SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

CZ2007: Introduction to Databases

Implementation of the Database Schema (Lab 4)
&
Final Demonstration (Lab 5)

SS4 Group 1

NAME	MATRICULATION NUMBER
Benjamin Ho JunHao	U2021521L
Benjamin Ong Chee Meng	U2021759D
Eang Sokunthea	U2022525L
Timothy Larry WIjaya	U2021178E
Saniya Nangia	U2022557J
Min Kabar Kyaw	U2021858K

Table creation 4

Constraint 1 7

Constraint 2 8

Query 1 9
- Explanation 9
- Supporting Tables 9
- Query Result 9

Query 2 10
- Explanation 10
- Supporting Tables 10
- Query Result 10

Query 3 11
- Explanation 11
- Supporting Tables 11
- Query Result 11

Query 4 12
- Explanation 12
- Query Result 12

Query 5 13
- Explanation 13
- Supporting Tables 13
- Query Result 13

Query 6 14
- Explanation 14
- Supporting Tables 14
- Query Result 14

Query 7 15
- Explanation 15
- Supporting Tables 16
- Query Result 16

Query 8 17
- Explanation 17
- Supporting Tables 17
- Query Result 18

Query 9 19
- Explanation 19
- Supporting Tables 21
- Query Result 21

Table creation

<table>
<tr><th>Table</th><th>Creation Query</th></tr>
<tr><td>Employees</td><td><pre>CREATE TABLE Employees(
    EID INT IDENTITY(1,1),
    EName VARCHAR(50) NOT NULL,
    Salary DECIMAL(19,2) CHECK (Salary > =0),
    PRIMARY KEY(EID));</pre></td></tr>
<tr><td>Complaints</td><td><pre>CREATE TABLE Complaints(
   CID INT IDENTITY(1,1),
   UID INT NOT NULL,
   Filed_Date_Time DATETIME NOT NULL,
   Complaint_Description VARCHAR(MAX) NOT NULL,
   Complaint_Status VARCHAR(MAX) NOT NULL DEFAULT
'pending' CHECK(Complaint_Status IN ('pending','being
handled','addressed')),
   EID INT,
   Handled_Date_Time DATETIME,
   PRIMARY KEY(CID),
   UNIQUE(UID, Filed_Date_Time),
   FOREIGN KEY (UID) REFERENCES
   Users(UID) ON DELETE CASCADE,
   FOREIGN KEY (EID) REFERENCES
   Employees(EID) ON DELETE CASCADE);</pre></td></tr>
<tr><td>Complaints_On_Shops</td><td><pre>CREATE TABLE Complaints_On_Shops (
    CID INT,
    SName VARCHAR(50) NOT NULL,
    PRIMARY KEY(CID),
    FOREIGN KEY (Sname) REFERENCES
    Shops(Sname) ON DELETE CASCADE ON UPDATE CASCADE,
    FOREIGN KEY (CID) REFERENCES
    Complaints(CID) ON DELETE CASCADE);</pre></td></tr>
<tr><td>Shops</td><td><pre>CREATE TABLE Shops(
    SName VARCHAR(50),
    PRIMARY KEY(SName));</pre></td></tr>
<tr><td>Complaints_On_Orders</td><td><pre>CREATE TABLE Complains_On_Orders(
   CID INT,
   OID INT NOT NULL,
   PRIMARY KEY(CID),
   FOREIGN KEY (OID) REFERENCES
   Orders(OID) ON DELETE CASCADE,
   FOREIGN KEY (CID) REFERENCES
   Complaints(CID));</pre></td></tr>
</table>

