CS 325 - SQL Reading Packet 1 - "Intro to Oracle SQL: basics of SQL scripts"

SOURCES:

- Sunderraman, "Oracle9i Programming: A Primer", Addison-Wesley.
- Sharon Tuttle, “Database Design”, Lecture readings, Cal Poly Humboldt.

Introduction

SQL, which stands for Structured Query Language, is a language with both International Organization for Standardization (ISO) and American National Standards Institute (ANSI) standards, and myriad implementations. We'll be using Oracle's implementation of SQL, Oracle SQL, for this course.

Database management system (DBMS) software can include and provide a wide variety of components and capabilities; all should include at least one data definition language (DDL), at least one data manipulation language (DML), and at least one data control language (DCL). (Note -- these happening to be called "languages" does not preclude them being possibly-graphical in nature. A graphical interface for manipulating database data can be considered a kind of DML.) Many DBMSs, especially many relational DBMSs (sometimes called RDBMSs), provide an implementation of SQL to serve as a DDL, a DML, and a DCL. That is, SQL is a DDL, DML, and DCL -- SQL can be used to define, manage, and update relational tables, query them, and control access to them.

Oracle's **SQL*Plus** program provides something of a shell for the Oracle DBMS. That is, like a `bash` or `csh` shell program provides an interactive environment to use within a UNIX or Linux operating system, 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 imperative/procedural programming features such as functions, procedures, local variables, branching, loops, and more; we will discuss it briefly at the end of this course, and it is discussed further in CS 328.)

Note that SQL*Plus and PL/SQL are **not** part of the SQL standard -- they are specific to Oracle. Other DBMSs may provide the capabilities 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 UNIX or Linux or DOS, 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 in this reading packet is to introduce enough SQL and SQL*Plus to create a SQL script that creates and populates a relational table, and demonstrates that it has done so.

How to reach the course Oracle DBMS

A version of the Oracle DBMS is running on an HSU campus server, and has been used to create a database named `student`. You each have an account on this `student` database, which can be reached from the HSU computer `nrs-projects-ssh.humboldt.edu`. You can access this computer, and thus your Oracle account, from any computer that is on the Internet using a program called `ssh` -- and, likewise, you can transfer files from another computer to `nrs-projects-ssh.humboldt.edu` using a program called `sftp`.

Use `ssh` to connect to `nrs-projects-ssh.humboldt.edu`, using your HSU username and password (which is the same as you use for e-mail, to access Canvas, to log into campus labs, etc.). **Please note:** When you enter passwords for nrs-projects and Oracle, you will not see **anything** output to the screen, not even dots or asterisks! This is a security feature, not a bug -- the system is reading what you are typing, even though you can't see it, so type your password and then type the enter/return key.

You then should see the `nrs-projects` prompt, which includes your username -- for user `who99`, for example, the prompt would be:

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

This is where you will type UNIX commands, commands for creating and editing your SQL scripts, managing your UNIX files and directories, etc. I will usually mention the basic UNIX commands you need along the way, and a handout of basic UNIX commands that might be useful for this course is posted on Canvas.

So, how do you create a SQL script? Ideally, you type it in while logged on to `nrs-projects`, using one of the UNIX text editors -- `nano (or vim)` on nrs-projects. If you have never used a text editor before, start with `nano (or vim)`:

```
[who99@nrs-projects ~]$ nano (or vim) <file_to_edit>.sql
```

Several notes about the above:

- You don't type the angle brackets, < >, above -- they mean you get to choose what goes there.
- You can use any file name, with any suffix, with `nano (or vim)` -- `.sql` is used above because SQL script file names should end with `.sql`, and that's usually what you will be using `nano (or vim)` for in this course.
- Please avoid blanks in your file names for CS 325 -- file names with blanks create unnecessary complications in UNIX/Linux.
- IF a file with that name you give doesn't yet exist, `nano (or vim)` creates it -- if it does, `nano (or vim) (or vim)` opens it for editing.

You can simply start typing to start adding contents to your new or existing file. Note that you **cannot** use the mouse to move around, but you can use the arrow keys. Also, the commands for `nano` are at the bottom of the screen -- in `nano` (as for a number of UNIX programs), when you see ^ followed by an uppercase letter, that is a common shorthand way of saying to type the ctrl key and that letter key at the same time (typing the shift key is not necessary). That is, when you see `^O` in the list of `nano` commands, that means to type the ctrl key and the `O` key at the same time -- and that is how you "writeOut", or save, the current contents to a file. And `^X` for exit means to type the ctrl key and the `X` key at the same time to exit `nano`.

(When you want more power than `nano` provides, I have a few links to `vim` online tutorials: https://coderwall.com/p/adv71w/basic-vim-commands-for-getting-started)

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-projects-ssh.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-projects ~]$ cd <directory_name>
```

(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!)

Under UNIX/Linux, you can start up SQL*Plus with the command `sqlplus /`. So, once you are in the directory you want to work from, type this command `sqlplus /` at the `nrs-projects` UNIX prompt:

```
[who99@nrs-projects ~]$ sqlplus /

SQL*Plus: Release 11.2.0.1.0 Production on Fri Jun 12 10:16:54 2015

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

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 - 64bit
Production

With the Partitioning, OLAP, Data Mining and Real Application Testing
options


SQL>
```

It is at this prompt that you can type SQL statements and SQL*Plus commands -- 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
```

...or:

```
SQL> quit
```

...or even typing `^D` (remember, in UNIX parlance that's typing the ctrl key and the `D` key at the same time, with no shift key necessary).

