Learning PySpark and Databricks: A Comprehensive Guide

1. Introduction to Apache Spark and PySpark

Apache Spark is a unified analytics engine for large-scale data processing and machine learning. It distributes data and computation across clusters for high performance and scalability. PySpark is the Python API for Spark, combining Spark's engine with Python's simplicity. Spark was developed to overcome Hadoop MapReduce's inefficiencies by keeping data in memory and reducing disk I/O, delivering 10–100× faster performance. Spark supports multiple languages (Scala, Java, Python, R) and APIs (RDDs, DataFrames, Spark SQL, MLlib, Structured Streaming) on a common execution engine.

- Spark **advantages** include easy parallelism, in-memory speed, and fault-tolerance via data lineage. It can handle batch and real-time (streaming) workloads.
- Spark **resiliency**: Its core data structure, the **Resilient Distributed Dataset (RDD)**, is an immutable, fault-tolerant collection split across cluster nodes. Spark tracks RDD lineage so lost partitions can be recomputed on failure.
- Spark **apis**: While RDDs offer low-level control (functional transformations), higher-level APIs like **DataFrames** (and the analogous Dataset in Scala/Java) provide schema, SQL querying, and optimizations. In Python, we generally use DataFrames and Spark SQL; Python lacks the statically-typed Dataset API.

2. Spark Programming Model: RDDs, DataFrames, Datasets

RDDs are the original Spark abstraction: an immutable, distributed collection of objects. They support **transformations** (map, filter, etc.) that produce new RDDs, and **actions** (count, collect, save) that trigger computation. Transformations are *lazy* – Spark builds a DAG (directed acyclic graph) of operations and only executes it when an action is called. This laziness allows Spark to optimize the execution plan (Catalyst optimizer) and avoid unnecessary work.

DataFrames are the recommended Spark API for structured data. A DataFrame is like a table: a distributed collection of rows with named columns and a schema. DataFrames support SQL-style queries and common data manipulations (select, filter, join, aggregate) in Python, Scala, Java or R. Internally, DataFrames are implemented on top of RDDs but provide many optimizations. Unlike RDDs, DataFrames know the schema and use the Catalyst optimizer to generate efficient bytecode (Tungsten engine). Because DataFrames are immutable, each transformation returns a new DataFrame; the original data is never altered.

Spark's **Datasets** (Scala/Java) are strongly-typed extensions of DataFrames. Python does not have a separate Dataset API; DataFrame itself covers Python use cases.

3. SparkSession and Getting Started with PySpark

All Spark functionality in PySpark is accessed via a **SparkSession**. The SparkSession (and the older SparkContext) manage connections to the cluster. To start a SparkSession, use the builder pattern: e.g.

```
from pyspark.sql import SparkSession
spark = SparkSession.builder \
    .appName("MyApp") \
    .master("local[*]") \
    .getOrCreate()
```

This creates (or retrieves) a singleton SparkSession. Once created, you use it to create DataFrames, execute SQL, or set configurations. SparkSession also provides Hive support (HiveQL, UDFs, Hive tables) without requiring a Hive installation. In Databricks, a SparkSession is created automatically for notebooks.

With a SparkSession, you can **create DataFrames** in several ways: from local collections (`spark.createDataFrame`), from files and tables (`spark.read` or SQL), or from existing RDDs. For example:

- **Local data**: `spark.createDataFrame([("Alice", 34), ("Bob", 45)], schema=["name","age"])`.
- **Files**: `df = spark.read.json("s3://.../data.json")`.
- **Tables**: `df = spark.table("database.table")` after a table is defined (Databricks Unity Catalog or metastore).

DataFrames have a **schema** (column names and types). Spark can often infer schema (e.g. from Parquet/JSON), or you can provide it manually using `StructType`. Rows in DataFrames are represented by `Row` objects and Columns by name/expressions. DataFrame API supports rich types (array, map, etc.).

4. Spark Transformations and Actions

Spark divides operations into **transformations** (which build a new RDD/DataFrame) and **actions** (which compute and return a result). For example, `df.filter(...)` or `rdd.map(...)` are transformations; `df.count()` or `df.write.save()` are actions that trigger the execution of the DAG. Because transformations are lazy, you can chain many transformations before the actual computation happens. Only when an action is called (like `show()`, `collect()`, `write`, or `count()`) does Spark execute the plan on the cluster.

Common transformations on DataFrames include:

- **Selection/Projection**: `.select()`, `.selectExpr()`, or column expressions.
- **Filtering**: `.filter()` or `.where()` with column conditions (using `col`, expression or SQL syntax).
- **Column Operations**: `.withColumn()` to add/transform columns, `.withColumnRenamed()`, `.drop()` to remove columns.
- **Joins**: `.join(other, on=..., how="inner")` for inner/left/right/outer joins.
- **Grouping and Aggregation**: `.groupBy(cols).agg(...)` using functions like `avg`, `sum`.
- **Union or Append**: `.union()` to append DataFrames with the same schema.
- **Sorting/Ordering**: `.sort()`, `.orderBy()` by columns.
- **Window functions**: (for advanced tasks, e.g. `over(Window.partitionBy(...).orderBy(...))`).

Examples from Databricks PySpark:

```
from pyspark.sql.functions import col, avg
df = spark.table("sales")
df_filtered = df.filter(col("amount") > 1000)              # filter rows
```

```
df_grouped = df.groupBy("region").agg(avg("amount"))      # aggregation
df_joined  = orders.join(customers, on="cust_id", how="left")  # join
```

Because of lazy evaluation, you can chain transformations for readability. For example:

```
df_result = (df.filter(col("status")=="F")
               .groupBy("order_priority")
               .agg(count("*").alias("n"))
               .sort(col("n").desc()))
df_result.show()
```

Here, Spark will build one execution plan for all steps.

Common **actions** include:

- `show()`, `display()`: Print top rows (Databricks uses `display()` in notebooks).
- `collect()`: Bring results to driver (only for small results).
- `count()`, `first()`, `take(n)`: Aggregations or small pickups.
- `write` or `saveAsTable()`: Write data to storage or a metastore.
- Any aggregation like `count`, `sum` triggers execution by itself.

For example, `.count()` is both an aggregation and an action. When you write or save a DataFrame, that is also an action. The documentation notes that in production pipelines, it's best to have only one action at the end (usually a write) to avoid unnecessary materialization.

5. Data Engineering with PySpark: ETL and Data Sources

PySpark excels at ETL (extract-transform-load) for big data. Typical data flows include: **Ingestion**, **Transformation**, and **Loading** into a sink (data warehouse, lake, analytics).

