CSE 4410

DATABASE MANAGEMENT SYSTEMS II LAB

SWE 21

CSE
IUT

Contents

Lab 1 Introduction

Welcome to CSE 4410.

1 Marks Distribution

Module	Mark (%)
Attendance	10
Lab Evaluation	40
Lab Report	20
Project	30

2 Approximate Course Outline

1. (Intro) + Basics of Relational Database Model
2. Tablespace
3. JDBC Connection + (Project Proposal Submission)
4. PL/SQL
    a. Function/Procedure
    b. Cursor
    c. Trigger
5. Project Progress Presentation
6. NoSQL [MongoDB]
    a. Theory
    b. Sessional
7. Graph-based Database [Neo4j]
    a. Theory
    b. Sessional
8. Project Presentation

3 Task - Group B

Consider the schema shown in Figure 1.1 for the database of a university:

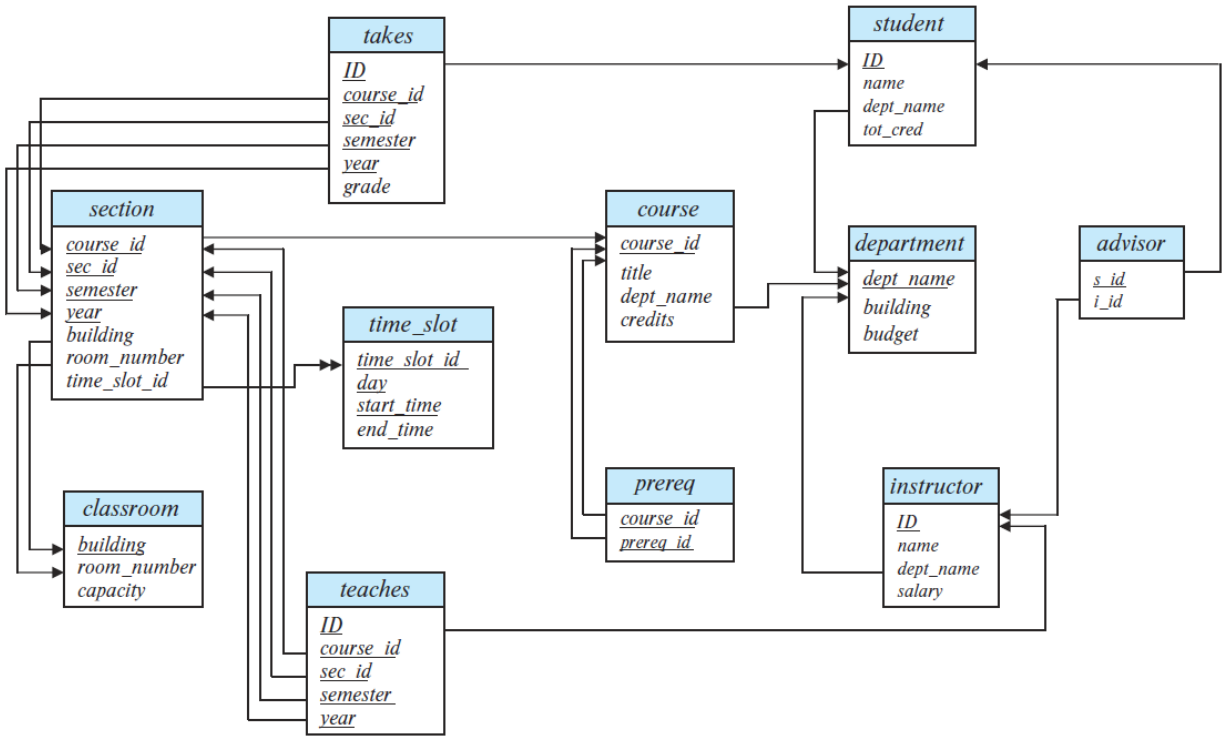

Figure 1.1. Schema diagram for a university database

Write the command `@"<file_path>\<file_name>.sql"` in your SQL command line to execute the provided .sql files. Now, write SQL statements to answer each of the following queries:

1. Find the names of all the instructors from the 'Biology' department.
2. Show the Course ID and the Title of all the courses registered for by the student with ID '12345'.
3. Find the names and department names of all the students who have taken a course offered by the 'Comp. Sci.' department.
4. Find the names of the students who take the 'CS-101' course in 'Spring, 2018'.
5. Find the names of students who have taken the highest number of courses with a specific prefix 'CS'.
6. Find the names of students who have taken courses taught by at least three different instructors
7. Find the course name and section having the minimum number of enrollments. Do not include the sections that do not have any students enrolled.
8. Find the name of the instructor, dept_name, and count of students he/she advising. If an instructor is not advising any student, show 0.
9. Find the name and department of the students who take more courses than the average number of courses taken by a student.
10. Insert each instructor as a student with total credit set to 0 in the same department they are teaching.
11. Remove all the newly added students from the previous query.
12. Update the 'tot_cred' for each student based on the credits taken.
13. Update the salary of each instructor to 10000 times the number of course sections they have taught.
14. Grades are mapped to a grade point as follows: A:10, B:8, C:6, D:4, and F:0. Create a table to store these mappings, and write a query to find the Credit Point Information (CPI) of each student, using this table. Make sure students who have not got a non-null grade in any course are displayed with a CPI of null.

4 Task - Group A

Consider the schema shown in Figure 1.2 for the database of a university:

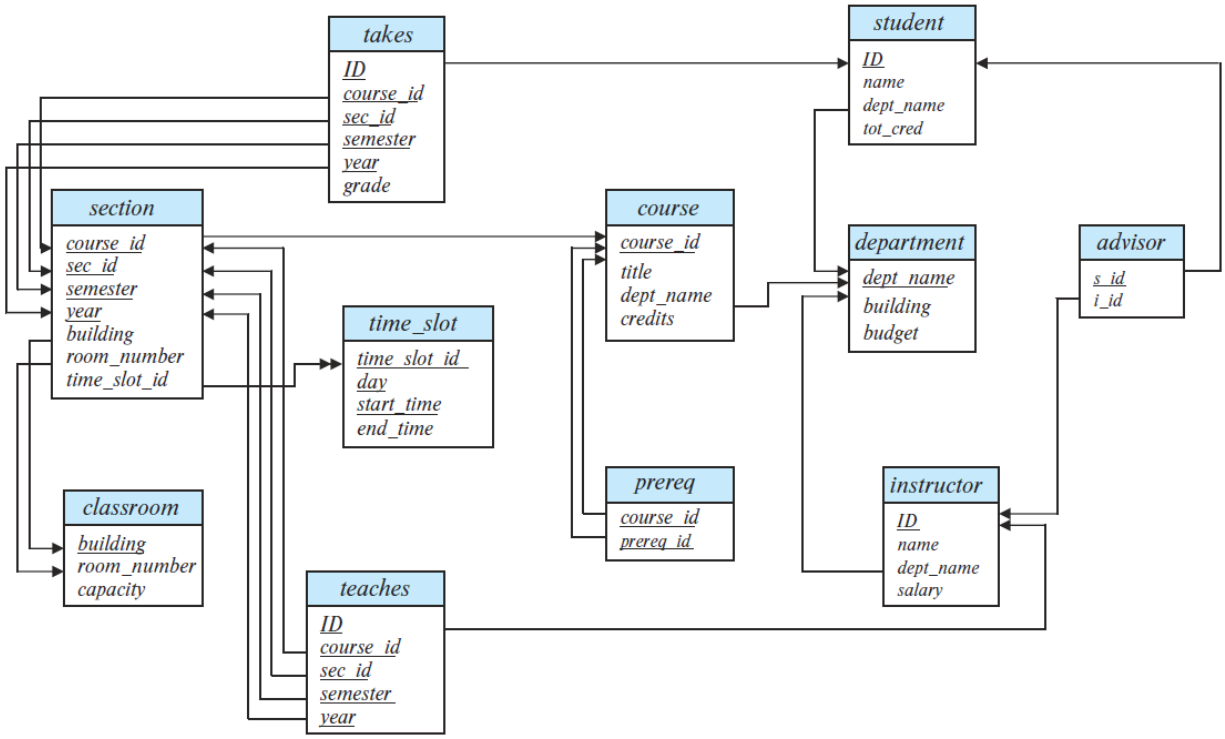

Figure 1.2. Schema diagram for a university database

Write the command `@"<file_path>\<file_name>.sql"` in your SQL command line to execute the provided .sql files. Now, write SQL statements to answer each of the following queries:

1. Find the names of courses offered by the 'Comp. Sci.' department which has 3 credits.
2. For each student, list their ID, name, and total credits s/he has taken. Do not include the students who did not register for any course.
3. Find the names and the department names of all instructors who have not taught a course.
4. Find all the course titles that do not have any prerequisites.
5. Find the name of the student who takes 2nd, 3rd, and 5th maximum total credits.
6. Find the names of the instructors who are taking courses with no students enrolled. Also, show the name of the courses.
7. Retrieve the course titles and the percentage of students who earned an 'A' grade in each course.
8. Find the number of instructors who have taught the same course in consecutive years.
9. Insert each student as a student with total credit set to 0 in the same department they are teaching.
10. Update the 'tot_cred' for each student based on the credits taken.
11. Update the salary of each instructor to 10000 times the number of course sections they have taught.
12. Find all rooms that have been assigned to more than one section at the same time.
13. Create a view that will show the instructor-wise time slot for 'Fall, 2017' sorted by the instructor_Id, course_ID, section_ID (Instructor_ID, name, his/her course information, section_ID, count of students in that section for the course, and time_slot).

Lab 2 Tablespace

A tablespace is a database storage unit that groups related logical structures together. The database data files are stored in tablespaces. A data file physically stores the data objects of the database such as tables and indices on disk. Using multiple tablespaces allows more flexibility in performing database operations.

- To enhance performance, segregate user data from data dictionary data. This minimizes conflicts over I/O resources.
- Prevent cross-application impact by keeping the data of each application separate. This safeguards against disruptions if a tablespace needs to be temporarily disabled.
- Enhance efficiency by storing data files from different tablespaces on different disk drives. This reduces contention for I/O resources.
- Increase overall system availability by selectively taking individual tablespaces offline while keeping others operational. This minimizes disruptions.
- Tailor tablespace allocation to specific database needs, such as high update activity, read-only activity, or temporary segment storage. This optimizes overall database performance.
- Enhance data security and recovery capabilities by conducting backups at the tablespace level. This facilitates targeted recovery and maintenance efforts.

Some operating systems set a limit on the number of files that can be opened simultaneously (For example, 512 in Windows, 1024 in Ubuntu, 4096 in CentOS, etc.). Efficient tablespace planning is essential to avoid surpassing the operating system limit. It is advisable to create tablespaces based on actual needs, keeping the number of tablespaces to a minimum. When expanding a tablespace, rather than creating numerous small data files, adding one or two substantial data files or opting for data files with autoextension enabled is recommended.

1 Default Tablespaces

Oracle comes with 5 default tablespaces:

- The primary tablespace in any database is the SYSTEM tablespace, which contains information basic to the functioning of the database server, such as the data dictionary and the system rollback segment. The SYSTEM tablespace is the first tablespace created at database creation. It is managed as any other tablespace but requires a higher level of privilege and is restricted in some ways. For example, it is not possible to rename or drop the SYSTEM tablespace or take it offline.
- The SYSAUX tablespace, which acts as an auxiliary tablespace to SYSTEM tablespace, is also created during database creation. It contains the schemas used by various Oracle products and features so that those products do not require their own tablespace. The management of the SYSAUX tablespace is similar to that of the SYSTEM tablespace.
- USERS is a permanent tablespace containing the application data. Oracle fills this space with the data created and entered by the users.
- UNDOTBS1 is an auto-extending tablespace containing the undo data. Oracle provides a fully automated mechanism, referred to as automatic undo management, for managing undo information and space. With automatic undo management, the database manages undo segments in an undo tablespace.
- TEMP is the temporary tablespace that is used for storing intermediate results. Oracle uses it as work areas for tasks such as sort operations for users and sorting during index creation. Oracle does not allow users to create objects in a temporary tablespace. By definition, the temporary tablespace holds data only for the duration of a user's session, and the data can be shared by all users.

2 Create Tablespace

Using **CREATE TABLESPACE** statement, a new tablespace can be created. As we have seen a tablespace consisting of one or more data files, we need to specify the path of the data files as well as their size.

```
CREATE TABLESPACE TBS1
   DATAFILE 'tbs1_data.dbf' SIZE 1M,
            'tbs2_data.dbf' SIZE 1M;
```

To allow extent management to be LOCAL, one can optionally add the statement `EXTENT MANAGEMENT LOCAL AUTOALLOCATE` or `EXTENT MANAGEMENT LOCAL UNIFORM SIZE size`. Locally Managed Tablespaces (LMT) have a bitmap of the blocks or groups of blocks, allowing them to track extent allocation without reference to the data dictionary. If UNIFORM is specified, all extents within the tablespace will be the same size, with 1M (which stands for 1MB) being the default extent size. The AUTOALLOCATE clause allows you to size the initial extent leaving Oracle to determine the optimum size for subsequent extents, with 64K being the minimum.
Once the tablespace is created, all the information about it is available in the DBA_DATA_FILES view.

```
SELECT TABLESPACE_NAME, FILE_NAME, BYTES/1024/1024 MB
    FROM DBA_DATA_FILES;
```

To assign a user to a specific tablespace, one can explicitly mention it at the time of user creation.

```
CREATE USER iutlearner
    IDENTIFIED BY test123
        DEFAULT TABLESPACE TBS1;
```

Usually, when we create any new table in Oracle, by default, that is placed in the USERS tablespace. However, to create a new table in a user-defined tablespace, one must add the name of the tablespace at the end of the **CREATE TABLE** statement.

```
CREATE TABLE T1
(
   ID INT,
   C1 VARCHAR2(32)
) TABLESPACE TBS1;
```

Then, if we want to check the free space of a certain tablespace, we can fetch that data from the DBA_FREE_SPACE view.

```
SELECT TABLESPACE_NAME, BYTES/1024/1024 MB
    FROM DBA_FREE_SPACE
        WHERE TABLESPACE_NAME='TBS1';
```

3 Extend Tablespace

Commonly, the tablespaces of the database get completely occupied. In that case, no further addition of data is possible. We have already learned about locally managed extent by the time on tablespace creation. But even if we forget to do that or do not want to automate that we can manually handle it using **ALTER TABLESPACE** statement. There are two ways of extension:

▸ By adding a new data file.

```
ALTER TABLESPACE TBS1
    ADD DATAFILE 'tbs3_data.dbf' SIZE 1M;
```

Here, if we use the **AUTOEXTEND ON** clause at the end of the code, Oracle will automatically extend the size of the data file as needed.

