1. Purpose of Compute in Databricks

- Compute is the engine that runs:
    - Notebook-based interactive workloads
    - SQL queries
    - ETL / ELT pipelines
    - Ad-hoc exploration
    - Machine learning experiments
- Every operation in Databricks requires a compute resource (cluster).

All-Purpose Compute (Classic Compute)

- A dedicated cluster of VMs created for interactive use.
- Used heavily by data engineers, analysts, and data scientists for:
    - Development
    - Debugging
    - Exploratory data analysis
    - Running long notebooks
    - Testing ETL logic

Characteristics

- Stays running until manually terminated or auto-terminated.
- Attached to notebooks or SQL queries.
- More expensive than serverless or jobs compute because it is persistent.
- Provides full cluster control (VM size, autoscaling, libraries, runtime version).

Cost Components of Databricks Compute

Compute cost = Azure VM cost + Databricks DBU cost

A. Azure Infrastructure Cost

- You pay Azure for the underlying VMs that Databricks uses.
- Cost varies based on:
    - VM size (CPU + RAM)
    - Region
    - Spot or On-Demand
    - Storage type

B. Databricks DBU Cost

- DBU = Databricks Unit (Databricks platform usage).
- You pay per hour for:
    - Databricks Runtime (Spark + optimizations)
    - Workspace services
    - Notebooks
    - Security + governance layer
    - Monitoring & execution engine
- Example:
    - Cluster uses 0.75 DBU/hour
    - Price: $1/DBU
    - DBU cost ≈ $0.75/hr

Total Cost Example

- Azure Infra ≈ $0.20/hr
- Databricks DBU ≈ $0.40/hr
  Total ≈ $0.60/hr

Typical Hardware Ratios

Clusters commonly follow a 1 CPU : 4 GB RAM ratio.

Examples:

- 4 cores → 16 GB RAM
- 4 cores → 32 GB RAM

When to Use All-Purpose Compute

Use for:

- Interactive exploration
- Development/debugging
- Running notebooks
- Collaborative analysis
- Training ML models in an exploratory phase
- Ad-hoc SQL queries

Avoid for:

- Automated production jobs
- Scheduled ETL pipelines
  (use Jobs Compute instead)

Job Compute – Core Concepts

What is Job Compute?

- A serverless-like compute that Databricks uses specifically to run Jobs (scheduled/pipeline executions).
- It is fully managed and automated.
- You cannot start or stop Job Compute manually.

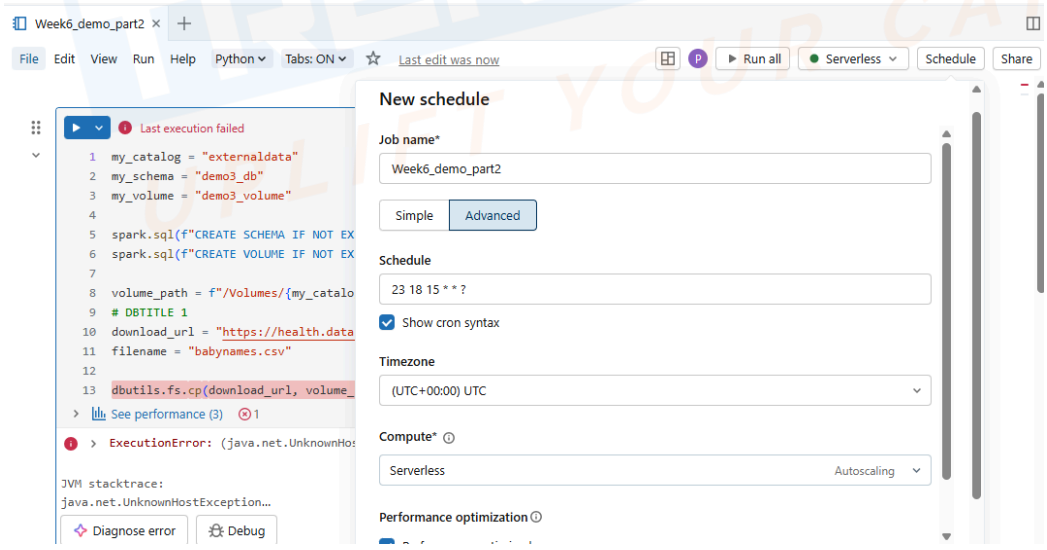
How Job Compute Works

- When a job is triggered:
    - Databricks automatically starts the Job Compute cluster.
    - Runs the workload (notebook / SQL workflow / pipeline).
    - Automatically terminates after the execution finishes.
- This ensures cost-efficiency because compute runs only when needed.