- **Data Sources**: Spark can read from many sources: HDFS, S3, Azure Blob, databases (via JDBC), Kafka streams, etc. Supported formats include Parquet, ORC, CSV, JSON, Avro, Delta Lake, etc. For example, `spark.read.parquet("/path/to/file")` or `spark.read.format("csv").option("header",True).load("s3://...")`.
- **Schema Management**: Some sources (like Parquet, Delta) carry schema; others (CSV/JSON) Spark can infer schema or use a provided one. A schema is a collection of fields (name, type, nullability).
- **ETL Workflows**: An ETL pipeline might read raw data into a Spark DataFrame, apply cleaning/transformations (filter, join, derive columns), and then write to a target (e.g. Databricks Delta table or SQL Warehouse). Databricks' **Delta Lake** is often used as the storage format, providing ACID transactions and efficient updates.
- **Batch vs Streaming**: Spark supports both batch (one-time processing) and real-time streaming in the same API (Structured Streaming). Databricks notes one unified API for both modes. You can write streaming queries with `.writeStream` to continuously process data from sources like Kafka or files.
- **Data Flow Patterns** (from Databricks glossary): Data flows have sources, transformations, sinks, and paths. ETL is a common pattern: ingest from multiple sources, transform in Spark (cleansing, aggregating), and load to a sink like a data warehouse or dashboard. Real-time pipelines might use Structured Streaming for continuous data, while batch jobs run on a schedule.

- **Notebooks/Workflows**: In Databricks, you typically build ETL pipelines in notebooks or Jobs (workflows) that chain together PySpark code and SQL, orchestrated by Jobs Scheduler.

6. Advanced PySpark Topics

6.1 Partitioning and Parallelism

Spark automatically partitions RDDs/DataFrames across the cluster, allowing parallel processing of data too large for one node. The number of partitions affects parallelism: more partitions enable more tasks concurrently. You can **repartition** or **coalesce** DataFrames to adjust partitions for performance.

- `repartition(n)`: Shuffles data into *n* partitions (can increase or decrease) by hash or specified columns; causes a full shuffle.
- `coalesce(n)`: Merges partitions without full shuffle (only decreases partitions), often more efficient for merging files. Care must be taken: coalescing to 1 partition causes all data on one node, which can be slow. Adjusting partitions helps balance loads and optimize joins or writes.

Partitioning also applies at storage: e.g. writing a DataFrame to Parquet/Delta with `partitionBy("date")` physically partitions files by that column, speeding up queries with filters on that column.

6.2 Caching and Persistence

Spark can **cache** or **persist** intermediate results in memory (or disk) to speed up repeated computation. For example, if you use the same DataFrame multiple times, call `df.cache()` or `df.persist()` to keep it in memory after the first action. The default persistence level is `MEMORY_AND_DISK`. This avoids recomputing expensive transformations. Note that caching is **lazy**: it happens only once an action triggers computation. Use caching for hot data reuse (joins, iterative algorithms).

6.3 Broadcast Variables and Accumulators

- **Broadcast variables**: Spark can broadcast a small read-only lookup dataset to all executors to avoid shipping it with every task. The driver calls `bc = spark.sparkContext.broadcast(myDict)`, and then tasks access `bc.value`. The broadcast data is distributed once to each node and kept in memory. This is efficient for joining a large DataFrame with a small static dataset (using broadcast joins). In essence, Spark "broadcasts the common data needed by tasks" and caches it on workers.
- **Accumulators**: (Not covered in detail here) are variables that workers can "add" to (like counters), and the driver can read aggregated results. Useful for debugging or global counters.

6.4 Fault Tolerance and Checkpointing

Spark recovers lost partitions using RDD lineage. For very long lineages, you can **checkpoint** to break the lineage (writing data to stable storage). (Typically used in iterative algorithms or streaming; see Spark docs for `rdd.checkpoint()`.)

6.5 Performance Tuning

To optimize Spark jobs:

- **Partitioning strategy**: choose good partition keys and count (e.g. use `repartition` on join keys) to avoid data skew and reduce shuffle overhead.

- **Filter early**: apply `.filter()` as early as possible to reduce data size.
- **Avoid user-defined Python loops**: use DataFrame API over Python `collect()`, `toPandas()`, or UDFs whenever possible, as native Spark transformations are faster.
- **Broadcast small tables**: in large joins, broadcasting the smaller DataFrame can avoid shuffle (use `broadcast()` hint).
- **Resource sizing**: configure executor cores and memory based on data size. Databricks' autoscaling and Photon runtime further optimize compute resources automatically.

Databricks recommends enabling *serverless compute* and *autoscaling* for ease: you don't manage nodes manually.

7. Spark Libraries: SQL, Streaming, MLlib, GraphX

Spark bundles powerful libraries:

- **Spark SQL & DataFrames**: Enables SQL queries on DataFrames (and Hive tables). You can mix `df.select(...)` with SQL via `spark.sql("SELECT * FROM table")`. Spark SQL uses the same engine as DataFrame API.
- **Structured Streaming**: A high-level API for stream processing. You express streaming computation with the same DataFrame/SQL syntax, and Spark manages continuous execution.
- **MLlib**: Scalable machine learning library with algorithms for classification, regression, clustering, recommendation, etc. It integrates with DataFrames for pipelines.
- **GraphX**: A graph processing API (mostly Scala/Java) for graph-parallel workloads.
- **Pandas API on Spark**: Allows using Pandas-like syntax on Spark DataFrames, making it easier to scale Pandas code.

These libraries let you build complex data science and machine learning pipelines in Spark.

8. Databricks Platform Overview

Databricks is a managed service for Spark (and now a full data lakehouse platform). Key components:

*Figure: Databricks architecture – the **Control Plane** (Databricks-managed; web interface, REST API, compute orchestration, Unity Catalog) is separate from the **Compute Plane** (customer-managed cluster or serverless compute resources).*

- **Control Plane vs Compute Plane**: Databricks separates the management plane (web UI, job scheduling, metadata, etc.) from the compute nodes. The control plane runs in Databricks' cloud account (managing your workspace), while compute (Spark clusters, SQL warehouses) run in your cloud account. This provides security isolation and allows Databricks to manage infrastructure automatically.
- **Workspace Storage (DBFS)**: Each Databricks workspace has a cloud storage bucket (DBFS) for files and tables. Users store notebooks, libraries, and data in DBFS or in Unity Catalog storage. Note: newer best practices encourage using Unity Catalog for data governance.
- **Clusters (Compute Plane)**: Databricks computes run on clusters. There are *interactive clusters* for notebooks and *jobs clusters* for scheduled jobs. Clusters can be managed (provisioned) or serverless. Databricks manages autoscaling, engine versions, etc.

