CS307 Project 1

Basic Information

Member	Student ID	Lab Session	Contribution Rate
李天宇	12212824	Thursday 3-4 Zhu	50%
李怡萱	12212959	Thursday 3-4 Zhu	50%

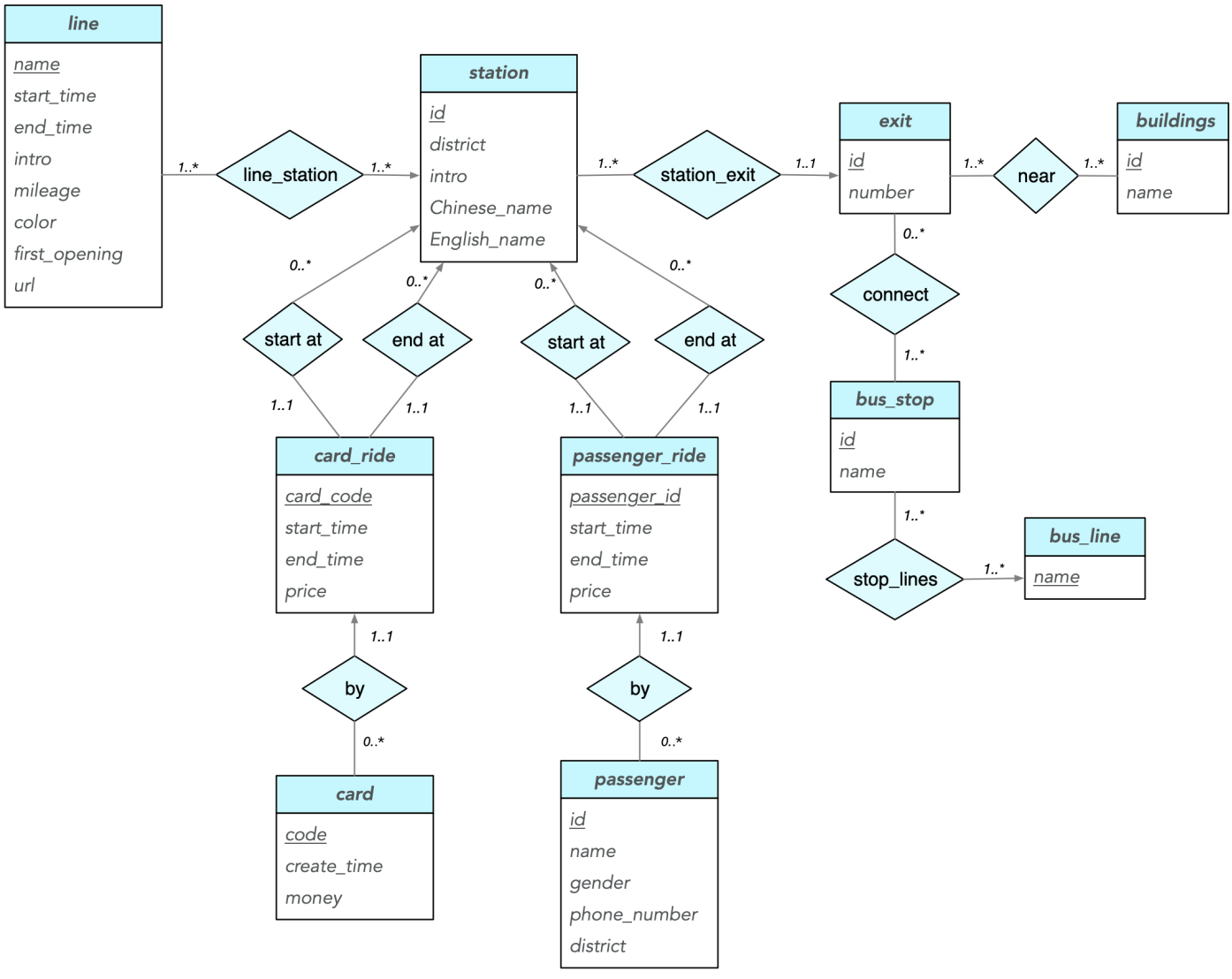
Contribution of work

李天宇：E-R diagram, data import (via Java), comparison between different systems, most of the report.

李怡萱：Table creation, data import (via Python) and its optimization (report included), data accuracy checking (report included), import data with different volumes (report included).

