CIS 315 - Reading Packet: "Order by, group by, and having"

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

more SELECT clauses: ORDER BY, GROUP BY, and HAVING
The SELECT statement can have some additional optional clauses, in addition to the clauses discussed thus far. In this lab, we'll be discussing three such clauses: ORDER BY, GROUP BY, and HAVING.

ORDER BY
As you have written your queries, have you ever wished that the rows in the result would appear in a different order? That's all that ORDER BY does -- it has absolutely no effect on what is stored in the database (since, indeed, a SELECT never effects what is stored in a database), but it does allow the user to specify the order in which he/she would like the resulting rows to be displayed.

This should always be the final clause of a SELECT (and indeed, syntactically, it only belongs on an outer-SELECT - since it is really just specifying a final row-display order, it wouldn't make sense inside of a subselect, if you think about it.) And, in its simplest form, you just follow the ORDER BY by the column (or the projected expression) that you want the rows to be ordered by.

(If you have ever used SORT BY to change the order of rows in an Excel database, it is a similar idea -- you specify the column you want to sort the Excel table's rows by, and then the rows are sorted in order of that column's values.)

For example, say that you want to select all of the rows of the empl table, but displaying those selected rows in order of increasing salary. Then, you'd ORDER BY salary:

```
select   *
from     empl
order by salary;
```

...resulting in:

<table>
<thead>
<tr>
<th>EMPL</th>
<th>EMPL_LAST_NAME</th>
<th>JOB_TITLE</th>
<th>MGR</th>
<th>HIREDATE</th>
<th>SALARY</th>
<th>COMMISSION</th>
<th>DEP</th>
</tr>
</thead>
<tbody>
<tr>
<td>7369</td>
<td>Smith</td>
<td>Clerk</td>
<td>7902</td>
<td>17-DEC-90</td>
<td>800</td>
<td>200</td>
<td></td>
</tr>
<tr>
<td>7900</td>
<td>James</td>
<td>Clerk</td>
<td>7698</td>
<td>03-DEC-91</td>
<td>950</td>
<td>300</td>
<td></td>
</tr>
<tr>
<td>7876</td>
<td>Adams</td>
<td>Clerk</td>
<td>7788</td>
<td>23-SEP-91</td>
<td>1100</td>
<td>400</td>
<td></td>
</tr>
<tr>
<td>7521</td>
<td>Ward</td>
<td>Salesman</td>
<td>7698</td>
<td>22-FEB-91</td>
<td>1250</td>
<td>500 300</td>
<td></td>
</tr>
<tr>
<td>7654</td>
<td>Martin</td>
<td>Salesman</td>
<td>7698</td>
<td>28-SEP-91</td>
<td>1250</td>
<td>1400 300</td>
<td></td>
</tr>
<tr>
<td>7934</td>
<td>Miller</td>
<td>Clerk</td>
<td>7782</td>
<td>23-JAN-92</td>
<td>1300</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>7844</td>
<td>Turner</td>
<td>Salesman</td>
<td>7698</td>
<td>08-SEP-91</td>
<td>1500</td>
<td>0 300</td>
<td></td>
</tr>
<tr>
<td>7499</td>
<td>Michaels</td>
<td>Salesman</td>
<td>7698</td>
<td>20-FEB-91</td>
<td>1600</td>
<td>300 300</td>
<td></td>
</tr>
<tr>
<td>7782</td>
<td>Raimi</td>
<td>Manager</td>
<td>7839</td>
<td>09-JUN-91</td>
<td>2450</td>
<td>100</td>
<td></td>
</tr>
</tbody>
</table>
If you'd like to see the selected rows in order of increasing hiredate,

```
select *
from empl
order by hiredate;
```

or in order of job_title:

```
select *
from empl
order by job_title;
```

...and so on. You can see the columns projected from the rows selected in any order that you would like.

Note that what you choose to order by does not affect what *columns* are projected, or the order across that the projected columns appear -- that is determined completely by the SELECT clause.

For example, I don't even have to project the column I'm ordering by:

```
select empl_last_name
from empl
order by salary;
```

Here, then, I would get:

```
EMPL_LAST_NAME
---------------
Smith
James
Adams
Ward
Martin
Miller
Turner
Michaels
Raimi
Blake
Jones

EMPL_LAST_NAME
---------------
Ford
```
Scott
King

14 rows selected.

