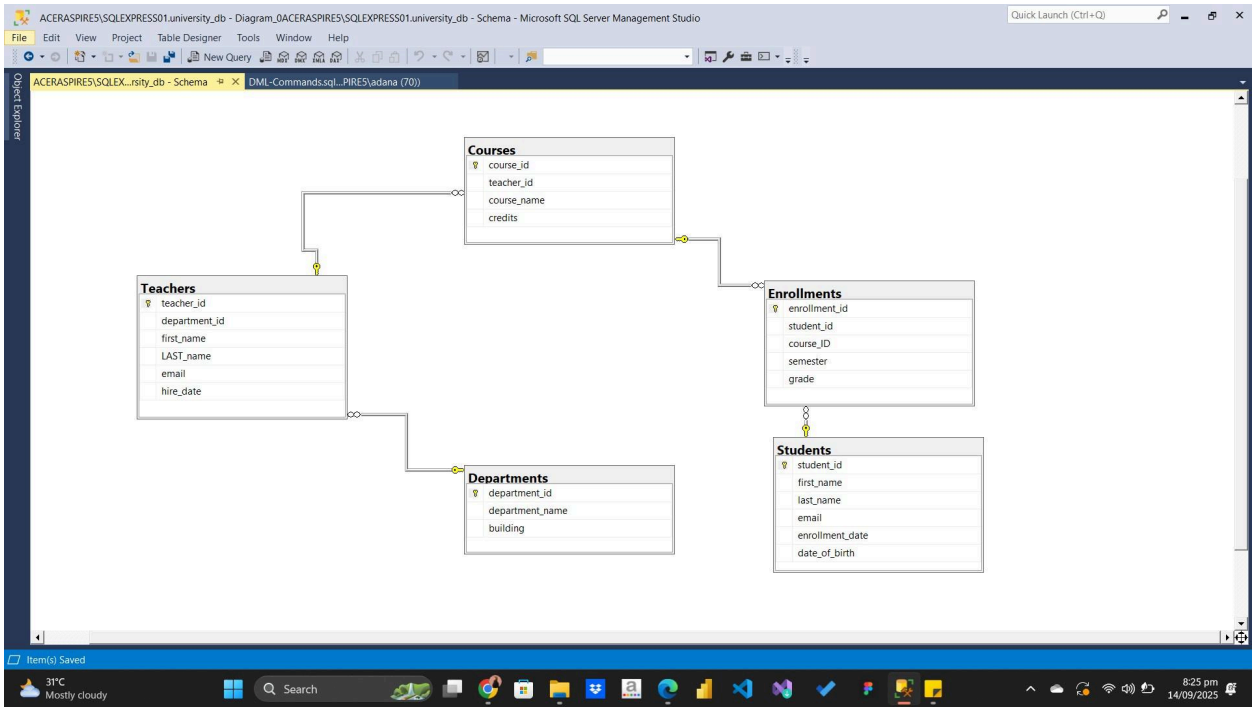
A Study on Query Optimization and Indexing Trade-offs

1.
SQL Schema (CREATE TABLE Statements)



2.
Python Data Generation Script

Click the link below to open the file:
[Data Generation Script]

3.
Timing Data Collected

Unindexed:

Scale	Query	Run 1 (ms)	Run 2 (ms)	Run 3 (ms)	Avg (ms)
1k	Q1	2	0	5	2.33
1k	Q2	32	20	14	22.00
1k	Q3	3	1	1	1.67
1k	Q4	9	7	8	8.00
1k	Q5	2	3	5	3.33

10k	Q1	110	155	129	131.33
10k	Q2	4	1	2	2.33
10k	Q3	1	0	0	0.33
10k	Q4	6	5	8	6.33
10k	Q5	25	23	8	18.67
100k	Q1	354	274	302	310.00
100k	Q2	2	1	4	2.33
100k	Q3	1	0	2	1.00
100k	Q4	13	6	6	8.33
100k	Q5	220	210	216	215.33
1M	Q1	1833	1754	1716	1767.67
1M	Q2	10982	8975	9437	9798.00
1M	Q3	1	10	0	3.67
1M	Q4	7	6	7	6.67
1M	Q5	2362	2378	2096	2278.67

