CIS 552 Database Design

Project Report

Performance Analysis of Normalized vs Non-Normalized Databases

GitHub Repository Link: https://github.com/Madhumithaakken/CIS552-database-design-project.git

Full Name: Akkenapally Madhumitha

Student ID: 02255825

1. Introduction

Databases play a critical role in modern applications, and the way data is designed and organized can greatly affect how efficiently a system performs. One common approach in relational database design is normalization, which structures data into multiple related tables to reduce redundancy and maintain data consistency. While normalization improves data quality and makes databases easier to maintain, it can sometimes slow down query performance because additional tables must be accessed and combined.

In real-world systems, database designers must carefully balance performance, scalability, and data integrity. Non-normalized databases often store all information in a single table, which can make simple queries faster and easier to execute. However, this approach leads to repeated data, larger storage requirements, and potential update problems as the dataset grows. On the other hand, normalized databases reduce duplication and improve consistency, but may require more complex queries involving joins across multiple tables.

This project explores the performance differences between non-normalized and normalized database designs using an academic salary dataset. Identical queries are executed on datasets of increasing size (1MB, 10MB, and 100MB) to observe how each design responds to different workloads. The queries include simple data retrieval, conditional filtering, highly selective searches, and aggregation operations.

By measuring execution times and visualizing the results, this study shows that normalization does not always lead to faster queries, but it can significantly improve performance for certain types of operations. The goal of this project is to better understand when normalization helps, when it does not, and how thoughtful database design decisions impact overall system performance.

2. Dataset Description

The data set used in this project contains salary and employment information for individuals working in academic institutions. It includes details about people, their job roles, associated schools, campuses, and yearly earnings. This type of dataset is well-suited for studying database normalization because it naturally contains repeated and related information across multiple records.

To analyze how database performance changes with scale, the dataset was prepared in three different sizes:

- 1MB
- 10MB
- 100MB

Each dataset represents the same type of information but with an increasing number of records. This allows us to observe how query execution time grows as the volume of data increases and how different database designs handle scalability.

Key attributes in the dataset include:

- Person details such as ID, name, and birthdate
- Job-related information such as job title and department
- School details including school name and campus
- Employment information such as earnings, earnings year, and working status

The same dataset was used for both the non-normalized and normalized database schemas to ensure a fair comparison. By keeping the data consistent and only changing the database design, the performance differences observed in this project can be directly attributed to normalization rather than data variation.

3. Tools Required:

The following tools and technologies were used throughout this project to design the database, execute queries, measure performance, and visualize results.

- **MySQL 8.x**

  Used as the primary database management system to store both non-normalized and normalized datasets, execute SQL queries, and manage indexing.

- **MySQL Workbench**

  Used as a graphical interface to interact with the MySQL database. It helped verify table structures, run SQL queries manually, and inspect query execution behavior.

- **Python 3.14**

Used to automate the execution of queries and measure their execution times in a consistent and repeatable manner.

- **MySQL Connector for Python**

  Enabled Python scripts to connect to the MySQL database and execute SQL queries programmatically.

- **Pandas**

  Used to process and organize query execution results stored in CSV files for further analysis.

- **matplotlib**

  Used to generate graphs that visually compare query performance across different dataset sizes and between normalized and non-normalized schemas.

- **Git and GitHub**

  Used for version control, project organization, and maintaining a clean and reproducible project repository.

Together, these tools provided a reliable environment for conducting performance experiments, analyzing results, and presenting findings in a clear and structured manner.

4. Database Design

4.1 Non-normalization schema

The non-normalized schema stores all attributes in a single table, directly loaded with CSV files:

- raw_data_1MB
- raw_data_10MB
- raw_data_100MB

Table Creation:

All non-normalized tables share the same schema. The following SQL statement shows the table structure used to store the raw data:

```
CREATE TABLE raw_data_1MB (
PersonID INT,
PersonName VARCHAR(255),
BirthDate DATE,
JobTitle VARCHAR(255),
DepartmentName VARCHAR(255),
SchoolName VARCHAR(255),
SchoolCampus VARCHAR(255),
Earnings DECIMAL(12,2),
EarningsYear INT,
StillWorking VARCHAR(10)
);
```

The same schema was reused for raw_data_10MB and raw_data_100MB.

