Introduce yourself:

Hi, I'm Ramkumar. I started my career in 2003 as a developer at ABT Limited. After that, I spent 8 years working on SQL Server development, administration and performance tuning. During this time, I worked with companies like Verizon, Cognizant, and Bank of America.

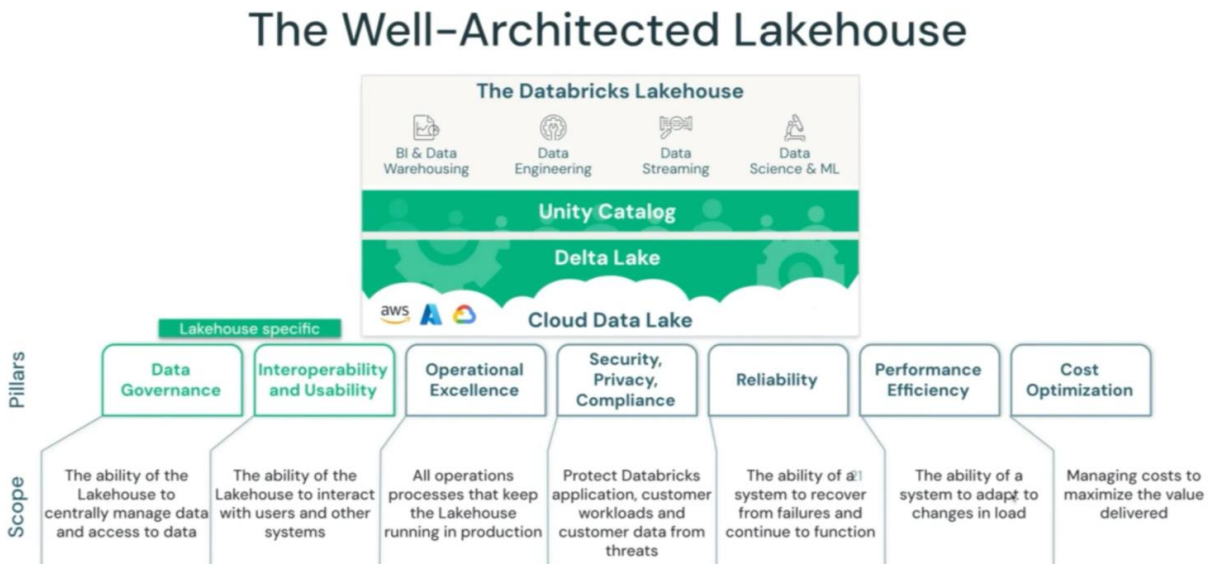
In 2015, joined Capgemini as a Technical Architect and I worked on both on-prem big data and aws cloud technologies.

In 2017, I joined Credit Suisse bank, which is recently acquired by UBS bank. Since then, I've been playing various roles such as Big Data and Cloud Architect, ARB member, Product Owner of an application. I have been driving few department-level cost optimization initiatives as-well.

My core focus areas are to make use of the full potentials of Databricks and Delta Lake and started exploring MLOps capabilities as-well.

Azure Well Architected Framework:

Databricks Well Architected Framework



Terms for Architects:

Data Governance, Cost optimization, Security, Reliability, Performance, Availability/Fault tolerance, Scalability, Risk/Compliance, DR

Responsibilities: Ensure that architectural decisions within an organization align with its strategic goals, standards, and best practices.

Data governance pillars: Data ownership, Data quality, Data security & compliance, metadata management, data lifecycle management.

Data architecture is a discipline that define the structure, processes, governance, standards and best practices for collecting, storing, transforming, and using data to support business needs

Concepts:

Issue in On premise: CAPEX & OPEX costs. Cost and delay in procuring server, cost for NW/System admins. Overhead for HW, OS, Libraries, license, maintenance, upgrade, security

Virtualization: Allows to run multiple OS on a singly host server

Virtualization software's: VMWare workstation/vSphere, KVM (AWS), XEN (AWS), VirtualBox

Cloud computing - Providing Infra/platform/software as services over the Internet.

Cloud Benefits: No CAPEX, Time to market, pay as-you-go, Agility, Auto scaling, Highly available, Fault tolerance Wide range of services to meet all our needs

IAAS: Control Host, OS, Runtime, Application, Data. We are responsible for license, maintenance, upgrade, security

PAAS: Control over Application, data. No need to worry about HW, OS, Libraries related license, maintenance, upgrade, security

SAAS: Control over data. No need to worry about HW, OS SW license, maintenance, upgrade, security

- Cloud providers have limits on Compute, storage.

- Elasticity - add, remove, modify compute capacity depends on the load.

- Vertical Scalability (Add more capacity to existing server. downtime is required) vs Horizontal Scalability (Add more servers in the cluster for better performance)

- Cloud service providers supports Auto scalling based on metrics (ex: cpu load) or schedule.

Ports: HTTP 80, HTTPS 443, FTP 21, SSH 22 (Putty/Mobaxterm), IMAP 143, DNS 54, SQL 3306

Benefits of distributed system: Scalability, Fault tolerance, parallism and high availability

Microsoft fabric is an end to end data and analytics platform enterprises that requires a unified solution.

Principles of Software Development: SOLID, DRY, KISS, and more

- **Single Responsibility Principle (SRP)**

a class should have only one well-defined responsibility.

- **Open/Closed Principle (OCP)**

Emphasizes designing code that is open for extension but closed for modification. In other words

- **Liskov Substitution Principle (LSP)**

LSP highlights the importance of adhering to contracts when inheriting classes. Specifically, if a class B is a subclass of class A, then it should be able to be used as a replacement for A without affecting the system's overall consistency

- **Interface Segregation Principle (ISP)**

(ISP) advocates for defining specific interfaces for clients rather than having a monolithic interface. In other words, clients should not be forced to implement methods they don't use.

- **Dependency Inversion Principle (DIP)**

The Dependency Inversion Principle (DIP) encourages the use of abstract dependencies rather than relying on concrete classes. In other words, high-level modules should not depend directly on low-level modules but on common abstractions.

DRY - DRY (Don't Repeat Yourself)

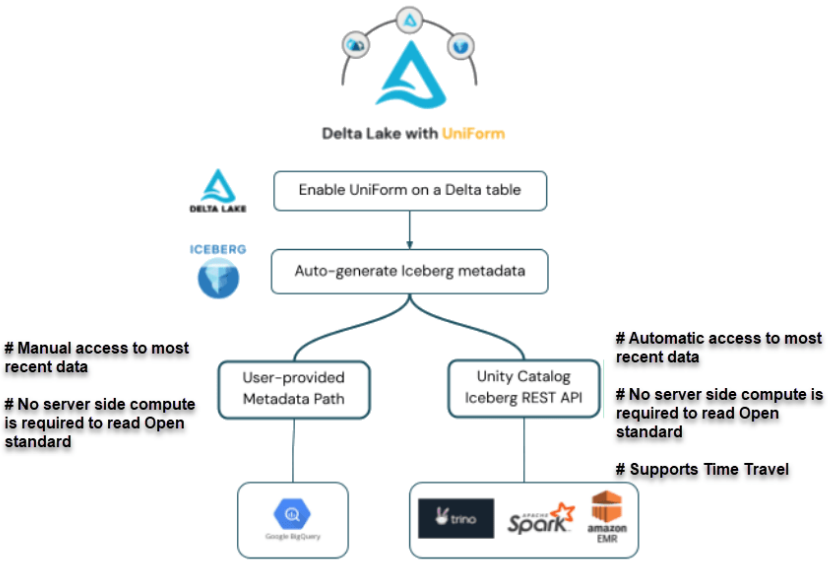
- Reduction of Complexity
- Elimination of Duplicate Code
- Grouping by Functionality
- Code Reusability

KISS (Keep It Simple, Stupid)
YAGNI (You Ain't Gonna Need It) – don't implement if not immediately required

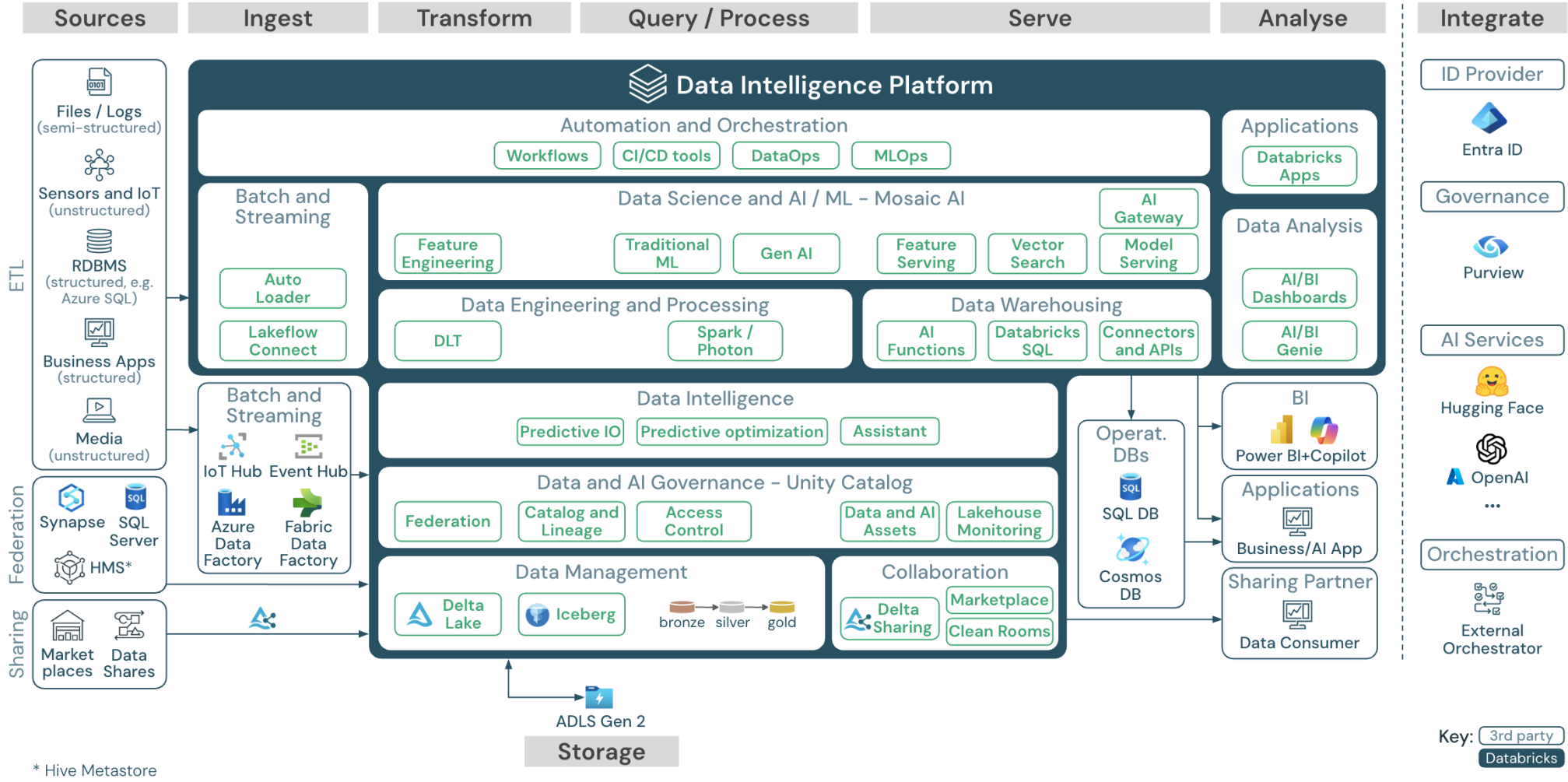
c:

Databricks UniForm, or Delta Lake Universal Format, allows Delta Lake tables to be read by Iceberg clients without rewriting the data. It essentially provides interoperability between Delta Lake and Iceberg, enabling you to use Iceberg clients with your Delta Lake tables

Billing Tip: Billing for databricks DBUs starts when Spark Context becomes available. Billing for the cloud provider starts when the request for compute is received and the VMs are starting up.


Delta Lake with UniForm
DELTA LAKE
ICEBERG
Enable UniForm on a Delta table
Auto-generate Iceberg metadata
Manual access to most recent data
No server side compute is required to read Open standard
User-provided Metadata Path
Unity Catalog Iceberg REST API
Automatic access to most recent data
No server side compute is required to read Open standard
Supports Time Travel
trino
Spark
amazon EMR

Azure Databricks Reference Architecture



Azure Data Service	Top 3 Use Cases	Top 3 Features
Azure Synapse Analytics	1. Data warehousing for BI 2. Big data analytics (Spark/SQL) 3. ETL/ELT with Pipelines	1. Unified SQL + Spark engine 2. Serverless & dedicated pools 3. Power BI integration
Azure Data Lake Storage (ADLS)	1. Centralized data lake for all formats 2. Staging for ETL pipelines 3. Data science and ML data source	1. Hierarchical namespace (HNS) 2. Fine-grained RBAC + ACLs 3. Scalable to petabytes
Azure Databricks	1. Scalable data engineering 2. Machine learning/AI 3. Streaming + batch pipelines	1. Optimized Apache Spark platform 2. Delta Lake for ACID transactions 3. MLflow for ML lifecycle
Azure Cosmos DB	1. Globally distributed transactional apps 2. Real-time IoT/telemetry 3. Product catalogs & personalization	1. Multi-model support (SQL, MongoDB, etc.) 2. Global replication 3. <1ms read/write latency
Azure SQL Database	1. OLTP applications 2. Web & mobile app backends 3. Modernizing on-prem SQL workloads	1. Fully managed PaaS 2. Built-in HA & backup 3. Intelligent performance tuning
Microsoft Fabric	1. Unified lakehouse analytics 2. Power BI + data engineering integration 3. Real-time pipelines and business insight	1. OneLake as a single data store 2. Integrated Spark + Power BI 3. No-code data pipelines

Cluster Type	Use Case	Recommended?	Notes
All-Purpose Cluster	Interactive dev, notebooks, ad-hoc work	⛔ Not for prod	Good for development. Costly if left running.
Job Cluster	Automated ETL jobs, batch pipelines	✅ Yes	Cost-efficient for jobs; auto-terminates after job ends.
DLT Cluster	Managed ETL pipelines (batch/streaming)	✅ Yes	Built-in reliability, monitoring, and data quality.
SQL Warehouse - Classic	Interactive SQL, BI dashboards	⛔ Outdated	Older option, being phased out for Pro/Serverless.
SQL Warehouse - Pro	Fast queries, high concurrency BI	✅ Yes	Good for shared dashboards and frequent queries.
SQL Warehouse - Serverless	Pay-per-use SQL for BI tools	✅ Yes (BI)	No cluster setup needed. Best for Power BI/Tableau/Looker.

Spark vs Databricks

Feature	Spark	Databricks	Key Features
Databricks Runtime	NO	YES	Photon Run multiple versions of Spark Built-in optimized for cloud storage Compute optimizations Multi-user cluster sharing Second level billing
Managed Data Lake	NO	YES	ACID Transations Schema management Batch/Stream read-write support Data versioning Performance optimizations
Integrated Workspace	NO	YES	Interactive Notebooks Real-time collaboration Github integration One-click visualizations
Enterprise Security	NO	YES	Access control for notebooks, clusters, jobs and data Audit logs Data encryption (at rest and motion) Compliance (HIPAA, SOC 2)
Integration	NO	YES	Connect with Power-BI/Tableau via JDBC/ODBC REST API Data Source Connectors
Expert Support	NO	YES	

Others feature: Delta Lake, Unity Catalog, DLT, MLOps, Cloud Integrations

Azure Services:

Azure VM Types

General purpose (A, B, and D family series): These VMs have a balanced CPU-to-memory ratio. They are best suited for testing and development (A series only), burstable workloads (B series only), and general-purpose workloads (D series only).

Compute optimized (F family series): These VMs have a high CPU-to-memory ratio. They are best suited for web servers, application servers, network appliances, batch processes, and any workload where bottlenecks and a lack of resources will typically originate in the CPU over memory.

• **Memory optimized (E and M family series):** These VMs have a high memory-to-CPU ratio. They are best suited for relational databases, in-memory analytics, and any workload where bottlenecks and a lack of resources will typically originate in the memory over the CPU.

• **Storage optimized (L family series):** These VMs have high disk throughput and I/O. They are best suited for data analytics, data warehousing, and any workload where bottlenecks and a lack of resources will typically relate to the disk over the memory and CPU.

• **GPU optimized (N family series):** These VMs have graphics processing units (GPUs). They are best suited for compute-intensive, graphics-/gaming-intensive, visualization, and video conferencing/streaming workloads.

• **High performance (H family series):** These are the most powerful CPU VMs that Azure provides, offering high-speed throughput network interfaces. They are best suited for compute and network workloads such as SAP HANA

[Family] + [Sub-Family] + [# of vCPUs] + [Additive Features] + [Version]

VM series naming:

• Family: This represents the VM family series.
• Sub-Family: This represents the specialized VM differentiations.
• # of vCPUs: This represents the number of vCPUs of the VM.
• Additive Features examples:
    a = AMD-based processor
    b = Block Storage performance
    d = diskful (that is, a local temp disk is present);
    i = isolated size
    l = low memory; a lower amount of memory than the memory intensive size
    m = memory intensive; the most amount of memory in a particular size
    p = ARM Cpu
    t = tiny memory; the smallest amount of memory in a particular size
    s = Premium Storage capable, including possible use of Ultra SSD

Some example breakdowns are as follows:
• B2ms: B series family, two vCPUs, memory-intensive, Premium Storage-capable
• D4ds v5: D series family, four vCPUs, local temp disk, Premium Storage-capable, version 5
• **D4as v5**: D series family, four vCPUs, AMD, Premium Storage-capable, version 5
• E8s v3: E series family, eight vCPUs, Premium Storage-capable, version 3
• NV16as v4: N series family, NVIDIA GRID, 16 vCPUs, AMD, Premium Storage-capable, version 4

Normalization

Normalization is the process of organizing the data in a relational database to minimize redundancy and dependency by dividing large tables into smaller ones and linking them using relationships.

Normal Forms

1. **First Normal Form (1NF):** Eliminate repeating groups and ensure that each column contains only atomic values (no arrays or lists).

   - Before 1NF:

CustomerID	Name	Phone Numbers
1	Alice	123, 456
2	Bob	789

   - After 1NF:

CustomerID	Name	Phone Number
1	Alice	123
1	Alice	456
2	Bob	789

2. **Second Normal Form (2NF):** Eliminate partial dependency, which means each non-prime attribute must depend on the whole of the primary key

- Before 2NF:

OrderID	ProductID	ProductName	Quantity
1	101	Widget	10
1	102	Gadget	5

Here, **ProductName depends on ProductID**, and **Quantity depends on both OrderID and ProductID**. To achieve 2NF, we split the table into two:

- **Orders Table**:

OrderID	ProductID	Quantity
1	101	10
1	102	5

3. **Third Normal Form (3NF):** Eliminate transitive dependency, meaning non-key attributes must not depend on other non-key attributes

- Before 3NF:

StudentID	Course	Instructor	InstructorPhone
101	Math	John	12345
102	Science	Mary	67890

Here, InstructorPhone depends on Instructor, and Instructor depends on Course, which is a transitive dependency. To achieve 3NF, we split the table:

- **Student-Course Table:**

StudentID	Course	Instructo
101	Math	John
102	Science	Mary

- **Instructor Table:**

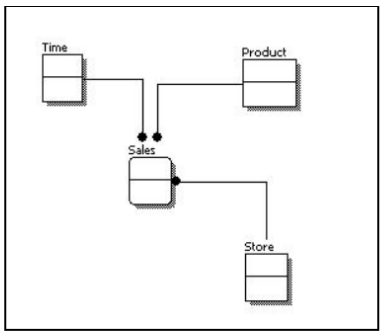
Instructor	InstructorPhone
John	12345
Mary	67890

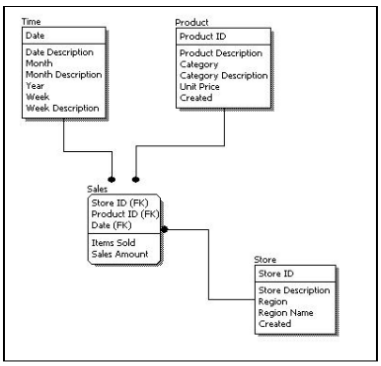
Data Modelling

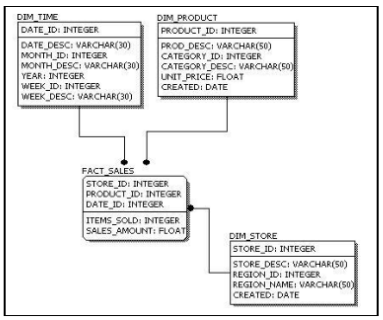
Definition: Data modeling is the process of creating a visual representation of data and its relationships to facilitate database design and ensure data integrity, performance, and usability. It helps structure data logically and physically for storage and analysis.

1. **Conceptual Data Model:**
   - High-level, abstract model focused on business requirements.
   - Defines entities, relationships, and attributes without technical details.
   - Example: Entities like Customer, Order, Product.
2. **Logical Data Model**:
   - Intermediate model that defines the structure of the data, including entity relationships, attributes, and data types, without focusing on the DBMS.
   - Example: Customer has attributes like CustomerID (Primary Key), Name, Address.
3. **Physical Data Model**:
   - Implementation-focused, includes DBMS-specific details like table structures, indexes, data types, and constraints.
   - Example: MySQL table creation with fields CustomerID INT AUTO_INCREMENT PRIMARY KEY.

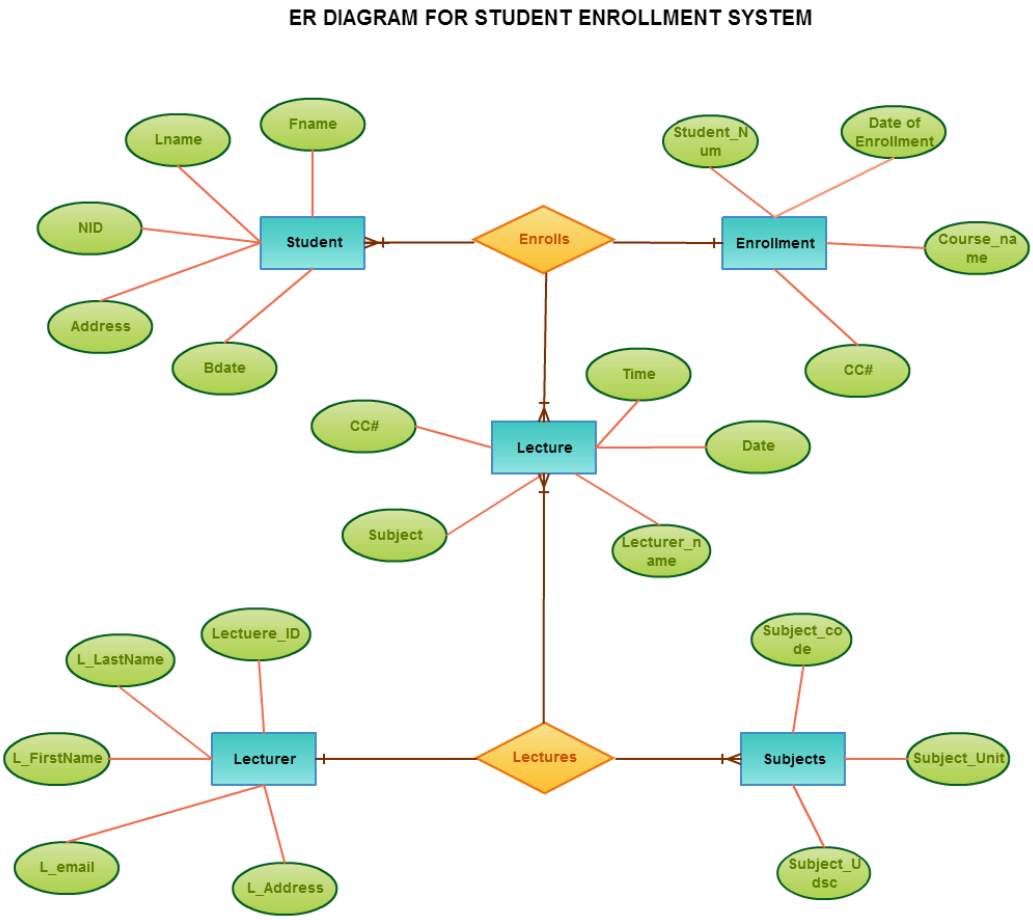
Conceptual Model Design Logical Model Design Physical Model Design







ER Diagram (conceptual data model):



Data Warehousing

A data warehouse is a centralized repository that combines data from various sources to support business intelligence (BI) and analytical activities.

- Denormalized. OLAP, Optimized for read, Historical data

BASIS	DATA WAREHOUSE	DATABASE
Definition	A kind of database optimized for gathering information from different sources for analysis and business reporting	Data storage or collection in an organized manner for storage, updating, accessing and recovering a data
Data Structure	Denormalized data structure is used for enhanced analytical response time	Normalized data structure is there in a database in separate tables
Data timeline	Historical data is stored for analytics while current data can also be used for real-time analysis	Day to day processing and transaction of data is done in a database
Optimization	Warehouse is optimized to perform analytical processing on large data through complex queries	Optimized for speedy updating of data to maximize enhanced data access
Analysis	Dynamic and quick analysis of data is done	Transactional function is carried out, though analytic is possible but are difficult to perform due to complexity of normalized data

Kimball's Dimensional Modeling Approach

Kimball's dimensional modeling is a widely used approach for data modeling in data warehousing and business intelligence (BI) applications.

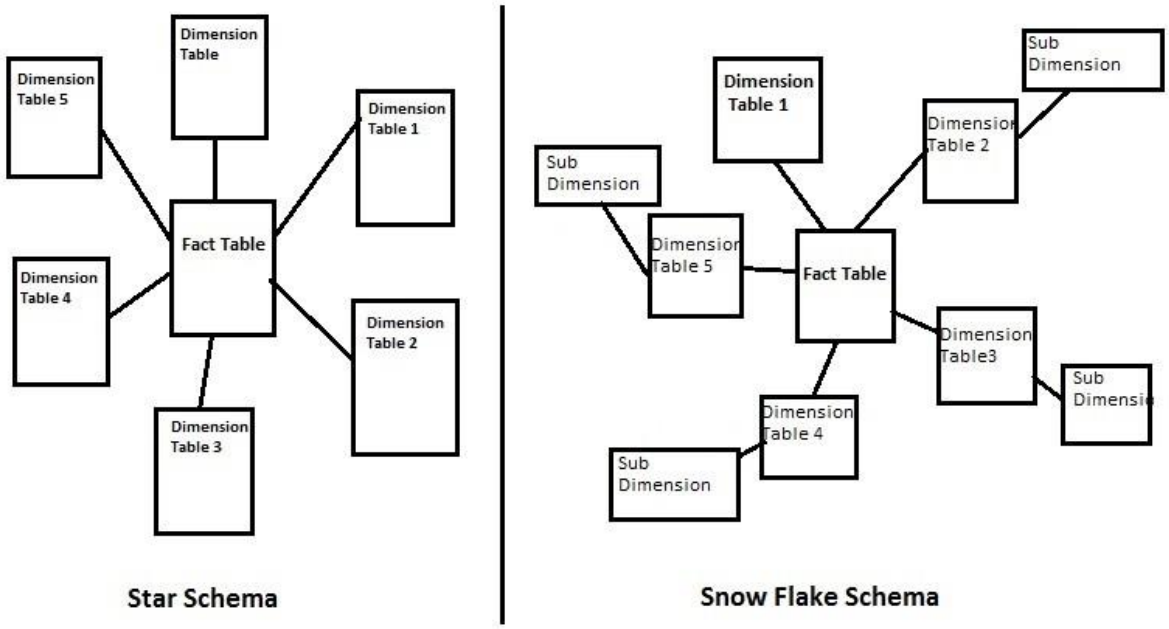
In Kimball's dimensional modeling approach, data is organized into two main types of tables: fact tables and dimension tables. Fact tables store the quantitative measures or metrics of interest, while dimension tables provide descriptive attributes or context for these measures. The relationships between fact tables and dimension tables are established through keys.

Star Schema

The central component of the star schema is the fact table, which holds the quantitative measures or metrics that are the focus of analysis. Surrounding the fact table are dimension tables

Snowflake Schema

The snowflake data modeling technique is an extension of the star schema. It aims to further normalize dimension tables to eliminate redundancy and improve data integrity.



Fact tables:

Fact Table: Facts are numeric measures that represent business metrics.

- Central table that contains measurable, numeric data (facts).
- Includes foreign keys referencing dimension tables.
- Example: Sales_Fact table with fields SaleID, DateID, ProductID, CustomerID, and Revenue.

Types of Facts -

- **Additive Facts:**
  - Can be summed across all dimensions.
  - Example: Sales_Amount can be totalled by date, region, or product.
- **Semi-Additive Facts:**
  - Can be summed across some dimensions but not others.
  - Example: Account_Balance can be totalled by region but not over time.
- **Non-Additive Facts:**
  - Cannot be summed across any dimension.
  - Example: Ratios or percentages like Profit_Margin.

Types of fact tables:

- **Transactional Fact Tables -** Transactional fact tables are designed to capture individual business events or transactions
- **Periodic Snapshot Fact Tables -** A periodic snapshot fact table captures a summary of data at a fixed interval, such as daily, weekly, or monthly.
- **Accumulating Snapshot Fact Tables -** An accumulating snapshot fact table tracks the progress of a transaction/event
- **Factless fact table** - is a link table that contains only foreign keys to dimensions, but no numeric facts.

Dimension Tables: Dimensions provide the descriptive context for facts, enabling users to analyze and filter data from various perspectives.

- Surround the fact table and store descriptive, textual information about the facts.
- Example: Product_Dimension with fields ProductID, ProductName, Category.

COMPARISON	FACT TABLE	DIMENSION TABLE
DATA	Contains large number of rows with consistent level of details.	Contains relatively small number of rows.
SHAPE	Tall and Skinny	Short & Squat
BUISNESS PURPOSE	Stores Observations or Events	Stores Business Entities
QUERY PURPOSE	Summarization	Filtering, Grouping
TABLE STRUCTURE	Contains Dimension Keys (Foreign Keys) and Numeric Measure Columns	Contains unique identifier (Primary Key) and Attribute Columns
COMMON USAGE	Sales Order, Stock Balance	Date, Products, People
COMMON DATA	Numbers	Words

Examples of Dimensions:

- Time Dimension:
  - Attributes: Date, Week, Month, Year.
- Product Dimension:
  - Attributes: ProductName, Category, Price.
- Customer Dimension:
  - Attributes: CustomerName, Region, Age.

Types of Dimensions:

- Conformed Dimensions:
  - Shared across multiple fact tables or data marts.
  - Example: Time_Dimension used in both Sales_Fact and Inventory_Fact.
- Junk Dimensions:
  - Combines unrelated attributes into a single dimension to reduce clutter.
  - Example: Flag_Dimension for binary indicators like NewCustomer, PromotionalSale.
- Degenerate Dimensions:
  - Dimension data stored in the fact table itself.
  - Example: OrderID in a sales fact table.
- Role-Playing Dimensions:
  - A single dimension table used in different contexts.
  - Example: Time_Dimension used as Order_Date and Ship_Date.
- Slowly Changing Dimensions (SCDs)
  Slowly Changing Dimensions (SCDs) are a methodology for handling changes in dimension data over time in a data warehouse while preserving the history of changes where required.

Types of SCDs

SCD Type 0 (Fixed Dimensions) - The dimension data is static and does not change over time.

SCD Type 1 (Overwrite) - When a change occurs, the old data is overwritten with the new data, and no history is maintained.

SCD Type 2 (Versioning) - Maintains full history by creating a new record for each change in dimension data.

Implementation Options:

1. **Row Versioning**: Add a Version column to identify different versions of the same dimension.
2. **Date Range**: Add StartDate and EndDate columns to define the validity period.

Example:

CustomerID	CustomerName	Region	StartDate	EndDate	CurrentFlag
101	John Smith	North	2023-01-01	2023-06-30	0
101	John Smith	South	2023-07-01	NULL	1

SCD Type 3 (Tracking Limited History) - Maintains limited history by adding columns to store previous values alongside the current value.

CustomerID	CustomerName	CurrentRegion	PreviousRegion
101	John Smith	South	North

SCD Type 4 (History Table) - Maintains history in a separate table, while the main dimension table holds only the current data

SCD Type 6 (Hybrid SCD - 1+2+3) - Combines elements of SCD Types 1, 2, and 3 to track both historical and current data while maintaining versioning.

Data Warehouse vs Data Lake vs Data Lakehouse

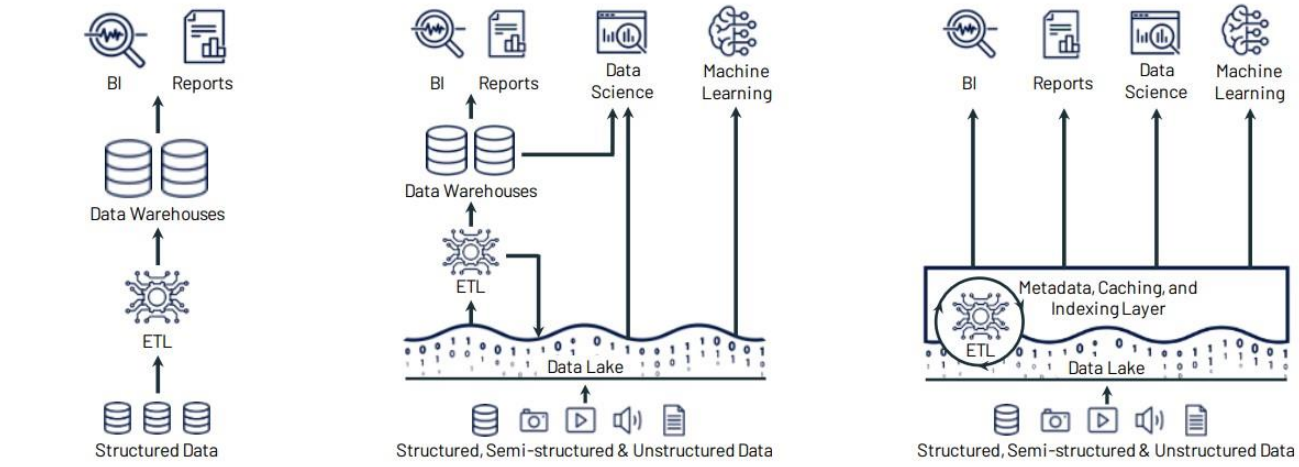
Difference between Lakehouse vs Delta lake vs Medallion Architecture

A lakehouse is a data platform that combines the features of data lakes and data warehouses into a single platform.

Delta Lake is an open-source storage layer that provides reliability, ACID transactions, and metadata management for tables within a lakehouse.

Medallion architecture is a data design pattern used to organize data within a lakehouse, typically into Bronze, Silver, and Gold layers, each representing a different level of data quality and structure

	Data warehouse	Data lake	Data lakehouse
ACID Support	YES	NO	YES
Schema on write	YES	NO	YES
Querying time	FAST	SLOW	FAST
Types of data	Structured	Structured. Semi/Un-structured.	Structured. Semi/Un-structured.
Cost	HIGH	LOW	LOW
Performance	HIGH	LOW	HIGH
Format	Closed, proprietary format	Open format	Open format
Scalability	No	Highly scalable	Highly scalable
Use Cases	BI, Reporting, Data analysis	Data scientists, Big Data	Unified: Big data, Data analysts, data scientists, machine learning engineers
Reliability	High quality, reliable data	Low quality, data swamp	High quality, reliable data
Ease of use	YES	NO	YES



(a) First-generation platforms. (b) Current two-tier architectures. (c) Lakehouse platforms.

Figure 1: Evolution of data platform architectures to today's two-tier model (a-b) and the new Lakehouse model (c).

Delta Lake:

Cheat-sheet: file:///C:/Users/geram/OneDrive/Desktop/Learning/Curated/delta-lake-cheat-sheet.pdf

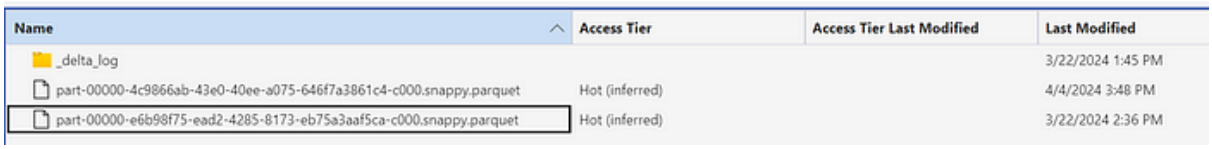
Delta lake is the storage layer within the lake house which uses cloud storages such as ADLS or S3 for storing the data in open source Delta format along with transaction logs which guarantees ACID transactions, time travel and history.

Delta Lake is an open-source storage framework that enables building a format agnostic Lakehouse architecture with compute engines including **Spark, PrestoDB, Flink, Trino, Hive, Snowflake, Google BigQuery, Athena, Redshift, Databricks, Azure Fabric and APIs for Scala, Java, Rust, and Python.** With Delta Universal Format aka UniForm, you can read now Delta tables with Iceberg and Hudi clients.

Components of Delta Lake:

1. **Storage Layer:** High performant, low-cost, scalable object storage in Azure, AWS or GCP.

2. **Delta Table:** Data in Parquet format gets stored in the cloud storage along with the Delta or transaction logs (JSON files).



A Delta table can be registered in the unity catalog metastore. Please refer to the below article to set up a unity catalog.

Delta Lake does come with a built-in processing engine called Delta Engine. This engine is part of Databricks and is designed to optimize the performance of queries against Delta Lake data

1.ACID Transactions:

Delta Lake guarantees ACID properties (Atomicity, Consistency, Isolation, Durability) for data operations, ensuring reliable and consistent data operations, even with concurrent reads and writes, according to a Databricks blog post.

2.Schema Evolution and Enforcement:

Delta Lake allows you to evolve the schema of your tables by adding, modifying, or deleting columns without rewriting existing data. Schema enforcement prevents data pipeline issues caused by schema mismatches, according to a Medium article.

3.Time Travel:

Delta Lake's time travel feature allows you to query previous versions of your data, enabling auditing, debugging, and data recovery.

4.Liquid Clustering:

This feature optimizes data layout for query performance by automatically rearranging data in a way that makes it more efficient to scan, according to Databricks documentation.

5.Z-Ordering (Work with OPTIMIZE. Works will for range based queries)

Delta Lake's Z-ordering feature **groups related data in the same files**, optimizing queries on multiple dimensions. This improves query performance by reducing the amount of data that needs to be scanned, according to a Databricks blog post.

Example: OPTIMIZE flights ZORDER BY (DayofMonth)

6. Auto-Compaction:

Automatically combines small files into larger ones, improving query performance and reducing storage costs

7. OPTIMIZE Command – Consolidate large number of small files into big files:

This command can be used to further consolidate and optimize data layout, especially beneficial for larger tables.

8. Data Skipping:

Delta Lake intelligently avoids reading unnecessary data files based on metadata, further improving query performance.

9. Unified Batch and Streaming:

Delta Lake supports both batch and streaming data processing, making it a versatile solution for various data workloads.

10. Scalable Metadata Handling:

Delta Lake efficiently manages metadata for large tables with billions of partitions and files.

11. Data Versioning

Delta Lake tracks every change to the table, providing a history of all versions, which is essential for auditing and recovery.

12. DML Operations (Merge):

Delta Lake's support for the MERGE operation allows you to efficiently perform updates and inserts based on specified conditions, simplifying incremental data updates

13. Access and Governance Controls:

Delta Lake provides robust access control and governance mechanisms.

