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?

- **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).

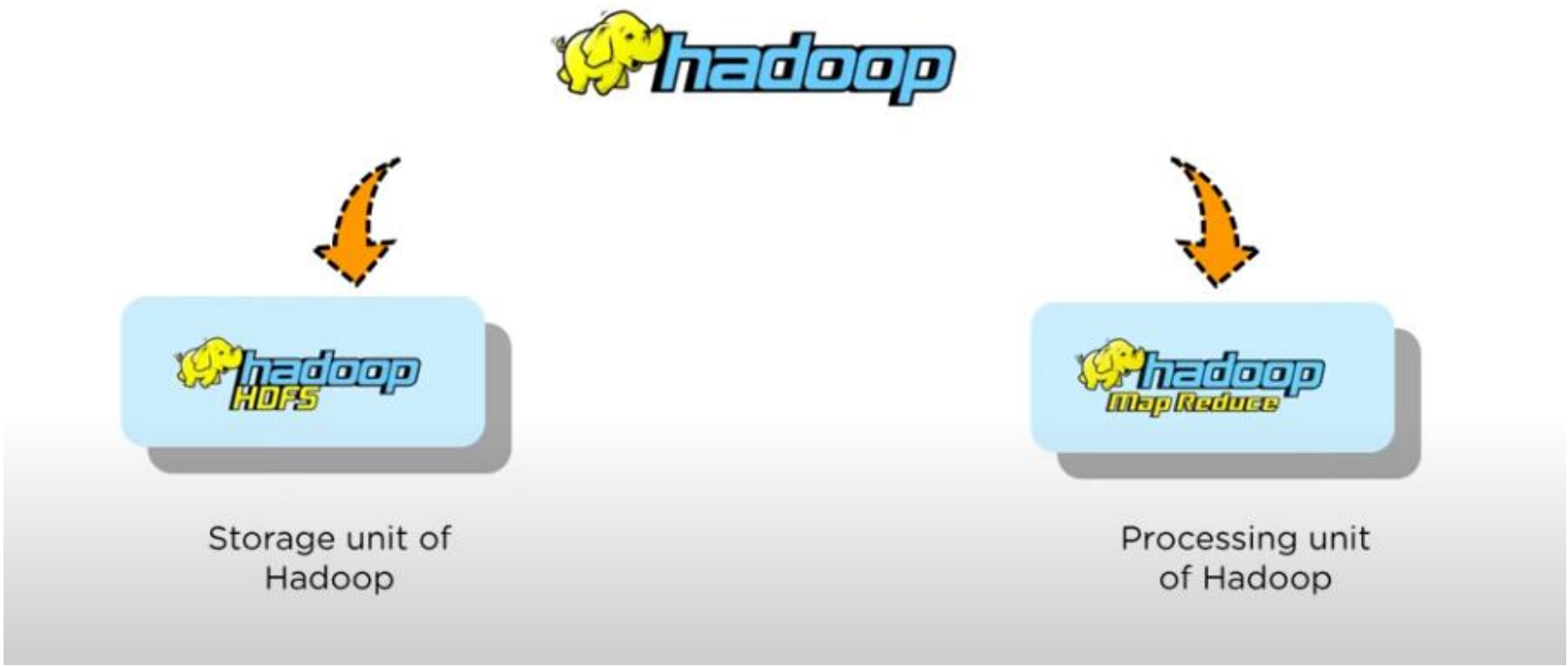
Understanding Distributed Data Processing with MapReduce

- **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

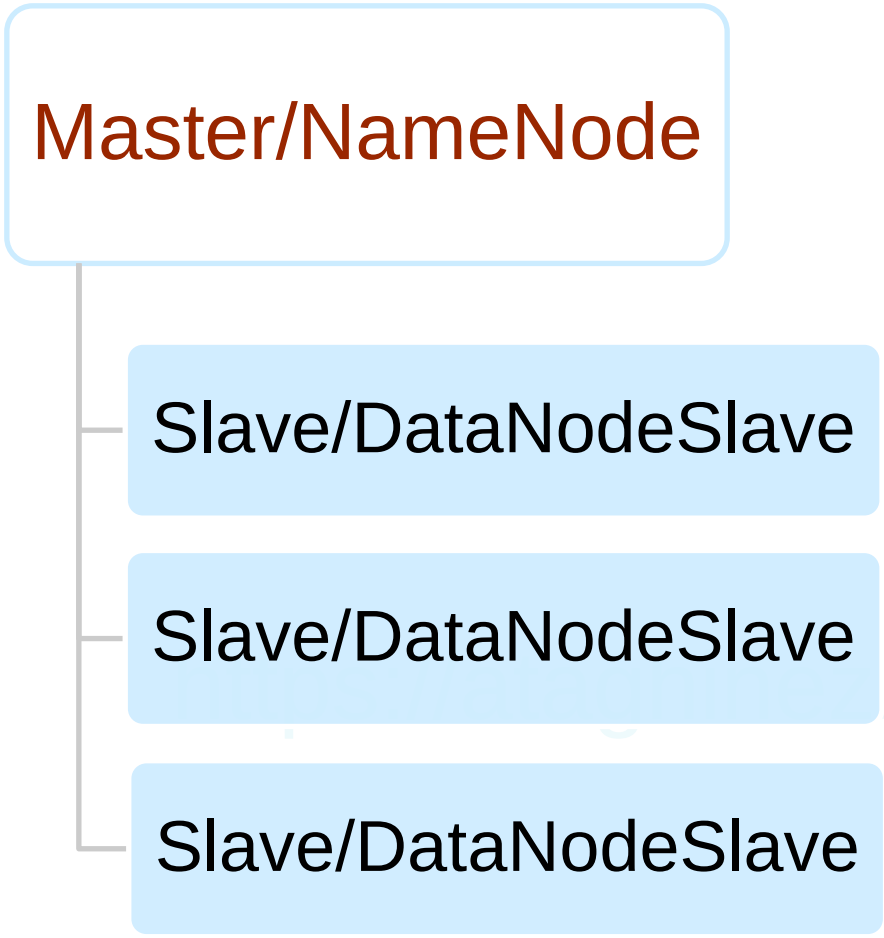
- **4) Machine Failures**
  - In the case of machine failure during the map or reduce operation, the operation is simply retried.
  - The system ensures that failure does not affect the overall computation.
- **5) Idempotency**
  - The system is **idempotent**, meaning that reapplying the map and reducing 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.