Loading Data from CSV files:

The data was loaded directly from CSV files using the LOAD DATA LOCAL INFILE command. This approach preserves the original denormalized structure of the dataset.

```
LOAD DATA LOCAL INFILE 'C:/data/salary_tracker_1MB.csv'
INTO TABLE raw_data_1MB
FIELDS TERMINATED BY ','
ENCLOSED BY '"'
LINES TERMINATED BY '\n'
IGNORE 1 ROWS;
```

Querying on Non-Normalized Data:

- Q1: Person name and birthdate
  ```
  SELECT DISTINCT PersonName, BirthDate
  FROM raw_data_1MB;
  ```
- Q2: Working employees and their school

```
SELECT DISTINCT PersonName, SchoolName, SchoolCampus
FROM raw_data_1MB
WHERE StillWorking = 'yes';
```

- Q3: Assistant Professors at UMass Dartmouth

```
SELECT PersonName, JobTitle
FROM raw_data_1MB
WHERE JobTitle = 'Assistant Professor'
AND SchoolName = 'University of Massachusetts'
AND SchoolCampus = 'Dartmouth'
AND StillWorking = 'yes';
```

- Q4: Count people per campus for latest year

```
SELECT SchoolCampus, COUNT(DISTINCT PersonID) AS num_people
FROM raw_data_1MB
WHERE EarningsYear = (SELECT MAX(EarningsYear) FROM raw_data_1MB)
AND StillWorking = 'yes'
GROUP BY SchoolCampus;
```

- Q5: Total earnings per person

```
SELECT PersonID, PersonName, SUM(Earnings) AS total_earnings
FROM raw_data_1MB
GROUP BY PersonID, PersonName;
```

I have queried all the tables by these queries.

Measuring Query Performance on Non-Normalized Data

To measure the execution time of queries on the non-normalized tables, a Python script named *measure_across_tables.py* was developed. This script automates the execution of SQL queries and ensures that performance measurements are consistent and reproducible across different dataset sizes.

Code Implementation

The following code snippet shows the Python script used to measure query performance on non-normalized tables:

```
# measure_across_tables.py

import os

import time

import csv

import statistics
```

```
from dotenv import load_dotenv
import mysql.connector
# Load database credentials
load_dotenv()
cfg = {
    'host': os.getenv('MYSQL_HOST', 'localhost'),
    'user': os.getenv('MYSQL_USER'),
    'password': os.getenv('MYSQL_PASSWORD'),
    'database': os.getenv('MYSQL_DATABASE'),
    'connect_timeout': 10
}
# Non-normalized tables and sizes
TABLES = [
    ("raw_data_1MB", "1"),
    ("raw_data_10MB", "10"),
    ("raw_data_100MB", "100")
]
# SQL queries
QUERIES = {
    'Q1': "SELECT DISTINCT PersonName, BirthDate FROM {table};",
    'Q2': "SELECT DISTINCT PersonName, SchoolName, SchoolCampus FROM {table} WHERE StillWorking='yes';",
    'Q3': "SELECT PersonName, JobTitle FROM {table} WHERE JobTitle='Assistant Professor' AND SchoolName='University of Massachusetts' AND SchoolCampus='Dartmouth' AND StillWorking='yes';",
    'Q4': "SELECT SchoolCampus, COUNT(DISTINCT PersonID) FROM {table} WHERE EarningsYear=(SELECT MAX(EarningsYear) FROM {table}) AND StillWorking='yes' GROUP BY SchoolCampus;",
```

```
    'Q5': "SELECT PersonID, PersonName, SUM(Earnings) FROM {table} GROUP BY PersonID, PersonName;"
}
WARMUPS = 2
REPEATS = 7
def measure_query(cursor, sql):
    for _ in range(WARMUPS):
        cursor.execute(sql)
        cursor.fetchall()
    times = []
    for _ in range(REPEATS):
        start = time.perf_counter()
        cursor.execute(sql)
        cursor.fetchall()
        times.append(time.perf_counter() - start)
    return statistics.median(times)
# Main execution
conn = mysql.connector.connect(**cfg)
cur = conn.cursor(buffered=True)

results = []
for table, size in TABLES:
    for qid, query in QUERIES.items():
        median_time = measure_query(cur, query.format(table=table))
        results.append([table, size, qid, median_time])
cur.close()
conn.close()
```

```
with open("results_all_tables.csv", "w", newline="") as f:
    writer = csv.writer(f)
    writer.writerow(["table", "size_mb", "query", "median_seconds"])
    writer.writerows(results)
```

Output:

```
=== Running queries on raw_data_1MB (1MB) ===
Q1 → median: 0.022751s
Q2 → median: 0.011824s
Q3 → median: 0.005886s
Q4 → median: 0.009562s
Q5 → median: 0.025736s

=== Running queries on raw_data_10MB (10MB) ===
Q1 → median: 0.153253s
Q2 → median: 0.071403s
Q3 → median: 0.050305s
Q4 → median: 0.090101s
Q5 → median: 0.176634s

=== Running queries on raw_data_100MB (100MB) ===
Q1 → median: 2.196986s
Q2 → median: 1.173234s
Q3 → median: 0.945263s
Q4 → median: 1.823574s
Q5 → median: 3.489822s
```

The output of this script is a CSV file containing median execution times for each query across all non-normalized dataset sizes. These results from the baseline for comparing performance with the normalized schema.

