Big Data Computing

Master's of Science in Computer Science

Sapienza University of Rome

2026/2027

Gabriele Tolomei

tolomei@di.uniroma1.it

Module 1: Big Data Infrastructure

Lecture 1.2: The MapReduce Execution Model

Philosophy of MapReduce

> *"The biggest difficulty in distributed computing is not the algorithm — it's making the algorithm survive the cluster."*
> *— Folklore of distributed computing*

MapReduce does not invent new algorithms. It invents a way to run very old, very simple ones — map and reduce — safely across thousands of unreliable machines.

Lecture Roadmap

This lecture builds the first computation layer on top of the storage foundation from Lecture 1.1.

Specifically, we will study:

1. **The MapReduce Abstraction** — map and reduce as distributed primitives
2. **Execution Architecture** — how a job becomes tasks on a cluster
3. **Shuffle, Sort & Reduce** — the most expensive phase, in detail
4. **Fault Tolerance & Stragglers** — surviving failure and slowness
5. **Synthesis** — strengths, limits, and what comes next

> *Let's turn distributed storage into distributed computation.*

Why This Matters: From Storage to Computation

Lecture 1.1 gave us petabyte-scale storage (GFS/HDFS), but storage alone does not compute anything:

- Data sits in blocks, replicated across thousands of machines — but someone still has to **process** it.
- Writing custom distributed code for every analytics task is slow, error-prone, and hard to make fault-tolerant.
- **MapReduce** is a programming model that lets a programmer write *sequential-looking* code, while the underlying system handles parallelism, scheduling, and failure.

Today's question: what is the minimal abstraction that makes this possible?

§ 1 — The MapReduce Abstraction

The Problem: Processing Petabytes Without Writing Distributed Code by Hand

Before MapReduce, every large-scale data processing job required re-solving the same hard problems:

- How do I split work across machines?
- How do I handle a machine crashing mid-job?
- How do I collect and combine partial results?

*MapReduce's insight: these problems are the same for almost every batch job — so solve them **once**, in a reusable framework, and let programmers focus only on their specific logic.*

Functional Programming Roots: Map and Reduce

MapReduce borrows two primitives directly from functional programming:

- **map(f, list):** apply function `f` independently to every element of a list, producing a new list.
- **reduce(g, list):** combine elements of a list pairwise using function `g`, producing a single aggregated result.

Crucially, `map` over independent elements has no inherent ordering or dependency — which is exactly what makes it parallelizable across machines.

The MapReduce Programming Model: User's Perspective

From the programmer's point of view, writing a MapReduce job means implementing just **two functions**:

```
map(key, value) → list(intermediate_key, intermediate_value)
reduce(intermediate_key, list(intermediate_value)) → list(output_value)
```

- The programmer never writes scheduling, networking, or failure-handling code.
- The framework (not the user) is responsible for distributing `map` and `reduce` calls across the cluster.

Map Function: Signature and Semantics

The **Map** function processes one input record at a time, independently:

- Takes a single (key, value) pair from the input.
- Emits zero, one, or many intermediate (key, value) pairs.
- Has **no visibility** into other input records — it cannot depend on state from other Map calls.

This statelessness is what allows the framework to run millions of Map calls in parallel, in any order, on any machine.

Reduce Function: Signature and Semantics

The **Reduce** function processes all values associated with **one intermediate key**, after they have been grouped together:

- Takes an intermediate key and the full list of values emitted for that key by all Map calls.
- Emits the final output value(s) for that key.
- Different keys are processed independently — different Reduce calls can also run in parallel.

Worked Example Setup: Word Count

The canonical MapReduce example: count occurrences of every word across a massive collection of documents.

- **Input:** a large set of text documents (e.g., stored as HDFS blocks from Lecture 1.1).
- **Desired output:** for every distinct word, the total number of times it appears across all documents.
- This simple problem illustrates every mechanic MapReduce provides — and nothing more.

Word Count: The Map Step

```
map(document_id, document_text):
    for each word w in document_text:
        emit(w, 1)
```

- Each Map call processes one document.
- For every word encountered, it emits the pair `(word, 1)` — "I saw this word once."
- No counting happens yet — Map only **observes and emits**.

Word Count: The Reduce Step

```
reduce(word, list_of_counts):
    total = sum(list_of_counts)
    emit(word, total)
```

- Each Reduce call receives one word and the full list of `1`s emitted for it across **all** documents.
- It simply sums them, producing the final total count for that word.
- The framework guarantees all `1`s for a given word arrive at the **same** Reduce call.

