Spark

1. Describe the architecture of Spark

Apache Spark is an open-source, distributed data processing and analytics framework designed for large-scale data processing tasks. Spark architecture consists of four components, including the spark driver, executors, cluster administrators, and worker nodes. It uses the Dataset and data frames as the fundamental data storage mechanism to optimise the Spark process and big data computation

**Driver Program:** The driver program is the entry point of a Spark application. It contains the main method and coordinates the execution of tasks across the cluster. The driver program defines the high-level logic of the application, splits it into tasks, and schedules these tasks for execution.

**Executor** is responsible for executing a job and storing data in a cache at the outset. Executors first register with the driver programme at the beginning. These executors have a number of time slots to run the application concurrently.

**Cluster Manager:** Spark can run on various cluster managers, such as Apache Mesos, Hadoop YARN, or Kubernetes. The cluster manager allocates resources (CPU, memory) and manages the execution of Spark applications on a cluster of machines.

**Worker Nodes:** Worker nodes are the machines in the cluster where the actual data processing takes place. Each worker node hosts one or more executors, which are responsible for executing tasks. Executors manage the storage and computation resources allocated to them.

1. What is a cluster manager? Which ones have you used?

A cluster manager is a system that manages the allocation of resources and coordinates the execution of tasks across a cluster of machines. It's a crucial component in distributed computing frameworks like Apache Spark, Apache Hadoop, and other distributed data processing systems. Cluster managers handle tasks such as resource allocation, task scheduling, fault tolerance, and load balancing

I used Apache Hadoop YARN (Yet Another Resource Negotiator) is a component of the Hadoop ecosystem that manages resources and schedules tasks. Originally designed for Hadoop's MapReduce, YARN has evolved to support various processing frameworks, including Spark. YARN allocates resources to applications based on their resource requirements and priorities.

1. Difference between SparkContext and SparkSession

**SparkContext**:

* SparkContext (often referred to as sc) was the main entry point in earlier versions of Apache Spark and is still present in the API for backward compatibility.
* It was used to interact with the Spark cluster and coordinate the execution of Spark applications.
* SparkContext was responsible for creating and managing RDDs (Resilient Distributed Datasets), which were the primary data abstraction in Spark's early versions.
* It was used for low-level operations like creating RDDs, broadcasting variables, and accessing cluster-level configuration settings.
* With the introduction of DataFrames and Datasets, the need for SparkContext has diminished, as these higher-level abstractions offer more optimized query optimization and execution.

**SparkSession**:

* SparkSession is the entry point for Spark's structured APIs, introduced to unify and simplify Spark application development.
* It encapsulates the functionality of both the older SparkContext, SQLContext, and HiveContext, providing a unified interface for working with various data sources and APIs.
* SparkSession is used to create DataFrames and Datasets, which are higher-level abstractions built on top of RDDs and provide optimizations for structured and semi-structured data.
* It handles the underlying configuration and session-specific settings, making it easier to manage resources and settings for different parts of your Spark application.
* SparkSession allows you to seamlessly switch between SQL, DataFrames, and Datasets APIs within the same application.
* The SparkSession API provides methods for reading and writing data from various sources, performing SQL queries, and interacting with other Spark components like Spark Streaming, MLlib, and GraphX.

1. Describe spark modes to execute the program.

Apache Spark supports multiple execution modes to run programs, allowing users to choose the appropriate mode based on their use case, infrastructure, and requirements. The choice of execution mode depends on factors like cluster size, resource availability, ease of management, integration with other tools, and the specific requirements of your Spark application.

Local mode is suitable for development and testing, while standalone mode, Mesos, YARN, Kubernetes, and cloud platforms like Amazon EMR are used for running Spark applications in production on larger clusters.

1. Difference between RDD and DF

|  |  |
| --- | --- |
| **RDD** | **DF** |
| low-level abstraction that represents a distributed collection of elements | higher-level abstraction that represents a distributed collection of data organized into named columns with a schema |
| Not optimized by Spark's Catalyst query optimizer | optimized by Spark's Catalyst query optimizer |
| RDDs are not type-safe, meaning errors in data types are often only detected at runtime. | DataFrames are schema-based and offer strong type checking at compile time |
| Lower performance | Higher performance |
| RDD is useful for non-tabular data structures | preferred choice for most structured and semi-structured data processing tasks |

1. Transformation vs Action

Transformations and Actions are two fundamental types of operations that you can perform on distributed data structures like RDDs (Resilient Distributed Datasets) or DataFrames.

