**CS307 Principles of Database Systems**

|  |
| --- |
| **Project 1 Report** |
| **Database for Shipping Company SUSTC** |

|  |  |
| --- | --- |
| **Student Name** | **Student ID** |
| Xuanyu Liu | 12110408 |
| Zexin Feng | 12110104 |

**Semester:** 2022 Autumn

**Lab Session:** Monday 7-8 (Class 2)

**Teacher:** Ran Cheng

**Lab Teacher:** Weiyu Wang

## Contributions

|  |  |  |
| --- | --- | --- |
| Member | Contribution | Ratio |
| Xuanyu Liu | 1. Design and Draw E-R Diagram 2. Compare DMBS with File I/O 3. Write the report | 50% |
| Zexin Feng | 1. Database Design 2. Data Import, and all operations for JDBC 3. Reformat the report | 50% |

## Task1: E-R Diagram

Based on the analysis to the data provided and the description about the data structure in project description, we designed following E-R diagram by using use Freedgo(<https://www.freedgo.com/>).

The diagram is compressed forcedly, so it is not clear enough. The original E-R diagram is attached as **ER-Diagram.png**.

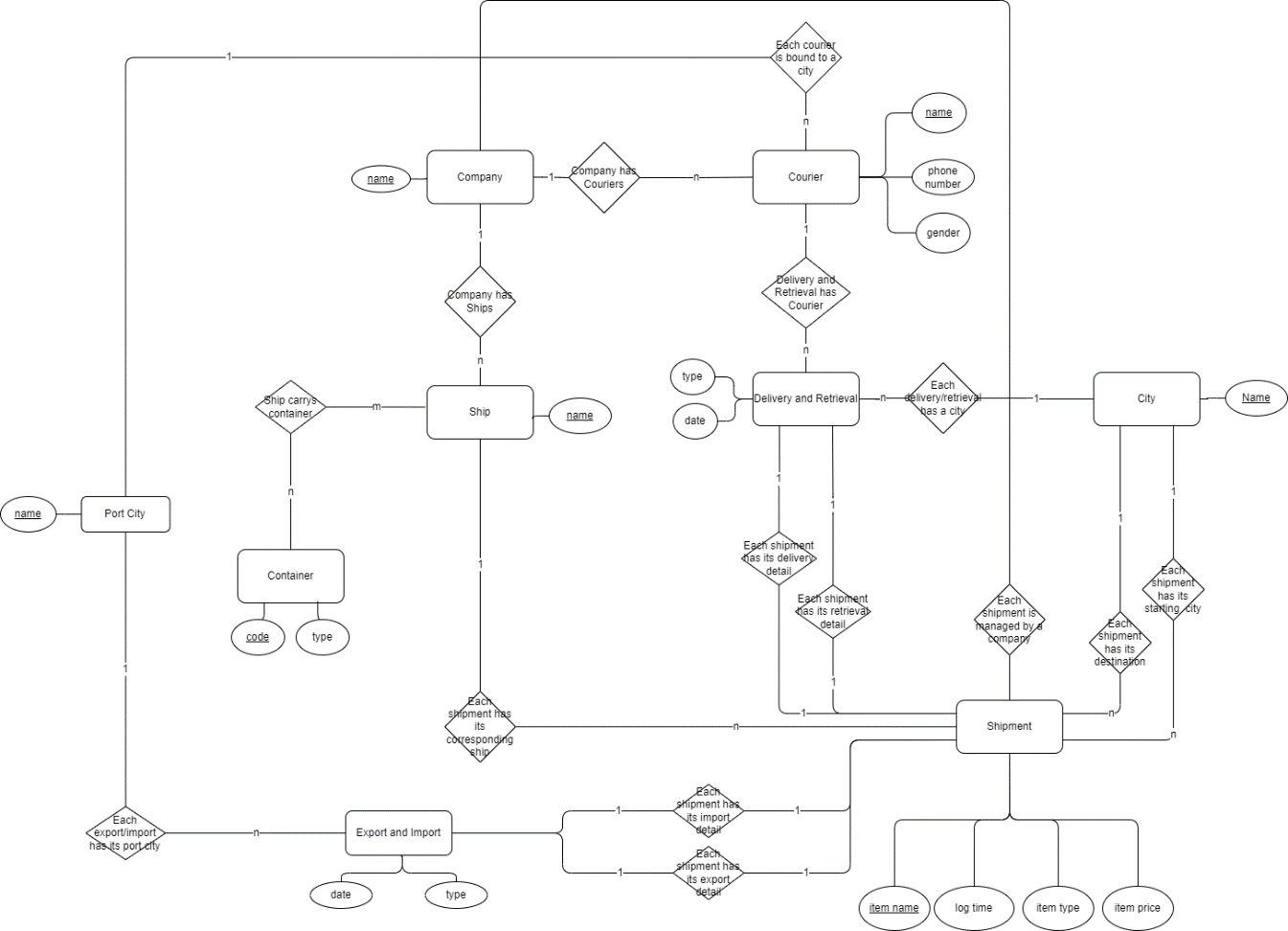


Figure 3.2.3‑1 E-R Diagram

## Task2: Database Design

### Database Diagram

Our database diagram exported from DataGrip is as follows:

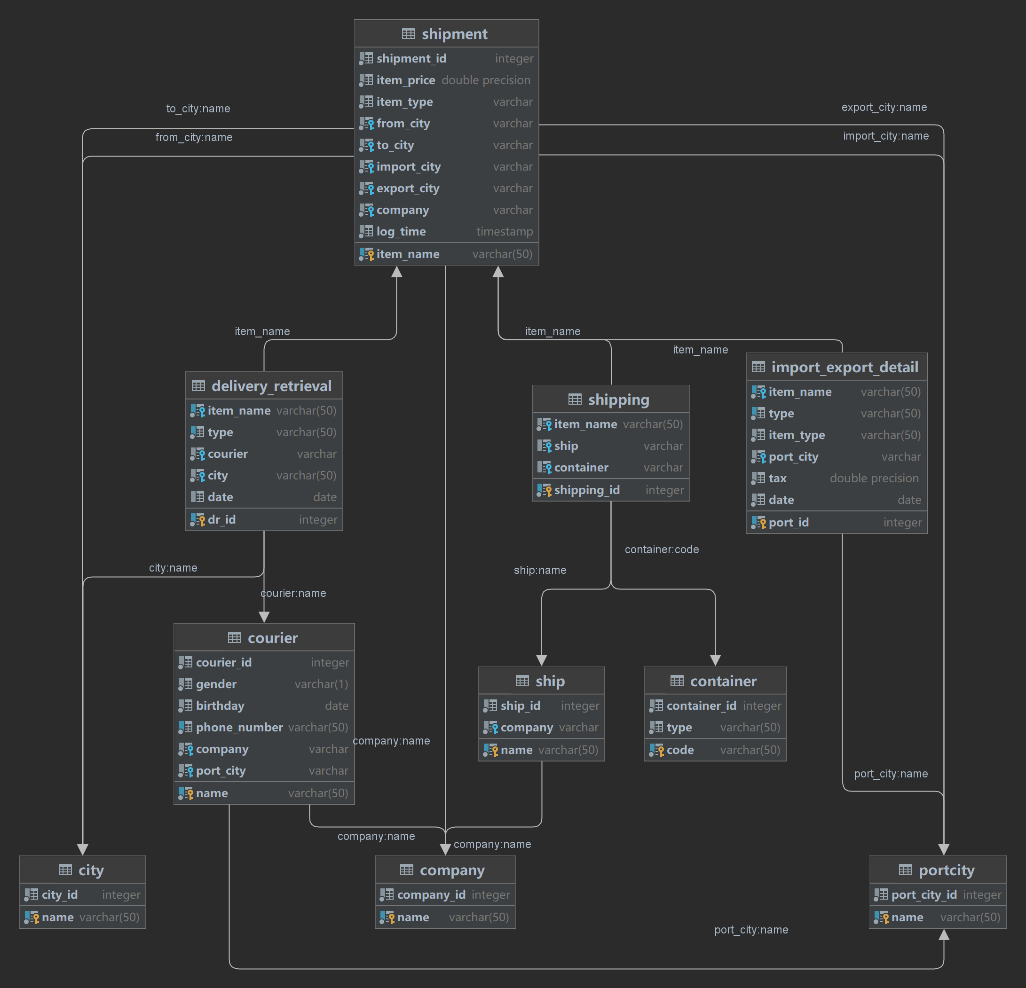


Figure 3.2.3‑1 E-R Diagram generated by DataGrip

The full DDL scripts are attached as **createTables.sql**.

### Database Structure Description

* + 1. **City Table**

**City** table is designed to store the cities where courier retrieve and deliver items. **Name** is the primary key.

* + 1. **PortCity Table**

**PortCity** table is designed to store the cities where export and import items. Similar to above, **Name** is the primary key.

#### Company Table

**Company** table is used to store the companies that manage the couriers, ships and items. **Name** columnis the primary key.

* + 1. **Courier Table**

1. **Courier** table is designed to store couriers who retrieve and deliver items.
2. **Gender, birthday, phone\_number** columnsare the basic information of a courier.
3. **Company** column is a foreign key referring to **Company** table, which represents that each courier works for a company.
4. **City** column is also a foreign key referring to **City** table, which means where the courier works. **Name** column is the primary key.
   * 1. **Ship Table**
5. **Ship** table stores the ships that carry items.
6. **Company** column is a foreign key referring to **Company** table, which represents that each ship belongs to a company.
   * 1. **Container Table**
7. **Container** table stores the containers information.
8. **Code** columnstores the identity of containers and is the primary key.
9. **Type** column stores the types of containers.
   * 1. **Shipment Table**

**Shipment** table stores all the items. **Please treat it the same as item table.**

1. **Item\_name, item\_type, item\_price** columns store the basic information of items, where **item\_name** column is the unique identity columns and also the primary key.
2. **From\_city, to\_city, company** columns are three foreign keys, the former two refer to **City** table, representing the starting city and destination of the item, the later refers to **Company** table, representing the company who manages the item.
3. **Log time** is the last update time of the item.
   * 1. **Shipping Table**

**Shipping** table is a connection table between **Shipment** tableand **Ship** table, **Shipment** table and **Container** table. So all columns of the tables are foreign keys.

1. **Ship** column refers to **Ship** table.
2. **Container** column refers to container table.
   * 1. **Delivery\_Retrieval**

**Delivery\_Retrieval** table is a connection table between **Shipment** table between **Courier** table.

1. **Date** column records when the item is delivered or retrieved.
2. **Type** column indicates that current record is delivery or retrieval.
   * 1. **Import\_Export\_Detail**

**Import\_Export\_Detail** table records the import or export detail of the item.

1. **Tax** column records the tax that the item needs to import or export.
2. **Type** column indicates that current record is import or export.
3. **Port\_city** column is a foreign key referring to **PortCity** Table, meaning where the item is imported or exported.
4. **Date** column records when the item is imported or exported.

## Task3: Data Import

### Environment Description

|  |  |
| --- | --- |
| **Hardware:**  **CPU:** Intel i7-1165G7 @ 2.80GHz  **RAM:** 16GB DDR4 4267Hz  **Disk:** Samsung SSD 512GB | **Software:**  **Operating System:** Windows 11  **File System:** NTFS  **DBMS:** PostgreSQL ver42.5.0[stable]  **Programing Language:** Java 17  **JDK Version:** 17  **PostgreSQL Version:** 14.2 |

