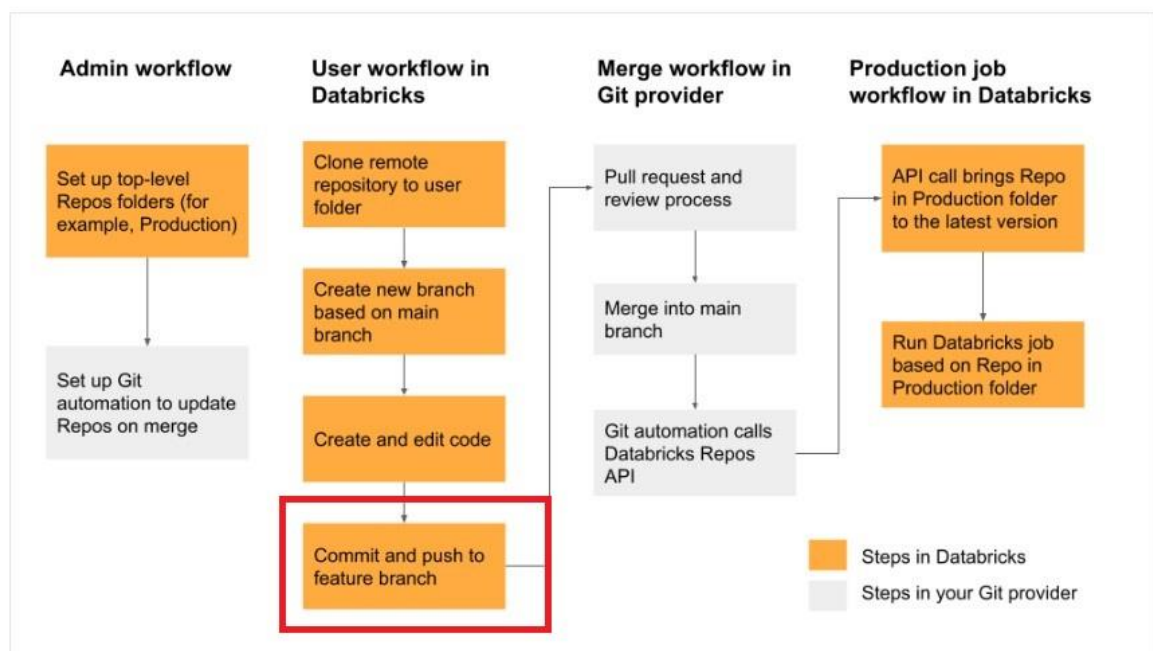


## Contents

<b>DE 1.1 - Create and Manage Interactive Clusters.....</b>	<b>4</b>
<b>DE 1.2 - Notebook Basics .....</b>	<b>4</b>
Magic Commands .....	5
%run.....	6
Display().....	6
Databricks Cloud.....	6
Databricks Utilities:.....	7
<b>DE 2.1 - Managing Delta Tables .....</b>	<b>9</b>
<b>DE 2.3 - Advanced Delta Lake Features .....</b>	<b>11</b>
DESCRIBE HISTORY.....	11
DESCRIBE DETAIL.....	11
DESCRIBE HISTORY.....	11
_delta_log Directory.....	11
Compacting Small Files and Indexing .....	11
OPTIMIZE & ZORDER BY.....	12
Time Travel.....	12
RESTORE.....	12
VACUUM operation:.....	12
<b>DE 3.1 - Databases and Tables on Databricks.....</b>	<b>14</b>
Databases.....	14
Tables .....	14
<b>DE 3.2A - Views and CTEs on Databricks .....</b>	<b>15</b>
Temp Views .....	15
Global Temp Views .....	15
Common Table Expressions (CTEs) .....	17
<b>04 - ETL with Spark SQL .....</b>	<b>18</b>
<b>DE 4.1 - Querying Files Directly.....</b>	<b>18</b>
Extract Text Files as Raw Strings.....	18
<b>DE 4.2 - Providing Options for External Sources .....</b>	<b>18</b>
Registering Tables on External Data with Read Options .....	18
REFRESH TABLE .....	19
Extracting Data from SQL Databases (SQLite).....	19
Managed Table.....	20
<b>DE 4.3 - Creating Delta Tables.....</b>	<b>21</b>

Create Table as Select (CTAS).....	21
Declare Schema with Generated Columns .....	22
Table Constraint.....	22
Enrich Tables with Additional Options and Metadata .....	23
Cloning Delta Lake Tables.....	24
<b>DE 4.4 - Writing to Tables.....</b>	<b>25</b>
CREATE OR REPLACE TABLE (CRAS).....	25
INSERT OVERWRITE.....	25
Append Rows .....	26
Merge Updates.....	26
Copy Into.....	27
<b>DE 4.6 - Cleaning Data .....</b>	<b>27</b>
Date Format and Regex .....	28
<b>DE 4.7 - Advanced SQL Transformations .....</b>	<b>28</b>
Interacting with JSON Data.....	28
Explode function.....	29
Collect_set function .....	29
Flatten function.....	30
Array_distinct function .....	30
Higher Order Functions.....	32
TRANSFORM.....	34
<b>DE 4.8 - SQL UDFs and Control Flow .....</b>	<b>35</b>
<b>DE 5.1 - Python Basics .....</b>	<b>36</b>
F-strings: .....	37
<b>DE 5.2 - Python Control Flow.....</b>	<b>39</b>
Assert .....	39
find() method .....	39
DE 6.1 - Incremental Data Ingestion with Auto Loader .....	41
<b>DE 6.2 - Reasoning about Incremental Data .....</b>	<b>41</b>
Unsupported Operations .....	42
Output Modes.....	43
Trigger Intervals .....	43
<b>DE 7.1 - Incremental Multi-Hop in the Lakehouse .....</b>	<b>44</b>
About Autoloader .....	47
Auto Loader vs COPY INTO?.....	48
Auto Loader Modes?.....	48

<b>Lakehouse Arch Advantages:</b> .....	50
<b>DE 8.1.1 - DLT UI Walkthrough</b> .....	52
Pipeline Modes.....	52
Enable the autoscaling.....	52
<b>DE 8.1.2 - SQL for Delta Live Tables</b> .....	53
Quality Control.....	53
<b>DE 9.1.1 - Task Orchestration with Databricks Jobs</b> .....	56
<b>DE 10.1 - Navigating Databricks SQL and Attaching to Endpoints</b> .....	61
Alerts.....	68
<b>DE 11.1 - Managing Permissions for Databases, Tables, and Views</b> .....	69
What is the Data Explorer?.....	69
Table ACLs.....	69
Dynamic View Functions .....	71
Databricks Repos .....	71

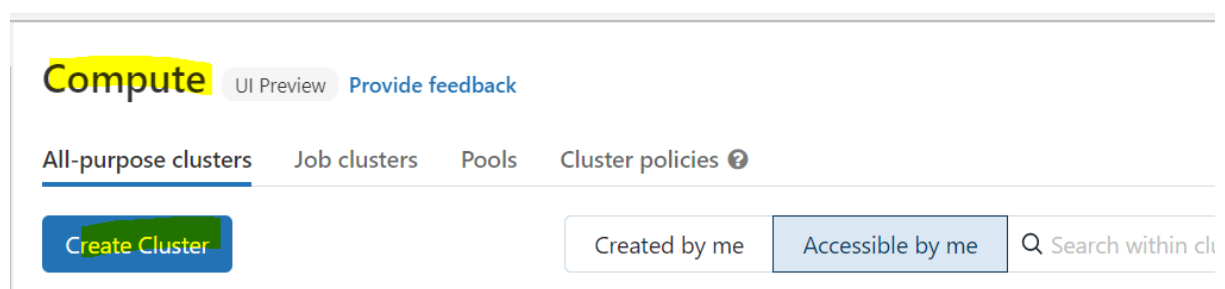
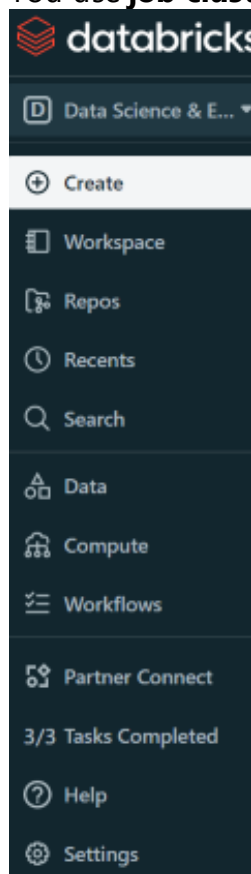


## DE 1.1 - Create and Manage Interactive Clusters

A Databricks cluster is a set of computation resources and configurations on which **you run data engineering, data science, and data analytics workloads**, such as production **ETL pipelines, streaming analytics, ad-hoc analytics, and machine learning**. You run these workloads as a **set of commands** in a notebook or as an automated job.

Databricks makes a distinction between all-purpose clusters and job clusters.

- You use **all-purpose clusters** to analyse data collaboratively using interactive notebooks.
- You use **job clusters** to run fast and robust automated jobs.



## DE 1.2 - Notebook Basics

- Notebooks provide cell-by-cell execution of code.
- Multiple languages can be mixed in a notebook. Users can add plots, images, and markdown text to enhance their code.
- Databricks notebooks support Python, SQL, Scala, and R.
- A language can be selected when a notebook is created, but this can be changed at any time.

### Magic Commands

- Magic commands are specific to the Databricks notebooks
- They are very similar to magic commands found in comparable notebook products
- These are built-in commands that provide the same outcome regardless of the notebook's language
- A single percent (%) symbol at the **start of a cell** identifies a magic command
  - You can only have one magic command per cell
  - A magic command must be the first thing in a cell
- %python
- %sql

In a notebook, all the cells contain code in Python language and you want to add another cell with a SQL statement in it. You changed the default language of the notebook to accomplish this task. What changes (if any) can be seen in the already existing Python cells?

In this case, the magic command `%python` will be added at the **beginning of all the cells** that contain Python code. If you try to change the default language for a notebook, all the cells containing code in previously selected default language will be updated and `%previous_default_language` (like `%sql`, `%scala`) will be added as the **first line** in the cells. This is done to assure that the **default language changes** will **not affect** the working of the notebook.

The following are the **language magic commands** supported in Databricks notebook:

1. `%scala`
2. `%r`
3. `%python`
4. `%sql`

More Info: [Effect of changing the default language of a Databricks notebook](#)

The magic command `%md` allows us to render **Markdown** in a cell:

## **%run**

- You can **run a notebook from another notebook** by using the magic command **%run**
- Notebooks to be run are specified with relative paths
- The referenced notebook executes as if it were part of the current notebook, so temporary views and other local declarations will be available from the calling notebook

```
%run ../Includes/Classroom-Setup-1.2
```

**Notebook\_A** has four commands:

1. `name = "John"`
2. `print(f"Hello {name}")`
3. `%run ./Notebook_B`
4. `print(f"Welcome back {full_name}")`

**uses Python to print some of those variables:**

```
print(f"DA: {DA}")
print(f"DA.username: {DA.username}")
print(f"DA.paths.working_dir: {DA.paths.working_dir}")
print(f"DA.db_name: {DA.db_name}")
```

From SQL:

```
%sql
SELECT '${da.username}' AS current_username,
       '${da.paths.working_dir}' AS working_directory,
       '${da.db_name}' as database_name
```

## **Display()**

Data will be displayed in a **rendered tabular format**.

The command has the following capabilities and limitations:

Preview of results limited to **1000 records**

- Provides a button to download results data as CSV
- Allows rendering plots
- 

```
files = dbutils.fs.ls("/databricks-datasets/nyctaxi-with-zipcodes/subsampled")
display(files)
```

## **Databricks Cloud**

The DBC (Databricks Cloud) file that is downloaded contains a zipped collection of the directories and notebooks

- When downloading a collection of DBCs, result previews and plots will also be exported.
- When downloading source notebooks, only code will be saved.

## Databricks Utilities:

- Databricks Utilities (dbutils) make it easy to perform powerful combinations of tasks.
- You can use the utilities to work with object storage efficiently, chain and parameterize notebooks, and work with secrets.
- dbutils are not supported outside of notebooks.
- dbutils utilities are available in Python, R, and Scala notebooks.