▸ By resizing the data file

```
ALTER DATABASE
    DATAFILE 'tbs1_data.dbf' RESIZE 15M;
```

4 Drop Tablespace

To remove a tablespace from the database, we use **DROP TABLESPACE** statement.

```
DROP TABLESPACE TBS1
    [INCLUDING CONTENTS [AND | KEEP] DATAFILES]
    [CASCADE CONSTRAINTS];
```

Here **INCLUDING CONTENTS** is necessary when there is any table created in the tablespace. Any attempt to remove a tablespace that has objects without specifying the clause will result in an error. If we do not use **AND DATAFILES**, it will by default keep the data files of those tables stored without any tablespace. And **CASCADE CONSTRAINTS** is necessary in case of referential integrity.

You can use the DROP TABLESPACE command to remove a tablespace regardless of whether it is online or offline. However, it is good practice to take the tablespace offline before removing it to ensure that no sessions are currently accessing any objects in the tablespace.

5 Read-Only or Read-Write Tablespace

The read-only tablespaces allow Oracle to avoid performing edits on large, static parts of a database. It allows you to remove objects such as tables and indexes from a read-only tablespace. However, it does not allow you to create or alter objects in a read-only tablespace.

```
ALTER TABLESPACE TBS1 READ ONLY;

ALTER TABLESPACE TBS1 READ WRITE;
```

By default, any newly created tablespace is in read-write mode.

6 Online and Offline Tablespace

Lastly, a tablespace can be online or offline. If a tablespace is offline, one cannot access data stored in it. On the other hand, if a tablespace is online, its data is available for reading and writing.

```
ALTER TABLESPACE TBS1 OFFLINE;

ALTER TABLESPACE TBS1 ONLINE;
```

Normally, a tablespace is online so that its data is available to users. However, we can take a tablespace offline to make data inaccessible to users when we update and maintain the applications.

7 Task - Group B

1. Create two tablespaces `tbs1` and `tbs2`.
2. Set quota for a single user on both tablespaces.
3. Create two tables `student(name, ID, fk[dept_ID])` and `department(ID, name)` in `tbs1`.
4. Create another table `course(course_code, name, credit, fk[offered_by_dept_ID])` in `tbs2`.
5. Insert a large amount of data in the `student` table and `course` table.
6. List the title and the name of the offering department of each course.
7. Check the free space of the tablespaces.
8. Extend `tbs1` by adding extra data files.
9. Extend `tbs2` by resizing data files.
10. Check the size of the tablespaces.
11. Set `tbs1` to offline and show that the data cannot be accessed.
12. Delete tablespace `tbs1` including the data files.
13. Delete tablespace `tbs2` excluding the data files.

8 Task - Group A

1. Create two tablespaces `tsp1` and `tsp2`.
2. Set quota for a single user on both tablespaces.
3. Create two tables `author(name, ID)` and `publisher(ID, name)` in `tsp1`.
4. Create another table `book(ISBN_code, name, price, fk[publisher_ID])` in `tsp2`.
5. Insert a large amount of data in the `book` table and `publisher` table.
6. List the title and the name of the publisher of each book.
7. Check the free space of the tablespaces.
8. Extend `tsp1` by adding extra data files.
9. Extend `tsp2` by resizing data files.
10. Check the size of the tablespaces.
11. Set `tsp1` to offline and show that the data cannot be accessed.
12. Delete tablespace `tsp1` including the data files.
13. Delete tablespace `tsp2` excluding the data files.

Lab 3 Java Database Connectivity (JDBC)

Java Database Connectivity (JDBC) is a Java API facilitating the interaction with relational databases. It provides a standard interface for connecting to databases, executing SQL queries, manipulating data, and handling the results. Prior to JDBC, ODBC was used but had dependencies on C-language drivers. Later JDBC introduced its own API, employing Java-based JDBC drivers.

The JDBC driver serves as a bridge, allowing the Java application to send queries, receive results, and generally interact with the database management system seamlessly. The driver essentially translates Java calls into a format that the database understands and vice versa, enabling effective and standardized communication between the Java application and the underlying database.

There are four types of drivers. The `JDBC-ODBC bridge driver`, the `Native-API driver` (partially Java driver), the `Network Protocol driver` (fully Java driver), and the `Thin driver` (fully Java driver). Among these, the Thin driver excels in performance, as it directly translates JDBC calls into the specific protocol of the associated database and does not require any supplementary software installation on either the client or server side.

1 Environment Setup

All the files mentioned below have been provided in Google Classroom.

1. Install Java Development Kit (JDK) using `jdk-19_windows-x64_bin.msi`.
2. If you want to use VSCode install the extension pack for Java.
3. Create a new Java Project and add `ojdbc14.jar` or `ojdbc6.jar` file as an external JAR file.
   a. For VSCode:
      from the Explorer Bar, go to `Java Projects` → `<projectname>` → `Referenced Libraries`. Then click on the `plus` sign and select the `ojdbc14.jar` or `ojdbc6.jar` and click the `Select Jar Libraries` button.
   b. For Eclipse IDE:
      from the Menu Bar, go to `Project` → `Properties` →. Then on the opened window, click on `Java Build Path` → `Classpath` → `Add External JARs`. This will open a File Explorer where you can navigate to the path where `ojdbc14.jar` or `ojdbc6.jar` is located to select and add it to your project.
   c. For IntelliJ IDE:
      from the Top Bar, go to `File` → `Project Structure` →. Then on the opened window, click on `Modules`. In the `Export` section, click on the `plus` sign → `JARs or Directories`. There provide the path to `ojdbc14.jar` or `ojdbc6.jar` and hit the `OK` button.

   If you are using any other IDE, Google "how to add external JAR files to <XYZ> IDE" to get help. Replace <XYZ> with the IDE that you are using.

2 Java Database Connectivity with Oracle

There are 5 steps to connect any Java application with the database using JDBC. These steps are as follows:

1. Register the Driver class
2. Create a connection
3. Create statement
4. Execute query
5. Close connection

Consider the Code Snippet 3.1 for connecting with JDBC step by step.

```
1  import java.sql.*;
2
3  class jdbc_practice {
4      public static void main(String args[]) {
5          try {
6              // step1 load the driver class
7              Class.forName("oracle.jdbc.driver.OracleDriver");
8
9              // step2 create the connection object
10             Connection con = DriverManager.getConnection(
11                     "jdbc:oracle:thin:@localhost:1521:xe", "DBMS", "dbms");
12
13             // step3 create the statement object
14             Statement stmt = con.createStatement();
15
16             // step4 execute query
17             // drop table
18             System.out.println("Drop table...");
19             String sql = "DROP TABLE REGISTRATION";
20             stmt.executeUpdate(sql);
21
22             // create table
23             String sql1 = "CREATE TABLE REGISTRATION " +
24                     "(id INTEGER not NULL, " +
25                     " first VARCHAR(255), " +
26                     " last VARCHAR(255), " +
27                     " age INTEGER, " +
28                     " PRIMARY KEY ( id ))";
29
30             stmt.executeUpdate(sql1);
31
32             // insert table
33             System.out.println("Inserting records into the table...");
34             String sql2 = "INSERT INTO Registration VALUES (100, 'Zara', 'Ali',
   18)";
35             stmt.executeUpdate(sql2);
36             sql2 = "INSERT INTO Registration VALUES (101, 'Mahnaz', 'Fatma', 25)";
37             stmt.executeUpdate(sql2);
38
39             // select table
40             System.out.println("Selecting records from the table...");
41             String QUERY = "SELECT id, first, last, age FROM Registration";
42             ResultSet rs = stmt.executeQuery(QUERY);
43             while (rs.next()) {
44                 // Display values
```

```
45                 System.out.print("ID: " + rs.getInt("id"));
46                 System.out.print(", Age: " + rs.getInt("age"));
47                 System.out.print(", First: " + rs.getString("first"));
48                 System.out.println(", Last: " + rs.getString("last"));
49             }
50             rs.close();
51 
52             // update table
53             System.out.println("Updating records from the table...");
54             String sql3 = "UPDATE Registration " +
55                     "SET age = 30 WHERE id in (100, 101)";
56             stmt.executeUpdate(sql3);
57 
58             // delete table
59             System.out.println("deleting records from the table...");
60             String sql4 = "DELETE FROM Registration " +
61                     "WHERE id = 101";
62             stmt.executeUpdate(sql4);
63 
64             // step5 close the connection object
65             con.close();
66 
67         } catch (Exception e) {
68             System.out.println(e);
69         }
70 
71     }
72 }
```

Code Snippet 3.1. Sample JDBC Connection code

Consider the following:

- Driver class: The driver class for the Oracle Database is "`oracle.jdbc.driver.OracleDriver`".
- Connection URL: The connection URL for the Oracle10G database is "`jdbc:oracle:thin:@localhost:1521:xe`" where `jdbc` is the API, `oracle` is the database, `thin` is the driver, `localhost` is the server name on which oracle is running (we may also use an IP address here), `1521` is the port number, and `XE` is the Oracle Service Name. You may get all these information from the `tnsnames.ora` file.
- Username: The default username for the Oracle Database is system.
- Password: It is the password given by the user at the time of installing the Oracle Database.

3 Some interfaces of JDBC API

The `java.sql` package contains classes and interfaces for JDBC API. Some popular interfaces and classes in the JDBC API include:

- `DriverManager: java.sql.DriverManager` is a class that manages a list of database drivers. It is used to establish a connection to the database by selecting an appropriate driver from the list.
- `Connection: java.sql.Connection` represents a connection to a relational database. It provides methods

for creating statements, committing or rolling back transactions, and managing other aspects of the connection.

- `Statement: java.sql.Statement` is an interface that represents an SQL statement. There are different types of statements, such as `Statement`, `PreparedStatement`, and `CallableStatement`, each serving a specific purpose. These are used to execute SQL queries and updates.
- `ResultSet: java.sql.ResultSet` represents the result set of a database query. It provides methods for iterating over the rows of the result set and retrieving data from each column.
- `PreparedStatement: java.sql.PreparedStatement` is a sub-interface of Statement. It is used to execute pre-compiled SQL queries with parameters. `PreparedStatement` is more efficient than `Statement` for executing queries repeatedly with different parameter values.
- `ResultSetMetaData: java.sql.ResultSetMetaData` is an interface that provides information about the columns of a `ResultSet`, such as the column names, types, and properties.
- `DatabaseMetaData: java.sql.DatabaseMetaData` provides methods to obtain metadata about the database, such as information about its tables, columns, and supported SQL features.

