Databricks Data Engineer Professional Master Cheat Sheet

Section 1: Databricks Tooling

Explain how Delta Lake uses the transaction log and cloud object storage to guarantee atomicity and durability

Atomicity and Durability in Delta Lake

- **Transaction Log:** Delta Lake uses a transaction log to record all changes made to the data. This log is an ordered record of every transaction that has been performed on the table. Each transaction is recorded as an atomic unit, meaning that either all changes within the transaction are applied, or none are.
- **Cloud Object Storage:** Delta Lake stores data in cloud object storage (e.g., AWS S3, Azure Blob Storage, Google Cloud Storage). This storage is highly durable and available. Delta Lake leverages the durability of the underlying object storage to ensure that data is not lost in the event of a failure.

How Delta Lake Guarantees Atomicity and Durability:

1. **Atomic Writes:** When a write operation is performed, Delta Lake first writes the new data files to the object storage. Then, it atomically adds a new record to the transaction log that points to these new files. This atomic operation is crucial for ensuring atomicity. If the write operation fails before the transaction log is updated, the changes are not considered part of the table.
2. **Durability through Object Storage:** Because the data files reside in durable cloud object storage, they are protected from data loss due to hardware failures. The transaction log itself is also stored in the object storage, ensuring its durability.
3. **Recovery:** In the event of a failure during a write operation, Delta Lake can use the transaction log to recover the table to a consistent state. It can identify the last successfully completed transaction and revert any incomplete changes.

Describe how Delta Lake's Optimistic Concurrency Control provides isolation, and which transactions might conflict

Optimistic Concurrency Control and Isolation

Delta Lake uses optimistic concurrency control to manage concurrent write operations. This approach assumes that conflicts between transactions are rare.

How Optimistic Concurrency Control Works:

1. **Read Phase:** When a transaction starts, it reads the current version of the table (as indicated by the latest entry in the transaction log).
2. **Write Phase:** The transaction performs its write operations and prepares to commit the changes.

3. **Conflict Check:** Before committing, the transaction checks if the table version it read at the beginning is still the latest version. If another transaction has committed changes in the meantime, a conflict is detected.
4. **Commit or Abort:** If there is no conflict, the transaction commits its changes by adding a new entry to the transaction log. If there is a conflict, the transaction is aborted, and the user is typically notified to retry the operation.

Isolation Levels and Potential Conflicts:

Delta Lake provides snapshot isolation. This means that each transaction reads a consistent snapshot of the data as of the transaction's start time.

Potential Conflicts:

Conflicts can occur when two or more transactions attempt to modify the same data concurrently. For example:

- **Two transactions trying to update the same rows:** If two transactions read the same row and then attempt to update it, a conflict will occur. The second transaction to commit will be aborted.
- **One transaction deleting rows while another is updating them:** If one transaction deletes rows that another transaction is trying to update, a conflict will occur.

Benefits of Optimistic Concurrency Control:

- **High Performance:** In cases where conflicts are rare, optimistic concurrency control offers better performance than pessimistic locking (where resources are locked for the duration of the transaction).
- **Reduced Latency:** Transactions do not need to acquire locks before reading data, reducing latency.

Describe basic functionality of Delta clone.

Delta Clone is a powerful feature in Delta Lake that allows you to create copies of Delta tables at specific versions. This is incredibly useful for various purposes like:

- **Data Migration and Archiving:** Create copies of tables for migration to new systems or for long-term archival.
- **Reproducible Machine Learning:** Replicate specific data versions used for training ML models to ensure reproducibility.
- **Experimentation and Testing:** Create clones to experiment with data transformations without affecting the production table.
- **Disaster Recovery:** Maintain up-to-date clones for quick recovery in case of data loss.

There are two types of clones:

- **Deep Clone:** This creates a full, independent copy of the source table, including both metadata and data files. Changes to the source table will not affect the clone, and vice versa.
- **Shallow Clone:** This creates a new table with the same metadata as the source table but reuses the existing data files. This is faster and cheaper than a deep clone, but changes to the source table's data files will be reflected in the shallow clone.

Apply common Delta Lake indexing optimizations including partitioning, zorder, bloom filters, and file sizes

Delta Lake provides several indexing optimizations to improve query performance:

1. Partitioning

- **Concept:** Dividing a table into smaller parts based on the values of one or more columns (partition keys). This allows queries to read only the relevant partitions, significantly reducing the amount of data scanned.
- **Example:** Partitioning a table of sales data by date or region.
- **Best Practices:**
    - Choose columns with high cardinality and frequently used in query filters.
    - Avoid partitioning on columns with very high cardinality (e.g., unique IDs) as it can lead to a large number of small partitions (partition explosion).

2. Z-Ordering

- **Concept:** A multi-dimensional clustering technique that arranges data within files to keep related data together. This improves data skipping by minimizing the number of files that need to be read.
- **Example:** Z-ordering a table with date and product_id columns.
- **Best Practices:**
    - Choose columns frequently used in query filters, especially equality filters.
    - Z-order on columns with a natural correlation.

3. Bloom Filters

- **Concept:** A space-efficient probabilistic data structure that tests whether an element is a member of a set. In Delta Lake, bloom filters are used to check if a data file contains values that match a query's filter conditions. This helps in skipping irrelevant files.
- **Example:** Creating a bloom filter on a customer_id column.
- **Best Practices:**
    - Suitable for high-cardinality columns with equality filters.
    - Consider the trade-off between filter accuracy and storage space. Larger bloom filters are more accurate but consume more space.

4. File Sizes

- **Concept:** Optimizing the size of data files is crucial for query performance. Small files can lead to excessive metadata overhead and increased I/O operations, while very large files can hinder parallelism.
- **Best Practices:**
    - Aim for file sizes around 1 GB.
    - Use the OPTIMIZE command to compact small files into larger ones.
    - Consider using auto compaction to automatically manage file sizes.

Implement Delta tables optimized for Databricks SQL service

Delta Lake is an open-source storage layer that brings ACID transactions to Apache Spark and big data workloads. It provides several features that enhance the performance and reliability of data pipelines, especially when used with Databricks SQL service. Here's how to implement Delta tables optimized for this environment:

1. Proper Table Design:

- **Schema Design:** Define a clear and efficient schema with appropriate data types. Avoid using generic types like STRING for all columns.
- **Primary Keys:** Define primary keys to enforce data integrity and enable efficient updates and deletes.
- **Data Clustering:** Use Z-Ordering to physically cluster data on disk based on frequently queried columns. This significantly improves query performance by reducing the amount of data scanned.

2. Optimizing Data Layout:

- **File Size:** Aim for files around 1GB in size. Smaller files lead to increased metadata overhead and slower queries. Use the OPTIMIZE command to compact small files into larger ones.
- **Partitioning:** Partition the table based on high-cardinality columns that are frequently used in queries (more on this below).
- **Data Skipping:** Delta Lake automatically collects statistics about data files, enabling it to skip irrelevant files during queries. Ensure this feature is enabled and consider using Z-Ordering to further enhance its effectiveness.

3. Leveraging Delta Lake Features:

- **Vacuuming:** Regularly remove old files that are no longer needed to reduce storage costs and improve query performance.
- **Compaction:** As mentioned earlier, use the OPTIMIZE command to compact small files.
- **Caching:** Utilize Databricks caching mechanisms to store frequently accessed data in memory for faster retrieval.

4. Monitoring and Tuning:

- **Query Profiling:** Use Databricks SQL's query profiling tools to identify performance bottlenecks.
- **Metrics:** Monitor key metrics like query execution time, data scanned, and file sizes to identify areas for improvement.

Contrast different strategies for partitioning data (e.g. identify proper partitioning columns to use)

Partitioning is a crucial technique for optimizing query performance in large datasets. It involves dividing a table into smaller parts based on the values of one or more columns. Here's a contrast of different partitioning strategies:

1. Choosing Partitioning Columns:

- **High Cardinality:** Select columns with a large number of distinct values to distribute data evenly across partitions.
- **Query Patterns:** Choose columns that are frequently used in WHERE clauses to filter data.
- **Data Distribution:** Consider the distribution of data in the column. Avoid columns with skewed data, as this can lead to uneven partition sizes and performance issues.

2. Partitioning Strategies:

- **Range Partitioning:** Divide data into partitions based on ranges of values (e.g., dates, numerical ranges). Suitable for ordered data.
- **List Partitioning:** Explicitly assign values to partitions (e.g., countries, categories). Useful when there's no natural order.
- **Hash Partitioning:** Use a hash function to distribute data across partitions. Ensures even distribution but can make it difficult to query specific partitions.

3. Identifying Proper Partitioning Columns:

- **Example 1: E-commerce Orders**
    - **Good:** order_date (range partitioning), country (list partitioning)
    - **Bad:** order_id (high cardinality but not used for filtering), order_status (low cardinality)
- **Example 2: Sensor Data**

- **Good:** sensor_id (hash partitioning or list partitioning if the number of sensors is small), date (range partitioning)
- **Bad:** temperature (continuous value, would lead to too many partitions)

4. Considerations:

- **Over-partitioning:** Creating too many small partitions can lead to increased metadata overhead and slower queries.
- **Under-partitioning:** Creating too few large partitions can negate the benefits of partitioning.
- **Data Skew:** Uneven data distribution can lead to some partitions being much larger than others, causing performance bottlenecks.

Section 2: Data Processing (Batch processing, Incremental processing, and Optimization)

Describe and distinguish partition hints: coalesce, repartition, repartition by range, and Rebalance

Partition hints in Databricks allow you to control how data is physically partitioned across the cluster. This can significantly impact query performance and resource utilization. Here's a breakdown of the different partition hints:

1. coalesce

- **Purpose:** Reduces the number of partitions.
- **How it works:** Combines existing partitions without shuffling the data. This is a fast operation but can lead to unevenly sized partitions if the original partitions were skewed.
- **Use case:** When you have too many small partitions and want to reduce the overhead of managing them.

