

Databricks Reference

From first notebook to production-grade lakehouse pipelines.

Last reviewed 29 Sep 2026 ·

Prerequisites: [PySpark](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/pyspark-reference/) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/pyspark-reference/) · [Cloud Storage](https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/cloud-storage/) (https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/cloud-storage/)

Related: [Apache Iceberg](https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/apache-iceberg/) (https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/apache-iceberg/) · [MLflow](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/mlflow/) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/mlflow/) · [Data Quality](https://sarangambekar1997.github.io/data-engineering-handbook/05-quality-governance/data-quality/) (https://sarangambekar1997.github.io/data-engineering-handbook/05-quality-governance/data-quality/) · [Glossary](https://sarangambekar1997.github.io/data-engineering-handbook/99-reference/glossary/) (https://sarangambekar1997.github.io/data-engineering-handbook/99-reference/glossary/)

Practice: [Lab 03 — Spark Lakehouse](https://github.com/sarangambekar1997/data-engineering-handbook/tree/main/labs/03-spark-lakehouse) (https://github.com/sarangambekar1997/data-engineering-handbook/tree/main/labs/03-spark-lakehouse)

Overview

Challenge: Organizations historically ran two separate systems: a low-cost *data lake* (files in object storage) for raw data and machine learning, and a *data warehouse* for curated SQL reporting. Data was copied between them, the copies diverged, and governance had to be implemented twice.

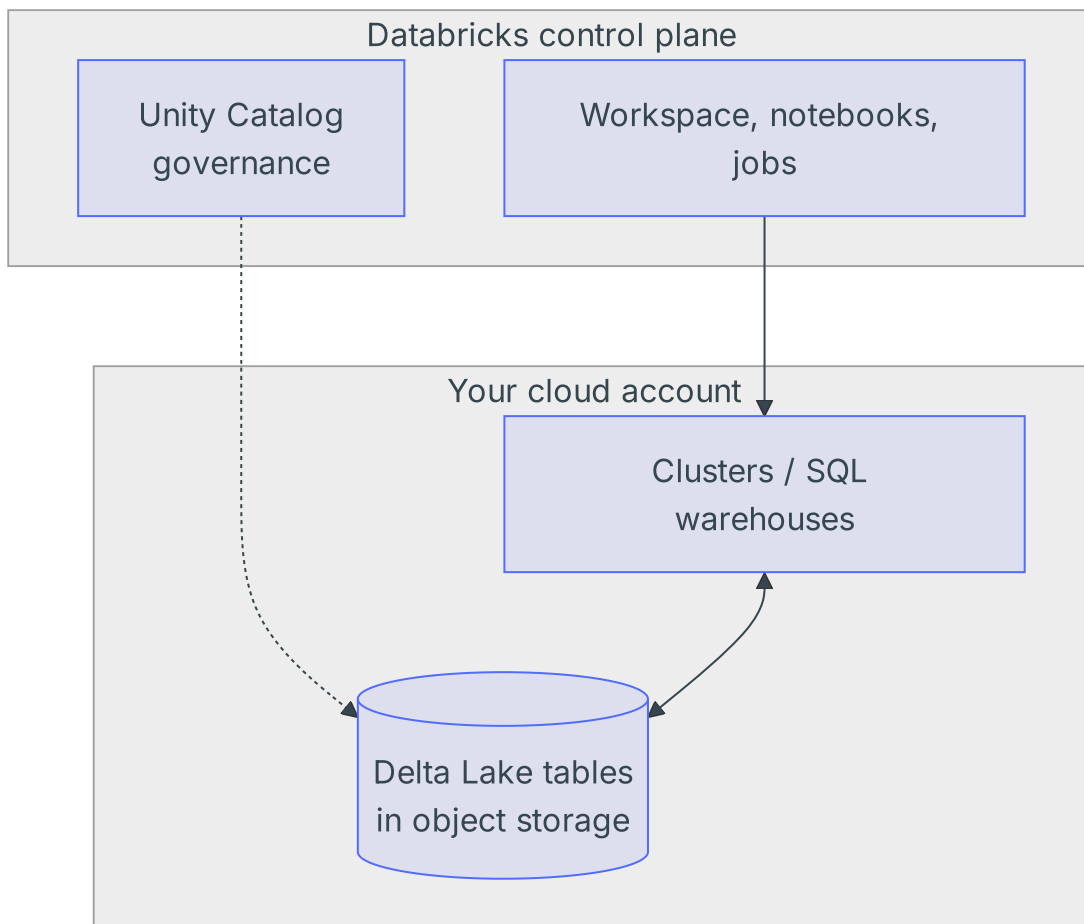
Solution — the lakehouse: a single copy of the data is kept in open file formats in your own cloud storage, with a transaction layer (Delta Lake) that adds warehouse capabilities — ACID transactions, schema enforcement, `MERGE`, and time travel. Databricks builds on this with managed Spark compute, SQL warehouses for BI, notebooks, job orchestration, ML tooling, and a unified governance layer (Unity Catalog).