- dbutils.fs.ls("/databricks-datasets")
- dbutils.fs.help()
- dbutils.notebook.exit("Exiting from My Other Notebook")
- dbutils.notebook.run("My Other Notebook", 60)

- dbutils.widgets
  - dbutils.widgets.help()
  - dbutils.widgets.get("fruits\_combobox")
  - dbutils.widgets.combobox
  - dbutils.widgets.dropdown
  - dbutils.widgets.multiselect
  - dbutils.widgets.remove('fruits\_combobox')
  - dbutils.widgets.removeAll()
  - dbutils.widgets.text

- dbutils.fs.mount
- dbutils.credentials.assumeRole("arn:aws:iam::123456789012:roles/my-role")
- dbutils.credentials.showCurrentRole()
- dbutils.credentials.showRoles()
- dbutils.data.help() dbutils.data provides utilities for understanding and interpreting datasets
- dbutils.data.summarize: Calculates and displays summary statistics of an Apache Spark DataFrame or pandas DataFrame. This command is available for Python, Scala and R.

```
df = spark.read.format('csv').load(  
    '/databricks-datasets/Rdatasets/data-001/csv/ggplot2/diamonds.csv',  
    header=True,
```

```

inferSchema=True
)
dbutils.data.summarize(df)
• dbutils.fs.cp("/FileStore/old_file.txt", "/tmp/new/new_file.txt")
• dbutils.fs.head("/tmp/my_file.txt", 25)
• dbutils.fs.ls("/tmp")
• dbutils.fs.mkdirs("/tmp/parent/child/grandchild")
• dbutils.fs.mount
  aws_bucket_name = "my-bucket"
  mount_name = "s3-my-bucket"
  dbutils.fs.mount("s3a://%s" % aws_bucket_name, "/mnt/%s" % mount_name)
• dbutils.fs.mounts()
• dbutils.fs.mv("/FileStore/my_file.txt", "/tmp/parent/child/grandchild")
• dbutils.fs.put("/tmp/hello_db.txt", "Hello, Databricks!", True)
• dbutils.fs.refreshMounts()
• dbutils.fs.rm("/tmp/hello_db.txt")
• dbutils.fs.unmount("/mnt/<mount-name>")
•
• dbutils.jobs.taskValues.get(taskKey = "my-task", \
                              key      = "my-key", \
                              default  = 7, \
                              debugValue = 42)
• dbutils.jobs.taskValues.set(key = "my-key", \
                              value = 5)

```

## Cluster pools

- Cluster pools allow us to reserve VM's ahead of time, when a new job cluster is created VM are grabbed from the pool. Note: when the VM's are waiting to be used by the cluster only cost incurred is Azure. Databricks run time cost is only billed once VM is allocated to a cluster.
- Here is a demo of how to setup and follow some best practices,

[https://www.youtube.com/watch?v=FVtITxOabxg&ab\\_channel=DatabricksAcademy](https://www.youtube.com/watch?v=FVtITxOabxg&ab_channel=DatabricksAcademy)

- Cluster pools** help **speed up** the cluster and its ability to **auto scale**. A set of clusters are attached to the pool. Whenever a cluster is required, it is created from the pool. **Worker nodes and driver node** can share the same pool but **different pools** for worker nodes and driver node can also exist.
- So, if a cluster is taking a lot of time to start, one of the reasons could be that it is not attached to a cluster pool.***

More Info: [Cluster pools in Databricks](#)



## DE 2.1 - Managing Delta Tables

- In Databricks Runtime 8.0 and above, Delta Lake is the default format, and you don't need **USING DELTA**.
- **CREATE TABLE IF NOT EXISTS** students (id INT, name STRING, value DOUBLE)
- **INSERT INTO** students **VALUES**  
(4, "Ted", 4.7),  
(5, "Tiffany", 5.5),  
(6, "Vini", 6.3)
- Databricks doesn't have a **COMMIT** keyword; transactions run as soon as they're executed, and commit as they succeed.
- Delta Lake guarantees that any read against a table will **always** return the most recent version of the table, and that you'll never encounter a state of deadlock due to ongoing operations.
- Standard operations to create and manipulate Delta Lake tables, including:
  - **CREATE TABLE**
  - **INSERT INTO**

```
INSERT INTO beans VALUES  
("black", "black", 500, true),  
("lentils", "brown", 1000, true),  
("jelly", "rainbow", 42.5, false)
```

- **SELECT FROM**
- **UPDATE**
- **DELETE**
- **MERGE**
- **DROP TABLE**
- Updating records provides atomic guarantees as well
- Deletes are also atomic, so there's no risk of only partially succeeding when removing data from your data lakehouse.
- you can permanently delete data in the lakehouse using a **DROP TABLE** command.
- **Merge**:
  - upsert, which allows updates, inserts, and other data manipulations to be run as a single command.
  - Databricks uses the **MERGE** keyword to perform this operation.
  - Change Data Capture (CDC) feed

```
MERGE INTO students b  
USING updates u  
ON b.id=u.id  
WHEN MATCHED AND u.type = "update"  
  THEN UPDATE SET *  
WHEN MATCHED AND u.type = "delete"  
  THEN DELETE  
WHEN NOT MATCHED AND u.type = "insert"  
  THEN INSERT *
```

```

Assert
%python
assert spark.table("beans"), "Table named `beans` does not exist"
assert spark.table("beans").columns == ["name", "color", "grams", "delicious"],
    "Please name the columns in the order provided above"
assert spark.table("beans").dtypes == [("name", "string"), ("color", "string"),
    ("grams", "float"), ("delicious", "boolean")], "Please make sure the column types
    are identical to those provided above"

```

## %python

```

assert spark.table("beans").count() == 6, "The table should have 6 records"
assert spark.conf.get("spark.databricks.delta.lastCommitVersionInSession") == "2",
    "Only 3 commits should have been made to the table"
assert set(row["name"] for row in spark.table("beans").select("name").collect()) ==
    {'beanbag chair', 'black', 'green', 'jelly', 'lentils', 'pinto'}, "Make sure you have not
    modified the data provided"

```

```

%python
version = spark.sql("DESCRIBE HISTORY
beans").selectExpr("max(version)").first()[0]
last_tx = spark.sql("DESCRIBE HISTORY beans").filter(f"version={version}")
assert last_tx.select("operation").first()[0] == "MERGE", "Transaction should be
    completed as a merge"
metrics = last_tx.select("operationMetrics").first()[0]
assert metrics["numOutputRows"] == "3", "Make sure you only insert delicious
    beans"
assert metrics["numTargetRowsUpdated"] == "1", "Make sure you match on
    name and color"
assert metrics["numTargetRowsInserted"] == "2", "Make sure you insert newly
    collected beans"
assert metrics["numTargetRowsDeleted"] == "0", "No rows should be deleted by
    this operation"

```

```

%python
assert spark.sql(f"SHOW DATABASES").filter(f"databaseName ==
'{DA.db_name}').count() == 1, "Database not present"

```

## DE 2.3 - Advanced Delta Lake Features

- Databricks uses a Hive meta store by default to register databases, tables, and views.

### DESCRIBE HISTORY

- Using **DESCRIBE EXTENDED** allows us to see important metadata about our table.
  - DESCRIBE EXTENDED students

### DESCRIBE DETAIL

- Describe Detail is another command that allows us to explore table metadata.
  - DESCRIBE DETAIL students
  - DESCRIBE DETAIL allows us to see some other details about our Delta table, including the number of files.

### DESCRIBE HISTORY

All changes to the Delta Lake table are stored in the transaction log, we can easily review the table history.

### \_delta\_log Directory

- contains a number of Parquet data files
- Records in Delta Lake tables are stored as data in Parquet files.
- Transactions to Delta Lake tables are recorded
- Each transaction results in a new JSON file being written to the Delta Lake transaction log
- Rather than overwriting or immediately deleting files containing changed data, Delta Lake uses the transaction log to indicate whether or not files are valid in a current version of the table.

```
display(dbutils.fs.ls(f"{DA.paths.user_db}/students/_delta_log"))
```

### Compacting Small Files and Indexing

Small files can occur for a variety of reasons; in our case, we performed a number of operations where only one or several records were inserted

Files will be combined toward an optimal size (scaled based on the size of the table) by using the **OPTIMIZE** command.

## OPTIMIZE & ZORDER BY

Optimize will replace existing data files by combining records and rewriting the results.

When executing **OPTIMIZE**, users can optionally specify one or several fields for **ZORDER** indexing. While the specific math of Z-order is unimportant, it speeds up data retrieval when filtering on provided fields by colocating data with similar values within data files.

**OPTIMIZE students ZORDER BY id**

## Time Travel

**SELECT \* FROM students VERSION AS OF 3**

**DELETE FROM students**

## RESTORE

Restore command is recorded as a transaction; you won't be able to completely hide the fact that you accidentally deleted all the records in the table, but you will be able to **undo the operation and bring your table back to the desired state.**

**RESTORE TABLE students TO VERSION AS OF 8**

## VACUUM operation:

- Databricks will automatically **clean up stale files in Delta Lake tables.**
- While Delta Lake versioning and time travel are great for querying recent versions and rolling back queries, keeping the data files for all versions of large production tables around indefinitely is very expensive (and can lead to compliance issues if PII is present).
- **If you wish to manually purge old data files, this can be performed with the **VACUUM** operation**
- **execute it with a retention of 0 HOURS to keep only the current version**
  - **VACUUM students RETAIN 0 HOURS**
- **By default, **VACUUM** will prevent you from deleting files less than 7 days old**
- **If you run **VACUUM** on a Delta table, you lose the ability to time travel back to a version older than the specified data retention period.**
- **DRY RUN** version of vacuum is to print out all records to be deleted and will not delete any files
-

- SET spark.databricks.delta.retentionDurationCheck.enabled = false;
  - SET spark.databricks.delta.vacuum.logging.enabled = true;
  - VACUUM students RETAIN 0 HOURS DRY RUN
- 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 (though systems should **not** be designed to expect this behavior). Restarting the cluster will ensure that these cached data files are permanently purged.

VACUUM employees RETAIN 48 HOURS

The **VACUUM** command accepts values in **hours** and **not days**.

## DE 3.1 - Databases and Tables on Databricks

### Databases

Creating databases in two ways:

- One with no LOCATION specified
- One with LOCATION specified

Default location of the database is under **dbfs:/user/hive/warehouse/** and the database directory is the name of the database with the **.db extension**

- **CREATE DATABASE IF NOT EXISTS** \${da.db\_name}\_default\_location;
- **CREATE DATABASE IF NOT EXISTS** \${da.db\_name}\_custom\_location **LOCATION** '\${da.paths.working\_dir}/\_custom\_location.db';
- **DESCRIBE DATABASE EXTENDED** \${da.db\_name}\_default\_location;
- **DESCRIBE DATABASE EXTENDED** \${da.db\_name}\_custom\_location;
- **USE** database\_name;
- **SHOW TABLES IN** database\_name;

### Tables

**USE** \${da.db\_name}\_default\_location;

```
CREATE OR REPLACE TEMPORARY VIEW temp_delays USING CSV OPTIONS (  
  path = '${da.paths.working_dir}/flights/departuredelays.csv',  
  header = "true",  
  mode = "FAILFAST" -- abort file parsing with a RuntimeException if any malformed lines are encountered  
);  
CREATE OR REPLACE TABLE external_table LOCATION  
'${da.paths.working_dir}/external_table'  
AS  
SELECT * FROM temp_delays;  
  
SELECT * FROM external_table;  
  
DESCRIBE TABLE EXTENDED external_table;
```

**DESCRIBE EXTENDED** on a table will show all of the metadata associated with the table definition.

**DROP TABLE external\_table;** The table definition no longer exists in the meta store, but the underlying data remain intact.

DROP DATABASE \${da.db\_name}\_default\_location **CASCADE**;

DROP DATABASE \${da.db\_name}\_custom\_location CASCADE;

## DE 3.2A - Views and CTEs on Databricks

-- mode "FAILFAST" will abort file parsing with a RuntimeException if any malformed lines are encountered

```
CREATE TABLE external_table
USING CSV OPTIONS (
  path = '${da.paths.working_dir}/flight_delays',
  header = "true",
  mode = "FAILFAST"
);
```

```
SELECT * FROM external_table;
```

```
SHOW TABLES;
```

### Temp Views

```
CREATE TEMPORARY VIEW temp_view_delays_gt_120
AS SELECT * FROM external_table WHERE delay > 120 ORDER BY delay ASC;
```

```
SELECT * FROM temp_view_delays_gt_120;
```

**isTemporary** column. It will tell you whether it is temporary or not

### Global Temp Views

```
CREATE GLOBAL TEMPORARY VIEW global_temp_view_dist_gt_1000
AS SELECT * FROM external_table WHERE distance > 1000;
SELECT * FROM global_temp.global_temp_view_dist_gt_1000;
```

In the context of Databricks Notebooks and Clusters

- A Temp View is available across the **context of a Notebook** and is a common way of sharing data across various language REPL - Ex:- Python to Scala.
- A Global Temp View is available to **all Notebooks running on that Databricks Cluster**
- Global temp views are "lost" when the cluster is restarted.

Note the following:

- The view is associated with the current database. This view will be available to any user that can access this database and will persist between sessions.
- The temp view is not associated with any database. The temp view is ephemeral and is only accessible in the current SparkSession.
- The global temp view does not appear in our catalog. Global temp views will always register to the `global_temp` database. The `global_temp` database is ephemeral but tied to the lifetime of the cluster; however, it is only accessible by notebooks attached to the same cluster on which it was created.

The global temporary views are registered in a temporary database named **global\_temp**. So, to access the view you need to add **global\_temp** before the name of the view i.e. **global\_temp.View\_name**

There are **3 types** of Views supported in Databricks:

1. **View**
2. **Temporary View**
3. **Global Temporary View**

Out of these three - **View and Global Temporary View are persisted through multiple sessions** while the **Temporary View is tied to a session** and is not available to other sessions. Also note that if a cluster is restarted, the **temp views and the global temp views** are both **GONE** and **cannot** be accessed.

