Seekho Bigdata
Data is the New Oil

50+ PySpark Interview Questions — With Answers

Karthik Kondpak
9989454737

50+ PySpark Interview Questions — With Answers

✅ 1. What is PySpark? Explain in detail.

Definition

PySpark is the **Python API for Apache Spark**, which allows developers to write Spark applications using Python instead of Scala or Java.

PySpark interacts with the underlying Spark engine through:

- **Py4J (Python → JVM communication bridge)**
- **Apache Arrow (for optimized columnar data transfer, especially in Pandas UDF)**

Why PySpark?

- Easy to code for Python developers
- Integrates well with ML libraries (Pandas, NumPy, MLlib)
- Supports distributed processing on huge datasets

- Optimized using Catalyst + Tungsten

Internal Flow

Python Code → Py4J → JVM → Spark Execution Engine → Executors

Real-Time Use Case

Processing 2TB clickstream logs from Flipkart across 300 executors in parallel.

✅ 2. What is SparkSession? Why is it important?

Definition

SparkSession is the **entry point** to all Spark functionalities.

Before Spark 2.0 → We had:

- SQLContext
- HiveContext
- SparkContext

After Spark 2.0 → All merged into **SparkSession**.

What does SparkSession manage?

- SQL Query Engine
- DataFrame creation
- Catalyst Optimizer
- Configuration
- Catalog (database/tables metadata)
- Communication with cluster manager

Example

```
spark = SparkSession.builder \
        .appName("MyApp") \
        .config("spark.sql.shuffle.partitions", 100)
\
        .getOrCreate()
```

Why is it important?

Without SparkSession, no DataFrame operations can run.

✅ 3. Explain Spark Architecture in detail.

Components

1. **Driver Program**
    a. Creates SparkSession
    b. Converts code into DAG
    c. Schedules tasks
2. **Cluster Manager**
    a. YARN / Kubernetes / Standalone
3. **Executors**
    a. JVM processes running tasks
    b. Store data in memory
    c. Send results back to driver

Flow

Python Code → Driver → DAG Scheduler → Task Scheduler →

Executors → Results → Driver

Example

If dataset = 1TB, and 100 executors are available →

Each executor processes **10GB** in parallel.

✅ 4. Explain DAG (Directed Acyclic Graph) in Spark.

Definition

DAG = Representation of your transformations as a flow of operations with no cycles.

Example DAG:

```
Read → Filter → Select → GroupBy → Write
```

Stages

A wide transformation causes a **new stage**:

Stage 1 → narrow ops

Stage 2 → shuffle ops

Stage 3 → write ops

Why important?

- Allows optimization

- Parallel execution
- Fault tolerance (can recompute lineage)

✅ 5. What are Transformations and Actions? Give deep explanation.

Transformations (Lazy)

- Do NOT execute immediately
- Build logical plan
- Are **lazy evaluated**

Examples:

- map, filter, select
- join
- groupBy
- withColumn

Actions (Trigger execution)

Examples:

- show

- count
- collect
- write

Why laziness?

- Spark optimizes full DAG end-to-end
- Reduces computation
- Avoids unnecessary processing

✅ 6. Explain Narrow vs Wide Transformations.

Narrow (Fast)

- Parent partition → single child partition
- No shuffle
- Executed within same stage

Examples:

- map
- filter

- withColumn
- select

Wide (Slow)

- Requires shuffle
- New stage is created

Examples:

- groupBy
- join
- distinct
- orderBy

Why wide transformations are expensive?

- Data movement between nodes
- Write to disk
- Network IO
- Serialization/deserialization

✅ 7. What is Catalyst Optimizer? Explain in depth.

Catalyst Optimizer is Spark's **rule-based + cost-based SQL optimizer**.

Optimization Phases

1. **Analysis**
   a. Resolve column names
   b. Type checking
2. **Logical Optimization**
   a. Filter pushdown
   b. Constant folding
   c. Predicate reordering
   d. Null propagation
   e. Projection pruning
3. **Physical Planning**
   a. Choose best join strategy
   b. Sort-merge join vs hash join
4. **Code Generation**
   a. Whole stage code generation

Why Catalyst is powerful?

- Minimizes CPU operations
- Minimizes IO
- Uses statistics (cost-based optimizer)

✅ 8. What is Tungsten Engine? Explain technically.

