DATA INGESTION WITH LAKEFLOW CONNECT

→ Data Engg. with Databricks

Unity Catalog
- Centralised data catalog that provides access control, data lineage, auditing, quality monitoring and data discovery across Databricks workspaces.

Lakeflow
- end to end data engineering solution to deliver high quality data
- provides a unified platform for data ingestion, transformation & orchestration

COMPONENTS
- Lakeflow Connect: ingestion connectors that simplify data ingestion
- Lakeflow Job: Workflow automation tool that orchestrates data processing workloads
- Lakeflow Declarative Pipelines: Framework for building batch and streaming data pipelines using SQL and Python to accelerate ETL development

→ what is Lakeflow Connect

Lakeflow Connect
- provides built-in connectors to streamline data ingestion

BENEFITS
- Reduces cost and accelerates time to value
- Self service interface enabling easy data ingestion from enterprise application
- Unified observability and governance

TYPES OF INGESTION
- Manual File Upload : upload local files directly into a volume or a table
- Standard Connectors: Batch/Incremental Batch/Streaming ingestion from various sources
- Managed Connectors : Purpose built for enterprise applications (databases, SaaS)

BATCH INGESTION
- Loads data as batches of rows into databricks, usually on a schedule
- Common Techniques
  - SQL : CREATE TABLE AS SELECT
  - Python : spark.read.load()

INCREMENTAL BATCH INGESTION
- automatically detects new records in data source and ingests those only
- Common Techniques
  - SQL → Copy Into
  - Python : spark.readStream [auto loader w/ ~~continuous~~ timed trigger]
  - Declarative Pipelines : CREATE OR REFRESH STREAMING TABLE

STREAMING INGESTION
- data is continuously loaded as it is generated
- Kafka, Kinesis, Google Pub/Sub, Apache Pulsar
- Common Techniques
  - Python : spark.readStream [auto loader w/ continuous trigger]
  - Declarative Pipelines : trigger mode continuous

→ Delta Lake Review

Delta Lake

- open source protocol for reading and writing files to cloud storage
- goal is to ingest files from external data sources into Delta Lake as Delta Tables

DELTA TABLE

- open format of table supporting Lakehouse architecture
- stores data within a folder directory
- allows time travel
- Delta Lake allows easy way to modify and manage data in a data lake
- Delta adds Delta logs to keep track of all transactions on data and table versions

KEY FEATURES

(i) ACID Transactions
(ii) DML - Data Manipulation Language (DML)
(iii) Time Travel
(iv) Schema Evolution and Enforcement
(v) Unified batch/streaming processing
(vi) Optimisation, performance and scalability

Medallion Architecture

- Bronze: Raw Data
- Silver: Cleaned, Transformed and Enriched data
- Gold: Curated, Aggregated, High quality data

→ Data Ingestion from Cloud Storage

Create Table As (CTAs)

- read_files() function is used to read files from a specified location and return the data in tabular format
- Supports JSON, CSV, Parquet, XML, Text etc.
- Automatically detects file format and infers unified schema across all files
- Can be used for streaming with autoloader

Copy Into

- bulk load from files in cloud object storage into table
- Ideal for scenarios where cloud storage location is continuously adding files
- skips any files that have already been loaded into table
- Supports parquet, XML, JSON and others
- Only new files are ingested

Auto Loader

- Automatically and efficiently process new datafiles in batch/streaming as they arrive in cloud storage
- Support both SQL and Python and can process billions of files
- Build upon Spark Structured Streaming

AUTO LOADER IN SQL

- Declarative Pipelines
- CREATE OR REFRESH STREAMING TABLE catalog.schema.table SCHEDULE EVERY 1 HOUR AS SELECT * FROM STREAM read_files('dir-path', format => 'file_type')

AUTO LOADER IN PYTHON

- spark.readStream.format("cloudFiles")
  - option("cloudFiles.format", "json")
  - option("cloudFiles.schemaLocation", "path")
  - load("/Volumes/catalog/schema/files")
  - writeStream.option("checkpointLocation", "path")
  - trigger(processingTime="5 seconds")
  - toTable("catalog.database.table")

