**Databricks Data Engineer Certification for Associate**

**Databricks Overview**

1. Unified Data Platform Solution
   * Combine **Data Engineer, Data Science, Data Analytics** into one platform
   * Data Lake
   * Data Warehouse
   * ETL
   * Orchestration
   * Governance
   * AL/ML
   * BI Dashboards
2. Utilize Open Source Solution for format compliance with vendors (AWS, Azure, Google)
   * Just act as an analytics platform based on Spark
   * Does not store the actual Data
3. Lakehouse Platform
   * Combine **Data Lake** and **Data Warehouse**
   * **Delta Lake**
     + Open Source storage layer (Data Lake)
     + Bring ACID transactions and data versioning (Data Warehouse)
     + Schema enforcement and evolution (Data Warehouse)
     + Data versioning and time travel
     + Unified Batch and Streaming processing
     + Scalable and cost effective
     + Data management features (Data Warehouse) – Data lineage, auditing

**Databricks Architecture**

1. Databricks Account
   * Contain Workspace(s)
   * Manage Metastore, Users, Group
2. Databricks High Level Architecture
   * Control Plane
     + Service that Databricks provides – Manage by Control Plane
     + Databricks Cloud Account
     + Web Application
     + Notebook Config
     + Cluster Config
     + Job Information - Log
   * Data Plane
     + Data is being processed
     + Customer Cloud Account
     + Actual Data – Version
     + Clusters created
3. Roles
   * Account Admin
     + Workspace
     + Metastore
     + User/Permission
   * Metastore Admin
     + Catalog
     + Data object
   * Workspace Admin
     + Admin of workspace
     + Users (Workspace level)
     + Workspace Assets
   * Owner
     + Create table, schema at workspace level

**Delta Lake, Delta Tables**

* Key features
  + ACID transactions
  + Scalable metadata
  + Unified Batch/Streaming
  + Time travel
  + Schema evolution/enforcement
  + Audit history
  + DML (Data manipulation language) operations (Merge, Update, Delete)
* Create Delta table
  + To create a delta table must use clause USING DELTA (SQL)
  + .write.format(“delta”) (Python) to create delta table
  + To be able to use the full features that it provides
* Time travel
  + @v0, @v1
* Schema evolution
  + Once new columns, the previous data will be null at the new column
* Can convert from parquet to delta
  + Read Delta Table using delta libraries
  + DeltaTable.convertToDelta(spark, “file\_format.`file\_path`”)
  + Can also convert metadata to delta
  + CONVERT TO DELTA metadata\_name
* Revert to a particular version
  + RESTORE TABLE table\_name TO VERSION AS OF version\_number
  + Can convert from Delta back to Parquet file
* Vacuum operations
  + dt.vacuum() – keeping the latest version
* **Partitioning in Spark for data scan**
  + Avoid partitioning on high cardinality or unique value column to avoid creating too many partitions (order ID)
  + Partition key selection is crucial
* Data Skipping and Z-Ordering in Delta Lake Tables
  + Data skipping
    - Technique for optimize query performance by avoid scanning unnecessary data
    - Metadata is collected, such as minimum and maximum
    - Query get executed, database engine use metadata to determine if the specific data block contains relevant records for the query
    - If data block does not match, then the engine skip reading that block
  + Z-Ordering
    - Data layout optimization technique based on columns specified
    - Preserve locality, meaning related records are stored closer to each other
    - Reduces data fragmentation
    - Optimized for multi-dimensional data, particularly useful when filtering on multiple columns

**Workspace and Notebook**

1. Notebook
   * Support multiple Languages
   * Need Compute Resources
   * By default, you will already have Spark Session created
   * Magic commands %sh (for Shell), %fs (for file system)
   * **Multiple people can use the same notebook – for collaboration**
   * Contain version history of notebook to restore (default display latest) and comments
   * Variable explorer store all declared variables

**How Databricks integrate with Cloud Services**

1. Storage
   * Create a managed resources group
     + Managed Identity
     + Storage Account
       - Store all managed data (on Cloud)
       - Cannot access these (managed by Databricks)
     + Access Connector
       - Use for connecting between Databricks and the storage
2. Compute Resources
   * Cluster
   * In the same managed resources group
     + Virtual machine created (total number of VMs = sum of driver and worker nodes) – will be installed with selected runtime on the VMs

**Data Governance and Unity Catalog**

1. Data Governance (means that data is secure, available, and accurate)

* Secure
* Available
* Accurate

1. Unity Catalog

* Open-source Unified governance solution for data assets
* Centralized
* Security
* Auditing
* Lineage
* Data discovery

