CS2102 Tutorial 5: SQL 3

Wk 7, Sem 2, 2021/22

1. Consider the following relational schema consisting of two tables.

```
CREATE TABLE R (
  a  INTEGER PRIMARY KEY,
  b  INTEGER,
  c  INTEGER
);

CREATE TABLE S (
  x  INTEGER PRIMARY KEY,
  y  INTEGER REFERENCES R(a)
);
```

For each of the following queries on the database, state whether the query is a valid SQL query.

(a) Query A

```
SELECT   a, b, SUM(c)
FROM     R JOIN S ON R.a = S.y
GROUP BY a, b;
```

(b) Query B

```
SELECT   b, SUM(c)
FROM     R JOIN S ON R.a = S.y
GROUP BY a, b;
```

(c) Query C

```
SELECT   a, b, SUM(c)
FROM     R JOIN S ON R.a = S.y
GROUP BY a;
```

(d) Query D

```
SELECT   a, b, x, SUM(c)
FROM     R JOIN S ON R.a = S.y
GROUP BY a, b;
```

(e) Query E

```
SELECT   a, b, y, SUM(c)
FROM     R JOIN S ON R.a = S.y
GROUP BY a, b;
```

(f) Query F

```
SELECT   a, b
FROM     R JOIN S ON R.a = S.y
GROUP BY a, b
HAVING   SUM(c) > 10;
```

(g) Query G

```
SELECT   a, b
FROM     R JOIN S ON R.a = S.y
GROUP BY a, b
HAVING   SUM(x) > 10;
```

(h) Query H

```
SELECT   a, b
FROM     R JOIN S ON R.a = S.y
WHERE    SUM(c) > 10;
```

(i) Query I

```
SELECT   a, b
FROM     R JOIN S ON R.a = S.y
GROUP BY a, b
WHERE    SUM(x) > SUM(c);
```

(j) Query J

```
SELECT   b, SUM(c)
FROM     R;
```

Solution:

(a) **Valid**.

(b) **Valid:** It is fine for an attribute in the `GROUP BY` clause to be absent from the `SELECT` clause.

(c) **Valid:** It is fine for attribute `b` to be in the `SELECT` clause since the *primary key* of `R` is included in the `GROUP BY` clause.

(d) **Invalid:** Attribute `x` is missing from the `GROUP BY` clause.

(e) **Invalid:** Attribute `y` and the primary key of `S` are missing from the `GROUP BY` clause.

(f) **Valid**.

(g) **Valid**.

(h) **Invalid:** Aggregate function cannot be used in the `WHERE` clause.

(i) **Invalid:** `HAVING` should be used instead.

(j) **Invalid:** A query that has no `GROUP BY` clause cannot have both aggregate function and non-aggregate attribute in the `SELECT` clause.

2. Consider the same relational schema from the previous question.

```
CREATE TABLE R (
  a  INTEGER PRIMARY KEY,
  b  INTEGER,
  c  INTEGER
);

CREATE TABLE S (
  x  INTEGER PRIMARY KEY,
  y  INTEGER REFERENCES R(a)
);
```

Are these two queries *equivalent*?

(a) Q_1

```
SELECT   COUNT(c)
FROM     R
WHERE    a = 10;
```

(b) Q_2

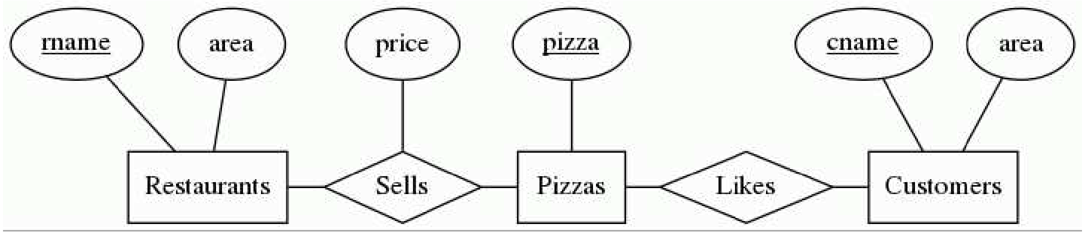
```
SELECT   COUNT(c)
FROM     R
WHERE    a = 10
GROUP BY a;
```

Solution: Q_1 and Q_2 are **NOT** equivalent queries. Consider the case where there is no record in `R` with `a = 10`.

(a) Q_1: The `WHERE` clause will evaluate to an *empty table* and the aggregate function `COUNT(c)` will evaluate to a *single-row* table with a *single-column* value of 0.

(b) Q_2: The `WHERE` clause will evaluate to an *empty table* so the `GROUP BY` clause will **NOT** produce any group. As such, the `COUNT(c)` will **NOT** be evaluated at all. As a result, the query result will be an *empty table*.

3. This question is based on the pizza database schema shown below.



Answer each of the following queries using SQL. For parts (a) to (e), remove duplicate records from all query results.

(a) Find the most expensive pizzas and the restaurants that sell them (*at the most expensive price*). Do **NOT** use any aggregate function in your answer.