Components of Hadoop



HDFS components

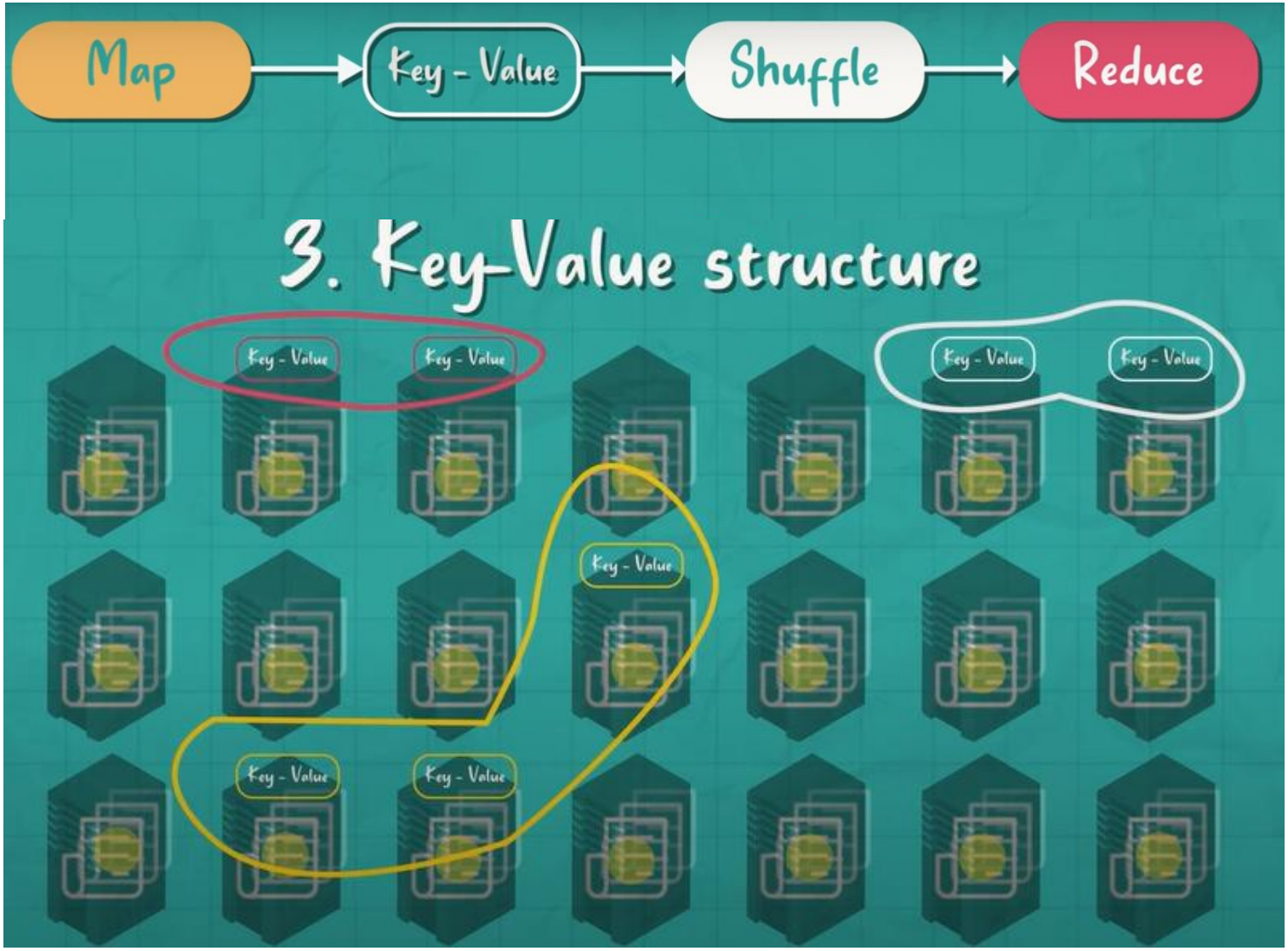
- Master/Slave nodes typically form the HDFS cluster