1. Without Unity Catalog/With Unity Catalog

* Manage User management for each Databricks workspace – w/o
* Mange Metastore for each Databricks workspace – w/o
* Centralized management of user and metastore for all workspaces
* Just worry about compute resources

1. Hierarchical structure (3 levels)

* Metastore
  + Metadata (data about data – schemas, access control list, permissions) – control plane
  + Data – data plane – location of metastore
  + 1 region – 1 metastore
* Catalog
  + Data (Securable Objects)
  + Non-data securable objects – support data
  + Maintain data assets – (all data objects – structured, unstructured…)
* Schema
* Table, views (for structured data)
* Volume (file system for structured, unstructured, semi)
* 3 level namespaces (define catalog, schema, table) (catalog.schema.table)

**Legacy Hive Metastore Catalog – Managed and External tables**

* All structured data used to be stored under the Hive Metastore – including schema
* Need compute resource to run
* Managed Table
  + Both data and metadata will be managed by the metastore
  + When dropped both the data and metadata will be deleted
  + Delta table by default
  + No location specified
* External Table
  + Define a location where data would be stored
  + When dropped only the metadata will be deleted, not the data files
  + The data files can be reused

**Enable Unity Catalog and Set up Metastore**

* Metastore
  + Top level container in Unity Catalog
* Access connectors for connection with azure Databricks
* Assign to a workspace

**Catalog, External Location, and Storage Credentials in Unity Catalog**

* Managed table
  + Metastore, catalog, schema, then table
  + By default, the managed table will be stored at the location specifies by the metastore
  + But when you specify a location at the catalog level, that will be the new location to store managed tables
  + Next up, if you declare an external location at the schema level, that is where the managed tables will be store
  + If the location of the metastore is not declared then, you would have to declare the location for the managed table at the catalog level
* Location of data stored for managed tables will change depending on the external location specified by levels – metastore, catalog, schema, table
* Create catalog using SQL
  + CREATE CATALOG catalog\_name COMMENT “” LOCATION “catalog\_location”
* Create external location using SQL
  + CREATE EXTERNAL LOCATION “external\_location” URL “external\_location\_url” WITH (STORAGE CREDENTIAL “storage\_credential\_name”)
* Should create storage credential

**Schemas with External Location in Unity Catalog**

* Create schema without external location in dev catalog
  + CREATE SCHEMA dev.bronze – dev is the catalog, bronze is the schema
  + Managed tables will be created under the metastore in a managed location by Databricks
* Create schema without external location in dev\_ext catalog – catalog where we declared external location on
  + CREATE SCHEMA dev\_ext.bronze – dev\_ext is the catalog without external location
  + Managed tables will be created under the catalog level in managed location
* Create external location for schema
  + CREATE SCHEMA dev\_ext.bronze\_ext MANAGED LOCATION ‘external\_location/bronze\_ext’
  + Managed tables are stored at the schema level
* Validate location for managed tables

**Managed and External Tables in Unity Catalog vs Legacy Hive Metastore (UNDROP Table)**

* How managed tables and external tables are differed in unity catalog
* Still requires external locations
* Managed Tables
  + Under Unity Catalog, managed table is dropped but the data is still there from 7 to 30 days
  + The data is removed automatically but not immediately
  + Allow for restoration of the managed table if needed for up to 7 days after table is dropped
* External Tables
  + Requires specification of an external location
  + The data will remain there forever
  + The table can be restored if the data is still there
* To check for dropped tables and restore
  + USE CATALOG catalog\_name
  + SHOW TABLES DROPPED IN schema\_name
  + UNDROP TABLE catalog.schema.table\_name
  + UNDROP TABLE WITH ID ‘table\_id’

**Delta Tables Deep & Shallow Clones | Temporary & Permanent Views | List Catalog, Schemas & Tables**

* Show catalog, schema, table
  + SHOW CATALOGS
  + SHOW SCHEMAS
  + SHOW TABLES
* Create table as select (creating a copy of a table)
* Deep clone
  + Copy both metadata and data
  + Exact replica in terms of both metadata and data
  + CREATE TABLE new\_table\_name DEEP CLONE table\_clone\_from
  + CTAS and Deep Clone does the same thing – but CTAS might lost during transition
* Shallow clone
  + Copy only metadata
  + The data there would still be pointing to the original data
  + CREATE TABLE new\_table\_name SHALLOW CLONE table\_clone\_from
  + If you do not want the data to be copied again
  + The shallow clone will point to a particular version of the original table and will still using that version even when the original table is updated
  + Only impacted when the VACUUM command is run on the original table
  + When inserted new records, it will start to store data