<table>
<tr><td>Orders</td><td><pre>CREATE TABLE Orders(
    OID INT IDENTITY(1,1),
    Order_Date DATETIME NOT NULL,
    Shipping_Address VARCHAR(MAX) NOT NULL,
    Total_Shipping_Cost DECIMAL(19,2) NOT NULL,
    UID INT NOT NULL,
    FOREIGN KEY (UID) REFERENCES
    Users(UID) ON DELETE CASCADE,
    UNIQUE(UID, Order_Date),
    PRIMARY KEY(OID));</pre></td></tr>
<tr><td>Users</td><td><pre>CREATE TABLE Users (
   UID INT IDENTITY(1,1) NOT NULL,
   UName VARCHAR(255) NOT NULL,
   PRIMARY KEY (UID)
);</pre></td></tr>
<tr><td>Products</td><td><pre>CREATE TABLE Products (
  PName VARCHAR(255)NOT NULL,
  Maker VARCHAR(255) NOT NULL,
  Category VARCHAR(255)NOT NULL,
  PRIMARY KEY (PName)
);</pre></td></tr>
<tr><td>Products_In_Shops</td><td><pre>CREATE TABLE Products_In_Shops (
  PName VARCHAR(255) NOT NULL,
  SName VARCHAR(50) NOT NULL,
  SPrice DECIMAL(19,2) NOT NULL CHECK (SPrice >= 0),
  SQuantity INT NOT NULL,
  SPID INT NOT NULL,
  UNIQUE(SName, SPID),
  PRIMARY KEY (PName, SName),
  FOREIGN KEY (PName) REFERENCES Products(PName) ON
DELETE CASCADE ON UPDATE CASCADE,
  FOREIGN KEY (SName) REFERENCES Shops(SName) ON
DELETE CASCADE ON UPDATE CASCADE,
);</pre></td></tr>
<tr><td>Products_In_Orders</td><td><pre>CREATE TABLE Products_In_Orders (
  OPID INT NOT NULL,
  OID INT NOT NULL,
  PName VARCHAR(255) NOT NULL,
  SName VARCHAR(50) NOT NULL,
  OQuantity INT NOT NULL,
  OPrice DECIMAL(19,2) NOT NULL CHECK (OPrice >= 0),
  Delivery_Date DATETIME NOT NULL ,
  Status VARCHAR(50) NOT NULL DEFAULT 'being
processed' CHECK(Status IN ('being</pre></td></tr>
</table>

	`` processed','shipped','delivered','returned')) , PRIMARY KEY (OID, OPID), FOREIGN KEY (PName, SName) REFERENCES Products_In_Shops(PName, SName) ON DELETE CASCADE ON UPDATE CASCADE, FOREIGN KEY (OID) REFERENCES Orders(OID) ON DELETE CASCADE); ```
Price_History (Weak Entity)	``` CREATE TABLE Price_History (PName VARCHAR(255) NOT NULL, SName VARCHAR(50) NOT NULL, Start_Date DATE NOT NULL, Price DECIMAL(19,2) NOT NULL, End_Date DATE, PRIMARY KEY (PName, SName, Start_Date), FOREIGN KEY (PName, SName) REFERENCES Products_In_Shops(PName, SName) ON DELETE CASCADE ON UPDATE CASCADE); ```
Feedback	``` CREATE TABLE Feedback (UID INT, OID INT, OPID INT, Rating INT NOT NULL CHECK(Rating IN (1,2,3,4,5)) , Feedback_Date DATETIME NOT NULL, Comment VARCHAR(255), PRIMARY KEY (UID, OID, OPID), FOREIGN KEY (OID, OPID) REFERENCES Products_In_Orders(OID, OPID) ON DELETE CASCADE, FOREIGN KEY (UID) REFERENCES Users(UID)); ```

Constraint 1

```sql
/* For a particular product in a shop, when the a new price is set, the old price has an end date equal to start date of the new price */
CREATE TRIGGER setOldPriceEndDate
ON Price_History
AFTER INSERT AS
BEGIN
DECLARE @StartDate DATE, @PName VARCHAR(255), @SName VARCHAR(50)
SELECT @StartDate = INSERTED.[Start_Date], @PName = INSERTED.[PName], @SName =
INSERTED.[SName] FROM INSERTED
UPDATE Price_History
SET End_Date = @StartDate
WHERE PName = @PName AND SName = @SName AND End_Date =  NULL
END
```

Explanation

For every new row inserted to Price_History (ie. when the product of a particular store has a price update), we get its Start_Date and update the End_Date of the same product's previous price to the new Start_Date.