You should then see something like:

```
Disconnected from Oracle Database 11g Enterprise Edition Release
11.2.0.1.0 - 64bit Production
```

```
With the Partitioning, OLAP, Data Mining and Real Application options
[who99@nrs-projects ~]$
```

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 -- for example, if your computer crashes while you are in SQL*Plus, or if there is a power failure while you are working, 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
```

If you receive this error message, and you don't feel like waiting until the former SQL*Plus session expires, you can, at the UNIX level, find and kill that earlier "rogue" SQL*Plus session. You can usually find the process-id (PID) of the rogue session with the UNIX command `ps x` (this works for the default `bash` shell on `nrs-projects` -- note that `ps` command options vary quite a bit amongst different UNIX shell programs). This should list all of the current processes that you "own", even if they were started under different UNIX shell sessions (such as the one you ended, deliberately or accidentally, under which you started an earlier `sqlplus` session). For example, `who99` might see something like this:

```
[who99@nrs-projects ~]$ 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 PID is `12971`. Then, you would kill the offending process with the `kill` command with the desired PID -- here, that would be:

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

If that doesn't work, try killing it with the "kill-it-even-if-unsafely" `-9` option:

```
[who99@nrs-projects ~]$ kill -9 12971
```

(Why don't we start with `kill -9`? Because it terminates the process in a less "polite", messier way, and so it is considered better UNIX practice to try to kill a process without using the `-9` option first.)

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 `-` at the end of the SQL*Plus command line.

*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. (This is 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 the enter/return key (or, in a script, the presence of a newline character) terminates them and executes them. If a particular SQL*Plus command is long enough that you need to continue to a new line, typing a single dash (`-`) signals SQL*Plus of this.

*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`. The command `spool off` causes this buffer to be flushed, or emptied, so you get all of the results. If you omit the command `spool off`, then whatever is left in the buffer won't get copied into `<filename>.txt` -- 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 command: `prompt`*

`prompt` *desired text*

When you use the `prompt` command, the characters after the command are simply output to the screen on their own line. (And, of course, if you have done a `spool` command, those characters are output to the file you are spooling to as well.)

If you follow the `prompt` command with nothing, a blank line will be output.

For example, if you happened to type the following into `sqlplus`, here's what you would see:

```
SQL> prompt Howdy there CS 325
Howdy there CS 325
SQL> prompt

SQL>
```

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

To execute a SQL script, you can 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> <attrib_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 (as shown above).

We will be discussing the concept of **foreign keys** later -- these are primary key attributes from one table that are also in another table, so that those tables can be related to each other. But, in the meantime, some of our examples will include such foreign keys. 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>
```

Notice that attribute declarations and 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 online 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` in `varchar2` 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.
  - (Yes, it is padded with blanks if you try to put anything shorter into such a column. As this can lead to hard-to-find errors when comparing column values in queries, it is considered **poor style** to use `char` for columns whose string contents 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 much later in the semester, but, in the meantime, know that two of the syntactically-allowed ways to write date literals are `'DD-Mon-YYYY'` and `'DD-Mon-YY'`, for example, `'06-Jun-2015'` and `'06-Jun-15'`
  - Also note that `sysdate` is a built-in function (called without parentheses!) that expects no arguments and returns the current date, that can be used in many SQL commands 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.
  - For example, you could store up to `999.99` in a column declared as `decimal(5, 2)`, but you cannot store `1000.00` in such a column.
- `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 in Oracle's implementation of SQL.)
- `smallint` - a smaller integer, in the range -32,768 to 32,767
- `number` - an all-purpose type for fixed and floating point numbers
  - This type is good for when a column's domain is numeric and wide-ranging.
  - according to http://docs.oracle.com/cd/B28359_01/server.111/b28318/datatype.htm (accessed 2015-09-04):

Using the `number` type, "Numbers of virtually any magnitude can be stored and are guaranteed portable among different systems operating Oracle Database, up to 38 digits of precision."

For example, here is an example table whose columns use a variety of the above types:

```
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 related 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, you destroy any previous versions of the tables and start with new versions whenever you re-run the script. 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.)

If a table is the "parent" to any foreign keys in other tables -- sometimes called "child" tables of such a "parent" -- then you will find that you either need:

```
drop table <tablename> cascade constraints;
```

...or you will need to drop such "child" tables before dropping the "parents" they reference.

*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).

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 in the `describe` command's output! And primary keys are automatically declared as `not null`, or unable to contain a null or empty value, because Oracle supports a concept called **entity integrity**, which we will cover in a later reading packet.

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 call `sysdate` 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, comma-separated 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`, containing no value, 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** `insert` 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, 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's implementation of SQL.)

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

```
insert into parts
values
(10603, 'hexagonal wrench', 13, 9.99, '003', '01-SEP-2015');

insert into parts
values
(10604, 'tire', 287, 39.99, '333', '02-SEP-2015');
```

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 reading packet.

SQL statement: the simplest form of `select`

Now, you have some data to query!

The `select` statement lets you query, or request, 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>;
```

For example,

```
select *
from parts;
```

...at this point (based on this packet's `create table` statement and two `insert` statements) would have the results:

```
  PART_NUM PART_NAME                     QUANTITY_ON_HAND PRICE      LEV LAST_INSP
---------- ------------------------- ---------------- ---------- --- ---------
```

```
    10603 hexagonal wrench                              13       9.99 003 01-SEP-15
    10604 tire                                         287      39.99 333 02-SEP-15
```

Submitting CS 325 Course Work

You will be submitting the SQL-related portions of your homeworks and lab exercises and your project on Canvas. make sure you use some version of `sftp`, such as the command line version, `FileZilla`, or `WinSCP`, to transfer your files from `nrs-projects`. Refer to the “sftp - how to transfer files to and from nrs-projects” handout on Canvas for more details.