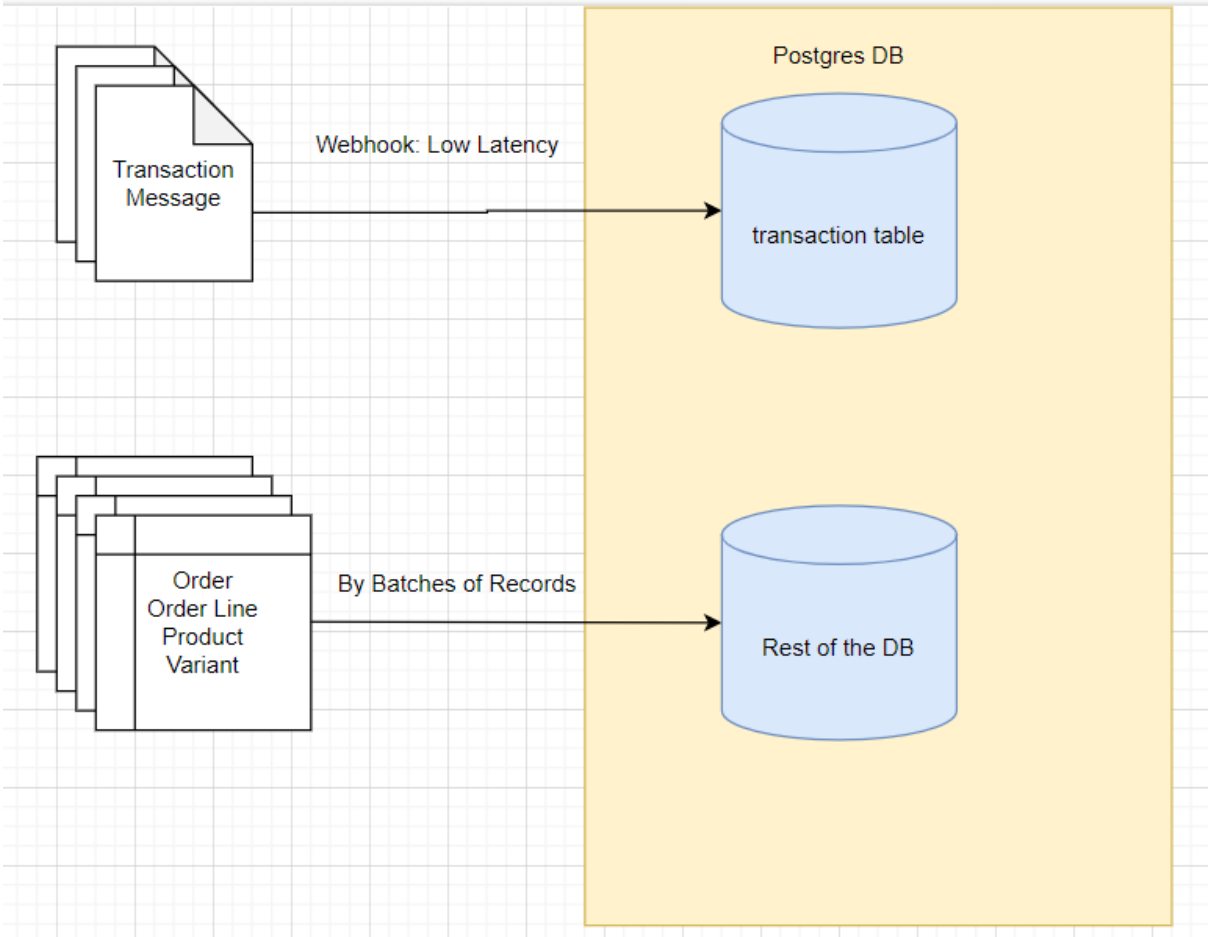
Design Constraints:

Based on the specification given to me, the data at the Postgres DB in the following fashion:



From the above figure, it is clear that when end users make orders, any updates to the transaction are updated through a webhook triggered by the backend API to update the transaction table in Postgres directly.

The details of the orders are cached somewhere in the backend service before batching it and sending it to the DB. So it is clear that there will be a significant latency between transaction status arrival in the Postgres Layer and the rest of the tables. Between the different datasets such as Order and Variant, the data arrival may also not be synchronous.

We further assume that the data formatting is fixed after it has landed, i.e. we cannot change the tables that is being created in Create_DB_tables_pg.sql

We further noted that the table structure of the **variant** table has an updated_at timestamp, created_at timestamp and ingested_at timestamp. This implies that besides ingesting the data batch-wise from some backend service, it is likely that engineers can **update** the product_id prices via another separate end point. When they do an update on this table, the price will be updated.

