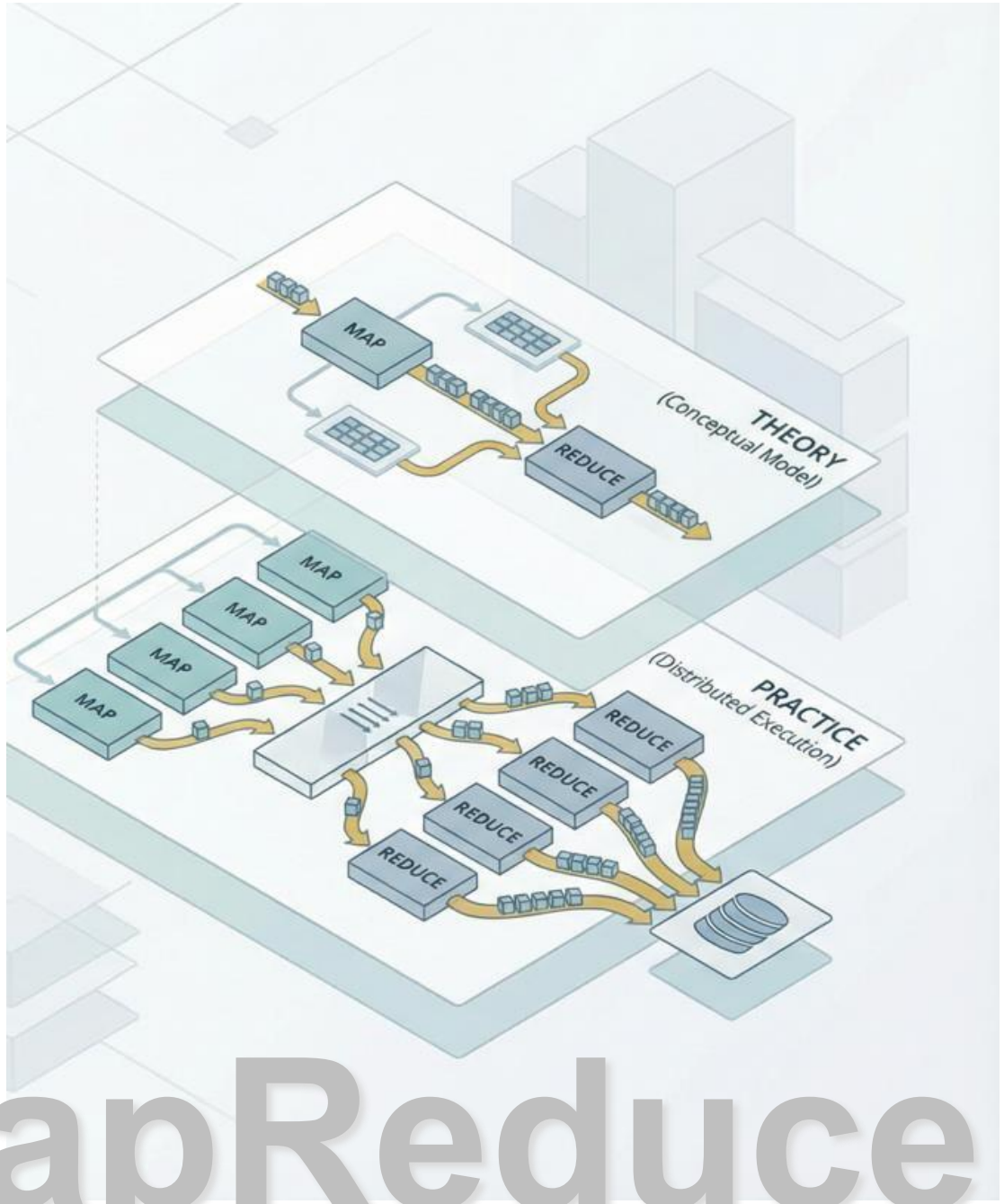
Architecture of Large-Scale Systems

By Dr. Taghinezhad

Mail:
a0taghinezhad@gmail.com

https://ataghinezhad.github.io/

Chapter:



MapReduce

Outline

- **Problem Statement / Motivation**
- An Example Program
- MapReduce vs Hadoop
- GFS (Google File System) / HDFS (Hadoop distributed File System)
- MapReduce Fundamentals
- Example Code
- Workflows
- Conclusion / Questions

Why MapReduce Exists

ezhad.github.io/

Data Tsunami vs. Single Server

Web-scale logs outgrow vertical scaling, scaling, forcing a shift to distributed computation. MapReduce turns unreliable unreliable hardware into a reliable service. service

Vertical Scaling

One massive, expensive server. server. Limited by physics and and budget.

$10M

Horizontal Scaling

Many cheap, unreliable nodes. nodes. The MapReduce philosophy.

$10k x 1000

The Core Trade-off:

Would you rather have one supercomputer for $10M with a with a 10-year MTBF, or 1000 cheap nodes, each with a 3-year a 3-year MTBF, for the same price?

Why MapReduce?

- **Early Distributed Computing Challenges:**
  - Managing **large concurrent systems** efficiently.
  - Utilization of **grid computing** for resource sharing.
  - Development of **custom solutions** ("roll your own").

Why MapReduce? Key Challenges in Distributed Systems

- **Complexity of Threading:**
  - Difficult to manage threads across distributed systems.
- **Scaling:**
  - Adding more machines without performance bottlenecks.
- **Failure Handling:**
  - How to recover gracefully when machines fail.
- **Communication Between Nodes:**
  - Ensuring nodes exchange data efficiently and reliably.
- **Scalability:**
  - Evaluating whether the solution can handle increasing workloads.

Before MapReduce: Background and Motivation

- In the era preceding MapReduce, large-scale data processing faced several challenges due to limitations in hardware, software, and methodologies.
- **Large Concurrent Systems**
  - **Definition**: Large concurrent systems refer to computing environments where multiple operations run simultaneously, sharing resources like CPU, memory, and storage.

•**Challenges**:
- **Concurrency Management**: Ensuring operations do not interfere with each other (e.g., locking resources).
- **Data Integrity**: Preventing data corruption due to simultaneous read/write operations.
- **Performance**: Maximizing throughput without sacrificing speed or efficiency.

•**Use Cases**: Early applications included web servers, database systems, and large-scale simulations.

Rolling Your Own Solution

- **Pre-MapReduce Era: Custom Solutions for Data Processing**
  Before frameworks like MapReduce, developers often created custom solutions to handle large-scale data processing.
- **Advantages**:
  - **Tailored Optimization**: Custom solutions could be highly optimized for specific tasks.
  - **Control**: Full control over the implementation, allowing flexibility.
- **Drawbacks**:
  - **Complexity**: Building a distributed processing system from scratch requires significant expertise.
  - **Maintenance**: Custom solutions are harder to maintain, especially as team members change.
  - **Scalability**: Many home-grown solutions struggled to scale efficiently.

Real-world example

- **Scenario**
  - You are tasked with analyzing a massive dataset containing metadata for millions of YouTube videos. The dataset includes fields such as:
  - **Likes:** Number of likes each video has received.
  - **Comments:** Number of comments.
  - **Engagement Rate:** Metric calculated using likes, views, and shares.
  - **Video Duration:** Length in seconds or minutes.
- **Challenges:**
  - The dataset is too large for a single machine and distributed across multiple servers.
- **Objective:**
  Using the MapReduce paradigm to:
  1. Efficiently process the dataset across distributed servers.
  2. Identify videos with a specific number of likes (e.g., exactly 1,000 likes or within a range).

Understanding Distributed Data Processing with MapReduce

- **1) Distributed Data:**
  - Large datasets are split into distributed chunks of data.
  - A central file system controller manages the locations of all data chunks in the system.
- **2) No Data Movement**
  - Instead of moving data to a centralized location, we send the computation (map) to the data to avoid the cost of moving large amounts of data.
- **3) Key-Value Structure of the Data**
  - Data chunks in MapReduce share a common structure: key-value pairs.
  - When reducing or processing data chunks, they originate from the same dataset, identified by a common key.