Constraint 2

```
/* When an order item is delivered, the delivery date is recorded */
CREATE TRIGGER updateDeliveryDate
ON Products_In_Orders
AFTER UPDATE AS
IF (UPDATE([Status]))
BEGIN
UPDATE Products_In_Orders
SET Delivery_Date = GETDATE()
FROM Products_In_Orders AS P INNER JOIN INSERTED AS i ON P.OID = i.OID AND P.OPID =
i.OPID AND i.Status = 'delivered'
END
```

Explanation

For every Products_In_Order record whose status changes to ‘delivered’, its delivery date is recorded as the time the status update is made.

Query 1

```
/* Find the average price of “iPhone Xs” on Shiokee from 1 August 2021 to 31 August
2021*/

SELECT PName, AVG(Price) AS AveragePrice
FROM Price_History
WHERE PName='IPhone Xs' AND ((End_Date BETWEEN '2021-08-01 00:00:00' AND
'2021-08-31 23:59:59') OR (Start_Date BETWEEN '2021-08-01 00:00:00' AND '2021-08-31
23:59:59'))
GROUP BY PName
```

<u>Explanation</u>

In this query, from the Price_History table, the where clause allows us to only consider the records whose products are ‘IPhone Xs’ and its price between 1 August to 31 August 2021 and from those relevant records, we calculate the average price using AVG aggregation function.

<u>Supporting Tables</u>

1. Price_History

<u>Query Result</u>

	PName	AveragePrice
1	IPhone Xs	1354.956666

Query 2

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

SELECT P.PName, AVG(F.Rating) AS AvgRating
FROM Feedback AS F JOIN Products_In_Orders AS P
ON F.OID = P.OID AND F.OPID =P.OPID
WHERE P.PName in (SELECT B.PName
              FROM Feedback AS A INNER JOIN Products_In_Orders AS B
              ON A.OID = B.OID AND A.OPID =B.OPID
              WHERE A.Rating = 5 AND A.Feedback_Date BETWEEN '2021-08-01 00:00:00'
AND '2021-08-31 23:59:59'
              GROUP BY B.PName
              HAVING COUNT(A.Rating) > 100)
GROUP BY P.PName
ORDER BY AvgRating DESC
```

<u>Explanation</u>

In this query, we join the Feedback table and Products_In_Orders based on OID and OPID. The where clause allows us to consider only the products that exist in the list of products that have at least 100 ratings of "5" in August 2021. Then we find the average rating of those products using AVG aggregation function.

<u>Supporting Tables</u>

1. Feedback
2. Products_In_Orders

<u>Query Result</u>

	PName	AvgRating
1	Best Denki Charger	4
2	Mega Shoes	4
3	TechX Super Phone	3

Query 3

```
/* For all products purchased in June 2021 that have been delivered, find the
average time from the ordering date to the delivery date */

