Databricks Complete Guide

Zero to Hero - Everything You Need to Know

What is Databricks?

Databricks is a unified data analytics platform built on Apache Spark. It combines data engineering, data science, and business analytics into one platform. Think of it as Spark + managed infrastructure + collaboration tools + Delta Lake.

Why Databricks Matters

- Unified Platform: ETL, analytics, ML in one place - no tool sprawl
- Managed Spark: No more cluster headaches, auto-scaling, spot instances
- Delta Lake: ACID transactions on data lakes (game changer)
- Performance: Photon engine is 2-8x faster than vanilla Spark
- Collaboration: Notebooks, Git integration, shared workspaces

Prerequisites for This Guide

- Basic SQL knowledge (SELECT, JOIN, GROUP BY)
- Python fundamentals (functions, loops, data types)
- Understanding of data concepts (tables, schemas, ETL)

Learning Timeline

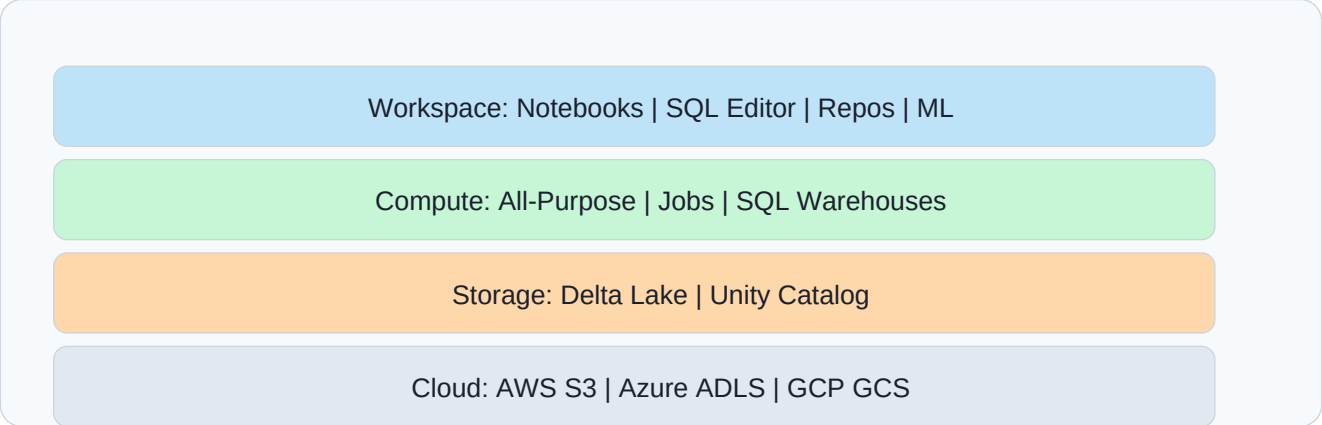
Phase	Topics	Duration
Foundation	Workspace, Clusters, Notebooks	Week 1-2
Core	DBFS, Spark basics, DataFrames	Week 3-4
Delta Lake	ACID, Time Travel, MERGE	Week 5-7
Unity Catalog	Governance, Security	Week 8-9
Architecture	Medallion Pattern	Week 10-11
Ingestion	Auto Loader, Streaming	Week 12-13
Production	Workflows, Jobs, CI/CD	Week 14-16

Platform Architecture

Understanding the Databricks Stack

Architecture Overview

Databricks runs on top of your cloud provider (AWS, Azure, or GCP). It manages Spark clusters and provides a unified interface for all data work.



Key Components Explained

Workspace:	Your home base - contains notebooks, files, repos, experiments
Clusters:	Compute resources - driver + workers that run your code
Delta Lake:	Storage layer with ACID transactions and time travel
Unity Catalog:	Governance layer - permissions, lineage, audit
DBFS:	Databricks File System - abstraction over cloud storage
Workflows:	Job orchestration - schedule and monitor pipelines

Workspace & Navigation

Getting Around Databricks

Workspace Structure

The workspace is organized like a file system. You can create folders, notebooks, and files. Everything is collaborative by default.

