Rainbow Project

Summary

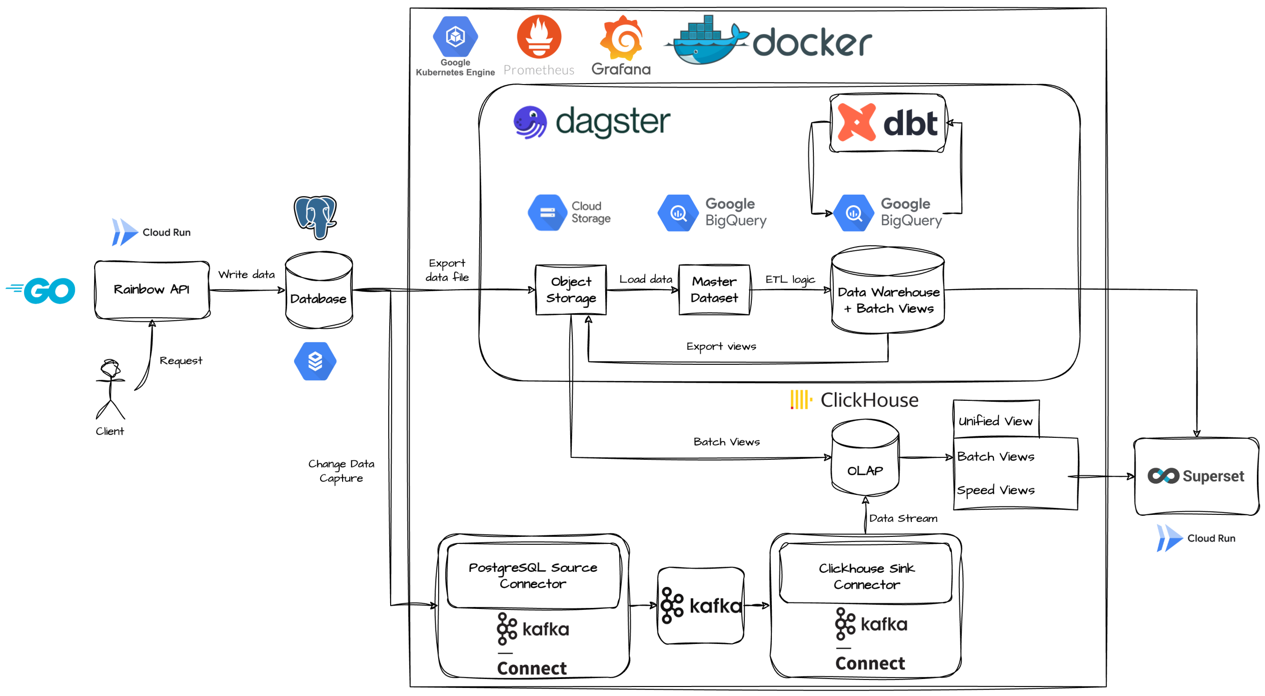
- By building a Data Engineering project from scratch, I have the opportunity to gain hands-on experience with various languages, frameworks, patterns, and tools that a Data Engineer uses throughout the product development lifecycle.

- I successfully migrated my project from local development to Google Cloud Platform, a process that took approximately four weeks. While there are still some areas for improvement, I have consistently put my best effort into developing, maintaining, and enhancing the project.

> ***NOTE:*** This project is an internal project that I have been working during my internship at ***Tiki***. The team size of the project is 1, which only involves me. The main purpose of this project is to build a knowledge base and gain practical experience in the Data Engineering career path.

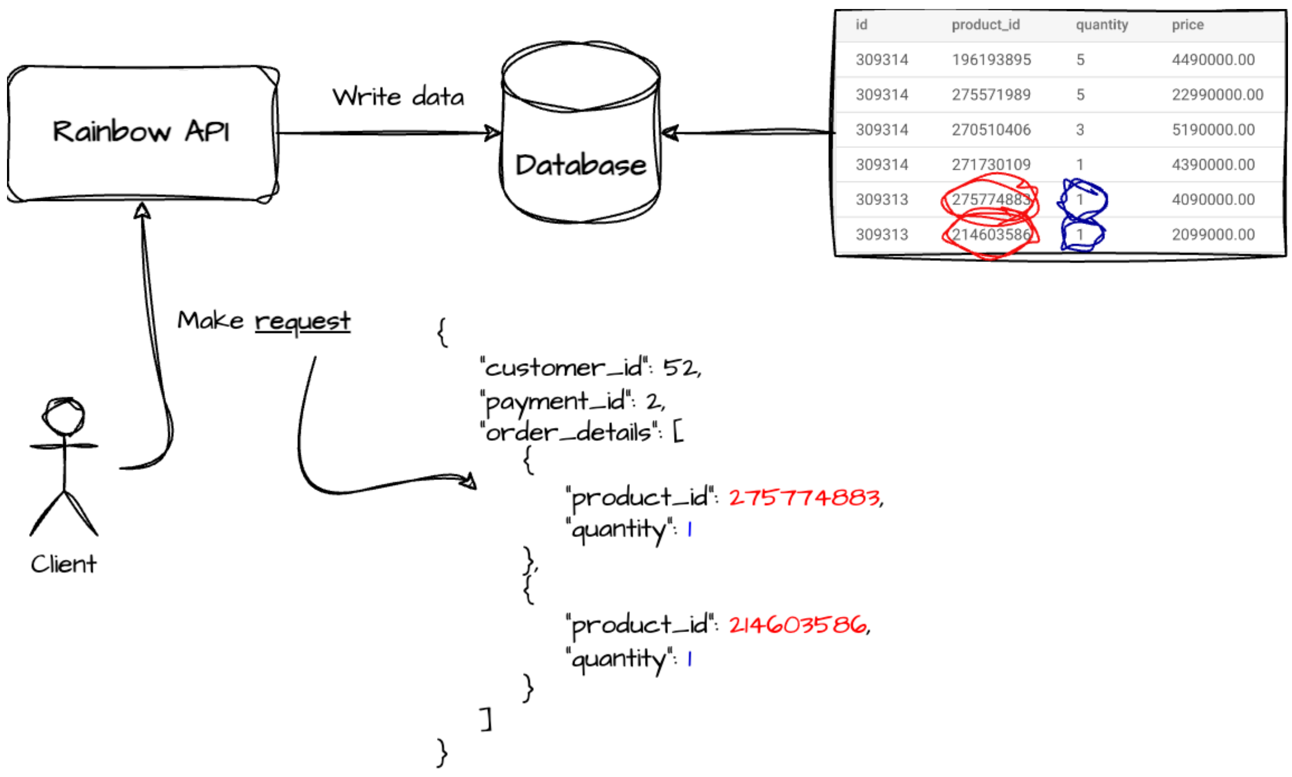
Architecture

- The architecture of the project is designed based on the **Lambda Architecture** pattern. There are multiple components in this architecture as we will discuss later.



Rainbow API

- This is our application that reflects the business process of an e-commerce platform. I assume that each time a customer places an order, the order data will be generated and stored in the database. This application is an API written in Golang that provides endpoints for managing orders.



id	product_id	quantity	price
309314	196193895	5	4490000.00
309314	275571989	5	22990000.00
309314	270510406	3	5190000.00
309314	271730109	1	4390000.00
309313	275774883	1	4090000.00
309313	214603586	1	2099000.00

```
{
    "customer_id": 52,
    "payment_id": 2,
    "order_details": [
        {
            "product_id": 275774883,
            "quantity": 1
        },
        {
            "product_id": 214603586,
            "quantity": 1
        }
    ]
}
```

- The order data includes the ID of the customer, the payment method used, and the list of products in the order. The API allows users to retrieve a list of all orders, retrieve an order by its ID, and create a new order.

Routes

1. **Base Routes**

   `GET /ping`

   - **Description**: Health check endpoint.
   - **Response**:
     - **200 OK**: The service is running.
     - **500 Internal Server Error**: Server Error.

   `GET /`

   - **Description**: Information about the service.
   - **Response**:
     - **200 OK**: Service information.
     - **500 Internal Server Error**: Server Error.

   ---

2. **Order Routes**

   `GET /api/v1/orders/:id`

   - **Description**: Retrieve an order by its ID.
   - **Parameters**:
     - `id`: The ID of the order.
   - **Response**:

    - **200 OK**: Returns the order details.
    - **404 Not Found**: Order not found.
    - **500 Internal Server Error**: Server Error.

`GET /api/v1/orders`

  - **Description**: Retrieve a list of all orders.
  - **Response**:
    - **200 OK**: Returns a list of orders.
    - **500 Internal Server Error**: Server Error.

`POST /api/v1/orders`

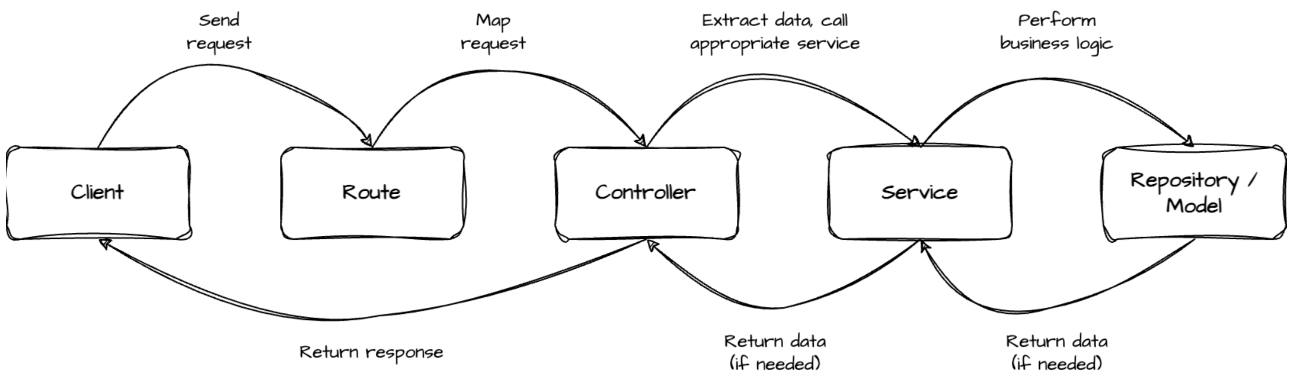
  - **Description**: Create a new order.
  - **Request Body**:
    - `customerId`: The ID of the customer placing the order.
    - `payment_id`: The ID of the payment method used.
    - `products`: List of products in the order, each containing:
      - `productId` (int): The ID of the product.
      - `quantity` (int): The quantity of the product.
  - **Response**:
    - **201 Created**: Order created successfully.
    - **400 Bad Request**: Invalid request body.
    - **500 Internal Server**: Server Error.

API Request Flow

- Some definitions when I mention about the API request flow:

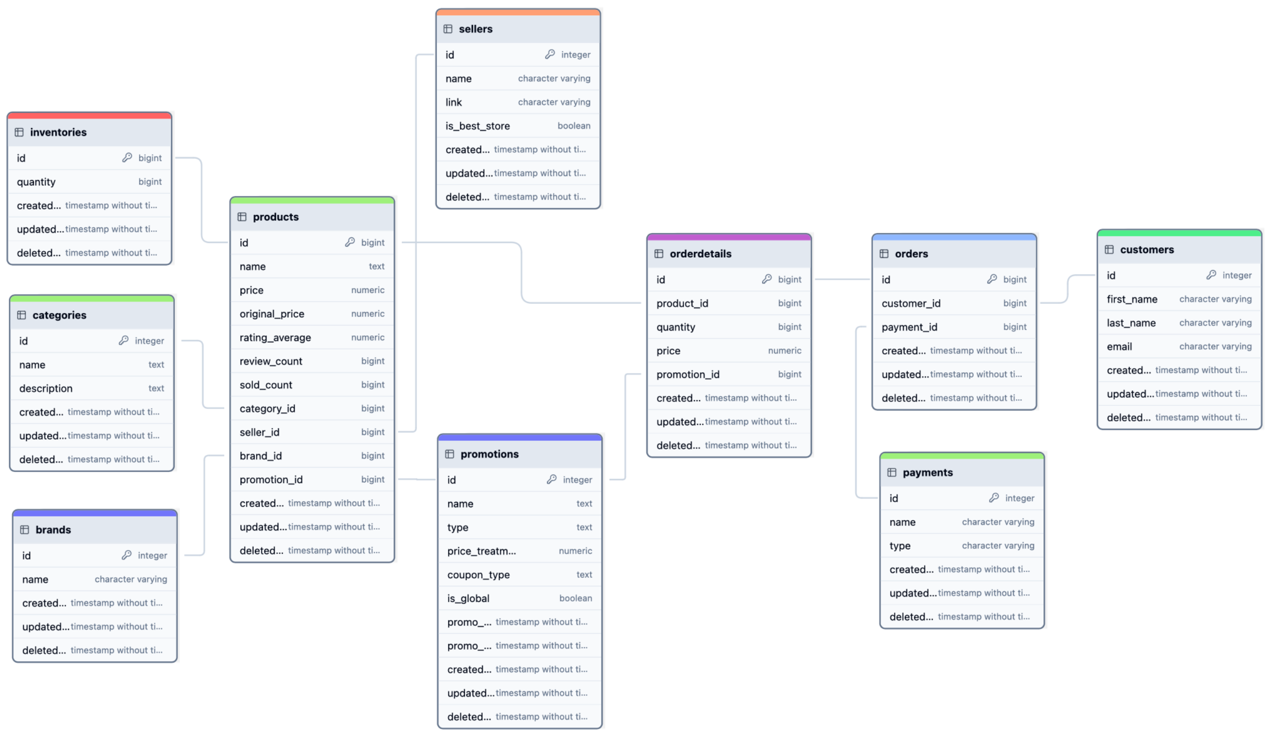
  - **Routes**: The endpoints of the API and maps HTTP requests to corresponding controller actions.
  - **Controllers**: Controllers handles HTTP requests, extracts necessary data, and calls appropriate service methods.
  - **Services**: Services contain the business logic of your application. It performs operations, calls the repository for data access.
  - **Repositories**: Repositories are responsible for data access. It performs CRUD operations on the database.
  - **Models**: Models represent the data in the application.
  - **Unit of Work**: Unit of Work is keeps track of all the changes made to the database and commits them as a single transaction.