- **Serverless Compute**: Databricks offers serverless SQL warehouses and jobs (the compute instances come up automatically). This eliminates infrastructure management and often speeds up startup.
- **Databricks Notebooks**: Interactive environment supporting multiple languages (Python, SQL, Scala). Notebooks connect to clusters or SQL warehouses, allowing queries, visualizations, and reporting. You write PySpark code in notebook cells.
- **SQL Warehouses (formerly SQL Endpoints)**: A specialized compute for SQL workloads. Useful for BI and ad-hoc queries. Serverless SQL warehouses auto-scale for concurrency. You can attach notebooks to SQL warehouses (with some limitations).
- **Data Lakehouse**: Databricks promotes the *lakehouse* paradigm: a single storage layer (Delta Lake) for both BI (SQL) and ML. Delta Lake provides ACID transactions on top of cloud storage, unifying reliability of data warehouses with the scale of data lakes. Unity Catalog adds fine-grained governance (table-level ACLs, data lineage).

Databricks Data Flow and ETL Pipelines

Databricks encourages a modular data flow: ingest data into the lake (often into Delta tables), transform with Spark/SQL, and serve analytics from the same data. Data flows may be batch or streaming. Databricks provides *Delta Live Tables (DLT)* to simplify declarative ETL pipelines (not covered in detail here).

As a Databricks user (coming from a Postgres/SQL background), you'll often:

- Use **notebooks** to author PySpark ETL and SQL queries.
- Store reference data or dimension tables in Unity Catalog.
- Run scheduled **Jobs** (workflows) for production pipelines.
- Possibly use **Databricks SQL** dashboards on top of your data.

Databricks documentation and examples are excellent resources for learning: see the PySpark on Databricks page and PySpark basics tutorial for hands-on guides.

9. Learning Resources and Plan

To master PySpark and Databricks in a week, follow a structured path:

- **Official Documentation**: The Apache Spark and Databricks docs are authoritative. Spark's **Getting Started** and **Programming Guide** cover fundamentals. Databricks docs have many tutorials (e.g. PySpark basics) .
- **Online Tutorials**: Platforms like SparkByExamples, TutorialsPoint, and DataCamp have tutorials (e.g. SparkByExamples has PySpark guides on RDDs, DataFrames, broadcasting). Tutorialspoint's PySpark tutorial can be helpful for syntax.
- **Videos/Courses**: DataCamp and Udemy offer courses on PySpark. Databricks Academy (some free materials) covers Spark fundamentals.
- **Books**: *Learning Spark* (O'Reilly) or *Spark: The Definitive Guide* can be referenced for deeper understanding (beyond one-week scope, though).
- **Practice**: Use Databricks Community Edition (free) or your company's Databricks environment to run PySpark notebooks. Try example datasets (e.g. databricks-datasets).

Suggested study plan (1 week intensive):

1. **Day 1-2**: Core Spark concepts – RDDs, DataFrames, SparkSession. Follow quickstarts and simple examples (create/read DataFrames, basic transforms).
2. **Day 3-4**: PySpark DataFrame API – Practice select, filter, groupBy, joins, and writes (CSV, Parquet, Delta). Explore Databricks notebooks interface. Learn lazy evaluation and caching (try `.cache()`).
3. **Day 5**: Advanced concepts – Partitioning (`repartition`, `coalesce`), broadcast variables, accumulators. Read about performance tuning (Databricks best practices).
4. **Day 6**: Spark SQL and Spark Streaming – write some `spark.sql()` queries on temp views. Try a simple Structured Streaming example (e.g. read CSV stream).
5. **Day 7**: Databricks platform specifics – study Databricks architecture (see Fig. 1), work through a Databricks ETL example, and understand SQL Warehouses. Review key PySpark interview Q&A to solidify knowledge.

Stay hands-on: running code and seeing results helps cement concepts. Use cluster logs and UI for debugging and optimization.

10. PySpark Interview Questions and Answers (200 Q&A)

Below is a comprehensive list of common PySpark interview questions and succinct answers, covering basics to advanced topics. (Answers include brief explanations; see references for details.)

1. **What is Apache Spark?** A fast, in-memory data processing engine for big data and machine learning. It distributes data and computation over a cluster, supporting parallel processing and fault tolerance via RDDs.

2. **What is PySpark?** The Python interface to Apache Spark. It lets you write Spark jobs in Python, leveraging Spark's parallel engine.

3. **Why use Spark over Hadoop MapReduce?** Spark keeps data in memory and minimizes disk I/O, giving 10–100× speedups over MapReduce. It supports both batch and streaming, whereas MapReduce is batch-only. Spark's DAG optimizer also improves query performance.

4. **What is an RDD?** Resilient Distributed Dataset: the basic Spark data structure. It's an immutable, partitioned collection of elements, resilient via lineage logging.

5. **How do you create an RDD?** From an existing collection using `spark.sparkContext.parallelize(...)`, or by loading data (textFile, etc.). You can also convert a DataFrame to RDD via `df.rdd`.

6. **What is a DataFrame?** A Dataset of Row with named columns (like a table). Conceptually similar to a pandas/R data frame or SQL table, but distributed. It's optimized (Catalyst) and supports high-level queries.

7. **What's the difference between RDDs and DataFrames?** RDDs: unstructured, low-level, fault-tolerant collections; no schema. DataFrames: high-level, structured with schema, more optimizations (Catalyst, off-heap memory). DataFrames are usually faster for SQL-like operations.

8. **What is SparkSession?** The entry point to Spark 2.x+. It replaces older SparkContext. It's created with `SparkSession.builder` and used to create DataFrames and run Spark SQL.

9. **How do you create a SparkSession in PySpark?**

```
from pyspark.sql import SparkSession
spark = SparkSession.builder \
     .appName("App") \
     .master("local[*]") \
     .getOrCreate()
```

This creates/configures the session.

10. **How to create a DataFrame?** Use `spark.read` on files (CSV, JSON, Parquet, etc.), or `spark.createDataFrame(data, schema)`, or `spark.table("db.table")`. E.g., `df = spark.read.csv("file.csv", header=True)` or `df = spark.table("samples.customer")`.

11. **What are transformations and actions in Spark?** Transformations (e.g., `map`, `filter`, `groupBy`) build a new RDD/DataFrame and are *lazy*. Actions (`count`, `show`, `write`) compute results and trigger execution.

12. **Explain lazy evaluation in Spark.** Spark delays execution of transformations until an action is called. It builds a logical plan (DAG) of all transformations and optimizes before running. This avoids unnecessary computations.