Tungsten introduces **low-level optimizations** for speed:

Main Enhancements

1. **Off-heap memory**
2. **Binary memory format**
3. **Whole-Stage Code Generation**
4. **Eliminate JVM object overhead**
5. **Columnar caching**

Result

- Faster joins
- Faster aggregations

- Less GC overhead

✅ 9. Explain Whole-Stage Code Generation with example.

Spark combines multiple execution operators into a **single optimized function**.

Without WSCG:

```
Row → Filter → Map → Reduce (multiple function calls)
```

With WSCG:

```
Single Java code block
```

Benefit

- 3–5x faster execution
- Fewer CPU cycles
- Better pipeline optimization

✅ 10. Explain Broadcast Join in detail.

Definition

Broadcast join sends a small table to **every executor node**.

When to use?

- Table size < 100–500 MB
- Dimension/LUT tables
- When avoiding shuffle is key

Example

```
from pyspark.sql.functions import broadcast
df1.join(broadcast(df2), "id")
```

Internal Working

- Table df2 is serialized
- Copied to each executor
- Join becomes local operation → no shuffle

✅ 11. What is Data Skew? How to detect and solve?

Definition

Uneven distribution of keys during join/groupBy.

Example:

```
customer_id = 99999 → 50 million records
customer_id = others → only 10,000 records
```

Executor handling 99999 becomes slow → job fails.

Detection

1. Spark UI → skewed partition sizes
2. High task duration in one executor

Fix

1. Salting

```
df = df.withColumn("salt", (rand()*10).cast("int"))
```

2. Broadcast smaller table

3. Skew join hints

```
df1.join(df2.hint("skew"), "id")
```

4. Increase partitions
5. Range partitioning

✅ 12. What is Partitioning in Spark? Explain deeply.

Definition

The way Spark splits data into parallel chunks.

Why important?

- Determines parallelism
- Impacts shuffle
- Affects memory distribution
- Impacts file writing

Types

1. Hash partitioning

2. **Range partitioning**
3. **Custom partitioning**

Example

```
df.repartition(200, "cust_id")
```

✅ 13. Difference between repartition() and coalesce().

Feature	repartition()	coalesce()
Shuffle	Yes	No
Use Case	Increase or decrease partitions	Decrease only
Performance	Expensive	Cheap
Guarantees	Even distribution	Uneven distribution possible

✅ 14. Explain Window Functions in PySpark with examples.

Used for:

- Ranking

- Moving averages
- Running totals
- LAG/LEAD
- Time-based aggregations

Example

```
from pyspark.sql.window import Window
from pyspark.sql.functions import lag

w = Window.partitionBy("dept").orderBy("salary")

df.withColumn("prev_salary", lag("salary",
1).over(w)).show()
```

Real-time Use Case

Finding last 3 transactions per customer.

✅ 15. Explain explode(), posexplode(), explode_outer().

explode

Flatten array/map column → emits one row per element.

posexplode

Emits **position + element**.

explode_outer

Same as explode but preserves null entries.

Example

```
df.withColumn("item", explode("items"))
```

Perfect — I will now continue with **Q16–Q50**, but with **EVEN MORE DETAIL**, suitable for **creating full PySpark notes / eBook / interview guide**.

These answers will include:

✓Internal architecture

✓Real-time data engineering scenarios

Best practices

✓ PySpark code

✓ Hidden concepts interviewers expect

✓ Spark UI + performance notes

✓

✅ 16. How do you read Nested JSON in PySpark? Explain in depth.

⭐ Why nested JSON is difficult?

- JSON contains complex structures (arrays, structs, maps, nested structs).
- Schema is hierarchical.
- Flattening JSON creates wide tables with many nested levels.

⭐ How Spark handles JSON

Spark's JSON reader:

- Infers schema (if enabled)
- Parses hierarchical fields into `StructType`, `ArrayType`, `MapType`
- Allows exploding & flattening

⭐ Example JSON

```
{
  "id": 101,
  "name":"Shivani",
  "orders":[
      {"order_id": 1, "amount": 2000},
      {"order_id": 2, "amount": 3500}
  ],
  "address": {
      "city": "Mumbai",
      "pin": 400001
  }
}
```

⭐ Read JSON

```
df = spark.read.json("/path/nested.json",
multiLine=True)
df.printSchema()
```

⭐ Flatten JSON

```
frompyspark.sql.functions import col, explode

df2=df.withColumn("order", explode("orders")) \
        .select(
            "id",
            "name",
            col("address.city").alias("city"),
            col("address.pin").alias("pincode"),
            col("order.order_id"),
            col("order.amount")
        )
```

⭐ Use Case (e-commerce)

Flipkart API logs, Shopify webhooks, Razorpay payment events.