Delta Lake tables = Parquet files + `_delta_log`
(transactions)

Your cloud storage (S3 / ADLS / GCS)

Summary: Databricks combines managed Spark with a warehouse layer over your own files. Engineers who know PySpark and SQL mainly need to learn its organization (catalogs, jobs, compute) and the Delta-specific operations (`MERGE` , `OPTIMIZE` , `VACUUM` , time travel).



On this page

Basics - [What is Databricks?](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/databricks-reference/#what-is-databricks) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/databricks-reference/#what-is-databricks) - [Workspace & Clusters](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/databricks-reference/#workspace--clusters) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/databricks-reference/#workspace--clusters) - [Notebooks](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/databricks-reference/#notebooks) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/databricks-reference/#notebooks) - [DBFS & Unity Catalog](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/databricks-reference/#dbfs--unity-catalog) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/databricks-reference/#dbfs--unity-catalog) - [Reading & Writing Data](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/databricks-reference/#reading--writing-data) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/databricks-reference/#reading--writing-data)

Intermediate - [Delta Lake](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/databricks-reference/#delta-lake) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/databricks-reference/#delta-lake) - [Delta Table Operations](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/databricks-reference/#delta-table-operations) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/databricks-reference/#delta-table-operations) - [Auto Loader](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/databricks-reference/#auto-loader) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/databricks-reference/#auto-loader) - [Databricks SQL](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/databricks-reference/#databricks-sql) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/databricks-reference/#databricks-sql) - [Databricks Workflows & Jobs](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/databricks-reference/#databricks-workflows--jobs) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/databricks-reference/#databricks-workflows--jobs)

Advanced - [Delta Live Tables \(DLT\)](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/databricks-reference/#delta-live-tables-dlt) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/databricks-reference/#delta-live-tables-dlt) - [Unity Catalog](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/databricks-reference/#unity-catalog) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/databricks-reference/#unity-catalog) - [Open Table Formats — Delta vs Iceberg vs Hudi](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/databricks-reference/#open-table-formats) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/databricks-reference/#open-table-formats) - [Performance Optimization](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/databricks-reference/#performance-optimization) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/databricks-reference/#performance-optimization) - [Databricks in Production](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/databricks-reference/#databricks-in-production) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/databricks-reference/#databricks-in-production)

Reference - [Common Pitfalls](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/databricks-reference/#common-pitfalls) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/databricks-reference/#common-pitfalls) - [Cheat Sheet](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/databricks-reference/#cheat-sheet) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/databricks-reference/#cheat-sheet) - [Interview Questions](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/databricks-reference/#interview-questions) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/databricks-reference/#interview-questions) - [Further Reading](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/databricks-reference/#further-reading) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/databricks-reference/#further-reading)

What is Databricks?

Databricks is a unified analytics platform built on Apache Spark. It provides managed Spark clusters, a collaborative notebook environment, a data lakehouse (Delta Lake), and a full pipeline orchestration layer — all on your cloud (AWS, Azure, GCP).

Databricks Platform

- Compute — managed Spark clusters, SQL warehouses, single-node
- Delta Lake — open table format: ACID transactions on object storage
- Unity Catalog — unified governance: one metastore across all workspaces
- Workflows — orchestrate notebooks, jars, Python scripts, dbt jobs
- Delta Live Tables — declarative pipeline framework
- Databricks SQL — BI-friendly SQL interface on the

lakehouse
└─ MLflow

– experiment tracking, model registry

Databricks vs raw Spark on EMR/Dataproc: - Managed cluster lifecycle (auto-scale, auto-terminate) - Optimized Spark runtime (Photon engine — 2–12x faster than open-source Spark) - Delta Lake built-in - Collaborative notebooks with version control - Unity Catalog for governance

Workspace & Clusters

Cluster types

Type	Use case	Cost
All-purpose	Interactive notebooks, development	Higher (long-running)
Job cluster	Automated pipelines — starts fresh, terminates when done	Lower
SQL Warehouse	Databricks SQL queries, BI tools	Pay per query
Single node	Small datasets, pandas, sklearn	Cheapest

Cluster config essentials

```
# Cluster config (set in UI or via API / Terraform)
{
  "cluster_name": "de-pipeline",
  "spark_version": "14.3.x-scala2.12",
  "node_type_id": "i3.xlarge",           # worker node
  type
  "driver_node_type_id": "i3.xlarge",
  "num_workers": 4,                     # fixed size
  # OR:
  "autoscale": {"min_workers": 2, "max_workers": 10},
  "auto_termination_minutes": 30,      # terminate
  after idle

  "spark_conf": {
```

```

        "spark.sql.shuffle.partitions": "auto",
        "spark.databricks.delta.optimizeWrite.enabled":
"true",
        "spark.databricks.delta.autoCompact.enabled":
"true",
    },

    "init_scripts": [{"dbfs": {"destination":
"dbfs:/init/install_libs.sh"}}}]
}

