# CS307 Project Report (Midterm)

## Part 1. Group Info and Contribution

| Name | SID | Specific Work | Contribution |
|---|---|---|---|
| 金肇轩 | 11911413 | **Task 2**-Write scripts to preprocess the json file<br>**Task 2**-Process the prerequisite (duplicate removal, post-order)<br>**Task 3**-Implement the C++ program<br>**Task 3**-Do the experiments | 40 |
| 孙永康 | 11911409 | **Task 1**-Table Design<br>**Task 2**-Accelerate the data import | 30 |
| 胡鸿飞 | 11911412 | **Task 2**-Build the structure of data import<br>**Task 3**-Visualize the test data | 30 |

## Part 2. Task 1 Implementation & Introduction

### 1. Table Diagram

Figure 2-1 shows the UML diagram of the tables generated by `DataGrip`.

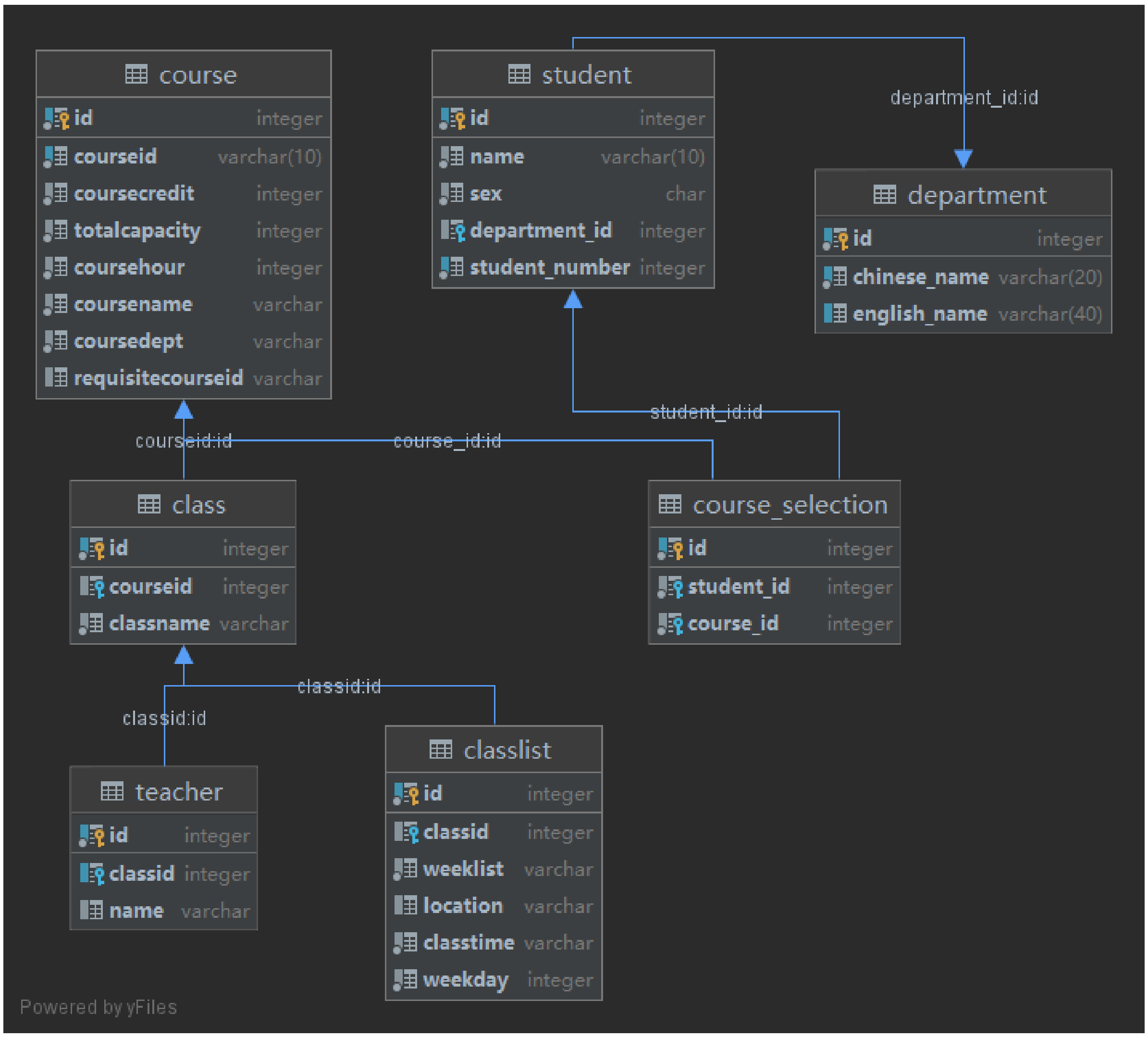


**Figure 2-1. UML diagram of the tables**

## 2. Introduction

There are 7 tables in total:

The four tables `course`, `class`, `teacher`, `classlist` are used to store the data in `course_info.json`. The other three tables `student`, `department`, `course_selection` are used to store the data in `select_course.csv`.

In `course_info.json`, each entry is the information of one class of one course. We extracted the information of the courses into the table `course`, and store the information of the classes in a one-to-many relation table `class`. We store the relation between teachers and classes, class lists and classes in the same way.

In `select_course.csv`, there are a one-to-many relation between student and department, and a many-to-many relation between student and course selection, So we divided them into three tables, with table `course_selection` having a foreign key with the table `course`.

# Part 3. Task 2 Implementation & Analysis

## 1. Preprocess the data in `course_info.json`

The script is `process_json.py`, some of the codes of processing `prerequisite` field are shown in Code 3-1.

```python
if not re.match("\\A[A-Za-z\\s]*\\Z", str(i['prerequisite'])):
    i['prerequisite'] = re.sub("\\((?P<inner>.)\\)", replace, str(i['prerequisite']))
    i['prerequisite'] = re.sub("\\A\\s", "", str(i['prerequisite']))
    i['prerequisite'] = re.sub("\\s(?P<lev>[ABCDI])", iden, str(i['prerequisite']))
    i['prerequisite'] = re.sub("\\(", "(|", re.sub("\\)", "|)", i['prerequisite']))
```

**Code 3-1. Preprocess Python script (Part)**

### (1) Make data more regular

First, we analyzed `course_info.json` and solved the following irregular data problems.

- **Space between Chinese course names and grades**

  **Description**

  Some of the Chinese courses have space between course names and grades, such as

  ```
  大学物理 B(下)
  化学原理 A
  ```

  Since we will split the prerequisite by space, this will cause difficulties to the following process.

  **Solution**

  We found there are only `A`, `B`, `C`, `D`, `I`, `II`, `III` in course grades, so we used regular expression `\s[ABCDI]` to match the spaces and delete them.

- **Half-angle brackets in Chinese course names**

  **Description**

  There are some half-angle brackets in Chinese course names, such as

  ```
  大学物理 B(下)
  ```

  Since we will judge the priority by the half-angle brackets, this will cause difficulties to the following process.

  **Solution**

  We found that this case will only happen in parts like `(上)`, `(下)`, so we used regular expression `\(.\)` to match and replace them.

- **Spaces and Tabs in data**

  **Description**

  This kind of problem is found in fields `teacher`, `courseName`, `className` and `location`.

  In these fields, there might be Spaces and Tabs (mainly at the beginning), such as

  ```
  \tHisao Ishibuchi
   全球生物多样性保护
  ```

  This will make the data in database irregular and have unnecessary white spaces.

  **Solution**

  For fields which are not supposed to have white spaces, like `className` and `location`, we can simply match the white spaces by regular expression `\s+` and delete them.

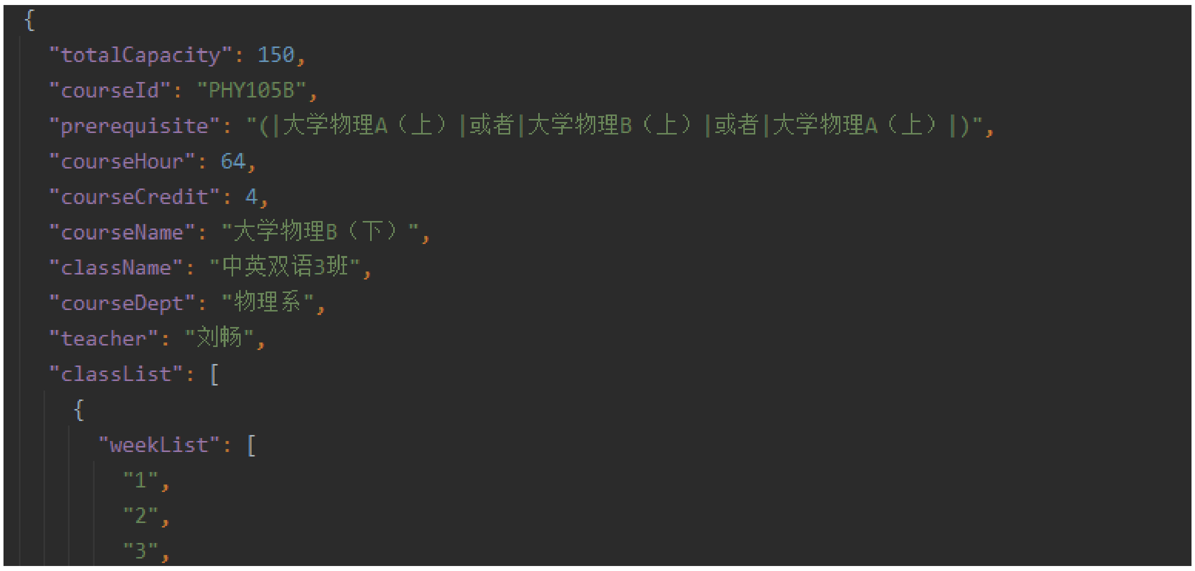
  For fields that might have white spaces itself, we used regular expression `\A\s` to match the white spaces at the beginning of the filed and delete them.

## (2) Change the separator

Though we mentioned below that we will use space to split the `prerequisite` field, we decided to change the separator to `|` at last because there are some English course names that have spaces.

We found that all of the `prerequisite` string that have English course names will only have one or less course in prerequisite, so we used regular expression `\A[A-Za-z\s]*\Z` to match English course names and exclude them. After that, we used `\s` to find all the spaces in `prerequisite` field and changed them into `|`.

We also match the brackets by using regular expression `\(` and `\)` and added the separator, so they will become `(|` and `|)`, which is more convenient to be split.



```json
{
  "totalCapacity": 150,
  "courseId": "PHY105B",
  "prerequisite": "(|大学物理A（上）|或者|大学物理B（上）|或者|大学物理A（上）|)",
  "courseHour": 64,
  "courseCredit": 4,
  "courseName": "大学物理B（下）",
  "className": "中英双语3班",
  "courseDept": "物理系",
  "teacher": "刘畅",
  "classList": [
    {
      "weekList": [
        "1",
        "2",
        "3",
```

**Figure 3-2. Preprocessd JSON file**

The preprocessed JSON file is shown in Figure 3-2.

## 2. Code structure of data importing

The basic structure of our data_importing scrips is constructed by 3 steps.

### (1) Database connection part

Using **JDBC** to connect with database, we need two things : Driver class and variables like `URL` `User` and `Password`, then we will get a connection object which can handle all operations we do to the database.

Consider the safety factor we catch the **Exception** the database send back, and do something about it like **role back** and **close database connection**.

- **The functions of open database**

```java
private static void OpenDB() {
  String url = "jdbc:postgresql://" + host + "/" + dbName;

  // 1. Load the Driver class, and the Driver class objects will automatically be registered with the DriverManager class
  try {
    Class.forName("org.postgresql.Driver");
  } catch (Exception e) {
    System.err.println("Cannot find the Postgres driver. Check CLASSPATH.");
    System.exit(1);
  }

  // 2. Connect to the database, returns the connection object
  try {
    conn = DriverManager.getConnection(url, user, password);
    System.out.println("Successfully connected to the database " + dbName + " as " user);
    conn.setAutoCommit(false);
  } catch (SQLException e) {
    System.err.println("Database connection failed");
```

```
      System.err.println(e.getMessage());
      System.exit(1);
    }
  }
```

- **The functions of close database**

```
  private static void CloseDB() {
    if (conn != null) {
      try {
        // If there is an unclosed Prepared_Statement commit, close it
        if (stmt != null) {
          stmt.close();
          stmt = null;
        }
        //Commit the contents of the Connection and close the Connection
        conn.commit();
        conn.close();
        conn = null;
        System.out.println("Successfully close the database " + dbName +
" as " + user);
      } catch (Exception e) {
        System.err.println("Close database failed");
        System.err.println(e.getMessage());
        System.exit(1);
      }
    }
  }
```

**Code 3-3. Connection to databse**

## (2) JSON file connection part

Using the **GSON** lib to read from JSON file

```
  //Load the information from the JSON file into the JSON Array
  public static void loadFromJson(String fileLocation) {
    try {
      //Create a new module to parse the JSON file
      JsonParser parser = new JsonParser();  //Create a JSON parser
      BufferedReader in = new BufferedReader(
          new InputStreamReader(new FileInputStream(fileLocation),
StandardCharsets.UTF_8),
          50 * 1024 * 1024); //Set the buffer and charset
      jsonArray = (JsonArray) parser.parse(in);  //Create the JSON Array
object
    } catch (FileNotFoundException e) {
      System.err.println("No JSON file found");
      System.err.println(e.getMessage());
      System.exit(1);
    }
  }
```

**Code 3-4. Read json files**

## (3) Main data importing part

We divide the whole things into 3 steps.

