**Name: Nikhil Prema chandra rao**
**Student ID: 02105149**

# Title: CSV Query Performance Benchmark - Documentation

---

## 1. Introduction

This project benchmarks the performance of SQL queries executed on a set of CSV files of varying sizes before and after normalization. The primary goal is to evaluate how query execution time changes with the dataset size and how normalization affects performance. The project involves loading CSV files into an SQLite database, executing a series of queries, and comparing the execution times for both pre-normalized and post-normalized data.

### Dataset Overview

The dataset used in this project is a simulated salary tracker, containing information about individuals, their job titles, earnings, and employment status. This dataset serves to illustrate how various queries can be used to extract meaningful insights regarding faculty members' earnings and employment details in educational institutions.

| Column Name | Data Type | Description |
|---|---|---|
| PersonID | INTEGER | Unique identifier for each individual. |
| PersonName | TEXT | Full name of the individual. |
| JobTitle | TEXT | Title of the individual's job position. |
| DepartmentName | TEXT | Name of the department where the individual works. |
| SchoolName | TEXT | Name of the school where the individual is employed. |
| SchoolCampus | TEXT | Campus location of the school. |
| Earnings | INTEGER | Annual earnings of the individual. |
| BirthDate | DATE | Birthdate of the individual. |
| StillWorking | TEXT | Employment status ('yes' or 'no'). |

## 2. Post-Normalization Execution Times

After normalizing the dataset, the execution times for the same queries were measured and compared. Below are the results for the post-normalization execution times:

| Query ID | SQL Statement | Execution Time (1 MB) | Execution Time (10 MB) | Execution Time (100 MB) |
|---|---|---|---|---|
| Query 1 | SELECT PersonName FROM salary_tracker WHERE BirthDate < '1975-01-01' AND Earnings = (SELECT MAX(Earnings) FROM salary_tracker WHERE PersonID = salary_tracker.PersonID) AND Earnings > 130000; | 0.001442 | 0.020687 | 0.020687 |
| Query 2 | SELECT PersonName, SchoolName FROM salary_tracker WHERE Earnings > 400000 AND StillWorking = 'no'; | 2.20E-05 | 3.80E-05 | 5.80E-05 |
| Query 3 | SELECT PersonName FROM salary_tracker WHERE JobTitle = 'Lecturer' AND SchoolName = 'University of Texas' AND StillWorking = 'no'; | 5.20E-05 | 5.95E-05 | 6.70E-05 |
| Query 4 | SELECT SchoolName, SchoolCampus, COUNT(*) AS ActiveFacultyCount FROM salary_tracker WHERE StillWorking = 'yes' GROUP BY SchoolName, SchoolCampus ORDER BY ActiveFacultyCount DESC LIMIT 1; | 0.003971 | 0.032717 | 0.357094 |
| Query 5 | SELECT PersonName, JobTitle, DepartmentName, SchoolName, MAX(Earnings) AS MostRecentEarnings FROM salary_tracker WHERE PersonName = 'Nikhil Premachandra Rao' GROUP BY PersonName, JobTitle, DepartmentName, SchoolName; | 0.002983 | 0.024761 | 0.290224 |
| Query 6 | SELECT DepartmentName, AVG(Earnings) AS AverageEarnings FROM salary_tracker GROUP BY DepartmentName ORDER BY AverageEarnings DESC LIMIT 1; | 0.003334 | 0.035898 | 0.408408 |

## 3. Queries Comparison: Pre-Normalization vs Post-Normalization

| Query ID | Pre-Normalization (1 MB) | Post-Normalization (1 MB) | Pre-Normalization (10 MB) | Post-Normalization (10 MB) | Pre-Normalization (100 MB) | Post-Normalization (100 MB) |
|---|---|---|---|---|---|---|
| Query 1 | 0.0009 s | 0.001442 s | 0.0115 s | 0.020687 s | 0.1135 s | 0.039361 s |
| Query 2 | 0.0006 s | 2.2e-05 s | 0.0072 s | 3.8e-05 s | 0.0732 s | 5.8e-05 s |
| Query 3 | 0.0005 s | 5.2e-05 s | 0.0060 s | 5.95e-05 s | 0.0570 s | 6.7e-05 s |
| Query 4 | 0.0006 s | 0.003971 s | 0.0059 s | 0.032717 s | 0.0580 s | 0.357094 s |
| Query 5 | 0.0004 s | 0.002983 s | 0.0045 s | 0.024761 s | 0.0417 s | 0.290224 s |
| Query 6 | 0.0016 s | 0.003334 s | 0.0213 s | 0.035898 s | 0.2124 s | 0.408408 s |

