

1 Two Primary Problems in Big Data Processing

1.1 Data Distribution

- How to distribute data among storages across computers in a cluster.

1.2 Compute Distribution

- Once data is distributed, the challenge is to coordinate computing functions across nodes and cores.
- This is the focus of the lecture (batch vs streaming).

2 Computing Paradigms

In distributed computing, two main paradigms:

2.1 Batch Processing (Offline)

- Operates on fixed input data known ahead of time.
- Processes the entire dataset and produces results once.
- Not sensitive to latency.
- Example: Printing papers, running benchmarks on 100 test cases.

2.2 Streaming Processing (Online)

- Responds to data as it arrives.
- Must handle requests quickly (e.g., user query in chat system).
- Focus on low latency and continuous availability.
- Example: GPT query service, online recommendation engine.
- Includes SLOs (Service Level Objectives) for response time.

3 Metrics for Distributed Systems

When building large-scale systems, four core metrics are considered:

3.1 Scalability

- If more nodes are added, does performance increase (linearly or approximately)?
- Measures system growth efficiency (weak vs strong scaling).

3.2 Consistency and Correctness

- Ensures reading/writing data produces correct, up-to-date values.
- Requires synchronization → *extra communication* → *slower performance*.
- Trade-off: High consistency reduces scalability.

3.3 Fault Tolerance / High Availability

- In large clusters, failures are expected.
- System should continue without losing data or availability.
- Goal: Failures isolated and recovered quickly.
- Excessive communication can harm fault tolerance.

3.4 Latency and Throughput

- Latency: Response time for a single request.
- Throughput: Number of requests processed per unit time.
- Often a trade-off depending on application type (interactive vs batch).

4 Batch Processing

4.1 Basic Computing System Paradigm

- Every computing system follows the pattern: Input → *Process* → *Output*.
- Applies to single machines and distributed clusters alike.
- Difference between batch and streaming lies in when and how data is processed.
- Batch: Fixed dataset, offline.
- Streaming: Continuous input, near real-time processing.

4.2 Batch Processing in Unix Systems

4.2.1 Unix as the First Batch Processing System

- Early operating systems used batch commands for data processing.
- Unix provides primitive functions (chained by pipes) to form a pipeline.

4.2.2 Common Unix Commands

- File management:

```
mkdir, rm, mv, cp, cd, du, df
```

- Text processing:

```
cat, sort, uniq, tr, sed, awk
```

- Compression/Info:

```
gzip, tar, zip, uname, date, cut
```

- Help command:

```
man <command>
```

for manual pages.

4.2.3 Example Command Pipeline

- ```
cat dups.txt | sort | uniq
```

Reads dups.txt, sorts the content, filters duplicates.

- ```
cat z.txt | tr a e
```

Replaces ‘a’ with ‘e’ in file z.txt.

- Each command acts as a primitive processing function.
- Pipes (—) connect the output of one command to the input of another.
- These pipelines can be composed and debugged by inspecting intermediate outputs.

4.3 Data Flow View of Unix Pipelines

- Each command = processing function (node).
- Arguments = input data.
- Pipes = edges connecting nodes.
- Forms a directed data flow graph.
- Design idea: Primitive operators that compose into a larger pipeline.

4.4 Example of Batch Processing with Unix Tools

- Read log file.
- Split lines into fields (by whitespace).
- Output 7th element (requested URL).
- Alphabetically sort results.
- Filter duplicates (uniq).
- Sort again numerically (-n).
- Output first five lines (head -n 5).
- Each step is a transformation in the pipeline.
- Data flows through a series of modular processing components.

4.5 Limitations of Unix Batch Tools

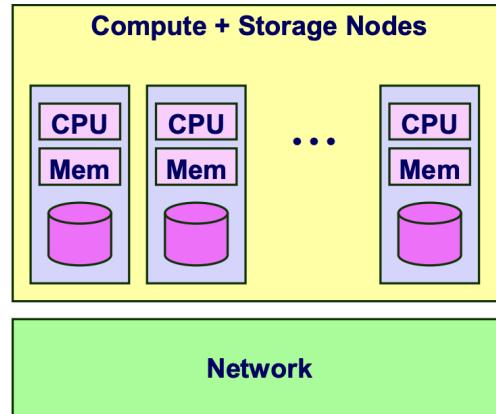
- Designed for small data on a single machine.
- Cannot scale to large-scale distributed data.
- Led to development of modern big-data frameworks such as: MapReduce and Spark
- These systems extend the same data-flow paradigm but execute on multi-node clusters with parallelism and fault tolerance.

