Final Notes

MapReduce

MapReduce Flow

MapReduce processes data using the following steps:

1. **Input Splitting:** Splits the input file into smaller chunks
2. **Mapping:** Each Mapper processes its chunk and emits key-value pairs
3. **Shuffling & Sorting:** The framework sorts intermediate data and assigns it to Reducers
4. **Reducing:** Reducers aggregate results based on the keys
5. **Final Output:** The output is stored back in HDFS

Key Components

Mapper

Processes input data line by line and emits intermediate key-value pairs.

Example ⇒

```
public class WordCountMapper extends Mapper<LongWritable, Text, Text, IntWritable> {
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();
    public void map(LongWritable key, Text value, Context context)
        throws IOException, InterruptedException {
            StringTokenizer itr = new StringTokenizer(value.toString());
            while (itr.hasMoreTokens()) {
                word.set(itr.nextToken());
                context.write(word, one);
            }
        }
    }
```

Combiner

Acts as a mini reducer before data is sent to the final Reducer. It helps in reducing data transfer between Mapper and Reducer. All of the operations in a combiner need to be **Commutative and Associative** (function like `max()` or `sum()` are fine, but `avg()` is not as it might yield wrong results)

Example ⇒ Word Counting

> Without a combiner, each word count is sent individually. With a combiner, local aggregation occurs before sending the data to the reducer.

Partitioner

Determines which Reducer receives a given key. For example: Assigning logs by month to different reducers. The default partitioner is the `HashPartitioner` that distributes keys evenly. Can implement a custom one to distribute the data over multiple reducers.

Example ⇒

```
public int getPartition(K key, V value, int numReduceTasks) {
      return Math.abs(key.hashCode() % numReduceTasks)
  }
```

Shuffle and Sort

Intermediate key-value pairs are grouped by key before reaching the Reducer.

Reducer

Aggregates values associated with each key.

Example ⇒

```
public class WordCountReducer extends Reducer<Text, IntWritable, Text, IntWritable> {
  public void reduce(Text key, Iterable<IntWritable> values, Context context)
      throws IOException, InterruptedException {
          int sum = 0;
          for (IntWritable val : values) {
              sum += val.get();
          }
          context.write(key, new IntWritable(sum));
          }
      }
```

MapReduce Programs in Java

1. Write a Mapper Class: `map()` method processes input and emits key-value pairs
2. Write a Reducer Class: `reduce()` method aggregates results based on keys

[ADD EXAMPLE]

Common MapReduce Algorithms

MapReduce is widely used for processing large-scale data in a distributed manner. Below are some common algorithms implemented using MapReduce, along with detailed explanations and examples.

Sorting

Sorting is one of the most fundamental MapReduce operations. It is useful when dealing with large datasets where sorting helps in efficient querying and retrieval.

Example: Sorting Employee Salaries

Input:

```
Alice, 60000
Bob, 50000
Charlie, 70000
David, 60000
```

Map Phase:

Each key-value pair is transformed to make salary the key so that sorting can be performed.

```
(60000, Alice)
(50000, Bob)
(70000, Charlie)
(60000, David)
```

Shuffle & Sort Phase:

MapReduce automatically sorts the intermediate key-value pairs.

Reduce Phase:

The reducer outputs the sorted data in ascending order.

```
(50000, Bob)
(60000, Alice)
(60000, David)
(70000, Charlie)
```

Sorting can also be used in conjunction with secondary sorting when sorting within partitions is needed.

Searching

Searching allows us to scan large text datasets and find relevant information efficiently.

Example: Finding a Keyword in Log Files

Input: Log file contents

```
log1.txt: Error at line 45
log1.txt: Warning at line 100
log2.txt: Error at line 200
```

Search pattern: `"Error"`

Map Phase:

Each mapper scans a line and emits results if the search pattern is found.

```
(Error, log1.txt:45)
(Error, log2.txt:200)
```

Reduce Phase:

The reducer aggregates the results and outputs matching occurrences.

```
Error found in:
- log1.txt: Line 45
- log2.txt: Line 200
```

This approach is useful for searching error messages in logs, detecting spam in emails, or retrieving documents matching a keyword.

