Database Project 1

Harrold Tok Kwan Hang 12212025, Saruulbuyan Munkthur 12212643

April 30, 2024

Basic Information of Your Group

1. Group Members: Harrold Tok 12212025, Saruulbuyan Munkhtur 12212643
2. Tuesday 2pm 507 Lab Session
   (a) **Saruulbuyan Munkhtur (Student ID: 12212643)(50%)**
   - Completed Task 1: E-R Diagram
   - Report Outline
   - Designed the tables and columns for Task 2: Relational Database Design
   - Imported the data using python
   - Import data with different volumes

   (b) **Harrold Tok (Student ID: 12212025) (50%)**
   - Wrote the scripts for Task 3.1: Data Import
   - Prepared the SQL queries for Task 3.2: Data Accuracy Checking
   - Experimented with MongoDB
   - Provided another method of importing data
   - Import data across multiple systems
   - Optimized java script

1 E-R Diagram

The E-R diagram for the subway system management database was created using the **** online diagramming tool. The diagram, as shown in Figure 1, depicts the key entities and their relationships within the database.

The relationships between the entities follow the standard E-R diagram notation, with one-to-many and many-to-many relationships represented with cardinality notations. By following the best practices (such as the three normal forms) for E-R diagram design, our team has created a robust and scalable database structure that can effectively store and manage the complex data related to the subway system.

The software used to create this ER-Diagram is draw.io (https://app.diagrams.net/)