**Merge and Upserts | SCD 1 in Delta | Soft Delete with Incremental data using Merge**

* Upsert
  + Insert new records
  + Update matching existing records
  + SCD 1
* Commands
  + MERGE INTO target\_table USING source\_table ON matching\_col
  + WHEN MATCHED THEN UPDATE SET old\_col = update\_col
  + WHEN NOT MATCHED THEN INSERT \*
  + WHEN NOT MATCHED BY SOURCE THEN DELETE (remove records that are not matched by source table)
* Soft delete at target table
  + Add new column is\_active for soft delete
  + When not matched by source then just update the is\_active column value to No
  + But this will make the schema mismatch between two tables therefore, cannot use insert \*

**Delta Tables Liquid Clustering and Deletion Vectors | Optimize Delta Tables | Delta Clustering**

* Liquid Clustering
  + Improve performance on delta tables with high cardinality
  + Normally, the data must be rewritten when trying to optimize the table
  + Improves existing partitioning and Z-order techniques by simplifying data layout decision to optimize query performance
  + Flexibility to redefine clustering columns without rewriting existing data
  + Use case
    - Tables partitioned by high cardinality columns
    - Tables with significant skew in data distribution
    - Tables grow quickly and require maintenance and tuning effort
    - Tables with access pattern that change over time
    - Tables where a typical partition column could result the table in too many or too few partitions
* Deletion Vectors
  + Normally, when you try to delete a record from a delta table, the whole parquet file is rewritten, this can be costly
  + Add deletion vectors
  + Deletion vectors mark existing rows as removed without rewriting the Parquet file
  + When read, only the rows that are not marked will be read
  + When run OPTIMIZE command, the parquet file will be rewritten and the rows marked will be permanently delete

**Volumes – Managed & External in Databricks | Volumes in Databricks Unity Catalog | Files in Volume**

* Volume is at the same level as table, view, and function and below schema
* Representing a logical volume of storage in a cloud object storage location
* Store structured and unstructured files
* Managed volume
  + CREATE VOLUME catalog.schema.volume\_name
* External volume
  + Need external location
  + CREATE EXTERNAL VOLUME catalog.schema.volume\_name LOCATION location\_store
* Dbutils commands to make new directories and copy files
* Same rules apply when dropped as with the managed and external tables

**DBUTILS Command | Databricks Utilities | Create Widgets in Databricks Notebook | DBUTILS FS Usage**

* DBUTILS commands only available in Python, R, and Scala notebook
* Works with
  + Files and Object storage efficiently
  + Secrets
* Commands
  + Dbutils.fs – Works with file system
    - Dbutils.fs.help() – list all the commands
    - Dbutils.fs.ls(path) – list all files, directories
    - Dbutils.fs.head(file\_path) – show the head content of a file
    - Dfutils.fs.mkdirs(path) – make directories
    - Dbutils.fs.cp(from\_path, to\_path) – copy a file
  + Widgets
    - Useful for parameterize queries and code
    - Dbutils.widgets.help() – list all commands for widgets help
    - Dbutils.widgets.get(name\_of\_widget) – to get input value of the widget
  + Secrets
    - Dbutils.secrets.help() – list all commands for secrets help
  + Notebook utilities
    - Dbutils.notebook.help() – list all commands for notebook utilities

**Orchestrating Notebook Jobs, Schedules using Parameters | Run Notebook from another Notebook**

* Create widgets and get widgets values in child notebook
* Create a run command in the parent notebook
  + Have the name of the notebook
  + Timeout seconds
  + Values for parameters in the form of dictionary
* Schedule for the notebook to run

**Databricks Computes – All Purpose & Job | Access Mode | Cluster Policies | Cluster Permissions**

* All Purpose compute
  + Provisioned compute – analyze data in the notebooks
  + Can create, terminate, and restart using UI, CLI, REST API
  + Multi node – multiple nodes
  + Single node – one node only
  + Access mode
    - Single user – All languages
    - Shared – Scala, Python, SQL
    - No isolation shared – All languages
      * Not supported by Unity Catalog
  + Photon Acceleration – boost the speed of the processing – increase cost
  + Worker type – number of workers
    - General purpose
    - Memory optimized
    - Compute optimized
    - Storage optimized
  + Driver type
    - Same as worker
    - Different from worker
  + Log location
  + Libraries
    - Install libraries with the cluster
    - .zar file, python – path of files
  + Metrics
    - CPU usage
    - Memory usage
    - Storage usage
    - Spark
    - GPU
  + Cluster permissions
    - Provide users and groups access permission
      * Can manage
      * Can restart
      * Can attach
  + Cluster policies
    - Unrestricted
    - Personal Compute
    - Power User Compute
    - Shared Compute
    - Legacy Shared Compute
* Job compute
  + Provisioned compute used to run automated jobs
  + Automatically started to run job and auto shutdown when the job is completed
* Serverless compute
  + On-demand, scalable compute
  + Used to execute SQL and Python code in notebook
  + Run Databricks jobs without configuring and deploying infrastructure

