CZ2007 Lab 5 Report

DSS2 Group 2

Authors: An Ruyi, Cheng Zhengxing, Christopher Arif Setiadharma,
Peng Wenxuan, Zhang Tianyu, Zhou Runbing

APPENDIX C: INDIVIDUAL CONTRIBUTION FORM

Name	Individual Contribution to Submission 1 (Lab 1)	Percentage of Contribution	Signature
cheng zhengxing	join group discussion, suggest possible improvements, modify the details of ER graphics	16.66%	
zhang tianyu	join group discussion, suggest possible improvements, modify the details of ER graphics	16.66%	
zhou runbing	join group discussion, suggest possible improvements, modify the details of ER graphics	16.66%	
chirstopher arif setiadharma	double-checking the relationships and entity attributes	16.66%	
an ruyi	design the ER relation, create skeleton of design, create ER diagram, write report, join group discussion	16.66%	
peng wenxuan	design the ER relation, create skeleton of design, create ER diagram, write report, join group discussion	16.66%	

Name	Individual Contribution to Submission 2 (Lab 3)	Percentage of Contribution	Signature
cheng zhenxing	Write up for entities sets, analyze their attributes and FDs, join group discussion	16.66%	
zhang tianyu	Write up for entities sets, analyze their attributes and FDs, join group discussion	16.66%	
zhou runbing	Write up for entities sets, analyze their attributes and FDs, join group discussion	16.66%	
chirstopher arif setiadharma	Write up for entities sets, analyze their attributes and FDs, join group discussion	16.66%	
an ruyi	Write up for entities sets, analyze their attributes and FDs, join group discussion	16.66%	
peng wenxuan	Write up for entities sets, analyze their attributes and FDs, join group discussion	16.66%	

Name	Individual Contribution to Submission 3 (Lab 5)	Percentage of Contribution	Signature
Cheng Zhengxing	write queries	16.66%	
Zhang Tianyu	write queries	16.66%	
ZHOU Runbing	write queries	16.66%	
Christopher Arif	Write report	16.66 %	
An Ruyi	create table, write queries	16.66 %	
Peng Wenxuan	data insertion	16.66 %	

Table of contents

1. Table Creation

```
-- Create Tables
CREATE TABLE users
(
   user_id   int          NOT NULL IDENTITY (1,1) PRIMARY KEY,
   user_name varchar(256) NULL
);

CREATE TABLE product
(
   product_name nvarchar(256) NOT NULL PRIMARY KEY,
   category     varchar(256)  NOT NULL,
   maker        nvarchar(256) NOT NULL
);

CREATE TABLE employee
(
   employee_id   int           NOT NULL IDENTITY (1,1) PRIMARY KEY,
   employee_name nvarchar(256) NOT NULL,
   salary        float(24)     NOT NULL DEFAULT 0.0 CHECK (salary >= 0.0)
);

CREATE TABLE shop
(
   shop_name nvarchar(256) NOT NULL PRIMARY KEY,
);

CREATE TABLE orders
(
   order_id                int           NOT NULL IDENTITY (1,1) PRIMARY KEY,
   total_shipping_cost     float(24)     NOT NULL DEFAULT 0.0 CHECK
(total_shipping_cost >= 0.0),
   shipping_addr           nvarchar(256) NOT NULL,
   order_placing_timestamp datetime              DEFAULT getdate(),
   user_id                 int FOREIGN KEY REFERENCES users (user_id) ON DELETE
CASCADE,
);

CREATE TABLE complaint
(
   complaint_id         int          NOT NULL IDENTITY (1,1) PRIMARY KEY,
   complain_description varchar(max) NOT NULL,
   file_timestamp       datetime     NOT NULL DEFAULT getdate(),
   resolved_timestamp   datetime     NULL,
```

```
   assigned_timestamp   datetime     NULL,
   complaint_status     varchar(16)           DEFAULT 'Pending'
       Check (complaint_status = 'Pending' OR
              complaint_status = 'Assigned' OR
              complaint_status = 'Resolved'),
   user_id              int          FOREIGN KEY REFERENCES users (user_id) ON
DELETE SET NULL,
   employee_id                  int          FOREIGN KEY REFERENCES employee
(employee_id) ON DELETE SET NULL,
   CHECK (file_timestamp <= assigned_timestamp AND
          assigned_timestamp <= resolved_timestamp)
);

CREATE TABLE complaint_on_shop
(
   complaint_id int FOREIGN KEY REFERENCES complaint (complaint_id) ON DELETE
CASCADE,
   shop_name    nvarchar(256) FOREIGN KEY REFERENCES shop (shop_name) ON DELETE
CASCADE ON UPDATE CASCADE
       PRIMARY KEY (complaint_id),
);

CREATE TABLE complaint_on_product
(
   complaint_id int FOREIGN KEY REFERENCES complaint (complaint_id) ON DELETE
CASCADE,
   product_name nvarchar(256) FOREIGN KEY REFERENCES product (product_name) ON
DELETE CASCADE ON UPDATE CASCADE,
   shop_name    nvarchar(256) FOREIGN KEY REFERENCES shop (shop_name) ON DELETE
CASCADE ON UPDATE CASCADE,
   order_id     int FOREIGN KEY REFERENCES orders (order_id) ON DELETE CASCADE,
   PRIMARY KEY (complaint_id),
);

CREATE TABLE product_in_shop
(
   product_name nvarchar(256) FOREIGN KEY REFERENCES product (product_name) ON
DELETE CASCADE ON UPDATE CASCADE,
   shop_name    nvarchar(256) FOREIGN KEY REFERENCES shop (shop_name) ON DELETE
CASCADE ON UPDATE CASCADE,
   quantity     int       NOT NULL DEFAULT 0 CHECK (quantity >= 0),
   price_in_shop        float(24) NOT NULL DEFAULT 0.0 CHECK (price_in_shop >=
0.0),
   PRIMARY KEY (product_name, shop_name),
);
```

```
/* error: used to be 500 and 100 does not match with previous*/
CREATE TABLE product_on_order
(
   product_name            nvarchar(256) FOREIGN KEY REFERENCES product
(product_name) ON DELETE CASCADE ON UPDATE CASCADE,
   shop_name               nvarchar(256) FOREIGN KEY REFERENCES shop (shop_name) ON
DELETE CASCADE ON UPDATE CASCADE,
   order_id                int FOREIGN KEY REFERENCES [orders] (order_id) ON DELETE
CASCADE,
   order_quantity          int         NOT NULL DEFAULT 0 CHECK (order_quantity >
0),
   dealing_price           float(24)   NOT NULL DEFAULT 0.0 CHECK (dealing_price >=
0.0),
   product_on_order_status varchar(50) NOT NULL DEFAULT 'being processed'
       Check (product_on_order_status = 'being processed' OR
              product_on_order_status = 'shipped' OR
              product_on_order_status = 'delivered' OR
              product_on_order_status = 'returned'),
   delivery_date           datetime             DEFAULT NULL,

   -- sanity check
   -- being processed / shipped items should not have a delivery date while the
rest should have one
   CHECK ((product_on_order_status = 'delivered' AND delivery_date IS NOT NULL)
       OR (product_on_order_status = 'returned' AND delivery_date IS NOT NULL)
       OR (product_on_order_status = 'being processed' AND delivery_date IS NULL)
       OR (product_on_order_status = 'shipped' AND delivery_date IS NULL)),
   PRIMARY KEY (product_name, shop_name, order_id),
);

CREATE TABLE price_history
(
   product_name nvarchar(256) FOREIGN KEY REFERENCES product (product_name) ON
DELETE CASCADE ON UPDATE CASCADE,
   shop_name    nvarchar(256) FOREIGN KEY REFERENCES shop (shop_name) ON DELETE
CASCADE ON UPDATE CASCADE,
   start_date   datetime  NOT NULL DEFAULT getdate(),
   end_date     datetime  NULL,
   actual_price float(24) NOT NULL DEFAULT 0.0 CHECK (actual_price >= 0.0),
   PRIMARY KEY (product_name, shop_name, start_date),
   CHECK (start_date < end_date),
);

CREATE TABLE feedback
```

```
(
   product_name nvarchar(256) FOREIGN KEY REFERENCES product (product_name) ON
DELETE CASCADE ON UPDATE CASCADE,
   shop_name    nvarchar(256) FOREIGN KEY REFERENCES shop (shop_name) ON DELETE
CASCADE ON UPDATE CASCADE,
   order_id     int FOREIGN KEY REFERENCES orders (order_id) ON DELETE CASCADE,
   rating       int          NOT NULL CHECK (rating >= 1 AND rating <= 5),
   comment      varchar(max) NULL,
   -- user_id      int          NOT NULL,
   -- should not be included because we have performed 3NF decomposition in lab 3
   -- mention in report!
   feedbackDate datetime DEFAULT getdate(),
   PRIMARY KEY (product_name, shop_name, order_id),
   --FOREIGN KEY (user_id) REFERENCES users (user_id) ON DELETE CASCADE ON UPDATE
CASCADE,
);









--- TRIGGERS
--Table Triggers
GO --syntax to define a batch of statements

--Update DeliveryDateTime when DeliveryStatus changed.
--If DeliveryStatus changed to 'Delivered', then DeliveryDateTime=GETDATE()
--If DeliveryStatus changed to 'Pending', then DeliveryDateTime=NULL
--One trigger, one batch of statements
CREATE TRIGGER UpdateDelivery
   ON product_on_order
   AFTER UPDATE
   NOT FOR REPLICATION
   AS
BEGIN
   UPDATE product_on_order
       --DeliveryDateTime will not be updated unless DeliveryStatus changes from
'shipped' to 'delivered'
   SET delivery_date= CASE
       --If previous DeliveryStatus='shipped' and is changed to 'delivered', then
DeliveryDateTime=GETDATE().
```

```
                        WHEN d.product_on_order_status = 'shipped' AND
i.product_on_order_status = 'delivered'
                            THEN GETDATE()
      --DeliveryDateTime retains the old value
                        ELSE
                            d.delivery_date
      END
    --DeliveryStatus will not be updated unless if follows the sequence: 'being
processed'->'shipped'->'delivered'->'returned'
    , product_on_order_status= CASE
      --If previous DeliveryStatus='being processed'. It can only be changed to
'shipped'.
                            WHEN d.product_on_order_status = 'being processed' AND
i.product_on_order_status <> 'shipped'
                              THEN 'being processed'
      --If previous DeliveryStatus='shipped'. It can only be changed to
'delivered'.
                            WHEN d.product_on_order_status = 'shipped' AND
i.product_on_order_status <> 'delivered'
                              THEN 'shipped'
      --If previous DeliveryStatus='delivered'. It can only be changed to
'returned'.
                            WHEN d.product_on_order_status = 'delivered' AND
i.product_on_order_status <> 'returned'
                              THEN 'delivered' --DeliveryStatus retains updated
value
                            WHEN d.product_on_order_status = 'returned'
                              THEN 'returned'

                            ELSE
                              i.product_on_order_status
      END
   FROM product_on_order o,
        inserted i,
        deleted d
        --Get all the records that have just been updated, and find the previous
value (inserted gives the updated rows, and deleted gives the previous values for
these rows)
   WHERE o.shop_name = i.shop_name
     AND o.product_name = i.product_name
     AND o.order_id = i.order_id
     AND o.shop_name = d.shop_name
     AND o.product_name = d.product_name
     AND o.order_id = d.order_id;
END
```