SELECT X.PName, AVG(X.TimeTaken) AS avgTimeTakenInHour
FROM (SELECT DATEDIFF(hour, Order_Date, Delivery_Date) AS TimeTaken, PName
 FROM Orders AS O INNER JOIN Products_In_Orders AS P
 ON O.OID = P.OID
 WHERE (P.Status='delivered'  OR  P.Status='returned') AND O.Order_Date BETWEEN
'2021-06-01 00:00:00' AND '2021-06-30 23:59:59') AS X
GROUP BY X.PName
```

Explanation

In this query, first we Order table joins Products_In_Orders table based on OID, we select those products whose status is "delivered" and ordered in June 2021 and find time difference between delivery date and order date as TimeTaken. We name the filtered table as X. From this X table, we find the average time it takes to deliver in hour using AVG aggregation function group by product name.

Supporting Tables

1. Orders
2. Products_In_Orders

Query Result

	PName	avgTimeTakenInHour
1	Addidas Shoe (M)	65
2	Dell Laptop	201
3	Galaxy Note 10	169
4	HP Laptop	183
5	iPhone 9S	145
6	iPhone Charger	144
7	iPhone X	178

Query 4

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

SELECT TOP(1) X.EID, AVG(X.processingTime) as avgLatencyInSecond
FROM ( SELECT EID, DATEDIFF(second, Filed_Date_Time, Handled_Date_Time) AS
processingTime
           FROM Complaints
           WHERE Complaint_Status = 'addressed' ) AS X
GROUP BY X.EID
ORDER BY AVG(X.processingTime) ASC;
```

Explanation

In this query, we consider “latency” as the average complaint processing time and we consider “processing time” as the time between the time a complaint is filed to the time it is handled. Processing time for each complaint that is handled by an employee is first calculated using DATEDIFF in seconds. Then grouping by EID, we compute average processing time using the AVG aggregation function. The result would be the latency for each employee. This result is then sorted in ascending order, with the lowest latency being at the top of the table. Finally, only one result is displayed using the SELECT TOP (1) function, showing only the employee with the lowest latency (in seconds).

Supporting Tables

1. Complaints

Query Result

	EID	avgLatencyInSecond
1	348	10000

Query 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 */

i) SELECT DISTINCT *
  FROM Products
  WHERE Maker = 'Samsung';

ii) SELECT  P.PName, COUNT(PS.SName) AS NumberOfShops
  FROM Products P, Products_In_Shops PS
  WHERE P.PName = PS.PName AND P.Maker = 'Samsung'
  GROUP BY P.PName;
```

Explanation

i) From the products table, only the records where the maker is Samsung would be selected.
ii) Firstly, Products and Products_In_Shops is joined based on product name (PName). From that table, we select only products whose maker is Samsung. Next, we group by PName in order to see the shops that carry each product. COUNT function is used to count the number of shops that sell each product, and this result is displayed as NumberOfShops, along with the PName.

Supporting Tables

1. Products
2. Products_In_Shops

Query Result

(i)

	PName	Maker	Category
1	Galaxy Note 10	Samsung	Phone
2	Samsung Laptop X	Samsung	Laptop
3	Samsung Laptop Y	Samsung	Laptop

(ii)

	PName	NumberOfShops
1	Galaxy Note 10	9
2	Samsung Laptop X	1
3	Samsung Laptop Y	2

Query 6

```sql
/* Find shops that made the most revenue in August 2021 */
WITH AugRevenue AS
       (SELECT P.SName, SUM(P.OQuantity * P.OPrice) AS Revenue
       FROM Orders AS O, Products_In_Orders AS P
       WHERE O.Order_Date BETWEEN '2021-08-01 00:00:00' AND '2021-08-31 23:59:59'
       AND O.OID = P.OID
       GROUP BY P.SName)

SELECT SName, Revenue
FROM AugRevenue
WHERE Revenue IN
       (SELECT MAX(Revenue)
       FROM AugRevenue)
```

Explanation

1. AugRevenue temporary view: Outputs shop name, and revenue (quantity of product * its respective price) during the month of August. GROUP BY ensures that only products under a specific shop are used in the calculation of that shop's revenue.
2. Finds maximum revenue over the month of August belonging to any shop. Then the maximum revenue is compared to the revenue of all shops over the month of August. Thus the shop name and revenue for all shops with revenue that is equivalent to the maximum revenue for August is outputted.

Supporting Tables

1. Orders
2. Products_In_Orders

Query Result

	SName	Revenue
1	TechX	12737789.36

Query 7