**Transformations** are used to define the sequence of operations to be performed on the data. They are lazily evaluated and create a lineage of transformations.

Examples of transformations include map, filter, join, groupBy, flatMap, and union.

**Actions** trigger the actual execution of the transformations and return results or perform side effects. They initiate the computation and materialize the data.

Examples of actions include count, collect, saveAsTextFile, reduce, and foreach.

1. Narrow transformation vs Wide transformation

**Narrow Transformations:**

* Narrow transformations are transformations where each input partition contributes to only one output partition, without the need for data shuffling or shuffling of partitions.
* These transformations maintain a one-to-one relationship between input and output partitions, making them more efficient in terms of data movement across the cluster.
* Narrow transformations are performed independently on each partition, and the resulting partitions are usually computed in parallel without the need for data exchange between partitions.
* Examples of narrow transformations include map, filter, union, and localMap.

**Wide Transformations:**

* Wide transformations are transformations where each input partition contributes to multiple output partitions, potentially requiring data shuffling or reorganization.
* These transformations involve redistributing data across partitions, which can lead to more complex execution plans and additional overhead.
* Wide transformations typically involve operations that require grouping, aggregating, or joining data across partitions.
* As wide transformations require data shuffling, they tend to be more expensive in terms of computational and network resources compared to narrow transformations.
* Examples of wide transformations include groupBy, reduceByKey, join, cogroup, and sortByKey.

1. What is lazy evaluation

Lazy evaluation is a programming concept employed by many modern data processing frameworks, including Apache Spark. It refers to the strategy of postponing the evaluation of an expression or computation until the result is actually needed. In the context of Spark, lazy evaluation plays a fundamental role in optimizing the execution of data transformations.

When you define a sequence of transformations on a dataset (such as RDD or DataFrame), these transformations are not immediately executed. Instead, Spark builds a directed acyclic graph (DAG) that represents the logical execution plan. This DAG captures the sequence of transformations and actions, along with their dependencies, but it does not perform any actual computations until an action is triggered.

1. What is DAG?

DAG stands for Directed Acyclic Graph. DAG is used to represent the sequence of transformations and dependencies between operations in a computation plan. It's a visual representation that helps illustrate how data flows through various stages of processing.

In a Spark DAG:

* Nodes represent transformations or actions that need to be executed.
* Edges represent dependencies between the transformations. If transformation B depends on the output of transformation A, there will be a directed edge from A to B.

1. What is lineage?

It refers to the history or the sequence of transformations that have been applied to a dataset to produce its current state. It is a fundamental concept that enables fault tolerance and data recovery in case of node failures or data loss.

Here's a simplified example of lineage:

Initial Data: [1, 2, 3, 4, 5]

Transformation A (map): doubled = [2, 4, 6, 8, 10]

Transformation B (filter): filtered = [4, 6, 8]

In this example, if the data or any intermediate RDD is lost, Spark can recreate the filtered RDD by using lineage to trace back the transformations applied to the original data.

1. Difference between DAG and Lineage?

DAG represents the sequence and dependencies of computations or operations to be executed, while lineage focuses on maintaining the history of transformations applied to data for the purpose of fault tolerance, data recovery, and lazy evaluation.

1. What happens when you submit a spark job

* **Compilation and Packaging:**

You develop your Spark application code using the Spark API (e.g., RDD, DataFrame, SparkSession).

You package your application code and any required dependencies (JARs, libraries) into an executable JAR or other appropriate format.

* **Submission to Cluster Manager**:

You use a command-line tool (spark-submit for standalone mode, or equivalent tools for other cluster managers) to submit your packaged application to the cluster.

The submission command specifies the main class of your application and other configuration parameters.

* **Driver Initialization:**

The cluster manager launches the driver program (the main class of your application) on one of the cluster's nodes.

The driver is responsible for coordinating the execution of tasks and managing the overall application flow.

* **Driver Setup:**

The driver initializes the SparkContext or SparkSession, which serves as the entry point for interacting with the cluster and managing resources.

* **Job and Stages Creation:**

When you perform transformations on your data (e.g., map, filter, groupByKey), Spark creates a directed acyclic graph (DAG) that represents the computation plan, divided into stages.

Stages are sets of transformations that can be executed in parallel with each other, often separated by wide transformations that require shuffling.

* **Stage Scheduling:**

The driver submits stages to the cluster manager for execution.

The cluster manager schedules tasks for each stage across the worker nodes, taking into account available resources and data locality.