- 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.***

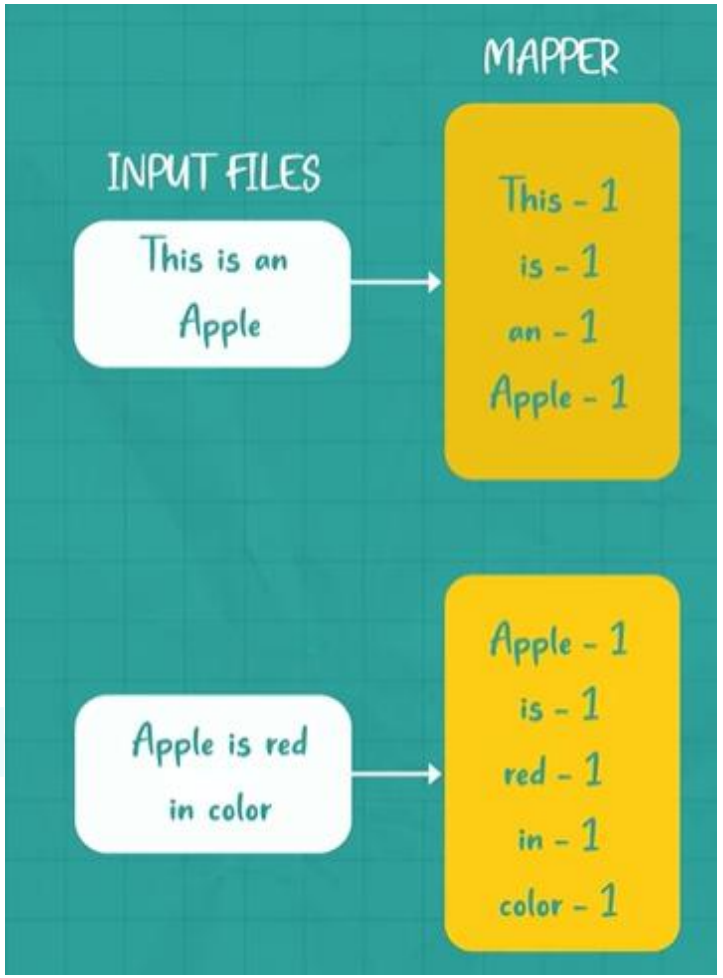
Word	Count
The	2
Doctor	1
Went	1
To	1
Store	1

- 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.

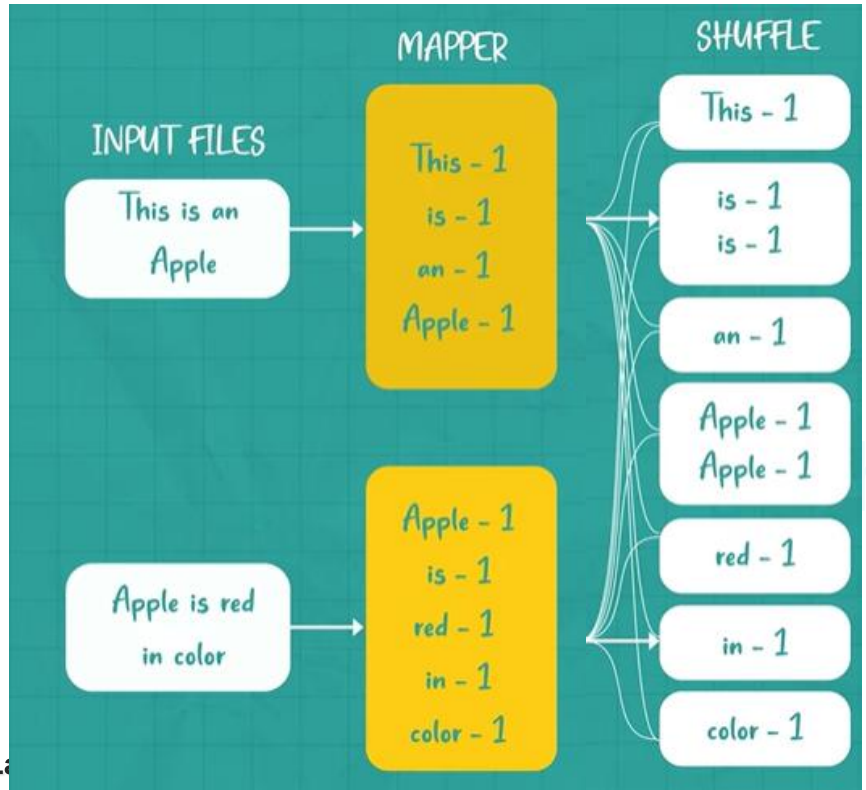
- **1. Mapper Phase**
  - **Input**: Each Mapper processes one or more files (or portions of them, called "splits") and extracts the words from these files.
  - **Key Output**: The word (e.g., "apple").
  - **Value**: The occurrence count, set to 1 for each instance of the word.



