MapReduce: Distributed Computation

COMS4113: Fundamentals of Distributed Systems

Instructor: Hubertus Franke

Columbia University

https://frankeh.github.io/cu-ds1-2026-fall

Outline

- **Motivation: large-scale analytics before MapReduce**
  - Two starting points: SQL and a hand-rolled distributed program
- **The MapReduce programming model**
  - Map, shuffle/sort, and Reduce
  - Worked example: word count (and word frequency by chaining)
- **The runtime system**
  - Architecture, data locality, fault tolerance, stragglers
- **MapReduce today (2026)**
  - Spark, Flink, Beam/Dataflow, and where the ideas live now
- *To read: the original OSDI 2004 paper; Spark RDDs (NSDI 2012)*

Large-scale analytics (circa 2005)

- The problem: run simple computations over enormous datasets spread across thousands of machines.
- Compute the frequency of words in a large corpus of documents.
- Count how many times users clicked on each of a large set of ads.
- PageRank: rank a web page by the importance of the pages linking to it.
- Build and maintain a search index over the whole web.
- *Each is easy on one machine. The hard part is doing it at scale, in parallel, and surviving the failures that are routine at thousands of machines.*

Option 1: SQL

- Before MapReduce, analytics were mostly done in SQL, or by hand.
- Example: count word appearances across a corpus.

```
SELECT word, COUNT(*)
  FROM (
        SELECT UNNEST(string_to_array(doc_content, ' ')) AS word
        FROM Corpus
  ) AS words
  GROUP BY word;
```

Very expressive and convenient, but in ~2005 no one knew how to scale SQL execution across thousands of commodity machines with failures. (Massively parallel SQL came later.)

Option 2: do it by hand

Count word appearances across a corpus:

Phase 1: Map locally

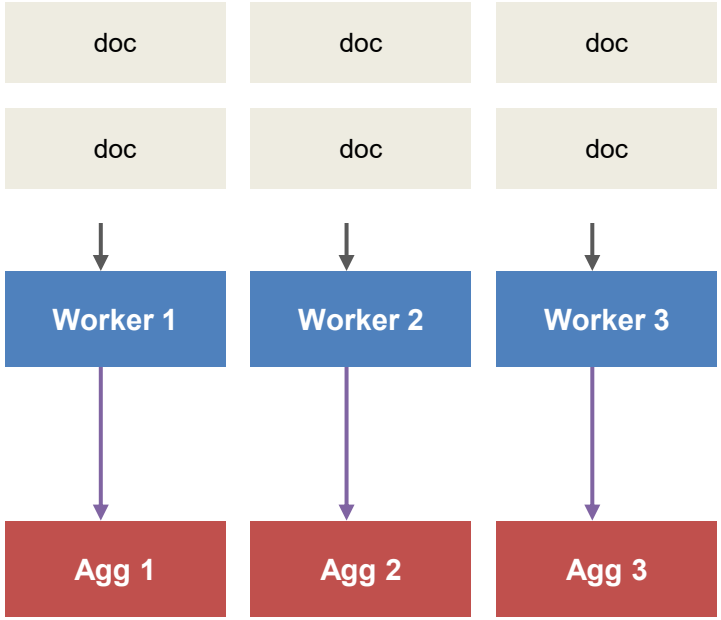
Assign documents to worker nodes; each builds a local {word: count} dictionary.

Phase 2: Aggregate

Nodes exchange partial counts so every word's counts land together.

Option (a): send everything to one node, but it can become a bottleneck.

Option (b): route each word to node hash(word) % N, so load spreads.



route each word by hash(word) % N

We have just re-derived an application-specific MapReduce, by hand.

