Additional Practices

Additional Practices: Overview

These additional practices are provided as a supplement to the course *Oracle Database 10g: Develop PL/SQL Program Units*. In these practices, you apply the concepts that you learned in the course.

The additional practices comprise two parts:

Part A provides supplemental exercises to create stored procedures, functions, packages, and triggers, and to use the Oracle-supplied packages with SQL Developer as the development environment. The tables used in this portion of the additional practice include `EMPLOYEES`, `JOBS`, `JOB_HISTORY`, and `DEPARTMENTS`.

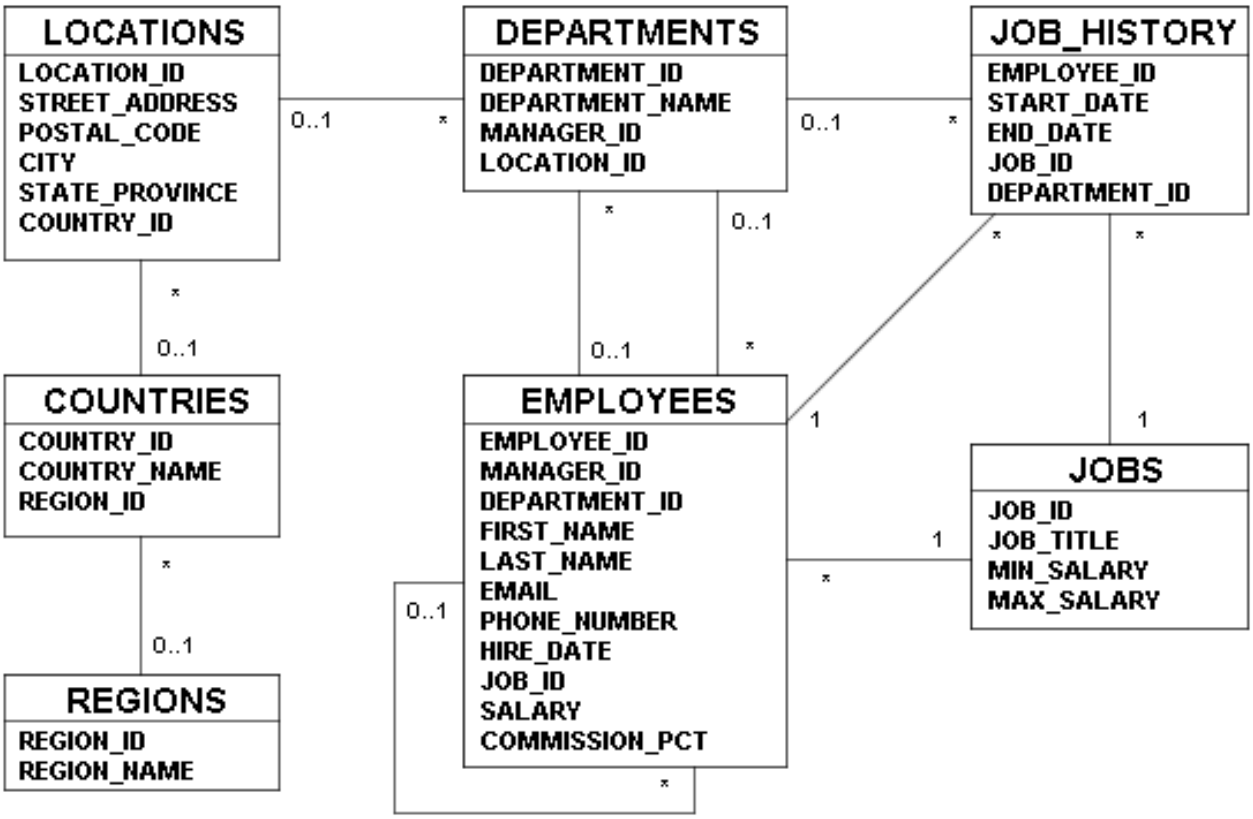
Part B is a case study that can be completed at the end of the course. This part supplements the practices for creating and managing program units. The tables used in the case study are based on a video database and contain the `TITLE`, `TITLE_COPY`, `RENTAL`, `RESERVATION`, and `MEMBER` tables.

