Practical Course: Cloud-Based Data Processing

Michalis Georgoulakis

Chair for Database Systems
School of Computation, Information and Technology
Technical University of Munich

21.04.2026

Lab 1 is over

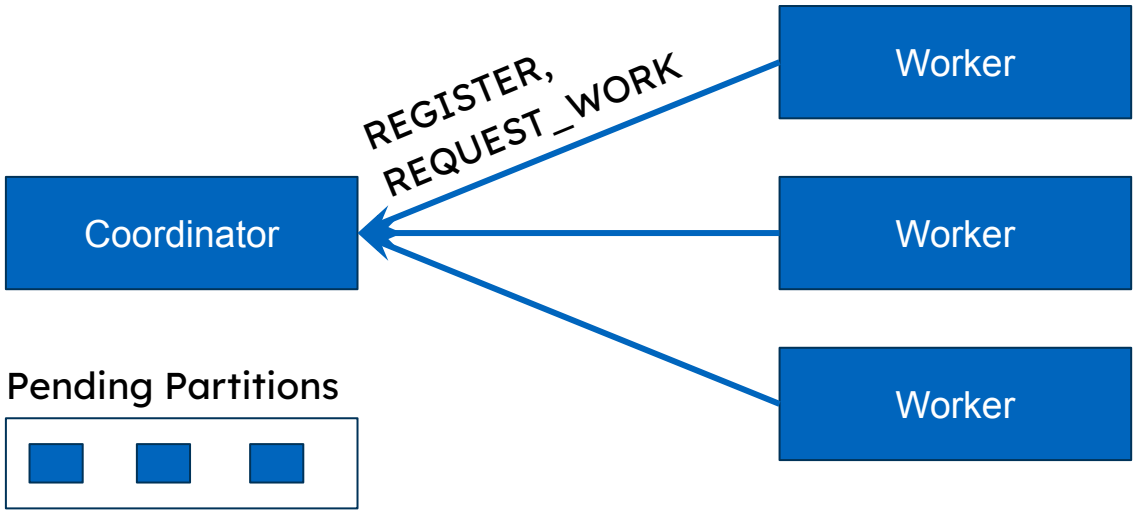
- You implemented
  - Reliable TCP framing
  - Shared-state coordination under a mutex
  - Local data analysis
  - Protocol as state machine, implemented in the worker loop

- Questions?

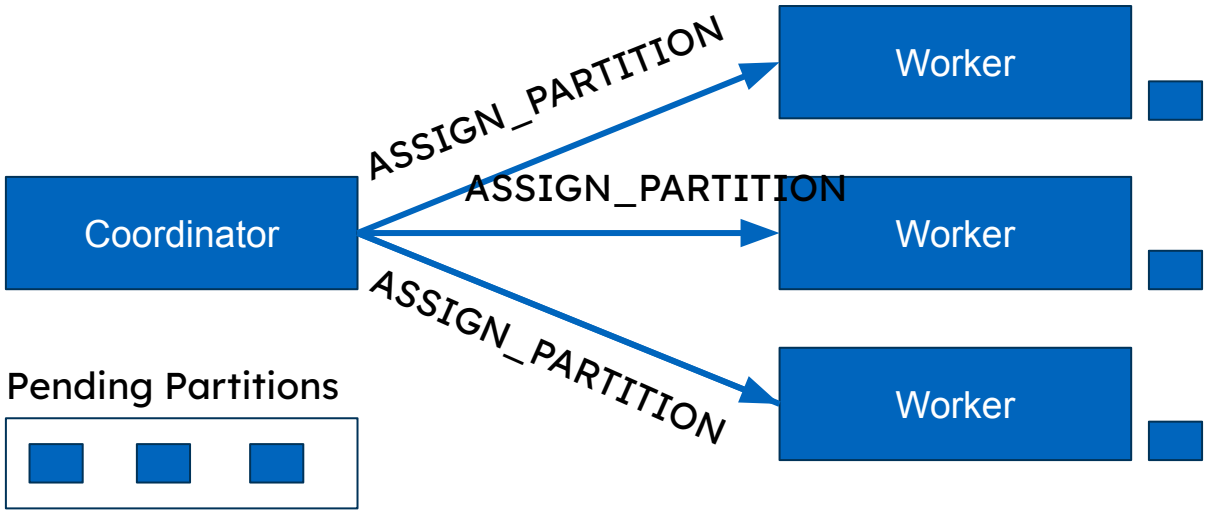
- Next Week: First Assignment Q&A
  - We will discuss it in more detail
  - Be prepared to present your solution

- Pushing your solution is sufficient
  - Do not open PRs to the parent repo.

