Here are **20 PySpark and Apache Spark** questions

Basic Level

1. **What is Apache Spark, and how is it different from Hadoop MapReduce?**
   - Apache Spark is a **fast, distributed data processing framework** that processes data **in-memory**, unlike Hadoop MapReduce, which relies on disk I/O. This makes Spark significantly faster for iterative computations and batch processing.
2. **What are the key components of Spark architecture?**
   - **Driver Program** (controls execution), **Cluster Manager** (allocates resources), **Executors** (run tasks on worker nodes), and the **Task Scheduler** (assigns tasks). These components work together to distribute and process data efficiently.
3. **What is RDD (Resilient Distributed Dataset), and why is it important in Spark?**
   - RDD is the **core data structure** in Spark, providing **fault tolerance and parallel processing**. It is immutable, distributed across nodes, and supports **lazy evaluation**, improving efficiency in large-scale computations.
4. **How does Spark handle fault tolerance?**
   - Spark maintains **RDD lineage**, allowing it to recompute lost partitions instead of replicating data. It also supports **checkpointing** for long computations to reduce recomputation overhead.
5. **What is the difference between Spark RDD, DataFrame, and Dataset?**
   - **RDD**: Low-level, unstructured, optimized for transformations.
   - **DataFrame**: Schema-based, optimized using Catalyst Optimizer, similar to SQL tables.
   - **Dataset**: Type-safe, combines benefits of RDD and DataFrame (**only in Scala/Java**).

Intermediate Level

6. **Explain the execution flow of a Spark job (DAG, Stages, and Tasks).**
   - Spark constructs a **DAG (Directed Acyclic Graph)** of transformations, breaks it into **stages** (based on shuffle operations), and further divides them into **tasks** that are executed in parallel by worker nodes.
7. **How does Spark handle lazy evaluation? Why is it useful?**
   - Transformations in Spark are **lazy**, meaning they don't execute immediately but build a logical execution plan (DAG). This prevents unnecessary computations and optimizes performance by combining operations where possible.
8. **What are transformations and actions in Spark? Give examples.**
   - **Transformations** (lazy) create a new RDD/DataFrame (e.g., `map()`, `filter()`).
   - **Actions** trigger execution and return results (e.g., `collect()`, `count()`, `show()`).
9. **What is a Spark driver, executor, and cluster manager?**

    - The **driver** coordinates execution, the **executors** run tasks on worker nodes, and the **cluster manager** (Standalone, YARN, Mesos, Kubernetes) manages resources and scheduling across the cluster.

10. **How does Spark handle data shuffling, and how can you optimize it?**

- Shuffling occurs when data is **redistributed across partitions**, often during operations like `groupByKey()`. It can be optimized by using **reduceByKey() instead of groupByKey()**, **broadcast joins**, and **increasing shuffle partitions**.

11. **Explain wide and narrow transformations in Spark.**

- **Narrow transformations** (e.g., `map()`, `filter()`) don't require data movement across partitions. **Wide transformations** (e.g., `groupBy()`, `reduceByKey()`) involve shuffling, which is costly but necessary for aggregation.

12. **What is a broadcast variable in Spark? When should you use it?**

- A **broadcast variable** is a read-only variable shared across executors to avoid sending large datasets multiple times. It is useful for **joining a small dataset with a large one**, reducing network overhead.

13. **What are accumulators in Spark, and how do they work?**

- **Accumulators** are shared, write-only variables used to aggregate values across tasks, commonly for counting or summing values. They are mainly used for **debugging, monitoring, and logging** purposes.

14. **How do you handle skewed data in Spark?**

- Data skew can be addressed by **salting (adding random keys to distribute data more evenly), using broadcast joins for small tables, and repartitioning data** to balance the load across worker nodes.

15. **What is the difference between cache() and persist() in Spark?**

- Both store data in memory for faster access, but `cache()` stores it only in memory, while `persist()` allows different storage levels (e.g., **disk, memory, or both**), providing flexibility in resource management.

Advanced Level

16. **How does PySpark optimize performance?**

- PySpark optimizes performance using **lazy evaluation, DAG execution, Catalyst Optimizer (for SQL queries), and Tungsten (for memory management)**. Additionally, optimizations like **broadcast joins, partitioning, and caching** improve execution speed.

17. **What is Spark SQL, and how does it integrate with DataFrames?**

- Spark SQL allows users to run **SQL queries on structured data** and integrates seamlessly with **DataFrames**, which are optimized for performance using the Catalyst Optimizer. You can use **SQL queries within PySpark** or convert DataFrames to temporary tables for SQL operations.

18. **What is the difference between Spark's default shuffle partitions and manually setting them?**

- By default, Spark sets **shuffle partitions to 200**, which may not be optimal for all workloads. Manually tuning partitions using `spark.sql.shuffle.partitions` can **improve performance** by balancing the load across executors based on the dataset size.

19. **How do you integrate Spark with external databases such as MySQL or Postgres?**

- Spark integrates with databases using **JDBC connectors**. You can use `spark.read.format("jdbc")` to load data from **MySQL, PostgreSQL, or other databases**, and **write back** using `df.write.jdbc()`. Performance can be optimized with **partitioning and parallel reads**.

20. **Can you explain the Catalyst Optimizer in Spark SQL? How does it improve query performance?**

- Catalyst Optimizer is an **advanced query optimizer** in Spark SQL that **analyzes and optimizes query execution plans**. It applies techniques like **predicate pushdown, constant folding, and column pruning** to improve performance and reduce computation overhead.