PassQueen
a.k.a dumpsqueen.com

Databricks Certified Data Engineer Associate Exam

Databricks Databricks-Certified-Data-Engineer-Associate

Version Demo

Total Demo Questions: 30

Total Premium Questions: 304

Buy Premium PDF

https://passqueen.com

support@passqueen.com

Topic Break Down

Topic	No. of Questions
Topic 1, Data Management	96
Topic 2, Data Ingestion and Processing	151
Topic 3, Data Modeling	14
Topic 4, Security and Governance	43
Total	304

QUESTION NO: 1

Which compute option should be chosen in a scenario where small-scale ad hoc Python scripts need to be run at high frequency and should wind down quickly after these queries have finished running?

A. All-purpose cluster

B. Job cluster

C. Serverless compute

D. SQL Warehouse

ANSWER: C

Explanation:

Serverless compute is the appropriate choice because it is designed to provide rapidly available, automatically managed compute for supported Databricks workloads. For small, frequent Python executions, avoiding cluster provisioning and manual lifecycle management reduces operational overhead and helps work start promptly. Databricks manages the underlying infrastructure, including capacity allocation and scaling, so the compute resources can be released when the execution is complete rather than requiring a user-managed cluster to remain available. This model is especially well suited to short-lived, intermittent workloads where maintaining dedicated infrastructure would create unnecessary idle time and cost. Databricks recommends using serverless compute when it is available and compatible with the workload, because it provides a simplified experience while Databricks handles compute operations. Python code can be executed in Databricks notebooks and jobs using serverless-capable configurations, subject to the workspace's supported serverless features and applicable limitations. See the Databricks guidance on serverless compute and its overview of compute options.

QUESTION NO: 2

Which of the following describes when to use the CREATE STREAMING LIVE TABLE (formerly CREATE INCREMENTAL LIVE TABLE) syntax over the CREATE LIVE TABLE syntax when creating Delta Live Tables (DLT) tables using SQL?

A. CREATE STREAMING LIVE TABLE should be used when the subsequent step in the DLT pipeline is static.

B. CREATE STREAMING LIVE TABLE should be used when data needs to be processed incrementally.

C. CREATE STREAMING LIVE TABLE is redundant for DLT and it does not need to be used.

D. CREATE STREAMING LIVE TABLE should be used when data needs to be processed through complicated aggregations.

E. CREATE STREAMING LIVE TABLE should be used when the previous step in the DLT pipeline is static.

ANSWER: B

Explanation:

`CREATE STREAMING LIVE TABLE should be used when data needs to be processed incrementally.` A streaming live table is designed for incremental processing: during pipeline updates, it processes newly arriving records rather than recomputing the full result from all available source data. This makes it the appropriate declaration for continuous or triggered ingestion from streaming sources and for append-oriented incremental loads from batch sources. The table maintains the streaming query's progress and state, allowing the pipeline to efficiently continue from where prior processing ended. In contrast, a non-streaming live table is appropriate for materialized computations whose result can be refreshed by evaluating the defining query over its input data. The key decision is therefore the required processing model: use the streaming declaration when the dataset must consume and transform newly available data incrementally. Databricks documentation describes streaming tables as incrementally processing new data as it arrives and distinguishes them from materialized views, which are refreshed from a query result. See the Databricks SQL pipeline transformations documentation and the Databricks streaming tables documentation.

QUESTION NO: 3

The Delta transaction log for the 'students' tables is shown using the 'DESCRIBE HISTORY students' command. A Data Engineer needs to query the table as it existed before the UPDATE operation listed in the log.

	version	timestamp	operation
1	8	2024–04–22T14:33:31.000	OPTIMIZE
2	7	2024–04–22T14:33:16.000	MERGE
3	6	2024–04–22T14:33:06.000	DELETE
4	5	2024–04–22T14:32:58.000	UPDATE
5	4	2024–04–22T14:32:47.000	WRITE
6	3	2024–04–22T14:32:44.000	WRITE
7	2	2024–04–22T14:32:23.000	WRITE
8	1	2024–04–22T14:32:20.000	WRITE
9	0	2024–04–22T14:31:49.000	CREATE TABLE

Which command should the Data Engineer use to achieve this? (Choose two.)

A. SELECT * FROM students@v4

B. SELECT * FROM students TIMESTAMP AS OF '2024-04-22T 14:32:47.000+00:00'

C. SELECT * FROM students FROM HISTORY VERSION AS OF 3

D. SELECT * FROM students VERSION AS OF 5

E. SELECT * FROM students TIMESTAMP AS OF '2024-04-22T 14:32:58.000+00:00'

ANSWER: A B

Explanation:

Delta Lake time travel allows a query to read a prior, consistent snapshot of a Delta table using either a transaction-log version or a timestamp. `SELECT * FROM students@v4` uses the concise version-addressing syntax to read version 4 of the `students` table. Because the UPDATE is recorded in the subsequent table version, version 4 represents the table state immediately before that UPDATE was committed.

`SELECT * FROM students TIMESTAMP AS OF '2024-04-22T 14:32:47.000+00:00'` is also a valid time-travel approach: it requests the latest committed table version at or before the specified point in time. The timestamp shown precedes the UPDATE transaction's commit time, so Delta resolves the query to the pre-update snapshot. This is particularly useful when a known business or event time is available instead of the precise Delta version number.