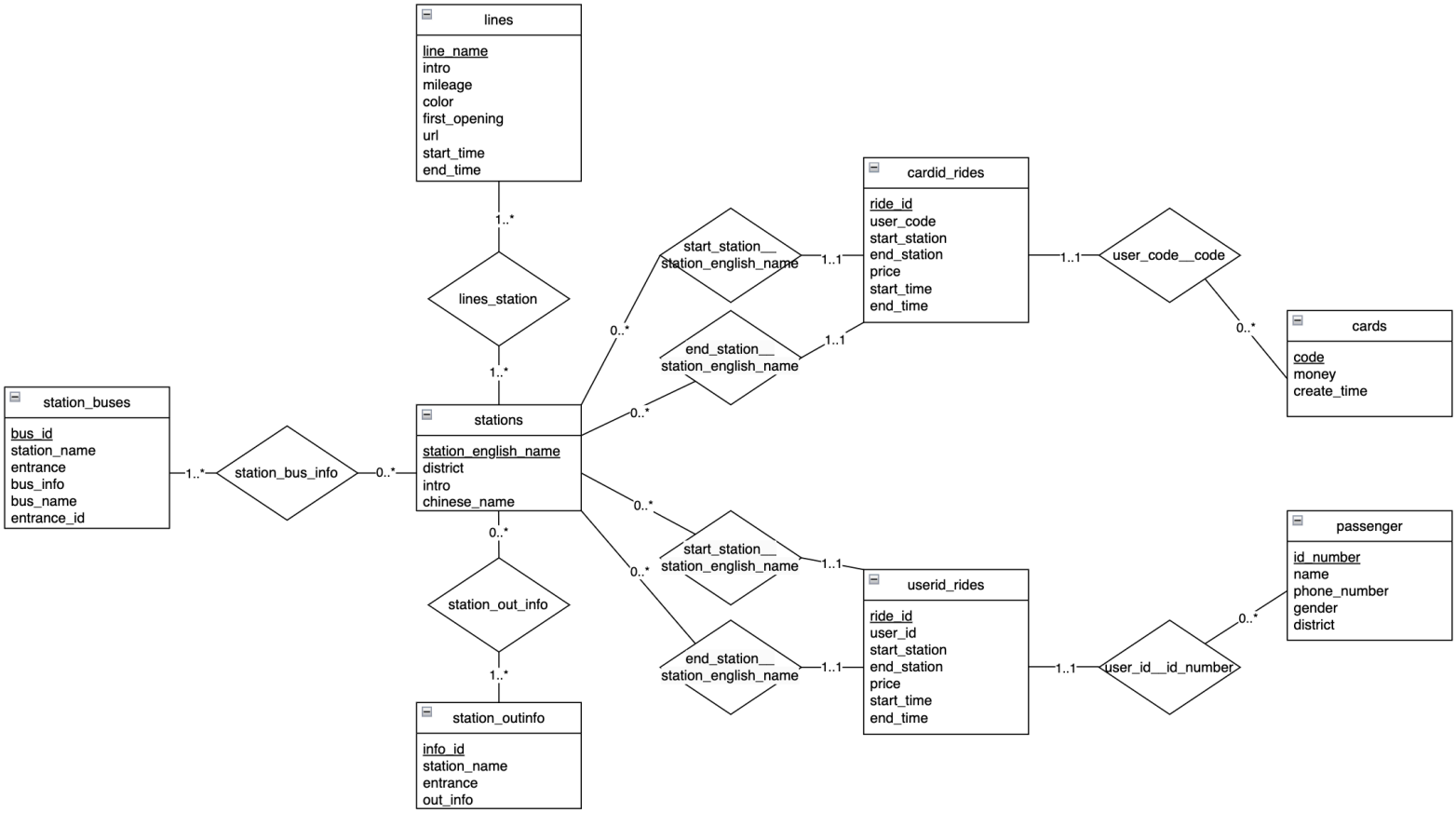

Figure 1: ER-Diagram by draw.io

2 Relational Database Design

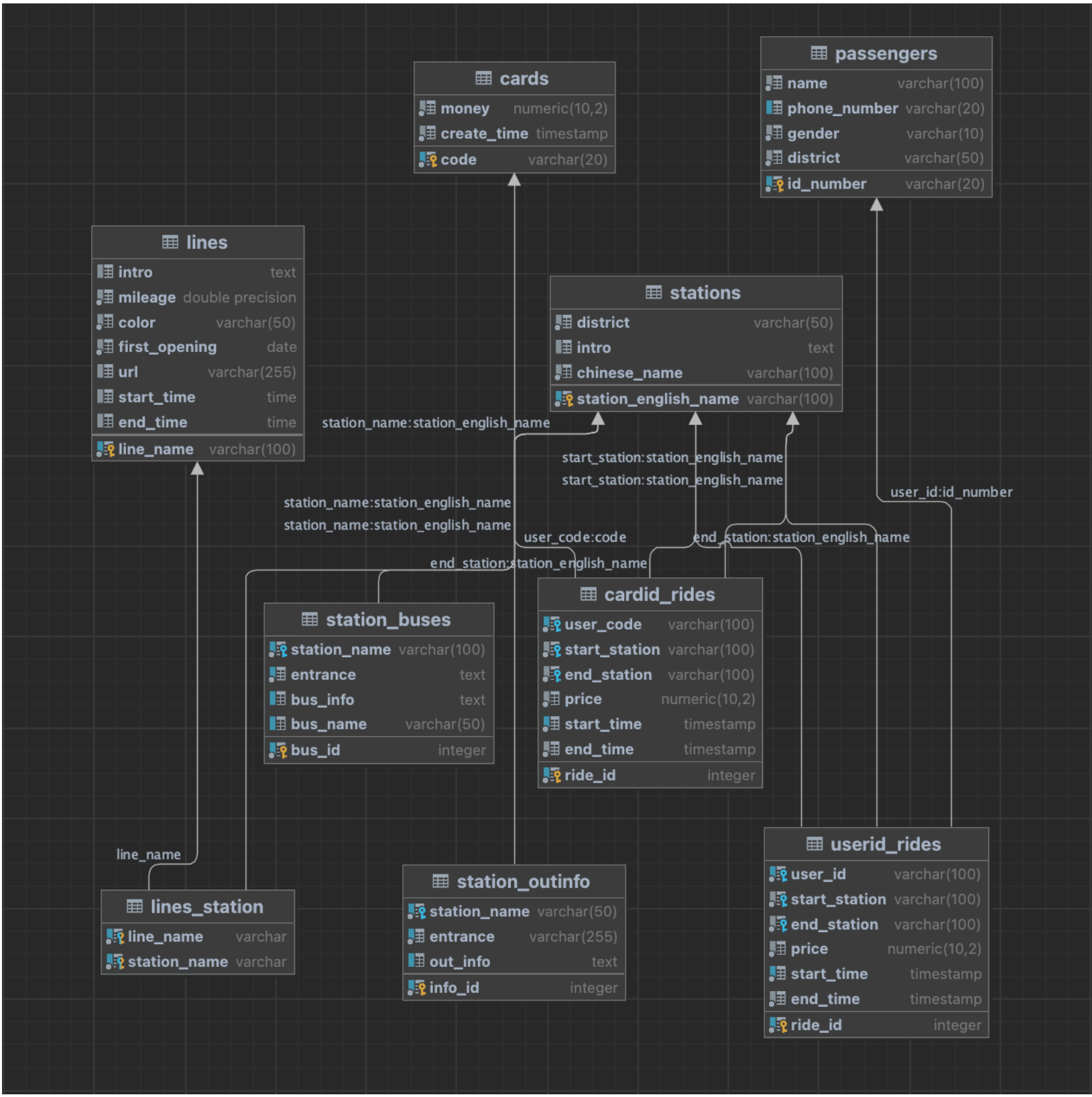

Figure 2: ER-Diagram by DataGrip

Table Designs and Descriptions

1. Passengers:

Contains information of all passengers taking the metro.

- `id_number`: [PK] Identifier for each passenger. (unique)
- `name`: Passenger's name. (Not Null)
- `phonenumber`: Passenger's phone number (unique).
- `gender`: Passenger's gender. (Not Null)
- `district`: Region/district (e.g., Chinese Mainland, Taiwan, Hong Kong, Macao). (Not Null)

2. Cards:

Contains information of all cards registered to access metro.

- `code`: [PK] Identifier for each card. (unique)
- `money`: Current balance on the card. (Not Null)
- `create_time`: Timestamp when the card was created. (Not Null)

3. Lines:

Information about the lines of the metro.

- `line_name`: [PK] Name of the subway line (unique).
- `intro`: Description of the subway line.
- `mileage`: Length of the subway line. (Not Null)
- `color`: Color associated with the subway line. (Not Null)
- `first_opening`: Date when the line was first opened. (Not Null)
- `url`: URL for more information.

4. Stations:

Entails information of each station.

- `station_english_name`: [PK] English name of each station. (unique)
- `chinese_name`: Name of the station in Chinese.
- `district`: District or area of the station.
- `intro`: Introduction to the station.

5. Station_outinfo:

Provides information about significant things near a station's exit.