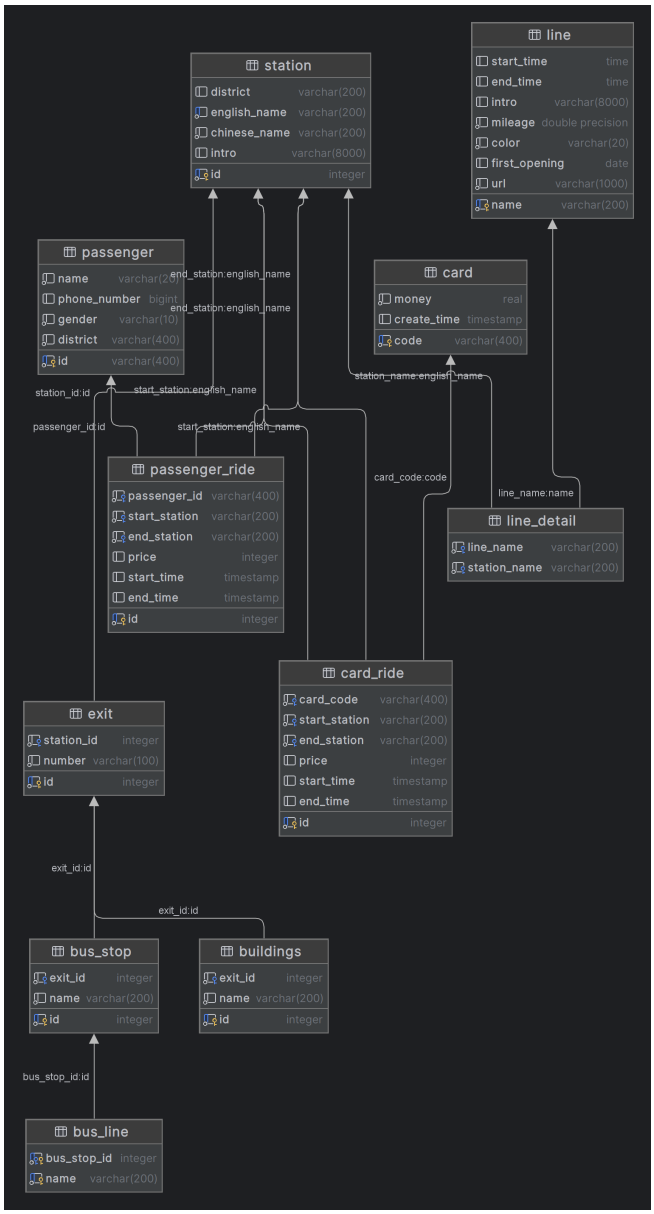
Task 1: E-R Diagram

We use **OmniGraffle** to draw the E-R diagram.



Task 2: Database Design

2.1 E-R Diagram By Datagrip



2.2 Table Design Description

By analyzing the initial data, we gain some key information:

- Column `stations` in `lines.json` has arrays consisting of dozens of stations.
- In `stations.json`, the data in column `bus_info` is nested, which contains more information including `busOutInfo` and `chukou`, and `busOutInfo` can be divided into many names of bus lines.
- Column `out_info` in `stations.json` is quite complex as well, because not only an exit may correspond to several buildings, but also a building might be surrounded by more than one exit.
- Users in `ride` are related to card codes or passengers' IDs, which makes it difficult to set foreign keys.

Such information results in the files not satisfying the three normal forms. Therefore, to make the database meet the **3NF Requirement** and be as easy to expand as possible, we designed the following **11 tables**.

line (basic information of subway lines in Shenzhen)

- `name`: the name of the metro line (in Chinese), which is the primary key of the table
- `start_time`: Time of the first train of the day on this line
- `end_time`: Time of the end train of the day on this line
- `intro`: a paragraph of introduction to the line, including its Chinese and English name
- `mileage`: length of the line (km)
- `color`: color of the line
- `first_opening`: the opening time of the line
- `url`: URL of Baidu Encyclopedia

line_detail (subway lines and their stations)

- `line_name`: the name of the line
- `station_id`: the id of a corresponding station of the line

station (basic information of subway stations in Shenzhen)

- `id`: each station is assigned one, which is a self-increasing primary key
- `English_name`: the English name of the station
- `Chinese_name`: the Chinese name of the station
- `intro`: the introduction to the station
- `district`: the district where the station is located

exit (exits of subway stations)

