

# Database Project1

Tianao Wang,Yancheng Mo

April 17, 2022

## Abstract

In order to help SUSTC, we build a database sustc and take a series of operations on the database. Firstly, We build a E-R Diagram. Then, we design the database to get eight tables. What's more, we import the given data into our tables by a Java script and optimize our script. Finally, we compare the running time of the api of the database and file. Besides, we achieve user privileges management in both database and file manipulation and test our experiments in different operating systems.

## 1 Task1 E-R Diagram

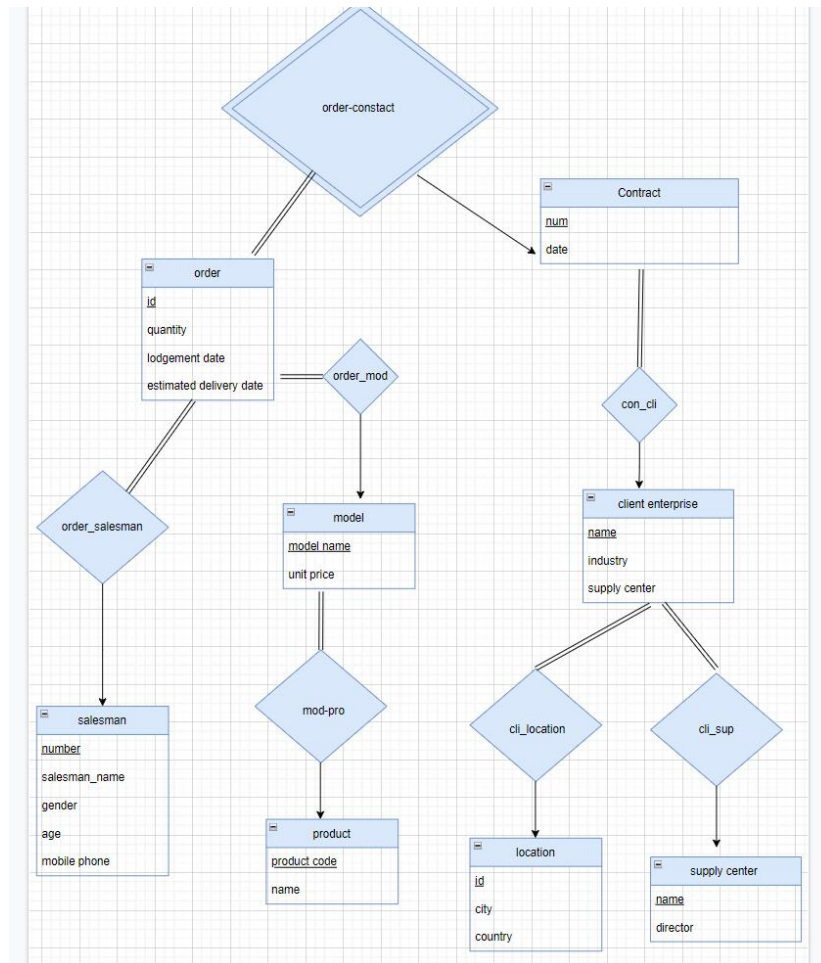


Figure 1: E-R Diagram.

## 2 Task2 Database Design

### 2.1 brief of tables

Our database (SUSTC) contains a total of 8 tables. The following is a basic description of each table:

1.contract: containing information of 5000 contracts. Column 'number' that is primary key indicates the contract number. Column 'contract\_date' means the date of creating the contract. Column 'client\_enterprise' which is a foreign key references client\_enterprise, is the name of the client enterprise that the contract serves.

2.contract\_order: store the order information of each contract, in total 50000. Column 'contract\_number' means the order belongs to which contract that is a foreign key references contract. Column 'id' means the order is the rank in the contract. The column 'id' and the column 'contract\_number' together are as primary key. Column 'model' means the product specific model that the order should supply. Column 'estimated\_delivery\_date' means the estimated date of the delivery of the product. Column 'lodgement\_date' whose element can be null means the actual date of the delivery of the product. Column 'quantity' means the quantity of the product the order need. Column 'salesman\_number' means the number of the each order's salesman that is a foreign key references salesman.

