

Storage Systems (StoSys)

XM_0092

Lecture 9: Distributed / Storage Systems - I


Animesh Trivedi

<https://stonet-research.github.io/>

Autumn 2023, Period 1



Syllabus outline

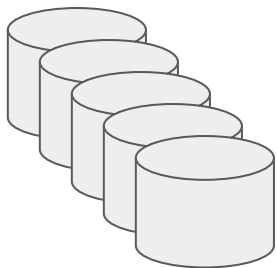
- ~~1. Welcome and introduction to NVM (today)~~
- ~~2. Host interfacing and software implications~~
- ~~3. Flash Translation Layer (FTL) and Garbage Collection (GC)~~
- ~~4. NVM Block Storage File systems~~
- ~~5. NVM Block Storage Key-Value Stores~~
- ~~6. Emerging Byte-addressable Storage~~
- ~~7. Networked NVM Storage~~
- ~~8. Programmable Storage~~
9. Distributed Storage / Systems - I 
10. Distributed Storage / Systems - II
11. Emerging Topics

Today's Agenda

1. We are going to learn about managing temporary/ephemeral data - a new class of data type
2. Building a distributed store with high-performance networking and storage devices
3. Data formats? JSON, Parquet, ORC, are they good enough?

What is Temporary/Ephemeral Data?

Any guesses?

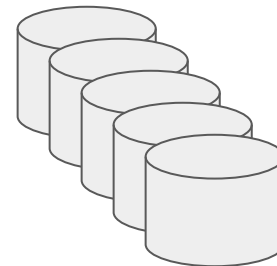


Input datasets



Distributed data processing frameworks

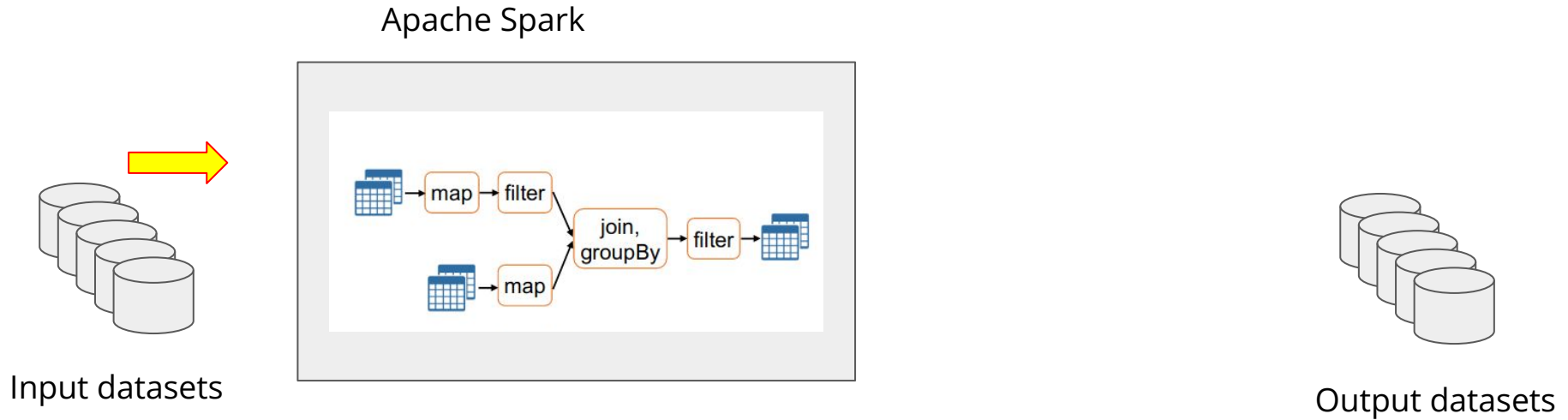
- Apache Spark
- Apache Hadoop (MapReduce)
- GraphLab
- Naiad (Dataflow)
- TensorFlow
- PyTorch
- ...



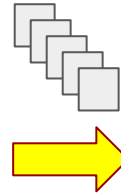
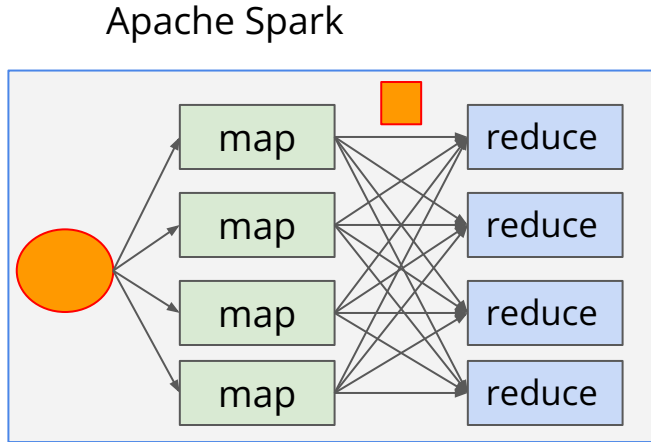
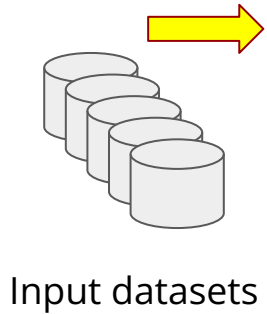
Output datasets

What is Temporary/Ephemeral Data?

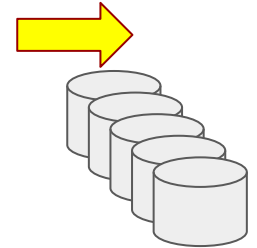
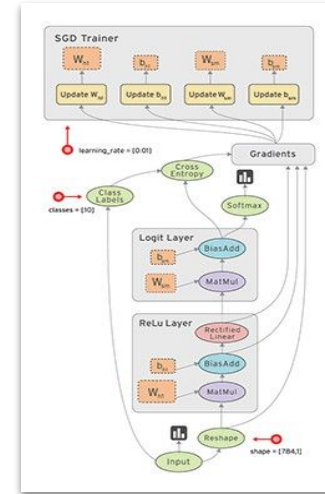
Any guesses?



What is Temporary/Ephemeral Data?



TensorFlow



Output datasets

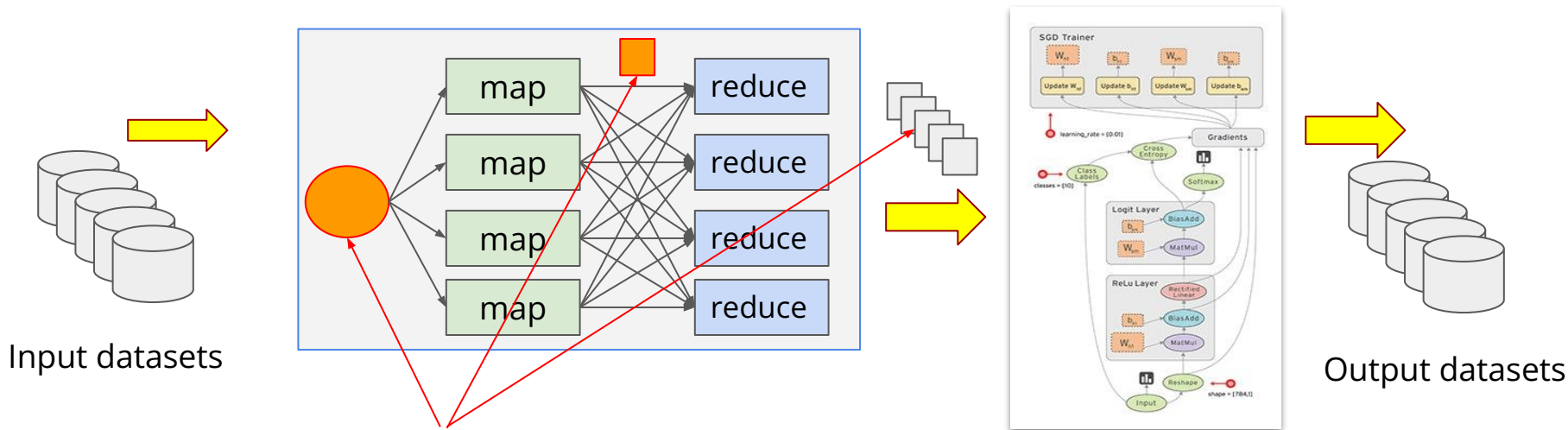
What is Temporary/Ephemeral Data?

1. Read images, transform

2. Feature extraction

3. Training

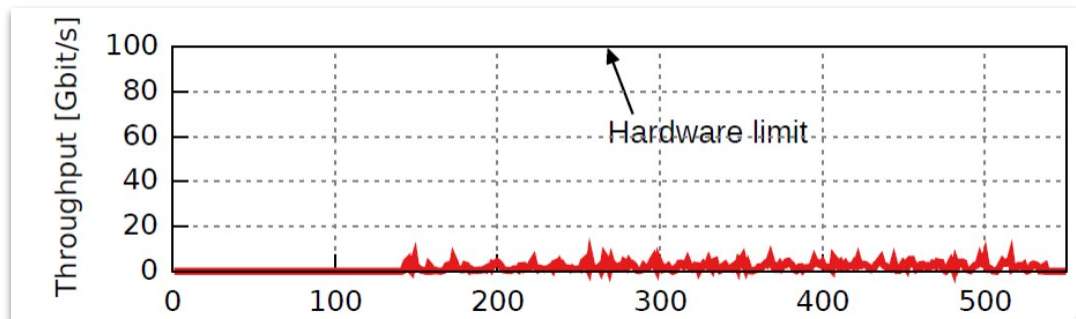
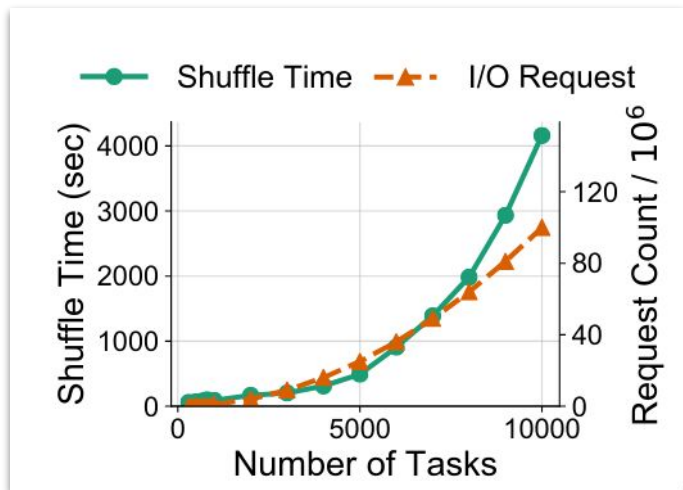
4. Saving the model



Between the initial dataset read, and the final dataset saved - there are many in-flight data objects which are temporary and ephemeral datasets

Challenges with Temporary Data Storage

1. Temporary data is performance critical - new network (100 Gbps) and storage (NVMe) can help

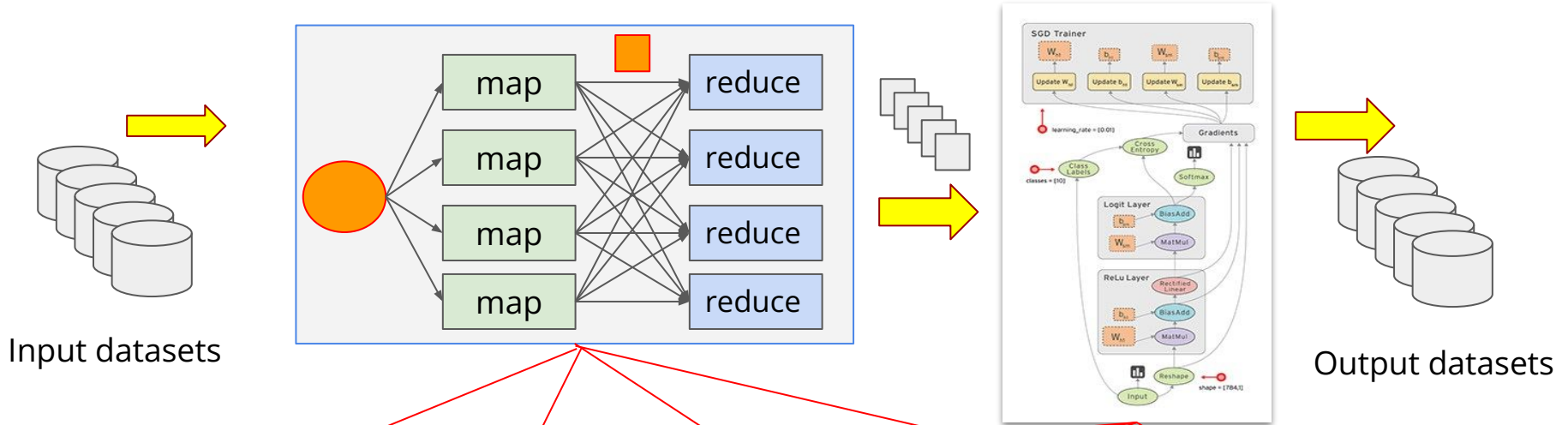


Zhang et al., Riffle: optimized shuffle service for large-scale data analytics. In EuroSys 2018
Ousterhout et al., Making sense of performance in data analytics frameworks. NSDI 2015.
Trivedi et al., On the [ir]relevance of network performance for data processing. HotCloud 2016.

Challenges with Temporary Data Storage

1. Temporary data is performance critical - new network (100 Gbps) and storage (NVMe) can help
2. Temporary data have different needs
 - a. No need to persist and provide fault tolerance
 - b. Fault tolerance is often baked in compute framework used - Spark or TensorFlow
3. Complex integration into the compute framework
 - a. Spark, TensorFlow, GraphLab, PyTorch -- all have their own way of processing data (RPCs)
 - b. New technologies are here - NAND Flash, Optane storage, PMEM, and mix of these
 - c. New deployment models: DAS vs Disaggregated
 - d. Programmable storage?

