SQL Part 2 - Joins and Subqueries

In our Part 1 note we only looked into querying from one table. Often, however, the data we need to answer a question will be stored in multiple tables. To query from two tables and combine the results we use a **join**.

1   Cross Join

The simplest join is called **cross join**, which is also known as a cross product or a cartesian product. A cross join is the result of combining every row from the left table with every row from the right table. To do a cross join, simply comma separate the tables you would like to join. Here is an example:

```sql
SELECT *
FROM courses, enrollment;
```

If the courses table looked like this:

num	name
CS186	DB
CS188	AI
CS189	ML

And the enrollment table looked like this:

c_num	students
CS186	700
CS188	800

The result of the query would be:

num	name	c_num	students
CS186	DB	CS186	700
CS186	DB	CS188	800
CS188	AI	CS186	700
CS188	AI	CS188	800
CS189	ML	CS186	700
CS189	ML	CS188	800

The cartesian product often contains much more information than we are actually interested in. Let's say we wanted all of the information about a course (num, name, and num of students enrolled in it). We cannot just blindly join every row from the left table with every row from the right table. There are rows that have two different courses in them! To account for this we will add a **join condition** in the `WHERE` clause to ensure that each row is only about one class.

To get the enrollment information for a course properly we want to make sure that `num` in the courses table is equal to `c_num` in the enrollment table because they are the same thing. The correct query is:

SQL Part 2 - Joins and Subqueries

```
SELECT *
FROM courses, enrollment
WHERE num = c_num;
```

which produces:

num	name	c_num	students
CS186	DB	CS186	700
CS188	AI	CS188	800

Notice that CS189, which was present in the `courses` table but not in the `enrollment` table, is not included. Since it does not appear as a `c_num` value in enrollment, it cannot fulfill the join condition `num = c_num`.

If we really want CS189 to appear anyway, there are ways to do this that we will discuss later.

2 Inner Join

The cross join works great, but it seems a little sloppy. We are including join logic in the `WHERE` clause. It can be difficult to find what the join condition is. In contrast, the **inner join** allows you to specify the condition in the `ON` clause. Here is the syntax:

```
SELECT column_name(s)
FROM table1
INNER JOIN table2
ON table1_column_name = table2_column_name;
```

The `table1_column_name = table2_column_name` is the join condition. Let's write the query that gets us all of the course information as an inner join:

```
SELECT *
FROM courses INNER JOIN enrollment
ON num = c_num;
```

This query is logically the exact same query as the query we ran in the previous section. The inner join is essentially syntatic sugar for a cross join with a join condition in the `WHERE` clause like we demonstrated before.

3 Outer Joins

Now lets address the problem we encountered before when we left out CS189 because it did not have any enrollment information. This situation happens frequently. We still want to keep all the data from a relation even if it does not have a "match" in the table we are joining it with. To fix this problem we will use a left outer join. The **left outer join** makes sure that every row from

the left table will appear in the output. If a row does not have any matches with the right table, the row is still included and the columns from the right table are filled in with `NULL`. Let's fix our query:

```
SELECT *
FROM courses LEFT OUTER JOIN enrollment
ON num = c_num;
```

This will produce the following output:

num	name	c_num	students
CS186	DB	CS186	700
CS188	AI	CS188	800
CS189	ML	NULL	NULL

Notice that CS189 is now included and the columns that should be from the right table (`c_num`, `students`) are `NULL`.

The **right outer join** is the exact same thing as the left outer join but it keeps all the rows from the right table instead of the left table. The following query is identical to the query above that uses the left outer join:

```
SELECT *
FROM enrollment RIGHT OUTER JOIN courses
ON num = c_num;
```

Notice that I flipped the order of the joins and changed `LEFT` to `RIGHT` because now `courses` is on the right side.

Let's say we add a row to our `enrollment` table now:

c_num	students
CS186	700
CS188	800
CS160	400

But we still want to present all of the information that we have. If we just use a left or a right join we have to pick between using all of the information in the left table or all of the information in the right table. With what we know so far, it is impossible for us to include the information that we have about *both* CS189 and CS160 because they occur in different tables and do not have matches in the other table. To fix this we can use the **full outer join** which guarantees that all rows from each table will appear in the output. If a row from either table does not have a match it will still show up in the output and the columns from the other table in the join will be `NULL`.

To include all of data we have let's change the query to be:

```sql
SELECT *
FROM courses FULL OUTER JOIN enrollment
ON num = c_num;
```

which produces the following output:

num	name	c_num	students
CS186	DB	CS186	700
CS188	AI	CS188	800
CS189	ML	NULL	NULL
NULL	NULL	CS160	400

4 Name Conflicts

Up to this point our tables have had columns with different names. But what happens if we change the enrollment table so that it's c_num column is now called num?

num	students
CS186	700
CS188	800
CS160	400

Now there is a num column in both tables, so simply using num in your query is ambiguous. We now have to specify which table's column we are referring to. To do this, we put the table name and a period in front of the column name. Here is an example of doing an inner join of the two tables now:

```sql
SELECT *
FROM courses INNER JOIN enrollment
ON courses.num = enrollment.num;
```

The result is:

num	name	num	students
CS186	DB	CS186	700
CS188	AI	CS188	800

It can be annoying to type out the entire table name each time we refer to it, so instead we can **alias** the table name. This allows us to rename the table for the rest of the query as something else (usually only a few characters). To do this, after listing the table in the FROM we add AS <alias_name>. Here is an equivalent query that uses aliases:

```sql
SELECT *
FROM courses AS c INNER JOIN enrollment AS e
ON c.num = e.num;
```

SQL Part 2 - Joins and Subqueries

num	name	num	students
CS186	DB	CS186	700
CS188	AI	CS188	800

5   Natural Join

Often in relational databases, the columns you want to join on will have the same name. To make it easier to write queries, SQL has the **natural join** which automatically does an equijoin (equijoin = checks if columns are equivalent) on columns with the same name in different tables.

