CSCI 403: Databases
8 - Miscellaneous SELECT queries, More DDL

Join Clauses

Inner Joins

Joins are conveniently constructed using the WHERE clause of a SELECT query; however, another approach is available, which is to use a JOIN clause.

Example:

```
SELECT course_id, instructor, office
FROM mines_courses JOIN
mines_cs_faculty ON instructor = name;
```

is equivalent to

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

This most common join is formally known as an "inner join". Thus, the above query can also be expressed as

```
SELECT course_id, instructor, office
FROM mines_courses INNER JOIN
mines_cs_faculty ON instructor = name;
```

Outer Joins

As we discussed when we looked at joins the first time, an inner join will only return rows that match in the two joined tables; in the example above, we only get rows from mines_courses with matching instructor names in mines_cs_faculty; similarly, we only get rows from mines_cs_faculty with matching names in mines_courses (in the latter case, it happens to be all rows).

Sometimes it is desirable to see all rows from one table, joining in data from another table *where available*. The technique is called an outer join, and comes in three flavors: left, right, and full.

A left outer join uses the same syntax as above, but with the keywords LEFT OUTER replacing INNER[1]. In the case of a left outer join, all rows in the table on the left hand side of the JOIN clause will be returned; where data is available in the table on the right hand side, it too will be returned, with NULLs filling in when data is not available.

E.g.,

```
SELECT course_id, instructor, office
FROM mines_courses LEFT OUTER JOIN
mines_cs_faculty ON instructor = name;
```

This shows all rows from mines_courses, with office info for CS faculty only (and NULLs for everyone else). If we instead do

```
SELECT course_id, name, office
FROM mines_courses RIGHT OUTER JOIN
mines_cs_faculty ON instructor = name;
```

we get a right outer join, and thus all rows from mines_cs_faculty together with any matching info from mines_courses. Note that this does *not* select all rows from mines_courses!

If we want to select all rows from both tables, we can use a full outer join. As you might expect, the keyword FULL replaces RIGHT in the above query. Now we get all rows from mines_courses and all rows from mines_cs_faculty, with the matching tuples showing data from both tables as usual.

Natural Joins (Equijoins)

Another join type lets us shorten the join clause a bit in the case when the join conditions simply equate all attributes sharing the same name in both tables. For example, mines_courses and mines_courses_meetings are only usefully joined by equating the crn attribute in both tables; since this is the only attribute which appears in both tables, we can do a natural join:

[1] In Oracle, there is a special operator, (+), which can be used in a WHERE clause join to effect outer joins. Watch out for this if you work with Oracle.

```
SELECT * FROM mines_courses NATURAL
JOIN mines_courses_meetings
WHERE instructor = ‘Painter-Wakefield,
Christopher’;
```

Set operations

Since we can view a relation as a set of tuples with the same schema, it is natural to expect that we can apply set operations to relations, and so we can. Specifically, SQL provides the operators UNION, INTERSECTION, and EXCEPT. These provide set union, intersection, and difference, and can be applied to the results of two or more SELECT queries.

For a trivial example, a UNION can replace an OR in a WHERE clause (this is not a recommended use, I provide it only for example):
SELECT * FROM mines_courses WHERE course_id LIKE 'CSCI%'
UNION
SELECT * from mines_courses WHERE course_id LIKE 'LAIS%';

Note that the two relations *must* have compatible schemas, that is, all the types must match. The *names* of the resulting relation are taken from the first query in the union; names in the second (and third, fourth, ...) queries do not have to match (just the types of the columns must match).

Note that all of the set operators will return a true set, not a multiset, unless you add the keyword ALL after the set operator. That is, query1 UNION query2 will result in a relation with all duplicate tuples removed, whereas query1 UNION ALL query2 retains any duplicates.

Other SELECT voodoo

There are many more techniques we cannot cover in this course, such as WITH queries, recursive queries, etc. Some of these will be covered in the textbook, so you can at least get some picture of their use.

More DDL

ALTER TABLE

ALTER TABLE lets us modify tables that have already been created. Here are a few of the more common uses:

Adding a primary key:
```
ALTER TABLE tablename ADD PRIMARY KEY
(attr1, attr2, ...);
```
Adding a foreign key:
```
ALTER TABLE table1
ADD FOREIGN KEY (attr1, attr2, ...)
REFERENCES table2 (attr1, attr2, ...);
```
Adding a column:
```
ALTER TABLE tablename
ADD COLUMN columnname type;
```

DROP

`DROP objecttype objectname;` lets us drop tables, views, constraints, etc.

Views

A view is like a saved query given a name; you can select from it just like any other relation, but underlying it is a SELECT query, not a simple table. So views can be as complex as desired (with joins, unions, whatever).

Syntax is identical to CREATE TABLE AS, just VIEW instead of TABLE:
```
CREATE VIEW viewname AS SELECT ...;
```
You can treat the view thus created as any other table for purposes of selection; you cannot perform insert, update, or delete queries on views, although you can achieve a similar behavior using triggers (a future topic). You also cannot add indices to the view - it will use the underlying table indices instead.

Sequences

A *sequence* is a database object which generates integer sequences. These are created for you, for instance, when you create a table with a column of type `serial`. To create a sequence manually, use the `CREATE SEQUENCE ...;` DDL command. Sequences can be created which are ascending, descending, starting at specific values, allowing cycling, etc. The basic sequence starts at 1, ascends by 1 on each use, and does not allow cycling, e.g:

```
CREATE SEQUENCE my_sequence;
```

To use and manipulate sequences, you use a set of sequence functions. The most often used function is `nextval`, which, as its name suggests, gets the next value from the sequence and increments the sequence. For example,

```
SELECT nextval('my_sequence');
```

will return a 1 on the first usage, 2 on the next usage, etc. Note that you have to pass in the sequence name as a string literal!

Other functions include `currval`, which returns the value most recently obtained using `nextval` for the specified sequence, and `setval`, which lets you change the state of the sequence (essentially dictating what `nextval` and `currval` will return on their next invocation.)