Data Engineering Cert Prep - Notes

1. Databricks basics:

1.1. Notebook basics - How to create a cluster, attach a notebook to a cluster, run different languages **(%sql, %python, %scala)** and use db utils **(dbutils.fs.ls)** for utility functions, magic commands(%) and markdown language **(%md)** to take notes within databricks notebooks

2. Delta Lake

2.1. Managing Delta Tables : **create table, insert into, select from, update and delete, drop** table queries can be used for delta lake operations. Default mode of operation in DBR 8+

2.2. Merge - Used for **Upserts (Updates + Inserts)**
Syntax - <<**Merge INTO** Destination Table using source table using conditions **When matched When not matched** >>
MERGE statements must have at least one field to match on, and each **WHEN MATCHED** or **WHEN NOT MATCHED** clause can have any number of additional conditional statements

2.3. **Advanced Options**

2.3.1. **OPTIMIZE** - Used to compact the files.`OPTIMIZE` will replace existing data files by combining records and rewriting the results.

2.3.2. **Z ORDER** - Used to index the file (partition and collocate the files). It speeds up data retrieval when filtering on provided fields by colocating data with similar values within data files.

2.3.3. **VACUUM** - Used to delete the stale data

- Defaults to 7 days
- Parameters for Vacuum retention duration and logging
- SET **spark.databricks.delta.retentionDurationCheck.enabled** = **false** to disable the check;
- SET **spark.databricks.delta.vacuum.logging.enabled = true** to enable the logging;
- Exception Note on Vacuum behavior - Because Delta Cache stores copies of files queried in the current session on storage volumes deployed to your currently active cluster, you may still be able to temporarily access previous table versions after executing vacuum (though systems should **not** be designed to expect this behavior).Restarting the cluster will ensure that these cached data files are permanently purged.

2.4. **Time travel - Key functions**

2.4.1. **VERSION -** HISTORY/TIME TRAVEL (Using Version and Time travel)

2.4.2. **DRY RUN -** Shows the fIles to be deleted before actually performing the operation

2.4.3. **RESTORE -** Used to rollback the previous version of a table

3. Relational Entities

3.1. Databases & Tables

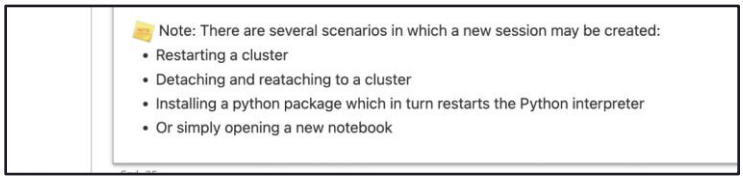
3.1.1. **Location -** Impacts the default storage location. Difference between external vs managed table

3.1.2. **Default -** By default, managed tables in a database without the location specified will be created in the hive directory i.e *`dbfs:/user/hive/warehouse/<database_name>.db/`*directory.

3.1.3. **DROP -** Deletes both metadata and data for MANAGED tables whereas Deletes **only Metadata** for UNMANAGED tables but the data still resides in the directory

3.2. Views, Temp Views & Global Temp Views

3.2.1. **Sessions:**



3.2.2. **Views** : Accessible within session

3.2.3. **Temp Views**: Accessible within session only for the specific notebooks. Cannot be accessed from another notebook

3.2.4. **Global Temp Views**: Accessible within session for the cluster under the global database. The global views are lost once the cluster is restarted.Global temp views behave much like other temporary views but differ in one important way. They are added to the **`global_temp`** database that exists on the **`cluster`**.As long as the cluster is running, this database persists and any notebooks attached to the cluster can access its global temporary views.

	database	tableName	isTemporary
1	dbacademy_jacob_parr_databricks_com_dewd_3_2	external_table	false
2	dbacademy_jacob_parr_databricks_com_dewd_3_2	view_delays_abq_lax	false
3		temp_view_delays_gt_120	true

	database	tableName	isTemporary
1	global_temp	global_temp_view_dist_gt_1000	true
2		temp_view_delays_gt_120	true

3.3. CTEs - Common Table Expressions are used in SQLs to catch temp results

3.3.1. Generally the scope is within Query. It is used for reusability of the complex queries and makes the code more readable.