Understanding Distributed Data Processing with MapReduce

- **4) Machine Failures**
  - In the case of machine failure during the map or reduce operation, the operation is simply retried.
  - The system ensures that failure does not affect the overall computation.
- **5) Idempotency**
  - The system is **idempotent**, meaning that reapplying the map and reducing functions multiple times will not alter the final output.
- **6) Engineering with Libraries (e.g., Hadoop)**
  - Engineers use libraries like **Hadoop** to manage distributed data processing.
  - The focus is on input/output operations and leveraging the distributed framework for computation.

Fault-Tolerance by Re-Computation

MapReduce achieves fault tolerance not through complex complex recovery protocols, but by re-running failed tasks. tasks. This is safe because map and reduce functions are are deterministic.

Heartbeat Protocol

Workers send "I'm alive" signals. No heartbeat? Master Master marks tasks as idle to be re-scheduled.

Replicated Intermediate Data

Map output data is stored on two nodes to prevent data loss data loss during shuffle.

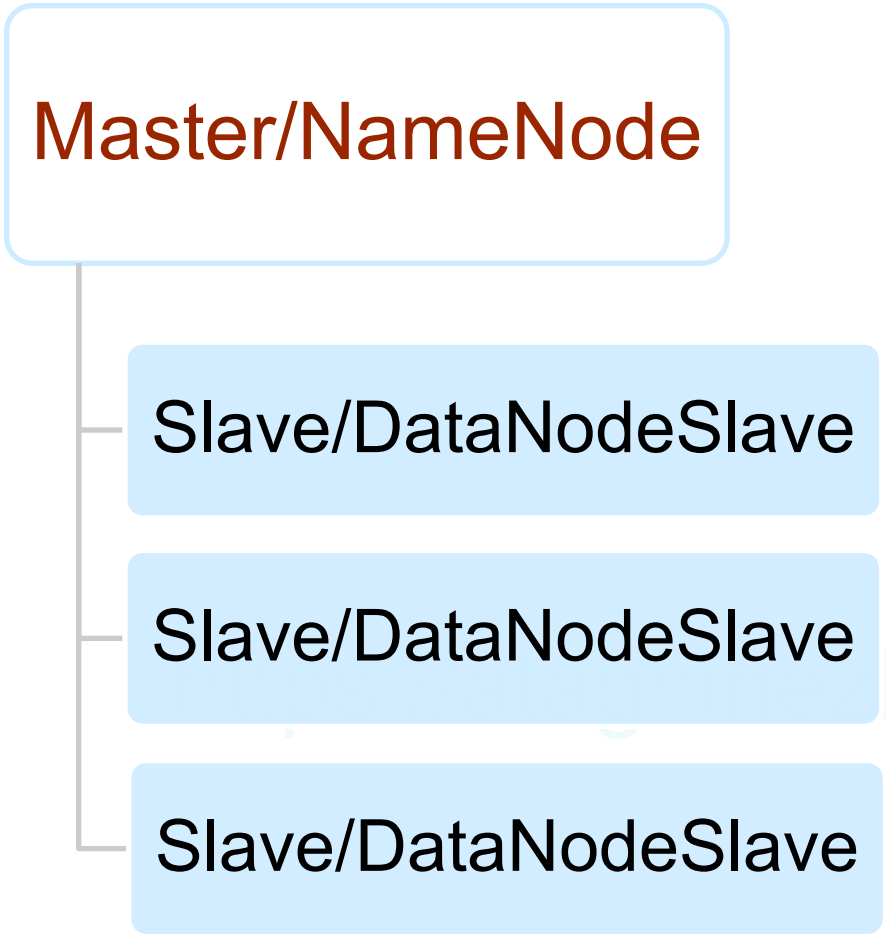
Components of Hadoop

Storage unit of Hadoop

Processing unit of Hadoop

HDFS components

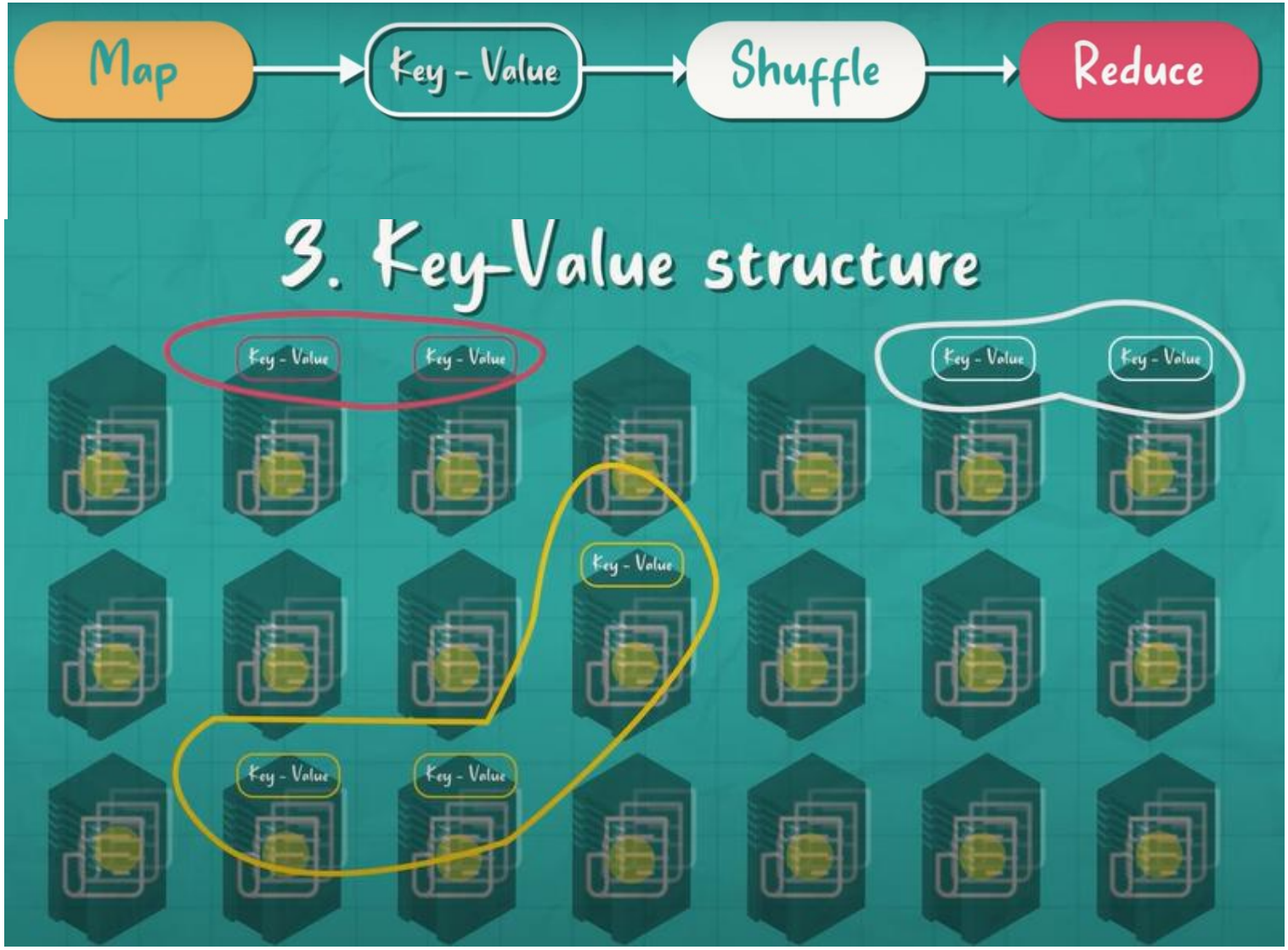
- Master/Slave nodes typically form the HDFS cluster



- Splits the input data.
- Coordinates the shuffle and sort process.
- Collects final results.

- Perform mapping of data chunks.
- Execute reduction tasks after receiving shuffled and sorted data.

