The Impact of Data Scale and Indexing on Query Performance

Objective

This lab is designed to provide a hands-on understanding of how database performance is affected by two critical factors: **the volume of data** and the use of **database indexes**. You will create a university database, populate it with a growing number of records, measure query execution times, and then analyze the dramatic performance improvements that indexing can provide.

Tools

- **Database:** PostgreSQL, MySQL, or any other relational database.
- **Programming Language:** Python for data generation and scripting.
- **Python Libraries:**
    - **`Faker`** : To generate realistic fake data (names, dates, addresses, etc.).
    - **`psycopg2-binary`** (for PostgreSQL) or **`mysql-connector-python`** (for MySQL): To connect Python to your database.
    - **`matplotlib`** or **`seaborn`** : For creating graphs to visualize your results.

Part 1: Database Schema Design

First, you need to design the structure of your university database. Create a new database (e.g., `university_db`) and then create the following five tables. Pay close attention to the primary keys (PK) and foreign keys (FK) which establish relationships between the tables.

1. `Departments` Table This table stores information about academic departments.

- `department_id` (PK, Integer, Auto-incrementing)
- `department_name` (Varchar) - e.g., 'Computer Science', 'Physics'
- `building` (Varchar) - e.g., 'Block A', 'Science Wing'

2. `Teachers` Table Stores information about the teachers. Each teacher belongs to one department.

- `teacher_id` (PK, Integer, Auto-incrementing)
- `first_name` (Varchar)
- `last_name` (Varchar)
- `email` (Varchar, Unique)
- `department_id` (FK, Integer) - References `Departments.department_id`
- `hire_date` (Date)

3. `Courses` Table Contains details about the courses offered. Each course is taught by a single teacher.

- `course_id` (PK, Integer, Auto-incrementing)
- `course_name` (Varchar) - e.g., 'Introduction to Algorithms', 'Quantum Mechanics'
- `credits` (Integer) - e.g., 3
- `teacher_id` (FK, Integer) - References `Teachers.teacher_id`

4. `Students` Table This is your largest table, storing student information.

- `student_id` (PK, Integer, Auto-incrementing)
- `first_name` (Varchar)
- `last_name` (Varchar)
- `email` (Varchar, Unique)
- `enrollment_date` (Date)
- `date_of_birth` (Date)

5. `Enrollments` Table This is a "junction table" that links students to the courses they are enrolled in, creating a many-to-many relationship.

- `enrollment_id` (PK, Integer, Auto-incrementing)
- `student_id` (FK, Integer) - References `Students.student_id`
- `course_id` (FK, Integer) - References `Courses.course_id`
- `semester` (Varchar) - e.g., 'Fall 2024'
- `grade` (Integer) - A score from 0 to 100.

Part 2: Data Generation and Insertion

Your task is to write a Python script to populate these tables. Use the **Faker** library to generate realistic data.

Data Generation Plan:

- **Departments:** 10 departments.
- **Teachers:** 100 teachers, distributed among the 10 departments.
- **Courses:** 200 courses, with each course assigned to a random teacher.
- **Students & Enrollments:** This is where you will scale the data. For each student, enroll them in 5-10 random courses.

Your script should be able to generate data for the following scales:

1. **Scale 1:** 1,000 Students (approx. 5k-10k enrollments)
2. **Scale 2:** 10,000 Students (approx. 50k-100k enrollments)
3. **Scale 3:** 100,000 Students (approx. 500k-1M enrollments)
4. **Scale 4:** 1,000,000 Students (approx. 5M-10M enrollments)

Part 3: The Queries (To be Timed)

You will run the following five queries at each data scale and record the execution time.

Query 1: Simple Filter

- **Task:** Find all students who enrolled in the year 2023.
- **Complexity:** A simple `WHERE` clause on the largest table (`Students`).

Query 2: Simple Join and Filter

- **Task:** Find the email addresses of all students taught by a specific teacher (e.g., a teacher with `teacher_id` = 50).
- **Complexity:** Involves joining `Students` , `Enrollments` , and `Courses` .

Query 3: Multi-Join with Text Search

- **Task:** List the full names of all teachers who teach a course with the word 'Advanced' in its name.
- **Complexity:** Involves joining `Teachers` and `Courses` with a `LIKE` clause for text matching.

Query 4: Join with Aggregation (GROUP BY)

- **Task:** For each department, count how many courses are being offered. List the department name and the total course count.
- **Complexity:** Requires joining `Departments` , `Teachers` , and `Courses` , and using `COUNT` with `GROUP BY` .

Query 5: Complex Join with Aggregation, Filtering, and Sorting

- **Task:** Find the top 10 students with the highest average grade in the 'Spring 2025' semester. Display the student's full name and their calculated average grade, sorted from highest to lowest.
- **Complexity:** This is the most challenging query. It requires joining `Students` and `Enrollments` , filtering by semester, grouping by student, calculating an average using `AVG` , sorting the results, and taking only the top 10 using `LIMIT` .

Part 4: The Experiment

Follow these steps methodically.

Phase 1: Performance Without Indexes

1. Start with a clean, un-indexed set of tables (besides the default Primary Key indexes).
2. **Run for Scale 1 (1k students):**
   - Insert the data.
   - Run each of the 5 queries 3 times and record the average execution time for each.
3. **Run for Scale 2 (10k students):** Repeat the process.
4. **Run for Scale 3 (100k students):** Repeat the process.
5. **Run for Scale 4 (1M students):** Repeat the process.

Phase 2: Introducing Indexes

Based on the queries you ran, identify the columns that are frequently used in `WHERE` clauses, `JOIN` conditions, and `ORDER BY` clauses. Just for an idea, you might consider indexing the following columns (but feel free to adjust based on your query patterns):

- On the `Students` table: an index on the `enrollment_date` column.
- On the `Enrollments` table: indexes on `student_id` , `course_id` , and `semester` .
- On the `Courses` table: an index on `teacher_id` and `course_name` .
- On the `Teachers` table: an index on `department_id` .

Phase 3: Performance With Indexes

1. Using your **1 Million student dataset** with the newly created indexes, re-run all 5 queries.
2. Run each query 3 times and record the average execution time for each.

Part 5: Analysis and Visualization

Create two graphs to visualize your findings.

Graph 1: Query Performance vs. Data Scale

- **Type:** Line Chart
- **X-axis:** Data Size (1k, 10k, 100k, 1M)
- **Y-axis:** Execution Time (in milliseconds)

- **Content:** Plot 5 separate lines, one for each query. This graph will visually demonstrate how the execution time for each query grows as the dataset size increases.

Graph 2: Impact of Indexing on 1 Million Records

- **Type:** Bar Chart
- **X-axis:** Query (Query 1, Query 2, Query 3, Query 4, Query 5)
- **Y-axis:** Execution Time (in milliseconds, consider a logarithmic scale if differences are huge)
- **Content:** For each query, show two bars side-by-side: one for the time **before indexing** and one for the time **after indexing**. This will powerfully illustrate the performance gain from indexing.

Part 6: Lab Report

Submit a report that includes the following:

1. Your SQL schema (`CREATE TABLE` statements).
2. Your Python data generation script.
3. A table containing all the timing data you collected.
4. The two graphs you generated, with clear labels and titles.
5. A conclusion section answering the following questions:
    - Which query was most affected by the increase in data volume? Why do you think that is?
    - Which query saw the most significant performance improvement after indexing? Why?
    - Was there any query that did not improve much with indexing? If so, explain why that might be.
    - What are the potential downsides of adding too many indexes to a database? (Think about `INSERT`, `UPDATE`, and `DELETE` operations).