CENG352-PA2 **Mert Sağcan / 2310449**

4.2-WRITTEN TASKS

4.2.1 Transaction Isolation Levels

1) Sign Up:

I would choose the Serializable isolation level because this action involves creating a new seller in the database. Using Serializable ensures that no other transactions can interfere with this process, preventing duplicate seller IDs and ensuring the integrity of the data.

2) Sign In:

I would choose the Read Committed isolation level because when signing in, the application checks the seller's credentials and the current session count. Read Committed ensures that only committed data is read, preventing the use of partially updated data. This level is sufficient as it allows for a consistent view of the data without the overhead of higher isolation levels.

3) Sign Out:

I would choose the Read Committed isolation level because, similar to sign in, signing out involves updating the session count. Read Committed ensures that the session count is accurately read and updated, avoiding issues with dirty reads.

4) Show Plans:

I would choose the Read Committed isolation level because this action involves reading available subscription plans. Read Committed is suitable because it ensures that only committed plans are displayed to the user.

5) Show Subscription:

I would choose the Read Committed isolation level because reading the subscription details for a seller ensures that the seller's subscription details are consistently read from the database.

6) Change Product Stock:

I would choose the Serializable isolation level because changing the stock of a product involves reading the current stock and updating it. Using Serializable ensures that no other transactions can change the stock simultaneously, preventing issues with inconsistent stock levels.

7) Subscribe To a A New Plan:

I would choose the Serializable isolation level because subscribing to a new plan involves updating the seller's subscription details. Serializable isolation level ensures that no other transactions can interfere with this update, maintaining data integrity.

8) Ship Orders:

I would choose the Serializable isolation level because shipping orders involves checking stock levels, updating them, and changing the order status. Serializable ensures that these operations are atomic and consistent, preventing issues with concurrent stock updates.

9) Quit:

I would choose the Read Committed isolation level because quitting the application involves signing out the user and closing the session. Read Committed ensures that the session count is updated correctly without needing the overhead of higher isolation levels.

10) Show Customer Cart:

I would choose the Read Committed isolation level because reading the items in the customer's cart ensures that only committed data is read, providing a consistent view of the cart items.

11) Change Customer Cart:

I would choose the Serializable isolation level because changing the cart involves adding or removing items, which requires checking stock levels and updating the cart. Serializable ensures that these operations are atomic and consistent, preventing issues with concurrent updates.

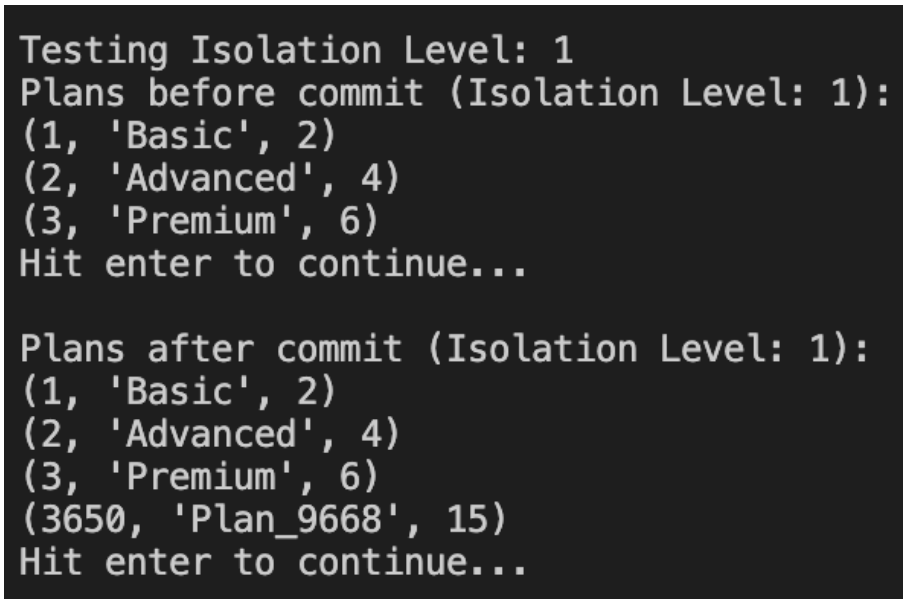
12) Purchase Cart:

I would choose the Serializable isolation level because purchasing the cart involves finalizing the order, updating stock levels, and changing the order status. Serializable ensures that these operations are atomic and consistent, preventing issues with concurrent stock updates and ensuring the integrity of the purchase process.

