SOLITA

Databricks Data Engineer Associate

Jeanne Guilbaud

Office :
Leuven (BE)

Year
2024-2025

Contents

1 Data Lakehouse **4**
- 1.1 Databricks Data Intelligence Platform . . . 5
- 1.2 Architecture . . . 5
  - 1.2.1 Open Data Lake . . . 5
  - 1.2.2 Delta Lake . . . 6
  - 1.2.3 Unity Catalog . . . 6

2 Databricks Architecture **8**
- 2.1 Workspaces and Accounts . . . 9
- 2.2 Clusters . . . 9
  - 2.2.1 Operating Modes . . . 10
  - 2.2.2 Access Modes . . . 10
  - 2.2.3 Databricks Runtime . . . 11
  - 2.2.4 Policy . . . 11
  - 2.2.5 Terminate and Restart . . . 11
  - 2.2.6 Access Control Lists . . . 12
- 2.3 Notebooks . . . 12
  - 2.3.1 Databricks Utilities . . . 13
- 2.4 Databricks Repos . . . 13

3 Questions (24%) - Databricks Lakehouse Platform **14**

4 ETL With Apache Spark **16**
- 4.1 Data Objects in the Lakehouse . . . 16
- 4.2 Extracting Data . . . 17
  - 4.2.1 Directly from Files . . . 17
- 4.3 Cleaning Data . . . 20
  - 4.3.1 Missing Data . . . 20
  - 4.3.2 Duplicate Rows . . . 21
  - 4.3.3 Data Format and Regex . . . 21
- 4.4 Complex Transformations . . . 21
  - 4.4.1 Work With Nested Data . . . 21
  - 4.4.2 Manipulating Arrays . . . 23
  - 4.4.3 Join Tables . . . 24
  - 4.4.4 Pivot Tables . . . 24
- 4.5 User-Defined Functions (UDFs) . . . 25
  - 4.5.1 Scoping And Permissions . . . 25
  - 4.5.2 Combine with Control Flow . . . 26
  - 4.5.3 Python User-Defined Functions . . . 26
  - 4.5.4 Pandas/Vectorized UDFs . . . 27
  - 4.5.5 Higher Order Functions . . . 27
- 4.6 Questions 29% - ELT With Spark SQL and Python . . . 28

5 Incremental Data Processing **33**
- 5.1 Delta Lake . . . 33
- 5.2 Schema and Tables . . . 33
  - 5.2.1 Managed Tables . . . 34
  - 5.2.2 External Tables . . . 34
- 5.3 Creating Delta Tables . . . 35
  - 5.3.1 Create Table as Select (CTAS) . . . 35

5.3.2 Filtering and Renaming Columns from Existing Tables . . . 35
5.3.3 Declare Schema with Generated Columns . . . 35
5.3.4 Table Constraints . . . 36
5.3.5 Enrich Tables with Additional Options and Metadata . . . 36
5.3.6 Cloning Delta Lake Tables . . . 37
5.4 Loading Data into Delta Tables . . . 38
5.4.1 Complete Overwrite . . . 38
5.4.2 Append Rows . . . 38
5.4.3 Merge Updates . . . 38
5.4.4 Insert-Only Merge for Deduplication . . . 39
5.4.5 Load Incrementally . . . 39
5.5 Version and Optimization . . . 40
5.5.1 Compacting Small Files and Indexing . . . 40
5.5.2 Reviewing Delta Lake Transactions . . . 40
5.5.3 Rollback Versions . . . 41
5.5.4 Cleaning Up Stale Files . . . 41
5.6 Build Data Pipelines with Delta Live Tables . . . 42
5.6.1 Medallion Architecture . . . 42
5.6.2 Delta Live Tables (DLT) . . . 42
5.6.3 Python VS SQL . . . 47
5.6.4 Change Data Capture (CDC) . . . 47
5.6.5 Dependent tables . . . 48
5.6.6 How to create DLT pipeline . . . 48
5.6.7 Operations . . . 49
5.7 Questions 22% - Incremental Data Processing . . . 51

6 Production Pipelines **54**
6.1 Databricks Workflows & Job . . . 54
6.2 Questions 16% - Production Pipelines . . . 56

7 Data Governance **59**
7.1 Unity Catalog . . . 59
7.1.1 Metastore . . . 60
7.1.2 Catalogs . . . 61
7.1.3 Schema . . . 61
7.1.4 Tables, View and Function . . . 61
7.1.5 External Storage . . . 61
7.1.6 Delta Sharing . . . 62
7.2 Architecture Unity Catalog . . . 62
7.3 Query Lifecyle in Unity Catalog Security Model . . . 63
7.4 Compute Resources and Unity Catalog . . . 63
7.5 Roles and Identities . . . 64
7.6 Data Access Control . . . 64
7.6.1 Dynamic Views . . . 66
7.7 Best Practices . . . 67
7.8 Questions 9% - Data Governance . . . 67

1 Data Lakehouse

Databrick's Data Lakehouse provides an open, unified foundation for all the company data. It is an open data management architecture that combines the flexibility, cost-efficiency, and scale of data lakes with the data management and ACID transactions of data warehouses, enabling business intelligence (BI) and machine learning (ML) on all data.

	Data Warehouse	**Data Lake**	**Data lakehouse**
Data format	Closed	Open	Open
Type of Data	Structured data	All types	All types
Data Access	SQL-only, no direct access to file	Open APIs for direct access to files with SQL, R, Python and others	Open APIs for direct access to files with SQL, R, Python and others
Reliability	High quality, reliable data with ACID transactions	Low Quality	High quality, reliable data with ACID transactions
Governance and security	Fine-grained security and governance for row/columnar level for tables	Poor governance as security needs to be applied to files	Fine-grained security and governance for row/columnar level for tables
Scalability	Exponentially more expensive	Scales to hold any amount of data at low cost	Scales to hold any amount of data at low cost
Use case support	Limited to BI,SQL applications and decision support	Limiter to ML	One data architecture for BI, SQL and ML

Here is the architecture of a Lakehouse in Databricks:

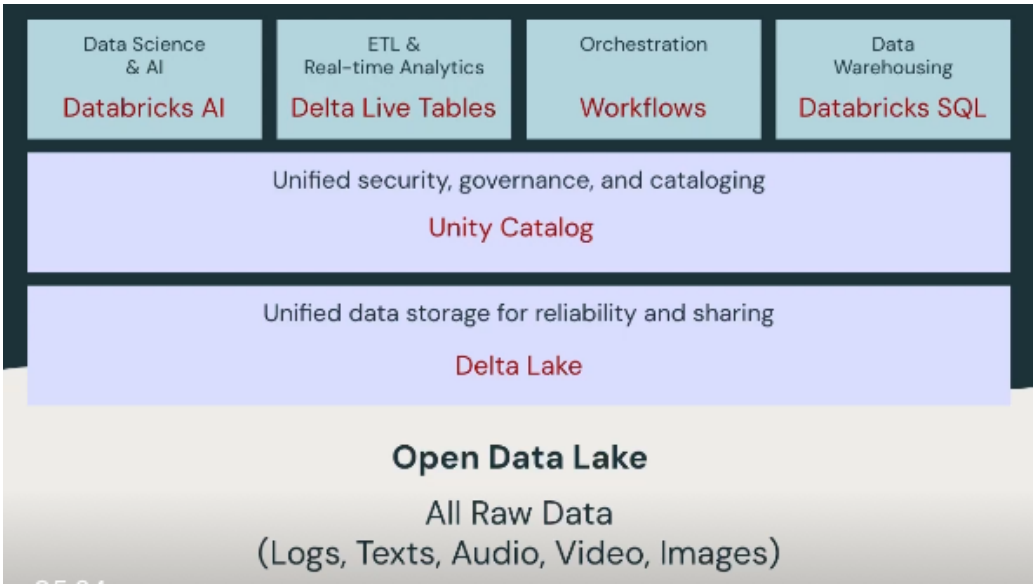


Figure 1.1: Data Lakehouse Architecture

Instead of having data in several data platform, the data lakehouse provides a cloud-provided data lake in an open-source format to store all kind of data (logs, texts, audio, video, images,...). The data are thus more shareable and available to all users. It also ensures data stored in the data lake

is secure, governable and cataloged. The open format promotes competition, allowing the users to test different vendors to see who best serves their needs. It also supports various workloads including data science & AI, ETL & Real-time analytics, Orchestration and Data Warehousing.

1.1 Databricks Data Intelligence Platform

Databricks created the Data Intelligence (DI) Platform that combines the Data Lakehouse and Generative AI, democratizing data and AI across your entire organization. DI Platform are build on the foundation of the lakehouse, but automatically analyze both the data (contents and metadata) and how it is used (queries, reports, lineage,...) to add new capabilities.

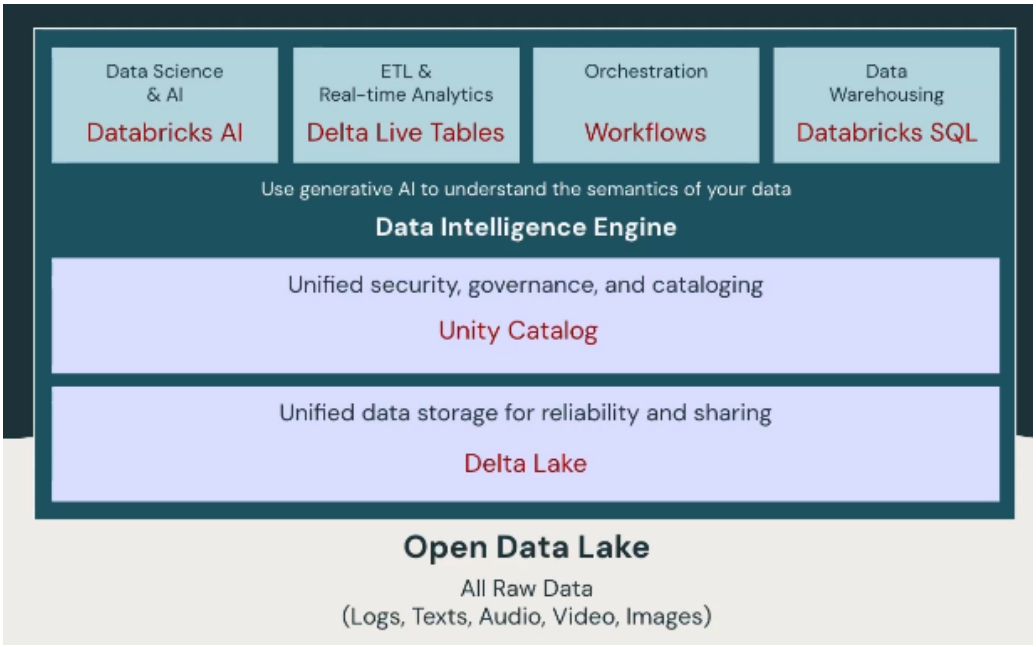


Figure 1.2: Data Intelligence Platform

Data Intelligence Platforms enable:

- **Natural Language Access** you can access specific data in the Unity catalog using simple prompts or you can generate SQL code based on a text prompt in Databricks SQL;
- **Automated Management and Optimization** AI models can optimize data layout, partitioning and indexing based on data usage, reducing the need for manual tuning and knob configuration. The job cost can also be optimized based on past runs;
- **Enhance Governance and Privacy** Delta Live Tables (streaming solution in Databricks) can do automated quality control ie automatically detect, classify and prevent misuse of sensitive data;
- **Databricks AI** allows you to create ML models from scratch, tune it and serve them effectively. You can also create conversational model using RAG. Once the model is ready, you have support for MLOps, AutoML (find the better model), Monitoring and Governance.

1.2 Architecture

As you can see in the figure above, there are several layers in the Data Intelligence Platform.

1.2.1 Open Data Lake

The data Lake allows you to store all type of raw data ie logs, texts, audio, video, images,...

1.2.2 Delta Lake

Delta Lake is an optimized storage layer that provides the foundation for tables in a lakehouse on Databricks. Ii is an open source software that extend Parquet data files with a file-based transaction log that supports ACID transactions, scalable metadata handling, audit history and time travel. All tables on Databricks are **Delta tables** by default.

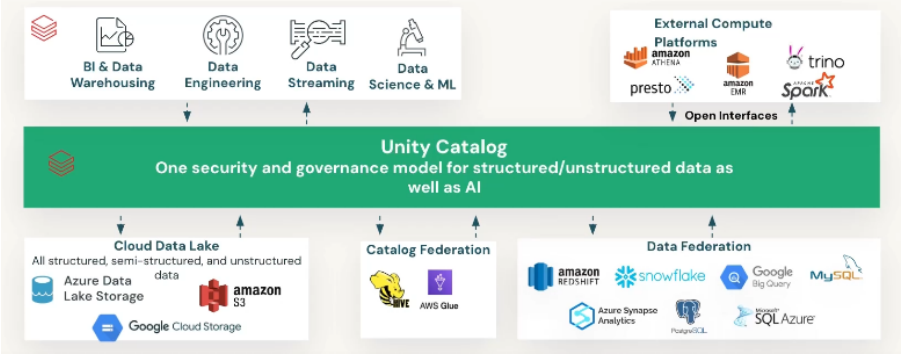
Delta Lake is fully compatible with Apache Spark APIs, and was developed for tight integration with **Structured Streaming**, allowing you to easily use a single copy of data for both **batch** and **streaming** operations and providing incremental processing at scale.

As it is an open-source project, it integrates with all major analytics tools and has a robust partnering solution ecosystem such as Fivetran, AmazonS3, Apache Spark, Snowflake,...

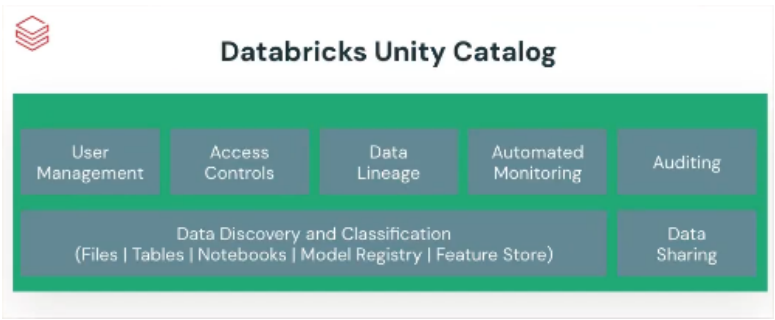
Figure 1.3: Delta Lake integration

1.2.3 Unity Catalog

Unity Catalog is an unified security, governance and cataloging layer where you can manage control, access and organization of the business data.



It provides a context-aware search and automatically describe tables and columns. It also has automated lineage for reproducibility and tracking. It includes user management and access controls to prevent unauthorized access to the ecosystem. It's also the starting point for data sharing and collaboration.



You can:

- Discover, classify and organize structured and unstructured data as well as notebooks, ML models, ML features and arbitrary files at one place;
- Leverage data federation to register and query data from external data sources without ingestion, expanding analysis capabilities;
- Drive better data understanding and faster insights with efficient tag-based search.

2 Databricks Architecture

Databricks operates out of a control plane and a compute plane:

1. The **control plane** includes the back-end services that Databricks manages in your Databricks account. It does not handle data processing directly but manages the resources and operations performed in the Data Plane, such as notebooks, repos,... Key components are:
    - *Databricks UI/Workspace Interface* provides a web-based interface for users to interact with the Databricks env, including notebooks, dashboards and admin settings.
    - *Job Scheduler* manages the scheduling and execution of jobs on clusters in the Data Plane.
    - *Cluster Manager* handles the creation, scaling and termination of clusters according to workload demands.
    - *Identity and Access Management (IAM)* manages user authentication and authorization
    - *API and Integrations* of external applications or services.
2. The **data plane** is where your data is processed by cluster of compute resources. It's hosted within the customer's cloud environment (AWS, Azure or GCP). Key components are:
    - *Databricks Workspace* while its interface is accessed in the Control Plane, the actual compute resources and data storage reside in the customer's cloud account within the Data Plane.
    - *Clusters* that run computations and data processing tasks. They can be a mix of all-purpose and job-specific clusters.
    - *Databricks File Ssytem (DBFS)* a distributed file system layer that sits atop the cloud storage services, facilitating data acess and management for Databricks clusters.
    - *Databricks Runtime* a set of optimized Spark & AI components that run on the clusters for processing tasks.

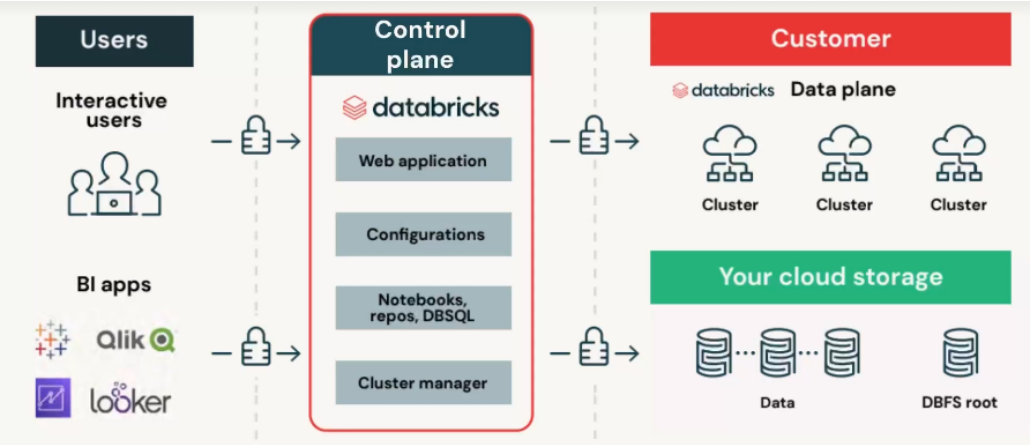


Figure 2.1: Data Governance

For most Databricks computation, the compute resources are in your Cloud account. For **serverless SQL warehouses** or **Model Serving**, the serverless compute resources run in a **serverless compute plane** in your Databricks account.

2.1 Workspaces and Accounts

In Databricks, a **workspace** is a Databricks deployment in the cloud that functions as an environment for your team to access Databricks assets. It organizes objects (notebooks, libraries, dashboards,...) into folders and provides access to data objects and computational resources.

A Databricks **account** represents a single entity that can include multiple workspaces. Accounts enabled for Unity Catalog can be used to manage users and their access to data centrally across all of the workspaces in the account. Billing and support are also handled at the account level.

Databricks bills based on **Databricks Units** (DBUs), units of processing capability per hour based on VM instance type.