3.product: store product information (including product name and code). Column 'code' is unique identifier of the product which is primary key. Column 'product\_name' means the name of the product.

4.model: store the specific product model information. Column 'model\_name' means name of the model which is primary key to differ from each other. Column 'unit price' means unit price per product model. Column 'product\_node' means the model belongs which product which is foreign key references product.

5.salesman: store each salesman's information. Column 'number' is unique identifier of the salesman which is primary key. Column 'name' means the name of salesman which can be repeated. Column 'gender' means the gender of the salesman. Column 'age' means the age of the salesman. Column 'phone\_number' means the phone number of the salesman.

6.client\_enterprise: store the information of client enterprises. Column 'name' means the name of the name of the client enterprise that is primary key to identify. Column 'industry' means the industry of client enterprise. Column 'location\_id' implies the location of client enterprise which is foreign key references location. Column 'supply\_center' which is a foreign key means the supply center which supply product to client enterprise.

7.location: store the information of location that all the client enterprises located in. Column 'id' is a serial primary key which used to identify different locations. Column 'city' means the city name of the location which can be null. Column 'country' means the country name of the location. The column 'country' and 'city' together is unique.

8.supply\_center: store the information of 7 supply centers and their managers. Column 'center\_name' means the name of supply center. Column 'director' means the manager of the supply center. Both of them is the table's primary key.

### 2.2 The relation of tables

The following Figure2 shows the relation between tables.

We can access every information by the relation. What's more, we also can join them together with the foreign keys to get the total information that is same with the csv file given.

## 3 Task3 Data Import

### 3.1 Overview of Data import principle

When importing the data from the given csv file into postgres database, I directly imported the data using the same method as GoodLoader given by teacher from the beginning, turning off autoCOMMIT and using batch commit and preset commands.

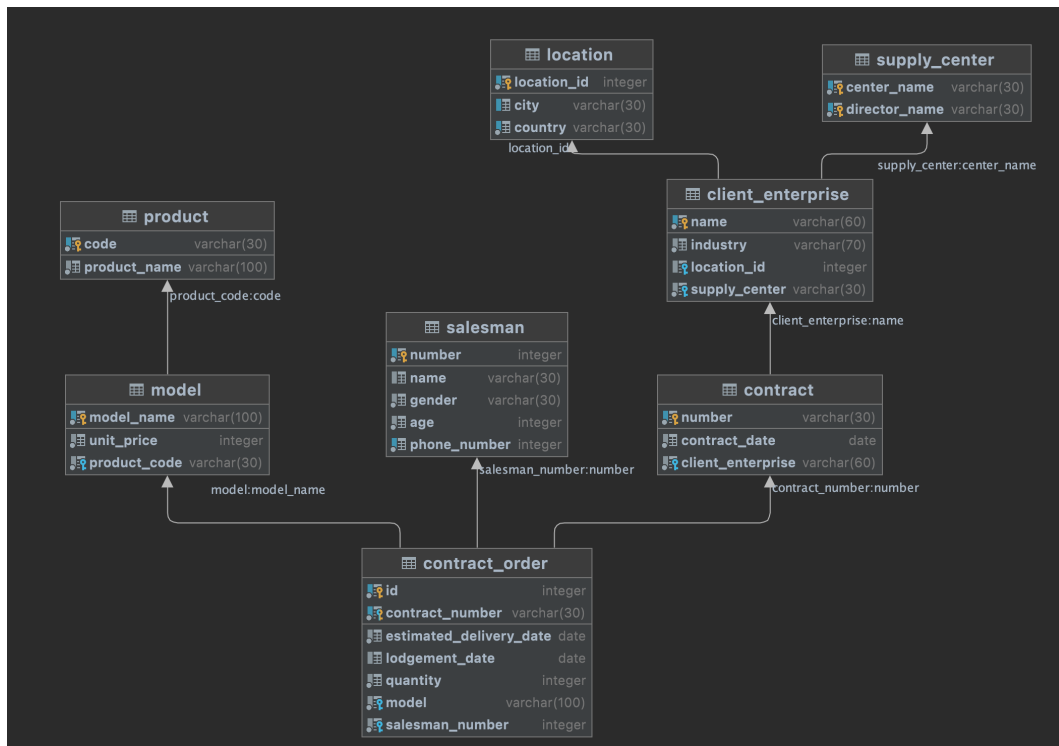


Figure 2: The relation of tables.