4 Task - Group B

Consider the schema shown in Figure 3.1 for the database of a university:

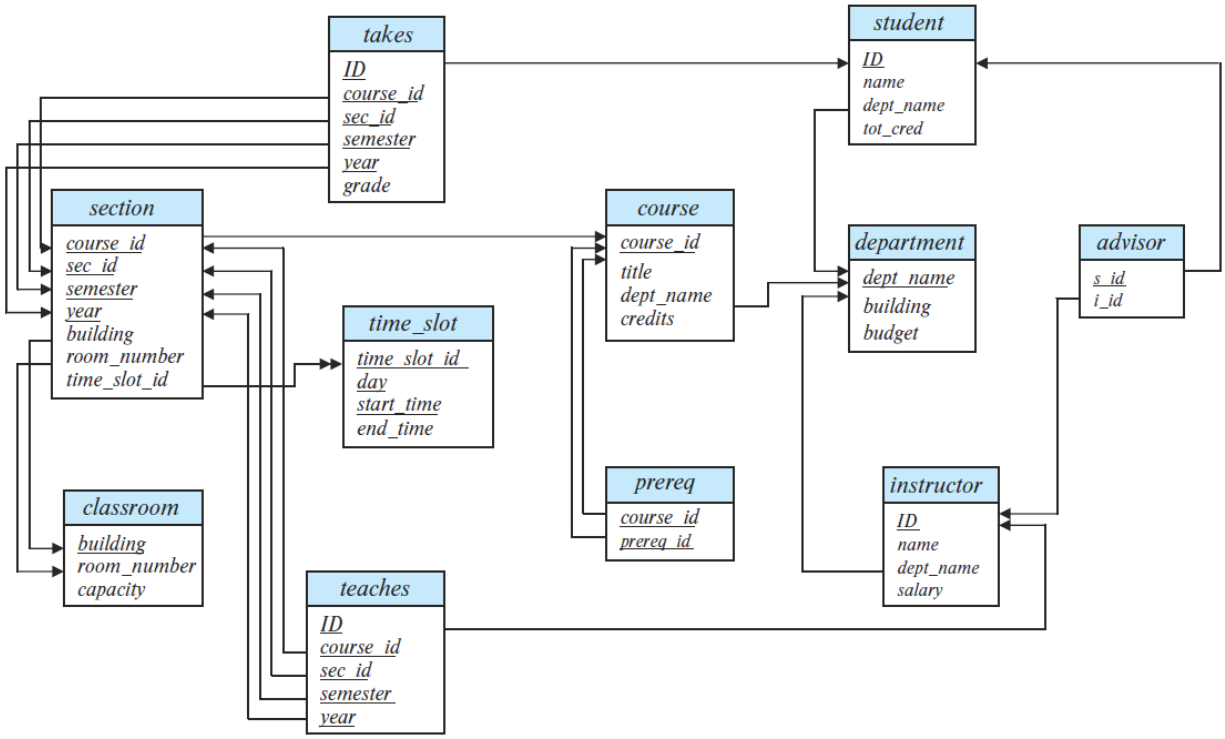

Figure 3.1. Schema diagram for a university database

Write the command `@"<file_path>\<file_name>.sql"` in your SQL command line to execute the provided .sql files. Now, write Java functions to perform each of the following tasks:

1. Decrease the budget of the departments having a budget of more than 99999 by 10%. Then show the number of departments that did not get affected.
2. Take the day of the week, starting hour, and ending hour as input from the user. Then print the names of the instructors who will be taking classes during that time.
3. Find the top N students based on the number of courses they are enrolled in. You should take N as input and print the ID, name, department name, and the number of courses taken by the student. If N is larger than the total number of students, print the information for all the students.
4. Insert a new student named 'Jane Doe' in the STUDENT table. The student should be enrolled in the department having the lowest number of students. The ID of the student will be $(X + 1)$, where X is the highest ID value among the existing students.
5. Find out the list of students who do not have any advisor assigned to them. Then assign them an advisor from their department. In case there are multiple instructors from a certain department, the advisor should be selected based on the least number of students advised. Finally, print the name of the students, the name of their advisor, and the number of students advised by the said advisor.

5 Task - Group A

Consider the schema shown in Figure 3.2 for the database of a university:

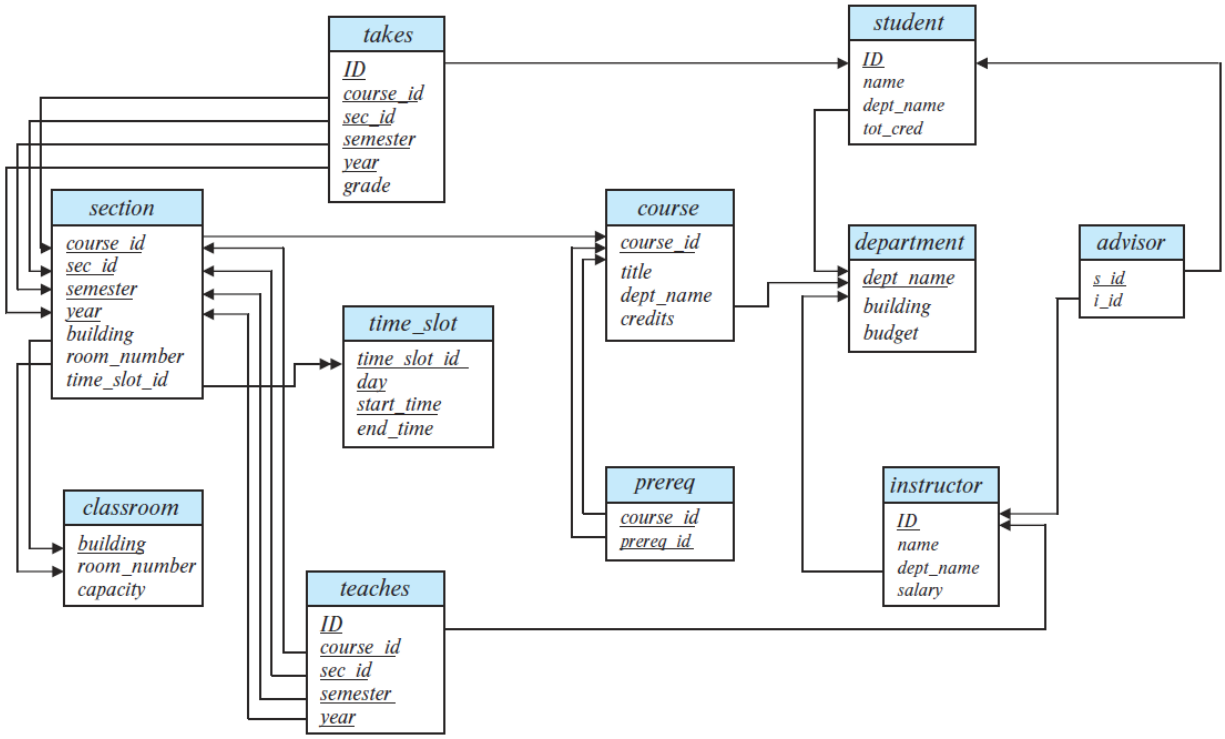

Figure 3.2. Schema diagram for a university database

Write the command `@"<file_path>\<file_name>.sql"` in your SQL command line to execute the provided .sql files. Now, write Java functions to perform each of the following tasks:

1. Set the salary of each instructor to $9000X$ where X is the total credits of the courses taught by the instructor. Print the names of the instructors whose salaries remain unchanged.
2. Considering the pre-requisite(s) for each course, print the course title and the names of the students who can enroll in them.
3. Take the name of the student as input from the user. Then print the weekly routine for the student. Each class should be printed in the following format:

   ```
   <Day>
   <Start Time> - <End Time>
   <Course ID> - <Title>
   <Building> - <Room>
   ```

   The days should be sorted based on the regular order of weekdays starting from Monday. If there are multiple classes scheduled on the same day, they should be sorted based on the starting time.
4. Find out the list of instructors who do not have any students assigned to them. Then assign them students from the same department who do not have any advisor. If there are multiple students from the same department that meet the criteria, then select the one with the lowest `tot_cred`. After that, print the names of the instructors who still do not have any students assigned to them.
5. Insert a new instructor named 'John Doe' in the `INSTRUCTOR` table. The instructor should be enrolled in the department having the highest number of students. The ID of the instructor will be $(X - 1)$, where X is the lowest ID value among the existing instructors. The salary of the instructor should be the average among all the instructors of the same department. Finally, print the information of the instructor.

Lab 4 Functions and Procedures

Functions and Procedures are two of the key components of PL/SQL stored programming units. They are used to build database-tier libraries to encapsulate application functionality, which is then co-located on the database tier for efficiency. Oracle maintains a unique list of stored object names for tables, views, sequences, stored programs, and types. This list is known as a namespace. Functions and procedures are in this namespace.
Stored functions and procedures provide a way to hide implementation details in a program unit. They also let you wrap the implementation from prying eyes on the server tier. However, the main motivation for Functions and Procedures is modular code. Modularization is the process by which you break up large blocks of code into smaller pieces (modules) that can be called by other modules. Modularization of code is analogous to normalization of data, with many of the same benefits and a few additional advantages. With modularization, your code becomes:

- **Reusable:** By breaking up a large program or entire application into individual components that plug-and-play together, you will usually find that many modules are used by more than one other program in your current application. Designed properly, these utility programs could even be of use in other applications!
- **Manageable:** Which would you rather debug: a 1,000-line program or five individual 200-line programs that call each other as needed? Our minds work better when we can focus on smaller tasks. You can also test and debug on a per program scale (called unit testing) before individual modules are combined for a more complicated integration test.
- **Readable:** Modules have names, and names describe behavior. The more you move or hide your code behind a programmatic interface, the easier it is to read and understand what that program is doing. Modularization helps you focus on the big picture rather than on the individual executable statements. You might even end up with that most elusive kind of software: self-documenting code.
- **Reliable:** The code you produce will have fewer errors. The errors you do find will be easier to fix because they will be isolated within a module. In addition, your code will be easier to maintain because there is less of it and it is more readable.

1 Procedure

A procedure is a module that performs one or more actions. Because a procedure call is a standalone executable statement in PL/SQL, a PL/SQL block could consist of nothing more than a single call to a procedure. Procedures are key building blocks of modular code, allowing you to both consolidate and reuse your program logic. The following illustrates a procedure prototype:

```
[CREATE [OR REPLACE]]
PROCEDURE procedure_name[(parameter[, parameter]...)]
[AUTHID {DEFINER | CURRENT_USER}] {IS | AS}
[PRAGMA AUTONOMOUS_TRANSACTION;]
    [local_declarations]
BEGIN
    executable_statements
[EXCEPTION
    exception_handlers]
END [procedure_name];
```

Procedures cannot be right operands. Nor can you use them in SQL statements. You move data into and out of PL/SQL stored procedures through their formal parameter list. Here, parameter modes define the behavior of formal parameters. These parameter modes offer you the ability to use pass-by-value or pass-by-reference formal parameters. The three parameter modes: `IN` (default), `OUT`, and `IN OUT` can be used with any subprogram:

- **IN:** The value of the actual parameter is passed into the procedure when the procedure is invoked. Inside the procedure, the formal parameter acts like a PL/SQL constant. It is considered read-only, i.e., it cannot be changed. You can assign a default value to a variable with IN parameter mode, making it optional.
- **OUT:** Any value the actual parameter has when the procedure is called is ignored. Inside the procedure, the formal parameter acts like an uninitialized PL/SQL variable and thus has a value of NULL. It can be read from and written to. Note that, you cannot assign a default value to a variable with OUT parameter, as it would make the variable optional.
- **IN OUT:** This mode is a combination of IN and OUT.

During the declaration of the parameters in Procedures, you must leave out the constraining part of the declaration. The following procedure can be used to determine the salary of an instructor given their ID:

```
/* Create procedure */
CREATE OR REPLACE
PROCEDURE FIND_SAL(I_ID IN NUMBER, SALARY OUT NUMBER)
AS
BEGIN
    SELECT MAX(SALARY) INTO SALARY
        FROM INSTRUCTOR
            WHERE ID = I_ID;
END;
/

/* Call it from an anonymous block */
DECLARE
    AMOUNT NUMBER;
BEGIN
    FIND_SAL(10101, AMOUNT);
    DBMS_OUTPUT.PUT_LINE(AMOUNT);
END;
/
```

Output:

```
65000
```

2 Function

A function is a module that returns data through its RETURN clause, rather than in an OUT or IN OUT parameter. Unlike a procedure call, which is a standalone executable statement, a call to a function can exist only as part of an executable statement, such as an element in an expression or the value assigned as the default in a declaration of a variable. Functions are convenient structures because you can call them directly from SQL statements or PL/SQL programs. They can also be used as right operands because they return a value. Since Functions return explicit values, it is recommended to not use OUT and IN OUT modes with functions. The following illustrates a function prototype:

```
[CREATE [OR REPLACE]]
FUNCTION function_name[(parameter[, parameter]...)]
RETURN return_type
[AUTHID {DEFINER | CURRENT_USER}] {IS | AS}
[PRAGMA AUTONOMOUS_TRANSACTION;]
    [local_declarations]
```

```
BEGIN
    executable_statements
[EXCEPTION
    exception_handlers]
RETURN statement;
END [function_name];
```

Both procedures and functions have a name, can take parameters, return values, and be called by many users. They are stored in the data dictionary. The difference is that a function must return a value, but in a procedure it is optional. Also, you cannot call procedures from an SQL statement.

The following program can be used to calculate the compound interest of a loan:

```
/* Declare function */
CREATE OR REPLACE
FUNCTION COMPOUND_INTEREST(PA NUMBER, AIR NUMBER := 5, TF)
RETURN NUMBER
IS
    CI NUMBER;
BEGIN
    CI := PA * ((1 + (AIR / 100)) ** TF) - PA;
RETURN CI;
END;
/

/* Call it from an anonymous block */
BEGIN
  DBMS_OUTPUT.PUT_LINE(COMPOUND_INTEREST(10000, 5, 5));
END;
/
```

Output:

```
2762.815625
```

Here, `PA` denotes the principle amount, `AIR` denotes the annual interest rate (which has a default value of `5`), `TF` denotes the compound period (years), and `CI` denotes the compound interest.

3 Parameter Notations

When calling a subroutine, such as a procedure or a function, positional, named, and mixed notations can be used.

3.1 Positional Notation

Positional notation means that you provide a value for each variable in the formal parameter list. The values must be in sequential order and must also match the datatype. You use positional notation to call the functions as follows:

```
BEGIN
    DBMS_OUTPUT.PUT_LINE(COMPOUND_INTEREST(10000, 5, 5));
END;
/
```

3.2 Named Notation

Named notation means that you pass actual parameters by using their formal parameter name, the association operator (=>), and the value. You call a function using named notation by:

```
BEGIN
    DBMS_OUTPUT.PUT_LINE(COMPOUND_INTEREST(TF => 5, PA => 10000, AIR => 5));
END;
/
```

Named notation lets you only pass values to required parameters, which means you accept the default values for any optional parameters.

```
BEGIN
    DBMS_OUTPUT.PUT_LINE(COMPOUND_INTEREST(TF => 5, PA => 10000));
END;
/
```

3.3 Mixed Noration

Mixed notation means that you can call subroutines by a combination of positional and named notation. This becomes very handy when parameter lists are defined with all mandatory parameters first and optional parameters next. It lets you name or avoid naming the mandatory parameters, and it lets you skip optional parameters where their default values work. It does not solve exclusionary notation problems. Exclusionary problems occur with positional notation when optional parameters are interspersed with mandatory parameters, and when you call some but not all optional parameters.

```
BEGIN
    DBMS_OUTPUT.PUT_LINE(COMPOUND_INTEREST(10000, TF => 5, AIR => 5));
END;
/
```

There is a restriction on mixed notation. All positional notation of actual parameters must occur first and in the same order as they are defined by the function signature.