14. Change Data Feed (CDF):

Delta Lake's CDF tracks changes to the data, enabling efficient data processing and synchronization.

15. VACUUM Command:

Removes outdated files from the Delta Lake directory, ensuring efficient storage utilization

Delta lake concurrency control:

Optimistic Concurrency Control (OCC)

Optimistic Concurrency Control is a method for handling concurrent transactions by assuming that conflicts are rare. Instead of locking resources to prevent conflicts, OCC allows transactions to proceed independently and checks for conflicts only at the time of commit. If a conflict is detected, the transaction is aborted and must be retried.

Key Principles of OCC:

1. No Locking: Transactions do not lock resources, allowing for high concurrency.
2. Conflict Detection: Conflicts are detected at commit time by comparing the transaction's changes with the current state of the data.
3. Retry Mechanism: If a conflict is detected, the transaction is aborted and must be retried.

How Delta Lake Implements OCC:

Isolation Levels in Delta Lake

Delta Lake provides serializable isolation, the highest level of isolation in database systems. This means that the result of executing multiple concurrent transactions is the same as if they were executed one after the other in some serial order.

How Isolation is Achieved:

- **Read Isolation:** Readers always see a consistent snapshot of the data as of the start of their transaction. This is achieved by reading the transaction log up to the version number at which the transaction started.
- **Write Isolation:** Writers are isolated from each other by checking for conflicts at commit time. If two transactions attempt to modify the same data, one will succeed, and the other will fail and need to retry.

Types of Conflicts in Delta Lake:

a. Write-Write Conflicts:

- Scenario: Two transactions attempt to modify the same file or partition.
- Example:
- Transaction A starts and reads version 1 of the table.
- Transaction B starts and reads version 1 of the table.
- Transaction A modifies file X and commits, creating version 2.
- Transaction B attempts to modify file X but detects that the table has been updated to version 2. Transaction B is aborted.

b. Append-Only Conflicts:

- Scenario: A transaction attempts to append data to a table while another transaction deletes or modifies the same data.
- Example:
- Transaction A starts and reads version 1 of the table.
- Transaction B deletes some rows and commits, creating version 2.
- Transaction A attempts to append data but detects that the table has been updated to version 2. Transaction A is aborted.

c. Schema Conflicts:

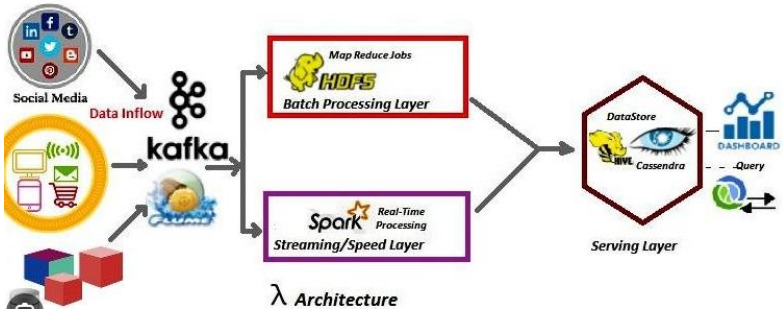
- Scenario: A transaction attempts to modify the schema of a table while another transaction is reading or writing data.
- Example:
- Transaction A starts and reads version 1 of the table.
- Transaction B modifies the schema and commits, creating version 2.
- Transaction A attempts to write data but detects that the schema has changed. Transaction A is aborted.

Lambda vs Kappa vs Medallion Architectures

Lambda, Kappa, and Medallion are distinct data processing architectures designed to handle big data

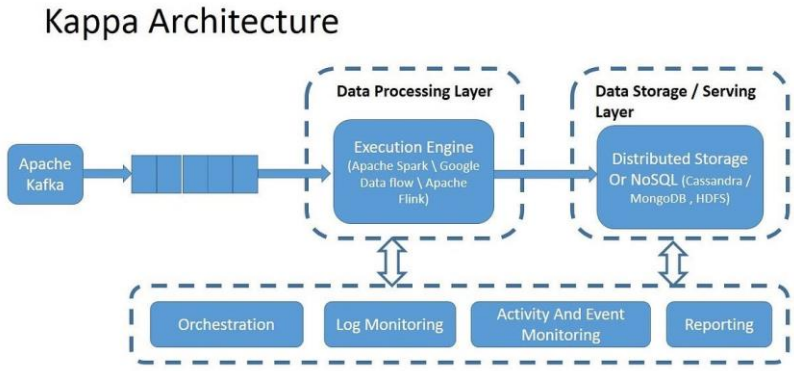
Lambda

Lambda Architecture is a data-processing architecture designed to handle massive quantities of data by taking advantage of both batch and stream-processing methods. It consists of three layers: batch processing (cold path) , real-time/speed (hot path) processing, and a serving layer for responding to queries.



Kappa

Kappa architecture is true event-base. It differs from the Lambda architecture in a sense that it avoids maintaining complexities of two different codebases between the batch and speed layers



Medallion architecture – Data quality is improved in each layer (Bronze – Silver – Gold)

Medallion Architecture is a **data design pattern** used to logically organize data in a **lakehouse**. It aims to incrementally and progressively improve the structure and quality of data as it flows through each layer of the architecture (from Bronze ⇒ Silver ⇒ Gold layer tables).

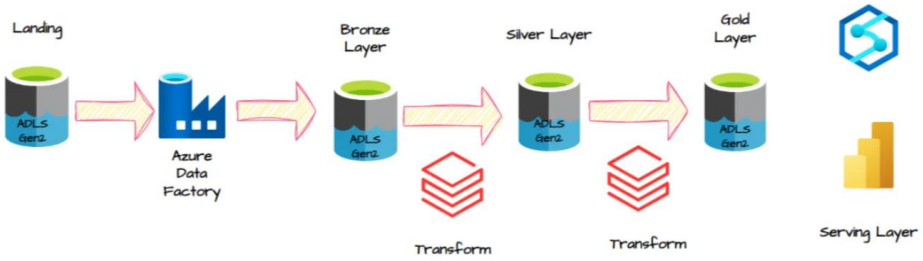
Advantages: ACID, Data quality, Data Lakehouse benefits

BRONZE – RAW/Unstructured. In Source Format. Maintain History

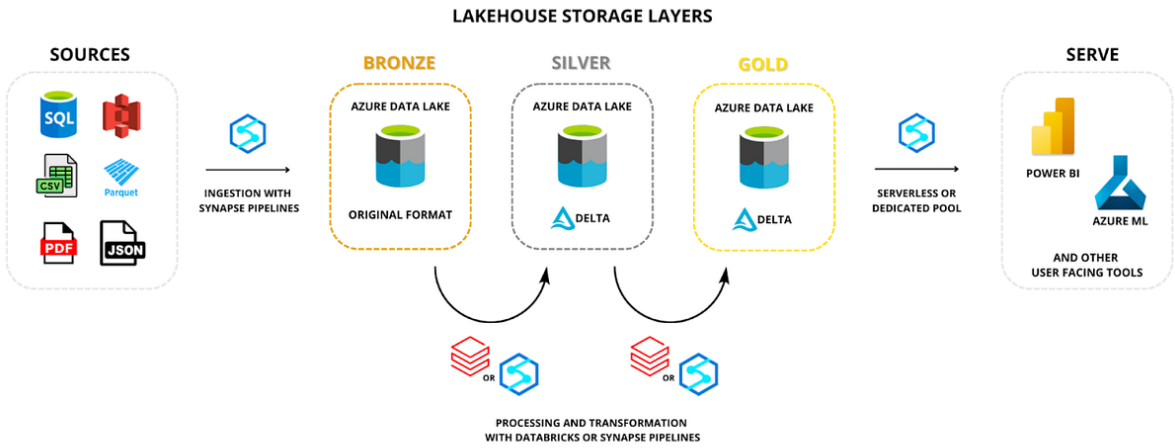
Silver – Structured (delta format). Refined. Filtered. Joined. Transformed.

Gold – Aggregated. Highly refined for BI/AI-ML needs

Medallion Architecture sample 1



Medallion Architecture sample 2



Lambda vs Kappa Comparison:

Aspect	Lambda Architecture	Kappa Architecture
Processing Model	Combines batch and real-time processing.	Focuses solely on stream (real-time) processing.
Layers	Batch layer, Speed layer, Serving layer.	Single pipeline for both real-time and historical data.
Complexity	Higher due to managing separate batch and	Simpler, as it uses only one stream processing layer.
Fault Tolerance	Fault-tolerant, as batch processing ensures	Fault-tolerant with real-time processing but depends on
Use Case	Suitable for both real-time and batch	Best for real-time processing; batch processing is less
Data Reprocessing	Batch layer allows accurate reprocessing of	Reprocessing is done by replaying the stream in real-time.
Latency	Higher latency in batch processing; low	Low latency for all data due to stream processing.
Accuracy	layer offers immediate but less accurate results.	Provides consistent results, but may not match the accura of dedicated batch processing.

Data Lake vs Medallion Architecture

Aspect	Traditional Data Lake	Medallion Architecture
Foundation	Generic files in storage (CSV, Parquet, etc.)	Built on Delta Lake, offering ACID transactions, schema enforcement, and versioning
Data Quality	Handled via external tools/scripts	Enforced natively via Delta constraints, expectations, and Delta Live Tables (DLT)
Lineage & Observability	Not native; added via external tools	Native support in Databricks via Unity Catalog, DLT, and data lineage tracking
Modularity	Often monolithic pipelines	Designed with modular, scalable modular pipeline patterns (e.g., CDC, streaming, batch)
Automation & CI/CD	Manual or semi-automated	Supports full CI/CD pipelines, testing, and alerting with Databricks tools
Governance	Difficult to enforce consistently	Built-in with Unity Catalog, RBAC, audit logs, and data classification

Apache Spark

Apache Spark is a open-source, in-memory, distributed computing framework designed for big data processing in parallel. It supports languages such as Pythan, Scala, Java, SQL and R. 100 times faster than Mapreduce

Unified means it offers both batch and stream processing

Spark has 3 components on high level:

- Low level API - Consists of RDDs Resilient Distributed Datasets & Distributed Variables
- Structured APIs – Built on top of RDD, Highly optimized. Dataframes, Datasets and Spark SQL
- Libraries & Ecosystem – Structured streaming, graph etc

Note: Shuffle is the boundary that divides jobs into stages

Transformations: Transformations in Spark involve **operations that modify Data structures (and create new data frame)**, they are lazily evaluated.

Narrow transformations process data within a single partition without requiring data to be shuffled or redistributed across partitions. **Ex:** SELECT, WITHCOLUMN, FILTER, UNION, DISTINCT, ORDER BY,

Wide transformations require data to be shuffled or redistributed across partitions.
Exa: Sort() or OrderBy() both are same. GroupBy(), Join(), Repartition(), distinct(), dropduplicates(),

Note: Wide transformations leads to shuffles (and stages in job).

Actions are **operations that trigger the execution of transformations** and return a value to the driver program or write data to an external.

Common Actions:

- count(): Gets the total number of rows in a DataFrame or RDD.
- collect(): Retrieves all rows of a DataFrame or RDD as a list.
- show(): Displays a preview of the first few rows of a DataFrame.
- save(): Writes the DataFrame to external storage (e.g., a file).
- take(n): Returns the first n rows of a DataFrame or RDD.
- first(): Returns the first row of a DataFrame or RDD.

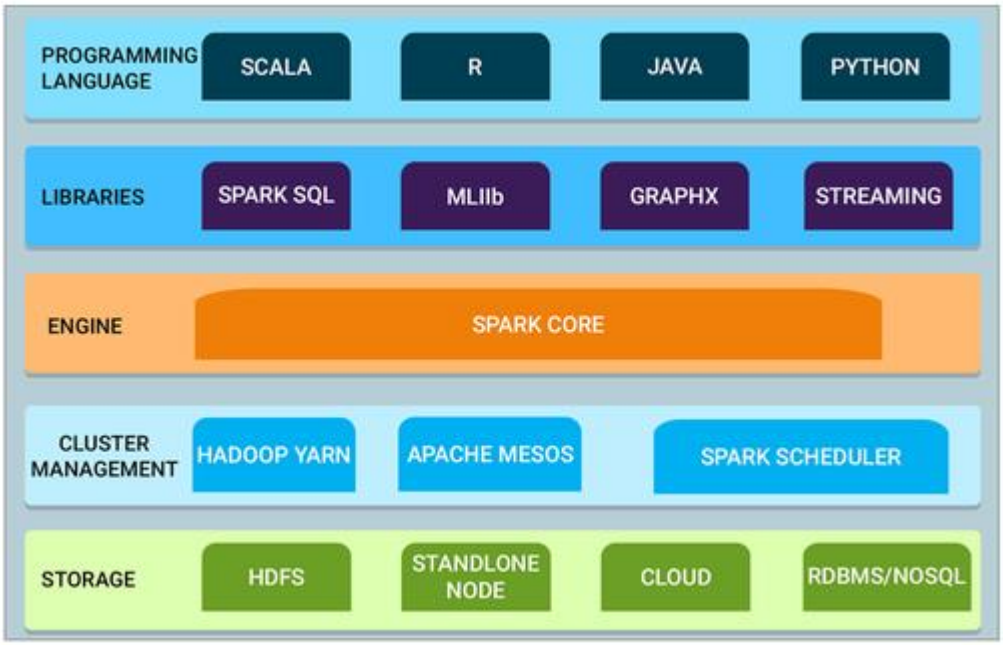
Driver process is known as SparkSession. 1 session per 1 sparkApplication.

DataFrames are most commonly used Structured API organized in table (row, column) format.

We can **Cascade multiple transformations** in one go. Ex: select.filter.orderby().show
They create lineage or Directed acyclic graph (DAG) of operations.

In Databricks, a Spark application's workflow involves a driver that manages the overall execution and executors that perform the actual computation. The driver splits the application's work into smaller tasks and distributes them to executors, which process the data in parallel and return results back to the driver.

Spark components:



- **Spark Core**: The underlying engine that supports all Spark functionalities.
- **Spark SQL**: For structured data processing.
- **Spark Streaming**: For real-time data processing.
- **MLlib**: Machine learning library.
- **GraphX**: For graph processing.

- **Languages Supported**: Java, Scala, Python, and R.

- **Deployment Modes**: Hadoop YARN, Apache Mesos, Kubernetes, Standalone cluster manager.

Spark Core component responsibility includes: managing memory, handling fault recovery, scheduling and distributing tasks, and interacting with storage systems

A SparkSession acts as a connection manager between the user and a Spark cluster. It's the entry point for working with Spark core functionalities , providing a unified interface for accessing various Spark functionalities, including DataFrame and Dataset APIs, as well as SQL queries.

This means you can load data, perform transformations, execute SQL queries, and interact with Spark's APIs all through this one object.

Spark Key Components:

1. **RDDs (Resilient Distributed Datasets)**:
    - Low-level abstraction.
    - Enables fault tolerance using lineage graphs.
    - **Operations**:
        - **Transformations**: map(), filter(), flatMap(), etc.
        - **Actions**: collect(), reduce(), count(), etc.
2. **DataFrames**:
    - High-level abstraction over RDDs with schema support.
    - Optimized for SQL-based operations.

3. **Datasets**:
    - Typed DataFrames (Scala/Java only).
    - Combines functional programming with SQL optimisations.

RDD vs Data Frames vs Datasets

In Apache Spark, an RDD (Resilient Distributed Dataset) is a fundamental data structure representing a distributed collection of objects. It's an immutable, partitioned collection of elements that can be operated on in parallel across a cluster. RDDs are fault-tolerant, meaning they can recover from node failures by tracking the lineage of operations.

Feature	RDD	DataFrame	Dataset
Immutable	Yes	Yes	Yes
Fault Tolerant	Yes	Yes	Yes
Type-Safe	Yes	No	Yes
Schema	No	Yes	Yes
Execution Optimization	No	Yes	Yes
Optimizer Engine	N/A	Catalyst Engine	Catalyst Engine
API Level for manipulating distributed collection of data	Low	High	High
language Support	Java, Scala, Pyt	Java, Scala, Python, R	Java, Scala

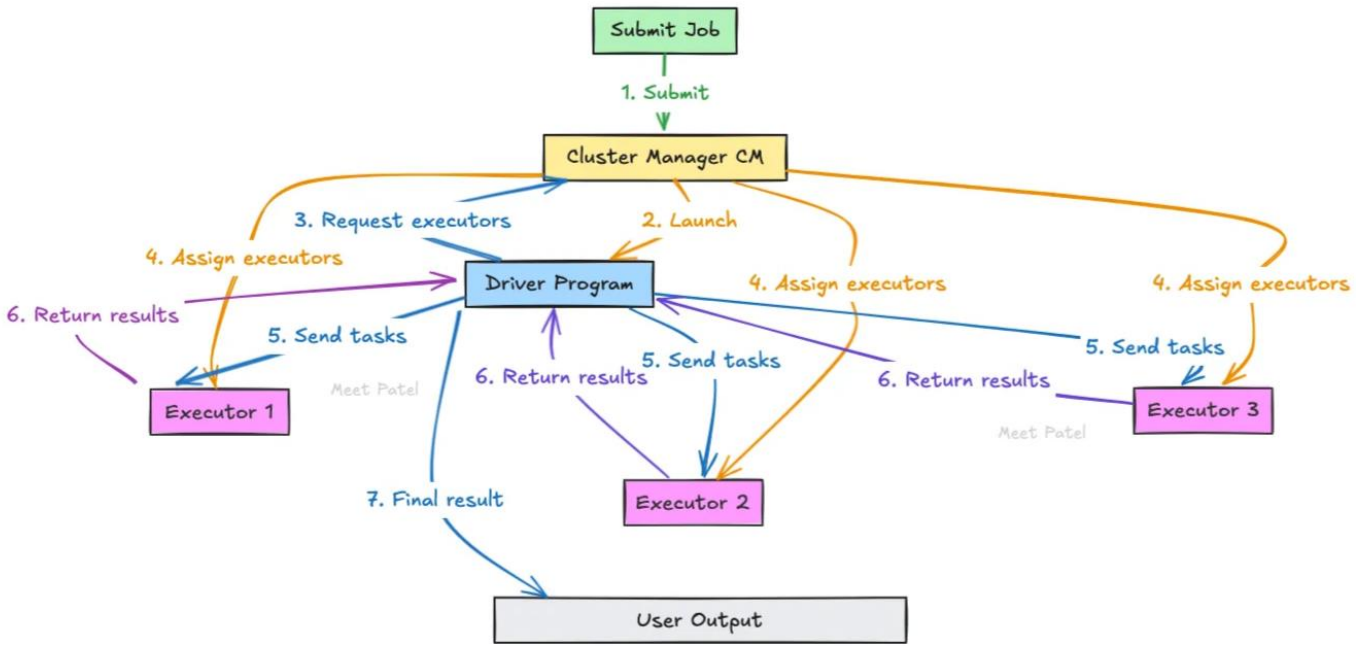
In dynamically typed languages, the type of a variable is not declared explicitly

Spark Datasets, **a type-safe structured API**, are **primarily designed for statically typed languages like Scala and Java**. Python, being a dynamically typed language, doesn't have the same compile-time type checking capabilities, making it unsuitable for Datasets

Python is considered a dynamically typed language, meaning the type of a variable is determined during program execution

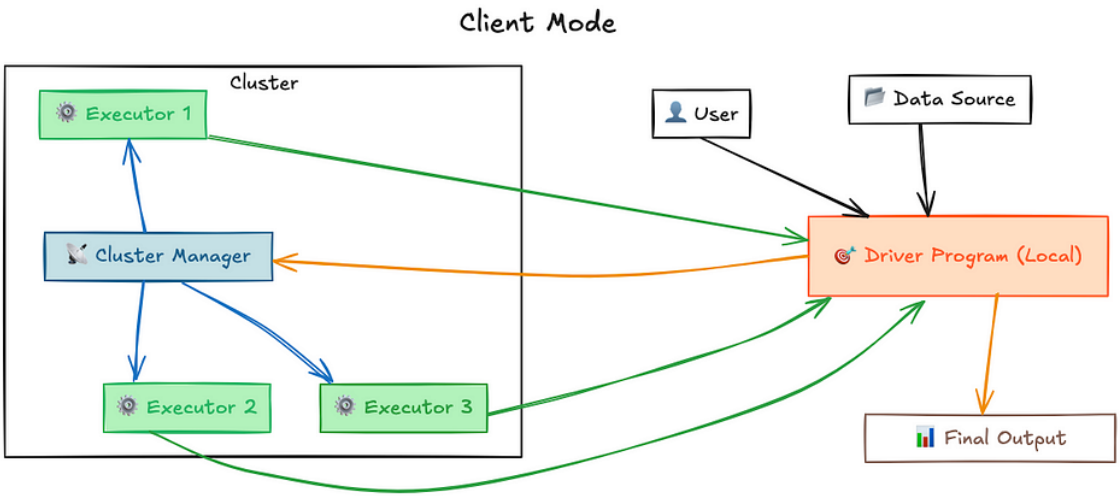
Spark Application Workflow

1. **Submit the Job**: The job is submitted to the **cluster manager**.
2. **Cluster Manager Starts the Driver**: The cluster manager **starts the driver program** if it isn't already running.
3. **Driver Creates a Plan and Asks for Executors**: The driver reads the code, creates a **plan for the job**, and sends a request for **executors** to the cluster manager.
4. **Cluster Manager Assigns Executors**: The cluster manager assigns **executors** on worker nodes based on available resources.
5. **Executors Process Data and Report to Driver**: The executors **run their tasks** in parallel and **send results back to the driver**.
6. **Driver Gathers Results**: Once all tasks are complete, the driver combines the results and presents them to the user.


Submit Job
1. Submit
Cluster Manager CM
3. Request executors
2. Launch
4. Assign executors
Driver Program
4. Assign executors
4. Assign executors
6. Return results
5. Send tasks
5. Send tasks
5. Send tasks
6. Return results
6. Return results
Executor 1
Executor 3
Executor 2
7. Final result
User Output

Apache Spark Deployment Modes

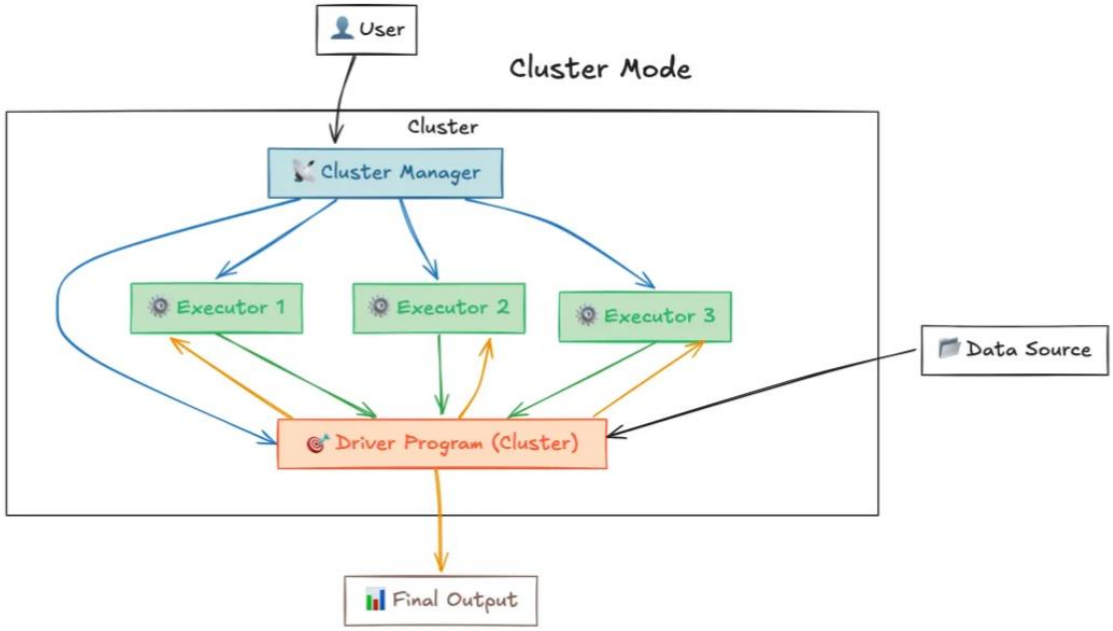
*** Client Mode (Driver in local machine)**



***Cluster Mode: (Driver in Cluster)**

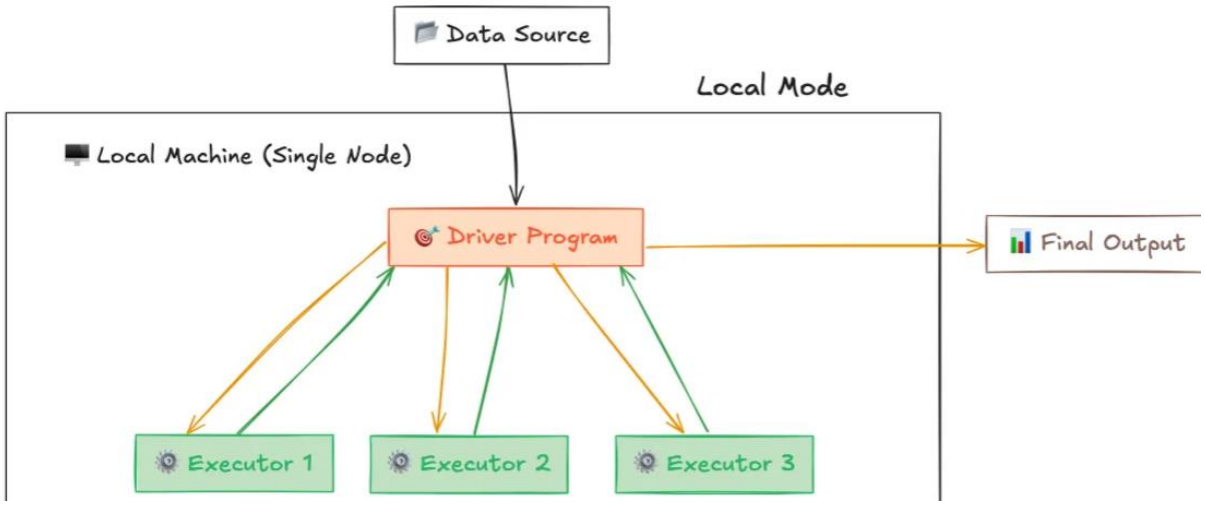
Once you submit a job in Cluster-Mode, the Cluster Manager assigns the driver program to one of the worker nodes in the cluster

Cluster Mode's driver program runs on a separate machine within the cluster, not your local machine. This means the Cluster Manager is responsible for scheduling jobs, allocating resources, and ensuring everything runs smoothly.



Local Mode (Drivers and Executors in Local machine)

In Spark's local mode, there is no actual cluster manager involved. The driver (the main Spark program) runs on the client machine, and all tasks are executed locally on the same machine

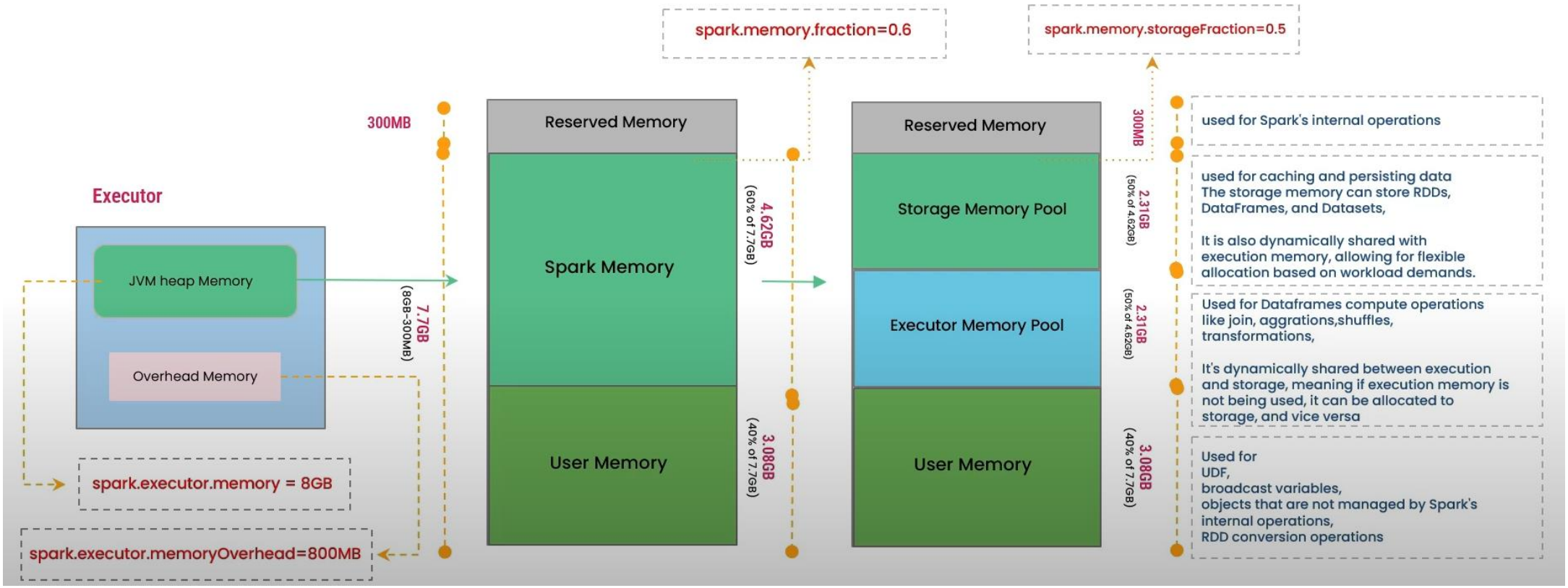


Spark cluster managers – Resource allocation and prioritize

Concept to remember: Preemption

- Standalone
  - Default scheduler FIFO (First In, First Out)
  - Optional: Faire
  - **Preemption:** Unlike YARN or Mesos, Spark Standalone does not support preemption out of the box. Once resources are allocated to a job, they are not preempted or reallocated unless the job finishes or is manually killed.
- YARN
  - - **Capacity Scheduler:** Divides cluster resources into multiple queues, each with a guaranteed capacity (e.g., a percentage of total resources). Jobs within each queue are scheduled based on priority or FIFO, depending on configuration.
  - **Fair Scheduler:** Allocates resources to jobs so that all jobs receive a fair share of resources over time. Jobs with higher priority are given more resources, but overall fairness is maintained across the cluster.
  - **FIFO**
- Kubernetes - for running Spark applications in containerized fashion.

Spark Memory Management



In Apache Spark, **execution memory (top priority)** can release memory for storage memory, but only up to a certain threshold. The execution memory is used for computations like **shuffles, joins, and aggregations**, while storage memory is used for **caching data**.

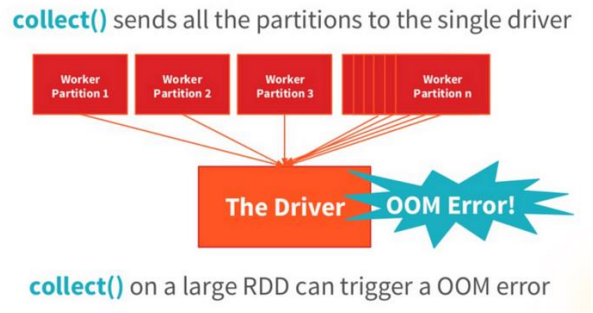
When no execution memory is used, storage can occupy all available memory, and vice versa. However, execution can evict storage memory if necessary, but only until the total storage memory usage falls below a threshold.

OOM error scenarios and solutions:

Driver Memory Errors:

Case 1: Collect Operation

The collect operation in Spark retrieves data from distributed workers and consolidates it on the driver. This can lead to OOM errors if the collected data is too large to fit into the driver's memory



Solution: Increase driver memory **spark.driver.memory** and increase spark.driver.maxResultSize

Case 2: Broadcast Join

Broadcast joins are useful for optimizing joins when one side of the join is small enough to fit in memory. However, if the broadcasted data is too large, it can exhaust driver memory:

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import broadcast

spark = SparkSession.builder.appName("BroadcastSenerio").getOrCreate()
large_df = spark.read.parquet("large_data_file.parquet")
small_df = spark.read.parquet("small_data_file.parquet")
joined_data = large_df.join(broadcast(small_df), "common_column")
```

Solution:

Increase driver memory and Adjust Broadcast Threshold **spark.sql.autoBroadcastJoinThreshold**

Executor Memory Errors:

Case 1: Inefficient Queries

Selecting all columns from a large table without filtering can lead to memory issues due to the in-memory state needed for each column:

Solution: Filter data as much as possible and use partition pruning.

Case 2: Incorrect Configuration

Each Spark application has unique memory requirements. Misconfigured settings can lead to OOM errors. Properly configuring executor memory and overhead is crucial:

--conf "spark.executor.memory=12g"
--conf "spark.yarn.executor.memoryOverhead=2048"

Solution: Adjust memory settings based on the application's requirements.

Addressing OOM Errors: Practical Configurations

- **For Executor Memory:**

  --conf "spark.executor.memory=12g"
  --conf "spark.yarn.executor.memoryOverhead=2048"
  --conf "spark.executor.pyspark.memory=2g"

- **For Driver Memory**

  --conf "spark.driver.memory=4g" --conf "spark.driver.maxResultSize=2g"

- **Managing Concurrency and Partitions:**

  --conf "spark.default.parallelism=100"
  --conf "spark.sql.shuffle.partitions=200"
  --conf "spark.executor.cores=4"

- **Using Off-Heap Memory:**

  --conf "spark.memory.offHeap.enabled=true"
  --conf "spark.memory.offHeap.size=4g"

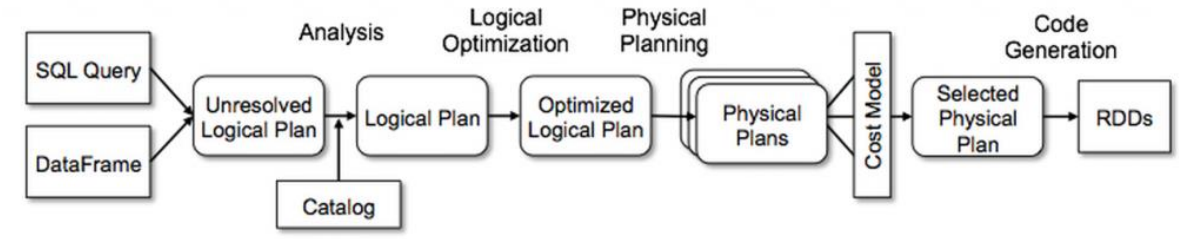
Spark can also allocate memory off-heap, outside the JVM heap space, which can reduce GC overhead. This memory is used for off-heap storage (like **caching**) and is managed by Spark's memory manage.

Using off-heap memory can improve performance as it reduces the frequency of garbage collection, as off-heap memory is not subject to the garbage collector's automatic reclamation.

AQE (Adaptive Query Execution)

AQE **dynamically perform coalescing shuffle partitions, switching join strategies** (like broadcast vs. shuffle), and handling skewed joins efficiently.

Catalyst Optimizer



Predicate Pushdown: This technique pushes down filter predicates to the data source to reduce the amount of data processed.

Projection Pushdown: This technique minimizes the amount of data transfer between data source and the Spark engine by eliminating unnecessary fields (cols) from the table scanning process.

Column Pruning: This technique removes unnecessary columns from the operations to reduce the amount of data processed.

Join Reordering: This technique reorders the join operations to minimize the amount of data shuffled between nodes.

Constant Folding: This technique evaluates constant expressions at compile time to reduce the amount of computation performed at runtime.

Subquery Optimization: This technique optimizes subqueries to reduce the amount of data processed.

Logical planning:

Optimizing the Logical Plan: Once the logical plan is resolved, Catalyst applies a series of optimization rules to improve its efficiency without changing its meaning.

- **Predicate Pushdown:** Moving filters (WHERE clauses) as early as possible in the plan to reduce the amount of data processed in subsequent steps.
- **Column Pruning:** Removing columns that are not needed for the final result, reducing the data read and shuffled.
- **Constant Folding:** Evaluating constant expressions at compile time instead of runtime.

  *df.selectExpr("col1 + 5 * 2 as result").show() # 5 * 2 is calculated to 10 at compile time*

  df.selectExpr("'abc' + 'def' as result").show() # "abc" + "def" is evaluated to "abcdef"

  df.selectExpr("CASE WHEN 2 > 1 THEN 1 ELSE 0 END as result").show() # The condition 2 > 1 is always true, resulting in 1

- **Join Reordering:** Finding the most efficient order for joining multiple tables.
- **Eliminating Common Subexpressions:** Identifying and removing redundant computations. These optimizations are applied repeatedly until the plan reaches a fixed point where no further improvements can be made.

Physical Plan Generation and Selection: This is where Catalyst translates the optimized logical plan into one or more physical execution plans and then selects the most efficient one.

- **Cost-Based Optimization (CBO):** In more recent versions of Spark, Catalyst can use statistics (like table size, distinct values, data distribution) to estimate the cost of different physical plans and choose the one with the lowest estimated cost.
- **Physical Plan Generation:** Creating multiple alternative physical execution plans based on the optimized logical plan. For example, a logical "Join" operator could be translated into a "Sort-Merge Join," "Broadcast Hash Join," or "Shuffle Hash Join" physical operator. The choice depends on factors like data size, distribution, and available memory.

Project Tungsten (memory optimization) includes a code generation framework that translates parts of Spark's physical execution plan into highly optimized bytecode.

Top 5 common Spark performance issues

- **Spill, Skew, Shuffle, Storage, and Serialization**
- **1. Spill:**

  When Spark's memory is insufficient, it writes **temporary data to disk (spill)** to avoid out-of-memory errors. This can slow down processing due to the increased I/O operations.
- **2. Skew:**

  Data skew occurs when **some partitions have significantly more data** than others, leading to uneven workload distribution among executors. This can cause some tasks to finish much faster than others, resulting in idle resources and increased overall job duration.
- **3. Shuffle:**

  Shuffles occur during transformations like joins or aggregations when Spark needs to **redistribute data across executors**. Excessive shuffling can increase network I/O and slow down job execution.
- **4. Storage:**

  The way data is stored on disk (e.g., using appropriate **data formats like Parquet**) and how it's persisted (e.g., **caching**) can significantly impact read/write speeds.
- **5. Serialization:**
  Serialization is the process of converting objects into a byte stream to be transmitted across the network for distributed processing. **Poor serialization techniques** can lead to increased serialization/deserialization overhead

Top 5 Performance Optimization Technique to resolve most common performance issues

1. Adaptive Query Execution (AQE)

Issue Solved: Poor join strategies and skewed data
Optimization:

- Spark 3.5 automatically adjusts join strategies, partition sizes, and parallelism at runtime.
- Enables dynamic switch from sort-merge join to broadcast join.
- Helps reduce shuffles and handle data skew more efficiently.

Enable it (usually on by default):
spark.conf.set("spark.sql.adaptive.enabled", "true")

2. Broadcast Hash Join

Issue Solved: Slow performance on joins with small dimension tables
Optimization:

- Small tables (e.g., lookup/dim tables) are broadcasted to all worker nodes, eliminating shuffles.
- Much faster than sort-merge joins for small datasets.

Manually broadcast if not auto-detected:

```
from pyspark.sql.functions import broadcast
df1.join(broadcast(df2), "key")
```

3. Optimize File Sizes in Delta Tables

Issue Solved: Too many small files (small file problem) hurt performance
Optimization:

- Use Auto Optimize and Auto Compaction or run OPTIMIZE manually.
- Improves scan and shuffle performance.