3.3.2. Think of a CTE as being a View that only lasts for the duration of the query.
Note - This biggest difference is that a CTE can only be used in the current query scope whereas a temporary table or table variable can exist for the entire duration of the session allowing you to perform many different DML operations against them

4. ETL WITH SPARK SQL

4.1. Querying files directly

4.1.1. Files can be queried directly (CSV, JSON, PARQUET, TEXT, BINARY)

4.1.2. Syntax To Query single file - **SELECT * FROM file_format.`/path/to/file`**

4.1.3. Same format can be used to query the directory.

4.1.4. *Important call out - The assumption is the schema & format is same in case we are trying to read from the directory*

4.2. Providing Options for External Sources

4.2.1. While directly querying files works well for self-describing formats, many data sources require additional configurations or schema declaration to properly ingest records.

4.2.2. Registering tables on external locations with READ OPTIONS
CREATE TABLE table_identifier (col_name1 col_type1, ...)
USING data_source
OPTIONS (key1 = val1, key2 = val2, ...)
LOCATION = path
Note - We cannot expect the performance guarantees associated with Delta Lake & Lakehouse when querying external tables.Use ***Refresh table*** *to ensure the latest table is correctly cached for external location*

4.2.3. **Extracting Data from SQL Databases -** SQL databases are an extremely common data source, and Databricks has a standard JDBC driver for connecting with many flavors of SQL.The general syntax for creating these connections is:
CREATE TABLE USING JDBC
OPTIONS (
url

="jdbc:{databaseServerType}://{jdbcHostname}:{jdbcPort}", dbtable = "{jdbcDatabase}.table", user = "{jdbcUsername}", password = "{jdbcPassword}")

*Note - Some SQL systems such as data warehouses will have custom drivers. Spark will interact with various external databases differently, but the two basic approaches can be summarized as either:*Moving the entire source table(s) to Databricks and then executing logic on the currently active cluster Pushing down the query to the external SQL database and only transferring the results back to Databricks

In either case, working with very large datasets in external SQL databases can incur significant overhead because of either:Network transfer latency associated with moving all data over the public internet or the execution of query logic in source systems not optimized for big data queries

4.3. Creating Delta tables

4.3.1. **CTAS** - CTAS automatically infer schema information from query results and do **not** support manual schema declaration.This means that CTAS statements are useful for external data ingestion from sources with well-defined schema, such as Parquet files and tables.CTAS statements also do not support specifying additional file options.

4.3.2. **Declare Schema with Generated Columns -** Generated column are a special type of column whose values are automatically generated based on a user-specified function over other columns in the Delta table (introduced in DBR 8.3).

4.3.3. **Table constraints** are shown in the **`TBLPROPERTIES`** field.Databricks currently support two types of constraints:
NOT NULL constraints
CHECK constraints

4.3.4. **Enrich Tables with Additional Options and Metadata**

4.3.4.1. Our **SELECT** clause leverages two built-in Spark SQL commands useful for file ingestion:

- **current_timestamp()** records the timestamp when the logic is executed
- **input_file_name()** records the source data file for each record in the table

4.3.4.2. **CREATE TABLE** clause contains several options:

- A **COMMENT** is added to allow for easier discovery of table contents
- A **LOCATION** is specified, which will result in an external (rather than managed) table
- The table is **PARTITIONED BY** a date column; this means that the data from each data will exist within its own directory in the target storage location

4.3.5. Cloning Delta lake Table

4.3.5.1. **DEEP CLONE** fully copies data and metadata from a source table to a target. This copy occurs **incrementally,** so executing this command again can sync changes from the source to the target location.

4.3.5.2. **SHALLOW CLONE:**If you wish to create a copy of a table quickly to test out applying changes without the risk of modifying the current table, **`SHALLOW CLONE`** can be a good option. Shallow clones just copy the Delta transaction logs, meaning that the data doesn't move.

4.3.5.3. **CTAS VS SHALLOW CLONE**: Clone is simpler to specify since it makes a faithful copy of the original table at the specified version and you don't need to re-specify partitioning, constraints and other information as you have to do with CTAS. In addition, it is much **faster**, **robust**, and can work in an **incremental** manner against failures! With deep clones, we copy additional metadata, such as your streaming application transactions and COPY INTO transactions, so you can continue your ETL applications exactly where it left off on a deep clone!