Both forms depend on the relevant historical Delta log and data files still being retained. For syntax and retention considerations, see the Databricks documentation on Delta Lake table history and time travel and the time-travel query examples.

QUESTION NO: 4

Which TWO items are characteristics of the Gold Layer?

Choose 2 answers

A. Read-optimized

B. Normalised

C. Raw Data

D. Historical lineage

E. De-normalised

ANSWER: A E

Explanation:

The Gold layer is the curated, consumption-oriented layer of the medallion architecture. It contains data that has been refined into business-ready datasets for reporting, dashboards, analytics, and machine learning consumption. Consequently, **Read-optimized** is a characteristic of this layer: Gold tables are designed around the access patterns of downstream users and applications, commonly with precomputed aggregations and structures that make common queries efficient.

De-normalised is also characteristic of Gold datasets. Gold commonly uses dimensional models, including fact and dimension tables, and may use wide or aggregated tables to provide understandable, performant data products for business users. This organization reduces the need for consumers to perform complex joins against lower-level operational-style records. The precise Gold design should be driven by the intended business use case, but its central role is to expose high-quality, easy-to-query data products rather than raw ingestion data.

Databricks describes the Gold layer as providing data in a form suitable for business intelligence, reporting, and other downstream consumption, often using aggregated or modeled data. See the Databricks medallion architecture documentation and the Databricks medallion architecture overview.

QUESTION NO: 5

A data engineer has been provided a PySpark DataFrame named df with columns product and revenue. The data engineer needs to compute complex aggregations to determine each product ' s total revenue, average revenue, and transaction count.

Which code snippet should the data engineer use?

A)

```
from pyspark.sql import functions as F
aggregated_df = df.groupBy("product").agg( F.sum("revenue").alias("total_revenue"),
F.avg("revenue").alias("avg_revenue"), F.count("*").alias("transaction_count") )
```

B)

```
aggregated_df = df.groupby("product").agg(
    "sum(revenue)",
    "avg(revenue)"
)
```

C)

```
aggregated_df = df.groupby("product").agg(
    "sum(revenue)",
    "avg(revenue)",
    "count(revenue)"
)
```

D)

```
aggregated_df =
df.groupBy("product").agg( {"revenue": "sum", "revenue": "avg", "revenue": "count"} )
```

A. Option A

B. Option B

C. Option C

D. Option D

ANSWER: A

Explanation:

The code shown in the first image is correct because this requirement needs grouped aggregation: records must first be grouped by `product`, then multiple aggregate expressions must be evaluated for each product group. In PySpark, the standard pattern is `df.groupBy("product").agg(...)`. The aggregation expressions should calculate the sum of `revenue` for total revenue, the average of `revenue` for average revenue, and a count for the number of transactions in that product's group. Aliases can then give these resulting columns clear names such as `total_revenue`, `average_revenue`, and `transaction_count`.

This approach produces one output row per distinct product and evaluates all requested measures in the same aggregation operation. It is also the scalable DataFrame API approach: Spark can plan the grouping and aggregate computation across the cluster rather than requiring row-by-row application logic. PySpark documents `groupBy` as creating grouped data for aggregation and `GroupedData.agg` as accepting multiple aggregate expressions. See the PySpark DataFrame.groupBy documentation and the GroupedData.agg documentation.

QUESTION NO: 6

A data engineer is configuring a Structured Streaming query that must recover correctly after a cluster failure.

Which two practices should the data engineer follow for the query checkpoint location? (Choose two.)

A. Store the checkpoint location in reliable, durable storage.

B. Reuse the same checkpoint location when restarting the same query.

C. Use one checkpoint location for all unrelated streaming queries.

D. Delete the checkpoint after each successful micro-batch.

E. Store all source data files inside the checkpoint location.

ANSWER: A B

Explanation:

A checkpoint location must be stored in reliable, durable storage so that the streaming query can recover its offset and state information after a failure. The checkpoint contains metadata that enables the query to determine its prior progress.

When restarting the same streaming query, the engineer should reuse the same checkpoint location. Reusing the checkpoint allows the restarted query to resume from the recorded progress rather than treating previously processed input as new data.

A checkpoint should not be deleted simply because one successful batch has completed, and a checkpoint location should not be shared by unrelated streaming queries. See the Databricks Structured Streaming checkpoint documentation.

QUESTION NO: 7

A data engineer is processing ingested streaming tables and needs to filter out NULL values in the order_datetime column from the raw streaming table orders_raw and store the results in a new table orders_valid using DLT.

Which code snippet should the data engineer use?