- `id`: each exit is assigned one, which is a self-increasing primary key
- `station_id`: the id of the station where the exit is situated at
- `number`: the code differentiating exits of the same station

buildings (buildings near subway stations' exits)

- `id`: each building is assigned one, which is a self-increasing primary key
- `exit_id`: the id of the exit which is near the building
- `name`: the Chinese name of the building

bus_stop (bus stops near subway stations' exits)

- `id`: each bus stop is assigned one, which is a self-increasing primary key
- `exit_id`: the id of the exit which connects the bus stop
- `name`: the name of the bus stop

bus_line (bus lines in each bus stop)

- `name`: the name of the bus line
- `bus_stop_id`: the id of the bus stop on this line

card_ride (rides by cards)

- `id`: each ride is assigned one, which is a self-increasing primary key
- `card_code`: the code of the card taking this ride
- `start_station`: the English name of the depart station of the passenger
- `end_station`: the English name of the arrival station of the passenger
- `price`: the price of the journey
- `start_time`: the time of entering into the station
- `end_time`: the time of leaving the station

passenger_ride (rides by passengers)

- `id`: each ride is assigned one, which is a self-increasing primary key
- `passenger_id`: the ID of the passenger taking this ride
- `start_station`, `end_station`, `price`, `start_time`, `end_time`: same as **card_ride**

card (basic information of cards)

- `code`: card number, which is the primary key of this table
- `money`: balance of this card
- `create_time`: opening time of card

passenger (basic information of passengers)

- `id`: ID number of the passenger, which is the primary key of this table
- `name`: Chinese name of the passenger
- `phone_number`: phone number of the passenger
- `gender`: gender of passenger
- `district`: The region the passenger comes from

Scalability Analysis

- Table `passenger` and `card` can store all passengers. It's also easy to modify rows. Meanwhile, it can easily retrieve the list of passengers of a certain gender or district by keyword `group by`.
- Table `station`, `ride`, `exit`, `buildings`, and `bus_stop` use self-increasing primary keys, which can make insertion easier.

Task 3 Data Import

3.1 Script description

In this task, we choose Python to be programming language and create a script `DataInsertion.py` to import data in five `name.json` files. The files we created are listed in the table below.

File name	Author	Description
Table Construction.sql	李天宇	Run this file in Datagrip to construct tables.
Data Insertion.py	李怡萱	Run this script to complete the import of all the data.

Our steps to insert data is as follows:

1. Run the file `Table Construction.sql` in Datagrip to create 11 tables.
2. Run the file `DataInsertion.py` in spider. Before running, please pay attention to the following tips:
    - You should modify db_name, user, password, host, port to your own information when connecting to your local database. In our script, the information is:

```
conn = psycopg2.connect(
    dbname="project1",
    user="checker",
    password="123456",
    host="localhost",
    port="5432")
```

    - The file path in our script is **relative path**. If you want to replicate our import process, please either change it into absolute path, or make sure your file structure is similar to:

```
   ├── Data v1 // folder
   │   └── ride.json
   │   └── cards.json
   │   └── lines.json
   │   └── stations.json
   │   └── passenger.json
   ├── DataInsertion.py
```

Then, the data should be successfully imported to your database. The number of data entries of every table is as follows:

Table name	Number of data entries	Table name	Number of data entries
`card`	10000	`line_datail`	373
`card_ride`	36497	`exit`	1201
`passenger`	10000	`buildings`	5356
`passenger_ride`	63503	`bus_stop`	664
`line`	16	`bus_line`	4709
`station`	306		

3.2 Data accuracy checking

In order to check whether all data have been correctly imported into our database, we have made the following SQL queries for testing:

1. **The number of stations, in each district, on each line or in total.**

```
select district, count(id) from station group by district; -- each district
select line_name, count(station_name)from line_detail group by line_name; --
each line
select count(id) from station; -- in total
```

2. **Number of female passengers and male passengers respectively.**

```
select count(id) from passenger where gender = '女'; -- female
select count(id) from passenger where gender = '男'; -- male
```

3. **List the number of passengers from Mainland China, Hong Kong, Macau, and Taiwan.**

```
select count(id) from passenger
where district in ('Chinese Mainland', 'Chinese Taiwan', 'Chinese Hong kong',
'Chinese Macao');
```

