Most Asked Spark Concepts

Data Engineering Interview Preparation

January 2026

1. Architecture & Execution Model (1–10)

1. What is the Spark Driver?

Answer

The Spark Driver is the main process that runs the user application. It creates the SparkSession, builds the DAG, schedules jobs, and tracks execution. The driver communicates with the cluster manager to request executors. If the driver fails, the entire application fails.

```
spark = SparkSession.builder.appName("Driver").getOrCreate()
```

2. What are Executors?

Answer

Executors are JVM processes running on worker nodes. They execute tasks assigned by the driver and store intermediate data. Executor memory and cores directly impact performance and parallelism.

```
--num-executors 4 --executor-memory 4G
```

3. SparkSession vs SparkContext

Answer

SparkSession is the unified entry point since Spark 2.0. It internally manages SparkContext and SQL functionality. Developers should always use SparkSession in modern Spark applications.

```
spark.sparkContext
```

4. RDD vs DataFrame

Answer

RDDs are low-level collections with no schema or optimization. DataFrames are schema-based and optimized by Catalyst and Tungsten. DataFrames are preferred for production workloads.

```
df = spark.createDataFrame([(1,"A")],["id","val"])
df.rdd
```

5. What is a DAG?

Answer

A DAG represents the logical execution plan of transformations. Spark analyzes the DAG to optimize execution. It helps Spark divide execution into stages.

```
df.groupBy("dept").count().explain()
```

6. Lazy Evaluation

Answer

Spark does not execute transformations immediately. Execution happens only when an action is triggered. This enables Spark to optimize query execution.

```
df.filter(df.age > 30)
df.count()
```

7. Transformations vs Actions

Answer

Transformations define computations but do not execute them. Actions trigger execution and return results. Understanding this helps debug performance issues.

```
df.select("id")   # transformation
df.collect()      # action
```

8. Narrow vs Wide Transformations

Answer

Narrow transformations operate within a single partition. Wide transformations require data redistribution. Wide transformations cause shuffle and new stages.

```
df.filter(df.age>20)
df.groupBy("dept").count()
```

9. What is Shuffle?

Answer

Shuffle redistributes data across executors. It involves network and disk I/O, making it expensive. Reducing shuffle is critical for performance.

```
df.join(df2, "id")
```

10. Job vs Stage vs Task

Answer

An action creates a Job. Shuffle boundaries define Stages. Each partition in a stage is a Task.

```
df.groupBy("dept").count().collect()
```

2. Performance Tuning & Optimization (11–20)

11. repartition vs coalesce

Answer

repartition performs a full shuffle. coalesce reduces partitions with minimal shuffle. Use coalesce before writing data.

```
df.repartition(10)
df.coalesce(2)
```

12. What is Data Skew?

Answer

Data skew occurs when some partitions contain much more data. This causes long-running tasks. Common in joins and aggregations.

```
-- Seen in Spark UI
```

13. Salting Technique

Answer

Salting adds randomness to skewed keys. It distributes data evenly across partitions. Useful for skewed joins.

```
df.withColumn("salt", F.rand())
```

14. Broadcast Join

Answer

Broadcast joins send a small table to all executors. This avoids shuffle. Used when one dataset is small.

```
df_big.join(broadcast(df_small),"id")
```

15. Adaptive Query Execution

Answer

AQE optimizes queries at runtime. It can change join strategies dynamically. Enabled by default in Spark 3.x.

```
spark.conf.set("spark.sql.adaptive.enabled","true")
```

16. Predicate Pushdown

Answer

Filters data at the storage layer. Reduces I/O and improves performance. Supported by Parquet and ORC.

```
spark.read.parquet("data").filter("age>30")
```

17. cache vs persist

Answer

cache stores data in memory only. persist allows disk-based storage. persist is safer for large datasets.

```
df.persist(StorageLevel.MEMORY_AND_DISK)
```

18. Whole-Stage Code Generation

Answer

Combines operators into one JVM function. Reduces CPU overhead. Improves query performance.

```
df.explain(mode="codegen")
```

19. Handling OOM Errors

Answer

OOM occurs when executor memory is insufficient. Can be fixed by repartitioning or increasing memory. Proper partition sizing is critical.

```
df.repartition(500)
```

20. Shuffle Partition Tuning

Answer

Default shuffle partitions is 200. Tune based on data size and cluster. Optimal task size is 128MB.

```
spark.conf.set("spark.sql.shuffle.partitions","500")
```

