4

# Interacting with
# the Oracle Server

ORACLE

# Objectives

**After completing this lesson, you should be able to do the following:**

- **Determine which SQL statements can be directly included in a PL/SQL executable block**
- **Manipulate data with DML statements in PL/SQL**
- **Use transaction control statements in PL/SQL**
- **Make use of the INTO clause to hold the values returned by a SQL statement**
- **Differentiate between implicit cursors and explicit cursors**
- **Use SQL cursor attributes**

ORACLE

**Lesson Aim**

In this lesson, you learn to embed standard SQL SELECT, INSERT, UPDATE, DELETE, and MERGE statements in PL/SQL blocks. You learn how to include data definition language (DDL) and transaction control statements in PL/SQL. You learn the need for cursors and differentiate between the two types of cursors. The lesson also presents the various SQL cursor attributes that can be used with implicit cursors.

# SQL Statements in PL/SQL

- **Retrieve a row from the database by using the SELECT command.**
- **Make changes to rows in the database by using DML commands.**
- **Control a transaction with the COMMIT, ROLLBACK, or SAVEPOINT command.**

ORACLE

**SQL Statements in PL/SQL**

In a PL/SQL block, you use SQL statements to retrieve and modify data from the database table. PL/SQL supports data manipulation language (DML) and transaction control commands. You can use DML commands to modify the data in a database table. However, remember the following points while using DML statements and transaction control commands in PL/SQL blocks:

- The keyword END signals the end of a PL/SQL block, not the end of a transaction. Just as a block can span multiple transactions, a transaction can span multiple blocks.
- PL/SQL does not directly support data definition language (DDL) statements, such as CREATE TABLE, ALTER TABLE, or DROP TABLE. PL/SQL supports early binding; as a result, compilation time is greater than execution time. If applications have to create database objects at run time by passing values, then early binding cannot happen in such cases. DDL statements cannot be directly executed. These statements are dynamic SQL statements. Dynamic SQL statements are built as character strings at run time and can contain placeholders for parameters. Therefore, you can use dynamic SQL to execute your DDL statements in PL/SQL. Use the EXECUTE IMMEDIATE statement, which takes the SQL statement as an argument to execute your DDL statement. The EXECUTE IMMEDIATE statement parses and executes a dynamic SQL statement.

## SQL Statements in PL/SQL (continued)

Consider the following example:

```
BEGIN
CREATE TABLE My_emp_table AS SELECT * FROM employees;
END;
/
```

The example uses a DDL statement directly in the block. When you execute the block, you see the following error:

```
create table My_table as select * from table_name; * ERROR
at line 5:
ORA-06550: line 5, column 1:
PLS-00103: Encountered the symbol "CREATE" when expecting
one of the following:
…
```

Use the EXECUTE IMMEDIATE statement to avoid the error:

```
BEGIN
EXECUTE IMMEDIATE 'CREATE TABLE My_emp_table AS SELECT *
FROM employees';
END;
/
```

- PL/SQL does not support data control language (DCL) statements such as GRANT or REVOKE. You can use EXECUTE IMMEDIATE statement to execute them.
- You use transaction control statements to make the changes to the database permanent or to discard them. COMMIT, ROLLBACK, and SAVEPOINT are three main transactional control statements that are used. COMMIT is used to make the database changes permanent. ROLLBACK is for discarding any changes that were made to the database after the last COMMIT. SAVEPOINT is used to mark an intermediate point in transaction processing. The transaction control commands are valid in PL/SQL and therefore can be directly used in the executable section of a PL/SQL block.

# SELECT Statements in PL/SQL

**Retrieve data from the database with a SELECT statement.**

**Syntax:**

```
SELECT    select_list
INTO      {variable_name[, variable_name]...
          | record_name}
FROM      table
[WHERE    condition];
```

## SELECT Statements in PL/SQL

Use the SELECT statement to retrieve data from the database.

| | |
|---|---|
| *select_list* | List of at least one column; can include SQL expressions, row functions, or group functions |
| *variable_name* | Scalar variable that holds the retrieved value |
| *record_name* | PL/SQL record that holds the retrieved values |
| *table* | Specifies the database table name |
| *condition* | Is composed of column names, expressions, constants, and comparison operators, including PL/SQL variables and constants |

### Guidelines for Retrieving Data in PL/SQL

- Terminate each SQL statement with a semicolon (;).
- Every value retrieved must be stored in a variable using the INTO clause.
- The WHERE clause is optional and can be used to specify input variables, constants, literals, and PL/SQL expressions. However, when you use the INTO clause, you should fetch only one row; using the WHERE clause is required in such cases.

