CS 186
Fall 2020

SQL Part 1 - Basic Queries

1 Database Lingo

Relational databases are made up of **tables** (aka relations). A table has a name (we'll call this one `Person`) and looks like this:

name	age	num_dogs
Ace	20	4
Ada	18	3
Ben	7	2
Cho	27	3

Tables have rows (aka tuples) and columns (aka attributes). In this example, the columns are `name`, `age`, `num_dogs`.

2 Querying a Table

The most fundamental SQL query looks like this:

```
SELECT <columns>
FROM <tbl>;
```

The `FROM` clause tells SQL which table you're interested in, and the `SELECT` clause tells SQL which columns of that table you want to see. For example, consider a table Person(`name`, `age`, `num_dogs`) containing the data below:

name	age	num_dogs
Ace	20	4
Ada	18	3
Ben	7	2
Cho	27	3

If we executed this SQL query:

```
SELECT name, num_dogs
FROM Person;
```

then we could get the following output.

name	num_dogs
Ace	4
Ada	3
Ben	2
Cho	3

In SQL, however, the order of the rows is nondeterministic unless the query contains an `ORDER BY` (we'll get to this later). So the following output is equally valid:

name	num_dogs
Ben	2
Cho	3
Ace	4
Ada	3

In fact, any ordering of those 4 rows is correct – so unless your query contains an `ORDER BY` clause, don't make any assumptions about the order of your results.

3 Filtering out uninteresting rows

Frequently we are interested in only a subset of the data available to us. That is, even though we might have data about many people or things, we often only want to see the data that we have about very specific people or things. This is where the `WHERE` clause comes in handy; it lets us specify which specific rows of our table we're interested in. Here's the syntax:

```
SELECT <columns>
FROM <tbl>
WHERE <predicate>;
```

Once again, let's consider our table `Person(name, age, num_dogs)`. Suppose we want to see how many dogs each person owns — same as before — but this time we only care about the dog-owners who are adults. Let's walk through this SQL query:

```
SELECT name, num_dogs
FROM Person
WHERE age >= 18;
```

When reasoning about query execution you can use the following rule: each clause in a SQL query happens in the order it's written, except for `SELECT` which happens last. This is not necessarily the order the database actually does these operations (more on this later in the semester), but it is easy to think about and will always give us the correct answer. The `FROM` clause tells us we're interested in the `Person` table, so this is the table we start with:

name	age	num_dogs
Ace	20	4
Ada	18	3
Ben	7	2
Cho	27	3

Next we move on to the `WHERE` clause. It tells us that we only want to keep the rows satisfying the predicate `age >= 18`, so we remove the row with Ben, and are left with the following table:

name	age	num_dogs
Ace	20	4
Ada	18	3
Cho	27	3

And finally, we `SELECT` the columns `name` and `num_dogs` to obtain our final result. (Again, any permutation of this result is equally valid so you shouldn't make any assumptions about the order of the rows.) This gives us our final table:

name	num_dogs
Ace	4
Ada	3
Cho	3

4 Boolean operators

If you want to filter on more complicated predicates, you can use the boolean operators NOT, AND, and OR. For instance, if we only cared about dog-owners who are not only adults, but also own more than 3 dogs, then we would write the following query:

```
SELECT name, num_dogs
FROM Person
WHERE age >= 18
  AND num_dogs > 3;
```

As in Python, this is the order of evaluation for boolean operators:

1. `NOT`
2. `AND`
3. `OR`

That said, it is good practice to avoid ambiguity by adding parentheses even when they are not strictly necessary.

5 Filtering Null Values

In SQL, there is a special value called `NULL`, which can be used as a value for any data type, and represents an "unknown" or "missing" value.

Bear in mind that some values in your database may be `NULL` whether you like it or not, so it's good to know how SQL handles them. It pretty much boils down to the following rules:

- If you do anything with `NULL`, you'll just get `NULL`. For instance if x is `NULL`, then `x > 3`, `1 = x`, and `x + 4` all evaluate to `NULL`. Even `x = NULL` would evaluate to `NULL`; if you want to check whether x is `NULL`, then write `x IS NULL` or `x IS NOT NULL` instead.
- `NULL` is falsey, meaning that `WHERE NULL` is just like `WHERE FALSE`. The row in question does not get included.
- `NULL` short-circuits with boolean operators. That means a boolean expression involving NULL will evaluate to:
  - `TRUE`, if it'd evaluate to `TRUE` regardless of whether the `NULL` value is really `TRUE` or `FALSE`.
  - `FALSE`, if it'd evaluate to `FALSE` regardless of whether the `NULL` value is really `TRUE` or `FALSE`.
  - Or `NULL`, if it depends on the `NULL` value.

