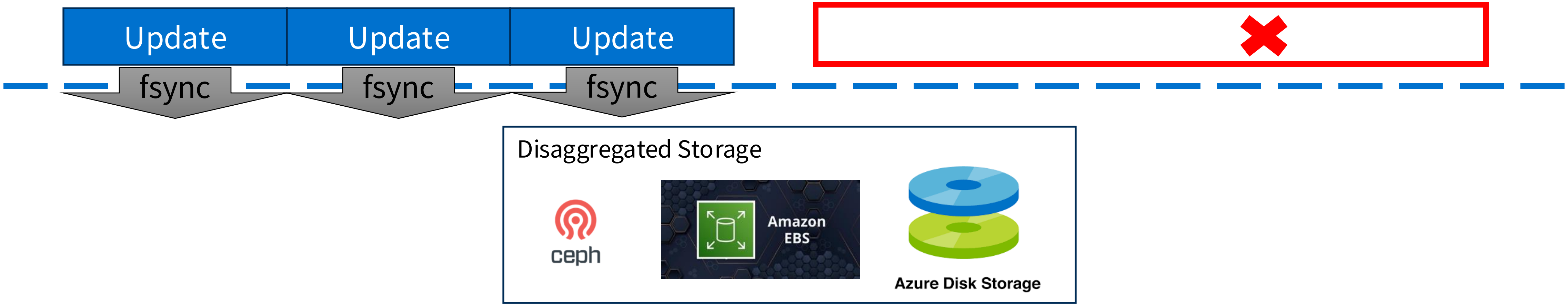
Synchronous flush:

- strong durability
- poor performance

Asynchronous flush:

- good performance

Strong Guarantee or Performance in DFT



Synchronous flush:

- strong durability
- poor performance

Asynchronous flush:

- good performance
- risk losing updates

Can We Achieve Both Strong Guarantees and Performance in DFT?

Our Observation

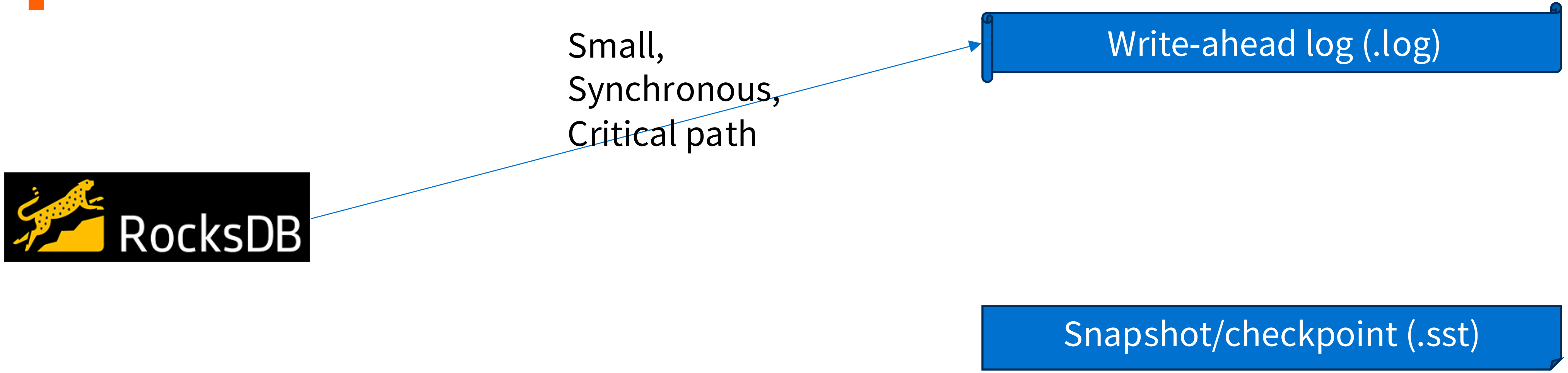
Write-ahead log (.log)

RocksDB

Snapshot/checkpoint (.sst)

Dual-nature of writes:

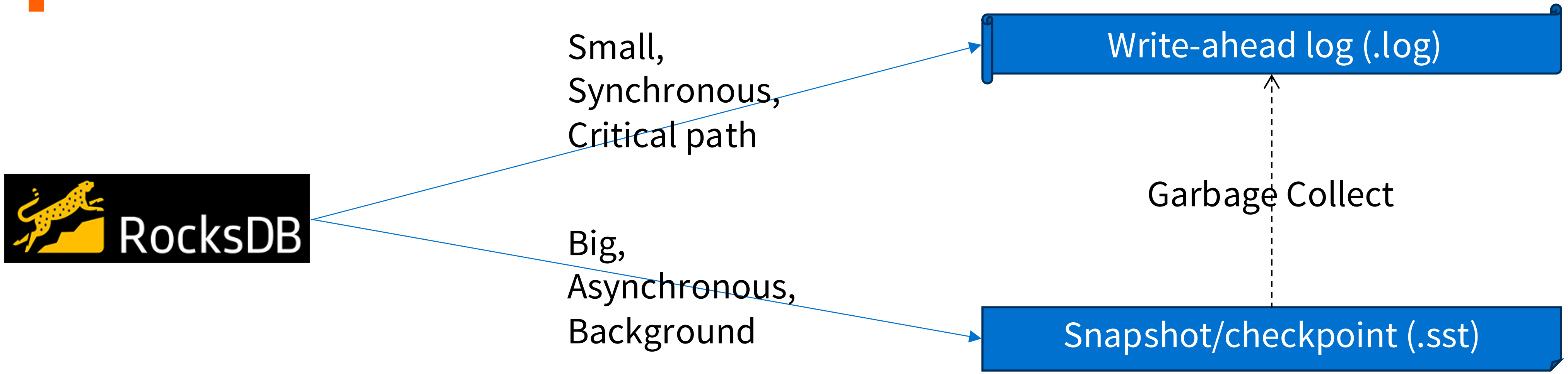
Our Observation



Dual-nature of writes:

- Small synchronous writes to log: durability, crash recovery

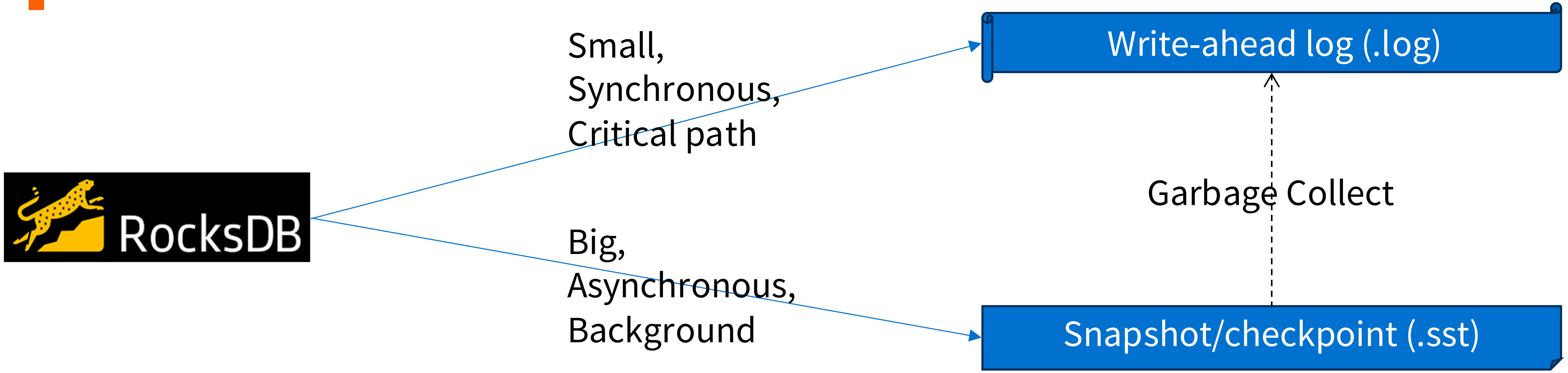
Our Observation



Dual-nature of writes:

- Small synchronous writes to log: durability, crash recovery
- Bulk asynchronous write to checkpoint: save snapshot, garbage-collect log

Our Observation



Dual-nature of writes:

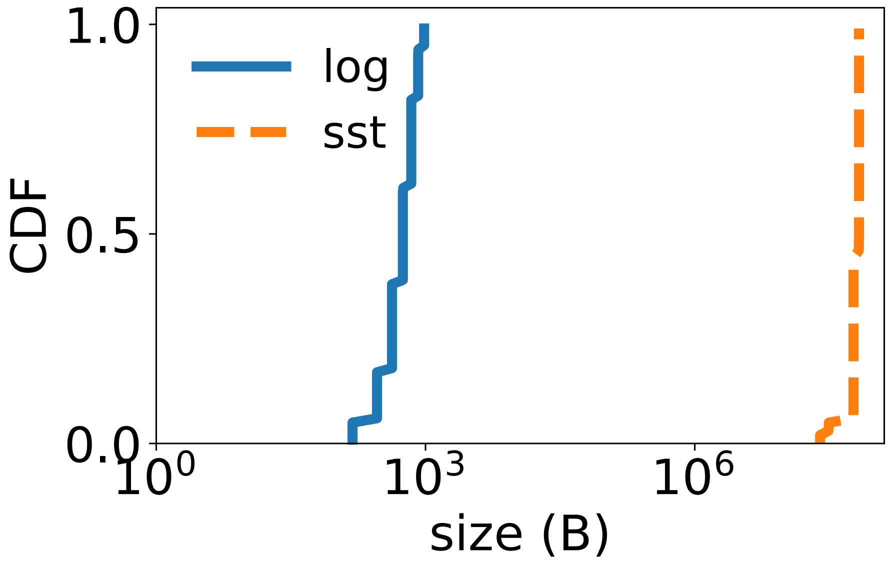
- Small synchronous writes to log: durability, crash recovery
- Bulk asynchronous write to checkpoint: save snapshot, garbage-collect log

Pervasive in many systems: SQLite mongoDB MariaDB

Large Writes vs. Small Writes

Takeaways:

Large Writes vs. Small Writes

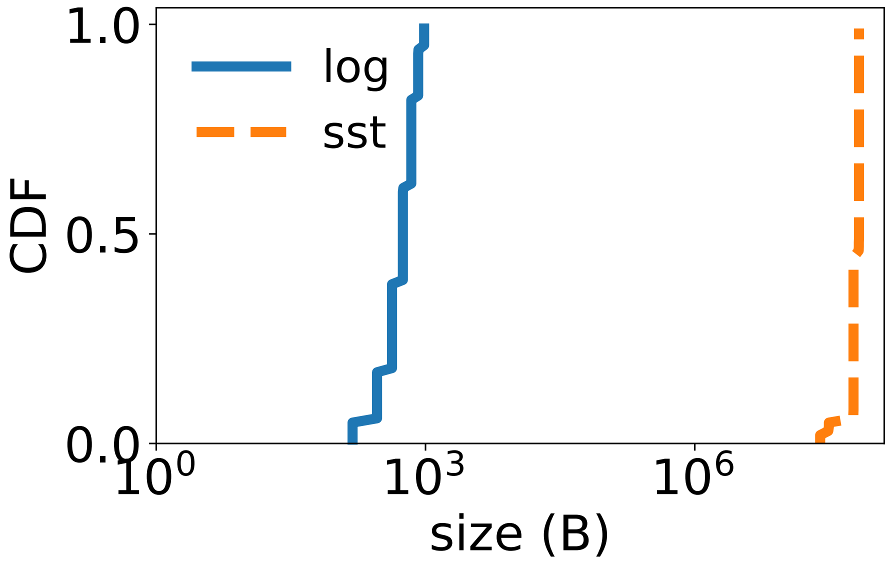


Size of writes in RocksDB

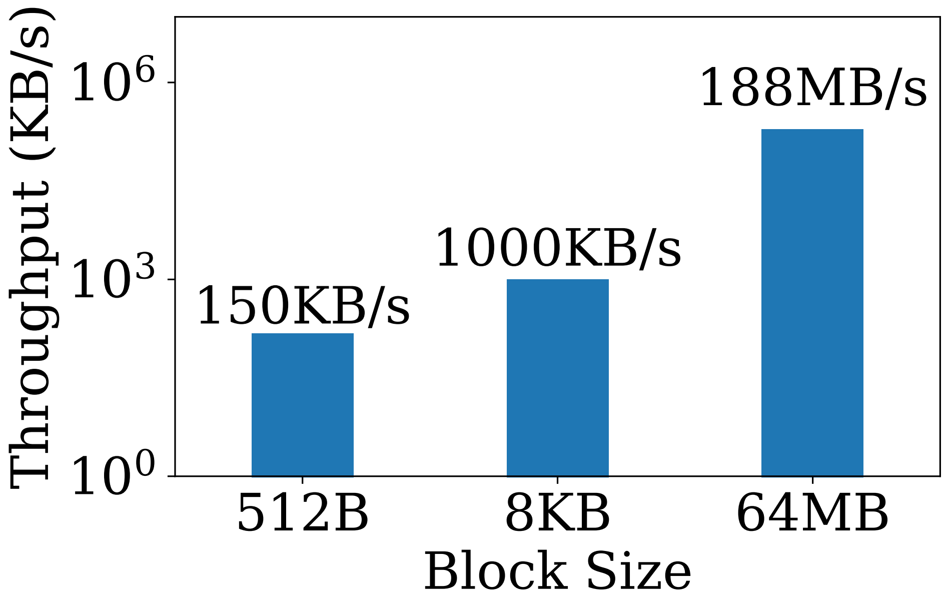
Takeaways:

- Writes to logs are significantly smaller

Large Writes vs. Small Writes



Size of writes in RocksDB

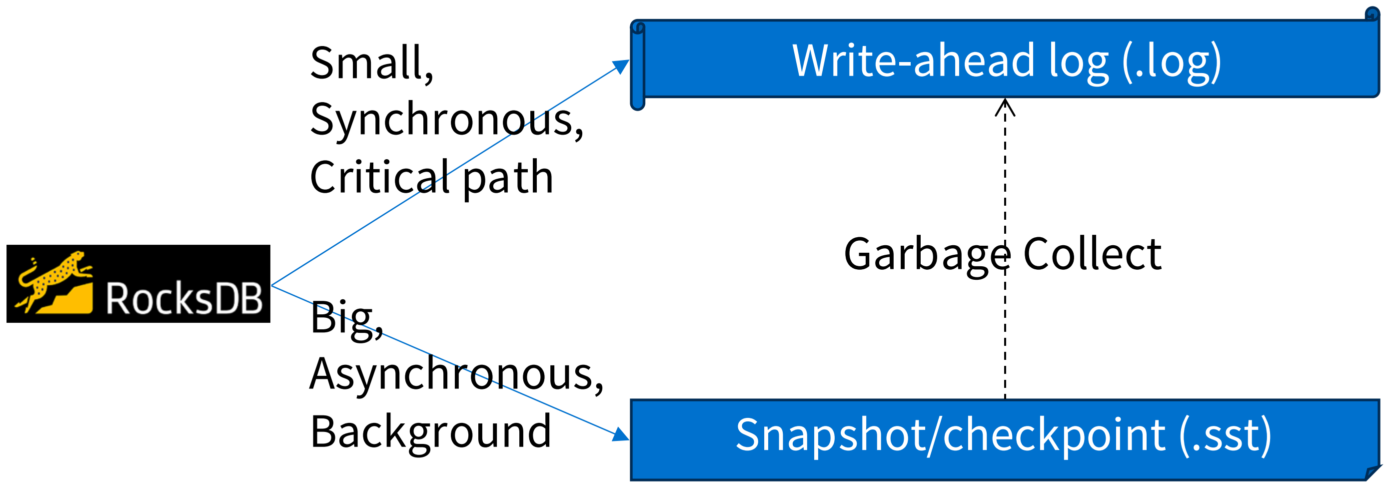


Influence of write size on throughput (sync write)

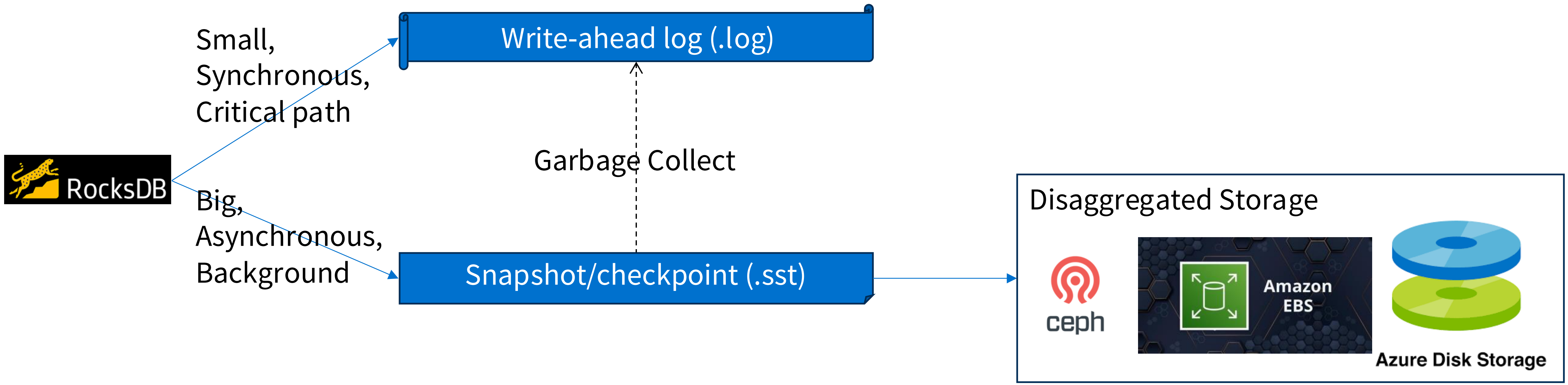
Takeaways:

- Writes to logs are significantly smaller
- Small writes are severely limited in throughput, while large writes don't

SplitFT: Split Small and Large Writes

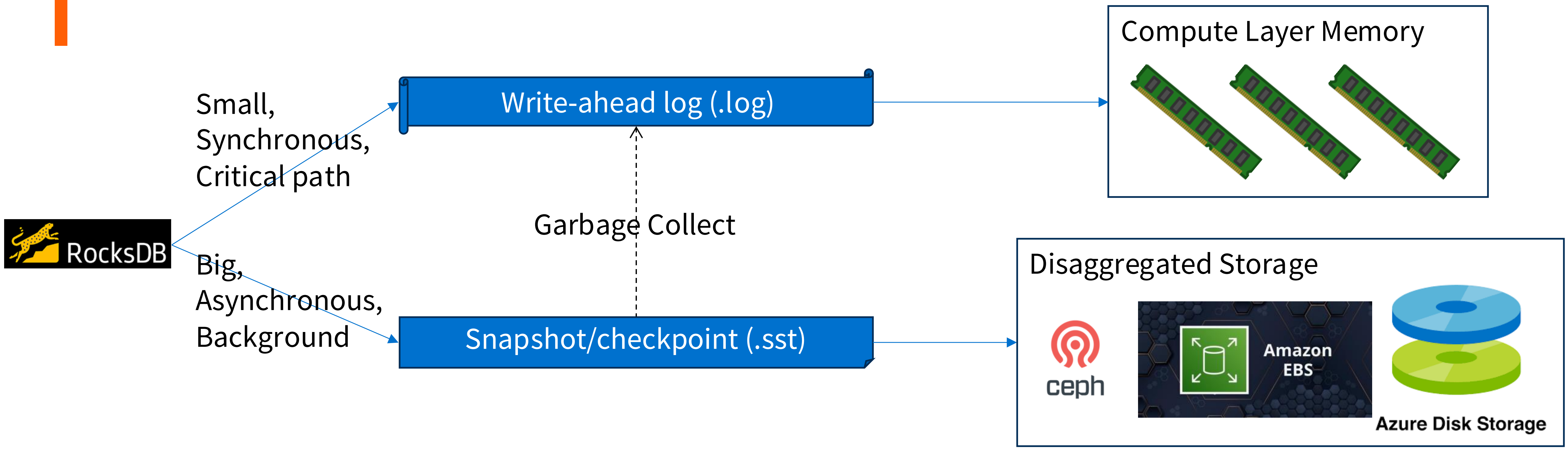


SplitFT: Split Small and Large Writes



- Large writes: directly go to disaggregated storage

SplitFT: Split Small and Large Writes

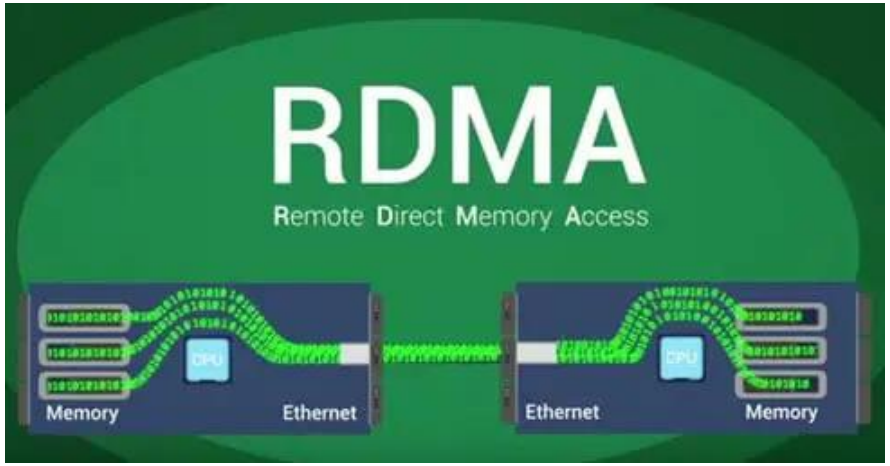


- Large writes: directly go to disaggregated storage
- Small writes: made fault-tolerant within the compute layer
  - A new abstraction called ***Near-Compute Log (NCL)***

Why is NCL Possible and Effective

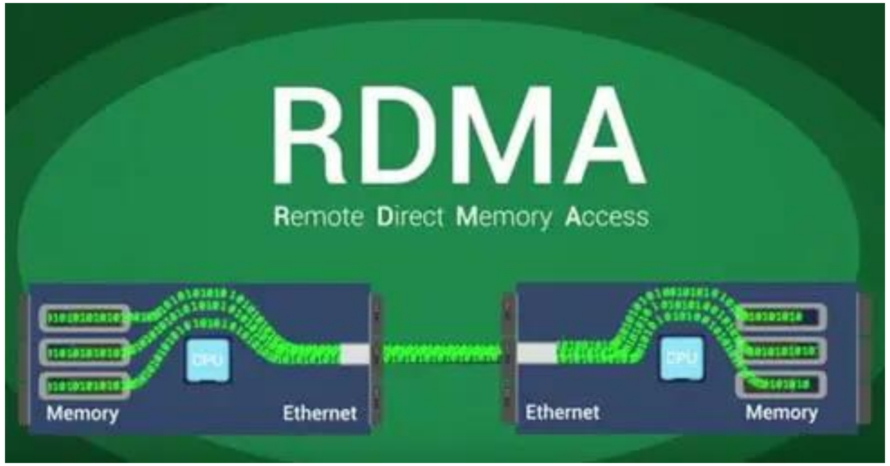
Why is NCL Possible and Effective

- Ubiquitous Low latency networking
- CPU-free remote memory access



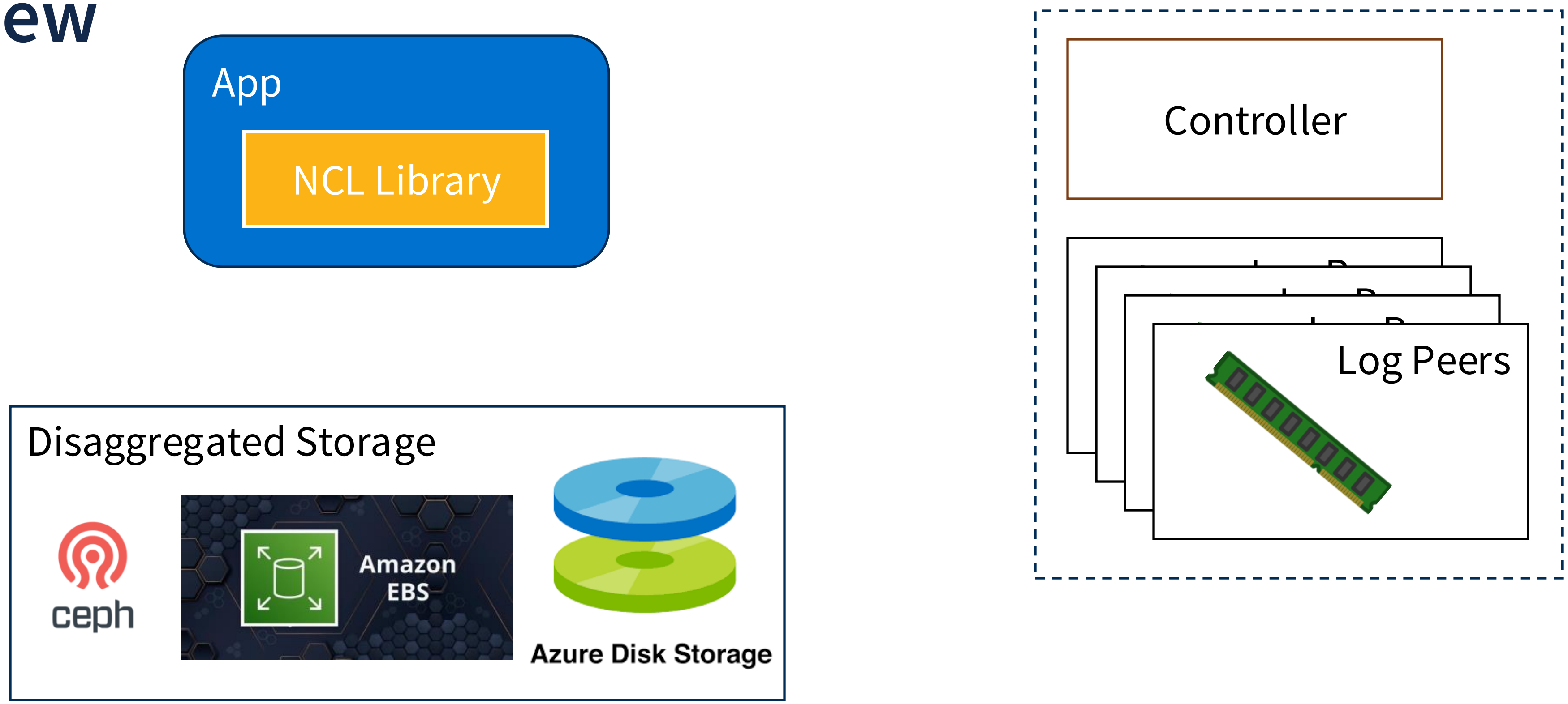
Why is NCL Possible and Effective

- Ubiquitous Low latency networking
- CPU-free remote memory access
- Memory is largely underutilized in data centers[1,2,3]
  - A new use case for remote memory: logging small writes