Map reduce Structure



MapReduce Example: Word Count

- **Objective: Word Count** Using MapReduce: Objective and Phases
  - **Input**: A large, unstructured text file of any size. This could range from a few KB to many TB.
  - **Output**: A list of each unique word in the text along with the number of times it occurs.
    - Given the sentence: ***The doctor went to the store.***

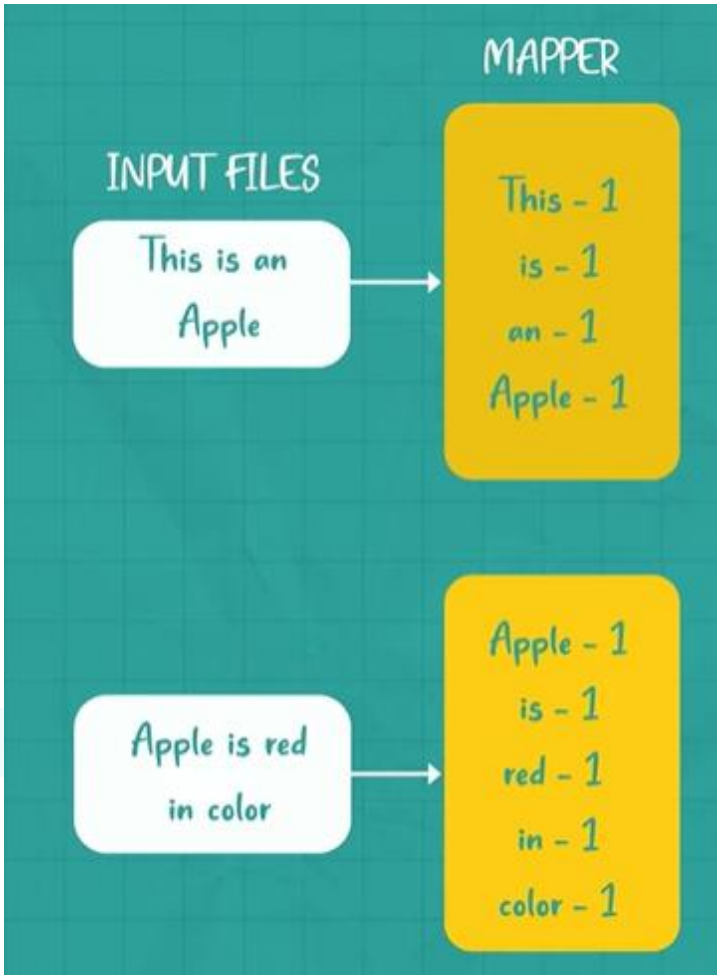
Word	Count
The	2
Doctor	1
Went	1
To	1
Store	1

- To achieve this, we assume:
  - **Case-Insensitive Counting**: "The" and "the" are counted as the same word.
  - **Punctuation Ignored**: Words are split correctly, and punctuation (like periods or commas) is ignored.
- **MapReduce Phases:** MapReduce divides the process into two main phases: **Map** and **Reduce**. We'll also see a **Shuffle and Sort** step that occurs between them.
  - **1) Map Phase**
  - **2) Shuffle and Sort Phase (Intermediary)**
  - **3) Reduce Phase**

Example, Word count

In this scenario, we'll use MapReduce to process two files. The task is to count occurrences of each word across these files, where each file contains different sets of text.

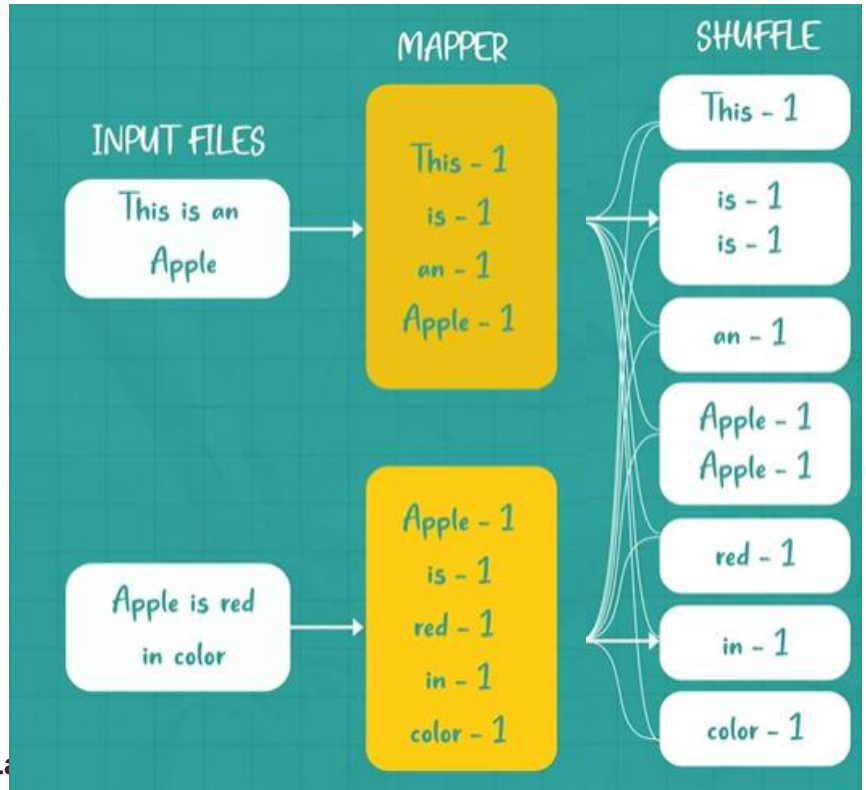
- **1. Mapper Phase**
  - **Input**: Each Mapper processes one or more files (or portions of them, called "splits") and extracts the words from these files.
  - **Key Output**: The word (e.g., "apple").
  - **Value**: The occurrence count, set to 1 for each instance of the word.



