I. SQL

1. View query optimization.
   a. The concept here is to determine the output of the view and rephrase the entire query based on your own logic. This method sets a specific target, allowing you the flexibility to choose the approach. I can identify an optimized way to achieve the same outcome using my experience and knowledge. During my last internship, I successfully migrated code from SQL to Spark by applying this approach.
   b. I managed to enhance the view's performance by **29.4%,** inclusive of subsequent improvements. Previously, the processing time was around 170ms, and now it has reduced to 120ms.
   c. I identified three unnecessary Common Table Expressions (CTEs) in the view, incurring additional costs in terms of grouping, result set data processing, and memory consumption.
   d. The sole essential CTE was 'total_orders_by_month' as it only grouped data by month. Postgres does not support nested aggregation.
   e. I eliminated the unnecessary typecast of 'order_month' to text data type.
   f. To streamline the process, I reduced the sub-result set, selecting only the necessary attributes from the tables involved in the joins.
   g. The order of joins, particularly in the case of inner joins, is crucial. Attempting to minimize the number of rows in the initial join can, at times, contribute to performance improvement.
   h. I have done all the above changes to the old view ‘v_product_orders_by_month’.
2. Further improvements.
   a. Indexing significantly enhances query performance. I've identified three indexes and implemented them (see the code below). Indexing on the **grouping column**, join clause attributes, distinct attributes, and the foreign key column has resulted in a 4-5ms improvement in query performance.
   b. Another optimization involves adding a new column, 'order_month,' to the 'orders' table (**pre calculated order month**) and creating an index on this column. Based on this, I have introduced a new view named 'v_product_orders_by_month_already_calculated' resulting in an additional 10ms improvement in performance.
   c. While materialized views can deliver exceptional performance, there is a trade-off. I've developed a materialized view named 'mv_product_orders_by_month,' which produces results in just 1ms. However, it's important to note that materialized views consume memory as results are stored. If the underlying table data is refreshed at short intervals, consistency issues may arise.
   d. Data partitioning by the 'order_datetime' of the 'orders' table can significantly enhance view performance, especially when grouping the table by datetime month. However, given that the 'orders' table is relatively small, this may not be the optimal solution due to the overhead associated with data partitioning.

Query optimized view

```
DROP VIEW v_product_orders_by_month;

CREATE VIEW v_product_orders_by_month AS
WITH total_orders_by_month AS(
SELECT  date_trunc('month', orders.order_datetime) AS order_month,
```

```
        count(DISTINCT orders.id) AS orders
           FROM public.orders
           GROUP BY (date_trunc('month', orders.order_datetime))
)
SELECT date_trunc('month', ord.order_datetime) AS order_month,
                 prd.product_name ,
            count(DISTINCT ord.id) AS orders,
            ((count(DISTINCT oi.order_id))::numeric / (
max(ttl_ords_mnth.orders))::numeric) AS share_of_orders,
            sum(oi.quantity) AS total_quantity,
            sum((prd.product_price * (oi.quantity)::double precision)) AS
total_price
FROM (SELECT order_id,product_id, quantity FROM public.order_items) oi
LEFT JOIN (SELECT id,product_name,product_price FROM public.products) prd
ON oi.product_id = prd.id
LEFT JOIN (SELECT id,order_datetime FROM public.orders )ord ON    ord.id =
oi.order_id
LEFT JOIN total_orders_by_month AS ttl_ords_mnth ON
ttl_ords_mnth.order_month = date_trunc('month', ord.order_datetime)
GROUP BY (date_trunc('month', ord.order_datetime)),prd.product_name
ORDER BY order_month, prd.product_name
```