4.4. Writing to Tables

4.4.1. Summary

Summary - Overwrite data tables using `INSERT OVERWRITE`,
Append to a table using **INSERT INTO ,**
Append, update, & delete from a table using **MERGE INTO**
Ingest data incrementally into tables using **COPY INTO**

4.4.2. Complete Overwrites-

We can use overwrites to atomically replace all of the data in a table. Spark SQL provides two easy methods to accomplish complete overwrites

4.4.2.1. **Why Overwrite?** - There are multiple benefits to overwriting tables instead of deleting and recreating tables:

- Overwriting a table is much faster because it doesn't need to list the directory recursively or delete any files.

- The old version of the table still exists; can easily retrieve the old data using Time Travel.
- It's an atomic operation. Concurrent queries can still read the table while you are deleting the table.
- Due to ACID transaction guarantees, if overwriting the table fails, the table will be in its previous state.

4.4.2.2. **CREATE OR REPLACE TABLE -**

- Replaces the complete content of the table

4.4.2.3. **INSERT OVERWRITE -**

- provides a nearly identical outcome as CREATE AND REPLACE TABLE data in the target table will be replaced by data from the query.
- Can only overwrite an existing table, not create a new one like our CRAS statement
- Can overwrite only with new records that match the current table schema -- and thus can be a "safer" technique for overwriting an existing table without disrupting downstream consumers
- Can overwrite individual partition

4.4.2.4. **CREATE OR REPLACE TABLE vs INSERT OVERWRITE -** A primary difference between CRAS and Insert Overwrite here has to do with how Delta Lake enforces schema on write.Whereas a CRAS statement will allow us to completely redefine the contents of our target table, INSERT OVERWRITE will fail if we try to change our schema (unless we provide optional settings)

4.4.3. Append Rows:

4.4.3.1. **INSERT INTO -** We can use **INSERT INTO** to atomically append new rows to an existing Delta table. This allows for incremental updates to existing tables, which is much more efficient than overwriting each time. Note - This doesn't avoid any duplication. One can insert the same records again with the above command.

4.4.3.2. **MERGE INTO -** is used for UPSERTS.Delta Lake supports inserts, updates and deletes in **MERGE**, and supports extended syntax beyond the SQL standards to facilitate advanced use cases.

The main benefits of **`MERGE`**: updates, inserts, and deletes are completed as a single transaction, multiple conditionals can be added in addition to matching fields and

it provides extensive options for implementing custom logic

Syntax **MERGE INTO** target a **USING** source b
ON {merge_condition} WHEN MATCHED THEN {matched_action}
WHEN NOT MATCHED THEN {not_matched_action}

Use Merge (Insert-Only) for Deduplication - Common ETL use case is to collect logs or other every-appending datasets into a Delta table through a series of append operations. Many source systems can generate duplicate records. With **merge,** you can avoid inserting the duplicate records by performing an insert-only merge.This optimized command uses the same **MERGE** syntax but only provides a **WHEN NOT MATCHED** clause.

*Note - **INSERT INTO VS MERGE INTO key difference is** that Insert Into doesn't avoid duplicates while MERGE INTO will help avoid duplicates*

4.4.4. Load Incrementally

- **COPY INTO** provides SQL engineers an **idempotent option (EXECUTE ONLY ONCE)** to incrementally ingest data from external systems.
  Note that this operation does have some expectations: -
  - Data schema should be consistent-
  - Duplicate records should try to be excluded or handled downstream
  - This operation is potentially much cheaper than full table scans for data that grows predictably.

