CS 325 - Reading Packet: "More options for the SQL `select` statement's `where` clause, column aliases, table aliases, computed columns, aggregate functions, and more"

SOURCES:

- "Oracle9i Programming: A Primer," Rajshekhar Sunderraman, Addison Wesley.
- Classic Oracle example tables `empl` and `dept`, adapted somewhat over the years

Combining relational operations within a single SQL `select` statement

Recall the SQL `select` statement semantics (for simple SQL selects) presented in the previous lab:

1. perform a Cartesian product of the tables listed in the `from` clause;
2. perform a relational selection of the rows from 1. that meet the condition in the `where` clause;
3. perform a "partial" projection from 2 (which is only guaranteed to be a "pure" projection if `DISTINCT` is included!) of the expressions/columns in the `select` clause.

Now that we (last lab) have discussed how the SQL `select` statement can be used to specify "pure" relational operations, we should be ready, keeping these SQL `select` semantics in mind, to combine relational operations within a single SQL `select` statement.

For example, we often don't do a "pure" natural join or equi-join -- more often, we simply project the desired columns from an equi-join. For example, if we only want to know each employee's last name, department name, and department location, we would only project those three columns from the equi-join of the `empl` and `dept` tables:

```
select  empl_last_name, dept_name, dept_loc
from    empl, dept
where   empl.dept_num = dept.dept_num;
```

For tables `empl` and `dept` with contents as inserted in SQL script `set-up-ex-tbls.sql`, this query has the results:

```
EMPL_LAST_NAME  DEPT_NAME       DEPT_LOC
--------------- --------------- ---------------
King            Management      New York
Jones           Research        Dallas
Blake           Sales           Chicago
Raimi           Accounting      New York
Ford            Research        Dallas
```

```
Smith           Research        Dallas
Michaels        Sales           Chicago
Ward            Sales           Chicago
Martin          Sales           Chicago
Scott           Research        Dallas
Turner          Sales           Chicago

EMPL_LAST_NAME  DEPT_NAME       DEPT_LOC
--------------- --------------- ---------------
Adams           Operations      Boston
James           Sales           Chicago
Miller          Accounting      New York

14 rows selected.
```

As another example, it is very common to combine projection and selection -- that is, one might select specific rows from some table, and then only project particular columns from that selection of rows. For example, if I am just interested in the last names and salaries of employees who are managers, then I could combine projection and selection within a single SQL `select` statement as so:

```
select empl_last_name, salary
from   empl
where  job_title = 'Manager';
```

(Keep in mind: according to our SQL `select` semantics, we are grabbing all of the rows from the `empl` table (since there is only one, there isn't really a Cartesian product), then selecting just those `empl` rows for which `job_title` is equal to `'Manager'`, and then projecting the `empl_last_name` and `salary` from **just** those rows.)

This query (for this table with its contents as inserted by `set-up-ex-tbls.sql`) has the results:

```
EMPL_LAST_NAME     SALARY
--------------- ----------
Jones                 2975
Blake                 2850
Raimi                 2450
```

And, as another example, if you only want `job_titles` and `hiredates` for employees with `commissions` greater than 0, you could do that with a combination of selection and projection as well:

```
select  job_title, hiredate
from    empl
where   commission > 0;
```

This query has the results:

```
JOB_TITLE  HIREDATE
---------- ---------
```

```
Salesman   20-FEB-91
Salesman   22-FEB-91
Salesman   28-SEP-91
```

SQL "gotcha" -- selecting rows in which a given column is `NULL`

Here is a SQL "gotcha" that you need to know: if you just want to select rows in which a particular column is null (or not null), then you have to ask that in a very particular way (and not in the way any sensible person would think would work!): you need to use `is null` or `is not null`.

