CS Store promotions

There are 100 points you can get in this assignment. Recall that the assignment contributes 15% to your overall grade in the course.

The assignment is to create a supermarket database for a CS Store supermarket. 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 ala buy one, get one free or buy 5 items, get the 3 cheapest free – you buy some items of a type and get some of the cheapest ones free.

Questions 2-7 gives some number of points each, marked on them. For each question, 2 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 you will also be able to see it when you upload your solution to CodeGrade (since you can upload any number of times before the deadline, I would suggest using it to test your solution against the test data) – the remaining points 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 and unless you hardcode your solution for the test data, it is very likely that if you get points for one part you will get points for the other.

To give you an idea of difficulty: Questions 1-4 are not that hard and I expect nearly everybody to solve these (think 85-95%). If you struggle with them, you can ask for help and I will try to help some (try for some time first – mainly because with 550 of you, I won't have time to help each of you that much). That said, they mostly follow directly from videos/lectures, so maybe go over those again. Questions 5-6 are somewhat hard – you will need to think some - and I would expect around half to do these. I will offer some help for these, but will expect you to try first and then ask for help for how to get further when you get stuck. Question 7 is meant to be quite hard. I would expect most to not do it (think <10% doing it). I will offer clarifications if you struggle to understand what you are meant to do (even though the 4 examples should make it clear). I will not help you solve it. It is meant to get you from 90% to 100% and I feel it fair that you need to think and master SQL for that.

As an aside, each question from 2 to 7 asks you to create a view. Each view should have each line only once (i.e. use DISTINCT! – that said, GROUP BY will automatically give you DISTINCT, even without specifying it, because of how GROUP BY works). Also, each view should be sorted in a specific way, given by the question – 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).

You may use as many views as you wish to solve each question (well, except question 1, since it would not be helpful), but there should be one with the specified name, which is the one that is getting checked for having the right output. E.g. you could have **view1ForQuestion2**, **view2ForQuestion2** and **view3ForQuestion2**, if you feel it would help to do question 2 – most other names are fine too!

Deadline and feedback

The deadline of the assignment is **Monday the 24th of October**. General feedback for the assignment will be given Thursday the 17th of November. The relative long period between those is because you can get extensions (you need to fill out a form online for that however) to hand in, but these can't be longer than 2 weeks or to whenever I give feedback, whichever comes first. There are very many students (~550) on this course and some will therefore, statistically speaking, have very good reasons why they need to delay their assignment as much as possible. To be as nice as I can to them, I will therefore first give feedback a bit over 2 weeks after the deadline… If you have specific concerns about your grade or similar for the first assignment, then, after the general feedback has been released, I will answer questions about your solution and grade over mail.

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 cs_store.sql: You can hand in precisely 1 file and it must have precisely that name (you can hand in multiple times though until the deadline, but only the most recent version count).

And each line should contain only the following:

1. CREATE TABLE statements for question 1 (6 in total)
2. CREATE VIEW statements for questions 2-7 (the number of views depends on how you solve the questions and how many you solve, if not all). Note, you may use any positive number of views to solve each question, but each questions specified view should have the properties requested.
3. SQL comments, i.e. the part of lines after "-- ", i.e. double - followed by space. You do not need to make any, but may do so if you wish.

In particular, **do not include CREATE DATABASE and USE statements**. They will make the tests on the hidden data not work (technically, I create two databases based on your construction, one with the public data given below and one with the hidden data. If you use CREATE DATABASE or USE statements, in essence only 1 is made and it will be marked as if you did not do the other one).

You can include INSERT statements, but the database will be emptied before checking, so it serves little purpose.

It is very unlikely that you would want to remove any views after you have made them.

Make sure that you can run the full file through MySQL when using the CS Store database (starting with an empty CS Store database) and after having done so, the CS_Store database should 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)...

