CS307 Principles of Database Systems - Spring 2024
Project 1 Report

Group Members:
- ZERHOUNI KHAL Jaouhara (12211456)
- HOK Layheng (12210736)

Lab Session: Tuesday (5-6)
Instructor: Dr. MA Yuxin

CONTENT

I. CONTRIBUTION:

Members	Tasks	Ratio
ZERHOUNI KHAL Jaouhara	Database design, E-R diagram, DDL queries, SQL queries and data accuracy check, Data import on Linux, Documentation (Report)	50%
HOK Layheng	Import scripts in three different methods, in three different volumes, in Java and Python, for PostgreSQL and MySQL, Data import on macOS and Windows	50%

II. TASK 1: E-R Diagram

The E-R diagram below was drawn using “draw.io” (https://app.diagrams.net/):

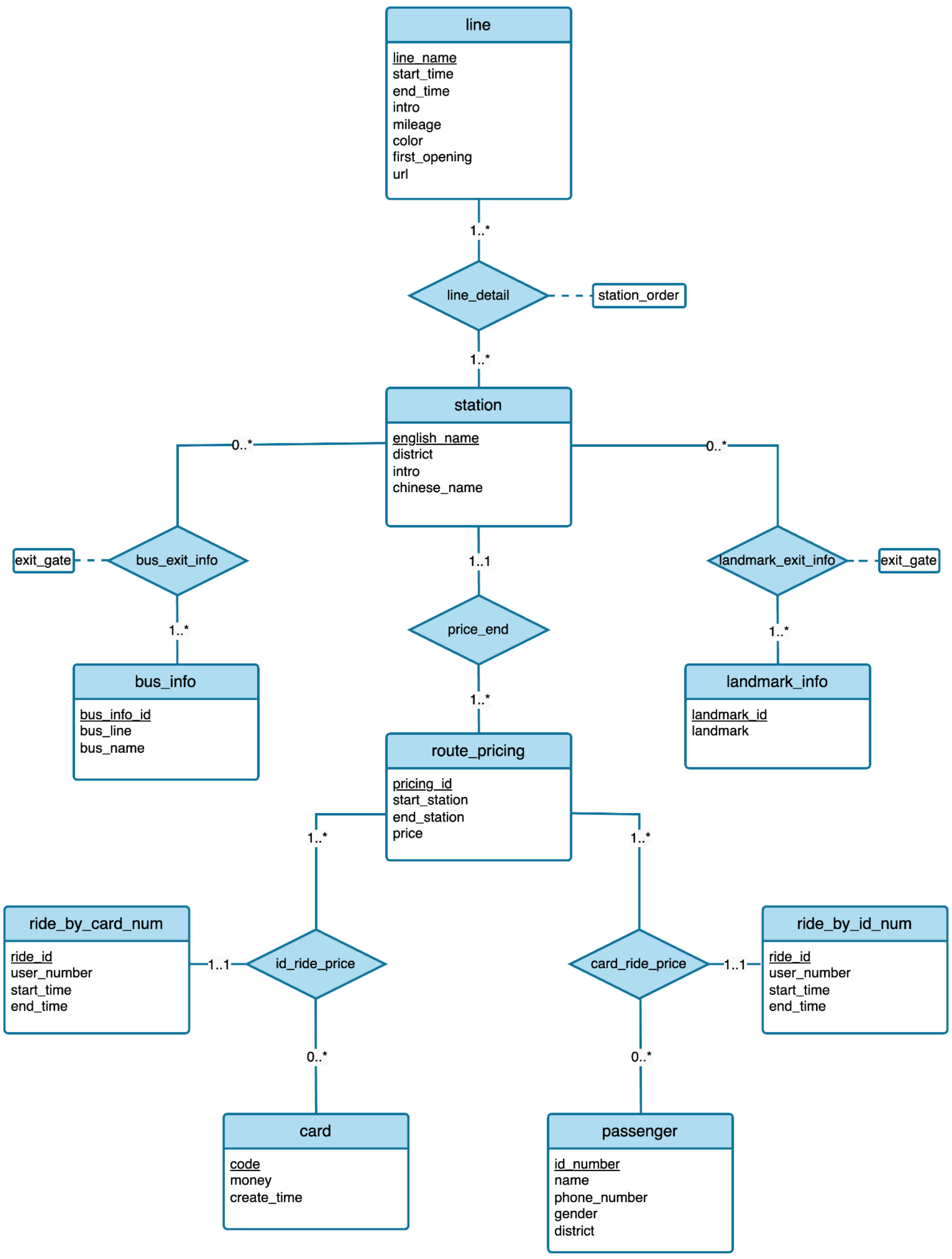
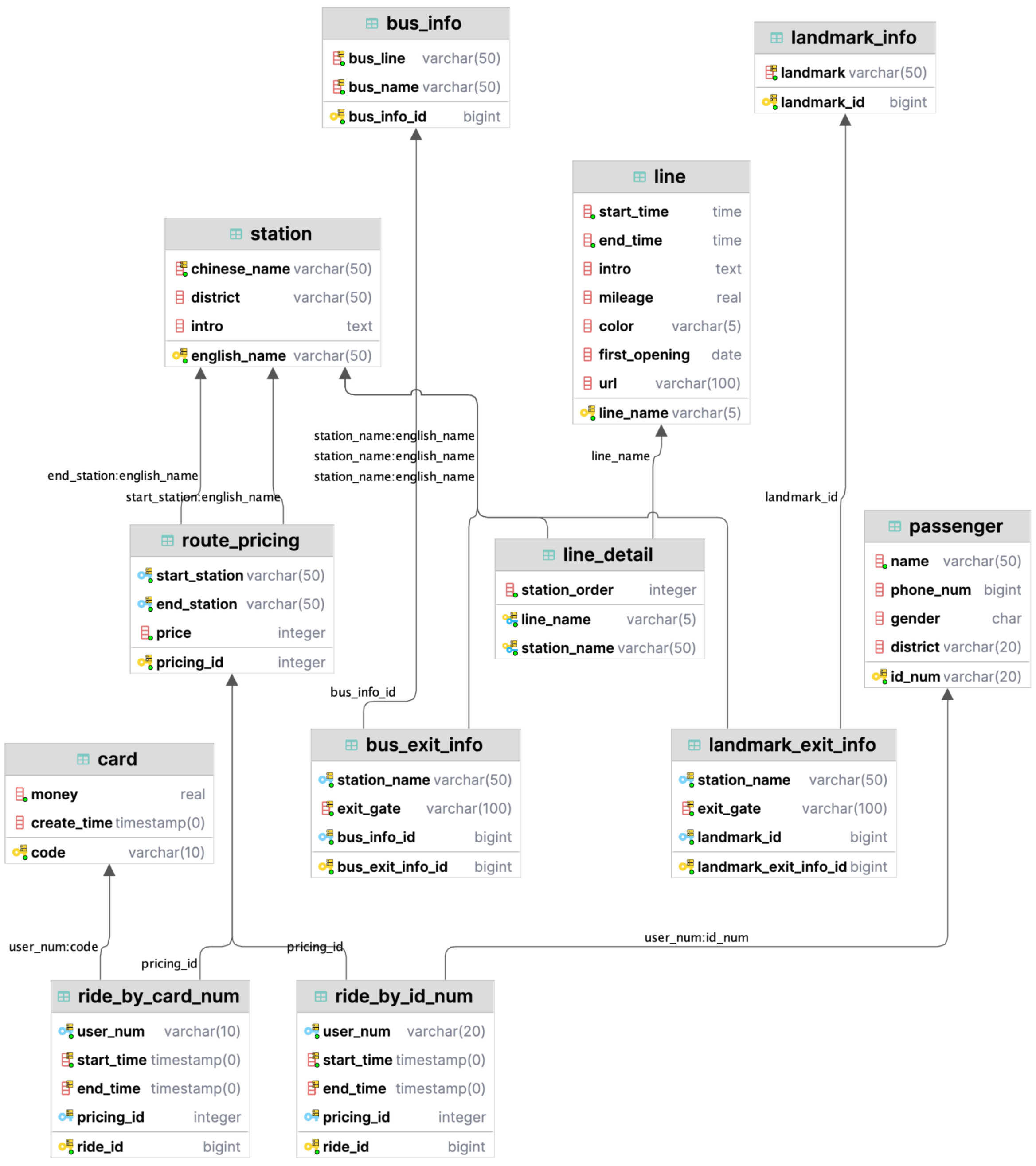


