Databricks Data Engineer Associate Exam - Refined Notes

🔰 INTRODUCTION

📌 What is Databricks?

Databricks is a **multi-cloud Lakehouse platform** built on **Apache Spark**. It provides a unified environment for:

- Data Engineering
- Data Science & Machine Learning
- Business Analytics

Databricks supports languages like **Python, SQL, Scala, R, and Java**. It abstracts the complexity of distributed computing by offering a collaborative, scalable, and managed workspace.

💡 What is a Lakehouse?

A **Lakehouse** is a modern data architecture that combines the **scalability of data lakes** with the **performance and reliability of data warehouses**.

Data Lake	Data Warehouse
Stores raw, unstructured/semi-structured data	Stores structured, curated data
Scalable and cost-efficient	High performance and reliable
Difficult to enforce governance	Supports strong governance

Lakehouse = Best of Both Worlds ✅

🧱 Components of the Lakehouse Platform

- **Workspace**: UI for notebooks, jobs, repos, and collaboration.
- **Runtime**: Optimized Apache Spark engine with built-in Delta Lake support.

- **Cloud Services**: APIs, job scheduler, authentication, data access control.

🏗️ Lakehouse Architecture

- **Control Plane** (managed by Databricks): Handles UI, cluster configuration, job orchestration.
- **Data Plane** (resides in your cloud): Where data is stored and computation happens. Data never leaves your cloud account.

📂 Databricks File System (DBFS)

- An abstraction over cloud object storage (e.g., ADLS, S3).
- Allows users to treat cloud storage like a POSIX file system.
- Path formats:
  - `dbfs:/mnt/my_mount/...` (recommended)
  - `/dbfs/mnt/my_mount/...` (for local reference)

👷 Clusters in Databricks

Cluster Types:

- **Single Node**: All operations run on a single machine (driver only).
- **Multi-Node**: One driver and multiple workers for distributed processing.

Cluster Components:

- **Driver Node**: Coordinates execution, manages metadata.
- **Worker Nodes**: Execute tasks in parallel.

DBUs:

- **Databricks Unit (DBU)**: A pricing unit based on cluster usage per hour.

Logs:

- **Driver Logs**: Outputs from the main Spark application.

- **Event Logs**: Track job/cluster lifecycle events.

📒 Notebooks & Magic Commands

Supports multiple languages in the same notebook using magic commands:

- `%python`, `%sql`, `%scala`, `%r`, `%md`, `%fs`, `%sh`, `%run`

Examples:

```
%sql SELECT * FROM customers;
%md # This is a markdown header
```

🔁 Databricks Repos

- Git integration within Databricks
- Supports branching, version control, pull requests
- Recommended for production code workflows

🧱 DATABRICKS LAKEHOUSE PLATFORM

✅ What is Delta Lake?

Delta Lake is an **open-source storage layer** that brings **ACID transactions**, **schema enforcement**, and **time travel** to data lakes.

Built on top of **Apache Parquet**, Delta Lake enables Lakehouse architecture by combining reliability, governance, and performance.

🔑 Key Features of Delta Lake

Feature	Description
ACID Transactions	Guarantees consistency, isolation
Schema Evolution	Modify schema on write/read
Time Travel	Access data from previous versions
Transaction Log	`_delta_log/` directory with all changes

Scalable Metadata	Works with petabytes of data

🔄 Transaction Log Internals

- Stored in JSON files in `_delta_log/`
- Captures all operations: INSERT, DELETE, UPDATE
- Each new transaction = new log file

📁 Delta Table Types

Managed Table

- Stored inside `dbfs:/user/hive/warehouse/`
- Dropping the table deletes both data and metadata

External Table

- Uses `LOCATION` keyword to store data outside default path
- Dropping only deletes metadata, data remains

🧬 Delta Features with Detailed Use Cases

⏳ Time Travel

Time Travel enables you to query older snapshots of a Delta table. This is useful for debugging, audits, recovering from accidental deletes or updates.

Syntax:

```
SELECT * FROM my_table VERSION AS OF 3;
SELECT * FROM my_table TIMESTAMP AS OF '2024-01-01';
DESCRIBE HISTORY my_table;
```

Use Cases:

- Rollback to a previous version
- Compare historical vs current data

- Audit trail of data changes

📦 OPTIMIZE & Z-ORDER

Delta tables often consist of many small files due to distributed writes. This can slow down queries. Use `OPTIMIZE` to compact files and `ZORDER` to optimize data skipping.