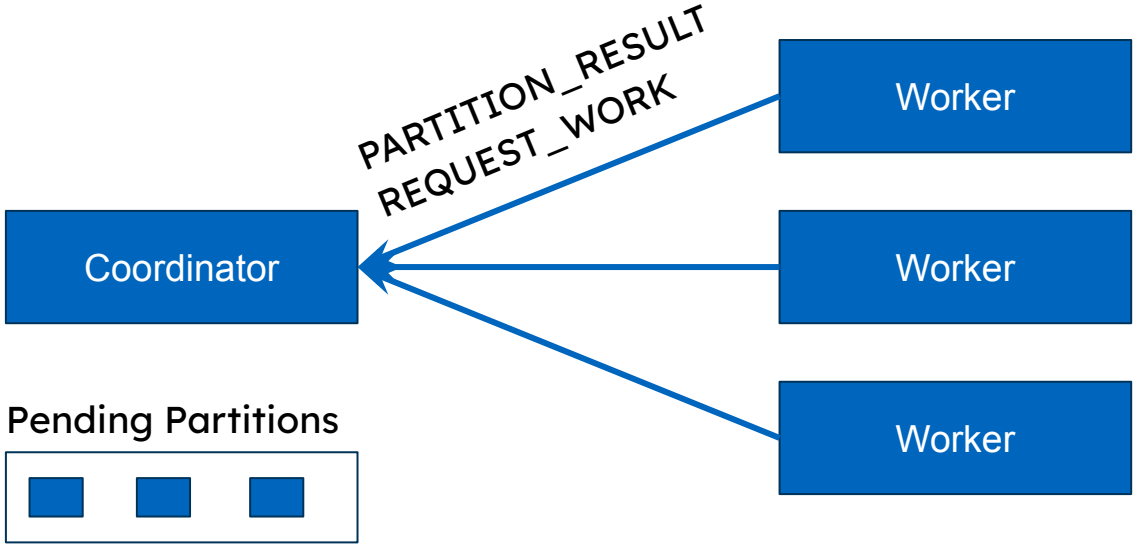
Failure Mode Example



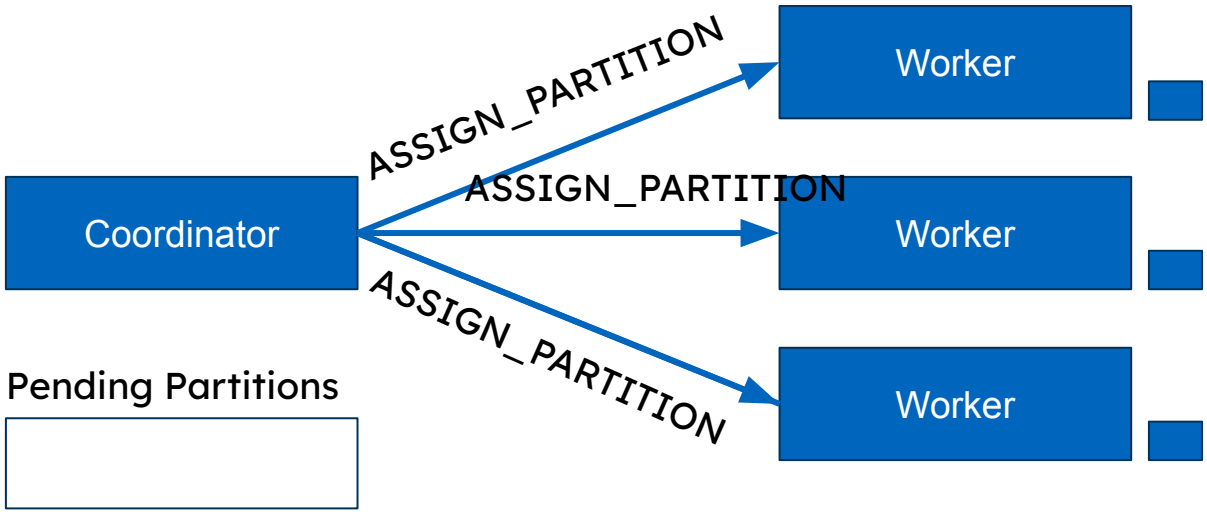
Failure Mode Example



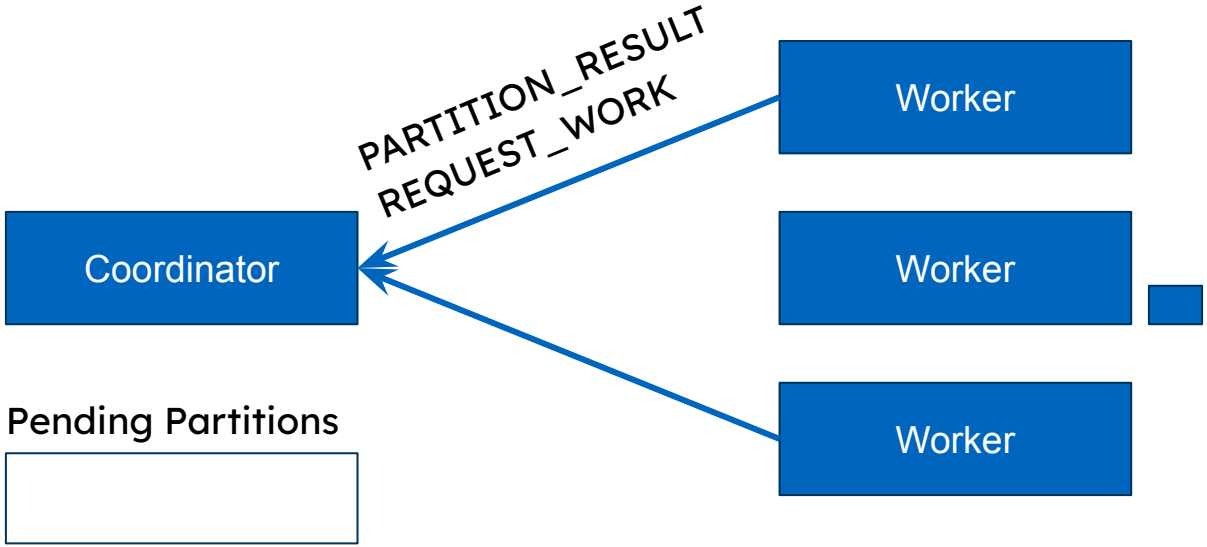
Failure Mode Example



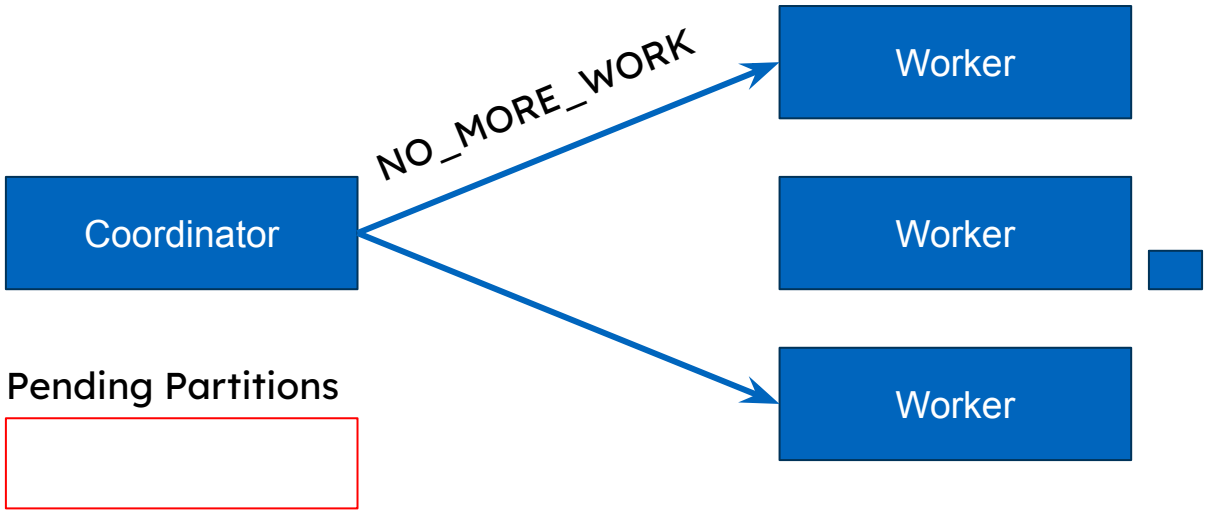
Failure Mode Example



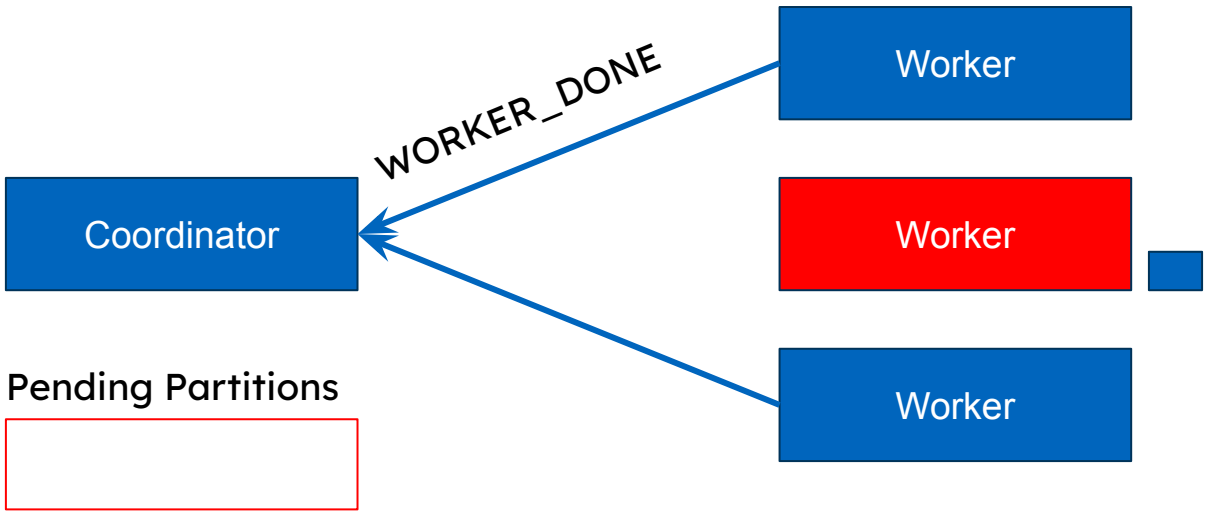
Failure Mode Example



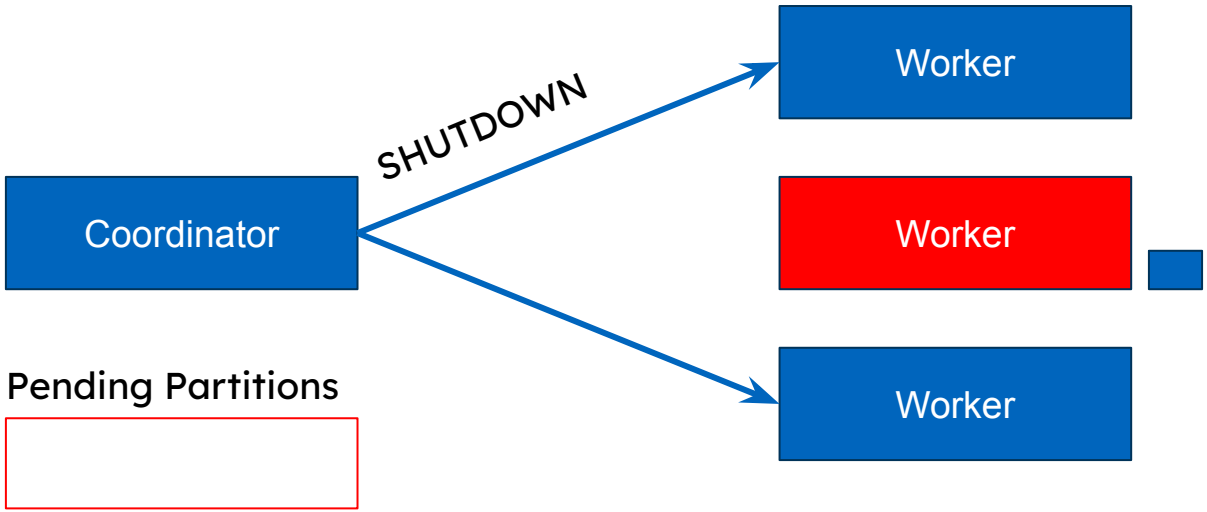
Failure Mode Example



Failure Mode Example



Failure Mode Example

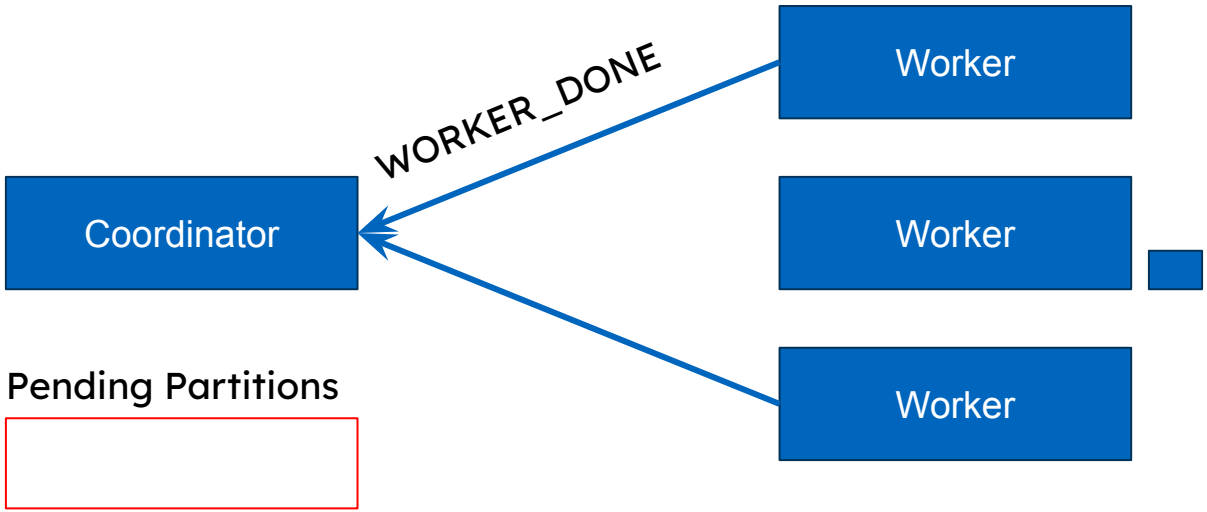


Failure Mode Example

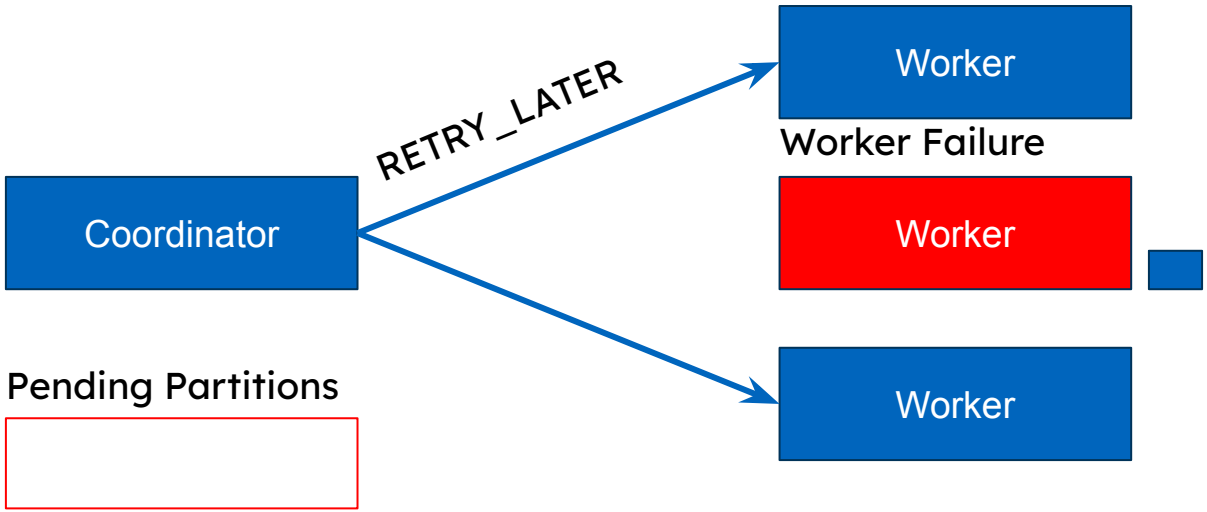
Failure Mode Example

Because there were no pending tasks, all workers have shut down and there is no one to take over the failed worker's task.

Failure Mode Example



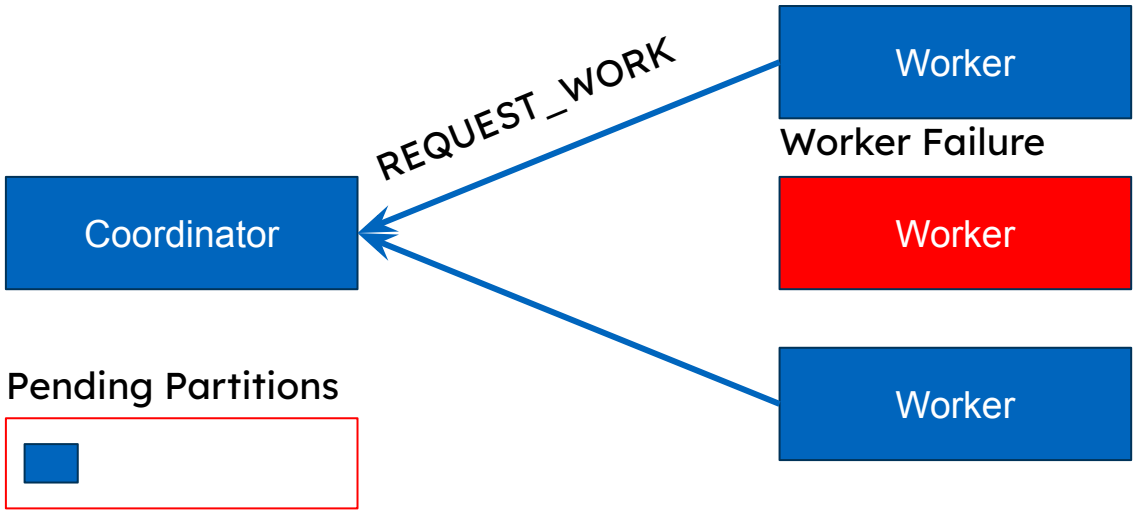
Failure Mode Example



Pending tasks are empty, but there still exist in-progress tasks.

- Sleep and retry until both pending and in-progress are empty.

Failure Mode Example



Pending tasks are empty, but there still exist in-progress tasks.

- Sleep and retry until both pending and in-progress are empty.

Distributed programming considerations

You have a correct system. Is it fast?

- You've just built the control plane of a distributed application.
- It handles single-worker crashes and stuck workers.
- But "works" ≠ "scales".