Example, Word count

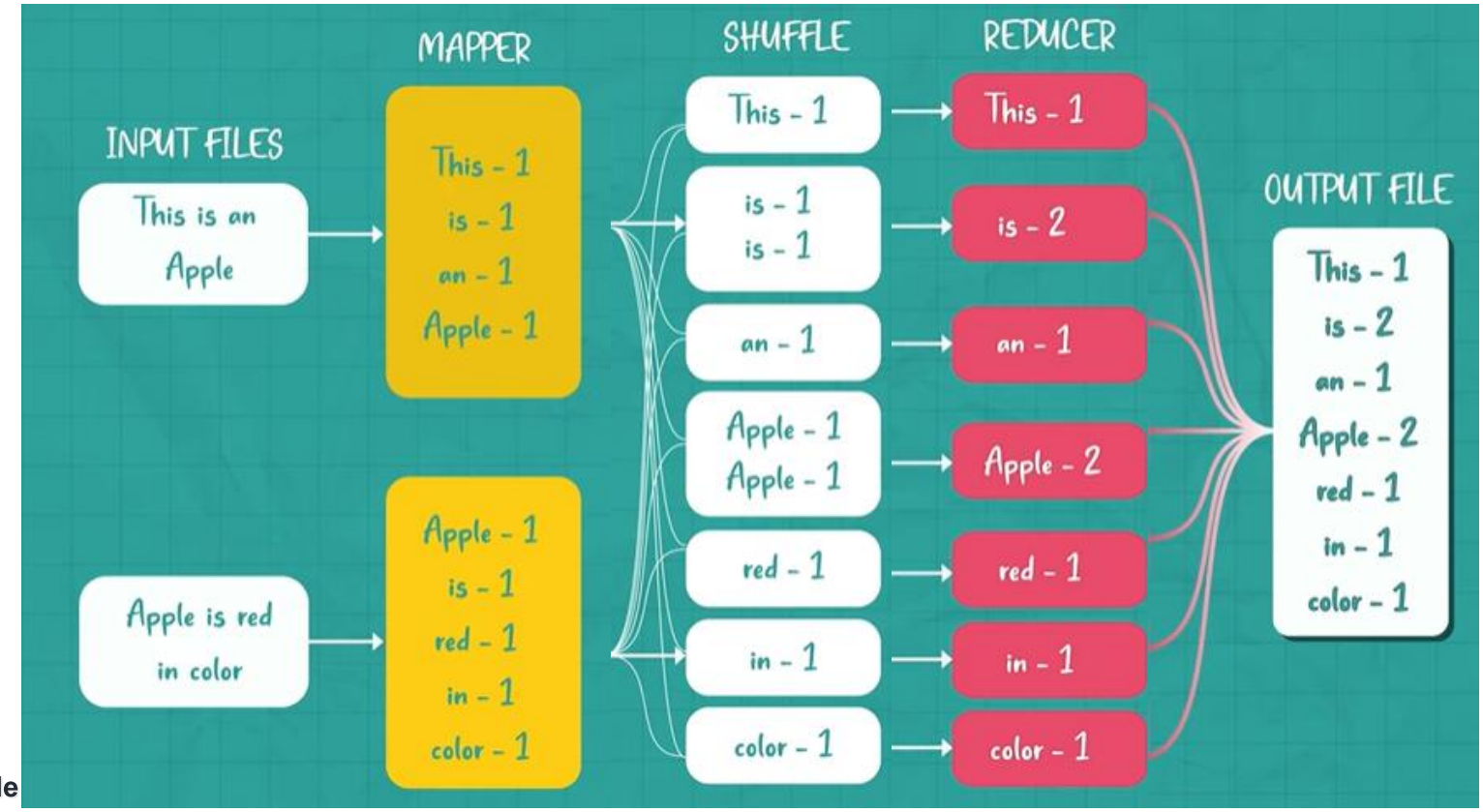
2. Shuffle and Sort Phase

- **Purpose**: The **shuffle** and **sort** phase **automatically** groups all **key-value pairs** by key. Each **unique** word becomes a single **group**, containing all its occurrences from all mappers.
- **Output**: The **shuffle** process outputs a list of grouped key-value pairs where:
  - **Key**: The word.
  - **Values**: A list of occurrence counts from all mapper



Example, Word count

- **3. Reducer Phase**
  - **Purpose**: The **reducer processes** each group of **key-value pairs** by **aggregating** all values for the same key (word) to produce a final count.
  - **Logic**: For each **unique key**, the **reducer** sums up all occurrences in the list of values.
  - **Output**: A single **output file** that contains each **unique word** and its total **occurrence** count across both files.



Large scale

MapReduce Example: Word Count (Cont.)

- **Map Phase**
  - In the Map phase, the input data (text file) is **split into smaller chunks**, **and each chunk is processed in parallel** by different machines or nodes.
  - The mapper function takes in these chunks and processes each one independently to produce a **series** of **intermediate key-value pairs**.
- **Map Function**:
  - For each chunk of **text**, the function:
    - **Tokenizes** the text into **words**.
    - **Converts** all words to **lowercase** (case-insensitive).
    - **Strips punctuation**.
    - **Outputs** a key-value **pair** for each word, where the **key** is the word itself, and the **value** is 1 (indicating one occurrence of the word in that chunk).
  - The **mapper** will **output** the following key-value pairs:
    - ("the", 1) ("doctor", 1) ("went", 1) ("to", 1) ("the", 1) ("store", 1)
  - If multiple mappers are running in parallel on different chunks, each mapper will output similar lists of key-value pairs for its assigned text chunk.

MapReduce Example: Word Count (Cont.)

2. Shuffle and Sort Phase

- The shuffle and sort phase occurs automatically after the mapping phase. It is critical for organizing and grouping data so that the reducers can work efficiently.

- **Purpose**:
  - **Grouping by Key**: All occurrences of the same word (key) are grouped together.
  - **Sorting**: Within each key group, values are sorted or simply collated.
- **Example Output After Shuffle and Sort**:
  - After this phase, the intermediate data might look like:

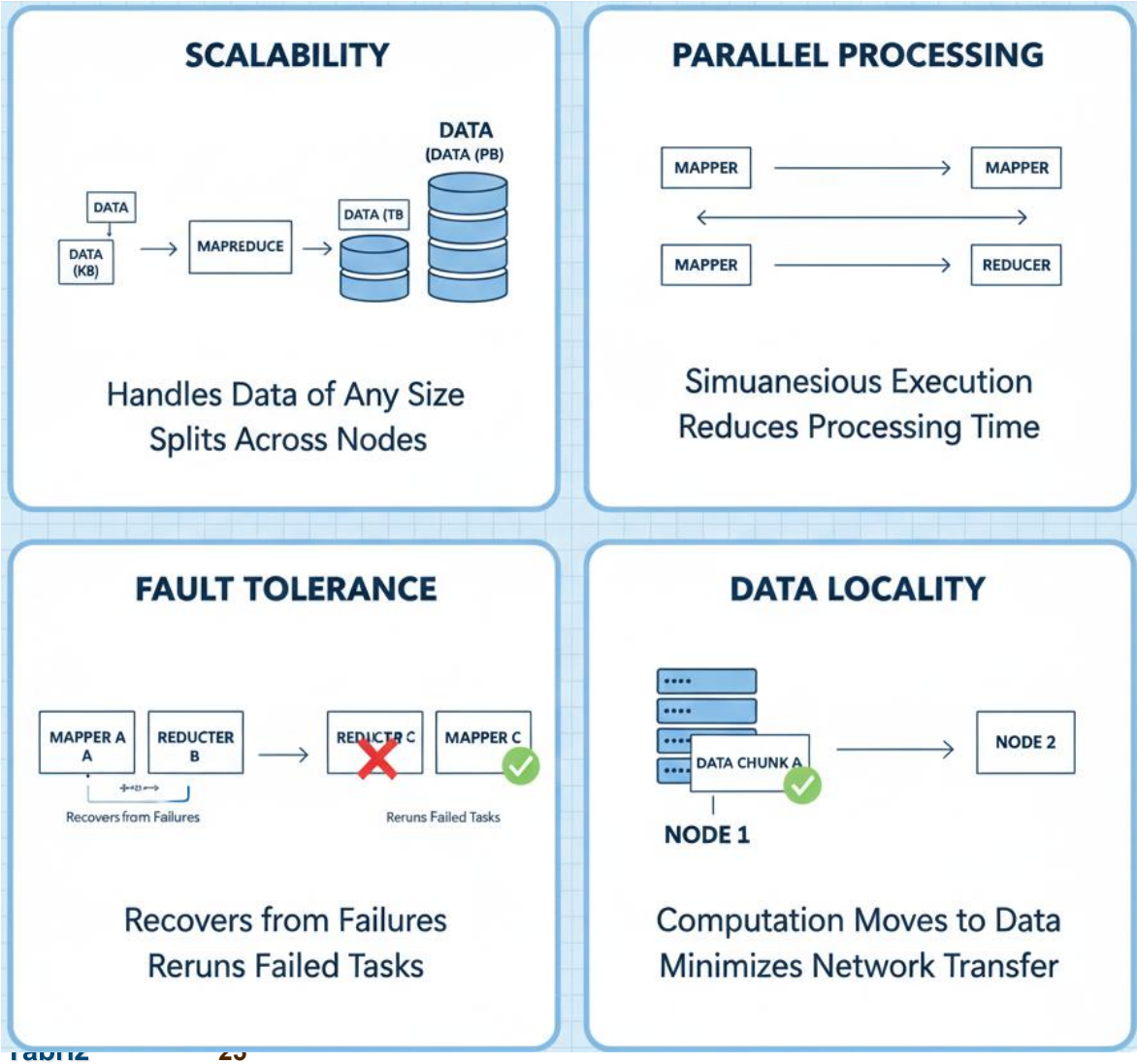
```
("doctor", [1])
("store", [1])
("the", [1, 1])
("to", [1])
("went", [1])
```

MapReduce Example: Word Count (Cont.)

- **3. Reduce Phase**
  - In the reduce phase, each group of key-value pairs (for each word) is sent to a reducer, which aggregates the values to compute a final count for each word.
- **Reduce Function**:
  - For each word (key), the reducer sums up the values in the list.
  - Outputs the word along with its total count.
- **Example**:
- For our grouped key-value pairs:
  - ("doctor", [1]) => ("doctor", 1)
  - ("store", [1]) => ("store", 1)
  - ("the", [1, 1]) => ("the", 2)
  - ("to", [1]) => ("to", 1)
  - ("went", [1]) => ("went", 1)
- The output is a set of word-count pairs that provide the final count of each unique word in the text.

Advantages of MapReduce in Word Count Example

- **Scalability:** Supports data processing from kilobytes to terabytes or petabytes.
- **Parallelism:** Independent mappers and reducers execute concurrently, reducing overall runtime.
- **Fault Tolerance:** Failed tasks are automatically re-executed on other nodes.
- **Data Locality:** Computation is scheduled near data to minimize network overhead.