- **Test connection, clear the table and make "prepared statement"**

```java
try {
  System.out.println("--START CLEAR TABLE COURSE--");
  OpenDB();
  if (conn != null) {
    Statement stmt0 = conn.createStatement();
    stmt0.execute("TRUNCATE TABLE course RESTART IDENTITY CASCADE");
    stmt0.close();
  }
  CloseDB();
  System.out.println("--FINISH CLEAR TABLE COURSE--\n");
} catch (SQLException e) {
  System.err.println("TEST Database connection failed");
  System.err.println(e.getMessage());
  System.exit(1);
}
```

**Code 3-5-a. Test connection and clear the table**

```java
  //Prepare the jdbc::PreparedStatement and the corresponding sql statement
  String sql_addCourseIfNotExist =
      "INSERT INTO course(id, totalCapacity, courseId, courseHour, courseCredit, courseName, courseDept) VALUES(?,?,?,?,?,?,?)" + "ON conflict(courseId)  DO NOTHING;";
  create_PS(sql_addCourseIfNotExist);
```

```java
  private static void create_PS(String sql) {
    try {
      if (conn != null) {
        stmt = conn.prepareStatement(sql);
      } else {
        System.err.println("Connection unaccomplished");
      }
    } catch (SQLException e) {
      System.err.println("Insert statement failed");
      System.err.println(e.getMessage());
      CloseDB();
      System.exit(1);
    }
  }
```

**Code 3-5-b. make prepared statement**

- **Loop reads from JSON file and load the information into the corresponding object**

  Although there is a time cost to re-depositing the information into the object， but we thought it would make our code more modular, which would help speed up and test later on.

```java
// Read information from course_info.json and use it to create course_info objects
course_info course_info = new course_info();
// Import the information in course_info.json into course_info objects
jsonToCourse(i, course_info, cnt);
```

Code 3-6 takes `Course` as an example, showed how to parse a `JSON` file.

```java
private static void jsonToCourse(int index, course_info course_info, int id) {
  //Gets one of the target ordines of the entire file's JSONArray as a JSONObject
  JsonObject jsonOBJ_courseArray = jsonArray.get(index).getAsJsonObject();
  //Extract information from JSONObjects and store it into corresponding objects
  course_info.setId(id);
  course_info.setCourseId(jsonOBJ_courseArray.get("courseId").getAsString());
  course_info.setCourseCredit(jsonOBJ_courseArray.get("courseCredit").getAsInt());
  course_info.setTotalCapacity(jsonOBJ_courseArray.get("totalCapacity").getAsInt());
  course_info.setCourseHour(jsonOBJ_courseArray.get("courseHour").getAsInt());
  course_info.setCourseDept(jsonOBJ_courseArray.get("courseDept").getAsString());
  course_info.setCourseName(jsonOBJ_courseArray.get("courseName").getAsString());
}
```

**Code 3-6. read from JSON files**

- **Load data into database**

  Loads the corresponding information to **Prepared Statement**, and then it loads into **Connection**

```java
courseToDatabase(course_info);
```

In this section, we used `addBatch()` functions, which we will talk about it later in **Part 3-4. Optimize the speed of importing**.

```java
  private static void courseToDatabase(course_info course_info) throws SQLException {
    PreparedStatement ps_addCourse = stmt;
    ps_addCourse.setInt(1, course_info.getId());
    ps_addCourse.setInt(2, course_info.getTotalCapacity());
    ps_addCourse.setString(3, course_info.getCourseId());
    ps_addCourse.setInt(4, course_info.getCourseHour());
    ps_addCourse.setInt(5, course_info.getCourseCredit());
    ps_addCourse.setString(6, course_info.getCourseName());
    ps_addCourse.setString(7, course_info.getCourseDept());
    ps_addCourse.addBatch();
  }
```

Flush the batch and execute the code in it.

```
stmt.executeBatch();
stmt.clearBatch();
```

**Code 3-7. load it into database**

## 3. Process of prerequisite

### (1) Implementation

We processed the prerequisite expression in two steps:

1. Change the string expression into an **integer** array.
2. Rewrite the expression in post-order.

When we get a prerequisite expression, we first split it into an array using the separator `|`. Then we go through the array.

- If we find a course name, then we query in the database to find the corresponding course id.
    - If we find the id, replace the course name with id.
    - If we don't find the id, drop it.
- If we find `(`, `)`, `并且`, `或者`, we will replace `(` with `-3`, replace `)` with `-4`, replace `或者` with `-1` and replace `并且` with `-2`.

We will delete the duplicate course names in the same time.

```
int[] stack = new int[100];
int top = 0, temp_int, ptr = 0;
for (int i = 0; i < t; i++) {
  switch (fin_arr[i]) {
    case -1:
    case -2:
    case -3:
      stack[top++] = fin_arr[i];
      break;
    case -4:
      while (true) {
        temp_int = stack[--top];
        if (temp_int == -3) {
          break;
        } else {
          fin_arr[ptr++] = temp_int;
        }
      }
      break;
    case -5:
      break;
    default:
      fin_arr[ptr++] = fin_arr[i];
      break;
  }
}
while (top >= 1) {
  fin_arr[ptr++] = stack[--top];
}
```

**Code 3-8. Transform into post-order**

Then we used **stack** to transform the expression into a post-order expression. Go through the array again.

- If we meet a course id (positive), put it into the expression directly.
- If we meet `-3`, which means `(`, push it into stack.
- If we meet `-4`, which means `)`, pop the stack until meet `-3`. Don't put `-3` into expression. Drop it.
- If we meet `-1` or `-2`, just push it into stack.

When we go through all the elements in the array, pop all the items left in the stack to the end of the expression.

Code 3-2 shows the Java code of generating the post-order expression.

## (2) Advantages

This implementation has four advantages.

- **High Compatibility**

  This method of transforming into post-order can process any form of Boolean expressions and store it into database, no matter it's a SOP, POS, or a complicated expression.

- **One row in table**

  This method only need one row to store in database, and it can allow column `prerequisite` to be directly stored in table `course` instead of storing it in a new table and adding foreign keys.

  This will reduce the complexity of query.

- **Easy to resolve**

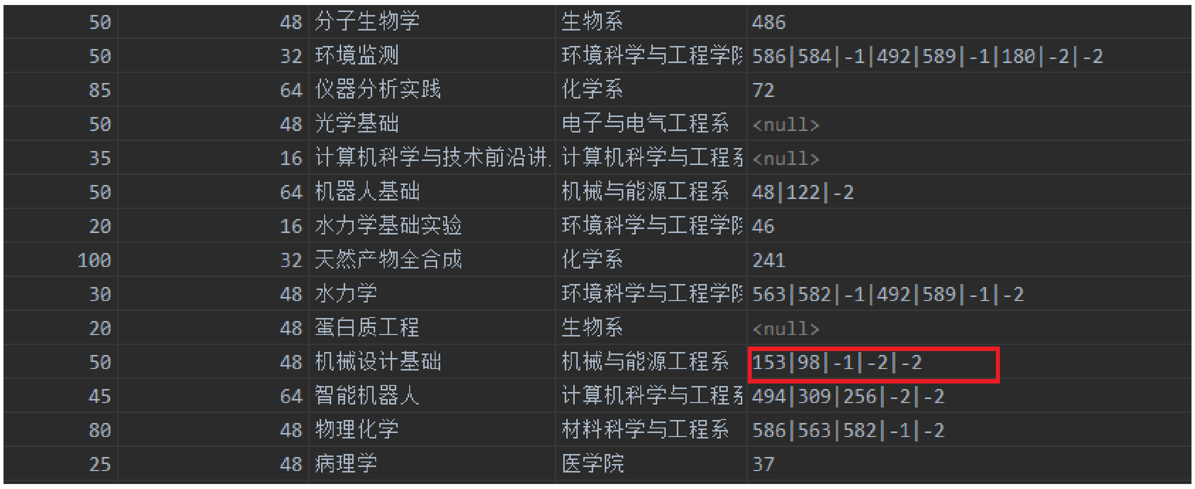
  Post-order expressions is a computer-friendly form of expression. Thus, it's easy to resolve when we use it in the future.

- **Optimizable**

  When we decide if someone satisfies the prerequisite, we need to connect to the database and calculate the expression while querying.

  If one expression have `n` courses, the number we query the database can be **equal or less than** `n` after optimizing.

## (3) Problems



| | | | | |
|---|---|---|---|---|
| 50 | 48 | 分子生物学 | 生物系 | 486 |
| 50 | 32 | 环境监测 | 环境科学与工程学 | 586\|584\|-1\|492\|589\|-1\|180\|-2\|-2 |
| 85 | 64 | 仪器分析实践 | 化学系 | 72 |
| 50 | 48 | 光学基础 | 电子与电气工程系 | <null> |
| 35 | 16 | 计算机科学与技术前沿讲. | 计算机科学与工程 | <null> |
| 50 | 64 | 机器人基础 | 机械与能源工程系 | 48\|122\|-2 |
| 20 | 16 | 水力学基础实验 | 环境科学与工程学 | 46 |
| 100 | 32 | 天然产物全合成 | 化学系 | 241 |
| 30 | 48 | 水力学 | 环境科学与工程学 | 563\|582\|-1\|492\|589\|-1\|-2 |
| 20 | 48 | 蛋白质工程 | 生物系 | <null> |
| 50 | 48 | 机械设计基础 | 机械与能源工程系 | 153\|98\|-1\|-2\|-2 |
| 45 | 64 | 智能机器人 | 计算机科学与工程 | 494\|309\|256\|-2\|-2 |
| 80 | 48 | 物理化学 | 材料科学与工程系 | 586\|563\|582\|-1\|-2 |
| 25 | 48 | 病理学 | 医学院 | 37 |

**Figure 3-3. Prerequisite in database**

Figure 3-3 shows the final form of prerequisite in database. There are still one problem that the number of Boolean operators might exceed the reasonable bound as we highlighted in the figure. The reason is shown below:

```
(A or B) and (C or D)   --[A and B Invalid]-->      () and (C or D)
() and (C or D)         --[postorder]-------->      C D or and
```

However, after some calculation, we found that only when there are more than two layers of brackets, this problem cause bugs. Since there are no data in prerequisite that have more than two layers of brackets, this problem has no influence. We only need to calculate the post-order expression in the normal steps and simply ignore the Boolean operators left.

## 4. Optimize the speed of importing

### (1) Batch

The first time we completed the import, we found that it took a lot of time,through searching the reason, we found that the long time was caused by our repeated communication with the database.

Every time we execute a database statement, a communication with the database occurs. The code in Code 3-9 is executed each time an SQL statement is generated. This can be a huge time drain.

```
psts.executeUpdate();
```

**Code 3-9. Bad code**

So we did some optimization on the connection.

- **Close the `auto_commit` of `connection`**

  We used `conn.setAutoCommit(false)` to close the `auto_commit`.

  If we don't close the `auto_commit`, it will commit each time we do something to the `connection`. As a result, the speed would stay slow.

- **Use batch**

  The code `psts.addBatch()` is executed after each SQL statement is generated, Its purpose is to put unprocessed SQL statements on a `stack`.

  In our test code we found that the first code add into batch will not execute until the last, just act like `FILO`.

  This code executes once after a certain number of passes, and that number of times is the `BATCH_SIZE` in the code. `BATCH_SIZE` is a very impotent variable, it will affect the speed of importing data.

  The greater `BATCH_SIZE` means that less time it will execute but more numbers of SQL it will execute in in the same time. Its a funny factors, because it generate a balance.

  There is **no best** `BATCH_SIZE`. It differs according to the CPU you use. Better CPU can handle more SQL statements at the same time, in this case a bigger `BATCH_SIZE` will be perfect. However in the same time, if the CPU is not good enough to handle so much statements, setting a very big `BATCH_SIZE` will more easily cause the risk of CPU errors. In this case, smaller `BATCH_SIZE` is more proper.

  ```
  if (cnt % BATCH_SIZE == 0) {
    stmt.executeBatch();
  stmt.clearBatch();
    conn.commit();
  }
  ```

  This code executes once when all SQL statement has load into Batch or already execute. This will ensure that after we insert into a table, all SQL will be executed, no data lose.

  ```
  if (cnt % BATCH_SIZE != 0) {
    stmt.executeBatch();
  stmt.clearBatch();
    conn.commit();
  }
  ```

**Code 3-10. addBatch()**

- **Why it is better**

  What we did was applying the batch feature in JDBC, this touches on the different ways SQL commands are loaded in JDBC: **statement** and **prepared statement**.

  - **statement**

    Use the Statement object. The Statement object is used for processing when only one-time access to the database is performed. Prepared Statement objects are more expensive than statements and provide no additional benefit for one-time operations.

  - **prepared statement**

    The SQL Statement is precompiled by the database system (if the JDBC driver supports it). The precompiled SQL query can be reused for future queries, making it faster than the query generated by the Statement object.

  So how can we choose from these two statements?

    - **prepared statement is designed to be optimal for multiple use**

      **The first execution of a Prepared Statement is expensive.** Its performance is reflected in the subsequent repeated execution. When we generate a basic query using **prepared statement** the JDBC driver sends a network request to parse the data and optimize the query. Execution generates another network request. **In JDBC drivers, reducing network traffic is the ultimate goal.** If my program requires only one request during execution, use **Statement**. For a **Statement**, the same query generates only one network-to-database communication.

    - **prepared statement has greatly improved security**

      If you use a precompiled statement. Anything you pass in will not have any matching relationship with the original statement. As long as you use all the precompiled statements, you don't have to worry about the incoming data. If you use a normal statement, you might want to specify a drop,; and so on to do painstaking judgment and overthinking.

  **Statement** and **prepared statement** performs differently while using **batch**.

  Using `Statement.addBatch(SQL)` to implement batch processing can send different types of SQL statements to the database, but these SQL statements are not precompiled and inefficient to execute. And you need to list each SQL statement. The `PreparedStatement.addBatch ()` can only be used in the same type parameters of different SQL statements, this form of batch is often used for bulk insert data in the same table, or batch update table data.

- **Conclusion**

  When the previous code communicates with the database, it establishes the connection first, and the cost of establishing the connection is the highest. Then it issues a SQL statement and closes the connection after execution. Another problem is that the SQL statements that are sent are sent over the network, which is also much more expensive than local calls .If you want to insert or update a batch of data into the database, or use the previous method, it will take a lot of time. However, if you use batch processing, you can save most of the cost in both convenience and the speed will be faster.