Scheduling Jobs

Jobs can be scheduled using Cron

- Databricks allows you to schedule jobs at specific intervals using Cron expressions.
- Example schedules:
    - Every day at midnight → 0 0 * * *
    - Every 5 minutes → */5 * * * *
    - Every Monday at 9AM → 0 9 * * MON
- - Use Cron expressions for scheduling job runs.
- - Examples: Daily, every 5 minutes, specific weekdays.



Cluster Policies

Cluster policies define what kind of compute users are allowed to create.
They enforce cost control, security, and standardization across teams.

Why Policies Are Needed

- Prevent users from choosing very expensive clusters.
- Enforce required security settings.
- Standardize compute size or instance types.
- Ensure compliance with organization rules.

Different Types of Policies

1) Personal Compute

- Creates a single-node cluster.
- Minimal configuration allowed.
- For individual development.

2) Shared Compute

- Multi-node cluster.
- Designed to be used by multiple users.
- More flexible than personal compute.

3) Power User Compute

- Larger multi-node clusters.
- Intended for advanced users or heavy workloads.

4) Job Compute

- Default compute used for jobs.
- Managed automatically.

Creating Custom Policies

A policy is created using **JSON configuration**.

Example Custom Internal Policy{
  "node_type_id": {
    "type": "fixed",

```
    "value": "Standard_D4s_v3"
  },
  "spark_version": {
    "type": "fixed",
    "value": "16.4.x-scala2.13"
  },
  "autotermination_minutes": {
    "type": "fixed",
    "value": 20
  }
}
```

Explanation:

- type: fixed → user cannot change it.
- Defines:
    - A fixed VM size.
    - A fixed runtime (Spark version).
    - A fixed auto-termination time.

Inline Policy Options

When applying a policy, you may see:

1) Fix All

- All configurations are locked.
- User cannot change anything.

2) Enforce Policy Compliance

- User can modify settings as long as they remain within policy limits.
- Example: they can pick any worker count between 1–4.

Permissions on Compute

Permissions define what level of access a user has on a cluster.

Can Attach To

- Can attach their notebook to the compute.
- Cannot start or restart.

Can Restart

- Can start, restart, and terminate compute.
- Includes can attach to.

Can Manage

- Highest access level.
- Can:
    - Edit compute settings.
    - Change size.
    - Modify permissions.
    - Restart or terminate.
- Includes can attach to + can restart.

Cluster Pools

Purpose

Cluster Pools are used to reduce cluster startup time and speed up autoscaling by keeping a set of pre-initialized virtual machines ready.

How Pools Work

- A pool maintains a group of idle, pre-warmed VMs.
- When a cluster is created or scaled out, it can instantly attach these VMs instead of provisioning from scratch.
- This significantly reduces:
    - Cluster creation time
    - Autoscaling delays
    - Job startup latency

Key Features of Pools

1. Min Idle Instances

- The number of VMs the pool keeps ready at all times.
- These VMs are pre-allocated and incur cost.
- Higher min idle = fastest startup but higher cost.

Example:
Min Idle = 2
→ 2 machines always warm and ready.

2. Max Capacity

- Maximum VMs the pool is allowed to scale up to.

3. Pool-backed Clusters

- All-purpose clusters can attach to a pool.
- Job clusters can also be configured to use pools.

Cost Implications

- Idle instances in the pool cost money because you are reserving cloud VMs.
- But pools reduce:
    - Job wait time
    - Developer waiting time in notebooks
    - Autoscaling delays

Spot Instances in Pools

- Pools support Spot VMs (Azure: Low-Priority VMs).
- Spot VMs are cheaper but may be reclaimed by the cloud provider at any time.

When to Use Spot Instances

- Recommended for:
    - ETL jobs
    - Non-critical compute
    - Retryable or stateless tasks
- Not recommended for:
    - Production-critical jobs
    - State-heavy workloads

Databricks Runtime (DBR)

DBR is the software environment that runs on your clusters.

DBR Includes:

- Delta Lake
- Python, Scala, Java, and R runtimes
- Ubuntu OS + system libraries
- GPU libraries (if using GPU runtime)
- ML libraries (in ML runtime flavors)

DBR Versioning Example: 16.4 LTS

Major version: 16

Large architectural and performance improvements.