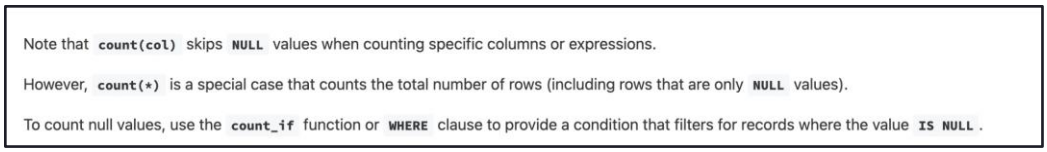
4.5. **Intentionally left blank**

4.6. Cleaning Data

4.6.1. **Count(col) vs Count (*)** - Note that **count(col) s**kips NULL`** values when counting specific columns or expressions.However, **count(*)** is a special case that counts the total number of rows (including rows that are only NULL values).

4.6.2. To count null values, use the **`count_if`** function or **`WHERE`** clause to provide a condition that filters for records where the value **`IS NULL`**.

4.6.3. *Note that when we called **DISTINCT(*)**, by default we ignored all rows containing **any** null values; as such, our result is the same as the count of user emails above*



4.6.4. Spark skips null values while counting values in a column or counting distinct values for a field, but does not omit rows with nulls from a **DISTINCT** query.

4.6.5. Other imp. functions: **WHERE, GROUP BY, ORDER BY**

4.7. Advance SQL Transformation - Using **.** and : syntax to query nested data, Working with JSON , Flattening and unpacking arrays and structs, Combining datasets using joins and set operators, Reshaping data using pivot tables, Using higher order functions for working with arrays

4.7.1. Interacting with JSON DATA

- **from_json function** is used to case the field to struct type
- **JSON.*** unpacks the json struct fields into a table columns i.e Once a JSON string is unpacked to a struct type, Spark supports * star) unpacking to flatten fields into columns.
- **Colon:** is used to extract data from JSON String
- **Dot .**is used for struct type
- **From JSON** helps read json in a table however it needs the JSON Schema as input
- Spark SQL also has a **schema_of_json** function to derive the JSON schema from an example

4.7.1.1. **Working with Arrays -** Spark SQL has a number of functions specifically to deal with arrays.

- **Explode ()** - function lets us put each element in an array on its own row.
- **Collect Arrays -**
  The **`collect_set`** function can collect unique values for a field, including fields within arrays.
  The **`flatten`** function allows multiple arrays to be combined into a single array.
  The **`array_distinct`** function removes duplicate elements from an array

4.7.2. Join Tables join operations (inner, outer, left, right, anti, cross, semi)

4.7.3. Set Operators

- ****`MINUS`**** returns all the rows found in one dataset but not the other; we'll skip executing this here as our previous query demonstrates we have no values in common.
- **UNION** returns the collection of two queries.
- **INTERSECT** returns all rows found in both relations.

4.7.4. Pivot Tables - The **PIVOT** clause is used for data perspective. We can get the aggregated values based on specific column values, which will be turned to multiple columns used in the **SELECT** clause. The **PIVOT** clause can be specified after the table name or subquery.

4.7.5. PIVOT: The first argument in the clause is an aggregate function and the column to be aggregated. Then, we specify the pivot column in the **FOR** subclause. The **IN** operator contains the pivot column values.

4.7.6. Higher Order Functions: Higher order functions in Spark SQL allow you to work directly with complex data types. When working with hierarchical data, records are frequently stored as array or map type objects. Higher-order functions allow you to transform data while preserving the original structure.

4.7.6.1. **FILTER** filters an array using the given lambda function.

4.7.6.2. **EXIST** tests whether a statement is true for one or more elements in an array.

4.7.6.3. **TRANSFORM** uses the given lambda function to transform all elements in an array.

4.7.6.4. **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.

4.8. SQL UDFs and Control Flow

4.8.1. SQL UDFs - At minimum, a SQL UDF requires a function name, optional parameters, the type to be returned, and some custom logic.

Syntax - <<*CREATE or REPLACE FUNCTION function_name(arg)*
RETURNS XYZ
RETURN actual function>>

4.8.2. Scoping and Permissions of SQL UDFs -

4.8.2.1. **Scope** - SQL UDFs will persist between execution environments (which can include notebooks, DBSQL queries, and jobs).

4.8.2.2. **Permissions** - SQL UDFs exist as objects in the metastore and are governed by the same Table ACLs as databases, tables, or views.In order to use a SQL UDF, a user must have **USAGE** and **SELECT** permissions on the function.