Example, Word count

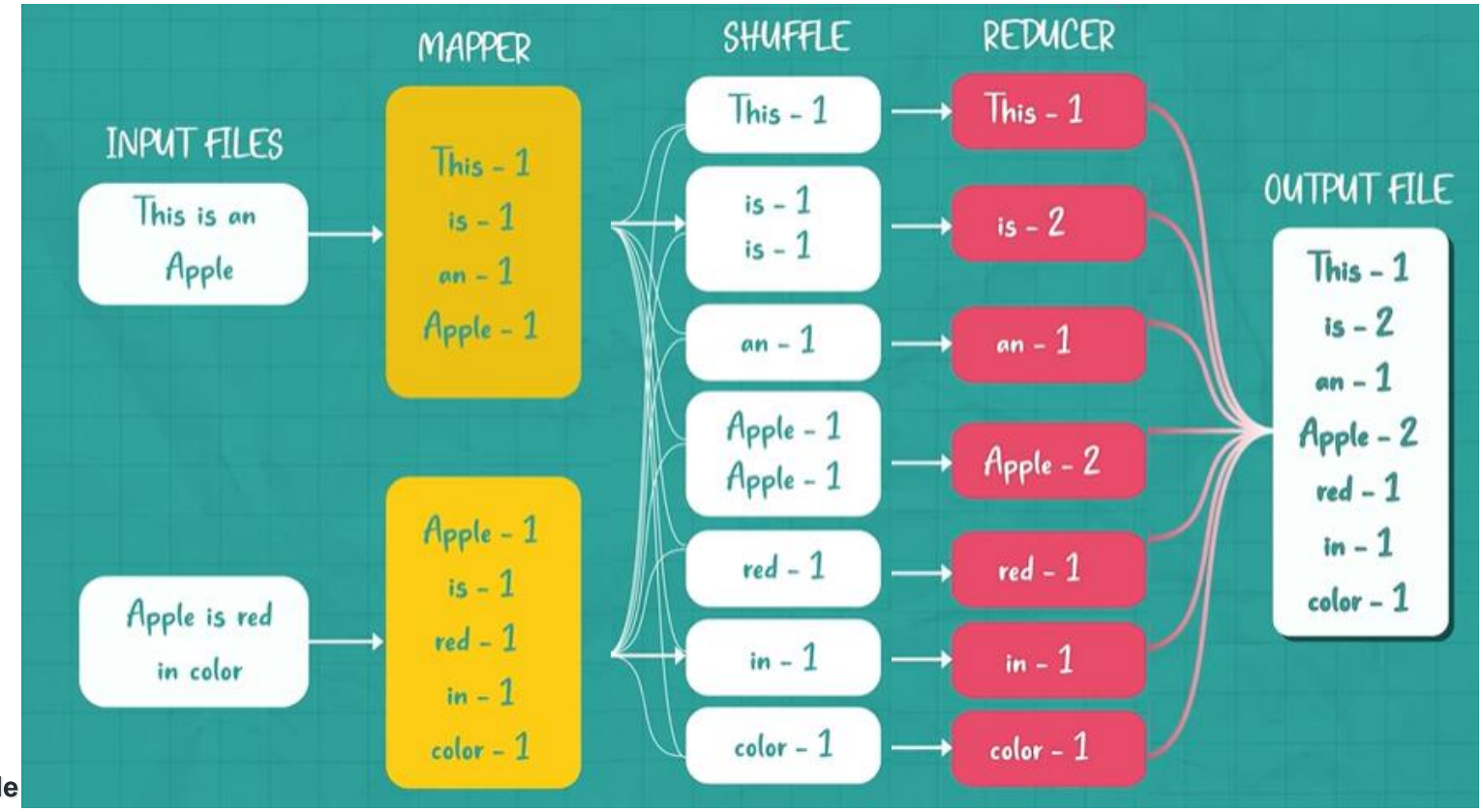
2. Shuffle and Sort Phase

- **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**: MapReduce can handle data of any size by splitting it across multiple nodes. This approach enables processing from KB to TB or even PB-sized data.
- **Parallel Processing**: Each mapper and reducer works independently on its data segment, allowing parallel execution and reducing the total processing time.
- **Fault Tolerance**: MapReduce is designed to recover from node failures. If a node fails during any phase, MapReduce can rerun the task on a different node.
- **Data Locality**: Mappers are often scheduled on nodes where the data chunks reside, minimizing data transfer across the network.

