Oracle Database 11*g*: Develop PL/SQL Program Units

Student Guide • Additional Practices

D49986GC10
Edition 1.0
August 2007
D52204

ORACLE®

Author

Lauran K. Serhal

Technical Contributors and Reviewers

Don Bates
Claire Bennett
Zarko Cesljas
Purjanti Chang
Ashita Dhir
Peter Driver
Gerlinde Frenzen
Steve Friedberg
Nancy Greenberg
Thomas Hoogerwerf
Akira Kinutani
Chaitanya Koratamaddi
Timothy Leblanc
Bryn Llewellyn
Lakshmi Narapareddi
Essi Parast
Alan Paulson
Alan Paulson
Manish Pawar
Srinivas Putrevu
Bryan Roberts
Grant Spencer
Tulika Srivastava
Glenn Stokol
Jenny Tsai-Smith
Lex Van Der Werff
Ted Witiuk

Graphic Designer

Rajiv Chandrabhanu

Editors

Arijit Ghosh
Atanu Raychaudhuri

Publishers

Nita Brozowski
Jobi Varghese
Giri Venugopal

Contents

Preface

Profile

Before You Begin This Course

Before you begin this course, you should have thorough knowledge of SQL and SQL*Developer or SQL*Plus, as well as working experience in developing applications.

Prerequisites

Prerequisites are any of the following Oracle University courses or combinations of courses:

- *Oracle Database 11g: PL/SQL Fundamentals*
- *Oracle Database 11g: Introduction to SQL*
- *Oracle Database 11g: SQL Fundamentals I* and *Oracle Database 11g: SQL Fundamentals II*
- *Oracle Database 11g: SQL and PL/SQL Fundamentals*

How This Course Is Organized

Oracle Database 11g: Develop PL/SQL Program Units is an instructor-led course featuring lectures and hands-on exercises. Online demonstrations and practice sessions reinforce the concepts and skills that are introduced.

Related Publications

Oracle Publications

Title	Part Number
Oracle® Database Reference 11g Release 1 (11.1)	B28320-01
Oracle® Database SQL Language Reference 11g Release 1 (11.1)	B28286-01
Oracle® Database Concepts 11*g* Release *1 (11.1)*	B28318-01
Oracle® Database Advanced Application Developer's Guide – 11g Release 1 (11.1)	B28424-01
*SQL*Plus® User's Guide and Reference Release 11.1*	B31189-01
Oracle Database SQL Developer User's Guide Release 1.2	B10406-01
Oracle® Database PL/SQL Language Reference 11g Release 1 (11.1)	B28370-01
Oracle® Database PL/SQL Packages and Types Reference 11g Release 1 (11.1)	B28419-01

Additional Publications

- System release bulletins
- Installation and user's guides
- Read-me files
- International Oracle User's Group (IOUG) articles
- *Oracle Magazine*

Typographic Conventions

Typographic Conventions In Text

Convention	**Element**	**Example**
Bold	Emphasized words and phrases in Web content only	To navigate within this application, do **not** click the Back and Forward buttons.
Bold italic	Glossary terms (if there is a glossary)	The ***algorithm*** inserts the new key.
Brackets	Key names	Press [Enter].
Caps and lowercase	Buttons, check boxes, triggers, windows	Click the Executable button. Select the Registration Required check box. Assign a When-Validate-Item trigger. Open the Master Schedule window.
Carets	Menu paths	Select File > Save.
Commas	Key sequences	Press and release these keys one at a time: [Alt], [F], [D]

Typographic Conventions (continued)

Typographic Conventions In Text (continued)

Convention	Object or Term	Example
Courier New, case sensitive	Code output, SQL and PL/SQL code elements, Java code elements, directory names, filenames, passwords, pathnames, URLs, user input, usernames	Code output: `debug.seti('I',300);` SQL code elements: Use the `SELECT` command to view information stored in the `last_name` column of the `emp` table. Java code elements: Java programming involves the `String` and `StringBuffer` classes. Directory names: `bin` (DOS), `$FMHOME` (UNIX) Filenames: Locate the `init.ora` file. Passwords: Use `tiger` as your password. Pathnames: Open `c:\my_docs\projects`. URLs: Go to `http://www.oracle.com`. User input: Enter `300`. Usernames: Log on as `scott`.
Initial cap	Graphics labels (unless the term is a proper noun)	Customer address (*but* Oracle Payables)
Italic	Emphasized words and phrases in print publications, titles of books and courses, variables	Do *not* save changes to the database. For further information, see *Oracle7 Server SQL Language Reference Manual*. Enter *`user_id@us.oracle.com`*, where *user_id* is the name of the user.
Plus signs	Key combinations	Press and hold these keys simultaneously: [Control] + [Alt] + [Delete]
Quotation marks	Lesson and chapter titles in cross references, interface elements with long names that have only initial caps	This subject is covered in Unit II, Lesson 3, “Working with Objects.” Select the “Include a reusable module component” and click Finish. Use the “`WHERE` clause of query” property.

Typographic Conventions (continued)

Typographic Conventions in Navigation Paths

This course uses simplified navigation paths, such as the following example, to direct you through Oracle Applications.

Example:

Invoice Batch Summary

(N) Invoice > Entry > Invoice Batches Summary (M) Query > Find (B) Approve

This simplified path translates to the following:

1. (N) From the Navigator window, select Invoice > Entry > Invoice Batches Summary.
2. (M) From the menu, select Query > Find.
3. (B) Click the Approve button.

Notation:

(N) = Navigator	(I) = Icon
(M) = Menu	(H) = Hyperlink
(T) = Tab	(B) = Button

Additional Practices

Additional Practices: Overview

These additional practices are provided as a supplement to the course *Oracle Database 11g: 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 or SQL*Plus as the development environment. The tables used in this portion of the additional practice include `EMPLOYEES`, `JOBS`, `JOB_HISTORY`, and `DEPARTMENTS`.

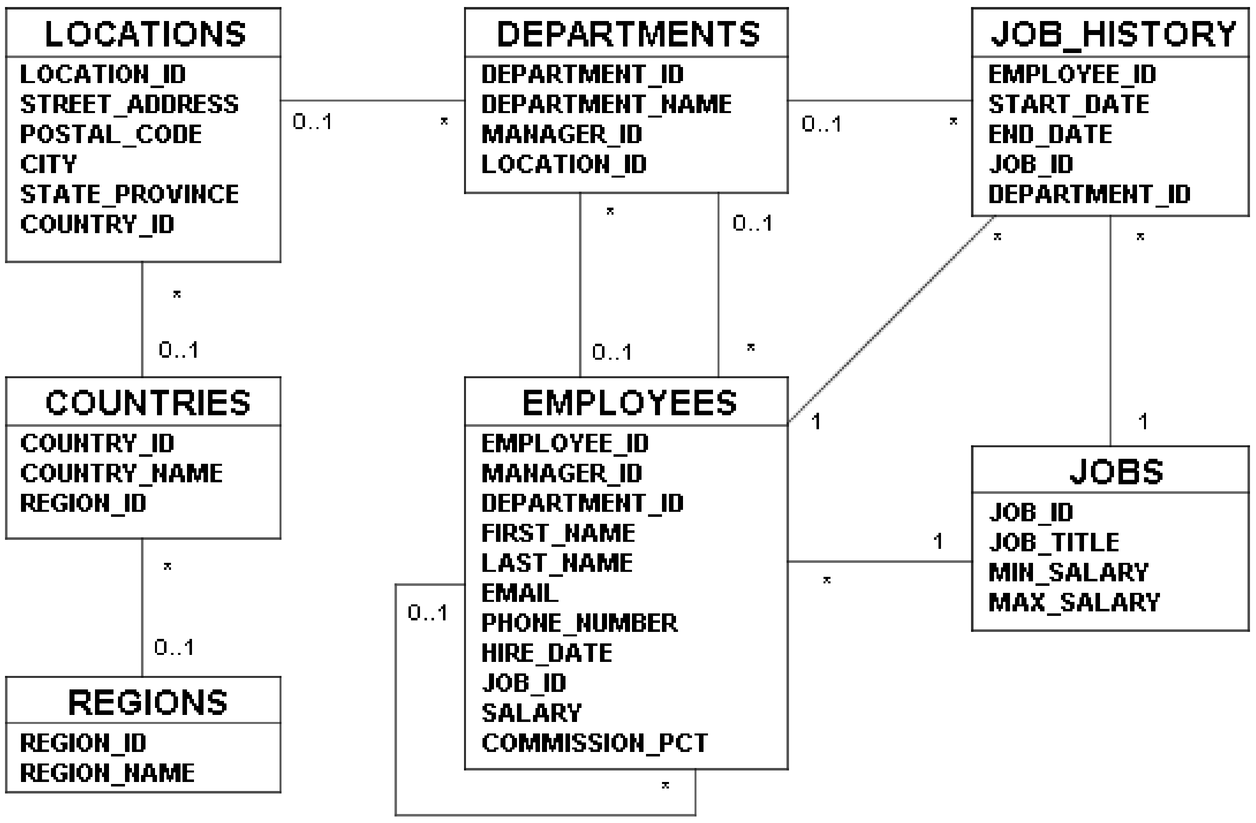
Part B is a case study that can be completed at the end of the course. This part supplements the practices for creating and managing program units. The tables used in the case study are based on a video database and contain the `TITLE`, `TITLE_COPY`, `RENTAL`, `RESERVATION`, and `MEMBER` tables.

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 order 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 1 b.
      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 start date and today's date as 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 + 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 that 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, then update the 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 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`, `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 `6250`.
   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 re-enable 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.