- When the request is being made to the API, it will go through the following steps:



Database

- This project uses a PostgreSQL relational database to store the data. The database schema is designed to store the order data, customer data, product data, and other relevant data.


sellers
id integer
name character varying
link character varying
is_best_store boolean
created... timestamp without ti...
updated... timestamp without ti...
deleted... timestamp without ti...
inventories
id bigint
quantity bigint
created... timestamp without ti...
updated... timestamp without ti...
deleted... timestamp without ti...
products
id bigint
name text
price numeric
original_price numeric
rating_average numeric
review_count bigint
sold_count bigint
category_id bigint
seller_id bigint
brand_id bigint
promotion_id bigint
created... timestamp without ti...
updated... timestamp without ti...
deleted... timestamp without ti...
categories
id integer
name text
description text
created... timestamp without ti...
updated... timestamp without ti...
deleted... timestamp without ti...
brands
id integer
name character varying
created... timestamp without ti...
updated... timestamp without ti...
deleted... timestamp without ti...
promotions
id integer
name text
type text
price_treatm... numeric
coupon_type text
is_global boolean
promo_... timestamp without ti...
promo_... timestamp without ti...
created... timestamp without ti...
updated... timestamp without ti...
deleted... timestamp without ti...
orderdetails
id bigint
product_id bigint
quantity bigint
price numeric
promotion_id bigint
created... timestamp without ti...
updated... timestamp without ti...
deleted... timestamp without ti...
orders
id bigint
customer_id bigint
payment_id bigint
created... timestamp without ti...
updated... timestamp without ti...
deleted... timestamp without ti...
customers
id integer
first_name character varying
last_name character varying
email character varying
created... timestamp without ti...
updated... timestamp without ti...
deleted... timestamp without ti...
payments
id integer
name character varying
type character varying
created... timestamp without ti...
updated... timestamp without ti...
deleted... timestamp without ti...

Data Warehouse

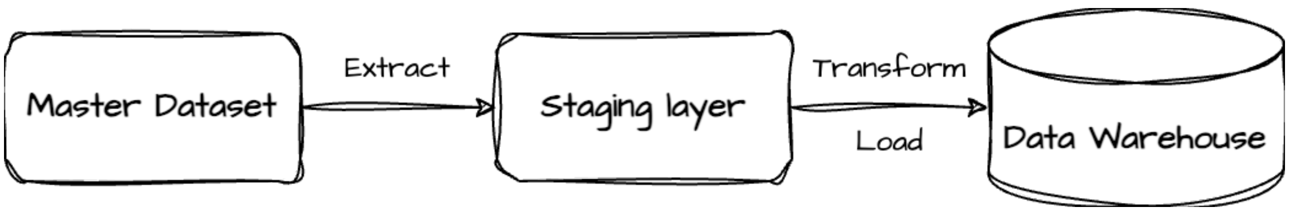
- The amount of data of the system is constantly increasing day by day, so it is necessary to have a data warehouse to store and analyze the data.

> ***4 Steps Dimensional Design Process ?***
>
> - **Step 1:** Select the business process.
> - **Step 2:** Declare the grain.
> - **Step 3:** Identity the facts.
> - **Step 4:** Identity the dimensions.

- Our goal is to perform sales analytics using our transactional data. To achieve this, we have built a data warehouse that centralizes the sales data along with other relevant information. For this project, I am using Google BigQuery as the data warehouse solution.

Pipeline Overview

- We built a data pipeline to extract data from the source, perform transformations, and load it into the data warehouse. Finally, we generate batch views to serve the needs of the Batch layer in the Lambda Architecture.



Data Source (Master Dataset)

- This is where the raw data originates which we define it as our **Master Dataset**. This dataset is immutable and append-only, meaning that once data is written, it cannot be changed or deleted.

- We can consider this layer as a backup of the data. If something goes wrong in the data warehouse, we can always go back to the master dataset and reprocess the data.

Staging Area

- Data from the sources is first loaded into a staging area. The schema in the staging layer is the same as the schema in the Master Dataset.

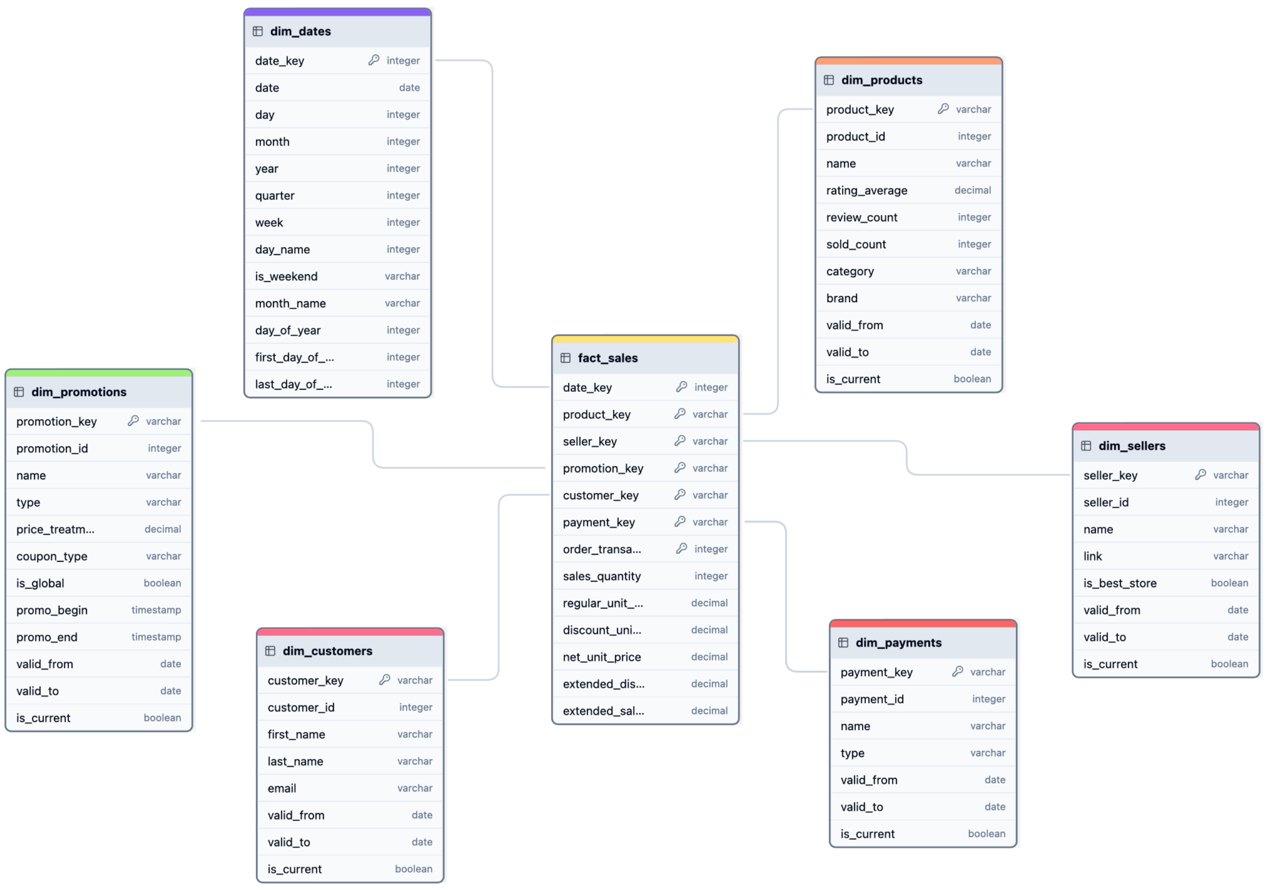
- I find some advantages of having a staging area:

    - This layer allows us to perform data consolidation and data cleaning before loading the data into the data warehouse. E.g: We can do renaming columns, converting data types to make data consistent between different sources.
    - If there are many pipelines run on different schedules, the staging area can be used to store the data temporarily before loading it into the data warehouse.

Data Warehouse Layer

- Our target is to perform sales analytics with our transactional data, we categorize the data into star schema in data warehouse.

    > ***NOTE:*** As our amount of data is not large, I allow the data warehouse to have data redundancy to improve query performance. Therefore, I choose to use the **star schema** instead of the snowflake schema.



- The **star schema** involves those components:

    1. **Fact table**:

        - Our fact table **`fact_sales`** contains the quantitative data related to sales transactions of our business process. Our quantitative data or facts are sales data contain quantity of products sold (`sales_quantity`), unit and extended total amount (`regular_unit_price`, `net_unit_price`, `extended_sales_amount`), discount amount (`discount_unit_price`, `extended_discount_amount`). The fact table also contains the foreign keys to the dimension tables.
        - Granularity: Each row in a fact table is a record of an item sold.

    2. **Dimension table**:

        - Our dimension tables contain the descriptive data related to the our business process. The dimension tables are used to filter, group, or aggregate the data in the fact table.
        - Dimension tables: `dim_dates`, `dim_customers`, `dim_products`, `dim_sellers`, `dim_promotions`, `dim_payments`.

Slowly Changing Dimensions (SCDs)

- We use the Slowly Changing Dimensions (SCDs) technique to handle changes in dimension data over time.

- In this project, I implement the SCD type 2 where a change occurs in a dimension record results in a new row added to the table to capture the new data, and the old record remains unchanged and also is marked as expired.

- For SCD Type 2, I need to include three more attributes `valid_from`, `valid_to` and `is_current` as shown below. The newest version of record will have the column `valid_to`'s value `2100-01-01` and also the column `is_current`'s value `true`.