Item	Description
Notebooks	Interactive documents with code, text, a
Repos	Git integration - clone, commit, push, p
Files	Upload data files, libraries, configurat
Workflows	Define and schedule jobs
SQL Editor	Write and run SQL queries
Compute	Manage clusters and SQL warehouses

Notebook Basics

Notebooks are where you write and run code. They support multiple languages in the same notebook using magic commands.

```
# Default language (set at notebook level)
df = spark.read.csv('/data/file.csv')

%sql
-- Switch to SQL
SELECT * FROM my_table LIMIT 10

%python
# Back to Python
print('Hello Databricks')

%md
## This is Markdown
Use for documentation
```

Magic Commands Reference

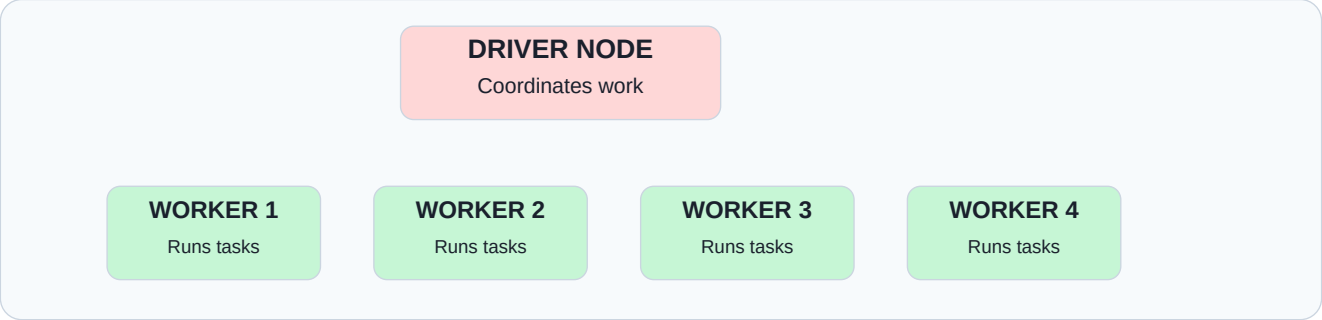
Command	Purpose	Example
%python	Run Python code	%python print('hi')
%sql	Run SQL queries	%sql SELECT * FROM t
%scala	Run Scala code	%scala val x = 1
%r	Run R code	%r print('hello')
%md	Render Markdown	%md # Title
%sh	Run shell commands	%sh ls -la
%fs	DBFS commands	%fs ls /data/
%run	Run another notebook	%run ./other_notebook

Clusters Deep Dive

Compute Resources in Databricks

What is a Cluster?

A cluster is a set of computers that work together to run your Spark code. It has one driver node (coordinator) and multiple worker nodes (executors).



Cluster Types

Type	Best For	Lifecycle	Cost
All-Purpose	Interactive development, notebooks	Stays running	$$$ (higher)
Job Clusters	Automated pipelines, scheduled jobs	Auto-terminates	$ (lower)
SQL Warehouses	SQL analytics, BI tools	Serverless option	$$ (medium)

Cluster Configuration

- Node Type: Choose based on workload (memory-optimized, compute-optimized)
- Workers: Start with 2-4, enable autoscaling (min=1, max=8)
- Spot Instances: Use for workers to save 60-80% cost
- Auto-termination: Set to 30-60 min for dev clusters
- Runtime: Use latest LTS (Long Term Support) version

Cost Saving Tip

For development: Use single-node cluster with auto-termination. For production jobs: Use job clusters with spot instances. This can reduce costs by 70-80%.

DBFS & File Operations

Working with Files in Databricks

Understanding DBFS

DBFS (Databricks File System) is an abstraction layer over cloud storage. It makes S3/ADLS/GCS feel like a local file system. All paths start with /.

```
# List files in DBFS
dbutils.fs.ls('/mnt/data/')

# Copy file
dbutils.fs.cp('/source/file.csv', '/dest/file.csv')

# Move file
dbutils.fs.mv('/old/path', '/new/path')

# Delete file
dbutils.fs.rm('/path/to/file', recurse=True)

# Read file content
dbutils.fs.head('/path/to/file.txt', 1000)
```

Key DBFS Paths

Path	Purpose
/FileStore/	Upload files via UI, accessible via web
/mnt/	Mounted cloud storage (S3, ADLS, GCS)
/tmp/	Temporary storage, cleared periodically
/user/hive/warehouse/	Default location for managed tables
/databricks-datasets/	Sample datasets for learning

Mounting Cloud Storage

```
# Mount S3 bucket (AWS)
dbutils.fs.mount(
  source = 's3a://my-bucket/path',
  mount_point = '/mnt/my-data',
  extra_configs = {
    'fs.s3a.access.key': dbutils.secrets.get('scope', 'key'),
    'fs.s3a.secret.key': dbutils.secrets.get('scope', 'secret')
  }
)

# Now access files like local
df = spark.read.csv('/mnt/my-data/file.csv')
```

Spark Fundamentals

Core Concepts You Must Know

SparkSession

SparkSession is your entry point to Spark. In Databricks, it's pre-configured as 'spark'. You don't need to create it.

```
# spark is already available
spark  # <-- This is your SparkSession

# Check Spark version
spark.version

# Access configuration
spark.conf.get('spark.executor.memory')
```

