CS Store promotions

The assignment is to create a supermarket database for our CS Store example. The database will be called CS_Store. Most of the tables are what you would expect from the videos on SQL (with a similar, but not always identical set of attributes). The twist compared to the slides (otherwise, if you could just follow the slides without thinking, it would be a bit too easy) is that we will also want to support promotions. I.e. each time you buy some specific set of items (the set of items could incl. multiple of the same item), you get a reduction on your bill, based on which set you brought.

Question 2-5 will each give 2 points. 1 point is given for getting the right output on the test data described at the end – each question will specify what this right output is – and another point is given for getting the right output on another data set. The latter set is kept hidden to avoid you hardcoding the right output in the queries. It will follow the same form as the test data at the end though. Question 6 is similar but will award 2 points for getting the right output on the hidden dataset (still 1 point for the test data at the end), because it is hard – doing Question 5 first should make it manageable for some of you hopefully.

As an aside, each question from 2-6 asks you to create a view, that you should be sure only has each line once (i.e. use DISTINCT! – that said, GROUP BY will automatically give you DISTINCT, even without specifying it, because of how GROUP BY works) and specifies how to sort it – it is necessary for how we grade it. Do make sure you do the latter, since it would be sad for you to lose points for not doing something relatively easy like that! Doing it this way ensures that each question has a unique correct output of the queries on both the test data at the end as well as the hidden set of test data (but there are multiple ways of doing all the queries) and grading will consists of checking that you get the right outputs (there are too many of you to do this by hand and it would lead to errors in grading if I did).

Format

The assignment should be done in .sql format (i.e. the output format from mySQLs workbench) – it is really just a basic text file with the SQL commands written in it and you could do it by writing the file directly, if you wish – I would suggest not to, but you could.

The name of the file should be: <your name>.sql – i.e. if I was handing in, the name of the file should be: Rasmus Ibsen-Jensen.sql. To be precise, use whatever name Canvas is using for you in the People tab as you can see on Canvas, spelled exactly like there (it has to be this way because I have to upload the grades into Canvas, which only supports doing it based on names).

After having created the assignment in .sql, create a .zip file containing just the .sql file and upload it (I am not certain Canvas does not mess with file names, hence why you should zip the file – it does in some cases, so I would not be surprised if it did in this case too). The name of the .zip file does not matter (it is just to make sure Canvas does not mess with the name in the content).

The first two lines of your .sql file should be:

```
CREATE DATABASE CS_Store;

USE CS_Store;
```

And the remainder should be one of 3 kinds of things:

1. CREATE TABLE statements for question 1 (7 in total)
2. CREATE VIEW statements for questions 2-6 (the number of views depends on how you solve the questions and how many you solve, if not all)
3. SQL comments, i.e. lines starting with – (you do not need to make any comments but if you want to write some, you can)

Make sure that you can run the full file through mySQL (starting without an CS_Store database) and after having done so, the CS_Store database should be created and contain the tables and views required from the questions you solved (and perhaps some more views if you feel it would be convenient). This means that **you should remove any statement that causes errors before handing in the assignment,** because mySQL stops when it encounters an error (meaning that the last statements are not executed)! If you do not, you risk getting a far lower grade than otherwise (because the part of your hand-in after the first error will not be graded)...

Question 1) (worth 4 points – ½ points for each table, rounded up)

Make the following set of tables.

- **Customers**(birth_day, first_name, last_name, c_id)
- **Employees**(birth_day, first_name, last_name, e_id)
- **Transactions**(e_id*, c_id*, date, t_id)
- **Items**(price_for_each, amount, name)
- **Promotions**(discount, p_id)
- **ItemsInPromotions**(name*, p_id*, amount)
- **ItemsInTransactions**(name*, t_id*, amount)

Only use data types in the following list: INT, VARCHAR(20), DATE. Each underlined attribute should be the primary key for the table and each attribute with * should have a foreign key to the table with a primary key of the same name (to be precise, the primary keys are the last attribute in each of the first five tables. The last two tables does not have primary keys), e.g. if the tables were R(a,b) and S(b*,c), b in R and c in S should be the primary keys and b in S should reference b in R as a foreign key. Also, to be very clear, each primary key and each foreign key (we reference from) consists of one attribute. Do not add in any constraints besides the mentioned ones (that said, name will be assumed UNIQUE for ItemsInPromotions, but you do not need to add that constraint – it won't matter if you do however. There is an explanation for why this is necessary to assume in Question 6).

