Queries

8 min. read · View original

If you start up SQL*Plus, you can start browsing around immediately with the SELECT statement. You don't even need to define a table; Oracle provides the built-in `dual` table for times when you're interested in a constant or a function:

```
SQL> select 'Hello World' from dual;

'HELLOWORLD
-----------
Hello World

SQL> select 2+2 from dual;

       2+2
----------
         4

SQL> select sysdate from dual;

SYSDATE
----------
1999-02-14
```

... or to test your knowledge of three-valued logic (see the "Data Modeling" chapter):

```
SQL> select 4+NULL from dual;

    4+NULL
----------
```

(any expression involving NULL evaluates to NULL).

There is nothing magic about the `dual` table for these purposes; you can compute functions using the `bboard` table instead of `dual`:

```
select sysdate,2+2,atan2(0, -1) from bboard;

SYSDATE           2+2 ATAN2(0,-1)
--------- ---------- -----------
1999-01-14          4  3.14159265
1999-01-14          4  3.14159265
1999-01-14          4  3.14159265
1999-01-14          4  3.14159265
...

1999-01-14          4  3.14159265
1999-01-14          4  3.14159265
1999-01-14          4  3.14159265

55010 rows selected.
```

but not everyone wants 55010 copies of the same result. The `dual` table is predefined during Oracle installation and, though it is just a plain old table, it is guaranteed to contain only one row because no user will have sufficient privileges to insert or delete rows from `dual`.

Getting beyond Hello World

To get beyond Hello World, pick a table of interest. As we saw in the introduction,

```
select * from users;
```

would retrieve all the information from every row of the `users` table. That's good for toy systems but in any production system, you'd be better off starting with

```
SQL> select count(*) from users;

  COUNT(*)
```

```
----------
      7352
```

You don't really want to look at 7352 rows of data, but you would like to see what's in the users table, start off by asking SQL*Plus to query Oracle's data dictionary and figure out what columns are available in the `users` table:

```
SQL> describe users
 Name                           Null?    Type
 ------------------------------ -------- ----
 USER_ID                        NOT NULL
NUMBER(38)
 FIRST_NAMES                    NOT NULL
VARCHAR2(100)
 LAST_NAME                      NOT NULL
VARCHAR2(100)
 PRIV_NAME
NUMBER(38)
 EMAIL                          NOT NULL
VARCHAR2(100)
 PRIV_EMAIL
NUMBER(38)
 EMAIL_BOUNCING_P                        CHAR(1)
 PASSWORD                       NOT NULL
VARCHAR2(30)
 URL
VARCHAR2(200)
 ON_VACATION_UNTIL                       DATE
 LAST_VISIT                              DATE
 SECOND_TO_LAST_VISIT                    DATE
 REGISTRATION_DATE                       DATE
 REGISTRATION_IP
VARCHAR2(50)
 ADMINISTRATOR_P                         CHAR(1)
 DELETED_P                               CHAR(1)
 BANNED_P                                CHAR(1)
 BANNING_USER
NUMBER(38)
 BANNING_NOTE
VARCHAR2(4000)
```

The data dictionary is simply a set of built-in tables that Oracle uses to store information about the objects (tables, triggers, etc.) that have been defined. Thus SQL*Plus isn't performing any black magic when you type `describe`; it is simply querying `user_tab_columns`, a view of some of the tables in Oracle's data dictionary. You could do the same explicitly, but it is a little cumbersome.

```
column fancy_type format a20
select column_name, data_type || '(' ||
data_length || ')' as fancy_type
from user_tab_columns
where table_name = 'USERS'
order by column_id;
```

Here we've had to make sure to put the table name ("USERS") in all-uppercase. Oracle is case-insensitive for table and column names in queries but the data dictionary records names in uppercase. Now that we know the names of the columns in the table, it will be easy to explore.