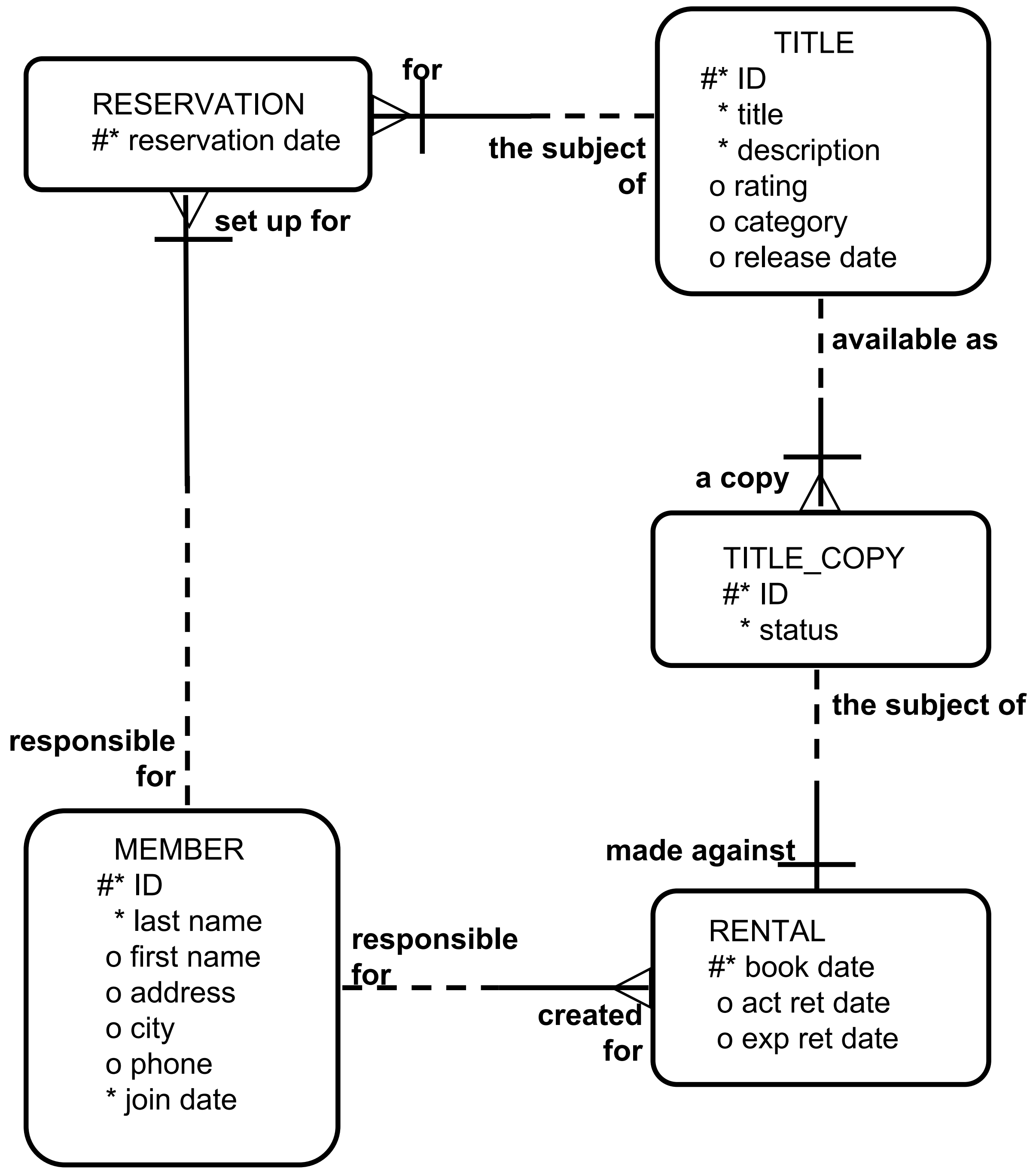
8. In this exercise, create a trigger to ensure that the minimum and maximum salaries of a job are never modified such that the salary of an existing employee with that job ID is out of the new range specified for the job.
   a. Create a trigger called `CHECK_SAL_RANGE` that is fired before every row 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 `5000`, and the new maximum salary to `7000`. 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 `18000`. 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*Plus 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 `D:\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 `D:\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, then 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 `E:\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 through 10:00 AM, Sunday through Friday, and 8:00 AM through 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.

Additional Practice: Solutions

Part A: Additional Practice 1 Solutions

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 order 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.

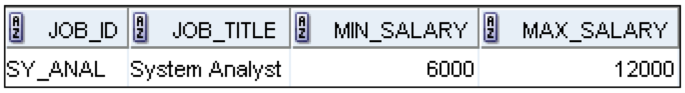
```
CREATE OR REPLACE PROCEDURE new_job(
  p_jobid  IN jobs.job_id%TYPE,
  p_title  IN jobs.job_title%TYPE,
  v_minsal IN jobs.min_salary%TYPE) IS
  v_maxsal  jobs.max_salary%TYPE := 2 * v_minsal;
BEGIN
  INSERT INTO jobs(job_id, job_title, min_salary, max_salary)
  VALUES (p_jobid, p_title, v_minsal, v_maxsal);
  DBMS_OUTPUT.PUT_LINE ('New row added to JOBS table:');
  DBMS_OUTPUT.PUT_LINE (p_jobid || '  ' || p_title ||' '||
                        v_minsal || '  ' || v_maxsal);
END new_job;
/
SHOW ERRORS

PROCEDURE new_job Compiled.
No Errors.
```

   b. Invoke the procedure to add a new job with job ID 'SY_ANAL', job title 'System Analyst', and minimum salary 6,000.

```
EXECUTE new_job ('SY_ANAL', 'System Analyst', 6000)

anonymous block completed
New row added to JOBS table:
SY_ANAL System Analyst 6000 12000
```

   c. Verify that a row was added, and note the new job ID for use in the next exercise. Commit the changes.

```
SELECT *
FROM   jobs
WHERE  job_id = 'SY_ANAL';
```



JOB_ID	JOB_TITLE	MIN_SALARY	MAX_SALARY
SY_ANAL	System Analyst	6000	12000

```
COMMIT;

Commit complete.
```

Part A: Additional Practice 2 Solutions

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.

```
CREATE OR REPLACE PROCEDURE add_job_hist(
  p_emp_id    IN employees.employee_id%TYPE,
  p_new_jobid IN jobs.job_id%TYPE) IS
BEGIN
  INSERT INTO job_history
    SELECT employee_id, hire_date, SYSDATE, job_id, department_id
    FROM   employees
    WHERE  employee_id = p_emp_id;
  UPDATE employees
    SET  hire_date = SYSDATE,
         job_id = p_new_jobid,
         salary = (SELECT min_salary + 500
                   FROM   jobs
                   WHERE  job_id = p_new_jobid)
   WHERE employee_id = p_emp_id;
  DBMS_OUTPUT.PUT_LINE ('Added employee ' || p_emp_id ||
                        ' details to the JOB_HISTORY table');
  DBMS_OUTPUT.PUT_LINE ('Updated current job of employee ' ||
                        p_emp_id|| ' to '|| p_new_jobid);
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    RAISE_APPLICATION_ERROR (-20001, 'Employee does not exist!');
END add_job_hist;
/
SHOW ERRORS

PROCEDURE add_job_hist( Compiled.
No Errors.
```

Part A: Additional Practice 2 Solutions (continued)

b. Disable all triggers on the EMPLOYEES, JOBS, and JOB_HISTORY tables before invoking the ADD_JOB_HIST procedure.

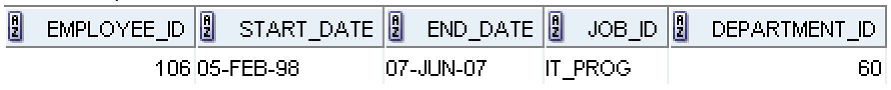
```
ALTER TABLE employees DISABLE ALL TRIGGERS;
ALTER TABLE jobs DISABLE ALL TRIGGERS;
ALTER TABLE job_history DISABLE ALL TRIGGERS;

ALTER TABLE employees succeeded.
ALTER TABLE jobs succeeded.
ALTER TABLE job_history succeeded.
```

c. Execute the procedure with employee ID 106 and job ID 'SY_ANAL' as parameters.

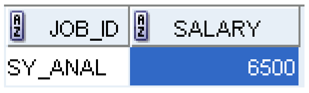
```
EXECUTE add_job_hist(106, 'SY_ANAL')

anonymous block completed
Added employee 106 details to the JOB_HISTORY table
Updated current job of employee 106 to SY_ANAL
```

d. Query the JOB_HISTORY and EMPLOYEES tables to view your changes for employee 106, and then commit the changes.

```
SELECT *   FROM   job_history
WHERE  employee_id = 106;

SELECT job_id, salary   FROM   employees
WHERE  employee_id = 106;

COMMIT;
```



EMPLOYEE_ID	START_DATE	END_DATE	JOB_ID	DEPARTMENT_ID
106	05-FEB-98	07-JUN-07	IT_PROG	60



JOB_ID	SALARY
SY_ANAL	6500

```
Commit complete.
```

e. Reenable the triggers on the EMPLOYEES, JOBS, and JOB_HISTORY tables.

```
ALTER TABLE employees ENABLE ALL TRIGGERS;
ALTER TABLE jobs ENABLE ALL TRIGGERS;
ALTER TABLE job_history ENABLE ALL TRIGGERS;

ALTER TABLE employees succeeded.
ALTER TABLE jobs succeeded.
ALTER TABLE job_history succeeded.
```

Part A: Additional Practice 3 Solutions

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. 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`.

```
CREATE OR REPLACE PROCEDURE upd_jobsal(
  p_jobid       IN jobs.job_id%type,
  p_new_minsal  IN jobs.min_salary%type,
  p_new_maxsal  IN jobs.max_salary%type) IS
  v_dummy           PLS_INTEGER;
  e_resource_busy  EXCEPTION;
  e_sal_error        EXCEPTION;
  PRAGMA           EXCEPTION_INIT (e_resource_busy , -54);
BEGIN
  IF (p_new_maxsal < p_new_minsal) THEN
    RAISE e_sal_error;
  END IF;
  SELECT 1 INTO v_dummy
    FROM jobs
    WHERE job_id = p_jobid
    FOR UPDATE OF min_salary NOWAIT;
  UPDATE jobs
    SET min_salary =  p_new_minsal,
        max_salary =  p_new_maxsal
    WHERE job_id  = p_jobid;
EXCEPTION
  WHEN e_resource_busy THEN
    RAISE_APPLICATION_ERROR (-20001,
      'Job information is currently locked, try later.');
  WHEN NO_DATA_FOUND THEN
    RAISE_APPLICATION_ERROR(-20001, 'This job ID does not exist');
  WHEN e_sal_error THEN
    RAISE_APPLICATION_ERROR(-20001,
     'Data error: Max salary should be more than min salary');
END upd_jobsal;
/
SHOW ERRORS

PROCEDURE upd_jobsal( Compiled.
No Errors.
```

Part A: Additional Practice 3 Solutions (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.

```
EXECUTE upd_jobsal('SY_ANAL', 7000, 140)

BEGIN upd_jobsal('SY_ANAL', 7000, 140); END;

*

ERROR at line 1:
ORA-20001: Data error: Max salary should be more than min salary
ORA-06512: at "ORA1.UPD_JOBSAL", line 28
ORA-06512: at line 1
```

c. Disable triggers on the `EMPLOYEES` and `JOBS` tables.

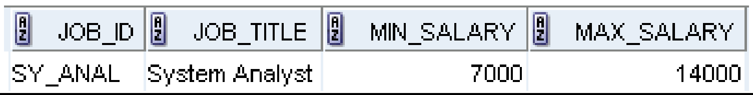
```
ALTER TABLE employees DISABLE ALL TRIGGERS;
ALTER TABLE jobs DISABLE ALL TRIGGERS;

ALTER TABLE employees succeeded.
ALTER TABLE jobs succeeded.
```

d. Execute the `UPD_JOBSAL` procedure using a job ID of `'SY_ANAL'`, a minimum salary of `7000`, and a maximum salary of `14000`.

```
EXECUTE upd_jobsal('SY_ANAL', 7000, 14000)

anonymous block completed.
```

e. Query the `JOBS` table to view your changes, and then commit the changes.

```
SELECT *
FROM  jobs
WHERE job_id = 'SY_ANAL';
```



JOB_ID	JOB_TITLE	MIN_SALARY	MAX_SALARY
SY_ANAL	System Analyst	7000	14000

f. Enable the triggers on the `EMPLOYEES` and `JOBS` tables.

```
ALTER TABLE employees ENABLE ALL TRIGGERS;
ALTER TABLE jobs ENABLE ALL TRIGGERS;

ALTER TABLE employees succeeded.
ALTER TABLE jobs succeeded.
```

Part A: Additional Practice 4 Solutions

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.

```
ALTER TRIGGER secure_employees DISABLE;

ALTER TRIGGER secure_employees succeeded.
```

   b. In the EMPLOYEES table, add an EXCEED_AVGSAL column for storing up to three characters and a default value of NO. Use a check constraint to allow the values YES or NO.

```
ALTER TABLE employees (
  ADD (exceed_avgsal VARCHAR2(3) DEFAULT 'NO'
    CONSTRAINT employees_exceed_avgsal_ck
    CHECK (exceed_avgsal IN ('YES', 'NO')));

ALTER TABLE employees succeeded.
```

   c. Write a stored procedure called CHECK_AVGSAL that checks whether each employee's salary exceeds the average salary for the JOB_ID. The average salary for a job is calculated from information in the JOBS table. If the employee's salary exceeds the average for his or her job, then 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.

