**PYTHON**

1. **Exception handling**

An Exception is an Unexpected Event, which occurs during the execution of the program. It is also known as a run time error. When that error occurs, python generates an exception during the execution and that can be handled, which prevents your program from interrupting**.**

**Exception handling with try, except, else, and finally**

* Try: This block will test the excepted error to occur
* Except:  Here you can handle the error
* Else: If there is no exception then this block will be executed
* Finally: Finally block always gets executed either exception is generated or not

**Working of ‘try’ and ‘except’**

* First **try** clause is executed i.e. the code between **try** and **except** clause.
* If there is no exception, then only **try** clause will run, **except** clause will not get executed.
* If any exception occurs, the **try** clause will be skipped and **except** clause will run.
* If any exception occurs, but the **except** clause within the code doesn’t handle it, it is passed on to the outer **try** statements. If the exception is left unhandled, then the execution stops.
* A **try** statement can have more than one **except** clause.

A screenshot of a computer code

AI-generated content may be incorrect.

**Else Clauses in Python**

The code enters the else block only if the try clause does not raise an exception.

**Python finally Keyword**

Python provides a keyword finally, which is **always executed** after try and except blocks. The finally block always executes after normal termination of try block or after try block terminates due to some exception. Even if you return in the except block still the finally block will execute

A screenshot of a computer code

AI-generated content may be incorrect.

1. **Multithreading and Multiprocessing**

multithreading and multiprocessing are powerful techniques in Python for achieving concurrency, but they serve different purposes and have distinct characteristics.

**Multithreading**

Multithreading involves running multiple threads (smaller units of a process) within the same process. It is particularly useful for I/O-bound tasks, such as reading/writing files, network operations, or user interactions.

**Multiprocessing**

Multiprocessing involves running multiple processes, each with its own Python interpreter and memory space. This is particularly useful for CPU-bound tasks, such as heavy computations, because it can bypass the Global Interpreter Lock (GIL) that limits multithreading performance in Python.

**Key Differences**

**Memory:** Threads share the same memory space, while processes have separate memory spaces.

**GIL:** Threads are affected by the Global Interpreter Lock (GIL), which can be a bottleneck for CPU-bound tasks. Processes are not affected by the GIL.

**Overhead:** Creating and managing threads is generally less resource-intensive compared to processes.

Both techniques have their own use cases and can be very effective when used appropriately.

**Pyspark**

1. **Lazy evaluation**

In PySpark, lazy evaluation is a powerful concept that optimizes data processing tasks by postponing the execution of transformations until an action is called. This approach can significantly improve performance by avoiding unnecessary computations and enabling the optimization of the execution plan.

How Lazy Evaluation Works in PySpark

1. Transformations and Actions:
   * Transformations: Operations like map, filter, and reduceByKey that produce a new RDD or DataFrame from an existing one. These are lazily evaluated.
   * Actions: Operations like count, take, and saveAsTextFile that trigger the computation and return a result to the driver program or write data to an external storage system.
2. Execution Plan:
   * When you apply transformations to an RDD or DataFrame, PySpark doesn't execute them immediately. Instead, it records the transformations in a query plan, which is a sequence of steps required to compute the final result.
   * When an action is called, PySpark evaluates the entire query plan, optimizing it and executing the necessary transformations to produce the result

**Benefits of Lazy Evaluation**

* Performance Optimization: By postponing the execution of transformations, PySpark can optimize the execution plan and avoid redundant computations, leading to faster processing times.
* Resource Efficiency: Lazy evaluation allows PySpark to minimize the amount of data processed, reducing memory and CPU usage.
* Enhanced Debugging: As the execution is deferred until an action is called, you can inspect the execution plan before it is carried out, making it easier to identify potential performance bottlenecks or issues

**A screenshot of a computer code

AI-generated content may be incorrect.**

1. **Lakehouse architecture**

The Databricks Lakehouse architecture combines the best elements of data lakes and data warehouses to create a unified platform for data, analytics, and AI.