Instead of specifying the datatypes explicitly, ensure that the test data defined at the end gets inserted correctly (it seems very likely that you would also guess the same datatypes as these suggests – well, after noting that strings should be VARCHAR(20)) and use DATE if all entries are dates - recall that you should only use data types in the list: INT, VARCHAR(20), DATE. If you follow all of these requirements, each attribute should have a clear, unique datatype (which happens to likely be what you would guess it to be). As an aside, the prices and discounts are measured in pennies (and not directly pounds), to avoid precision issues with floating point numbers.

Question 2)

(worth 2 points – 1 point for getting the right output on the test data and another for the hidden data – see the beginning for more detail!)

Find each distinct customers that has been serviced by an employee with the first name of David. More precisely, you are asked to create a view **DavidSoldTo**, with birth_day, first_name, last_name of the customers an employee with the first name of David has sold to, sorted by birth_day ascending. HINT: You could use NATURAL JOIN for some of it, but you have to be careful if you do and it might be easier not to (also do use DISTINCT please)…

The view should be such that the output of

SELECT * FROM **DavidSoldTo**;

when run on the CS_Store database (after inserting the test data at the end) should be:

birth_day	first_name	last_name
1980-10-10	Evelyn	Lee
1993-07-11	Victor	Davis

Question 3)

(worth 2 points – 1 point for getting the right output on the test data and another for the hidden data – see the beginning for more detail!)

It was found out that someone in the shop on the 2020-9-07 had COVID-19 and we are supposed to reach out to the involved people. Find the (distinct) people (i.e. both employees and customers) in the shop 2020-9-07. People are assumed to be in the shop if and only if they did a transaction that day. More precisely, you are asked to create a view **PeopleInShop** (use DISTINCT), with birth_day, first_name, last_name of the involved people, sorted by birth_day ascending. HINT: You should likely use UNION from Tutorial 2 (which is week 4 of the course) and you would need to use a sub-query here to do the sorting (that or use an intermediate view).

The view should be such that the output of

SELECT * FROM **PeopleInShop**;

when run on the CS_Store database (after inserting the test data at the end) should be:

birth_day	first_name	last_name
1965-12-11	David	Jones
1990-07-23	Olivia	Brown
1999-11-21	Mia	Taylor
2001-03-28	Katarina	Williams

Question 4)

(worth 2 points – 1 point for getting the right output on the test data and another for the hidden data – see the beginning for more detail!)

We want to manage our stock! For each distinct item, find how many are left after all the transactions (the amount in items is from before we made the transactions). More precisely, you are asked to create a view **ItemsLeft**, with name and amount_left (which contains the original amount from items minus the amount sold in total – note that the items sold by a transaction is defined in the

ItemsInTransactions). The view should be sorted by name ascending. HINT: For each typically means GROUP BY… It might be worthwhile to create an intermediate view – just make sure not to overwrite any from the other questions (it is not needed – the query can be done using a single SQL query with no sub-queries – but it might be easier for you to see what to do using an intermediate view)

The view should be such that the output of

SELECT * FROM **ItemsLeft**;

when run on the CS_Store database (after inserting the test data at the end) should be:

name	amount_left
2l of milk	5
3kg sugar	11
6 cans of lemonade	11
Banana	17
Earl Grey tea	8
Pack of butter	8
Pack of rice	18
Roast chicken	0
Toast bread	4

Question 5)

(worth 2 points – 1 point for getting the right output on the test data and another for the hidden data – see the beginning for more detail!)

(This question is really just meant to help guide you to a solution to Question 6). For each distinct transaction, promotion and item in that promotion, find the number of times (i.e. an integer) the transaction satisfies the promotions requirement for that item. More precisely, create a view **PromotionItemsSatisfiedByTransactions**, that returns t_id, p_id, name (for the related transaction, promotion and item respectively) and number_of_times (for how many times the amount of the item in the transaction satisfies the amount of that item in the promotion), the latter should be 0 even if the item is not in the transaction. The view should be sorted by t_id, p_id, name (in that order and all ascending). To explain what you are meant to do, let us look at some examples: if you consider transaction 4, promotion 4 and milk in the test data, then transaction 4 has 10L of milk (more precisely, 5 times 2L of milk) and promotion 4 requires 4L of milk. Thus, the corresponding row should be: 4,4,'2L of milk',2, because the promotion is satisfied 2 times but not 3 times. If you instead consider transaction 2 (but still promotion 4 and milk), that transaction does not incl. milk and the row should be 2,4,'2L of milk',0. As mentioned when creating the tables, it is important to observe that name is UNIQUE in ItemsInPromotions. This is also the case in the hidden data and you should assume this. Why we assume this is explained in Question 6. HINT: Again, for each means GROUP BY and intermediate view(s) might help. Also, you will likely need to use UNION from Tutorial 2 (which is week 4 of the course). To give you a benchmark, my reference solution uses 3 SELECT statements for this (that said, 2 of them as well as the UNION is used to get the 0s if the transaction did not involve the item). A final hint, if your output has .0000 or similar at the end for the numbes, then apply the ROUND function. It will round to the nearest integer.