Second Example: Inverted Index Sketch

To see the abstraction generalize, consider building a search engine's **inverted index** (word → list of documents containing it):

```
map(document_id, document_text):
    for each word w in document_text:
        emit(w, document_id)

reduce(word, list_of_document_ids):
    emit(word, unique_sorted(list_of_document_ids))
```

Same Map/Reduce structure as word count — only the emitted values and the aggregation logic change. This is the abstraction's real power.

Why This Abstraction Hides Distribution From the Programmer

Notice what the programmer **never** had to specify, in either example:

- Which machine runs which Map or Reduce call.
- How intermediate data physically moves between machines.
- What happens if a machine crashes mid-computation.

All of this is handled entirely by the MapReduce runtime — covered in the rest of this lecture.

Determinism: Why Map/Reduce Functions Must Be Side-Effect-Free

MapReduce's fault-tolerance guarantees (Section 4) depend on a critical assumption:

- Map and Reduce functions must be **pure**: given the same input, they always produce the same output.
- They must have **no side effects** (no writing to external state, no relying on external mutable state).

If a task fails and is simply re-executed elsewhere, determinism guarantees the re-run produces identical results — this is what makes "just retry it" a valid fault-tolerance strategy.

✅ Checkpoint 1 — The MapReduce Model

Let's test our basic understanding:

1. What are the signatures of the `map` and `reduce` functions in MapReduce?
2. In the word count example, what does each Map call emit, and why is no counting done yet?
3. Why must Map and Reduce functions be deterministic and side-effect-free?
4. What distributed-systems problems does the MapReduce abstraction hide from the programmer?

Write down your thoughts before moving to Section 2.

§ 2 — Execution Architecture

From Program to Cluster Job: The Execution Overview

Turning the user's `map` / `reduce` functions into a running cluster job involves several coordinated steps:

1. Split input data into independent pieces.
2. Schedule a Map task for each piece, across available machines.
3. Partition and shuffle intermediate output to the right Reduce tasks.
4. Schedule Reduce tasks, each handling a subset of intermediate keys.
5. Collect final output.

The rest of this section walks through each of these steps in detail.

The Master/JobTracker Coordinator Role

A central coordinator (often called the **Master** or **JobTracker**) manages the overall job execution:

- Splits the input into chunks suitable for Map tasks.
- Assigns Map and Reduce tasks to available worker machines.
- Tracks task progress and re-assigns tasks on failure (Section 4).
- Conceptually plays a role analogous to the GFS Master/HDFS NameNode — but for **computation**, not storage.

Input → Map Tasks: Linking to Lecture 1.1's Blocks

MapReduce input typically comes directly from the distributed file system studied in Lecture 1.1:

- Input data is already split into **blocks** (HDFS) or **chunks** (GFS).
- By default, **one Map task is created per input block** — the natural unit of parallel work is inherited directly from the storage layer.
- This is not a coincidence: MapReduce was designed hand-in-hand with GFS.

Data Locality in Task Scheduling

Recall the data locality principle from Lecture 1.1, Section 4: *move computation to data, not data to computation.*

- The Master schedules each Map task on (or near) the machine that already holds the corresponding input block.
- This minimizes network traffic during the Map phase — by far the largest input volume in the entire job.
- When a perfectly local machine is unavailable, the Master falls back to the same rack, then to any available machine.

Number of Map Tasks vs. Number of Machines

The number of Map tasks is typically **much larger** than the number of physical machines:

- One Map task per input block, and large jobs may have thousands of blocks.
- A cluster with, say, 100 machines will run these thousands of Map tasks in many successive **waves**.
- This over-decomposition (many small tasks per machine) helps with load balancing and makes straggler mitigation (Section 4) more effective.

Intermediate Output: Where Map Results Go

Each Map task's output does **not** go directly to the final destination:

- Map output is written to **local disk** on the machine that ran the Map task, not to the distributed file system.
- This intermediate data is only ever needed temporarily, by the Reduce phase — writing it to local disk avoids unnecessary replication overhead.

Partitioning Intermediate Keys for Reduce

Before Reduce tasks can run, intermediate (key, value) pairs must be grouped by key and routed to the correct Reduce task:

- A **partitioning function** maps each intermediate key to one of the Reduce tasks.
- All pairs with the same key must be routed to the **same** Reduce task, regardless of which Map task produced them.