- `info_id`: [PK] Unique ID for each out_info.
- `station_name`: Foreign key referencing the `Stations` table. (Not Null)
- `entrance`: Exit name (e.g., Exit A, Exit B). (Not Null)
- `outinfo`: Information about nearby landmark.

(station_name, entrance, out_info) is unique

6. Station_buses:

Provides bus information near the station's exit.

- `bus_id`: [PK] Unique identifier for each bus.
- `station_name`: Foreign key referencing `station_english_name` from the `Stations` table. (Not Null)
- `entrance`: Identifier for the exit. (Not Null)
- `bus_info`: Information about bus or schedule at the station and its relative entrance.

(station_name, entrance, bus_info) is unique

7. Lines_station:

Shows which stations belong to which lines.

- `line_name`: [PK] Foreign key referencing the `lines` table. (Not Null)
- `station_name`: Foreign key referencing `Stations`. (Not Null)

Many-to-many relationship between subway lines and stations. (line_name, station_name) is unique

8. Cardid_rides:

Contains previous rides taken using card.

- `ride_id`: [PK] Unique identifier for each ride.
- `user_code`: Foreign key referencing `code` in `Cards`. (Not Null)
- `start_station`: Foreign key referencing `station_english_name` in `Stations`. (Not Null)
- `end_station`: Foreign key referencing `station_english_name` in `Stations`. (Not Null)
- `price`: Ride price. (Not Null)
- `start_time`: Timestamp when the ride started. (Not Null)
- `end_time`: Timestamp when the ride ended.

