CSC 370 Project Final Sprint

July 14th - July 28th 2024
By Carolina Kierulff & Makayla Savege

Overview from last Sprint

- Overview:
    - 3NF and 4NF in database
    - Construct an accurate input domain model to identify suitable unit tests
    - Reconfigure select SQL queries to be more expressive
    - Create a database connection and cursor in python to execute transactions
    - Synchronize object oriented programming in python to our relational database

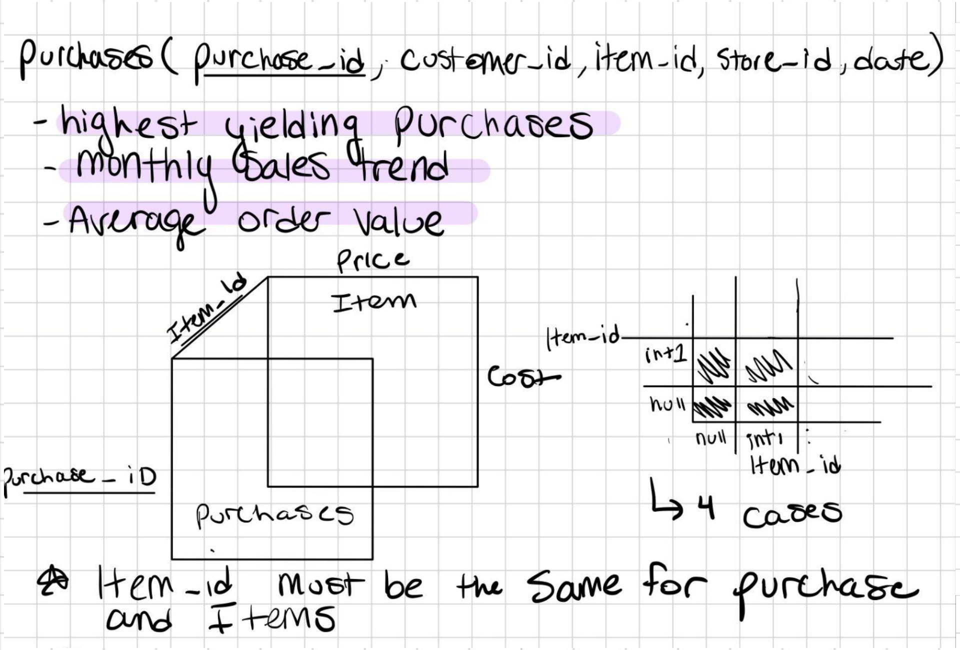
3NF and 4NF

3NF and 4NF are not needed for this model. All sets of functional dependencies are already in 3NF and 4NF. This includes all functional dependencies in the items table, the customer table, the store table, the purchases table, and the managers table.

No relation contains transitive dependencies, and there are no multivalued dependencies, all dependencies are in BCNF.

Input Domain Model

Purchases Table

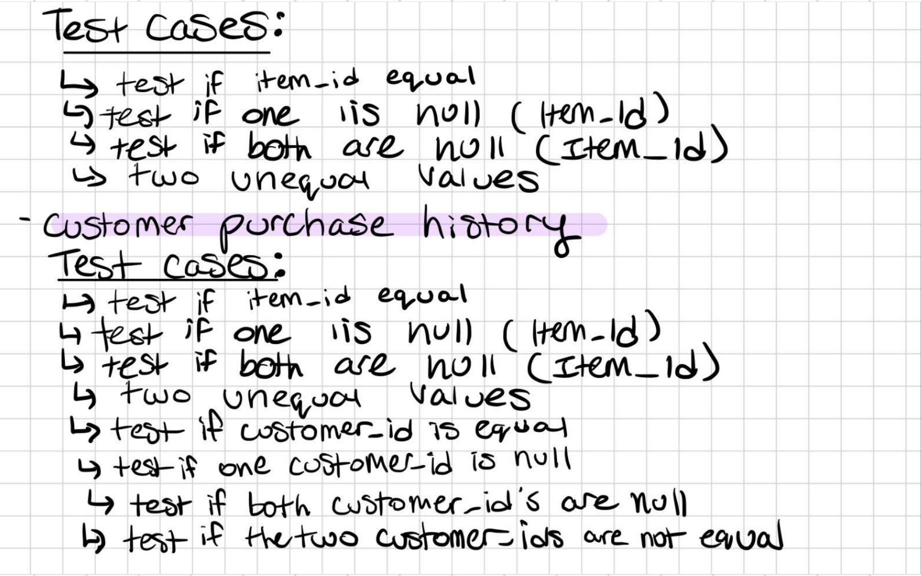


purchases(purchase_id, customer_id, item-id, store-id, date)

- highest yielding purchases
- monthly Sales trend
- Average order Value

Price
Item_id
Item
Cost
purchase_iD
Purchases
Item-id
int1
null
null
int1
Item_id
↳ 4 cases

✱ Item-id must be the Same for purchase and Items



Test Cases:

↳ test if item-id equal
↳ test if one is null (Item-Id)
↳ test if both are null (Item_Id)
↳ two unequal values

- Customer purchase history

Test cases:

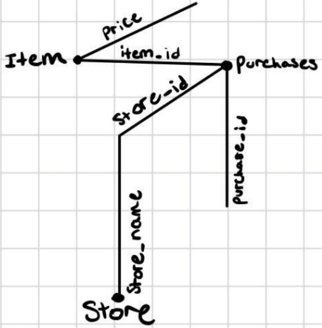
↳ test if item-id equal
↳ test if one is null (Item-Id)
↳ test if both are null (Item_Id)
↳ two unequal values
↳ test if customer-id is equal
↳ test if one customer-id is null
↳ test if both customer-id's are null
↳ test if the two customer-ids are not equal

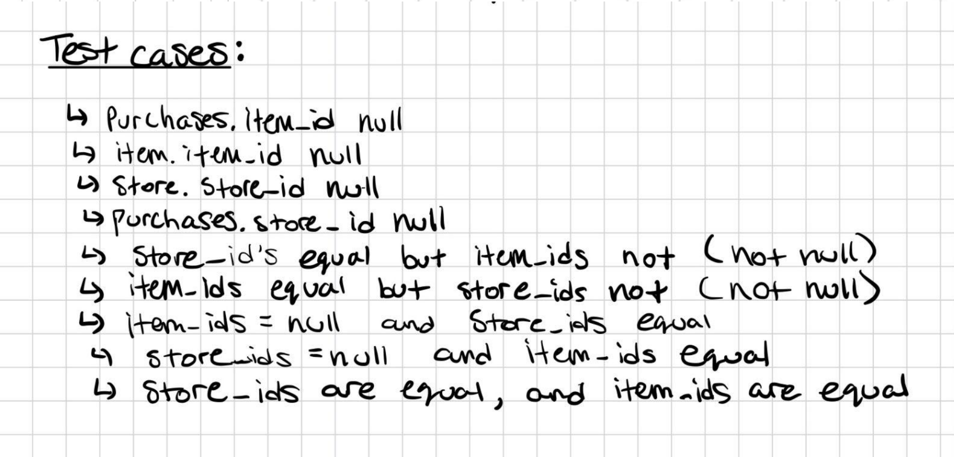
Input Domain Model

Store Table

Store(<u>store_id</u>, store_name, managr_id, location)

- Store revenue for selected store



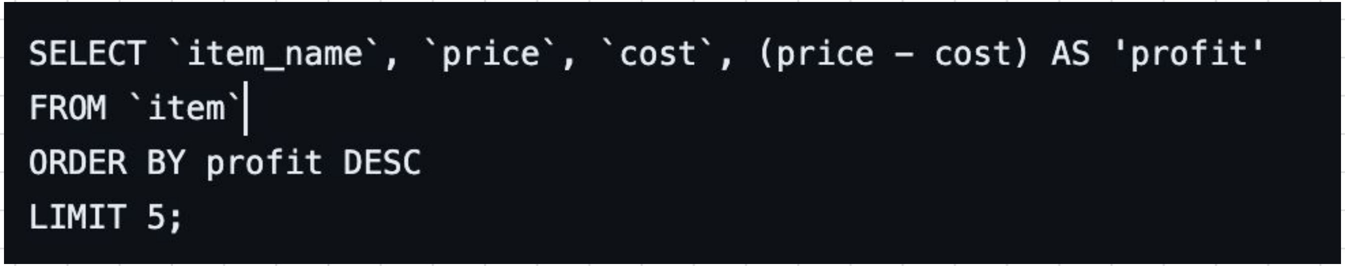

Test cases:

↳ Purchases.item_id null
↳ item.item_id null
↳ Store.Store_id null
↳ purchases.store_id null
↳ Store_id's equal but item_ids not (not null)
↳ item_ids equal but store_ids not (not null)
↳ item_ids = null and Store_ids equal
↳ store_ids = null and item_ids equal
↳ store_ids are equal, and item_ids are equal

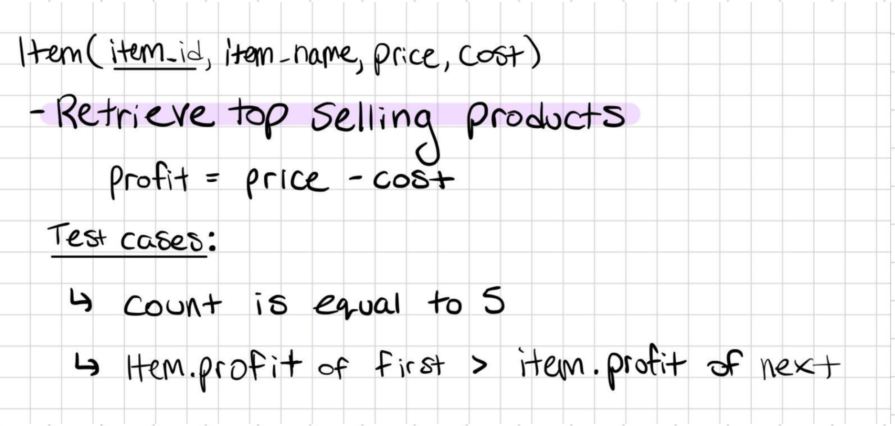
★ Store(store_id) must equal purchases(store_id)
★ Purchases(item_id) must equal Item(item_id)

Input Domain Model

Item Table



```sql
SELECT `item_name`, `price`, `cost`, (price - cost) AS 'profit'
FROM `item`
ORDER BY profit DESC
LIMIT 5;
```



