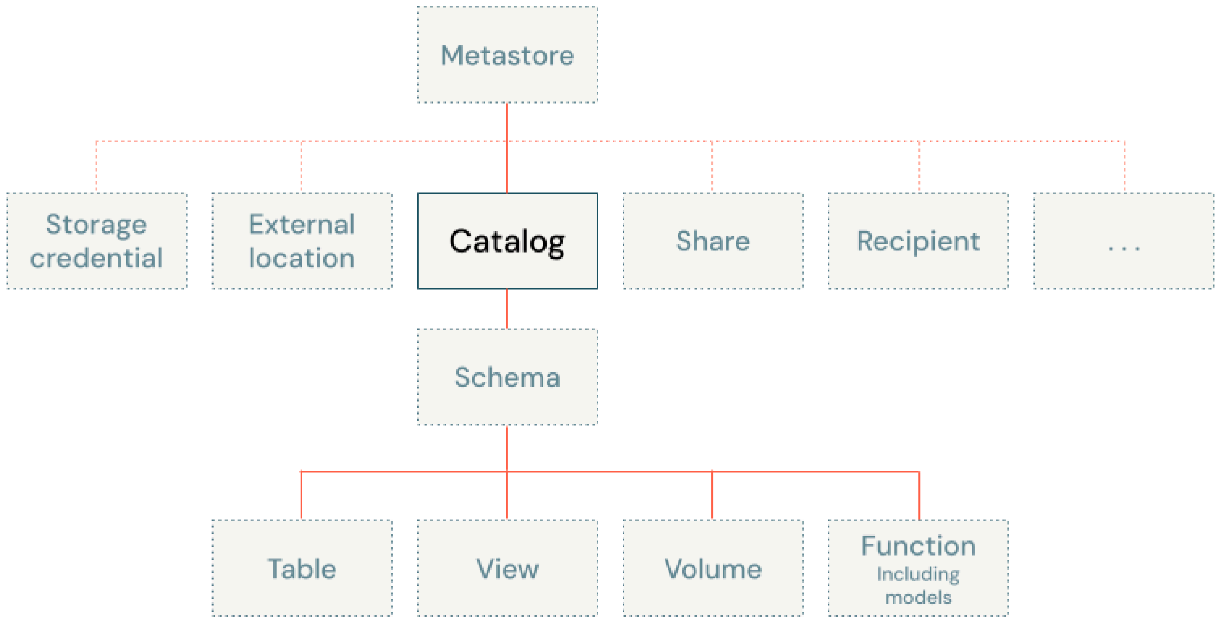
Week 1: Introduction to Databricks & Workspace Fundamentals

What is Databricks?

Databricks is a unified, open analytics platform for building, deploying, sharing, and maintaining enterprise-grade data, analytics, and AI solutions at scale. The Databricks Data Intelligence Platform integrates with cloud storage and security in your cloud account, and manages and deploys cloud infrastructure for you.

What are catalogs in Databricks?

A catalog is the primary unit of data organization in the Databricks Unity Catalog data governance model. This article gives an overview of catalogs in Unity Catalog and how best to use them.
Catalogs are the first layer in Unity Catalog's three-level namespace (catalog.schema.table-etc). They contain schemas, which in turn can contain tables, views, volumes, models, and functions. Catalogs are registered in a Unity Catalog metastore in your Databricks account.



