1. What is Apache Spark and why is it used in data engineering?

- Apache Spark is an open-source distributed computing engine designed for large-scale data processing.
- It is widely used in data engineering because it performs in-memory computation, making it much faster than traditional frameworks like Hadoop MapReduce.
- Spark supports various workloads including batch processing, streaming, machine learning, and SQL.
- It integrates well with cloud storage and big data tools, making it ideal for ETL pipelines, data lake processing, and real-time analytics.

2. What are the main components of Apache Spark?

- **Spark Core**: Handles basic I/O functions, task scheduling, and memory management.
- **Spark SQL**: Enables SQL queries on structured data using DataFrames.
- **Structured Streaming**: Processes real-time data as an unbounded table.
- **MLlib**: Scalable machine learning library for regression, classification, etc.
- **GraphX**: For graph processing and computation.
- **Catalyst Optimizer**: Optimizes queries for execution.
- **Tungsten Engine**: Handles physical execution, memory, and CPU efficiency.

3. What is the difference between RDD and DataFrame in Spark?

- **RDD (Resilient Distributed Dataset) :**
  - **fundamental data structure** in Spark.
  - They are immutable, distributed collections of objects that can be **processed in parallel.**
  - RDDs support **fault tolerance** through **lineage information** and **lazy evaluation** for transformations
  - **Low-Level API:** Provides fine-grained control over data and transformations.
  - **No Schema:** RDDs do not have a schema, so you work directly with objects or raw data.
  - **Transformations and Actions:** Supports transformations (e.g., map, filter) and actions (e.g., collect, reduce).
  - **Advantages :**
    - **Flexibility:** Can handle any type of data.
    - **Fault-Tolerant:** Automatically recovers from node failures.
    - Suitable for unstructured data.
  - **Disadvantages :**
    - **Performance:** No optimization or query execution planning; uses Java serialization.
    - **Lack of Schema:** Difficult to perform SQL-like operations.
  - **Use case:**
    - When you need fine-grained control or are working with unstructured or semi-structured data.

- **DataFrames :**
    - DataFrames are similar to tables in a relational database.
    - They are **built on top of RDDs** but offer a higher-level API for manipulating structured data.
    - **Catalyst Optimizer :** DataFrames benefit from the Catalyst optimizer, which can optimize query execution plans, making them easier to use and more performant than RDDs
    - **Advantages :**
        - **Performance:** Uses Catalyst for query optimization and Tungsten for efficient memory management.
        - **Readability:** SQL-like syntax for easier and concise code.
        - **Interoperability:** Supports reading and writing data from various sources (e.g., CSV, Parquet, JSON).
    - **Disadvantages :**
        - **Static Typing:** No compile-time type safety.
        - **Limited Flexibility:** Less control compared to RDDs.
    - **UseCase :**
        - Structured data processing (e.g., relational data, log files).
        - When you need query optimization or SQL-style operations.
- **Datasets :**
    - Datasets combine the best of RDDs and DataFrames.
    - They provide the type safety of RDDs and the optimized execution of DataFrames.
    - Datasets offer a typed API and are also optimized using the Catalyst optimizer, making them ideal for working with structured data where compile-time type checking is beneficial.
    - **Advantages :**
        - **Compile-Time Safety:** Strongly typed API ensures fewer runtime errors.
        - **Performance:** Optimized execution with Catalyst and Encoders.
        - **Versatility:** Can perform both object-oriented transformations and SQL-like operations.
    - **Disadvantages :**
        - **Limited Language Support:** Fully supported only in Scala and Java (not Python).
        - **Slightly Higher Complexity:** Compared to DataFrame for simple operations.
    - **Use Case:**
        - When working with structured data and you need type safety (e.g., processing typed business objects like Case Classes in Scala).