Item(item_id, item_name, price, cost)

- Retrieve top selling products

    profit = price - cost

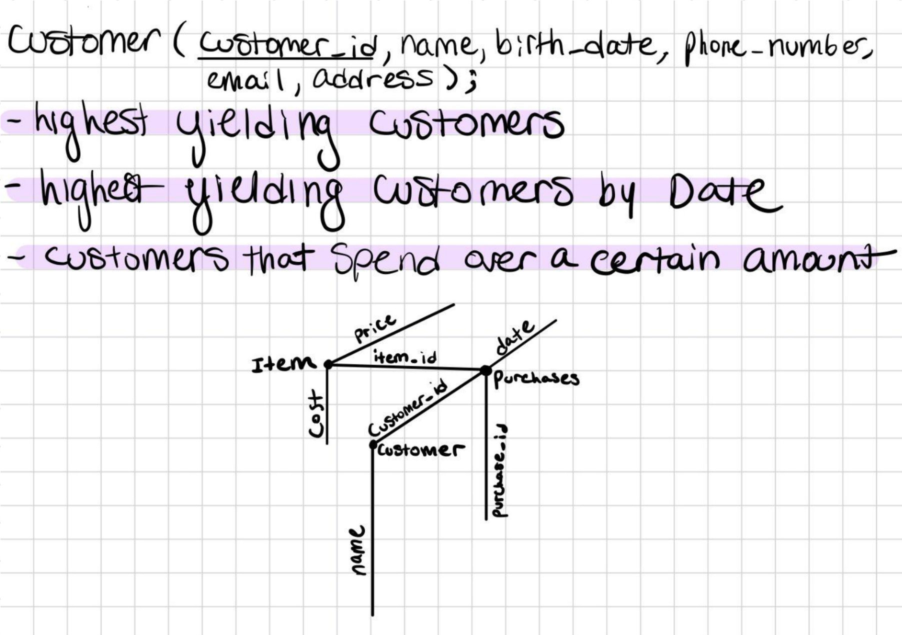
Test cases:

↳ count is equal to 5

↳ Item.profit of first > item.profit of next

Input Domain Model

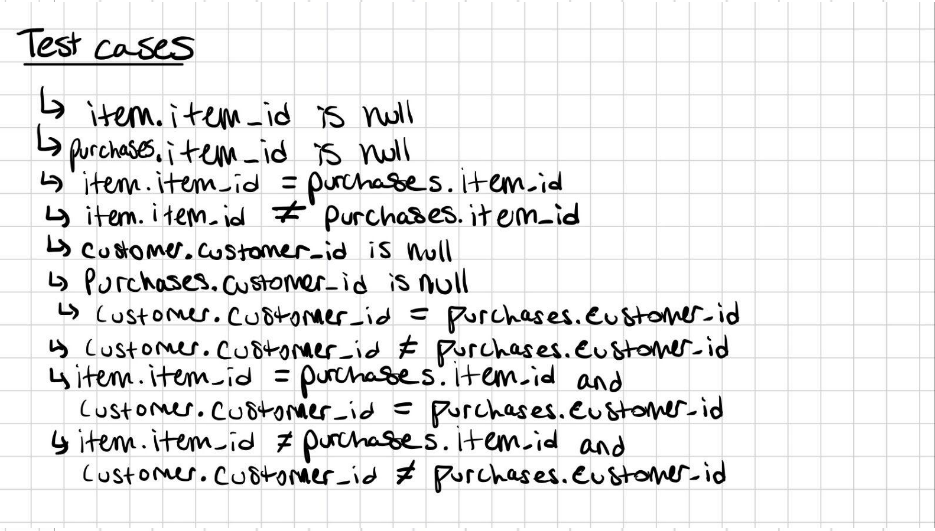
Customer Table



customer (customer_id, name, birth_date, phone_number, email, address);

- highest yielding customers
- highest yielding customers by Date
- customers that Spend over a certain amount

price
date
Item
item_id
Purchases
Cost
customer_id
Customer
purchase_id
name



Test cases

↳ item.item_id is null
↳ purchases.item_id is null
↳ item.item_id = purchases.item_id
↳ item.item_id ≠ purchases.item_id
↳ customer.customer_id is null
↳ Purchases.customer_id is null
↳ Customer.customer_id = purchases.customer_id
↳ Customer.customer_id ≠ purchases.customer_id
↳ item.item_id = purchases.item_id and Customer.customer_id = purchases.customer_id
↳ item.item_id ≠ purchases.item_id and Customer.customer_id ≠ purchases.customer_id