Outline

- Problem Statement / Motivation
- An Example Program
- MapReduce vs Hadoop
- GFS / HDFS
- MapReduce Fundamentals
- Example Code
- Workflows
- Conclusion / Questions

MapReduce vs Hadoop

- The MapReduce paradigm was formally introduced in the seminal paper:
- *"MapReduce: Simplified Data Processing on Large Clusters"*
  Jeffrey Dean and Sanjay Ghemawat, Google
- In this paper, the authors describe Google's internal system for large-scale data processing, motivated by challenges in indexing the web, log analysis, and large graph computations. The contribution is **conceptual and architectural**, not an open-source software release.

MapReduce vs Hadoop

Hadoop MapReduce

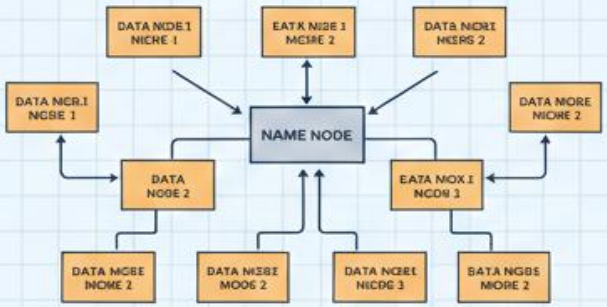
- **Open-source implementation** of the MapReduce paradigm
- **Originally initiated by**: Yahoo
- **Inspired by**: Google's MapReduce paper
- **Language**: Java
- **Ecosystem**: Apache Hadoop (with HDFS, YARN, etc.)
- Key distinctions:
- Hadoop is **not developed by Google**
- It re-implements the *ideas* of MapReduce, not Google's code
- Designed for general-purpose clusters and enterprise environments

Hadoop Ecosystem

- Hadoop evolved beyond MapReduce into a full ecosystem supporting a range of data processing needs, including:
  - **HDFS (Hadoop Distributed File System)**: Hadoop's distributed storage system, optimized for scalability and fault tolerance.
  - **Hive**: A data warehousing tool on top of Hadoop, allowing SQL-like querying.
  - **Pig**: A platform for complex data transformations.
  - **Spark**: Although separate from Hadoop's MapReduce, Spark became popular for faster, in-memory data processing.

Distributed Filesystems (GFS and HDFS)

- Distributed file systems managing data across multiple machines.
- Designed for large-scale data processing (e.g., MapReduce).



COMPARISONS

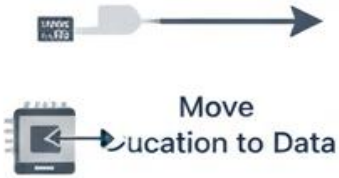
GFS (Google File System)
- Web Indexing
- Search Ranking

HDFS (Datributed File System)
- Web Indexing
- Log Analysis
- Big Data Workflows

KEY FEATURES

FAULT TOLERANCE

3 Copies



Move Computation to Data

DATA LOCALITY

Write-Once, Read-Many

Large scale

GFS/HDFS

1.Fault Tolerance

- Replicates data across nodes (default 3 copies in HDFS).

2.Data Locality

- Moves computation to where data resides, reducing network overhead.

3.Optimized for Big Data

- Write-once, read-many model for efficient processing.

Comparison

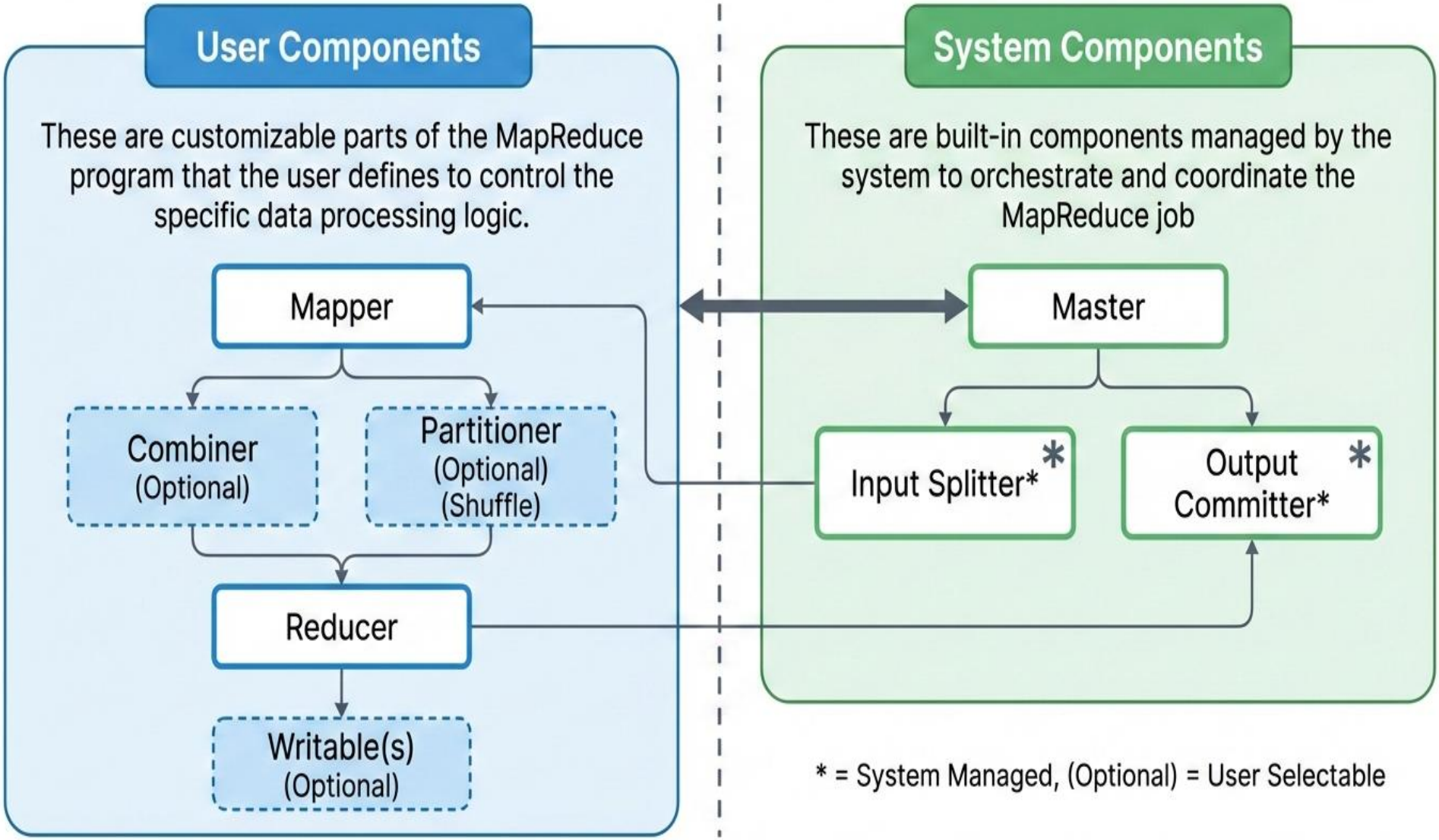
- **GFS**: Web indexing, search ranking at Google.
- **HDFS**: Log analysis, big data workflows with Hadoop tools.