4. **List the buses, buildings, or landmarks near a specific station exit.**

```
select sub1.English_name, sub1.number, buses, buildings
from (select distinct sub.English_name, number, array_agg(bln) over (partition
by (sub.English_name, number)) buses
      from (select distinct English_name, number, bus_line.name bln
            from exit
                     left join station s on exit.station_id = s.id
                     left join bus_stop bs on exit.id = bs.exit_id
                     left join bus_line on bs.id = bus_line.bus_stop_id
                     left join buildings b on exit.id = b.exit_id) sub) sub1
         join (select distinct sub.English_name, number, array_agg(bn) over
(partition by (sub.English_name, number)) buildings
               from (select distinct English_name, number, b.name bn
                     from exit
                              left join station s on exit.station_id = s.id
                              left join bus_stop bs on exit.id = bs.exit_id
                              left join bus_line on bs.id =
bus_line.bus_stop_id
                              left join buildings b on exit.id = b.exit_id)
sub) sub2 on sub1.English_name = sub2.English_name and sub1.number =
sub2.number
```

5. **List all information about a specific passenger's journey, including passenger name, entry station, exit station, date, and time.**

```sql
select name passenger_name, start_station entry_station, end_station
exit_station,
       split_part(to_char(start_time, 'YYYY-MM-DD HH24:MI:SS'), ' ', 1) as date,
       split_part(to_char(start_time, 'YYYY-MM-DD HH24:MI:SS'), ' ', 2) as time
from passenger_ride
         join passenger p on passenger_ride.passenger_id = p.id;
```

6. **List all journey records for a specific travel card, including card number, entry station, exit station, date, and time.**

```sql
select card_code, start_station entry_station, end_station exit_station,
split_part(to_char(start_time, 'YYYY-MM-DD HH24:MI:SS'), ' ', 1) as date,
split_part(to_char(start_time, 'YYYY-MM-DD HH24:MI:SS'), ' ', 2) as time
from card_ride
         join card c on c.code = card_ride.card_code;
```

7. **Query information about a specific subway station, including Chinese name, English name,number of exits, the district it is located in, and the subway line it belongs to.**

```sql
select distinct station_id, Chinese_name, English_name, district,
                array_agg(line_name) over (partition by Chinese_name) lines
from (select distinct station_id, Chinese_name, English_name, district,
line_name
      from station
               left join exit e on station.id = e.station_id
               join line_detail ld on station.English_name = ld.station_name) as
sub
where sub.English_name = 'Laojie'
order by station_id;
```

8. **Query information about a specific subway line, including start time, end time, first opening time, number of stations, and an introduction.**

```sql
select distinct name, start_time, end_time, first_opening first_opening_time,
count_station, intro introduction
from line
      join line_detail ld on line.name = ld.line_name
         join (select line_name, count(station_name) count_station
               from line_detail
               group by line_name) sub on sub.line_name = line.name
where ld.line_name = '16号线';
```

3.3 Extension to our tasks

3.3.1 Optimize our Python script

There are three main optimization directions for us :

1. **Using prepared statement**. It can allow us to reuse the same SQL query structure as it is executed, speeding up queries and reducing the risk of SQL injection.
2. **Using batch processing**. Batch processing can perform multiple inserts at once, which provides a significant performance gain.
3. **Multithreading**. Multithreading may provide performance gains since it can processing data through multiple threads simultaneously.

Considering data volume, our research will only focus on data insertion of table `ride`, since it has data volume of 100,000. And we create three files to pursue optimization based on the directions.

File name	Author	Description
Op_pps.py	李怡萱	Using prepared statements to optimize the import of data in table `ride`.
Op_batch.py	李怡萱	Using batch to optimize the import of data in table `ride`.
Op_MT.py	李怡萱	Using Multithreading to optimize the import of data in table `ride`.

Then we run these files, calculate their time of data processing and compare the results.

Noted: the time cost is the **average value** calculated after five-time testing.

Optimization Direction	Time cost before optimization (ms)	Time cost after optimization (ms)	Optimization rate
Preparedstatement	4670.38	4612.84	1.23%
Batch (`batch_size=500`)	4670.38	4525.55	3.10%
Multithreading	4670.38	2724.32	41.67%

Analysis

- Prepared statement has **slightly optimised** the time consumption, because SQL statements are written before the data is imported, and the time of rewriting is saved when invoking. However, since time consumption is mainly due to data sorting and import execution, the improvement is not obvious.
- Batch processing is a **further improvement** in time consumption, but since data is preprocessed (split into two lists according to the conditions of passengers and cards) in the loop, these additional actions increase the overall execution time. So even with batches, the result is not significantly improved.
- Multithreading **significantly speeds up** the data import process compared with the previous two methods. We set `CHUNK_SIZE = 1000`, which means each thread should process 1000 pieces of data at a time, and parallel processing proved to be very efficient when processing large amounts of data.