Building an Inverted Index

An inverted index is a key data structure used in search engines to quickly find documents containing a given word.

Example: Creating an Inverted Index for Documents

Input:

```
doc1.txt: Hadoop is a big data framework
doc2.txt: Hadoop uses MapRedu
```

Map Phase:

The mapper emits word-document pairs.

```
(Hadoop, doc1.txt)
(Hadoop, doc2.txt)
(big, doc1.txt)
(data, doc1.txt)
(framework, doc1.txt)
(uses, doc2.txt)
(MapReduce, doc2.txt)
```

Reduce Phase:

The reducer groups words and outputs a list of documents where each word appears.

```
Hadoop → [doc1.txt, doc2.txt]
big → [doc1.txt]
data → [doc1.txt]
framework → [doc1.txt]
uses → [doc2.txt]
MapReduce → [doc2.txt]
```

This structure helps search engines like Google retrieve relevant documents based on keyword searches.

Word Co-Occurrence

Word co-occurrence measures how often two words appear together in a dataset. This is useful for NLP applications, topic modeling, and recommendation systems.

Example: Measuring Co-Occurrence in a Sentence

Input:

```
doc1.txt: data science is fun
doc2.txt: big data is useful
```

Map Phase:

The mapper emits word pairs with a count of 1.

```
(data, science) → 1
(science, is) → 1
(is, fun) → 1
(big, data) → 1
(data, is) → 1
(is, useful) → 1
```

Reduce Phase:

The reducer sums up the occurrences.

```
(data, science) → 1
(science, is) → 1
(is, fun) → 1
(big, data) → 1
(data, is) → 1
(is, useful) → 1
```

This helps in building word association models, such as for predicting the next word in a sentence.

Secondary Sort

Secondary sorting allows sorting within a key's partition, often using a composite key.

Example: Sorting People by Last Name and Birth Year

Input:

```
John Smith, 1985
Alice Brown, 1990
Bob Smith, 1983
Charlie Brown, 1988
```

Map Phase:

The mapper outputs composite keys for sorting.

```
(Smith, 1985) → John
(Smith, 1983) → Bob
(Brown, 1990) → Alice
(Brown, 1988) → Charli
```

Partitioning and Sorting:

- **Partitioner** ensures all people with the same last name go to the same reducer.
- **Sort Comparator** sorts by birth year.

Reduce Phase:

The reducer receives sorted values within each partition.

```
Brown:
   - Charlie (1988)
   - Alice (1990)
Smith:
   - Bob (1983)
   - John (1985)
```

This approach is useful when sorting within a category, such as sorting transactions within a user's account.

These MapReduce algorithms form the backbone of big data processing in distributed systems, enabling tasks such as sorting, searching, indexing, and text analysis at scale.

Data Analysis with Spark

Spark Basics

Apache Spark is a distributed computing engine designed for fast and efficient big data processing.

Spark Architecture ⇒

- **Driver Program:** Runs the Spark application logic and coordinates execution
- **Cluster Manager:** Manages resource allocation (YARN)
- **Executors:** Performs distributed computations on worker nodes
- **Tasks:** Smallest unit of execution within an executor

Spark Execution Model ⇒

- When a job is submitted, the driver schedules tasks. Those tasks are then distributed across executors.
- **RDD transformations are computed lazily** (computed only when needed to save resources)
- Results are collected or stored in distributed storage

Common Spark APIs ⇒

- **RDD API:** Low-level, flexible, distributed collections

- **DataFrame API:** Optimized structured data processing (SQL tables)

RDD (Resilient Distributed Dataset)