Feature	GFS (Google File System)	HDFS (Hadoop File System)
Purpose	Proprietary, Google-specific	Open-source, Hadoop ecosystem
Integration	Google's internal tools	Hive, Pig, Spark integration
Chunk Size	64 MB (typical)	128 MB (default, configurable)

Outline

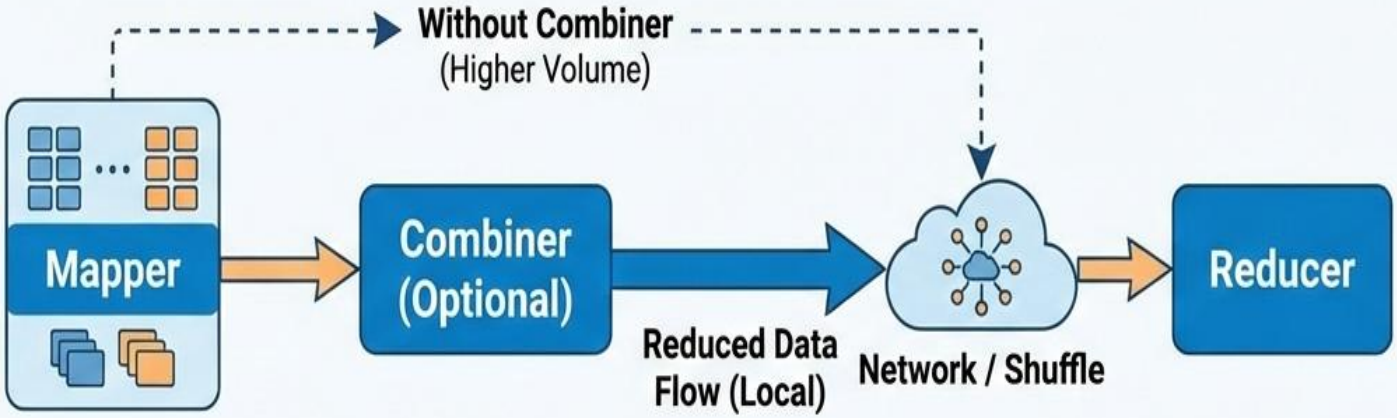
- Problem Statement / Motivation
- An Example Program
- MapReduce vs Hadoop
- GFS / HDFS
- MapReduce Fundamentals
- Example Code
- Workflows
- Conclusion / Questions

Major Components



Combine (Optional) in MapReduce

Purpose: Acts as a mini-reducer to reduce data volume before network transfer. Optimizes performance by minimizing data shuffling.



Purpose: Acts as a **mini-reducer** to reduce data volume before network transfer. **Optiimizes** performance by minimizing data shuffling.

Functionality: Runs on mapper output, performs preliminary aggregation on the same machine, minimizing data sent to reducers.

Example (Word Count):

Before Combiner
(Mapper Output)

```
("word", 1)
("word", 1)
("word", 1)
("data", 1)
("data", 1)
```

After Combiner
(Local Aggregation)

```
("word", local_count)
("data", local_count)
```

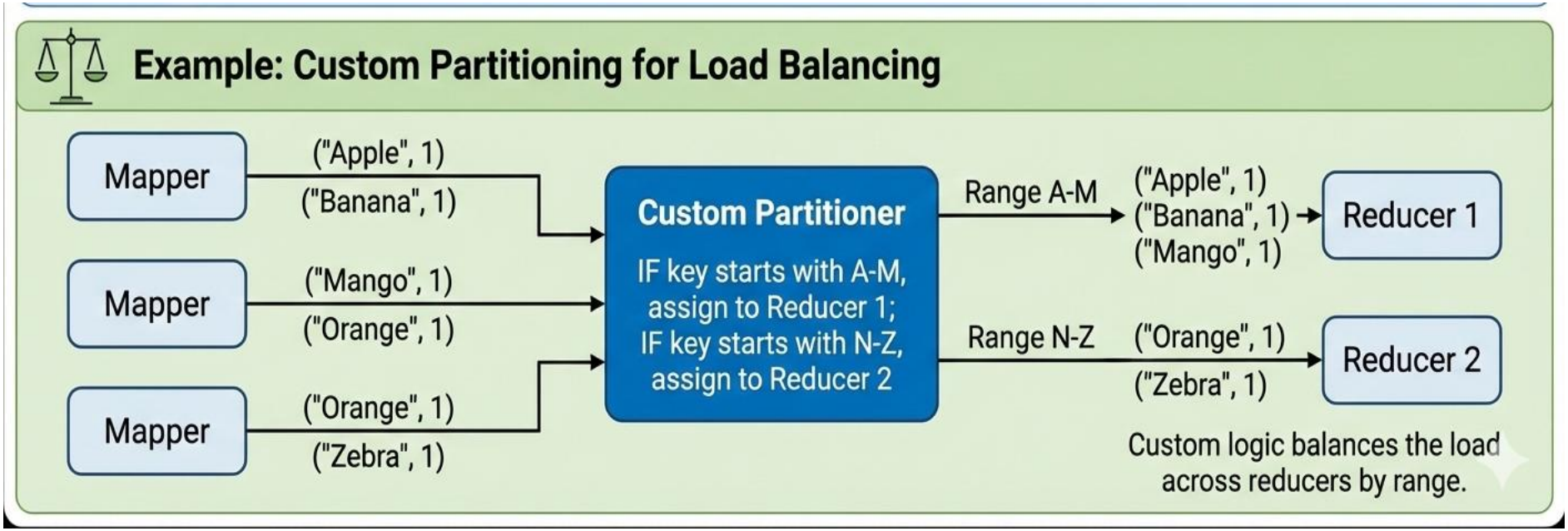
The combiner sums counts for each word on a single machine, reducing multiple pairs to a single pair containing the local count.

Combiner (Optional)

- **Functionality**:
  - Runs on the output of the mapper and performs a preliminary aggregation of data on the same machine as the mapper.
  - Runs **locally** on the same node as the mapper.
  - Helps optimize performance by minimizing the data sent to the reducers.
- **Example (Word Count)**: The combiner could sum up the counts for each word on a single machine, reducing multiple pairs like
  - ("word", 1) to a single pair ("word", local_count).

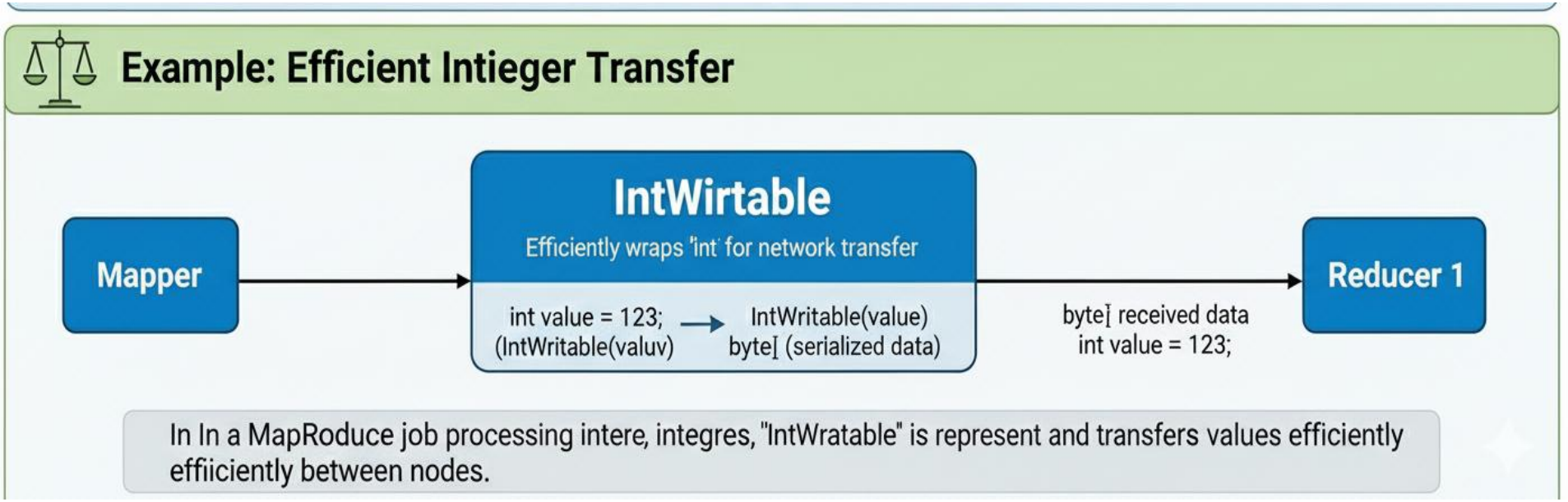
Partitioner (Optional) (Shuffle)

- **Purpose**:
  - The partitioner determines how intermediate key-value pairs from the mappers are assigned to specific reducers.
- **Functionality**:
  - Ensures that all values for a specific key go to the same reducer by assigning keys to reducers based on a hash function or custom logic.
  - Controls data distribution across reducers, which is especially useful when certain keys are more frequent than others.