Parallelism maximum speedup

Amdahl's Law tells us that speedup from parallelism is limited by the fraction of a task that cannot be parallelized.

➔ Even with infinitely many machines, the serial part becomes the performance bottleneck.

Amdahl's Law (Speedup with n machines)

$$\text{Speedup}(n) = \frac{1}{(1-p) + \frac{p}{n}}$$

Where *p* = parallelizable fraction of work (0 ≤ p ≤ 1), *1–p* = inherently serial fraction

Parallelism maximum speedup

Amdahl's Law tells us that speedup from parallelism is limited by the fraction of a task that cannot be parallelized.

➔ Even with infinitely many machines, the serial part becomes the performance bottleneck.

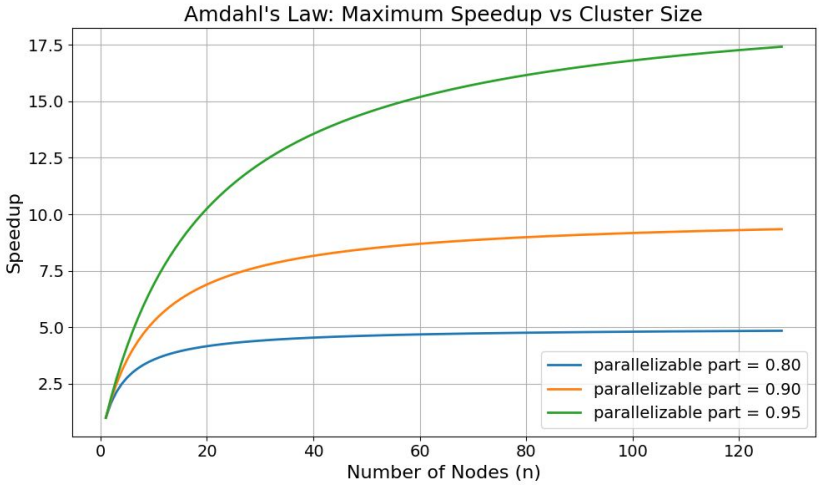
Amdahl's Law (Speedup with n machines)

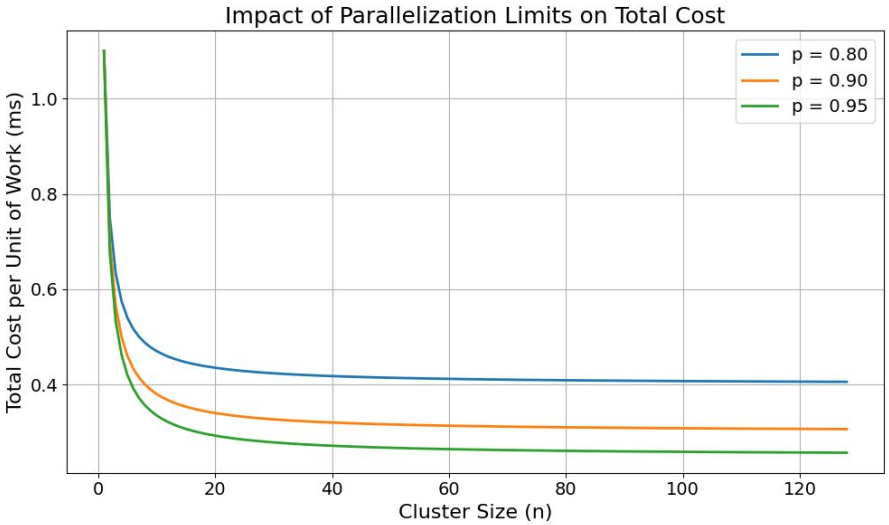
$$\text{Speedup}(n) = \frac{1}{(1-p) + \frac{p}{n}}$$

Consequently, per-unit compute cost becomes:

$$\text{ComputeCost}(n) = (1-p)\,\text{compute\_base} + \frac{p\,\text{compute\_base}}{n}$$

Parallelism maximum speedup





Performance At Scale

Distributed Systems Change How We Think About Performance

In large-scale systems, a single user request often involves:

- Many microservice calls
- Many partition lookups
- Many replicas or nodes

Overall latency = latency of the slowest component.
As fan-out increases, slow outliers dominate.

Why this matters

- Optimizing only average latency is not enough.
- At scale, rare slow requests happen frequently.
- Responsiveness directly impacts conversions & usage (e.g., Amazon found every +100ms latency → ~1% drop in sales).

Tail latency - Fanout

Average ≠ Typical

- Average latency hides long-tail outliers.
- Tail latency typically refers to:
  - **P95** → 95h percentile, **P99** → 99th percentile, **P99.9** → 99.9th percentile
  - The “slowest X%” of requests

With N parallel sub-requests:

- Probability all are fast = (0.99)^N
- Tail requests multiply with fan-out

Example: If each service has: 1 ms average, 1 s P99 latency:

- Then: 1 server → 1% slow requests
- 100 servers → (1 - 0.99^100) → 63% slow requests

The more distributed your system, the more your worst-case behaviors matter.

Tail Latency at Google Scale

At Google's scale:

- Single-request P99: ~10 ms
- For a query involving many back-end calls:
- Combined P99: ~140 ms
- Combined P95: ~70 ms

Half of total tail latency comes from waiting for the slowest 5% of internal calls.

	50%ile latency	**95%ile latency**	**99%ile latency**
One random leaf finishes	1ms	5ms	10ms
95% of all requests finish	12ms	32ms	70ms
100% of all requests finish	40ms	87ms	140ms

"The Tail at Scale" by Jeffrey Dean and Luiz Andre Barroso in Comm. Of the ACM, 2013

Design guidelines that follow

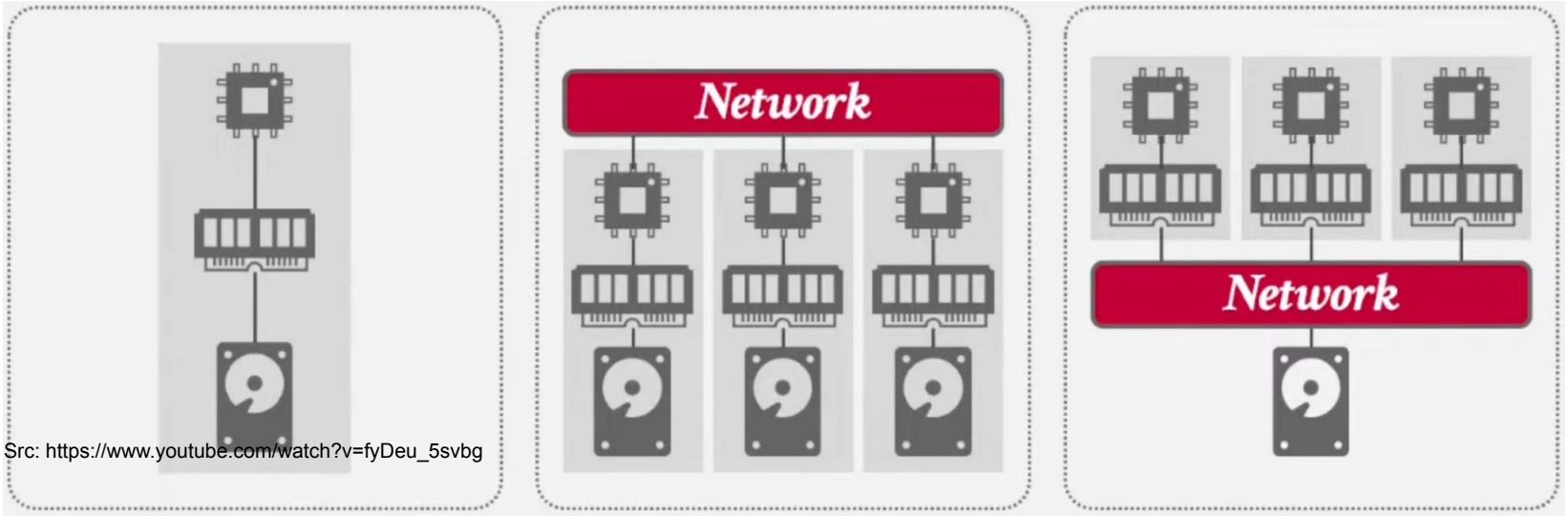
- Keep the coordinator off the data path.
- Partition data so each node works independently.
- Use local combiners, aggregate before shuffling.
- Expect fan-out stragglers, use timeouts, speculative retries.
- Pay for network only when you must (or when you know it is fast enough).

“The Tail at Scale” by Jeffrey Dean and Luiz Andre Barroso in Comm. Of the ACM, 2013

Distributed Cluster Architectures

Cluster Architecture

- Specifies what shared resources are directly accessible to the CPU.
- This affects how CPUs coordinate with each other and where they read/write objects in the database
- Two main architectures: shared-nothing, shared storage



Src: https://www.youtube.com/watch?v=fyDeu_5svbg

Shared everything (single node)

Shared Nothing

Shared Storage (Disk)

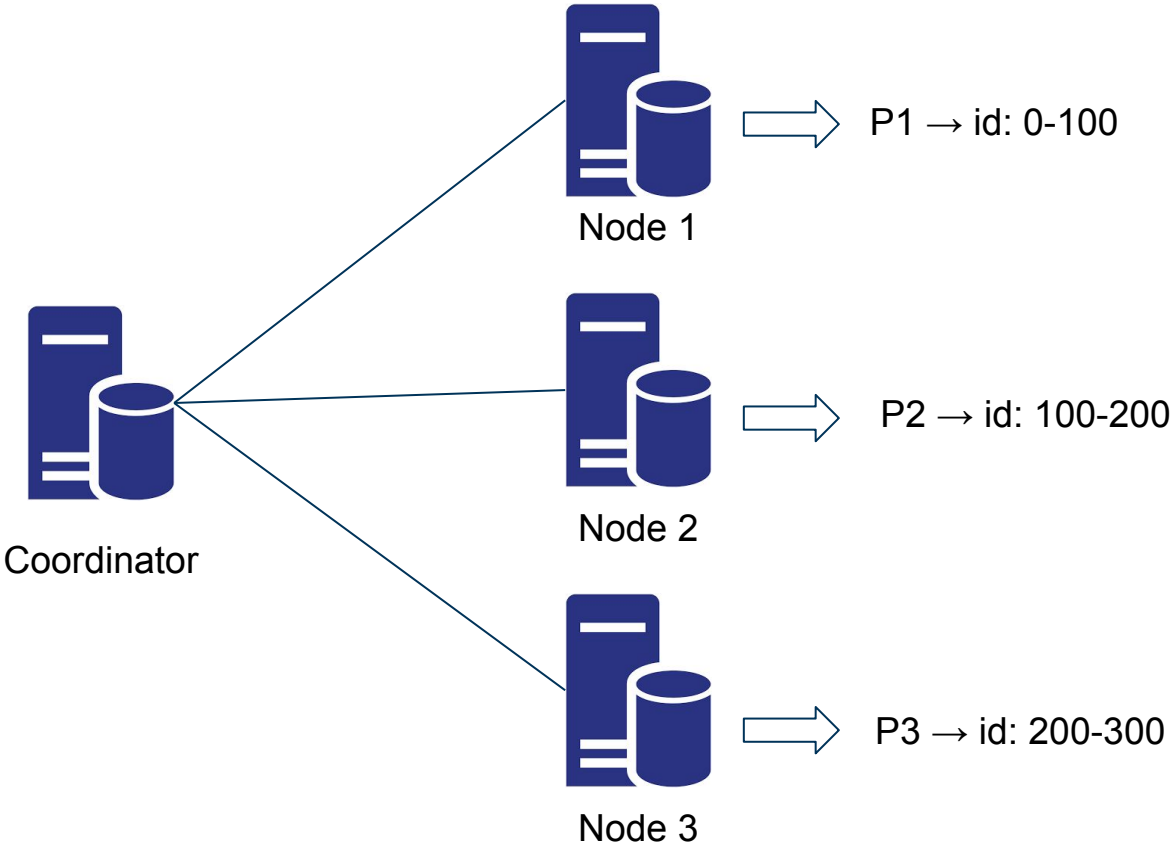
Shared Nothing Architecture

What is Shared-Nothing? A distributed architecture where each node has its own CPU, RAM, and local disk, and no storage or memory is shared across nodes.