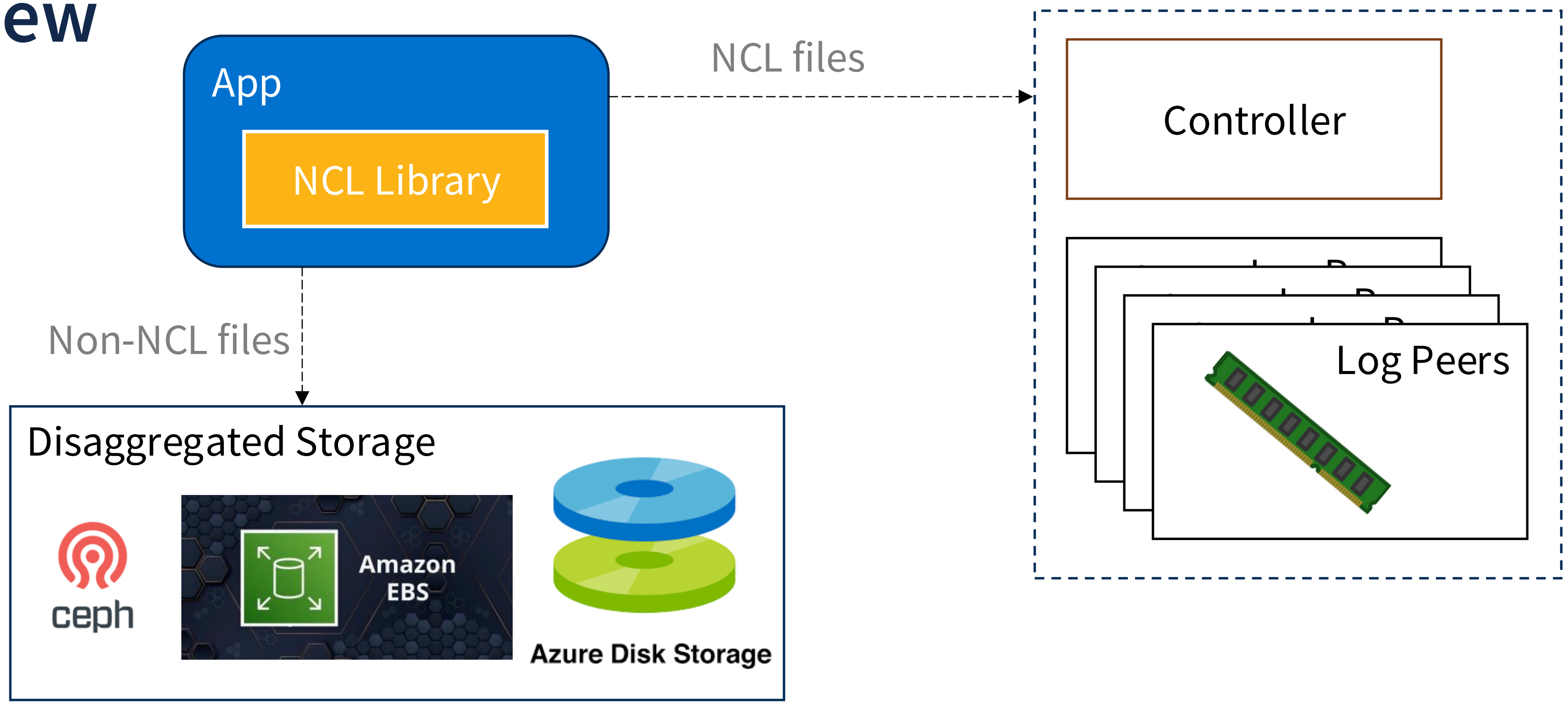


[1] *Redy: remote dynamic memory cache*, Zhang et al. VLDB'21
[2] *LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation*, Shan et al. OSDI'18
[3] *Efficient memory disaggregation with infiniswap*, Gu et al. NSDI'17

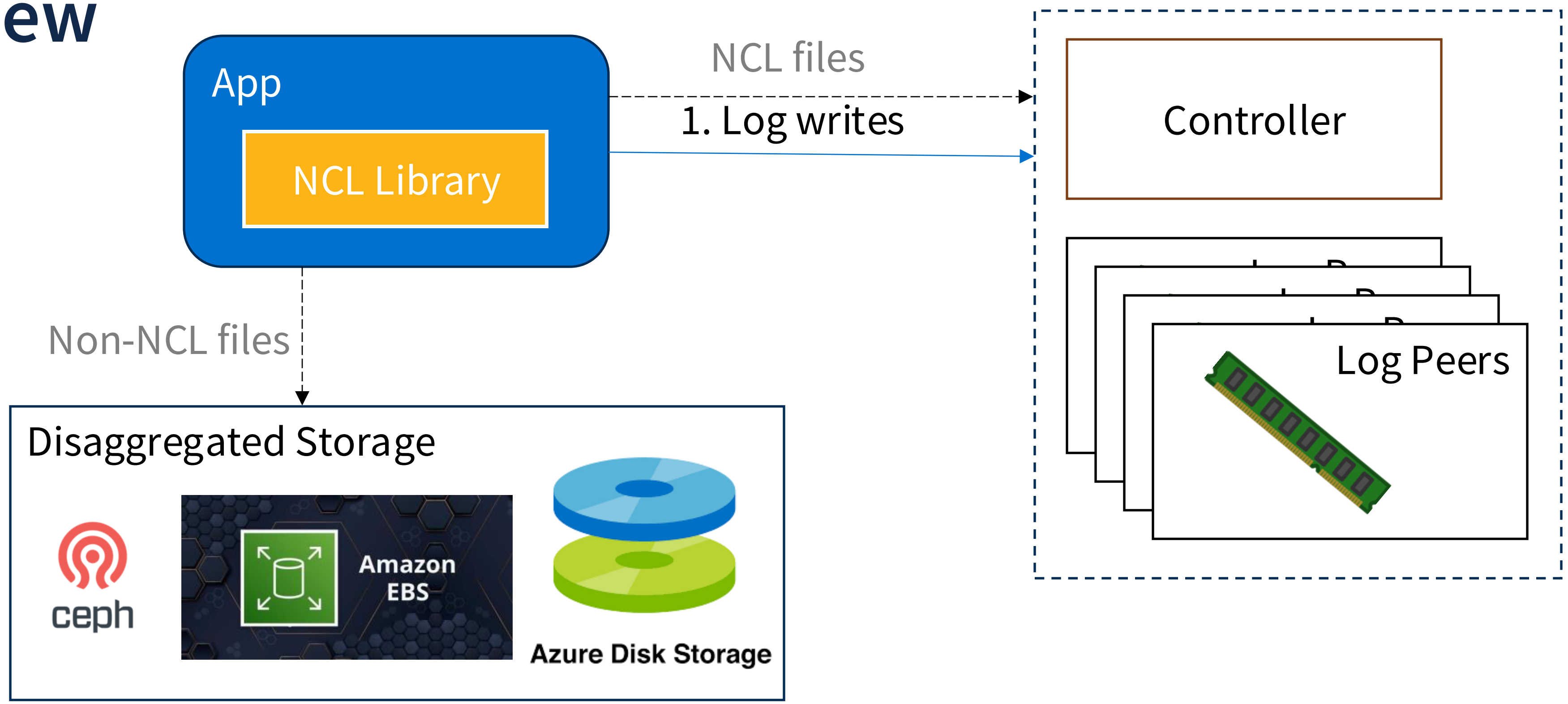
Overview


App
NCL Library
Disaggregated Storage
ceph
Amazon
EBS
Azure Disk Storage
Controller
Log Peers

Overview

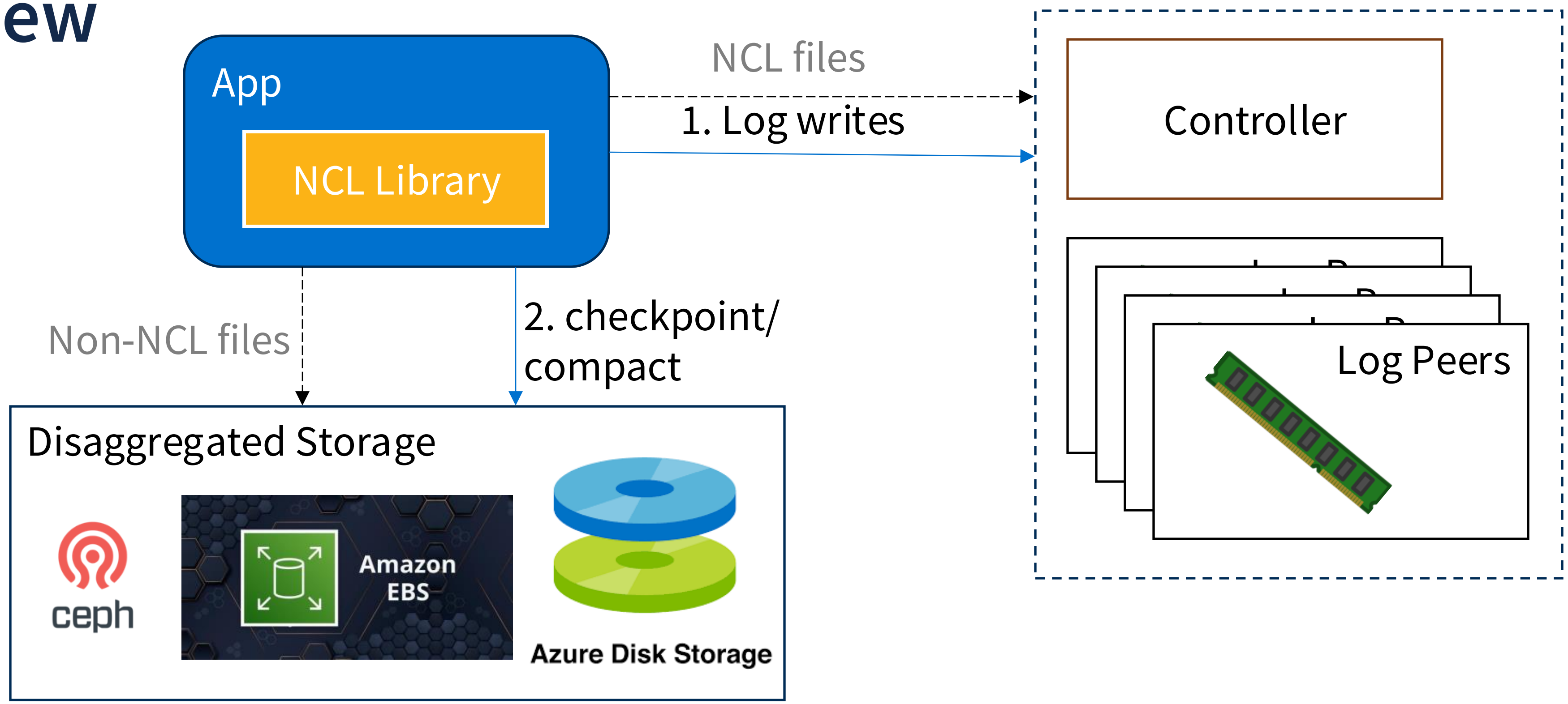

App
NCL Library
NCL files
Controller
Log Peers
Non-NCL files
Disaggregated Storage
ceph
Amazon
EBS
Azure Disk Storage