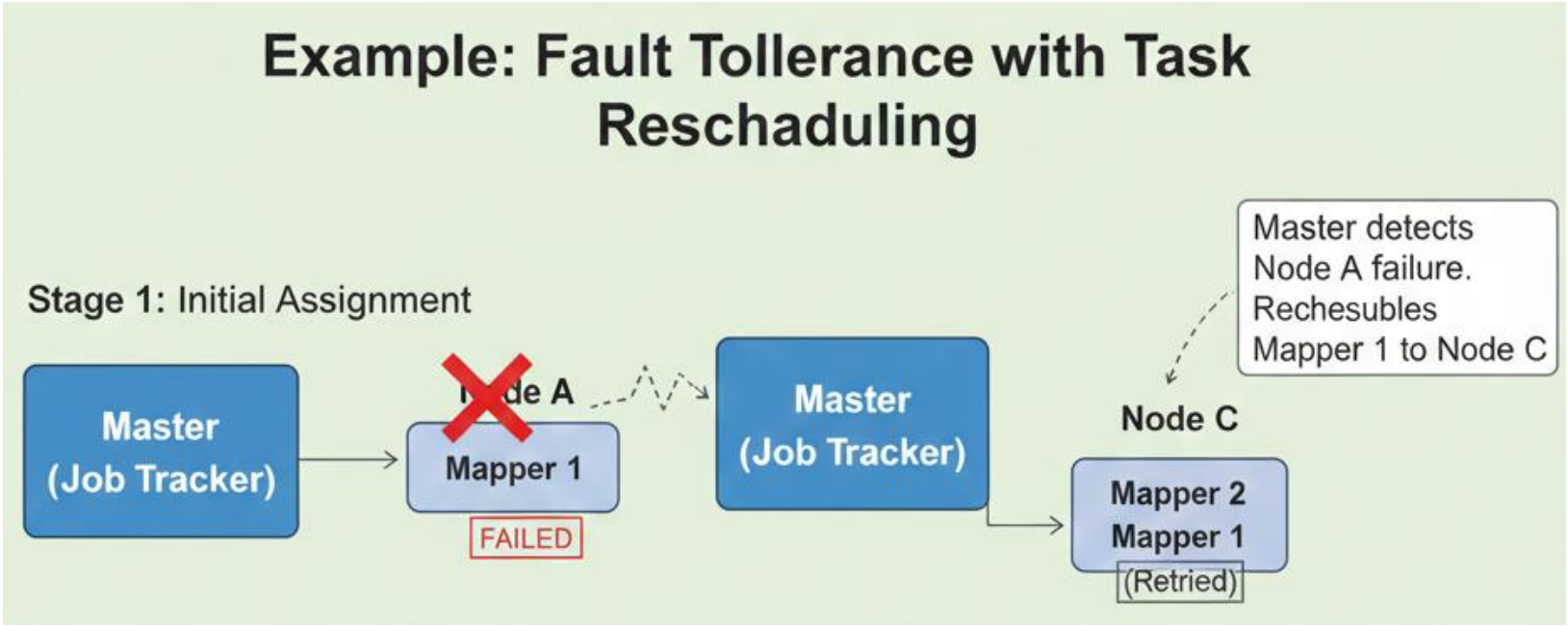
Writable(s) (Optional)

- **Purpose**: Writable are **data types in Hadoop's** Java-based MapReduce framework, designed to handle serialization and deserialization of data.
- **Functionality**:
  - Hadoop provides several Writable classes (e.g., *IntWritable, Text, LongWritable*) that wrap basic data types and support efficient data transmission.
  - Custom **writables** can be created to represent complex data structures.



System Components. Master (Job Tracker)

- **Purpose:**The master **component**, also known as the job **tracker**, coordinates the **execution** of a MapReduce job by **assigning** tasks on Nodes and **monitoring** their progress.
- **Functionality:** Manages **job scheduling**, **task distribution**, and **resource allocation**.
  - Tracks **task completion** or **failure** and **reassigns** tasks in case of node failure to ensure fault tolerance.

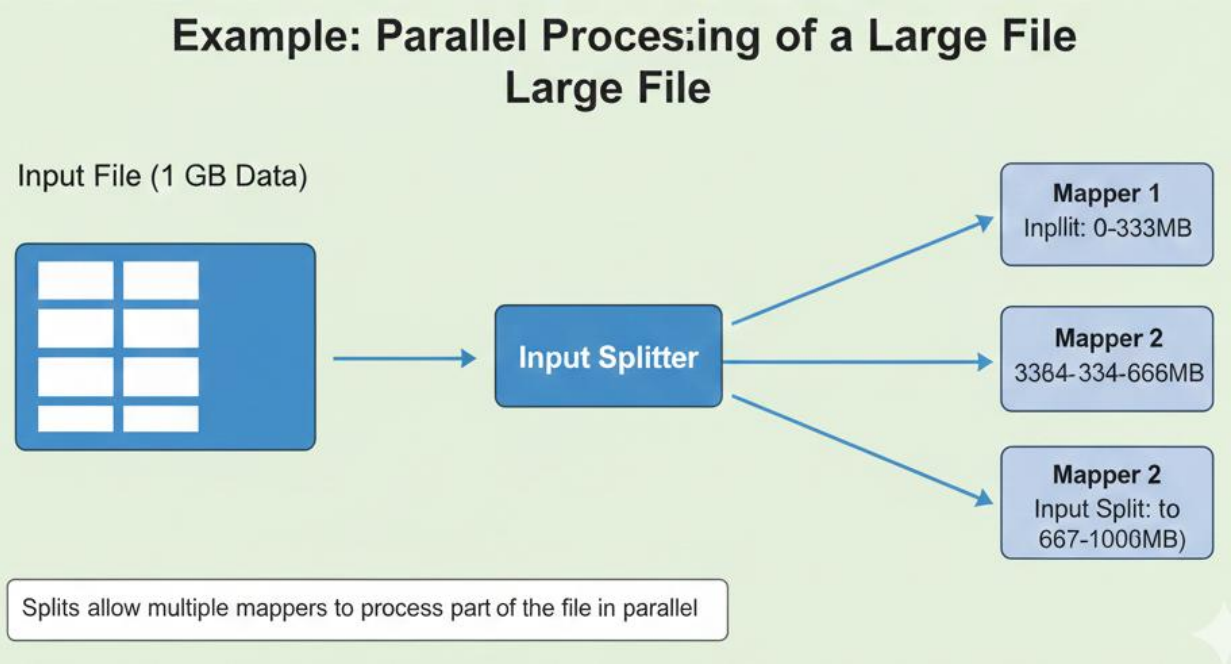


System Components: Input Splitter

Purpose: **Splits** the input file into "splits" based on factors such as file **size**, **format**, or the number of records.

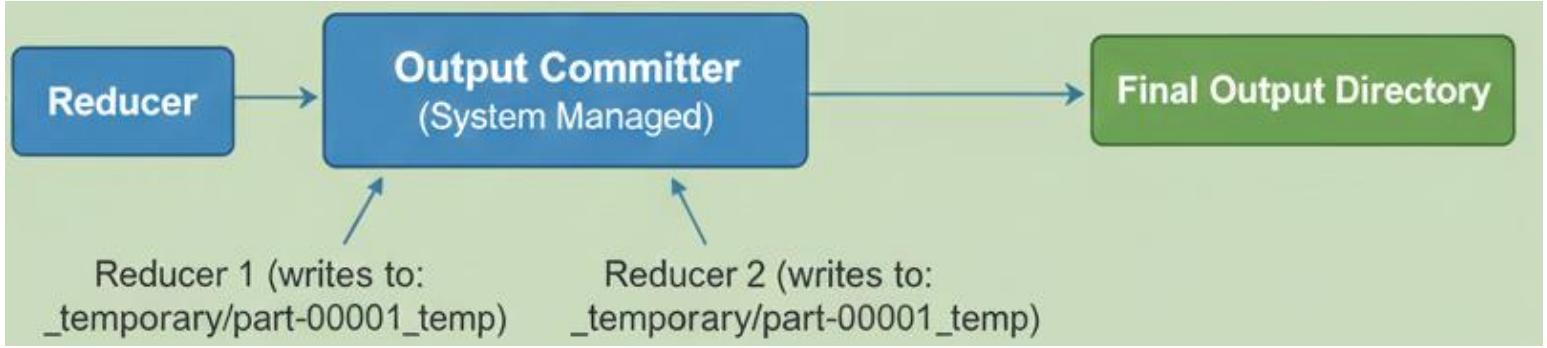
- Assigns each split to a mapper, ensuring that processing is distributed across multiple nodes.