2. repartition

- **Purpose:** Increases or decreases the number of partitions and evenly distributes data across them.
- **How it works:** Performs a full shuffle of the data, ensuring even distribution but incurring significant overhead.
- **Use case:** When you need to redistribute data evenly or change the number of partitions for subsequent operations.

3. repartition by range

- **Purpose:** Partitions data based on a specified column or columns, ensuring that rows with similar values are in the same partition.
- **How it works:** Samples the data to determine partition boundaries and then shuffles the data accordingly.
- **Use case:** When you have range-based queries and want to improve performance by keeping relevant data together.

4. rebalance

- **Purpose:** Balances the size of partitions to avoid skew and improve parallelism.
- **How it works:** Analyzes partition sizes and redistributes data to create more evenly sized partitions.
- **Use case:** When you have skewed data or encounter performance issues due to uneven partition sizes.

Contrast different strategies for partitioning data (e.g. identify proper partitioning column to use)

Choosing the right partitioning strategy is crucial for optimizing query performance. Here are some key considerations:

1. Identify proper partitioning column

- **Cardinality:** Choose columns with low to moderate cardinality (i.e., a limited number of distinct values). High-cardinality columns can lead to too many small partitions.
- **Query patterns:** Select columns frequently used in query filters or joins. This allows Databricks to prune unnecessary partitions.
- **Data distribution:** Consider the distribution of values in the column. Avoid columns with highly skewed distributions, as this can lead to uneven partition sizes.

2. Partitioning techniques

- **Range partitioning:** Divide data into partitions based on ranges of values in a column. Useful for range-based queries.
- **Hash partitioning:** Distribute data evenly across partitions based on a hash function. Useful for distributing data randomly and ensuring even partition sizes.
- **List partitioning:** Assign partitions based on specific lists of values in a column. Useful for handling specific data subsets.

Additional Tips

- **Start with a small number of partitions and increase if necessary.**
- **Monitor partition sizes and adjust partitioning strategy if needed.**
- **Consider using Delta Lake, which provides additional partitioning and data skipping capabilities.**
- **Use EXPLAIN to understand how Databricks is executing your queries and identify potential partitioning issues.**

Articulate how to write Pyspark dataframes to disk while manually controlling the size of individual part-files.

Spark's default behavior when writing DataFrames to disk (e.g., in Parquet format) is to create a number of part-files based on the number of partitions in the DataFrame. This can lead to issues like:

- **Small file problem:** Too many small files can negatively impact read performance.
- **Uneven file sizes:** Partitions might not be evenly sized, resulting in some files being much larger than others.

To manually control the size of individual part-files, you can use the following strategies:

1. Repartitioning:

- Use the repartition() transformation to explicitly control the number of partitions before writing the DataFrame.
- The number of partitions directly influences the number of output files.
- **Example:**

Python

```
df.repartition(10).write.parquet("path/to/output") # Creates 10 part-files
```

- **Limitation:** repartition() performs a full shuffle of the data, which can be expensive.

2. Coalescing:

- Use the coalesce() transformation to reduce the number of partitions.
- coalesce() tries to avoid a full shuffle if possible, making it more efficient than repartition() for reducing partitions.
- **Example:**

Python

```
df.coalesce(5).write.parquet("path/to/output") # Reduces to 5 part-files if possible
```

- **Limitation:** coalesce() is less effective for increasing the number of partitions.

3. Using maxRecordsPerFile (for CSV):

- When writing to CSV format, you can use the maxRecordsPerFile option to limit the number of records in each file.

- **Example:**

Python

```
df.write.option("maxRecordsPerFile", 10000).csv("path/to/output")
```

4. Estimating and Adjusting Partitions:

- For more precise control over file sizes, you can estimate the size of the output data and calculate the desired number of partitions.
- **Steps:**
    1. Estimate the size of the DataFrame in bytes.
    2. Divide the estimated size by the desired file size to get the approximate number of partitions.
    3. Use repartition() with the calculated number of partitions.
- **Example:**

Python

```
estimated_size = ... # Estimate DataFrame size in bytes

desired_file_size = 128 * 1024 * 1024 # 128 MB

num_partitions = int(estimated_size / desired_file_size)

df.repartition(num_partitions).write.parquet("path/to/output")
```

5. Using RepartiPy:

- RepartiPy is a third-party library that simplifies the process of controlling part-file sizes.
- It provides functions to estimate DataFrame size and automatically calculate the desired number of partitions.
- **Example:**

Python

```
import repartipy

with repartipy.SizeEstimator(spark=spark, df=df) as se:
    desired_partition_count =
se.get_desired_partition_count(desired_partition_size_in_bytes=1073741824) # 1 GB
    se.reproduce().repartition(desired_partition_count).write.parquet("path/to/output")
```

Choosing the Right Strategy:

- For simple cases where you just need to reduce the number of files, coalesce() is usually the best option.
- If you need to increase the number of files or have precise control over file sizes, repartition() or the estimation method is necessary.

- For complex scenarios or when you want to avoid manual calculations, RepartiPy can be very helpful.

Articulate multiple strategies for updating 1+ records in a spark table (Type 1)

In Spark, updating records in a traditional sense (in-place modification) is not directly supported due to its immutable data model. Instead, you need to rewrite the affected partitions or the entire table. Here are several strategies for performing Type 1 updates (overwriting existing records):

1. Using merge (Delta Lake):

- If you're using Delta Lake, the merge operation is the most efficient and recommended way to perform updates.
- It allows you to specify conditions for matching records and update only the matching rows.
- **Example:**

Python

```
from delta.tables import DeltaTable


delta_table = DeltaTable.forName(spark, "my_table")
updates_df = spark.createDataFrame([(1, "new_value")], ["id", "value"])


delta_table.alias("old") \
  .merge(updates_df.alias("updates"), "old.id = updates.id") \
  .whenMatchedUpdate(set={"value": "updates.value"}) \
  .execute()
```

2. Overwriting Partitions:

- If your table is partitioned, you can overwrite only the partitions containing the records to be updated.
- This is more efficient than rewriting the entire table if the updates are limited to a small number of partitions.
- **Steps:**
    1. Identify the partitions containing the records to be updated.
    2. Read the data from the affected partitions.
    3. Apply the updates to the DataFrame.
    4. Overwrite the affected partitions with the updated data.
- **Example:**

Python

```
partition_values = ["value1", "value2"]

df = spark.read.table("my_table")

updated_df = df.filter(~col("partition_column").isin(partition_values)) \

   .union(updated_data) # updated_data contains the updated records


updated_df.write.mode("overwrite").partitionBy("partition_column").saveAsTable("my_table")
```

3. Rewriting the Entire Table:

- If the updates are spread across many partitions or the table is not partitioned, you might need to rewrite the entire table.
- **Steps:**
    1. Read the entire table into a DataFrame.
    2. Apply the updates to the DataFrame.
    3. Overwrite the table with the updated DataFrame.
- **Example:**

Python

```
df = spark.read.table("my_table")

updated_df = df.join(updates_df, "id", "left").withColumn("value", when(col("id").isNotNull(), updates_df["value"]).otherwise(df["value"]))

updated_df.write.mode("overwrite").saveAsTable("my_table")
```

4. Using SQL UPDATE (with caution):

- Spark SQL supports the UPDATE statement, but it's important to understand its behavior.
- Under the hood, it performs a rewrite of the affected data, similar to the strategies mentioned above.
- **Example:**

SQL

```
UPDATE my_table SET value = 'new_value' WHERE id = 1
```

- **Caution:** Using UPDATE directly can be less efficient than using the DataFrame API or merge (with Delta Lake) for complex updates.

Choosing the Right Strategy:

- For Delta Lake tables, merge is the most efficient and recommended approach.
- For partitioned tables with updates limited to a few partitions, overwriting partitions is a good option.

- If updates are widespread or the table is not partitioned, rewriting the entire table might be necessary.
- Use SQL UPDATE with caution and consider its performance implications.

Implement common design patterns unlocked by Structured Streaming and Delta Lake.

Structured Streaming and Delta Lake together provide a powerful platform for building robust and efficient streaming data pipelines. Here are some common design patterns they unlock:

1. Lambda Architecture with Delta Lake as the Serving Layer:

- **Pwroblem:** Traditional Lambda Architecture maintains two separate systems for batch and speed layers, leading to complexity.
- **Solution:** Delta Lake acts as the unified serving layer, storing both batch and streaming data. Structured Streaming continuously updates Delta tables, which are then queried for real-time insights. This simplifies the architecture and ensures data consistency.

2. Incremental Data Processing with Delta Lake:

- **Problem:** Processing large datasets repeatedly is inefficient.
- **Solution:** Delta Lake's change data capture (CDC) capabilities track changes (inserts, updates, deletes) in source data. Structured Streaming reads these changes incrementally and applies them to downstream tables, reducing processing time and resource consumption.

3. Stream-Static Joins for Enrichment:

- **Problem:** Streaming data often needs to be enriched with static data (e.g., customer demographics, product catalogs).
- **Solution:** Structured Streaming can join streaming data with static data stored in Delta Lake. This allows for real-time enrichment and contextualization of streaming data.

4. Exactly-Once Processing with Delta Lake:

- **Problem:** Ensuring that each record is processed exactly once in a streaming pipeline is challenging.
- **Solution:** Delta Lake's ACID transactions and Structured Streaming's fault tolerance mechanisms guarantee exactly-once processing, even in the face of failures.

5. Data Deduplication with Structured Streaming and Delta Lake:

- **Problem:** Streaming data can contain duplicates, especially in scenarios with retries or failures.
- **Solution:** Structured Streaming can use stateful operations to track processed records and filter out duplicates. Delta Lake can then be used to store the deduplicated data reliably.

