CS 325 - SQL Reading Packet 4: "Sub-selects, concatenating columns, and projecting literals"

SOURCES:

- "Oracle9i Programming: A Primer," Rajshekhar Sunderraman, Addison Wesley.
- Classic Oracle example tables `empl` and `dept`, adapted somewhat over the years.
- Sharon Tuttle, “Database Design”, Lecture readings, Cal Poly Humboldt.

Aside: two useful SQL*Plus commands for debugging

Before we get started discussing nested selects/subselects, you may have noticed that, when you look at the spooled results of running a SQL script (or when you look at a SQL script's results within `sqlplus`), it can be hard to figure out which command resulted in which results.

Here are two SQL*Plus commands that can help with this: `prompt` and `set echo on`

The `prompt` command simply displays whatever text follows it to the screen. So, for example, if you are in SQL*Plus, and at the `SQL>` prompt you type:

```
SQL> prompt MOOOO
```

...you'll simply see:

```
MOOO
```

...which might not seem very useful. But if you put a `prompt` command in a SQL script, that extra text can make your output results easier to read and follow. For example, if you had the following inside of a SQL script `ck-this.sql`:

```
prompt dept table:

select *
from   dept;
```

...then, when you run that script in SQL*Plus:

```
SQL> start ck-this.sql
```

...now you have a little label before this table's contents:

```
dept table:

DEP DEPT_NAME      DEPT_LOC
--- -------------- --------------
100 Accounting     New York
200 Research       Dallas
300 Sales          Chicago
400 Operations     Boston
500 Management     New York
```

Note that a `prompt` with no text simply prints a blank line to the screen (or to the spool file), which is

also a nice effect at times.

I will likely start having you precede homework and lab exercise problems with `prompt` commands indicating which problem the subsequent statements are for, since it makes it easier for both of us to be able to tell which results are which!

The other command is not as "pretty" as `prompt`, but is still useful to know about. In SQL*Plus, you can set various aspects of the SQL*Plus environment, if you will, using the `set` command. (We'll be doing a lot with this when we talk about creating "prettier" SQL*Plus-based plain-text reports, later in the semester.)

`echo`, for example, is something you can set on or off in SQL*Plus. By default, it is set to `off`. But if you type the SQL*Plus command:

```
set echo on
```

... then that specifies that you'd like to turn echoing on within your SQL*Plus session. When echoing is turned on, and you run a SQL script, SQL*Plus then echoes, or outputs, each command in the script immediately before that command's results. That is, if you had set echoing on and then ran the above script `ck-this.sql`, you'd actually see:

```
SQL> prompt dept table:
dept table:
SQL>
SQL> select *
  2  from   dept;

DEP DEPT_NAME      DEPT_LOC
--- -------------- --------------
100 Accounting     New York
200 Research       Dallas
300 Sales          Chicago
400 Operations     Boston
500 Management     New York

SQL>
```

...and then you'd see the `dept` table contents (give or take some blank lines).

SQL*Plus will continue this behavior for the rest of your current SQL*Plus session or until you turn echoing off, using the command:

```
set echo off
```

The behavior when echoing is turned on can be annoying when you are not debugging, so you may want to simply use it when you need it (for example, for debugging), and be **sure** to set echoing off again when you are done debugging (and before your final run of a homework script...!)

Nested selects/subselects

Recall that, last time, we wrote `select` statements -- queries -- with more interesting `where` clauses. We're taking the `where` clause possibilities even further in this week's lab exercise.

You can actually write another `select` statement within a `where` clause (or even within a `from`

clause, it turns out). When you have a `select` statement inside of a `select` statement like this, that's called a **nested select**, or a **subselect** -- a `select` statement **nested** within another `select` statement, a **subselect** inside of a `select` statement.

It can be helpful to develop `select` statements using nested `select`s from the "inside-out" (although that isn't required). For example, what if you want to find out the employee number for the manager of the highest-paid clerk(s)?

If you are not sure how else to start, you might start by finding out the salary of the highest-paid clerk(s):

```
select max(salary)
from   empl
where  job_title = 'Clerk';
```

Note that this doesn't tell you which clerk or clerks have this salary -- but it does give that highest salary for any clerk.