You can submit any number of times before the deadline: We are using CodeGrade for checking these things and whenever you submit, you will see whether your file works for the public dataset. I suggest using it…

Do *not* do the following: Any of the following should ***not*** be done:

- End by removing the database (i.e. DROP DATABASE CS_Store; or similar). It would be the same as handing in an empty file. It will be very easy to see on CodeGrade since everything will stop working.
- Create comments like "----------". MySQL workbench will accept it, but the command line version of MySQL does not, which is what is used to check your file... Just insert an extra space after the second -.
- Use any other order for the columns than what is specified. Since the insert command does not state which columns they insert into, you will put the information in the wrong column and then get hard to understand issues when you attempt to solve the questions.
- Use PARTITION OVER or other new commands. It was not taught in class and is specific to newer versions of MySQL (like the one you would install on your own laptop). Unfortunately, CodeGrade is using an old version of Ubuntu (or at least was last year), where the newest versions of MySQL do not work. Therefore, it will not work when we grade you and you will fail that (and later questions – MySQL will report an error on that line and will not run the rest of your file).

Question 1) (worth 18 points – 3 point for each table)

Make the set of tables that matches the following set of schemas.

- **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)
- **Promotion**(number_to_buy,how_many_are_free,type)
- **Items**(price_for_each, type, amount_in_stock, name)
- **ItemsInTransactions**(name*, t_id*,iit_id)

Each underlined attribute should be the primary key for the table (it happens to be the last attribute in each 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. To be very clear, each primary key and each foreign key (we reference from) consists of one attribute.

Only use data types in the following list: INT, VARCHAR(20), DATE. 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) and use DATE if all entries are dates. 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 are measured in pennies (and not directly pounds), to avoid precision issues with floating point numbers.

Question 2) (worth 15 points – 2 point for getting the right output on the test data and another 13 for the hidden data – see the beginning for more detail!)

We are considering giving Louis Davies a raise but want to check how many transactions she has made in September 2022 first. More precisely, you are asked to create a view **LouisTransactions** with number_of_transactions which should be how many transactions was done by Louis in September 2022 (you may assume she did some and that she is the only employee with that full name – you should NOT assume that her employee id is 4 just because it is in the public data!). As the output is meant to be a single number sorting matters little (still, if you for some reason have chosen a way that generates multiple lines of the same answer, do use DISTINCT).

Note that in the test data, Louis did 3 transactions, of which 2 where in September 2022.

The view should be such that the output of

SELECT * FROM **LouisTransactions**;

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

number_of_transactions
2

Question 3) (worth 15 points – 2 point for getting the right output on the test data and another 13 for the hidden data – see the beginning for more detail!)

The 29th of September, Finn Wilson, one of the employees, had come down with COVID-19 and it was decided that the store would reach out to any customers and employees that had been in the store the 28th, the only day in the week upto the 29th when Finn had been in the store. Find the (distinct) people (i.e. both employees and customers) in the shop the 2022-9-28. 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 and you would need to use a sub-query here to do the sorting (that or use an intermediate view).

Note that in the test data, 3 transactions, but two had the employee in common and two had the customer in common.

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
1983-02-11	Jamie	Johnson
1990-07-03	Anita	Taylor
1991-02-19	Finn	Wilson
1998-08-12	Louis	Davies

Question 4) (worth 15 points – 2 point for getting the right output on the test data and another 13 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). Since that turns out to be tricky if some items have been brought by others not, we will split the question in two. Here, in the first part, we are looking at items of type 1-2. Those types happens to be such that each item of those types have been brought by someone. E.g. banana happens to be an items of one of those types and some of the customers has brought bananas. To be precise, there were 7 bananas before we started selling and we sold one. Therefore, the output has 6 bananas. Similar for the other items of those types.

More precisely, you are asked to create a view **ItemsLeft1**, with name, type 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 type should be either 1 or 2. The view should be sorted by type and then 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).