What makes this hard at scale

- How do we generalize this pattern to other applications?
- Failures: at thousands of machines, something is always broken.
- Stragglers: one slow node can dominate the total completion time.
- Load balancing: documents vary wildly in size.
- Skew: a few words (“the”, “of”) are enormously frequent, hammering their aggregator.
- *We want to write these challenges once, in a runtime, and never again in application code.*

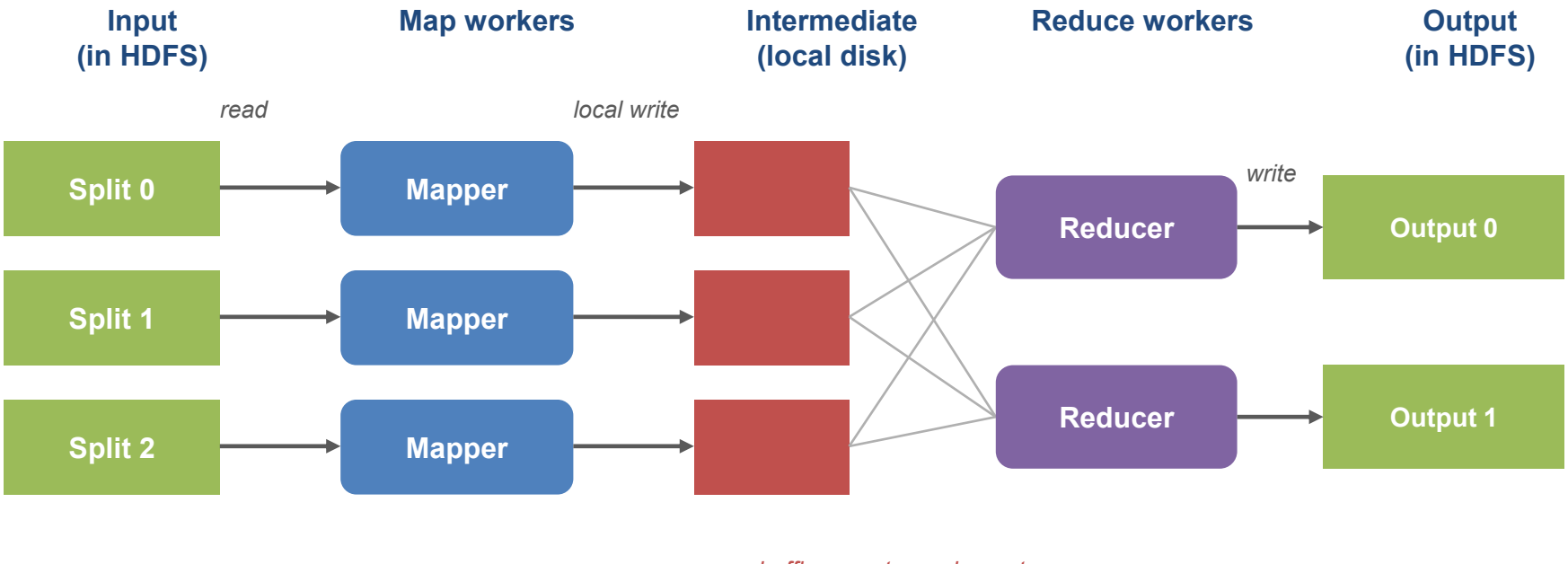
MapReduce: the programming model

- **A parallelizable programming model + a scalable, fault-tolerant runtime.**
  - Applies to a broad class of analytics; less expressive than SQL, but far easier to scale.
- **Three phases (only two written by the programmer), each intrinsically parallel:**
  - Map: process each input record independently, emit (key, value) pairs.
  - Shuffle & sort (runtime): group all values for each key: (key, [values]). “Shuffle” is Hadoop’s term; the 2004 paper just calls it grouping/sorting.
  - Reduce: aggregate the values for each key into the output.
- **The runtime handles parallelism, data movement, failures, and stragglers.**

How it works

- Input: a collection of (key, value) pairs.
- The programmer defines two functions:
  - Map(k, v) → list of (k′, v′) (zero, one, or many pairs)
  - Reduce(k′, [v′]) → output
- The execution engine:
  - Applies Map to input pairs in parallel across keys.
  - Groups and sorts the emitted pairs into (k′, [v′]).
  - Applies Reduce to each group in parallel across keys.
  - Output is the union of all Reduce outputs.