Optimized view by adding new column order_month to the orders table.

```
ALTER TABLE orders
ADD COLUMN order_month timestamp;

UPDATE orders
SET order_month = date_trunc('month', orders.order_datetime);

CREATE VIEW v_product_orders_by_month_already_calculated AS
WITH total_orders_by_month AS(
SELECT   orders.order_month,
        count(DISTINCT orders.id) AS orders
           FROM public.orders
           GROUP BY  orders.order_month
)
SELECT      ord.order_month,
                 prd.product_name ,
            count(DISTINCT ord.id) AS orders,
            ((count(DISTINCT oi.order_id))::numeric / (
max(ttl_ords_mnth.orders))::numeric) AS share_of_orders,
            sum(oi.quantity) AS total_quantity,
            sum((prd.product_price * (oi.quantity)::double precision)) AS
total_price
FROM (SELECT order_id,product_id, quantity FROM public.order_items) oi
LEFT JOIN (SELECT id,product_name,product_price FROM public.products) prd
ON oi.product_id = prd.id
LEFT JOIN (SELECT id,order_month FROM public.orders )ord ON       ord.id =
oi.order_id
LEFT JOIN total_orders_by_month AS ttl_ords_mnth ON
ttl_ords_mnth.order_month = ord.order_month
GROUP BY ord.order_month,prd.product_name
ORDER BY order_month, prd.product_name;
```

Materialized view

```
CREATE MATERIALIZED VIEW IF NOT EXISTS mv_product_orders_by_month AS
WITH total_orders_by_month AS(
```

```
SELECT date_trunc('month', orders.order_datetime) AS order_month,
        count(DISTINCT orders.id) AS orders
           FROM public.orders
           GROUP BY (date_trunc('month', orders.order_datetime))
)
SELECT date_trunc('month', ord.order_datetime) AS order_month,
                  prd.product_name ,
            count(DISTINCT ord.id) AS orders,
            ((count(DISTINCT oi.order_id))::numeric / (
max(ttl_ords_mnth.orders))::numeric) AS share_of_orders,
            sum(oi.quantity) AS total_quantity,
            sum((prd.product_price * (oi.quantity)::double precision)) AS
total_price
FROM (SELECT order_id,product_id, quantity FROM public.order_items) oi
LEFT JOIN (SELECT id,product_name,product_price FROM public.products) prd
ON oi.product_id = prd.id
LEFT JOIN (SELECT id,order_datetime FROM public.orders )ord ON    ord.id =
oi.order_id
LEFT JOIN total_orders_by_month AS ttl_ords_mnth ON
ttl_ords_mnth.order_month = date_trunc('month', ord.order_datetime)
GROUP BY (date_trunc('month', ord.order_datetime)),prd.product_name
ORDER BY order_month, prd.product_name
WITH DATA;
```

Indexes

```
CREATE INDEX idx_order_items_ord_prd_id
ON order_items (order_id,product_id);

CREATE INDEX idx_products_product_name
ON products (product_name);

CREATE INDEX idx_orders_order_date
ON orders(order_datetime);

CREATE INDEX idx_orders_order_month
ON orders(order_month);
```

II. Python

a. The initial step involved retrieving the 'id' and 'contact_email' from the 'customers' table into a Pandas data frame.
b. Extracting the domain name from the 'contact_email' served as the basis for the enrichment API query parameter.
c. Due to duplicate domain names in contact_email (reduced to 2816 from 3701), a separate data frame was created to minimize the number of API calls.
d. Retrieving API responses posed limitations of 200 calls per hour, 50 calls per minute, and 600 calls per day. For those with the premium package, these constraints are alleviated, allowing for a comprehensive retrieval. In my last internship, I encountered a similar situation with the Denodo application, necessitating data batching to obtain information.
e. To address the API limitations, I created batches of 50, usable four times per hour and 12 times per day.
f. Once industry information for all domains was obtained, the data could be joined with the original data frame.

g. Adding the new 'industry' column to the 'customer' table, the 'id' of the customer served as the reference point to accurately insert the industry values into the table.
h. For interactions outside the Python environment, particularly with APIs and databases, exception handling is essential.
i. I have implemented all the above and submitted Jupyter notebook.