**Other Tools:**

IntelliJ IDEA Ultimate 2022.2

DataGrip 2022.2

Microsoft Office 365

**Disk Performance:**

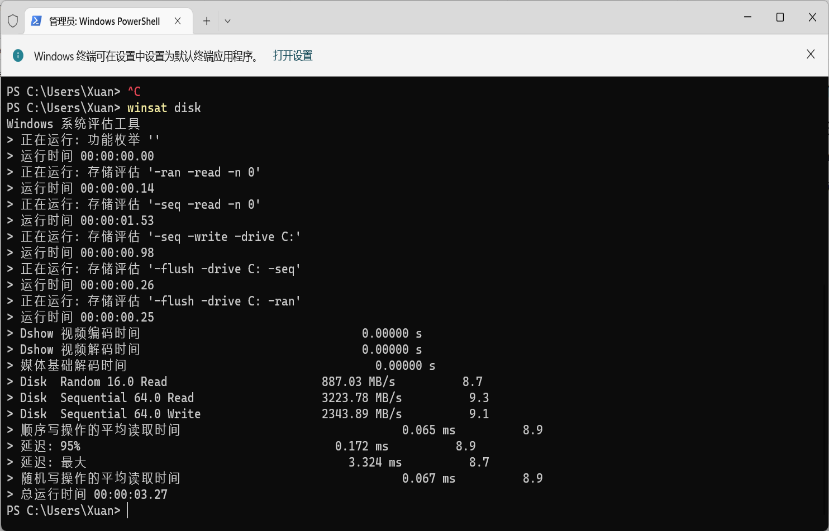


Figure 3.2.3‑1 Disk Performance

**Data:** shipment\_records.csv

### Different Ways of Importing

#### Using Java

We use Java with PostgreSQL driver to import data. The source code is attached as **Loader.java**. BufferedReader is used to read the csv file, the unique constrains is protected by PostgreSQL’s constrain. And we use System.currentTimeMillis() to count the executing time.

How To Run:

1. Copy shipment\_records.csv to the **data** directory, or change the path of source data file in **Main.java**
2. Import Libs: File >> Project Structure >> Project Settings >> Libraries >> “+”>> Java >> Select the folder “lib” >> OK >> OK
3. Set your database information and test your database connection at **JDBC.java**
4. New object **PostgreSQLTest jt = new PostgreSQLTest()** and run **jt.TestAllLoader(filePath)** in **Main.java**

Cautions:

Make sure you have tables created by **createTables.sql**, execute it in DataGrip.

#### Using Python

We use *Pandas* package to read csv file and *psycopg2* to connect database and execute SQL. The source code is attached at **PythonImporter/main.py**.

How To Run:

1. To install all the packages we need, please execute: pip3 install pandas psycopg2
2. Copy shipment\_records.csv to the data directory, or change the path at the bottom at **main.py**.
3. Set your database connection information at **getPostgreSQLConnection()** method.
4. Run **main.py**

Cautions:

1. Make sure you have tables created by **createTables.sql**, execute it in DataGrip
2. The only version of python we tested is 3.10. Using python with version lower than 3.10 to perform our script may lead to unknown error.

### Result and Analysis

The result of using Java to import data is as follows:

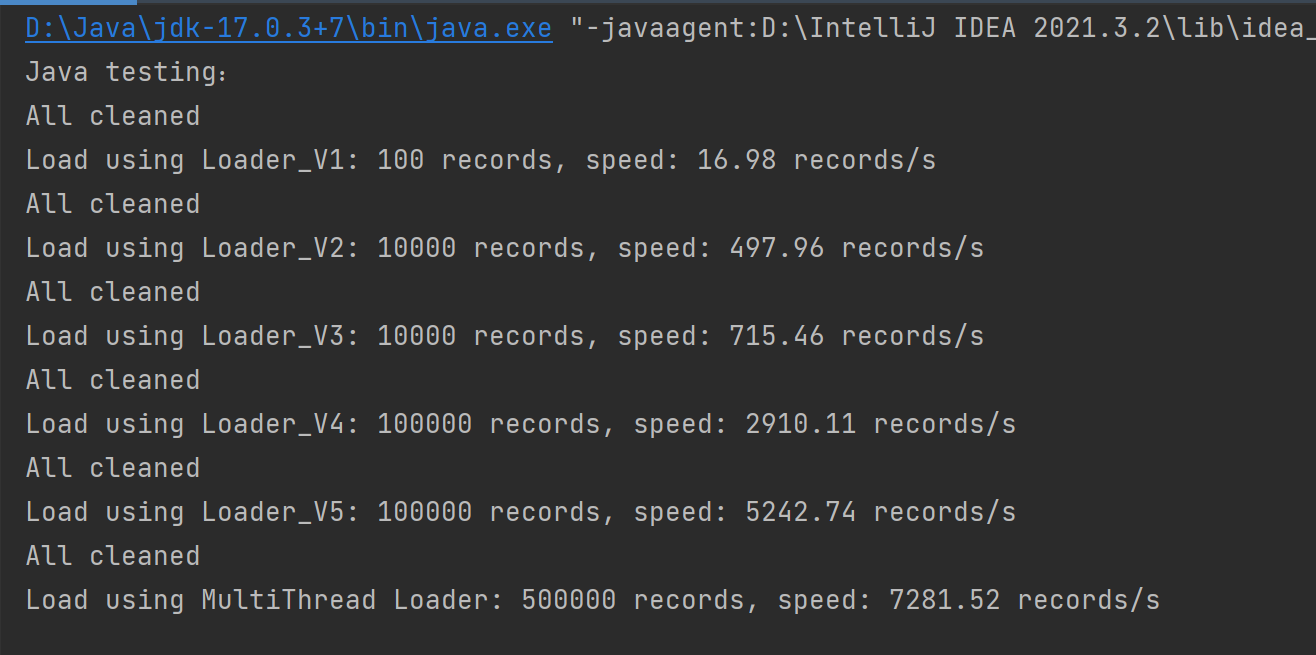


Figure 4.2.2‑1 Loader Test Result by Java JDBC

The result of using Python to import data is as follows:

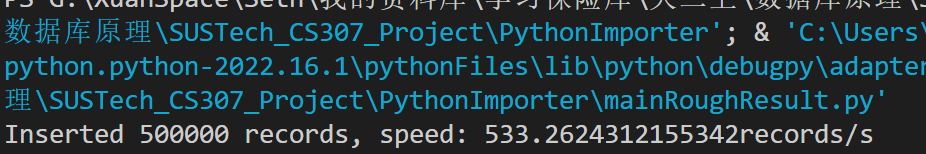


Figure 4.2.2‑2 Loader Test Result by Python

As we can see, the speed of using Python to import data is much slower than using Java. The possible reasons are as follow:

1. Java has higher processing speed than Python, meaning Java can perform more procedures than python with same time.
2. We disabled foreign key constraints and triggers while using Java to insert, but Python not. However, even compared with over second version of Java script, which didn’t disable such things and had a speed of 2869 records/s, Java is still much quicker than Python.
3. We use pure I/O APIs to read source data file while using Java, but third-party package (Pandas) while using Python, which may cause different speed of reading files.

