



Report for Project One of CS307

Basic Information

Team of 12010903 林雨航, 12012338 曾宪清 in Lab 2.

	林雨航	曾宪清
Task 1	Design all tables and relations	Check the design back to csv file
Task 2	Draw the first vision of E-R diagram	Make suggestions for revision
Task 3	Import using python	Import using Java
Task 4	Operate using Java	Operate using python
Score	50%	50%
Hardware	设备规格 设备名称 LAPTOP-EOE1H7A 处理器 AMD Ryzen 7 4800H with Radeon Graphics 2.90 GHz 机带 RAM 16.0 GB (15.4 GB 可用) 设备 ID 产品 ID 系统类型 64 位操作系统, 基于 x64 的处理器 笔和触控 没有可用于此显示器的笔或触控输入	设备名称 DESKTOP-KOBRUV5 处理器 Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz 2.59 GHz 机带 RAM 8.00 GB (7.80 GB 可用) 设备 ID 产品 ID 系统类型 64 位操作系统, 基于 x64 的处理器 笔和触控 没有可用于此显示器的笔或触控输入
Python version	3.10	3.6.3
Java version		

Software:

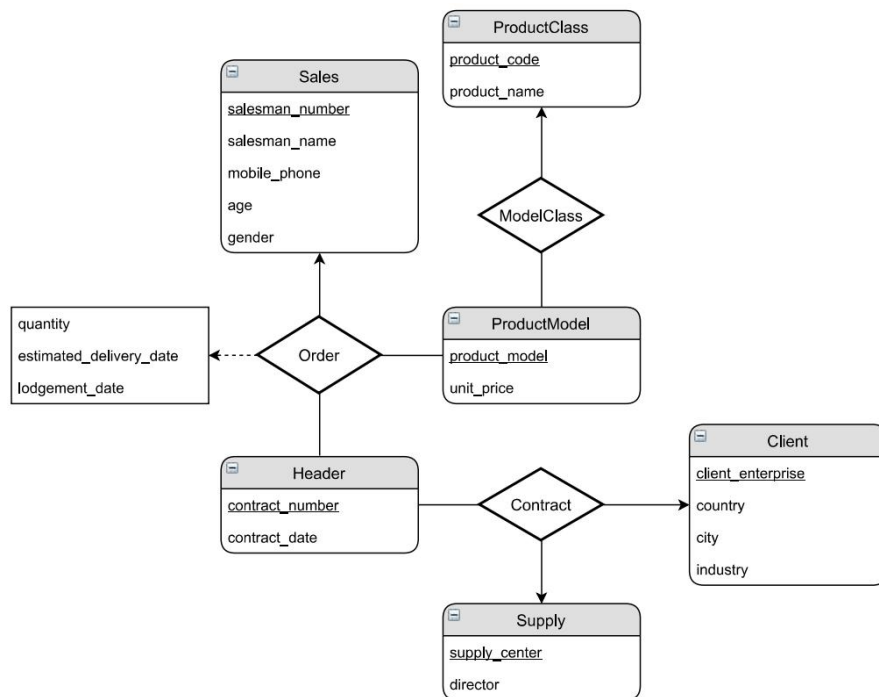
psycopg2	2.9.3
Pandas	1.1.5

Task 1: E-R Diagram

I followed instinct and made a very different design from the E-R diagram above at beginning. But it is hard to draw the E-R diagram for it. Then I find that textbook shows the translation from E-R diagram to relationship in section 6.7, so I think I should make E-R diagram first then translate E-R into postgresql code, rather than make database design first then draw a E-R for it. That is, task2 should be done at the same time with task 1. Hence, I start all over again.

I find a very useful theory in section 6.9.1: if we can use the primary key of an entity set as an attribute of another entity set, then there must be a relationship set between two entity sets. Following this theory, I design the E-R diagram below.

The diagram is generated from website <https://www.freedgo.com/>, on 2022/4/4.



In section 6.5.2 of textbook, it writes that “we permit only one arrow out of a nonbinary relationship set”. Hence the drawing of “Contract” and “Order” above is incorrect in theory.

