SQL Practice Exercises

Exercise 1: Creating Tables

Objective: Create two tables, `Students` and `Courses`.

Instructions:

1. Create a table called `Students` with the following columns:

```sql
CREATE TABLE Students (
    StudentID INTEGER PRIMARY KEY,
    FirstName TEXT,
    LastName TEXT,
    Age INTEGER
);
```

2. Create a table called `Courses` with the following columns:

```sql
CREATE TABLE Courses (
    CourseID INTEGER PRIMARY KEY,
    CourseName TEXT,
    StudentID INTEGER,
    FOREIGN KEY (StudentID) REFERENCES Students(StudentID)
);
```

Exercise 2: Inserting Data

Objective: Insert data into the `Students` and `Courses` tables.

Instructions:

1. Insert the following data into the `Students` table:

```sql
INSERT INTO Students (StudentID, FirstName, LastName, Age) VALUES
(1, 'Alice', 'Johnson', 20),
(2, 'Bob', 'Smith', 22),
(3, 'Carol', 'Williams', 19);
```

2. Insert the following data into the `Courses` table:

```sql
INSERT INTO Courses (CourseID, CourseName, StudentID) VALUES
(1, 'Math', 1),
(2, 'Science', 2),
(3, 'History', 1),
(4, 'Literature', 3);
```

Exercise 3: Basic SELECT Queries

Objective: Retrieve data from the `Students` and `Courses` tables.

Instructions:

1. Select all columns from the `Students` table.

```sql
SELECT * FROM Students;
```

2. Select the first name and last name of all students.

```sql
SELECT FirstName, LastName FROM Students;
```

3. Select the names of all courses.

```sql
SELECT CourseName FROM Courses;
```

Exercise 4: WHERE Clause

Objective: Use the `WHERE` clause to filter data.

Instructions:

1. Select all students who are older than 20 years.

   ```sql
   SELECT * FROM Students WHERE Age > 20;
   ```

2. Select all courses taken by student with `StudentID` 1.

   ```sql
   SELECT * FROM Courses WHERE StudentID = 1;
   ```

Exercise 5: JOIN Queries

Objective: Use `JOIN` to combine data from multiple tables.

Instructions:

1. Perform an `INNER JOIN` to select students and their courses.

   ```sql
   SELECT Students.FirstName, Students.LastName, Courses.CourseName
   FROM Students
   INNER JOIN Courses ON Students.StudentID = Courses.StudentID;
   ```

2. Perform a `LEFT JOIN` to select all students and their courses (including those without any courses).

   ```sql
   SELECT Students.FirstName, Students.LastName, Courses.CourseName
   FROM Students
   LEFT JOIN Courses ON Students.StudentID = Courses.StudentID;
   ```

Exercise 6: UPDATE and DELETE

Objective: Update and delete data in the tables.

Instructions:

1. Update the age of student `Bob Smith` to 23.

   ```sql
   UPDATE Students
   SET Age = 23
   WHERE FirstName = 'Bob' AND LastName = 'Smith';
   ```

2. Delete the course with `CourseID` 4.

   ```sql
   DELETE FROM Courses WHERE CourseID = 4;
   ```

Exercise 7: Aggregate Functions

Objective: Use aggregate functions to perform calculations on data.

Instructions:

1. Count the number of students.

   ```sql
   SELECT COUNT(*) AS NumberOfStudents FROM Students;
   ```

2. Find the average age of students.

   ```sql
   SELECT AVG(Age) AS AverageAge FROM Students;
   ```

Exercise 8: GROUP BY Clause

Objective: Use the `GROUP BY` clause to group data and use aggregate functions.

Instructions:

1. Group students by age and count the number of students in each age group.

```sql
SELECT Age, COUNT(*) AS NumberOfStudents
FROM Students
GROUP BY Age;
```

2. Group courses by `StudentID` and count the number of courses each student is taking.

   ```sql
   SELECT StudentID, COUNT(*) AS NumberOfCourses
   FROM Courses
   GROUP BY StudentID;
   ```

Exercise 9: HAVING Clause

Objective: Use the `HAVING` clause to filter grouped data.

Instructions:

1. Select age groups that have more than one student.

   ```sql
   SELECT Age, COUNT(*) AS NumberOfStudents
   FROM Students
   GROUP BY Age
   HAVING COUNT(*) > 1;
   ```

2. Select students who are enrolled in more than one course.

   ```sql
   SELECT StudentID, COUNT(*) AS NumberOfCourses
   FROM Courses
   GROUP BY StudentID
   HAVING COUNT(*) > 1;
   ```

Exercise 10: ORDER BY Clause

Objective: Use the `ORDER BY` clause to sort data.

Instructions:

1. Select all students and sort them by last name in ascending order.

   ```sql
   SELECT * FROM Students
   ORDER BY LastName ASC;
   ```

2. Select all courses and sort them by course name in descending order.

   ```sql
   SELECT * FROM Courses
   ORDER BY CourseName DESC;
   ```

Exercise 11: LIMIT Clause

Objective: Use the `LIMIT` clause to restrict the number of returned rows.

