352 Database Management Systems

Project 2

1.Programming Task

In this part, I will talk about some assumptions and implementations that I did for functions I wrote. Since I believe there are some ambiguities in some of tasks. In the mp2.py file I wrote some comments about the functions: 'change_stocks', 'ship', 'calc_gross', 'change_cart' and 'purschase_cart'.

- In shipping function, there was no information about how much of the stock that the sellers ship out. So, I decided to remove only 1 stock per product from the seller_stocks.

- In the calc_gross, I didn't know how to calculate gross sum, so I just sum the prices.

- For the remaining functions, I thought that seller and customers should be able add new rows to seller_stocks or customer_carts with their functionalities.

So, when there is an add comment that obeys the rule of maximum and minimum. A new row will be added to the tables (customer_carts or seller_stocks depending on function). Also, while removing if calculation results in new value smaller than 0 then function will fail, but if they result in exactly 0 the row will be deleted from the tables (customer_carts or seller_stocks).

You can find my comments in the file.

2.Written Task

A)

I would use multiuser access mode because the functions are able work for different kind of users by implementation.

For sign up I would use repeatable read isolation level, because we don't want an uncommitted user info to be usable.

For show_plans, show_subscription, show_quota, calc_gross and show_cart functions, I would use read uncommitted isolation level because these are printing functions that modifies nothing, so being fast is I think better.

For sign_in, change_stock, ship, change_cart and purchase_cart, I would use read committed because these functions all make changes in database and reading uncommitted data may result in incorrect results. Such as user taking 5 stocks from both of his\her sessions and end up with 10 products even though there wasn't that many product in the beginning.

B)

```
explain select seller_id,
              product_category_name,
              EXTRACT(YEAR FROM order_purchase_timestamp) as Year,
              EXTRACT(MONTH FROM order_purchase_timestamp) as Month,
              count(*)
from (select product_id, product_category_name from products)as p,
     (select order_id,product_id,seller_id from order_items) as oi,
     (select order_id, order_purchase_timestamp from orders) as oo
where p.product_id = oi.product_id and oi.order_id = oo.order_id
group by seller_id,
         product_category_name,
         order_purchase_timestamp;
```

This is the query I wrote to obtain the information wanted. And below is the output I received after the Explain command.

This is the time it took to execute.

```
ceng352_2022_hw2.public> select seller_id,
                                product_category_name,
                                EXTRACT(YEAR FROM order_purchase_timestamp) as Year,
                                EXTRACT(MONTH FROM order_purchase_timestamp) as Month,
                                count(*)
                         from (select product_id, product_category_name from products)as p,
                              (select order_id,product_id,seller_id from order_items) as oi,
                              (select order_id, order_purchase_timestamp from orders) as oo
                         where p.product_id = oi.product_id and oi.order_id = oo.order_id
                         group by seller_id,
                                  product_category_name,
                                  order_purchase_timestamp
[2023-06-12 23:09:38] 500 rows retrieved starting from 1 in 683 ms (execution: 665 ms, fetching: 18 ms)
```

Query plan before any indexes

```
QUERY PLAN
Finalize GroupAggregate  (cost=20231.14..31529.33 rows=112651 width=128)
  Group Key: order_items.seller_id, products.product_category_name, orders.order_purchase_timestamp
  ->  Gather Merge  (cost=20231.14..29176.92 rows=66265 width=64)
        Workers Planned: 1
        ->  Partial GroupAggregate  (cost=19231.13..20722.09 rows=66265 width=64)
              Group Key: order_items.seller_id, products.product_category_name, orders.order_purchase_timestamp
              ->  Sort  (cost=19231.13..19396.79 rows=66265 width=56)
                    Sort Key: order_items.seller_id, products.product_category_name, orders.order_purchase_timestamp
                    ->  Parallel Hash Join  (cost=4818.56..11656.64 rows=66265 width=56)
                          Hash Cond: ((order_items.order_id)::text = (orders.order_id)::text)
                          ->  Hash Join  (cost=1179.40..5052.19 rows=66265 width=81)
                                Hash Cond: ((order_items.product_id)::text = (products.product_id)::text)
                                ->  Parallel Seq Scan on order_items  (cost=0.00..2961.65 rows=66265 width=99)
                                ->  Hash  (cost=767.51..767.51 rows=32951 width=48)
                                      ->  Seq Scan on products  (cost=0.00..767.51 rows=32951 width=48)
                          ->  Parallel Hash  (cost=2392.96..2392.96 rows=58496 width=41)
                                ->  Parallel Seq Scan on orders  (cost=0.00..2392.96 rows=58496 width=41)
```

From this query plan, we can see there is a lot of computational power going to grouping. Also, database uses hash for joining order_id's and product_id's. There are sequential scans on order_items, products and orders.