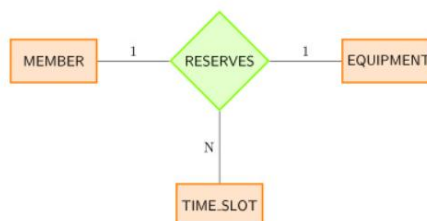
What needs to be emphasized is that I do have tried to solve this problem.

In section 6.5.2, textbook says “A simpler approach is to use functional dependencies, which we study in Chapter 7 (Section 7.4)”, but after some attempt to reading materials at section 7.4, I found it too difficult to learn functional dependency only by textbook.

Also in section 6.5.2, textbook suggest to “treat each instance of the non-binary relationship set as an entity”, but the book does not explain the drawing method of connecting two tables directly by lines in E-R, nor explain whether this drawing is legal.

In section 6.9.4, textbook offers an example to transmit a many-to-many ternary relationship into three binary many-to-one relationship. But after thinking, I think this method can't be used in my problem. Textbook also writes “There may not be a way to translate constraints on the ternary relationship into constraints on the binary relationships” and gives an many-to-one example later, which is like my problem.

So I asked this problem in QQ group. Yuxin Ma says just need to put mapping cardinality on the line, like examples in slides. Later he offers an example of drawing on one website:



Since the cardinality of 1 can be replaced by a directed line, then I maintain the drawing method at the beginning (the E-R diagram above). But to prevent ambiguity, I still decide to explain each relationship set in my E-R diagram:

model_class: An entity in *product_class* is associated with any number (zero or more) of entities in *product_model*. An entity in *product_model*, however, can be associated with at most one entity in *product_class*.

order: A particular combination of entities from *header*, *product_model* can be associated with at most one combination of entities from *sales* and *attached information*. A particular combination of entities from *sales* and *attached information* can be associated with any number of combinations of entities from *header* and *product_model*. Thus, the primary key for the relationship *Order* is the union of primary keys in *header* and *product_model*.

contract: A particular entity from *header* can be associated with at most one combination of entities from *client*

and *supply*. A particular combination of entities from *client* and *supply* can be associated with any number of entity from *header*. Thus, the primary key for the relationship *contract* is the primary key of *header*.

Task 2: Database Design

Because of our special thinking about task1 and task2, the work of task2 has already been done in task1, especially primary keys for each relationship. The rest of this phrase is just to translate E-R diagram into postgresql code. All entity sets with their primary keys are shown in E-R diagram clearly, and all relationship set with their primary keys have been explained in task 1, so no more repetition here. Instead, we introduce our thought about E-R diagram.

Looking into **contract_info.csv**, it can be clearly seen that this table consists of four main parts:

Meaning of table	Content
unique contract information	contract number, contract date
supplier in SUSTC	supply center, director (this relationship is pointed out in the project requests)
client who purchases goods in SUSTC	client enterprise, city, industry, country
salesman for client	salesman number, salesman name, gender, age, mobile phone
purchased product	product code, product name, unit price

Combining the contract in real life, we created a relationship set to connect these parts. (The connected part does not include salesman because we do not find explicit relation between salesman and contract number in **contract_info.csv**)

Meaning of table	Content
relationship set to connect contract, supplier and client	contract number, supply center, client enterprise

Then we look **contract_info.csv** more carefully and find that one contract number can be associated with more than one same product code, but only associated with one different product model. Also, unit price may be different for different model even these models belong to the same product code. So, we break "purchased product" into two more smaller part, and add one relationship set to connect them:

Meaning of table	Content
the specific model of product	product model, unit price
the abstract classification of product	product code, product name
relationship set to connect them	product model, product code

Now look back at what have not been classified: the information of salesman, quantity of products, estimated delivery date and lodgement date of product. Among these entities, quantity of products is associated with product model in each contract number. So we create another relationship set to connect all remains together:

Meaning of table	Content
the order in one contract	product model, contract number, salesman number, quantity, estimated delivery date, lodgement date

Now, all of design of database is finished, corresponding resultant E-R diagram is shown in task 1. After checking, all of designs meets the requirements in project file.

Easy to see, most of type is *varchar*. The exceptions are below:

date : contract date, estimated delivery date, lodgement date

int : unit price, quantity, age (of the salesman)

Since only city and lodgement may be NULL, so we set all other attributes to NOT NULL.