If the `empl` table has the contents as inserted by the SQL script `set-up-ex-tbls.sql` (which we'll assume for all of the examples in this reading packet), then this query has the results:

```
MAX(SALARY)
-----------
       1300
```

So, to find out the values of `mgr` for clerks with that salary (since there **could** be more than one, after all), you could select the `mgr` values of clerks with that highest salary by nesting the above `select` within another `select` as so:

```
select mgr
from   empl
where  job_title = 'Clerk'
       and    salary =
              /* this subselect COMPUTES the desired salary to
                 compare salary to!
              */
             (select     max(salary)
              from       empl
              where      job_title = 'Clerk');
```

Make sure you understand: in the "outer" query, the rows of `empl` are selected for which `job_title` is `'Clerk'` **and** for which the `salary` is equal to that maximum `salary` that is the result of the "inner", nested query (or subquery). Then the `mgr` of those selected row or rows is projected as the final result. You never "see" the subquery's results -- they are simply used in selecting the desired rows for the "outer" query.

This query has the results:

```
MGR
----
7782
```

What if you decide you'd like the last names of such managers? That would provide an excuse to note

that there isn't any limit to how deeply you can nest -- a subquery can contain a subquery, which can contain a subquery, as deeply as you want...!

```
select   empl_last_name
from     empl
where    empl_num in
         /* "build" the list of desired managers' empl nums */
         (select  mgr
          from    empl
          where   job_title = 'Clerk'
          and     salary =
                  (select max(salary)
                   from   empl
                   where  job_title = 'Clerk'));
```

This query has the results:

```
EMPL_LAST_NAME
---------------
Raimi
```

You can have more than one table involved in nested selects (although the columns relating such tables will also often be involved). For example, what if you would like the names and salaries of employees who work in Dallas?

You can find the `dept_num`'s for departments whose location is Dallas with the following query:

```
select   dept_num
from     dept
where    dept_loc = 'Dallas';
```

This query has the results:

```
DEP
---
200
```

So, to find the names and salaries of employees who work in Dallas, the following query featuring a nested query/subquery would work:

```
select   empl_last_name, salary
from     empl
where    dept_num in
         (select dept_num
          from   dept
          where  dept_loc = 'Dallas');
```

You are selecting the rows of `empl` whose `dept_num` happens to be in the set of `dept_num`'s whose location is Dallas, and are then projecting the `empl_last_name` and `salary` from just those rows.

So, this query has the results:

```
EMPL_LAST_NAME     SALARY
```

```
--------------- ----------
Scott                 3000
Jones                 2975
Ford                  3000
Smith                  800
```

Some common errors related to nested selects/subselects

Now, there are a number of common errors that people make related to nested selects.

One is that some students will look at queries such as the above, see that "`where job_title = 'Clerk'`" appears twice, and think that there is some way to get rid of one of what they see as "duplicated" `where` conditions. This desire to abstract out repeated code is admirable in many kinds of programming, but it is a mistake here -- the "innermost" query is projecting the maximum salary of **just** clerks, and so its `where` clause is essential, and the "middle" query is supposed to project just the `mgr` column of employees who are both clerks and whose salary is the maximum for a clerk, and so its `where` clause is essential, too. You don't want the `mgr` of a sales employee who may happen to have the same salary as the highest paid clerk, for example! The moral here is to be careful that `select` statements at each level are selecting exactly the rows they need.

Another issue is related to the following: you'll notice that, in the previous examples, each subselect is the right-hand-side of some operation -- the right-hand-side of an =, for example, or the right-hand-side of an `in` operation. A sub-select has to be placed somewhere appropriate. (That's not to say that = or `in` are the only options for operators along with a subselect -- there are numerous others as well. These two operators are just quite frequent.)

Does it matter whether you use = or `in` with a subselect? It can matter -- you need to know that = expects a **single** value on its right-hand-side. Since projecting `max(salary)` is guaranteed to always project exactly one value, it is safe to have that subselect on the right-hand-side of an =, as in the above example. However, if a subselect might **ever** project more than one value, it is better style (because it will remain correct even as rows are added and deleted) to use `in`. You should remember that `in` is true if the value on the left-hand-side is equal to any of the set of values on its right-hand-side -- when that right-hand-side is a subselect, then that subselect is essentially defining the set of values being compared. There is no problem if this set has just one value in it -- a set can have a single value. It can even be empty (although of course the `in` will not be satisfied, then). So, `in` is safer if you have any doubt how many values will be projected by a subselect.