* **Task Execution:**

The worker nodes execute the tasks in parallel based on the stages assigned to them.

Tasks process the data based on the transformations defined in your code.

* **Data Shuffling:**

If wide transformations like groupBy or join are encountered, data shuffling may occur. This involves redistributing and exchanging data between partitions on different nodes.

* **Task Completion and Result Aggregation:**

As tasks complete, they return their results to the driver or intermediate stages, which are then used as inputs for subsequent tasks.

* **Action Execution:**

When you invoke an action like count, collect, or saveAsTextFile, the actual computation is triggered.

Actions trigger the execution of the DAG transformations.

* **Job Completion and Cleanup:**

Once all tasks and stages are successfully completed, the driver program finalizes the job execution.

The driver may write results to external storage or perform other actions as needed.

Resources are released, and temporary data structures are cleaned up.

* **Application Termination:**

The driver program terminates, and the cluster manager releases the resources used by the application.

1. Client mode vs cluster mode

**Client mode** is more suitable for interactive development and debugging scenarios, allowing you to have direct control and visibility over the driver process.

**Cluster mode** is recommended for production deployments, as it ensures better resource utilization, isolation, and fault tolerance by running the driver within the cluster.

1. Difference between a DF and a DS

DataFrames and Datasets are powerful abstractions for working with structured data in Spark.

DataFrames are optimized for query performance and offer a more SQL-like interface, while Datasets provide strong typing and improved code safety.

1. Difference between a Pandas DF and a Spark DF

Pandas DF is suitable for single-node data manipulation on smaller datasets, whereas Spark DF is designed for distributed processing of large-scale datasets across a cluster

1. Coalesce vs repartition

Use **coalesce** when you want to reduce the number of partitions without shuffling and the number of partitions is already larger than necessary.

Use **repartition** when you want to either increase or decrease the number of partitions, and you want to redistribute data evenly for better data distribution and parallelism.

1. What’s a shuffle?

Shuffle is a process of redistributing and reorganizing data across partitions or nodes in a cluster.

Shuffling typically involves three main steps:

**Map**:

During the map phase, data is processed and transformed according to the specified operation. This can involve filtering, mapping, or applying any transformation that doesn't require data from other partitions.

**Shuffle**:

In the shuffle phase, data is redistributed across partitions or nodes to ensure that data with the same keys (or join keys) end up on the same partition.

Shuffling involves network communication and data movement, as data from one partition needs to be exchanged with other partitions in order to group or join data correctly.

**Reduce**:

During the reduce phase, the processed and shuffled data is further processed to achieve the desired outcome, such as calculating aggregates, performing joins, or applying other operations that require combining data.

1. What is a logical plan vs a physical plan?

**Logical Plan**: Represents the high-level structure and sequence of operations in a query without considering the physical execution details. It captures the query's intent.

**Physical Plan**: Represents the actual execution strategy by mapping the logical operations to specific physical operations that take into account data distribution, parallelism, and resource utilization.

1. What is a driver?

Driver refers to a specific component or program that serves as the entry point for executing a user's application or job. The driver program manages the overall execution, coordinates tasks, and interacts with the cluster manager to run the job on a distributed cluster of machines.

1. What is an executor?

Spark Executor is a process that runs on a worker node in a Spark cluster and is responsible for executing tasks assigned to it by the Spark driver program.

1. When would you use a broadcast join?

Broadcast join in spark is preferred when we want to join one small data frame with the large one. The requirement here is we should be able to store the small data frame easily in the memory so that we can join them with the large data frame in order to boost the performance of the join.

When we call broadcast on the smaller DF, Spark sends the data to all the executor nodes in the cluster. Once the DF is broadcasted, Spark can perform a join without shuffling any of the data in the large DataFrame

22. What is a broadcast variable?

Broadcast variables are used in scenarios where a piece of data is too large to fit in memory on each worker node, but it's still required for computations on those nodes. By broadcasting the variable, Spark ensures that each worker node has a local copy of the data available for use.

How broadcast variables work:

**Creating a Broadcast Variable:**

In Spark, you create a broadcast variable by calling the .broadcast() method on the data you want to broadcast. This method wraps the data and prepares it for distribution.

**Distribution:**

When a broadcast variable is created, Spark serializes the data and sends it to each worker node in the cluster.

**Local Copies:**

Each worker node stores the broadcasted data in memory, making it available for local operations. This avoids the need to send the data over the network repeatedly.

**Efficient Access:**

When a task on a worker node requires the data, it can access the local copy of the broadcast variable without incurring the overhead of network communication.