13. **Give an example of Spark transformations.** Selecting or filtering DataFrame columns:

```
df2 = df.select("col1", "col2")           # projection
df3 = df2.filter(df2.col1 > 100)         # filtering
```

These do not execute until an action (like `df3.show()`) is called. 14. **Give an example of Spark actions.** `df.count()`, `df.show()`, `df.collect()`, `df.write.save()`, `df.take(5)`. For instance, `df.count()` triggers a job that computes the number of rows. 15. **What is the Spark SQL engine?** Spark SQL is the module for structured data processing. It provides DataFrame API and ability to run SQL queries on data. It uses the same execution engine as the DataFrame API. 16. **How do you run SQL queries in PySpark?** Use `spark.sql("SELECT * FROM table")` after registering a temp view or accessing a table. Results come back as a DataFrame. Spark SQL queries go through the same optimizer as DataFrame operations. 17. **What is a schema in Spark?** The column names and data types of a DataFrame. Spark infers schemas for formats like Parquet/ORC/JSON, or you can specify one manually using `StructType`. Schemas enable Spark to optimize queries. 18. **What are some common DataFrame operations?** - `select`, `filter`, `withColumn`, `drop`, `alias` (column ops), - `groupBy`, `agg` (aggregations), - `join` (inner/left/right/outer), - `sort`, `orderBy`, `limit`. 19. **How do you rename or add columns?** Use `withColumnRenamed("old", "new")` or `withColumn("newcol", expr)`. Example: `df.withColumnRenamed("balance", "balance_usd")`. Columns are immutable, so it returns a new DataFrame. 20. **How do you remove duplicate rows?** Use `df.distinct()` to return only unique rows. You can also use `dropDuplicates()`. 21. **How do you handle null values?** Use `df.na.drop()` to drop rows with nulls, or `df.na.fill(value)` to fill nulls, or `df.fillna()`. For numeric imputations, use `Imputer` from `pyspark.ml`. 22. **How do you handle missing data (nulls) in PySpark?** You can drop or fill. For example: `df.dropna()` drops rows with any nulls, and `df.fillna(0)` replaces nulls with 0. PySpark's MLlib also has `Imputer` for median/mean imputation. 23. **What are transformations vs actions?** Transformations define a new RDD/DataFrame (e.g. `map`, `filter`); actions compute and return results (e.g. `count`, `show`, `write`). 24. **What is SparkContext?** (Legacy) The low-level entry point in Spark 1.x. In PySpark 2.x+, use SparkSession. `spark.sparkContext` gives the underlying SparkContext if needed (for low-level RDD

ops or broadcasting). 25. **What is a broadcast variable?** A read-only variable cached on each worker node. It is used to efficiently share a small lookup dataset across tasks. Spark "broadcasts" the data once to all executors and caches it there. Useful for joining a large DataFrame with a small one. 26. **How do broadcast variables work?** Use `bc = spark.sparkContext.broadcast(data)`. When tasks run, they access `bc.value`. Internally, Spark sends this data to each node only once (not with every task). 27. **What is an accumulator?** A write-only variable that workers can add to (e.g. counters). The driver can read the accumulators after job completion. Useful for counting or aggregating metrics across tasks. 28. **What is lazy evaluation?** Spark delays executing transformations until an action is called. This allows optimization: Spark builds a DAG of transformations and plans efficient execution. 29. **What is a DAG in Spark?** Directed Acyclic Graph of stages. It represents the sequence of transformations. Spark splits the plan into stages separated by shuffles. The DAG is used for scheduling tasks. 30. **Explain Spark's execution model (jobs, stages, tasks).** A Spark job (triggered by an action) is divided into stages based on shuffle boundaries. Each stage is a set of parallel tasks (one per partition) that can run concurrently. 31. **What are Spark transformations?** Operations that produce a new RDD/DataFrame without computing immediately. Examples: `map`, `filter`, `join`, `groupBy`. They are **lazy** and only define the computation. 32. **What are Spark actions?** Operations that trigger execution of transformations and return results to the driver or write data. Examples: `collect`, `count`, `take`, `show`, `saveAsTable`. 33. **What is caching in Spark?** Saving a DataFrame/RDD in memory (or disk) to speed up future uses. Use `df.cache()` or `df.persist()`. This avoids recomputing expensive operations. 34. **Difference between cache() and persist():** Both store data, but `cache()` defaults to memory. `persist()` can take a storage level (e.g. `Memory_and_Disk`). In PySpark, `cache()` is shorthand for `persist()` with default level. 35. **What is checkpointing?** Writing intermediate RDD to durable storage (e.g. HDFS/DBFS) to truncate lineage. Used in iterative jobs/streaming to avoid long lineage. (Often used with `rdd.checkpoint()`). 36. **How to optimize joins in Spark?** Use broadcast joins if one table is small (Spark will skip shuffle). Partition both sides on the join key. Also filter data early and select only needed columns. 37. **What is a broadcast join?** When you hint or Spark decides to broadcast a small dataset to all workers, avoiding shuffle. Achieved via `broadcast(df_small)`. 38. **Explain partitionBy when writing data.** `df.write.partitionBy("col")` writes files organized by unique values of "col". This speeds up queries with filters on that column (predicate pushdown). 39. **What are shuffle operations?** Operations like `repartition`, `groupByKey`, `join` that redistribute data across partitions. Shuffles incur network I/O and can be expensive. Spark sorts data during shuffle. 40. **How do you handle data skew?** Ensure partition key distributes evenly. If some keys have very large groups, consider salting keys, or using `map-side combine` (aggregate before shuffle) if possible. 41. **What is shuffle partition number, and how to change it?** Spark default shuffle partitions is 200. You can set `spark.sql.shuffle.partitions` or use `repartition(numPartitions)` to control parallelism. 42. **How to count rows in a DataFrame?** Use `df.count()`. This is an action that triggers a job and returns the number of rows. 43. **How to show data in Databricks notebook?** Use `display(df)` (Databricks helper) or `df.show()`. `display` offers GUI charts and SQL after the fact. 44. **Explain DataFrame immutability.** DataFrames are immutable: any transformation returns a new DataFrame and the original remains unchanged. This means all data transformations are functional and build new representations. 45. **How to add a column to a DataFrame?** Use `withColumn("newcol", expr)`. Example: `df2 = df.withColumn("isAdult", df.age > 18)`. This returns a new DataFrame with the added column. 46. **How to drop a column?** Use `df.drop("colname")` to return a new DataFrame without that column. You can drop multiple columns at once: `df.drop("a", "b")`. 47. **How to rename a column?** Use `df.withColumnRenamed("oldName", "newName")`. You can also use `alias`: e.g. `df.select(df.col("oldName").alias("newName"))`. 48. **How to change column datatype?** Use the `cast` method: `df2 = df.withColumn("age", df.age.cast("string"))`. Spark `cast` supports converting between types. 49. **How to select columns by wildcard or pattern?** You can use `select("*")` to get all