Figure 1: E-R diagram by draw.io

II. TASK 2: Relational Database Design

1. Snapshot of the E-R diagram generated by DataGrip



2. Design Description

Our main goal during the design procedure was to make it a simple, logical, and understandable design.
Here is a breakdown of the main components:

a. Tables for Basic Entities:

1. **station:** Records details about stations
   - *english_name:* Primary key of the table, refers to the English name of the station.
   - *chinese_name:* Name of the station in Chinese; must be *UNIQUE* and *NOT NULL*.
   - *district:* The district or area where the station is located.
   - *intro:* A brief description or introduction to the station.

2. **bus_info:** Stores information on buses
    - *bus_info_id:* A unique identifier for each bus entry, auto-incremented.
    - *bus_line:* The number or identifier of the bus line, *NOT NULL.*
    - *bus_name:* The name of the bus service, *NOT NULL.*
    - *(bus_line, bus_name) UNIQUE* combination.
3. **landmark_info:** Details landmarks
    - *landmark_id:* A unique identifier for each landmark, auto-incremented.
    - *landmark:* The name of the landmark or building, *NOT NULL* and *UNIQUE.*
4. **line:** Describes subway transit lines
    - *line_name:* A short identifier for the line, used as a *primary key.*
    - *start_time:* The time when services on this line begin each day, *NOT NULL.*
    - *end_time:* The time when services on this line end each day, *NOT NULL.*
    - *intro:* A brief description of the line.
    - *mileage:* Total distance covered by the line.
    - *color:* Color code or description for the line.
    - *first_opening:* Date when the line was first opened.
    - *url:* A URL for more information about the line.
5. **card:** Stores information about payment card
    - *code:* A unique identifier for the transit card, used as the *primary key*.
    - *money:* The current balance on the card, *NOT NULL.*
    - *create_time:* Timestamp of the card's creation.
6. **passenger:** Contains passenger details
    - *id_num:* A unique identifier for each passenger, used as the *primary key.*
    - *name:* The passenger's name, *NOT NULL.*
    - *phone_num:* Contact phone number.
    - *gender:* Gender of the passenger ("女" or "男").
    - *district:* The district where the passenger resides.
7. **route_pricing:** Manages pricing between start and end stations
    - *pricing_id:* A unique identifier for pricing between stations， auto-incremented.
    - *start_station:* Starting station name, *NOT NULL.*
    - *end_station:* Destination station name, *NOT NULL.*
    - *price:* Cost of the journey between the start and end stations, *NOT NULL.*
    - *(start_station, end_station)* as a *UNIQUE* combination.

b. Tables for Relationships:

1. **bus_exit_info:** Links bus information to specific station exits
    - *bus_exit_info_id:* Unique identifier for the bus info, auto-incremented.
    - *station_name:* References the English name of the station, *NOT NULL.*
    - *exit_gate:* Describes the exit of the station used to access the bus, *NOT NULL.*
    - *bus_info_id:* Links to the bus_info table as a foreign key, *NOT NULL.*
    - (station_name, exit_gate, bus_info_id) as a *UNIQUE* combination.
2. **landmark_exit_info:** Associates landmarks with station exit
    - *landmark_exit_info_id:* Unique identifier for the landmark info, auto-incremented.

- *station_name:* Connects to the station by its English name, *NOT NULL.*
- *exit_gate:* Specifies the station exit leading to the landmark, *NOT NULL.*
- *landmark_id:* Foreign key linking to the *landmark_info, NOT NULL.*
- *(station_name, exit_gate, landmark_id)* as a *UNIQUE* combination.