Explore and tune state information using stream-static joins and Delta Lake

Stream-static joins combine a continuous stream of data with a static dataset. In the context of Databricks and Delta Lake:

- The **stream** is typically ingested using Structured Streaming from sources like Kafka or cloud storage.
- The **static dataset** is stored as a Delta table, providing efficient querying and updates.

Exploring and tuning state information in this context involves:

- **Understanding State Management in Structured Streaming:** Structured Streaming maintains state information for operations like aggregations, joins, and watermarking. This state is crucial for correct processing but can also consume significant resources.
- **Optimizing Join Operations:**
    - **Broadcast Joins:** If the static Delta table is small enough, it can be broadcasted to all executors, avoiding shuffles and improving performance.
    - **Partitioning:** Properly partitioning both the stream and the static table can minimize data shuffling during joins.
- **Managing State Size:**
    - **Watermarking:** Using watermarks to define how late data can arrive helps to clean up old state and reduce memory usage.
    - **State Store Configuration:** Tuning the state store (e.g., using RocksDB) can improve performance and scalability.
- **Monitoring State:** Databricks provides tools to monitor state size and usage, allowing you to identify potential bottlenecks and optimize your queries.

Delta Lake's role in state management:

- **Efficient Data Access:** Delta Lake's optimized file format and indexing capabilities ensure fast access to the static data for joins.
- **Data Updates:** Delta Lake allows for efficient updates to the static data, ensuring that the stream-static joins use the latest information.
- **Data Versioning:** Delta Lake's versioning allows you to track changes to the static data and even perform time travel queries for debugging or analysis.

Implement stream-static joins

In Spark Structured Streaming, a stream-static join involves joining a continuous stream of data with a static dataset (a DataFrame or table that doesn't change). This is useful for enriching streaming data with reference information.

- **How it works:** Spark performs the join for each micro-batch of the stream against the entire static dataset. This means the static data must fit comfortably in memory on the executors.
- **Use Cases:**
    - Joining a stream of transaction data with a customer database to add customer details.
    - Enriching log data with metadata from a configuration table.

- Adding geographic information to a stream of location updates using a static location database.

- **Implementation:**

Python

```
from pyspark.sql.functions import *
from pyspark.sql.types import *


# Static DataFrame (e.g., loaded from a table or Parquet)
static_df = spark.read.table("customer_table")


# Streaming DataFrame (e.g., reading from Kafka)
streaming_df = spark.readStream \
    .format("kafka") \
    .option("kafka.bootstrap.servers", "your_kafka_servers") \
    .option("subscribe", "transactions_topic") \
    .load()


# Define schema for the streaming data (important for correct parsing)
schema = StructType([
    StructField("transaction_id", IntegerType()),
    StructField("customer_id", IntegerType()),
    StructField("amount", DoubleType())
])
streaming_df = streaming_df.select(from_json(col("value").cast("string"),
schema).alias("data")).select("data.*")
# Perform the stream-static join
joined_df = streaming_df.join(static_df, "customer_id", "inner") # or "left", "right", etc.


# Process the joined data (e.g., write to a sink)
query = joined_df.writeStream \
    .outputMode("append") \
    .format("delta") \
```

```
    .option("checkpointLocation", "/path/to/checkpoint") \
    .start("/path/to/output")
query.awaitTermination()
```

- **Key Considerations:**
    - **Size of the Static Data:** The static dataset must fit in memory on the executors. If it's too large, consider using a stream-stream join (which is more complex) or pre-joining the data offline.
    - **Updates to Static Data:** If the static data changes frequently, you'll need to reload it periodically. This can be done using a trigger (e.g., once or a time-based trigger) to refresh the DataFrame. However, this isn't truly real-time. For real-time updates to reference data, consider using a key-value store or a change data capture (CDC) mechanism.
    - **Join Type:** Choose the appropriate join type (inner, left, right, full) based on your requirements.

Implement necessary logic for deduplication using Spark Structured Streaming

Deduplication is the process of removing duplicate records from a dataset. In the context of Spark Structured Streaming, this means ensuring that each record is processed only once, even if it appears multiple times in the input stream. This is crucial for maintaining data accuracy and consistency in streaming applications.

Here's how to implement deduplication using Spark Structured Streaming:

1. Using dropDuplicates() with Watermarking:

- This is the simplest approach for basic deduplication based on specific columns.
- It requires defining a watermark to handle late-arriving data. The watermark specifies the maximum delay expected for late data, allowing Spark to maintain state for deduplication within that window.

Python

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

# Assuming 'df' is your streaming DataFrame with an event time column 'event_time'
watermark_df = df.withWatermark("event_time", "10 seconds") # Watermark of 10 seconds

dedup_df = watermark_df.dropDuplicates(["id", "event_time"]) # Deduplication based on 'id' and 'event_time'
```

Write the deduplicated data to a sink

dedup_df.writeStream.format("delta").outputMode("append").start("path/to/delta/table")

- **Explanation:**
    - withWatermark("event_time", "10 seconds") sets the watermark on the event_time column, allowing for 10 seconds of late data.
    - dropDuplicates(["id", "event_time"]) removes duplicate records based on the id and event_time columns within the watermark window.

2. Using mapGroupsWithState() for Complex Deduplication:

- This approach provides more flexibility for complex deduplication logic, such as:
    - Deduplication based on multiple criteria or fuzzy matching.
    - Keeping the latest record based on a timestamp.
    - Handling deduplication across longer time windows.

Python

```
from pyspark.sql.functions import *
from pyspark.sql.streaming import GroupStateTimeout


def deduplicate_function(key, iterator, state):
  if state.hasTimedOut():
    return []


  if state.exists:
    previous_record = state.get
    current_record = list(iterator)[0] # Assuming only one record per key per batch
    if current_record["event_time"] > previous_record["event_time"]:
      state.update(current_record)
      return [current_record]
    else:
      return []
  else:
    current_record = list(iterator)[0]
    state.update(current_record)
    return [current_record]
```

```
dedup_df = df.groupBy("id").mapGroupsWithState(
   deduplicate_function,
   outputMode="append",
   timeoutConf=GroupStateTimeout.NoTimeout
)
dedup_df.writeStream.format("delta").outputMode("append").start("path/to/delta/table")
```

- **Explanation:**
    - groupBy("id") groups the data by the id column.
    - mapGroupsWithState() applies the deduplicate_function to each group, maintaining state across batches.
    - The deduplicate_function compares the event_time of the current record with the previous record in the state and updates the state with the latest record.

Enable CDF on Delta Lake tables and re-design data processing steps to process CDC output instead of incremental feed from normal Structured Streaming read

Change Data Capture (CDC) is a technique for tracking changes to data in a database. Delta Lake's Change Data Feed (CDF) feature provides a way to efficiently access these changes, enabling more efficient and robust data processing pipelines.

Enabling CDF on Delta Lake tables:

- CDF is enabled at the table level when creating a Delta table or can be enabled on existing tables.

Python

```
# Enable CDF when creating a Delta table
df.write.format("delta").option("delta.enableChangeDataCapture",
"true").save("path/to/delta/table")


# Enable CDF on an existing Delta table
spark.sql("ALTER TABLE delta.`path/to/delta/table` SET TBLPROPERTIES
('delta.enableChangeDataCapture' = 'true')")
```

Processing CDC output:

- Instead of reading the entire Delta table or using incremental reads from Structured Streaming, you can now read the CDC output to process only the changes.

Python

```
# Read the changes from a specific version

cdf_df = spark.read.format("delta").option("readChangeFeed", "true").option("startingVersion", 1).table("path/to/delta/table")


# Read the changes from a specific timestamp

cdf_df = spark.read.format("delta").option("readChangeFeed", "true").option("startingTimestamp", "2024-07-26T10:00:00Z").table("path/to/delta/table")


# Process the changes

cdf_df.write.format("delta").mode("append").save("path/to/another/delta/table")
```

- **Key improvements with CDF:**
    - **Efficiency:** Process only the changed data, reducing processing time and resource consumption.
    - **Simplicity:** No need for complex incremental read logic or maintaining offsets.
    - **Reliability:** Ensures that all changes are captured and processed, even in case of failures.

Redesigning data processing steps for CDC:

- By using CDF, you can redesign your data processing pipelines to be more efficient and robust.
- Instead of processing the entire dataset or using incremental reads, you can now focus on processing only the changes.
- This can significantly improve the performance of your pipelines, especially for large datasets with frequent updates.

Example:

- Instead of reading the entire orders table every hour to find new orders, you can use CDF to read only the changes that occurred in the last hour.
- This will significantly reduce the amount of data processed and improve the performance of your pipeline.

Leverage CDF to easily propagate deletes

What is CDF? Delta Lake's Change Data Feed (CDF) provides a record of every change made to a Delta table, including inserts, updates, and *deletes*. These changes are captured in a structured format, making it easy to track and propagate them to downstream systems.

Why is it important for propagating deletes? Traditionally, propagating deletes to downstream systems (e.g., data warehouses, search indexes, caches) was complex. You'd often have to implement complex logic involving joins, snapshots, or timestamps. CDF simplifies this significantly.

How does it work for deletes? When a row is deleted from a Delta table, CDF records a special "delete" event along with the deleted row's identifying information (e.g., primary key). Downstream systems can then consume this feed and apply the corresponding deletes in their own data stores.

Example Scenario: Imagine you have an e-commerce platform using Delta Lake to store customer orders. You have a downstream data warehouse used for reporting. When a customer cancels an order (resulting in a delete in the Delta table), CDF captures this delete. Your data pipeline can then read the CDF, identify the deleted order IDs, and execute corresponding delete statements in the data warehouse, keeping the two systems synchronized.

Benefits:

- **Simplified Delete Propagation:** Eliminates complex logic for tracking and applying deletes downstream.
- **Near Real-Time Synchronization:** Enables faster propagation of deletes, keeping downstream systems more up-to-date.
- **Reduced Development Effort:** Streamlines data pipeline development.
- **Improved Data Consistency:** Ensures data consistency across different systems.

How to enable CDF: You enable CDF at the table level using the delta.enableChangeDataFeed table property:

SQL

CREATE TABLE my_table (...)

TBLPROPERTIES ('delta.enableChangeDataFeed' = 'true');

Reading CDF: You can read the CDF using Spark SQL:

SQL

SELECT * FROM table_changes('my_table', 0, 100); -- Reads changes from version 0 to 100

SELECT * FROM table_changes('my_table', '2023-10-26 00:00:00', '2023-10-27 00:00:0

Demonstrate how proper partitioning of data allows for simple archiving or deletion of data

What is partitioning? Partitioning involves dividing a table into smaller parts based on the values of one or more columns (partition keys). This physically organizes the data into separate directories on the storage system.

Why is it important for archiving/deletion? When you need to archive or delete a large subset of data (e.g., data older than a certain date), partitioning allows you to operate on entire partitions rather than scanning and filtering the whole table. This is significantly more efficient.

How does it work for archiving? If you partition your data by date (e.g., year, month, day), you can archive older data by simply moving the corresponding partition directories to a separate storage location (e.g., cold storage).

How does it work for deletion? Similarly, you can delete data by dropping the corresponding partition directories. This is a metadata operation, making it extremely fast compared to deleting individual rows.

Example Scenario: You have a log table partitioned by year and month. To archive logs from 2022, you can simply move the directories corresponding to year=2022 to an archive location. To delete logs from January 2023, you can drop the partition year=2023/month=01.

Benefits:

- **Faster Archiving/Deletion:** Significantly reduces the time and resources required for archiving or deleting large datasets.
- **Improved Query Performance:** Queries that filter on partition keys can avoid scanning irrelevant partitions, leading to faster execution.
- **Simplified Data Management:** Makes it easier to manage and organize large datasets.

Example of partitioning in table creation:

SQL

