**CIS 552: Database Design – Fall 2025  
 Database Design Project Specification**

**Abstract**

This project investigates the performance impact of database normalization on query execution time when working with large CSV datasets in MySQL. CSV files of varying sizes (1 MB, 10 MB, 100 MB, and 1000 MB) were imported into MySQL and queried using a non-normalized schema. Execution times for multiple query types, including selection, filtering, joins, and aggregation, were measured and analyzed. The dataset was then normalized to a relational schema following Second Normal Form (2NF) and Third Normal Form (3NF) principles. Queries were rewritten to operate on the normalized schema, and performance measurements were collected again for comparison. The results demonstrate the trade-off between reduced redundancy and increased join complexity, highlighting how normalization affects query performance on large datasets.

**Introduction**

With the increasing availability of large-scale structured datasets, efficient data storage and query execution have become critical concerns in database systems. CSV files are commonly used for data exchange due to their simplicity and portability; however, querying large CSV files directly is inefficient and impractical. As a result, relational database systems such as MySQL are often used to store and manage such data.

Database normalization is a fundamental design principle that reduces redundancy and improves data integrity by organizing data into multiple related tables. While normalization improves maintainability and consistency, it can introduce additional joins that may negatively impact query performance. This project aims to evaluate the performance implications of normalization by comparing query execution times on a non-normalized schema and a normalized schema using MySQL.

The primary objectives of this project are:

1. To measure query execution time on large CSV datasets imported into MySQL.
2. To design and implement a normalized database schema (up to 3NF).
3. To compare the performance of queries executed on non-normalized and normalized schemas.
4. To analyze and visualize the performance trends using graphs.

Data set description

The dataset used in this project consists of CSV files representing structured information related to individuals, schools, departments, and employment details. The same dataset was provided in four different sizes to evaluate scalability and performance trends:

* **1 MB**
* **10 MB**
* **100 MB**

**3. Database Schema Design**

**3.1 Non-Normalized Schema**

In the initial phase of the project, the CSV data was imported directly into MySQL as a single table. This table contained all attributes from the CSV file, resulting in significant data redundancy. Attributes related to schools, departments, and jobs were repeatedly stored for multiple records, violating normalization principles.

This schema does not satisfy Second Normal Form (2NF) or Third Normal Form (3NF) because non-key attributes depend on other non-key attributes, leading to transitive dependencies and redundancy.

3.2 Normalized schema

To address redundancy and dependency issues, the dataset was normalized into multiple relational tables. The normalization process involved:

* Separating entity-specific attributes into independent tables.
* Assigning primary keys to each table.
* Establishing foreign key relationships to preserve referential integrity.

The resulting schema includes tables such as:

* **Person**
* **School**
* **Department**
* **Job**
* **Employment**

This design satisfies Second Normal Form by removing partial dependencies and Third Normal Form by eliminating transitive dependencies. As a result, data redundancy is minimized, and consistency is improved.

**4. Methodology**

This section explains the systematic approach followed to load the CSV data into MySQL, execute queries, measure performance, and compare results between non-normalized and normalized database schemas.

**4.1 Data Preparation and CSV Handling**

The dataset was provided in CSV format with four different file sizes: 1 MB, 10 MB and 100 MB.Each file contained the same schema and attributes, with larger files created by replicating records to increase data volume while maintaining structural consistency.

Before loading the data into the database, the CSV files were inspected to ensure consistent column ordering and data types. This step ensured that the same loading and querying logic could be applied across all dataset sizes without modification.

**4.2 Database Environment Setup**

A MySQL database was used as the embedded database system for this project. A dedicated database was created to store the CSV data and experimental results. The database environment was accessed programmatically using Python, which provided automation for data loading, query execution, and performance measurement.

**4.3 Non-Normalized Schema Design and Data Loading**

In the initial phase, the CSV data was imported into MySQL using a single non-normalized table. The table schema closely matched the structure of the CSV file, with each row representing a complete record containing person, school, department, and job information.

A Python script was developed to:

1. Create the non-normalized table in MySQL.
2. Read the CSV file line by line.
3. Insert records into the table using bulk insertion techniques.

This approach allowed direct querying of the CSV data without applying any normalization, serving as the baseline for performance comparison.

**4.4 SQL Query Design for the Non-Normalized Schema**

A set of predefined SQL queries was written and executed on the non-normalized table. These queries were designed to evaluate different aspects of query performance and included:

* Selection queries using filtering conditions (WHERE).
* Aggregation queries using functions such as COUNT, SUM, and AVG.
* Grouping queries using GROUP BY.

These queries did not require join operations because all attributes were stored within a single table.