**Key Features**

1. Unified Architecture:
   * Integrates storage, processing, governance, sharing, analytics, and AI within a single platform.
   * Supports both structured and unstructured data, providing an end-to-end view of data lineage and provenance
2. Open Standards:
   * Built on open-source projects like Apache Spark™, Delta Lake, and MLflow.
   * Ensures data is always under your control, free from proprietary formats and closed ecosystems
3. Scalability:
   * Optimizes performance and storage automatically, ensuring low total cost of ownership (TCO).
   * Capable of handling both data warehousing and AI use cases, including large language models (LLMs)
4. Delta Lake:
   * Provides ACID transactions, scalable metadata handling, and unifies streaming and batch data processing.
   * Enhances data reliability and performance
5. Delta Sharing:
   * Enables secure sharing of live data from your lakehouse to any computing platform without replication and complicated ETL processes

**Benefits**

* Cost Efficiency: Reduces costs by eliminating data silos and simplifying data management.
* Performance: Delivers high performance for both analytics and AI workloads.
* Flexibility: Supports various data formats and integrates with major cloud providers.

**Data Plane (Compute Plane)**

The data plane, also known as the compute plane, is where your data processing tasks are executed. Here are some key aspects:

* Compute Resources: This plane includes the clusters and compute resources that run your data processing tasks. These resources can be either serverless or classic compute.
  + Serverless Compute: In this setup, Databricks manages the compute resources within its own account, providing a fully managed experience
  + Classic Compute: Here, the compute resources run within your cloud provider's account, giving you more control over the infrastructure
* Data Processing: All transformations, queries, and machine learning tasks are executed in the data plane. This includes running Apache Spark jobs, SQL queries, and other data processing tasks

**Control Plane**

The control plane is responsible for managing and orchestrating the various services and resources within the Databricks platform. Key components include:

* Backend Services: This includes the web application, REST APIs, and other services that Databricks manages. These services handle tasks such as job scheduling, cluster management, and workspace administration
* Security and Governance: The control plane manages security features like access control, data encryption, and compliance with data governance policies
* User Interface: The Databricks web application, which users interact with to create notebooks, manage clusters, and perform other tasks, is part of the control plane

**Interaction Between Data Plane and Control Plane**

* Job Submission: When you submit a job through the Databricks UI or API, the control plane schedules and manages the job, while the actual execution happens in the data plane.
* Cluster Management: The control plane handles the creation, scaling, and termination of clusters, while the data plane runs the workloads on these clusters.
* Monitoring and Logging: The control plane collects logs and metrics from the data plane to provide insights into job performance and resource utilization

This separation of concerns allows Databricks to provide a scalable, secure, and efficient platform for data processing and analytics.

1. **Performance optimization techniques for handling large dataset**

When working with large datasets in PySpark, performance optimization is crucial to improve execution speed and reduce resource consumption. Below are key techniques for optimizing PySpark performance:

**Use DataFrames Instead of RDDs**

* PySpark DataFrames are optimized with Catalyst Optimizer and Tungsten execution engine.
* Avoid RDD transformations unless necessary.

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

**Use Efficient File Formats**

* Prefer Parquet or ORC over CSV/JSON because they support columnar storage and compression.
* Example: Writing in Parquet format.

**df.write.parquet("output.parquet")**

**Repartitioning & Coalescing**

* Use repartition(n) to increase partitions (shuffling involved).
* Use coalesce(n) to reduce partitions efficiently (minimizes shuffling).

**df = df.repartition(10)**

**df = df.coalesce(5)**

**Partitioning on Key Columns**

* Partitioning large tables by a frequently filtered column speeds up queries.