2.2 Clusters

A **cluster** is a set of computational resources (VM instances) and configurations on which we can run data engineering, data science and data analytics workloads. A cluster has a **Driver Node** alongside one or more **Worker Nodes** (Databricks provide a single node model as well). The driver distributes the workloads against the different available worker nodes. You can run these workloads as a set of commands in a Notebook or as a job (example of typical applications - Production, ETL/ELT, ML,...). The cluster lives in the Data Plane; within your organization cloud account BUT the cluster management is a function of the Control Plane.

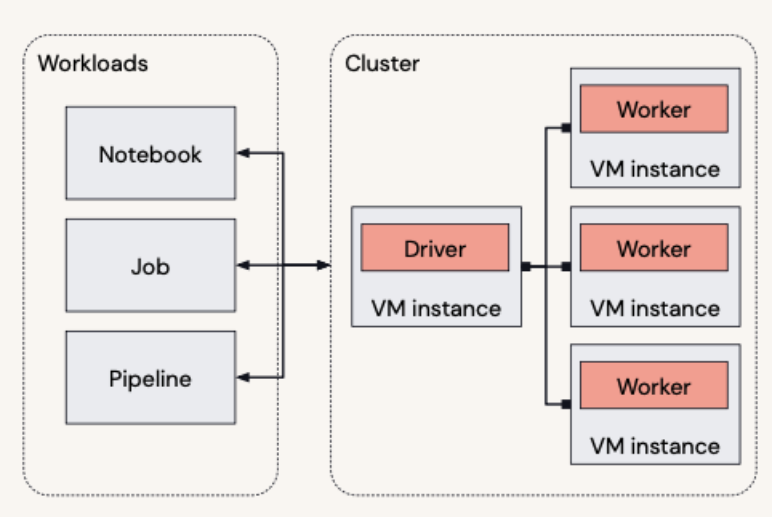


Figure 2.2: Clusters

Databricks provides 2 main types of clusters:

1. **All-Purpose Clusters** are interactive clusters that users can attach notebooks to and run ad-hoc commands. These clusters are designed for exploratory data analysis, data engineering tasks, and collaborative data science work. They remain active until the user decides to terminate them, making them suitable for tasks requiring continuous access to computational resources. Configuration information retained for up to 70 clusters for up to 30 days. Beyond these days, the admin must pin the cluster if he wants to keep it.

2. **Job Clusters** are launched specifically to run automating job or pipelines and are terminated automatically upon the job's completion. These clusters are optimized for running scheduled jobs and batch processes, providing a cost-effective way to process data as they consume resources only for the duration of the job. Configuration information is retained for up to 30 most recently terminated clusters.

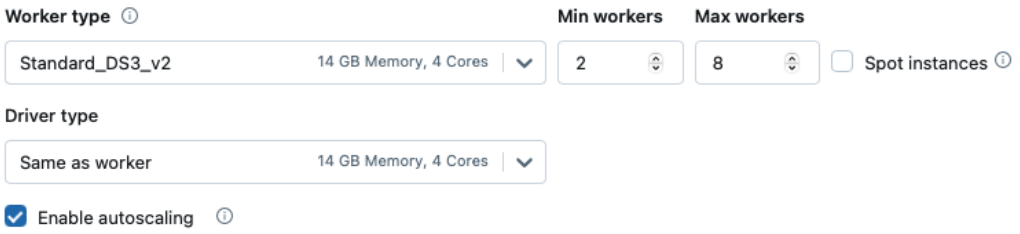
2.2.1 Operating Modes

There are two **operating modes** for the clusters:

1. *Standard (Multi Node)* default mode for workloads developed in any supported language (requires at least two VM instances, one driver and one worker).

2. *Single Node* low-cost single-instance cluster (no worker nodes) catering to single-node machine learning workloads (load and save data with Spark) and lightweight exploratory analysis.

If you have selected the standard mode, you can specify the type of virtual machine that will serve as the driver node, which coordinates tasks among worker nodes. Often it's recommended to have the driver be of the same type or larger than worker nodes. If enabled auto-scaling, your cluster can automatically scale the number of worker nodes up or down within a specified range based on the workload. This helps optimize costs and ensure performance.



2.2.2 Access Modes

We also have the **Access Mode** specify the overall security model of the cluster. We have 4 types:

1. *Single User* cluster with this mode can only be used by a single user. Supports Python, SQL, Scala and R.

2. *Shared* (premium plan feature) can be shared but supports only Python and SQL.

3. *No Isolation Shared*

4. *Custom*

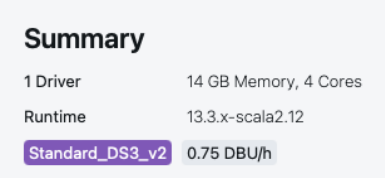
Access mode dropdown	Visible to user	Unity Catalog support	Supported languages
Single user	Always	Yes	Python, SQL, Scala, R
Shared	Always (Premium plan required)	Yes	Python (DBR 11.1+), SQL
No isolation shared	Can be hidden by enforcing user isolation in the admin console or configuring account-level settings	No	Python, SQL, Scala, R
Custom	Only shown for existing clusters *without* access modes (i.e. legacy cluster modes, Standard or High Concurrency); not an option for creating new clusters.	No	Python, SQL, Scala, R

Figure 2.3: Clusters - Access Mode

2.2.3 Databricks Runtime

The software powering Databricks Clusters is the **Databricks Runtime**, a set of core components running on the cluster providing an optimized big data analytics experiences.. Databricks Runtime versions are regularly updated to include the lastest Apache Spark version, performance optimizations, and new features. When creating a cluster, users can select the Databricks Runtime version that best suits their needs, including versions specialized for ML or Genomics eg. You also have an add-on **Photon** to optimize SQL workloads. This flexibility ensures that clusters are always running on an optimized and stable platform.

Below an example of an all-purpose cluster, with 1 node (1 driver) type Standard_DS3_v2 and a Databricks Runtime version 13.3.x-scala2.12 and its associated costs in DBU/h



Best practice = select the more recent version of the runtime. For the Unity Catalog, the minimum required version is 10.1

2.2.4 Policy

A policy is a tool workspace admins can use to limit a user or group's compute creation permissions based on a set of policy rules. Policies provide the following benefits:

- Limit users to creating clusters with prescribed settings.
- Limit users to creating a certain number of clusters.
- Simplify the user interface and enable more users to create their own clusters (by fixing and hiding some values).
- Control cost by limiting per cluster maximum cost (by setting limits on attributes whose values contribute to hourly price).

2.2.5 Terminate and Restart

In the Databricks UI, clusters can be terminated manually or automatically:

1. **Manual** users can terminate their clusters at any time. This action deletes all computations and releases the associated resources. This means:
    - Associated VMs and operational memory will be purged
    - Attached volume storage will be deleted
    - Network connections between nodes will be removed
2. **Automatic** users can set an automatic termination policy for clusters, specifying a period of inactivity after which the cluster will be automatically terminated. When creating the cluster, you can specify a period of inactivity (in minutes) after which the cluster will automatically terminate.

Another option is to **Restart** the cluster. This can be useful if we need to completely clear out the cache on the cluster or wish to completely reset our compute environment.

2.2.6 Access Control Lists

When you have created your cluster, you can create **Access Control Lists** (ACLs) on the *Permissions settings* options. You will have to select the User, Group or service principal and the permissions accored to it. There are 4 type of permissions and you can see below the actions that each permission grant:

- Can Manage
- Can Restart
- Can Attache To
- No Permissions

	No Permissions	Can Attach To	Can Restart	Can Manage
Attach notebook		✔	✔	✔
View Spark UI, cluster metrics, driver logs		✔	✔	✔
Start, restart, terminate			✔	✔
Edit				✔
Attach library				✔
Resize				✔
Change permissions				✔

Figure 2.4: Clusters Access Control Permissions

2.3 Notebooks

Notebooks are the primary means of developing and executing code interactively on Databricks. It provides a cell-by-cell execution of code. You set a default language for the Notebook (that can be change at any time) but you can mix multiple languages (Python, SQL, Scala, R) in a notebook using the **language magic command** % at the start of a cell, allowing the execution of code in languages other than the notebook's default:

```
%python
%sql
```

You can use the magic command %run to run a notebook from another notebook. You specify the notebooks to be run with relative paths. The referenced notebooks 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-01.2
```

There are number of options for downloading either individual notebooks or collections of notebooks. File → Export → choose an option:

- *Source File* the notebook will download to your personal laptop
- *DBC Archive* Databricks Compression file
- *Python Notebook* .ipynb

- *HTML*

If we want to download a collection of notebooks, we choose a root directory → Export → DBC Archive (or Source file or HTML).

2.3.1 Databricks Utilities

Databricks notebooks include a **dbutils** object that provides a number of utility commands for configuring and interacting with the enviromnent.

Utility	Description	Example
`fs`	Manipulates the Databricks filesystem (DBFS) from the console	`dbutils.fs.ls()`
`secrets`	Provides utilities for leveraging secrets within notebooks	`dbutils.secrets.get()`
`notebook`	Utilities for the control flow of a notebook	`dbutils.notebook.run()`
`widgets`	Methods to create and get bound value of input widgets inside notebooks	`dbutils.widget.text()`
`jobs`	Utilities for leveraging jobs features	`dbutils.jobs.taskValues.set()`

Figure 2.5: Databricks Utilities

For example, the following function will list out directories of files from Python cells:

```
path = f"{DA.paths.datasets}"
dbutils.fs.ls(path)
```

When running SQL queries from cells, results will always be displayed in a rendered tabular format. When we have tabular data returned by a Python cell, we can call the **Display()** function to get the same type of preview.

	ID	Course
1	1	Databricks
2	2	Azure
3	3	Big_Data

Figure 2.6: Example display(df)

The display() command has the following capabilities and limitations:

- Preview of results limited to 1000 records
- Provides button to download results data as CSV
- Allows rendering plots

2.4 Databricks Repos

Databricks Git folders is a visual Git client and API in Databricks. It supports commons Git Operations such as cloning a repository, committing and pushing, pulling, branch management, and visual comparison of diffs when committing BUT NOT MERGE. Within Git folders you can

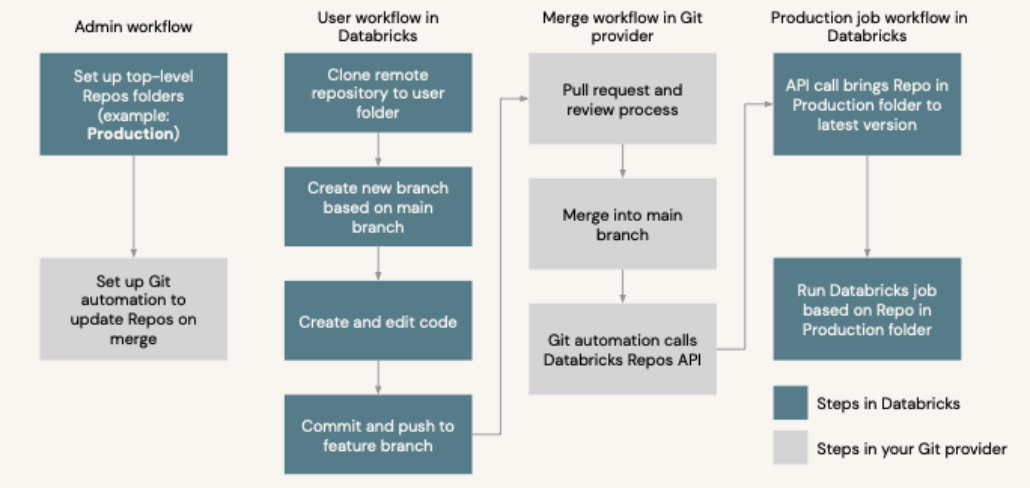


Figure 2.7: Git Integration

develop code in notebooks or other files and follow data science and engineering code dev best practices using Git for version control, collaboration and CI/CD.

It supports the following cloud Git Providers:

- GitHub
- Atlassian BitBucket Cloud
- GitLab
- Microsoft Azure DevOps
- AWS CodeCommit

3 Questions (24%) - Databricks Lakehouse Platform

1. *A data engineer needs to determine whether to use the built-in Databricks Notebooks versioning or version their project using Databricks Repos. Which of the following is an advantage of using Databricks Repos over the Databricks Notebooks versioning?* **Databricks Repos supports the use of multiple branches**.

2. *Which of the following Git operations must be performed outside of Databricks Repos?* **We can do Commit, Pull, Push and Clone in the Repos, but Merge should be done outside Databricks Repos.**

3. *Which of the following datalakehouse features results in improved data quality over a traditional data lake?* **A data lakehouse supports ACID-compliant transactions.**

4. *Which of the following benefits of using the Databricks Lakehouse Platform is provided by Delta Lake?* **They support batch and streaming workloads.**

5. *Which of the following describes the storage organizations of a Delta table?* **Delta tables are stored in a collection of files that contain data, history, metadata and other attributes.**

6. *Which of the following is hosted completely in the Control Pane of the classic Databricks Architecture?* **Databricks web application**

7. *Which of the following describes a scenario in which a data team will want to utilize cluster pools?* **An automated report needs to be refreshed as quickly as possible.**

8. *A data organization leader is upset about the data analysis team's report being different from the data engineering team's reports. The leader believes the siloed nature of their organization's data engineering and data analysis architecture is to blame. Which of the following describes how a data lakehouse could alleviate this issue?* **Both teams would use the same source of truth for their work, leading to consistent and aligned reports.**

9. *A data engineer is running code in a Databricks Repo that is cloned from a central Git repository. A colleague of the data engineer informs them that changes have been made and synced to the central Git repository. The data engineer now needs to sync their Databricks Repo to get the changes from the central Git repository. Which of the following Git operations does the data engineer need to run to accomplish this task?* **Pull**

10. *Which of the following is a benefit of the Databricks Lakehouse Platform embracing open source technologies?* **By avoiding Vendor lock-in**

11. *Which of the following is stored in the Databricks customer's cloud account?* **Data**

12. *Which of the following can be used to simplify and unify siloed data architectures that are specialized for specific use cases?* **Data Lakehouse**

13. *In which of the following file formats is data from Delta Lake tables primarily stored?* **Parquet**

4 ETL With Apache Spark

4.1 Data Objects in the Lakehouse

In the Lakehouse, Databricks combine data stored with the delta lake protocol in cloud object storage with metadata registered in the first object called **Metastore**.

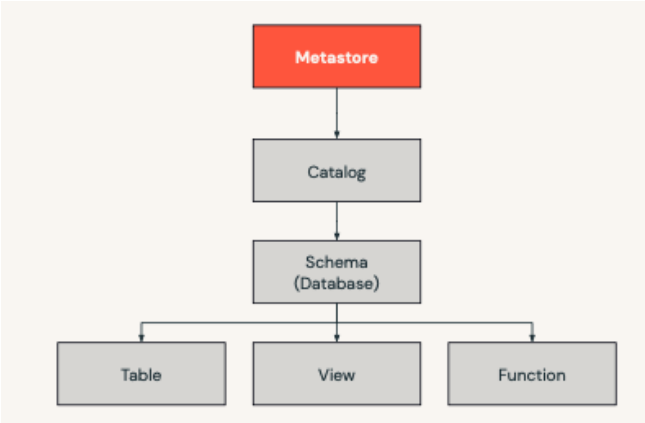


Figure 4.1: Data Objects in the Lakehouse

Under the Metastore, there are 5 primary objects in the Databricks Lakehouse:

1. **Catalog** which is a grouping of databases
2. **Schema (Database)** which is a grouping of objects in a catalog. It contains:
    - **Table** that can be *managed* or *external*. A *Managed Table* is based on a file, stored in the managed storage location, that is configured to the metastore. When you drop a managed table, this will delete the table and all underlying data. An *external table* has its data file stored in a cloud storage location, outside of the managed storage location. When you drop an external table, the underlying data remain.

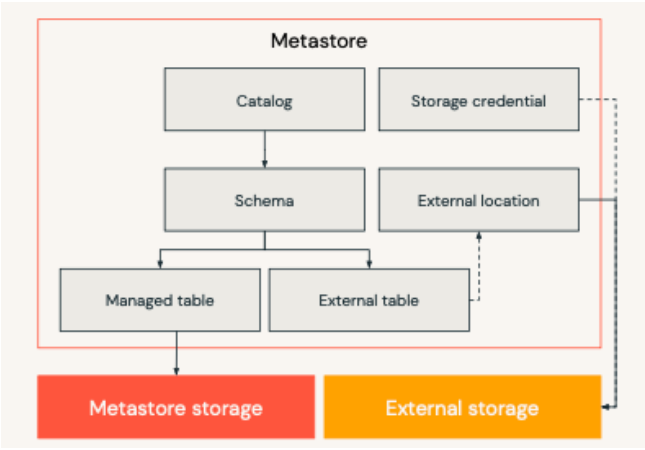


Figure 4.2: Managed vs External Tables

- **View** is a saved query against one or more table/data sources. They can be *temporary* or *global temporary*.
- **Function** is a saved logic that return a scalar value or a set of rows.

4.2 Extracting Data

4.2.1 Directly from Files

You can extract data directly from files using Spark SQL on Databricks. We can use the following pattern:

```
SELECT * FROM file_format.‘path/to/file‘
```

A number of file format support this option but it is most useful for self-describing data formats such as Parquet and JSON:

- The **json.** format will pulls schema from the underlying data.
- The **text.** will load each line of the file as a row with one string column named "value". This can be useful when data sources with text-based files (which includes JSON, CSV,TSV and TXT formats) are prone to corruption. Custom text parsing functions will be used to extract values from text files.

Figure 4.3: .json vs .text

- The **binaryFile.** It will provide file metadata alongside the binary representation of the file contents. Ideal when dealing with images or unstructured data. The fileds created will indicate the path, modificationTime, length and content.

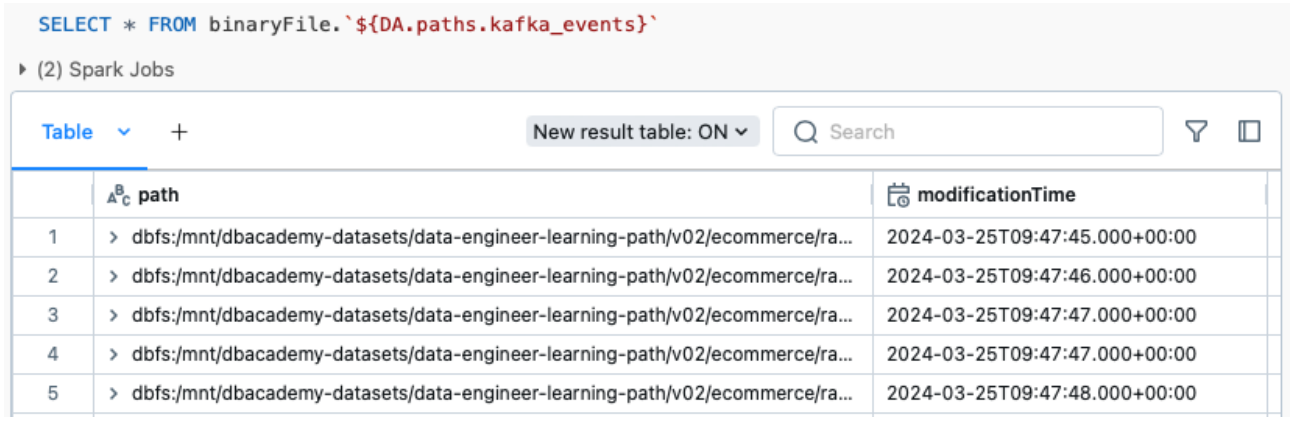


Figure 4.4: .binaryFile

You can also query all of the files in a directory having the same format and schema, by specifying the directory path rather than an individual file.

Views, Temporary Views and CTEs

We can create reference to files by using a view:

```
CREATE OR REPLACE VIEW event_view
AS SELECT * FROM json.‘path/to/directory‘
```

As long as a user has permission to access the view and the underlying storage location, that user will be able to use this view definition to query the underlying data.

Similarly, we can use temporary views to alias queries. They only exist for the current SparkSession ie they are isolated to the current notebook, job or DBSQL query.

