Q1. What is the difference between **RDD**, **DataFrame**, and **Dataset**?

**RDD (Resilient Distributed Dataset)**

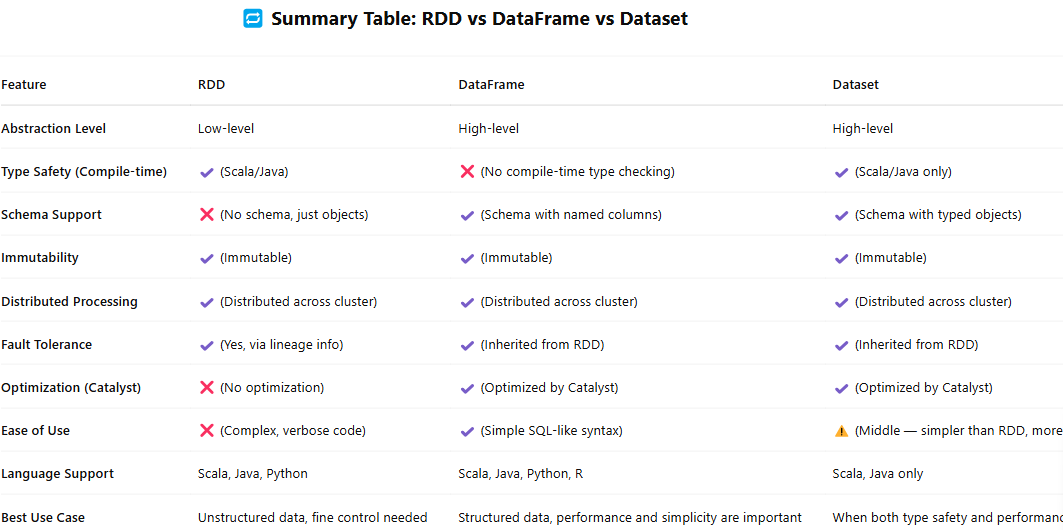
* **RDD is the core abstraction in Apache Spark**, which means it is the fundamental data structure used to represent distributed data in memory across a cluster.
* **RDD is designed for low-level data operations**, which allows developers to apply custom transformations and actions using functions like map, filter, reduce, and more.
* **RDD does not use a schema**, so the structure of the data is not organized into columns like in DataFrames; instead, it simply stores objects.
* **RDD is an immutable and distributed collection of objects**, which means once created, the data inside an RDD cannot be changed, and it is automatically divided and processed across multiple nodes in the cluster.
* **RDD is fault-tolerant**, which means if any part of the data is lost due to a node failure, Spark can automatically recompute the lost data using the RDD’s lineage (history of transformations).
* **RDD is not optimized by Spark’s Catalyst optimizer**, so it does not benefit from query optimization, which can lead to slower performance compared to DataFrames and Datasets for large-scale structured data.
* **RDDs are strongly typed in Scala and Java**, which means type errors can be caught at compile time.
* **RDD is mostly used when you need fine control over data or are working with unstructured or semi-structured data**, especially when schema and SQL-like operations are not needed.

**DataFrame**

* **DataFrame is a high-level abstraction in Apache Spark**, built on top of RDDs, and is designed to work with structured data efficiently.
* **DataFrames support schema**, which means the data is organized into named columns, just like a table in a relational database.
* **DataFrame operations are optimized by Spark’s Catalyst optimizer**, which automatically improves the performance of queries through logical and physical optimization.
* **DataFrame is designed for easier and faster data operations**, especially for big data processing, using SQL-like expressions and built-in functions.
* **DataFrames are immutable and distributed**, meaning the data is processed across multiple nodes, and once created, the data cannot be changed.
* **DataFrames are not strongly typed**, which means you do not get compile-time type checking, and errors may be caught only during execution.
* **DataFrame is suitable for processing large volumes of structured data**, where performance and ease of use are important, and the structure of the data is known.
* **DataFrame supports multiple languages**, including Scala, Java, Python, and R, making it more flexible for different developers.

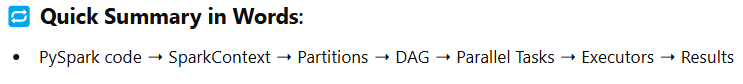
**Dataset**

* **Dataset is a high-level, strongly typed abstraction in Apache Spark**, which combines the advantages of both RDDs and DataFrames.
* **Datasets provide compile-time type safety**, meaning the compiler can catch errors early when you define a specific type or class for your data.
* **Datasets support schema**, so the data is organized into named columns, just like in DataFrames, which helps Spark understand the structure of the data.
* **Datasets are optimized using Spark's Catalyst optimizer**, which improves performance by applying logical and physical query optimizations automatically.
* **Datasets are immutable and distributed collections of typed objects**, which means once the dataset is created, it cannot be changed, and it is automatically divided and processed across the Spark cluster.
* **Datasets allow both functional programming (like in RDD)** and SQL-like operations (like in DataFrames), giving you the flexibility to work in both styles.
* **Datasets are available only in Scala and Java**, so they are not supported in Python or R as of now.
* **Datasets are useful when you want the performance of DataFrames along with the type safety and object-oriented benefits of RDDs.**



Q2. How does PySpark achieve parallel processing?