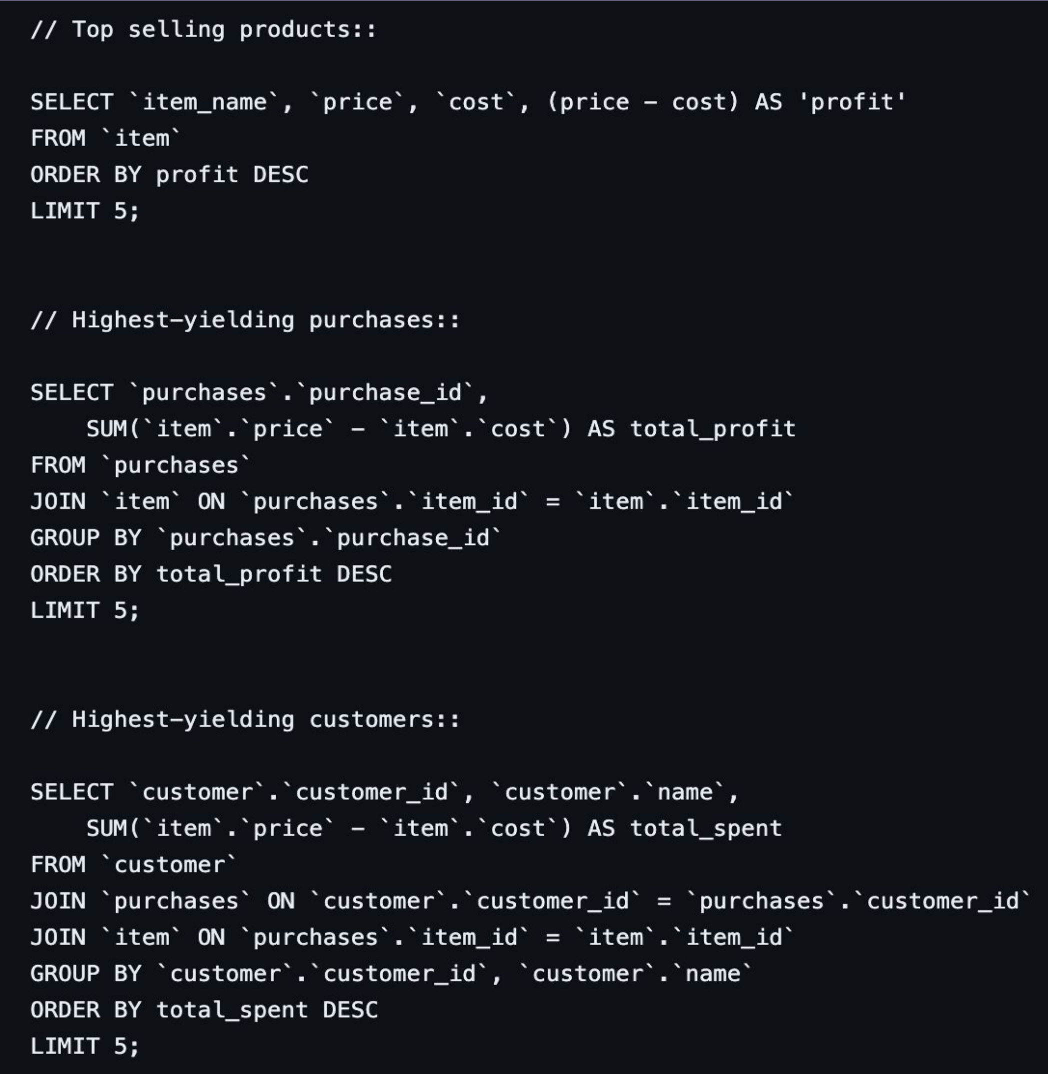
Reconfigured SQL Queries

After analysis of our complex SQL queries, it is determined that all complex queries are in their simplest form, accomplished in our previous two sprints

Two sub queries were simplified in this sprint.

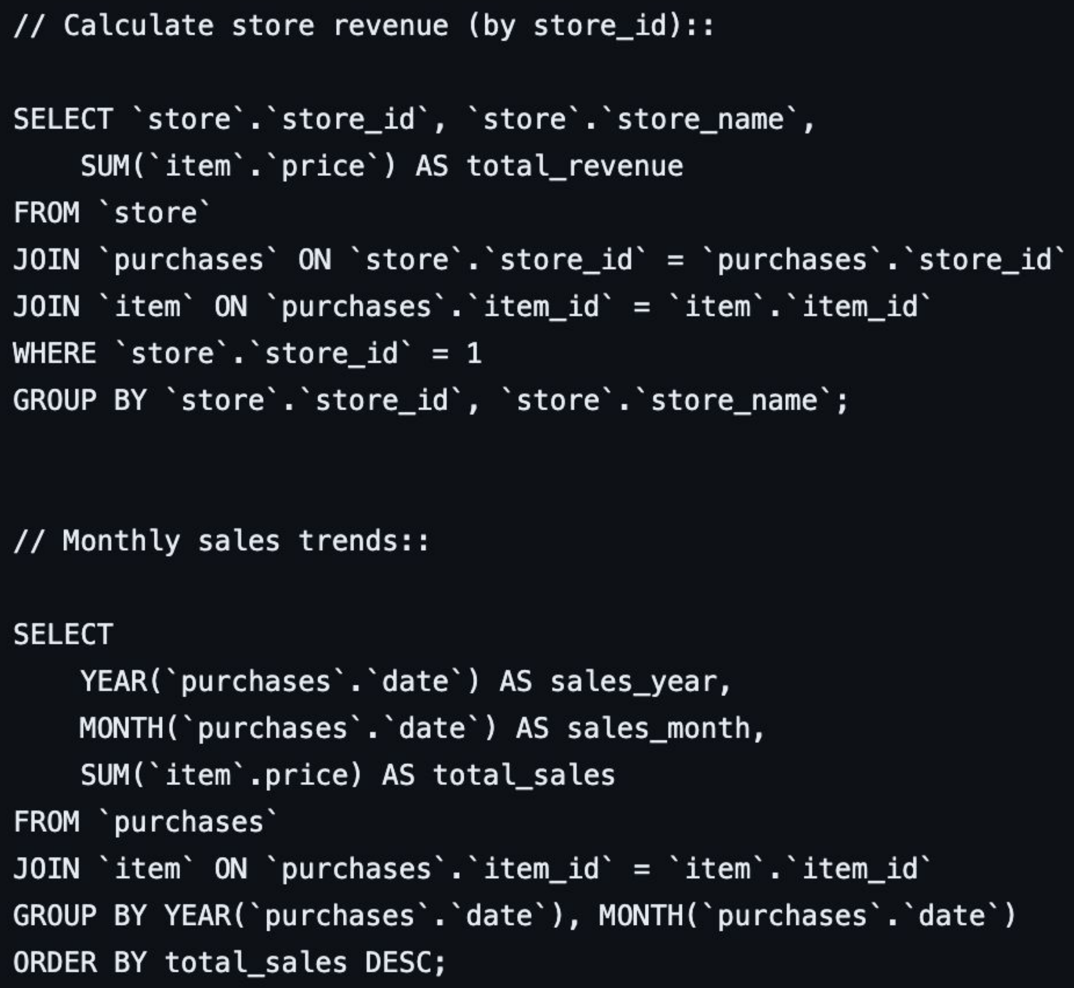
See the following complex MYSQL queries already simplified, then the two simplified sub-queries

