DATABASE SYSTEMS

Coursework 2

Dr Paolo Guagliardo

Released: Sun 30 October 2015 – **Due: Fri 11 November 2016 at 16:00**[*]

Before you begin, please read carefully the policy on Late coursework and Academic misconduct and related links, in particular the rules for the publication of solutions to coursework.

Database schema. For this assignment we will use the schema available at:

`<dbshome>/assignment2/schema.sql` (MD5: `12a5a40ea517a4bbf1ed9426f1ca8d23`)

where `<dbshome>` is `/afs/inf.ed.ac.uk/group/teaching/dbs`. This is again a variation of the schema we have used in Coursework 1. Table PRODUCTS now has an additional column *ptype* that indicates the type of product. The same product cannot have two different values of *ptype*. In the test database, there are three types of products: `BOOK`, `MUSIC` and `MOVIE`. If a query asks for books, this means products where the value of *ptype* is the string `BOOK` (case-sensitive), and similarly for the other two product types. However, if a query asks something *for each product type*, you cannot assume that the values for *ptype* will only be `BOOK`, `MUSIC` and `MOVIE`: there is no constraint in the database schema that enforces that, and your query must work independently of which and how many types of products there are in the database it is run against. Indeed, the instances on which your queries will be executed may well contain other, different types of products (as well as customers' countries).

Assignment. Write the following queries in SQL.

(01) Total number of orders placed in 2016 by customers of each country. If all customers from a specific country did not place any orders in 2016, the country *appears in the output with 0 total orders.* Return customer ID and total.

(02) For each product type calculate the average number of times products of that type have been included in an order (taking into account quantities). Orders that do not include any product of a certain type do not contribute to the average for that type (rather than contributing 0). Return the product type and the corresponding average; the latter must be a number with *exactly two decimal places.* Types of products that have never been ordered are not included in the output.

(03) Find invoices that have been vatted. The amount of such invoices is the total of the order they refer to plus 20%. Return the invoice ID. Keep in mind the datatype of "amount" in invoices and *pay special attention to rounding.* Do not use PostgreSQL's `round` function, which is not SQL standard and would likely give you a wrong result.

(04) For each type of product, find the customer who ordered the highest number of products of that type (taking into account quantities). Return the product type and the ID of the customer. If two or more customers are tied for a specific product type they will all be included in the output. If no products of a specific type have ever been ordered, that type will not be in the output.

[*]The submissions will be collected on Sunday 13 November at 16:00. Coursework that has been submitted after the "official" deadline up until the collection time will be considered. There is **no exception** after that time, unless you have been granted a coursework extension.

(05) For each customer, calculate the number of orders placed and the average spend, which is the average total (taking into account quantities and unit prices of ordered products) across all orders placed by that customer. Return the customer ID, the number of orders, and the corresponding average spend. The latter must be a number with *exactly two decimal places*. Orders without detail must be considered in the calculation of the average as having a total of 0. Customers who did not place any orders will be included in the output with 0 orders (not 0.00 or anything else, just 0) and `NULL` average spend.

(06) For each product type, calculate the number of orders consisting *only* of products of that type. Return the product type and the number of orders.

(07) Find "poor readers": customers who have not purchased any books in 2016. Return the customer ID.

(08) Find customers who spent less than 50 in music in the period between January and June 2016 (extremes included). Return the customer ID and the corresponding spend. Customers who have not bought any music in the given period must be returned with 0.00 spend (precisely in this form).

(09) For each customer who has placed at least two orders, calculate the longest number of days ever elapsed between any one order and the next. The only case in which this interval (which you can calculate as the difference of two dates) will be zero is for customers who placed at least two orders and *all of them* were placed in the same day. Whether an order has a detail is irrelevant. Return the customer ID and the corresponding interval (which will be an integer).

(10) Find customers who have placed at least 5 orders, all in different dates, and the interval between any one of their orders and the next (not on the same date) is less than 30 days *on average*. The interval between any two orders placed in the same day (which is 0) does *not* contribute to the average. Whether an order has a detail or not is irrelevant. Return the customer ID only.

To some extent, this query is similar to the previous one, but subtly different. Take a moment to think before doing copy & paste.

Submission instructions.