4.2 Normalization schema

Table Creation:

To eliminate redundancy and improve data consistency, the dataset was transformed into a normalized relational schema following Third Normal Form (3NF). Each table represents a single logical entity, and relationships are maintained using primary and foreign keys.

```
CREATE TABLE IF NOT EXISTS Person (
  PersonID INT PRIMARY KEY,
  PersonName VARCHAR(255),
  BirthDate VARCHAR(50)
);
CREATE TABLE IF NOT EXISTS School (
```

```
  SchoolID VARCHAR(50) PRIMARY KEY,
  SchoolName VARCHAR(255)
);
CREATE TABLE IF NOT EXISTS Campus (
  CampusID INT AUTO_INCREMENT PRIMARY KEY,
  SchoolID VARCHAR(50),
  CampusName VARCHAR(255),
  FOREIGN KEY (SchoolID) REFERENCES School(SchoolID)
);
CREATE TABLE IF NOT EXISTS Department (
  DepartmentID VARCHAR(50) PRIMARY KEY,
  DepartmentName VARCHAR(255)
);
CREATE TABLE IF NOT EXISTS Job (
  JobID VARCHAR(50) PRIMARY KEY,
  JobTitle VARCHAR(255)
);
CREATE TABLE IF NOT EXISTS Employment (
  EmploymentID INT AUTO_INCREMENT PRIMARY KEY,
  PersonID INT,
  JobID VARCHAR(50),
  SchoolID VARCHAR(50),
  SchoolCampus VARCHAR(255),
  DepartmentID VARCHAR(50),
  StillWorking VARCHAR(10),
  Earnings DOUBLE,
  EarningsYear INT,
```

```
  FOREIGN KEY (PersonID) REFERENCES Person(PersonID),
  FOREIGN KEY (JobID) REFERENCES Job(JobID),
  FOREIGN KEY (SchoolID) REFERENCES School(SchoolID),
  FOREIGN KEY (DepartmentID) REFERENCES Department(DepartmentID)
);
```

Normalizing and Loading Data Using Python:

After creating the normalized schema, the data from the non-normalized tables was transformed and loaded into the normalized tables using a Python script. This approach ensured consistency, avoided duplicate entries, and automated the normalization process.

```
import os
from dotenv import load_dotenv
import mysql.connector
load_dotenv(dotenv_path=os.path.join(os.path.dirname(__file__), '.env'))
cfg = {
   'host': os.getenv('MYSQL_HOST'),
   'user': os.getenv('MYSQL_USER'),
   'password': os.getenv('MYSQL_PASSWORD'),
   'database': os.getenv('MYSQL_DATABASE'),
   'allow_local_infile': True
}
conn = mysql.connector.connect(**cfg)
cur = conn.cursor()
print("Starting normalization from raw_data_100MB...")
cur.execute("""
INSERT IGNORE INTO Person (PersonID, PersonName, BirthDate)
SELECT DISTINCT PersonID, PersonName, BirthDate
```

```
FROM raw_data_100MB;
""")
conn.commit()
cur.execute("""
INSERT IGNORE INTO School (SchoolID, SchoolName)
SELECT DISTINCT SchoolID, SchoolName
FROM raw_data_100MB;
""")
conn.commit()
cur.execute("""
INSERT IGNORE INTO Campus (SchoolID, CampusName)
SELECT DISTINCT SchoolID, SchoolCampus
FROM raw_data_100MB;
""")
conn.commit()
cur.execute("""
INSERT IGNORE INTO Department (DepartmentID, DepartmentName)
SELECT DISTINCT DepartmentID, DepartmentName
FROM raw_data_100MB;
""")
conn.commit()
cur.execute("""
INSERT IGNORE INTO Job (JobID, JobTitle)
SELECT DISTINCT JobID, JobTitle
FROM raw_data_100MB;
""")
conn.commit()
```

```
cur.execute("""
INSERT INTO Employment (PersonID, JobID, SchoolID, SchoolCampus, DepartmentID, StillWorking, Earnings, EarningsYear)
SELECT PersonID, JobID, SchoolID, SchoolCampus, DepartmentID, StillWorking, Earnings, EarningsYear
FROM raw_data_100MB;
""")
conn.commit()
print("Normalization and load complete.")
cur.close()
conn.close()
```

Querying on Normalized data:

- Q1: Person name and birthdate

```
SELECT DISTINCT p.PersonName, p.BirthDate
FROM Person p;
```

- Q2: Working employees and their school

```
SELECT DISTINCT p.PersonName, s.SchoolName, s.SchoolCampus
FROM Person p, Employment e, School s
WHERE p.PersonID = e.PersonID
  AND e.SchoolID = s.SchoolID
  AND e.StillWorking = 'yes';
```