**Custom Cluster Policy in Databricks | Create Instance Pools | Warm Instance Pool**

* Define a policy using JSON
* Edit existing policies
* Customize new policy based on existing ones – clone new policy
* Adjust existing policies to enforce on the existing compute
* Databricks Instance Pools
  + Predefined group of nodes that are readily available to reduce compute start up time
  + Min Idle – minimum number of readily available nodes
  + Max capacity – maximum number of nodes in the pool
  + Warm instance pool – there is always at least one available node in the pool

**Workflows, Jobs & Tasks | Pass values within Tasks | If Else condition | For Each Loop & Re-Run Jobs**

* Workflows
  + To schedule or orchestrate or even trigger data processing jobs
  + Jobs
    - Pipeline of tasks
    - Create a new job
    - Rename
    - Define the tasks – type of the tasks
    - Specify the source and path of the source
    - Choose the cluster type – Job cluster will automatically terminate after the job is done
    - Widgets to pass values from task to task – using dbutils commands
    - Parameters
      * Job parameters – the value will be given to all tasks inside the job
      * Task parameters – the value will only be given to specific task
    - Queue
      * On – If the resource is not available for the job to run, job will be waiting for resource for the next 48 hours
* If, Else
  + Select as the task type of the new task in the workflow
  + Specify the condition
  + Choose the depends on and run if dependencies
* Repair failed tasks
  + No need to run the entire workflow
  + Just need to repair the failed tasks – click on repair run only
* Manually trigger run with override parameters
* For each loop
  + Create a loop over for a specific task
  + Specify all the parameters in the array of inputs
  + Select the parameters to be the array of inputs provided above

**COPY INTO Command | COPY INTO Metadata | Idempotent Pipeline | Exactly Once processing**

* Use to load multiple different file formats into data lake
* It is retriable and idempotent
  + If use it on the same file again and again, the number of records will not change
  + Like exactly once, data files will only be processed exactly once
* Support
  + Schema inference
  + Mapping
  + Merging
  + Schema evolution
* Specify the COPY\_OPTIONS() and FORMAT\_OPTIONS()
* Metadata
  + Stored in containers

**Auto Loader in Databricks | Auto Loader Schema Evolution Modes | File Detection Mode in Auto Loader**

* A utility for incrementally process new files arriving in your Cloud Storage account
* Co-related to Structured Streaming ingestion of Cloud Storage data files with source format defined as “cloudFiles”
* Provide a Structured Streaming source called cloudFiles
  + Behave just like ingesting data using structured streaming
  + Add .trigger() for either batch mode or streaming mode
* Exactly once file processing
  + File detection modes
    - Directory Listing
      * Scan-Based detection
      * .option(“cloudFiles.useNotifications”, “true”)
      * Mechanism
        + Use API calls to detect new files
        + Periodically lists all files in the directory and compare them to a checkpoint to identify new files
      * Simple setup with no external configuration
      * Less efficient for directories with very large number of files since time-consuming for listing all files
    - File Notification
      * Event-Driven detection
      * .option(“cloudFiles.useNotifications”, “false”)
      * Mechanism
        + Use Notifications and Queue services (requires elevated cloud permissions for set up)
        + File paths of new files are added to a queue managed by Auto Loader for processing
      * Efficient for large scale directories because avoid scanning entire directories with minimal latency
* Schema handle of Auto Loader
  + Schema location
    - .option(“cloudFiles.schemaLocation”, “file\_path”)
  + Schema hints – define schema for certain specific columns
    - .option(“cloudFiles.schemaHints”, “col\_name data\_type”)
  + Schema evolution
    - addNewColumns
      * Stream fails
      * New columns are added to the schema
      * Existing columns do not evolve data types
      * .option(“cloudFiles.schemaEvolutionMode”, “addNewColumns”)
    - rescue
      * Schema is never evolved and stream does not fail
      * All new columns are recorded in the rescued data column
      * .option(“cloudFiles.schemaEvolutionMode”, “rescue”)
    - none
      * Default mode if schema is provided
      * Does not evolve the schema, new columns are ignored
      * Stream does not fail
      * .option(“cloudFiles.schemaEvolutionMode”, “none”)
    - failOnNewColumns
      * Stream fails
      * Stream does not restart unless the provided schema is updated, or the offending data files are removed
      * .option(“cloudFiles.schemaEvolutionMode”, “failOnNewColumns”)
* RocksDB
  + Auto Loader identifies files and ensures that their metadata is saved in a scalable key-value store (RocksDB) in its pipeline’s checkpoint location
* Core features
  + Checkpoints to track processed files
    - Checkpoint to persist metadata about processed files
      * .option(“checkpointLocation”, “location\_path”)
    - Also stored on cloud storage
    - Include
      * File paths
      * Ingestion timestamps
      * File version identifiers (optional)
  + Schema evolution
    - Automatic schema detection and evolution
    - New columns or changes will be handled dynamically
  + Idempotent processing
    - Ensures that if files are detected multiple times
      * Retries
      * Storage anomalies
    - Still processed exactly once
  + Incremental load via Structured Streaming
    - Integrates with Spark Structured Streaming, enabling ingestion of new files as a continuous stream or in batch mode