Product Key	Product ID	Name	Rating Average	Review Count	Sold Count	Category	Brand	Valid From	Valid To	Is Current
f054b7cb90f8fcc7109bf09ab41fc586	193366048	Điện thoại OnePlus 10T 5G - Hàng Chính Hãng	0.0	0	9637	Điện thoại smartphone	OnePlus	2024-09-08	2024-09-08	false
f3456923-0aa1-4d2b-9fa3-4f6c12aa3adc	193366048	Điện thoại OnePlus 10T 5G - Hàng Chính Hãng	0.0	10	9637	Điện thoại smartphone	OnePlus	2024-09-08	2100-01-01	true

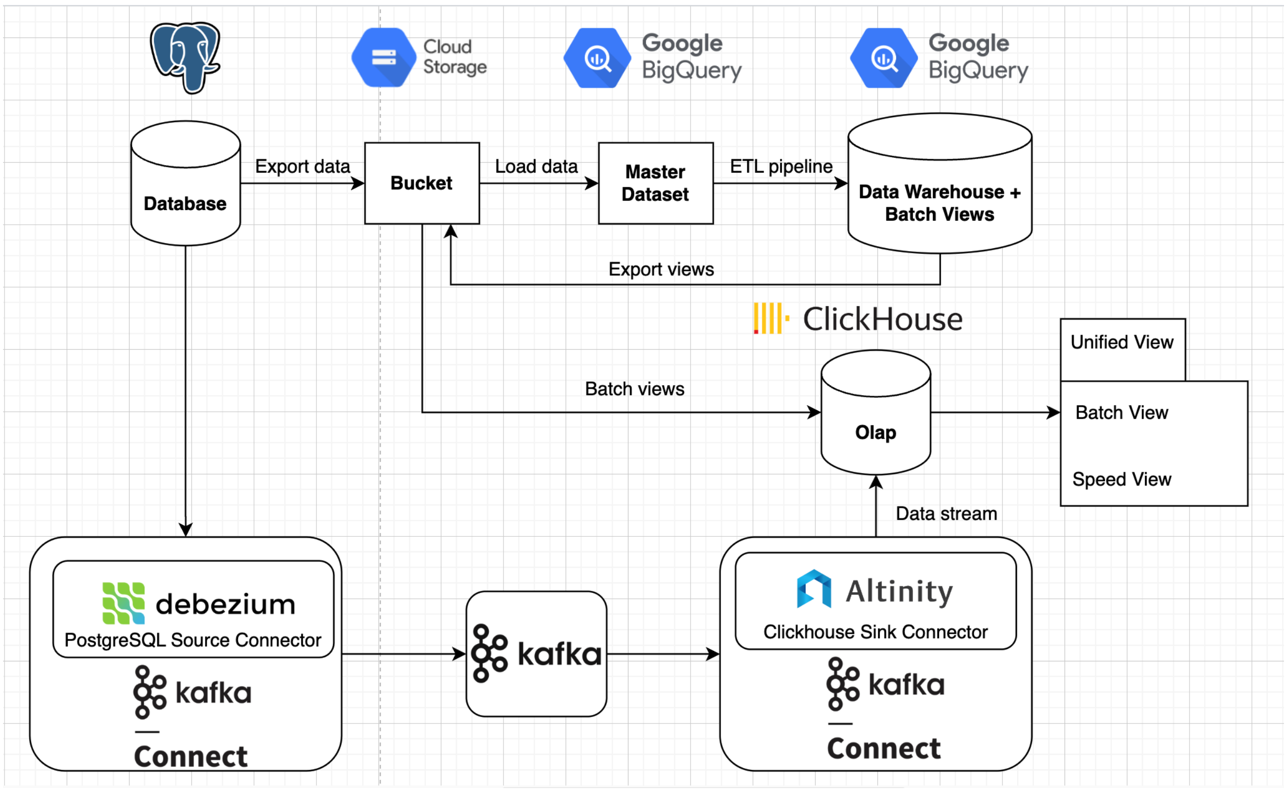
ETL Logic

Access Layer

- With data stored in data warehouse, we can connect to visualization tools or applications for building charts, dashboards.

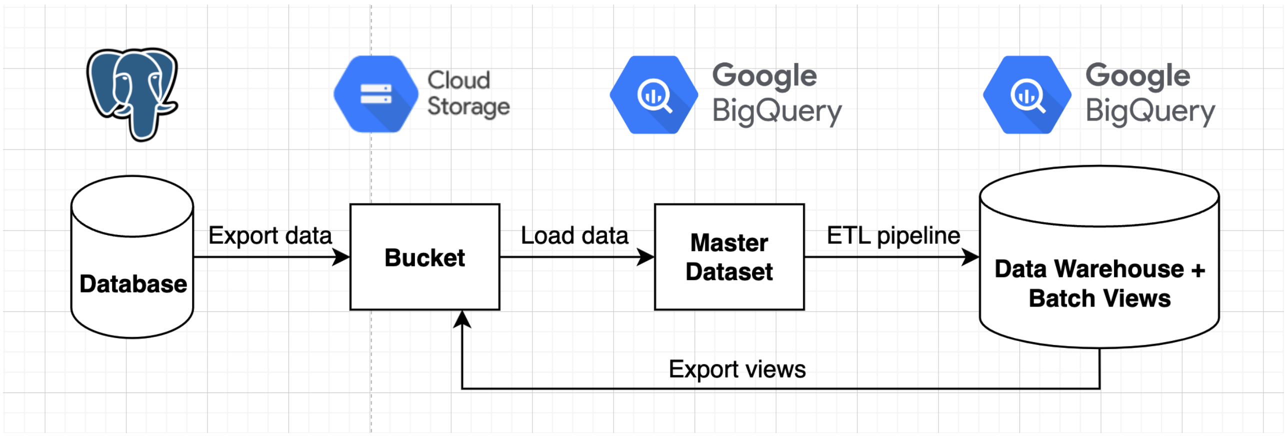
Lambda Architecture

- Lambda Architecture is a data processing architecture that takes advantages of both batch and streaming processing methods.

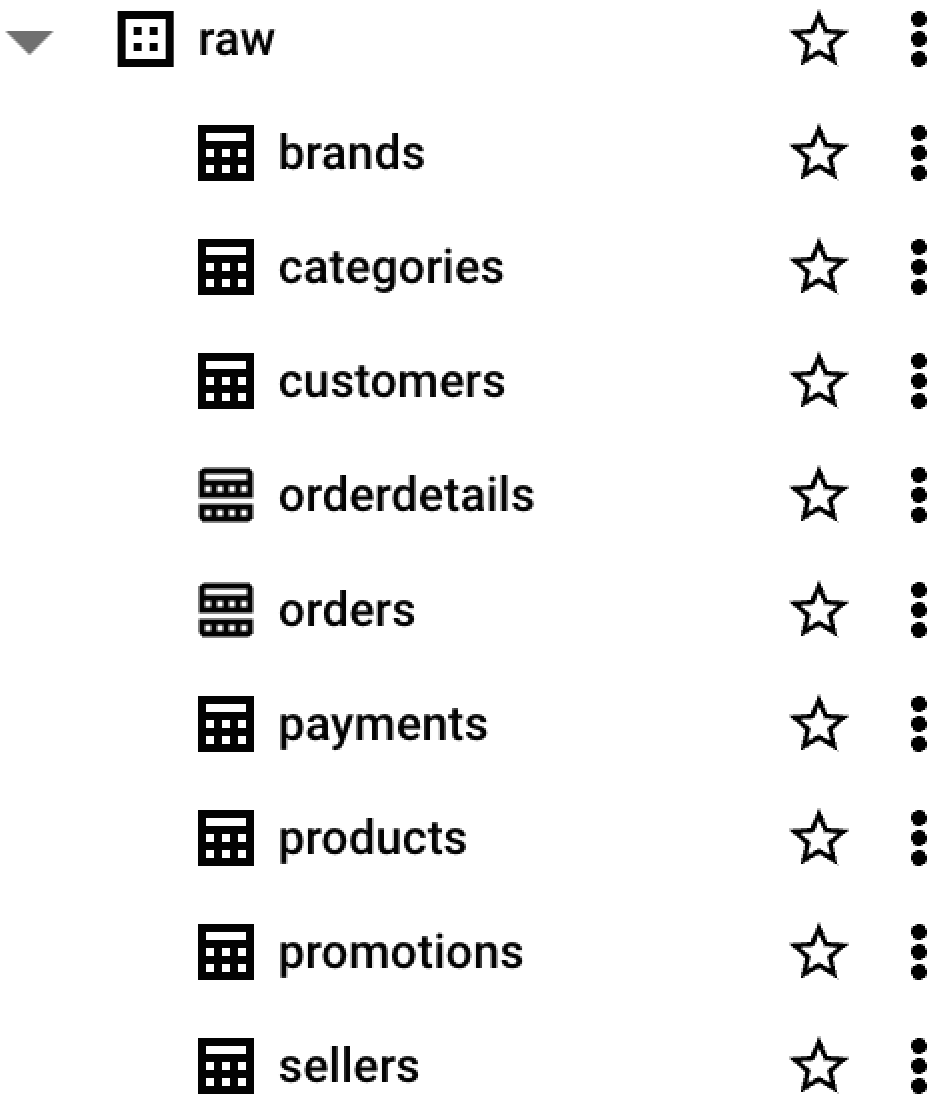


Batch Layer

- Batch layer is responsible for 2 purposes: **Handling the historical data** and **Generating batch views** of precomputed results.

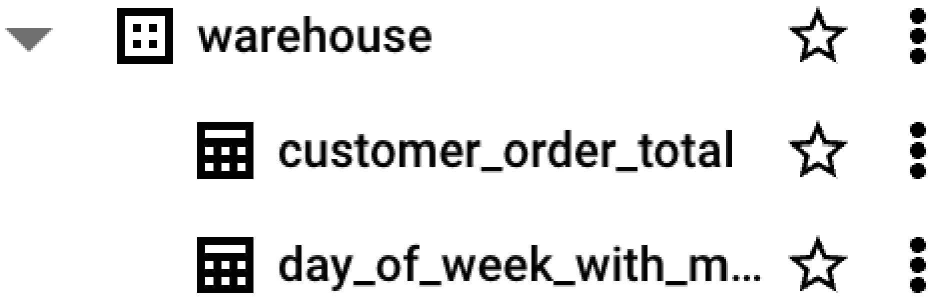


- It manages the master dataset where the data is immutable and append-only, preserving a trusted historical records of all the incoming data from source. I created a dataset `raw` for storing master dataset.



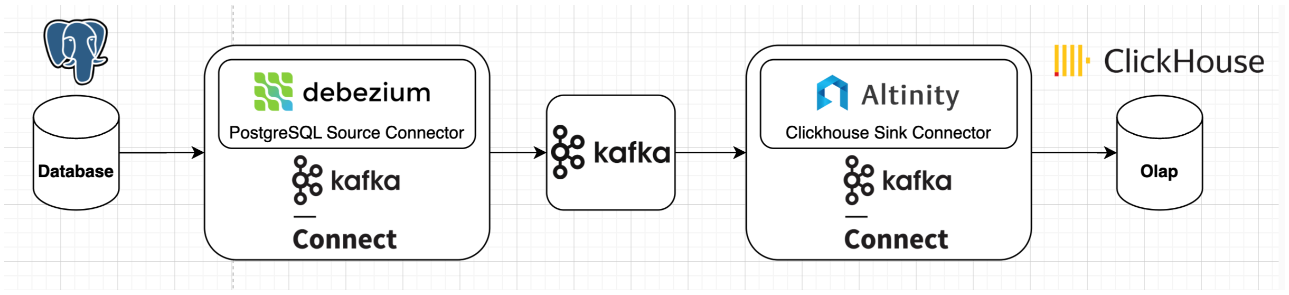
- raw
  - brands
  - categories
  - customers
  - orderdetails
  - orders
  - payments
  - products
  - promotions
  - sellers

- Batch views are precomputed using BigQuery with SQL scripts. They are stored in a dataset `warehouse` and exported into flat files where OLAP Database (Clickhouse) can ingest from this.



Speed Layer

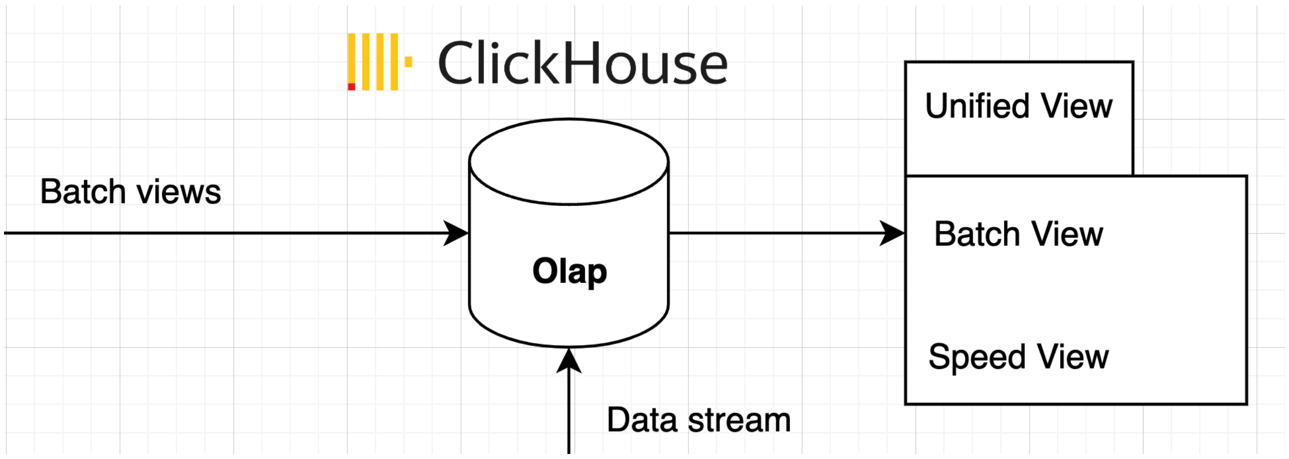
- By design, the batch layer has a high latency, typically delivering batch views to the serving layer at a rate of once or twice per day. Speed layer handles realtime data streams and provides up-to-date views.



