CS 318 - Homework 1

Deadline:

Due by 11:59 pm on Wednesday, February 6, 2013

How to submit:

Submit your files for this homework using `~st10/318submit` on nrs-projects, with a homework number of `1`

Instructions for using the tool ~st10/318submit:

- If they are not already on nrs-projects, transfer your files to be submitted to a directory on nrs-projects.
  - You can do so by using `sftp` (secure file transfer) and connecting to `nrs-projects.humboldt.edu`, and then transferring them
- Once all of your files to be submitted are in a directory on nrs-projects, then use `ssh` (or Putty in an Academic Computing lab) to connect to `nrs-projects.humboldt.edu` .
- use `cd` to change to the directory containing the files to be submitted -- for example,

  ```
  cd 318hw01
  ```

- type the command:

  ```
  ~st10/318submit
  ```

  ...and give the number of the homework being submitted (or whatever number you have been asked to do for lab-related files) when asked, and answer that, `y`, you do want to submit all of the files with of-interest-to-318 suffixes in the current directory. (Note that I don't mind if a few extraneous files get submitted as well -- I'd rather receive too many files that too few, and typing in all of the file names for each assignment is just too error-prone...)
- you are expected to carefully check the list of files that the tool believes have been submitted, and make sure all of the files you hoped to submit were indeed submitted! (The most common error is to try to run `~st10/318submit` while in a different directory than where your files are...)

Purpose:

To practice with PL/SQL stored procedures and functions

Important notes:

Here is an opening/beginning set of CS 318 SQL and PL/SQL Style Standards -- your SQL and PL/SQL code is expected to conform to these standards:

- THOU SHALT precede each PL/SQL stored procedure and stored function with an opening comment block that includes at least:
  - the procedure's or function's name
  - a purpose statement which explicitly describes what the procedure or function expects and what it does and/or returns
- THOU SHALT put a blank line before and after each `SELECT` statement, before and after each PL/SQL trigger, stored procedure, and stored function, and usually before each multi-line SQL statement, for better readability. And logically group SQL*Plus statements within a script as well.
- THOU SHALT write a `SELECT` statement's `FROM` clause on its OWN line.
- THOU SHALT write the beginning of a `SELECT` statement's `WHERE` clause on its OWN line.
- THOU SHALT indent nested selects by at least 4 spaces.
- When a `SELECT` statement has N tables/relations in its `FROM` clause (OR for joins written using the INNER JOIN syntax), THOU SHALT have (N-1) join conditions (unless one REALLY, TRULY wants a true Cartesian product, a rare occurrence...).
- THOU SHALT use mnemonic table aliases (`d` and `e` for tables `dept` and `empl`, for example, not `x` and `y`...).
- THOU SHALT only use `GROUP BY` clauses when one has a good reason (usually a computation that you wish done to those groups).
- THOU SHALT only use `ORDER BY` clauses for an outermost `SELECT` (not within any sub-selects), and it shall be indented to make clear that it "belongs" to the outermost `SELECT`.

...and I reserve the right to add to this list over the course of the semester.

The Problems:

Problem 1:

We will be practicing many of the concepts and languages we will be using this semester on a little bookstore database, whose tables can be found in the scripts `create-bks.sql` and `pop-bks.sql`. Since you will be spending a lot of time using this database, you would be well-served to start getting familiar with it now.

An one means of getting familiar with this database, write each of the relations in this database in **relation structure form**. (Recall that one writes a relation in relation-structure form by writing the name of the relation, followed by an open parenthesis, followed by a comma-separated list of all of the attributes of that relation, writing the primary-key attributes in all-uppercase.)

The biggest variation in different versions of relation-structure form is, how do you indicate the foreign keys? One way is to write SQL-style foreign key clauses after the relation structure for each of the foreign keys within that relation; that's what you are to do for this problem.

Submit a file `design-bks.txt` containing your resulting relation structures.

Problem 2:

Create a file `318hw1.sql` . Start this file with comments containing at least your name, `CS 318 - Homework 1`, and the last-modified date.

Include the command to `set serveroutput on`, followed by a SQL*Plus `spool` command to spool the results of running this SQL script to a file named `318hw1-out.txt`. Be sure to `spool off` at the end of this script (after your statements for the remaining problems).

Then, write a SQL*Plus `prompt` command that says `problem 2`. (You may add additional `prompt` commands around this to make it more visible, if you would like.)