**Medallion Architecture in Data Lakehouse | Bronze, Silver, Gold Layers**

* A data design pattern used to logically organize data in a lake
* Incrementally and progressively improving the structure and quality of data as it flows through each layer of the architecture
* Bronze layer
  + Raw data, store history
  + Land all data from external source systems
* Silver layer
  + Filtered, De-duplicated, and cleaned
  + Data from bronze layer is matched, merged, conformed, and cleansed
  + So that the silver layer can provide an “Enterprise view”
* Gold layer
  + Aggregated as per Business needs
  + Highly refined
  + For reporting and uses more de-normalized and read-optimized data models
  + Few joins
* All layers are backed by Data Lakehouse principles (Delta Lake), Data Quality checks and Data Governance

**DLT/ Delta Live Tables | DLT Part 1 | Streaming Tables and Materialized Views in DLT pipeline**

* Declarative framework for ETL processing – serves both streaming and batch ingestion
* Powered by Delta Lake – comes with all Delta Lake features
* DLT pipelines are made of 3 types of Datasets
  + Streaming Tables
    - Used to process incremental data
    - Allow for append data
  + Materialized Views
    - Used for aggregations, transformations, computations
  + Views
    - Used for intermediate transformations, not stored in target schema
* Requires jobs cluster – and available only in 2 languages (Python, SQL)
* Create streaming table in Python
  + import dlt
  + @dlt.table(table\_properties = {“quality”:”bronze”})
  + def bronze():
    - df = Spark.readStream.table(“raw\_table”)
    - return df
* Create materialized view
  + import dlt
  + @dlt.table(table\_properties = {“quality”:”bronze”})
  + def bronze():
    - df = Spark.read.table(“raw\_table”)
    - return df
* Create view
  + Views are only temprorary
  + @dlt.view(comment=”view”)
  + def view():
    - df\_1 = Spark.read.table(“LIVE.raw\_table”)
    - df\_2 = Spark.read.table(“LIVE.raw\_table”)
    - df\_join = df1.join(df\_2, how = “left\_outer”, on=df\_1.key==df\_2.key)
    - return df\_join
* LIVE keyword is used to reference DLT tables and views created in the same pipeline
* Can add new column using the withColumn(“col\_name”, col\_condition)
* Create DLT pipeline
  + Select Delta Live Tables
  + Specify the requirements
    - Select the source code to be the notebook you want
  + Development mode
    - Allows the DLT cluster to continue to run after the execution (fail/success)
    - For debugging purpose
  + Production mode
    - Immediately kills the cluster after execution

**DLT Internals and Incremental Load | DLT Part 2 | Add or Modify columns | Rename table | Data Lineage**

* All datasets in DLT are backed by Delta Tables and are managed by DLT pipelines
  + So, delete DLT pipeline
  + All datasets of that pipeline will be deleted
* Incremental Load
  + State management with checkpoints
    - Delta Lake’s transactional log and checkpoints to keep track
    - For streaming data
      * Offset or sequence number for each partition
      * Watermark define a range of lateness and ignores late-arriving data
  + Declarative Incremental pipelines
    - Incremental logic is applied by tracking unique identifiers (PK, timestamp, file\_names)
    - For streaming data
      * Defined the table using LIVE TABLE and use STREAM() for input, will automatically signals DLT to only processed new rows or changes from the source
  + Change Data Capture
    - Use the MERGE operations for upserts
    - Use APPLY CHANGES for CDC input handling
  + Materialized View management
    - DLT transform data into materialized views – stored as Delta Tables (updated incrementally)
      * Streaming live tables
        + Automatically receive new data or updates from upstream sources
        + Changes are incrementally applied to downstream tables in the pipeline
      * Incremental update logic
        + For streaming, only new or updated rows are processed
        + For batch, ensures idempotency by tracking changes – id, or updated\_at
  + Utilize structured streaming
* Streaming tables
  + Used to process data incrementally
  + Data is always appended
* Can easily add or modify columns
* Can easily rename tables
* Data Lineage are tracked by Unity Catalog
  + Full data lineage
  + Even columns information