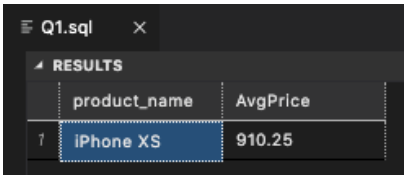
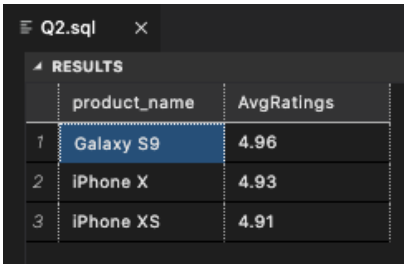
```
GO


GO
CREATE TRIGGER ComplainStatus
   ON complaint
   AFTER UPDATE
   NOT FOR REPLICATION
   AS
BEGIN
   UPDATE complaint
       SET complaint_status= CASE
                            WHEN d.complaint_status = 'Pending' AND
i.complaint_status = 'Assigned' AND i.employee_id IS NULL
                                THEN 'Pending'

                            WHEN d.complaint_status= 'Pending' AND
i.complaint_status = 'Assigned' AND
                                 i.employee_id  IS NOT NULL
                                THEN 'Assigned'

                            WHEN d.complaint_status = 'Assigned' AND
i.complaint_status <> 'Resolved'
                                THEN 'Assigned'

                            WHEN d.complaint_status = 'Pending' AND
i.complaint_status <> 'Assigned'
                                THEN 'Pending'
                            WHEN d.complaint_status = 'Resolved'
                                THEN 'Resolved'
                            ELSE
                                i.complaint_status
       END,
       resolved_timestamp= CASE
                            WHEN d.complaint_status = 'Assigned' AND
i.complaint_status = 'Resolved'
                                THEN getdate()
                            ELSE
                                d.resolved_timestamp
          END
   FROM complaint o,
        inserted i,
        deleted d
   WHERE o.complaint_id = i.complaint_id
     AND o.complaint_id = d.complaint_id
```

```
END
GO
CREATE TRIGGER NoUserUpdate
   ON users
   AFTER UPDATE
   AS
   IF UPDATE(user_id)
       BEGIN
           ;THROW 51000, 'You cannot update the primary key user_id', 1;
       END
GO
CREATE TRIGGER NoEmployeeUpdate
   ON employee
   AFTER UPDATE
   AS
   IF UPDATE(employee_id)
       BEGIN
           ;THROW 51000, 'You cannot update the primary key employeemployee_id', 1;
       END
GO
CREATE TRIGGER NoOrderUpdate
   ON orders
   AFTER UPDATE
   AS
   IF UPDATE(order_id)
       BEGIN
           ;THROW 51000, 'You cannot update the primary key orderID', 1;
       END
CREATE TRIGGER NoComplaintIDUpdate
    ON complaint
    AFTER UPDATE
    AS
    IF UPDATE(complaint_id)
        BEGIN
            ;THROW 51000, 'You cannot update the primary key complaint
_ID', 1;
        END
```

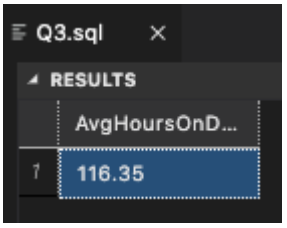
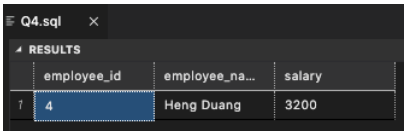
Figure 1.1 SQL DDL commands for table creation. It also includes triggers.

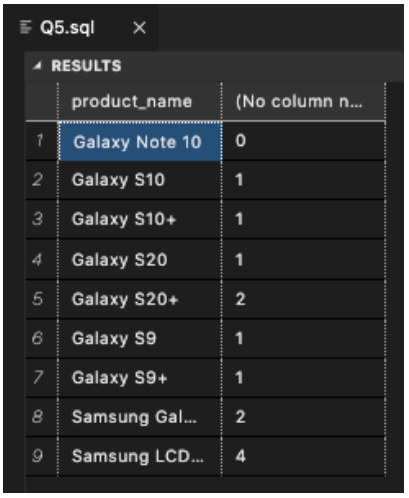
2. SQL Queries with Result for Lab Manual Appendix B Questions

The SQL Queries are pasted in this document at Appendix A for readability. Queries with results screenshot are available after the table.

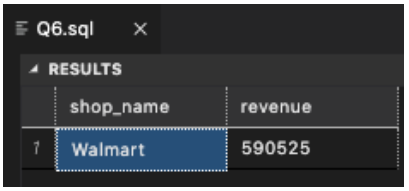
Qn	Query	Results
1	-- Find the average price of "iPhone Xs" on Shiokee from 1 August 2021 to 31 August 2021. SELECT product_name, ROUND(AVG(Cast(actual_price as Float)), 2) AS AvgPrice FROM price_history WHERE product_name = 'iPhone XS' --select "iPhone Xs" AND ((start_date >= '2021.08.01 00:00:00' AND start_date < '2021.09.01 00:00:00') -- select start_date from 1 August 2021 to 31 August 2021 OR (end_date >= '2021.08.01 00:00:00' AND end_date < '2021.09.01 00:00:00')) -- select end_date from 1 August 2021 to 31 August 2021 GROUP BY product_name; -- Additional Group By clause added to print the product name 'iPhone XS'	
2	-- Find products that received at least 100 ratings of "5" in August 2021, -- and order them by their average ratings. Drop table if exists good_products -- Clarification: in our implentation, the overall average ratings for desired products are found	

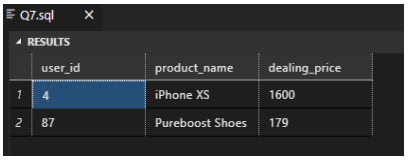
```
-- create temporary table "good_products"
which stores product name with more than 100
ratings of "5"
SELECT product_name
INTO good_products
FROM feedback
WHERE rating = 5
 AND MONTH(feedbackDate) = 8
 AND YEAR(feedbackDate) = 2021
GROUP BY product_name
HAVING COUNT(rating) >= 100;

-- -- printing the average ratings for these
products
SELECT product_name, ROUND(AVG(Cast(rating as
Float)), 2) AS AvgRatings
FROM feedback
WHERE product_name IN (SELECT * FROM
good_products) --from the temporary table
"good_products"
GROUP BY product_name
ORDER BY AvgRatings DESC; -- AvgRatings in
decreasing order


/*
-- If we refer the "average ratings" as the
average ratings received by the desired
products in August 2021
-- We can provide the simplified queries as
follows:

SELECT product_name, ROUND(AVG(Cast(rating as
Float)), 2) AS AvgRatings
FROM feedback
WHERE rating = 5
 AND MONTH(feedbackDate) = 8
 AND YEAR(feedbackDate) = 2021
GROUP BY product_name
HAVING COUNT(rating) >= 100
ORDER BY AvgRatings DESC;
*/
```

3	-- For all products purchased in June 2021 that have been delivered, find the average time from the -- ordering date to the delivery date. -- Clarification: both 'order_placing_timestamp' and 'delivery_date' are timestamps with an accuracy of 1 milisecond -- calculate average time of delivery in hours SELECT ROUND(AVG(CAST(DATEDIFF(second, order_placing_timestamp, delivery_date) AS FLOAT)) / 3600, 2) AS AvgHoursOnDelivery -- calculate average time of delivery in days -- SELECT ROUND(AVG(CAST(DATEDIFF(day, order_placing_timestamp, delivery_date) AS FLOAT)), 2) AS AvgDaysOnDelivery FROM orders, product_on_order WHERE orders.order_id = product_on_order.order_id AND order_placing_timestamp >= '2021-06-01 00:00:00' -- start from 06.01 (06.01 included) AND order_placing_timestamp < '2021-07-01 00:00:00' -- until 07.01 (07.01 not included) AND (product_on_order_status = 'delivered' OR product_on_order_status = 'returned'); -- select all the products that have been delivered or the products that have been delivered to the customer and returned to the store	
4	-- Let us define the “latency” of an employee by the average that he/she takes to process a complaint. -- Find the employee with the smallest latency. -- Subquery: latency_record -- We first aggregate the latency defined in the question -- Then we select the employee(s) by clauses TOP 1 WITH TIES and ORDER BY -- Note: We use WITH TIES to allow multiple	

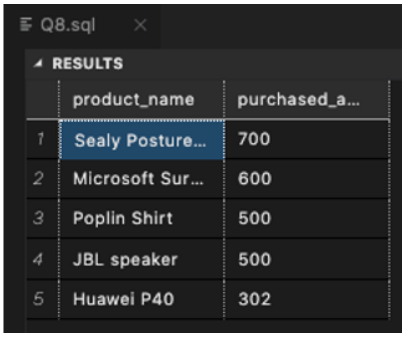
	results WITH latency_record AS(SELECT TOP 1 WITH TIES employee_id, AVG(CAST(DATEDIFF(second, resolved_timestamp, assigned_timestamp) as FLOAT)) AS latency FROM complaint GROUP BY employee_id ORDER BY latency) -- Fetch all information of the employee(s) from table employee SELECT employee_id, employee_name, salary FROM employee WHERE employee_id IN (SELECT employee_id FROM latency_record)	
5	-- Produce a list that contains -- (i) all products made by Samsung, and -- (ii) for each of them, the number of shops on Shiokee that sell the product. -- Note: -- We specially engineered the data: -- No shop sells the Samsung product Galaxy Note 10 even though it appears in the products -- Left join is adopted to produce list contain all products made by Samsung -- We use WHERE clause to filter products made by Samsung -- Then GROUP BY each product name, and COUNT the shops that sell them -- (NULL is atomatically treated as 0) SELECT p.product_name, COUNT(pis.shop_name) -- SUM(CASE WHEN pis.product_name IS NULL THEN 0 ELSE 1 END) AS shop_count FROM product AS p LEFT JOIN product_in_shop AS pis ON p.product_name = pis.product_name WHERE p.maker = 'Samsung' GROUP BY p.product_name /* -- Version that cannot print out products if	 Q5.sql RESULTS product_name | (No column n... 1 Galaxy Note 10 | 0 2 Galaxy S10 | 1 3 Galaxy S10+ | 1 4 Galaxy S20 | 1 5 Galaxy S20+ | 2 6 Galaxy S9 | 1 7 Galaxy S9+ | 1 8 Samsung Gal... | 2 9 Samsung LCD... | 4