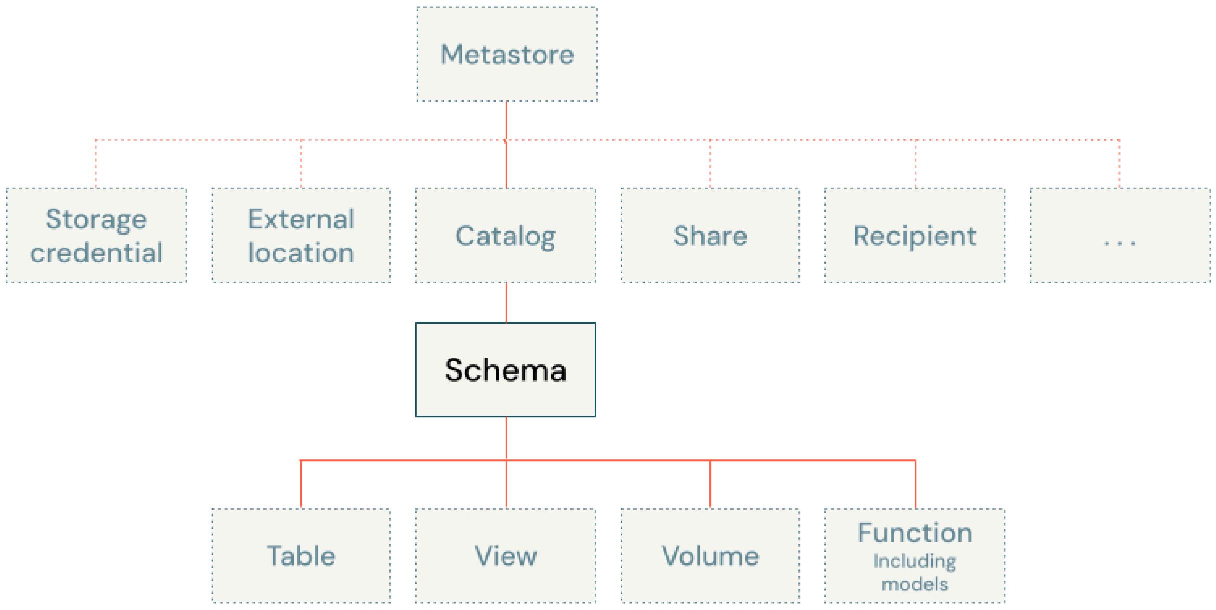
Catalog types

When you create a catalog, you're given two options:

Standard catalog: the typical catalog, used as the primary unit to organize your data objects in Unity Catalog. This is the catalog type that is discussed in this article.
Foreign catalog: a Unity Catalog object that is used only in Lakehouse Federation scenarios. A foreign catalog mirrors a database in an external data system, enabling you to perform read-only queries on that data system in your Databricks workspace.

What is a schema?

In Unity Catalog, a schema is the second level of Unity Catalog's three-level namespace (catalog.schema.table-etc).
A schema organizes data and AI assets into logical categories that are more granular than catalogs. Typically a schema represents a single use case, project, or team sandbox. Regardless of category type, schemas are a useful tool for managing data access control and improving data discoverability.



What are volumes?

Volumes are Unity Catalog objects that enable governance over non-tabular datasets. Volumes represent a logical volume of storage in a cloud object storage location. Volumes provide capabilities for accessing, storing, governing, and organizing files.

While tables provide governance over tabular datasets, volumes add governance over non-tabular datasets. You can use volumes to store and access files in any format, including structured, semi-structured, and unstructured data.
You cannot register files in volumes as tables in Unity Catalog. Volumes are intended for path-based data access only. Use tables when you want to work with tabular data in Unity Catalog.

What is a managed volume?

A managed volume is a Unity Catalog-governed storage volume created within the managed storage location of the containing schema.

What is an external volume?

An external volume is a Unity Catalog-governed storage volume registered against a directory within an external location using Unity Catalog-governed storage credentials.Unity Catalog does not manage the lifecycle and layout of the files in external volumes. When you drop an external volume, Unity Catalog does not delete the underlying data.

The path to access files in volumes uses the following format:
/Volumes/<catalog>/<schema>/<volume>/<path>/<file-name>

What are tables in Databricks?

In Databricks, a table is a structured collection of data stored within a schema. Tables are used to store, query, and manage data using SQL or Spark. The default table type is a Unity Catalog (UC) managed table, which uses Delta Lake for reliable data storage.
Databricks supports three main table types, each with different ownership and data management characteristics:

1. Managed Table

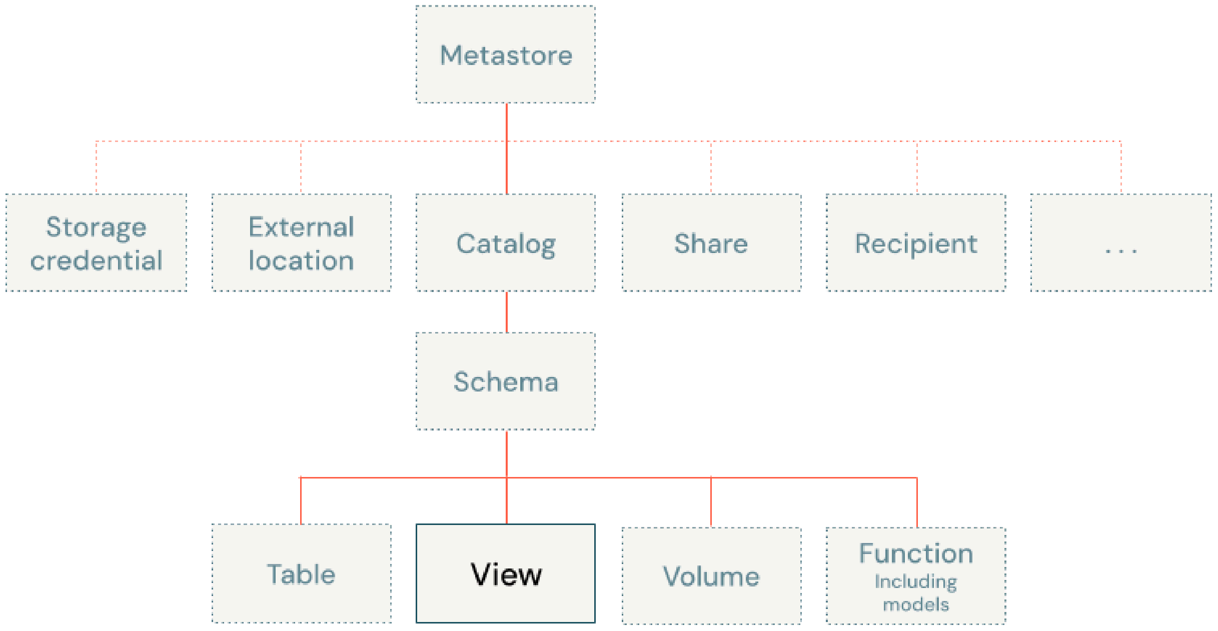
Description: Databricks manages both the metadata and the actual data files.
Storage: Data is stored in a default location within the Databricks-managed storage (e.g., /user/hive/warehouse).
Use Case: Simplifies management when you don't need control over physical storage.
Governance: Works with Unity Catalog for access control and lineage.
Write Support: Yes