columns. With Spark SQL functions, you can use `col("pattern*")` or regex if needed. In SQL you can do `SELECT * EXCEPT(col)` in Spark 3.5+. 50. **What is `selectExpr` in PySpark?** A DataFrame method that takes SQL expressions as strings. For example, `df.selectExpr("col1 as c", "round(col2) as col2r")`. 51. **How do joins work in PySpark?** Use `df1.join(df2, on=condition, how="type")`. The `how` can be `inner` (default), `left`, `right`, `outer`. For example: `df1.join(df2, on=df1.id==df2.id, how="inner")`. 52. **Give an example of a left join.** `df1.join(df2, on=df1.id==df2.id, how="left")` – returns all rows from `df1`, matching from `df2`, null if no match. 53. **What is an outer join?** `how="outer"` returns all rows from both DataFrames, joining where keys match and filling nulls otherwise. 54. **How do you perform a multi-condition join?** Use combined expressions in `on`. For example: `python condition = (df1.id==df2.id) & (df1.date == df2.date) df_join = df1.join(df2, on=condition, how="inner")` 55. **What is DataFrameWriter?** The interface for saving DataFrames to storage. E.g. `df.write.format("parquet").save(path)` or `df.write.saveAsTable("db.table")`. Modes like `overwrite` or `append` control behavior. 56. **How to write DataFrame as table?** Use `df.write.saveAsTable("catalog.schema.table")` to save into Unity Catalog or metastore table. 57. **How to write DataFrame to CSV?** `df.write.format("csv").option("header",True).mode("overwrite").save(path)`. The `mode` can be `overwrite`, `append`, or `ignore`. 58. **What does `mode("overwrite")` do when writing?** It deletes existing data at the target location and writes new data. `append` adds to existing data; `ignore` does nothing if data exists. 59. **What is Spark SQL vs DataFrame API difference?** They are unified: SQL queries on DataFrames are converted to the same execution engine. You can use either; Spark builds the same DAG. SQL is useful for complex queries or when porting existing SQL. 60. **What is Spark Thrift Server / JDBC?** Allows external BI tools to query Spark SQL via JDBC/ODBC. Databricks has connectors for JDBC/ODBC to SQL warehouses.

61. **What are accumulators?** Write-only variables that workers can "add" values to. The driver can read accumulated results after job. Useful for counters or sums across tasks.

62. **How to filter with multiple conditions?** Combine conditions with bitwise operators: `df.filter((df.a>0) & (df.b=="X"))`. Use `&` for AND, `|` for OR, and parentheses for grouping.

63. **How to use SQL functions in DataFrame API?** Import functions: e.g. `from pyspark.sql.functions import col, expr, round`. Use them like `df.select(col("c1"), expr("c2*10"))`.

64. **What is the `expr` function?** It allows SQL syntax in expressions: `expr("c_custkey")`. Useful for complex calculations as strings.

65. **How to perform groupBy and aggregation?** Use `groupBy("col").agg(funct)`. Example:

```
from pyspark.sql.functions import avg
df_seg = df.groupBy("segment").agg(avg(df["balance"]))
```

This is similar to SQL `GROUP BY`. 66. **How do you count rows by group?** `python from pyspark.sql.functions import count df_counts = df.groupBy("category").agg(count("*").alias("n"))` Or `df.groupBy("category").count()` (shortcut) – returns a column named `count` of rows per category. 67. **Explain DataFrame immutability.** A DataFrame is a **logical plan** of transformations on a data source. No data is changed in place. Each transformation (select, filter, etc.) returns a new DataFrame representing a new plan. The old DataFrame remains usable. 68. **What is partitionBy when writing?** Partitioning the output data by a column. E.g.

`df.write.partitionBy("year","month").parquet(path)`. This creates subdirectories for each year/month, improving later read performance if you filter on those columns. 69. **What is a temp view?** A temporary table created from a DataFrame: `df.createOrReplaceTempView("tempView")`. Then you can run `spark.sql("SELECT * FROM tempView")`. The view lives only in the session. 70. **What is a global temp view?** A view available across sessions, created by `df.createGlobalTempView("name")`. It's tied to a system `global_temp` database.

71. **What is Spark's default parallelism?** By default, number of partitions is set by `spark.default.parallelism` (usually total cores) or `spark.sql.shuffle.partitions` (default 200). These can be tuned.

72. **What are broadcast joins and broadcast variables?** As above: broadcast a small table to avoid shuffle. Use `from pyspark.sql.functions import broadcast` to hint: `df.join(broadcast(df_small), ...)`. Internally uses SparkContext.broadcast().

73. **When to use RDDs over DataFrames?** Rarely in PySpark. RDDs if you need fine-grained control or work with unstructured data. DataFrames are almost always preferred for performance. RDDs are useful if you need complex functions not supported in DataFrame API.

74. **What is whole stage codegen?** (Advanced) Spark's optimizer can combine many operations into a single Java function for efficiency (Tungsten engine). This speeds up execution by reducing CPU overhead.

75. **What is Catalyst Optimizer?** Spark's query optimizer that generates efficient execution plans for DataFrame/SQL queries by analyzing predicates, generating code, etc.

76. **How does Spark achieve fault tolerance?** RDD lineage keeps track of how to recompute partitions. If a node fails, Spark re-runs only the missing partitions using the recorded transformations.

77. **Explain data locality in Spark.** Spark tries to schedule tasks on nodes where the data is already located (HDFS blocks). This reduces network traffic.

78. **What is lazy evaluation good for?** It lets Spark optimize across multiple transformations, pipelining operations and reducing disk IO. It also avoids running unnecessary steps.

79. **Explain Spark's DAG.** A Directed Acyclic Graph of stages representing a job. Spark builds a DAG from all transformations and breaks it into stages. Each stage's tasks run in parallel.

80. **What is RDD lineage?** The logical graph of transformations used to build an RDD. Spark retains lineage so it can recompute lost data if needed.

81. **What is a transformation vs an action?** (Restatement) Transformation: returns new RDD (lazy). Action: computes result or writes out (eager).

82. **How is Spark different from Flink?** Spark is micro-batch (or now also supports streaming), Flink is natively stream-first. Spark maintains lineage for fault tolerance. (This is more conceptual and might not need source).

83. **What is Spark Streaming?** (Legacy) Discretized Streams (DStreams) API for streaming on Spark 1.x. Now replaced by Structured Streaming which treats streams as tables that are continuously updated.

