CSCI 403: Databases
3 - Basic SQL Retrieval Queries

SELECT

The most basic retrieval query looks like

```
SELECT * FROM <tablename>;
```

or

SELECT a_1, a_2, ... FROM <tablename>;

- a_i is an attribute of the relation (a column of the table)
- * stands for all attributes

This query selects all rows. By using the second form of the query, we choose only the attributes we are interested in, in order, from the relation (table). This is called a "projection".

The result of a query is a *relation*. It is anonymous (has no name), but otherwise is like any other relation; the attributes for the result relation are the attributes selected in the query (and have the same name and order). This has certain implications that we'll examine further later in the lecture.

WHERE

Typically want only *some* rows:

```
SELECT <attributes>
FROM <tablename>
WHERE <condition>;
```

The <condition> is a Boolean expression on the attributes or functions of the attributes of the table.

E.g.,

```
SELECT course_id FROM mines_courses
WHERE instructor = 'Painter-Wakefield,
Christopher';
```

gets the course_ids of all courses taught by me.

There are many operators and functions applicable to the WHERE condition. E.g., to get all courses *not* taught by me, use the "not equals" operator, which is <>, e.g., `SELECT ... WHERE instructor <> 'Painter-Wakefield, Christopher';`

You have access to all the other usual relational operators: <, >, <=, >=. There is also a shortcut operator, BETWEEN, which tests for (inclusive) containment in some interval. E.g.,

```
SELECT * FROM mines_courses
WHERE enrollment BETWEEN 20 AND 45;.
```

This is equivalent to

```
SELECT * FROM mines_courses
WHERE 20 <= enrollment
AND 45 >= enrollment;
```

Combine expressions into compound expressions using AND and OR; AND has precedence over OR (or can use parentheses to force order of evaluation). NOT can be used to invert any Boolean expression.

Recall that NULL values are not comparable - you can not, for instance, test to see if an attribute is NULL by comparing using =. Instead, there is a special operator, IS NULL, which tests for NULL values:

```
SELECT * FROM mines_courses
WHERE instructor IS NULL;.
```

To test for non-NULL values, use IS NOT NULL.

A useful operator on string attributes is LIKE, which lets you do wildcard comparisons. To get all courses taught by me or Dr. Mehta, for instance:

```
SELECT course_id FROM mines_courses
WHERE instructor LIKE 'Painter%' OR
instructor LIKE 'Mehta%';
```

In wildcard expressions, % stands in for zero or more characters, so '%foo%' would find all text containing the string "foo", whereas 'bar%' only finds strings that start with "bar". Another wildcard is the underscore, _, which matches a single character. This forces there to be exactly one character in that position, but it can be any character. E.g., `SELECT ... WHERE course_id LIKE '____4__';` would get all rows for 400-level courses in any subject (e.g., CSCI403, LAIS404, CEEN443, etc.)

Operators in Other Clauses

In addition to the relational operators discussed above, you have access to the normal mathematical operators, which can be used in expressions in most any clause. The relational operators above (evaluating to true/false) can also be used. For instance, you can SELECT the result of a mathematical operation on some attributes or constants; you can even use the database query engine as a calculator, if you want:

```
SELECT 16 * 22 - 17;
```

See the PostgreSQL documentation for all the operators available.

Functions

Many functions available, including mathematical, string, type conversion, etc. See function reference in PostgreSQL documents for examples (not all functions available in PostgreSQL are standard SQL.) An example of a useful function is substring():

```
SELECT substring(course_id from 1 for
4) AS subject FROM mines_courses WHERE
...;
```

This query gets just the 4-letter subjects (e.g., CSCI, LAIS, CEEN) from the mines_courses table.

Names and Aliasing

Note above the use of the keyword AS. This keyword lets us *rename* attributes in our SELECT query for the output relation. In the example above, the application of the substring() function results in a not very informative attribute name (substring) for the output relation, so I renamed it to be "subject". This affects e.g., output column names in your query tool, names of columns (entries in a dictionary, e.g.) when querying from a programming language, and so forth. It will also come in handy in some situations when we join two or more tables together (to avoid name collisions) and in ORDER BY clauses and GROUP BY clauses.

Name collisions when joining (see below) can also be resolved by specifying the table name together with the column:

```
SELECT mines_courses.course_id
FROM mines_courses;
```

We can also *alias* tables and use the aliases to resolve columns in a join query (very useful!) For example, the above query could be rewritten as

```
SELECT mc.course_id
FROM mines_courses AS mc;
```

The AS keyword is optional, so this is equivalent to

```
SELECT mc.course_id
FROM mines_courses mc;
```

Joins

A SELECT query can specify more than one table as the source of data:

```
SELECT table1.a_1, table1.a_2, ...,
table2.a_1, ...
FROM table1, table2, ...
WHERE ...;
```

With no WHERE conditions, this gives the *cross product* of the two tables; the resulting relation's tuples are concatenations of each tuple from table1 with each tuple from table2.

E.g.,

```
SELECT mc.instructor, mcf.name FROM
mines_courses AS mc, mines_cs_faculty
AS mcf;
```

results in tuples like ('Painter-Wakefield, Christopher', 'Painter-Wakefield, Christopher') and ('Painter-Wakefield, Christopher', 'Mehta, Dinesh') and ('Mehta, Dinesh', 'Fisher, Wendy') etc.