OPTIMIZE:

```
OPTIMIZE my_table;
```

ZORDER (Data Skipping):

```
OPTIMIZE my_table ZORDER BY (customer_id);
```

Best Practices:

- Use on large tables with frequent queries
- ZORDER on high-cardinality columns used in filters
- Improves performance for filter-based queries

🧹 VACUUM (Garbage Collection)

Delta stores old versions of files to enable Time Travel. `VACUUM` removes these files after a retention period.

Syntax:

```
VACUUM my_table RETAIN 168 HOURS;
```

Important:

- Default retention = 7 days (168 hours)
- Time Travel is no longer possible for removed versions
- To override protection (not recommended in prod):

```
SET spark.databricks.delta.retentionDurationCheck.enabled = false;
VACUUM my_table RETAIN 0 HOURS;
```

Use Cases:

- Free up storage
- Retain only current data state
- Perform cleanup after data is finalized

🔄 CLONING Delta Tables

Cloning is used to make a copy of a Delta table.

Shallow Clone:

```
CREATE TABLE new_table SHALLOW CLONE existing_table;
```

- Copies metadata only (not data files)
- Fast, minimal storage
- Useful for testing, exploration

Deep Clone:

```
CREATE TABLE new_table DEEP CLONE existing_table;
```

- Copies full data + metadata
- Used for backup, DR, migrations

Use Cases:

- Safe experimentation
- Create test environments
- Archive snapshot of data
- Backup before risky operations

📊 Hive Metastore & Metadata

- **CREATE DATABASE db_name LOCATION '...';**
- **CREATE TABLE ... LOCATION '...';**
- **DESCRIBE DATABASE EXTENDED db_name;**
- **DESCRIBE EXTENDED table_name;**

⚡ AUTO LOADER IN DATABRICKS

🔍 What is Auto Loader?

Auto Loader is a **streaming file ingestion tool** in Databricks built on top of Spark Structured Streaming.

It incrementally and efficiently processes **new files** as they arrive in **cloud object storage**, without reprocessing previously loaded files.

💡 **Using** `.format("cloudFiles")` **automatically enables Auto Loader.**
There's no separate service to turn on — simply using `cloudFiles` format activates Auto Loader logic and behavior.

✅ Key Advantages:

- **Increments only**: Loads only new/changed files
- **Schema inference**: Auto-detects and evolves schema
- **Scales effortlessly**: Handles millions of files
- **Reliable**: Uses checkpoints to track processing progress
- **Serverless**: No extra infrastructure or configuration

🛠️ Basic Syntax

```
spark.readStream \
  .format("cloudFiles") \
  .option("cloudFiles.format", "parquet") \
```

```
  .option("cloudFiles.schemaLocation", "/mnt/schema") \
  .load("/mnt/source")
```

💡 Output Sink Example:

```
df.writeStream \
  .format("delta") \
  .option("checkpointLocation", "/mnt/checkpoints") \
  .table("target_table")
```

🔐 Understanding Checkpoints in Auto Loader

Checkpoints are crucial to ensure **fault tolerance and exactly-once processing** in streaming pipelines.

- Spark stores metadata such as:
  - What files have already been processed
  - Progress of the stream
  - State of aggregations (if any)
- This metadata is stored in the **checkpoint directory** you define via `.option("checkpointLocation", "/path")`

Why Checkpoints Matter:

- If your stream fails or is stopped, it **resumes exactly where it left off** using the checkpoint state.
- Prevents duplication or data loss

Best Practices:

- Use a **unique checkpoint location per stream**
- Never share checkpoint directories across multiple streams
- Place checkpoint directories in **reliable storage** (e.g., DBFS, ADLS, S3)

🔑 Important Parameters

Parameter	Description
`cloudFiles.format`	File format (e.g., `csv`, `json`, `parquet`)
`cloudFiles.schemaLocation`	Path to store inferred schema
`checkpointLocation`	Stores stream progress, required for recovery
`cloudFiles.includeExistingFiles`	If true, processes existing files too
`cloudFiles.useNotifications`	Uses native events from S3/GCS for faster detection
`cloudFiles.maxBytesPerTrigger`	Controls data volume per batch
`cloudFiles.maxFilesPerTrigger`	Controls number of files per batch

🛠️ ETL WITH SPARK SQL & PYTHON (DETAILED)

Databricks supports flexible ETL workflows using both SQL and PySpark. Delta Lake allows transactional operations and simplifies handling of big data in both batch and streaming pipelines.