So, if you would like the last names of employees who have **no** commission (a `commission` column value of `null`, empty), then you would write:

```
select empl_last_name
from   empl
where  commission is null;
```

And this query has the expected results:

```
EMPL_LAST_NAME
---------------
King
Jones
Blake
Raimi
Ford
Smith
Scott
Adams
James
Miller

10 rows selected.
```

Here's the sad part: assume you, quite sensibly, used = instead of `is` in this query:

```
select empl_last_name
from   empl
where  commission = null;
```

This would not be an error -- however, it would not give the same results, either! (Try it!) You'll find that this query simply gives the result:

```
no rows selected
```

Likewise, if you want to project just the salaries of employees who have non-null commissions, this would give you the results you want:

```
select salary
```

```
from   empl
where  commission is not null;
```

This query has the results:

```
    SALARY
----------
      1600
      1250
      1250
      1500
```

...while this would result in no rows:

```
select salary
from   empl
where  commission != null;
```

The moral of this particular story is to try to remember to use `is` instead of = whenever you want to select rows based on a column being or not being `null`.

More examples of combining relational operations within a single SQL `select` statement

If you want to have further selection of rows from an equi-join, you will typically indicate this within a SQL `select` statement by using logical `AND`s within the `where` clause. (That is, you will only select rows for which the join-condition is true `AND` additional criteria are met.)

So, if you would like employee last names, the names of their departments, and their department locations only for employees hired since 12-1-1991, then you can use a combination of selection, equi-join, and projection as follows:

```
select  empl_last_name, dept_name, dept_loc
from    empl, dept
where   empl.dept_num = dept.dept_num
        and hiredate > '01-dec-1991';
```

Here, we are selecting those rows from the Cartesian product of `empl` and `dept` for which `empl.dept_num = dept.dept_num` and `hiredate > '01-dec-1991'`.

This query has the results:

```
EMPL_LAST_NAME  DEPT_NAME       DEPT_LOC
--------------- --------------- ---------------
Ford            Research        Dallas
James           Sales           Chicago
Miller          Accounting      New York
```

Such combinations of relational operations are very common -- they are extremely useful and versatile.

When table names are REQUIRED before a column name

Consider the preceding example -- what if I decided that I'd like to project the department number, also, right before the department name, for each employee hired after December 1, 1991? I might try this:

```
/* this WON'T WORK: */

select  empl_last_name, dept_num, dept_name, dept_loc
from    empl, dept
where   empl.dept_num = dept.dept_num
        and     hiredate > '01-dec-1991';
```

...but you would find that the above does not work, and indeed gives an error that includes the phrase "column ambiguously defined":

```
ERROR at line 1:
ORA-00918: column ambiguously defined
```

What could be ambiguous here? Consider again what a Cartesian product does: it includes all pairings of all columns from both tables. So, the Cartesian product of empl and dept has two columns with the name dept_num. To indicate which table's dept_num is intended, you precede it by the specific table name, followed by a dot -- and, indeed, this is what we have been doing in our join conditions:

```
empl.dept_num = dept.dept_num
```

It just happens that you need to use this table_name.col_name notation anywhere within the SQL select that you use a column name that appears in more than one table in the from clause (even in the select clause!).

So, since dept_num appears in both empl and dept, we can project dept_num by using either:

```
/* ... this DOES work: */

select  empl_last_name, dept.dept_num, dept_name, dept_loc
from    empl, dept
where   empl.dept_num = dept.dept_num
and     hiredate > '01-dec-1991';

/* as does: */

select  empl_last_name, empl.dept_num, dept_name, dept_loc
from    empl, dept
where   empl.dept_num = dept.dept_num
and     hiredate > '01-dec-1991';
```

Do you see that, since you are only selecting rows for which empl.dept_num = dept.dept_num, both of these must give the same results? And these identical results are:

```
EMPL_LAST_NAME DEP DEPT_NAME      DEPT_LOC
```

```
-------------- --- -------------- --------------
Ford           200 Research       Dallas
James          300 Sales          Chicago
Miller         100 Accounting     New York
```

More possibilities for the `where` clause -- `AND`, `OR`, `NOT`, `!=`, `<>`

Before we go on, let's expand the possibilities for specifying which rows we would like to select, since Oracle SQL provides a nicely-rich set of options for this.