4. What is a transformation and an action in Spark?

- **Transformations :**
    - They are the operations that are applied to create a new RDD.
    - This function is used to transform the data into new RDD without altering the original data. e.g. groupBy, sum, rename, sort,etc

- **Narrow transformations :**
  - Narrow transformations are those for which each input partition will contribute to only one output partition.
  - These transformations are performed on individual partition of data in an RDD in parallel.
  - They do not require shuffling
- **Wide Transformations :**
  - Wide transformation will have **input partitions contributing to many output partitions.**
  - These transformations **require shuffling data between partitions.**

- **Actions:**
  - They are applied on an RDD to instruct Apache Spark to apply computation and send the result back to the driver.
  - e.g. write, collect, show, display, count etc

5. Explain Spark's lazy evaluation model.

- Lazy evaluation in Spark means that **transformations are not executed immediately** when they're defined.
- Instead, Spark builds up a **logical execution plan** in the background and waits until an **action** is triggered to actually run the computation.
- **Transformations** like map, filter, select, or join are **lazy**—they only define what should be done but don't execute anything. These operations return a new DataFrame or RDD, but no actual data processing happens at that point.
- It's only when we perform an **action**—such as show(), collect(), count(), or write()—that Spark **evaluates the entire chain of transformations** and executes the corresponding jobs.
- **Advantages:**
  - **Optimization**: Spark can analyze the full logical plan and optimize it before execution using Catalyst optimizer (for DataFrames)
  - **Efficiency**: It avoids unnecessary computations and can collapse multiple transformations into a single stage during execution.
- For example, if I apply multiple filter operations on a DataFrame, Spark will combine them during execution to reduce I/O and shuffle overhead.
- In short, lazy evaluation helps Spark to be **efficient and scalable** by delaying execution until it has the full context of the job.

6. How does Spark handle fault tolerance?

- Spark handles fault tolerance primarily through a concept called **lineage**.
- Spark tracks the sequence of transformations used to build a dataset—this is known as the **lineage graph**.

- If a partition of data is lost due to a node failure, Spark can **recompute just the lost partition** by replaying the transformations from the original source, rather than restarting the entire job.
- Additionally, for long or expensive lineage chains, Spark supports **checkpointing**. This means persisting data to reliable storage like HDFS, which helps reduce recomputation time in the event of a failure.
- In **Structured Streaming**, Spark provides fault tolerance through **checkpointing and write-ahead logs**, ensuring that even in case of failures, the stream can resume from the last consistent state. When reading from sources like Kafka, Spark tracks **offsets**, ensuring **exactly-once processing semantics**.

7. What is a DAG in Spark and how does it help execution?

- Directed Acyclic Graph is a finite direct graph that performs a sequence of computations on data
- In a Spark DAG, there are **consecutive computation stages** that **optimize the execution plan.**
- You can achieve **fault-tolerance** in Spark with DAG.
- When you define multiple transformations in Spark, they are not executed immediately due to **lazy evaluation**.
- Spark builds a DAG where:
    - **Nodes** represent RDDs or DataFrames.
    - **Edges** represent the transformations (like map, filter, join, etc.) that link one dataset to another.
- Once an **action** is triggered (like count(), collect(), or write()), Spark submits the DAG to the **DAG Scheduler**, which breaks it down into **stages** and **tasks**. These are then submitted to the **Task Scheduler** for parallel execution across the cluster.

8. What is a broadcast variable and when should you use it?

- A **broadcast variable** in Spark is a read-only, shared variable that is **cached on each executor node**, instead of being sent with every task.
- It helps **reduce data transfer overhead**, especially when you have a **large lookup table or reference dataset** that needs to be used across multiple tasks.
- By default, when a variable is used inside a Spark operation, it is sent to each executor with every task, which can be inefficient.
- With a broadcast variable, Spark sends the data only **once to each node**, and then **reuses it** across tasks running on that node.
- **Usecase :**
    - When you have a **small-to-medium sized static dataset** (like a lookup table or config data).
    - When you are performing **joins between a large dataset and a small dataset**—use **broadcast join** for efficiency.

9. What is a shuffle in Spark and why is it expensive?

- In Spark, a **shuffle** is the process of **redistributing data across partitions**, usually between different nodes, to perform operations that require data from multiple partitions.
- During a shuffle, Spark writes intermediate data to **disk**, transfers it **over the network**, and **repartitions** it based on keys or logic.
- This makes it one of the **most expensive operations** in terms of **I/O, network bandwidth, and execution time**.
- **Why it is expensive :**
    - **Disk I/O**: Intermediate data is written to and read from disk.
    - **Network Overhead**: Data is transferred across the cluster, increasing latency and load.
    - **Memory Pressure**: Large shuffles can cause memory spills or even OOM errors if not managed.
    - **Increased Execution Time**: Shuffles often lead to more stages and slower job completion.