Feature version: 4

Bug fixes, minor features, enhancements.

LTS (Long-Term Support)

- Most stable version
- Recommended for production
- Receives extended security patches

SQL Warehouses (SQL Endpoints)

SQL Warehouses are used for:

- Databricks SQL Editor
- BI dashboards
- Data warehouse workloads
- Ad hoc queries

These are dedicated compute engines optimized for SQL-only workloads, separate from clusters.

Types of SQL Warehouses

1. Serverless SQL Warehouse

- Fully managed
- Starts instantly
- No cluster configuration
- Fastest performance
- Best for analysts & BI dashboards

2. Pro SQL Warehouse

- More customization
- Supports higher concurrency
- Slightly slower startup than serverless

3. Classic SQL Warehouse

- Legacy
- Not recommended anymore
- Exists only for backward compatibility

Starter Warehouse

- Automatically created when using SQL Editor for the first time
- Lightweight
- Serverless
- Auto-stops quickly
- Best for beginners and small workloads

Classic Compute vs Serverless Compute

Classic Compute

- Has a long wait time (typically 5–7 minutes) to start clusters.
- Requires manual configuration of:
    - Node types
    - Autoscaling settings
    - Number of workers
    - Spot vs on-demand
    - DBR version
- Cost structure:
    - Infra cost → Azure
    - Software cost (DBUs) → Databricks
- Users must manage performance and cost trade-offs manually

Serverless Compute

What it solves

- Zero cluster startup wait time (starts in seconds).
- No need to configure infra — Databricks handles everything.

What the user selects

Just pick a goal:

- Standard mode → balanced cost/performance
- Performance mode → faster execution

Everything else (node types, scaling, machine selection) is handled automatically by Databricks.

Cost model in Serverless

- Infra → Databricks
- Software → Databricks
- You only manage cost using Budget Policy.

Environment Setting (right pane)

- This appears only in serverless compute.
- Environment refers to the version of the Databricks Execution Runtime used behind the scenes.

Memory in Serverless Notebook

- Refers to memory consumed by the REPL session.
- Notebook → acts as a REPL
  REPL → connects via Spark Connect → to Serverless Spark Cluster

Budget Policy

Why it is important

- Helps track how many DBUs a user consumes.
- Admins use this to:
  - Control cost
  - Enforce spending limits
  - Detect misuse or unexpected job execution

Where to configure

- Settings → Compute → Budget Policy

How to analyze consumption

- Query System Tables:
  - system.billing.usage
  - system.compute.events
  - System.job.run_data

Serverless Compute – Extra Key Points

1. Simple / Performant / Maintenance-Free

- Serverless compute is designed to remove infrastructure complexity.
- No need to pick VM types, node types, autoscaling rules, or runtime versions.
- Databricks handles provisioning, optimization, scaling, and patching.

2. Serverless Is a Versionless Product

- Unlike Classic Compute, where you choose a DBR Runtime version (e.g., 16.4 LTS), serverless does not expose DBR versions.
- Databricks automatically keeps serverless compute:
    - Upgraded
    - Patched
    - Secured
    - Optimized for performance

You only choose Standard or Performance mode.

3. Serverless Overspend Protection

- Protects your workspace from unexpected high compute bills.
- Automatically monitors long-running SQL / Notebook workloads.
- If a query/job runs unusually long:
    - It can auto-terminate
    - Or ask for approval depending on policy settings
- Helps prevent runaway costs due to accidental infinite loops or heavy workloads.

4. Serverless Notebook Timeout

- Every serverless notebook cell has a default execution timeout of 2.5 hours.
- This applies to:
    - Python
    - SQL
- Prevents infinite execution and unnecessary cloud costs.

Data Lake Overview

A data lake is a centralized storage system that holds large volumes of raw data in its native format. All data is stored as files, allowing storage of structured, semi-structured, and unstructured data. Common cloud storage systems used as data lakes include Amazon S3, Azure ADLS Gen2, and Google Cloud Storage.

Advantages of a Data Lake

- Cost-effective storage
- Highly scalable
- Supports any form of data
- Decouples storage and compute

Challenges with Traditional Data Lakes

Traditional data lakes do not provide database-style ACID guarantees, which leads to reliability and consistency issues.

ACID properties are defined as:

- **Atomicity:** All steps in a transaction must succeed; otherwise, none should apply.
- **Consistency:** Data must remain valid and follow defined rules before and after transactions.
- **Isolation:** Concurrent operations should not interfere or reveal partial results.
- **Durability:** Once a transaction is complete, it must persist even if failures occur.