And, just to make sure this is clear: ORDER BY just affects the order that the rows selected by the rest of the SELECT are displayed; if you only select a few rows, then only those rows are in the ordered result:

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

results in:

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

**Multiple attributes in an ORDER BY clause**

What happens if you give multiple attributes (or expressions) in the ORDER BY clause, separated by commas? Then you are specifying additional ordering information -- you are saying what to sort by in case of TIES in the previous expression(s) given in the ORDER BY.

Say that I want to select all the rows of empl, displaying the rows in order of job_title, and if they have the same job_title, display the rows within that job_title by mgr, and if they have the same job_title and mgr, display the rows within that job_title and mgr by hiredate:

```sql
select *
from empl
order by job_title, mgr, hiredate;
```

...and you can see that this is indeed the case in the resulting rows:

```
EMPL EMPL_LAST_NAME JOB_TITLE MGR HIREDATE SALARY COMMISSION DEP
---- --------------- ------- ---------- --------- -------- ---------- --
7788 Scott Analyst 7566 09-NOV-91 3000 200
7902 Ford Analyst 7566 03-DEC-91 3000 200
7900 James Clerk 7698 03-DEC-91 950 300
7934 Miller Clerk 7782 23-JAN-92 1300 100
7876 Adams Clerk 7888 23-SEP-91 1100 400
7369 Smith Clerk 7902 17-DEC-90 800 200
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
7839 King President 17-NOV-91 5000 500
7499 Michaels Salesman 7698 20-FEB-91 1600 300 300
EMPL EMPL_LAST_NAME JOB_TITLE MGR HIREDATE SALARY COMMISSION DEP
```
You could say that we are displaying the rows in primary order by job_title, in secondary order by mgr, and in 3rd-level order by hiredate.

**NULL columns and ORDER BY**

You might wonder: what happens with NULL columns if you order by those columns?

```
select   empl_last_name, job_title, commission
from     empl
order by commission;
```

Try it, and you'll see that the result is:

<table>
<thead>
<tr>
<th>EMPLOYEE_NAME</th>
<th>JOB_TITLE</th>
<th>COMMISSION</th>
</tr>
</thead>
<tbody>
<tr>
<td>Turner</td>
<td>Salesman</td>
<td>0</td>
</tr>
<tr>
<td>Michaels</td>
<td>Salesman</td>
<td>300</td>
</tr>
<tr>
<td>Ward</td>
<td>Salesman</td>
<td>500</td>
</tr>
<tr>
<td>Martin</td>
<td>Salesman</td>
<td>1400</td>
</tr>
<tr>
<td>King</td>
<td>President</td>
<td></td>
</tr>
<tr>
<td>Jones</td>
<td>Manager</td>
<td></td>
</tr>
<tr>
<td>Adams</td>
<td>Clerk</td>
<td></td>
</tr>
<tr>
<td>Miller</td>
<td>Clerk</td>
<td></td>
</tr>
<tr>
<td>James</td>
<td>Clerk</td>
<td></td>
</tr>
<tr>
<td>Scott</td>
<td>Analyst</td>
<td></td>
</tr>
<tr>
<td>Blake</td>
<td>Manager</td>
<td></td>
</tr>
</tbody>
</table>

14 rows selected.

Similarly, if you perform the query:

```
select * 
from  empl
order by mgr;
```

...you'll find that the row for President King is the last displayed, as it is the only row containing a value of NULL for mgr.

**ORDER BY: DESC option and ASC default**

Have you noticed that all of our orderings have been in ascending order so far? That's the default for
ORDER BY. You can order rows in **descending** order of some expression by writing a blank, and then **DESC**, after the expression you want to order in descending order.