Overview



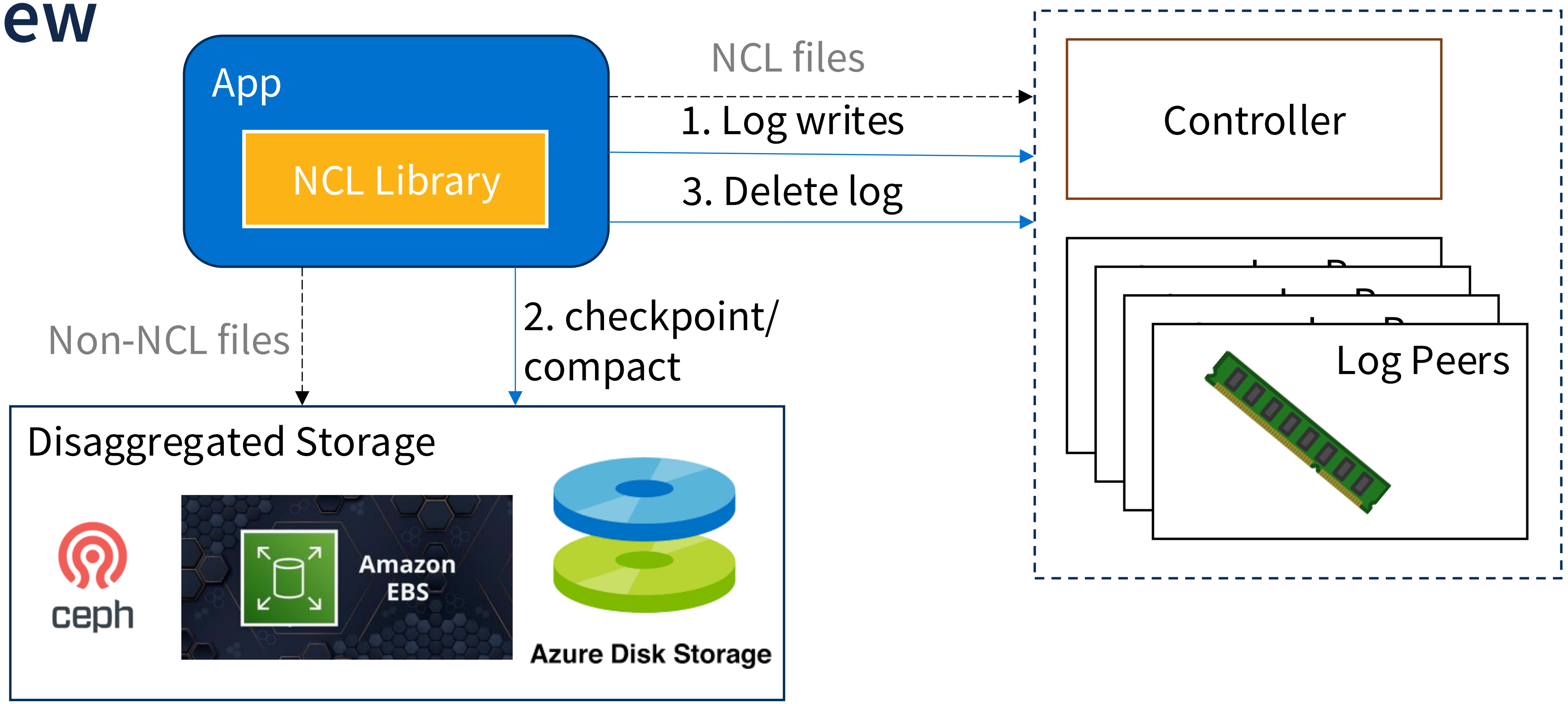
1. Sync small writes are sent to NCL layer

Overview



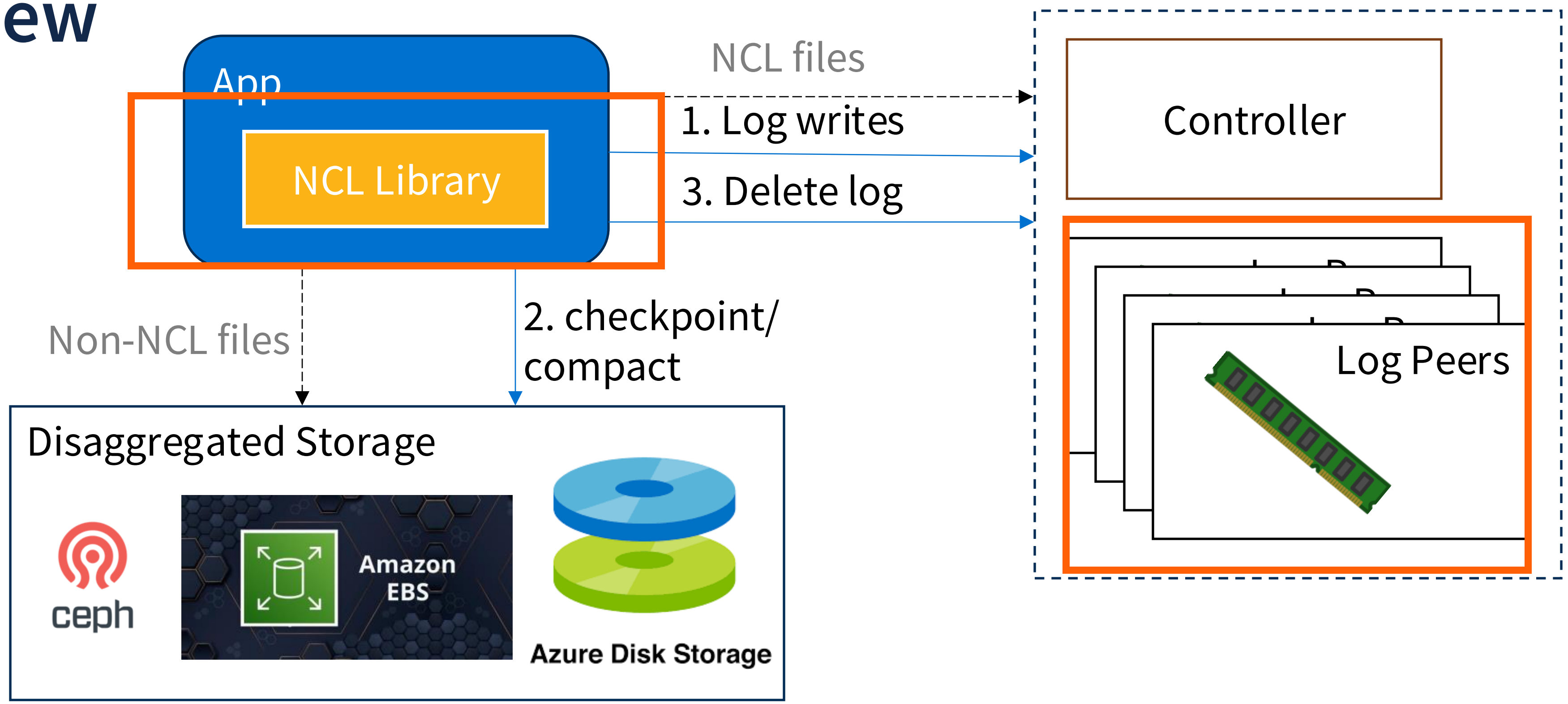
1. Sync small writes are sent to NCL layer
2. Checkpoint or compact states to disaggregated storage

Overview



1. Sync small writes are sent to NCL layer
2. Checkpoint or compact states to disaggregated storage
3. Logs are garbage-collected from NCL layer

Overview



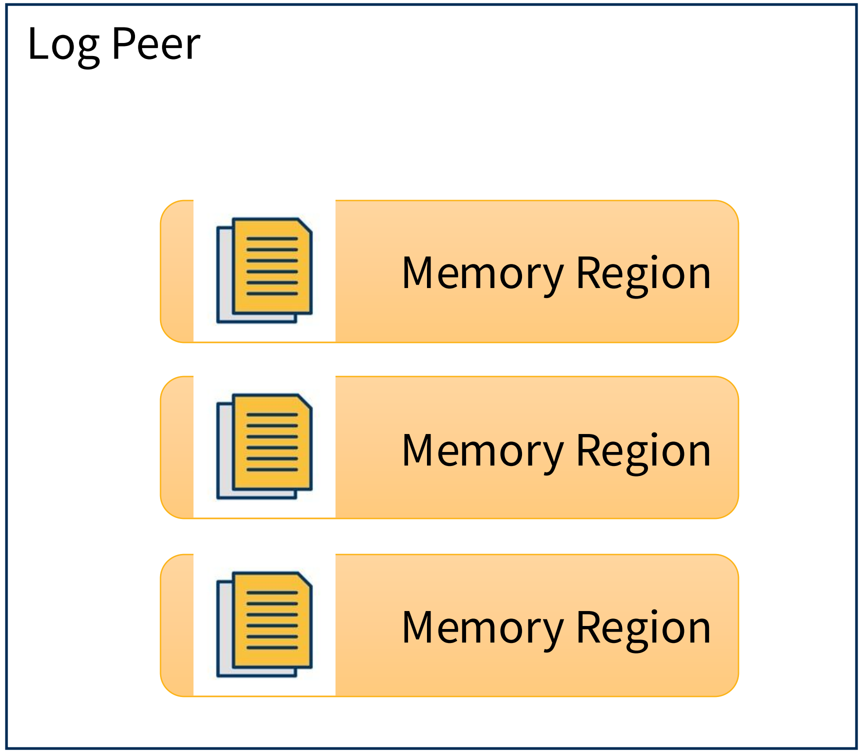
1. Sync small writes are sent to NCL layer
2. Checkpoint or compact states to disaggregated storage
3. Logs are garbage-collected from NCL layer

Log Peer

Log Peer

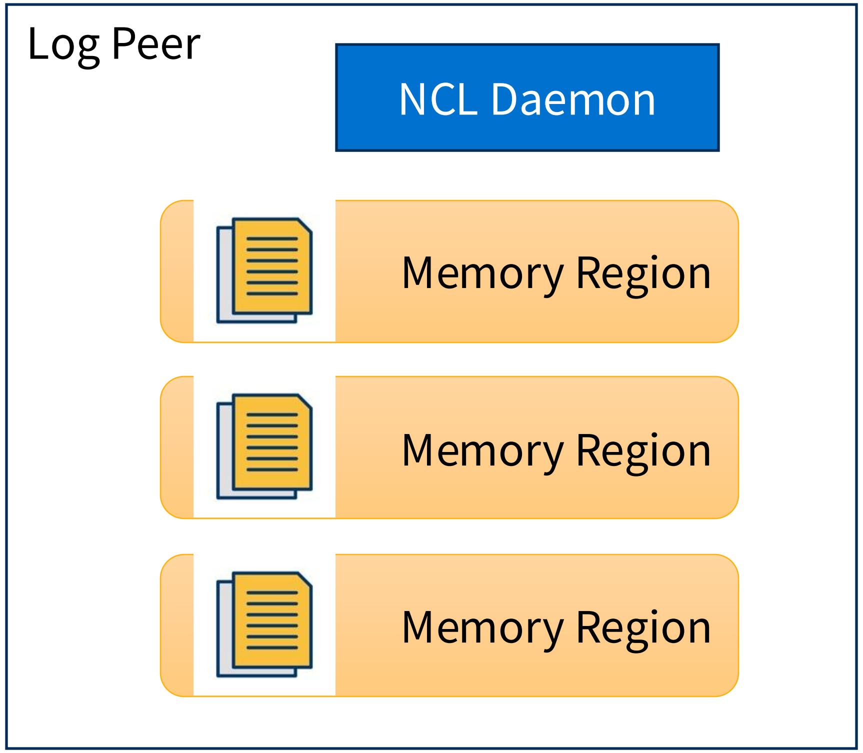
Log Peer

- Lend spare memory for the use of NCL



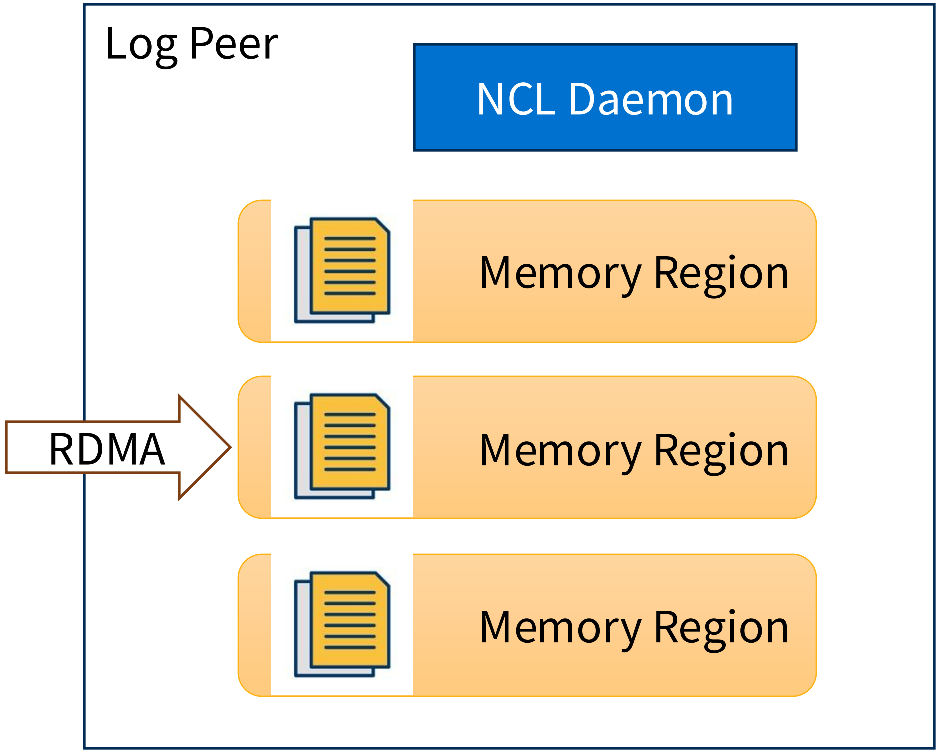
Log Peer

- Lend spare memory for the use of NCL
- Runs a NCL daemon process that manages NCL replica data on it