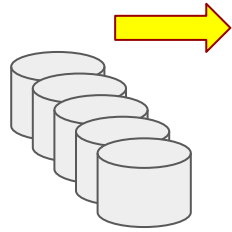
Temporary Data Management Spaghetti



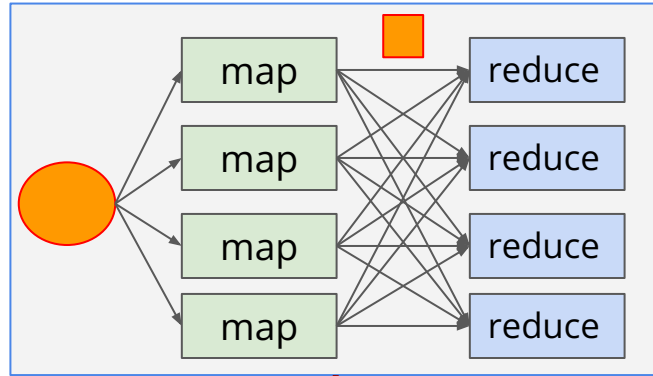
NVM Express™ over Fabrics



Temporary Data Management Spaghetti



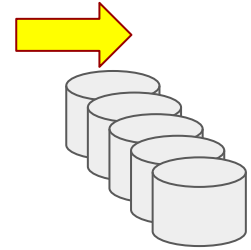
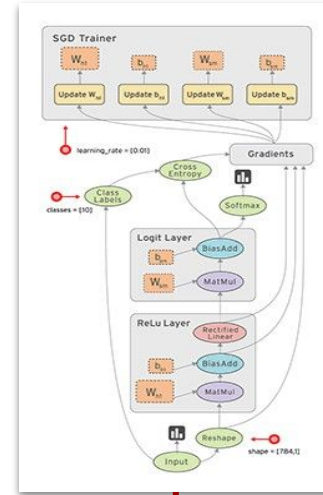
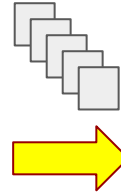
Input datasets



Build a temporary data storage framework

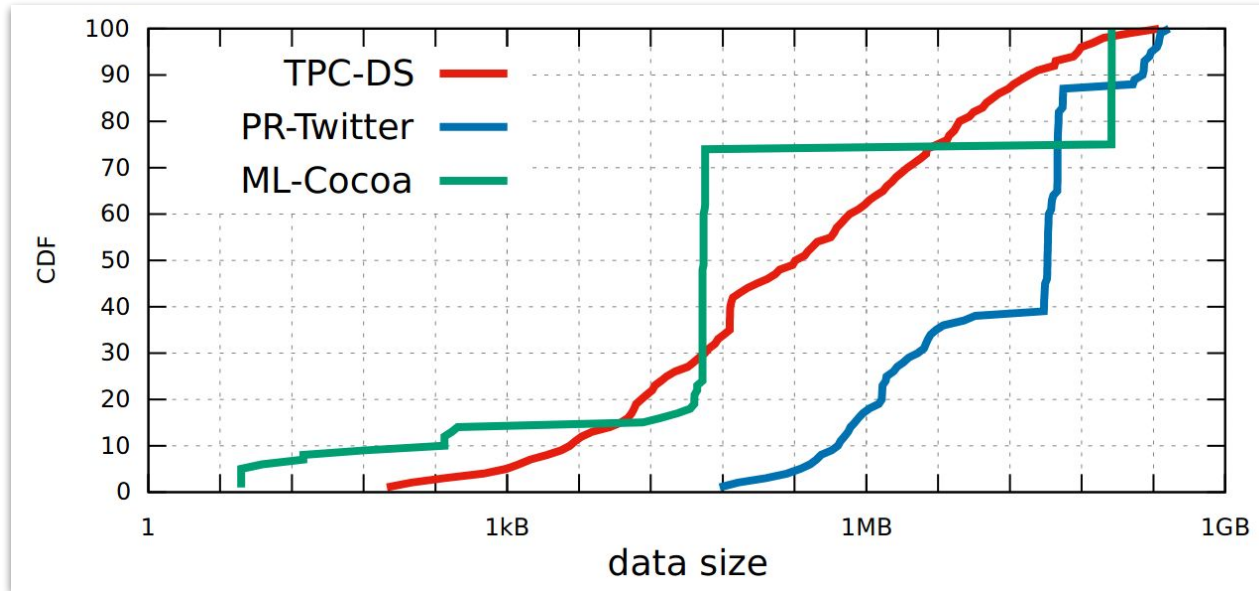


NVM Express™ over Fabrics



Output datasets

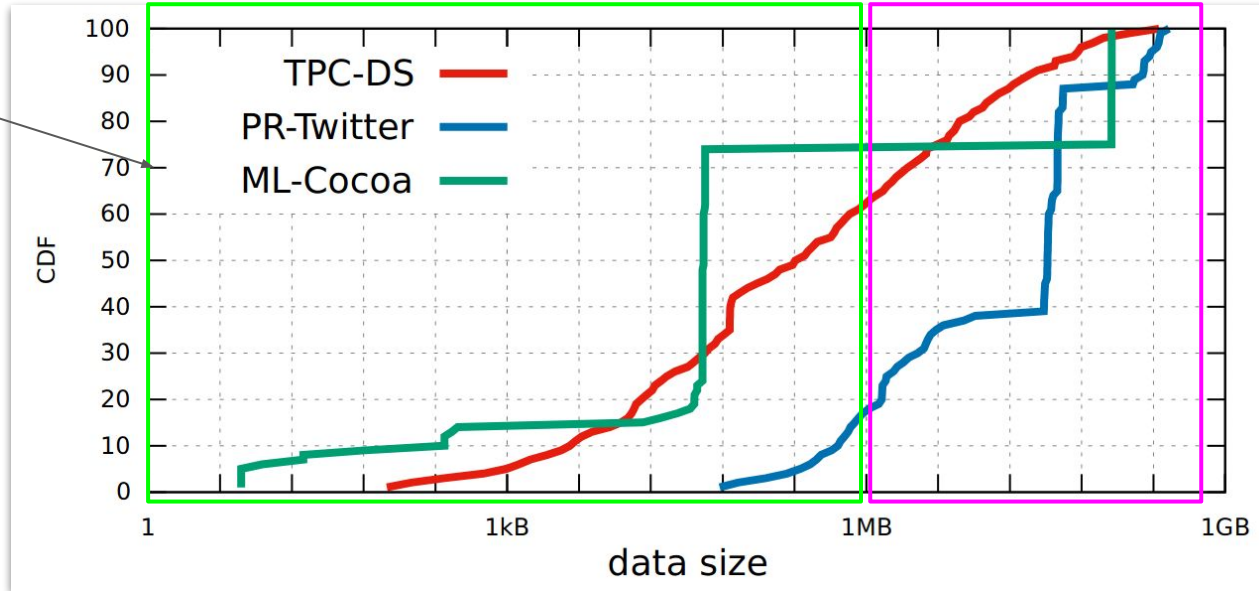
Can Existing Solutions Work?



Temporary data size distribution for three workloads (i) analytics; (ii) graph processing; (iii) ML

Can Existing Solutions Work?

Typically small values can be stored in KV Stores (latency driven)

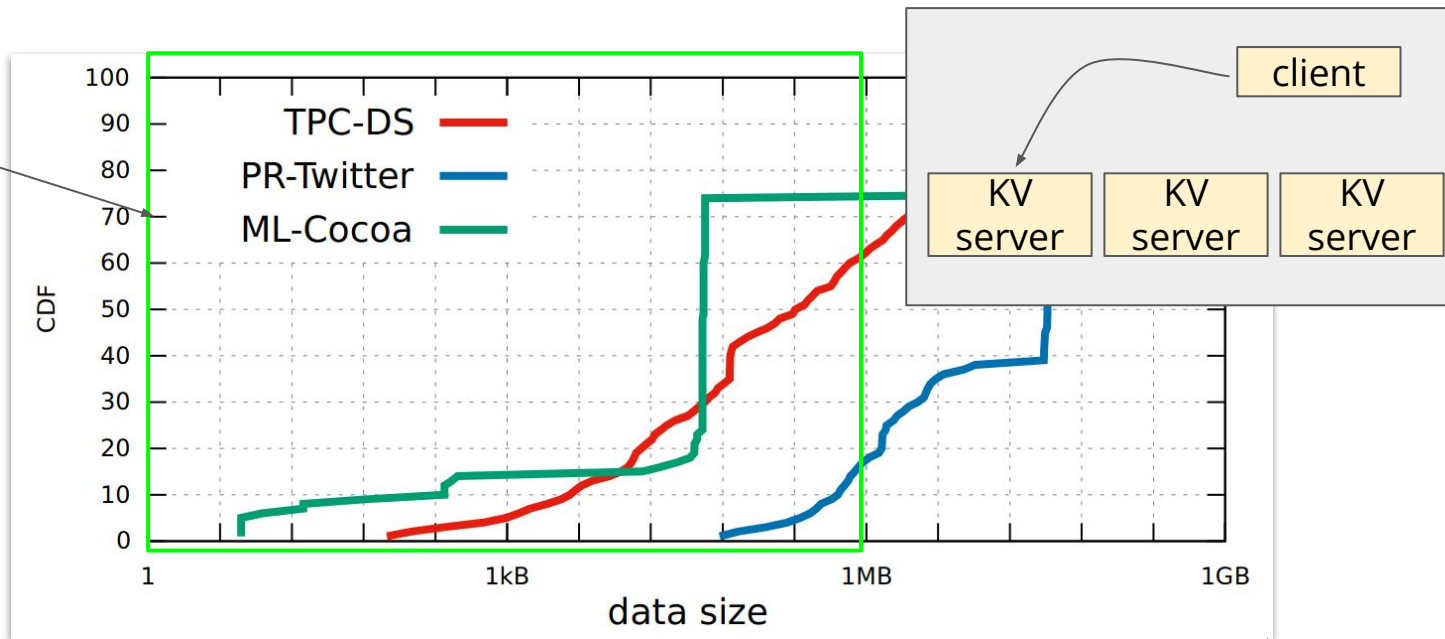


Typically large values are stored in file systems (bandwidth driven)

Temporary data size distribution for three workloads (i) analytics; (ii) graph processing; (iii) ML

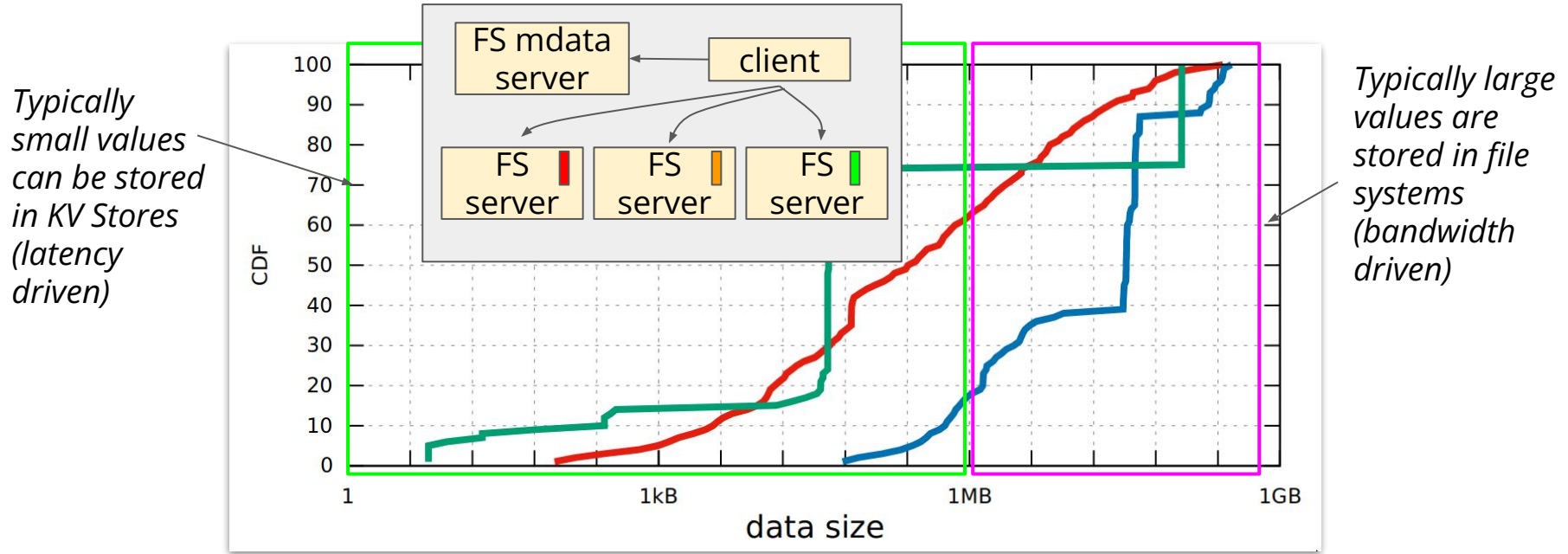
Can Existing Solutions Work?

Typically small values can be stored in KV Stores (latency driven)



Temporary data size distribution for three workloads (i) analytics; (ii) graph processing; (iii) ML

Can Existing Solutions Work?



Temporary data size distribution for three workloads (i) analytics; (ii) graph processing; (iii) ML

Unification of Temporary Storage in the NodeKernel Architecture (2019)

Unification of Temporary Storage in the NodeKernel Architecture

Patrick Stuedi[†] Animesh Trivedi[‡] Jonas Pfeifferle[†] Ana Klimovic[§]

Adrian Schuepbach[†] Bernard Metzler[†]

[†]IBM Research [‡]Vrije Universiteit [§]Stanford University

Abstract

Efficiently exchanging temporary data between tasks is critical to the end-to-end performance of many data processing frameworks and applications. Unfortunately, the diverse nature of temporary data creates storage demands that often fall between the sweet spots of traditional storage platforms, such as file systems or key-value stores.

We present NodeKernel, a novel distributed storage architecture that offers a convenient new point in the design space by fusing file system and key-value semantics in a common storage kernel while leveraging modern networking and storage hardware to achieve high performance and cost-efficiency. NodeKernel provides hierarchical naming, high scalability, and close to bare-metal performance for a wide range of data sizes and access patterns that are characteristic of temporary data. We show that storing temporary data in Crail, our concrete implementation of the NodeKernel architecture which uses RDMA networking with tiered DRAM/NVMe-Flash storage, improves NoSQL workload performance by up to 4.8× and Spark application performance by up to 3.4×. Furthermore, by storing data across NVMe Flash and DRAM storage tiers, Crail reduces storage cost by up to 8× compared to DRAM-only storage systems.

1 Introduction

Managing temporary data efficiently is key to the performance of cluster computing workloads. For example, application frameworks often cache input data or share intermediate data, both both within a job (e.g., shuffle data in a map-reduce job) and between jobs (e.g., pre-processed images in a machine learning training workflow). Temporary data storage is also increasingly important in serverless computing for exchanging

may consist of a large number of files which are organized hierarchically, vary widely in size, are written randomly, and read sequentially. While file systems (e.g., HDFS) offer a convenient hierarchical namespace and efficiently store large datasets for sequential access, distributed key-value stores are optimized for scalable access to a large number of small objects. Similarly, DRAM-based key-value stores (e.g., Redis) offer the required low latency, but persistent storage platforms (e.g. S3) are more suitable for high capacity at low cost. Overall, we find that existing storage platforms are not able to satisfy all the diverse requirements for temporary data storage and sharing in distributed data processing workloads.

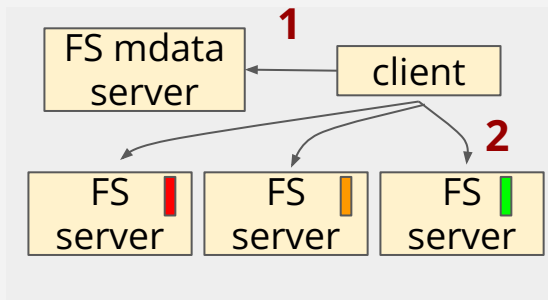
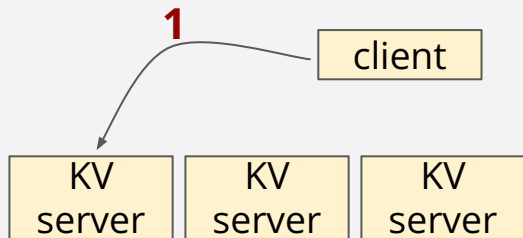
In this paper we present *NodeKernel*, a new distributed storage architecture designed from the ground up to support fast and efficient storage of temporary data. As its most distinguishing property, the NodeKernel architecture fuses file system and key-value semantics while leveraging modern networking and storage hardware to achieve high performance. NodeKernel is based on two key observations. First, many features offered by long-term storage platforms, such as durability and fault-tolerance, are not critical when storing temporary data. We observe that under such circumstances, the software architectures of file systems and key-value stores begin to look surprisingly similar. The fundamental difference is that file systems require an extra level of indirection to map offsets in file streams to distributed storage resources, while key-value stores map entire key-value pairs to storage resources. The second observation is that low-latency networking hardware and multi-CPU many-core servers dramatically reduce the cost of this indirection in a distributed setting by enabling scalable RPC communication at latencies of a few microseconds.