So this is typically not what is desired. There are two main mechanisms for specifying *join conditions* to determine which rows from table 1 go with which rows from table 2 (and so forth). We'll talk about JOIN clauses later, for now we'll focus on what we can do in the WHERE clause.

Thinking conceptually of the result relation from the above query as a cross-product result, we can apply the WHERE clause to filter out the nonsense rows and retain the ones that actually make sense:

```
SELECT * FROM mines_courses AS mc,
mines_cs_faculty AS mcf
WHERE mc.instructor = mcf.name;
```

We can join on multiple conditions, or add additional conditions as usual. For instance, if we

crn	course_id	section	instructor	title
10120	CSCI262	B	Painter-Wakefield, Christopher	DATA STRUCTURES
12693	CSCI262	R02	Painter-Wakefield, Christopher	DATA STRUCTURES
12048	CSCI403	A	Painter-Wakefield, Christopher	DATABASE MANAGEMENT
12623	CSCI406	A	Mehta, Dinesh	ALGORITHMS
10300	CSCI406	A	Mehta, Dinesh	ALGORITHMS
12621	CSCI303	A	Fisher, Wendy	DATA SCIENCE

Figure 1: Some rows from the mines_courses table

name	office	email
Painter-Wakefield, Christopher	BB 280I	cpainter@mines.edu
Mehta, Dinesh	BB 280J	dmehta@mines.edu
Rader, Cynthia		crader@mines.edu
Fisher, Wendy	BB 280D	wfisher@mines.edu

Figure 2: Some rows from the mines_cs_faculty table

crn	days	building	room	begin_time	end_time
10120	MW	CO	209	13:00:00	13:50:00
12623	MWF	BB	W210	14:00:00	14:50:00
10300	MWF	MZ	235	12:00:00	12:50:00
12693	F	MZ	022	13:00:00	13:50:00
12621	TR	BB	W210	08:00:00	09:15:00
12048	MWF	BE	108	11:00:00	11:50:00

Figure 3: Some rows from the mines_course_meetings table. The data in this table is separate from the mines_courses table because some sections of some courses can have multiple entries due to different meeting times/locations on different days (e.g., a lab section in a computer classroom).

provided instructor names as two fields (last name and first name) instead of one, we would need two join conditions ANDed together.

Here's a query to try:

```
SELECT course_id, instructor, office,
email
FROM mines_courses, mines_cs_faculty
WHERE instructor = name;
```

Note that this query uses no aliases or fully specified column names; as long as this is unambiguous, the SQL command processor will accept it. However, if instead of "name", we had used the attribute name "instructor" in the mines_eecs_faculty table, then it would make no sense to have a join with the condition `WHERE instructor = instructor`. In that case we must fully specify the column names (either using the table names or aliases for the tables). In general, even where unambiguous, table aliases and fully specified column names are often preferable for making the query more readable and understandable.

We can join more than two tables. The mines_courses_meetings table contains info about the days/times/locations where course sections are held. This table can be joined to the mines_courses table through the common attribute "crn". Here's a query to try:

```
SELECT mc.instructor, mc.course_id,
mcm.days, mcm.building, mcm.room,
mcm.begin_time,
mcf.office, mcf.email
FROM mines_courses mc,
mines_cs_faculty mcf,
mines_courses_meetings mcm
WHERE mc.crn = mcm.crn
AND mcf.name = mc.instructor;
```

Notes on joins

N.B., in a join, missing entries or NULLs in attributes in the join condition mean that rows are simply excluded, since NULL never equals anything (even NULL). For instance, in joining mines_courses with mines_cs_faculty, you will only get back information for courses that are taught by CS faculty (since that is all that is in the table), and only for courses whose instructor was known when the data was created (NULL instructors cannot match any name). Later we'll learn about outer joins, which will let us include rows that do not match in a join.

Also note that, even though we describe joins conceptually as forming a cross product which we then filter down with our join conditions, in actual practice this is not what happens - it would be way too expensive to make a cross product with a billion rows only to select seven of them! When we study relational algebra we'll see how queries can be reordered to make for much more efficient join queries. (Unless the user actually *wants* a cross-product result, of course.)

DISTINCT and ORDER BY

Since the result of a SELECT query is a relation, which is (in the relational model) a *set* of tuples, we might reasonably expect our queries to return no duplicate rows. However, relations in a relational DBMS actually do not adhere to this particular part of the model; duplicate rows are acceptable in tables, and duplicate rows in results are also normal. E.g., if we do
`SELECT instructor FROM mines_courses;`,
my name will pop up three times (one for each course section I teach).

To eliminate duplicate tuples, use the DISTINCT keyword:

```
SELECT DISTINCT instructor
FROM mines_courses;
```

On the other hand, relational DBMSes do follow the relational model in that rows have no intrinsic ordering. A query can provide results in any order (even in different orders for the same query done twice!). To set the order, use an ORDER BY clause, where ORDER BY is followed by the columns on which to sort:

```
SELECT firstname, lastname, otherstuff
FROM sometable
ORDER BY lastname, firstname;
```

This will sort alphabetically (assuming text attributes) by lastname and then firstname. If we want to reverse the order of sorting for any attribute, we can include the DESC keyword:

```
SELECT earnings, salesperson
FROM sales
ORDER BY earnings DESC;
```