Simplified SQL Queries



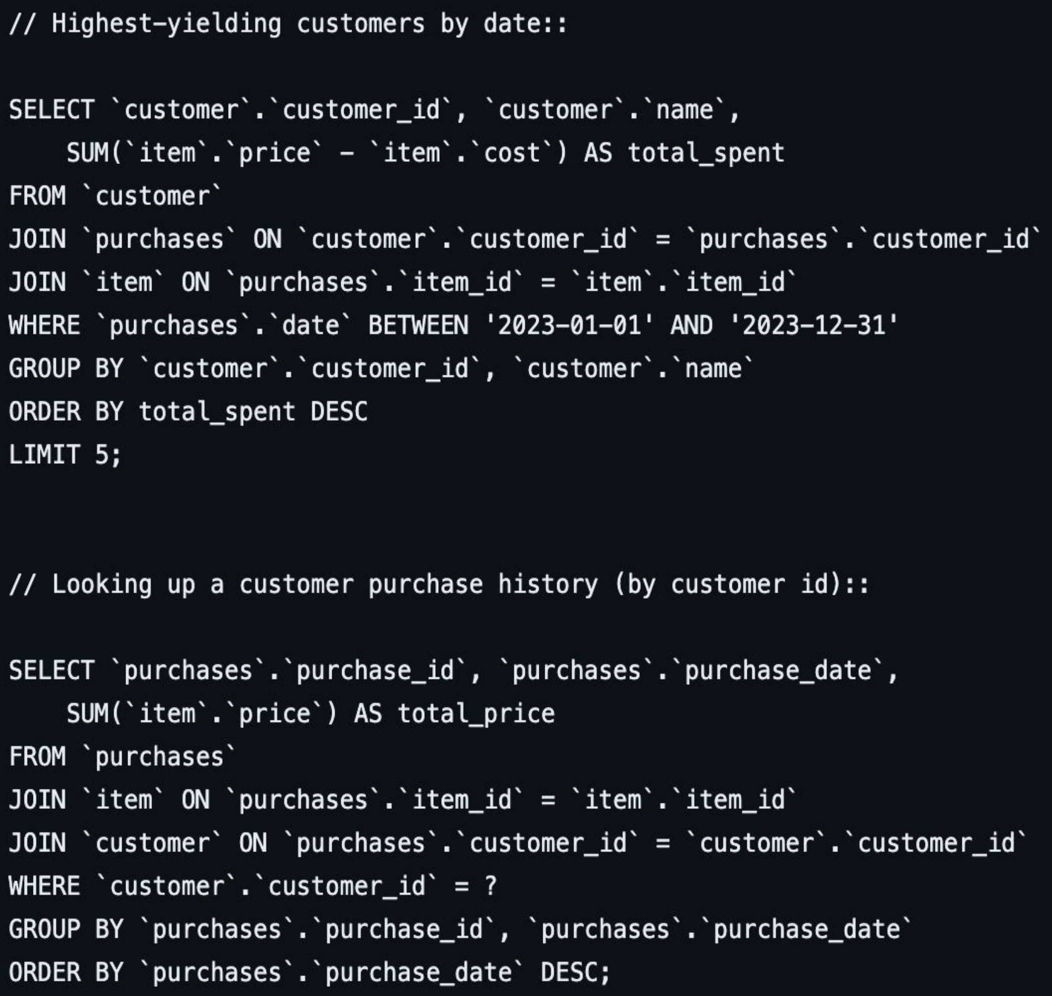
```
// Top selling products::

SELECT `item_name`, `price`, `cost`, (price - cost) AS 'profit'
FROM `item`
ORDER BY profit DESC
LIMIT 5;



// Highest-yielding purchases::

SELECT `purchases`.`purchase_id`,
    SUM(`item`.`price` - `item`.`cost`) AS total_profit
FROM `purchases`
JOIN `item` ON `purchases`.`item_id` = `item`.`item_id`
GROUP BY `purchases`.`purchase_id`
ORDER BY total_profit DESC
LIMIT 5;



// Highest-yielding customers::

SELECT `customer`.`customer_id`, `customer`.`name`,
    SUM(`item`.`price` - `item`.`cost`) AS total_spent
FROM `customer`
JOIN `purchases` ON `customer`.`customer_id` = `purchases`.`customer_id`
JOIN `item` ON `purchases`.`item_id` = `item`.`item_id`
GROUP BY `customer`.`customer_id`, `customer`.`name`
ORDER BY total_spent DESC
LIMIT 5;
```