2. External Table

Description: Metadata is stored in Databricks, but data files are stored externally (e.g., in Azure Data Lake, S3).
Storage: You specify the storage location.
Use Case: You want to manage data storage yourself or share it across multiple tools.
Governance: Can be used with or without Unity Catalog.
Write Support: Yes
3. Foreign Table
Description: A read-only reference to data stored in an external system (e.g., PostgreSQL, MySQL, Snowflake).
Storage: No data is stored in Databricks; queries are federated to the external system.
Use Case: You want to query external databases without ingesting data.
Governance: Controlled by the external system.
Write Support: No

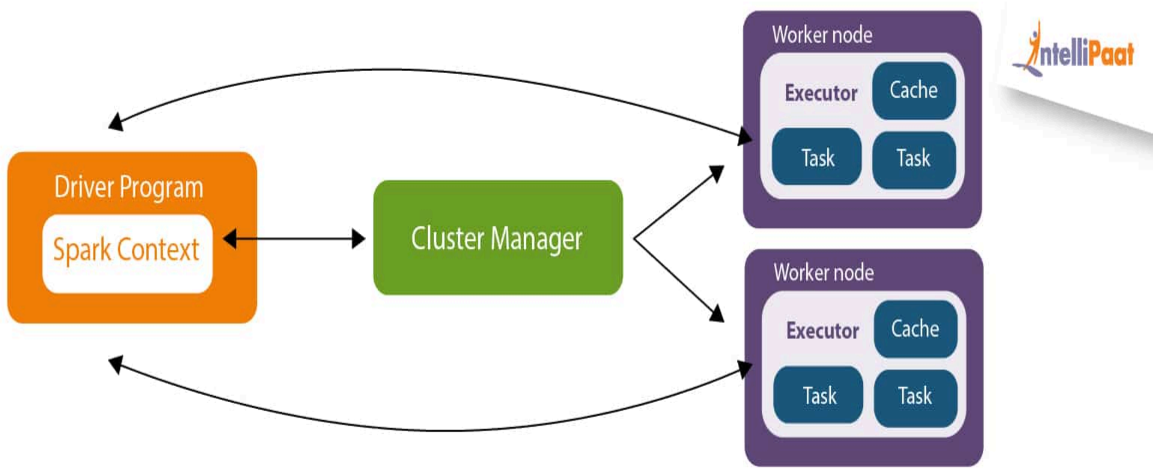
What is a view?

A view is a read-only object that is the result of a query over one or more tables and views in a Unity Catalog metastore. You can create a view from tables and from other views in multiple schemas and catalogs.
This article describes the views that you can create in Databricks and provides an explanation of the permissions and compute required to query them.



What is Compute/Cluster?

Databricks compute refers to the selection of computing resources available in the Databricks workspace. Users need access to compute to run data engineering, data science, and data analytics workloads, such as production ETL pipelines, streaming analytics, ad-hoc analytics, and machine learning.
Users can either connect to existing compute or create new compute if they have the proper permissions.
You can view the compute you have access to using the Compute section of the workspace:



Types of compute

These are the types of compute available in Databricks:
Serverless compute for notebooks: On-demand, scalable compute used to execute SQL and Python code in notebooks.
Serverless compute for jobs: On-demand, scalable compute used to run your Lakeflow Jobs without configuring and deploying infrastructure.
All-purpose compute: Provisioned compute used to analyze data in notebooks. You can create, terminate, and restart this compute using the UI, CLI, or REST API.
Jobs compute: Provisioned compute used to run automated jobs. The Databricks job scheduler automatically creates a job compute whenever a job is configured to run on new compute. The compute terminates when the job is complete. You cannot restart a job compute. See Configure compute for jobs.
Instance pools: Compute with idle, ready-to-use instances, used to reduce start and autoscaling times. You can create this compute using the UI, CLI, or REST API.

Serverless SQL warehouses: On-demand elastic compute used to run SQL commands on data objects in the SQL editor or interactive notebooks. You can create SQL warehouses using the UI, CLI, or REST API.
Classic SQL warehouses: Used to run SQL commands on data objects in the SQL editor or interactive notebooks. You can create SQL warehouses using the UI, CLI, or REST API.