I created the indexes below,

```
14  create index p1 on products using hash(product_id);
15  create index p2 on order_items using hash(product_id);
16  create index o1 on order_items using hash(order_id);
17  create index o2 on orders using hash(order_id);
```

The time it took is given below. I executed this and original query one after the other so system weight difference is I think is minimum and nothing changed for both plan and speed.

```
ceng352_2022_hw2.public> select seller_id,
                                product_category_name,
                                EXTRACT(YEAR FROM order_purchase_timestamp) as Year,
                                EXTRACT(MONTH FROM order_purchase_timestamp) as Month,
                                count(*)
                         from (select product_id, product_category_name from products)as p,
                              (select order_id,product_id,seller_id from order_items) as oi,
                              (select order_id, order_purchase_timestamp from orders) as oo
                         where p.product_id = oi.product_id and oi.order_id = oo.order_id
                         group by seller_id,
                                  product_category_name,
                                  order_purchase_timestamp
[2023-06-12 23:09:57] 500 rows retrieved starting from 1 in 693 ms (execution: 671 ms, fetching: 22 ms)
```

Then I created:

```
14  create index p1 on products (product_category_name);
15  create index p2 on order_items using hash(product_id);
16  create index o1 on order_items using hash(order_id);
17  create index o2 on orders (order_purchase_timestamp);
18  cluster products using p1;
19  cluster orders using o2;
```

Before creting the indexes, I got:

```
ceng352_2022_hw2.public> select seller_id,
                                product_category_name,
                                EXTRACT(YEAR FROM order_purchase_timestamp) as Year,
                                EXTRACT(MONTH FROM order_purchase_timestamp) as Month,
                                count(*)
                    from (select product_id, product_category_name from products)as p,
                         (select order_id,product_id,seller_id from order_items) as oi,
                         (select order_id, order_purchase_timestamp from orders) as oo
                    where p.product_id = oi.product_id and oi.order_id = oo.order_id
                    group by seller_id,
                             product_category_name,
                             order_purchase_timestamp
[2023-06-12 23:15:35] 500 rows retrieved starting from 1 in 673 ms (execution: 652 ms, fetching: 21 ms)
```

With indexes:

```
ceng352_2022_hw2.public> select seller_id,
                                    product_category_name,
                                    EXTRACT(YEAR FROM order_purchase_timestamp) as Year,
                                    EXTRACT(MONTH FROM order_purchase_timestamp) as Month,
                                    count(*)
                         from (select product_id, product_category_name from products)as p,
                              (select order_id,product_id,seller_id from order_items) as oi,
                              (select order_id, order_purchase_timestamp from orders) as oo
                         where p.product_id = oi.product_id and oi.order_id = oo.order_id
                         group by seller_id,
                                  product_category_name,
                                  order_purchase_timestamp
[2023-06-12 23:16:31] 500 rows retrieved starting from 1 in 652 ms (execution: 633 ms, fetching: 19 ms)
```

So, I believe I can say, there is a bit of improvement in the execution speed with these. But I couldn't be too sure because the query plan I get is:

```
QUERY PLAN
Finalize GroupAggregate  (cost=20231.14..31529.33 rows=112651 width=128)
  Group Key: order_items.seller_id, products.product_category_name, orders.order_purchase_timestamp
  ->  Gather Merge  (cost=20231.14..29176.92 rows=66265 width=64)
        Workers Planned: 1
        ->  Partial GroupAggregate  (cost=19231.13..20722.09 rows=66265 width=64)
              Group Key: order_items.seller_id, products.product_category_name, orders.order_purchase_timestamp
              ->  Sort  (cost=19231.13..19396.79 rows=66265 width=56)
                    Sort Key: order_items.seller_id, products.product_category_name, orders.order_purchase_timestamp
                    ->  Parallel Hash Join  (cost=4818.56..11656.64 rows=66265 width=56)
                          Hash Cond: ((order_items.order_id)::text = (orders.order_id)::text)
                          ->  Hash Join  (cost=1179.40..5052.19 rows=66265 width=81)
                                Hash Cond: ((order_items.product_id)::text = (products.product_id)::text)
                                ->  Parallel Seq Scan on order_items  (cost=0.00..2961.65 rows=66265 width=99)
                                ->  Hash  (cost=767.51..767.51 rows=32951 width=48)
                                      ->  Seq Scan on products  (cost=0.00..767.51 rows=32951 width=48)
                          ->  Parallel Hash  (cost=2392.96..2392.96 rows=58496 width=41)
                                ->  Parallel Seq Scan on orders  (cost=0.00..2392.96 rows=58496 width=41)
```

Which is the same with last one. So, the difference between speeds may be simply because of my computer speed.