- Q3: Assistant Professors at UMass Dartmouth

```
SELECT p.PersonName, j.JobTitle
FROM Person p, Employment e, Job j, School s
WHERE p.PersonID = e.PersonID
  AND e.JobID = j.JobID
  AND e.SchoolID = s.SchoolID
```

```
AND j.JobTitle = 'Assistant Professor'
AND s.SchoolName = 'University of Massachusetts'
AND s.SchoolCampus = 'Dartmouth'
AND e.StillWorking = 'yes';
```

- Q4: Count people per campus for latest year

```
SELECT s.SchoolCampus, COUNT(DISTINCT p.PersonID) AS num_people
FROM Person p, Employment e, School s
WHERE p.PersonID = e.PersonID
 AND e.SchoolID = s.SchoolID
 AND e.EarningsYear = (SELECT MAX(EarningsYear) FROM Employment)
 AND e.StillWorking = 'yes'
GROUP BY s.SchoolCampus;
```

- Q5: Total earnings per person

```
SELECT p.PersonID, p.PersonName, SUM(e.Earnings) AS total_earnings
FROM Person p, Employment e
WHERE p.PersonID = e.PersonID
GROUP BY p.PersonID, p.PersonName;
```

Measuring Performance on the Normalized Schema

To measure query execution time on the normalized schema, a Python script named measure_normalized_100mb.py was developed. This script executes the normalized queries on the 100MB dataset and records median execution times.

```
# measure_normalized_100mb.py
print("SCRIPT STARTED")
import os
import time
```

```
import csv
import statistics
import socket
from dotenv import load_dotenv
import mysql.connector
# Load .env (must be in the SAME folder)
load_dotenv(dotenv_path=os.path.join(os.path.dirname(__file__), '.env'))

cfg = {
    'host': os.getenv('MYSQL_HOST', 'localhost'),
    'user': os.getenv('MYSQL_USER'),
    'password': os.getenv('MYSQL_PASSWORD'),
    'database': os.getenv('MYSQL_DATABASE'),
    'connect_timeout': 10
}
WARMUPS = 2
REPEATS = 7
# ============================
# OPTIMIZED NORMALIZED QUERIES
# ============================
QUERIES = {
    # Q1
    'Q1': (
        "SELECT PersonName, BirthDate "
        "FROM Person;"
    ),
    # Q2
```

```
    'Q2': (
        "SELECT DISTINCT p.PersonName, s.SchoolName, e.SchoolCampus "
        "FROM Employment e, Person p, School s "
        "WHERE e.PersonID = p.PersonID "
        "AND e.SchoolID = s.SchoolID "
        "AND e.StillWorking = 'yes';"
    ),
    # Q3
    'Q3': (
    "SELECT p.PersonName, j.JobTitle "
    "FROM Employment e, Person p, Job j, School s "
    "WHERE e.PersonID = p.PersonID "
    "AND e.JobID = j.JobID "
    "AND e.SchoolID = s.SchoolID "
    "AND e.StillWorking = 'yes' "
    "AND j.JobTitle = 'Assistant Professor' "
    "AND s.SchoolName = 'University of Massachusetts' "
    "AND e.SchoolCampus = 'Dartmouth';"
),
    # Q4
    'Q4': (
        "SELECT e.SchoolCampus, COUNT(DISTINCT e.PersonID) AS num_people "
        "FROM Employment e "
        "WHERE e.StillWorking = 'yes' "
        "AND e.EarningsYear = (SELECT MAX(EarningsYear) FROM Employment) "
        "GROUP BY e.SchoolCampus;"
    ),
```

```
    # Q5
    'Q5': (
        "SELECT p.PersonID, p.PersonName, t.total_earnings "
        "FROM Person p, "
        "    (SELECT PersonID, SUM(Earnings) AS total_earnings "
        "     FROM Employment "
        "     GROUP BY PersonID) t "
        "WHERE p.PersonID = t.PersonID;"
    )
}
# ============================
# SAFE INDEX CREATION
# ============================
INDEX_SQL = [
    "ALTER TABLE Employment ADD INDEX idx_emp_personid (PersonID)",
    "ALTER TABLE Employment ADD INDEX idx_emp_schoolid (SchoolID)",
    "ALTER TABLE Employment ADD INDEX idx_emp_earningsyear (EarningsYear)",
    "ALTER TABLE Employment ADD INDEX idx_emp_personid_earn (PersonID, Earnings)",
    "ALTER TABLE Employment ADD INDEX idx_emp_stillworking (StillWorking)",
    "ALTER TABLE Job ADD INDEX idx_job_title (JobTitle(100))",
    "ALTER TABLE School ADD INDEX idx_school_name (SchoolName(100))"
]
def measure_query(cursor, sql):
    # warm-up runs
    for _ in range(WARMUPS):
        cursor.execute(sql)
```

```
        cursor.fetchall()
    times = []
    for _ in range(REPEATS):
        start = time.perf_counter()
        cursor.execute(sql)
        cursor.fetchall()
        times.append(time.perf_counter() - start)
    return statistics.median(times), times
def main():
    try:
        conn = mysql.connector.connect(**cfg)
    except Exception as e:
        print("ERROR: Cannot connect to MySQL")
        print(e)
        return
    cur = conn.cursor(buffered=True)
    print("\n=== Creating indexes (safe mode) ===")
    for stmt in INDEX_SQL:
        try:
            cur.execute(stmt)
            print("Created:", stmt)
        except Exception as e:
            print("Index warning (already exists or skipped):", e)
    conn.commit()
    print("Index step completed.\n")
    results = []
    timestamp = time.strftime("%Y-%m-%d %H:%M:%S")
```

```
    host = socket.gethostname()
    print("=== Running OPTIMIZED NORMALIZED queries (100MB) ===")
    for qid, sql in QUERIES.items():
        print(f"Running {qid}")
        try:
            median_time, samples = measure_query(cur, sql)
            print(f"{qid} → median: {median_time:.6f}s")
        except Exception as e:
            print(f"ERROR in {qid}:", e)
            median_time, samples = None, []
        results.append({
            'query': qid,
            'median_s': "" if median_time is None else f"{median_time:.6f}",
            'samples': "" if not samples else ";".join(f"{t:.6f}" for t in samples),
            'timestamp': timestamp,
            'host': host
        })
    cur.close()
    conn.close()
    out_path = os.path.join(os.path.dirname(__file__), "..",
"results_normalized_100MB.csv")
    with open(out_path, "w", newline="") as f:
        writer = csv.DictWriter(
            f,
            fieldnames=['query', 'median_s', 'samples', 'timestamp', 'host']
        )
        writer.writeheader()
```

```
        for r in results:
            writer.writerow(r)
    print("\nSaved results to:", os.path.abspath(out_path))
if __name__ == "__main__":
    main()
```

