Databricks Certified Data Engineer Associate Study Guide

This study guide provides a brief overview of the topics covered in the Databricks Certified Data Engineer Associate online course. There are 45 multiple-choice questions on the certification exam. The questions will be distributed by high-level topic in the following way:

Databricks Lakehouse Platform – 24% (11/45)
ELT with Spark SQL and Python – 29% (13/45)
Incremental Data Processing – 22% (10/45)
Production Pipelines – 16% (7/45)
Data Governance – 9% (4/45)

Following the breakdown of sections within the exam listed above, all of the major topics covered on the exam are further split into the subcategories highlighted for the *minimally qualified candidate* on the certification page for the course (as of May 2022). The links in the table of contents below can be used to quickly navigate to a specific section to review a specific topic. Additionally, there is a link included in the footer of every page to return to the table of contents for easy navigation. Feel free to contact me with any questions on Microsoft Teams or at devmisra@kpmg.com.

Databricks Lakehouse Platform

Understand how to use and the benefits of using the Databricks Lakehouse Platform and its tools, including:

Data Lakehouse (architecture, descriptions, benefits)

- Architecture
    - Different data types (structured, semi-structured, unstructured, streaming, etc.) come into the platform and land in open data lake
    - Delta Lake is added to the data lake, providing an added layer that adds structure and governance to data
    - Data is then used to drive business use-cases in data engineering, BI and SQL analytics, real-time data applications, and data science and machine learning
- Combines capabilities of data lakes and warehouses
    - Uses similar data structures and data management features to those in a data warehouse directly on low cost storage used for data lakes
    - Allows data teams to move faster as they can use data without accessing multiple systems
- Transaction support to ensure that multiple parties can concurrently read or write data
- Data schema enforcement to ensure data integrity (writes to a table are rejected if they do not match the table's schema)
- Governance and audit mechanisms to make sure you can see how data is being used
- BI support so that BI tools can work directly on source data - this reduces data staleness
- Storage is decoupled from compute, which means that it is easier for your system to scale to more concurrent users and data sizes
- Openness - Storage formats used are open and standard. Plus, APIs and various other tools make it easy for team members to access data directly
- Support for all data types - structured, unstructured, semi-structured
- End-to-end streaming so that real-time reporting and real-time data can be integrated into data analytics processes just as existing data is
- Support for diverse workloads, including data engineering, data science, machine learning, and SQL analytics - all on the same data repository

Data Science and Engineering workspace (clusters, notebooks, data storage)

- A database cluster is a set of computation resources and configurations on which you run data engineering, data science, and data analytics workloads
    - ETL pipelines, streaming analytics, ad-hoc analytics, machine learning, etc.
- Clusters are made up of one or more virtual machine (VM) instances that consist of:

    - A driver that coordinates activities of executors
    - Executors that run the tasks composing a spark job
- These workloads are run as a set of commands in a notebook or as an automated job. Databricks makes a distinction between all-purpose clusters and job clusters
    - All-purpose clusters are used to analyze data using interactive notebooks
        - Can be created using the UI, CLI, or REST API
        - Can be manually terminated and restarted
    - Job clusters are used to run fast and robust automated jobs
        - Databricks job scheduler creates a job cluster when you run a job on a new job cluster and terminates the cluster when the job is complete
        - Cannot restart a job cluster

Delta Lake (general concepts, table management and manipulation, optimizations)

- An open approach to bringing data management and governance to data lakes

- Delta Lake follows an architecture pattern that consists of organizing ingested data into successively cleaner tables (bronze, silver, gold), which can then be leveraged for business use-cases such as Streaming Analytics and AI & Reporting
- Elements of Delta Lake
    - Delta Files
        - Delta Lake uses Parquet files to store a customer’s data in the customer's cloud storage account
        - Parquet is a columnar file format that provides optimizations to speed up queries and is a far more efficient file format than CSV or JSON (the columnar storage allows you to quickly skip over non-relevant data while executing queries)
        - Delta files leverage all of the technical capabilities of Parquet files but have an additional layer over them. This additional layer tracks data versioning and metadata, stores transaction logs to keep track of changes made to a data table or object storage directory, and provides ACID transactions
    - Delta Tables
        - Collection of data kept using Delta Lake technology consisting of:
            - Delta files containing the data and kept in object storage
            - A Delta table registered in a Metastore (a metastore is simply a catalog that tracks your data’s metadata - data about your data)
            - The Delta Transaction Log saved with Delta files in object storage
    - Delta Optimization Engine
        - High-performance query engine that provides an efficient way to process data in data lakes