Instructions:

1. Select the first two students from the `Students` table.

   ```sql
   SELECT * FROM Students
   LIMIT 2;
   ```

2. Select the top two courses from the `Courses` table.

   ```sql
   SELECT * FROM Courses
   LIMIT 2;
   ```

Exercise 12: DISTINCT Clause

Objective: Use the `DISTINCT` clause to return unique values.

Instructions:

1. Select all unique ages from the `Students` table.

   ```
   SELECT DISTINCT Age FROM Students;
   ```

2. Select all unique course names from the `Courses` table.

   ```
   SELECT DISTINCT CourseName FROM Courses;
   ```

Exercise 13: Subqueries

Objective: Use subqueries to perform complex queries.

Instructions:

1. Select the names of students who are taking the 'Math' course.

   ```
   SELECT FirstName, LastName FROM Students
   WHERE StudentID IN (SELECT StudentID FROM Courses WHERE CourseName = 'Math');
   ```

2. Select the names of courses taken by students who are older than 20.

   ```
   SELECT CourseName FROM Courses
   WHERE StudentID IN (SELECT StudentID FROM Students WHERE Age > 20);
   ```

Exercise 14: ALTER TABLE

Objective: Modify the structure of an existing table.

Instructions:

1. Add a column `Email` to the `Students` table.

   ```
   ALTER TABLE Students
   ADD COLUMN Email TEXT;
   ```

2. Remove the `Age` column from the `Students` table.

   ```
   ALTER TABLE Students
   DROP COLUMN Age;
   ```

Exercise 15: Creating Indexes

Objective: Create indexes to improve query performance.

Instructions:

1. Create an index on the `LastName` column of the `Students` table.

   ```
   CREATE INDEX idx_lastname ON Students(LastName);
   ```

2. Create a unique index on the `CourseName` column of the `Courses` table.

   ```
   CREATE UNIQUE INDEX idx_coursename ON Courses(CourseName);
   ```

Exercise 16: Using Transactions

Objective: Use transactions to ensure data integrity.

Instructions:

1. Start a transaction, insert a new student, and then commit the transaction.

   ```
   BEGIN TRANSACTION;
   INSERT INTO Students (StudentID, FirstName, LastName, Age) VALUES (4, 'Dave', 'Miller', 21);
   COMMIT;
   ```

2. Start a transaction, insert a new course, and then roll back the transaction.

```
BEGIN TRANSACTION;
INSERT INTO Courses (CourseID, CourseName, StudentID) VALUES (5, 'Art', 3);
ROLLBACK;
```

Exercise 17: Using Views

Objective: Create and use views to simplify complex queries.

Instructions:

1. Create a view called `StudentCourses` that shows student names and their courses.

   ```
   CREATE VIEW StudentCourses AS
   SELECT Students.FirstName, Students.LastName, Courses.CourseName
   FROM Students
   INNER JOIN Courses ON Students.StudentID = Courses.StudentID;
   ```

2. Select data from the `StudentCourses` view.

   ```
   SELECT * FROM StudentCourses;
   ```

Exercise 18: Using Triggers

Objective: Create and use triggers to automate database tasks.

Instructions:

1. Create a trigger to log changes to the `Students` table.

   ```
   CREATE TABLE StudentChanges (
       ChangeID INTEGER PRIMARY KEY,
       StudentID INTEGER,
       ChangeType TEXT,
       ChangeTime TIMESTAMP DEFAULT CURRENT_TIMESTAMP
   );

   CREATE TRIGGER LogStudentChanges
   AFTER INSERT ON Students
   BEGIN
       INSERT INTO StudentChanges (StudentID, ChangeType) VALUES (NEW.StudentID, 'INSERT');
   END;
   ```

2. Test the trigger by inserting a new student.

   ```
   INSERT INTO Students (StudentID, FirstName, LastName, Age) VALUES (5, 'Eve', 'Taylor', 23);
   SELECT * FROM StudentChanges;
   ```

Exercise 19: Handling NULL Values

Objective: Work with NULL values in the database.

Instructions:

1. Select all students who do not have an email address.

   ```
   SELECT * FROM Students WHERE Email IS NULL;
   ```

2. Update the email address for student `Alice Johnson` and then select all students again.

   ```
   UPDATE Students SET Email = 'alice.johnson@example.com' WHERE StudentID = 1;
   SELECT * FROM Students;
   ```

Exercise 20: Exporting and Importing Data

Objective: Export data to a file and import data from a file.

Instructions:

1. Export the `Students` table to a CSV file.

   ```
   .mode csv
   .headers on
   .output students.csv
   SELECT * FROM Students;
   .output stdout
   ```

2. Import data from a CSV file into the `Courses` table.

   ```
   .mode csv
   .import courses.csv Courses
   SELECT * FROM Courses;
   ```