```
CREATE OR REPLACE TEMP VIEW event_temp_view
AS SELECT * FROM json.‘path/to/directory‘
```

We can also use **Common Table Expressions (CTEs)** that are short-lived, human-readable reference to the results of a query. It only alias the results of a query while that query is being planned and executed.

```
WITH cte_json
AS (SELECT * FROM json.‘${path/to/directory}‘)
SELECT * FROM cte_json
```

In this case, we can't do a SELECT * on cte_json afterwards, it does not exist anymore. It only exists when the cell is running.

Options for External Sources

Many data sources require **additional configurations or schema declaration** to properly ingest records. For example, using direct query against a CSV file rarely returns the desired results (header as a row, no column, pipe-delimited instead of comma, data truncated,...). The syntax below demonstrates the essentials required to extract data from most formats (there are other configurations if needed):

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

Keep in mind that, similar to when we directly queried our files and created a view, we are still just pointing to files stored in an external location. Let's take the example with USING CSV and options header and delimiter.

```
CREATE TABLE IF NOT EXISTS 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.sales_csv}"
```

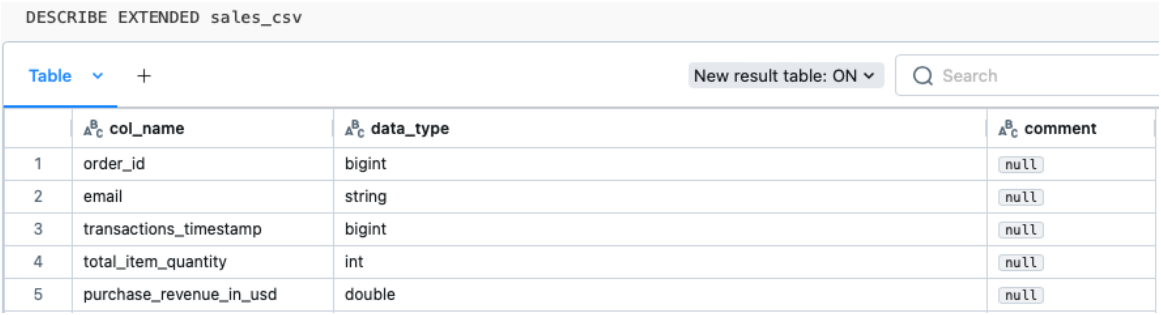
Using PyPark, you can wrap this SQL code with the **spark.sql()** function to use SQL in a Python env as below:

```
%python
spark.sql(f"""
SQL Statement
""")
```

Note taht no data has moved during table declaration. Similar to when we directly queried our files and created a view, we are still just poiting to files stored in an external location.

All the metadata and options passed during table declaration will be persisted to the metastore, ensuring that data in the location will always be read with these options. The "DESCRIBE EXTENDED" code running on a table will show all of the metadata associated with the table definition (which catalog it's in, database or table, location, last access,...):

```
DESCRIBE EXTENDED sales_csv
```



DESCRIBE EXTENDED sales_csv

Table + New result table: ON Search

	col_name	data_type	comment
1	order_id	bigint	null
2	email	string	null
3	transactions_timestamp	bigint	null
4	total_item_quantity	int	null
5	purchase_revenue_in_usd	double	null

Problem of External Data Sources

Whenever we're defining tables or queries against external data sources, we cannot expect the performance guarantees associated with Delta Lake and Lakehouse. For example: while Delta Lake tables will guarantee that you always query the most recent version of your source data, tables registered against other data sources may represent older cached versions.

When we queried a data source, Spark automatically cached the underlying data in local storage. This ensure that on subsequent queries, Spark will provide the optimal performance by just querying this local cache. When using external data source, if we update the files underlying the table, the cache will still point to our previous data, because the external data source is not configured to tell Spark that it should refresh this data (unlike delta table source format). A solution is to manually refresh the cache of our data by running REFRESH TABLE command.

```
REFRESH TABLE sales_csv
```

However this will invalidate our cache, meaning that we'll need to re-scan our original data source and pull all data back into memory. For very large datasets, this may take a significant amount of time.

Extracting Data from SQL Databases

SQL databases are an extremely common data source, and Databricks has a standard JDBC driver for connecting with it. The general syntax is the following, where you add the url, user password,...

```
CREATE TABLE
USING JDBC
OPTIONS (
    url = "jdbc:{databaseServerType}://{jdbcHostname}:{jdbcPort}",
    dbtable = "{jdbcDatabase}.table",
    user = "{jdbcUsername}",
    password = "{jdbcPassword}"
)
```

Warning - the back-end configuration of the JDBC server assumes you are running this notebook on a single-node cluster. If you are running on a cluster with multiple workers, the client running in the executors will not be able to connect to the driver.

Note that some SQL systems such as data warehouses will have custom drivers. Spark will interact with various external db differently, but the two basic approaches are:

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.

In either case, working with very large datasets in external SQL databases can incur significant overhead because of either:

1. Network transfer latency associated with moving all data over the public internet.
2. Execution of query logic in source systems not optimized for big data queries.

4.3 Cleaning Data

As we inspect and clean our data, we'll need to construct various column expressions and queries to express transformations to apply on our dataset.

4.3.1 Missing Data

We can see the count of values in each field of our data, using the function count():

- SELECT COUNT(*) FROM users - will return the total number of records in the dataset "users" (it counts null value)
- SELECT COUNT(user_id) FROM users - will return the number of non-null value in the column "user_id" in the dataset "users" (it does not count null values for a specified column)

If the second operation does not give the same number as the first, it means that there are some null values in the field. There are two ways to count null values in a field, either in SQL or Python:

- In **SQL**:

```
SELECT count_if(email IS NULL) FROM users_dirty;
OR
SELECT count(*) FROM users_dirty WHERE email is NULL;
```

- In **Python**:

```
from pyspark.sql.functions import col
usersDF = spark.read.table("users_dirty")

usersDF.selectExpr("count_if(email is NULL)")
OR
usersDF.where(col("email").isNull()).count()
```

4.3.2 Duplicate Rows

We can use DISTINCT(*) to remove true duplicate records where entire rows contain the same values:

```
%sql
SELECT DISTINCT(*) FROM users_dirty
%python
usersDF.distinct().display()
```

If we want to remove duplicate records based on certain columns values, we can use the GROUP BY method.

4.3.3 Data Format and Regex

The Regex can be used to extract domains in a column. For example, the column "timestamp" has the following format: "1593879630135690" that is non human-readable. We can use the function "date_format()" and a regex to extract information in this format:

```
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
)
```

Or in Python:

```
from pyspark.sql.functions import data_format, regexp_extract

display(dedupedDF
    .withColumn("first_touch",(col("user_first_touch_timestap")/1e6).cast("timestamp"))
    .withColumn("first_touch_date",date_format("first_touch","MMM d, yyyy"))
    .withColumn("first_touch_time",date_format("first_touch","HH:mm:ss"))
    .withColumn("email_domain",regexp_extract("email","(?<=@).+",0))
```

4.4 Complex Transformations

Querying tabular data gets more complicated as the data structure becomes less regular, when many tables needs to be used in a single query, or when the shape of data needs to be changed dramatically. There are number of functions present in Spark SQL to help engineers complete even the most complicated transformations.

4.4.1 Work With Nested Data

Let's imagine we have a table "events_raw" with data that are binary-encoded JSON values. There are represented with a key-value pair, where the value contains the nested data:

key	value
UA000000107384208	{"device":"macOS","ecommerce":{},"event_name":"checkout","event_previous_timestamp":1593880801027797,"event_timestamp":1593880822506642,"geo":{"city":"Traverse City","state":"MI"},"items":[{"item_id":"M_STAN_T","item_name":"Standard Twin Mattress","item_revenue_in_usd":595.0,"price_in_usd":595.0,"quantity":1}],"traffic_source":"google","user_first_touch_timestamp":1593879413256859,"user_id":"UA000000107384208"}
UA000000107388621	{"device":"Windows","ecommerce":{},"event_name":"email_coupon","event_previous_timestamp":1593880770092554,"event_timestamp":1593880829320848,"geo":{"city":"Hickory","state":"NC"},"items":[{"coupon":"NEWBED10","item_id":"M_STAN_F","item_name":"Standard Full Mattress","item_revenue_in_usd":850.5,"price_in_usd":945.0,"quantity":1}],"traffic_source":"direct","user_first_touch_timestamp":1593879889503719,"user_id":"UA000000107388621"}

Spark SQL has built-in functionality to directly interact with nested data stored as JSON string. To access subfields in **JSON strings** you can use the syntax value:col_name

```
%sql
SELECT * FROM events_strings WHERE value:event_name = "finalize" ORDER BY key
    LIMIT 1
%python
display(events_stringsDF
    .where("value:event_name = 'finalize'")
    .orderBy("key")
    .limit(1)
```

You can use the JSON string above to derive the schema using the schema_of_json() function:

```
% sql
SELECT
    schema_of_json('{"device":"Linux","ecommerce":{"purchase_revenue_in_usd":1075.5,"total_it
"event_timestamp":1593879335779563,"geo":{"city":"Houston","state":"TX"},"items":[{"coupon":
    "item_name":"Standard King
    Mattress","item_revenue_in_usd":1075.5,"price_in_usd":1195.0,"quantity":1}],
"traffic_source":"email","user_first_touch_timestamp":1593454417513109,
"user_id":"UA000000106116176"}') AS schema
```

In this **struct type**, you can acces subfields using the syntax value.col_name:

```
SELECT * FROM schema WHERE value.event_name = "finalize"
```

The return schema can serve to create a temporary view much more readable. This can be done using the function from_json() that parses a column containing a JSON string into a struct type using the specified schema:

```
CREATE OR REPLACE TEMP VIEW parsed_events AS SELECT json.* FROM (
SELECT from_json(value, 'STRUCT<device: STRING, ecommerce:
    STRUCT<purchase_revenue_in_usd: DOUBLE, total_item_quantity: BIGINT,
    unique_items: BIGINT>, event_name: STRING,
event_previous_timestamp: BIGINT, event_timestamp: BIGINT, geo: STRUCT<city:
    STRING, state: STRING>, items: ARRAY<STRUCT<coupon: STRING, item_id: STRING,
    item_name: STRING, item_revenue_in_usd: DOUBLE, price_in_usd: DOUBLE,
    quantity: BIGINT>>, traffic_source: STRING, user_first_touch_timestamp:
    BIGINT, user_id: STRING>') AS json
FROM events_strings);
```

This temporary view gives a table with the right schema (columns).

In Python, those two operations can be done by:

```
from pyspark.sql.functions import from_json, schema_of_json

json_string = """
{"device":"Linux","ecommerce":{"purchase_revenue_in_usd":1047.6,"total_item_quantity":2,"uni
    Park","state":"CA"},"items":[{"coupon":"NEWBED10","item_id":"M_STAN_Q","item_name":"Stanc
    Queen
    Mattress","item_revenue_in_usd":940.5,"price_in_usd":1045.0,"quantity":1},{"coupon":"NEWE
    Down
    Pillow","item_revenue_in_usd":107.10000000000001,"price_in_usd":119.0,"quantity":1}],"tra
"""
parsed_eventsDF = (events_stringsDF
```

```
    .select(from_json("value", schema_of_json(json_string)).alias("json"))
    .select("json.*")
)
display(parsed_eventsDF)
```

4.4.2 Manipulating Arrays

Spark SQL has a number of function for manipulating array data:

1. **explode()** separates the elements of an array into multiple rows; this creates a new row for each element;
2. **size()** provides a count for the number of elements in an array for each row;

For example, this code explodes items field (an array of structs) into multiple rows and shows events containing arrays with 3 or more items:

```
%sql
CREATE OR REPLACE TEMP VIEW exploded_events AS
SELECT *,explode(items) AS item
FROM parsed_events;

SELECT * FROM exploded_events WHERE size(items) > 2

%python
from pyspark.sql.functions import explode, size

exploded_eventsDF = (parsed_eventsDF
    .withColumn("item",explode("items"))
)
display(exploded_eventsDF.where(size("items")>2))
```

Array transformations

There are 3 functions used to transform arrays:

- collect_set() collects unique values for a field, including fields within arrays;
- flatten() combines multiple arrays into a single array;
- array_distinct() removes duplicates elements from an array.

For example, this code below creates a table that contains the unique collection of actions (event_history) and the associated items (cart_history) grouped by user_id:

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

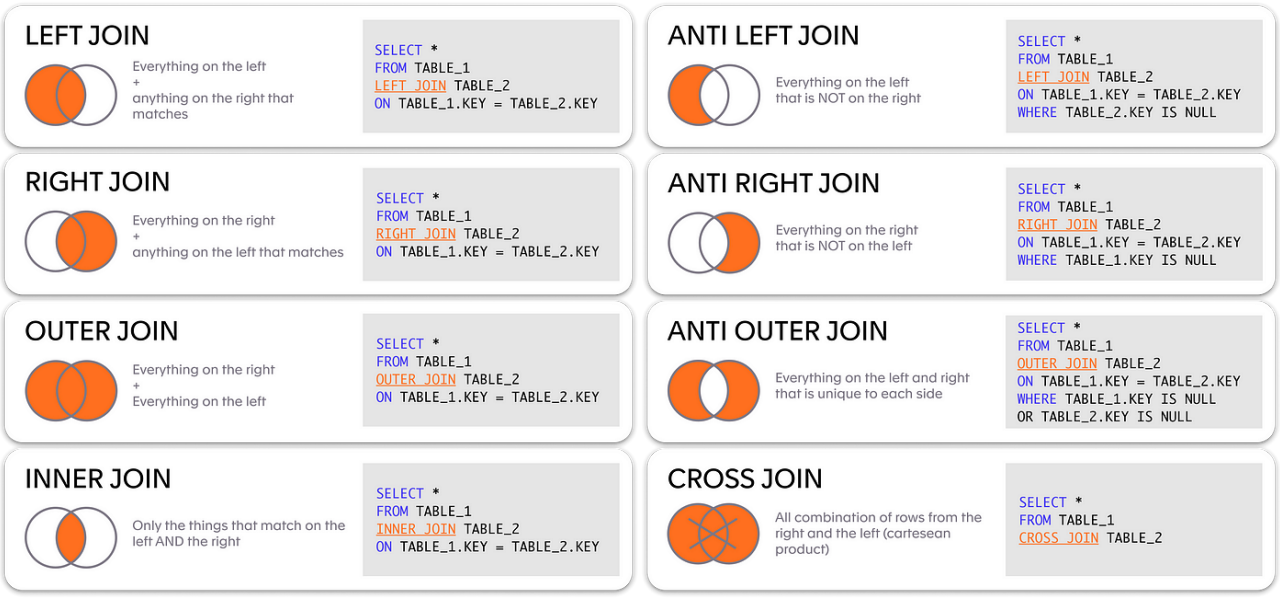
Or in Python

```
from pyspark.sql.functions import array_distinct, collect_set, flatten

display(exploded_eventsDF
    .groupby("user_id")
    .agg(collect_set("event_name").alias("event_history"),
         array_distinct(flatten(collect_set("items.item_id"))).alias("cart_history"))
)
```

4.4.3 Join Tables

Spark SQL supports standard JOIN operations (inner, outer, left, right, anti, cross, semi).

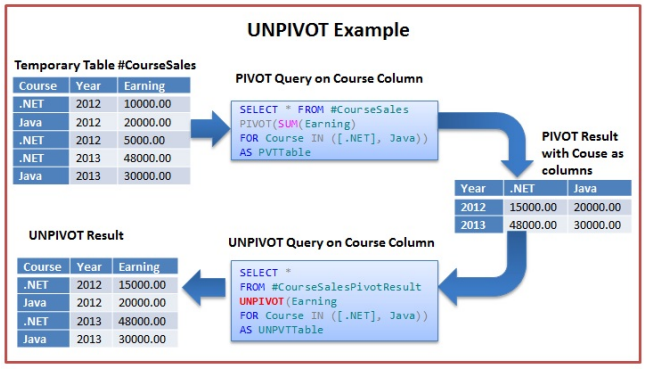


4.4.4 Pivot Tables

We can use PIVOT to view data from different perspectives by rotating unique values in a specified pivot column into multiple columns based on an aggregate function.

1. The PIVOT clause follows the table name or subquery specified in a FROM clause, which is the input for the pivot table.

2. Unique values in the pivot column are grouped and aggregated using the provided aggregate expression, creating a separate column for each unique value in the resulting pivot table.



In Python, the example above for pivot is:

```
PivotDF = (DF
    .groupBy("Course",
    "Year",
    "Earning")
    .pivot("Course")
    .sum("Earning")
```

4.5 User-Defined Functions (UDFs)

User Defined Functions (UDFs) in Spark SQL allow you to register custom SQL logic as functions in a database, making these methods reusable anywhere SQL can be run on Databricks. These functions are registered natively in SQL and maintain all of the optimizations of Spark when applying custom logic to large datasets. At minimum, creating a SQL UDF requires a function name, optional parameters, the type to be returned, and some custom logic.

For example, a simple function named sale_announcement takes an item_name and item_price as parameters. It returns a string that announces a sale for an item at 80% of its original price.