We've already mentioned that SQL provides the boolean `AND` operation, that is true only if both operands are **true**. So, note that SQL also provides the boolean `OR` operation, that is true if **either** operand is true, as well as the boolean `NOT` operation, true if its operand is false.

For example, what if you would like to see the last names of employees who are **either** sales people **or** have a salary of $1500 or more? Then you could use `OR` for this:

```
select  empl_last_name
from    empl
where   job_title = 'Salesman'
        or salary >= 1500;
```

This query has the results:

```
EMPL_LAST_NAME
---------------
King
Jones
Blake
Raimi
Ford
Michaels
Ward
Martin
Scott
Turner

10 rows selected.
```

Be careful when you combine `AND` and `OR` within the same SQL `select` statement -- to make it perfectly clear what is being `AND`-ed and what is being `OR`-ed, you should use parentheses to make that explicitly clear. For example, if I want the names and hiredates of only employees hired after March 1, 1991, who are also either sales people or make $1500 or more, then this would accomplish this (and be clear to the reader):

```
select empl_last_name, hiredate
from   empl
where  hiredate > '01-Mar-1991'
```

```
       and (job_title = 'Salesman'
            or salary >= 1500);
```

This query has the results:

```
EMPL_LAST_NAME  HIREDATE
--------------- ---------
King            17-NOV-91
Jones           02-APR-91
Blake           01-MAY-91
Raimi           09-JUN-91
Ford            03-DEC-91
Martin          28-SEP-91
Scott           09-NOV-91
Turner          08-SEP-91

8 rows selected.
```

As an example of the logical NOT operator, consider one of the several ways you can select those employee rows for employees who are not sales people:

```
select  *
from    empl
where   not job_title = 'Salesman';
```

Interestingly, though, SQL has two different "not equal" operators, both <> and != :

```
select  *
from    empl
where   job_title <> 'Salesman';

select  *
from    empl
where   job_title != 'Salesman';
```

All three of these queries have the same results:

```
EMPL EMPL_LAST_NAME  JOB_TITLE  MGR  HIREDATE      SALARY COMMISSION DEP
---- --------------- ---------- ---- --------- ---------- ---------- ---
7839 King            President       17-NOV-91       5000            500
7566 Jones           Manager    7839 02-APR-91       2975            200
7698 Blake           Manager    7839 01-MAY-91       2850            300
7782 Raimi           Manager    7839 09-JUN-91       2450            100
7902 Ford            Analyst    7566 03-DEC-91       3000            200
7369 Smith           Clerk      7902 17-DEC-90        800            200
7788 Scott           Analyst    7566 09-NOV-91       3000            200
7876 Adams           Clerk      7788 23-SEP-91       1100            400
7900 James           Clerk      7698 03-DEC-91        950            300
7934 Miller          Clerk      7782 23-JAN-92       1300            100
```

```
10 rows selected.
```

The `between` operator

Oracle SQL also includes a `between` operator. The expression:

```
attrib between val1 AND val2
```

...is true if the value of `attrib` is greater than or equal to `val1` and less than or equal to `val2` -- that is, it is true if the value of `attrib` is, well, between `val1` and `val2`, inclusive. Or, it has the same value as the expression:

```
(attrib >= val1) AND (attrib <= val2)
```

So, one could write a SQL `select` to select the rows of `empl` for employees whose salary is between $1100 and $1600, inclusive, using:

```
select   *
from     empl
where    salary between 1100 and 1600;
```