🔽 Reading External Files

1. SQL Approach – Ad Hoc Queries

```
SELECT * FROM json.`/mnt/data/customers/`;
SELECT * FROM parquet.`/mnt/data/orders/`;
```

- Great for quick exploration or analysis.
- Does not support complex options (like header/delimiter for CSV).

2. PySpark Approach – Recommended for Production

```
df = spark.read.format("csv") \
  .option("header", True) \
```

```
  .option("delimiter", ";") \
  .load("/mnt/data/books")
```

- Can be extended to validate schema, apply transformations, handle corrupt records.
- Preferred for production due to better control.

💾 Writing as Delta Tables

```
df.write.format("delta").mode("overwrite").saveAsTable("books_delta")
```

- You can use modes like `overwrite`, `append`, `ignore`, `errorifexists`
- Creates a managed Delta table if no `LOCATION` is specified.

🛠️ Creating Tables from Files

1. CTAS – Create Table As Select

```
CREATE TABLE customers AS
SELECT * FROM json.`/mnt/data/customers/`;
```

- Infers schema automatically
- Creates a managed Delta table
- Not suited for CSV due to limited options

2. Create Table Referencing External Files

```
CREATE TABLE books_csv (
  id INT,
  title STRING,
  author STRING
)
USING CSV
```

```
OPTIONS (
  header = "true",
  delimiter = ";"
)
LOCATION "/mnt/data/books";
```

- Data is read directly from source files
- No time travel, schema evolution, or transaction logs

🔁 Insert, Overwrite, Merge

1. Overwrite Entire Table

```
CREATE OR REPLACE TABLE orders AS
SELECT * FROM parquet.`/mnt/orders_data`;
```

- Replaces both schema and data

2. Insert Overwrite

```
INSERT OVERWRITE TABLE orders
SELECT * FROM parquet.`/mnt/new_orders`;
```

- Replaces existing rows
- Schema must match

3. Append (Insert Into)

```
INSERT INTO orders
SELECT * FROM parquet.`/mnt/orders_new`;
```

- Adds new rows to table without affecting existing ones

4. Merge Into (Upsert)

```
MERGE INTO customers AS target
USING customer_updates AS source
ON target.customer_id = source.customer_id
WHEN MATCHED THEN
  UPDATE SET target.email = source.email
WHEN NOT MATCHED THEN
  INSERT *;
```

- Ideal for Change Data Capture (CDC)
- Supports deduplication, schema evolution

📦 Handling Complex and Nested Data

JSON Field Access

```
SELECT
  customer_id,
  profile:first_name AS first_name,
  profile:address:city AS city
FROM customers;
```

- Allows direct access to nested JSON without schema conversion

---

💥 Exploding Arrays

What is `explode()`?

Converts an array column into multiple rows, one per element.

Input (before explode):

```
{
  "customer_id": 1,
  "order_id": 101,
  "books": [
```

```
    {"book_id": "B1", "price": 10},
    {"book_id": "B2", "price": 20}
  ]
}
```

Output (after explode):

customer_id	order_id	book_struct
1	101	{B1, 10}
1	101	{B2, 20}

```
SELECT customer_id, order_id, explode(books) AS book_struct FROM orders;
```

📚 Flattening Nested Arrays

What is `flatten()`?

Used to flatten arrays of arrays into a single array.

Input:

```
{
  "nested_books": [["B1", "B2"], ["B3"]]
}
```

Output:

```
["B1", "B2", "B3"]
```

```
SELECT flatten(nested_books) AS flat_books FROM orders;
```

Often used with `array_distinct()` and `collect_set()`:

```
SELECT customer_id, array_distinct(flatten(collect_set(book_ids))) AS unique_books FROM orders GROUP BY customer_id;
```

🧠 Higher-Order Functions

Functions that operate on array elements using lambda syntax.