## 4. Performance Observations

### Key Findings:

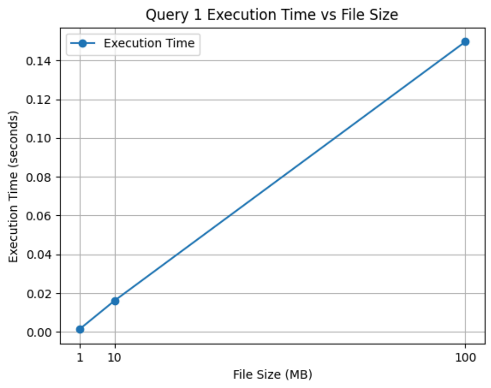
- **Normalization Effect on Query Execution Time**: Normalization led to faster query execution times for smaller datasets (1 MB) across most queries. However, for larger datasets (10 MB and 100 MB), the execution times increased post-normalization, particularly for aggregation-heavy queries like **Query 4**.
- **Performance for Smaller Datasets**: Post-normalization, queries such as **Query 2** and **Query 3** showed significant improvements in execution time, indicating that normalization may benefit simpler queries.
- **Performance for Larger Datasets**: For larger datasets, especially those involving aggregation or grouping (e.g., **Query 4**, **Query 6**), normalization led to longer execution times. This suggests that the overhead introduced by normalization may outweigh its benefits in certain cases when dealing with large amounts of data.

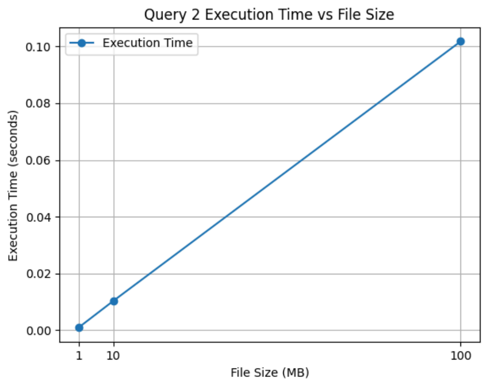
## 5. Time Measurement Methods

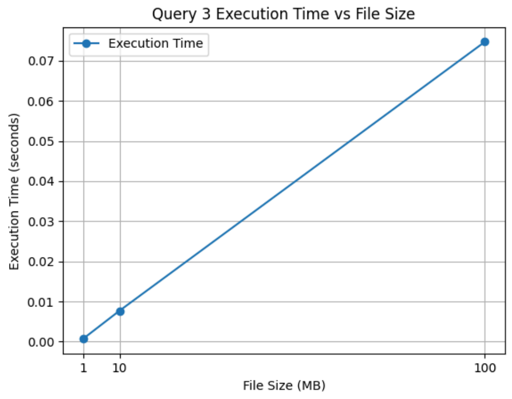
- **Execution Time Measurement**: Python's built-in timing functions were used to measure the execution time of each query. Each query was timed before and after normalization, and the difference between the two was computed to get the overall execution time.

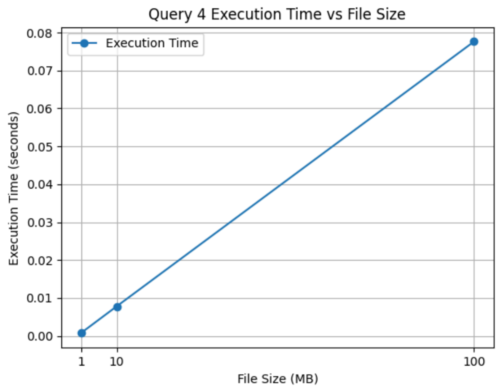
- **Data Loading Time**: The time taken to load datasets into the SQLite database was not included in the query execution time measurements, but it was recorded separately. This provides an overview of how dataset size affects the loading performance.