**DLT Append Flow (Union) & Auto Loader | DLT Part 3 | Pass parameter in DLT pipeline | Generate tables dynamically**

* Auto Loader
  + Still spark.readStream
  + But with format(“cloudFiles”)
    - .option(“cloudFiles.schemaHints”, “schema”)
    - .option(“cloudFiles.schemaLocation”, “schema\_path”)
    - .option(“cloudFiles.format”, “file\_format”)
    - .option(“cloudFiles.schemaEvolutionMode”, “none”)
    - .trigger(processingTime=’time’)
    - Load(“files\_location”)
* Union between two streaming tables
  + Union by using Union will make the entire data being processed over and over
  + Union by using Append Flow will make sure only incremental data is processed
  + Steps
    - Create a union table – dlt.create\_streaming\_table(“table\_name”)
    - Append Flow – @dlt.append\_flow
      * target = “union\_table\_name”
    - Create function for union table – def union\_table\_append():
      * df = spark.readStream.table(“LIVE.delta\_table\_name”)
      * return df
* Pass parameters in DLT pipeline
  + Generate DLT tables dynamically
  + Settings of the Delta Live Tables
    - Advanced for passing parameters
    - Passing that into a variable in the notebook
    - Create a loop that would dynamically create DLT tables
    - Fill in all the placeholders using parameters values (variables declared)

**DLT SCD2 & SCD1 table | Apply changes | CDC | Back loading SCD2 table | Delete/Truncate SCD table**

* SCD1
  + Create a view using a function
  + Then create a streaming table
  + Use dlt.apply\_changes for apply SCD1 characteristics to the table
    - target
    - source
    - keys
    - stored\_as\_scd\_type = 1
    - apply\_as\_deletes = expr(“\_src\_action = ‘D’ ”)
    - apply\_as\_truncates = expr(“\_src\_action = ‘T’ ”)
    - sequence\_by = “\_src\_insert\_dt”
* SCD2
  + ALTER TABLE – ADD COLUMNS
  + UPDATE TABLE – SET DEFAULT VALUE
  + Use dlt.apply\_changes for apply SCD1 characteristics to the table
    - target
    - source
    - keys
    - stored\_as\_scd\_type = 2
    - except\_column\_list = [] – exclude unnecessary columns from the target table
    - sequence\_by = “\_src\_insert\_dt
  + To Back loading, filling data in SCD2
    - Select the table in the pipeline UI and select table for refresh, no need to recreate the entire table
* To delete data from table
  + Specify the \_src\_action=’D’ for SCD1 table
  + Only append new rows, does not delete for SCD2 table
* To truncate data
  + Specify the \_src\_action=’T’ for SCD1 table
  + Set the column in the \_src\_insert\_dt to null in SCD2 table

**DLT Data Quality & Expectations | Monitor DLT pipeline using SQL | Define DQ rule | Observability**

* Expectations
  + Optional clauses that can be used to manage the data quality checks
  + Apply data quality checks on each record passing through a query
  + Components
    - Description
    - Boolean statement based on stated conditions
    - Action to take when a record fails the expectation
  + Actions
    - Warn
      * Warning only but invalid records are still inserted
    - Drop
      * Drop invalid records and continue pipeline
    - Fail
      * Fail data pipeline if any record does not pass quality checks
  + Define rules for data quality
    - Using python dictionary
      * Rule for valid status
      * Rule for valid price
      * Etc.…
      * Separate tables should get separate rules
  + Specify using the rules for dlt
    - @dlt.expect\_all(rules\_dictionary) – if there are many rules – Warn
    - @dlt.expect(rule\_dictionary) – if there is only one rule
    - @dlt.expect\_all\_or\_fail(rules\_dictionary) – Fail
    - @dlt.expect\_all\_or\_drop(rules\_dictionary) – Drop
    - Can use both table and view to define data quality rules
  + Check the statistics of the data quality
    - Delta live tables tabs in Databricks UI
    - Click on specific tables
    - Choose the data quality tab to check the data quality
* Monitor DLT pipelines or Observability
  + SQL
    - SELECT \* FROM event\_log(“data\_pipeline\_id”)
    - Check documentation for a full view created for data quality monitoring