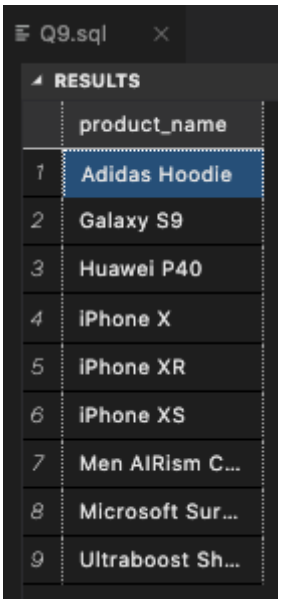
```
no shop sells them
SELECT product_name, COUNT(DISTINCT shop_name)
AS shop_count
FROM product_in_shop
WHERE product_name IN (
   SELECT product_name
   FROM product
   WHERE maker = 'Samsung'
)
GROUP BY product_name
*/

-- Find shops that made the most revenue in
August 2021.

-- We first join order with product on order
to achieve dealing_price and qorder_quantity
-- and use WHERE clause to filter records in
Aug 2021
-- We then GROUP BY each shop, and aggregate
the revenue as: SUM(t2.dealing_price *
t2.order_quantity)
-- Finally, we select the shop(s) made most
revenue by clauses TOP 1 WITH TIES and ORDER
BY
-- Note: We use WITH TIES to allow multiple
results

SELECT TOP 1 WITH TIES t2.shop_name,
SUM(t2.dealing_price * t2.order_quantity) AS
revenue
FROM orders as t1 JOIN product_on_order as t2
ON t1.order_id = t2.order_id
WHERE YEAR(t1.order_placing_timestamp) = 2021
 AND MONTH(t1.order_placing_timestamp) = 8
GROUP BY t2.shop_name
ORDER BY revenue DESC

/*
-- Version with subquery
-- less compact than the single-query version

WITH shop_revenue AS (
 SELECT t2.shop_name, SUM(t2.dealing_price *
```

	t2.order_quantity) AS revenue FROM orders as t1 JOIN product_on_order as t2 ON t1.order_id = t2.order_id WHERE YEAR(t1.order_placing_timestamp) = 2021 AND MONTH(t1.order_placing_timestamp) = 8 GROUP BY t2.shop_name) SELECT shop_name, revenue FROM shop_revenue WHERE revenue = (SELECT MAX(revenue) FROM shop_revenue); */	
6	-- Find shops that made the most revenue in August 2021. -- We first join order with product on order to achieve dealing_price and qorder_quantity -- and use WHERE clause to filter records in Aug 2021 -- We then GROUP BY each shop, and aggregate the revenue as: SUM(t2.dealing_price * t2.order_quantity) -- Finally, we select the shop(s) made most revenue by clauses TOP 1 WITH TIES and ORDER BY -- Note: We use WITH TIES to allow multiple results SELECT TOP 1 WITH TIES t2.shop_name, SUM(t2.dealing_price * t2.order_quantity) AS revenue FROM orders as t1 JOIN product_on_order as t2 ON t1.order_id = t2.order_id WHERE YEAR(t1.order_placing_timestamp) = 2021 AND MONTH(t1.order_placing_timestamp) = 8 GROUP BY t2.shop_name ORDER BY revenue DESC /* -- Version with subquery -- less compact than the single-query version	

	`` WITH shop_revenue AS (SELECT t2.shop_name, SUM(t2.dealing_price * t2.order_quantity) AS revenue FROM orders as t1 JOIN product_on_order as t2 ON t1.order_id = t2.order_id WHERE YEAR(t1.order_placing_timestamp) = 2021 AND MONTH(t1.order_placing_timestamp) = 8 GROUP BY t2.shop_name) SELECT shop_name, revenue FROM shop_revenue WHERE revenue = (SELECT MAX(revenue) FROM shop_revenue); */ ```	
7	``` -- For users that made the most amount of complaints, find the most expensive products he/she has -- ever purchased. -- Clarification: all the steps are deemed necessary for the desired outcome -- Counts the total number of complaints each user has made --Assumption: the tie condition may happens.(examples: 1.two users make the same amount of complaints --and this number is the largest. 2.Two products have the same price and they are all the most expensive products of a specific user). WITH A1 AS(-- Select the users that have made the most complaints --We GROUP BY user id, and aggregate the no. of complaints as: COUNT(user_id) --(Use "TOP 1 WITH TIES" to deal with the tie condition in choosing users that made the most amount of complaints) ```	 *User 4 and User 87 made the same amount of complaints, so there are two person made the most amounts of complaints. We query the corresponding most expensive products for each of them.

```
   SELECT TOP 1 WITH TIES user_id,
COUNT(user_id) as noOfComplaints
   FROM complaint
   GROUP BY user_id
   ORDER BY noOfComplaints DESC
   ),

   -- Split to multiple subqueries for
efficiency (too many joins at one query lose
efficiency)

   -- Select the users in A1 that has made the
most complaints
   -- and their orderID through joining A1
with orders according to same user_id
   A2 AS(
        SELECT t1.user_id, t2.order_id
        FROM A1 as t1 JOIN orders as t2
        ON t1.user_id = t2.user_id
   ),

   -- Find all products that these users in A2
has ever purchased
   -- and dealing price through joining A2
with product_on_order according to same
order_id
   A3 AS (
        SELECT t1.user_id, t2.order_id,
t2.product_name, t2.dealing_price
        FROM A2 as t1 JOIN product_on_order as
t2
        ON t1.order_id = t2.order_id
   ),

   -- Find the most expensive product that
each user in A3 has purchased
   -- this step is necessary for finding the
most expensive products in the last part
   A4 AS(
        SELECT user_id, MAX(dealing_price) as
maxProductPrice
        FROM A3
        GROUP BY user_id
   )
```

	-- Result: Get the most expensive products' name through joining A4 and A3 by matching user_id and the Product price. -- show the user_id, corresponding most expensive product name and prices SELECT t1.user_id, t2.product_name, t2.dealing_price FROM A4 as t1 JOIN A3 as t2 ON t1.user_id = t2.user_id AND t1.maxProductPrice = t2.dealing_price;	
8	-- Find products that have never been purchased by some users, but are the top 5 most purchased -- products by other users in August 2021. --We first join product_on_order and orders by matching the same order id to get the timestamp of each product on order. --Then we filter records in August 2021. --We find the products that never been purchased in Aug 2021 by aggregating on product_name --and select the the product that has less number of corresponding user_ids than the total number of users. --Finally, we use "TOP 5 WITH TIES" to choose the top 5 most purchased products by other users in August 2021. --(Use "TOP 5 WITH TIES" to deal with the tie conditions in choosing the top 5 most purchased product) SELECT TOP 5 WITH TIES t1.product_name, SUM(t1.order_quantity) AS purchased_amount FROM product_on_order as t1 JOIN orders as t2 ON t1.order_id = t2.order_id WHERE YEAR(t2.order_placing_timestamp) = 2021 AND MONTH(t2.order_placing_timestamp) = 8 GROUP BY t1.product_name HAVING count(distinct t2.user_id) < (SELECT count(distinct user_id) FROM users)	

	`` ORDER BY purchased_amount DESC; ```	
9	``` -- Find products that are increasingly being purchased over at least 3 months. -- Create monthly count for total purchased quantity of each product. -- First join the product_on_order and orders to get corresponding product name and timestamp. -- Then aggregating on the product name, month and year to count each total purchased quantity . WITH Monthly_count AS(SELECT t1.product_name, YEAR(t2.order_placing_timestamp) AS Purchased_year, MONTH(t2.order_placing_timestamp) AS Purchased_Month, SUM(t1.order_quantity) AS Purchased_amount FROM product_on_order as t1 JOIN orders as t2 ON t1.order_id = t2.order_id GROUP BY t1.product_name, YEAR(t2.order_placing_timestamp), MONTH(t2.order_placing_timestamp)) --- we use a three copies of Monthly count tables to filter the product names --- that are increasing purchased over at least three monts by WHERE clause --- which mathching the product names,year,names and make sure the purchase amount is keep increasing. SELECT t1.product_name FROM Monthly_count AS t1, Monthly_count AS t2, Monthly_count AS t3 WHERE t1.product_name = t2.product_name AND t2.product_name = t3.product_name AND (-- Deal with consecutive months (t1.Purchased_year = t2.Purchased_year AND t2.Purchased_year = t3.Purchased_year AND ```	

```
t1.Purchased_Month = t2.Purchased_Month + 1
AND t2.Purchased_Month = t3.Purchased_Month +
1)
     OR -- Cross year consecutive month
senario 1: previous year month 11, month 12,
next year month 1
     (t1.Purchased_year = t2.Purchased_year +
1 AND t2.Purchased_year = t3.Purchased_year
AND t1.Purchased_Month = 1 AND
t2.Purchased_Month = 12 AND t3.Purchased_Month
= 11)
     OR -- Cross year consecutive month
senario 2: previous year month 12, next year
month 1, month 2
     (t1.Purchased_year = t2.Purchased_year
AND t2.Purchased_year = t3.Purchased_year + 1
AND t1.Purchased_Month = 2 AND
t2.Purchased_Month = 1  AND t3.Purchased_Month
= 12)
  )
  AND t1.Purchased_amount >
t2.Purchased_amount
  AND t2.Purchased_amount >
t3.Purchased_amount
GROUP BY t1.product_name         --- We use
the group by clause to show the distinct
product name that satisfies the conditions


--- it's more interpretable and less
complicated when joing three copies in one
query.
```

Q1.sql

Lab5 > Queries > Q1.sql

```sql
-- Find the average price of "iPhone Xs" on Shiokee from 1 August 2021 to 31 August 2021.

SELECT product_name, ROUND(AVG(Cast(actual_price as Float)), 2) AS AvgPrice
FROM price_history
WHERE product_name = 'iPhone XS' --select "iPhone Xs"
  AND ((start_date >= '2021.08.01 00:00:00' AND start_date < '2021.09.01 00:00:00') -- select start_date from 1 August 2021 to 31 August 2021
    OR (end_date >= '2021.08.01 00:00:00' AND end_date < '2021.09.01 00:00:00')) -- select end_date from 1 August 2021 to 31 August 2021
GROUP BY product_name; -- Additional Group By clause added to print the product name 'iPhone XS'

```

Q1.sql

RESULTS

	product_name	AvgPrice
1	iPhone XS	910.25

Figure 2.1 SQL Query and Results for Question 1

Q2.sql

Lab5 > Queries > Q2.sql

```sql
-- Find products that received at least 100 ratings of "5" in August 2021,
-- and order them by their average ratings.

Drop table if exists good_products

-- Clarification: in our implentation, the overall average ratings for desired products are found

-- create temporary table which stores product name with more than 100 ratings of "5"
SELECT product_name
INTO good_products
FROM feedback
WHERE rating = 5
  AND MONTH(feedbackDate) = 8
  AND YEAR(feedbackDate) = 2021
GROUP BY product_name
HAVING COUNT(rating) >= 100;

-- -- printing the average ratings for these products
SELECT product_name, ROUND(AVG(Cast(rating as Float)), 2) AS AvgRatings
FROM feedback
WHERE product_name IN (SELECT * FROM good_products) --from the temporary table "good_products"
GROUP BY product_name
ORDER BY AvgRatings DESC; -- AvgRatings in decreasing order


/*
```