Design Objective:

From the problem statement, I am supposed to design for the base layer (described in Create_DB_tables_pg.sql) and OLAP layer from the OLTP layer.

From the data and the SQL create table script, one can understand the following business flow:

1. A product has a title and product_type which are its natural keys. It corresponds to a surrogate key called PRODUCT_ID. (Seen in the product table in create_db_table.sql)
2. A product may have different variants and each variant has a different price and its own variant title. The price can be updated from a different route, aside from normal data ingestion.
3. When a customer (denoted by the surrogate key customer_id) placed an order from the Front End UI, a transaction operation will occur in the backend resulting in an ORDER_ID.
4. An ORDER_ID may result in “success” or “failure” as its STATUS.
5. An ORDER_ID may have a different PRODUCT_ID in ORDER_LINE data. This means that a customer can order multiple different variant of different products.

As our objective is to design for the OLAP layer, it is critical to consider the use cases from the end users:

They have to analyse the following metrics:

1. Revenues (int)

2. Item Sales (it is the total number of items for a given PRODUCT_ID or its could be the distinct PRODUCT_ID for a given set of dimensions)
3. Variant Price Changes (This implies that for a given PRODUCT_ID)
4. Payment has been made and time it has occurred on a specific order_id

The metrics that the BI may requires to aggregate by (or dimensions) are:

1. PRODUCT_ID
2. CUSTOMER_ID
3. DATE

Data SLA:

In this exercise, from all the specs, we want to have the following latency:

1. Operational Data Store (ODS) layer for almost real time query performance for business unit to query backend data with low latency. No complex queries will be performed in this base layer
2. BI requirement from the given requirement.md file, it is clear daily T-1 is sufficient to fulfil the end user requirements.

Tech Design Base Layer:

For the base layer or Operational Data Store (ODS) Layer, we consider the following:

1. We assume that the data in batches provided to us from the OLTP layer will land as a file from S3. (This was the assumption given in the data pipeline question)
2. The orders table is about 20 - 30 M records assuming some reasonable data retention.

Let's consider the loading patterns for each tables one at a time:

Product/ Variant:

We can safely assume that the product and variant data is populated through a separate business flow as the order transaction flow. Business team will make use of some backend API to produce batches of product updates. The backend API will then create 2 products and variants csv files and save in S3.

Product and Variant table is designed to reflect the correct existing product offers of the company at any point in time. Hence, the retention of the table will not be set. Instead, the business team will perform clean up through an API to remove products and variants that are no longer being offered.

Product will be the parent table in this case as a Product_id can have multiple variant_ids. As we want to ensure the referential integrity of the product_id (as this is an important guarantee), we will make sure the product_id is a hard foreign key.

If a product_id is deleted, we want to make sure the variants are all also deleted as well!

Here's the schema of the product and variant table

```
CREATE TABLE product (
  id VARCHAR NOT NULL,
  TITLE VARCHAR(255) NOT NULL,
  PRODUCT_TYPE VARCHAR(45) NOT NULL,
```

```
  CREATED_AT TIMESTAMP NOT NULL
  PRIMARY KEY (id)
);

CREATE TABLE variant (
  id VARCHAR NOT NULL,
  PRODUCT_ID VARCHAR NOT NULL,
  PRICE INT NOT NULL,
  TITLE VARCHAR(255) NOT NULL,
  CREATED_AT TIMESTAMP NOT NULL,
  UPDATED_AT TIMESTAMP NOT NULL,
  INGESTED_AT TIMESTAMP NOT NULL
  PRIMARY KEY (id)
  CONSTRAINT fk_product_id
     FOREIGN KEY(fk_product_id)
          REFERENCES product (fk_product_id)
          ON DELETE CASCADE
        DEFERRABLE
        INITIALLY DEFERRED

);
```

By initially deferring the constraint checks, we avoid complicated issues with the insertion orders.

For a new products and their variants, the backend API can trigger the following sql logic either through trigger a store procedure:

```
BEGIN;

INSERT INTO product VALUES (...)
INSERT INTO variant VALUES (...)

COMMIT;
```

Backend API can also write their own transactions using their native jdbc implementations of transactions. An example of this: https://www.postgresqltutorial.com/postgresql-jdbc/transaction/

Transaction/ Orders/ Order_Line:

We assume that users cannot order items that are not in the Product or variant table. Whatever that the end user is order must exists in the product page which is a 100 percent match to the product and variant table.