What if you'd just like the last names of all managers of clerks? The following query will give you this information:

```
select   empl_last_name
from     empl
where    empl_num in
         (select mgr
          from   empl
          where  job_title = 'Clerk');
```

This query has the results:

```
EMPL_LAST_NAME
--------------
Blake
```

```
Raimi
Scott
Ford
```

If you replace the `in` in the above example with =, you will see the error message:

```
ERROR at line 3:
ORA-01427: single-row subquery returns more than one row
```

You'll see this whenever you use the = operator with a subquery that returns more than one row.

Another common error is to think you can use aggregate functions anywhere you would like. Until we add more features to our basic `select` statement, note that aggregate functions can **only** be used within a `select` clause, to specify that a computation of particular columns from particular rows is to be projected. (Even after adding those features, where you can use aggregate function calls is still quite constrained.)

The most common error I see students making in this regard is to attempt to use aggregate functions "by themselves" within a where clause:

```
/* COMMON ERROR: aggregate functions ONLY work in a query -- they
       don't make sense "on their own". The following WON'T WORK
       (and what would it mean, anyway? max salary of WHAT rows?)
*/

select empl_num
from   empl
where  job_title = 'Clerk'
and    salary = max(salary);
```

You'll see the following error message for the above query:

```
ERROR at line 4:
ORA-00934: group function is not allowed here
```

So, if you see an error message noting that "`group function is not allowed here`", you have probably used an aggregate function call somewhere that it does not belong.

Nesting a select within a `from` clause

You will recall that, in the `from` clause, you put the table whose rows (or the tables whose Cartesian product) you want. Usually we use table names -- but since the result of a `select` statement is a relation/table, albeit unnamed, you can use a subselect or subselects within a `from` clause as well. (It is the same as an unnamed table, you see.)

That is, the following is a perfectly legal `select` statement:

```
select empl_last_name, dept_name
from   (select *
        from   empl e, dept d
        where  e.dept_num = d.dept_num)
where  dept_name = 'Operations';
```

This query has the results:

```
EMPL_LAST_NAME  DEPT_NAME
--------------- --------------
Adams           Operations
```

The subselect projects the equi-join of `empl` and `dept`, so the `from` of the outer `select` is the rows of that equi-join. The `where` clause of the outer `select`'s join then selects only those rows of the resulting equi-join in which the `dept_name` is `'Operations'`, and then the `empl_last_name` and `dept_name` columns from those selected rows are projected.

Do note that, as far as the outer select is concerned, the names of the columns it "knows" are exactly those projected by the subselect in the `from` clause -- that is, if you put:

```
select ename
from   (select dept_name dname, empl_last_name ename
        from   empl e, dept d
        where  e.dept_num = d.dept_num)
where  dname = 'Operations';
```

...this will work, but **only** if you use `ename` and `dname` in the outer select. As far as the outer query is concerned, the `from` clause here contains a 2-column table whose columns' names are `dname` and `ename`, respectively.

This query has the results:

```
ENAME
--------------
Adams
```

More nested `select` examples

Here are some more examples involving nested `selects`.

What if you would like to select the rows for employees who are clerks making more than the lowest-paid sales employee?

You can find the salary of the lowest-paid sales employee using the query:

```
select   min(salary)
from     empl
where    job_title = 'Sales';
```

This query has the results:

```
MIN(SALARY)
-----------
       1250
```

Note that you can use >, <, >=, <=, <> and != as well as = as the operator involving a subselect, but remember that, like =, you should **only** use them with subselects guaranteed to project exactly one value.

And so the above query can be a subquery within a query giving the rows from `empl` for clerks

making more than this minimum sales employee salary as so:

```
select   *
from     empl
where    job_title = 'Clerk'
         and    salary >
                (select min(salary)
                 from   empl
                 where  job_title = 'Sales');
```

This query has the results:

```
EMPL EMPL_LAST_NAME  JOB_TITLE  MGR  HIREDATE      SALARY COMMISSION DEP
---- --------------- ---------- ---- --------- ---------- ---------- ---
7934 Miller          Clerk      7782 23-JAN-16       1300            100
```

If you'd like to project just the last names of managers of clerks who make more than the average salary for clerks, then the following select would work:

```
select  empl_last_name
from    empl
where   empl_num in
        (select mgr
         from   empl
         where  job_title = 'Clerk'
                and    salary >
                       (select avg(salary)
                        from   empl
                        where  job_title = 'Clerk'));
```

Note that the innermost `select` statement is projecting the average salary of all `empl`s whose `job_title` is `'Clerk'`. The middle `select` statement is projecting the `mgr` column for `empl`s whose `job_title` is `'Clerk'` and whose `salary` is strictly greater than the average salary for clerks. And so the outermost `select` is projecting the last names of employees whose `empl_num` happens to be in the set of `mgr` values for just those clerks.