The Default Partitioning Function: Hash Partitioning

The most common partitioning strategy uses a hash function:

```
partition(key) = hash(key) mod num_reduce_tasks
```

- Distributes keys roughly evenly across Reduce tasks, assuming a reasonably uniform hash.
- Deterministic: the same key always maps to the same Reduce task, from any Map task.
- Custom partitioning functions can be supplied for specialized load-balancing needs.

Number of Reduce Tasks: A User-Controlled Parameter

Unlike the number of Map tasks (driven by input block count), the number of Reduce tasks is typically **chosen by the user**:

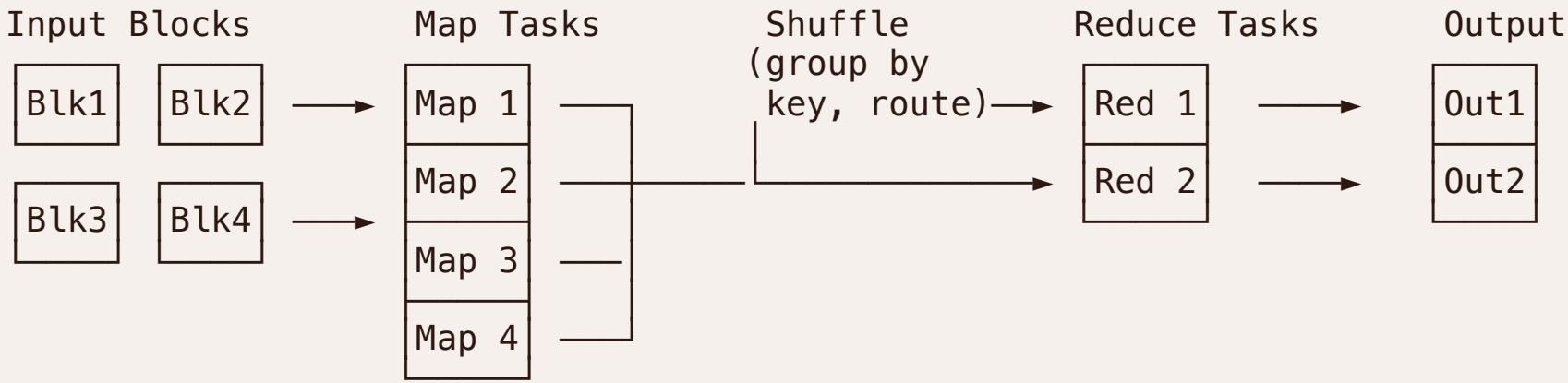
- More Reduce tasks → finer-grained parallelism, but more output files and shuffle overhead.
- Fewer Reduce tasks → simpler output, but less parallelism and potentially larger per-task workloads.

Trade-offs: Too Few vs. Too Many Reduce Tasks

Choice	Consequence
Too few Reduce tasks	Each handles a huge share of keys; poor parallelism; risk of a single overloaded "hot" reducer
Too many Reduce tasks	More output files to manage; higher shuffle coordination overhead; diminishing returns past available machines

A common rule of thumb: choose a number of Reduce tasks on the order of the number of available worker machines, sometimes a small multiple of it.

Execution Overview Diagram



One Map task per input block; intermediate output is shuffled and routed by key; Reduce tasks each produce one output file.

Combiners: Local Pre-Aggregation Before Shuffle

A **combiner** is an optional optimization: a mini-reduce step run **locally** on each Map task's output, before shuffling:

- Same aggregation logic as the Reduce function (often the literal same function, when associative/commutative).
- Reduces the volume of intermediate data that must be transferred across the network during shuffle.
- *Example:* in word count, a combiner can sum `(word, 1)` pairs locally per document **before** sending them across the network.

Sharded Output Files

Just as MapReduce input arrives as multiple blocks, its output is also written as multiple **sharded files**, not one monolithic file:

- Each Reduce task writes its own output file, conventionally named something like `part-r-00000`, `part-r-00001`, etc.
- This mirrors the distributed storage model from Lecture 1.1: large datasets are naturally represented as a *set* of files, not a single file.
- Downstream jobs can read these sharded outputs directly as their own distributed input.