Let's walk through this query as an example:

```
SELECT name, num_dogs
FROM Person
WHERE age <= 20
   OR num_dogs = 3;
```

Let's assume we change some values to `NULL`, so after evaluating the `FROM` clause we are left with:

name	age	num_dogs
Ace	20	4
Ada	NULL	3
Ben	NULL	NULL
Cho	27	NULL

Next we move on to the `WHERE` clause. It tells us that we only want to keep the rows satisfying the predicate `age <= 20` OR `num_dogs = 3`. Let's consider each row one at a time:

- For Ace, `age <= 20` evaluates to `TRUE` so the claim is satisfied.
- For Ada,`age <= 20` evaluates to `NULL` but `num_dogs = 3` evaluates to `TRUE` so the claim is satisfied.
- For Ben, `age <= 20` evaluates to `NULL` and `num_dogs = 3` evaluates to `NULL` so the overall expression is `NULL` which has a falsey value.
- For Cho, `age <= 20` evaluates to `FALSE` and `num_dogs = 3` evaluates to `NULL` so the overall expression evaluates to `NULL` (because it depends on the value of the `NULL`). Because NULL is falsey, this row will be excluded.

Thus we keep only Ace and Ada.

SQL Part 1 - Basic Queries

6 Grouping and Aggregation

When you're working with a very large database, it is useful to be able to summarize your data so that you can better understand the general trends at play. Let's see how.

6.1 Summarizing columns of data

With SQL you are able to summarize entire columns of data using built-in aggregate functions. The most common ones are `SUM`, `AVG`, `MAX`, `MIN`, and `COUNT`. Here are some important characteristics of aggregate functions:

- The input to an aggregate function is the name of a column, and the output is a single value that summarizes all the data within that column.
- Every aggregate ignores NULL values except for `COUNT(*)`. (So `COUNT(<column>)` returns the number of non-NULL values in the specified column, whereas `COUNT(*)` returns the number of rows in the table overall.)

For example, consider this variant of our table `People(name, age, num_dogs)` from earlier, where we are now unsure how many dogs Ben owns:

name	age	num_dogs
Ace	20	4
Ada	18	3
Ben	7	NULL
Cho	27	3

With this table in mind ...

- `SUM(age)` is 72.0, and `SUM(num_dogs)` is 10.0.
- `AVG(age)` is 18.0, and `AVG(num_dogs)` is 3.3333333333333333.
- `MAX(age)` is 27, and `MAX(num_dogs)` is 4.
- `MIN(age)` is 7, and `MIN(num_dogs)` is 3.
- `COUNT(age)` is 4, `COUNT(num_dogs)` is 3, and `COUNT(*)` is 4.

So, if we desired the range of ages represented in our database, then we could use the query below and it would produce the result 20. (Technically it would produce a one-by-one table containing the number 20, but SQL treats it the same as the number 20 itself.)

```
SELECT MAX(age) - MIN(age)
FROM Person;
```

Or, if we desired the average number of dogs owned by adults, then we could write this:

```
SELECT AVG(num_dogs)
FROM Person
WHERE age >= 18;
```

6.2 Summarizing Groups of Data

Now you know how to summarize an entire column of your database into a single number. More often than not, though, we want a little finer granularity than that. This is possible with the `GROUP BY` clause, which allows us to split our data into groups and then summarize each group separately. Here's the syntax:

```
SELECT <columns>
FROM <tbl>
WHERE <predicate>   -- Filter out rows (before grouping).
GROUP BY <columns>
HAVING <predicate>; -- Filter out groups (after grouping).
```

Notice we also have a brand new `HAVING` clause, which is actually very similar to `WHERE`. The difference?

- `WHERE` occurs *before* grouping. It filters out uninteresting *rows*.
- `HAVING` occurs *after* grouping. It filters out uninteresting *groups*.

To explore all these new mechanics let's see another step-by-step example. This time our query will find the average number of dogs owned, for each adult age represented in our database. We will exclude any age for which we only have one datum.

```
SELECT age, AVG(num_dogs)
FROM Person
WHERE age >= 18
GROUP BY age
HAVING COUNT(*) > 1;
```

Let us assume the `Person` table is now:

name	age	num_dogs
Ace	20	4
Ada	18	3
Ben	7	2
Cho	27	3
Ema	20	2
Ian	20	3
Jay	18	5
Mae	33	8
Rex	27	1

Next we move on to the `WHERE` clause. It tells us that we only want to keep the rows satisfying the predicate `age >= 18`, so we remove the row with Ben.

name	age	num_dogs
Ace	20	4
Ada	18	3
Cho	27	3
Ema	20	2
Ian	20	3
Jay	18	5
Mae	33	8
Rex	27	1

Now for the interesting part. We arrive at the `GROUP BY` clause, which tells us to categorize the data by age. We end up with a group of all the adults 20 years old, a group of all the adults 18 years old, a group of all the adults 27 years old, and a group of all the adults 33 years old. You can now think of the table like this:

name	age	num_dogs
Ace	20	4
Ema	20	2
Ian	20	3
Ada	18	3
Jay	18	5
Cho	27	3
Rex	27	1
Mae	33	8

The `HAVING` clause tells us we only want to keep the groups satisfying the predicate `COUNT(*) > 1` — that is, the groups that contain more than one row. We discard the group that contains only Mae.

name	age	num_dogs
Ace	20	4
Ema	20	2
Ian	20	3
Ada	18	3
Jay	18	5
Cho	27	3
Rex	27	1

Last but not least, every group gets collapsed into a single row. According to our `SELECT` clause, each such row must contain two things:

- The `age` corresponding to the group.
- The `AVG(num_dogs)` for the group.