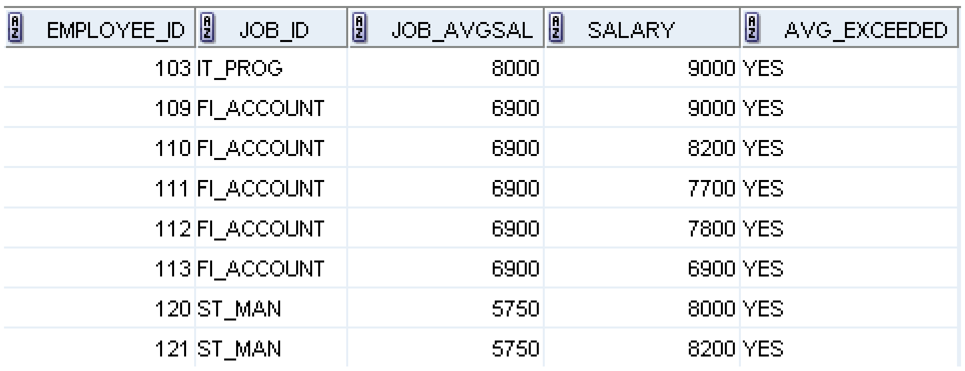
```
CREATE OR REPLACE PROCEDURE check_avgsal IS
  emp_exceed_avgsal_type employees.exceed_avgsal%type;
  CURSOR  c_emp_csr IS
    SELECT  employee_id, job_id, salary
    FROM employees
    FOR UPDATE;
  e_resource_busy  EXCEPTION;
  PRAGMA EXCEPTION_INIT(e_resource_busy, -54);
  FUNCTION get_job_avgsal (jobid VARCHAR2) RETURN NUMBER IS
    avg_sal employees.salary%type;
  BEGIN
    SELECT (max_salary + min_salary)/2 INTO avg_sal
    FROM jobs
    WHERE job_id = jobid;
    RETURN avg_sal;
  END;

BEGIN
  FOR emprec IN c_emp_csr
  LOOP
    emp_exceed_avgsal_type := 'NO';
    IF emprec.salary >= get_job_avgsal(emprec.job_id) THEN
      emp_exceed_avgsal_type := 'YES';
    END IF;
    UPDATE employees
      SET exceed_avgsal = emp_exceed_avgsal_type
      WHERE CURRENT OF c_emp_csr;
  END LOOP;
EXCEPTION
  WHEN e_resource_busy THEN
    ROLLBACK;
    RAISE_APPLICATION_ERROR (-20001, 'Record is busy, try later.');
END check_avgsal;
/
SHOW ERRORS

PROCEDURE check_avgsal Compiled.
No Errors.
```

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.

```
EXECUTE check_avgsal

SELECT e.employee_id, e.job_id, (j.max_salary-j.min_salary/2) job_avgsal,
       e.salary, e.exceed_avgsal avg_exceeded
FROM   employees e, jobs j
WHERE  e.job_id = j.job_id
and e.exceed_avgsal = 'YES';

COMMIT;
```

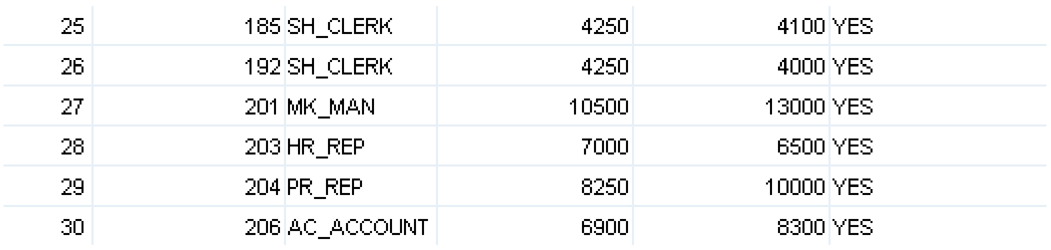
Part A: Additional Practice 4 Solutions (continued)

```
anonymous block completed.
```



EMPLOYEE_ID	JOB_ID	JOB_AVGSAL	SALARY	AVG_EXCEEDED
103	IT_PROG	8000	9000	YES
109	FI_ACCOUNT	6900	9000	YES
110	FI_ACCOUNT	6900	8200	YES
111	FI_ACCOUNT	6900	7700	YES
112	FI_ACCOUNT	6900	7800	YES
113	FI_ACCOUNT	6900	6900	YES
120	ST_MAN	5750	8000	YES
121	ST_MAN	5750	8200	YES

. . .



25	185	SH_CLERK	4250	4100	YES
26	192	SH_CLERK	4250	4000	YES
27	201	MK_MAN	10500	13000	YES
28	203	HR_REP	7000	6500	YES
29	204	PR_REP	8250	10000	YES
30	206	AC_ACCOUNT	6900	8300	YES

```
Commit complete.
```

Part A: Additional Practice 5 Solutions

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.

```
CREATE OR REPLACE FUNCTION get_years_service(
  p_emp_empid_type IN employees.employee_id%TYPE) RETURN NUMBER IS
  CURSOR c_jobh_csr IS
    SELECT MONTHS_BETWEEN(end_date, start_date)/12 v_years_in_job
    FROM   job_history
    WHERE  employee_id = p_emp_empid_type;
  v_years_service NUMBER(2) := 0;
  v_years_in_job  NUMBER(2) := 0;
BEGIN
  FOR jobh_rec IN c_jobh_csr
  LOOP
    EXIT WHEN c_jobh_csr%NOTFOUND;
    v_years_service := v_years_service + jobh_rec.v_years_in_job;
  END LOOP;
  SELECT MONTHS_BETWEEN(SYSDATE, hire_date)/12 INTO v_years_in_job
  FROM   employees
  WHERE  employee_id = p_emp_empid_type;
  v_years_service := v_years_service + v_years_in_job;
  RETURN ROUND(v_years_service);
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    RAISE_APPLICATION_ERROR(-20348,
      'Employee with ID '|| p_emp_empid_type ||' does not exist.');
    RETURN NULL;
END get_years_service;
/
SHOW ERRORS

FUNCTION get_years_service( Compiled.
No Errors.
```

   b. Invoke the `GET_YEARS_SERVICE` function in a call to `DBMS_OUTPUT.PUT_LINE` for an employee with ID `999`.

```
EXECUTE DBMS_OUTPUT.PUT_LINE(get_years_service (999))


Error starting at line 1 in command:
EXECUTE DBMS_OUTPUT.PUT_LINE(get_years_service (999))
Error report:
ORA-20348: Employee with ID 999 does not exist.
ORA-06512: at "ORA61.GET_YEARS_SERVICE", line 22
ORA-06512: at line 1
```

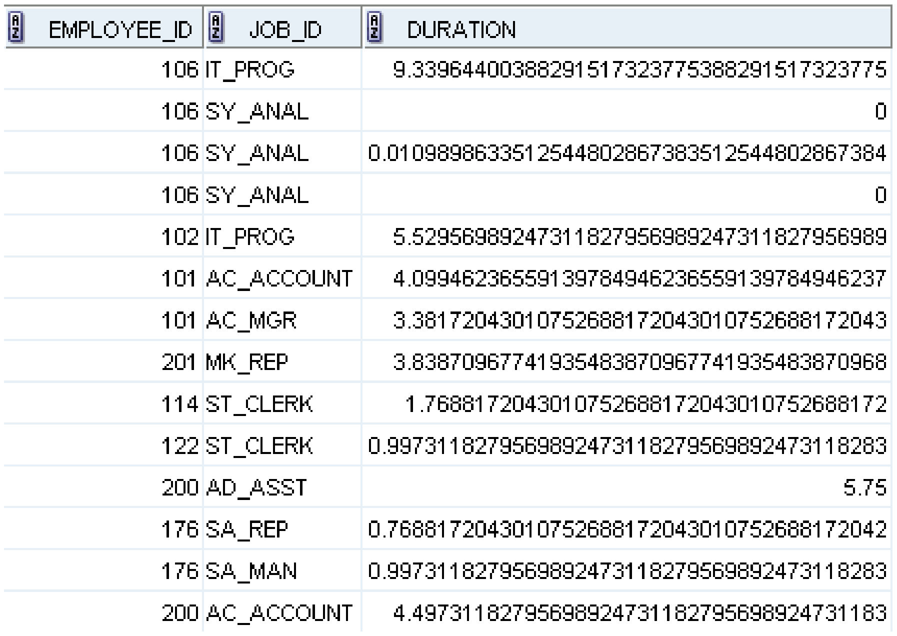
Part A: Additional Practice 5 Solutions (continued)

c. Display the number of years of service for employee 106 with `DBMS_OUTPUT.PUT_LINE` invoking the `GET_YEARS_SERVICE` function.

```
BEGIN
  DBMS_OUTPUT.PUT_LINE (
    'Employee 106 has worked ' || get_years_service(106) || ' years');
END;
/

anonymous block completed
Employee 106 has worked 9 years.
```

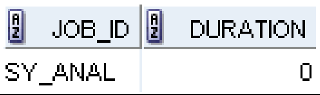
d. Query the `JOB_HISTORY` and `EMPLOYEES` tables for the specified employee to verify that the modifications are accurate.
Note: The values represented in the results on this page may differ from those you get when you run these queries.

```
SELECT employee_id, job_id,
       MONTHS_BETWEEN(end_date, start_date)/12 duration
FROM   job_history;
```



EMPLOYEE_ID	JOB_ID	DURATION
106	IT_PROG	9.33964400388291517323775388291517323775
106	SY_ANAL	0
106	SY_ANAL	0.0109898633512544802867383512544802867384
106	SY_ANAL	0
102	IT_PROG	5.52956989247311827956989247311827956989
101	AC_ACCOUNT	4.09946236559139784946236559139784946237
101	AC_MGR	3.38172043010752688172043010752688172043
201	MK_REP	3.83870967741935483870967741935483870968
114	ST_CLERK	1.768817204301075268817204301075268817 2
122	ST_CLERK	0.997311827956989247311827956989247311828 3
200	AD_ASST	5.75
176	SA_REP	0.7688172043010752688172043010752688172042
176	SA_MAN	0.997311827956989247311827956989247311828 3
200	AC_ACCOUNT	4.49731182795698924731182795698924731183

```
SELECT job_id, MONTHS_BETWEEN(SYSDATE, hire_date)/12 duration
FROM   employees
WHERE  employee_id = 106;
```



JOB_ID	DURATION
SY_ANAL	0

Part A: Additional Practice 6 Solutions

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.

```
CREATE OR REPLACE FUNCTION get_job_count(
  p_emp_empid_type IN employees.employee_id%TYPE) RETURN NUMBER IS
  TYPE jobs_table_type IS TABLE OF jobs.job_id%type;
  v_jobtab jobs_table_type;
  CURSOR c_empjob_csr IS
    SELECT job_id
    FROM job_history
    WHERE employee_id = p_emp_empid_type
      UNION
    SELECT job_id
    FROM employees
    WHERE employee_id = p_emp_empid_type;
BEGIN
  OPEN c_empjob_csr;
  FETCH c_empjob_csr BULK COLLECT INTO v_jobtab;
  CLOSE c_empjob_csr;
  RETURN v_jobtab.count;
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    RAISE_APPLICATION_ERROR(-20348,
      'Employee with ID '|| p_emp_empid_type ||' does not exist!');
    RETURN NULL;
END get_job_count;
/
SHOW ERRORS

FUNCTION get_job_count( Compiled.
No Errors.
```

