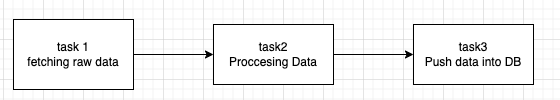
Assignment 1: Pipeline design

Macro layout:

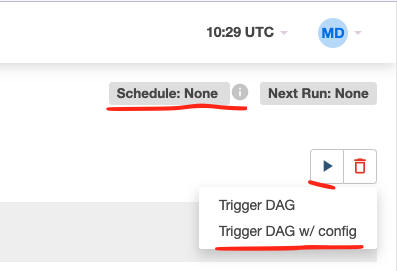


Technologies uses: airflow, spark, PostgreSQL



Flow Stages:

1. First stage loads the raw data, can be done on demand with Airflow input JSON payload which will provide input parameters, or by predefined timing.



Sort the input raw data, create validation for its content and load into spark DF

1. Use the Spark DF to create which even data manipulation and processing we wish.
2. Push the data into PostgreSQL SQL

Python Airflow code example:

from datetime import datetime, timedelta

from airflow import DAG

from airflow.operators.python\_operator import PythonOperator

from airflow.utils.dates import days\_ago

# Define custom functions for tasks

def fetch\_raw\_data(\*\*kwargs):

# Logic to fetch raw data

pass

def process\_data\_with\_pyspark(\*\*kwargs):

# Logic to process data with PySpark

from pyspark.sql import SparkSession

# Initialize SparkSession

spark = SparkSession.builder \

.appName("Data Processing with PySpark") \

.getOrCreate()

# PySpark logic to process data

# Example:

# df = spark.read.csv('path\_to\_input\_file.csv')

# processed\_df = df.select(...)

# Stop SparkSession

spark.stop()

def push\_processed\_data\_to\_db(\*\*kwargs):

# Logic to push processed data into database

pass

# Define default arguments for the DAG

default\_args = {

'owner': 'airflow',

'depends\_on\_past': False,

'start\_date': days\_ago(1),

'email\_on\_failure': False,

'email\_on\_retry': False,

'retries': 1,

'retry\_delay': timedelta(minutes=5),

}

# Define the DAG

dag = DAG(

'data\_processing\_pipeline',

default\_args=default\_args,

description='A DAG to fetch, process, and push data',

schedule\_interval=timedelta(days=1),

)

# Define tasks

fetch\_data\_task = PythonOperator(

task\_id='fetch\_raw\_data',

python\_callable=fetch\_raw\_data,

dag=dag,

)

process\_data\_task = PythonOperator(

task\_id='process\_data\_with\_pyspark',

python\_callable=process\_data\_with\_pyspark,

dag=dag,

)

push\_to\_db\_task = PythonOperator(

task\_id='push\_processed\_data\_to\_db',

python\_callable=push\_processed\_data\_to\_db,

dag=dag,

)

# Define dependencies

fetch\_data\_task >> process\_data\_task >> push\_to\_db\_task

DB + Schema:

1. Tasks table:

task\_id (Primary Key)

task\_name

task\_description

task\_code

created\_at

updated\_at

2. TaskRuns table:

run\_id (Primary Key)

task\_id (Foreign Key)

start\_time

end\_time

status (e.g., running, completed, failed)

result

telemetry

created\_at

updated\_at

3. ProcessedData table:

data\_id (Primary Key)

task\_run\_id (Foreign Key, referencing TaskRuns)

data (stores the processed data)

created\_at

updated\_at

comments (**red** – tables, **black** – columns):

* The **ProcessedData** table is introduced to store the processed data along with metadata about the task run.
* **data\_id** is a primary key for the **ProcessedData** table.
* **task\_run\_id** is a foreign key referencing the **TaskRuns** table, linking each record in **ProcessedData** to the corresponding task run.
* **data** column stores the processed data. The data type of this column depends on the nature of your processed data (e.g., JSON, text, binary, etc.).
* **created\_at** and **updated\_at** columns track the creation and last update timestamps for records in the ProcessedData table.

This schema extension allows you to store the processed data in the database along with the necessary metadata to track its origin and the task run associated with it.