(Not Null)

(user_code, start_time) is unique

9. Userid_rides:

Contains previous rides registered with a passenger's id_number.

- `ride_id`: [PK] Unique identifier for each ride.
- `user_id`: Foreign key referencing `passenger_id` in `Passengers`. (Not Null)
- `start_station`: Foreign key referencing `station_english_name` in `Stations`. (Not Null)
- `end_station`: Foreign key referencing `station_english_name` in `Stations`. (Not Null)
- `price`: Ride price. (Not Null)
- `start_time`: Timestamp when the ride started. (Not Null)
- `end_time`: Timestamp when the ride ended. (Not Null)

(user_id, start_time) is unique

3 Data Import

3.1 Table of scripts

3.1.1 Importer.java

This script is responsible for importing cards and passengers. First, we establish a connection using Java Database Connectivity (JDBC) to the local database. After that, we initialize a list of type Cards and use ReadJSON.readJsonArray to deserialize the file Cards.json and extract the necessary data to import. Thirdly, after having added all the data of Cards into a list, we iterate through the list. For each object in the list, it is turned into the appropriate prepared statement by using the superclass function .toString(). The prepared statement follows the general semantic (INSERT INTO table_name (attribute1, attribute 2, ...) VALUES (value1, value2, ...)). The prepared statement is then executed with pstmt.executeUpdate(); then the statement is closed. After all data has been imported, the connection is closed for good practice. This processes is repeated for

3.1.2 Lines.java

Establishes a connection to the database. It iterates over the keys in the JSONObject to extract data for different lines (e.g., railway or subway lines). Then, A PreparedStatement is used to create an SQL INSERT INTO query for inserting line details into the PostgreSQL database.

Script Name	Author	Description
Importer.java	Harrold	Imports cards, passengers. Each has a respective java file (e.g Cards.java) containing preparedStatement.
Lines.java	Harrold	Imports lines data into lines table.
station_outinfo.java	Saruulbuyan	Imports station_outinfo table
stations_buses.java	Saruulbuyan	Imports station_buses table
lines_stations	Saruulbuyan	Imports lines_stations table
OptimizedImporter	Harrold	Optimized Importer for cardid_rides and userid_rides
stationsImporter	Harrold	Imports data for stations table
DifferentVolumes	Saruulbuyan	Import For Different Volume
mongoImporter	Harrold	Import For MongoDB
CsvImporter	Harrold	Import using csv
Importer.py, OptimizedImporter.py	Harrold	Import using python

Table 1: Scripts Information

3.1.3 station_outinfo.java

Breaks apart station.json nested arrays to extract entrance as well as individual out_info. We perform data cleaning to replace delimeters such as "[、.,, ();;]" found in the JSON file to a single comma ','. Then, we split the data. This allows for more congruent and accurate data when importing. It fixes nonsensical data. Then, we use INSERT INTO query to insert the data. If we find any duplicate entries, we ignore it and do not import.

3.1.4 stations_buses.java

Breaks apart station.json nested arrays to extract entrance, bus_name as well as individual bus_info. We perform data cleaning to replace delimeters such as "[、.,, ();;]" found in the JSON file to a single comma ','. Then, we split the data. This allows for more congruent and accurate data when importing. It fixes nonsensical data. Then, we use INSERT INTO query to insert the data. If we find any duplicate entries, we ignore it and do not import.

3.1.5 lines_stations.java

Creates a JSON object lines and extracts all the stations associated with that line. For each stations in each line, it uses INSERT INTO query to insert station and the line it belongs to.

3.1.6 OptimizedImporter.java

This file is an optimized version of Importer.java such that it uses a techinque called batching which increases the effeciency of importing.

3.1.7 stationsImporter.java

Imports data for stations table. Establishes a connection to the database and uses INSERT INTO query to load data into table.

3.1.8 DifferentVolumes.java

Script to experiment importing with different data volumes. First, we split original file into N seperate JSON files, then add each file to a list. Secondly, for each file, we batch them and then import into the database.