10. What is Catalyst Optimizer in Spark SQL?

- An optimizer that automatically finds out the most efficient plan to execute data operations specified in the user's program.
- **How It Works?**
    - **Logical Plan :** series of algebraic or language constructs, as for example: SELECT, GROUP BY or UNION keywords in SQL. It's usually represented as a tree where nodes are the constructs.
    - **Physical Plan :** similar to logical because also represented as a tree. But the difference is that physical plan concerns low level operations.
    - **Unoptimized/Optimized plans :** a plan is considered as unoptimized when optimizer hasn't worked on it yet. The plan becomes optimized when optimizer passed on it and made some optimizations (e.g.: merging filter() methods, replacing some instructions by another ones, most performant).
- **Working**
    - Catalyst applies a **series of rule-based optimizations** to the logical plan. These rules are transformations that convert the logical plan into a more efficient version by simplifying expressions, removing redundant operations, and optimizing predicates.
    - Examples of rule-based optimizations:
        - **Constant Folding**: Simplifies expressions by evaluating constants during query planning.
        - **Predicate Pushdown**: Moves filters as close to the data source as possible, reducing the amount of data read.
        - **Column Pruning**: Removes unnecessary columns from operations to reduce data transfer and memory usage.

- **Catalyst Optimizer Workflow**

1. **Logical Plan**:
    a. The logical plan is an abstract representation of the query's desired outcome, with each stage represented by a tree structure.

   b. The logical plan is created based on DataFrame/Dataset transformations or SQL queries but does not yet consider the actual execution details.
   c. **Example :** When you perform a select or filter on a DataFrame, Spark generates a logical plan showing what operations need to happen without specifying how they will happen.
2. **Optimization Rules**:
   a. Catalyst applies a series of rule-based optimizations to the logical plan.
   b. These rules are transformations that convert the logical plan into a more efficient version by simplifying expressions, removing redundant operations, and optimizing predicates.
   c. Examples of rule-based optimizations:
      i. **Constant Folding**: Simplifies expressions by evaluating constants during query planning.
      ii. **Predicate Pushdown:** Moves filters as close to the data source as possible, reducing the amount of data read.
      iii. **Projection Pruning:** Removes unnecessary columns from operations to reduce data transfer and memory usage.
3. **Physical Plan**:
   a. Once the logical plan is optimized, Spark converts it into **one or more physical plans**, which detail the actual operations to perform on the data.
   b. Each physical plan consists of a series of operators (like scans, filters, joins) that will be executed on Spark's distributed computing engine.
   c. **Multiple physical plans are generated**, and Catalyst then evaluates the cost of each plan to select the most efficient one.
4. **Cost-Based Optimization (CBO)**:
   a. Catalyst uses a **cost model** to **evaluate multiple possible execution plans** and **choose the most optimal one.**
   b. This includes estimating the **cost based on factors like data statistics** (e.g., data size, distribution) and **choosing the plan with the lowest execution cost.**
   c. CBO helps in selecting optimal join strategies (e.g., broadcast joins, shuffle joins) and ordering joins efficiently.
5. **Code Generation (Project Tungsten)**:
   a. Catalyst incorporates Project Tungsten, which **optimizes physical execution by generating Java bytecode at runtime** to **speed up low-level operations.**
   b. This **reduces memory usage** and allows for **efficient use of CPU** resources by inlining and avoiding complex method calls.
6. **Execution**:
   a. The final physical plan is then executed by Spark's distributed execution engine.

11. How does Spark SQL differ from Hive?

- While both support SQL-like syntax and can use Hive Metastore, Spark SQL executes queries in memory using its Catalyst optimizer and Tungsten engine, making it significantly faster than Hive's MapReduce execution model. Spark also integrates seamlessly with DataFrames and Streaming.