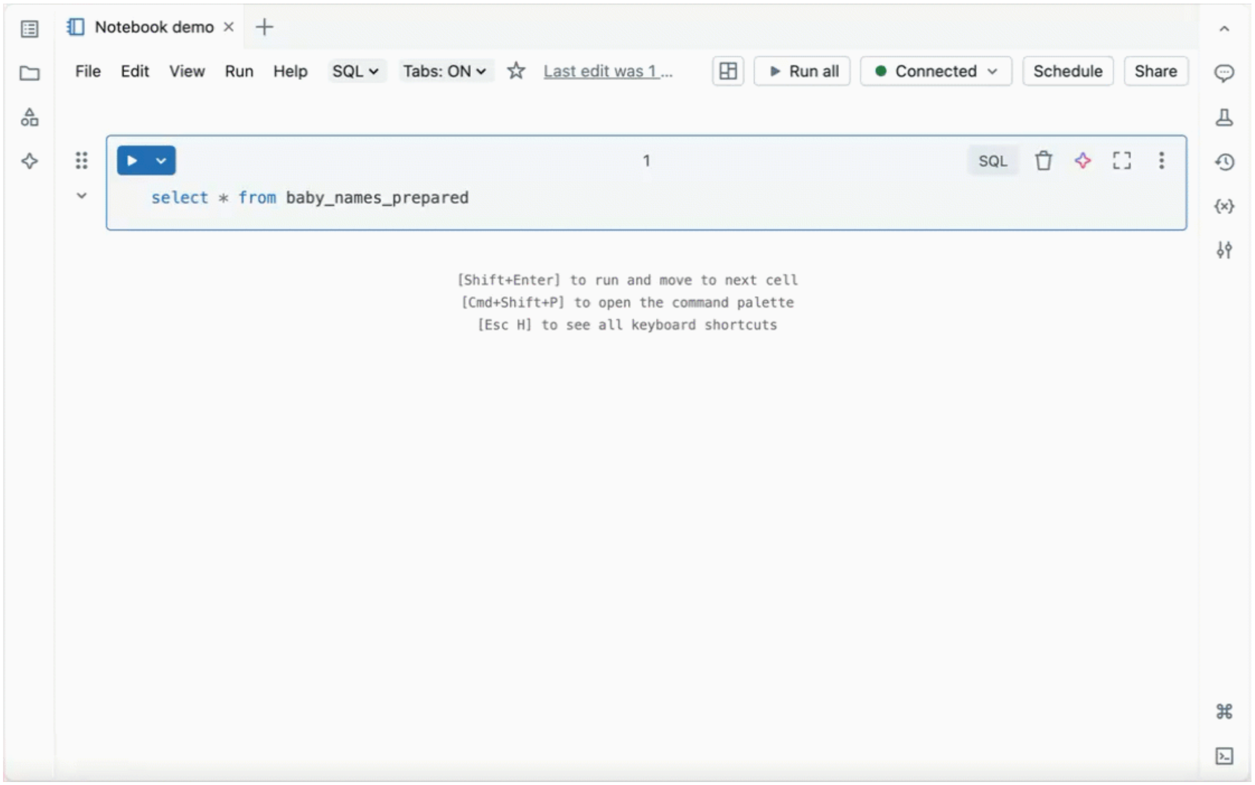
Compute configuration recommendations

This article includes recommendations and best practices related to compute configuration.If your workload is supported, Databricks recommends using serverless compute rather than configuring your own compute resource. Serverless compute is the simplest and most reliable compute option. It requires no configuration, is always available, and scales according to your workload. Serverless compute is available for notebooks, jobs, and Lakeflow Declarative Pipelines.

If you are creating new compute from scratch, Databricks recommends using compute policies. Compute policies let you create preconfigured compute resources designed for specific purposes, such as personal compute, shared compute, power users, and jobs. Policies limit the decisions you need to make when configuring compute settings.

Databricks notebooks:

A Databricks notebook is an interactive environment where you can write and run code in multiple languages — like SQL, Python, Scala, R, or Markdown — for data analysis, machine learning, ETL, and visualization.
Notebooks are the primary tool for creating data science and machine learning workflows on Databricks. Databricks notebooks provide real-time coauthoring in multiple languages, automatic versioning, and built-in data visualizations for developing code and presenting results.



```sql
select * from baby_names_prepared
```

```
[Shift+Enter] to run and move to next cell
[Cmd+Shift+P] to open the command palette
[Esc H] to see all keyboard shortcuts
```

What is Delta Lake in Databricks?

Delta Lake is the optimized storage layer that provides the foundation for tables in a lakehouse on Databricks. Delta Lake is open source software that extends Parquet data files with a file-based transaction log for ACID transactions and scalable metadata handling. Delta Lake is fully compatible with Apache Spark APIs, and was developed for tight integration with Structured Streaming, allowing you to easily use a single copy of data for both batch and streaming operations and providing incremental processing at scale.

Delta Lake is the default format for all operations on Databricks. Unless otherwise specified, all tables on Databricks are Delta tables. Databricks originally developed the Delta Lake protocol and continues to actively contribute to the open source project. Many of the optimizations and products in the Databricks platform build upon the guarantees provided by Apache Spark and Delta Lake. For information on optimizations on Databricks.

ingesting data to Delta Lake

Databricks provides a number of products to accelerate and simplify loading data to your lakehouse.