4.2.2 Transaction Isolation Levels Experiment

1) READ COMMITED Level:

As we can see from the screenshot below, after writer commits reader sees the change. In this level, the reader sees the commited data immediately after if commits. This doesn't prevent concurrency.



2) REPEATABLE READ Level:

In this level, the reader sees a consistent snapshot of the database, it doesn't see the changes made by the other transactions until the current transaction ends. This prevents non-repeatable reads.

```
Testing Isolation Level: 2
Plans before commit (Isolation Level: 2):
(1, 'Basic', 2)
(2, 'Advanced', 4)
(3, 'Premium', 6)
(3650, 'Plan_9668', 15)
Hit enter to continue...

Plans after commit (Isolation Level: 2):
(1, 'Basic', 2)
(2, 'Advanced', 4)
(3, 'Premium', 6)
(3650, 'Plan_9668', 15)
Hit enter to continue...
```

3) SERIALIZABLE Level:

As we can see in the screenshots, the reader does not see the changes made by other transactions until the current one ends. In serializable level, the changes appear to be executed sequentially. This prevents all concurrency issues, but it may increase wait time for some transactions.

```
Testing Isolation Level: 3
Plans before commit (Isolation Level: 3):
(1, 'Basic', 2)
(2, 'Advanced', 4)
(3, 'Premium', 6)
(3650, 'Plan_9668', 15)
(3128, 'Plan_1727', 12)
Hit enter to continue...

Plans after commit (Isolation Level: 3):
(1, 'Basic', 2)
(2, 'Advanced', 4)
(3, 'Premium', 6)
(3650, 'Plan_9668', 15)
(3128, 'Plan_1727', 12)
Hit enter to continue...
```

4.2.3 Indexes

Here is the script without indexes:

```
EXPLAIN ANALYZE
SELECT
     s.seller_id,
     COUNT(*) AS product_count,
     DATE_TRUNC('month', o.order_time) AS month
FROM
     sellers s
JOIN
     shopping_carts sc ON s.seller_id = sc.seller_id
JOIN
     orders o ON sc.order_id = o.order_id
JOIN
     products p ON sc.product_id = p.product_id
WHERE
     p.category_id = 10
GROUP BY
     s.seller_id, month
ORDER BY
     month, product_count DESC;
```

Here are the indexes I have defined:

```
CREATE INDEX idx_sellers_seller_id ON sellers(seller_id);
CREATE INDEX idx_shopping_carts_seller_id ON shopping_carts(seller_id);
CREATE INDEX idx_shopping_carts_order_id ON shopping_carts(order_id);
CREATE INDEX idx_orders_order_id ON orders(order_id);
CREATE INDEX idx_orders_order_time ON orders(order_time);
CREATE INDEX idx_products_product_id ON products(product_id);
CREATE INDEX idx_products_category_id ON products(category_id);
```

Observations:

Without Indexes: The lack of indexes results in sequential scans on large tables, leading to increased I/O operations and longer execution times. Nested loops contribute to the higher cost and slower performance due to repeated scans of the same data. (You can see the results in the screenshot below)

Execution Time: 28.991 ms

Execution Plan:

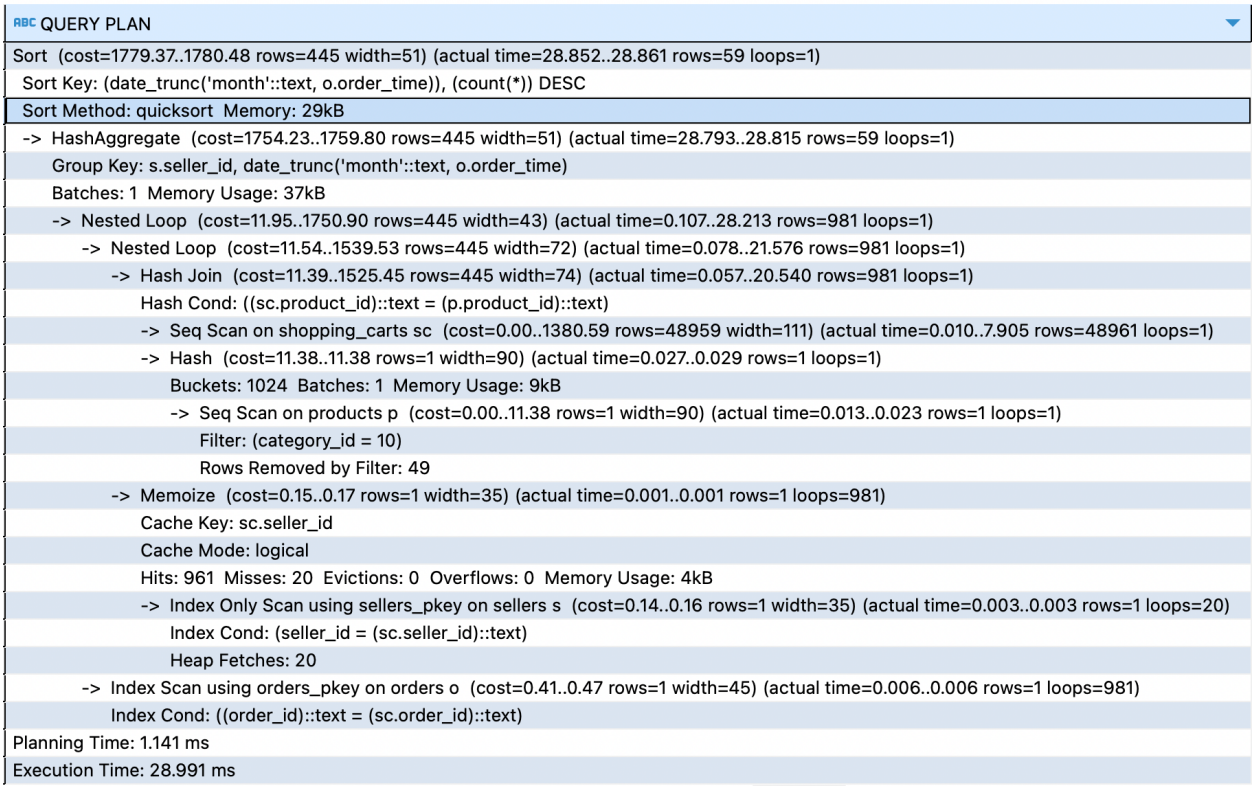
Sort: The query sorts the results by month and product_count using quicksort with a memory usage of 29 kB.

HashAggregate: This step aggregates the data by seller_id and month, with a memory usage of 37 kB.

Nested Loop: The nested loop joins the tables, causing multiple scans of the shopping_carts and products tables.
Seq Scan on shopping_carts: Sequential scans are performed on the shopping_carts table, leading to 48,961 rows being read.

Seq Scan on products: Sequential scans are also performed on the products table, filtering by category_id.

Memoize: This step uses memoization to improve performance but still involves index scans and heap fetches, resulting in a higher execution time.



```
QUERY PLAN
Sort  (cost=1779.37..1780.48 rows=445 width=51) (actual time=28.852..28.861 rows=59 loops=1)
  Sort Key: (date_trunc('month'::text, o.order_time)), (count(*)) DESC
  Sort Method: quicksort  Memory: 29kB
  ->  HashAggregate  (cost=1754.23..1759.80 rows=445 width=51) (actual time=28.793..28.815 rows=59 loops=1)
        Group Key: s.seller_id, date_trunc('month'::text, o.order_time)
        Batches: 1  Memory Usage: 37kB
        ->  Nested Loop  (cost=11.95..1750.90 rows=445 width=43) (actual time=0.107..28.213 rows=981 loops=1)
              ->  Nested Loop  (cost=11.54..1539.53 rows=445 width=72) (actual time=0.078..21.576 rows=981 loops=1)
                    ->  Hash Join  (cost=11.39..1525.45 rows=445 width=74) (actual time=0.057..20.540 rows=981 loops=1)
                          Hash Cond: ((sc.product_id)::text = (p.product_id)::text)
                          ->  Seq Scan on shopping_carts sc  (cost=0.00..1380.59 rows=48959 width=111) (actual time=0.010..7.905 rows=48961 loops=1)
                          ->  Hash  (cost=11.38..11.38 rows=1 width=90) (actual time=0.027..0.029 rows=1 loops=1)
                                Buckets: 1024  Batches: 1  Memory Usage: 9kB
                                ->  Seq Scan on products p  (cost=0.00..11.38 rows=1 width=90) (actual time=0.013..0.023 rows=1 loops=1)
                                      Filter: (category_id = 10)
                                      Rows Removed by Filter: 49
                    ->  Memoize  (cost=0.15..0.17 rows=1 width=35) (actual time=0.001..0.001 rows=1 loops=981)
                          Cache Key: sc.seller_id
                          Cache Mode: logical
                          Hits: 961  Misses: 20  Evictions: 0  Overflows: 0  Memory Usage: 4kB
                          ->  Index Only Scan using sellers_pkey on sellers s  (cost=0.14..0.16 rows=1 width=35) (actual time=0.003..0.003 rows=1 loops=20)
                                Index Cond: (seller_id = (sc.seller_id)::text)
                                Heap Fetches: 20
              ->  Index Scan using orders_pkey on orders o  (cost=0.41..0.47 rows=1 width=45) (actual time=0.006..0.006 rows=1 loops=981)
                    Index Cond: ((order_id)::text = (sc.order_id)::text)
Planning Time: 1.141 ms
Execution Time: 28.991 ms
```

