CIS 315 - Reading Packet - "Intro to Oracle SQL: basics of SQL scripts, DCL and DML part 1"

Sources include:

Oracle9i Programming: A Primer, Rajshekhar Sunderraman, Addison Wesley.

Introduction

SQL, which stands for Structured Query Language, is a language with both ISO and ANSI standards, and myriad implementations. We'll be using Oracle's implementation of SQL, Oracle SQL, for this course.

We discussed how a DBMS needs to provide at least one DDL (data definition language), DML (data manipulation language), and DCL (data control language); many DBMS's, especially relational DBMS's, provide an implementation of SQL for these. That is, SQL is a DDL, DML, and DCL -- it can be used to define, manage, and update relations, query them, and control access to them.

Oracle's **SQL*Plus** program provides something of a shell for the Oracle DBMS -- like a `bash` or `csh` shell program provides an interactive environment to use with UNIX or Linux, SQL*Plus provides an interactive environment to use with the Oracle DBMS. And, as there are shell commands within a UNIX or Linux shell, there are also "shell" commands in SQL*Plus -- by tradition, these are called SQL*Plus commands. So, within the SQL*Plus shell, at the SQL*Plus command prompt, you can type **three** kinds of commands:

- **SQL** statements
- **SQL*Plus** commands
- **PL/SQL** statements [PL/SQL is Oracle's extension of SQL to include standard imperativeprogramming features such as procedures, branching, and loops; we will discuss it briefly toward the end of this course, and it is discussed more in CIS 318]

Note that SQL*Plus and PL/SQL are **not** part of the SQL standard - they are specific to Oracle. Other DBMS's may provide the facilities that these provide in different ways.

SQL Scripts

It would not be convenient to always have to type individual SQL commands at a SQL*Plus prompt. In DOS or UNIX or Linux, you can create a file of shell commands to be executed all at once as a batch file; likewise, you can also collect SQL, SQL*Plus, and PL/SQL commands together in a file, and execute those all at once as a batch file within SQL*Plus. This file is called a **SQL script**.

Our goal here is to introduce enough SQL and SQL*Plus to create a SQL script that creates and populates a table, and demonstrates that it has done so.

How to reach the course Oracle DBMS

You each have an account on the Oracle **student** database on the HSU computer **nrs-labs.humboldt.edu**. You can access this computer, and thus your Oracle account, from any computer that it on the Internet using a program called `ssh` -- and, likewise, you can transfer files from another computer to nrs-labs.humboldt.edu using a program called `sftp`. See the handout "CIS 315 - useful details - ssh, sftp, and ~st10/315submit" for where to find help with `ssh` and `sftp`.

Use `ssh` to connect to `nrs-labs.humboldt.edu`, using your HSU username and password (which is the same as you use for e-mail, to access Moodle, to log into campus labs, etc.!). You should see the nrs-labs prompt, which includes your username --- for user `who99`, for example, the prompt would probably be:

```
[who99@nrs-labs ~]$
```

This is where you will type UNIX commands, commands for creating and editing your SQL scripts, managing your UNIX files and directories, etc. You should already be familiar with the basic UNIX commands for these activities from CIS 250 - Intro to Operating Systems, which is a prerequisite to this course, but just in case I will usually mention the basic UNIX commands you need along the way, and also a handout of basic UNIX commands that might be useful for this course is posted on the public course web page.

So, how do you create a SQL script? Ideally, you type it in **within nrs-labs**, using one of the UNIX text editors -- `nano`, `vi`, and `emacs` are all available on nrs-labs. If you have never used a text editor before, start with `nano`:

```
[who99@nrs-labs ~]$ nano file-to-edit.sql
```

Here are a few `nano`-tips:

- The commands for `nano` are at the bottom of the screen
- ^ means the ctrl key -- when you see ^O, then, it means to type the ctrl key and the letter o at the **same** time
- Note that you CANNOT use the mouse to move around but you can use the arrow keys
- Note that ^O (for "writeOut" is what you do to **save** a file
- Note that ^X is how you **exit** `nano`

And when you want more power than `nano` provides, I will be putting a few links to `vi` and `emacs` on-line tutorials and references further down on my main campus web page, `users.humboldt.edu/smtuttle`.