DataFrames - The Core Abstraction

DataFrames are distributed collections of data organized into rows and columns. Think of them as distributed tables.

```
# Create DataFrame from list
data = [('Alice', 25), ('Bob', 30)]
df = spark.createDataFrame(data, ['name', 'age'])

# Read from file
df = spark.read.csv('/path/file.csv', header=True, inferSchema=True)
df = spark.read.json('/path/file.json')
df = spark.read.parquet('/path/file.parquet')

# Read from table
df = spark.table('database.table_name')
```

Essential DataFrame Operations

```
# View data
df.show(5)           # Show 5 rows
df.display()         # Databricks rich display
df.printSchema()     # Show column types

# Select columns
df.select('name', 'age')
df.select(df.name, df.age + 1)

# Filter rows
df.filter(df.age > 25)
df.where('age > 25')  # Same thing

# Add/rename columns
df.withColumn('age_next', df.age + 1)
df.withColumnRenamed('name', 'full_name')
```

Spark Operations Continued

Transformations & Actions

Aggregations

```
from pyspark.sql.functions import *

# Basic aggregations
df.groupBy('department').count()
df.groupBy('department').agg(
    count('*').alias('total'),
    avg('salary').alias('avg_salary'),
    max('salary').alias('max_salary')
)

# Multiple groupings
df.groupBy('department', 'year').sum('revenue')
```

Joins

```
# Inner join (default)
df1.join(df2, df1.id == df2.id)

# Left join
df1.join(df2, df1.id == df2.id, 'left')

# Join types: inner, left, right, outer, semi, anti
df1.join(df2, 'common_column', 'left')  # Simpler syntax

# Broadcast join (for small tables - IMPORTANT!)
from pyspark.sql.functions import broadcast
df_large.join(broadcast(df_small), 'key')
```

Important Functions

```
from pyspark.sql.functions import *

# String functions
df.select(upper('name'), lower('name'), trim('name'))
df.select(concat('first', lit(' '), 'last'))

# Date functions
df.select(current_date(), current_timestamp())
df.select(year('date'), month('date'), dayofweek('date'))
df.select(datediff('end_date', 'start_date'))

# Null handling
df.select(coalesce('col1', 'col2', lit('default')))
df.na.fill({'col1': 0, 'col2': 'unknown'})
df.na.drop()  # Drop rows with any null
```

Window Functions

```
from pyspark.sql.window import Window

# Define window
window = Window.partitionBy('dept').orderBy('salary')

# Ranking
df.withColumn('rank', rank().over(window))
df.withColumn('row_num', row_number().over(window))

# Running totals
df.withColumn('running_total', sum('amount').over(window))
```

Delta Lake Fundamentals

The Heart of Modern Databricks

What is Delta Lake?

Delta Lake is an open-source storage layer that brings ACID transactions to data lakes. It's what makes Databricks reliable for production workloads.

Key Features

- **ACID Transactions** — No more partial writes or corrupted data
- **Time Travel** — Query any previous version of your data
- **Schema Enforcement** — Prevent bad data from entering tables
- **Schema Evolution** — Add columns without breaking pipelines
- **Audit History** — Track all changes with DESCRIBE HISTORY
- **Unified Batch/Stream** — Same table for batch and streaming

Delta Table Structure

A Delta table consists of Parquet data files + a transaction log (_delta_log). The log tracks every change.

Creating Delta Tables

```
# Method 1: Write DataFrame as Delta
df.write.format('delta').save('/path/to/table')

# Method 2: Create managed table
df.write.format('delta').saveAsTable('my_database.my_table')

# Method 3: SQL
# CREATE TABLE my_table USING DELTA AS SELECT * FROM source

# Method 4: Convert existing Parquet
# CONVERT TO DELTA parquet.`/path/to/parquet`
```

Delta Lake Operations

CRUD and More

Basic CRUD Operations

```
# INSERT
df.write.format('delta').mode('append').saveAsTable('my_table')

# UPDATE (SQL)
# UPDATE my_table SET status = 'active' WHERE id = 1

# DELETE (SQL)
# DELETE FROM my_table WHERE created_at < '2023-01-01'

# UPSERT using MERGE (most important pattern!)
# See next section
```

MERGE - The Most Important Pattern

MERGE handles insert, update, and delete in one atomic operation. Essential for CDC (Change Data Capture) and SCD (Slowly Changing Dimensions).

```
MERGE INTO target_table AS target
USING source_table AS source
ON target.id = source.id

WHEN MATCHED AND source.is_deleted = true THEN
  DELETE

WHEN MATCHED THEN
  UPDATE SET *

WHEN NOT MATCHED THEN
  INSERT *
```

