1 - SQL Vs. Relational Algebra

Consider the following example database schema:

- Student (Id, Name, Addr, Status)
- Professor (Id, Name, DeptId)
- Course (DeptId, CrsCode, CrsName, Descr)
- Transcript (StudId, CrsCode, Semester, Grade)
- Teaching (ProfId, CrsCode, Semester)
- Department (DeptId, Name)

A relational algebra expression operating on this scheme is given as follows:

$$\pi_{ProfId}(\sigma_{CrsCode1 \neq CrsCode2}(\rho_{(ProfId1,CrsCode1,Semester1)} Teaching \bowtie_{ProfId1=ProfId2\ AND\ Semester1=Semester2} \rho_{(ProfId2,CrsCode2,Semester2)} Teaching))$$

What is/are the equivalent SQL query(queries) to retrieve the same data?

✔ SELECT T1.ProfId
FROM Teaching T1, Teaching T2
WHERE T1.ProfId = T2.ProfId
AND T1.Semester = T2.Semester
AND T1.CrsCode <> T2.CrsCode

- The relational algebra expression uses the conditional join to find the Ids of all the professors who taught at least two courses in the same semester. Thus, to find an equivalent SQL query, we create the tuple variables T1T1 and T2T2to refer to the two relations being joined. The conditions of the conditional join are then placed in the WHERE clause. These are T1T1.ProfId = T2T2.ProfId AND T1T1.Semester = T2T2.Semester. Next, we look at the selection operator's condition and include it in the WHERE clause as well, which is T1T1.CrsCode <><> T2T2.CrsCode. Lastly, we extract the ProfId attribute satisfying these conditions using the SELECT operation.

✔ SELECT T1.ProfId
FROM Teaching T1, Teaching T2
WHERE T1.Semester = T2.Semester
AND T1.ProfId = T2.ProfId
AND T1.CrsCode <> T2.CrsCode

- The relational algebra expression uses the conditional join to find the Ids of all the professors who taught at least two courses in the same semester. Thus, to find an equivalent SQL query, we create the tuple variables T1T1 and T2T2to refer to the two relations being joined. The conditions of the conditional join are then placed in the WHERE clause. These are T1T1.Semester = T2T2.Semester AND T1T1.ProfId = T2T2.ProfId. Next, we look at the selection operator's condition and include it in the WHERE clause as well, which is T1T1.CrsCode <><> T2T2.CrsCode. Lastly, we extract the ProfId attribute satisfying these conditions using the SELECT operation.

✖ SELECT T1.ProfId
FROM Teaching T1, Teaching T2
WHERE T1.ProfId = T2.ProfId
AND T1.CrsCode <> T2.CrsCode

---

2 - Null Values

It is possible to test for NULL values using comparison operators, such as =, <<, or <><>. Is this statement **True** or **False**?

✔ False

- It is not possible to test for NULL values using comparison operators, such as =, <<, or <><>. Therefore, SQL provides the IS NULL and IS NOT NULL operators for this purpose.

✖ True
✖ True under certain special scenarios.

---

3 - Logical Operators

Consider the following database schema as shown in the figure:

- Products (ProductID, ProductName, SupplierID, CategoryID, Price)
- Orders (OrderDetailsID, OrderID, ProductID, Quantity)

Consider the following SQL query:

```
SELECT  OrderID
FROM  Orders
```

```
GROUP BY OrderID
HAVING max(Quantity) > ALL (SELECT avg(Quantity)
                            FROM Orders
                            GROUP BY OrderID);
```

Which of the following statements are true? Select all.