84. **What is Structured Streaming?** A high-level API where you define streaming computation via DataFrame/Dataset operations. Spark automatically handles streaming as continuous execution. You use `df.writeStream` to start.

85. **How does Spark Structured Streaming ensure exactly-once semantics?** Through checkpointing and write-ahead logs for sources/sinks. (Requires detailed knowledge; skip citation).

86. **What libraries come with Spark?** Spark SQL/DataFrames, Spark Streaming (Structured), MLlib, GraphX, SparkR (R), Sparkling Water, etc. (From [51†L153-L160]).

87. **What is MLlib?** Spark's machine learning library. Includes scalable implementations of common ML algorithms and utilities for feature extraction, etc.

88. **How do you convert DataFrame to Pandas?** Use `df.toPandas()`, which collects all data to the driver into a Pandas DataFrame (beware memory!). For very large data, use PyArrow-enabled conversion or Pandas API on Spark.

89. **What is Pandas API on Spark?** It allows using pandas-like syntax on Spark DataFrames. It runs on Spark while giving a familiar pandas interface.

90. **How to use UDFs in Spark?** You can define user-defined functions in Python and register them for use in DataFrames:

```
from pyspark.sql.functions import udf
def sq(x): return x*x
udf_sq = udf(sq)
df.select(udf_sq(col("value"))).show()
```

Note: UDFs serialize data and are slower than built-ins.

91. **What is a Window function?** Spark SQL window (analytics) functions. E.g. `over(Window.partitionBy(...).orderBy(...))` to compute running totals, ranks. Used with `withColumn` or `select`.

92. **How to handle skew in joins?** If a key has many records, consider broadcasting the smaller side or adding random "salt" to the join key to break it up, then remove salt after join.

93. **What's the use of `persist()` vs `unpersist()`?** `persist()` (or `cache()`) marks an RDD/DataFrame to be cached. `unpersist()` removes it from cache when no longer needed, freeing memory.

94. **Difference between `cache()` and `persist()`?** `cache()` is shorthand for `persist()` with default storage level (`MEMORY_AND_DISK_DESER` by default). You can use `persist()` with different storage levels (e.g., `MEMORY_ONLY`, `DISK_ONLY`).

95. **What is SparkContext.getOrCreate()?** Returns an existing SparkContext or creates one if none. Usually not needed with SparkSession (use `SparkSession.getOrCreate()`).

96. **What languages can Spark run?** Scala, Java, Python (PySpark), and R (SparkR).

97. **What is Spark's master URL?** In `SparkSession.builder.master()`, you specify where to run: e.g. `local[*]` for local, or `yarn`, `spark://host:port`, etc. In Databricks, the master is handled by the cluster.

98. **What is the default parallelism in Spark?** Usually the number of CPU cores in the cluster. The SQL shuffle partition default is 200 (configurable via `spark.sql.shuffle.partitions`).

99. **What is a task and executor in Spark?** A task is a unit of work on one partition. An executor is a JVM process on a worker node that runs tasks.

100. **Difference between client and cluster mode?** In client mode, the Spark driver runs on the submitting machine; in cluster mode, the driver runs on one of the worker nodes. (ProjectPro Q: use cluster mode when client is not local to cluster to avoid latency).

101. **What is a Catalog in Spark/Databricks?** Metadata store of tables and views. Databricks Unity Catalog manages tables/views with permissions. Spark's metastore (Hive) catalogs tables for Spark SQL.

102. **How to create a table in Databricks?** Using SQL: `CREATE TABLE IF NOT EXISTS db.table USING delta AS SELECT * FROM source`, or save DataFrame with `df.write.saveAsTable("db.table")`.

103. **How do DataFrames compare to SQL tables?** A DataFrame is like a temporary table. You can create a temp view with `df.createOrReplaceTempView("t")` and query it via SQL; or save a DataFrame as a permanent table.

104. **What is a Delta table?** A table stored in Databricks Delta format (Parquet + transaction log) that provides ACID transactions and versioned data.

105. **Why use Delta Lake?** It brings ACID transactions, schema enforcement, and time travel to data lakes. Databricks writes use Delta by default for reliability.

106. **What are ACID transactions in Databricks?** Databricks uses Delta Lake to provide **Atomicity, Consistency, Isolation, Durability** for table writes. This means partial updates aren't visible, reads see consistent snapshots, etc.

107. **How does Databricks implement ACID?** Via a transaction log and optimistic concurrency. Each write is a transaction: write new files, then commit new version in log. Writes are atomic and durable in cloud storage.

108. **How do you read/write data from Databricks notebooks?** Use `spark.read` (for files or tables) and DataFrameWriter. For example, `spark.read.parquet("/mnt/data/file.parquet")` or `spark.read.table("db.table")`. Writing: `df.write.format("delta").save("/mnt/delta/dt")` or `df.write.saveAsTable("db.table")`.

109. **What is Unity Catalog?** Databricks' unified governance layer. It manages tables, files, ML models with centralized access controls. Not to be confused with Spark's metastore (though catalog conceptually similar). (No direct ref cited).

110. **What is DBFS?** Databricks File System: an abstraction over your cloud storage (S3/ADLS/GCS). You access it via `/dbfs/` path or `dbutils.fs`. It stores workspace files and table data.

111. **What is `dbutils`?** A Databricks utilities module in notebooks for filesystem (dbutils.fs), secrets, widgets, etc. For example, `dbutils.fs.ls("/databricks-datasets")`.

112. **Explain Databricks Job vs Notebook.** A Notebook is an interactive environment. A Job is a scheduled or triggered workflow (can run notebooks or JARs). Jobs can also be chains of tasks with dependencies.

113. **How to schedule a notebook?** Use Databricks Jobs: create a Job that runs a notebook on a schedule or triggered. (No citation needed.)

114. **What is Databricks SQL?** Databricks' SQL analytics product: run SQL queries on Delta tables. It uses SQL warehouses under the hood and integrates with BI tools.

115. **What are Delta Live Tables (DLT)?** (If asked) A higher-level ETL tool by Databricks for declarative pipelines. Not directly part of PySpark. (Probably skip details.)

116. **What is a Spark cluster?** A set of machines (driver + executors) running Spark applications. In Databricks, a cluster is created from the workspace UI or API.

117. **What languages in Databricks notebooks?** You can mix Python (PySpark), SQL, Scala, and R. Use `%python` or `%sql` magic in notebooks.

118. **How to enable Spark UI in Databricks?** Click "View Spark UI" from the cluster or job page in the workspace to access the Spark web UI for jobs.

119. **What is a Spark executor?** A JVM process on a worker node that runs tasks and holds in-memory data. Executors talk to the Spark driver via TCP.