An entity relationship diagram is provided at the start of part A and part B. Each entity relationship diagram displays the table entities and their relationships. More detailed definitions of the tables and the data contained in them is provided in the appendix titled "Additional Practices: Table Descriptions and Data."

Part A

Entity Relationship Diagram

Human Resources:



Part A (continued)

Note: These exercises can be used for extra practice when discussing how to create procedures.

1. In this exercise, create a program to add a new job into the JOBS table.
   a. Create a stored procedure called NEW_JOB to enter a new job into the JOBS table. The procedure should accept three parameters. The first and second parameters supply a job ID and a job title. The third parameter supplies the minimum salary. Use the maximum salary for the new job as twice the minimum salary supplied for the job ID.
   b. Invoke the procedure to add a new job with job ID 'SY_ANAL', job title 'System Analyst', and minimum salary of 6000.
   c. Check whether a row was added and note the new job ID for use in the next exercise. Commit the changes.
2. In this exercise, create a program to add a new row to the JOB_HISTORY table for an existing employee.
   a. Create a stored procedure called ADD_JOB_HIST to add a new row into the JOB_HISTORY table for an employee who is changing his job to the new job ID ('SY_ANAL') that you created in exercise 1b.
      The procedure should provide two parameters: one for the employee ID who is changing the job and the second for the new job ID. Read the employee ID from the EMPLOYEES table and insert it into the JOB_HISTORY table. Make the hire date of this employee as the start date and today's date as the end date for this row in the JOB_HISTORY table.
      Change the hire date of this employee in the EMPLOYEES table to today's date. Update the job ID of this employee to the job ID passed as parameter (use the 'SY_ANAL' job ID) and salary equal to the minimum salary for that job ID plus 500.
      Note: Include exception handling to handle an attempt to insert a nonexistent employee.
   b. Disable all triggers on the EMPLOYEES, JOBS, and JOB_HISTORY tables before invoking the ADD_JOB_HIST procedure.
   c. Execute the procedure with employee ID 106 and job ID 'SY_ANAL' as parameters.
   d. Query the JOB_HISTORY and EMPLOYEES tables to view your changes for employee 106, and then commit the changes.
   e. Reenable the triggers on the EMPLOYEES, JOBS, and JOB_HISTORY tables.
3. In this exercise, create a program to update the minimum and maximum salaries for a job in the JOBS table.
   a. Create a stored procedure called UPD_JOBSAL to update the minimum and maximum salaries for a specific job ID in the JOBS table. The procedure should provide three parameters: the job ID, a new minimum salary, and a new maximum salary. Add exception handling to account for an invalid job ID in the JOBS table. Raise an exception if the maximum salary supplied is less than the minimum salary, and provide a message that will be displayed if the row in the JOBS table is locked.
      Hint: The resource locked/busy error number is –54.

Part A (continued)

b. Execute the UPD_JOBSAL procedure by using a job ID of 'SY_ANAL', a minimum salary of 7000, and a maximum salary of 140.
Note: This should generate an exception message.
c. Disable triggers on the EMPLOYEES and JOBS tables.
d. Execute the UPD_JOBSAL procedure using a job ID of 'SY_ANAL', a minimum salary of 7000, and a maximum salary of 14000.
e. Query the JOBS table to view your changes, and then commit the changes.
f. Enable the triggers on the EMPLOYEES and JOBS tables.

4. In this exercise, create a procedure to monitor whether employees have exceeded their average salaries for their job type.
   a. Disable the SECURE_EMPLOYEES trigger.
   b. In the EMPLOYEES table, add an EXCEED_AVGSAL column to store up to three characters and a default value of NO. Use a check constraint to allow the values YES or NO.
   c. Write a stored procedure called CHECK_AVGSAL, which checks whether each employee's salary exceeds the average salary for the JOB_ID. The average salary for a job is calculated from the information in the JOBS table. If the employee's salary exceeds the average for his or her job, update his or her EXCEED_AVGSAL column in the EMPLOYEES table to a value of YES; otherwise, set the value to NO. Use a cursor to select the employee's rows using the FOR UPDATE option in the query. Add exception handling to account for a record being locked.
   Hint: The resource locked/busy error number is -54. Write and use a local function called GET_JOB_AVGSAL to determine the average salary for a job ID specified as a parameter.
   d. Execute the CHECK_AVGSAL procedure. Then, to view the results of your modifications, write a query to display the employee's ID, job, the average salary for the job, the employee's salary, and the exceed_avgsal indicator column for employees whose salaries exceed the average for their job, and finally commit the changes.