**DLT Truncate Load Source | Workflow File Arrival Triggers | Full Refresh | Schedule DLT Pipelines**

* Truncate load source
  + DLT pipeline will always fail because it needs an append only data source
  + Add .option(“skipChangeCommits”, “true”)
  + For table that are truncate and then load incrementally
  + Prevent DLT table from fully refreshed when the full refresh is triggered
    - table\_properties = {“pipelines.reset.allowed”: “false”}
* Set cluster to be in Production mode
  + When execution completed
  + Databricks will shut down cluster running for this job
* DLT can also be created using SQL

**Databricks Secret Management & Secret Scope | Save secrets in Databricks | Use of Azure Key Vault**

* Accessing data requires authentication to external data sources through JDBC
  + Instead of directly entering credentials into a notebook
  + Use Azure Databricks secrets to store credentials and reference them
  + Use Secret Scope
* Secret Scope
  + Azure Key Vault – backed
    - Reference secrets stored in Azure Key Vault using Azure Key Vault – backed secret scope, a read-only interface to the Key Vault, management of secrets in Azure
  + Databricks – backed
    - A Databricks-backed secret scope is stored in an encrypted database owned and managed by Azure Databricks
* Create Azure Key Vault backed Secret Scope
  + Go to the url: https://<databricks-workspace-url>#secrets/createScope
  + Provide scope name
  + Manage principal
  + Grab DNS name (Vault URL) from Azure Key Vault
  + Copy Resource ID from Azure Key Vault properties
* Access the secret inside Databricks notebook
  + Dbutils.secrets.help() – all commands
  + Dbutils.secrets.listScopes() – list all the scope
  + Dbutils.secrets.list(“scope\_name”) – list the secrets inside the scope
  + Dbutils.secrets.get(“scope\_name”, “secret\_name”) – retrieve the secret value inside the scope
* Create Databricks backed Secret Scope
  + Requires Databricks CLI
  + Manage the secrets and scope using Databricks CLI

**User Management in Databricks | How to add Users, Service Principal & Groups in Unity Catalog**

* Unity Catalog
  + Centralized system called that manage the users (User Management) and manage the metadata (Metastore)
* Adding users in Microsoft Entra ID
  + Add users with a principal name and domain name
  + [User@email.com](mailto:User@email.com) – User is principal name, email.com is domain name
* Adding users in Databricks Account Console
  + Go to the user management tab
    - Users – contain users that use the service
    - Service principals – used for automation, majority are robot accounts
  + Add users with the email is the same user principal as in the Microsoft Entra ID
  + SCIM – auto provision users created on Microsoft Entra ID to Databricks Account Console
    - Go to settings tab, and check the user provisioning tab
* Create and use Groups in Databricks
  + Create and manage users in Group in user management tab
* Assign users to Workspace
  + Go to Workspace tab
  + Select permissions and add permissions
    - Users (one by one)
    - Groups (everyone in the group)
    - Service principal
  + Specify the permission
* Workspace level access (Persona)
  + Click on user icon on the top right
  + Select settings, choose identity and access
  + Manage users, groups, service principals on the workspace level
    - Manage specific permissions at the entitlements tab
* Create service principal in Microsoft Azure
  + Can be either
    - Databricks managed
    - Microsoft Entra ID managed
      * Must create app registrations in Microsoft Entra ID
      * For Microsoft Entra application ID
  + Add permissions for the service principal, roles for the service principal so that the user can also manipulate it

**ELT with Spark SQL and Python**

1. Querying files

* Querying files directly
  + SELECT \* FROM file\_format.`/path/to/file`
    - Good for self-describing formats (JSON, parquet)
    - Not good for non-self-describing formats (CSV, TSV)
    - Query for single file or multiple file (file\_\*.json)
    - Query for entire directory (/path/dir/)
  + Raw data
    - Extract text files as raw strings – text-based files (JSON, CSV, TSV, TXT)
    - Extract files as raw bytes – images or unstructured data
  + CTAS: Registering tables from files
    - CREATE TABLE table\_name AS SELECT \* FROM file\_format.`/path/to/file`
    - Auto infer schema from query results – Do not support manual schema declaration
    - Do not support file options
  + CTU: Registering tables on external data sources
    - CREATE TABLE table\_name (col1, col2…) USING data\_sources OPTIONS (key1 = val1, key2 = val2…) LOCATION = path
    - Always external table – not moving data, but pointing to data files in external location
    - Not delta table – using external sources – cannot use features, performances of Delta Lake and Lakehouse
    - Solution: create temp view using CTU and then create table from view using CTAS – for files that requires options (csv, tsv – header=True, delimiters=”;”)