To be more rigorous, we set maximal length for every varchar. For some fixed-length variable like phone number, we set its type into char. Most variables, like country, do not have an explicit maximal length, we sort the content of this variable in **contract_info.csv** in the order of decreasing length to find the maximal length in the table. Then we look back at the code examples in each lab class. Combine **contract_info.csv** and lab together, we set the maximal length for each variable as below.

Variable	Type
contract number	char(10)
client enterprise	varchar(100)
supply center	varchar(30)
country	varchar(50)
city	varchar(20)
industry	varchar(50)
product code	varchar(7)
product name	varchar(100)
product model	varchar(100)
unit price	integer
quantity	integer
contract date	date
estimated delivery date	date
lodgement date	date
director	varchar(30)
salesman name	varchar(50)
salesman number	char(11)
gender	varchar(6)
age	integer
mobile phone	char(11)

Task 3: Data Import

Methods in video

To complete this task, I firstly reference the “data-import-material” offered in sakai. In the video, 4 optimization methods are summarized to improve the efficiency of importing data:

1. only connect and close database once
2. use the *preparestatement* instead of *statement*
3. Batch processing mechanism
4. commit only once after all statements have been executed.

However, we cannot get each column in and insert required information into 9 tables designed by ourselves, since it will lead to the Error: “Duplicate key violates unique/primary constraint”. To avoid such error, I split **contract_info.csv** into 9 corresponding **.csv** files as required in 3 steps.

1. To get accessed to all given data, I created a corresponding table *contract_info* in database, which has the same 20 columns. Based on the above improvement, I wrote *CSVLoader.java* to import the whole data into the table *contract_info* reference to the given *goodloader.java*. Within the *loadData* method, there are two particular points to note:

① if the city is "NULL" or the lodgement date is blank in files, we should set them to *null*

```
1.  if (parts[2].equals("NULL"))
2.      stmts[5].setNull(3, Types.VARCHAR);
3.  else
4.      stmts[5].setString(3, parts[2]);
```

② the date information should be transformed from *String* into *java.sql.Date*.

```
1.  java.sql.Date cont_date = new java.sql.Date(new java.util.Date(parts[1].replace("-", "/")).getTime());
```

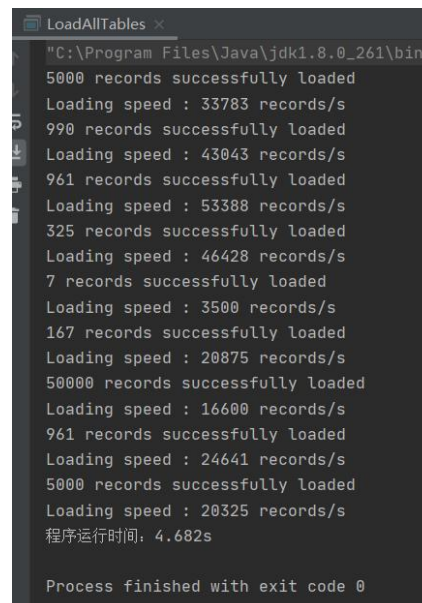
2. Call the *lcopy* command in psql to get 9 splited files.



```
SQL Shell (psql)
Server [localhost]: localhost
Database [postgres]: sustc
Port [5432]: 5432
Username [postgres]: checker
用户 checker 的口令:
psql (14.2)
输入 "help" 来获取帮助信息.

sustc=# \copy (select distinct contrac_number, contract_date from contract_info order by contrac_number ASC ) to
'C:\Users\21414\Documents\Curriculum\database\project01\model_class.csv' with csv;
COPY 5000
sustc=#
```

3. Write *LoadAllTables.java* to import 9 files obtained in step 2 into corresponding tables. The actual time cost is as followed:



```
LoadAllTables x
"C:\Program Files\Java\jdk1.8.0_261\bin\
5000 records successfully loaded
Loading speed : 33783 records/s
990 records successfully loaded
Loading speed : 43043 records/s
961 records successfully loaded
Loading speed : 53388 records/s
325 records successfully loaded
Loading speed : 46428 records/s
7 records successfully loaded
Loading speed : 3500 records/s
167 records successfully loaded
Loading speed : 20875 records/s
50000 records successfully loaded
Loading speed : 16600 records/s
961 records successfully loaded
Loading speed : 24641 records/s
5000 records successfully loaded
Loading speed : 20325 records/s
程序运行时间: 4.682s

Process finished with exit code 0
```

Another method in Java

After successfully importing data, I tried to achieve in other ways.