### 3.2 Details of the Data import

Firstly, I build 8 PreparedStatement variable, 1 int variable that record the batch size and 1 boolean variable to record whether we successfully connect our database.

#### 1) Build 8 PreparedStatement variables

```

1  private static PreparedStatement productStmt = null;
2  private static PreparedStatement contractStmt = null;
3  private static PreparedStatement orderStmt = null;
4  private static PreparedStatement supplyCenterStmt = null;
5  private static PreparedStatement salesmanStmt = null;
6  private static PreparedStatement modelStmt = null;
7  private static PreparedStatement locationStmt = null;
8  private static PreparedStatement clientStmt = null;
9  private static final int BATCH_SIZE = 500;
10 private static boolean verbose = false;
  
```

Then, write an OpenDB function to connect the database and prepare the SQL statement templates.

#### 2) Establish a connection to the database

```

1  try {
2      Class.forName("org.postgresql.Driver");
3  } catch (Exception e) {
4      System.err.println("Cannot find the Postgres driver. Check
5          CLASSPATH.");
6      System.exit(1);
7  }
8  String url = "jdbc:postgresql://" + host + "/" + dbname;
9  Properties props = new Properties();
  
```

```

9      props.setProperty("user", user);
10     props.setProperty("password", pwd);
11     try {
12         con = DriverManager.getConnection(url, props);
13         if (verbose) {
14             System.out.println("Successfully connected to the database "
15                               + dbname + " as " + user);
16         }
17         con.setAutoCommit(false);
18     } catch (SQLException e) {
19         System.err.println("Database connection failed");
20         System.err.println(e.getMessage());
21         System.exit(1);
22     }

```

### 3) Prepare SQL templates (listing orderStmt as example)

```

1      orderStmt=con.prepareStatement("INSERT INTO contract_order(id,
2      contract_number,estimated_delivery_date,lodgement_date," +
3      "quantity,model,salesman_number)"
4      +" values(?,?,to_date( ? , 'yyyy-mm-dd'),to_date( NULLIF(?, '') , '
5      yyyy-mm-dd')," +
6      "to_number(?, '999999999'),?,to_number(?, '999999999'))");

```

Then we wrote about 8 functions to pass the parameters to our SQL statement templates.

### 4) Pass SQL statement the parameters list method of loadOrderData as example

```

1  private static void loadOrderData(int id, String contract_num, String
2  edate, String ldate, String quantity, String product, String salenum)
3  throws SQLException {
4      if (con != null) {
5          orderStmt.setInt(1, id);
6          orderStmt.setString(2, contract_num);
7          orderStmt.setString(3, edate);
8          orderStmt.setString(4, ldate);
9          orderStmt.setString(5, quantity);
10         orderStmt.setString(6, product);
11         orderStmt.setString(7, salenum);
12         orderStmt.addBatch();
13     }
14 }

```

In the main function, this is basically the case for every table insert, using Java to split the statements and simple processing, and then inserting them into the database. If there are data inserts, run them in batches before closing the database connection.

### 5) list loading Product Data as example

```

1  while ((line = infile.readLine()) != null) {
2      parts = line.split(",");
3      product_code = parts[6].replace(",", "");
4      product_name = parts[7].replace(",", "");
5      loadProductData(product_code, product_name);
6      cnt++;
7      if (cnt % BATCH_SIZE == 0) {
8          orderStmt.executeBatch();
9          orderStmt.clearBatch();
10     }
11 }

```

### 3.3 SQL Optimization

We find the column 'city' and 'lodgement\_date' can be null. So we should replace the "NULL" and "" given by the csv file. At first, I use *update* SQL statement. But I find if update. Then I change into *NULLIF* SQL function. Take the 'city' as example below:

#### 1) Before

```
1 "INSERT INTO location(city, country)"+ " values(?, ?)"+  
2 "ON CONFLICT(city, country) DO UPDATE set (city, country)=(null  
    , excluded.country) where excluded.city='NULL';"
```

#### 2) After

```
1 "INSERT INTO location(city, country)"+ " values(NULLIF(?, 'NULL'), ?)"+  
2 "ON CONFLICT(city, country) DO NOTHING;"
```

The result is that the load time reduce from about 15s into about 7s. Data importing speed almost be twice as fast.