**df.write.partitionBy("date").parquet("output/")**

**Minimize Shuffling**

* Use broadcast joins when joining a large table with a small table.

from pyspark.sql.functions import broadcast

**df\_join = df\_large.join(broadcast(df\_small), "key")**

* Avoid using Pandas UDFs when Spark built-in functions exist.

**Caching and Persistence**

* Use cache() if the DataFrame will be used multiple times.

**df.cache()**

**df.count()**

* Use persist(storage\_level) for specific storage options.

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

**Adjust Parallelism**

* Set the number of shuffle partitions based on cluster resources.

**spark.conf.set("spark.sql.shuffle.partitions", "200")**

**Optimize Joins**

* Use sort-merge join for large tables.
* Use broadcast join for small tables.

**Data Skew Handling**

* Salting technique: Append a random number to keys before join.
* Skew-aware partitioning: Identify skewed keys and handle them separately.

**Optimize Queries Using explain()**

* Use **df.explain(True)** to analyze the query plan and detect inefficiencies.

1. **Upsert**

Delta Lake supports Upsert (MERGE) using the MERGE INTO command. This allows you to insert new records and update existing records in a Delta table based on a matching condition.

A screenshot of a computer code

AI-generated content may be incorrect.

Matches records by id between the source and target Delta table.

If a match is found (WHEN MATCHED), it updates the existing records.

If no match is found (WHEN NOT MATCHED), it inserts new records.

1. **pivot vs unpivot**

Pivot and Unpivot are used to reshape data:

* Pivot: Converts row values into column headers.
* Unpivot (Melt): Converts columns into row values**.**

**PIVOT (Rows → Columns)**

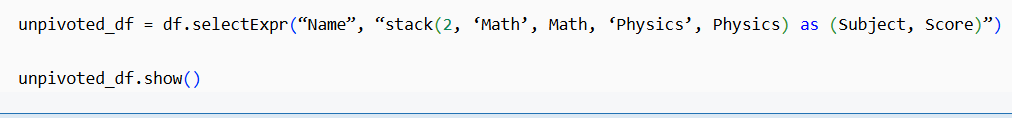
* Transforms unique row values of a column into separate columns.
* Aggregation is required when pivoting**.**

**A computer screen shot of a program

AI-generated content may be incorrect.**

**Unpivot:**

If you have a DataFrame in wide format (subjects as columns) and you want to unpivot it back to long format, you can use the melt operation. Unfortunately, PySpark does not have a direct melt function like some other libraries (e.g., pandas), so you need to achieve this using a combination of selectExpr and union operations.



1. **Different write modes in pyspark**

When writing DataFrames to storage (e.g., files, databases, Delta Lake), PySpark provides different write modes that control how existing data is handled.

**Overwrite Mode**

* **Deletes existing data and replaces it with new data.**
* Useful when you want to refresh a dataset completely.

**df.write.mode("overwrite").parquet("/path/to/output")**

**Append Mode**

* **Adds new data without deleting existing data.**
* Useful for incremental data loading.

**df.write.mode("append").parquet("/path/to/output")**

**Ignore Mode (Skip if Exists)**

* **Does nothing if the data already exists.**
* Avoids errors but won’t update existing data

**df.write.mode("ignore").parquet("/path/to/output")**

**Error (Default) / Fail Mode**

* **Fails if the data already exists** (default mode).
* Ensures no accidental overwriting.

**df.write.mode("error").parquet("/path/to/output")**

1. **Types of joins, left anti and left semi**

**Type of Joins**

1. Inner Join
2. Outer (Full) Join
3. Left Join
4. Right Join
5. Left Semi Join
6. Left Anti Join
7. Cross Join

**Creating Dataframe:**

**A screenshot of a computer code

AI-generated content may be incorrect.**

**Inner Join**

An inner join returns rows from both dataframes that have matching keys. In other words, it returns only the rows that have common keys in both dataframes. This is the default join type in PySpark.

A screenshot of a computer

AI-generated content may be incorrect.