```sql
/* For users that made the most amount of complaints, find the most
expensive products he/she has ever purchased. */
WITH
	ComplaintCount AS
	(SELECT C.UID, COUNT(C.CID) AS NumComplaints
	FROM Complaints AS C
	GROUP BY C.UID),

	MostComplaints AS
	(SELECT UID, NumComplaints
	FROM ComplaintCount AS CC
	WHERE NumComplaints IN
		(SELECT MAX(CC.NumComplaints) AS MaxComplaints
		FROM ComplaintCount AS CC)
	),

	MaxPriceProducts AS
	(SELECT MostComplaints.UID, MostComplaints.NumComplaints, P.PName, P.OPrice
	FROM MostComplaints
	INNER JOIN Orders AS O ON MostComplaints.UID = O.UID
	INNER JOIN Products_In_Orders AS P ON P.OID = O.OID)

SELECT MPP.UID, MPP.NumComplaints, MPP.PName, MPP.OPrice
FROM MaxPriceProducts AS MPP
WHERE MPP.OPrice IN
	(SELECT MAX(MPP.OPrice)
	FROM MaxPriceProducts AS MPP)
GROUP BY MPP.UID, MPP.NumComplaints, MPP.PName, MPP.OPrice
```

Explanation

1. ComplaintCount temporary view: Groups records by user ID and counts complaints to output the number of complaints made by each user.
2. MostComplaints temporary view: Finds maximum number of complaints made by any user. Then compares the maximum complaints to the number of complaints made by each user in ComplaintCount temporary view, to select users who have made the most number of complaints.
3. MaxPriceProducts temporary view: Outputs user ID, maximum complaints, products bought and respective product prices for users who have made the most number of complaints.
4. Finds maximum price of any product bought by users with the most complaints. Then it compares this to prices of products bought by all users with the most complaints. Thus identifies user ID, maximum complaints, and most expensive product names

and prices for users who have made the most number of complaints. GROUP BY ensures that if users have bought a particular product multiple times, it is only displayed once.

Supporting Tables

1. Complaints
2. Products_In_Orders
3. Orders

Query Result

	UID	NumComplaints	PName	OPrice
1	244	7	TechX Super Phone	5000.00

Query 8

```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 */

WITH ProductUIDPair AS
(SELECT P.PName,O.UID
FROM Products_In_Orders AS P, Orders AS O
WHERE P.OID = O.OID),
ProductNotPurchasedBySome AS (SELECT PName
  FROM Products AS PR
  WHERE exists
  ((SELECT UID
    FROM users)
    EXCEPT
    (SELECT UID
    FROM ProductUIDPair
    WHERE PR.PName =  ProductUIDPair.PName)))
SELECT TOP 5 ProductNotPurchasedBySome.PName, SUM(Products_In_Orders.OQuantity) AS
QuantitySold
FROM ProductNotPurchasedBySome, Products_In_Orders, Orders
WHERE ProductNotPurchasedBySome.PName = Products_In_Orders.PName AND Orders.OID =
Products_In_Orders.OID AND Orders.Order_Date BETWEEN '2021-8-1' AND '2021-8-31'
GROUP BY ProductNotPurchasedBySome.PName
ORDER BY QuantitySold DESC
```

Explanation

1. Query interpretation: To find the top 5 items that are not purchased by some users.
2. First we create a temporary views, ProductUIDPair, to find the UID of users for the purchase of the items.
3. Second, we create another temporary 'views', ProductNotPurchasedBySome, to find the products that are not purchased by some customers, or in other words, products that have never been bought by all users previously.
4. After getting this list of products, we rank them based on the number of quantity sold in August 2021

Supporting Tables

1. Products_In_Orders
2. Orders
3. Products
4. Users

Query Result

	PName	QuantitySold
1	iPhone 9S	32
2	HP Laptop	27
3	Galaxy Note 10	26
4	IPhone Xs	23
5	Addidas Shoe (M)	18

Query 9

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

WITH ProductPurchaseHistory AS
   (SELECT P.PName, O.Order_Date, P.OQuantity
   FROM Orders AS O, Products_In_Orders AS P
   WHERE O.OID = P.OID),
ProductMonthlyPurchaseHistory AS
   (SELECT PName, SUM(H.OQuantity) AS PurchaseCount, CAST(CAST(YEAR(H.Order_Date)
AS VARCHAR(4))+'-'+CAST(MONTH(H.Order_Date) AS VARCHAR(2))+'-01'AS DATETIME) AS
Order_Month
   From ProductPurchaseHistory as H
   GROUP BY H.PName, CAST(CAST(YEAR(H.Order_Date) AS
VARCHAR(4))+'-'+CAST(MONTH(H.Order_Date) AS VARCHAR(2))+'-01'AS DATETIME) )
SELECT DISTINCT PName
FROM
(
SELECT Order_Month, PName,PurchaseCount,
PurchaseCount - LAG(PurchaseCount,1,NULL) OVER (PARTITION BY PName ORDER BY
Order_Month) AS Diff1,
PurchaseCount - LAG(PurchaseCount,2,NULL) OVER (PARTITION BY PName ORDER BY
Order_Month) AS Diff2,
PurchaseCount - LAG(PurchaseCount,3,NULL) OVER (PARTITION BY PName ORDER BY
Order_Month) AS Diff3
FROM ProductMonthlyPurchaseHistory
) as X
WHERE Diff3>Diff2 AND Diff2>Diff1 AND Diff1>0
```