The following query is the same as explicitly doing an inner join on the `num` columns in each table:

```
SELECT *
FROM courses NATURAL JOIN enrollment;
```

The join condition: `courses.num = enrollment.num` is implicit. While this is convenient, natural joins are not often used in practice because they are confusing to read and because adding columns that are not related to the query can change the output.

6   Subqueries

Subqueries allow you to write more powerful queries. Let's look at an example...

Let's say you want to find the course num of every course that has a higher than average num of students. You cannot include an aggregation expression (like `AVG`) in the `WHERE` clause because aggregation happens after rows have been filtered. This may seem challenging at first, but subqueries make it easy:

```
SELECT num
FROM enrollment
WHERE students >= (
    SELECT AVG(students)
    FROM enrollment;
);
```

The output of this query is:

num
CS188
CS186

The inner subquery calculated the average and returned one row. The outer query compared the `students` value for each row to what the subquery returned to determine if the row should be kept.

Note that this query would be invalid if the subquery returned more than one row because `>=` is meaningless for more than one number. If it returned more than one row we would have to use a set operator like `ALL`.

7   Correlated Subqueries

The subquery can also be correlated with the outer query. Each row essentially gets plugged in to the subquery and then the subquery uses the values of that row. To illustrate this point, let's write a query that returns all of the classes that appear in both tables.

```
SELECT *
FROM classes
WHERE EXISTS (
  SELECT *
  FROM enrollment
  WHERE classes.num = enrollment.num
);
```

As expected, this query returns:

num	name
CS188	AI
CS186	DB

Let's start by examining the subquery. It compares the `classes.num` (the num of the class from the current row) to every `enrollment.num` and returns the row if they match. Therefore, the only rows that will ever be returned are rows with classes that occur in each table. The `EXISTS` keyword is a set operator that returns true if any rows are returned by the subquery and false if otherwise. For `CS186` and `CS188` it will return true (because a row is returned by the subquery), but for `CS189` it will return false. There are a lot of other set operators you should know (including `ANY`, `ALL`, `UNION`, `INTERSECT`, `DIFFERENCE`, `IN`) but we will not cover any others in this note (there is plenty of documentation for these operators online).

8   Subqueries in the From

You can also use subqueries in the `FROM` clause. This lets you create a temporary table to query from. Here is an example:

```
SELECT *
FROM (
    SELECT num
    FROM classes
) as a
WHERE num = 'CS186';
```

Returns:

num
CS186

The subquery returns only the `num` column of the original table, so only the `num` column will appear in the output. One thing to note is that subqueries in the `FROM` cannot usually be correlated with other tables listed in the `FROM`. There is a work around for this, but it is out of scope for this course.

A cleaner way of doing this is using common table expressions (or views if you want to reuse the temporary table in other queries) but we will not cover this in the note.

9 Practice Questions

We will reuse the dogs table from part 1:

```sql
CREATE TABLE dogs (
 dogid integer,
 ownerid integer,
 name varchar,
 breed varchar,
 age integer,
 PRIMARY KEY (dogid),
 FOREIGN KEY (ownerid) REFERENCES users(userid)
);
```

and add an owners table that looks like this:

```sql
CREATE TABLE users (
 userid integer,
 name varchar,
 age integer,
 PRIMARY KEY (userid)
);
```

The users own dogs. The `ownerid` column in the dogs table corresponds to the `userid` column of the users table (`ownerid` is a foreign key that references the users table).

1. Write a query that lists the names of all the dogs that "Josh Hug" owns.

2. Write the query above using a different kind of join (i.e. if you used an INNER JOIN, try using a cross join with the join condition in the WHERE).

3. Write a query that finds the name of the user and the number of dogs that user owns for the user that owns the most dogs in the database. Assume that there are no ties (i.e. this query should only return 1 user). Users may share the same name.

4. Now write the same query again, but you can no longer assume that there are no ties.

10   Solutions

1)

```
SELECT dogs.name
FROM dogs INNER JOIN users ON dogs.ownerid = users.userid
WHERE users.name="Josh Hug";
```

We now need information from both tables (the dog name is only in the dogs table and the owner name is only in the users table). The join condition is `dogs.ownerid=users.userid` because we only want to get rows with the dog and its owner in it. Finally we add the predicate to the WHERE clause to only get Josh's dogs.

2)

```
SELECT dogs.name
FROM dogs, users
WHERE dogs.ownerid = users.userid and users.name="Josh Hug";
```

We first do a cross join and then add our join condition to the WHERE clause.

3)

```
SELECT users.name, COUNT(*)
FROM users INNER JOIN dogs on users.userid = dogs.ownerid
GROUP BY users.userid, users.name
ORDER BY COUNT(*) DESC
LIMIT 1;
```

Similarly to question 2 in the part 1 notes, we can use an ORDER BY combined with a LIMIT to select the first n most rows (with n being 1 in this case). We GROUP BY the name because we want our groups to be all about one user. We have to include userid in the GROUP BY, because users may share the same name.

4)

```
SELECT users.name, COUNT(*)
FROM users INNER JOIN dogs ON users.userid = dogs.ownerid
GROUP BY users.userid, users.name
HAVING COUNT(*) >= all(
        SELECT COUNT(*)
        FROM dogs
        GROUP BY ownerid
```

)

The inner query gets the number of dogs owned by each owner. The owner(s) with the max number of dogs must have a number of dogs that is `>=` all these rows in order to be the max. We put this condition in the HAVING rather than the WHERE clause because it pertains to the groups not the individual rows.