(b) Find all restaurant pairs (R_1, R_2) such that the price of the most expensive pizza sold by R_1 is *higher* than than that of R_2. Exclude restaurant pairs where R_2 do not sell any pizza.

(c) For each restaurant that sels some pizza, find the restaurant name and the average price of its pizzas if its average price is higher than \$22. Do **NOT** use the `HAVING` clause in your answer.

(d) For each restaurant R that sells some pizza, let $totalPrice(R)$ denote the total price of all the pizzas sold by R. Find all pairs $(R, totalPrice(R))$ where $totalPrice(R)$ is *higher* than the average of $totalPrice()$ over all the restaurants.

(e) Find the customer pairs (C_1, C_2) such that $C_1 < C_2$ and they like *exactly* the same pizzas. Exclude customer pairs that do not like any pizza. Do **NOT** use the `EXCEPT` operator in your answer.

(f) Write an SQL statement to *increase* the selling prices of pizzas as follows:
- Increase by \$3 if the restaurant is located in `'Central'`.
- Increase by \$2 if the restaurant is located in `'East'`.
- Increase by \$1 *otherwise*.

Solution:

(a) **Solution 1**:

```
SELECT pizza, rname
FROM   Sells
WHERE  price >= ALL (SELECT price FROM Sells);
```

Solution 2:

```
SELECT pizza, rname
FROM   Sells S1
WHERE  NOT EXISTS (
  SELECT 1
  FROM   Sells S2
  WHERE S2.price > S1.price
);
```

(b) **Solution 1**:

```
SELECT R1.rname, R2.rname
FROM   Restaurants R1, Restaurants R2
WHERE  (SELECT MAX(price) FROM Sells
        WHERE rname = R1.rname)
       >
       (SELECT MAX(price) FROM Sells
        WHERE rname = R2.rname);
```

Solution 2:

```
WITH RestMaxPrice AS (
  SELECT rname, (SELECT MAX(price) FROM Sells
                 WHERE rname = R.rname) as maxPrice
  FROM   Restaurants R
)
SELECT R1.rname, R2.rname
FROM   RestMaxPrice R1, RestMaxPrice R2
WHERE  R1.maxPrice > R2.maxPrice;
```

(c) A possible solution

```
WITH RestAvgPrice AS (
  SELECT   rname, AVG(price) as avgPrice
  FROM     Sells
  GROUP BY rname
)
SELECT *
FROM   RestAvgPrice
WHERE  avgPrice > 22;
```

(d) **Solution 1**:

```
WITH RestTotalPrice AS (
  SELECT   rname, SUM(price) as totalPrice
  FROM     Sells
  GROUP BY rname
)
SELECT rname, totalPrice
FROM   RestTotalPrice
WHERE  totalPrice > (SELECT AVG(totalPrice) FROM
    RestTotalPrice);
```

Solution 2:

```
SELECT   rname, SUM(price) as totalPrice
FROM     Sells S
GROUP BY rname
HAVING   SUM(price) > (SELECT SUM(price) / COUNT(
    DISTINCT rname) FROM Sells);
```

WRONG ANSWER 1: The following query is an **invalid** query

```
SELECT   rname, SUM(price) as totalPrice
FROM     Sells S
GROUP BY rname
HAVING   totalPrice > (SELECT SUM(price) / COUNT(
    DISTINCT rname) FROM Sells);
```

totalPrice is ***undefined*** in the HAVING clause as the SELECT clause is conceptually evaluated *after* the HAVING clause.

WRONG ANSWER 2: The following query produce the wrong result

```
SELECT   rname, SUM(price)
FROM     Sells
GROUP BY rname
HAVING   SUM(price) > SUM(price) / COUNT(*);
```

The query above is ***incorrect*** because both *SUM(price)* and *COUNT(*)* in the HAVING clause are computed w.r.t. a group.

(e) **Solution 1**: Customer C! and C2 like exactly the same pizzas if and only if (a) for every pizza that C1 likes, C2 also likes that pizza, and (b) for every pizza that C2 likes, C1 also likes that pizza. Conditions (a) holds if there does not exists any pizza that C1 likes that C2 does not like.

```
SELECT C1.cname, C2.cname
FROM   Customers C1, Customers C2
WHERE  C1.cname < C2.cname
  AND  EXISTS (SELECT 1 FROM Likes L WHERE L.cname = C1.cname)
  AND  NOT EXISTS (
         SELECT 1
         FROM   Likes L1
         WHERE  L1.cname = C2.cname
           AND  NOT EXISTS (
                  SELECT 1
                  FROM   Likes L2
                  WHERE  L2.cname = C2.cname
                    AND  L2.pizza = L1.pizza
                )
       )
  AND  NOT EXISTS (
         SELECT 1
         FROM   Likes L2
         WHERE  L2.cname = C2.cname
           AND  NOT EXISTS (
                  SELECT 1
                  FROM   Likes L1
                  WHERE  L1.cname = C1.cname
                    AND  L1.pizza = L2.pizza
                )
       );
```

Solution 2: This solution is based on the property that if customer C1 likes the set of pizzas $S1$ and customer C2 likes the set of pizzas $S2$, then $S1 = S2$ iff $|S1 \cap S2| = |S1| = |S2|$ where $|X|$ denote the cardinality of set X. Alternatively, you can also use the following equality $|S1 \cap S2| = |S1 \cup S2|$.