### (2) `Hash_map`

To my surprise, after we use `BATCH` to make our code faster, our code has not become faster even a little bit. It's because that beside the `INSERT` statement, we still use a lot of `SELECT` statement while inserting data. It is the same reason why make our code slow. So reduce `SELECT` must be done.

But how ? We choose `hash_map`.

- **when we need `hash_map` ?**

  The answer is simple: when we meets **foreign key**.

  In relational database, foreign key is unavoidable.

  If we want to insert a row with foreign key, we have to get the information of what foreign key is pointing to.

  Take the foreign key between table `course` and `class` for example.

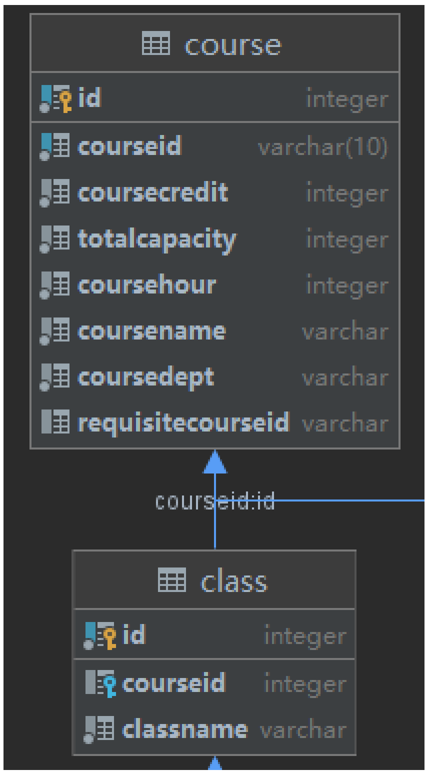


**Figure 3-4. Foreign key**

Every time we insert a row into table `class`, we need to get the information of `course(Id)` by querying table `course` with `courseid`. This is a huge cost.

So we can maintain a `Hash_MAP` to store the relations of foreign keys.

- **Generating `hash_map`**

  First, initialize a `hash_map`.

```java
private static final Map<String, Integer> MAP_course = new HashMap<>();
    //courseId->Id
```

  Then we store the relation into it while inserting the **referenced** data into database.

```java
//add relation of course_Id->id into hash_map
if (!MAP_course.containsKey(course_info.getCourseId())) {
  MAP_course.put(course_info.getCourseId(), cnt);
}
```

**Code 3-11. Generate hash_map**

- **Using `hash_map`**

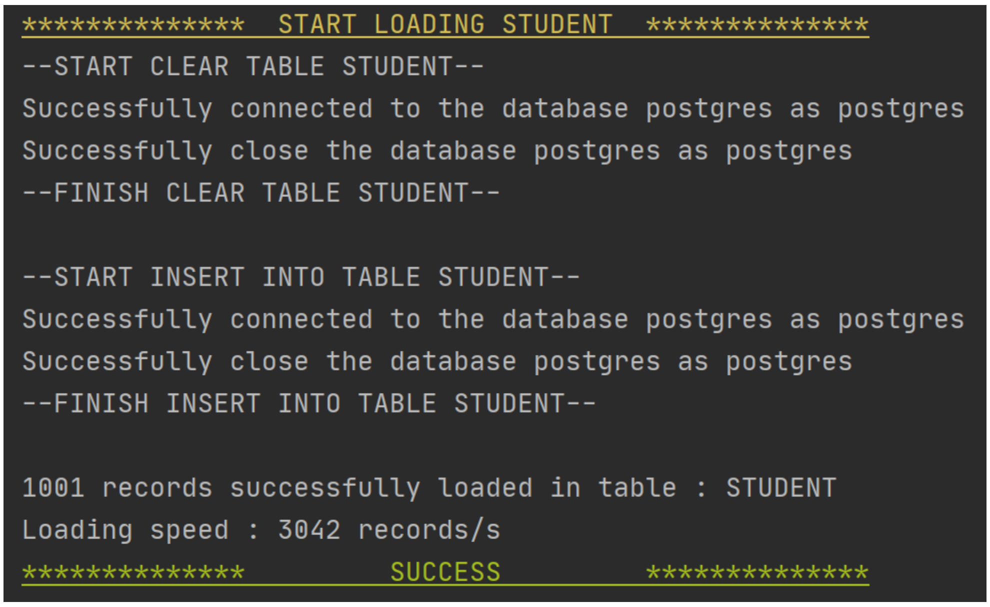
  When we inserting rows with foreign keys, we only need to search from the `hash_map` to get what we need in O(1) time cost.

```java
String course_Id_real = jsonOBJ_classArray.get("courseId").getAsString();
int courseId = MAP_course.get(course_Id_real);
class_info.setId(id);
```

**Code 3-12. Using hash_map**

## (3) Compare

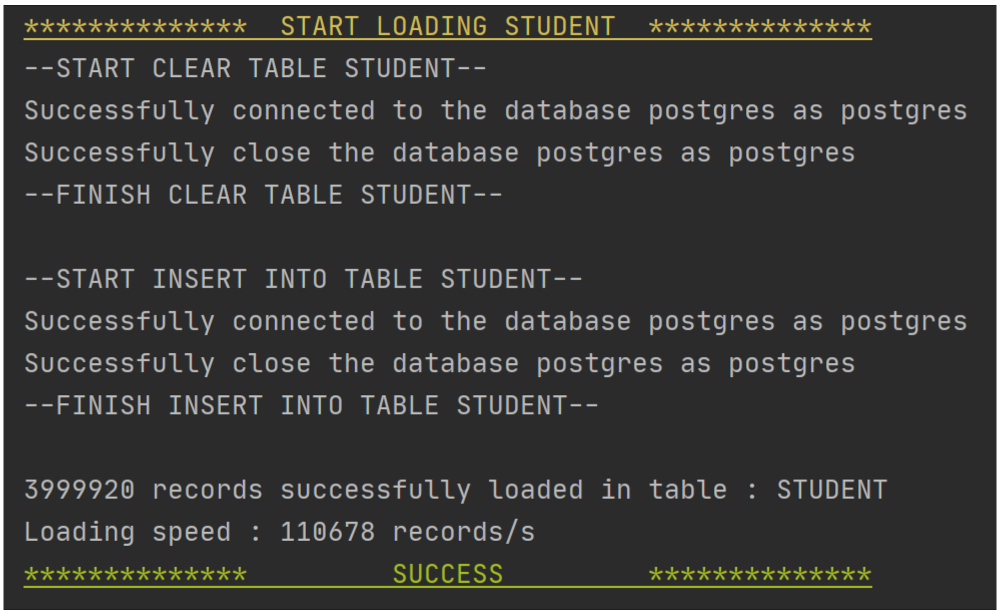
For table `student` in `select_course.csv`, we got `3042 records/s` on **localhost** without speeding up.



```
**************  START LOADING STUDENT  **************
--START CLEAR TABLE STUDENT--
Successfully connected to the database postgres as postgres
Successfully close the database postgres as postgres
--FINISH CLEAR TABLE STUDENT--

--START INSERT INTO TABLE STUDENT--
Successfully connected to the database postgres as postgres
Successfully close the database postgres as postgres
--FINISH INSERT INTO TABLE STUDENT--

1001 records successfully loaded in table : STUDENT
Loading speed : 3042 records/s
**************         SUCCESS         **************
```

**Figure 3-5. Diagram of csv localhost no acceleration**

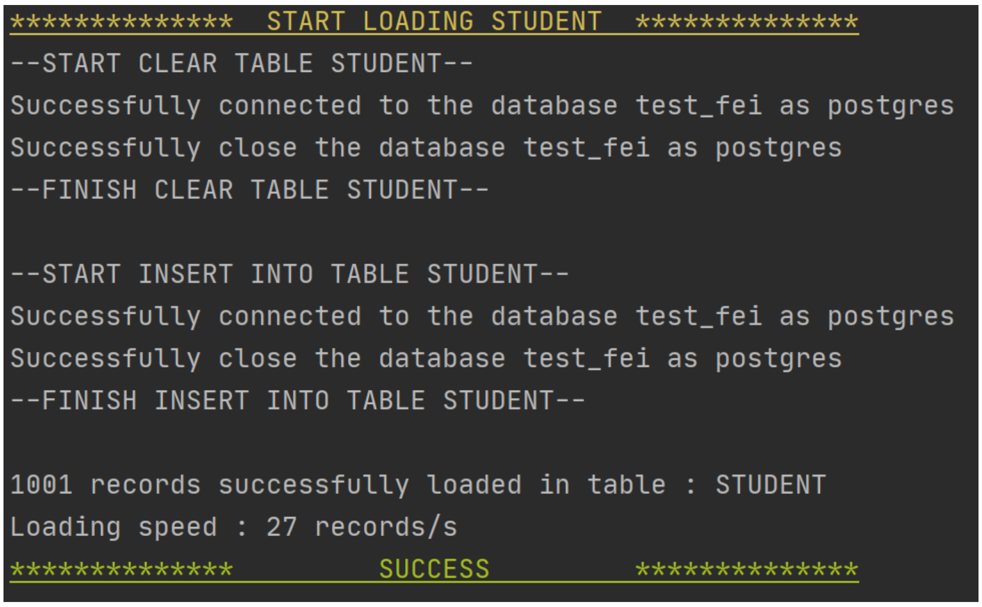
After speeding up, we got `110678 records/s` on **localhost**. Unbelievably fast.



```
**************  START LOADING STUDENT  **************
--START CLEAR TABLE STUDENT--
Successfully connected to the database postgres as postgres
Successfully close the database postgres as postgres
--FINISH CLEAR TABLE STUDENT--

--START INSERT INTO TABLE STUDENT--
Successfully connected to the database postgres as postgres
Successfully close the database postgres as postgres
--FINISH INSERT INTO TABLE STUDENT--

3999920 records successfully loaded in table : STUDENT
Loading speed : 110678 records/s
**************         SUCCESS         **************
```

**Figure 3-6. Diagram of csv localhost with acceleration**

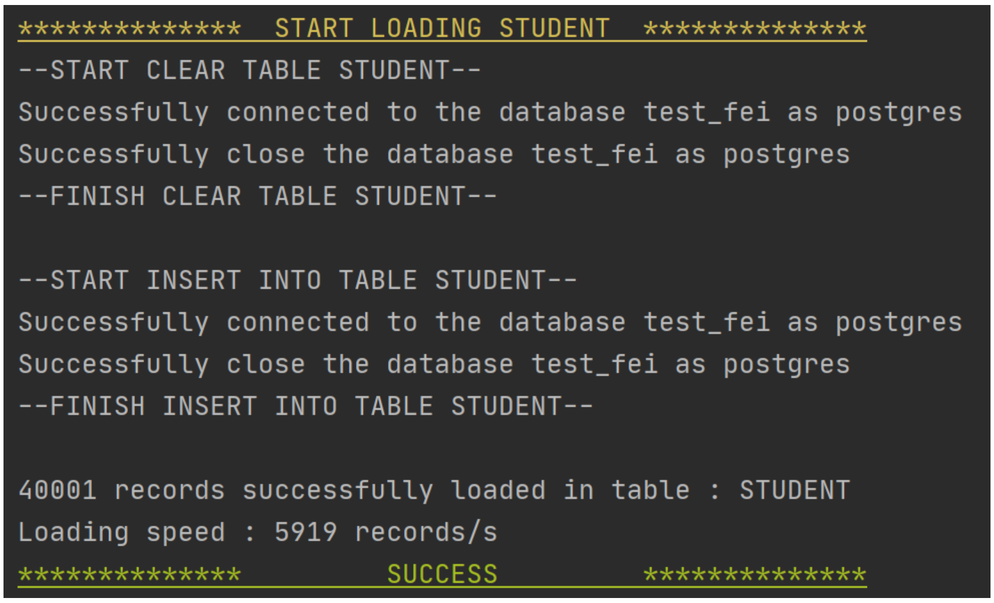
When it comes to the **online database server**, the speed is pretty slow. We got `27 records/s` before speeding up, which is unbearable.



```
**************  START LOADING STUDENT  **************
--START CLEAR TABLE STUDENT--
Successfully connected to the database test_fei as postgres
Successfully close the database test_fei as postgres
--FINISH CLEAR TABLE STUDENT--

--START INSERT INTO TABLE STUDENT--
Successfully connected to the database test_fei as postgres
Successfully close the database test_fei as postgres
--FINISH INSERT INTO TABLE STUDENT--

1001 records successfully loaded in table : STUDENT
Loading speed : 27 records/s
**************         SUCCESS         **************
```

**Figure 3-7. Diagram of csv server no acceleration**

After speeding up we got `5919 records/s` on **online server**, this is much better.



```
**************  START LOADING STUDENT  **************
--START CLEAR TABLE STUDENT--
Successfully connected to the database test_fei as postgres
Successfully close the database test_fei as postgres
--FINISH CLEAR TABLE STUDENT--

--START INSERT INTO TABLE STUDENT--
Successfully connected to the database test_fei as postgres
Successfully close the database test_fei as postgres
--FINISH INSERT INTO TABLE STUDENT--

40001 records successfully loaded in table : STUDENT
Loading speed : 5919 records/s
**************         SUCCESS         **************
```

**Figure 3-8. Diagram of csv server with acceleration**

On `localhost` we got `36` times faster, but on `server` we got `108` times faster!

The speed up code only have a slight influence on **localhost**. This is because the cost of connecting with database on local host is relatively small.

This diagram on Figure 3-9 is really shocking but unfortunately we can't see the data on `unaccelerated server` because it is too small (only 27) .

Another thing we can see is that, with different `Batch_Size` the **server** performance differently with **localhost**. Data on the **localhost** shows that the less `Batch_size` we choose, the better performance we got. But when it comes to **server**, 30000 seems to be the best choice. This is because the CPU we got on **localhost** works better than CPU on **server**. It validates what I said earlier .