3.4 Exclusionary Notation

If the formal parameters are defined as optional, you can exclude one or more of the actual parameters. For example, consider the following function:

```
CREATE OR REPLACE
FUNCTION ADD_THREE_NUMBERS
(A NUMBER := 0, B NUMBER := 0, C NUMBER := 0)
RETURN NUMBER
IS
    SUM NUMBER;
BEGIN
    SUM := A + B + C;
RETURN SUM;
END;
/
```

Here, all 3 parameters are optional. So we can write programs like:

```
BEGIN
    DBMS_OUTPUT.PUT_LINE(ADD_THREE_NUMBERS(3, C => 4));
END;
/
```

```
7
```

When you opt to not provide an actual parameter, it acts as if you are passing a null value. This is known as exclusionary notation. Oracle recommends that you should list optional parameters last in function and procedure signatures. They also recommend that you sequence optional variables so that you never have to skip an optional parameter in the list. These recommendations exist to circumvent errors when making positional notation calls.
You cannot really skip an optional parameter in a positional notation call. This is true because all positional calls are in sequence by datatype, but you can provide a comma-delimited null value when you want to skip an optional parameter in the list. However, Oracle 11g now lets you use mixed notation calls. You can now use positional notation for your list of mandatory parameters, and named notation for optional parameters. This lets you skip optional parameters without naming all parameters explicitly.

4 Error Handling

When creating a procedure/function, you might face compilation errors. For example, if you execute the following code:

```
CREATE OR REPLACE
PROCEDURE FIND_SAL(I_ID IN NUMBER, SALARY OUT NUMBER)
AS
BEGIN
    SELECT MAX(SALARY) INTO SALARY
        FROM INSTRUCTOR
            WHERE ID = I_ID /* ERROR: Missing semicolon */
END;
/
```

Output:

```
Warning: Procedure created with compilation errors.
```

Very helpful(!) You can use SHOW ERROR command (or SHO ERR for short) right after the warning message to find out the errors:

```
SHOW ERROR
```

Output:

```
Errors for PROCEDURE FIND_SAL:

LINE/COL ERROR
-------- -----------------------------------------------------------------
4/5      PL/SQL: SQL Statement ignored
6/29     PL/SQL: ORA-00933: SQL command not properly ended
7/4      PLS-00103: Encountered the symbol "end-of-file" when expecting
         one of the following:
         ( begin case declare end exception exit for goto if loop mod
         null pragma raise return select update while with
         <an identifier> <a double-quoted delimited-identifier>
         <a bind variable> << continue close current delete fetch lock
         insert open rollback savepoint set sql execute commit forall
         merge pipe purge
```

This should help you identify that you missed a semicolon (;) in line 7.

5 Task - Group B

Write PL/SQL statements to perform the following tasks:

1. **Warm-up**
   a. Print your first name.
   b. Take your student ID as input and print its length.
2. Consider the schema shown in Figure 4.1 for the database of a university:

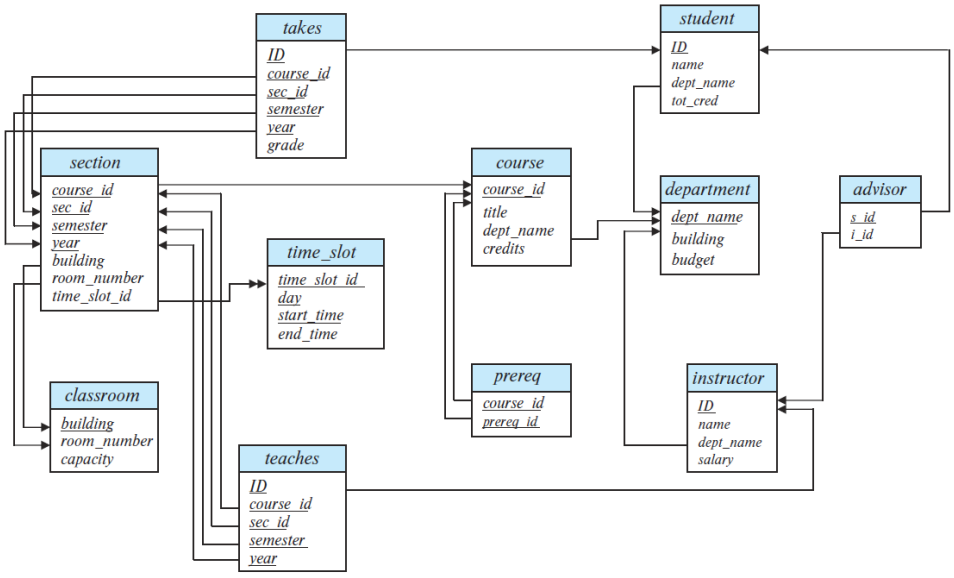

Figure 4.1. Schema diagram for a university database

Write the command @"<file_path>\<file_name>.sql" in your SQL command line to execute the provided .sql files. Now, write PL/SQL statements to perform each of the following tasks:

a. Set the salary of each instructor to $9000X$ where X is the total credits of the courses taught by the instructor. Print the names of the instructors whose salaries remain unchanged.

b. Considering the pre-requisite(s) for each course, print the course title and the names of the students who can enroll in them.

c. Take the name of the student as input from the user. Then print the weekly routine for the student. Each class should be printed in the following format:

```
<Day>
<Start Time> - <End Time>
<Course ID> - <Title>
<Building> - <Room>
```

The days should be sorted based on the regular order of weekdays starting from Monday. If there are multiple classes scheduled on the same day, they should be sorted based on the starting time.

d. Find out the list of instructors who do not have any students assigned to them. Then assign them students from the same department who do not have any advisor. If there are multiple students from the same department that meet the criteria, then select the one with the lowest `tot_cred`. After that, print the names of the instructors who still do not have any students assigned to them.

e. Insert a new instructor named 'John Doe' in the `INSTRUCTOR` table. The instructor should be enrolled in the department having the highest number of students. The ID of the instructor will be $(X - 1)$, where X is the lowest ID value among the existing instructors. The salary of the instructor should be the average among all the instructors of the same department. Finally, print the information of the instructor.

6 Task - Group A

Write PL/SQL statements to perform the following tasks:

1. **Warm-up**
   a. Print your student ID.
   b. Take your first name as input and print its length.
2. Consider the schema shown in Figure 4.2 for the database of a university:

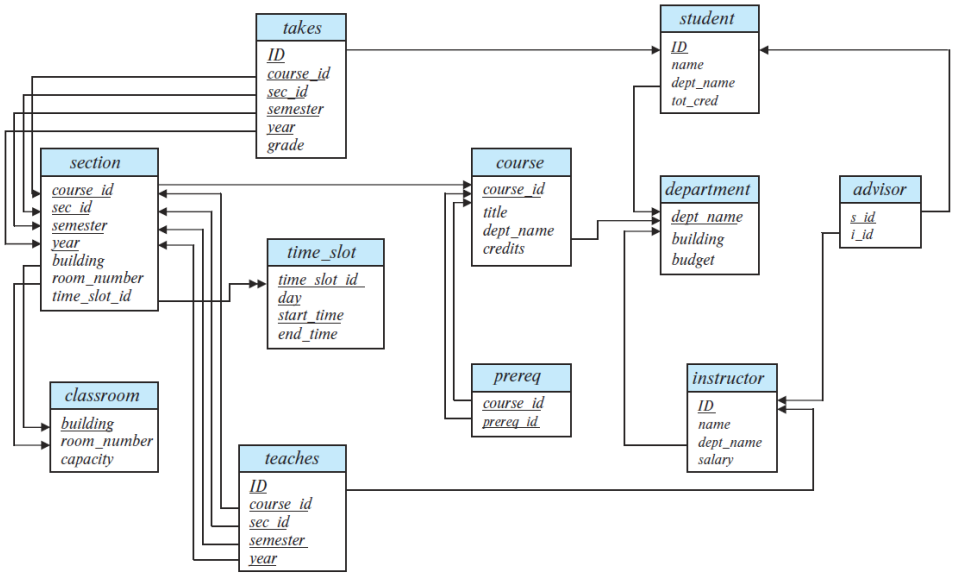

Figure 4.2. Schema diagram for a university database

Write the command @"<file_path>\<file_name>.sql" in your SQL command line to execute the provided .sql files. Now, write PL/SQL statements to perform each of the following tasks:

a. Decrease the budget of the departments having a budget of more than 99999 by 10%. Then show the number of departments that did not get affected.
b. Take the day of the week, starting hour, and ending hour as input from the user. Then print the names of the instructors who will be taking classes during that time.
c. Find the top N students based on the number of courses they are enrolled in. You should take N as input and print the ID, name, department name, and the number of courses taken by the student. If N is larger than the total number of students, print the information for all the students.
d. Insert a new student named 'Jane Doe' in the STUDENT table. The student should be enrolled in the department having the lowest number of students. The ID of the student will be $(X + 1)$, where X is the highest ID value among the existing students.
e. Find out the list of students who do not have any advisor assigned to them. Then assign them an advisor from their department. In case there are multiple instructors from a certain department, the advisor should be selected based on the least number of students advised. Finally, print the name of the students, the name of their advisor, and the number of students advised by the said advisor.

Lab 5 Cursor

Cursor structures are the return results from SQL SELECT statements. You can process SELECT statements row-by-row using cursors.

In response to any DML statement, the database creates a memory area, known as the context area, for processing an SQL statement, which contains all information needed for processing the statement, for example, the number of rows processed, etc. A cursor is a pointer to this context area. PL/SQL controls the context area through a cursor. A cursor holds the rows (one or more) returned by an SQL statement. The set of rows the cursor holds is referred to as the active set.

You can name a cursor so that it can be referred to in a program to fetch and process the rows returned by the SQL statement, one at a time. There are two types of cursors:

- Implicit cursor
- Explicit cursor

1 Implicit Cursor

Implicit cursors are automatically created by Oracle whenever an SQL statement is executed when there is no explicit cursor for the statement. Programmers cannot control the implicit cursors and the information in it.

Whenever a DML statement is issued, an implicit cursor is associated with this statement. For INSERT operations, the cursor holds the data that needs to be inserted. For UPDATE and DELETE operations, the cursor identifies the rows that would be affected.

In PL/SQL, you can refer to the most recent implicit cursor as the SQL cursor, which always has the attributes like:

Table 5.1. Implicit Cursors

Attribute	Description
%FOUND	Returns true if an INSERT, UPDATE, or DELETE statement affected one or more rows, or a SELECT INTO statement returned one or more rows. Otherwise, it returns false.
%NOTFOUND	The logical opposite of %FOUND. It returns true if an INSERT, UPDATE, or DELETE statement affected no rows, or a SELECT INTO statement returned no rows.
%ISOPEN	Always returns false for implicit cursors, because Oracle closes the SQL cursor automatically after executing its associated SQL statement.
%ROWCOUNT	This attributes returns the number of rows changed by an INSERT, UPDATE, or DELETE statement or the number of rows returned by a SELECT INTO statement.

For example, the following program counts the number of rows affected by the UPDATE statement:

```
1 DECLARE
2     TOTAL_ROWS NUMBER(2);
3 BEGIN
4     UPDATE INSTRUCTOR
5         SET SALARY = SALARY + 5000
6             WHERE SALARY < 65000;
```

```
    IF SQL%NOTFOUND THEN
        DBMS_OUTPUT.PUT_LINE('No instructor satisfied the condition');
    ELSIF SQL%FOUND THEN
        TOTAL_ROWS := SQL%ROWCOUNT;
        DBMS_OUTPUT.PUT_LINE(TOTAL_ROWS || ' instructors updated');
    END IF;
END;
/
```

Output:

```
3 instructors updated
```

The reserved word SQL before the cursor attribute stands for any implicit cursor.

1.1 Single-Row Implicit Cursors

The SELECT INTO statement is present in all implicit cursors that query data. It works only when a single row is returned by a select statement. You can select a column or list of columns in the SELECT clause and assign the row columns to individual variables or collectively to a record datatype. For example,

```
DECLARE
    /* For one-to-one assignment */
    I_ID INSTRUCTOR.ID%TYPE;
    I_NAME INSTRUCTOR.NAME%TYPE;
    I_SALARY INSTRUCTOR.SALARY%TYPE;
    /* For group assignment */
    TYPE INSTRUCTOR_RECORD IS RECORD
    (
        I_ID INSTRUCTOR.ID%TYPE,
        I_NAME INSTRUCTOR.NAME%TYPE,
        I_SALARY INSTRUCTOR.SALARY%TYPE
    );
    INS INSTRUCTOR_RECORD;
BEGIN
    /* Individual selection */
    SELECT ID, NAME, SALARY
        INTO I_ID, I_NAME, I_SALARY
            FROM INSTRUCTOR
                WHERE ROWNUM < 2;
    DBMS_OUTPUT.PUT_LINE('Name: ' || I_NAME);
    /* Group selection */
    SELECT ID, NAME, SALARY
        INTO INS
            FROM INSTRUCTOR
                WHERE ROWNUM < 2;
    DBMS_OUTPUT.PUT_LINE('Name: ' || INS.I_NAME);
END;
/
```

Output:

```
Name: Srinivasan
Name: Srinivasan
```

Note that the declarations in lines 16 to 25 anchor all variables to the target table and limits the query to one row by using the Oracle SQL `ROWNUM` pseudocolumn.