1. Firstly, I consider improvement in file reading. The original *goodloader.java* apply *BufferedReader* to get accessed to the file. Thus I rewrote *CSVLoader.java* to get *LoadWithCSVpackage.java* with importing *com.csvreader.CsvReade* and get file input via *CsvReader*. The import efficiency of two program are showed as followed, however, there is no much difference.

```
"C:\Program Files\Java\jdk1.8.0_261\bin\java.exe" ...
50000 records successfully loaded
Loading speed : 19098 records/s
程序运行时间: 2.619s
```

CSVLoader.java

```
"C:\Program Files\Java\jdk1.8.0_261\bin\java.exe" ...
50000 records successfully loaded
Loading speed : 19275 records/s
程序运行时间: 2.598s
```

LoadWithCSVpackage.java

2. Since the execute \copy command in psql is executed very rapidly, I considered implement the command in java program with CopyManager class.

```
1. file = new FileInputStream(filename);
2. String copy = "COPY " + "contract_info" + " FROM STDIN DELIMITER AS ','";
3. copyManager.copyIn(copy, file);
```

The runtime cost is as followed:

We can see that the speed of single operation and the whole runtime are both improved.

```
"C:\Program Files\Java\jdk1.8.0_261\bin\java.exe" ...
50000 records successfully loaded
Loading speed : 22114 records/s
程序运行时间: 2.262s
```

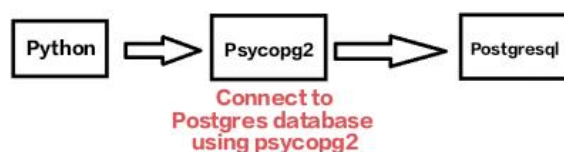
(All the above programs are timed from the first statement of the main method until the last statement)

Some methods in Python

Using psycopg2

Besides Java, we also tried some method in python.

According to the psycopg2 document, Psycopg is the most popular PostgreSQL database adapter for the Python programming language. And its main features are the complete implementation of the Python DB API 2.0 specification and the thread safety. Thus we try to connect to database and import data with psycopg2 firstly.



The most basic way to import data from csv into postgresql database is inserting the rows one by one using the function *execute()* in package *psycopg2*. Program need to transform data into memory, and then import it into database. And the constant context switching between the program and the database must slow it down.

A faster way to insert is *executemany()*, a function allows us to execute more than one sql statement in one time. This function will transform all of data into memory, and only then import then into database, so it will save the time of transformation. But this improvement is not satisfying.

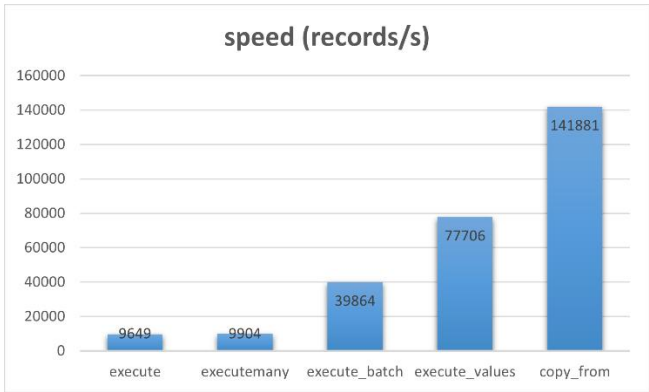
Another faster way is `execute_batch()`, a function in `psycopg2.extras`. This function can execute groups of statements in fewer server roundtrips, which saves time for server roundtrips. This is quite a lot improvement, it can treble import speed. But this is not the fastest way.

One more faster way is `execute_values()`. This function works by generate a huge VALUES object list to the query. VALUES make this function works more rapidly than batch.

	1/time	many/time	fewer server roundtrip	VALUES object
<code>execute()</code>	√			
<code>executemany()</code>	√	√		
<code>execute_batch()</code>	√	√	√	
<code>execute_values()</code>	√	√	√	√

However, VALUES is still not the fastest way. As have said before, according to document of postgresql, the `copy` command is the fastest way to import data from csv into database. The document just says that it is optimized for loading large numbers of rows, not mentioning any deep theory about it. I only find that this function uses buffer to import data.