120. **What is data locality and why is it important?** Tasks run on nodes where data resides (e.g. HDFS block). Reduces network transfer, improving performance.

121. **How do you read data from Azure/S3 in Databricks?** Mount or use credentials. Example: `spark.read.csv("s3://bucket/file.csv")` after setting AWS keys, or use `dbutils.fs.mount`.

122. **Why use `spark.sql()` vs `spark.read`?** `spark.sql()` can query tables/views or perform complex SQL. `spark.read` is lower-level for file I/O. But both can achieve similar results since DataFrame API and SQL share the engine.

123. **What are Data Sources in Spark?** APIs to read/write various formats (CSV, JSON, Parquet, JDBC, Kafka, Delta, etc.). You specify format in `spark.read.format("...")`. Spark has DataSource API v2 for extensibility.

124. **Explain Catalyst optimizer.** Spark SQL's query optimizer that analyzes logical plans and rewrites them into efficient physical plans (predicate pushdown, projection pruning, etc.).

125. **What is whole-stage code generation?** An optimization where Spark generates Java bytecode at runtime for entire pipelines of operations, reducing CPU overhead.

126. **What is Tungsten?** Spark's off-heap memory and code generation engine that improves in-memory processing efficiency.

127. **Difference between DataFrame and Dataset in Spark?** (In Java/Scala) DataFrame = untyped `Dataset[Row]`. `Dataset` has type safety (`Dataset[MyClass]`). Python only has `DataFrame`. Datasets

are type-safe and can use lambda functions.

128. **Can PySpark use Datasets?** No; Python's dynamic typing means there's no compile-time type safety. The DataFrame in PySpark is equivalent to Scala's `Dataset<Row>`.

129. **Why is Spark fast?** In-memory computation, lazy evaluation, and optimized execution (Catalyst+Tungsten) give Spark significant speedups over disk-based engines.

130. **Explain in-memory computing.** Spark tries to keep intermediate data in RAM instead of writing to disk (like MapReduce does). This is the basis for its high speed.

131. **What file formats are most efficient in Spark?** Columnar formats like Parquet or ORC are efficient for Spark because Spark can skip columns (projection) and compress data. Delta Lake (Parquet + transaction log) is recommended for read/write.

132. **What is predicate pushdown?** Spark and data sources (like Parquet) can apply filters at the storage level, reading only necessary data. For instance, a Parquet file reader will only load columns that are used.

133. **What is JDBC in Spark?** Use `spark.read.jdbc(url, table, properties)` to read from relational DBs (MySQL, Postgres). You need the JDBC driver JAR.

134. **What is `spark-submit`?** The command-line tool to run Spark applications. You specify `--master`, application JAR/Python file, etc. (In Databricks, use the UI or Jobs instead.)

135. **Explain `foreachPartition`.** An action to apply a function to each partition's iterator, often used for writing to external systems (since it yields no return).

136. **Explain spark UI stages.** The Spark UI shows jobs, stages, tasks. It helps you see time taken, shuffle sizes, and diagnose performance.

137. **What is `local[*]` in master?** It runs Spark locally on all cores. E.g., `local[4]` uses 4 threads. In Databricks, clusters use YARN or Kubernetes by default.

138. **How to monitor Spark jobs?** Use the Spark UI (available in Databricks under Cluster → Spark UI), or Ganglia/Prometheus metrics. Databricks also provides Job details pages with metrics.

139. **What is the role of a driver program?** The main program that creates the SparkContext/SparkSession and issues transformations/actions. It schedules tasks on executors. In Databricks, each notebook session has a driver process.

140. **Spark version vs Databricks Runtime.** Databricks Runtime includes Spark plus optimizations (Photon, Delta Lake). For example, DBR 11.x uses Spark 3.x.

141. **What is a DataFrame's `show()` method?** An action that prints the first 20 rows in a tabular form (truncated by default). Doesn't trigger a job if already cached.

142. **What is an example of using Spark SQL in code?** `python df.createOrReplaceTempView("t") result = spark.sql("SELECT col, COUNT(*) FROM t GROUP BY col")` Here Spark runs SQL and returns a DataFrame.

143. **Why avoid `collect()`?** It brings all data to driver's memory. Use only if data is small. Otherwise use `take(n)` or write to storage.

144. **What is `.show(truncate=False)`?** Shows full content of each column in output. By default Spark truncates long strings.

145. **How to cast a column?** Use `df.withColumn("col", df.col.cast("newType"))`. Example: `df.withColumn("id", df.id.cast("string"))`.

146. **Explain window function example.** (Advanced) E.g., `from pyspark.sql.window import Window; w = Window.partitionBy("dept").orderBy("salary"); df.withColumn("rank", rank().over(w))`.

147. **What is `spark.range()`?** A convenience function to create a DataFrame with a column `id` from a range of longs. Useful for testing.

148. **What is AQE?** Adaptive Query Execution in Spark (3.x) – Spark can optimize shuffle and join strategies at runtime. (Example: change number of partitions after shuffle based on data size).

149. **Explain `.coalesce()` vs `.repartition()`.**

- `.coalesce(n)`: reduce partitions to `n` without full shuffle (narrow dependency).
- `.repartition(n)`: reshuffle all data into `n` partitions (wide dependency). Use `repartition()` when you want more or evenly-distributed partitions.

150. **What is caching vs storing?** (Synonymous in Spark). `cache()` and `persist()` store data in memory for reuse.

151. **What is Spark's memory management?** Spark divides JVM heap into execution (for shuffle, sort) and storage (caching) regions. Modern Spark uses unified memory by default.

152. **What is ShuffleFile?** (Internal) Spark writes intermediate shuffle data (files) during shuffle. These can reside on executor local disk.

153. **What is `df.take(n)`?** Action that returns first `n` rows as a list of `Row` objects. Useful instead of `collect()` when n is small.

154. **What is `df.count()` under the hood?** It performs a map-reduce: counts partitions and sums them. Triggers a full scan of the data.

155. **How to handle skewed keys in groupBy?** Use `df.repartition("skewKey").groupBy("skewKey")` to force shuffle, or pre-aggregate with map-side combine.

156. **What is Catalyst's predicate pushdown?** Spark filters data early when reading from data sources (like Parquet), pushing the filter to data scan.

157. **What is unified memory (in Spark 1.6+)?** A combined region for execution and storage, allowing flexible use of memory. Before 1.6, storage (cache) had fixed memory.

158. **What is the difference between `cache()` and `persist(MEMORY_ONLY)`?** In Spark 2.0+, they are equivalent (cache uses MEMORY_AND_DISK by default).

159. **How to select distinct values of a column?** Use `df.select("col").distinct()`. To drop duplicates of whole rows, use `df.dropDuplicates()`.