```

Notebooks

Databricks notebooks run on the cluster and support Python, SQL, Scala, R — and can mix them with `%language` magic commands.

```

# Default language is set per notebook (Python shown here) 

# — Magic commands —————
%python  # switch to Python cell
%sql     # switch to SQL cell
%scala   # switch to Scala cell
%r       # switch to R cell
%sh      # run shell command
%fs      # DBFS filesystem commands
%md      # Markdown cell
%run ./other_notebook  # run another notebook (shares
session state)

# — Display —————
display(df)          # rich table/chart view (Databricks-
specific)
dbutils.fs.ls("/mnt") # list DBFS path

# — Widgets – parameterize notebooks —————
dbutils.widgets.text("date", "2024-03-15", "Processing
Date")
dbutils.widgets.dropdown("env", "dev", ["dev", "staging",
"prod"])

```

```
date = dbutils.widgets.get("date")
env  = dbutils.widgets.get("env")

# — Notebook utilities —————
dbutils.notebook.exit("success")          # exit
with a value
dbutils.notebook.run("./child_notebook", 60, # run
child with timeout
    {"date": "2024-03-15", "env": "prod"})
```

DBFS & Unity Catalog

DBFS (Databricks File System)

DBFS is an abstraction layer over cloud storage (S3/ADLS/GCS) that makes it accessible like a local filesystem.

```
# dbutils.fs – filesystem operations
dbutils.fs.ls("/mnt/datalake/raw/")
dbutils.fs.cp("dbfs:/source/file.csv",
"dbfs:/dest/file.csv")
dbutils.fs.mv("dbfs:/staging/", "dbfs:/archive/")
dbutils.fs.rm("dbfs:/tmp/", recurse=True)
dbutils.fs.mkdirs("dbfs:/mnt/datalake/bronze/orders/")

# Mount cloud storage (legacy – Unity Catalog preferred)
dbutils.fs.mount(
    source="s3a://my-bucket/data",
    mount_point="/mnt/datalake",
    extra_configs={"fs.s3a.access.key": key,
"fs.s3a.secret.key": secret}
)

# Read mounted path
df = spark.read.parquet("/mnt/datalake/raw/orders/")

# Unmount
dbutils.fs.unmount("/mnt/datalake")
```

Secrets — never hardcode credentials

```
# Store secrets in Databricks Secret Scope (backed by Azure Key Vault or Databricks)
# Then retrieve at runtime:
storage_key = dbutils.secrets.get(scope="my-scope",
key="storage-account-key")
db_password = dbutils.secrets.get(scope="prod-secrets",
key="db-password")
```

Reading & Writing Data

```
from pyspark.sql import functions as F

# — Read —————
# Parquet
df = spark.read.parquet("/mnt/datalake/raw/orders/")

# Delta
df =
spark.read.format("delta").load("/mnt/datalake/bronze/orders/")
df = spark.table("catalog.schema.orders") # Unity Catalog table

# CSV with options
df = spark.read \
    .option("header", "true") \
    .option("inferSchema", "true") \
    .csv("/mnt/landing/uploads/customers.csv")

# JSON
df = spark.read.json("/mnt/landing/events/")

# JDBC (external database)
df = spark.read \
    .format("jdbc") \
    .option("url", "jdbc:postgresql://host:5432/db") \
    .option("dbtable", "public.orders") \
```

```

.option("user", "username") \
.option("password", dbutils.secrets.get("db-scope",
"db-pass")) \
.option("numPartitions", 10) \
.option("partitionColumn", "id") \
.option("lowerBound", 1) \
.option("upperBound", 10000000) \
.load()

# — Write —————
# Delta – standard write
df.write \
    .format("delta") \
    .mode("overwrite") \
    .partitionBy("order_date") \
    .save("/mnt/datalake/bronze/orders/")

# Save as managed table (Unity Catalog)
df.write \
    .format("delta") \
    .mode("overwrite") \
    .saveAsTable("catalog.bronze.orders")

# Optimized write – automatically coalesces small files
spark.conf.set("spark.databricks.delta.optimizeWrite.enabled", "true")
df.write.format("delta").mode("append").save("/mnt/datalake/bronze/orders/")

```

Delta Lake

Delta Lake adds ACID transactions, schema enforcement, and time travel on top of Parquet files in object storage. It's the foundation of the Databricks lakehouse.

What Delta Lake provides

Feature	Benefit
ACID transactions	Concurrent reads/writes don't corrupt data

Feature	Benefit
Schema enforcement	Rejects data that doesn't match the table schema
Schema evolution	Safely add/rename columns with <code>mergeSchema</code>
Time travel	Query any previous version of the table
Audit log	Full history of every operation
Upsert (MERGE)	Row-level updates without full rewrites
Streaming + batch	Same table supports both read/write patterns
Auto-optimize	Small file compaction + data skipping indexes

Delta table internals

```

/mnt/datalake/bronze/orders/
├── _delta_log/                    # transaction log
│   ├── 00000000000000000000000000000000.json # version 0 – table
│   │   created
│   │   ├── 00000000000000000000000000000001.json # version 1 – data
│   │   │   added
│   │   │   ├── 00000000000000000000000000000002.json # version 2 – update
│   │   │   └── ...
│   └── part-00000-abc.snappy.parquet
│   └── part-00001-def.snappy.parquet
│   └── ...