Based on these insights we develop the NodeKernel architecture by implementing file system and key-value semantics

The NodeKernel Architecture (2019)

A fused KV + File system distributed storage designed for temporary data storage, basic ideas

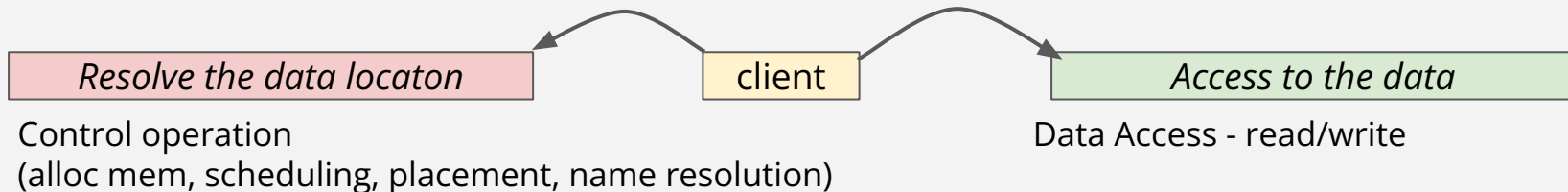
1. With fast network - FS and KV semantics can be provided in a single system



The NodeKernel Architecture (2019)

A fused KV + File system distributed storage designed for temporary data storage, basic ideas

1. With fast network - FS and KV semantics can be provided in a single system
 - a. Key Value Store = contact a single server + data transfer
 - b. File Systems = contact metadata server + data servers + data transfer
 - c. Nodes can be specialized : Tables, Directories, Files, workload specific files, Append-only, etc.
2. Split control and data planes



The NodeKernel Architecture (2019)

A fused KV + File system distributed storage designed for temporary data storage, basic ideas

- **Data Plane** - code path or calls where the actual work is done
 - Data r/w, make it straight forward, no blocking calls, everything is ready to go
- **Control Plane** - code path or call where resources are managed
 - Slow(er), resourced need to be allocated and managed, can block
- **Fast path** - common case execution (typically few branches, decision making, very simple code)
 - Read a file from start to finish, all blocks arrive in order
- **Slow path** - more sanity checks (more branches, hence poor(er) performance)
 - Read a file in fragments with random accesses in between, error handling

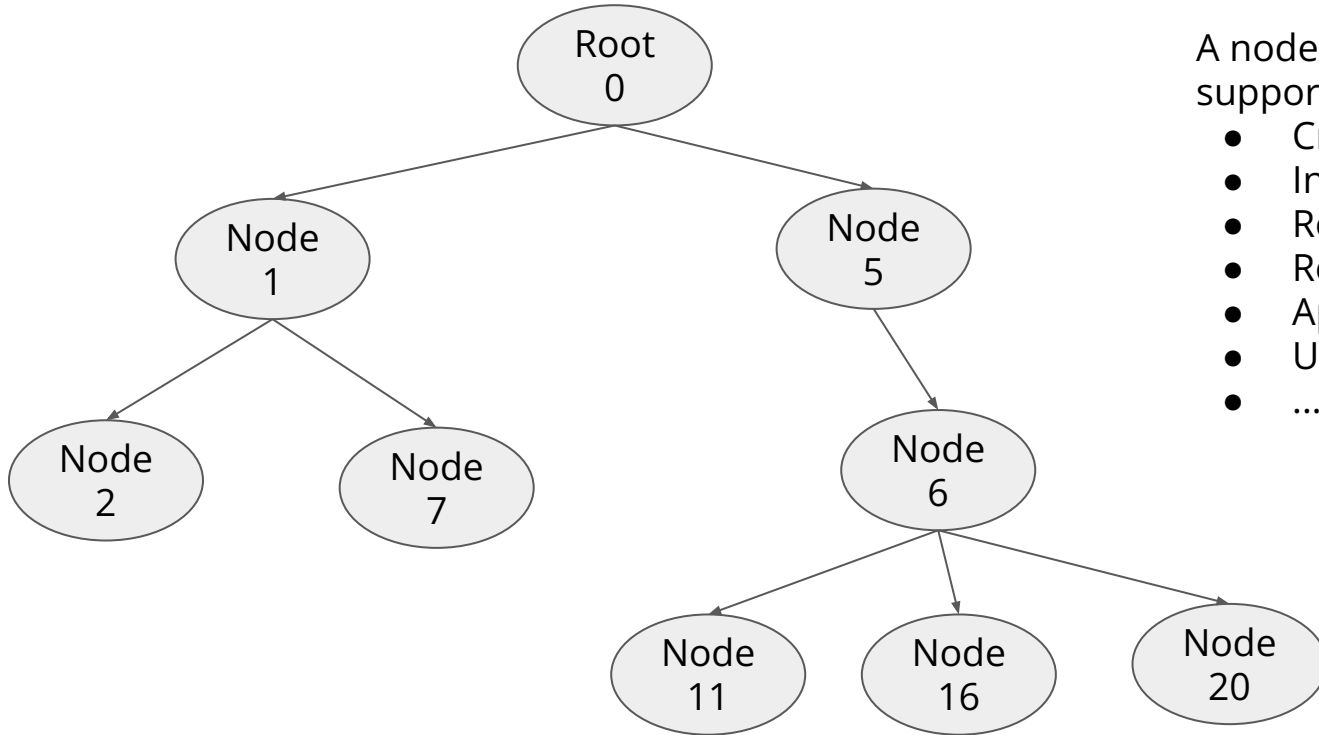
The NodeKernel Architecture (2019)

A fused KV + File system distributed storage designed for temporary data storage, basic ideas

1. With fast network - FS and KV semantics can be provided in a single system
 - a. Key Value Store = contact a single server + data transfer
 - b. File Systems = contact metadata server + data servers + data transfer
 - c. Nodes can be specialized : Tables, Directories, Files, workload specific files, Append-only, etc.
2. Split control and data planes
 - a. Control plane = fast asynchronous RPCs
 - b. Data plane = One-sided RDMA operations and NVMeF for I/O from DRAM and Flash storage

Trick: prepare and allocate all resources (carefully manage the NVM runtime) and do not intervene in offloaded I/O access operations

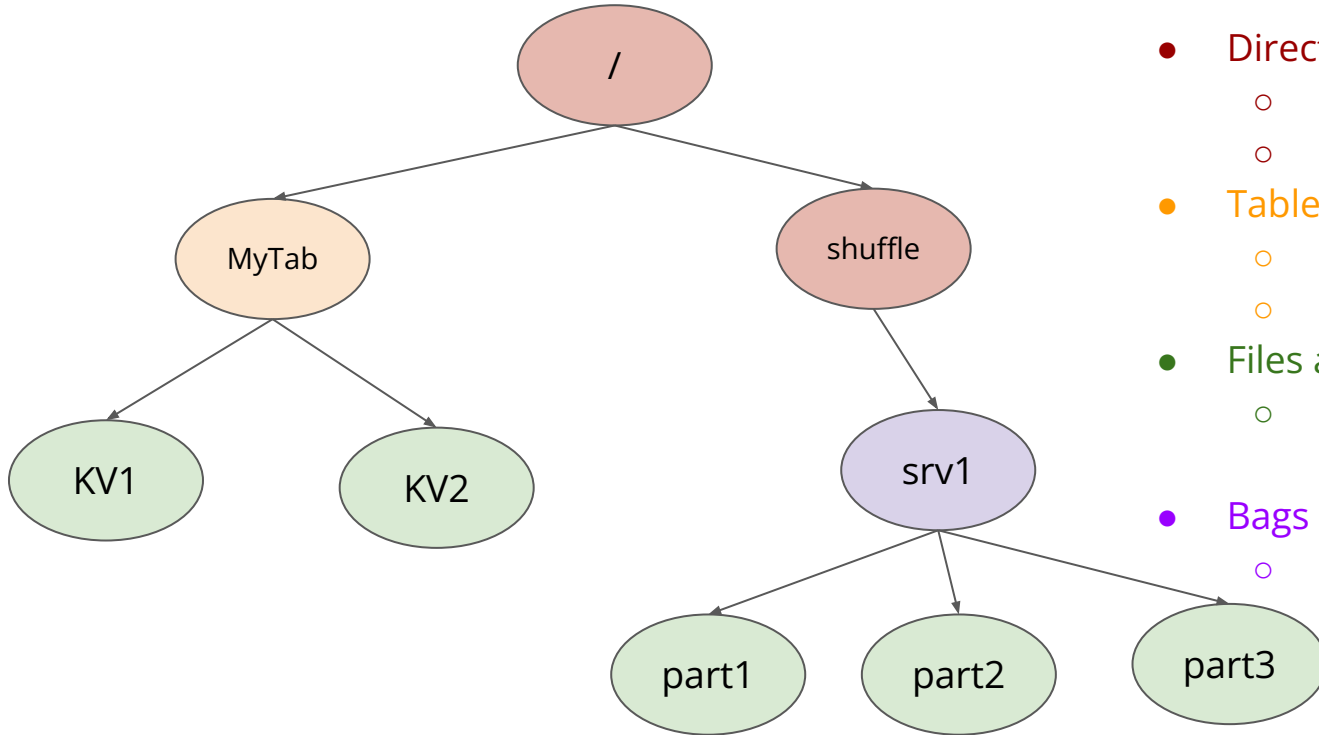
NodeKernel: A High-Level Idea



A node is an abstract type that supports

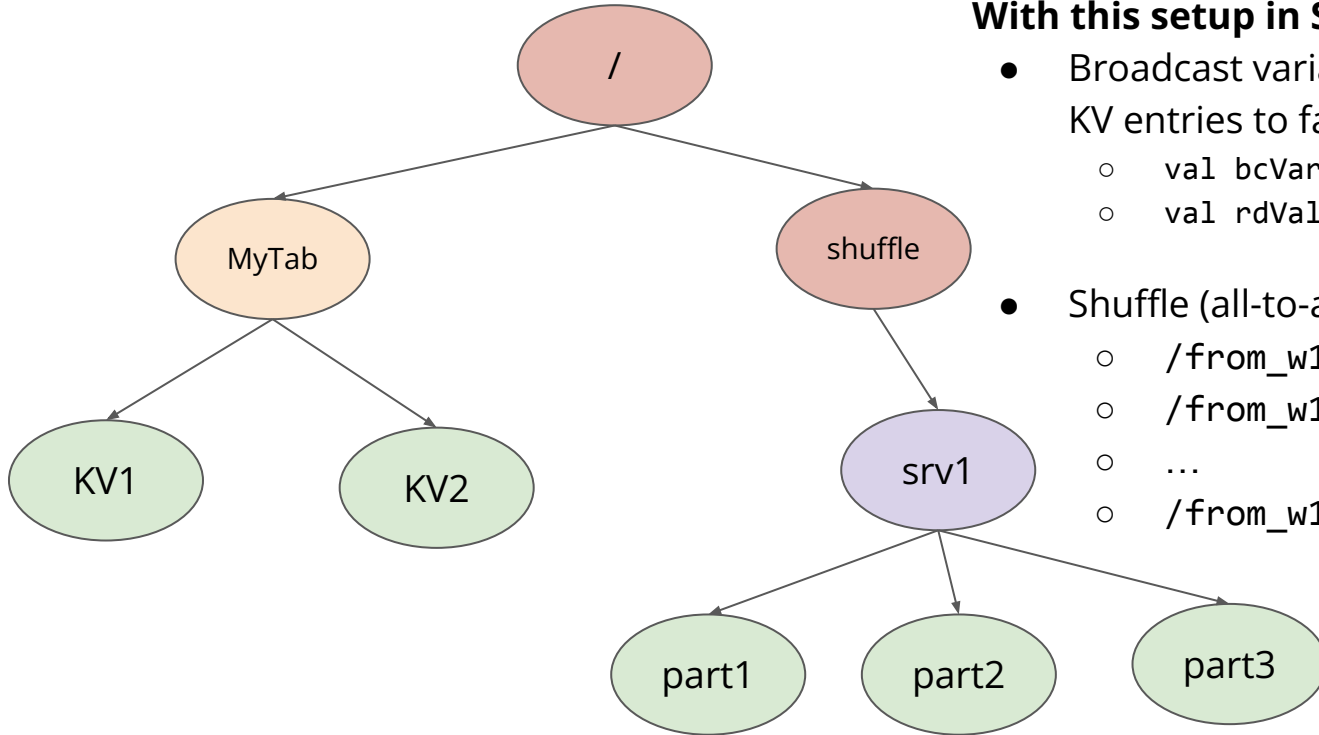
- Creating a node
- Inserting into the tree
- Removing from the tree
- Read
- Append
- Update
- ...

NodeKernel: A High-Level Idea



- Directory
 - Enumerate
 - Add/remove files
- Tables
 - Collection of KVs
 - Add, remove KV files
- Files and KV files
 - Last winner vs error on concurrent creator
- Bags of directories
 - Fast data reading over multiple directories and files