Once you have a SQL script typed in, you will want to run it. You'll need to start up SQL*Plus.

To start up SQL*Plus on nrs-labs.humboldt.edu, FIRST navigate to the directory that you want to work in (usually, the directory where your SQL script files are!)

Remember that, in UNIX, you can change to a directory with the command `cd`:

```
[who99@nrs-labs ~]$ cd myDirectory
```

(and remember, too, that `ls` will always let you list the names of the files in your current directory, that `pwd` will tell you the name of the present working directory, and `cd` with NOTHING after it will

always take you "home", to your home directory!)

Once you are in the directory you want to work from, type the command `sqlplus` at the nrs-labs UNIX prompt:

```
[who99@nrs-labs ~]$ sqlplus
```

Hopefully, you will see something like this:

```
SQL*Plus: Release 11.2.0.1.0 Production on Wed Sep 1 11:22:30 2010

Copyright (c) 1982, 2009, Oracle.  All rights reserved.

Enter user-name:
```

Now you will enter your username and password yet again -- you'll know you have successfully entered SQL*Plus when you see the `SQL>` prompt:

```
Connected to:
Oracle Database 11g Enterprise Edition Release 11.2.0.1.0 - Production
With the Partitioning, OLAP, Data Mining and Real Application Testing options

SQL>
```

It is at this prompt that you can type SQL statements, SQL*Plus commands, and even PL/SQL, if you would like -- although more often we will type in SQL*Plus commands to run SQL scripts we have created in a file outside of SQL*Plus.

IMPORTANT: it is **important** that you explicitly **LOG OFF** from SQL*Plus before leaving -- do NOT just close your `ssh` window! Several commands work for logging off -- for example,

```
SQL> exit
```

```
SQL> quit
```

...or even typing the CTRL key and the letter `d` at the same time (`^D`).

You should then see something like:

```
Disconnected from Oracle Database 11g Enterprise Edition Release 11.2.0.1.0 -
Production
With the Partitioning, OLAP, Data Mining and Real Application Testing options
[who99@nrs-labs ~]$
```

Why is it important to explicitly exit from your SQL*Plus session? Because just closing your window doesn't actually end your SQL*Plus session, and for some period of time the session is still live, with your tables possibly still locked for use by that session. (This could happen in spite of your best intentions, by the way, if your computer crashes while you are in SQL*Plus, or there's a power failure then, etc.)

You'll know this has occurred if you try to do something very ordinary in SQL*Plus, and it fails with the Oracle error message:

```
ORA-00054: resource busy and acquire with NOWAIT specified
```

You should, at the UNIX level, find and kill the "excess" SQL*Plus session. You can find the process-

id of the offending session with the UNIX command `ps x` (for those who may be using a different UNIX shell, note that the option for the `ps` command varies under different shells...). For example, `who99` might see something like this:

```
[who99@nrs-labs ~]$ ps x
  PID TTY      STAT   TIME COMMAND
12925 ?        S      0:00 sshd: st10@pts/1
12926 pts/1    Ss     0:00 -bash
12971 pts/1    S+     0:00 sqlplus
13822 ?        S      0:00 sshd: st10@pts/6
13823 pts/6    Ss     0:00 -bash
13860 pts/6    S      0:00 sqlplus
14784 pts/6    S      0:00 /bin/bash
14866 pts/6    R+     0:00 ps x
```

See how there are two processes whose command is `sqlplus`? We don't want that! We want to kill at least the "orphaned" one. Let's assume, in this case, that was the one whose process id is `12971`. Then, you would kill the offending process with the `kill` command with the desired process id -- here, that would be:

```
[who99@nrs-labs ~]$ kill 12971
```

A beginning selection of SQL*Plus commands and SQL statements

*SQL*Plus comments*

A line that begins with `--` is a COMMENT and will be ignored; that's a single-line comment in SQL*Plus.

```
-- a comment
```

It turns out that the current Oracle DBMS also supports multi-line comments as in C++:

```
/* a
   comment
   too
*/
```

Note that experience has shown me that you **don't** want to use `--` for an in-line comment (to make the remainder of some line containing code into a comment). It seems to collide with the way one indicates that a SQL*Plus command line should extend to the next line, which is with a single dash `-` .