**Remember that views do not have any physical existence. If you need to store a data object physically on the file system, you should consider creating a table instead.**

More Info: [Views in Databricks](#)



# Common Table Expressions (CTEs)

```
WITH common_table_expression [, ...]
```

```
common_table_expression
```

```
view_identifier [ ( column_identifier [, ...] ) ] [ AS ] ( query )
```

```
-- CTE with multiple column aliases
> WITH t(x, y) AS (SELECT 1, 2)
  SELECT * FROM t WHERE x = 1 AND y = 2;
   1    2

-- CTE in CTE definition
> WITH t AS (
  WITH t2 AS (SELECT 1)
  SELECT * FROM t2)
  SELECT * FROM t;
   1

-- CTE in subquery
> SELECT max(c) FROM (
  WITH t(c) AS (SELECT 1)
  SELECT * FROM t);
   1

-- CTE in subquery expression
> SELECT (WITH t AS (SELECT 1)
  SELECT * FROM t);
   1

-- CTE in CREATE VIEW statement
> CREATE VIEW v AS
  WITH t(a, b, c, d) AS (SELECT 1, 2, 3, 4)
  SELECT * FROM t;
> SELECT * FROM v;
   1    2    3    4

-- CTE names are scoped
> WITH t AS (SELECT 1),
  t2 AS (
  WITH t AS (SELECT 2)
  SELECT * FROM t)
  SELECT * FROM t2;
   2
```

## 04 - ETL with Spark SQL

### DE 4.1 - Querying Files Directly

- `SELECT * FROM file_format.`/path/to/file``
- `SELECT * FROM json.`${da.paths.datasets}/raw/events-kafka/001.json``
- Query a Directory of Files
- `SELECT * FROM json.`${da.paths.datasets}/raw/events-kafka``
- `CREATE OR REPLACE TEMP VIEW events_temp_view`
  - `AS SELECT * FROM json.`${da.paths.datasets}/raw/events-kafka`;`
  - `SELECT * FROM events_temp_view`

### Extract Text Files as Raw Strings

```
SELECT * FROM text.`${da.paths.datasets}/raw/events-kafka/`
```

When working with text-based files (which include JSON, CSV, TSV, and TXT formats), you can use the **text format to load each line of the file as a row with one string column named value**. This can be useful when data sources are prone to corruption and custom text parsing functions will be used to extract value from text fields.

```
SELECT * FROM binaryFile.`${da.paths.datasets}/raw/events-kafka/`
```

Using **binaryFile** to query a directory will provide file metadata alongside the binary representation of the file contents.

### DE 4.2 - Providing Options for External Sources

When Direct Queries Don't Work: CSV files are one of the most common file formats, but a direct query against these files rarely returns the desired results.

```
SELECT * FROM csv.`${da.paths.working_dir}/sales-csv`
```

### Registering Tables on External Data with Read Options

```
CREATE TABLE table_identifier (col_name1 col_type1, ...)  
USING data_source  
OPTIONS (key1 = val1, key2 = val2, ...)  
LOCATION = path
```

```
CREATE TABLE sales_csv
```

```
(order_id LONG, email STRING, transactions_timestamp LONG, total_item_quantity INTEGER,
purchase_revenue_in_usd DOUBLE, unique_items INTEGER, items STRING)
```

```
USING CSV
```

```
OPTIONS (
```

```
  header = "true",
```

```
  delimiter = "|"
```

```
)
```

```
LOCATION "${da.paths.working_dir}/sales-csv"
```

```
DESCRIBE EXTENDED sales_csv
```

```
%python
```

```
(spark.table("sales_csv")
```

```
  .write.mode("append")
```

```
  .format("csv")
```

```
  .save(f"{DA.paths.working_dir}/sales-csv"))
```

## REFRESH TABLE

Our external data source is not configured to tell Spark that it should refresh this data.

We **can** manually refresh the cache of our data by running the **REFRESH TABLE** command.

Refresh table <table\_name>;

## Extracting Data from SQL Databases (SQLite)

```
CREATE TABLE
```

```
USING JDBC
```

```
OPTIONS (
```

```
  url = "jdbc:{databaseServerType}://{jdbcHostname}:{jdbcPort}",
```

```
  dbtable = "{jdbcDatabase}.table",
```

```
  user = "{jdbcUsername}",
```

```
  password = "{jdbcPassword}"
```

```
)
```

**NOTE:** SQLite uses a local file to store a database, and doesn't require a port, username, or password.

```
DROP TABLE IF EXISTS users_jdbc;

CREATE TABLE users_jdbc
USING JDBC
OPTIONS (
  url = "jdbc:sqlite:${da.username}_ecommerce.db",
  dbtable = "users"
)
```

## Managed Table

While the table is listed as **MANAGED**, listing the contents of the specified location confirms that no data is being persisted locally.

Note that 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:

1. Moving the entire source table(s) to Databricks and then executing logic on the currently active cluster
2. Pushing down the query to the external SQL database and only transferring the results back to Databricks

**For external or unmanaged tables Databricks manages only the metadata associated with the table.**

More Info: [Managed table in Databricks](#)

## DE 4.3 - Creating Delta Tables

### Create Table as Select (CTAS)

CREATE TABLE AS SELECT statements create and populate Delta tables using data retrieved from an input query.

```
CREATE OR REPLACE TABLE sales AS  
SELECT * FROM parquet. `${da.paths.datasets}/raw/sales-historical/`;
```

```
CREATE OR REPLACE TABLE sales_unparsed AS  
SELECT * FROM csv. `${da.paths.datasets}/raw/sales-csv/`;
```

```
DESCRIBE EXTENDED sales;
```

- CTAS statements 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**.
- We can see how this would present significant limitations when trying to ingest data from CSV files.

```
CREATE OR REPLACE TEMP VIEW sales_tmp_vw  
(order_id LONG, email STRING, transactions_timestamp LONG, total_item_quantity INTEGER,  
purchase_revenue_in_usd DOUBLE, unique_items INTEGER, items STRING)  
USING CSV  
OPTIONS (  
  path = "${da.paths.datasets}/raw/sales-csv",  
  header = "true",  
  delimiter = "|" )  
);
```

```
CREATE TABLE sales_delta AS  
SELECT * FROM sales_tmp_vw;
```

```
SELECT * FROM sales_delta
```

## Declare Schema with Generated Columns

Generated columns are a special type of column whose values are automatically generated based on a user-specified function over other columns in the Delta table

```
CREATE OR REPLACE TABLE purchase_dates (  
  id STRING,  
  transaction_timestamp STRING,  
  price STRING,  
  date DATE GENERATED ALWAYS AS (  
    cast(cast(transaction_timestamp/1e6 AS TIMESTAMP) AS DATE))  
  COMMENT "generated based on `transactions_timestamp` column")
```

Because **date** is a generated column, if we write to **purchase\_dates** without providing values for the **date** column, Delta Lake automatically computes them.

**NOTE:** The cell below configures a setting to allow for generating columns when using a Delta Lake **MERGE** statement.

```
SET spark.databricks.delta.schema.autoMerge.enabled=true;
```

```
MERGE INTO purchase_dates a  
USING purchases b  
ON a.id = b.id  
WHEN NOT MATCHED THEN  
  INSERT *
```

## Table Constraint

Databricks currently supports two types of constraints:

- NOT NULL constraints
- CHECK constraints

In both cases, you must ensure that no data violating the constraint is already in the table prior to defining the constraint. Once a constraint has been added to a table, data violating the constraint will result in write failure.

```
ALTER TABLE purchase_dates ADD CONSTRAINT valid_date CHECK (date > '2020-01-01');
```

DESCRIBE EXTENDED purchase\_dates;

Table constraints are shown in the **TBLPROPERTIES** field.

## Enrich Tables with Additional Options and Metadata

We show evolving a CTAS statement to include a number of additional configurations and metadata.

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

We also include logic to create a new date column derived from timestamp data in the source.

The **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

**As a best practice, you should default to non-partitioned tables for most use cases when working with Delta Lake.**

**NOTE:** Most Delta Lake tables (especially small-to-medium-sized data) will not benefit from partitioning. Because partitioning physically separates data files, this approach can result in a **small file problem** and prevent file compaction and efficient data skipping. The benefits observed in Hive or HDFS do not translate to Delta Lake, and you should consult with an experienced Delta Lake architect before partitioning tables.

```

CREATE OR REPLACE TABLE users_pii
COMMENT "Contains PII"
LOCATION "${da.paths.working_dir}/tmp/users_pii"
PARTITIONED BY (first_touch_date)
AS
  SELECT *,
    cast(cast(user_first_touch_timestamp/1e6 AS TIMESTAMP) AS DATE)
first_touch_date,
    current_timestamp() updated,
    input_file_name() source_file
  FROM parquet.`${da.paths.datasets}/raw/users-historical/`;

SELECT * FROM users_pii;

```

All of the comments and properties for a given table can be reviewed using **DESCRIBE TABLE EXTENDED**.

**NOTE:** Delta Lake automatically adds several table properties on table creation.

## Cloning Delta Lake Tables

Delta Lake has two options for efficiently copying Delta Lake tables.

**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.

```
CREATE OR REPLACE TABLE purchases_clone DEEP CLONE purchases
```

**SHALLOW CLONE** can be a good option. Shallow clones just copy the Delta transaction logs, meaning that the data doesn't move.

```
CREATE OR REPLACE TABLE purchases_shallow_clone SHALLOW CLONE purchases
```



## DE 4.4- Writing to Tables

We can use **overwrites** to atomically replace all of the data in a table. 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.

Spark SQL provides two easy methods to accomplish complete overwrites.

### CREATE OR REPLACE TABLE (CRAS)

**CREATE OR REPLACE TABLE** (CRAS) statements fully replace the contents of a table each time they execute.

```
CREATE OR REPLACE TABLE events AS  
SELECT * FROM parquet.`${da.paths.datasets}/raw/events-historical`
```

### DESCRIBE HISTORY events

### INSERT OVERWRITE

**INSERT OVERWRITE** provides a nearly identical outcome as above: data in the target table will be replaced by data from the query.

**INSERT OVERWRITE:**

- 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 partitions

```
INSERT OVERWRITE sales  
SELECT * FROM parquet.`${da.paths.datasets}/raw/sales-historical`
```

**INSERT OVERWRITE** command helps to insert new data to a table while overwriting the old data but for the command to run smoothly, the schema of **source data** and **target table** should be **identical**.

If the schema of the source data and the target table is different which results in a **schema mismatch error**. Alternatively, to run the insert overwrite query without any errors or warnings `option("overwriteSchema", "true")` can be added.

More Info: [Using INSERT OVERWRITE to overwrite a table in Spark SQL](#)

DESCRIBE HISTORY sales

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).

## Append Rows

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.

INSERT INTO sales

SELECT \* FROM parquet.`\${da.paths.datasets}/raw/sales-30m`

## Merge Updates

You can **upsert data** from a source table, view, or DataFrame into a target Delta table using the **MERGE** SQL operation. Delta Lake supports inserts, updates and deletes in **MERGE**, and supports extended syntax beyond the SQL standards to facilitate advanced use cases.

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

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
- provides extensive options for implementing custom logic

MERGE INTO events a

```
USING events_update b
ON a.user_id = b.user_id AND a.event_timestamp = b.event_timestamp
WHEN NOT MATCHED AND b.traffic_source = 'email' THEN
  INSERT *
```

## Copy Into

**COPY INTO** provides SQL engineers an idempotent option 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.

```
COPY INTO sales
FROM "${da.paths.datasets}/raw/sales-30m"
FILEFORMAT = PARQUET
```

## DE 4.6 - Cleaning Data

Note that `count(col)` skips 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).

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**.

```
SELECT
  count_if(user_id IS NULL) AS missing_user_ids,
  count_if(user_first_touch_timestamp IS NULL) AS missing_timestamps,
```

```
count_if(email IS NULL) AS missing_emails,  
count_if(updated IS NULL) AS missing_updates  
FROM users_dirty
```

**COUNT(DISTINCT(\*))** ignored the nulls.

Select

```
count(DISTINCT(*)) AS unique_non_null_rows  
FROM users_dirty
```

## Date Format and Regex

- Uses **regexp\_extract** to extract the domains from the email column using regex

```
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_users  
)
```

## DE 4.7 - Advanced SQL Transformations

### Interacting with JSON Data

In most cases, Kafka data will be binary-encoded JSON values.

The **e-commerce** field is a struct that contains double and 2 longs.

We can interact with the subfields in this field using standard `.` syntax like how we might traverse nested data in JSON.

- [Unpacking JSON column in SQL](#)
- Silver to Gold transition always (at least most of the times) involves aggregation.
- More Info: [Aggregating data in PySpark](#)

Spark SQL has a number of functions specific to deal with arrays.

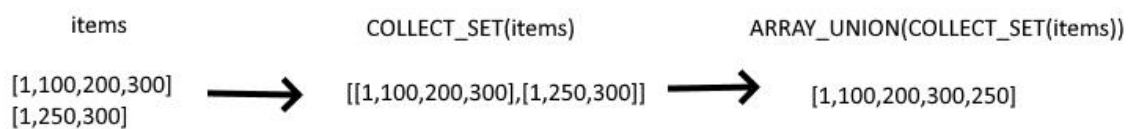
## Explode function

- The **explode** function lets us put each element in an array on its own row.

```
SELECT user_id, event_timestamp, event_name, explode(items) AS item
FROM events
```

## Collect\_set function

- The **collect\_set** function can collect unique values for a field, including fields within arrays.
- **COLLECT SET** is a kind of aggregate function that combines a column value from all rows into a unique list
- **ARRAY\_UNION** combines and removes any duplicates,



COLLECT\_SET: Collects unique values, including arrays

Input:

Id	value
1	['A', 'B']
1	['A', 'B']
1	['B', 'C']
1	['B', 'C']

SELECT id, COLLECT\_SET (value) FROM TABLE GROUP BY id

Id	value
1	[['A','B'], ['B','C']]

SELECT id, COLLECT\_LIST (value) FROM TABLE GROUP BY id

Id	value
1	[['A','B'], ['A','B'], ['B','C'], ['B','C']]

## Flatten function

- The **flatten** function allows multiple arrays to be combined into a single array.