Time Travel

Query any previous version of your data. Useful for debugging, auditing, and recovering from mistakes.

```
# Query by version number
SELECT * FROM my_table VERSION AS OF 5

# Query by timestamp
SELECT * FROM my_table TIMESTAMP AS OF '2024-01-15 10:00:00'

# In PySpark
spark.read.format('delta').option('versionAsOf', 5).load('/path')

# View history
DESCRIBE HISTORY my_table

# Restore to previous version
RESTORE TABLE my_table TO VERSION AS OF 5
```

Delta Lake Optimization

Performance Tuning

OPTIMIZE Command

OPTIMIZE compacts small files into larger ones. Small files = slow queries. Run OPTIMIZE regularly on frequently updated tables.

```
# Basic optimize
OPTIMIZE my_table

# Optimize with Z-ORDER (co-locate related data)
OPTIMIZE my_table ZORDER BY (date, region)

# Z-ORDER improves query performance for filtered columns
# Choose columns you frequently filter on
```

VACUUM Command

VACUUM removes old files that are no longer needed. Without it, storage grows forever. Default retention is 7 days.

```
# Remove files older than 7 days (default)
VACUUM my_table

# Remove files older than 24 hours
VACUUM my_table RETAIN 24 HOURS

# WARNING: After VACUUM, you can't time travel beyond retention
```

Performance Best Practices

- Partitioning: Use for large tables, partition by date/region (low cardinality)
- Z-Ordering: Use for high-cardinality filter columns (user_id, product_id)
- File Size: Target 1GB files, use OPTIMIZE if files are small
- Statistics: Databricks auto-collects, used for query planning
- Caching: df.cache() for repeatedly accessed DataFrames
- Broadcast: Use broadcast() for joins with small tables (<100MB)

Optimization Schedule

Run OPTIMIZE daily or weekly on active tables. Run VACUUM weekly with 7-day retention. Monitor table size and query performance.

Unity Catalog

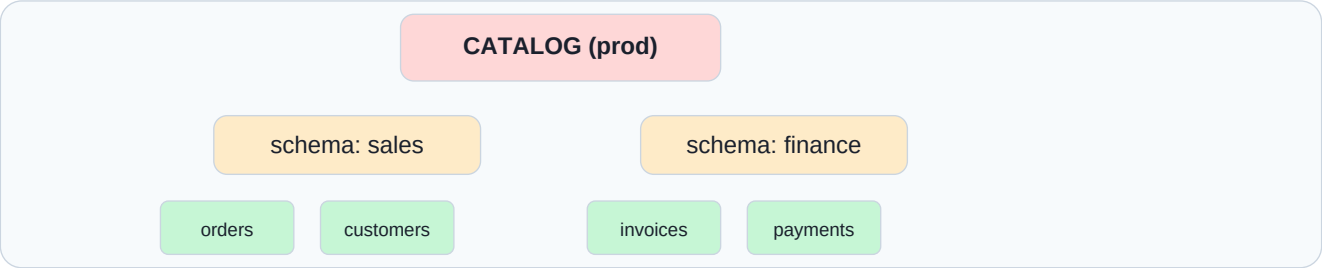
Governance & Security

What is Unity Catalog?

Unity Catalog is Databricks' governance layer. It provides centralized access control, auditing, and data lineage across all your data assets.

Three-Level Namespace

Unity Catalog organizes data in a three-level hierarchy: Catalog → Schema → Table/View




```
# Access pattern: catalog.schema.table
SELECT * FROM prod.sales.orders

# Set default catalog and schema
USE CATALOG prod;
USE SCHEMA sales;
SELECT * FROM orders;  -- Now just table name works
```

Access Control

```
# Grant read access to a table
GRANT SELECT ON TABLE prod.sales.orders TO user@company.com

# Grant schema-level access
GRANT USE SCHEMA ON SCHEMA prod.sales TO analysts
GRANT SELECT ON SCHEMA prod.sales TO analysts

# Create catalog and schema
CREATE CATALOG IF NOT EXISTS prod
CREATE SCHEMA IF NOT EXISTS prod.sales
```

Medallion Architecture

The Standard Data Pattern

What is Medallion Architecture?

Medallion is a data design pattern with three layers: Bronze (raw), Silver (cleaned), Gold (aggregated). It's the standard way to organize data in Databricks.