A)

```
CREATE OR REFRESH STREAMING TABLE orders_valid
(
  CONSTRAINT valid_date EXPECT (order_datetime IS NOT NULL)
  ON VIOLATION DROP ROW
)
AS SELECT * FROM orders_raw;
```

B)

```
CREATE OR REFRESH STREAMING TABLE orders_valid(
  CONSTRAINT valid_date
  EXPECT (order_datetime IS NOT NULL)
  ON VIOLATION DROP ROW
)
AS SELECT * FROM STREAM orders_raw;
```

C)

```
CREATE OR REFRESH STREAMING TABLE orders_valid
AS
SELECT *
FROM STREAM(orders_raw)
WHERE order_datetime IS NOT NULL;
```

D)

```
CREATE OR REPLACE STREAMING TABLE orders_valid
(
  FILTER (order_datetime IS NOT NULL)
)
AS SELECT * FROM STREAM(orders_raw);
```

A. Option A

B. Option B

C. @dlt.table def orders_valid(): return dlt.read_stream("orders_raw").filter("order_datetime IS NOT NULL")

D. Option D

ANSWER: C

Explanation:

`@dlt.table`
`def orders_valid():`
`  return dlt.read_stream("orders_raw").filter("order_datetime IS NOT NULL")` is correct because it

defines `orders_valid` as a DLT-managed streaming table derived from the upstream `orders_raw` table. The `dlt.read_stream()` function declares that the input is consumed incrementally as a stream, rather than as a one-time batch read. This preserves streaming semantics through the pipeline, allowing newly ingested records in `orders_raw` to be processed into the downstream table.

The DataFrame `filter()` transformation applies the SQL predicate `order_datetime IS NOT NULL`. This retains only records that have a usable value in the required datetime column, so null-valued records are excluded before DLT materializes the output. The function name becomes the target table name, meaning DLT creates and maintains the `orders_valid` dataset and tracks its dependency on `orders_raw`. This is the standard Python pattern for building a streaming transformation in a Delta Live Tables pipeline. See the Databricks DLT Python reference and the documentation for streaming transformations in DLT.

QUESTION NO: 8

A data engineer uses `MERGE INTO` to apply a daily source dataset to a Delta table containing the current customer record for each customer ID.

Which two clauses can be used in the MERGE statement? (Choose two.)

A. WHEN MATCHED THEN UPDATE

B. WHEN NOT MATCHED THEN INSERT

C. WHEN DUPLICATE THEN SKIP

D. WHEN EXISTS THEN OVERWRITE

E. WHEN UNMATCHED THEN CREATE TABLE

ANSWER: A B

Explanation:

`WHEN MATCHED THEN UPDATE` can update a target row when a source row satisfies the merge condition. This is commonly used to apply changed attributes for an existing customer.

`WHEN NOT MATCHED THEN INSERT` can insert a source row when no target row satisfies the merge condition. Together, these clauses implement the common upsert pattern for maintaining a current-state Delta table.

See the Databricks MERGE INTO documentation and the MERGE INTO SQL reference.

QUESTION NO: 9

A data engineer needs to optimize the data layout and query performance for an e-commerce transactions Delta table. The table is partitioned by " purchase_date " a date column which helps with time-based queries but does not optimize searches on user statistics " customer_id " , a high-cardinality column.

The table is usually queried with filters on " customer_i

d " within specific date ranges, but since this data is spread across multiple files in each partition, it results in full partition scans and increased runtime and costs.

How should the data engineer optimize the Data Layout for efficient reads?

A. Alter table implementing liquid clustering on " customerid " while keeping the existing partitioning.

B. Alter the table to partition by " customer_id " .

C. Enable delta caching on the cluster so that frequent reads are cached for performance.

D. Alter the table implementing liquid clustering by " customer_id " and " purchase_date " .

ANSWER: D

Explanation:

Alter the table implementing liquid clustering by `customer_id` and `purchase_date`. Liquid clustering is designed to organize Delta table data incrementally according to the columns most often used to filter queries. In this workload, queries commonly constrain both the customer identifier and a date range. Clustering on those two columns colocates rows with similar values more effectively than date partitioning alone, enabling Delta data skipping to avoid reading files that cannot satisfy the `customer_id` and `purchase_date` predicates. This reduces scanned data, query runtime, and associated compute cost while retaining flexibility as the table grows or query patterns evolve.

For an existing partitioned table, the data layout should be changed to liquid clustering rather than combined with the existing partitioning layout. Databricks documents that liquid clustering replaces partitioning and `ZORDER` as the table-layout strategy, and clustering keys should be selected from columns frequently used in query filters. The table can be configured with a `CLUSTER BY (customer_id, purchase_date)` clause, after which `OPTIMIZE` can progressively recluster existing and newly written data. See the Databricks liquid clustering documentation and the Databricks data skipping documentation.

QUESTION NO: 10

Which two statements describe Databricks compute options? (Choose two.)

A. Job compute can be configured to terminate after a job run completes.

B. All-purpose compute can be used for interactive notebook development.

C. Job compute cannot execute notebook tasks.

D. Each task in a multi-task Job must use separate compute.

E. SQL warehouses must be used to execute all Databricks Jobs.

ANSWER: A B

Explanation:

Job compute is intended for automated workloads. Databricks can create it when a job run begins and terminate it after the job run completes, which helps avoid leaving compute resources running after scheduled processing is finished.

All-purpose compute is designed for interactive use, such as developing and running notebooks. Multiple users with the required permissions can attach notebooks to the same all-purpose compute resource. See the Databricks compute configuration documentation and job compute documentation.

QUESTION NO: 11

A data engineering team wants to use a Databricks SQL alert to monitor a query that returns the daily count of records rejected by a pipeline.