## Array\_distinct function

- The **array\_distinct** function removes duplicate elements from an array.

```
SELECT user_id,  
       collect_set(event_name) AS event_history,  
       array_distinct(flatten(collect_set(items.item_id))) AS cart_history  
FROM events  
GROUP BY user_id
```

```
CREATE OR REPLACE VIEW sales_enriched AS  
SELECT *
```

```
FROM (
  SELECT *, explode(items) AS item
  FROM sales) a
INNER JOIN item_lookup b
ON a.item.item_id = b.item_id;

SELECT * FROM sales_enriched
```

- **UNION** returns the collection of two queries.
- **INTERSECT** returns all rows found in both relations.
- **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.
- 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 **SELECT** clause. The **PIVOT** clause can be specified after the table name or subquery.

**SELECT \* FROM ()**: The **SELECT** statement inside the parentheses is the input for this table.

**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.

Here we use **PIVOT** to create a new **transactions** table that flattens out the information contained in the **sales** table.

This flattened data format can be useful for dashboarding, but also useful for applying machine learning algorithms for inference or prediction.

```
CREATE OR REPLACE TABLE transactions AS
```

```
SELECT * FROM (
  SELECT
    email,
    order_id,
    transaction_timestamp,
    total_item_quantity,
    purchase_revenue_in_usd,
```

```

        unique_items,
        item.item_id AS item_id,
        item.quantity AS quantity
    FROM sales_enriched
) PIVOT (
    sum(quantity) FOR item_id in (
        'P_FOAM_K',
        'M_STAN_Q',
        'P_FOAM_S',
        'M_PREM_Q',
        'M_STAN_F',
        'M_STAN_T',
        'M_PREM_K',
        'M_PREM_F',
        'M_STAN_K',
        'M_PREM_T',
        'P_DOWN_S',
        'P_DOWN_K'
    )
);

```

```
SELECT * FROM transactions
```

## 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 an array or map-type objects. Higher-order functions allow you to transform data while preserving the original structure.

Higher order functions include:

- **FILTER** filters an array using the given lambda function.
- **EXIST** tests whether 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.
- **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 apply a finishing function on the final buffer.



We can use the **FILTER** function to create a new column that excludes that value from each array.

```
FILTER (items, i -> i.item_id LIKE "%K") AS king_items
```

In the statement above:

- **FILTER** : the name of the higher-order function
- **items** : the name of our input array
- **i** : the name of the iterator variable. You choose this name and then use it in the lambda function. It iterates over the array, cycling each value into the function one at a time.
- **->** : Indicates the start of a function
- **i.item\_id LIKE "%K"** : This is the function. Each value is checked to see if it ends with the capital letter K. If it is, it gets filtered into the new column, **king\_items**

-- filter for sales of only king-sized items

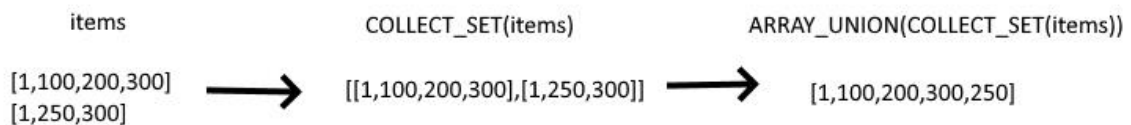
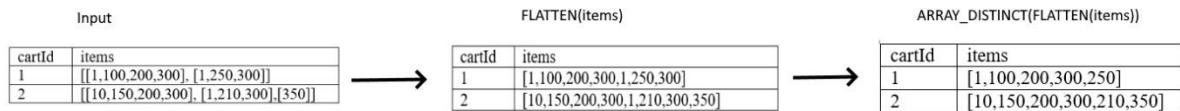
```
SELECT  
order_id,  
items,  
FILTER (items, i -> i.item_id LIKE "%K") AS king_items  
FROM sales
```

```
CREATE OR REPLACE TEMP VIEW king_size_sales AS
```

```
SELECT order_id, king_items  
FROM (  
  SELECT  
    order_id,  
    FILTER (items, i -> i.item_id LIKE "%K") AS king_items  
  FROM sales)  
WHERE size(king_items) > 0;
```

```
SELECT * FROM king_size_sales
```

- **FLATTEN** -> Transforms an array of arrays into a single array.
- **ARRAY\_DISTINCT** -> The function returns an array of the same type as the input argument where all duplicate values have been removed.



## TRANSFORM

**TRANSFORM** can be particularly useful when you want to apply an existing function to each element in an array.

Compute the total revenue from king-sized items per order.

**TRANSFORM(king\_items, k -> CAST(k.item\_revenue\_in\_usd \* 100 AS INT)) AS item\_revenues**

-- get total revenue from king items per order

CREATE OR REPLACE TEMP VIEW king\_item\_revenues AS

```
SELECT
  order_id,
  king_items,
  TRANSFORM (
    king_items,
    k -> CAST(k.item_revenue_in_usd * 100 AS INT)
  ) AS item_revenues
FROM king_size_sales;
```

SELECT \* FROM king\_item\_revenues

## DE 4.8 - SQL UDFs and Control Flow

Databricks added support for User Defined Functions (UDFs) registered natively in SQL starting in DBR 9.1.

This feature allows users to register custom combinations of SQL logic as functions in a database, making these methods **reusable anywhere SQL** can be run on Databricks. These functions leverage Spark SQL directly, maintaining all of the optimizations of Spark when applying your custom logic to large datasets.

```
CREATE OR REPLACE FUNCTION yelling(text STRING)
RETURNS STRING
RETURN concat(upper(text), "!!!")
```

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

```
DESCRIBE FUNCTION yelling
```

```
DESCRIBE FUNCTION EXTENDED yelling
```

**In order to use a SQL UDF, a user must have `USAGE` and `SELECT` permissions on the function.**

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

```
SELECT *,
CASE
  WHEN food = "beans" THEN "I love beans"
  WHEN food = "potatoes" THEN "My favorite vegetable is potatoes"
```

```
    WHEN food <> "beef" THEN concat("Do you have any good recipes for ", food ,"?")
    ELSE concat("I don't eat ", food)
END
FROM foods
```

## DE 5.1 - Python Basics

Databricks notebooks allow users to write SQL and Python and execute logic cell-by-cell. PySpark has extensive support for executing SQL queries, and can easily exchange data with tables and temporary views.

"This is a string"

```
print("This is a string")
```

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

```
print("""
This
is
a
multi-line
string
""")
```

When we execute SQL from a Python cell, we will pass a string as an argument to **spark.sql()**.

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.)

```
my_string = "This is a string"
```

```
print("This is a new string and " + my_string)
```

```
string1+"."+string2
```

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**.

```
def print_string(arg):
```

```
    print(arg)
```

```
def return_new_string(string_arg):
```

```
    return "The string passed to this function was " + string_arg
```

## **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 (**{}**).

```
name="This is Phani Reddy Janga"
```

```
f"hello {name}"
```

```
f"I can substitute functions like {return_new_string('foobar')} here"
```

```
table_name = "users"
```

```
filter_clause = "WHERE state = 'CA'"
```

```
query = f"""
```

```
SELECT *
```

```
FROM {table_name}
```

```
{filter_clause}
```

```
"""
```

```
print(query)
```

**More Info:** [Using print\(\) function in python](#)

## DE 5.2 - Python Control Flow

```
if food == "biryani":  
    print(f"I love {food}")  
else:  
    print("I love food")
```

```
if food == 'biryani':  
    print("biryani")  
elif food == 'potatoes':  
    print("vegetables")  
else:  
    print("others")
```

While **if / else** clauses allow us to define conditional logic based on evaluating conditional statements, **try / except** focuses on providing robust error handling.

```
def a(number):  
    return a*3
```

```
try:  
    {  
        a("testing")  
    }  
except:  
    {  
        print("except")  
    }
```

### Assert

**Assert** statements allow us to run simple tests of Python code. If an **assert** statement evaluates to true, nothing happens.

```
assert type(2) == int
```

```
def simple_query_function(query, preview=True):  
    query_result = spark.sql(query)  
    if preview:  
        display(query_result)  
    return query_result
```

### find() method

We can use the **find()** method to test for multiple SQL statements by looking for a semicolon.

```
injection_query = "SELECT * FROM demo_tmp_vw; DROP DATABASE prod_db  
CASCADE; SELECT * FROM demo_tmp_vw"
```

```
injection_query.find(";")
```

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



## DE 6.1 - Incremental Data Ingestion with Auto Loader

Auto Loader is configured to incrementally process files from a directory in cloud object storage into a Delta Lake table.

Auto Loader with automatic schema inference and evolution, the 4 arguments

argument	what it is	how it's used
<code>data_source</code>	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 <code>.load()</code> method
<code>source_format</code>	The format of the source data	While the format for all Auto Loader queries will be <code>cloudFiles</code> , the format of the source data should always be specified for the <code>cloudFiles.format</code> option
<code>table_name</code>	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 <code>.table()</code> method. Note that you can either append to an existing table or create a new table
<code>checkpoint_directory</code>	The location for storing metadata about the stream	This argument is passed to the <code>checkpointLocation</code> and <code>cloudFiles.schemaLocation</code> options. Checkpoints keep track of streaming progress, while the schema location tracks updates to the fields in the source dataset

```
spark.readStream
  .format("cloudFiles")
  .option("cloudFiles.format", source_format)
  .option("cloudFiles.schemaLocation", checkpoint_directory)
  .load(data_source)
  .writeStream
  .option("checkpointLocation", checkpoint_directory)
  .option("mergeSchema", "true")
  .table(table_name))
```

## DE 6.2 - Reasoning about Incremental Data

Spark Structured Streaming is optimized on Databricks to integrate closely with Delta Lake and Auto Loader.

Structured Streaming ensures **end-to-end exactly-once** fault-tolerance guarantees through **checkpointing (discussed below) and Write Ahead Logs**.

Structured Streaming sources, sinks, and the underlying execution engine work together to track the progress of stream processing. If a failure occurs, the streaming engine attempts to restart and/or reprocess the data.

At a high level, the underlying streaming mechanism relies on a couple of approaches:

First, Structured Streaming uses checkpointing and write-ahead logs to record the offset range of data being processed during each trigger interval.

Next, the streaming sinks are designed to be idempotent - that is, multiple writes of the same data (as identified by the offset) do not result in duplicates being written to the sink.

Taken together, replayable data sources and idempotent sinks allow Structured Streaming to ensure end-to-end, exactly-once semantics under any failure condition.

The `spark.readStream()` method returns a `DataStreamReader` used to configure and query the stream.

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

### Following Command to stop all active streaming queries.

for s in **spark. streams. Active:**

```
    print("Stopping " + s.id)
    s.stop()
    s.awaitTermination()
```

Consider the model of the data as a constantly appending table. **Sorting is one of a handful of operations that is either too complex or logically not possible** to do when working with streaming data.

## Unsupported Operations

There are a few `DataFrame/Dataset` operations that are not supported with streaming `DataFrames/Datasets`. Some of them are as follows.

- Multiple **streaming aggregations** (i.e. a chain of aggregations on a streaming DF) are not yet supported on streaming `Datasets`.
- **Limit and take the first N rows** are not supported on streaming `Datasets`.
- **Distinct operations on streaming Datasets** are not supported.
- **Deduplication operation is not supported** after aggregation on a streaming `Datasets`.
- **Sorting operations are supported on streaming Datasets only after an aggregation and in Complete Output Mode.**
- Few types of outer joins on streaming `Datasets` are not supported. See the [support matrix in the Join Operations section](#) for more details.

To persist the results of a streaming query, we must write them out to durable storage. The `DataFrame.writeStream` method returns a `DataStreamWriter` used to configure the output.