Bronze Layer (Raw)

- Ingest data exactly as received from source
- No transformations except adding metadata (load_time, source_file)
- Append-only - never update or delete
- Partition by ingestion date
- Keep full history for replay capability

Silver Layer (Cleaned)

- Apply data quality rules (nulls, types, ranges)
- Deduplicate records
- Standardize formats (dates, currencies, names)
- Join with reference data (lookup tables)
- Handle schema evolution

Gold Layer (Business)

- Create business-level aggregations
- Build fact and dimension tables
- Optimize for query patterns (ZORDER, partitioning)
- This is what analysts and BI tools consume
- May have multiple gold tables for different use cases

Medallion Implementation

Code Examples

Bronze Layer Code

```
# Bronze: Ingest raw data
df_raw = spark.read.json('/raw/events/')

df_bronze = df_raw.withColumn('_load_time', current_timestamp()) \
                  .withColumn('_source_file', input_file_name())

df_bronze.write.format('delta') \
    .mode('append') \
    .partitionBy('_load_date') \
    .saveAsTable('bronze.events')
```

Silver Layer Code

```
# Silver: Clean and validate
df_silver = spark.table('bronze.events')

df_silver = df_silver \
    .dropDuplicates(['event_id']) \
    .filter(col('event_type').isNotNull()) \
    .withColumn('event_date', to_date('event_timestamp')) \
    .withColumn('amount', col('amount').cast('decimal(10,2)'))

# MERGE for incremental updates
# (see previous MERGE example)
```

Gold Layer Code

```
# Gold: Aggregate for business
df_gold = spark.sql('''
    SELECT
        event_date,
        region,
        COUNT(*) as total_events,
        SUM(amount) as total_amount,
        COUNT(DISTINCT user_id) as unique_users
    FROM silver.events
    GROUP BY event_date, region
''')

df_gold.write.format('delta') \
    .mode('overwrite') \
    .option('overwriteSchema', 'true') \
    .saveAsTable('gold.daily_summary')
```

Auto Loader

Incremental Data Ingestion

What is Auto Loader?

Auto Loader automatically detects and processes new files as they arrive in cloud storage. It's the recommended way to ingest data into Bronze layer.

Key Benefits

- Exactly-once processing - no duplicates, no missed files
- Automatic schema inference and evolution
- Scales to millions of files
- Checkpointing - knows what's been processed
- Cost-effective - uses cloud notifications (not listing)

Auto Loader Code

```
# Basic Auto Loader
df = spark.readStream \
    .format('cloudFiles') \
    .option('cloudFiles.format', 'json') \
    .option('cloudFiles.schemaLocation', '/schema/events') \
    .load('/raw/events/')

df.writeStream \
    .format('delta') \
    .option('checkpointLocation', '/checkpoints/events') \
    .trigger(availableNow=True)  # Process all available, then stop
    .table('bronze.events')
```

Common Options

Option	Purpose	Example
cloudFiles.format	Source file format	json, csv, parquet
cloudFiles.schemaLocation	Where to store inferred schema	/schema/path
cloudFiles.inferColumnTypes	Infer types or use strings	true
cloudFiles.schemaHints	Override specific columns	id INT, name STRING

Production Tip

Always set schemaLocation to persist schema across restarts. Use trigger(availableNow=True) for batch-style processing in workflows, or trigger(processingTime='1 minute') for continuous streaming.

Structured Streaming

Real-Time Data Processing

Streaming Concepts

Structured Streaming treats a live data stream as a table that's continuously appended. You write queries just like batch, and Spark handles the streaming.

Stream-to-Table Pattern

```
# Read stream
df_stream = spark.readStream \
    .format('delta') \
    .table('bronze.events')

# Transform (same as batch)
df_transformed = df_stream \
    .filter(col('event_type') == 'purchase') \
    .groupBy(window('event_time', '1 hour')) \
    .agg(sum('amount').alias('hourly_total'))

# Write stream
df_transformed.writeStream \
    .format('delta') \
    .outputMode('complete') \
    .option('checkpointLocation', '/checkpoints/hourly') \
    .table('silver.hourly_sales')
```

Output Modes

Mode	Description	Use Case
append	Only new rows to output	No aggregations, just filtering
complete	Full result table each time	Aggregations (groupBy)
update	Only changed rows	Aggregations with watermark

Watermarks for Late Data

```
# Handle late-arriving data
df_stream.withWatermark('event_time', '2 hours') \
    .groupBy(window('event_time', '1 hour')) \
    .count()

# Watermark says: ignore data more than 2 hours late
```

Workflows & Jobs

Production Orchestration

What are Workflows?