Single-row implicit cursors are great quick fixes, but they have weaknesses. It is a weakness that many developers attempt to exploit by using it to raise exceptions when cursors return more than one row. They do this because single-row implicit cursors raise an "exact fetch returned too many rows" error when returning more than one row. Better solutions are available to detect errors before fetching the data. You should explore alternatives when developing your code and where possible explicitly handle errors. Explicit cursors are typically better solutions every time.

1.2 Multiple-Row Implicit Curosrs

There are two ways you can create multiple-row implicit cursors. The first is done by writing any DML statement in a PL/SQL block. DML statements are considered multiple-row implicit cursors, even though you can limit them to a single row. We already saw that in the `UPDATE` example. The second is to write an embedded query in a cursor FOR loop rather than defined in a declaration block.

For example, the following program can be used to see the names and the monthly salaries of different instructors:

```
1 BEGIN
2     FOR ROW IN (SELECT NAME, SALARY FROM INSTRUCTOR) LOOP
3         DBMS_OUTPUT.PUT_LINE('Name: ' || ROW.NAME);
4         DBMS_OUTPUT.PUT_LINE('Monthly Salary: ' || TRUNC(ROW.SALARY/12));
5     END LOOP;
6 END;
7 /
```

Output:

```
Name: Srinivasan
Monthly Salary: 5416
Name: Wu
Monthly Salary: 7500
Name: Mozart
Monthly Salary: 3333
Name: Einstein
Monthly Salary: 7916
Name: El Said
Monthly Salary: 5000
Name: Gold
Monthly Salary: 7250
Name: Katz
Monthly Salary: 6250
Name: Califieri
Monthly Salary: 5166
Name: Singh
Monthly Salary: 6666
Name: Crick
Monthly Salary: 6000
Name: Brandt
Monthly Salary: 7666
Name: Kim
Monthly Salary: 6666
```

Note that this implicit cursor is available in the scope of the cursor FOR loop index. Hence, the SQL%ROWCOUNT attribute returns a null value for this type of cursor.

2 Explicit Cursor

Explicit cursors are programmer-defined cursors for gaining more control over the context area. It should be defined in the declaration section of the PL/SQL block. It is created on a SELECT statement which returns one or more rows. The prototype for declaring an explicit cursor is as follows:

```
1 CURSOR cursor_name IS select_statement;
```

Four steps are required for using explicit cursors:

1. Declaring the cursor for initializing the memory
2. Opening the cursor for allocating memory
3. Fetching the cursor for retrieving data
4. Closing the cursor to release allocated memory.

The attributes of the explicit cursors are shown in Table 5.2. The attributes work the same way whether an explicit cursor is dynamic or static but differently than the limited set that work with implicit cursors. The explicit cursor attributes return different results, depending on where they are called in reference to the OPEN, FETCH, and CLOSE statements.

Table 5.2. Explicit Cursor Attributes

Statement	State	%FOUND	%NOTFOUND	%ISOPEN	%ROWCOUNT
OPEN	Before	Exception	Exception	FALSE	Exception
	After	NULL	NULL	TRUE	0
1st FETCH	Before	NULL	NULL	TRUE	0
	After	TRUE	FALSE	TRUE	1
Next FETCH	Before	TRUE	FALSE	TRUE	1
	After	TRUE	FALSE	TRUE	$n + 1$
Last FETCH	Before	TRUE	FALSE	TRUE	$n + 1$
	After	FALSE	TRUE	TRUE	$n + 1$
CLOSE	Before	FALSE	TRUE	TRUE	$n + 1$
	After	Exception	Exception	FALSE	Exception

The %FOUND cursor attribute signals that rows are available to retrieve from the cursor and the %NOTFOUND attribute signals that all rows have been retrieved from the cursor. The %ISOPEN attribute lets you know that the cursor is already open, and is something you should consider running before attempting to open a cursor. Like implicit cursors, the %ROWCOUNT attribute tells you how many rows you have fetched at any given point. Only the %ISOPEN cursor attribute works anytime without an error. The other three raise errors when the cursor is not open.

Static SELECT Cursor

Static SELECT statements return the same query each time with potentially different results. The result changes as the data changes in the table or views.

For example, the following program can be used to iterate through department budgets:

```
DECLARE
    D_NAME DEPARTMENT.DEPT_NAME%TYPE;
    D_BUDGET DEPARTMENT.BUDGET%TYPE;
    CURSOR C_DEPT
    IS
        SELECT DEPT_NAME, BUDGET
            FROM DEPARTMENT;
BEGIN
    OPEN C_DEPT;
    LOOP
        FETCH C_DEPT INTO D_NAME, D_BUDGET;
        EXIT WHEN C_DEPT%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(D_NAME || ' ' || D_BUDGET);
    END LOOP;
    CLOSE C_DEPT;
END;
/
```

Output:

```
Biology 90000
Comp. Sci. 100000
Elec. Eng. 85000
Finance 120000
History 50000
Music 80000
Physics 70000
```

Dynamic SELECT Cursor

Dynamic SELECT statements act like parameterized subroutines. They run different queries each time, depending on the actual parameters provided when they are opened.

This can be achieved in two ways. An explicit cursor query can reference any variable in its scope. When you open an explicit cursor, PL/SQL evaluates any variables in the query and uses those values when identifying the result set. Changing the values of the variables later does not change the result set.

For example, the following program can be used to find the list of courses for a particular credit:

```
1  DECLARE
2      C_TITLE COURSE.TITLE%TYPE;
3      C_CREDIT COURSE.CREDITS%TYPE := 4;
4      CURSOR C_COURSE
5      IS
6          SELECT TITLE
7              FROM COURSE
8                  WHERE CREDITS = C_CREDIT;
9  BEGIN
10     OPEN C_COURSE;
11     LOOP
12         FETCH C_COURSE INTO C_TITLE;
13         EXIT WHEN C_COURSE%NOTFOUND;
14         DBMS_OUTPUT.PUT_LINE(C_TITLE);
15     END LOOP;
16     CLOSE C_COURSE;
17 END;
18 /
```

Output:

```
Intro. to Biology
Genetics
Intro. to Computer Science
Game Design
Physical Principles
```

The value of C_CREDIT is evaluated and used during the opening of the cursor. Even if we change the value after that, it will not affect the rows that are returned.

Relying on local variables can be confusing and more difficult to support the code. Another way can be defining cursors that accept formal parameters. You can create an explicit cursor that has formal parameters, and then pass different parameters to the cursor each time you open it. In the cursor query, you can use a formal cursor parameter anywhere

that you can use a constant. Outside the cursor query, you cannot reference the formal cursor parameters. We can rewrite the previous program:

```
DECLARE
    C_TITLE COURSE.TITLE%TYPE;
    CURSOR C_COURSE
    (C_CREDIT COURSE.CREDITS%TYPE)
    IS
        SELECT TITLE
            FROM COURSE
                WHERE CREDITS = C_CREDIT;
BEGIN
    OPEN C_COURSE(4);
    LOOP
        FETCH C_COURSE INTO C_TITLE;
        EXIT WHEN C_COURSE%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(C_TITLE);
    END LOOP;
    CLOSE C_COURSE;
END;
/
```

Output:

```
Intro. to Biology
Genetics
Intro. to Computer Science
Game Design
Physical Principles
```

The benefit of this approach is that we can open the cursor multiple times using separate parameters to get different information. For example, the following program can identify the students in a certain department who are slacking off by taking a lesser number of courses:

```
/* Create procedure for printing student information. */
CREATE OR REPLACE
PROCEDURE PRINT_SLACKING_STUDENT
(S_DEPT IN STUDENT.DEPT_NAME%TYPE, S_AVG IN STUDENT.TOT_CRED%TYPE)
IS
    S_NAME STUDENT.NAME%TYPE;
    CURSOR C_STUDENT
    (DEPT STUDENT.DEPT_NAME%TYPE,
    AVERAGE STUDENT.TOT_CRED%TYPE)
    IS
        SELECT NAME
            FROM STUDENT
                WHERE DEPT_NAME = DEPT AND TOT_CRED < AVERAGE;
BEGIN
    OPEN C_STUDENT(S_DEPT, S_AVG);
    DBMS_OUTPUT.PUT_LINE('----------------');
```

```
    DBMS_OUTPUT.PUT_LINE('Slacking students from ' || S_DEPT);
    DBMS_OUTPUT.PUT_LINE('----------------');
    LOOP
        FETCH C_STUDENT INTO S_NAME;
        EXIT WHEN C_STUDENT%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(S_NAME);
    END LOOP;
    CLOSE C_STUDENT;
END;
/

/* Call procedure to find students */
DECLARE
    S_AVG STUDENT.DEPT_NAME%TYPE;
BEGIN
    SELECT AVG(TOT_CRED) INTO S_AVG
        FROM STUDENT
            WHERE DEPT_NAME = 'Comp. Sci.';
    PRINT_SLACKING_STUDENT('Comp. Sci.', S_AVG);

    SELECT AVG(TOT_CRED) INTO S_AVG
        FROM STUDENT
            WHERE DEPT_NAME = 'Physics';
    PRINT_SLACKING_STUDENT('Physics', S_AVG);
END;
/
```

Output:

```
----------------
Slacking students from Comp. Sci.
----------------
Shankar
Williams
Brown
----------------
Slacking students from Physics
----------------
Snow
```

3 Cursor using FOR Loop

The cursor FOR loop is an elegant and natural extension of the numeric FOR loop in PL/SQL. The body of the for loop is executed for each row returned by the query. The benefit of using this loop is that Oracle handles the OPEN, FETCH, and CLOSE internally.

For example, the following program can be used to determine the yearly cost for each department in terms of the salary of the instructors:

```
/* Declare function */
```

```
CREATE OR REPLACE FUNCTION
TOTAL_SALARY(D_NAME INSTRUCTOR.DEPT_NAME%TYPE)
RETURN NUMBER
IS
    CURSOR C_INSTRUCTOR
    IS
        SELECT SALARY
            FROM INSTRUCTOR
                WHERE DEPT_NAME = D_NAME;
    TOTAL NUMBER := 0;
BEGIN
    FOR INS_ROW IN C_INSTRUCTOR
    LOOP
        TOTAL := TOTAL + INS_ROW.SALARY;
    END LOOP;
RETURN TOTAL;
END;
/

/* Call it from an anonymous block */
BEGIN
    DBMS_OUTPUT.PUT_LINE(TOTAL_SALARY('Comp. Sci.'));
END;
/
```

Output:

```
232000
```

4 Task - Group B

Consider the schema shown in Figure 5.1 for the database of a university:

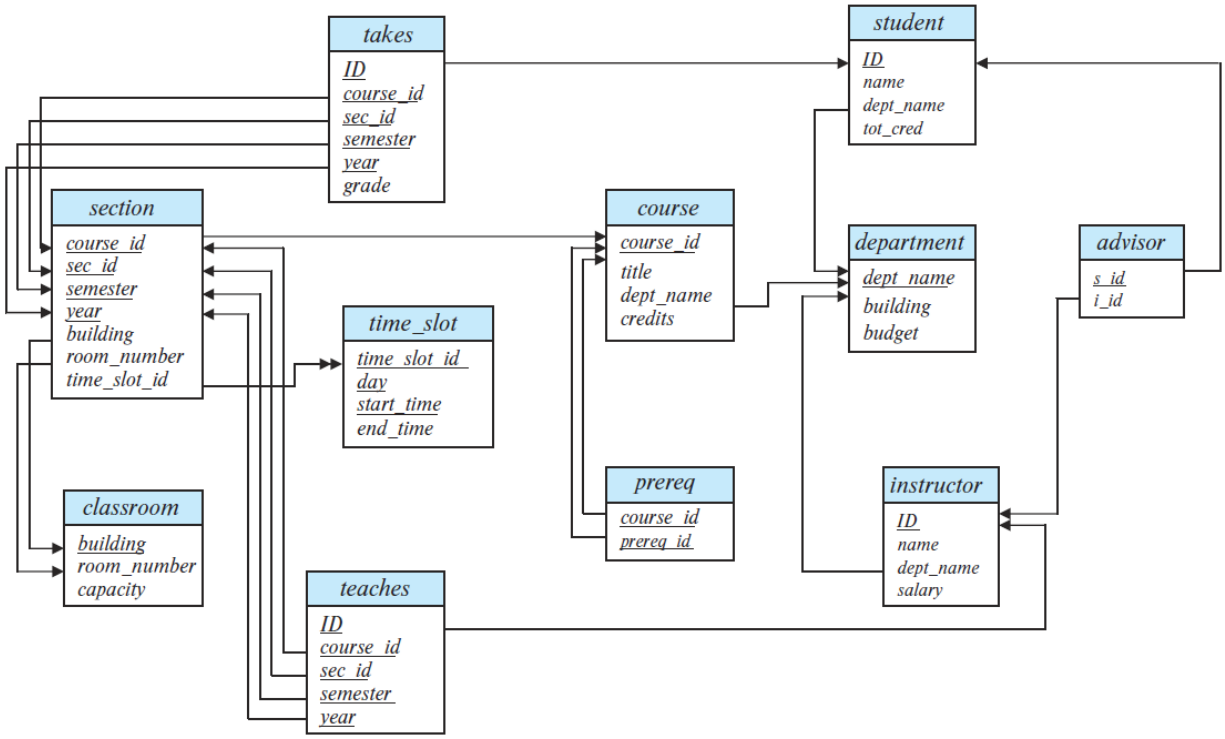

Figure 5.1. Schema diagram for a university database

Write the command `@"<file_path>\<file_name>.sql"` in your SQL command line to execute the provided .sql files. Now, write PL/SQL statements to perform each of the following tasks:

1. Decrease the budget of the departments having a budget of more than 99999 by 10%. Then show the number of departments that did not get affected.
2. Take the day of the week, starting hour, and ending hour as input from the user. Then print the names of the instructors who will be taking classes during that time.
3. Find the top N students based on the number of courses they are enrolled in. You should take N as input and print the ID, name, department name, and the number of courses taken by the student. If N is larger than the total number of students, print the information for all the students.
4. Insert a new student named 'Jane Doe' in the STUDENT table. The student should be enrolled in the department having the lowest number of students. The ID of the student will be $(X + 1)$, where X is the highest ID value among the existing students.
5. Find out the list of students who do not have any advisor assigned to them. Then assign them an advisor from their department. In case there are multiple instructors from a certain department, the advisor should be selected based on the least number of students advised. Finally, print the name of the students, the name of their advisor, and the number of students advised by the said advisor.