Why Data Lakes Fail Without ACID Guarantees

Failed Job During Append

Spark jobs write multiple part files.
If 5 files already exist and a new append job creates only 2 files before failing, the folder ends up with 7 files. A reader will incorrectly read all 7 files, breaking atomicity and consistency.

Failed Job During Overwrite

Overwrite happens in two steps:

1. Delete existing files
2. Write new files

If step 2 partially completes and the job fails, the dataset ends up with incomplete data, violating atomicity, consistency, and durability.

Simultaneous Reads and Writes

When a writer is generating new part files while a reader is trying to read the same dataset, the reader may process partial or intermediate files. This breaks isolation and results in incorrect outputs.

Schema Mismatch During Appends

If existing files contain 5 columns and a new job appends files with 7 columns, the data lake ends up with files of inconsistent schema.
This leads to:

- Data corruption
- Lack of validation
- Poor data quality
- No support for robust DML operations
- Difficulty maintaining historical versions

How Delta Lake Solves These Problems

Delta Lake adds a transaction log and enforces ACID guarantees on top of a traditional data lake. It ensures reliable file operations, schema enforcement, schema evolution, versioning (time travel), and consistent reads/writes. As a result, it transforms a raw data lake into a reliable, high-quality, and production-ready data storage layer.

Introduction to Delta Lake

Delta Lake is an open-source storage layer that brings reliability and ACID transactions to data lakes.
It works on top of existing storage such as ADLS, S3, and GCS, enabling reliable, consistent, and high-quality data processing.

Delta Lake = Parquet Files + Transaction Log (_delta_log)

The transaction log contains JSON files (00000.json, 00001.json, etc.) that track every change, ensuring consistent reads and writes.

How Delta Lake Handles Write Operations

For every write operation:

1. All Parquet part files are written first.
2. A new JSON transaction log file is added to _delta_log.

For every read operation:

1. Delta reads the transaction logs to determine the latest valid state.
2. Only the committed Parquet files are read.

This approach ensures reliability and prevents partial or corrupted data from being exposed to readers.

How Delta Lake Solves Data Lake Problems

A. Job Failing While Appending

Problem in traditional data lake:
If a job writing 5 new files fails after writing only 2, readers will see all 7 files, causing inconsistent data.

Delta Lake behavior:
If the append job fails, no new transaction log entry is committed.
Readers continue to see only the previously committed 5 files.

This preserves Atomicity and Consistency.

B. Job Failing While Overwriting

Traditional overwrite:

1. Delete existing files
2. Write new files
   If failure occurs → irreversible data corruption

Delta Lake overwrite:

1. New files are written first
2. Only after successful completion, the transaction log marks the old files as removed

If failure occurs mid-way, the old data remains untouched.
Ensures Atomicity, Consistency, and Durability.

C. Simultaneous Reads and Writes

Without Delta Lake:
Readers may see partially written files.

With Delta Lake:
Readers see only the last fully committed version (e.g., 00000.json).
Ongoing writes (e.g., 00001.json) are not visible until committed.

This ensures Isolation.

Unity Catalog Storage Structure

Hierarchy

- Metastore
    - Catalog
        - Schema (Database)
            - Tables
            - Views
            - Volumes
            - Models

Data Storage Types

Managed Tables

- Databricks manages the physical storage location.
- Used for structured data.

External Tables

- User specifies an external ADLS/S3/GCS path.
- Useful when you want full control over storage location.

Volumes

- Used to store unstructured, semi-structured, or structured files.

How Delta Tracks File Changes

Each transaction log records operations using “add” and “remove” entries.

Example:

- add parquet1
- add parquet2
- remove parquet_old

Readers follow the transaction logs to determine which Parquet files currently represent the valid table state.

Deletion Vectors

Delta Lake uses deletion vectors for efficient row-level deletes and updates.

Why is it needed?

→ Parquet files may have millions of rows.
→ Deleting even a single row would require rewriting the entire file.

How it works:

- Instead of rewriting Parquet files, Delta creates a deletion vector marking the rows that should be excluded during reads.
- Only metadata is updated, not the full file.

This improves the performance of:

- Deletes
- Updates
- MERGE operations

Example:

- File1 contains rows 1, 2, 3
- Deleting row 1 creates a deletion vector indicating “remove row 1”
- File1 remains intact but row 1 becomes invisible to queries