```
CREATE OR REPLACE FUNCTION sale_annoucement(item_name STRING, item_price INT)
RETURNS STRING
RETURN concat("The ", item_name, " is on sale for $", round(item_price*0.8,0));

SELECT *, sale_annoucement(name,prince) AS message FROM item_lookup
```

Note that this function is applied to all values of the column in a parallel fashion within the Spark processing engine. SQL UDFs are an efficient way to define custom logic that is optimized for execution on Databricks.

4.5.1 Scoping And Permissions

There are 4 rules, SQL UDFs:

1. persist between execution environments (which can include notebooks, DBSQL, queries and jobs)
2. exist as objects in the metastore and are governed by the same Table ACLs as databases, tables or views
3. To create a SQL UDF, you need USE CATALOG on the catalog, and USE SCHEMA and CREATE FUNCTION on the schema
4. To use a SQL UDF, you need USE CATALOG on the catalog, USE SCHEMA on the schema and EXECUTE on the function.

We can use DESCRIBE FUNCTION to see where a function was registered and basic information about expected inputs and what is returned (and more with DESCRIBE FUNCTION EXTENDED, such as "Body" which is the SQL sentence used in the function)

```
DESCRIBE FUNCTION EXTENDED sale_announcement
```

4.5.2 Combine with Control Flow

Combining SQL UDFs with control flow in the form of CASE/WHEN clauses provides optimized execution for control flows within SQL workloads. It allows the evaluation of multiple conditional statements with alternative outcomes based on table contents. For example:

```
CREATE OR REPLACE FUNCTION item_preference(name STRING, price INT)
RETURNS STRING
RETURN CASE
  WHEN name = "Standard Queen Mattress" THEN "This is my default mattress"
  WHEN name = "Premium Queen Mattress" THEN "This is my favorite mattress"
  WHEN price > 100 THEN concat("I'd wait until the ", name, " is on sale for $",
      round(price * 0.8, 0))
  ELSE concat("I don't need a ", name)
END;
```

These same basic principles can be used to add custom computations and logic for native execution in Spark SQL.

4.5.3 Python User-Defined Functions

UDF is a custom column transformation function with the following qualities:

- Can't be optimized by Catalyst Optimizer;
- Function is serialized and send to executors;
- Row data is deserialized from Spark's native binary format to pass to the UDF, and the results are serialized back into Spark's native format;
- For Python UDFs, additional interprocess communication overhead between the executor and a Python interpreter running on each worker node.

In Python:

1. You define the function on the driver eg function that gets the first letter of a string :

```
def first_letter_function(email):
    return email[0]
first_letter_function("coucou@bonjour.com") # will return "c"
```

2. You register the function as a UDF. This serializes the function and sends it to executors to be able to transform DataFrame records.

```
first_letter_udf = udf(first_letter_function)
```

3. You apply the UDF on the column that you want

```
from pyspark.sql.functions import col
display(sales_df.select(first_letter_udf(col("email"))))
```

Alternatively you can define and register a UDF using **Python Decorator Syntax**. The @udf decorator parameter is the Column datatype the function returns. There are defined with Python type hints. You will no longer be able to call the local Python function (ie first_letter_udf('email') will not work).

```
@udf("string")
def first_letter_udf(email:str) -> str:
    return email[0]
```

And you can use the decorator UDF here:

```
from pyspark.sql.functions import col
sales_df = spark.table("sales")
display(sales_df.select(first_letter_udf(col("email"))))
```

4.5.4 Pandas/Vectorized UDFs

Pandas UDFs are available in Python to improve the efficiency of UDFs. Pandas UDFs utilize:

- Apache Arrow, an in-memory columnar data format that speed up computation.
- Pandas inside the function, to work with Pandas instances and APIs.

As of Spark 3.0, you should **always** define your Pandas UDF using Python type hints.

```
import pandas as pd
from pyspark.sql.functions import pandas_udf

# We have a string input/output
@pandas_udf("string")
def vectorized_udf(email: pd.Series) -> pd.Series:
    return email.str[0]

display(sales_df.select(vectorized_udf(col("email"))))
```

You can also register these Pandas UDFs to the SQL namespace and uses it from SQL:

```
spark.udf.register("sql_vectorized_udf", vectorized_udf)
%sql
SELECT sql_vectorized_udf(email) AS firstLetter FROM sales
```

4.5.5 Higher Order Functions

- **filters()** 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 the apply a finishing function on the final buffer.

4.6 Questions 29% - ELT With Spark SQL and Python

1. *A data engineer has a Job with multiple tasks that runs nightly. Each of the tasks runs slowly because the clusters take a long time to start. Which of the following actions can the data engineer perform to improve the start up time for the clusters used for the job?*

    - They can use endpoints available in Databricks SQL? No, this will help for performance but not for improving the start up time for the clusters.
    - They can configure the clusters to autoscale for larger data sizes? No, this will help for performance but not for improving the start up time for the clusters.
    - They can configure the clusters to be single node? No
    - They can use jobs clusters instead of all-purpose clusters? No
    - **They can use clusters that are from a cluster pool? Yes because you will have a pool of pre-run clusters, reducing significantly the start up time**.

2. *A single Job runs two notebooks as two separate tasks. A Data engineer has noticed that one of the notebooks is running slowly in the Job's current run. The data engineer asks a tech lead for help in identifying why this might be the case. Which of the following approaches can the tech lead use to identify why the notebook is running slowly as part of the Job?* **They can navigate to the Runs tab in the Jobs UI and click on the active run to review the processing notebook.**

3. *A data engineer has been using a Databricks SQL dashboard to monitor the cleanliness of the input data to an ELT job. The ELT job has its Databricks SQL query that returns the number of input records containing unexpected NULL values. The data engineer wants their entire team to be notified via a messaging webhook whenever this value reaches 100. Which of the following approaches can the data engineer use to notify their entire team via a messaging webhook whenever the number of NULL values reaches 100?* **They can set up an Alert with a new webhook alert destination.**

4. *A data engineer is attempting to drop a Spark SQL table "my_table". The data engineer wants to delete all table metadata and data. They run the following command: DROP TABLE IF EXSITS my_table. While the obkect no longer appears when they run SHOW TABLES, the data files still exist. Which of the following describes why the data files still exist and the metadata files were deleted?* **The table was external.**

5. *A data engineer only wants to execute the final block of a Python program if the Python variable day_of_week is equal to 1 and the Python variable review_period is True. Which of the following control flow statemetns should the data engineer use to begin this conditionally executed code block?*

```
if day_of_week == 1 and review_period:
```

6. *A data engineering team has two tables. The first table march_transactions is a collection of all retail transactions in the month of March. The second table april_transactions is a collection of all retail transactions in the month of April. There are no duplicate records between the tables. Which of the following commands should be run to create a new tabble all_transactions that contains all records from march_transactions and april_transactions without duplicate records?*

```
CREATE TABLE all_transactions AS
SELECT * FROM march_transactions
UNION SELECT * FROM april_transactions;
```

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

```
CREATE TABLE jdbc_customer360
USING ___________
OPTIONS(
    url "jdbc:sqlite:/customers.db"
    dbtable "customer360"
)
```

Which of the following lines of code fills in the above blank to successfully complete the task? **As the url is a jdbc, you need to use the jdbc driver**

```
USING org.apache.spark.sql.jdbc
```

8. *A data engineer runs a statement every day to copy the previous day's sales into the table transactions. Each day's sales are in their own file in the location "/transactions/raw". Today, the data engineer runs the following commands to complete this task:*

```
COPY INTO transactions
FROM "/transactions/raw"
FILEFORMAT = PARQUET;
```

After running the command today, the data engineer notices that the number of records in table transactions has not changed. Which of the following describes why the statement might not have copied any new records into the table? **The previous day's file has already been copied into the table.**

9. *A data analyst has a series of queries in a SQL program. The data analyst wants this program to run every day. They only want the final query in the program to run on Sundays. They ask for help from the data engineering team to complete this task. Which of the following approaches could be used by the data engineering team to complete this task?* **They could wrap the queries using PySpark and use Python's control flow system to determine when to run the final query.**

10. *A data engineer needs to apply custom logic to string column city in table stores for a specific use case. In order to apply this custom logic at scale, the data engineer wants to create a SQL UDFs. Which of the following code blocks creates this SQL UDF?*

```
CREATE FUNCTION combine_nyc(city STRING)
RETURN STRING
RETURN CASE
    WHEN city = "brooklyn" then "new york"
    ELSE city
END;
```

11. *Which of the following commands can be used to write data into a Delta table while avoiding the writing of duplicate records?* **MERGE**

12. *Which of the following benefits is provided by the array functions from Spark SQL?* **An ability to work with complex, nested data ingested from JSON files.**

13. *Which of the following commands will return the location of database customer360?*

```
DESCRIBE DATABASE customer360
```

14. *A data analyst has created a Delta table sales that is used by the entire data analysis team. They want help from the data engineering team to implement a series of tests to ensure the data is clean. However, the data engineering team uses Python for its tests rather than SQL. Which of the following commands could the data engineering team use to access sales in PySpark?*

```
spark.table("sales")
```

15. *A data engineer has realized that they have made a mistake when making a daily update to a table. They need to use Delta time travel to restore the table to a version that is 3 days old. However, when the data engineer attemps to time travel to the older version, they are unable to restore the data because the data files have been deleted. Which of the following explains why the data files are no longer present?* **The VACCUM command was run on the table.**

16. *Which of the following code blocks will remove the rows where the value in column age is greater than 25 from the existing Delta table my_table and save the updated table?*

```
DELETE FROM my_table WHERE age > 25;
```

17. *A data engineer has been given a new record of data:*

```
id String = 'a1'
rank INTEGER = 6
rating FLOAT = 9.4
```

Which of the following SQL commands can be used to append the new record to an existing Delta table my_table?

```
INSERT INTO my_table VALUES ('a1',6,9.4)
```

18. *A data engineer has realized that the data files associated with a Delta table are incredibly small. They want to compact the small files to form larger files to improve performance. Which of the following keywords can be used to compact the small files?* **OPTIMIZE**

19. *A data architect has determined that a table of the following format is necessary Which of the*

EmployeeID	startDate	avgRating
a1	2009-01-06	5.5
a2	2018-11-21	7.1
...	...	...

following code blocks uses SQL DDL commands to create an empty Delta table in the above format regardless of whether a table already exists with this name?

```
CREATE OR REPLACE TABLE table_name(
    employeeID STRING,
    startDate DATE,
    avgRating FLOAT
)
```

CREATE TABLE IF NOT EXISTS wouldn't work because if the table already exists, it will not create our particular table with the wanted format.

20. *A data engineer has a Python notebook in Databricks, but they need to use SQL to accomplish a specific task within a cell. They still want all of the other cells to use Python without making*

any changes to those cells. Which of the following describes how the data engineer can use SQL within a cell of their Python Notebook? **They can add %sql to the first line of the cell.**

21. *Which of the following SQL keywords can be used to convert a table from a long format to a wide format?* **PIVOT**

22. *Which of the following describes a benefit of creating an external table from Parquet rather than CSV when using CREATE TABLE AS SELECT statement?* **Parquet files have a well-defined schema.**

23. *A data engineer wants to create a relational object by pulling data from two tables. The relational object does not need to be used by other data engineers in other sessions. In order to save on storage costs, the data engineer wants to avoid copying and storing physical data. Which of the following relational objects should the data engineer create?* **Temporary View because do not need to be used by other data engineers in other sessions.**

24. *A data analyst has developed a query that runs against Delta table. They want help from the data engineering team to implement a series of tests to ensure the data returned by the query is clean. However, the data engineering team uses Python for its tests rather than SQL. Which of the following operations could the data engineering team use to run the query and operate with the results in PySpark?*

```
spark.sql()
```

spark.table() can be use to load a table, not run a query

25. *Which of the following commands will return the number of null values in the member_id column?*

```
SELECT count_if(member_id IS NULL) FROM my_table;
```

26. *A data engineer needs to apply custom logic to identify employees with more than 5 years of experience in an array column employees in table stores. The custom logic should create a new column exp_employees that is an array of all the employees with more than 5y of experience for each row. In order to apply this custom logic at scale, the data engineer wants to use the FILTER high-order function. Which of the following code blocks successfully completes this task?*

```
SELECT
    store_id,
    employees,
    FILTER (employees, i -> i.years_exp > 5 ) AS exp_employees
FROM stores;
```

27. *A data engineer has a Python variable table_name that they would like to use in a SQL query. They want to construct a Python code block that will run the query using table_name. They have the following incomplete code block:*

```
____(f"SELECT customer_id, spend FROM {table_name}")
```

Which of the following can be used to fill in the blank to successfully complete the task?

```
spark.sql(f"SELECT customer_id, spend FROM {table_name}")
```

28. *A data engineer has created a new database using the following command: CREATE DATABASE IF NOT EXISTS customer360. In which of the following locations will the customer360 database be located?* **If you are not specifying the location in the query, the database will be located in dbfs:/user/hive/warehouse (default location of any object created)**

29. *A data engineer is attempting to drop a Spark SQL table my_table and runs the following command: DROP TABLE IF EXISTS my_table. After running this command, the engineer notices that the data files and metadata files have been deleted from the file system. Which of the following describes why all of these files were deleted?* **The table was managed.**

30. *A data engineer that is new to using Python needs to create a Python function to add two integers together and return the sum. Which of the following code blocks can the data engineer use to complete this task?*

```
def add_integers(x,y):
return x+y
```

31. *In which of the following scenarios should a data engineer use the MERGE INTO command instead of the INSERT INTO command?* **When the target table cannot contain duplicate records.**

32. *A data engineer is working with two tables. Each of these tables is displayed below in its entirety*

sales

customer_id	spend	units
a1	28.94	7
a3	874.12	23
a4	8.99	1

favorite_stores

customer_id	store_id
a1	s1
a2	s1
a4	s2

The data engineer runs the following query to join these tables together:

```
SELECT
    sales.customer_id,
    sales.spend,
    favorite_stores.store_id
FROM sales
LEFT JOIN favorite_stores
ON sales.customer_id = favorite_stores.customer_id;
```

Which of the following will be returned by the above query? **You have all rows of sales, and add store_id for rows concerned. In terms of columns, you don't take the col "units"**

customer_id	spend	store_id
a1	28.94	s1
a3	874.12	null
a4	8.99	s2

33. *A data engineer needs to create a table in Databricks using data from a CSV file at location /path/to/csv. They run the following command:*

```
CREATE TABLE new_table
________
OPTIONS (
    header = "true",
    delimiter = "|"
)
LOCATION "path/to/csv"
```

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

```
USING CSV
```

5 Incremental Data Processing

5.1 Delta Lake

Delta Lake is an open-source project that enables building a data lakehouse on top of existing cloud storage. It's built upon standard data formats and optimized for cloud object storage. It decomposes compute and storage costs. It's built for scalable metadata handling. It is NOT a storage format, a storage medium, a database service or a data warehouse. It brings ACID to object storage:

1. **Atomicity** means all transactions either succeed or fail completely;
2. **Consistency** guarantees relate to how a given state of the data is observed by simultaneous operations;
3. **Isolation** refers to how simultaneous operations conflict with one another. The isolation guarantees that Delta Lake provides do differ from other systems;
4. **Durability** means that committed changes are permanent.

Delta Lake is the default format for tables created in Databricks, so you don't need to indicate the format during the creation.

5.2 Schema and Tables

We can create a schema (also called database) either at the default location:

dbfs:/user/hive/warehouse/...

or using a custom location:

```
CREATE SCHEMA IF NOT EXISTS ${da.schema_name}_default_location;

CREATE SCHEMA IF NOT EXISTS ${da.schema_name}_custom_location LOCATION
  '${da.paths.working_dir}/$(da.schema_name)_custom_location.db'
```

And then we use it:

```
USE ${da.schema_name}_default_location;
```

5.2.1 Managed Tables

Once the schema is created, we can create a **managed table** in it (by not specifying a path for the location) and insert data. Since there are no data from which to infer the table's columns and data types, we must specify it:

```
CREATE OR REPLACE TABLE managed_table (width INT, length INT, height INT);
INSERT INTO managed_table
VALUES (3,2,1);
SELECT * FROM managed_table
```

We can look at the **extended table description** by using DESCRIBE DETAIL:

```
DESCRIBE DETAIL managed_table;
```

By default, **managed tables** in a schema without the location specified will be stored in the location :

dbfs:/user/hive/warehouse/< *schema_name* >.db/

By dropping the table, the table's directory and its log and data files are deleted. Only the schema (database) directory remains.

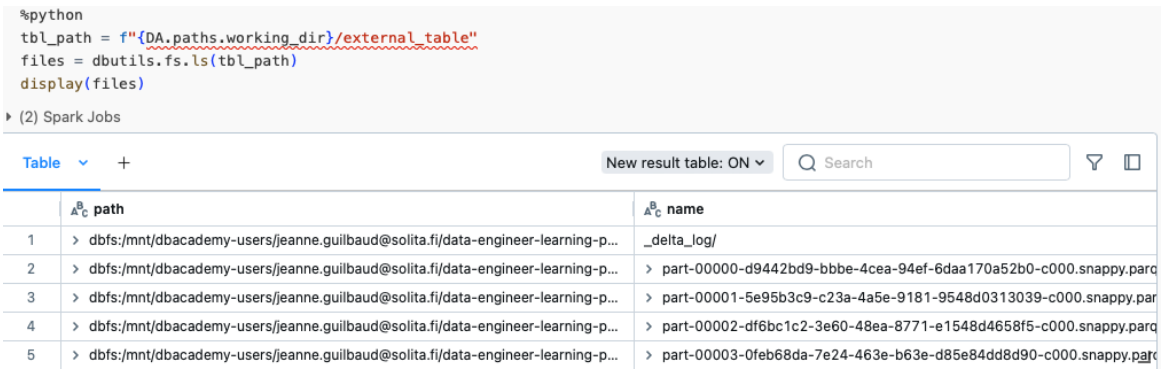
```
DROP TABLE managed_table;
```

5.2.2 External Tables

The data we are going tu use are in CSV format. First we create a temporary view that contains our external data. After, we can create the external table with a location provided in the directory of our choice.

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

If we drop the external table, the table definition no longer exists in the metastore, but the underlying data remain intact.



5.3 Creating Delta Tables

After extracting data from external data sources, we will load data into the Lakehouse to ensure all the benefits of the Databricks platform can be fully leveraged. Databricks recommend that early tables represent a mostly raw version of the data, and that validation and enrichment occur in later stages. This pattern ensures that even if data doesn't match expectations with regards to data types or column names, no data will be dropped, meaning that programmatic or manual intervention can still salvage data in a partially corrupted or invalid state.

5.3.1 Create Table as Select (CTAS)

The most used pattern to create tables is **CREATE TABLE AS SELECT (CTAS) statements**. It creates and populates Delta tables using data retrieved from an input query.