(H) Managed Table vs External Table

MANAGED TABLE

(i) Databricks manages both data and metadata
(ii) Stored within Databricks managed storage
(iii) Dropping table also deletes data
(iv) Recommended for creating new tables

EXTERNAL TABLE

(i) Databricks only manages the metadata
(ii) Dropping table does not delete the data
(iii) Supports multiple table formats, including Delta Lake
(iv) Ideal for sharing data across platforms or using external data

(#) CTAS vs COPY INTO vs Auto Loader

	CTAS	COPY INTO	AUTO LOADER
INGESTION TYPE	Batch	Incremental Ingestion Batch	Incremental (Batch/Streaming)
INTERFACE	Python, SQL	SQL	Python, SQL w/ Declarative Pipelines
IDEMPOTENCY	No	Yes	Yes
SCHEMA EVOLUTION	Manual/Inferred during reading	Supported with options	Automatically detected and evolved as needed
LATENCY	High	Moderate	Depends on configuration
EASE OF USE	Simple	Simple, SQL-based	Intermediate to advanced

→ Append metadata columns on ingest

(#) Metadata

- we can append metadata columns information from input data source files

METADATA COLUMNS SUPPORTED

- file_path
- file_name
- file_size
- file_modification_time
- file_block_start
- file_block_length

Metadata columns must be declared when creating/selecting the data

→ Rescued Data Column

(#) Rescuing Malformed Rows on Ingestion

└ read_files/spark.read/Autoloader provide a rescued data column if the raw data does not match schema

PYTHON

└ .option("rescuedDataColumn", "_rescued_data")

→ Ingesting JSON

(#) JSON

- Key-value pair
- Value: String / Number / Object / Array / Boolean
- Objects can be flat, meaning all key-value pairs are at one level or they can be nested
- Columns in tables can hold JSON formatted strings ~~without~~ as value

(#) Working with JSON data

STRING

- can hold any JSON string without constraints
- less performant

STRUCT

- has a defined schema, enforces JSON schema
- more efficient
- JSON String → String
- JSON Number → INT/FLOAT/DOUBLE
- JSON Boolean → BOOLEAN
- JSON Object → STRUCT<>
- JSON Array → ARRAY<>
- JSON String Structure Determining

(i) Use built-in schema_of_json function to automatically derive the schema from JSON string

(ii) Use from_json function to return a struct column using JSON string and specified schema

VARIANT

- Can store any type of data
- Ideal for semi-structured data
- Highly flexible
- Improved performance

→ Ingesting Enterprise Data

(#) Lakeflow Connect Managed Connectors

- Designed to simplify process of ingesting data from a variety of enterprise databases and apps
- Provides easy to use UI
- Fully managed by Databricks
- Works with Workday, Salesforce, PostgreSQL, SQL Server and more

SAAS INTEGRATION
- Collects data from external sources to Streaming Delta Tables using serverless compute
- Lakeflow Serverless Declarative Pipelines job collects credential from Unity Catalog
- Job reaches out to publically accessible data source
- Service transforms data and stores it to Streaming Delta Table

DATABASE INTEGRATION
- Architecture is designed to move data into Streaming Delta Tables - but from external databases
- Declarative Pipelines job retrieves credential securely from Unity Catalog
- Uses those credentials to connect to external Database sources
- Job collects the latest state and change logs, storing staged data in Unity Catalog volume
- Serverless Declarative Pipelines job processes that staged data and loads it into a Streaming Delta Table

(A) Ingestion Gateway
- Dedicated pipeline that connects database to extract metadata, snapshots and change logs
- Unity Catalog Volume acts as intermediate staging layer, enabling next pipeline to pick up and stream data

(B) Partner Connect
- Directly connects to partner platforms

→ Additional Features

(#) Lakehouse Federation
- Allows you to query external data sources without moving your data.
- Useful for ad-hoc reporting, proof of concept work, exploratory phase of new ETL pipelines or reports and supporting workloads during incremental migration

(#) Zerobus
- Lakeflow Connect API allows developers to write event data directly to their lakehouse at high throughput, low latency
- Useful for IoT, Clickstreams and Telemetry

(#) Delta Sharing
- Allows to securely share data across platforms, clouds and regions.