Part A: Additional Practice 6 Solutions (continued)

b. Invoke the function for an employee with ID `176`.

```
BEGIN
  DBMS_OUTPUT.PUT_LINE('Employee 176 worked on ' ||
     get_job_count(176) || ' different jobs.');
END;
/

Employee 176 worked on 2 different jobs.
PL/SQL procedure successfully completed.
```

Part A: Additional Practice 7 Solutions

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 public. Move any subprogram local-defined types into the package specification.

```
CREATE OR REPLACE PACKAGE empjob_pkg IS
  TYPE jobs_table_type IS TABLE OF jobs.job_id%type;

  PROCEDURE add_job_hist(
     p_emp_id IN employees.employee_id%TYPE,
     p_new_jobid IN jobs.job_id%TYPE);

  FUNCTION get_job_count(
     p_emp_id IN employees.employee_id%TYPE) RETURN NUMBER;

  FUNCTION get_years_service(
     p_emp_id IN employees.employee_id%TYPE) RETURN NUMBER;

  PROCEDURE new_job(
    p_jobid IN jobs.job_id%TYPE,
    p_title IN jobs.job_title%TYPE,
    p_minsal IN jobs.min_salary%TYPE);

  PROCEDURE upd_jobsal(
    p_jobid IN jobs.job_id%type,
    p_new_minsal IN jobs.min_salary%type,
    p_new_maxsal IN jobs.max_salary%type);
END empjob_pkg;
/
SHOW ERRORS

PACKAGE empjob_pkg Compiled.
No Errors.
```

Part A: Additional Practice 7 Solutions (continued)

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.

```
CREATE OR REPLACE PACKAGE BODY empjob_pkg IS
  PROCEDURE add_job_hist(
    p_emp_id IN employees.employee_id%TYPE,
    p_new_jobid IN jobs.job_id%TYPE) IS
  BEGIN
    INSERT INTO job_history
      SELECT employee_id, hire_date, SYSDATE, job_id, department_id
      FROM employees
      WHERE employee_id = p_emp_id;
    UPDATE employees
      SET hire_date = SYSDATE,
          job_id = p_new_jobid,
          salary = (SELECT min_salary + 500
                    FROM jobs
          WHERE job_id = p_new_jobid)
      WHERE employee_id = p_emp_id;
    DBMS_OUTPUT.PUT_LINE ('Added employee ' || p_emp_id ||
       ' details to the JOB_HISTORY table');
    DBMS_OUTPUT.PUT_LINE ('Updated current job of employee ' ||
       p_emp_id|| ' to '|| p_new_jobid);
  EXCEPTION
    WHEN NO_DATA_FOUND THEN
      RAISE_APPLICATION_ERROR (-20001, 'Employee does not exist!');
  END add_job_hist;

  FUNCTION get_job_count(
    p_emp_id IN employees.employee_id%TYPE) RETURN NUMBER IS
    v_jobtab jobs_table_type;
    CURSOR c_empjob_csr IS
      SELECT job_id
      FROM job_history
      WHERE employee_id = p_emp_id
      UNION
      SELECT job_id
      FROM employees
      WHERE employee_id = p_emp_id;
  BEGIN
    OPEN c_empjob_csr;
    FETCH c_empjob_csr BULK COLLECT INTO v_jobtab;
    CLOSE c_empjob_csr;
    RETURN v_jobtab.count;
  EXCEPTION
    WHEN NO_DATA_FOUND THEN
      RAISE_APPLICATION_ERROR(-20348,
        'Employee with ID '|| p_emp_id ||' does not exist!');
      RETURN 0;
  END get_job_count;
```

```
FUNCTION get_years_service(
  p_emp_id IN employees.employee_id%TYPE) RETURN NUMBER IS
  CURSOR c_jobh_csr IS
    SELECT MONTHS_BETWEEN(end_date, start_date)/12 v_years_in_job
    FROM job_history
    WHERE employee_id = p_emp_id;
  v_years_service NUMBER(2) := 0;
  v_years_in_job NUMBER(2) := 0;
BEGIN
  FOR jobh_rec IN c_jobh_csr
  LOOP
    EXIT WHEN c_jobh_csr%NOTFOUND;
    v_years_service := v_years_service + jobh_rec.v_years_in_job;
  END LOOP;
  SELECT MONTHS_BETWEEN(SYSDATE, hire_date)/12 INTO v_years_in_job
  FROM employees
  WHERE employee_id = p_emp_id;
  v_years_service := v_years_service + v_years_in_job;
  RETURN ROUND(v_years_service);
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    RAISE_APPLICATION_ERROR(-20348,
      'Employee with ID '|| p_emp_id ||' does not exist.');
    RETURN 0;
END get_years_service;

PROCEDURE new_job(
  p_jobid IN jobs.job_id%TYPE,
  p_title IN jobs.job_title%TYPE,
  p_minsal IN jobs.min_salary%TYPE) IS
  v_maxsal jobs.max_salary%TYPE := 2 * p_minsal;
BEGIN
  INSERT INTO jobs(job_id, job_title, min_salary, max_salary)
  VALUES (p_jobid, p_title, p_minsal, v_maxsal);
  DBMS_OUTPUT.PUT_LINE ('New row added to JOBS table:');
  DBMS_OUTPUT.PUT_LINE (p_jobid || ' ' || p_title ||' '||
                        p_minsal || ' ' || v_maxsal);
END new_job;

PROCEDURE upd_jobsal(
  p_jobid IN jobs.job_id%type,
  p_new_minsal IN jobs.min_salary%type,
  p_new_maxsal IN jobs.max_salary%type) IS
  v_dummy PLS_INTEGER;
  e_resource_busy EXCEPTION;
  e_sal_error EXCEPTION;
  PRAGMA EXCEPTION_INIT (e_resource_busy , -54);
BEGIN
  IF (p_new_maxsal < p_new_minsal) THEN
    RAISE e_sal_error;
  END IF;
  SELECT 1 INTO v_dummy
  FROM jobs
```

```
    WHERE job_id = p_jobid
    FOR UPDATE OF min_salary NOWAIT;
    UPDATE jobs
      SET min_salary = p_new_minsal,
          max_salary = p_new_maxsal
    WHERE job_id = p_jobid;
  EXCEPTION
    WHEN e_resource_busy THEN
      RAISE_APPLICATION_ERROR (-20001,
        'Job information is currently locked, try later.');
    WHEN NO_DATA_FOUND THEN
      RAISE_APPLICATION_ERROR(-20001, 'This job ID does not exist');
    WHEN e_sal_error THEN
      RAISE_APPLICATION_ERROR(-20001,
        'Data error: Max salary should be more than min salary');
  END upd_jobsal;
END empjob_pkg;
/
SHOW ERRORS

PACKAGE BODY empjob_pkg Compiled.
No Errors.
```

c. Invoke your `EMPJOB_PKG.NEW_JOB` procedure to create a new job with ID `PR_MAN`, job title `Public Relations Manager`, and salary `6250`.

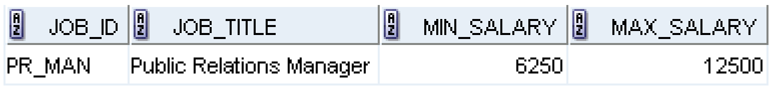
```
EXECUTE empjob_pkg.new_job('PR_MAN', 'Public Relations Manager', 6250)

anonymous block completed
New row added to JOBS table:
PR_MAN Public Relations Manager 6250 12500.
```

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.

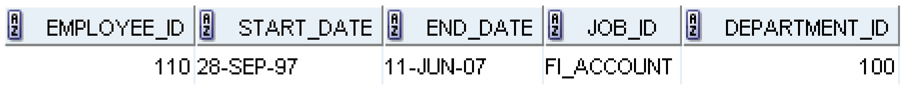
```
ALTER TRIGGER update_job_history DISABLE;
EXECUTE empjob_pkg.add_job_hist(110, 'PR_MAN')
ALTER TRIGGER update_job_history ENABLE;

ALTER TRIGGER update_job_history succeeded.
anonymous block completed
Added employee 110 details to the JOB_HISTORY table
Updated current job of employee 110 to PR_MAN

ALTER TRIGGER update_job_history succeeded.
```

e. Query the `JOBS`, `JOB_HISTORY`, and `EMPLOYEES` tables to verify the results.

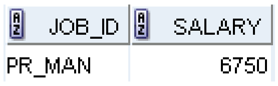
```
SELECT * FROM jobs WHERE job_id = 'PR_MAN';
SELECT * FROM job_history WHERE employee_id = 110;
SELECT job_id, salary FROM employees WHERE employee_id = 110;
```



JOB_ID	JOB_TITLE	MIN_SALARY	MAX_SALARY
PR_MAN	Public Relations Manager	6250	12500



EMPLOYEE_ID	START_DATE	END_DATE	JOB_ID	DEPARTMENT_ID
110	28-SEP-97	11-JUN-07	FI_ACCOUNT	100



JOB_ID	SALARY
PR_MAN	6750

Part A: Additional Practice 8 Solutions

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 outside the new range specified for the job.

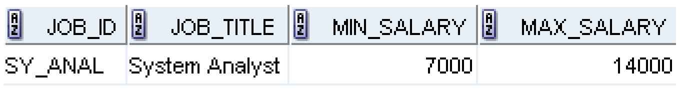
   a. Create a trigger called `CHECK_SAL_RANGE` that is fired before every row that is updated in the `MIN_SALARY` and `MAX_SALARY` columns in the `JOBS` table. For any minimum or maximum salary value that is changed, check whether the salary of any existing employee with that job ID in the `EMPLOYEES` 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.

```
CREATE OR REPLACE TRIGGER check_sal_range
BEFORE UPDATE OF min_salary, max_salary ON jobs
FOR EACH ROW
DECLARE
  v_minsal employees.salary%TYPE;
  v_maxsal employees.salary%TYPE;
  e_invalid_salrange EXCEPTION;
BEGIN
  SELECT MIN(salary), MAX(salary) INTO v_minsal, v_maxsal
  FROM employees
  WHERE job_id = :NEW.job_id;
  IF (v_minsal < :NEW.min_salary) OR (v_maxsal > :NEW.max_salary) THEN
    RAISE e_invalid_salrange;
  END IF;
EXCEPTION
  WHEN e_invalid_salrange THEN
    RAISE_APPLICATION_ERROR(-20550,
     'Employees exist whose salary is out of the specified range. '||
     'Therefore the specified salary range cannot be updated.');
END check_sal_range;
/
SHOW ERRORS

TRIGGER check_sal_range Compiled.
No Errors.
```

Part A: Additional Practice 8 Solutions (continued)

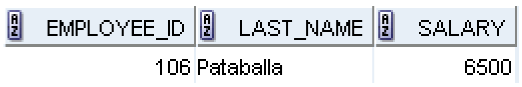
b. Test the trigger using the `SY_ANAL` job, setting the new minimum salary to `5000` and the new maximum salary to `7000`. Before you make the change, write a query to display the current salary range for the `SY_ANAL` job ID, and another query to display the employee ID, last name, and salary for the same job ID. After the update, query the change (if any) to the `JOBS` table for the specified job ID.

```
SELECT * FROM jobs
WHERE job_id = 'SY_ANAL';
```