Properties of RDDs

- **Immutability:** Cannot be modified after creation, but can be copied
- **Distributed:** Partitions stored across worker nodes
- **Fault-tolerant:** Can be recomputed from lineage
- **Lazy Evaluation:** Transformations are not executed until an action is triggered

```
# RDD example
rdd = spark.SparkContext.parallelize([1, 2, 3, 4, 5])

# RDD TextFile example
rdd = spark.SparkContext.textFile("data.txt")
```

RDD Tranformations

- `map()` → Mapper function (applies function to each element)
- `filter()` → Keeps elements that satisfy the condition
- `flatMap()` → Flattens nested structures
- `groupByKey()` vs `reduceByKey()` →
  - `reduceByKey()` is preferred because it performs local aggregation first
- `sortByKey()` → Sorts RDD based on keys

```
data = sc.parallelize([("Alice", 50),
                       ("Bob", 40),
                       ("Alice", 70),
                       ("Bob", 60),
                       ("Charlie", 30)])

# 1. map(): Increase each score by 10
mapped_rdd = data.map(lambda x: (x[0], x[1] + 10))

# 2. filter(): Keep only scores greater than 50
filtered_rdd = mapped_rdd.filter(lambda x: x[1] > 50)

# 3. flatMap(): Split names into characters (flattens results)
flat_mapped_rdd = filtered_rdd.flatMap(lambda x: list(x[0]))

# 4. groupByKey(): Group scores by user (less efficient)
grouped_rdd = filtered_rdd.groupByKey().mapValues(list)

# 5. reduceByKey(): Sum scores for each user (efficient)
reduced_rdd = filtered_rdd.reduceByKey(lambda a, b: a + b)
```

```
# 6. sortByKey(): Sort results alphabetically by user
sorted_rdd = reduced_rdd.sortByKey()
```

RDD Actions

- `collect()` → Returns all elements to the driver
- `count()` → Counts number of elements
- `take(n)` → Returns first `n` elements
- `reduce()` → Aggregates elements using a function or a lambda
- `foreach()` → Applies function or lambda to each element
- `count()` → Counts the elements in the RDD
- `parallelize()` → Converts a local collection into an RDD

```
rdd = spark.sparkContext.textFile("data.txt")
word_counts = rdd.flatMap(lambda line: line.split()) \
                         .map(lambda word: (word, 1))  \
                         .reduceByKey(lambda a,b: a + b)  \

# Print all
print(word_counts.collect())

# Print 5
print(word_counts.take(5))
```

Spark Application

Structure ⇒

- Initialize the SparkContext
- Define Transformations
- Execute Actions
- Write results to storage

`spark-submit --master yarn <my_scipt.py> <input[optional]> <output[optional]>`

RDD Persistence

Need for persistence

- Avoids recomputation of RDDs across multiple actions
- Improves performance when using iterative algorithms
- `rdd.persist()`

In essence, persisting an RDD is like caching the intermediate results to prevent recomputation and save on time and resources. Different than just caching as caching only stores data in RAM.

Iterative Algorithms in Spark

[ADD MORE EXAMPLES]

```
links = spark.sparkContext.parallelize([(1, [2, 3]), (2, [1, 3]), (3, [1])])
ranks = links.mapValues(lambda _: 1.0)

for _ in range(10):  # Iterative updates
    contribs = links.join(ranks).flatMap(
        lambda url_rank: [(dest, url_rank[1][1] / len(url_rank[1][0])) for dest in url_rank[1][0]]
    )
    ranks = contribs.reduceByKey(lambda a, b: a + b).mapValues(lambda rank: 0.85 * rank + 0.15)

print(ranks.collect())
```

Spark SQL and DataFrames

SparkSQL is a module for structured data processing in Apache Spark. It provides an SQL like interface on top of Spark. Supports both SQL queries and programmatic DataFrame API