```

Every write appends a new entry to the `_delta_log`. Reads check the log to know which files are valid (no full scan needed to find current state).

Delta Table Operations

```

from delta.tables import DeltaTable

# — Table history —————
spark.sql("DESCRIBE HISTORY
delta.`/mnt/datalake/bronze/orders`")

```

```

# or
dt = DeltaTable.forPath(spark,
"/mnt/dataLake/bronze/orders")
display(dt.history())

# — Time travel —————
# By version
df_v3 = spark.read.format("delta") \
    .option("versionAsOf", 3) \
    .load("/mnt/dataLake/bronze/orders")

# By timestamp
df_yesterday = spark.read.format("delta") \
    .option("timestampAsOf", "2024-03-14") \
    .load("/mnt/dataLake/bronze/orders")

# — MERGE (upsert) —————
target = DeltaTable.forPath(spark,
"/mnt/dataLake/silver/orders")

target.alias("t").merge(
    source_df.alias("s"),
    "t.order_id = s.order_id"
) \
    .whenMatchedUpdate(set={
        "status": "s.status",
        "updated_at": "s.updated_at"
    }) \
    .whenNotMatchedInsertAll() \
    .execute()

# — DELETE —————
dt.delete("order_date < '2020-01-01'")
spark.sql("DELETE FROM delta.`/mnt/dataLake/bronze/orders`
WHERE status = 'test'")

# — UPDATE —————
dt.update(
    condition="status = 'SHIPPED'",
    set={"status": "'shipped'"} # normalize case
)

```

```

# — Schema evolution —————
df_with_new_col.write \
    .format("delta") \
    .mode("append") \
    .option("mergeSchema", "true") \
    .save("/mnt/datalake/bronze/orders")    # mergeSchema
adds new columns to the table schema

# — OPTIMIZE – compact small files —————
spark.sql("OPTIMIZE delta.`/mnt/datalake/bronze/orders`")
spark.sql("OPTIMIZE delta.`/mnt/datalake/bronze/orders`
WHERE order_date = '2024-03-15'")

# — ZORDER – co-locate related data for faster queries —
spark.sql("""
    OPTIMIZE delta.`/mnt/datalake/silver/events`
    ZORDER BY (user_id, event_date)
""")

# — VACUUM – remove old files no longer referenced —
spark.sql("VACUUM delta.`/mnt/datalake/bronze/orders`
RETAIN 168 HOURS")
# Default retention: 7 days. Don't go below 7 days – breaks
time travel.

# — Restore —————
spark.sql("RESTORE TABLE my_table TO VERSION AS OF 5")
spark.sql("RESTORE TABLE my_table TO TIMESTAMP AS OF '2024-
03-14'")

```

Auto Loader

Auto Loader (`cloudFiles`) incrementally ingests new files from cloud storage into Delta Lake. It tracks which files have been processed so you never reprocess or miss a file.

```

# Streaming Auto Loader – processes new files as they
arrive
checkpoint_path = "/mnt/checkpoints/orders-autoloader"
target_path      = "/mnt/datalake/bronze/orders"

```

```

df_stream = spark.readStream \
    .format("cloudFiles") \
    .option("cloudFiles.format", "parquet") \
    .option("cloudFiles.schemaLocation", checkpoint_path +
"/schema") \
    .option("cloudFiles.inferColumnTypes", "true") \
    .load("s3://my-bucket/landing/orders/")

# Add ingestion metadata
df_stream = df_stream \
    .withColumn("_ingested_at", F.current_timestamp()) \
    .withColumn("_source_file",
F.col("_metadata.file_path"))

# Write to Delta
df_stream.writeStream \
    .format("delta") \
    .outputMode("append") \
    .option("checkpointLocation", checkpoint_path) \
    .trigger(availableNow=True) \
    .start(target_path)          # availableNow: process
all pending files, then stop

# Trigger options:
# .trigger(availableNow=True)          - batch: process all
new files, stop
# .trigger(processingTime="10 minutes") - micro-batch every
10 min
# .trigger(once=True)                  - legacy; use
availableNow instead

```

Why Auto Loader over plain readStream? - Scales to millions of files (uses file notifications, not directory listing) - Schema inference + evolution built-in - Exactly-once delivery via checkpointing - No custom file-tracking state to maintain