`FILTER()` – Keep elements that match a condition

```
SELECT FILTER(books, book → book.qty > 1) AS multi_copies FROM orders;
```

`TRANSFORM()` – Modify each element

```
SELECT TRANSFORM(books, book → book.price * 0.9) AS discounted_prices FROM orders;
```

`AGGREGATE()` – Reduce array to a single value

```
SELECT AGGREGATE(books, 0D, (acc, book) → acc + book.price) AS total_price FROM orders;
```

🔧 User-Defined Functions (UDFs)

Custom logic reusable in SQL.

Create URL from email

```
CREATE OR REPLACE FUNCTION get_url(email STRING)
RETURNS STRING
RETURN CONCAT('http://', SPLIT(email, '@')[1]);
```

Detect Provider

```
CREATE OR REPLACE FUNCTION detect_provider(email STRING)
RETURNS STRING
RETURN CASE
  WHEN email LIKE '%gmail.com' THEN 'Google'
  WHEN email LIKE '%yahoo.com' THEN 'Yahoo'
  WHEN email LIKE '%edu' THEN 'University'
  ELSE 'Unknown'
END;
```

Drop Function

```
DROP FUNCTION IF EXISTS get_url;
```

📊 Pivoting and Aggregations

```
SELECT * FROM (
  SELECT customer_id, category, price FROM order_enriched
)
PIVOT (
  SUM(price) FOR category IN ('Fiction', 'Science', 'History', 'Tech')
);
```

- Turns categorical rows into columns for wide-format analytics

---

🔁 Set Operations

```
SELECT * FROM orders
UNION
SELECT * FROM orders_updates;

SELECT * FROM orders
INTERSECT
SELECT * FROM orders_updates;
```

```
SELECT * FROM orders
EXCEPT
SELECT * FROM orders_updates;
```

- Compare datasets
- Track differences or overlaps between two snapshots

🔁 INCREMENTAL DATA LOADING

Incremental data loading means **processing only new or modified data** instead of reprocessing the entire dataset every time. It is essential for scalability and efficiency in both batch and streaming pipelines.

💡 WHY Incremental Loading?

- Avoids scanning entire datasets repeatedly
- Improves performance and reduces cost
- Ensures low-latency processing for real-time use cases

✅ APPROACHES TO INCREMENTAL INGESTION

Tool	Interface	Ideal Use Case
`COPY INTO`	SQL	Periodic ingestion of few hundred/thousand files
Auto Loader	PySpark API	Large-scale ingestion, real-time pipelines

---

✅ COPY INTO – SQL-Based Incremental Load

🔷 What it does:

- Loads **only new files** from a directory (tracked internally)
- Creates a **Delta table** (if not already present)

- Supports schema merging

🧾 Syntax:

```
COPY INTO target_table
FROM '/mnt/data/source/'
FILEFORMAT = CSV
FORMAT_OPTIONS ('header' = 'true')
COPY_OPTIONS ('mergeSchema' = 'true');
```

🧠 How it works:

- Databricks creates a **load history** (tracked by file name/hash)
- Skips already ingested files in future runs

🔍 Good to know:

- Schema evolution (adding new columns) is supported if `mergeSchema=true`
- Not meant for sub-second latency
- Works best when files are appended (not updated in place)

⚡ AUTO LOADER – Real-Time Incremental Load

Auto Loader provides continuous ingestion by integrating **file arrival detection + streaming + Delta Lake**.

🔷 Key Concepts:

- Uses **cloudFiles** to enable Auto Loader with a single line
- Internally uses **Structured Streaming**
- Supports schema inference and evolution

🧾 Syntax:

```
df = spark.readStream \
  .format("cloudFiles") \
```

```
  .option("cloudFiles.format", "parquet") \
  .option("cloudFiles.schemaLocation", "/mnt/schema") \
  .load("/mnt/data")

df.writeStream \
  .format("delta") \
  .option("checkpointLocation", "/mnt/checkpoints") \
  .table("target_table")
```

🧠 What Happens:

- Auto Loader watches the folder
- When a new file lands → it's automatically picked up
- Metadata & progress are logged in **checkpoint** & **schema** locations

---

✅ CHECKPOINTING – CRITICAL FOR STREAMING & AUTO LOADER

Checkpoints are how Spark keeps track of what it has already read and written.

