Data Science 2019/2020 Take-Home Re-Exam

This is the individual 24-hour take-home portion of the exam of Data Science (DS). The exam is made available via Digital Exam on August 18, 2020, 9:00 and your solution should be handed in via Digital Exam by August 19, 2020, 9:00. A well-formed solution to this exam consists of a PDF file including the answers to all the questions posed below.

Hand-ins for this exam must be individual. Cooperation on or discussion of the contents of the exam with other students is strictly forbidden. The solution you provide should reflect your knowledge of the material alone. The exam is open-book and you are allowed to make use of the book and other reading material of the course. If you use on-line sources for any of your solutions, they must be cited appropriately. By submitting a solution, you commit to have abided by the academic integrity expectations at the University of Copenhagen.

Question 1: Queries (30%)

Rental of summer houses have experienced a new boom, so you have been called upon to analyze data related to summer house bookings collated from multiple booking websites. The data have been represented in the following relational database schema:

summer_house(<u>sid</u>, address, weekly_price, distance_to_beach)
booking(<u>bid</u>, *sid*, month, year)

The summer_house relation records each property along with its ID, address, rental price for a week in DKK, and distance in meters to reach the beach. The booking relation includes all the bookings that have been found for the summer houses in the database. Each booking tuple has a booking ID, the ID of the summer house booked, the month in which the booking took place (e.g., 'July' or 'August'), and the year of the booking (e.g., 2019 or 2020). Note that in the relations above, underlines denote primary keys while italics denote foreign keys.

Answer each of the queries below in the language requested. *NOTE: If the query is formulated in a language different than the one requested, the answer will be disconsidered.*

a) List the addresses of all summer houses that had bookings in July/2020, but no bookings in May/2020. [Language: Relational algebra]

```
bookedJuly := \pi_{sid,address}(
            summer_house
            \natjoin
            \sigma_{month='July' \wedge year = 2020}(booking))

bookedMay := \pi_{sid,address}(
            summer_house
            \natjoin
            \sigma_{month='May' \wedge year = 2020}(booking))

\pi{address}(bookedJuly - bookedMay)
```

b) List the ID(s) of the most booked house(s) ever, i.e., each such house has at least as many bookings as any other house in the database. [Language: Extended relational algebra]

```
houseBookingCount := \gamma_{sid, COUNT(*) → numBookings}(
                                summer_house \natjoin booking)
maxBookings := \gamma_{MAX(numBookings)→ numBookings}(
                                houseBookingCount)
\pi{sid}(houseBookingCount \natjoin maxBookings)
```

c) For each month and year, list the average distance to the beach of houses booked in that month and year. NOTE: Even if a house is booked multiple times in the same month, it should only be counted once towards the average. [Language: SQL]

```
SELECT month, year, AVG(distance_to_beach)
FROM (SELECT DISTINCT month, year, sid
      FROM booking)
     NATURAL JOIN summer_house
GROUP BY month, year
```

NOTE: OK to use `GROUP BY month, year, sid` in inner query instead of `DISTINCT`.

d) List the addresses of all summer houses that are good deals, defined as follows: A house is a good deal if its weekly price is no more than 10% higher than the minimum across all houses and its distance to the beach is no farther than 10% than the closest house to the beach across all houses. [Language: SQL]

```
SELECT address
FROM summer_house
WHERE weekly_price <= 1.1*(SELECT MIN(weekly_price)
                           FROM summer_house)
  AND distance_to_beach <= 1.1*(SELECT MIN(distance_to_beach)
                                FROM summer_house)
```

Question 2: Materialized Views (10%)

You are tasked with supporting analysis of airline sales data recorded in a relational database. The data are stored in a relation with the schema:

```
air_ticket(tid, sales_ts, airline, price)
```

The `air_ticket` relation records all tickets sold including a ticket ID, the timestamp of the sale (e.g., '2020-08-07 10:53:42'), the airline, and the price of the ticket.

An important analysis that is often requested is the quarterly sales volume per airline for the current year and last year. Here, we wish to list the quarter (either 1, 2, 3, or 4), the year, the airline, and the sum of the prices of tickets sold for the quarter and airline. The current quarter is never included in the analysis, since sales are still ongoing. The data for such sales is only considered when all airlines have finished reporting and the quarter is closed by the reporting deadline. Additionally, past sales are almost never updated; such an event only happens in exceptional cases when an airline posts an official correction of their sales data.