Query example which involves those 3 tables:  
SELECT

t.task\_name,

tr.start\_time AS task\_start\_time,

tr.end\_time AS task\_end\_time,

tr.status AS task\_status,

pd.data AS processed\_data,

pd.created\_at AS processed\_data\_created\_at

FROM

Tasks t

JOIN

TaskRuns tr ON t.task\_id = tr.task\_id

JOIN

ProcessedData pd ON tr.run\_id = pd.task\_run\_id;

**Why Postgress?**

1. relational model, suit for structed data
2. Robustness, stable and can handle large volume
3. ACID – Ensure Reliability of the data
4. Support json format, got extension for spark, airflow and so on…

**In this use case I want the relational ability to create queries, ability to integrated with plenty of other frameworks (like python , spark…) , and with the addition of the spark it will be pretty scalable**

In which case I would have chosen **MongoDB?**

**If I would want to use unstructured, flexible, high volume and Realtime and still support querying ability**

Pros for mongo:

1. **Unstructured or Semi-Structured Data**: If the raw data or processed data in your pipeline is unstructured or semi-structured (e.g., JSON documents), MongoDB's flexible document-based data model can be advantageous. MongoDB allows you to store and query data without a predefined schema, making it easier to handle data with varying structures.
2. **Flexible Data Model**: MongoDB's document-based data model allows you to store nested or hierarchical data structures easily. If your data processing tasks involve working with complex data structures or nested data, MongoDB's flexible schema can simplify data storage and retrieval.
3. **High Volume and Variety of Data**: MongoDB is designed to handle high volumes of data with ease, making it suitable for scenarios where you need to store and process large amounts of data. Additionally, MongoDB's sharding capabilities allow you to distribute data across multiple nodes for horizontal scalability, enabling you to handle growing data volumes efficiently.
4. **Real-Time Data Processing**: If your data processing pipeline needs to handle real-time data streams or process data in near real-time, MongoDB's support for fast writes and reads can be beneficial. MongoDB's write performance is optimized for high throughput, making it suitable for applications that require low-latency data processing.

In which case I would have chosen **Casandra**?

**If I would care about scalability , and heavy use**

Pros for Casandra:

1. **Scalability and High Throughput**: Cassandra is designed for horizontal scalability and can handle massive amounts of data across multiple nodes. If your data processing pipeline needs to scale to accommodate growing data volumes or handle high write throughput, Cassandra's distributed architecture and linear scalability make it a suitable choice.
2. **Time-Series Data**: If your pipeline deals with time-series data, such as sensor data, logs, or telemetry data, Cassandra's data model is well-suited for storing and querying time-series data efficiently. Cassandra's partitioning and clustering keys allow you to organize data by time, making it easy to retrieve and analyze data based on time ranges.
3. **High Availability and Fault Tolerance**: Cassandra provides built-in features for ensuring high availability and fault tolerance, making it suitable for mission-critical applications that require continuous availability. Cassandra's distributed architecture ensures that data remains available even in the event of node failures or network partitions.
4. **Write-Heavy Workloads**: If your data processing pipeline involves write-heavy workloads, such as ingesting large volumes of data or capturing real-time events, Cassandra's optimized write performance can handle thousands of writes per second across multiple nodes without sacrificing performance or data consistency.

Monitoring

1. With **airflow** I can visually see each task progress and logs
2. With **SQL queries** which kept the results of each run and each task I can create queries to provide monitoring statistics
3. **Grafana** – we will use Grafana as visualization tool. We can attach to it the Postgress as well as the spark to provide monitoring charts, dashboards even set alerts in cases of failures

Query examples

This query retrieves all tasks created within the specified time range:

SELECT \*

FROM Tasks

WHERE created\_at >= 'start\_timestamp' AND created\_at <= 'end\_timestamp';

Retrieve the count of task runs by status:

SELECT status, COUNT(\*) AS count

FROM TaskRuns

GROUP BY status;

This query calculates the average, minimum, and maximum telemetry values associated with a specific task. It joins the TaskRuns and ProcessedData tables on the run\_id and task\_run\_id columns, respectively, and filters the data based on the task\_id

SELECT

AVG(telemetry) AS average\_value,

MIN(telemetry) AS min\_value,

MAX(telemetry) AS max\_value

FROM

TaskRuns tr

JOIN

ProcessedData pd ON tr.run\_id = pd.task\_run\_id

WHERE

tr.task\_id = 'specific\_task\_id';

spark use :

Apache Spark offers a powerful and flexible platform for building scalable, fault-tolerant, and high-performance data processing pipelines. Make me “play” with the data without worries of the performance.

Further enhancements:

1. **Streaming:**

If I the requirement here was for streaming, I would have use this design:  
SYSTEM ARCHITECTURE 
kafka 

Using the Airflow to control the streaming rate , attach Kafka as a streaming tool , still use spark to handle the incoming data for manipulation , and pushing the data in more suitable DB which can the real time volumes , like Casandra

1. **Devops** – would have use docker to keep all our dependencies and deploy it easily
2. **CDC** - Change Data Capture. It is a feature in SQL databases that captures and tracks changes made to data in tables. CDC enables you to identify and capture the data modifications (inserts, updates, and deletes) that occur in a database table over time.

If some change being made in DB (this technologies exists in many DBs also in postgress) It can trigger via webhook alert , which can be attached to the pipeline - in other words if change in the DB occur we can trigger recalculate of monitoring data or new task. It can save us many calculations , timely triggering and effort .