Simple Queries from One Table

A simple query from one table has the following structure:

- the select list (which columns in our report)
- the name of the table
- the where clauses (which rows we want to see)
- the order by clauses (how we want the rows arranged)

Let's see some examples. First, let's see how many users from MIT are registered on our site:

```
SQL> select email
from users
where email like '%mit.edu';

EMAIL
------------------------------
philg@mit.edu
andy@california.mit.edu
ben@mit.edu
...
```

```
wollman@lcs.mit.edu
ghomsy@mit.edu
hal@mit.edu
...
jpearce@mit.edu
richmond@alum.mit.edu
andy_roo@mit.edu
kov@mit.edu
fletch@mit.edu
lsandon@mit.edu
psz@mit.edu
philg@ai.mit.edu
philg@martigny.ai.mit.edu
andy@californnia.mit.edu
ty@mit.edu
teadams@mit.edu

68 rows selected.
```

The `email like '%mit.edu'` says "every row where the email column ends in 'mit.edu'". The percent sign is Oracle's wildcard character for "zero or more characters". Underscore is the wildcard for "exactly one character":

```
SQL> select email
from users
where email like '___@mit.edu';

EMAIL
------------------------------
kov@mit.edu
hal@mit.edu
...
ben@mit.edu
psz@mit.edu
```

Suppose that we notice in the above report some similar email addresses. It is perhaps time to try out the ORDER BY clause:

```
SQL> select email
from users
```

```
where email like '%mit.edu'
order by email;

EMAIL
------------------------------
andy@california.mit.edu
andy@californnia.mit.edu
andy_roo@mit.edu
...
ben@mit.edu
...
hal@mit.edu
...
philg@ai.mit.edu
philg@martigny.ai.mit.edu
philg@mit.edu
```

Now we can see that this users table was generated by grinding over pre-ArsDigita Community Systems postings starting from 1995. In those bad old days, users typed their email address and name with each posting. Due to typos and people intentionally choosing to use different addresses at various times, we can see that we'll have to build some sort of application to help human beings merge some of the rows in the users table (e.g., all three occurrences of "philg" are in fact the same person (me)).

Restricting results

Suppose that you were featured on Yahoo in September 1998 and want to see how many users signed up during that month:

```
SQL> select count(*)
from users
where registration_date >= '1998-09-01'
and registration_date < '1998-10-01';

  COUNT(*)
----------
       920
```

We've combined two restrictions in the WHERE clause with an AND. We can add another restriction with another AND:

```
SQL> select count(*)
from users
where registration_date >= '1998-09-01'
and registration_date < '1998-10-01'
and email like '%mit.edu';

  COUNT(*)
----------
        35
```

OR and NOT are also available within the WHERE clause. For example, the following query will tell us how many classified ads we have that either have no expiration date or whose expiration date is later than the current date/time.

```
select count(*)
from classified_ads
where expires >= sysdate
or expires is null;
```

Subqueries

You can query one table, restricting the rows returned based on information from another table. For example, to find users who have posted at least one classified ad:

```
select user_id, email
from users
where 0 < (select count(*)
           from classified_ads
           where classified_ads.user_id =
users.user_id);

   USER_ID EMAIL
---------- ----------------------------------
     42485 twm@meteor.com
     42489 trunghau@ecst.csuchico.edu
     42389 ricardo.carvajal@kbs.msu.edu
     42393 gon2foto@gte.net
     42399 rob@hawaii.rr.com
     42453 stefan9@ix.netcom.com
```

```
     42346 silverman@pon.net
     42153 gallen@wesleyan.edu
...
```

Conceptually, for each row in the `users` table Oracle is running the subquery against `classified_ads` to see how many ads are associated with that particular user ID. Keep in mind that this is only *conceptually*; the Oracle SQL parser may elect to execute this query in a more efficient manner.

Another way to describe the same result set is using EXISTS:

```
select user_id, email
from users
where exists (select 1
              from classified_ads
              where classified_ads.user_id =
users.user_id);
```

This may be more efficient for Oracle to execute since it hasn't been instructed to actually count the number of classified ads for each user, but only to check and see if any are present. Think of EXISTS as a Boolean function that

1. takes a SQL query as its only parameter
2. returns TRUE if the query returns any rows at all, regardless of the contents of those rows (this is why we can use the constant 1 as the select list for the subquery)

JOIN

A professional SQL programmer would be unlikely to query for users who'd posted classified ads in the preceding manner. The SQL programmer knows that, inevitably, the publisher will want information from the classified ad table along with the information from the users table. For example, we might want to see the users and, for each user, the sequence of ad postings:

```
select users.user_id, users.email,
classified_ads.posted
```

```
from users, classified_ads
where users.user_id = classified_ads.user_id
order by users.email, posted;

   USER_ID EMAIL
POSTED
---------- -------------------------------- --
--------
     39406 102140.1200@compuserve.com
1998-09-30
     39406 102140.1200@compuserve.com
1998-10-08
     39406 102140.1200@compuserve.com
1998-10-08
     39842 102144.2651@compuserve.com
1998-07-02
     39842 102144.2651@compuserve.com
1998-07-06
     39842 102144.2651@compuserve.com
1998-12-13
...
     41284 yme@inetport.com
1998-01-25
     41284 yme@inetport.com
1998-02-18
     41284 yme@inetport.com
1998-03-08
     35389 zhupanov@usa.net
1998-12-10
     35389 zhupanov@usa.net
1998-12-10
     35389 zhupanov@usa.net
1998-12-10
```