- In this project, the stream flows are just to deliver data from source (**Rainbow Database**) to our sink (**OLAP Database**) in near realtime without processing. Data after being transmitted to the destination will be aggregated and formed speed views.
- I am using Kafka Connect to establish data streams from the source to sink by configuring two plugins `io.debezium.connector.postgresql.PostgresConnector` and `com.clickhouse.kafka.connect.ClickHouseSinkConnector`.

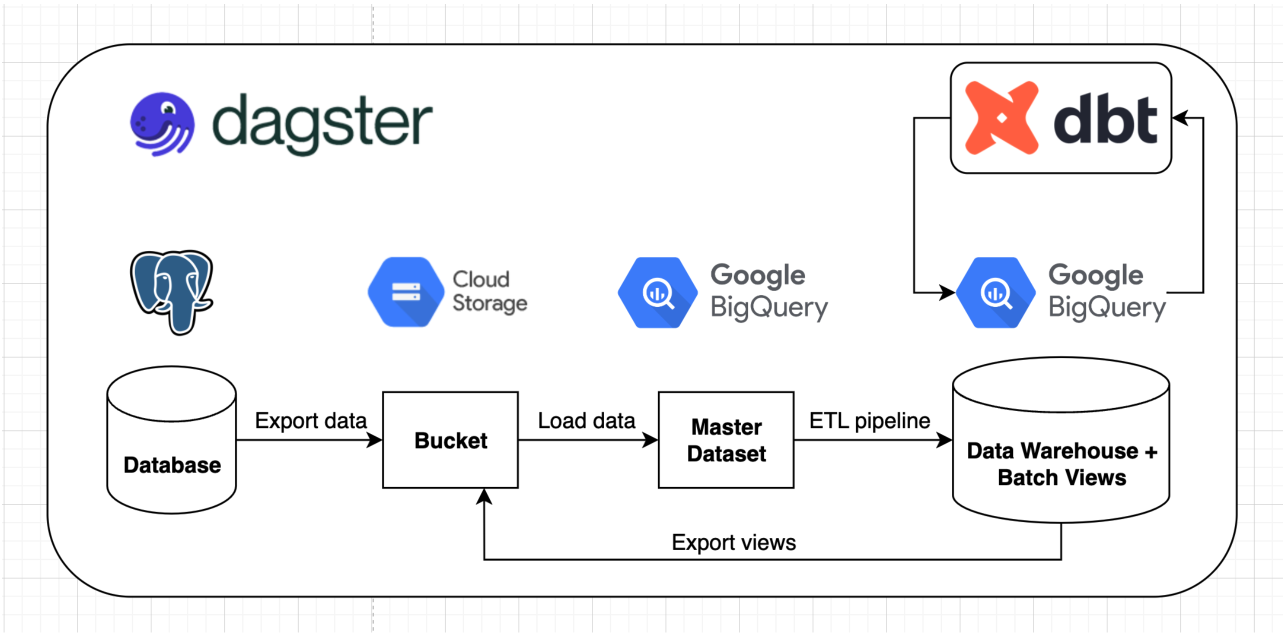
Serving Layer

- The data serving layer receives the batch views from the batch layer and also receives the near real-time views streaming in from the speed layer.
- Serving layer merges results from those two layer into final unified views.



Data Pipeline

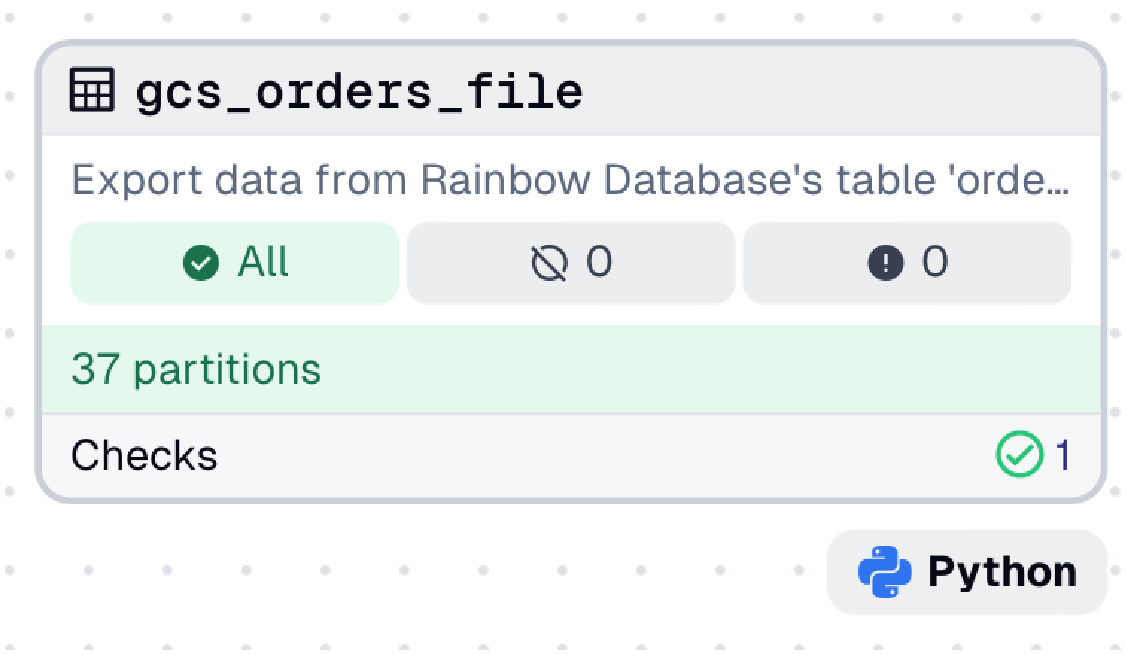
- Our data pipeline mainly focus on moving, transforming and organizing data to serve the need of Batch layer in the lambda architecture.
- For orchestrating, scheduling the pipeline in an automatically way, I am using open source tool called **Dagster**. Moreover, **dbt** is used for transformation steps in our data pipeline.



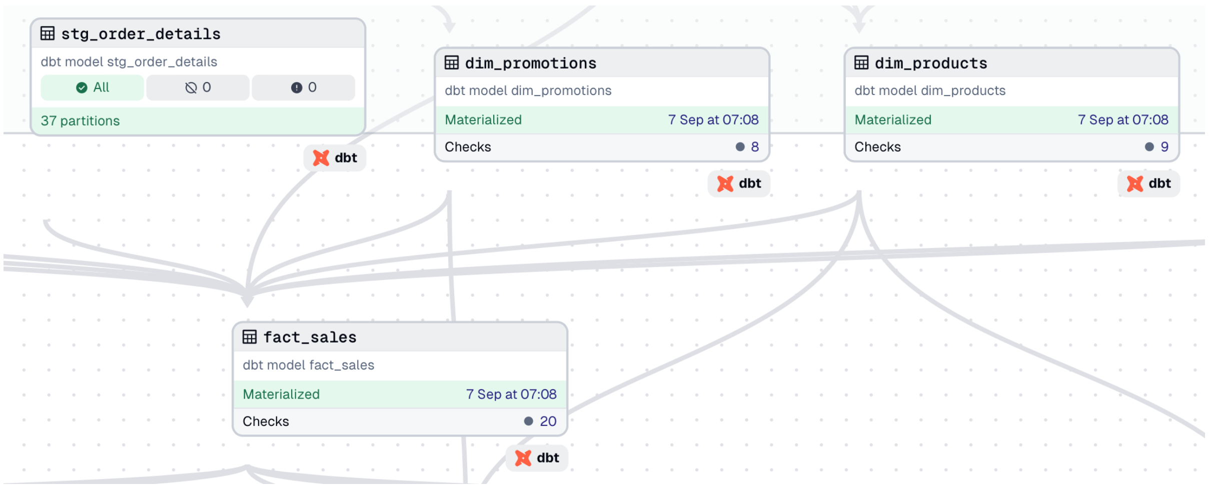
Dagster

- Dagster models pipelines in terms of the data assets. With Dagster, you declare functions in Python that you want to run and the data assets that those functions produce or update.

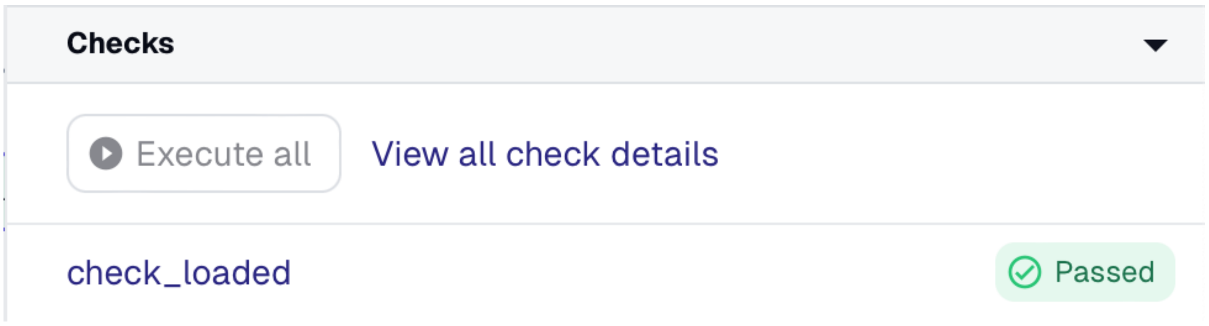
1. **Assets**
    - Dagster is asset-centric, which declare that the primary focus is on the ***data products*** **(assets)** generated by the pipeline. An **asset** is an object in persistent storage, such as a table, file. In this project, I defined many assets, each asset is responsible for a result of one step.



- For example, the asset `gcs_orders_file` is responsible for extracting data of table `orders` from Rainbow Database and saving as a parquet file in Google Cloud Storage. Or the asset `fact_sales` is the fact table in our data warehouse, it is created by aggregating and transforming data from staging and dimension tables.



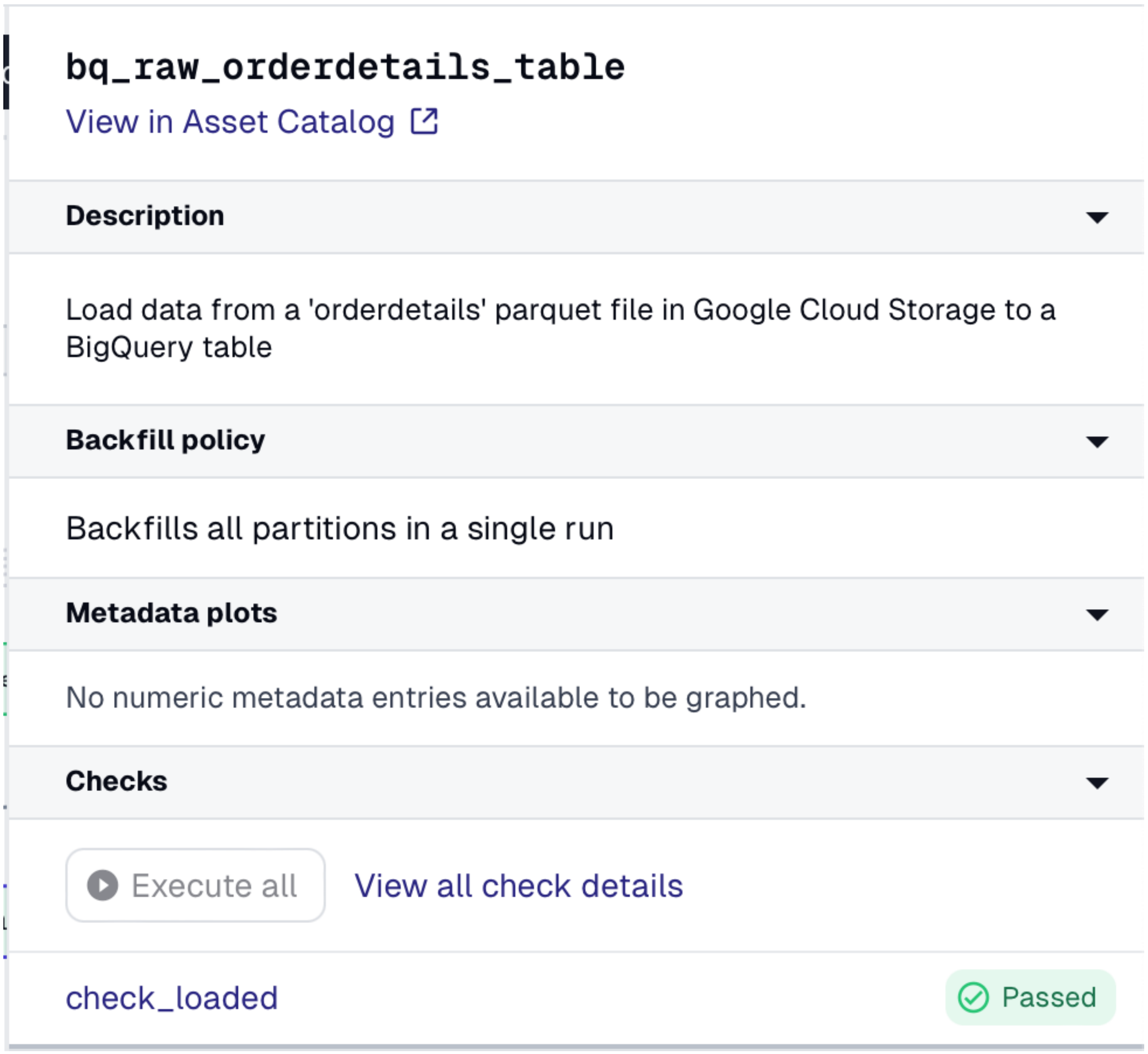
- Dagster automatically tracks asset dependencies and it allows you to track the current state of your assets. Assume if upstream assets are materialized but the downstream assets, it will mark the state of downstream assets as `Out of sync`.
- Dagster provides **Asset Checks** to define and execute different types of checks on your data assets directly in Dagster. This help us to perform some kind of data quality tests or data existing checks.