### Script Optimization

We optimize our Java program and inserting SQLs to maximize the performance (importing speed):

**Version 1 (18.31 records/s) Initial (100 Records tested)**

Insert one by one, creating and closing new connection every step.

**Version 2 (503.4 records/s) +2648.32% (10,000 Records tested)**

Insert one by one, but using a single connection.

**Version 3 (725 records/s) +44.02% (10,000 Records tested)**

Insert by using Prepared SQL statement

**Version 4 (2923 records/s) +303.17% (100,000 Records tested)**

Insert by using batch, and disable the auto commit

**Version 5 (5235 records/s) +81.70% (100,000 Records tested)**

Disable the triggers in table then add it back after loading (You have to confirm that the data source is valid)

**Version 6(7362 records/s) +79.09% (500,000Records tested)**

Insert each table separately with 10 connections.

**Notes:**

The final version of the loader got 10 threads in total, but the test devices has only 8 threads, so we test it again on the other computer which performance is shown below, that could show the improvement accurately.

|  |  |
| --- | --- |
| **Hardware:**  **CPU:** AMD Ryzen 5800X 8C16T  **RAM:** 32GB DDR4 3600MHZ  **DISK:** SSD Hikvision C2000Pro 1024GB | **Software:**  **Operating System:** Windows 11  **File System:** NTFS  **DBMS:** PostgreSQL ver42.5.0[stable]  **JDK Version:** 18.0.2 |

**Result:**

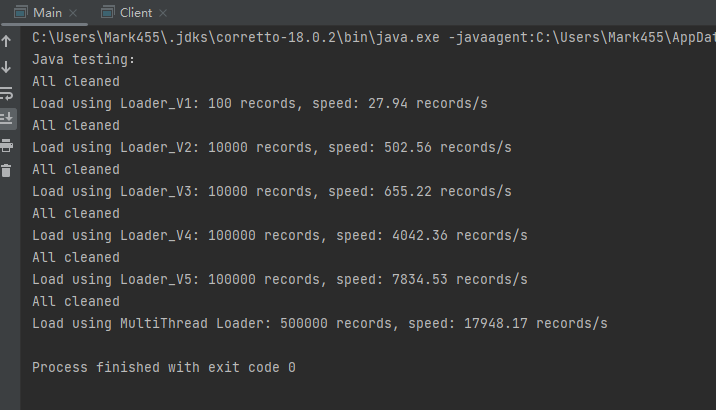


Figure 4.2.2‑1 Result by 16T CPU

## Task4: Compare DBMS with File I/O

### Environment Description

|  |  |
| --- | --- |
| **Hardware:**  **CPU:** Intel i7-1165G7 @ 2.80GHz  **RAM:** 16GB DDR4 4267Hz  **Disk:** Samsung SSD 512GB | **Software:**  **Operating System:** Windows 11  **File System:** NTFS  **DBMS:** PostgreSQL ver42.5.0[stable]  **Programing Language:** Java 17  **JDK Version:** 17  **PostgreSQL Version:** 14.2 |

**Other Tools:**

IntelliJ IDEA Ultimate 2022.2

DataGrip 2022.2

**Data:** shipment\_records.csv

### File I/O DB Description

We constructed a simple File DB with Gson and Java I/O APIs which have models sharing the generally same structure as PostgresSQL’s tables. What’s more, we implemented sub query with Java8 lambda property. Hence, we can deal with complex query even with our File DB. All the codes related to our File DB is placed at FileDB directory. What’s more, we also implemented foreign key check and primary key check (avoiding duplicated data) while inserting data.

### Table Structure

Because the models of our File DB share the same structure with tables of PostgreSQL, so there is no need to create new tables for testing. That is to say, all tables we used here are the same as what we declared at **Task 2**, and the data we used is also from shipment\_records.csv.

### Method Introduction

On account of the extremely poor performance of our File DB, we only imported first 10000 records of the shipment\_records.csv while testing Update, Delete and Select.

#### Insert

Insert the first 5000 rows of shipment\_records.csv.

#### Update

First, import 10000 rows of shipment\_records.csv. Then, update first 100, 500, 1000 or 5000 rows of **Courier** table with the same information.

#### Delete

First, import 10000 rows of shipment\_records.csv. Then, delete first 100, 500 or 1000 rows of **Shipment**. Attention: The corresponding records in **Retrieval\_Delivery, Import\_Export\_Detail, Shipping** are deleted cascade.

#### Select

First, import 10000 rows of shipment\_records.csv. Then, query all columns of first 100, 500 or 1000 rows of **Shipment** table.

### Script Description

#### PostgreSQL

* **Project File Structure:**
* **JDBC.java**

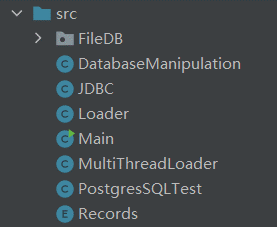


Figure 5.5.1‑1 The structure of files

This class is used to get new connections. The **information of the connection** should be written in here. Then getNewCon() will return a Connection Object.

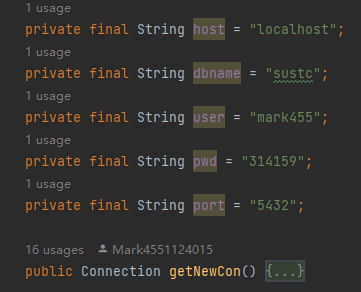


Figure 5.5.1‑2 Info for connection and get connection

* **Records.java**

This one got only enum in it, which contains different type of records. This will be easier to distinguish the tables and data. Such as: city, shipping, ship etc.

* DatabaseManipulation.java:

This is the main class to manipulate the database. Which contains a bunch of functions, the usage should be written in the source file already.