        - Accelerates data lake operations and supports variety of workloads
    - Delta Lake Storage Layer
        - Organization stores data in a Delta Lake Storage Layer and then accesses that data via Databricks
        - Organization keeps all of this data in files in object storage, which is beneficial because data is kept in a lower-cost and easily scalable environment
- How Delta Lake addresses common Data Lake challenges:
    - ACID Transactions
        - Atomicity, Consistency, Isolation, Durability
        - Each transaction is handled as having a distinct beginning and end
            - Data in a table is not updated until a transaction successfully completes
            - Each transaction will either succeed or fail fully
    - Schema Management
        - Delta Lake gives the ability to specify and enforce data schema
        - Automatically validates that schema of data being written is compatible with schema of table it is being written into
    - Scalable Metadata Handling
        - With Delta Lake, metadata is processed just like regular data - with distributed processing
    - Unified Batch and Streaming Data
        - Delta Lake is designed from the ground up to allow a single system to support both batch and stream processing of data
        - Each micro-batch transaction creates a new version of a table that is instantly available for insights
    - Data Versioning and Time Travel
        - Transaction logs used to ensure ACID compliance create an auditable history of every version of the table, indicating which files have changed between versions
        - Easy to retain historical versions of data to fulfill compliance requirements such as GDPR and CCPA
    - Improve ETL pipelines
        - Data pipelines are simplified through streamlined development, improved data reliability, and cloud-scale production operations

ETL with Spark SQL and Python

Build ETL pipelines using Apache Spark SQL and Python, including:

Relational entities (databases, tables, views)

- Databases
    - Create databases in specific locations
        - CREATE DATABASE IF NOT EXISTS database LOCATION
    - Retrieve locations of existing databases
        - DESCRIBE DATABASE EXTENDED database
    - Modify and delete databases
        - DROP DATABASE database CASCADE;
- Tables
    - Managed vs external tables
        - Managed table - default location
        - External (unmanaged table) - specified location
    - Create and drop managed and external tables
        - CREATE OR REPLACE TEMPORARY VIEW temp_view USING CSV OPTIONS options
        - CREATE OR REPLACE TABLE external_table LOCATION location AS
SELECT * FROM temp_view
        - DROP TABLE external_table
- Views and CTEs
    - Views vs temporary views
        - Temp views are tied to a Spark session and are not accessible after losing connection to a cluster, restarting the Python interpreter, or from another notebook
        - Exception to this are global views, which are added to global_temp database that exists on the cluster
    - Views vs Delta Lake tables
        - Views query data each run (SELECT statement) while tables store data
    - Creating views and CTEs
        - CREATE VIEW view AS
SELECT * FROM table WHERE criteria
        - CREATE TEMPORARY VIEW view AS
SELECT * FROM table WHERE criteria
        - CREATE GLOBAL TEMPORARY VIEW view AS
SELECT * FROM table WHERE criteria

- Common Table Expression (CTE) - think of it as a query whose results are stored as an instance of a variable within a query
- WITH cte(field1, field2) AS (SELECT 1, 2 FROM TABLE)
  SELECT * FROM cte WHERE field1 criteria

ETL (creating tables, writing data to tables, cleaning data, combining and reshaping tables, SQL UDFs)