So, if you'd like to select the rows of the `empl` table, displaying the resulting rows by salary, with the HIGHEST salary first (in descending order of salary), you just write:

```sql
select   *
from     empl
order by salary desc;
```

Make sure this is clear: you put DESC after EACH attribute that you want in descending order; if you are specifying secondary or additional orderings, you must put DESC after each expression that you want to be displayed in DESC order. For example, if you are selecting the rows of the `empl` table and you'd like to display the rows:

* in descending alphabetical order by `job_title`,
* but for rows with the same `job_title`, in ascending order by `mgr`,
* but for rows with the same `job_title` and `mgr`, in descending order by `hiredate`,

...you'd put:

```sql
select   *
from     empl
order by  job_title desc, mgr, hiredate desc;
```

And, if you'd like to display those those rows in primary order of increasing salary, and in secondary order of decreasing hiredate, you'd put:

```sql
select   *
from     empl
order by salary, hiredate desc;
```

**ORDER BY style warning**

One final comment with regard to ORDER BY: do not use it in a nested select! First, it is not good style, and second, it doesn't make sense, anyway, if you really think about it. It is only reasonable at the END of a top-level (or "outermost") select. This will be a Course SQL Coding Standard, that ORDER BY clauses must only be given for top-level/"outermost" selects.

For example, then, it will go AFTER and OUTSIDE a nested select (and thus as part of the top-level select):

```sql
select   *
from     empl
where     salary >
          (select min(salary)
           from   empl
           where  job_title = 'Manager')
order by salary;
```
GROUP BY

GROUP BY is a clause that takes more effort to get comfortable with than ORDER BY, but allows for some quite nifty queries of your data. GROUP BY provides a way to "group" rows sharing common characteristics, usually so you can perform aggregate function computations on rows within those "groups".

The easiest way to get used to GROUP BY is by example. You already know how to get the average salary of all employees, or for all employees whose job_title is 'Manager', or for all employees who work in the 'Research' department -- respectively:

```sql
select avg(salary)
from   empl;
```

```sql
select avg(salary)
from   empl
where  job_title = 'Manager';
```

```sql
select avg(salary)
from   empl e, dept d
where  e.dept_num = d.dept_num
    and dept_name = 'Research';
```

But each of these queries returns just a single result, just a single row.

What GROUP BY provides is a way to get computations for different groups of rows from a single query -- if you would like to get the average salary for employees who are Managers, AND for employees who are Clerks, AND for employees who are Salesmen, etc., for all job_titles, then you GROUP BY job_title:

```sql
select   avg(salary)
from   empl
group by job_title;
```

That is, this says, get the rows of empl, group those rows by job_title, and project the average salary for each of those groups. So, the above query results in:

```
AVG(SALARY)
-----------
  3000
  1037.5
 2758.33333
   5000
   1400
```

You are also allowed to project a column you are grouping by along with any computations on those groups -- so, if you group by job_title, then you can also project job_title, if you would like, and result is that you see WHICH job_title has each average:

```sql
select   job_title, avg(salary)
from   empl
```
group by job_title;

...resulting in:

<table>
<thead>
<tr>
<th>JOB_TITLE</th>
<th>AVG(SALARY)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Analyst</td>
<td>3000</td>
</tr>
<tr>
<td>Clerk</td>
<td>1037.5</td>
</tr>
<tr>
<td>Manager</td>
<td>2758.33333</td>
</tr>
<tr>
<td>President</td>
<td>5000</td>
</tr>
<tr>
<td>Salesman</td>
<td>1400</td>
</tr>
</tbody>
</table>

Where does GROUP BY "fit" in terms of the SELECT statement syntax?
* You still perform any Cartesian products given in the FROM clause first,
* and then you select those rows from that Cartesian product that satisfy the condition(s) given in the WHERE clause.
* Then, if there is a GROUP BY clause, you group only the selected rows by the expression given in the GROUP BY clause,
* and then you project what is specified in the SELECT clause, usually the desired computations for each of those groups, and the expression you are grouping by if desired,
* ordering the rows as specified by the ORDER BY clause if it is there!

So, consider this query:

```
select   dept_name, avg(salary)
from     empl e, dept d
where     e.dept_num = d.dept_num
group by  dept_name
order by  avg(salary);
```

This will:
* perform a Cartesian product of the empl and dept tables,
* then select those rows of the Cartesian product in which e.dept_num = d.dept_num (thus performing an equi-join!),
* then, only in the rows for which e.dept_num = d.dept_num, it will group the rows by dept_name,
* then it will project the dept_name and the average salary for each set of rows grouped by dept_name,
* displaying the resulting rows in order of (increasing) average salary.