This query has the results:

```
EMPL_LAST_NAME
---------------
Raimi
Scott
```

What if you decide you'd like both the manager's name, and the clerk's name, and the salary, for clerk(s) making the highest salary? You can combine a join of the `empl` table with itself and a nested `select` to make this work:

```
select e2.empl_last_name "Manager" , e1.empl_last_name "Clerk's name",
       e1.salary "Clerk's salary"
from   empl e1, empl e2
where  e1.mgr = e2.empl_num
       and    e1.job_title = 'Clerk'
       and    e1.salary =
```

```
              (select max(salary)
              from   empl
              where  job_title = 'Clerk');
```

Or, using ANSI join notation,

```
select e2.empl_last_name "Manager" , e1.empl_last_name "Clerk's name",
       e1.salary "Clerk's salary"
from   empl e1 join empl e2
       on e1.mgr = e2.empl_num
where  e1.job_title = 'Clerk'
       and    e1.salary =
              (select max(salary)
              from   empl
              where  job_title = 'Clerk');
```

Both of these queries have the results:

```
Manager         Clerk's name    Clerk's salary
--------------- --------------- --------------
Raimi           Miller                    1300
```

Reading the above queries is a good test of your basic `select` statement understanding. Walking through the first version (that isn't using the ANSI join notation):

1. the outer `select`'s `from` clause is computing a Cartesian product of the `empl` table with itself, so the result is all combinations of the `empl` table rows with `empl` table rows. But, table aliases are being used, so the columns of the first `empl` table have the names `e1.empl_last_name`, `e1.empl_num`, `e1.salary`, etc. And the columns of the second `empl` table have the names `e2.empl_last_name`, `e2.empl_num`, `e2.salary`, etc.
2. the outer `select`'s `where` clause is then selecting only those rows from 1's Cartesian product in which:

```
e1.mgr = e2.empl_num
```

...can you see that now, the only rows left are those that combine an employee's information with the information about that employee's manager? If `e1.mgr = e2.empl_num`, then all of the `e2` columns are the data about the `e1` columns' manager.

```
AND e1.job_title = 'Clerk'
```

...and we'll only keep rows for employees who are clerks (along with their manager information)

```
AND e1.salary =
       (select max(salary)
       from   empl
       where job_title = 'Clerk')
```

...and we'll only keep rows for clerks whose salary is the maximum salary for a clerk (along with their manager information).

3. Finally, from those selected row or rows for clerks whose salary is the maximum salary for a clerk (along with their manager information), we'll project:

```
e2.empl_last_name "Manager" , e1.empl_last_name "Clerk's name",
```

```
e1.salary "Clerk's salary"
```

or, as the column aliases imply, each such clerk's manager's last name (`e2`, remember, is the manager's information), the clerk's last name, and the clerk's salary (since `e1` is the employee's information).

In SQL, you can often ask your question more than one way...

You might be noticing, by now, an interesting feature of SQL: there is often more than one query that provides the same information. (Sometimes one of those queries might require more work for the DBMS to provide the answer -- such performance considerations are beyond the scope of this course, but note that a DBMS might provide tools that allow you to find out some idea of the relative costs of two queries before actually performing them. This isn't a big deal in a small database, but it can be a very big issue when dealing with very large databases.)

Consider our nested query from earlier projecting the names and salaries of employees who work in Dallas:

```
select   empl_last_name, salary
from     empl
where    dept_num in
         (select dept_num
          from   dept
          where  dept_loc = 'Dallas');
```

This query has the results:

```
EMPL_LAST_NAME     SALARY
--------------- ----------
Scott                3000
Jones                2975
Ford                 3000
Smith                 800
```

This is a query that could also be written as a join, without using nesting:

```
select   empl_last_name, salary
from     empl e, dept d
where    e.dept_num = d.dept_num
         and dept_loc = 'Dallas';
```

And, of course, it can be written using ANSI join notation as well:

```
select   empl_last_name, salary
from     empl e join dept d
         on e.dept_num = d.dept_num
where    dept_loc = 'Dallas';
```

This is a just a fact of life in SQL -- there is usually more than one way to write a particular query!