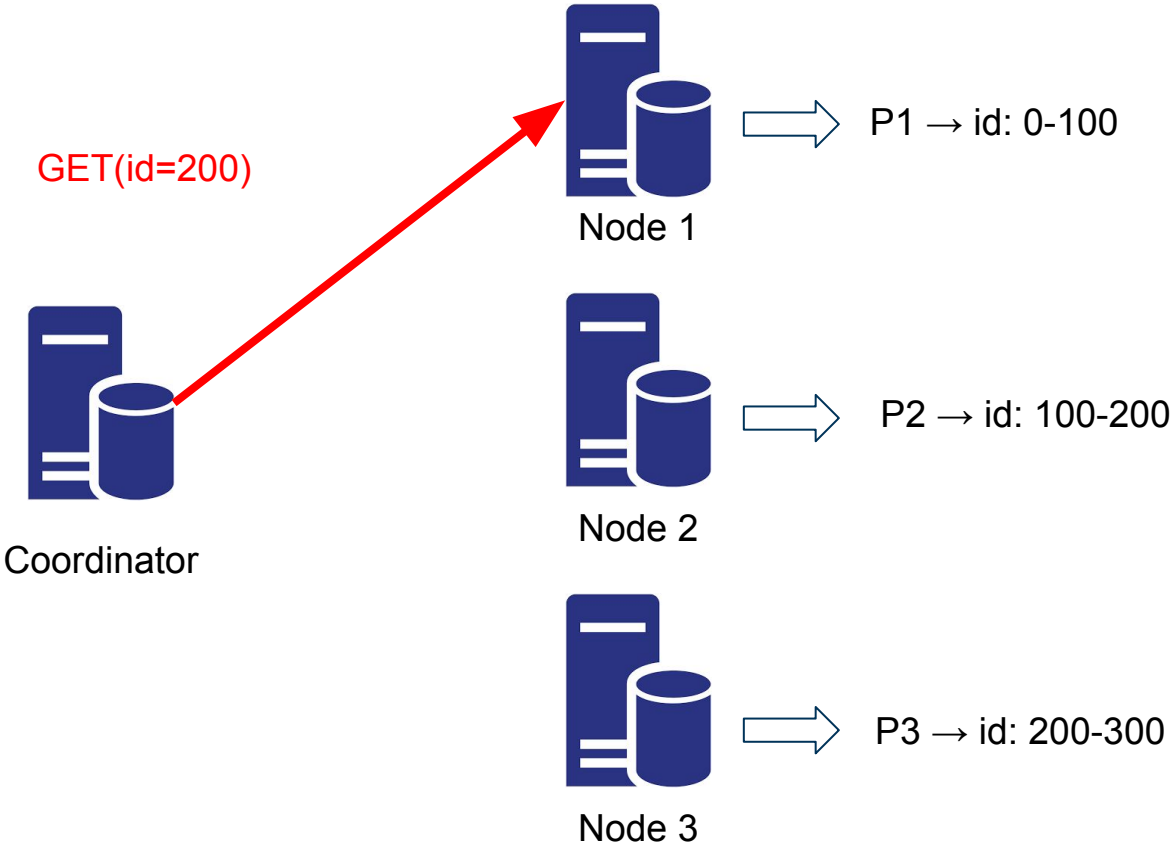
Key Properties

- Data is partitioned: each node owns a different partition of the dataset
- Nodes operate independently
- All coordination happens over the network (e.g., shuffle)
- Enables horizontal scaling and fault isolation

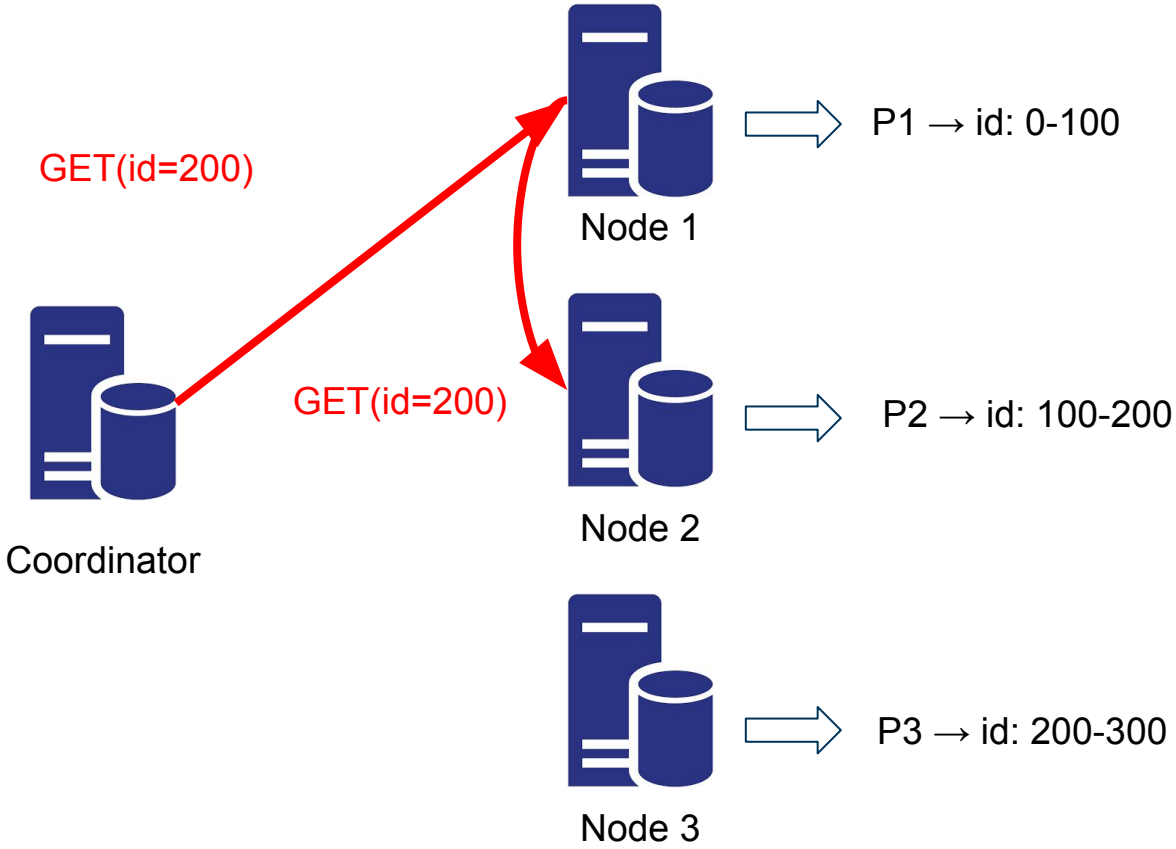
Shared Nothing Example



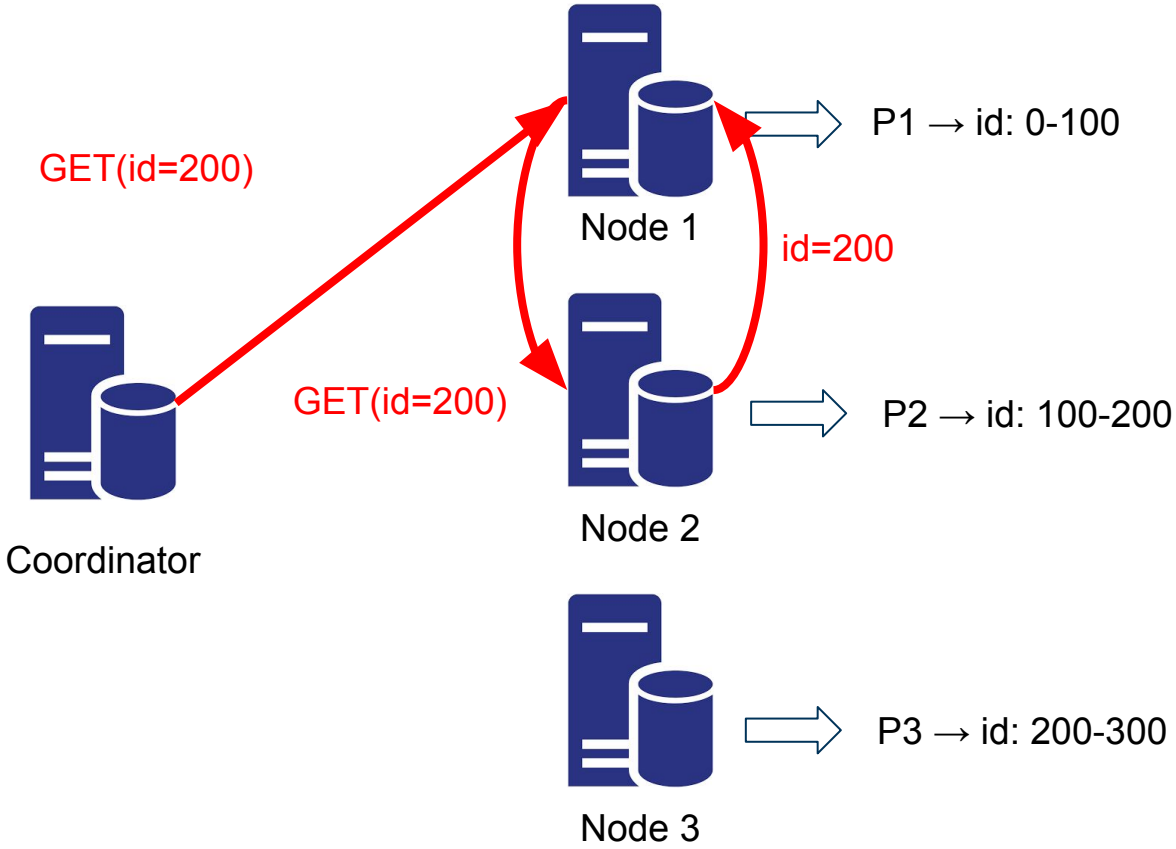
Shared Nothing Example (Read)



Shared Nothing Example (Read)



Shared Nothing Example (Read)



Shared Storage Architecture

What is Shared-Storage? A shared-storage architecture is one where multiple compute nodes access the same underlying storage system, rather than each node owning its own disks.

Key Properties

- **Decoupled storage and compute**
  - Multiple compute nodes share access to the same physical storage.
  - Storage is typically:
    - a centralized storage appliance (SAN/NAS), or
    - a distributed shared-disk system.
- Each node has its own CPU and memory, **but all read/write through the same storage layer.**
- Nodes may still have local SSDs, but these serve only as caches or temporary scratch space

Shared Storage Architecture

What is Shared-Storage? A shared-storage architecture is one where multiple compute nodes access the same underlying storage system, rather than each node owning its own disks.