HINT: You might want to make an intermediate view for finding the amount sold (likely using GROUP BY) and can then give the result by subtracting the amount sold from the amount in stockThe view should be such that the output of

SELECT * FROM **ItemsLeft1**;

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

name	type	amount_left
Bread	1	0
Lemonade	1	0
Banana	2	6
Garlic	2	2

Question 5*) (worth 14 points – 2 point for getting the right output on the test data and another 12 for the hidden data – see the beginning for more detail!)

This question is like question 4, but covers items of type 3 and 4. Items of those types are such that some items of those types have been brought but for others, no items have been brought. E.g. oranges and kiwis are examples of items of those types. Some oranges have been brought, but no kiwis. Specifically, 8 oranges out of the 10 original ones, meaning 2 remain and the 10 original kiwis still remain.

More precisely, you are asked to create a view **ItemsLeft2**, with name, type 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 type should be either 3 or 4. The view should be sorted by type and then name ascending. HINT: changing your query for question 4 slightly should let you cover items like oranges. It is likely however that it will not return kiwis. Conversely, it is also easy to find a view that returns the original amount for each item, which would return the kiwis correctly but would return a too large number for the oranges, compared to what we want here. Consider how to construct from those two views a new view that returns the right amounts. GROUP BY might be helpful…

The view should be such that the output of

SELECT * FROM **ItemsLeft2**;

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

name	type	amount_left
Chicken	3	1
Pasta	3	2
Rice	3	2
Apple	4	2
Grape	4	12
Kiwi	4	10
Orange	4	2

Question 6*) (worth 13 points – 2 point for getting the right output on the test data and another 11 points 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 7 and therefore there is no story to it). Create a view, **IITRanking**, with iit_id, t_id, type, price and rnk (rank is not an allowed name), as follows:

Each row in **itemsInTransactions** is associated with a transaction and an item, which in turn is associated with a type and a price. To each iit_id X in **itemsInTransactions**, we want to have a row in **IITRanking** with transaction id, the type and price of the associated item as well as a rank Y. The rank Y should be such that if you searched for only the type and the transaction of that row, and sorted the output by price (descending) and iit_id (descending), then iit_id X appears on row Y (i.e. it is the Y-th most expensive item brought of that type in that transaction, with iit_id serving as a tie-breaker). Sort the output by transaction id, type, price and iit_id. Each descending. It is perhaps easier to see when you look at the output: Whenever we change t_id or type, we change background color between white and light gray. Note how the rnk increases as price decreases in each color group and we reset rnk to 1 each time we change background color!

HINT: There is a solution based on GROUP BY and then getting rank as a COUNT. You will likely need both **itemsInTransactions** and **items** twice in FROM…

The view should be such that the output of

SELECT * FROM **IITRanking**;

when run on the CS_Store database (after inserting the test data at the end) should be – the output continues on the next page – note, your output is not expected to be colored, it is just for readability:

iit_id	t_id	type	price	rnk
32	6	4	250	1
31	6	4	250	2
30	6	4	250	3
29	5	3	450	1
28	5	3	200	2
27	5	3	200	3

26	5	3	200	4
25	4	2	25	1
24	4	1	200	1
23	4	1	200	2
16	3	4	250	1
15	3	4	250	2
14	3	4	250	3
19	3	4	150	4
18	3	4	150	5
17	3	4	150	6
22	3	4	100	7
21	3	4	100	8
20	3	4	100	9
13	2	4	150	1
12	2	4	150	2
11	2	4	150	3
10	2	4	150	4
7	2	1	200	1
6	2	1	200	2
9	2	1	100	3
8	2	1	100	4
5	1	4	150	1
3	1	2	70	1
4	1	2	25	2
1	1	1	200	1
2	1	1	100	2

Question 7**) (worth 10 points – 2 point for getting the right output on the test data and another8 for the hidden data – see the beginning for more detail!)

The goal of this question is to construct a view, **TransactionCost**, that for each transaction, gives the total cost of it, taking the promotions into account. See below for an explanation of how it works.