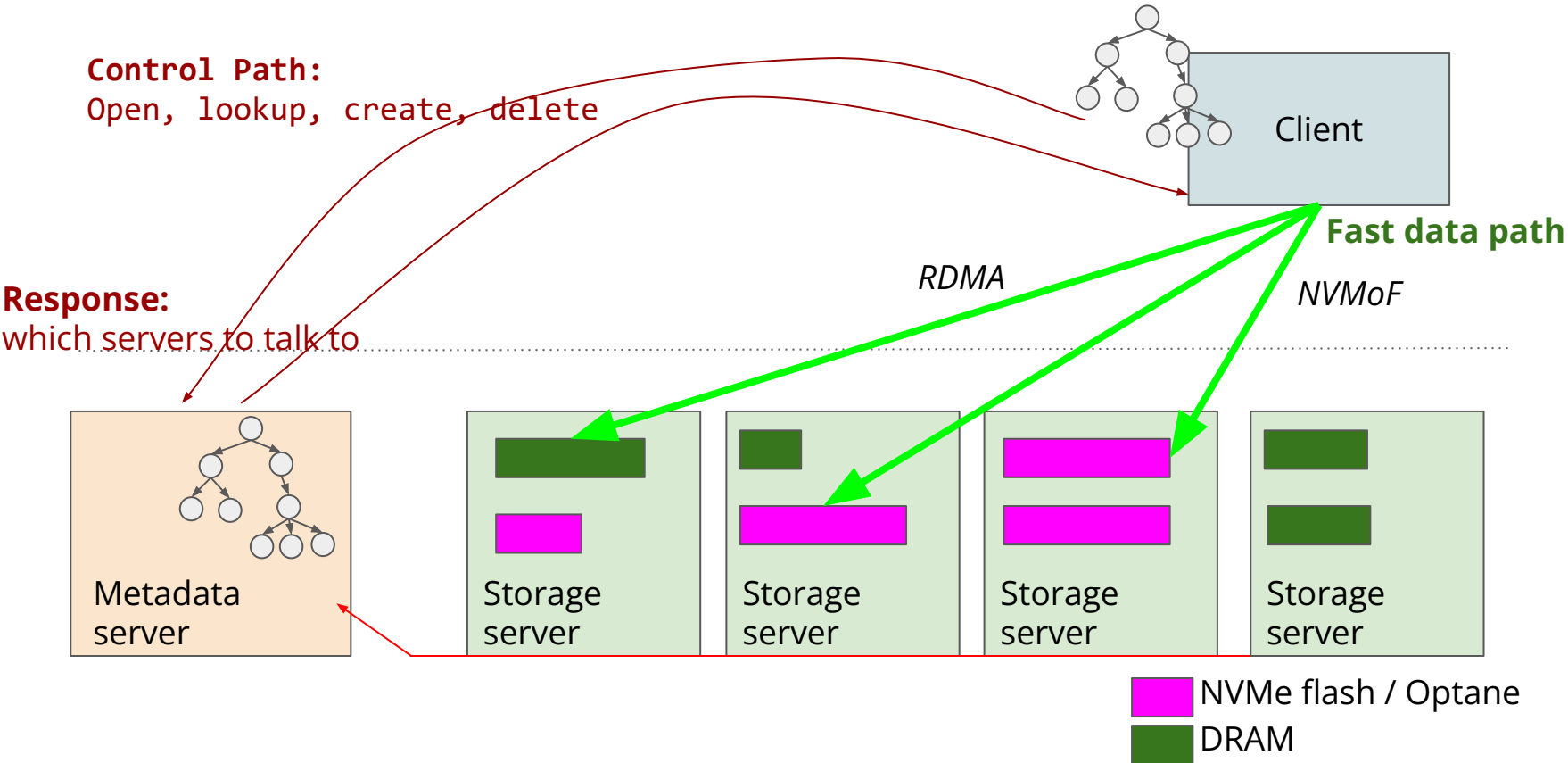
NodeKernel: A High-Level Idea



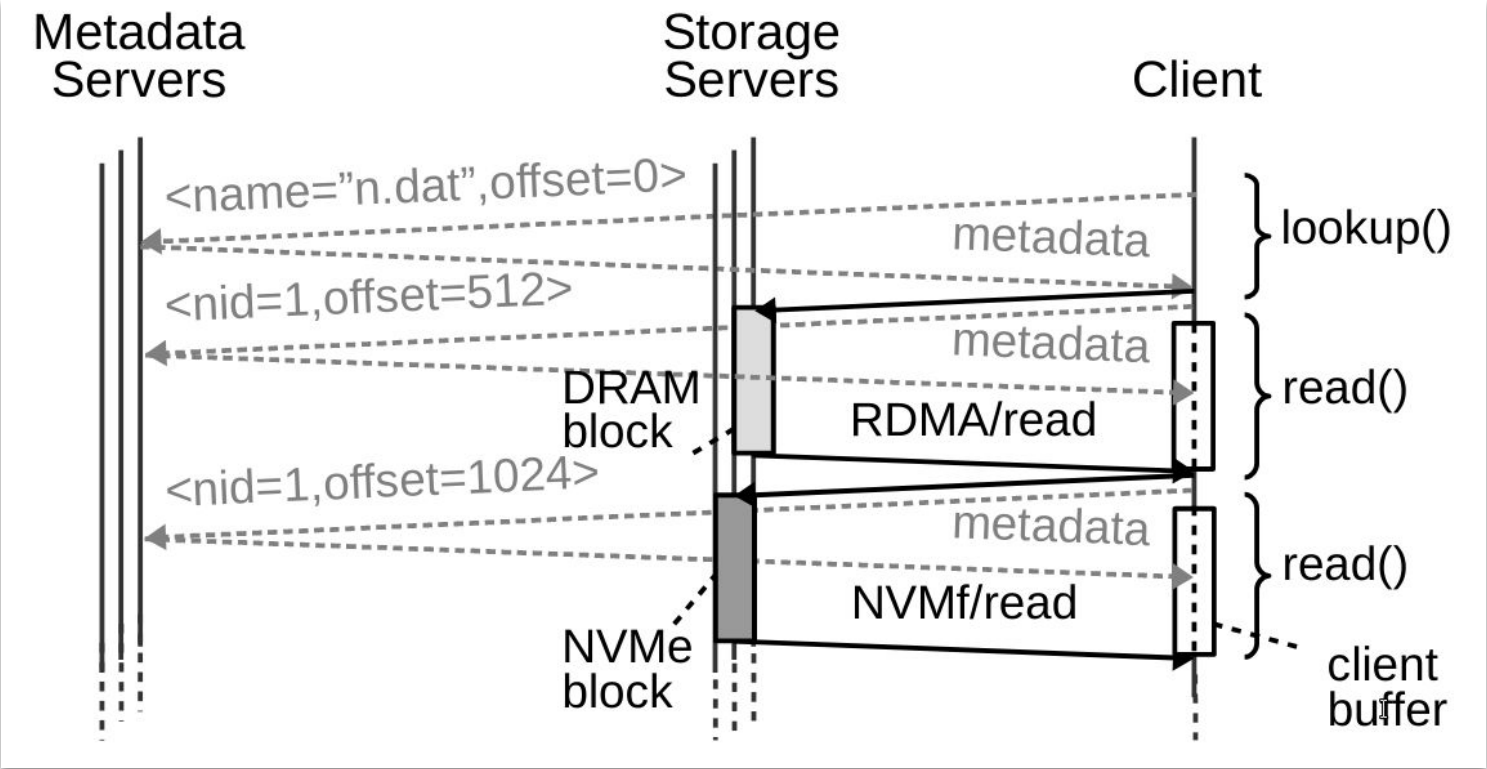
With this setup in Spark

- Broadcast variables can be stored as a fast KV entries to fast lookups
 - `val bcVar = sc.broadcast("10") //put`
 - `val rdVal = bcVar.value.get() //get`
- Shuffle (all-to-all) can be (path enumeration)
 - `/from_w1/to_w2/file1`
 - `/from_w1/to_w3/file1`
 - ...
 - `/from_w10/to_w1/file`

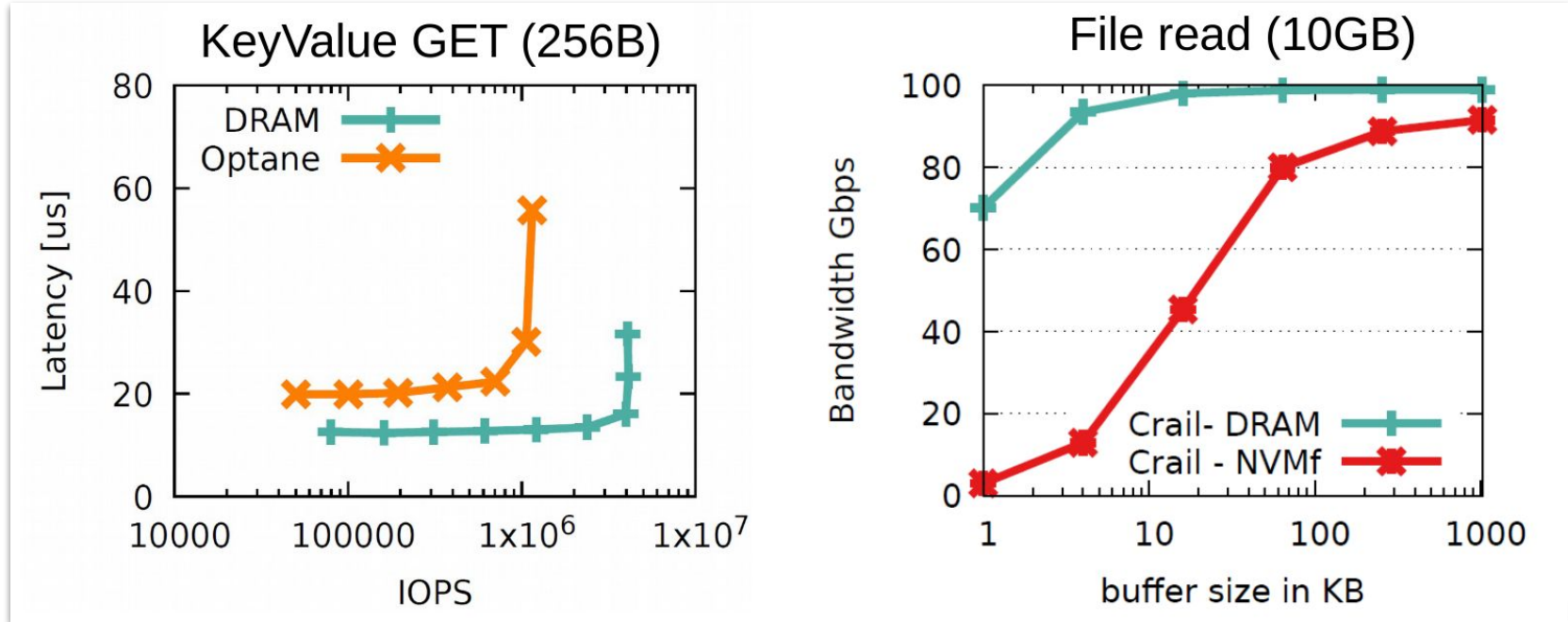
Implementation in Apache Crail



Heavily Pipelined Architecture



Performance

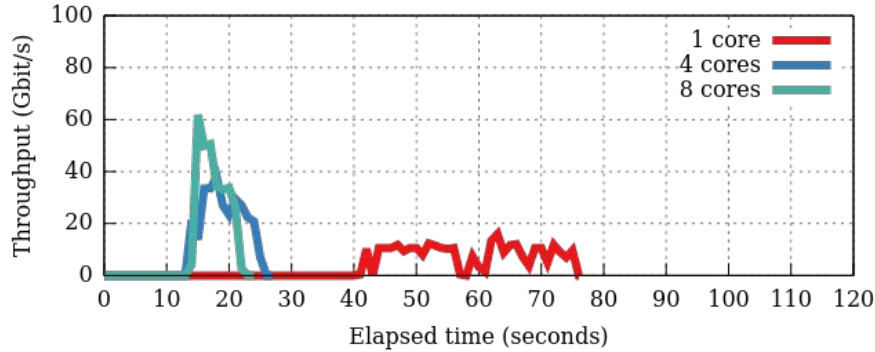


Small data sets (in KV mode): low latency with high IOPS

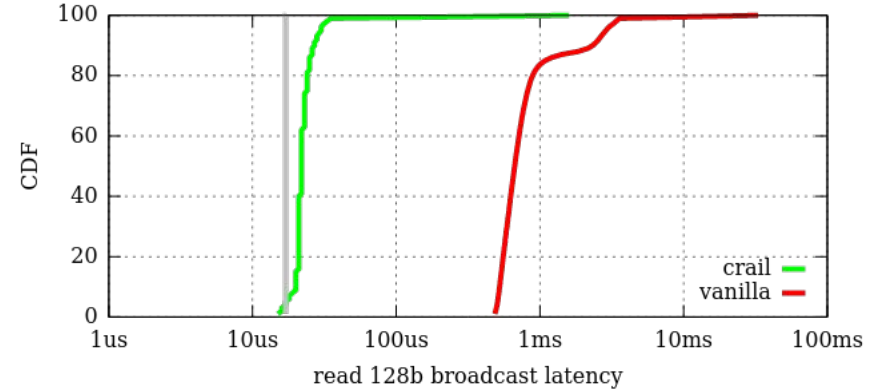
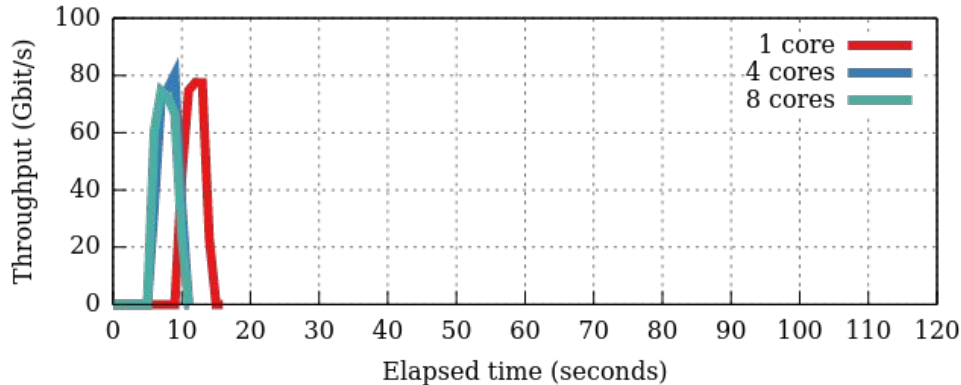
Large data sets (~10s GB, in FS mode): deliver high bandwidth

Integration with Spark: Shuffle and Broadcast

Groupby Vanilla Spark



Groupby Spark/Crail

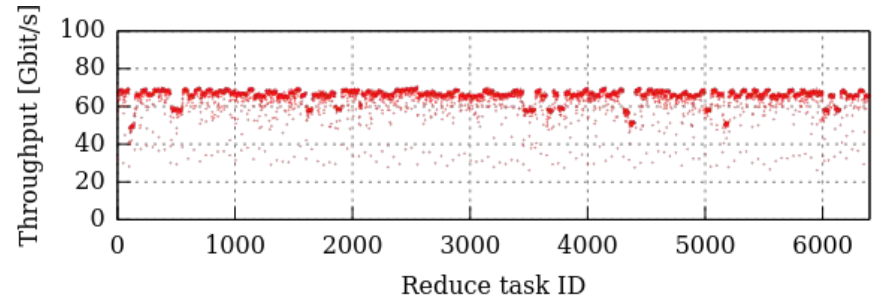
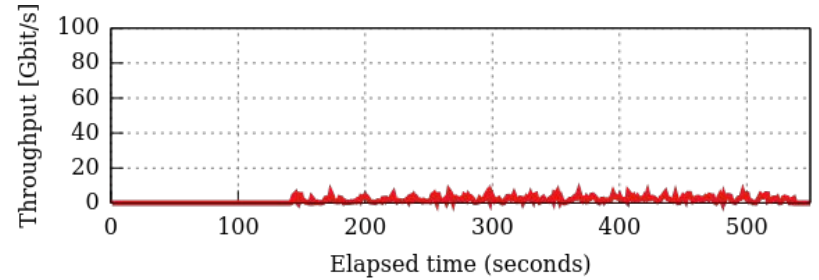
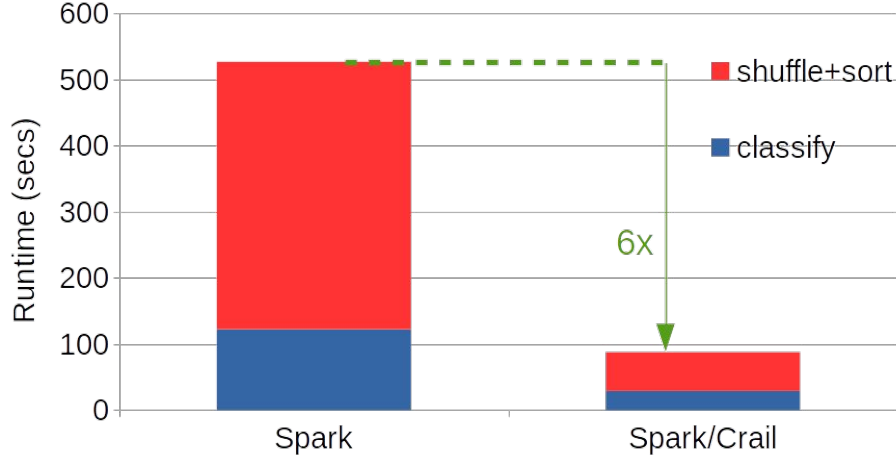


- 2-5x performance improvement in shuffle
- More than 10x gains for broadcast

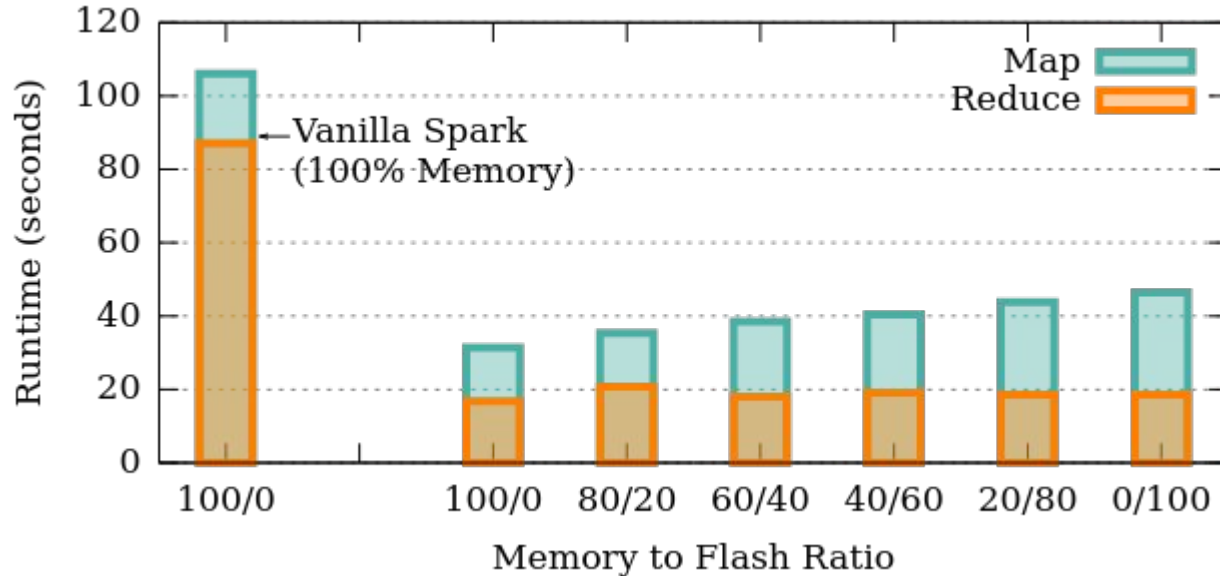
<https://crailabs.github.io/blog/2017/08/crail-memory.html>