## `SELECT` Statements in PL/SQL (continued)

- Specify the same number of variables in the `INTO` clause as the number of database columns in the `SELECT` clause. Be sure that they correspond positionally and that their data types are compatible.
- Use group functions, such as `SUM`, in a SQL statement, because group functions apply to groups of rows in a table.

# SELECT Statements in PL/SQL

- **The INTO clause is required.**
- **Queries must return only one row.**

## Example

```
SET SERVEROUTPUT ON
DECLARE
  fname VARCHAR2(25);
BEGIN
  SELECT first_name INTO fname
  FROM employees WHERE employee_id=200;
  DBMS_OUTPUT.PUT_LINE(' First Name is : '||fname);
END;
/
```

ORACLE

### SELECT Statements in PL/SQL (continued)

#### INTO Clause

The INTO clause is mandatory and occurs between the SELECT and FROM clauses. It is used to specify the names of variables that hold the values that SQL returns from the SELECT clause. You must specify one variable for each item selected, and the order of the variables must correspond with the items selected.

Use the INTO clause to populate either PL/SQL variables or host variables.

#### Queries Must Return Only One Row

SELECT statements within a PL/SQL block fall into the ANSI classification of embedded SQL, for which the following rule applies: queries must return only one row. A query that returns more than one row or no row generates an error.

PL/SQL manages these errors by raising standard exceptions, which you can handle in the exception section of the block with the NO_DATA_FOUND and TOO_MANY_ROWS exceptions. Include a WHERE condition in the SQL statement so that the statement returns a single row. You learn about exception handling later in the course.

## SELECT Statements in PL/SQL (continued)

### How to Retrieve Multiple Rows from a Table and Operate on the Data

A SELECT statement with the INTO clause can retrieve only one row at a time. If your requirement is to retrieve multiple rows and operate on the data, you can make use of explicit cursors. You learn about cursors later in this lesson.

# Retrieving Data in PL/SQL

**Retrieve the `hire_date` and the `salary` for the specified employee.**

## Example

```
DECLARE
 emp_hiredate    employees.hire_date%TYPE;
 emp_salary      employees.salary%TYPE;
BEGIN
  SELECT   hire_date, salary
  INTO     emp_hiredate, emp_salary
  FROM     employees
  WHERE    employee_id = 100;
END;
/
```

ORACLE

**Retrieving Data in PL/SQL**

In the example in the slide, the variables emp_hiredate and emp_salary are declared in the declarative section of the PL/SQL block. In the executable section, the values of the columns hire_date and salary for the employee with the employee_id 100 are retrieved from the employees table; they are stored in the emp_hiredate and emp_salary variables, respectively. Observe how the INTO clause, along with the SELECT statement, retrieves the database column values into the PL/SQL variables.

**Note:** The SELECT statement retrieves hire_date and then salary. The variables in the INTO clause must thus be in the same order. For example, if you exchange emp_hiredate and emp_salary in the statement in the slide, the statement results in an error.

# Retrieving Data in PL/SQL

**Return the sum of the salaries for all the employees in the specified department.**

## Example

```
SET SERVEROUTPUT ON
DECLARE
   sum_sal  NUMBER(10,2);
   deptno   NUMBER NOT NULL := 60;
BEGIN
   SELECT  SUM(salary)  -- group function
   INTO sum_sal FROM employees
   WHERE  department_id = deptno;
   DBMS_OUTPUT.PUT_LINE ('The sum of salary is '
   || sum_sal);
END;
/
```

**ORACLE**

**Retrieving Data in PL/SQL (continued)**

In the example in the slide, the sum_sal and deptno variables are declared in the declarative section of the PL/SQL block. In the executable section, the total salary for the employees in the department with the department_id 60 is computed using the SQL aggregate function SUM.  The calculated total salary is assigned to the sum_sal variable.