## Output Modes

Streaming jobs have output modes similar to static/batch workloads

Mode	Example	Notes
Append	<code>.outputMode("append")</code>	<b>This is the default.</b> Only newly appended rows are incrementally appended to the target table with each batch
Complete	<code>.outputMode("complete")</code>	The Results Table is recalculated each time a write is triggered; the target table is overwritten with each batch

## Trigger Intervals

When defining a streaming write, the trigger method specifies when the system should process the next set of data

Trigger Type	Example	Behavior
Unspecified		<b>This is the default.</b> This is equivalent to using <code>processingTime="500ms"</code>
Fixed interval micro-batches	<code>.trigger(processingTime="2 minutes")</code>	The query will be executed in micro-batches and kicked off at the user-specified intervals
Triggered micro-batch	<code>.trigger(once=True)</code>	The query will execute a single micro-batch to process all the available data and then stop on its own
Triggered micro-batches	<code>.trigger(availableNow=True)</code>	The query will execute multiple micro-batches to process all the available data and then stop on its own

NOTE: Trigger.AvailableNow is a new trigger type that is available in DBR 10.1 for Scala only and available in DBR 10.2 and above for Python and Scala.

```
(spark.table("device_counts_tmp_vw")
  .writeStream
  .option("checkpointLocation", f"{DA.paths.checkpoints}/silver")
  .outputMode("complete")
  .trigger(availableNow=True)
  .table("device_counts")
  .awaitTermination() # This optional method blocks execution of the next cell until the incremental
  batch write has succeeded
)
```

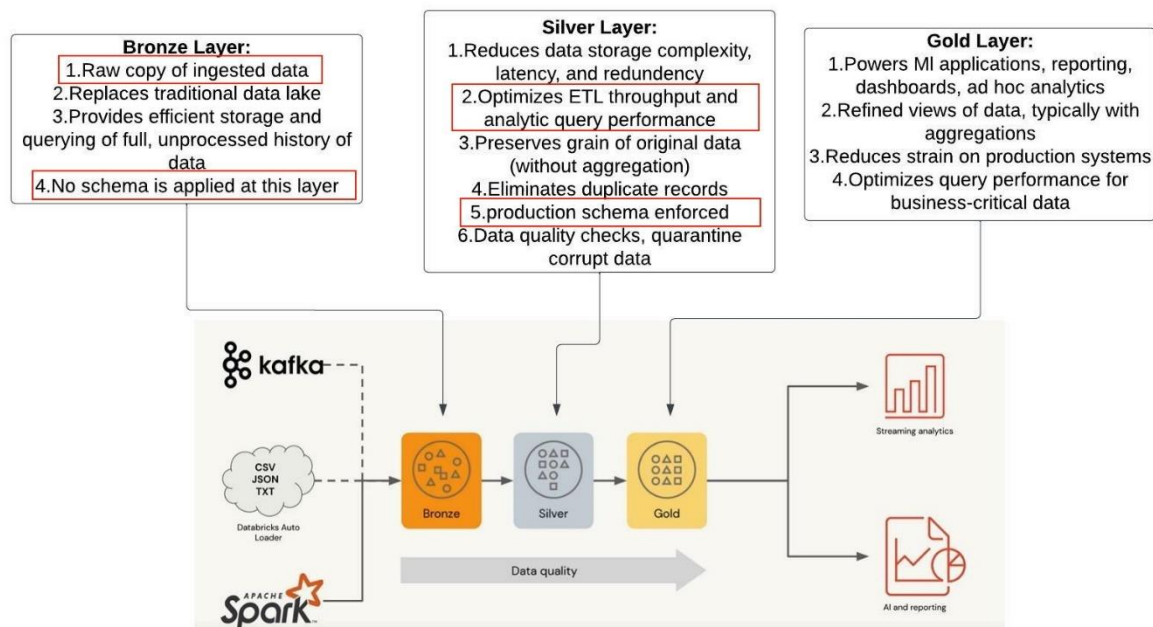
Validation by using Assert.

```
from pyspark.sql import Row
```

```
assert Row(tableName="customer_count_by_state_temp", isTemporary=True) in spark.sql("show tables").select("tableName", "isTemporary").collect(), "Table not present or not temporary"
```

```
assert spark.table("customer_count_by_state_temp").dtypes == [('state', 'string'), ('customer_count', 'bigint')], "Incorrect Schema"
```

## DE 7.1 - Incremental Multi-Hop in the Lakehouse



- **Bronze** tables contain **raw data ingested** from various sources (JSON files, RDBMS data, IoT data, to name a few examples).
- **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 the department.

In a multi-hop architecture, the **Bronze table** mainly performs the task of **applying schema** over the ingested data. Rest of the options are all false. The following are the major tasks of each table.

**Bronze** – *Apply schema to the raw data*

**Silver** – Conversion of timestamps into human readable format and performing joins with other tables

**Gold** – Performing aggregations

As part of **Silver table enrichments**, **timestamps** are converted to **human-readable** format. So, the correct hop from the above options is **Bronze to Silver** hop.

The main functionalities of other hops in multi-hop architecture.

**Raw to Bronze** – Apply **schema** to the raw data.

**Bronze to Silver** – Conversion of **timestamps** into **human-readable** format and performing **joins** with other tables.

**Silver to Gold** – Performing **aggregations**.

More Info: [Medallion architecture in Databricks](#)

**NOTE:** For a JSON data source, Auto Loader will default to inferring each column as a string. Here, we demonstrate specifying the data type for the **time** column using the **cloudFiles.schemaHints** option. Note that specifying improper types for a field will result in null values.

**Autoloader:**

```
(spark.readStream
  .format("cloudFiles")
  .option("cloudFiles.format", "json")
  .option("cloudFiles.schemaHints", "time DOUBLE")
  .option("cloudFiles.schemaLocation", f"{DA.paths.checkpoints}/bronze")
  .load(DA.paths.data_landing_location)
  .createOrReplaceTempView("recordings_raw_temp"))
```

**Spark:**

```
(spark.read
  .format("csv")
  .schema("mrn STRING, name STRING")
  .option("header", True)
  .load(f"{DA.paths.data_source}/patient/patient_info.csv")
  .createOrReplaceTempView("pii"))

(spark.table("recordings_w_pii")
  .writeStream
  .format("delta")
  .option("checkpointLocation", f"{DA.paths.checkpoints}/recordings_enriched"))
```

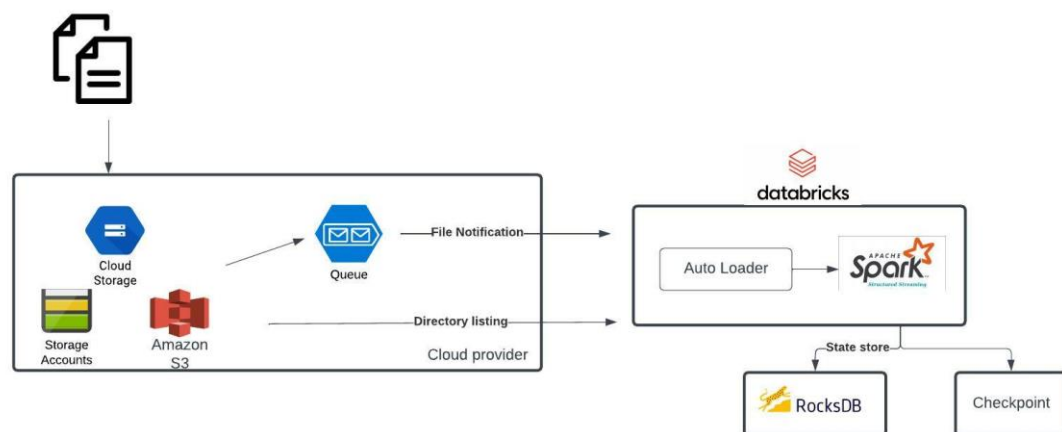
```
.outputMode("append")  
.table("recordings_enriched"))  
  
(spark.table("patient_avg")  
  .writeStream  
  .format("delta")  
  .outputMode("complete")  
  .option("checkpointLocation", f"{DA.paths.checkpoints}/daily_avg")  
  .trigger(availableNow=True)  
  .table("daily_patient_avg"))
```

When using **complete** output mode, we rewrite the entire state of our table each time our logic runs.

## About Autoloader

AutoLoader Efficiently process new data incrementally from cloud object storage, AUTO LOADER only supports ingesting files stored in a cloud object storage. Auto Loader cannot process streaming data sources like Kafka or Delta streams, use Structured streaming for these data sources.

### Auto Loader & Cloud Storage Integration



\*Directory listing also supports incremental file listing

### Auto Loader and Cloud Storage Integration

Auto Loader supports a couple of ways to ingest data incrementally

1. **Directory listing** - List Directory and maintain the state in **RocksDB**, supports incremental file listing
2. **File notification** - Uses a trigger+queue to store the file notification which can be later used to retrieve the file, unlike Directory listing File notification can scale up to millions of files per day.

## Auto Loader vs COPY INTO?

### Auto Loader

Auto Loader incrementally and efficiently processes new data files as they arrive in cloud storage without any additional setup.

Auto Loader provides a new Structured Streaming source called cloudFiles. Given an input directory path on the cloud file storage, the cloudFiles source automatically processes new files as they arrive, with the option of also processing existing files in that directory.

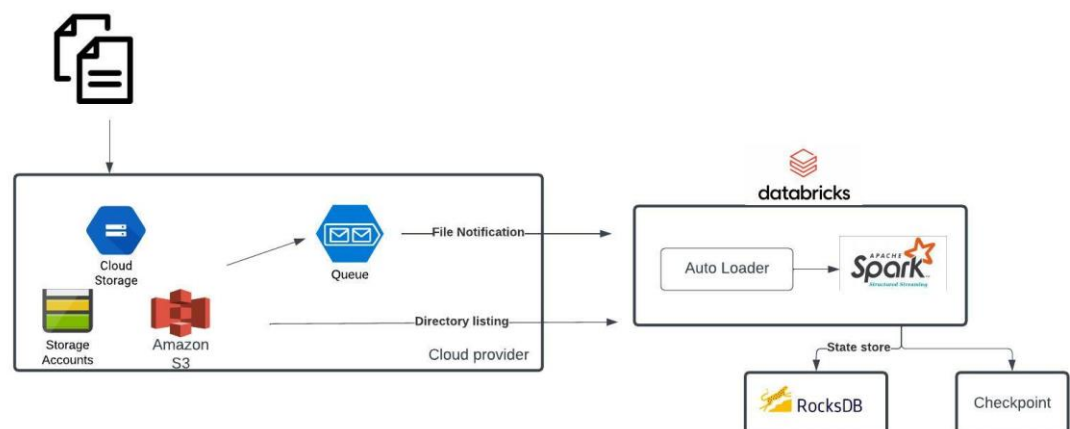
### When to use Auto Loader instead of the COPY INTO?

- You want to load data from a file location that contains files in the order of millions or higher. Auto Loader can discover files more efficiently than the COPY INTO SQL command and can split file processing into multiple batches.
- You do not plan to load subsets of previously uploaded files. With Auto Loader, it can be more difficult to reprocess subsets of files. However, you can use the COPY INTO SQL command to reload subsets of files while an Auto Loader stream is simultaneously running.

### Auto Loader Modes?

- Auto Loader supports two modes when ingesting new files from cloud object storage
- Directory listing: Auto Loader identifies new files by listing the input directory, and uses a directory polling approach.
- File notification: Auto Loader can automatically set up a notification service and queue service that subscribe to file events from the input directory.

## Auto Loader & Cloud Storage Integration



\*Directory listing also supports incremental file listing



- File notification is more efficient and can be used to process the data in real-time as data arrives in cloud object storage.

[Choosing between file notification and directory listing modes | Databricks on AWS](#)

## Lakehouse Arch Advantages:

Lakehouse combines the benefits of a data warehouse and data lakes,

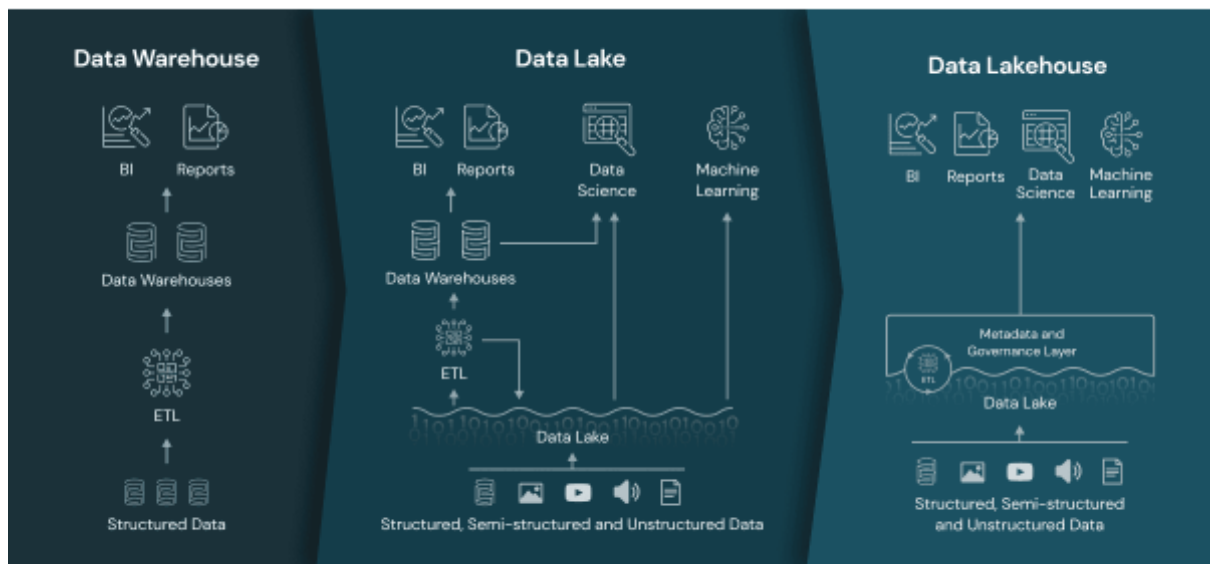
**Lakehouse = Data Lake + DataWarehouse**

Here are some of the major benefits of a lakehouse

**A lakehouse has the following key features:**

- **Transaction support:** In an enterprise lakehouse many data pipelines will often be reading and writing data concurrently. Support for ACID transactions ensures consistency as multiple parties concurrently read or write data, typically using SQL.
- **Schema enforcement and governance:** The Lakehouse should have a way to support schema enforcement and evolution, supporting DW schema architectures such as star/snowflake-schemas. The system should be able to **reason about data integrity**, and it should have robust governance and auditing mechanisms.
- **BI support:** Lakehouses enable using BI tools directly on the source data. This reduces staleness and improves recency, reduces latency, and lowers the cost of having to operationalize two copies of the data in both a data lake and a warehouse.
- **Storage is decoupled from compute:** In practice this means storage and compute use separate clusters, thus these systems are able to scale to many more concurrent users and larger data sizes. Some modern data warehouses also have this property.
- **Openness:** The storage formats they use are open and standardized, such as Parquet, and they provide an API so a variety of tools and engines, including machine learning and Python/R libraries, can efficiently access the data **directly**.
- **Support for diverse data types ranging from unstructured to structured data:** The lakehouse can be used to store, refine, analyze, and access data types needed for many new data applications, including images, video, audio, semi-structured data, and text.
- **Support for diverse workloads:** including data science, machine learning, and SQL and analytics. Multiple tools might be needed to support all these workloads but they all rely on the same data repository.
- **End-to-end streaming:** Real-time reports are the norm in many enterprises. Support for streaming eliminates the need for separate systems dedicated to serving real-time data applications.