Note: These exercises can be used for extra practice when discussing how to create functions.

5. Create a subprogram to retrieve the number of years of service for a specific employee.
   a. Create a stored function called GET_YEARS_SERVICE to retrieve the total number of years of service for a specific employee. The function should accept the employee ID as a parameter and return the number of years of service. Add error handling to account for an invalid employee ID.
   b. Invoke the GET_YEARS_SERVICE function in a call to DBMS_OUTPUT.PUT_LINE for an employee with ID 999.
   c. Display the number of years of service for employee 106 with DBMS_OUTPUT.PUT_LINE invoking the GET_YEARS_SERVICE function.
   d. Query the JOB_HISTORY and EMPLOYEES tables for the specified employee to verify that the modifications are accurate. The values represented in the results on this page may differ from those that you get when you run these queries.

Part A (continued)

6. In this exercise, create a program to retrieve the number of different jobs that an employee worked on during his or her service.
   a. Create a stored function called `GET_JOB_COUNT` to retrieve the total number of different jobs on which an employee worked.
      The function should accept the employee ID in a parameter, and return the number of different jobs that the employee worked on until now, including the present job. Add exception handling to account for an invalid employee ID.
      Hint: Use the distinct job IDs from the `JOB_HISTORY` table, and exclude the current job ID, if it is one of the job IDs on which the employee has already worked. Write a `UNION` of two queries and count the rows retrieved into a PL/SQL table. Use a `FETCH` with `BULK COLLECT INTO` to obtain the unique jobs for the employee.
   b. Invoke the function for the employee with the ID of `176`.

Note: These exercises can be used for extra practice when discussing how to create packages.

7. Create a package called `EMPJOB_PKG` that contains your `NEW_JOB`, `ADD_JOB_HIST`, and `UPD_JOBSAL` procedures, as well as your `GET_YEARS_SERVICE` and `GET_JOB_COUNT` functions.
   a. Create the package specification with all the subprogram constructs as public. Move any subprogram local-defined types into the package specification.
   b. Create the package body with the subprogram implementation; remember to remove, from the subprogram implementations, any types that you moved into the package specification.
   c. Invoke your `EMPJOB_PKG.NEW_JOB` procedure to create a new job with the ID `PR_MAN`, the job title `Public Relations Manager`, and the salary `6,250`.
   d. Invoke your `EMPJOB_PKG.ADD_JOB_HIST` procedure to modify the job of employee ID `110` to job ID `PR_MAN`.
      Note: You need to disable the `UPDATE_JOB_HISTORY` trigger before you execute the `ADD_JOB_HIST` procedure, and reenable the trigger after you have executed the procedure.
   e. Query the `JOBS`, `JOB_HISTORY`, and `EMPLOYEES` tables to verify the results.

Note: These exercises can be used for extra practice when discussing how to create database triggers.

8. In this exercise, create a trigger to ensure that the minimum and maximum salaries of a job are never modified such that the salary of an existing employee with that job ID is out of the new range specified for the job.
   a. Create a trigger called `CHECK_SAL_RANGE` that is fired before every row that is updated in the `MIN_SALARY` and `MAX_SALARY` columns in the `JOBS` table. For any minimum or maximum salary value that is changed, check whether the salary of any existing employee with that job ID in the `EMPLOYEES` table falls within the new range of salaries specified for this job ID. Include exception handling to cover a salary range change that affects the record of any existing employee.