Putting Everything Together: TeraSort

- 128 nodes x 100 Gbps RoCE cluster, total dataset 12.8 TB



Storage Disaggregation

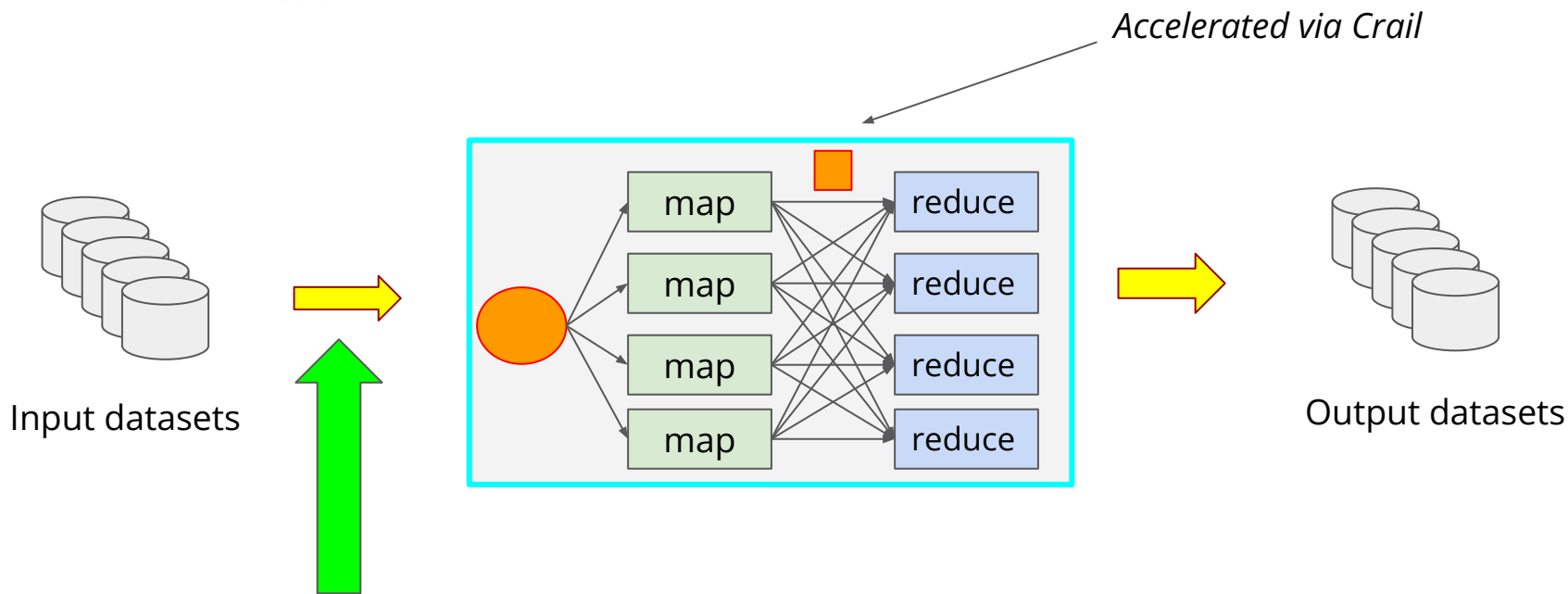


Storage disaggregation and moving all data from memory to flash for storage

- Flash is cheaper, more energy efficient, and denser than the DRAM

The whole in-flash shuffle storage is still faster than vanilla Spark in DRAM

So Far ...



***We have seen that we can shuffle data very close to the hardware limits.
Can we actually feed data at that speed too?***

Relational Data Processing Stacks in the Cloud

Relational
Engines



One of the most popular data processing paradigms

- Data organized in tables
- Analyzed using DSL like SQL
- Integrity protected using variants

But unlike classical RDBMs systems, they don't manage their own storage

Relational Data Processing Stack in the Cloud

Relational
Engines



File
Formats



Distributed
Storage



Back to the Future - It is 2010

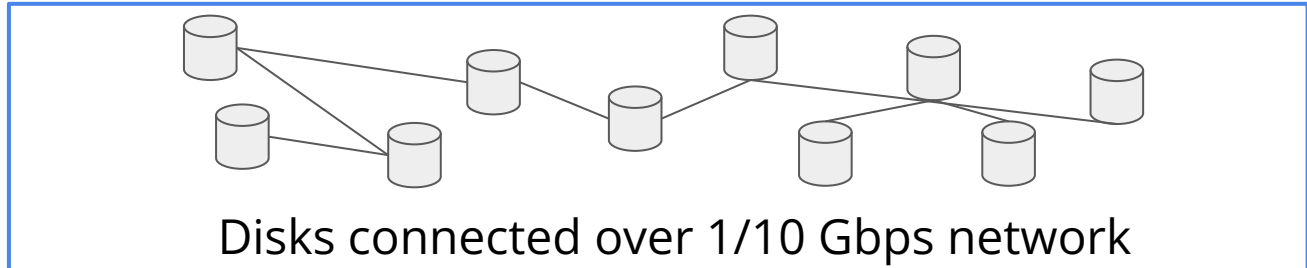
Relational
Engines



File
Formats



Hardware



Back to the Future - It is 2010

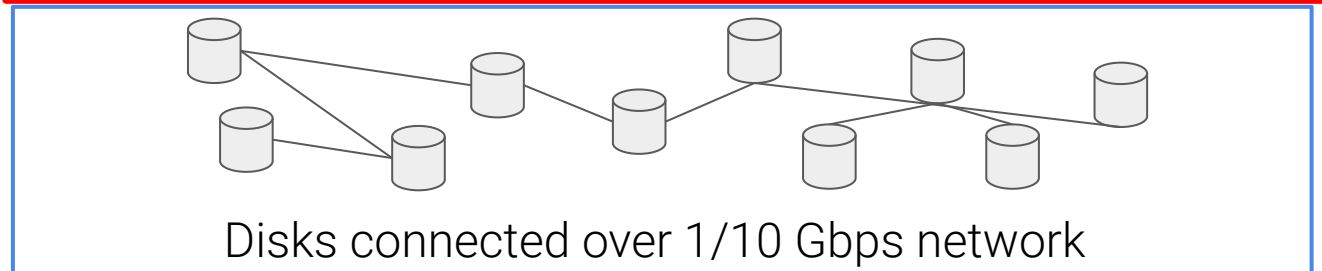
Relational
Engines



File
Formats



Hardware



Albis File Format (2018)

Albis: High-Performance File Format for Big Data Systems

Animesh Trivedi, Patrick Stuedi, Jonas Pfefferle, Adrian Schuepbach, and Bernard Metzler
IBM Research, Zurich

Abstract

Over the last decade, a variety of external file formats such as Parquet, ORC, Arrow, etc., have been developed to store large volumes of relational data in the cloud. As high-performance networking and storage devices are used pervasively to process this data in frameworks like Spark and Hadoop, we observe that none of the popular file formats are capable of delivering data access rates close to the hardware. Our analysis suggests that multiple antiquated notions about the nature of I/O in a distributed setting, and the preference for the “storage efficiency” over performance is the key reason for this gap.

In this paper we present Albis, a high-performance file format for storing relational data on modern hardware. Albis is built upon two key principles: (i) reduce the CPU cost by keeping the data/metadata storage format simple; (ii) use a binary API for an efficient object management to avoid unnecessary object materialization. In our evaluation, we demonstrate that in micro-benchmarks Albis delivers $1.9 - 21.4\times$ faster bandwidths than other formats. At the workload-level, Albis in Spark/SQL reduces the runtimes of TPC-DS queries up to a margin of $3\times$.

1 Introduction

Relational data management and analysis is one of the most popular data processing paradigms. Over the last decade, many distributed relational data *processing* systems (RDPS) have been proposed [15, 53, 29, 38, 35, 24]. These systems routinely process vast quantities of (semi-)structured relational data to generate valuable insights [33]. As the volume and velocity of the data increase, these systems are under constant pressure to deliver ever higher performance. One key factor that determines the performance is the data access rate. However, unlike the classic relational database management systems (RDBMS) which are jointly designed for optimal data storage and processing, modern cloud-based

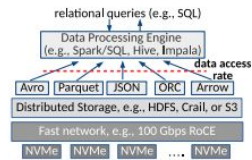


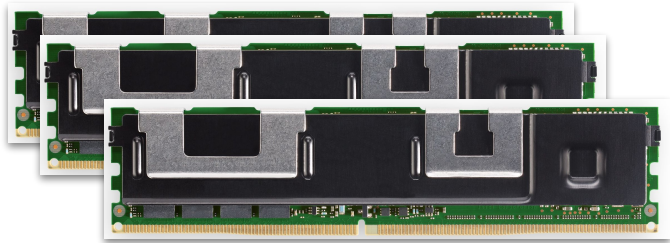
Figure 1: Relational data processing stack in the cloud.

RDPS systems typically do not manage their storage. They leverage a variety of external file formats to store and access data. Figure 1 shows a typical RDPS stack in the cloud. This modularity enables RDPS systems to access data from a variety of sources in a diverse set of deployments. Examples of these formats are Parquet [10], ORC [9], Avro [6], Arrow [5], etc. These formats are now even supported by the RDBMS solutions which add Hadoop support [49, 41, 31]. Inevitably, the performance of a file format plays an important role.

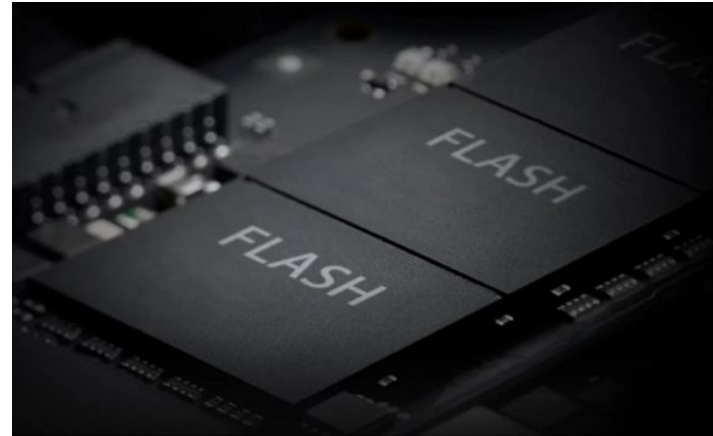
Historically, file formats have put the top priority as the “storage efficiency”, and aim to reduce the amount of I/O as much as possible because I/O operations are considered slow. However, with the recent performance advancements in storage and network devices, the fundamental notion of “a fast CPU and slow I/O devices” is now antiquated [44, 40, 54]. Consequently, many assumptions about the nature of storage in a distributed setting are in need of revision (see Table 1). Yet, file formats continue to build upon these antiquated assumptions without a systematic consideration for the performance. As a result, only a fraction of raw hardware performance is reflected in the performance of a file format.

In this paper, we re-visit the basic question of storage and file formats for modern, cloud-scale relational data processing systems. We first start by quantifying the impact of modern networking and storage hardware

The I/O Revolution



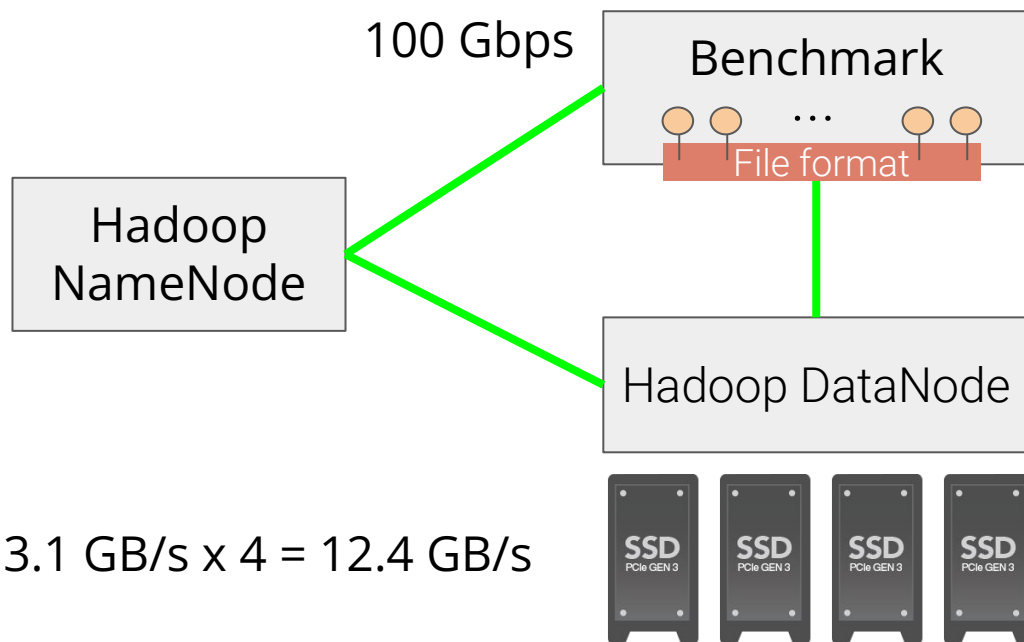
SPEED
LIMIT
100
Gbps



2-3 orders of magnitude performance improvements

- latency : from msecs to μ secs
- bandwidth : from MBps to GBps
- IOPS : from 100s to 100K

The Impact of the Revolution



Micro-benchmark*

16 cores in parallel, reading TPC-DS data set.
What is the bandwidth?

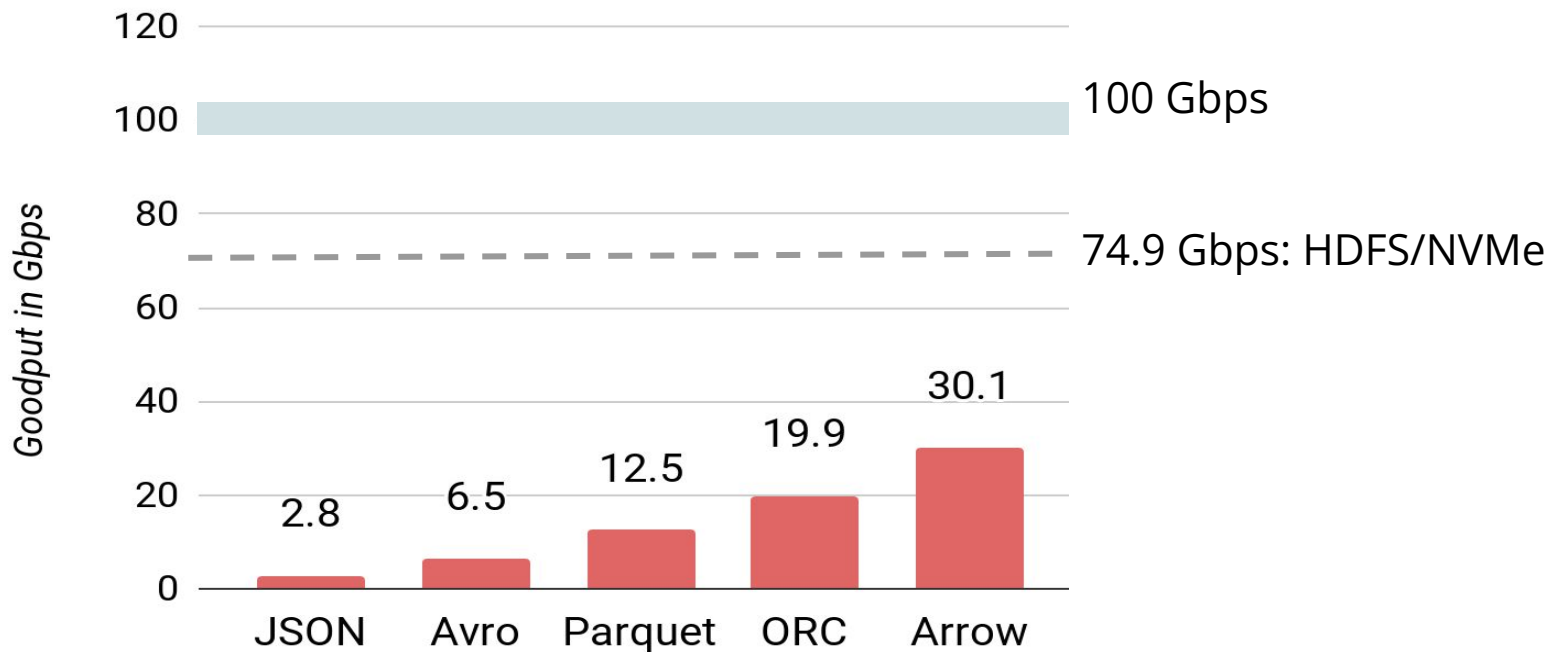
Why micro-benchmark?
Decouple from the SQL engine

*<https://github.com/animeshtrivedi/fileformat-benchmarks>

The Impact of the Revolution



The Impact of the Revolution



None of the modern file formats delivered performance close to the hardware

The Outdated Assumptions and Impact



End-host
assumptions



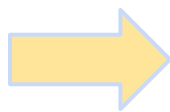
Distributed systems
assumptions



Language/runtimes
assumptions

The Outdated Assumptions and Impact

End-host
assumptions



1. *CPU is fast, I/O is slow*

- trade CPU for I/O
- compression, encoding

But why now? CPU core speed is stalled, but ...