**4.5 Query Execution and Time Measurement (Non-Normalized Schema)**

Query execution was automated using Python. For each query, a high-resolution timer was started immediately before executing the SQL statement and stopped after the complete result set was retrieved. This ensured that only the execution time of the query was measured, excluding database connection and data loading overhead.

Each query was executed multiple times for each dataset size (1 MB, 10 MB, and 100 MB). The average execution time was recorded to reduce the impact of system variability and caching effects.

**4.6 Database Normalization and Schema Redesign**

To improve data organization and reduce redundancy, the non-normalized schema was transformed into a normalized relational design following Second Normal Form (2NF) and Third Normal Form (3NF) principles.

The normalization process involved:

* Identifying distinct entities such as person, school, department, and job.
* Separating these entities into individual tables.
* Assigning primary keys to uniquely identify records.
* Defining foreign key relationships to maintain referential integrity.

This redesign eliminated transitive and partial dependencies present in the non-normalized schema.

**4.7 Data Loading into the Normalized Schema**

A modified data-loading program was developed to populate the normalized database schema. The CSV file was parsed, and data was inserted into multiple tables based on entity type. Parent tables were populated before child tables to ensure that foreign key constraints were satisfied.

**4.8 Query Execution and Time Measurement (Normalized Schema)**

The normalized queries were executed using the same Python-based timing mechanism used for the non-normalized schema. Execution time was measured by capturing the duration between query submission and result retrieval.

Each normalized query was executed multiple times on the 100 MB dataset, and the average execution time was recorded to ensure consistency with the non-normalized measurements.

## **4.9 Performance Data Collection and Visualization**

All execution times were stored in structured CSV files containing the query identifier, schema type, dataset size, and execution time. These records were used to generate performance graphs for each query.

Each graph displays execution time on the y-axis and dataset size on the x-axis, allowing clear visualization of performance trends and direct comparison between non-normalized and normalized schemas.

## **4.10 Summary of Methodology**

In summary, this project followed a systematic methodology involving CSV data ingestion, schema design, SQL query execution, precise time measurement, and performance visualization. By applying the same queries to both non-normalized and normalized schemas and measuring execution time under controlled conditions, the methodology enabled a meaningful comparison of query performance and the impact of database normalization.

**5. SQL Queries**

This section documents the SQL queries used for performance evaluation. For each query, two versions were implemented: one operating on the non-normalized schema and another rewritten for the normalized schema. Both versions of the queries are logically equivalent to ensure a fair comparison of performance.

Query Description 1:  
List the unique PersonNames along with their BirthDates.

For 1mb

A screen shot of a computer

AI-generated content may be incorrect.

Normalized

A screen shot of a computer code

AI-generated content may be incorrect.

This query measures how long it takes to execute a SELECT DISTINCT operation on the employment\_raw table. First, the current timestamp with microsecond precision is stored in the variable @start\_time. Then, the query retrieves unique combinations of PersonName and BirthDate, eliminating duplicate records that may exist due to multiple jobs or years per person. Finally, the time difference between the start and end of execution is calculated in microseconds, converted to seconds, rounded to six decimal places, and returned as Q1\_execution\_time\_seconds.

Query Description 2:

List the unique PersonNames of people who are currently actively working, along with the School Name and School Campus they are working at.

A computer screen shot of a computer code

AI-generated content may be incorrect.

Normalized

A screen shot of a computer code

AI-generated content may be incorrect.

This query measures the execution time required to retrieve unique records of people who are currently working. It first stores the current timestamp with microsecond precision in the variable @start\_time. The main query selects distinct combinations of PersonName, SchoolName, and SchoolCampus from the employment\_raw table where the StillWorking status is 'yes', filtering only active employees. Finally, the elapsed time is calculated in seconds by taking the difference between the start time and the current time, rounding the result to six decimal places, and returning it as Q2\_execution\_time\_seconds.

Query Description 3:

Show the names and job titles of all Assistant Professors (job title) currently working at University of Massachusetts Dartmouth.

A computer screen shot of a computer code

AI-generated content may be incorrect.

Normalized

A screenshot of a computer code

AI-generated content may be incorrect.

This query measures the time required to find currently working individuals with a specific job role at a specific institution and campus. It first records the current timestamp with microsecond precision in @start\_time. The main query selects distinct combinations of PersonName and JobTitle from the employment\_raw table, filtering for records where the person is still working, the job title is Assistant Professor, and the school is the University of Massachusetts at the Dartmouth campus. Finally, the elapsed execution time is calculated in seconds, rounded to six decimal places, and returned as Q3\_execution\_time\_seconds.

Query Description 4:

Show how many people are working at each Campus currently (most recent year, actively working).

A computer screen shot of a program

AI-generated content may be incorrect.

Normalized

A computer screen shot of a program

AI-generated content may be incorrect.