Upload files to Databricks

This article details patterns to load local files to Databricks.

Databricks does not provide any native tools for downloading data from the internet, but you can use open source tools in supported languages.

Add data from local files

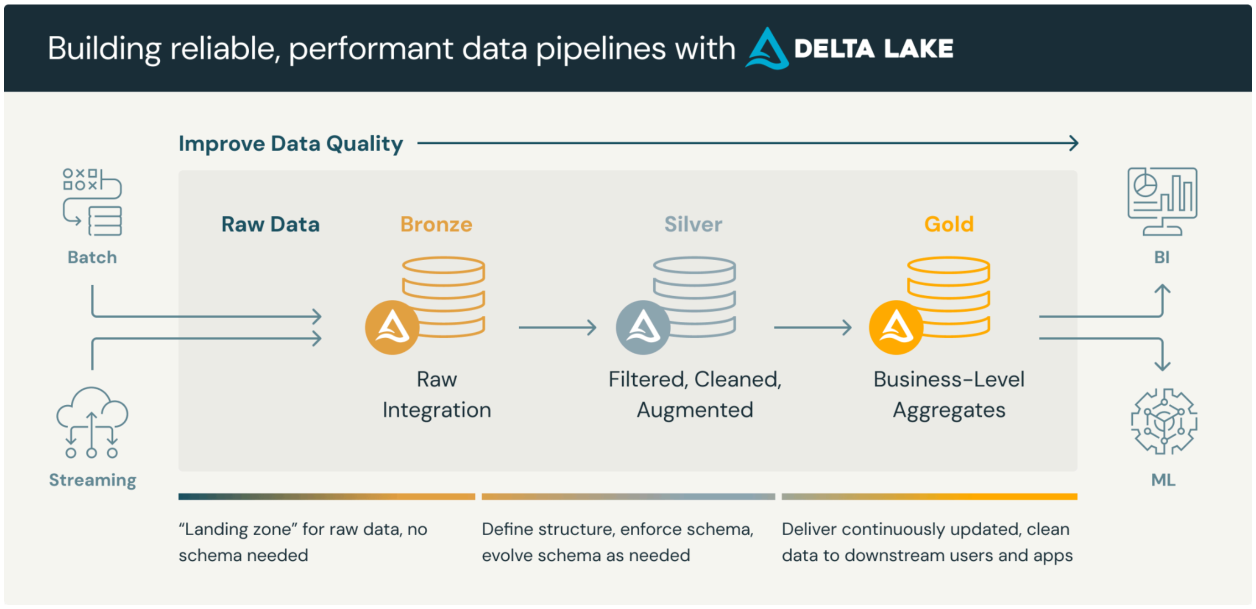
You can upload local files to Databricks to create a Delta table or store data in volumes. To access these and other data source options, click ⊕ **New > Data**.

- Click **Create or modify table** to upload CSV, TSV, JSON, XML, Avro, Parquet, or text files into Delta Lake tables.
- Click **Upload files to volume** to upload files in any format to a Unity Catalog volume, including structured, semi-structured, and unstructured data.

Week 2: Data Engineering Workflows & Medallion Architecture

What is a medallion architecture?

A medallion architecture is a data design pattern used to logically organize data in a lakehouse, with the goal of incrementally and progressively improving the structure and quality of data as it flows through each layer of the architecture (from Bronze ⇒ Silver ⇒ Gold layer tables). Medallion architectures are sometimes also referred to as "multi-hop" architectures.



Building reliable, performant data pipelines with DELTA LAKE
Improve Data Quality
Batch
Streaming
Raw Data
Bronze
Raw Integration
Silver
Filtered, Cleaned, Augmented
Gold
Business-Level Aggregates
BI
ML
"Landing zone" for raw data, no schema needed
Define structure, enforce schema, evolve schema as needed
Deliver continuously updated, clean data to downstream users and apps

Building data pipelines with medallion architecture

Databricks provides tools like Lakeflow Declarative Pipelines that allow users to instantly build data pipelines with Bronze, Silver and Gold tables from just a few lines of code. And, with streaming tables and materialized views, users can create streaming Lakeflow pipelines built on Apache Spark™ Structured Streaming that are incrementally refreshed and updated.

Bronze layer (raw data)

The Bronze layer is where we land all the data from external source systems. The table structures in this layer correspond to the source system table structures "as-is," along with any additional metadata columns that capture the load date/time, process ID, etc. The focus in this layer is quick Change Data Capture and the ability to provide an historical archive of source (cold storage), data lineage, auditability, reprocessing if needed without rereading the data from the source system.

Silver layer (cleansed and conformed data)

In the Silver layer of the lakehouse, the data from the Bronze layer is matched, merged, conformed and cleansed ("just-enough") so that the Silver layer can provide an "Enterprise view" of all its key business entities, concepts and transactions. (e.g. master customers, stores, non-duplicated transactions and cross-reference tables).