Our final result looks like this:

age	AVG(num_dogs)
20	3.0
18	4.0
27	2.0

So, to recap, here's how you should go about a query that follows the template above:

- Start with the table specified in the `FROM` clause.
- Filter out uninteresting rows, keeping only the ones that satisfy the `WHERE` clause.
- Put data into groups, according to the `GROUP BY` clause.
- Filter out uninteresting groups, keeping only the ones that satisfy the `HAVING` clause.
- Collapse each group into a single row, containing the fields specified in the `SELECT` clause.

7 A Word of Caution

So that's how grouping and aggregation work, but before we move on I must emphasize one last thing regarding illegal queries. We'll start by considering these two examples:

1. Though it's not immediately obvious, this query actually produces an error:

```
SELECT age, AVG(num_dogs)
FROM Person;
```

What's the issue? `age` is an entire column of numbers, whereas `AVG(num_dogs)` is just a single number. This is problematic because a properly formed table must have the same amount of rows in each column.

2. This query does not work either, for a very similar reason:

```
SELECT age, num_dogs
FROM Person
GROUP BY age;
```

After grouping by age we obtain a table like this:

name	age	num_dogs
Ace	20	4
Ema	20	2
Ian	20	3
Ada	18	3
Jay	18	5
Cho	27	3
Rex	27	1
Mae	33	8

Then the `SELECT` clause's job is to collapse each group into a single row. Each such row must contain two things:

- The `age` corresponding to the group, which is a single number.
- The `num_dogs` for the group, which is an entire column of numbers.

The takeaway from all this? If you're going to do any grouping / aggregation at all, then you must only `SELECT` grouped / aggregated columns.

8 Order By

Before, we mentioned that the ordering of the output rows was usually nondeterministic in SQL. If you want the rows of your table to appear in a certain order you must use an `ORDER BY` clause.

Here is an example query using an `ORDER BY` clause:

```
SELECT name, num_dogs
FROM Person
ORDER BY num_dogs, name;
```

You can include as many columns as you want in the ORDER BY clause. We first sort on the first column listed, and then break any ties with the second column listed, and then break any remaining ties with the third column listed, and so on. By default, the sort order is ascending, but if we want the order in descending order we add the DESC keyword after the column name. If we wanted to sort by the num_dogs ascending and break ties by name descending, we would use the following query:

```
SELECT name, num_dogs
FROM Person
ORDER BY num_dogs, name DESC;
```

SQL Part 1 - Basic Queries

9 Limit

Sometimes we only want to see a few rows in our table, even if more rows match all of our other conditions. To do this we can add a `LIMIT` clause to the end of our query to cap the number of rows that will be returned. Note: the same rows may not always be returned by queries using `LIMIT` if an `ORDER BY` is not used or if there are ties in the ordering. Here is a query that only returns one row:

```
SELECT name, num_dogs
FROM Person
LIMIT 1;
```

10 Conclusion

Congratulations - we have covered a lot of SQL! To help remember the ordering of the SQL stages, here is the syntax of a query involving the expressions we have learned so far:

```
SELECT <columns>
FROM <tbl>
WHERE <predicate>
GROUP BY <columns>
HAVING <predicate>
ORDER BY <columns>
LIMIT <num>;
```

11 Practice Questions

We have a dogs table that looks like this:

```
CREATE TABLE dogs (
dogid integer,
ownerid integer,
name varchar,
breed varchar,
age integer
)
```

1. Write a query that finds the name of every dog that has an `ownerid=3`.
2. Write a query that lists the 5 oldest dogs' name and age. Break ties by the dog's name in ascending alphabetical order.

3. Write a query that shows how many dogs we have for each breed, but only for breeds that have multiple dogs.

12 Solutions

1.
```
SELECT name
FROM dogs
WHERE ownerid = 3;
```

We're only looking for the name, so that is the only column that goes in the SELECT clause. We are selecting it from the dogs table so the dogs table goes in the FROM clause. We add the only predicate (`ownerid=3`) to the WHERE clause.

2.
```
SELECT name, age
FROM dogs
ORDER BY age DESC, name
LIMIT 5;
```

To find the 5 oldest dogs we will order them by their age (in descending order) and then limit the number of rows returned by our query to 5. The final piece is we need to add a second column to the ORDER BY clause to break ties.

3.
```
SELECT breed, COUNT(*)
FROM dogs
GROUP BY breed
HAVING COUNT(*) > 1;
```

To get statistics for the breeds, we will GROUP BY the breeds column. To only select the groups that have multiple dogs, we add the `COUNT(*) > 1` predicate to the having clause.

Thanks to Sequioa who we have taken a large portion of this note from: https://sequoia-tree.github.io/relational-databases/sql-dml.html