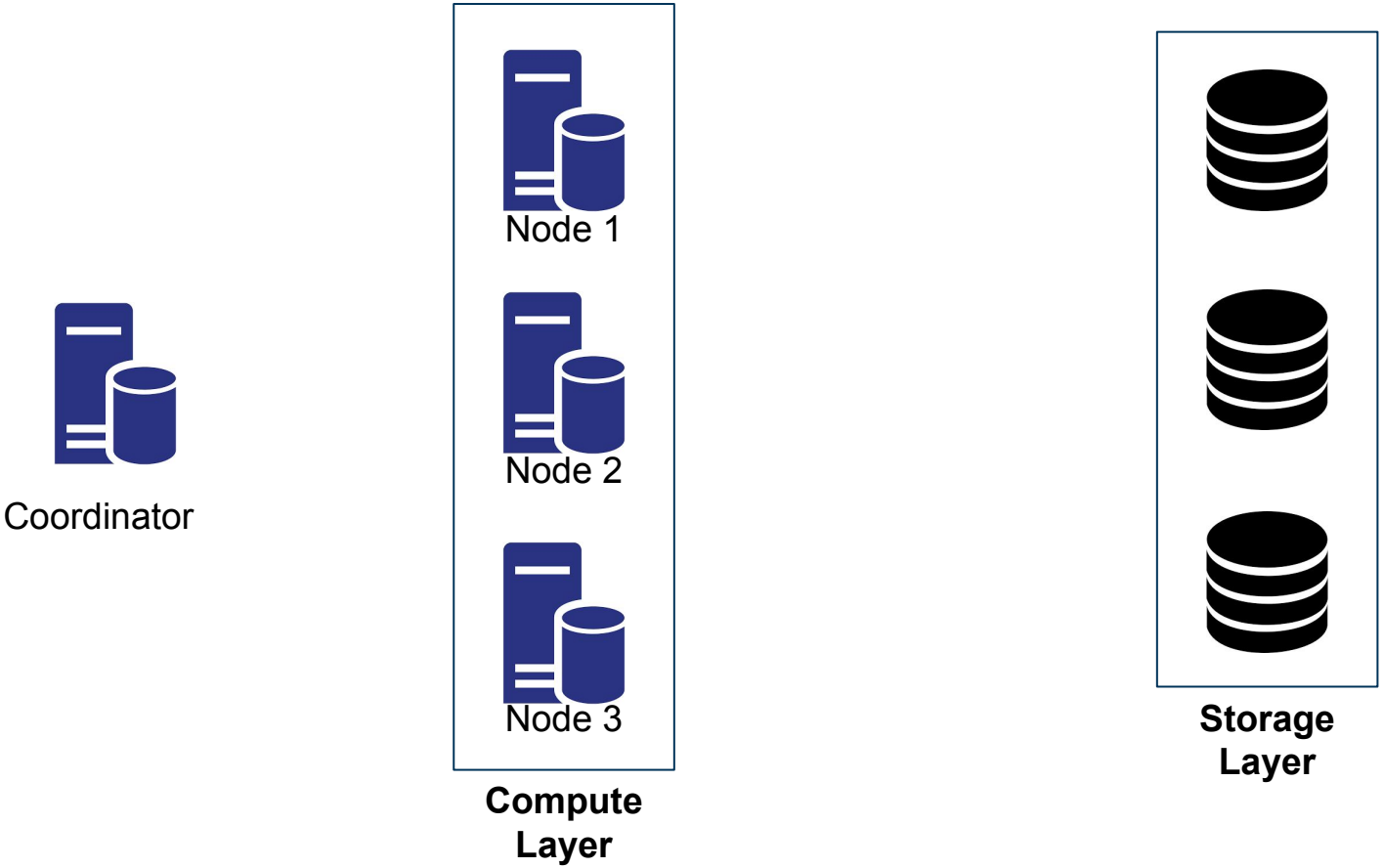
Key Properties

- **Decoupled storage and compute**
  - Multiple compute nodes share access to the same physical storage.
  - Storage is typically:
    - a centralized storage appliance (SAN/NAS), or
    - a distributed shared-disk system.
- Each node has its own CPU and memory, **but all read/write through the same storage layer.**
- Nodes may still have local SSDs, but these serve only as caches or temporary scratch space

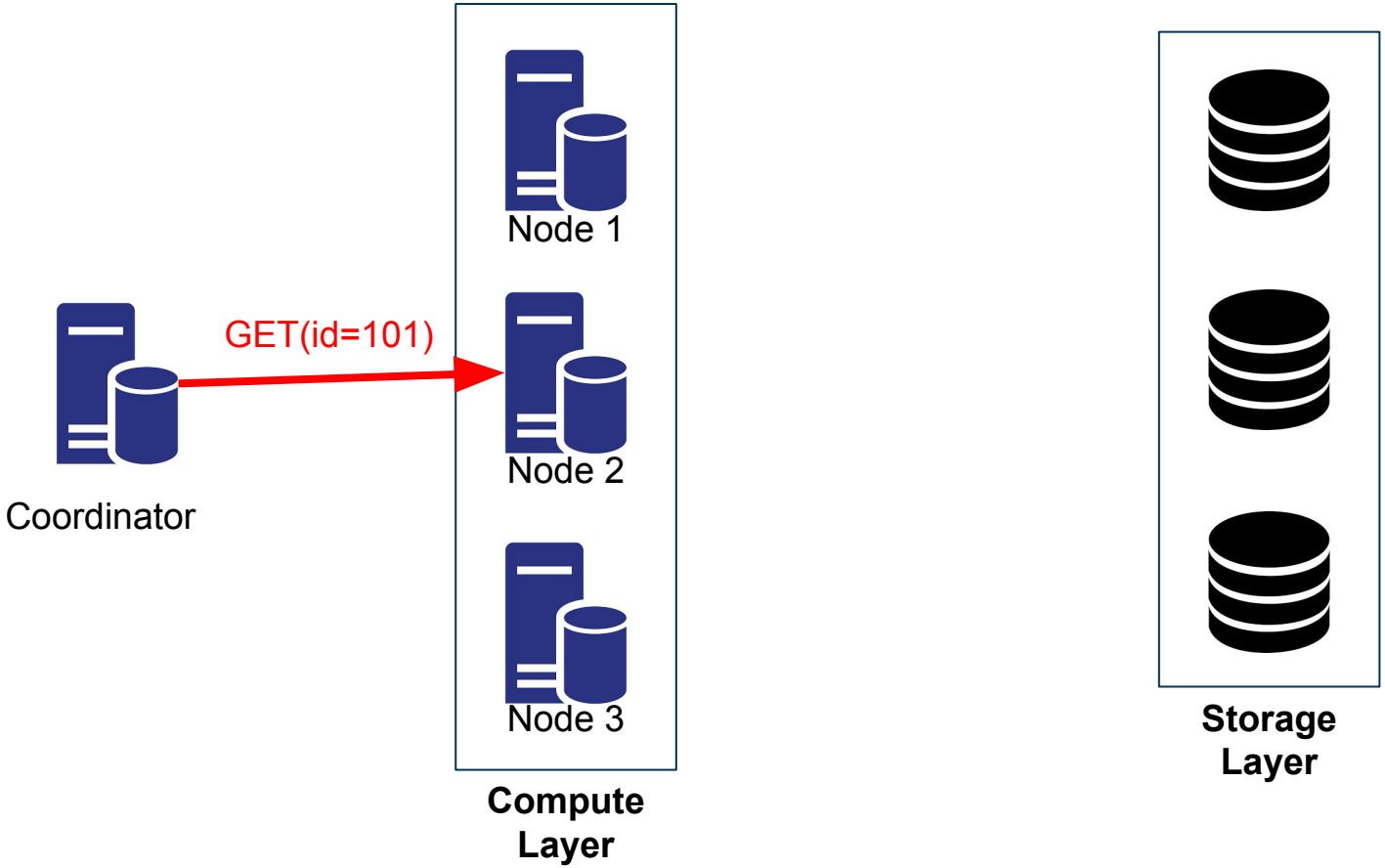
Shared Storage in the Cloud

- Cloud object stores like Amazon S3, Google Cloud Storage, and Azure Blob Storage act as a shared storage layer
- Independent compute clusters (Spark, Presto/Trino, Snowflake, etc.) can read and write the same buckets/containers