*Terminating SQL statements and SQL*Plus commands*

We'll see that, for readability's sake, most SQL statements should be written across more than one line. How does SQL*Plus know that a SQL statement is ended, then? It expects that a SQL statement will be terminated by either a **semicolon** (following the command), or by a **forward slash** (/) on its own line (following the end of the command). Either indicates to SQL*Plus that it should immediately execute the now-terminated SQL statement.

However -- it appears that entering a blank line while entering a SQL statement signals to SQL*Plus that the SQL statement should be buffered without yet executing it. You can actually edit this buffered latest SQL statement, although that is beyond the scope of this course (you can probably easily Google how to do so if you are interested). If you then type a `/` on its own line, it will execute the latest-buffered SQL statement (useful for redoing that latest SQL statement, by the way! And it is as close as SQL*Plus gets to history, sadly...).

But what if you enter another SQL statement, without ever typing `;` or `/`? Then the SQL statement that you never asked to have executed is simply replaced in the buffer by the new SQL command, never to have been executed (and with nary a warning or error message, either).

There are two common confusing repercussions to this:

- if you omit the semicolon on a SQL statement in your SQL script, that statement will be totally, quietly ignored (which can be baffling to debug!)
- if you insert a blank line in the MIDDLE of a SQL statement in your SQL script, you will likely get a bizarre error message, as SQL*Plus will treat the part after the blank line as a "new" SQL statement and complain accordingly.

So, **always** terminate your SQL statements with `;` or `/` and avoid blank lines inside of individual SQL statements, and you can avoid these problems.

(Also note: when you type a SQL statement directly at the `SQL>` prompt, SQL*Plus helpfully numbers each line after the first (2, 3, 4, etc.) These are NOT part of the statement, and you SHOULDN'T type them into any SQL script!! They are just for display.)

SQL*Plus commands, on the other hand, are expected to take no more than one line, and so typing ENTER (or, in a script, a newline character) terminates them and executes them. If one is really long enough that you need to continue to a new line, typing a single dash (`-`) signals SQL*Plus of this.

(But, I've found that SQL*Plus doesn't seem to complain if you end a SQL*Plus command with a semicolon, anyway. Some students, thus, tend to just end everything with a semicolon...)

*SQL*Plus commands: `spool` and `spool off`*

```
spool filename.txt
spool off
```

When you type `spool filename.txt`, a copy of everything that goes to the SQL*Plus screen is also spooled, or copied, into the file `filename.txt` (in the directory where you were when you started `sqlplus`) until the `spool off` command is reached.

Beware -- it actually buffers for a while before actually writing into *`filename.txt`*. `spool off` causes this buffer to be flushed, or emptied, so you get all of the results. If you omit `spool off`, you won't get whatever is in the buffer but not copied across yet -- you will be missing data at the end of your *`filename.txt`* file.

(You can actually spool to a file with any suffix -- it is a **COURSE STYLE STANDARD** that you spool into a `.txt` file, both to differentiate results files from SQL scripts (which we will ALWAYS name with a `.sql` suffix) and because the tool you will be submitting homeworks with expects files to end with either `.txt` or `.sql`.)

*SQL*Plus commands: `start` and `@`*

To execute a SQL script, you type:

```
SQL> start scriptname.sql
```

Note that this looks for *`scriptname.sql`* in the directory in which you started up `sqlplus` -- if it is elsewhere, you have to give the script's name relative to that directory!

Also note that SQL*Plus loves to have abbreviated forms of SQL*Plus commands -- `@` means the same as `start`. And, furthermore, because SQL*Plus expects SQL scripts to end in `.sql`, you can omit it in the `start` or `@` command, and it will assume it is there and grab the named file assuming it has the suffix `.sql`. So, all of these have the same effect: executing a SQL script `myscript.sql` in the same directory that I started `sqlplus` from:

```
SQL> start myscript.sql
SQL> start myscript
SQL> @ myscript.sql
SQL> @ myscript
```

In the interests of sanity, note that I will NOT typically give all of the possible abbreviations that SQL*Plus allows...