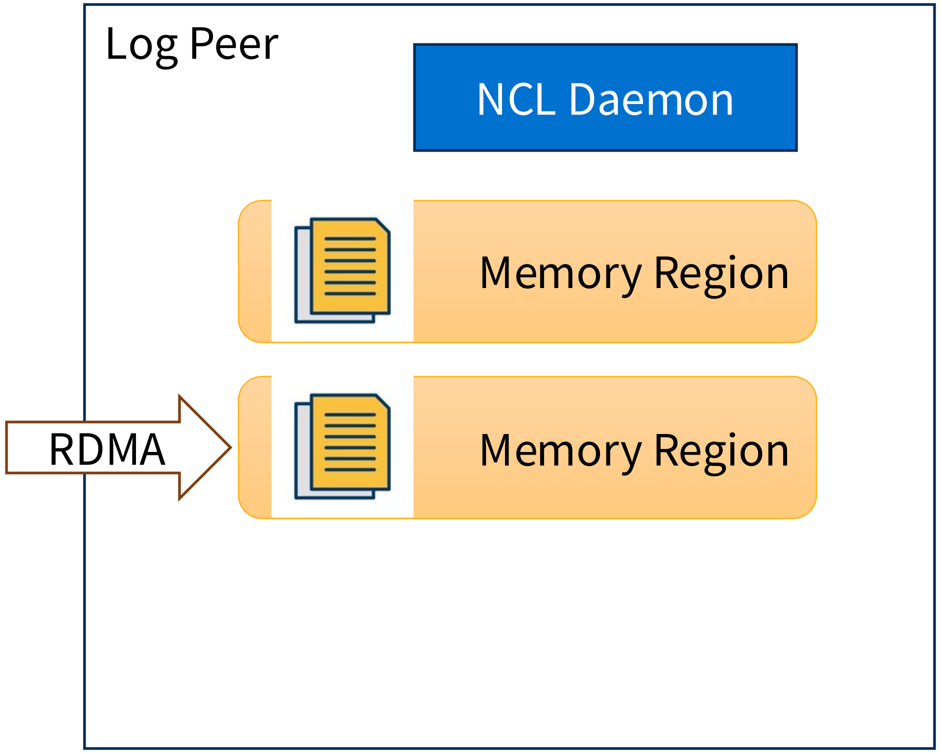
Log Peer

- Lend spare memory for the use of NCL
- Runs a NCL daemon process that manages NCL replica data on it
- Use RDMA, no CPU cycles are spent during regular write operations
  - Passive memory units
  - CPU is only used during initial setup

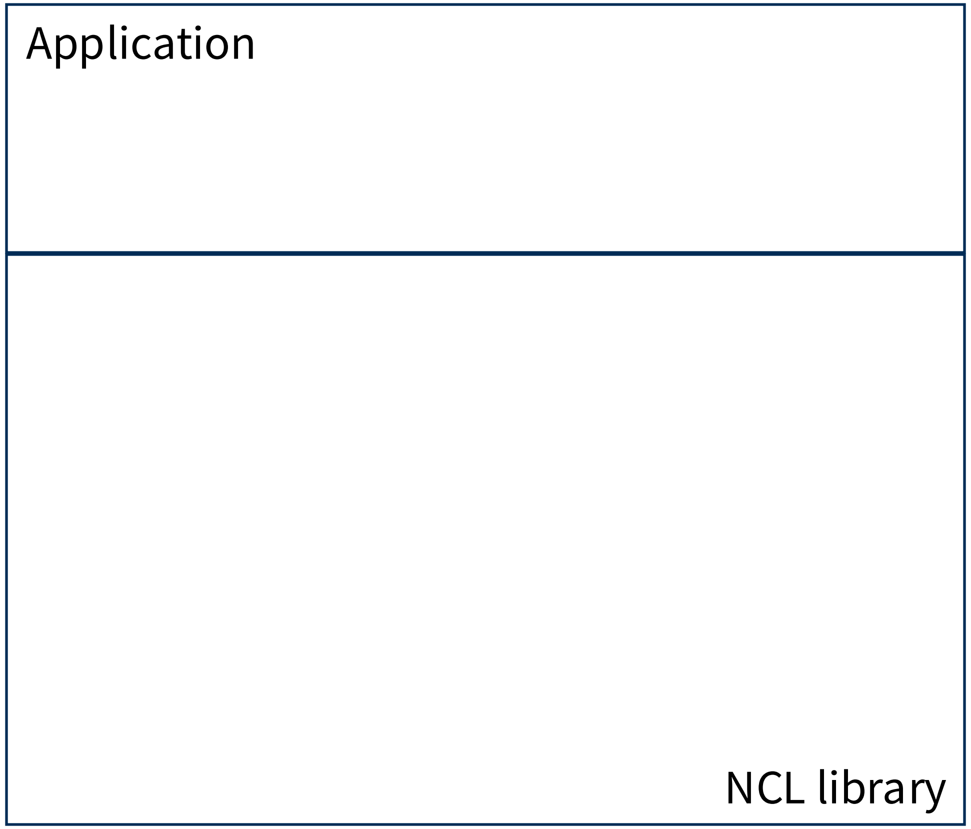


Log Peer

- Lend spare memory for the use of NCL
- Runs a NCL daemon process that manages NCL replica data on it
- Use RDMA, no CPU cycles are spent during regular write operations
  - Passive memory units
  - CPU is only used during initial setup
- Can reclaim the lent-out memory at any time