<u>Explanation</u>

1. Query interpretation: Find Products that have a history of being increasingly purchased for 3 months in a row. We decide that this means it can be any 3 months **interval** (hence we take 4 months of data) of increasing purchases as long as they are consecutive.
2. First, we create a views, ProductPurchaseHistory, to find the purchase history of products with their date and quantity
3. Secondly, we create a view, ProductMonthlyPurchaseHistory, to find the monthly total quantity ordered of the products.
4. After that by using LAG we are able to find the difference between the total quantity of purchase. We find the 3 differences: current month with 3 months ago (diff3), current month with 2 months ago (diff2) and lastly, current month with 1 month ago (diff1).

	Order_Month	PName	PurchaseCount	Diff1	Diff2	Diff3
1	2021-03-01 00:00:00.000	Addidas Shoe (F)	3	NULL	NULL	NULL
2	2021-04-01 00:00:00.000	Addidas Shoe (F)	12	9	NULL	NULL
3	2021-05-01 00:00:00.000	Addidas Shoe (F)	17	5	14	NULL
4	2021-06-01 00:00:00.000	Addidas Shoe (F)	16	-1	4	13
5	2021-07-01 00:00:00.000	Addidas Shoe (F)	20	4	3	8
6	2021-08-01 00:00:00.000	Addidas Shoe (F)	18	-2	2	1
7	2021-09-01 00:00:00.000	Addidas Shoe (F)	35	17	15	19
8	2021-10-01 00:00:00.000	Addidas Shoe (F)	32	-3	14	12
9	2021-11-01 00:00:00.000	Addidas Shoe (F)	13	-19	-22	-5
10	2021-12-01 00:00:00.000	Addidas Shoe (F)	23	10	-9	-12
11	2022-01-01 00:00:00.000	Addidas Shoe (F)	34	11	21	2
12	2022-02-01 00:00:00.000	Addidas Shoe (F)	15	-19	-8	2
13	2021-03-01 00:00:00.000	Addidas Shoe (M)	14	NULL	NULL	NULL
14	2021-04-01 00:00:00.000	Addidas Shoe (M)	16	2	NULL	NULL
15	2021-05-01 00:00:00.000	Addidas Shoe (M)	40	24	26	NULL
16	2021-06-01 00:00:00.000	Addidas Shoe (M)	40	0	24	26
17	2021-07-01 00:00:00.000	Addidas Shoe (M)	25	-15	-15	9
18	2021-08-01 00:00:00.000	Addidas Shoe (M)	22	-3	-18	-18
19	2021-09-01 00:00:00.000	Addidas Shoe (M)	42	20	17	2
20	2021-10-01 00:00:00.000	Addidas Shoe (M)	41	-1	19	16

5. We then select those that have Diff3> Diff2 >Diff1 >0, to find items that have at least 3 consecutive months of being increasingly purchased.