🔐 Purpose:

- Prevents reprocessing already seen data (ensures **exactly-once** delivery)
- Stores:
  - File offsets (which files were read)
  - Watermarks for aggregations
  - Progress of stateful operations

✅ Best Practices:

- **Use a unique path** per stream/job
- Store on a **persistent location** (e.g., DBFS, ADLS, S3)
- **Don’t share checkpoint directories** across streams

⏱️ STRUCTURED STREAMING TRIGGERS

Trigger Type	Behavior Description
Default (micro-batch)	Continuously processes as files arrive
`processingTime="5s"`	Triggers batches every 5 seconds
`trigger(once=True)`	Runs once on all available data then stops
`trigger(availableNow=True)`	Batches all current data and stops (batch-style)

🔁 OUTPUT MODES

Output Mode	Description	Use Case
`append`	Adds new rows only	Most streaming inserts
`complete`	Rewrites full result (for aggregations)	Aggregated summaries

📘 STRUCTURED STREAMING EXAMPLE: FROM SOURCE TO DELTA

1️⃣ Read Streaming Source

```
stream_df = spark.readStream.format("delta").load("/mnt/source")
stream_df.createOrReplaceTempView("streaming_view")
```

2️⃣ Transform / Aggregate (SQL)

```
CREATE OR REPLACE TEMP VIEW author_counts_view AS
SELECT author, COUNT(*) AS book_count
FROM streaming_view
GROUP BY author;
```

3️⃣ Write to Delta Sink

```
df_agg = spark.table("author_counts_view")
df_agg.writeStream \
  .format("delta") \
  .outputMode("complete") \
```

```
    .trigger(processingTime="10 seconds") \
    .option("checkpointLocation", "/mnt/checkpoints/authors") \
    .table("author_counts")
```

4️⃣ Insert new data to source to trigger change

```
INSERT INTO books VALUES (...);
```

🔄 MULTI-HOP (MEDALLION) ARCHITECTURE – INCREMENTAL PIPELINES

Medallion architecture is a **layered design pattern**:

Layer	Description	Table Name
Bronze	Raw ingestion with metadata	`orders_bronze`
Silver	Cleaned, enriched with lookups/filters	`orders_silver`
Gold	Business aggregates and summaries	`daily_customer_books`

Each layer can stream from the previous one, ensuring real-time updates.

⚙️ PRODUCTION PIPELINES WITH DELTA LIVE TABLES (DLT)

Delta Live Tables (DLT) is a Databricks framework that simplifies the creation, deployment, and monitoring of reliable production data pipelines using Delta Lake.

💡 WHY USE DLT?

- Automates complex engineering tasks
- Ensures reliability through **auto retries, schema checks, and testing**
- Built-in support for **Auto Loader**, **Quality Enforcement**, and **Incremental Processing**
- Visualize pipelines with **DAG** (Directed Acyclic Graph)

🧱 DLT CORE CONCEPTS

Term	Meaning
`LIVE`	Prefix for referencing tables/views in the DLT pipeline
`STREAMING`	Indicates streaming input (e.g., Auto Loader)
`cloud_files`	SQL function that enables Auto Loader
`CONSTRAINT`	Adds data quality rule (drop/bypass/fail rows)
Development Mode	Fast dev/test mode (no retries, reuses clusters)
Production Mode	Full retry/recovery mode (new cluster, managed compute)

🏗️ DLT TABLE TYPES

Type	Description
Table	Materialized and persisted Delta table
View	Logical transformation, not materialized
Temporary	Used for intermediate transformations (not written)

🧪 DATA QUALITY CHECKS (CONSTRAINTS)

```
CREATE STREAMING LIVE TABLE orders_raw
AS SELECT * FROM cloud_files("/mnt/orders", "parquet")

CONSTRAINT valid_order EXPECT (order_id IS NOT NULL)
ON VIOLATION DROP ROW;
```

Mode	Action
`DROP ROW`	Skip bad rows
`FAIL UPDATE`	Pipeline fails on data quality violation
None	Row processed, but logged as violation