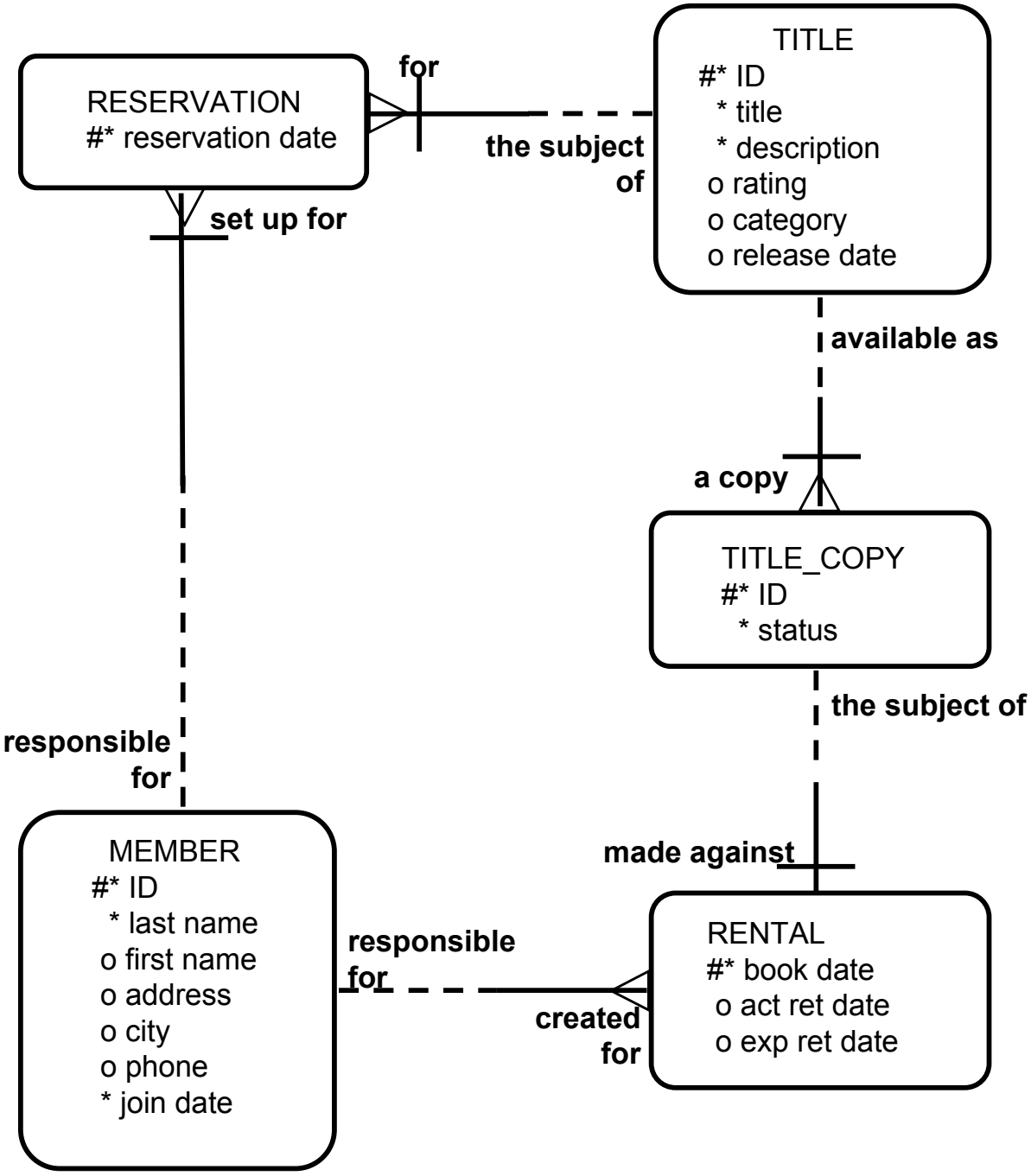
Part A (continued)

b. Test the trigger using the `SY_ANAL` job, setting the new minimum salary to `5,000`, and the new maximum salary to `7,000`. Before you make the change, write a query to display the current salary range for the `SY_ANAL` job ID, and another query to display the employee ID, last name, and salary for the same job ID. After the update, query the change (if any) to the `JOBS` table for the specified job ID.

c. Using the `SY_ANAL` job, set the new minimum salary to `7,000`, and the new maximum salary to `18,000`. Explain the results.