* **PySpark achieves parallel processing by running on top of the Apache Spark engine**, which is designed to distribute data and computation across multiple nodes in a cluster.
* **When a PySpark program runs, it first creates a SparkContext**, which connects the application to the Spark cluster and coordinates resource usage.
* **PySpark code (written in Python) is translated into Spark jobs**, which are broken down into **tasks** and distributed to **executors** running on different worker nodes.
* **Spark uses the DAG (Directed Acyclic Graph) scheduler**, which plans the execution flow and distributes the tasks across the cluster in an optimized manner.
* **Although PySpark is written in Python, it interacts with the JVM-based Spark engine** through a component called **Py4J**, allowing it to execute tasks in parallel using the power of the Spark engine.
* **Each transformation in PySpark is lazily evaluated**, meaning operations are not executed until an action is called (like collect(), count(), or save()), at which point Spark schedules the entire job and runs it in parallel.
* **Data is divided into partitions**, and Spark processes each partition in parallel using multiple cores or machines, which results in faster execution.
* **The actual computation happens on executors in the cluster**, which perform the tasks on the data partitions and return results back to the driver.



Q3. Explain **lazy evaluation** in PySpark ?

**Lazy evaluation** in PySpark means that **operations on data are not executed immediately** when they are defined. Instead, they are only executed when an **action** (like collect(), count(), or save()) is triggered. This helps optimize performance by reducing the number of operations performed and enabling **Spark to intelligently optimize the execution plan**.

Q4. What is **SparkContext**, and why is it important?

**SparkContext** is the entry point to using Spark, connecting your PySpark application to the Spark cluster. It manages the execution of the program, scheduling tasks, and resource allocation

* **Entry Point**: Connects PySpark application to the Spark cluster.
* **Job Scheduler**: Manages job execution and schedules tasks across the cluster.
* **Cluster Resource Manager**: Requests memory and CPU from cluster manager (YARN, Mesos, Kubernetes).
* **Creates RDDs/DataFrames**: Facilitates creation of RDDs and DataFrames for distributed data processing.
* **Parallel Processing**: Distributes data into partitions and processes them in parallel across nodes.
* **Initialization**: Must be created before performing any Spark operations.
* **Resource Allocation**: Manages resource allocation to ensure efficient execution.
* **Cluster Connectivity**: Acts as a bridge between the Spark application and the cluster manager.

Q5. How do you handle large file processing in PySpark?

* **Distributed Processing**: Spark splits large files into **partitions** and processes them in parallel across the cluster.
* **Optimized File Formats**: Use **Parquet/ORC** for better read/write performance.
* **Increase Partitions**: Use **repartition/coalesce** to increase parallelism and reduce shuffling.
* **Efficient Data Structures**: Use **DataFrames/Datasets** for better optimization.
* **Lazy Evaluation**: Transformations are executed only when an **action** is triggered.
* **Avoid .collect()**: Don’t use .collect() for large datasets to prevent memory overload.
* **Tune Spark Configurations**: Optimize **memory**, **cores**, and **shuffle partitions** settings.
* **Broadcast Joins**: Use **broadcast** for small file joins to avoid large shuffling.
* **Partition by Key**: Partition data by key to balance distribution and improve joins.
* **Compression**: Compress files using **Snappy** or **Gzip** to reduce I/O overhead.
* **Caching for Iterations**: Use .cache() for iterative operations to save intermediate results.
* **Spark Streaming**: Use **Spark Streaming** for real-time data processing.

Q6. What is the difference between **actions** and **transformations** in PySpark?

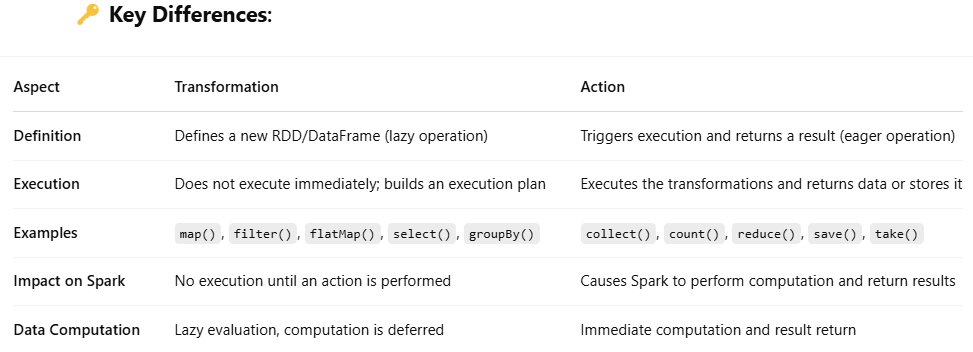
In **PySpark**, **actions** and **transformations** are two types of operations that are performed on RDDs (Resilient Distributed Datasets) and DataFrames. Here's the difference:

**Transformations:**

* **Definition**: Transformations are **lazy operations** that define a new RDD or DataFrame based on the original data. They **do not trigger execution** immediately but rather build an execution plan.
* **Behavior**: When you apply a transformation, Spark just **records the operation** and doesn’t execute it until an action is performed.
* **Examples**: map(), filter(), flatMap(), groupBy(), select(), repartition().
* **Key Point**: They are **lazy**, meaning they don’t compute results immediately.