1. What is accumulator?

Accumulator is a special variable in Apache Spark that is used for aggregating values across multiple tasks or worker nodes in a parallel and distributed computation. Accumulators provide a way to efficiently perform counters or sums in parallel without the need for explicit synchronization between tasks.

1. Spark Streaming vs Structured Streaming

**Spark Streaming** when you have a use case that aligns well with micro-batch processing, and you're comfortable with RDD-based transformations and actions.

**Structured Streaming** when you want a more unified, SQL-like programming model and lower latency, while enjoying built-in fault tolerance with exactly-once semantics. Structured Streaming is generally recommended for new projects, as it provides a more modern and versatile approach to real-time processing.

1. What is Dynamic Partition Pruning?

Dynamic Partition Pruning (DPP) is a query optimization technique used in data processing systems to improve the efficiency of partitioned data retrieval. The goal of dynamic partition pruning is to optimize queries by skipping unnecessary partitions during query execution, thereby reducing the amount of data read and improving query performance.

1. Cache v/s persist

**Cache** when you want a simple way to store data in memory and improve performance. It's convenient when you don't need to worry about different storage levels or serialization.

**Persist** when you want more control over how data is stored. You can choose specific storage levels and serialization options based on your needs.

The choice between them depends on your need for control over storage levels, serialization, and resiliency against memory eviction.

1. Advantages n disadvantages of big data File formats

1. Parquet:

Advantages:

Columnar storage: Parquet stores data in a columnar format, which is highly efficient for analytics queries, as it allows for better compression and skip-reads.

Compression: Parquet supports various compression algorithms, which can significantly reduce storage requirements and improve I/O performance.

Schema evolution: Parquet files can store metadata about the schema, allowing for schema evolution without data conversion.

Compatibility: Parquet is widely supported in various big data tools, including Apache Spark, Hive, Impala, and Presto.

Disadvantages:

Write overhead: Writing Parquet files involves additional metadata and compression, which might slightly increase write overhead compared to some other formats.

Complexity: Parquet's columnar storage format can be more complex to work with in some cases, especially when doing data updates.

2. ORC (Optimized Row Columnar):

Advantages:

Columnar storage: Similar to Parquet, ORC's columnar storage provides efficient storage and query performance.

Compression: ORC uses lightweight compression algorithms, which can lead to better compression ratios and faster decompression.

Predicate pushdown: ORC supports predicate pushdown, where data filtering is performed before loading into memory, improving query performance.

Disadvantages:

Hive-centric: ORC was initially developed for Hive, so compatibility might be slightly limited compared to more universal formats.

Limited support: While ORC is supported by various tools, it might not be as universally supported as Parquet.

3. Avro:

Advantages:

Schema evolution: Avro supports schema evolution and provides a compact binary format, making it suitable for cases where schemas might change over time.

Self-describing: Avro files contain schema information, making them self-describing and enabling better data validation.

Compatibility: Avro is supported by various big data tools and programming languages.

Disadvantages:

Slower read performance: Avro's row-based format can lead to slower query performance compared to columnar formats like Parquet or ORC, especially for analytics workloads.

Compression: While Avro supports compression, it might not achieve the same compression ratios as some other formats.

4. JSON:

Advantages:

Human-readable: JSON is easy to read and understand, making it suitable for small-scale or semi-structured data.

Wide adoption: JSON is a widely used data interchange format and is supported by many programming languages and tools.

Disadvantages:

High storage overhead: JSON's textual nature results in higher storage overhead and slower I/O compared to binary formats.

Schema-less: JSON is schema-less, which can lead to challenges in data validation and consistency when dealing with large datasets.

5. CSV (Comma-Separated Values):

Advantages:

Simplicity: CSV files are simple, human-readable, and can be easily generated and processed with basic tools.

Universal support: CSV is supported by virtually all programming languages and tools.

Disadvantages:

No schema: Like JSON, CSV lacks a defined schema, which can lead to data consistency issues and the need for manual schema inference.

Limited features: CSV files lack advanced features like compression, data type preservation, and schema evolution, which some other formats offer.

1. what are compression formats and its specialities

1. gzip:

Specialty: Gzip is a widely used and well-known compression format. It's based on the DEFLATE algorithm, which provides good compression ratios and relatively fast decompression.

Advantages: Gzip is lightweight and suitable for compressing single files. It's commonly used for web content compression and on Unix-like systems.

Disadvantages: Gzip doesn't handle compression of multiple files or complex directory structures natively. It also doesn't provide the best compression ratios compared to more modern algorithms.

