CENG 352 - Database Management Systems
2023-2
Mini Project 2 Report

Anıl Eren Göçer
e2448397@ceng.metu.edu.tr

May 18, 2024

4.2.1 Transaction Isolation Levels

I have taken into account the following table while selecting the isolation levels. The table shows which violations are allowed and which are prevented in each isolation levels:

Violation / Isolation Level	Dirty Read	Non-repeatable Read	Phantom
READ UNCOMMITTED	✓	✓	✓
READ COMMITTED	x	✓	✓
REPEATABLE READ	x	x	✓
SERIALIZABLE	x	x	x

Sign Up

I would use the isolation level **READ COMMITTED** for **sign up** action, because during the execution of sign up transaction, a sign in action might be started in another transaction and if we do not suspend the sign in action until sign up is committed, we suffer from dirty read problem.

Sign In

I would use the isolation level **REPEATABLE READ** for **sign in** action, because it is critical to prevent non-repeatable reads for the current session count data.

Sign Out

I would use the isolation level **REPEATABLE READ** for **sign out** action, because it is critical to prevent non-repeatable reads for the current session count data.

Show Plans

I would use the isolation level **READ UNCOMMITTED** for **show plans** action, because we do not have any actions that updates the plans table, so using read uncommitted level is enough and safe.

Show Subscription

I would use the isolation level **REPEATABLE READ** for **show subscription** action, because it is critical to prevent non-repeatable reads for the subscription plan data. If we do not use this level we may suffer from the non-repeatable reads caused by subscribe to a new plan actions in other transactions.

Change Product Stock

I would use the isolation level **SERIALIZABLE** for **change product stock** action, because this action may add a new record (insert) which may cause a phantom problem. And only way to solve phantom problem is using serializable isolation level.

Subscribe To A New Plan

I would use the isolation level **REPEATABLE READ** for **subscribe to a new plan** action, because it is critical to prevent non-repeatable reads for the subscription plan data. If we do not use this level we may suffer from the non-repeatable reads in subscribe action in other transactions.

Ship Orders

I would use the isolation level **READ COMMITTED** for **ship orders** action, because this action changes the stock data it might cause dirty read problems.

Quit

I would use the isolation level **REPEATABLE READ** for **quit** action, because it is critical to prevent non-repeatable reads for the current session count data.

Show Customer Cart

I would use the isolation level **READ COMMITTED** for **show customer cart** action, because customer cart data might be changing due to change customer cart. So, we can avoid this using read committed isolation level.

Change Customer Cart

I would use the isolation level **REPEATABLE READ** for **change customer cart** action, because this action requires reading multiple times and that can be changed by purchase cart action, there is a possibility of non-repeatable read.

Purchase Cart

I would use the isolation level **READ COMMITTED** for **purchase cart** action, because this action reads the stock data which might be changed by ship orders actions in another transaction.

4.2.3 Transaction Isolation Levels Experiment

Explanation of the experiment process

- Start reader code.
- Start writer code.
- Perform read before committing write.
- Perform write.
- Commit write.
- Perform read after committing write.

This procedure has been repeated for all three isolation levels.

I inserted a different plan for each isolation level.

Comments about the observations

In this experiment, I have clearly observed **non-repeatable read** problem in the **READ UNCOMMITTED** isolation level. As you can see in the next pages, the record (4, new_plan _read _committed, 8) is not available in the read output before write commit and is available in the read output after write commit.

This problem is solved in the isolation levels **REPEATABLE READ** and **SERIALIZABLE**. You can see that read output before and after the write commit are the same in this cases in the next pages.

These observations are consistent with the table I provided in the previous question.
The **non-repeatable read** problem is solved in the isolation levels **REPEATABLE READ** and **SERIALIZABLE** thanks to **Long Duration Read Locks** in the implementation of these isolation levels.