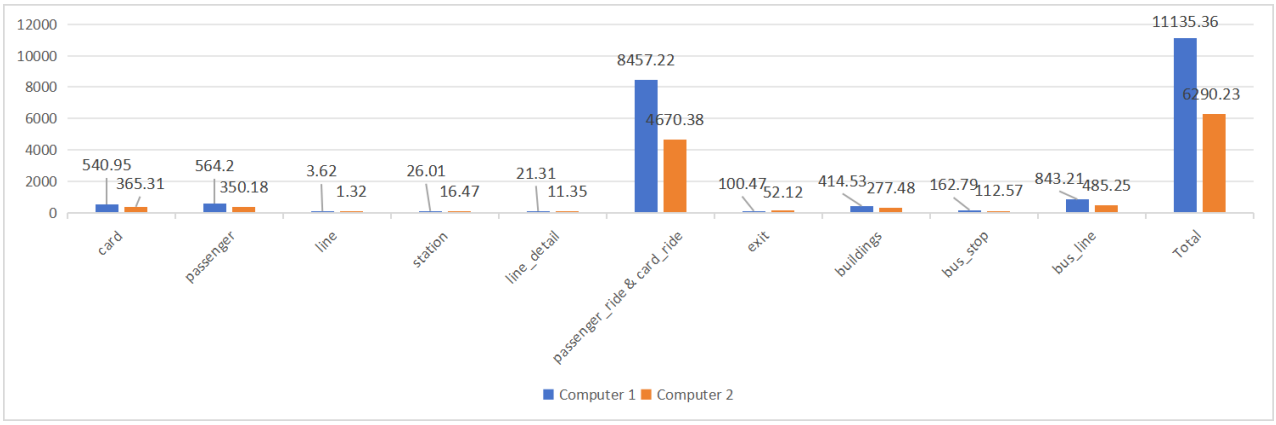
3.3.2 Import data in different systems

To test how well our scripts can perform in different systems, the two members of the group test with Pythons scripts on each one's computer (with different operation systems) respectively.

1. Test environment

	Computer 1	Computer 2
CPU	12th Gen Intel(R) Core(TM) i7-1260P @ 2.10GHz	Apple M1 2020
RAM	16.0 GB	8.0 GB
DBMS	`postgresql-16`	`postgresql-14.11`
OS	Windows 11 Home Chinese Version	macOS 12.3 (21E230)
PL	`python 3.11`	`python 3.11.3`
IDE	`Pycharm Community Edition 2021.3.1` for python	`Spider` for python

2. Result (insertion time / ms)



3. Analysis

- After comparison, we find that **MacOS system is significantly more efficient**, since computer 1 consumes nearly twice as much time in data insertion as computer 2. We speculate that in terms of operating systems, MacOS systems may be more efficient than Windows in terms of **system scheduling algorithms**, **kernel optimization strategies and system services**.
- However, due to different computer configurations, there are differences in CPU,RAM, DBMS version, IDE version, and Python version, which may also affect the results.

3.3.3 Import data with Java

We have also tried to import the data through Java scripts (with batches processing as optimization).

There are 2 Jar packages (`json-20240303.jar`, `postgresql-42.7.3.jar`) and 10 Java files (listed below) we used in this part, where each Java file corresponds with one table (except `ReadRide.java`) in our database:

`ReadPassenger.java` `ReadCard.java` `ReadLine.java` `ReadStation.java` `ReadLineStation.java` `ReadRide.java` `ReadBuilding.java` `ReadExit.java` `ReadBusStop.java` `ReadBusLine.java`