5 Task - Group A

Consider the schema shown in Figure 5.2 for the database of a university:

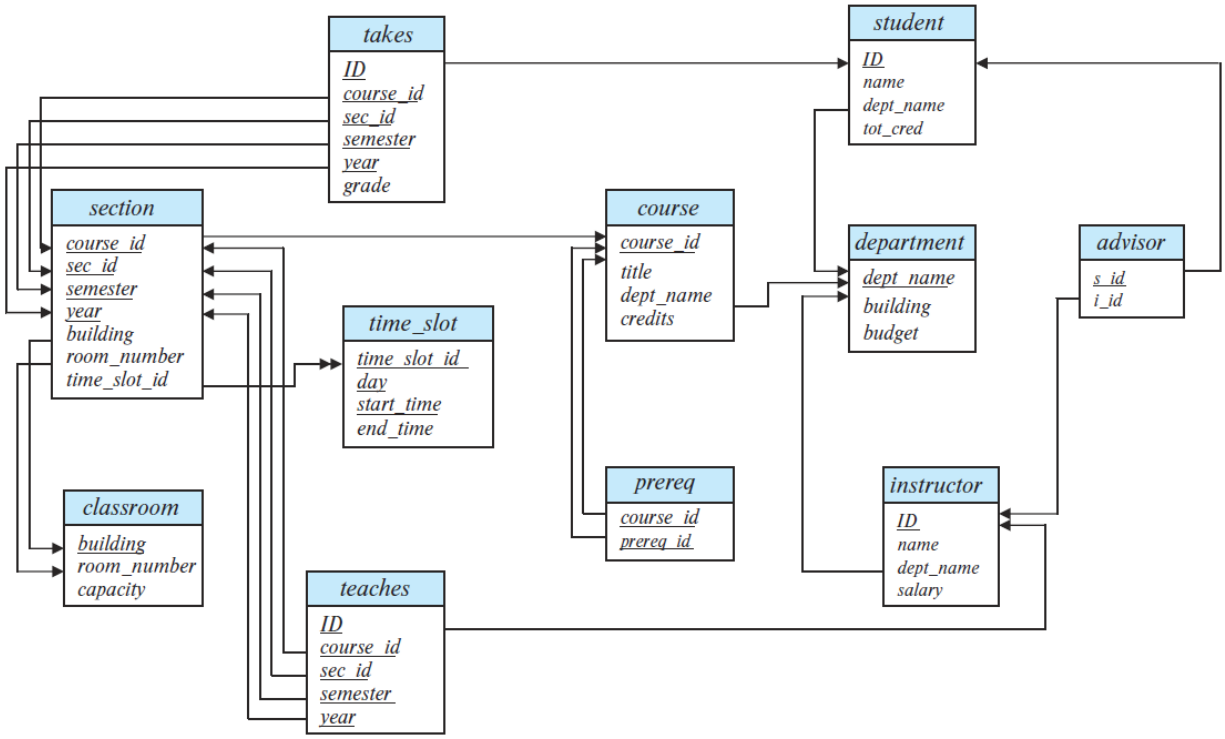

Figure 5.2. Schema diagram for a university database

Write the command `@"<file_path>\<file_name>.sql"` in your SQL command line to execute the provided .sql files. Now, write PL/SQL statements to perform each of the following tasks:

1. Set the salary of each instructor to $9000X$ where X is the total credits of the courses taught by the instructor. Print the names of the instructors whose salaries remain unchanged.
2. Considering the pre-requisite(s) for each course, print the course title and the names of the students who can enroll in them.
3. Take the name of the student as input from the user. Then print the weekly routine for the student. Each class should be printed in the following format:

   ```
   <Day>
   <Start Time> - <End Time>
   <Course ID> - <Title>
   <Building> - <Room>
   ```

   The days should be sorted based on the regular order of weekdays starting from Monday. If there are multiple classes scheduled on the same day, they should be sorted based on the starting time.
4. Find out the list of instructors who do not have any students assigned to them. Then assign them students from the same department who do not have any advisor. If there are multiple students from the same department that meet the criteria, then select the one with the lowest `tot_cred`. After that, print the names of the instructors who still do not have any students assigned to them.
5. Insert a new instructor named 'John Doe' in the `INSTRUCTOR` table. The instructor should be enrolled in the department having the highest number of students. The ID of the instructor will be $(X - 1)$, where X is the lowest ID value among the existing instructors. The salary of the instructor should be the average among all the instructors of the same department. Finally, print the information of the instructor.

Lab 6 Triggers

Database triggers are specialized stored programs. As such, they are defined by very similar DDL rules. Likewise, triggers can call SQL statements and PL/SQL functions and procedures. They run between the time you issue a command and the time you perform the database management system action. You can write triggers in PL/SQL or Java. Triggers can capture events that create, modify, or drop objects, and they can capture inserts to, updates of, and deletes from tables or views. They can also monitor changes in the state of the database or schema, and the actions of users.

Triggers differ from stored functions and procedures because you cannot call them directly. Database triggers are fired when a triggering event occurs in the database. This makes them very powerful tools in your efforts to manage the database. You are able to limit or redirect actions through triggers. Using triggers, you can:

- Control the behavior of DDL statements, as by altering, creating, or renaming objects
- Control the behavior of DML statements, like inserts, updates, and deletes
- Enforce referntial integrity, complex business rules, and security policies
- Control and redirect DML statements when they change data in a view
- Audit information of system access and behavior by creating transparent logs

On the other hand, you cannot control the sequence of or synchronize calls to triggers, and this can present problems if you rely too heavily on triggers. The only control you have is to designate them as before or after certain events. Oracle 11g delivers compound triggers to help you manage larger events, like those triggering events that you would sequence. There are risks with triggers. The risks are complex because while SQL statements fire triggers, triggers call SQL statements. A trigger can call a SQL statement that in turn fires another trigger. The subsequent trigger could repeat the behavior and fire another trigger. This creates cascading triggers. Oracle 11g and earlier releases limit the number of cascading trigger to 32, after which an exception is thrown. The trigger body can be no longer than 32,760 bytes. That is because trigger bodies are stored in `LONG` datatype columns. This means you should consider keeping your trigger bodies small. You do that by placing the coding logic in other schema-level components, like functions, procedures, and packages.

There are five types of triggers: DDL triggers, DML triggers, Compound triggers, Instead-of triggers, and System or database event triggers. In this course, we will be focusing on DDL and DML triggers.

1 DDL Triggers

DDL triggers fire when you create, change, or remove objects in a database schema. They are useful to control or monitor DDL statements. An *instead-of create* table trigger provides you with a tool to ensure table creation meets your development standards, like including storage or partitioning clauses. You can also use them to monitor poor programming practices, such as when programs *create* and *drop* temporary tables rather than using Oracle collections. Temporary tables can fragment disk space and degrade database performance over time.

A list of data definition events that work with DDL triggers are shown below:

Table 6.1. Available Data Definition Events

DDL Event	Description
`ALTER`	Changes object constraints, names, storage clauses, or structures.
`ANALYZE`	Computes statistics for the cost optimizer.
`ASSOCIATE STATISTICS`	Links a statistics type to a column, function, package, type, domain, index, or index type.

DDL Event	Description
AUDIT	Enables auditing on an object or system.
COMMENT	Document column or table purposes.
CREATE	Creates objects in the database, like objects, privileges, roles, tables, users, and views.
DDL	Represents any of the primary data definition events. It effectively says any DDL event acting on anything.
DISASSOCIATE STATISTICS	Unlinks a statistic type from a column, function, package, type, domain index, or index type.
DROP	Drops objects in the database, like objects, privileges, roles, tables, users, and views.
GRANT	Grants privileges or roles to users in the database. The privileges enable a user to act on objects, like objects, privileges, roles, tables, users, and views.
NOAUDIT	Disables auditing on an object or system.
RENAME	Renames objects in the database, like columns, constraints, objects, privileges, roles, synonyms, tables, users, and views.
REVOKE	Revokes privileges or roles from users in the database. The privileges enable a user to act on objects, like objects, privileges, roles, tables, users, and views.
TRUNCATE	Truncates tables, which drops all rows from a table and resets the high-water mark to the original storage clause initial extent value. Unlike the DML DELETE statement, it cannot be reversed by a ROLLBACK command. You can use the new flashback to undo the change.

You often use DDL triggers to monitor significant events in the database. Sometimes you use them to monitor errant code. Errant code can perform activities that corrupt or destabilize your database. More often, you use these in development, test, and stage systems to understand and monitor the dynamics of database activities. DDL triggers are very useful when you patch your application code. They can let you find potential changes between releases. You can also use the instead-of create trigger during an upgrade to enforce table creation storage clauses or partitioning rules. However, the overhead of these types of triggers should be monitored carefully in production systems.
DDL triggers can also track the creation and modification of tables by application programs that lead to database fragmentation. They are also effective security tools when you monitor GRANT and REVOKE privilege statements.

1.1 Building DDL Triggers

The prototype for building DDL triggers is:

```
CREATE [OR REPLACE]
TRIGGER trigger_name
{BEFORE | AFTER | INSTEAD OF} ddl_event ON {DATABASE | SCHEMA}
[WHEN (logical_expression)]
[DECLARE]
  declaration_statements;
BEGIN
```

```
  execution_statements;
END [trigger_name];
```

You can use the INSTEAD OF clause only when auditing a creation event. Before triggers make sure the contents of the trigger body occur before the triggering DDL command, while after triggers run last.
Note that, table and trigger can share the same name. This is possible because there are two namespaces in Oracle databases: one for triggers and another for everything else.
Let's see an example of triggers. If we want to log CREATE, ALTER, and DELETE for a particular user (schema):

```
CREATE OR REPLACE
TRIGGER DDL_INFO_TRIGGER
BEFORE CREATE OR ALTER OR DROP ON SCHEMA
BEGIN
  DBMS_OUTPUT.PUT_LINE('Who did it? ' || ORA_DICT_OBJ_OWNER);
  DBMS_OUTPUT.PUT_LINE('What was the operation? ' || ORA_SYSEVENT);
  DBMS_OUTPUT.PUT_LINE('On what? ' || ORA_DICT_OBJ_NAME);
  DBMS_OUTPUT.PUT_LINE('On what type of object it was? ' || ORA_DICT_OBJ_TYPE);
END;
/
```

Here, we use some keywords called Event Attribute Functions.

Sidenote: Event Attribute Functions

Event Attribute Functions can be used to supplement DDL triggers. A few of the system-defined event attribute functions are:

- ORA_CLIENT_IP_ADDRESS
  Return: the client IP address as a VARCHAR2 datatype.

```
DECLARE
  IP_ADDRESS VARCHAR2(11);
BEGIN
  IF ORA_SYSEVENT = 'LOGON' THEN
    IP_ADDRESS := ORA_CLIENT_IP_ADDRESS;
  END IF;
END;

```

- ORA_DATABASE_NAME
  Return: the database name as a VARCHAR2 datatype.

```
DECLARE
  DATABASE VARCHAR2(50);
BEGIN
  DATABASE := ORA_DATABASE_NAME;
END;
```

- ORA_DES_ENCRYPTED_PASSWORD
  Return: the DES-encrypted password as a VARCHAR2 datatype.
  Equivalent to the value in the SYS.USER$ table PASSWORD column in Oracle 11g. Previously DBA_USERS or ALL_USERS view contained this information.

```
DECLARE
  PASSWORD VARCHAR2(60);
BEGIN
  IF ORA_DICT_OBJ_TYPE = 'USER' THEN
    PASSWORD := ORA_DES_ENCRYPTED_PASSWORD;
  END IF;
END;
```

▸ ORA_DICT_OBJ_NAME

Return: an object name of the target of the DDL statement as a VARCHAR2 datatype. It also updates the

```
DECLARE
  DATABASE VARCHAR2(50);
BEGIN
  DATABASE := ORA_DICT_OBJ_NAME;
END;
```

▸ ORA_DICT_OBJ_NAME_LIST

Parameter: a table of VARCHAR2(64) datatype containig list of object names touched by the triggering event.

