SE446: Big Data Engineering

Week 3A: MapReduce Concepts

Prof. Anis Koubaa

SE 446
Alfaisal University
https://github.com/aniskoubaa/big_data_course

Spring 2026

Today's Agenda

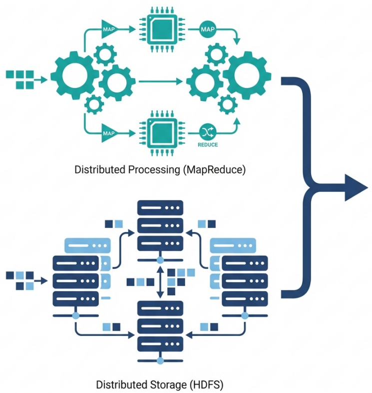
Recap: Why HDFS Alone Is Not Enough

HDFS provides:

- ✔ Distributed storage
- ✔ Fault tolerance
- ✔ Scalability

But how do we process data?

- Read 10 TB from HDFS?
- Analyze on one machine?
- ✘ Bottleneck!



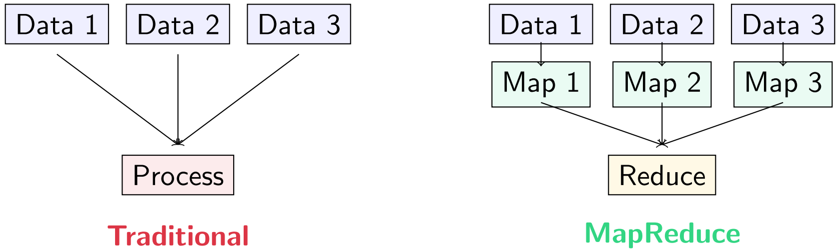
The Big Idea: Move Computation to Data

Traditional Approach

Move data to computation $\rightarrow$ Network bottleneck

MapReduce Approach

Move computation to data $\rightarrow$ Process locally, combine results



MapReduce: Origin Story

- **2004**: Google publishes landmark paper
- **Problem**: Index the entire web (billions of pages)
- **Solution**: Simple programming model for distributed processing
- **2006**: Doug Cutting implements open-source version → Hadoop

Key Insight

Many data processing tasks follow the same pattern:

1. Process each record independently (Map)
2. Group by some key
3. Aggregate groups (Reduce)

MapReduce in One Slide

$$\textbf{map}(\text{key, value}) \rightarrow \text{list(key', value')}$$

$$\textbf{reduce}(\text{key', list(value')}) \rightarrow \text{list(value'')}$$

Map Phase:

- Process each input record
- Emit (key, value) pairs
- Runs in parallel across nodes

Reduce Phase:

- Receive all values for a key
- Aggregate/combine values
- Produce final output

Crucial Concept: Mapper Input

Common Confusion

Students often ask: "What is the key passed to the Mapper?"

It is **NOT** the same as the key you emit!

- **Input Format**: Defines how files are read.
- **TextInputFormat** (Default):
  - **Key**: Byte offset of the line (e.g., 0, 104, 256...)
  - **Value**: The actual line of text

Typical Mapper Implementation

We usually **ignore** the input key (offset) and just process the value (text).

The Hidden Step: Shuffle & Sort (System Phase)

Important

Between **Map** and **Reduce**, Hadoop runs a hidden phase called **Shuffle & Sort**.

What Shuffle does (automatically):

1. Collects mapper outputs `(key, value)`
2. Sends each key to the correct reducer
3. Sorts keys and groups all values per key

Tiny Example:

Mapper outputs:

`(Hello,1) (World,1) (Hello,1) (World,1) (Hello,1)`

After Shuffle:

`Hello` $\rightarrow$ `[1,1,1]`
`World` $\rightarrow$ `[1,1]`

Key Message

You **do not write Shuffle code**.
It is done by the framework (system-level).

Why it matters

Shuffle is often the **slowest** step (network transfer).