**Actions:**

* **Definition**: Actions are operations that **trigger execution** of the transformations and **return a result** or **save the data** to external storage (e.g., HDFS, S3).
* **Behavior**: Actions force the execution of the transformations and compute the final result.
* **Examples**: collect(), count(), reduce(), save(), first(), take().
* **Key Point**: They are **eager**, meaning they cause Spark to perform the computation and return the result.



Q7. How does Spark handle data partitioning in distributed environments?

* **Initial Partitioning**: Spark splits data into partitions when reading from storage.
* **Repartitioning**: You can control partitions with repartition() and coalesce().
* **Shuffling**: Data movement across partitions, often triggered by certain operations.
* **Custom Partitioning**: Allows optimizing data distribution, especially for joins.
* **Broadcasting**: Broadcast small datasets to avoid costly shuffling.
* **Fault Tolerance**: Partitions provide fault tolerance by allowing recomputation on failure.
* **Spark SQL Partitioning**: Enables optimized query execution with partitioned data.

Q8. Explain the concept of **fault tolerance** in PySpark.

* **RDDs (Resilient Distributed Datasets)**: RDDs are designed to be fault-tolerant by storing the lineage (execution plan) of transformations. If a partition is lost due to node failure, Spark can **recompute** the lost data using its lineage information.
* **Data Replication**: In distributed storage systems like **HDFS** or **S3**, data is often replicated across multiple nodes. If one node fails, the data can be retrieved from another replica.
* **Task Re-execution**: If a task fails during execution, Spark can **re-execute** the task on a different node based on the **lineage** information.
* **Checkpointing**: Checkpointing is the process of saving the RDD’s state to **persistent storage** (HDFS, S3) at certain intervals, which helps in recovering data in case of failure during long-running computations.
* **Executor and Node Failures**: Spark ensures fault tolerance by managing executor failures. If an executor fails, the task is rescheduled on another available node.
* **Speculative Execution**: Spark supports **speculative execution**, where if a task is running slower than others, it can be re-executed on another node to improve performance.
* **Distributed Data**: Spark distributes data across multiple nodes, reducing the risk of data loss and improving fault tolerance by ensuring the system can recover from node failures.

Q9. How do you **broadcast** **variables** in Spark, and when should you use them?

* **Definition**: Broadcast variables allow efficient sharing of read-only data across all worker nodes.
* **How to Use**: Use sc.broadcast(variable) to broadcast a variable.
* **When to Use**: Use for small datasets that are reused across tasks, like lookup tables, to avoid data shuffling.

Q10. What are **accumulators** in PySpark, and how do they differ from **broadcast** variables?

**Accumulators**:

* **Definition**: Accumulators are variables that allow **only additive operations** (e.g., sum, count) across tasks in Spark, enabling **fault-tolerant aggregation** of values.
* **Use**: They are typically used for debugging or gathering aggregate information during distributed computations (e.g., counting errors).
* **Behavior**: Only the driver can read the accumulator; tasks can add to the accumulator, but the value can only be read by the driver after execution.

**Broadcast Variables**:

* **Definition**: Broadcast variables allow **efficient sharing of read-only data** across worker nodes to avoid data duplication.
* **Use**: They are used for large datasets that remain constant during execution (e.g., lookup tables).
* **Behavior**: Broadcast variables are **read-only** and can be accessed by tasks on worker nodes.

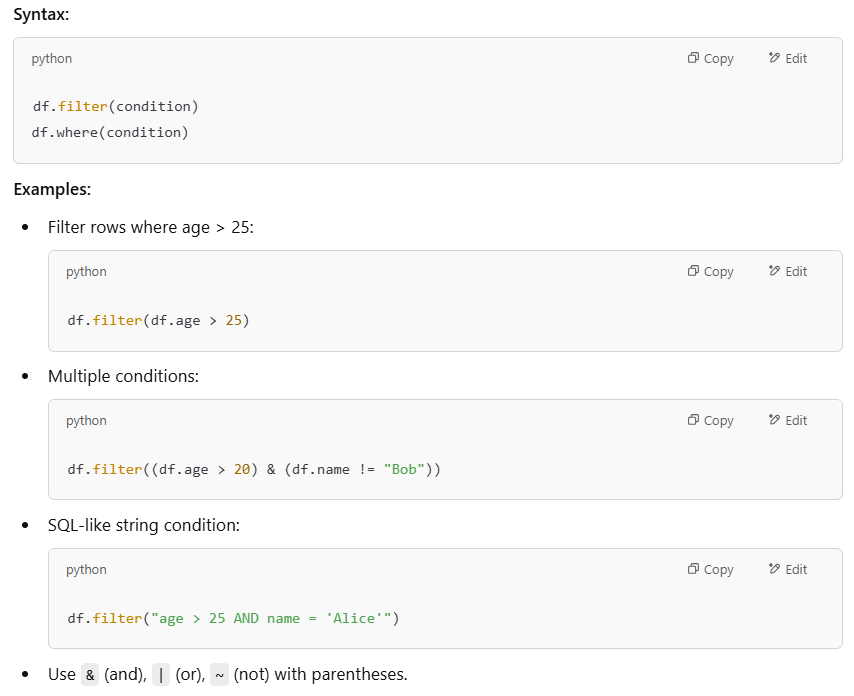
--------------------------------------------------------------------------------------------------------------------------