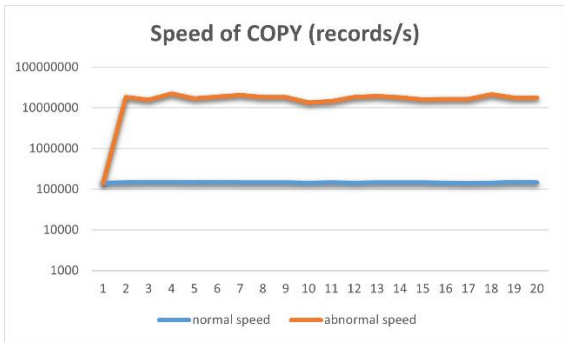
To compare speed of these five functions, I write a test python program. Each function will be repeated 100 times and print the average speed. All five functions will import data from one same file named `header.csv`, which containing 5000 lines. The page size (which will explain later) of `execute_batch()` and `execute_values()` are both 100.



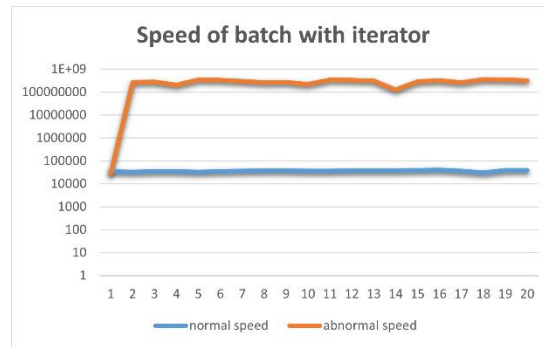
Confusion: before start, I find that the speed of `copy` will become extremely high, even more than ten billion records per second. Only the first `copy` has normal speed ($\approx 140,000$). If open csv file just before `copy`, that is, open file 100 times and `copy` 100 times, then speed for each `copy` is normal. If only open file 1 time, then execute `copy` without open again, then the speed will become too huge. I don't understand this situation.

In the above speed comparison, the first-time normal speed of `copy` is tested.

The below figure shows 20 continual speed test of normal and abnormal `copy`.



In python, there is a special object called iterator. If we use iterator to store data, and import data from iterator rather than csv file, the same confusion will occur again. In continual tests, only the first result is normal, the other result is even more extreme than `copy`, that is, more than 100 billion! This amplification amount is also more than `copy`. Below figure shows the normal and abnormal result of continual 20 tests of iterator in `execute_batch()`, both have page size 100 (explain later).



I have learned in CS202 that, time to access data in register is thousands faster than time to access data in memory. Inspired by this, I searched online and find that the speed to load data in memory is much faster than in disk. Also, I find that memory can be divided into several more types, each type with different access speed. So, I produce following hypothesis: this wired phenomenon is because the loading speed difference between memory and disk.

copy use buffer to import data. The first-time buffer opens the file, it will load all data from file (in disk) into buffer itself (in memory). Of course, buffer can only contain limited content. But our test csv file is too small, so all information in the file can be put into buffer together. That is why the first-time speed looks normal: other functions and first-time *copy* all need to access disk to get data, then import data into database. After the first-time, *copy* only needs to read data from buffer (in memory) rather than in file (in disk). So, from the second-time on, *copy* become much more faster than the first-time.

Iterator is another type of memory, different from type of memory in buffer. Iterator also need to read data from csv file (in disk) for the first-time. So, the speed of first-time is normal compared with other functions. After first-time, iterator only need to access data from its memory, which is much faster than read from csv file (in disk), so the speed after first-time is much faster than speed of first-time. And type of iterator memory is faster to access than the type of buffer in *copy*. So after first-time, iterator speed is much faster than *copy* speed.