**Outer (Full) Join**

An outer join, also known as a full join, returns all rows from both dataframes. If a key is present in one dataframe but not in the other, the missing values are filled with nulls.

**A screenshot of a computer program

AI-generated content may be incorrect.**

**Left Join**

A left join returns all rows from the left dataframe and the matched rows from the right dataframe. If no match is found for a key in the right dataframe, the result will contain null values.

A screenshot of a computer program

AI-generated content may be incorrect.

**Right Join**

A right join returns all rows from the right dataframe and the matched rows from the left dataframe. If no match is found for a key in the left dataframe, the result will contain null values.

A screenshot of a computer

AI-generated content may be incorrect.

**Left Semi Join**

A left semi join returns only the columns from the left dataframe for the rows with matching keys in both dataframes. It is similar to an inner join but only returns the columns from the left dataframe.

A screenshot of a computer

AI-generated content may be incorrect.

**Left Anti Join**

A left anti join returns the rows from the left dataframe that do not have matching keys in the right dataframe. It is the opposite of a left semi join.

A screenshot of a computer

AI-generated content may be incorrect.

**Cross Join**

A cross join, also known as a cartesian join, returns the cartesian product of both dataframes. It combines each row from the left dataframe with each row from the right dataframe.

A screenshot of a computer

AI-generated content may be incorrect.

1. **replace where in pyspark**

The **replaceWhere** option in PySpark is used **with Delta tables** to **dynamically overwrite specific partitions** instead of the entire table.

**Prevents full table overwrites** → Only modifies specific partitions.  
**Improves performance** → Reduces unnecessary data writes.  
**Works with partitioned Delta tables**.

**A white background with colorful text

AI-generated content may be incorrect.**

1. **Partitioning**

Apache Spark is a powerful distributed computing framework designed to process large datasets in parallel across multiple nodes in a cluster. To maximize performance and minimize data movement, Spark divides datasets into partitions that can be processed independently. In this blog post, we'll discuss partitioning and shuffling in PySpark, exploring how these concepts impact the efficiency of your data processing tasks and how to optimize them for your specific use cases.

**Partitioning**

Partitioning is the process of dividing a dataset into smaller, non-overlapping chunks called partitions. Each partition is processed independently on a separate node in the [Spark cluster](https://www.sparkcodehub.com/spark-cluster). Partitioning is crucial for parallel processing, as it allows Spark to distribute data across the cluster and achieve high levels of data locality, minimizing data movement and network overhead.

**Types of Partitioning**

**Default Partitioning (Auto Partitioning)**

* PySpark automatically determines the number of partitions based on data size and cluster settings.
* The default shuffle partition count is 200.

spark.conf.get("spark.sql.shuffle.partitions")

**Manual Partitioning**

You can explicitly specify the number of partitions to control data distribution.

1. repartition(n): Increases Partitions (Expensive)

* Performs full shuffle of data across the cluster.
* Used when you want even distribution.

df = df.repartition(10) # Increase partitions

**2. coalesce(n):** Reduces Partitions (Optimized)

* Avoids full shuffle, merges existing partitions efficiently.
* Used to reduce partitions after transformations.

df = df.coalesce(5)

**Partitioning in File Writes**

Partitioning is useful when writing large datasets.

* Saves data in separate folders based on partition column.
* Improves query speed by filtering specific partitions.

df.write.partitionBy("year", "month").parquet("/path/to/output")

**Partitioning Best Practices**

* Choose an appropriate number of partitions: Too few partitions may lead to underutilization of resources, while too many partitions can cause overhead and slow down processing. A good starting point is to use the number of cores in your cluster, but you should experiment and monitor performance to find the best value for your use case.
* Use domain knowledge: If you have information about the distribution of your data or the expected access patterns, use that knowledge to design an effective partitioning strategy that minimizes data movement and network overhead.