Outputs of mp2_transaction_reader_.py

```
---------- Running experiment with isolation level: READ COMMITTED ----------

Plans before writer commits (READ COMMITTED):

#|Name|Max Sessions
1|Basic|2
2|Advanced|4
3|Premium|6

Hit enter after writer commits to read plans again...

Plans after writer commits (READ COMMITTED):

#|Name|Max Sessions
1|Basic|2
2|Advanced|4
3|Premium|6
4|new_plan_read_committed|8

Database connection closed

Experiment completed. Hit enter to proceed to the next isolation level...


---------- Running experiment with isolation level: REPEATABLE READ ----------

Plans before writer commits (REPEATABLE READ):

#|Name|Max Sessions
1|Basic|2
2|Advanced|4
3|Premium|6
4|new_plan_read_committed|8

Hit enter after writer commits to read plans again...

Plans after writer commits (REPEATABLE READ):

#|Name|Max Sessions
1|Basic|2
2|Advanced|4
3|Premium|6
4|new_plan_read_committed|8

Database connection closed

Experiment completed. Hit enter to proceed to the next isolation level...
```

```
---------- Running experiment with isolation level: SERIALIZABLE ----------

Plans before writer commits (SERIALIZABLE):

#|Name|Max Sessions
1|Basic|2
2|Advanced|4
3|Premium|6
4|new_plan_read_committed|8
5|new_plan_repeatable_read|10

Hit enter after writer commits to read plans again...

Plans after writer commits (SERIALIZABLE):

#|Name|Max Sessions
1|Basic|2
2|Advanced|4
3|Premium|6
4|new_plan_read_committed|8
5|new_plan_repeatable_read|10

Database connection closed

Experiment completed. Hit enter to proceed to the next isolation level...
```

Outputs of mp2_transaction_writer_.py

```
Running experiment with isolation level: READ COMMITTED
Inserted new_plan_read_committed but not committed yet (READ COMMITTED). Hit enter to commit

Transaction committed (READ COMMITTED). Hit enter to finish...

Database connection closed
Experiment completed. Hit enter to proceed to the next isolation level...


Running experiment with isolation level: REPEATABLE READ
Inserted new_plan_repeatable_read but not committed yet (REPEATABLE READ). Hit enter to comm

Transaction committed (REPEATABLE READ). Hit enter to finish...

Database connection closed
Experiment completed. Hit enter to proceed to the next isolation level...


Running experiment with isolation level: SERIALIZABLE
Inserted new_plan_serializable but not committed yet (SERIALIZABLE). Hit enter to commit...

Transaction committed (SERIALIZABLE). Hit enter to finish...

Database connection closed
Experiment completed. Hit enter to proceed to the next isolation level...
```

4.2.3 Indexes

Here is the query that I wrote for the statistics:

```
SELECT sc.seller_id, pc.name , to_char(O.order_time, 'MM'), COUNT(*)
FROM shopping_carts sc, orders o, products p, product_category pc
WHERE sc.product_id = p.product_id
AND sc.order_id = o.order_id
AND p.category_id = pc.category_id
GROUP BY (sc.seller_id, pc.name, to_char(O.order_time, 'MM'));
```

Here are the indexes that I wrote to speed up the query:

```
CREATE INDEX IF NOT EXISTS my_index_shopping_cart_product_id ON shopping_carts USING HASH
    (product_id);
CREATE INDEX IF NOT EXISTS my_index_shopping_cart_order_id ON shopping_carts USING HASH
    (order_id);
CREATE INDEX IF NOT EXISTS my_index_shopping_cart_seller_id ON shopping_carts USING HASH
    (seller_id);
CREATE INDEX IF NOT EXISTS my_index_product_product_id ON products USING HASH (product_id);
CREATE INDEX IF NOT EXISTS my_index_seller_seller_id ON sellers USING HASH (seller_id);
CREATE INDEX IF NOT EXISTS my_index_category_category_id ON product_category USING HASH
    (category_id);
CREATE INDEX IF NOT EXISTS my_index_orders_order_id ON orders USING HASH (order_id);
```

Note that PostgreSQL creates default indexes for primary keys. Since those indexes cannot be dropped, I recreated the tables by removing primary and foreign key constraints so that I can notice the time differences between the case with indexes and without indexes.