Products Table

ProductID	ProductName	SupplierID	CategoryID	Price
1	USB Drive	1	1	10
2	Mouse	1	1	8
3	Speakers	1	2	12
4	Monitor	2	2	100
5	Coffee Cup	3	2	12
6	Fighting Illini T-shirt	3	2	25
7	Binder	3	7	3
8	Postcards	3	2	1
9	Illinois Jacket	4	6	70

Orders Table

OrderDetailsID	OrderID	ProductID	Quantity
1	1000	1	12
2	1000	2	10
3	1000	3	15
4	1001	1	8
5	1001	4	4
6	1001	5	6
7	1002	3	5
8	1002	4	18

-

✔ The given SQL query finds the OrderID(s) whose maximum Quantity among all products of that OrderID is greater than the average quantity of every OrderID.

- The subquery after ALL uses the aggregate function avgavg to calculate the average Quantity for each OrderID. The condition max(Quantity) >> ALL (...subquery...) enforces that the highest Quantity for an OrderID should be greater than the average quantity of every OrderID in the Orders table.

✔ The SQL query returns OrderIDs 1000, 1002.

- Average Quantity for Order 1000 = 12.33
- Average Quantity for Order 1001 = 6
- Average Quantity for Order 1002 = 11.5
-
- ALL implies max(Quantity) >> 12.33. The OrderIDs satisfying this are 1000 (Quantity 15) and 1002 (Quantity 18).

✖ The given SQL query finds the OrderID(s) whose maximum Quantity among all products of that OrderID is greater than the average quantity of some OrderID.

---

4 - Subqueries

Consider the two slightly modified "Academic World" relations: **Students(id)** and **Enrolls(id, number, term, instructor, grade)**. To retrieve a table of IDs of the students who take at least one course taught by Prof. Chang, we propose the following two SQL queries:

```
SELECT id
FROM Enrolls
WHERE instructor = "kcchang"
```

```
SELECT id
FROM Students
WHERE id IN (
   SELECT id
   FROM Enrolls
   WHERE instructor = "kcchang"
)
```

Do you expect that these two queries *always* return the same relation? Hint: think about how SELECT handles duplicate tuples by default?

✔ No

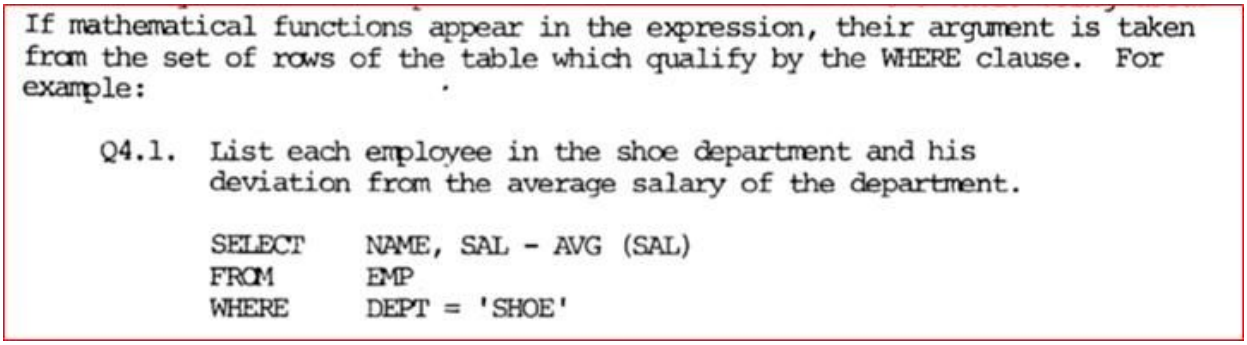
- A student can take more than one course taught by the instructor, and hence the first query can have duplicate Enrolls.id tuples (by default SELECT does not eliminate duplicates). The second query will guarantee uniqueness in the result, because in the Students table, id is the key.

✖ Yes

---

5 - Illegal Aggregate Queries

[Food for Thought] The SQL query below is from the original SEQUEL paper. With what you have learned from the lectures, do you see anything wrong?

- 
If mathematical functions appear in the expression, their argument is taken from the set of rows of the table which qualify by the WHERE clause. For example:

Q4.1. List each employee in the shoe department and his deviation from the average salary of the department.

```
SELECT    NAME, SAL - AVG (SAL)
FROM      EMP
WHERE     DEPT = 'SHOE'
```

✔ Using NAME and SAL along with AVG(SAL) in SELECT is illegal.