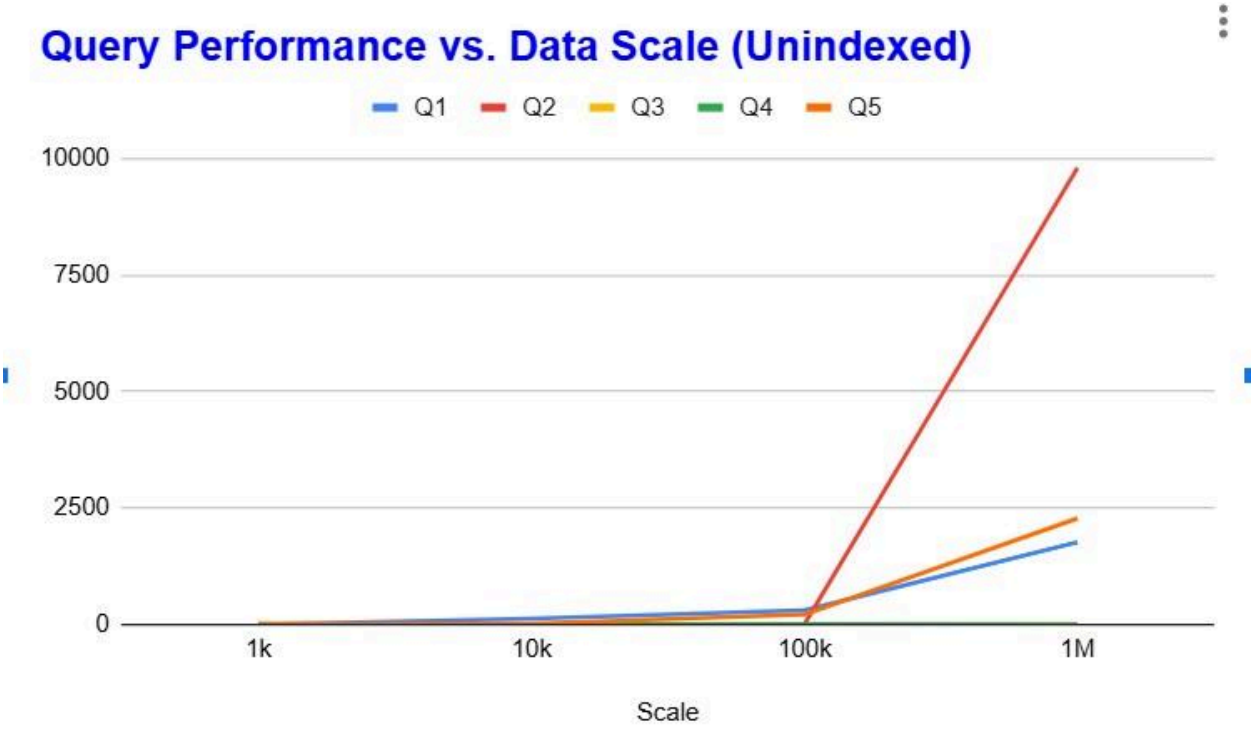
Indexed:

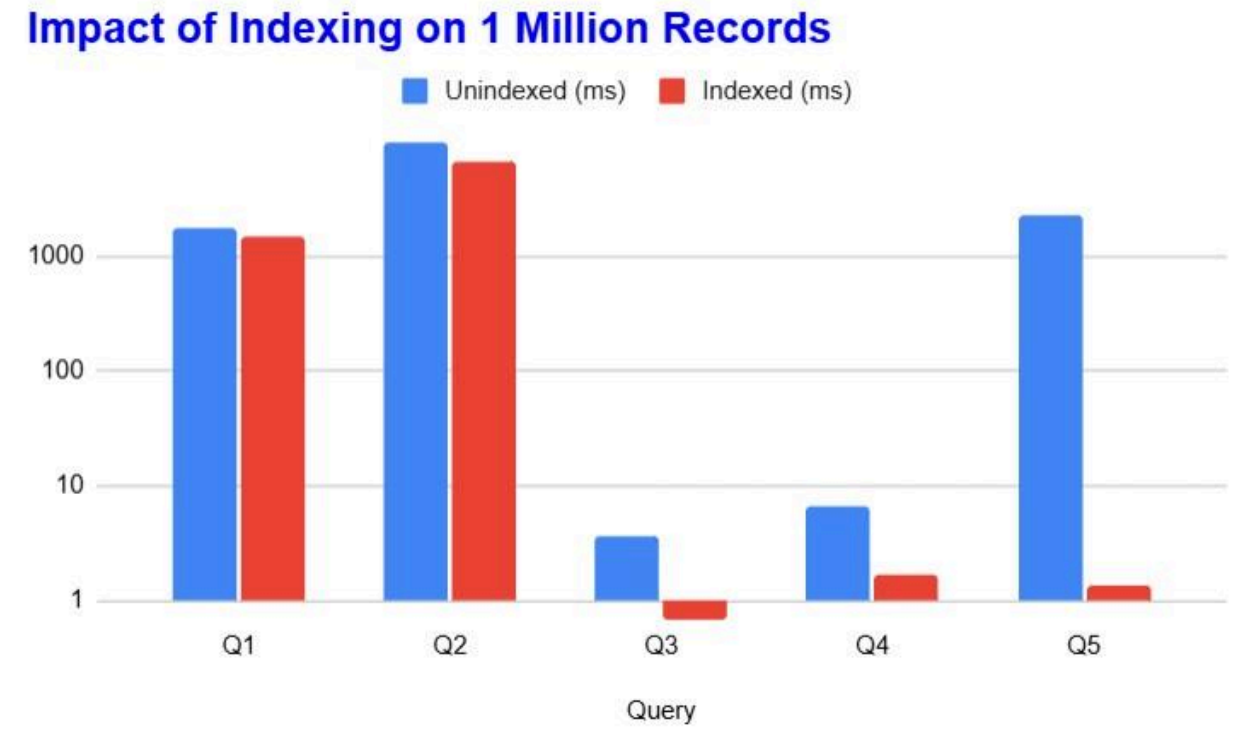
Scale	Query	Run 1 (ms)	Run 2 (ms)	Run 3 (ms)	Avg (ms)
1k	Q1	6	0	0	2.00
1k	Q2	5	1	2	2.67
1k	Q3	3	0	0	1.00
1k	Q4	4	0	1	1.67
1k	Q5	2	0	0	0.67
10k	Q1	143	155	166	154.67
10k	Q2	1	0	0	0.33
10k	Q3	3	0	0	1.00
10k	Q4	6	0	0	2.00
10k	Q5	1	0	0	0.33
100k	Q1	277	214	235	242.00
100k	Q2	1	0	0	0.33
100k	Q3	3	0	4	2.33
100k	Q4	6	1	1	2.67
100k	Q5	3	0	0	1.00