When you try out this query in sqlplus, take note of how the result includes a row with salary 1100 and a row with salary 1600:

```
EMPL EMPL_LAST_NAME  JOB_TITLE  MGR  HIREDATE      SALARY COMMISSION DEP
---- --------------- ---------- ---- --------- ---------- ---------- ---
7499 Michaels        Salesman   7698 20-FEB-91       1600        300 300
7521 Ward            Salesman   7698 22-FEB-91       1250        500 300
7654 Martin          Salesman   7698 28-SEP-91       1250       1400 300
7844 Turner          Salesman   7698 08-SEP-91       1500          0 300
7876 Adams           Clerk      7788 23-SEP-91       1100            400
7934 Miller          Clerk      7782 23-JAN-92       1300            100

6 rows selected.
```

The `like` operator

Oracle SQL also includes an operator that can be used for selecting rows whose attributes match some pattern: the `like` operator. You use the `like` operator with the attribute of interest and a string pattern, which contains what you are trying to match, which may also include the **wildcard** characters `%` or _, where `%` matches any 0 or more characters, and _ matches any single character.

Examples will likely make this clearer: what if you would like to select the `empl` rows for employees whose employee number ends with a 9? Then this query would select these rows:

```
select   *
from     empl
where    empl_num like '%9';
```

Used with `like` and written as a string, the `%` here matches any number of characters that an

empl_num begins with, but the 9 at the end means that the empl_num must end with a 9 to be selected. So, the following rows are selected:

```
EMPL EMPL_LAST_NAME  JOB_TITLE  MGR  HIREDATE      SALARY COMMISSION DEP
---- --------------- ---------- ---- --------- ---------- ---------- ---
7839 King            President       17-NOV-91       5000            500
7369 Smith           Clerk      7902 17-DEC-90        800            200
7499 Michaels        Salesman   7698 20-FEB-91       1600        300 300
```

It takes some practice to get the hang of writing patterns for what you want to match -- for example, what pattern would match an employee number with an 8 **anywhere** in it (beginning, middle, or end)? Can you see that the pattern '%8%' would work for that?

- an employee number that starts with an 8 matches: 0 characters before the 8 match the first %, then the 8 matches, then the 3 characters after the 8 match the second %;
- an employee number that ends with an 8 matches: 3 characters before the 8 match the first %, then the 8 matches, then 0 characters after the 8 match the second %;
- an employee number with an 8 (or even two 8s) in the middle matches: 1 or 2 characters before an 8 match the first %, then the 8 matches, then 1 or 2 characters after an 8 match the second % (even if that includes another 8).

```
select   *
from     empl
where    empl_num like '%8%';
```

So, this query has the results:

```
EMPL EMPL_LAST_NAME  JOB_TITLE  MGR  HIREDATE      SALARY COMMISSION DEP
---- --------------- ---------- ---- --------- ---------- ---------- ---
7839 King            President       17-NOV-91       5000            500
7698 Blake           Manager    7839 01-MAY-91       2850            300
7782 Raimi           Manager    7839 09-JUN-91       2450            100
7788 Scott           Analyst    7566 09-NOV-91       3000            200
7844 Turner          Salesman   7698 08-SEP-91       1500          0 300
7876 Adams           Clerk      7788 23-SEP-91       1100            400

6 rows selected.
```

(No employee in the current rows happens to have an employee number that begins with an 8, but you can and should insert such a row and re-try this query if you'd like to see for yourself that it would also be selected by the query's where clause.)

As another example, what if you would like to select empl rows for employees who are managers, but you cannot remember if the job_title column begins with an 'm' or an 'M'? Then a query such as this would select any row with a job title of 'Manager' or 'manager' (OK, and also 'banager' or '7anager' and any other character followed by 'anager' -- but not 'Omanager', 'Super-Duper-Manager', etc.)

```
select    *
```

```
from      empl
where     job_title like '_anager';
```

This query has the results:

```
EMPL EMPL_LAST_NAME  JOB_TITLE  MGR  HIREDATE      SALARY COMMISSION DEP
---- --------------- ---------- ---- --------- ---------- ---------- ---
7566 Jones           Manager    7839 02-APR-91       2975            200
7698 Blake           Manager    7839 01-MAY-91       2850            300
7782 Raimi           Manager    7839 09-JUN-91       2450            100
```

Computed columns and column aliases

It turns out that you can project some things besides just column names in a SQL `select` statement's `select` clause. For example, SQL supports such operations as + (addition), − (subtraction), * (multiplication), and / (division) -- and when you use such operators with column names in expressions in the `select` clause, then that computation is projected.