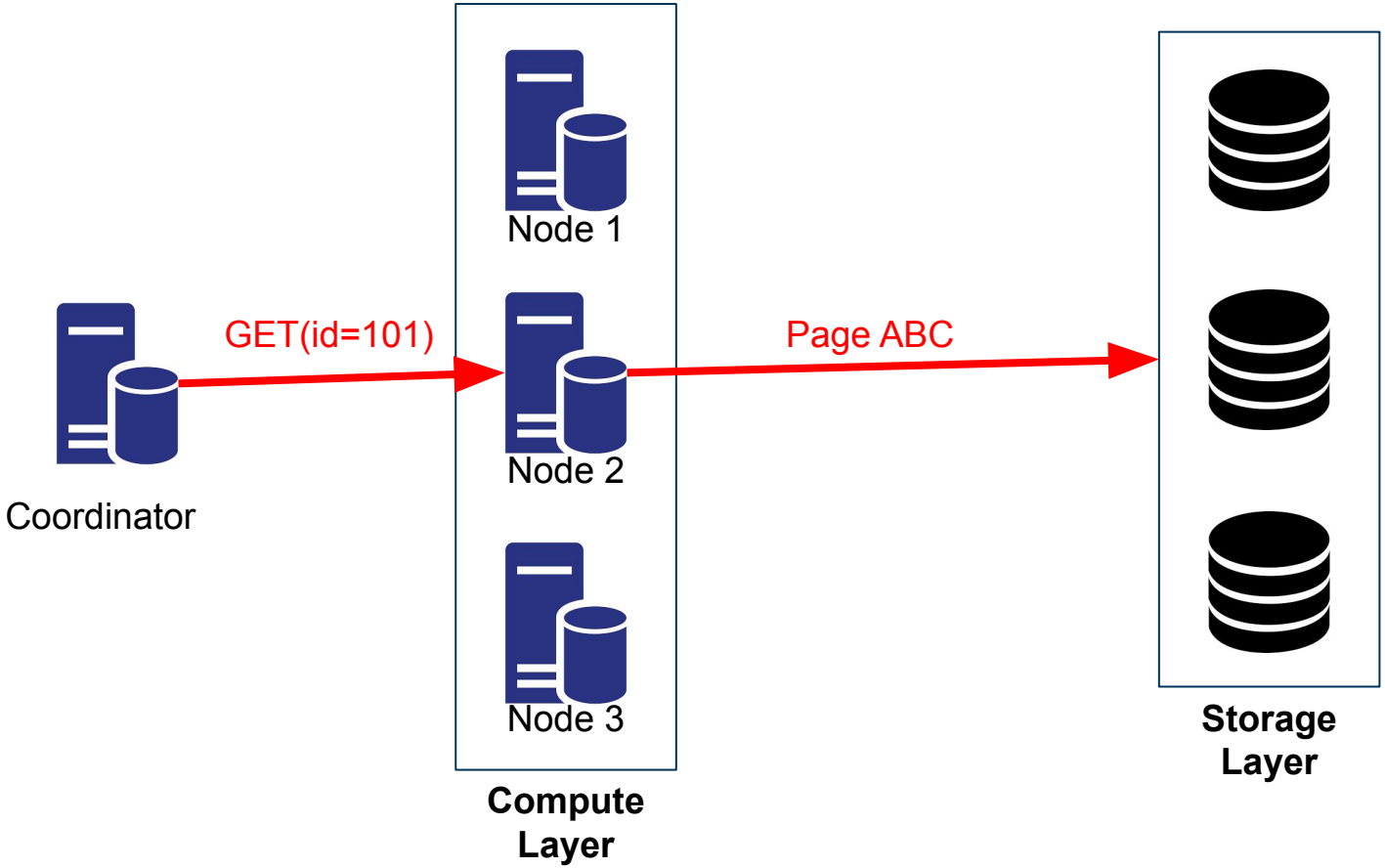
Shared Storage Example



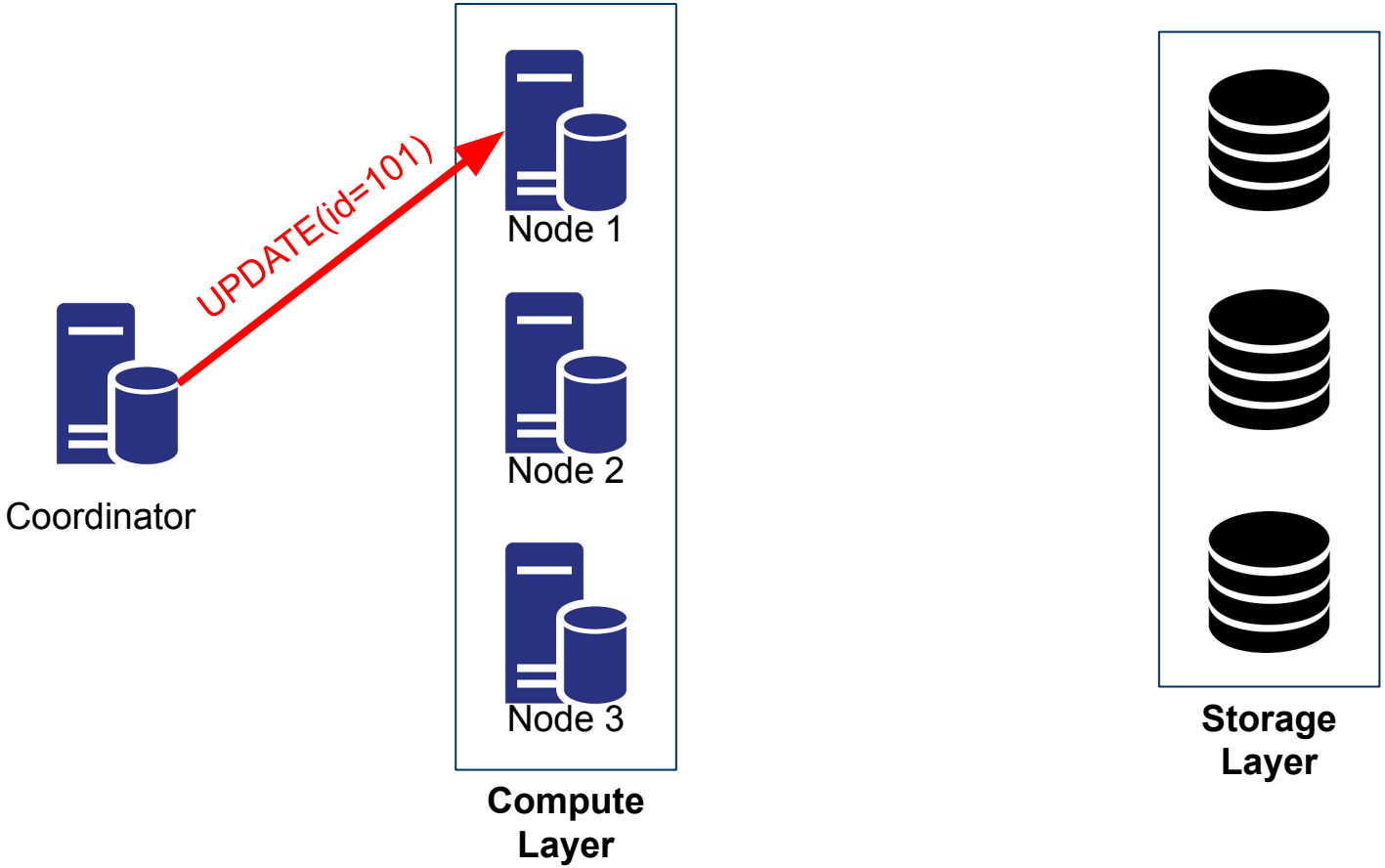
Shared Storage Example (Read)



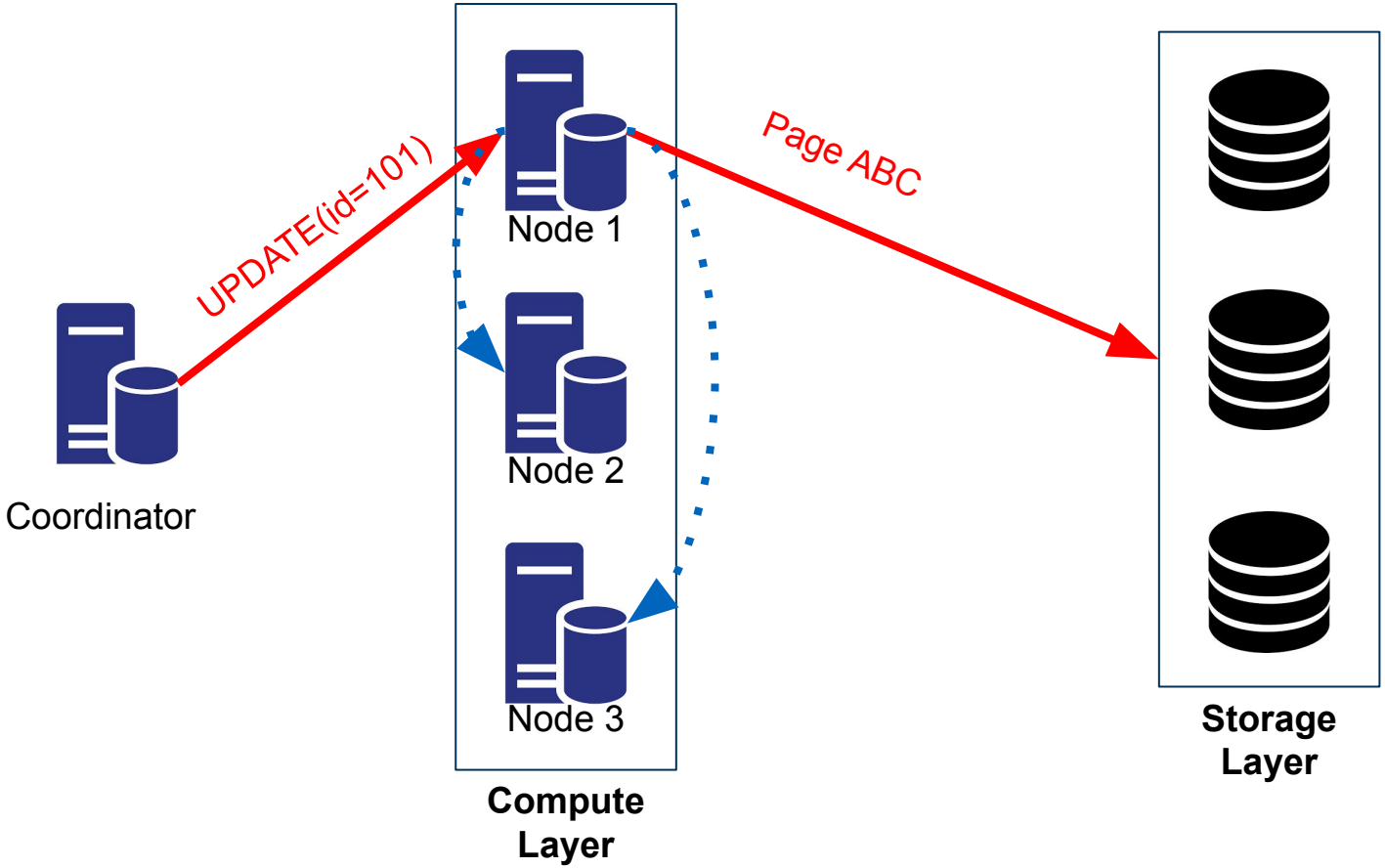
Shared Storage Example (Read)



Shared Storage Example (Write)



Shared Storage Example (Write)



Shared-nothing: the shuffle problem

In shared-nothing, data is partitioned across nodes.

- Any aggregation across partitions requires moving bytes between nodes.
- That movement is called the shuffle.

It's the expensive part of every MPP query, every Spark stage, every Lab 2 run.

What's a good programming model for the shuffle?

- Point-to-point?
- Centralized?

Programming for a Data Center - Tools

By the early 2010s, cluster computing had become the norm:

- Thousands of commodity servers
- Distributed storage systems
- Widespread cloud adoption and elastic compute

To make these clusters usable, a proliferation of tools appeared to help programmers use thousands of machines effectively.

- Their goal: hide distributed-system complexity while letting developers work at scale

The Case for a New Abstraction

We already saw how single-node data processing works.

And later we will see how it can be scaled using:

- partitioning / sharding
- Replication
- data exchange operators

But:

- Writing your own distributed solution is slow,
- Error-prone,
- And usually reinventing the wheel.

⇒ We need a higher-level abstraction for large-scale computation.

- One of the earliest and most influential distributed solutions was **MapReduce**:

MapReduce

MapReduce - The Idea

A simple functional abstraction for data-parallel computation

- MapReduce lets developers express large-scale distributed jobs using just two functions: **map** and **reduce**

Everything else is handled by the runtime:

- Parallel execution across many nodes
- Distribution of data to map tasks
- Node failures
- Communication

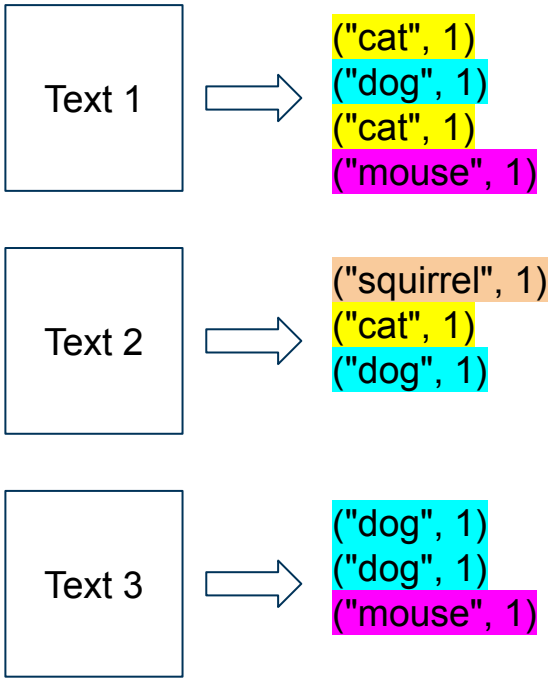
Program formulation with

- *Map* all input records to key-value pairs
- *Reduce*: Computation on all pairs with the same key

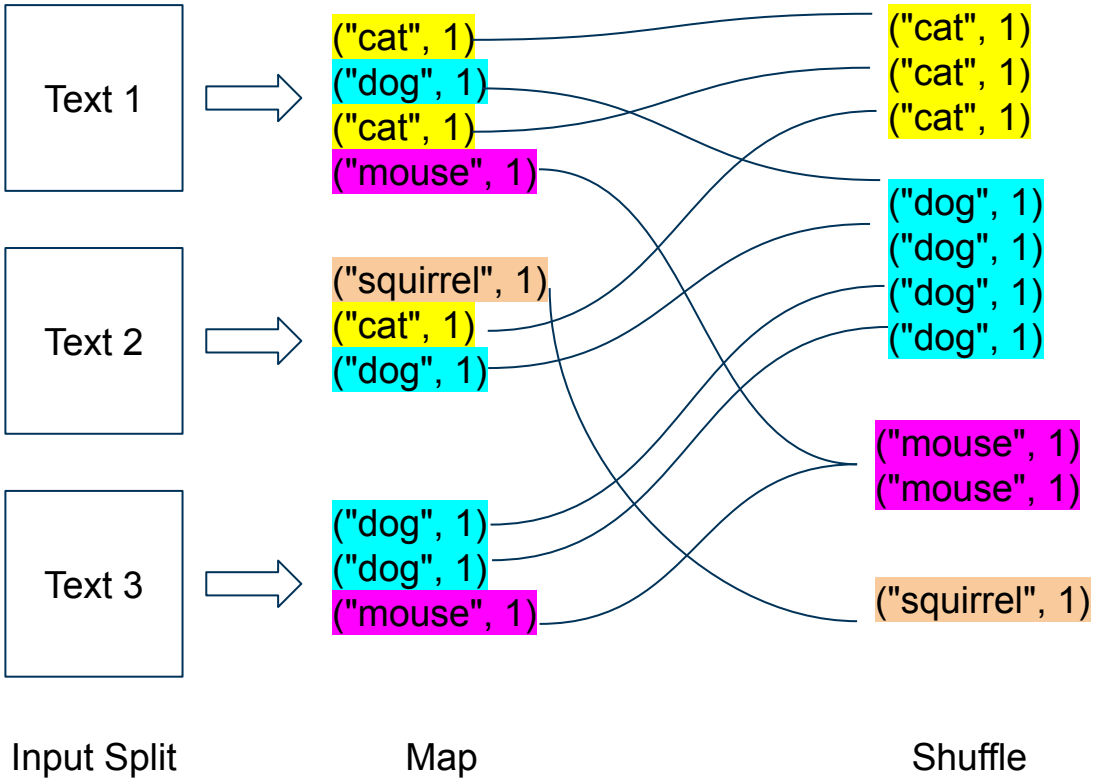
MapReduce - The Idea

Why this pattern is powerful: The system (written once) handles all the above. Tens of thousands of programmers only need to write their map and reduce computations, not distributed systems code