⭐ **Optimization Tips**

- Disable inferSchema and provide schema manually
- Always use multiLine=True for multi-row JSON objects

✅ 17. What are UDFs? Why are they slow? Detailed explanation.

⭐ **What is a UDF?**

A User Defined Function lets you write Python functions and apply them to Spark DataFrames.

```
from pyspark.sql.functions import udf

def multiplier(x): return x * 2
udf_mult = udf(multiplier)
df.withColumn("result", udf_mult("value"))
```

⭐ **Why are Python UDFs slow?**

Reason	Explanation
Breaks Catalyst Optimization	Spark cannot optimize custom Python logic
Row-by-row execution	UDF processes each row individually

Serialization cost	Python → JVM → Python
Cannot use vectorization	No batch operations
Not memory-efficient	Creates many Python objects

⭐ **Better Alternatives**

- **Built-in functions**
- **SQL expressions**
- **Pandas UDFs**
- **Scala UDF for speed-critical tasks**

⭐ **Real-Time Example**

Banking company applying regex-based fraud detection logic on 600M records — Python UDF was 20x slower, replaced with **Spark SQL built-ins**.

✅ 18. What are Pandas UDFs? Why are they faster?

⭐ **Definition**

Pandas UDFs (also known as vectorized UDFs) use **Apache Arrow** for **columnar, batch-based** execution.

⭐ Why faster?

Feature	Python UDF	Pandas UDF
Execution	Row-by-row	Batch-vectorized
Data transfer	Python ↔ JVM (slow)	Arrow-optimized columnar
Speed	Slow	5x–20x faster
Optimized by Catalyst?	No	Yes

⭐ Example

```
import pandas as pd
from pyspark.sql.functions import pandas_udf

@pandas_udf("double")
def multiply(col: pd.Series) -> pd.Series:
    return col * 10

df.withColumn("new_col", multiply("value"))
```

⭐ When to use?

- ML feature engineering
- Complex math operations
- Statistical calculations

✅ 19. What is cache() vs persist()? Deep explanation.

⭐ cache()

Default storage = MEMORY_ONLY

```
df.cache()
```

⭐ persist()

Allows custom storage level:

```
from pyspark import StorageLevel
df.persist(StorageLevel.MEMORY_AND_DISK)
```

⭐ Storage options

- MEMORY_ONLY
- MEMORY_AND_DISK
- MEMORY_ONLY_SER
- MEMORY_AND_DISK_SER
- DISK_ONLY

⭐ **When to use caching?**

✔Data reused multiple times

✔Iterative algorithms

✔Avoid recomputation of expensive transformations

⭐ **When NOT to use caching?**

✗ One-time use DataFrames

✗ Cache too many large DataFrames → executor OOM

⭐ **Real example**

Reusing customer dataset 8 times during product recommendation creation → caching reduces execution from **50 min → 7 min**.

✅ 20. Explain the top 10 PySpark Optimization Techniques (very detailed).

⭐ **1. Avoid Wide Transformations**

groupBy, join, distinct → cause shuffle

Shuffle = disk IO + network IO → slowest operation in Spark.

⭐ 2. Use Broadcast Join

```
df1.join(broadcast(df2), "id")
```

Avoids shuffle.

⭐ 3. Reduce Shuffle Partitions

Default partitions = 200 / 300 (too high)

Set:

```
spark.conf.set("spark.sql.shuffle.partitions", 50)
```

⭐ 4. Use Filter Early (Predicate Pushdown)

Filter before join/groupBy:

```
df.filter("year = 2024")
```

⭐ 5. Avoid Python UDFs

Use built-in or Pandas UDFs.

⭐ 6. File Format Choice

Use:

✔ Parquet

✔ ORC

✔ Delta

Avoid CSV/JSON during processing.