The Silver layer brings the data from different sources into an Enterprise view and enables self-service analytics for ad-hoc reporting, advanced analytics and ML.

Gold layer (curated business-level tables)

Data in the Gold layer of the lakehouse is typically organized in consumption-ready "project-specific" databases. The Gold layer is for reporting and uses more de-normalized and read-optimized data models with fewer joins. The final layer of data transformations and data quality rules are applied here. Final presentation layer of projects such as Customer Analytics, Product Quality Analytics, Inventory Analytics, Customer Segmentation, Product Recommendations, Marking/Sales Analytics etc. fit in this layer. We see a lot of Kimball style star schema-based data models or Inmon style Data marts fit in this Gold Layer of the lakehouse.

Benefits of a lakehouse architecture

- Simple data model
- Easy to understand and implement
- Enables incremental ETL
- Can recreate your tables from raw data at any time
- ACID transactions, time travel

Lakeflow Jobs

Lakeflow Jobs is workflow automation for Databricks, providing orchestration for data processing workloads so that you can coordinate and run multiple tasks as part of a larger workflow. You can optimize and schedule the execution of frequent, repeatable tasks and manage complex workflows.

This article introduces concepts and choices related to managing production workloads using Lakeflow Jobs.

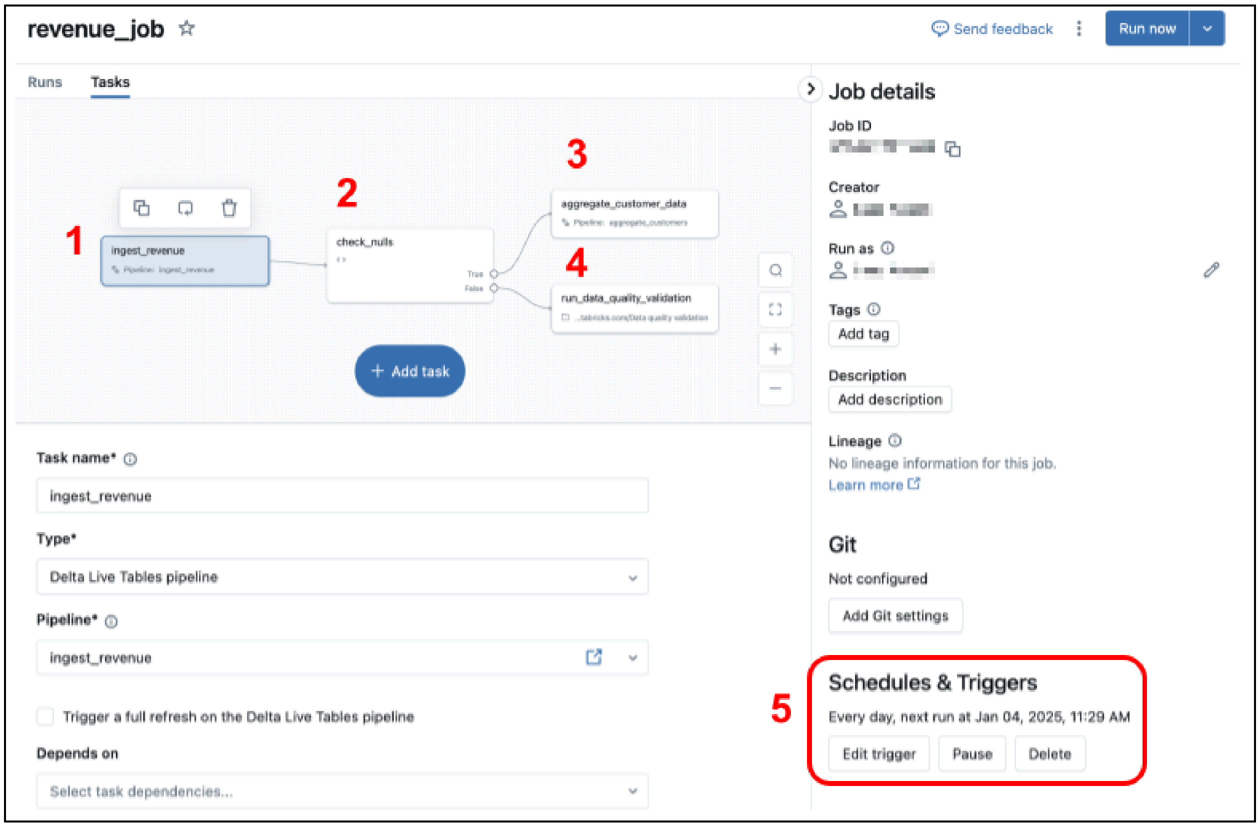
What are jobs?

In Databricks, a job is used to schedule and orchestrate tasks on Databricks in a workflow. Common data processing workflows include ETL workflows, running notebooks, and machine learning (ML) workflows, as well as integrating with external systems like dbt.