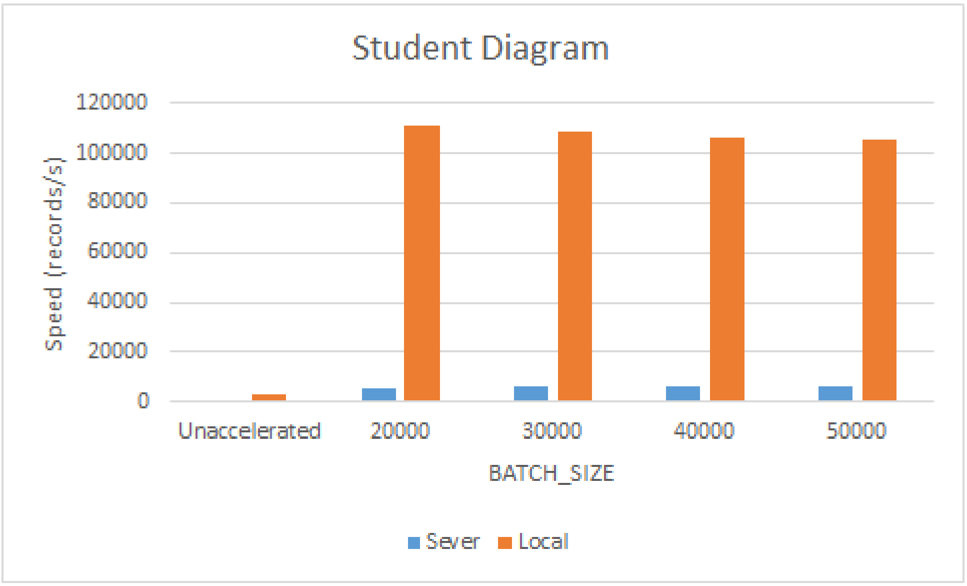


**Figure 3-9. Diagram of csv Files**

This is the diagram of `JSON` files after acceleration. We want to show that, since reading the `JSON` files needs more time than reading `CSV` files, there are speed differences between `JSON` and `CSV`. Also, because the `JSON` files is much smaller than the `CSV` file, the acceleration rate of `JSON` files is smaller than the `CSV` file. It validates what I said earlier.

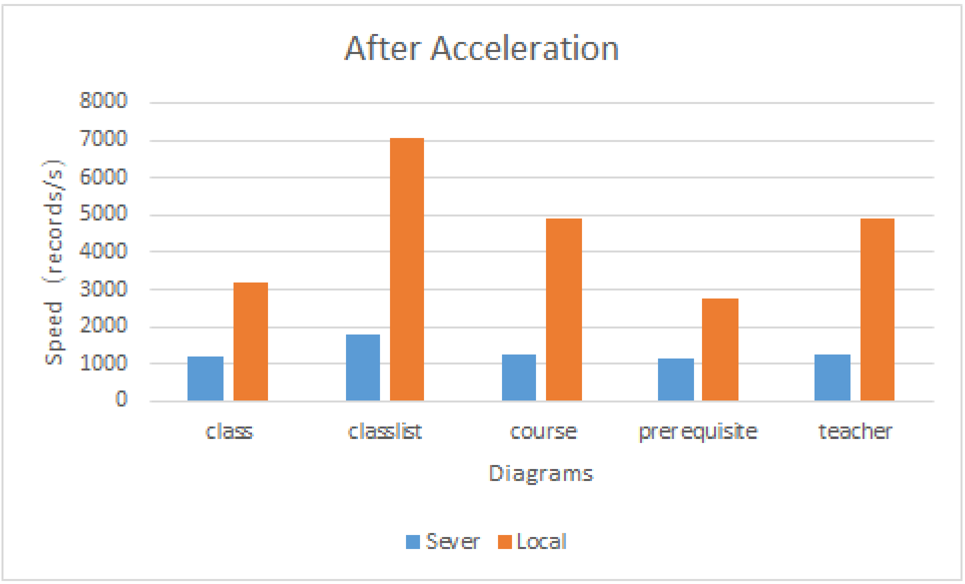
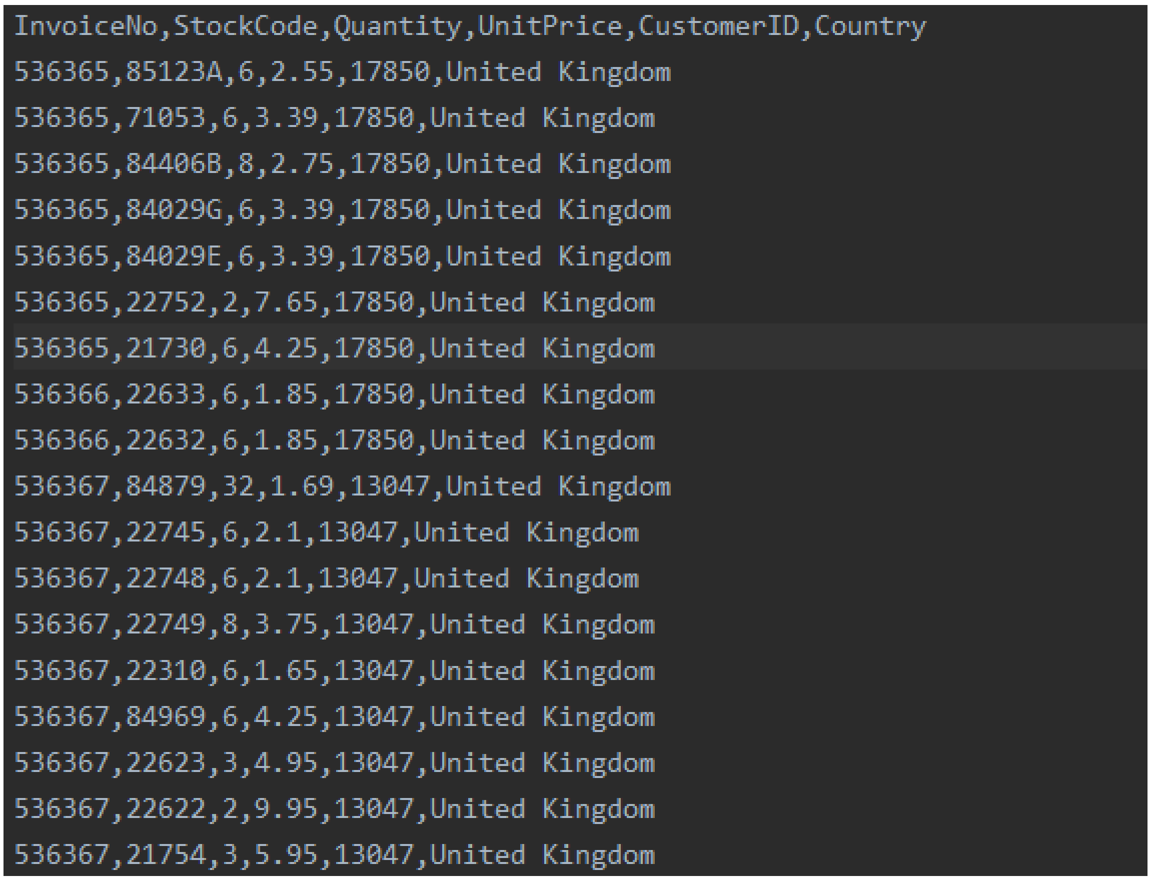


**Figure 3-10. Diagram of JSON File**

# Part 4. Task 3 Implementation & Analysis

## 1. Brief Introduction

Our dataset is a dataset about E-Commerce found in Kaggle. Figure 4-1 shows the detail of the dataset.



```
InvoiceNo,StockCode,Quantity,UnitPrice,CustomerID,Country
536365,85123A,6,2.55,17850,United Kingdom
536365,71053,6,3.39,17850,United Kingdom
536365,84406B,8,2.75,17850,United Kingdom
536365,84029G,6,3.39,17850,United Kingdom
536365,84029E,6,3.39,17850,United Kingdom
536365,22752,2,7.65,17850,United Kingdom
536365,21730,6,4.25,17850,United Kingdom
536366,22633,6,1.85,17850,United Kingdom
536366,22632,6,1.85,17850,United Kingdom
536367,84879,32,1.69,13047,United Kingdom
536367,22745,6,2.1,13047,United Kingdom
536367,22748,6,2.1,13047,United Kingdom
536367,22749,8,3.75,13047,United Kingdom
536367,22310,6,1.65,13047,United Kingdom
536367,84969,6,4.25,13047,United Kingdom
536367,22623,3,4.95,13047,United Kingdom
536367,22622,2,9.95,13047,United Kingdom
536367,21754,3,5.95,13047,United Kingdom
```

**Figure 4-1. E-Commerce dataset**

We implemented this using `C++`, and we ran this program on `Windows(x86)` and `Linux(x86)`.

## 2. Basic structure

I designed two `structs`, `Table` and `Record` to represent the structure of a table.

The basic structure of a table is a `vector` of `Record*`.

```cpp
void filter_index() {
  cur = 0;
  index_idx = -1;
  indexused = false;
  while (true) {
    if (cur > 6) {
      error_msg = "Error: Too much condition added.\n";
      error_hap = true;
      return;
    }
    cin >> key[cur];
    if (key[cur] == "-1") {
      break;
    } else if (!indexused) {
      if (key[cur] == "InvoiceNo" || key[cur] == "invoiceno") {
        indexused = true;
        index_idx = cur;
        temp_index = InvoiceNo_index;
      } else if (key[cur] == "StockCode" || key[cur] == "stockcode") {
        indexused = true;
        index_idx = cur;
        temp_index = StockCode_index;
      } else if (key[cur] == "CustomerID" || key[cur] == "customerid") {
        indexused = true;
        index_idx = cur;
        temp_index = CustomerID_index;
      }
    }
    cin >> value[cur];
    cur++;
  }
  if (indexused) {
    if (temp_index->count(value[index_idx])) {
      rset = temp_index->at(value[index_idx]);
    }
  }
  for (int j = 0; j < cur; j++) {
    if (j == index_idx) {
      continue;
    }
    if (key[j] == "InvoiceNo" || key[j] == "invoiceno") {
      for (int i = 0; i < rset.size(); i++) {
        if (rset[i]->InvoiceNo != value[j]) {
          rset[i]->unselected = true;
        }
      }
    } else if (key[j] == "StockCode" || key[j] == "stockcode") {
      for (int i = 0; i < rset.size(); i++) {
        if (rset[i]->StockCode != value[j]) {
          rset[i]->unselected = true;
        }
      }
    } else if (key[j] == "Quantity" || key[j] == "quantity") {
      if (rset.size()) {
        for (int i = 0; i < rset.size(); i++) {
          if (rset[i]->Quantity != value[j]) {
            rset[i]->unselected = true;
          }
        }
      } else {
        for (int i = 0; i < t->records.size(); i++) {
          if (t->records[i]->Quantity == value[j]) {
            rset.push_back(t->records[i]);
          }
        }
      }
    } else if (key[j] == "UnitPrice" || key[j] == "unitprice") {
      if (rset.size()) {
        for (int i = 0; i < rset.size(); i++) {
          if (rset[i]->UnitPrice != value[j]) {
            rset[i]->unselected = true;
          }
        }
      } else {
        for (int i = 0; i < t->records.size(); i++) {
          if (t->records[i]->UnitPrice == value[j]) {
            rset.push_back(t->records[i]);
          }
        }
      }
    } else if (key[j] == "CustomerID" || key[j] == "customerid") {
      for (int i = 0; i < rset.size(); i++) {
        if (rset[i]->CustomerID != value[j]) {
          rset[i]->unselected = true;
        }
      }
    } else if (key[j] == "Country" || key[j] == "country") {
      if (rset.size()) {
        for (int i = 0; i < rset.size(); i++) {
          if (rset[i]->Country != value[j]) {
            rset[i]->unselected = true;
          }
        }
      } else {
        for (int i = 0; i < t->records.size(); i++) {
          if (t->records[i]->Country == value[j]) {
            rset.push_back(t->records[i]);
          }
        }
      }
    } else {
      error_msg = "Error: Invalid column name '" + key[j] + "'.\n";
      error_hap = true;
      return;
    }
  }
};
```

**Code 4-1. Function of finding rows**

Code 4-1 shows the function of finding rows we wanted, which is the core code of this program.

We first generate a result set by one of the select conditions, then we go through the other conditions to exclude the rows from the current result set. At last, the rows remaining in the result set are the rows we want.

## 3. Index

We want to prove that adding index can speed up the operations of selecting, but it will slow down the speed of updating, deleting, inserting and table loading.

### (1) Implementation

We implemented index by `map` in `C++`, which is actually supported by a binary tree.

This is similar to the index structure in database.

```
map<string, vector<Record*>> *InvoiceNo_index = new map<string, vector<Record*>>(),
                             *StockCode_index = new map<string, vector<Record*>>(),
                             *CustomerID_index = new map<string, vector<Record*>>();

```

```
void add_index(map<string, vector<Record*>>* index, string key, Record* value) {
  map_itr = index->find(key);
  if (map_itr != index->end()) {
    map_itr->second.push_back(value);
  } else {
    temp.push_back(value);
    index->insert(pair<string, vector<Record*>>(key, temp));
    temp.pop_back();
  }
};
```

**Code 4-2. Function of adding indexes**

Since we don't have column with unique constraints, our index is a map from `string` (column values) to a `vector` of `Record*`. The function of adding indexes is shown in Code 4-2.