✅ Checkpoint 2 — Execution Architecture

Let's test our understanding:

1. Why does MapReduce typically create one Map task per input block?
2. Explain how data locality from Lecture 1.1 directly informs Map task scheduling.
3. What problem does a combiner solve, and where in the pipeline does it run?
4. What trade-off governs the choice of the number of Reduce tasks?

Discuss with your colleague or write down your answers.

§ 3 — Shuffle, Sort & the Reduce Phase

The Shuffle Phase: What Actually Happens

Shuffle is the process of moving intermediate Map output to the correct Reduce task across the network:

- Every Reduce task must receive **all** intermediate pairs matching its assigned keys — regardless of which machine produced them.
- This requires data to potentially travel from **every** Map task to **every** Reduce task — an all-to-all communication pattern.

Each Mapper Partitions Its Own Output

Before any network transfer happens, every individual Map task already organizes its own local output:

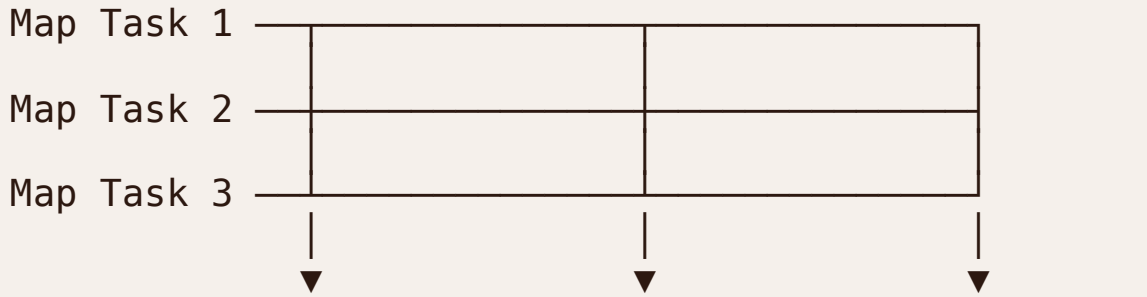
- As a Map task emits intermediate pairs, it immediately applies the partitioning function to each key.
- Locally, the Map task's output is split into **R** separate regions on local disk (one per Reduce task), already sorted or bucketed by destination.
- Reduce tasks later pull only the regions assigned to them from each Map task's local output.

Why Shuffle Is Expensive: All-to-All Data Movement

The shuffle phase is widely regarded as the most expensive part of a MapReduce job:

- With **M** Map tasks and **R** Reduce tasks, the shuffle pattern can involve up to **M × R** distinct data transfers.
- Unlike the Map phase (which benefits from data locality), shuffle data must generally cross the network — there is no way to make every Reduce task "local" to every Map task simultaneously.
- Disk I/O is also significant: data is written to local disk by Map tasks, then read again to be transferred.

Shuffle Diagram



Every Map task potentially sends data to every Reduce task — the defining "all-to-all" shape of the shuffle.

Network Cost: All-to-All Transfer Volume

Consider a job with **M** Map tasks, **R** Reduce tasks, and total intermediate data volume **D**:

$$\text{Worst-case per-link transfer} \approx \frac{D}{M \times R}$$

- Total network traffic across the cluster during shuffle is on the order of **D** (the full intermediate dataset size) — *not* reduced by parallelism, only **distributed** across many smaller transfers.
- This is fundamentally different from the Map phase, where data locality lets most reads avoid the network entirely.

Sorting Intermediate Data by Key

After receiving its assigned partitions from all Map tasks, each Reduce task **sorts** its received data by intermediate key:

- Sorting ensures all values for the same key end up **adjacent** to one another.
- The framework typically performs this sort as part of merging data received from multiple Map tasks (a merge-sort-like process).

Why Sorting Simplifies the Reduce Implementation

Sorting before Reduce is a deliberate design choice that simplifies the programmer's job:

- The Reduce function can assume all values for a given key arrive together, in one contiguous group.
- The programmer does not need to implement their own grouping/hashing logic inside `reduce` — the framework guarantees grouping via sorting.
- This is the same "do the hard distributed-systems work once, in the framework" philosophy seen throughout this lecture.

The Reduce Phase: Consuming Sorted Groups

With data sorted and grouped by key, the Reduce task processes one group at a time:

```
for each (key, sorted_list_of_values) in sorted_input:
    call reduce(key, sorted_list_of_values)
    emit result to output file
```

- Each call to the user's `reduce` function handles exactly one key and its full value list.
- Memory usage can be a concern if a single key has an extremely large number of values (data skew) — a limitation revisited in Section 5.