4.8.2.3. **`CASE` / `WHEN` -** The standard SQL syntactic construct **`CASE` / `WHEN`** allows the evaluation of multiple conditional statements with alternative outcomes based on table contents.

4.8.2.4. **Simple Control flow functions** - Combining SQL UDFs with control flow in the form of **`CASE` / `WHEN`** clauses provides optimized execution for control flows within SQL workloads

5. Python for Spark Sql (Optional)

5.1. Print and manipulate multi-line Python strings -

By wrapping a string in triple quotes (**"""**), it's possible to use multiple lines.

5.2. Define variables and functions

5.2.1. Variables - Python variables are assigned using the **=**.Python variable names need to start with a letter, and can only contain letters, numbers, and underscores. (Variable names starting with underscores are valid but typically reserved for special use cases.). Many Python programmers favor snake casing, which uses only lowercase letters and underscores for all variables.The cell below creates the variable **my_string**.

5.2.2. Functions - Functions allow you to specify local variables as arguments and then apply custom logic. We define a function using the keyword **def** followed by the function name and, enclosed in parentheses, any variable arguments we wish to pass into the function. Finally, the function header has a **:** at the end.

5.3. F-strings -

Use f-strings for variable substitution in the string. By adding the letter **f** before a Python string, you can inject variables or evaluated Python code by inserted them inside curly braces (**{}**)

5.4. Python Control Flow

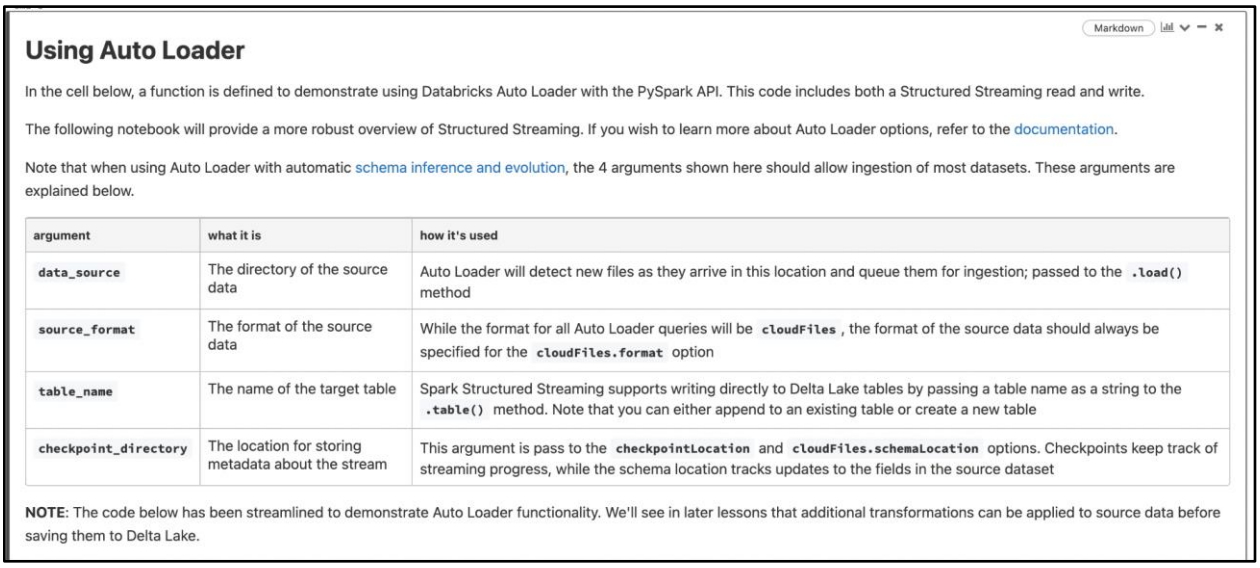
5.4.1. if/else - clauses are common in many programming languages. SQL has the **CASE WHEN ... ELSE** construct, which is similar.
Note - *If you're seeking to evaluate conditions within your tables or queries, use CASE WHEN.Python control flow should be reserved for evaluating conditions* ***outside of your query.***

5.4.2. **elif -** Python keyword **elif** (short for **else** + **if**) allows us to evaluate multiple conditions.Note that conditions are evaluated from top to bottom. Once a condition evaluates to true, no further conditions will be evaluated.

