Second Submission
Assignment 4. Monte Carlo Methods---Benchmark Design and Analysis
Ryan Fallon

Abstract
This study evaluates the performance of Python and Go in executing SQL queries on a PostgreSQL database using Monte Carlo simulations. The benchmarks test query performance under varying loads (1,000, 10,000, and 100,000 iterations) for operations such as aggregations, joins, and filtering. Benchmarks were conducted on a MacBook Pro (M1 Max, 64GB RAM), ensuring consistent conditions. Interestingly, Python outperformed Go in all scenarios, prompting further analysis. After investigating, I believe the performance gap stems from differences in connection handling, with Python benefiting from efficient connection reuse and Go encountering overhead due to its lack of native connection pooling.

Introduction
Database query performance is critical in data science and engineering workflows. This study benchmarks Python (psycopg2) and Go (lib/pq) for executing SQL queries on PostgreSQL, a popular relational database. Python, known for its extensive libraries and ease of use, competes against Go, which excels in concurrency and efficiency. I aim to assess which language is better suited for repetitive query tasks in a controlled environment. All code and output results can be found at this GitHub repository.

Literature Review
Previous benchmarks, have explored query optimization in PostgreSQL using Python and Go. A noteworthy example is a Medium article, which tested Python and Go in handling JWT authentication and MySQL queries—an intensive, real-world scenario. In this study, Go significantly outperformed Python by a margin of 265%, driven largely by its compiled nature and efficient handling of CPU-intensive tasks like JWT verification. The experiment involved preloading 100K JWTs and querying a MySQL database with 100K user records, simulating high-concurrency, real-world workloads. While Python used FastAPI with libraries such as jwt and mysql-connector, Go utilized the Gin web framework and golang-jwt for similar functionality.

The stark difference in performance between Go and Python was attributed to Go's compiled execution model and its efficient concurrency handling through goroutines, enabling it to handle more requests per second with lower latency. Despite Python's flexibility and robust library ecosystem, this benchmark highlighted Go's ability to outperform in highly parallel and computationally demanding tasks.

This comparison informed the current study by reinforcing the need to analyze connection handling and concurrency for SQL query execution. While the Monte Carlo simulation in this research focuses on PostgreSQL queries rather than JWTs, the insights from the

Medium study emphasize Go's potential advantage in performance-critical environments when properly optimized.

Methods

Setup

I used the saleco PostgreSQL database from Northwetsern's SQL Learning studio containing customer, invoice, and product tables. Key queries included counting invoices, listing customers, and finding the most expensive product. Python's psycopg2 and Go's lib/pq libraries were used to execute the queries. All benchmarks were run on a MacBook Pro (M1 Max, 64GB RAM).

Simulation

Each query was executed 1,000, 10,000, and 100,000 times using a Monte Carlo loop to simulate high-frequency workloads. Python reused a single database connection across iterations, while Go utilized concurrency with separate goroutines.

Queries

Sample queries included (see GitHub repository for all queries/results):

1. Count Invoices: SELECT COUNT(*) FROM INVOICE;
2. List Customers: SELECT cus_code, CONCAT(cus_fname, ' ', cus_lname) FROM customer;
3. Most Expensive Product: SELECT p_code, p_price FROM product ORDER BY p_price DESC LIMIT 1;

Results

Execution Times

Python consistently outperformed Go across all benchmarks:

- At 1,000 iterations, Python queries averaged 0.0003–0.0007 seconds, while Go averaged 0.0022–0.0025 seconds.
- At 100,000 iterations, Python maintained averages of 0.002–0.009 seconds, while Go ranged from 0.021 to 0.025 seconds.

Observations

Upon investigating the surprising performance gap, I identified the key issue: connection handling. Python's psycopg2 reused a single connection across all iterations, minimizing overhead. In contrast, Go's lib/pq lacked native connection pooling, leading to inefficiencies as new connections were frequently established.

Discussion

Why Python Outperformed Go

1. Connection Reuse: Python's approach of reusing a single connection reduced the overhead associated with establishing connections.
2. Go's Concurrency Trade-offs: While Go's goroutines enabled concurrent query execution, the absence of a connection pool in lib/pq introduced significant delays.

Recommendations for Go
To mitigate Go's inefficiencies, a connection pooling library such as github.com/jackc/pgx should be used. This would allow connections to be reused across goroutines, significantly improving performance.

Conclusions
Python's superior performance highlights the efficiency of connection reuse, making it an excellent choice for data scientists performing repetitive database operations. Go, while powerful for concurrency, requires additional optimization, such as connection pooling, to match Python's performance for query-intensive workloads.

Recommendations
- For data scientists: Python is recommended for ease of use and efficient query execution.
- For engineering-focused teams: Go remains viable but requires careful setup, particularly for connection management.

Limitations and Future Work
This study focused solely on PostgreSQL and specific libraries (psycopg2 and lib/pq). Expanding the benchmarks to include other databases or Go libraries like pgx would provide a more comprehensive view. Additionally, exploring R as a third language could further enrich the analysis.