(#) Databricks Marketplace
- Open marketplace for all data, analytics and AI

→ Ingesting into Existing Delta Tables

(#) MERge Into
- supports schema evolution/enforcement
- (1) Matched rows — Update / Delete
- (∞) Unmatched rows by target — Insert
- (1∞) Unmatched rows by source — Update / Delete

Example:

```
MERGE INTO target_table target USING source_table source ON target.id = source.id
WHEN MATCHED AND source.status = 'update' THEN
UPDATE SET target.email = source.email, target.status = source.status
WHEN MATCHED AND source.status = 'delete' THEN DELETE
WHEN NOT MATCHED THEN INSERT (id, name, email)
VALUES (source.id, source.name, source.email);
```

DEPLOY WORKLOADS w/ LAKEFLOW JOBS

→ Lakeflow Jobs

Lakeflow Jobs

- Unified orchestration for data, analytics and AI workloads directly on the Data Intelligence Platform

BENEFITS

- Simple authoring
- Actionable Insights
- Proven Reliability

COMPONENTS

- Processing Engine → Photon
- Governance → Unity Catalog
- Storage → Delta Lake

PROCESS

- Workflow Engine coordinates everything
- Compute layer supports various workloads type
- Multiple trigger types: scheduled (time-based), continuous (always running), File arrival (event driven), Table Updates
- Observability → Monitoring and Troubleshooting
- Control Flow → Managing task dependencies and execution order

→ Building Blocks of Lakeflow Jobs

JOB

- Primary resource for scheduling, coordinating and running operations like ETL, data processing etc.
- consists of one or multiple tasks

TASK

- Single unit of work within a job that executes a specific workload
- Each task has a specific purpose like defining path, adding libraries, adding parameters etc.
- Languages supported: Python, SQL, Scala, R etc.
- Various control flows can be established between tasks, including sequential, parallel, ~~continuous~~ conditional, run job, for each etc.

Types of computes

INTERACTIVE CLUSTER

- Can be shared by multiple users
- Used for ad-hoc analysis, data exploration and development

JOB CLUSTERS

- 50% cheaper but subject to cloud providers' start-up time
- can use the same cluster across tasks for better price to performance

SERVERLESS

- Fully managed service with faster clusters and auto scaling capabilities
- provides the lowest overall TCO

SQL warehouse

- Purpose built for SQL queries, dashboards and BI

* Use performance optimised setting in job details page to choose between lower cost and faster executions for serverless computes.
  - Faster job startup and execution

(*) A job can have one or more tasks, and each task can be assigned its own compute resource
└ Tasks in same job can either share compute or use different computes as required

→ DAG

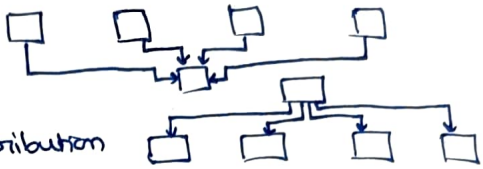
(#) DAG – Directed Acrylic Graph
├ conceptual representation of series of activities
└ Databricks jobs support task orchestration, which can be done as DAG via Databricks UI, API, SDK etc

COMMON WORKLOAD PATTERNS

(i) Sequence – Data Transformation (Bronze/Silver/Gold)

(ii) Funnel – Multi data source, Data collection

(iii) Fan Out – Single data source, data ingestion/distribution



→ Common Task Configuration Options

(#) Types of Parameters

TASK PARAMETERS
├ key-value pairs that lets us pair values into tasks
└ supports conditional execution, looping and passing context between tasks