Classic Example: Word Count

Input: 3 documents with text
Goal: Count occurrences of each word

Mapper (Python):

```
def mapper(doc_id, text):
    for word in text.split():
        emit(word, 1)
```

Input: "Hello World Hello"
Output:

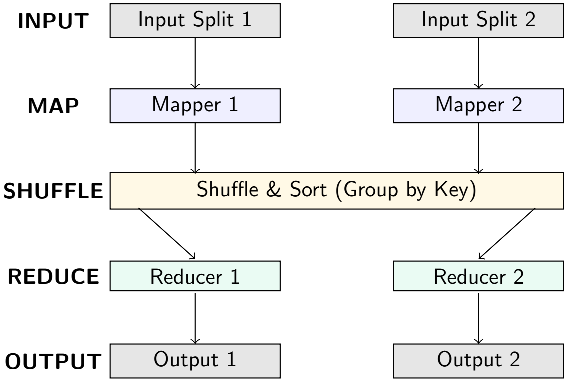
- (Hello, 1)
- (World, 1)
- (Hello, 1)

Reducer (Python):

```
def reducer(word, counts):
    emit(word, sum(counts))
```

Input: (Hello, [1, 1])
Output: (Hello, 2)

Data Flow: Step by Step



Data Flow Example: Step-by-Step (Word Count)

1) Input Splits (from HDFS)

Split 1: `"Hello World"`
Split 2: `"Hello Hadoop"`

2) Map Output (each mapper emits pairs)

Mapper 1 emits:

- `(Hello,1)`
- `(World,1)`

Mapper 2 emits:

- `(Hello,1)`
- `(Hadoop,1)`

3) Shuffle & Sort (System-level, automatic)

Group all values with the same key:

- `Hello` → `[1,1]`
- `World` → `[1]`
- `Hadoop` → `[1]`

4) Reduce Output (final result)

- `(Hello, 2)`
- `(World, 1)`
- `(Hadoop, 1)`

Map creates pairs → **Shuffle** groups by key → **Reduce** aggregates

Real-World Example: Chicago Crime Data

Dataset: Chicago Crime Reports

Real-world dataset from the City of Chicago (6GB+, millions of rows).

Key Columns:

- `ID`, `Case Number`: Unique identifiers
- `Date`: Timestamp of incident
- `Primary Type`: Category (THEFT, BATTERY...)
- `Description`: Detailed description
- `Arrest`: Boolean (True/False)

Sample Data (CSV):

Date	Primary Type	Arrest
01/01/2023	**THEFT**	False
01/01/2023	**BATTERY**	True
01/02/2023	**THEFT**	False
01/02/2023	**ASSAULT**	True
...	...	...

The Challenge

How many crimes of each type occurred in total?

Stop & Think: Model the Solution

Before Looking at the Code...

Problem: **Count crimes by type**

What should my KEY be?

Think for 15 seconds...

What should my VALUE be?

Think for 15 seconds...

Stop & Think: Model the Solution

Before Looking at the Code...

Problem: **Count crimes by type**

What should my KEY be?

What should my VALUE be?

The Answer

Key = Crime Type (e.g., "THEFT") — **Value** = 1 (for counting)

Crime Count: Mapper

```python
def crime_mapper(record):
    """
    Input: One crime record (dictionary)
    Output: (crime_type, 1)
    """
    crime_type = record['Primary Type']
    return (crime_type, 1)

# Example
record = {'Primary Type': 'THEFT', 'District': 11, ...}
crime_mapper(record)  # Output: ('THEFT', 1)
```

Key Point: Mapper processes ONE record at a time, emits (key, value)

Crime Count: Reducer

```python
def crime_reducer(crime_type, counts):
    """
    Input: crime_type (key), list of counts (values)
    Output: (crime_type, total_count)
    """
    total = sum(counts)
    return (crime_type, total)

# Example
crime_reducer('THEFT', [1, 1, 1, 1, 1])
# Output: ('THEFT', 5)
```

Key Point: Reducer receives ALL values for ONE key

Complete Data Flow (Color-Coded)

Phase	What goes in	What happens	What comes out
Input	CSV file	Split into individual records	One record at a time
Map	`{"Primary Type": "THEFT"}`	Extract key and emit count	`("THEFT", 1)`
Shuffle	All mapper outputs *(system-controlled)*	**Group values by key**	`"THEFT" → [1,1]`
Reduce	`"THEFT", [1,1]`	Aggregate values (sum)	`("THEFT", 2)`

Data shape evolution: Record → (Key,Value) → Key→[Values] → Final Result

Key Design Cheatsheet

Designing Your Key is 90% of the Battle

Goal	Map Output Key	Map Output Value
Counting (e.g., Word Count)	Item "apple"	1 1
Grouping (e.g., Crimes by Type)	Group ID "THEFT"	Full Record {id: 1, loc: "Street"...}
Filtering (Map-only job)	- (None)	- (None)
Inverted Index (Search Engine)	Keyword "Hadoop"	Document ID "doc1.txt"

Quick Quiz: Model These Problems (1/2)

For each scenario, identify the Key and Value:

1. Total sales per store

Key: ______________

Value: ______________

2. Website visits per country

Key: ______________

Value: ______________

3. Average temperature per city

Key: ______________

Value: ______________

Hint: For averages, what EXTRA info do you need?