Outline

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

MapReduce vs Hadoop

- The paper is written by two researchers at Google, and describes their programming paradigm
- Open Source implementation is Hadoop MapReduce
  - Not developed by Google
  - Started by Yahoo

- Google's implementation (at least the one described) is written in C++
- Hadoop is written in Java

The Origins of MapReduce

- **Key Points from the Google MapReduce Paper:**
  - **Google's Needs**: The **Google researchers** needed a way to process and analyze massive **datasets**, such as **indexing** the **web**, **across distributed infrastructure**. The MapReduce paradigm was born out of the necessity to efficiently manage data processing at an enormous scale.
  - **Programming Model**: The paper introduces a simple model with two primary functions –**Map** and **Reduce** - that make it easier to split tasks into parallel operations, allowing for scalability and fault tolerance.
  - **Implementation at Google**: The original MapReduce implementation described in the paper was developed internally at Google in C++. This system was proprietary to Google and not available for public use, but it demonstrated a model that would inspire similar frameworks.
- **Why MapReduce Wasn't Publicly Available**

Hadoop MapReduce: The Open-Source Alternative

- To make MapReduce accessible outside of Google, **Yahoo! spearheaded an open-source implementation known as Hadoop MapReduce**. Hadoop brought the power of MapReduce to everyone and became an industry standard for processing big data on distributed systems.
  - **Hadoop's Beginnings**: Originally initiated as part of the Apache Nutch search engine project, Hadoop's development was taken on by Yahoo, which aimed to create a scalable and reliable framework similar to Google's MapReduce. Hadoop has since grown into a full ecosystem of tools under the Apache Foundation.
  - **Written in Java**: Unlike Google's C++ implementation, Hadoop MapReduce is written in Java. This difference makes Hadoop portable across various operating systems and platforms, leveraging Java's platform independence.
  - **Industry Impact**: Hadoop MapReduce has been widely adopted across industries due to its open-source nature, enabling organizations to manage large datasets without needing proprietary tools or infrastructure.

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.

GFS/HDFS

- **Distributed Filesystems (GFS and HDFS)**
  - **Definition**: Distributed filesystems like the Google File System (GFS) and Hadoop Distributed File System (HDFS) are specialized storage systems that manage data across multiple machines in a network.
  - **Purpose in MapReduce**: These filesystems are designed to handle the storage and retrieval of massive datasets, supporting the large-scale, distributed data processing model of MapReduce.
- A few concepts are key to MapReduce though:
  - Google File System (GFS) and Hadoop Distributed File System (HDFS) are essentially distributed file-systems ?
  - Are fault tolerant through replication?
  - Allows data to be local to computation ?

Outline

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

Major Components

- MapReduce relies on two types of components:
  - **User Components**: These are customizable parts of the MapReduce program that the user defines to control the specific data processing logic.
    - Mapper
    - Reducer
    - Combiner (Optional)
    - Partitioner (Optional) (Shuffle)
    - Writable(s) (Optional)
  - **System Components**: These are built-in components managed by the system to orchestrate and coordinate the MapReduce job
    - Master
    - Input Splitter*
    - Output Committer*

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

Combiner (Optional)

- **Purpose**: The combiner is an optional component that acts as a "mini-reducer" to reduce the amount of data that needs to be shuffled across the network.
- **Functionality**:
  - Runs on the output of the mapper and performs a preliminary aggregation of data on the same machine 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.
- **Example**: For a large dataset, a custom partitioner might assign certain ranges of words (e.g., words starting with "A-M" to one reducer, "N-Z" to another) to balance the load.

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.
- **Example**: In a MapReduce job processing integers, *IntWritable* might be used to represent and transfer integer values efficiently.

System Components

- These components are integral to the MapReduce framework, managing the core processes required to execute a MapReduce job.
- **1. Master (Job Tracker)**
  - **Purpose:**
    - The master component, also known as the job tracker, coordinates the execution of a MapReduce job by assigning tasks 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.
  - **Example:**
    In Hadoop MapReduce, the job tracker decides which mapper and reducer tasks run on which nodes. If a node fails, it reschedules the tasks to another available node.

System Components

- **2. Input Splitter**
  - **Purpose:**
    - The input splitter **divides** the **input dataset** into smaller, manageable chunks for the mappers, enabling parallel processing.
  - **Functionality:**
    - 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 (Cont.)

- **3. Output Committer**
  - **Purpose**: The output committer manages how the final output of each reducer is written to storage.
  - **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!