3. **line_detail:** Details which stations are on which lines and in what order
   - *line_name:* Ties back to the line table.
   - *station_name:* References station by its English name.
   - *station_order:* The order of the station on the line, *NOT NULL.*
   - *(line_name, station_name)* as a composite primary key.
4. **ride_by_id_num and ride_by_card_num:** Associates rides and pricing information
   - Both tables record trips taken by passengers, identified either by an ID number or a card code.
   - *ride_id:* Unique identifier for the trip, auto-incremented.
   - *user_num:* Foreign key linking to either a passenger ID or a card code, *NOT NULL.*
   - *start_time:* Timestamp when the trip started, *NOT NULL.*
   - *end_time:* Timestamp when the trip ended.
   - *pricing_id:* Links to the *route_pricing* for cost details.
   - (user_num, start_time, end_time, pricing_id) as a *UNIQUE* combination.

Based on the specifications stated in the requirements, the data is divided into smaller manageable parts respecting the principles of 3NF ensuring that the database is adaptable and flexible to further changes. Our design consists of 12 tables, each designed to fulfill specific roles within the transit system, from managing user and payment information to detailing the logistics of transportation routes and their associated costs.
The principle used in our implementation is mainly based on symmetry. For instance, the station table is connected to the bus_info table on the one hand, which holds the information regarding the available buses at the exits of each station, on the other hand, it is also connected to the corresponding landmarks and buildings in the same area. Another case includes the rides, which filters the rides based on the method used (card code or passenger's id_number). This schema not only facilitates complex queries and operations necessary for the system's functionality but also leaves room to accommodate future modifications or expansions as requirements evolve.

III. TASK 3: Data Import

1. Basic Requirements

a. Scripts used to import data

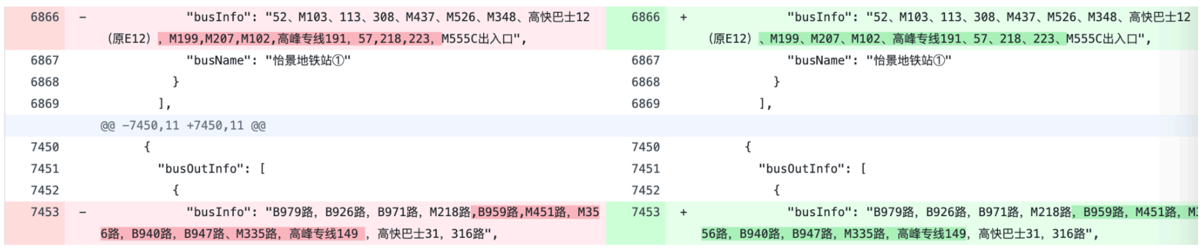
Script name	Author	Description
ImportScript.java	HOK Layheng	- When the script is run, it reads all JSON files and writes data to all tables. - It prompts the user to input a database server, an import method, and data volume. - The class then instantiates StationImport.java, LineImport.java, CardImport.java, PassengerImport.java, and RideImport.java to read the respective JSON file and write data to the correct tables.

import_script.py	HOK Layheng	- The script reads all JSON files and writes data to all tables with just that one .py file.

b. Description of how the script is used

The data were imported based on the provided JSON files with some modifications to ease the import process. Some changes to the JSON files are shown below:

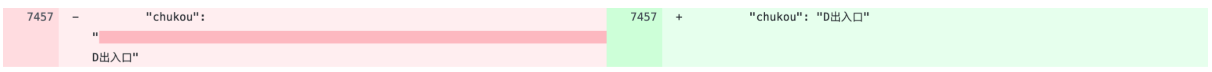
- Most lines in busInfo and textt in stations.json were modified so that each line has the same type of word separator (delimiter).



```
-                "busInfo": "52、M103、113、308、M437、M526、M348、高快巴士12
     (原E12) , M199,M207,M102,高峰专线191, 57,218,223, M555C出入口",
                 "busName": "怡景地铁站①"
               }
             ],
     @@ -7450,11 +7450,11 @@
           {
             "busOutInfo": [
               {
-                "busInfo": "B979路, B926路, B971路, M218路,B959路,M451路, M35
     6路, B940路, B947路、M335路, 高峰专线149 , 高快巴士31, 316路",
```

```
+                "busInfo": "52、M103、113、308、M437、M526、M348、高快巴士12
     (原E12) 、M199、M207、M102、高峰专线191、57、218、223、M555C出入口",
                 "busName": "怡景地铁站①"
               }
             ],
           {
             "busOutInfo": [
               {
+                "busInfo": "B979路, B926路, B971路, M218路, B959路, M451路, M
     56路, B940路, B947路, M335路, 高峰专线149, 高快巴士31, 316路",
```