Jobs consist of one or more tasks, and support custom control flow logic like branching (if / else statements) or looping (for each statements) using a visual authoring UI. Tasks can load or transform data in an ETL workflow, or build, train and deploy ML models in a controlled and repeatable way as part of your machine learning pipelines.

Example: Daily data processing and validation job

The example below shows a job in Databricks.



Task name*

ingest_revenue

Type*

Delta Live Tables pipeline

Pipeline*

ingest_revenue

☐ Trigger a full refresh on the Delta Live Tables pipeline

Depends on

Select task dependencies...

Job details

Job ID

[redacted]

Creator

[redacted]

Run as

[redacted]

Tags

Add tag

Description

Add description

Lineage

No lineage information for this job.
Learn more

Git

Not configured

Add Git settings

Schedules & Triggers

Every day, next run at Jan 04, 2025, 11:29 AM

Edit trigger Pause Delete

This example job has the following characteristics:

1. The first task ingests revenue data.
2. The second task is an if / else check for nulls.
3. If not, then a transformation task is run.
4. Otherwise, it runs a notebook task with a data quality validation.
5. It is scheduled to run every day at 11:29 AM.

Common use cases

From foundational data engineering principles to advanced machine learning and seamless tool integration, these common use cases showcase the breadth of capabilities that drive modern analytics, workflow automation, and infrastructure scalability.

Orchestration concepts

There are three main concepts when using Lakeflow Jobs for orchestration in Databricks: jobs, tasks, and triggers.

Job - A job is the primary resource for coordinating, scheduling, and running your operations. Jobs can vary in complexity from a single task running a Databricks notebook to hundreds of tasks with conditional logic and dependencies. The tasks in a job are visually represented by a Directed Acyclic Graph (DAG). You can specify properties for the job, including:

- Trigger - this defines when to run the job.
- Parameters - run-time parameters that are automatically pushed to tasks within the job.
- Notifications - emails or webhooks to be sent when a job fails or takes too long.
- Git - source control settings for the job tasks.

Task - A task is a specific unit of work within a job. Each task can perform a variety of operations, including:

- A notebook task runs a Databricks notebook. You specify the path to the notebook and any parameters that it requires.
- A pipeline task runs a pipeline. You can specify existing Lakeflow Declarative Pipelines, such as a materialized view or streaming table.
- A Python script tasks runs a Python file. You provide the path to the file and any necessary parameters.

There are many types of tasks.Tasks can have dependencies on other tasks, and conditionally run other tasks, allowing you to create complex workflows with conditional logic and dependencies.

Trigger - A trigger is a mechanism that initiates running a job based on specific conditions or events. A trigger can be time-based, such as running a job at a scheduled time (for example, ever day at 2 AM), or event-based, such as running a job when new data arrives in cloud storage.

Monitoring and observability

Jobs provide built-in support for monitoring and observability. The following topics give an overview of this support. For more details about monitoring jobs and orchestration.

Job monitoring and observability in the UI - In the Databricks UI you can view jobs, including details such as the job owner and the result of the last run, and filter by job properties. You can view a history of job runs, and get detailed information about each task in the job.

Job run status and metrics - Databricks reports job run success, and logs and metrics for each task within a job run to diagnose issues and understand performance.

Notifications and alerts - You can set up notifications for job events via email, Slack, custom webhooks and a host of other options.

Custom queries through system tables - Databricks provides system tables that record job runs and tasks across the account. You can use these tables to query and analyze job performance and costs. You can create dashboards to visualize job metrics and trends, to help monitor the health and performance of your workflows.

Limitations

The following limitations exist:

- A workspace is limited to 2000 concurrent task runs. A `429 Too Many Requests` response is returned when you request a run that cannot start immediately.
- The number of jobs a workspace can create in an hour is limited to 10000 (includes “runs submit”). This limit also affects jobs created by the REST API and notebook workflows.
- A workspace can contain up to 12000 saved jobs.

- A job can contain up to 1000 tasks.

Can I manage workflows programmatically?

Databricks has tools and APIs that allow you to schedule and orchestrate your workflows programmatically, including the following:

- Databricks CLI
- Databricks Asset Bundles
- Databricks extension for Visual Studio Code
- Databricks SDKs
- Jobs REST API

Data modeling

This article introduces considerations, caveats, and recommendations for data modeling on Databricks. It is targeted toward users who are setting up new tables or authoring ETL workloads, with an emphasis on understanding Databricks behaviors that influence transforming raw data into a new data model. Data modeling decisions depend on how your organization and workloads use tables. The data model you choose impacts query performance, compute costs, and storage costs. This includes an introduction to the foundational concepts in database design with Databricks.

Database management concepts

A lakehouse built with Databricks shares many components and concepts with other enterprise data warehousing systems. Consider the following concepts and features while designing your data model.

Transactions on Databricks

Databricks scopes transactions to individual tables. This means that Databricks does not support multi-table statements (also called multi-statement transactions).