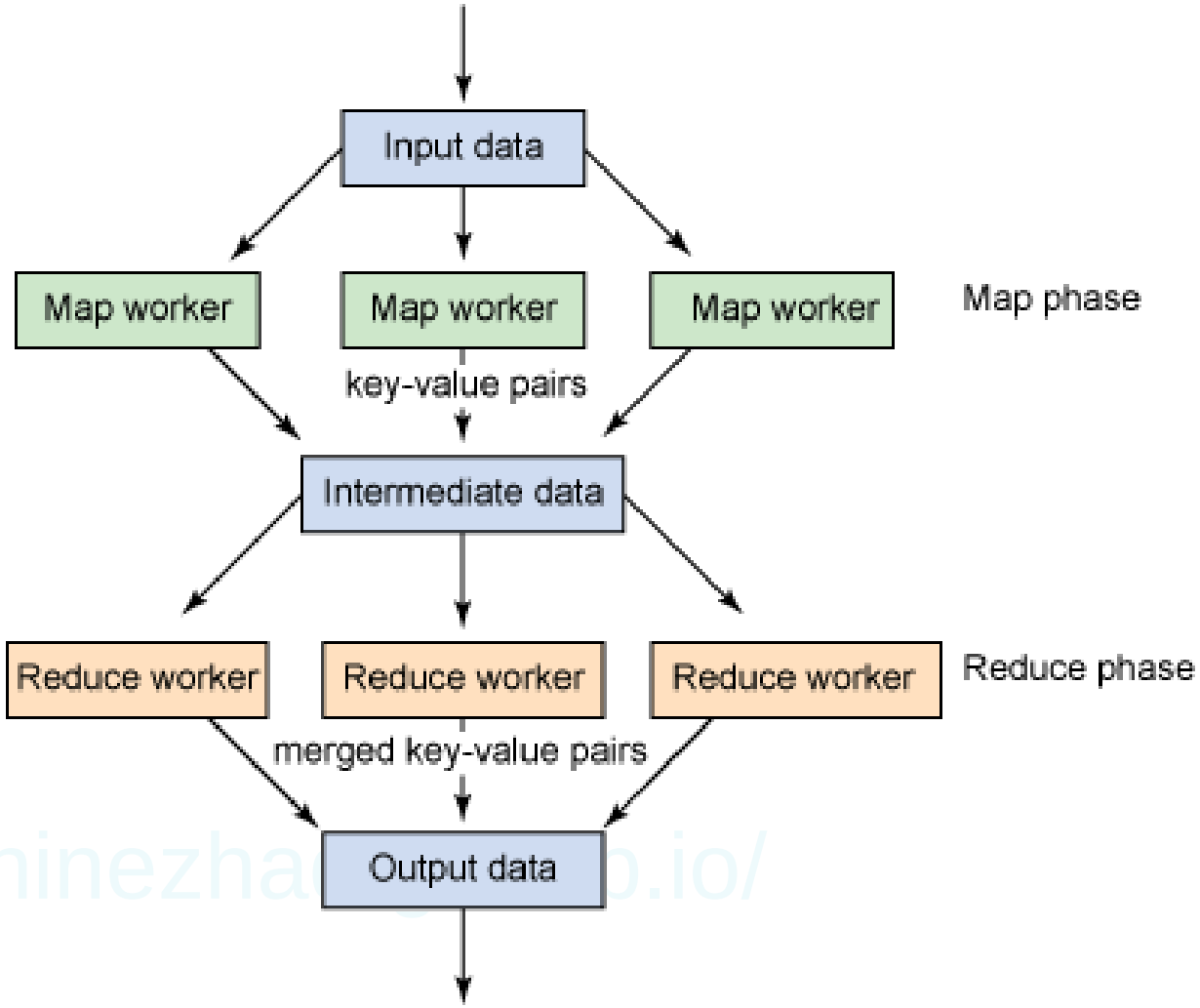
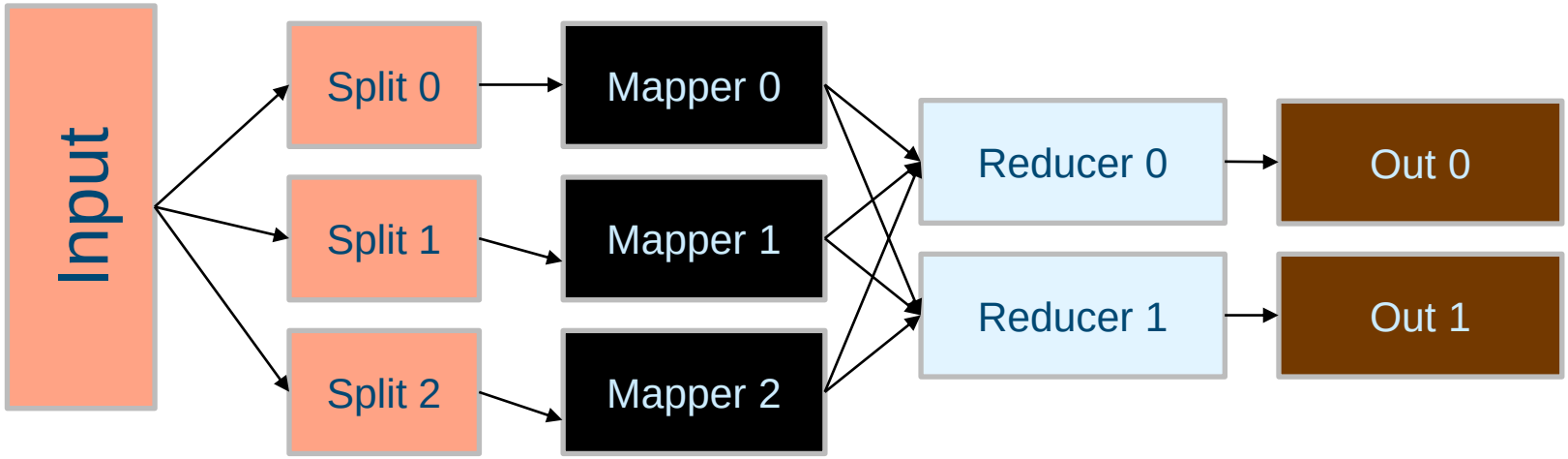