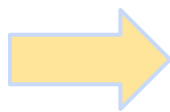
Distributed systems
assumptions

	1 Gbps	HDD	100 Gbps	Flash
Bandwidth	117 MB/s	140 MB/s	12.5 GB/s	3.1 GB/s
cycle/unit	38,400	10,957	360	495

Language/runtimes
assumptions

The Outdated Assumptions and Impact

End-host assumptions



2. Avoid slow, random small I/O

- preference for large block scans

But leads to bad CPU cache performance

Distributed systems assumptions

C0	C4
C1	C5
C2	C6
C3	C7



128 MB \equiv 1 GB cache size?

Language/runtimes assumptions

Bounded by the number of instructions/row

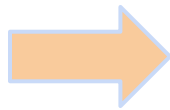


Bounded by the poor cache/IPC performance

The Outdated Assumptions and Impact

End-host
assumptions

Distributed systems
assumptions

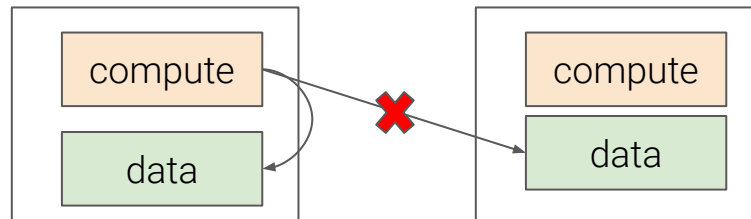


Language/runtimes
assumptions

3. Remote I/O is slow

- pack data/metadata together
- schedule tasks on local blocks

But now network/storage is super fast? then why still pack all data in a single block and try to co-schedule tasks?

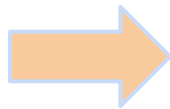


The Outdated Assumptions and Impact

End-host
assumptions

Distributed systems
assumptions

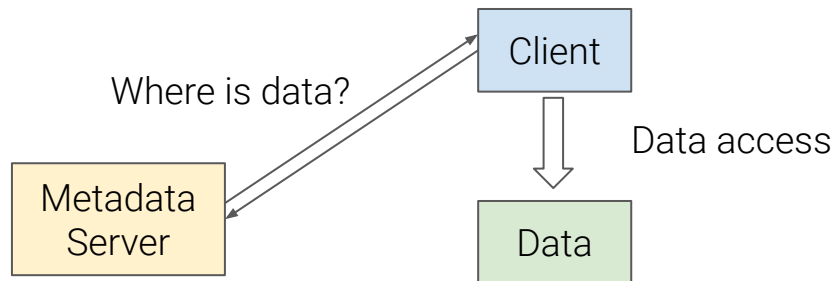
Language/runtimes
assumptions



4. Metadata lookups are slow

- decrease number of lookups by decreasing number of files/directories

RAMCloud, Crail can do 10 millions of lookups/sec. Does this design still make sense?

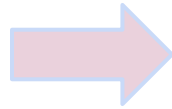


The Outdated Assumptions and Impact

End-host
assumptions

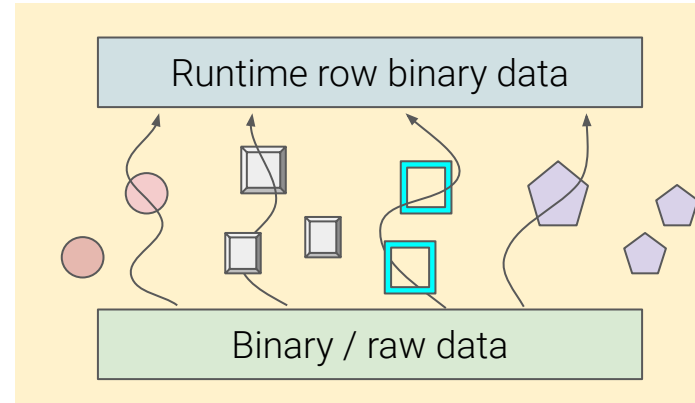
Distributed systems
assumptions

Language/runtimes
assumptions



5. *Disregard for the runtime environment:*

- group encoded/decoded
- heavy object pressure
- independent layers, no shared object
- materialize all objects



Albis

- Albis - A file format to store relational tables for read-heavy analytics workloads
- Supports all basic primitive types with data and schema
 - nested schemas are flattened and data is stored in the leaves
- Three fundamental design decisions:
 1. **avoid CPU pressure**, i.e., no encoding, compression, etc.
 2. **simple data/metadata management** on the distributed storage
 3. **carefully managed runtime** - simple row/column storage with a binary API

Table Storage Logic

Int double byte[] char float[]

00	01	02	03	04
10	11	12	13	14
20	21	22	23	24
30	31	32	33	34
40	41	42	43	44

Table Storage Logic

Int double byte[] char float[]

00	01	02	03	04
10	11	12	13	14
20	21	22	23	24
30	31	32	33	34
40	41	42	43	44

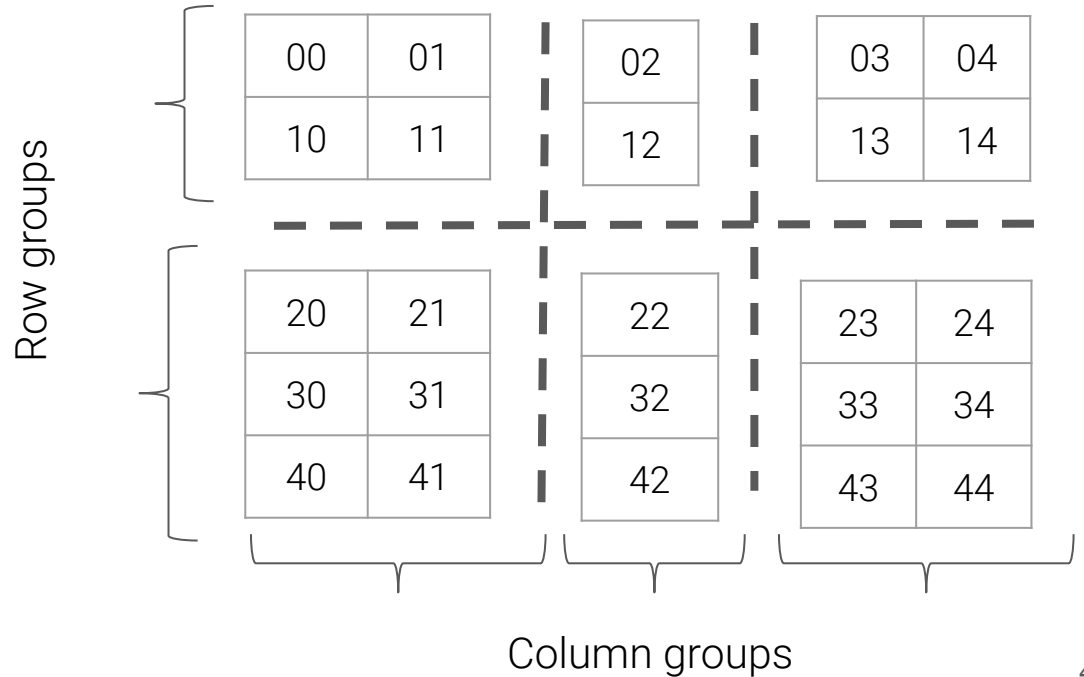


Table Storage Logic

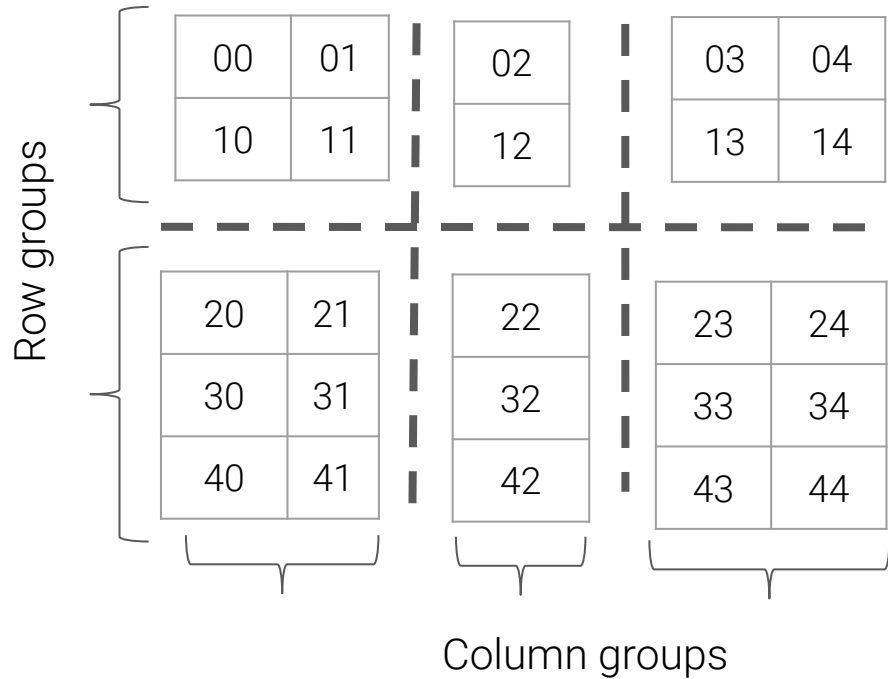
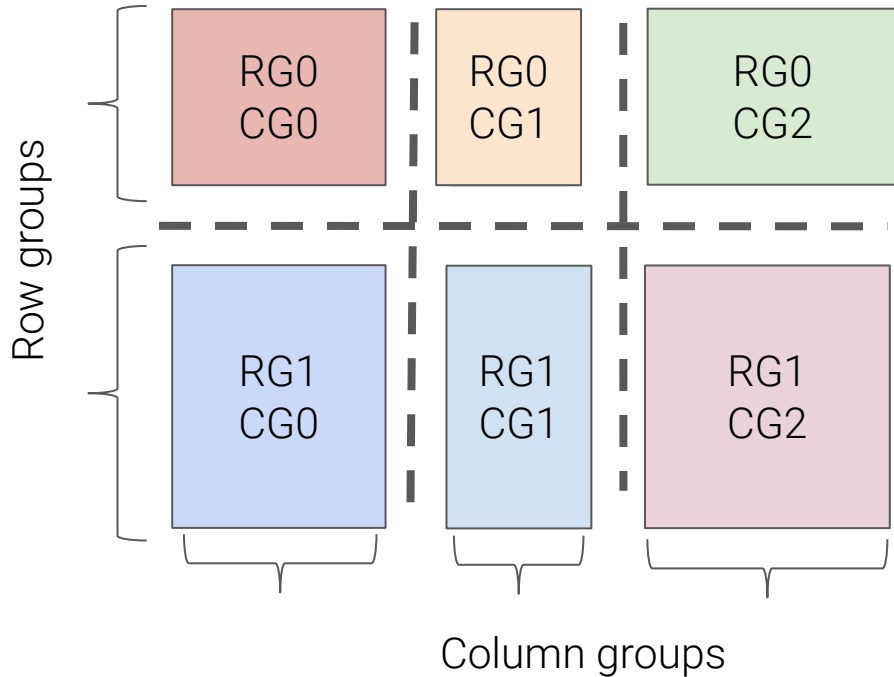
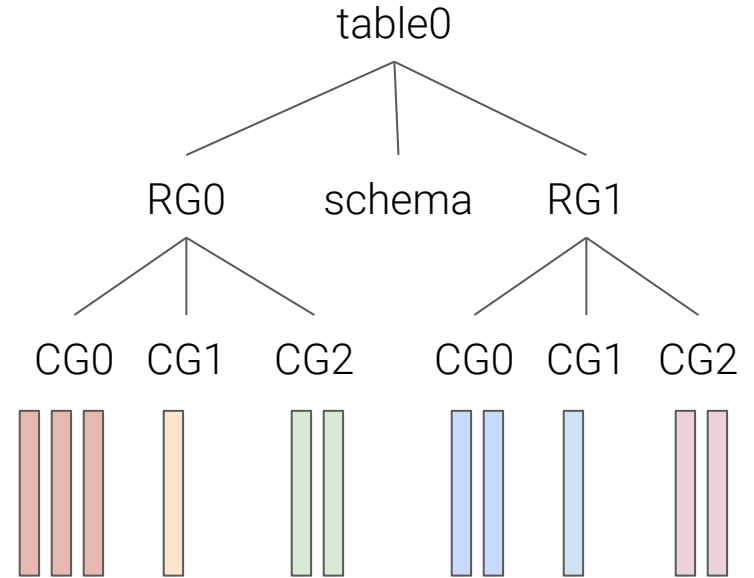
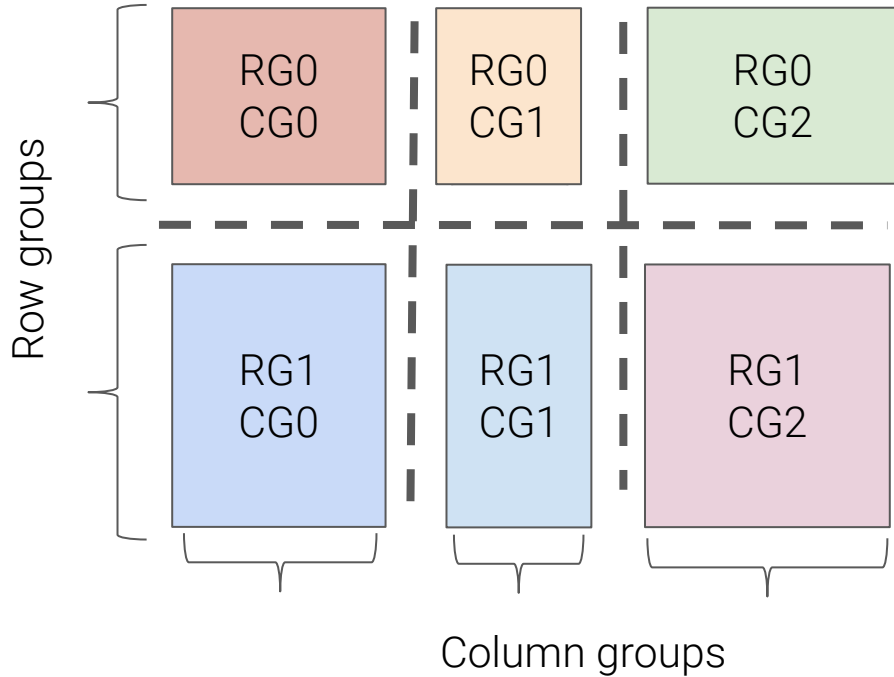


Table Storage Logic

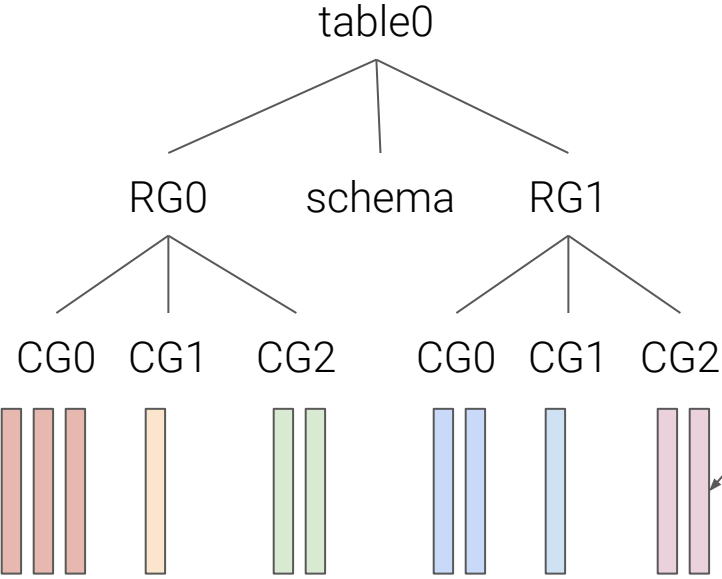


If there is only 1 column group : Row store
If there are 'n' column groups : Columns store

Table Storage Logic



Row Storage Format



How is a single row of data stored in these files?