Which two configurations are required for the alert to notify the team when the rejected-record count exceeds 50? (Choose two.)

A. Configure an alert condition for a value greater than 50

B. Configure a notification destination for the alert

C. Create a dashboard containing the query

D. Grant MODIFY on the source table to all alert recipients

E. Enable Delta change data feed on the source table

ANSWER: A B

Explanation:

The team must configure an alert condition that evaluates the query result and triggers when the returned count exceeds 50. They must also configure at least one notification destination, such as an email recipient or a webhook destination, so Databricks has a target for delivering alert notifications.

Alerts evaluate a saved query according to the configured schedule and send notifications when the specified condition is met. A dashboard is not required to create or run an alert. See Databricks SQL alerts documentation.

QUESTION NO: 12

A data engineer is deploying a dashboard through a Declarative Automation Bundle. The dashboard resource references ${var.dataset_catalog}, and the bundle contains the following configuration:

bundle:

name: workspace_assets

variables:

dataset_catalog:

default: catalog_dev

targets:

dev:

variables:

dataset_catalog: catalog_dev

prod:

variables:

dataset_catalog: catalog_prod

Which action deploys the dashboard to the production target using catalog_prod without changing the resource definition?

A. Run databricks bundle deploy --var dataset_catalog=catalog_prod so that the CLI automatically selects targets.prod.

B. Run databricks bundle deploy --profile prod so that the CLI selects targets.prod and applies catalog_prod.

C. Run databricks bundle execute --profile prod so that the CLI selects targets.prod and applies catalog_prod.

D. Run databricks bundle deploy --target prod so that the deployment uses targets.prod and its dataset_catalog override.

ANSWER: D

Explanation:

Run `databricks bundle deploy --target prod` so that the deployment uses `targets.prod` and its `dataset_catalog` override. In a Databricks Asset Bundle, a target represents a deployment environment and can provide target-specific configuration, including variable values. Specifying `--target prod` instructs the Databricks CLI to select the configuration nested under `targets.prod`. The target-level value for `dataset_catalog`, `catalog_prod`, is then resolved for references such as `${var.dataset_catalog}` in the dashboard resource during deployment. This lets the same resource definition be reused across development and production, while the selected target supplies environment-specific settings. The `bundle deploy` command is the command that validates and deploys bundle resources to the selected target. A CLI profile controls authentication and workspace connection settings; it does not itself select a bundle target. Similarly, passing a variable with `--var` can override a variable value, but does not cause the CLI to automatically

select the named production target. See the Databricks documentation for bundle targets and the databricks bundle deploy command.

QUESTION NO: 13

A data engineer wants to create an external table in Databricks that references data stored in an Azure Data Lake Storage (ADLS) location. The goal is to enable Databricks to access and query this external data without moving it into Databricks-managed storage.

Which step should the data engineer take to successfully create the external table?

A. Use the **CREATE TABLE** statement and specify the **LOCATION** clause with the path to the external data.

B. Use the **CREATE UNMANAGED TABLE** statement without specifying a **LOCATION** clause.

C. Use the **CREATE EXTERNAL TABLE** statement without specifying a **LOCATION** clause.

D. Use the **CREATE MANAGED TABLE** statement and specify the **LOCATION** clause with the path to the external data.

ANSWER: A

Explanation:

Use the **CREATE TABLE** statement and specify the **LOCATION** clause with the path to the external data. In Databricks, a table is external when its data location is explicitly supplied using the `LOCATION` clause. The table metadata is registered in the metastore, while the underlying files remain in the specified ADLS directory; Databricks therefore queries the existing data rather than relocating it to managed table storage. For example, the statement can define a data source format and schema, then point to an ADLS URI such as `abfss://container@account.dfs.core.windows.net/path`. In a Unity Catalog-enabled environment, that path must also be covered by an external location, and the principal creating the table needs the required privileges on that external location. The external location associates the cloud path with a storage credential so Databricks can securely access ADLS. Databricks documents that specifying `LOCATION` creates an external table, while omitting it creates a managed table by default. See the CREATE TABLE documentation and the external locations documentation.

QUESTION NO: 14

A data engineer is defining data quality rules for a Lakeflow Spark Declarative Pipelines dataset.

Which two statements describe expectation behavior? (Choose two.)

A. An expectation without an ON VIOLATION action records metrics while retaining invalid records.

B. ON VIOLATION DROP ROW excludes invalid records from the target dataset.

C. ON VIOLATION FAIL UPDATE writes all valid records before failing the pipeline update.

D. An expectation automatically corrects values that violate its condition.

E. Expectations can be applied only to streaming tables.

ANSWER: A B

Explanation:

An expectation that does not specify an `ON VIOLATION` action retains records that fail the expectation in the target dataset while recording the quality metrics. This behavior is appropriate when the engineer wants to monitor a quality issue without discarding data.

An expectation configured with `ON VIOLATION DROP ROW` excludes records that fail the condition from the target dataset and records the failures in pipeline metrics. This allows valid records to continue through the pipeline while invalid records are removed from that dataset.

See the Databricks expectations documentation.

QUESTION NO: 15

Which of the following describes a scenario in which a data engineer will want to use a single-node cluster?