Output:

```
=== Running OPTIMIZED NORMALIZED queries (100MB) ===
Running Q1
Q1 → median: 0.864768s
Running Q2
Q2 → median: 1.136600s
Running Q3
Q3 → median: 0.000871s
Running Q4
Q4 → median: 0.098018s
Running Q5
Q5 → median: 1.438146s
```

5. Graphical Analysis

Python Script for Visualization (plot_sketch_query.py)

To visually compare the performance of normalized and non-normalized queries, a Python script named plot_sketch_query.py was developed. This script reads the execution time results generated by earlier measurement scripts and produces sketch-style graphs for each query.

The goal of this visualization step is to clearly show:

- How query execution time increases with data size for non-normalized tables
- What a normalized performance at 100MB compares against non-normalized performance

The plotting script was designed to:

- Read execution time results from CSV files
- Group results by query
- Generate one graph per query

- Plot non-normalized results (1MB, 10MB, 100MB)
- Overlay normalized results for direct comparison
- Present results in a clear, hand-drawn (sketch-style) visual format
- This approach makes performance trends easy to understand and interpret.

Python Code for Plotting:

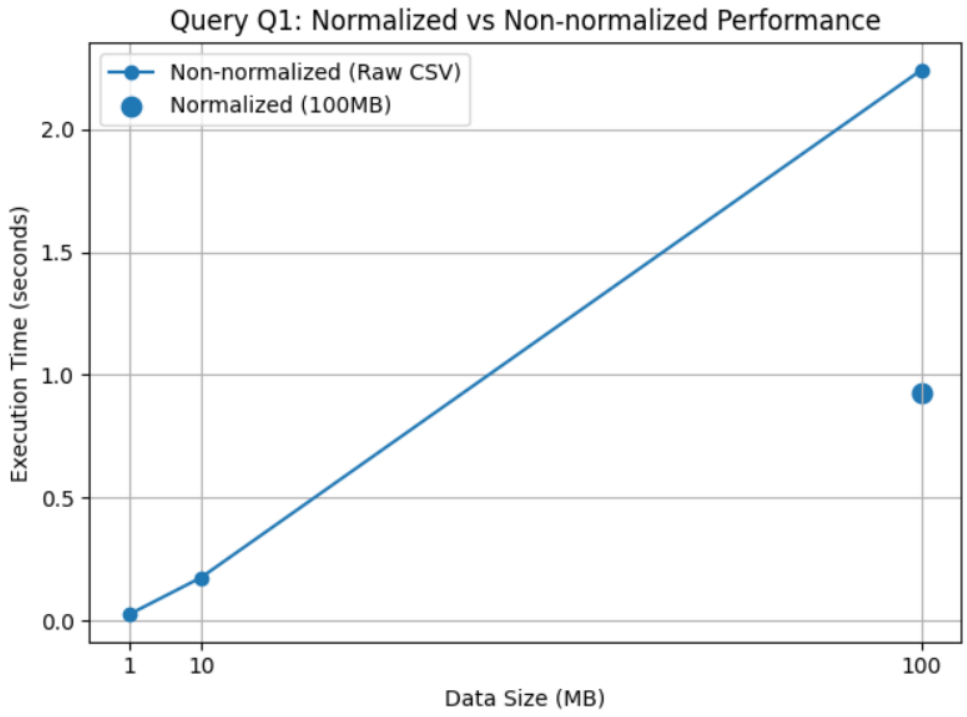
```
# plot_sketch_style.py

# Line graph per query matching hand-drawn sketch

import pandas as pd

import matplotlib.pyplot as plt

import os

BASE_DIR = os.path.dirname(__file__)