For data modeling workloads, this translates to having to perform multiple independent transactions when ingesting a source record requires inserting or updating rows into two or more tables. Each of these transactions can succeed or fail independent of other transactions, and downstream queries need to be tolerant of state mismatch due to failed or delayed transactions.

Primary and foreign keys on Databricks

Primary and foreign keys are informational and not enforced. This model is common in many enterprise cloud-based database systems, but differs from many traditional relational database systems.

Joins on Databricks

Joins can introduce processing bottlenecks in any database design. When processing data on Databricks, the query optimizer seeks to optimize the plan for joins, but can struggle when an individual query must join results from many tables. The optimizer can also fail to skip records in a table when filter parameters are on a field in another table, which can result in a full table scan.

Working with nested and complex data types

Databricks supports working with semi-structured data sources including JSON, Avro, and ProtoBuff, and storing complex data as structs, JSON strings, and maps and arrays.

Normalized data models

Databricks can work well with any data model. If you have an existing data model that you need to query from or migrate to Databricks, you should evaluate performance before rearchitecting your data.

If you are architecting a new lakehouse or adding datasets to an existing environment, Databricks recommends against using a heavily normalized model such as third normal form (3NF).

Models like the star schema or snowflake schema perform well on Databricks, as there are fewer joins present in standard queries and fewer keys to keep in sync. In addition, having more data fields in a single table allows the query optimizer to skip large amounts of data using file-level statistics.

Data Validation Techniques

Clean and validate data with batch or stream processing

Cleaning and validating data is essential for ensuring the quality of data assets in a lakehouse. This article outlines Databricks product offerings designed to facilitate data quality, as well as providing recommendations for defining business logic to implement custom rules.

Schema enforcement on Databricks

Delta Lake provides semantics to enforce schema and constraint checks on write, which provides guarantees around data quality for tables in a lakehouse.

Schema enforcement ensures that data written to a table adheres to a predefined schema. Schema validation rules vary by operation.

To handle schema evolution, Delta provides mechanisms for making schema changes and evolving tables. It is important to carefully consider when to use schema evolution to avoid dropped fields or failed pipelines. For details on manually or automatically updating schemas.

Table constraints

Constraints can take the form of informational primary key and foreign key constraints, or enforced constraints.

Table constraints on Databricks are either enforced or informational.

Enforced constraints include `NOT NULL` and `CHECK` constraints.

Informational constraints include primary key and foreign key constraints.

Deal with null or missing values

NOT NULL can be enforced on Delta tables. It can only be enabled on an existing table if no existing records in the column are null, and prevents new records with null values from being inserted into a table.

Pattern enforcement

Regular expressions (regex) can be used to enforce expected patterns in a data field. This is particularly useful when dealing with textual data that needs to adhere to specific formats or patterns.

To enforce a pattern using regex, you can use the `REGEXP` or `RLIKE` functions in SQL. These functions allow you to match a data field against a specified regex pattern.

Here's an example of how to use the `CHECK` constraint with regex for pattern enforcement in SQL:

SQL

```sql
CREATE TABLE table_name (

 column_name STRING CHECK (column_name REGEXP '^[A-Za-z0-9]+$')

);
```

Value enforcement

Constraints can be used to enforce value ranges on columns in a table. This ensures that only valid values within the specified range are allowed to be inserted or updated.

To enforce a value range constraint, you can use the `CHECK` constraint in SQL. The `CHECK` constraint allows you to define a condition that must be true for every row in the table.

Here's an example of how to use the `CHECK` constraint to enforce a value range on a column:

SQL

```sql
CREATE TABLE table_name (

 column_name INT CHECK (column_name >= 0 AND column_name <= 100)

);
```

Data monitoring

Databricks provides data quality monitoring services, which let you monitor the statistical properties and quality of the data in all of the tables in your account.

Cast data types

When inserting or updating data in a table, Databricks casts data types when it can do so safely without losing information.

See the following articles for details about casting behaviors:

- `cast` function
- SQL data type rules
- ANSI compliance in Databricks Runtime

Custom business logic

You can use filters and `WHERE` clauses to define custom logic that quarantines bad records and prevents them from propagating to downstream tables. `CASE WHEN ... OTHERWISE` clauses allow you to define conditional logic to gracefully apply business logic to records that violate expectations in predictable ways.

SQL

```sql
DECLARE current_time = now()
```

```sql
INSERT INTO silver_table

 SELECT * FROM bronze_table

 WHERE event_timestamp <= current_time AND quantity >= 0;



INSERT INTO quarantine_table

 SELECT * FROM bronze_table

 WHERE event_timestamp > current_time OR quantity < 0;
```

For example, you might have an upstream system that isn't capable of encoding `NULL` values, and so the placeholder value `-1` is used to represent missing data. Rather than writing custom logic for all downstream queries in Databricks to ignore records containing `-1`, you could use a case when statement to dynamically replace these records as a transformation.

SQL

```sql
INSERT INTO silver_table
 SELECT
   * EXCEPT weight,
   CASE
     WHEN weight = -1 THEN NULL
     ELSE weight
   END AS weight
 FROM bronze_table;
```