- Creating Tables, Different file formats and data sources, Create Table as Select (CTAS) Statements
  - External sources vs Delta Lake tables
    - CREATE TABLE table USING data_source OPTIONS (key1=val1) LOCATION
    - CREATE TABLE csv_table USING CSV
    - CREATE TABLE jdbc_table USING JDBC
  - Methods to create tables and use cases
    - CREATE OR REPLACE TABLE table AS SELECT * FROM parquet
      - Well defined schema
    - CREATE OR REPLACE TEMP VIEW sales_temp (field DOUBLE) USING CSV OPTIONS (options)
      CREATE TABLE sales_delta AS SELECT * FROM sales_temp
      - Not well defined schema, so temp view used to create options
  - Delta table configurations
    - Filtering and Renaming columns from existing tables
      - CREATE OR REPLACE TABLE table AS
        SELECT field1 AS 1 FROM source
    - Generated columns
      - Special column whose values are automatically generated based on a user-specified function over other columns in Delta table
    - Constraints
      - NOT NULL constraints
        - Indicates that values in specific columns cannot be null
      - CHECK constraints
        - Indicates that a specified Boolean expression must be true for each input row
        - ALTER TABLE t ADD CONSTRAINT a CHECK (count > 1)
    - Enrich Tables with Additional Options and Metadata
      - current_timestamp(), input_file_name()
      - COMMENT, LOCATION, PARTITIONED BY
- Writing Data to Tables
  - Methods to write to tables and use cases, Efficiency for different operations, Resulting behaviors in target tables

- Complete overwrite (CREATE OR REPLACE TABLE)
  - Overwriting is much faster than deleting and recreating because directory does not need to be listed recursively/files do not need to be deleted
  - The old version of table still exists and can be retrieved after overwriting
  - Atomic operation (supports concurrent operations/queries while overwriting) and ACID transaction guarantees (if overwrite fails, table remains in previous state)
- INSERT OVERWRITE
  - Can only overwrite an existing table/individual partitions with new records that match table schema
- Append rows (INSERT INTO)
  - Allows for incremental updates into existing tables, which is much more efficient than complete overwrites
  - Does not prevent addition of duplicate records
- MERGE INTO
  - Apply updates, inserts, and deletes in a single transaction from another source table, view or DataFrame
  - Multiple conditionals can be added
  - Extensive options for implementing custom logic
  - MERGE INTO target a USING source b ON merge condition
    WHEN MATCHED THEN matched action
    WHEN NOT MATCHED THEN not_matched_action
- Insert-Only Merge for Deduplication
  - Merge used to avoid inserting duplicate records
  - MERGE INTO with only one clause of WHEN NOT MATCHED with INSERT * (Insert-Only Merge)
- COPY INTO
  - Schema must be consistent
  - Duplicate records should be excluded/handled downstream
  - Operation is potentially much cheaper than full table scans for data that grows predictably
  - No check for duplicates

- Cleaning Data with Common SQL
  - Methods to deduplicate data
    - Commands
      - count(col)
        - Count of columns/expressions, skips NULL values

      - count(*)
        - Counts total number of rows, including NULL
      - count_if
        - value IS NULL to count NULL values
      - DISTINCT
        - Distinct values, does not omit NULL rows
    - Deduplicate rows
      - CREATE OR REPLACE TEMP VIEW deduped AS
        SELECT DISTINCT(*) FROM users
    - Deduplicate based on specific columns
      - SELECT COUNT(DISTINCT(col1))
        FROM table WHERE col1 IS NOT NULL
    - GROUP BY
      - max() is used as a hack to capture non-null values from strings
      - CREATE OR REPLACE TEMP VIEW deduped AS
        SELECT col1, col 2, max(col3) AS column3 FROM table
        WHERE col1 IS NOT NULL
        GROUP BY col1, col2
  - Common cleaning methods for different types of data
    - SELECT *,
        date_format(first_touch, "MMM d, yyyy") AS first_touch_date,
        date_format(first_touch, "HH:mm:ss") AS first_touch_time,
        regexp_extract(email, "(?<=@).+", 0) AS email_domain
      FROM (
       SELECT *,
         CAST(user_first_touch_timestamp / 1e6 AS timestamp) AS first_touch
      FROM deduped)
- Combining Data
  - Join types and strategies
    - Spark SQL supports standard join operations (inner, outer, left, right, anti, cross, semi)
      - CREATE OR REPLACE VIEW view AS SELECT * FROM (
        SELECT *, explode (items) as item FROM sales) a
        INNER JOIN item_lookup b
        ON a.item.item_id = b.item_id
  - Set operators and applications
    - Spark SQL supports UNION, MINUS, and INTERSECT set operators
      - UNION returns the collection of two queries

