05_subqueries

SQL Sub-queries

- A *sub-query* is a query within another query (or another sub-query)
  - we can have many levels of sub-queries
  - sometimes we refer to sub-queries as *inner* queries and to enclosing queries as *outer* queries

University Database

```
set search_path to university;
```

Non-Correlated Sub-queries

- Also called ***simple*** sub-queries
- *Correlated* sub-queries will be covered later
- Non-correlated sub-queries are independent of their outer (enclosing) queries
  - they can run by themselves
  - they don't depend on something defined in the outer queries

Examples: with 2 non-correlated sub-queries

1. Find the students and instructors with a `@example.com` email address

```
select name, email
from student
where email like '%@example.com'
union
select name, email
from instructor
where email like '%@example.com'
```

2. Find whether or not if there are students and instructors with the same email address

```
select email
from student
intersect
```

```
select email
from instructor;
```

3. Find courses which have never been offered

```
select cid
from course
except
select cid
from offering;

select c.cid
from course c left join offering o on c.cid = o.cid
where o.oid is null;
```

4. Find students not enrolled in any course

```
select sid
from student
except
select sid
from enrollment;
```

5. Find offerings in which no students are enrolled in

```
select oid
from offering
except
select oid
from enrollment;
```

Scalar (Sub-)queries

- Scalar (sub-)queries are the simplest kind of (sub-)queries
- They always return exactly 1 row containing exactly 1 column
- They are often (but not always) obtained by calculating some aggregate function
- *Example*:
  - Find the number of students with a `@example.com` email address

```
select count(sid) as n_students
from student
where email like '%@example.com'
```

- Scalar sub-queries can be used where a scalar (a single value, usually a number) can be used in another query
- We can use scalar sub-queries as operands of operators expecting single values, such as
  - comparison operators `=`, `<`, . . . , and
  - arithmetic operators `+`, `*`, . . .

Examples

1. Find courses that have been offered more often than the DB course
    1. Find the number of times that DB has been offered

   ```
   select count(c.cid)
   from course c
          left join offering o on c.cid = o.cid
   where c.code = 'DB';
   ```

    2. Plug 1 into the HAVING clause of a query calculating the number of times each course has been offered - It might be better to start with a fixed number (say number 2) instead of plugging the first query immediately in the second

   ```
   select c.cid, c.code
   from course c
            left join offering o on c.cid = o.cid
   group by c.cid, c.code
   having count(o.oid) > 3;
   select c.cid, c.code
   from course c
            left join offering o on c.cid = o.cid
   group by c.cid, c.code
   having count(o.oid) > (select count(c.cid)
                          from course c
                                   left join offering o on c.cid = o.cid
                          where c.code = 'DB'
   );
   ```

2. Find the average number of times each course has been offered
    - start by finding the number of times each course has been offered
    - then take the average

```
select c.cid, c.code, count(o.oid) as n_offerings
from course c
         left join offering o on c.cid = o.cid
group by c.cid;
```

```
-- will not work
select c.cid, c.code, avg(count(o.oid)) as n_offerings
from course c
         left join offering o on c.cid = o.cid
group by c.cid;
```

```
select round(avg(n_offerings), 2) as avg_n_offerings
from (select count(o.oid) as n_offerings
      from course c
               left join offering o on c.cid = o.cid
      group by c.cid) as T;
```

3. Find the courses that have been offered more often than the average

(number of times each course has been offered)

```
select c.cid, c.code
from course c
         left join offering o on c.cid = o.cid
group by c.cid
having count(o.oid) > (select avg(n_offerings) as avg_n_offerings
                       from (select count(o.oid) as n_offerings
                             from course c
                                      left join offering o on c.cid = o.cid
                             group by c.cid) as T);
```

with SQL Select Query Format

- In order to express queries with many levels of sub-queries more easily, we can use the `with ... select ...` query style
- We (kind of) define temporary tables before the main `select` query begins
- Then we use the temporary tables in the main `select` query as if they were tables stored in the database

```
with T2 as (
    select avg(n_offerings) as avg_n_offerings
    from (select c.cid, c.code, count(o.oid) as n_offerings
          from course c
                   left join offering o on c.cid = o.cid
          group by c.cid) as T1)
select c.cid, c.code
from course c
         left join offering o on c.cid = o.cid
group by c.cid
having count(o.oid) > (select * from T2);
```