Simplified SQL Queries



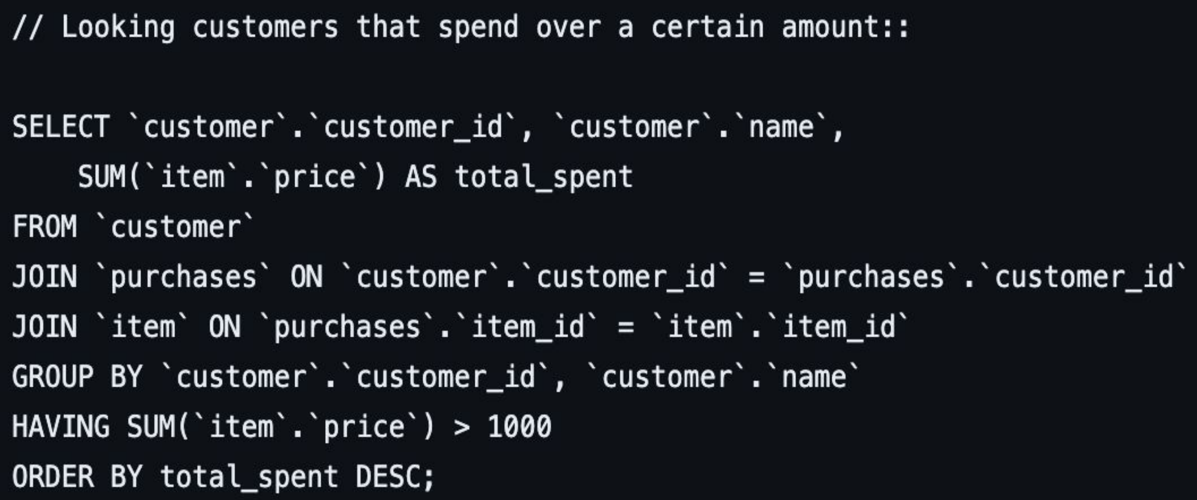
```
// Calculate store revenue (by store_id)::

SELECT `store`.`store_id`, `store`.`store_name`,
    SUM(`item`.`price`) AS total_revenue
FROM `store`
JOIN `purchases` ON `store`.`store_id` = `purchases`.`store_id`
JOIN `item` ON `purchases`.`item_id` = `item`.`item_id`
WHERE `store`.`store_id` = 1
GROUP BY `store`.`store_id`, `store`.`store_name`;



// Monthly sales trends::

SELECT
    YEAR(`purchases`.`date`) AS sales_year,
    MONTH(`purchases`.`date`) AS sales_month,
    SUM(`item`.price) AS total_sales
FROM `purchases`
JOIN `item` ON `purchases`.`item_id` = `item`.`item_id`
GROUP BY YEAR(`purchases`.`date`), MONTH(`purchases`.`date`)
ORDER BY total_sales DESC;
```