1. **Medallion architecture**

A medallion architecture serves as a data design blueprint tailored for organizing data within a lake house environment. Its primary aim is to enhance the structure and quality of data gradually as it traverses through successive layers of the architecture, progressing from Bronze to Silver to Gold layers.

**Bronze layer**

The Bronze layer serves as the initial landing ground for all data originating from external source systems. Datasets within this layer mirror the structures of the source system tables in their original state, supplemented by extra metadata columns such as load date/time and process ID. The primary emphasis here is on Change Data Capture, enabling historical archiving of the source data, maintaining data lineage, facilitating audit trails, and allowing for reprocessing if necessary, without requiring a fresh read from the source system.

**Silver layer**

The next layer of the lakehouse is the silver layer. Within this layer, data from the bronze layer undergoes a series of operations to a “just-enough” state (which will be discussed in detail later). This prepares the data in the silver layer to offer an encompassing “enterprise view” comprising essential business entities, concepts, and transactions.

**Gold layer**

The last layer of the lakehouse is the gold layer. Data within the Gold layer is typically structured into subject area specific databases, primed for consumption. This layer is dedicated to reporting and employs denormalized, read-optimized data models with minimal joins. It serves as the ultimate stage for applying data transformations and quality rules. Commonly, you will observe the integration of Kimball-style star schema-based data marts within the Gold Layer of the lakehouse.

**SQL**

1. **Difference between rank and dense rank**

In SQL, both RANK and DENSE\_RANK are window functions used to assign rankings to rows within a result set based on specified criteria. However, they differ in how they handle ties and gaps in the ranking sequence.

**Definitions and Key Differences**

* RANK: This function assigns a unique rank to each row within its partition, with gaps in the ranking sequence if there are ties. For example, if two rows tie for the first position, both will receive a rank of 1, and the next row will receive a rank of 3, skipping rank 2.
* DENSE\_RANK: This function also assigns a unique rank to each row within its partition but without gaps in the ranking sequence. If two rows tie for the first position, both will receive a rank of 1, and the next row will receive a rank of 2.

A white background with black text

AI-generated content may be incorrect.

Use Cases

* RANK: Useful when you want to differentiate between tied values distinctly and are okay with gaps in the ranking sequence.
* DENSE\_RANK: Useful when you need a continuous and unbroken sequence of ranks, especially when dealing with tied values

1. **Difference between groupby and window function when to use**

**GROUPBY**

* Purpose: Filters rows before any groupings are made.
* Usage: Applied to individual rows in a table.

SELECT \* FROM employees

WHERE department = 'Sales';

**HAVING Clause**

* Purpose: Filters groups after the GROUP BY clause has been applied.
* Usage: Applied to aggregated data.

SELECT department, COUNT(\*) as employee\_count

FROM employees

GROUP BY department

HAVING COUNT(\*) > 10;

1. **Difference between dropduplicates and distinct**

Both **dropDuplicates()** and **distinct()** are used to remove duplicate rows from a DataFrame, but they have key differences

**distinct() –** Removes Fully Duplicate Rows

* Removes all rows that are completely identical across all columns.
* Works on the entire DataFrame.
* Similar to SELECT DISTINCT in SQL.

**dropDuplicates(["columns"]) –** Removes Duplicates Based on Specific Columns

* Removes duplicates based on selected columns while keeping the first occurrence.
* Other columns may have different values and still be retained.

1. **Window function - row number, rank, dense rank**
2. **ROW\_NUMBER()**: This function assigns a unique sequential number to each row within a window. It's like numbering the rows in order.
3. **RANK()**: The **RANK()** function handles tied values by assigning the same rank to them. However, it may skip subsequent ranks, leaving gaps in the sequence.
4. **DENSE\_RANK()**: Similar to **RANK()**, **DENSE\_RANK()** also handles tied values by assigning the same rank. However, it does not skip ranks, resulting in no gaps in the sequence.
5. **where and having clause**

**WHERE Clause**

The WHERE clause is used to filter records before any groupings are made. It is used to specify conditions on individual rows.