Execution workflow



Example: word count

- We have a directory containing many documents.
- Documents contain words separated by whitespace and punctuation.
- **Goal: count how many times each distinct word appears across all files.**
- *The “hello world” of MapReduce, and still the clearest way to see all three phases.*

Word count in MapReduce


```
Map(key, value):          // key: document ID; value: document text
    for each word w in value:
        emit(w, 1);
```

```
Reduce(key, values):      // key: a word; values: a list of counts
    result = 0;
    for each v in values:
        result += v;
    emit(key, result);
```

Optimization: a combiner runs Reduce's logic locally on each mapper before the shuffle. It is safe here because addition is associative and commutative (paper §4.3).

Map phase (worked example)

- Input pair (document ID, content):

(D1, "The teacher went to the store. The store was closed; the store opens in the morning. The store opens at 9am.")

Map emits one (word, 1) pair per word occurrence:

<The,1> <teacher,1> <went,1> <to,1> <the,1> <store,1> <the,1> <store,1> <was,1> <closed,1> <the,1> <store,1> <opens,1> <in,1> <the,1> <morning,1> <the,1> <store,1> <opens,1> <at,1> <9am,1>

In reality many documents run through many mappers in parallel; we show a single document so the full pipeline fits on a few slides.

Shuffle & sort phase

The runtime groups and sorts emitted pairs by key, then partitions keys across reducers:

```
<9am,1>
<at,1>
<closed,1>
<in,1>
<morning,1>
<opens,1>
<opens,1>
<store,1>
<store,1>
<store,1>
<store,1>
```

```
<teacher,1>
<the,1>
<the,1>
<the,1>
<the,1>
<the,1>
<to,1>
<went,1>
<was,1>
<The,1>
```

Reduce phase (worked example)

Reduce(key, values) runs once per unique key and sums the counts. Final output:

Reducer 1

```
<9am, 1>
<at, 1>
<closed, 1>
<in, 1>
<morning, 1>
<opens, 2>
<store, 4>
```

Reducer 2

```
<teacher, 1>
<the, 5>
<to, 1>
<went, 1>
<was, 1>
<The, 1>
```

Note: “The” and “the” are distinct keys here; real jobs normalize case in Map first.

Chaining MapReduce jobs

- The model looks restrictive, but many analytics fit, especially by chaining jobs.
- If reducers emit (key, value) pairs, those can feed another Map/Reduce job.
- Chaining supports a wide variety of analytics.
- Its weakness: each stage writes to disk between jobs. Iterative algorithms (PageRank, k-means, gradient descent) become impractically slow. This motivated Spark.