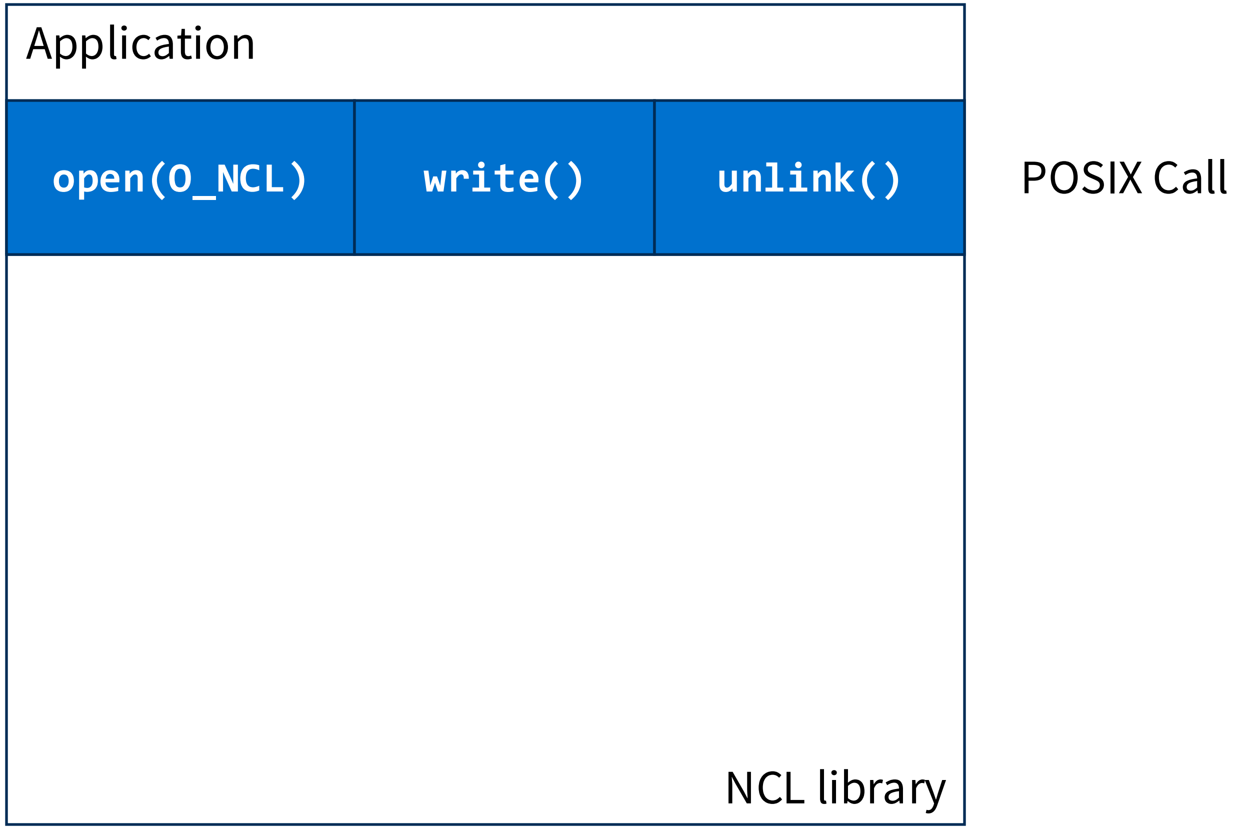


Client Library



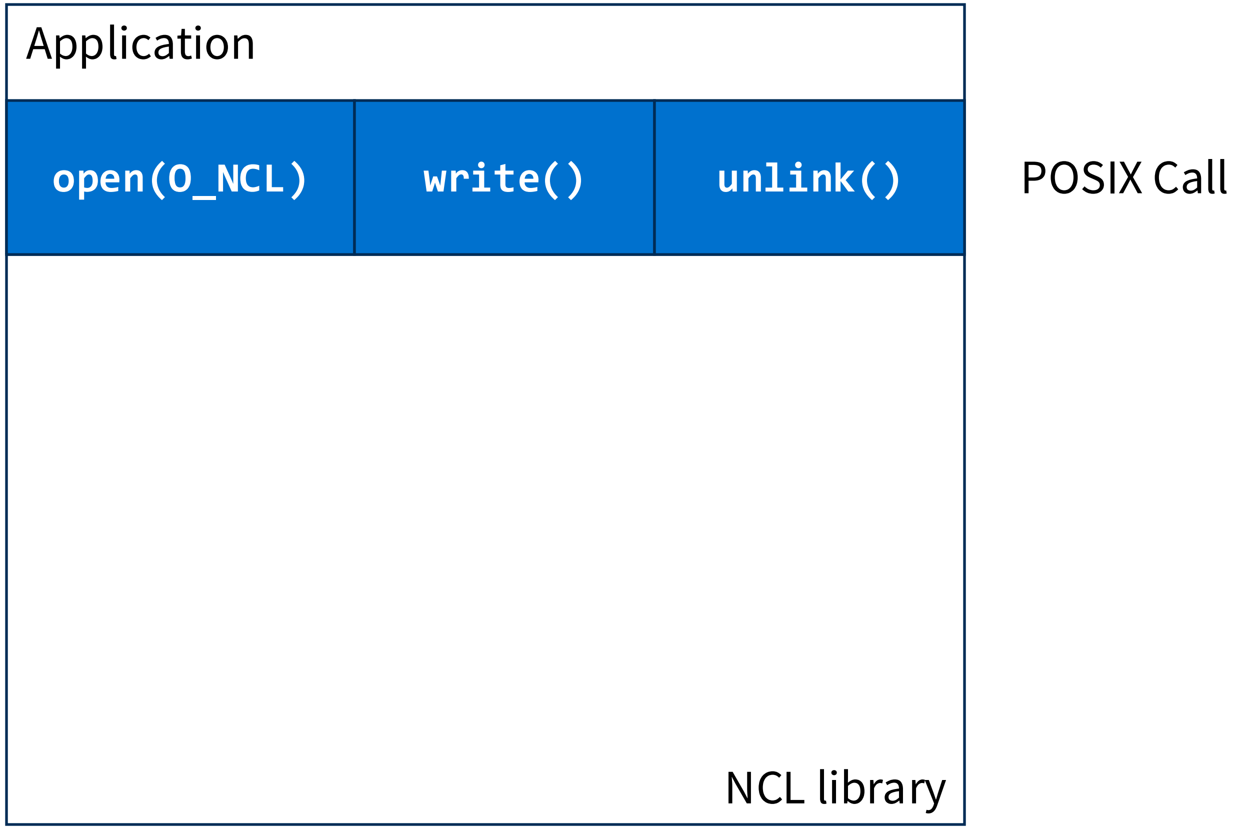
Client Library

- Rewrite POSIX file interface: write, open, unlink



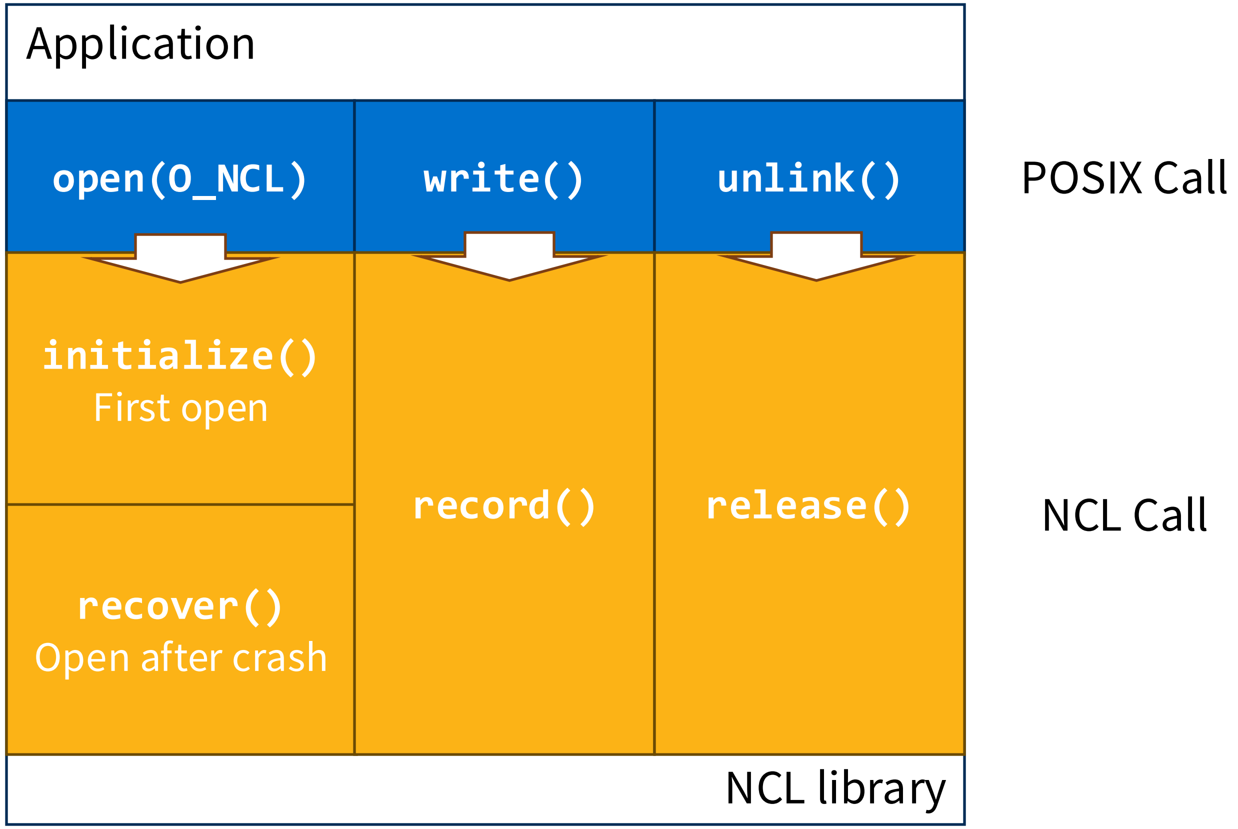
Client Library

- Rewrite POSIX file interface: write, open, unlink
- File-level classification
  - Specific open flag: **`O_NCL`**



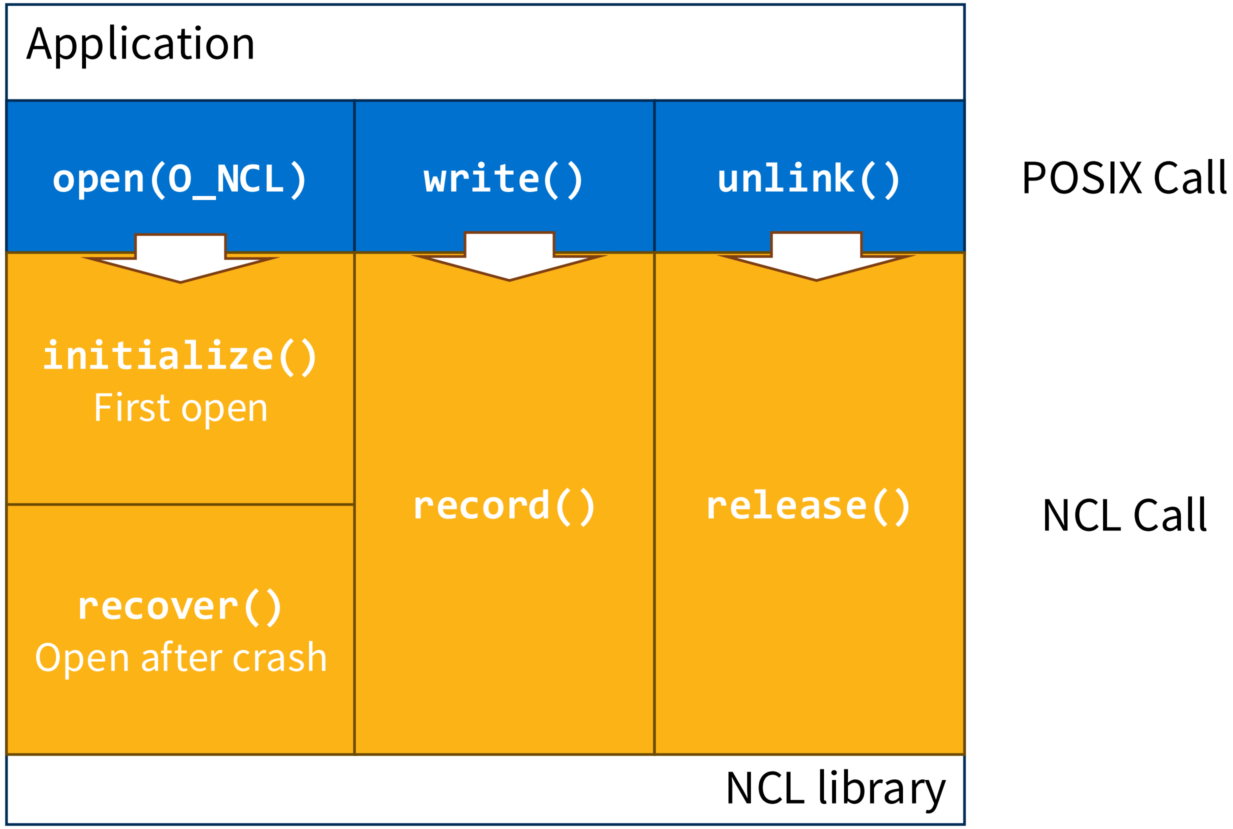
Client Library

- Rewrite POSIX file interface: write, open, unlink
- File-level classification
  - Specific open flag: **O_NCL**
- Preload at application start to override glibc implementation of certain POSIX calls



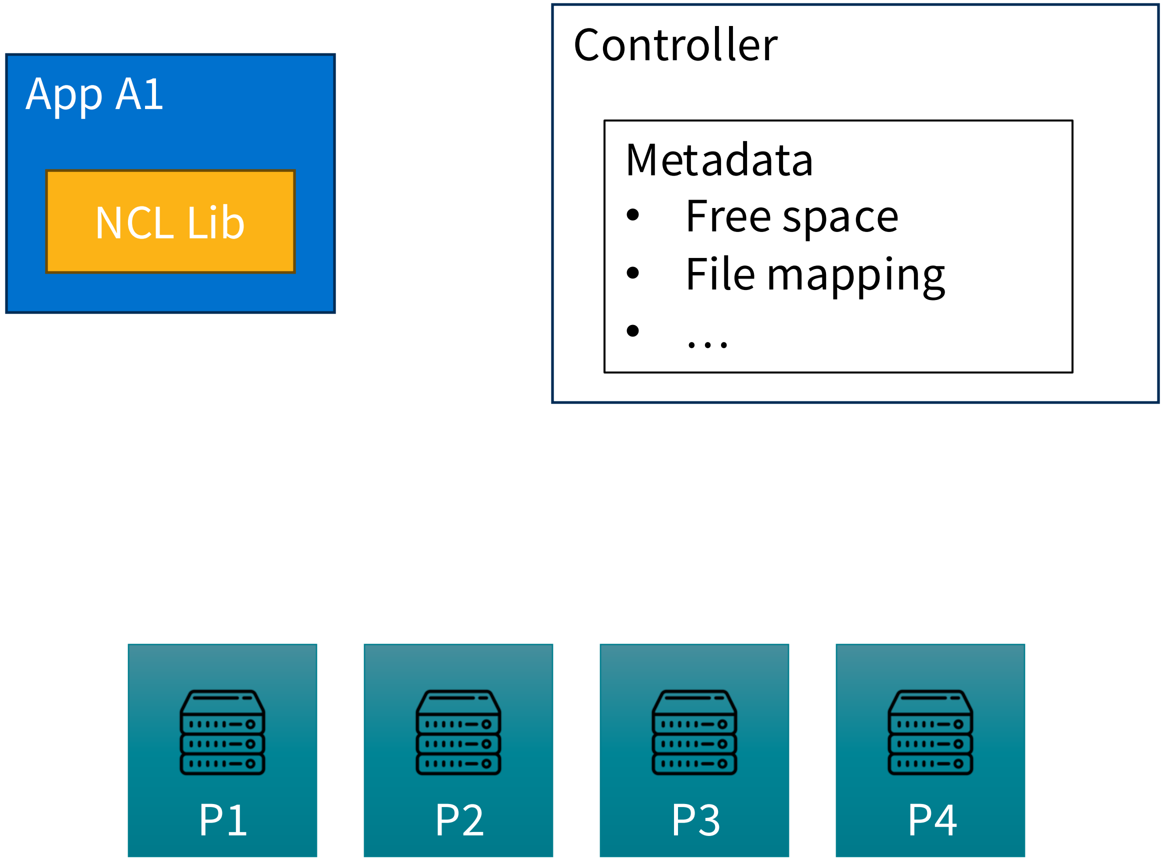
Client Library

- Rewrite POSIX file interface: write, open, unlink
- File-level classification
    - Specific open flag: **O_NCL**
- Preload at application start to override glibc implementation of certain POSIX calls
- Transparent to application