SQL statement: `create table`

A vital SQL statement is that for creating a table, `create table`. Here is the basic syntax:

```
create table name
(attrib_name   attrib_type    any_additional_constraints,
 attrib_name   atrib_type     any_additional_constraints,
 ...
 primary key   (attrib_name, ...)
);
```

It is a **COURSE STYLE STANDARD** that every table you create must have an explicitly-defined primary key (a shown above). We will talking in more detail about primary keys, but for now note that a primary key is a **set** of one or more attributes that uniquely identifies a row in a table.

If your table has foreign keys, each is indicated using:

```
foreign key (attrib_name, ...) references tbl(attrib_name, ...)
```

But, if the foreign key has the SAME name in this table as in parent table *`tbl`*, then you can omit repeating it:

```
foreign key (attrib_name, ...) references tbl
```

(We will discuss foreign keys in more detail later, too, but for now note that foreign keys are used to relate the data in different tables/relations.)

Notice that attribute declarations/primary key and foreign key constraints are **separated** by commas within the `create table` statement.

We are NOT going to go into all of the possible options for a `create table` statement -- Oracle has lovely on-line documentation for that (although you may have to complete a free registration to access it). But, here are some of the common data types that Oracle supports for attributes:

- `varchar2(`*`n`*`)` - a varying-length character string up to *`n`* characters long.
  - (that `2` is not a typo -- in Oracle, `varchar` is an older type, and `varchar2` is preferred.)
- `char(`*`n`*`)` - a fixed-length character string EXACTLY *`n`* characters long.
  - (yup, it is PADDED with blanks if you try to put anything shorter in. As this can lead to hard-to-find errors when comparing column values in queries, it is considered POOR STYLE to use `char` for strings that might vary in length)
- (Note: string literals in SQL are written USING SINGLE QUOTES!!!! `'Like this'`)
- `date` - a true date type, that even includes time! We'll talk about lovely date-related functions later, but in the meantime know that a `date` literal looks like `'DD-Mon-YYYY'`, for example, `'02-Apr-2008'`. Also note that `sysdate` can be used just about anywhere in SQL to indicate the current date.
- `decimal(`*`p`*`, `*`q`*`)` - a decimal number with up to *`p`* total places, *`q`* of which are fractional, although this can vary slightly with different versions of SQL. But, for example, you could store up to 999.99 in a column declared as `decimal(5, 2)`
- `integer` - an integer in the range -2,147,482,648 to 2,147,483,647
  - (Note that, unless Oracle has changed it since I last tried it, you **cannot** specify the number of digits desired for an `integer`.)
- `smallint` - a smaller integer, in the range -32768 to 32767

For example:

```
create table parts
(part_num               integer,
 part_name              varchar2(25),
 quantity_on_hand       smallint,
 price                  decimal(6,2),
 level_code             char(3),       -- level code must be 3 digits
 last_inspected         date,
 primary key            (part_num)
);
```

SQL statement: `drop table`

You can DROP a table --- destroy it, and any contents --- using the `drop table` command:

`drop table` *`tablename`*`;`

When the purpose of a SQL script is just to create a set of tables, we'll see that it is not uncommon to immediately precede each `create table` statement with a `drop table` statement for that table -- why?

...Because you cannot have two tables with the same name, and you cannot re-create an existing table --

by putting in these `drop table` commands, then whenever you re-run the script, you destroy any previous versions of the tables, and start with a new version of each. Obviously you don't do this for production tables, but it is useful for our course purposes, while we are "experimenting" with and learning SQL!

(Of course, the `drop table` commands will fail the first time you run such a script, since the tables don't exist to be dropped yet! But that error will disappear the second time you run that script.)

*SQL*Plus command: `describe`*

Once you've created a table, it is there, it is part of your database -- it is empty, and it has no rows, but it is there! (And it **persists**, and does not go away even when you exit `sqlplus`, until you **drop** it. That is what makes databases a viable alternative to files for long-term storage management.)