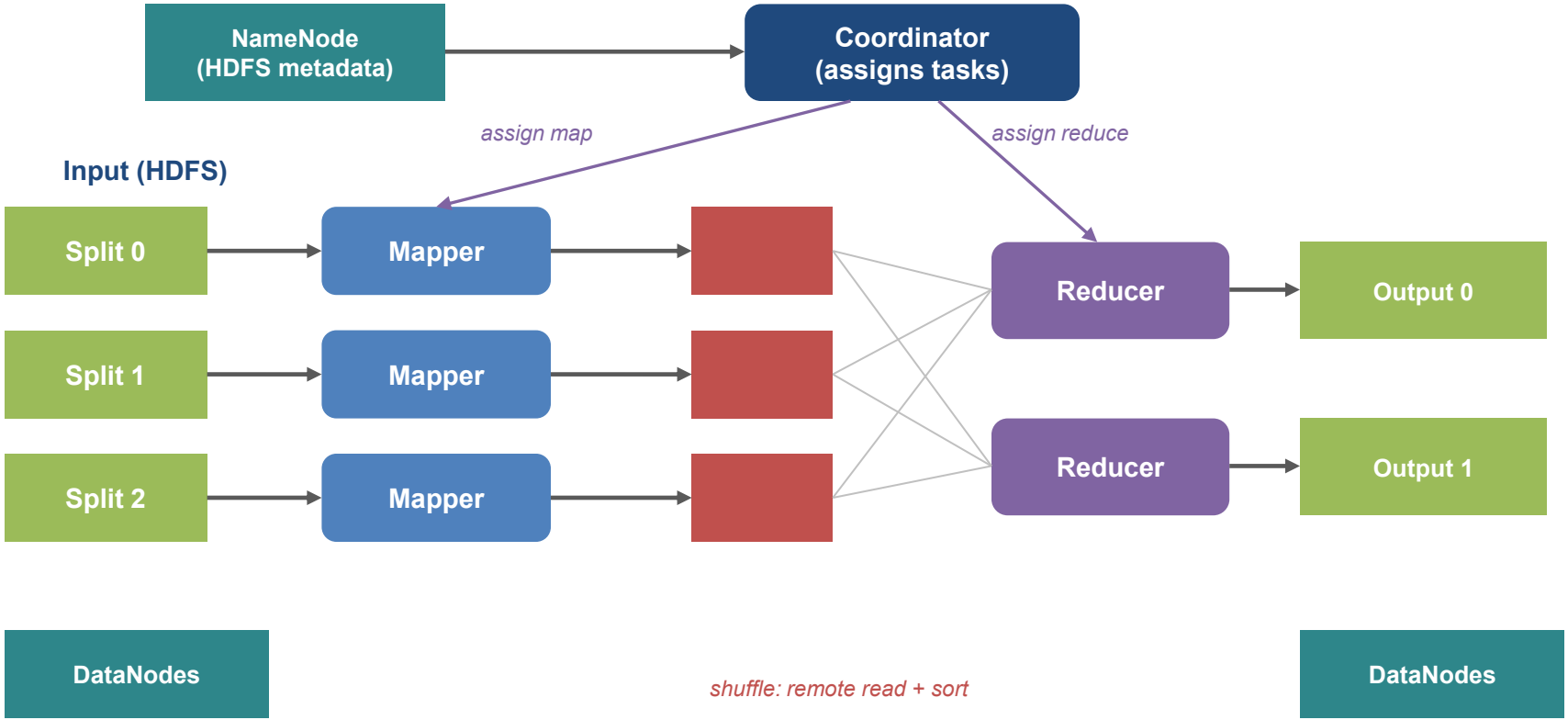
Example: word frequency

- Instead of counts, compute each word's frequency: its fraction of all words in the corpus.
- Needs a global total, which a single Map/Reduce cannot produce directly.
- **Solution: chain two jobs.**
  - **Job 1: Word count (as before)**
    - Map: documents → (word, 1). Reduce: (word, count).
  - **Job 2: Normalize**
    - Map: (word, count) → (1, (word, count)) so all pairs share one key.
    - One Reducer, two passes: pass 1 sums counts → total; pass 2 emits (word, count/total).
  - Scales fine: job 1's output is a small dictionary (one integer per distinct word).

Back to the hard parts

- Generalize to other applications? → the model + many examples in the paper.
- **Failures? Slow nodes? Load balancing? Skew?**
- The runtime hides these behind the map/reduce abstraction. Next we look at how:
  - Architecture and data locality (performance)
  - Heartbeats and re-execution (fault tolerance)
  - Backup tasks for stragglers (tail latency)

Architecture



Data locality for performance

- **Coordinator scheduling policy:**
  - Ask HDFS for the locations of each input block's replicas.
  - Schedule each map task on a machine that holds a replica, or, failing that, one on the same rack/switch (paper §3.4).
- **Effect:**
  - Most map tasks read their input from local disk.
  - The cluster avoids shipping raw input across the network, removing a key bottleneck.