⭐ 7. Avoid Collect()

collect() pulls entire dataset to driver → OOM.

⭐ 8. Repartition Based on Join Key

```
df1.repartition("id").join(df2.repartition("id"))
```

⭐ 9. Use checkpoint() to break lineage

Very long DAG may cause stack overflow:

```
df.checkpoint()
```

⭐ 10. Use AQE (Adaptive Query Execution)

Automatically adjusts:

- join strategy
- shuffle partitions
- skew optimization

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

✅ 21. Explain Predicate Pushdown in Spark.

⭐ What is Predicate Pushdown?

Spark pushes filters to the file reader **before** reading full data.

Example:

```
SELECT * FROM sales WHERE year = 2024
```

Spark reads **only partitions where year=2024**, not entire dataset.

⭐ Works with

- Parquet
- ORC
- Delta
- JDBC (limited)

⭐ Does NOT work with

- CSV
- JSON

⭐ **Code Example**

```
df.filter("year = 2023")
```

⭐ **Benefit**

- Reduces IO
- Reduces scan time
- Faster joins/aggregations

✅ 22. Explain Column Pruning.

Spark reads only required columns.

⭐ **Example**

```
df.select("id")
```

If file has 100 columns, Spark reads **1 column** only.

⭐ **Benefit**

- Faster scan
- Less memory usage

- Small memory footprint

✅ 23. What is Delta Lake? Explain in detail.

⭐ Delta Lake = Parquet + Transaction Log (.delta)

⭐ Features

Feature	Description
ACID transactions	Safe writes in big data
Time travel	Query old versions
Schema evolution	Auto add columns
Schema enforcement	Prevents corrupted writes
Upsert/Merge	SQL MERGE support
Faster reads	File pruning + metadata optimization

⭐ Example

```
df.write.format("delta").save("/path")
df = spark.read.format("delta").load("/path")
```

⭐ Real-Time Use Case

Data Lakehouse architecture on Databricks / AWS EMR.

✅ 24. Explain Delta Merge (UPSERT) with real-time example.

⭐ Scenario

Updating daily incremental data to customer master table.

⭐ Code

```
from delta.tables import DeltaTable

dt = DeltaTable.forPath(spark, "/delta/customer")

(
 dt.alias("t")
 .merge(
     updates.alias("u"),
     "t.id = u.id"
 )
 .whenMatchedUpdateAll()
 .whenNotMatchedInsertAll()
 .execute()
```

)

⭐ **Optimizations**

- Z-Ordering
- Vacuum
- OPTIMIZE command

✅ 25. What is Schema Evolution vs Schema Enforcement?

⭐ **Schema Evolution (Allow changes)**

Spark automatically updates schema.

```
spark.c onf.se t("sp ark.d atabri cks.del ta.sch ema.a utoMe r
ge.enabled", True)
```

⭐ **Schema Enforcement (Reject mismatches)**

Spark blocks bad schemas.

Example error:

```
column price expected INT got STRING
```

⭐ **Use Cases**

- Changing JSON structures
- Adding new fields in Kafka streams
- Modifying CDC tables

✅ 26. How to optimize join performance? (Very detailed)

⭐ **Best Practices**

Broadcast small table

Repartition both tables on join key

```
df1.repartition("id").join(df2.repartition("id"),
"id")
```

Use Sort-Merge Join only when sorted

Avoid Skewed Keys

Use Bucketing for large static tables

Use AQE → auto tune joins

✅ 27. What is File Pruning?

Spark reads only **required partitions** based on partition column.

Directory structure:

```
/year=2024/month=07/day=01
```

Query:

```
WHERE year = 2024 AND month = 7
```

Spark does NOT read all folders → only required ones.

✅ 28. What are Accumulators?

⭐ Definition

Write-only variables used in:

- Counting
- Debugging
- Monitoring job metrics

⭐ **Example**

```
acc =
spark.sparkContext.longAccumulator("error_count")
```

⭐ **Limitation**

Executors can only increment, cannot read.

✅ 29. What are Broadcast Variables?

⭐ **Definition**

Read-only variable distributed to executors.

Used for:

- Lookup tables
- Config constants

⭐ **Example**

```
bv = spark.sparkContext.broadcast({"IN": "India",
"US": "United States"})
```

✅ 30. How to handle large datasets efficiently in PySpark?