Image source: http://www.ibm.com/developerworks/java/library/l-hadoop-3/index.html

Key Notes

- **Mappers and Reducers are Single-Threaded:** They work on one task at a time and are deterministic, meaning they always produce the same output for the same input.
- **Determinism:** Ensures failed jobs can be restarted or executed speculatively (on multiple machines for reliability).
- **Scalability:** To handle more data, simply add more Mappers and Reducers without worrying about writing complex multithreaded code.
- **Independence:** Each Mapper and Reducer works independently, allowing you to use almost any number of machines.
- **Execution:** Mappers and Reducers run on arbitrary machines, with each machine typically having multiple slots (usually one per CPU core) for running them.
- **Isolation:** In Hadoop, Mappers and Reducers run in their own Java Virtual Machines (JVMs), ensuring process isolation and stability.

Data Flow



Input Splitter

- Is responsible for splitting your input into multiple chunks
- These chunks are then used as input for your mappers
- Splits on logical boundaries. The default is 64MB per chunk
  - Depending on what you're doing, 64MB might be a LOT of data! You can change it
- Typically, you can just use one of the built in splitters, unless you are reading in a specially formatted file

Mapper

- **Example:** In a Word Count job:
- **Input:** The text:
  `"The teacher went to the store. The store was closed; the store opens in the morning. The store opens at 9am."`
- **Output:** The Mapper splits the text into words and emits key-value pairs, with each word as a key and `1` as its value:
  `<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>`

- This intermediate output is later grouped and aggregated by Reducers to produce the final result.

Reducer

- Accepts the Mapper output, and collects values on the key
  - All inputs with the same key *must* go to the same reducer!
- Input is typically sorted, output is output exactly as is
- For our example, the reducer input would be:
  - <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>
- The output would be:
  - <The, 6> <teacher, 1> <went, 1> <to, 1> <store, 3> <was, 1> <closed, 1> <opens, 1> <morning, 1> <at, 1> <9am, 1>

Combiner

- Essentially an intermediate reducer
- Is optional
- Reduces output from each mapper, reducing bandwidth and sorting
- Cannot change the type of its input
  - Input types must be the same as output types

Output Committer

- Is responsible for taking the reduce output, and committing it to a file
- Typically, this committer needs a corresponding input splitter (so that another job can read the input)
- Again, usually built in splitters are good enough, unless you need to output a special kind of file

Partitioner (Shuffler)

- Decides which pairs are sent to which reducer
- Default is simply:
  - Key.hashCode() % numOfReducers

- User can override to:
  - Provide (more) uniform distribution of load between reducers
  - Some values might need to be sent to the same reducer
    - Ex. To compute the relative frequency of a pair of words <W1, W2> you would need to make sure all of word W1 are sent to the same reducer
  - Binning of results

Master

- Responsible for scheduling & managing jobs
- Scheduled computation should be close to the data if possible
  - Bandwidth is expensive! (and slow)
  - This relies on a Distributed File System (GFS / HDFS)!
- If a task fails to report progress (such as reading input, writing output, etc), crashes, the machine goes down, etc, it is assumed to be stuck, and is killed, and the step is re-launched (with the same input)
- The Master is handled by the framework, no user code is necessary

Master

- The **Master** in a distributed framework is responsible for scheduling and managing jobs, ensuring efficient and reliable execution.
- **Key Responsibilities:**

1. **Job Scheduling:**
    1. Tasks are scheduled to run close to the data whenever possible to minimize bandwidth usage, as bandwidth is expensive and slow.
    2. This relies on the use of a **Distributed File System** like GFS or HDFS, which ensures that data is distributed and accessible across nodes.
2. **Fault Tolerance:**
    1. If a task fails to make progress (e.g., during input reading, output writing, or due to machine failure), it is assumed to be stuck.
    2. The task is terminated and restarted on another node using the same input.
    3. The deterministic nature of tasks ensures consistency when they are restarted.
3. side effects to ensure that retries or duplicates do not cause inconsistencies in the system.

Master (Cont.)

3. **Data Replication:**
    3. HDFS can replicate data to ensure it is locally available for tasks when needed, improving efficiency.
4. **Handling Slow Nodes:**
    1. If a node is completing its task too slowly (due to hardware issues, network delays, etc.), the Master may launch a duplicate of the task on another node.
    2. The first one to finish is accepted, and the duplicate task is terminated.
5. **Simplicity for Users:**
    1. The Master and its responsibilities are handled by the framework. Users do not need to write additional code for scheduling or fault management.
6. **No Side Effects:**
    1. Tasks should have no side effects to ensure that retries or duplicates do not cause inconsistencies in the system.