Promotion(number_to_buy,how_many_are_free,type)

The idea of the promotions is as follows: Some typesate of item has a promotion which specifies how many to buy of that type and then how many of the cheapest items you brought becomes free. So, e.g. a buy one, get one free on item type 5 means that it contains a row (2,1,5) – you buy 2, the cheapest is free of type 5.

In this store, we want to be nice to the customers that buy promotions (I am unsure whether there is a law or similar requiring this. Nevertheless, it is how it works in CS_Store): Hence, if there is a buy 5 items, get the cheapest 3 for free on some type X, and they buy 14 of that type in a transaction, we make the 3-rd, 4-th, 5-th, 8-th, 9-th and 10-th most expensive item they buy of that type free (we are not so nice that we make the 13-th and/or 14-th most expensive free as well – they would need to buy 15 for that). I.e. we view it as if they brought the 5 most expensive items together of that type and in another transaction the next 5 and so on. If you think about it, this is the best for the customer.

You can view it as us preferring for them to check out fast instead of breaking their transaction into many smaller transactions.

Note that type is the primary key of Promotion, meaning there can't be two promotions on the same type. The question would become too hard otherwise.

Let us go through some of the output on the public data! There are 3 promotions in the public data (the private data will contain other data, including promotions). On type 1 items, there is a buy 3, cheapest is free promotion, on times of type 2, there is a buy 5 get the 3 cheapest free and finally on type 4 there is a buy 4 get the two cheapest for free.

Transaction 1 contains one of each of bread, lemonade, banana, garlic and orange. Bread and lemonade are of type 1, banana and garlic are of type 2 and orange type 4. In none of those cases did we buy enough for a promotion, so we just pay the sum of the costs of those items (the prices are in **Items**). Specifically, we pay 200+100+70+25+150=545.

Transaction 2 brought 2 bread, 2 lemonade and 4 oranges. Bread and lemonade are of type 1 and therefore, the 3rd most expensive should be free, which is one of the lemonades. Oranges are of type 4 and thus 2 of them should be free. We pay 200*2+100+150*2=800.

Transaction 3 contains 3 of each of apples, oranges and grapes. These are of type 4 and there is thus a buy 4, get the 2 cheapest free. Specifically, one of the apples, one of the oranges and 2 of the grapes are free. We pay 250*2+150*2+100=900.

Transaction 4 and 5 did not buy enough for a promotion, so it is just the sum of costs.

Transaction 6 brought 3 apples. Note that it requires 4 items for the promotion, so nothing is free and it is therefore more expensive to buy 3 apples than 4 (because in the latter case, we only paid for 2 apples). The cost is therefore 250*3=750.

HINT: You should likely use **IITRanking** from question 5 as well as modulo (you can use % in MySQL for that) to solve this question. This question might be easier to do with a number of intermediate views.

The view should be such that the output of

SELECT * FROM **TransactionCost**;