        - INTERSECT returns all rows found in both relations
        - MINUS returns all rows found in one dataset but not the other
            - SELECT * FROM table1
              UNION/INTERSECT/MINUS
              SELECT * FROM table2
- Reshaping Data
    - Different operations to transform arrays
        - Interact with subfields using . syntax
            - SELECT field.subfield FROM table WHERE field.subfield IS/IS NOT criteria
        - explode function to put each element in an array on its own row
            - SELECT f1, f2, explode(f3) AS field3 FROM table
        - collect_set function can collect unique values for a field
            - collect_set(event_name) AS event_history
        - flatten function allows multiple arrays to be combined into a single array
            - flatten(collect_set(items.itemID)) AS item_history
        - array_distinct function remove duplicate elements from an array
            - array_distinct(flatten(collect_set(items.itemID))) AS item_history
    - Benefits of array functions
        - Prevents SQL engineers from having to rely on custom logic when dealing with highly complex data structures
    - Applying higher order functions
        - FILTER
            - Filters an array using the given lambda function
            - FILTER (items, i -> i.item_id LIKE "%K") AS king_items
        - EXIST
            - Tests if a statement is true for one or more elements in an array
        - TRANSFORM
            - Uses the given lambda function to transform all elements in an array
            - TRANSFORM(king_items, k -> CAST(k.item_revenue_in_usd * 100 AS INT)) AS item_revenues
        - REDUCE
            - Takes two lambda functions to reduce the elements of an array to a single value by merging the elements into a buffer, and then applying a finishing function on the final buffer
- Advanced Operations
    - Manipulating nested data fields
        - : syntax to traverse nested data structures

      - SELECT value:device, value:geo:city
        FROM table
    - From_json function to parse JSON objects into struct types
      - Schema_of_json function to derive JSON schema from an example
        - SELECT from_json(value, schema_of_json(schema)) AS json
          FROM table
    - * (star) unpacking to flatten fields into columns
      - CREATE OR REPLACE TEMP VIEW temp AS
        SELECT json.* FROM table
  - Applying SQL UDFs for custom transformations
    - User defined function that requires a function name, optional parameters, the type to be returned, and some custom logic
      - CREATE OR REPLACE FUNCTION yelling(text STRING)
        RETURNS STRING
        RETURN concat(upper(text), "!!!")
      - SELECT yelling(food) FROM foods
    - SQL UDFs will persist between execution environments (which can include notebooks, DBSQL queries, and jobs)
    - DESCRIBE FUNCTION [EXTENDED]
    - Governed by the same Table ACLs as databases, tables, or views
    - CASE/WHEN
      - Allows for evaluation of multiple conditions statements with alternative outcomes based on table contents

        -
          ```
          SELECT *,
            CASE
                    WHEN food = “pizza” THEN “I like pizza”
                    WHEN food = “peas” THEN “I like peas”
                    ELSE concat(“I don’t like “, food)
            END
          FROM foods
          ```
    - Combining SQL UDFS with control flow in the form of CASE/WHEN clauses provides optimized execution for control flows within SQL workloads

Python (facilitating Spark SQL with string manipulation and control flow, passing data between PySpark and Spark SQL)

- Spark SQL Queries
  - Executing Spark SQL Queries in Pyspark
    - Run SQL in a Python cell by passing the string query to spark.sql()