- Some unnecessary spaces were removed if found during the data cleaning process.



```
-          "chukou":
   "
   D出入口"
```

```
+          "chukou": "D出入口"
```

2. Data Accuracy checking

To check our design's accuracy, and whether all data have been correctly imported into our database, we have prepared in advance the following SQL queries:

SELECT district, line_name, COUNT(station_name) no_stations FROM station s JOIN line_detail ld ON s.english_name = ld.station_name GROUP BY district, line_name;
 	
SELECT COUNT(DISTINCT english_name) total_station_num
FROM station;

SELECT CASE WHEN gender = '女' THEN 'FEMALE' WHEN gender = '男' THEN 'MALE' END AS gender, COUNT(DISTINCT id_num) FROM ride_by_id_num rbin JOIN passenger p ON rbin.user_num = p.id_num GROUP BY gender;
 	
SELECT CASE WHEN gender = '女' THEN 'FEMALE' WHEN gender = '男' THEN 'MALE' END AS gender, COUNT(id_num)
FROM passenger GROUP BY gender;

SELECT district, COUNT(id_num) passenger_num FROM ride_by_id_num rbin join passenger p on rbin.user_num = p.id_num GROUP BY district; SELECT district, COUNT(id_num) passenger_num FROM passenger GROUP BY district;
SELECT s.english_name, e.exit_gate, e.bus_line, e.landmark FROM station s JOIN (SELECT station_name, exit_gate, bus_line, NULL AS landmark FROM bus_exit_info bei JOIN bus_info bi ON bei.bus_info_id = bi.bus_info_id UNION ALL SELECT station_name, exit_gate, NULL AS bus_line, landmark FROM landmark_exit_info lei JOIN landmark_info li ON lei.landmark_id =

li.landmark_id) e ON s.english_name = e.station_name ORDER BY s.english_name, e.exit_gate;
SELECT name, start_station, end_station, start_time, end_time FROM passenger p JOIN ride_by_id_num rbin ON p.id_num = rbin.user_num JOIN route_pricing rp ON rp.pricing_id = rbin.pricing_id;
SELECT c.code, start_station, end_station, start_time, end_time FROM card c JOIN ride_by_card_num rbcn ON c.code = rbcn.user_num JOIN route_pricing rp ON rp.pricing_id = rbcn.pricing_id;
SELECT chinese_name, english_name, no_exits, district, line_name FROM station s JOIN line_detail ld ON s.english_name = ld.station_name LEFT JOIN (SELECT station_name, COUNT(DISTINCT exit_gate) AS no_exits FROM (SELECT station_name, exit_gate FROM landmark_exit_info UNION SELECT station_name, exit_gate FROM bus_exit_info) AS unique_exits GROUP BY station_name) exits_info ON s.english_name = exits_info.station_name;
SELECT l.line_name, l.start_time, l.end_time, l.first_opening, COUNT(ld.station_name), l.intro FROM line l JOIN line_detail ld ON l.line_name = ld.line_name GROUP BY l.line_name, l.start_time, l.end_time, l.first_opening, l.intro;

3. Advanced Requirements

Table 1: Testing environment

ID	OS	Chip	Memory	SSD	IDE Tool	Database Tool
1	macOS Sonoma 14.4.1	Apple M3 Pro	18GB	1TB	IDEA 2024.1 (CE) or PyCharm 2023.3.4 (CE)	Datagrip 2024.1
2	Windows 11 Home 23H2	12th Gen Intel(R) Core(TM) i9-12900H	32GB	1TB	IDEA 2024.1 (CE)	Datagrip 2024.1
3	Ubuntu 22.04.4 (VM)	Apple M1 Pro	16GB	512GB	IDEA 2024.1 (CE)	Datagrip 2024.1

a. Different Import Methods

- Method 1 *(original script)*:

This method utilizes the java.sql library. Firstly, we established a connection to our PostgreSQL server. Then we read all data from JSON files. Next, we iterated through each datum and created PreparedStatement for each insert statement. Lastly, we called the *executeUpdate()* method to execute each statement individually.