**Lakehouse = Data Lake + DataWarehouse**



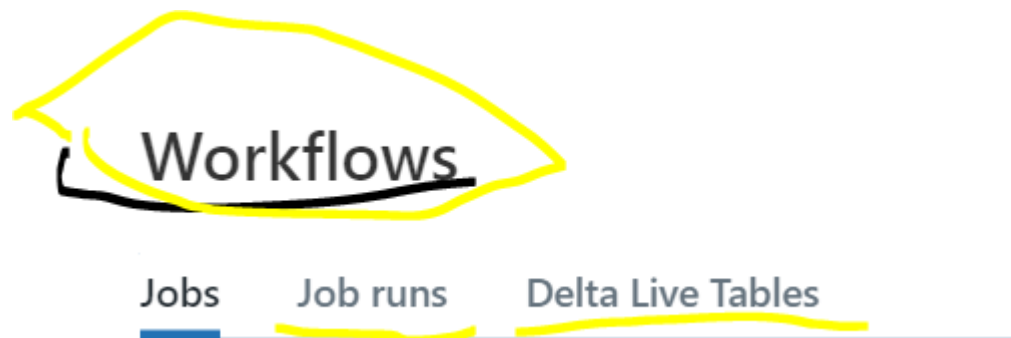
What is Delta?

Delta lake is

- Open source
- Builds up on standard data format
- Optimized for cloud object storage
- Built for scalable metadata handling

Delta lake is not

- Proprietary technology
- Storage format
- Storage medium
- Database service or data warehouse



create a pipeline using a notebook

1. Click the **Jobs** button on the sidebar.
2. Select the **Delta Live Tables** tab.
3. Click **Create Pipeline**.

## Pipeline Modes

This field specifies how the pipeline will be run.

**Triggered** pipelines run once and then shut down until the next manual or scheduled update.

**Continuous** pipelines run continuously, ingesting new data as it arrives. Choose the mode based on latency and cost requirements.

More Info: [Triggered vs Continuous pipelines in Databricks](#)

Pipeline mode ?

☒ Triggered ☐ Continuous

## Enable the autoscaling

box, and set the number of workers to **1** (one).

- **Enable autoscaling**, **Min Workers** and **Max Workers** control the worker configuration for the underlying cluster processing the pipeline. Notice the DBU estimate provided, similar to that provided when configuring interactive clusters.

☒ Enable autoscaling ⓘ  
[Learn how to enable Enhanced Autoscaling](#) [Preview](#)

Cluster

\* Min workers   \* Max workers

## DE 8.1.2 - SQL for Delta Live Tables

DLT tables and views will always be preceded by the LIVE keyword.

Incremental processing via Auto Loader (which uses the same processing model as Structured Streaming), requires the addition of the STREAMING keyword in the declaration as seen below. The cloud\_files() method enables Auto Loader to be used natively with SQL. This method takes the following positional parameters:

- The source location, as mentioned above
- The source data format, which is JSON in this case
- An arbitrarily sized array of optional reader options. In this case, we set cloudFiles.inferColumnTypes to true
  - CREATE OR REFRESH STREAMING LIVE TABLE sales\_orders\_raw
  - COMMENT "The raw sales orders, ingested from /databricks-datasets."
  - AS SELECT \* FROM cloud\_files("/databricks-datasets/retail-org/sales\_orders/", "json", map("cloudFiles.inferColumnTypes", "true"))
- Bronze tables represent data in its rawest form, but captured in a format that can be retained indefinitely and queried with the performance and benefits that Delta Lake has to offer.
- Silver tables represent a refined copy of data from the bronze layer, with the intention of optimizing downstream applications. At this level, we apply operations like data cleansing and enrichment.
- Gold table delivering an aggregation with business value

## Quality Control

### Constraints

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 a constraint is violated
DROP ROW	Discard records that violate constraints

ON VIOLATION	Behavior
Omitted	Records violating constraints will be included (but violations will be reported in metrics)

References to streaming DLT tables use the **STREAM()**, supplying the table name as an argument.

```
CREATE OR REFRESH STREAMING LIVE TABLE sales_orders_cleaned(
CONSTRAINT valid_order_number EXPECT (order_number IS NOT NULL) ON VIOLATION
DROP ROW
```

```
)
```

```
COMMENT "The cleaned sales orders with valid order_number(s) and partitioned by
order_datetime."
```

```
AS
```

```
  SELECT f.customer_id, f.customer_name, f.number_of_line_items,
         timestamp(from_unixtime((cast(f.order_datetime as long)))) as order_datetime,
         date(from_unixtime((cast(f.order_datetime as long)))) as order_date,
         f.order_number, f.ordered_products, c.state, c.city, c.lon, c.lat, c.units_purchased,
```

```
c.loyalty_segment
```

```
  FROM STREAM(LIVE) sales_orders_raw) f
```

```
  LEFT JOIN LIVE.customers c
```

```
    ON c.customer_id = f.customer_id
```

```
    AND c.customer_name = f.customer_name
```

```
CREATE OR REFRESH LIVE TABLE sales_order_in_la
```

```
COMMENT "Sales orders in LA."
```

```
AS
```

```
  SELECT city, order_date, customer_id, customer_name, ordered_products_explode.curr,
         sum(ordered_products_explode.price) as sales,
         sum(ordered_products_explode.qty) as quantity,
         count(ordered_products_explode.id) as product_count
```

```
  FROM (SELECT city, order_date, customer_id, customer_name, explode(ordered_products) as
ordered_products_explode
```

```
        FROM LIVE.sales_orders_cleaned
```

```
        WHERE city = 'Los Angeles')
```

```
  GROUP BY order_date, city, customer_id, customer_name, ordered_products_explode.curr
```

```
%sql
```

```
SELECT * FROM delta.`${da.paths.storage_location}/system/events`
```

```
%sql
```

```
SELECT * FROM ${da.db_name}.sales_order_in_la
```

### *DLT Expectations*

Delta live tables support three types of expectations to fix bad data in DLT pipelines

## DLT Expectations

```
%sql
CREATE OR REPLACE LIVE TABLE orders_valid
(CONSTRAINT valid_timestamp EXPECT (timestamp > '2020-01-01'))
SELECT * FROM LIVE.orders_vw

%python
@dlt.expect("valid timestamp", "timestamp > '2012-01-01'")
```

### Retain Invalid Records

Records violate expectations are  
Added to target table  
Flagged in invalid in event log  
Pipeline continues

```
%sql
CREATE OR REPLACE LIVE TABLE orders_valid
(CONSTRAINT valid_timestamp EXPECT (timestamp > '2020-01-01') ON VIOLATION DROP ROW)
SELECT * FROM LIVE.orders_vw

%python
@dlt.expect_or_drop("valid timestamp", "timestamp > '2012-01-01'")
```

### Drop Invalid Records

Records violate expectation are  
Dropped from target table  
Flagged invalid in event log  
Pipeline Continues

```
%sql
CREATE OR REPLACE LIVE TABLE orders_valid
(CONSTRAINT valid_timestamp EXPECT (timestamp > '2020-01-01') ON VIOLATION FAIL)
SELECT * FROM LIVE.orders_vw

%python
@dlt.expect_or_fail("valid timestamp", "timestamp > '2012-01-01'")
```

### Fail on Invalid Records

Records violate expectation  
Cause the job to fail

syntax for all three:

1. **CONSTRAINT constraint\_name EXPECT (condition) ON VIOLATION FAIL UPDATE** – As soon as the condition is **not** met, the pipeline will **fail**
2. **CONSTRAINT constraint\_name EXPECT (condition) ON VIOLATION DROP ROW** – If the condition is **not** met, the **record will be dropped**
3. **CONSTRAINT constraint\_name EXPECT (condition)** – The **row will be added to the target table** but the record will be shown under **failed\_records** in the **Data Quality dashboard**.

As the question states that you need to **drop** the row violating the **NOT NULL** constraint of the **temperature** column, adding **CONSTRAINT temp\_in\_range EXPECT (temperature is NOT NULL) ON VIOLATION DROP ROW** will get the desired results.

More Info: [Data quality constraints in Delta Live Tables](#)

### DE 9.1.1 - Task Orchestration with Databricks Jobs

Databricks Jobs UI has the ability to schedule multiple tasks as part of a job, allowing Databricks Jobs to fully handle orchestration for most production workloads.

**Note:** When selecting your all-purpose cluster, you will get a warning about how this will be billed as all-purpose compute.

Production jobs should always be scheduled against new job clusters appropriately sized for the workload, as this is billed at a much lower rate.

Steps:

1. Navigate to the Workflows-> Jobs UI using the Databricks left side navigation bar.
2. Click the blue **Create Job** button
3. Configure the task:
  1. Enter **reset** for the task name
  2. Select the notebook **DE 9.1.2 - Reset** using the notebook picker.
  3. From the **Cluster** dropdown, under **Existing All Purpose Clusters**, select your cluster
  4. Click **Create**
4. In the top-left of the screen rename the job (not the task) from **reset** (the defaulted value) to the **Job Name** provided for you in the previous cell.
5. Click the blue **Run now** button in the top right to start the job.

Job UI provides extensive options for setting up **chronological scheduling of your Jobs**. Settings configured with the UI can also be output in **chron** syntax, which can be edited if a custom configuration not available with the UI is needed.

To Review the Job Run

1. Select the **Runs** tab in the top-left of the screen (you should currently be on the **Tasks** tab)
2. Find your job. If **the job is still running**, it will be under the **Active runs** section. If **the job finished running**, it will be under the **Completed runs** section
3. Open the Output details by click on the timestamp field under the **Start time** column
4. If **the job is still running**, you will see the active state of the notebook with a **Status** of **Pending**
5. **ng** or **Running** in the right side panel. If **the job has completed**, you will see the full execution of the notebook with a **Status** of **Succeeded** or **Failed** in the right side panel

While setting up all purpose cluster a warning message would display, "Jobs running on all-purpose cluster are considered all purpose compute"

Pricing for All-purpose clusters are more expensive than the job clusters



Workflows > Jobs > Create

## Add a name for your job...

Runs **Tasks**

Task name \* ⓘ

test\_job




Type \*

Notebook | ▾


Source \* ⓘ


Workspace | ▾

Path \* ⓘ

/Users/[REDACTED]/Data Analysis   

Cluster \* ⓘ

SingleNode (DBR 10.4 LTS | Spark 3.2.1 | Scala 2.12)  | ▾

 Jobs running on all-purpose clusters are considered all-purpose compute. [Learn more](#)

Parameters ⓘ


UI | JSON

Add

▼ Advanced options

Cancel

Create

aws		Standard	Premium	Enterprise
		One platform for your data analytics and ML workloads	Data analytics and ML at scale across your business	Data analytics and ML for your mission critical workloads
CLASSIC COMPUTE	Jobs Light Compute Run data engineering pipelines to build data lakes. 	\$0.07 / DBU	\$0.10 / DBU	\$0.13 / DBU
	Jobs Compute Jobs Compute Photon Run data engineering pipelines to build data lakes and manage data at scale.	\$0.10 / DBU	\$0.15 / DBU	\$0.20 / DBU
	Delta Live Tables Delta Live Tables Photon Easily build high quality streaming or batch ETL pipelines using Python or SQL with the DLT Edition that is best for your workload. <a href="#">Learn more</a>	\$0.20 - \$0.36 / DBU	\$0.20 - \$0.36 / DBU	\$0.20 - \$0.36 / DBU
	SQL Compute Run SQL queries for BI reporting, analytics and visualization to get timely insights from data lakes.	-	\$0.22 / DBU	\$0.22 / DBU
	All-Purpose Compute All-Purpose Compute Photon Run interactive data science and machine learning workloads. Also good for data engineering, BI and data analytics.	\$0.40 / DBU	\$0.55 / DBU	\$0.65 / DBU
US East (N. Virginia)				

You have the ability to view current active runs or completed runs, once you click the run you can see the

Workflows > Jobs > run\_process\_all >

### run\_process\_all

Runs

Active runs

Start time	Run ID	Launched	Operator	Status	Actions
Aug 7 2025 1:30:04 EDT	17020	Manually	tek-kl	Running	

Completed runs (past 60 days)

Start time	Run ID	Launched	Operator	Status	Actions
Aug 12 2025 1:30:04 EDT	14098	Manually	tek-kl	Succeeded	

Click on the run to view the notebook output

Workflows > Jobs > run\_process\_all > Run 17020 > accounting >

accounting run

Original (latest) · ✓ Succeeded

## Output

```
from time import sleep

dbutils.widgets.text("param1", "default")
param1 = dbutils.widgets.get("param1")
sleep(20)
dbutils.notebook.exit(f"param1 passed as {param1}")
```

Notebook exited: param1 passed as default

Command took 20.41 seconds

Magic command **%run** to call an additional notebook using a relative path1

```
%run ../Includes/Classroom-Setup-9.1.1
```

The **system** directory captures events associated with the pipeline.