2. bzip2:

Specialty: Bzip2 uses the Burrows-Wheeler transform and Huffman coding to achieve high compression ratios at the cost of slower compression and decompression speeds.

Advantages: Bzip2 achieves better compression than gzip and is useful when space savings are a priority. It's often used for archiving and distribution.

Disadvantages: Bzip2's slower compression and decompression can be a drawback for performance-sensitive applications. It's not as commonly used as some other formats.

3. LZ4:

Specialty: LZ4 is a fast compression format that provides high compression and decompression speeds. It's designed to be used when low latency is crucial.

Advantages: LZ4 offers excellent compression and decompression performance, making it suitable for real-time applications and high-speed data transfers.

Disadvantages: LZ4 sacrifices some compression ratios for speed. It might not achieve the same compression levels as some other formats.

4. Snappy:

Specialty: Snappy is designed for high-speed compression and decompression. It's often used when performance is a priority.

Advantages: Snappy is extremely fast and provides moderate compression ratios. It's suitable for applications where processing speed is crucial, such as real-time analytics.

Disadvantages: Snappy doesn't achieve as high compression ratios as some other formats, so it might not be the best choice for scenarios where space savings are a top priority.

5. Zstandard (zstd):

Specialty: Zstandard is a modern compression format that offers a balance between compression ratios and speed. It's designed to be highly customizable.

Advantages: Zstandard achieves competitive compression ratios while maintaining fast compression and decompression speeds. It's suitable for a wide range of applications.

Disadvantages: While Zstandard is versatile, it might not be the absolute best in terms of compression ratios or speed for specific use cases.

1. Spark optimization techniques. Share use case

1. Data Partitioning and Skew Handling:

Optimization: Partition your data properly to avoid data skew, where a few partitions contain much more data than others. Use techniques like salting keys, bucketing, or custom partitioning to evenly distribute data.

Use Case: In a clickstream analysis, user data might be skewed towards certain users. By partitioning the data based on user IDs and handling skew, you can ensure balanced processing and prevent bottlenecks.

2. Broadcast Joins:

Optimization: Use broadcast joins for small tables in a join operation to avoid shuffling and network traffic.

Use Case: Joining a large fact table with a small lookup table, such as joining sales data with product information. Broadcasting the product information table can improve join performance.

3. Data Caching and Persistence:

Optimization: Cache or persist intermediate data that will be reused in multiple operations. This avoids recomputation and improves performance.

Use Case: In iterative machine learning algorithms like gradient descent, caching the feature vectors can significantly speed up the training process.

4. Column Pruning and Projection:

Optimization: Only select and process the columns needed for a specific operation. Avoid processing unnecessary columns.

Use Case: When running SQL queries on a large dataset, projecting only the required columns can reduce memory consumption and improve query speed.

5. Coalesce and Repartition:

Optimization: Use coalesce to reduce the number of partitions and repartition to increase parallelism when needed.

Use Case: After filtering and transformations, you might end up with many small partitions. Coalescing or repartitioning can help balance the workload and improve processing efficiency.

6. Use of Broadcast Variables and Accumulators:

Optimization: Use broadcast variables to efficiently share read-only data across tasks, and use accumulators to perform aggregations in a distributed manner.

Use Case: In graph algorithms like PageRank, using accumulators to calculate the sum of rank contributions for each node can help optimize the computation.

7. Optimal Memory Management:

Optimization: Configure the memory settings of Spark to effectively utilize resources and avoid out-of-memory errors.

Use Case: In memory-intensive applications like natural language processing, adjusting memory fractions for caching and execution can prevent memory-related issues.

8. Data Serialization:

Optimization: Use efficient serialization formats like Kryo to reduce memory usage and improve serialization/deserialization performance.

Use Case: When dealing with large data structures, using optimized serialization can lead to significant memory and performance improvements.

9. Pipeline and DAG Optimization:

Optimization: Design the execution pipeline carefully to minimize unnecessary shuffling and data movement. Leverage transformations that can benefit from narrow dependencies.

Use Case: When performing multiple transformations and actions on a dataset, optimizing the sequence of operations can reduce the overall execution time.

10. Dynamic Allocation and Resource Management:

Optimization: Enable dynamic allocation to allocate and release resources based on the workload's needs, optimizing cluster utilization.

Use Case: In a shared cluster environment, enabling dynamic allocation can ensure efficient resource usage by adjusting resources as needed for different jobs.