**DataFrame and Dataset Operations Based Questions**

--------------------------------------------------------------------------------------------------------------------------

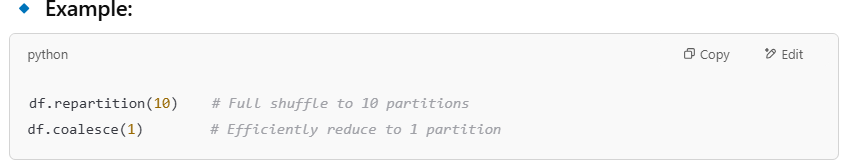
Q11. How do you perform data **filtering** using PySpark DataFrames?

* Use .filter() or .where() to filter rows based on conditions.



Q12. What is the difference between **repartition()** and **coalesce()**, and when would you use each?

* **repartition(n)**: Use when you need to **increase** partitions or ensure better data distribution before a wide transformation (like joins).
* **coalesce(n)**: Use when you want to **decrease** the number of partitions (e.g., before writing to a single file) to avoid a full shuffle.

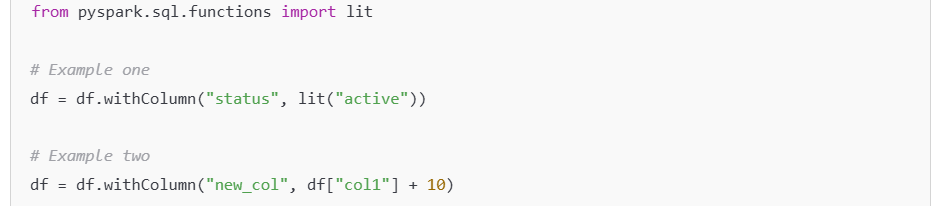


Q13. How do you handle **missing or null values** in PySpark?



Q14. How can you add a **new column** to a DataFrame using withColumn()?

Add new column with **withColumn**():



Q15. How do you perform a **left join** between two DataFrames in PySpark?



* **df1.join(df2, df1["name"] == df2["name"], "left")**:Performs a left join on the name column, keeping all rows from df1, and matching rows from df2.
* If there is no match in df2, null values are filled for columns from df2.

Q16. What are **temporary views** in PySpark, and how do they differ from **global temporary** views?

**Temporary Views in PySpark**

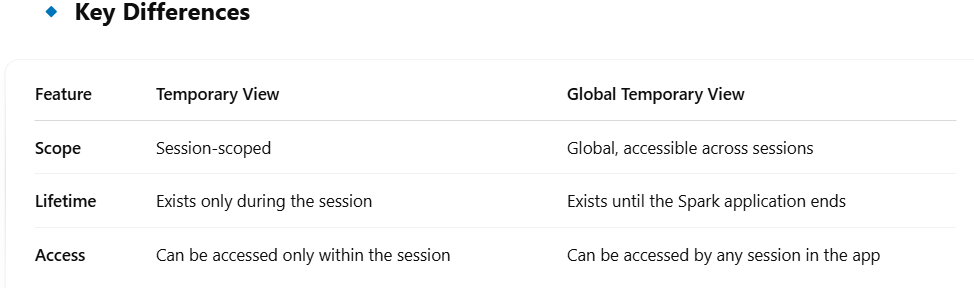
* **Definition:** A temporary view is a **session-scoped** table or view that exists only for the duration of the session.
* **Lifetime:** The view is dropped when the session is terminated.
* **Usage:** Typically used when you want to perform SQL operations in a single session.



**Global Temporary Views**

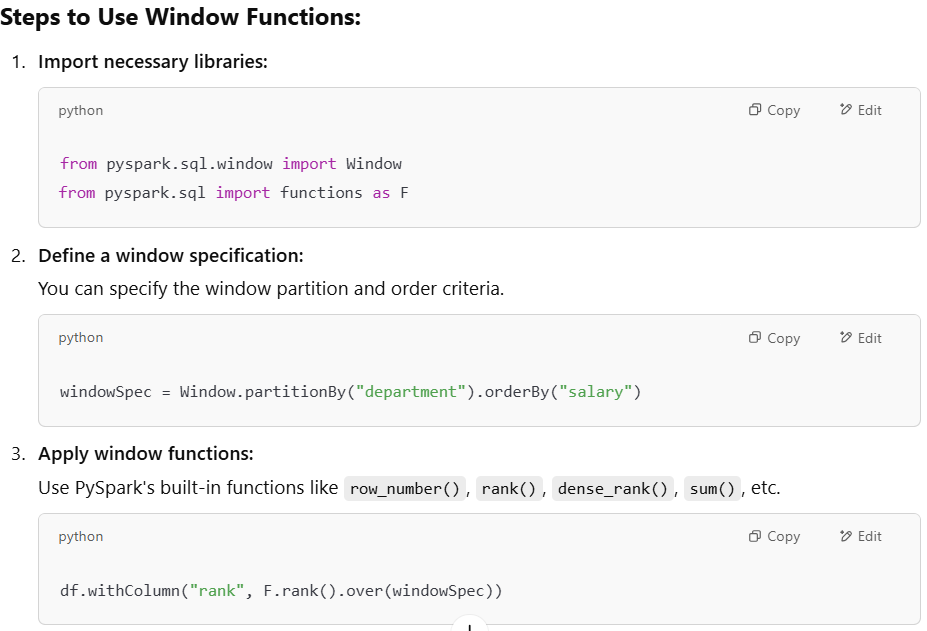
* **Definition:** A global temporary view is a **shared, global view** across all sessions. It can be accessed by any session.
* **Lifetime:** The view exists until the Spark application terminates.
* **Usage:** Useful for sharing views across multiple sessions within the same Spark application.