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

t_id	cost
1	545
2	800
3	900
4	425
5	1050
6	750

Test data

```
-- Data about Customers(birth_day, first_name, last_name, c_id)
INSERT INTO Customers VALUES('1983-02-11','Jamie','Johnson',1);
INSERT INTO Customers VALUES('1995-10-26','Birgit','Doe',2);
INSERT INTO Customers VALUES('1991-05-15','Finn','Smith',3);
INSERT INTO Customers VALUES('1990-07-03','Anita','Taylor',4);

-- Data about Employees(birth_day, first_name, last_name, e_id)
INSERT INTO Employees VALUES('1964-12-01','Carla','Brown',1);
INSERT INTO Employees VALUES('1984-03-14','Bryan','Williams',2);
INSERT INTO Employees VALUES('1991-02-19','Finn','Wilson',3);
INSERT INTO Employees VALUES('1998-08-12','Louis','Davies',4);

-- Data about Transactions(e_id*, c_id*, date, t_id)
INSERT INTO Transactions VALUES(1,3,'2022-8-9',1);
INSERT INTO Transactions VALUES(4,2,'2022-8-14',2);
INSERT INTO Transactions VALUES(4,4,'2022-9-28',3);
INSERT INTO Transactions VALUES(3,4,'2022-9-28',4);
INSERT INTO Transactions VALUES(4,1,'2022-9-28',5);
INSERT INTO Transactions VALUES(1,4,'2022-9-30',6);

-- Data about Promotion(number_to_buy,how_many_are_free,type)
INSERT INTO Promotion VALUES(3,1,1);
INSERT INTO Promotion VALUES(5,3,2);
INSERT INTO Promotion VALUES(4,2,4);

-- Data about Items(price_for_each, type, amount_in_stock, name)
INSERT INTO Items VALUES(200,1,5,'Bread');
INSERT INTO Items VALUES(100,1,3,'Lemonade');
INSERT INTO Items VALUES(70,2,7,'Banana');
```

```
INSERT INTO Items VALUES(25,2,4,'Garlic');
INSERT INTO Items VALUES(200,3,5,'Rice');
INSERT INTO Items VALUES(450,3,2,'Chicken');
INSERT INTO Items VALUES(300,3,2,'Pasta');
INSERT INTO Items VALUES(100,4,15,'Grape');
INSERT INTO Items VALUES(150,4,10,'Orange');
INSERT INTO Items VALUES(200,4,10,'Kiwi');
INSERT INTO Items VALUES(250,4,8,'Apple');

-- Data about items in ItemsInTransactions(name*, t_id*,iit_id)
INSERT INTO ItemsInTransactions VALUES('Bread',1,1);
INSERT INTO ItemsInTransactions VALUES('Lemonade',1,2);
INSERT INTO ItemsInTransactions VALUES('Banana',1,3);
INSERT INTO ItemsInTransactions VALUES('Garlic',1,4);
INSERT INTO ItemsInTransactions VALUES('Orange',1,5);
INSERT INTO ItemsInTransactions VALUES('Bread',2,6);
INSERT INTO ItemsInTransactions VALUES('Bread',2,7);
INSERT INTO ItemsInTransactions VALUES('Lemonade',2,8);
INSERT INTO ItemsInTransactions VALUES('Lemonade',2,9);
INSERT INTO ItemsInTransactions VALUES('Orange',2,10);
INSERT INTO ItemsInTransactions VALUES('Orange',2,11);
INSERT INTO ItemsInTransactions VALUES('Orange',2,12);
INSERT INTO ItemsInTransactions VALUES('Orange',2,13);
INSERT INTO ItemsInTransactions VALUES('Apple',3,14);
INSERT INTO ItemsInTransactions VALUES('Apple',3,15);
INSERT INTO ItemsInTransactions VALUES('Apple',3,16);
INSERT INTO ItemsInTransactions VALUES('Orange',3,17);
INSERT INTO ItemsInTransactions VALUES('Orange',3,18);
INSERT INTO ItemsInTransactions VALUES('Orange',3,19);
INSERT INTO ItemsInTransactions VALUES('Grape',3,20);
```

```
INSERT INTO ItemsInTransactions VALUES('Grape',3,21);
INSERT INTO ItemsInTransactions VALUES('Grape',3,22);
INSERT INTO ItemsInTransactions VALUES('Bread',4,23);
INSERT INTO ItemsInTransactions VALUES('Bread',4,24);
INSERT INTO ItemsInTransactions VALUES('Garlic',4,25);
INSERT INTO ItemsInTransactions VALUES('Rice',5,26);
INSERT INTO ItemsInTransactions VALUES('Rice',5,27);
INSERT INTO ItemsInTransactions VALUES('Rice',5,28);
INSERT INTO ItemsInTransactions VALUES('Chicken',5,29);
INSERT INTO ItemsInTransactions VALUES('Apple',6,30);
INSERT INTO ItemsInTransactions VALUES('Apple',6,31);
INSERT INTO ItemsInTransactions VALUES('Apple',6,32);
```