* **Loader.java**

Loader.java is the main function that stored every single version of the loader we made (Sigle thread). You can use Loader before Version 4 to load the data which you can’t confirmed that is valid. The function V5 should be the fastest (single threading)

* **MultiThreadLoader.java**

We have tested the multi connections to insert in same tables, but it won’t work. Because the Lock set in SQL (Dead Lock). But we can separate the full table loading tasks into single table loading tasks. So, we set 10 threads to load the data from .csv.

* **PostgreSQLTest.java and Main.java**

We put some preset testing function in PostgreSQLTest.java, you can just run it in Main.java. Main.java should has some tips for you to help with.

#### File I/O DB

The structure of File DB is as follow:

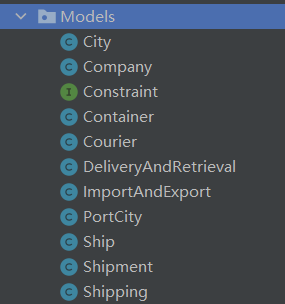
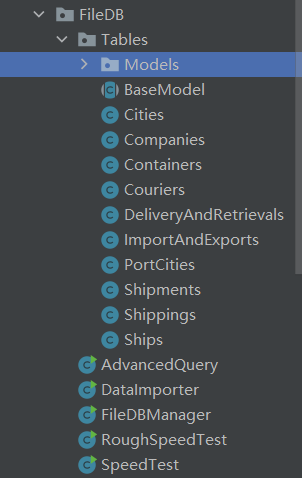


Figure 5.5.2‑1 Project Structure of File I/O

* **FileDB/FileDBManager.java**

The entrance of our File DB. You can get corresponding database, reload database, or truncate database through the APIs it provides.

* **FileDB/DataImporter.java**

The data importer of File DB. You can specific the source data path and the max number to import at **main()** method.

* **FileDB/RoughSpeedTest.java FileDB/SpeedTest.java**

All in one speed test scripts for File DB. **RoughSpeedTest.java** will only output the speed of all queries, but **SpeedTest.java** will output the time that each query costs.

* **FileDB/AdvancedQuery.java**

The all in one test script for **Task 4 Advanced Requirements.** Details will be introduced at later parts.

* **FileDB/Tables/\*.java**

Each class represents a table, storing multiple records. Each class provides with the interface of insert, update (some may not), delete and select.

**Note:** The tables cannot be instantiated directly, because it won’t initialize itself automatically. If you want to get a table, you should get **FileDBManager** instance first, then get the table through the manager.

* **FileDB/Tables/Models/\*.java**

Each class represents a model, which stores only one record. The attributes of each model have the same meaning of columns in PostgreSQL. Each model belongs to its corresponding table.

Here we use **FileDB/RoughSpeedTest.java** to test the speed of File DB, which can be executed directly. Similarly, adjust the params in this file and run it, then can we get the result of next section.

### Result and Analysis

#### PostgreSQL Test Result

Different range of data operation test is shown below, and we also got a diagram to show those result clearly in the later of this page:

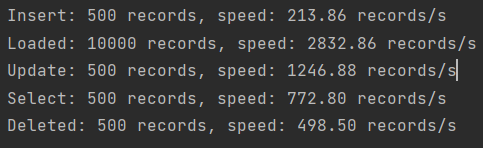


Figure 5.6.1‑2 Result2

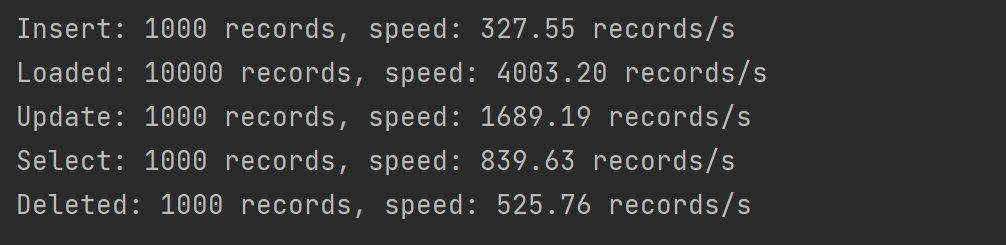


Figure 5.6.1‑3 Result 3

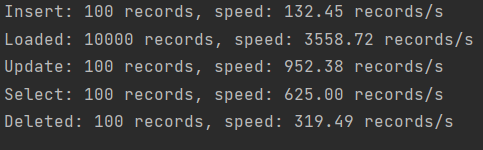


Figure 5.6.1‑1: Result 1

**Tips:** *Loaded* means the speed of importing data in bulk, not the speed of insertion.

#### File I/O Test Result

Figure 5.6.1‑3 File I/O 1000 Records

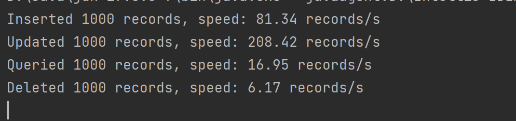
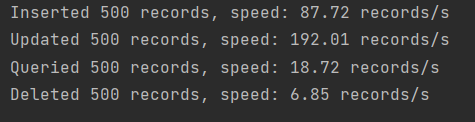