For each case, I have run the query 10 times and taken the average of execution time:

Without indexes: 0.086s
With indexes: 0.071s

As you can see from the execution times, we gain important speed up by using the indexes.

Now, let's inspect execution plans for the query execution with and without indexes in the next pages.

Execution plan for the query execution without indexes:

EXPLAIN SELECT sc.seller_id, pc.name , to_ch | Enter a SQL expression to filter results (use Ctrl+Space)

Grid | Text

	QUERY PLAN
1	GroupAggregate (cost=20361.01..21584.99 rows=48959 width=593)
2	Group Key: sc.seller_id, pc.name, (to_char(o.order_time, 'MM'::text))
3	-> Sort (cost=20361.01..20483.41 rows=48959 width=585)
4	Sort Key: sc.seller_id, pc.name, (to_char(o.order_time, 'MM'::text))
5	-> Hash Join (cost=810.92..3660.28 rows=48959 width=585)
6	Hash Cond: ((sc.order_id)::text = (o.order_id)::text)
7	-> Hash Join (cost=26.88..2080.65 rows=48959 width=590)
8	Hash Cond: ((sc.product_id)::text = (p.product_id)::text)
9	-> Seq Scan on shopping_carts sc (cost=0.00..1380.59 rows=48959 width=111)
10	-> Hash (cost=25.50..25.50 rows=110 width=606)
11	-> Hash Join (cost=12.47..25.50 rows=110 width=606)
12	Hash Cond: (pc.category_id = p.category_id)
13	-> Seq Scan on product_category pc (cost=0.00..11.40 rows=140 width=520)
14	-> Hash (cost=11.10..11.10 rows=110 width=94)
15	-> Seq Scan on products p (cost=0.00..11.10 rows=110 width=94)
16	-> Hash (cost=534.02..534.02 rows=20002 width=45)
17	-> Seq Scan on orders o (cost=0.00..534.02 rows=20002 width=45)