### (2) Advantages

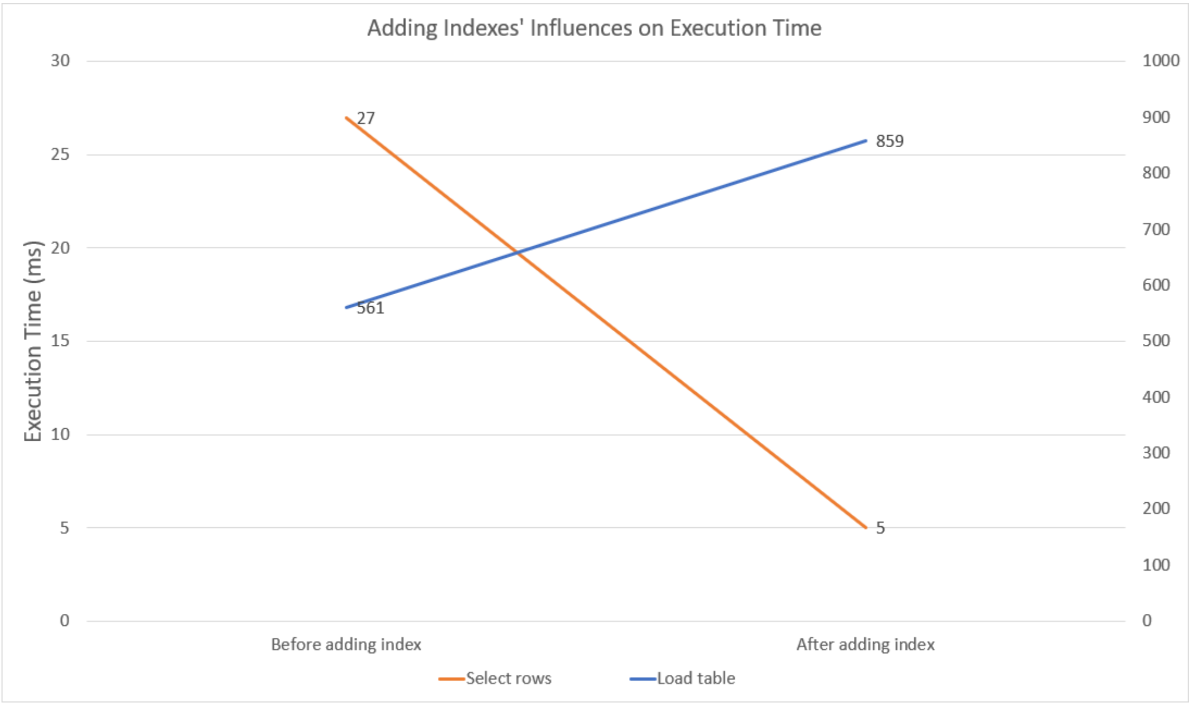


**Figure 4-4. Adding Indexes' Influences on Execution Time**

As we mentioned above, when we are selecting rows, we will first generate a result set (actually a `vector` of `Record*`) according to one of the conditions.

After adding indexes, if we find one of the conditions is applied on a column with index, then we will use it to generate the result set. Just find the value of the index map (a `vector` of `Record*`) and copy it to the result set. This is also shown in Code 4-1.

Adding index will save a lot of time than matching all the rows one by one. In testcases, the average query time reduced about `80%` after adding index. This is shown in Figure 4-4.

### (3) Drawbacks

After adding index, every time a new row is inserted, the index must be updated. More indexes, more time cost.

We added index on three of the columns. After adding indexes, the time cost in loading table raised about `60%`. This is also shown in Figure 4-4.

Also, after adding index, there are a lot more memory used to store the indexes (`map` in my program).

### (4) Analysis

Adding index can improve the speed of finding corresponding rows, while it will slow down the speed of inserting, updating, deleting rows. Also, it will cause more storage cost.

In most cases, there are far more select operations that other operations, so adding index on some of the columns is recommended in most cases.

## 4. User privileges management

We want to prove that database have complicated user privilege management and schema managements.

### (1) Our implementation

I used a struct to represent the users. They have three different privileges: `SuperUser`, `Admin` and `NormalUser`. This is shown in Code 4-3.

```
enum priv { SuperUser, Admin, NormalUser };
struct User {
  string userName;
  string password;
  priv identity;
};
map<string, User*> user_list;
```

**Code 4-3. User privileges implementation**

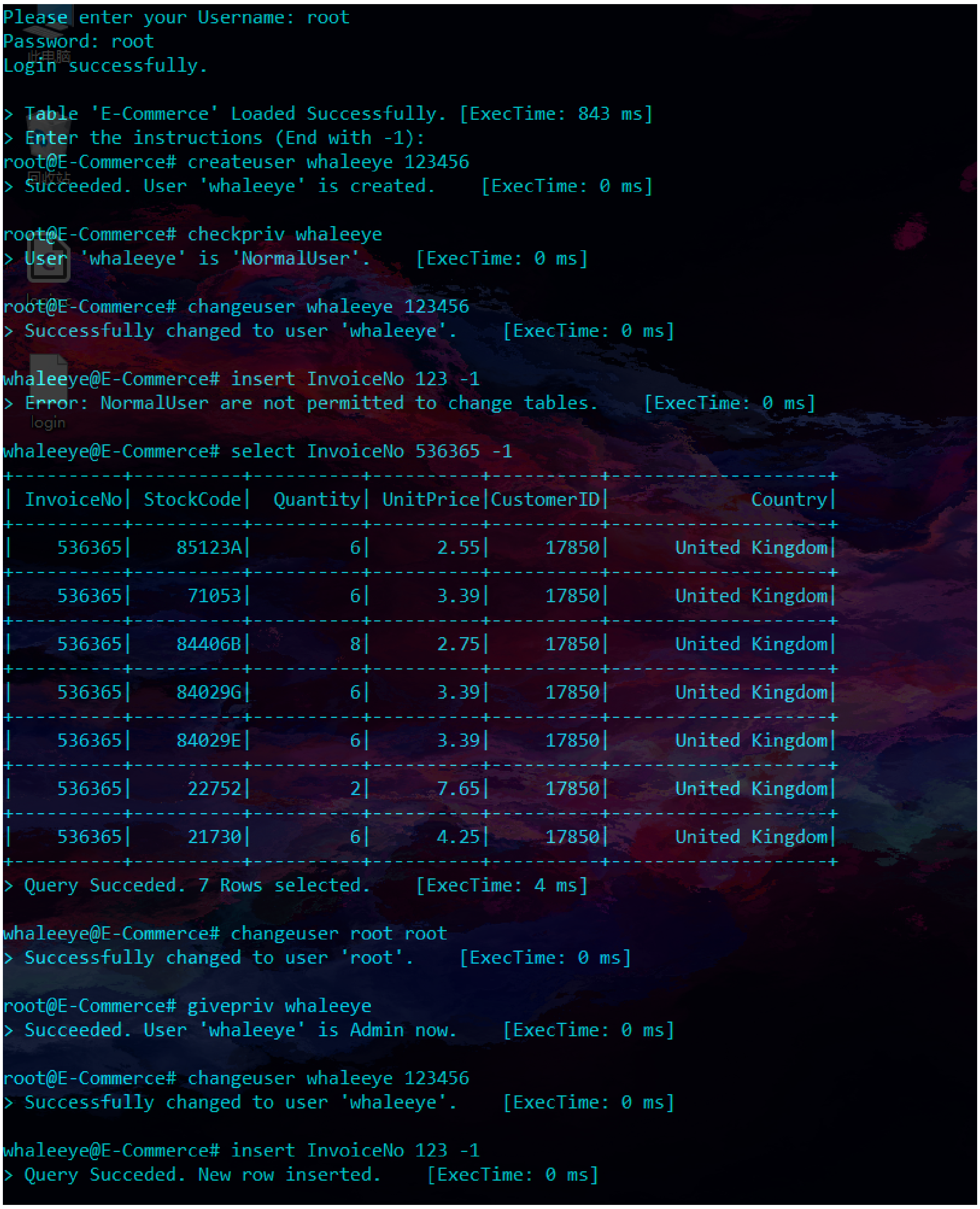
There are only one `SuperUser`, who can create users, lift privileges and check privileges.

`Admin` can `insert`, `delete`, `update`, `select` the table.

`NormalUser` can only `select` the table.

Any user can switch to any other user by username and password.

These are all demonstrated in Figure 4-6.

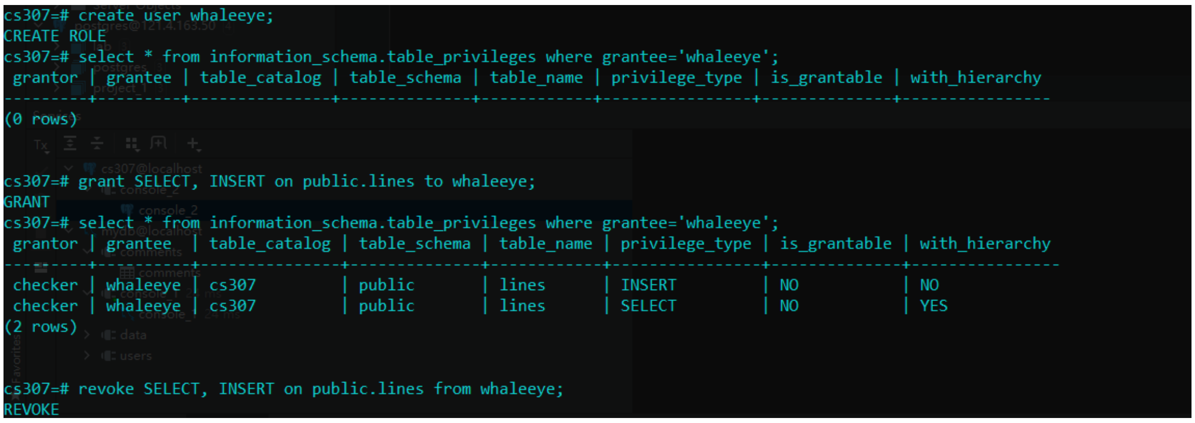


```
Please enter your Username: root
Password: root
Login successfully.

> Table 'E-Commerce' Loaded Successfully. [ExecTime: 843 ms]
> Enter the instructions (End with -1):
root@E-Commerce# createuser whaleeye 123456
> Succeeded. User 'whaleeye' is created.    [ExecTime: 0 ms]

root@E-Commerce# checkpriv whaleeye
> User 'whaleeye' is 'NormalUser'.    [ExecTime: 0 ms]

root@E-Commerce# changeuser whaleeye 123456
> Successfully changed to user 'whaleeye'.    [ExecTime: 0 ms]

whaleeye@E-Commerce# insert InvoiceNo 123 -1
> Error: NormalUser are not permitted to change tables.    [ExecTime: 0 ms]

whaleeye@E-Commerce# select InvoiceNo 536365 -1
+----------+----------+----------+----------+----------+--------------------+
| InvoiceNo| StockCode|  Quantity| UnitPrice|CustomerID|             Country|
+----------+----------+----------+----------+----------+--------------------+
|    536365|    85123A|         6|      2.55|     17850|      United Kingdom|
+----------+----------+----------+----------+----------+--------------------+
|    536365|     71053|         6|      3.39|     17850|      United Kingdom|
+----------+----------+----------+----------+----------+--------------------+
|    536365|    84406B|         8|      2.75|     17850|      United Kingdom|
+----------+----------+----------+----------+----------+--------------------+
|    536365|    84029G|         6|      3.39|     17850|      United Kingdom|
+----------+----------+----------+----------+----------+--------------------+
|    536365|    84029E|         6|      3.39|     17850|      United Kingdom|
+----------+----------+----------+----------+----------+--------------------+
|    536365|     22752|         2|      7.65|     17850|      United Kingdom|
+----------+----------+----------+----------+----------+--------------------+
|    536365|     21730|         6|      4.25|     17850|      United Kingdom|
+----------+----------+----------+----------+----------+--------------------+
> Query Succeded. 7 Rows selected.    [ExecTime: 4 ms]

whaleeye@E-Commerce# changeuser root root
> Successfully changed to user 'root'.    [ExecTime: 0 ms]

root@E-Commerce# givepriv whaleeye
> Succeeded. User 'whaleeye' is Admin now.    [ExecTime: 0 ms]

root@E-Commerce# changeuser whaleeye 123456
> Successfully changed to user 'whaleeye'.    [ExecTime: 0 ms]

whaleeye@E-Commerce# insert InvoiceNo 123 -1
> Query Succeded. New row inserted.    [ExecTime: 0 ms]
```

**Figure 4-4. User privileges demonstration**

## (2) Database Privileges

We tested PostgreSQL on it's privilege management and schema managements. The details are shown in Figure 4-5.



```
cs307=# create user whaleeye;
CREATE ROLE
cs307=# select * from information_schema.table_privileges where grantee='whaleeye';
 grantor | grantee | table_catalog | table_schema | table_name | privilege_type | is_grantable | with_hierarchy
---------+---------+---------------+--------------+------------+----------------+--------------+---------------
(0 rows)

cs307=# grant SELECT, INSERT on public.lines to whaleeye;
GRANT
cs307=# select * from information_schema.table_privileges where grantee='whaleeye';
 grantor | grantee  | table_catalog | table_schema | table_name | privilege_type | is_grantable | with_hierarchy
---------+----------+---------------+--------------+------------+----------------+--------------+----------------
 checker | whaleeye | cs307         | public       | lines      | INSERT         | NO           | NO
 checker | whaleeye | cs307         | public       | lines      | SELECT         | NO           | YES
(2 rows)

cs307=# revoke SELECT, INSERT on public.lines from whaleeye;
REVOKE
```

**Figure 4-5. PostgreSQL user privileges**

### (3) Compare

PostgreSQL's user privilege management is based on schema management, and the privileges one user have can be divided into small parts like `select`, `insert`, `update`, `delete` and so on. Also, if we have gave a user some privileges, we cannot delete the user until we revoke the privileges.

Compared with our program, the database's implementation of user privilege management is far more complicated compact. In the same time, it's also more flexible because it can be divided into small parts.

## 5. Query & IO Speed

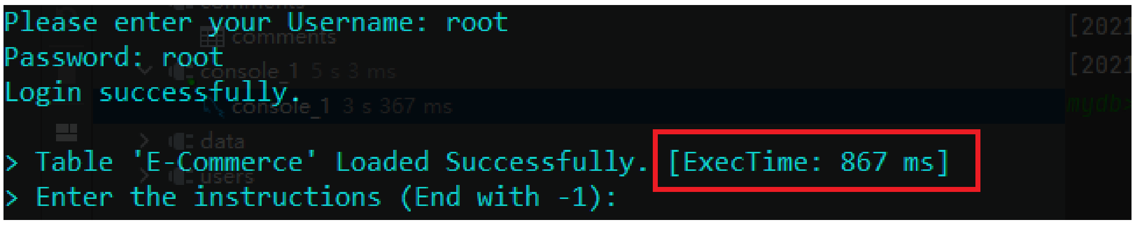
We want to prove that DBMS has more effective data management methods, so DBMS will have higher speed in query and IO.