Transaction Table ODS Design

As the transaction data arrives at a different rate compared to orders/ order line, we will first populate a staging table of transaction 1 to 1 via a webhook.

The table will be created using the following:

```
CREATE TABLE transaction (
  ID VARCHAR NOT NULL,
  ORDER_ID VARCHAR NOT NULL,
  STATUS VARCHAR(45) NOT NULL,
  CREATED_AT TIMESTAMP NOT NULL,
  PRIMARY KEY (id)
);

CREATE INDEX order_id_idx
  ON transaction_ods (ORDER_ID  ASC);
```

Assuming that the maximum latency for order/ order_line is 1 day, and also allow for potential downtime, we can set this ods table to have a retention of 3 days (to be safe).

Order/ Orderline ODS Table Design

As a batch of orders arrives as a csv file on S3, we will load this data into an ODS layer in Postgres. We assume that order and orderline batch data is

likely to be generated as a transaction by the backend, hence both batches will be written to S3 about the same time.

Let’s look at the table design for this:

```
CREATE TABLE order (
   id VARCHAR NOT NULL,
  PROCESSED_TIMESTAMP TIMESTAMP NOT NULL,
  CUSTOMER_ID VARCHAR NOT NULL,
  PRIMARY KEY (id)
);

CREATE INDEX customer_id_idx
  ON order (CUSTOMER_ID  ASC);


CREATE TABLE order_lines (
  id VARCHAR NOT NULL,
  QUANTITY INT NOT NULL,
  VARIANT_ID VARCHAR NOT NULL,
  REVENUE INT NOT NULL,
  ORDER_ID VARCHAR NOT NULL,
  PRIMARY KEY (id)
 CONSTRAINT fk_order_id
     FOREIGN KEY(fk_order_id)
          REFERENCES order(fk_order_id)
          ON DELETE CASCADE
        DEFERRABLE
        INITIALLY DEFERRED
);
```

We don’t expect that much data for 3 days of data retention of this ods table, hence, there is no need to create a partitioned table.

Also, it is likely that some coworkers i.e. customer service may need to query low latency data from this table. Having a customer_id as a index allow quick filtering.

Now, we will insert data from the CSV file on S3 to the table. A simple design could be as follows:

1. Backend Orders data in the cache layer is ready to be batched and stored in S3 bucket.
2. Backend API service triggers the data to save into the S3 bucket.
3. With an aws_s3 extension, Backend API service triggers and loads the csv file to the ODS table into the AWS RDS Postgres Serivce. https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/USER_PostgreSQL.S3Import.html

One order will have different order_items, hence the order will be taken as a parent table and the order_line table will be taken as a child. We will use similar strategies as the product/ variant to insert / delete the tables. This is especially important, as we are setting a table retention of 3 days on both order_line and order. Notice that order_line do not have a timestamp field which makes it difficult to perform retention operation if the fk is a soft one. By deleting old order_id records in orders table using processed_timestamp as a predicate, we are able to cascade the delete operation to the product table.

The order_id is a foreign key between orders, transaction.. However, as it is difficult to maintain referential integrity via a transaction without too complicated design as the data arrives at different times. Instead, the fk will be a soft one.

Tech Design OLAP Layer:

Replica from ODS:

Since the data is already landing in S3 AWS, the more cost efficient way to run OLAP query may not be on the same Postgres instance. as computation logic could be very complex. Building the OLAP layer on the same DB service is not advisable as it will lead to workload optimization issues.

In addition. Postgres is a row base database and it really not optimal for column based aggregation operation that is common in OLAP queries!

Instead, this is the proposal:

1. We will do T-1 scheduled job to dump the ODS data (with 1-2 hours buffer based on the estimated worst case arrival of the late data) to an S3 bucket.
2. We will employ Snowflake as a data warehouse that is optimal for compute. We can do simple COPY INTO a replica staging table **with the exact same table schema.** For instance, we will create or replace a temporary staging table of orders in snowflake. The table name can be called orders_staging
3. Then once we point the table to the data in the S3 bucket, we will make use of the micro-partition concepts of Snowflake to optimise our query pruning. Instead of doing this:

```
INSERT INTO orders
SELECT *
FROM orders_staging;
```

We can do this:

```
INSERT INTO orders
SELECT *
FROM orders_staging
order by order_id
```

By sorting this column and inserting them into immutable micro-partition, snowflake compute engine will be better prune the tree to identify micro-partitions that contains the rows contains a certain order_id!