- Don't *over*use the `with` syntax
  - for example, don't rewrite this query

```
select name, email
from student
where email like '%@example.com'
union
select name, email
from instructor
where email like '%@example.com'
```

as

```
with students_example as (
    select name, email
    from student
    where email like '%@example.com'),
    instructor_example as (
        select name, email
        from instructor
        where email like '%@example.com')
select *
from students_example
union
select *
from instructor_example;
```

- While this query is technically correct and equivalent to the original query, the use of `with` to define 2 temporary tables is overkill here, and actually reduces readability
- Use `with` only when sub-queries are complicated, or when there are many levels of sub-queries
- We can also use `with recursive` to write recursive queries

`NULL` Values in SQL

- SQL is using a *3-valued logic* instead of Boolean logic (a 2-valued logic)
- The 3 values are `true`, `false`, and `null` (or `T`, `F` and `N` in the table below)
- The first 2 values have the usual meaning, while `null` can have different meanings:
    - *unknown*
    - *not applicable*
    - *does not matter*
- Logical operators have to be updated to account for `null` values

A	B	NOT A	A OR B	A AND B
T	T	F	T	T
T	F	F	T	F
T	N	F	T	N
F	T	T	T	F
F	F	T	F	F
F	N	T	N	F
N	T	N	T	N
N	F	N	N	F
N	N	N	N	N

### Non-Scalar (Sub)queries

- If a (sub-)query returns more than 1 row and/or more than 1 column, then it is **not** a scalar sub-query
- Attempting to use the normal comparison or arithmetic operators with non-scalar sub-queries will fail if there is more than 1 row

- Some DBMS, such as PostgreSQL, allow some operators to work with sub-queries giving exactly 1 row but many columns
  - PostgreSQL is an ORDBMS, so it is more flexible with data types
  - It will see the single row with multiple columns as a single object with multiple fields
- In general, we need to use special operators to deal with non-scalar sub-queries
  - `IN`, `NOT IN`, `EXISTS`, `NOT EXISTS`, `ANY`, `ALL`

`IN`

- `expression IN (sub-query)`
  - this is the same as $\in$ in mathematical notation (except that we have to deal with `null` values)
  - the sub-query must return exactly 1 column
  - `true` if the expression is equal to 1 of the rows in the sub-query results
  - `false` if the expression is not `null`
    - and there are no `null` values in the sub-query
    - and the expression is not equal to any row in the sub-query
  - `null` if the expression is `null`
    - or if the expression is not equal to any row in the sub-query
    - and there is at least 1 `null` value in the sub-query
- Because SQL is using a 3-valued logic, evaluating `IN` is more complicated
- Recall that is we want to know if a column value is `null`, we cannot use the equality operator `=` because it will always return `null`
- `null` means *unknown* in this case, so we don't know how to compare values to some unknown value
- So we need to use `is null` instead of `= null`
- The `IN` operator is comparing values with `=`, so has soon as it compares with a `null`, it will evaluate to `null`
- So if the expression is equal to `null`, `IN` will evaluate to `null`
- If the expression is not `null`, then it will compare the expression with non-null values first in the sub-query
  - if it finds a match, then the value of `IN` will be true
  - if we don't find a match, then it will check if the sub-query contains `null` values
    - if not, then we know for sure the expression is not in the sub-query, so the value of `IN` will be `false`
    - if there are `null` values, then we don't know for sure if the expression is in the sub-query because we have some *unknown* (`null`) values, so the value of `IN` is `null`
- This example works as expected

```
-- note that (1, 2, 3) is not really a sub-query, but acts like a sub-query
-- it is used to simplify the example
select *
```

```
from course
where cid in (1, 2, 3);
```

```
-- note that (2, 3, 4, null) is not really a sub-query, but acts like a sub-query
-- it is used to simplify the example
select *
from course
where cid in (2, 3, 4, null);
```