A. When they are working interactively with a small amount of data

B. When they are running automated reports to be refreshed as quickly as possible

C. When they are working with SQL within Databricks SQL

D. When they are concerned about the ability to automatically scale with larger data E. When they are manually running reports with a large amount of data

ANSWER: A

Explanation:

When they are working interactively with a small amount of data is the appropriate use case for a single-node cluster. A single-node compute resource has a Spark driver but no worker nodes, so processing runs locally on that one machine rather than being distributed across a cluster. This makes it a practical, lower-overhead choice for lightweight interactive development, exploration, testing, and analysis where the data volume and required compute fit comfortably within the resources of one node.

Single-node compute can still run Spark workloads and access supported data sources, including Delta tables. It is therefore useful when an engineer wants the familiar Spark APIs for reading, transforming, and writing data but does not need distributed execution. It can also be suitable for small, non-distributed machine learning or notebook-oriented tasks. The key consideration is that the workload is modest enough to run effectively within the CPU, memory, and storage available on a single driver machine. Databricks documents that single-node compute is intended for workloads that do not require distributed computing and notes its driver-only architecture. See the Databricks documentation on single-node compute and the Spark configuration documentation for configuration and runtime details.

QUESTION NO: 16

A data engineer is configuring a Unity Catalog environment. A group of analysts must query tables in the `sales` schema of the `production` catalog.

Which two privileges must be granted to the analyst group at a minimum? (Choose two.)

A. USE CATALOG on production

B. USE SCHEMA on production.sales

C. CREATE TABLE on production.sales

D. MODIFY on production.sales

E. OWN on production.sales

ANSWER: A B

Explanation:

The group needs `USE CATALOG` on the `production` catalog and `USE SCHEMA` on the `sales` schema to access the namespace containing the tables. These privileges allow the group to resolve the catalog and schema when referencing objects.

The group must also receive `SELECT` on each table it needs to query, but that privilege is not sufficient by itself because access to the parent catalog and schema is also required. See Unity Catalog privileges and securable objects.

QUESTION NO: 17

A data engineer is maintaining a Silver table that contains customer records. An incoming batch includes a new column named `loyalty_status`, and the engineer needs to add this column to the existing Delta table before loading the batch.

Which of the following SQL commands should the data engineer use?

A. CREATE TABLE customers (loyalty_status STRING);

B. ALTER TABLE customers ADD COLUMNS (loyalty_status STRING);

C. UPDATE customers SET loyalty_status = STRING;

D. INSERT INTO customers ADD COLUMN loyalty_status STRING;

ANSWER: B

Explanation:

`ALTER TABLE customers ADD COLUMNS (loyalty_status STRING);` modifies the existing table schema by adding the named column with the specified data type. After the schema change, subsequent writes can populate `loyalty_status`.

`CREATE TABLE` creates a new table rather than modifying the existing table. `UPDATE` changes row values but does not add a column, and `INSERT INTO` adds rows without changing the target table definition. See the Databricks ALTER TABLE reference.

QUESTION NO: 18

Which of the following describes the relationship between Bronze tables and raw data?

A. Bronze tables contain less data than raw data files.

B. Bronze tables contain more truthful data than raw data.

C. Bronze tables contain aggregates while raw data is unaggregated.

D. Bronze tables contain a less refined view of data than raw data.

E. Bronze tables contain raw data with a schema applied.

ANSWER: E

Explanation:

Bronze tables contain raw data with a schema applied. In the medallion architecture, the bronze layer is the initial landing layer for data ingested from source systems such as databases, files, event streams, and APIs. Its purpose is to preserve source fidelity while making the ingested data reliably available in a managed table format, commonly Delta tables in Databricks. Data is generally appended with minimal transformation rather than being cleansed, joined, deduplicated, or aggregated. A bronze table may retain the source structure and can include ingestion metadata, such as the source file name, ingestion timestamp, or batch identifier, to support traceability and replayability. Applying or capturing a schema allows the raw records to be queried and processed consistently using Spark and SQL, while still retaining the raw, historical representation needed for downstream processing. Subsequent silver and gold layers are where data is progressively

validated, refined, modeled, and aggregated for analytical use. Databricks describes the bronze layer as containing raw data from various source systems and emphasizes preserving historical data in its original form. See the Databricks medallion architecture documentation and the Databricks data ingestion documentation.

QUESTION NO: 19

Which two components function in the DB platform architecture's control plane? (Choose two.)

A. Virtual Machines

B. Compute Orchestration

C. Serverless Compute

D. Compute

E. Unity Catalog

ANSWER: B E

Explanation:

Compute Orchestration and Unity Catalog function in the Databricks control plane. The control plane hosts Databricks-managed services that provide the platform's management, coordination, and governance capabilities. Compute Orchestration is responsible for coordinating compute resources and executing control functions such as cluster lifecycle management, scheduling, and communication between the Databricks workspace services and compute resources. This centralized orchestration lets Databricks manage workloads without placing the platform-management services directly alongside customer data-processing workloads.