There are some rules of thumb that can guide you -- for example, I've found that, if you are selecting rows based on some computation, nesting is usually required. And, if you are projecting columns from more than one table, you'll generally have to have a join in the "outermost" level of that query, because a `select` clause can only project columns (or computations based on columns) that appear in its

`from` clause.

But this is not to imply that nesting and joining are either-or options within a query -- a single query can involve both nesting and joins (as we saw in the query projecting the manager's name, the clerk's name, and the clerk's salary for the highest-paid clerk(s)).

Here's another example that happens to involve both a join and nesting: what if you would like the name of the employee, and the department name, for clerks making more than the minimum sales employee salary?

```
select   empl_last_name, dept_name
from     empl e, dept d
where    e.dept_num = d.dept_num
         and    job_title = 'Clerk'
         and    salary >
                (select min(salary)
                 from   empl
                   where  job_title = 'Sales');
```

Or, using ANSI join notation:

```
select   empl_last_name, dept_name
from     empl e join dept d
         on e.dept_num = d.dept_num
where    job_title = 'Clerk'
         and salary >
             (select min(salary)
              from   empl
                where  job_title = 'Sales');
```

You have to include both `empl` and `dept` in the outermost `select`'s `from` clause if you want to project `empl_last_name` (which is in the `empl` table) and `dept_name` (which is in the `dept` table). So, you need a join there. And you need the nested `select` to be able to select rows from that join in which the salary is greater than the minimum salary for a sales employee.

Both of these queries have the results:

```
EMPL_LAST_NAME  DEPT_NAME
--------------- --------------
Miller          Accounting
```

Be careful, though, to remember the meaning of the `select` statements that you are writing -- sometimes there can be subtle differences between two similar queries. For example, consider a query that projects which departments and locations have employees hired before December 1, 2017. These two queries will return those departments and locations, but with one difference:

```
select dept_name, dept_loc
from   dept
where  dept_num in
       (select dept_num
        from   empl
        where  hiredate < '01-DEC-2017');
```

```
select dept_name, dept_loc
from   dept d, empl e
where  d.dept_num = e.dept_num
and    hiredate < '01-DEC-2017';
```

Can you tell the difference? The first version is projecting those rows **from** `dept` that meet the criterion -- since each department has one row in the `dept` table, it can thus project each department at most once.

However, consider the second version -- it is computing the join of the `dept` and `empl` tables, and since these are linked via `dept_num`, you have a row for **each** employee combined with the details of that employee's department. What if two employees from the **same** department were hired before December 1, 2017? Then you'll project the `dept_name` and the `dept_loc` for each of those employees -- you might see some department names and locations more than once.

So, the first version has the results:

```
DEPT_NAME       DEPT_LOC
--------------- ---------------
Management      New York
Research        Dallas
Sales           Chicago
Accounting      New York
```

...and the second version has the results:

```
DEPT_NAME       DEPT_LOC
--------------- ---------------
Management      New York
Research        Dallas
Sales           Chicago
Accounting      New York
Research        Dallas
Research        Dallas
Accounting      New York
```

This is not a fatal flaw -- you can use `distinct` to get a true relational projection from the second version, after all -- but depending on what you want to do with the results, you might prefer one of these results to the other, and in some cases computing one might take more effort than computing the other (and, again, while we aren't getting into that in this course, there do exist tools for estimating the effort a query will take before running it, which could be important for extremely large databases).

Bizarre aside: more projecting options: projecting literals, and concatenation

We have one more operation related to nested selects to discuss. But that discussion will be smoothed a bit if we precede it with a slight aside.

First of all, it is an odd-but-true fact that one can project a literal value -- something like a number `3`, or a string `'Howdy'` -- if one wishes. And since you project the expressions in the `select` clause for each row selected in the `where` clause, you'd simply see that literal once for each row selected.

Consider, for example:

```
select 'hi'
from   dept;
```

This results in the following:

```
'H
--
hi
hi
hi
hi
hi
```

...projecting a `'hi'` for each row in `dept`! (and the heading is what is being projected, chopped off because it is longer than the contents of the column...! We will have ways to prevent this chopping later, but in the interests of less-complexity-at-once we're putting up with this chopping for now.)

Now, I'll grant you that this looks like a fairly useless feature. We'll have a nested-select-related reason to do it in a moment, however. And even before that, if you combine this with another SQL operator, **concatenation**, you can get some quite useful results.