Fault tolerance

- Failures are the norm, not the exception, at data-center scale.
- **Worker failure:**
  - The coordinator pings workers periodically (Hadoop later called this a heartbeat).
  - Re-execute in-progress tasks, and re-run completed map tasks too: their output sits on the failed worker's local disk. Reduce output lives in the DFS, so it need not re-run.
- **Coordinator failure:**
  - The 2004 system aborted the job (checkpointing was called easy but left unimplemented, since a single coordinator rarely fails). Hadoop later added ResourceManager high availability.
- *Robustness: a job kept making progress while groups of 80 machines dropped out for maintenance (paper §3.3); Jeff Dean's talks cite a run that lost ~1600 of 1800 machines and still finished.*

Stragglers and backup tasks

- A straggler is an unusually slow task; the slowest task can dominate total completion time.
  - Causes: contention from other jobs, a failing disk that reads slowly, bad configuration.
- **Solution: near the end of a phase, launch backup (redundant) copies of the remaining tasks; the first copy to finish wins.**
  - Disabling backup tasks made a sort job run ~44% longer (paper §5.3).
  - Hadoop calls this speculative execution.
- *This is the tail-latency insight generalized in Dean & Barroso, "The Tail at Scale" (CACM 2013): in large fan-out systems, the 99th-percentile straggler is what you must design against.*

Take-aways

- **MapReduce is a restricted programming model plus a runtime for scalable, fault-tolerant analytics.**
  - Write only Map and Reduce; the runtime handles parallelism, locality, failures, and stragglers.
- Key ideas that outlived it: auto-parallelized restricted model, re-execution for fault tolerance, backup tasks for tail latency, data locality.
  - It is a poor fit for iterative/interactive work, which motivated the successors.
- Landscape: Hadoop MapReduce is the canonical open-source implementation (now legacy). Spark and Flink are successor DAG/streaming engines that generalize the model, not MapReduce implementations.

MapReduce today (2026)

- Google itself moved on: “We don’t really use MapReduce anymore” (Urs Hölzle, Google I/O 2014), toward FlumeJava and the Dataflow model (VLDB 2015, now Apache Beam).
- **The mainstream stack:**
  - Apache Spark 4.0 (May 2025): in-memory RDD/DataFrame DAGs; batch, streaming, and ML (MLlib).
  - Apache Flink: true low-latency, exactly-once stream processing.
  - Beam / Google Cloud Dataflow: one model for batch and streaming.
  - Apache Hadoop 3.5 (2024–26): MapReduce maintained but legacy; new work runs on YARN via Spark/Flink/Tez.
  - Storage moved to the lakehouse (Delta Lake, Apache Iceberg, Hudi) with SQL engines (Spark SQL, Trino, DuckDB) instead of hand-written jobs.

Why Spark superseded MapReduce for iteration

Iterative algorithm (e.g., PageRank, k-means, gradient descent):

MapReduce

Every iteration writes the full dataset to disk and re-reads it: the bottleneck.

Spark (RDDs)

↻ reuse across iterations, no disk round-trip

Result: order-of-magnitude speedups on iterative and interactive workloads. (Zaharia et al., “Resilient Distributed Datasets,” NSDI 2012.)

References and further reading

- Dean & Ghemawat, “MapReduce: Simplified Data Processing on Large Clusters,” OSDI 2004 (repr. CACM 51(1), 2008). Read this.
- Zaharia et al., “Resilient Distributed Datasets,” NSDI 2012. (Spark / RDDs)
- Dean & Barroso, “The Tail at Scale,” CACM 56(2), 2013. (stragglers, 99th-percentile latency)
- Akidau et al., “The Dataflow Model,” VLDB 2015. (Beam / Cloud Dataflow)
- Apache Spark 4.0 and Apache Hadoop 3.5 release notes (spark.apache.org, hadoop.apache.org).

Acknowledgement

- Structure and word-count example adapted from a Columbia Distributed Systems (COMS 4113) lecture, itself based on Asaf Cidon's 2020 guest lecture.
- Technical content verified against the original OSDI 2004 paper and current Apache project documentation (August 2026).