Checks

Execute all View all check details

check_loaded Passed

- Dagster also allow us to provide rich metadata for each assets.


bq_raw_orderdetails_table
View in Asset Catalog

Description

Load data from a 'orderdetails' parquet file in Google Cloud Storage to a BigQuery table

Backfill policy

Backfills all partitions in a single run

Metadata plots

No numeric metadata entries available to be graphed.

Checks

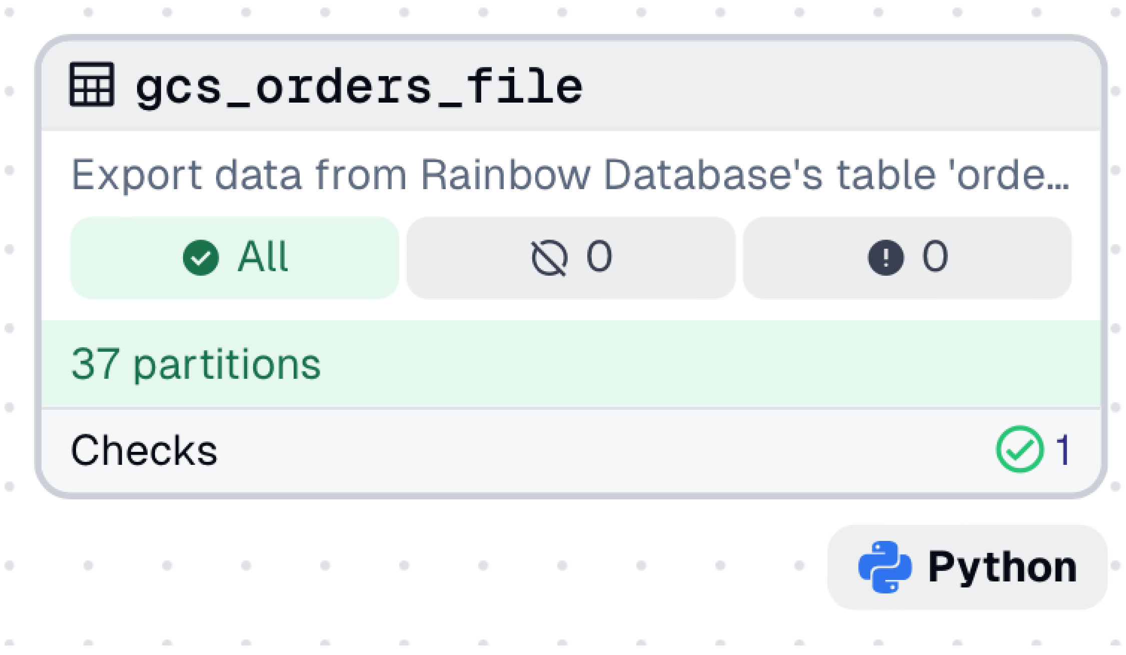
Execute all View all check details

check_loaded Passed

2. **Partitions & Backfill**

    - An asset definition can represent a collection of **partitions** that can be tracked and materialized independently. In other words, you can think each partition functions like its own mini-asset, but they all share a common materialization function and dependencies.

    - For example, I have a `DailyPartition` asset where each partition key represent a date like `2024-09-01`, `2024-02-09`, ...



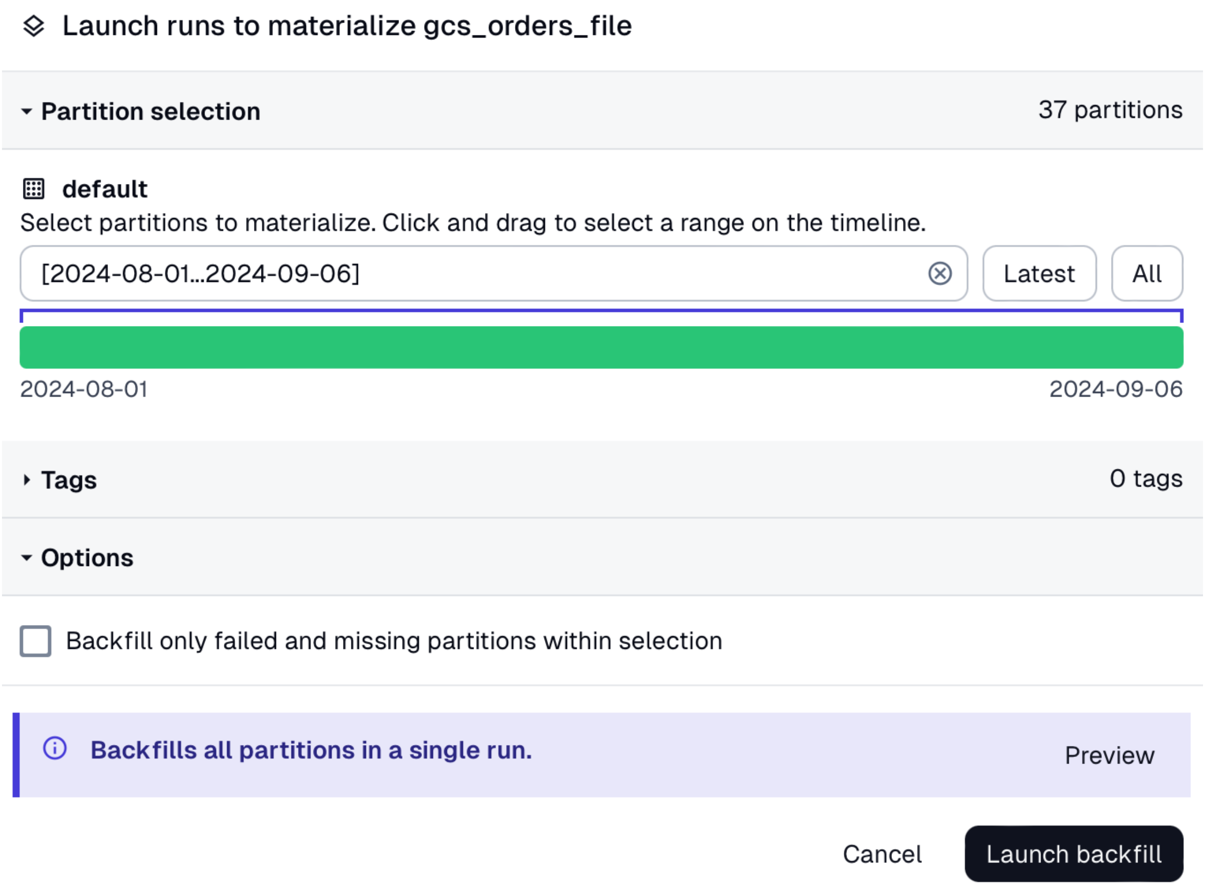
- Using partitions provides the following benefits:

  - **Cost efficiency**: Run only the data that’s needed on specific partitions.
  - **Speed up compute**: Divide large datasets into smaller, allow boosting computational speed with parallel processing.

- **Backfilling** is the process of running partitions for assets or updating existing records. Dagster supports backfills for each partition or a subset of partitions. Backfills are common when setting up a pipeline where:

  - Run the pipeline for the first time where you might want to backfill all historical and current data.
  - Scenario when you changed the logic for an asset and need to update historical data with the new logic.

- In the project, this backfill job will materialize on all 37 partitions.


Launch runs to materialize gcs_orders_file

Partition selection — 37 partitions

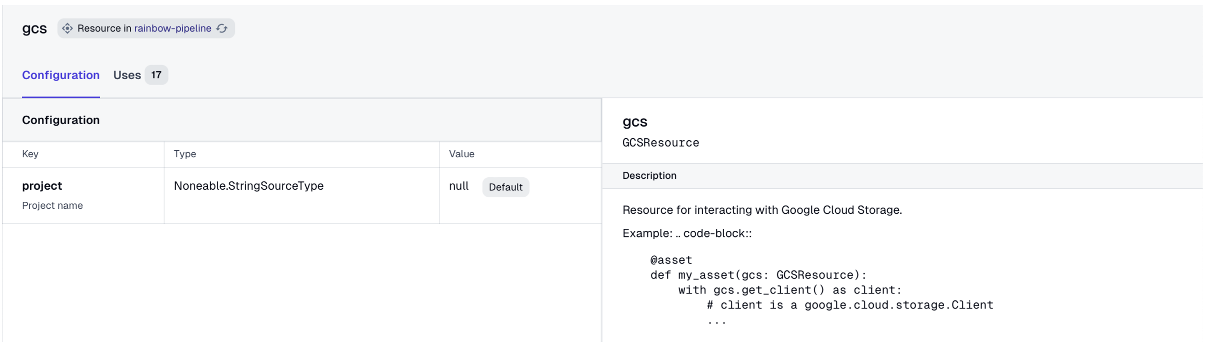
default
Select partitions to materialize. Click and drag to select a range on the timeline.
[2024-08-01...2024-09-06]
2024-08-01 — 2024-09-06

Tags — 0 tags

Options
☐ Backfill only failed and missing partitions within selection

Backfills all partitions in a single run.

3. **Resources**
   - In data engineering, resources are the external services, tools, and storage you use to do your job.
   - In this project, the pipeline often interacts with Rainbow Database, Google Cloud Storage, BigQuery. A `gcs` resource is described as below.


gcs — Resource in rainbow-pipeline

Configuration

Key	Type	Value
project Project name	Noneable.StringSourceType	null Default

gcs
GCSResource

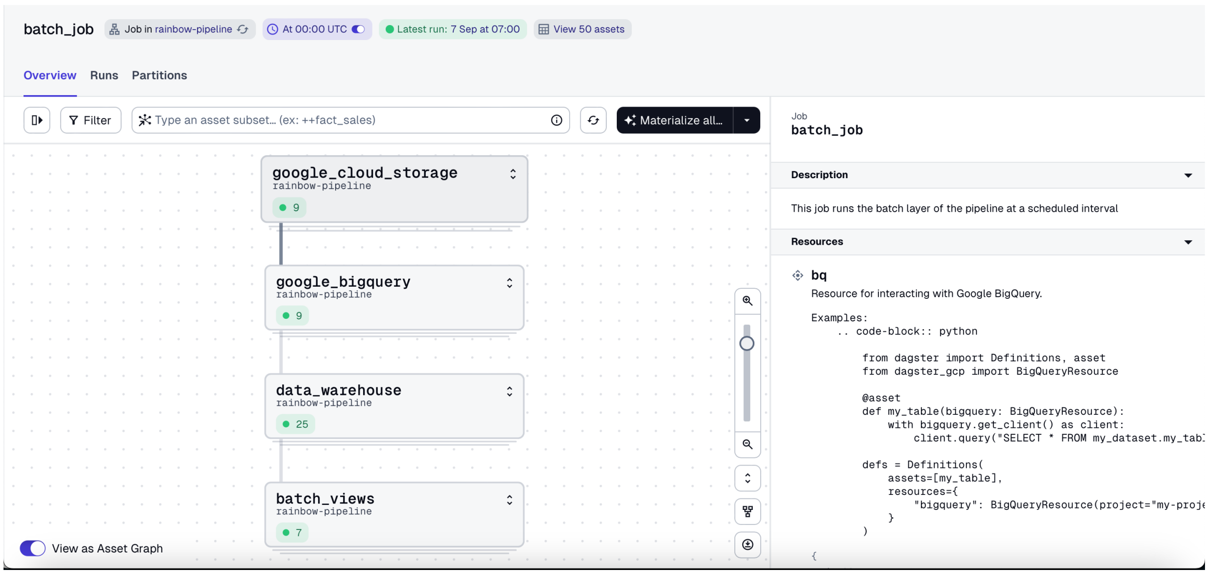
Description

Resource for interacting with Google Cloud Storage.
Example: .. code-block::

```
@asset
def my_asset(gcs: GCSResource):
    with gcs.get_client() as client:
        # client is a google.cloud.storage.Client
        ...
```

   - The concept of **resource** is similar to the concept of **connection** in Airflow. I observe that the two biggest advantages of the resource in Dagster are the reusability in code and resource dependency in asset.