	Order_Month	PName	PurchaseCount	Diff1	Diff2	Diff3
1	2021-11-01 00:00:00.000	Dell Laptop	28	2	10	12
2	2021-09-01 00:00:00.000	Galaxy Note 10	29	3	4	13
3	2021-10-01 00:00:00.000	Galaxy Note 10	38	9	12	13

6. We then select the DISTINCT PName to find the products

	PName
1	Dell Laptop
2	Galaxy Note 10

Supporting Tables

1. Orders
2. Products_In_Orders

Query Result

	PName
1	Dell Laptop
2	Galaxy Note 10

Additional Queries

Q1: Frequent shoppers are shoppers who have purchased more than 2 items per shop for at least 5 shops in the last 30 days. Who are the top 3 frequent shoppers in terms of the total cost of the items they have purchased?

```
WITH ShopperHistory AS(
  SELECT O.UID,P.SName, COUNT(DISTINCT P.PName) AS itemcount
  FROM Orders O, Products_In_Orders P
  WHERE O.OID = P.OID AND DATEDIFF(day,O.Order_Date,GETDATE()) between 0 and 30
  GROUP By O.UID, P.SName
  ),
  ShopCount AS(
  SELECT UID, Count(SName) NumberOfShop
  FROM ShopperHistory SH
  WHERE itemcount>=2
  GROUP by UID),
  FrequentShopper AS(
  SELECT *
  FROM ShopCount
  WHERE NumberOfShop>=5)
SELECT TOP 5 FS.UID, SUM(P.OQuantity*P.OPrice) AS  TotalCost
FROM FrequentShopper FS, Orders O, Products_In_Orders P
WHERE FS.UID = O.UID AND O.OID = P.OID
GROUP by FS.UID
Order by TotalCost DESC
```

(Explanation given to Lab TA)

Q2: Popular shops are shops which have sold more than 3 items in the last 30 days. Who are the top three shoppers in these popular shops in terms of the number of items they have purchased?

```
WITH PopularShop AS
(SELECT PO.SName, SUM(PO.OQuantity) as ProductSold
  FROM Products_In_Orders AS PO JOIN Orders AS O
  ON PO.OID = O.OID
  WHERE DATEDIFF(day,O.Order_Date,GETDATE()) between 0 and 30
  GROUP BY PO.SName
  HAVING SUM(PO.OQuantity) > 3)

SELECT TOP 3 O.UID, SUM(P.OQuantity) as ItemsPurchased
FROM Orders AS O
   INNER JOIN Products_In_Orders AS P ON P.OID = O.OID
   INNER JOIN PopularShop AS S ON S.SName = P.SName
GROUP BY O.UID
ORDER BY ItemsPurchased DESC;
```

Explanation

1. PopularShop temporary view: Select shops and quantity of products sold by each of the shops in the past 30 days. The data is grouped by the name of the shops, and only shops having sold more than 3 products are included. Items sold do not have to be distinct.
2. Count the number of products bought from any of these popular shops by each shopper (GROUP BY user ID), and select the top 3 shoppers who bought the most products. Items bought do not have to be distinct.

Description of Additional Effort

Creation of mock data

In order to generate enough data for the database, we used an online tool called mockaroo which can generate realistic mock data for values such as names and addresses, generate filler word passages for descriptions, as well as create custom data types and reference existing tables via Ruby.