Simplified SQL Queries



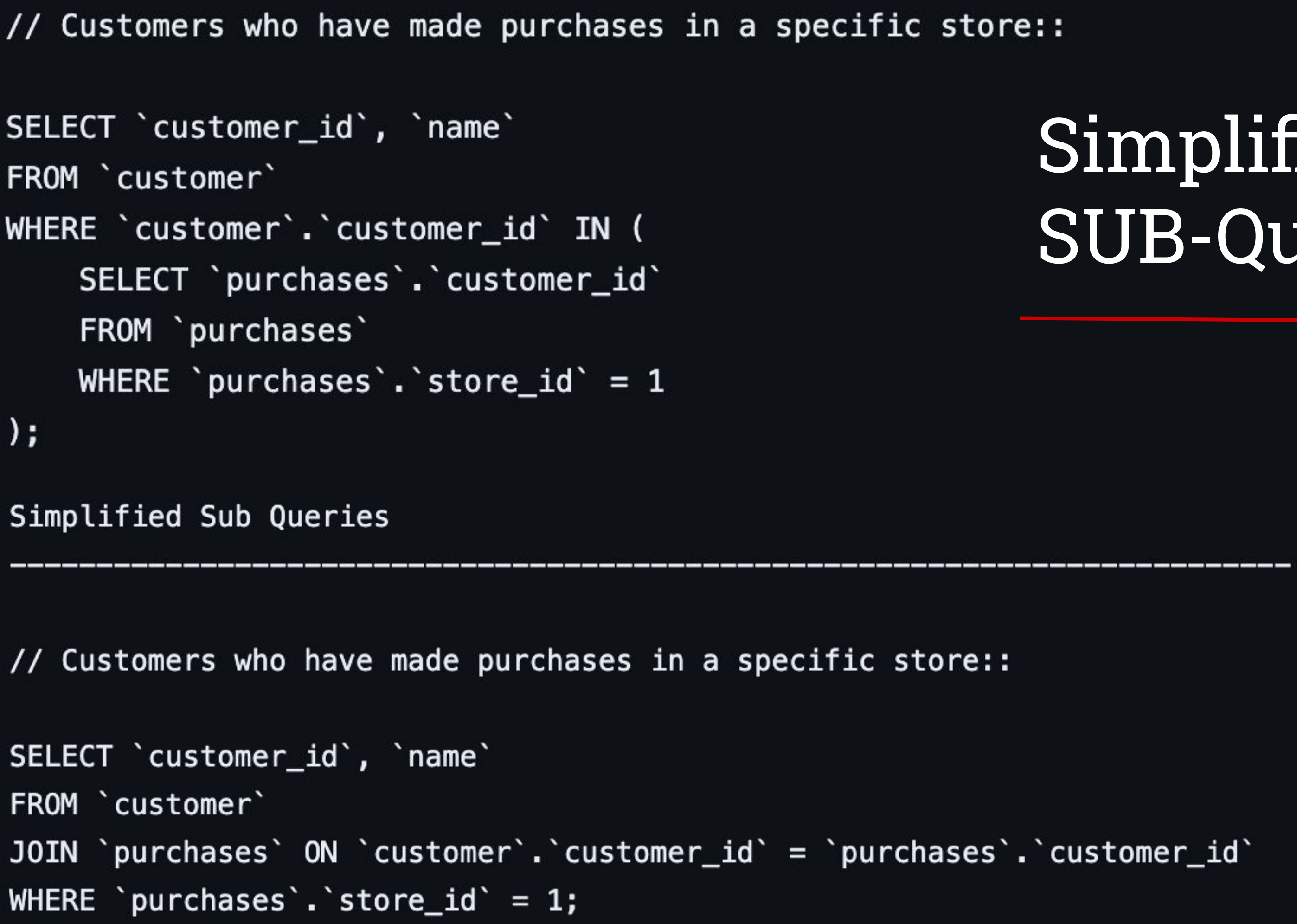
```
// Highest-yielding customers by date::

SELECT `customer`.`customer_id`, `customer`.`name`,
    SUM(`item`.`price` - `item`.`cost`) AS total_spent
FROM `customer`
JOIN `purchases` ON `customer`.`customer_id` = `purchases`.`customer_id`
JOIN `item` ON `purchases`.`item_id` = `item`.`item_id`
WHERE `purchases`.`date` BETWEEN '2023-01-01' AND '2023-12-31'
GROUP BY `customer`.`customer_id`, `customer`.`name`
ORDER BY total_spent DESC
LIMIT 5;



// Looking up a customer purchase history (by customer id)::

SELECT `purchases`.`purchase_id`, `purchases`.`purchase_date`,
    SUM(`item`.`price`) AS total_price
FROM `purchases`
JOIN `item` ON `purchases`.`item_id` = `item`.`item_id`
JOIN `customer` ON `purchases`.`customer_id` = `customer`.`customer_id`
WHERE `customer`.`customer_id` = ?
GROUP BY `purchases`.`purchase_id`, `purchases`.`purchase_date`
ORDER BY `purchases`.`purchase_date` DESC;
```

Simplified SQL Queries



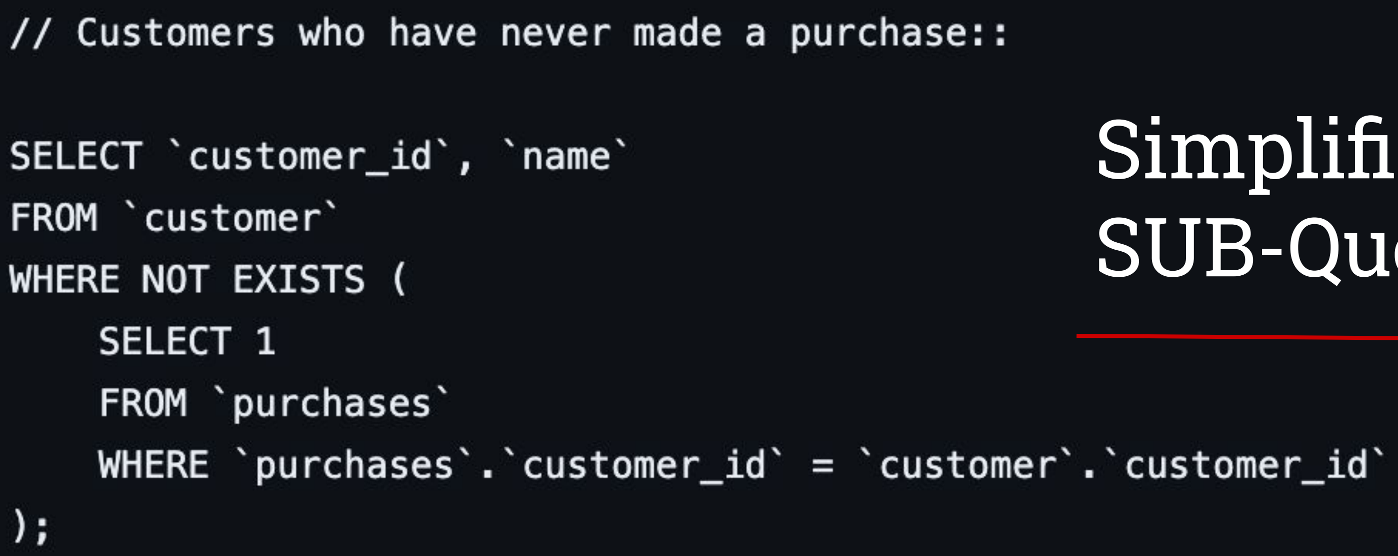
```
// Looking customers that spend over a certain amount::

SELECT `customer`.`customer_id`, `customer`.`name`,
    SUM(`item`.`price`) AS total_spent
FROM `customer`
JOIN `purchases` ON `customer`.`customer_id` = `purchases`.`customer_id`
JOIN `item` ON `purchases`.`item_id` = `item`.`item_id`
GROUP BY `customer`.`customer_id`, `customer`.`name`
HAVING SUM(`item`.`price`) > 1000
ORDER BY total_spent DESC;
```