Figure 5.6.2‑1 File I/O 500 records

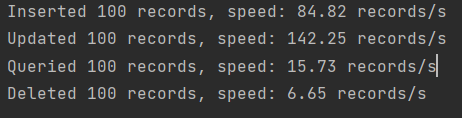


Figure 5.6.2‑1 File I/O 100 records

**Tips:** *Queried* has the same meaning of *select* in former figures.

#### Comparative Diagrams

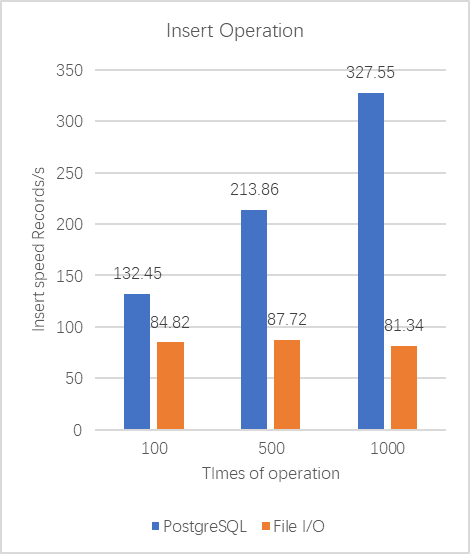


Figure 5.6.3‑1 Insert operation

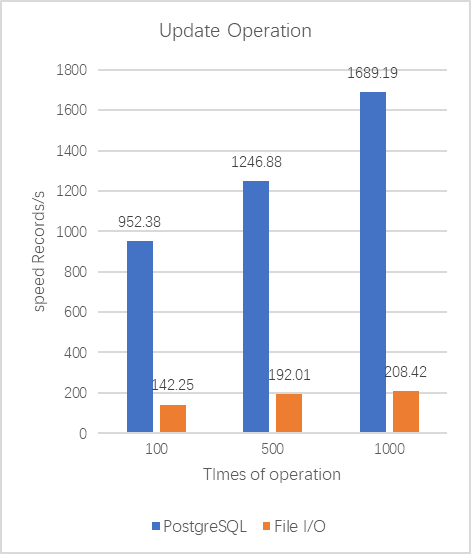


Figure 5.6.3‑2 Update operation



Figure 5.6.3‑3 Comparison Between Loaders in second computer

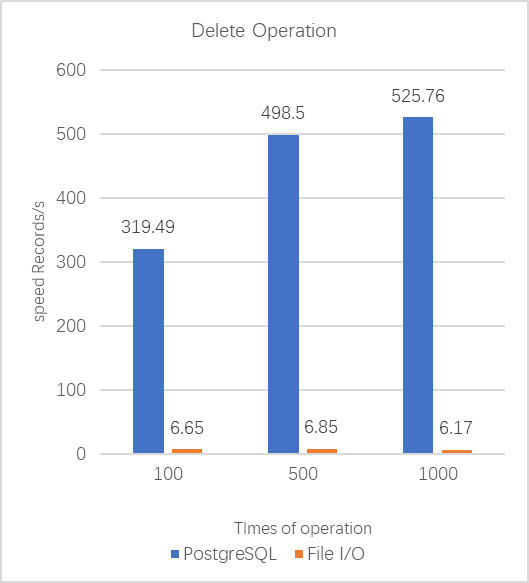


Figure 5.6.3‑4 Delete operation

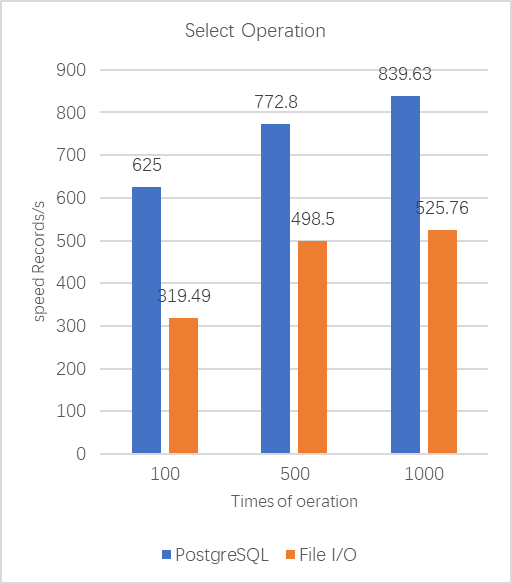


Figure 5.6.3‑5 Select operation

#### Conclusion

As we can see, PostgreSQL has an absolutely faster speed than our File DB, no matter with respect to insert, update, select or delete. As the number of operations increases, PostgreSQL even shows a better performance, but File DB only has a slight improvement. We thought the possible reason may as follows:

1. PostgreSQL needs time to preheat at the first query. So as the operation number increases, the time of preheat has a smaller impact.
2. File DB has a poor performance of sub-query. While testing delete operation, we not only delete the records of **Shipment**, but also all the related records, which decrease the delete speed of File DB greatly.

### Advanced Tasks

We set functions to query the served time of each container, you can have the source code in the attachment. We only put the result and usage here.

#### Advanced Search 1

* **PostgreSQL：**

The function is **QueryServedContainer(int expired). Only argument** in the function is the expired date, and the function will print the container that served more than this value. First it will print the types we got in database. Then you can **type in** the type you want to query. The result will be shown below.