3.1.9 mongoImporter.java

First, we import various libraries and packages to connect and import data to a MongoDB database. Since MongoDB uses BSON as its native storage format, we set up a custom BSON codec registry to deserialize nested arrays for MongoDB, this is because MongoDB does not have a built-in way to handle nested arrays. The script then connects to the database.

Next, it reads in data from various JSON files and creates collections within the database for each type of data (such as cards, rides, passengers, etc.).

The script then reads through the JSON data and inserts it into the appropriate collections in the database.

3.1.10 CsvImporter

This program reads a JSON file, writes selected information to a CSV files, then stores the csv file in the resource section. After that, it reads the CSV file from the resource section and imports into a PostgreSQL database using PostgreSQL COPY FROM statement.

3.1.11 Python Scripts

Importer.py imports data from a JSON file into a PostgreSQL database using psycopg2. First, it connects to the database, then reads the JSON file and one-by-one parses each prepared statement into the database

OptimizedImporter.py first connects to a PostgreSQL database, then reads JSON data, and performs batch insertions into the table.

3.2 Data Accuracy Checking

```
--1.
SELECT district, count(station_english_name) FROM stations GROUP BY district;
SELECT line_name, count(station_name) FROM lines_station GROUP BY line_name;
SELECT count(station_english_name) total_stations FROM stations;

--2.
SELECT count(gender) Male, (SELECT count(gender) FROM passengers WHERE gender = '女') Female
FROM passengers WHERE gender = '男';

--3.
SELECT count(district) Mainland,(SELECT count(district) FROM passengers
WHERE district = 'Chinese Hong Kong') HONG_KONG,
(SELECT count(district) FROM passengers WHERE district = 'Chinese Macao') MACAU,
(SELECT count(district) FROM passengers WHERE district = 'Chinese Taiwan') TAIWAN
FROM passengers WHERE district = 'Chinese Mainland';
```

```
--4.
SELECT distinct sb.station_name, sb.entrance, bus_info
FROM station_buses sb
WHERE sb.station_name = 'Luohu' AND sb.entrance = 'D出入口';
SELECT distinct so.station_name, so.entrance, out_info
FROM station_outinfo so
WHERE so.station_name = 'Luohu' AND so.entrance = 'D出入口';

--5.
SELECT * FROM passengers p JOIN userid_rides r ON p.id_number = r.user_id
WHERE id_number = '340181197110010511';

--6.
SELECT * FROM cards JOIN cardid_rides c ON cards.code = c.user_code WHERE code = '882132348';

--7.
SELECT chinese_name, station_english_name, (SELECT COUNT(DISTINCT entrance) AS distinct_entrances FROM (
    SELECT entrance FROM station_buses WHERE station_name = 'Laojie'
    UNION
    SELECT entrance FROM station_outinfo WHERE station_name = 'Laojie') AS combined_entrances),
district, line_name as belongs_to
FROM stations JOIN lines_station ON stations.station_english_name = lines_station.station_name
WHERE station_english_name = 'Laojie';

--8.
SELECT lines.line_name, start_time, end_time, first_opening, count(station_name) as number_of_stations,
mileage, color, url, intro
FROM lines JOIN lines_station ON lines.line_name = lines_station.line_name
WHERE lines.line_name = '8号线' group by lines.line_name;
```

3.3 Advance Requirements

3.3.1 Optimized Script

The optimized script for import can be found among the script files named OptimizedImporter.java. The improved method makes use of a technique called batching to improve it's efficiency when inserting into the database. In the following example, we timed how long it took to import from rides.json (100,000 data points). The timing method uses System.nanoTime() to calculate the time. For the time taken to import, the non optimized script took 723.8 seconds (12minutes) whereas the optimized script took just over 1 second, which is a world of difference. The initial design was slow because for every data in rides.json, a new preparedStatement was created then imported individually into the database. The multiple initialization of the PreparedStatement as well as the execution of the prepareStatement was costly.

Secondly, we experimented with a method where we convert the json file into a csv file and use the command "COPY lab3.cardid_rides FROM '" + csvFilePathCode + "' WITH (FORMAT CSV, HEADER TRUE)" to insert the csv file into our database. The program is called CsvImporter.java. This method of importing data ended up being slightly slower than OptimizedImporter(Batching) by around 0.1 seconds. This took around 6 hours to complete.