NCL Initialization Process

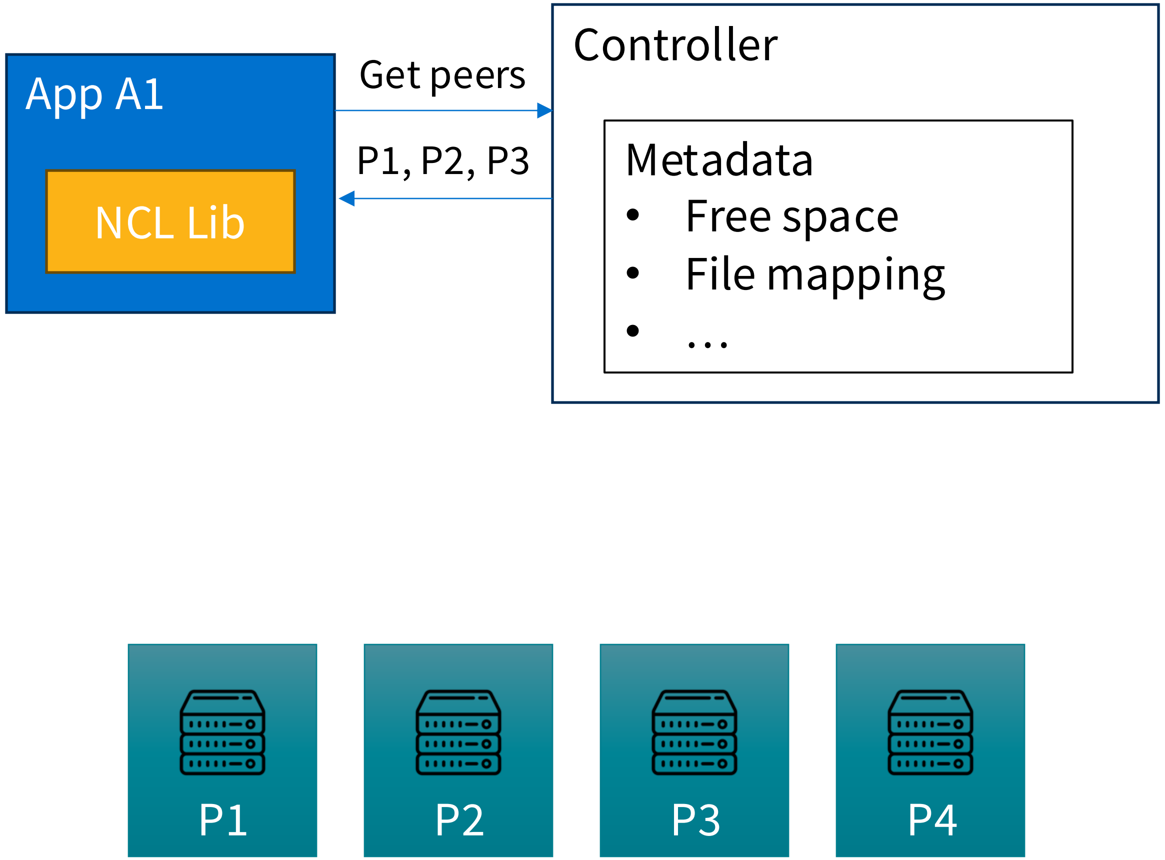
Upon opening the file



NCL Initialization Process

Upon opening the file

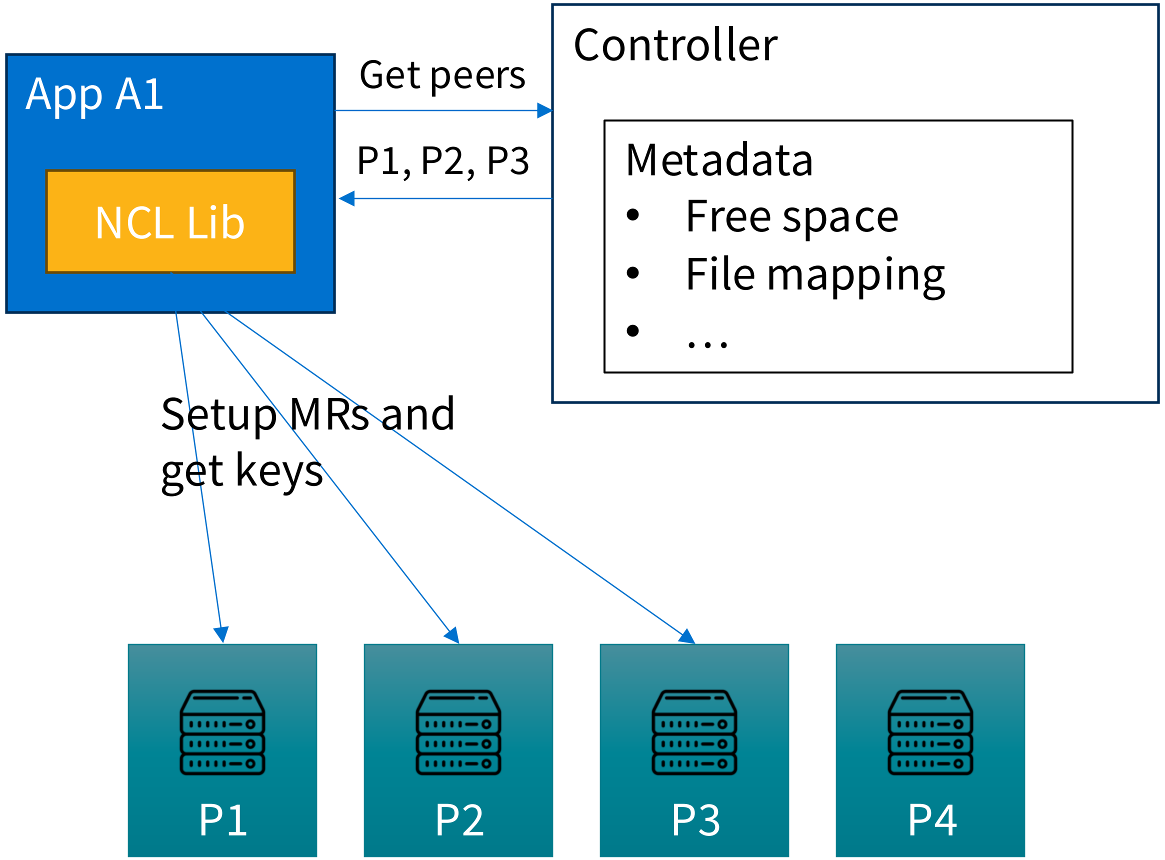
1. Client asks controller for the list of peers



NCL Initialization Process

Upon opening the file

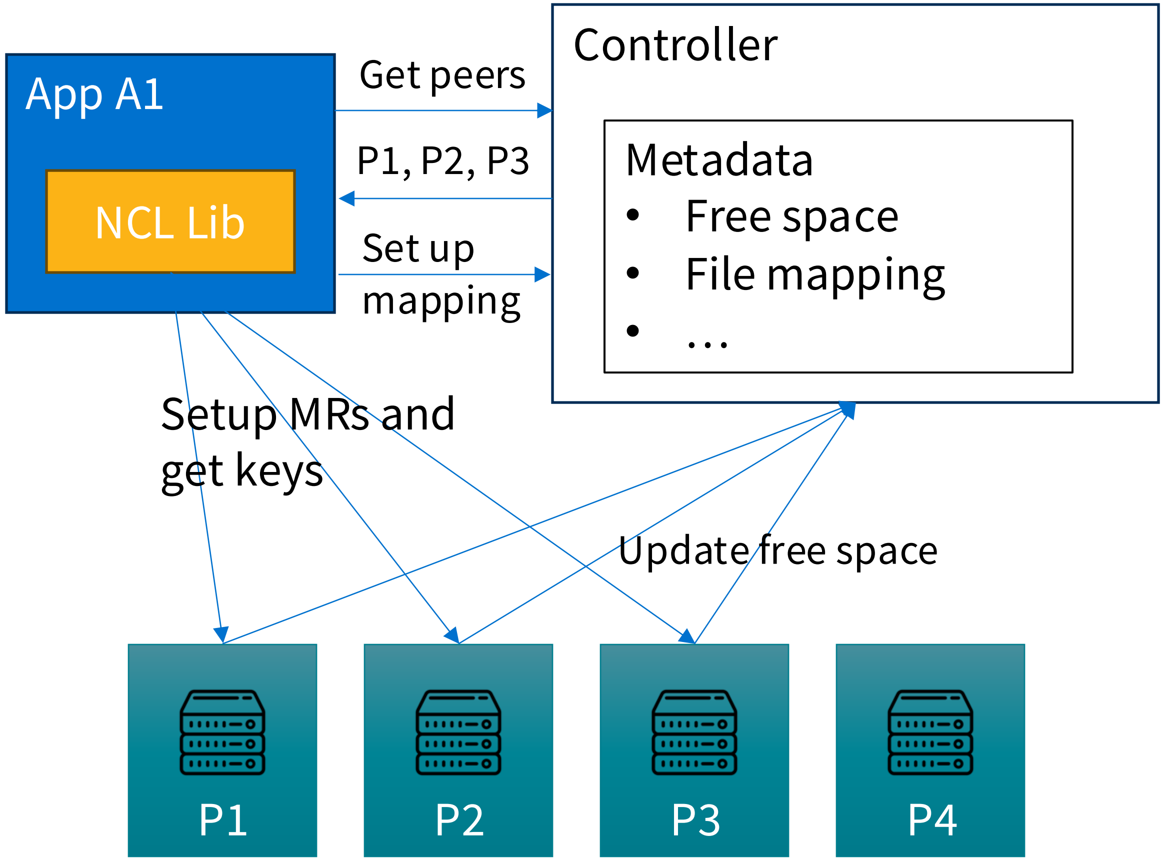
1. Client asks controller for the list of peers
2. Client contacts peers to setup MR and get RDMA keys



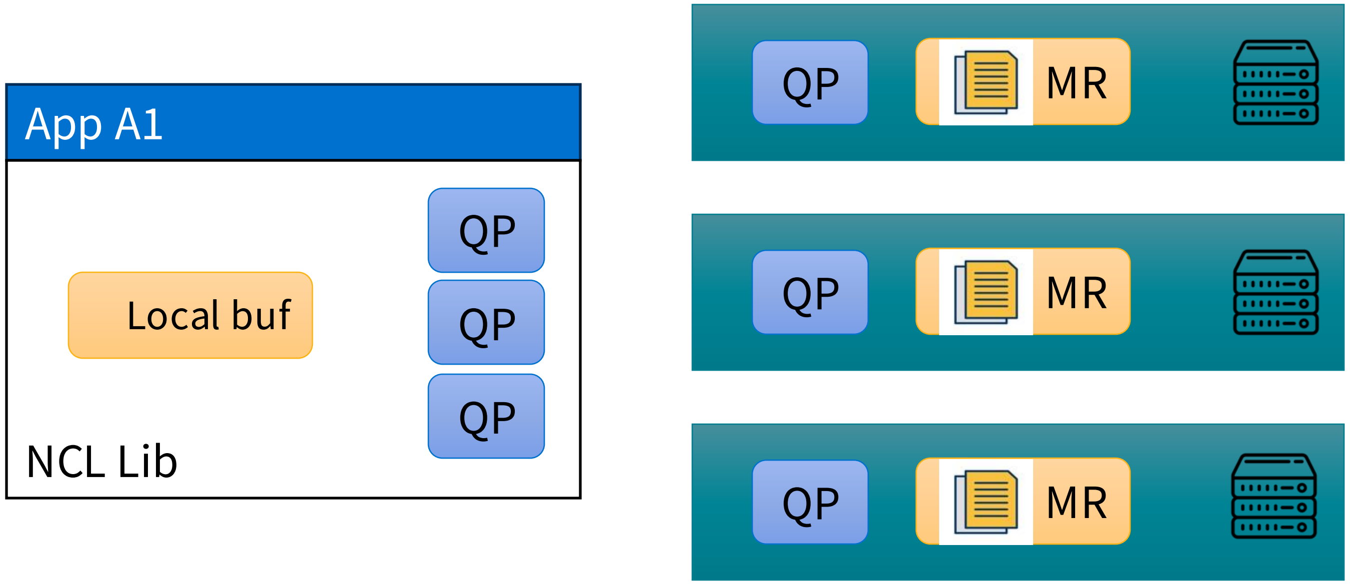
NCL Initialization Process

Upon opening the file

1. Client asks controller for the list of peers
2. Client contacts peers to setup MR and get RDMA keys
3. Update metadata on Controller

