Index

1. MapReduce Overview
2. Map Phase
3. Reduce Phase
4. Key Components of MapReduce
   * 4.1 Mappers
   * 4.2 Reducers
   * 4.3 Partition
   * 4.4 Hash Function
   * 4.5 Shuffle and Sort
   * 4.6 Local Aggregation
   * 4.7 Combiner
   * 4.8 Custom Partitioning
5. Challenges with MapReduce
6. Apache Spark
7. Differences between MapReduce and Apache Spark
8. Apache Spark vs Databricks
9. Spark Core API
10. Differences between RDD, DataFrame, and Spark SQL
11. Spark Execution Plan
12. Why are transformations lazy in Spark?
13. Spark Session vs Spark Context
14. Python Basics
15. Python Functions
16. Lambda Functions
17. Parallelize & Parallelism in Spark
18. Default Spark Functions & Transformations
19. Wide vs Narrow Transformations
20. Stages, Jobs, and Tasks in Spark
21. ReduceByKey vs GroupByKey
22. Spark Joins
23. Repartition and Coalesce
24. Cache and Persist in Spark

1. MapReduce Overview

MapReduce is a programming model used for processing large datasets in a distributed manner. It allows applications to process vast amounts of data by breaking it down into smaller, manageable chunks that can be processed in parallel across a distributed system. It works by performing two main operations: *Map* and *Reduce*. The model is primarily used in Hadoop for batch processing.

2. Map Phase

In the Map phase, the input data is divided into smaller chunks, and a mapper function is applied to each chunk. Each chunk typically consists of key-value pairs. The mapper processes each input record and outputs intermediate key-value pairs, which will be further processed in the next phase.

3. Reduce Phase

The Reduce phase aggregates the intermediate key-value pairs generated by the mappers. It groups the intermediate values by their keys and applies a reduce function to each group. The final output consists of the results after the aggregation, often representing some summarized or transformed data.

4. Key Components of MapReduce

4.1 Mappers

Mappers are responsible for processing input data and generating intermediate key-value pairs. The mapper function takes a key-value pair as input and outputs a set of intermediate key-value pairs.

4.2 Reducers

Reducers take the intermediate key-value pairs generated by mappers, group them by keys, and apply a reduce function to each group to produce the final output.

4.3 Partition

Partitioning refers to dividing the intermediate key-value pairs into different partitions, where each partition is assigned to a reducer. The partitioning strategy determines how the data is distributed among the reducers.

4.4 Hash Function

A hash function is used in partitioning to determine which partition a key-value pair should go to. It maps the key to a partition ID based on the hash value.

4.5 Shuffle and Sort

The Shuffle and Sort phase occurs between the Map and Reduce phases. During this phase, the intermediate data is shuffled, and all values associated with the same key are grouped together. Sorting helps in optimizing the Reduce phase by ensuring that data is processed in a sorted order.

4.6 Local Aggregation

Local aggregation happens within the mappers before the Shuffle and Sort phase. It involves reducing intermediate key-value pairs locally within each mapper to reduce the amount of data transferred to the reducers.

4.7 Combiner

A combiner is an optional operation that can be applied on the output of the map phase. It acts as a mini-reducer, which helps in reducing the amount of data that needs to be transferred to the reducers by performing local aggregation.

4.8 Custom Partitioning

Custom partitioning refers to defining your own partitioning logic instead of using a default hash-based partitioning mechanism. It can help optimize performance by controlling the distribution of data across reducers.

5. Challenges with MapReduce

* Scalability Issues: MapReduce can have difficulty scaling efficiently for complex operations involving large amounts of data or multiple stages.
* Latency: The model may have higher latency, as the data must pass through several stages, including mapping, shuffling, and reducing.
* Complexity: Writing MapReduce programs can be more complex compared to higher-level abstractions like SQL or DataFrames.
* Fault Tolerance: While MapReduce has fault tolerance, failures in the system can lead to long recovery times due to the need to restart tasks from scratch.

6. Apache Spark

Apache Spark is an open-source, distributed computing system designed for speed and ease of use. It provides in-memory processing, making it significantly faster than MapReduce. Spark supports batch and real-time processing, and can handle workloads such as ETL, machine learning, graph processing, and SQL queries.

7. Differences between MapReduce and Apache Spark

* Performance: Spark provides in-memory processing, which results in much faster performance than MapReduce, which relies on disk-based storage for intermediate data.
* Ease of Use: Spark provides higher-level APIs (DataFrames, SQL, MLlib) that are easier to use compared to the low-level MapReduce programming model.
* Data Processing Models: MapReduce works only for batch processing, while Spark supports both batch and stream processing.
* Fault Tolerance: Both MapReduce and Spark provide fault tolerance, but Spark’s RDDs provide more efficient recovery mechanisms.