Then, design and write a PL/SQL stored procedure `title_total_cost` that expects one parameter, a title's ISBN, and prints to the screen a message that gives the total **COST** (**not** price!) of all of the current quantity-on-hand for that title. This message should also include the title's ISBN and its current quantity on hand. (If there is no title with that ISBN, it should print a message saying so, also including that non-existent ISBN in the error message.)

After this stored procedure, include the following testing calls for this procedure:

```
prompt
prompt ************************
prompt TESTING title_total_cost
prompt ************************
prompt

prompt test passes if it shows 35 copies with a cost of $1137.5:
prompt ========================================================
exec title_total_cost('0871507870')

prompt
prompt test passes if it shows 3 copies with a cost of $79.5
prompt ======================================================
exec title_total_cost('087150331X')
prompt

prompt
prompt test passes if it notes there is NO title w/this ISBN
prompt ======================================================
exec title_total_cost('1313131313')
prompt
```

Debug and correct `title_total_cost` as necessary until it passes the above tests and you are convinced that it is working. You may add additional testing calls if you would like.

Problem 3:

Write a SQL*Plus `prompt` command that says `problem 3`.

Then, design and write a PL/SQL stored procedure `silly_shout`, to try your hand at PL/SQL's LOOP and IF statements. `silly_shout` expects two parameters, an integer parameter and a

VARCHAR2 parameter. If the integer is less than or equal to 0, the procedure should print a message to the screen saying that it cannot show the parameter string that many times; otherwise, it should print that parameter string to the screen that many times, once per line, each time concatenating an exclamation point character ('!') to the end (get it? so it is "shouting" that parameter to the screen? 8-))

After this stored procedure, include the following testing calls for this procedure:

```
prompt
prompt ************************
prompt TESTING silly_shout
prompt ************************
prompt

prompt test passes if it shows 3 "shouts" of howdy:
prompt ============================================================
exec silly_shout(3, 'howdy')

prompt
prompt test passes if it complains that it can't "shout" hi 0 times:
prompt ============================================================
exec silly_shout(0, 'hi')

prompt
prompt test passes if it complains that it can't "shout" hi -1 times:
prompt ============================================================
exec silly_shout(-1, 'hi')
```

Debug and correct `silly_shout` as necessary until it passes the above tests and you are convinced that it is working. You may add additional testing calls if you would like.

Problem 4:

Write a SQL*Plus `prompt` command that says `problem 4`.

Then, design and write a PL/SQL stored function `max_on_hand`. `max_on_hand` expects one parameter, the publisher name, and returns the maximum current quantity on hand amongst all of the titles from the publisher with that name. (If there is no publisher with that name, it should just return `0`) For example, the call `max_on_hand('Prentice-Hall')` returns `10`, the call `max_on_hand('Merrill')` returns `0`, and the call `max_on_hand('nonexistent')` returns `0`.

After this stored function, include the following testing calls:

```
prompt
prompt ********************************
prompt TESTING max_on_hand
prompt ********************************
prompt

prompt test passes if it shows a maximum quantity of 10:
prompt ============================================================
var prent_max number;
```

```
exec :prent_max := max_on_hand('Prentice Hall')
print prent_max

prompt
prompt test passes if it shows a maximum quantity of 0:
prompt ============================================================
var merr_max number;
exec :merr_max := max_on_hand('Merrill')
print merr_max

prompt
prompt test passes if it shows a maximum quantity of 0:
prompt ============================================================
var nonexist_max number;
exec :nonexist_max := max_on_hand('nonexistent')
print nonexist_max
```

Debug and correct `max_on_hand` as necessary until the above tests succeed and you are convinced that it is working. You may add additional testing calls if you would like.

Problem 5:

Write a SQL*Plus `prompt` command that says `problem 5`.

Then, design and write a PL/SQL stored procedure `which_titles` which expects two parameters, which I'll call `quant_limit` and `price_limit`. `which_titles` should print the title name, quantity on hand, and price (separated by hyphens, with a '$' before the price) of each title whose `qty_on_hand` >= `quant_limit` and whose `price` >= `price_limit`.

For example,

```
SQL> exec which_titles(15, 20)
```

...should result in:

```
Computers and Data Processing-15-$34.95
Data Base Management-20-$37.95
Intro to Biology: A Human-35-$41.95
SPSS-75-$28.95
Management Information Sy-30-$28.95
```

...being printed to the screen.