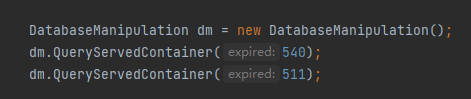


Figure 5.7.1‑1 Usage of the Function

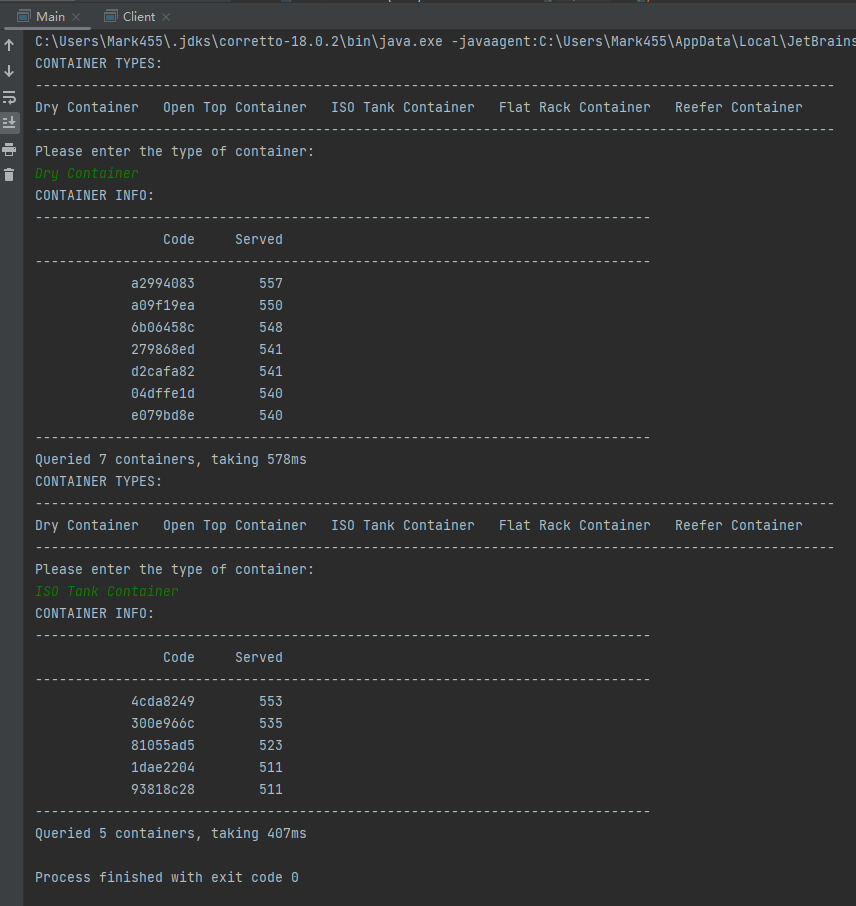


Figure 5.7.1‑2 Result of containers

* **File DB:**
* Execute the function **queryContainersWithWorkingTime(int minDays, String type)** at **FileDB/AdvancedQuery.java**, then the program will output the containers with specific **type and** working time more than **minDays.**
* If you want to output all the containers with its types and working time, you can execute **queryContainersWithWorkingTime()** in the same file.

#### Advanced Search 2

* **PostgreSQL:**

The function is **QueryBestCourier()**, there are **three arguments** in this function. First one is the **city**, second one is **company name**, third one is **top number of couriers.** This function will print top of the courier for a company in particular city.

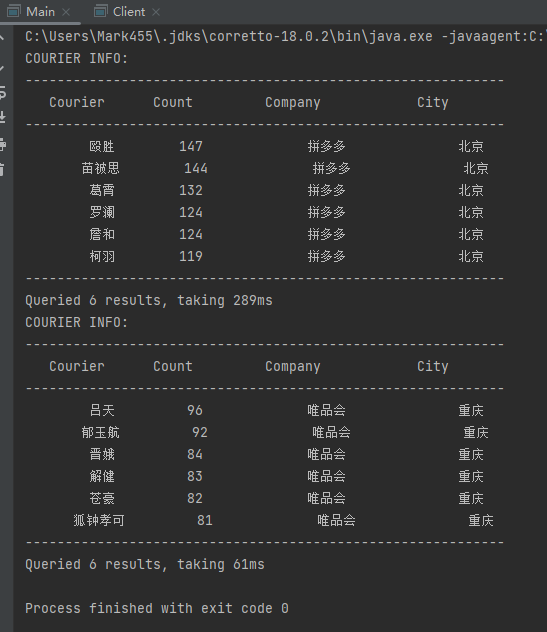


Figure 5.7.2‑1 Result of good couriers

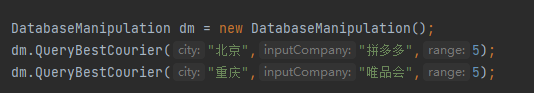


Figure 5.7.2‑2 The usage of Function

* **File DB:**
* Execute the function **queryBestCourier(String company, String city)** at **FileDB/AdvancedQuery.java**, then the program will output the couriers who delivery and retrieve most for the specific **company** in the specific **city** respectively.
* If you want to output all the retrieval and delivery couriers who collect/send the greatest number of items for each company in each city, you can execute **queryBestCourier()** in the same file.

#### Advanced Search 3

* **PostgreSQL：**

The function is **QueryBestPort()**. There are **two arguments**. First one is the **type of items**. Second one is **“export”** or **“import”**. Then the function will print the city that has the **lowest tax rate**, which means less cost on export or import.

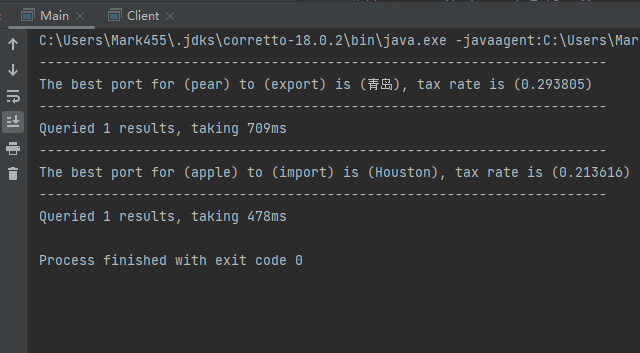


Figure 5.7.3‑1 The result of best import/export city

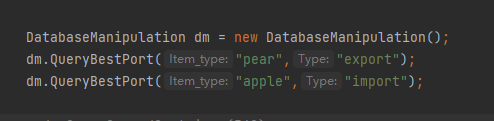


Figure 5.7.3‑2 The usage of function

* **File DB:**
* Execute the function **queryBestExportCity(String itemType)** at **FileDB/AdvancedQuery.java**, then the program will output the port city which has the lowest export tax rate with respect to the specific **itemType.**
* If you want to get lowest export tax rate city for all items, you can execute **queryBestExportCity()** in the same file.

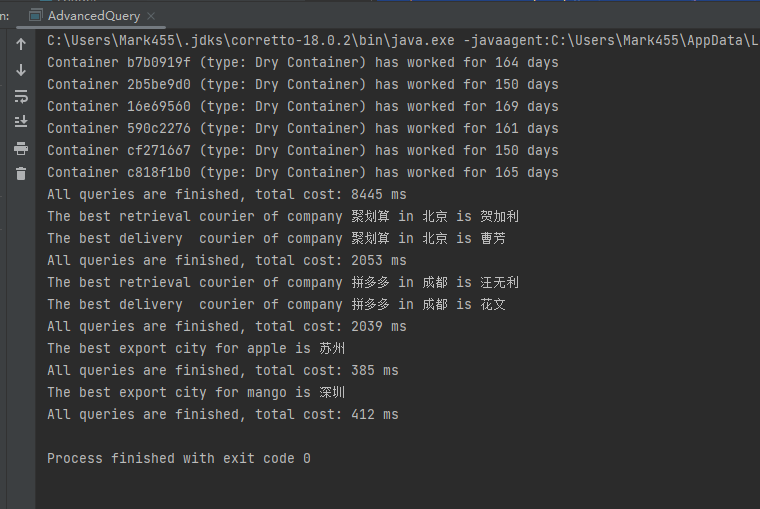


Figure 5.7.3‑3 The usage and result of File DB Query

#### High Concurrency

We tested high currency of our DBMS, it works fluently. The selection test in **PostgreSQLTest.java** designed to do operations in new connection, and it works well when we set selection number to 50,000, which can process 2073 times of select query. And all the result was printed out correctly in **log.txt** for you to check.