- **Customization:**
  Users can implement custom input **splitting** logic for specialized file formats or unique data structures to optimize performance.



System Components: Output Committer

- **Purpose**: The output committer manages how the **final output** of **each reducer** is written to storage. Move Temporary files to final locations, insureing they are valid
- **Functionality**:
  - Ensures that partial or temporary output files generated by reducers are finalized and committed atomically.
  - Prevents errors in case of **partial output** (e.g., if a reducer fails during writing).
- **Customization**: Users can provide custom output committers if specific handling is required for the final output.



Major Components

- User Components:
  - Mapper
  - Reducer
  - Combiner (Optional)
  - Partitioner (Optional) (Shuffle)
  - Writable(s) (Optional)
- System Components:
  - Master
  - Input Splitter*
  - Output Committer*

* You can use your own if you really want!

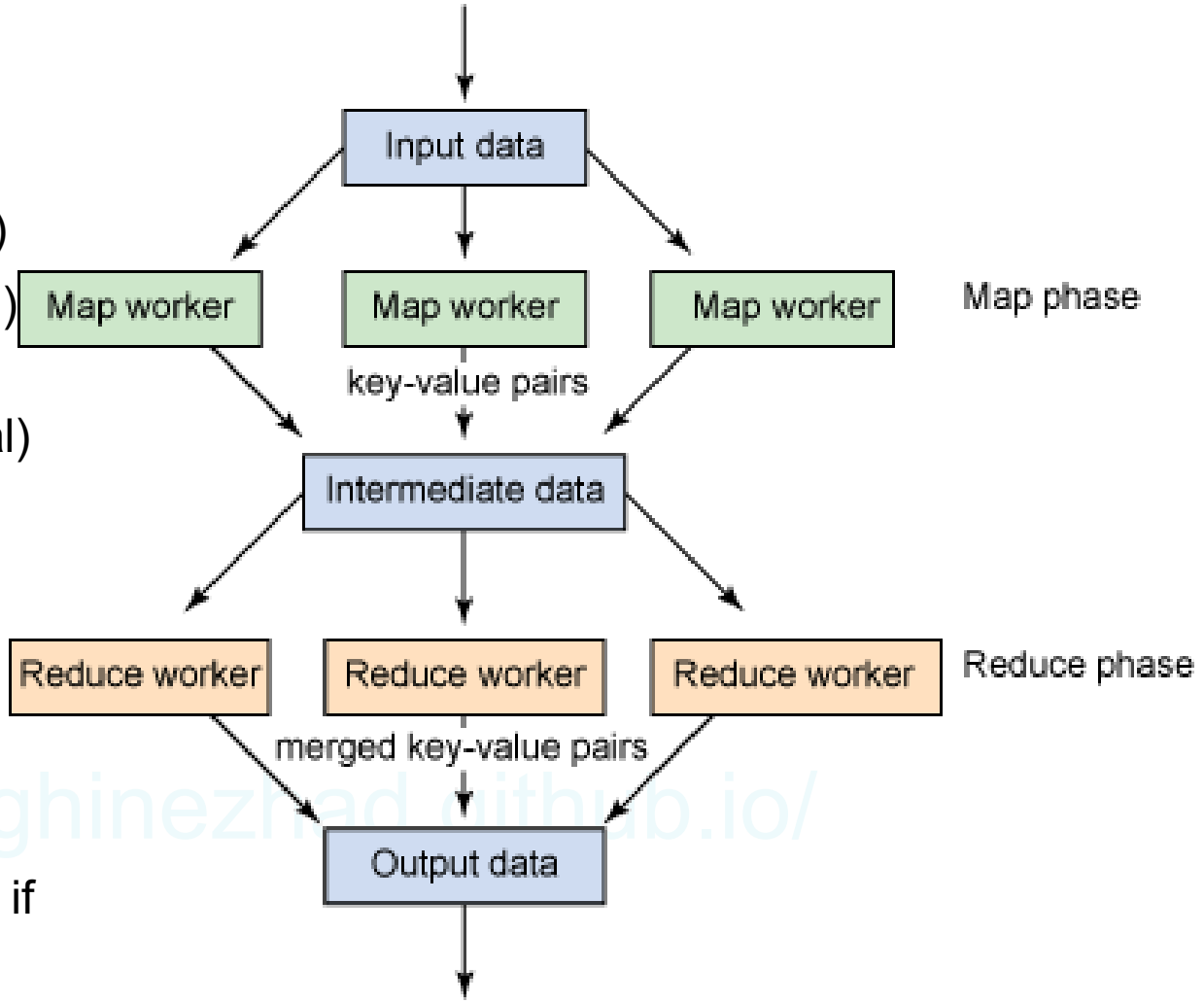


Image source: http://www.ibm.com/developerworks/java/library/l-hadoop-3/index.html

Key Notes

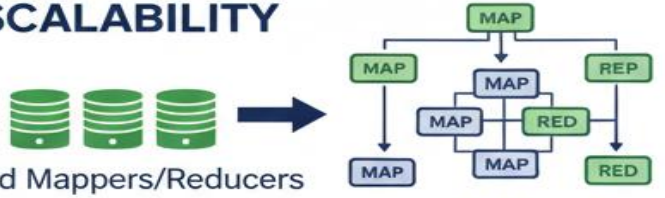
1 Task / Time

SINGLE-THREADDED & DETEMINTSTIC

Input → Process Output

Same Input = Same Output

SCALABILITY



Add Mappers/Reducers

Handle More Data

No Complex Multiithrrading

INDEPNDENCE

Operate Acrass Many Machines

EXECUTION

Slots (1/CPU Core)

Arbirary Machines

Paralelisim

ISOLATION

Mapper/ Task | Mapper/Reducer Task | Mapper/Reducer Task

Each in Own JVM → Stability & Isolation

Large

Programming Model

Map function

- Input: $\langle key_1, value_1 \rangle$
- Output: list of intermediate $\langle key_2, value_2 \rangle$ pairs
- Purpose: filtering, transformation, and local aggregation

Reduce function

- Input: $\langle key_2, list(value_2) \rangle$
- Output: $\langle key_3, value_3 \rangle$
- Purpose: global aggregation or summarization

Between these stages, the framework automatically performs:

- **Shuffle**: grouping values by identical keys
- **Sort**: ordering intermediate keys

Mapper Code: Java- word count example

- **Key (LongWritable):** Typically represents the file position, unused here.
- **Value (Text):** Represents content, e.g., a line from the document.
- **Tokenization:** tokenizeString(line) splits the line into words.
- **Output:** For each word, writes (word, 1) as the key-value pair.

```
public void map(LongWritable key, Text value, Context context) {
    String line = value.toString(); // Convert the input Text value to a string
    for (String part : tokenizeString(line)) { // Tokenize the line into words
        context.write(new Text(part), new LongWritable(1)); // Write the token and its count (1)
    }
}
```

Reducer Code

1. The **key** (Text) represents the token (word).
2. The **values** are the iterable collection of LongWritable objects, each holding a 1 that was produced by the Mapper for each occurrence of the token.
3. The reduce function sums up all the LongWritable values for that token.
4. The final result is a <Text, LongWritable> pair, where the key is the token, and the value is the total count of how many times that token appeared across the dataset.

```
public void reduce(Text key, Iterable<LongWritable> values, Context
context) {
    long sum = 0; // Initialize the sum variable.
    // Iterate through each LongWritable value (each count of the token).
    for (LongWritable val : values) {
        sum += val.get(); // Add the value of each occurrence to the sum.
    }
    // Write the final output (token and its total count) to the context.
    context.write(key, new LongWritable(sum));
}
```

Conclusion

- MapReduce provides a simple way to scale your application
- Scales out to more machines, rather than scaling up
- Effortlessly scale from a single machine to thousands
- Fault tolerant & High performance
- If you can fit your use case to its paradigm, scaling is handled by the framework