Here, we use CTE to solve the problem but a solution without CTE is possible.

```sql
WITH NumLike AS (
  SELECT   cname, COUNT(*) AS num FROM Likes L
  GROUP BY cname
), NumBothLike AS (
  SELECT   L1.cname AS cname1, L2.cname AS cname2, COUNT
    (*) AS num
  FROM     Likes L1, Likes L2
  WHERE    (L1.pizza = L2.pizza) AND (L1.cname < L2.
    cname)
  GROUP BY L1.cname, L2.cname
)
SELECT cname1, cname2
FROM   NumBothLike B
WHERE  num = (SELECT num FROM NumLike N WHERE N.cname =
    B.cname1)
  AND  num = (SELECT num FROM NumLike N WHERE N.cname =
    B.cname2);
```

Solution 3: This solution is based on the property that if customers C1 and C2 like exactly the same pizzas, then for the subset of all tuples $S \subseteq$ `Likes` that are associated with C1 or C2, there must be exactly two tuples in S associated with each distinct pizza in S.

```sql
SELECT C1.cname, C2.cname
FROM   Customers C1, Customers C2
WHERE  C1.cname < C2.cname
  AND  C1.cname IN (SELECT cname FROM Likes)
  AND  NOT EXISTS (
         SELECT   1
         FROM     Likes
         WHERE    cname IN (C1.cname, C2.cname)
         GROUP BY pizza
         HAVING   COUNT(*) <> 2
       );
```

(f) **Solution**: This is simply done using the `UPDATE` statement with `CASE` expression.

```sql
UPDATE Sells S
SET    price = CASE (
                 SELECT area
                 FROM   Restaurants
                 WHERE  rname = S.rname
               )
                 WHEN 'Central' THEN price + 3
                 WHEN 'East' THEN price + 2
                 ELSE price + 1
               END;
```

4. In the lecture, we have discussed an algebraic approach to derive a SQL query involving universal quantification. In this question, we consider another approach to derive the same SQL query based on predicate logic.

   The following predicate logic expression specifies the set of names of students who have enrolled in every course offered by the CS department.

$$\{S.name \mid S \in Students \wedge \forall C(C \in Courses \wedge C.dept = \texttt{'CS'} \implies$$

$$\exists E(E \in Enrolls \wedge E.sid = S.studentId \wedge E.cid = C.courseId))\}$$

   Rewrite the above expression into an equivalent expression without using any universal quantifier and implication operators, and translate it into an equivalent SQL query.

Solution: The query is read as

> A set of students who have taken **ALL** courses (*i.e.*, modules) offered by CS department.

The **ALL** in bold is the universal quantification (*i.e.*, called *forall* with the symbol $\forall$). We can remove this by rewriting the query as:

> A set of students such that there **IS NO** courses (*i.e.*, modules) offered by CS department that the student has not taken.

We have now remove universal quantification and replace it with existential quantification but negated. The new query now can be written as follows:

```sql
SELECT S.name
FROM   Students S
WHERE  NOT EXISTS (
  SELECT 1
  FROM   Courses C
  WHERE  C.dept = 'CS'
    AND  NOT EXISTS (
           SELECT 1
           FROM   Enrolls E
           WHERE  E.sid = S.studentId
             AND  E.cid = C.courseId
  )
);
```

Another way to read this is to see it logically with set comprehension using double negation:

1. $\forall C(C \in Courses \wedge C.dept = \texttt{'CS'} \implies \exists E(E \in Enrolls \wedge E.sid = S.studentId \wedge E.cid = C.courseId))$
2. $\neg\neg\forall C(C \in Courses \wedge C.dept = \texttt{'CS'} \implies \exists E(E \in Enrolls \wedge E.sid = S.studentId \wedge E.cid = C.courseId))$ (*double negation*)

3. $\neg\exists C \neg(C \in Courses \land C.dept = \texttt{'CS'} \implies \exists E(E \in Enrolls \land E.sid = S.studentId \land E.cid = C.courseId))$ *(De Morgan's law)*

4. $\neg\exists C \neg(\neq (C \in Courses \land C.dept = \texttt{'CS'}) \lor (\exists E(E \in Enrolls \land E.sid = S.studentId \land E.cid = C.courseId)))$ *(definition of $\implies$)*

5. $\neg\exists C((C \in Courses \land C.dept = \texttt{'CS'}) \land (\neg\exists E(E \in Enrolls \land E.sid = S.studentId \land E.cid = C.courseId)))$ *(De Morgan's law)*

From here, we can implement $\neg\exists$ can be implemented as `NOT EXISTS` in SQL.