As a rather silly first example, you could decide to project employee last names and two times their current salary:

```
select   empl_last_name, salary * 2
from     empl;
```

This will result in the following:

```
EMPL_LAST_NAME   SALARY*2
--------------- ----------
King                10000
Jones                5950
Blake                5700
Raimi                4900
Ford                 6000
Smith                1600
Michaels             3200
Ward                 2500
Martin               2500
Scott                6000
Turner               3000

EMPL_LAST_NAME   SALARY*2
--------------- ----------
Adams                2200
James                1900
Miller               2600

14 rows selected.
```

It is very important that you realize that using a SQL `select` statement -- that querying a table -- does

not change the tables in your database in **any** way -- and so, choosing to project a computation like this doesn't change the salaries of employees in the `empl` table!

If you look at the query result above, you might notice that the computed column's default column heading is, well, the computation! We'll have more sophisticated ways to change the default column headings from queries later in the semester, but in the meantime you can change the column heading in a single query's projected result by **renaming** that column using a **column alias** in that SQL `select` statement.

The syntax for this is simple -- in the `select` clause, you put a blank after the expression to be projected, and then put the desired column alias (before the comma, if any, "ending" this projection). If you don't surround the column alias with double quotes, then it will appear in all-uppercase no matter how you type it, and it mustn't contain blanks; if you do surround the column alias with double quotes, then it will appear in exactly the case you type it with, and it can contain blanks.

For example,

```
select   empl_last_name last_name, salary * 2 "double salary"
from     empl;
```

...gives the result:

```
LAST_NAME       double salary
--------------- -------------
King                    10000
Jones                    5950
Blake                    5700
Raimi                    4900
Ford                     6000
Smith                    1600
Michaels                 3200
Ward                     2500
Martin                   2500
Scott                    6000
Turner                   3000

LAST_NAME       double salary
--------------- -------------
Adams                    2200
James                    1900
Miller                   2600

14 rows selected.
```

Do you see how using the column alias `last_name`, without double quotes, appeared as `LAST_NAME` in the result, but using double quotes around the column alias `"double salary"` allowed it to contain a blank and appear in all-lowercase as given in the query?

Also be sure to note: a column alias **only** applies to the results from the single query it appears in; it,

too, cannot change the actual database or the tables in it. It only affects the displayed results of that one query.

One more caveat, in dealing with computed columns: it turns out that computations are **only** done when all of the columns involved in the computation have **non-null** values. This can sometimes look very strange in query results -- consider the result you get if you project the employee last names and the sum of the salary and commission columns as so:

```
select   empl_last_name, salary + commission "combined gross"
from     empl;
```

You might be quite surprised at the result:

```
EMPL_LAST_NAME  combined gross
--------------- --------------
King
Jones
Blake
Raimi
Ford
Smith
Michaels                  1900
Ward                      1750
Martin                    2650
Scott
Turner                    1500

EMPL_LAST_NAME  combined gross
--------------- --------------
Adams
James
Miller

14 rows selected.
```

Because only sales people have non-null commissions, they are the only employees for whom the computation salary + commission will project with a non-null result!

Table Aliases

We have mentioned **column aliases** -- there is another alias that turns out to be handy within a SQL `select` statement: **table aliases**. A table alias is when, in the `from` clause, you give a nickname (usually shorter...) to one or more of the tables in that `from` clause.