1. Writing to Tables

* Better to overwrite the tables instead of deleting and then creating new tables
  + Keep old versions for time travel if needed
  + Overwrite is faster than delete because no need for listing directory recursively or delete files
  + Atomic operations – concurrent queries (simultaneous queries) can still read the table
  + According to ACID – If fails the table will remain in the previous state
* Method to overwrite tables
  + CREATE OR REPLACE TABLE table\_name AS – schema on write
  + INSERT OVERWRITE – schema on read
    - Can only overwrite existing table if it already exists
    - Only overwrite new records that match the existing schema
    - Safer technique for overwriting an existing table without modifying the table schema
* Appending records to the tables
  + INSERT INTO
    - No guarantee of duplicated records
    - Solution: use MERGE INTO, WHEN MATCHED (UPDATE), WHEN NOT MATCHED (INSERT) – Instead of insert, it will be upsert
    - Merge operations – update, insert, delete are completed in a single atomic transaction

1. Advanced Transformations

* Spark SQL has built in functionality to interact with JSON data stored as strings
  + Use ‘:’ to traverse nested data structured (profile:first\_name, address:street)
  + Convert data to struct type then use (profile\_struct.\*) to flatten to a column
* Transformation functions
  + explode function - Put each element of an array into its own role
  + collect\_set function - Collect unique values for a field including fields within arrays
  + array\_distinct – Get unique elements of arrays
  + flatten – Flatten of nested arrays into just one array
* Joins
  + Standard joins
* Set Operations
  + Unions (all records)
  + Intersect (records found in both)
  + Minus (records from just from one minus other)
* Pivot Clause
  + Change data perspective
  + Aggregated values based on specific column values

1. Higher Order Functions and SQL UDFs

* Filter – filter an array based on a given lambda function
* Transform – apply a transformation to fields to create new column
* User-defined function – Custom combination of logic for function
* Describe function (extended)– for details about a user-defined function
* Drop user-defined function when no longer necessary

**Incremental Data Processing**

1. Structured Streaming

* Data Stream
  + Any data source that grows over time
  + New files landing in cloud storage
  + Updates to database captured in CDC feed
  + Events queued in pub/sub messaging feed like Kafka
* Process Data Stream
  + Reprocess the entire source dataset each time
  + Only process new data added since last update – use Spark Structured Streaming
* Spark Structured Streaming
  + Scalable streaming processing engine
  + Query infinite data source, auto detects new data and persists result incrementally to a data sink (files, tables)
  + Treating infinite data source as a table – Unbounded table
  + New data is just rows being appended to the table
  + Read Stream
    - streamDF = spark.ReadStream.table(“Input\_Table”)
  + Write Stream
    - Can configure the output
    - streamDF.writeStream.trigger(processingTime=”2 minutes”).outputMode(“append”).option(“checkpointLocation”, “/path”).table(“Output\_Table”)
    - Trigger interval
      * Default – 500ms
      * Fixed interval – trigger(processingTime=”5minutes”)
      * Triggered batch – trigger(once=True)
      * Triggered micro-batches – trigger(availableNow=True)
    - Output modes
      * Append (default) – outputMode(“append”) – append new rows
      * Complete – outputMode(“complete”) – target table is overwritten
    - Checkpointing
      * Store stream state
      * Track the progress of your stream processing
  + Guarantees
    - Fault-tolerance
      * In case of failure, can resume from where it left off
      * Checkpointing + Write ahead logs
    - Exactly-once guarantee
      * Each record only be written once
      * Idempotent sinks
  + Unsupported Operations
    - Sorting
    - Deduplication
    - Advanced methods to help
      * Windowing
      * Watermarking
  + If create a temp view from a streaming temp view -> temp view
    - spark.table(“example\_temp\_view”) to load data from a temp view into a data frame

1. Incremental Data Ingestion

* Loading new data files since last ingestion
* Reduces redundant processing
* 2 mechanisms
  + COPY INTO (SQL Command)
    - Load data from a file location into Delta tables
    - Idempotently and incrementally load new data files (only new files)
    - COPY INTO my\_table FROM ‘/path/to/files’ FILEFORMAT = <csv,parquet> FORMAT\_OPTIONS (<’header’=’true’> COPY\_OPTIONS (<’mergeSchema’=’true’>);
    - Thousands of files
    - Less efficient at scale
  + Auto Loader
    - Use Structured Streaming
    - Process billion of files
    - Support near real-time ingestion of millions of files per hour
    - Use checkpointing to track the ingestion process
      * Store metadata of discovered files
      * Exactly-once guarantees
      * Fault tolerance
      * readStream and writeStream method
      * Can auto infer schema of arriving files
      * Millions of files
      * Efficient at scale (split processing into multiple batches)