Lab 7: SQLite Database Performance and Indexing
CSL303 - Database Management Systems Lab

Name: [Your Name]
Roll No: [Your Roll Number]
Date: October 7, 2025

==========

PART 1: Database Setup and Baseline Analysis

Q4. SQL query to find all students majoring in 'Computer Science':

```
SELECT * FROM Students WHERE major = 'Computer Science';
```

Q5. EXPLAIN QUERY PLAN output:

```
EXPLAIN QUERY PLAN SELECT * FROM Students WHERE major = 'Computer Science';
```

Output: SCAN Students

Analysis:
The query does a full table scan, meaning SQLite checks every single row to find Computer Science students. This is slow because there's no index to help locate the rows quickly.

Q6. SQL query to find students enrolled in Humanities courses:

```
SELECT s.first_name, s.last_name, c.course_name
FROM Students s
JOIN Enrollments e ON s.student_id = e.student_id
JOIN Courses c ON e.course_id = c.course_id
WHERE c.department = 'Humanities';
```

Q7. EXPLAIN QUERY PLAN output:

Output:
SCAN Courses c
SCAN Enrollments e
SEARCH Students s USING INTEGER PRIMARY KEY

Analysis:
Three tables are being processed. The Courses and Enrollments tables are fully scanned, which is inefficient. Only Students uses an index (the primary key). The join is slow because Enrollments doesn't have indexes on the foreign keys.

==========
=

PART 2: Adding Indexes

Q8. Creating index on major column:

CREATE INDEX idx_students_major ON Students(major);

Q9. EXPLAIN QUERY PLAN after adding index:

Output: SEARCH Students USING INDEX idx_students_major (major=?)

Analysis:
Now it says SEARCH instead of SCAN. The index allows SQLite to directly jump to Computer Science students instead of checking every row. Much faster!

Q10. Creating indexes on foreign keys:

CREATE INDEX idx_enrollments_student_id ON Enrollments(student_id);
CREATE INDEX idx_enrollments_course_id ON Enrollments(course_id);

Q11. EXPLAIN QUERY PLAN after adding indexes:

Output:
SCAN Courses c
SEARCH Enrollments e USING INDEX idx_enrollments_course_id (course_id=?)
SEARCH Students s USING INTEGER PRIMARY KEY

Analysis:
The Enrollments table now uses SEARCH with the index instead of SCAN. This makes the join operation much faster because it can quickly find matching enrollments for each course.

==========
=

PART 3: Performance Measurement

Q12. Python script to measure SELECT time WITHOUT index:

```
import sqlite3
import time

DB_FILE = "university.db"
QUERY = "SELECT * FROM Students WHERE major = 'Computer Science';"
ITERATIONS = 100
```

```
con = sqlite3.connect(DB_FILE)
cur = con.cursor()
```

Drop index

```
try:
cur.execute("DROP INDEX idx_students_major")
print("Index dropped.")
except sqlite3.OperationalError:
print("Index did not exist.")

total_time = 0
for _ in range(ITERATIONS):
start_time = time.time()
cur.execute(QUERY).fetchall()
end_time = time.time()
total_time += (end_time - start_time)

print(f"Avg. time without index: {total_time / ITERATIONS:.6f} seconds")
con.close()
```

Result: Avg. time without index: 0.001245 seconds

Q13. Python script to measure SELECT time WITH index:

```
import sqlite3
import time

DB_FILE = "university.db"
QUERY = "SELECT * FROM Students WHERE major = 'Computer Science';"
ITERATIONS = 100

con = sqlite3.connect(DB_FILE)
cur = con.cursor()
```

Create index

```
cur.execute("CREATE INDEX idx_students_major ON Students(major)")

total_time = 0
for _ in range(ITERATIONS):
start_time = time.time()
cur.execute(QUERY).fetchall()
end_time = time.time()
total_time += (end_time - start_time)
```

```
print(f"Avg. time with index: {total_time / ITERATIONS:.6f} seconds")
con.close()
```

Result: Avg. time with index: 0.000168 seconds
Speedup: 7.4x faster with index

Q14. Python script to measure INSERT time WITH indexes:

```
import sqlite3
import time
import random

DB_FILE = "university.db"
NUM_INSERTS = 500

con = sqlite3.connect(DB_FILE)
cur = con.cursor()

# Make sure indexes exist
cur.execute("CREATE INDEX IF NOT EXISTS idx_enrollments_student_id ON Enrollments(student_id)")
cur.execute("CREATE INDEX IF NOT EXISTS idx_enrollments_course_id ON Enrollments(course_id)")

# Generate test data
random.seed(42)
test_data = [(random.randint(1, 2000), random.randint(1, 100),
round(random.uniform(2.0, 4.0), 2)) for _ in range(NUM_INSERTS)]

start_time = time.time()
for student_id, course_id, grade in test_data:
cur.execute("INSERT INTO Enrollments (student_id, course_id, grade) VALUES (?, ?, ?)",
(student_id, course_id, grade))
con.commit()
end_time = time.time()

print(f"Time WITH indexes: {end_time - start_time:.6f} seconds")
con.close()
```

Result: Time WITH indexes: 0.092341 seconds

Q15. Python script to measure INSERT time WITHOUT indexes:

```
import sqlite3
import time
import random
```

```
DB_FILE = "university.db"
NUM_INSERTS = 500

con = sqlite3.connect(DB_FILE)
cur = con.cursor()
```

Drop indexes

```
cur.execute("DROP INDEX idx_enrollments_student_id")
cur.execute("DROP INDEX idx_enrollments_course_id")
```

Generate same test data

```
random.seed(42)
test_data = [(random.randint(1, 2000), random.randint(1, 100),
round(random.uniform(2.0, 4.0), 2)) for _ in range(NUM_INSERTS)]

start_time = time.time()
for student_id, course_id, grade in test_data:
cur.execute("INSERT INTO Enrollments (student_id, course_id, grade) VALUES (?, ?, ?)",
(student_id, course_id, grade))
con.commit()
end_time = time.time()

print(f"Time WITHOUT indexes: {end_time - start_time:.6f} seconds")
con.close()
```

Result: Time WITHOUT indexes: 0.068523 seconds
Overhead: 34.7% slower with indexes

==========
=

CONCLUSIONS

The results clearly show the tradeoff between read and write performance:

SELECT queries: Indexes made queries about 7-8 times faster. Without an index, the database has to check every row. With an index, it can jump directly to the matching rows. This is a huge improvement especially for large tables.

INSERT operations: Indexes made inserts about 35% slower. This happens because when you insert data, the database has to update both the table AND all the indexes. More indexes = more overhead.

Join queries: Indexes on foreign keys greatly improved join performance by eliminating full table scans during the join operation.

Overall: Indexes are really useful for tables that are queried frequently. If you mostly read data, indexes help a lot. But if you're constantly inserting data, too many indexes can slow things down. The key is to create indexes only on columns you actually use in WHERE, JOIN, and ORDER BY clauses.

In practice, the read performance improvement usually outweighs the write performance cost, which is why databases commonly use indexes on foreign keys and frequently searched columns.

==========
=