Reduce Output: Writing Final Results

Once a Reduce task finishes processing its assigned keys, it writes its output:

- Output is written back to the **distributed file system** (HDFS/GFS) — unlike intermediate Map output, which stayed on local disk.
- Each Reduce task produces its own output shard (as introduced in Section 2), which is then replicated like any other file in the DFS.

The Shuffle Bottleneck: Network and Disk I/O at Scale

Bringing together the cost factors discussed in this section, shuffle is expensive for two compounding reasons:

- **Network:** all-to-all transfer pattern moves the full intermediate dataset volume across the cluster network.
- **Disk:** intermediate data is written to local disk by Map tasks, then read back for transfer, then often re-written/sorted on the Reduce side.

For many real-world MapReduce jobs, shuffle — not the Map or Reduce computation itself — dominates total job runtime.

Mitigations: Combiners Revisited, Compression

Two practical techniques reduce the impact of the shuffle bottleneck:

- **Combiners** (Section 2): reduce intermediate data volume *before* it ever needs to be shuffled.
- **Compression:** intermediate data is frequently compressed before network transfer, trading CPU time for reduced network and disk I/O — usually a favorable trade at cluster scale.

Worked Example: Word Count End-to-End Trace (Part 1)

Tracing word count through the full pipeline, for two tiny input documents:

```
Doc A: "the cat sat"        Doc B: "the dog ran"
```

Map phase:

```
Map(DocA) → (the,1) (cat,1) (sat,1)
Map(DocB) → (the,1) (dog,1) (ran,1)
```

Partitioning (assume 2 Reduce tasks, hash-based): pairs are split by key hash into Reduce 1's region and Reduce 2's region on each Map task's local disk.

Worked Example: Word Count End-to-End Trace (Part 2)

Shuffle + Sort (assume `the`, `cat` → Reduce 1; `sat`, `dog`, `ran` → Reduce 2):

```
Reduce 1 receives, sorted: (cat,[1]) (the,[1,1])
Reduce 2 receives, sorted: (dog,[1]) (ran,[1]) (sat,[1])
```

Reduce phase:

```
Reduce 1 emits: (cat,1) (the,2)
Reduce 2 emits: (dog,1) (ran,1) (sat,1)
```

Notice "the" appeared in both documents, but both copies were routed to the same Reduce task — exactly as the partitioning function guarantees.

✅ Checkpoint 3 — Shuffle, Sort, Reduce

Let's test our understanding:

1. Why is the shuffle phase described as an "all-to-all" communication pattern?
2. Why does sorting intermediate data simplify the Reduce function's implementation?
3. Why is shuffle, rather than Map or Reduce computation, often the dominant cost of a MapReduce job?
4. In the worked word count trace, why must both occurrences of "the" be routed to the same Reduce task?

Review your answers before moving on.

§ 4 — Fault Tolerance & Stragglers

Failure During Computation: A New Dimension Beyond Storage Failures

Lecture 1.1 covered failures of **storage** (disks, chunkservers, the Master). MapReduce introduces a second dimension: failures during **active computation**:

- A worker machine can crash while running a Map or Reduce task.
- A task can hang or run abnormally slowly without technically "failing" (the straggler problem, covered shortly).
- The Master coordinating the job can itself fail.

Worker Failure: Detecting a Dead Map/Reduce Task

The Master detects worker failure using the same fundamental mechanism as the storage layer:

- Workers send periodic **heartbeats** to the Master, reporting task progress.
- If a worker misses heartbeats beyond a timeout threshold, the Master marks it as **failed**.
- Any in-progress (and even already-completed, in some cases — see below) tasks on that worker must be handled.

Re-executing Failed Map Tasks

When a worker running a Map task fails:

- The Master simply **re-schedules** that Map task on a different, healthy worker.
- This is safe even if the original Map task had already partially or fully completed — its output lived on the failed machine's **local disk**, which is now unreachable.
- Because Map functions are deterministic (Section 1), re-running produces identical output.

Re-executing Failed Reduce Tasks

Recovering a failed Reduce task is similar, but only requires retracing the **shuffle** step, not the original Map computation:

- The Master re-schedules the Reduce task on a different worker.
- The new worker re-fetches its assigned intermediate data from the (still-available) Map task outputs.
- The Map outputs themselves do not need to be recomputed — only re-fetched.