- This example is equivalent, and shows how `IN` operators are evaluated internally

```
select *
from course
where cid = 2
   or cid = 3
   or cid = 4
   or cid = null;
```

- This works for the courses with a `cid` value of 2, 3 or 4 because at least 1 of the comparisons will be true and we will get something like `T OR F OR F OR N`, which is true
- But for courses with a `cid` not in the provided set, we will get `null` because `F OR F OR F OR N` is `N`
- This doesn't create an issue because rows with a `where` condition will be dropped
- But if we negate `IN` to get a `NOT IN` operator, we will get into trouble

```
select *
from course
where cid not in (2, 3, 4, null);

select *
from course
where cid not in (select cid from offering);

select *
from course
except
select c.*
from course c inner join offering o on c.cid = o.cid;

select c.*
from course c left join offering o on c.cid = o.cid
where o.oid is null;

select *
from instructor
```

```
where iid not in (select iid from offering);
```

- We get nothing
- But the course with `cid = 1` is not in the sub-query, so why don't we get it?
- It's because of the `null` value
  - `1 in (2, 3, 4, null)` evaluates to `null`
  - and `1 not in (2, 3, 4, null)` evaluates to `not null`, which is `null`
- So `NOT IN` queries are dangerous because of `null` values
- The following query is correct because we know for sure that `cid` in course cannot be `null`
- So we can find courses that have never been offered in this way

```
insert into course(name, code, credits)
values ('Data Structures', 'DS', 3);
-- delete from course where code = 'DS';
select *
from course
where cid not in (
    select cid
    from offering);
```

- But trying to do something similar for instructors will create problems because `iid` in offering can be `null`
- We need to explicitly discard `null` values in the sub-query in order for the query to return the correct results

```
insert into instructor(name, email, department)
values ('John', 'john@bbb.com', 'ECE');
-- delete from instructor where name = 'John';
select *
from instructor
where iid not in (
    select iid
    from offering);
```

```
select *
from instructor
where iid not in (
    select iid
    from offering
    where iid is not null);
```

Recommendation: don't use `NOT IN`

Recommendation: use a `left join` instead

- Not only the left join (or outer joins in general) forces you to think about `null` values (and deal with them correctly), but performance-wise, left joins will usually be more efficient
- Using left joins avoids dealing with SQL's 3-valued-logic

```
select i.*
from instructor i
         left join offering o on i.iid = o.iid
where o.iid is null;
```

`ANY` and `ALL`

- `ANY` and `ALL` are used as modifiers to operators (usually comparison operators)
    - `expression operator ANY (sub-query)`
        - `true` when there exists a row *r* in the sub-query such that `expression operator r` is true
        - `false` when for all rows r in the sub-query, `expression operator r` is false and there are no `null` values in the sub-query
        - `null` when for all rows r in the sub-query, `expression operator r` is false and there is at least 1 `null` value in the sub-query
    - `IN` is equivalent to `=ANY`
- `expression operator ALL (sub-query)`
    - `true` when for all rows *r* in the sub-query, `expression operator r` is true
    - `false` when `expression operator r` is false for at least 1 row in the sub-query
    - `null` when for all rows *r* in the sub-query, `expression operator r` is not false and there is at least 1 `null` value in the sub-query
- `NOT IN` is equivalent to `<> ALL`

Example

1. Find the courses that have been offered the most often
    - Note that there can be many "most offered" courses
    - Start from the other query *Find the courses that have been offered more often than the average* and replace `avg` by `max`

```
with T as (
    select max(n_offerings) as max_n_offerings
    from (select c.cid, c.code, count(o.oid) as n_offerings
          from course c
                   left join offering o on c.cid = o.cid
          group by c.cid) as T)
select c.cid, c.code
```

```
from course c
         left join offering o on c.cid = o.cid
group by c.cid
having count(o.oid) = (select * from T);
```