Figure 1: Query Execution Plan Without Indexes by EXPLAIN keyword in PostgreSQL

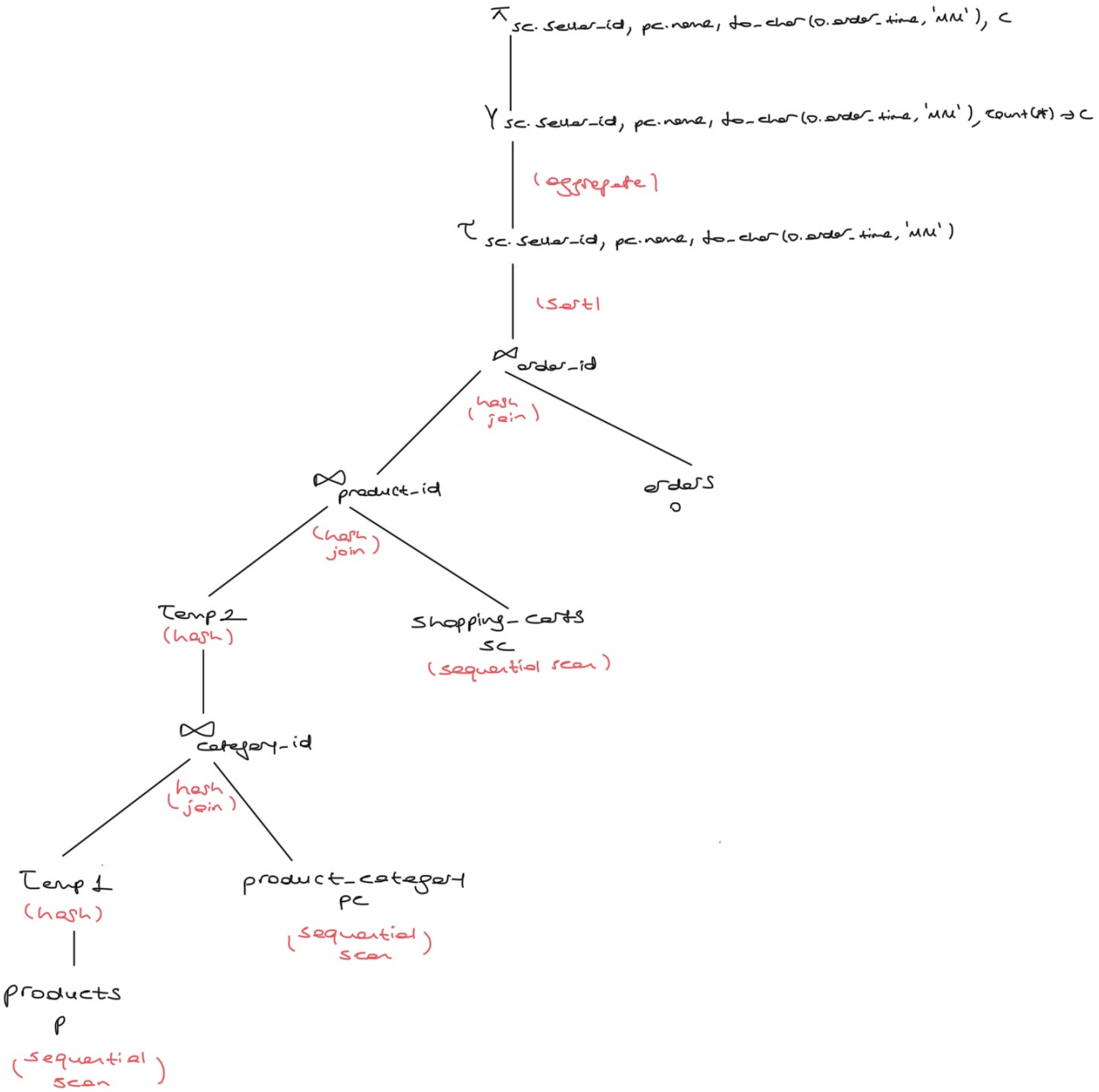

Figure 2: Visualization of the Query Execution Plan Without Indexes

Execution plan for the query execution with indexes:

EXPLAIN SELECT sc.seller_id, pc.name , to_ch | Enter a SQL expression to filter results (use Ctrl+Space)

Grid | Text

	QUERY PLAN
1	GroupAggregate (cost=8006.33..8398.01 rows=15667 width=593)
2	Group Key: sc.seller_id, pc.name, (to_char(o.order_time, 'MM'::text))
3	-> Sort (cost=8006.33..8045.50 rows=15667 width=585)
4	Sort Key: sc.seller_id, pc.name, (to_char(o.order_time, 'MM'::text))
5	-> Nested Loop (cost=3.41..2788.20 rows=15667 width=585)
6	-> Hash Join (cost=3.41..1724.26 rows=15667 width=590)
7	Hash Cond: ((sc.product_id)::text = (p.product_id)::text)
8	-> Seq Scan on shopping_carts sc (cost=0.00..1380.59 rows=48959 width=111)
9	-> Hash (cost=3.21..3.21 rows=16 width=606)
10	-> Hash Join (cost=1.36..3.21 rows=16 width=606)
11	Hash Cond: (p.category_id = pc.category_id)
12	-> Seq Scan on products p (cost=0.00..1.50 rows=50 width=94)
13	-> Hash (cost=1.16..1.16 rows=16 width=520)
14	-> Seq Scan on product_category pc (cost=0.00..1.16 rows=16 width=520)
15	-> Index Scan using my_index_orders_order_id on orders o (cost=0.00..0.06 rows=1 width=45)
16	Index Cond: ((order_id)::text = (sc.order_id)::text)

