

CIS 315 - Reading Packet: "Views, and Simple Reports - Part 1"

Sources:

- * Oracle9i Programming: A Primer, Rajshekhar Sunderraman, Addison Wesley.
- * Classic Oracle example tables **emp1** and **dept**, adapted somewhat over the years

Introduction to SQL views

We've seen at least two "things" that can be created and stored within an Oracle database -- tables and sequences. Now we are introducing a third "thing" that can be created and stored within an Oracle database: a **view**.

A **view** is a "derived" table -- unlike a regular table, which contains zero or more rows of data, a **view** just contains how to **generate** the desired information whenever the view is used. It can give someone a specific "picture", or view, of certain data, without concerns about update hassles and perhaps allowing greater data security (as we will discuss).

A **view** is created based on a query, and then once it is created, it can be used as if it were an "actual" table in select statements (and it can *sometimes*, but not always, also be used within carefully-considered inserts, deletes, and updates as well, although views are most useful within select statements). But, "under the hood", the DBMS uses the view's underlying query to re-create the view every time a SQL statement uses the view.

You create a view using a **create view** statement, and you remove/delete a view using a **drop view** statement. The **drop view** statement has the syntax you would likely expect:

```
drop view view_to_remove;
```

The basic form of the **create view** statement has the following syntax:

```
create view view_name as  
select_statement ;
```

The view created then has the name *view_name*, has whatever columns are projected by the *select_statement*, and has the contents selected by the *select_statement*.

Since we'll be mucking with the example tables for this lab, I'll start with a "fresh" copy of the emp1 and dept tables (this assumes that I've made a copy of set_up_ex_tbls.sql ins whatever directory I started up sqlplus from, of course):

```
start set_up_ex_tbls.sql
```

Now, for example, the following drops and creates a view named `short_emp1` that has just four

columns: employee number, employee last name, employee job_title, and the employee number of that employee's manager:

```
drop view short_empl;  
  
create view short_empl as  
select      empl_num, empl_last_name, job_title, mgr  
from        empl;
```

Once this view has been created, you can query it as if it were a "real" table -- the only difference is, that view is "re-created" using its underlying query every time it is used. So, if I do:

```
select      *  
from        short_empl;
```

I'll get the results:

EMPL	EMPL_LAST_NAME	JOB_TITLE	MGR
7839	King	President	
7566	Jones	Manager	7839
7698	Blake	Manager	7839
7782	Raimi	Manager	7839
7902	Ford	Analyst	7566
7369	Smith	Clerk	7902
7499	Michaels	Salesman	7698
7521	Ward	Salesman	7698
7654	Martin	Salesman	7698
7788	Scott	Analyst	7566
7844	Turner	Salesman	7698

EMPL	EMPL_LAST_NAME	JOB_TITLE	MGR
7876	Adams	Clerk	7788
7900	James	Clerk	7698
7934	Miller	Clerk	7782

14 rows selected.

But if I delete rows from empl:

```
delete from empl  
where job_title = 'Clerk';
```

...and then rerun:

```
select      *  
from        short_empl;
```

...now I will see different contents in this view:

EMPL	EMPL_LAST_NAME	JOB_TITLE	MGR
------	----------------	-----------	-----

```
7839 King           President
7566 Jones          Manager    7839
7698 Blake          Manager    7839
7782 Raimi           Manager    7839
7902 Ford            Analyst    7566
7499 Michaels        Salesman   7698
7521 Ward            Salesman   7698
7654 Martin          Salesman   7698
7788 Scott           Analyst    7566
7844 Turner          Salesman   7698
```

10 rows selected.

If `short_empl` were an "actual" table, duplicating the contents of `empl`, I'd have a real data integrity headache, since I'd need to remember to change `short_empl` *every time* that `empl` was changed. But since it is a view, re-created whenever it is used based on `empl`, I don't have that worry -- every time I use `short_empl`, it will have the "right" contents, based on the current contents of `empl`.

Now, we said that a view can be used as if it were a real table -- that's not just in simple queries like that above. That's in any queries -- involving natural joins, group-by's, nested selects, whatever you wish. Here's just one example:

```
select  empl_last_name, cust_lname
from    short_empl, customer
where   short_empl.empl_num = customer.empl_rep;
```

...resulting in:

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

You can even use a view in creating another view...!

```
drop view cust_rep_display;

create view cust_rep_display as
select  empl_last_name, cust_lname
from    short_empl se, customer c
where   se.empl_num = c.empl_rep;

select  *
from    cust_rep_display;
```

Views and Database Security

There are a number of reasons for creating views -- you might create a view simply as a convenience, to make a frequently-done query more convenient. You might create one to make other queries easier. Another important reason for views is that you might create a view to improve data security.