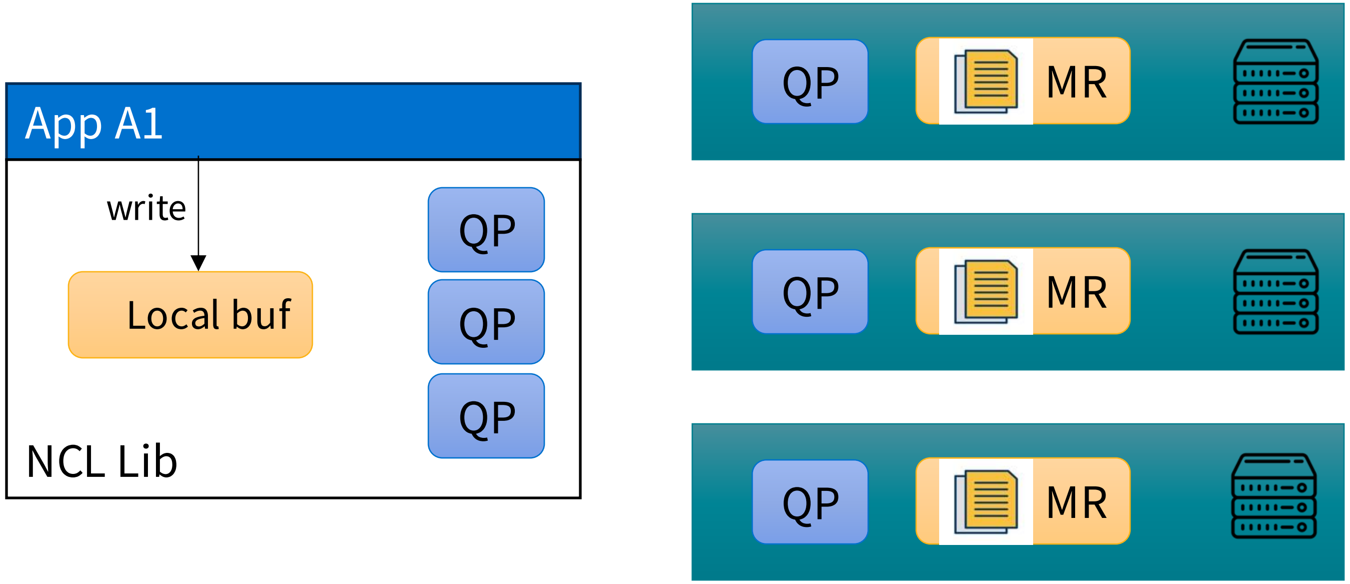


Replication Process



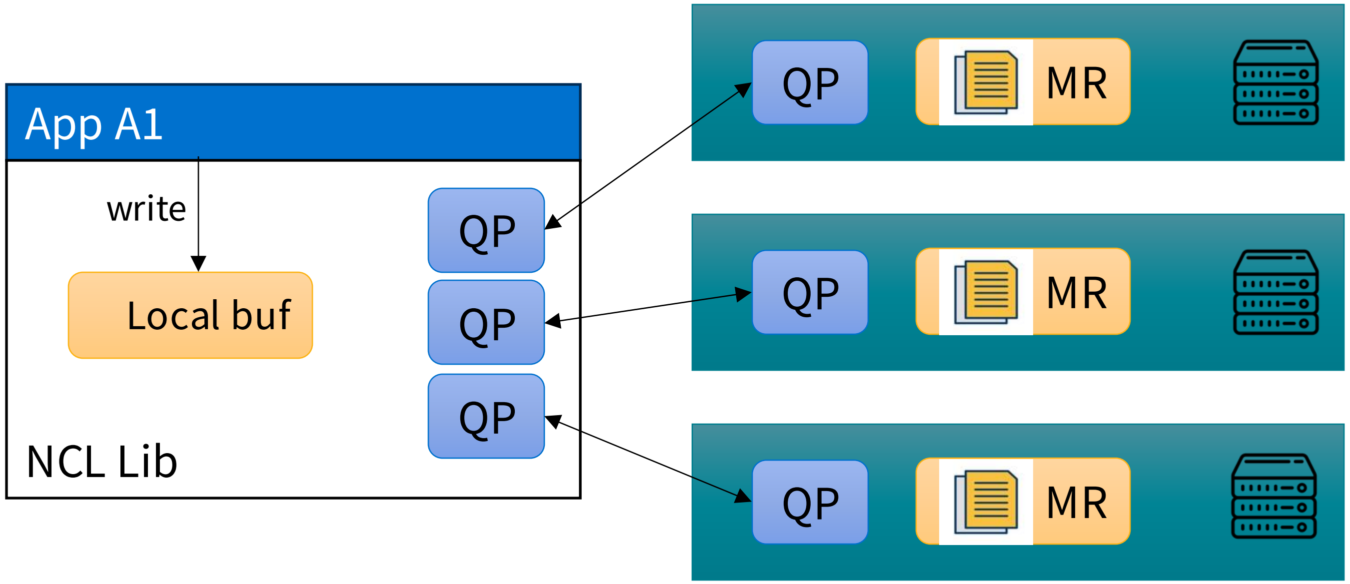
Replication Process

1. Write to local buffer



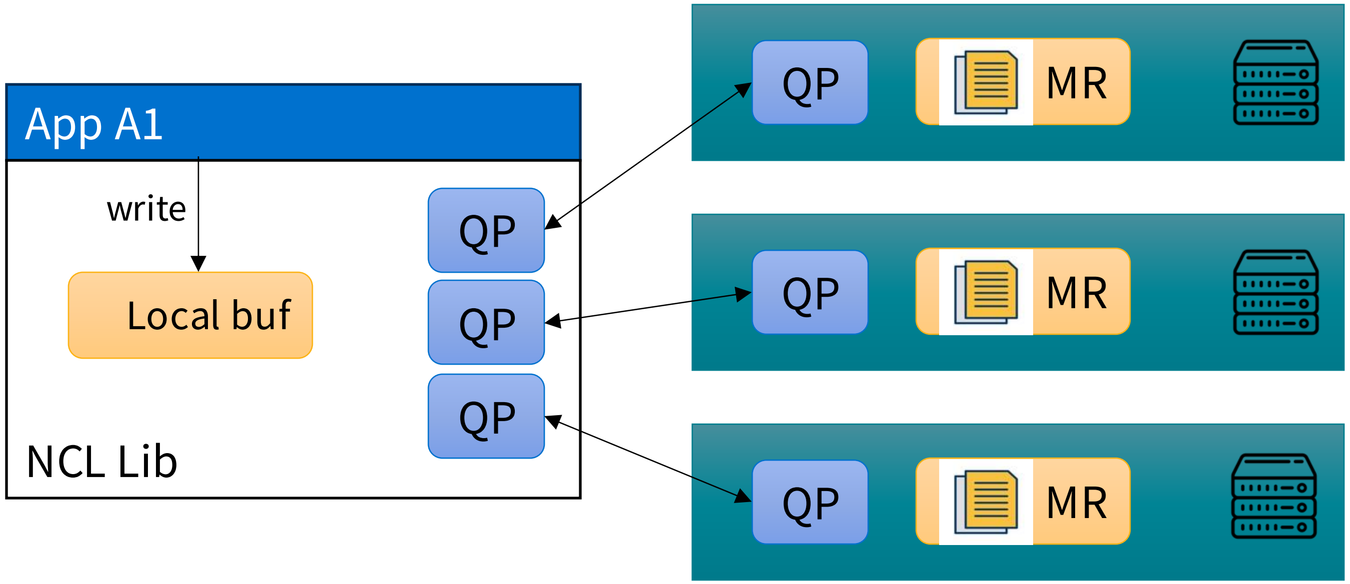
Replication Process

1. Write to local buffer
2. RDMA write to all log peers



Replication Process

1. Write to local buffer
2. RDMA write to all log peers
3. Write returns after replicated on a majority of log peers



Failure Handling

- Application failure
- Log peer failure

Please check the paper for details.

Questions to Answer in Evaluation

- How do applications perform in SplitFT compare to DFT for write-only workload?
- How do applications perform in SplitFT under different YCSB workloads?
- How quickly do applications in SplitFT recover?

Setup

Setup

8* CloudLab xl170 machines

- 10-core(20-thread) CPU
- 64GB Memory
- 480GB SATA SSD

- 1x client machine
- 1x server machine
- 3x CephFS replicas
- 3x log peers

Setup

8* CloudLab xl170 machines

- 10-core(20-thread) CPU
- 64GB Memory
- 480GB SATA SSD

- 1x client machine
- 1x server machine
- 3x CephFS replicas
- 3x log peers

Port 3 Database Applications

1. RocksDB (only 10 LoC change)
2. Redis (only 19 LoC change)
3. SQLite (only 6 LoC change)

Setup

8* CloudLab xl170 machines

- 10-core(20-thread) CPU
- 64GB Memory
- 480GB SATA SSD

- 1x client machine
- 1x server machine
- 3x CephFS replicas
- 3x log peers

Port 3 Database Applications

1. RocksDB (only 10 LoC change)
2. Redis (only 19 LoC change)
3. SQLite (only 6 LoC change)

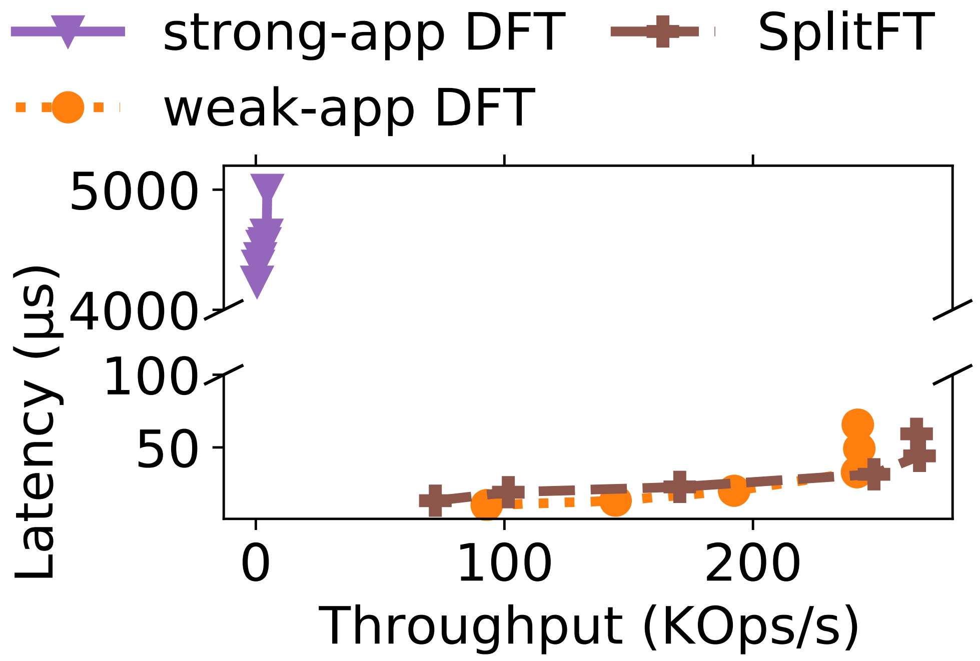
Baseline

- Strong-app DFT: synchronous log write
- Weak-app DFT: asynchronous log write
- SplitFT: NCL

Insert-only Latency and Throughput

Insert-only workload, SplitFT has:

- Same level throughput as weak-app DFT
    - With stronger durability
- Significantly faster than strong-app DFT

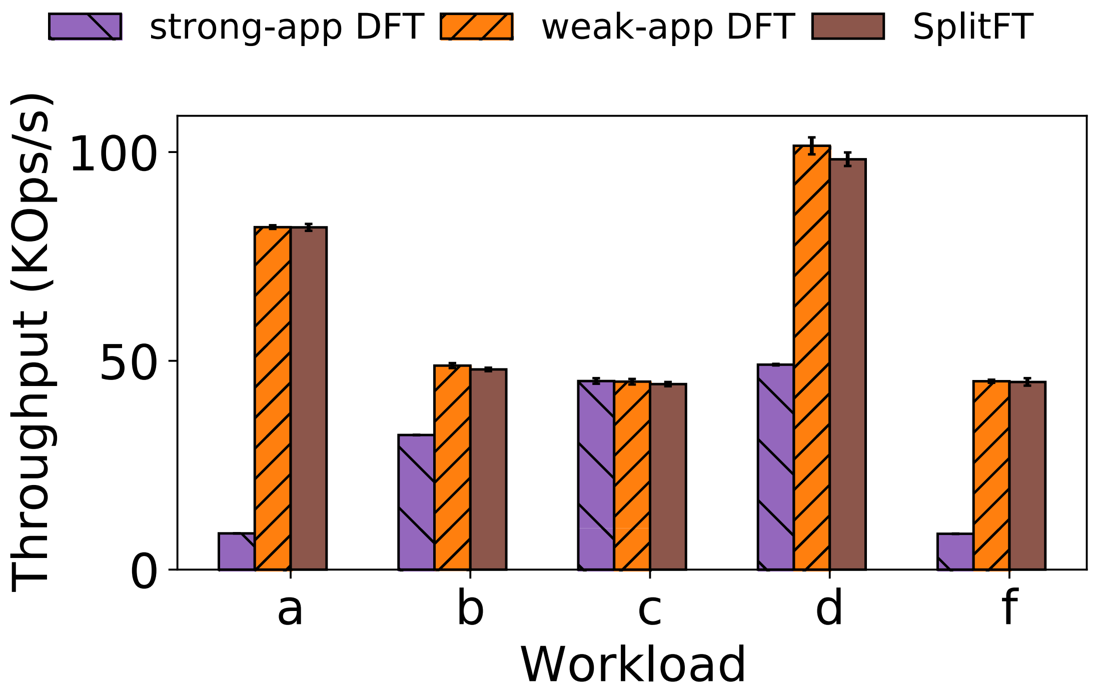


RocksDB Latency vs. Throughput

YCSB Throughput

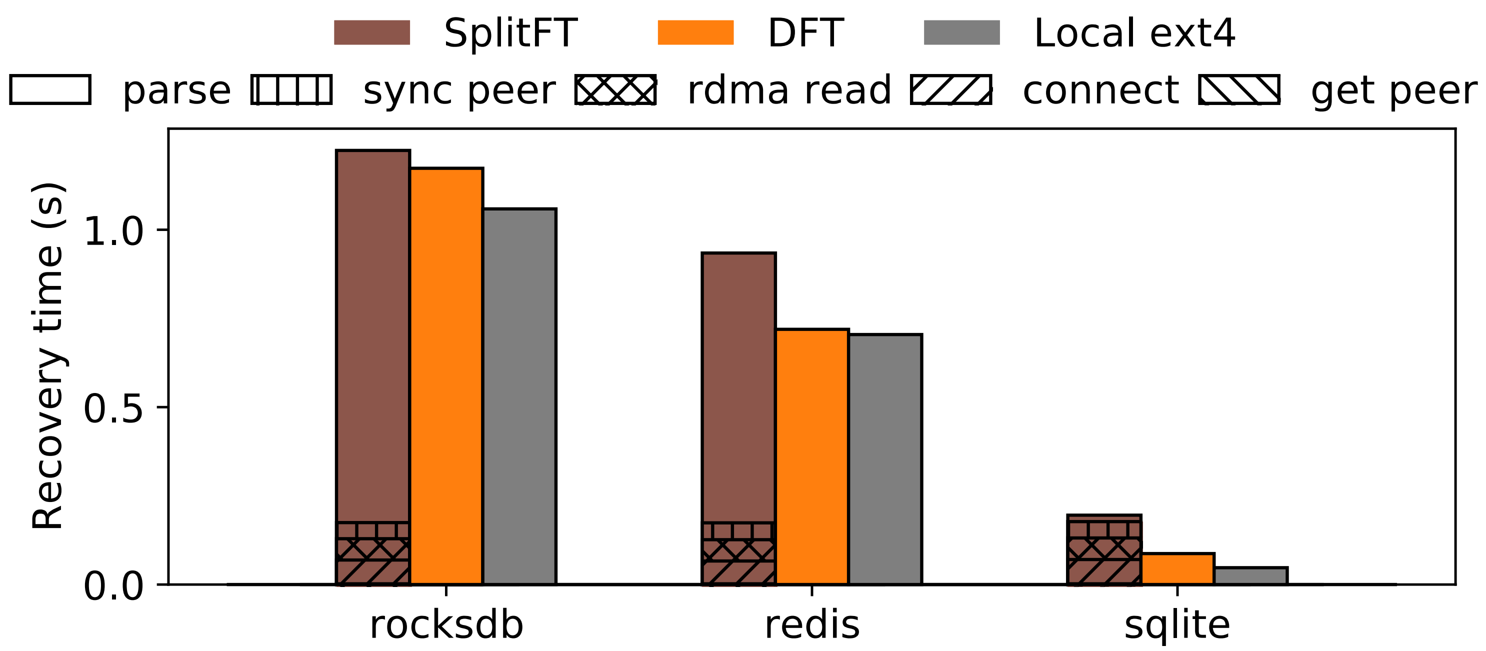
SplitFT has

- Only 0.1% to 3.2% throughput downgrade than weak-app DFT
- Much higher throughput than strong-app DFT in update-heavy workload (A & F)



RocksDB YCSB Throughput

Application Recovery Speed



Application recovery time with 60MB of log

Similar level of log recovery speed as DFT and local FS for all 3 applications

Conclusion

- Introduce SplitFT, a new fault-tolerance approach for storage-centric applications to achieve both performance and strong guarantee
- Split the fault-tolerance of large, bulk writes from small, frequent writes
- NCL, a new abstraction for replicating small writes using remote memory
- Ported and evaluated 3 popular applications
- New use case for data center spare memory

GitHub

https://github.com/dassl-uiuc/compute-side-log