```
CREATE OR REPLACE TABLE sales AS
SELECT * FROM parquet.`${path/to/files}`;
```

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 (eg ingesting data from csv files would not work well). To correctly ingest this data to a Delta Lake table, we'll need to use a reference to the files that allows us to specify options. For that, we can create an external table or temporary view and then use this as the source for a CTAS statement to successfully register the Delta table.

```
CREATE OR REPLACE TEMP VIEW sales_temp_vw (....)
USING CSV
OPTIONS (
...
);

CREATE TABLE sales_delta AS
   SELECT * FROM sales_temp_vw
```

5.3.2 Filtering and Renaming Columns from Existing Tables

Simple transformations like changing column names or omitting columns from target tables can be easily accomplished during table creation. For example, we can create a new table (or view) containing a subset of columns from the "sales" table, and we are renaming the order_id column with the name "id".

```
CREATE OR REPLACE TABLE purchases AS
SELECT order_id AS id, transaction_timestamp, purchase_revenue_in_usd AS price
FROM sales;
```

5.3.3 Declare Schema with Generated Columns

As CTAS do not support schema declaration, some data type might not be adequate (for example Unix timestamp appears differently). This is a situation where generated columns would be beneficial. **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. For example,

we specify the column names and types and we add a generated column to calculate the exact data (as timestamp):

```
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")
```

Values for the date column will be provided automatically.

If a field that would otherwise be generated is included in an insert to a table, this insert will fail if the value provided does not exactly match the value that would be derived by the logic used to define the generated column. For example, this code will raise an error:

```
INSERT INTO purchase_date VALUES
(1,600000,42.0,"2020-06-18")
```

Because the transaction_timestamp is not in the same format as the original table. Consequently, the calculation of the Date value gives a wrong result that do not match the suppose Data column format.

5.3.4 Table Constraints

In reality, the error above refers to a **CHECK constraint**. Indeed, generated columns are a special implementation of check constraints.

Because Delta Lake enforces schema on write, Databricks can support standard SQL constraint management clauses to ensure the quality and integrity of data added to a table. Databricks currently supports 2 types of constraints:

1. *NOT NULL* constraints
2. *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.

We can add a CHECK constraint to the data column of our table like this:

```
ALTER TABLE purchase ADD CONSTRAINT valid_date CHECK (date > ’2020-01-01’);
```

Table Constraints are shown in the **TBLPROPERTIES** field in the DESCRIBE EXTENDED statement.

5.3.5 Enrich Tables with Additional Options and Metadata

CTAS statement can include a number of additional configurations and metadata.

1. **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.

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

   - *COMMENT* is added to allow for easier discovery of table contents;
   - *LOCATION* if it's specified, it will result in an external (rather than managed) table;
   - *PARTITIONED BY* a data 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.**

```
CREATE OR REPLACE TABLE users_pii
COMMENT "Contains PII"
LOCATION ${path/to/location"}
PARTITIONED BY (touch_date)
AS
    SELECT *,
        cast(cast(... AS DATE) touch_date,
        current_timestamp() updated,
        input_file_name() source_file
    FROM parquet.‘${path/to/second/location}‘
```

The metadata fields added to the table provide useful information to understand when records were inserted and from where. This can be especially helpful if troubleshooting problems in the source data becomes necessary. All of the comments and properties for a given table can be reviewed using DESCRIBE TABLE EXTENDED.

NOTE: Partitioning is shown here primarily to demonstrate syntax and impact. 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 files 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. As a best practice, you should default to non-partitioned tables for most use cases when working with Delta Lake.

5.3.6 Cloning Delta Lake Tables

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

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.

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

   Because all the data files must be copied over, this can take quite a while for large datasets.

2. **SHALLOW CLONE** quickly creates a copy of a table to test out applying changes without the risk of modifying the current table. It clones just copy the Delta transactions logs, meaning that the data doesn't move.

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

In either case, data modifications applied to the cloned version of the table will be tracked and stored separately from the source. Cloning is a great way to set up tables for testing SQL code while still in development.

5.4 Loading Data into Delta Tables

5.4.1 Complete Overwrite

Overwriting a table is much faster than deleting/recreating tables because it doesn't need to list the directory recursiverly or delete any files. It's an atomic operation ie concurrent queries can still read the table while you are deleting the table. If overwriting the table fails, the table will be in its previous state. Spark SQL provides 2 easy methods to accomplish overwrites:

1. **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.`${path/to/file}`
```

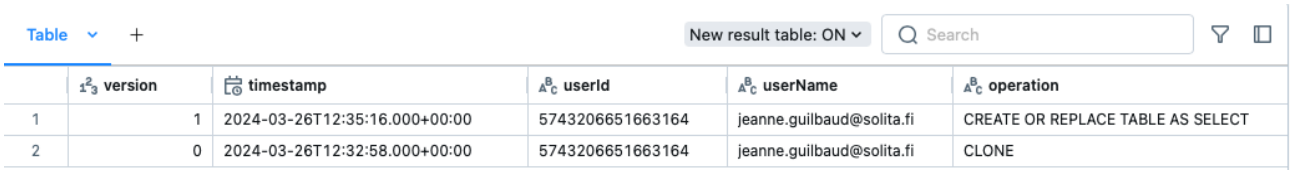
2. **INSERT OVERWRITE** provides a nearly identical outcome as CRAS - data in the target table will be replaced by data from the query. It can only overwrite an existing table, not create a new one like our CRAS statement. It can overwrite only with new records that match the current table schema ("safer" technique), unless we provide optional settings.

```
INSERT OVERWRITE sales
SELECT * FROM parquet.`${path/to/file}`
```

The old version of the table still exists and can easily be retrieve using **Time Travel**. You can review the table history to see the previous version of a table using DESCRIBE HISTORY:

```
DESCRIBE HISTORY sales
```

In this example below, we can see the history for a table overwrite using CRAS statement. The table history also records the operation differently for INSERT OVERWRITE statement.



	version	timestamp	userId	userName	operation
1	1	2024-03-26T12:35:16.000+00:00	5743206651663164	jeanne.guilbaud@solita.fi	CREATE OR REPLACE TABLE AS SELECT
2	0	2024-03-26T12:32:58.000+00:00	5743206651663164	jeanne.guilbaud@solita.fi	CLONE

5.4.2 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.`${path/to/file}`
```

5.4.3 Merge Updates

You can upsert data from a source table, view of DF into a target Delta table using the **MERGE** SQL operation. Delta Lake supports inserts, updates and deletes in MERGE, and support 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 are:

1. updates, inserts, and deletes are completed as a single transaction;
2. multiple conditionals can be added in addition to matching fields;
3. provides extensive options for implementing custom logic.

For example below, we will only update records if the current row has a NULL email and the new row does not. All unmatched records from the new batch will be inserted.

```
MERGE INTO users a
USING users_update b
ON a.user_id = b.user_id
WHEN MATCHED AND a.email IS NULL AND b.email IS NOT NULL THEN$
    UPDATE SET email = b.email, updated = b.updated
WHEN NOT MATCHED THEN INSERT *
```

5.4.4 Insert-Only Merge for Deduplication

A 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 provided a WHEN NOT MATCHED clause.

Below, we use this to confirm that records with the same user_id and event_timestamp aren't already in the events table.

```
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 *
```

5.4.5 Load Incrementally

COPY INTO statement provides SQL engineers to incrementally ingest data from external systems. It's an idempotent option ie files in the source location that have already been loaded are skipped. This operation is potentially much cheaper than full table scans for data that grows predictably.

```
COPY INTO sales
FROM "${path/to/external/source}"
FILEFORMAT = PARQUET
```

This operation does have some expectations:

1. Data schema should be consistent;

2. Duplicate records should try to be excluded or handled downstream.

5.5 Version and Optimization

We can see that a Delta Lake table is backed by a collection of files stored in cloud object storage. The directory contains:

1. a number of Parquet data files that represent records in Delta Lake table
2. A directory named *_delta_log* that contains transactions of the Delta Lake table. If we look in it:

	path	name
1	dbfs:/mnt/dbacademy-users/jeanne.guilbaud@solita.fi/data-engineer-learning-p...	00000000000000000000.crc
2	dbfs:/mnt/dbacademy-users/jeanne.guilbaud@solita.fi/data-engineer-learning-p...	00000000000000000000.json
3	dbfs:/mnt/dbacademy-users/jeanne.guilbaud@solita.fi/data-engineer-learning-p...	00000000000000000001.00000000000000000006.compacted.json
4	dbfs:/mnt/dbacademy-users/jeanne.guilbaud@solita.fi/data-engineer-learning-p...	00000000000000000001.crc
5	dbfs:/mnt/dbacademy-users/jeanne.guilbaud@solita.fi/data-engineer-learning-p...	00000000000000000001.json
6	dbfs:/mnt/dbacademy-users/jeanne.guilbaud@solita.fi/data-engineer-learning-p...	00000000000000000002.crc
7	dbfs:/mnt/dbacademy-users/jeanne.guilbaud@solita.fi/data-engineer-learning-p...	00000000000000000002.json
8	dbfs:/mnt/dbacademy-users/jeanne.guilbaud@solita.fi/data-engineer-learning-p...	00000000000000000003.crc

Each transaction results in a new JSON file being written to the Delta Lake transaction log. Here, we can see that there are 8 total transactions against this table (Delta Lake is 0 indexed).

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. When we query a Delta Lake table, the query engine uses the transaction logs to resolve all the files that are valid in the current version, and ignores all other data files.

5.5.1 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. It 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**. It speeds up data retrieval when filtering on provided fields by colocating data with similar values within data files.

```
OPTIMIZE students
ZORDER BY id
```

5.5.2 Reviewing Delta Lake Transactions

For each action taken on our table, there is a new version stored in the transaction log. These provide us with the ability to query previous versions of our table, by specifying either the integer version or a timestamp (both can be found in DESCRIBE HISTORY table).

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

We are not recreating a previous state of the table by undoing transactions against our current version; rather, we're just querying all those data files that were indicated as valid as of the specified version.

5.5.3 Rollback Versions

If you accidentally execute a DELETE FROM table, removing the entire directory of data, you can simply rollback this commit:

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

Keep in mind that a RESTORE command is recorded as a transaction (describe history); 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 at a desired state.

5.5.4 Cleaning Up Stale Files

Databricks will automatically clean up stale log files (> 30 days by default) in Delta Lake tables. Each time a checkpoint is written, Databricks automatically cleans up log entries older than this retention interval. 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.

```
VACUUM students RETAIN 168 HOURS
```

By default, **VACUUM will prevent you from deleting files less than 7 days old**, just to ensure that no long-running operations are still referencing any of the files to be deleted. If you run VACUUM on a Delta table, you lose the ability time travel back to a version older than the specified data retention period.

You can allow of VACUUM < 7 days it by putting the spark.databricks.delta.retentionDurationCheck.enabled to false (overrides the retention threshold check) and by enabling the logging of VACUUM commands (vacuum op is recorded in the transaction log). If you wish to first print out all records to be deleted, you can use the DRY RUN version of vacuum:

```
SET spark.databricks.delta.retentionDurationCheck.enabled = false;
SET spark.databricks.delta.vacuum.logging.enabled = true;

VACUUM students RETAIN 0 HOURS DRY RUN
```

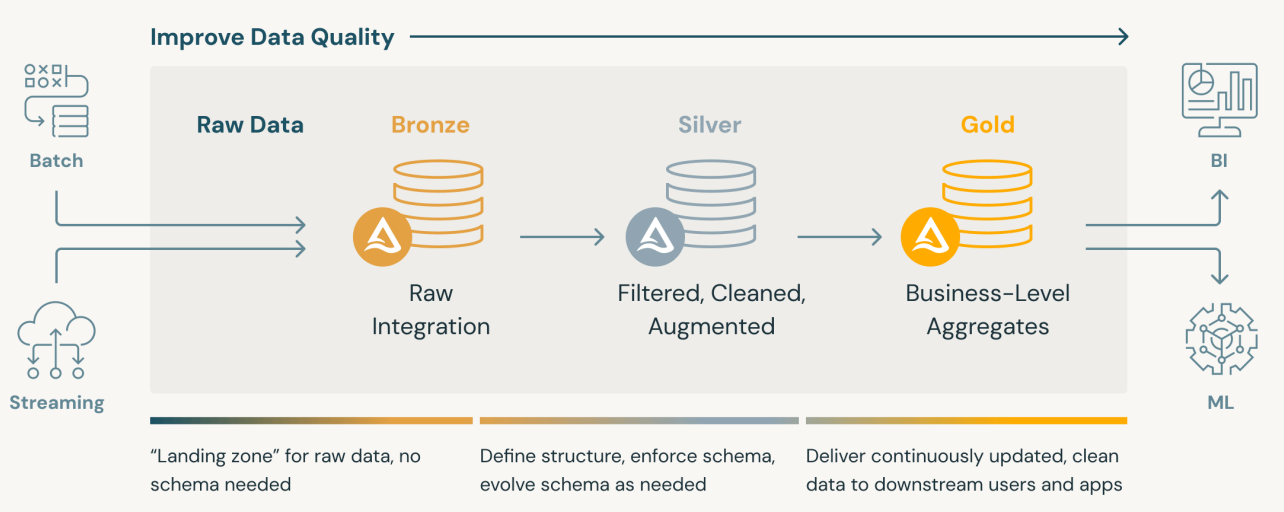
Don't forget to turn the retention duration check back on. When you are done checking if the files to be deleted are ok, you can run VACUUM and delete these files. We will permanently remove access to versions of the table that require these files to materialize.

```
VACUUM students RETAIN 0 HOURS
SET spark.databricks.delta.retentionDurationCheck.enabled = true;
```

5.6 Build Data Pipelines with Delta Live Tables

5.6.1 Medallion Architecture

Medallion Architecture, also referred to as "multi-hop" architectures, is a data design pattern used to logically organize data in a lakehouse, with the goal of incrementally and progressively improving the structure and quality of data as it flows through each layer of the architecture.



There are 3 layer tables:

1. **Bronze Layer** contains typically just a raw copy of ingested data. It replaces traditional data lake. It provides efficient storage and querying of full, unprocessed history of data.

2. **Silver Layer** represents a validated enriched version of data with a reduced storage complexity, latency and redundancy. It preserves the grain of original data (without aggregations) but eliminates duplicate records. Data quality checks ensure that corrupt data are quarantined. It optimizes ETL throughput and analytics query performance.

3. **Gold Layer** is a refined views of data, typically with aggregations. It optimizes query performance for business-critical data and powers ML applications, reporting, dashboards and ad-hoc analytics. It reduces strain on production systems.

5.6.2 Delta Live Tables (DLT)

Large scale ETL is in reality much more complex and brittle than the figure above. The development of pieplines is complex because it is hard to build and maintain table dependencies. It's also difficult to switch between batch and stream processing. It is difficult to monitor and enforce data quality and almost impossible to trace data lineage. Finally, there is a poor observability at granular, data level. Therefore, error handling and recovery is laborious.

Databricks provides tools like **Delta Live Tables (DLT)** that allow users to instantly build batch and streaming data pipelines with Bronze, Silver and Gold tables from just a few lines of code. DLT has built-in quality controls and data quality monitoring features that ensure accurating useful data built on top of these pipelines. Finally it scales with reliability alongside your data.

Live Table and Streaming Live Table

There are two distinct types of persistent tables that can be created with DLT:

1. **Live Table** is a materialized view for the lakehouse, they will return the current results of any query with each refresh.

2. **Streaming Live Table** designed for incremental, near-real time data processing. There is a exactly-once processing of input rows so it allow you to reduce costs and latency by avoiding reprocessing of old data. It uses Spark Structured Streaming for ingestion. It only supports reading from "append-only" streaming sources.

For both kind of tables, DLT takes the approach of a slightly modified CTAS statement. The SQL DLT query is:

```
CREATE OR REFRESH [STREAMING] LIVE TABLE table_name
AS select_statement
```

In Python, we add the decorator @dlt.table to any Python function taht returns a Spark DataFrame. As such, the basic form of a DLT table definition will look like this:

```
@dlt.table
def <function-name>():
return (<query>)
```

Streaming Ingestion with Auto Loader

Databricks has developed **Auto Loader** functionality to provide optimized execution for incrementally loading data from *cloud object storage* into Delta Lake. It will automatically detect new files as they land in the source cloud object storage location, incrementally processing new records without the need to perform expensive scans and recomputing results for infinitely growing datasets. For example, we create a streaming live table with all the json data stored in "/data", using the **cloud_files()**, a Structured Streaming method that enables Auto Loader to be used natively with SQL:

```
CREATE STREAMING LIVE TABLE report
AS SELECT sum(profit)
FROM cloud_files("/data","json", map("cloudFiles.inferColumnTypes", "true"))
```

The cloud_files() method takes positional parameters:

1. the source location, which should be cloud-based object storage "/data";

2. the source data format, which is JSON in this case;

3. An arbitrarily sized comma-separated list of optional reader options. In this case, we set cloudFiles.inferColumnTypes to true. If cloudFiles.inferColumnTypes is not specified, fields will have the correct names but will all be stored as STRING type.

Cloud_files() keeps track of which files have been read to avoid duplication and wasted work. It supports both listing and notifications for arbitrary scale. It has a configurable schema inference and schema evolution that help handle data changes over time.

In Python, you must configure the **format("cloudFiles")** setting to use Auto Loader. The query below returns a streaming live table from a source configured with AutoLoader:

```
@dlt.table
def orders_bronze():
    return (
        spark.readStream
            .format("cloudFiles")
            .option("cloudFiles.format", "json")
            .option("cloudFiles.inferColumnTypes", True)
            .load(f"{source}/orders")
            .select(
                F.current_timestamp().alias("processing_time"),
                F.input_file_name().alias("source_file"),
                "*"
            )
    )
```

In addition to passing cloudFiles as the format, we specify:

1. the option cloudFiles.format as json (this indicates the format of the files in the cloud object storage location
2. the option cloudFiles.inferColumnTypes as True (to auto-detect the types of each column)
3. the path of the cloud object storage to the load method
4. A select statement that includes a couple of pyspark.sql.functions to enrich the data alongside all the source fields.

   By default, dlt.table will use the name of the function as the name for the target table.

Validating, Enriching and Transforming Data

DLT adds new functionality for data quality checks and provides a number of options to allow users to enrich the metadata for created tables.

1. **Comments and Table Properties** Table comments can be used to provide useful information to users throughout your organization. Table properties can be used to pass any number of key/value pairs for custom tagging of data. In the example below, we add as a comment a short human-readable description of the table and we set the value silver for the key quality as a table property:

   ```
   % sql
   CREATE OR REFRESH STREAMING LIVE TABLE orders_silver
   COMMENT "Append only orders with valid timestamps"
   TBLPROPERTIES ("quality"="silver")
   AS SELECT ....
   FROM ...