Unity Catalog is also a control-plane service because it supplies centralized governance for data and AI assets. It maintains and enforces metadata, permissions, lineage, auditing, and discovery across workspaces and cloud environments. By managing identities, access policies, and object definitions centrally, Unity Catalog provides the governance layer that controls how users and workloads can discover and access governed resources. Databricks documents the separation between the control plane and customer-managed compute/data-plane resources, as well as Unity Catalog's centralized governance role, in its architecture and governance documentation: Databricks platform overview and What is Unity Catalog?.

QUESTION NO: 20

The Delta transaction log for the ˜students' tables is shown using the ˜DESCRIBE HISTORY students' command. A Data Engineer needs to query the table as it existed before the UPDATE operation listed in the log.

	version	timestamp	operation
1	8	2024–04–22T14:33:31.000	OPTIMIZE
2	7	2024–04–22T14:33:16.000	MERGE
3	6	2024–04–22T14:33:06.000	DELETE
4	5	2024–04–22T14:32:58.000	UPDATE
5	4	2024–04–22T14:32:47.000	WRITE
6	3	2024–04–22T14:32:44.000	WRITE
7	2	2024–04–22T14:32:23.000	WRITE
8	1	2024–04–22T14:32:20.000	WRITE
9	0	2024–04–22T14:31:49.000	CREATE TABLE

Which command should the Data Engineer use to achieve this? (Choose two.)

A. SELECT * FROM students@v4

B. SELECT * FROM students TIMESTAMP AS OF '2024-04-22T14:32:47.000+00:00'

C. SELECT * FROM students FROM HISTORY VERSION AS OF 3

D. SELECT * FROM students VERSION AS OF 5

E. SELECT * FROM students TIMESTAMP AS OF '2024-04-22T14:32:58.000+00:00'

ANSWER: A B

Explanation:

Delta Lake time travel lets a query read the snapshot of a table at a specific committed version or at a point in time. `SELECT * FROM students@v4` uses the supported shorthand version-reference syntax and explicitly reads version 4 of the `students` table. Because version 4 is the committed snapshot immediately preceding the logged UPDATE, it returns the data as it existed before that operation.

`SELECT * FROM students TIMESTAMP AS OF '2024-04-22T14:32:47.000+00:00'` is also valid Delta time-travel syntax. A timestamp reference resolves to the latest table version committed at or before that timestamp. The supplied timestamp corresponds to the pre-update state in the displayed history, so this query likewise reads the snapshot before the UPDATE was committed. Version-based time travel is particularly precise when the required transaction-log version is known, while timestamp-based time travel is useful when the desired state is identified by its commit time.

Both forms depend on the historical Delta log files and data files still being retained; aggressive vacuuming can remove files needed to query older snapshots. Databricks documents version and timestamp time travel, including the `@v` shorthand, in its Delta table history and time travel documentation. See also the Databricks SQL SELECT reference.

QUESTION NO: 21

Which SQL code snippet will correctly demonstrate a Data Definition Language (DDL) operation used to create a table?

A. DROP TABLE employees;

B. INSERT INTO employees (id, name) VALUES (1, ' Alice ');

C. CREATE TABLE employees (id INT, name STRING);

D. ALTER TABLE employees ADD COLUMN salary DECIMAL(10,2);

ANSWER: C

Explanation:

`CREATE TABLE employees (id INT, name STRING);` is the correct DDL statement because `CREATE TABLE` defines a new table and specifies its schema. In this example, the table is named `employees`, with an `id` column using the integer data type and a `name` column using Databricks SQL's `STRING` data type. DDL commands manage the structure of database objects, including creating tables, defining columns, and setting table properties. In Databricks SQL, a basic `CREATE TABLE` statement can create a table by supplying a table name followed by a parenthesized list of column definitions. The statement shown uses valid SQL keywords and valid Databricks SQL data types, so it correctly demonstrates creation of a table. For additional optional behavior, Databricks supports clauses such as `IF NOT EXISTS`, `USING DELTA`, and `LOCATION`, but none is required for this basic managed-table example. See the Databricks CREATE TABLE reference and the SQL data types reference for the supported syntax and types.

QUESTION NO: 22

A data engineer is working on a personal laptop and needs to perform complex transformations on data stored in a Delta Lake on cloud storage. The engineer decides to use Databricks Connect to interact with Databricks clusters and work in their local IDE.

How does Databricks Connect enable the engineer to develop, test, and debug code seamlessly on their local machine while interacting with Databricks clusters?

A. By allowing direct execution of Spark jobs from the local machine without needing a network connection

B. By providing a local environment that mimics the Databricks runtime, enabling the engineer to develop, test, and debug code using a specific IDE that is required by Databricks

C. By providing a local environment that mimics the Databricks runtime, enabling the engineer to develop, test, and debug code using their preferred ide

D. By providing a local environment that mimics the Databricks runtime, enabling the engineer to develop, test, and debug code only through Databricks ' own web interface

E. By connecting code run from the engineer's preferred local IDE to a remote Databricks cluster, where Spark processing is executed.