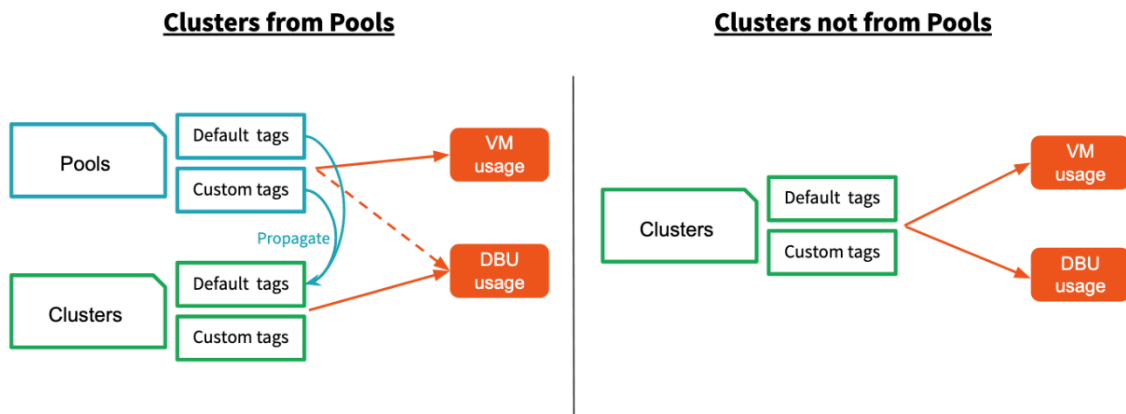
```
files = dbutils.fs.ls(f"{DA.paths.working_dir}/storage/system/events")  
display(files)
```

## Databricks Object Tagging Hierarchy

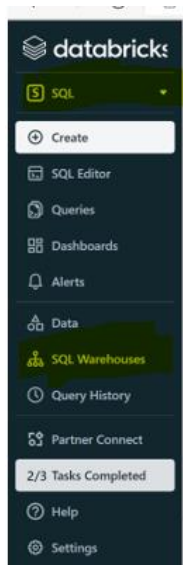
Use Tags, during job creation so cost can be easily tracked

# Databricks Object Tagging Hierarchy

— Direct pass  
- - Indirect pass



## DE 10.1 - Navigating Databricks SQL and Attaching to Endpoints



## New SQL Warehouse ⓘ



Name

Cluster size ⓘ  4 DBU ▼

Auto stop ☒ After  minutes of inactivity.

Scaling ⓘ Min.  Max.  clusters (4 DBU)

---

Advanced options ▼

Tags ⓘ

---

Advanced options ▼

Tags ⓘ

Unity Catalog ☒

Spot instance policy ⓘ  ▼

Channel ⓘ

Cost optimized ✓

Cost optimized ✓

Reliability optimized

- Scaling: Scale clusters to handle more concurrent users.
- Tags: These tags are automatically added to cluster instances for tracking usage in your cloud provider.
- Spot Instance Policy:

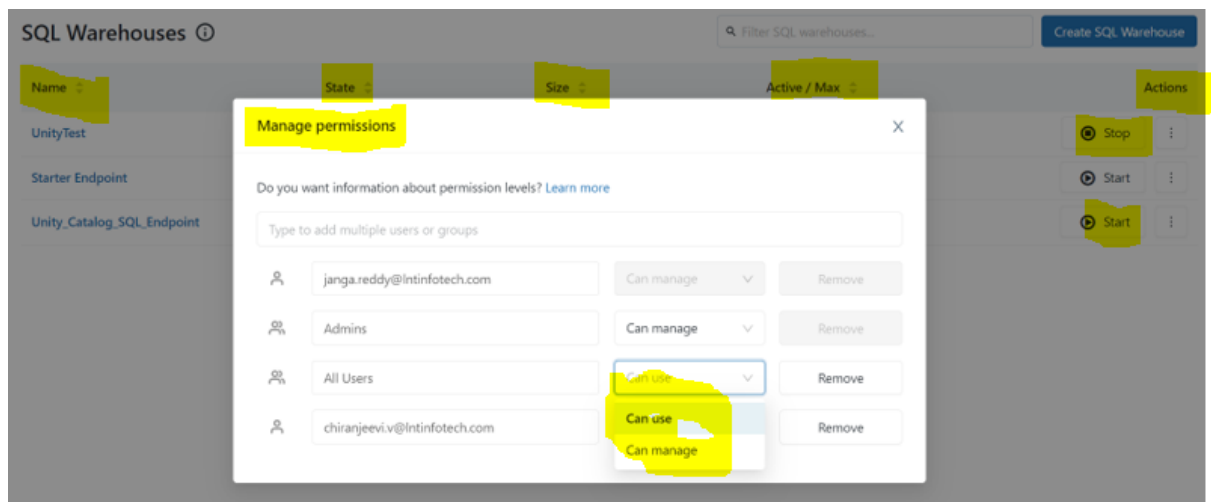
### Cost-optimized

Uses spot instances at 100% bid price for all workers, with auto-fallback to on-demand instances. If a spot instance is reclaimed, queries running on that instance will need to be resubmitted.

### Reliability optimized

Uses on-demand instances for all workers.

- Channels: Channels are used to introduce new features and releases while minimizing disruption.



Name	State	Size	Active / Max	Actions
UnityTest	Running	X-Small	1 / 1	Stop
Starter Endpoint	Stopped	Small	0 / 1	Start
Unity_Catalog_SQL_Endpoint	Stopped	X-Small	0 / 1	Start

The screenshot also shows a context menu for the 'Start' button of the 'Unity\_Catalog\_SQL\_Endpoint' row, with options: Permissions, Edit, Delete, and Query history.

- Techniques help in reducing **SQL endpoint** cost.
  - Select the minimum cluster size
  - The cost of an endpoint is measured in DBU/hour. Thus, decreasing the cluster size will help in decreasing the cost as the number of DBUs are proportional to the cluster size.
  - Turn on the Auto-stop feature
  - The cost of a SQL endpoint also depends on the number of hours it remains active. In order to decrease the cost, you can turn on the Auto-stop feature which automatically turns off the endpoint after the specified minutes of inactivity.
  - Select minimum values for Scaling the endpoint
  - By selecting minimum values for scaling, the number of DBUs decreases which, in turn, decreases the overall cost.
- As per the latest changes from Databricks, SQL endpoint has been renamed to SQL warehouse.**
- More Info: [Reducing cost of a SQL endpoint\(now known as SQL warehouse\)](#)

## DBSQL Dashboard

Query History

6 queries

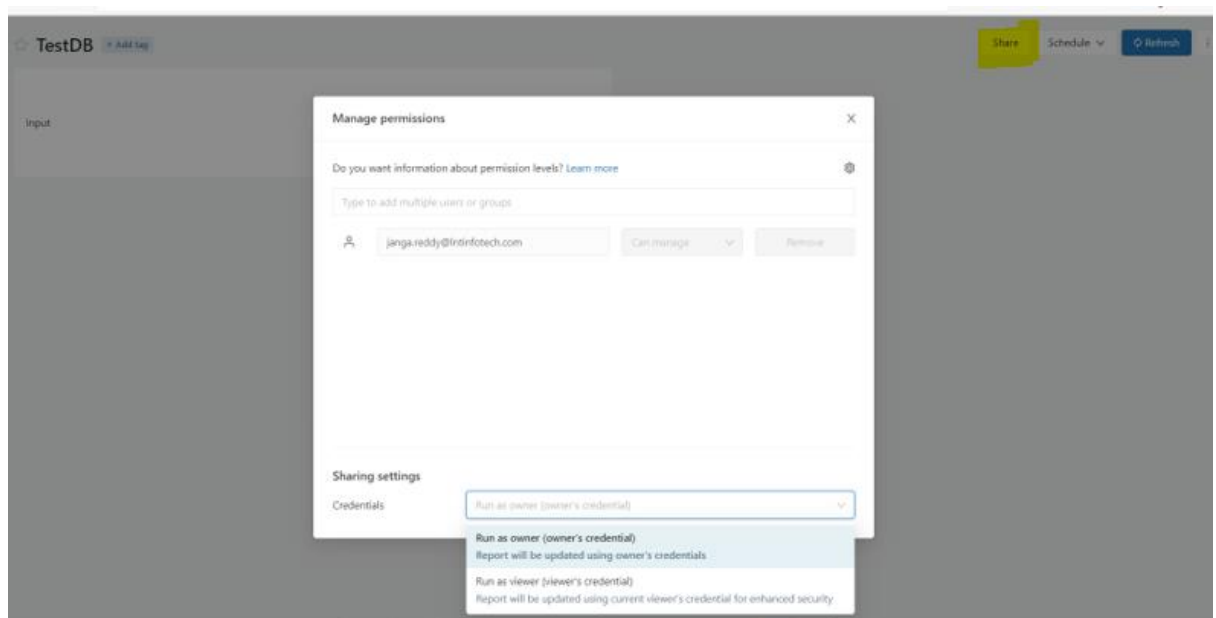
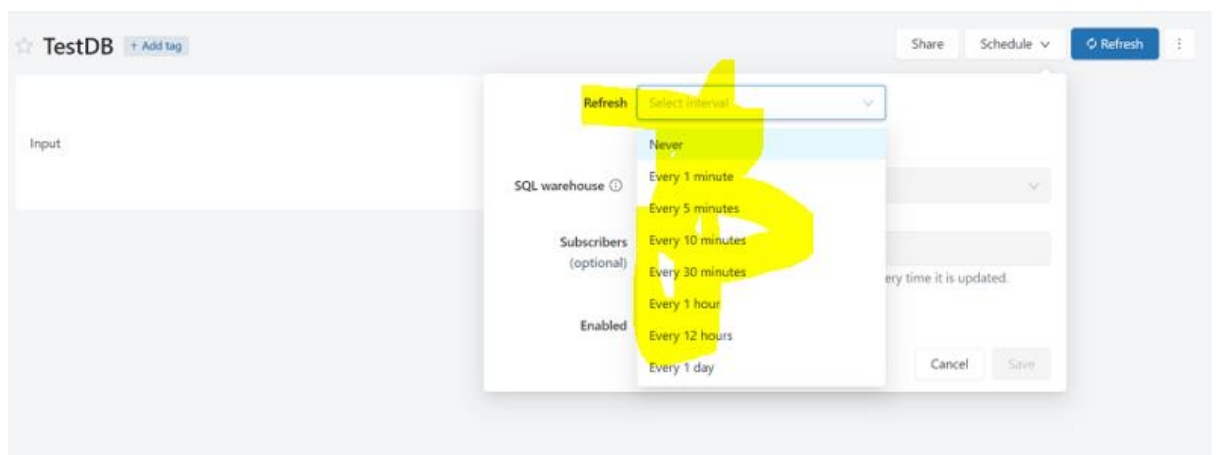
All users

Last 14 days

Unity\_Catalog\_SQL\_Endpoint

All statuses

Query	Started at	Duration	SQL Warehouse	User
SHOW FUNCTIONS	2022-08-08 10:34	1.02 s	Unity_Catalog_SQL_Endp...	janga.reddy@intinfotech.com
show databases	2022-08-08 10:34	469 ms	Unity_Catalog_SQL_Endp...	janga.reddy@intinfotech.com
SHOW DATABASES	2022-08-08 10:34	7.23 s	Unity_Catalog_SQL_Endp...	janga.reddy@intinfotech.com



Databricks many types of parameters in the dashboard, a drop-down list can be created based on a query that has a unique list of store locations.