✔ Use Parquet/ORC/Delta

✔ Avoid collect()

Write partitioned data

✔ Cache wisely

✔ Avoid Python UDFs

✔ Push filters early

✔ Use correct cluster config

✔ Use coalesce while writing small files

✔

✔Avoid small file problem

✔Use efficient serialization (Kryo)

Great — continuing with **Q31–Q50**, in the same ultra-detailed, note-style format ready for conversion into notes/ebook/carousel slides. Each question includes: concept, internals, code snippets, real-world use cases, performance tips, and interview-ready talking points.

✅ 31. What is Structured Streaming? Explain internals, modes, and code.

Concept

Structured Streaming is Spark's high-level API for stream processing built on top of the DataFrame/Dataset APIs. It treats streaming data as an **unbounded table** and provides exactly-once semantics for many sources/sinks.

Modes

- **Micro-batch** (default): processes data in small batches (e.g., every 1s).
- **Continuous** (experimental): low-latency processing (ms), limited operators.

Internals

- Stream is represented as a logical plan; continuous mini DAGs executed periodically.
- Checkpointing stores progress & state.
- Watermarking handles late data and state eviction.
- State store (RocksDB/Memory) manages stateful operators.

Example (micro-batch)

```
df = spark.readStream.format("kafka") \
      .option("kafka.bootstrap.servers","host:9092")
\
      .option("subscribe","topic").load()

parsed = df.selectExpr("CAST(value AS STRING) as
json") \
           .select(from_json(col("json"),
```

```
schema).alias("data")) \
           .select("data.*")

agg = parsed.groupBy(window("event_time","10
minutes"), "country").count()

query = agg.writeStream \
           .outputMode("append") \
           .format("parquet") \
           .option("path", "/mnt/stream_output") \
           .option("checkpointLocation",
"/mnt/checkpoints/agg") \
           .start()
```

Use cases

Real-time analytics, fraud detection, live dashboards, CDC ingestion.

Performance tips

- Tune trigger interval and processing time.
- Use watermarking to drop old state.
- Keep state small; use mapGroupsWithState cautiously.

- Persist state store to durable storage.

✅ 32. What is Watermarking and why use it?

Concept

Watermarking tells Spark how long to wait for late data for event-time aggregations. It bounds state size and allows emitting results while tolerating late arrivals up to the watermark threshold.

How it works

`withWatermark(eventTimeCol, "delay")` marks the maximum lateness; Spark drops state older than maxEventTime-delay .

Example

```
agg = events.withWatermark("event_time", "10 minutes") \
            .groupBy(window("event_time","5 minutes"), "user_id") \
            .count()
```

Use cases

Session windows, rolling aggregates, page-view counts with late logs.

Pitfalls

- If watermark too small → drop valid late events.
- Too large → state grows and memory pressure rises.

✅ 33. What is checkpointing (streaming & batch) and why is it needed?

Purpose

- **Streaming:** Save progress (offsets, state, metadata) to recover from failures and provide exactly-once guarantees.
- **Batch:** Shorten lineage (checkpointing RDDs/DataFrames) to avoid very long DAGs and stack overflow, or to make stable points for iterative algorithms.

Example (streaming)

```
query = df.writeStream \
          .format("parquet") \
          .option("checkpointLocation",
 "/mnt/checkpoint/stream1") \
          .start()
```

Example (batch)

```
spark.s parkCo ntext .setC heckpo intDir( "/tmp/ check point s
")
df.checkpoint()
```

Tips

- Always provide checkpointLocation for production streams.
- Use durable storage (S3/ADLS/DBFS) for checkpoint directories.
- Periodically clean up old checkpoints carefully when upgrading logic.

✅ 34. Explain Adaptive Query Execution (AQE) and how it helps.

Concept

AQE allows Spark to modify the physical execution plan at runtime using statistics collected during execution. It can:

- Dynamically switch join strategies.
- Coalesce shuffle partitions based on actual partition sizes.
- Handle skewed data by splitting/handling large partitions.

How to enable

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

Benefits

- Better resource utilization.
- Avoids over/under partitioning.
- Automatic skew mitigation.

Caveats

- Slight overhead for statistics collection.

- Works best with Parquet/ORC/Delta with file size stats.