The `concatenation` operator, `||` (that's 2 vertical bars typed with **no** space in between), simply combines the expressions on either side into a single value. If this concatenation is in a `select` clause, then you project that combined value.

For example, the query:

```
select dept_num || dept_name
from   dept;
```

results in:

```
DEPT_NUM||DEPT_NAM
------------------
100Accounting
200Research
300Sales
400Operations
500Management
```

OK, that is a rather ugly resulting column. Ah, but what if you concatenated some spaces and a dash in between the dept_num and the dept_name, and then used a column alias?

```
select dept_num || ' - ' || dept_name "Department"
from   dept;
```

...that is, concatenating a `dept_num` and a `' - '` and a `dept_name` into one column, giving the resulting column the alias `"Department"`? Then you get a much more attractive result:

```
Department
--------------------
100 - Accounting
```

```
200 - Research
300 - Sales
400 - Operations
500 - Management
```

Being able to project such combinations is an important reason why you should not fear having more-detailed columns (for example, `first_name`, `last_name`) in your tables instead of less-detailed ones (`name`) -- you can always project combinations of those columns as desired. Indeed, it is much easier to concatenate columns as desired than it is to "break up" more "composite" columns.

Imagine the possibilities here:

- If, today, you want to project a last name, then a comma, then a first name?

  ```
  last_name || ', ' || first_name
  ```

- If, tomorrow, you want to project first names, then a blank, then last names?

  ```
  first_name || ' ' || last_name
  ```

- If you want to project city, then a comma and blank, then state, then two blanks, then a zip code?

  ```
  city || ', ' || state || '  ' || zip
  ```

- ...but later you want to project the state, with the zip code in parentheses?

  ```
  state || '(' || zip || ')'
  ```

- Think of all the ways you might want to format area codes and telephone numbers over time -- if `area_code` is one column and `telephone_num` is another, you can now choose to put parentheses around the area code or not, put a dash after the area code or not, omit the area code -- just by changing what you choose to concatenate together in a projection:

  ```
  '(' || area_code || ') ' || phone_num
  ```

  ```
  area_code || '-' || phone_num
  ```

  ```
  phone_num
  ```

You can even use concatenation in this way to create a comma-separated version of your database data suitable for reading into a spreadsheet (for convenient use with its charting and graphing tools, for example), or for importing into other programs.

the `exists` predicate

A predicate is an operator that results in a value of true or false -- `in` is a predicate operator, and so are <, >, <=, >=, !=, <>, and =.

Now we are discussing another predicate operator: `exists`.

To help to explain this operator, we could use a department that happens to have no employees yet. So, we'll add a new department to the `dept` table:

```
insert into dept
values
('600', 'Computer', 'Arcata');
```

`exists` is odd in that it doesn't exactly have a left-hand-side, but its right-hand-side is a subselect that

has a rather interesting relationship to the outer select. As Sunderraman puts it, for each row in the outer select, "the `exists` predicate is true if [its] sub-select results in a **non-empty** set of values, and it is false otherwise."

So, for each row in the outer select, that row will satisfy the `exists` predicate, and be selected, if the `exists`' sub-query is **non-empty** for that row. But if that sub-query is empty for that row, then the `exists` predicate will be false for that row, and that row will **not** be selected.

Now, why wouldn't this be all-or-nothing -- why wouldn't either every row resulting from a `from` clause be selected, or none of them be selected? Because `exists` is almost always used with a so-called **correlation condition**, with a **correlated subquery**: when the subselect uses data from the outer select. That is, you may have noticed that our subqueries so far have always been able to be run independently -- if you carefully pasted them in without the parentheses around them, they would run on their own. A **correlated subquery** is different: it refers to at least one attribute from table(s) not in the `from` clause of subselect, but that are in the `from` clause of the outer select it is nested within! If you tried to run such a correlated subquery by itself, then, it would fail (since it references attributes not part of that subselect's `from` clause).

In this correlated subquery, combined with `exists`, then, each row in the `from` clause from the outer-select has the subquery tried based on **its** attribute values as referenced in that subselect; if there are **any** rows in the result, `exists` is true, and that row is selected. Otherwise, `exists` is false for that row, and it is not selected.