Q2.sql

RESULTS

	product_name	AvgRatings
1	Galaxy S9	4.96
2	iPhone X	4.93
3	iPhone XS	4.91

Figure 2.2.1 SQL Query and Results for Question 2

Figure 2.2.2 Alternate Query for Question 2

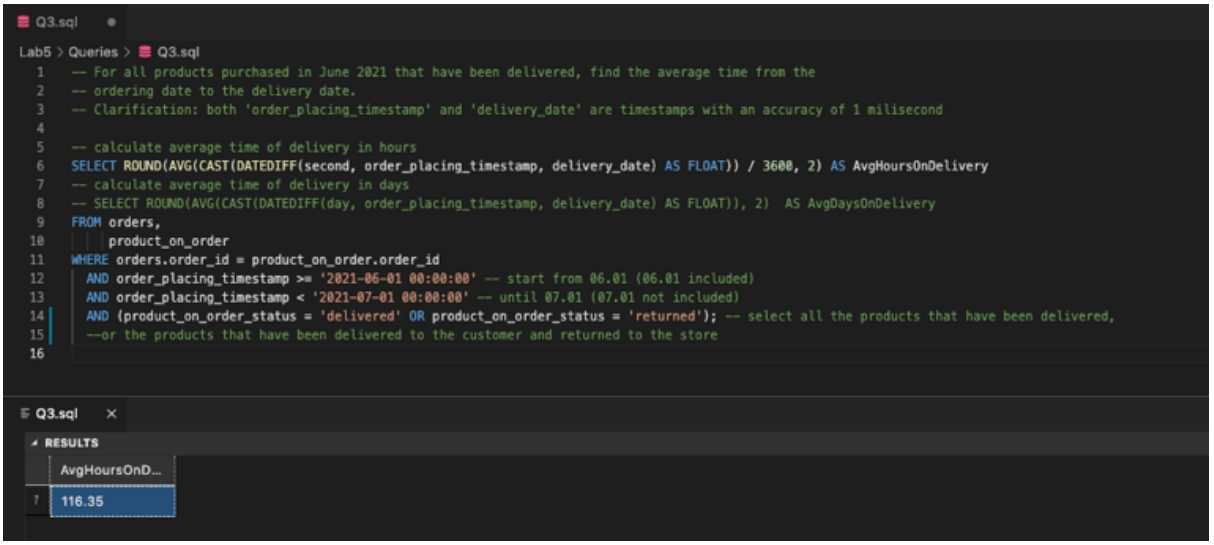


Figure 2.3 SQL Query and Results for Question 3

Q4.sql

Lab5 > Queries > Q4.sql

```sql
-- Let us define the "latency" of an employee by the average that he/she takes to process a complaint.
-- Find the employee with the smallest latency.

-- Subquery: latency_record
-- We first aggregate the latency defined in the question
-- Then we select the employee(s) by clauses TOP 1 WITH TIES and ORDER BY
-- Note: We use WITH TIES to allow multiple results
WITH latency_record AS(
    SELECT TOP 1 WITH TIES employee_id, AVG(CAST(DATEDIFF(second, resolved_timestamp, assigned_timestamp) as FLOAT)) AS latency
    FROM complaint
    GROUP BY employee_id
    ORDER BY latency
)

-- Fetch all information of the employee(s) from table employee
SELECT employee_id, employee_name, salary
FROM employee
WHERE employee_id IN (SELECT employee_id FROM latency_record)
```

Q4.sql

RESULTS

	employee_id	employee_na...	salary
1	4	Heng Duang	3200

Figure 2.4 SQL Query and Results for Question 4

Q5.sql

Lab5 > Queries > Q5.sql

```sql
-- Produce a list that contains
-- (i) all products made by Samsung, and
-- (ii) for each of them, the number of shops on Shiokee that sell the product.

-- Note:
-- We specially engineered the data:
-- No shop sells the Samsung product Galaxy Note 10 even though it appears in the products

-- Left join is adopted to produce list contain all products made by Samsung
-- We use WHERE clause to filter products made by Samsung
-- Then GROUP BY each product name, and COUNT the shops that sell them
-- (NULL is atomatically treated as 0)

SELECT p.product_name, COUNT(pis.shop_name)
    -- SUM(CASE WHEN pis.product_name IS NULL THEN 0 ELSE 1 END) AS shop_count
FROM product AS p
    LEFT JOIN product_in_shop AS pis
    ON p.product_name = pis.product_name
WHERE p.maker = 'Samsung'
GROUP BY p.product_name

/*
```

Q5.sql

RESULTS

	product_name	(No column n...
1	Galaxy Note 10	0
2	Galaxy S10	1
3	Galaxy S10+	1
4	Galaxy S20	1
5	Galaxy S20+	2
6	Galaxy S9	1
7	Galaxy S9+	1
8	Samsung Gal...	2
9	Samsung LCD...	4

Figure 2.5.1 SQL Query and Results for Question 5

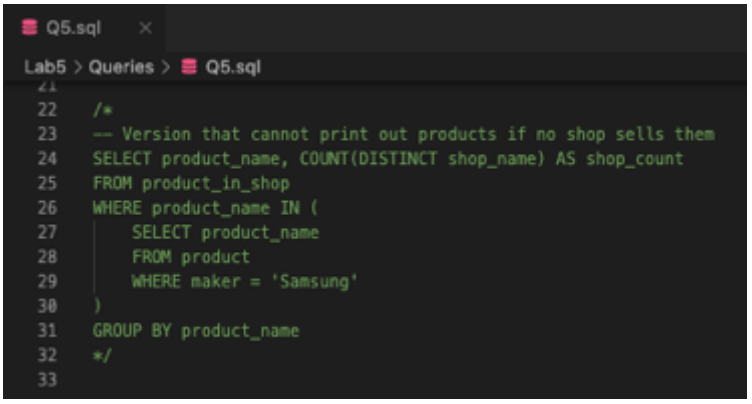


Figure 2.5.2 Alternate Query for Question 5

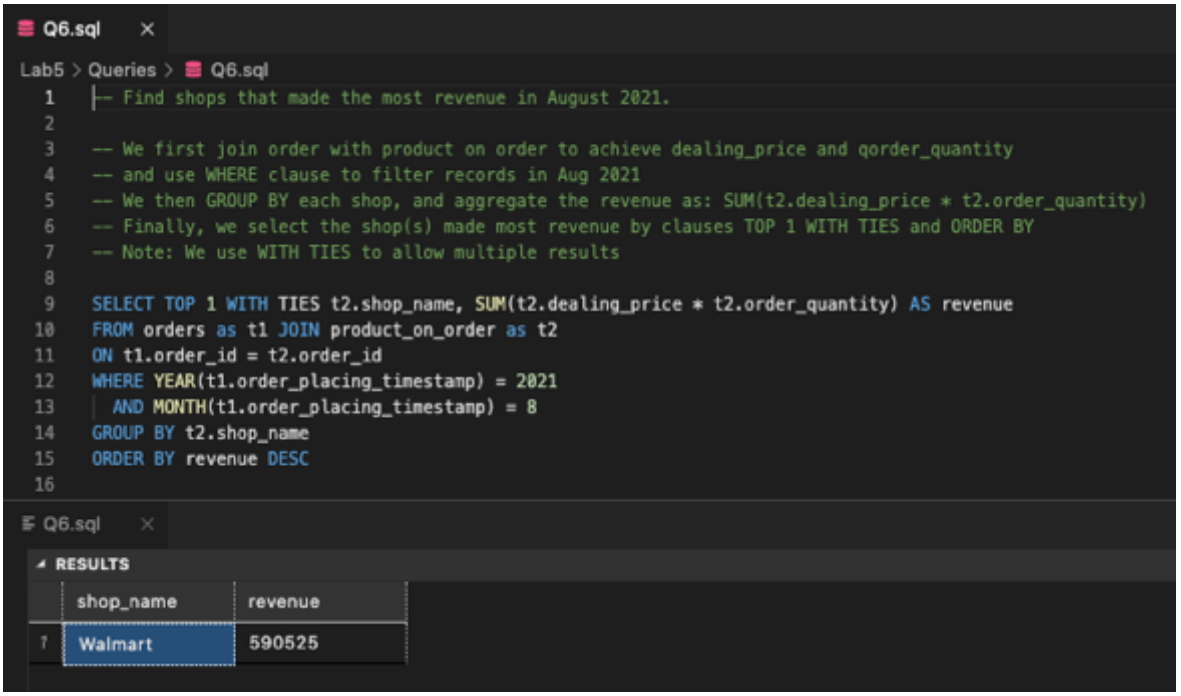


Figure 2.6.1 SQL Query and Results for Question 6

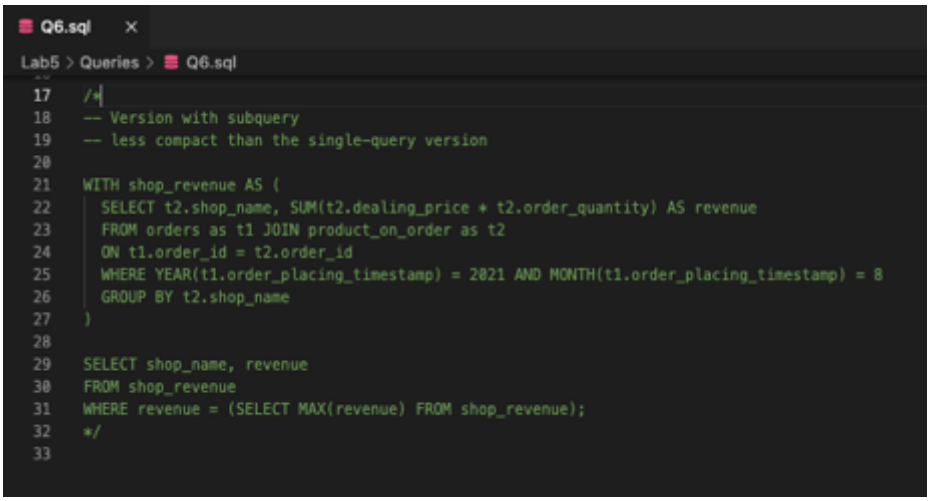
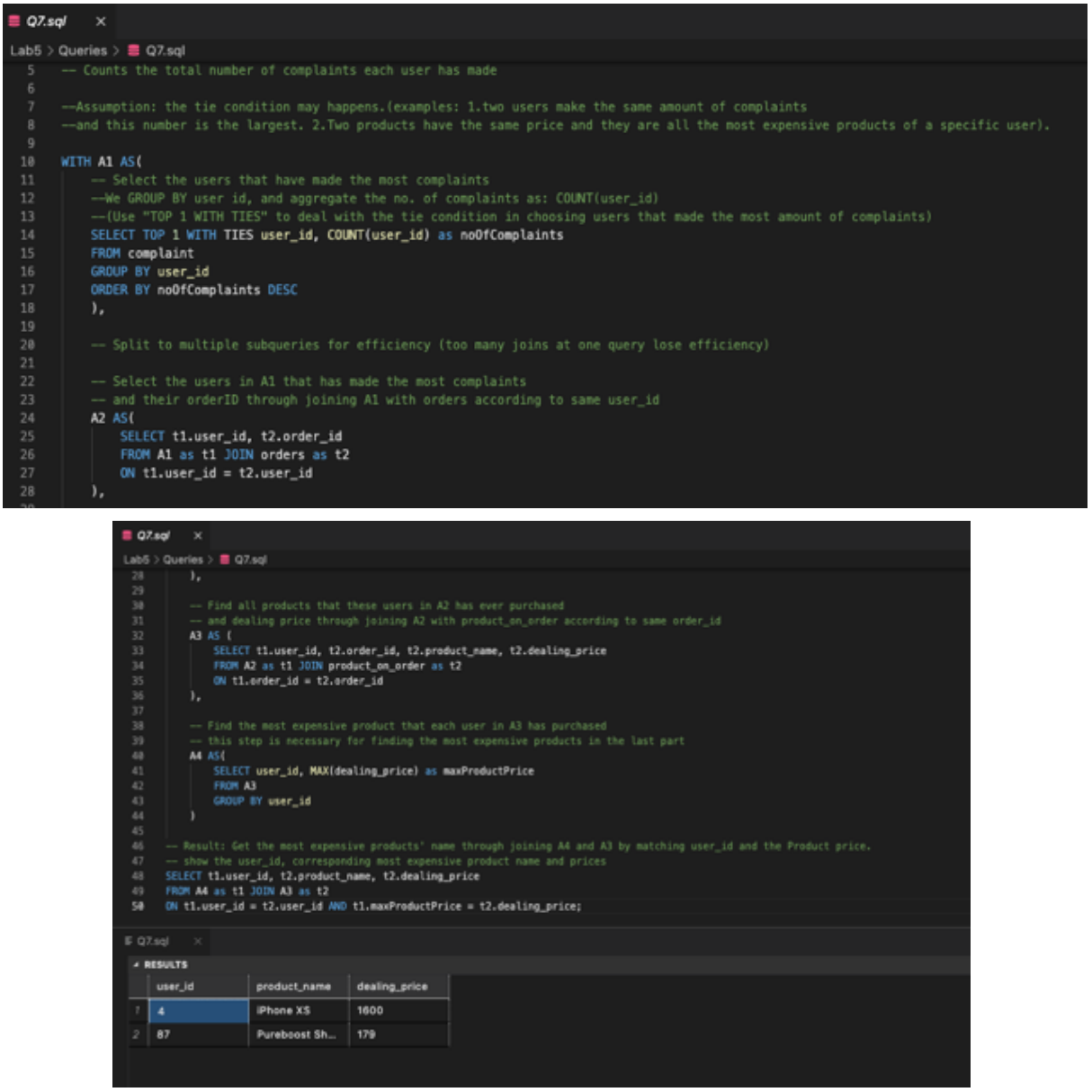