✅ 35. How to handle data skew — techniques with code & pros/cons

Detection

- Spark UI: one or few tasks take much longer.
- Highly imbalanced partition sizes.

Techniques

1. **Salting**
    a. Add random suffix to skewed keys then join/group, then aggregate/sum back.

```
from pyspark.sql.functions import rand, concat, lit
df_salted = df.withColumn("salt",
(rand()*10).cast("int"))
df_salted = df_salted.withColumn("salted_key",
concat(col("key"), lit("_"), col("salt")))
```

    b. Pros: simple. Cons: increases data size & complexity.

2. **Broadcast the smaller side**
    a. If one table is small.

```
df_large.join(broadcast(df_small), "key")
```

    b. Pros: avoids shuffle. Cons: limited by broadcast size.
3. **Skew join hint / AQE**
    a. Let Spark auto-handle when enabled or use hints.
    b. Pros: automated. Cons: may not always pick perfect strategy.
4. **Range partitioning**
    a. Pre-partition data to distribute heavy keys.
5. **Aggregate then join**
    a. Pre-aggregate big group values to reduce volume.

✅ 36. How to compute quantiles and median reliably at scale?

Methods

1. **approxQuantile** (fast, approximate)

```
quantiles = df.approxQuantile("salary", [0.25, 0.5,
0.75], 0.01)
```

2. **percentile_approx** (SQL function)

```
from pyspark.sql.functions import expr
df.selectExpr("percentile_approx(salary, 0.5) as
median").show()
```

3. **Exact median** (costly)

- Sort and compute middle row(s) — requires full shuffle and is expensive; typically not used on massive datasets.

Tradeoffs

- Use approximate for speed (bounded error).
- Use exact only when dataset is small or exactness is mandatory.

✅ 37. Explain Bucketing: why, when and how (with code).

Concept

Bucketing writes data into a fixed number of files (buckets) based on a hash of one or more columns. It improves join performance when both tables are bucketed on the same key and number of buckets match.

How to write bucketed table

```
df.write.bucketBy(50,
"id").s ortBy( "id") .mode ("over write") .saveA sTabl e("bu c
keted_table")
```

Benefits

- Avoids shuffle on joins if both sides bucketed identically.
- Faster joins for repeatable static datasets.

Limitations

- Works best for static datasets (not for streaming).

- Need same number of buckets & same bucketing column for both tables.
- More complex to manage with schema evolution.

✅ 38. Explain Spark UI — key tabs and what to look for when tuning.

Key UI areas

- **Jobs tab**: high-level job breakdown.
- **Stages tab**: identifies wide vs narrow stages; shows shuffle read/write.
- **Executors tab**: per-executor memory, GC times, task times.
- **SQL tab** (if enabled): physical plans, executed queries.
- **Storage tab**: cached RDD/DataFrame partitions.

What to inspect

- Long-running tasks → skew.
- Large shuffle read/writes → reduce shuffles or optimize joins.
- Long GC times → tune executor memory or use MEMORY_AND_DISK_SER.

- Skewed partition sizes → repartition or salt.
- Number of tasks vs cores → parallelism mismatch.

Interview tip

Walk interviewer through an example: identify a slow job → check stages with the largest shuffle → inspect top tasks → hypothesize fixes (broadcast, repartition, reduce shuffle partitions).

✅ 39. What is Kryo serialization and why use it?

Concept

Kryo is a faster, compact binary serializer alternative to Java serialization. It reduces serialization/deserialization overhead and network IO.

How to enable

```
spark.conf.set("spark.serializer",
"org.apache.spark.serializer.KryoSerializer")
spark.conf.set("spark.kryo.registrationRequired",
```

```
"false" ) #optional
```

When to use

- When you serialize large custom objects (case classes/pickles).
- For performance-sensitive workloads across the network.

Caveats

- You may register custom classes for maximum speed and smaller serialized size.
- Debugging serialized objects is harder than Java serialization.

✅ 40. How to design idempotent pipelines and why it matters?

Why idempotency

- Reprocessing shouldn't duplicate or corrupt downstream state.
- Essential for retries, failure recovery, and exactly-once semantics.

Techniques

- Use **UPSERT (MERGE)** into Delta tables keyed by unique id.
- Use **transactional sinks** (Delta, ACID stores).
- Use **deduplication** via unique keys + latest timestamp.
- Use **atomic renames** when writing output files: write to temp dir → atomically move.