**Note:** Group functions cannot be used in PL/SQL syntax. They are used in SQL statements within a PL/SQL block as shown in the example. You cannot use them as follows:

         sum_sal := SUM(employees.salary);

The output of the PL/SQL block in the slide is the following:

```
The sum of salary is 28800
PL/SQL procedure successfully completed.
```

# Naming Conventions

```
DECLARE
  hire_date        employees.hire_date%TYPE;
  sysdate          hire_date%TYPE;
  employee_id      employees.employee_id%TYPE := 176;
BEGIN
  SELECT      hire_date, sysdate
  INTO        hire_date, sysdate
  FROM        employees
  WHERE       employee_id = employee_id;
END;
/
```

```
DECLARE
*
ERROR at line 1:
ORA-01422: exact fetch returns more than requested number of rows
ORA-06512: at line 6
```

ORACLE

## Naming Conventions

In potentially ambiguous SQL statements, the names of database columns take precedence over the names of local variables.

The example shown in the slide is defined as follows: Retrieve the hire date and today's date from the employees table for employee_id 176. This example raises an unhandled run-time exception because in the WHERE clause, the PL/SQL variable names are the same as the database column names in the employees table.

The following DELETE statement removes all employees from the employees table where the last name is not null (not just "King") because the Oracle server assumes that both occurrences of last_name in the WHERE clause refer to the database column:

```
DECLARE
    last_name VARCHAR2(25) := 'King';
BEGIN
    DELETE FROM employees WHERE last_name = last_name;
. . .
```

# Naming Conventions

- **Use a naming convention to avoid ambiguity in the WHERE clause.**
- **Avoid using database column names as identifiers.**
- **Syntax errors can arise because PL/SQL checks the database first for a column in the table.**
- **The names of local variables and formal parameters take precedence over the names of database *tables*.**
- **The names of database table *columns* take precedence over the names of local variables.**

ORACLE

**Naming Conventions (continued)**

Avoid ambiguity in the WHERE clause by adhering to a naming convention that distinguishes database column names from PL/SQL variable names.

- Database columns and identifiers should have distinct names.
- Syntax errors can arise because PL/SQL checks the database first for a column in the table.

**Note:** There is no possibility for ambiguity in the SELECT clause because any identifier in the SELECT clause must be a database column name. There is no possibility for ambiguity in the INTO clause because identifiers in the INTO clause must be PL/SQL variables. There is the possibility of confusion only in the WHERE clause.

# Manipulating Data Using PL/SQL

**Make changes to database tables by using DML commands:**

- **INSERT**
- **UPDATE**
- **DELETE**
- **MERGE**
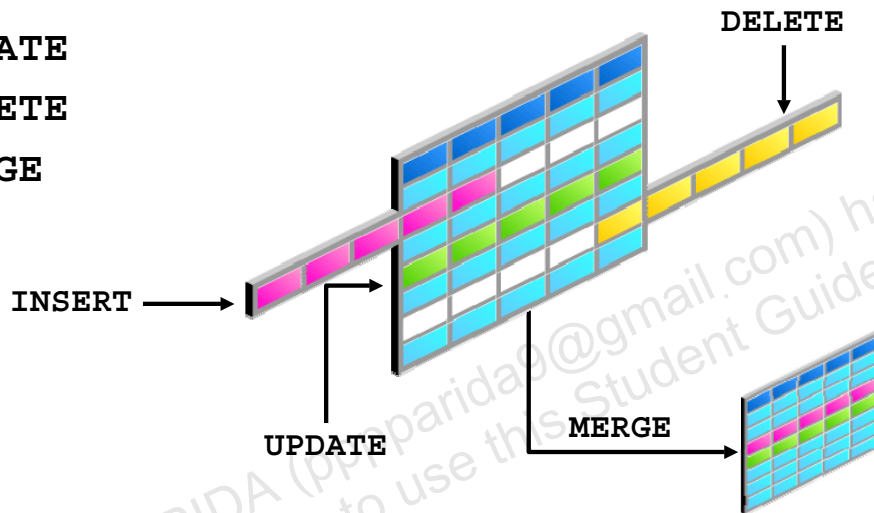
DELETE

INSERT

UPDATE     MERGE

**Manipulating Data Using PL/SQL**

You manipulate data in the database by using the DML commands. You can issue the DML commands INSERT, UPDATE, DELETE and MERGE without restriction in PL/SQL. Row locks (and table locks) are released by including COMMIT or ROLLBACK statements in the PL/SQL code.