Figure 2.6.2 Alternate Version Query for Question 6



```
Q7.sql
Lab5 > Queries > Q7.sql
-- Counts the total number of complaints each user has made

--Assumption: the tie condition may happens.(examples: 1.two users make the same amount of complaints
--and this number is the largest. 2.Two products have the same price and they are all the most expensive products of a specific user).

WITH A1 AS(
    -- Select the users that have made the most complaints
    --We GROUP BY user id, and aggregate the no. of complaints as: COUNT(user_id)
    --(Use "TOP 1 WITH TIES" to deal with the tie condition in choosing users that made the most amount of complaints)
    SELECT TOP 1 WITH TIES user_id, COUNT(user_id) as noOfComplaints
    FROM complaint
    GROUP BY user_id
    ORDER BY noOfComplaints DESC
    ),

    -- Split to multiple subqueries for efficiency (too many joins at one query lose efficiency)

    -- Select the users in A1 that has made the most complaints
    -- and their orderID through joining A1 with orders according to same user_id
    A2 AS(
        SELECT t1.user_id, t2.order_id
        FROM A1 as t1 JOIN orders as t2
        ON t1.user_id = t2.user_id
    ),
```



```
Q7.sql
Lab5 > Queries > Q7.sql
    ),

    -- Find all products that these users in A2 has ever purchased
    -- and dealing price through joining A2 with product_on_order according to same order_id
    A3 AS (
        SELECT t1.user_id, t2.order_id, t2.product_name, t2.dealing_price
        FROM A2 as t1 JOIN product_on_order as t2
        ON t1.order_id = t2.order_id
    ),

    -- Find the most expensive product that each user in A3 has purchased
    -- this step is necessary for finding the most expensive products in the last part
    A4 AS(
        SELECT user_id, MAX(dealing_price) as maxProductPrice
        FROM A3
        GROUP BY user_id
    )

-- Result: Get the most expensive products' name through joining A4 and A3 by matching user_id and the Product price.
-- show the user_id, corresponding most expensive product name and prices
SELECT t1.user_id, t2.product_name, t2.dealing_price
FROM A4 as t1 JOIN A3 as t2
ON t1.user_id = t2.user_id AND t1.maxProductPrice = t2.dealing_price;
```

Q7.sql

RESULTS

	user_id	product_name	dealing_price
1	4	iPhone XS	1600
2	87	Pureboost Sh...	179

Figure 2.7.1 & Figure 2.7.2 SQL Queries and Results for Question 7

Q8.sql

Lab5 > Queries > Q8.sql

```sql
-- Find products that have never been purchased by some users, but are the top 5 most purchased
-- products by other users in August 2021.


--We first join product_on_order and orders by matching the same order id to get the timestamp of each product on order.
--Then we filter records in August 2021.
--We find the products that never been purchased in Aug 2021 by aggregating on product_name
--and select the the product that has less number of corresponding user_ids than the total number of users.
--Finally, we use "TOP 5 WITH TIES" to choose the top 5 most purchased products by other users in August 2021.
--(Use "TOP 5 WITH TIES" to deal with the tie conditions in choosing the top 5 most purchased product)

SELECT TOP 5 WITH TIES t1.product_name, SUM(t1.order_quantity) AS purchased_amount
FROM product_on_order as t1
JOIN orders as t2 ON t1.order_id = t2.order_id
WHERE YEAR(t2.order_placing_timestamp) = 2021
AND MONTH(t2.order_placing_timestamp) = 8
GROUP BY t1.product_name
HAVING count(distinct t2.user_id) < (SELECT count(distinct user_id) FROM users)
ORDER BY purchased_amount DESC;
```

Q8.sql

RESULTS

	product_name	purchased_a...
1	Sealy Posture...	700
2	Microsoft Sur...	600
3	Poplin Shirt	500
4	JBL speaker	500
5	Huawei P40	302

Figure 2.8 SQL Queries and Results for Question 8

Q9.sql

Lab5 > Queries > Q9.sql

```sql
-- Find products that are increasingly being purchased over at least 3 months.

-- Create monthly count for total purchased quantity of each product.
-- First join the product_on_order and orders to get corresponding product name and timestamp.
-- Then aggregating on the product name, month and year to count each total purchased quantity .
WITH Monthly_count AS(
SELECT t1.product_name,
    YEAR(t2.order_placing_timestamp) AS Purchased_year,
    MONTH(t2.order_placing_timestamp) AS Purchased_Month,
    SUM(t1.order_quantity) AS Purchased_amount
FROM product_on_order as t1 JOIN orders as t2 ON t1.order_id = t2.order_id
GROUP BY t1.product_name,
    YEAR(t2.order_placing_timestamp),
    MONTH(t2.order_placing_timestamp)
)
```

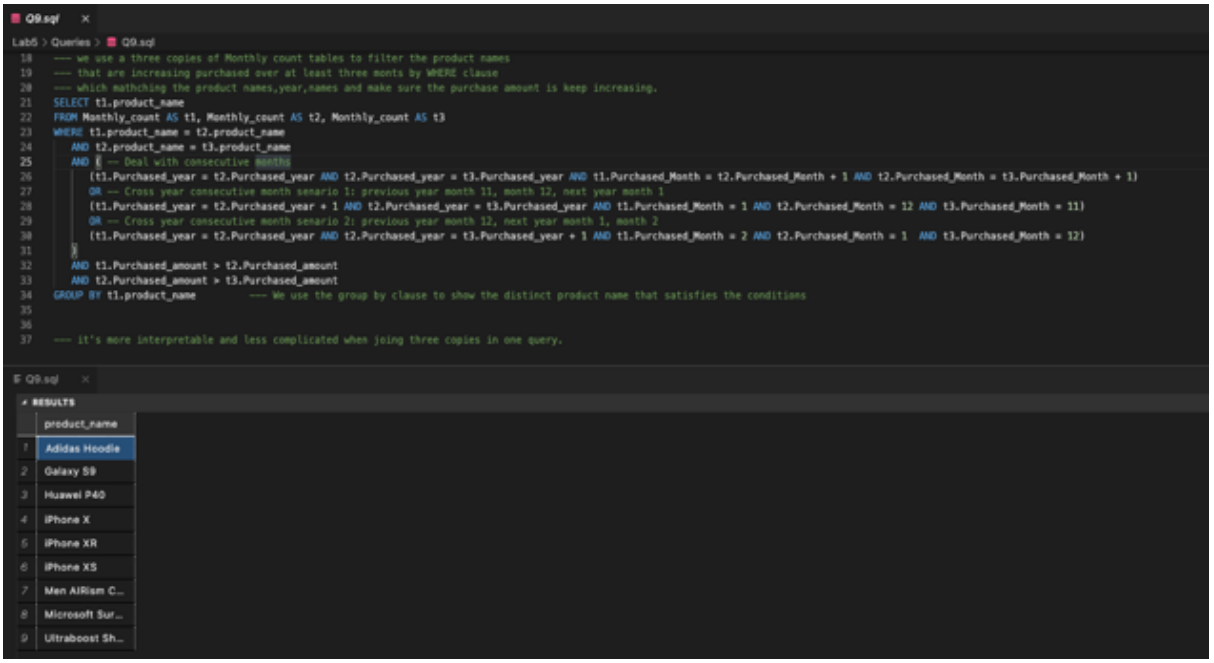


Figure 2.9.1 & Figure 2.9.2 SQL Queries and Results for Question 9

3. SQL Queries for Additional Queries

4. Printout of All Table Records

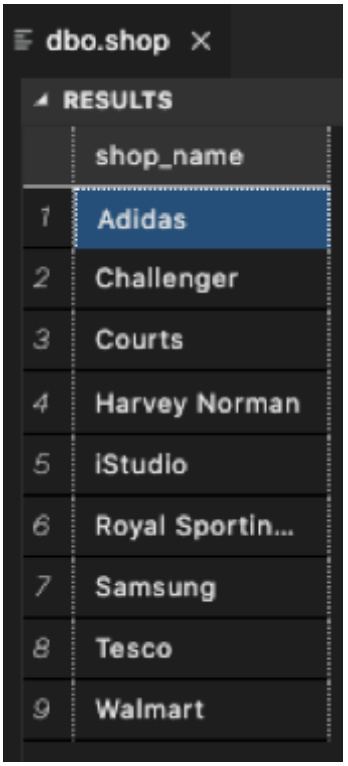
For the full table records, please open the attached .csv files as some table records are long (about 800 records).