```
CREATE TABLE my_logs (
  timestamp TIMESTAMP,
  message STRING,
  -- other columns
)
PARTITIONED BY (year INT, month INT)
USING DELTA
LOCATION 'dbfs:/path/to/my_logs';
```

Example of dropping a partition:

SQL

```
ALTER TABLE my_logs DROP PARTITION (year = 2023, month = 1);
```

Articulate, how “smalls” (tiny files, scanning overhead, over partitioning, etc) induce performance problems into Spark queries

- **Scanning Overhead:** Spark is designed to process large files efficiently. When dealing with numerous small files, the system spends a disproportionate amount of time on file metadata operations (opening, closing, listing files) rather than actual data processing. This overhead becomes a bottleneck, especially with cloud storage like S3 or Azure Blob Storage, where each file operation involves network communication.
- **Task Management Overhead:** Spark distributes work into tasks, each processing a partition of data. Ideally, each task should handle a substantial chunk of data for optimal parallelism. With small files, you end up with many small tasks, increasing scheduling and management overhead. The cluster spends more time managing tasks than processing data.

- **Inefficient I/O:** Small files lead to more random I/O operations, which are slower than sequential I/O. This is particularly relevant for disk-based storage.

Related Concepts

- **Over-partitioning:** This occurs when you have too many partitions relative to the data size. While partitioning is essential for parallelism, excessive partitioning creates similar issues to small files:
    - Increased task management overhead.
    - Small tasks that don't fully utilize cluster resources.
    - Increased shuffle overhead if data needs to be redistributed between partitions during operations like joins or aggregations.
- **File Size and Partitioning:** The ideal file size in Spark is often cited as around 128MB (matching the HDFS block size). This provides a good balance between I/O efficiency and parallelism. When partitioning data, aim for partitions that result in files of this approximate size.

Solutions

- **Compaction:** Combine small files into larger ones. This can be done using:
    - OPTIMIZE command in Delta Lake.
    - Hadoop's FileSystem API or command-line tools.
- **Repartitoning:** Adjust the number of partitions to better suit the data size. Use repartition() or coalesce() transformations in Spark. repartition() shuffles data, creating an even distribution but is more expensive. coalesce() minimizes data shuffling but may result in uneven partition sizes.
- **Writing Data Efficiently:** When writing data, avoid creating small files in the first place. For example, when using structured streaming, configure appropriate trigger intervals and output modes to control file output.

Example

Imagine you have a dataset of 1GB.

- **Scenario 1: 10 files of 100MB each:** Relatively efficient. Spark can create 10 tasks, each processing a reasonable amount of data.
- **Scenario 2: 1000 files of 1MB each:** Inefficient. Spark has to manage 1000 tasks, leading to significant overhead.

Section 3: Data Modeling

Describe the objective of data transformations during promotion from bronze to silver

In a Databricks Lakehouse architecture, data is typically organized in a multi-layered approach known as the "medallion architecture," with Bronze, Silver, and Gold layers. Each layer serves a specific purpose in the data refinement process.

Bronze Layer: This is the raw data layer, where data is ingested directly from source systems with minimal transformation. The primary objective here is to land the data as is, preserving its original form for auditing and historical purposes.

Silver Layer: This layer contains data that has been transformed and refined for analytical use. The key objectives of data transformations during the promotion from Bronze to Silver are:

- **Data Quality:**
  - **Cleaning:** Removing inconsistencies, errors, and null values.
  - **Deduplication:** Eliminating duplicate records.
  - **Validation:** Ensuring data conforms to defined rules and constraints.
- **Data Integration:**
  - **Standardization:** Converting data to consistent formats and units.
  - **Schema Enforcement:** Applying a consistent schema to the data.
  - **Joining and Merging:** Combining data from different sources.
- **Data Enrichment:**
  - **Adding context:** Deriving new attributes or adding metadata to enhance the data.
  - **Data type conversion:** Converting data into appropriate data types for analysis.
- **Performance Optimization:**
  - **Partitioning and Bucketing:** Organizing data for efficient querying.
  - **Data format optimization:** Converting data to efficient storage formats like Parquet.

By achieving these objectives, the Silver layer provides a reliable and consistent source of data for downstream analytics, reporting, and machine learning.

Discuss how Change Data Feed (CDF) addresses past difficulties propagating updates and deletes within Lakehouse architecture

Change Data Feed (CDF) is a feature in Databricks that tracks changes to Delta tables at the row level. It records inserts, updates, and deletes, along with metadata about each change, such as timestamps and version numbers.

Past Difficulties in Propagating Updates and Deletes:

Traditionally, propagating updates and deletes in a Lakehouse architecture posed challenges:

- **Identifying Changes:** Determining which records had changed in the source system required complex logic and often involved comparing snapshots of data.
- **Propagating Deletes:** Deleting records in the data lake required careful handling to avoid data inconsistencies.
- **Performance Overhead:** Processing large volumes of data to identify and propagate changes could be computationally expensive.

How CDF Addresses These Difficulties:

CDF simplifies the process of propagating updates and deletes by:

- **Providing a Stream of Changes:** CDF provides a structured stream of changes that can be easily consumed by downstream processes.
- **Capturing All Changes:** CDF captures all types of changes, including inserts, updates, and deletes, ensuring data consistency.
- **Improving Performance:** CDF is designed to be efficient, minimizing the performance overhead of change data capture.

By using CDF, data engineers can easily build incremental data pipelines that efficiently propagate changes from the Bronze layer to the Silver and Gold layers, ensuring that the data in the Lakehouse is always up-to-date.

Apply Delta Lake clone to learn how shallow and deep clone interact with source/target tables.

Delta Lake's clone feature allows you to create copies of tables, which can be very useful for various purposes like testing, experimentation, and disaster recovery. There are two types of clones:

1. Shallow Clone:

- **Metadata Only:** A shallow clone copies only the metadata of the source table (schema, partitioning, etc.) but **not the data files**. It points to the data files in the source table's location.
- **Faster and Cheaper:** Creating a shallow clone is very fast and cost-effective as it doesn't involve copying large amounts of data.

- **Dependencies:** Shallow clones are dependent on the source table. If you delete data files from the source table (e.g., using VACUUM), the shallow clone will be affected and may become unusable.
- **Use Cases:** Suitable for short-lived operations like testing a new query or experimenting with schema changes without affecting the original data.

2. Deep Clone:

- **Full Copy:** A deep clone copies both the metadata and **all the data files** from the source table to a new location.
- **Independent:** A deep clone is completely independent of the source table. Changes to the source table will not affect the deep clone, and vice versa.
- **More Time and Storage:** Creating a deep clone takes longer and requires more storage space as it involves copying all the data.
- **Use Cases:** Ideal for creating backups, setting up development or testing environments, and performing operations that might modify the data without impacting the original table.

Interactions with Source/Target Tables:

- **Shallow Clone:** Changes to the data in the source table are immediately reflected in the shallow clone because they share the same data files. Deleting data from the source can break the shallow clone.
- **Deep Clone:** Changes to either the source or the deep clone do not affect the other. They are completely isolated.

Example:

SQL

-- Create a shallow clone

CREATE TABLE shallow_clone_table SHALLOW CLONE source_table;

-- Create a deep clone

CREATE TABLE deep_clone_table DEEP CLONE source_table;

Design a multiplex bronze table to avoid common pitfalls when trying to productionalize streaming workloads.

In a typical data lakehouse architecture, the bronze layer is the first landing zone for raw data ingested from various sources. When dealing with streaming workloads, a common pitfall is directly writing streaming data to individual bronze tables, which can lead to:

- **Small Files:** Continuous streaming can generate a large number of small files, impacting query performance.
- **Schema Evolution Challenges:** Handling schema changes in multiple tables can become complex.

- **Increased Management Overhead:** Managing numerous small tables adds to operational complexity.

To avoid these issues, a **multiplex bronze table** can be used.

What is a Multiplex Bronze Table?

A multiplex bronze table is a single, partitioned table that stores raw data from multiple sources. It uses additional columns to identify the source and other relevant metadata.

Key Design Elements:

- **Source Identifier:** A column (e.g., source_system) to identify the origin of the data.
- **Ingestion Timestamp:** A column (e.g., ingestion_time) to track when the data was ingested.
- **Partitioning:** Partition the table by source_system and/or ingestion_date to improve query performance.
- **Data Column:** A column (e.g., payload) to store the raw data, often in JSON or Avro format to preserve the original schema.

Benefits:

- **Reduced Small Files:** By writing all data to a single table, you can leverage Delta Lake's optimizations to avoid small files.
- **Simplified Schema Evolution:** Schema changes can be handled within the payload column or by adding new columns as needed.
- **Lower Management Overhead:** Managing one table is much simpler than managing many individual tables.
- **Improved Query Performance:** Partitioning and optimized file sizes lead to faster queries.

Example:

Let's say you have streaming data from two sources: orders and customers.

SQL