🔄 DLT & AUTO LOADER IN SQL

DLT enables Auto Loader directly in SQL using `cloud_files()`:

```
CREATE STREAMING LIVE TABLE books_bronze
AS SELECT * FROM cloud_files("/mnt/books", "json")
```

- Automatically triggers Auto Loader
- Inherits schema inference and checkpointing

🥈 EXAMPLE: BRONZE → SILVER → GOLD

🟫 BRONZE TABLE – Ingest Raw Data

```
CREATE STREAMING LIVE TABLE orders_bronze
AS SELECT *, current_timestamp() AS ingestion_time
FROM cloud_files("/mnt/orders", "parquet")
```

🥈 SILVER TABLE – Clean and Enrich

```
CREATE LIVE TABLE orders_silver
AS SELECT
  o.*,
  c.customer_name
FROM LIVE.orders_bronze o
JOIN customers c ON o.customer_id = c.id
WHERE o.amount > 0
```

🥇 GOLD TABLE – Aggregate for BI

```
CREATE LIVE TABLE customer_daily_orders
AS SELECT
  customer_id,
  DATE(order_date) AS day,
  COUNT(*) AS total_orders
```

```
FROM LIVE.orders_silver
GROUP BY customer_id, DATE(order_date)
```

🚀 CREATING A DLT PIPELINE (UI STEPS)

1. Go to **Workflows → Delta Live Tables → Create Pipeline**
2. Set pipeline name, storage location, and target schema
3. Choose **mode** (development or production)
4. Attach a notebook or SQL script with your `LIVE TABLE` definitions
5. Click **Start** to deploy

📈 MONITORING & DEBUGGING

- View DAG of data flow
- Inspect per-task metrics (rows read/written, time)
- Access logs via `/system/events`
- Automatically retries failed jobs in Production mode

🔄 CDC SUPPORT – APPLY CHANGES INTO

DLT supports Change Data Capture using `APPLY CHANGES INTO`

```
APPLY CHANGES INTO live.customers_silver
FROM stream_updates
KEYS (customer_id)
APPLY AS DELETE WHEN operation = 'DELETE'
SEQUENCE BY version_column
COLUMNS (customer_id, name, email, version_column)
```

✅ Supports out-of-order updates
✅ Performs upserts or deletes
✅ Ideal for streaming CDC data from Kafka or lake sources

✅ DLT helps you build **production-grade, observable, and maintainable** pipelines with minimal code and built-in reliability!

📆 JOBS & TASK ORCHESTRATION

Databricks Jobs let you **schedule and automate** multiple tasks in sequence — great for building and maintaining pipelines, automating reporting, and alerting.

🧰 TYPICAL MULTI-TASK JOB EXAMPLE

Task Name	Task Type	Description
`Land_New_Data`	Notebook	Simulates new file drop using helper function
`DLT_Pipeline`	Delta Live Tables	Executes Bronze → Silver → Gold transformation flow
`Pipeline_Results`	Notebook	Reads from the final Gold table and reports results

🔧 CONFIGURING JOBS

- Use **workflows UI** to add new tasks
- Define dependencies between tasks (e.g., Task 2 runs after Task 1)
- Choose compute (interactive or job cluster)

📆 SCHEDULING JOBS

- Use **cron expressions** to define frequency (hourly, daily, etc.)
- Can trigger jobs **manually** or based on events (coming soon)

📣 NOTIFICATIONS

- Send email, Slack, or webhook alerts on:
  - Job start
  - Success/failure
  - Skipped or retried tasks

🔁 REPAIR RUNS

- If a job fails, you can **repair** it
- Re-executes only the failed task (not the entire chain)

✅ BEST PRACTICES

- Use **job clusters** for cost efficiency in production
- Leverage **versioned notebooks/repos** for CI/CD integration
- Keep tasks modular for easier debugging and reusability

📊 DATABRICKS SQL (DBSQL) – BI & DASHBOARDS

Databricks SQL enables analytics & dashboards over Delta Lake tables using SQL Warehouses.