```
OPTIMIZE delta.`/path/to/table`
```

Or enable auto-optimize on Delta table:

```
ALTER TABLE my_table SET TBLPROPERTIES (
  'delta.autoOptimize.optimizeWrite' = 'true',
  'delta.autoOptimize.autoCompact' = 'true'
)
```

4. Caching & Persistence

Issue Solved: Repeated recomputation of expensive transformations
Optimization:

- Use .cache() or .persist() when a DataFrame is reused multiple times.
- Avoid caching intermediate results that are used only once.

df.cache()

df.count() # triggers caching

5. Partition Pruning & Predicate Pushdown

Issue Solved: Reading unnecessary data leading to slow queries
Optimization:

- Use partitioned tables effectively and filter on partition columns.
- Spark will skip unnecessary partitions (partition pruning).
- Pushes filters to data source for efficiency.

df = spark.read.format("delta").load("/data/sales")
df_filtered = df.filter("country = 'US'")

These five techniques cover execution planning (AQE, joins), data layout (file size, partitioning), and caching, which **together resolve over 80% of common performance issues** in production Spark workloads.

Actions vs Transformations:

Transformations in PySpark DataFrames are operations that modify the structure or content of the DataFrame, creating a new DataFrame in the process. These operations are lazy, meaning they don't execute immediately

Actions in PySpark execute the transformations and either send the results back to the user or store the results.

Common Actions:

- count(): Gets the total number of rows in a DataFrame or RDD.
- collect(): Retrieves all rows of a DataFrame or RDD as a list.
- show(): Displays a preview of the first few rows of a DataFrame.
- save(): Writes the DataFrame to external storage (e.g., a file).
- take(n): Returns the first n rows of a DataFrame or RDD.
- first(): Returns the first row of a DataFrame or RDD.

Common Transformation:

- select(): Selects specific columns in a DataFrame
- limit(): Limits the number of rows in a DataFrame.
- filter(): Selects elements that meet a specific condition, creating a new RDD or DataFrame.
- join(): Combines two RDDs or DataFrames based on a common key.
- distinct(): Removes duplicate elements.
- union(): Combines two RDDs or DataFrames.
- sortByKey(): Sorts an RDD by its keys.
- groupByKey(): Groups elements with the same key, useful for aggregation.
- withColumnRenamed(): Renames a column in a DataFrame.
- dropDuplicates(): Removes duplicate rows from a DataFrame.
- sample(): Creates a random sample of a DataFrame.

Comparisons:

Feature	Databricks Cache (Delta Cache)	Apache Spark Cache
Caching Type	Disk-based caching	RAM-based caching
Handling Large Datasets	More efficient for large datasets, avoids memory pressure	May experience memory issues with large datasets
GC Overhead	Reduces GC overhead by using disk storage	GC overhead can slow down execution with large data in memory
Deserialization Overhead	Avoids deserialization overhead for faster access to data	Deserialization overhead when reading from disk
Partial Caching Support	Supports partial caching, only caches relevant parts of the data	Does not support partial caching, caches the entire dataset
Cache Management	Automatically evicts stale data and updates when needed	Requires manual cache management for stale data

Spark Cache vs Persist

Feature	cache()	persist()
Purpose	Store data in memory for quick access	Store data in memory or disk with flexible storage levels
Default Storage	Memory only (MEMORY_ONLY)	Flexible, can be memory only, memory and disk, etc.
Storage Levels	Limited to memory	Multiple storage levels, including memory and disk
Convenience	Shorthand for persist() with default memory storage	More control over storage location and type
Flexibility	Less flexible	More flexible for different performance needs

Repartition vs Collesce

Feature	repartition()	coalesce()
Purpose	Redistribute data across partitions	Reduce the number of partitions
Shuffle?	Always, for even distribution	Sometimes, to avoid full shuffle
Increase/Decrease	Both (increase or decrease)	Only decrease
Partition Size	Generally even (balanced)	May be uneven (unbalanced)
Use Cases	Balancing data, increasing parallelism	Optimizing after transformations

GroupBy vs ReduceBy

Feature	groupByKey()	reduceByKey()
Purpose	Group values for each key into an iterable of values	Combine values for each key using a reduce function
Data Shuffling	Shuffles all values for each key, potentially causing performance issues	Performs local reductions on each partition before shuffling, minimizing data movement
Memory Usage	Can be memory-intensive as it collects all values for each key	Less memory-intensive as it reduces values locally before shuffling
Aggregation	Groups values, doesn't allow custom aggregation logic	Performs a fixed, predefined aggregation (e.g., sum, average)
Use Cases	When you need to group data and process all values together	When you need to aggregate values based on keys (e.g., calculate sums, averages)
Function Signature	rdd.groupByKey()	rdd.reduceByKey(func)

Broadcast Join vs Shuffle Join

Aspect	Normal Join (Shuffle Join)	Broadcast Join
Operation	Both datasets are shuffled across the cluster based on the join keys.	The smaller dataset is broadcasted to all nodes, avoiding shuffling of the larger dataset.
Use Case	Suitable for large datasets where neither side is small enough to fit in memory.	Ideal when one dataset is significantly smaller than the other.
Performance	Can be slower due to extensive data shuffling; network I/O and disk writes can be bottlenecks.	Typically faster as it minimizes data shuffling and reduces network I/O and disk write overhead.
Scalability	Scales well with large datasets but can be inefficient if not optimized (e.g., by partitioning).	Works well for joins with small datasets; not suitable for large-to-large dataset joins.
Data Movement	High, as data needs to be shuffled across nodes.	Low, as only the smaller dataset is broadcasted to each node.
Memory Consideration	Less memory-intensive as data is shuffled and not stored entirely in memory.	Requires sufficient memory to hold the smaller dataset in the memory of each worker node.
Suitability for Skewed Data	May struggle with data skewness as shuffling can lead to uneven data distribution across nodes.	Handles data skew more effectively as it doesn't require shuffling the larger dataset.
Flexibility	More flexible in handling different types of joins, including complex ones.	Limited by the size of the smaller dataset and not suitable for all join types (e.g., full outer joins).
Network Load	Higher due to the shuffling of data across the cluster.	Lower as it minimizes the movement of data across the network.
Optimization Needs	Often requires careful optimization like partitioning to enhance performance.	Requires less optimization, mainly ensuring the smaller dataset fits in memory.

Parquet vs Delta

Feature	Parquet	Delta Tables
File Format	Columnar file format	Based on Parquet, with added metadata and transaction log
Data Integrity	No native transaction support	Supports ACID transactions (Atomicity, Consistency, Isolation, Durability)
Schema Evolution	Supports schema evolution but without strict enforcement	Enforces schema integrity with controlled evolution
Time Travel	No built-in time travel	Supports time travel to access previous versions of data
Data Updates	No in-place updates	Supports in-place updates (upsert, merge, delete)
Performance	Optimized for read-heavy workloads, query performance	Offers performance optimizations like file compaction and Z-Ordering
Use Cases	Static or batch datasets, data lakes	Transactional workloads, real-time analytics, managing data lakes
Integration	Broad compatibility across the Hadoop ecosystem	Seamless integration with the Spark ecosystem

CSV vs Parquet:

Feature	CSV	Parquet
Storage	Row-based	Columnar
Data Format	Plain text (human readable)	Binary (machine readable)
File Size	Larger	Smaller (due to compression)
Performance	Slower for large datasets, especially when querying specific columns	Faster, especially for large datasets and analytical workloads
Compression	Limited compression options	Advanced compression options (e.g., Snappy, LZ4)

Schema	No built-in schema, requires external definition	Includes schema metadata
Data Types	Limited to basic types, manual data type detection	Supports a wide range of data types, including nested and complex structures
Querying	Requires reading entire rows, even if only specific columns are needed	Optimized for querying specific columns, can skip irrelevant data
Data Integrity	Lacks built-in validation, potential for data inconsistencies	Includes schema metadata for data integrity and consistency
Ease of Use	Simple and widely supported, easy to read and write manually	Requires libraries and tools for reading and writing, not ideal for human editing
Use Cases	Smaller datasets, quick manual analysis, data exchange	Large-scale data processing, analytical workloads, data lakes

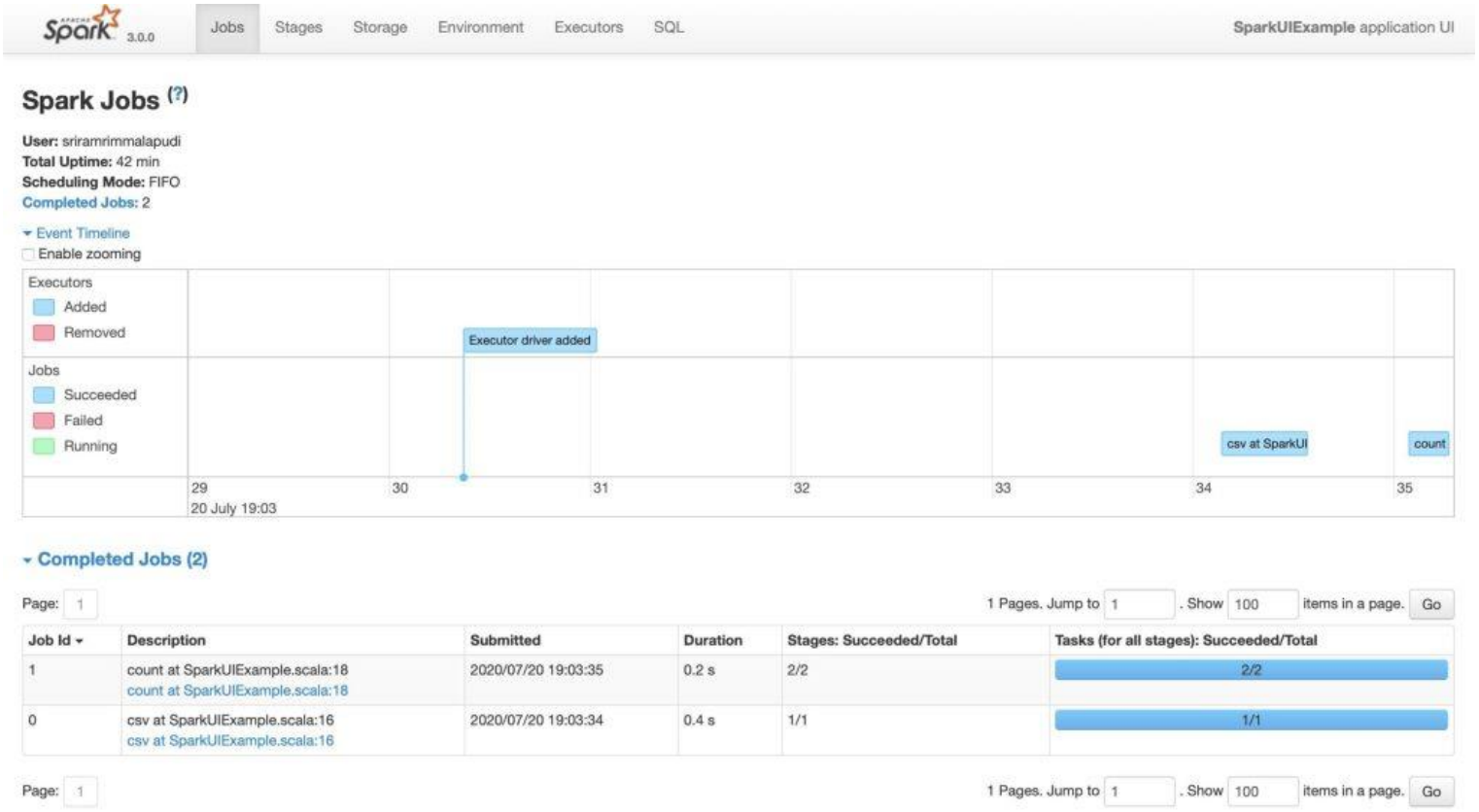
Storage levels

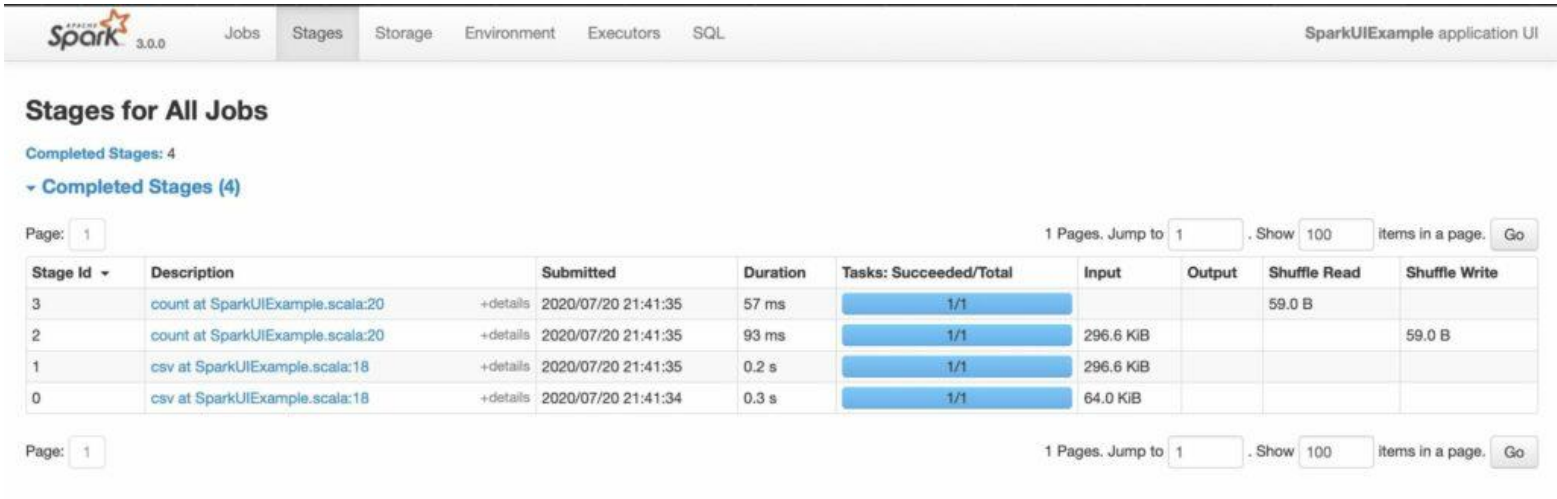
Storage Level	Meaning
MEMORY_ONLY	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, some partitions will not be cached and will be recomputed on the fly each time they're needed. This is the default level.
MEMORY_AND_DISK	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, store the partitions that don't fit on disk, and read them from there when they're needed.
MEMORY_ONLY_SER (Java and Scala)	Store RDD as *serialized* Java objects (one byte array per partition). This is generally more space-efficient than deserialized objects, especially when using a fast serializer, but more CPU-intensive to read.
MEMORY_AND_DISK_SER (Java and Scala)	Similar to MEMORY_ONLY_SER, but spill partitions that don't fit in memory to disk instead of recomputing them on the fly each time they're needed.
DISK_ONLY	Store the RDD partitions only on disk.
MEMORY_ONLY_2, MEMORY_AND_DISK_2, etc.	Same as the levels above, but replicate each partition on two cluster nodes.
OFF_HEAP (experimental)	Similar to MEMORY_ONLY_SER, but store the data in off-heap memory. This requires off-heap memory to be enabled.

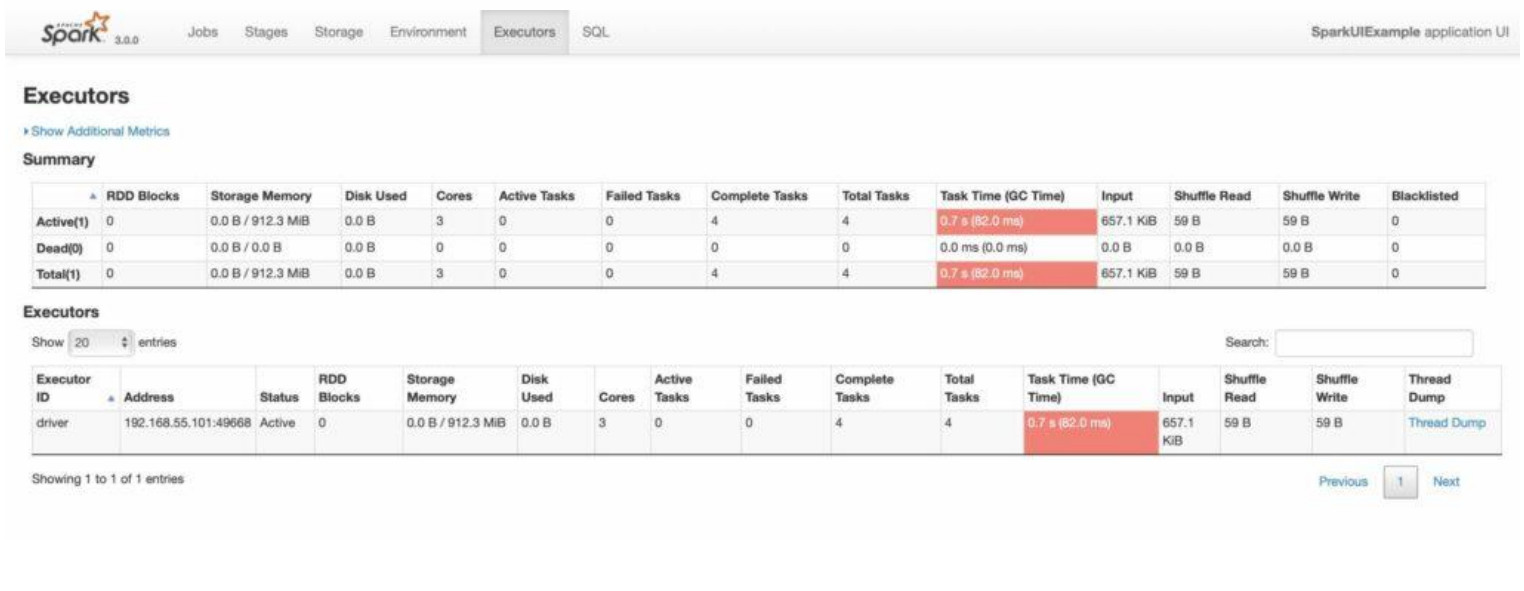
Static vs Dynamic Resource Allocation

PySpark manages resources (like CPU and memory) by using executors to run tasks. With static allocation, you pre-define a fixed number of executors for the entire job. This is simple but can be inefficient if workloads vary. Dynamic allocation is more flexible. PySpark requests more executors when tasks are numerous or demanding, and releases them when they are no longer needed. This optimizes resource usage and can save costs, especially for unpredictable workloads. It works by interacting with a cluster manager like YARN or Kubernetes.

Spark Web UI


Spark 3.0.0
Jobs
Stages
Storage
Environment
Executors
SQL
SparkUIExample application UI
Spark Jobs (?)
User: sriramrimmalapudi
Total Uptime: 42 min
Scheduling Mode: FIFO
Completed Jobs: 2
Event Timeline
Enable zooming
Executors
Added
Removed
Executor driver added
Jobs
Succeeded
Failed
Running
csv at SparkUI
count
29
20 July 19:03
30
31
32
33
34
35
Completed Jobs (2)
Page: 1
1 Pages. Jump to 1 . Show 100 items in a page. Go
Job Id
Description
Submitted
Duration
Stages: Succeeded/Total
Tasks (for all stages): Succeeded/Total
1
count at SparkUIExample.scala:18
count at SparkUIExample.scala:18
2020/07/20 19:03:35
0.2 s
2/2
2/2
0
csv at SparkUIExample.scala:16
csv at SparkUIExample.scala:16
2020/07/20 19:03:34
0.4 s
1/1
1/1
Page: 1
1 Pages. Jump to 1 . Show 100 items in a page. Go


Spark 3.0.0
Jobs
Stages
Storage
Environment
Executors
SQL
SparkUIExample application UI
Stages for All Jobs
Completed Stages: 4
Completed Stages (4)
Page: 1
1 Pages. Jump to 1 . Show 100 items in a page. Go
Stage Id
Description
Submitted
Duration
Tasks: Succeeded/Total
Input
Output
Shuffle Read
Shuffle Write
3
count at SparkUIExample.scala:20 +details
2020/07/20 21:41:35
57 ms
1/1
59.0 B
2
count at SparkUIExample.scala:20 +details
2020/07/20 21:41:35
93 ms
1/1
296.6 KiB
59.0 B
1
csv at SparkUIExample.scala:18 +details
2020/07/20 21:41:35
0.2 s
1/1
296.6 KiB
0
csv at SparkUIExample.scala:18 +details
2020/07/20 21:41:34
0.3 s
1/1
64.0 KiB
Page: 1
1 Pages. Jump to 1 . Show 100 items in a page. Go


Spark 3.0.0
Jobs
Stages
Storage
Environment
Executors
SQL
SparkUIExample application UI
Executors
Show Additional Metrics
Summary
RDD Blocks
Storage Memory
Disk Used
Cores
Active Tasks
Failed Tasks
Complete Tasks
Total Tasks
Task Time (GC Time)
Input
Shuffle Read
Shuffle Write
Blacklisted
Active(1) 0 0.0 B / 912.3 MiB 0.0 B 3 0 0 4 4 0.7 s (82.0 ms) 657.1 KiB 59 B 59 B 0
Dead(0) 0 0.0 B / 0.0 B 0.0 B 0 0 0 0 0 0.0 ms (0.0 ms) 0.0 B 0.0 B 0.0 B 0
Total(1) 0 0.0 B / 912.3 MiB 0.0 B 3 0 0 4 4 0.7 s (82.0 ms) 657.1 KiB 59 B 59 B 0
Executors
Show 20 entries
Search:
Executor ID
Address
Status
RDD Blocks
Storage Memory
Disk Used
Cores
Active Tasks
Failed Tasks
Complete Tasks
Total Tasks
Task Time (GC Time)
Input
Shuffle Read
Shuffle Write
Thread Dump
driver 192.168.55.101:49668 Active 0 0.0 B / 912.3 MiB 0.0 B 3 0 0 4 4 0.7 s (82.0 ms) 657.1 KiB 59 B 59 B Thread Dump
Showing 1 to 1 of 1 entries
Previous 1 Next

Kafka or Azure Event Hub (Azure based, fully managed)

1. What is Kafka?

1. Kafka is a distributed event streaming platform used to build real-time data pipelines.
2. Kafka is highly scalable, fault-tolerant, and optimized for high-throughput, low-latency messaging between producers and consumers.
3. It supports publish-subscribe and message queue models, making it ideal for decoupled microservices, log aggregation, real-time analytics, and ETL workflows.

🔷 2. Kafka Architecture Components

4. Producer: Sends (publishes) messages to Kafka topics. Can send data continuously, such as logs, metrics, or transactional events.
5. Consumer: Reads (subscribes to) messages from Kafka topics. Typically part of a consumer group that helps with parallel processing.
6. Topic: Logical channel or stream to which messages are published and from which consumers read. Topics can have one or more partitions.
7. Partition (2 copies. Leader/followers): Kafka splits each topic into partitions for parallelism and scalability. Each partition is an ordered, immutable sequence of messages.
8. Broker: Kafka server that stores and delivers messages. A Kafka cluster consists of multiple brokers.
9. Cluster: A group of brokers that collectively manage topics, partitions, and client requests.
10. Zookeeper: Manages cluster metadata, broker coordination, and leader election for partitions. (Note: Being phased out with KRaft mode in newer Kafka versions.)
11. Offset: A unique identifier for each message within a partition. Consumers use offsets to track their read position.
12. Replication: Kafka replicates partitions across multiple brokers for high availability and data durability.

🔷 3. End-to-End Message Lifecycle in Kafka

15. A producer writes a message to a topic, optionally with a key (used for partitioning logic).
16. Kafka routes the message to a specific partition within the topic.
17. The broker stores the message in that partition and assigns it a sequential offset.
18. Kafka replicates the message to follower brokers as per the topic's replication factor.
19. A consumer (or consumer group) reads the message from the partition via its offset.
20. After processing, the consumer commits the offset, either automatically or manually, to avoid reprocessing in case of a failure.

```
from pyspark.sql import SparkSession
from pyspark.sql.types import StructType, StructField, StringType
import json

# Initialize SparkSession
spark = SparkSession.builder.appName("KafkaToDelta").getOrCreate()
spark.sparkContext.setLogLevel("ERROR")

# Kafka configuration
kafka_bootstrap_servers = "your_kafka_broker:9092"  # Replace with
your Kafka broker
kafka_topic = "your_kafka_topic" # Replace with your Kafka topic
delta_table_path = "/path/to/your/delta/table"  # Replace with your
desired Delta table path

# Define the schema of the Kafka messages (adjust according to your
needs)
# Example schema for JSON messages
kafka_schema = StructType([
    StructField("id", StringType(), True),
    StructField("event_data", StringType(), True),
])

# Read streaming data from Kafka
df = (spark.readStream
      .format("kafka")
      .option("kafka.bootstrap.servers", kafka_bootstrap_servers)
      .option("subscribe", kafka_topic)
      .option("startingOffsets", "earliest") # or "latest" to start
from the most recent
      .load()
)

# Extract and parse the message value (assuming it's in JSON format)
df_value = df.selectExpr("CAST(value AS STRING)")
# Parse the JSON message
df_parsed = df_value.withColumn("json_data",
json.loads(df_value.value))
df_final = df_parsed.selectExpr("json_data.*")

# Write the streaming data to Delta table
(df_final.writeStream
        .format("delta")
        .option("checkpointLocation", "/path/to/your/checkpoint")  #
Create a checkpoint directory
        .outputMode("append")  # Or "update" if you want to update
the table
        .start(delta_table_path))

# Keep the program running until explicitly stopped
df_final.writeStream.awaitTermination()
```

Read from Azure Event Hub

```
from pyspark.sql import SparkSession
from pyspark.sql.types import StructType, StructField, StringType,
TimestampType
from pyspark.sql.functions import *
import json

# Event Hubs configuration
EVENT_HUB_NAMESPACE = "your-event-hub-namespace"
EVENT_HUB_NAME = "your-event-hub-name"
EVENT_HUB_KEY_NAME = "your-event-hub-key-name"
EVENT_HUB_KEY_VALUE = dbutils.secrets.get(scope = "your-secrets-
scope", key = EVENT_HUB_KEY_NAME)

# Delta table path
DELTA_TABLE_PATH = "/path/to/your/delta/table"

# Create SparkSession
spark =
SparkSession.builder.appName("EventHubToDelta").getOrCreate()

# Define schema for Event Hub messages
eventHubSchema = StructType([
    StructField("device_id", StringType(), True),
    StructField("temperature", StringType(), True),
    StructField("humidity", StringType(), True),
    StructField("timestamp", TimestampType(), True)
])

# Configure Event Hubs source
event_hub_options = {
    "eventhubs.namespace": EVENT_HUB_NAMESPACE,
    "eventhubs.eventhubname": EVENT_HUB_NAME,
    "eventhubs.sasl.jaas.config":
"org.apache.kafka.common.security.plain.PlainSaslClientFactory",
    "eventhubs.sasl.mechanism": "PLAIN",
    "eventhubs.security.protocol": "SASL",
    "eventhubs.sasl.client.callback.handler.class":
"org.apache.kafka.common.security.plain.PlainSaslClientCallbackHandl
er",
    "eventhubs.eventhubs.sasl.username": "your-event-hub-key-name",
# Replace with your key name
    "eventhubs.eventhubs.sasl.password": EVENT_HUB_KEY_VALUE  #
Replace with your key value
}

# Read streaming data from Event Hubs
streamingDF = spark.readStream \

    .format("eventhubs") \
```

```
    .options(**event_hub_options) \
    .schema(eventHubSchema) \
    .load()

# Apply transformations (e.g., convert JSON to data types, etc.)
# For demonstration purposes, we assume the messages are already in the correct format
# You might need to add transformations here to parse JSON, convert data types, etc.
# transformedDF = streamingDF.selectExpr("device_id", "cast(temperature as double) as temperature", "cast(humidity as double) as humidity", "timestamp")

# Apply watermark (handle late data)
transformedDF = streamingDF.selectExpr("device_id", "cast(temperature as double) as temperature", "cast(humidity as double) as humidity", "timestamp")
watermarkedDF = transformedDF.withWatermark("timestamp", "10 minutes")

# Write the stream to a Delta table
def write_batch(df, batch_id):
    df.write.mode("append").delta(DELTA_TABLE_PATH)

query = watermarkedDF.writeStream \
    .outputMode("append") \
    .foreachBatch(write_batch) \
    .start()

query.awaitTermination()
```

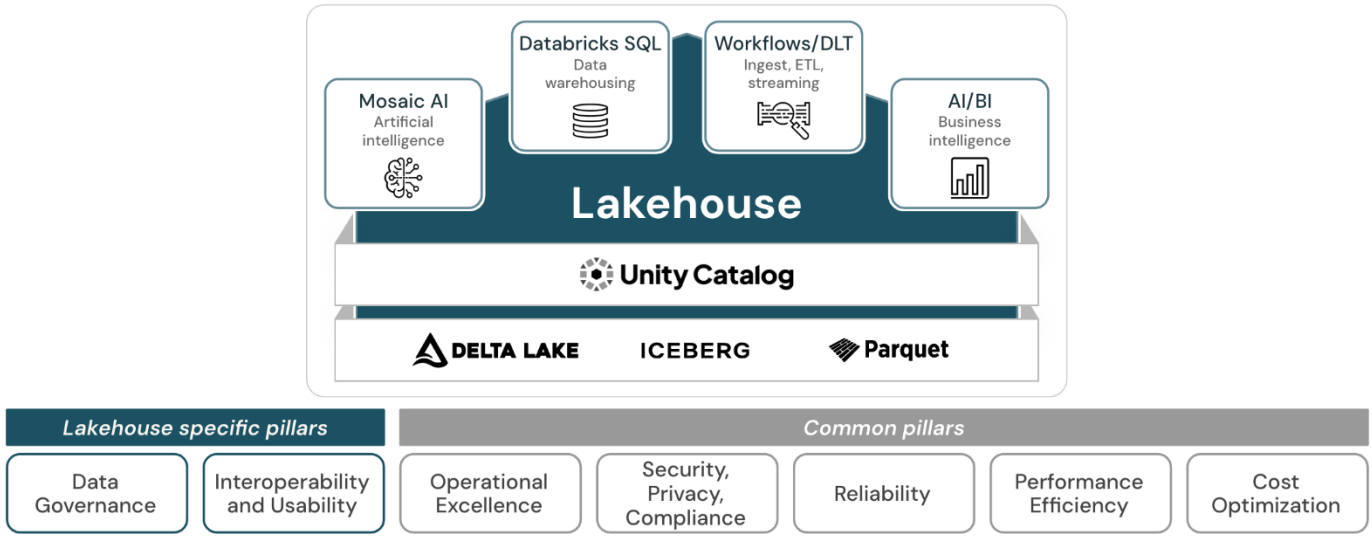
PySpark vs Spark SQL file:///C:/Users/geram/OneDrive/Desktop/Learning/Curated/Pyspark%20quick%20notes.pdf

Task	SQL Command	PySpark Command
Selecting Data	SELECT column1, column2 FROM table_name	df.select("column1", "column2")
Filtering Data	SELECT * FROM table_name WHERE condition;	df.filter(condition)
Grouping and Aggregating	SELECT column1, COUNT(*) FROM table_name GROUP BY column1;	df.groupBy("column1").count()
Joining Data	SELECT * FROM table1 JOIN table2 ON table1.column = table2.column;	df1.join(df2, df1.column == df2.column)
Ordering Data	SELECT * FROM table_name ORDER BY column ASC;	df.orderBy("column", ascending=True)
Creating Temporary Views	CREATE OR REPLACE TEMP VIEW view_name AS SELECT * FROM table_name;	df.createOrReplaceTempView("view_name")
Running SQL Queries	SELECT * FROM view_name WHERE condition;	spark.sql("SELECT * FROM view_name WHERE condition").show()
Writing Data to Table	INSERT INTO table_name (column1, column2) VALUES (value1, value2);	df.write.mode("overwrite").saveAsTable("table_name")
Reading Data from Table	SELECT * FROM table_name;	df = spark.read.table("table_name")
Counting Rows	SELECT COUNT(*) FROM table_name;	df.count()
Dropping Duplicates	SELECT DISTINCT column1, column2 FROM table_name;	df.dropDuplicates(["column1", "column2"])
Aggregating by Condition	SELECT column1, AVG(column2) FROM table_name GROUP BY column1;	df.groupBy("column1").avg("column2")
Renaming Columns	SELECT column1 AS new_column_name FROM table_name;	df.withColumnRenamed("column1", "new_column_name")
Limiting Rows	SELECT * FROM table_name LIMIT 10;	df.limit(10)
Calculating Sum (similar for Max, MIN, AVG)	SELECT SUM(column1) FROM table_name;	df.selectExpr("sum(column1)").show()
Dropping Columns	SELECT * FROM table_name;	df.drop("column1", "column2")
Handling Null or Not null Values	SELECT column1 FROM table_name WHERE column2 IS NULL/NOT NULL;	df.filter(df.column2.isNull()) / df.filter(df.column2.isNotNull())
Greater Than or Equal (same as <,>,<=,==,!=)	SELECT * FROM table_name WHERE column >= value;	df.filter(df.column >= value).show()
IN	SELECT * FROM table_name WHERE column IN (value1, value2);	df.filter(df.column.isin(value1, value2)).show()
Between	SELECT * FROM table_name WHERE column BETWEEN value1 AND value2;	df.filter(df.column.between(value1, value2)).show()
LIKE	SELECT * FROM table_name WHERE column LIKE 'pattern';	df.filter(df.column.like('pattern')).show()
CAST	SELECT CAST(column AS data_type) FROM table_name;	df.selectExpr("CAST(column AS data_type)").show()

Databricks

Azure Databricks is a cloud-based, unified analytics platform built on Apache Spark that provides a collaborative and interactive workspace for data engineering, data science, and machine learning tasks

- **Commercial Product**: Created by the original creators of Apache Spark. While Spark is open-source, Databricks is a commercial product that provides additional features on top of Spark.



Databricks Key Features:

1. Unity Catalog

- Centralized **data governance** and **fine-grained access control** across workspaces.
- Supports **table, column, row-level security**, and **data lineage** tracking.

2. Delta Live Tables (DLT)

- Simplifies **ETL pipeline development** using **declarative syntax**.
- Supports **automatic lineage**, **error handling**, **testing**, and **auto-scaling**.
- Ensures **data quality with expectations** (like unit tests for data).

3. Delta Lake

- Brings **ACID transactions**, **time travel**, and **schema evolution** to your data lake.
- Supports **Merge (Upserts)** and **CDC (Change Data Capture)** use cases.

4. Medallion Architecture (Bronze, Silver, Gold)

- Standard design pattern in Databricks for **structured data transformation pipelines**.
- Promotes **modularity**, **data quality**, and **performance**.

5. Databricks SQL

- Fully managed SQL interface to **query Delta tables** using BI tools (Power BI, Tableau).

- Offers **query history**, **alerts**, **dashboards**, and **serverless SQL endpoints**.

6. Photon Engine

- Next-gen vectorized query engine for **blazing-fast SQL performance**.
- Optimized for **Delta Lake** and **Databricks SQL** workloads.

7. Lakehouse Architecture

- Combines **data lake** and **data warehouse** benefits in a single platform.
- Supports both **BI/SQL analytics** and **ML workloads** on the same data.

8. Job Orchestration & Workflows

- Schedule and manage **notebooks, scripts, JARs** with dependencies.
- Supports **parameter passing**, **retry policies**, and **multi-task pipelines**.

9. Data Lineage

- Unity Catalog and DLT provide **automated lineage** for datasets, helping with **audits** and **impact analysis**.

10. Access Control & Auditing

- Unity Catalog enables **central policy enforcement** and **audit logging** across catalogs.
- Integration with **Azure Active Directory** for secure role-based access.

11. Built-in MLflow Integration

- Native support for **ML lifecycle management** – tracking, packaging, and deploying ML models.

12. Workspace Repos & Git Integration

- First-class support for **GitHub, Azure DevOps**, etc., for version-controlling notebooks and jobs.
- Enables **CI/CD** best practices in data workflows.

13. Auto-Optimize & Auto-Compaction

- Delta Lake optimizations that manage **file sizes** and **data layout** automatically for better performance.

14. Serverless Compute Options

- Serverless SQL endpoints and **serverless Model Serving** reduce infrastructure management overhead.

15. Powerful Runtime Optimizations

- Databricks Runtime includes **Spark optimizations**, **I/O caching**, **adaptive query execution**, and support for **pandas API on Spark**.

16. Intelligent Workload Management

Efficient job scheduling, autoscaling, and prioritization.

- **Cluster reuse**, **job prioritization**, **resource limits**
- Auto-terminate idle clusters

17. Autoloader

Efficient ingestion of streaming/batch data from cloud storage.

- Supports **schema inference**, **incremental processing**
- Scales with **millions of files** (JSON, CSV, Parquet)

Databricks Architecture:

Databricks architecture has two main components:

* **Control plane** is the management layer, handling the workspace WebApp, notebooks, configuration and clusters. Objects are **stored in a Microsoft-managed subscription**, not in your customer's subscription. Objects are managed securely through a combination of **encryption at rest, customer-managed keys, and network isolation**.

* **Data Plane:** The data plane, where your data is actually stored and processed, **resides in your customer's Azure subscription**

Databricks' main components include the Control Plane, Compute Plane, and Unity Catalog

Databricks Runtime

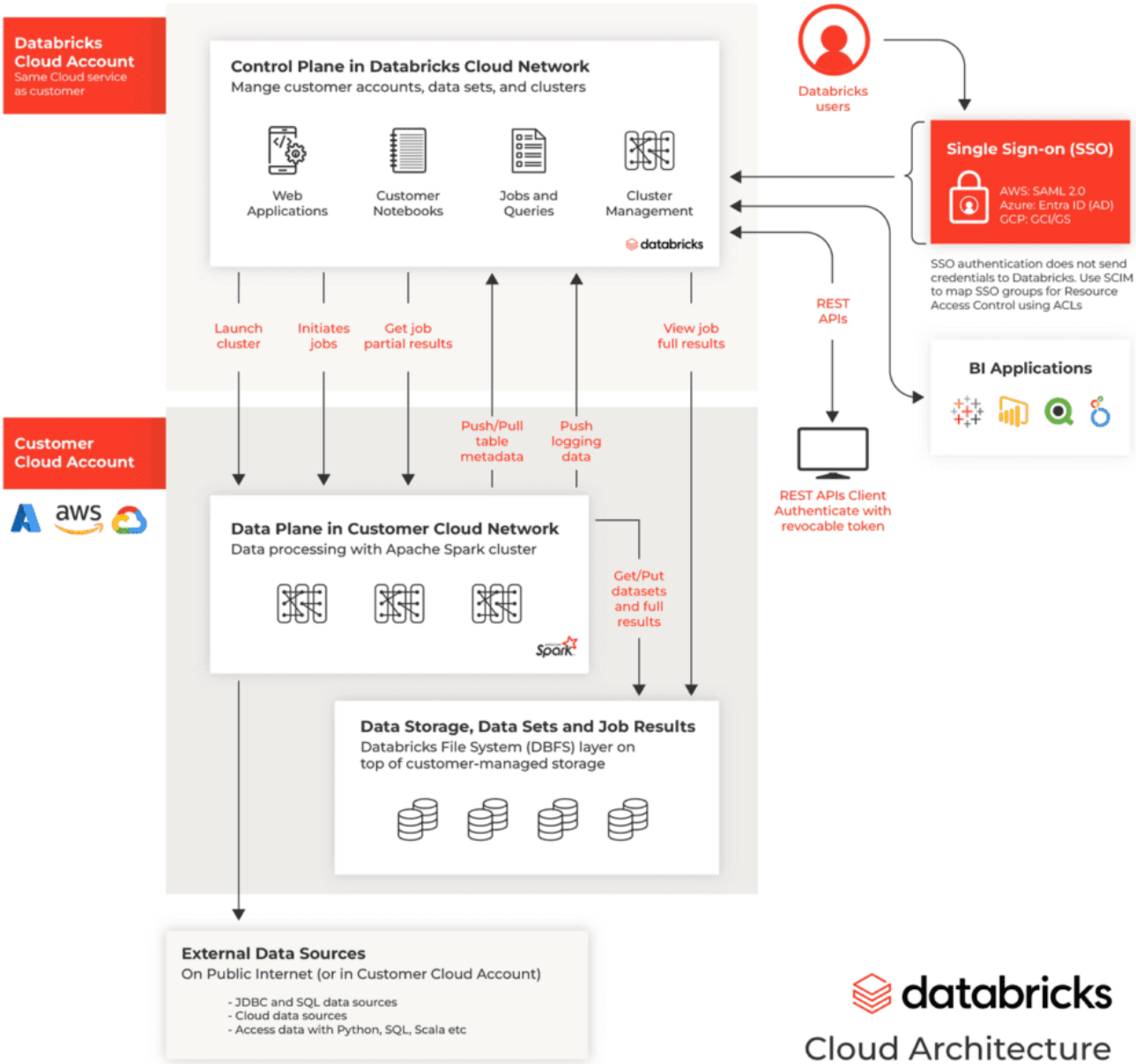
Databricks Runtime is the set of core components that run on your cluster.

Runtime includes:

- Ubuntu OS
- Spark
- Java, Python, Scala, R libraries
- Delta Lake
- GPU libraries
- Azure databricks services (Notebooks, jobs)
- Photon

Photon Engine (rewritten in C++, Vectorized processing and optimized algorithms)**:**

- Photon is a high-performance, native vectorized query engine within Databricks.
- Accelerates Spark SQL and DataFrame workloads, potentially reducing overall costs.
- It achieves this by replacing parts of Spark's execution engine with its own C++-based implementation, **leveraging vectorized processing and optimized algorithms**.



databricks

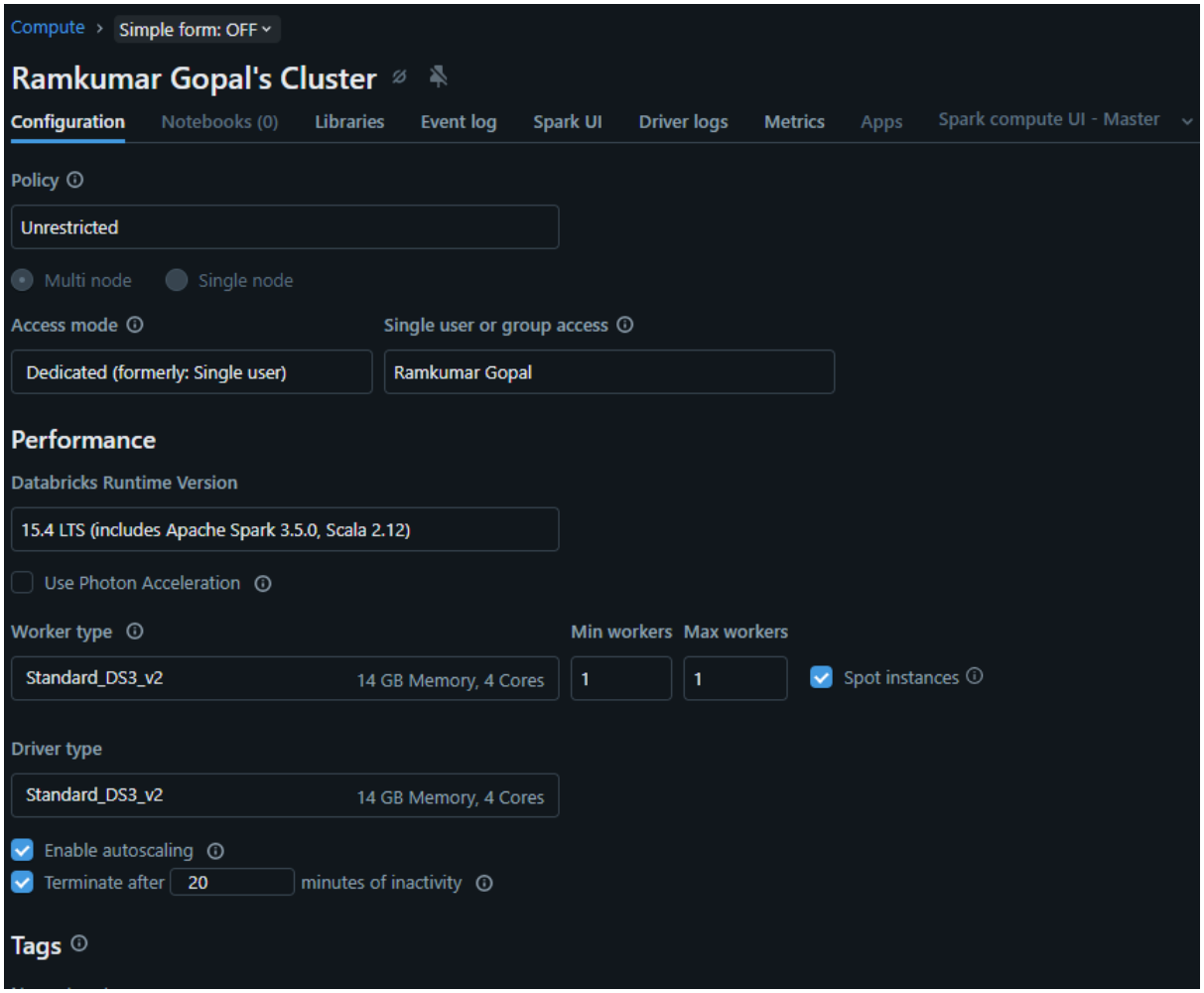
Cloud Architecture

Types of Clusters:

All purpose

Access Mode	Visible to user	UC Support	Supported Languages	Notes
Single user	Always	Yes	Python, SQL, Scala, R	Can be assigned to and used by a single user.
Shared	Always (**Premium plan required**)	Yes	Python (on Databricks Runtime 11.3 LTS and above), SQL, Scala (on Unity Catalog-enabled clusters using Databricks Runtime 13.3 LTS and above)	Can be used by multiple users with data isolation among users.
No Isolation Shared	Admins can hide this cluster type by enforcing user isolation in the admin settings page.	No	Python, SQL, Scala, R	There is a related account-level setting for No Isolation Shared clusters.

Permission Settings: Can Manage (owner), Can Restart, Can attach

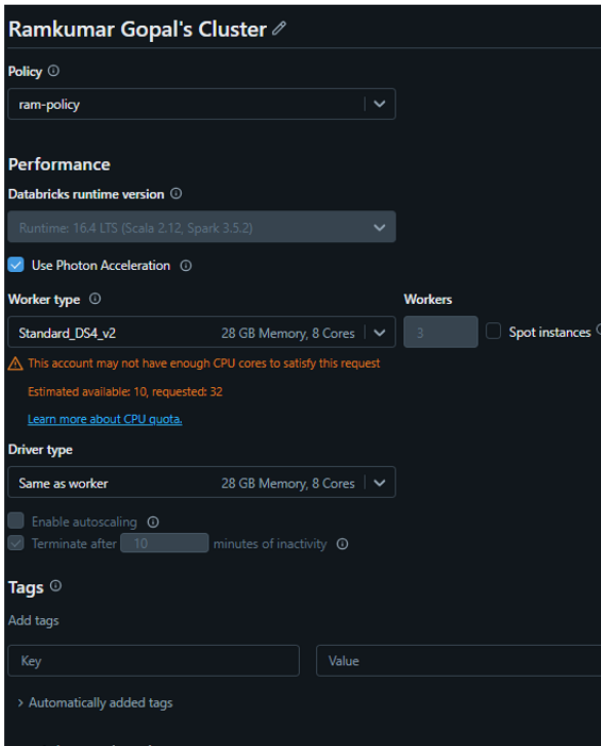


Sample Custom Policy (Overrides the shared policy):