SELECT \*

FROM Employees

WHERE Department = 'Sales';

This query selects all employees who work in the Sales department.

**HAVING Clause**

The HAVING clause is used to filter records after the GROUP BY clause has been applied. It is used to specify conditions on groups.

SELECT Department, COUNT(\*)

FROM Employees

GROUP BY Department

HAVING COUNT(\*) > 10;

This query selects departments that have more than 10 employees.

SELECT Department, AVG(Salary)

FROM Employees

WHERE JobTitle = 'Engineer'

GROUP BY Department

HAVING AVG(Salary) > 70000;

This query first filters employees who are Engineers, then groups them by department, and finally selects departments where the average salary of Engineers is greater than 70,000.

1. **primary and unique key difference**

In relational databases, **primary keys** and **unique keys** are essential for ensuring data integrity and efficient access to records. Both keys provide a guaranteed uniqueness for a column or a set of columns in a table, but they have distinct differences in their usage and constraints.

**Primary Key**

A **primary key** is a column or a combination of columns that uniquely identifies each row in a table. It enforces integrity constraints and does not allow duplicate or NULL values. Each table can have only one primary key. For example, in a Student table, the Roll\_number can be a primary key as it uniquely identifies each student.

A screenshot of a computer code

AI-generated content may be incorrect.

**Features of Primary Key**

* **Uniqueness**: No duplicate rows.
* **Single Key**: Only one primary key per table.
* **Not Null**: Cannot contain NULL values.
* **Clustered Index**: Creates a clustered index by default.
* **Auto Increment**: Supports auto-increment values.

**Unique Key**

A **unique key** also ensures that all values in a column or a set of columns are unique, but it allows one NULL value. A table can have multiple unique keys. For example, in the Student table, the Citizen\_ID can be a unique key as it must be unique for each student but can be NULL if the student does not have a citizen ID.

**A screenshot of a computer program

AI-generated content may be incorrect.**

**Features of Unique Key**

* **Multiple Keys**: More than one unique key per table.
* **Allows NULL**: Can contain one NULL value.
* **Non-Clustered Index**: Creates a non-clustered index by default.
* **Modifiable**: Unique key values can be changed.

**Key Differences**

1. **NULL Values**: Primary keys cannot accept NULL values, while unique keys can accept one NULL value.
2. **Number of Keys**: Only one primary key per table, but multiple unique keys are allowed.
3. **Index Type**: Primary keys create clustered indexes, whereas unique keys create non-clustered indexes.
4. **Auto Increment**: Primary keys support auto-increment, unique keys do not.
5. **Modification**: Primary key values cannot be changed, but unique key values can be modified.

**Conclusion**

Both primary and unique keys are crucial for maintaining data integrity in relational databases. The primary key is ideal for uniquely identifying each record without allowing NULL values, while the unique key is useful for ensuring uniqueness in columns that can contain NULL values. Understanding their differences helps in designing efficient and reliable database schemas

**ADF**   
**19. Until vs for each activity**

In Azure Data Factory (ADF), "Until" activity and "ForEach" activity are both used for looping, but they serve different purposes.

**Until Activity (Conditional Looping)**

* Runs a set of activities in a loop until a condition is met.
* Similar to a while loop in programming.
* Best used when the number of iterations is unknown in advance.

**Example: Run Until a File Arrives**

* Check a storage location every 5 minutes and continue until a file appears.

🔹 **Use Case:**

* Polling (e.g., waiting for an external process to complete).
* Retry mechanisms with timeouts.

**ForEach Activity (Iterative Looping)**

* Loops over an array or list of items and runs activities in parallel or sequentially.
* Similar to a for loop in programming.
* Best used when the number of iterations is known.

**Example: Process Multiple Files**

* Loop through a list of file names and process each file.