Q17. How do you use **window functions** in PySpark for advanced analytics?

Window functions allow you to perform operations on a specific window of rows, similar to SQL’s **OVER()** clause. They are often used for advanced analytics like ranking, cumulative sums, or calculating moving averages.

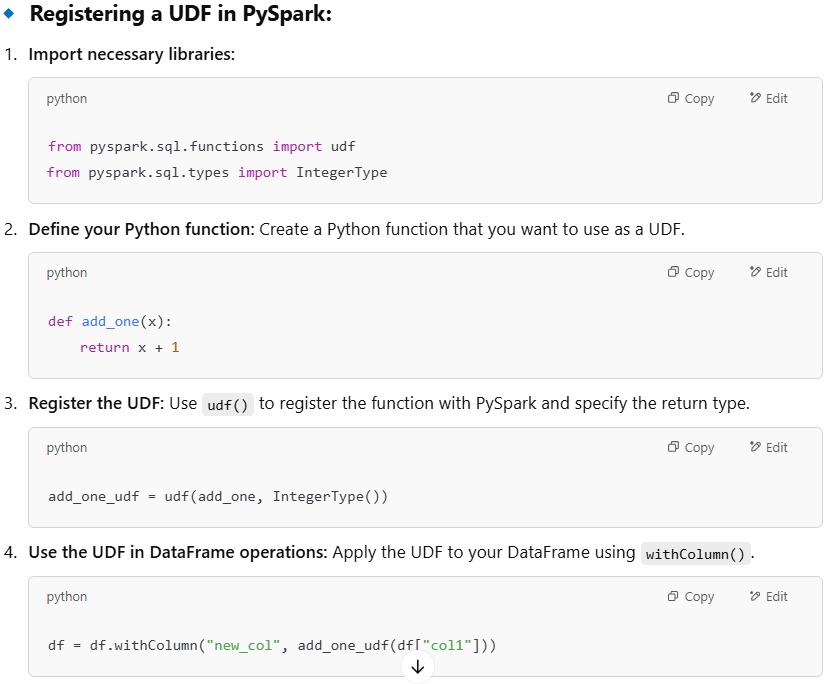


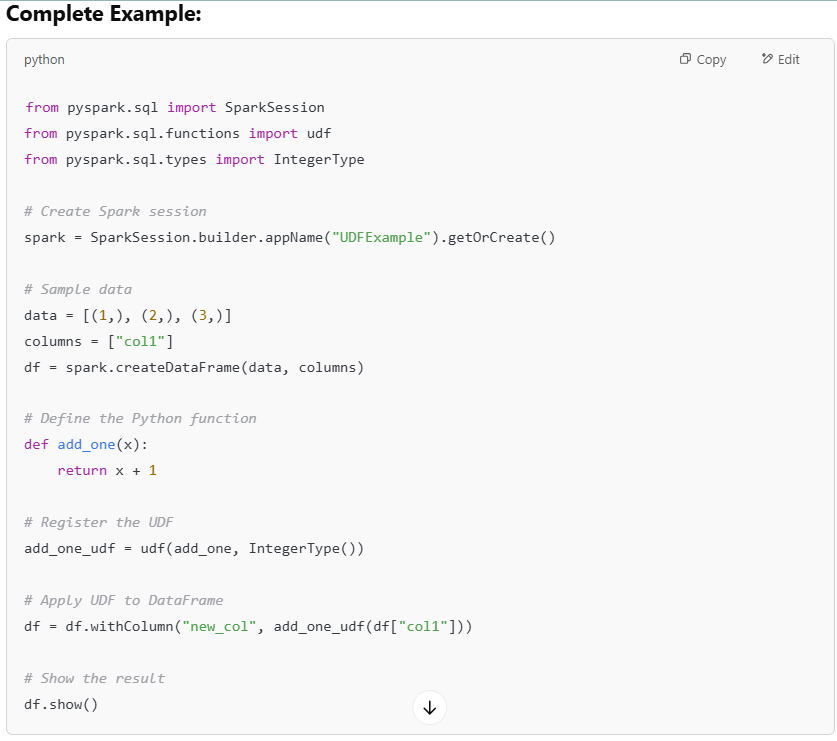


**Key Window Functions:**

* **row\_number()**: Assigns a unique row number.
* **rank()**: Ranks rows, skipping numbers for ties.
* **dense\_rank()**: Ranks rows without skipping numbers for ties.
* **sum(), avg(), max(), min()**: Aggregation functions within the window.