Databricks SQL

Databricks SQL provides a BI-friendly SQL editor and serverless SQL warehouses for querying Delta tables.



```

-- Use Unity Catalog 3-part naming
SELECT * FROM catalog_name.schema_name.table_name;
USE CATALOG my_catalog;
USE SCHEMA bronze;

-- Delta-specific SQL
DESCRIBE HISTORY orders;
DESCRIBE DETAIL orders;

-- Time travel in SQL
SELECT * FROM orders VERSION AS OF 5;
SELECT * FROM orders TIMESTAMP AS OF '2024-03-14';

-- OPTIMIZE and VACUUM
OPTIMIZE orders ZORDER BY (customer_id);
VACUUM orders RETAIN 168 HOURS;

-- Create a live view (refreshed on each query)
CREATE OR REPLACE VIEW gold.daily_revenue AS
SELECT DATE(created_at) AS order_date, SUM(amount) AS
revenue
FROM silver.orders
GROUP BY 1;

-- Dynamic views for row-level security — map users to
regions in a table
CREATE VIEW finance_orders AS
SELECT o.*
FROM orders o
WHERE o.region IN (SELECT region FROM
security.user_regions
WHERE user_email = current_user());

```

Databricks Workflows & Jobs

Workflows orchestrate multi-task pipelines — notebooks, Python scripts, dbt commands, SQL statements, and Spark JARs.

Job definition (API / Terraform), scheduled daily at 02:00 UTC:

```

{
  "name": "orders_daily_pipeline",
  "schedule": {
    "quartz_cron_expression": "0 0 2 * * ?",
    "timezone_id": "UTC",
    "pause_status": "UNPAUSED"
  },
  "tasks": [
    {
      "task_key": "extract",
      "notebook_task": {
        "notebook_path": "/Repos/de-
team/pipelines/extract_orders",
        "base_parameters": {"date": "
{{job.start_time.iso_date}}"}
      },
      "job_cluster_key": "pipeline_cluster"
    },
    {
      "task_key": "transform",
      "depends_on": [{"task_key": "extract"}],
      "notebook_task": {
        "notebook_path": "/Repos/de-
team/pipelines/transform_orders"
      },
      "job_cluster_key": "pipeline_cluster"
    },
    {
      "task_key": "dbt_models",
      "depends_on": [{"task_key": "transform"}],
      "dbt_task": {
        "project_directory": "/Repos/de-team/dbt-project",
        "commands": ["dbt run --select orders+", "dbt test
--select orders+"]
      }
    }
  ],
  "job_clusters": [{
    "job_cluster_key": "pipeline_cluster",
    "new_cluster": {
      "spark_version": "14.3.x-scala2.12",
      "node_type_id": "i3.xlarge",

```

```

        "num_workers": 4,
        "auto_termination_minutes": 30
    }
}],
    "email_notifications": {
        "on_failure": ["data-team@example.com"]
    }
}

```

Delta Live Tables (DLT)

Naming note: in 2025 Databricks renamed DLT to **Lakeflow Declarative Pipelines** (and Workflows to **Lakeflow Jobs**). The open-source equivalent is Spark Declarative Pipelines (from `pyspark import pipelines as dp`). The `import dlt` API below still works.

DLT is a declarative pipeline framework. You define **what** the data should look like (using `@dlt.table` decorators), and Databricks manages the execution order, retries, and data quality.

```

import dlt
from pyspark.sql import functions as F

# Bronze – raw ingestion
@dlt.table(
    name="raw_orders",
    comment="Raw orders from landing zone",
    table_properties={"quality": "bronze"}
)
def raw_orders():
    return (
        spark.readStream
            .format("cloudFiles")
            .option("cloudFiles.format", "parquet")
            .load("s3://my-bucket/landing/orders/")
    )

# Silver – cleaned and validated
@dlt.table(
    name="orders",
    comment="Cleaned orders",

```

```

        table_properties={"quality": "silver"}
    )
    @dlt.expect("valid_amount", "amount > 0")
    @dlt.expect_or_drop("not_null_order_id", "order_id IS NOT
    NULL")
    @dlt.expect_or_fail("valid_status", "status IN
    ('placed','shipped','delivered','cancelled')")
    def orders():
        return (
            dlt.read_stream("raw_orders")
                .with_column("amount",
            F.col("amount").cast("double"))
                .with_column("order_date",
            F.to_date("created_at"))
                .where("order_id IS NOT NULL")
        )

# Gold – aggregated
@dlt.table(
    name="daily_revenue",
    comment="Daily revenue aggregation"
)
def daily_revenue():
    return (
        dlt.read("orders")
            .group_by("order_date")
            .agg(
                F.count("*").alias("order_count"),
                F.sum("amount").alias("revenue")
            )
    )

```

DLT expectations

Decorator	On violation
<code>@dlt.expect</code>	Log violation in quality metrics; keep row
<code>@dlt.expect_or_drop</code>	Drop the violating row
<code>@dlt.expect_or_fail</code>	Fail the entire pipeline

Unity Catalog

Unity Catalog is Databricks' unified governance layer — one metastore for all workspaces, clouds, and data assets.