### 3.4 Preprocessing of insert data

In order to speed up the data import, I preprocess some data that should be import into our tables. For example, The primary key of the order is both the order id and the contract\_number, so I should make id by myself. What's more, take the location as an example. The location id is serial. However I use the *NULLIF* SQL function to speed up, and then many "NULL" will be replaced into *null* to insert into our location table. But null can't be judge whether the location is repeated. So I will count the location whether is repeated by myself.

First I make class named locationDI.

#### 1) class

```
1 class locationDI{  
2     String country;  
3     String city;  
4     public locationDI(){  
5         country=null; city=null;  
6     }  
7     public locationDI(String city, String country){  
8         this.country=country;  
9         this.city=city;  
10    }  
11 }
```

Then I make an ArrayList to store the location has been recored.

#### 2) Arraylist

```
1 ArrayList<locationDI> location=new ArrayList<>();  
2 location.add(new locationDI());
```

Finally use a boolean variable and following code to import into the loaction table.

#### 2) main function

```
1 boolean locationNoRepeat=true;  
2 for (int i=0; i<location.size(); i++) {  
3     if (city.equals(location.get(i).city) && country.equals(location.get(  
4         i).country)) {  
5         locationNoRepeat = false;  
6         break;  
7     }  
8 }
```

```

6      }
7    }
8    if (locationNoRepeat){
9      loadLocationData(city, country);
10     location.add(new locationDI(city, country));
11   }

```

Based on it, I also make the id at the same time to import into our client\_enterprise table. I will not list it, if you want more details you can look my code.

### 3.5 Result of the Data import

By using the method mentioned above, our Data import speed is about 9000 records/s and total time about 5.5 s inserting all data given by the csv into our 8 tables at the same time finally.

The following shows our result:

```

Successfully connected to the database sustc as checker
50000 records successfully loaded
Loading total time : 5.264 s
Loading speed : 9498 records/s

进程已结束，退出代码为 0

```

Figure 3: The result of data importing.

### 3.6 Brief introduction of other ways to import data

#### 3.6.1 Datagrip + Separate table operations

First, use Datagrip to import all the information. And then update the "NULL" and "" in the tables to null. Finally, use Java script to divided into our 8 tables. Obviously, it much more complex than our method. I will list the datagrip operation as examples as follows. We can find the time not become less.

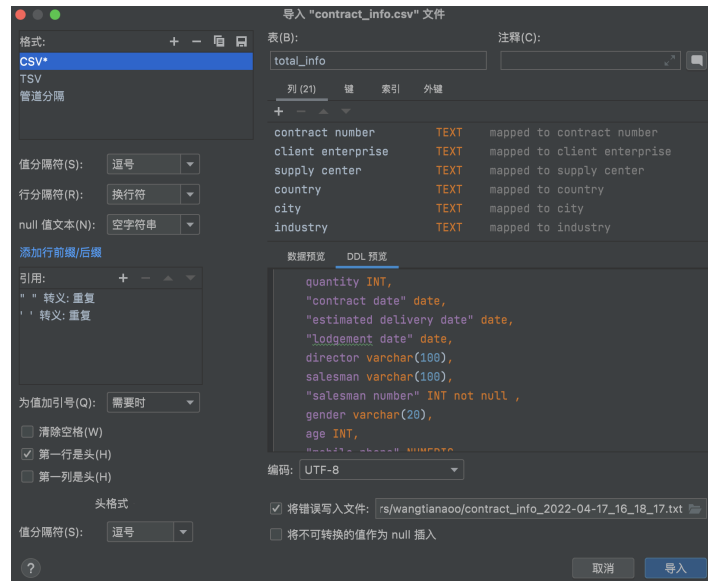


Figure 4: Operation.

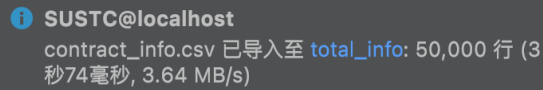


Figure 5: Result.(3.74s)

### 3.6.2 Another Java scrip + Separate table operations

There is a COPY statement. Use the statement:"COPY table\_name FROM STDIN;" In Java, we can set the local client files to be imported by setting streams.