You do this by following the table name in the `from` clause with a blank, and then the desired table alias (before the comma, if any, preceding a next table name). Once you do this, you are expected to use this alias **instead** of the table name **throughout** that query -- in the `select` clause, in the `where` clause, and in all other `select` statement clauses that we will be adding as the semester progresses.

Why would you do this? Usually, to save typing in join-conditions, although sometimes also to permit certain advanced queries (such as joining a table with itself (!), which we'll discuss later in the semester).

Here's an example, projecting the department number and employee last name for all employees:

```
select  d.dept_num, empl_last_name
from    dept d, empl e
where   d.dept_num = e.dept_num;
```

Here, in the `from` clause, `d` is being set up as a table alias for table `dept`, and `e` is being set up as a table alias for table `empl`. And so, in the `select` clause and the join-condition, one can now say `d.dept_num` instead of `dept.dept_num`, and `e.dept_num` instead of `empl.dept_num`. This works, and has as its results:

```
DEP EMPL_LAST_NAME
--- ---------------
500 King
200 Jones
300 Blake
100 Raimi
200 Ford
200 Smith
300 Michaels
300 Ward
300 Martin
200 Scott
300 Turner

DEP EMPL_LAST_NAME
--- ---------------
400 Adams
300 James
100 Miller

14 rows selected.
```

Indeed, once you set up a table alias in the `from` clause, you don't get a choice about whether to use it or not elsewhere in that one query -- you'll get an error if you don't! For example, the query below will result in an Oracle error message:

```
/* SQL*Plus WON'T like having dept.dept_num in this select clause! */

select  dept.dept_num, empl_last_name
from    dept d, empl e
where   d.dept_num = e.dept_num;
```

This query will result in the error message:

```
ERROR at line 1:
ORA-00904: "DEPT"."DEPT_NUM": invalid identifier

/* but this is fine (it is the same query from earlier in this
   section
*/

select  d.dept_num, empl_last_name
from    dept d, empl e
where   d.dept_num = e.dept_num;
```

Again, like for column aliases, table aliases **only** apply for the **one** query they appear in -- they don't affect any other `select` statement.

And, a style note: as you can see, table aliases are often quite short. However, you are expected to choose them based on the names of the tables they are aliases for -- for example, it is clear, in a query involving tables named `dept` and `empl`, that `d` should stand for `dept` and that `e` should stand for `empl`. It would not be nearly so clear if you used aliases such as `x` and `y` for `dept` and `empl`...! So, you are expected to avoid choosing confusing table aliases.

Joins involving more than two tables

Note that, although we have been doing equi-joins and natural joins involving pairs of tables so far, you can have equi-joins and natural joins involving as many tables as you would like (as long as they are related to each other appropriately...!) You just have to include all of the involved tables in the SQL `select` statement's `from` clause, and include enough join-conditions to keep your result from being a partial Cartesian product!

How many join-conditions do you need? The general rule-of-thumb to remember is that, in an equi-join or natural join of X tables, you had better have at least (X-1) join-conditions (sometimes more, depending on how the tables are related to each other, but **always** at least (X-1)). If you have fewer, then you will not have a join, but a partial Cartesian product (and usually **more** rows than you want, some of which don't really make much sense...)

So, for example, to join 3 tables, you will need at least (and usually just) 2 join conditions.

As an example, what if I would like to project, for each customer, the customer's last name, the name of that customer's employee rep, and the department location of that employee? Then I need to have the `customer` and the `empl` and the `dept` tables involved, and so I need at least **two** join conditions. How can I determine what those should be? Look at how the tables are related (usually, look at the foreign keys):

- since `empl_rep` in customer is a foreign key referencing `empl`'s `empl_num`, then one join condition, relating the customer and `empl` tables, can be:

```
customer.empl_rep = empl.empl_num
```

- and since `empl`'s `dept_num` is a foreign key referencing `dept`, then another join-condition, relating the `empl` and `dept` tables, can be:

```
empl.dept_num = dept.dept_num
```