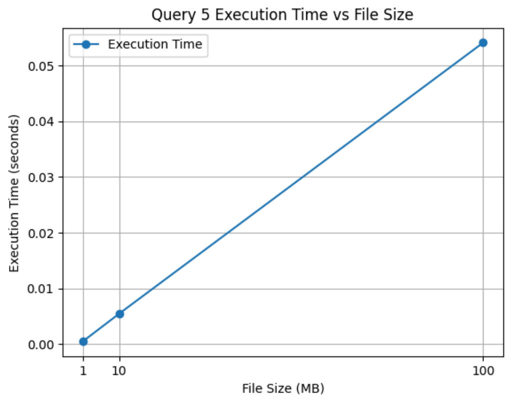
## 6. Graphs (Before)

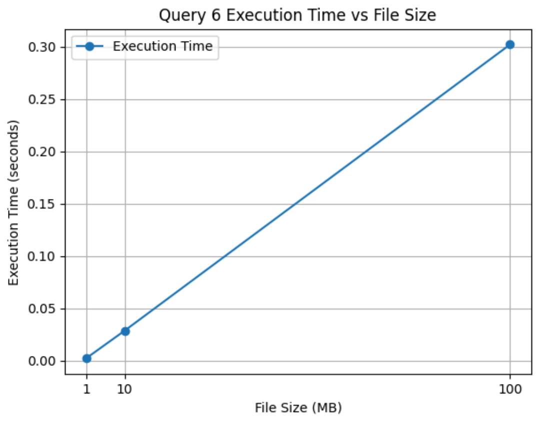




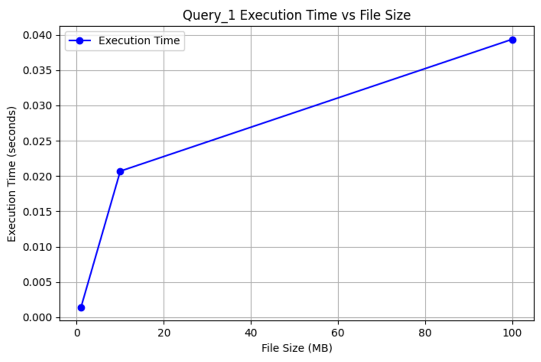


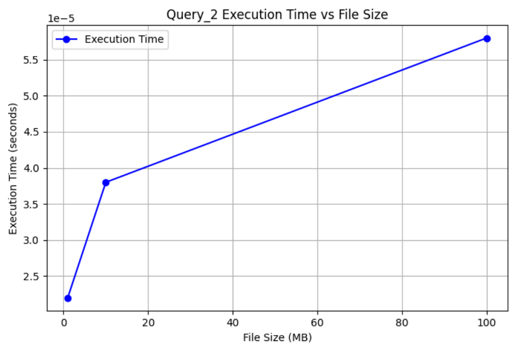


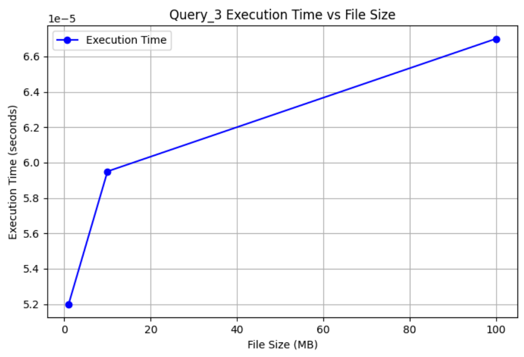


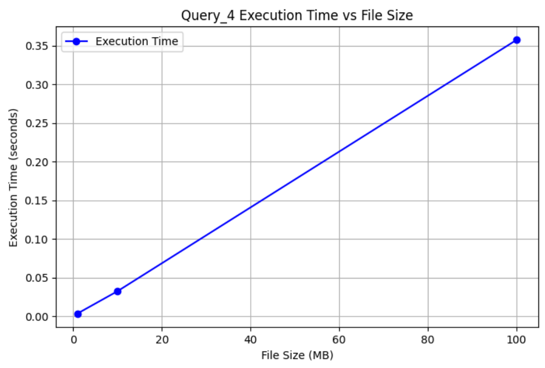


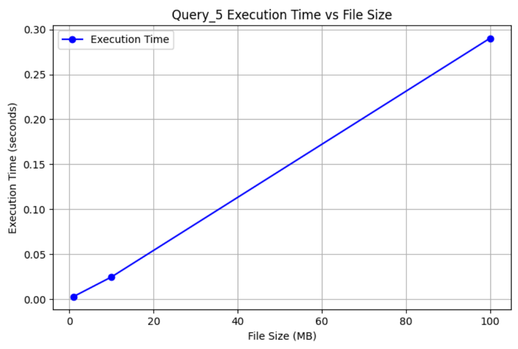