#### 1) core code

```

1 public void copyFromFile(Connection connection, String filePath, String
   tableName)
2     throws SQLException, IOException {
3     FileInputStream fileInputStream = null;
4     try {
5         CopyManager copyManager = new CopyManager((BaseConnection)
           connection);
6         fileInputStream = new FileInputStream(filePath);
7         copyManager.copyIn("COPY " + tableName + " FROM STDIN",
           fileInputStream);
8     } finally {
9         if (fileInputStream != null) {
10             try {
11                 fileInputStream.close();
12             } catch (IOException e) {
13                 e.printStackTrace();
14             }
15         }
16     }
17 }

```

According to test, although it faster than the import speed of DataGrip itself, we also think it's too complex to import the data step by step.

### 3.6.3 Python

1. Using executemany() method batches data into the database, the same principle as our method mentioned above.

#### 1) example core code

```

1 def batchInsert(sql, data, size):
2     try:
3         psycopg_pool = PooledDB(psycopg2, mincached=5, blocking=True,
           user='checker', password='123456', database='sustc', host=
           'localhost', port='5432')
4         connection = psycopg_pool.connection()
5         cursor = connection.cursor()
6     except Exception as e:
7         print(e)
8     try:
9         cycles = math.ceil(data.shape[0] / size)
10        for i in range(cycles):
11            val = data[i * size:(i + 1) * size].values
12            cursor.executemany(sql, val)

```

```

13         connection.commit()
14     except Exception as e:
15         print(e)
16         connection.rollback()
17     finally:
18         connection.close()

```

(Data: dataframe data, size: batch size, sql: SQL statement.)

2. Insert data into the database using to\_sql of the dataframe.

#### 2) example core code

```

1 from sqlalchemy import create_engine
2 result = pd.DataFrame(data)
3 engine = create_engine('postgresql://user:password@host:port/database
4                          ')
5 pd.io.sql.to_sql(result, table_name, engine, index = False, if_exists
6                  ='append')
7 pd.io.sql.to_sql(result, table_name, engine, index = False, if_exists
8                  ='replace')

```

3. copy\_from() function of PostgreSQL, the same principle as the java another scrip mentioned above.

## 4 Task4 Compare DBMS with File I/O

### 4.1 test environment

#### 4.1.1 Hardware specification

cpu: Inter(R) i7-10875H/Inter Core i5  
memory: solid-state disk

#### 4.1.2 Software specification

DBMS: postgresql  
programming language: java operating systems:Linux(Ubuntu)/windows/Macos environment: JDK  
11 /gson-2.9.0.jar /postgresql-42.2.5.jar

### 4.2 data organization

#### 4.2.1 organize data in database

We use the same database in the Task 3, which is provide by this project and was talked about in Task 1 and Task 2. The DDL statement could be found in the attachment "load.sql"

#### 4.2.2 organize data in file

First of all, we use plain text formats JSON to store the data.

In order to achieve the same structure in the database, we design 8 classes corresponding the 8 tables we design in the Task 2. In each class, we have a member variable corresponding to the name of the column and each instance of these classed is corresponding to row in the database. To better organize data, we design a class named "JsonDataCtrl" to control all the data in the file. Using "JsonDataCtrl" class to manipulate data could ensure the same prime key and foreign key in the database.

In the "JsonDataCtrl" class, we have 8 ArrayLists contain all the instances for the eight classes, which contain all the data we need to use. In this class, we provide methods to load JSON/CSV data



from file or save data to JSON format. What's more, we could use this class to create new data and make accounts for exist users(director/salesman). And this class is also used as powerful tool for data manipulation for providing many methods to operate data.