12. What are accumulators in Spark?

- Accumulators are write-only variables used for counters or sums that are updated in tasks and read only by the driver.
- They're commonly used for debugging or logging metrics like number of errors, skipped records, etc., during execution.

13. How do you handle skewed data in Spark?

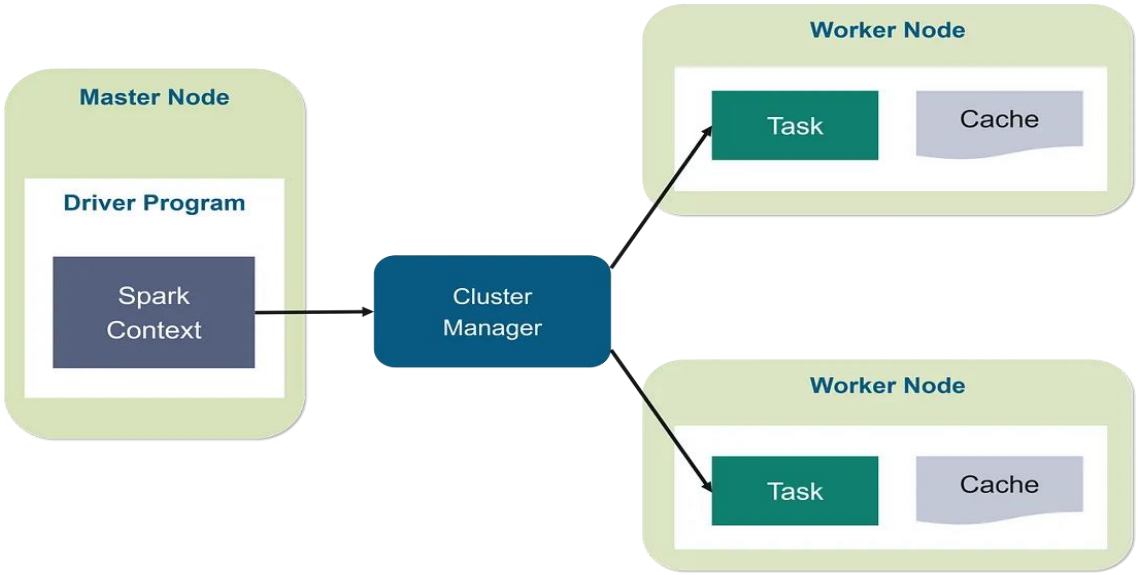
- Handling **skewed data** in Spark is important because it can lead to **performance bottlenecks**, where a few tasks process significantly more data than others.
- This results in uneven load distribution, **stragglers**, and longer job execution times.
- **Data skew can be mitigated by techniques like:**
    a. Salting keys to distribute load.
    b. Using broadcast joins for small tables.
    c. Filtering out or handling skewed keys separately.
    d. Increase parallelism
        i. Using repartition
        ii. using AQE(spark configs)
    e. Custom partitioning/liquid clustering during writes to balance task loads.

14. What is checkpointing in Spark and when is it used?

- **Checkpointing** in Spark is the process of **persisting RDD or DataFrame data to reliable storage**, like HDFS or cloud storage, to **truncate lineage** and make the data fault-tolerant.
- Normally, Spark relies on **lineage information** to recompute lost partitions.
- But in cases of **long or complex transformation chains**, this recomputation becomes expensive or even unstable. Checkpointing helps by saving the actual data to disk, so Spark can **reload it directly**, instead of replaying the entire lineage.
- **When to use :**
    a) **Long Lineage Chains**:
        i) When an RDD or DataFrame has a long chain of transformations, checkpointing is used to cut the lineage and avoid stack overflow or recomputation issues.
    b) **Iterative Algorithms**:
        i) In algorithms like PageRank or GraphX jobs, intermediate RDDs are reused multiple times. Checkpointing ensures stability across iterations.
    c) **Fault Tolerance in Streaming**:
        i) In **Structured Streaming**, checkpointing is critical. It saves:
        ii) **Metadata about progress** (e.g., Kafka offsets).
        iii) **Intermediate state** for stateful operations (like window aggregations).
        iv) This enables the stream to recover and resume from the last successful batch in case of failure.

d) **Data Recovery**:
   i) When reliability is more important than performance (e.g., in critical pipelines), checkpointing ensures that computation can resume from a stable point.