Why Re-execution Is Safe

Simply "trying again" as the fault-tolerance strategy relies entirely on the determinism property introduced in Section 1:

- If Map/Reduce functions were not deterministic, re-execution could silently produce **different** results than the original (now-lost) run.
- Because they are pure functions of their input, re-execution after failure is functionally indistinguishable from the task having succeeded on the first attempt.

This is why the determinism requirement is non-negotiable in the MapReduce model.

Idempotency and Output Safety Under Speculation

A subtlety arises once we allow tasks to be re-executed (due to failure) *or* run speculatively (Section ahead) **while a previous attempt might still be running**:

- The framework must ensure only **one** attempt's output is ever considered "final" for a given task, even if multiple attempts run concurrently.
- This is typically handled via **atomic commit**: a task attempt writes to a temporary location, and only an atomic rename/commit operation, coordinated by the Master, makes one attempt's output the official result.

Master Failure Handling

The Master coordinating the MapReduce job is itself a potential point of failure:

- In the original MapReduce design, Master failure is rare (a single job-tracking process) and is typically handled by **restarting the entire job** from scratch.
- This is a deliberate simplicity trade-off: unlike the GFS/HDFS metadata node (which must survive for the cluster's entire lifetime), a MapReduce Master only needs to survive for the duration of one job.

The Straggler Problem: Slow Tasks, Not Failed Ones

A **straggler** is a task that is still running, has not crashed, but is taking far longer than other equivalent tasks:

- Stragglers do not trigger the heartbeat-timeout failure-detection mechanism — the worker is alive and responsive.
- Without mitigation, the entire job's completion time is bound by the **slowest single task**, even if 999 out of 1000 tasks finished quickly.

Causes of Stragglers

Several real-world conditions can cause a healthy-but-slow task:

- **Hardware degradation:** a disk or CPU performing below its rated specification without outright failing.
- **Resource contention:** other jobs or processes competing for CPU, disk, or network bandwidth on the same machine.
- **Data skew:** a Reduce task assigned an unusually large number of values for one key, taking disproportionately long to process.

Speculative Execution: Running Backup Copies of Slow Tasks

MapReduce's solution to the straggler problem is **speculative execution**:

- Near the end of a job, the Master identifies tasks that are running much slower than the average for similar tasks.
- It launches a **backup copy** of that task on a different, idle worker.
- Whichever copy (original or backup) finishes **first** is taken as the result; the other is discarded.

Worked Numeric Example: Speculative Execution's Effect on Job Time

Suppose a job has 1,000 Map tasks, each normally taking ~10 seconds, but one straggling task takes 200 seconds due to hardware degradation:

Scenario	Job Completion Time
Without speculation	~200 seconds (bound by the straggler)
With speculation (backup launched, finishes in ~10s)	~10–20 seconds (bound by the backup copy)

A single straggler can dominate total job time by an order of magnitude — speculative execution directly targets this failure mode.

Speculative Execution Trade-offs

Speculative execution is not free — it involves a direct trade-off:

- **Benefit:** can dramatically reduce overall job completion time when a small number of tasks are anomalously slow.
- **Cost:** wastes cluster resources on backup copies that are ultimately discarded if the original finishes first (or vice versa).
- **Tuning consideration:** speculation is typically only triggered for tasks significantly slower than their peers, and only when spare cluster capacity exists.

✅ Checkpoint 4 — Fault Tolerance & Stragglers

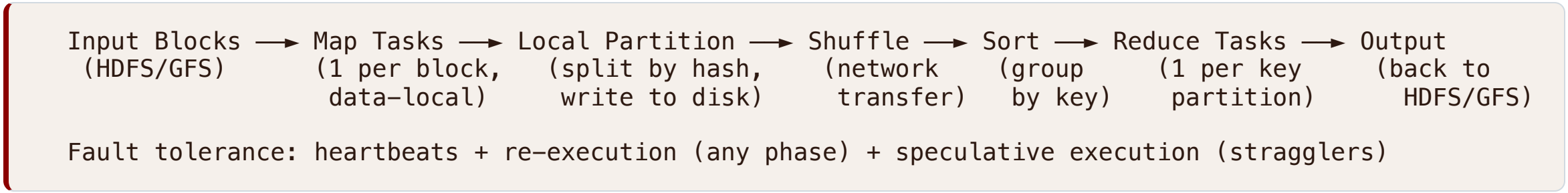
Let's test our understanding:

1. Why is re-executing a failed Map task on a different machine a safe recovery strategy?
2. What is the key difference between recovering from a failed Map task vs. a failed Reduce task?
3. Why doesn't heartbeat-based failure detection catch stragglers?
4. Explain how speculative execution works, and what its main cost is.