8. Apache Spark vs Databricks

Databricks is a cloud-based platform built on top of Apache Spark. It provides an interactive workspace for collaboration, optimized Spark clusters, and integrated tools for data engineering, machine learning, and analytics. While Spark is the underlying engine, Databricks simplifies its usage with additional features, such as notebooks, job scheduling, and advanced visualization.

9. Spark Core API

The Spark Core API provides essential functionality for distributed data processing, including the management of RDDs, job execution, and task scheduling. It is the foundation for other Spark components such as Spark SQL, MLlib, and GraphX.

10. Differences between RDD, DataFrame, and Spark SQL

* RDD (Resilient Distributed Dataset): The fundamental data structure in Spark. RDDs are immutable, distributed collections of objects. They provide low-level operations for data manipulation.
* DataFrame: A higher-level abstraction built on top of RDDs. It provides a more optimized way to work with data using a schema (like a table) and supports SQL operations.
* Spark SQL: A module for working with structured data using SQL queries. DataFrames are the core abstraction for Spark SQL.

11. Spark Execution Plan

Spark executes a query by first generating a logical plan, then optimizing it into a physical plan, and finally executing it on a cluster. The logical plan describes what operations need to be performed, while the physical plan specifies how those operations should be executed efficiently.

12. Why are transformations lazy in Spark?

Transformations in Spark are lazy to optimize performance. Lazy evaluation means that Spark does not immediately execute the transformation but rather builds a logical plan. It only executes when an action is triggered, allowing Spark to optimize the execution by combining multiple transformations into fewer stages.

13. Spark Session vs Spark Context

* SparkContext: The entry point for Spark functionality. It provides access to the underlying Spark cluster and allows creating RDDs, accumulators, and broadcast variables.
* SparkSession: A newer entry point that combines functionality from both SQLContext and SparkContext. It is the entry point for working with structured data (DataFrames and SQL).

14. Python Basics

Python is a high-level, interpreted programming language known for its simplicity and readability. It supports multiple programming paradigms, including procedural, object-oriented, and functional programming.

15. Python Functions

Functions in Python are defined using the def keyword. They allow the grouping of code into reusable blocks. Functions can take parameters and return values.

16. Lambda Functions

Lambda functions are small anonymous functions defined using the lambda keyword. They are often used for short, one-off operations like filtering or transforming data.

17. Parallelize & Parallelism in Spark

Parallelism in Spark refers to dividing tasks into smaller sub-tasks that can run simultaneously across multiple nodes. The parallelize() function allows converting a Python collection into an RDD that can be distributed across a cluster for parallel processing.

18. Default Spark Functions & Transformations

* Transformations: map(), flatMap(), filter(), distinct(), groupBy(), reduceByKey(), etc.
* Actions: count(), collect(), save(), reduce(), first(), etc.

19. Wide vs Narrow Transformations

* Narrow Transformations: Involve a single partition (e.g., map(), filter()). These transformations are more efficient as they do not require shuffling data.
* Wide Transformations: Involve multiple partitions (e.g., groupBy(), reduceByKey()). These require data shuffling between partitions and are generally more expensive.

20. Stages, Jobs, and Tasks in Spark

* Jobs: A job is a high-level operation triggered by an action, such as collect() or save().
* Stages: A stage is a set of tasks that can be executed in parallel. Stages are separated by wide transformations.
* Tasks: Tasks are units of work that are distributed across the cluster. Each task processes data from a single partition.

21. ReduceByKey vs GroupByKey

* reduceByKey: Combines values for each key using an associative and commutative function before shuffling data. It is more efficient than groupByKey because it reduces the amount of data transferred.
* groupByKey: Groups all values for each key together before performing any operations. This can lead to excessive shuffling and performance issues for large datasets.

22. Spark Joins

Spark supports several types of joins, including:

* Inner Join: Combines rows with matching keys.
* Left Join: Includes all rows from the left DataFrame, even if no match is found in the right DataFrame.
* Right Join: Includes all rows from the right DataFrame, even if no match is found in the left DataFrame.
* Outer Join: Includes all rows from both DataFrames, filling in missing values with null.

23. Repartition and Coalesce

* Repartition: Increases or decreases the number of partitions by reshuffling the data.
* Coalesce: Merges adjacent partitions into fewer partitions, which is more efficient than repartition().

24. Cache and Persist in Spark

* Cache: A shorthand for persist() with the default storage level (MEMORY\_AND\_DISK).
* Persist: Allows the user to specify the storage level for an RDD, such as MEMORY\_ONLY, MEMORY\_AND\_DISK, etc.