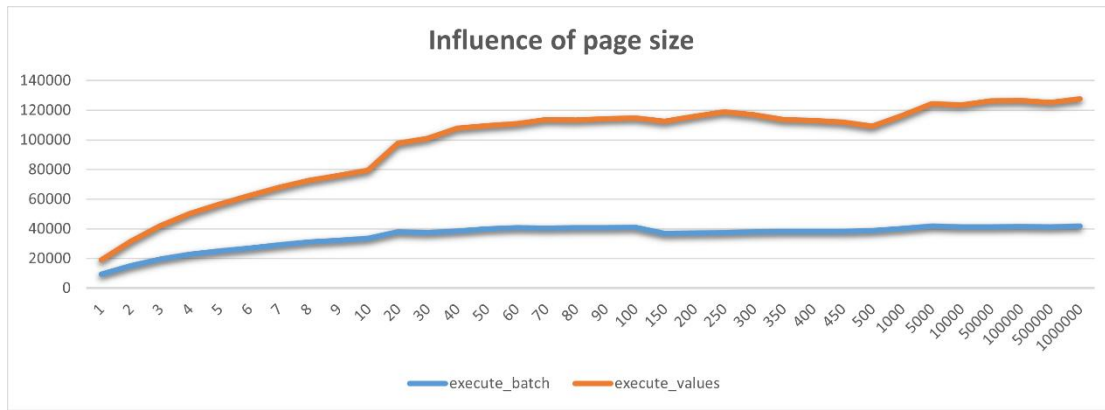
This hypothesis also explain why the amplification amount of iterator is more than amount of *copy*. That is because, the difference between disk reading speed and *copy*-type memory reading speed, is less than difference between disk reading speed and iterator-type memory reading speed.

In function *execute_batch* and *execute_values*, there is a parameter called "page size". These two functions will join the statements into multi-statement commands, each one containing at most "page size" statements. So, when the data is infinite, the bigger the page size is, the faster the speed will be.

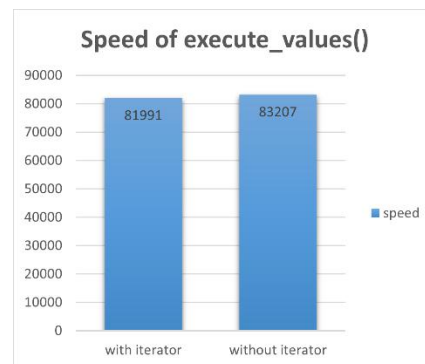
But there do not exist an infinite data file. The reality trend can be divided into five parts:

- ① Given data with limited size, the speed will first go up because the increase of page size.
- ② There will be a maximum speed in some page size.
- ③ After this optimal page size, the speed will go down for a while because the decrease of server roundtrip is less than the time to execute so many statements in one trip.
- ④ As page size approaching the data amount, the speed will gradually become a constant.
- ⑤ When page size exceed the number of statement needed to import data, the speed will totally become a constant. Just fluctuate because of some random factors.

The following figure shows the average speed of different page size for these two functions, which complies the reasoning. The average value is calculated by 100 times continual import tests.



We have mentioned that python has a special object called iterator. Both `execute_batch()` and `execute_values()` can use iterator to import data into database. To find out whether iterator can influence speed of import data into database, we do another comparison. The iterator is first-time used to keep the influence of difference between memory and disk out of this comparison. The page size of two function is both 100. The value shown in figure is the average value of 100 tests.

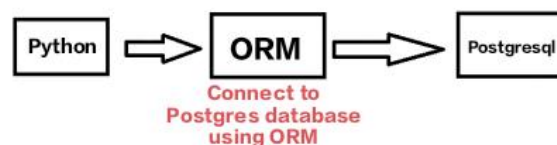


The results show that, the iterator will decrease the speed of import data. This situation can also be explained by the hypothesis about `copy` and iterator above.

Iterator needs to do two things while reading data: read data from csv file (in disk), and store one copy of data into itself (in memory). Without iterator, program only need to read from csv, do not need to store it again. So iterator will decrease the import speed of the first-time.

Using SQLAlchemy

If we use ORM (Object-Relational Mapping) technology to map relational database table structures to objects to achieve object-oriented programming, which will make it unnecessary to deal with complex SQL statements. Then only simply manipulation on the properties and methods of entity objects are required. A great ORM framework to use in Python is SQLAlchemy.



To operate on database by using SQLAlchemy, there are several steps:

① All SQLAlchemy applications start with an Engine object.