RAW_FILE = os.path.join(BASE_DIR, "..", "results_all_tables.csv")

NORM_FILE = os.path.join(BASE_DIR, "..", "results_normalized_100MB.csv")

# Load data

raw_df = pd.read_csv(RAW_FILE)

norm_df = pd.read_csv(NORM_FILE)

raw_df["median_s"] = raw_df["median_s"].astype(float)

raw_df["size_mb"] = raw_df["size_mb"].astype(int)

norm_df["median_s"] = norm_df["median_s"].astype(float)


queries = sorted(raw_df["query"].unique())

OUT_DIR = os.path.join(BASE_DIR, "..", "sketch_style_graphs")

os.makedirs(OUT_DIR, exist_ok=True)

# Plot per query

for q in queries:

    raw_q = raw_df[raw_df["query"] == q].sort_values("size_mb")

    norm_q = norm_df[norm_df["query"] == q]
```

```
plt.figure()
# Raw line (1MB → 10MB → 100MB)
plt.plot(
    raw_q["size_mb"],
    raw_q["median_s"],
    marker="o",
    linestyle="-",
    label="Non-normalized (Raw CSV)"
)
# Normalized point (only 100MB)
if not norm_q.empty:
    plt.scatter(
        [100],
        norm_q["median_s"],
        s=80,
        label="Normalized (100MB)"
    )
plt.xlabel("Data Size (MB)")
plt.ylabel("Execution Time (seconds)")
plt.title(f"Query {q}: Normalized vs Non-normalized Performance")
plt.xticks([1, 10, 100])
plt.grid(True)
plt.legend()
plt.tight_layout()
out_file = os.path.join(OUT_DIR, f"{q}_sketch_style.png")
plt.savefig(out_file)
plt.show()
```

```
    print(f"Saved {out_file}")
print("All sketch-style graphs created.")
```

Graphs:

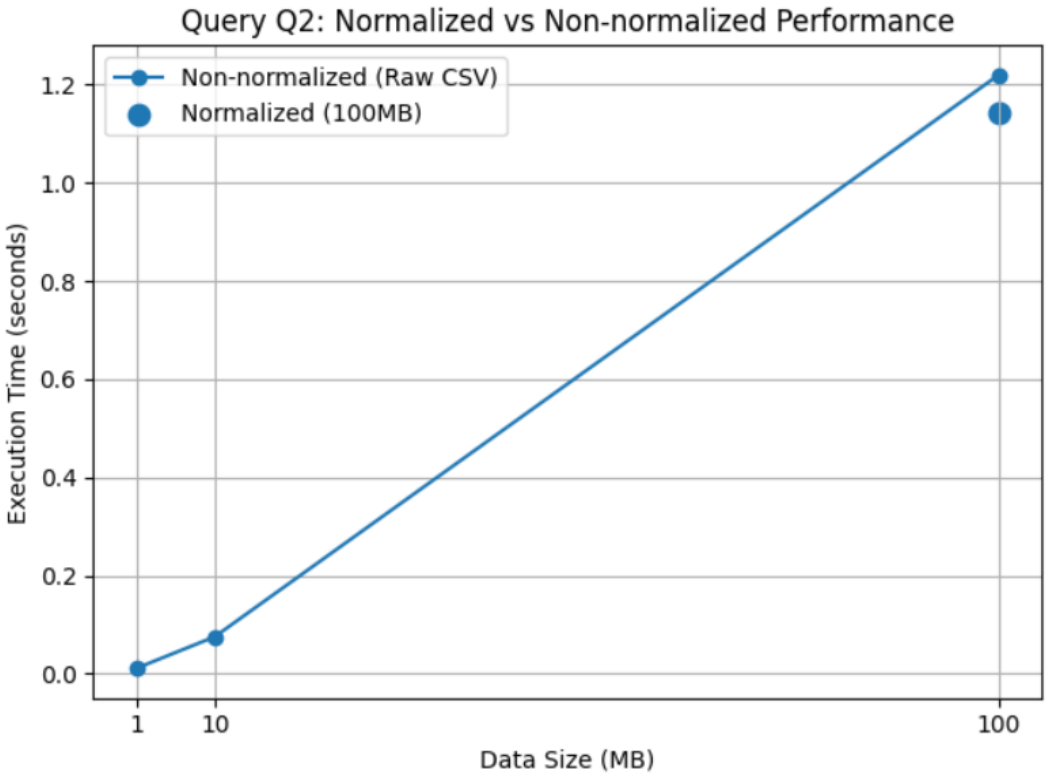
Query 1:



Observation:

- Execution time for the non-normalized schema increases almost linearly as data size grows from 1MB to 100MB.
- The normalized schema shows significantly lower execution time at 100MB, despite involving multiple tables.
- This indicates that removing redundant personal attributes reduces the amount of data scanned.
- Normalization improves scalability even for simple projection queries.

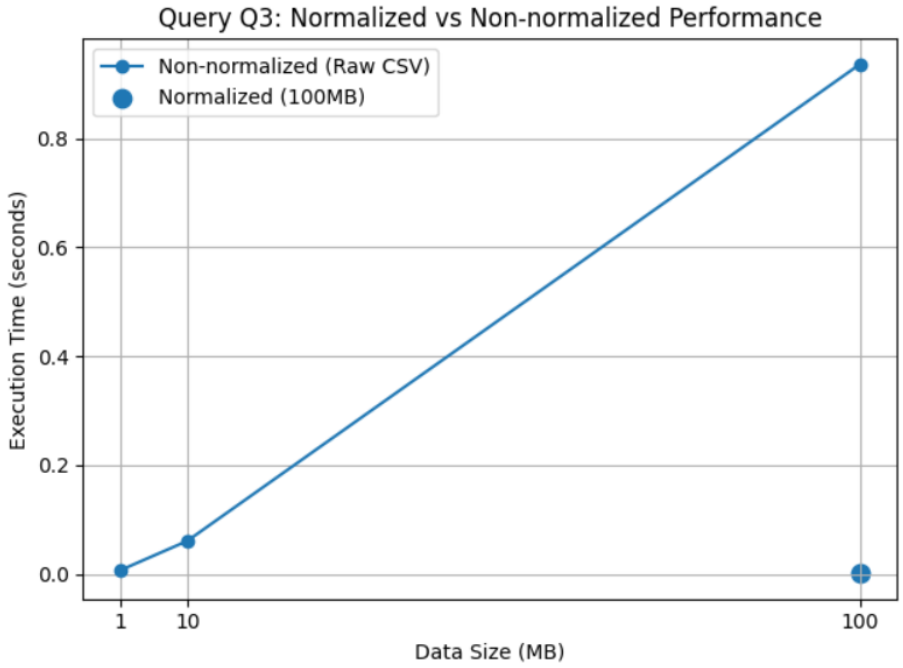
Query 2:



Observation:

- The non-normalized schema experiences a sharp increase in execution time at higher data sizes due to repeated school and employment information.
- The normalized query performs slightly faster at 100MB, showing improved efficiency when filtering across related entities.
- Although the performance gain is moderate, normalization reduces unnecessary repeated comparisons.
- This highlights normalization's benefit for filtering queries involving multiple attributes.

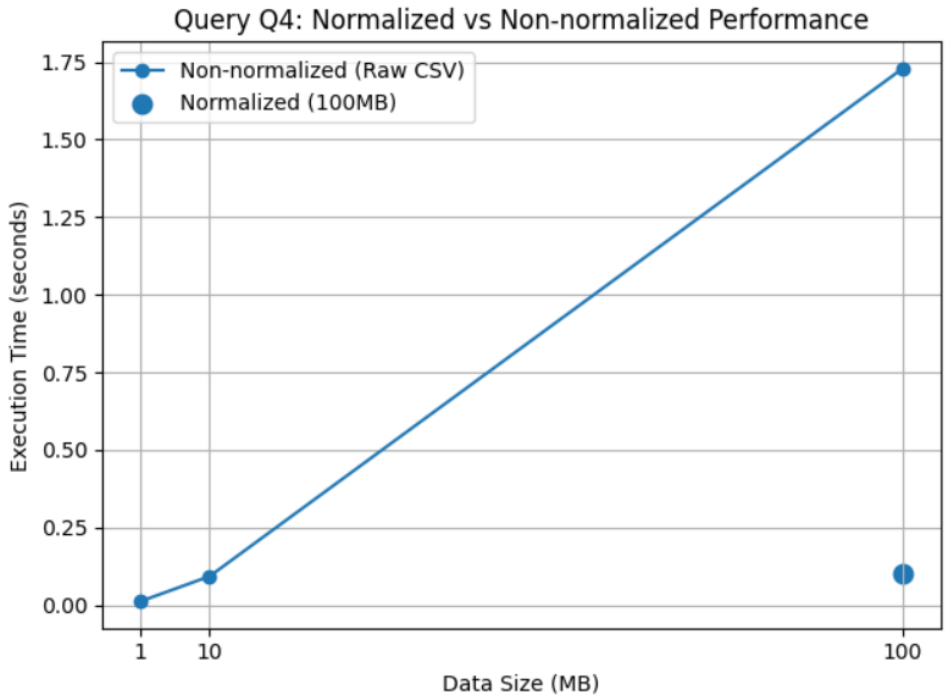
Query 3:



Observation:

- The non-normalized query shows a steep increase in execution time at 100MB.
- The normalized query executes almost instantly at the same data size.
- This demonstrates that selective queries benefit greatly from normalized schemas.
- Applying conditions on smaller, well-structured tables significantly improves performance.

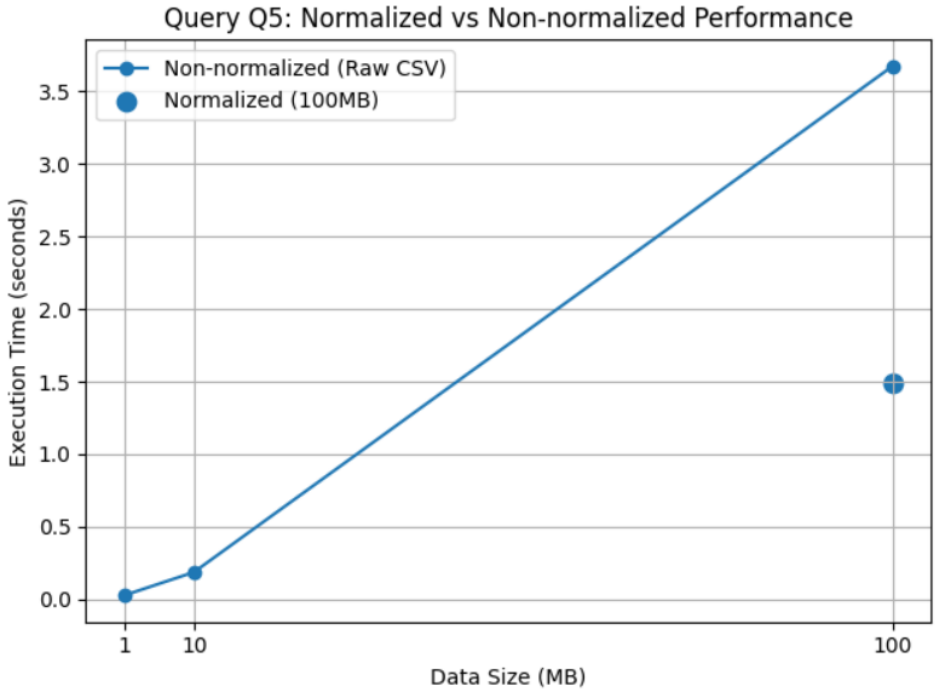
Query 4:



Observation:

- Aggregation over the non-normalized schema becomes increasingly expensive as data size grows.
- The normalized schema performs the same aggregation much faster at 100MB.
- This indicates reduced grouping overhead due to separation of entities.
- Normalization is highly effective for analytical queries involving grouping and counting.

Query 5:



Observation:

- The non-normalized schema shows the highest execution time increase for cumulative aggregation queries.
- The normalized query performs better at 100MB but still incurs noticeable cost due to aggregation.
- While normalization does not eliminate aggregation overhead, it reduces redundant data processing.
- This confirms normalization's advantage for large-scale summary queries.

6. Conclusion

In this project, we compared the performance of non-normalized and normalized database designs using datasets of different sizes. The non-normalized approach stored all data in a single table, which made queries simple and easy to write. However, as the size of the data increased, this design showed clear performance issues due to repeated and redundant information.

The normalized schema organized the data into multiple related tables, reducing duplication and improving structure. Although normalized queries involved accessing more than one table, they performed better when the dataset became large. This improvement was especially noticeable for selective queries and aggregation-based queries, where normalization helped reduce unnecessary data scanning and processing.

The results show that while non-normalized databases may work well for small datasets, they do not scale efficiently. Normalization helps databases handle larger amounts of data more effectively and leads to more consistent performance as data grows. This project demonstrates that good database design is not just about making queries simpler, but about choosing a structure that performs well as the system scales.

Overall, the findings emphasize that normalization is an important design choice for building scalable and efficient database systems, especially when working with large datasets.