5.4.3. **if** / **else** control flow patterns:

- Must contain an **if** clause
- Can contain any number of **elif** clauses
- Can contain at most one **else** clause

6. Incremental Data Processing using Autoloader



Markdown

Using Auto Loader

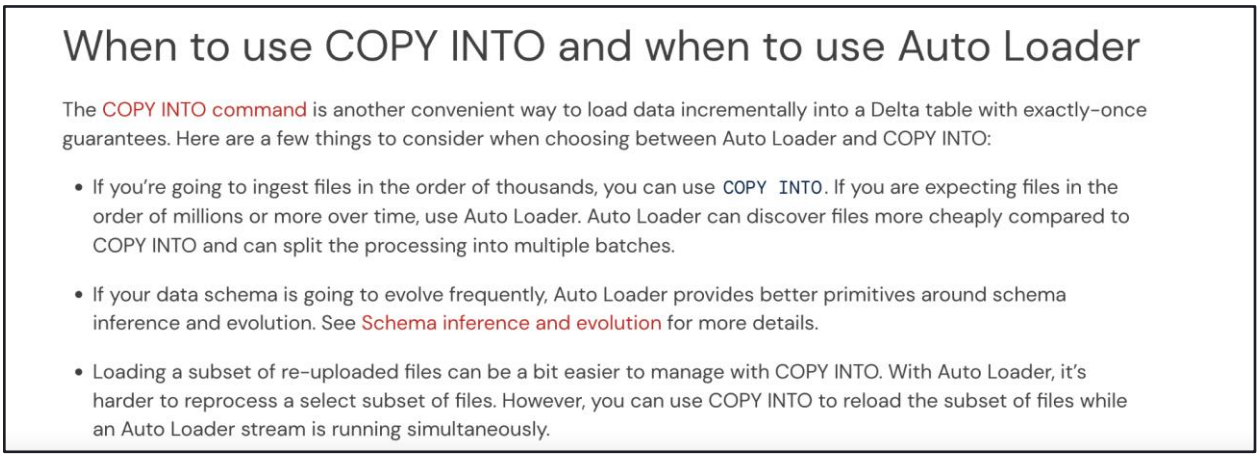
In the cell below, a function is defined to demonstrate using Databricks Auto Loader with the PySpark API. This code includes both a Structured Streaming read and write.

The following notebook will provide a more robust overview of Structured Streaming. If you wish to learn more about Auto Loader options, refer to the documentation.

Note that when using Auto Loader with automatic schema inference and evolution, the 4 arguments shown here should allow ingestion of most datasets. These arguments are explained below.

argument	what it is	how it's used
`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 source data	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

NOTE: The code below has been streamlined to demonstrate Auto Loader functionality. We'll see in later lessons that additional transformations can be applied to source data before saving them to Delta Lake.



When to use COPY INTO and when to use Auto Loader

The COPY INTO command is another convenient way to load data incrementally into a Delta table with exactly-once guarantees. Here are a few things to consider when choosing between Auto Loader and COPY INTO:

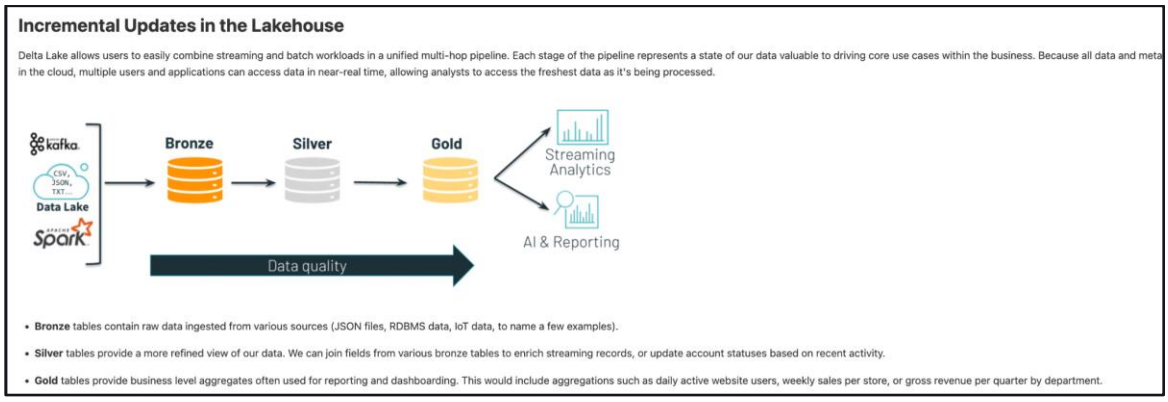
- If you're going to ingest files in the order of thousands, you can use `COPY INTO`. If you are expecting files in the order of millions or more over time, use Auto Loader. Auto Loader can discover files more cheaply compared to COPY INTO and can split the processing into multiple batches.
- If your data schema is going to evolve frequently, Auto Loader provides better primitives around schema inference and evolution. See Schema inference and evolution for more details.
- Loading a subset of re-uploaded files can be a bit easier to manage with COPY INTO. With Auto Loader, it's harder to reprocess a select subset of files. However, you can use COPY INTO to reload the subset of files while an Auto Loader stream is running simultaneously.

7. Medallion Architecture - Bronze, Silver & Gold Architecture

7.1. **Bronze -** Raw data read usually in append mode

7.2. **Silver -** Cleansed, filtered and augmented read from bronze tables usually in append mode. **provide a more refined view of our data**

7.3. **Gold -** Involves **business level aggregates** and metrics usually run in **Complete mode.** Mainly used for dashboarding and reporting



8. Delta Live Table

8.1. **Declarative language** to define the ETL.

8.2. LIVE - At its simplest, you can think of DLT SQL as a slight modification to traditional CTAS statements. DLT tables and views will always be preceded by the **LIVE** keyword

8.3. **Two modes-** Continuous vs Triggered Mode

8.4. **Quality Control -**

8.4.1. **The CONSTRAINT** keyword introduces quality control. Similar in function to a traditional **WHERE clause, CONSTRAINT** integrates with DLT, enabling it to collect metrics on constraint violations. Constraints provide an optional **ON VIOLATION** clause, specifying an action to take on records that violate the constraint. The three modes currently supported by DLT include:

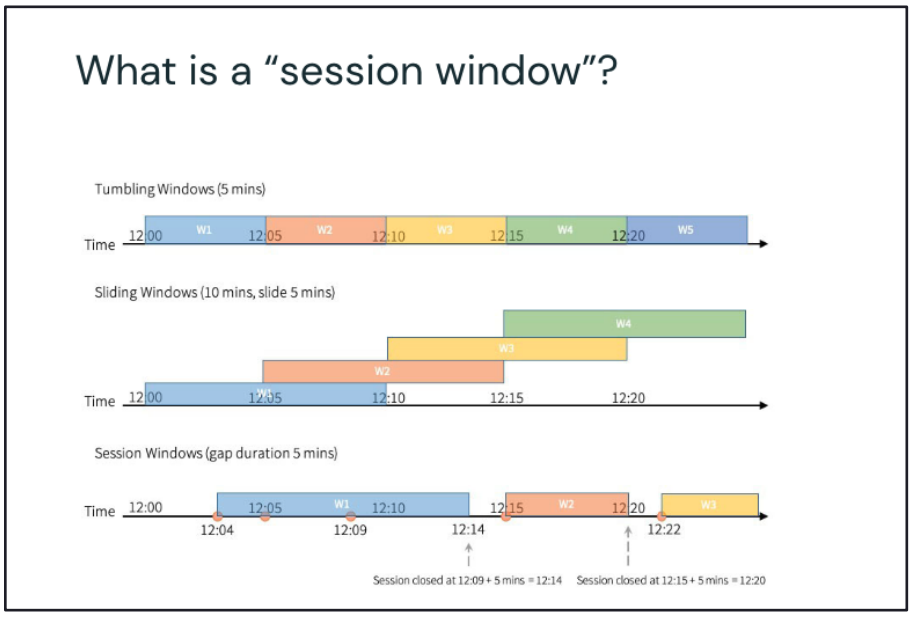
ON VIOLATION	Behavior
❖ FAIL UPDATE	❖ Pipeline failure when constraint is violated
❖ DROP ROW	❖ Discard records that violate constraints
❖ Omitted	❖ Records violating constraints will be included (but violations will be reported in metrics)

8.4.2. **References to DLT Tables and Views** - References to other DLT tables and views will always include the live. prefix. A target database name will automatically be substituted at runtime, allowing for easy migration of pipelines between DEV/QA/PROD environments.