- If we had many courses equal for the first place, they would be listed
- We can rewrite the query with a `>=ALL`
    - but be careful with `null` values in general
    - here it's not an issue since the sub-query cannot return `null` values because of the aggregate function
    - but it's not the case of all possible sub-queries

```
with T as (
    select c.cid, c.code, count(o.oid) as n_offerings
    from course c
             left join offering o on c.cid = o.cid
    group by c.cid)
select cid, code
from T
where n_offerings >= ALL (select n_offerings from T);
```

- If we were not using `with`, we would have to write essentially the same query twice, including the `group by`, and move the `where` to a `having`

```
select c.cid, c.code
from course c
         left join offering o on c.cid = o.cid
group by c.cid
having count(o.oid) >= ALL (
    select count(o.oid) as n_offerings
    from course c
             left join offering o on c.cid = o.cid
    group by c.cid);
```

Meaning of some `operator ALL` and `operator ANY` queries

- special cases: `null` values
- `>=ALL`: (greater than or) equal to the largest value in the sub-query
- `>=ANY`: not smaller that the smallest value in the sub-query
- `<>ALL`: same as `NOT IN`
- `<>ANY`: different from at least 1 value in the sub-query
- `=ALL`: all values in the sub-query are the same (there's no value that is different)
- `=ANY`: same as `IN`

Recommendation: try to avoid queries that can be messed up by `null` values

`EXISTS` and `NOT EXISTS`

- Checks whether or not a sub-query is empty (returns 0 rows)
- In other words, checks if there exists (or not) some rows in the sub-query
- `EXISTS` is a unary operator since it takes only 1 argument to the right of it
  - Likewise `NOT` is a unary operator, negating its argument on the right
- Most of the time, `EXISTS` is used with correlated sub-queries

Correlated Sub-queries

- When a sub-query depends on a table specified in the outer query, then the sub-query is said to be *correlated*
- It means the correlated sub-query cannot be executed by itself in isolation
- *Example*:
  - Find courses that have been offered at least twice

```
-- without correlated sub-queries
select c.cid, name, code
from course c
         inner join offering o on c.cid = o.cid
group by c.cid
having count(oid) >= 2;
```

```
-- with correlated sub-queries
select distinct c.cid, name, code
from course c
         inner join offering o1 on c.cid = o1.cid
where exists(
              select *
              from offering o2
              where o1.cid = o2.cid
                and o1.oid <> o2.oid
          );
```

```
-- without correlated sub-queries, but with 2 copies of offering
select distinct c.cid, name, code
from course c
         inner join offering o1 on c.cid = o1.cid
         inner join offering o2 on o1.cid = o2.cid
where o1.oid <> o2.oid;
```

Examples

1. Find courses that have been offered at most once

```
-- without correlated sub-queries
-- need a left join here because we need the never offered courses
select c.cid, name, code
from course c
```

```
         left join offering o on c.cid = o.cid
group by c.cid
having count(oid) <= 1;

-- with correlated sub-queries
-- need a left join here because we need the never offered courses
select distinct c.cid, name, code
from course c
         left join offering o1 on c.cid = o1.cid
where not exists(
        select *
        from offering o2
        where o1.cid = o2.cid
          and o1.oid <> o2.oid
    );
```

2. Find courses that have never been offered
   - We have seen 2 ways previously
     - with `except` (awkward to get course names and codes, need a join anyway to get these)
     - with a left join (recommended)

   ```
   select cid, name, code
   from course
   except
   select o.cid, name, code
   from offering o
            inner join course c on o.cid = c.cid;
   select c.*
   from course c
            left join offering o on c.cid = o.cid
   where oid is null;
   ```

   - It is also possible with a correlated sub-query, but it's less readable
     - it reads as "select all course for which an offering doesn't exist"

   ```
   select c.cid, name, code
   from course c
   where not exists(
           select *
           from offering o
           where c.cid = o.cid
       );
   ```