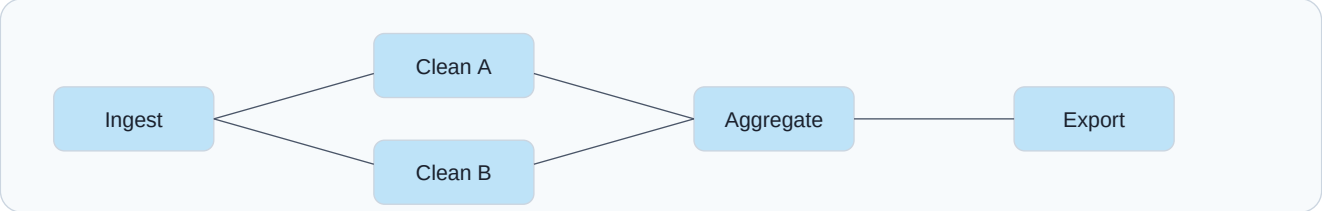
Workflows let you schedule and orchestrate notebooks, Python scripts, and SQL queries. They're essential for production pipelines.

Job Types

Type	Description	Best For
Notebook	Run a Databricks notebook	ETL pipelines
Python Script	Run Python file from repo	Complex logic
SQL	Run SQL queries	Simple transformations
dbt	Run dbt models	dbt users
JAR	Run Scala/Java JAR	Legacy or performance

Task Dependencies

Create DAGs (Directed Acyclic Graphs) where tasks depend on each other. Failed tasks can be retried independently.



Scheduling Options

- Cron: Standard cron expressions (0 8 * * * = 8 AM daily)
- Manual: On-demand execution via UI or API
- File Arrival: Trigger when new files land
- Continuous: Keep running, restart on failure

Best Practices

- Use job clusters (not all-purpose) to save costs
- Enable retries for transient failures (network, spot termination)
- Set alerts for failures (email, Slack, PagerDuty)
- Use parameters to make jobs reusable

Delta Live Tables (DLT)

Declarative Pipelines

What is DLT?

Delta Live Tables is a declarative framework for building reliable data pipelines.
You define WHAT you want, not HOW to build it.

Key Advantages

- Automatic dependency management - no manual ordering
- Built-in data quality with expectations
- Automatic error handling and recovery
- Simplified streaming with one syntax
- Visual lineage and monitoring

DLT Syntax

```
import dlt
from pyspark.sql.functions import *

# Bronze table
@dlt.table
def bronze_events():
    return spark.readStream.format('cloudFiles') \
        .option('cloudFiles.format', 'json') \
        .load('/raw/events/')

# Silver table with quality expectations
@dlt.table
@dlt.expect_or_drop('valid_amount', 'amount > 0')
@dlt.expect_or_fail('valid_id', 'id IS NOT NULL')
def silver_events():
    return dlt.read_stream('bronze_events') \
        .filter(col('event_type').isNotNull())

# Gold aggregation
@dlt.table
def gold_daily_summary():
    return dlt.read('silver_events') \
        .groupBy('event_date').count()
```

Expectations (Data Quality)

Expectation	On Failure	Use Case
@dlt.expect	Track metric only	Monitoring, non-critical
@dlt.expect_or_drop	Drop bad rows	Filter invalid data
@dlt.expect_or_fail	Fail pipeline	Critical validations

SQL in Databricks

For SQL-First Users

SQL is First-Class

You can do almost everything in SQL. Databricks SQL extends standard SQL with Delta Lake features.

Essential SQL Commands

```
-- Create database (schema)
CREATE SCHEMA IF NOT EXISTS my_db;

-- Create Delta table
CREATE TABLE my_db.users (
    id INT,
    name STRING,
    created_at TIMESTAMP
) USING DELTA;

-- Create table from query
CREATE TABLE my_db.summary AS
SELECT region, COUNT(*) as cnt FROM events GROUP BY region;

-- Insert data
INSERT INTO my_db.users VALUES (1, 'Alice', current_timestamp());
```

Delta-Specific SQL

```
-- MERGE (upsert)
MERGE INTO target USING source
ON target.id = source.id
WHEN MATCHED THEN UPDATE SET *
WHEN NOT MATCHED THEN INSERT *;

-- Time travel
SELECT * FROM my_table VERSION AS OF 5;
SELECT * FROM my_table TIMESTAMP AS OF '2024-01-01';

-- Table maintenance
OPTIMIZE my_table ZORDER BY (user_id);
VACUUM my_table RETAIN 168 HOURS;
DESCRIBE HISTORY my_table;
```

SQL Warehouses

SQL Warehouses are serverless compute for SQL workloads. They auto-scale based on query load and are perfect for BI tools.