Given the above scenario, answer the following questions:

a) Provide a SQL statement creating a materialized view that calculates the analysis above of quarterly sales volume per airline for the current year and last year. NOTE: You may wish to use the date manipulation syntax in https://www.postgresql.org/docs/current/functions-datetime.html, Section 9.9.1.

```
CREATE MATERIALIZED VIEW quarterly_sales AS
SELECT quarter, year, airline, total_sold
FROM (SELECT EXTRACT(QUARTER FROM TIMESTAMP sales_ts) quarter,
             EXTRACT(YEAR FROM TIMESTAMP sales_ts) year,
             airline,
             SUM(price) total_sold
      FROM air_ticket
      GROUP BY EXTRACT(QUARTER FROM TIMESTAMP sales_ts),
               EXTRACT(YEAR FROM TIMESTAMP sales_ts),
               airline)
WHERE year = EXTRACT(YEAR FROM DATE CURRENT_DATE)
   OR year = EXTRACT(YEAR FROM DATE CURRENT_DATE) - 1
```

NOTE: The above is written assuming PostgreSQL dialect, so there is no built-in maintenance policy in the view declaration (i.e., view maintenance must be requested explicitly via the `REFRESH MATERIALIZED VIEW` command). Please see item (b) for a possible maintenance policy. Furthermore, current_date is interpreted at transaction start time (see https://www.postgresql.org/docs/current/functions-datetime.html).

b) Discuss what view maintenance policy you would employ for your materialized view and why.

There are two main classes of update events that need to be considered: insertion of new data for the current quarter and occasional update of past data. For insertion of new data, we remark that it only needs to be considered once the quarter closing deadline is past. So the maintenance can be deferred and periodic (snapshot), executed after each such deadline. For occasional updates due to data correction, we can organize the update process as follows: load the airline corrections in an auxiliary table in the database then issue a single UPDATE statement to air_ticket by referring (joining) agains this auxiliary table. In such a case, a statement-level after update trigger can be created over air_ticket to trigger a refresh of the materialized view. This effectively implements an immediate maintenance policy for data corrections. PostgreSQL does not support incremental view maintenance at the time of writing (see https://wiki.postgresql.org/wiki/Incremental_View_Maintenance), but this kind of policy would be preferred for decomposable operations as in this case.

Question 3: Database Design (35%)

The authorities of multiple countries have convened to discuss how to address the problem of vehicle theft. As a first step, they decide to collect data on the problem by creating a database with a focus on stolen cars. The authorities enlist you to help with the design of their relational database. They have come up with a single relation schema for the data they wish to record:

`stolen_cars(license_plate, country, make, model, year, owner)`

Additionally, the authorities provide you with the following information on functional dependencies for this relation schema:

`license_plate, country → make, model, year, owner`

`make, model, year, owner → license_plate, country`

`owner → country`

Given this input, answer the following questions:

a) What are all the keys of `stolen_cars`? Why?

The first two FDs include all attributes in `stolen_cars`*, so their left sides are superkeys. As it turns out, neither* `license_plate` *nor* `country` *alone are superkeys, so* `{license_plate, country}` *is minimal and a key. Furthermore, no combination of a proper subset of {*`make, model, year, owner`*} forms a superkey. So {*`make, model, year, owner`*} is also minimal and a key. Since* `owner → country`*, then {*`license_plate, owner`*} is also a key. By the FDs, any key must either include {*`license_plate`*} or {*`make, model, year`*}. These sets are not superkeys and neither is their union. Finally, {*`make, model, year, country`*} is not a superkey either.*

b) The relation `stolen_cars` is *not* in BCNF. Why?

Consider the FD `owner → country`*. Here the left side is not a superkey, so this violates the conditions for BCNF.*

c) Is the relation `stolen_cars` in 3NF? Why or why not?

All attributes of `stolen_cars` *are prime, so the right sides of the FDs only include prime attributes. Thus, the relation is in 3NF.*

d) Decompose `stolen_cars` into a set of relations in BCNF, justifying the steps of your decomposition process.

As discussed above, `owner → country` *is an offending nontrivial FD, so we can decompose* `stolen_cars` *into:*

```
stolen_cars0(license_plate, make, model, year, owner)

owners(owner, country)
```

We observe that `stolen_cars0` *is now in BCNF with keys* `{license_plate, owner}` *and* `{make, model, year, owner}`*, as well as the following nontrivial FDs:* `license_plate, owner → make, model, year` *and* `make, model, year, owner → license_plate`*. In addition,* `owners` *is in BCNF with key {*`owner`*} and a single nontrivial FD* `owner → country`*.*

e) Is the decomposition you have derived dependency-preserving? Why or why not?

Since the attribute `country` *is the only one moved out of* `stolen_cars` *and* `owner → country` *is perserved in* `owners`*, we need to check if the following FDs would be preserved by the decomposition*

```
model, make, year, owner → country
license_plate, country → make, model, year, owner
```

(NOTE: The other FDs without the `country` *attribute are directly preserved in* `stolen_cars0`*)*

Dependencies are preserved if $(F_{stolen\_cars0} \cup F_{owners})^+ = F_{stolen\_cars}^+$*. The first FD is preserved since* `owner → country` *is in* F_{owners}*. The second FD is not preserved, however.* F_{stolen_cars0} *includes* `license_plate, owner → make, model, year, owner`*; however, we cannot infer that* `country → owner` *from the projected FDs (and it would be very odd indeed that each country would have only one person in it ☺). So the decomposition is not dependency-preserving.*