- The INSERT statement adds new rows to the table.
- The UPDATE statement modifies existing rows in the table.
- The DELETE statement removes rows from the table.
- The MERGE statement selects rows from one table to update or insert into another table. The decision whether to update or insert into the target table is based on a condition in the ON clause.

**Note:** MERGE is a deterministic statement. That is, you cannot update the same row of the target table multiple times in the same MERGE statement. You must have INSERT and UPDATE object privileges in the target table and the SELECT privilege on the source table.

# Inserting Data

**Add new employee information to the EMPLOYEES table.**

**Example**

```
BEGIN
 INSERT INTO employees
  (employee_id, first_name, last_name, email,
   hire_date, job_id, salary)
   VALUES(employees_seq.NEXTVAL, 'Ruth', 'Cores',
   'RCORES',sysdate, 'AD_ASST', 4000);
END;
/
```

ORACLE

**Inserting Data**

In the example in the slide, an INSERT statement is used within a PL/SQL block to insert a record into the employees table. While using the INSERT command in a PL/SQL block, you can:

- Use SQL functions, such as USER and SYSDATE
- Generate primary key values by using existing database sequences
- Derive values in the PL/SQL block

**Note:** The data in the employees table needs to remain unchanged. Inserting, updating, and deleting are thus not allowed on this table.

# Updating Data

**Increase the salary of all employees who are stock clerks.**

**Example**

```
DECLARE
  sal_increase    employees.salary%TYPE := 800;
BEGIN
  UPDATE      employees
  SET         salary = salary + sal_increase
  WHERE       job_id = 'ST_CLERK';
END;
/
```

ORACLE

## Updating Data

There may be ambiguity in the SET clause of the UPDATE statement because, although the identifier on the left of the assignment operator is always a database column, the identifier on the right can be either a database column or a PL/SQL variable. Recall that if column names and identifier names are identical in the WHERE clause, the Oracle server looks to the database first for the name.

Remember that the WHERE clause is used to determine which rows are affected. If no rows are modified, no error occurs (unlike the SELECT statement in PL/SQL).

**Note:** PL/SQL variable assignments always use :=, and SQL column assignments always use =.

# Deleting Data

**Delete rows that belong to department 10 from the `employees` table.**

## Example

```
DECLARE
  deptno    employees.department_id%TYPE := 10;
BEGIN
  DELETE FROM    employees
  WHERE   department_id = deptno;
END;
/
```

## Deleting Data

The DELETE statement removes unwanted rows from a table. If the WHERE clause is not used, all the rows in a table can be removed if there are no integrity constraints.

# Merging Rows

**Insert or update rows in the `copy_emp` table to match the `employees` table.**

```
DECLARE
      empno employees.employee_id%TYPE := 100;
BEGIN
MERGE INTO copy_emp c
     USING employees e
     ON (e.employee_id = c.empno)
   WHEN MATCHED THEN
     UPDATE SET
       c.first_name      = e.first_name,
       c.last_name       = e.last_name,
       c.email           = e.email,
       . . .
   WHEN NOT MATCHED THEN
     INSERT VALUES(e.employee_id, e.first_name, e.last_name,
         . . .,e.department_id);
END;
/
```

ORACLE

**Merging Rows**

The MERGE statement inserts or updates rows in one table by using data from another table. Each row is inserted or updated in the target table depending on an equijoin condition.

The example shown matches the employee_id in the COPY_EMP table to the employee_id in the employees table. If a match is found, the row is updated to match the row in the employees table. If the row is not found, it is inserted into the copy_emp table.

The complete example for using MERGE in a PL/SQL block is shown on the next notes page.

## Merging Rows (continued)

```
DECLARE
     empno EMPLOYEES.EMPLOYEE_ID%TYPE := 100;
BEGIN
MERGE INTO copy_emp c
     USING employees e
     ON (e.employee_id = c.empno)
   WHEN MATCHED THEN
     UPDATE SET
       c.first_name     = e.first_name,
       c.last_name      = e.last_name,
       c.email          = e.email,
       c.phone_number   = e.phone_number,
       c.hire_date      = e.hire_date,
       c.job_id         = e.job_id,
       c.salary         = e.salary,
       c.commission_pct = e.commission_pct,
       c.manager_id     = e.manager_id,
       c.department_id  = e.department_id
   WHEN NOT MATCHED THEN
     INSERT VALUES(e.employee_id,  e.first_name, e.last_name,
          e.email, e.phone_number, e.hire_date, e.job_id,
          e.salary, e.commission_pct, e.manager_id,
          e.department_id);
END;
/
```