```
{
  "node_type_id": {
    "type": "unlimited",
    "defaultValue": "Standard_DS4_v2",
    "isOptional": true
  },
  "spark_version": {
    "type": "fixed",
    "defaultValue": "auto:latest-lts",
    "value": "auto:latest-lts"
  },
  "num_workers": {
    "type": "fixed",
    "hidden": false,
    "value": 3
  },
  "data_security_mode": {
    "type": "fixed",
    "value": "USER_ISOLATION",
    "hidden": true
  },
  "driver_instance_pool_id": {
    "type": "forbidden",
    "hidden": true
  },
  "azure_attributes.spot_bid_max_price": {
    "type": "fixed",
    "value": -1,
    "hidden": true
  },
  "cluster_type": {
    "type": "fixed",
    "value": "all-purpose"
  },
  "instance_pool_id": {
    "type": "forbidden",
    "hidden": true
  },
  "azure_attributes.availability": {
    "type": "unlimited",
    "defaultValue": "ON_DEMAND_AZURE"
  },
  "spark_conf.spark.databricks.cluster.profile": {
    "type": "forbidden",
    "hidden": true
  },
  "autoscale.min_workers": {
    "type": "forbidden",
    "defaultValue": 2
  },
```

```
  "autotermination_minutes": {
    "type": "fixed",
    "defaultValue": 10,
    "value": 10
  },
  "autoscale.max_workers": {
    "type": "forbidden",
    "defaultValue": 10
  }
}
```

Shared Compute Policy vs Custom Policy



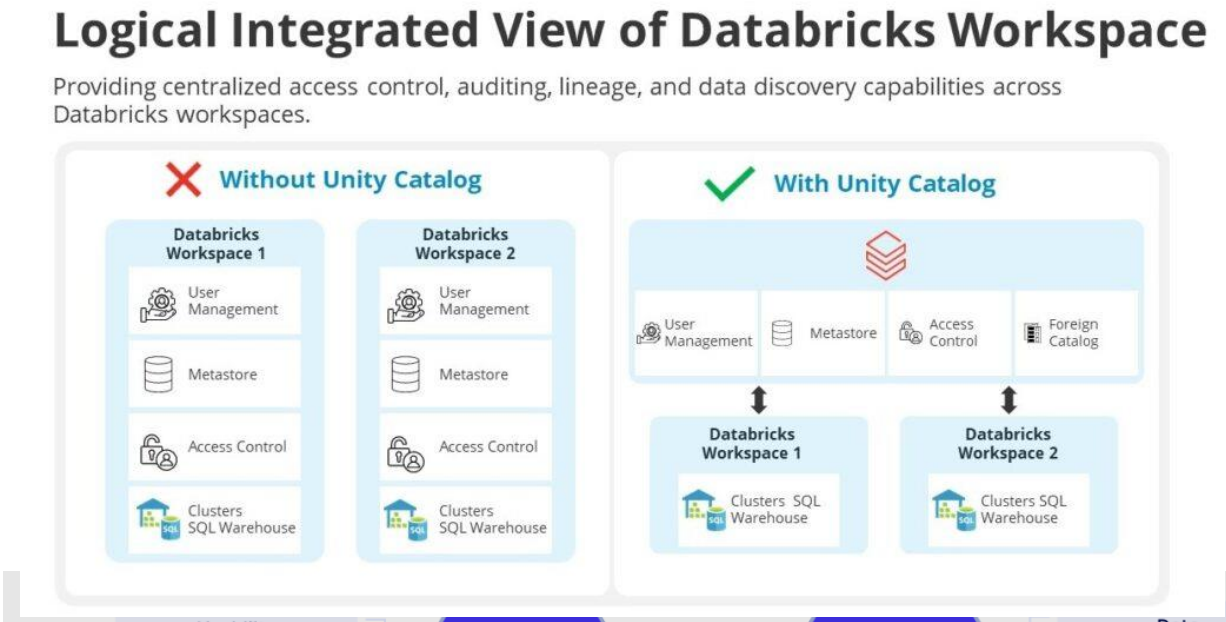
SQL Warehouse

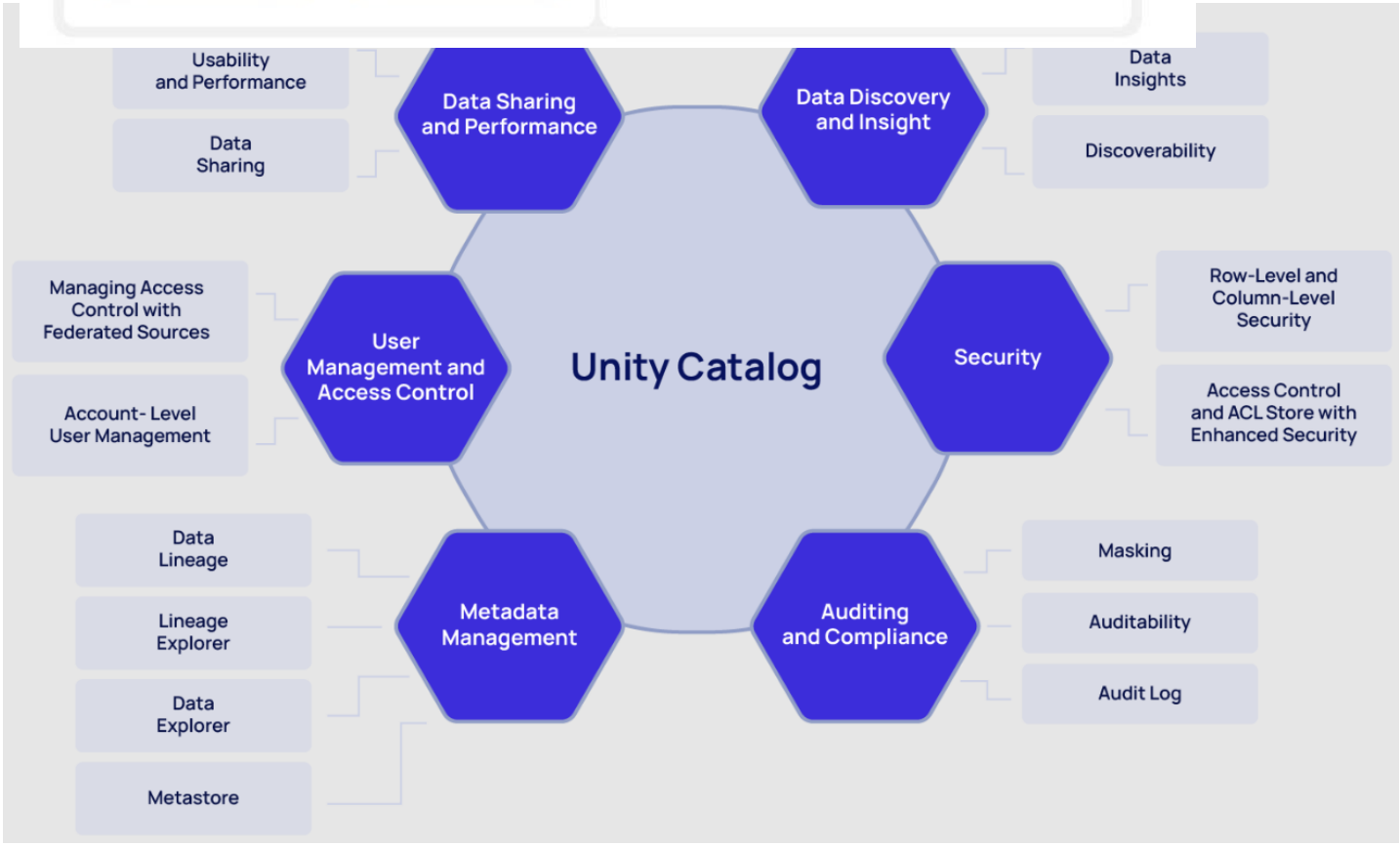
Warehouse type	Photon Engine	Predictive IO	Intelligent Workload Management
Serverless	X	X	X
Pro	X	X	
Classic	X		

Unity Catalog - is a centralized data governance solution for Databricks.

Features:

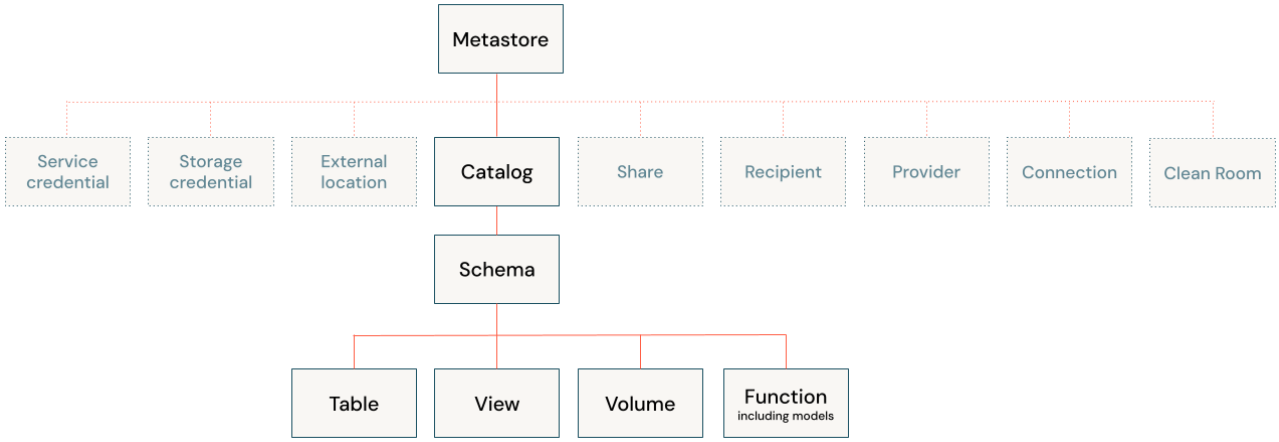
- Access Control
- Row level & Column level security.
- Audit
- Lineage
- Data Discovery
- Data sharing
- and system tables





Unity Catalog Object Model

you could have catalogs for different environments (dev, test, prod) or departments (finance, marketing). This approach allows for clear separation of data, simplified access control, and easier data discovery



Metadata of unity catalog is stored in Control plane. Data in User Storage Accounts.

By using catalog binding, you can assign specific catalogs to designated workspaces, enabling clear segmentation between teams and environments. This approach ensures that each workspace handles only the data relevant to its specific tasks.

Managed Table Data Storage Location Hierarchy

If you create a table, data will be stored in metastore location.

If path is mentioned in catalog, data will be stored in catalog location

If path is mention in schema, data will be stored in schema location.

If no path is mentioned in metastore, it's mandatory to provide location for catalog

Why Managed Tables are better

◈ Simplified Data Management

With managed tables, Databricks automatically handles the underlying storage, file organization, and cleanup. This eliminates the headache of orphaned files or leftover data when a table is dropped. Managed tables ensure your data storage remains clean and organized without manual intervention.

◈ Enhanced Data Governance and Security

One of the biggest advantages of Unity Catalog managed tables is their built-in governance capabilities. Managed tables are tightly integrated with Unity Catalog, ensuring:

- Seamless enforcement of access controls.
- End-to-end data lineage tracking.
- Comprehensive auditing to track data access and changes.

In contrast, external tables pose a higher security risk because they allow direct access to the underlying storage paths. This creates two major challenges:

1. Users may bypass Databricks access controls and gain unauthorized access to data.
2. Auditing becomes harder since users can access the data directly via cloud storage services instead of Databricks itself.

Databricks with Unity Catalog supports two types of tables:

- **Managed Tables:** These are fully managed by Unity Catalog, including their location, lifecycle, and file layout. By default, managed tables are stored in the root storage location configured when the metastore is created. Alternatively, you can specify a storage location at the catalog or schema level.

  ```
  CREATE TABLE catalog.schema.sales
  (
   id INT,
   deptname  STRING,
   amt DECIMAL(10,2)
  );
  ```

- **External Tables:** Unity Catalog does not manage these tables' lifecycle and file layout.

  ```
  CREATE TABLE catalog.schema.sales
  (  id INT,
   deptname  STRING,
   amt DECIMAL(10,2)
  )
  LOCATION 'abfss://container@account.dfs.core.windows.net/sales';
  ```

- **Volumes -** Volumes provide a mechanism to access files stored in a storage account. They are useful when scripts need to access files directly. The file location structure for volumes is Volumes/catalog/schema/directory

  There are two types of volumes:

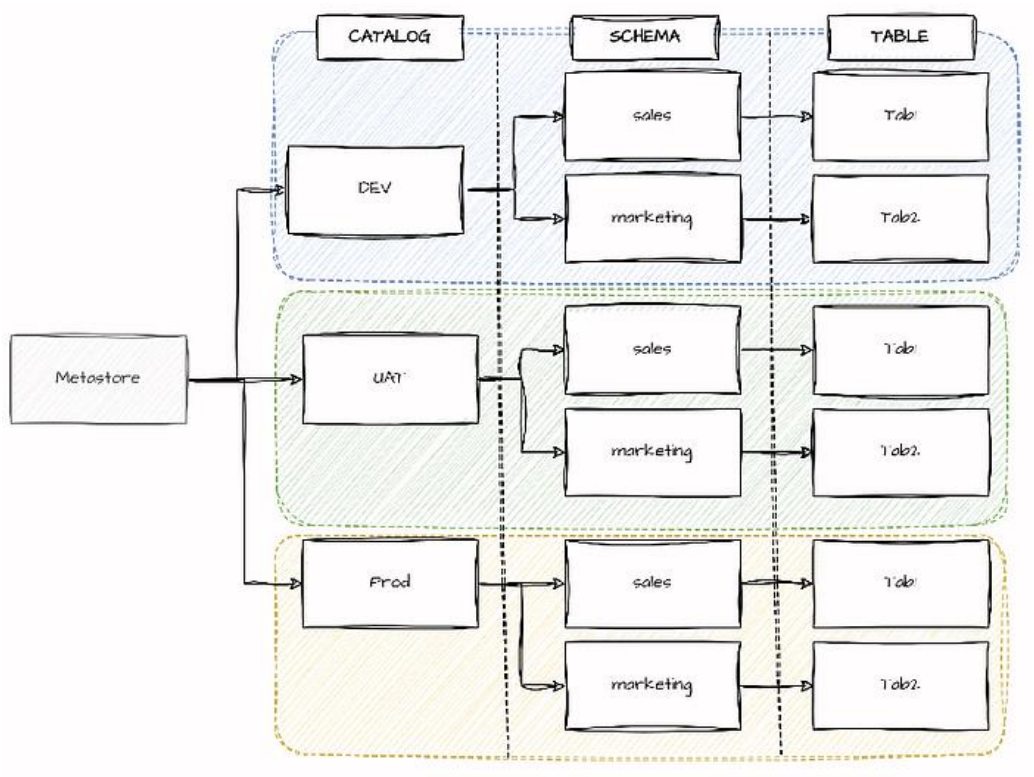
  - **Managed Volumes:** Created within the default storage location of the containing schema.

    CREATE VOLUME <catalog>.<schema>.<volume-name>;

- **External Volumes:** These allow specifying an external location not governed by Unity Catalog. They are ideal for accessing files in a landing area, for instance. Users do not need direct access to the storage account; Unity Catalog manages access through volumes.

  ```
  CREATE EXTERNAL VOLUME <catalog>.<schema>.<volume-name>
  LOCATION 'abfss://container@account.dfs.core.windows.net/path/directory';
  ```

Data Organization Using Unity Catalog



Unity Catalog's hierarchical structure means that privileges assigned at higher levels (catalog or schema) are inherited by objects at lower levels. Thus, assigning a privilege at the catalog or schema level will automatically apply it to all existing and future objects within that catalog or schema.

Unity Catalog integrates with Microsoft Entra ID (Azure Active Directory), enabling the creation of entities such as users and groups in Microsoft Entra ID to manage access to Unity Catalog assets.

Catalog Creation:

CREATE CATALOG IF NOT EXISTS demo_catalog MANAGED LOCATION 'abfss://mycontainer@mystorageaccount.dfs.core.windows.net/demo_catalog';

create catalog dev_sql comment 'created using sql'

drop catalog dev_sql cascade;

Create Catalog with external location:

1. **Create storage credential under workspace/catalog**
2. **Create external location**

   create external location ext_catalog
   url 'abfss://data@sauccentralindia.dfs.core.windows.net/adb/catalog'
   with (storage credential `sc_catalog_storage`)
3. **Create Catalog**
   create catalog dev_ext

managed location
'abfss://data@sauccentralindia.dfs.core.windows.net/adb/catalog'

4. *describe catalog extended dev_ext*

Schema creation

CREATE SCHEMA test MANAGED LOCATION
'abfss://container@account.dfs.core.windows.net/path/directory'

After creating schemas, you can proceed with developing tables, views, functions, volumes, and models using SQL commands in notebooks or the Databricks GUI, depending on the specific use case.

CREATE TABLE test.my_table (id INT, name STRING);

Grant Access:

GRANT USE CATALOG ON CATALOG < catalog_name > TO < group_name >;

GRANT USE SCHEMA ON SCHEMA < catalog_name >.< schema_name > TO < group_name >;

GRANT SELECT ON < catalog_name >.< schema_name >.< table_name >; TO < group_name >;

Row and Column Level Access Control with Unity Catalog

Unity Catalog enables the implementation of column-level security, data masking using dynamic views, and row-level security and column masking using functions.

For example:

CREATE VIEW < catalog_name >.< schema_name >.< view_name > as
SELECT
id,
***CASE WHEN is_account_group_member(< group_name >) THEN email ELSE 'REDACTED' END** AS email, country, product, total FROM*
< catalog_name >.< schema_name >.< table_name >;

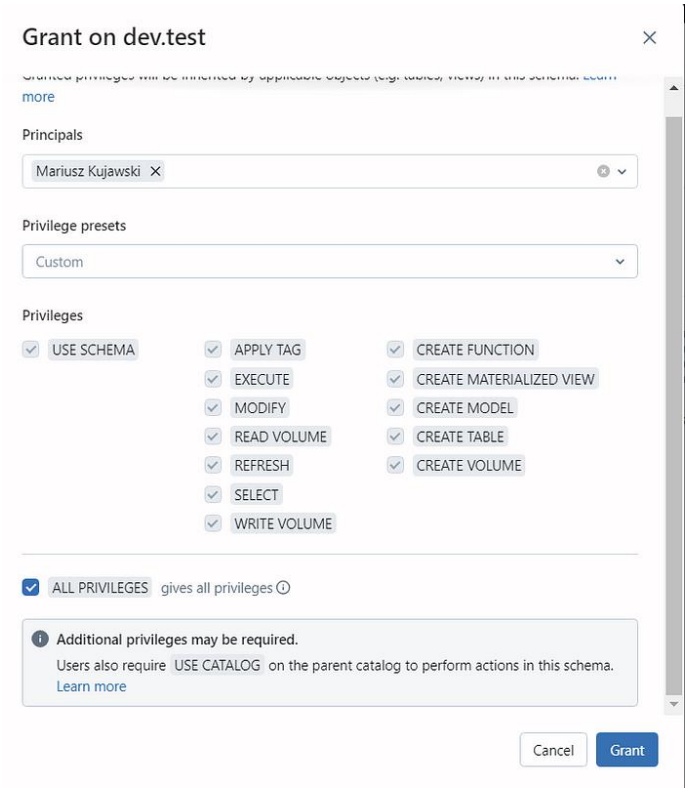
Using user-defined functions we can apply security roles on columns or rows to mask data or prevent part of users accessing data.

CREATE OR REPLACE FUNCTION region_filter(region_param STRING)
RETURN
is_account_group_member('bu_admin') or -- bu_admin can access all regions
region_param like "US%"; -- non bu_admin's can only access regions containing US

ALTER TABLE customers SET ROW FILTER region_filter ON (country);

Example of masking PII data using a masking function:

CREATE OR REPLACE FUNCTION simple_mask(column_value STRING)
*RETURN IF(is_account_group_member('bu_admin'), column_value, "****");*



Drop table functionality difference in Hive and Unity catalog

- In both, data of external tables are not deleted
- In Hive managed tables, managed table data will be immediately deleted
- In UC managed tables, managed table data will be deleted after 7 days.
  we can undrop the deleted managed table

  drop table dev.bronze.emp;
  ***show tables dropped** in dev.bronze;*
  undrop table dev.bronze.emp;

Delta Tables:

Show catalogs like 'dev'; <- REGX pattern*
Show schemas in <catalog> like 'xx';*
show tables in catalog.schema like 'xx';*

spark.catalog.tableExists("dev.bronze.sales")

CREATE SCHEMA <name> IF NOT EXISTS dev.bronze;
CREATE TABLE <table_name> IF NOT EXISTS dev.bronze.emp(…)

DESCRIBE HISTORY <table_name>; -- to see the change log

*SELECT * from <table>@1*

CREATE TEMPORARY VIEW <catalog>.<schema>.<view> -> session scope

CREATE OR REPLACE VIEW <catalog>.<schema>.<view>

WAYS to Clone Tables

1. ***CTAS*** *(Create table as) – Create copy of the table*

   CREATE TABLE dev.bronze.emp_ctas
   AS
   *SELECT * FROM dev.bronze.emp;*

   DESCRIBE EXTENDED dev.bronze.emp;

2. DEEP CLONE (Both metadata and Data): - Recommended approach

   CREATE TABLE dev.bronze.emp_deep
   DEEP CLONE dev.bronze.emp;

   DESCRIBE EXETENDED dev.bronze.emp_deep;
   Note: Location will be emp_deep path

3. SHALLOW CLONE (Only metadata. Data will be pointed **to original table of specific data version**)

   CREATE TABLE dev.bronze.emp_shallow
   SHALOW CLONE dev.bronze.emp;

   DESCRIBE EXETENDED dev.bronze.emp_shallow;
   Note: Location will be emp path.

   If we make change in original table, that won't reflect reflect in shallow table.
   If VACCUM command is executed in emp, then it will impact shallow table.

MERGE:

```
MERGE INTO training.person TARGET
USING vw_person SOURCE
ON TARGET.PersonId = SOURCE.PersonId
WHEN MATCHED THEN
  UPDATE SET
    TARGET.FirstName = SOURCE.FirstName,
    TARGET.LastName = SOURCE.LastName,
    TARGET.Country = SOURCE.Country
WHEN NOT MATCHED THEN
  INSERT (  PersonId,  FirstName,  LastName,  Country  )
  VALUES (  SOURCE.PersonId,  SOURCE.FirstName,  SOURCE.LastName,  SOURCE.Country )
```

Deletion Vectors:

- Delete Optimization technique. Enabled by default
- When huge number of records are deleted in parquet, the records will not be deleted immediately.
- Those will be marked for deletion and later deleted when OPTIMIZE command is executed
- Introduced in Delta Lake 3.1 or DB Runtime 13.3 LTS

Liquid Clustering:

Problem statement:

Let's say you have a large sales dataset stored on disk as a single Parquet file.

The traditional way to deal with this problem is to partition your data on a partitioning column. This is often called **Hive-style partitioning**.. Here's what a Hive-style partitioned table might look like on disk:

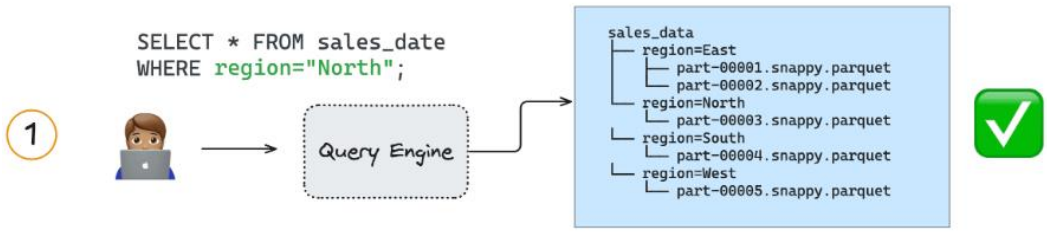
```
spark-warehouse/sales_data
├── region=East
│   ├── part-00001.snappy.parquet
│   └── part-00002.snappy.parquet
└── region=North
    └── part-00003.snappy.parquet
└── region=South
    └── part-00003.snappy.parquet
    └── part-00004.snappy.parquet
    └── part-00005.snappy.parquet
└── region=West
    └── part-00006.snappy.parquet
```

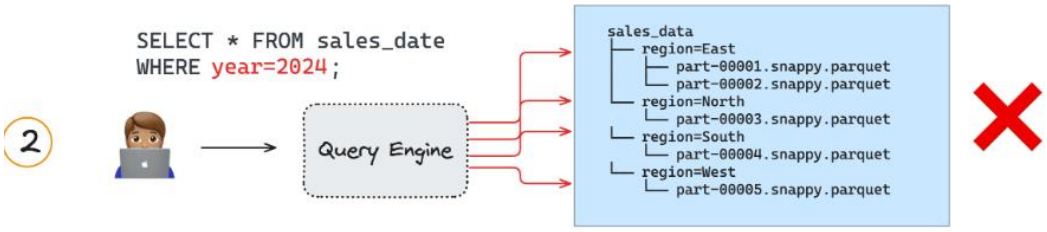
Queries on the partition column region will now run faster.

On top of this, you could also Z-order your data inside the partitions. **Z-Ordering is a Delta feature that groups related rows together in the same files and can sort your data in multiple dimensions**. For example, rows with similar values for region and date will end up close together. This will make your queries on either or both of those columns run faster.

The problem with partitioning and Z-ordering is that they are not flexible. You have to decide on your partitioning columns ahead of time and **if your query patterns change, you will have to rewrite the entire dataset to optimize it for that new query**.

Partitioning



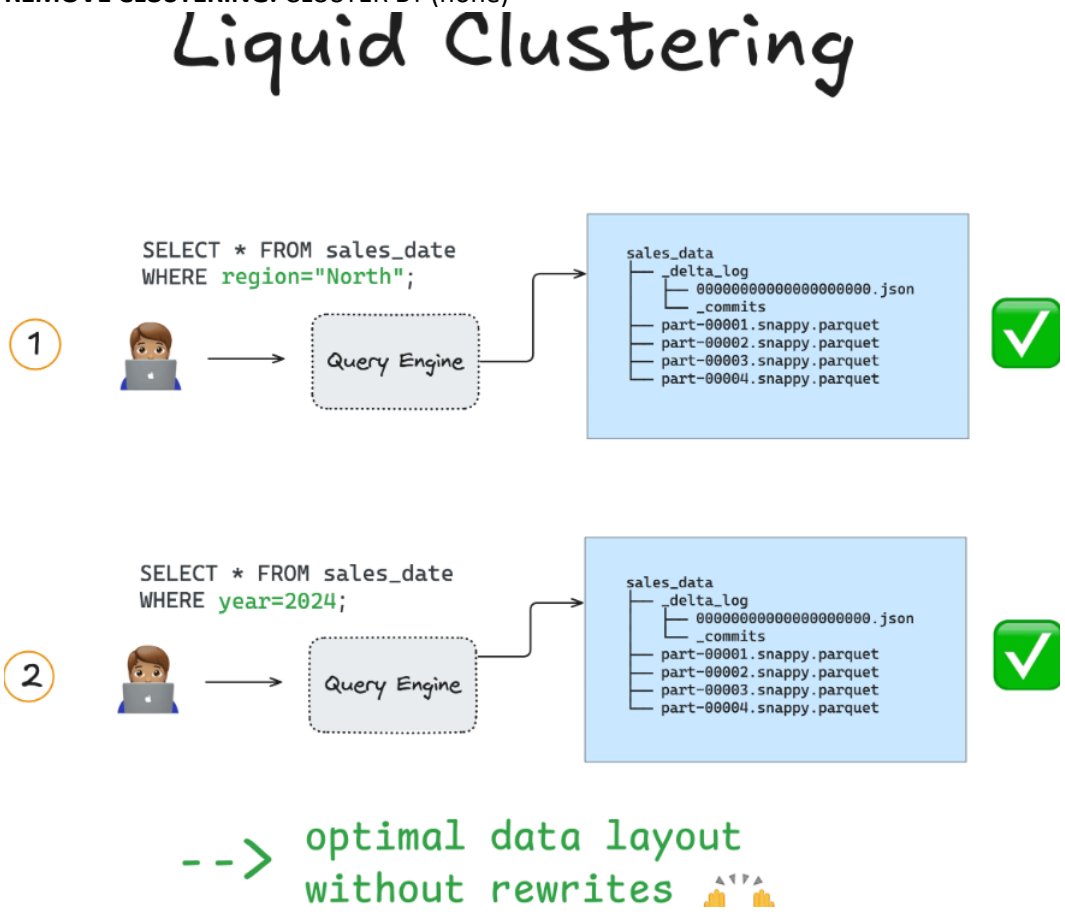


--> complete rewrite required 😰

Solution:

Liquid clustering gives you flexibility**. With liquid clustering enabled, you can redefine your clustering columns without having to rewrite any existing data**. This allows your data layout to evolve in parallel with changing query patterns.

CREATE TABLE table1(customer string, sales int) USING DELTA CLUSTER BY (customer);
OR
ALTER TABLE TABLE_NAME CLUSTER BY (column)
REMOVE CLUSTERING: CLUSTER BY (none)



When should I use Liquid Clustering?

Liquid clustering is especially useful when:

- Data is skewed
- Filtering columns have different cardinalities
- Tables regularly ingest lots of new data
- System needs to support concurrent writes
- Query filter patterns change over time
- Data might lead to the small file problem if traditionally partitioned

Liquid clustering is not compatible with Hive-style partitioning and Z-ordering

Note: Cluster by column should be within first 32 columns

- You can specify up to 4 clustering columns per Delta table.
- Your clustering columns need to be columns with statistics collected in the Delta logs. By default, the first 32 columns in a Delta table have statistics collected

Volumes in Unity Catalog:

Govern semi/un-structured data using UC.

Types: Managed. External

Working with managed Volumes:

create volume dev.bronze.managed_volume
comment 'create managed volume'

%sh wget https://media.githubusercontent.com/media/subhamkharwal/pyspark-zero-to-hero/refs/heads/master/datasets/emp.csv

%python dbutils.fs.mkdirs('/Volumes/dev/bronze/managed_volume/files')

%python

dbutils.fs.cp('file:/Workspace/Users/geramkumar_gmail.com#ext#@geramkumargmail.onmicrosoft.com/emp.csv','/Volumes/dev/bronze/managed_volume/files')

not recommended: *select * from* ***csv.****`/Volumes/dev/bronze/managed_volume/files/emp.csv`*

recommended*:*

*SELECT * FROM* ***read_files****(*

'/Volumes/dev/bronze/managed_volume/files/emp.csv',

format => 'csv',

header => true,

mode => 'FAILFAST')

Working with external Volumes:

create external volume dev.bronze.external_volume
location 'abfss://data@sauccentralindia.dfs.core.windows.net/adb/ext_volume'

%python dbutils.fs.mkdirs('/Volumes/dev/bronze/external_volume/files')

%python

dbutils.fs.cp('file:/Workspace/Users/geramkumar_gmail.com#ext#@geramkumargmail.onmicrosoft.com/emp.csv','/Volumes/dev/bronze/external_volume/files')

*SELECT * FROM read_files(*

'/Volumes/dev/bronze/external_volume/files/emp.csv',

format => 'csv',

header => true,

mode => 'FAILFAST')

DROP VOLUME dev.bronze.managed_volume

DROP VOLUME dev.bronze.external_volume

IMPORTANT DIFFERENCE between managed and external volumes:

When we drop a **managed volume**, volume definition will be deleted immediately. Data will be marked for deletion and deleted after **7 days**.

When we drop an **external volume**, only the volume will be deleted. Data **will not be deleted**.

DBUTILS (only for Python or Scala):

Common Utilities:

```
data: Work with DataFrames.
fs: Interact with files and directories.
jobs: Manage Databricks jobs.
library: Manage libraries and dependencies.
notebook: Work with notebooks and their metadata.
secrets: Manage secrets securely.
widgets: Create interactive widgets in notebooks.
Utilities API library: Access the underlying API for advanced customization.
```

```
dbutils.help()
```

DBUTILS.FS

```
dbutils.fs.help("cp")
df = spark.read.format('csv').load(
  '/databricks-datasets/Rdatasets/data-001/csv/ggplot2/diamonds.csv',
  header=True, inferSchema=True)
dbutils.data.summarize(df)
```

```
dbutils.fs.ls("/Volumes/main/default/my-volume/")
dbutils.fs.cp("/Volumes/main/default/my-volume/data.csv",
"/Volumes/main/default/my-volume/new-data.csv")
dbutils.fs.mv("/Volumes/main/default/my-volume/rows.csv",
"/Volumes/main/default/my-volume/my-data/")
dbutils.fs.put("/Volumes/main/default/my-volume/hello.txt", "Hello, Databricks!",
True)
dbutils.fs.rm("/Volumes/main/default/my-volume/my-data/", True)
dbutils.fs.head("/Volumes/main/default/my-volume/data.csv", 25)
```

```
dbutils.fs.mkdirs("/Volumes/main/default/my-volume/my-data")

dbutils.fs.mounts()


dbutils.fs.mount(

  source = "wasbs://<container-name>@<storage-account-name>.blob.core.windows.net",

  mount_point = "/mnt/<mount-name>",

  extra_configs = {"<conf-key>":dbutils.secrets.get(scope = "<scope-name>", key =
"<key-name>")})

dbutils.fs.unmount("/mnt/<mount-name>")
```

DBUTILS.JOBS

```
dbutils.jobs.taskValues.set(key   = "my-key", \

                            value = 5)

dbutils.jobs.taskValues.get(taskKey    = "my-task", \

                            key        = "my-key", \

                            default    = 7, \

                            debugValue = 42)
```

DBUTILS.NOTEBOOK

```
dbutils.notebook.run("My Other Notebook", 60) <- 60 timeout seconds

dbutils.notebook.exit("Exiting from My Other Notebook")
```

DBUTILS.SECRETS

```
dbutils.secrets.get(scope="my-scope", key="my-key")

dbutils.secrets.list("my-scope")
```

DBUTILS.WIDGETS

```
dbutils.widgets.text(

  name='your_name_text',

  defaultValue='Enter your name',

  label='Your name'

)

print(dbutils.widgets.get("your_name_text"))


dbutils.widgets.combobox(

  name='fruits_combobox',

  defaultValue='banana',
```

```
  choices=['apple', 'banana', 'coconut', 'dragon fruit'],
  label='Fruits'
)
print(dbutils.widgets.get("fruits_combobox"))
```

```
CREATE WIDGET COMBOBOX fruits_combobox DEFAULT "banana" CHOICES SELECT * FROM
(VALUES ("apple"), ("banana"), ("coconut"), ("dragon fruit"))
SELECT :fruits_combobox
```

```
dbutils.widgets.dropdown(
  name='toys_dropdown',
  defaultValue='basketball',
  choices=['alphabet blocks', 'basketball', 'cape', 'doll'],
  label='Toys'
)
print(dbutils.widgets.get("toys_dropdown"))
```

```
CREATE WIDGET DROPDOWN toys_dropdown DEFAULT "basketball" CHOICES SELECT * FROM
(VALUES ("alphabet blocks"), ("basketball"), ("cape"), ("doll"))
SELECT :toys_dropdown
```

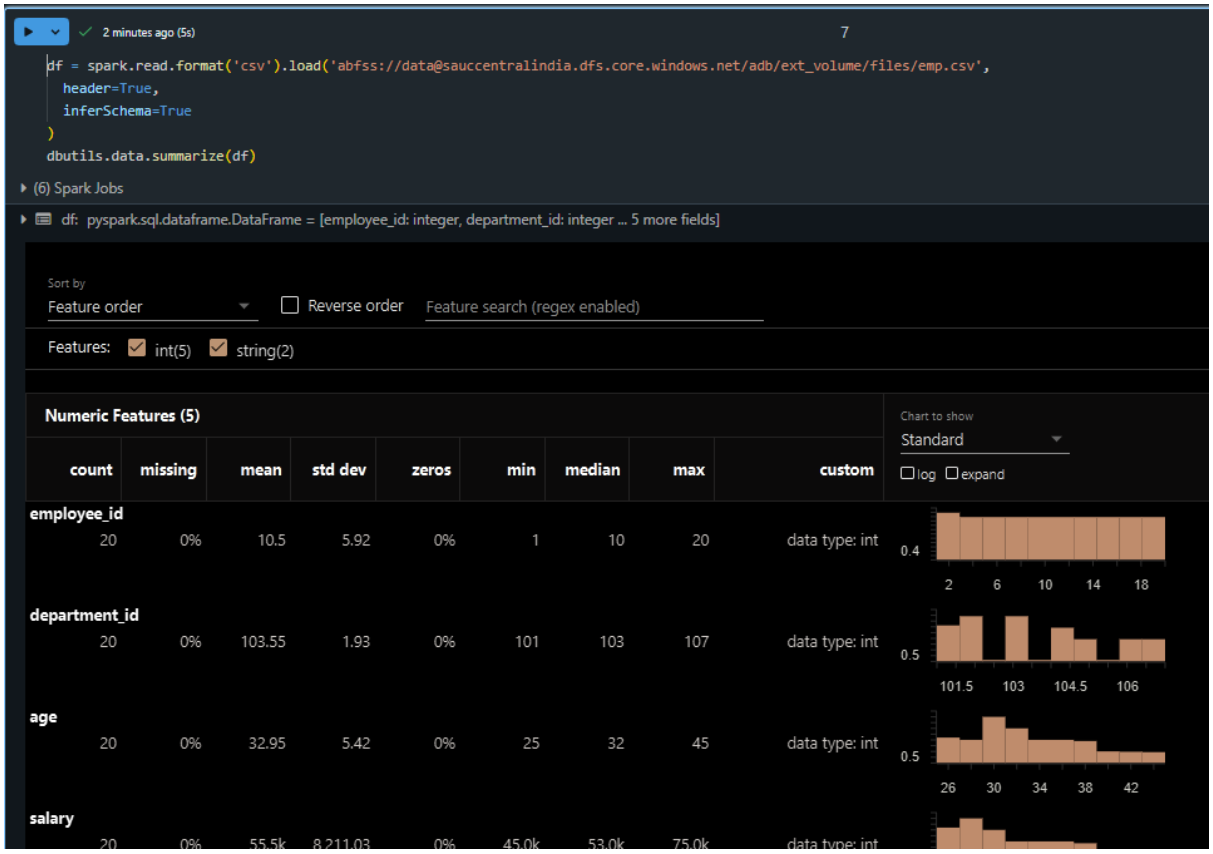
```
df = spark.sql("SELECT * FROM table where col1 = :param", dbutils.widgets.getAll())
```

```
dbutils.widgets.multiselect(
  name='days_multiselect',
  defaultValue='Tuesday',
  choices=['Monday', 'Tuesday', 'Wednesday', 'Thursday',
    'Friday', 'Saturday', 'Sunday'],
  label='Days of the Week'
)
print(dbutils.widgets.get("days_multiselect"))
```

```
dbutils.widgets.removeAll()
```

DBUTILS.DATA

```
df = spark.read.format('csv').load('abfss://data@sauccentralindia.dfs.core.windows.net/adb/ext_volume/files/emp.csv', header=True, inferSchema=True)
dbutils.data.summarize(df)
```



Jobs and Workflow

Databricks Workflows is a managed orchestration service within the Databricks platform that allows users to define, manage, and monitor data pipelines for ETL, analytics, and machine learning.

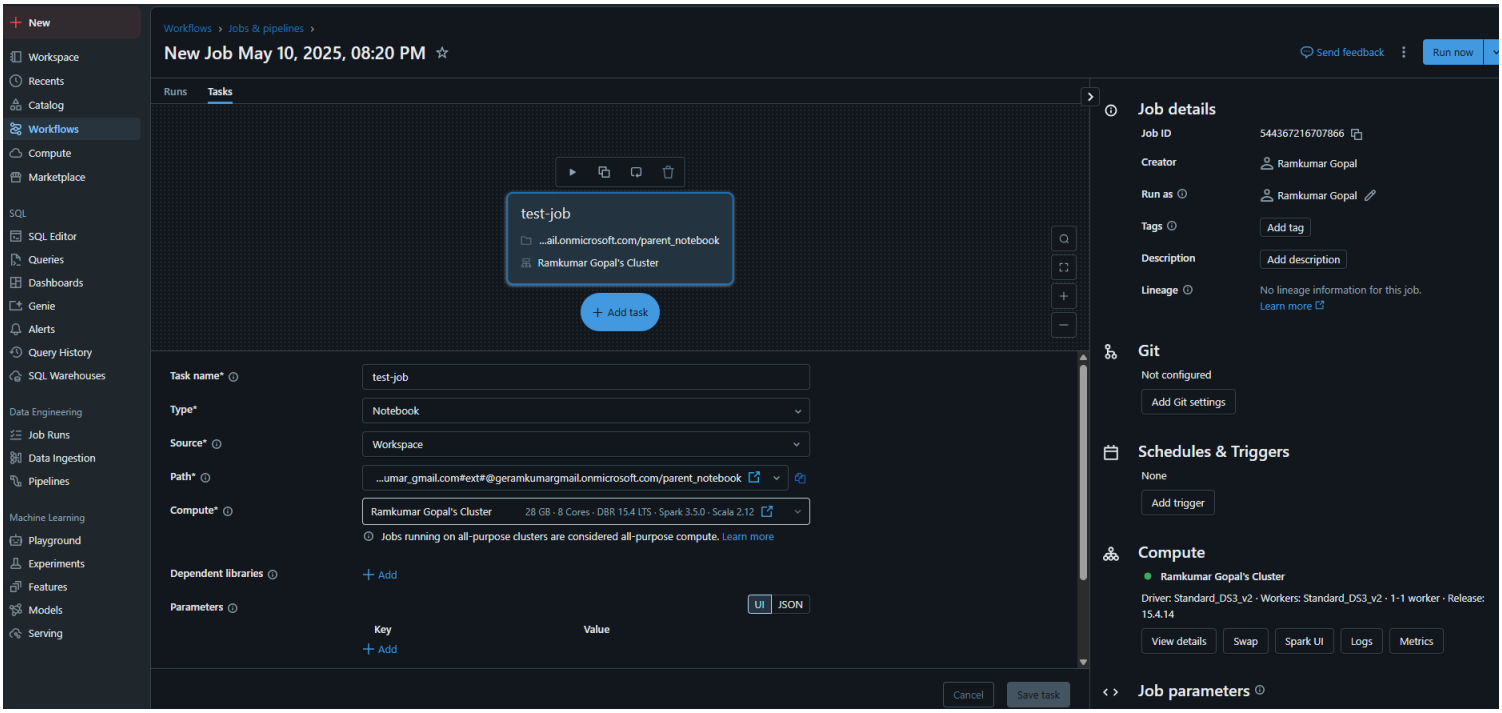
Job types:

Databricks Workflows support various job types to handle different data processing and analysis tasks. These include **Notebooks, Python scripts, JAR files, Databricks SQL queries, Delta Live Tables pipelines, dbt tasks**, and more. Jobs can also incorporate external systems like Azure Data Factory or dbt.

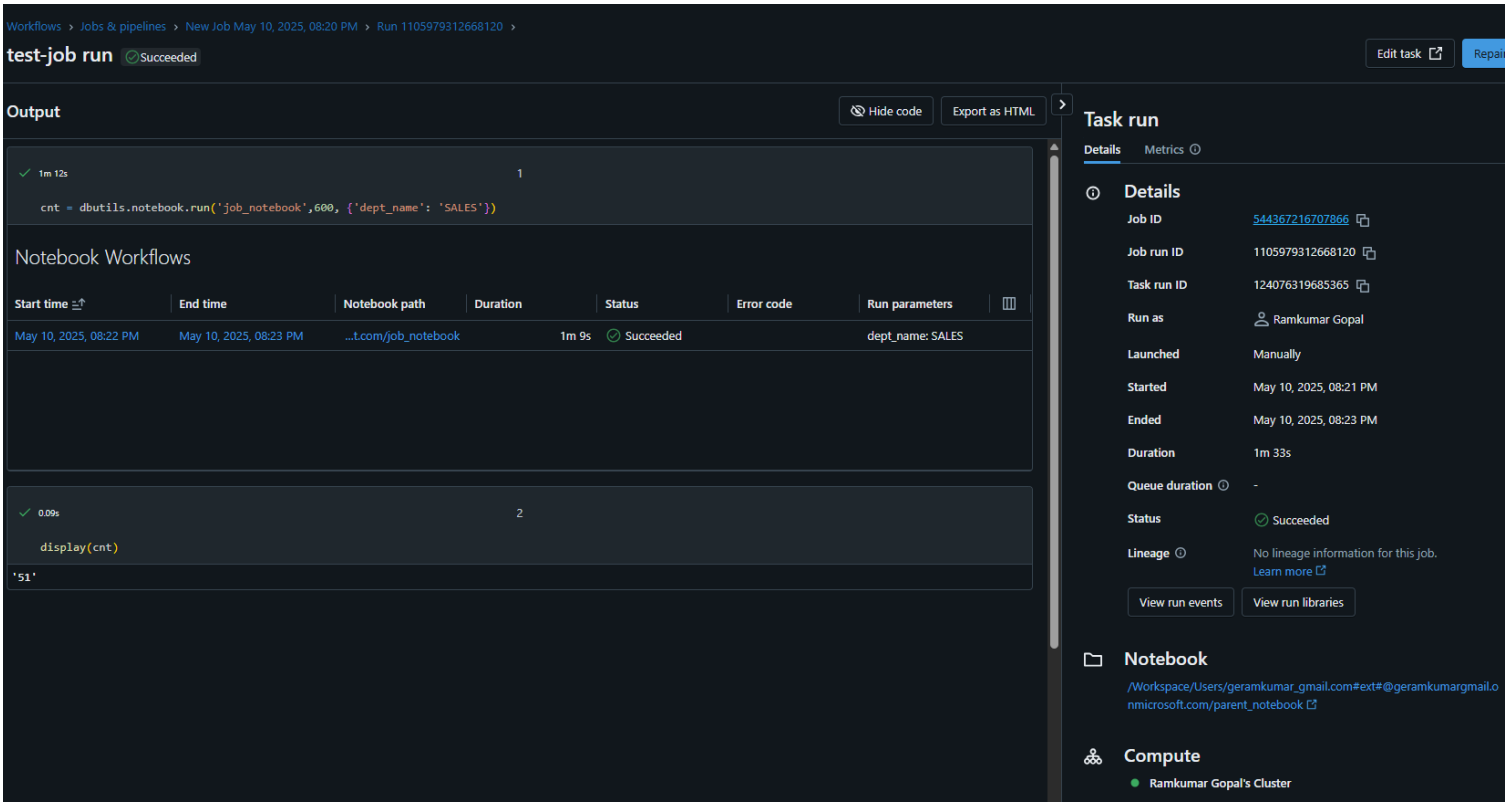
Types of workflows:

- **Jobs** are the underlying mechanism for scheduling and orchestrating tasks.
- **ETL pipelines (DLT)** focus on extracting, transforming, and loading data
- **ingestion pipelines** handle the initial data intake from various sources.

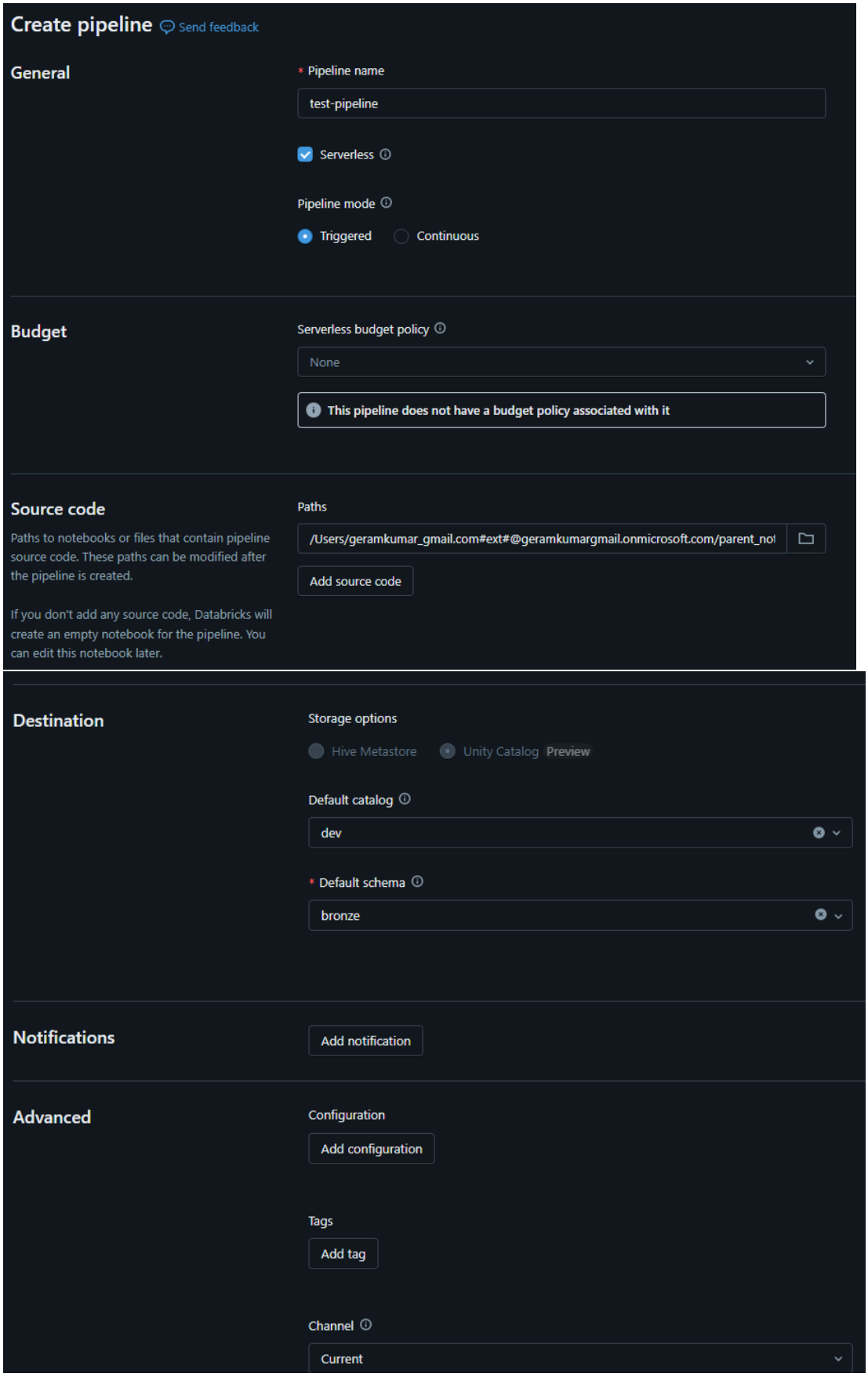
1. **Jobs**
   Creating a job

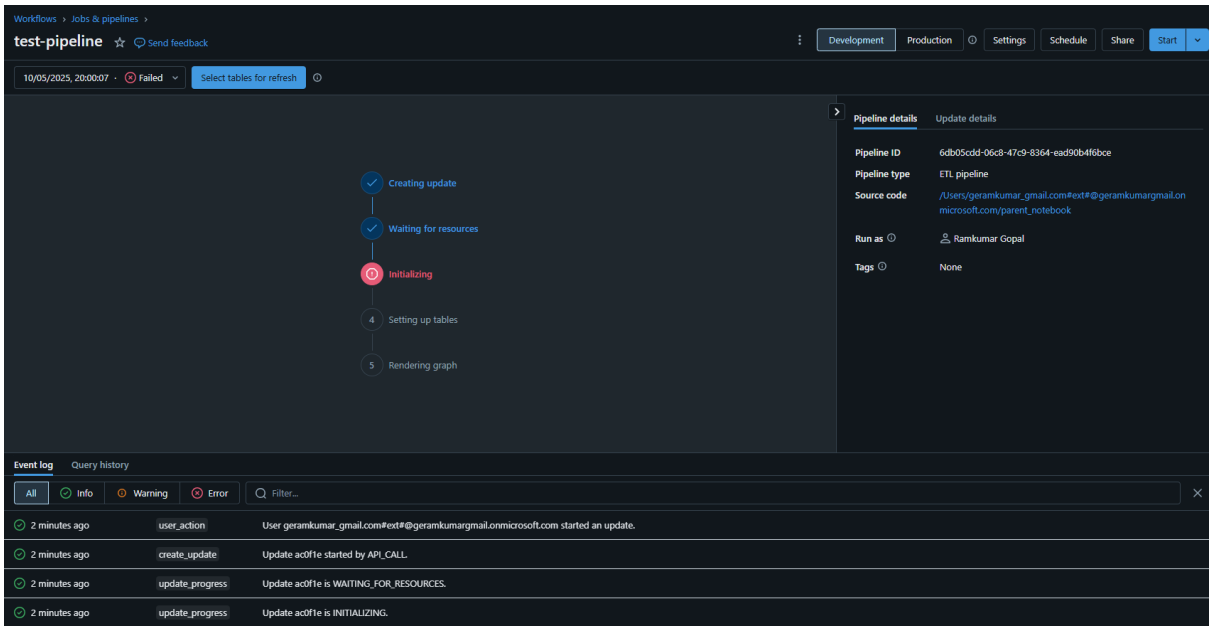


- **Running a job**

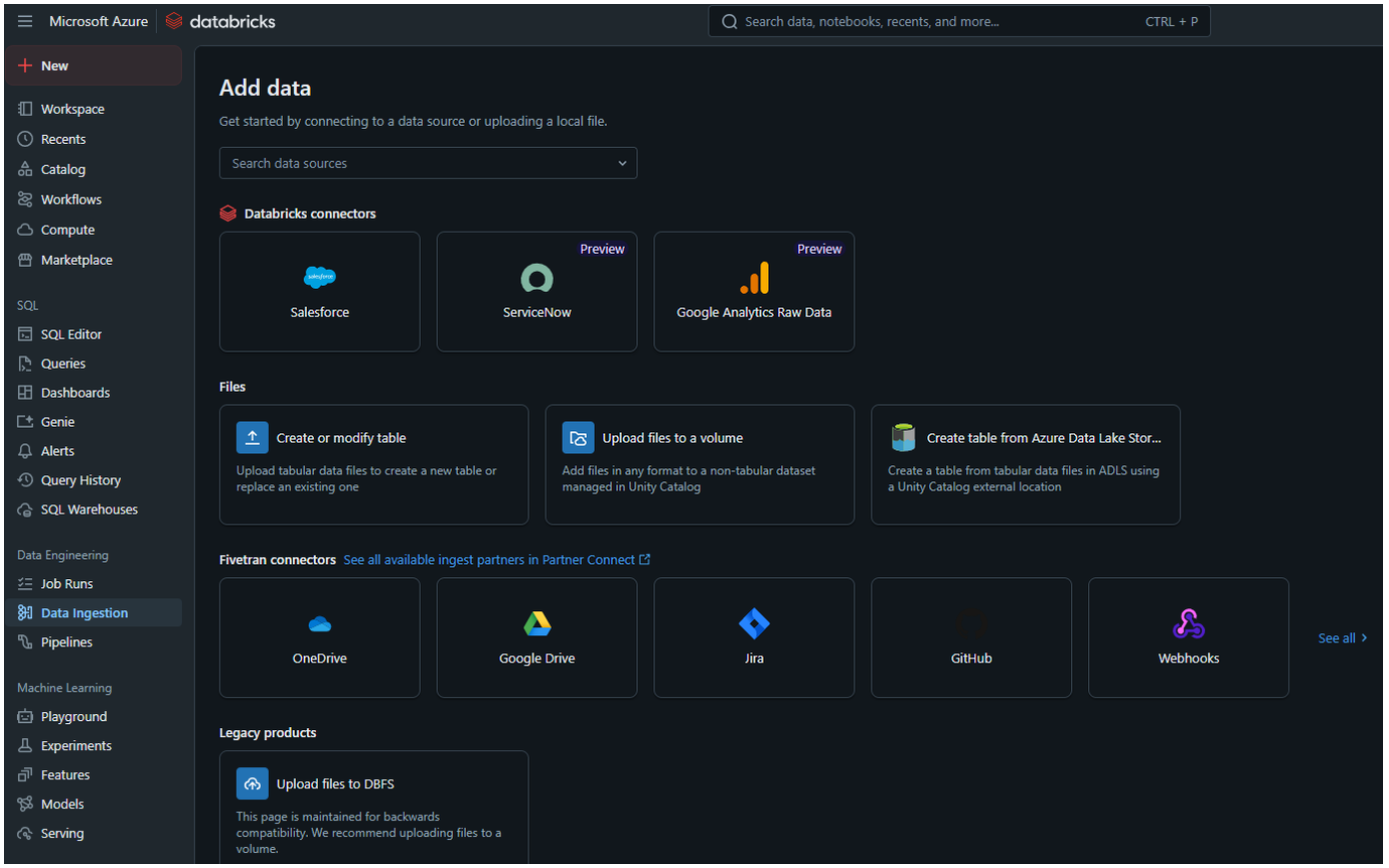


2. Pipeline (DLT – Core, Pro, Advanced Compute types)





3. Data Ingestion Pipeline



Types of triggers

1. Schedule-Based Trigger

- **Description**: Runs workflows at specific intervals (daily, weekly, hourly, etc.).
- **Use Case**: Automate ETL jobs or data refresh tasks on a routine basis.
- **Customization Options**:
- Cron expressions for precise scheduling.
- Timezone selection.

2. Event-Based Trigger (File Arrival)

- **Description**: Triggers workflows when a new file arrives in cloud storage (e.g., Azure Blob, AWS S3).
- **Use Case**: Automatically process files as soon as they arrive in a data lake.
- **Common Example**: Run a job when a new CSV or Parquet file is uploaded to Azure Blob Storage.

3. Job Dependency Trigger

- **Description**: Executes a workflow when another Databricks job completes (successfully or otherwise).
- **Use Case**: Chain multiple jobs where the output of one serves as the input to another.
- **Configuration**: You can specify whether the child workflow should trigger based on success, failure, or any status of the upstream job.

4. API Trigger (REST API)

- **Description**: Starts workflows through API calls.
- **Use Case**: Integrate Databricks workflows with external systems or CI/CD pipelines (like GitHub Actions or Jenkins).
- **How to Use**: Issue a POST request to the Databricks Jobs API endpoint to trigger the workflow programmatically.

5. Manual Trigger

- **Description**: Allows users to run the workflow manually via the Databricks UI or CLI.
- **Use Case**: Ad-hoc execution of workflows for testing or one-time processes.

6. Delta Live Tables (DLT) Trigger

- **Description**: Triggers pipelines based on changes in Delta Lake tables.
- **Use Case**: Automatically refreshes downstream data when new data is ingested into a Delta Lake table.

7. Webhook Trigger

- **Description**: Triggers workflows in response to external events using webhooks.
- **Use Case**: Activate Databricks jobs in response to external notifications (e.g., via an IoT device or web application).

COPY INTO (Retriable and idempotent. Data will be loaded exactly once. Process 1000s of files):

In Databricks, the COPY INTO command is used to load data from a source location (like cloud object storage) into a Delta table.

Key Features and Capabilities:

- **Simplified Ingestion:**

COPY INTO provides a direct and straightforward way to load data from cloud storage (S3, ADLS, GCS, etc.) into Databricks tables.

- **Format Support:**

It supports multiple source file formats, including CSV, JSON, XML, Avro, ORC, Parquet, and others.

- **Schema Evolution:**

COPY INTO can be configured to evolve the schema of the target table based on the input data.

- **Idempotency:**

If you rerun COPY INTO, it will not re-load data that has already been loaded.

- **File Filtering:**

You can use file or directory filters to select specific files for loading.

- **Concurrency:**

COPY INTO supports concurrent invocations against the same table, but it's generally better to use a single invocation with multiple files for performance.

- **Validate Mode:**

The COPY INTO Validate mode allows you to preview and validate source data before ingesting.

- **Scheduling:**

COPY INTO can be run ad-hoc or scheduled using Databricks Workflows.

- **Error Handling:**

COPY INTO provides options for handling corrupted records during parsing

Example:

```
CREATE TABLE IF NOT EXISTS my_table (
  loan_id BIGINT,
  funded_amnt INT,
  paid_amnt DOUBLE,
  addr_state STRING
);

COPY INTO my_table
FROM '/databricks-datasets/learning-spark-v2/loans/loan-risks.snappy.parquet'
FILEFORMAT = PARQUET
FORMAT_OPTIONS ('mergeSchema' = 'true')
COPY_OPTIONS ('mergeSchema' = 'true');

SELECT * FROM my_table;
```

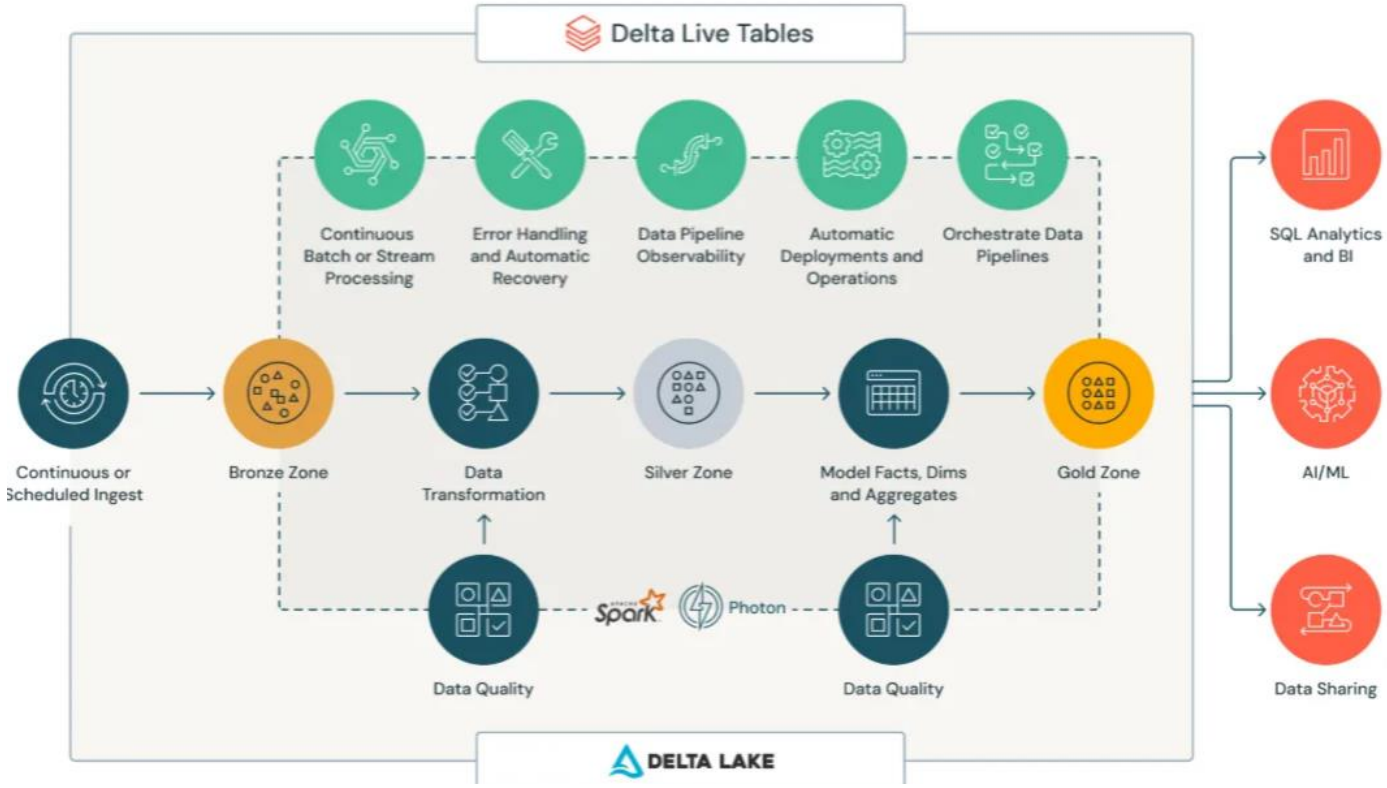
Auto Loader (Process millions of files per minute)

Databricks Auto Loader is a feature that simplifies and optimizes data ingestion from cloud storage into Databricks. Its key features include

- efficient file discovery
- schema inference and evolution
- exactly-once processing guarantees
- cost-effectiveness
- scalability
- support for various file formats.

```
# Assuming you have a Databricks cluster and have configured cloud storage access
from pyspark.sql.functions import *
from pyspark.sql import SparkSession

# Create a Spark session
spark = SparkSession.builder.appName("AutoLoaderExample").getOrCreate()

# Define the source path and checkpoint location
source_path = "s3://my-bucket/data/"
checkpoint_path = "dbfs:/my/checkpoint/path"

# Create a read stream from the cloudFiles source
read_stream = spark.readStream \
    .format("cloudFiles") \
    .option("cloudFiles.format", "json") \
    .load(source_path)

# You can perform any transformations on the data if needed
# For example, you might want to extract specific fields or perform aggregations
transformed_data = read_stream \
    .select("field1", "field2") # Example transformation

# Create a write stream to a Delta Lake table
transformed_data \
    .writeStream \
    .format("delta") \
    .option("checkpointLocation", checkpoint_path) \
    .outputMode("append") \
    .toTable("my_table")  # Replace with your desired table name
```

DLT



- ✅ **Incremental data transformation**
  ✅ **Change Data Capture (CDC) support**
  ✅ **Type 2 Slowly Changing Dimensions (SCD2) handling out of the box**
- **However, DLT is not recommended for:**
- ❌ **Writing to external tables**
  ❌ **Complex transformations that require extensive custom logic**

DLT Terminologies (data is updated incrementally. Streaming helps to append the data):

- **DLT is a framework for creating batch and streaming data pipelines in SQL and Python**
- **Use cases:**
    - Ingestion from sources such as cloud storage (such as Amazon S3, Azure ADLS Gen2, and Google Cloud Storage)
    - Ingestion from message buses (such as Apache Kafka, Azure EventHub
    - Incremental batch and streaming transformations.
- Streaming tables vs Materialized Views
    - Streaming tables are used for ingesting and transforming append-only data streams.
    - Streaming tables are ideal when you need to process data as it arrives in real-time or near real-time

    - Materialized views are preferred for pre-computing slow queries and frequently used computations.
    - Materialized views are better for optimizing queries on relatively static data.

You can and often should use streaming tables for the bronze and silver layers in Delta Live Tables (DLT) and materialized views for the gold layer

Features of DLT:

- CDC Support
- SDC Support
- Data Quality Enforcement
- Data Lineage and Monitoring

CREATE INCREMENTAL LIVE TABLE – Depreciated. Still supported
CREATE STEAMING LIVE TABLE - Depreciated. Still supported
CREATE STREAMING TABLE – Recommended

Sample script 1:

```
CREATE OR REPLACE STREAMING TABLE customers
COMMENT "Customers RAW"
AS SELECT * FROM
cloud_files("/databricks-datasets/retail-org/customers/", "csv");

CREATE OR REPLACE STREAMING TABLE sales_orders_raw
COMMENT "Sales Orders RAW"
AS
SELECT * FROM cloud_files("/databricks-datasets/retail-org/sales_orders/",
"json", map("cloudFiles.inferColumnTypes", "true"))

CREATE  OR REPLACE STREAMING TABLE sales_orders_cleaned(
  CONSTRAINT valid_order_number EXPECT (order_number IS NOT NULL) ON
VIOLATION DROP ROW)
PARTITIONED BY (order_date)
AS
SELECT
      f.customer_id,
```

```
      f.customer_name,
      f.number_of_line_items,
      TIMESTAMP(from_unixtime((cast(f.order_datetime as long)))) as
order_datetime,
      DATE(from_unixtime((cast(f.order_datetime as long)))) as order_date,
      f.order_number,
      f.ordered_products,
      c.state,
      c.city,
      c.lon,
      c.lat,
      c.units_purchased,
      c.loyalty_segment
FROM STREAM(LIVE.sales_orders_raw) f
LEFT JOIN LIVE.customers c
      ON c.customer_id = f.customer_id AND c.customer_name =
f.customer_name

CREATE OR REPLACE MATERIALIZED VIEW sales_order_in_la
AS
SELECT city, order_date, customer_id, customer_name,
ordered_products_explode.curr,
      SUM(ordered_products_explode.price) as sales,
      SUM(ordered_products_explode.qty) as qantity,
      COUNT(ordered_products_explode.id) as product_count
FROM (
  SELECT city, order_date, customer_id, customer_name,
EXPLODE(ordered_products) as ordered_products_explode
  FROM LIVE.sales_orders_cleaned
  WHERE city = 'Los Angeles'
  )
GROUP BY order_date, city, customer_id, customer_name,
ordered_products_explode.curr
```

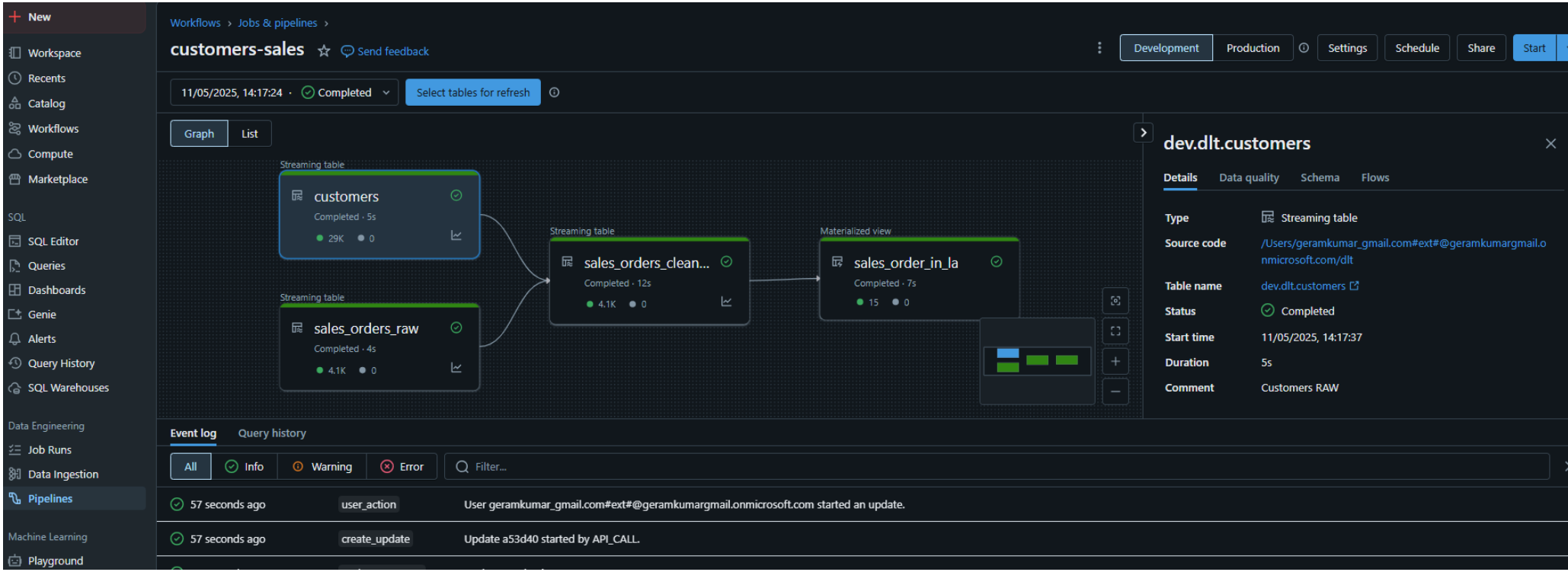
Sample for SCD 1 and SCD 2