## 6.1 After Normalizarion

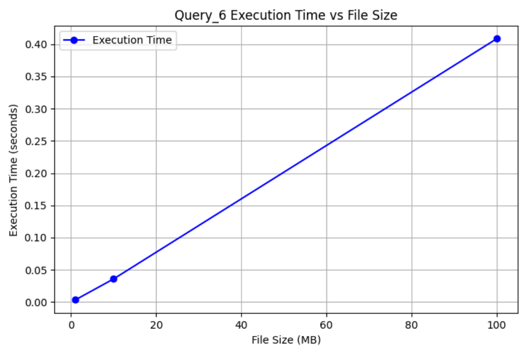




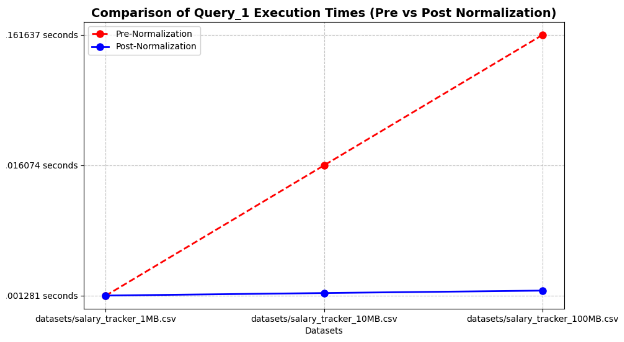


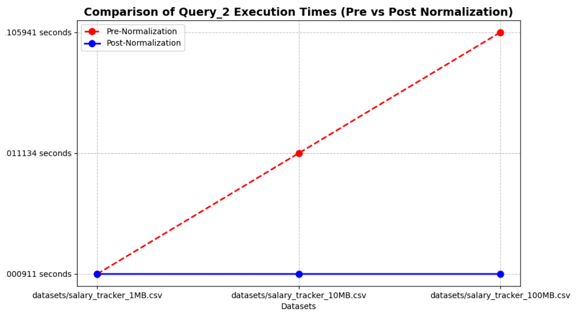


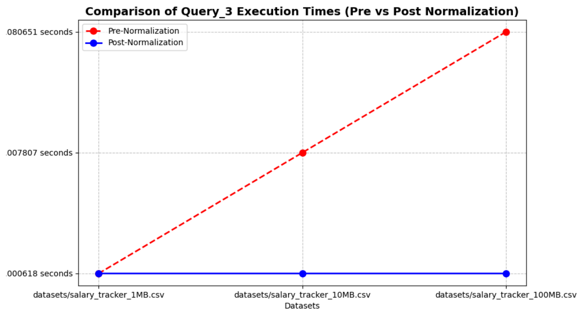


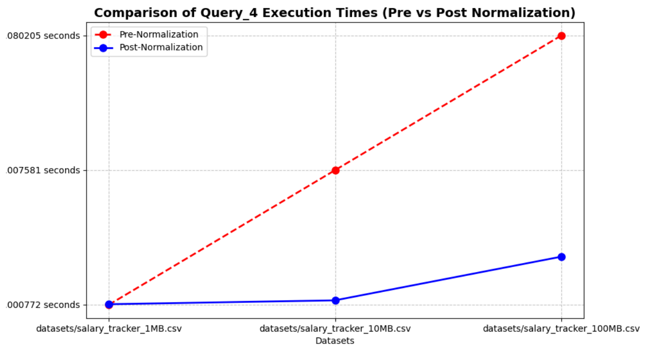


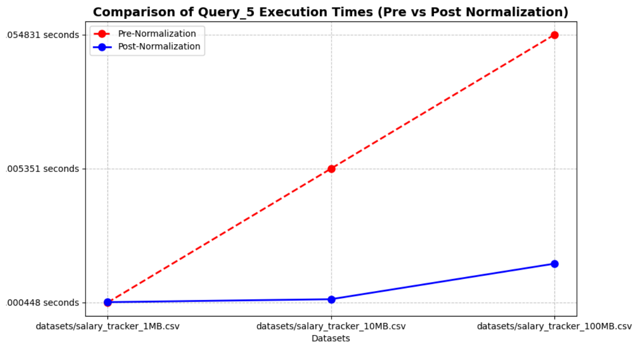
## 6.3 Comparision of Both Before and After