Because of the JOIN restriction, `where users.user_id = classified_ads.user_id`, we only see those users who have posted at least one classified ad, i.e., for whom a matching row may be found in the `classified_ads` table. This has the same effect as the subquery above.

The `order by users.email, posted` is key to making sure that the rows are lumped together

by user and then printed in order of ascending posting time.

OUTER JOIN

Suppose that we want an alphabetical list of all of our users, with classified ad posting dates for those users who have posted classifieds. We can't do a simple JOIN because that will exclude users who haven't posted any ads. What we need is an OUTER JOIN, where Oracle will "stick in NULLs" if it can't find a corresponding row in the `classified_ads` table.

```
select users.user_id, users.email,
classified_ads.posted
from users, classified_ads
where users.user_id = classified_ads.user_id(+)
order by users.email, posted;

...
   USER_ID EMAIL
POSTED
---------- ---------------------------------- --
--------
     52790 dbrager@mindspring.com
     37461 dbraun@scdt.intel.com
     52791 dbrenner@flash.net
     47177 dbronz@free.polbox.pl
     37296 dbrouse@enter.net
     47178 dbrown@cyberhighway.net
     36985 dbrown@uniden.com
1998-03-05
     36985 dbrown@uniden.com
1998-03-10
     34283 dbs117@amaze.net
     52792 dbsikorski@yahoo.com
...
```

The plus sign after `classified_ads.user_id` is our instruction to Oracle to "add NULL rows if you can't meet this JOIN constraint".

Extending a simple query into a JOIN

Suppose that you have a query from one table returning almost everything that you need, except for one column that's in another table. Here's a way to develop the JOIN without risking breaking your application:

1. add the new table to your FROM clause
2. add a WHERE constraint to prevent Oracle from building a Cartesian product
3. hunt for ambiguous column names in the SELECT list and other portions of the query; prefix these with table names if necessary
4. test that you've not broken anything in your zeal to add additional info
5. add a new column to the SELECT list

Here's an example from Problem Set 2 of a course that we give at MIT (see http://www.photo.net/teaching/psets/ps2/ps2.adp). Students build a conference room reservation system. They generally define two tables: `rooms` and `reservations`. The top level page is supposed to show a user what reservations he or she is current holding:

```
select room_id, start_time, end_time
from reservations
where user_id = 37
```

This produces an unacceptable page because the rooms are referred to by an ID number rather than by name. The name information is in the `rooms` table, so we'll have to turn this into a JOIN.

Step 1: add the new table to the FROM clause

```
select room_id, start_time, end_time
from reservations, rooms
where user_id = 37
```

We're in a world of hurt because Oracle is now going to join every row in `rooms` with every row in `reservations` where the `user_id` matches that of the logged-in user.

Step 2: add a constraint to the WHERE clause

```
select room_id, start_time, end_time
from reservations, rooms
where user_id = 37
and reservations.room_id = rooms.room_id
```

Step 3: look for ambiguously defined columns

Both `reservations` and `rooms` contain columns called "room_id". So we need to prefix the `room_id` column in the SELECT list with "reservations.". Note that we don't have to prefix `start_time` and `end_time` because these columns are only present in `reservations`.

```
select reservations.room_id, start_time, end_time
from reservations, rooms
where user_id = 37
and reservations.room_id = rooms.room_id
```

Step 4: test

Test the query to make sure that you haven't broken anything. You should get back the same rows with the same columns as before.

Step 5: add a new column to the SELECT list

We're finally ready to do what we set out to do: add `room_name` to the list of columns for which we're querying.

```
select reservations.room_id, start_time,
end_time, rooms.room_name
from reservations, rooms
where user_id = 37
and reservations.room_id = rooms.room_id
```

Reference

Next: complex queries

---

philg@mit.edu

Add a comment