Row Storage Format

Null bitmap



Marking null columns values

Row Storage Format

Null bitmap

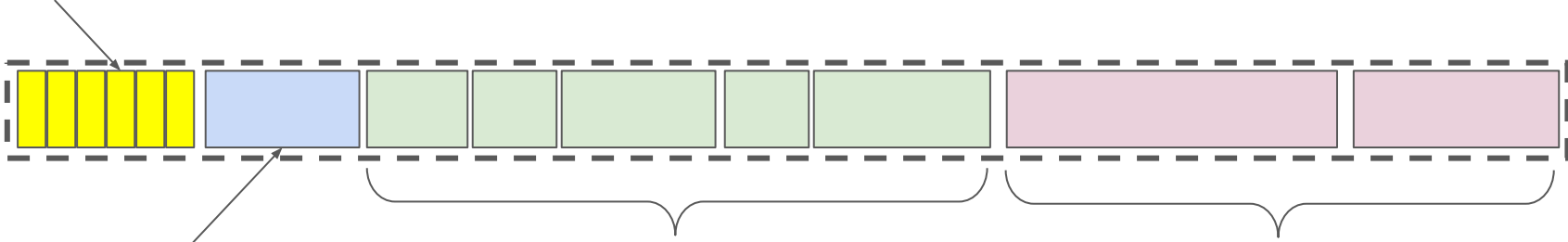


complete row size



Row Storage Format

Null bitmap

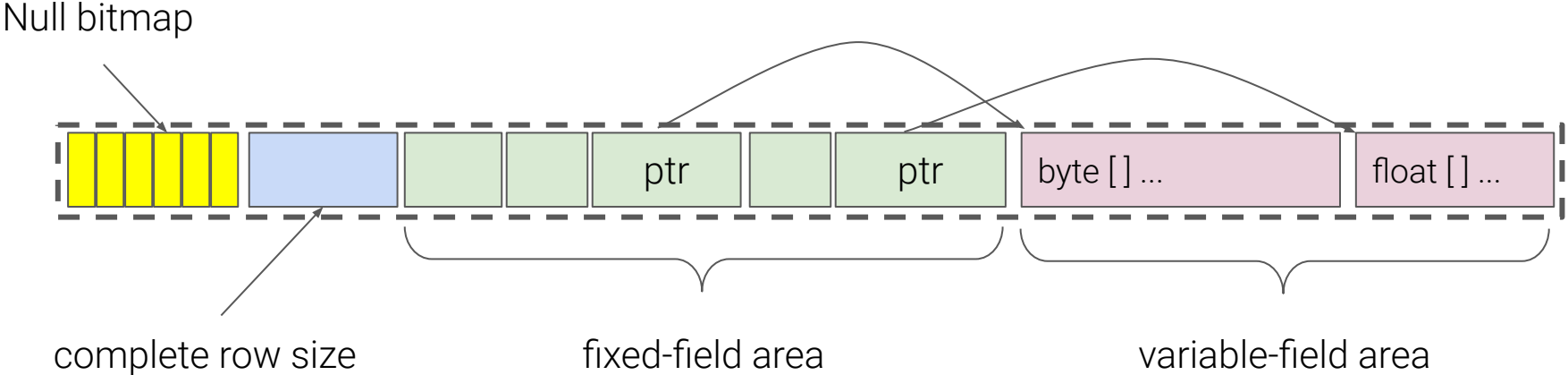


complete row size

fixed-field area

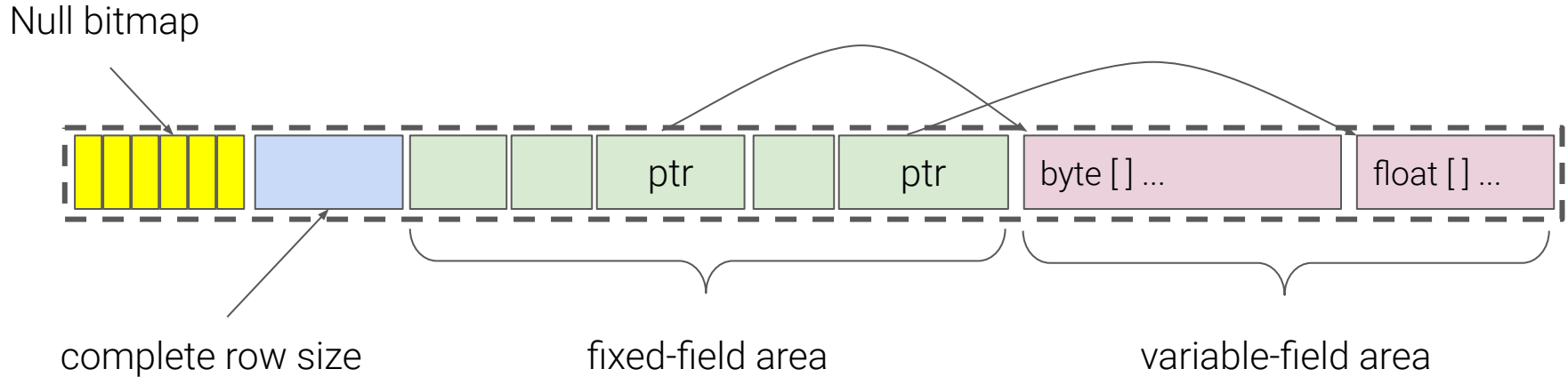
variable-field area

Row Storage Format



Schema of { int, double, byte[], char, float[] } :

Row Storage Format



Schema of { int, double, byte[], char, float[] } :

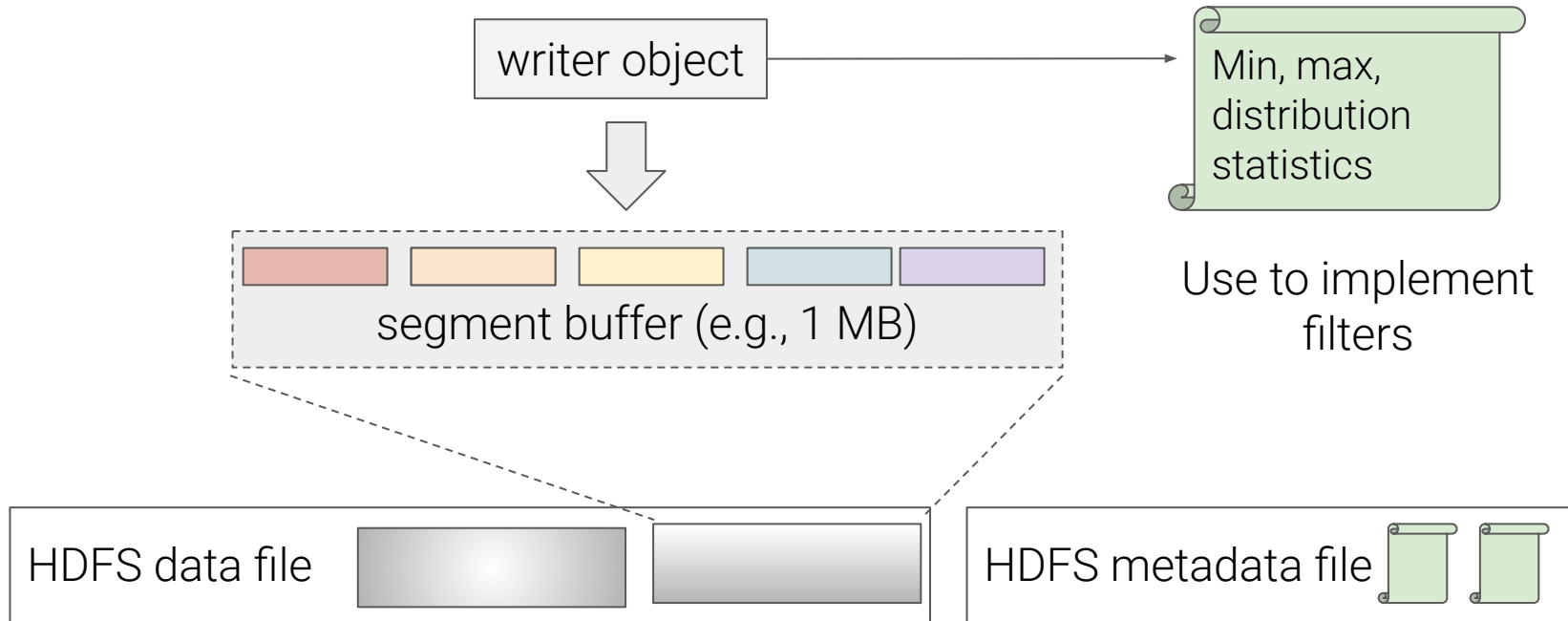
+ 1 byte bitmap (because there are 5 columns)

+ 4 byte size

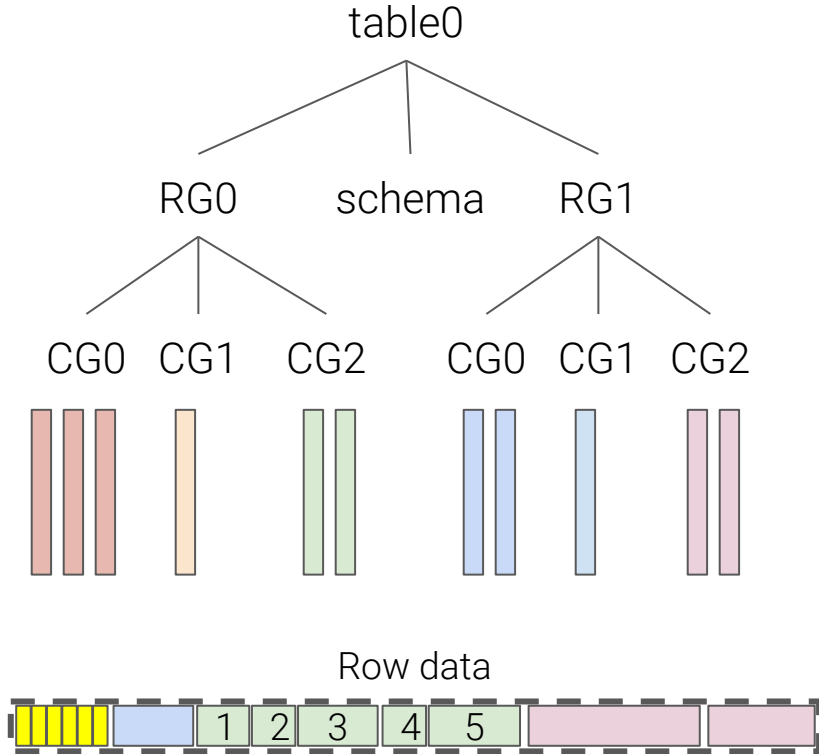
+ 4 byte (int) + 8 byte (double) + 8 byte (offset + size, ptr) + 1 byte (char) + 8 byte (offset + size, ptr)

= **34 bytes + variable area.**

Writing Rows



Reading Rows



1. Read schema file
2. Check projection to figure out which files to read
 - a. Complete CGs
 - b. Partial CGs
3. Evaluate filters to skip segments
4. Materialize values
 - a. Skip value materialization in partial CG reads

Evaluation

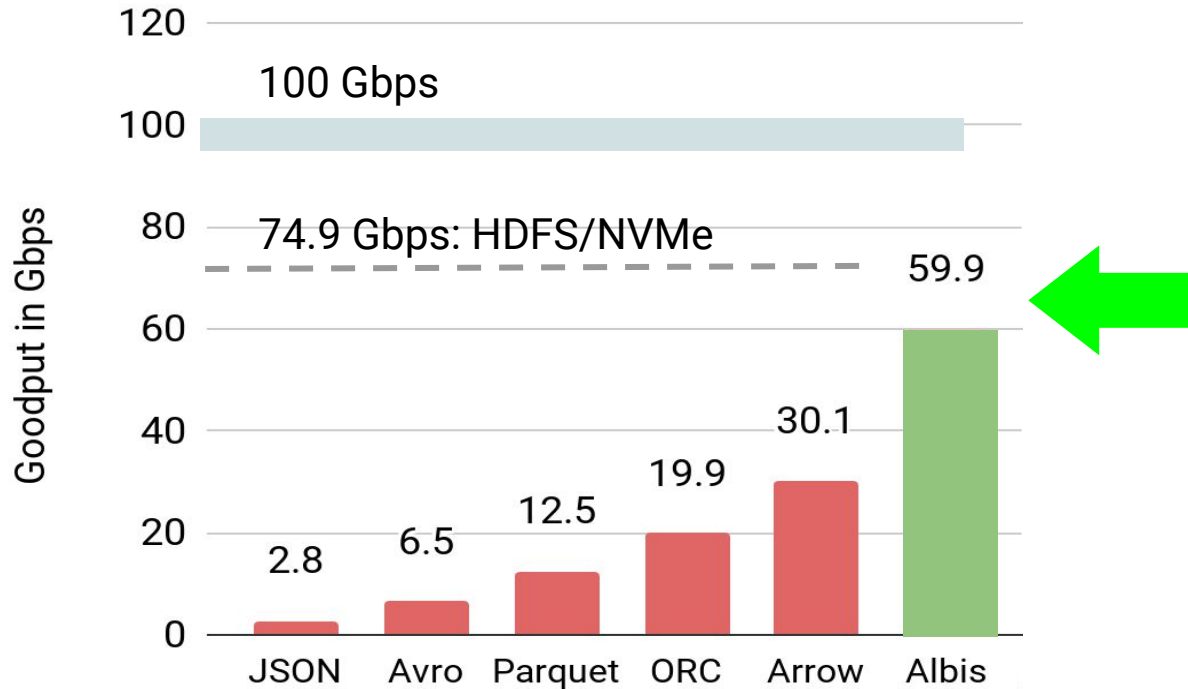
All experiments on a 4-node cluster with 100 Gbps network and flash devices

Dataset is TPC-DS tables with the scale factor of 100 (~100 GB of data)

Three fundamental questions

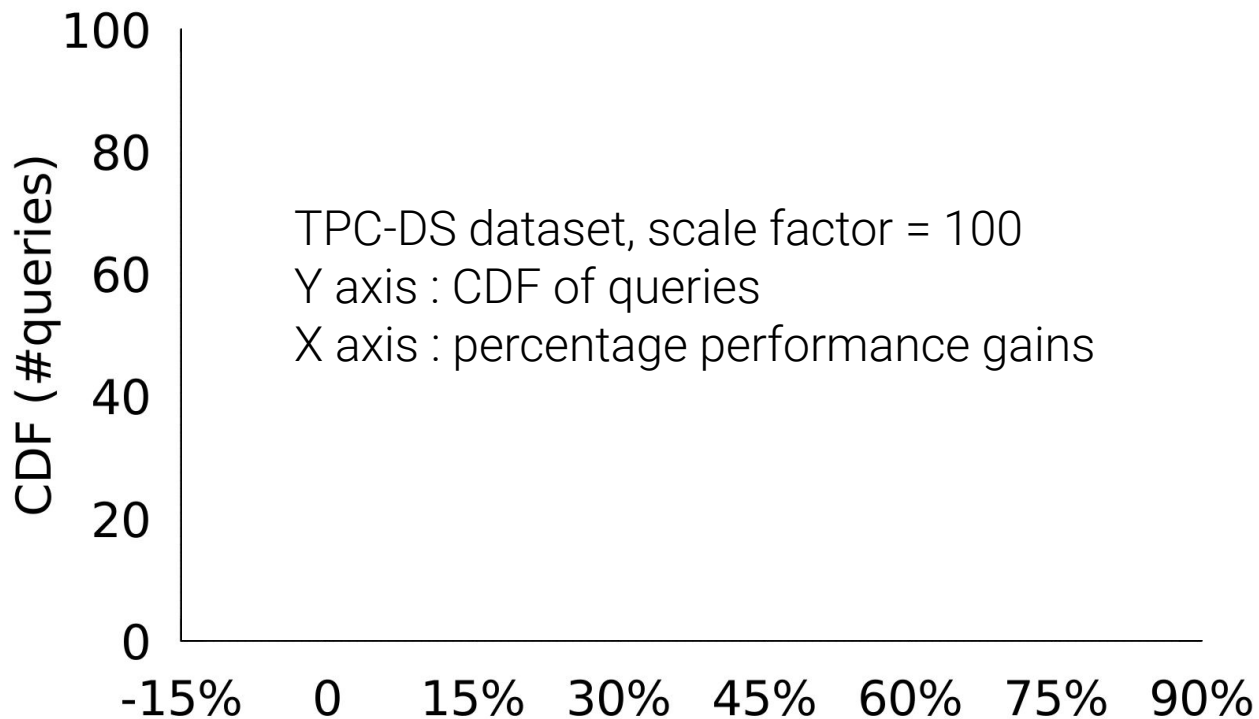
- Does Albis deliver better performance for micro-benchmarks?
- Does micro-benchmark performance translate to better workload performance?
- What is the performance and space trade-off in Albis?