   % python
   @dlt.table(
       comment = "Append only orders with valid timestamps"
       table_properties = {"quality":"silver"})
   def orders_silver():
       return(
       ...
       )
   ```

2. **Data Quality Constraints** by using simple boolean statements to allow quality enforcement checks on data. For example, we declare a constraint named "valid_date", we define the conditional check that the filed "order_timestamp" must contain a value greater than January 1,2012 and we instruct DLT to fail the current transaction if any records violate the constraint. In SQL, it gives:

```
CREATE OR REFRESH STREAMING LIVE TABLE orders_silver
(CONSTRAINT valid_date EXPECT (order_timestamp > "2021-01-01") ON
   VIOLATION FAIL UPDATE)
AS SELECT ....
FROM ....
```

In Python, we uses decorator functions to set data quality constraints. It gives:

```
@dlt.table()
@dlt.expect_or_fail("valid_date",F.col("order_timestamp") > "2021-01-01")
def orders_silver():
    return(
    ...
    )
```

Each constraint can have multiple conditions, and multiple constraints can be set for a single table. In addition to failing the update, constraint violation can also automatically:

- *Drop records*

```
%sql
CONSTRAINT valid_timestamp
EXPECT(timestamp>'2012-01-01')
ON VIOLATION DROP

%python
@dlt.expect_or_drop('valid_timestamp',F.col('timestamp') >
    '2012-01-01')
```

- *Record the number of violations while still processing these invalid records*

3. **DLT Read Methods for Python** dlt module provides two methods to easily configure references to other tables and views in your DLT pipeline. This syntax allows you to reference these datasets by name without any database reference.

   - **read()** is used to read data in a batch manner. It's designed for scenario where your data source is static or when you're performing batch processing on a snapshot of your data.
   - **read_stream()** is designed for streaming data sources. It's used when you're dealing with continuously updating or real-time data. It enables your DLT pipeline to ingest and process this data in real-time.

   For example:

```
@dlt.table(
def orders_silver():
    return (
        dlt.read_stream("orders_bronze")
            .select(
                "processing_time",
                "customer_id",
                "notifications",
                "order_id",
                F.col("order_timestamp").cast("timestamp").alias("order_timestamp")
            )
    )
```

4. **SELECT-FROM statement** the FROM clause has two constructs that specify how data is handled and processed:

   - *LIVE keyword* used in place of the schema name to refer to the target schema configured for the current DLT pipeline
   - *STREAM method* allows users to declare that the data source being queried is a streaming source:

   For example:

```
CREATE OR REFRESH STREAMING LIVE TABLE orders_silver
AS SELECT timestamp(order_timestamp) AS order_timestamp, * EXCEPT
    (order_timestamp, source_file, _rescued_data)
FROM STREAM(LIVE.orders_bronze)
```

   Note that the only syntactic differences between streaming live tables and live tables are the lack of the STREAMING keyword in the create clause and not wrapping the source table in the STREAM() method. The LIVE keyword indicates that you're accessing the "live" version of a table ie the latest data. For example, below we create a simple query that returns a live table of some aggregated data:

```
CREATE OR REFRESH LIVE TABLE orders_by_date
AS SELECT date(order_timestamp) AS order_date, count(*) AS
    total_daily_orders
FROM LIVE.orders_silver
GROUP BY date(order_timestamp)
```

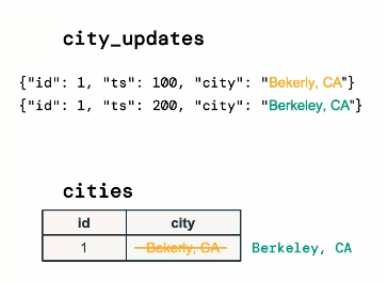
5.6.3 Python VS SQL

Python	SQL	Notes
Python API	Proprietary SQL API	
No syntax check	Has syntax checks	In Python, if you run a DLT notebook cell on its own it will show in error, whereas in SQL it will check if the command is syntactically valid and tell you. In both cases, individual notebook cells are not supposed to be run for DLT pipelines.
A note on imports	None	The dlt module should be explicitly imported into your Python notebook libraries. In SQL, this is not the case.
Tables as DataFrames	Tables as query results	The Python DataFrame API allows for multiple transformations of a dataset by stringing multiple API calls together. Compared to SQL, those same transformations must be saved in temporary tables as they are transformed.
@dlt.table()	SELECT statement	In SQL, the core logic of your query, containing transformations you make to your data, is contained in the SELECT statement. In Python, data transformations are specified when you configure options for @dlt.table().
@dlt.table(comment = "Python comment",table_properties = {"quality": "silver"})	COMMENT "SQL comment" TBLPROPERTIES ("quality" = "silver")	This is how you add comments and table properties in Python vs. SQL

5.6.4 Change Data Capture (CDC)

DLT support a function called **APPLY CHANGES INTO** for **Change Data Capture (CDC)** which allow to maintain an up-to-date replica of a table stored elsewhere. By definition, city_updates has to be a stream source. We need a unique key that can be used to identify a given row (here id). We also need a sequence that can be used to order changes eg log sequence number (lsn), timestamp(ts), ingestion time,... The EXCEPT keyword is here to specify which column to ignore.

```
APPLY CHANGES INTO LIVE.cities
FROM STREAM(LIVE.city_updates)
KEYS(id)
SEQUENCE BY ts
COLUMN * EXCEPT (operation)
```



5.6.5 Dependent tables

In many cases, we will have multiple tables that depend on each other. The solution is to **declare LIVE dependencies** using the LIVE virtual schema:

```
CREATE LIVE TABLE events
AS SELECT ... FROM prod.raw_data

CREATE LIVE TABLE report
AS SELECT ... FROM LIVE.events
```

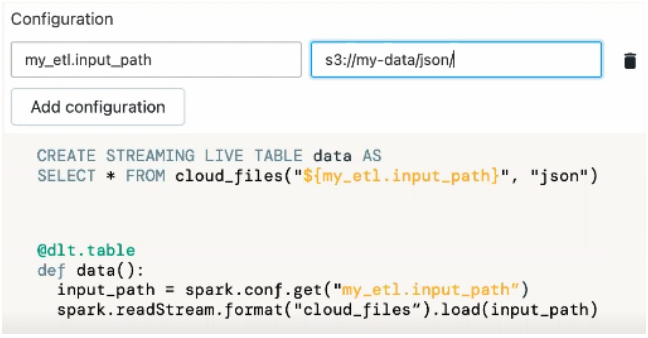
LIVE dependencies, from the same pipeline, are read from the LIVE schema. DLT detects LIVE dependencies and executes all operations in correct order. DLT handles parallelism and captures the lineage of the data. Dependencies owned by other producers are just read from the catalog or spark data source as normal.

5.6.6 How to create DLT pipeline

To create a DLT pipeline, there are 3 steps:

1. Create a cluster to provide the compute resources needed to execute commands and do exploratory data analysis and data engineering.
2. Create a Databricks Notebook with a unique name and default language (Python or Scala) and connect it to the cluster created in step 1.
3. Create a Delta Live Table in Notebooks (but not run because DLT syntax is not intended for interactive execution in a notebook, only as part of a DLT pipeline).
4. Create a pipeline using Workflows → Delta Live Tables → Create Pipeline.
5. Configure the newly created pipeline:
    - **Pipeline mode** specifies how the pipeline will be run. Choose the mode based on latency and cost requirements.
        - *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.
    - **Notebook Libraries** - Use the navigator to select or enter the Notebook path
    - **Storage Location** - optional, allows the user to specify a location to store logs, tables and other information related to the pipeline execution. If not specified, DLT will automatically generate a directory.
    - **Target Schema** If this optional field is not specified, tables will not be registered to a metastore, but will still be available in the DBFS. You can use this schema to access the tables created during the pipeline execution.
    - **Cluster mode, Min Workers, Max Workers** These fields control the worker configuration for the underlying cluster processing the pipeline.

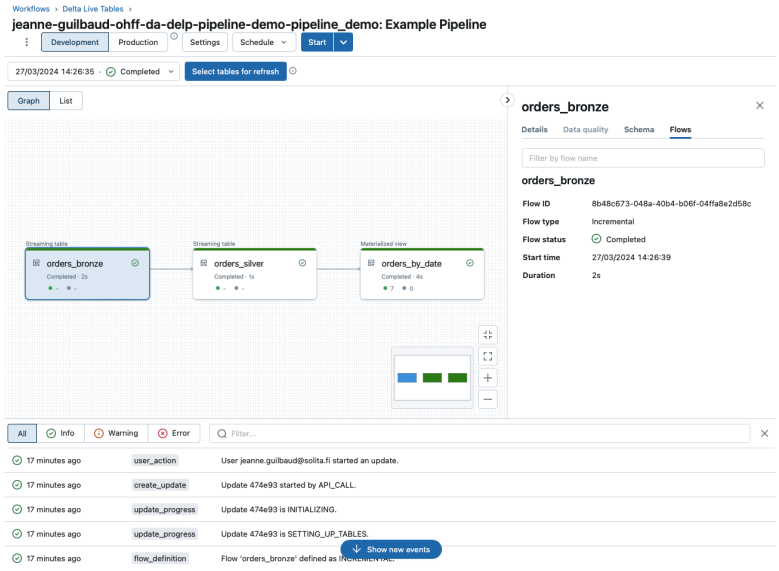
- **Configuration** It's a map of key value pairs that can be used to parametrized your code. Be careful, these keys are cap sensitive.



6. Go to the workflow dashboard and choose the **pipeline mode**
   - *Development* reuses a long-running cluster running for fast iteration. No retries on errors enabling faster debugging.
   - *Production* cuts costs by turning off clusters as soon as they are done (within 5 min). Escalating retries, including cluster restarts, ensure reliability in the face of transient issues.
7. Click start. As the pipeline completes, the execution flow is graphed. You can select the tables to review the details.

5.6.7 Operations

The **Pipelines UI** provides a one stop shop for ETL debugging and operations.



You can:

1. Visualize data flows between tables and realize the lineage of data
2. Discover metadata and quality of each table
3. Access to historical updates
4. Control operations (switch dev-prod, delete pipeline, share pipeline, re-configure pipeline, schedule,...)

5. Dive deep into events (we can see what happens at each stage of the pipeline run)

DLT uses the **event logs** to store much of the important information used to manage report, and understand what's happening during pipeline execution. It is managed as a Delta Lake Table with some of the more important fields stored as nested JSON data. Here is how you can query the Event Log:

```
event_log_path = f"{DA.paths.storage_location}/system/events"

event_log = spark.read.format('delta').load(event_log_path)
event_log.createOrReplaceTempView("event_log_raw")

display(event_log)
```

It includes:

- *Operational Statistics* such as time and current status, pipeline and cluster config, row counts...
- *Provenance* table schemas, definitions and declared properties, table-level lineare, query plans used to update tables
- *Data Quality* expectation pass/failure/drop statistics, input/output rows that caused expectations failures

Perform Audit Loggin

Events related to running pipelines and editing configurations are captured as "user_action".

```
SELECT timestamp, details:user_action:action, details:user_action:user_name
FROM event_log_raw
WHERE event_type = 'user_action'
```

You can thus see all type of action (CREATE, Start, END, EDIT,...) made by the user_name at a specific timestamp.

Query Lineage Information

DLT provides built-in lineage information for how data flows through your table. The query below indicates the direct predecessors for each table. We can combine these information to trace data in any table back to the point it entered the lakehouse and create the lineage:

```
SELECT details:flow_definition.output_dataset,
details:flow_definition.input_datasets
FROM event_log_raw
WHERE event_type = 'flow_definition' AND
      origin.update_id = '${latest_update.id}'
```

Examine Data Quality Metrics

Finally data quality metrics can be extremely useful for both long term and short term insights into your data. If you define expectations on datasets in your pipeline, the data quality metrics are stored in the:

```
details:flow_progress.data_quality.expectations object
```

Events containing information about data quality have the event type "flow_progress".

5.7 Questions 22% - Incremental Data Processing

1. *A data engineer has three tables in a DLT pipeline. They have configured the pipeline to drop invalid records at each table. They notice that some data is being dropped due to quality concerns at some point in the DLT pipeline. They would like to determine at which table in their pipeline the data is being dropped. Which of the following approaches can the data engineer take to identify the table that is dropping the records?* **They can navigate to the DLT pipeline page, click on each table and view the data quality statistics.**

2. *Which of the following Structured Streaming Queries is performing a hop from a Silver table to a Gold table?* **As we are going from a Silver to a Gold, we should have some aggregation function.**

```
spark.table("sales")
    .groupBy("store")
    .agg(sum("sales"))
    .writeStream
    .option("checkpointLocation",checkpointPath)
    .outputMode("complete")
    .table("new_sales")
```

3. *A data engineer is designing a data pipeline. The source system generates files in a shared directory that is also used by other processes. As a result, the files should be kept as is and will accumulate in the directory. The data engineer needs to identify which files are new since the previous run in the pipeline, and set up the pipeline to only ingest those new files with each run. Which of the following tools can the data engineer use to solve his problem?* **Auto Loader as it incrementally and efficiently processes new data files as they arrive in the cloud storage.**

4. *A dataset has been defined using Delta Live Tables and includes an expectations clause: CONSTRAINT valid_timestamp EXPECT (timestap>'2020-01-01') ON VIOLATION DROP ROW. What is expected behavior when a batch of data containing data that violates these constraints is processed?* **Records that violate the expectation are dropped from the target dataset and recorded as invalid in the event log.**

5. *A data engineer has configured a Structured Streaming job to read from a table, manipulate the data, and then perform a streaming write into a new table. The cade block used by the data engineer is below:*

```
spark.table("sales")
    .withColumn("avg_price",col("sales")/col("units))
    .writeStream
    .option("checkpointLocation",checkpointPath)
    .outputMode("complete")
    .--------------
    .table("new_sales")
```

If the data engineer only wants the query to execute a micro-batch to process data every 5 seconds, which of the following lines of code should the data engineer use to fill in the blank? **We need a .trigger() function that contains a "processingTime" parameter that indicates that we are doing micro-batch every x seconds.**

```
.trigger(processingTime="5 seconds")
```

6. *Which of the following tools is used by Auto Loader to process data incrementally?* **Spark Structured Streaming**

7. *Which of the following describes the relationship between Bronze tables and raw data?* **Bronze tables contain raw with a schema applied.**

8. *Which of the following describes the relationship between Gold tables and Silver tables?* **Gold tables are more likely to contain aggregations than Silver tables**.

9. *In order for Structured Streaming to reliably track the exact progress of the processing so that it can handle any kind of failure by restarting and/or reprocessing, which of the following two approaches is useb by Spark to record the offset range of the data being processed in each trigger?* **Checkpointing is one of the option because it helps to reliably track the exact progress on processing. The other option is Write-ahead Logs that records all changes made to a date, allowing structured streaming to recover from failures.**

10. *A DLT pipeline includes two datasets defined using STREAMING LIVE TABLE. Three datasets are defined against DLT sources using LIVE TABLE. The table is configured to run in Production mode using the Continuous Pipeline Mode. Assuming previously unprocessed data exists and all definitions are valid, what is the expected outcome after clicking Start to update the pipeline?* **All datasets will be updated at set intervals until the pipeline is shut down. The compute resources will be deployed for the update and terminated when the pipeline is stopped.**

11. *A data engineer is maintaining a data pipeline. Upon data ingestion, the data engineer notices that the source data is starting to have a lower level of quality. The data engineer would like to automate the process of monitoring the quality level. Which of the following can the data engineer use to solve this problem?* **Delta Live Tables, it provides some tools that can be used to monitor data quality for example data lineage, data quality checks and alerts.**

12. *A data engineer has configured a Structured Streaming job to read from a table, manipulate the data and then perform a streaming write into a new table. The code block used by the data engineer is below:*

```
(spark.readStream
    .table("sales")
    .withColumn("avg_price",col("sales")/col("units"))
    .writeStream
    .option("checkpointLocation",checkpointPath)
    .-----------
    .table("new_sales")
```

If the data engineer only wants the query to process all of the available data in as many batches as required, which of the following lines of code should the data engineer use to fill in the blank?

