Module 2

Q. Write a MapReduce pseudo code for word count problem. Apply MapReduce working on the following document:

"This is NoSQL. NoSQL handles complex data."

=>

MapReduce is a programming model used for processing large datasets in a distributed environment. It consists of two main phases:

- **Map phase:** Splits input data into key-value pairs and processes them independently.
- **Reduce phase:** Aggregates intermediate results by combining values associated with the same key.

This model ensures scalability and fault tolerance while performing parallel computations.

MapReduce Pseudo Code for Word Count

```
Map(key, value):
   // key: line number, value: line content
   for each word w in value:
      emit(w, 1)

Reduce(key, values):
   // key: word, values: list of counts
   sum = 0
   for each count c in values:
      sum = sum + c
   emit(key, sum)
```

Step-by-step working on the document

Input document: This is NoSQL. NoSQL handles complex data.

Splitting into words (ignoring case and punctuation):
→ [this, is, nosql, nosql, handles, complex, data]

Map Phase Output:

(this, 1)
(is, 1)
(nosql, 1)
(nosql, 1)

(handles, 1)
(complex, 1)
(data, 1)

Shuffle & Sort Phase:
Group by key (word):

this → [1]
is → [1]
nosql → [1, 1]
handles → [1]
complex → [1]
data → [1]

Reduce Phase Output:

(this, 1)
(is, 1)
(nosql, 2)
(handles, 1)
(complex, 1)
(data, 1)

Final Word Count Result:
this → 1
is → 1
nosql → 2
handles → 1
complex → 1
data → 1

Q. MapReduce Execution Pipeline

=>

1. Introduction

MapReduce is a programming model for distributed data processing using two main phases: Map and Reduce. It simplifies complex tasks by breaking them into independent parallel operations.

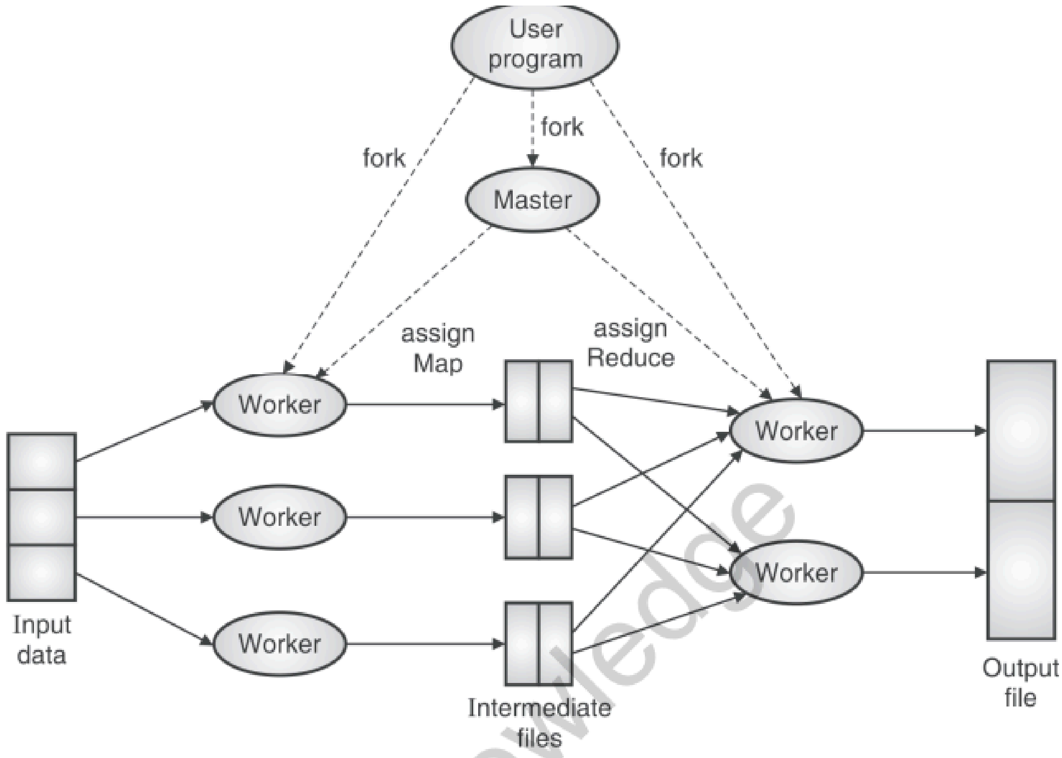

Fig. 3.2.3 : MapReduce program execution

2. Components of MapReduce

- **Input Data**
    1. Stored in HDFS and split into fixed-size blocks.
    2. Ensures parallel processing by assigning each block to a mapper.
- **Mapper**
    1. Processes input data and generates <key, value> pairs.
    2. Operates independently, allowing parallel execution on all nodes.