We tested our program and two different DBMS, PostgreSQL and MySQL, on the platform of Windows(x86) and Linux(x86).

### (1) Our Program

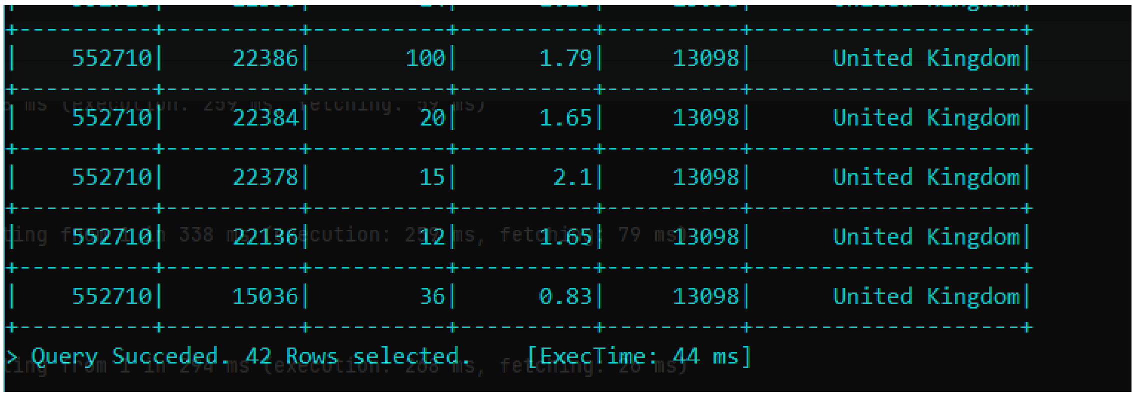
- **Windows**

  The load time (I/O time cost) and CRUD time on Windows are shown in Figure 4-6-a, 4-6-b, 4-6-c.



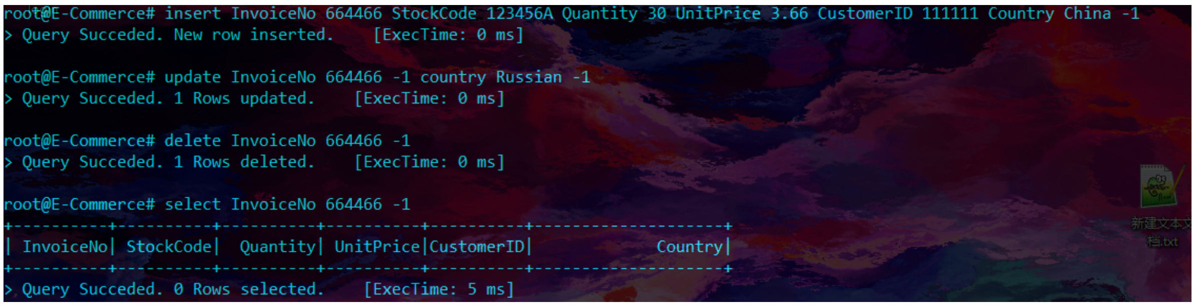
```
Please enter your Username: root
Password: root
Login successfully.

> Table 'E-Commerce' Loaded Successfully. [ExecTime: 867 ms]
> Enter the instructions (End with -1):
```

**Figure 4-6-a. Windows I/O time (Program)**



```
+----------+----------+----------+----------+----------+--------------------+
|    552710|     22386|       100|      1.79|     13098|      United Kingdom|
+----------+----------+----------+----------+----------+--------------------+
|    552710|     22384|        20|      1.65|     13098|      United Kingdom|
+----------+----------+----------+----------+----------+--------------------+
|    552710|     22378|        15|       2.1|     13098|      United Kingdom|
+----------+----------+----------+----------+----------+--------------------+
|    552710|     22136|        12|      1.65|     13098|      United Kingdom|
+----------+----------+----------+----------+----------+--------------------+
|    552710|     15036|        36|      0.83|     13098|      United Kingdom|
+----------+----------+----------+----------+----------+--------------------+
> Query Succeded. 42 Rows selected.    [ExecTime: 44 ms]
```

**Figure 4-6-b. Windows select time (Program)**

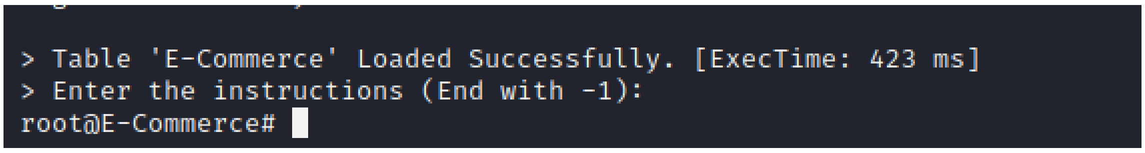


```
root@E-Commerce# insert InvoiceNo 664466 StockCode 123456A Quantity 30 UnitPrice 3.66 CustomerID 111111 Country China -1
> Query Succeded. New row inserted.    [ExecTime: 0 ms]

root@E-Commerce# update InvoiceNo 664466 -1 country Russian -1
> Query Succeded. 1 Rows updated.    [ExecTime: 0 ms]

root@E-Commerce# delete InvoiceNo 664466 -1
> Query Succeded. 1 Rows deleted.    [ExecTime: 0 ms]

root@E-Commerce# select InvoiceNo 664466 -1
+----------+----------+----------+----------+----------+--------------------+
| InvoiceNo| StockCode|  Quantity| UnitPrice|CustomerID|             Country|
+----------+----------+----------+----------+----------+--------------------+
> Query Succeded. 0 Rows selected.    [ExecTime: 5 ms]
```

**Figure 4-6-c. Windows CRUD time (Program)**

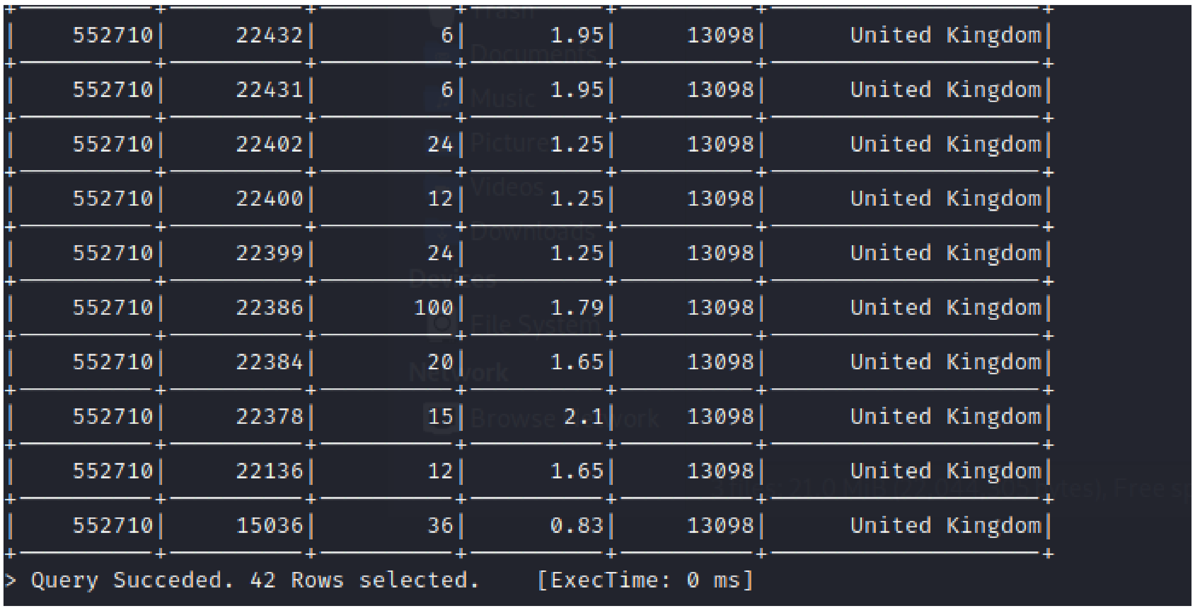
- **Linux**

The load time (I/O time cost) and CRUD time on Linux are shown in Figure 4-7-a, 4-7-b, 4-7-c.



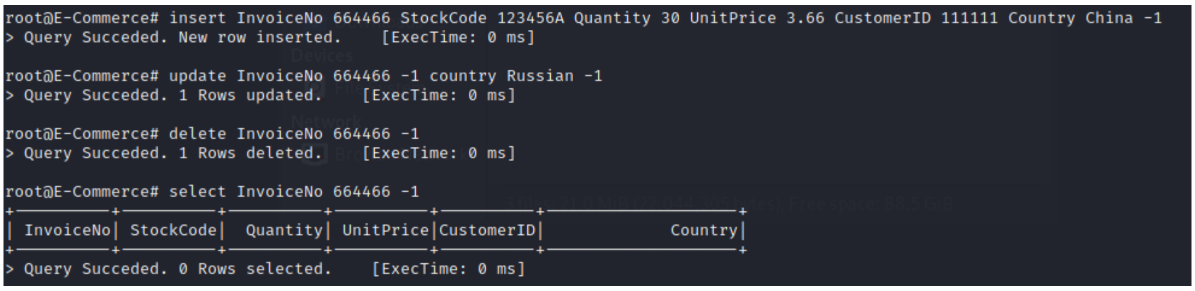
```
> Table 'E-Commerce' Loaded Successfully. [ExecTime: 423 ms]
> Enter the instructions (End with -1):
root@E-Commerce# 
```

**Figure 4-7-a. Linux I/O time (Program)**



```
|    552710|     22432|         6|      1.95|     13098|      United Kingdom|
+----------+----------+----------+----------+----------+--------------------+
|    552710|     22431|         6|      1.95|     13098|      United Kingdom|
+----------+----------+----------+----------+----------+--------------------+
|    552710|     22402|        24|      1.25|     13098|      United Kingdom|
+----------+----------+----------+----------+----------+--------------------+
|    552710|     22400|        12|      1.25|     13098|      United Kingdom|
+----------+----------+----------+----------+----------+--------------------+
|    552710|     22399|        24|      1.25|     13098|      United Kingdom|
+----------+----------+----------+----------+----------+--------------------+
|    552710|     22386|       100|      1.79|     13098|      United Kingdom|
+----------+----------+----------+----------+----------+--------------------+
|    552710|     22384|        20|      1.65|     13098|      United Kingdom|
+----------+----------+----------+----------+----------+--------------------+
|    552710|     22378|        15|       2.1|     13098|      United Kingdom|
+----------+----------+----------+----------+----------+--------------------+
|    552710|     22136|        12|      1.65|     13098|      United Kingdom|
+----------+----------+----------+----------+----------+--------------------+
|    552710|     15036|        36|      0.83|     13098|      United Kingdom|
+----------+----------+----------+----------+----------+--------------------+
> Query Succeded. 42 Rows selected.    [ExecTime: 0 ms]
```

**Figure 4-7-b. Linux select time (Program)**



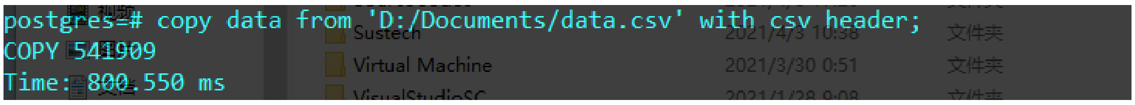
```
root@E-Commerce# insert InvoiceNo 664466 StockCode 123456A Quantity 30 UnitPrice 3.66 CustomerID 111111 Country China -1
> Query Succeded. New row inserted.    [ExecTime: 0 ms]

root@E-Commerce# update InvoiceNo 664466 -1 country Russian -1
> Query Succeded. 1 Rows updated.    [ExecTime: 0 ms]

root@E-Commerce# delete InvoiceNo 664466 -1
> Query Succeded. 1 Rows deleted.    [ExecTime: 0 ms]

root@E-Commerce# select InvoiceNo 664466 -1
+----------+----------+----------+----------+----------+--------------------+
| InvoiceNo| StockCode|  Quantity| UnitPrice|CustomerID|             Country|
+----------+----------+----------+----------+----------+--------------------+
> Query Succeded. 0 Rows selected.    [ExecTime: 0 ms]
```

**Figure 4-7-c. Linux CRUD time (Program)**

## (2) PostgreSQL

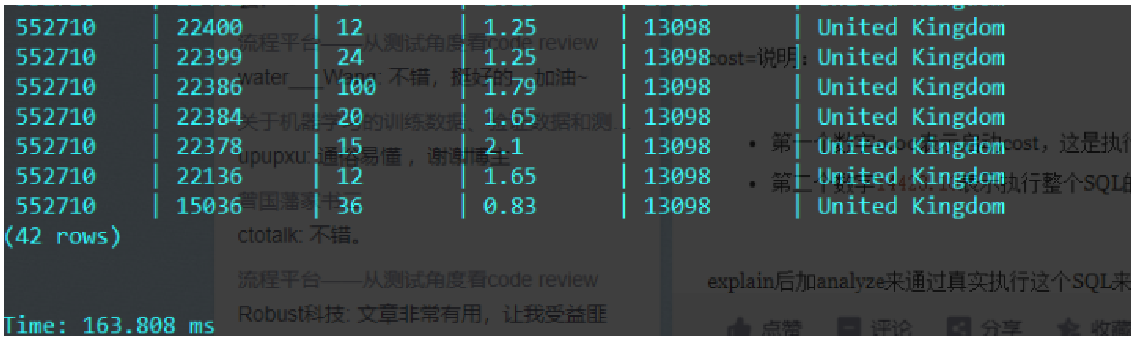
- **Windows**

  The load time (I/O time cost) and CRUD time on Windows are shown in Figure 4-8-a, 4-8-b, 4-8-c.