3. Spark SQL & DataFrames (21–30)

21. Why Parquet?

Answer

Parquet is a columnar storage format. Supports compression and predicate pushdown. Ideal for analytics workloads.

```
df.write.parquet("/data")
```

22. Broadcast Variables

Answer

Broadcast variables share read-only data. Data is sent once per executor. Used to optimize joins and lookups.

```
sc.broadcast({1:"A"})
```

23. Accumulators

Answer

Accumulators are write-only shared variables. Used for counters and metrics. Not reliable for business logic.

```
acc = sc.accumulator(0)
```

24. explode()

Answer

explode converts array elements into rows. Used to flatten nested data. Common in JSON processing.

```
df.select(explode("items"))
```

25. Window Functions

Answer

Window functions compute values over partitions. They do not collapse rows. Used for ranking and running totals.

```
Window.partitionBy("dept").orderBy("salary")
```

26. Rank vs Dense Rank

Answer

Rank skips numbers when ties occur. Dense rank does not skip numbers. Used in leaderboard scenarios.

```
rank().over(w)
dense_rank().over(w)
```

27. Bucketing

Answer

Bucketing organizes data into fixed buckets. Optimizes joins and aggregations. Requires same bucket count.

```
df.write.bucketBy(8,"id").saveAsTable("t")
```

28. UDF Optimization

Answer

UDFs bypass Spark optimizations. They are slower than built-in functions. Prefer Spark SQL or Pandas UDFs.

```
@pandas_udf("int")
def add1(x): return x+1
```

29. mapPartitions()

Answer

mapPartitions processes data partition-wise. Useful for database connections. More efficient than row-wise operations.

```
rdd.mapPartitions(func)
```

30. Catalyst Optimizer

Answer

Catalyst optimizes logical plans. Applies rule-based and cost-based optimization. Generates efficient physical plans.

```
df.explain()
```

4. Structured Streaming (31–35)

31. Structured Streaming

Answer

Structured Streaming is built on Spark SQL. Provides exactly-once semantics. Supports event-time processing.

```
spark.readStream.format("kafka")
```

32. Watermark

Answer

Watermarks define allowed lateness. Prevent unbounded state growth. Used with event-time windows.

```
df.withWatermark("ts","10 minutes")
```

33. Checkpointing

Answer

Checkpointing stores stream state. Enables fault recovery. Required for exactly-once processing.

```
.option("checkpointLocation","/chk")
```

34. Output Modes

Answer

Append writes only new rows. Update writes changed rows. Complete writes entire result.

```
.outputMode("append")
```

35. Trigger Types

Answer

Triggers control execution frequency. Supports processing-time and once. Continuous triggers enable low latency.

```
.trigger(processingTime="5 seconds")
```

5. Advanced Engineering (36–50)

36. Dynamic Resource Allocation

Answer

Automatically scales executors. Improves resource utilization. Useful in shared clusters.

```
spark.dynamicAllocation.enabled=true
```

37. Speculative Execution

Answer

Re-executes slow tasks. Reduces job completion time. Useful in unstable clusters.

```
spark.speculation=true
```

38. Small File Problem

Answer

Too many small files increase metadata overhead. Common in streaming pipelines. Solved using compaction.

```
df.coalesce(1)
```

39. Data Locality

Answer

Executes tasks close to data. Reduces network I/O. Improves performance.

```
-- Managed by Spark automatically
```

40. Delta Lake ACID

Answer

Provides ACID transactions. Supports time travel and schema enforcement. Widely used in Databricks.

```
df.write.format("delta")
```

41. Schema Evolution

Answer

Allows tables to handle schema changes. Prevents pipeline failures. Useful in evolving data sources.

```
.option("mergeSchema","true")
```

42. Z-Ordering

Answer

Optimizes data skipping. Improves query performance. Used on high-cardinality columns.

```
OPTIMIZE table ZORDER BY (col)
```

43. Partitioning vs Z-Order

Answer

Partitioning works for low cardinality. Z-order works for high cardinality. Both improve query pruning.

```
df.write.partitionBy("date")
```

44. Join Hinting

Answer

Explicitly tells Spark join strategy. Used for performance tuning. Overrides optimizer decisions.

```
df.hint("broadcast")
```

45. Spark UI

Answer

Spark UI helps monitor jobs. Shows stages, tasks, and memory usage. Critical for debugging performance.

```
http://localhost:4040
```