- **Combiner**
    1. Performs local aggregation on mapper output.
    2. Reduces data transfer between mapper and reducer.
- **Partitioner**
    1. Distributes intermediate keys among reducers.
    2. Ensures balanced load by assigning keys to specific reducers.
- **Reducer**
    1. Aggregates values for each unique key.
    2. Produces final summarized results.
- **Output Format**
    1. Writes the reduced results back into HDFS.
    2. Stores data in structured formats like text or sequence files.

3. MapReduce Execution Pipeline

Step 1 – Input Splitting

The input file is divided into splits (e.g., 128MB blocks), each processed by a separate map task.

Step 2 – Mapping Phase

The mapper reads input records and emits intermediate key-value pairs.

Pseudo-code:

```
map(key, value):
   for word in value:
      emit(word, 1)
```

Step 3 – Shuffling and Sorting

Intermediate data is grouped by keys and sorted before being passed to reducers.

Step 4 – Reducing Phase

The reducer combines values for each unique key to produce the final result.

Pseudo-code:

```
reduce(key, values):
   sum = 0
   for val in values:
      sum += val
   emit(key, sum)
```

4. Example: Word Count Problem

Input Document

"This is NoSQL. NoSQL handles complex data."

Map Phase Output

```
("This",1)
("is",1)
("NoSQL",1)
("NoSQL",1)
("handles",1)
("complex",1)
("data",1)
```

Shuffle and Sort Output

```
"This"    -> [1]
"is"      -> [1]
"NoSQL"   -> [1,1]
"handles" -> [1]
"complex" -> [1]
"data"    -> [1]
```

Reduce Phase Output

```
This      1
is        1
NoSQL     2
handles   1
complex   1
data      1
```

Q. Natural Join and Grouping & Aggregation using MapReduce

=>

1. Introduction

- **Relational algebra** operations like *natural join* and *grouping with aggregation* are fundamental for combining and summarizing data in databases.

- **MapReduce**, a programming model by Google, is commonly used to process huge datasets in parallel by breaking tasks into **Map** and **Reduce** phases.

- Using MapReduce, these relational operations can be implemented efficiently on distributed systems.

2. Natural Join in Relational Algebra

Definition

- A **natural join** combines two relations (tables) based on **common attribute names**.

- It automatically matches rows where the values of these common attributes are equal.

- Output contains **all attributes from both tables**, but the common attributes appear only once.

Example

Consider two tables:
Employee(EmpID, Name, DeptID)
Department(DeptID, DeptName)

- A natural join will link employees to their department names by matching **DeptID**.

- Result: **(EmpID, Name, DeptID, DeptName)**

Implementing Natural Join with MapReduce

Step 1: Map Phase

- **Input:** Both Employee and Department tables are given as input.

- **Mapper task:**
  - For every record, **emit key = join attribute (DeptID)**.
  - Value = tuple with source tag to identify which table it came from.
- **Example Output:**
  - From Employee: (DeptID=10, [E, EmpID=1, Name="A"])
  - From Department: (DeptID=10, [D, DeptName="HR"])

Step 2: Shuffle and Sort

- MapReduce automatically **groups values by key (DeptID)** so that all records with the same DeptID come together.

Step 3: Reduce Phase

- For each key (DeptID), the reducer:
  - Combines employee records with department records.
  - Emits joined tuples with attributes from both tables.
- **Output Example:** (1, "A", 10, "HR")

Advantages:

- Can handle very large datasets distributed over many machines.
- Automatically takes care of parallelism and fault tolerance

3. Grouping and Aggregation in Relational Algebra

Definition

- **Grouping:** Divides tuples into groups based on an attribute (e.g., group employees by DeptID).
- **Aggregation:** Applies functions like **COUNT, SUM, AVG, MAX, MIN** to each group.