- Method 2 *(optimized script)*:

This method also utilizes the java.sql library and employs the same data reading algorithm as Method 1. The difference is now we make use of the *executeBatch()* method. So we iterated through the whole data, created each insert statement with PreparedStatement, and added each PreparedStatement to a batch for a batch execution.

- Method 3 *(running a .sql file)*:

We used a Java program to generate SQL insert statements and wrote them into a .sql file by employing the same data reading algorithm mentioned above. Then we run the file in DataGrip.

Since we are using the same data reading algorithm across all three methods, we will use an average runtime for our subsequent tests. We initially gathered three different runtimes—504 ms, 546 ms, and 552 ms—and calculated an average runtime of 534 ms.

Table 2: Importing methods

Testing Environment	Method	Average Read Time (ms)	Write Time (ms)	Total Time (ms)	Throughput (statements/s)
1	1	534	206396	206930	8636.91
1	2	534	2114	2648	97632.92
1	3	534	13581	14115	15197.41

Table 2 illustrates varying performance metrics across different methods, with Method 2 showing the highest throughput and Method 1 having the slowest total time. This makes Method 2 the standard testing method in the upcoming experiments.

b. Data import across multiple systems (macOS, Windows, Linux)

In testing the process of importing data across multiple systems, we employed the fastest import method, which is Method 2, with 100% import volume.

Particularly in a Linux environment, we utilized *VMware Fusion*, a virtual machine manager, to facilitate the operations. This setup allowed us to efficiently switch between different system states and test the robustness and compatibility of the data import process.
Note: When running Java code on Linux through a virtual machine, it is essential to consider the potential impact on performance and resource allocation, as virtualization can introduce overhead and affect the efficiency of the system.