## 4.3 SQL script and program

### 4.3.1 SQL script

We test the following SQL script in java and realize same function by file IO

1)query the max id for an certain contract

```
1 private String MaxIdSql="select max(id) from (\n" +  
2 "select * from contract_order\n" +  
3 "where contract_number=?) sub";
```

2)query all the information in order

```
1 select * from contract_order
```

In order to test the time of easy query, we design this SQL

3)query all the information in order

```
1 private String SelectOrderByProSql = "select id,contract_number from (( "  
2 +  
3 "      select id,contract_number,model_name,product_code " +  
4 "      from contract_order co join model m on co.model=m.model_name)  
5 sub1 " +  
6 "      join product p on sub1.product_code=p.code) " +  
7 "where product_name=? ";
```

In order to test the time of JOIN, we design this SQL

4)query the num of orders in each country

```
1 private String SelectOrderByCountrySql="select country,count(*) as cnt\n"  
2 +  
3 "from contract_order co\n" +  
4 "inner join (\n" +  
5 "      select number,country from contract c\n" +  
6 "inner join (\n" +  
7 "      select name,country from client_enterprise\n" +  
8 "      inner join location l on client_enterprise.location_id = l.  
9 location_id\n" +  
10 "      ) sub1\n" +  
11 "on sub1.name=c.client_enterprise\n" +  
12 "      )sub2\n" +  
13 "on co.contract_number=sub2.number\n" +  
14 "group by country";
```

In order to test the time of GROUP BY, we design this SQL.

5)query all the order between two dates

```
1 private String SelectOrderByEDateSql="select id,contract_number from  
2 contract_order" +  
3 " where estimated_delivery_date>=? and "+  
4 "estimated_delivery_date<=?"
```

In order to test the time of Compare data, we design this SQL.

#### 6)query the order belong to self(for salesman/director)

```
1 private String queryDirectSelfOrderSql ="select id,contract_number from
   contract_order\n" +
2 "inner join (\n" +
3 "select number from contract\n" +
4 "inner join (\n" +
5 "select name from client_enterprise\n" +
6 "inner join (\n" +
7 "select center_name from supply_center\n" +
8 "where director_name=?) sub1 on client_enterprise.supply_center=sub1.
   center_name) sub2\n" +
9 "on contract.client_enterprise=name) sub3\n" +
10 "on contract_order.contract_number=sub3.number";
11 private String querySalesmanSelfOrderSql="select id,contract_number,s.
   number from contract_order\n" +
12 "inner join salesman s on s.number = contract_order.salesman_number\n" +
13 "where s.number=?";
```

In order to test the user privileges management, we design these SQL.

### 4.3.2 program

In order to test program better, we design a base abstract named "DataManipulation", and we achieve two derived classes named "FileManipulation" and "DatabaseManipulation", which use different ways to operate the data. What's more, we use a data factory named "DataFactory" to get the instance of these two derived classes.

In the class "client", we simulate the behavior of the real client and test the running time for each operator. On the other word, we use the class "client" to finish all the comparative study and experiments.

For managing the user privileges, we design a class named "User" according the real user account and "UserCtrl" to control the instances of class "User". We achieve all the function of user privileges management in the base class "DataManipulation" to get the reuse of the code. And the class "UserCtrl" is set as a member variable for the class "DataManipulation" to management user privileges. In class "Client", we also do the privileges test to make sure the user privileges management is achieved.

There are four different type of users: Nothing/root/director/salesman

Nothing user could do nothing without any privileges.

Root user could do anything with all the privileges.

Director user could query/add/delete/alter own order.

Salesman user could query own order.

## 4.4 comparative study and experiments

In this part, we will design experiments to compare the runing time between database api and our own file manipulation on the SQL we talk about above.

### 4.4.1 import data from csv

In loading data from csv file, the database and file manipulation will all analysis the data and operate quite number of add manipulation, which will handle the prime key and for foreign key. So, it is meaningful to compare this. We could find out that the file manipulation is faster than database manipulation

### 4.4.2 query all the information in the order

We want to compare the select operation without limitation. On the other word, we could compare performance of simply query.

load from csv							
window/java				Mac/java		Linux/java	
file	db/pgsql			file	db/pgsql	file	db/pgsql
4425 ms	6865 ms			3346 ms	5264 ms	5272 ms	22667 ms

Figure 6: loadFromCsv

select * from order							
	window/java			Mac/java		Linux/java	
	file	db/pgsql		file	db/pgsql	file	db/pgsql
50k	35 ms	95 ms		29 ms	83 ms	47 ms	251 ms
500k	291 ms	751 ms		170 ms	532 ms	226 ms	885 ms
1000k	307 ms	1488 ms		438 ms	1642 ms	1162 ms	1461 ms