Example

If we have:
 Employee(EmpID, Name, Salary, DeptID)

- To find *total salary per department*:
  - Group by **DeptID**
  - Apply **SUM(Salary)**

Result: **(DeptID, TotalSalary)**

Implementing Grouping & Aggregation with MapReduce

Step 1: Map Phase

- **Input:** Employee table records.
- **Mapper task:**
  - Emit key = group attribute (DeptID).
  - Emit value = measure to aggregate (Salary).
- **Example Output:** (DeptID=10, 50000), (DeptID=10, 60000)

Step 2: Shuffle and Sort

- MapReduce groups all salaries for the same DeptID automatically.

Step 3: Reduce Phase

- For each DeptID, the reducer:
  - Sums up salaries (or applies other aggregation function).
  - Emits (DeptID, TotalSalary)
- **Output Example:** (10, 110000)

Advantages:

- Handles massive datasets with parallel computation.
- Can be extended to multiple aggregation functions simultaneously.

4. Key Points

Natural Join with MapReduce

- Mapper emits join keys (common attributes).
- Reducer combines records from both tables having the same key.

Grouping and Aggregation with MapReduce

- Mapper emits group key and measure.
- Reducer performs aggregation (SUM, COUNT, AVG, etc.) on grouped values.

Q. Selection and Projection using MapReduce

=>

1. Introduction

- **Relational algebra** defines core operations for manipulating data in a database.
- Two basic operations are **selection** (filtering rows) and **projection** (selecting columns).
- Using **MapReduce**, these operations can be scaled to handle **very large datasets** distributed across many machines.

2. Selection in Relational Algebra

Definition

- **Selection (σ)** chooses **specific rows** from a relation (table) that satisfy a **given condition**.

- It filters records but **does not change columns**.

Example

If we have:
Employee(EmpID, Name, Salary, DeptID)

- Query: *Find employees with Salary > 50,000*

- **Selection operation:** σ(Salary > 50000)(Employee)

- **Result:** Only rows meeting this condition are returned.

Implementing Selection with MapReduce

Step 1: Map Phase

- **Input:** All records of the Employee table.

- **Mapper task:**

    - For every record, check if it satisfies the selection condition.

    - If **true**, emit the entire tuple as key-value or simply pass it forward.

    - If **false**, emit nothing (record is discarded).

- **Example Output:** (null, [EmpID=1, Name="A", Salary=60000, DeptID=10])

Step 2: Shuffle and Sort

- Since selection only filters rows, **no special grouping is needed**.
- MapReduce still sorts and groups intermediate keys automatically, though here keys may be null.

Step 3: Reduce Phase

- In many cases, **reducer is optional** because filtering is already done by the mapper.
- If used, reducer just passes through the filtered records.

Advantages:

- Filtering happens **in parallel** on each mapper.
- No need for complex reduce logic.

3. Projection in Relational Algebra

Definition

- **Projection (π)** chooses **specific columns (attributes)** from a relation.
- It removes unwanted columns and may eliminate **duplicate rows**.

Example

If we have:
Employee(EmpID, Name, Salary, DeptID)

- Query: *Get only employee names and department IDs*
- **Projection operation:** π(Name, DeptID)(Employee)
- **Result:** Only these columns are returned.

Implementing Projection with MapReduce

Step 1: Map Phase

- **Input:** All records of Employee table.
- **Mapper task:**
  - For each record, emit only the required columns as the value.
  - Key can be **null** or the projected tuple if duplicates need to be removed.
- **Example Output:** (null, [Name="A", DeptID=10])

Step 2: Shuffle and Sort

- If **duplicate removal** is required, MapReduce automatically groups identical keys or values together during shuffle.

Step 3: Reduce Phase

- Reducer receives groups of identical projected tuples.
- It emits **unique records** (removes duplicates if necessary).

Advantages:

- Can process huge datasets to extract required attributes quickly.
- Duplicates can be removed easily using reducer.

4. Key Points

Selection with MapReduce

- Mapper filters rows based on a condition.
- Reducer is optional if no post-processing is required.

Projection with MapReduce

- Mapper emits only required columns.
- Reducer removes duplicates if needed.

Q. Function of Map Tasks in the MapReduce Framework

=>

1. Introduction

- **MapReduce** is a programming model designed to process large datasets across a cluster of machines.
- It splits the entire computation into two main phases:
  - **Map phase** (handled by Map tasks)
  - **Reduce phase** (handled by Reduce tasks)
- **Map tasks** are the first step and play a crucial role in transforming and preparing data for reduction.

2. Function of Map Tasks

What do Map Tasks do?