Unity Catalog hierarchy:

Metastore (one per region)

- └─ Catalog ← like a database in Snowflake
 - └─ Schema ← like a schema
 - └─ Table / View / Function / Volume

Example: `my_catalog.bronze.orders`

```
-- Create catalog and schema
CREATE CATALOG IF NOT EXISTS analytics;
CREATE SCHEMA IF NOT EXISTS analytics.bronze;
CREATE SCHEMA IF NOT EXISTS analytics.silver;
CREATE SCHEMA IF NOT EXISTS analytics.gold;

-- Create a managed Delta table
CREATE TABLE analytics.bronze.orders (
  order_id    STRING NOT NULL,
  customer_id BIGINT,
  amount      DOUBLE,
  created_at  TIMESTAMP
)
USING DELTA
CLUSTER BY (created_at); -- liquid clustering; Delta
can't partition by an expression like DATE(created_at)

-- Create an external table (data stays in your storage)
CREATE TABLE analytics.bronze.events
USING DELTA
LOCATION 's3://my-bucket/bronze/events/';

-- Grant permissions (USE CATALOG + USE SCHEMA are needed
before any table access)
GRANT USE CATALOG ON CATALOG analytics TO `data-
engineers`;
GRANT USE CATALOG ON CATALOG analytics TO
```

```

`analysts`;
GRANT USE SCHEMA, SELECT ON SCHEMA analytics.gold TO
`analysts`;
GRANT MODIFY ON TABLE analytics.bronze.orders TO `data-
engineers`;
GRANT ALL PRIVILEGES ON SCHEMA analytics.bronze TO `data-
engineers`;

-- Column-level masking (dynamic data masking)
CREATE FUNCTION mask_email(email STRING)
RETURNS STRING
RETURN CASE
    WHEN is_account_group_member('data-engineers') THEN
email
    ELSE CONCAT(LEFT(email, 2), '***@***.com')
END;

ALTER TABLE analytics.silver.customers
    ALTER COLUMN email SET MASK mask_email;

-- Row-level security
CREATE FUNCTION region_filter(region STRING)
RETURNS BOOLEAN
RETURN is_account_group_member(CONCAT('region-', region));

ALTER TABLE analytics.silver.orders
    ADD ROW FILTER region_filter ON (region);

```

Open Table Formats

Delta Lake, Apache Iceberg, and Apache Hudi solve the same problem — ACID transactions on object storage — but with different trade-offs.

	Delta Lake	Apache Iceberg	Apache Hudi
Origin	Databricks	Netflix	Uber
Best engine	Spark / Databricks	Multi-engine (Spark, Flink, Trino, BigQuery)	Spark

	Delta Lake	Apache Iceberg	Apache Hudi
Upsert performance	Good	Good	Best (optimized for <u>CDC</u>)
Schema evolution	Good	Excellent	Good
Time travel	Yes	Yes	Yes
Partition evolution	Limited	Excellent — hidden partitioning	Limited
Streaming	Good	Good	Excellent
Cloud native	AWS, Azure, GCP	All	All
When to use	All-in Databricks stack	Multi-engine, vendor-neutral	Heavy <u>CDC</u> / upsert workloads

In practice: Delta if your stack is Databricks-first. Iceberg if you need multi-engine access (Trino, BigQuery, Athena, Flink all reading the same tables). Hudi for CDC-heavy pipelines with high upsert volume.

Performance Optimization

```
# — Photon Engine (Databricks-specific) —————
# Enable on cluster config – automatically accelerates SQL
and DataFrame ops
# No code changes needed; 2-12x faster than open-source
Spark for many workloads

# — Adaptive Query Execution (AQE) —————
spark.conf.set("spark.sql.adaptive.enabled", "true")
# default in DBR 10+
spark.conf.set("spark.sql.adaptive.coalescePartitions.enabled", "true")
spark.conf.set("spark.sql.adaptive.skewJoin.enabled", "true") # handles data skew

# — Delta optimizations —————
# Optimize writes – coalesces small files during write
```

```

spark.conf.set("spark.databricks.delta.optimizeWrite.enabled", "true")

# Auto compact – triggers OPTIMIZE automatically after writes
spark.conf.set("spark.databricks.delta.autoCompact.enabled", "true")

# Data skipping – bloom filter indexes for high-cardinality columns
spark.sql("""
    CREATE BLOOMFILTER INDEX ON TABLE orders
    FOR COLUMNS (order_id OPTIONS (fpp=0.1, numItems=500000000))
""")

# — Caching —————
# Delta cache – caches remote files on local SSD (different from Spark cache)
spark.conf.set("spark.databricks.io.cache.enabled", "true")

# Spark cache – keeps DataFrame in memory/disk across actions
df.cache()
df.count() # materialize
# ... multiple operations on df ...
df.unpersist()

# — Partition hints —————
# Use broadcast for small tables
from pyspark.sql.functions import broadcast
result = large_df.join(broadcast(small_df), "key")

# Repartition before write
df.repartition(8, "order_date") \
    .write.format("delta").mode("append").save(path)

```

Databricks in Production

Repos — Git integration

Databricks Repos syncs notebooks directly with Git (GitHub, GitLab, ADO).

One repo per environment (dev branch → dev workspace, main → prod workspace).