1M	Q1	1610	1359	1470	1479.67
1M	Q2	6878	7306	6201	6795.00
1M	Q3	2	0	0	0.67
1M	Q4	3	1	1	1.67
1M	Q5	4	0	0	1.33

4.

Performance Graph





5.
Conclusion & Analysis

What is Indexing in Databases?

Think of an index in a book:

- Without an index, if you want to find "Topic X," you'd flip through every single page (slowly).
- With an index at the back, you jump directly to the pages where "Topic X" appears (fast).

A database index works the same way.
It's a data structure (usually B-tree or hash) that helps the database quickly locate rows without scanning the entire table.

- ## How It Works

  Suppose you have a `Students` table with 1 million rows, and you run:

```
SELECT *
FROM Students
WHERE email = 'adan@example.com';
```

  - Without an index:
    Database checks every row (1M comparisons) → full table scan.
  - With an index on `email`:
    The database jumps directly to the memory location of `'adan@example.com'` in logarithmic time (like looking up a word in a dictionary).

1. Which query was most affected by the increase in data volume? Why do you think that is?

- Query 2 (students joined with enrollments and courses filtered by `teacher_id`) was most affected.
- At 1M records (Unindexed), it took ~9.8 seconds compared to just a few milliseconds at smaller scales.

- Reason: This query has multiple joins across large tables, and without indexes, the database has to perform full table scans and join them, which grows very expensive as data volume increases.

2. Which query saw the most significant performance improvement after indexing? Why?

- Query 5 (top 10 students with highest average grade for Spring 2025) saw the biggest gain.
- Execution dropped from ~2279 ms (Unindexed) → ~1.3 ms (Indexed), a ~99.9% improvement.
- Reason: Indexing on `Enrollments.semester` and `Enrollments.student_id` allows the database to quickly filter Spring 2025 records and efficiently group by student, instead of scanning the entire enrollments table.

3. Was there any query that did not improve much with indexing? If so, explain why that might be.

- Query 1 (`WHERE YEAR(enrollment_date) = 2023`) showed only modest improvement (1767 ms → 1479 ms).
- Reason: The `YEAR()` function applied on the column makes the index less useful because it forces a calculation on every row, preventing efficient index seek.

So in summary:

- **Q2** struggled most with large data.
- **Q5** benefitted the most from indexing.
- **Q1** didn't improve much due to function usage on the indexed column.

4. What are the potential downsides of adding too many indexes to a database? (Think about INSERT , UPDATE , and DELETE operations)

1. Slower `INSERT` Operations

- When you insert a new row, the database not only adds it to the table but must also update **all indexes** that reference the indexed columns.

- More indexes = more work = slower inserts.

2. Slower `UPDATE` Operations

- If an updated column is part of an index:
    - The DB must **rearrange the index** (because the old value is removed, and the new value inserted).
- If multiple indexes exist, this overhead multiplies.

3. Slower `DELETE` Operations

- When deleting a row, the DB must remove references to that row from **all related indexes**.
- More indexes → more cleanup → slower deletes.

*If your database is **read-heavy** (like analytics dashboards, reporting, search queries) → indexes are worth it.*

*If it's **write-heavy** (like logging systems, real-time transactions) → too many indexes can hurt performance.*