- Serverless: No cluster management, instant start
- Auto-scaling: Scales up/down based on demand

- Cost: Pay per query, not idle time
- BI Integration: Connect Tableau, Power BI, etc.

Performance Tuning

Make It Fast

Common Performance Issues

- **Shuffle** — Data movement across nodes - expensive!
- **Skew** — One partition has much more data than others
- **Small Files** — Too many small files = slow reads
- **No Predicate Pushdown** — Reading more data than needed
- **Memory Pressure** — Not enough memory for operations

Solutions

```
# 1. Reduce shuffles - use broadcast for small tables
from pyspark.sql.functions import broadcast
df_large.join(broadcast(df_small), 'key')

# 2. Fix skew - salt the key
df.withColumn('salted_key', concat(col('key'), lit('_'), (rand()*10).cast('int')))

# 3. Compact small files
OPTIMIZE my_table

# 4. Enable predicate pushdown - filter early
df.filter(col('date') == '2024-01-01').select('user_id', 'amount')

# 5. Cache reused DataFrames
df_reused = df.filter(...).cache()
```

Spark UI Tips

- Check 'Stages' tab - look for skewed tasks (one much longer)
- Check 'SQL' tab - see query plan and metrics
- Look for 'spill' - means data didn't fit in memory
- Check shuffle read/write size - minimize this

Photon Engine

Photon is Databricks' native vectorized engine. It's 2-8x faster than regular Spark for most queries. Enable it on your cluster for free performance.

Interview Questions

Common Questions & Answers

Conceptual Questions

Q: What is Delta Lake?

A: Open-source storage layer that adds ACID transactions, time travel, and schema enforcement to data lakes.

Q: Explain medallion architecture.

A: Data design pattern with Bronze (raw), Silver (cleaned), Gold (aggregated) layers for progressive data refinement.

Q: Driver vs Worker nodes?

A: Driver coordinates work and runs main(). Workers execute tasks in parallel.

Q: What is Unity Catalog?

A: Governance layer providing centralized access control, auditing, and lineage across all data assets.

Coding Questions

```
# Q: Deduplicate DataFrame keeping latest record
from pyspark.sql.window import Window
window = Window.partitionBy('id').orderBy(col('updated_at').desc())
df.withColumn('rn', row_number().over(window)).filter(col('rn')==1)

# Q: Calculate running total
window = Window.partitionBy('user').orderBy('date').rowsBetween(
    Window.unboundedPreceding, Window.currentRow)
df.withColumn('running_total', sum('amount').over(window))
```

Scenario Questions

- Small file problem: Use OPTIMIZE, increase write partition size
- Pipeline keeps failing: Check cluster logs, add retries, use DLT
- Query too slow: Check Spark UI, add ZORDER, use broadcast
- Handle late data: Use watermarks in streaming

Quick Reference Cheat Sheet

Copy-Paste Ready

Read Data

```
df = spark.read.csv('/path', header=True, inferSchema=True)
df = spark.read.json('/path')
df = spark.read.parquet('/path')
df = spark.table('catalog.schema.table')
df = spark.read.format('delta').load('/path')
```

Write Data

```
df.write.format('delta').mode('overwrite').saveAsTable('my_table')
df.write.format('delta').mode('append').save('/path')
df.write.partitionBy('date').format('delta').save('/path')
```

Common Transformations

```
df.select('col1', 'col2')                    # Select columns
df.filter(col('x') > 10)                     # Filter rows
df.withColumn('new', col('old') * 2)         # Add column
df.groupBy('key').agg(sum('val'))            # Aggregate
df1.join(df2, 'key', 'left')                 # Join
df.dropDuplicates(['id'])                    # Dedupe
df.orderBy(col('date').desc())               # Sort
```

Delta Operations

```
OPTIMIZE table ZORDER BY (col)               -- Compact files
VACUUM table RETAIN 168 HOURS                -- Clean old files
DESCRIBE HISTORY table                       -- View history
SELECT * FROM table VERSION AS OF 5          -- Time travel
RESTORE TABLE table TO VERSION AS OF 5       -- Restore
```

Useful Functions

```
from pyspark.sql.functions import *
col('x'), lit('value'), when(cond, val).otherwise(val)
concat(), substring(), trim(), upper(), lower()
to_date(), year(), month(), datediff()
sum(), avg(), count(), max(), min()
row_number(), rank(), lead(), lag()
```

Certification Guide

Get Certified

Data Engineer Associate

Entry-level certification. Covers everything in this guide. 90 minutes, 45 questions, passing score ~70%.