Consolidate your answers before proceeding.

§ 5 — Synthesis & Wrap-up

MapReduce Recap Diagram (Full Pipeline)



Every box in this diagram was built, mechanism by mechanism, across Sections 1–4.

Storage vs. Compute Layer: What Changed, What Stayed the Same

Comparing this lecture to Lecture 1.1's storage foundation:

Aspect	Storage Layer (Lec. 1.1)	Compute Layer (Lec. 1.2)
Coordinator	Master / NameNode	Master / JobTracker
Worker unit	Chunkserver / DataNode	Map / Reduce task
Failure detection	Heartbeats	Heartbeats
Recovery strategy	Re-replication	Re-execution
New concept introduced	—	Speculative execution (stragglers ≠ failures)

Strengths of the MapReduce Model

Reflecting on why this model succeeded at the scale it did:

- **Simplicity:** programmers write two sequential-looking functions, not distributed systems code.
- **Fault tolerance built-in:** determinism + re-execution handles failures transparently.
- **Scales with data locality:** inherited directly from the GFS/HDFS storage layer.
- **General enough** to express a surprisingly wide range of batch analytics tasks.

Limitations of MapReduce

These same design choices introduce real limitations, motivating later course material:

- **Disk-heavy:** intermediate data written to disk between every Map and Reduce step — costly for iterative algorithms (e.g., many machine learning training loops).
- **Rigid two-stage structure:** complex pipelines require chaining multiple MapReduce jobs, each paying the full shuffle cost again.
- **High latency:** the model favors batch throughput, not interactive or low-latency queries.

These exact limitations are what motivate Apache Spark's in-memory DAG execution model — the topic of Lecture 1.3.

Discussion Prompt

Before the summary, consider this scenario:

> *A data scientist needs to run 10 successive MapReduce jobs, where each job's output feeds directly into the next job's input (e.g., iterative graph algorithms).*

Discussion: Based on what you now know about where intermediate data lives and how shuffle cost accumulates, what would you expect to be the dominant cost of this pipeline — and why might an in-memory execution model help?

Hold this question in mind as we move into Lecture 1.3.

Summary: Key Concepts of This Lecture

1. **The abstraction:** map and reduce as simple, deterministic, parallelizable primitives.
2. **The execution model:** Map tasks scheduled with data locality; intermediate data partitioned, shuffled, sorted, and reduced.
3. **The cost center:** shuffle is the dominant, network- and disk-bound cost of most MapReduce jobs.
4. **Fault tolerance:** re-execution handles failures; speculative execution handles stragglers — two distinct problems, two distinct mechanisms.

Key Terms Glossary

Term	Meaning
Map task	Processes one input split, emits intermediate (key, value) pairs
Reduce task	Processes all values for one partition of intermediate keys
Shuffle	Network transfer of intermediate data from Map to Reduce tasks
Combiner	Local pre-aggregation before shuffle, reduces data volume
Straggler	A task running abnormally slowly without having failed
Speculative execution	Running a backup copy of a slow task on another worker

Preview: Lecture 1.3 — Apache Spark & Dataflow Graph Architectures

In the next lecture, we address MapReduce's core limitations directly:

- **Beyond two stages:** generalizing to arbitrary Directed Acyclic Graphs (DAGs) of operations.
- **Resilient Distributed Datasets (RDDs):** in-memory, fault-tolerant data abstractions.
- **Lazy evaluation:** building and optimizing an execution plan before running it.

> *MapReduce taught the cluster how to survive failure. Spark teaches it how to stay fast. See you in Lecture 1.3!*

Lecture 1.2 Complete

Recommended Reading

- Dean, J., & Ghemawat, S. (2004). *MapReduce: Simplified Data Processing on Large Clusters*. OSDI / Comm. of the ACM, 51(1).
- Course reference materials on MapReduce execution internals

Contact & Q&A

- **Lecturer:** Gabriele Tolomei
- **Email:** tolomei@di.uniroma1.it

- **Office Hours:** Drop me a message, and we'll arrange a time to meet, either in person or online! 🙂

Thank you for your attention!