```
CREATE TABLE bronze_multiplex (
  source_system STRING,
  ingestion_time TIMESTAMP,
  ingestion_date DATE GENERATED ALWAYS AS (DATE(ingestion_time)),
  payload STRING -- JSON or Avro data
)
PARTITIONED BY (source_system, ingestion_date);
```

When ingesting data:

Python

```
# Example using PySpark Structured Streaming
query = (
  spark.readStream.format("kafka")
  # ... Kafka configuration ...
  .select(
    lit("orders").alias("source_system"),
    current_timestamp().alias("ingestion_time"),
    col("value").cast("string").alias("payload") # Assuming value is JSON
  )
  .writeStream.format("delta")
  .option("checkpointLocation", "...")
  .table("bronze_multiplex")
)
```

Implement best practices when streaming data from multiplex bronze tables.

What are Multiplex Bronze Tables? In a Medallion architecture (Bronze, Silver, Gold), the bronze layer is the raw, ingested data. "Multiplex" implies that a single bronze table might contain data from multiple sources or different event types, often distinguished by a source identifier or event type column. This is common for efficiency in ingestion but requires careful handling downstream.

Streaming from Bronze: Instead of batch processing, streaming allows for near real-time processing of data as it lands in the bronze layer. This is crucial for timely insights and actions. Databricks Structured Streaming is the primary tool for this.

Best Practices for Streaming from Multiplex Bronze:

- **Schema Enforcement:** Even in bronze, having a well-defined schema (or evolving schema) is vital. This prevents data quality issues from propagating downstream. Use schema inference cautiously and prefer explicit schema definition.
- **Source/Event Type Identification:** Ensure a reliable way to differentiate data from different sources within the multiplex table. This is usually a column in the data itself (e.g., source_system, event_type).
- **Partitioning/Bucketing:** Partitioning the bronze table by the source or event type can dramatically improve query performance, especially in streaming scenarios. Bucketing can further optimize data access.
- **Change Data Capture (CDC):** If your source systems support CDC, leverage it. This provides only the changes to the data (inserts, updates, deletes), making streaming more efficient. If CDC isn't available, consider using timestamps or other mechanisms to identify new or changed data.

- **Checkpointing:** Structured Streaming relies on checkpoints to ensure fault tolerance. Configure checkpoints appropriately (e.g., to cloud storage) to recover from failures without data loss.
- **Watermarking:** For aggregations and joins in streaming, watermarking is essential to handle late-arriving data. It defines a threshold for how long the stream should wait for late data before completing an aggregation.

Apply incremental processing, quality enforcement, and deduplication to process data from bronze to silver

- **What is Incremental Processing?** Instead of reprocessing the entire bronze table every time, incremental processing only processes new or changed data since the last processing run. This significantly reduces processing time and cost.
- **How to Implement Incremental Processing in Databricks:**
    - **Using Trigger.AvailableNow:** This trigger in Structured Streaming processes all available data at the time of execution and then stops. It's suitable for micro-batch processing.
    - **Using Trigger.Once:** This trigger processes all available data and then stops. It is suitable for initial loads or backfills.
    - **Using Trigger.Continuous (with caution):** This trigger processes data continuously as it arrives. While providing the lowest latency, it requires careful resource management and is often less efficient than micro-batching for many use cases.
    - **Delta Lake's Change Data Feed:** If using Delta Lake for your bronze and silver tables (highly recommended), the Change Data Feed efficiently tracks changes to the table (inserts, updates, deletes). This simplifies incremental processing significantly. You can directly query the change data feed to process only the changes.
    - **Tracking Processed Offsets/Timestamps:** If not using Delta Lake's Change Data Feed, you'll need to maintain a record of the last processed offset or timestamp. This can be stored in a separate table or using checkpointing within Structured Streaming.

3. Quality Enforcement (Bronze to Silver)

- **Why is Quality Enforcement Important?** The silver layer should contain clean, validated data ready for business use. Quality checks prevent bad data from polluting downstream processes.
- **How to Enforce Quality:**
    - **Schema Validation:** Enforce strict schemas in the silver layer. Any data that doesn't conform should be rejected or handled separately (e.g., moved to a quarantine area).
    - **Data Type Checks:** Ensure data types are correct (e.g., numeric values are actually numbers, dates are valid dates).
    - **Null Checks:** Identify and handle null values according to business rules.

- **Business Rule Validation:** Implement checks for domain-specific rules (e.g., valid product codes, acceptable ranges for measurements).
- **Expectations (using expect clause in Delta Live Tables or similar):** Define expectations on your data. If an expectation fails, you can choose to fail the pipeline, warn, or drop the violating records. This is a powerful way to enforce data quality.
- **Data Profiling:** Before processing to silver, profile the data in bronze to identify potential quality issues. This can be done using Databricks utilities or by querying the data.

4. Deduplication (Bronze to Silver)

- **Why Deduplication?** Source systems can sometimes send duplicate data. Deduplication ensures data consistency and accuracy in the silver layer.
- **How to Deduplicate:**
  - **Using Watermarking and Deduplication in Structured Streaming:** For streaming deduplication, use watermarking to handle late-arriving duplicates. Deduplicate based on a unique identifier (e.g., a combination of columns).
  - **Using dropDuplicates() in batch or micro-batch processing:** In batch or micro-batch scenarios, use the dropDuplicates() method in Spark. Specify the columns to consider for deduplication.
  - **Using Delta Lake's MERGE INTO:** If using Delta Lake, the MERGE INTO operation is the most efficient way to handle updates and deduplication. You can use it to insert new records and update existing ones based on a unique key, effectively deduplicating the data.

Example Code Snippet (Conceptual - Delta Lake and Python):

Python

```
from pyspark.sql.functions import *

from pyspark.sql.streaming import *


# Read stream from bronze (assuming Delta Lake)

bronze_stream = spark.readStream.format("delta").table("bronze_table")


# Deduplication and quality checks

silver_stream = bronze_stream \

  .withWatermark("event_timestamp", "10 minutes") \

  .dropDuplicates(["unique_id", "event_timestamp"]) \

  .filter("value > 0") # Example quality check
```

```
# Write to silver (Delta Lake) in append mode for new data
silver_stream.writeStream \
    .format("delta") \
    .option("checkpointLocation", "path/to/checkpoint") \
    .table("silver_table")