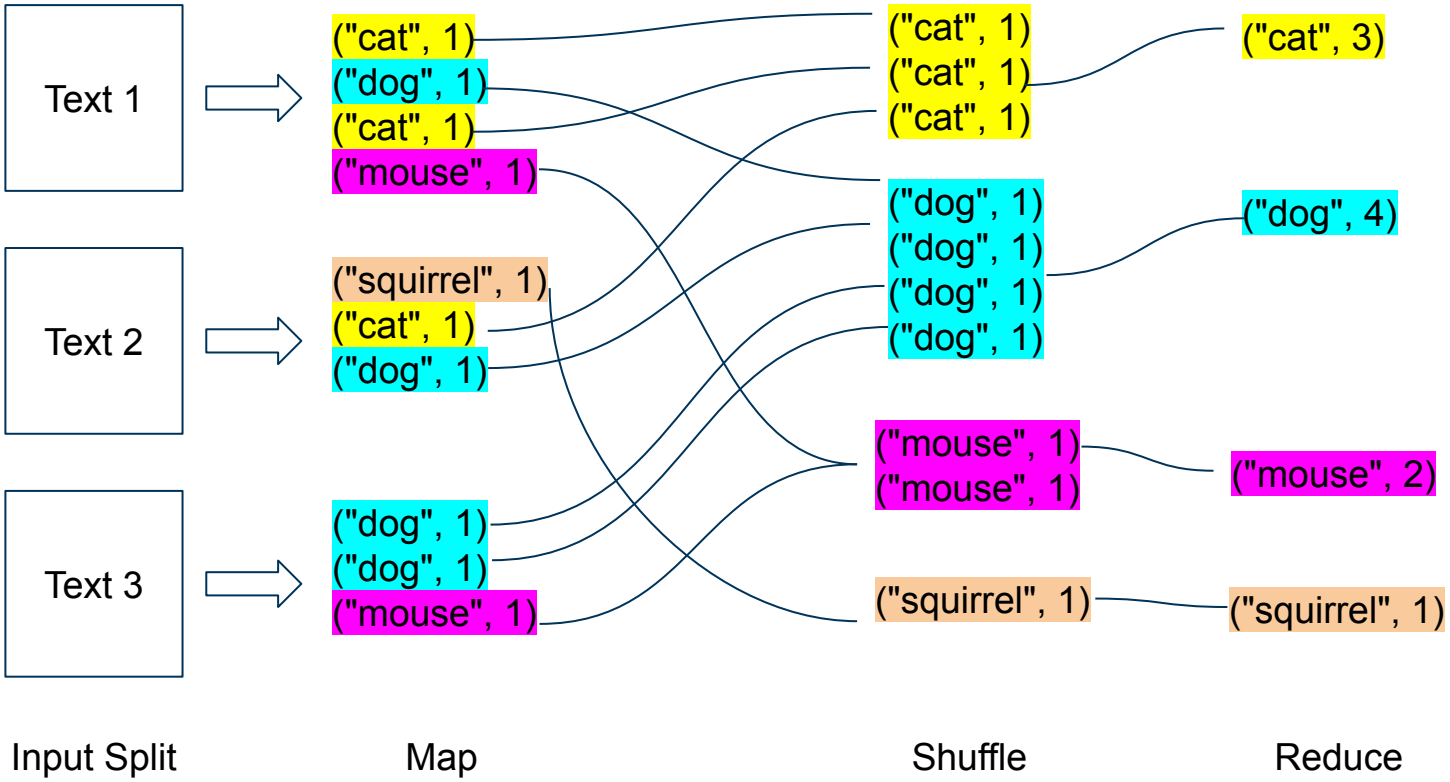
Example: WordCount in MapReduce



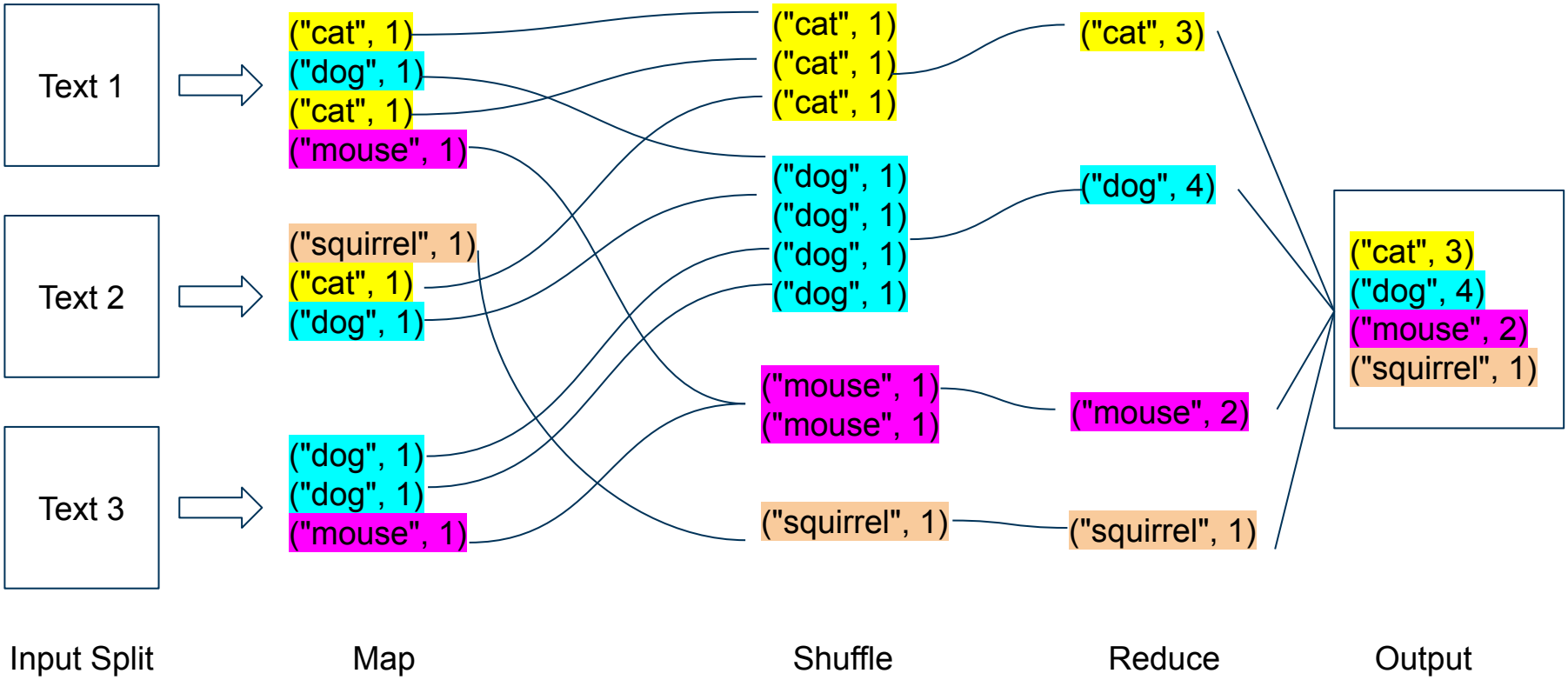
Example: WordCount in MapReduce



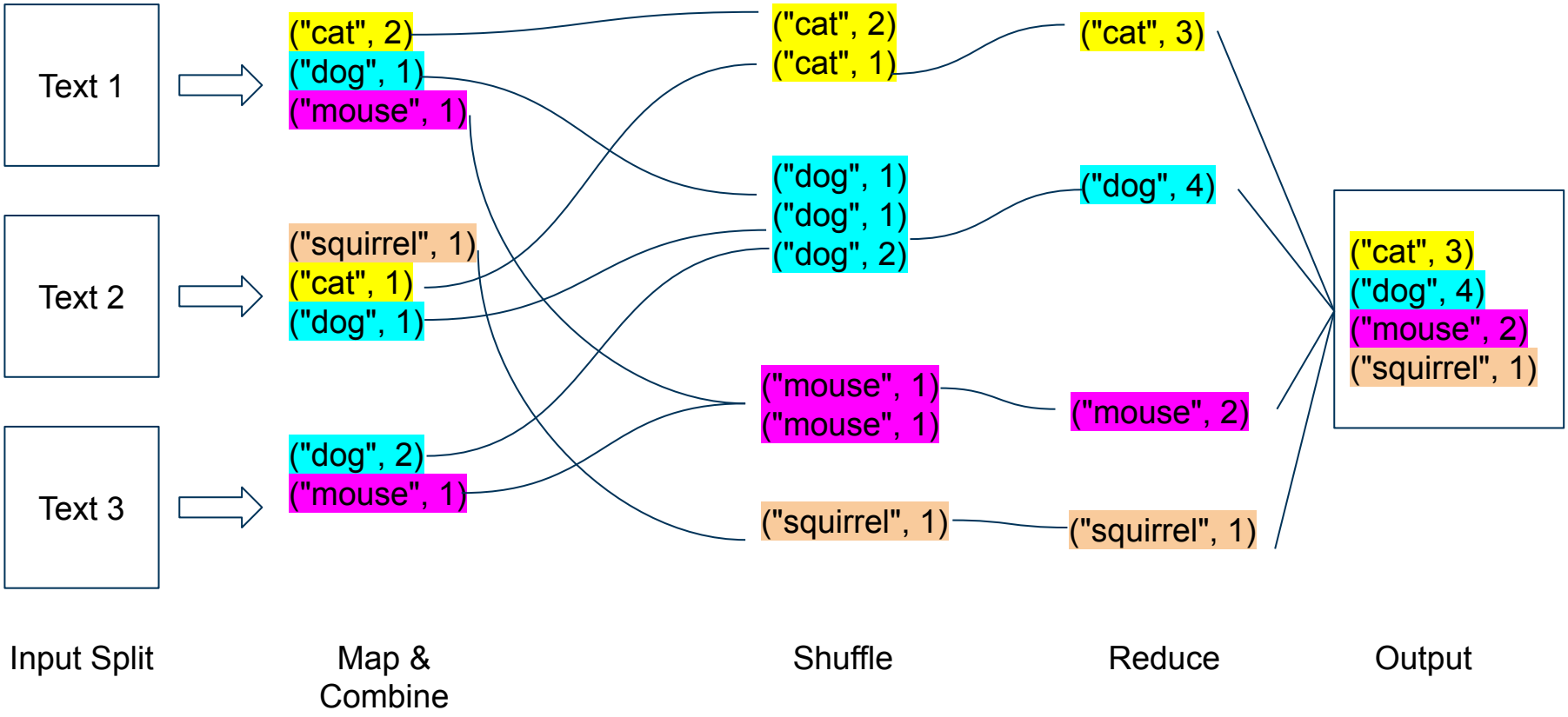
Example: WordCount in MapReduce



Example: WordCount in MapReduce



Example: WordCount in MapReduce



CloudLab

CloudLab: Scientific infrastructure for research

- Lab 1 ran on localhost: identical workers, no stragglers, no network communicatoin.
- Tail latency, shuffle cost are not representative on your laptop.
- To measure them, we need bare-metal with a real fabric.