Since we've related customer to empl, and empl to dept, that should be sufficient for equi-joining these three tables:

```
select   *
from     empl, customer, dept
where    customer.empl_rep = empl.empl_num
         and empl.dept_num = dept.dept_num;
```

Using the ANSI join notation, this could be written as:

```
select   *
from     empl
             join customer on empl.empl_num = customer.empl_rep
             join dept on empl.dept_num = dept.dept_num;
```

Of course, if you are only interested in the customer's last name, the name of that customer's employee rep, and the department location of that employee, as we originally mentioned, then we could choose to just project those columns from the equi-join of those three tables:

```
select   cust_lname, empl_last_name, dept_loc
from     empl, customer, dept
where    customer.empl_rep = empl.empl_num
         and empl.dept_num = dept.dept_num;

select   cust_lname, empl_last_name, dept_loc
from     empl
             join customer on empl.empl_num = customer.empl_rep
             join dept on empl.dept_num = dept.dept_num;
```

Interestingly, these two queries result in the same relation, but the rows are displayed in a different order -- the first has the result:

```
CUST_LNAME           EMPL_LAST_NAME  DEPT_LOC
-------------------- --------------- ---------------
Firstly              Michaels        Chicago
Secondly             Martin          Chicago
Thirdly              Michaels        Chicago
```

...and the second has the result:

```
CUST_LNAME           EMPL_LAST_NAME  DEPT_LOC
-------------------- --------------- ---------------
Thirdly              Michaels        Chicago
Secondly             Martin          Chicago
Firstly              Michaels        Chicago
```

And, of course, one might choose to further restrict the rows selected -- what if, for example, I want to project the above only for customers represented by employee Michaels?

```
select   cust_lname, empl_last_name, dept_loc
from     empl, customer, dept
where    customer.empl_rep = empl.empl_num
         and     empl.dept_num = dept.dept_num
         and     empl_last_name = 'Michaels';

select   cust_lname, empl_last_name, dept_loc
from     empl
             join customer on empl.empl_num = customer.empl_rep
             join dept on empl.dept_num = dept.dept_num
where    empl_last_name = 'Michaels';
```

Both of these have the result:

```
CUST_LNAME           EMPL_LAST_NAME  DEPT_LOC
-------------------- --------------- ---------------
Firstly              Michaels        Chicago
Thirdly              Michaels        Chicago
```

The IN predicate

This is yet-another-Oracle SQL possibility for the select statement where clause. A predicate is an operator whose result is true or false -- so, the IN predicate is an operator that is true if the given attribute has a value that is one of those in the list of values on the right-hand-side of the IN predicate, and is false otherwise. The IN predicate is very useful, for example, if you would like to select rows in which some attribute is one of a small set of values. (It is also very useful in some other situations that we'll be discussing later in the semester!) You put an attribute, then the predicate IN, then a comma-separated list of values within a set of parentheses.

Say that you want to project the last names and job titles and salaries of those employees who are either managers or analysts -- you know that you can use OR for that:

```
select   empl_last_name, job_title, salary
from     empl
where    job_title = 'Analyst'
         or job_title = 'Manager';
```

...but you could also use the IN predicate for that:

```
select   empl_last_name, job_title, salary
from     empl
where    job_title IN ('Analyst', 'Manager');
```

Isn't it easy to tell, in the above query, that you want to select those rows in which job_title is either 'Analyst' or 'Manager' -- that you want to select those rows for which the row's job_title is **in** that set ('Analyst', 'Manager')?