1. **Read Input Data:**
   - The input dataset is divided into fixed-size **splits (chunks)**.
   - Each split is processed by an individual **Map task** running on a node.
2. **Transform Data into Key-Value Pairs:**
   - The map function processes each record and **emits key-value pairs** as intermediate data.
3. **Filter, Parse, or Preprocess:**
   - Data can be **cleaned, filtered, or transformed** in this stage before being sent to reducers.
4. **Partition Data for Reducers:**
   - Keys determine which reducer will handle a group of values later.
   - This ensures all related data goes to the same reducer.

3. Example of Map Task Function

Problem: Word Count using MapReduce

- **Goal:** Count how many times each word appears in a text document.

Step 1: Input Splitting

Suppose we have the document:
"MapReduce makes data processing easy. Reduce makes it scalable."

- The input is split into chunks, and each chunk is sent to a separate Map task.

Step 2: Map Phase

- Each **Map task** reads a line (or chunk), splits it into words, and **emits key-value pairs**:

("MapReduce", 1)
("makes", 1)
("data", 1)
("processing", 1)
("easy.", 1)
("Reduce", 1)
("makes", 1)
("it", 1)
("scalable.", 1)

- These are **intermediate outputs**, not yet combined.

Step 3: Shuffle & Sort (automatic)

- The MapReduce framework groups all values by the same key (word).
- Example: all occurrences of "makes" are grouped together before going to the reducer.

4. Key Characteristics of Map Tasks

- **Run in parallel:** Each task handles a different input split on different nodes.
- **No knowledge of other tasks:** Map tasks are independent.
- **Output format:** Always emits data as (key, value) pairs.
- **Fault tolerance:** If a map task fails, it is automatically re-executed on another node.

Q. Why HDFS is More Suited for Large Datasets and Not Small Files ?

=>

1. Introduction

- **HDFS (Hadoop Distributed File System)** is a distributed storage system designed to store **very large files** reliably across many machines.
- It is **highly fault-tolerant**, **scalable**, and **optimized for throughput rather than low latency**.
- While excellent for storing **large datasets (GBs to TBs)**, it performs poorly with **millions of small files**.

2. How HDFS Works

1. **Block-based storage:**
    - Files in HDFS are split into **large fixed-size blocks** (default **128 MB** or **64 MB**).
    - Each block is stored across **multiple DataNodes** for fault tolerance (replication).
2. **Metadata management by NameNode:**
    - The **NameNode** stores metadata like file names, block locations, and permissions.
    - The **actual data** resides on DataNodes.

3. Why HDFS is Better for Large Files

Reason 1: Reduced Metadata Overhead

- For **large files**, only a **few blocks** are required.
- Example: A 1 GB file with 128 MB blocks → only **8 blocks** and **8 metadata entries**.
- The NameNode can handle this easily, ensuring **efficient memory usage**.

Reason 2: High Throughput for Sequential Access

- HDFS is designed for **streaming reads/writes** of big files rather than quick random access.
- Large files allow HDFS to **optimize sequential reads and writes** over the network.

Reason 3: Efficient Replication and Fault Tolerance

- Large files mean **fewer blocks to replicate**, reducing replication overhead.
- HDFS automatically copies each block to **3 different DataNodes**, which works better with fewer large blocks.

4. Problems with Small Files in HDFS

Problem 1: Metadata Overload on NameNode

- Every file, even if only **1 KB**, still requires a metadata entry.
- Millions of small files → millions of metadata entries stored in **NameNode memory (RAM)**.
- This **exhausts NameNode memory** and can cause performance degradation or failure.

Problem 2: Inefficient Block Usage

- HDFS block size is much larger than most small files.
- A 1 KB file still **occupies an entire 128 MB block slot**, wasting storage.
- This leads to **space inefficiency**.

Problem 3: Reduced Throughput

- Opening and closing many small files is expensive.
- HDFS is optimized for **streaming access**, not frequent random access or metadata lookups.

5. Practical Example

- **Case 1: Large File (1 GB)**

    - Stored as **8 blocks of 128 MB each**.
    - NameNode only needs **8 entries** to track.
- **Case 2: Many Small Files (1 KB × 1 million)**
    - Stored as **1 million separate blocks**.
    - NameNode must keep **1 million metadata entries** → high memory usage, slower lookups, and risk of crashes.

6. Solutions for Small Files in HDFS

- **HAR (Hadoop Archive):** Combines many small files into a single archive file.
- **Sequence Files / Avro / Parquet:** Stores small files as key-value pairs inside larger containers.
- **HBase:** For applications needing fast access to small records, HBase is better suited than HDFS.