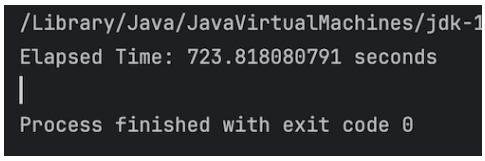

Figure 3: Time For Non-Optimized Script

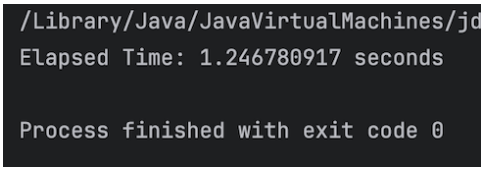

Figure 4: Time For Optimized Script

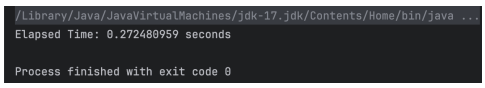

Figure 5: Time For Importing Cards With Batching

Figure 6: Time For Importing Cards with a Different Method

3.3.2 Import Across Multiple Systems

This section presents a comparative analysis of data import performance on macOS and Windows using the OptimizedImporter.java script. When running the script, the time taken to import all data from rides was 2.10 seconds on Windows, while on macOS, the same process completed in just 1.25 seconds. We believe the reason for the difference in execution time is due to systems resource management as well as file system operations. Since the dependencies were already set up on our windows, it took 2 hours.

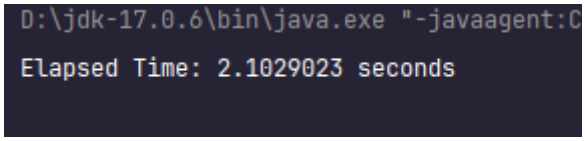

Figure 7: Import Time For Windows

3.3.3 Import Using Various Programming Languages

In this subsection, we demonstrate importing data using Java and Python and providing a comparative analysis. As we saw in Figure 2 above, the non optimized script took 10 minutes to import ride.json and 1.25 seconds with the optimized version.

There is not much difference between the Non-Optimized and Optimized version of the Python script. This is most likely to due low latency to the database so batching would not impact that much. The reason for such disparity in the java programs, if we were to guess, is due to instantiating the PreparedStatement Object 100,000 times, versus only a few times by batching. Finally, we believe the reason the optimized Java script is much faster than the optimized Python script is due to java's performance characteristics that python lacks such as Compile language, Just-In-Time Compilation, Strong Type Safety and Multi-threading which allows for a more efficacious execution on computationally taxing operations. Overall, 5 hours to complete.

Figure 8: Time For Non Optimized Python Script

Figure 9: Time For Optimized Python Script

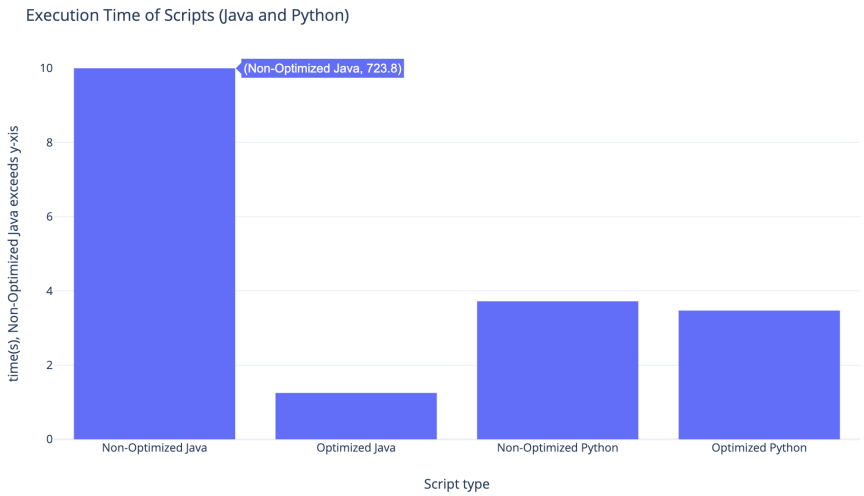

Figure 10: Graph: Comparison Of Scripts: Java (L) vs Python (R)