15. Explain the role of driver and executor in Spark.



- The Spark follows the **master-slave architecture**. Its cluster consists of **a single master and multiple slaves.**
- **In your master node, you have the driver program**, which drives your application. The **code you are writing behaves as a driver program** or if you are using the interactive shell, the **shell acts as the driver program**.
- Driver Program in the Apache Spark architecture **calls the main program** of an application and **creates SparkContext.**
- The **purpose of SparkContext is to coordinate with the spark applications**, running as independent sets of processes on a cluster.
- The role of the **cluster manager is to allocate resources across applications.** It maintains a cluster of machines that will run Spark applications.
- It **consists of various types of cluster managers** such as Hadoop YARN, Apache Mesos, and Standalone Scheduler. Here, the **Standalone Scheduler** is a standalone spark cluster manager that **facilitates to the installation of Spark on an empty set of machines.**
- Worker nodes are the slave nodes whose job is to basically execute the tasks.
- Spark Context takes the job, breaks the job into tasks, and distributes them to the worker nodes. These tasks work on the partitioned RDD, perform operations, collect the results and return to the main Spark Context.

- **WORKING OF SPARK:**
  - When the Driver Program in the Apache Spark architecture executes, it calls the real program of an application and creates a SparkContext. SparkContext contains all of the basic functions.

- The **Spark Driver includes** several other components, including a **DAG Scheduler, Task Scheduler, Backend Scheduler, and Block Manager**, all of which are **responsible for translating user-written code into jobs** that are actually executed on the cluster.
- The **Cluster Manager manages the execution of various jobs in the cluster**. **Spark Driver works in conjunction with the Cluster Manager to control the execution of various other jobs.**
- The cluster Manager does the **task of allocating resources** for the job. Once the job has been broken down into smaller jobs, which are then distributed to worker nodes, SparkDriver will control the execution.
- Many worker nodes can be used to process an RDD created in SparkContext, and the results can also be cached.
- The Spark Context receives task information from the Cluster Manager and enqueues it on worker nodes.
- The executor is in charge of carrying out these duties. The lifespan of executors is the same as that of the Spark Application. We can increase the number of workers if we want to improve the performance of the system. In this way, we can divide jobs into more coherent parts.

16. How do you optimize Spark jobs?

1. **Parquet/Delta file format** :
    a. Use for data stored on disk or transmitted over the network, because it has very efficient compression and It has lot of other features like columnar format, file statistics, partitioning, z ordering etc. **compressing with Snappy** (or another efficient codec) helps cut down on I/O times.
2. **Partitioning :**
    a. Partitioning plays a huge role in Spark's parallelism. **adjust partition sizes** to balance load (ideally around **128 MB per partition**), and **repartition()** to increase partitions or **coalesce()** to reduce them based on the stage requirements. For **skewed data**, use techniques like adding salts or custom partitioning to avoid hotspot partitions.
3. **Cache and Persistance :**
    a. When a dataset is reused across stages, use cache() or persist() to avoid re-computation. Choosing the **right storage level** (e.g., MEMORY_ONLY or MEMORY_AND_DISK) is also critical depending on the cluster's memory constraints. Ensure **Partition count should be in multiple of core count.**
4. **Join Optimization :**
    a. Spark joins can be costly, so I focus on **avoiding shuffles** by partitioning join keys and leveraging **broadcast hash joins** for smaller tables. If I encounter skewed joins, I address them by partitioning or using skew hints. **Bucketing** also helps in join optimization, by shuffling, write data prior to joins
5. **Serialization:**
    a. To reduce the data size in transit, We can use **Kryo Serialization** over the default Java serialization because it's **faster** and more **memory-efficient.**
6. **Predicate Pushdown:**