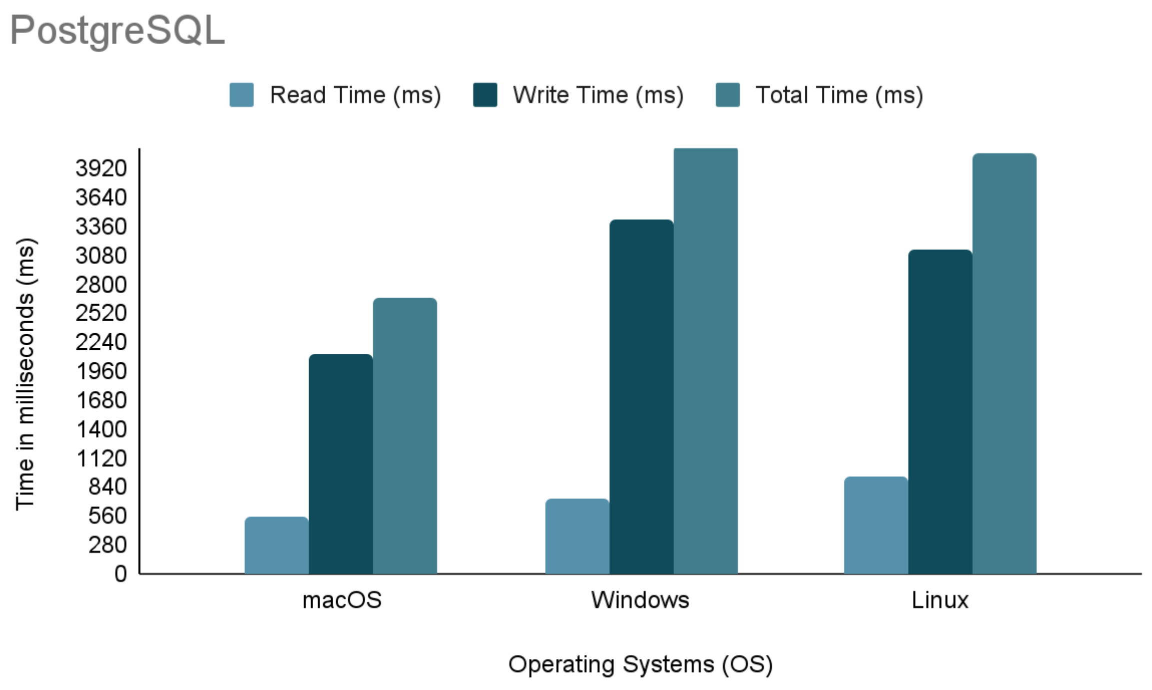


Figure 2: Runtime comparison between different OS

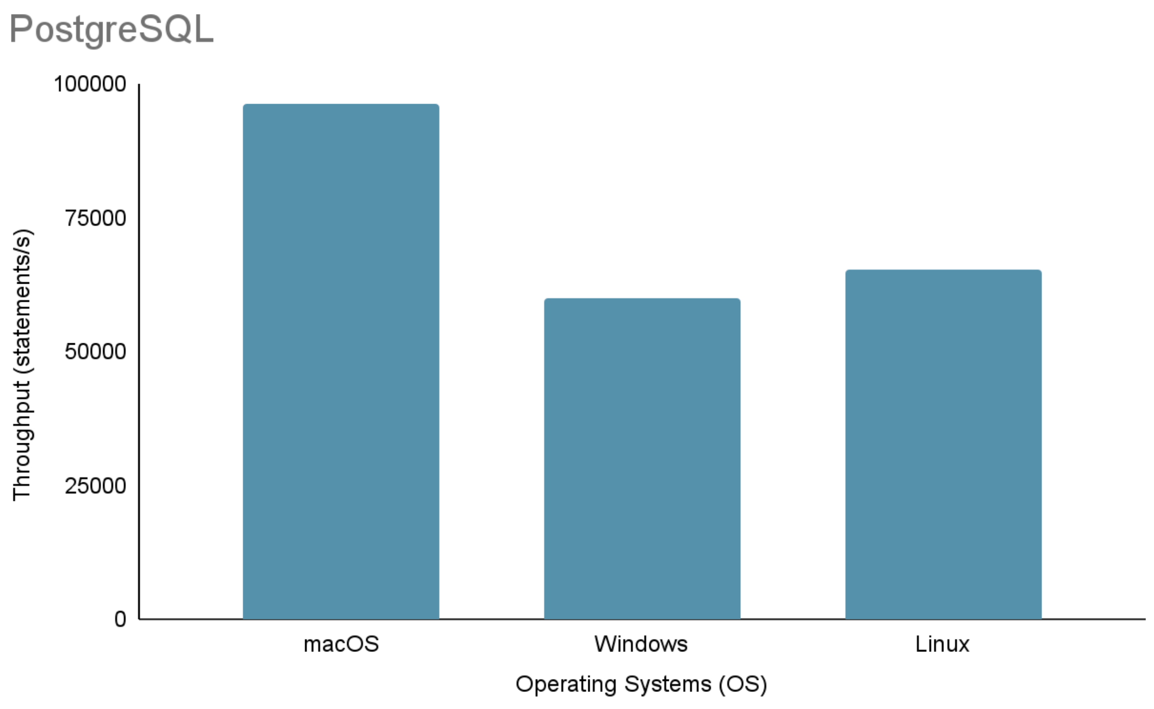


Figure 2: Throughput comparison between different OS

Based on the analysis above, it is clear that Environment 1 (macOS) shows the best performance with the shortest total time of 2648 ms and the highest throughput at 97,632.92 statements/s. In contrast, Environment 2 (Windows) is the slowest with a total time of 4117 ms and the lowest throughput at 60,669.02 statements/s.

c. Import using various programming languages

In this experiment, we used Method 2 mentioned above for Java code since it is the fastest. Meanwhile, for Python, we wrote an import method that made use of Psycopg2 to communicate with PostgreSQL database server.

