Introduction to Databases

Tutorial 4 (SQL formative coursework)

Dr Paolo Guagliardo

Fall 2020 (week 6)

Database. For this assignment we will use the database available at

> https://ifile.inf.ed.ac.uk/?path=/afs/inf.ed.ac.uk/group/teaching/dbs/2020/coursework/sql-test/data/db1.sql

To import this file into your personal PostgreSQL database, issue the following command in the terminal of a DICE machine:

```
psql -h pgteach -1 -f <dbshome>/2020/coursework/sql-test/data/db1.sql
```

where `<dbshome>` is `/afs/inf.ed.ac.uk/group/teaching/dbs`. This will create several tables and populate them with data.[1] You can learn about the schema by inspecting the first part of the above file. Essentially, it models a typical business scenario where customers place orders consisting of several products, and these orders are invoiced and paid for.

CUSTOMERS have a unique ID (integer), a name (string), and a country (specified as an ISO alpha-3 code, but this is not enforced). You can safely assume that *no two customers will ever have the same ID*.

PRODUCTS have a unique code (integer), a name (string), a textual description (string), and a price of type `NUMERIC`$(5, 2)$. You can safely assume that *no two products will ever have the same code*.

ORDERS have a unique ID (integer), a date (indicating when the order was placed), and a reference (integer) to the customer who placed the order. You can safely assume that *the value of the attribute* `ocust` *will always be a valid customer's ID* (that is, the ID of a customer listed in the CUSTOMERS table).

DETAILS have a reference (integer) to an order, a reference (integer) to a product, and a quantity (`smallint`) indicating how many pieces of a particular product were included in a particular order. Think of this table as a shopping basket. You can safely assume that

- *the value of the attribute* `ordid` *will always be a valid order ID* (that is, the ID of an order listed in the ORDERS table);
- *the value of the attribute* `pcode` *will always be a valid product code* (that is, the code of a product listed in the PRODUCTS table);
- *the value of attribute* `qty` *is at least 1* (the quantity of a product included in an order cannot be less than 1).
- *no two rows will have the same values for the pair of attributes* `ordid` *and* `pcode` (i.e., a product does not belong to the same order more than once, but of course it can be included with a quantity greater than 1).

[1] Tables with the same names that may already be in your database will be deleted, so make a backup if needed. As a matter of fact, to avoid ending up working with the wrong tables, it would be best if you removed everything from your database (`DROP TABLE <table-name>` for all tables) before importing the file.

INVOICES have a unique ID, a reference (integer) to an order, an amount of type `NUMERIC`$(8, 2)$, a date of issue, and a due date. You can safely assume that *no two invoices will have the same ID* and that *the value of the attribute* `ordid` *will always be a valid order ID*. Observe that the amount of an invoice is not necessarily the total worth of the order it refers to: this is to account for additional charges (e.g., shipping costs, VAT) or discounts (e.g., coupons), or even errors in the data.

PAYMENTS have a unique ID (integer), a timestamp (indicating when the payment was made), an amount of type `NUMERIC`$(8, 2)$, and a reference (integer) to an invoice. You can assume that *no two payments will have the same ID* and *the value of the attribute* `invid` *will always be a valid invoice ID*. Note that there may be more than one payment referring to the same invoice and the total amount of all the payments received for a given invoice need not be equal to the amount indicated in the invoice. This is again to model cases when invoices are over-/under-paid and to account for mistakes in the data.

Observe that you cannot make assumptions based on the data in `db1.sql`: your queries will be executed on different instances (of the same schema) on which such assumptions may not hold. For example, you cannot assume that customers' IDs are non-negative; other instances may well use negative integers for the ID of customers. The only assumptions you can safely make are mentioned above for each table: these are schema constraints (we will talk about them in class soon) that are enforced on every instance. You can also assume that no nulls appear in the base tables.

Assignment

Write the following queries in SQL.

(01) Invoices issued after their due date. Return all attributes.

```
SELECT *
FROM   Invoices I
WHERE  I.issued > I.due ;
```

(02) Invoices that were issued before the date in which the order they refer to was placed. Return the ID of the invoice, the date it was issued, the ID of the order associated with it and the date the order was placed.

```
SELECT I.invid, I.issued, O.ordid, O.odate
FROM   Invoices I JOIN Orders O ON I.ordid = O.ordid
WHERE  I.issued < O.odate ;
```

(03) Orders that do not have a detail and were placed before 6 September 2016. Return all attributes.

```
SELECT *
FROM   Orders O
WHERE  O.odate < '2016-09-06'
       AND NOT EXISTS (
           SELECT *
           FROM   Details D
           WHERE  D.ordid = O.ordid );
```

(04) Customers who have not placed any orders in 2016. Return all attributes.

```
SELECT *
FROM   Customers C
WHERE  C.custid NOT IN (
       SELECT O.ocust
       FROM   Orders O
       WHERE  O.odate >= '2016-01-01' AND O.odate <= '2016-12-31' );
```

(05) ID and name of customers and the date of their last order. For customers who did not place any orders, no rows must be returned. For each customer who placed more than one order on the date of their most recent order, only one row must be returned.

```
SELECT   C.custid, C.cname, MAX(O.odate) AS last_order_date
FROM     Customers C JOIN Orders O ON C.custid = O.ocust
GROUP BY C.custid, C.cname ;
```

(06) Invoices that have been overpaid. Observe that there may be more than one payment referring to the same invoice. Return the invoice number and the amount that should be reimbursed.

```
SELECT   I.invid, SUM(P.amount)-I.amount AS overpaid
FROM     Invoices I JOIN Payments P ON I.invid = P.invid
GROUP BY I.invid
HAVING   SUM(P.amount) > I.amount ;
```

(07) Products that were ordered more than 10 times in total, by taking into account the quantities in which they appear in the order details. Return the product code and the total number of times it was ordered.

```
SELECT   D.pcode, SUM(D.qty) AS total
FROM     Details D
GROUP BY D.pcode
HAVING   SUM(D.qty) > 10 ;
```

(08) Products that are usually ordered in bulk: whenever one of these products is ordered, it is ordered in a quantity that *on average* is equal to or greater than 8. For each such product, return product code and price.

```
SELECT   P.pcode, P.price
FROM     Products P JOIN Details D ON P.pcode = D.pcode
GROUP BY P.pcode, P.price
HAVING   AVG(qty) >= 8 ;
```

(09) 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 will not appear in the output.

```
SELECT   C.country, COUNT(O.ordid) AS total_orders
FROM     Orders O JOIN Customers C ON O.ocust = C.custid
WHERE    O.odate >= '2016-01-01' AND odate <= '2016-12-31'
GROUP BY C.country ;
```

(10) For each order without invoice, list its ID, the date it was placed and the total price of the products in its detail, taking into account the quantity of each ordered product and its unit price. Orders without detail must not be included in the answers.

```
SELECT O.ordid, O.odate, SUM(D.qty * P.price) AS total
FROM   Orders O JOIN Details D  ON O.ordid = D.ordid
                JOIN Products P ON P.pcode = D.pcode
WHERE  NOT EXISTS ( SELECT *
                    FROM   Invoices I
                    WHERE  I.ordid = O.ordid )
GROUP BY O.ordid, O.odate ;
```

Mandatory requirement for tutorial marks: attempt at least 5 queries.