This query measures the execution time required to count the number of currently working individuals at each school campus for the most recent earnings year. It first records the current timestamp with microsecond precision in @start\_time. The main query filters records where StillWorking is 'yes' and the EarningsYear equals the maximum year present in the table, then groups the results by SchoolCampus and counts distinct PersonID values for each campus. Finally, the elapsed execution time is calculated in seconds, rounded to six decimal places, and returned as Q4\_execution\_time\_seconds.

Query Description 5:

Show the total earnings of each unique person along with their person names across all years.

A screenshot of a computer code

AI-generated content may be incorrect.

Normalized

A screenshot of a computer code

AI-generated content may be incorrect.

This query measures the execution time needed to calculate total earnings for each individual. It first stores the current timestamp with microsecond precision in @start\_time. The main query groups records by PersonID and PersonName and computes the sum of earnings for each person using the SUM(Earnings) aggregate function. Finally, the elapsed execution time is calculated in seconds, rounded to six decimal places, and returned as Q5\_execution\_time\_seconds.

**5.2 normalized-schema**

After completing the performance analysis on the non-normalized dataset, the database schema was redesigned using normalization principles to reduce data redundancy and improve consistency. The original employment\_raw table was decomposed into multiple related tables, including person\_norm, school\_norm, department\_norm, and job\_norm, each storing entity-specific information. An employment\_norm fact table was created to store employment-related attributes such as earnings, earnings year, and employment status while referencing the corresponding dimension tables through foreign keys.

The normalized tables were populated using data from the original raw table. Unique records were inserted into the dimension tables using SELECT DISTINCT along with INSERT IGNORE to prevent duplicate entries. After populating the dimension tables, all employment records were inserted into the fact table while maintaining referential integrity.

Finally, validation checks were performed by comparing record counts between the raw and normalized tables to ensure that the data migration was accurate and complete.

A screenshot of a computer program

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

## 5.3 **Python Implementation for Data Loading and Performance Measurement**

Python was used to automate the process of loading CSV data into MySQL, executing SQL queries, and measuring query execution time. Separate scripts were implemented for handling the non-normalized and normalized database schemas to ensure clarity and reproducibility.

The script load\_raw.py was used to import CSV files directly into a single non-normalized MySQL table. For the normalized schema, load\_normalized.py was implemented to parse the same CSV file and populate multiple related tables while maintaining primary and foreign key relationships.

Query execution and performance measurement were handled using a dedicated benchmarking script (benchmark.py). This script executes each SQL query programmatically and records execution time using high-resolution timing functions. The timer is started immediately before executing a query and stopped after fetching the complete result set, ensuring that only query execution time is measured.

A screenshot of a computer program

AI-generated content may be incorrect.

## **6. Results**

This section presents the execution time measurements collected for the SQL queries executed on both the non-normalized and normalized database schemas. Query execution times were recorded programmatically and exported to an Excel spreadsheet, which served as the source for generating all performance graphs.

**6.1 Execution Time Measurements**

A screenshot of a calculator

AI-generated content may be incorrect.

The execution times obtained from the Excel sheet for the selected query across different dataset sizes. The non-normalized schema was evaluated using 1 MB, 10 MB, and 100 MB datasets, while the normalized schema was evaluated using the 100 MB dataset, as specified in the project requirements.

### **6.2 Performance Graph Analysis**

Figure shows the relationship between dataset size and query execution time for both schema designs. The x-axis represents the data size in megabytes, while the y-axis represents execution time in seconds.

A graph with a line pointing up

AI-generated content may be incorrect.

From the graph, it can be observed that query execution time increases as the dataset size grows for the non-normalized schema. This trend is expected, as larger datasets require scanning and processing more rows. The increase from 1 MB to 10 MB is more pronounced, while the growth from 10 MB to 100 MB is more gradual.

The normalized schema exhibits a higher execution time at the 100 MB data size compared to the non-normalized schema. This increase is primarily due to the additional JOIN operations required to combine data from multiple tables in the normalized design.

**7 conclusion**

This project evaluated the performance of SQL query execution on large CSV datasets using MySQL by comparing a non-normalized schema with a normalized schema designed up to Third Normal Form (3NF). CSV files of varying sizes were successfully loaded into MySQL, and execution times were measured systematically using Python-based automation. The results show that query execution time increases as the dataset size grows, reflecting the higher cost of scanning and processing larger volumes of data.

Overall, queries executed on the non-normalized schema generally performed faster because they operate on a single table and avoid join overhead. In contrast, the normalized schema required additional join operations, leading to increased execution time for some queries, particularly at larger data sizes. Despite this performance cost, normalization provided significant benefits in terms of reduced data redundancy, improved data consistency, and better long-term maintainability. These findings highlight the trade-off between performance and schema design quality and emphasize the importance of choosing a database design that aligns with both performance needs and data management requirements.