📍 ACCESS VIA SQL PERSONA

- Switch workspace persona → `SQL`
- UI opens tabs: Queries, Dashboards, Alerts, SQL Warehouses

⚙️ CREATE A SQL WAREHOUSE

1. Go to `SQL Warehouses` tab
2. Click **Create Warehouse**
3. Choose size (e.g., 2X Small) and start it

✍️ CREATE QUERIES

- Use **SQL Editor**
- Browse available tables via schema browser
- Write and run SQL like:

```
SELECT pickup_zip, SUM(fare_amount) AS total_fare
FROM trips
GROUP BY pickup_zip;
```

- Save queries and share links

📈 BUILD DASHBOARDS

- Save query results → Add Visualization
- Choose chart type: bar, pie, map, line
- Drag tiles to design layout

📤 SHARE DASHBOARDS

- Choose permissions: `Can View`, `Can Edit`, `Can Run`
- Share with specific users or groups

🚨 ALERTS

- Create alert from a saved query
- Define trigger condition: e.g., `fare > 10000`
- Notification options:
  - Email
  - Slack
  - Webhook
- Managed via `Alert Destinations` in UI

✅ Dashboards + Alerts = powerful for **BI reporting**, **SLAs**, and **monitoring data pipelines**

🔐 DATA GOVERNANCE & UNITY CATALOG

Data governance in Databricks ensures that access to data is **secured, auditable, and manageable** using SQL-based permissions. Unity Catalog brings **centralized, cross-workspace governance**.

🎯 PURPOSE

- Define **who can access, modify, or view** data and metadata
- Govern access to **tables, views, functions, files**
- Maintain **fine-grained control** at every level — catalog to row

---

🗃️ SECURABLE OBJECT TYPES

Object	Description
`CATALOG`	A top-level namespace (container of schemas)
`SCHEMA`	A database within a catalog
`TABLE`	Managed or external Delta table
`VIEW`	Logical view over one or more tables
`FUNCTION`	UDFs or SQL functions
`ANY FILE`	Direct file-level access bypassing tables

🔑 PRIVILEGES

Privilege	Grants Ability To
`USAGE`	Reference catalog/schema/table (required prerequisite)
`SELECT`	Read data
`MODIFY`	Insert/update/delete data
`CREATE`	Create new tables, views, functions
`READ_METADATA`	View schema and metadata
`ALL PRIVILEGES`	Grant all of the above

⚠️ `USAGE` must be granted alongside `SELECT` or `MODIFY` for access to work.

✅ WHO CAN GRANT ACCESS?

Role	Can Grant On
Admin	Everything
Catalog Owner	All schemas and objects within the catalog
Schema Owner	Tables, views, functions in the schema

Table Owner	That specific table

🛠️ COMMON SQL COMMANDS

```
-- Grant SELECT permission
GRANT SELECT ON TABLE customers TO `data_analyst`;

-- Allow insert/update/delete
GRANT MODIFY ON TABLE customers TO `data_engineer`;

-- Revoke privileges
REVOKE SELECT ON TABLE customers FROM `user1`;

-- Deny access
DENY SELECT ON TABLE customers TO `user1`;

-- View assigned permissions
SHOW GRANTS ON TABLE customers;
```

🏗️ WORKSPACE-LEVEL EXAMPLE

```
-- Create database and table
CREATE DATABASE hr_db;
CREATE TABLE hr_db.employees (id INT, name STRING, salary DOUBLE);

-- Create a view
CREATE VIEW hr_db.high_salary_employees AS
SELECT * FROM hr_db.employees WHERE salary > 5000;

-- Grant permissions
GRANT USAGE, SELECT ON DATABASE hr_db TO `HR Team`;
GRANT SELECT ON VIEW hr_db.high_salary_employees TO `analyst@exampl
e.com`;
```

🧭 USING DATA EXPLORER (UI)

- Go to **Data** tab → Select object
- Click **Permissions** tab
- Grant/revoke/edit permissions for users/groups

✅ Useful for quick permission changes, audits, or non-technical users

---

🔐 UNITY CATALOG – ENTERPRISE DATA GOVERNANCE

Unity Catalog is the next-gen governance framework that enables **centralized access control** across **multiple workspaces and clouds**.