# SQL Cursor

- **A cursor is a pointer to the private memory area allocated by the Oracle server.**
- **There are two types of cursors:**
  - **Implicit: Created and managed internally by the Oracle server to process SQL statements**
  - **Explicit: Explicitly declared by the programmer**

ORACLE

## SQL Cursor

You have already learned that you can include SQL statements that return a single row in a PL/SQL block. The data retrieved by the SQL statement should be held in variables using the INTO clause.

### Where Does Oracle Process SQL Statements?

The Oracle server allocates a private memory area called the *context area* for processing SQL statements. The SQL statement is parsed and processed in this area. Information required for processing and information retrieved after processing is all stored in this area. You have no control over this area because it is internally managed by the Oracle server.

A cursor is a pointer to the context area. However, this cursor is an implicit cursor and is automatically managed by the Oracle server. When the executable block issues a SQL statement, PL/SQL creates an implicit cursor.

There are two types of cursors:
- **Implicit:** Implicit cursors are created and managed by the Oracle server. You do not have access to them. The Oracle server creates such a cursor when it has to execute a SQL statement.

## SQL Cursor (continued)

- **Explicit:** As a programmer you may want to retrieve multiple rows from a database table, have a pointer to each row that is retrieved, and work on the rows one at a time. In such cases, you can declare cursors explicitly depending on your business requirements. Cursors that are declared by programmers are called *explicit cursors*. You declare these cursors in the declarative section of a PL/SQL block. Remember that you can also declare variables and exceptions in the declarative section.

# SQL Cursor Attributes for Implicit Cursors

**Using SQL cursor attributes, you can test the outcome of your SQL statements.**

| `SQL%FOUND` | Boolean attribute that evaluates to TRUE if the most recent SQL statement returned at least one row |
|---|---|
| `SQL%NOTFOUND` | Boolean attribute that evaluates to TRUE if the most recent SQL statement did not return even one row |
| `SQL%ROWCOUNT` | An integer value that represents the number of rows affected by the most recent SQL statement |

ORACLE

**SQL Cursor Attributes for Implicit Cursors**

SQL cursor attributes enable you to evaluate what happened when an implicit cursor was last used. Use these attributes in PL/SQL statements but not in SQL statements.

You can test the attributes SQL%ROWCOUNT, SQL%FOUND, and SQL%NOTFOUND in the executable section of a block to gather information after the appropriate DML command. PL/SQL does not return an error if a DML statement does not affect rows in the underlying table. However, if a SELECT statement does not retrieve any rows, PL/SQL returns an exception.

Observe that the attributes are prefixed with SQL. These cursor attributes are used with implicit cursors that are automatically created by PL/SQL and for which you do not know the names. Therefore, you use SQL instead of the cursor name.

The SQL%NOTFOUND attribute is opposite to SQL%FOUND. This attribute may be used as the exit condition in a loop. It is useful in UPDATE and DELETE statements when no rows are changed because exceptions are not returned in these cases.

You learn about explicit cursor attributes later in the course.

# SQL Cursor Attributes for Implicit Cursors

**Delete rows that have the specified employee ID from the `employees` table. Print the number of rows deleted.**

**Example**

```
VARIABLE rows_deleted VARCHAR2(30)
DECLARE
  empno employees.employee_id%TYPE := 176;
BEGIN
  DELETE FROM  employees
  WHERE employee_id = empno;
  :rows_deleted := (SQL%ROWCOUNT ||
                         ' row deleted.');
END;
/
PRINT rows_deleted
```

ORACLE

**SQL Cursor Attributes for Implicit Cursors (continued)**

The example in the slide deletes a row with `employee_id` 176 from the `employees` table. Using the `SQL%ROWCOUNT` attribute, you can print the number of rows deleted.

# Summary

**In this lesson, you should have learned how to:**
- **Embed DML statements, transaction control statements, and DDL statements in PL/SQL**
- **Use the `INTO` clause, which is mandatory for all `SELECT` statements in PL/SQL**
- **Differentiate between implicit cursors and explicit cursors**
- **Use SQL cursor attributes to determine the outcome of SQL statements**

ORACLE

**Summary**

The DML commands and transaction control statements can be used in PL/SQL programs without restriction. However, the DDL commands cannot be used directly.

