Day 1 – PySpark Core Fundamentals (L4 Google/Uber Interview Level)

---

1. What is the architecture of Spark and how does it process a PySpark job?

Problem Context:
You need to explain how Spark executes PySpark code end-to-end and how it scales computation.

Explanation:

- PySpark is a Python API on top of Apache Spark.
- Spark runs in a cluster (YARN/Mesos/K8s or Standalone) with 3 main components:
    - **Driver:** Initiates SparkSession, parses the job, creates DAG.
    - **Cluster Manager:** Allocates resources.
    - **Executors:** Perform the actual computation.
- A PySpark job is translated to a DAG of stages → tasks → executed by executors.

Trade-offs / Design Insights:

- Centralized driver becomes a bottleneck for long lineage.
- Need careful executor memory tuning.

Real-world Example (Uber):

- Fare prediction job: PySpark script reads ride logs, cleans data, computes features → DAG splits into 2 stages due to shuffle.

Key Takeaways:

- Always visualize Spark DAGs for performance bottlenecks.
- Spark jobs = Logical DAG → Physical Plan → Tasks.

---

2. RDD vs DataFrame vs Dataset. Why is DataFrame preferred?

Explanation:

- **RDD:** Low-level, strongly-typed distributed collection. Full control.
- **DataFrame:** Optimized, schema-aware tabular structure.
- **Dataset:** Java/Scala only. Combines both worlds.

Why DataFrame:

- Catalyst Optimizer → Logical to Physical plan conversion
- Tungsten engine → Code generation
- Less boilerplate code, better performance

Trade-offs:

- RDD needed for fine control (e.g., custom partitioners).
- DataFrames hide complexity but reduce flexibility.

Example (Uber):

- Fraud detection → DataFrame for SQL-style joins on large volumes.

Takeaway:
Use DataFrame unless deep customization is needed.

---

3. Lazy Evaluation in Spark

Explanation:

- Transformations (map, filter, join) are lazy → Not executed immediately
- Actions (count, collect, write) trigger execution

Why important:

- Enables pipeline optimization
- Minimizes data shuffling

Trade-off:

- Harder debugging; need `.explain()` to verify plan

Uber Use Case:

- ETL pipeline does `filter → map → write`. Spark optimizes this chain before execution.

Takeaway:
Understand how lazy evaluation builds DAG → always inspect before triggering.

---

4. Transformations vs Actions

Transformations:

- Return a new RDD/DataFrame
- Examples: map, filter, join, select

Actions:

- Trigger execution
- Examples: count, show, write, collect

Trade-off:

- Using too many actions like `.collect()` → driver OOM.

Uber Use Case:

- Driver behavior aggregation → groupBy (transformation), count (action)

Takeaway:
Minimize actions; avoid collect on large datasets.

---

5. What is Lineage in Spark?

Explanation:

- Tracks all transformations applied to data → like a recipe
- Enables **Fault Tolerance**: lost partitions can be recomputed

Trade-off:

- Long lineage chains = expensive to recompute → use `checkpoint()`

Uber Example:

- Long transformation chain on ride events → checkpoint at critical points

Takeaway:
Checkpoint periodically for resilience in large jobs

---

6. What is DAG in Spark?

Explanation:

- Directed Acyclic Graph: Nodes = stages; Edges = dependencies
- Built during job creation → optimized before execution

Trade-off:

- Wide dependencies = shuffle = expensive

Uber Example:

- `groupBy → join → write`: generates multiple stages; must visualize DAG

Takeaway:
Use Spark UI to understand DAG execution, stage splits.

---

7. Partitioning in Spark

Explanation:

- Controls parallelism and data locality
- `repartition(n)` → full shuffle
- `coalesce(n)` → reduce partitions without shuffle

Trade-off:

- Over-partitioning = overhead; Under-partitioning = skew

Uber Example:

- Repartition by driver ID before join → avoids skewed join

Takeaway:
Use `repartition` before wide ops; `coalesce` before sink.

---

8. Narrow vs Wide Transformation

Narrow:

- No shuffle. Example: map, filter

Wide:

- Requires shuffle. Example: groupBy, join

Trade-off:

- Wide = costly in network I/O

Uber Example:

- Join ride data with payment = wide → optimize via bucketing

Takeaway:
Minimize wide transformations or optimize them with partitions.