Would you like to be able to see something about your table, even though it is empty? The SQL*Plus command:

`describe` *`tablename`*

...will describe the table by listing its columns and their types. (You'll notice that some of our column types are aliases - those types may appear differently here! And primary keys are automatically declared as `NOT NULL`, or unable to contain a null or empty value, because Oracle **does** support **entity integrity**, as we will be discussing.)

SQL statement: `insert`

You probably want to eventually add rows to your table! Here is the simplest form of the SQL `insert` statement:

```
insert into tablename
values
(attrib1_val, attrib2_val, ...);
```

You need to type the row's values as literals -- so remember to put those single quotes around string literals and date literals! (You can use `sysdate` without quotes to indicate you'd like the current date inserted, however).

For this version, you MUST give a value for every column of the table. (If you only want to specify values for some columns in a row, then you follow the *`tablename`* with a parenthesized list of the columns you want to set, and then list that many attribute values in that order after `values`. Any unspecified columns will be `NULL`, the special name for the lack of a value in a column, for that row (or a default value, if that has been specified -- more on that later).

There are some more variations on `insert`, but one sad thing is true: NONE allow you to insert MORE than one row at a time. Yes, you REALLY have to type an ENTIRE select statement for EVERY single row. Honest. Try it if you don't believe me. (There are tools for **importing** data from files, such as **SQL*Loader**, which we'll discuss later in the semester, and shortcuts when you are moving rows from an existing table into another table -- and of course, in an application those individual `insert` statements are buried in the application code, where they are less annoying -- but this really seems to be the case for Oracle.)

Here, then, are some example `insert` statements for the `parts` table created earlier:

```
insert into parts
values
(10603, 'hexagonal wrench', 13, 9.99, '003', '05-SEP-2000');

insert into parts
values
(10604, 'tire', 287, 39.99, '333', '06-SEP-2000');
```

One more thing: note that `insert` ONLY works to insert a whole new row --- to MODIFY an EXISTING row, you'll need to use `update`, which we'll be covering in a later lab.

SQL statement: the SIMPLEST form of `SELECT`

Now, you have some data to query!

The `select` statement lets you **query** database data in ways myriad and powerful. But we'll start here with its simplest form, which simply shows all of the contents of a table:

```
select *
from   tbl_name;
```

SUBMITTING CIS 315 HOMEWORKS

You will be submitting your homeworks (and eventually most of your project milestones as well) using a tool (a Perl script) on nrs-labs called `~st10/315submit`. Here is how you use it:

1. FIRST, make sure your files are ON nrs-labs. (When you are creating and running your SQL scripts on nrs-labs, like we are today, this should be trivial! But when you use the **SQL Developer** software, and often for project milestones, then you will either use `sftp` or you will go to a campus lab and save the files to the U: drive, which for you is also nrs-labs.)
2. Use `ssh` to connect to hostname `nrs-labs.humboldt.edu`, and then use `cd` to change to the directory where the files you want to submit are stored. For example,

```
[who99@nrs-labs ~]$ cd 315lab2
```

   – It is also a really good idea to double check that your files are indeed there -- remember that `ls` lists the files in the current directory:

```
[who99@nrs-labs ~]$ ls
```

3. Once you are satisfied that you are in the right directory, type the following command:

```
[who99@nrs-labs ~]$ ~st10/315submit
```

   – It will ask you for a homework number -- for a lab exercise and for project milestones, I'll usually tell you what number to use. (For homeworks, give the number of that homework...!)

   – It will then ask you if you want to submit ALL the `.sql` and `.txt` files in the current directory. Answer `y` (for yes).

That should be it -- but NOTE that it **LISTS** the files it tried to submit, and then gives a message if it thinks the submission succeeded. **Check that list** -- do those look like the files you wanted to submit?!

Also note that you now have a `submitted` directory containing a compressed directory with what you just submitted -- THIS IS YOUR "RECEIPT" OF SUBMISSION. KEEP this directory and its contents until your grade for that work is posted on the course Moodle site!