So, you would see the following:

<table>
<thead>
<tr>
<th>DEPT_NAME</th>
<th>AVG(SALARY)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operations</td>
<td>1100</td>
</tr>
<tr>
<td>Sales</td>
<td>1566.66667</td>
</tr>
<tr>
<td>Accounting</td>
<td>1875</td>
</tr>
<tr>
<td>Research</td>
<td>2443.75</td>
</tr>
<tr>
<td>Management</td>
<td>5000</td>
</tr>
</tbody>
</table>

What if you would like the minimum and maximum salaries, minimum and maximum hiredates, salary
totals, and number of employees for each value of dept_num? Then this would do the trick (although I'm projecting this information in a different order than stated above, just to make the point that you can project these columns in any order you want, and I'm happening to order the resulting rows by minimum salary):

```
select count(*), dept_num, min(salary), max(salary), min(hiredate),
       max(hiredate), sum(salary)
from empl
group by dept_num
order by min(salary);
```

...resulting in:

<table>
<thead>
<tr>
<th>COUNT(*)</th>
<th>DEP</th>
<th>MIN(SALARY)</th>
<th>MAX(SALARY)</th>
<th>MIN(HIRED)</th>
<th>MAX(HIRED)</th>
<th>SUM(SALARY)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>200</td>
<td>800</td>
<td>3000</td>
<td>17-DEC-90</td>
<td>03-DEC-91</td>
<td>9775</td>
</tr>
<tr>
<td>6</td>
<td>300</td>
<td>950</td>
<td>2850</td>
<td>20-FEB-91</td>
<td>03-DEC-91</td>
<td>9400</td>
</tr>
<tr>
<td>1</td>
<td>400</td>
<td>1100</td>
<td>1100</td>
<td>23-SEP-91</td>
<td>23-SEP-91</td>
<td>1100</td>
</tr>
<tr>
<td>2</td>
<td>100</td>
<td>1300</td>
<td>2450</td>
<td>09-JUN-91</td>
<td>23-JAN-92</td>
<td>3750</td>
</tr>
<tr>
<td>1</td>
<td>500</td>
<td>5000</td>
<td>5000</td>
<td>17-NOV-91</td>
<td>17-NOV-91</td>
<td>5000</td>
</tr>
</tbody>
</table>

I cannot stress the following two points enough:

* If you want to project MORE than one row in a query involving a projected aggregate function call, then you MUST use GROUP BY; otherwise, you can ONLY get one row in the result.

* When you DO use GROUP BY, you get ONE row for EACH value of the attribute(s) or expression(s) you are grouping by. You can only project, then, either computations on the attributes of the rows within each group, or the attribute(s) or expression(s) you are grouping by.

Oracle is a stickler on the second part of that second point -- when using GROUP BY, you really cannot project anything except the expression(s) you are grouping by or aggregate function calls for those groups. (Think about it -- since GROUP BY essentially gives you "one row" per group, what would it mean to try to project another attribute? To project empl_last_name when grouping by job_title?) So, it is an error to try to do so, even if you know that the attribute's value happens to be the same for all rows in a group. For example, this query will FAIL:

```
select dept_num, empl_last_name, min(salary), max(salary),
       min(hiredate), max(hiredate)
from empl
group by dept_num;
```

...giving the error message:

```
ERROR at line 1:
ORA-00979: not a GROUP BY expression
```

When you see this error message, chances are good you are using GROUP BY and trying to project something that is not an aggregate function call and not what you are grouping by.
Likewise, this fails with the same error message; even though I know that dept_name is the same for all rows with a given dept_num, Oracle doesn't know that:

```sql
select d.dept_num, dept_name, min(salary), max(salary),
       min(hiredate), max(hiredate)
from empl e, dept d
where e.dept_num = d.dept_num
group by d.dept_num;
```

...giving the error message:

```
ERROR at line 1:
ORA-00979: not a GROUP BY expression
```

Now, you CAN use multiple expressions after GROUP BY, separated by commas -- when you do that, you will get a group for each distinct collection of values of those expressions. So, you could project both dept_num and dept_name if you were to group by both dept_num and dept_name:

```sql
select d.dept_num, dept_name, min(salary), max(salary),
       min(hiredate), max(hiredate)
from empl e, dept d
where e.dept_num = d.dept_num
group by d.dept_num, dept_name;
```

...resulting in:

<table>
<thead>
<tr>
<th>DEP</th>
<th>DEPT_NAME</th>
<th>MIN(SALARY)</th>
<th>MAX(SALARY)</th>
<th>MIN(HIRED)</th>
<th>MAX(HIRED)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>Accounting</td>
<td>1300</td>
<td>2450</td>
<td>09-JUN-91</td>
<td>23-JAN-92</td>
</tr>
<tr>
<td>200</td>
<td>Research</td>
<td>800</td>
<td>3000</td>
<td>17-DEC-90</td>
<td>03-DEC-91</td>
</tr>
<tr>
<td>300</td>
<td>Sales</td>
<td>950</td>
<td>2850</td>
<td>20-FEB-91</td>
<td>03-DEC-91</td>
</tr>
<tr>
<td>400</td>
<td>Operations</td>
<td>1100</td>
<td>1100</td>
<td>23-SEP-91</td>
<td>23-SEP-91</td>
</tr>
<tr>
<td>500</td>
<td>Management</td>
<td>5000</td>
<td>5000</td>
<td>17-NOV-91</td>
<td>17-NOV-91</td>
</tr>
</tbody>
</table>

But remember -- each distinct combination of values of those expressions is considered a separate group:

```sql
select job_title, mgr, avg(salary), count(*)
from empl
group by job_title, mgr;
```

...resulting in:

<table>
<thead>
<tr>
<th>JOB_TITLE</th>
<th>MGR</th>
<th>AVG(SALARY)</th>
<th>COUNT(*)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clerk</td>
<td>7698</td>
<td>950</td>
<td>1</td>
</tr>
<tr>
<td>Clerk</td>
<td>7782</td>
<td>1300</td>
<td>1</td>
</tr>
<tr>
<td>Clerk</td>
<td>7788</td>
<td>1100</td>
<td>1</td>
</tr>
<tr>
<td>Clerk</td>
<td>7902</td>
<td>800</td>
<td>1</td>
</tr>
<tr>
<td>Analyst</td>
<td>7566</td>
<td>3000</td>
<td>2</td>
</tr>
<tr>
<td>Manager</td>
<td>7839</td>
<td>2758.33333</td>
<td>3</td>
</tr>
<tr>
<td>Salesman</td>
<td>7698</td>
<td>1400</td>
<td>4</td>
</tr>
</tbody>
</table>
Each distinct (job_title, mgr) pair is a SEPARATE group, as you can see in the above results.

Finally, it is Course SQL Style Standard that you should only use GROUP BY for a reason (usually, because you want some computation for the rows in each group). If you aren't performing some computation on the rows in each group, do not use group by. In particular, don't use it just to suppress duplicate rows -- that is what DISTINCT is for!

-- POOR style: (you will lose points for this!)

```sql
select   dept_name, job_title
from     empl e, dept d
where    e.dept_num = d.dept_num
group by dept_name, job_title;
```

-- BETTER style:

```sql
select   distinct dept_name, job_title
from     empl e, dept d
where    e.dept_num = d.dept_num;
```

-- ALSO good:

```sql
select   dept_name, job_title, count(*), avg(salary)
from     empl e, dept d
where    e.dept_num = d.dept_num
group by dept_name, job_title;
```

GROUP BY can be part of any select, including a nested select, although you should be careful to use a proper operator in the condition including the nested select in this case. In particular, note that, IF you are using GROUP BY, you can have an aggregate function call whose expression is another aggregate function -- you'd like the minimum of all of the averages, or the count of all of the maximums, or the sum of all of the minimums, etc.

For example, to see which employees make more than or equal to the average salary for any one department (even if not their own), you could write:

```sql
select empl_last_name, salary
from   empl
where  salary >=
   (select   min(avg(salary))
    from     empl
    group by dept_num);
```

That subquery would work on its own, too -- if you just want to know the minimum average salary for the employees with the same value of dept_num, this would do it:

```sql
select   min(avg(salary))
from     empl
group by dept_num;
```
Do not confuse ORDER BY and GROUP BY!

If you want your resulting rows to be displayed in a certain order, you STILL need ORDER BY -- it is quite common for a query to have both GROUP BY and ORDER BY clauses. Using GROUP BY does not guarantee that the resulting rows will be projected in a certain order; if you want a particular order for the resulting rows of any query, ORDER BY is needed.