**Use Case:**

* Processing multiple files in storage.
* Iterating through database records for transformation.

**GDAI**   
**20. GDAI migration process**

GDP is the centre for strong all the data from all domains. Now all the data are migrating to GDAI platform where unity catalog is enable.

we are using 2 frameworks – MDMF and E- prep

**MDMF-Metadata-Driven Model Framework(source to bronze)**

MDMF consists of three main components

1. Data Ingestion
2. Data Standardization
3. Data Segregation

**Data Ingestion:**

This involves ingesting the data into the system. Getting the data from the source system

**Data Standardization:**

This step includes performing basic transformations, excluding business logic.

**Data Seggregation:**

This involves controlling access to data

**Confidential:**



Highly sensitive data that cannot be accessed by everyone

Even though you (employee) are a part of the organisation, only limited persons (higher officials) with permitted access will be able to access the sensitive information.

**E-prep (bronze to gold):**

Bronze layerdata cleanse and transformed in this framework using python

**21. what is unity catalog**

Unity Catalog is a unified governance and data catalog solution for Databricks, providing:

* Centralized metadata management across workspaces and cloud providers.
* Fine-grained access control (RBAC) at the table, schema, and row level.
* Data lineage tracking for improved auditability and compliance.
* Multi-cloud compatibility (Azure, AWS, GCP).

**Key Features of Unity Catalog**

**1. Centralized Metadata & Governance**

* Manage databases, tables, views, and permissions in one place.
* Works across workspaces (unlike the traditional Hive Metastore, which is workspace-specific).

**2. Fine-Grained Access Control (RBAC)**

* Control access at the Catalog, Schema, Table, Column, and Row levels.
* Uses SQL-based GRANT/DENY commands.

**3. Data Lineage Tracking**

* Automatically tracks data movement across tables, notebooks, and dashboards.
* Helps in auditing and compliance.

**4. Multi-Cloud & Multi-Workspace Support**

* Manage data across AWS, Azure, and GCP in a single catalog.
* Share data between Databricks workspaces.

**5. Delta Sharing for Cross-Organization Access**

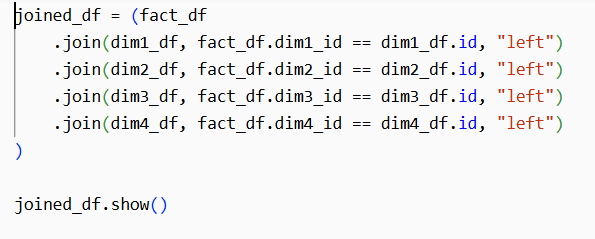
* Securely share Delta tables across different organizations.
* No need to copy data

Unity Catalog introduces **three hierarchical levels**:

|  |  |
| --- | --- |
| **Catalog** | Highest-level namespace (like a database cluster) |
| **Schema** (Database) | Groups tables and views within a catalog |
| **Table** | Stores actual data |

**Queries**

1. How would you join fact and 4-dimension table



1. Get unmatched records from two dataframe

A close-up of a computer screen

AI-generated content may be incorrect.

1. Broadcast join of two dataframe

A close-up of a logo

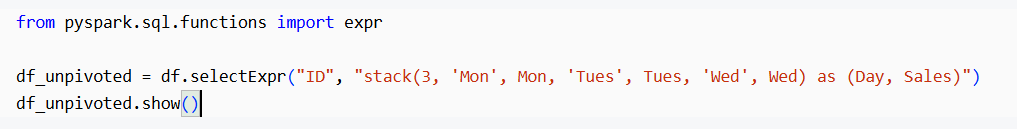
AI-generated content may be incorrect.

1. Replace where
2. Sql query groupby and having

A white background with black text

AI-generated content may be incorrect.

1. Pyspark code – unpivot



1. Top 3 salary from employee table

A close-up of text

AI-generated content may be incorrect.

1. Upsert syntax