Quick Quiz: Model These Problems (1/2) - Answers

For each scenario, identify the Key and Value:

1. Total sales per store

Key: **Store ID**
Value: **Sale Amount**

2. Website visits per country

Key: **Country**
Value: **1**

3. Average temperature per city

Key: **City**
Value: **(temp, 1)**
✔ Need **sum** and **count** to compute average!

Quick Quiz: Model These Problems (2/2)

More challenging scenarios:

4. Inverted Index (Search Engine)

Build an index: which documents contain each word?

Key: ______________

Value: ______________

6. Grouping: All employees per department

Collect all records belonging to the same group.

Key: ______________

Value: ______________

5. Filtering: Find all orders ¿ $1000

Only output records matching a condition.

Key: ______________

Value: ______________

Quick Quiz: Model These Problems (2/2) - Answers

More challenging scenarios:

4. Inverted Index (Search Engine)

Build an index: which documents contain each word?

Key: **Word**
Value: **Document ID**

5. Filtering: Find all orders ¿ $1000

Only output records matching a condition.

Key: **(any / None)**
Value: **Full record**
Map-only job: emit if condition met, else skip!

6. Grouping: All employees per department

Collect all records belonging to the same group.

Key: **Department ID**
Value: **Employee record**
Reducer concatenates all employee records!

Input Splits vs. HDFS Blocks

HDFS Blocks (Storage)

- Physical division of data.
- Fixed size (e.g., 128MB).
- Replicated for safety.

Input Splits (Processing)

- Logical division for Mapper.
- **1 Split = 1 Mapper Instance**.
- Usually, Split Size $\approx$ Block Size.

Exam Key Takeaway

HDFS Blocks are for storage reliability.
Input Splits control the number of mappers.

Visual Example: 400MB File Processing

1. Original File (400MB)

mydata.csv (400MB)

↓

2. HDFS Blocks (Storage)

Block 1 128MB	Block 2 128MB	Block 3 128MB	Block 4 16MB

↓

3. Input Splits (Processing)

Split 1 Mapper 1	Split 2 Mapper 2	Split 3 Mapper 3	Split 4 Mapper 4

Key Points:

✔ 4 blocks = 4 replicated chunks

✔ 4 splits = 4 parallel mappers

✔ Usually 1 block ↔ 1 split

The Shuffle Phase

Often Overlooked but Critical!

Shuffle is the **most expensive** phase - involves network transfer

What happens during Shuffle:

1. Mapper outputs are **partitioned** by key
2. Data is **sorted** by key
3. Data is **transferred** to appropriate reducer
4. Values for same key are **grouped** together

Example:

- Mapper 1 emits: (A, 1), (B, 2), (A, 3)
- Mapper 2 emits: (B, 4), (A, 5)
- After shuffle: A $\rightarrow$ [1, 3, 5], B $\rightarrow$ [2, 4]

Reverse Engineering Exercise

After Shuffle, the reducer receives:

- `"THEFT" → [1, 1, 1, 1, 1]`
- `"ASSAULT" → [1, 1, 1]`
- `"BATTERY" → [1, 1]`

Questions:

1. How many THEFT records were in the input?
2. What did the Mapper emit for each THEFT record?
3. What will the Reducer output for THEFT?

Answers

1. 5 THEFT records
2. `("THEFT", 1)`
3. `("THEFT", 5)`

Key Insight

Why Do We Sort Before Reducing?

The "Shuffle" phase guarantees that the Reducer receives keys in **sorted order**.

Why is this mandatory?

1. **Grouping**: All values for Key K must be contiguous to be passed as a single list (iterator).
2. **Memory Efficiency**: The Reducer doesn't need to fit all data in RAM. It can "stream" through the sorted keys.

Golden Rule

Reducers *never* see unsorted keys.

Combiners: Local Aggregation

Problem: Shuffle transfers lots of data
Solution: Combine locally before shuffle!

Without Combiner	**With Combiner**
$\boxed{\text{Mapper 1}}$	$\boxed{\text{Mapper 1}}$
(A,1)(A,1)(A,1)	(A,1)(A,1)(A,1)
↓	$\boxed{\text{Combiner}}$
Send 3 values	(A,3)
	↓
	Send 1 value!

Note: Combiner must be **associative** and **commutative** (like sum, max, min)

Critical Nuance: Combiners are Optional

The "Mini-Reducer" Rule

A Combiner is an **optimization**, NOT a guarantee.