---

9. Shuffle in Spark

Explanation:

- Data movement across partitions → triggered by wide transformations
- Requires sorting, aggregating, writing to disk

Downside:

- High network + disk I/O, slows down jobs

Uber Example:

- `groupBy location` causes huge shuffle → use `salting` to reduce skew

Takeaway:
Shuffle = bottleneck; reduce keys, pre-partition when possible.

---

10. Stages and Tasks in Spark

Explanation:

- **Stage:** Set of tasks with no shuffle boundary
- **Task:** Unit of work per partition

Uber Use Case:

- Job split into 2 stages: Read, Transform (Stage 1); Shuffle & Write (Stage 2)

Takeaway:
Check Spark UI for stage durations; optimize stages that take longest.

---

11. What are UDFs in PySpark?

UDFs (**User Defined Functions**) let you apply **custom row-level logic** using Python/Scala/Java when built-in functions don't suffice.

🔹 Why UDFs Are Slow and Not Optimized (Trade-offs)

❌ Limitation	⚠️ Impact / Trade-off
Not Catalyst-Optimized	Treated as black-box → no predicate pushdown, column pruning, or query plan rewrites.
Bypass Tungsten Engine	No JVM codegen or low-level memory optimization → **slower execution**.
Serialization Overhead	Data shuffles between JVM ↔ Python (via Py4J) → high latency and CPU cost.
Opaque to Monitoring	Hard to introspect or tune in Spark UI → difficult to debug and optimize.
🔴 **Overall Performance**	**SLOW** compared to native functions — bad for large data or critical paths.

🔹 Better Alternatives

✅ Option	Why It's Better
Built-in SQL Functions	Fully optimized by Catalyst + Tungsten → fastest, supports pushdowns.
Pandas UDFs	Vectorized, Arrow-based → faster than row-wise Python UDFs.
`expr()`, `selectExpr()`	Native SQL → enables Catalyst optimization and efficient query planning.

Scala/Java UDFs	JVM-native → better than PySpark UDFs but still lacks Catalyst support.

🔹 When to Use UDFs

✔ Use **only when** logic **cannot** be expressed using SQL or built-in functions.
✘ **Avoid** in joins, filters, or performance-critical pipelines.

12) SparkContext vs SparkSession :

🔍 Aspect	⚙️ SparkContext	⚡ SparkSession
API Level	Low-level (RDD only)	High-level (DataFrame, SQL, RDD, Streaming, etc.)
Version	Legacy (Spark ≤1.x)	Modern (Spark ≥2.0)
Features Supported	RDD, Accumulators, Broadcast	All features (SQL, Hive, Catalog, UDFs, RDD, etc.)
Entry Point For	Core engine	Full Spark stack
Optimization Support	❌ No Catalyst / Tungsten	✅ Supports Catalyst + Tungsten (SQL/DataFrame)
Usage in Modern Apps	Not used directly	✅ Preferred interface

Production level Scenario Question

1) You notice your PySpark job is running slower than expected. After investigating, you realize it's processing a large number of small files (a classic small-file problem).🔁 How would you solve this issue and improve performance?

Ans : Repartition/ Coalesce the DF before writing

Trade-offs:

- Coalesce reduces partitions but no shuffle, efficient but limited control.
- Repartition triggers shuffle, higher overhead but better data balancing.

Takeaway:
Always validate using Spark UI → check if your stages have many small tasks (e.g., 1–2 MB input each).
Use coalesce during writes to minimize file explosion.
I answered : **Broadcast Join -** Which is somewhat wrong

2) Over time, your Spark job is consuming **more memory and slowing down**. You suspect **unused cached data** is bloating memory.

🔁 **How would you fix or prevent this in a production pipeline?**

Ans : In one case, replacing `.cache()` **with** `.persist(StorageLevel.MEMORY_AND_DISK_SER)` **and explicitly calling** `.unpersist()` **after joins saved 6GB memory per executor and cut job time by 20%.**
I answered - only persist - memory and disk