```
.trigger(availableNow=True)
```

13. *A Delta Live Table pipeline includes two datasets defined using STREAMING LIVE TABLE. Three datasets are defined against Delta Lake table sources using LIVE TABLE. The table is configured to run in Development Mode using the Continuous Pipeline Mode. Assuming previously unprocessed data exists and all definitions are valid, what is the expected outcome after clicking Start to update to the pipeline?* **All datasets will be updated set intervals (because we are using streaming live tables in the pipeline) and until the pipeline is shut down. The compute resources will persist to allow for additional testing.**

14. *A dataset has been defined using DLT and includes an expectations clause: CONSTRAINT valid_timestamp EXPECT (timestamp<'2020-01-01') ON VIOLATION FAIL UPDATE. What is the expected behavior when a batch of data containing data that violates these constraints is processed?* **Records that violate the expectation cause the job to fail.**

6 Production Pipelines

6.1 Databricks Workflows & Job

Databricks Workflows is a fully-managed cloud-based general-purpose task orchestration service for the entire Lakehouse. It is a service for data engineers, data scientists and analysts to build reliable data, analytics and AI workflows on any cloud. As it sits on top of the Databricks Lakehouse platform, it can be fully-integrated with other services.

Workflow Jobs (Workflows) is a way to run your data processing and analysis applications in a Databricks workspace. It can consist of a single **task** or can be a large, multi-task workflow with complex dependencies. Job tasks can run notebooks, JARS, DLT pipelines, or Python, Scala, Spark submit and Java applications. It can also orchestrate **Databricks SQL** queries, alerts and dashboards to create analyses and visualizations. You can also add a task that runs a different job.

	Delta Live Tables	Workflow Jobs
Source	Notebooks only	JARs, notebooks, DLT, application written in Scala, Java, Python
Dependencies	Automatically determined	Manually set
Cluster	Self-provisioned	Self-provisioned or existing
Timeouts and Retries	Not supported	Supported
Import Libraries	Not supported	Supported

Figure 6.1: Differences between DLT and Workflow jobs

Each task is associated to a **cluster**. Depending on the job requirement, you can use a new cluster or an existing one. Workflow supports all-purposes clusters and shared job clusters (use one for more than one tasks).

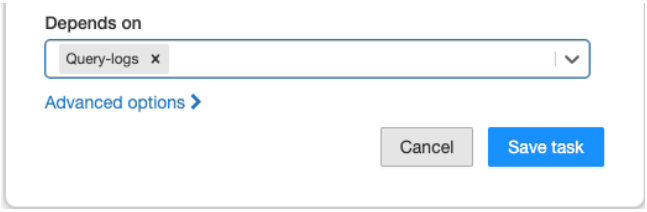
Job **Access Control** enables job owners and administrators to grant fine-grained permissions on their jobs. Job owner can choose which other users or groups can view the job results. Owners can also choose who can manage their job runs (Run Now an Cancel Run permissions). You must have "CAN MANAGE" or "IS OWNER" permission on the job in order to manage permissions on it.

Ability	NO PERMISSIONS	CAN VIEW	CAN MANAGE RUN	IS OWNER	CAN MANAGE
View job details and settings		x	x	x	x
View results		x	x	x	x
View Spark UI, logs of a job run			x	x	x
Run now			x	x	x
Cancel run			x	x	x
Edit job settings				x	x
Delete job				x	x
Modify permissions				x	x

Figure 6.2: Job ACLs

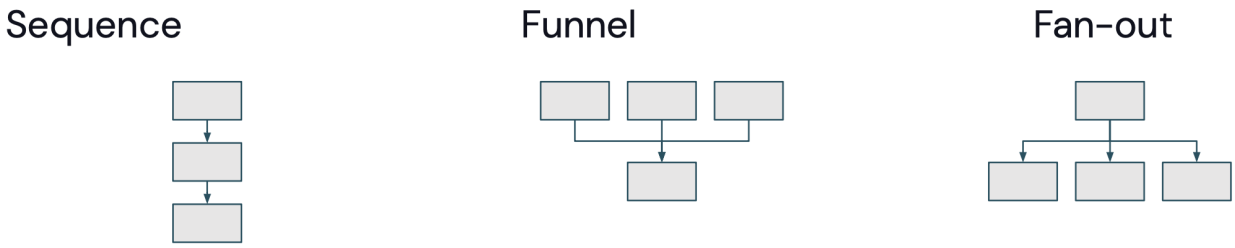
You can configure a **Retry Policy** that determines when and how many times failed task runs are retried. The retry interval is calculated in milliseconds between the start of the failed run and the subsequent retry run.

You can define the order of execution of tasks in a job using the **Depends on** drop-down menu. You can set this field to one or more tasks in the job:



There are 3 major dependencies patterns, that can be combined or use alone:

- **Sequence** all tasks need to be executed sequentially.
- **Funnel** when there are multiple data sources.
- **Fan-out** When there is a single data source and multiple data stores.

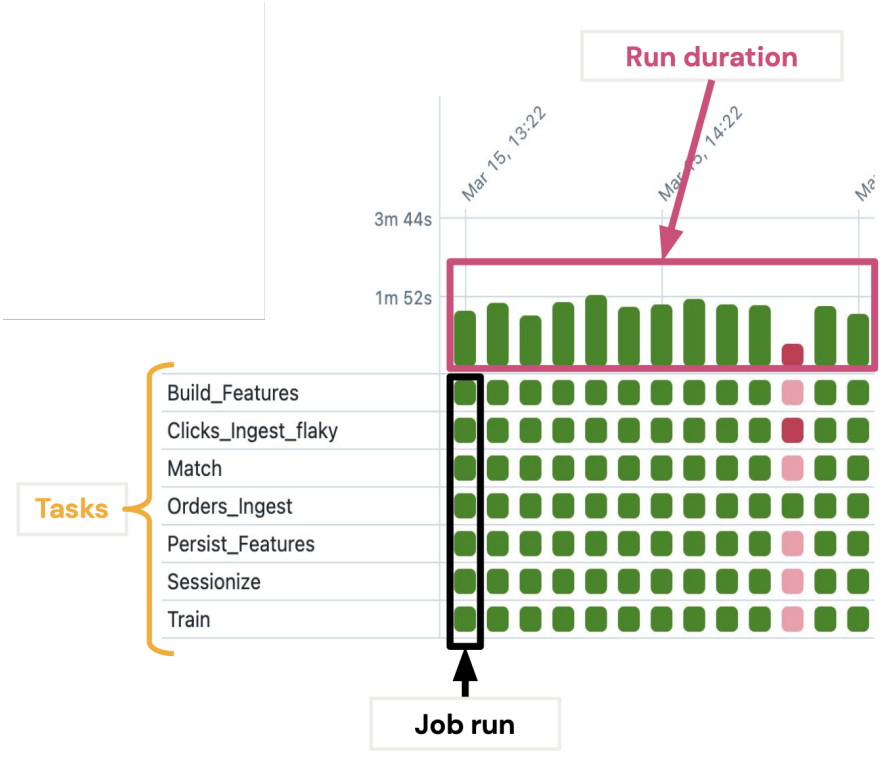


You can run your Databricks job:

1. **Manually**
2. **Scheduled trigger type** periodically. You can define a schedule to run your job on minute, hourly, daily, weekly, or monthly periods and at specified times. You can also specify a time zone for your schedule and pause a scheduled job at any time. You can optionally select the *Show Cron Syntax* checkbox to display and edit the schedule in Quartz Cron Syntax (other data format, string comprised of 6-7 fields separated by white space). Note that there is a minimum interval of 10 seconds between subsequent runs triggered by schedule.
3. **Continuous trigger type** ensure there's always an active run of the job. Databricks Jobs ensures there is always one active run of the job. A new job run starts after the previous run completes successfully or with a failed status, or if there is no instance of the job currently running. You can't use *task dependencies* or *retry policies* with a continuous job.

You can monitor the runs of a job and the tasks that are part of the job by configuring **Notifications** when a run starts, completes successfully, fails, or its duration exceeds a configured threshold. Notifications can be sent to one or more email addresses or system destinations such as Slack, Teams, PagerDuty or any webhook-based service.

In the Databricks UI Workflows, if you choose the **Runs** tab, you will have a **Job Run History** that keeps track of job runs and save information about the success (green), ongoing (pink) or failure (red) of each task in the job run. The visualizations for tasks will update in real time to reflect which tasks are actively running.



If you click on a **task box**, it will render the scheduled notebook in the UI. For DLT pipeline, it will redirect you back to the DLT pipeline GUI for the scheduled pipeline.

A task run may be unsuccessful because it failed or was skipped because a dependent task failed. Using the matrix view, you can quickly identify the **task failures** for your job run (are in red). By hovering over a failed task, you will see associated metadata including the start and end dates, the status, duration cluster details and, in some cases, an error message. To help identify the cause of failure, click the failed task. The **Task run details** page appears, displaying the task's output, error message and associated metadata.

After identifying the cause of failure, you can repair failed or canceled multi-task job by running only the subset of unsuccessful tasks using the **Repair Feature**. It reduces the time and resources required to recover from unsuccessful job runs.

6.2 Questions 16% - Production Pipelines

1. *A data engineer wants to schedule their Databricks SQL dashboard to refresh once per day, but they only want the associated SQL endpoint to be running when it is necessary. Which of the following approaches can the data engineer use to minimize the total running time of the SQL endpoint used in the refresh schedule of their dashboard?* **They can turn on the Auto Stop feature for the SQL endpoint**.

2. *A data analysis team has noticed that their Databricks SQL queries are running too slowly when connected to their always-on SQL endpoint. They claim that this issue is present when many members of the team are running small queries simultaneously. They ask the data engineering team for help. The data engineering team notices that each of the team's queries uses the same SQL endpoint. Which of the following approaches can the data engineering team use*

to improve the latency of the team's queries? **As they run queries simultaneously, we need to scale out (and not up by increasing the size of the cluster or using Auto Stop feature). For that we can increase the maximum bound of the SQL endpoint's scaling range.**

3. *An engineering manager wants to monitor the performance of a recent project using a Databricks SQL query. For the first week following the project's release, the manager wants the query results to be updated every minute. However, the manager is concerned that the compute resources used for the query will be left running and cost the organization a lot of money beyond the first week of the project's release. Which of the following approaches can the engineering team use to ensure the query does not cost the organization any money beyond the first week of the project's release?* **They cannot ensure the query does not cost the organization money beyond the first week of the project's release unless there are some manual intervention.**

4. *A data engineer has a single-task Job that runs each morning before they begin working. After identifying an upstream data issue, they need to set up another task to run a new notebook prior to the original task. Which of the following approaches can the data engineer use to set up the new task?* **They can create a new task in the existing job and then add it as a dependency of the original task.**

5. *A data engineer needs to use a Delta table as part of a data pipeline, but they do not know if they have appropriate permissions. In which of the following locations can the data engineer review their permissions on the table?* **Data Explorer**

6. *Which of the following describes a scenario in which a data engineer will want to use a single-node cluster?* **When they are working interactively with a small amount of data**

7. *A data engineer has developed a data pipeline to ingest data from a JSON source using Auto Loader, but the engineer has not provided any type inference or schema hints in their pipeline. Upon reviewing the data, the data engineer has noticed that all of the columns in the target table are of the string type despite some of the fields only including float or boolean values. Which of the following describes why Auto Loader inferred all of the columns to be of the string type?* **JSON data is a text-based format, so it will infers the string type to all column.**

8. *Which of the following data workloads will utilize a Gold table as its source?* **A job that queries aggregated data to feed into a dashboard.**

9. *Which of the following must be specified when creating a new Delta Live Tables pipeline?* **At least one notebook library to be executed.**

10. *A data engineer has joined an existing project and they see the following query in the project repository:*

```
CREATE STREAMING LIVE TABLE loyal_customers AS
SELECT customer_id
FROM STREAM(LIVE.customers)
WHERE loyalty_level = "high";
```

Which of the following describes why the STREAM function is included in the query? **The STREAM function is needed because the customers table is a streaming live table.**

11. *Which of the following describes the type of workloads that are always compatible with Auto Loader?* **Streaming workloads**

12. *A data engineer and data analyst are working together on a data pipeline. The data engineer is working on the raw, bronze and silver layers of the pipeline using Python, and the analyst is working on the gold layer of the pipeline using SQL. The raw source of the pipeline is*

a streaming input. They now want to migrate their pipeline to use Delta Live Tables. Which of the following changes will need to be made to the pipeline when migrating to DLT? **None of these changes will need to be made. Assumptions - they are working on the same notebook and declaring STREAMING or LIVE keywords during development, before addind to the dlt workflow.**

13. *A data engineer is using the following code block as part of a batch ingestion pipeline to read from a composable table*

```
transactions_df = (spark.read
    .schema(schema)
    .format("delta")
    .table("transactions")
```

Which of the following changes needs to be made so this code block will work when the transactions table is a stream source? **You should replace spark.read with spark.readStream**

14. *A data engineering team has noticed that their Databricks SQL queries are running too slowly when they are submitted to a non-running SQL endpoint. The data engineering team wants this issue to be resolved. Which of the following approaches can the team use to reduce the time it takes to return results in this scenario?* **They can turn on the Serverless feature for the SQL endpoint.**

15. *A data engineer has a Job that has a complex run schedule, and they want to transfer that schedule to other Jobs. Rather than manually selecting each value in the scheduling form in Databricks, which of the following tools can the data engineer use to represent and submit the schedule programmatically?* **Using Cron Syntax**

16. *An engineering manager uses a Databricks SQL query to monitor ingestion latency for each data source. The manager checks the results of the query every day, but they are manually rerunning the query each day and waiting for the results. Which of the following approaches can the manager use to ensure the results of the query are updated each day?* **They can schedule the query to refresh every 1 day from the query's page in Databricks SQL.**

17. *In which of the following scenarios should a data engineer select a Task in the Depends On field of a new Databricks Job Task?* **When another task needs to successfully complete before the new task begins.**

18. *A data engineer has been using a Databricks SQL dashboard to monitor the cleanliness of the input data to a data analytics dashboard for a retail use case. The job has a Databricks SQL query that returns the number of store-level records where sales is equal to zero. The data engineer wants their entire team to be notified via a messaging webhook whenever this value is greater than 0. Which of the following approaches can the data engineer use to notify their entire team via a messaging webhook whenever the number of stores with 0$ in sales is greater than 0?* **They can set up an Alert with a new webhook alert destination.**

19. *A data engineer wants to schedule their Databricks SQL dashboard to refresh every hour, but they only want the associated SQL endpoint to be running when it is necessary. The dashboard has multiple queries on multiple datasets associated with it. The data that feeds the dashboard is automatically processed using a Databricks Job. Which of the following approaches can the data engineer use to minimize the total running time of the SQL endpoint used in the refresh schedule of their dashboard?* **They can turn on the Auto Stop feature for the SQL endpoint. You can't set up the dashboard's SQL endpoint to be serverless otherwise it will run all the time and the cost will be higher.**

7 Data Governance

Data governance is a comprehensive approach that comprises the principles, practices and tools to manage an organization's data assets throughout their lifecycle. It is essential for unlocking the value of data, which is a critical asset for organizations.

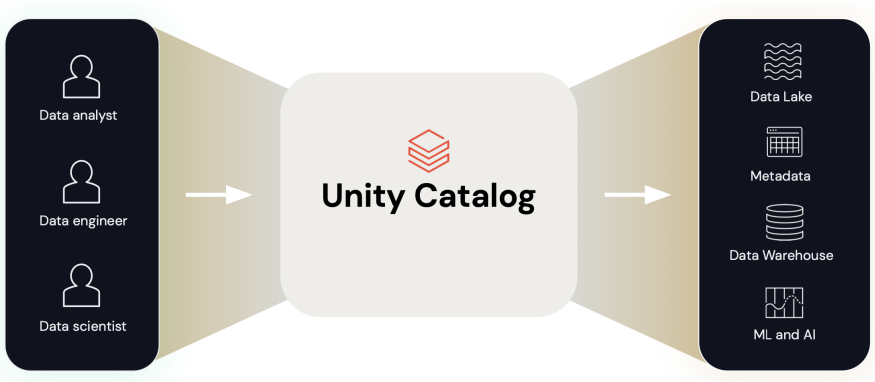
There are 4 keys functional areas:

1. **Data Access Control** controls who has access to which data;
2. **Data Access Audit** captures and records all access to data
3. **Data Lineage** captures upstream sources and downstream consumers, providing an end-to-end view of how data flows across an organization's data estate.
4. **Data Discovery** ability to search for and discover authorized assets across the organization.

To unify governance for data, analytics and AI, Databricks created the **Unity Catalog**.

7.1 Unity Catalog

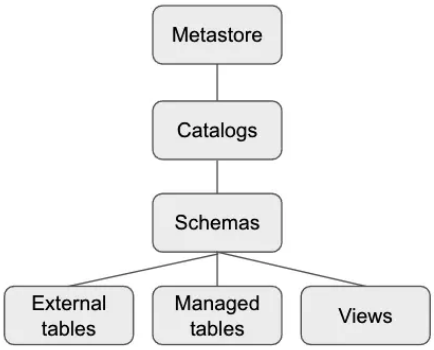
Databricks Unity Catalog offers a unified governance layer for data and AI within the Databricks DI Platform. With Unity Catalog, organizations can seamlessly govern their structured and unstructured data, machine learning models, notebooks, dashboards and files on any cloud or platform. Data scientists, analysts and engineers can use Unity Catalog to securely discover, access and collaborate on trusted data and AI assets, leveraging AI to boost productivity and unlock the full potential of the lakehouse architecture. This unified approach to governance accelerates data and AI initiatives while simplifying regulatory compliance.



It provides:

- **Unified governance across clouds** - fine-grained governance for data lakes across clouds
- **Unified data and AI assets** - centrally share, audit, secure and manage all data types with one simple interface
- **Unified existing catalogs** works in concert with existing data, storage and catalogs : no hard migration required.

In Unity Catalog, the hierarchy of primary data objects flows from metastore to table:

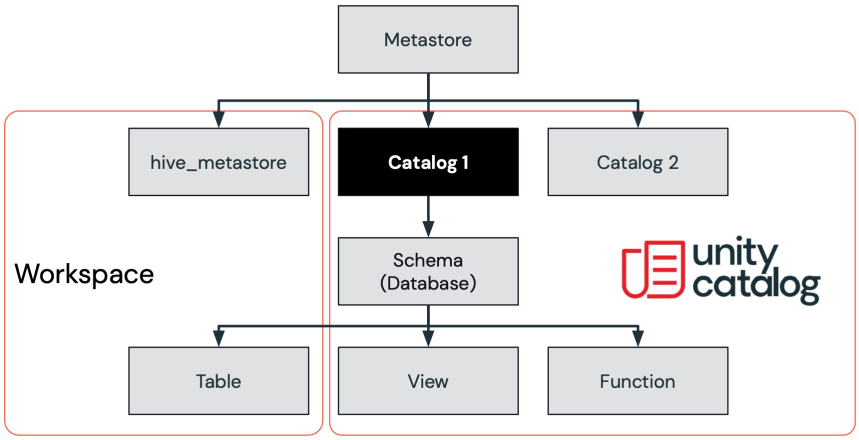


1. **Metastore** top-level container for metadata.
2. **Catalog** The first layer of the object hierarchy, used to organize our data assets.
3. **Schema (Database)** second layer of the object hierarchy and contain tables and views
4. **Table** lowest level in the object hierarchy, tables can be external (stored in external locations in cloud storage) or managed tables (stored in a container in cloud storage that we create expressly for Databricks). We can also create real-only views from tables.

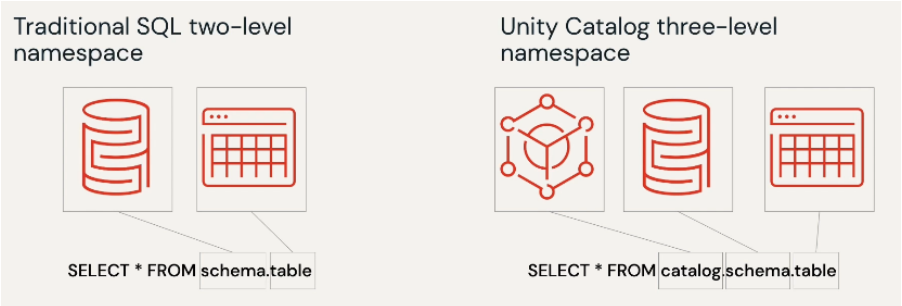
7.1.1 Metastore

It registers metadata about data and AI assets, as well as the permissions that govern access to them (access control lists). When an administrator configures the metastore, he should specify storage account, Owner to give access to metastore, and enable delta sharing allows securely transfer of data. Each metastore that is configured will assign to databricks workspace.

This unity catalog metastore is distinct from the metastore included in Databricks workspaces created before Unity Catalog was released (Hive Metastore). If your workspace includes a legacy Hive metastore, the data in that metastore is available in Unity Catalog in a catalog named hive_metastore.



Each metastore exposes a three-level namespace (catalog.schema.table) that organizes our data:



7.1.2 Catalogs

It's used to organize your data assets. We can create a new catalog in our metastore using:

```
CREATE CATALOG IF NOT EXISTS ${DA.my_new_catalog}
```

Then we can USE this created catalog as the default. Any schemas references will be assumed to be in this catalog unless explicitly overriden by a catalog reference:

```
USE CATALOG ${DA.my_new_catalog}
```

Users can see all catalogs on which they have been assigned the USE CATALOG data permission. Depending on how your workspace was created and enabled for Unity Catalog, your users may have default permissions on automatically provisioned catalogs, including either the "main" catalog or the workspace catalog.

7.1.3 Schema

To access (or list) a table or view in a schema, users must have the USAGE data permission on the schema and its parent catalog, and they must have the SELECT permission on the table or view.

We can create and use a schema, that will be set as the default schema, as follows:

```
CREATE SCHEMA IF NOT EXISTS example;
USE SCHEMA example
```

7.1.4 Tables, View and Function

Tables contain rows of data. To create a table, users must have CREATE and USAGE permissions on the schema, and they must have the USAGE permission on its parent catalog.