✅ KEY FEATURES

- One metastore for **all workspaces**
- Fine-grained permissions on tables, views, functions, files
- Integrated with **SCIM + identity providers**
- Supports **column/row-level security** (coming enhancements)

🏗️ UNITY CATALOG STRUCTURE

Level	Example	Description
Metastore	(hidden layer)	Governs catalogs
Catalog	`main`	Container of schemas
Schema	`sales`	Contains tables, views, functions
Table	`orders`	Actual data asset

🔁 Namespace: `catalog.schema.table`

🔐 UNITY CATALOG ACCESS CONTROL

Object	Privileges
Table/View	`SELECT`, `MODIFY`, `CREATE`

Schema/Catalog	`USAGE`, `CREATE`
File Storage	`READ FILES`, `WRITE FILES`
Functions	`EXECUTE`

```
-- Example: Grant read access
GRANT SELECT ON TABLE main.sales.orders TO `analyst@example.com`;

-- Grant create access to a schema
GRANT CREATE ON SCHEMA main.sales TO `data_engineer`;
```

👥 IDENTITY TYPES IN UNITY CATALOG

- **Users** – Identified via email (SSO-linked)
- **Groups** – From identity provider (e.g., Azure AD)
- **Service Principals** – For programmatic access

🛠️ STORAGE CONFIGURATION

- Create **Storage Credentials** for cloud containers
- Create **External Locations** (mapped mount paths)

```
-- Example
CREATE EXTERNAL LOCATION bronze_data
URL 'abfss://raw@datalake.dfs.core.windows.net/books'
WITH STORAGE CREDENTIAL my_adls_credential;
```

📈 LINEAGE & AUDITABILITY

- Track **table/view dependencies**
- Visualize **data flow from ingestion → output**
- Log access for audit compliance

🔐 ROW-LEVEL & COLUMN-LEVEL SECURITY (RLS & CLS)

Unity Catalog supports fine-grained access control using **dynamic views** to implement **Row-Level Security (RLS)** and **Column-Level Security (CLS)**.

✅ ROW-LEVEL SECURITY (RLS)

Restrict rows returned based on user identity.

```
CREATE OR REPLACE VIEW secure_employees AS
SELECT * FROM hr_db.employees
WHERE city = current_user();

-- Grant access to view (not base table)
GRANT SELECT ON VIEW secure_employees TO `hr_analyst`;
```

🧠 Useful when users should only see data relevant to their region, department, or role.

✅ COLUMN-LEVEL SECURITY (CLS)

Restrict certain columns based on user roles or use masking.

```
CREATE OR REPLACE VIEW employees_masked AS
SELECT
  id,
  name,
  CASE
    WHEN current_user() = 'hr_manager@example.com' THEN salary
    ELSE NULL
  END AS salary
FROM hr_db.employees;
```

🧠 Only authorized users see sensitive columns (e.g., salary).

✅ **Best Practice**: Apply security via **views**, never directly on base tables.

🌐 DELTA SHARING – CROSS-ORG DATA COLLABORATION

Delta Sharing is an **open protocol** to share live data between organizations **without duplication or ETL**.

✅ WHAT IT ENABLES

- Share Delta tables **across clouds or regions**
- Share data with **external clients** (even if they don't use Databricks)
- No need to copy/export large datasets

📦 KEY COMPONENTS

Component	Role
Share	Defines which tables are shared
Recipient	External user/org who receives access
Table	The actual Delta table(s) to be shared
Credential	Used to authenticate (e.g., token-based or open)

🔧 HOW TO CREATE A SHARE (UI or SQL)

```
-- Step 1: Create a share
CREATE SHARE customer_usage_data;

-- Step 2: Add table to share
ALTER SHARE customer_usage_data ADD TABLE main.sales.usage_summary;

-- Step 3: Create recipient
CREATE RECIPIENT vendor_a USING IDENTITY 'vendor_a@external.com';

-- Step 4: Grant share to recipient
GRANT USAGE ON SHARE customer_usage_data TO RECIPIENT vendor_a;
```

🔗 Recipients access data via URL or native connectors (e.g., Tableau, Power BI)

🔐 SECURITY & AUDIT

- Data is read-only for external users
- Full audit logs are available via Unity Catalog
- Supports **token expiration, access revocation**

✅ Data stays in your control. External orgs only read from it.

With Unity Catalog + Delta Sharing + RLS/CLS, Databricks offers **fine-grained, secure, and flexible data governance** suitable for multi-org, multi-cloud architectures.