            - query = “SELECT * FROM table”
              result = spark.sql(query)
              display(result)
    - Passing Data to SQL
        - Temporary views
            - tempDF.createOrReplaceTempView("AirportCodes")
        - Converting tables to and from DataFrames
            - Converting table to DataFrame
                - df = spark.sql(f"SELECT '{course}' AS course_name")
                  display(df)
            - Converting DataFrame to table
                - tempDF.write.saveAsTable("tbl_AirportCodes")
    - Python Syntax
        - Functions and variables
            - F-strings
                - By adding the letter f before a Python string, you can inject variables or evaluated Python code by inserted them inside curly braces ({})
                    - f"I can substitute {my_string} here"
            - if/elif/else statements
                - Used to evaluate whether or not certain conditions are true in execution environment
                - Python control flow should be reserved for evaluating conditions outside of query
                    - If seeking to evaluate conditions within tables or queries, use CASE/WHEN
            - assert statements
                - If true, nothing happens. If false, an error is raised.
                    - assert type(2) == int → nothing happens
                    - Assert type(“2”) == int → AssertionError
            - Check if string can be safely cast as numeric value
                - assert “2”.isnumeric()
            - Cast string into numeric
                - int() for whole numbers, float() for decimals
            - try/except statements
                - Errors stop execution of notebook script, all cells after an error will be skipped when notebook is scheduled as production job
                - When error occurs in try statement, alternate logic can be defined in the except statement

- Control flow
    - Functions with if/elif/else

        -
          ```
          def foods_i_like(food):
             if food == "beans":
                print(f"I love {food}")
             elif food == "potatoes":
                print(f"My favorite vegetable is {food}")
             else:
                print(f"I don't eat {food}")
          ```
- Error handling
    - Functions with try/except

        -
          ```
          def try_int(num_string):
             try:
                int(num_string)
                result = f"{num_string} is a number."
             except:
                result = f"{num_string} is not a number!"
             print(result)
          ```
    - find() method to test for specific values
        - Returns -1 if value is not found

        -
          ```
          injection _query =
          "SELECT * FROM demo_tmp_vw; DROP DATABASE prod_db CASCADE;
          SELECT * FROM demo_tmp_vw"
          ```
        - `injection_query.find(“;”)`
    - Simple search function for error handling

        -
          ```
          def injection_check(query):
             semicolon_index = query.find(";")
             if semicolon_index >= 0:
                raise ValueError(f"Query contains semicolon at index
          {semicolon_index}\nBlocking execution to avoid SQL injection attack")
          ```

Incremental Data Processing

Incrementally process data, including:

Structured Streaming (general concepts, triggers, watermarks)

- General Concepts
  - Basic Concepts
    - Spark Structured Streaming allows users to interact with ever-growing data sources as if they were just a static table of records
    - A data stream describes any data source that grows over time, such as:
      - A new JSON log file landing in cloud storage
      - Updates to a database captured in a CDC feed
      - Events queued in a pub/sub messaging feed
      - A CSV file of sales closed the previous day
    - Structured Streaming lets us define a query against the data source and automatically detect new records and propagate them through previously defined logic
    - Spark Structured Streaming is optimized on Databricks to integrate closely with Delta Lake and Auto Loader
  - Programming Model
    - The developer defines an input table by configuring a streaming read against a source. The syntax for doing this is similar to working with static data
    - A query is defined against the input table. Both the DataFrames API and Spark SQL can be used to easily define transformations and actions against the input table
    - This logical query on the input table generates the results table. The results table contains the incremental state information of the stream
    - The output of a streaming pipeline will persist updates to the results table by writing to an external sink. Generally, a sink will be a durable system such as files or a pub/sub messaging bus
    - New rows are appended to the input table for each trigger interval. These new rows are essentially analogous to micro-batch transactions and will be automatically propagated through the results table to the sink
  - Configurations for streaming reads and writes
    - Reading a Stream
      - spark.readStream() method returns a DataStreamReader used to configure and query the stream
      - (spark.readStream

```
.table("bronze")
.createOrReplaceTempView("streaming_tmp_vw"))
```