JOB PARAMETERS
├ key-value pairs defined at job level that provides default values for entire workflow
└ applied to all tasks and overrides task parameters

(*) DYNAMIC VALUE REFERENCES
├ referencing values at runtime from job and task context
└ references uses {{ }} notion. e.g. {{ start_time.day }}, {{ task_name }}

(#) Notification Configuration
├ Job Level Notification: notification alert is sent after successful completion of job
└ Task Level Notification: notification alert is sent after successful completion of task

NOTIFICATION
├ Job triggers notification when task begins, completes and fails
├ Can also be triggered in case of late jobs
├ Issued for duration threshold warning / timeout alerts
└ Streaming backlog – issue with getting streaming data

(#) Retry Policy
└ A policy that determines when and how many times failed runs are retried

→ Schedules and Triggers

(#) Triggers
└ rule that automatically starts a job run based on a specific condition/schedule

TYPES
├ Time based schedules
├ Manual trigger
├ Continuous execution
├ File arrival events
└ Table updates

SCHEDULED TRIGGER
- Automatically runs job
- Scheduled jobs using cron expressions

FILE ARRIVAL TRIGGER
- Automatically trigger jobs when new files are detected
- Event driven processing, ideal for automatic jobs with unpredicatable/ irregular data ingestion patterns

CONTINUOUS TRIGGER
- Continously runs job by starting a new run as soon as the previous one finishes/fails
- Built-in reply logic, ideal for streaming workloads

MANUAL TRIGGERS
- Runs on-demand, can be started from UI
- Best for ad-hoc runs, debugging and one-off backfills

TABLE UPDATE TRIGGER
- automatically starts a job when specified tables are updated
- monitors one or more tables for changes

→ Conditional and Iterative Tasks

(#) Run If Conditional Task Dependencies
- Control Task execution based on outcomes of upstream tasks
- Supports a variety of dependency condition like all succeeded, all failed, atleast one succeed and more
- Task dependencies provide control flow logic based on upstream task status

WORKFLOW
- Job Initialisation
- Waiting for all tasks to complete
- Dependent task started

(#) If Else Task Dependencies
- Add sophisticated boolean logic to workflows
- Boolean operators (==, !=, >, >=, <, <=) evaluate expressions against task results, parameter values or computed metrics

(#) For Each Task Dependencies
- Input Processing: Task loops over an input array, passing each item as {{input}} to the nested table
- Parallel Execution: Configurable concurrency allows multiple iterations to run simultaneously
- Dependency Management: Downstream tasks depend on the completion of the entire For Each container, not individual iterations
- For Each tasks need defining two components: For Each task and Nested task

FOR EACH TASK
- Top level container task that manages the loop

NESTED TASK
- Actual task that executes for each iterations.

→ Handling Task Failures

(#) Repair and Run

- Repair feature allows to re-run and override task parameters
- Reduces time and resources required to recover
- In case of task failure, we can modify task and run again, or modify parameters and run again

BENEFITS

- Resource Optimisation
- Reduced Risk
- Faster Resolution

→ Monitoring Jobs ~~Parameters~~ Performance

(#) System Tables

- system. lakeflow is a built-in, read only catalog that logs all job activity across workspaces in a region
- Timeline tables slice long runs hourly using period_start_time and period_end_time, enabling reliable duration, concurrency and SLA analytics
- Key Tables
  - (i) Jobs → Job basic info
  - (ii) Job_tasks → tasks basic definition
  - (iii) job_run_timeline → each job run over time
  - (iv) Job_task_run_timeline → each task run over time
  - (v) pipelines → pipelines basic info

(#) Spark UI

- provides detailed performance insight for optimisation

BENEFITS

- (i) Timeline Analysis
- (ii) Task level details
- (iii) Query Performance Details

→ Lakeflow Jobs in Production

(#) Compute Strategy

- selecting right compute options for performance, cost and operational requirement

(#) Modular Design

- Implementing architectural patterns that support maintainability, reusability and team collaboration