- Databricks workspace and notebooks
- ELT with Spark SQL and Python
- Delta Lake basics (CRUD, time travel)
- Data pipelines and workflows
- Unity Catalog basics

Data Engineer Professional

Advanced certification. Requires hands-on experience. 120 minutes, 60 questions.

- Advanced Delta Lake (streaming, CDC, optimization)
- Data modeling and medallion architecture
- Production pipelines (DLT, monitoring)
- Security and governance
- Performance tuning

Study Tips

- Complete Databricks Academy free courses
- Practice with Community Edition daily
- Build a real project (medallion pipeline)
- Review official exam guide
- Take practice tests

Free Resources

- **Databricks Academy** — academy.databricks.com - free courses
- **Community Edition** — Free workspace for practice
- **Documentation** — docs.databricks.com - comprehensive
- **Delta Lake Docs** — delta.io - deep technical details

Hands-On Project

Build This to Master Databricks

Project: E-Commerce Analytics Pipeline

Build a complete data pipeline that processes e-commerce events, creates clean data models, and generates business reports.

Step 1: Setup

- Create Databricks Community Edition account
- Create catalog: ecommerce, schemas: bronze, silver, gold
- Download sample data or use databricks-datasets

Step 2: Bronze Layer

```
# Create bronze tables for raw events
CREATE TABLE bronze.raw_orders (
    order_id STRING,
    customer_id STRING,
    product_id STRING,
    quantity INT,
    price DECIMAL(10,2),
    order_timestamp TIMESTAMP,
    _load_time TIMESTAMP DEFAULT current_timestamp()
) USING DELTA;
```

Step 3: Silver Layer

```
# Clean and validate orders
CREATE TABLE silver.orders AS
SELECT DISTINCT
    order_id,
    customer_id,
    product_id,
    quantity,
    price,
    quantity * price as total_amount,
    DATE(order_timestamp) as order_date
FROM bronze.raw_orders
WHERE order_id IS NOT NULL
  AND quantity > 0;
```

Step 4: Gold Layer

```
# Daily sales summary
CREATE TABLE gold.daily_sales AS
SELECT order_date,
       COUNT(DISTINCT order_id) as total_orders,
       COUNT(DISTINCT customer_id) as unique_customers,
       SUM(total_amount) as revenue
FROM silver.orders
GROUP BY order_date;
```

Project Continued

Advanced Steps

Step 5: Add Incremental Loading

```
# Use MERGE for incremental updates
MERGE INTO silver.orders AS target
USING (
    SELECT * FROM bronze.raw_orders
    WHERE _load_time > (SELECT MAX(_load_time) FROM silver.orders)
) AS source
ON target.order_id = source.order_id
WHEN NOT MATCHED THEN INSERT *;
```

Step 6: Create Workflow

- Create a new Workflow in Databricks UI
- Add Task 1: Bronze ingestion notebook
- Add Task 2: Silver transformation (depends on Task 1)
- Add Task 3: Gold aggregation (depends on Task 2)
- Set schedule: Daily at 6 AM
- Configure alerts for failures

Step 7: Add Data Quality

```
# Add expectations with DLT
@dlt.table
@dlt.expect_or_drop('valid_quantity', 'quantity > 0')
@dlt.expect_or_drop('valid_price', 'price > 0')
@dlt.expect('reasonable_total', 'total_amount < 10000')
def silver_orders():
    return dlt.read_stream('bronze_raw_orders') \
        .dropDuplicates(['order_id'])
```

What You'll Learn

- Complete medallion architecture implementation
- Delta Lake CRUD and MERGE patterns
- Auto Loader for incremental ingestion
- Workflow orchestration and scheduling
- Data quality with DLT expectations
- Unity Catalog governance

Portfolio Tip

Document this project on GitHub with screenshots. Add it to your resume and LinkedIn. This single project demonstrates all key Databricks skills employers want.

Summary & Next Steps

You've Got This!

What You Learned

- Databricks architecture and workspace navigation
- Cluster types and cost optimization
- Spark DataFrame operations and SQL
- Delta Lake - ACID, time travel, MERGE, optimization
- Unity Catalog for governance
- Medallion architecture pattern
- Auto Loader and structured streaming
- Workflows and production pipelines
- Delta Live Tables
- Performance tuning techniques

Your Action Plan

- Week 1-2: Set up Community Edition, complete basic tutorials
- Week 3-4: Build bronze/silver/gold pipeline with sample data
- Week 5-6: Add streaming with Auto Loader
- Week 7-8: Create workflows, add monitoring
- Week 9-10: Study for Associate certification
- Week 11-12: Take certification exam

Remember: The best way to learn is by doing.

Start building today. Make mistakes. Learn from them.
You're already ahead by having this roadmap!