- A query on a streaming temp view will continue to update as new data arrives in the source (always-on incremental query)
- Writing a stream
    - DataFrame.writeStream method returns a DataStreamWriter used to configure the output
    - 3 main settings
        - Checkpointing (discussed below)
        - Output Modes
            - Append (default) - .outputMode(“append”)
                - Only newly appended rows are incrementally appended to target table with each batch
            - Complete - .outputMode(“complete”)
                - Results Table is recalculated each time a write is triggered; the target table is overwritten with each batch
        - Trigger Intervals (discussed below)
- Requirements for end-to-end fault tolerance
    - Structured streaming allows for end-to-end exactly once fault tolerance guarantees through checkpointing and write ahead logs
    - Checkpointing
        - Databricks creates checkpoints by storing the current state of the streaming job to cloud storage
        - Checkpointing combines with write ahead logs to allow a terminated stream to be restarted and continued from where it left off
        - Checkpoints cannot be shared between separate streams
        - A checkpoint is required for every streaming write to ensure processing guarantees
- Interacting with streaming queries
    - Most transformations that can be applied to static data can be applied to streaming temp views
    - When querying a streaming temp view, a streamy query that executes indefinitely is created (rather than completing after retrieving a single set of results)
    - For such streaming queries, Databricks Notebooks include interactive dashboards that allow users to monitor streaming performance

- Click Stop Execution at the top of the notebook or Cancel under the cell to stop all active streaming queries

- Triggers
  - Setting up streaming writes with different trigger behaviors
    - The trigger method specifies when the system should process the next set of data
    - Triggers are specified when defining how data will be written to a sink and control the frequency of micro-batches. By default, Spark will automatically detect and process all data in the source that has been added since the last trigger
    - Unspecified
      - This is the default. This is equivalent to using processingTime="500ms"
    - Fixed interval micro-batches
      - .trigger(processingTime=”2 minutes”)
      - The query will be executed in micro-batches and kicked off at the user-specified intervals
    - Triggered micro-batch
      - .trigger(once=True)
      - The query will execute a single micro-batch to process all the available data and then stop on its own
    - Triggered micro-batches
      - .trigger(availableNow=True)
      - The query will execute multiple micro-batches to process all the available data and then stop on its own
- Watermarks
  - Unsupported operations on streaming data
    - Sorting is one, full discussion of exceptions is out of scope for course
  - Scenarios in which watermarking data would be necessary
    - Advanced streaming methods like windowing and watermarking can be used to add additional functionality to incremental workloads
  - Managing state with watermarking
    - Watermarking lets the engine automatically track the current event time in the data and attempt to clean up the old state accordingly
    - The watermark of a query is defined by specifying the event time column and the threshold on how late the data is expected to be in terms of event time
    - A watermark delay (set with withWatermark) of “2 hours” guarantees that the engine will never drop any data that is less than 2 hours delayed. In other words, any data less than 2 hours behind (in terms of event-time) the latest data

processed till then is guaranteed to be aggregated

Auto Loader (streaming reads)

- Incrementally process data to power analytic insights with Spark Structured Streaming and Auto Loader
  - ○ Streaming Reads
    - ■ Define streaming reads with Auto Loader and Pyspark to load data into Delta
      - ● 4 main arguments
        - ○ data_source - The directory of the source data
          - ■ Auto Loader will detect new files as they arrive in this location and queue them for ingestion; passed to the .load() method
        - ○ source_format - The format of the data source
          - ■ While the format for all Auto Loader queries will be cloudFiles, the format of the source data should always be specified for the cloudFiles.format option
        - ○ table_name - The name of the target table
          - ■ Spark Structured Streaming supports writing directly to Delta Lake tables by passing a table name as a string to the .table() method. Note that you can either append to an existing table or create a new table
        - ○ checkpoint_directory - The location for storing metadata about the stream
          - ■ This argument is pass to the checkpointLocation and cloudFiles.schemaLocation options. Checkpoints keep track of streaming progress, while the schema location tracks updates to the fields in the source dataset
        - ○ Example:
          - ■
            ```
            query = autoload_to_table(data_source =
            f"{DA.paths.working_dir}/tracker",
                                      source_format = "json",
                                      table_name = "target_table",
                                      checkpoint_directory =
            f"{DA.paths.checkpoints}/target_table")
            ```
    - ■ Define streaming reads on tables for SQL manipulation
      - ●
        ```
        customers_count_checkpoint_path =
        f"{DA.paths.checkpoints}/customers_count"
        ```
      - ●
        ```
        query = (spark.table("customer_count_by_state_temp")
                 .writeStream
                 .format("delta")
        ```

```
.option("checkpointLocation", customers_count_checkpoint_path)
.outputMode("complete")
.table("customer_count_by_state"))
```