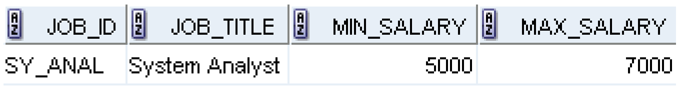
JOB_ID	JOB_TITLE	MIN_SALARY	MAX_SALARY
SY_ANAL	System Analyst	7000	14000

```
SELECT employee_id, last_name, salary
FROM  employees
WHERE job_id = 'SY_ANAL';
```



EMPLOYEE_ID	LAST_NAME	SALARY
106	Pataballa	6500

```
UPDATE jobs
  SET min_salary = 5000, max_salary = 7000
  WHERE job_id = 'SY_ANAL';

1 row updated.

SELECT * FROM jobs
WHERE job_id = 'SY_ANAL';
```



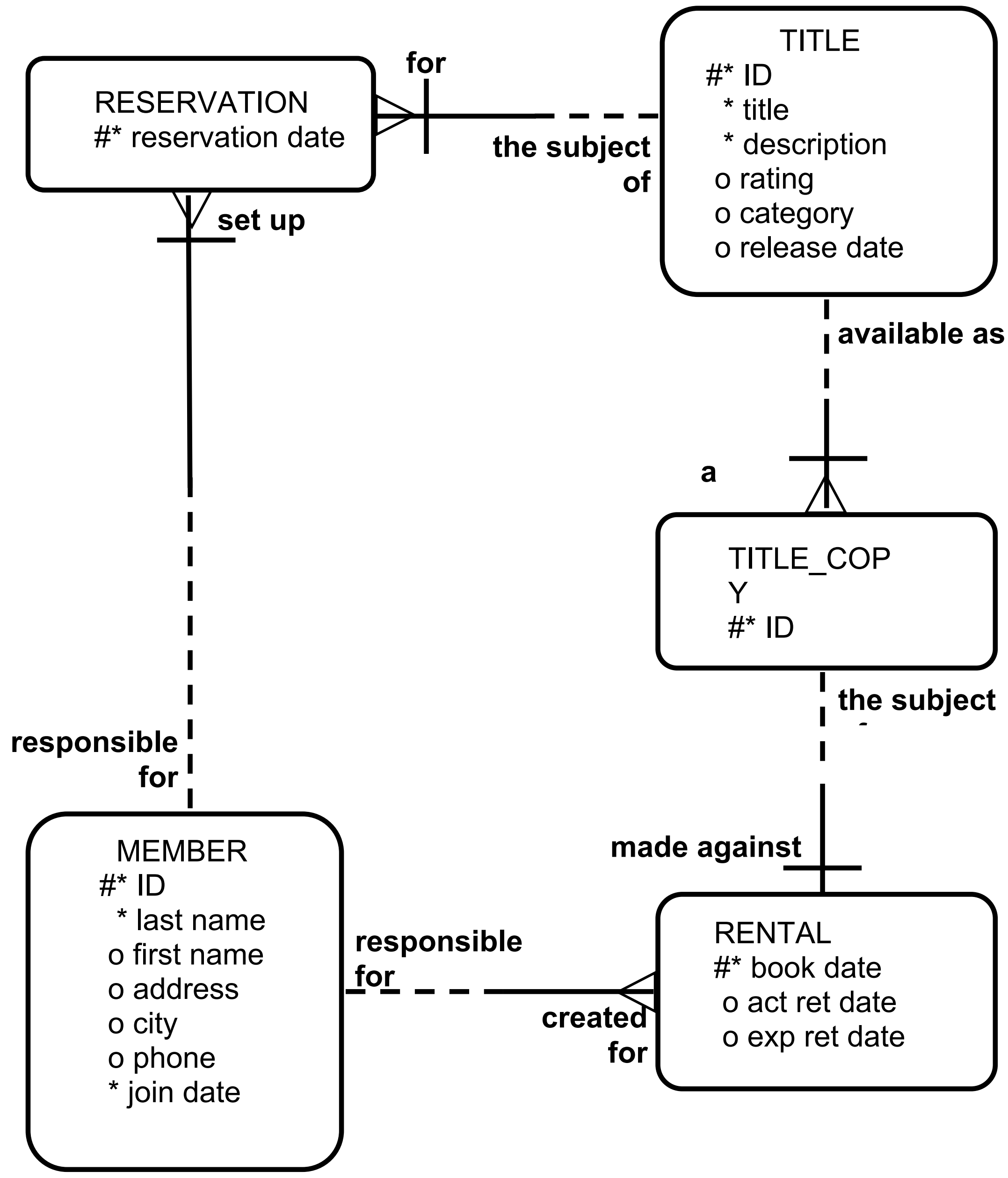
JOB_ID	JOB_TITLE	MIN_SALARY	MAX_SALARY
SY_ANAL	System Analyst	5000	7000

c. Using the job `SY_ANAL`, set the new minimum salary to `7000` and the new maximum salary to `18000`. Explain the results.

```
UPDATE jobs
  SET min_salary = 7000, max_salary = 18000
  WHERE job_id = 'SY_ANAL';

Error starting at line 1 in command:
UPDATE jobs
  SET min_salary = 7000, max_salary = 18000
  WHERE job_id = 'SY_ANAL'
Error report:
SQL Error: ORA-20550: Employees exist whose salary is out of the
specified range. Therefore the specified salary range cannot be updated.
ORA-06512: at "ORA61.CHECK_SAL_RANGE", line 14
ORA-04088: error during execution of trigger 'ORA61.CHECK_SAL_RANGE'
```

The update fails to change the salary range due to the functionality provided by the `CHECK_SAL_RANGE` trigger because employee 106 who has the `SY_ANAL` job ID has a salary of `6500`, which is less than the minimum salary for the new salary range specified in the `UPDATE` statement.

Part B: Entity Relationship Diagram



Part B (continued)

In this case study, 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, create a trigger to ensure that any data in the video tables is modified only during business hours.

Create the package by using *i*SQL*Plus 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: Additional Practice 1 Solutions

1. Load and execute the `E:\labs\PLPU\labs\buildvid1.sql` script to create all the required tables and sequences that are needed for this exercise.

```
SET ECHO OFF
/* Script to build the Video Application (Part 1 - buildvid1.sql)
   for the Oracle Introduction to Oracle with Procedure Builder course.
   Created by: Debby Kramer Creation date: 12/10/95
   Last upated: 2/13/96
   Modified by Nagavalli Pataballa on 26-APR-2001
    For the course Introduction to Oracle9i: PL/SQL
    This part of the script creates tables and sequences that are used
    by Part B of the Additional Practices of the course.
    Ignore the errors which appear due to dropping of table.
*/

DROP TABLE rental CASCADE CONSTRAINTS;
DROP TABLE reservation CASCADE CONSTRAINTS;
DROP TABLE title_copy CASCADE CONSTRAINTS;
DROP TABLE title CASCADE CONSTRAINTS;
DROP TABLE member CASCADE CONSTRAINTS;

PROMPT Please wait while tables are created....

CREATE TABLE MEMBER
  (member_id  NUMBER (10)        CONSTRAINT member_id_pk PRIMARY KEY
 , last_name  VARCHAR2(25)
    CONSTRAINT member_last_nn NOT NULL
 , first_name VARCHAR2(25)
 , address    VARCHAR2(100)
 , city       VARCHAR2(30)
 , phone      VARCHAR2(25)
 , join_date  DATE DEFAULT SYSDATE
    CONSTRAINT join_date_nn NOT NULL)
/

CREATE TABLE TITLE
  (title_id   NUMBER(10)
     CONSTRAINT title_id_pk PRIMARY KEY
 , title      VARCHAR2(60)
     CONSTRAINT title_nn NOT NULL
 , description VARCHAR2(400)
     CONSTRAINT title_desc_nn NOT NULL
 , rating      VARCHAR2(4)
     CONSTRAINT title_rating_ck CHECK (rating IN
('G','PG','R','NC17','NR'))
 , category     VARCHAR2(20) DEFAULT 'DRAMA'
     CONSTRAINT title_categ_ck CHECK (category IN
('DRAMA','COMEDY','ACTION', 'CHILD','SCIFI','DOCUMENTARY'))
 , release_date DATE)
/
```

Part B: Additional Practice 1 Solutions (continued)

```
CREATE TABLE TITLE_COPY
  (copy_id   NUMBER(10)
 , title_id  NUMBER(10)
    CONSTRAINT copy_title_id_fk
       REFERENCES title(title_id)
 , status    VARCHAR2(15)
     CONSTRAINT copy_status_nn NOT NULL
     CONSTRAINT copy_status_ck CHECK (status IN ('AVAILABLE',
'DESTROYED',
                                  'RENTED', 'RESERVED'))
 , CONSTRAINT copy_title_id_pk  PRIMARY KEY(copy_id, title_id))
/
CREATE TABLE RENTAL
  (book_date DATE DEFAULT SYSDATE
 , copy_id   NUMBER(10)
 , member_id NUMBER(10)
    CONSTRAINT rental_mbr_id_fk REFERENCES member(member_id)
 , title_id  NUMBER(10)
 , act_ret_date DATE
 , exp_ret_date DATE DEFAULT SYSDATE+2
 , CONSTRAINT rental_copy_title_id_fk FOREIGN KEY (copy_id, title_id)
              REFERENCES title_copy(copy_id,title_id)
 , CONSTRAINT rental_id_pk PRIMARY KEY(book_date, copy_id, title_id,
member_id))
/
CREATE TABLE RESERVATION
  (res_date  DATE
 , member_id NUMBER(10)
 , title_id  NUMBER(10)
 , CONSTRAINT res_id_pk PRIMARY KEY(res_date, member_id, title_id))
/

PROMPT Tables created.
DROP SEQUENCE title_id_seq;
DROP SEQUENCE member_id_seq;

PROMPT Creating Sequences...
CREATE SEQUENCE member_id_seq
  START WITH 101
  NOCACHE

CREATE SEQUENCE title_id_seq
  START WITH 92
  NOCACHE
/

PROMPT Sequences created.

PROMPT Run buildvid2.sql now to populate the above tables.
```

Part B: Additional Practice 2 Solutions

2. Load and execute the `E:\labs\PLPU\labs\buildvid2.sql` script to populate all the tables created by the `buildvid1.sql` script.

```
/* Script to build the Video Application (Part 2 - buildvid2.sql)
   This part of the script populates the tables that are created using
   buildvid1.sql
   These are used by Part B of the Additional Practices of the course.
   You should run the script buildvid1.sql before running this script to
   create the above tables.
*/

INSERT INTO member
  VALUES  (member_id_seq.NEXTVAL, 'Velasquez', 'Carmen',
    '283 King Street', 'Seattle', '587-99-6666', '03-MAR-90');
INSERT INTO member
  VALUES   (member_id_seq.NEXTVAL, 'Ngao', 'LaDoris',
    '5 Modrany',  'Bratislava', '586-355-8882', '08-MAR-90');
INSERT INTO member
  VALUES  (member_id_seq.NEXTVAL,'Nagayama', 'Midori',
    '68 Via Centrale', 'Sao Paolo', '254-852-5764', '17-JUN-91');
INSERT INTO member
  VALUES  (member_id_seq.NEXTVAL,'Quick-To-See','Mark',
    '6921 King Way', 'Lagos', '63-559-777', '07-APR-90');
INSERT INTO member
   VALUES  (member_id_seq.NEXTVAL, 'Ropeburn', 'Audry',
    '86 Chu Street',  'Hong Kong', '41-559-87', '04-MAR-90');
INSERT INTO member
  VALUES (member_id_seq.NEXTVAL, 'Urguhart', 'Molly',
    '3035 Laurier Blvd.',  'Quebec', '418-542-9988','18-JAN-91');
INSERT INTO member
  VALUES (member_id_seq.NEXTVAL, 'Menchu', 'Roberta',
    'Boulevard de Waterloo 41', 'Brussels', '322-504-2228', '14-MAY-90');
INSERT INTO member
  VALUES (member_id_seq.NEXTVAL, 'Biri', 'Ben',
    '398 High St.', 'Columbus', '614-455-9863', '07-APR-90');
INSERT INTO member
  VALUES (member_id_seq.NEXTVAL, 'Catchpole', 'Antoinette',
    '88 Alfred St.', 'Brisbane', '616-399-1411', '09-FEB-92');

COMMIT;
```