Hadoop may run the combiner:

- **Zero times**: If the spill file is small.
- **Once**: Standard case.
- **Multiple times**: During merge sorts on disk.

Implication for Code

Your Combiner logic must be:

1. **Associative** and **Commutative** (Order doesn't matter).
2. Safe to run repeatedly (Idempotent-ish).
3. Input/Output types must match the Reducer input.

Partitioner: Key Distribution

Default: Hash partitioning (hash(key) % num_reducers)

Why customize?

- Ensure related keys go to same reducer
- Balance load across reducers

Example: Analyzing crimes by year

- Key: "2023-THEFT", "2023-ASSAULT", "2024-THEFT"
- Partitioner: Extract year, send same year to same reducer
- Result: Each reducer handles one year's data

The Problem of Data Skew (Hot Keys)

Scenario:

- You count words in Wikipedia.
- Word "the" appears 1 billion times.
- Word "zygote" appears 100 times.

Result: The reducer handling "the" runs for 5 hours. All other reducers finish in 1 minute. The job waits for the slow reducer ("Straggler").

Impact

1 Reducer bottleneck = Slow Job

Solutions:

- **Salt keys**: "the" $\rightarrow$ "the-1", "the-2"...
- **Custom Partitioner**
- **Combine** early

Design Challenge: Handling Skewed Data

Scenario: Analyze 10 Billion Tweets

- 5 billion mention "Taylor Swift" ♫
- 5 billion mention everyone else combined

Standard Approach:

- Key = celebrity_name
- Value = 1

✗ Problem: One reducer handles 5B values while others are idle!

Your Challenge:

How would you modify the **KEY** to distribute the load?

Hint: Can you "spread" Taylor Swift across multiple reducers?

✓ Solution: Salted Keys

Key = "Taylor Swift-**1**", "Taylor Swift-**2**", ..., "Taylor Swift-**100**"
Then run a second MapReduce job to combine the partial counts!

Fault Tolerance: The "Secret Sauce"

Question: What happens if a node crashes during a 10-hour job?

Mapper Failures

- Master detects failure.
- Reschedules the task on another node containing the data replica.
- ✔ Re-execution is safe (Idempotent).
- *Note: Output is local, so it must be re-run.*

Reducer Failures

- Master detects failure.
- Reschedules the task on another node.
- Retrieves mapper outputs again from source nodes.

User code does NOT need to handle checkpoints or failures. The framework handles it!

Job Lifecycle & Output Files

One Output File Per Reducer

- HDFS does not support concurrent writes to a single file.
- Each Reducer writes its own part file.
- **Naming Convention**: `part-r-00000`, `part-r-00001`, ...

Practical Tip: If you want exactly **one** final output file, set `NumReduceTasks = 1`. (Warning: This kills parallelism for the Reduce phase!)

MapReduce: Good vs Bad Use Cases

Good for:

- Batch processing
- Large datasets (TB+)
- Embarrassingly parallel tasks
- Aggregations, counts, sums
- Log analysis
- ETL pipelines

Bad for:

- Real-time queries
- Iterative algorithms
- Small datasets
- Interactive analysis
- Graph processing
- Machine learning

Rule of Thumb

If you need results in ¡ 1 minute, MapReduce is probably wrong choice

MapReduce Limitations

1. **High Latency**: Minutes to hours for jobs
2. **Disk I/O**: Intermediate results written to disk
3. **Only Two Phases**: Complex workflows need chaining
4. **No Iteration**: Each job reads from scratch

The Evolution

MapReduce limitations led to:

- **Spark**: In-memory processing, DAG execution
- **Flink**: True streaming
- **Hive**: SQL on MapReduce

We'll cover these in later weeks!

Key Takeaways

1. **MapReduce** = programming model for distributed processing
2. **Map**: Process records in parallel → emit (key, value)
3. **Shuffle**: Group by key (automatic, expensive)
4. **Reduce**: Aggregate values per key → final result
5. **Data locality**: Computation moves to data, not vice versa
6. **Combiner**: Local aggregation to reduce shuffle

The MapReduce Design Formula

KEY	**What do I want to GROUP BY?**
VALUE	**What do I want to AGGREGATE?**

Write this on your exam notecard!

Next Session: Hands-On Practice

Week 3B: Implementing MapReduce in Python

- Implement mappers and reducers
- Complete crime analysis exercises
- Start Milestone 2

Pre-class preparation:

- Watch: Python MapReduce Tutorial
- Clone team repo and pull latest
- Review Chicago crime dataset