Figure 7: select all

#### 4.4.3 query orders by estimatedDeliveryData

We want to measure the select operation with comparison. In this part, the query only contains simply query and comparison, which could be used to test the performance of data comparison.

query orders by estimated_delivery_data							
	window/java			Mac/java		Linux/java	
	file	db/pgsql		file	db/pgsql	file	db/pgsql
50k	17 ms	8 ms		31 ms	13 ms	24 ms	10 ms
500k	283 ms	69 ms		1025 ms	63 ms	453 ms	84 ms
1000k	488 ms	160 ms		2362 ms	280 ms	1293 ms	168 ms

Figure 8: compare

#### 4.4.4 query order by product\_name

We want to compare the select operation with many join operations, which could measure the performance of hard query

#### 4.4.5 count orders by country

We want to compare the select operation with many join operations and the Aggregate function group by and count(), which could measure the performance of a harder query (in figure 10)

#### 4.4.6 alter orders by adding quantity by one

We want to measure the performance of altering the value of a certain table. In order to ignore the difference coming from the query, we design a easy alter SQL to run a large number of times to get the real difference about alter data between database and file.(in figure 11)

#### 4.4.7 remove orders before(LDate)

We want to measure performance of deleting data from a certain table between database and file. In order to ignore the difference coming from the query, we design a easy delete SQL without too much query.(in figure 12)

#### 4.4.8 add data to order

We want to measure performance of adding new data to a certain table. In order to ignore the difference coming from the query, we design a easy insert SQL to run a large number of times to get the real difference about alter data between database and file. (in figure 13)

query order by productName							
window/java				Mac/java		Linux/java	
	file	db/pgsql		file	db/pgsql	file	db/pgsql
50k	82 ms	8 ms		95 ms	9 ms	92 ms	7 ms
500k	809 ms	225 ms		731 ms	50 ms	901 ms	92 ms
1000k	1628 ms	241 ms		1578 ms	65 ms	1591 ms	225 ms

Figure 9: qbyProduct

count orders by country							
window/java				Mac/java		Linux/java	
	file	db/pgsql		file	db/pgsql	file	db/pgsql
50k	617 ms	28 ms		1010 ms	29 ms	653 ms	26 ms
500k	7301 ms	179 ms		7381 ms	112 ms	7032 ms	109 ms
1000k	11871 ms	310 ms		14942 ms	221 ms	13626 ms	266 ms

Figure 10: qbycountry

#### 4.4.9 data analysis

**difference between different operating systems** From the data we get above, we could find there are not too much difference among these three operating systems when the data is small. But when the data grows bigger, there are still not obvious difference between Windows and MacOS. But about the Linux, the performance is quite slower in same function which is above our expectation. After analysis, we think is difference is coming from the difference between CPU and SSD. For the Linux is base on a virtual machine with limited CPU and memory. So, we drew a conclusion: There are not obvious difference between Windows and MacOS. In the future, if we get a real machine with Linus, we will finish this experiments.

**difference between database and file** In this part, we will only focused on the Windows. We drew some pictures to get the data visualization. First of all, we could find a huge advantage for file in adding/removing/altering data in the table, especially when the data is huge(according to the Figure 11/Figure 12/Figure 13). I think the most important reason is that we manipulate file data in the RAM memory and it is easily to modify, but the I/O to the SSD is needed for the database. In this condition, file manipulation for adding/removing/altering could be fast but volatile, which is also limited by the size of memory. And the database could be safer and without the worry about memory, but the speed must be slower.

On the second part, we will compare the query performance between database and file in different data size and to achieve different functions. From Figure 14, we could make a good observation to the data we get. There are three lines in the figure corresponding to three different data size. The y-axis corresponding to the running time of a certain query while the x-axis corresponding to different types of the query .

We could find when the data is small, all the query is fast, and when the data grows bigger the difference between querying is becoming bigger. We could find out all the time is positive correlation to the data size and when the data grows double all the running time about is double. It is apparent for the the file since all the query we achieve by file is  $O(n)$ . So, we could infer that, without index, all the query achieve in the database is  $O(n)$  by our experiment.