(You can refer to my tech blog on this. https://medium.com/@421250/title-lessons-from-snowflake-migrations-part-1-snowflake-clustering-keys-for-cost-savings-and-8bd1c6c2ca7c)

So by doing the 1st step which is replicate data from Postgres, we will create the following tables:

1. transaction (incremental table)

```
CREATE TABLE transaction (
  ID VARCHAR NOT NULL,
  ORDER_ID VARCHAR NOT NULL,
  STATUS VARCHAR(45) NOT NULL,
  CREATED_AT TIMESTAMP NOT NULL,
  PARTITION_DATE DATE,
  PRIMARY KEY (id)
);
```

2. orders (incremental table)

```
CREATE TABLE orders (
  id VARCHAR NOT NULL,
  PROCESSED_TIMESTAMP TIMESTAMP NOT NULL,
  CUSTOMER_ID VARCHAR NOT NULL,
  PARTITION_DATE DATE,
  PRIMARY KEY (id)
);
```

3. orders_lines (incremental table)

```
CREATE TABLE order_lines (
  id VARCHAR NOT NULL,
  QUANTITY INT NOT NULL,
  VARIANT_ID VARCHAR NOT NULL,
  REVENUE INT NOT NULL,
  ORDER_ID VARCHAR NOT NULL,
  PARTITION_DATE DATE,
```

```
  PRIMARY KEY (id)
);
```

4. Product (snapshot table, each partition_date represents a full dump of the data)

```
CREATE TABLE product (
  id VARCHAR NOT NULL,
  TITLE VARCHAR(255) NOT NULL,
  PRODUCT_TYPE VARCHAR(45) NOT NULL,
  CREATED_AT TIMESTAMP NOT NULL,
  PARTITION_DATE DATE,
  PRIMARY KEY (id)
);
```

5. variant (snapshot table, each partition_date represents a full dump of the data)

OLAP Layer:

Now we then focus on the OLAP layer:

To facilitate computations of the following facts:

1. Revenue
2. Item Sales
3. Variant Price Changes

We need to first build a dimensional layer according to the star schema.

DIM Layer:

We will need to build a dim_product table and dim_customer_orders table. In order to allow more efficient queries, the dim_product will be a snapshot table that represents the latest product offerings for a given day.
Dim_customer_order will be a summary of the dimensions for a given day for each of the PARTITION_DATE

The dim_product table will consist of the following:

```
CREATE table dim_product (
  PRODUCT_ID VARCHAR NOT NULL,
  TITLE VARCHAR(255) NOT NULL,
  PRODUCT_TYPE VARCHAR(45) NOT NULL,
  CREATED_AT TIMESTAMP NOT NULL,
  VARIANT_ID VARCHAR NOT NULL,
  PRICE INT NOT NULL,
  VARIANT_TITLE VARCHAR NOT NULL,
  CREATED_AT TIMESTAMP NOT NULL,
  UPDATED_AT TIMESTAMP NOT NULL,
  INGESTED_AT TIMESTAMP NOT NULL
  PARTITION_DATE DATE,
);
```

We will join product and variant table for the same PARTITION_DATE to get this table. We will order by product_id and variant_id when inserting into dim_product snowflake table to maximise micro-partitioning pruning.

The dim_customer_orders table:

```
CREATE table dim_customer_orders (
  CUSTOMER_ID VARCHAR NOT NULL,
  ORDER_ID VARCHAR(255) NOT NULL,
  PARTITION_DATE DATE,
);
```

We will join the orders table with the yesterday's dim_customer_orders and form a superset of the orders.

We will sort by customer_id and order_id before inserting into the dim_customer_orders table

Fact Layer (Datawarehouse Detailed Layer, DWD):

We will build a fact table called dwd_orders_variant_status

```
CREATE table dwd_orders_variant_status (
  ORDER_ID VARCHAR(255) NOT NULL,
  ORDER_STATUS VARCHAR(45) NOT NULL,
  ORDER_CREATED_AT TIMESTAMP NOT NULL,
  ORDER_PROCESSED_TIMESTAMP TIMESTAMP NOT NULL,
  QUANTITY INT NOT NULL,
  VARIANT_ID VARCHAR NOT NULL,
  PRODUCT_ID VARCHAR NOT NULL,
  REVENUE INT NOT NULL,
  PARTITION_DATE DATE
);
```

This table is formed by joining transactions, orders and order_line tables. Noticed that column names is also changed. This is to allow downstream users to not be confused with the namings

Application Layers:

The final layer is formed by joining the dims and fact tables and perform pre-aggregations based on the business needs. For requirements that changes rapidly, we can choose to use to build materialized views/ views. For requirements that changes slowly, or no changes, we can actually build physical application tables for their use cases.