5 MapReduce

5.1 History of MapReduce / Hadoop

- Early 2000s clusters paired **compute and storage on each node**.
 - CPU: few-core Intel processors.
 - Memory: ~32 GB.
 - Local storage: 1–2 disks.

- Network: ~ 10 Gb/s within rack, ~ 100 Gb/s across racks.
- Compared to today's NVLink (400+ Gb/s) or InfiniBand, these were very slow.
- Systems were designed to read data from local disks and avoid expensive inter-node communication.
- **Design principle:** move computation to data, not data to computation.
- Hardware capabilities historically drive software system design.
- 20 years ago, goal: make many slow unreliable machines work together instead of a single supercomputer.
- This philosophy led to Google's MapReduce model and the open-source Hadoop implementation.



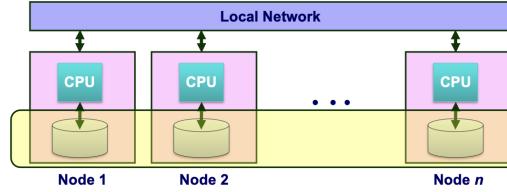
(Figure – Compute + Storage Nodes connected via Network, Slide 15.)

5.2 Data-Intensive System Challenge

- Example: computation accessing 1 TB in 5 minutes.
- A single HDD ≈ 100 MB/s read speed. \Rightarrow need ≈ 100 disks in parallel.
- Each machine \Rightarrow 1 disk \Rightarrow 100 processors + 100 disks.
- Machines must be co-located to minimize cross-network traffic.
- Requirements:
 - Many disks and processors.
 - High-speed local network.
- To achieve near-interactive performance, cluster design must reduce communication cost.
- These design pressures motivated MapReduce a programming model that coordinates many mediocre nodes for large-scale data processing.

5.3 Hadoop Project

- Hadoop combines distributed storage and computation.
- **HDFS (Hadoop Distributed File System):**
 - Files are split into blocks and replicated (default: 3 copies).
 - Failure of a node does not cause data loss.
 - Scheduler runs tasks where the data resides (locality optimization).
- **MapReduce environment:**
 - Executes tasks on nodes.
 - Manages fault tolerance, retries, and load balancing.



(Figure – Local Network with CPU Nodes and Map/Reduce Environment, Slide 17.)

5.4 Count the Number of Occurrences of a Word

- Task: count occurrences of each word across a collection of documents.
- Historical context: before embeddings (2014), web search relied on TF-IDF.
 - **TF (Term Frequency):** frequency of a word within a document.
 - **IDF (Inverse Document Frequency):** penalizes very common words.
- High TF-IDF word important in one document but rare in others.
- Google computed TF-IDF for every webpage massive parallel task.
- MapReduce made this computation feasible at web scale.

5.5 Data Models

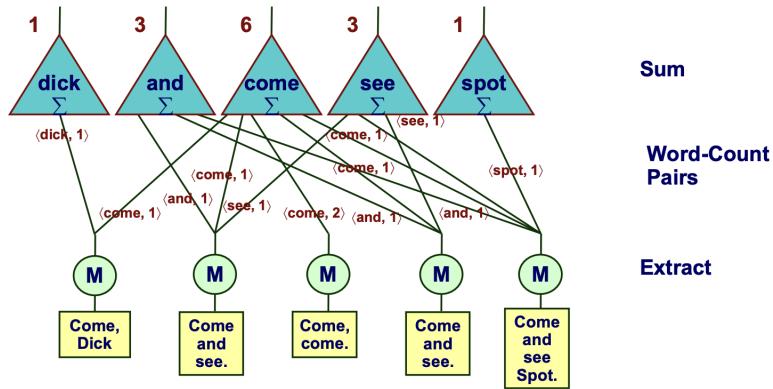
- Each document is treated as a record.
- Goal: create a word index mapping words → frequency across all documents.
- **Map:** tokenizes text into $\langle \text{word}, 1 \rangle$ pairs. **Reduce:** aggregates counts per word.