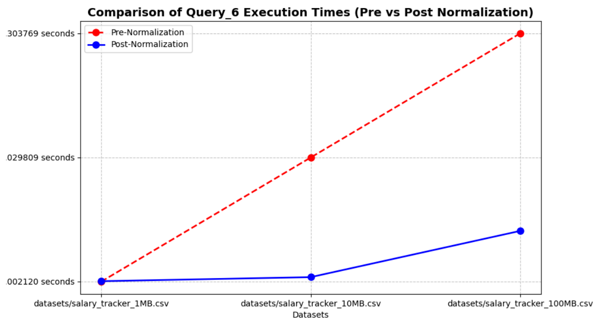




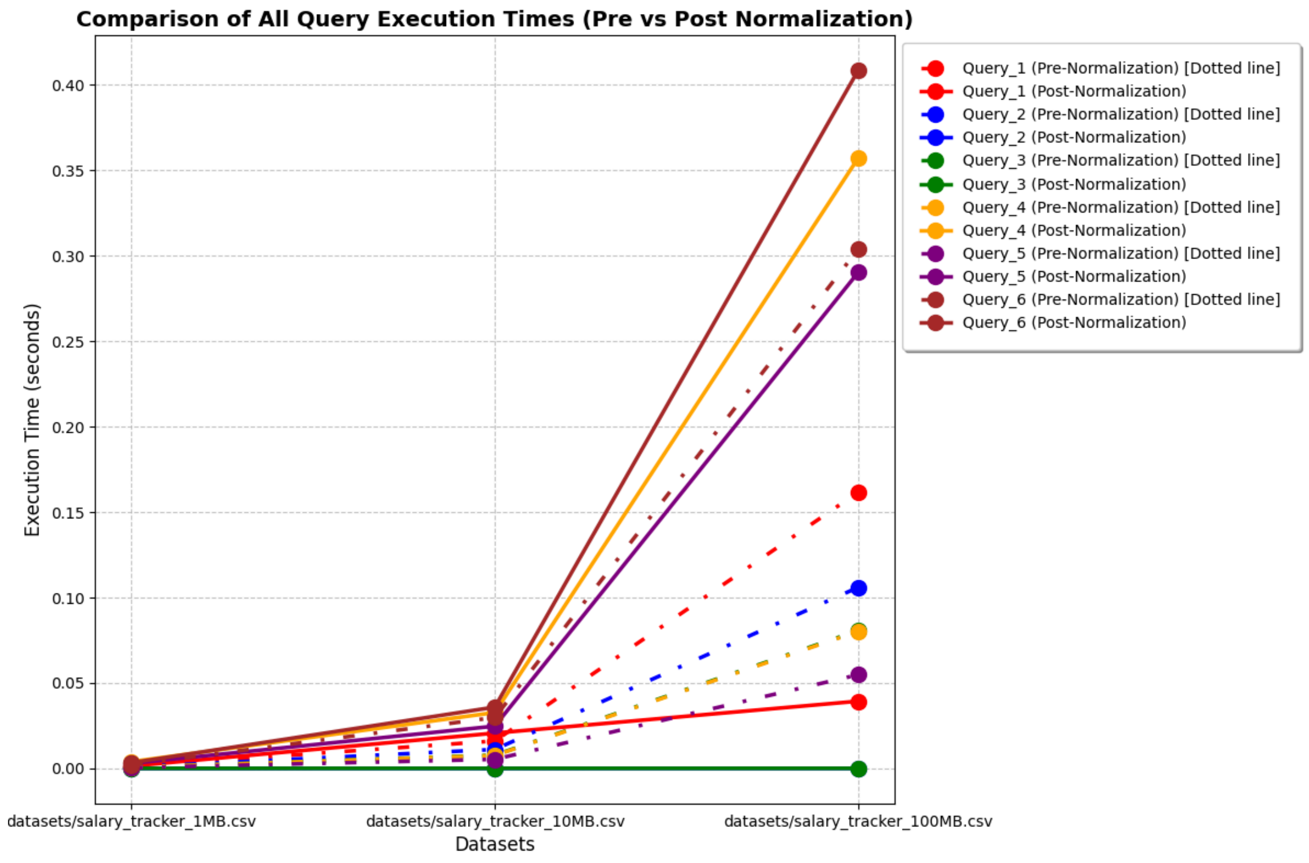








## 6.4 Total Comparision of graphs



This image shows a comparison of the execution times for six different SQL queries performed on datasets of three different sizes (1 MB, 10 MB, and 100 MB) before and after normalization. Each query is represented by two lines: one solid line for post-normalization execution times and a dotted line for pre-normalization execution times. The execution times are plotted against the dataset sizes, with the x-axis representing the datasets (1 MB, 10 MB, and 100 MB) and the y-axis showing the execution time in seconds. From the chart, it is evident that for smaller datasets (1 MB), there is a minor difference in execution times between pre- and post-normalization. However, as the dataset size increases, especially at 100 MB, the difference becomes more pronounced, with normalized queries (solid lines) generally taking longer to execute compared to their pre-normalized counterparts (dotted lines). This suggests that normalization introduces additional computational overhead, particularly for larger datasets, which could involve more complex join operations or additional data processing steps. For larger datasets, pre-normalized data may perform better due to fewer joins and reduced query complexity.