dbo.orders

RESULTS

	order_id	total_shipping...	shipping_addr	order_placing...	user_id
1	1	50	4338 Rutledg...	2021-01-02 0...	1
2	2	30	2147 Novick T...	2021-01-04 0...	2
3	3	0	36133 Namek...	2021-01-04 0...	3
4	4	0	40591 Stephe...	2021-01-05 0...	4
5	5	0	41 Fallview Dr...	2021-01-05 0...	5
6	6	0	4 Corscot Lane	2021-01-08 0...	6
7	7	0	06 Commerci...	2021-01-10 0...	7
8	8	0	1789 Morning...	2021-01-12 0...	8
9	9	0	3444 Northla...	2021-01-16 0...	9
10	10	0	68280 Granb...	2021-01-18 0...	10
11	11	0	7 Karstens Pl...	2021-01-19 0...	11
12	12	0	464 Carioca T...	2021-01-19 0...	12
13	13	0	884 Vernon C...	2021-01-20 0...	13
14	14	0	0113 Memoria...	2021-01-22 0...	14
15	15	0	9 Randy Center	2021-01-24 0...	15
16	16	0	21 Ryan Place	2021-01-25 0...	16
17	17	0	77809 Bartill...	2021-01-26 0...	17
18	18	0	22 Callangt P...	2021-01-28 0...	18
19	19	0	1236 Hanover...	2021-01-28 0...	19
20	20	0	8 Dorton Ave...	2021-01-28 0...	20
21	21	0	9811 Doe Cro...	2021-02-01 0...	21
22	22	0	4922 Waubes...	2021-02-02 0...	22
23	23	60	5458 Annama...	2021-02-02 0...	23
24	24	45	1142 Pleasure...	2021-02-03 0...	24
25	25	0	552 Menomo...	2021-02-04 0...	25
26	26	0	24 Grayhawk ...	2021-02-05 0...	26
27	27	0	27 Hallows All...	2021-02-06 0...	27

Figure 4.1 Orders Table Record

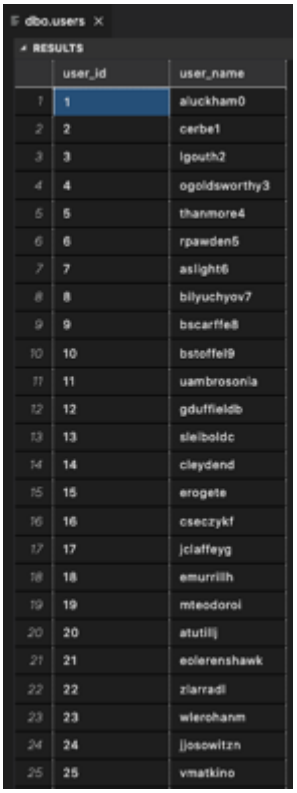


dbo.shop

RESULTS

	shop_name
1	Adidas
2	Challenger
3	Courts
4	Harvey Norman
5	iStudio
6	Royal Sportin...
7	Samsung
8	Tesco
9	Walmart

Figure 4.2 Shop Table Record



dbo.users

RESULTS

	user_id	user_name
1	1	aluckham0
2	2	cerbe1
3	3	lgouth2
4	4	ogoldsworthy3
5	5	thanmore4
6	6	rpawden5
7	7	aslight6
8	8	bilyuchyov7
9	9	bscarffe8
10	10	bstoffel9
11	11	uambrosonia
12	12	gduffieldb
13	13	sleiboldc
14	14	cleydend
15	15	erogete
16	16	cseczykf
17	17	jclaffeyg
18	18	emurrillh
19	19	mteodoroi
20	20	atutilij
21	21	eolerenshawk
22	22	zlarradl
23	23	wlerohanm
24	24	jjosowitzn
25	25	vmatkino

Figure 4.3 Users Table Record

dbo.employee

RESULTS

	employee_id	employee_na...	salary
1	1	Josephine Tan	2000
2	2	Jun Xiong	2600
3	3	Wan Qian	3500
4	4	Heng Duang	3200

Figure 4.4 Employee Table Record

dbo.complaint

RESULTS

	complaint_id	complain_des...	file_timestamp	resolved_time...	assigned_tim...	complaint_sta...	user_id	employee_id
1	1	Horrible Servi...	2021-08-03 0...	2021-09-03 2...	2021-09-03 2...	Resolved	4	1
2	2	Horrible pack...	2021-08-01 0...	2021-09-01 0...	2021-09-01 0...	Resolved	13	1
3	3	Terrible replyi...	2021-08-01 0...	2021-09-01 0...	2021-09-01 0...	Resolved	27	2
4	4	Rude	2021-08-01 0...	2021-09-20 0...	2021-09-20 0...	Resolved	87	2
5	5	Broken Product	2021-08-01 0...	2021-09-01 0...	2021-09-01 0...	Resolved	69	3
6	6	Product looks...	2021-08-01 0...	2021-09-01 0...	2021-09-01 0...	Resolved	73	3
7	7	Wrong colour	2021-08-01 0...	2021-08-20 0...	2021-08-20 0...	Resolved	4	4
8	8	Delivery was ...	2021-08-01 0...	2021-09-01 0...	2021-09-01 0...	Resolved	20	4
9	9	Horrible Servi...	2021-08-01 0...	2021-09-01 0...	2021-09-01 0...	Resolved	19	1
10	10	D:<	2021-08-01 0...	2021-08-20 0...	2021-08-20 0...	Resolved	87	2

Figure 4.5 Complaint Table Record

dbo.complaint_on_shop

RESULTS

	complaint_id	shop_name
1	1	Tesco
2	2	Walmart
3	3	Challenger
4	4	iStudio
5	8	Samsung
6	9	Samsung
7	10	Adidas

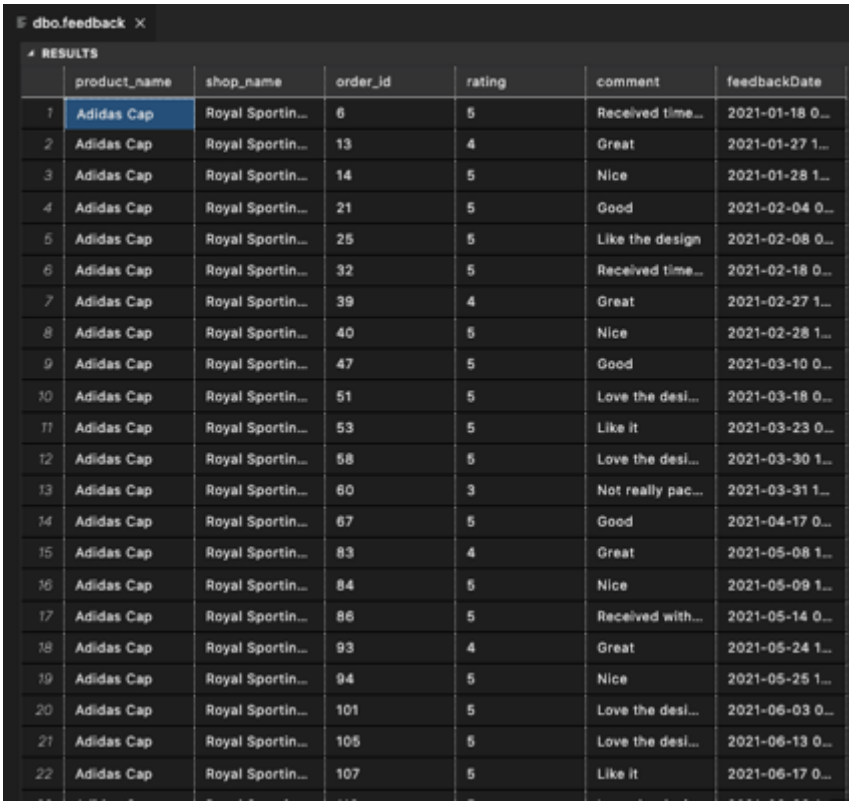
Figure 4.6 Complaint_on_shop Table Record

dbo.complaint_on_product

RESULTS

	complaint_id	product_name	shop_name	order_id
1	5	Ultraboost Sh...	Adidas	10
2	6	iPhone X	iStudio	5
3	7	iPhone XS	iStudio	124

Figure 4.7 Complaint_on_product Table Record



dbo.feedback

RESULTS

	product_name	shop_name	order_id	rating	comment	feedbackDate
1	Adidas Cap	Royal Sportin...	6	5	Received time...	2021-01-18 0...
2	Adidas Cap	Royal Sportin...	13	4	Great	2021-01-27 1...
3	Adidas Cap	Royal Sportin...	14	5	Nice	2021-01-28 1...
4	Adidas Cap	Royal Sportin...	21	5	Good	2021-02-04 0...
5	Adidas Cap	Royal Sportin...	25	5	Like the design	2021-02-08 0...
6	Adidas Cap	Royal Sportin...	32	5	Received time...	2021-02-18 0...
7	Adidas Cap	Royal Sportin...	39	4	Great	2021-02-27 1...
8	Adidas Cap	Royal Sportin...	40	5	Nice	2021-02-28 1...
9	Adidas Cap	Royal Sportin...	47	5	Good	2021-03-10 0...
10	Adidas Cap	Royal Sportin...	51	5	Love the desi...	2021-03-18 0...
11	Adidas Cap	Royal Sportin...	53	5	Like it	2021-03-23 0...
12	Adidas Cap	Royal Sportin...	58	5	Love the desi...	2021-03-30 1...
13	Adidas Cap	Royal Sportin...	60	3	Not really pac...	2021-03-31 1...
14	Adidas Cap	Royal Sportin...	67	5	Good	2021-04-17 0...
15	Adidas Cap	Royal Sportin...	83	4	Great	2021-05-08 1...
16	Adidas Cap	Royal Sportin...	84	5	Nice	2021-05-09 1...
17	Adidas Cap	Royal Sportin...	86	5	Received with...	2021-05-14 0...
18	Adidas Cap	Royal Sportin...	93	4	Great	2021-05-24 1...
19	Adidas Cap	Royal Sportin...	94	5	Nice	2021-05-25 1...
20	Adidas Cap	Royal Sportin...	101	5	Love the desi...	2021-06-03 0...
21	Adidas Cap	Royal Sportin...	105	5	Love the desi...	2021-06-13 0...
22	Adidas Cap	Royal Sportin...	107	5	Like it	2021-06-17 0...

Figure 4.8 Feedback Table Record

dbo.product

RESULTS

	product_name	category	maker
1	Adidas Cap	Clothing	Adidas
2	Adidas Hoodie	Clothing	Adidas
3	Apple - North...	Grocery	NULL
4	Beans - Yellow	Grocery	NULL
5	Dell XPS Note...	Electronic Ga...	Dell
6	Doilies - 8, Pa...	Grocery	NULL
7	Galaxy Note 10	Electronic Ga...	Samsung
8	Galaxy S10	Electronic Ga...	Samsung
9	Galaxy S10+	Electronic Ga...	Samsung
10	Galaxy S20	Electronic Ga...	Samsung
11	Galaxy S20+	Electronic Ga...	Samsung
12	Galaxy S9	Electronic Ga...	Samsung
13	Galaxy S9+	Electronic Ga...	Samsung
14	Glass Cup	Tableware	IKEA
15	Huawei P40	Electronics	Huawei
16	iphon	Clothing	G2000
17	iPhone X	Electronic Ga...	Apple
18	iPhone XR	Electronic Ga...	Apple
19	iPhone XS	Electronic Ga...	Apple
20	JBL speaker	Electronics	JBL