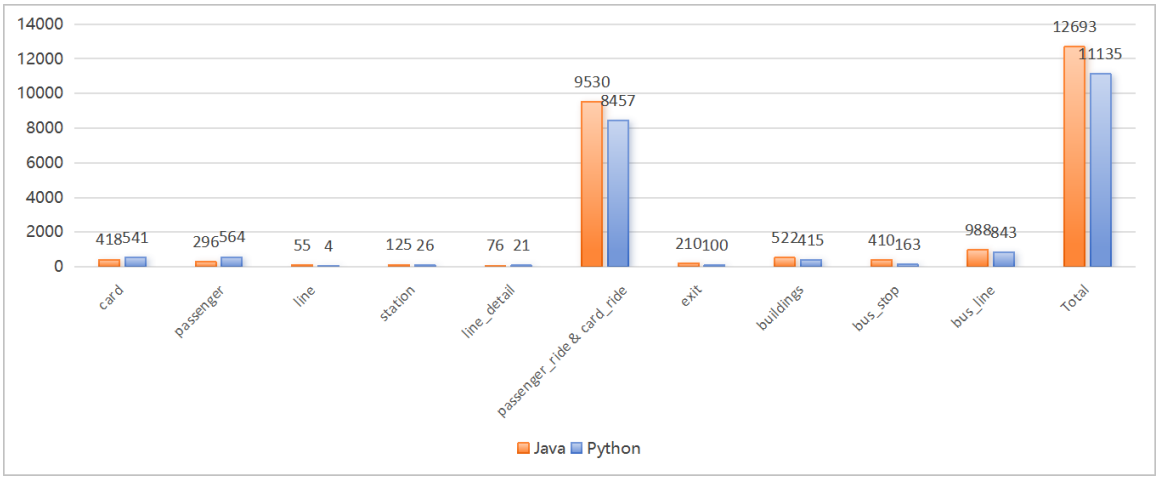
However, the structures of the scripts are similar. (Take `ReadPassenger.java` as an example)

```
public class ReadPassenger {
    private static final int BATCH_SIZE;
    private Connection conn;
    private PreparedStatement stmt;
    private static String filepath;
    private int count = 0;
    public void openDB();
    public void closeDB();
    public void truncatePassenger();
    private void loadPassenger(JSONObject jsonObject);
    private void insertPassenger();}
```

The general steps to import data in scripts are as follows:

1. Open database connection
2. Use buffered reader to read the data from `*.json` files and store them as JSON objects
3. Do **string operations** to get the information of every data(which is extraordinarily complex in `ReadBuilding.java` , `ReadExit.java` , `ReadBusStop.java`, `ReadBusLine.java`)
4. Use `Java` to execute `sql` sentences to import data into tables
5. Close database connection

After that, we use them to import data into our tables, and the bar plot of time cost (ms) are as below:



In conclusion, **the total insertion time of Java is still longer than that of using Python**, though having used batches with the import of some of the tables.

3.3.4 Import data with different data volumes

1. Data generation

Considering the difference of data volumes in data insertion process, we can already import data with volume of 100, 1000, 10000, 100000. And we still want to take a further look at how our files perform when importing data for larger volumes.

So, to test how well our data import scripts perform when the data volumes change, we write a file `generate data.py` to generate more random data entries based on the original data.

```
new_data = []
for _ in range(200000):  # generate 200,000 data
    new_entry = {}
    for key, value in original_data[0].items():
        if key in ['user', 'start_station', 'end_station']:
            new_entry[key] = random.choice([entry[key] for entry in original_data])
        elif key in ['start_time', 'end_time']:
            random_timestamp = random.randint(1577836800,
int(datetime.now().timestamp()))
            new_entry[key] = datetime.fromtimestamp(random_timestamp).strftime('%Y-
%m-%d %H:%M:%S')
        elif key == 'price':
            new_entry[key] = random.randint(1, 20)
        else:
            new_entry[key] = value
    new_data.append(new_entry)
```

After running the file, a new file `new_ride.json` with data volume of 200,000 is created.

2. Data insertion and comparison

To compare the efficiency of importing different volumes of data, we run the file `DataInsertion.py` and `Op_MT.py` (Multithreading) separately to test the time costs of inserting `ride.json` and `new_ride.json`. The result is:

Data name	Time cost before optimization (ms)	Time cost after optimization (ms)	Optimization rate
`ride.json` (volume: 100,000)	4679.38	2724.32	41.67%
`new_ride.json` (volume: 200,000)	9537.60	4125.78	56.74%

So, we can conclude that **our file can flexibly and efficiently import data with different volumes**. In addition, the larger the volume of data, the more efficient our optimization will be, which further proves that our optimization works well.