4. **Assets Job**
   - Jobs are the main unit for executing and monitoring asset definitions in Dagster. An asset job is a type of job that targets a selection of assets and can be launched.
   - In this project, I define an asset job called `batch_job` with the purpose of running the whole pipeline in batch layer. This job is launched at fixed interval, by using **schedules**.



Job
batch_job

Description

This job runs the batch layer of the pipeline at a scheduled interval

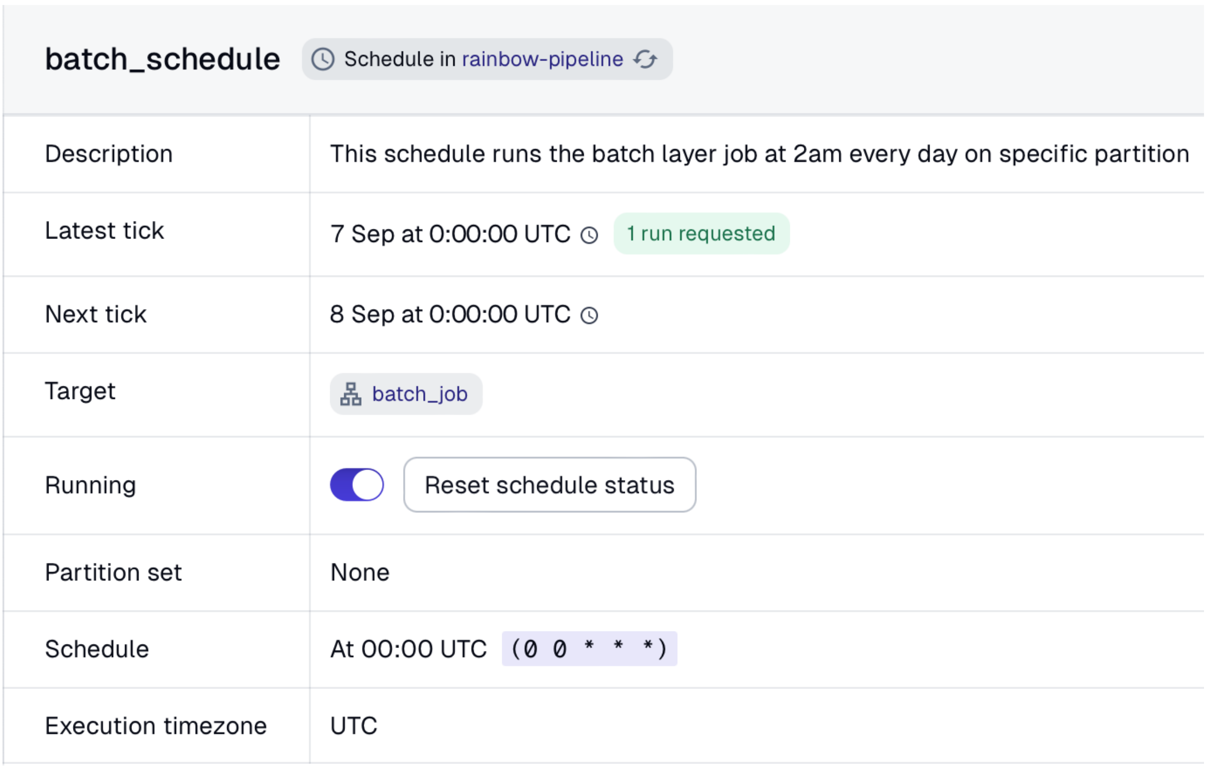
Resources

bq
Resource for interacting with Google BigQuery.

```
Examples:
    .. code-block:: python

        from dagster import Definitions, asset
        from dagster_gcp import BigQueryResource

        @asset
        def my_table(bigquery: BigQueryResource):
            with bigquery.get_client() as client:
                client.query("SELECT * FROM my_dataset.my_tab

        defs = Definitions(
            assets=[my_table],
            resources={
                "bigquery": BigQueryResource(project="my-proj
            }
        )

{
```

5. **Schedules**

   - Schedules are Dagster's way of supporting traditional methods of automation, which allow you to specify when a job should run. Using schedules in this project, the whole pipeline is set up to run at a fixed time at **0:00 AM UTC** every day.



batch_schedule   Schedule in rainbow-pipeline

Description	This schedule runs the batch layer job at 2am every day on specific partition
Latest tick	7 Sep at 0:00:00 UTC 1 run requested
Next tick	8 Sep at 0:00:00 UTC
Target	batch_job
Running	Reset schedule status
Partition set	None
Schedule	At 00:00 UTC `(0 0 * * *)`
Execution timezone	UTC

   - In this project, partitioned assets will be run on the latest partition key. In other words, if I have `DailyPartition` asset, each day the pipeline will run on the partition key of the previous date. For example, if the current date is `2024-09-02`, then the pipeline will be provided a partition key of the previous date `2024-09-01` when it starts to run.

Dbt

- Our project consists of ETL pipeline where we extract data from Rainbow Database, transform and load them to the Data Warehouse. The letters **E** and **L** are the process of moving data between two places in which we can easily achieve this by writing script and defining them as assets in Dagster.

- The remaining letter **T** represents for the transformation step. To achieve this step, I decide to transform using SQL script and dbt is the most suitable to this situation.
- Dbt is **a data transformation tool** that enables data analysts and engineers to transform data in a cloud analytics warehouse.

1. **Model**

   - Dbt model is essentially a SQL file that contains a transformation query. This query decides how data is transformed from its raw form to a usable form in the data warehouse.

   - The SQL query is saved as a view or table in the data warehouse. This means that we don't need to care about scripts to create a view or a table and instead focus on the transformation logic only.

   - For example, I have a dbt model `dim_products` which represents the transformation logic of creating a dimension table for product data. Every time this dbt model runs, the new data is inserted incrementally to the table `dim_products` in our data warehouse by specifying `materialized='incremental'`.

     ```
     {{
          config(
              materialized='incremental',
              unique_key='product_key',
              incremental_strategy='merge'
          )
      }}

      WITH latest_version AS (
          SELECT
              id,
              name,
              rating_average,
              review_count,
              sold_count,
              category_id,
              brand_id,
              updated_at,
              ROW_NUMBER() OVER (PARTITION BY id ORDER BY updated_at
     DESC) AS row_num
          FROM {{ ref('stg_products') }}
      ),
      current_products AS (
          SELECT
              ls.id,
              ls.name,
              ls.rating_average,
              ls.review_count,
              ls.sold_count,
              ls.category_id,
              ls.brand_id,
              dc.name AS category,
              db.name AS brand,
     ```

```
            ls.updated_at
        FROM latest_version ls
        JOIN {{ ref('dim_categories') }} AS dc ON ls.category_id =
dc.category_id
        JOIN {{ ref('dim_brands') }} AS db ON ls.brand_id =
db.brand_id
        WHERE row_num = 1
    ),
    existing_products AS (
        SELECT DISTINCT
            dp.product_key,
            dp.product_id,
            dp.name,
            dp.rating_average,
            dp.review_count,
            dp.sold_count,
            dp.category,
            dp.brand,
            dp.valid_from,
            EXTRACT(DATE FROM cp.updated_at) AS valid_to,
            FALSE AS is_current
        FROM {{ this }} AS dp
        JOIN current_products AS cp ON dp.product_id = cp.id
        WHERE dp.is_current = TRUE
        AND (
            dp.name != cp.name OR
            dp.rating_average != cp.rating_average OR
            dp.review_count != cp.review_count OR
            dp.sold_count != cp.sold_count OR
            dp.category != cp.category OR
            dp.brand != cp.brand
        )
    ),
    new_products AS (
        SELECT DISTINCT
            cp.id AS product_id,
            cp.name AS name,
            cp.rating_average AS rating_average,
            cp.review_count AS review_count,
            CAST(cp.sold_count AS INT64) AS sold_count,
            cp.category AS category,
            cp.brand AS brand,
            EXTRACT(DATE FROM cp.updated_at) AS valid_from,
            DATE '2100-01-01' AS valid_to,
            TRUE AS is_current
        FROM current_products AS cp
        LEFT JOIN {{ this}} AS dp ON cp.id = dp.product_id
        WHERE dp.product_id IS NULL
        OR (
            dp.is_current = TRUE AND
            (
                dp.name != cp.name OR
                dp.rating_average != cp.rating_average OR
                dp.review_count != cp.review_count OR
                dp.sold_count != cp.sold_count OR
                dp.category != cp.category OR
                dp.brand != cp.brand
            )
        )
    )
    SELECT
        *
    FROM existing_products
    UNION ALL
    SELECT
        generate_uuid() AS product_key,
        *
    FROM new_products
```

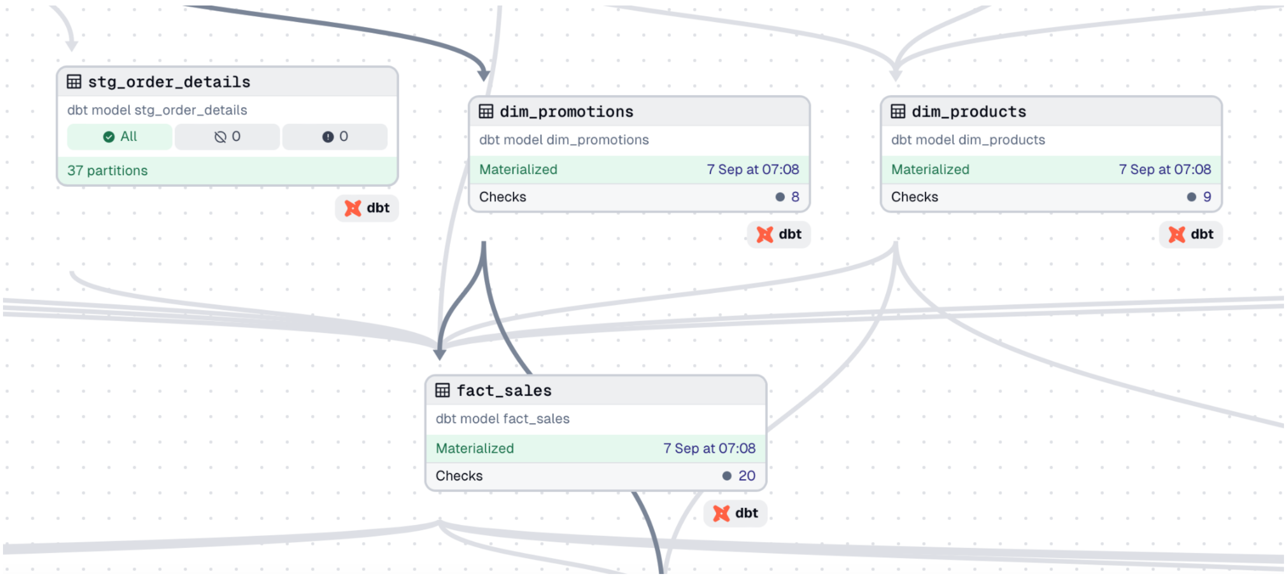
2. **Test**

   - Data tests are assertions you make about your models and other resources in your dbt project. This improve the integrity of the SQL in each model by making assertions about the results generated.
   - Data tests are SQL queries. In particular, they are `select` statements that seek to grab "failing" records, ones that disprove your assertion.
   - For example, I have a dbt test `assert_consistent_total_sales.sql` in which I want to ensure that the total sales value after transforming and loading to data warehouse is consistent to the total sales value in the raw form. This test will fail in case this query returns the count differ than 0.