Simplified SQL SUB-Queries

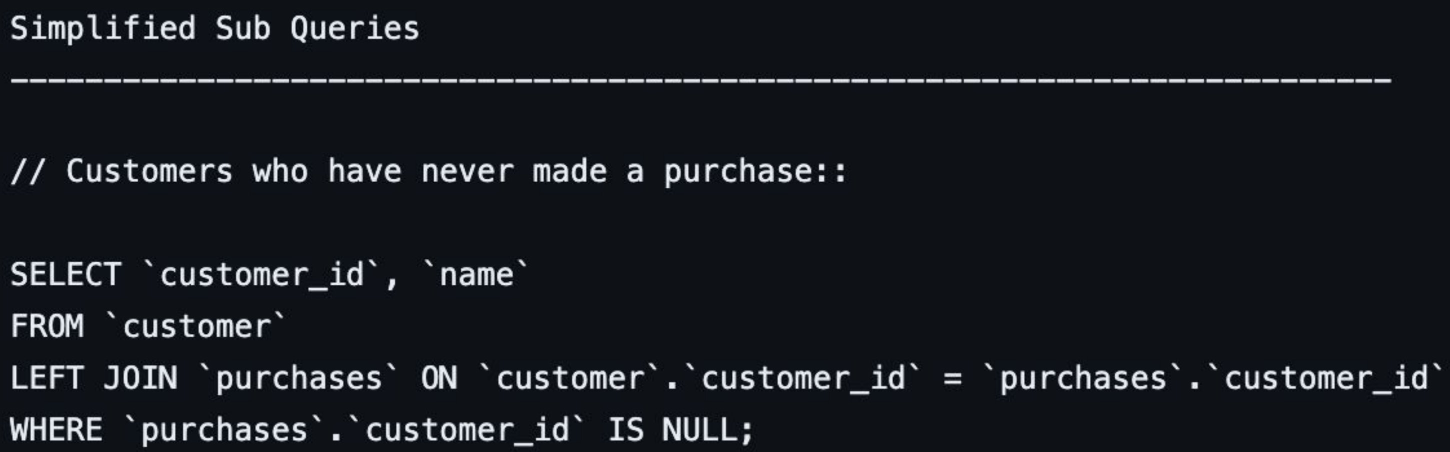


```
// Customers who have made purchases in a specific store::

SELECT `customer_id`, `name`
FROM `customer`
WHERE `customer`.`customer_id` IN (
    SELECT `purchases`.`customer_id`
    FROM `purchases`
    WHERE `purchases`.`store_id` = 1
);

Simplified Sub Queries
--------------------------------------------------------------------------


// Customers who have made purchases in a specific store::

SELECT `customer_id`, `name`
FROM `customer`
JOIN `purchases` ON `customer`.`customer_id` = `purchases`.`customer_id`
WHERE `purchases`.`store_id` = 1;
```

Simplified SQL SUB-Queries



```
// Customers who have never made a purchase::

SELECT `customer_id`, `name`
FROM `customer`
WHERE NOT EXISTS (
    SELECT 1
    FROM `purchases`
    WHERE `purchases`.`customer_id` = `customer`.`customer_id`
);
```



```
Simplified Sub Queries
----------------------------------------------------------------------------

// Customers who have never made a purchase::

SELECT `customer_id`, `name`
FROM `customer`
LEFT JOIN `purchases` ON `customer`.`customer_id` = `purchases`.`customer_id`
WHERE `purchases`.`customer_id` IS NULL;
```

DB Connection in Python

- Initially install DB connector

  pip install mysql-connector-python
- Connect to the MYSQL Database by creating a class
- Create a class to execute queries in Python
- Connect to DB and perform operations through OOP in Python

We were able to install the DB connector, but without the lessons that were executed beyond July 28th, this goal was not in scope for our final sprint

Final Remarks

- No future sprint goals
- Only one goal from the last sprint was not reached due to class time constraints
- The overall goal to achieve the course level competency of Back-end Engineering will continue to be a work in progress beyond this class
- Thank you to Ninad and Sean for all your hard work this semester, and for setting up the course in a way that allowed us to execute what we learn from you in real time