- Since this is an aggregate query (using AVG), the SELECT clause can only use group attributes (only AVG(SAL)), not tuple attributes. To fix the query, we need to use a subquery to obtain AVG(SAL) as a single scalar value.

✖ Using DEPT in WHERE along with AVG(SAL) in SELECT is illegal.

✖ It works fine, because AVG(SAL) produces a single scalar value.

---

6 - Group-by

Consider the relations **Students(id, major)** and **Enrolls(id, number, term)**. You are asked to find, for every CS major, the total number of distinct courses taken; i.e., your goal is to produce a table **R(id, count)** that includes an entry for every student majoring in CS. Assume that there are brand-new CS majors who haven't registered for any course yet. Does the following SQL query meet the requirement?

```
SELECT Students.id, COUNT(DISTINCT Enrolls.number)
FROM Students, Enrolls
WHERE Students.id = Enrolls.id AND Students.major = "CS"
GROUP BY Students.id
```

✔ No

- Because those brand-new CS majors haven't registered yet, they're absent in Enrolls, therefore the join operation will eliminate such CS majors, resulting in an incomplete table. The fix is to add those students back to the table via a UNION. Alternatively, the idea of outer join does exactly that: it preserves the tuples in Students that do not match any tuple in Enrolls. Specifically, a "left" outer join will include all the results of an inner join (the type of join we have learned so far), plus the rest of the tuples in the left relation that do not have any matching (according to the join condition) tuple in the right relation.

Those non-matching tuples will appear in the outer join result, with Null values filled in for the attributes from the right relation.

✖ Yes

---

7 - Group-by, Having

To find the "tougher than average" courses in **Enrolls(id, number, grade)**, we need to construct an SQL query resulting in a table **R(number, AVG(grade))** such that each course number in **R** has an average grade (over all the students having taken the course) that is lower than the mean of the average grades over all the courses (each course is equally weighted regardless of how many students have taken it). To get started, we have the following construct:

```
SELECT number, AVG(grade)
FROM Enrolls
GROUP BY number
HAVING AVG(grade) < (___)
```

How would you complete the HAVING clause to get the right result?

✔

```
SELECT AVG(avg_grade)
FROM
(
   SELECT AVG(grade) AS avg_grade
   FROM Enrolls
   GROUP BY number
)
```

- Since each course is weighed equally in computing the mean, we first need to compute the average grade for each course in the inner subquery before taking the average of the averages in the outer subquery.

✖

```
SELECT AVG(grade)
FROM Enrolls
GROUP BY number
```

- This only results in a table of individual average grades of all the courses, we need to average them.

✖

```
SELECT AVG(grade)
FROM Enrolls
```

- This average will weigh each course non-uniformly, based on how many students take the course. This is against the requirement.

---

8 - Set Operations

Using the relation **Enrolls(id, number, term)**, we want to find the IDs of the students who take CS 511 but not CS 411. Of course the easiest way to formulate the SQL query is to use the EXCEPT operator:

```
(
SELECT id
FROME Enrolls
WHERE number = "CS511"
)
EXCEPT
(
SELECT id
FROM Enrolls
WHERE number = "CS411"
)
```

Now say the database system you use was found to have a buggy implementation of EXCEPT and you need to find an alternative approach to get the answer correctly. Check all correct responses below.

✔ With an additional relation **Students(id)**, we can get the same result efficiently without using EXCEPT:

```
SELECT id
FROM Students
WHERE id IN
(
```

```
    SELECT id
    FROM Enrolls
    WHERE number = "CS511"
)
AND id NOT IN
(
    SELECT id
    FROM Enrolls
    WHERE number = "CS411"
)
```

- A Difference B means the set of elements IN A AND NOT IN B, and this is exactly what the query above asks for. Using the additional Students(id) relation is advantageous because 1) it is mostly likely much smaller than Enrolls, hence it's less work to scan through, and 2) there is no need to deduplicate, because id is the key.