```
CREATE OR REFRESH STREAMING TABLE product_silver;
APPLY CHANGES INTO live.product_silver
FROM stream(LIve.products)
KEYS (product_id)
APPLY AS DELETE WHEN operation = "DELETE"
SEQUENCE BY seqNum
COLUMNS * EXCEPT (operation,seqNum ,_rescued_data,ingestion_date)
STORED AS SCD TYPE 1;

CREATE OR REFRESH STREAMING TABLE customer_silver;
APPLY CHANGES INTO live.customer_silver
FROM stream(LIve.customers)
KEYS customer_id)
APPLY AS DELETE WHEN operation = "DELETE" SEQUENCE BY sequenceNum
COLUMNS * EXCEPT (operation,sequenceNum ,_rescued_data,ingestion_date)
STORED AS SCD TYPE 2;
```

COPY INTO Vs Auto Loader Vs Delta Live Table

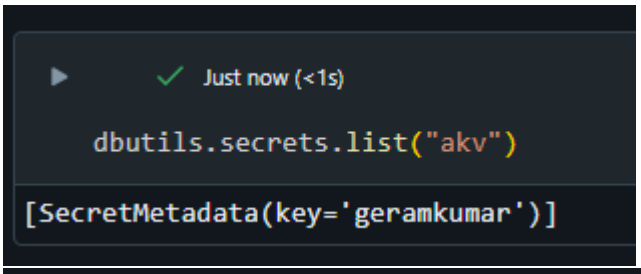
Feature / Use Case	`COPY INTO`	Auto Loader	Delta Live Tables (DLT)
Ingestion Mode	Batch	Streaming (micro-batch)	Streaming or Batch (Managed)
Ideal for	Historical/batch loads	Continuous ingestion	Full pipelines (Bronze → Silver → Gold)
Automation	Manual or scheduled	Semi-automatic	Fully managed, orchestrated
Schema Evolution	Supported (limited)	✅ Yes	✅ Yes
File Formats Supported	CSV, JSON, Parquet	CSV, JSON, Parquet, Avro, ORC	Same as Auto Loader
Change Data Capture (CDC) Support	❌ No	❌ No	✅ Yes (via `apply_changes`)
Data Quality Checks	❌ No	❌ No	✅ Yes (via `@dlt.expect`)
Lineage and Monitoring	❌ No	❌ No	✅ Yes (in UI)
Retry / Failure Handling	❌ Manual	✅ Checkpoints	✅ Built-in retries
Code Type	SQL	Python (Structured Streaming)	SQL / Python (Declarative)
Best Fit	Simple file ingestion	Real-time Bronze layer ingestion	Full production pipeline with validation & tracking

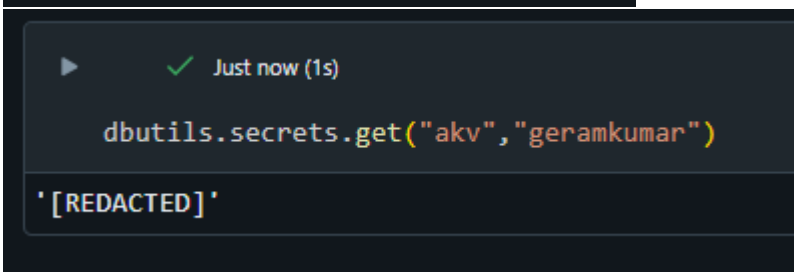
DLT Pipeline sample



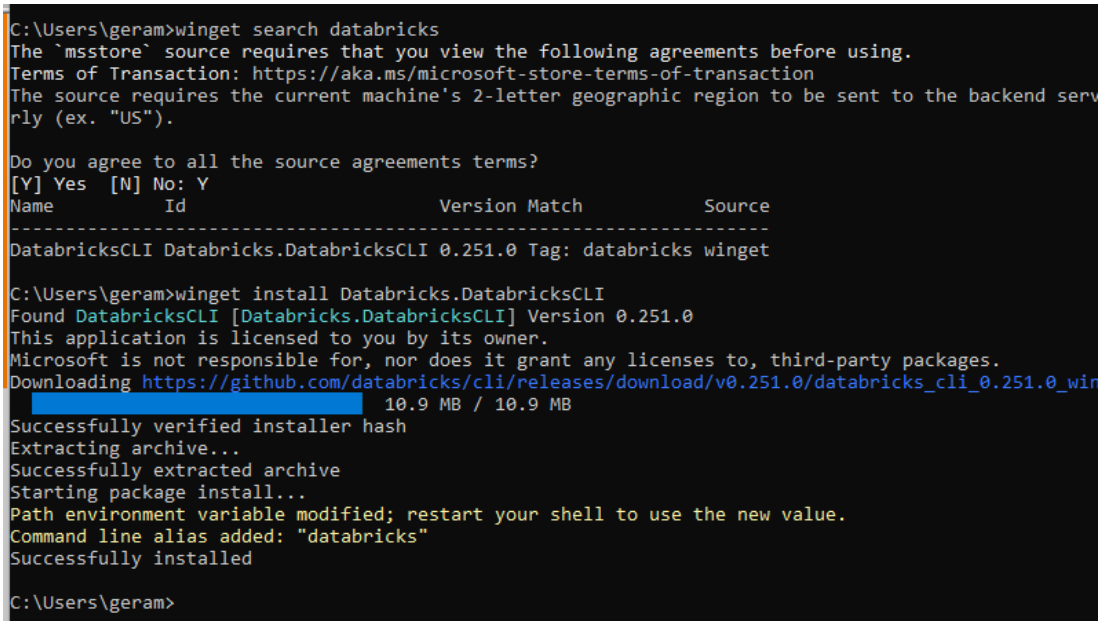
Databricks Secret Management:

Secret Scopes:

- **Azure Key Vault Backed**
  - Create KV in Azure.
  - In AKV, Create a new secret. geramkumar/Password@123
  - In Databricks workspace url, include #secrets/createScope
    https://adb-4248328932240573.13.azuredatabricks.net/?o=4248328932240573**#secrets/createScope**
  - **create a new secret scope named "akv"**
  - 

  - 

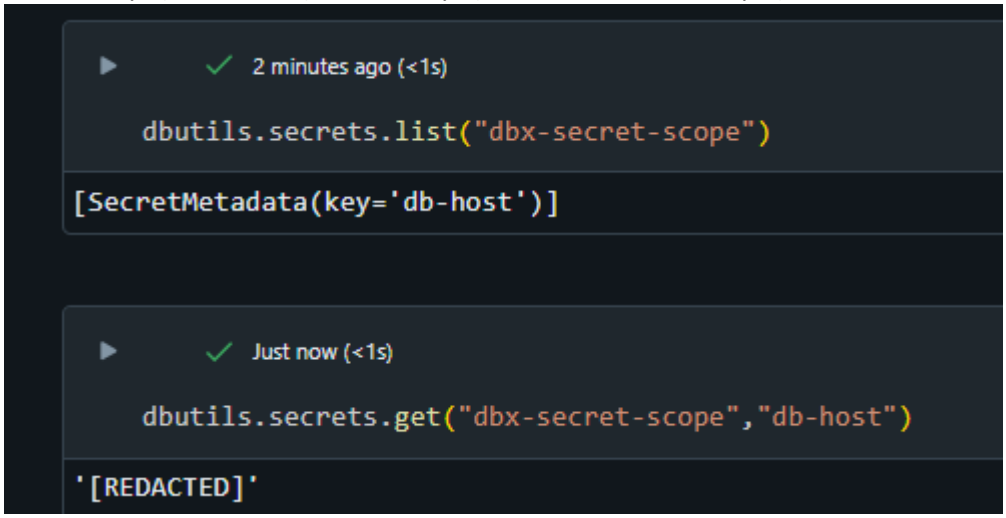
- **Databricks Backed**
  - **Install Databricks CLI (command prompt)**
    - **winget search databricks**
      winget install Databricks.DatabricksCLI



- C:\Users\geram>databricks -v
  Databricks CLI v0.251.0

- C:\Users\geram>**databricks auth login**
  Databricks profile name: dbxauth
  Databricks host (e.g. https://<databricks-instance>.cloud.databricks.com):
  https://adb-4248328932240573.13.azuredatabricks.net
  Profile dbxauth was successfully saved
  C:\Users\geram>

- **C:\Users\geram>databricks secrets list-scopes**
  Scope Backend Type
  akv AZURE_KEYVAULT
- C:\Users\geram>**databricks secrets create-scope dbx-secret-scope**
- C:\Users\geram>**databricks secrets list-scopes**
  Scope Backend Type
  akv AZURE_KEYVAULT
  dbx-secret-scope DATABRICKS
- C:\Users\geram>**databricks secrets put-secret dbx-secret-scope db-host --string-value=secrethost.com**

- In Databricks Workspace
  - dbutils.secrets.listScopes()
    [SecretScope(name='akv'), SecretScope(name='dbx-secret-scope')]
  - 

Databricks CLI

- **Install Databricks CLI (command prompt)**
  - **winget search databricks**
    winget install Databricks.DatabricksCLI

    Generate token in DBX Workspace -> Developer -> Access Token

- C:\Users\geram>**databricks configure --token**

  Databricks host: https://adb-4248328932240573.13.azuredatabricks.net
  Personal access token: **********

- C:\Users\geram>**databricks fs ls /**

- C:\Users\geram>**databricks workspace list /**

ID	Type	Language	Path
2739774414175509	DIRECTORY		/Users
2739774414175510	DIRECTORY		/Shared
2739774414175515	DIRECTORY		/Repos

- C:\Users\geram>**databricks clusters list**

ID	Name	State
0511-075927-1s4wa59j	dlt-execution-1b84eb9a-4264-472a-87da-e1580af5f663	TERMINATED
0511-080711-u1sf4d3q	dlt-execution-1b84eb9a-4264-472a-87da-e1580af5f663	TERMINATED
0511-082029-z2pz8pue	dlt-execution-1b84eb9a-4264-472a-87da-e1580af5f663	TERMINATED
0511-083944-531gzb8o	dlt-execution-1b84eb9a-4264-472a-87da-e1580af5f663	TERMINATED
0510-145853-mzf2hek6	dlt-execution-230c7173-5af2-4fd3-8d16-2642aba827e8	TERMINATED
0510-143405-mijahcw3	Ramkumar Gopal's Cluster	TERMINATED

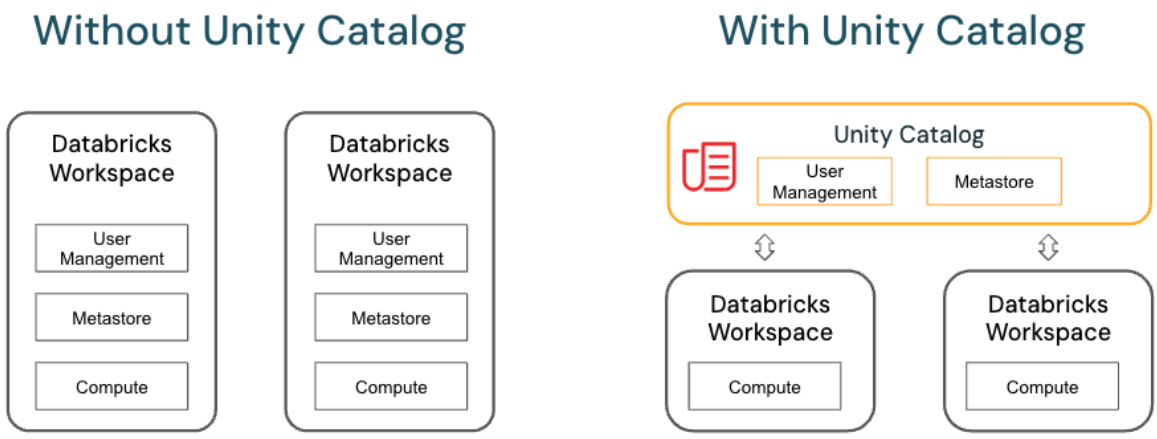
- **Databricks clusters start <cluster-id>**
- **databricks libraries list**
- **databricks instance-pools list**
- **databricks jobs list**

Databricks Identity Management:

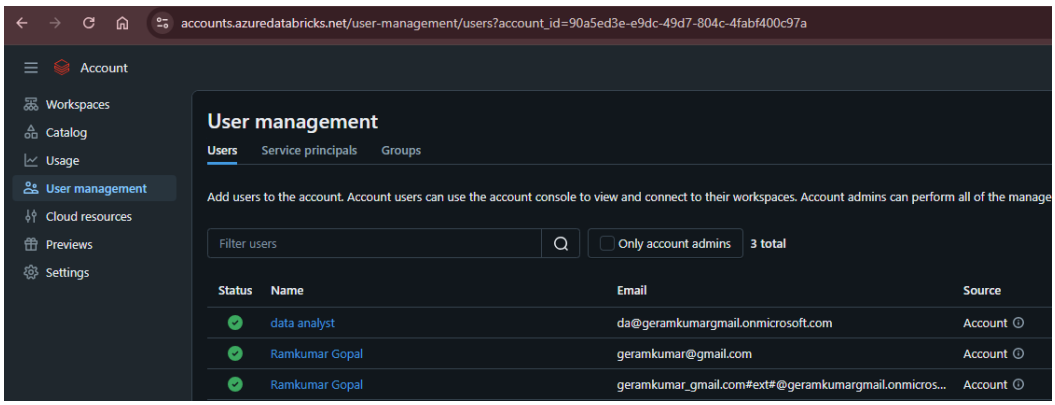
- **Account Admin:** Administrative access on the account console to configure account-level components (workspaces, NCC, etc)
- **Metastore Admin:** Manage Unity Catalog metastore and its components and design downstream permissions model
- **Workspace Admin:** Administrative access on a Databricks workspace to configure workspace-level components (clusters, jobs, users/groups, warehouses etc)

Databricks Users, Groups and Service Account Management

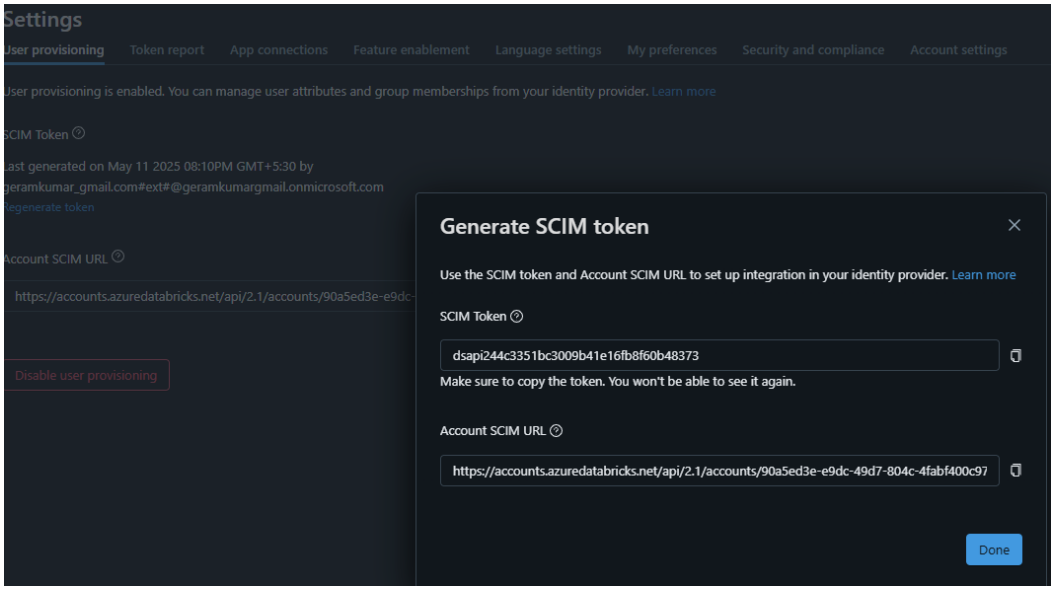
- With unity catalog, users of all workspaces can be managed centrally



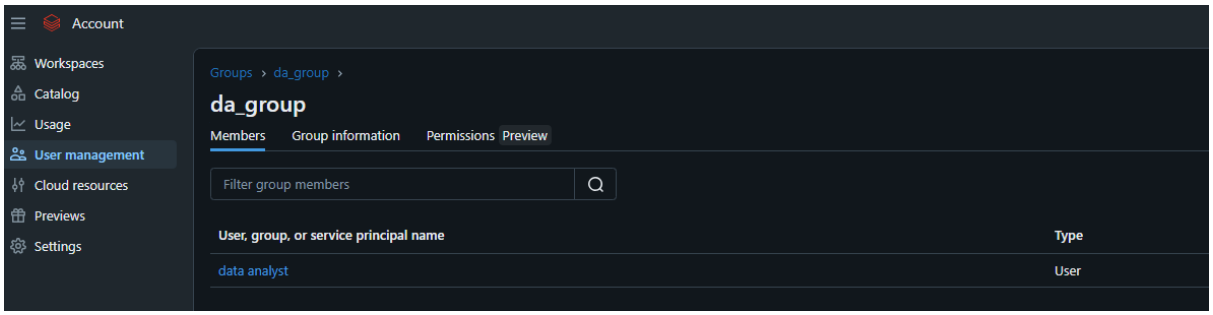
- **Add a new user named "da" in Entra ID**
- **Add the same user in accounts.databricks**



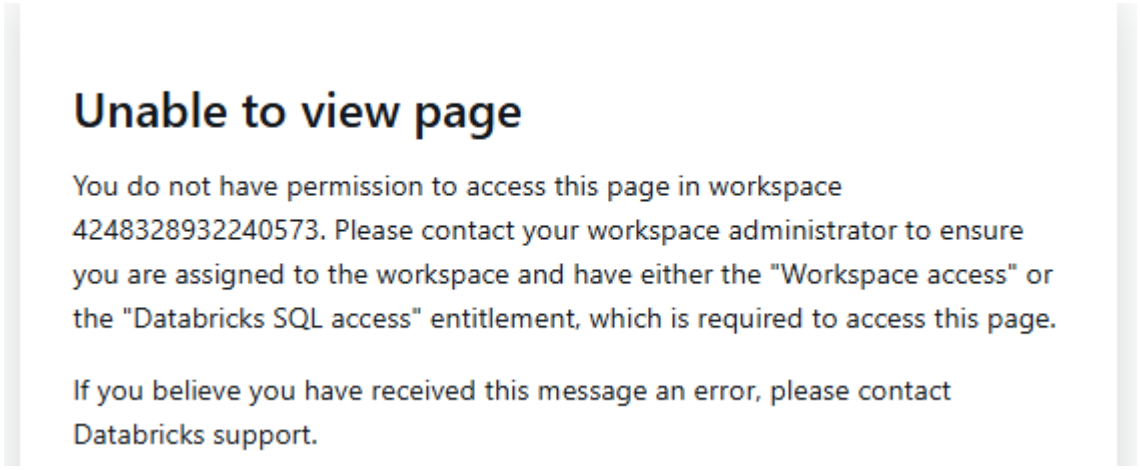
Note: To auto provision Azure users in Databricks automatically via SCIM seeting.
Under accounts.databricks.com -> Settings -> User provision



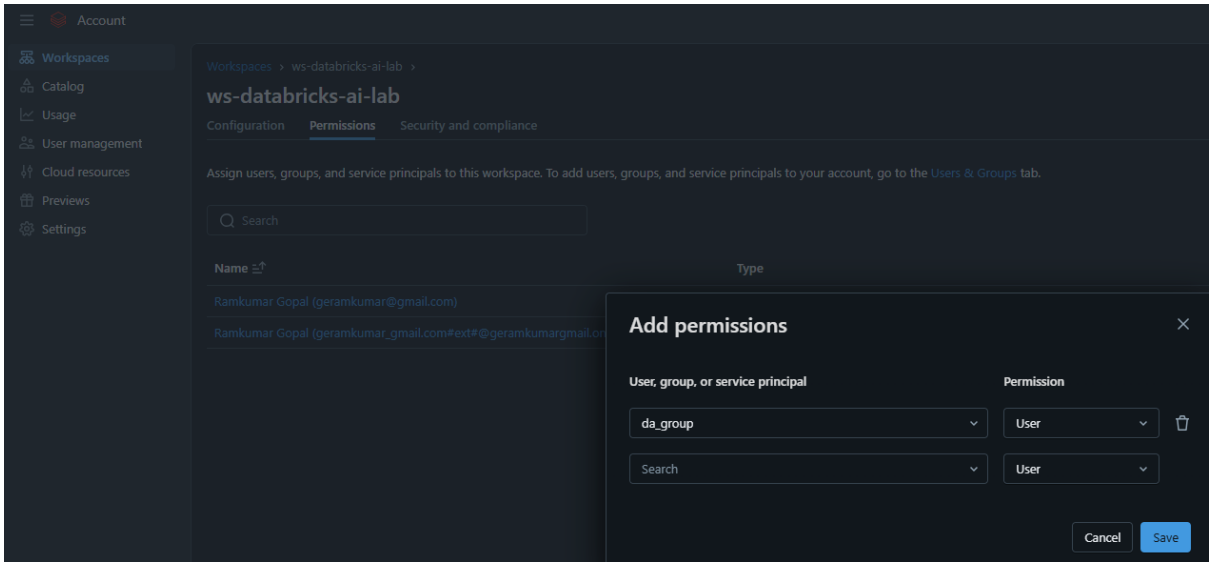
- Create Group and Add members



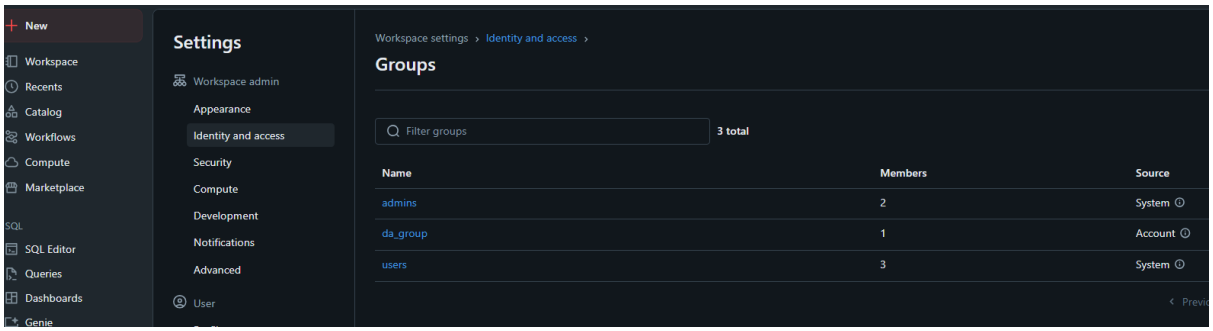
- Now, Da User can’t access the workspace as he doesn’t have the access



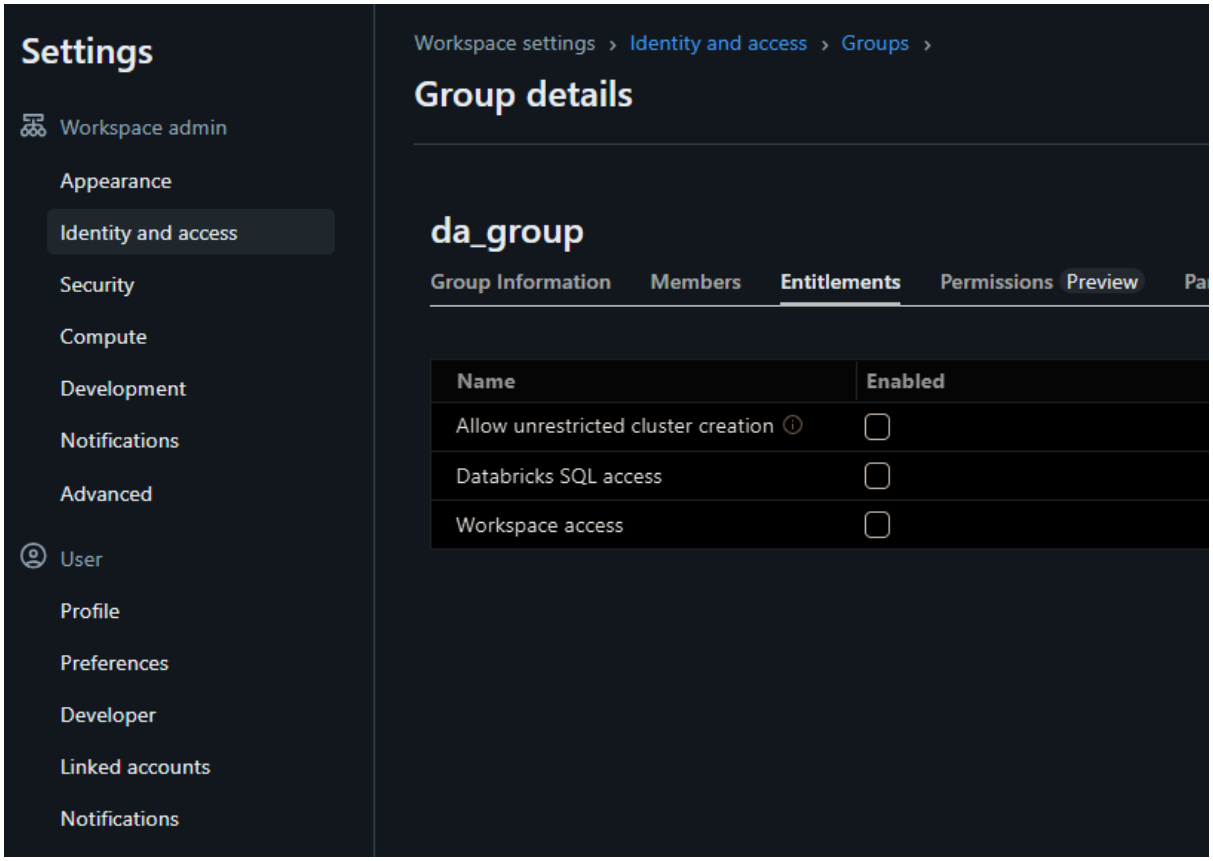
- In accounts.databricks -> workspace -> Permissions
  Add the user or group in the workspace
  Give the permission as user or admin



- **Then provide workspace level access within the workspace**
  Note: user/group created in accounts.databricks will automatically reflect in workspace

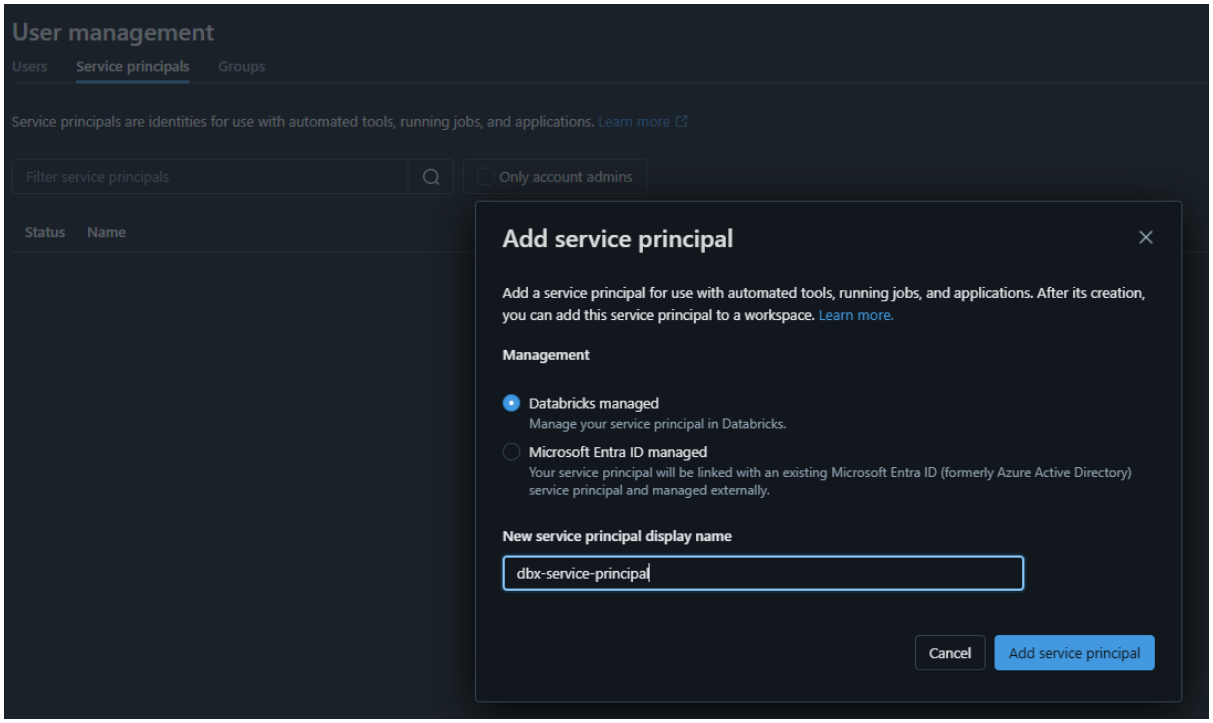


By default, workspace level entitiements won't be given to the users created in accounts.

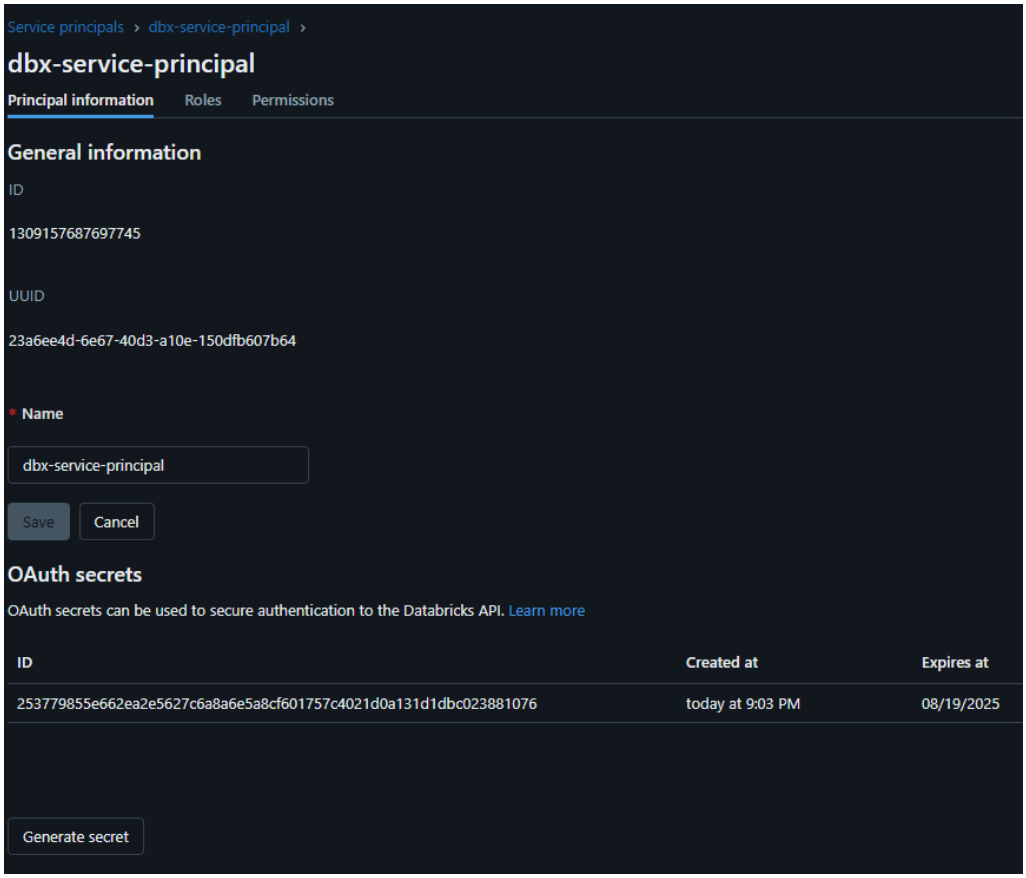


Service Principal (Databricks scope, Entra Scope):

Primarily for running jobs

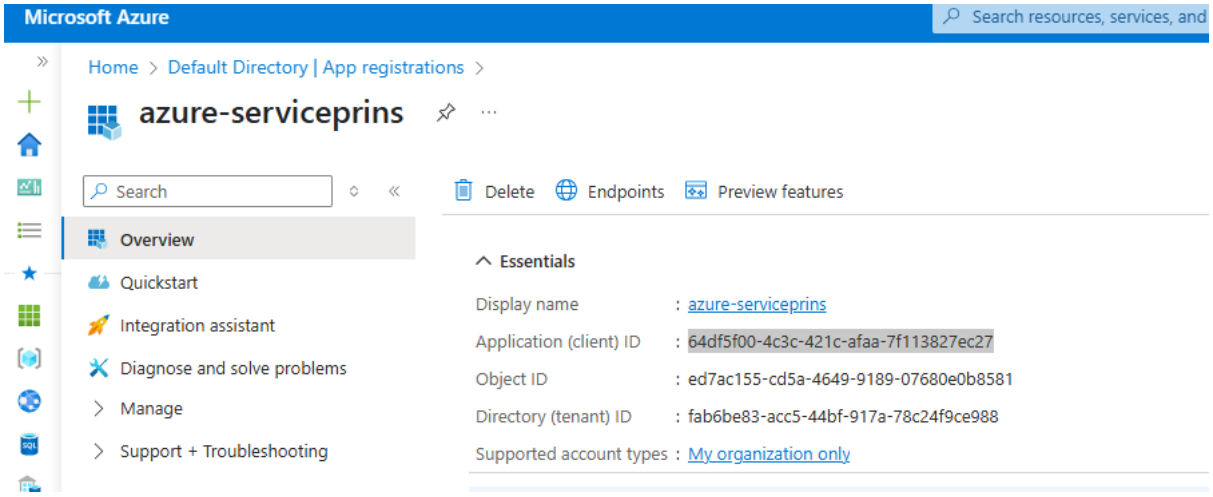


Then generate secret

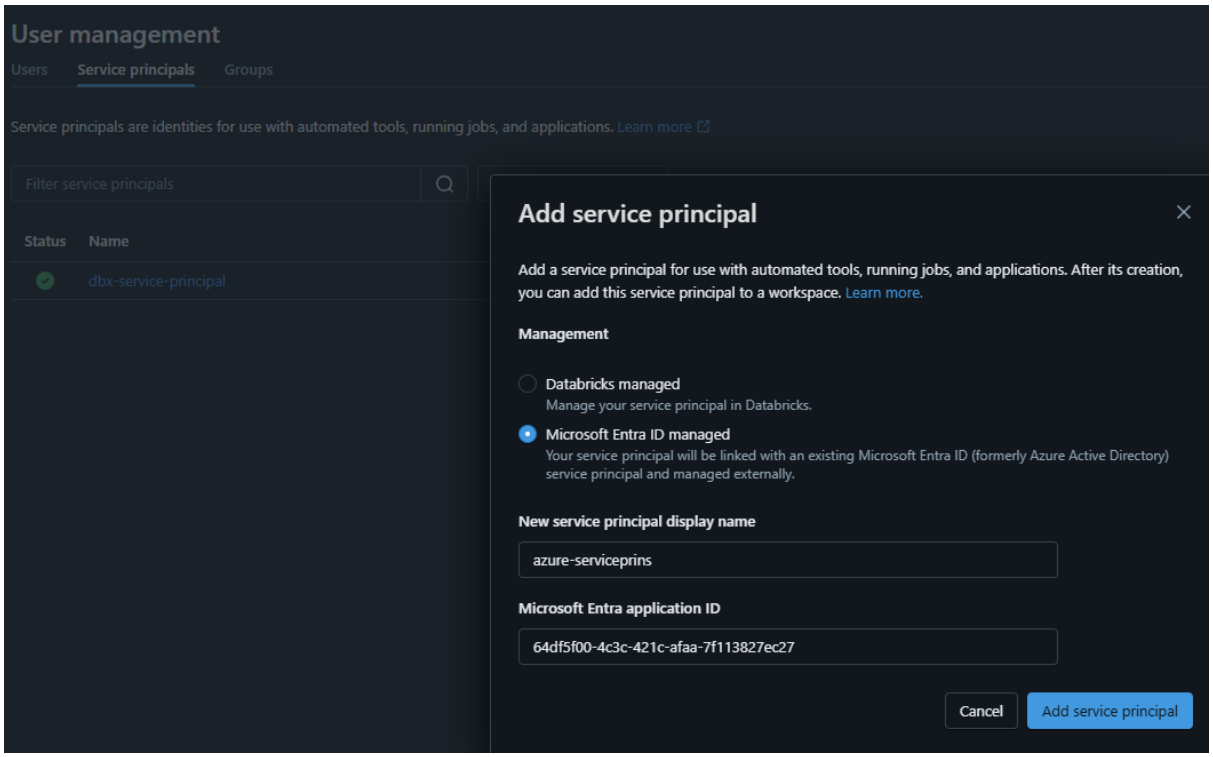


Azure Entra based Service Principal:

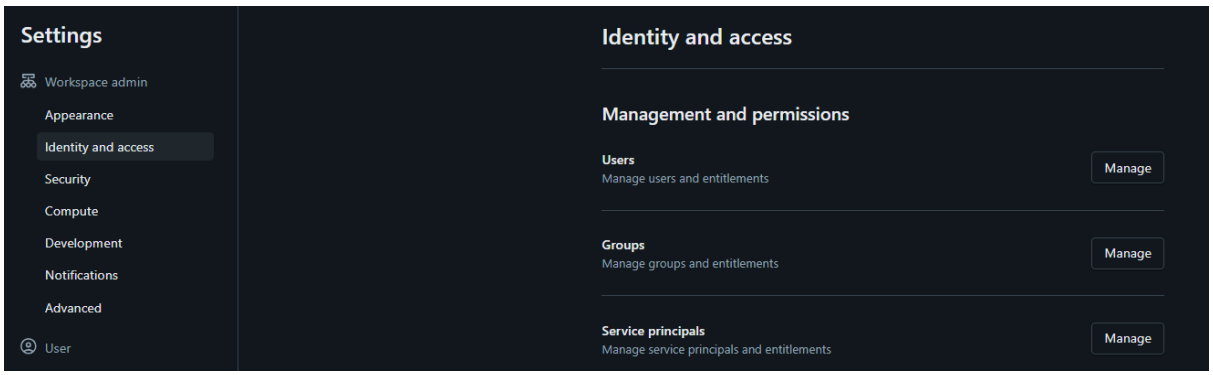
Azure -> Entra ID -> App registration



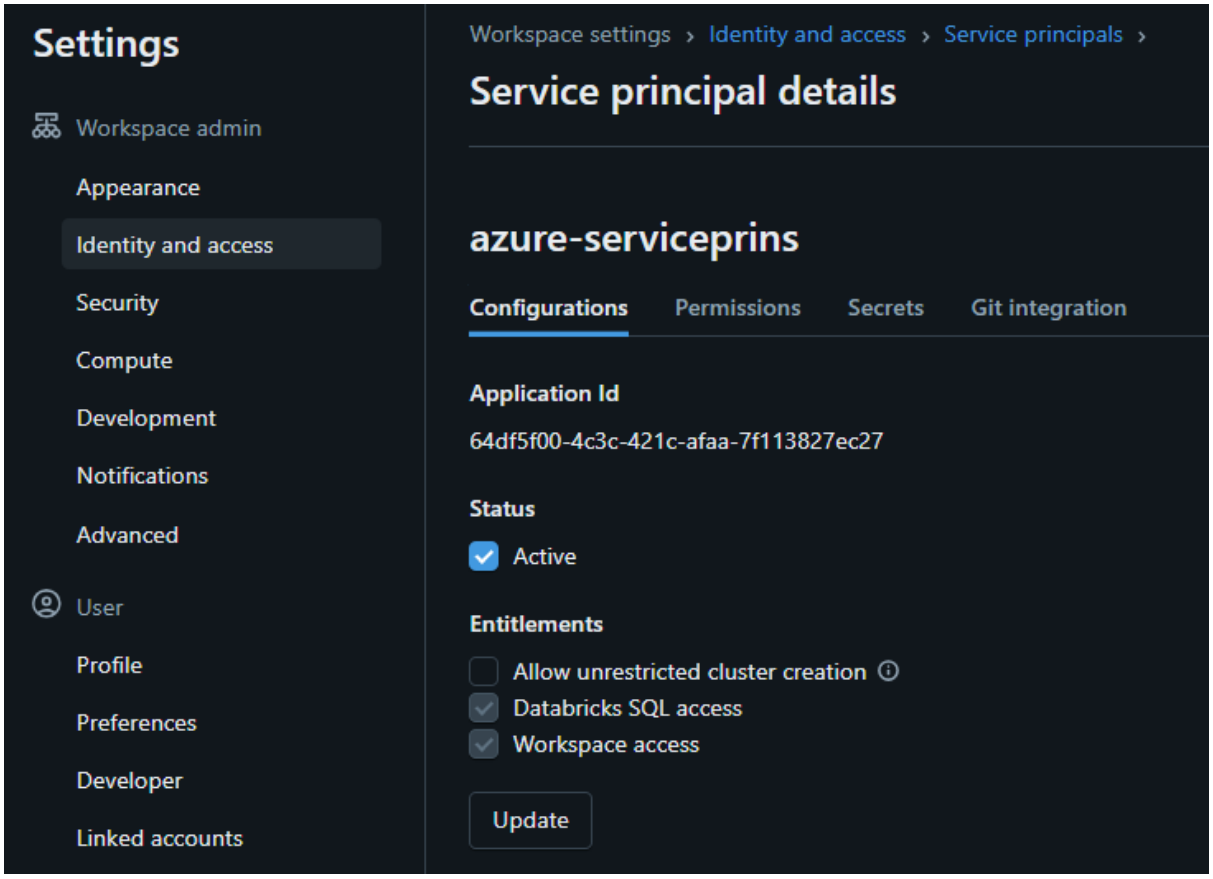
And add the sp in accounts.databricks.com



Then in workspace -> “Identity and Access” -> Select “Service Principal”

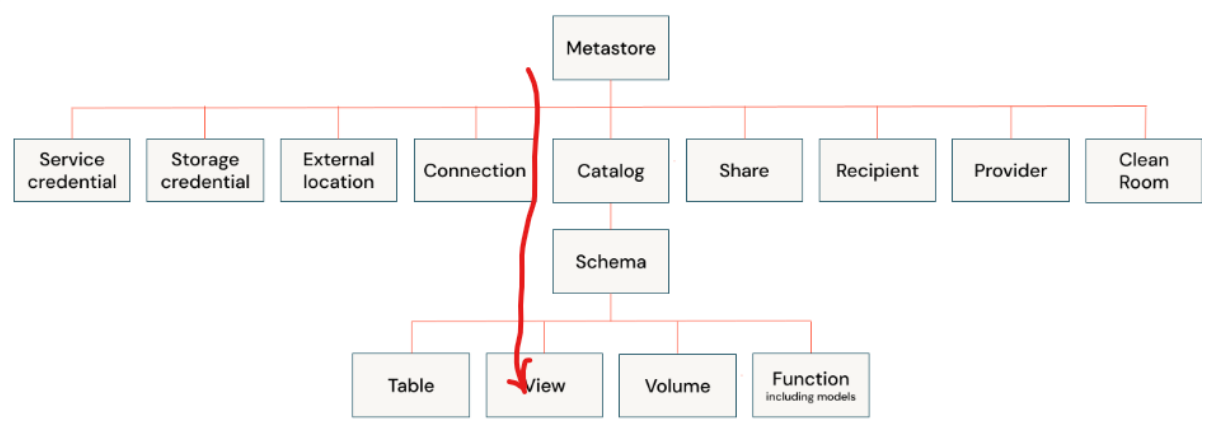


Its automatically sync’d in workspace:


Settings
Workspace admin
Appearance
Identity and access
Security
Compute
Development
Notifications
Advanced
User
Profile
Preferences
Developer
Linked accounts
Workspace settings › Identity and access › Service principals ›
Service principal details
azure-serviceprins
Configurations
Permissions
Secrets
Git integration
Application Id
64df5f00-4c3c-421c-afaa-7f113827ec27
Status
Active
Entitlements
Allow unrestricted cluster creation
Databricks SQL access
Workspace access
Update

Object Level Permissions in Unity Catalog

objects in Unity Catalog are hierarchical, and privileges are inherited downward.



Top down Inheritence:

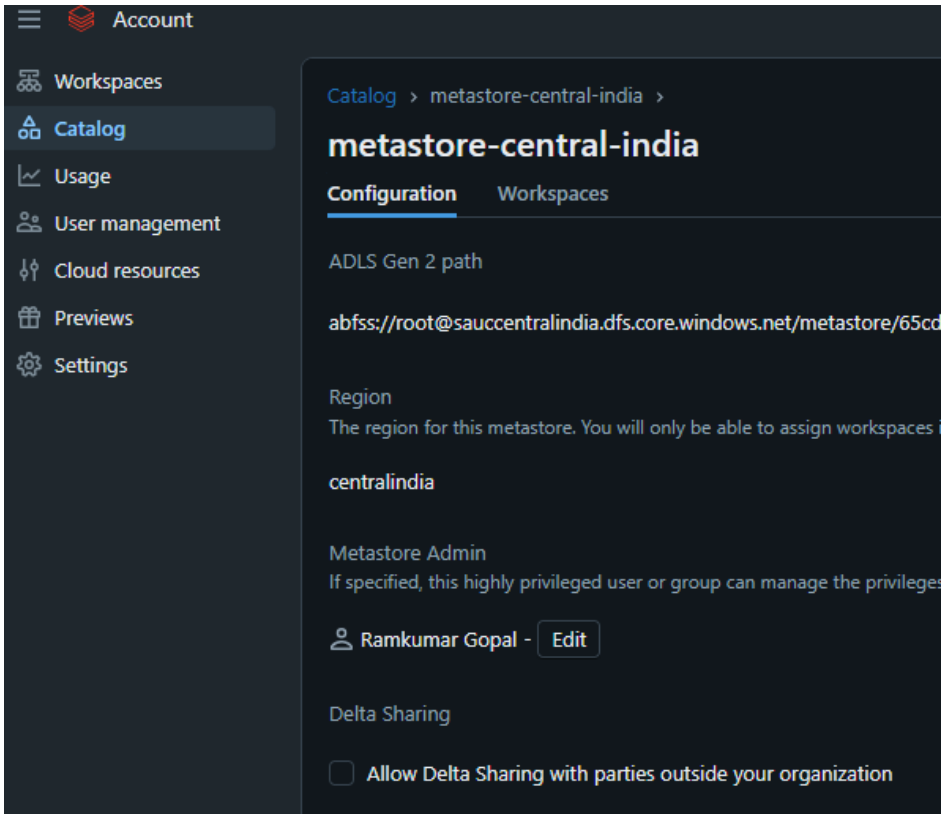
- Metastore admin has access to all catalogs
- If permission is given in CATALOG level, it will be inherited in SCHEMA level
- If permission is given in SCHEMA level, it will be inherited in all Objects level

Bottem:

- **If select table is given to user, user can access only that object**

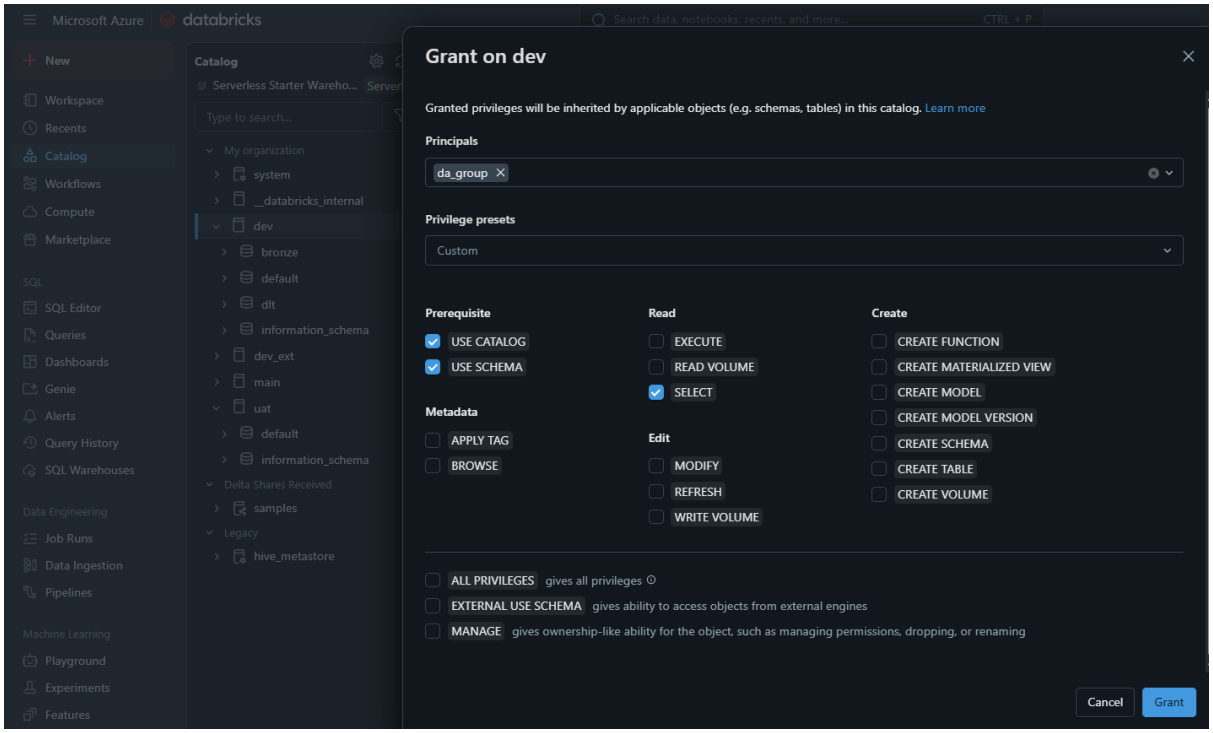
Note: Owners of an object are automatically granted all privileges on that object

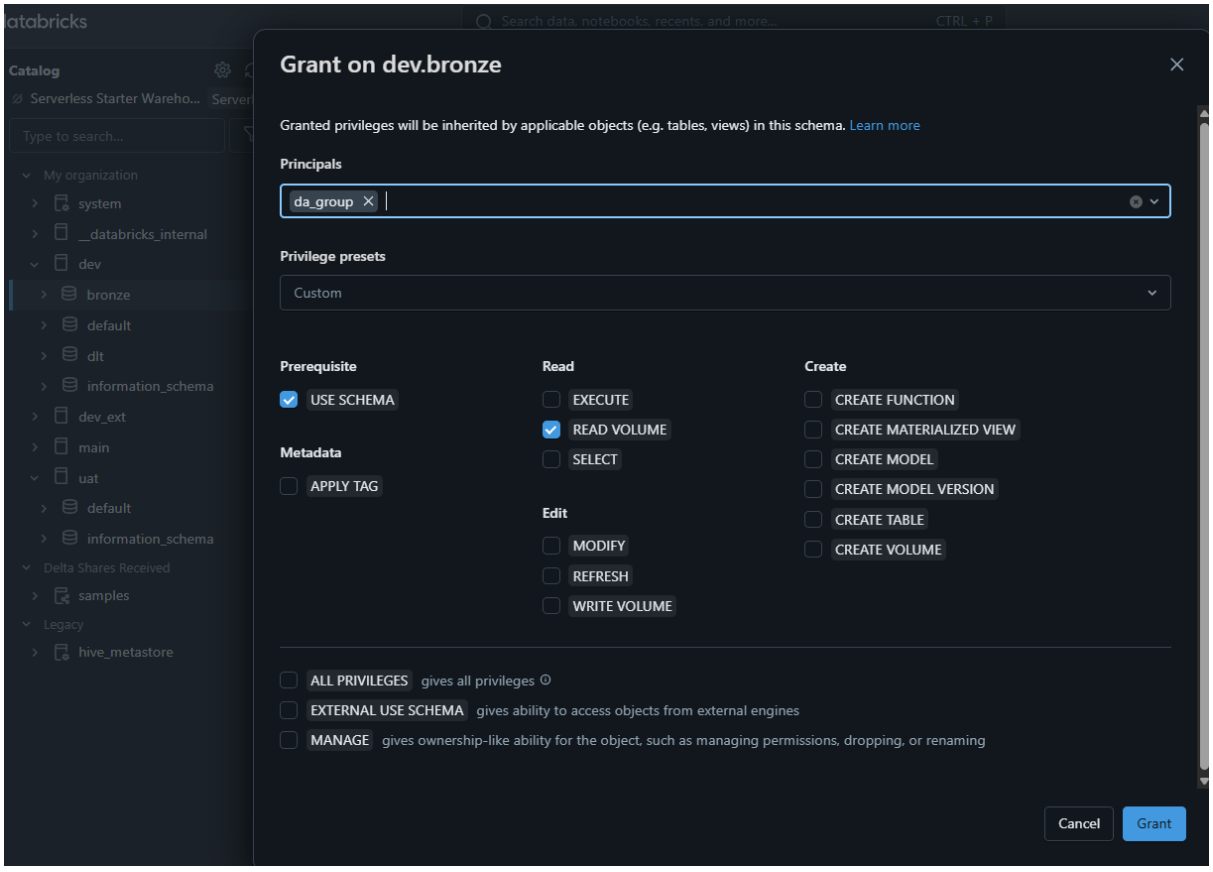
Metastore Admins:



BY DEFAULT ALL USERS HAVE BROWSE PERMISSION IN CATALOG level.

In Databricks Unity Catalog, all users have USE CATALOG privilege on the main catalog by default. Additionally, workspace users receive USE CATALOG, USE SCHEMA, CREATE TABLE, CREATE VOLUME, CREATE MODEL, CREATE FUNCTION, and CREATE MATERIALIZED VIEW privileges on the workspace catalog's default schema





In Databricks SQL, backticks (` `) are used to enclose table, column, and schema names that contain special characters (like hyphens, spaces, or reserved words), or when the name is case-sensitive. They are also necessary for referencing names in a case-sensitive manne

Commands:

%sql SELECT current_user();

%sql SHOW GRANTS `geramkumar_gmail.com#ext#@geramkumargmail.onmicrosoft.com` ON METASTORE

CATALOG LEVEL:

1. **Specify Privileges:** Choose the appropriate privileges for the catalog. Examples include:
    - **USE CATALOG**: Allows users to use the catalog to access its schemas and tables.
    - **CREATE SCHEMA**: Allows users to create schemas within the catalog.
    - **CREATE TABLE**: Allows users to create tables within the catalog.
    - **CREATE FUNCTION**: Allows users to create functions within the catalog.
    - **ALL PRIVILEGES**: Grants all available privileges on the catalog.

EXAMPLE: GRANT USE CATALOG, CREATE SCHEMA ON CATALOG main TO `user@example.com`;

SCHEMA LEVEL:

You can grant permissions like SELECT, CREATE TABLE, MODIFY, and ALL_PRIVILEGES at the schema level.

OBJECT LEVEL:

Understanding Privileges:

- SELECT: Grants read access to data within the object (e.g., a table).
- CREATE: Allows creating new objects within a schema (e.g., a new table).
- MODIFY: Grants the ability to add, delete, and modify data in the object.
- USAGE: An additional requirement to perform any action on a schema object.
- ALL PRIVILEGES: Grants all applicable privileges on the object.

Examples:

grant create catalog on metastore to `da_group`

revoke create catalog on metastore from `da_group`

GRANT SELECT ON CATALOG dev TO `da@geramkumargmail.onmicrosoft.com`

GRANT SELECT ON SCHEMA dev.bronze TO `da@geramkumargmail.onmicrosoft.com`

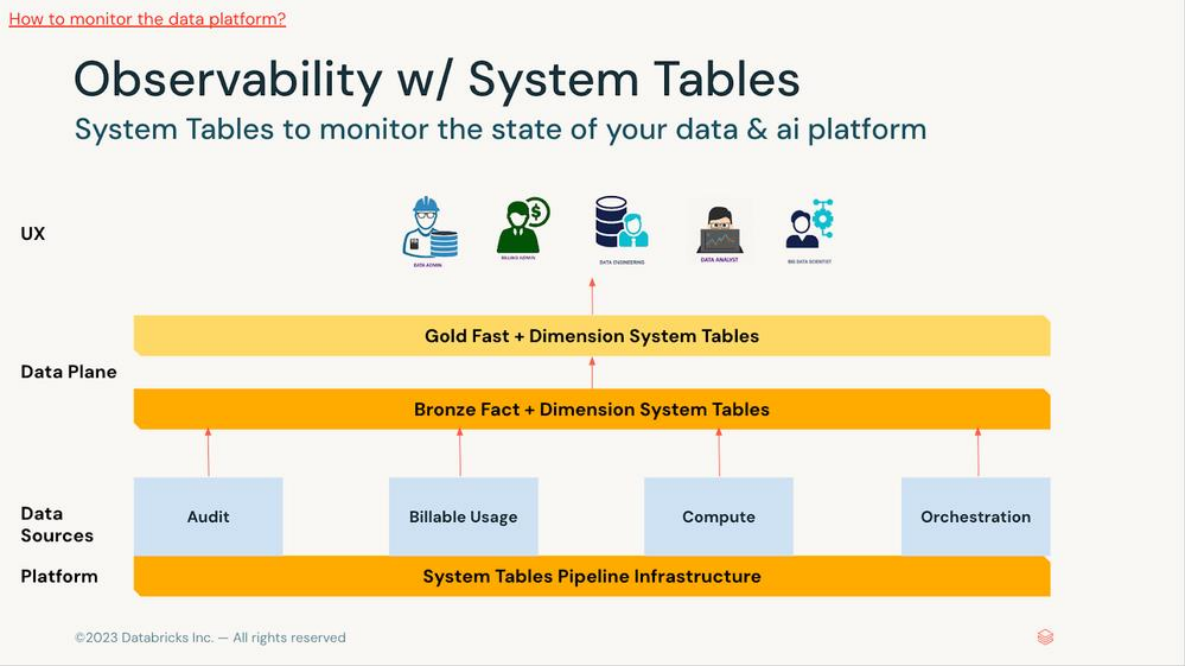
SHOW GRANTS `da@geramkumargmail.onmicrosoft.com` ON SCHEMA dev.bronze

GRANT CREATE TABLE ON SCHEMA main.default TO `finance-team`;
GRANT USE SCHEMA ON SCHEMA main.default TO `finance-team`;
GRANT USE CATALOG ON CATALOG main TO `finance-team`;

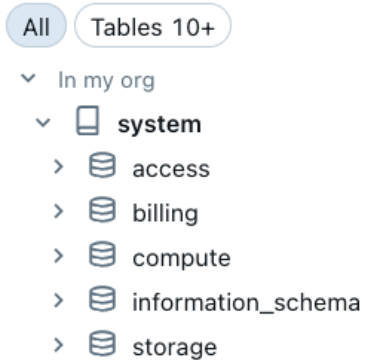
GRANT EXECUTE ON FUNCTION prod.ml_team.iris_model TO `ml-team-acme`;

REVOKE <privilege-type> ON <securable-type> <securable-name> FROM <principal>
REVOKE CREATE TABLE ON SCHEMA main.default FROM `finance-team`;

System tables in Unity Catalog



System catalog tables:



```
SELECT * FROM system.information_schema.catalogs
SELECT * FROM system.information_schema.schemata where catalog_name = 'dev'
SELECT * FROM system.information_schema.tables where table_catalog = 'dev'
SELECT * FROM system.information_schema.views
SELECT * FROM system.information_schema.columns

SELECT * FROM system.information_schema.metastore_privileges
SELECT * FROM system.information_schema.catalog_privileges
SELECT * FROM system.information_schema.schema_privileges
SELECT * FROM system.information_schema.table_privileges

SELECT * FROM system.information_schema.volumes
SELECT * FROM system.access.audit
SELECT * FROM system.access.audit

SELECT * FROM system.storage.predictive_optimization_operations_history


WITH agg_data AS (
 SELECT ds, sku1 as sku, workspace_id, SUM(dbus) AS dbus, SUM(cost_at_list_price) AS cost_at_list_price
 FROM
  (
    select workspace_id, ds, dbus, cost_at_list_price, sku,
      CASE
        WHEN sku LIKE '%ALL_PURPOSE%' THEN 'ALL_PURPOSE'
        WHEN sku LIKE '%JOBS%' THEN 'JOBS'
        WHEN sku LIKE '%DLT%' THEN 'DLT'
        WHEN sku LIKE '%SQL%' THEN 'SQL'
        WHEN sku LIKE '%INFERENCE%' THEN 'MODEL_INFERENCE'
        ELSE 'OTHER'
      END AS sku1
    from
      (
        SELECT u.workspace_id, u.usage_date AS ds, u.sku_name AS sku,
          CAST(u.usage_quantity AS DOUBLE) AS dbus,
          CAST(lp.pricing.default * usage_quantity AS DOUBLE) AS cost_at_list_price
        FROM
          system.billing.usage u
          INNER JOIN system.billing.list_prices lp ON u.cloud = lp.cloud
          AND u.sku_name = lp.sku_name
          AND u.usage_start_time >= lp.price_start_time
        WHERE
          u.usage_unit = 'DBU'
      )
  )
 GROUP BY ds, sku1, workspace_id
)
select ds, sku, sum(dbus) as dbus, sum(cost_at_list_price) as cost_at_list_price
from agg_data
where
 ds <= now()
 and ds >= SUBSTRING_INDEX('{{start_date}}', ' |', 1)
group by ds, sku
```

How to enable system.query.history

databricks system-schemas list 65cd58cc-8a1a-41df-8a57-f01360651e83

databricks system-schemas enable 65cd58cc-8a1a-41df-8a57-f01360651e83 query

select * from system.query.history

__databricks_internal Catalog:

__databricks_internal is a hidden, system-managed space for DLT pipeline materializations, designed for efficiency and security rather than direct user access or modification.

User Defined Functions

- **Scalar UDFs -** operate on a single row and return a single result for each row.

```
CREATE OR REPLACE FUNCTION capitalize_string(input_str STRING)
RETURNS STRING
RETURN UPPER(input_str);

SELECT capitalize_string("hello") AS capitalized_word;
```

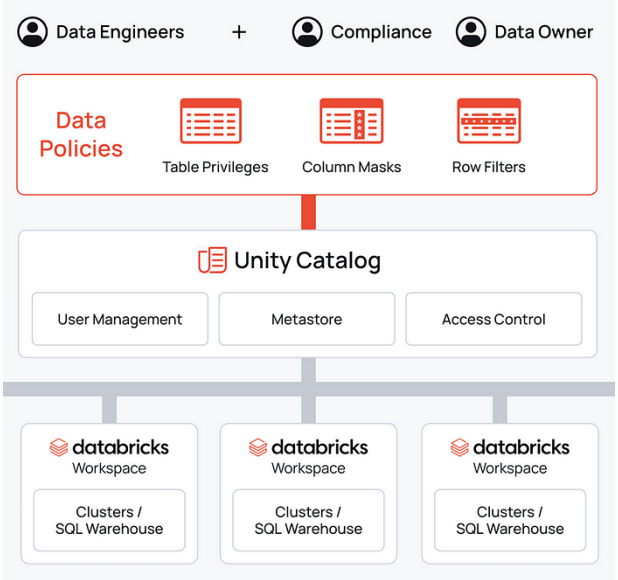
- **SQL UDFs - SQL UDFs are functions written in SQL that extend SQL capabilities**

```
CREATE FUNCTION my_catalog.my_schema.get_name_length(name STRING)
RETURNS INT
RETURN LENGTH(name);

SELECT name, my_catalog.my_schema.get_name_length(name) AS name_length
FROM your_table;
```

- **Python UDTFs (User Defined Table Functions) – Returns a table**

```
CREATE FUNCTION my_udtf()
RETURNS TABLE(my_column VARCHAR) AS $$
  SELECT 'Hello'
  UNION
  SELECT 'World'
$$;


SELECT my_column FROM TABLE(my_udtf());
```

Row level filter in Unity Catalog – Filter sensitive data



Challenges in Data Security:

- Sensitive Data Exposure: Broad access can risk exposing sensitive information.
- Complex Access Controls: Managing detailed access permissions can be intricate.
- Regulatory Compliance: Adhering to regulations requires precise data access control.

Solution: Row-Level Masking in Unity Catalog

```
SELECT * FROM system.information_schema.catalogs
SELECT * FROM system.information_schema.schemata where catalog_name = 'dev'
SELECT * FROM system.information_schema.tables where table_catalog = 'dev'
SELECT * FROM system.information_schema.views
SELECT * FROM system.information_schema.columns

SELECT * FROM system.information_schema.metastore_privileges
SELECT * FROM system.information_schema.catalog_privileges
SELECT * FROM system.information_schema.schema_privileges
SELECT * FROM system.information_schema.table_privileges

SELECT * FROM system.information_schema.volumes
SELECT * FROM system.access.audit
SELECT * FROM system.access.audit

SELECT * FROM system.storage.predictive_optimization_operations_history


WITH agg_data AS (
 SELECT ds, sku1 as sku, workspace_id, SUM(dbus) AS dbus, SUM(cost_at_list_price) AS
cost_at_list_price
 FROM
  (
```

```
    select workspace_id, ds, dbus, cost_at_list_price, sku,
     CASE
       WHEN sku LIKE '%ALL_PURPOSE%' THEN 'ALL_PURPOSE'
       WHEN sku LIKE '%JOBS%' THEN 'JOBS'
       WHEN sku LIKE '%DLT%' THEN 'DLT'
       WHEN sku LIKE '%SQL%' THEN 'SQL'
       WHEN sku LIKE '%INFERENCE%' THEN 'MODEL_INFERENCE'
       ELSE 'OTHER'
     END AS sku1
    from
     (
       SELECT u.workspace_id, u.usage_date AS ds, u.sku_name AS sku,
         CAST(u.usage_quantity AS DOUBLE) AS dbus,
         CAST(lp.pricing.default * usage_quantity AS DOUBLE) AS cost_at_list_price
       FROM
         system.billing.usage u
         INNER JOIN system.billing.list_prices lp ON u.cloud = lp.cloud
         AND u.sku_name = lp.sku_name
         AND u.usage_start_time >= lp.price_start_time
       WHERE
         u.usage_unit = 'DBU'
     )
  )
 GROUP BY ds, sku1, workspace_id
)
select ds, sku, sum(dbus) as dbus, sum(cost_at_list_price) as cost_at_list_price
from
 agg_data
where
 ds <= now()
 and ds >= SUBSTRING_INDEX('{{start_date}}', ' |', 1)
group by ds, sku
```

Earlier Approach:
CREATE OR REPLACE <table_name>_kolkata_view AS Select * from <table> where region='kolkata'

OR

CREATE VIEW <table_name>_kolkata_view
AS SELECT *
FROM <table_name>
WHERE IS_ACCOUNT_GROUP_MEMBER('kolkata');

The is_account_group_member function in Databricks SQL is used to check if a user is a member of a specific account-level group.
It returns true if the user belongs to the group and false otherwise

Example:
SELECT is_account_group_member('admins'); -- Returns false if the user is not a member of the 'admins' group
SELECT is_account_group_member('dev');   -- Returns true if the user is a member of the 'dev' group

In Row Level Security

Step 1: Create a row filter
CREATE FUNCTION <function_name> (<parameter_name> <parameter_type>, ...)
RETURN {filter clause whose output must be a boolean};

Step 2: Apply the row filter to a table

CREATE TABLE <table_name> (…) **WITH ROW FILTER** user_filter ON (Email)

ALTER TABLE <table_name>
SET ROW FILTER <function_name> ON (<column_name>, ...);

TO REMOVE RLS: ALTER TABLE <table_name> DROP ROW FILTER;

Example:

```
CREATE TABLE IF NOT EXISTS supplier_manager_acl (
  group_id STRING,
  users ARRAY<STRING>
);

INSERT OVERWRITE supplier_manager_acl (group_id, users) VALUES
  ('SupplierManager(Packaging)', Array("mcole@hitachisolutions.com", "jonsmith@domain.com")),
  ('SupplierManager(Clothing)',  Array("billybob@domain.com"));


CREATE OR REPLACE FUNCTION supplier_manager_row_filter (CategoryName STRING)
  RETURN is_account_group_member('admins')
  OR is_account_group_member('Executives')
  OR EXISTS(
    SELECT 1 FROM supplier_manager_acl v
    WHERE CONCAT('SupplierManager(',CategoryName,')') = v.group_id AND
ARRAY_CONTAINS(v.users, SESSION_USER()));

/* Apply the UDF Row Filter */
ALTER TABLE gold.dim_supplier SET ROW FILTER supplier_manager_row_filter ON (CategoryName);
```

Column Masking