ANSWER: E

Explanation:

Databricks Connect enables local development by allowing code written and run from the engineer's preferred local IDE to establish a connection to a remote Databricks compute resource. The local application uses the Databricks Connect client libraries and a configured workspace authentication method, while Spark operations are executed by the Databricks cluster or serverless compute. This gives the engineer an interactive development workflow on the laptop—such as setting breakpoints, using local tooling, and iterating on Python code—while retaining access to the scalable Databricks runtime and the Delta Lake data available from the remote compute environment. It is therefore not an offline execution model or a local replica of the complete Databricks Runtime. A network connection and an available configured Databricks compute resource are required for commands that interact with Spark. Databricks Connect is specifically intended to support development, testing, and debugging from familiar IDEs, including tools such as VS Code and PyCharm, while delegating distributed data processing to Databricks. See the Databricks Connect for Python documentation and the Databricks Connect overview.

QUESTION NO: 23

A data engineer is getting a partner organization up to speed with Databricks account. Both teams share some business use cases. The data engineer has to share some of your Unity-Catalog managed delta tables and the notebook jobs creating those tables with the partner organization.

How can the data engineer seamlessly share the required information?

A. Zip all the code and share via email and allow data ingestion from your data lake

B. Data and Notebooks can be shared simply using Unity Catalog.

C. Share access to codebase via Github and allow them to ingest datasets from your Datalake.

D. Share required datasets and notebooks via Delta Sharing. Manage permissions via Unity Catalog.

ANSWER: D

Explanation:

Share required datasets and notebooks via Delta Sharing. Manage permissions via Unity Catalog. Delta Sharing is Databricks' open sharing protocol for securely sharing live data with organizations outside the provider's Databricks account, without copying data to email attachments, source-control repositories, or a separate data lake location. A provider can use Unity Catalog to create a share, add the required Unity Catalog managed Delta tables, and grant access to a recipient organization. The recipient accesses the shared tables as governed data, while the provider remains in control of the underlying data and can revoke or modify access when needed.

Databricks-to-Databricks sharing also supports sharing notebooks through Delta Sharing, allowing the partner to receive the notebook assets associated with the shared use case. Unity Catalog supplies the centralized governance model for these shared assets: it manages shares and recipients and applies permission controls and auditing. This approach is therefore seamless because it provides a native Databricks mechanism for distributing both the governed datasets and the relevant

notebook content across organizational boundaries, rather than requiring manual exports or separate access arrangements. See the Databricks documentation for Delta Sharing and managing Delta Sharing shares, recipients, and permissions.

QUESTION NO: 24

In which of the following scenarios should a data engineer use the MERGE INTO command instead of the INSERT INTO command?

A. When the location of the data needs to be changed

B. When the target table is an external table

C. When the source table can be deleted

D. When the target table cannot contain duplicate records

E. When the source is not a Delta table

ANSWER: D

Explanation:

When the target table cannot contain duplicate records is the appropriate scenario for using `MERGE INTO`. In a Delta Lake pipeline, incoming data often includes records for entities that may already exist in the target, such as customers, orders, devices, or account balances. Rather than blindly appending every incoming row, a merge uses a match condition—normally a business key—to identify whether a source record corresponds to an existing target record. It can then update the existing record when a match is found and insert the record only when no match exists. This provides the core upsert pattern needed to maintain one current target row per key and supports repeatable, incremental ingestion.

`MERGE INTO` also supports conditional update, insert, and delete clauses, enabling data engineers to express synchronization logic directly against a Delta table. To maintain correct results, the merge condition should identify target rows unambiguously, and source data should be deduplicated when multiple source records could match the same target row. Databricks documents `MERGE INTO` as the Delta Lake operation for combining source changes with a target table using matched and unmatched clauses. See the Databricks MERGE INTO documentation and the Delta Lake merge (upsert) documentation.

QUESTION NO: 25

A data engineer is developing an ETL process based on Spark SQL. The execution fails. The data engineer checks the Spark UI and can see the ERRORS as follows:

Which two corrective actions should the data engineer perform to resolve this issue?

Choose 2 answers - (Q) Narrow the filters in order to collect less data in the query

A. Upsize the worker nodes and activate autoshuffle partitions

B. Upsize the driver node and deactivate autoshuffle partitions

C. Cache the dataset in order to boost the query performance

D. Fix the shuffle partitions to 50 to ensure the allocation

E. Narrow the filters in order to collect less data in the query

ANSWER: A E

Explanation:

"Upsize the worker nodes and activate autoshuffle partitions" is appropriate because Spark SQL execution distributes transformations and shuffle work across executor processes on worker nodes. When the failure is caused by insufficient executor resources or inefficiently sized shuffle partitions, workers with more available memory and compute capacity provide the resources needed to process larger partitions. Databricks' auto-optimized shuffle, used with Adaptive Query Execution, can select a suitable shuffle partition count at runtime based on observed shuffle data rather than relying on a static setting. This helps avoid partitions that are disproportionately large and can put excessive pressure on individual executors. "Narrow the filters in order to collect less data in the query" also directly reduces the volume of records scanned, shuffled, and retained during execution. Applying selective filters as early as possible lowers the amount of data each worker must process and is a fundamental Spark SQL optimization for resource-constrained workloads. Together, reducing the input workload and allowing the cluster to scale processing and shuffle handling addresses the execution pressure indicated by the Spark UI errors. See Databricks Adaptive Query Execution documentation and Spark SQL performance tuning documentation.

QUESTION NO: 26

A data engineer is developing a Structured Streaming query that aggregates events by customer over a time window. The source contains event-time data that can arrive late.