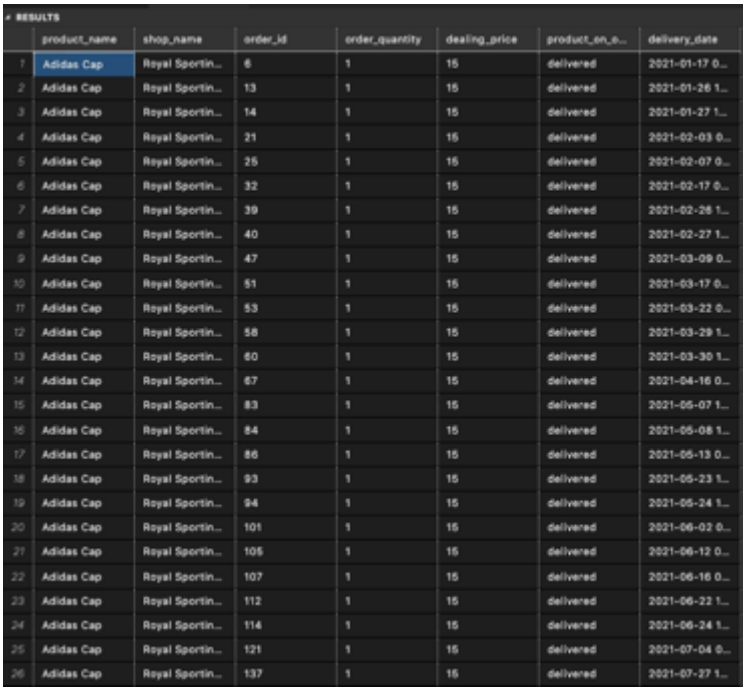
Figure 4.9 Product Table Record

dbo.product_in_shop

RESULTS

	product_name	shop_name	quantity	price_in_shop
1	Adidas Cap	Adidas	10000	20
2	Adidas Cap	Royal Sportin...	5000	20
3	Adidas Hoodie	Adidas	5000	50
4	Adidas Hoodie	Royal Sportin...	5000	50
5	Dell XPS Note...	Challenger	2000	1500
6	Galaxy Note 10	Samsung	2000	1800
7	Galaxy S10	Samsung	2000	1000
8	Galaxy S10+	Samsung	2000	1100
9	Galaxy S20	Samsung	2000	1200
10	Galaxy S20+	Harvey Norman	1000	1500
11	Galaxy S20+	Samsung	2000	1400
12	Galaxy S9	Samsung	2000	900
13	Galaxy S9+	Samsung	2000	950
14	Glass Cup	Tesco	10000	5000
15	Glass Cup	Walmart	10000	5000
16	Huawei P40	Tesco	10000	999
17	Huawei P40	Walmart	10000	1000
18	iPhone X	iStudio	5000	1000
19	iPhone XR	iStudio	5000	1000
20	iPhone XS	iStudio	5000	1200
21	JBL speaker	Tesco	10000	5
22	JBL speaker	Walmart	10000	5

Figure 4.10 Product_in_shop Table Record



	product_name	shop_name	order_id	order_quantity	dealing_price	product_on_e...	delivery_date
1	Adidas Cap	Royal Sportin...	6	1	15	delivered	2021-01-17 0...
2	Adidas Cap	Royal Sportin...	13	1	15	delivered	2021-01-26 1...
3	Adidas Cap	Royal Sportin...	14	1	15	delivered	2021-01-27 1...
4	Adidas Cap	Royal Sportin...	21	1	15	delivered	2021-02-03 0...
5	Adidas Cap	Royal Sportin...	25	1	15	delivered	2021-02-07 0...
6	Adidas Cap	Royal Sportin...	32	1	15	delivered	2021-02-17 0...
7	Adidas Cap	Royal Sportin...	39	1	15	delivered	2021-02-26 1...
8	Adidas Cap	Royal Sportin...	40	1	15	delivered	2021-02-27 1...
9	Adidas Cap	Royal Sportin...	47	1	15	delivered	2021-03-09 0...
10	Adidas Cap	Royal Sportin...	51	1	15	delivered	2021-03-17 0...
11	Adidas Cap	Royal Sportin...	53	1	15	delivered	2021-03-22 0...
12	Adidas Cap	Royal Sportin...	58	1	15	delivered	2021-03-29 1...
13	Adidas Cap	Royal Sportin...	60	1	15	delivered	2021-03-30 1...
14	Adidas Cap	Royal Sportin...	67	1	15	delivered	2021-04-16 0...
15	Adidas Cap	Royal Sportin...	83	1	15	delivered	2021-05-07 1...
16	Adidas Cap	Royal Sportin...	84	1	15	delivered	2021-05-08 1...
17	Adidas Cap	Royal Sportin...	86	1	15	delivered	2021-05-13 0...
18	Adidas Cap	Royal Sportin...	93	1	15	delivered	2021-05-23 1...
19	Adidas Cap	Royal Sportin...	94	1	15	delivered	2021-05-24 1...
20	Adidas Cap	Royal Sportin...	101	1	15	delivered	2021-06-02 0...
21	Adidas Cap	Royal Sportin...	105	1	15	delivered	2021-06-12 0...
22	Adidas Cap	Royal Sportin...	107	1	15	delivered	2021-06-16 0...
23	Adidas Cap	Royal Sportin...	112	1	15	delivered	2021-06-22 1...
24	Adidas Cap	Royal Sportin...	114	1	15	delivered	2021-06-24 1...
25	Adidas Cap	Royal Sportin...	121	1	15	delivered	2021-07-04 0...
26	Adidas Cap	Royal Sportin...	137	1	15	delivered	2021-07-27 1...

Figure 4.11 product_on_order Table Record

dbo.price_history

RESULTS

	product_name	shop_name	start_date	end_date	actual_price
1	iPhone X	Challenger	2021-07-01 0...	2021-08-20 0...	945
2	iPhone X	Challenger	2021-08-22 0...	2021-09-01 0...	999
3	iPhone X	Courts	2021-07-09 0...	2021-07-29 0...	1080
4	iPhone X	Courts	2021-08-09 0...	2021-08-27 0...	988
5	iPhone XS	iStudio	2021-08-28 0...	2021-10-01 0...	911

Figure 4.12 Price_History Table Record

5. Additional Effort

1. For Query 5, we particularly engineered the data that no shop sells the Samsung product “Galaxy Note 10”, to showcase that all products are displayed by the query, and the query is therefore well-designed.
2. We made our best to make the queries more compact (avoid subqueries if possible) and more efficient (join one by one instead of pilling all tables together)

Triggers made to ensure data integrity upon updates.

i, Trigger of delivery status:
The trigger, UpdateDelivery, is to ensure that the delivery status can only move up by a stage at a time in the following logical sequences.

'being processed'->'shipped'->'delivered'->'returned'

If the sequence is not obeyed, the update will be prevented by the trigger.

In addition, it acts as a sanity check for delivery status on delivery date. If delivery_status is updated to 'Delivered', then delivery_date = GETDATE(). In addition, delivery date time will only be updated upon this change. In addition, when performancing status change from ‘delivered’ to ‘returned’, the delivery_date should not be changed. This is to create ease for database users with simple update commands involving the need to set delivery dates.

Furthermore, we have an internal sanity check in create table regarding delivery_date and status.

```
-- sanity check
-- being processed / shipped items should not have a delivery date while the rest should have one
CHECK ((product_on_order_status = 'delivered' AND delivery_date IS NOT NULL)
    OR (product_on_order_status = 'returned' AND delivery_date IS NOT NULL)
    OR (product_on_order_status = 'being processed' AND delivery_date IS NULL)
    OR (product_on_order_status = 'shipped' AND delivery_date IS NULL)),
PRIMARY KEY (product_name, shop_name, order_id),
```

Where being processed / shipped items should not have a delivery date (null value is then placed) while the delivered and returned items should have one

ii. Trigger of complaint status
This trigger, ComplaintStatus, is to ensure that the complaint status can only move up by a stage at a time in the following logical sequences.

‘pending'->’assigned'->’resolved’

If the sequence is not obeyed, the update will be prevented by the trigger.

In addition, it enacts to help to add more to the database by updating the respective date upon status change. For instance, when one the status is changed from “assigned” to “resolved” upon updates, the resolved_timestamp attribute is automatically set to current time. This is to create ease for database users with simple update commands involving the need to set dates.

“GO
UPDATE Complaint
SET complainstatus='addressed'
WHERE complaint_id=11;” would work where keying in of date is not necessary. However, this update will only happen if the complaint is at the state of ‘assigned’.

Iii. Triggers on update IDs

We disable updating of a few attributes, namely user_id, employee_id, order_id and complaint_id. This is because these should be generated automatically upon creation or generation, and logically they should not be changed.

iv. ON DELETE / ON UPDATE CASCADE

Cascading was on foreign key references. It is used to prevent updating or deletion made to only one table, for example in the case when a record is deleted in the users table then all the subsequent records in the child tables are also deleted. It's the same when the record is updated, all the tables associated with it are updated as well.

3. Data Insertion

We typically insert all data based on the queries we need to perform, in order for all queries to execute accordingly and give a reasonable resultant table.

- We insert about 200 records of user information, 9 shops and various kinds of products to our database system.
- For “order”, we insert nearly 300 records to the “order” table, containing many orders placed in June 2021, in order for query 3 to get correct results.
- For “complaint”, we insert some records with
    - different latencies ([resolved_timestamp] – [assigned_timestamp]) of different employees to resolve the filed complaints, in order for query 4 to find the employees with highest work efficiency;
    - different [user_id] in order for query 7 to find users who made the most complaints.
- For “product in shop”, we provide nearly 50 records, including products from “Samsumg” and many overlapping products from different shops in order for query 5 to get correct results.
- For “product on order”, we insert about 800 records, with attributes including
    - [order_id] to connect it with the “order” table;
    - different [product_on_order_status] and [delivery_date] for query 3 to find the “delivered” products and compute the delivery time.
    - different [dealing_price] and [order_quantity] for query 6 to compute the revenue.
- For “complaint on shop” and “complaint on product”, we insert several records to match the complaints filed with its corresponding products and shops.
- For “feedback”, we add nearly 700 records in order for query 2 to find products with more than 100 ratings of score 5, and compute the average rating scores for those products.
- For price_history, we insert some records containing the price change of “iPhone XS” during August 2021 as well as some price records of other kinds of ”product in shop”, in order for query 1 to select “iPhone XS” and compute its average price in Augut.