Features ⇒

- **Unified API** → Works with RDDs, DataFrames, and Datasets
- **Optimized Query Execution** → Uses Catalyst optimizer for efficient processing
- **Supports Multiple Data Sources** → Works with Parquet, JSON, ORC, Hive,...
- **Interoperability with SQL Databases** → Can integrate with Hive, MySQL, PostgreSQL,...
- **In-memory processing** → Faster execution than traditional SQL engines

Spark SQL Data Frame Functions

- `show()` → displays the data frame
- `head()` → displays the head of the data frame
- `count()` → returns the number of rows
- `select("col1")` → selects a column
- `filter(df["col"] > value)` → filters based on some condition
- `where("col" > value)` → SQL style filtering
- `distinct()` → returns unique values
- `dropDuplicates()` → drops duplicate values in the table
- `sql()` → allows for SQL queries as a string
- `groupBy("col")` → groups by a column, used with aggregate functions
- `agg({"col" : "avg|sum|..."})` → compute aggregate functions (use with group by)

- `col("col")` → reference a column
- `when(condition, value).otherwise(value2)` → conditional logic
- `isnull("col)` → check for NULL values
- `join(df2, "col", "inner|left|right|outer|cross|...")` → joins
- `dropna()` → drop columns with `Na` or `NULL` values
- `replace({"old" : "new"})` → replace old value with new value
- `orderBy("col")` → sort the data

Data Frames

A DataFrame is a distributed collection of data organized into named columns (similar to SQL tables). It is built on top of RDDs but optimized for query execution. It also supports multiple operations like filtering, aggregation, grouping,...

Creating Data Frames

```
# From a CSV file
df = spark.read.csv("data.csv", header=True, inferSchema=True)
df.show(5)

# From an RDD
rdd = spark.sparkContext.parallelize([("alice", 25), ("bob", 30)])
df = spark.createDataFrame(rdd, ["name", "age"])
df.show()
```

DataFrame Operations

```
# Selecting columns
df.select("name", "age").show()

# Filtering Data
df.filter(df.age > 25).show()
df.where(df.age < 25).show()

# Adding new columns
df = df.withColumn("new_age", df.age + 5).show()

# Dropping columns
df = df.drop("new_age").show()

# Grouping and Aggregation
df.groupBy("gender").agg({"salary": "avg"}).show()
df.groupBy("gender").agg(avg("salary")).show()

# Sorting data
df.orderBy("age").show()
```

```
df.orderBy(df.age.desc()).show()

# Using SQL queries
spark.sql("""SELECT name, age FROM people WHERE age > 25""")

# Joins
df1.join(df2, df1.id == df2.id, "inner|left|right|outer").show()

# Writing to a file
df.write.csv("<filename>", header=True)
```

Data Analysis with Hive

Hive vs Relational DBs

Hive and traditional relational DBs (RDBMS) serve different purposes. Hive is optimized for batch processing of big data, while RDBMS is designed for transactional processing.

Key Differences

Feature	Hive	Relational Databases (RDBMS)
Schema	Schema-on-read	Schema-on-write
Data Storage	HDFS (distributed storage)	Local disk or SAN storage
Query Language	HiveQL (SQL-like)	SQL
Execution	Runs on Hadoop (MapReduce, Tez, or Spark)	Uses traditional query engines
Performance	Optimized for batch queries	Optimized for transactions
Indexing	Limited	Strong indexing support
Concurrency	Low	High

Use Hive when you need to process huge datasets. Queries are read-heavy and involve aggregations. Data is semi-structured (logs, clickstream data,...)