Query Parameter types

Here is a simple query that takes a parameter for

```
SELECT * FROM sales WHERE field IN ( {{ Multi Select Parameter }} )
```

Or

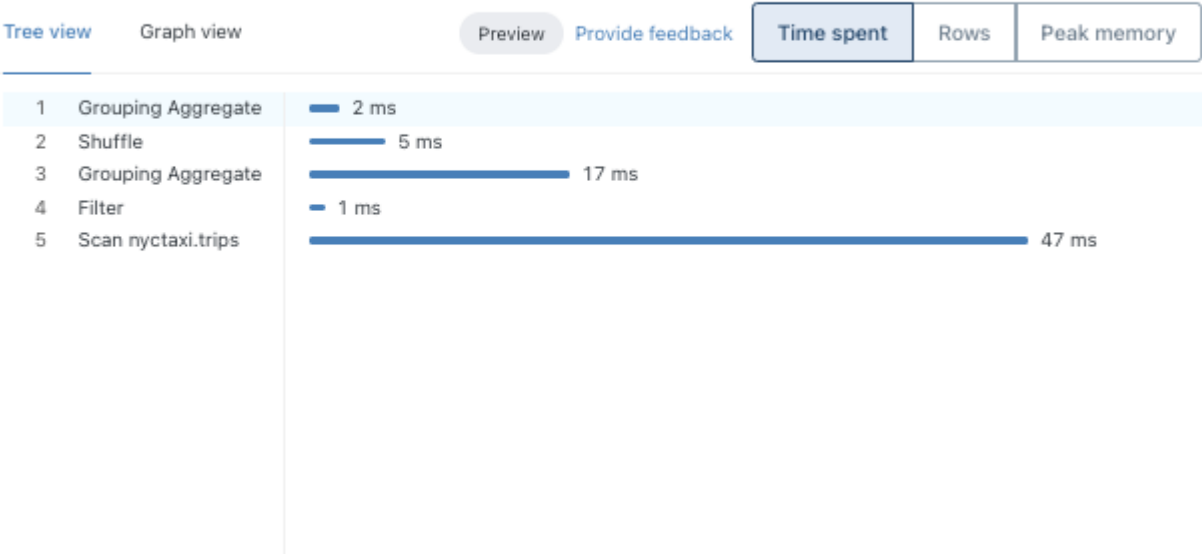
```
SELECT * FROM sales WHERE field = {{ Single Select Parameter }}
```

Query parameter types

- Text
- Number
- Dropdown List
- Query Based Dropdown List
- Date and Time

Databricks SQL query profile is much different to Spark UI and provides much more clear information on how time is being spent on different queries.

> ID: 01ec788b-b2ae-1485-a5ef-17dca9137d8b Finished | jdoe@example.com



Grouping Aggregate

Photon operation

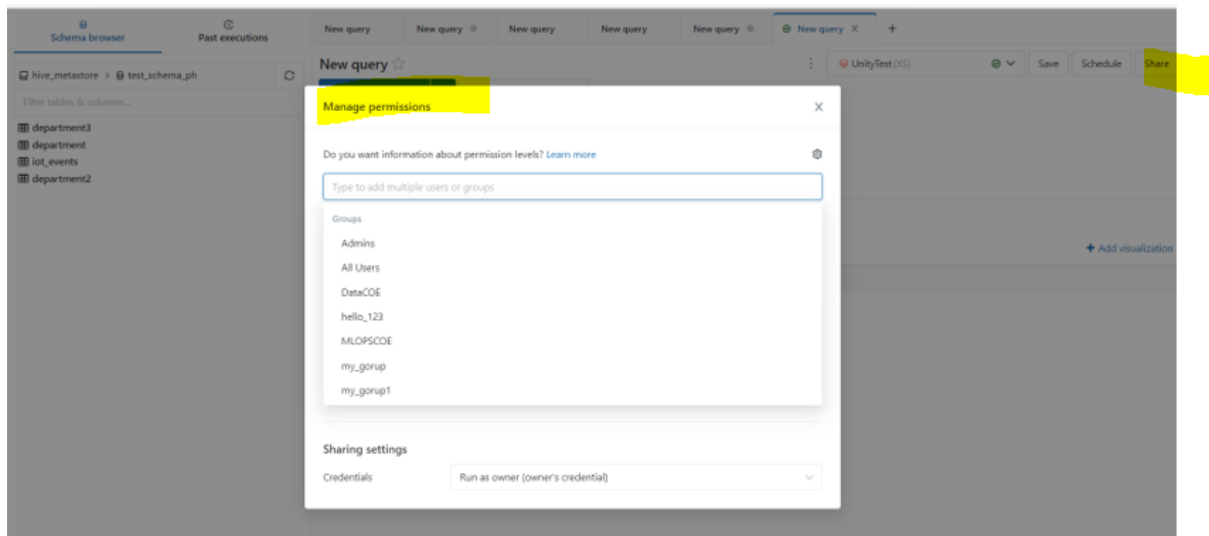
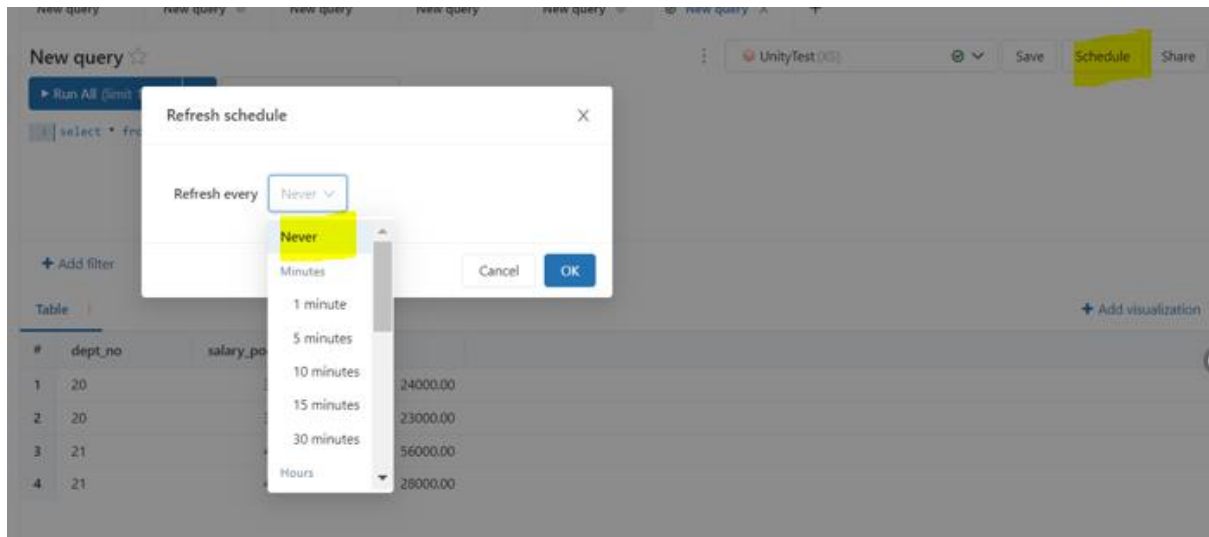
Time spent  
Rows  
Peak memory

Metrics

Time in produce  
Number of output batches  
Time in hash build  
Num bytes spilled to disk due to  
memory pressure  
Number of output rows  
Peak memory usage  
Cumulative time



## Query



# Alerts

## New Alert

Start by selecting the query that you would like to monitor using the search bar. [Setup Instructions ?](#)  
Keep in mind that Alerts do not work with queries that use parameters.

Query : 

Search a query by name

Create Alert



Start by selecting the query that you would like to monitor using the search bar. [Setup Instructions ?](#)  
Keep in mind that Alerts do not work with queries that use parameters.

Query : 

Generic Query 4

⚠ This query has no refresh schedule. [Why it's recommended ?](#)

	Value column	Condition	Threshold
Trigger when:	<div>value</div>	<div>&gt;</div>	<div>1</div>

Top row value is

When triggered, send notification:

Just once

Template:

Use default template

Create Alert

## DE 11.1 - Managing Permissions for Databases, Tables, and Views

Databricks table ACLs let **data engineers programmatically grant and revoke access to tables**.

### What is the Data Explorer?

The data explorer allows users and admins to:

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

By default, admins will have the ability to view all objects registered to the metastore and will be able to control permissions for other users in the workspace.

### 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).

To enable the ability to create databases and tables in the default catalog using Databricks SQL, have a workspace admin run the following command in the DBSQL query editor:

```
GRANT usage, create ON CATALOG `hive_metastore` TO `users`
```

To confirm this has run successfully, execute the following query:

```
SHOW GRANT ON CATALOG `hive_metastore`
```

- **REVOKE [privilege\_type] ON [data\_object\_type] [data\_object\_name] FROM [user\_or\_group\_name]**
  - REVOKE – Command to revoke permission
  - privilege\_type – It includes type of privileges like SELECT, VIEW or ALL PRIVILEGES
  - data\_object\_type – It can be one of TABLE, SCHEMA or DATABASE, CATALOG, FUNCTION etc.
  - data\_object\_name – The name of the database, table, catalog etc.
  - user\_or\_group\_name – Name of the user or group from which the privileges need to be revoked.
- **To view grants on a data object (table, database, view etc.) you need to be one of the following:**
  - 1. Databricks admin or
  - 2. Owner of the data object or
  - 3. The user for which the grants are being viewed.
  - Syntax for viewing the grants for a user on a specific database is:  
SHOW GRANTS `user\_name` ON DATABASE database\_name
  - Also note the usage of backticks (``) instead of inverted commas("").

### [Transferring ownership of a data object](#)

**TBLPROPERTIES** allow you to set key-value pairs

### [Table properties and table options \(Databricks SQL\) | Databricks on AWS](#)

recap of some of the things we've learned so far:

1. The Databricks workspace contains a suite of tools to simplify the data engineering development lifecycle
2. Databricks notebooks allow users to mix SQL with other programming languages to define ETL workloads
3. Delta Lake provides ACID compliant transactions and makes incremental data processing easy in the Lakehouse
4. Delta Live Tables extends the SQL syntax to support many design patterns in the Lakehouse, and simplifies infrastructure deployment
5. Multi-task jobs allows for full task orchestration, adding dependencies while scheduling a mix of notebooks and DLT pipelines
6. Databricks SQL allows users to edit and execute SQL queries, build visualizations, and define dashboards
7. Data Explorer simplifies managing Table ACLs, making Lakehouse data available to SQL analysts (soon to be expanded greatly by Unity Catalog)

## Dynamic View Functions

Dynamic view function to filter rows

```
1. CREATE VIEW sales_redacted AS
2. SELECT user_id, country, product, total
3. FROM sales_raw
4. WHERE CASE WHEN is_member('managers') THEN TRUE ELSE total <= 1000000 END;
```

Dynamic view function to hide a column

```
5. CREATE VIEW sales_redacted AS
6. SELECT user_id,
7.    CASE WHEN is_member('auditors') THEN email ELSE 'REDACTED' END AS email,
8.    country,
9.    product,
10.   total
11. FROM sales_raw
12.
```

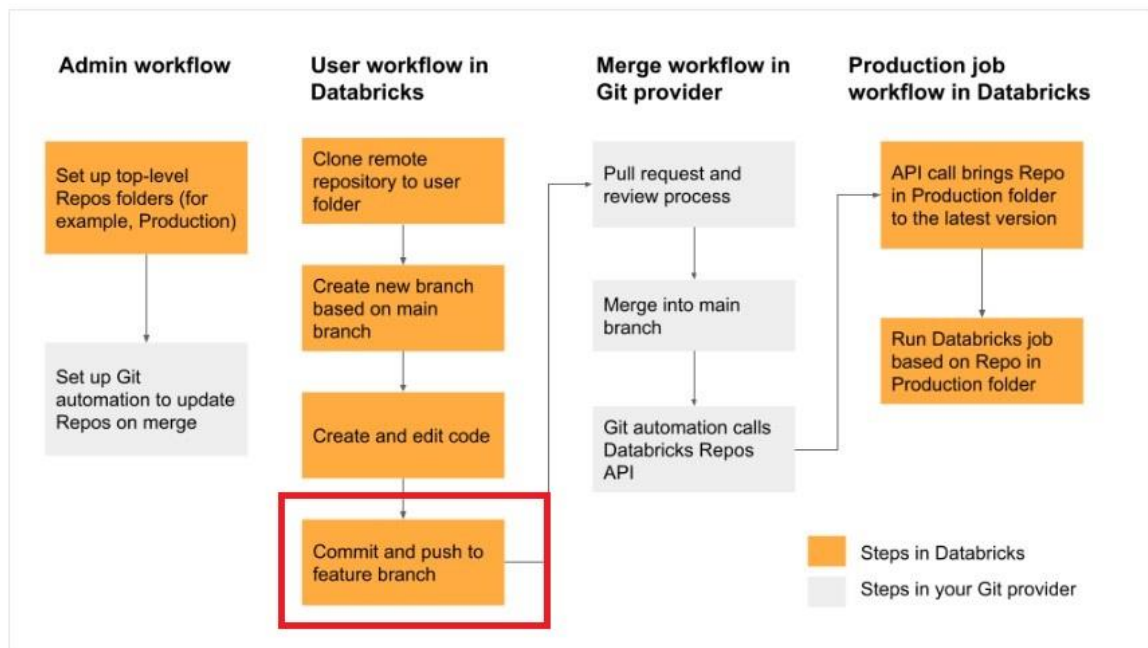
Please review below for more details

<https://docs.microsoft.com/en-us/azure/databricks/security/access-control/table-acls/object-privileges#dynamic-view-functions>

## Databricks Repos

See below diagram to understand the role Databricks Repos and Git provider plays when building a CI/CD workflow.

All the steps highlighted in yellow can be done Databricks Repo, all the steps highlighted in Gray are done in a git provider like Github or Azure Devops



- `git add -> git commit -> git push`
- `git add` adds the files to a **staging area** while `git commit` is used for **recording the changes** in the repository. `git push` is used to push the changes with its logs to a **central repository**.
- Also note, if you need to get a **remote repository** in Databricks, you would need to **clone** the repository first. To get the **latest updates** from a remote repository, you can use `git pull` which makes your local repo in **sync** with the remote repo.
- More Info: [Basic git commands](#)