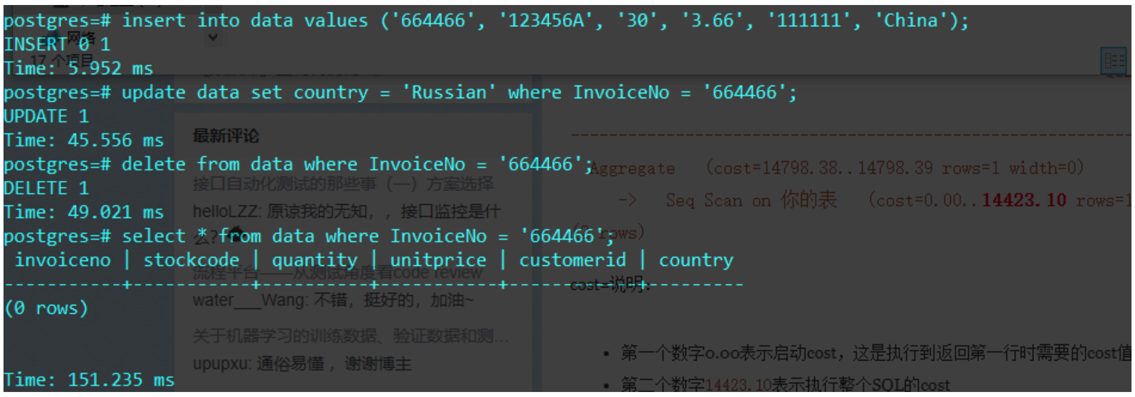
```
postgres=# copy data from 'D:/Documents/data.csv' with csv header;
COPY 541909
Time: 800.550 ms
```

**Figure 4-8-a. Windows I/O time (PostgreSQL)**



```
 552710    | 22400   | 12     | 1.25    | 13098     | United Kingdom
 552710    | 22399   | 24     | 1.25    | 13098     | United Kingdom
 552710    | 22386   | 100    | 1.79    | 13098     | United Kingdom
 552710    | 22384   | 20     | 1.65    | 13098     | United Kingdom
 552710    | 22378   | 15     | 2.1     | 13098     | United Kingdom
 552710    | 22136   | 12     | 1.65    | 13098     | United Kingdom
 552710    | 15036   | 36     | 0.83    | 13098     | United Kingdom
(42 rows)


Time: 163.808 ms
```

**Figure 4-8-b. Windows select time (PostgreSQL)**

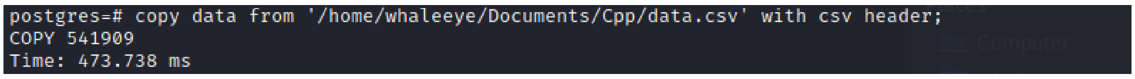


```
postgres=# insert into data values ('664466', '123456A', '30', '3.66', '111111', 'China');
INSERT 0 1
Time: 5.952 ms
postgres=# update data set country = 'Russian' where InvoiceNo = '664466';
UPDATE 1
Time: 45.556 ms
postgres=# delete from data where InvoiceNo = '664466';
DELETE 1
Time: 49.021 ms
postgres=# select * from data where InvoiceNo = '664466';
 invoiceno | stockcode | quantity | unitprice | customerid | country
-----------+-----------+----------+-----------+------------+---------
(0 rows)


Time: 151.235 ms
```

**Figure 4-8-c. Windows CRUD time (PostgreSQL)**

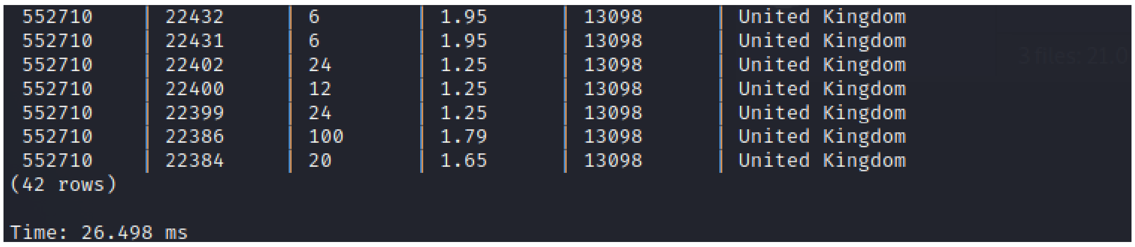
- **Linux**

  The load time (I/O time cost) and CRUD time on Linux are shown in Figure 4-9-a, 4-9-b, 4-9-c.



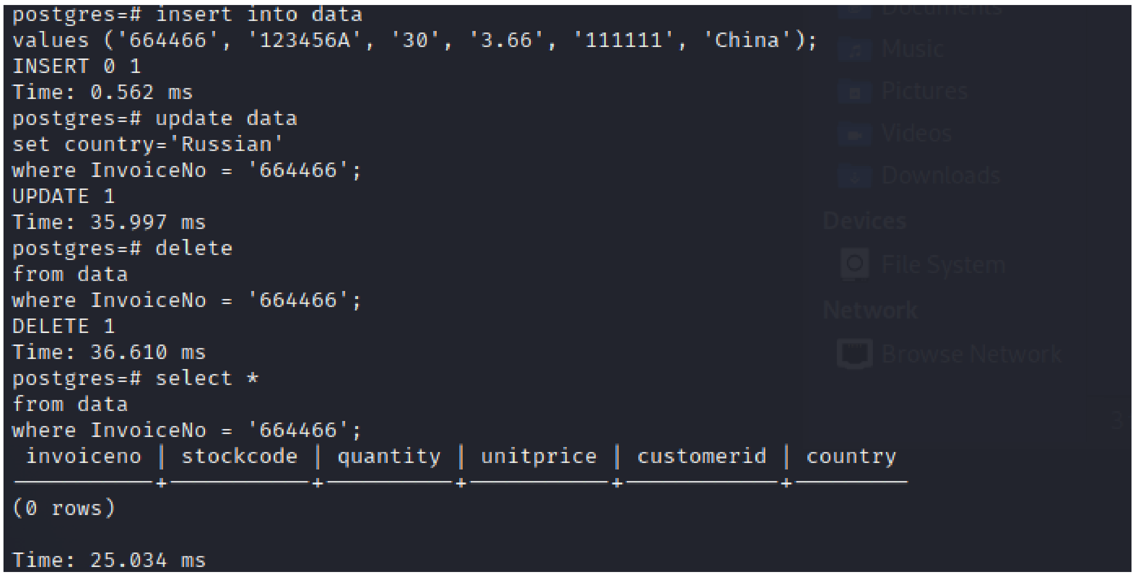
```
postgres=# copy data from '/home/whaleeye/Documents/Cpp/data.csv' with csv header;
COPY 541909
Time: 473.738 ms
```

**Figure 4-9-a. Linux I/O time (PostgreSQL)**



```
 552710    | 22432    | 6       | 1.95    | 13098     | United Kingdom
 552710    | 22431    | 6       | 1.95    | 13098     | United Kingdom
 552710    | 22402    | 24      | 1.25    | 13098     | United Kingdom
 552710    | 22400    | 12      | 1.25    | 13098     | United Kingdom
 552710    | 22399    | 24      | 1.25    | 13098     | United Kingdom
 552710    | 22386    | 100     | 1.79    | 13098     | United Kingdom
 552710    | 22384    | 20      | 1.65    | 13098     | United Kingdom
(42 rows)


Time: 26.498 ms
```

Figure 4-9-b. Linux select time (PostgreSQL)



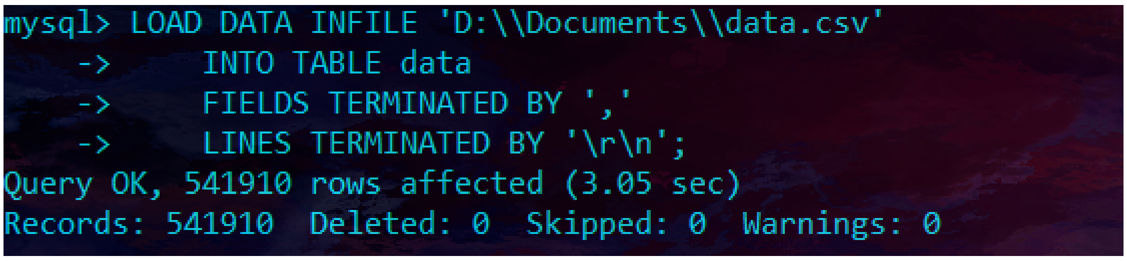
```
postgres=# insert into data
values ('664466', '123456A', '30', '3.66', '111111', 'China');
INSERT 0 1
Time: 0.562 ms
postgres=# update data
set country='Russian'
where InvoiceNo = '664466';
UPDATE 1
Time: 35.997 ms
postgres=# delete
from data
where InvoiceNo = '664466';
DELETE 1
Time: 36.610 ms
postgres=# select *
from data
where InvoiceNo = '664466';
 invoiceno | stockcode | quantity | unitprice | customerid | country
-----------+-----------+----------+-----------+------------+---------
(0 rows)

Time: 25.034 ms
```

Figure 4-9-c. Linux CRUD time (PostgreSQL)

## (3) MySQL

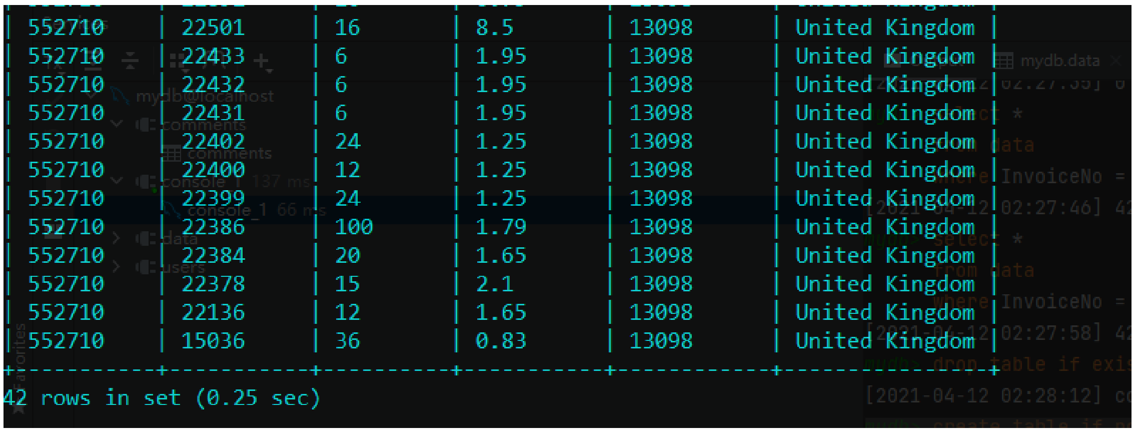
- **Windows**

  The load time (I/O time cost) and CRUD time on Windows are shown in Figure 4-10-a, 4-10-b, 4-10-c.



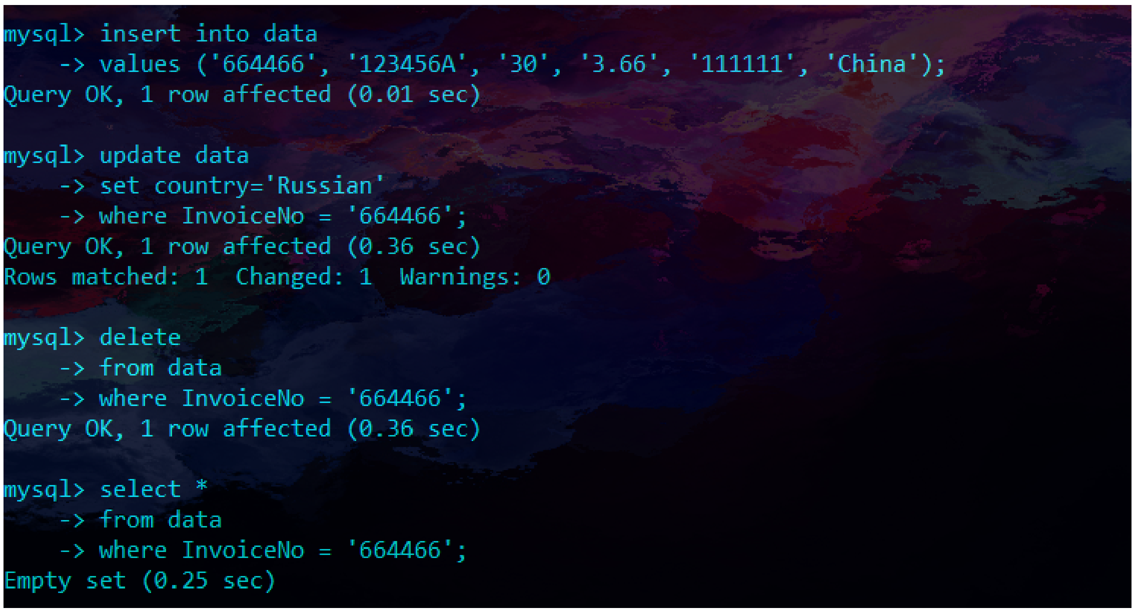
```
mysql> LOAD DATA INFILE 'D:\\Documents\\data.csv'
    ->      INTO TABLE data
    ->      FIELDS TERMINATED BY ','
    ->      LINES TERMINATED BY '\r\n';
Query OK, 541910 rows affected (3.05 sec)
Records: 541910  Deleted: 0  Skipped: 0  Warnings: 0
```

Figure 4-10-a. Windows I/O time (MySQL)



```
| 552710    | 22501     | 16        | 8.5        | 13098      | United Kingdom |
| 552710    | 22433     | 6         | 1.95       | 13098      | United Kingdom |
| 552710    | 22432     | 6         | 1.95       | 13098      | United Kingdom |
| 552710    | 22431     | 6         | 1.95       | 13098      | United Kingdom |
| 552710    | 22402     | 24        | 1.25       | 13098      | United Kingdom |
| 552710    | 22400     | 12        | 1.25       | 13098      | United Kingdom |
| 552710    | 22399     | 24        | 1.25       | 13098      | United Kingdom |
| 552710    | 22386     | 100       | 1.79       | 13098      | United Kingdom |
| 552710    | 22384     | 20        | 1.65       | 13098      | United Kingdom |
| 552710    | 22378     | 15        | 2.1        | 13098      | United Kingdom |
| 552710    | 22136     | 12        | 1.65       | 13098      | United Kingdom |
| 552710    | 15036     | 36        | 0.83       | 13098      | United Kingdom |
+-----------+-----------+-----------+------------+------------+----------------+
42 rows in set (0.25 sec)
```