Both of these queries have the results:

```
EMPL_LAST_NAME  JOB_TITLE       SALARY
--------------- ---------- ----------
Jones           Manager          2975
Blake           Manager          2850
Raimi           Manager          2450
Ford            Analyst          3000
Scott           Analyst          3000
```

`NOT IN` is also permitted, and it means what you probably expect: it selects those rows for which the attribute's value is `NOT IN` the given list. So, to project the last name and job title for anyone who **isn't** an analyst or a manager, you could use:

```
select   empl_last_name, job_title
from     empl
where    job_title NOT IN ('Analyst', 'Manager');
```

This query has the results:

```
EMPL_LAST_NAME  JOB_TITLE
--------------- ----------
King            President
Smith           Clerk
Michaels        Salesman
Ward            Salesman
Martin          Salesman
Turner          Salesman
Adams           Clerk
James           Clerk
Miller          Clerk

9 rows selected.
```

Aggregate functions

The last topic we'll discuss in this lab are **aggregate functions**. These are odd, but useful!

Computed columns perform a computation for **each** selected row; **aggregate functions** are functions that perform a **single** computation on **all** of the selected rows, returning the **single** result. (Aggregate functions alway return a single result in **simple** `select` statements; we'll talk later about more-advanced `select` statements in which an aggregate function can result in multiple results.)

Oracle SQL supports at least the following aggregate functions:

- `avg(<expr>)` - computes the average of `<expr>` in all selected rows
- `min(<expr>)` - computes the minimum value of `<expr>` in all selected rows
- `max(<expr>)` - computes the maximum value of `<expr>` in all selected rows

- `sum(<expr>)` - computes the sum of `<expr>` for all selected rows
- `count(<expr>)` - computes the number of rows for which `<expr>` is non-null over all selected rows

For example, if you would like to project the average employee salary, the average commission, the minimum salary, the maximum salary, the sum of all salaries, a count of how many managers there are, the minimum hiredate, and a count of how many employees have non-null commissions, you can get all of that with the query:

```
select avg(salary), avg(commission), min(salary), max(salary),
       sum(salary), count(salary), count(mgr), min(hiredate),
       count(commission)
from   empl;
```

...resulting in the (admittedly badly-line-wrapped!) results:

```
AVG(SALARY) AVG(COMMISSION) MIN(SALARY) MAX(SALARY) SUM(SALARY) COUNT(SALARY)
----------- -------------- ----------- ----------- ----------- -------------
COUNT(MGR) MIN(HIRED COUNT(COMMISSION)
---------- --------- -----------------
 2073.21429             550         800        5000       29025            14
       13 17-DEC-90                 4
```

You could certainly do fewer of these at a time...! Here's a query focusing on commissions, to emphasize the point that the aggregate functions only operate on **non-null** values:

```
select  avg(commission) "Avg Comm", min(commission) "Min Comm",
        max(commission) "Max Comm",
        sum(commission) "Comm Sum",
        count(commission) "How many have comm"
from    empl;
```

...which results in:

```
  Avg Comm   Min Comm  Max Comm   Comm Sum  How many have comm
---------- ---------- ---------- ---------- ------------------
       550          0       1400       2200                  4
```

You can use `*` with the `count` aggregate function to simply count how many rows are selected by this query -- consider the following query's results:

```
select  count(salary), count(commission), count(mgr), count(*)
from    empl;
```

...which results in:

```
COUNT(SALARY) COUNT(COMMISSION) COUNT(MGR)   COUNT(*)
------------- ----------------- ---------- ----------
           14                 4         13         14
```

Make sure that you understand the difference that the `count` aggregate function's argument makes: `count` with a column name argument projects how many of the selected rows have non-null values for that column, while `count` with `*` as its argument gives how many selected rows there are, period.

To make sure that it is clear that these aggregate functions return these computations just for the selected rows, here is an example that projects just the number of clerks and those clerks' average salary:

```
select count(*), avg(salary)
from   empl
where  job_title = 'Clerk';
```

...which results in:

```
  COUNT(*) AVG(SALARY)
---------- -----------
         4      1037.5
```

These additional simple SQL `select` statement features give you a great number of possibilities for querying the data in a database -- and these are only the beginning of the power available in this statement, as we shall see in upcoming reading packets.