Q18. How can you register a **UDF (User-Defined Function)** in PySpark?

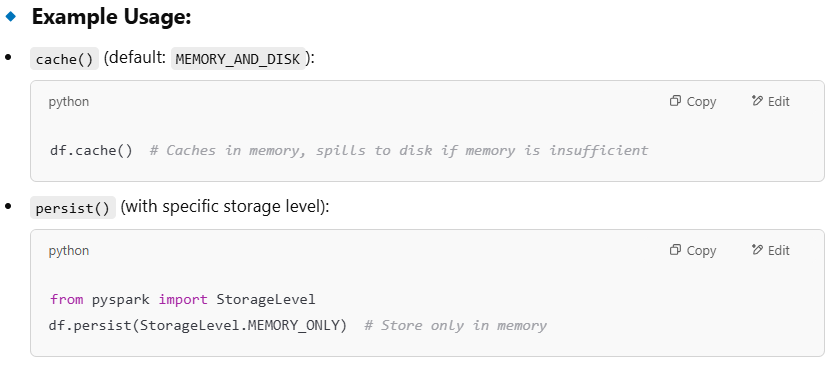




**Key Notes:**

* UDFs in PySpark are slower than built-in functions because they require serialization and deserialization between Python and Spark's JVM.
* It's best to use UDFs only when necessary, as built-in Spark SQL functions are usually more optimized.

Q19. What is the difference between **persist()** and **cache()?**



**Key Differences:**

* **cache()** is simply a **default shortcut** for **persist(StorageLevel.MEMORY\_AND\_DISK)**.
* **persist()** allows you to choose different **storage levels** for performance optimization

Q20. How do you **read and write** data in **Parquet, CSV, and JSON formats** in PySpark?



**Key Points:**

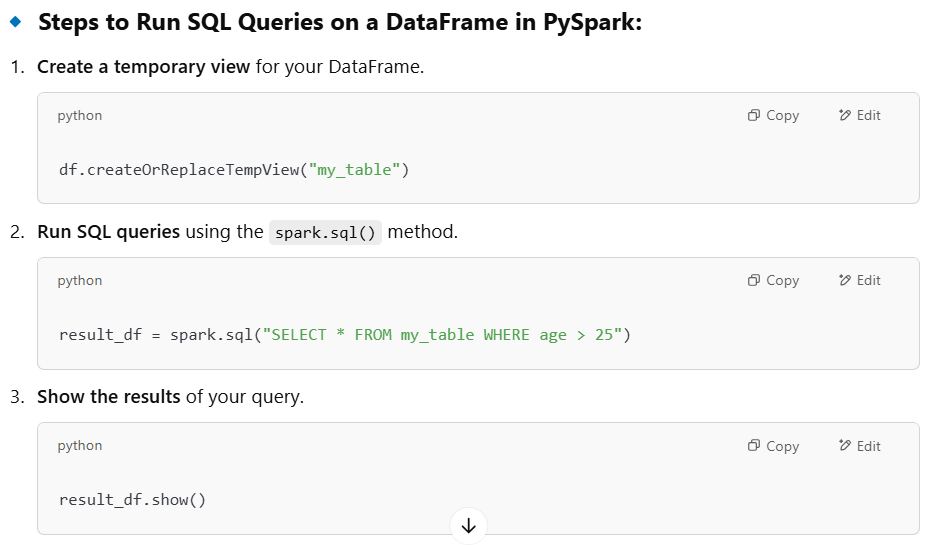
* **Writing:** You can use format() to specify the format while saving the data (Parquet, CSV, JSON).
* **Reading:** You can also use format() to load data in the specified formats.
* **Options for CSV:** .option("header", "true") for including headers, and .option("inferSchema", "true") for automatic schema inference.

-----------------------------------------------------------------------------------------------------------------------

**Spark SQL and Query Optimization Questions**

-----------------------------------------------------------------------------------------------------------------------

Q21. How do you run **SQL queries** on a DataFrame in PySpark?