CloudLab: Scientific infrastructure for research

- Goal: facilitate repeatable research (same hardware + software), so results aren't artifacts of a noisy shared cloud.

- Why we use it for this Praktikum (Labs 2 and 3).
  - Isolated environments
  - Easy reset to a known-clean state → consistent starting point.
  - Root access on every node → you can change the kernel, install anything, touch the hardware.
  - **Network testbed** → multi-node labs work the way distributed systems actually behave.

CloudLab: Resource Allocation

Policy: You only hold nodes while you use them

Workflow:
– START an experiment when you're ready to work.
– TERMINATE it the moment you're done.

We have the following reservation:

- 8 × c6525-25g nodes (Utah cluster) reserved during the assignment

Launching an experiment in that window means that the reservation is used automatically.

- If 8 nodes already in use, no guarantee of success.

Our reservation

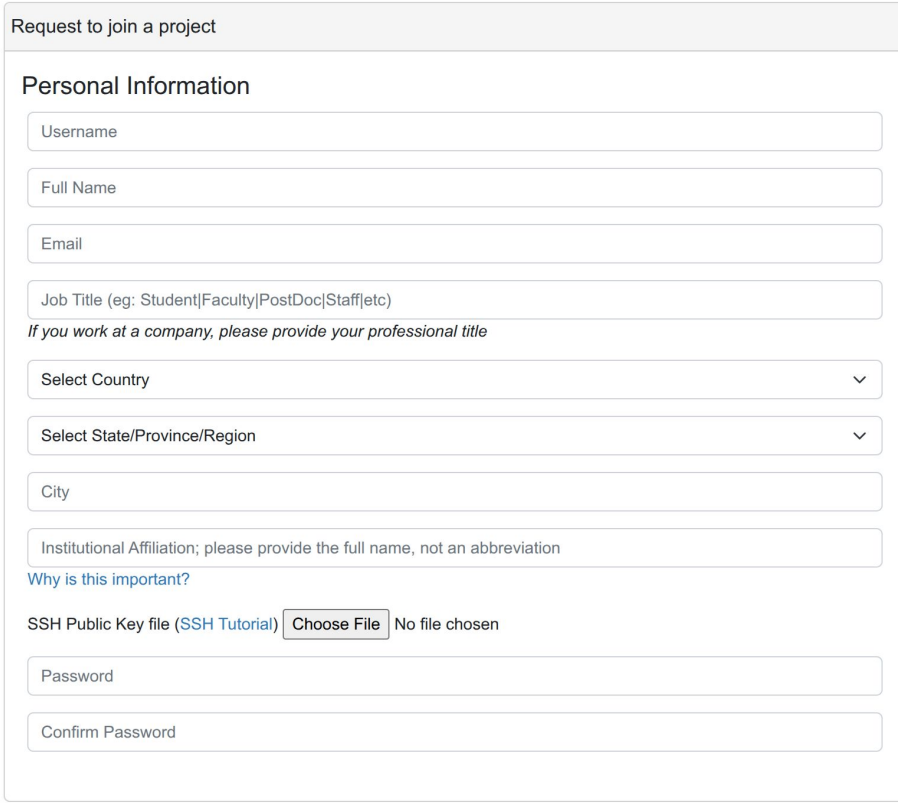
8 × c6525-25g nodes (Utah cluster)

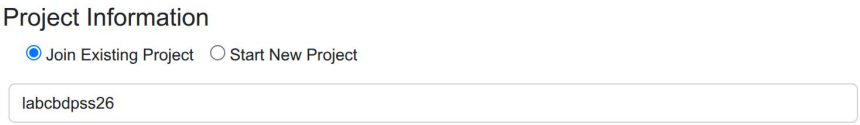
- CPU: AMD EPYC 7402P, 24 cores / 48 threads @ 2.8 GHz (single socket)
- RAM: 128 GB ECC DDR4 (8 × 16 GB)
- Storage: 1 × 480 GB SATA SSD (boot) + 1 × 1.6 TB NVMe (scratch)
- Network:
  - Experiment fabric: 2 × Mellanox ConnectX-5 25 GbE (the fast one, on the 10.10.x.x subnet)
  - Control fabric: 1 GbE (public hostname, Internet-routable)

Your data disappears when the experiment ends

Rule of thumb:
– Code lives in git (push often).
– Results / logs you care about → scp off the nodes before you terminate.

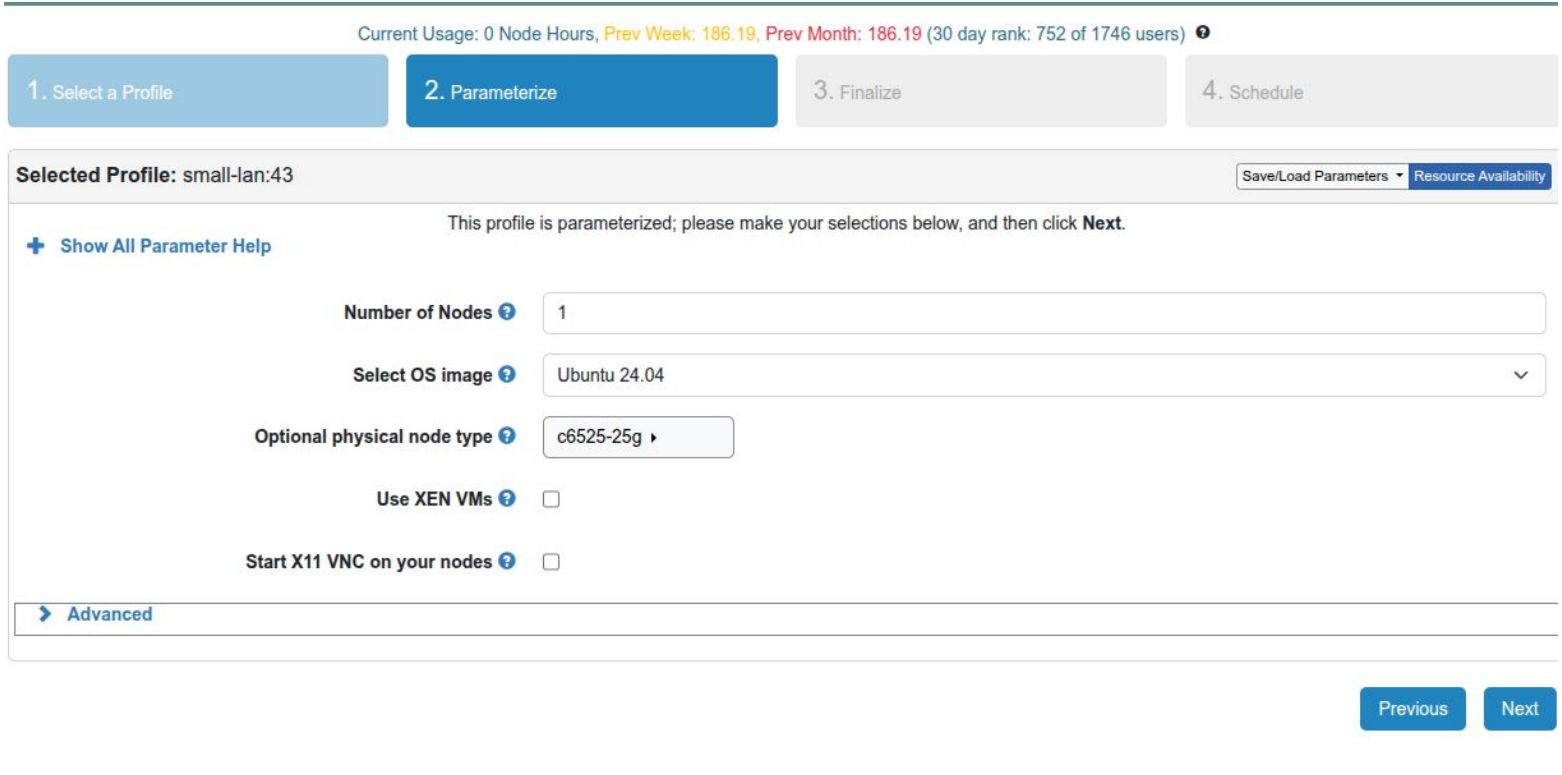
Joining the course project





- Go to https://www.cloudlab.us/
- Click on Sign Up
- Add your personal information
- Provide an SSH public key.
- Choose “Join Existing Project”
  - labcbdpss26
- I will approve you asap.

Reserving an Experiment



Programming DOs and DON'Ts

Do

- Use the 10.10.x.y experiment-net IPs for all inter-node traffic.
- Use `geni-get key` for cross-node SSH (don't copy private keys around).
- Stash anything worth keeping under `/proj/CBDP-SS26/` or back on your laptop before terminating.
- Extend the experiment if you need longer than the default.

Don't

- Don't bind services to `0.0.0.0` and push heavy traffic, as it exposes you on the control net (manual §11.2.1).
- Don't use the full DNS names (nodeN.expname.CBDP-SS26.utah.cloudlab.us) in your code, as they resolve to the control net.
- Don't use `/proj/…/` as your shuffle directory. It's NFS over 1 GbE and every worker hammers it.
- Don't leave nodes idle; release them when you're done.

Lab 2 Kickoff

Lab 2 Kickoff: Shared-Nothing Aggregation

We already worked on shared everything (single node), first assignment.

Goal: compute the global top-25 (PULocationID, DOLocationID) pairs on the NYC TLC HVFHS dataset, this time without going through the coordinator.

- Lab 1 sent every partial back to the coordinator for merging.
- Now the key includes more distinct pairs, which would make the coordinator the bottleneck.

Two-phase shuffle, the fix

- **Phase 1 (map+shuffle):**
  - Workers fetch partitions, build keys, partition them into R shards, write one partial per shard.
- **Phase 2 (merge):**
  - each worker owns a subset of shard IDs, reads every contributing worker's partial for those shards, produces a local top-K, sends it to the coordinator for final merge

This is literally MapReduce

MapReduce	Lab 2
Input splits → map tasks	Partition URLs → map_phase (**TODO 4**)
Partitioner hash(key) mod R	Partitioner::shard_of (**TODO 6**)
Combiner (pre-aggregate per map)	Per-shard hashmap inside map_phase
Intermediate files on local disk	Storage::write_shard per shard
Shuffle: reducers pull their partition	Storage::read_shard (**TODOs 7, 8**)
Reduce: merge one partition's data	merge_shard (**TODO 5**)
Driver collects final outputs	merge_global_topk (**TODO 3**)

The storage abstraction

LocalDirStorage	Peer Exchange
Writes to shared directory visible to all	Writes to worker-local disk only
Reader lists <base>/shard_N/*.dat (TODO 6)	Reader pulls via TCP from every peer (TODO 7)
Needs a shared filesystem (NFS, GPFS)	Needs only point-to-point network
Shared-storage MPP	**Shared-nothing MPP**

Lab 2 Kickoff: Shared-Nothing Aggregation

The 3 knobs to test

(1) **Input locality**
Vary input source.

- Course HTTP (https://db.in.tum.de/..., 1 GbE)
- vs cloudlab SSD (http://node0-10.10.../, 25 GbE)
- vs shared-nothing (file:///tmp/nyc/..., local NVMe)

(2) **Shared-filesystem vs peer shuffle.** Compare the two shuffle modes.

(3) **Partitioner and skew.** Compare and identify the more suitable partitioning strategy.

Assignment 2

Deadline: Session 4 **(05.05.2026), 2 weeks.**

Next session (28.04):

- Oral discussion on Assignment 1
- Q&A on Assignment 2 - make sure you have started working on assignment and have used CloudLab
- Further material:
  - Intro to Distributed Databases (CMU Intro to Database Systems)
  - MapReduce Paper
  - Snowflake Paper
  - The Tail at Scale

Questions

Thank you for your attention!