Example (Delta upsert)

```
deltaTable.alias("t").merge(
    updates.alias("u"),
    "t.id = u.id"
).whenM atched Updat eAll( ).when NotMatc hedIns ertAl l().e x
ecute()
```

✅ 41. How to build ML feature pipelines in PySpark (high-level)?

Components

- **Data ingestion** → reading structured + unstructured sources.

- **Cleaning & Imputation** → fillna, outlier handling.
- **Feature extraction** → StringIndexer, OneHotEncoder, Tokenizer, TF-IDF.
- **Feature scaling** → StandardScaler, MinMaxScaler.
- **Pipelines API** → combine transformers & estimators.

Example

```
from pyspark.ml import Pipeline
from pyspark.ml.feature import StringIndexer,
OneHotEncoder, VectorAssembler, StandardScaler

si = StringIndexer(inputCol="cat",
outputCol="cat_idx")
ohe = OneHotEncoder(inputCols=["cat_idx"],
outputCols=["cat_vec"])
va =
VectorAssembler(inputCols=["num1","num2","cat_vec"],
outputCol="features")
scaler = StandardScaler(inputCol="features",
outputCol="scaled_features")
pipeline = Pipeline(stages=[si, ohe, va, scaler])
model = pipeline.fit(df)
```

```
df_transformed = model.transform(df)
```

Tips

- Use VectorAssembler to prepare MLlib features.
- Persist intermediate large datasets only when reused.
- Use MLflow or Model Registry for artifacts.

✅ 42. How to do SCD Type 2 (Slowly Changing Dimensions) in PySpark?

Concept

Maintain historical records with `start_date`, `end_date`, and is_current flags when dimension attributes change.

Steps (high-level)

1. Read existing dimension Delta table.
2. Read incoming updates.
3. Identify rows to expire (existing rows where keys match but attributes differ).
4. Update existing rows end_date and is_current=false.

5. Insert new rows with `start_date=now, is_current=true.`

Example (Delta merge pattern)

```
from delta.tables import DeltaTable
from pyspark.sql.functions import current_timestamp

updates = new_data.withColumn("start_date",
current_timestamp()).withColumn("is_current",
lit(True))
dt = DeltaTable.forPath(spark, "/delta/dim_customer")

dt.alias("t").merge(
    updates.alias("u"),
    "t.customer_id = u.customer_id AND t.is_current =
true"
).whenMatchedUpdate(
    condition = "t.attr1 != u.attr1 OR t.attr2 !=
u.attr2",
    set = {"end_date": "current_timestamp()",
"is_current": "false"}
).whenNotMatchedInsertAll().execute()
```

Tips

- Maintain audit columns (etl_ts, source, batch_id).
- Compact table periodically to remove old files and optimize queries.

✅ 43. How to avoid the small files problem when writing Parquet/Delta?

Problem

Many small files cause high metadata overhead and slow reads.

Solutions

- **Write with coalesce/repartition**: reduce number of output files.

```
df.repartition(10).write.parquet("/path")
```

- **Use larger partition sizes**: avoid partitioning at too granular a level.
- **Use maxRecordsPerFile** when writing Parquet.

```
df.write.option("maxRecordsPerFile",
1000000).parquet("/path")
```

- **Compaction (OPTIMIZE)**: Delta OPTIMIZE or Spark job to compact small files.
- **Streaming: use batch windows to write fewer larger files.**

✅ 44. How to perform schema drift handling in streaming pipelines?

Issues

Source schema can change (new fields, type changes) causing downstream failures.

Strategies

- **Use schema-on-read**: read JSON as string and parse with from_json using evolving schema.
- **Schema registry**: enforce Avro/Schema registry for Kafka producers/consumers.
- **Schema evolution with Delta**: allow schema merge:

```
spark.c onf.se t("sp ark.d atabri cks.del ta.sch ema.a utoMe r
ge.enabled", "true")
```

- **Versioned schema handling**: parse versions inside payload and transform accordingly.

✅ 45. Explain how to test PySpark jobs locally and in CI.

Local testing

- Use `spark-submit` with local master: `--master local[*]`.
- Use small sample datasets with representative edge cases.

Unit testing

- Use `pytest` + `pyspark.sql.SparkSession` fixture.
- Test transformations functionally: assert schemas, row counts, sample values.