So, if I would like various statistics for each department with the resulting rows ordered by minimum salary, this should be used:

```sql
select dept_num, min(salary), max(salary), min(hiredate),
       max(hiredate), sum(salary)
from empl
group by dept_num
order by min(salary);
```

Finally, it is important to remember that the selection of rows specified by a WHERE clause is done BEFORE the grouping of the resulting rows specified by a GROUP BY clause. For example, what if you'd like the average salary by department, but only for employees hired after July 15, 1991? Then this will do that, because the selection based on hiredate will be done BEFORE the grouping based on dept_num:

```sql
select dept_num, avg(salary), count(*)
from empl
where hiredate > '15-Jul-1991'
group by dept_num
order by count(*);
```

That query's results will be different from the average salary by department overall:

```sql
select dept_num, avg(salary), count(*)
from empl
group by dept_num
order by count(*);
```

HAVING

Our final new SELECT clause for this lab has a direct relationship to the GROUP BY clause. We've discussed how a select statement's WHERE clause lets you specify which rows you want to select. What if, however, you are using GROUP BY, but you don't really want to see the results for all of the groups? What if you only want to see the results for some of the resulting groups?

That's what the HAVING clause lets you do. HAVING is to groups what WHERE is to rows -- it simply gives you a way limit which groups you see in your result.

For example, what if I want dept_num's and average salaries for employees in each dept_num, but I'm only interested in dept_nums with an average salary greater than 1500. You must use HAVING to get this:

```sql
select dept_num, avg(salary)
from empl
```
group by dept_num
having avg(salary) > 1500;

And, of course, if you'd like to see those results in order of descending average salary:

```sql
select dept_num, avg(salary)
from empl
group by dept_num
having avg(salary) > 1500
order by avg(salary) desc;
```

You can limit the groups in your result based on a variety of criteria, BUT those criteria have to be "related" to the group, based on the grouped-by attributes, or on expressions using those attributes, or to computations on the group.

So, to see the department number and average salary of those with that department number, but only for dept_num's whose latest employee hiredate is after January 1, 1992:

```sql
select dept_num, avg(salary)
from empl
group by dept_num
having max(hiredate) > '01-Jan-1992'
order by avg(salary) desc;
```

And, to see the department number and average salary of those with that department number, but only for dept_num's only for dept_nums 100, 200, and 300:

```sql
select dept_num, avg(salary)
from empl
group by dept_num
having dept_num in (100, 200, 300)
order by avg(salary) desc;
```

What if I am interested in the average salaries within each department of employees hired after July 15, 1991, and only for departments with average salary greater than 1500, displaying the resulting rows in order of decreasing average salary?

```sql
select dept_num, avg(salary)
from empl
where hiredate > '15-Jul-1991'
group by dept_num
having avg(salary) > 1500
order by avg(salary) desc;
```

If we'd like the above, except projecting the department name instead of the department number:

```sql
select dept_name, avg(salary)
from empl e, dept d
where e.dept_num = d.dept_num
    and hiredate > '15-Jul-1991'
group by dept_name
having avg(salary) > 1500
order by avg(salary) desc;
```
And, a HAVING clause can be as interesting as we'd like...

```
select   dept_name, avg(salary)
from     employee, dept d
where    e.dept_num = d.dept_num
group by dept_name
having   avg(salary) > 1500
         and min(salary) < 4000
order by avg(salary) desc;
```

**Distinct with Aggregate Functions**

One final little SQL tidbit: you know that DISTINCT can be used in a SELECT to get a "pure" relational projection, to get a result with no duplicate rows. It turns out that you can use DISTINCT *within* an aggregate function call, inside its parentheses, to get that function's results just for each distinct value of that attribute.

For example, this simply counts how many rows of employee have non-NULL values for the attribute job_title:

```
select count(job_title)
from   employee;
```

...which happens to be

```
COUNT(JOB_TITLE)
----------------
  14
```

If I instead put:

```
select count(distinct job_title)
from   employee;
```

...this will instead count how many distinct, or different, job_title values there are, and since there are 5 different job_title values amongst the 14 rows of employee, this query returns:

```
COUNT(DISTINCT job_title)
------------------------
    5
```

And this gives you a slightly prettier result:

```
select   count(distinct job_title) "How Many Job-titles"
from     employee;
```

...resulting in:

```
How Many Job-titles
-------------------
    5
```