How might a view help data security? Remember the SQL `grant` and `revoke` commands? For example:

```
grant select
on painter
to abc999, cde888, fgh777;
```

```
revoke select
on painter
from abc99, cde88, fgh77;
```

So, if a DBMS supports these commands, then one can explicitly indicate what access (select, insert, update, and/or delete) a user has to a database object. But notice this access is granted or revoked on an object-by-object basis -- you either have, say, select access to a particular object, or you don't. You can't grant select access to a user to just some columns in a table.

What if, then, a user needs to be able to have access to just some columns in a table? Someone working in a Payroll department might need access to just some of employee data, but not, perhaps, to employee home phone numbers. One solution is to create a view containing just the data that user needs, and then grant select access to that user for just that view, but not for the underlying table.

The payroll employee can then be granted select access for a view with just the employee data needed to create and process paychecks; a public kiosk in a bookstore could have select access granted for, and thus be able to display to the public, the columns of a view of bookstore inventory that doesn't include the price the bookstore paid for each title in stock. One can design the database based on its model, and then create views as needed to show different users just the "view" of the data that they need to know. This careful use of views and `grant` can help enhance database security, while at the same time, since these views are dynamically "created" whenever used, not leading to the data integrity headaches of needing to be kept up-to-date.

More view details

I mentioned that the view syntax given earlier was the "basic" form. It turns out that your view does not have to use the column names from the "original" table(s) -- there are at least two ways to specify the column names you would like for a new view. Indeed, we will see that sometimes you are required to specify a different name for a view's column.

One way to specify the column names you would like for a view is to give the desired names in a comma-separated list after the view name:

```
create view view_name (view_col1, view_col2, ...) as
select_statement;
```

Note that, using this syntax, you need to provide a column name for each column projected by the given *select_statement*.

The view **short_empl2** demonstrates this syntax:

```
drop view short_empl2;
```

```
create view short_empl2(name, "job category", manager) as
select      empl_last_name, job_title, mgr
from        empl;
```

Now see what column names you see when you query this view:

```
select      *
from        short_empl2;
```

...with the results (recall that we deleted the 4 Clerks earlier in this lab):

NAME	job category	MANAGER
King	President	
Jones	Manager	7839
Blake	Manager	7839
Raimi	Manager	7839
Ford	Analyst	7566
Michaels	Salesman	7698
Ward	Salesman	7698
Martin	Salesman	7698
Scott	Analyst	7566
Turner	Salesman	7698

10 rows selected.

Or, consider the SQL*Plus command:

```
describe short_empl2
```

...which has the results:

Name	Null?	Type
NAME	NOT NULL	VARCHAR2 (15)
job category		VARCHAR2 (10)
MANAGER		CHAR (4)

Now, it is important to realize that whatever names you give the columns of a view, you must use those column names in queries involving that view -- as far as Oracle is concerned, those are the **only** names it knows for those columns.

Thus, this FAILS:

```
select empl_last_name
from short_empl2;
```

...with the error message:

```
ERROR at line 1:
ORA-00904: "EMPL_LAST_NAME": invalid identifier
```

To Oracle, short_empl2 only has the columns name, "job category", and manager.

(I included the quoted column name as an example for short_empl2, but note that I think you should **avoid** such quoted column names for views -- they are annoying to deal with in queries, as they must **always** be quoted. For example, if I just want to project short_empl2's second column, in reverse alphabetical order of that column, I must use:

```
select  "job category"  
from    short_empl2  
order  by "job category" desc;
```

...which results in:

```
job catego  
-----  
Salesman  
Salesman  
Salesman  
Salesman  
President  
Manager  
Manager  
Manager  
Analyst  
Analyst
```

10 rows selected.

I think a one-shot column alias, or another SQL*Plus command we'll be discussing shortly, are better means for getting column names with blanks when you want them.)

I said that there were at least two ways to set the column names for a view, however. What's the other way? The other way is to simply use column aliases in the select statement used to define the view:

```
drop view short_empl3;  
  
create view short_empl3 as  
select  empl_last_name last_name, job_title position  
from    empl;  
  
select  position, last_name  
from    short_empl3  
order  by last_name;
```

And, you'll see that the above query of view short_empl3 results in:

```
POSITION  LAST_NAME  
-----  -  
Manager   Blake  
Analyst   Ford  
Manager   Jones
```

```
President   King
Salesman   Martin
Salesman   Michaels
Manager    Raimi
Analyst    Scott
Salesman   Turner
Salesman   Ward
```

10 rows selected.

Which is better? It depends on the situation. I think it is easier for the reader to tell what the view's column names are with the version where they are given after the view name. But if you are only renaming a few of the columns from the original table, using table aliases will require less typing.