✖ By self-joining Enrolls we can get the right answer without using EXCEPT:

```
SELECT DISTINCT id
FROM Enrolls E1, Enrolls E2
WHERE E1.id = E2.id AND E1.number = "CS511" AND E2.number != "CS411"
```

- What this query returns is the set of students who have taken CS 511, and another course that is not CS 411 (including CS 511 itself). It doesn't guarantee that the student does not take CS 411.

✖ No alternative query can achieve the exact same result as using EXCEPT.

---

9 - Aggregate Queries

From the lectures, you have learned that in aggregate queries, the SELECT clause can only have group attributes, but not tuple attributes. Suppose the database implementation you use does not reject such illegal queries with errors. Is there any condition under which using a tuple attribute in the SELECT clause of an aggregate query does NOT result in unpredictable behavior in the query result?

✔ Yes, when the tuple attribute is functionally determined by the group attributes.

- This type of illegal aggregate queries leads to unpredictable behavior (if the system does not error out first) is due to the fact that, very often, for each value instance of the group attributes, there exists more than one corresponding instance of the tuple attribute. Therefore, before the actual grouping, there are multiple such "candidate values"

associated with each group, and thus the system does not know which one to put in the grouping result, hence the unpredictable behavior. Now if, for each group, there is only one value of the tuple attribute (i.e. functional dependence), then there is no other choice but that only value.

✖ Yes, when the tuple attribute functionally determines the group attributes.

- FD in the other direction does not guarantee that for every group there is only one value of the tuple attribute.

✖ No such condition exists, using such illegal queries always result in unpredictable behavior.

---

10 - Querying Databases, The Relational Way

Consider the relations **Students(id, gpa)** and **Enrolls(id, number, term)**. You are asked to find the difference between the average GPA of the students who take CS 411 and that of the students who take CS 511. Which of the following SQL queries will give you the correct answer (as a single-value relation)? Assume that there are a number of students who have taken CS 411 or CS 511 more than once.

✔

```
SELECT S411.avg_gpa - S511.avg_gpa
FROM
(
   SELECT AVG(gpa) AS avg_gpa
   FROM Students
   WHERE id IN
   (
      SELECT id
      FROM Enrolls
      WHERE number = "CS411"
   )
) AS S411,
(
   SELECT AVG(gpa) AS avg_gpa
   FROM Students
   WHERE id IN
   (
      SELECT id
      FROM Enrolls
      WHERE number = "CS511"
```

```
    )
) AS S511
```

-
-

✖

```
SELECT S411.avg_gpa - S511.avg_gpa
FROM
(
    SELECT AVG(gpa) AS avg_gpa
    FROM Students, Enrolls
    WHERE Students.id = Enrolls.id AND number = "CS411"
) AS S411,
(
    SELECT AVG(gpa) AS avg_gpa
    FROM Students, Enrolls
    WHERE Students.id = Enrolls.id AND number = "CS511"
) AS S511
```

- Since the same student can take CS411 or CS511 more than once, those students' GPA will be counted multiple times in the average.

✖

```
SELECT
(
    SELECT AVG(gpa)
    FROM Students
    WHERE id IN
    (
        SELECT id
        FROM Enrolls
        WHERE number = "CS411"
    )
)
-
(
    SELECT AVG(gpa)
    FROM Students
    WHERE id IN
    (
        SELECT id
        FROM Enrolls
```

```
      WHERE number = "CS511"
   )
)
FROM Students
```

- This will get the correct difference in average GPA; however, this value will be repeatedly computed for every student and the query will not return just a single value. This is because the SELECT clause will be processed for every tuple of the Students relation in the FROM clause (and there is no WHERE clause to filter out any tuples, either). Use DISTINCT to deduplicate and get a single value.