Part B

Entity Relationship Diagram



Part B (continued)

In this case study, you create a package named VIDEO_PKG that contains procedures and functions for a video store application. This application enables customers to become a member of the video store. Any member can rent movies, return rented movies, and reserve movies. Additionally, you create a trigger to ensure that any data in the video tables is modified only during business hours.

Create the package by using SQL Developer and use the DBMS_OUTPUT Oracle-supplied package to display messages.

The video store database contains the following tables: TITLE, TITLE_COPY, RENTAL, RESERVATION, and MEMBER. The entity relationship diagram is shown on the previous page.

Part B (continued)

1. Load and execute the `/home/oracle/labs/PLPU/labs/buildvid1.sql` script to create all the required tables and sequences that are needed for this exercise.
2. Load and execute the `/home/oracle/labs/PLPU/labs/buildvid2.sql` script to populate all the tables created through the `buildvid1.sql` script.
3. Create a package named `VIDEO_PKG` with the following procedures and functions:
   a. **`NEW_MEMBER`:** A public procedure that adds a new member to the `MEMBER` table. For the member `ID` number, use the sequence `MEMBER_ID_SEQ`; for the join date, use `SYSDATE`. Pass all other values to be inserted into a new row as parameters.
   b. **`NEW_RENTAL`:** An overloaded public function to record a new rental. Pass the title `ID` number for the video that a customer wants to rent, and either the customer's last name or his member ID number into the function. The function should return the due date for the video. Due dates are three days from the date the video is rented. If the status for a movie requested is listed as `AVAILABLE` in the `TITLE_COPY` table for one copy of this title, update this `TITLE_COPY` table and set the status to `RENTED`. If there is no copy available, the function must return `NULL`. Then insert a new record into the `RENTAL` table identifying the booked date as today's date, the copy `ID` number, the member `ID` number, the title `ID` number, and the expected return date. Be aware of multiple customers with the same last name. In this case, have the function return `NULL` and display a list of the customers' names that match and their `ID` numbers.
   c. **`RETURN_MOVIE`:** A public procedure that updates the status of a video (available, rented, or damaged) and sets the return date. Pass the title `ID`, the copy `ID`, and the status to this procedure. Check whether there are reservations for that title and display a message if it is reserved. Update the `RENTAL` table and set the actual return date to today's date. Update the status in the `TITLE_COPY` table based on the status parameter passed into the procedure.
   d. **`RESERVE_MOVIE`:** A private procedure that executes only if all the video copies requested in the `NEW_RENTAL` procedure have a status of `RENTED`. Pass the member `ID` number and the title `ID` number to this procedure. Insert a new record into the `RESERVATION` table and record the reservation date, member `ID` number, and title `ID` number. Print a message indicating that a movie is reserved and its expected date of return.
   e. **`EXCEPTION_HANDLER`:** A private procedure that is called from the exception handler of the public programs. Pass the `SQLCODE` number to this procedure and the name of the program (as a text string) where the error occurred. Use `RAISE_APPLICATION_ERROR` to raise a customized error. Start with a unique key violation (`-1`) and foreign key violation (`-2292`). Allow the exception handler to raise a generic error for any other errors.

Part B (continued)

4. Use the following scripts located in the `/home/oracle/labs/PLPU/soln` directory to test your routines:
   a. Add two members using `sol_apb_04_a_new_members.sql`.
   b. Add new video rentals using `sol_apb_04_b_new_rentals.sql`.
   c. Return movies using the `sol_apb_04_c_return_movie.sql` script.
5. The business hours for the video store are 8:00 AM to 10:00 PM, Sunday through Friday, and 8:00 AM to 12:00 AM on Saturday. To ensure that the tables can be modified only during these hours, create a stored procedure that is called by triggers on the tables.
   a. Create a stored procedure called `TIME_CHECK` that checks the current time against business hours. If the current time is not within business hours, use the `RAISE_APPLICATION_ERROR` procedure to give an appropriate message.
   b. Create a trigger on each of the five tables. Fire the trigger before data is inserted, updated, and deleted from the tables. Call your `TIME_CHECK` procedure from each of these triggers.
   c. Test your triggers.