    a. Predicate Pushdown is an optimization technique used in databases and file storage systems. It's a methodology where the filtering of data, i.e., the 'WHERE' clause of your SQL query, is pushed as down as possible in the database query execution plan.
    b. The idea is to **move the filtering operation close to the data source**, thereby reducing the amount of data that needs to be processed or transferred over the network. This results in improved performance and speed.
7. **Catalyst Query Optimizer (Spark SQL):**
    a. When working with DataFrames or Spark SQL, rely on **Catalyst optimizer** to **automatically apply** optimizations like **predicate pushdown, column pruning, and join reordering.**
    b. Additionally, **Cost-Based Optimization (CBO)** can further help reorder joins efficiently by considering table statistics.
8. **Adaptive Query Execution (AQE):**
    a. Enable **Adaptive Query Execution (AQE)** for runtime optimizations. AQE dynamically optimizes shuffle partitions, joins, and partition pruning based on the data characteristics, which often leads to better performance without manual tuning.
9. **Resource Allocation and Parallelism :**
    a. Allocate executor memory and cores based on job needs, and adjust **spark.sql.shuffle.partitions** and **spark.default.parallelism** to optimize parallelism.
    b. Enabling **dynamic resource allocation** also allows Spark to add or remove executors based on demand.
10. **Avoiding Wide Transformations :**
    a. Wide transformations like groupByKey and distinct create shuffles. use reduceByKey instead of groupByKey when possible to minimize shuffles, and generally aim to reduce wide transformations to keep stages lightweight.
11. **GC Tuning:**
    a. For large datasets, I tune **garbage collection settings** (e.g., -XX:MaxGCPauseMillis, -XX:+UseG1GC) to manage memory effectively, reducing overhead from frequent GC pauses.

17. Difference between repartition() and coalesce()?

- **repartition()**
    - **Increases or decreases** the number of partitions.
    - Always performs a **full shuffle** of the data across the cluster.
    - Creates **equal-sized partitions**, which is useful for optimizing parallelism.
    - Costlier due to the shuffle, but more balanced in terms of data distribution.
    - **Use case**: When you need to **increase partitions** or **evenly distribute** data before a heavy transformation or write operation.
- **Coalesce()**
    - Used only to **reduce** the number of partitions.
    - Avoids full shuffle by **merging adjacent partitions**, keeping data movement minimal.
    - Faster than repartition() but can lead to **uneven partition sizes** (data skew).

- **Use case**: Ideal for **writing to fewer output files**, like saving to Parquet or CSV, where fewer partitions = fewer files

18. What is PySpark, how does it differ from pandas?

- **PySpark** is the **Python API for Apache Spark**, which allows you to write Spark applications using Python.
- It enables distributed computing for large-scale data processing using Spark's core engine, but with Pythonic syntax.
- In contrast, **pandas** is a popular Python library for **in-memory data manipulation and analysis**, mostly used on single-machine datasets that fit into memory.

1. **Execution Model :**
   a. **Pyspark :** Distributed (across cluster nodes)
   b. **Pandas :** Single-machine, in-memory
2. **Performance :**
   a. **Pyspark :** Optimized for large-scale (TBs) datasets
   b. **Pandas :** Fast for small-to-medium datasets
3. **Memory Usage :**
   a. **Pyspark :** Handles out-of-memory data via partitioning
   b. **Pandas :** Limited by local machine RAM
4. **Parallelism :**
   a. **Pyspark :** Built-in across nodes
   b. **Pandas :** Limited to single core/thread (unless using Dask)
5. **Fault Tolerance :**
   a. **Pyspark :** Yes (via lineage and DAG)
   b. **Pandas :** No built-in fault tolerance
6. **Best Use Case :**
   a. **Pyspark :** Big data processing, ETL, analytics pipelines (Large datasets)
   b. **Pandas :** Small-scale data analysis, prototyping

19. How does Spark Streaming work?

- **Spark Streaming** is a component of Apache Spark that enables **real-time data processing**.
- It works by **ingesting live data streams**, dividing them into small batches (called **micro-batches**), and then processing each batch using Spark's core APIs.

20. Common use cases of Apache Spark in data engineering?

- Batch and real-time ETL pipelines.
- Log processing and monitoring.
- Real-time analytics (e.g., Spark + Kafka).

- Machine learning model training and scoring.
- Building data lakes using Delta Lake.
- Fraud detection and anomaly detection pipelines.
- Handling large-scale data transformations in the cloud.