What's more, we could find out that when the query is easy, the file is much faster than the database. What is not as expected is that query with easy comparison need less time than the easy select \*. What we could explain is that the query is so easy that the time it need is quite small and the main part of time consumption is the transportation of data. Without the need of the transportation of data, file will have a better performance.

And about the hard and harder query, the performance of database is better than file remarkably. In this part, we could find the superiority for database on dealing with complex query. We could find when we use "join" many times with "group by", the running time of file will become unacceptable.

alter orders by adding quantity by one							
	window/java			Mac/java		Linux/java	
	file	db/pgsql		file	db/pgsql	file	db/pgsql
50k	1 ms	562 ms		4 ms	597 ms	4 ms	361 ms
500k	10 ms	6098 ms		8 ms	6190 ms	8 ms	3533 ms
1000k	11 ms	12333 ms		11 ms	14287 ms	9 ms	13834 ms

Figure 11: alterByOne

remove orders before(LDate) 2019-01-01							
	window/java			Mac/java		Linux/java	
	file	db/pgsql		file	db/pgsql	file	db/pgsql
50k	22 ms	19 ms		45 ms	24 ms	53 ms	23 ms
500k	235 ms	347 ms		161 ms	583 ms	198 ms	265 ms
1000k	259 ms	1442 ms		366 ms	1299 ms	251 ms	1919 ms

Figure 12: remove

## 5 Reference

Thanks for the following website for us to study :

[1]<https://www.cnblogs.com/sunxun/p/6674430.html>

[2][https://blog.csdn.net/weixin\\_40746796/article/details/94428585](https://blog.csdn.net/weixin_40746796/article/details/94428585)

add new data							
	window/java		Mac/java		Linux/java		
	file	db/pgsql db/mysql	file	db/pgsql	file	db/pgsql	
450k	11466 ms	132083 ms	14455 m	153535	13743 ms	373211 ms	
950k	24414 ms	241468 ms	31102 m	275695	30974 ms	648943 ms	

Figure 13: addNewData

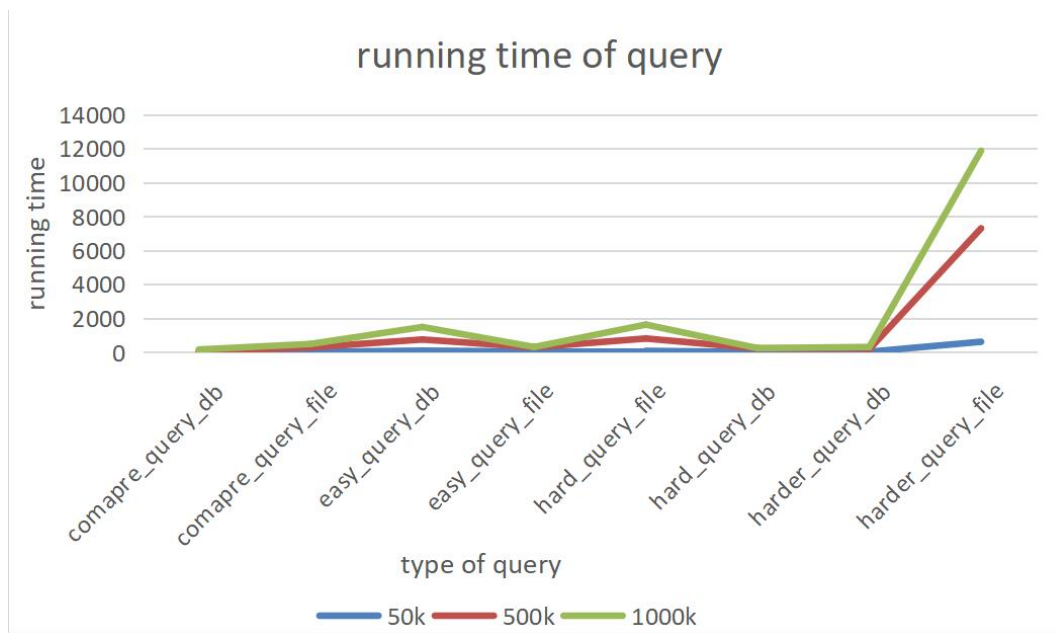


Figure 14: query