Typical workflow:

1. Develop in feature branch → dev workspace
2. PR → code review
3. Merge to main → CI runs tests (pytest, dbt test)
4. CD deploys job definitions to prod workspace

CI/CD with Databricks Asset Bundles (DAB)

```
# databricks.yml – defines all assets (jobs, pipelines, clusters)
```

```
bundle:
```

```
  name: orders_pipeline
```

```
targets:
```

```
  dev:
```

```
    mode: development
```

```
    default: true
```

```
    workspace:
```

```
      host: https://adb-xxx.azuredatabricks.net
```

```
  prod:
```

```
    mode: production
```

```
    workspace:
```

```
      host: https://adb-yyy.azuredatabricks.net
```

```
resources:
```

```
  jobs:
```

```
    orders_daily:
```

```
      name: "orders_daily_pipeline"
```

```
      schedule:
```

```
        quartz_cron_expression: "0 0 2 * * ?"
```

```
        timezone_id: "UTC"
```

```
      tasks:
```

```
        - task_key: extract
```

```
notebook_task:
  notebook_path: ./notebooks/extract_orders
```

```
databricks bundle validate      # check config
databricks bundle deploy        # deploy to target workspace
databricks bundle run orders_daily
```

Monitoring

```
# Structured logging from notebooks / jobs
import logging
logger = logging.getLogger(__name__)

# Record metrics to a Delta table for monitoring
def log_pipeline_run(pipeline_name, records_processed,
                    duration_sec, status):
    metrics = [(pipeline_name, records_processed,
                  duration_sec,
                  status, datetime.utcnow())]
    df = spark.createDataFrame(metrics,
                               ["pipeline", "records", "duration_sec", "status",
                                "run_at"])
    df.write.format("delta").mode("append") \
        .saveAsTable("analytics.ops.pipeline_metrics")
```

Common Pitfalls

Pitfall	Symptom	Fix
Running scheduled jobs on all-purpose clusters	Bills 2–3× higher than necessary	Job clusters or serverless jobs for production; all-purpose clusters only for development
No auto-termination on interactive clusters	Idle clusters running all weekend	<code>auto_termination_minutes</code> (e.g. 30) plus cluster policies that enforce it

Pitfall	Symptom	Fix
Still using DBFS mounts with access keys	Credentials shared by everyone on the workspace; no fine-grained access	Unity Catalog external locations and storage credentials; volumes for files
Over-partitioning Delta tables	Many small files, slow queries	Don't partition tables under ~1 TB; use liquid clustering (<code>CLUSTER BY</code>) instead
Never running <code>OPTIMIZE</code> / <code>VACUUM</code>	Small files pile up; storage grows forever	Predictive optimization, or scheduled <code>OPTIMIZE</code> + <code>VACUUM</code>
<code>VACUUM ... RETAIN 0 HOURS</code>	Time travel gone; concurrent readers and streams fail	Keep the default 7 days unless you understand the consequences
<code>MERGE</code> with duplicate keys in the source	Cannot perform Merge as multiple source rows matched...	Deduplicate the source on the merge key first
Business logic spread across <code>%run</code> notebook chains	Hard to test, review, and reuse	Python modules/wheels in Git folders, imported into thin notebooks; unit tests in CI
Hardcoded paths, workspace URLs, and secrets	Can't promote from dev to prod	Asset Bundles with per-target variables; <code>dbutils.secrets</code>
One giant cluster for every workload	Streaming, ETL, and ad hoc queries fight for resources	Separate jobs and compute per workload; SQL warehouses for BI
<code>inferSchema</code> / schema inference in Bronze without evolution rules	Jobs break or silently add junk columns	Auto Loader with <code>schemaLocation</code> , schema hints, and a chosen <code>schemaEvolutionMode</code>

Cheat Sheet

Task	Code
Read a UC table	<code>spark.table("catalog.schema.table")</code>

Task	Code
Write a managed table	<code>df.write.mode("overwrite").saveAsTable("cat.sch.t")</code>
Upsert	<code>MERGE INTO t USING s ON t.id = s.id WHEN MATCHED THEN UPDATE SET * WHEN NOT MATCHED THEN INSERT *</code>
Overwrite one slice	<code>df.write.mode("overwrite").option("replaceWhere", "dt = '2024-03-15']").saveAsTable("t")</code>
Table history / time travel	<code>DESCRIBE HISTORY t · SELECT * FROM t VERSION AS OF 5</code>
Undo a bad write	<code>RESTORE TABLE t TO VERSION AS OF 5</code>
Compact / cluster	<code>OPTIMIZE t · ALTER TABLE t CLUSTER BY (col)</code>
Clean old files	<code>VACUUM t</code> (default retention 7 days)
Table details	<code>DESCRIBE DETAIL t</code> (location, size, number of files)
Incremental file ingest	<code>spark.readStream.format("cloudFiles").option("cloudFiles.format", "json")...</code>
Run as batch	<code>.trigger(availableNow=True)</code>
Secrets	<code>dbutils.secrets.get(scope="s", key="k")</code>
Notebook parameters	<code>dbutils.widgets.text("date", "") → dbutils.widgets.get("date")</code>
Files in a volume	<code>/Volumes/catalog/schema/volume/path/file.csv</code>
Deploy with bundles	<code>databricks bundle validate → deploy -t prod → run job_name</code>
Grant read access	<code>GRANT USE CATALOG ON CATALOG c TO g; GRANT USE SCHEMA, SELECT ON SCHEMA c.s TO g</code>