5.6 MapReduce Example

- Goal: create a word index of a set of documents.
- Input documents:
 1. Come, Dick
 2. Come and see.
 3. Come, come.
 4. Come and see.
 5. Come and see Spot.
- Each document is processed independently by a Map function.
- The **Map phase** scans and tokenizes text, emitting intermediate pairs $\langle \text{word}, 1 \rangle$ for each word occurrence.
- These pairs are the fundamental key-value elements that will later be grouped and reduced.
- After mapping, system performs Shuffle / Group step: all emitted pairs with the same key (word) are sent to the same reducer.
- **Reduce phase:** each reducer aggregates all counts for a key.
- Example output:

Word	Count
dick	1
and	3
come	6
see	3
spot	1

- Functional pattern:
 - **Map:** generate $\langle \text{word}, \text{count} \rangle$ pairs for all words in documents.
 - **Reduce:** sum word counts across documents.
- Each function is stateless → no shared memory, easy parallelization.
- Demonstrates complete MapReduce dataflow: many mappers feed into fewer reducers for global aggregation.



(Figure – Map → Shuffle → Reduce Pipeline for Word Count, Slide 25.)

5.7 Discussion

- MapReduce system handles task distribution, scheduling, and fault tolerance.
- User defines only two interfaces: `map()` and `reduce()`.
- Operates entirely on $\langle \text{key}, \text{value} \rangle$ pairs; keys and values may be any type.
- Applicable to text, logs, images, graphs—any data that can be represented as key–value pairs.
- Performance depends on how map/reduce functions are designed (e.g., shuffle cost, data balance).
- **Key idea:** MapReduce abstracts away complex system logic and provides a simple, flexible programming model for large-scale data processing.

5.8 Execution

MapReduce execution involves processing partitioned input files into the desired results. The components are:

- input files: data is partitioned into blocks and stored on different nodes
- task manager: schedules mapper and reducer tasks
- mapper: each mapper processes input blocks, transforming it into key-value pairs
- shuffle: collect data from mappers and distribute across reducers
- reducer: receives subset of data and executes reduce action
- final results: stored in distributed storage

5.8.1 Mapping

- each mapper reads input file blocks, generates key-value pairs $\langle K, V \rangle$, and writes to a (intermediate) local file
- a hash function h maps each key K to an integer i s.t. $0 \leq i \leq R$ (R is the number of local files)
- each mapper creates R files, one for each reducer

5.8.2 Reducing

- each reducer is given 1 of the R key values to process
- the reducer function is executed
- output values are written to parallel file system

In sum, MapReduce consists of reading a set of files from a file system and producing a new set of files.

5.9 Dataflow

5.9.1 Sparse MatMul Example

Matrix multiplication with sparse matrices is a common problem and is difficult to implement with parallel processing.

$$\begin{bmatrix} 10 & 20 \\ 30 & 40 \\ 50 & 60 & 70 \end{bmatrix} X \begin{bmatrix} -1 & -3 \\ -2 & -4 \end{bmatrix} = \begin{bmatrix} -10 & -80 \\ -60 & -250 \\ -170 & -460 \end{bmatrix}$$

task: compute $C = A * B$

1. Represent matrix as list of nonzero entries $\langle \text{row}, \text{col}, \text{value}, \text{matrixID} \rangle$

i.e. $\langle 1, 1, 10, A \rangle$

2. Compute all products $a_{i,k}, b_{k,j}$

Map entries from matrix A based on their column k with entries from matrix B based on their row k .

$k = 1$

$$10 * -1 = -10 \quad (a_{1,1} * b_{1,1}) \rightarrow c_{1,1}$$

$$50 * -1 = -50 \quad (a_{3,1} * b_{1,1}) \rightarrow c_{3,1}$$

$k = 2$

$$30 * -2 = -60 \quad (a_{2,2} * b_{2,1}) \rightarrow c_{2,1}$$

$$30 * -3 = -90 \quad (a_{2,2} * b_{2,2}) \rightarrow c_{2,2}$$

$$60 * -2 = -120 \quad (a_{3,2} * b_{2,1}) \rightarrow c_{3,1}$$

$$60 * -3 = -180 \quad (a_{3,2} * b_{2,2}) \rightarrow c_{3,2}$$

$k = 3$

$$\begin{aligned}
 20 * -4 &= -80 \quad (a_{1,3} * b_{3,2}) \rightarrow c_{1,2} \\
 40 * -4 &= -160 \quad (a_{2,3} * b_{3,2}) \rightarrow c_{2,2} \\
 70 * -4 &= -280 \quad (a_{3,3} * b_{3,2}) \rightarrow c_{3,2}
 \end{aligned}$$