After this stored procedure, include the following testing calls for this procedure:

```
prompt
prompt ********************************
prompt TESTING which_titles
prompt ********************************
prompt

prompt test passes if 5 titles, quantities, and prices are shown
```

```
prompt    (but see Homework 1 handout for exact expected values):
prompt ================================================================
exec which_titles(15, 20)

prompt
prompt test passes if NO titles, quantities, and prices are shown:
prompt ================================================================
exec which_titles(15, 100)

prompt
prompt test passes if NO titles, quantities, and prices are shown:
prompt ================================================================
exec which_titles(100, 20)

prompt
prompt test passes if NO titles, quantities, and prices are shown:
prompt ================================================================
exec which_titles(100, 100)
```

Debug and correct `which_titles` as necessary until it passes the above tests and you are convinced that it is working. You may add additional testing calls if you would like.

Problem 6:

Write a SQL*Plus `prompt` command that says `problem 6`.

Consider: sometimes, you just would like a nice unique next-key for a table (and for the sake of some PL/SQL practice, assume you have conveniently forgotten about the existence of sequences...).

For example, consider the `order_needed` table. We're going to find that rows are added to this table when sales reduce a book's quantity below the order point -- so, it might be handy to have a function return the next suitable key value for the `order_needed` table's primary key, `on_key`.

Design and write a small PL/SQL stored function `next_on_key`; it doesn't need any parameters, and it should simply return a value that is one more than the largest current value of `on_key`. (BUT: what if the table is ever empty when this is called? Then this function should return a key of `1`.)

After this stored function, include the following testing calls:

```
prompt
prompt ****************************************
prompt TESTING next_on_key
prompt ****************************************
prompt

prompt test passes if the next on_key suggested is 1011:
prompt ================================================================
var result_key number
exec :result_key := next_on_key
print result_key

commit;
```

```
-- temporarily remove all rows from order_needed
delete from order_needed;

prompt
prompt test passes if the next on_key suggested is 1:
prompt ============================================================
exec :result_key := next_on_key
print result_key

-- "put back" all rows from order_needed
rollback;

-- temporarily modify a row from order_needed
update order_needed
set on_key = 2012
where on_key = 1006;

prompt
prompt test passes if the next on_key suggested is 2013:
prompt ============================================================
exec :result_key := next_on_key
print result_key

-- undo the temporary modification
rollback;
```

Debug and correct `next_on_key` as necessary until the above tests succeed and you are convinced that it is working. You may add additional testing calls if you would like.

Problem 7:

To practice some PL/SQL exception handling...

Design and write a PL/SQL stored function `is_on_order` that takes as its parameter an ISBN, and returns boolean `true` if that ISBN is currently on order (if its `on_order` attribute has the value `'T'`) and returns boolean `false` otherwise. (Note that we specifically want a return type of boolean!)

Within this function, use an exception section to handle the exception of the ISBN not being in the title table (let the system raise this `NO_DATA_FOUND` exception; your function should merely be able to handle it.) Have the function return boolean `false` in this case. (That is, if `is_on_order` is called with an ISBN not in the title table, then the function should simply return `false` -- it cannot be on order if it isn't a title -- rather than fail with an exception error message.)

Uh oh -- remember how we mentioned in class that PL/SQL has a `boolean` data type, but SQL does not? That turns out to be an annoyance in testing this -- it turns out that one can't declare a SQL*Plus variable of type `boolean`! (Or, I haven't gotten it to work yet.) You also cannot use a `select` to project this function's result, either. Bother.

So -- also write a little PL/SQL stored function `bool_to_string` that expects a `boolean` parameter, and returns the `varchar2(5)` string `'true'` if the parameter is `true` and returns the `varchar2(5)` string `'false'` if the parameter is `false`.

After this stored function, include the following testing calls:

```
prompt
prompt ****************************************
prompt TESTING is_on_order and bool_to_string
prompt ****************************************
prompt

prompt test passes if 0805343024 is shown on-order (returns true)
prompt ======================================================
var on_order_status varchar2(5);
exec :on_order_status := bool_to_string(is_on_order('0805343024'))
print on_order_status

prompt
prompt test passes if 087150331X is shown NOT on-order (rets false)
prompt ======================================================
exec :on_order_status := bool_to_string(is_on_order('087150331X'))
print on_order_status

prompt
prompt test passes if 1313131313 is shown NOT on-order (rets false)
prompt ======================================================
exec :on_order_status := bool_to_string(is_on_order('1313131313'))
print on_order_status
```

Debug and correct `is_on_order` and `bool_to_string` as necessary until the above tests succeed and you are convinced that it is working. You may add additional testing calls if you would like.

FINALLY, turn spooling off. The resulting `318hw1.sql` and `318hw1-out.txt` files should now be ready to submit. (Although I have no problem with you submitting incomplete versions of these early on if, for example, you want to submit after you have completed each problem... 8-))