Figure 4-10-b. Windows select time (MySQL)

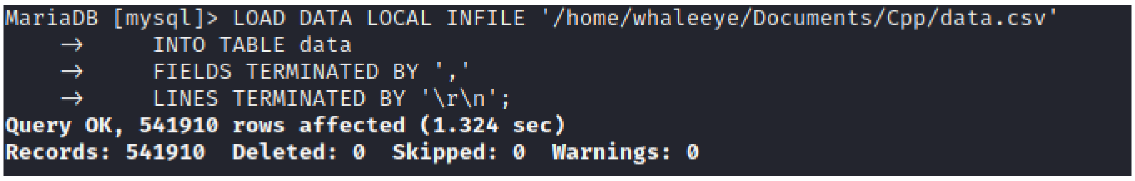


```
mysql> insert into data
    -> values ('664466', '123456A', '30', '3.66', '111111', 'China');
Query OK, 1 row affected (0.01 sec)

mysql> update data
    -> set country='Russian'
    -> where InvoiceNo = '664466';
Query OK, 1 row affected (0.36 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> delete
    -> from data
    -> where InvoiceNo = '664466';
Query OK, 1 row affected (0.36 sec)

mysql> select *
    -> from data
    -> where InvoiceNo = '664466';
Empty set (0.25 sec)
```

**Figure 4-10-c. Windows CRUD time (MySQL)**

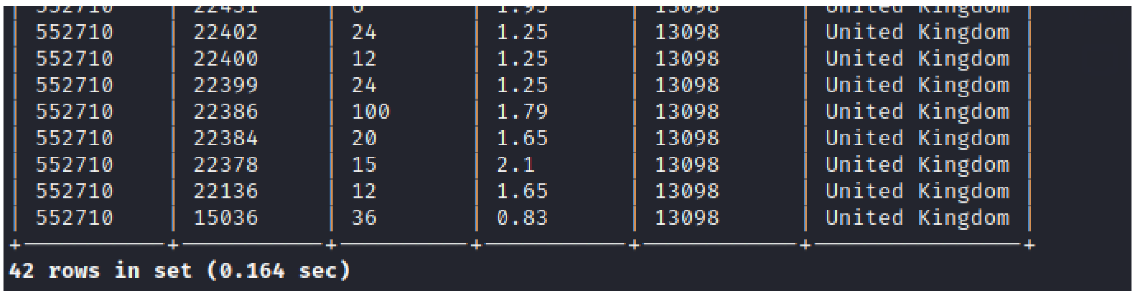
- **Linux**

  The load time (I/O time cost) and CRUD time on Linux are shown in Figure 4-11-a, 4-11-b, 4-11-c.



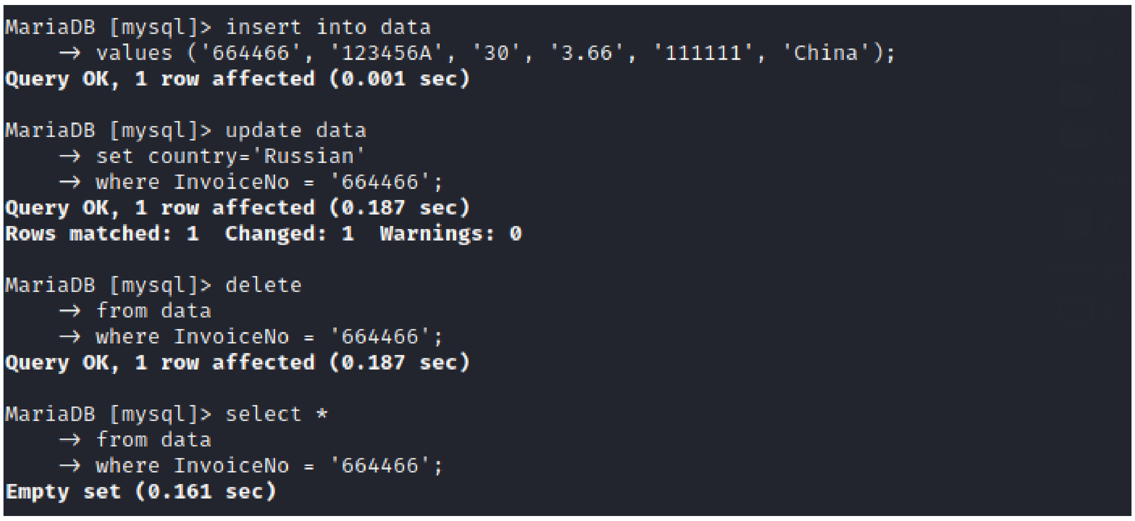
```
MariaDB [mysql]> LOAD DATA LOCAL INFILE '/home/whaleeye/Documents/Cpp/data.csv'
    ->     INTO TABLE data
    ->     FIELDS TERMINATED BY ','
    ->     LINES TERMINATED BY '\r\n';
Query OK, 541910 rows affected (1.324 sec)
Records: 541910  Deleted: 0  Skipped: 0  Warnings: 0
```

**Figure 4-11-a. Linux I/O time (MySQL)**



```
| 552710    | 22402     | 24        | 1.25      | 13098     | United Kingdom |
| 552710    | 22400     | 12        | 1.25      | 13098     | United Kingdom |
| 552710    | 22399     | 24        | 1.25      | 13098     | United Kingdom |
| 552710    | 22386     | 100       | 1.79      | 13098     | United Kingdom |
| 552710    | 22384     | 20        | 1.65      | 13098     | United Kingdom |
| 552710    | 22378     | 15        | 2.1       | 13098     | United Kingdom |
| 552710    | 22136     | 12        | 1.65      | 13098     | United Kingdom |
| 552710    | 15036     | 36        | 0.83      | 13098     | United Kingdom |
+-----------+-----------+-----------+-----------+-----------+----------------+
42 rows in set (0.164 sec)
```

**Figure 4-11-b. Linux select time (MySQL)**



```
MariaDB [mysql]> insert into data
    -> values ('664466', '123456A', '30', '3.66', '111111', 'China');
Query OK, 1 row affected (0.001 sec)

MariaDB [mysql]> update data
    -> set country='Russian'
    -> where InvoiceNo = '664466';
Query OK, 1 row affected (0.187 sec)
Rows matched: 1  Changed: 1  Warnings: 0

MariaDB [mysql]> delete
    -> from data
    -> where InvoiceNo = '664466';
Query OK, 1 row affected (0.187 sec)

MariaDB [mysql]> select *
    -> from data
    -> where InvoiceNo = '664466';
Empty set (0.161 sec)
```

**Figure 4-11-c. Linux CRUD time (MySQL)**

## (4) Analysis

We designed two diagrams shown as Figure 4-12-a and 4-12-b to visualize the data.

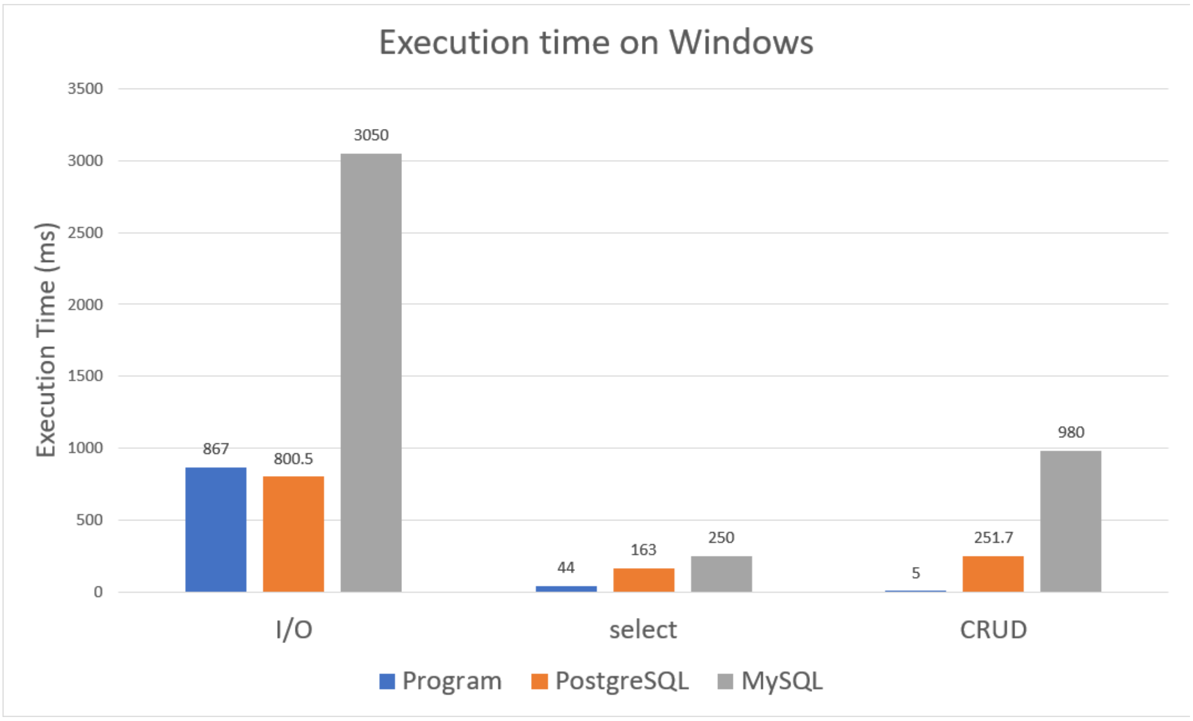


**Figure 4-12-a. Execution time compare (Windows)**

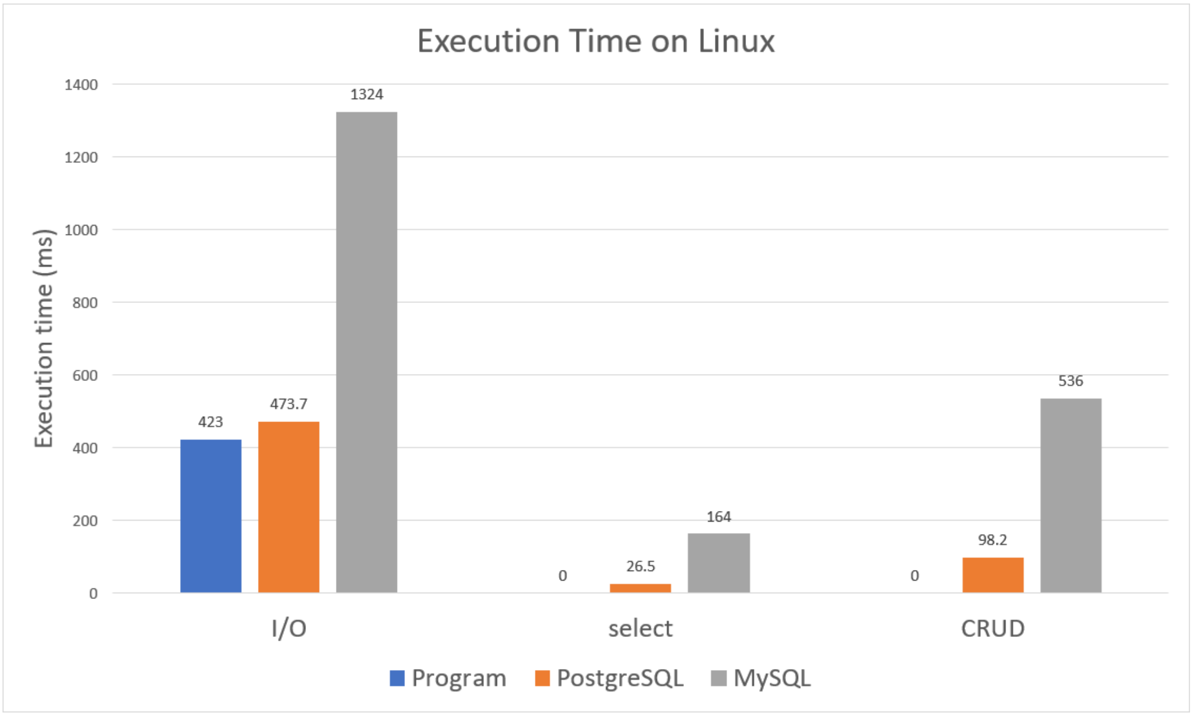


**Figure 4-12-b. Execution time compare (Linux)**

## (5) Compare

Compared with Windows, Linux system's speed is about one time faster on the whole.

- In I/O speed, our program is similar to PostgreSQL, while MySQL is always about 2 times slower than our program.
- In select and CRUD speed, DBMS are much more slower than our program, especially MySQL.

So generally, our program has the highest speed, and MySQL has the lowest speed.

We want to prove that DBMS are more effective, but the result is contradictory to our expectation.

We think there are some possible explanations:

1. For DBMS, there are much more "unnecessary" things to handle while executing a DML statement, like log recording, data organizing and so on to make sure it will be stable and reliable, while our program will just do the "necessary" things: just update, select, delete or insert rows.
2. DBMS need to manage multiple tables and databases at the same time, along with complicated privilege management and schemas, while our program just simulated one table with no schema and a simple privilege management.
3. DBMS might interactive with hard disk during execution, for they need to store the data into disk if there are too much data in buffer, so there might be I/O time cost, while our program will just load the table into memory and interactive with memory all the time.

## 6. Conclusion

DBMS is effective on the premise of full and reliable functions.

Just as what we compared above, DBMS have a lot of excellent designs, such as index, user privilege management and so on. It's meaningless if we just compare the execution time of our program with DBMS if we don't take all of the other functions into consideration.

Anyhow, DBMS have it's own reason for being so classic and existing for such a long time.