160. **How to show DataFrame schema?** Use `df.printSchema()` or `df.schema` to view column names and types.

161. **How to filter with SQL expression?** Use `df.filter("colA > 5 AND colB == 'X'")` or SQL-style syntax `df.where("colA > 5")`.

162. **What is `.alias()`?** A method to rename a column or expression inline, e.g. `df.c_acctbal.alias("balance")`. Often used in aggregations (e.g. `.agg(avg("x").alias("avg_x"))`).

163. **What are some built-in functions?** Many in `pyspark.sql.functions`, e.g. `avg`, `sum`, `max`, `round`, `floor`, `expr`, `col`, etc. Import them to use.

164. **How to drop rows with nulls?** `df.na.drop()`. You can specify subset or threshold. E.g., `.drop("any")`. Or `df.filter(df.col.isNotNull())`.

165. **How to fill nulls?** `df.na.fill(0)` to replace all numeric nulls with 0. Can do per-column: `df.fillna({"col":value})`.

166. **What is Window partition?** A way to define groups for window functions. Example: `Window.partitionBy("dept").orderBy("salary")`.

167. **What is `sort` vs `orderBy`?** They are synonyms in DataFrame API. Both return a new DataFrame sorted by given columns.

168. **What is saveAsTable vs save?** `saveAsTable("db.tbl")` writes to managed table. `save(path)` just writes files. Tables are managed in metastore (or Unity Catalog).

169. **What is Delta Lake?** Databricks' open-source storage layer that adds ACID transactions, scalable metadata, and time travel on Parquet data.

170. **How to do time travel in Delta?** Read a Delta table as of an earlier version: e.g., `spark.read.format("delta").option("versionAsOf", 2).load(path)`. (Databricks topic.)

171. **What is data lineage?** RDD lineage graph, or in Databricks, the ability to trace how data flowed (Unity Catalog lineage). In Spark, lineage is the record of transformations for fault recovery.

172. **What is a broadcast hash join?** When Spark broadcasts the smaller table and does a hash join on each executor, reducing shuffle. It's faster when one side is small.

173. **What is partition pruning?** Spark can avoid scanning partitions by using partition filters. For example, if data is partitioned by date and you filter by date, only relevant partition files are read.

174. **What is a black box UDF vs pandas UDF?** (Advanced) Pandas UDF (vectorized UDF) uses Apache Arrow to process batches in Python, giving better performance than standard UDF.

175. **How to configure Spark in code?** Use `SparkSession.builder.config("spark.executor.memory","4g")` or set configurations with `--conf` in `spark-submit`. In Databricks, cluster config is done in UI.

176. **Explain `mapPartitions`.** A transformation that gives an iterator of each partition to a function. Useful for doing expensive setup once per partition (e.g. database connection).

177. **What are eager vs lazy operations?** Eager = actions (immediately run). Lazy = transformations (deferred).

178. **How to persist DataFrame across sessions?** Write to an external storage (Delta table); DataFrames themselves are ephemeral to SparkSession. You cannot share an in-memory cache across sessions.

179. **What is reduceByKey?** An RDD operation (not DataFrame) that aggregates by key. Equivalent to `groupByKey` + reduce. (In DataFrame, use `groupBy(key).agg(...)`.)

180. **Explain join vs crossJoin.** `join` with no condition does a Cartesian product. `crossJoin()` explicitly does full Cartesian. Use carefully (explodes rows).

181. **What is a UDT (User Defined Type)?** (Rarely asked) A way to define a custom type for DataFrames in Scala/Java. (Skip in PySpark context).

182. **How to tune Spark jobs?** Adjust partitions, caching, memory. Use Spark UI to identify slow stages (e.g. large shuffles). Databricks auto-configures many settings but you can still tune memory and cores per executor.

183. **What is the difference between coalesce and repartition?** (Covered) Coalesce without shuffle (reduce partitions only) vs repartition with full shuffle.

184. **What is Web UI / Spark UI?** The web interface showing jobs, stages, tasks, executors. Helps profile Spark applications. Accessible from Databricks cluster UI.

185. **What is a broadcast hint?** In DataFrame API, `from pyspark.sql.functions import broadcast; df.join(broadcast(other), ...)` to force broadcast join.

186. **What is a Cartesian product?** A join without keys, or using `crossJoin()`. It multiplies row counts (do not do unless intentional).

187. **What are accumulators used for?** Counting or summing values across tasks. For example, counting bad records in a distributed job. Only "addable" in tasks and readable by driver.

188. **How to use accumulators?** `acc = spark.sparkContext.accumulator(0)`, then in RDD map: `acc += 1`. After action, `acc.value` has total.

189. **What are PySpark broadcast variables?** (Restatement) See Q25–26 above.

190. **How does Spark handle failures?** If an executor fails, Spark re-schedules its tasks elsewhere. If a driver fails (in cluster mode), job fails.

191. **What is speculative execution?** Spark can launch duplicate tasks if a task is slow, using the first result. Useful for straggler tasks.

192. **Explain Spark's cluster modes (client vs cluster).** (Covered) Client = driver on local; cluster = driver on worker.

193. **How to count distinct values?** `df.select("col").distinct().count()`. Or `df.agg(countDistinct("col"))`.

194. **How to merge two DataFrames?** Use `df1.union(df2)` (schemas must match). For SQL-like full join across all columns, use join.

195. **What are options to optimize writes?** - Use `coalesce` to write fewer files, - write in parallel (one file per partition). - Avoid small files. - Choose appropriate `mode` (e.g. overwrite with `spark.sql.sources.parallelPartitionDiscovery.threshold` to parallelize table overwrite).

196. **What is `cacheTable`?** A Spark SQL method: `spark.catalog.cacheTable("table")` caches a table into memory.

197. **How to uncache a DataFrame?** `df.unpersist()` (or `cache=False` when writing) frees its cached data.

198. **What is a Snowflake/Spark connector?** Databricks often uses a proprietary connector to query Snowflake via Spark. (Not a core Spark concept.)

199. **What is dynamic partition overwrite?** (Databricks feature) When writing to a partitioned table, overwrite only the partitions being written (instead of whole table).

200. **Why learn PySpark and Databricks?** (Meta question) Because Spark is a leading big data engine and Databricks is a popular managed platform. Mastering them is valuable for ETL, data engineering, and analytics roles.

These questions cover fundamentals (Spark concepts, DataFrames, SQL), PySpark specifics (Python API details), and Databricks environment knowledge (clusters, notebooks, Delta Lake). Use the references above for deeper study of each topic.

Sources: Spark and Databricks official docs, IBM article, Databricks resources, and tutorial/interview Q&A sites.