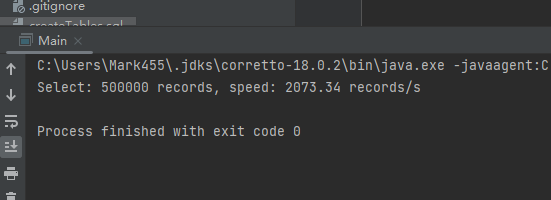


Figure 5.7.4‑1 HighCurrency

#### Performance Comparison With Other SQL

|  |  |
| --- | --- |
| **Hardware:**  **CPU:** Intel i7-1165G7 @ 2.80GHz  **RAM:** 16GB DDR4 4267Hz  **Disk:** Samsung SSD 512GB | **Software:**  **Operating System:** Windows 11  **File System:** NTFS  **DBMS:** PostgreSQL ver42.5.0[stable]  **Programing Language:** Java 17  **JDK Version:** 17  **PostgreSQL Version:** 14.2  **MySQL Version:** 5.1.72  **MariaDB Version:** 10.6.10 |

As we can see, PostgreSQL has a relatively better performance with respect to MySQL and MariaDB. With the number of importing data increase, PostgreSQL shows a better performance but MySQL and MariaDB not.

One thing to note is that, the version of MySQL we selected is released at 9 years ago, and the version of MariaDB we select is released at last month. However, we are surprised to find that they don’t have much difference on the speed of insertion. Maybe it is our SQL not friendly for MySQL and MariaDB, but only PostgreSQL.

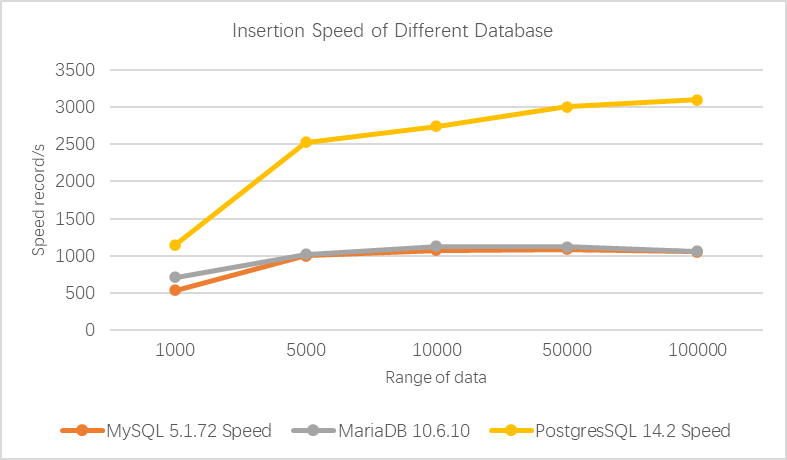


Figure 5.7.5‑1 Comparison between SQLs

#### Performance Comparison Between Different Operating System

|  |  |
| --- | --- |
| **Hardware:**  **CPU:** Apple Silicon M1  **RAM:** 16GB LPDDR4X -4266Mhz  **Disk:** 256GB | **Software:**  **Operating System:** macOS 12.6  **File System:** APFS  **DBMS:** PostgreSQL ver42.5.0[stable]  **Programing Language:** Java 18.0.2  **JDK Version:** corretto**-**18.0.2  **PostgreSQL Version:** 14.2 |

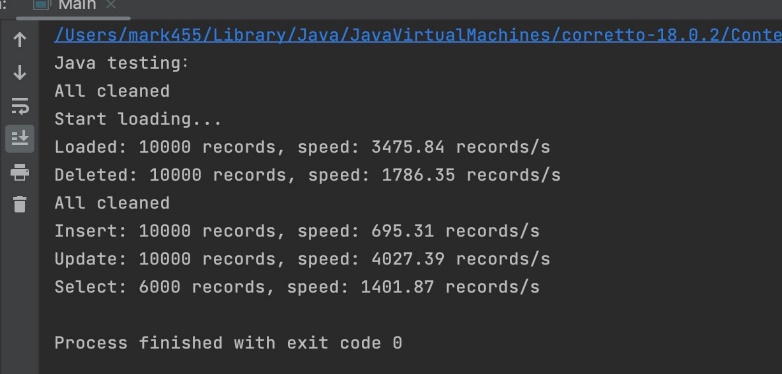


Figure 5.7.6‑1 Result Tested By apple M1 macOS

And here is the comparison:

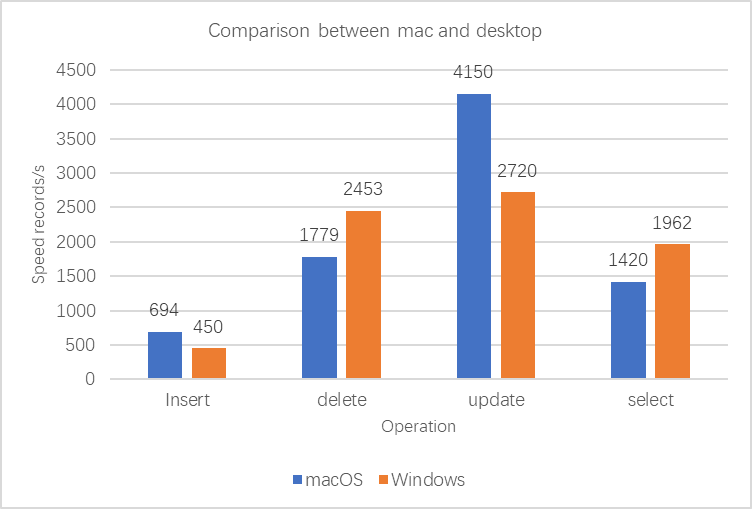


Figure 5.7.6‑2

The speed is close, but we do think macOS device has better performance in writing data into the disk, so it shows faster Update and Insert