Microbenchmark Performance - Revised

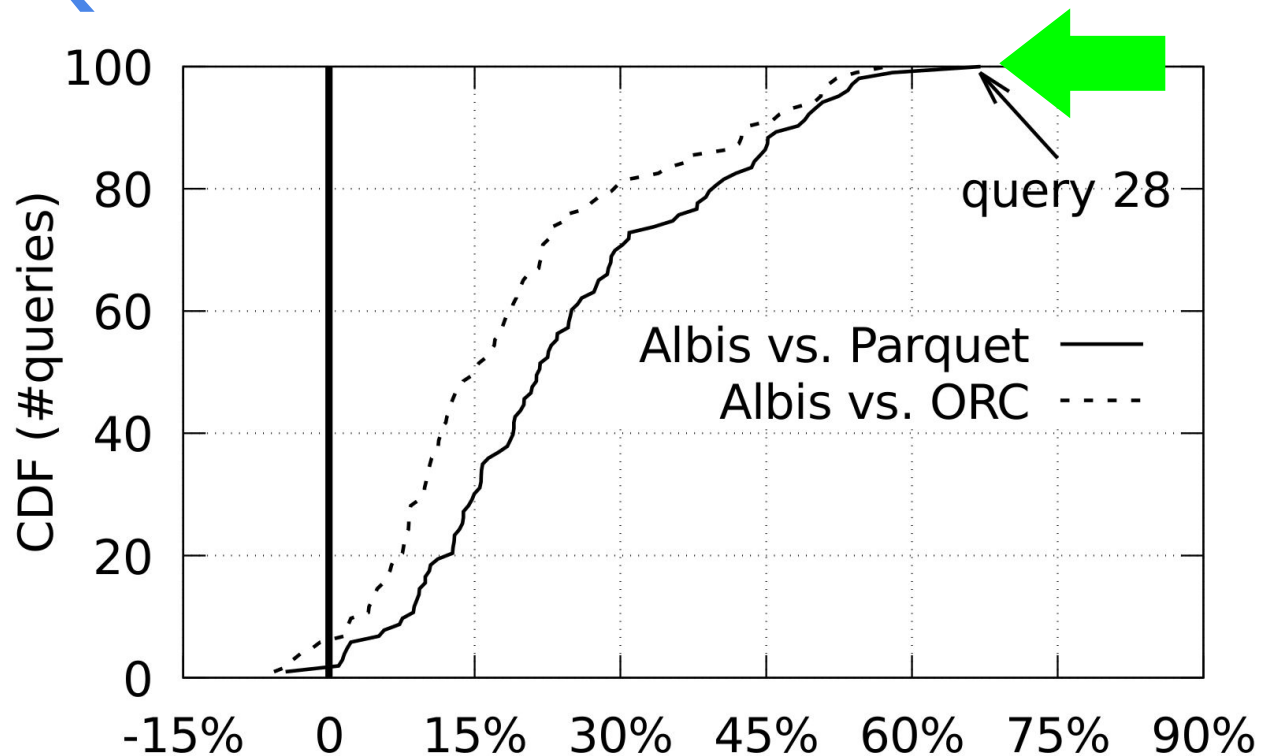


Albis delivers 1.9 - 21.3x performance improvements over other formats

Spark/SQL TPC-DS Performance



Spark/SQL TPC-DS Performance



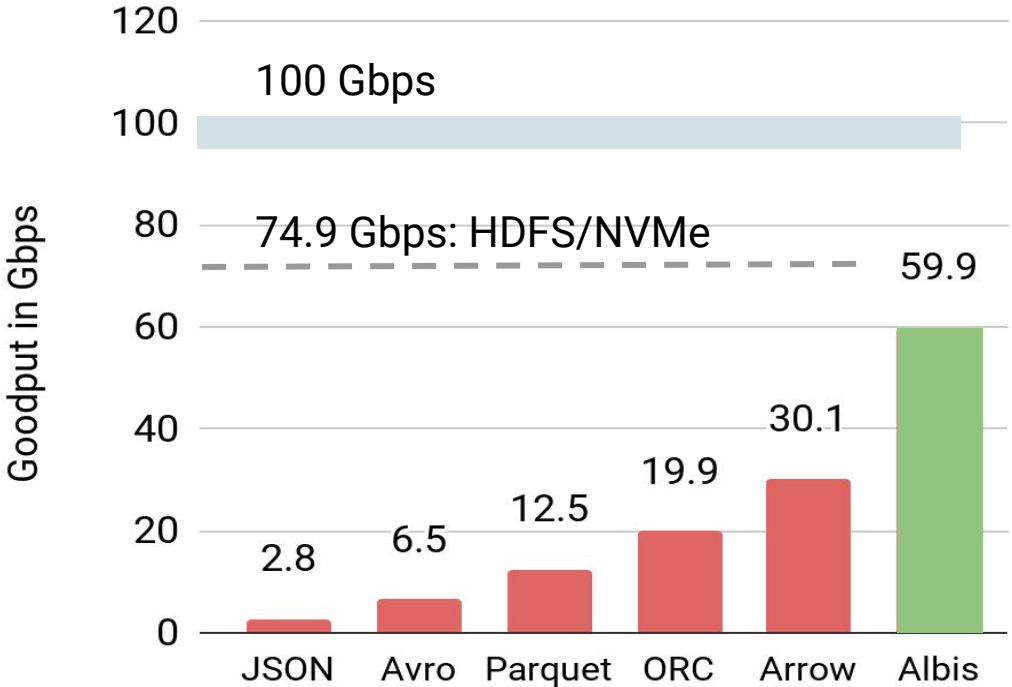
Albis delivers up to 3x performance gains for TPC-DS queries

Space vs. Performance Trade-off

	None	Snappy	Gzip	zlib
Parquet	58.6 GB 12.5 Gbps	44.3 GB 9.4 Gbps	33.8 GB 8.3 Gbps	N/A
ORC	72.0 GB 19.1 Gbps	47.6 GB 17.8 Gbps	N/A	36.8 GB 13.0 Gbps
Albis	94.5 GB 59.9 Gbps	N/A	N/A	N/A

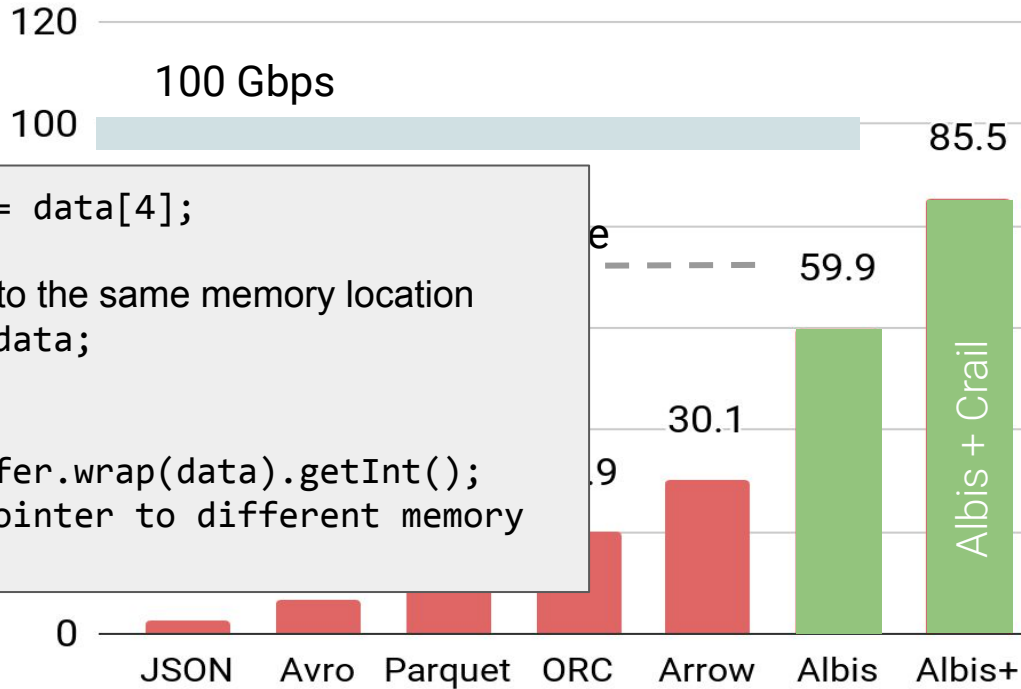
Albis inflates data by 1.3 - 2.7x, but gives 3.4 - 7.2x performance gains

Microbenchmark Performance - Revised



What would it take to deliver 100 Gbps?

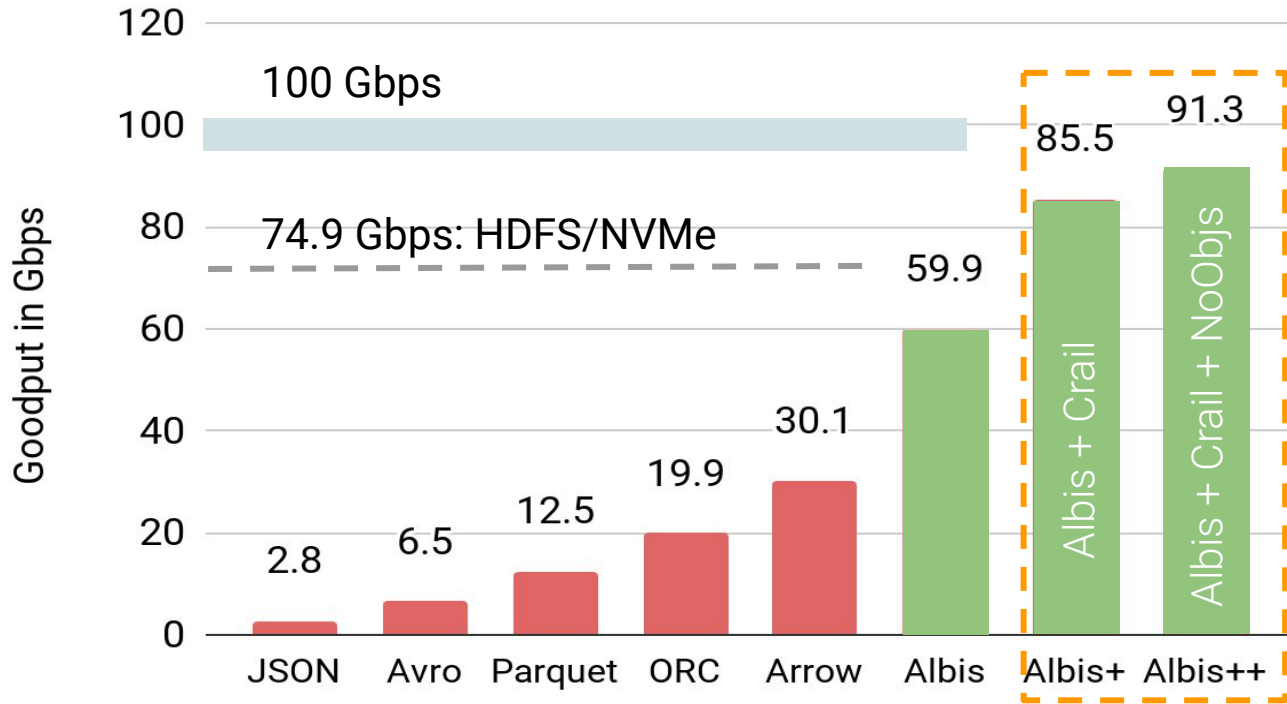
Microbenchmark Performance - Revised



```
byte [] raw_data = data[4];  
  
// In C/C++ , pointing to the same memory location  
int val = (int*) data;  
  
// in Java/Scala  
int val = ByteBuffer.wrap(data).getInt();  
// val and data pointer to different memory  
// locations
```

JVM object
overheads

Microbenchmark Performance - Revised



Albis can deliver performance within 10% of hardware

Think about

When does Albis-type data storage format does not make sense?

1. CPU is fast enough to compute (compress, encode, materialize objects) faster than I/O device bandwidth
 - a. Is CPU getting faster? Are I/O devices getting faster?
2. Is space vs. performance trade-off acceptable?
 - a. Not all data is equally performance sensitive
 - b. Not all data is hot - cold data needs to be compressed and stored efficiently
3. Anything else? Albis is only evaluated in the Cloud/HDFS/Crail
 - a. Building Albis on OCSSDs would be an interesting exercise

From this Lecture You Should Know

1. What is temporary data
2. Why does temporary data needs special treatment
 - a. In the critical path
 - b. Large size distribution
 - c. No fault tolerance (can be supported by the framework itself)
3. How does modern networking (RDMA) and storage (NVMe/NVMeF) help to build fast Crail-type system
 - a. What does control and data path split means
 - b. What does unification of abstractions in the NodeKernel model mean
4. What is Albis and how does its design leverage modern networking and storage hardware
 - a. Reduce CPU involvement - simple format and easy layout on file system

Further Reading

1. Disaggregating Persistent Memory and Controlling Them Remotely: An Exploration of Passive Disaggregated Key-Value Stores, USENIX ATC 2020.
2. Youyou Lu, Jiwu Shu, Youmin Chen, and Tao Li. 2017. Octopus: an RDMA-enabled distributed persistent memory file system. In Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '17). USENIX Association, USA, 773–785.
3. Animesh Trivedi, Patrick Stuedi, Jonas Pfefferle, Adrian Schuepbach, and Bernard Metzler. 2018. Albis: high-performance file format for big data systems. In Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '18). USENIX Association, USA, 615–629.
4. Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, Ana Klimovic, Adrian Schuepbach, and Bernard Metzler. 2019. Unification of temporary storage in the nodekernel architecture. In Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '19). USENIX Association, USA, 767–781.
5. <https://www.alluxio.io/>
6. RAMCloud Project, <https://ramcloud.atlassian.net/wiki/spaces/RAM/overview>
7. V. Srinivasan, Brian Bulkowski, Wei-Ling Chu, Sunil Sayyaparaju, Andrew Gooding, Rajkumar Iyer, Ashish Shinde, and Thomas Lopatic. Aerospike: Architecture of a real-time operational dbms.Proc. VLDB Endow.,9(13):1389–1400, September 2016.
8. Shuotao Xu, Sungjin Lee, Sang-Woo Jun, Ming Liu, Jamey Hicks, and Arvind. Bluecache: A scalable distributed flash-based key-value store. Proc. VLDB Endow., 10(4):301–312, November 2016