- Identifying source locations
  - files = dbutils.fs.ls(f"{DA.paths.working_dir}/tracker")
    display(files)
- Use cases for using Auto Loader
  - Incremental data ingestion that requires integration of various data sources
    - Reporting
    - Data Analysis
    - Machine Learning

Multi-hop Architecture (bronze-silver-gold, streaming applications)

- Propagate new data through multiple tables in the data lakehouse
- Bronze
  - Bronze vs raw tables
    - Bronze tables contain raw data ingested from various sources (ex. JSON files, RDBMS data, IoT data, etc.)
  - Workloads using bronze tables as source
    - Enrichments and refinements to raw data
- Silver and Gold
  - Silver vs gold tables
    - Silver tables provide a more refined view of our data. We can join fields from various bronze tables to enrich streaming records, or update account statuses based on recent activity
    - Gold tables provide business level aggregates often used for reporting and dashboarding. This would include aggregations such as daily active website users, weekly sales per store, or gross revenue per quarter by department
  - Workloads using silver table as source
    - Providing business level aggregates often used for reporting and dashboarding
- Structured Streaming in Multi-hop
  - Converting data from bronze to silver levels with validation
    - Clean and enhance data from bronze table with temp streaming view
  - Converting data from silver to gold levels with aggregation
    - Create aggregations of data from silver table with temp streaming view

Delta Live Tables (benefits and features)

- Leverage Delta Live Tables to simplify productionalizing SQL data pipelines with Databricks

- General Concepts
    - Benefits of using Delta Live Tables for ETL
        - Declarative ETL pipelines
        - Data quality
        - Error handling and recovery
        - Easy pipeline development and maintenance
        - Continuous, always-on processing
        - Pipeline visibility
        - Simple deployments
        - Automatic testing
    - Scenarios that benefit from Delta Live Tables
        - Development in environments separate from production
        - Scenarios where new data is constantly being received
        - Scenarios with large volume of different files being processed
- UI
    - Deploying DLT pipelines from notebooks
        - A pipeline is a graph that links together the datasets that have been defined by the SQL/Python code
        - Navigate to Jobs and select 'Create Pipeline'
        - Create a live table in the notebook
            - CREATE OR REFRESH STREAMING LIVE TABLE table
    - Executing updates
        - Triggered
            - Run once and shut down until next manual or scheduled update
        - Continuous
            - Run continuously, ingesting new data as it arrives
    - Explore and evaluate results from DLT pipelines
        - Query gold table for insights
        - DAG (Directed Acyclic Graph) represents the entities involved in the pipeline and the relationships between them
        - Clicking on each shows a summary that includes:
            - Run status
            - Metadata summary
            - Schema
            - Data quality metrics
- SQL Syntax
    - Converting SQL definitions to Auto Loader syntax

  - References to Streaming Tables require addition of the STREAMING keyword in the declaration
  - cloud_files() method enables Auto Loader to be used natively with SQL
    - Parameters of cloud_files() method
      - Source location
      - Source data format
      - Arbitrarily sized array of optional reader options
- Common differences in DLT SQL syntax
  - References to DLT tables and views will always be preceded by the LIVE keyword
  - CONSTRAINT/ON VIOLATION for quality control
    - CONSTRAINT integrates with DLT, enabling it to collect metrics on constraint violations
    - ON VIOLATION modes supported by DLT
      - FAIL UPDATE
        - Pipeline failure when constraint is violated
      - DROP ROW
        - Discard records that violate constraints
      - Omitted
        - Records violating constraints are included (violations still reported in metrics)
  - References to other DLT tables and views include the live. prefix
  - References to streaming DLT tables use STREAM()
    - Table name as argument

Production Pipelines

Build production pipelines for data engineering applications and Databricks SQL queries and dashboards, including:

Jobs (scheduling, task orchestration, UI)