Table 3: Java VS Python

Testing Environment	Programming Language	Read Time (ms)	Write Time (ms)	Total Time (ms)	Throughput (statements/s)
1	Java	534	2114	2648	96263.95
1	Python	390	7237	7627	28119.66

The data from Table 3 show that Java outperforms Python in both speed and throughput in this test, indicating its superior efficiency for read-write operations.

d. Experiment with other databases (MySQL)

We developed three different import methods for both PostgreSQL and MySQL, but we only ran an experiment on Method 2 as it is a developer's choice for importing data. Both PostgreSQL and MySQL have a similar database implementation design (Data Definition Language) and a similar import code design.

Table 4: PostgreSQL VS MySQL

Testing Environment	Method	Database	Read Time (ms)	Write Time (ms)	Total Time (ms)	Throughput (statements/s)
1	2	PostgreSQL	534	2114	2648	96263.95
1	2	MySQL	534	42315	42849	4809.22

Despite both being SQL-based databases, PostgreSQL is approximately 16 times faster when it comes to write time. This might be due to differences in database engine architecture and driver implementation (postgresql-42.2.5.jar for PostgreSQL and mysql-connector-j-8.3.0.jar for MySQL).

e. Data import with different data volumes

Managing and importing data of varying volumes is a crucial aspect of ensuring the performance, scalability, and reliability of a database system.

Before the import process, we noticed that the 'ride' data was notably larger in volume compared to the others. Based on this idea, we decided to test data import with different volumes on the ride.json only. Since the weight of data is not consistent, importing less volume for the other tables might result in a serious issue due to the connectivity between the tables.

Initially, we started by importing the full data (100% volume) for all the other tables except the ‘rides_by_id_num’ and ‘rides_by_card_num’’ tables to ensure the consistency of our design. To manage this effectively, we adopted a phased import strategy for the 'ride' data, beginning with a 20% subset of the data, which equated to 20,000 records. This initial phase allowed us to assess the impact on system performance and make necessary adjustments to the import process without compromising the database's stability. After successful validation and performance tuning, we proceeded with importing 50% of the data, and finally, the remaining portion to complete the 100% data import. Note that we used Method 2 for all imports as it is the fastest.

The table below shows the impact of each subset on the overall performance throughout the process:

Table 5: Import volumes

Testing environment	Method	Volume	Read time (ms)	Write time (ms)	Total time (ms)	Statement count	Throughput (statements/s)
1	2	20%	507	808	1315	80165	99214.11
1	2	50%	521	1338	1859	130849	97794.47
1	2	100%	534	2114	2648	203502	96263.95

From Table 5, as the volume of data increases (from 20% to 50% and to 100%), both read and write times tend to increase, resulting in longer total times for the operations. However, these numbers do not give any insightful meaning as we had different import volumes. If we look at the number of throughputs instead, the throughput (statements/s) gradually decreases with increasing data volume, indicating that the system becomes less efficient at processing statements as the workload increases.

V. Conclusion

Throughout this project, our team successfully designed, implemented, and tested a robust database system for managing a transit system. This experience provided us with invaluable insights into the practical aspects of database management, from the conceptualization with an E-R diagram to the intricacies of data import across different environments using multiple programming languages.

Our experiments with different import methods, database tools, and programming languages taught us the importance of optimization and adaptability in database design. Particularly, we learned that performance and efficiency can vary significantly across different systems and setups, highlighting the need to test and tailor database solutions according to specific operational requirements. Moreover, the project allowed us to understand the significance of data integrity and accuracy in real-world applications, as we implemented various SQL queries to validate the data imported into our database. These experiences have not only enhanced our technical skills but also enriched our understanding of the theoretical principles discussed in class, making them more tangible and applicable.

In conclusion, Project 1 was a comprehensive learning journey that equipped us with the skills and knowledge necessary to design and manage efficient database systems, which are pivotal in the functioning of modern technological infrastructures.