- Each query must be written in a text file named `<xx>.sql` where `<xx>` is the two-digit number in the list of queries above. For example, the first query will be written in file `01.sql` and the last one in file `10.sql`. **The character encoding of the file must be ASCII; if you use UNICODE your queries will fail.**
- Each file consists of a single SQL statement, terminated by semicolon (`;`). Submitted files that do not contain *exactly one semicolon* will be discarded when the submission is processed and consequently they will not be assessed. **Please pay attention to this; even if it looks like a trivial detail, it is not.**
- If a file contains more than one statement, or a statement that is not a query (such as `CREATE VIEW` or `UPDATE`), it will fail. Do not use comments either, as they may cause a correct query to fail if they contain semicolons.
- Submission is via the `submit` command on DICE.[1] You can submit all your files as follows:

  ```
  submit dbs 1 01.sql 02.sql 03.sql 04.sql 05.sql 06.sql 07.sql 08.sql 09.sql 10.sql
  ```

  You can also submit your files one by one, or in smaller groups, and in any order. For example:

  ```
  submit dbs 1 03.sql
  submit dbs 1 01.sql
  submit dbs 1 04.sql 02.sql
  ```

[1]See how to submit a practical exercise.

Files can be submitted more than once, in which case the previously submitted version will be overwritten (`submit` will ask you whether you want to proceed).

- Before submitting your files, you should check that they run smoothly in PostgreSQL on DICE[2] with the command `psql -h pgteach -f <file>.sql`

Assessment. Your queries will be executed on 10 database instances *without nulls*, initially generated by datafiller and then fine tuned by me. Each query is worth 10 marks, which are allocated as follows:

- 1 mark is given for each test database on which the query returns *all and only* the correct answers *with the correct multiplicities*. If a tuple is expected once and your query returns it twice, this will count as a wrong answer and the mark will be 0.

The queries will not be assessed for their performance or style, as long as they terminate after a "reasonable" time; each query should not take more than a few seconds to run.

Test data. One of the 10 instances on which your queries will be assessed is publicly available at:

- `<dbshome>/assignment2/data/db01.sql` (MD5: `4bc849caf978e8c197d1b6e2f2ea5e8a`)

You can load this instance into your PostgreSQL database on DICE with the following command:

```
psql -h pgteach -1 -f <dbshome>/assignment2/data/db01.sql
```

Few additional test instances will be made available in due course.

Query answers. The answers your queries are expected to return on the test database are available in CSV format at:

- `<dbshome>/assignment2/answers/db01/`

`01.csv`	(MD5: `229cbd738da62c68f4a8fa15f51579e4`)
`02.csv`	(MD5: `c9f0147b48278708683ffcedced57175`)
`03.csv`	(MD5: `8ba7ec79ad33bd86f12ff034183b11b9`)
`04.csv`	(MD5: `565a7bd7e4e05f1caa0020863e4bb408`)
`05.csv`	(MD5: `0873ed8e75acbf5430a98d16d2a96551`)
`06.csv`	(MD5: `34d6d6f23f86e2e0b86f2cf0f6563fe9`)
`07.csv`	(MD5: `cf6836a9ad115a8947e21553d10a1de4`)
`08.csv`	(MD5: `2d8034068b0b20a9373efb5d1bebb821`)
`09.csv`	(MD5: `5278888168d3d71fb040b22ad3b1ebb2`)
`10.csv`	(MD5: `7f489b8f2ad9a8414e1b9a879b4bb111`)

The order in which the rows appear in the answer to your queries is irrelevant for this assignment (no ordering is enforced on the answers, so the DBMS will output rows in an arbitrary order). The names of the columns in the answers are also irrelevant (the above CSV files have no header) so they can be renamed as you wish in the `SELECT` clause. **What is important is the number of columns, their format and the order in which they appear in the output**: the pair $(1, 0.00)$ is not the same as $(0.00, 1)$ or $(1, 0)$. Your query gives a correct answer if it outputs all and only the rows (with repetitions, if that is the case) listed in the corresponding CSV file, no matter on which line.

Other comments. All of the queries can be written using the constructs we have seen in class. Do not use exotic features, especially non-standard ones you may find online in some internet forums. Remember that you are here to learn and think with your own head. You may find useful two standard SQL constructs that we have not seen: `CASE` and `COALESCE` (see the PostgreSQL documentation). The automarker does not have privileges to modify the database; if you want to use views in your queries define them using the `WITH` construct, not the `CREATE VIEW` statement.

[2] See how to use PostgreSQL on DICE machines.