Writables

Writable types in Hadoop are classes designed for serialization and deserialization of data to and from a stream. They are crucial for managing input, output, and intermediate data in the Hadoop framework.

-Key Features:

1. **Serialization/Deserialization:**
   Writable types can convert data into a stream for storage or transfer and reconstruct it when needed.
2. **Framework Requirement:**
   The Hadoop framework requires Writable types for input and output because it serializes data before writing it to disk or transferring it across nodes.
3. **Custom Writables:**
   Users can implement the Writable interface to create custom data types for their specific input, output, or intermediate values.
   This allows flexibility in handling complex data structures like arrays, maps, or user-defined objects.

Writables

4. **Default Writable Types:**
   Hadoop provides default Writable classes for basic types, such as Strings (Text), Integers (IntWritable), and Longs (LongWritable).
5. **Usage in Applications:**
   A typical MapReduce application requires six Writables:
   1. **2 for Input:** To represent the key-value pairs read by the mappers.
   2. **2 for Intermediate Values:** To transfer data between the mappers and reducers.
   3. **2 for Output:** To write the final key-value pairs to the output.

Outline

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

Mapper Code: Java

- The `key` (LongWritable) is not used in this example but would typically represent the position in the file.
- The `value` (Text) is the content (like a line from the text document).
- The function `tokenizeString(line)` is assumed to break the line into individual tokens (words).
- For each token (word), the `map` function writes an output pair with the token as the key and the value `1` (count) as the value.

```
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));
}
```

Combiner Code

- *What is a Combiner?*
  - A **Combiner** is an optional component in Hadoop MapReduce, used to reduce the amount of data transferred between the **Mapper** and **Reducer**.
  - The combiner runs **locally** on the Mapper output, performing partial aggregation before sending the data to the Reducer. This reduces the bandwidth needed for transferring data between the Mapper and Reducer.
  - *Do We Need a Combiner?*
- **No, a combiner is not required**, but it can significantly **reduce bandwidth**.
  - The **Combiner** is essentially a "mini-reducer" that runs on the output of the Mapper and performs partial aggregation, minimizing the amount of data sent to the Reducer.

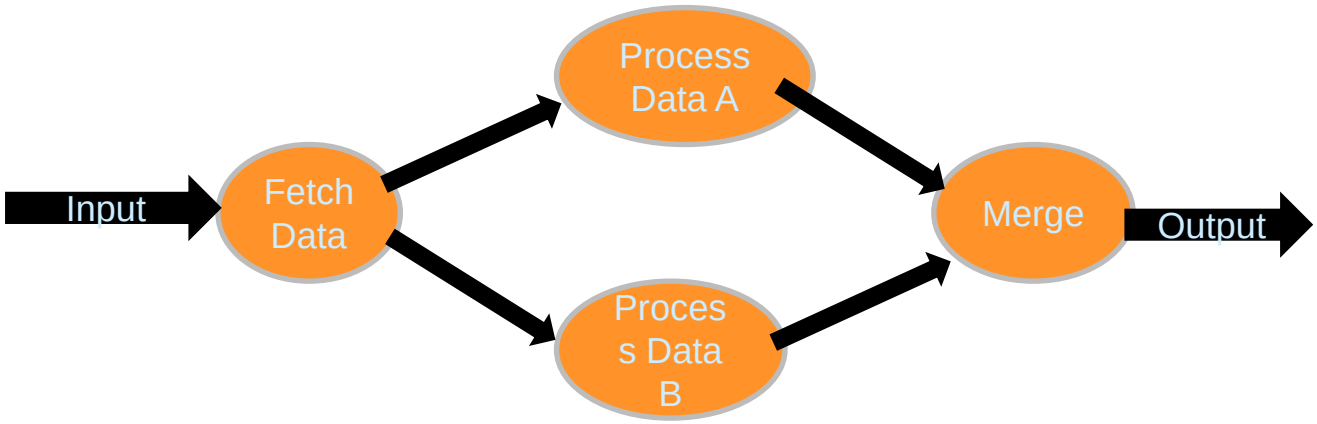
Combiner Code

- *Can the Reducer Act as a Combiner?*
  - **Yes**, in some cases, the same function that you use in the **Reducer** can also act as the **Combiner**.
    - In practice, if the logic of your **Reducer** can be applied locally to the Mapper output, it will help in reducing the volume of data transferred.
    - In the case of simple aggregations like **sum** or **count**, the **Reducer's** code is identical to the **Combiner's** code. You don't need to define a separate combiner.
- *Simple Runner Class for MapReduce*
  - To run the above logic (Combiner, Mapper, Reducer), all that is needed is a **simple runner class**.
    - This class **specifies the components** (Mapper, Reducer, Combiner) to be used in the job.
    - It also defines **input/output directories** for the MapReduce job.

Workflows

- Sometimes you need multiple steps to express your design
- MapReduce does not directly allow for this, but there are solutions that do
- Hadoop YARN allows for a Directed Acyclic Graph of nodes
- Oozie also allows for a graph of nodes

Handling Data By Type



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