Return: the number of elements in the list as a PLS_INTEGER datatype. It also updates the list since it is passed by reference.

```
DECLARE
  NAME_LIST DBMS_STANDARD.ORA_NAME_LIST_T;
  COUNTER PLS_INTEGER;
BEGIN
  IF ORA_SYSEVENT = 'ASSOCIATE_STATISTICS' THEN
    COUNTER := ORA_DICT_OBJ_NAME_LIST(NAME_LIST);
  END IF;
END;
```

▸ ORA_DICT_OBJ_OWNER

Return: an owner of the object acted upon by the event as a VARCHAR2 datatype.

```
DECLARE
  OWNER VARCHAR2(30);
BEGIN
  OWNER := ORA_DICT_OBJ_OWNER;
END;
```

▸ ORA_DICT_OBJ_OWNER_LIST

Parameter: a table of VARCHAR2(64) datatype containing the list of object owners where their statistics were analyzed by a triggering event.

Return: the number of elements in the list indexed by a PLS_INTEGER datatype. It also updates the list since it is passed by reference.

```
DECLARE
  OWNER_LIST DBMS_STANDARD.ORA_NAME_LIST_T;
  COUNTER PLS_INTEGER;
BEGIN
  IF ORA_SYSEVENT = 'ASSOCIATE_STATISTICS' THEN
```

```
    COUNTER := ORA_DICT_OBJ_OWNER_LIST(OWNER_LIST);
  END IF;
END;
```

▸ ORA_DICT_OBJ_TYPE

Return: the datatype of the dictionary object changed by the event as a VARCHAR2 datatype.

```
DECLARE
  OBJ_TYPE VARCHAR2(19);
BEGIN
  OBJ_TYPE := ORA_DICT_OBJ_TYPE;
END;
```

▸ ORA_GRANTEE

Parameter: a table of VARCHAR2(64) datatypes containing the list of users granted privileges or roles by the triggering event.

Return: the number of elements in the list indexed by a PLS_INTEGER. It also updates the list since it is passed by reference.

```
DECLARE
  USER_LIST DBMS_STANDARD.ORA_NAME_LIST_T;
  COUNTER PLS_INTEGER;
BEGIN
  IF ORA_SYSEVENT = 'GRANT' THEN
    COUNTER := ORA_GRANTEE(USER_LIST);
  END IF;
END;
```

▸ ORA_INSTANCE_NUM

Return: the current database instance number as a NUMBER datatype.

```
DECLARE
  INSTANCE NUMBER;
BEGIN
  INSTANCE := ORA_INSTANCE_NUM;
END;
```

▸ ORA_IS_ALTER_COLUMN

Parameter: A column name.

Return: True or False as a BOOLEAN datatype indicating whether the column has been altered or not.

```
DECLARE
  TYPE COLUMN_LIST IS TABLE OF VARCHAR2(32);
  COLUMNS COLUMN_LIST := COLUMN_LIST('CREATED_BY', 'LAST_UPDATED_BY');
BEGIN
  IF ORA_SYSEVENT = 'ALTER' AND ORA_DICT_OBJ_TYPE = 'TABLE' THEN
    FOR i IN 1 .. COLUMNS.COUNT LOOP
      IF ORA_IS_ALTER_COLUMN(COLUMNS(i)) THEN
        INSERT INTO LOGGING_TABLE
        VALUES(ORA_DICT_OBJ_NAME || '.' || COLUMNS(i) || ' changed.');
      END IF;
```

```
    END LOOP;
  END IF;
END;
```

▸ ORA_IS_CREATING_NESTED_TABLE

Return: True or False value as a BOOELAN datatype indicating whether a nested table is created.

```
BEGIN
  IF ORA_SYSEVENT = 'CREATE' AND
     ORA_DICT_OBJ_TYPE = 'TABLE' AND
     ORA_IS_CREATING_NESTED_TABLE THEN
      INSERT INTO LOGGING_TABLE
      VALUES(ORA_DICT_OBJ_NAME || '.' || ' created with nested table.');
  END IF;
END;
```

▸ ORA_IS_DROP_COLUMN

Parameter: Column name.

Return: True or False value as a BOOLEAN datatype indicating whether the column has been dropped or not.

```
DECLARE
  TYPE COLUMN_LIST IS TABLE OF VARCHAR2(32);
  COLUMNS COLUMN_LIST := COLUMN_LIST('CREATED_BY', 'LAST_UPDATED_BY');
BEGIN
  IF ORA_SYSEVENT = 'DROP' AND ORA_DICT_OBJ_TYPE = 'TABLE' THEN
    FOR i IN 1 .. COLUMNS.COUNT LOOP
      IF ORA_IS_DROP_COLUMN(COLUMNS(i)) THEN
        INSERT INTO LOGGING_TABLE
        VALUES(ORA_DICT_OBJ_NAME || '.' || COLUMNS(i) || ' changed.');
      END IF;
    END LOOP;
  END IF;
END;
```

▸ ORA_IS_SERVERERROR

Parameter: An error number.

Return: True or False value as a BOOLEAN datatype indicating whether the error is on the error stack or not.

```
BEGIN
  IF ORA_IS_SERVERERROR(4082) THEN
    INSERT INTO LOGGING_TABLE
    VALUES('ORA-0408 error thrown.');
  END IF;
END;
```

▸ ORA_LOGIN_USER

Return: The current schema name as a VARCHAR2 datatype.

```
BEGIN
  INSERT INTO LOGGING_TABLE
  VALUES(ORA_LOGIN_USER || ' is the current user.');
END;
```

▸ ORA_PARTITION_POS

Return: The numeric position with the SQL text where you can insert a partition clause.
Only available in an INSTEAD OF CREATE trigger.

```
DECLARE
  SQL_TEXT ORA_NAME_LIST_T;
  SQL_STMT VARCHAR2(32767);
  PARTITION VARCHAR2(32767) := 'partitioning_clause';
BEGIN
  FOR i IN 1 .. ORA_SQL_TXT(SQT_TEXT) LOOP
    SQL_STMT := SQL_STMT || SQL_TEXT(i);
  END LOOP;
  SQL_STMT := SUBSTR(SQT_TEXT, 1, ORA_PARTITIONING_POS - 1) || ' '
              || PARTITION || ' ' || SUBSTR(SQL_TEXT, ORA_PARTITION_POS);
  -- Add logic to prepend schema because this runs under SYSTEM.
  SQL_STMT := REPLACE(UPPER(SQL_STMT), 'CREATE TABLE ', 'CREATE TABLE ' ||
   ORA_LOGIN_USER || '.');
  EXECUTE IMMEDIATE SQL_STMT;
END;
```

▸ ORA_PRIVILEGE_LIST

Parameter: a table of VARCHAR2(64) datatypes containing the list of privileges or roles granted by the triggering event.
Return: the number of elements in the list indexed by a PLS_INTEGER datatype. It also updates the list since it is passed by reference.

```
DECLARE
  PRIV_LIST DBMS_STANDARD.ORA_NAME_LIST_T;
  COUNTER PLS_INTEGER;
BEGIN
  IF ORA_SYSEVENT = 'GRANT' OR
     ORA SYSEVENT = 'REMOVE' THEN
    COUNTER := ORA_PRIVILEGE_LIST(PRIV_LIST);
  END IF;
END;
```

▸ ORA_REVOKEE

Parameter: a table of VARCHAR2(64) datatypes containing the list of users that had privileges or roles revoked by the triggering event.
Return: the number of elements in the list indexed by a PLS_INTEGER datatype. It also updates the list since it is passed by reference.

```
DECLARE
  REVOKEE_LIST DBMS_STANDARD.ORA_NAME_LIST_T;
  COUNTER PLS_INTEGER;
BEGIN
  IF ORA_SYSEVENT = 'REVOKE' THEN
    COUNTER := ORA_PRIVILEGE_LIST(REVOKEE_LIST);
  END IF;
END;
```

▸ ORA_SERVER_ERROR

Parameter: the position on the error stack, where 1 is the top of the error stack.

Return: an error number as a NUMBER datatype.

```
DECLARE
  ERROR NUMBER;
BEGIN
  FOR i IN 1 .. ORA_SERVER_ERROR_DEPTH LOOP
    ERROR := ORA_SERVER_ERROR(i);
  END LOOP;
END;
```

▸ ORA_SERVER_ERROR_DEPTH

Return: the number of errors on the error stack as a PLS_INTEGER datatype.

▸ ORA_SERVER_ERROR_MSG

Parameter: Position on the error stack, where 1 is the top of the error stack.

Return: an error message text as a VARCHAR2 datatype.

```
DECLARE
  ERROR VARCHAR2(64);
BEGIN
  FOR i IN 1 .. ORA_SERVER_ERROR_DEPTH LOOP
    ERROR := ORA_SERVER_ERROR_MSG(i);
  END LOOP;
END;
```

▸ ORA_SERVER_ERROR_NUM_PARAMS

Return: the count of any subsituted strings from error messages as a PLS_INTEGER datatype.

▸ ORA_SERVER_ERROR_PARAM

Parameter: the position in an error message, where 1 is the first occurrence of a string in the error message.

Return: an error message text as a VARCHAR2 datatype.

```
DECLARE
  PARAM VARCHAR2(32);
BEGIN
  FOR i IN 1 .. ORA_SERVER_ERROR_DEPTH LOOP
    FOR j IN 1 .. ORA_SERVER_ERROR_NUM_PARAMS(i) LOOP
      PARAM := ORA_SERVER_ERROR_PARAM(j);
    END LOOP;
  END LOOP;
END;
```

▸ ORA_SQL_TXT

Parameter: A table of VARCHAR2(64) datatypes containing the substrings of the procedded SQL statement that triggered the event.

Return: the number of element in the list indexed by a PLS_INTEGER datatype.

▸ ORA_SYSEVENT

Return: the system event that was responsible for firing the trigger as a VARCHAR2 datatype.

```
BEGIN
```

```
  INSERT INTO LOGGING_TABLE
  VALUES(ORA_SYSEVENT || ' fired the trigger.');
END;
```

▸ ORA_WITH_GRANT_OPTION

Return: True or False value as a BOOLEAN datatype indicating whether a privilege is granted with grant option.

```
DECLARE
  USER_LIST DBMS_STANDARD.ORA_NAME_LIST_T;
  COUNTER PLS_INTEGER;
BEGIN
  IF ORA_SYSEVENT = 'GRANT' THEN
    COUNTER := ORA_GRANTEE(USER_LIST);
    IF ORA_WITH_GRANT_OPTION THEN
      FOR i IN USER_LIST.FIRST .. USER_LIST.LAST LOOP
        INSERT INTO LOGGING_TABLE
        VALUES(i || ' is granted privilege with grant option');
      END LOOP;
    END IF;
  END IF;
END;
```

▸ SPACE_ERROR_INFO

Parameter: error number, error type, object owner, table space name, object name, sub object name.

Return: True or False indicating whether the triggering event is related to an out-of-space condition or not. When the value is true, it also updates the variables since they are passed by reference.

```
DECLARE
  ERROR_NUMBER NUMBER;
  ERROR_TYPE VARCHAR2(12);
  OBJECT_OWNER VARCHAR2(30);
  TABLESPACE_NAME VARCHAR2(30);
  OBJECT_NAME VARCHAR2(128);
  SUBOBJECT_NAME VARCHAR2(30);
BEGIN
  IF SPACE_ERROR_INFO(ERROR_NUMBER, ERROR_TYPE,
                      OBJECT_OWNER, TABLESPACE_NAME,
                      OBJECT_NAME, SUBOBJECT_NAME) THEN
    INSERT INTO LOGGING_TABLE
    VALUES( /* implementation dependent */);
  END IF;
END;
```

2 DML Triggers

DML triggers can fire before or after INSERT, UPDATE, and DELETE statements. They can be statement-level or row-level activities.

▸ Statement-level triggers are executed once for each DML statement no matter how many rows are affected by the statement.

▹ Row-level triggers are executed for each row changed by a DML statement.

The prototype for building DML trigger is:

```
CREATE [OR REPLACE]
TRIGGER trigger_name
{BEFORE | AFTER}
{INSERT | UPDATE | UPDATE OF column1, [column 2 [, column(n+1)]] | DELETE}
ON table_name
[FOR EACH ROW]
[WHEN (logical_expression)]
[DECLARE]
  [PRAGMA AUTONOMOUS_TRANSACTION;]
  declaration_statements;
BEGIN
  execution_statements;
END [trigger_name];
```

The BEFORE or AFTER clause determines whether the trigger fires before or after the change is written to your local copy of the data. You can define a BEFORE or AFTER clause on tables but not views. While the prototype shows either an insert, update, update of (a column), or delete, you can also use an inclusion OR operator between the events. Using one OR between two events creates a single trigger that runs for two events. You can create a trigger that supports all four possible events with multiple inclusion operations.
You can use FOR EACH ROW clause to indicate that the trigger should fire for each row as opposed to once per statement. The WHEN clause acts as a filter specifying when the trigger fires.

2.1 Statement-Level Triggers

Statement-level triggers are also known as table-level triggers because they are triggered by a change to a table. They capture and process information when a user inserts, updates, or deletes one or more rows in a table. You can also restrict (filter) UPDATE statement triggers by constraining them to fire only when a specific column value changes. You can restrict the trigger by using a UPDATE OF clause. The clause can apply to a column name or a comma-delimited list of column names. You cannot use a WHEN clause in a statement-level trigger.
Consider that we want to keep a log of what DML statements are performed in our database. We have a table EVALUATIONS_LOG(EVENT_DATE, ACTION) to keep track of the actions happening on the EVALUATIONS table:

```
CREATE OR REPLACE
TRIGGER EVAL_CHANGE_TRACKER
AFTER INSERT OR UPDATE OR DELETE
ON EVALUATIONS
DECLARE
  CURR_ACTION EVALUATIONS_LOG.ACTION%TYPE;
BEGIN
  IF INSERTING THEN
    CURR_ACTION := 'Insert';
  ELSIF UPDATING THEN
    CURR_ACTION := 'Update';
  ELSIF DELETING THEN
    CURR_ACTION := 'Delete';
  END IF;
```

```
  INSERT INTO EVALUATIONS_LOG
  VALUES(SYSDATE, CURR_ACTION);
END;
/
```

Note that, it is possible that the trigger can fail if you have already declared another `EVAL_CHANGE_TRACKER` trigger on another table. The `REPLACE` command only works when the `CREATE OR REPLACE TRIGGER` command works against the same table.