3. Group entries by the new \langle row, col \rangle pairs and sum.

```

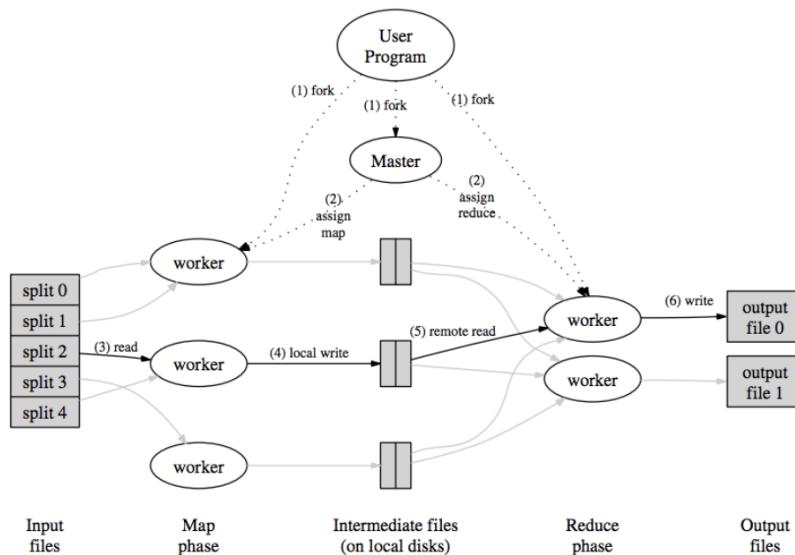
< 1, 1 >
- 10
< 1, 2 >
- 80
< 2, 1 >
- 60
< 2, 2 >
- 90 - 160 = -250
< 3, 1 >
- 50 - 120 = -170
< 3, 2 >
- 180 - 280 = -460

```

$$\begin{bmatrix} -10 & -80 \\ -60 & -250 \\ -170 & -460 \end{bmatrix}$$

6 Beyond MapReduce

6.1 MapReduce System Architecture



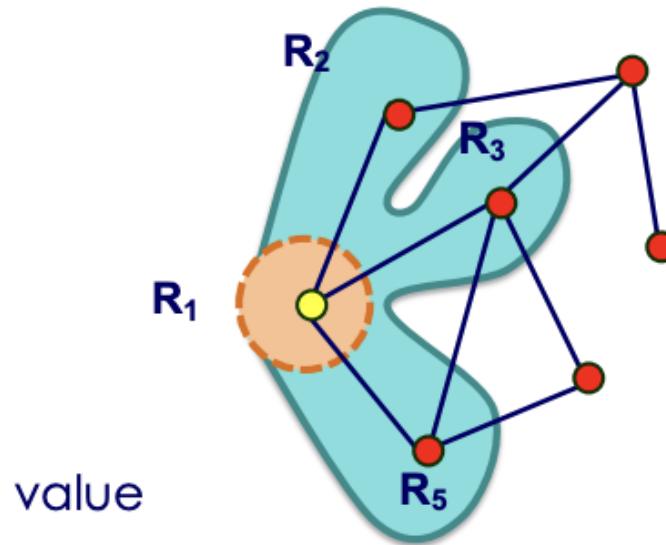
(Figure – Diagram from original MapReduce paper, slide 44.)

6.2 Map/Reduce Summary

- Typical Map/Reduce applications
 - Most are sequences of steps, each of which require a map and a reduce and are a series of data transformations
- Strengths
 - User-written functions are very simple – it's up to the system to manage the complexities of things like mapping, synchronization, fault tolerance
 - General and good for large-scale data analysis
 - First successful system behind "big data"; many companies worked on variations of MapReduce/Hadoop with specific applications in the 5-10 years after its publication
- Cons
 - Disk I/O overhead is super high (the way the mapper and reducer must communicate is reading and writing from local disks, to maintain fault tolerance – this slows the process down, especially in longer/more complicated pipelines)
 - Inflexible (each step must be completed before the next one)
- Not suitable for workloads (iterative and real-time processing)
- Still difficult to program with

6.3 PageRank Computation

- Initially assign a weight of 1.0 to each page
- Iteratively select an arbitrary node and update its value
- They converge to unique results regardless of selection ordering



$$(R_1 \leftarrow 0.1 + 0.9(\frac{1}{2}R_2 + \frac{1}{4}R_3 + \frac{1}{3}R_5)).$$

Slide 48.)

Question: How to express PageRank using MapReduce?