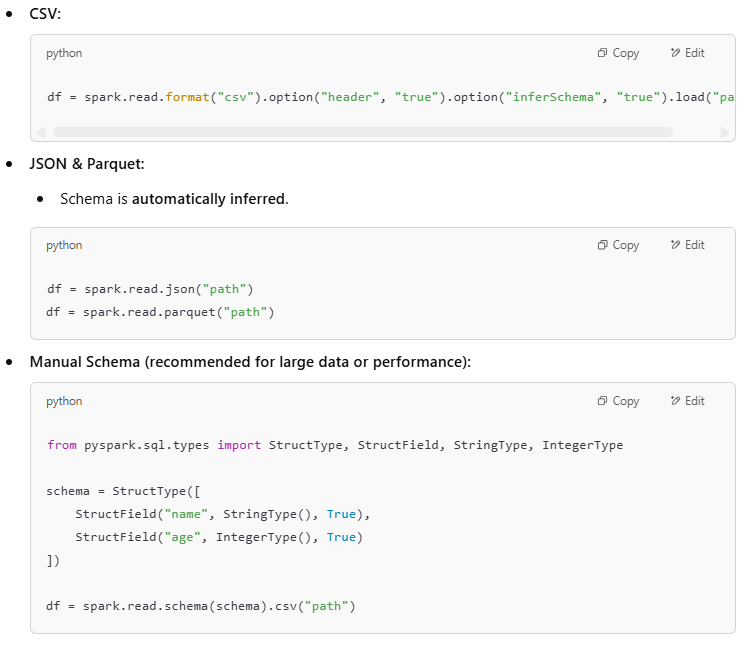
**Important Notes:**

* **Temporary Views:** You must register your DataFrame as a temporary view before running SQL queries on it.
* **SQL Syntax:** The SQL syntax is similar to traditional SQL, and you can use SELECT, WHERE, JOIN, etc.
* **Global Temporary Views:** You can also register global temporary views using createOrReplaceGlobalTempView() if you need the view to be accessible across sessions.

Q22. What is the purpose of **Catalyst Optimizer** in Spark SQL?

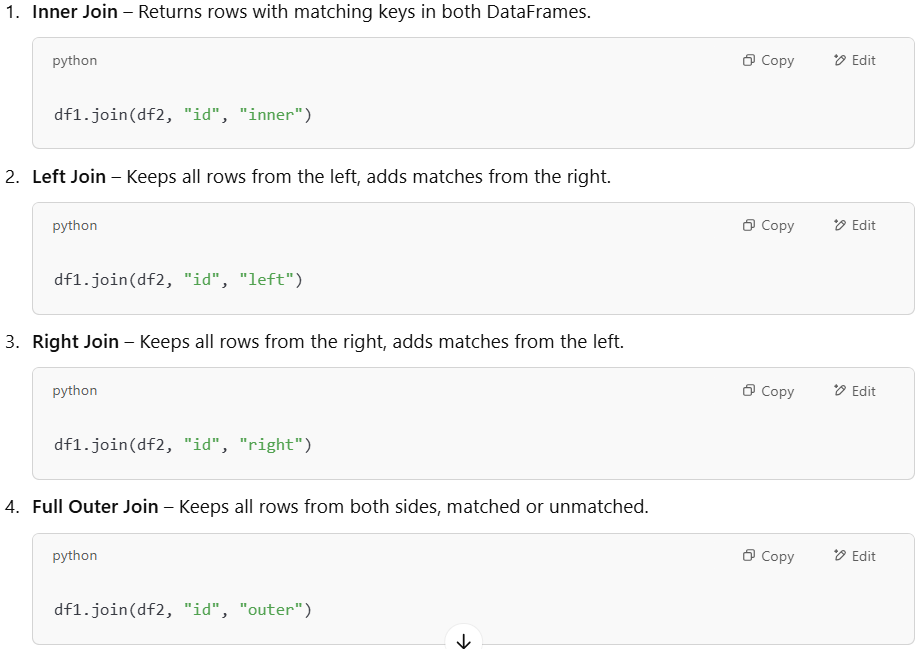
* **Purpose:** Optimizes query execution for better performance in Spark SQL and DataFrame operations.
* **Key Functions:**
  + **Query Parsing:** Converts SQL or DataFrame operations into an abstract syntax tree (AST).
  + **Logical Optimization:** Applies optimizations like predicate pushdown and constant folding.
  + **Physical Planning:** Generates multiple physical plans and selects the best one.
  + **Cost-Based Optimization:** Chooses the most efficient plan using a cost model.
  + **Code Generation:** Generates optimized bytecode for execution.
* **Benefits:**
  + Automatic performance improvements.
  + Supports complex optimizations (cost-based, code generation).
  + Makes query execution more efficient without manual tuning.

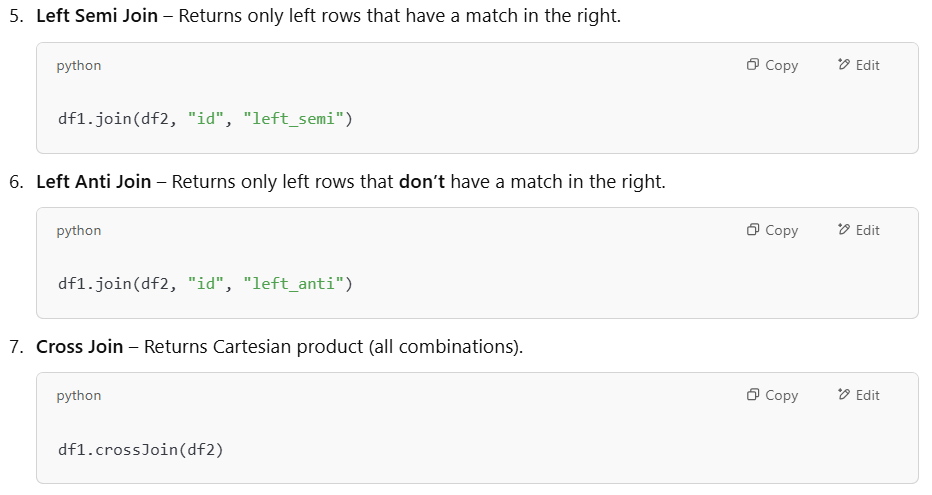
Q23. How do you handle **schema inference** when reading data from **external sources**?