# For updates/upserts with deduplication using MERGE INTO (batch or micro-batch):

# Read new data (e.g., from a batch or micro-batch)
new_data = spark.read.format("delta").table("bronze_table_changes")

# Merge into silver table
new_data.merge(
    spark.table("silver_table").alias("silver"),
    "silver.unique_id = new_data.unique_id"
).whenMatchedUpdate(set = {
    "value": "new_data.value",
    "updated_at": current_timestamp()
}).whenNotMatchedInsert(values = {
    "unique_id": "new_data.unique_id",
    "value": "new_data.value",
    "event_timestamp": "new_data.event_timestamp",
    "updated_at": current_timestamp()
}).execute()
```

Make informed decisions about how to enforce data quality based on strengths and limitations of various approaches in Delta Lake

Data quality is crucial for reliable analytics and decision-making. Delta Lake offers several mechanisms to enforce data quality, each with its own strengths and limitations:

a) Schema Enforcement:

- **Strengths:** Ensures that data written to a Delta table conforms to a predefined schema. Prevents accidental ingestion of data with incorrect data types or missing columns.

- **Limitations:** Only checks the schema at write time. Doesn't validate data values against specific rules or business logic.

b) Constraints:

- **Strengths:** Allows defining rules that data must adhere to, such as CHECK constraints (e.g., value > 0) and NOT NULL constraints. Enforces data integrity beyond basic schema checks.
- **Limitations:** Constraints are currently informational in Delta Lake. They are not enforced at write time, but can be used for data validation and quality checks.

c) Expectations (with Delta Live Tables):

- **Strengths:** Enables defining data quality rules as "expectations" within Delta Live Tables (DLT) pipelines. Expectations can be used to monitor data quality, trigger alerts, or even prevent bad data from entering the table.
- **Limitations:** Primarily available within the DLT framework. Requires using DLT for pipeline development.

d) Data Validation with Spark:

- **Strengths:** Provides flexibility to implement custom data quality checks using Spark transformations and functions. Allows for complex validation logic and integration with external systems.
- **Limitations:** Requires writing and maintaining custom code. Can be more complex to implement compared to built-in features.

Making Informed Decisions:

Choosing the right approach depends on the specific data quality requirements and the context of your data pipeline. Consider the following factors:

- **Complexity of validation rules:** For simple schema checks, schema enforcement is sufficient. For more complex rules, constraints, expectations, or custom Spark validation may be necessary.
- **Pipeline framework:** If using DLT, leverage expectations for integrated data quality monitoring and enforcement.
- **Performance considerations:** Complex validation logic can impact pipeline performance. Optimize your code and choose appropriate techniques to minimize overhead.

Implement tables avoiding issues caused by lack of foreign key constraints

Delta Lake, like most data lakes, does not natively enforce foreign key constraints in the same way as traditional relational databases. This can lead to data integrity issues if not handled carefully. Here's how to mitigate these issues:

a) Data Modeling and Design:

- **Properly design your data model:** Clearly define relationships between tables and ensure data consistency through careful planning.
- **Use surrogate keys:** Instead of relying on natural keys that might change, use surrogate keys (e.g., auto-incrementing IDs) to uniquely identify records.

b) Data Validation and Quality Checks:

- **Implement data validation rules:** Use constraints, expectations, or custom Spark code to validate data relationships and ensure referential integrity.
- **Perform regular data quality checks:** Monitor your data for inconsistencies and errors that might arise due to the lack of foreign key constraints.

c) Data Transformation and Enrichment:

- **Use joins and lookups:** When querying data, use joins to combine related tables and ensure data consistency.
- **Enrich data with related information:** Instead of relying on separate tables with foreign key relationships, consider denormalizing data by including relevant information from related tables in a single table.

d) Data Governance and Documentation:

- **Establish data governance policies:** Define clear rules and procedures for data management and quality control.
- **Document data relationships:** Clearly document the relationships between tables to ensure that everyone understands the data model and how to query data correctly.

Example:

Suppose you have two tables: customers and orders. In a relational database, you would use a foreign key in the orders table to reference the customers table. In Delta Lake, you can achieve similar integrity by:

- Including a customer_id column in both tables.
- Using Spark joins to combine the tables when querying data.
- Implementing data validation rules to ensure that all customer_id values in the orders table exist in the customers table.

Add constraints to Delta Lake tables to prevent bad data from being written

Constraints in Delta Lake help maintain data quality and integrity by preventing bad data from being written into your tables. Delta Lake supports standard SQL constraint management clauses, ensuring that data added to a table is automatically verified. When a constraint is violated, Delta Lake throws an InvariantViolationException to signal that the new data can't [1] be added.

Here are the key types of constraints you can use in Delta Lake:

1. **NOT NULL Constraint:** This constraint ensures that values in specific columns cannot be null. If you attempt to insert a null value into a column with a NOT NULL constraint, the operation will fail.

SQL

ALTER TABLE my_table ALTER COLUMN my_column SET NOT NULL;

2. **CHECK Constraint:** This constraint allows you to specify a condition that must be true for all rows in the table. If a row violates the condition, the insertion or update operation will fail.

SQL

ALTER TABLE my_table ADD CONSTRAINT positive_value CHECK (my_column > 0);

3. **Primary Key and Foreign Key Constraints:** Delta Lake also supports primary key and foreign key constraints to enforce relationships between tables. These constraints are informational and not enforced by Delta Lake itself but can be used by query optimizers and other tools.

SQL

ALTER TABLE my_table ADD CONSTRAINT primary_key PRIMARY KEY (id);

ALTER TABLE another_table ADD CONSTRAINT foreign_key FOREIGN KEY (my_table_id) REFERENCES my_table(id);

Important Notes:

- Adding a constraint automatically upgrades the table writer protocol version.
- Delta Lake verifies the validity of CHECK constraints against both new and existing data.
- Constraints can be added or dropped using ALTER TABLE commands.

Implement lookup tables and describe the trade-offs for normalized data models

Lookup tables are used to store reference data that is frequently used in queries. They can improve query performance by avoiding the need to repeatedly join large tables. In a normalized data model, data is divided into multiple tables to reduce redundancy. Lookup tables can be used to store the values that are referenced by foreign keys in other tables.

Trade-offs for Normalized Data Models:

While normalization offers benefits like reduced data redundancy and improved data integrity, it can also lead to more complex queries that require multiple joins. This is where lookup tables can help. By storing frequently used reference data in a separate table, you can simplify queries and improve performance.

However, there are also trade-offs to consider:

- **Increased Storage Space:** Storing the same data in multiple tables can increase storage space requirements.
- **Data Duplication:** While normalization aims to reduce redundancy, lookup tables can introduce some level of duplication.
- **Data Consistency:** Maintaining data consistency across multiple tables can be more challenging.

Example:

Suppose you have a table of customer orders with a foreign key referencing a table of customer information. You could create a lookup table for customer information that includes only the columns that are frequently used in queries, such as customer name and address. This would allow you to avoid joining the large customer information table in many queries.

Diagram architectures and operations necessary to implement various Slowly Changing Dimension tables using Delta Lake with streaming and batch workloads.

In data warehousing, dimensions contain descriptive attributes that provide context to facts (measurements or events). These attributes can change over time, and SCDs are techniques to manage these changes. Different SCD types offer different ways to track history and updates.

Delta Lake

Delta Lake is an open-source storage layer that brings ACID (Atomicity, Consistency, Isolation, Durability) transactions to Apache Spark and big data workloads. It [1] provides features like:

- **Schema Enforcement:** Ensures data quality by preventing ingestion of data with incorrect schemas.
- **Time Travel:** Allows querying historical versions of data.
- **Upserts and Deletes:** Enables efficient updates and deletes of data.
- **Streaming and Batch Unification:** Supports both streaming and batch data processing.

SCD Types

Here's a breakdown of the SCD types you mentioned:

- **SCD Type 0 (Retain Original):** This type does not track changes at all. The original data is retained, and updates are not reflected. This is useful for attributes that should never change.
- **SCD Type 1 (Overwrite):** This type overwrites the old value with the new value. No history is retained. This is suitable for correcting errors or for attributes where history is not important.
- **SCD Type 2 (Add New Row):** This type creates a new row in the dimension table whenever an attribute changes. The old row is marked as expired, and the new row is marked as current. This maintains a full history of changes.

Implement SCD Type 0, 1, and 2 tables

Delta Lake's features make implementing SCDs more efficient and reliable. Here's how you can implement SCD Type 0, 1, and 2 using Delta Lake with streaming and batch workloads:

1. SCD Type 0

- **Implementation:** Simply load the data into a Delta table. No special handling is required for updates.
- **Streaming/Batch:** Works seamlessly with both. New data is simply appended to the table.

2. SCD Type 1

- **Implementation:** Use Delta Lake's merge operation to update existing records with new values.
    - **Batch:** Read the new data and use merge to update the Delta table based on a join condition (e.g., primary key).
    - **Streaming:** Use foreachBatch with the merge operation to apply updates in each micro-batch.
- **Example (PySpark):**

Python

```
from pyspark.sql.functions import *


def upsert_to_delta(micro_batch_df, batch_id):
  micro_batch_df.alias("updates") \
    .merge(
      delta_table.alias("customers"),
      "updates.customer_id = customers.customer_id"
    ) \
    .whenMatchedUpdate(set = {
      "customer_name": "updates.customer_name",
      "city": "updates.city"
    }) \
    .execute()


streaming_df.writeStream.foreachBatch(upsert_to_delta).start()
```

3. SCD Type 2

- **Implementation:** Use Delta Lake's merge operation with additional logic to track history.
    - Add columns like effective_start_date, effective_end_date, and is_current.
    - When an update occurs:
        - Update the effective_end_date of the old row.
        - Insert a new row with the new values and the current timestamp as effective_start_date.
    - **Batch:** Similar logic as SCD Type 1, but with additional columns and conditions.
    - **Streaming:** Use foreachBatch with the more complex logic to handle history tracking.
- **Example (Conceptual):**

SQL

MERGE INTO customers AS target

USING updates AS source

ON target.customer_id = source.customer_id

WHEN MATCHED AND (target.customer_name <> source.customer_name OR target.city <> source.city) THEN UPDATE SET target.effective_end_date = current_timestamp(), target.is_current = false

WHEN NOT MATCHED THEN INSERT (customer_id, customer_name, city, effective_start_date, effective_end_date, is_current) VALUES (source.customer_id, source.customer_name, source.city, current_timestamp(), null, true);

Diagrams

Diagrams for these architectures would generally involve:

- **Data Sources:** Representing the source systems providing data.
- **Data Ingestion:** Showing how data is ingested into Databricks (e.g., using Auto Loader for streaming).
- **Delta Lake Tables:** Representing the Delta tables storing the dimension data.
- **Processing Logic:** Showing the Spark transformations and Delta Lake operations (e.g., merge) used to implement the SCD logic.
- **Downstream Systems:** Representing the systems consuming the dimension data (e.g., reporting tools).

Key Considerations

- **Performance:** Optimize merge operations by using appropriate join conditions and partitioning.
- **Data Quality:** Implement data quality checks to ensure accurate history tracking.

- **Complexity:** SCD Type 2 is more complex to implement but provides the most comprehensive history.

Section 4: Security & Governance

Create Dynamic views to perform data masking

Data masking is crucial for protecting sensitive information while still allowing users to work with data for analysis, testing, or development. Dynamic views provide a powerful way to implement this by applying masking logic at query time. This means the underlying data remains unchanged, but the view presents a modified version based on user context or other criteria.

Here's how you can achieve this in Databricks using SQL:

SQL

```
CREATE OR REPLACE VIEW masked_customer_data AS
SELECT
  customer_id,
  CASE
    WHEN current_user() IN ('analyst_user', 'data_scientist') THEN email  -- Allow full email for specific users
    ELSE '***@***.com'  -- Mask email for others
  END AS masked_email,
  CASE
    WHEN is_member THEN phone_number -- Show phone number for members
    ELSE NULL -- Hide phone number for non-members
  END AS masked_phone_number,
  -- Apply other masking techniques as needed
  CASE
    WHEN current_user() NOT IN ('sensitive_data_user') THEN SUBSTRING(credit_card_number, -4) -- Show last 4 digits only
    ELSE credit_card_number
  END AS masked_credit_card,
  --Unmasked column
  customer_name,
  address
FROM
  customer_table;
```

Explanation:

- **CREATE OR REPLACE VIEW masked_customer_data**: This creates or updates a view named masked_customer_data.
- **CASE WHEN ... THEN ... ELSE ... END**: This is the core of the masking logic. It checks conditions and applies different masking techniques accordingly.
- **current_user()**: This function returns the username of the user executing the query. This allows you to tailor the masking based on user roles or permissions.
- **Masking Techniques:**
    - **Substitution:** Replacing sensitive data with a fixed value (e.g., '***@***.com').
    - **Partial Masking:** Showing only a portion of the data (e.g., last four digits of a credit card).
    - **Nulling Out:** Hiding data completely by returning NULL.
- **is_member**: This could be a column in your table indicating membership status, providing another condition for masking.

Use dynamic views to control access to rows and columns

Dynamic views can also be used to control which rows and columns a user can access. This is essential for implementing fine-grained access control and enforcing data governance policies.

Here's an example:

SQL