Compute choice: development → all-purpose cluster (auto-terminate) · scheduled ETL → job cluster / serverless jobs · BI and SQL → SQL warehouse (serverless) · small Python work → single-node cluster

Delta table layout: under ~1 TB → no partitions · large tables → liquid clustering on common filter columns · Z-ORDER only on older runtimes

Interview Questions

Q: What is Delta Lake and how does it provide ACID transactions on object storage? A:

Delta Lake is an open table format made of Parquet data files plus a transaction log (`_delta_log/`) of JSON commits and periodic Parquet checkpoints. Each write adds new data files, then atomically writes the next numbered log entry that lists added and removed files. Readers reconstruct the current table state from the log, so they only ever see committed versions. Concurrent writers use optimistic concurrency: if two try to write the same log version, one wins and the other re-checks for conflicts and retries or fails.

Q: What is Auto Loader and why use it instead of a plain file read? A: Auto Loader (`cloudFiles`) is a Structured Streaming source that incrementally discovers new files in cloud storage and processes each exactly once, tracking progress in a checkpoint. It scales to millions of files using directory listing or cloud file notifications, infers and evolves schemas (with a rescued-data column for unexpected fields), and can run continuously or as a batch with `availableNow`. It replaces hand-written "which files have I already loaded?" logic.

Q: What does `OPTIMIZE` do, and what is Z-ordering vs liquid clustering? A: `OPTIMIZE` compacts many small files into fewer large ones. Z-ordering additionally sorts data within files by multiple columns so file-level min/max statistics let queries skip more files — but it has to be re-run and rewrites data each time. Liquid clustering is the newer replacement: you declare clustering columns with `CLUSTER BY`, can change them without rewriting the table, and clustering is applied incrementally. It also replaces most uses of partitioning.

Q: What is Unity Catalog? A: Databricks' governance layer: a single metastore per region shared across workspaces, with a three-level namespace (`catalog.schema.table`). It centralizes permissions (grants on catalogs, schemas, tables, volumes, functions, models), row filters and column masks, automatic lineage, audit logs, and managed access to cloud storage through storage credentials and external locations — replacing per-workspace Hive metastores and mounts.

Q: What's the difference between a managed and an external table? A: For a managed table, Unity Catalog controls both the metadata and the storage location; dropping the table deletes the data (after a retention period), and features like predictive optimization work automatically. For an external table, you specify a `LOCATION` you manage; dropping the table removes only the metadata. Use managed by default, and external when other tools must own or directly access the files.

Q: How would you deploy Databricks pipelines across dev and prod? A: Keep code in Git and define jobs, pipelines, and their compute in a Databricks Asset Bundle (`databricks.yml`) with a target per environment. CI runs unit tests and `databricks bundle validate`, then deploys to a dev or staging workspace for integration tests; merging to `main` deploys to production, running as a service principal. Environment-specific values (catalog names, paths) come from bundle variables, so the same code runs everywhere.

Q: Delta Live Tables / Lakeflow Declarative Pipelines vs regular jobs — when would you use each? A: Declarative pipelines suit a medallion flow of tables: you declare each table as a query, and the framework works out dependencies, incremental processing, retries, and data quality expectations. Plain jobs (notebooks or Python scripts orchestrated by Lakeflow Jobs) give full control, which suits complex custom logic, external API calls, or non-table outputs. Many teams use pipelines for Bronze → Silver → Gold and jobs for everything around them.

Further Reading

- [Databricks documentation](https://docs.databricks.com/) (https://docs.databricks.com/)
 - [Delta Lake documentation](https://docs.delta.io/) (https://docs.delta.io/) — the open-source format, usable outside Databricks
 - [Unity Catalog best practices](https://docs.databricks.com/en/data-governance/unity-catalog/best-practices.html) (https://docs.databricks.com/en/data-governance/unity-catalog/best-practices.html)
 - [Liquid clustering](https://docs.databricks.com/en/delta/clustering.html) (https://docs.databricks.com/en/delta/clustering.html)
 - [Databricks Asset Bundles](https://docs.databricks.com/en/dev-tools/bundles/index.html) (https://docs.databricks.com/en/dev-tools/bundles/index.html)
 - *Delta Lake: The Definitive Guide* — Denny Lee, Tristen Wentling, Scott Haines & Prashanth Babu (O'Reilly)
-

Previous: [PySpark](https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/pyspark-reference/) (https://sarangambekar1997.github.io/data-engineering-handbook/02-processing/pyspark-reference/) • **Next:** [Apache Iceberg](https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/apache-iceberg/) (https://sarangambekar1997.github.io/data-engineering-handbook/01-storage/apache-iceberg/) • **Back to:** [Index](https://sarangambekar1997.github.io/data-engineering-handbook/) (https://sarangambekar1997.github.io/data-engineering-handbook/)

 September 29, 2026