3.3.4 Other Databases: MongoDB

We experimented with MongoDB. All in all, to set this up took about 15-20 hours in total, including installing the script, customizing codec as well as database design. The method of importing the data into the Mongo Database can be found in mongoImporter.java. Since MongoDB uses documents, each row in our postgresql database contains the same information as one document in MongoDB. We found that Querying to get information in MongoDB is in general tougher than postgresql.

```
var specific_passenger_name = "王俊宏"
db.Passengers.aggregate([
  {
    $match: {
      name: specific_passenger_name
    }
  },
  {
    $lookup: {
      from: "userid_rides",
      localField: "id_number",
      foreignField: "user_code",
      as: "matched_rides"
    }
  },
  {
    $match: {
      matched_rides: { $ne: [] }
    }
  },
  {
    $project: {
      _id: 0,
      passenger_name: "$name",
      ride_details: "$matched_rides"
    }
  }
])
```

Figure 11: Query to obtain information of all rides taken by a specific passenger

3.3.5 Import With Different Data Volumes

Importing data efficiency depends on the size of the input. Larger files face more difficulties in importing. For example, certain compression techniques, indexing strategies, or data partitioning methods may work better on smaller data chunks. The ride.json file was chosen to split because it's the largest file containing 100,000 rows. Using the json library in python, the data volume was read first and then partitioned into either 2, 5 or 10 separate json files. Each of the partitions' import time was measured.

The results tell us that as the number of batches increase, the execution time decreases exponentially. This shows batching is a suitable optimization strategy for importing data. Due to the exponential nature of the graph, it suggests anything greater than 10 would not be a significant improvement. All in all, 2 hours to complete.

```
/Library/Java/JavaVirtualMachines/jdk-17.jdk/Contents/Home/bin/java ...
Elapsed Time For 2 Seperated Volumes (50000): 1.077838125 seconds

Process finished with exit code 0
```

Figure 12: Execution Time For 2 Datasets

```
/Library/Java/JavaVirtualMachines/jdk-17.jdk/Contents/Home/bin/java ...
Elapsed Time For 5 Seperated Volumes (20000): 1.049534792 seconds

Process finished with exit code 0
```

Figure 13: Execution Time For 5 Datasets

```
/Library/Java/JavaVirtualMachines/jdk-17.jdk/Contents/Home/bin/java ...
Elapsed Time For 10 Seperated Volumes (10000): 1.029110917 seconds

Process finished with exit code 0
```

Figure 14: Execution Time For 10 Datasets

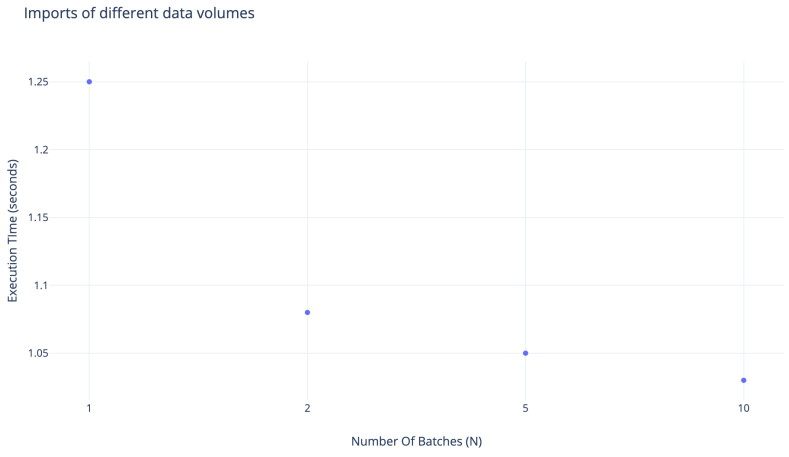

Figure 15: Graph: Number Of Batches VS Execution Time