```
CREATE OR REPLACE VIEW restricted_sales_data AS
SELECT
  order_id,
  order_date,
  product_name,
  sales_amount
FROM
  sales_table
WHERE
  CASE
    WHEN current_user() = 'regional_manager_east' THEN region = 'East'
    WHEN current_user() = 'regional_manager_west' THEN region = 'West'
    ELSE TRUE -- Allow all rows for other users (e.g., admins)
```

```
END;


CREATE OR REPLACE VIEW restricted_hr_data AS
SELECT
  CASE WHEN current_user() IN ('hr_manager') THEN employee_id ELSE NULL END as masked_employee_id,
  CASE WHEN current_user() IN ('hr_manager') THEN salary ELSE NULL END as masked_salary,
  employee_name,
  department
FROM hr_table;
```

Explanation:

- **WHERE Clause with CASE Statement:** The WHERE clause filters rows based on the user executing the query.
- **Regional Access Control:** The first example restricts regional managers to only see sales data for their respective regions.
- **Role-Based Access Control:** The second example grants access to sensitive HR information (employee ID and salary) only to users with the 'hr_manager' role. Others can see non-sensitive information like employee name and department.

Key Advantages of Dynamic Views:

- **Centralized Masking Logic:** The masking logic is defined in the view, making it easy to manage and update.
- **No Data Duplication:** The underlying data is not modified, saving storage space and ensuring data consistency.
- **Real-Time Masking:** Masking is applied at query time, ensuring that users always see the appropriate level of data.
- **Flexible and Customizable:** You can implement complex masking rules based on various criteria.
- **Integration with Unity Catalog:** In Databricks, Unity Catalog enhances data governance by providing a centralized place to manage permissions and access control on these views.

Important Considerations:

- **Performance:** Complex masking logic can impact query performance. Optimize your views and consider using caching if necessary.
- **Security:** Ensure that the view definitions themselves are protected with appropriate permissions.

- **Testing:** Thoroughly test your views to ensure that the masking and access control are working as expected.

Section 5: Monitoring & Logging

Describe the elements in the Spark UI to aid in performance analysis, application debugging, and tuning of Spark applications.

The Spark UI is organized into several tabs, each providing specific information about your Spark application. Here are some of the key tabs and elements:

- **Jobs Tab:** This tab provides a high-level overview of all the jobs that have been executed by your application. You can see the status of each job, the time it took to complete, and the number of stages and tasks it comprised.
    - **Job ID:** A unique identifier for each job.
    - **Description:** A description of the job, which can be helpful for identifying the purpose of the job.
    - **Submitted Time:** The time at which the job was submitted.
    - **Completion Time:** The time at which the job was completed.
    - **Duration:** The total time it took to complete the job.
    - **Stages:** The number of stages in the job.
    - **Tasks:** The number of tasks in the job.
- **Stages Tab:** This tab provides detailed information about each stage in a job. You can see the status of each stage, the time it took to complete, and the number of tasks it comprised.
    - **Stage ID:** A unique identifier for each stage.
    - **Description:** A description of the stage, which can be helpful for understanding what the stage is doing.
    - **Submitted Time:** The time at which the stage was submitted.
    - **Completion Time:** The time at which the stage was completed.
    - **Duration:** The total time it took to complete the stage.
    - **Tasks:** The number of tasks in the stage.
    - **Input:** The amount of data read by the stage.
    - **Output:** The amount of data written by the stage.
    - **Shuffle Read:** The amount of data read from the shuffle.
    - **Shuffle Write:** The amount of data written to the shuffle.
- **Storage Tab:** This tab provides information about the RDDs and DataFrames that are stored in memory. You can see the size of each RDD or DataFrame, the number of partitions, and the memory usage.

- **Environment Tab:** This tab provides information about the Spark environment, including the Spark version, the Java version, and the system properties.
- **Executors Tab:** This tab provides information about the executors that are running your Spark application. You can see the status of each executor, the memory usage, and the number of tasks it has completed.
- **SQL Tab:** This tab provides information about the Spark SQL queries that have been executed by your application. You can see the SQL query, the execution plan, and the metrics.

Inspect event timelines and metrics for stages and jobs performed on a cluster

The Spark UI also provides event timelines and metrics for stages and jobs. These can be invaluable for identifying performance bottlenecks and debugging errors.

- **Event Timeline:** The event timeline shows the sequence of events that occurred during the execution of a stage or job. This can be helpful for understanding the flow of execution and identifying any delays or bottlenecks.
- **Metrics:** The Spark UI provides a variety of metrics for stages and jobs, including the time taken for each task, the amount of data read and written, and the shuffle read and write times. These metrics can be used to identify performance bottlenecks and tune your Spark applications.

Using the Spark UI for Performance Analysis, Debugging, and Tuning

Here are some tips for using the Spark UI for performance analysis, debugging, and tuning:

- **Identify Long-Running Jobs and Stages:** Look for jobs and stages that are taking a long time to complete. These are the most likely candidates for performance bottlenecks.
- **Analyze Task Execution:** Look at the task execution times to identify any tasks that are taking significantly longer than others. This can indicate data skew or other performance issues.
- **Monitor Memory Usage:** Monitor the memory usage of your Spark application to ensure that you have enough memory allocated. If you are running out of memory, you may need to increase the amount of memory allocated to your executors.
- **Analyze Shuffle Read and Write Times:** The shuffle read and write times can be a significant source of overhead in Spark applications. Look for high shuffle read and write times, which can indicate that you need to optimize your data partitioning or reduce the amount of data being shuffled.
- **Use the Event Timeline to Understand Execution Flow:** The event timeline can be helpful for understanding the flow of execution and identifying any delays or bottlenecks.

Draw conclusions from information presented in the Spark UI, Ganglia UI, and the Cluster UI to assess performance problems and debug failing applications.

- **Spark UI:** This web-based interface provides detailed information about Spark applications, including:
    - **Jobs:** Stages, tasks, and their execution details.
    - **Stages:** DAG visualization, task metrics (time, shuffle, etc.).

    - **Storage:** RDD persistence and memory usage.
    - **Environment:** Configuration properties and dependencies.
    - **Executors:** Resource usage and task distribution.
- **Ganglia UI:** A monitoring system that provides cluster-level metrics like CPU, memory, network, and disk I/O.
- **Cluster UI:** Databricks provides a UI to monitor cluster health, resource allocation, and active jobs.

By analyzing these UIs, a data engineer can:

- Identify performance bottlenecks (e.g., long-running tasks, data skew).
- Debug failing applications (e.g., task failures, out-of-memory errors).
- Optimize resource allocation and configuration.

Design systems that control for cost and latency SLAs for production streaming jobs.

Cost Control:

- Right-sizing clusters and using spot instances.
- Efficient data processing and minimizing data movement.
- Monitoring resource usage and setting budgets.

Latency SLAs:

- Choosing appropriate stream processing frameworks (e.g., Spark Streaming, Structured Streaming).
- Tuning batch intervals and micro-batch sizes.
- Optimizing data serialization and partitioning.
- Monitoring end-to-end latency and setting alerts.

Deploy and monitor streaming and batch jobs

Deployment:

- Packaging and deploying code to Databricks clusters.
- Configuring job execution and dependencies.
- Automating deployments using CI/CD pipelines.

Monitoring:

- Setting up monitoring dashboards and alerts.
- Tracking job execution and performance metrics.
- Troubleshooting and resolving issues.

- Ensuring job reliability and availability.

Section 6: Testing & Deployment

Adapt a notebook dependency pattern to use Python le dependencies

This focuses on managing external Python libraries within your Databricks notebooks. Traditionally, notebooks might have relied on %run commands to include other notebooks or relied on installing libraries directly within the notebook using %pip install. These approaches have limitations in terms of reproducibility, dependency management, and version control.

The preferred approach is to leverage Python library dependencies, primarily using requirements.txt or pyproject.toml (with build tools like Poetry or Flit).

- **requirements.txt:** This simple file lists the required Python packages and their versions (e.g., pandas==1.5.0, requests>=2.28.1). You can install these dependencies in a Databricks cluster or within a notebook's environment using %pip install -r requirements.txt. This ensures consistent environments across different runs and users.
- **pyproject.toml:** This file is used for more complex projects and can specify build requirements, dependencies, and other metadata. Tools like Poetry and Flit use this file to manage dependencies and create distributable packages (wheels).

By using these methods, you gain:

- **Reproducibility:** Consistent environments across different runs.
- **Version control:** Track changes to dependencies.
- **Easier deployment:** Streamlined dependency installation in production.
- **Dependency resolution:** Tools handle complex dependency relationships.

Example: Instead of %run ./my_helper_functions.ipynb, you would create a Python package (even a simple one) containing those functions and declare it as a dependency in requirements.txt.

Adapt Python code maintained as Wheels to direct imports using relative paths

This addresses organizing your Python code into reusable modules and packages. Wheels (.whl files) are the standard distribution format for Python packages. While distributing code as wheels is good for sharing and deployment, within a Databricks project, especially during development, using relative imports is often more convenient.

- **Wheels:** Packages are built and distributed as wheels. This is suitable for deploying to production clusters or sharing with others.
- **Relative imports:** Within your project's directory structure, you can use relative imports to refer to other modules or packages. For example, if you have a structure like:

my_project/

├── my_package/

| ├── __init__.py

| ├── module_a.py