1. Spark performance tuning. Share use case

Spark performance tuning involves optimizing various aspects of your Spark applications to achieve better execution speed, resource utilization, and efficiency

Use Case: ETL Processing of Large Log Data

Suppose you are working for a web-based company that generates massive log data from its online services. You need to process these logs to extract useful insights and metrics for analysis. The log data is stored in text files on HDFS, and you're using Spark to perform ETL (Extract, Transform, Load) processing on this data.

1. Challenges faced in spark projects you worked on?

Slow Query Performance: Complex transformations or queries involving multiple shuffles can lead to slow query performance due to excessive data movement and disk I/O.

Real-time Processing Challenges: Implementing real-time processing using Spark Streaming or Structured Streaming introduces challenges related to low latency, event-time handling, and fault tolerance.

1. What is OOM error ? what are the possible reasons ?

An OOM error occurs when a program or application attempts to allocate more memory than what is available in the system, leading to exhaustion of the available memory resources. This can cause the program to crash, freeze, or become unresponsive.

Possible reasons for OOM errors in a Spark project:

* Inefficient Memory Management
* Large Data Shuffling
* Complex Transformations
* Data Volume
* Poor Garbage Collection
* Large Driver Broadcasts

1. How does Spark memory management works?

Spark's memory management is a critical aspect of its performance and resource utilization. It involves allocating and managing memory for various components of a Spark application, including the driver, executors, cached data, shuffle operations, and more. Spark uses a combination of on-heap and off-heap memory to efficiently manage its memory requirements

1. How many stages and task are created.

85 stages and 20 tasks

1. How are executors created in spark.

Executors are dynamically created by the Spark Application Master or Cluster Manager as needed to process tasks from the application

Once resources are allocated, the cluster manager starts creating executors on available worker nodes. Each executor runs as a separate process and manages tasks and data for the application.

1. Explain spark-submit common parameters?

Main Class:

-class or --class: Specifies the fully qualified name of the main class containing the Spark application code.

Deploy Mode:

--deploy-mode: Specifies how the application should be deployed. It can be set to client (run on the client machine) or cluster (run on the cluster).

Master URL:

--master: Specifies the URL of the cluster manager where the application should be submitted. Common values include yarn, local, local[\*], mesos, etc.

Executor Memory:

--executor-memory: Sets the amount of memory to be allocated per executor. Example: 1g, 2g.

Number of Executors:

--num-executors: Specifies the number of executors to allocate for the application.

Driver Memory:

--driver-memory: Sets the amount of memory to be allocated to the driver program.

Application Name:

--name: Specifies a name for the Spark application.

Jar/File Submission:

--jars: Comma-separated list of JAR files to include in the classpath for both the driver and executors.

--files: Comma-separated list of files to be distributed with the application.

Environment Variables:

--conf: Allows you to set various Spark configuration properties. Example: --conf spark.executor.cores=4.

Python Application:

--py-files: Comma-separated list of .zip or .py files to be distributed with a Python application.

Application Arguments:

Pass any arguments required by your Spark application after the spark-submit command.

Package Dependencies:

--packages: Specifies Maven coordinates of packages to be included with the application.

Supervise Mode:

--supervise: Restarts the application if it fails or is killed.

Dynamic Allocation:

--conf spark.dynamicAllocation.enabled=true: Enables dynamic allocation of executors.

Python Specific:

--master local[\*] --py-files your\_file.py: Run a Python script locally.

37. What is data skew? How do you fix it?

Uneven distribution of data among partitions or tasks, causing some tasks to take significantly longer to process than others. Data skew can lead to performance bottlenecks, resource underutilization, and overall inefficiency in data processing jobs.

Fixing Data Skew:

Salting:

Add a random number or string to keys during data insertion to distribute data more evenly. This requires preprocessing the data before it's ingested into Spark.

Bucketing:

Use bucketing to distribute data more evenly. This involves hashing data into predefined buckets, ensuring that each bucket contains a relatively balanced amount of data.

38. What is key salting?

Key salting is a technique used to address data skew in distributed data processing systems like Apache Spark. It involves adding a random or pseudo-random value (called a "salt") to keys before they are used for data partitioning, shuffling, or grouping. The purpose of key salting is to evenly distribute data across partitions, mitigating the effects of skewed data distribution and improving overall performance and resource utilization.

39. What is Adaptive Query Execution?

Adaptive Query Execution (AQE) is a feature introduced in Apache Spark to dynamically optimize and adapt the execution plan of Spark SQL queries based on runtime feedback and statistics