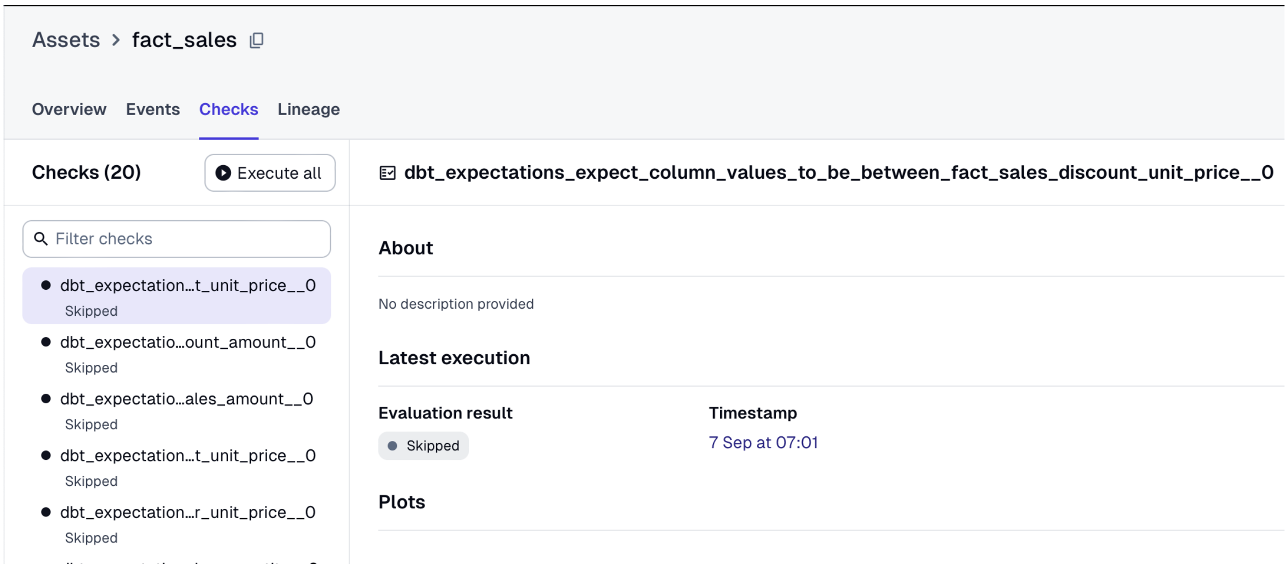
```
WITH revenue_fact_sales AS (
    SELECT
        SUM(extended_sales_amount) AS total_extended_sales_amount
    FROM {{ ref('fact_sales') }}
),
revenue_raw_order_details AS (
    SELECT
        SUM(quantity * price) AS total_extended_sales_amount
    FROM {{ source('rainbow', 'orderdetails') }}
)
SELECT
    COUNT(*)
FROM revenue_fact_sales
JOIN revenue_raw_order_details
    ON revenue_fact_sales.total_extended_sales_amount !=
revenue_raw_order_details.total_extended_sales_amount
```

Dagster & dbt

- You can think of dbt model as an asset in Dagster. When a dbt model runs, it can create a view or table in our data warehouse which we can call it as a Dagster asset. The dependencies between dbt models are also dependencies of corresponding assets in Dagster.



- Dagster can orchestrates, schedules the run of those dbt models. Dagster makes use of dbt tests as asset checks, therefore, it ensure the data quality when integrating them together.



dbt_expectations_expect_column_values_to_be_between_fact_sales_discount_unit_price__0

- dbt_expectation…t_unit_price__0 Skipped
- dbt_expectatio…ount_amount__0 Skipped
- dbt_expectatio…ales_amount__0 Skipped
- dbt_expectation…t_unit_price__0 Skipped
- dbt_expectation…r_unit_price__0 Skipped

About

No description provided

Latest execution

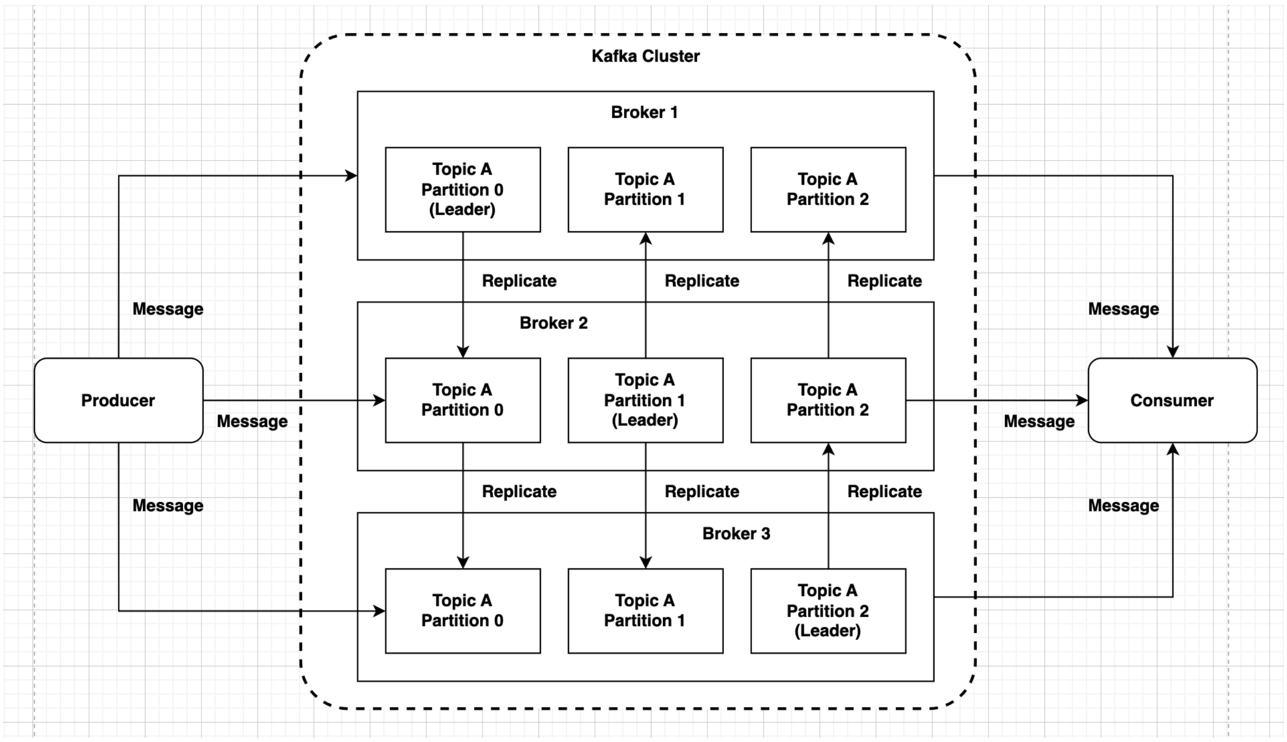
Evaluation result	Timestamp
Skipped	7 Sep at 07:01

Plots

Kafka

Architecture

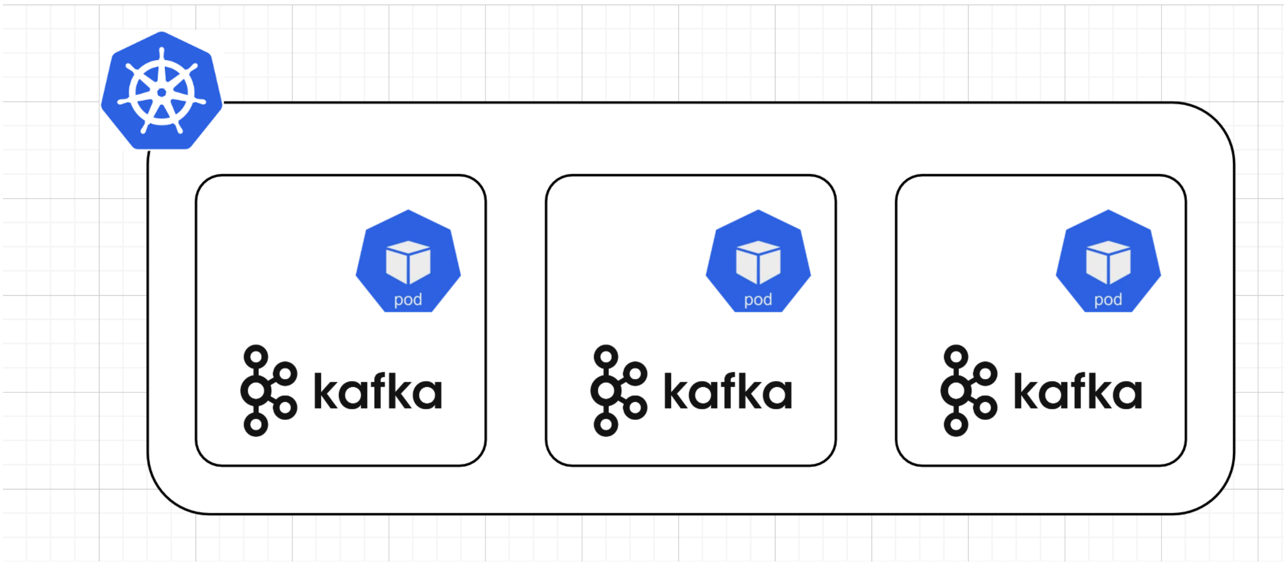
- In this project, I create a single Apache Kafka cluster with 3 brokers. Each topic within this cluster have 3 partitions and 2 replicas. The amount of data is not large, so that this setup is appropriate for serving the need of near realtime streaming.



- **Message** is the Kafka's unit of data like a row or a record of data. **Producer** publish messages into Kafka cluster. **Consumer** then consume those messages by pulling them from the system.
- The published messages are stored in a server called **broker**. The broker receives messages from producers, assigns offsets, and writes them on disk. It also serves consumers by responding to the message fetch requests.
- Messages are organized into **topics**. Each topic is divided into one or many **partitions**. A partition of a topic can be replicated to different brokers, so this means a topic can be scaled horizontally across multiple servers.

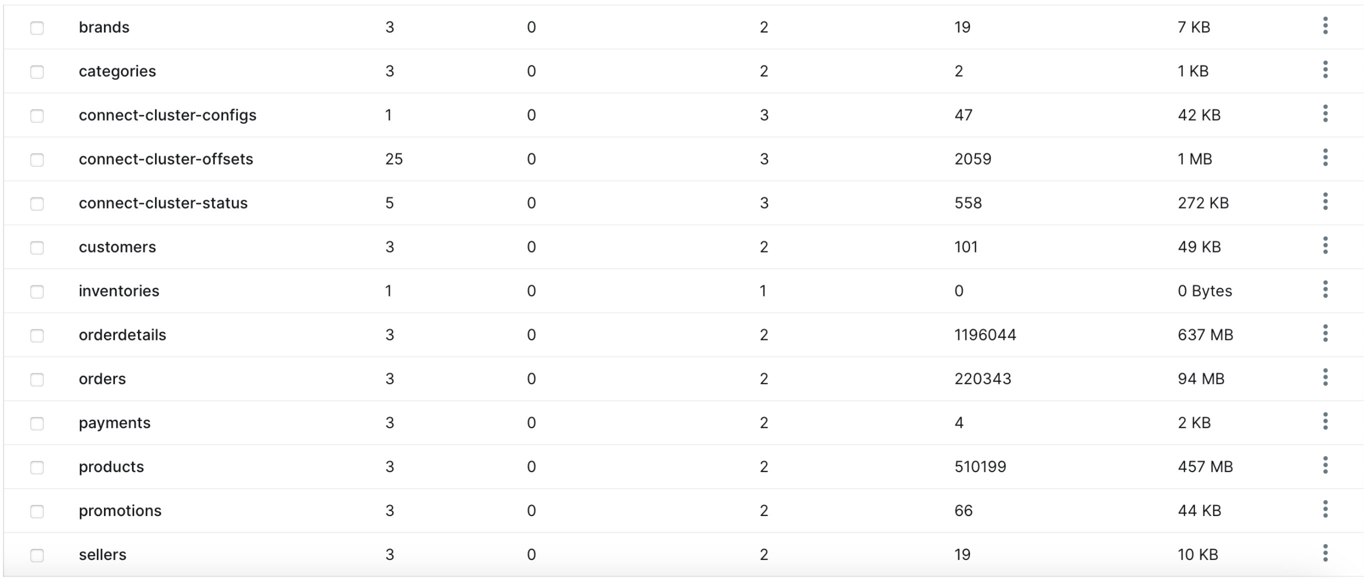
Broker

- Single Kafka cluster is deployed into Kubernetes environment. Each broker lives within one separate Kubernetes pod.