- Orchestrate tasks with Databricks jobs
    - 1. Create and configure a pipeline
    - 2. Schedule a Notebook Job
        - Chron Scheduling for manual/scheduled
        - Production jobs should always be scheduled against new job clusters, as they are billed at a much lower rate than all-purpose clusters
    - 3. Add a DLT pipeline as task that will execute after successful completion of previous task
- Automation
    - Setting up retry policies
        - Advanced options, Edit Retry Policy
    - Using cluster pools and why
        - Create a pool and configure job's cluster to use the pool
        - Decreases new job cluster start time
- Task Orchestration
    - Benefits of using multiple tasks in Job
        - Simplifies creation, management, and monitoring of data and machine learning workflows at no additional cost
        - Simple task orchestration
            - Eases the burden on data teams by enabling data scientists and analysts to build and monitor their own jobs, making key AI and ML initiatives more accessible
        - Fully integrated in Databricks and requires no additional infrastructure or DevOps resources
    - Configuring predecessor tasks
        - Workflows in sidebar, Name column, job name, Tasks tab
            - Order of execution of tasks in job can be defined using the Depends on drop-down
                - This creates a Directed Acyclic Graph (DAG) of task execution, a common way of showing execution order
- UI
    - Using notebook parameters in Jobs

        - Workflows in sidebar, Create Job, create task
            - Pass parameters for Notebook task
                - Add parameter and specify key and value of each parameter to pass to the task
                - Parameters set the value of the notebook widget specified by the key of the parameter
    - Locating job failures using Jobs UI
        - Jobs in sidebar, Name column
            - Shows active runs and completed runs, including unsuccessful runs

Dashboards (endpoints, scheduling, alerting, refreshing)

- Use Databricks SQL for on-demand queries

- Databricks SQL Endpoints
    - Creating SQL endpoints for different use cases
        - A SQL endpoint is a computation resource that lets you run SQL commands on data objects within Databricks SQL
            - Running queries on data lake, creating visualizations to explore query results, building and sharing dashboards, etc.
        - Navigate to Databricks SQL and go to SQL endpoints in the sidebar
        - Click Create SQL Endpoints
        - Many options that can be configured to meet specific needs
- Query Scheduling
    - Scheduling query based on scenario, query reruns based on interval time
        - Refresh Schedule at bottom right of SQL query editor box
            - Refresh time can be altered as necessary
            - End condition can be altered as necessary
- Alerting
    - Configure notifications for different conditions and/or alerts for failure
        - Navigate to Alerts in left side bar, Create Alert
            - Configure “Trigger when” field to desired condition within query
- Refreshing
    - Scheduling dashboard refreshes
        - Navigate to Dashboards on left sidebar
            - Click the Schedule button to review scheduling options
    - Query reruns impact on dashboard performance
        - Scheduling a dashboard to update will execute all queries associated with that dashboard

Data Governance

Understand and follow best security practices, including:

Unity Catalog (benefits and features)

- Lets organizations manage fine-grained data permissions using standard ANSI SQL or UI
- Works uniformly across clouds and data types
- Open, standard interface
- Central control to manage and audit shared data across organizations
- Secure access from any platform

Entity Permissions (team-based permissions, user-based permissions)

- Configuring access to production tables and database
    - Table ACLs - permissions can be configured for the following objects:
        - CATALOG - entire data catalog
        - DATABASE - specific database
        - TABLE - managed or external table
        - VIEW - SQL views
        - FUNCTION - named function
        - ANY FILE - underlying filesystem (bypasses other restrictions)
- Granting different levels of permissions for users and groups
    - Granting Privileges (who can grant permissions for what)
        - Databricks admin - all objects in catalog and underlying filesystem
        - Catalog owner - all objects in the catalog
        - Database owner - all objects in the database
        - Table owner - only the table (similar options for views and functions)
    - Privileges
        - ALL PRIVILEGES - gives all privileges (listed below)
        - SELECT - gives read access to an object
        - MODIFY - gives ability to add, delete, and modify data to/from an object
        - READ_METADATA - gives ability to view an object and its metadata
        - USAGE - additional req to perform any action on a database object
        - CREATE - gives ability to create an object (ex. A table in a database)