Part B: Additional Practice 2 Solutions (continued)

```
INSERT INTO TITLE (title_id, title, description, rating, category,
release_date)
  VALUES (TITLE_ID_SEQ.NEXTVAL, 'Willie and Christmas Too',
   'All of Willie''s friends made a Christmas list for Santa, but Willie
has yet to create his own wish list.', 'G', 'CHILD', '05-OCT-95');
INSERT INTO TITLE (title_id, title, description, rating, category,
release_date)
  VALUES (TITLE_ID_SEQ.NEXTVAL, 'Alien Again', 'Another installment of
science fiction history. Can the heroine save the planet from the alien
life form?', 'R', 'SCIFI',             '19-MAY-95');
INSERT INTO TITLE (title_id, title, description, rating, category,
release_date)
  VALUES (TITLE_ID_SEQ.NEXTVAL, 'The Glob', 'A meteor crashes near a
small American town and unleashes carivorous goo in this classic.', 'NR',
'SCIFI', '12-AUG-95');
INSERT INTO TITLE (title_id, title, description, rating, category,
release_date)
  VALUES (TITLE_ID_SEQ.NEXTVAL, 'My Day Off', 'With a little luck and a
lot of ingenuity, a teenager skips school for a day in New York.', 'PG',
'COMEDY', '12-JUL-95');
INSERT INTO TITLE (title_id, title, description, rating, category,
release_date)
  VALUES (TITLE_ID_SEQ.NEXTVAL, 'Miracles on Ice', 'A six-year-old has
doubts about Santa Claus. But she discovers that miracles really do
exist.', 'PG', 'DRAMA', '12-SEP-95');
INSERT INTO TITLE (title_id, title, description, rating, category,
release_date)
  VALUES (TITLE_ID_SEQ.NEXTVAL,  'Soda Gang', 'After discovering a cached
of drugs, a young couple find themselves pitted against a vicious gang.',
'NR', 'ACTION', '01-JUN-95');
INSERT INTO title (title_id, title, description, rating, category,
release_date)
  VALUES (TITLE_ID_SEQ.NEXTVAL, 'Interstellar Wars', 'Futuristic
interstellar action movie. Can the rebels save the humans from the evil
Empire?', 'PG', 'SCIFI','07-JUL-77');

COMMIT;

INSERT INTO title_copy VALUES (1,92, 'AVAILABLE');
INSERT INTO title_copy VALUES (1,93, 'AVAILABLE');
INSERT INTO title_copy VALUES (2,93, 'RENTED');
INSERT INTO title_copy VALUES (1,94, 'AVAILABLE');
INSERT INTO title_copy VALUES (1,95, 'AVAILABLE');
INSERT INTO title_copy VALUES (2,95, 'AVAILABLE');
INSERT INTO title_copy VALUES (3,95, 'RENTED');
INSERT INTO title_copy VALUES (1,96, 'AVAILABLE');
INSERT INTO title_copy VALUES (1,97, 'AVAILABLE');
INSERT INTO title_copy VALUES (1,98, 'RENTED');
INSERT INTO title_copy VALUES (2,98, 'AVAILABLE');

COMMIT;
```

Part B: Additional Practice 2 Solutions (continued)

```
INSERT INTO reservation VALUES (sysdate-1, 101, 93);
INSERT INTO reservation VALUES (sysdate-2, 106, 102);

COMMIT;

INSERT INTO rental VALUES (sysdate-1, 2, 101, 93, null, sysdate+1);
INSERT INTO rental VALUES (sysdate-2, 3, 102, 95, null, sysdate);
INSERT INTO rental VALUES (sysdate-3, 1, 101, 98, null, sysdate-1);
INSERT INTO rental VALUES (sysdate-4, 1, 106, 97, sysdate-2, sysdate-2);
INSERT INTO rental VALUES (sysdate-3, 1, 101, 92, sysdate-2, sysdate-1);

COMMIT;

PROMPT ** Tables built and data loaded **
```

Part B: Additional Practice 3 Solutions

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 the 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 or her 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, then 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: Additional Practice 3 Solutions (continued)

`VIDEO_PKG` Package Specification

```
CREATE OR REPLACE PACKAGE video_pkg IS
  PROCEDURE new_member
    (p_lname       IN member.last_name%TYPE,
     p_fname       IN member.first_name%TYPE   DEFAULT NULL,
     p_address     IN member.address%TYPE      DEFAULT NULL,
     p_city        IN member.city%TYPE         DEFAULT NULL,
     p_phone       IN member.phone%TYPE        DEFAULT NULL);

  FUNCTION new_rental
    (p_memberid   IN rental.member_id%TYPE,
     p_titleid    IN rental.title_id%TYPE)
    RETURN DATE;

  FUNCTION new_rental
    (p_membername IN member.last_name%TYPE,
     p_titleid    IN rental.title_id%TYPE)
    RETURN DATE;

  PROCEDURE return_movie
    (p_titleid    IN rental.title_id%TYPE,
     p_copyid     IN rental.copy_id%TYPE,
     p_sts        IN title_copy.status%TYPE);
END video_pkg;
/
SHOW ERRORS

PACKAGE video_pkg Compiled.
No Errors.
```

`VIDEO_PKG` Package Body

```
CREATE OR REPLACE PACKAGE BODY video_pkg IS
  PROCEDURE exception_handler(errcode IN  NUMBER, context IN VARCHAR2) IS
  BEGIN
    IF errcode = -1 THEN
      RAISE_APPLICATION_ERROR(-20001,
        'The number is assigned to this member is already in use, '||
        'try again.');
    ELSIF errcode = -2291 THEN
      RAISE_APPLICATION_ERROR(-20002, context ||
        ' has attempted to use a foreign key value that is invalid');
    ELSE
      RAISE_APPLICATION_ERROR(-20999, 'Unhandled error in ' ||
        context || '. Please contact your application '||
        'administrator with the following information: '
        || CHR(13) || SQLERRM);
    END IF;
  END exception_handler;
```

Part B: Additional Practice 3 Solutions (continued)

```
  PROCEDURE reserve_movie
    (memberid  IN  reservation.member_id%TYPE,
     titleid   IN  reservation.title_id%TYPE) IS
    CURSOR rented_csr IS
      SELECT exp_ret_date
        FROM rental
        WHERE title_id = titleid
        AND act_ret_date IS NULL;
  BEGIN
    INSERT INTO reservation (res_date, member_id, title_id)
    VALUES (SYSDATE, memberid, titleid);
    COMMIT;
    FOR rented_rec IN rented_csr LOOP
      DBMS_OUTPUT.PUT_LINE('Movie reserved. Expected back on: '
        || rented_rec.exp_ret_date);
      EXIT WHEN rented_csr%found;
    END LOOP;
  EXCEPTION
    WHEN OTHERS THEN
      exception_handler(SQLCODE, 'RESERVE_MOVIE');
  END reserve_movie;

 PROCEDURE return_movie(
   titleid IN rental.title_id%TYPE,
   copyid IN rental.copy_id%TYPE,
   sts IN title_copy.status%TYPE) IS
    v_dummy VARCHAR2(1);
    CURSOR res_csr IS
      SELECT *
      FROM reservation
      WHERE title_id = titleid;
  BEGIN
    SELECT '' INTO v_dummy
      FROM title
      WHERE title_id = titleid;
    UPDATE rental
      SET act_ret_date = SYSDATE
      WHERE title_id = titleid
      AND copy_id = copyid AND act_ret_date IS NULL;
    UPDATE title_copy
      SET status = UPPER(sts)
      WHERE title_id = titleid AND copy_id = copyid;
    FOR res_rec IN res_csr LOOP
      IF res_csr%FOUND THEN
        DBMS_OUTPUT.PUT_LINE('Put this movie on hold -- '||
          'reserved by member #' || res_rec.member_id);
      END IF;
    END LOOP;
  EXCEPTION
    WHEN OTHERS THEN
      exception_handler(SQLCODE, 'RETURN_MOVIE');
  END return_movie;
```

Part B: Additional Practice 3 Solutions (continued)

```
FUNCTION new_rental(
  memberid  IN  rental.member_id%TYPE,
  titleid   IN  rental.title_id%TYPE) RETURN DATE IS
  CURSOR copy_csr IS
    SELECT * FROM title_copy
    WHERE title_id = titleid
    FOR UPDATE;
  flag   BOOLEAN  := FALSE;
BEGIN

  FOR copy_rec IN copy_csr LOOP
    IF copy_rec.status = 'AVAILABLE' THEN
      UPDATE title_copy
        SET status = 'RENTED'
        WHERE CURRENT OF copy_csr;
      INSERT INTO rental(book_date, copy_id, member_id,
                         title_id, exp_ret_date)
      VALUES (SYSDATE, copy_rec.copy_id, memberid,
                         titleid, SYSDATE + 3);
      flag := TRUE;
      EXIT;
    END IF;
  END LOOP;
  COMMIT;
  IF flag THEN
    RETURN (SYSDATE + 3);
  ELSE
    reserve_movie(memberid, titleid);
    RETURN NULL;
  END IF;
EXCEPTION
  WHEN OTHERS THEN
    exception_handler(SQLCODE, 'NEW_RENTAL');
END new_rental;

FUNCTION new_rental(
  membername IN member.last_name%TYPE,
  titleid    IN rental.title_id%TYPE) RETURN DATE IS
  CURSOR copy_csr IS
    SELECT * FROM title_copy
      WHERE title_id = titleid
      FOR UPDATE;
  flag  BOOLEAN  := FALSE;
  memberid  member.member_id%TYPE;
  CURSOR member_csr IS
    SELECT member_id, last_name, first_name
      FROM member
      WHERE LOWER(last_name) = LOWER(membername)
      ORDER BY last_name, first_name;
```