Which two statements describe how a watermark can help with this streaming query? (Choose two.)

A. It defines how long the query waits for late event-time data.

B. It allows old state to be removed from stateful aggregations.

C. It guarantees that all late records are processed.

D. It eliminates the need for a checkpoint location.

E. It converts a streaming query into a batch query.

ANSWER: A B

Explanation:

A watermark defines how long the streaming query should wait for late data based on an event-time column. It also enables Spark to remove old aggregation state once records are older than the configured watermark threshold, which helps bound the state that must be retained for stateful operations.

A watermark does not guarantee that every late record will be included. Records arriving later than the threshold can be dropped, so the configured delay should reflect the expected lateness of the source data. See the Apache Spark Structured Streaming watermarking documentation.

QUESTION NO: 27

A data analysis team uses a SQL warehouse for many concurrent short-running queries.

Which two warehouse configuration changes can help the warehouse serve more concurrent query demand? (Choose two.)

A. Increase the maximum number of clusters in the warehouse scaling range.

B. Increase the minimum number of clusters in the warehouse scaling range.

C. Enable auto stop for a shorter idle period.

D. Reduce the warehouse size.

E. Decrease the maximum number of clusters in the warehouse scaling range.

ANSWER: A B

Explanation:

Increasing the maximum number of clusters in the warehouse scaling range permits the warehouse to add more clusters as concurrent demand increases. This can reduce queuing when many users submit queries at the same time.

Increasing the minimum number of clusters keeps more capacity available before a demand increase occurs. This can reduce latency during predictable periods of sustained concurrent usage, although it can also increase the cost of idle capacity.

Auto stop is primarily a cost-management setting for idle warehouses and does not increase query concurrency. Reducing warehouse size or lowering the maximum cluster count reduces available capacity. See the Databricks SQL warehouse behavior documentation.

QUESTION NO: 28

A data engineer is creating a multi-task Databricks Job. A downstream notebook task should run only after two upstream tasks have both completed successfully.

Which two actions should the data engineer take? (Choose two.)

A. Add the first upstream task as a dependency of the downstream task

B. Add the second upstream task as a dependency of the downstream task

C. Set the downstream task to use a separate job schedule

D. Configure the downstream task with the At least one succeeded run-if condition

E. Run all three tasks concurrently

ANSWER: A B

Explanation:

The engineer should add both upstream tasks as dependencies of the downstream notebook task. The downstream task should use the default `All succeeded` run-if condition so that it begins only when each dependency completes successfully.

Task dependencies define the workflow order in a Databricks Job. If either upstream task fails, the downstream task will not run under the `All succeeded` condition. See Databricks task dependency documentation.

QUESTION NO: 29

A data engineer needs to create a table in Databricks using data from their organization's existing SQLite database. They run the following command:

CREATE TABLE jdbc_customer360

USING

OPTIONS (

url "jdbc:sqlite:/customers.db", dbtable "customer360"

)

Which line of code fills in the above blank to successfully complete the task?

A. autoloader

B. org.apache.spark.sql.jdbc

C. sqlite

D. org.apache.spark.sql.sqlite

E. jdbc

ANSWER: E

Explanation:

The correct completion is `jdbc`, resulting in `CREATE TABLE jdbc_customer360 USING jdbc OPTIONS (...)`. Databricks SQL and Spark SQL use `JDBC` as the built-in data source provider name for creating a table backed by, or reading from, a JDBC-accessible relational database. The `url` option supplies the database connection string—in this case, an SQLite JDBC URL—and `dbtable` identifies the source table to query. When Spark resolves `USING jdbc`, it invokes its JDBC data source and passes these options to the JDBC connection. A suitable SQLite JDBC driver must also be available on the Databricks compute resource for the SQLite URL to be handled successfully. Databricks documents JDBC as a supported data source format and shows the provider specified using `USING JDBC`, with connection details such as `url` and `dbtable` supplied in the options clause. The concise provider identifier is the appropriate SQL syntax; it is case-insensitive, so `jdbc` is valid. See the Databricks CREATE TABLE USING data source documentation and the Apache Spark JDBC data source guide.

QUESTION NO: 30

A data engineer is configuring an Auto Loader stream that ingests JSON files. The source schema can evolve over time, and the engineer wants newly discovered columns to be captured for later processing without immediately adding them to the target table schema.

Which two actions can the data engineer take? (Choose two.)

A. Set cloudFiles.schemaLocation to a persistent storage location

B. Set cloudFiles.schemaEvolutionMode to rescue

C. Run VACUUM after each ingestion batch

D. Set mergeSchema to false on the target write

E. Create a SQL warehouse for the input directory

ANSWER: A B

Explanation:

The engineer can configure a schema location using `cloudFiles.schemaLocation`. Auto Loader uses this location to persist inferred schema information and track schema evolution across runs. The engineer can also set `cloudFiles.schemaEvolutionMode` to `rescue`, which places fields that do not match the known schema into the rescued data column rather than adding them directly to the schema.

This pattern allows ingestion to continue while preserving unexpected fields for review. The rescued data can then be evaluated before the target schema is deliberately updated. See Databricks Auto Loader schema inference and evolution documentation.