The view should be such that the output of

SELECT * FROM **PromotionItemsSatisfiedByTransactions**;

when run on the CS_Store database (after inserting the test data at the end) should be:

t_id	p_id	Name	number_of_times
1	1	6 cans of lemonade	0
1	2	Pack of rice	0
1	2	Roast chicken	1
1	3	Pack of butter	1
1	3	Toast bread	0
1	4	2l of milk	1
1	4	3kg sugar	0
1	4	Banana	1
2	1	6 cans of lemonade	2
2	2	Pack of rice	1
2	2	Roast chicken	0
2	3	Pack of butter	0
2	3	Toast bread	0
2	4	2l of milk	0
2	4	3kg sugar	0
2	4	Banana	0
3	1	6 cans of lemonade	1
3	2	Pack of rice	1
3	2	Roast chicken	2
3	3	Pack of butter	1
3	3	Toast bread	0
3	4	2l of milk	0
3	4	3kg sugar	0
3	4	Banana	0
4	1	6 cans of lemonade	0
4	2	Pack of rice	0
4	2	Roast chicken	0
4	3	Pack of butter	0
4	3	Toast bread	0
4	4	2l of milk	2
4	4	3kg sugar	4
4	4	Banana	6
5	1	6 cans of lemonade	5
5	2	Pack of rice	10
5	2	Roast chicken	10
5	3	Pack of butter	10
5	3	Toast bread	5
5	4	2l of milk	5
5	4	3kg sugar	5
5	4	Banana	3

Question 6)

(worth 3 points – 1 point for getting the right output on the test data and another 2 points for the hidden data – see the beginning for more detail!)

We want to know how much each customer should pay for their transaction: For each distinct transaction, find its price after promotions (i.e. find the cost of the transaction without promos and subtract the discount multiplied by the amount of times the transaction satisfies the promotion – a promotion is satisfied at least X times if each involved item is satisfied at least X times. In other words, you need to buy each of the items the required number of times to get the promotion once). More precisely, create a view **PriceOfTransaction**, with t_id and total_cost (the latter should be the cost after promotions of the former), sorted after t_id (ascending). Note that you should assume that name is UNIQUE in ItemsInPromotions, as both mentioned at the beginning and in Question 5. HINT: Again, for each means GROUP BY. Solve Question 5 first and use **PromotionItemsSatisfiedByTransactions** together with some more intermediate view(s). To give you a benchmark, my refence solution has 5 SELECT statements for this question – besides using the view from 5 - (less could be done, but I think it would either be harder to follow or use things that has not been in videos/tutorials – or both)

As an aside, if you do not assume that name is UNIQUE in **ItemsInPromotions** (but still only let an item count towards one promotion), the problem of finding the best promotions to apply, would be NP-complete (it is easy to see when you know what NP-complete is – e.g. using a problem called exact-cover-by-3-sets). While I believe you might not have heard about that before – you will in later courses – it means that 1) most computer scientists does not believe that you can solve it with relative few, simple queries and 2) if you did solve it – even with a general algorithm running in polynomial time (of which a few, simple queries would be a special case) – you would solve likely the hardest problem in computer science, called P vs. NP, would get very famous (in computer science, math and maybe elsewhere), get a million $ from the Cray institute and that is if you did not use it to its full potential!

The view should be such that the output of

SELECT * FROM **PriceOfTransaction**;

when run on the CS_Store database (after inserting the test data at the end) should be:

t_id	total_cost
1	1329
2	396
3	1147
4	2250
5	10585

Test data

```
-- Data for Customers(birth_day, first_name, last_name, c_id)

INSERT INTO Customers VALUES ('1993-07-11','Victor','Davis',1);

INSERT INTO Customers VALUES ('2001-03-28','Katarina','Williams',2);

INSERT INTO Customers VALUES ('1965-12-11','David','Jones',3);

INSERT INTO Customers VALUES ('1980-10-10','Evelyn','Lee',4);

-- Data for Employees(birth_day, first_name, last_name, e_id)

INSERT INTO Employees VALUES ('1983-09-02','David','Smith',1);

INSERT INTO Employees VALUES ('1990-07-23','Olivia','Brown',2);

INSERT INTO Employees VALUES ('1973-05-11','David','Johnson',3);

INSERT INTO Employees VALUES ('1999-11-21','Mia','Taylor',4);

-- Data for Transactions(e_id*, c_id*, date, t_id)

INSERT INTO Transactions VALUES (1,1,'2020-8-11',1);

INSERT INTO Transactions VALUES (3,1,'2020-8-15',2);

INSERT INTO Transactions VALUES (1,4,'2020-9-01',3);

INSERT INTO Transactions VALUES (2,2,'2020-9-07',4);

INSERT INTO Transactions VALUES (4,3,'2020-9-07',5);

-- Data for Items(price_for_each, amount, name)

INSERT INTO Items VALUES (110,22,'2l of milk');

INSERT INTO Items VALUES (99,30,'6 cans of lemonade');

INSERT INTO Items VALUES (150,20,'Pack of butter');

INSERT INTO Items VALUES (450,13,'Roast chicken');

INSERT INTO Items VALUES (99,30,'Pack of rice');

INSERT INTO Items VALUES (20,50,'Banana');

INSERT INTO Items VALUES (200,30,'3kg sugar');

INSERT INTO Items VALUES (150,15,'Toast bread');

INSERT INTO Items VALUES (150,18,'Earl Grey tea');

-- Data for Promotions(discount, p_id)

INSERT INTO Promotions VALUES (99,1);

INSERT INTO Promotions VALUES (200,2);

INSERT INTO Promotions VALUES (150,3);
```

```sql
INSERT INTO Promotions VALUES (150,4);

-- Data for ItemsInPromotions(name*, p_id*, amount)

INSERT INTO ItemsInPromotions VALUES ('6 cans of lemonade',1,2);

INSERT INTO ItemsInPromotions VALUES ('Roast chicken',2,1);

INSERT INTO ItemsInPromotions VALUES ('Pack of rice',2,1);

INSERT INTO ItemsInPromotions VALUES ('Pack of butter',3,1);

INSERT INTO ItemsInPromotions VALUES ('Toast bread',3,2);

INSERT INTO ItemsInPromotions VALUES ('2l of milk',4,2);

INSERT INTO ItemsInPromotions VALUES ('Banana',4,3);

INSERT INTO ItemsInPromotions VALUES ('3kg sugar',4,2);

-- Data for ItemsInTransactions(name*, t_id*, amount)

INSERT INTO ItemsInTransactions VALUES ('6 cans of lemonade',1,1);

INSERT INTO ItemsInTransactions VALUES ('Roast chicken',1,1);

INSERT INTO ItemsInTransactions VALUES ('Pack of butter',1,1);

INSERT INTO ItemsInTransactions VALUES ('Toast bread',1,1);

INSERT INTO ItemsInTransactions VALUES ('2l of milk',1,2);

INSERT INTO ItemsInTransactions VALUES ('Banana',1,3);

INSERT INTO ItemsInTransactions VALUES ('3kg sugar',1,1);

INSERT INTO ItemsInTransactions VALUES ('6 cans of lemonade',2,5);

INSERT INTO ItemsInTransactions VALUES ('Pack of rice',2,1);

INSERT INTO ItemsInTransactions VALUES ('6 cans of lemonade',3,3);

INSERT INTO ItemsInTransactions VALUES ('Roast chicken',3,2);

INSERT INTO ItemsInTransactions VALUES ('Pack of rice',3,1);

INSERT INTO ItemsInTransactions VALUES ('Pack of butter',3,1);

INSERT INTO ItemsInTransactions VALUES ('2l of milk',4,5);

INSERT INTO ItemsInTransactions VALUES ('Banana',4,20);

INSERT INTO ItemsInTransactions VALUES ('3kg sugar',4,8);

INSERT INTO ItemsInTransactions VALUES ('6 cans of lemonade',5,10);

INSERT INTO ItemsInTransactions VALUES ('Roast chicken',5,10);

INSERT INTO ItemsInTransactions VALUES ('Pack of rice',5,10);

INSERT INTO ItemsInTransactions VALUES ('Pack of butter',5,10);
INSERT INTO ItemsInTransactions VALUES ('Toast bread',5,10);

INSERT INTO ItemsInTransactions VALUES ('2l of milk',5,10);

INSERT INTO ItemsInTransactions VALUES ('Banana',5,10);

INSERT INTO ItemsInTransactions VALUES ('3kg sugar',5,10);

INSERT INTO ItemsInTransactions VALUES ('Earl Grey tea',5,10);
```