With Indexes: The introduction of indexes allows for index scans instead of sequential scans, greatly reducing the number of rows read and processed. The use of hash joins and memoization with indexed columns reduces the overall cost of join operations. The query benefits from faster data retrieval, reduced I/O operations, and overall lower execution times.

Execution Time: 20.981 ms

Execution Plan:

Sort: Similar to the previous plan, but with improved performance due to indexing.

HashAggregate: This step aggregates the data with a memory usage of 73 kB.
Nested Loop: The nested loop joins are still present but benefit from the indexes.

Index Scan on shopping_carts: The use of indexes replaces the sequential scans, significantly reducing the number of rows read.

Index Scan on products: Indexed scans filter the products table by category_id, reducing the number of rows processed.

Hash Join: Hash joins on indexed columns improve the efficiency of the join operations.

QUERY PLAN
Sort (cost=2056.33..2058.78 rows=979 width=51) (actual time=20.770..20.773 rows=59 loops=1)
Sort Key: (date_trunc('month'::text, o.order_time)), (count(*)) DESC
Sort Method: quicksort Memory: 29kB
-> HashAggregate (cost=1995.46..2007.70 rows=979 width=51) (actual time=20.703..20.715 rows=59 loops=1)
Group Key: s.seller_id, date_trunc('month'::text, o.order_time)
Batches: 1 Memory Usage: 73kB
-> Nested Loop (cost=3.57..1988.12 rows=979 width=43) (actual time=0.145..20.263 rows=981 loops=1)
-> Hash Join (cost=3.15..1526.23 rows=979 width=72) (actual time=0.111..14.889 rows=981 loops=1)
Hash Cond: ((sc.seller_id)::text = (s.seller_id)::text)
-> Hash Join (cost=1.64..1521.66 rows=979 width=74) (actual time=0.053..14.446 rows=981 loops=1)
Hash Cond: ((sc.product_id)::text = (p.product_id)::text)
-> Seq Scan on shopping_carts sc (cost=0.00..1380.61 rows=48961 width=111) (actual time=0.006..5.464 rows=48961 loops=1)
-> Hash (cost=1.62..1.62 rows=1 width=90) (actual time=0.025..0.025 rows=1 loops=1)
Buckets: 1024 Batches: 1 Memory Usage: 9kB
-> Seq Scan on products p (cost=0.00..1.62 rows=1 width=90) (actual time=0.010..0.020 rows=1 loops=1)
Filter: (category_id = 10)
Rows Removed by Filter: 49
-> Hash (cost=1.23..1.23 rows=23 width=35) (actual time=0.035..0.035 rows=23 loops=1)
Buckets: 1024 Batches: 1 Memory Usage: 10kB
-> Seq Scan on sellers s (cost=0.00..1.23 rows=23 width=35) (actual time=0.012..0.018 rows=23 loops=1)
-> Index Scan using idx_orders_order_id on orders o (cost=0.41..0.47 rows=1 width=45) (actual time=0.005..0.005 rows=1 loops=981)
Index Cond: ((order_id)::text = (sc.order_id)::text)
Planning Time: 4.412 ms
Execution Time: 20.981 ms