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

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

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.

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

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

4. In this exercise, create a procedure to monitor whether employees have exceeded their average salaries for their job type.

a. 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.

b. Write a local function called GET_JOB_AVGSAL to determine the average salary for a job ID specified as a parameter. The average salary for a job is calculated from the information in the JOBS table.

c. Write a stored procedure called CHECK_AVGSAL that checks whether each employee's salary exceeds the average salary for the JOB_ID.

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.

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.

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.