And all that probably sounds very bizarre, and, really, nothing but some practice (OK, maybe lots of practice) will make it clearer. This example uses `exists` with a correlated subquery to list only the locations and names of departments **with** employees:

```
select  dept_loc, dept_name
from    dept
where   exists
        (select  'a'
         from    empl
         where   empl.dept_num = dept.dept_num);
```

Remembering that we just added a row to `dept` for a `Computer` department in `Arcata`, which currently has no employees, note that this query's results properly does not include the new department:

```
DEPT_LOC        DEPT_NAME
--------------- --------------
New York        Management
Dallas          Research
Chicago         Sales
New York        Accounting
Boston          Operations
```

See how the subselect has a `where` clause using the attribute `dept.dept_num`, even though its `from` clause only contains `empl`? This would be illegal, except that the outer query does have `dept` in its `from` clause, and so that makes this subselect a correlated query, and we call `empl.dept_num = dept.dept_num` a **correlation condition, in this case**.

The effect here is that, for each row in `dept`, the DBMS will see if there is a row in `empl` for which

`empl.dept_num` is the same as **that row's** `dept_num`. If there is, then there is an employee with that department's `dept_num` -- that means that rows for this subquery `exist` for this department row, and that department's row will be selected. However, if a department has no employees, then there will be no `empl` rows in which `empl.dept_num` is the same as that department row's `dept_num`, and so since the subselect results in **no** rows for this subquery for that `dept_num`, the `exists` would be false and this row would not be selected.

That is, we are selecting only those rows of `dept` for which there exists at least one `empl` row with that row's `dept_num`.

Try it -- you'll see that the new `Computer` department indeed does not show up in the results of this query.

Why in the world are we projecting a literal in this subselect? That's considered good style, because, in this case, it is more efficient -- notice that `exists` is true or false based simply on whether the subselect has **any** rows in its result or not. `exists` does not care **what** those values are, just that rows with **something** in them result -- so why bother projecting something fancy? Projecting a small literal is about the "cheapest" projection that there is.

So, it is important to remember the following when writing queries using `exists`:

- Make sure the subselect used with `exists` is a correlated subquery, with a correlation condition -- make sure it includes a condition in its `where` clause that refers to an attribute **not** in the subquery's `from` clause, but in the outer query's `from` clause.
  - It will be considered poor style (and against course style standards) to use `exists` without such a correlation condition.
- Likewise, our course style standard will be to project a literal in a correlated subquery used with `exists`.
- Finally, because of the way that `exists` works, **sometimes** using it with a Cartesian product (where you really ought to have a join -- that is, where you should include a join condition) nevertheless gives a correct answer, if you use `distinct` to filter out the **many** excess copies of the desired rows.

This, however, will be considered poor style in this class, and will **not** be accepted for credit. It is just too easy for this kind of approach to lead to incorrect and hard-to-read and hard-to-understand queries.

Our course style (and correctness) rule-of-thumb: whenever you have N tables in a `select`'s `from` clause, you are expected to have at least (N-1) appropriate corresponding join conditions in either its `where` clause (or within `on` clauses for ANSI join notation).

not exists

`not exists` does the opposite of `exists` -- it selects those rows from the outer select for which the subselect is empty (those rows from the outer `select` for which rows do **not exist** in the subquery).

Let's use `not exists` to list which departments currently have **no** employees:

```
select  dept_loc, dept_name
from    dept
where   NOT exists
        (select   'a'
```

```
          from     empl
          where    empl.dept_num = dept.dept_num);
```

Now you would see the new `Computer` department show up in the results of this query:

```
DEPT_LOC        DEPT_NAME
--------------- ---------------
Arcata          Computer
```

I find it useful to think of the subquery being executed for **each** row of the outer query.

Aside - "cleaning up" from experiments/playing around

Note -- I am going to delete this new `Computer` department "now", before giving the results for examples in the next SQL reading packet, to hopefully avoid confusion. At this point, I could accomplish this in at least three different ways:

1. I could simply delete the Computer department's row -- here is one of the possible `delete` statements for this:

```
delete from dept
where  dept_num = '600';
```

2. I could re-run the SQL script `set-up-ex-tbls.sql`, since this drops and recreates the `dept` table -- here is one of the possible ways of doing this:

```
@ set-up-ex-tbls
```

3. If I am in the same SQL session (if I have not yet logged out since creating the `Computer` department), I can undo -- or roll back -- all changes since the database was last committed using a SQL `rollback` command:

```
rollback;
```

We'll be discussing this command more in a later packet, including discussing what is meant by committing changes to a database.