1. **SQL**
2. View query optimization.
3. The idea here is to find what the view outputs, and rewrite the whole query in your own logic. Using this method gives you a specific target to achieve, and you are open to choose the path you want. I can find the optimized way to achieve the same with my experience and knowledge. In my experience, I had to migrate code from SQL to spark in my last internship, this idea worked very well for me.
4. I was able to improve the performance of the view by **29.4%** (this includes the further improvements). Earlier it was hovering around 170ms, now it came down to 120ms.
5. I found there were **3** unnecessary CTE’s in the view. Which were costing the extra grouping cost, result set data processing and memory consumption.
6. The only necessary CTE was total\_orders\_by\_month. Because it was grouping data only by month. Postgres does not allow nested aggregation.
7. The unnecessary typecast of order\_month to text was removed.
8. Reduce the sub result set, select only the required attributes from the tables in the joins.
9. The orders of join (mostly inner joins case) are important, if we try to reduce the number of rows in initial join can sometime help improve performance.
10. Further improvements.
11. Indexes can improve the performance of the queries. I have identified 3 indexes and created them (below I have copied the code). The **grouping column**, the join clause attributes, distinct attributes and foreign key column can be indexed for better performance. This further improved the performance of the query by 4-5ms.
12. We can add new column to the table orders called as **order\_month**. Add index to the same column. Based on this I created a new view called v\_product\_orders\_by\_month\_already\_calculated. This further improved the performance by 10ms.
13. Materialized view can generate amazing performance, but at a cost. I have created this materialized view mv\_product\_orders\_by\_month. This generates result in 1ms. The trade off here is that it consumes memory as the results are stored, if the underlying table data is refreshed in small intervals, then there can be consistency issues.
14. Partition of data by the order\_datetime of the table orders can help improve the performance of the view, as we are grouping the table by datetime month. If the table has huge amount of data this can be beneficial. The orders table is relatively small so this is not the best solution in this case.

**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

**INNER** **JOIN** (**SELECT** id,product\_name,product\_price **FROM** public.products) prd **ON** oi.product\_id = prd.id

**INNER** **JOIN** (**SELECT** id,order\_datetime **FROM** public.orders )ord **ON** ord.id = oi.order\_id

**INNER** **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

**INNER** **JOIN** (**SELECT** id,product\_name,product\_price **FROM** public.products) prd **ON** oi.product\_id = prd.id

**INNER** **JOIN** (**SELECT** id,order\_month **FROM** public.orders )ord **ON** ord.id = oi.order\_id

**INNER** **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

**INNER** **JOIN** (**SELECT** id,product\_name,product\_price **FROM** public.products) prd **ON** oi.product\_id = prd.id

**INNER** **JOIN** (**SELECT** id,order\_datetime **FROM** public.orders )ord **ON** ord.id = oi.order\_id

**INNER** **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);

1. **Python**
2. First step was to retrieve the id and contact\_email from the customers table to pandas data frame.
3. Extract the domain name from the contact\_email for the enrichment API query parameter.
4. There were duplicate domain names (reduced to 2816 from 3701), so had to create a separate data frame to reduce the number of API calls.
5. While retrieving the response from the API, it had limitations of 200 calls per hour, 50 calls per minute and 600 calls per day. If you have their premium package, you do not have to worry about this, there you can loop through all at once and get the industry information.
6. Here I had to create batch of 50 which can be used 4 times an hour and 12 times a day. I had faced a similar issue due to limitation of the Denodo application in my last internship. Where I had to batch the data to retrieve information from Denodo.
7. After I have the industry information for all the domain, you can join the data with the original data frame.
8. Add the new column industry to the table customer, Now using the id of the customer as the reference you can insert the industry values correctly to the table.
9. Whenever you interact outside of python environment (API and Database), use exception handling.