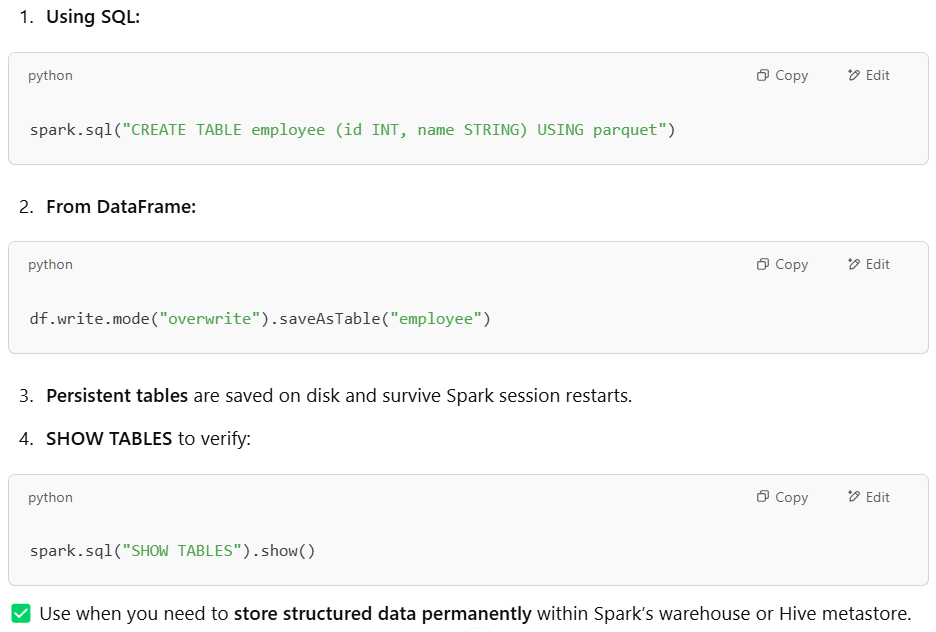
* Manual schema is faster and more efficient for large datasets. Use **inferSchema=true** only for small or unknown data.

Q24. What are the different **join types in Spark SQL**, and when would you use each?





Q25. How do you create a **persistent** table in Spark SQL?



Q26. How does **dynamic** **partition** pruning improve query performance?

* **Definition:** Dynamically skips unnecessary partitions **at runtime** during query execution.
* **Improves performance by:** Reading **only relevant partitions**, reducing **I/O and scan time**.
* **Used in: Join queries** where filter values come from another table.
* **Enabled by default:** From **Spark 3.0+**

**Result:** Faster queries, especially on large **partitioned tables**.

Q27. Explain how to **use broadcast joins** to optimize query performance.

* **Definition:** Broadcast join sends a **small table** to all worker nodes to avoid data shuffling.
* **Usage:** df1.join(broadcast(df2), "id")
* **When to use:** Use when one DataFrame is **small enough to fit in memory**.
* **Benefit:** Speeds up joins by reducing **network I/O** and **shuffle time**.

**Ideal for:** Joining **large fact** tables with **small dimension** tables.

Q28. What is **data skew**, and how do you handle it in **Spark SQL**?

* **Definition:** Data skew happens when **some partitions** have **much more data** than others.
* **Problems caused:**
  + **Slow tasks**
  + **Uneven resource usage**
  + **Query performance issues**
* **Handling techniques:**
  + **Salting keys:** Add random prefix to skewed keys to distribute data.
  + **Broadcast join:** Use when one side is small to avoid skew.
  + **Repartitioning:** Use repartition() to balance data.
  + **Skew join optimization:** Enable Spark’s skew join handling:

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

**Goal:** Balance load across tasks and improve **parallelism** and **performance**.

Q29. How can you perform **aggregations using SQL** queries on large datasets?

* **SQL Query Example:**

SELECT department, COUNT(\*) FROM employees GROUP BY department

* **Optimization Techniques:**
* **Partitioning:** Partition data by the aggregation column (e.g., department).
* **Bucketing:** Organize data into buckets to speed up aggregations.
* **Minimize Shuffling:** Use partitioning or **broadcast joins**.
* **Adaptive Query Execution (AQE):**

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

**Key Tip:** Use **partitioning** and **broadcast joins** to reduce data shuffling and improve performance.

Q30. How do you **enable query caching** in Spark SQL?

1. **What it is:** Query caching stores intermediate query results in memory to avoid recomputing them.
2. **How to enable caching:**
   * **Cache a DataFrame:** df.cache()
   * **Cache a SQL table:** spark.sql("CACHE TABLE my\_table")
3. **To persist with different storage levels:**

from pyspark import StorageLevel

df.persist(StorageLevel.MEMORY\_AND\_DISK)

1. **Clearing cached data:** spark.catalog.clearCache()

**Benefit:** Boosts performance by storing frequently accessed data in memory, reducing computation time for repeated queries.