Figure 3: Query Execution Plan With Indexes by EXPLAIN keyword in PostgreSQL

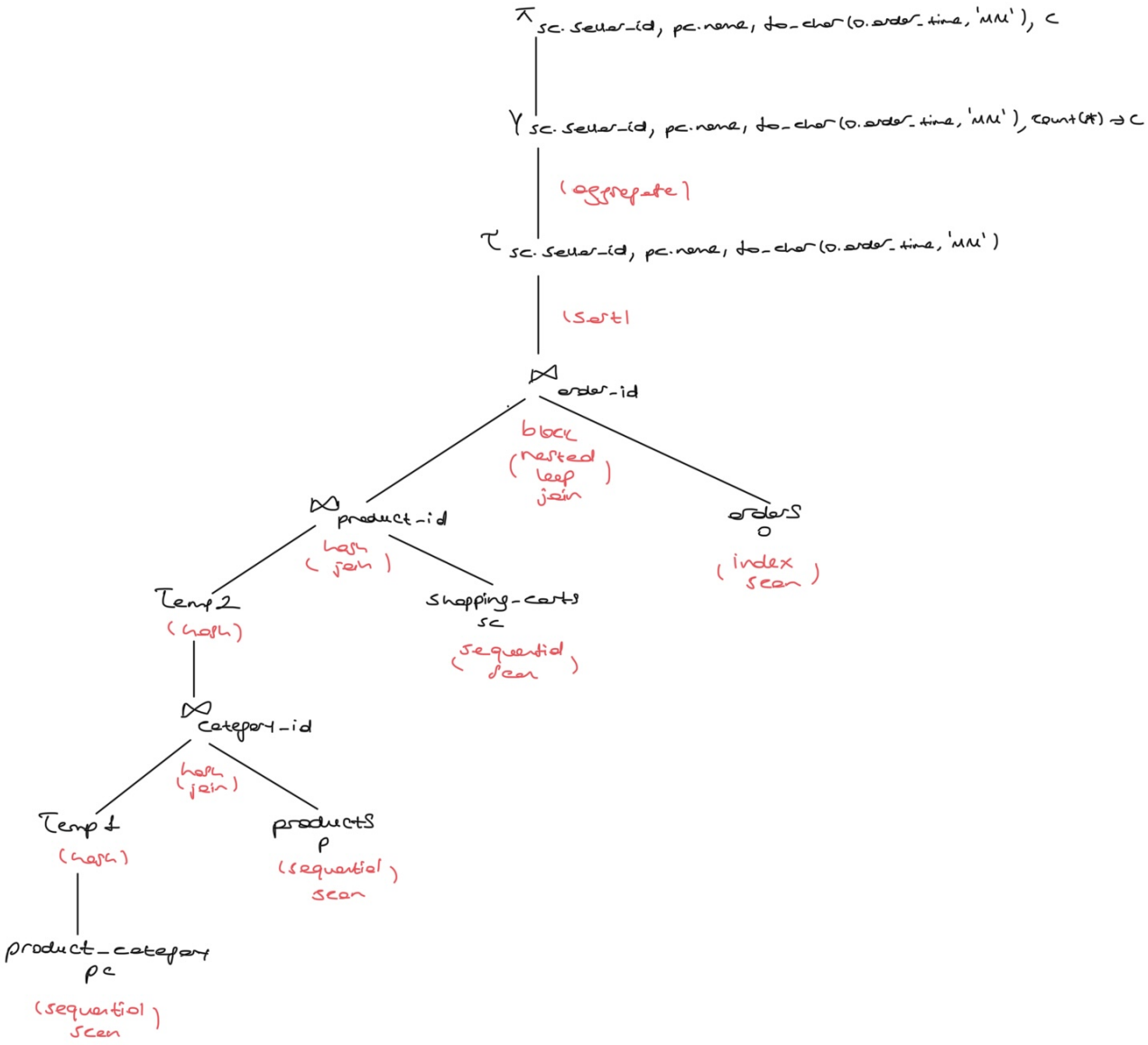

Figure 4: Visualization of the Query Execution Plan With Indexes