I mentioned that sometimes you *have* to rename the columns. That situation is when one of the view's columns is the result of a computation or function -- since such an expression is not a "legal" column names for a table, including for a view, you must, using one of these two methods, give a proper name to such a column for your view.

For example, say that you would like a view that gives the average salary per job category -- let's call this view `salary_avgs`.

The following **WILL NOT WORK**: it will complain that you need a column alias for `avg(salary)`:

```
drop view salary_avgs;

create view salary_avgs as
select   job_title, avg(salary)
from     empl
group by job_title;
```

...which will fail with the message:

```
ERROR at line 2:
ORA-00998: must name this expression with a column alias
```

The following **WILL** work, though:

```
create view salary_avgs(job, salary_avg) as
select   job_title, avg(salary)
from     empl
group by job_title;

-- (this would work, too...:
--
--create view salary_avgs as
--select   job_title, avg(salary) salary_avg
--from     empl
--group by job_title;

select   *
from     salary_avgs;
```

...where this query of this view now results in:

JOB	SALARY_AVG
Analyst	3000
Manager	2758.33333
President	5000
Salesman	1400

Beginning of Introduction to enhancing simple ASCII reports with the help of SQL*Plus commands

You've seen how query results are displayed by default in SQL*Plus; they are usually OK, but sometimes you'd like something that looks "nicer". "Nicer" here might mean numbers formatted to the same number of decimal places, or with a nice title, or with a complete column heading, or even without ugly line-wrapping.

So, in this section we'll talk about SQL*Plus commands you can use to change how a query's results are *displayed*, so that they are more suitable for use as a *report* (a presentation of data that is well-formatted, attractive, and self-explanatory on its own to a reader).

One very short first command: if you simply type /,

/

...in SQL*Plus, that will cause the previous *SQL* command to be re-run. (Not the previous SQL*Plus command, mind you -- the previous SQL command.) This can be handy when you are tweaking your query formatting for a report.

For example, the last SQL command I performed was querying the salary_avgs view. If I now type just

/

...I'll again see the results of that query:

JOB	SALARY_AVG
Analyst	3000
Manager	2758.33333
President	5000
Salesman	1400

clear command

We'll be discussing setting up break, column, and compute commands. A report script should first make sure that some *previous* values for these are not about to mess up our results. So, it is good form to **clear** any previous values for these at the beginning of a report script:


```
clear      breaks
clear      columns
clear      computes
```

```
-- compliments of S. Griffin: yes, this works, too!!!
```

```
clear breaks columns computes
```

feedback

You know that little line that follows some query results, indicating how many rows were selected? It has a name -- it is called **feedback**.

It turns out that SQL*Plus includes commands that let you tweak this feedback, changing when it appears or even turning it off altogether.

First, if you just want to know the current value for feedback, this SQL*Plus command will tell you:

```
show feedback
```

And, here is how to set the feedback to a different number:

```
set feedback 3
```

The following, then, would let you see the effects of this:

```
show feedback
```

```
-- this'll note that 3 rows were selected.
```

```
select  *
from    painter;
```

```
-- this will not note that one row was:
```

```
select  *
from    painter
where   ptr_lname = 'Monet';
```

And sometimes, for a formal report, you just want to turn feedback off:

```
set feedback off
```

pagesize

pagesize is the number of lines in a "page" (the quantum that Oracle will display before re-displaying column headings, etc.)

You can see the current value of pagesize with:

```
show pagesize
```

...and you can set the pagesize to a desired value as so (here, I am setting it to 30 lines):

```
set pagesize 30
```

One nice trick to know: if you are essentially trying to write queries to generate a flat file of data for another program, you might set the pagesize to 0 to mean that you NEVER want page breaks.

```
set pagesize 0
```

linesize

linesize is used to indicate how many characters are in a line (before line-wrapping will occur).

You can see its current value with:

```
show linesize
```

...and you can reset it with something like this (here, I am setting it to 50 characters):

```
set linesize 50
```

newpage

If you have been looking closely, you may have noticed that each query has a blank line before its column headings. It so happens that there is a name for the number of blank lines that appear before the column headings or top title (if there is one) for each page: this is called **newpage**.

(It also appears that each SQL select statement's result starts on a new "page", pagesize- and and newpage-wise.)

To see the current value of newpage:

```
show newpage
```

Here's an example of setting it (here, I am setting it to 5 lines):

```
set newpage 5
```

Now I can also admit that, oddly enough, the number of lines in a page, in practice, is actually
pagesize + newpage

...odd but true!

And, again, when your goal is to create a flat file of data, setting newpage to 0 is a very good idea.