**Apache Spark**

**1. Spark Architecture**

**1.1 Driver and Executor**

* **Driver Program:**
  + Runs the main() function.
  + Coordinates Spark application.
  + Translates user code into a Directed Acyclic Graph (DAG) of stages/tasks.
  + Manages the cluster resource manager (YARN, Kubernetes, or Standalone).
  + Sends tasks to executors.
* **Executors:**
  + Run on worker nodes.
  + Execute tasks and return results to the driver.
  + Responsible for memory storage, task execution, and caching.

**1.2 Cluster Manager**

* Allocates resources and manages containers (Spark on YARN, Kubernetes, Mesos).

**1.3 Shuffle**

* Occurs when data is redistributed between stages (e.g., groupByKey, join).
* Expensive because it involves disk I/O and network transfer.
* Spark breaks jobs into **stages** separated by shuffle boundaries.

**2. DataFrame API Essentials**

**2.1 Key Operations**

* **Creation:**

python

df = spark.read.csv("data.csv", header=True, inferSchema=True)

* **Transformation:**
  + .select(), .filter(), .groupBy(), .agg()

python

df.filter("amount > 100").groupBy("category").agg(sum("amount"))

**2.2 Column Operations**

* Column expressions:

python

df.withColumn("tax", df.amount \* 0.18)

df.selectExpr("amount \* 0.18 as tax")

**3. Cache vs Checkpoint**

**3.1 Caching**

* Keeps intermediate results in memory.
* Useful for iterative algorithms or reused datasets.
* Syntax:

python

df.cache()

**3.2 Checkpoint**

* Saves DataFrame/RDD to disk and truncates the lineage.
* Prevents long lineage failures and memory issues.
* Requires setting a checkpoint directory:

python

spark.sparkContext.setCheckpointDir("/tmp/checkpoint")

df.checkpoint()

| **Use When** | **Cache** | **Checkpoint** |
| --- | --- | --- |
| Reusing DataFrame | ✅ | ❌ |
| Long/Complex lineage | ❌ | ✅ |
| Stored on | Memory | Disk |

**4. Spark UI Deep-Dive**

**4.1 Components**

* **Jobs Tab**: Shows all triggered jobs.
* **Stages Tab**: Breaks jobs into stages with shuffle info.
* **Tasks Tab**: Per-stage task execution metrics.
* **SQL Tab**: Shows logical and physical plans for SQL/DataFrame operations.
* **Storage Tab**: Cached RDD/DataFrame details.
* **Executors Tab**: Memory and task usage per executor.

**4.2 Shuffle Metrics**

* Look for “Shuffle Read” / “Shuffle Write” sizes.
* Uneven task times may indicate skewed data or improper partitioning.

**5. RDD vs DataFrame**

| **Feature** | **RDD** | **DataFrame** |
| --- | --- | --- |
| Abstraction Level | Low-level (row-by-row) | High-level (schema + optimization) |
| Performance | Slower, no optimization | Optimized via Catalyst & Tungsten |
| Type Safety | Compile-time (Java/Scala) | Runtime (Python/SQL API) |
| Use Cases | Complex transformations | Structured data, analytics |

**Recommendation:**  
Use **DataFrame API** for most applications unless low-level control is required.

**6. Parquet Partition Discovery**

**6.1 Why Partition?**

* Speeds up reads using **partition pruning**.
* Directory layout:

bash

/data/year=2023/month=01/

**6.2 How Spark Uses It**

* When reading:

python

df = spark.read.parquet("/data/")

df.filter("year = 2023 AND month = 01")

* Use .explain(True) to confirm partition pruning.

**7. Broadcast Hints**

**7.1 Problem**

* Joining large datasets causes expensive shuffles.

**7.2 Solution – Broadcast Join**

* Broadcast the smaller table to all executors.
* Spark decides automatically, or you can force it:

python

from pyspark.sql.functions import broadcast

df = large\_df.join(broadcast(small\_df), "key")

**7.3 Use Case**

* Small lookup tables (regions, reference data) with large fact tables.

**8. External Source Read (Blob/ADLS)**

**8.1 Azure Data Lake / Blob Storage Access**

* Mount or access via service principal or SAS token.
* Sample config:

python

spark.conf.set("fs.azure.account.key.<account>.blob.core.windows.net", "<key>")

df = spark.read.parquet("wasbs://container@account.blob.core.windows.net/data/")

**8.2 Partition Pruning and Performance**

* Ensure folder layout matches partition columns.
* Use .explain() to confirm predicate pushdown.

**9. (Optional) Advanced UDF Performance Profiling**

**9.1 Regular UDFs**

* Slower due to JVM-Python serialization (PySpark).
* Not optimized by Catalyst.

**9.2 pandas\_udf**

* Vectorized, efficient, operates on batches.
* Faster for row-wise computation:

python

from pyspark.sql.functions import pandas\_udf

@pandas\_udf("double")

def compute\_score(col):

return col \* 1.1

**9.3 Profiling**

* Use %time or time.time() before/after transformations.
* Compare stage duration via Spark UI.