A `SELECT` statement in PL/SQL block can return only one row. It is mandatory to use the `INTO` clause to hold the values retrieved by the `SELECT` statement.

A cursor is a pointer to the memory area. There are two types of cursors. Implicit cursors are created and managed internally by the Oracle server to execute SQL statements. You can use SQL cursor attributes with these cursors to determine the outcome of the SQL statement. Explicit cursors are declared by programmers.

# Practice 4: Overview

**This practice covers the following topics:**

- **Selecting data from a table**
- **Inserting data into a table**
- **Updating data in a table**
- **Deleting a record from a table**

ORACLE

**Practice 4**

Note: It is recommended to use *i*SQL*Plus for this practice.

1. Create a PL/SQL block that selects the maximum department ID in the departments table and stores it in the max_deptno variable. Display the maximum department ID.

   a. Declare a variable max_deptno of type NUMBER in the declarative section.

   b. Start the executable section with the keyword BEGIN and include a SELECT statement to retrieve the maximum department_id from the departments table.

   c. Display max_deptno and end the executable block.

   d. Execute and save your script as lab_04_01_soln.sql. Sample output is

   The maximum department_id is : 270
   PL/SQL procedure successfully completed.

2. Modify the PL/SQL block you created in exercise 1 to insert a new department into the departments table.

   a. Load the script lab_04_01_soln.sql. Declare two variables:
   dept_name of type departments.department_name.
   Bind variable dept_id of type NUMBER.
   Assign 'Education' to dept_name in the declarative section.

   b. You have already retrieved the current maximum department number from the departments table. Add 10 to it and assign the result to dept_id.

   c. Include an INSERT statement to insert data into the department_name, department_id, and location_id columns of the departments table. Use values in dept_name, dept_id for department_name, department_id and use NULL for location_id.

   d. Use the SQL attribute SQL%ROWCOUNT to display the number of rows that are affected.

   e. Execute a select statement to check if the new department is inserted. You can terminate the PL/SQL block with "/" and include the SELECT statement in your script.

   f. Execute and save your script as lab_04_02_soln.sql. Sample output is

   The maximum department_id is : 270
   SQL%ROWCOUNT gives 1
   PL/SQL procedure successfully completed.

| DEPARTMENT_ID | DEPARTMENT_NAME | MANAGER_ID | LOCATION_ID |
|---|---|---|---|
| 280 | Education | | |

**Practice 4 (continued)**

3. In exercise 2, you have set `location_id` to null. Create a PL/SQL block that updates the `location_id` to 3000 for the new department. Use the bind variable `dept_id` to update the row.

   **Note:** Skip step a if you have not started a new *i*SQL*Plus session for this practice.

   a. If you have started a new *i*SQL*Plus session, delete the department that you have added to the `departments` table and execute the script `lab_04_02_soln.sql`.

   b. Start the executable block with the keyword `BEGIN`. Include the `UPDATE` statement to set the `location_id` to 3000 for the new department. Use the bind variable `dept_id` in your `UPDATE` statement.

   c. End the executable block with the keyword `END`. Terminate the PL/SQL block with "/" and include a `SELECT` statement to display the department that you updated.

   d. Finally, include a `DELETE` statement to delete the department that you added.

   e. Execute and save your script as `lab_04_03_soln.sql`. Sample output is shown below.

   PL/SQL procedure successfully completed.

   | DEPARTMENT_ID | DEPARTMENT_NAME | MANAGER_ID | LOCATION_ID |
   |---|---|---|---|
   | 280 | Education | | 3000 |

   1 row deleted.

4. Load the script `lab_03_05b.sql` to the *i*SQL*Plus workspace.

   a. Observe that the code has nested blocks. You will see the declarative section of the outer block. a. Look for the comment "INCLUDE EXECUTABLE SECTION OF OUTER BLOCK HERE" and start an executable section

   b. Include a single `SELECT` statement, which retrieves the `employee_id` of the employee working in the 'Human Resources' department. Use the `INTO` clause to store the retrieved value in the variable `emp_authorization`.

   c. Save your script as `lab_04_04_soln.sql`.