Part B: Additional Practice 3 Solutions (continued)

```
  BEGIN
    SELECT member_id INTO memberid
      FROM member
      WHERE lower(last_name) = lower(membername);
    FOR copy_rec IN copy_csr LOOP
      IF copy_rec.status = 'AVAILABLE' THEN
        UPDATE title_copy
          SET status = 'RENTED'
          WHERE CURRENT OF copy_csr;
        INSERT INTO rental (book_date, copy_id, member_id,
                            title_id, exp_ret_date)
          VALUES (SYSDATE, copy_rec.copy_id, memberid,
                            titleid, SYSDATE + 3);
        flag := TRUE;
        EXIT;
      END IF;
    END LOOP;
    COMMIT;
    IF flag THEN
      RETURN(SYSDATE + 3);
    ELSE
      reserve_movie(memberid, titleid);
      RETURN NULL;
    END IF;
  EXCEPTION
    WHEN TOO_MANY_ROWS THEN
      DBMS_OUTPUT.PUT_LINE(
       'Warning! More than one member by this name.');
      FOR member_rec IN member_csr LOOP
        DBMS_OUTPUT.PUT_LINE(member_rec.member_id || CHR(9) ||
          member_rec.last_name || ', ' || member_rec.first_name);
      END LOOP;
      RETURN NULL;
    WHEN OTHERS THEN
      exception_handler(SQLCODE, 'NEW_RENTAL');
  END new_rental;

  PROCEDURE new_member(
    lname       IN member.last_name%TYPE,
    fname       IN member.first_name%TYPE   DEFAULT NULL,
    address     IN member.address%TYPE      DEFAULT NULL,
    city        IN member.city%TYPE         DEFAULT NULL,
    phone       IN member.phone%TYPE        DEFAULT NULL) IS
  BEGIN
    INSERT INTO member(member_id, last_name, first_name,
                       address, city, phone, join_date)
      VALUES(member_id_seq.NEXTVAL, lname, fname,
              address, city, phone, SYSDATE);
    COMMIT;
CREATE OR REPLACE PACKAGE BODY video_pkg IS
  PROCEDURE exception_handler(errcode IN  NUMBER, p_context IN VARCHAR2)
IS
```

```
  BEGIN
    IF errcode = -1 THEN
      RAISE_APPLICATION_ERROR(-20001,
        'The number is assigned to this member is already in use, '||
        'try again.');
    ELSIF errcode = -2291 THEN
      RAISE_APPLICATION_ERROR(-20002, p_context ||
        ' has attempted to use a foreign key value that is invalid');
    ELSE
      RAISE_APPLICATION_ERROR(-20999, 'Unhandled error in ' ||
        p_context || '. Please contact your application '||
        'administrator with the following information: '
        || CHR(13) || SQLERRM);
    END IF;
  END exception_handler;

  PROCEDURE reserve_movie
    (p_memberid  IN  reservation.member_id%TYPE,
     p_titleid   IN  reservation.title_id%TYPE) IS
    CURSOR c_rented_csr IS
      SELECT exp_ret_date
        FROM rental
        WHERE title_id = p_titleid
        AND act_ret_date IS NULL;
  BEGIN
    INSERT INTO reservation (res_date, member_id, title_id)
    VALUES (SYSDATE, p_memberid, p_titleid);
    COMMIT;
    FOR rented_rec IN c_rented_csr LOOP
      DBMS_OUTPUT.PUT_LINE('Movie reserved. Expected back on: '
        || rented_rec.exp_ret_date);
      EXIT WHEN c_rented_csr%found;
    END LOOP;
  EXCEPTION
    WHEN OTHERS THEN
      exception_handler(SQLCODE, 'RESERVE_MOVIE');
  END reserve_movie;

PROCEDURE return_movie(
   p_titleid IN rental.title_id%TYPE,
   p_copyid IN rental.copy_id%TYPE,
   p_sts IN title_copy.status%TYPE) IS
   v_dummy VARCHAR2(1);
   CURSOR c_res_csr IS
      SELECT *
      FROM reservation
      WHERE title_id = p_titleid;
  BEGIN
    SELECT '' INTO v_dummy
      FROM title
      WHERE title_id = p_titleid;
    UPDATE rental
      SET act_ret_date = SYSDATE
      WHERE title_id = p_titleid
      AND copy_id = p_copyid AND act_ret_date IS NULL;
    UPDATE title_copy
      SET status = UPPER(p_sts)
      WHERE title_id = p_titleid AND copy_id = p_copyid;
    FOR res_rec IN c_res_csr LOOP
      IF c_res_csr%FOUND THEN
        DBMS_OUTPUT.PUT_LINE('Put this movie on hold -- '||
          'reserved by member #' || res_rec.member_id);
      END IF;
    END LOOP;
  EXCEPTION
    WHEN OTHERS THEN
      exception_handler(SQLCODE, 'RETURN_MOVIE');
  END return_movie;

  FUNCTION new_rental(
    p_memberid  IN  rental.member_id%TYPE,
    p_titleid   IN  rental.title_id%TYPE) RETURN DATE IS
    CURSOR c_copy_csr IS
      SELECT * FROM title_copy
      WHERE title_id = p_titleid
      FOR UPDATE;
    v_flag   BOOLEAN  := FALSE;
  BEGIN
    FOR copy_rec IN c_copy_csr LOOP
      IF copy_rec.status = 'AVAILABLE' THEN
        UPDATE title_copy
          SET status = 'RENTED'
          WHERE CURRENT OF c_copy_csr;
        INSERT INTO rental(book_date, copy_id, member_id,
                           title_id, exp_ret_date)
        VALUES (SYSDATE, copy_rec.copy_id, p_memberid,
                           p_titleid, SYSDATE + 3);
        v_flag := TRUE;
        EXIT;
      END IF;
    END LOOP;
    COMMIT;
    IF v_flag THEN
      RETURN (SYSDATE + 3);
    ELSE
      reserve_movie(p_memberid, p_titleid);
      RETURN NULL;
    END IF;
  EXCEPTION
    WHEN OTHERS THEN
      exception_handler(SQLCODE, 'NEW_RENTAL');
      RETURN NULL;
  END new_rental;

  FUNCTION new_rental(
    p_membername IN member.last_name%TYPE,
    p_titleid    IN rental.title_id%TYPE) RETURN DATE IS
    CURSOR c_copy_csr IS
      SELECT * FROM title_copy
        WHERE title_id = p_titleid
        FOR UPDATE;
    v_flag  BOOLEAN  := FALSE;
    v_memberid  member.member_id%TYPE;
    CURSOR c_member_csr IS
      SELECT member_id, last_name, first_name
        FROM member
        WHERE LOWER(last_name) = LOWER(p_membername)
        ORDER BY last_name, first_name;
  BEGIN
    SELECT member_id INTO v_memberid
      FROM member
      WHERE lower(last_name) = lower(p_membername);
    FOR copy_rec IN c_copy_csr LOOP
      IF copy_rec.status = 'AVAILABLE' THEN
        UPDATE title_copy
          SET status = 'RENTED'
          WHERE CURRENT OF c_copy_csr;
        INSERT INTO rental (book_date, copy_id, member_id,
                            title_id, exp_ret_date)
          VALUES (SYSDATE, copy_rec.copy_id, v_memberid,
                            p_titleid, SYSDATE + 3);
        v_flag := TRUE;
        EXIT;
      END IF;
    END LOOP;
    COMMIT;
    IF v_flag THEN
      RETURN(SYSDATE + 3);
    ELSE
      reserve_movie(v_memberid, p_titleid);
      RETURN NULL;
    END IF;
  EXCEPTION
    WHEN TOO_MANY_ROWS THEN
      DBMS_OUTPUT.PUT_LINE(
       'Warning! More than one member by this name.');
      FOR member_rec IN c_member_csr LOOP
        DBMS_OUTPUT.PUT_LINE(member_rec.member_id || CHR(9) ||
          member_rec.last_name || ', ' || member_rec.first_name);
      END LOOP;
      RETURN NULL;
    WHEN OTHERS THEN
      exception_handler(SQLCODE, 'NEW_RENTAL');
      RETURN NULL;
  END new_rental;

  PROCEDURE new_member(
    p_lname       IN member.last_name%TYPE,
    p_fname       IN member.first_name%TYPE   DEFAULT NULL,
    p_address     IN member.address%TYPE      DEFAULT NULL,
    p_city        IN member.city%TYPE         DEFAULT NULL,
    p_phone       IN member.phone%TYPE        DEFAULT NULL) IS
  BEGIN
    INSERT INTO member(member_id, last_name, first_name,
                       address, city, phone, join_date)
      VALUES(member_id_seq.NEXTVAL, p_lname, p_fname,
              p_address, p_city, p_phone, SYSDATE);
    COMMIT;
  EXCEPTION
    WHEN OTHERS THEN
      exception_handler(SQLCODE, 'NEW_MEMBER');
  END new_member;
END video_pkg;
/
SHOW ERRORS

PACKAGE BODY video_pkg Compiled.
No Errors.
```

Part B: Additional Practice 4 Solutions

4. Use the following scripts located in the `E:\labs\PLPU\soln` directory to test your routines:

   a. Add two members using `sol_apb_04_a.sql`.

```
EXECUTE video_pkg.new_member('Haas', 'James', 'Chestnut Street',
'Boston', '617-123-4567')
EXECUTE  video_pkg.new_member('Biri', 'Allan',  'Hiawatha Drive', 'New
York', '516-123-4567')

anonymous block completed
anonymous block completed.
```

   b. Add new video rentals using `sol_apb_04_b.sql`.

```
EXEC DBMS_OUTPUT.PUT_LINE(video_pkg.new_rental(110, 98))
EXEC DBMS_OUTPUT.PUT_LINE(video_pkg.new_rental(109, 93))
EXEC DBMS_OUTPUT.PUT_LINE(video_pkg.new_rental(107, 98))
EXEC DBMS_OUTPUT.PUT_LINE(video_pkg.new_rental('Biri', 97))
EXEC DBMS_OUTPUT.PUT_LINE(video_pkg.new_rental(97, 97))

anonymous block completed
14-JUN-07

anonymous block completed
14-JUN-07

anonymous block completed
Movie reserved. Expected back on: 10-JUN-07


anonymous block completed
Warning! More than one member by this name.
111   Biri, Allan
108   Biri, Ben



Error starting at line 5 in command:
EXEC DBMS_OUTPUT.PUT_LINE(video_pkg.new_rental(97, 97))
Error report:
ORA-20002: NEW_RENTAL has attempted to use a foreign key value that is
invalid
ORA-06512: at "ORA61.VIDEO_PKG", line 9
ORA-06512: at "ORA61.VIDEO_PKG", line 103
ORA-06512: at line 1
```

Part B: Additional Practice 4 Solutions (continued)

c. Return movies by using the `sol_apb_04_c.sql` script.

```
EXECUTE video_pkg.return_movie(98, 1, 'AVAILABLE')
EXECUTE video_pkg.return_movie(95, 3, 'AVAILABLE')
EXECUTE video_pkg.return_movie(111, 1, 'RENTED')

anonymous block completed
Put this movie on hold -- reserved by member #107

anonymous block completed

Error starting at line 3 in command:
EXECUTE video_pkg.return_movie(111, 1, 'RENTED')
Error report:
ORA-20999: Unhandled error in RETURN_MOVIE. Please contact your
application administrator with the following information:
ORA-01403: no data found
ORA-06512: at "ORA61.VIDEO_PKG", line 12
ORA-06512: at "ORA61.VIDEO_PKG", line 69
ORA-06512: at line 1
```

Part B: Additional Practice 5 Solutions

5. The business hours for the video store are 8:00 AM through 10:00 PM, Sunday through Friday, and 8:00 AM through 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.

```
CREATE OR REPLACE PROCEDURE time_check IS
BEGIN
  IF ((TO_CHAR(SYSDATE,'D') BETWEEN 1 AND 6) AND
      (TO_DATE(TO_CHAR(SYSDATE, 'hh24:mi'), 'hh24:mi') NOT BETWEEN
       TO_DATE('08:00', 'hh24:mi') AND TO_DATE('22:00', 'hh24:mi')))
       OR ((TO_CHAR(SYSDATE, 'D') = 7)
       AND  (TO_DATE(TO_CHAR(SYSDATE, 'hh24:mi'), 'hh24:mi') NOT BETWEEN
       TO_DATE('08:00', 'hh24:mi') AND TO_DATE('24:00', 'hh24:mi'))) THEN
    RAISE_APPLICATION_ERROR(-20999,
       'Data changes restricted to office hours.');
  END IF;
END time_check;
/
SHOW ERRORS

PROCEDURE time_check Compiled.
No Errors.
```

   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.

```
CREATE OR REPLACE TRIGGER member_trig
  BEFORE INSERT OR UPDATE OR DELETE ON member
CALL time_check
/

CREATE OR REPLACE TRIGGER rental_trig
  BEFORE INSERT OR UPDATE OR DELETE ON rental
CALL time_check
/

CREATE OR REPLACE TRIGGER title_copy_trig
  BEFORE INSERT OR UPDATE OR DELETE ON title_copy
CALL time_check
/

CREATE OR REPLACE TRIGGER title_trig
  BEFORE INSERT OR UPDATE OR DELETE ON title
CALL time_check
/
```

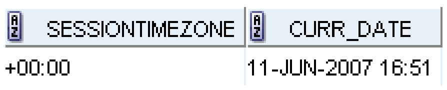
Part B: Additional Practice 5 Solutions (continued)

```
CREATE OR REPLACE TRIGGER reservation_trig
  BEFORE INSERT OR UPDATE OR DELETE ON reservation
CALL time_check
/

TRIGGER member_trig Compiled.
TRIGGER rental_trig Compiled.
TRIGGER title_copy_trig Compiled.
TRIGGER title_trig Compiled.
TRIGGER reservation_trig Compiled.
```

c. Test your triggers.