Use RDBMS when you need real-time, transactional processing. Queries involve frequent updates, deletes or inserts. You need **ACID** compliance (Atomicity, Consistency, Isolation, Durability)

Data Manipulation (Hive QL)

Select

```
-- Select
SELECT column1, column2 FROM table_name;

-- Select ALL
SELECT * FROM table_name;

-- Select Unique
SELECT DISTINCT column1 FROM table_name;
```

Limit

```
-- Limit the number of rows returned by a query
SELECT * FROM table_name LIMIT 10;
```

Combine Query Results

```
-- UNION removes dups
SELECT column1 FROM table_name UNION SELECT column1 FROM table_name2;

-- UNION ALL keeps them
SELECT column1 FROM table_name UNION ALL SELECT column1 FROM table_name2;
```

Subqueries

```
-- Queries inside queries
-- ADD EXAMPLE WITH NAME
SELECT name, age FROM eployees WHERE age > (SELECT AVG(age) FROM employees)
SELECT name FROM customers WHERE customer_id IN (SELECT customer_id FROM orders);
```

Complex Data Types (array, map, struct)

```
---------------------------------------------------------------------
-- Array Example
CREATE TABLE employees (name STRING, skills ARRAY<STRING>);
INSERT INTO employees VALUES ('Alice', ARRAY('Java', 'Python', 'SQL'));

-- Query
SELECT name, skills[0] FROM employees;

---------------------------------------------------------------------
-- Map Example
CREATE TABLE student_grades (name STRING, grades MAP<STRING, INT>);
INSERT INTO student_grades VALUES ('Bob', MAP('Math', 90, 'Science', 85));

-- Query
SELECT name, grades['Math'] FROM student_grades;  -- Returns 90
---------------------------------------------------------------------
-- Struct Example
CREATE TABLE employees (name STRING, address STRUCT<city:STRING, zip:INT>);
INSERT INTO employees VALUES ('Alice', STRUCT('Seattle', 98101));

SELECT name, address.city FROM employees;
---------------------------------------------------------------------
```

Join

Joins allow combining data from multiple tables.

```
-- Inner Join (default)
SELECT e.name, d.department FROM employees e JOIN departments d ON e.dept_id = d.dept_id

-- Left Outer Join
SELECT e.name, d.department FROM employees e LEFT JOIN departments d ON e.dept_id = d.de

-- Right Outer Join
SELECT e.name, d.department FROM employees e RIGHT JOIN departments d ON e.dept_id = d.d

-- Full Outer Join
SELECT e.name, d.department FROM employees e FULL JOIN departments d ON e.dept_id = d.de
```

Grouping

`GROUP BY` groups the records and applies aggregate functions like `COUNT`, `AVG`, `SUM`

```
SELECT department, COUNT(*) FROM employees GROUP BY department;
```

Hive Optimization

Hive Query Performance Patterns

- Use partitioning to reduce data scans
- Prefer columnar storage over row storage (Parquet vs. CSV)
- Use map-side joins for smaller tables
- Avoid `SELECT *`, only fetch required columns

Execution Plan

- To see how Hive executes a query, use the `EXPLAIN` keyword. Best practice is to always check execution plans before running complex queries.

Order By vs. Sort By

Clause	Behavior
ORDER BY	Fully sorts all data using a **single reducer** (slow).
SORT BY	Partially sorts data using **multiple reducers** (faster).

```
SELECT * FROM employees ORDER BY salary DESC;  -- Global sorting
SELECT * FROM employees SORT BY salary DESC;   -- Partial sorting
```

Partitioning

Divides a table into multiple directories. Best practice is to always filter by partition columns to optimize performance.

```
CREATE TABLE sales (id INT, amount FLOAT) PARTITIONED BY (year INT);

-- Query
SELECT * FROM sales WHERE year = 2023;
```

Bucketing

Bucketing distributes data into fixed “buckets” based on a column’s hash. Best practice is to use bucketing for frequent joins on the bucketed column.

```
CREATE TABLE user (id INT, name STRING) CLUSTERED BY (id) INTO 4 BUCKETS;
```

Reduce-side join

Default join. Data is shuffled across reducers. Suitable for large tables but expensive due to network transfer.

```
SELECT * FROM orders o JOIN customers c ON o.customer_id = c.customer_id;
```

Map-side join

Loads smaller table into memory for faster joins.

```
SELECT /*+MAPJOIN(b)*/ * FROM orders a JOIN customers b ON a.customer_id = b.customer_id
```