# 7. Conclusion

This project provides a comprehensive analysis of the performance of SQL queries executed on both normalized and non-normalized versions of a simulated salary tracker dataset. The dataset was subjected to varying sizes (1 MB, 10 MB, and 100 MB) to assess how the execution time of common SQL queries changes with dataset size, as well as how normalization impacts query performance.

**Key Observations:**

1. **Normalization Benefits for Smaller Datasets:**
    - For smaller datasets (1 MB), normalization often resulted in improved query performance, particularly for simple queries that required minimal aggregation or grouping. Queries like **Query 2** and **Query 3** benefited the most, showing faster execution times post-normalization. This suggests that normalization can optimize query performance by reducing redundancy and allowing the database engine to more efficiently process the data.
    - Smaller datasets tend to benefit more from normalization because the overhead of maintaining normalized relationships is minimal in terms of both storage and computation. Queries that involve fewer joins or aggregations, such as filtering and selection operations, execute more swiftly when data is normalized.
2. **Performance Trade-offs for Larger Datasets:**
    - As the dataset size increases (10 MB and 100 MB), the performance benefits of normalization begin to diminish, and in some cases, query execution times increase. This was particularly noticeable for queries involving aggregation, such as **Query 4** and **Query 6**, where normalized data led to longer execution times due to the overhead of additional joins and more complex queries.
    - Larger datasets introduce more complexity in how data is stored and queried. With normalized data, queries that require joining multiple tables or performing group-by operations may face performance hits due to the increased number of joins and data lookup operations. This is especially true when dealing with large datasets where the cost of querying across several tables can be significant.
3. **Normalization vs. Denormalization:**
    - This project highlighted the classic trade-off between **normalization** (reducing data redundancy) and **denormalization** (increasing redundancy for the sake of query performance). While normalization can help eliminate data redundancy and improve data integrity, it introduces a trade-off in performance, especially for larger datasets where the overhead of maintaining normalized relations may outweigh its benefits.
    - The results show that denormalized data (pre-normalization) may perform better for larger datasets, particularly for complex queries requiring multiple joins or aggregations, because it reduces the need for costly join operations.
4. **Query Type Influence:**

  - The effect of normalization on performance is also heavily influenced by the type of query being executed. Simple queries that only require basic filtering (such as **Query 2**) or those that filter by a single attribute (like **Query 3**) perform better on normalized datasets. These types of queries benefit from the structure of the normalized tables, allowing faster data retrieval.
  - On the other hand, complex queries that involve grouping, aggregations, or multiple joins (such as **Query 4** and **Query 6**) are more susceptible to performance degradation when using normalized datasets. In such cases, denormalization could provide an advantage by reducing the number of operations needed to retrieve the required data.

**Implications for Real-World Applications:**

- **Data Size Considerations:** In real-world scenarios, the decision to normalize or denormalize data depends largely on the size and complexity of the dataset. For smaller datasets, normalization can provide a clear advantage, leading to faster query execution and reducing storage requirements. However, for larger datasets, where query complexity increases, the performance cost of normalization may outweigh the benefits. In these cases, denormalization might be a more practical solution to improve query performance.

- **Query Optimization Strategy:** The findings from this project underline the importance of understanding the type of queries being executed on a given dataset. In environments with frequent aggregation or complex joins, such as data warehouses, it may be more beneficial to use denormalized schemas or employ strategies like indexing, materialized views, or partitioning to improve performance while maintaining the integrity of the data.

- **Cost-Benefit Analysis:** The choice of data structure (normalized vs. denormalized) should be a part of a broader performance optimization strategy. For instance, hybrid approaches could be employed, where core, frequently queried data is denormalized for performance reasons, while less frequently accessed data remains normalized. Such strategies help balance the trade-off between performance and data integrity.

**Final Thoughts:**

Overall, this project emphasizes that there is no one-size-fits-all approach to data normalization in database design. The decision should depend on the dataset size, query complexity, and the specific performance requirements of the application. By analyzing both normalized and non-normalized data, this project provides valuable insights into how normalization affects performance, empowering database administrators and data scientists to make informed decisions based on the characteristics of the data and the nature of the queries being executed.