```
CREATE FUNCTION dev.bronze.phone_mask(phone STRING)
RETURN IF(
   IS_ACCOUNT_GROUP_MEMBER('admin'),
   phone,
   '****'
 );
```

ALTER TABLE dev.bronze.customer **ALTER COLUMN c_phone SET MASK** rsoni.tpch.phone_mask;

ALTER TABLE dev.bronze.customer **ALTER COLUMN c_phone DROP MASK**

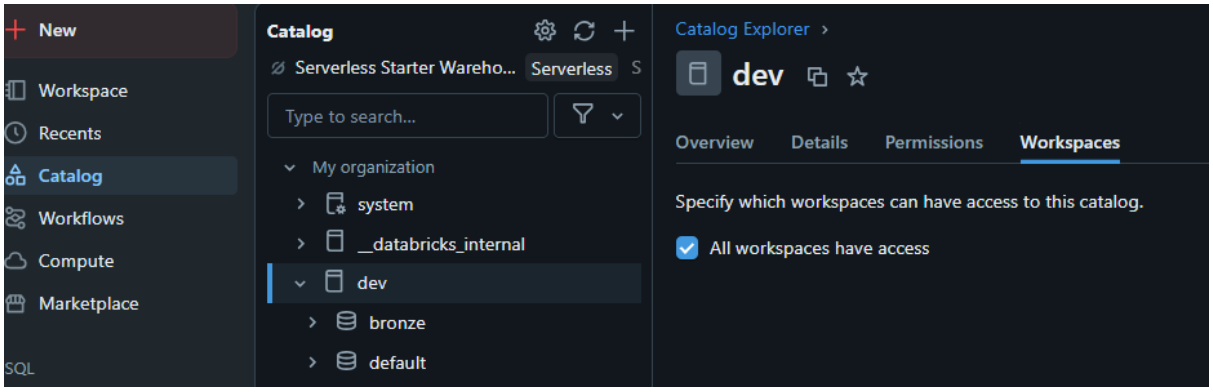
Workspace catalog binding in UC

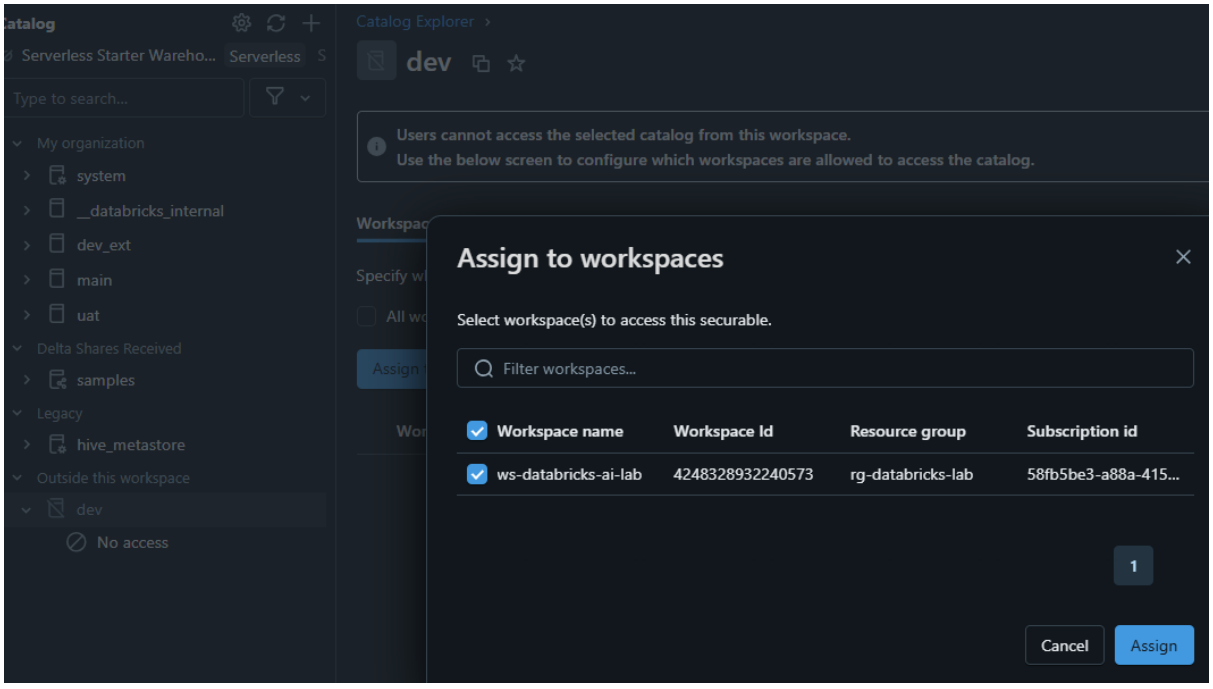
In databricks, we can create multiple workspaces and share a catalog across workspaces.

We can restrict a catalog to specific workspace.

Ex: dev catalogs shouldn't be available in QA workspace.

Do do that, Uncheck "All workspace have access" and next "Assign to specific workspace"



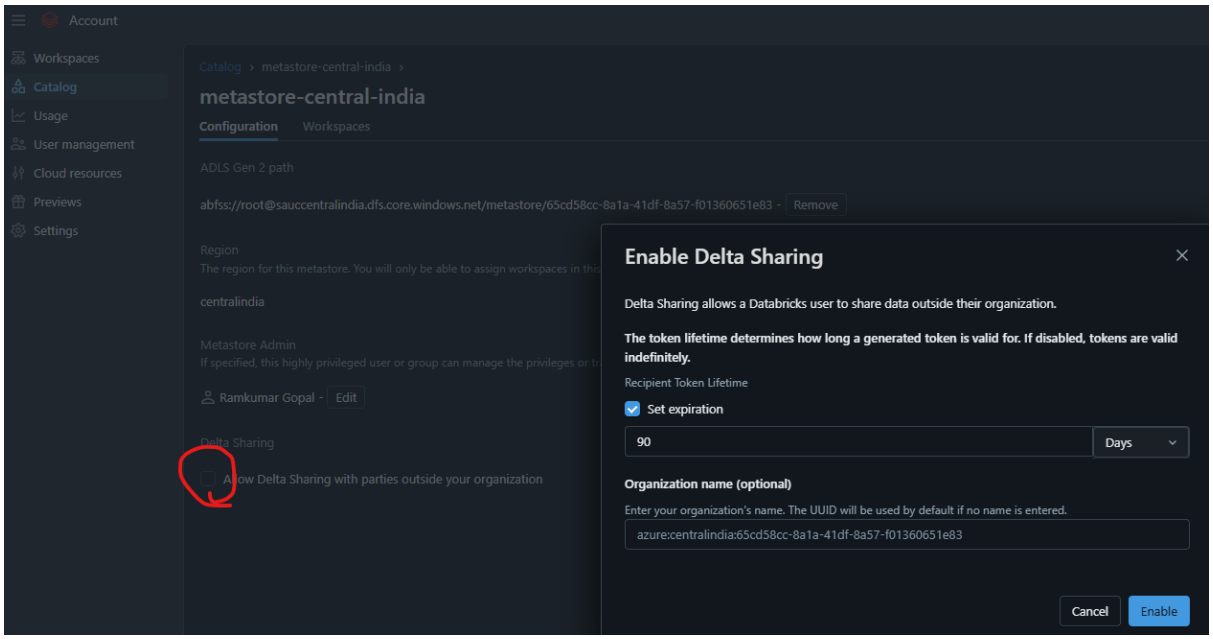


Delta Sharing :

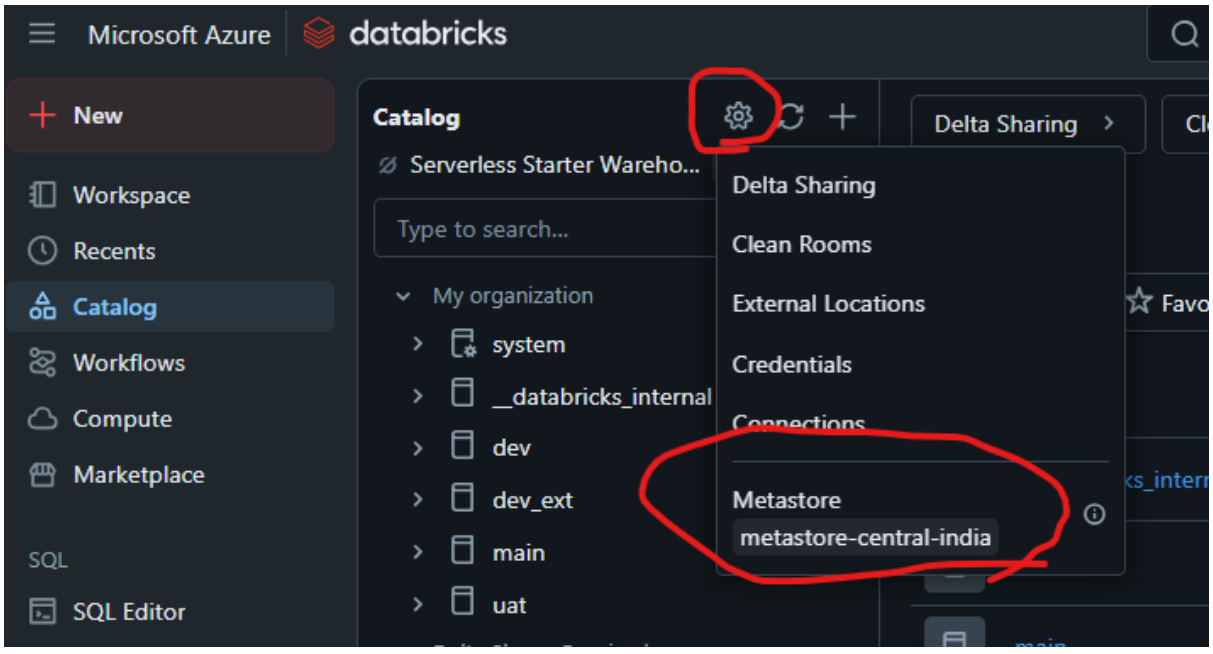
Share data within and outside in read only mode.

- Open sharing – share with external party (doesn't use Databricks)
- Databricks to Databricks

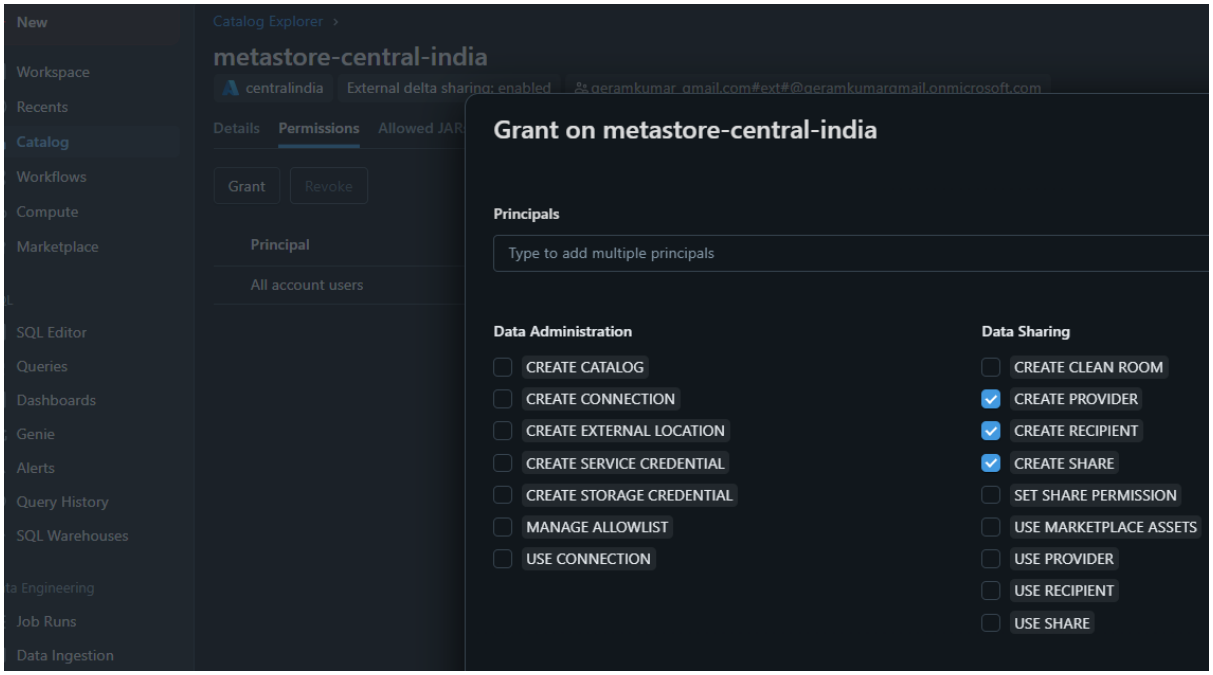
Step 1 : Enable "Allow data sharing outside your organization" in Accounts console
Note: this step is not required if we share delta within the same organization to other metastore



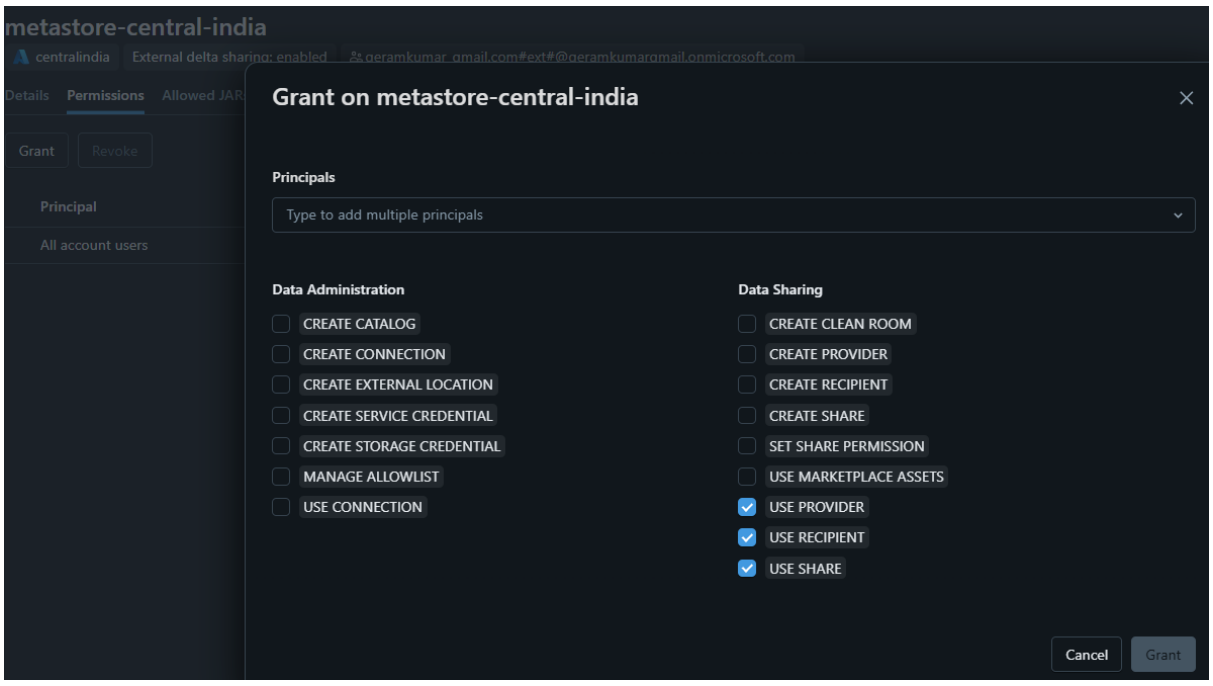
Step 2: In Workspace, select the metastore -> permission



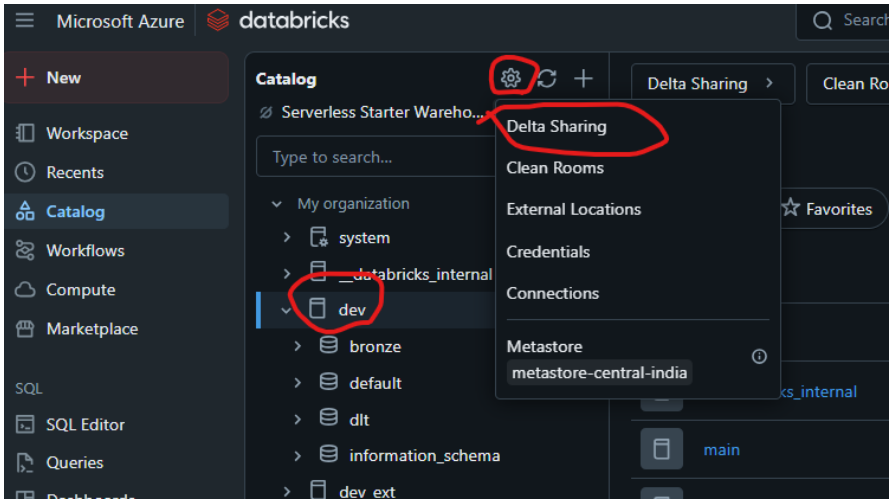
To Provide Sharing: Step 3 Provide Create Provider, Create Recepient, Create share permission

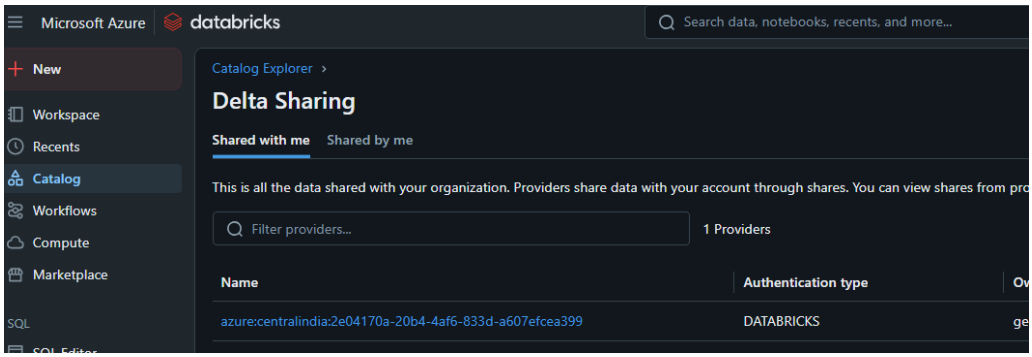


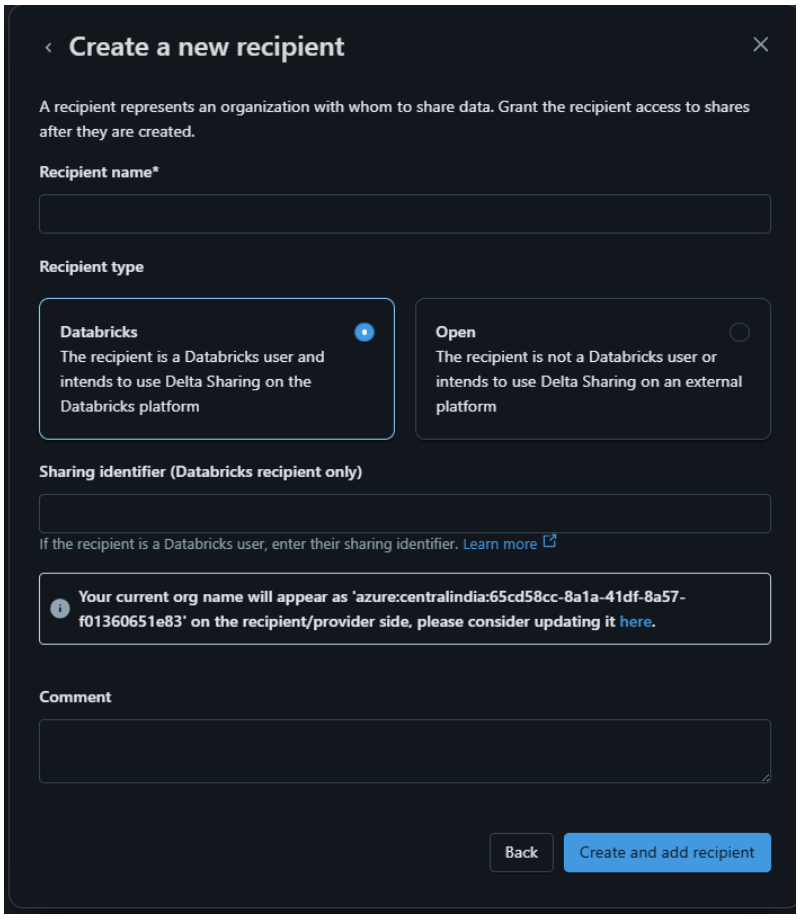
Note: To use delta sharing we need below 3 permissions



Step 4: select the catalog -> select delta sharing

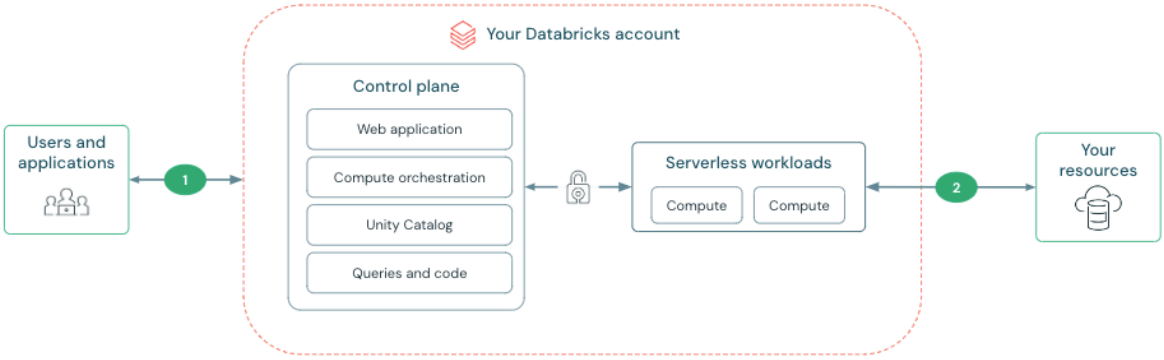

Microsoft Azure
databricks
New
Workspace
Recents
Catalog
Workflows
Compute
Marketplace
SQL
SQL Editor
Queries
Catalog
Serverless Starter Wareho...
Type to search...
My organization
system
dev
bronze
default
dlt
information_schema
dev_ext
Delta Sharing
Clean Rooms
External Locations
Credentials
Connections
Metastore
metastore-central-india
Delta Sharing
Clean Ro
Favorites
main


Microsoft Azure
databricks
Search data, notebooks, recents, and more...
New
Workspace
Recents
Catalog
Workflows
Compute
Marketplace
SQL
Catalog Explorer
Delta Sharing
Shared with me
Shared by me
This is all the data shared with your organization. Providers share data with your account through shares. You can view shares from pro
Filter providers...
1 Providers
Name
Authentication type
azure:centralindia:2e04170a-20b4-4af6-833d-a607efcea399
DATABRICKS


Create a new recipient
A recipient represents an organization with whom to share data. Grant the recipient access to shares after they are created.
Recipient name*
Recipient type
Databricks
The recipient is a Databricks user and intends to use Delta Sharing on the Databricks platform
Open
The recipient is not a Databricks user or intends to use Delta Sharing on an external platform
Sharing identifier (Databricks recipient only)
If the recipient is a Databricks user, enter their sharing identifier. Learn more
Your current org name will appear as 'azure:centralindia:65cd58cc-8a1a-41df-8a57-f01360651e83' on the recipient/provider side, please consider updating it here.
Comment
Back
Create and add recipient

Serverless Compute in Databricks:

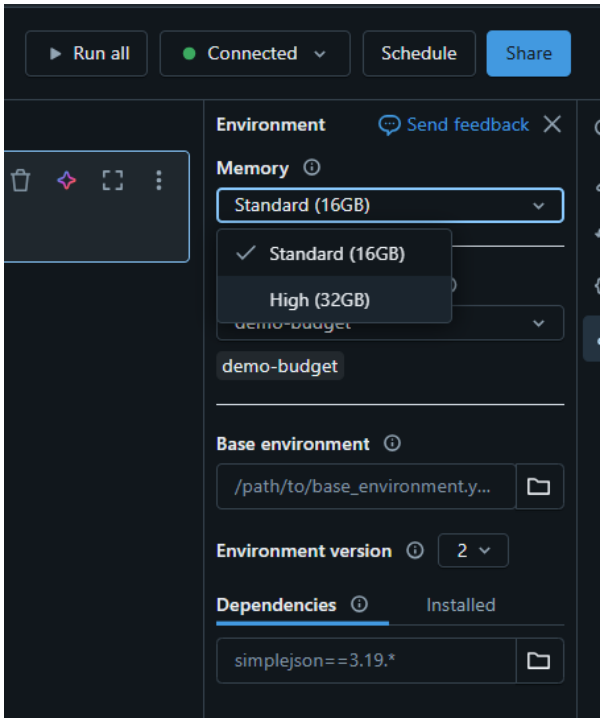
- Guaranteed isolation via NW isolation. Will be auto scaled based on workload.
- Databricks will take care of Compute cluster. Both compute and runtime will be billed by Databricks.
- Cluster will reside in Control plane in your databricks account
- Serverless is available for some regions. WIP.
- Need to be enabled.



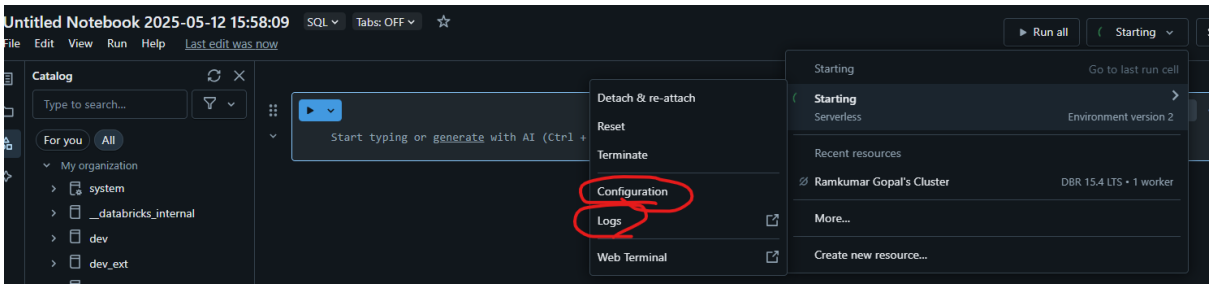
Enable Serverless compute:

Under accounts -> settings -> Enable “Serverless compute for Notebooks and workbooks”

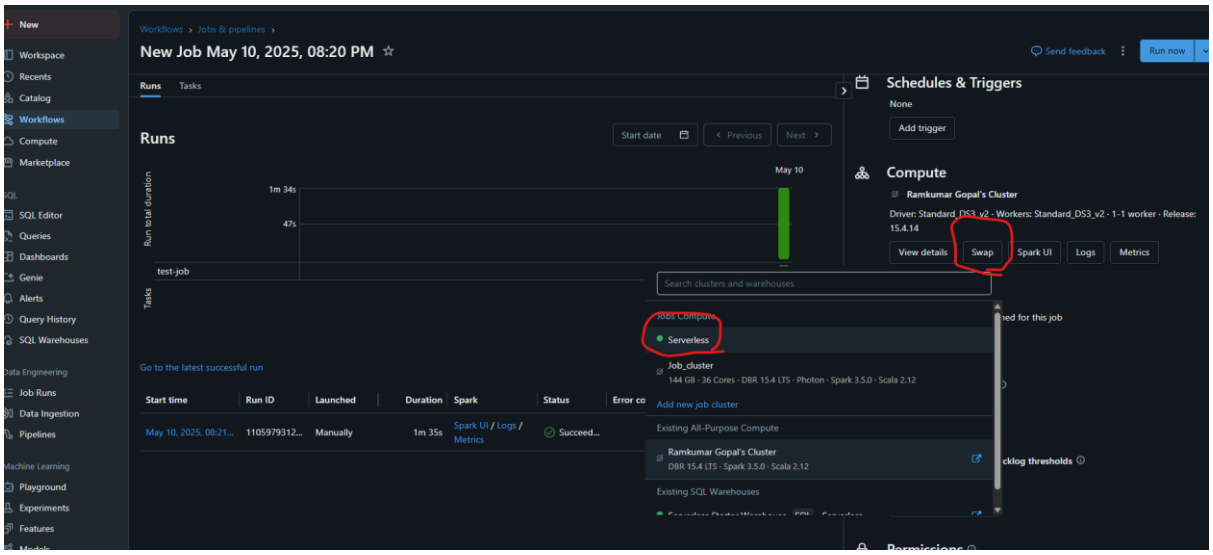
Serverless compute (Memory Standard 16 GB, High 32 GB):



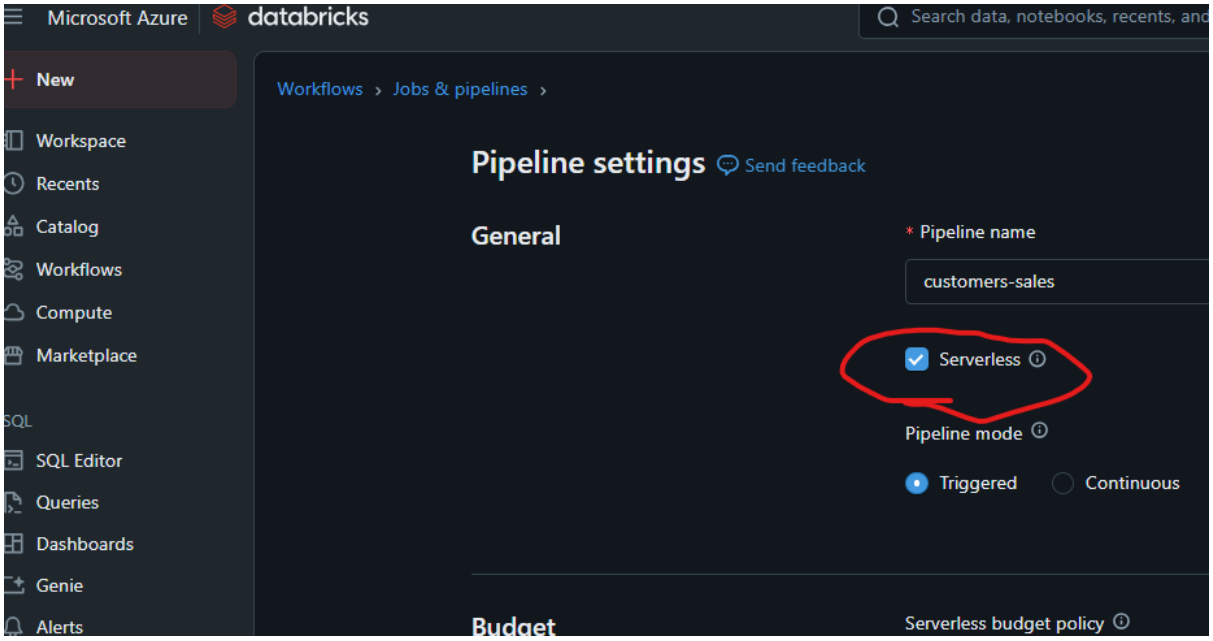
Serverless configurations and Logs:



Use Serverless in Jobs:



Serverless Compute in DTL pipelines



Parameters across databricks:

select * from sales where product :=pproduct and amount :>pcountry

this will add a text boxes on top of notebook for passing the value

introduced in 15.4 Databricks runtime.

The **named parameter marker syntax** can be used by simply adding a colon (:)

SQL Warehouse and DBSQL

Azure Databricks Standard vs Premium Tier – Key Feature Comparison

Category	Feature	Standard Tier	Premium Tier
Security & Governance	Table Access Control (Table ACLs)	❌ Not supported	✅ Supported – Fine-grained access on tables
	Unity Catalog Support	❌ Limited/Not supported	✅ Full support – central metadata & RBAC across workspaces
	Cluster Policies	❌ Not available	✅ Enforced for standardizing cluster configurations
Monitoring & Auditing	Audit Logs (Diagnostic Settings)	❌ Basic (limited diagnostic logs)	✅ Full – via Azure Monitor, Log Analytics, Event Hub
Data Access & Controls	Notebook Permissions	❌ No notebook-level permissions	✅ Fine-grained notebook access control
	Repos Access Control	❌ Not supported	✅ Git-integrated with permissions
	Databricks SQL Permissions	❌ Not available	✅ Row-level, column-level security via Unity Catalog
Compliance & Enterprise Readiness	HIPAA, SOC, ISO Compliance	✅ Partial	✅ Full – meets enterprise compliance needs

Delta Live Tables (DLT) Features

Feature	DLT Core	DLT Pro	DLT Advanced
Basic Capabilities	✓	✓	✓
Change Data Capture		✓	✓
Data Quality			✓

Pricing:

Delta Live Tables (DLT)

Workload	DLT Core	DLT Pro	DLT Advanced
Delta Live Tables (DLT)*	$0.30/DBU-hour	$0.38/DBU-hour	$0.54/DBU-hour

Workload & Pricing:

Workload	Standard Tier Prices	Premium Tier Prices
Interactive Serverless Compute*****	-	**$0.70**/DBU-hour**
All-Purpose Compute***	**$0.40**/DBU-hour	**$0.55**/DBU-hour
Automated Serverless Compute*****	-	**$0.24**/DBU-hour****
Jobs Compute***	**$0.15**/DBU-hour	**$0.30**/DBU-hour
Jobs Light Compute	**$0.07**/DBU-hour	**$0.22**/DBU-hour
SQL Compute	-	**$0.22**/DBU-hour
SQL Pro Compute	-	**$0.55**/DBU-hour
Serverless SQL*****	-	**$0.70**/DBU-hour

DBU Pre-purchase & Discounts:

1-year pre-purchase plan

Databricks commit unit (DBCU)	Price (with discount)	Discount
12,500	$12,000	4%
25,000	$23,500	6%
50,000	$46,000	8%
100,000	$89,000	11%
200,000	$172,000	14%
350,000	$287,000	18%
500,000	$400,000	20%
750,000	$577,500	23%
1,000,000	$730,000	27%
1,500,000	$1,050,000	30%
2,000,000	$1,340,000	33%

Azure VM cost depends on:

- Enterprise level agreements
- Region
- OS (Windows, Linux)
- VM Size and series
- Type of storage (HDD, SSD – standard/premium/ultra
- Azure Hybrid benefit
- Pricing Model
    - Pay as you go
    - Reserved instance
    - Spot Instances

Commonly used Azure VM in Databricks – General Purpose

standard_d16ads_v5

- General purpose VM – AMD based, Temp disk, Premium storage

Ddsv5 series

Instance	vCPU(s)	RAM	DBU Count	DBU Price	Pay As You Go Total Price	1 Year Savings Plan (% Savings) Total Price	3 Year Savings Plan (% Savings) Total Price	Spot (% Savings) Total Price
D4ds v5	4	16.00 GiB	1.00	$0.55/hour	$0.816/hour	$0.734/hour ~10% savings	$0.676/hour ~17% savings	$0.603/hour ~26% savings
D8ds v5	8	32.00 GiB	2.00	$1.10/hour	$1.632/hour	$1.468/hour ~10% savings	$1.351/hour ~17% savings	$1.206/hour ~26% savings
D16ds v5	16	64.00 GiB	4.00	$2.20/hour	$3.264/hour	$2.935/hour ~10% savings	$2.701/hour ~17% savings	$2.411/hour ~26% savings
D32ds v5	32	128.00 GiB	8.00	$4.40/hour	$6.528/hour	$5.869/hour ~10% savings	$5.401/hour ~17% savings	$4.821/hour ~26% savings
D48ds v5	48	192.00 GiB	12.00	$6.60/hour	$9.792/hour	$8.803/hour ~10% savings	$8.101/hour ~17% savings	$7.231/hour ~26% savings
D64ds v5	64	256.00 GiB	16.00	$8.80/hour	$13.056/hour	$11.737/hour ~10% savings	$10.801/hour ~17% savings	$9.641/hour ~26% savings

Comprehensive Guide to Optimize Databricks, Spark and Delta Lake Workloads

Databricks Performance Benefits

1. Photon Engine (for SQL & Python Workloads)

Databricks' Photon is a vectorized query engine written in C++, delivering 2x–20x faster performance for SQL and Python workloads compared to the traditional Spark engine.

2. Databricks Runtime Optimizations

Databricks Runtime includes **optimized libraries**, improved **Spark scheduling**, and execution enhancements that outperform open-source Apache Spark.

3. **Autoscaling Clusters (Vertical & Horizontal Scaling)**

4. Intelligent Caching (Databricks IO Cache)

5. Adaptive Query Execution (AQE)

6. Broadcast Hash Joins for Small Tables

7. File Compaction and Partition Pruning

df.write.partitionBy("year", "month").parquet("/mnt/data/sales_partitioned")

df = spark.read.parquet("/mnt/data/sales_partitioned")

df.filter("year = 2023 AND month = 1").count()

8. Cluster Warm Pools (Cluster Reuse)

Databricks provides **job clusters** with pre-warmed pools, reducing cluster start-up time and boosting job SLAs.

Delta Lake Performance Tuning (Mostly in-built. No action required)

 Data Skipping via Statistics: Faster queries by skipping irrelevant data files using metadata.

-- Delta Lake automatically leverages statistics for data skipping

SELECT * FROM delta_table WHERE date = '2025-05-13';

 File Compaction: Improved read speed by **reducing small file** overhead.

OPTIMIZE delta_table

 Z-Ordering: Enhanced filtering performance through **data co-location**.

OPTIMIZE delta_table ZORDER BY (user_id, event_timestamp)

 Delta Cache: Blazing-fast **reads from local disk** for frequent data (**automatic**).

-- Delta Cache is automatically utilized by the Databricks runtime

SELECT count(*) FROM delta_table WHERE country = 'USA'; -- Subsequent runs faster

 Transactional Writes: Predictable ingestion, avoiding partial/corrupted data.

-- Transactions ensure atomicity of write operations

INSERT INTO delta_table VALUES (...);

DELETE FROM delta_table WHERE session_id = '...';

 Bloom Filters: Further data skipping for high-cardinality columns (configuration dependent).

-- Enabling Bloom filters (implementation details may vary by Databricks runtime)

ALTER TABLE delta_table SET TBLPROPERTIES ('delta.enableBloomFilter' = 'true',
'delta.bloomFilter.columns' = 'user_id');

 Faster Schema Evolution: Avoids costly rewrites during schema changes.

ALTER TABLE delta_table ADD COLUMNS (new_column STRING);

 Deletion Vector Optimized Updates/Deletes: Efficient modification of specific records.

UPDATE delta_table SET status = 'inactive' WHERE last_activity < '2025-05-01';

DELETE FROM delta_table WHERE age < 18;

 Time Travel: Fast historical queries by accessing relevant past data.

SELECT * FROM delta_table VERSION AS OF 1;

SELECT * FROM delta_table AT TIMESTAMP AS OF '2025-05-10 10:00:00';

Public Cloud – Common bad practices and recommendations

Bad practices:

- Procure and forget
- Do not review the utilization and bill
- Lack of knowledge on the choices and cost implications
- Oversizing and under-utilization
- Forget to setup budgets and alarm
- Not spending time for reviews, resource clean-ups, right-sizing

Recommendations:

- Shutdown unused resources
- Right-size underutilized resources
- Enable auto-scaling
- Terminate after x minutes
- Choose the right instance type and size
    - Reserved, PAY, Spot
- Choose PAAS over IAAS
- Tag your Azure resources
- Set budgets/alerts
- Regularly review the resources and utilizations

Azure Blob Storage vs ADLS Gen2

Feature	Azure Blob Storage	Azure Data Lake Storage Gen2 (ADLS Gen2)
Purpose	General-purpose object storage for various storage scenarios.	Optimized for big data analytics workloads, including machine learning, data warehousing, and data lake architectures.
Namespace	Flat namespace (containers and blobs)	Hierarchical namespace (directories and files) - like a file system
File System	No built-in file system semantics	Hierarchical namespace, file system semantics
Analytics	Good for storing data for analysis, but not optimized for analytical workloads.	Optimized for big data analytics, offering better performance and cost-effectiveness in analytics scenarios.
Security	Access control through containers, shared access signatures (SAS).	Role-based access control (RBAC), Azure Active Directory (AAD) authentication, POSIX permissions, ACLs.
Hadoop Compatibility	Not natively Hadoop compatible.	Hadoop-compatible storage (supports Apache Hadoop ecosystem, including Hive and Spark).
Performance	Good storage retrieval performance, but may be slower for large-scale analytics.	Better storage retrieval performance, especially for analytical workloads, due to hierarchical namespace.
Cost	Potentially higher cost for analytical workloads due to less optimized architecture.	Lower cost for analytical workloads due to optimized architecture and features.
Soft Delete	Soft delete feature available.	No soft delete feature.
Hierarchical Organization	Data organized into containers and blobs, no inherent hierarchical structure.	Data organized into directories and files, allowing for better organization and management of large datasets.
Use Cases	General-purpose data storage, media storage, web hosting, backups, etc.	Big data analytics, machine learning, data warehousing, data lake architectures.

ADLS Redundancy Types

Redundancy Type	Description
Locally Redundant Storage (LRS)	Stores three copies of your data within a single physical location (data center). Suitable for applications needing high availability within a region but not for regional disasters.
Zone-Redundant Storage (ZRS)	Replicates your data synchronously across three separate availability zones within the same region. Each zone has independent power, cooling, and networking, offering protection against zonal failures.
Geo-Redundant Storage (GRS)	Stores three copies of your data in the primary region and replicates it to a secondary region (geographically distant). This protects against regional outages and disasters.
Geo-Zone-Redundant Storage (GZRS)	Combines the benefits of ZRS and GRS. It replicates your data within three zones in the primary region and also asynchronously replicates it to a secondary region.

Azure Data Services for Azure Data Architect

Azure Data Lake Storage Gen2 (ADLS Gen2)

1. Features

Feature	Description
Hierarchical Namespace	Organize data in directories for efficient data management.
Security & Access Control	Supports Azure AD integration and fine-grained ACLs.
Scalability	Scalable to handle large volumes of structured/unstructured data.
Delta Lake Integration	Supports Delta Lake for ACID transactions, data versioning, and schema enforcement.
Cost-Effective Storage	Offers Hot, Cool, and Archive tiers for cost-efficient data storage.
High Availability	Use RA-GRS replication for geo-redundant storage.

2. Use Cases

Use Case	Description
Big Data Analytics	Store massive data for Spark, Hadoop workloads, and analytics.
Machine Learning	Ideal for storing datasets for ML model training and experimentation.
Data Warehousing	Acts as raw and staging storage for data warehousing in Azure Synapse.
IoT Data Storage	Stores real-time sensor data or logs for IoT applications.
Data Lakehouse	Use Delta Lake for combining data lake flexibility with data warehouse features.
Archiving	Store infrequently accessed data in Cool and Archive tiers for long-term cost-effective storage.

3. Approximate Cost per Hour

Cost Component	Approximate Cost
Hot Tier	$0.0184 per GB/month (varies by region)
Cool Tier	$0.01 per GB/month
Archive Tier	$0.002 per GB/month (cheaper but retrieval costs higher)

Cost Component	Approximate Cost
Transaction Costs	$0.0004 per 10,000 operations (e.g., PUT, GET, LIST requests)
Retrieval Costs	$0.02 per GB (for Cool and Archive tiers)
Replication Costs	RA-GRS adds extra cost based on replication region and data volume.

4. Service Offerings

Service Offering	Description
ADLS Gen2	Combines Azure Blob Storage with enhanced features for big data analytics.
Hot Tier	For frequently accessed data; optimal for real-time analytics and processing.
Cool Tier	For infrequent access data; lower cost for long-term data storage.
Archive Tier	Cheapest option for infrequently accessed data; higher retrieval latency and costs.
RA-GRS (Geo-Redundant Storage)	Provides geo-replication across regions for disaster recovery and high availability.
ZRS (Zone-Redundant Storage)	Provides multi-zone replication within a region for high availability and fault tolerance.
Delta Lake	ACID transaction support, schema enforcement, and data versioning for analytics and machine learning.

5. Recommendations

Recommendation	Description
Use ADLS Gen2 for Big Data Workloads	Best suited for storing large datasets for Hadoop, Spark, and other analytics frameworks.
Leverage Delta Lake for ACID Transactions	Use Delta Lake for managing large-scale data with reliability and consistency.
Store Active Data in Hot Tier	Store frequently accessed data in the Hot Tier for low-latency access.
Archive Infrequently Accessed Data	Use Cool and Archive tiers to store cold data at a lower cost.
Enable RA-GRS for High Availability	Use RA-GRS for critical workloads that need geo-redundancy and disaster recovery.

Recommendation	Description
Design Efficient Partition Keys	Plan partitioning strategy to improve query and processing performance for large datasets.

📌 **Key Considerations for Azure Data Architect:**

- Delta Lake on ADLS Gen2: Delta Lake enhances ADLS Gen2 by adding ACID transactions, schema evolution, and time travel for better data management.
- Scalability: ADLS Gen2 is designed to scale to exabytes of data, making it a robust choice for big data analytics, ML, and IoT workloads.
- Cost Optimization: Use Cool and Archive tiers for long-term storage while ensuring Hot Tier is used only for frequently accessed data.
- Security: Leverage Azure Active Directory (AAD) integration and Access Control Lists (ACLs) for fine-grained data access management.

can you provide 50 databricks tips which only databricks employees know? one example is DBU billing starts when the sparkcontext is available. each not more than 15 words

- DBU billing starts when SparkContext is available, not notebook execution start.
- Jobs clusters can silently reuse terminated instances if configured properly.
- %run does not carry over all environment variables to the child notebook.
- Autoloader doesn't support schema evolution with nested columns yet.
- Photon engine doesn't support all legacy Spark operations or Python UDFs.
- Delta Lake time travel can significantly impact performance if used heavily.
- Unity Catalog policies can override workspace ACLs silently.
- Autoloader uses checkpoint to avoid reprocessing files even after schema changes.
- DBFS is not recommended for production due to potential mount consistency issues.
- Delta cache is local to the cluster and not shared across all clusters.
- Job cluster startup time improves with pinned runtime versions.
- DBU cost differs based on cluster mode, even for same node type.
- UDFs in SQL can cause Photon to fall back to slower execution paths.
- Workspace UI changes often roll out without clear version updates.
- External locations in Unity Catalog can fail silently if IAM isn’t configured correctly.
- Autoloader with trigger once + cloudFiles.backfillInterval can miss files if misconfigured.
- Repos sync only with default branch; other branches need manual checkout.
- Interactive cluster logs are not retained unless explicitly exported.
- Repos don't support Git LFS or submodules.
- Delta Merge operations are heavy and can bloat transaction logs fast.
- Enable GANG scheduling for optimal Spark + GPU workloads.
- Notebooks executed via Jobs UI have different permission scopes than interactive notebooks.
- Photon accelerates only SQL and Delta operations, not general Spark tasks.
- Serverless compute might take longer to start due to provisioning delays.
- DBFS root is shared across users but not truly collaborative.
- Unity Catalog doesn't yet support streaming Autoloader workflows directly.
- Default cluster policies can silently cap node limits or types.
- Each Job run creates a unique cluster unless configured as shared.

 Notebook widgets don’t persist across Jobs unless redefined each run.

 Delta table vacuum requires table to be closed on all clusters.

 Cluster termination may take minutes even after idle timeout triggers.

 Unity Catalog audit logs aren’t fully visible without enabling diagnostic settings.

 MLflow experiments created via notebooks default to the user’s workspace.

 Cluster pinning prevents automatic runtime upgrades, which can block fixes.

 DBFS mounts may become stale if backing storage credentials expire.

 Managed tables move to workspace storage, not cloud storage, unless external.

 SQL Warehouses don’t support Hive metastore; only Unity Catalog or external.

 Autoloader can skip files with malformed names or extensions silently.

 Global Init Scripts can affect all clusters, including SQL Warehouses.

 Notebooks store execution history locally unless exported manually.

 Serverless SQL warehouses auto-scale within limits defined in SQL endpoint config.

 JDBC connections to Unity Catalog require token scope alignment.

 Overuse of .collect() can silently crash clusters without logs.

 Delta file compaction doesn’t run unless explicitly scheduled.

 Spark UI shows physical plan, but Photon uses a separate optimizer.

 Autoloader logs ingestion metadata to checkpoint path, not to Delta log.

 Table ACLs are not enforced without Unity Catalog being enabled.

 SQL dashboard refreshes may time out without alerts.

 Workspace backups are not automatic—export notebooks to persist.

 Jobs API v2 has different behavior for timeout than v1.

thanks. provide similar 20 tips on databricks billing which only databricks employees know. each not more than 25 words

 Billing starts when SparkContext is initialized, not when the first cell is run.

 DBUs accrue during cluster startup and shutdown, not just during active notebook execution.

 Photon DBUs are cheaper but only apply when Photon is actually used in the workload.

 Job clusters accrue DBUs even if your job fails early in execution.

 SQL Warehouses bill per second, clusters bill per minute—important for short-lived workloads.

 Interactive clusters continue billing even when idle, unless an auto-termination policy is enforced.

 Serverless SQL has a higher DBU rate, but no infrastructure management overhead.

 Streaming jobs continue billing as long as cluster is running, regardless of trigger interval.

 Stopping a job in UI doesn’t immediately stop billing, due to graceful cluster shutdown time.

 Free tier usage is capped by workspace, not by user, which can cause surprise billing.

 Unity Catalog billing includes audit logging overhead, especially if diagnostic settings are enabled.

 DBU cost differs by cloud provider, even for the same instance size.

 Use cluster pools to reduce startup time, indirectly reducing billed time for jobs.

 Auto-scaling delays can increase DBU cost if jobs wait long for resources.

 Pinning runtimes avoids unplanned upgrades, which can introduce billing inefficiencies or failures.

 Running notebooks via dbutils.notebook.run() doesn’t reduce DBU cost—child notebooks still run fully.

 Jobs with retries can double or triple DBU usage if failures aren’t managed.

 Notebooks left open don’t accrue DBUs unless the cluster is running.

 Cluster policies can enforce lower DBU configurations, saving costs without user intervention.

 Using Spot instances can lower infra cost, but DBU cost stays constant.