Topic

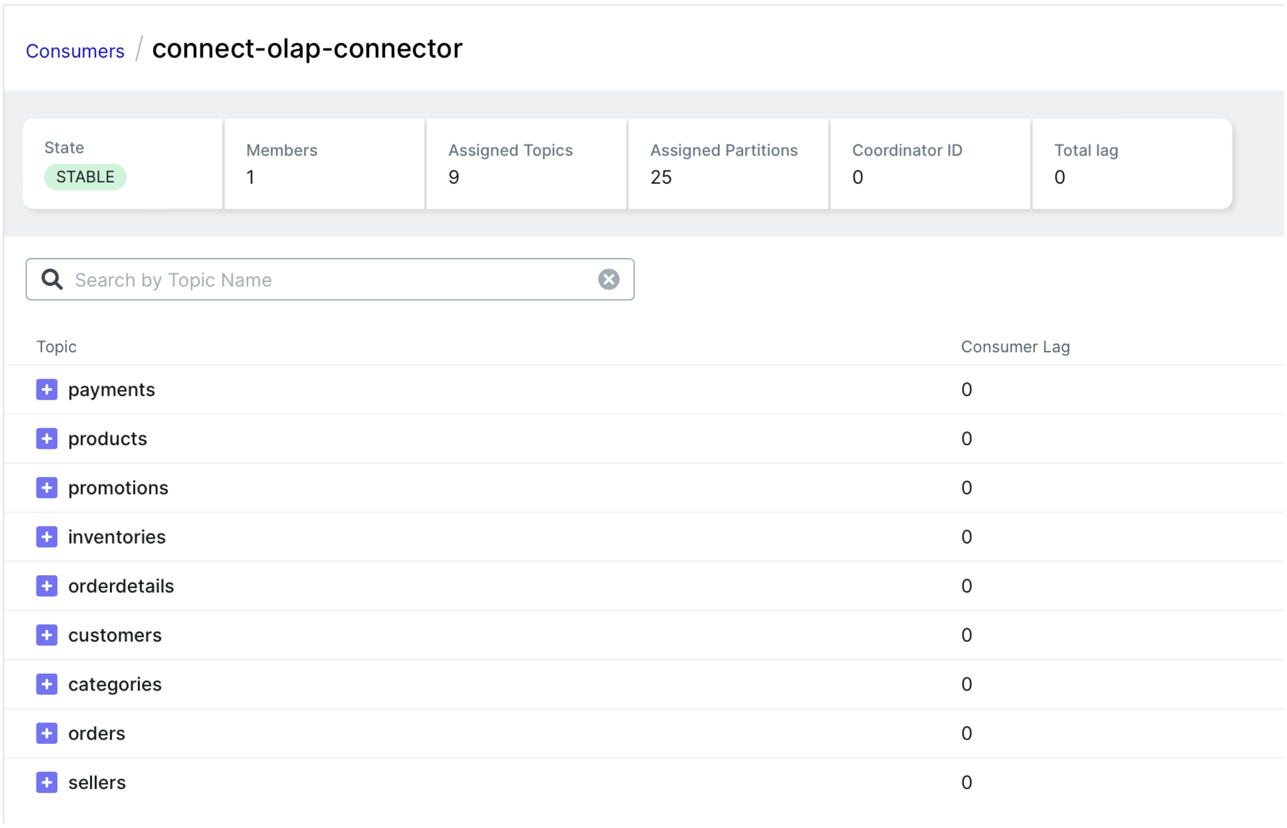
- Messages are organized into topics. For each table in Rainbow Database, it has a corresponding topic. E.g: Kafka will create a topic `products` for capturing data changes from table `products`.



☐	brands	3	0	2	19	7 KB
☐	categories	3	0	2	2	1 KB
☐	connect-cluster-configs	1	0	3	47	42 KB
☐	connect-cluster-offsets	25	0	3	2059	1 MB
☐	connect-cluster-status	5	0	3	558	272 KB
☐	customers	3	0	2	101	49 KB
☐	inventories	1	0	1	0	0 Bytes
☐	orderdetails	3	0	2	1196044	637 MB
☐	orders	3	0	2	220343	94 MB
☐	payments	3	0	2	4	2 KB
☐	products	3	0	2	510199	457 MB
☐	promotions	3	0	2	66	44 KB
☐	sellers	3	0	2	19	10 KB

- **Debezium** acts as the Kafka producer which it capture changes (inserts, updates, deletes) from our Rainbow Database. After capturing these changes, Debezium publishes them as events to Kafka topics.

- The **Clickhouse Sink Connector** serves as the Kafka consumer which it subscribes to the relevant topics and then writes the consumed data into the appropriate tables in ClickHouse.



Consumers / connect-olap-connector

State	Members	Assigned Topics	Assigned Partitions	Coordinator ID	Total lag
STABLE	1	9	25	0	0

Search by Topic Name

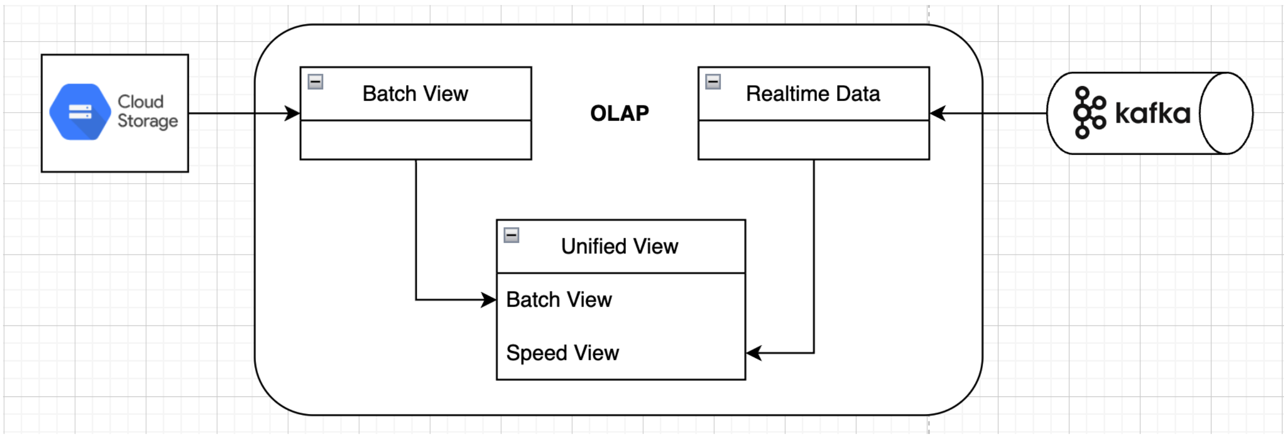
Topic	Consumer Lag
payments	0
products	0
promotions	0
inventories	0
orderdetails	0
customers	0
categories	0
orders	0
sellers	0

OLAP

- The serving layer of the Lambda Architecture is meant to provide near real-time analytics.

- Clickhouse is used as an OLAP Database in the Serving layer. ClickHouse uses a columnar database model, so it fits the requirements of delivering near real-time data and views due to fast query execution.

- Clickhouse get precomputed batch views by ingesting from GCS and receives real-time data by ingesting from Kafka.

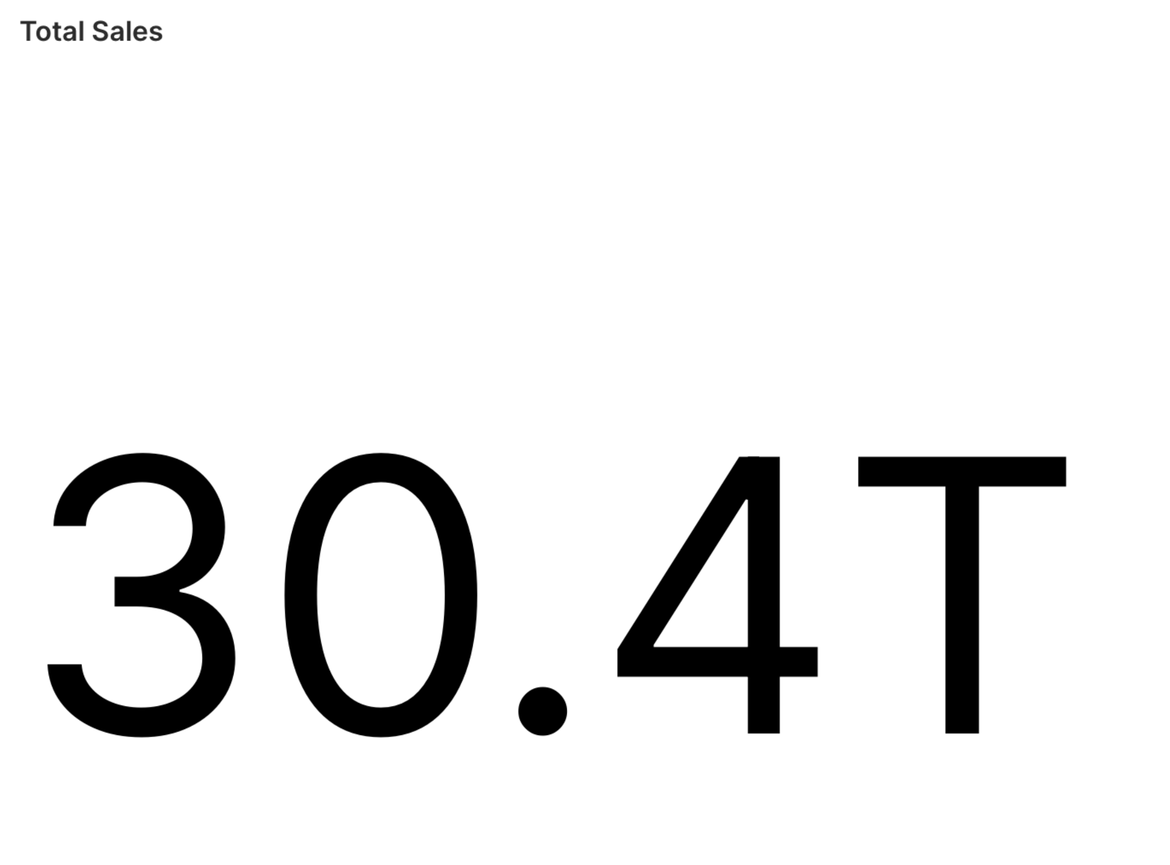


Visualization

- I use Apache Superset for creating and building charts, dashboards. In this project, I want to have analysis on sales data along with others data.

Total Sales

- Analyze the total quantity of products sold and the total amount of revenue generated.

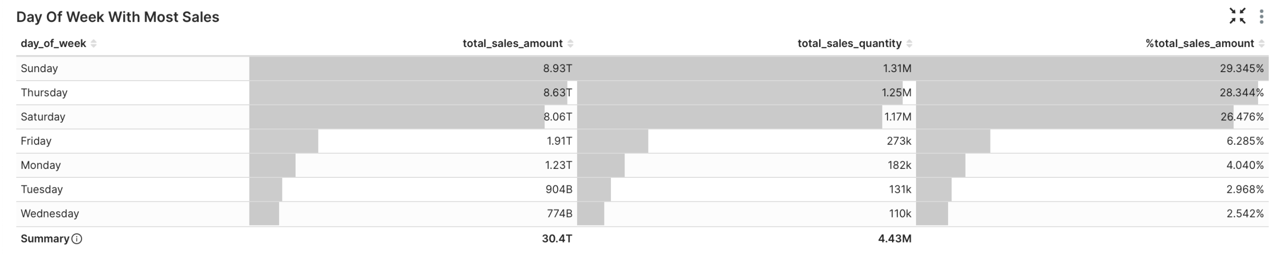


Total Sales

30.4T

Day Of Week With Most Sales

- Determine which day of the week experiences the highest sales volume. It helps in identifying peak sales days, useful for inventory management and marketing campaigns.



Day Of Week With Most Sales

day_of_week	total_sales_amount	total_sales_quantity	%total_sales_amount
Sunday	8.93T	1.31M	29.345%
Thursday	8.63T	1.25M	28.344%
Saturday	8.06T	1.17M	26.476%
Friday	1.91T	273k	6.285%
Monday	1.23T	182k	4.040%
Tuesday	904B	131k	2.968%
Wednesday	774B	110k	2.542%
Summary	**30.4T**	**4.43M**	

Top Sellers

- Identify and display the top-performing sellers.

Top 5 Most Profitable Products By Category

- Analyze the top 5 products with the highest profit within each category.

Most Common Payment Methods

- Visualize the distribution of payment methods used by customers (credit card, cash, …).

Customer Ranking

- Rank customers based on metrics such as total purchases