CI best practices

- Use containerized Spark (Docker) or lightweight Spark images.

- Run tests on representative sample, not full dataset.
- Mock external dependencies (S3, Kafka) or use in-memory/local test doubles.

Example pytest fixture

```
import pytest
from pyspark.sql import SparkSession

@pytest.fixture(scope="session")
def spark():
    spark =
SparkSe ssion. build er.ma ster(" local[2 ]").ap pName ("tes t
").getOrCreate()
    yield spark
    spark.stop()
```

✅ 46. How to monitor and alert Spark jobs in production?

Monitoring tools

- Spark UI (live)
- Ganglia / Prometheus / Grafana for metrics
- Cloud provider logs (Databricks, EMR, GCP Dataproc dashboards)
- Structured logging into ELK stack for driver/executor logs

Key metrics to alert on

- Job duration > SLA
- Task failure rate > threshold
- GC time high or executor OOM
- Queue length in cluster manager
- Lag in streaming (processing time >> trigger interval)

Practices

- Emit custom metrics (via Dropwizard/Prometheus) from executors.
- Centralize logs and set alerts on patterns (Exceptions, OOM).

- Use lineage & run metadata for observability (Airflow metadata, run IDs).

✅ 47. How to securely handle secrets, credentials in Spark jobs?

Do NOT hardcode secrets in code. Use:

- **Cluster secret stores** (Databricks Secrets, AWS Secrets Manager, Azure Key Vault).
- **Environment variables and instance roles** (IAM roles) for cloud storage access.
- **Encrypted config files** and read them at runtime from secure stores.
- **Short-lived tokens** for services.

Example (AWS)

- Use **IAM role for EC2/EMR** so no AWS keys in code.
- Or fetch credentials at runtime from AWS Secrets Manager securely.

✅ 48. Explain how Spark integrates with Hive / external metastore.

Integration points

- Spark can use Hive Metastore for table/catalog metadata.
- `spark.sql.catalogImplementation can be set to hive.`
- Use enableHiveSupport() in SparkSession to access Hive tables.

Example

```
spark =
SparkSe ssion. build er.en ableHi veSuppo rt().g etOrC reate (
)
spark.sql("select * from db.table").show()
```

Notes

- External metastore lets multiple engines (Presto, Hive, Spark) share metadata.
- Manage Hive-compatible partitioning and file formats (Parquet/ORC).

✅ 49. How to migrate PySpark jobs from on-prem to cloud (high-level checklist)?

Checklist

1. **Inventory**: list jobs, dependencies, data sources, connectors.
2. **Data Storage**: move to S3/ADLS/GCS or connect via VPC.
3. **Secrets & IAM**: adopt cloud-native auth (IAM roles, key vaults).
4. **Cluster sizing**: choose instance types and autoscaling policies.
5. **Storage format**: use Parquet/Delta; consider converting CSV/JSON.
6. **Networking**: set up VPC/Subnets, private endpoints for S3.
7. **CI/CD**: deploy with IaC (Terraform), containerize where needed.
8. **Monitoring**: integrate cloud logging & metrics.
9. **Performance testing**: run scale tests on target cloud resources.
10. **Cost optimization**: spot instances/preemptible, right-sizing.

✅ 50. Top 10 Spark configs every data engineer should know (and why)

1. `spark.sql.shuffle.partitions` — controls number of shuffle partitions (affects parallelism & small-files).
2. spark.executor.memory — memory per executor (avoid OOM or underutilization).
3. spark.executor.cores — cores per executor (controls task parallelism).
4. spark.driver.memory — driver memory for collect/joins on driver side.
5. spark.serializer — Kryo for faster serialization.
6. spark.sql.adaptive.enabled — enable AQE for runtime optimizations.
7. `spark.sql.autoBroadcastJoinThreshold` — broadcast join size threshold.
8. `spark.default.parallelism` — default number of partitions for RDD operations.
9. spark.speculation — speculative execution for slow tasks (defaults off, useful with skew).

10. `spark.l ocal.d` ~~ir~~ tmp dirs for shuffle and spill.

Seekho Bigdata
Data is the New Oil

Let's build your Data Engineering journey together!

Call us directly at: 9989454737

https://seekhobigdata.com/