Note: In order for your trigger to fail, you may need to change the time to be outside the range of your current time in class. For example, while testing, you may want valid video hours in your trigger to be from 6:00 PM through 8:00 AM.

```
-- First determine current timezone and time
SELECT SESSIONTIMEZONE,
       TO_CHAR(CURRENT_DATE, 'DD-MON-YYYY HH24:MI') CURR_DATE
FROM DUAL;
```



SESSIONTIMEZONE	CURR_DATE
+00:00	11-JUN-2007 16:51

```
-- Change your time zone usinge [+|-]HH:MI format such that the current
-- time returns a time between 6pm and 8am
ALTER SESSION SET TIME_ZONE='-07:00';

ALTER SESSION SET succeeded.
```

Part B: Additional Practice 5 Solutions (continued)

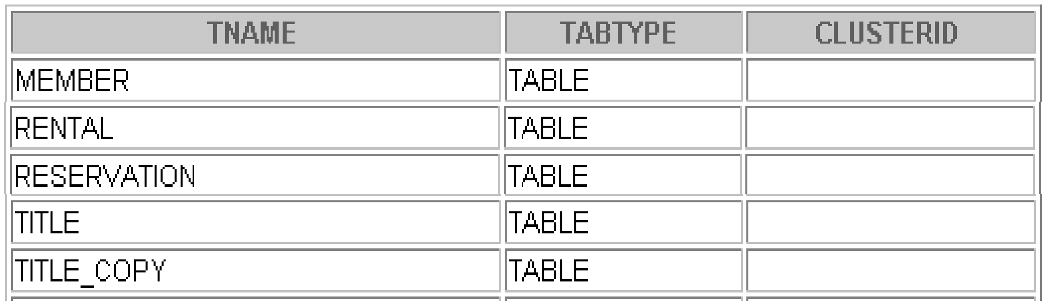
```
-- Add a new member (for a sample test)
EXECUTE  video_pkg.new_member('Elias', 'Elliane',  'Vine Street',
'California', '789-123-4567')

BEGIN video_pkg.new_member('Elias', 'Elliane',  'Vine Street',
'California', '789-123-4567'); END;

*

ERROR at line 1:
ORA-20999: Unhandled error in NEW_MEMBER. Please contact your application
administrator with the following information: ORA-20999: Data changes
restricted to office hours.
ORA-06512: at "ORA1.TIME_CHECK", line 9
ORA-06512: at "ORA1.MEMBER_TRIG", line 1
ORA-04088: error during execution of trigger 'ORA1.MEMBER_TRIG'
ORA-06512: at "ORA1.VIDEO_PKG", line 12
ORA-06512: at "ORA1.VIDEO_PKG", line 171
ORA-06512: at line 1

-- Restore the original time zone for your session.
ALTER SESSION SET TIME_ZONE='-00:00';

Session altered.
```

Additional Practices: Table Descriptions and Data

Part A

The tables and data used in part A are the same as those in Appendix B, “Table Descriptions and Data.”

Part B: Tables Used



TNAME	TABTYPE	CLUSTERID
MEMBER	TABLE	
RENTAL	TABLE	
RESERVATION	TABLE	
TITLE	TABLE	
TITLE_COPY	TABLE	

Part B: `MEMBER` Table

```
DESCRIBE member
```

Name	Null?	Type
MEMBER_ID	NOT NULL	NUMBER(10)
LAST_NAME	NOT NULL	VARCHAR2(25)
FIRST_NAME		VARCHAR2(25)
ADDRESS		VARCHAR2(100)
CITY		VARCHAR2(30)
PHONE		VARCHAR2(25)
JOIN_DATE	NOT NULL	DATE

```
SELECT * FROM member;
```

MEMBER_ID	LAST_NAME	FIRST_NAME	ADDRESS	CITY	PHONE	JOIN_DATE
101	Velasquez	Carmen	283 King Street	Seattle	587-99-6666	03-MAR-90
102	Ngao	LaDoris	5 Modrany	Bratislava	586-355-8882	08-MAR-90
103	Nagayama	Midori	68 Via Centrale	Sao Paolo	254-852-5764	17-JUN-91
104	Quick-To-See	Mark	6921 King Way	Lagos	63-559-777	07-APR-90
105	Ropeburn	Audry	86 Chu Street	Hong Kong	41-559-87	04-MAR-90
106	Urguhart	Molly	3035 Laurier Blvd.	Quebec	418-542-9988	18-JAN-91
107	Menchu	Roberta	Boulevard de Waterloo 41	Brussels	322-504-2228	14-MAY-90
108	Biri	Ben	398 High St.	Columbus	614-455-9863	07-APR-90
109	Catchpole	Antoinette	88 Alfred St.	Brisbane	616-399-1411	09-FEB-92

9 rows selected.

Part B: RENTAL Table

```
DESCRIBE rental
```

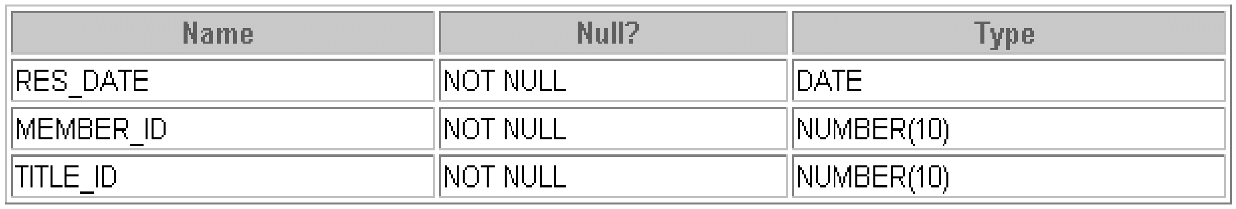
Name	Null?	Type
BOOK_DATE	NOT NULL	DATE
COPY_ID	NOT NULL	NUMBER(10)
MEMBER_ID	NOT NULL	NUMBER(10)
TITLE_ID	NOT NULL	NUMBER(10)
ACT_RET_DATE		DATE
EXP_RET_DATE		DATE

```
SELECT * FROM rental;
```

BOOK_DATE	COPY_ID	MEMBER_ID	TITLE_ID	ACT_RET_D	EXP_RET_D
02-OCT-01	2	101	93		04-OCT-01
01-OCT-01	3	102	95		03-OCT-01
30-SEP-01	1	101	98		02-OCT-01
29-SEP-01	1	106	97	01-OCT-01	01-OCT-01
30-SEP-01	1	101	92	01-OCT-01	02-OCT-01

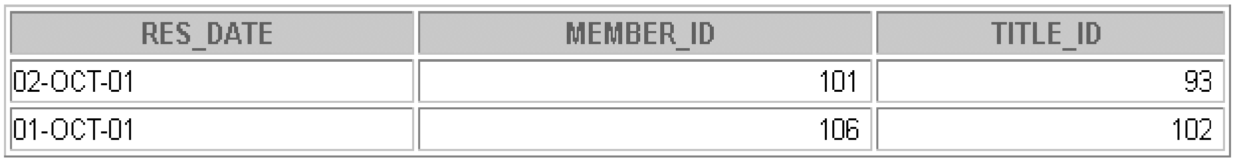
Part B: `RESERVATION` Table

```
DESCRIBE reservation
```



Name	Null?	Type
RES_DATE	NOT NULL	DATE
MEMBER_ID	NOT NULL	NUMBER(10)
TITLE_ID	NOT NULL	NUMBER(10)

```
SELECT * FROM reservation;
```



RES_DATE	MEMBER_ID	TITLE_ID
02-OCT-01	101	93
01-OCT-01	106	102

Part B: `TITLE` Table

```
DESCRIBE title
```

Name	Null?	Type
TITLE_ID	NOT NULL	NUMBER(10)
TITLE	NOT NULL	VARCHAR2(60)
DESCRIPTION	NOT NULL	VARCHAR2(400)
RATING		VARCHAR2(4)
CATEGORY		VARCHAR2(20)
RELEASE_DATE		DATE

```
SELECT * FROM title;
```

TITLE_ID	TITLE	DESCRIPTION	RATI	CATEGORY	RELEASE_D
92	Willie and Christmas Too	All of Willie's friends made a Christmas list for Santa, but Willie has yet to create his own wish list.	G	CHILD	05-OCT-95
93	Alien Again	Another installment of science fiction history. Can the heroine save the planet from the alien life form?	R	SCIFI	19-MAY-95
94	The Glob	A meteor crashes near a small American town and unleashes carivorous goo in this classic.	NR	SCIFI	12-AUG-95
95	My Day Off	With a little luck and a lot of ingenuity, a teenager skips school for a day in New York.	PG	COMEDY	12-JUL-95
96	Miracles on Ice	A six-year-old has doubts about Santa Claus. But she discovers that miracles really do exist.	PG	DRAMA	12-SEP-95
97	Soda Gang	After discovering a cached of drugs, a young couple find themselves pitted against a vicious gang.	NR	ACTION	01-JUN-95
98	Interstellar Wars	Futuristic interstellar action movie. Can the rebels save the humans from the evil Empire?	PG	SCIFI	07-JUL-77

7 rows selected.

Part B: `TITLE_COPY` Table

```
DESCRIBE title_copy
```

Name	Null?	Type
COPY_ID	NOT NULL	NUMBER(10)
TITLE_ID	NOT NULL	NUMBER(10)
STATUS	NOT NULL	VARCHAR2(15)

```
SELECT * FROM title_copy;
```

COPY_ID	TITLE_ID	STATUS
1	92	AVAILABLE
1	93	AVAILABLE
2	93	RENTED
1	94	AVAILABLE
1	95	AVAILABLE
2	95	AVAILABLE
3	95	RENTED
1	96	AVAILABLE
1	97	AVAILABLE
1	98	RENTED
2	98	AVAILABLE

11 rows selected.