2.2 Row-Level Triggers

Row-level triggers let you capture new and prior values from each row. This information can let you audit changes, analyze behavior, and recover prior data without performing a database recovery operation.
There are two pseudo-records when you use the `FOR EACH ROW` clause in a row-level trigger. They both refer to the columns referenced in the DML statement. The pseudo-records are composite variables; `NEW` and `OLD` are the pseudo-record variable names in the `WHEN` clause, and `:NEW` and `:OLD` are the bind variables in the trigger body. They differ because the trigger declaration and body are separate PL/SQL blocks. These pseudo-records can be used to access the columns that are changed by the DML statement:

- For an `INSERT` trigger, `OLD` contains no value, and `NEW` contains the new values.
- For an `UPDATE` trigger, `OLD` contains the old values, and `NEW` contains the new values.
- For a `DELETE` trigger, `OLD` contains the old values, and `NEW` contains no value.

Assume that when a student enrolls in a course, the credit for that course is added to their total credit. Let's also assume that, one student cannot take more than 180 credits. To make sure that the student does not violate this rule, we can use a trigger:

```
CREATE OR REPLACE
TRIGGER TOTAL_CREDIT_LIMIT_CHECKER
BEFORE INSERT ON TAKES
FOR EACH ROW
DECLARE
  COURSE_CREDIT COURSE.CREDITS%TYPE;
  CURRENT_TOT_CRED STUDENT.TOT_CRED%TYPE;
  NEW_TOT_CRED STUDENT.TOT_CRED%TYPE;
BEGIN
  SELECT TOT_CRED INTO CURRENT_TOT_CRED
  FROM STUDENT
  WHERE STUDENT.ID = :NEW.ID;
  SELECT CREDITS INTO COURSE_CREDIT
  FROM COURSE
  WHERE COURSE.COURSE_ID = :NEW.COURSE_ID;
  NEW_TOT_CRED := CURRENT_TOT_CRED + COURSE_CREDIT;
  IF NEW_TOT_CRED > 180 THEN
    RAISE_APPLICATION_ERROR(-2000, 'Registering for ' || :NEW:COURSE_ID || '
   exceeds the total credit limit of 180.');
  ELSE
    UPDATE STUDENT
    SET TOT_CRED = NEW_TOT_CRED
    WHERE STUDENT.ID = :NEW.ID;
```

```
  END IF;
END;
/
```

Here, RAISE_APPLICATION_ERROR procedure is used to issue an user-defined error. It not only shows the error message, but also halts the execution of the INSERT statement.
Let's consider another example. Assume that you have developed a website where users can register by providing their username and password. The username can contain alphanumeric characters and underscore. If, by mistake, some user enters the username with whitespace in it, you want to replace them with underscore. The following program can be helpful:

```
CREATE OR REPLACE
TRIGGER SANITIZE_USERNAME
BEFORE INSERT OR UPDATE OF USERNAME ON USER
FOR EACH ROW
WHEN (REGEXP_LIKE(NEW.USERNAME, ' '))
BEGIN
  :NEW.USERNAME := REGEXP_REPLACE(:NEW.USERNAME, ' ', '_', 1, 1);
END;
/
```

The WHEN clause checks whether the value of the pseudo-field for the USERNAME column in the USER table contains a whitespace or not. If the condition is met, the trigger passes control to the trigger body. The trigger body has one statement; the REGEXP_REPLACE function takes a copy of the pseudo-field as an actual parameter. It changes any whitespace in the string to an underscore, and returns the modified value as a result. The result is assigned to the pseudo-field, and becomes the value in the INSERT/UPDATE statement. The reason we consider UPDATE statement here is because if it were only for the INSERT statement, the user could register and then update their profile to put whitespace in their usernames. This is an example of using a DML trigger to enforce a business policy of entering all usernames with alphanumeric characters and underscore.

Side Note: Sequences

Sequences are counting structures that maintain a persistent awareness of their current value. They are simple to create when you want them to start at 1 and increment by 1. The basic sequence also sets no cache, minimum, or maximum values and accepts both NOCYCLE and NOORDER properties. A sequence caches values by groups of 20 by default, but you can overwrite the cache size when creating the sequence or by altering it after creation. Following is the syntax for building a generic SEQUENCE:

```
CREATE SEQUENCE sequence_name
MINVALUE value
MAXVALUE value
START WITH value
INCREMENT BY value
CACHE value;
```

One example use case for sequences can be the following:

```
CREATE SEQUENCE STUDENT_SEQ
MINVALUE 1001
MAXVALUE 9999
START WITH 1
```

```
INCREMENT BY 1
CACHE 20;

CREATE OR REPLACE
TRIGGER STUDENT_ID_GENERATOR
BEFORE INSERT ON STUDENT
FOR EACH ROW
DECLARE
  NEW_ID STUDENT.ID%TYPE;
BEGIN
  SELECT STUDENT_SEQ.NEXTVAL INTO NEW_ID
    FROM DUAL;
  :NEW.ID := NEW_ID;
END;
/
```

This process is simplified in Oracle 11g:

```
CREATE OR REPLACE
TRIGGER STUDENT_ID_GENERATOR
BEFORE INSERT ON STUDENT
FOR EACH ROW
BEGIN
  :NEW.ID := STUDENT_SEQ.NEXTVAL;
END;
/
```

Many designs simply build these generic sequences and enable rows to be inserted by the web application interface. Some tables require specialized setup rows. These rows are inserted by administrators. They often use special primary key values below the numbering sequence assigned the regular application. When you have a table requiring manual setup rows, some developers leave the first 100 values and start the sequence at 101. Other applications leave more space and start the sequence at 1001. Both approaches are designed to provide your application with the flexibility to add new setup rows after initial implementation. This lets you isolate setup row values in a range different than ordinary applications data.

Sequences are typically built to support primary key columns in tables. Primary key columns impose a combination contraint on their values - they use both `UNIQUE` and `NOT NULL` constraints. During normal Ornline Transaction Processing (OLTP), some insertions are rolled back because other transactional components fail. When transactions are rolled back, the captured sequence value is typically lost. This means that you may see numeric gaps in the primary key column sequence values. Typically, you ignore small gaps. Larger gaps in sequence values occur during after-hours batch processing where you are performing bulk inserts into tables. Failures in batch processing typically involve operation staff intervention in conjunction with programming teams to fix the failure and process the data. Part of fixing this type of failure is resetting the next sequence value. While it would be nice to simply use an `ALTER` statement to reset the next sequence value, you cannot reset the `START WITH` number using an `ALTER` statement. YOu can reset every other criterion of a sequence, but you must drop and recreate the sequence to change the `START WITH` value.

There are three steps in the process to successfuly modify a sequence `START WITH` value: querying the primary key that uses the sequence to find the highest current value, dropping the existing sequence with the `DROP SEQUENCE sequence_name;` command, and recreating the sequence with a `START WITH` value one greater than the highest value in the primary key column. Naturally, the gap does not hurt anything, and you can skip this step, but as a rule, it is recommended to eliminate gaps during maintenance operations.

3 Order of Firing Triggers

Oracle allows more than one trigger to be created for the same timing point, but it has never guaranteed the execution order of those triggers. Oracle 11g trigger syntax now includes the FOLLOWS clause to guarantee execution order for triggers defined with the same timing point. The following example creates a table with two triggers for the same timing point:

```
CREATE TABLE TRIGGER_FOLLOWS_TEST
(
  ID NUMBER,
  DESCRIPTION VARCHAR2(50);
);

CREATE OR REPLACE
TRIGGER TRIGGER_FOLLOWS_TEST_TRG_1
BEFORE INSERT ON TRIGGER_FOLLOWS TEST
FOR EACH ROW
BEGIN
  DBMS_OUTPUT.PUT_LINE('TRIGGER_FOLLOWS_TEST_TRG_1 - executed');
END;
/

CREATE OR REPLACE
TRIGGER TRIGGER_FOLLOWS_TEST_TRG_2
BEFORE INSERT ON TRIGGER_FOLLOWS TEST
FOR EACH ROW
BEGIN
  DBMS_OUTPUT.PUT_LINE('TRIGGER_FOLLOWS_TEST_TRG_2 - executed');
END;
/
```

If we insert data into the table, there is no guarantee of the execution order. Oracle 11g provides FOLLOWS clause that can be used to specify the one trigger will follow another.

```
CREATE OR REPLACE
TRIGGER TRIGGER_FOLLOWS_TEST_TRG_2
BEFORE INSERT ON TRIGGER_FOLLOWS_TEST
FOR EACH ROW
FOLLOWS TRIGGER_FOLLOWS_TEST_TRG_1
BEGIN
  DBMS_OUTPUT.PUT_LINE('TRIGGER_FOLLOWS_TEST_TRG_2 - executed');
END;
/
```

4 Task - Group B

Consider the schema shown in Figure 6.1 for the database of a university:

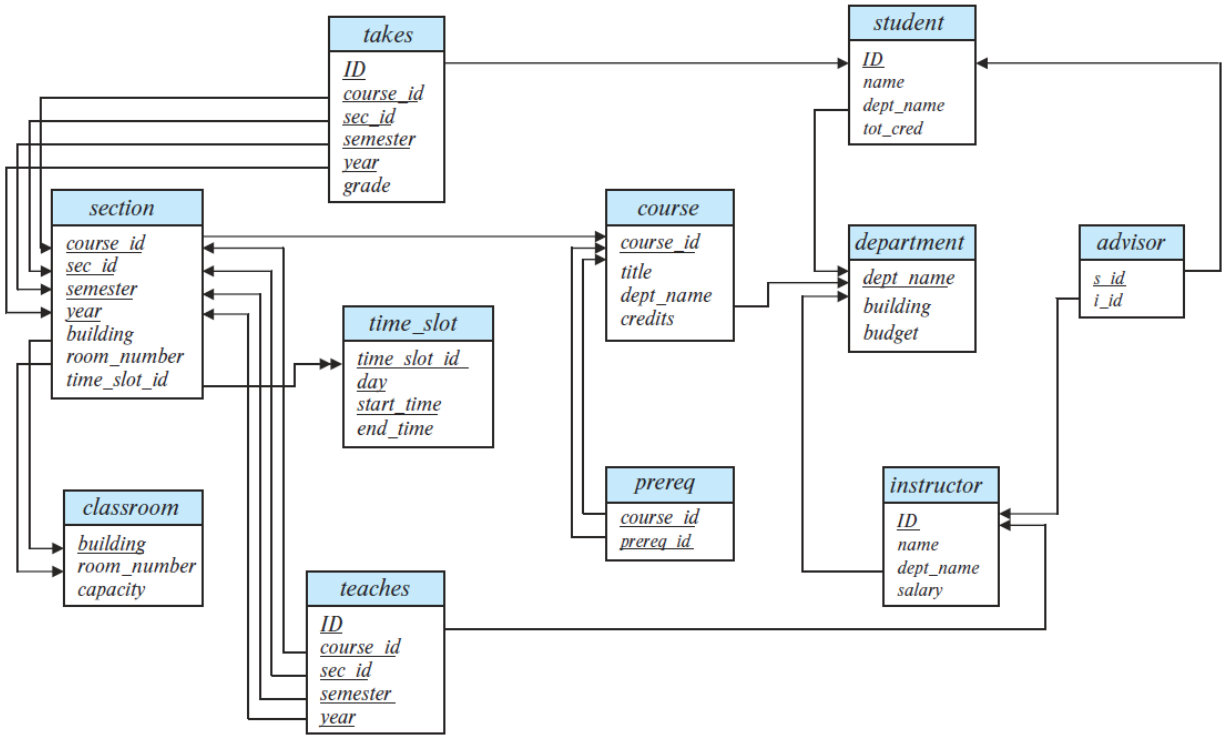

Figure 6.1. Schema diagram for a university database

Write the command `@"<file_path>\<file_name>.sql"` in your SQL command line to execute the provided .sql files. Now, write PL/SQL statements to perform each of the following tasks:

1. Create a function that takes the department name of a student as input and generates a student ID in the form: XYYYY. Here, X is 1 for Biology, 2 for Comp. Sci., 3 for Elec. Eng., 4 for Finance, 5 for History, 6 for Music, and 7 for Physics. Again, for each department, YYYY will be generated from a sequence starting from 1. If the digit count of the number generated by the sequence is less than 4, then 0s will be padded to the left of the number to make it 4 digits.
2. Using the function created in task 1, update the IDs of the existing students. Assume that the students of a particular department are enrolled in the alphabetical order of their names.
3. Write a trigger to automatically generate the IDs of the newly admitted student.
4. Write a trigger that automatically updates the `tot_cred` of a student when they enroll in a particular course.
5. Write a trigger that enrolls a newly admitted student to all the courses offered by their own department, that do not have any prerequisites. If a student is admitted to a department that does not offer any courses, then enroll them in courses offered by other departments that do not have any prerequisites. In case the course is not offered in the current semester and year (more on that later), a new section will be created for the course to enroll the student. The semester will be Winter if the current date falls between November to February, Summer if it falls between March to July, and Fall if it falls between August to October. The selected room should reside in the same building as the department. The time slot must be empty and selected sequentially from Monday to Friday.