```
|   └── module_b.py
└── my_notebook.ipynb
```

In module_b.py, you could import module_a using from . import module_a or from .module_a import some_function. In my_notebook.ipynb you would import the package using from my_package import module_b.

This approach is better for development because:

- It's easier to navigate and understand the code structure.
- You don't need to constantly rebuild and install wheels during development.
- Changes to modules are immediately reflected.

When you're ready to deploy, you can package your code as a wheel.

Repair and rerun failed jobs

This covers the essential aspects of job monitoring and troubleshooting in Databricks.

- **Monitoring:** Databricks provides tools to monitor job execution, including logs, metrics, and event logs. You can use the Databricks UI, the REST API, or monitoring tools like Grafana to track job status, resource usage, and errors.
- **Repairing:** When a job fails, you need to diagnose the cause. Common causes include:
    - **Code errors:** Bugs in your Python, Scala, or SQL code.
    - **Data issues:** Incorrect or missing data.
    - **Resource limitations:** Insufficient memory, CPU, or disk space.
    - **Dependency conflicts:** Issues with library versions.
    - **External system failures:** Problems with databases, APIs, or other services.

Once you've identified the cause, you need to fix it. This might involve code changes, data corrections, resource adjustments, or dependency updates.

- **Rerunning:** After repairing the issue, you can rerun the failed job. Databricks provides options to rerun the entire job or just the failed tasks.

Create Jobs based on common use cases and patterns

This involves understanding typical data engineering tasks and translating them into Databricks Jobs. Here's a breakdown:

- **Understanding Common Use Cases:**
    - **Data Ingestion:** Moving data from various sources (databases, APIs, cloud storage, streaming platforms) into the lakehouse. Common patterns include:
        - **Batch Ingestion:** Regularly scheduled jobs (e.g., daily, hourly) that move large volumes of data.
        - **Incremental Ingestion:** Capturing only changes in the source data since the last ingestion, often using change data capture (CDC) techniques.
        - **Streaming Ingestion:** Continuously ingesting data as it arrives, typically using Apache Kafka or similar technologies.
    - **Data Transformation:** Cleaning, transforming, and enriching data to prepare it for analysis. Common patterns include:
        - **ETL (Extract, Transform, Load):** Classic data warehousing approach.
        - **ELT (Extract, Load, Transform):** Modern approach leveraging the power of cloud data warehouses and lakehouses.
    - **Data Quality Checks:** Implementing checks to ensure data accuracy, completeness, and consistency.
    - **Data Modeling:** Creating optimized data models (e.g., star schema, snowflake schema) for analytical queries.
    - **Data Export/Delivery:** Moving processed data to downstream systems (e.g., data warehouses, reporting tools).
- **Translating Use Cases into Databricks Jobs:**
    - Using Databricks notebooks (Python, Scala, SQL, R) to implement the data processing logic.
    - Configuring job schedules (e.g., using cron expressions) to automate job execution.
    - Setting up job dependencies to ensure tasks run in the correct order.
    - Using Databricks Workflows (formerly Jobs) to orchestrate complex data pipelines.
- **Example:** A common use case is ingesting sales data from a transactional database daily, transforming it, and loading it into a data warehouse for reporting. This would translate into a Databricks Job with the following steps:

  Extract: Read sales data from the database using JDBC.

  Transform: Clean and aggregate the data using Spark SQL or DataFrames.

  Load: Write the processed data to a Delta table in the lakehouse.

Create a multi-task job with multiple dependencies

This involves orchestrating complex data pipelines with multiple interconnected tasks.

- **Understanding Dependencies:** Dependencies define the order in which tasks must execute. A task can only start after its dependencies have successfully completed.
- **Implementing Dependencies in Databricks:** Databricks Workflows provide a visual interface and programmatic APIs for defining dependencies between tasks.
- **Example:** Imagine a data pipeline with the following steps:
    1. **Ingest Raw Data:** Ingest data from various sources.
    2. **Cleanse Data:** Clean and validate the ingested data.
    3. **Transform Data:** Perform complex transformations and aggregations.
    4. **Load to Data Warehouse:** Load the transformed data into a data warehouse.
    5. **Generate Reports:** Generate analytical reports based on the data in the data warehouse.

In this case, the "Cleanse Data" task depends on the "Ingest Raw Data" task, the "Transform Data" task depends on the "Cleanse Data" task, and so on. Databricks Workflows allows you to define these dependencies, ensuring that each task runs only after its prerequisites are met.

Design systems that control for cost and latency SLAs for production streaming jobs.

This is crucial for real-time data processing.

- **Cost Control:**
    - **Right-Sizing Clusters:** Choosing appropriate cluster sizes (number of workers, instance types) to avoid over-provisioning resources.
    - **Autoscaling:** Configuring clusters to automatically scale up or down based on workload demands.
    - **Spot Instances:** Using spot instances (discounted but interruptible instances) for fault-tolerant workloads.
    - **Monitoring Resource Usage:** Tracking CPU, memory, and network usage to identify areas for optimization.
    - **Optimizing Spark Code:** Writing efficient Spark code to minimize resource consumption and execution time.
- **Latency SLAs (Service Level Agreements):**
    - **Understanding Latency Requirements:** Defining acceptable latency thresholds for different use cases.
    - **Micro-Batching:** Processing streaming data in small batches to reduce latency.
    - **Structured Streaming:** Using Structured Streaming for fault-tolerant and exactly-once processing of streaming data.

- **State Management:** Efficiently managing state in streaming applications to avoid performance bottlenecks.
- **Monitoring Latency Metrics:** Tracking end-to-end latency to ensure SLAs are met.
- **Alerting:** Setting up alerts to notify when latency exceeds predefined thresholds.

- **Example:** A real-time fraud detection system requires low latency to detect fraudulent transactions quickly. To meet latency SLAs, you might use Structured Streaming with micro-batching, optimize state management, and monitor latency metrics closely. To control costs, you might use autoscaling and right-size your clusters based on the expected data volume.

Configure the Databricks CLI and execute basic commands to interact with the workspace and clusters.

What is the Databricks CLI? The Databricks Command-Line Interface (CLI) is a powerful tool that allows you to interact with your Databricks workspace from your terminal or command prompt. It provides a way to automate tasks, manage resources, and perform various operations without relying on the web UI.

Configuration:

- **Installation:** You typically install the Databricks CLI using pip:

Bash

pip install databricks-cli

- **Authentication:** The most common way to authenticate is using a personal access token (PAT). You generate a PAT in your Databricks workspace (User Settings > Access Tokens). Then, you configure the CLI using the databricks configure command:

Bash

databricks configure --token

You'll be prompted for your Databricks workspace URL and the PAT.

Alternatively, you can set environment variables:

Bash

export DATABRICKS_HOST="<your-workspace-url>"

export DATABRICKS_TOKEN="<your-pat>"

- **Profiles (Optional):** You can create multiple profiles for different workspaces or authentication methods. This allows you to easily switch between them using the --profile option.

Basic Commands:

- databricks workspace ls: Lists the contents of a workspace path (like files and folders).
- databricks workspace mkdirs: Creates directories in the workspace.
- databricks workspace rm: Removes files or directories from the workspace.

- databricks clusters list: Lists available clusters in your workspace.
- databricks clusters get <cluster-id>: Retrieves details about a specific cluster.
- databricks clusters create: Creates a new cluster (requires a JSON configuration).
- databricks clusters start/stop/restart <cluster-id>: Manages cluster lifecycle.
- databricks fs ls: Lists files in DBFS (Databricks File System).
- databricks fs cp: Copies files between local filesystem and DBFS.

Execute commands from the CLI to deploy and monitor Databricks jobs.

Deploying Jobs:

- databricks jobs create: Creates a new job. This requires a JSON configuration file specifying the job's tasks, libraries, and other settings.
- databricks jobs reset: Updates an existing job with a new configuration.

Monitoring Jobs:

- databricks jobs list: Lists all jobs in your workspace.
- databricks jobs get <job-id>: Retrieves details about a specific job.
- databricks jobs run-now <job-id>: Triggers a new run of a job.
- databricks jobs runs list <job-id>: Lists runs for a specific job.
- databricks jobs runs get <run-id>: Retrieves details about a specific job run, including its state, start time, and end time.
- databricks jobs runs get-output <run-id>: Gets the output of a job run.
- databricks jobs runs cancel <run-id>: Cancels a running job.

Use REST API to clone a job, trigger a run, and export the run output

- **What is the Databricks REST API?** The Databricks REST API provides programmatic access to almost all Databricks functionalities. It allows you to integrate Databricks with other tools and automate complex workflows.
- **Authentication:** Similar to the CLI, you typically authenticate using a personal access token (PAT). You include the token in the Authorization header of your HTTP requests:
- Authorization: Bearer <your-pat>
- **Cloning a Job:** You would use the POST /jobs/create endpoint with the configuration of an existing job (retrieved using GET /jobs/get/<job-id>), modifying the name or other relevant parameters to create a clone.
- **Triggering a Run:** You use the POST /jobs/run-now endpoint, providing the job ID and any necessary parameters for the run (e.g., notebook parameters).
- **Exporting Run Output:** You use the GET /jobs/runs/get-output endpoint, providing the run ID. This will return the output of the job run, which you can then process or save as needed. The output can be structured (e.g., JSON) or unstructured (e.g., logs).

- **Example (using curl): Triggering a run:**

Bash

```
curl -X POST \
 https://<your-workspace-url>/api/2.1/jobs/run-now \
 -H 'Authorization: Bearer <your-pat>' \
 -H 'Content-Type: application/json' \
 -d '{
   "job_id": <job-id>
 }'
```

Key Differences between CLI and REST API:

- **CLI:** Easier for simple, interactive tasks and scripting.
- **REST API:** More flexible and powerful for complex integrations, custom applications, and programmatic control.