```
CREATE OR REPLACE TABLE example (x type, y type,...);
```

To query a table, users must have the SELECT permission on the table, and they must have the USAGE permission on its parent schema and catalog.

Views are stored queries, read-only objects created from one or more tables and views in a metastore.

```
CREATE OR REPLACE VIEW example2 AS (SELECT ... FROM ... GROUP BY...);
```

In this case, SELECT queries on the table and the view will work because we are the data owner. No object-level permissions are required to access these resources.

User-Defined Functions encapsulate custom functionalities in a function that can be involved through queries.

7.1.5 External Storage

To manage access to the underlying cloud storage for external tables, Unity Catalog uses the following object types:

1. **Storage Credentials** encapsulate a long-term cloud credential that provides access to cloud storage. Not very convenient in terms of Access Control when dealing with external cloud storage - it applies to the entire cloud storage container so granting privileges on them apply it to the entire container.

2. **External locations** contain a reference to a storage credential and a cloud storage path. We can sub-divide containers in several pieces and manage control for each of them.

7.1.6 Delta Sharing

Delta Sharing is an open protocol for secure data sharing, making it simple to share data with other organizations regardless of which computing platforms they use.

The primary concepts underlying Delta Sharing in Databricks are shares and recipients:

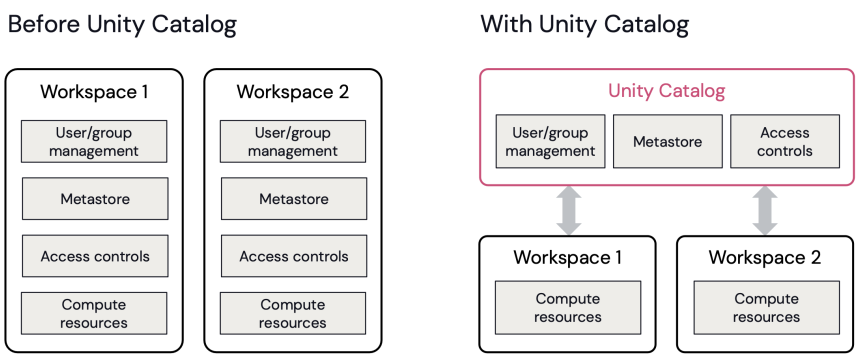
1. **Share** it's a read-only collection of tables and table partitions that a provider wants to share with one or more recipients. You can add or remove tables, views, volumes, models, and notebook files from a share at any time, and you can assign or revoke data recipient access to a share at any time.

2. **Recipients** is an entity that receives shares from a provider. As a sharer, you can define multiple recipients for any given Unity Catalog metastore, but if you want to share data from multiple metastores with a particular user or group of users, you must define the recipient separately for each metastores. A recipient can have access to multiple shares.

7.2 Architecture Unity Catalog

Before Unity Catalog, User/group management where a function of the workspace. User and group were defined within a particular workspace, with no access to other workspace. By default, the metastore was furnished by a Hive Metastore, local to each workspace. Altough integrating an external metastore was also supported.

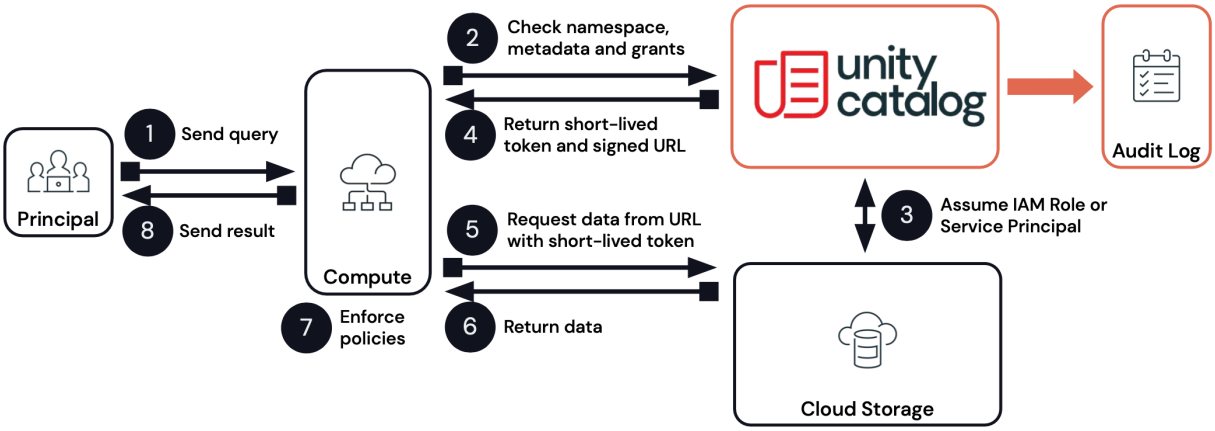
By contrast, the Unity Catalog model separates the element of security out of the workspace. In this architecture, User/group management are done in the Unity Catalog, and are then assigned to one or more workspace(s). Any changes in the security policy defined in the Unity Catalog are automatically propagated to all assigned workspaces and their associated clusters.



7.3 Query Lifecyle in Unity Catalog Security Model

Let's see a query lifecycle:

1. The Principal user send query to the compute resource and the latter begins to process the query.
2. The request is dispatched to the Unity Catalog that will logs it and validates the query against all security constraints within the metastore to which the compute resource is associated.
3. For each object reference in the query, Unity Catalog assumes the correct Cloud Credential object provided by a Cloud Administrator
4. For each object referenced in the query, Unity Catalog generates a temporary token to enable a client to access the data directly from storage and return that token along with an access URL. It allows the cluster or SQL WH to access data directly and securely.
5. The cluster requests data from the URL and token provided.
6. Data are returned by the Cloud Storage.
7. Some enforce policies are applied such as row/column security level,...
8. Finally the data is obtained by the Principal User



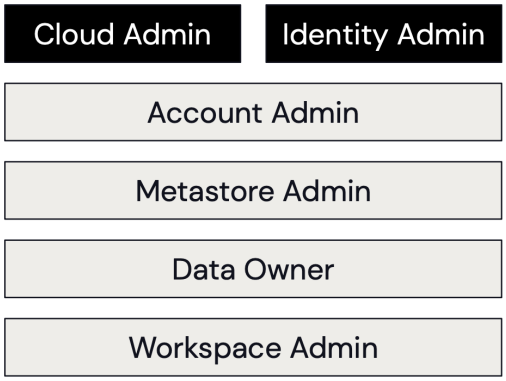
7.4 Compute Resources and Unity Catalog

There are 3 **security access mode** for clusters to choose from:

1. **Supported by UC** there are 2:
   - *Single User* cluster with this mode can only be used by a single user designated at the creation of the cluster. The single user can be changed afterwards. It has multiple language support and features support (libraries installation, dbfs fuse mounts,...). Dynamic views are not supported currently on "single user" mode.
   - *Shared* between multiple users but only Python and SQL are allowed and there is no features support. It has legacy table ACL.
2. **Not Supported by UC** it's the *No Isolation Shared* mode that support multiple language.

Access mode	Supported languages	Legacy table ACL	Credential passthrough	Shareable	RDD API	DBFS Fuse mounts	Init scripts and libraries	Dynamic views	Machine learning
No Isolation Shared	All			●	●	●	●		●
Single user	All				●	●	●	●	●
Shared	SQL Python	●		●				●	

7.5 Roles and Identities



There are several roles:

1. **Cloud Administrator** has the ability to manage underlying cloud resources such as storage accounts/buckets and IAM role (or service principals or managed identities).

2. **Identity Administrators** have the ability to manage users and groups in the Identity Provider (IdP). It provisions identity in the account level through connectors.

3. **Account Admin** has the ability to create or delete metastores and assign metastores to workspaces. It also manages users and groups (integrated with IdP) and it has full access to all data objects

4. **Metastore Admin** creates or drops, grants privileges on, and change ownership of catalogs and other data objects in the store they own. They have full access to all data objects within their store. Same as Account Admin but restricted to the metastore they owned. The Metastore Admin can be changed by the Metastore Admin or an Account Admin.

5. **Data Owner** owns data objects they created. They can create nested objects, grand privileges on and change ownership of owned objects. It can be changed by the Owner, Metastore Admin or Account Admin.

6. **Workspace Admin** manages permissions on workspace assets, restricts access to cluster creation, adds/removes users, elevates users permissions, grant privileges to others and change job ownership. Can be designated by a Account Admin when assigning users to the workspace.

7.6 Data Access Control

Let's see how that security model applies to data objects:

1. **Catalog and Schema** supports CREATE and USAGE privileges; all permissions must be granted and are not implied. Privileges are inherited by child objects.

    - *Create* allows object to create childs objects (schemas for Catalog, tables/views/functions for schemas).
    - *Usage* allows to traverse the container in order to access child objects eg to access a table you need USAGE on schema and catalog as well as appropriate privileges on the table itself.

For example, we grant USAGE on the catalog and schema to the "account users":

```
GRANT USAGE ON CATALOG ${path/to/catalog} TO `account users`;
GRANT USAGE ON SCHEMA example TO `account users`;
```

2. **External and Managed Tables** supports SELECT and MODIFY privileges
   - *Select* allows querying of the table
   - *Modify* allows modifications of the table data (insert, delete) and metadata (alter)
3. **VIEWS** supports *SELECT* as they are read-only tables. They also need USAGE privilege to pass through containers. Users don't need access to the underlying source table to access the view → are a good way to enhance table access control! **Dynamic Views** enables a deeper data access control with their column/row level security and data masking.

   ```
   GRANT SELECT ON VIEW agg_hearthrate TO `account users`;
   ```

   We can query the view:

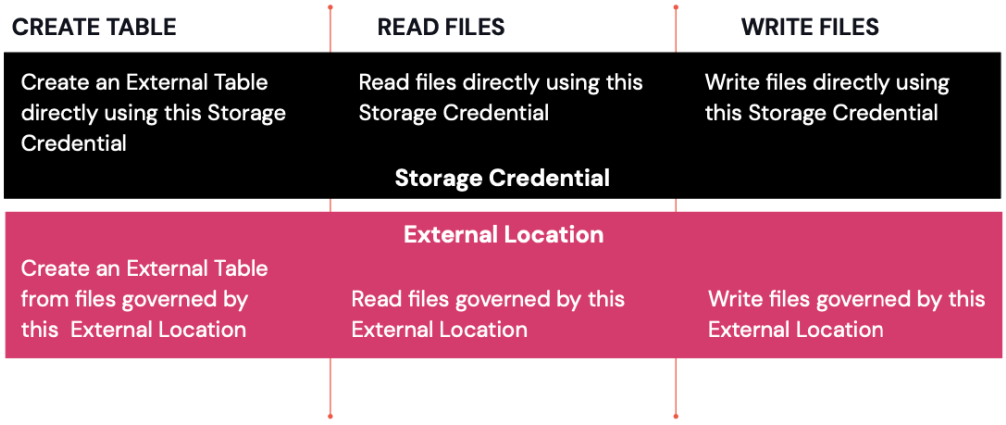
   ```
   SELECT "SELECT * FROM ${path/to/catalog}.example.agg_hearthrate AS Query
   ```

4. **UDFs** supports EXECUTE and USAGE privileges.
   - *Execute* enables usage of the function. The user also requires the USE CATALOG privilege on the catalog and the USE SCHEMA privilege on the schema.

     ```
     GRANT EXECUTE ON FUNCTION fct to 'account users';
     ```

5. **Storage Credential & External Location** support 3 privileges:
   - *READ FILES* allows direct reading of files
   - *CREATE TABLE* allows a table to be created based on stored files
   - *WRITE FILES* allows direct modifications of files

   The meaning of these privileges is different if we used storage credential or external location, because we can have one storage credential for many external locations. Granting privileges on the Storage Credential will automatically grant privileges to the external locations!



6. **Share & Recipient** support *SELECT* only

We can also explore permissions on objects using the function SHOW GRANTS ON:

```
SHOW GRANTS ON VIEW example2
SHOW GRANTS ON TABLE example
SHOW GRANTS ON SCHEMA test
SHOW GRANTS ON CATALOG ${DA.my_new_catalog}
SHOW GRANTS ON FUNCTION fct
```

We can revoke previously issued grants:

```
REVOKE EXECUTE ON FUNCTION fct FROM 'account users'
```

Keep in mind that, as the view is running as its owner (that owns the function and the source table), it did not require access to the table being queried and the function continues to work for the same reason.

7.6.1 Dynamic Views

Dynamic views provide the ability to do fine-grained access control of columns and rows within a table, conditional on the principal running the query. Dynamic views are an extension to standard views that allow us to do things like:

- partially obscure column values or redact them entirely
- omit rows based on specific criteria

Access control with dynamic views is achieved through the use of functions within the definition of the view. These functions include:

- *current_user()* returns the email address of the user querying the view;
- *is_account_group_member()* returns TRUE if the user querying the view is a member of the specified group

For example, suppose we want account users to be able to see some data in the view, but we don't want to disclose patient PII. Let's redefine the view using the is_account_group_member() on specific column:

```
CREATE OR REPLACE VIEW example2 AS
SELECT
    CASE WHEN
        is_account_group_member('account users') THEN 'REDACTED'
        ELSE name
    END AS name
FROM
GROUP BY

% Re-issue the grant
GRANT SELECT ON VIEW example2 to `account users`
```

Now let's suppose we want a view that simply filters out rows from the source. Let's apply the same function to create a view that passes through only rows whose device_id is less than 30. Row filtering is done by applying the conditional as a WHERE clause.

```
CREATE OR REPLACE VIEW example2 AS
SELECT ...
FROM example
WHERE
```

```
    CASE WHEN
    is_account_group_member('account users') THEN device_id < 30
    ELSE TRUE
END
% Re-issue the grant
GRANT SELECT ON VIEW example2 to `account users`
```

One final use case for dynamic views is data masking, or partially obscuring data. We are displaying some of the data rather than replacing it entirely. We'll use a mask() user-defined function that replace all caracters by * except the 2 lasts.

```
CREATE OR REPLACE VIEW example2 AS
SELECT
    CASE WHEN
        is_account_group_member('account users') THEN mask(name)
        ELSE name
    END AS name
FROM
GROUP BY

% Re-issue the grant
GRANT SELECT ON VIEW example2 to `account users`
```

7.7 Best Practices

- UC allows only *1 Metastore per region*
- Metastores in different regions or cloud can share data if needed using *Databricks Delta Sharing* protocol. They will be read-only and you need to set up again Access Control rules.
- Use *catalogs* (NOT metastores) to segregate data by environment score, business unit or sandboxes.
- Manage all identities at the *account-level* and enable UC for workspaces to enable identity federation.
- Use groups rather than users to assign access and ownership to securable objects.
- Use service principals to run production jobs.

7.8 Questions 9% - Data Governance

1. *A new data engineering team, called team, has been assigned to work on a project. The team will need access to database customers in order to see what tables already exist. The team has its own group team. Which of the following commands can be used to grant the necessary permission on the entire database to the new team?*

```
GRANT USAGE ON DATABASE customers TO team;
```

2. *A new data engineering team, called team, has been assigned to an ELT project. The new data engineering team will need full privileges on the database customers to fully manage the project. Which of the following commands can be used to grant full permissions on the database to the new data engineering team?*

```
GRANT ALL PRIVILEGES ON DATABASE customers TO team;
```

3. *A data engineer wants to create a data entity from a couple of tables. The data entity must be used by other data engineers in other sessions. It also must be saved to a physical location. Which of the following data entities should the data engineer create?* **He should create a table**.

4. *A data engineer wants to create a new table containing the names of customers that live in Franc. They have written the following command:*

```
CREATE TABLE customersInFrance
_____ AS
SELECT id,
       first name,
       lastName,
FROM customerLocations
WHERE country = "FRANCE";
```

A senior data engineer mentions that it is organization policy to include a table property indicating that the new table includes personally identifiable information (PII). Which of the following lines of code fills in the above blank to successfully complete the tast?

```
COMMENT "Contains PII"
```

5. *A data engineer has left the organization. The data team needs to transfer ownership of the data engineer's Delta tables to a new data engineer. The new data engineer is the lead engineer on the data team. Assuming the original data engineer has no longer access, which of the following individuals must be the one to transfer ownership of the Delta tables in Data Explorer?* **Workspace administrator**

6. *A new data engineering team "team" has been assigned to an ELT project. The new data engineering team will need full privileges on the table "sales" to fully manage the project. Which of the following commands can be used to grant full permissions on the database to the new data engineering team?*

```
GRANT ALL PRIVILEGES ON TABLE sales TO team;
```

7. *Which of the following statements regarding the relationship between Silver and Bronze tables is always true?* **Silver tables contain a more refined and cleaner view of data than Bronze Tables. Let's note that silver tables contain less data than bronze tables but that is not always the case.**

8. *Which of the following approaches should be used to send the Databricks Job Owner an email in the case that the Job fails?* **Setting up an Alert in the Job page.**

9. *A data engineer needs access to a table new_table, but they do not have the correct permissions. They can ask the table owner for permission, but they do not know who the table owner is. Which of the following approaches can be used to identify the owner of new_table?* **Review the Owner field in the table's page in Data Explorer**