8.4.3. **References to Streaming Tables - R**eferences to streaming DLT tables use the **STREAM(),** supplying the table name as an argument.

8.5. **Windows and Watermarking -**

8.5.1. Used for aggregate functions in streaming**.** Three types of windows - **Sliding, Tumbling, Session Windows**

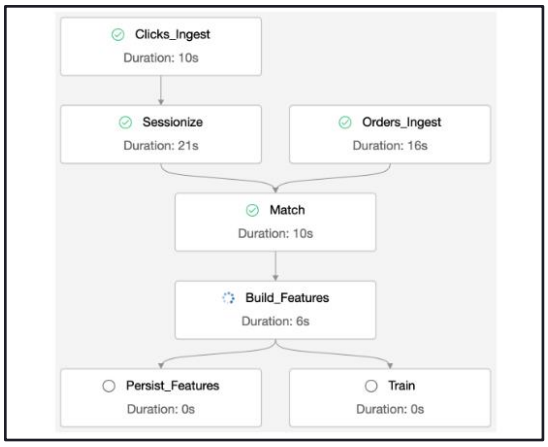


8.5.2. **Watermarking** - To bound the state size, we have to be able to drop old aggregates that are not going to be updated any more, for example seven day old averages. We achieve this using *watermarking*.Enables automatic dropping of old state data. It discards the data outside the watermark duration

8.5.3. Windows and watermarking is not available in detail in the notebooks. Watch Daniel's/Lorin's recording for the same. **(Starting ~ 33 mins at [this Link]**)

9. Task Orchestration with jobs -

9.1. Run a sample job in notebooks. (It's a DAG in which dependencies across tasks can be established. You can mix DLT & regular Notebooks)



10. DB SQL & Dashboard:

10.1.1. It's very intuitive and straightforward. Provides ability to execute SQL Queries, that can be scheduled and used to create the dashboards

10.1.2. Alerts can be setup to notify end user

10.1.3. Learn about Alerts, Serverless and try the labs on DB SQL

11. Managing Permission

11.1. **Data Explorer -** Use data explorer to set up the permissions.

11.1.1. **What is the Data Explorer? -** The data explorer allows users to:

- Set and modify permissions of relational entities
- Navigate databases, tables, and views
- Explore data schema, metadata, and history

11.1.2. **Syntax - GRANT/REVOKE <<PRIVILEGES>> ON <<OBJECT>> TO <<USERS/GROUP>>**

Table ACLs

Databricks allows you to configure permissions for the following objects:

Object	Scope
CATALOG	controls access to the entire data catalog.
DATABASE	controls access to a database.
TABLE	controls access to a managed or external table.
VIEW	controls access to SQL views.
FUNCTION	controls access to a named function.
ANY FILE	controls access to the underlying filesystem. Users granted access to ANY FILE can bypass the restrictions put on the catalog, databases, tables, and views by reading from the file system directly.

NOTE: At present, the `ANY FILE` object cannot be set from Data Explorer.

Granting Privileges

Databricks admins and object owners can grant privileges according to the following rules:

Role	Can grant access privileges for
Databricks administrator	All objects in the catalog and the 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).

NOTE: At present, Data Explorer can only be used to modify ownership of databases, tables, and views. Catalog permissions can be set interactively with the SQL Query Editor.

Privileges

The following privileges can be configured in Data Explorer:

Privilege	Ability
ALL PRIVILEGES	gives all privileges (is translated into all the below privileges).
SELECT	gives read access to an object.
MODIFY	gives ability to add, delete, and modify data to or from an object.
READ_METADATA	gives ability to view an object and its metadata.
USAGE	does not give any abilities, but is an additional requirement to perform any action on a database object.
CREATE	gives ability to create an object (for example, a table in a database).