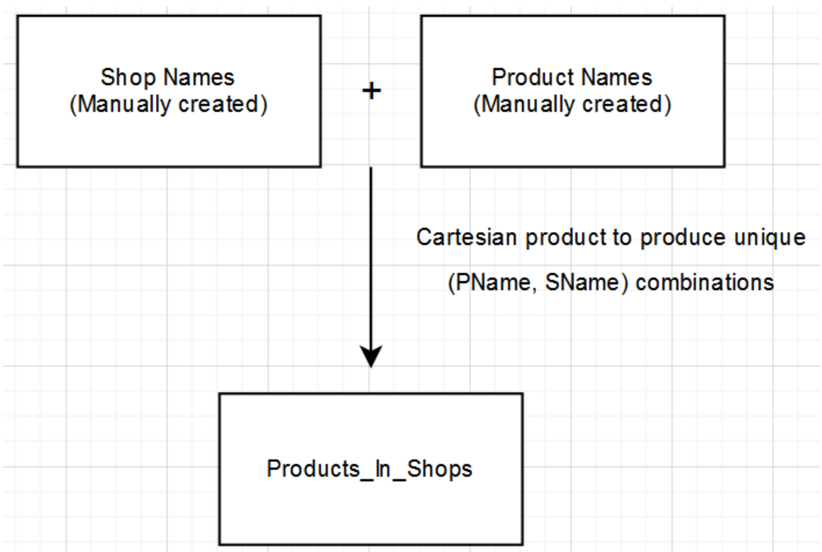
Limitations

While the tool is able to generate random data, there are some incompatibilities if we try to directly use it to generate data for our database tables in a naive approach.

Creating unique data: The service is unable to create **unique values** for attributes (such as words) with the exception of row numbers. This means it is unable to generate tables with names for primary keys such as Shops and unique combinations such as PName, SName.

Cross-referencing data from another table: The service is unable to simultaneously create and cross reference from more than one table at once, meaning it cannot create two tables where one is dependent on the other. This is especially important when considering that many tables have foreign keys which reference an entry in another table.

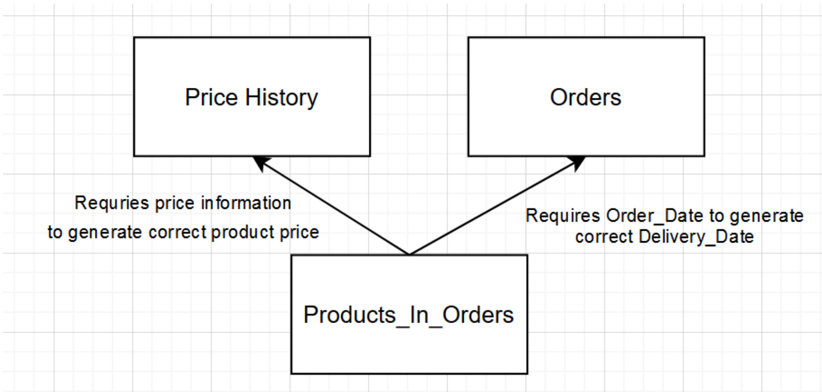
Solutions



In order to get unique keys for PName and SName, we manually create a small set of PName and SName, upon thereafter we are also able to create a table using the cartesian product of the two to create unique combinations of PName and SName. In order to keep things simple, our data carries the assumption that every SName will be selling every

PName, which is also the ‘side-effect’ of the cartesian product.(Every shop sells every product)

In addition, in order to create tables which have data continuity for foreign keys, we built up the tables sequentially and created other tables which may have data dependent on it.



For example, we first generated **Price History** to get data of product prices with changing time, and **Orders** which contain the order date. When creating **Products_In_Orders,** we referenced our earlier two tables; the field **Delivery_Date in Products_In_Orders** is dependent on **Order_Date** in **Orders,** as we cannot have a product delivered before it is ordered. Likewise, the field **OPrice in Products_In_Orders** is dependent on **Price History** as well as **Order_Date,** as we need to record the correct price that the product was selling for when the order was made.