(#) Git Integration

- Version control and deployment practices that ensure consistency and enable CI/CD workflows

(#) Performance Monitoring: Proactive monitoring and optimization strategies that ensure SLA compliance and cost efficiency

(#) Types of clusters

INTERACTIVE

- Best for performing ad-hoc analytics, data exploration or development but not production
- Costly for Job runs
- Limited Scalability
- Availability may be an issue

JOB CLUSTERS
- Cheaper as they terminate when the job ends, reducing resource usage and cost
- startup latency
- Subject to cloud provider startup time
- Maintainence burden

SERVERLESS
- Simplicity and reliability
- High efficiency
- Reliability

Traditional Pricing model
- Direct costs: DBUs paid to Databricks and infrastructure costs paid directly to cloud providers
- Operational Overhead: often overlooked but includes time spend on infrastructure development, automation development etc.
- Hidden Complexity: Managing multiple billing relationship optimising across different cost categories etc.

Serverless Pricing
- Fully managed service: operationally simple and more reliable
- Fast cluster, autoscaling: better user experience and lower cost
- Out of the box performance and optimisations: lower overall TCO

Modular Orchestration
- Decomposition Strategy: break complex DAGs into logical business units rather than technical components
- Parent Child Relationship: creates clean seperation of concerns while maintaining overall workflow coordination

BENEFITS
- Maintainability
- Reusability
- Team Collaboration
- Testing

Git
- Prevents unintentional changes to production jobs
- Simplifying the job definition
- Makes deployment easy for notebook through CI/CD
- Can connect with various Git platforms

→ Best Practices

Compute and Cost Optimisation
- Use job/serverless clusters and avoid interactive clusters
- Enable photon for faster, cheaper execution
- Reuse clusters to reduce startup time/cost

Orchestration and Modularity
- Break complex pipelines into modular/run jobs
- Use multitask jobs for parallel and scalable execution
- Apply Run If, If/Else, For Each tasks
- Limit jobs to manage task counts

Monitoring and Governance
- Use service principals for job ownership and data access
- Configure alerts for failures and delays

- Leverage repair and run for reducing cost on reprocessing
- Parameterise tasks for reusability and flexibility

JOB CLUSTERS
- Cheaper as they terminate when the job ends, reducing resource usage and cost
- startup latency
- Subject to cloud provider startup time
- Maintainence burden

SERVERLESS
- Simplicity and reliability
- High efficiency
- Reliability

Traditional Pricing model
- Direct costs: DBUs paid to Databricks and infrastructure costs paid directly to cloud providers
- Operational Overhead: often overlooked but includes time spend on infrastructure development, automation development etc.
- Hidden Complexity:- Managing multiple billing relationship optimising across different cost categories etc.

Serverless Pricing
- Fully managed service : operationally simple and more reliable
- Fast cluster, autoscaling: better user experience and lower cost
- Out of the box performance and optimisations : lower overall TCO

Modular Orchestration
- Decomposition Strategy : break complex DAGs into logical business units rather than technical components
- Parent Child Relationship: creates clean seperation of concerns while maintaining overall workflow coordination

BENEFITS
- Maintainability
- Reusability
- Team Collaboration
- Testing

Git
- Prevents unintentional changes to production job
- Simplifying the job definition
- Makes deployment easy for notebook through CI/CD
- Can connect with various Git platforms

→ Best Practices

Compute and Cost Optimisation
- Use job/serverless clusters and avoid interactive clusters
- Enable photon for faster, cheaper execution
- Reuse clusters to reduce startup time/cost

Orchestration and Modularity
- Break complex pipelines into modular/run jobs
- Use multitask jobs for parallel and scalable execution
- Apply Run If, If/Else, For Each tasks
- Limit jobs to manage task counts

Monitoring and Governance
- Use service principals for job ownership and data access
- Configure alerts for failures and delays

- Leverage repair and run for reducing cost on reprocessing
- Parameterise tasks for reusability and flexibility