6. Appendix A (Complete SQL Queries to Answer Lab Manual Questions)

Question 1.

```
-- Find the average price of "iPhone Xs" on Shiokee from 1 August 2021 to 31 August
2021.

SELECT product_name, ROUND(AVG(Cast(actual_price as Float)), 2) AS AvgPrice
FROM price_history
WHERE product_name = 'iPhone XS' --select "iPhone Xs"
 AND ((start_date >= '2021.08.01 00:00:00' AND start_date < '2021.09.01 00:00:00')
-- select start_date from 1 August 2021 to 31 August 2021
   OR (end_date >= '2021.08.01 00:00:00' AND end_date < '2021.09.01 00:00:00')) --
select end_date from 1 August 2021 to 31 August 2021
GROUP BY product_name; -- Additional Group By clause added to print the product
name 'iPhone XS'
```

Question 2.

```
-- Find products that received at least 100 ratings of "5" in August 2021,
-- and order them by their average ratings.

Drop table if exists good_products

-- Clarification: in our implementation, the overall average ratings for desired
products are found

-- create temporary table "good_products" which stores product name with more than
100 ratings of "5"
SELECT product_name
INTO good_products
FROM feedback
WHERE rating = 5
 AND MONTH(feedbackDate) = 8
 AND YEAR(feedbackDate) = 2021
GROUP BY product_name
HAVING COUNT(rating) >= 100;

-- -- printing the average ratings for these products
SELECT product_name, ROUND(AVG(Cast(rating as Float)), 2) AS AvgRatings
FROM feedback
```

```
WHERE product_name IN (SELECT * FROM good_products) --from the temporary table
"good_products"
GROUP BY product_name
ORDER BY AvgRatings DESC; -- AvgRatings in decreasing order


/*
-- If we refer the "average ratings" as the average ratings received by the desired
products in August 2021
-- We can provide the simplified queries as follows:

SELECT product_name, ROUND(AVG(Cast(rating as Float)), 2) AS AvgRatings
FROM feedback
WHERE rating = 5
 AND MONTH(feedbackDate) = 8
 AND YEAR(feedbackDate) = 2021
GROUP BY product_name
HAVING COUNT(rating) >= 100
ORDER BY AvgRatings DESC;
*/
```

Question 3.

```
-- For all products purchased in June 2021 that have been delivered, find the
average time from the
-- ordering date to the delivery date.
-- Clarification: both 'order_placing_timestamp' and 'delivery_date' are timestamps
with an accuracy of 1 milisecond

-- calculate average time of delivery in hours
SELECT ROUND(AVG(CAST(DATEDIFF(second, order_placing_timestamp, delivery_date) AS
FLOAT)) / 3600, 2) AS AvgHoursOnDelivery
-- calculate average time of delivery in days
-- SELECT ROUND(AVG(CAST(DATEDIFF(day, order_placing_timestamp, delivery_date) AS
FLOAT)), 2)  AS AvgDaysOnDelivery
FROM orders,
    product_on_order
WHERE orders.order_id = product_on_order.order_id
 AND order_placing_timestamp >= '2021-06-01 00:00:00' -- start from 06.01 (06.01
included)
 AND order_placing_timestamp < '2021-07-01 00:00:00' -- until 07.01 (07.01 not
included)
```

```
 AND (product_on_order_status = 'delivered' OR product_on_order_status =
'returned'); -- select all the products that have been delivered or the products
that have been delivered to the customer and returned to the store
```

Question 4.

```
-- Let us define the "latency" of an employee by the average that he/she takes to
process a complaint.
-- Find the employee with the smallest latency.

-- Subquery: latency_record
-- We first aggregate the latency defined in the question
-- Then we select the employee(s) by clauses TOP 1 WITH TIES and ORDER BY
-- Note: We use WITH TIES to allow multiple results
WITH latency_record AS(
  SELECT TOP 1 WITH TIES employee_id, AVG(CAST(DATEDIFF(second, resolved_timestamp,
assigned_timestamp) as FLOAT)) AS latency
  FROM complaint
  GROUP BY employee_id
  ORDER BY latency
)

-- Fetch all information of the employee(s) from table employee
SELECT employee_id, employee_name, salary
FROM employee
WHERE employee_id IN (SELECT employee_id FROM latency_record)
```

Question 5.

```
-- Produce a list that contains
-- (i) all products made by Samsung, and
-- (ii) for each of them, the number of shops on Shiokee that sell the product.

-- Note:
-- We specially engineered the data:
-- No shop sells the Samsung product Galaxy Note 10 even though it appears in the
products

-- Left join is adopted to produce list contain all products made by Samsung
-- We use WHERE clause to filter products made by Samsung
-- Then GROUP BY each product name, and COUNT the shops that sell them
-- (NULL is atomatically treated as 0)
SELECT p.product_name, COUNT(pis.shop_name)
  -- SUM(CASE WHEN pis.product_name IS NULL THEN 0 ELSE 1 END) AS shop_count
FROM product AS p
```

```
   LEFT JOIN product_in_shop AS pis
   ON p.product_name = pis.product_name
WHERE p.maker = 'Samsung'
GROUP BY p.product_name

/*
-- Version that cannot print out products if no shop sells them
SELECT product_name, COUNT(DISTINCT shop_name) AS shop_count
FROM product_in_shop
WHERE product_name IN (
   SELECT product_name
   FROM product
   WHERE maker = 'Samsung'
)
GROUP BY product_name
*/
```

Question 6.

```
-- Find shops that made the most revenue in August 2021.

-- We first join order with product on order to achieve dealing_price and
qorder_quantity
-- and use WHERE clause to filter records in Aug 2021
-- We then GROUP BY each shop, and aggregate the revenue as: SUM(t2.dealing_price *
t2.order_quantity)
-- Finally, we select the shop(s) made most revenue by clauses TOP 1 WITH TIES and
ORDER BY
-- Note: We use WITH TIES to allow multiple results

SELECT TOP 1 WITH TIES t2.shop_name, SUM(t2.dealing_price * t2.order_quantity) AS
revenue
FROM orders as t1 JOIN product_on_order as t2
ON t1.order_id = t2.order_id
WHERE YEAR(t1.order_placing_timestamp) = 2021
 AND MONTH(t1.order_placing_timestamp) = 8
GROUP BY t2.shop_name
ORDER BY revenue DESC

/*
-- Version with subquery
-- less compact than the single-query version

WITH shop_revenue AS (
 SELECT t2.shop_name, SUM(t2.dealing_price * t2.order_quantity) AS revenue
```

```
 FROM orders as t1 JOIN product_on_order as t2
 ON t1.order_id = t2.order_id
 WHERE YEAR(t1.order_placing_timestamp) = 2021 AND
MONTH(t1.order_placing_timestamp) = 8
 GROUP BY t2.shop_name
)

SELECT shop_name, revenue
FROM shop_revenue
WHERE revenue = (SELECT MAX(revenue) FROM shop_revenue);
*/
```

Question 7.

```
-- For users that made the most amount of complaints, find the most expensive
products he/she has
-- ever purchased.

-- Clarification: all the steps are deemed necessary for the desired outcome
-- Counts the total number of complaints each user has made

--Assumption: the tie condition may happens.(examples: 1.two users make the same
amount of complaints
--and this number is the largest. 2.Two products have the same price and they are
all the most expensive products of a specific user).

WITH A1 AS(
   -- Select the users that have made the most complaints
   --We GROUP BY user id, and aggregate the no. of complaints as: COUNT(user_id)
   --(Use "TOP 1 WITH TIES" to deal with the tie condition in choosing users that
made the most amount of complaints)
   SELECT TOP 1 WITH TIES user_id, COUNT(user_id) as noOfComplaints
   FROM complaint
   GROUP BY user_id
   ORDER BY noOfComplaints DESC
   ),

   -- Split to multiple subqueries for efficiency (too many joins at one query lose
efficiency)

   -- Select the users in A1 that has made the most complaints
   -- and their orderID through joining A1 with orders according to same user_id
   A2 AS(
       SELECT t1.user_id, t2.order_id
       FROM A1 as t1 JOIN orders as t2
```

```
        ON t1.user_id = t2.user_id
    ),

    -- Find all products that these users in A2 has ever purchased
    -- and dealing price through joining A2 with product_on_order according to same
order_id
    A3 AS (
        SELECT t1.user_id, t2.order_id, t2.product_name, t2.dealing_price
        FROM A2 as t1 JOIN product_on_order as t2
        ON t1.order_id = t2.order_id
    ),

    -- Find the most expensive product that each user in A3 has purchased
    -- this step is necessary for finding the most expensive products in the last
part
    A4 AS(
        SELECT user_id, MAX(dealing_price) as maxProductPrice
        FROM A3
        GROUP BY user_id
    )

-- Result: Get the most expensive products' name through joining A4 and A3 by
matching user_id and the Product price.
-- show the user_id, corresponding most expensive product name and prices
SELECT t1.user_id, t2.product_name, t2.dealing_price
FROM A4 as t1 JOIN A3 as t2
ON t1.user_id = t2.user_id AND t1.maxProductPrice = t2.dealing_price;
```

Question 8.

```
-- Find products that have never been purchased by some users, but are the top 5
most purchased
-- products by other users in August 2021.


--We first join product_on_order and orders by matching the same order id to get
the timestamp of each product on order.
--Then we filter records in August 2021.
--We find the products that never been purchased in Aug 2021 by aggregating on
product_name
--and select the the product that has less number of corresponding user_ids than
the total number of users.
--Finally, we use "TOP 5 WITH TIES" to choose the top 5 most purchased products by
other users in August 2021.
--(Use "TOP 5 WITH TIES" to deal with the tie conditions in choosing the top 5 most
purchased product)
```

```
SELECT TOP 5 WITH TIES t1.product_name, SUM(t1.order_quantity) AS purchased_amount
FROM product_on_order as t1
JOIN orders as t2 ON t1.order_id = t2.order_id
WHERE YEAR(t2.order_placing_timestamp) = 2021
AND MONTH(t2.order_placing_timestamp) = 8
GROUP BY t1.product_name
HAVING count(distinct t2.user_id) < (SELECT count(distinct user_id) FROM users)
ORDER BY purchased_amount DESC;
```

Question 9.

```
-- Find products that are increasingly being purchased over at least 3 months.

-- Create monthly count for total purchased quantity of each product.
-- First join the product_on_order and orders to get corresponding product name and
timestamp.
-- Then aggregating on the product name, month and year to count each total
purchased quantity .
WITH Monthly_count AS(
SELECT t1.product_name,
  YEAR(t2.order_placing_timestamp) AS Purchased_year,
  MONTH(t2.order_placing_timestamp) AS Purchased_Month,
  SUM(t1.order_quantity) AS Purchased_amount
FROM product_on_order as t1 JOIN orders as t2 ON t1.order_id = t2.order_id
GROUP BY t1.product_name,
  YEAR(t2.order_placing_timestamp),
  MONTH(t2.order_placing_timestamp)
)


--- we use a three copies of Monthly count tables to filter the product names
--- that are increasing purchased over at least three monts by WHERE clause
--- which mathching the product names,year,names and make sure the purchase amount
is keep increasing.
SELECT t1.product_name
FROM Monthly_count AS t1, Monthly_count AS t2, Monthly_count AS t3
WHERE t1.product_name = t2.product_name
  AND t2.product_name = t3.product_name
  AND ( -- Deal with consecutive months
     (t1.Purchased_year = t2.Purchased_year AND t2.Purchased_year =
t3.Purchased_year AND t1.Purchased_Month = t2.Purchased_Month + 1 AND
t2.Purchased_Month = t3.Purchased_Month + 1)
     OR -- Cross year consecutive month senario 1: previous year month 11, month
12, next year month 1
```

```
     (t1.Purchased_year = t2.Purchased_year + 1 AND t2.Purchased_year =
t3.Purchased_year AND t1.Purchased_Month = 1 AND t2.Purchased_Month = 12 AND
t3.Purchased_Month = 11)
     OR -- Cross year consecutive month senario 2: previous year month 12, next
year month 1, month 2
     (t1.Purchased_year = t2.Purchased_year AND t2.Purchased_year =
t3.Purchased_year + 1 AND t1.Purchased_Month = 2 AND t2.Purchased_Month = 1  AND
t3.Purchased_Month = 12)
  )
  AND t1.Purchased_amount > t2.Purchased_amount
  AND t2.Purchased_amount > t3.Purchased_amount
GROUP BY t1.product_name          --- We use the group by clause to show the
distinct product name that satisfies the conditions


--- it's more interpretable and less complicated when joing three copies in one
query.
```