1. `db = sqlalchemy.create_engine('postgresql+psycopg2://checker:123456@localhost:5432/cs307_2')`

② We need to create a database table as a Model in the format it recognizes and it can be used to declare models.

1. `base = declarative_base(db)`

```

2. class Header(base):
3.     __tablename__ = 'header'
4.     contract_number = sqlalchemy.Column(sqlalchemy.String(150), primary_key=True) # varchar()
5.     contract_date = sqlalchemy.Column(sqlalchemy.String(150))

```

③ With *create_all()*, we can create the initial database and then create an object *session* similar to a cursor with bind a database instance.

```

1. base.metadata.create_all(db)
2. s = sessionmaker(bind=db)
3. session = s()

```

④ Import csv file with pandas and convert the *DataFrame* to *dictionary*.

```

1. df = pd.read_csv("headerWithTitle.csv")
2. lines = list()
3. read_result = df.reset_index().T.to_dict()

```

⑤ For each record in dictionary initialize it as an instance, then use *session* to add it.

```

1. header = Header(contract_number=read_result[_index]['contract number'],
2.                  contract_date=read_result[_index]['contract date'])
3. session.add(header)

```

⑥ Commit all operations.

```

1. session.commit()

```

The running result is as followed (we have repeated importing table operation 1000 times to get an accurate count for time):

```

5000 records successfully loaded
SQLAlchemy() : 46739 records/s

```

Task 4: Compare DBMS with File I/O

Test data description

Some notes about time: we do not know what should be included in “operation time” for each type of operation. At first, we only count time of execute sql statement for “operation time”, and we try to remove all other factors like the judgement in for(;;) iteration. But soon we discover it hard to give a formal definition of “operation time”, and the measurement is also difficult to implement in I/O area. So, we turn to SA for some help.

After some discussion, we decide to start timer at the beginning of the whole function (including initialize constants, prepare sql statement, etc.) and stop timer at the last code sentence of function. Besides, we use 1000 as cardinality of speed in both I/o and database speed calculation. That is, $speed = \frac{1000}{stop-start} operations/sed$.

Consider the amount of data, we decided to choose the contract_info.csv as the test data, which has 50000records. The DDL of the table is as below:

```

1. create table contract_info
2. (
3.     contract_number    varchar,
4.     client_enterprise  varchar,
5.     supply_center      varchar,

```

```

6.      country          varchar,
7.      city             varchar,
8.
9.      industry          varchar,
10.     product_code      varchar,
11.     product_name      varchar,
12.     product_model     varchar,
13.     unit_price        int,
14.
15.     quantity          int,
16.     contract_date      date,
17.     estimated_delivery_date date,
18.     lodgement_date     date,
19.     director          varchar,
20.
21.     salesman          varchar,
22.     salesman_number    varchar,
23.     gender            varchar,
24.     age               int,
25.     mobile_phone      varchar
26. );

```

The tests are divided into four parts: **Select, Insert, Update, Delete**. In each corresponding test, we respectively implement these operations and all single operation will be repeated 1000 times to calculate its average value:

1. Query the data whose *contract_number* is less than "CSE0001000".
2. Insert 1000 data with *contract_number* from "CSE0005000" to "CSE0005999".
3. Update the *product_model* of the data inserted in test 3 from "PosterKIK" to "PosterFRANXX".
4. Delete the data inserted in test 3.

Python

DBMS

Connecting to database using psycopg2 and SQLAlchemy

First of all, we try to implement *insert, select, update, delete* operations using psycopg2, which we have mentioned in task3.

In *operateWithPsycpg2.py*, I created a new connection object *con* and then tested with *conn.cursor()*.

```

1.     conn = psycopg2.connect(database="cs307_2", user="checker", password="123456", host="localhost", port="5432")
2.     cur = conn.cursor()

```

During the operations, we also compared the efficiency of the four kinds of operations before and after optimization.

execute()

```

1.     cur.execute("SELECT * from contract_info where contract_number = '" + contract_number + "'")

```

execute_batch() with iterator

1. `extras.execute_batch(cur,`
2. `"SELECT * from contract_info where contract_number = %s",`
3. `iter_list, page_size=page_size)`

The operation results are in the last part of this task, we can see that there is obvious difference with execute statement and execute_batch with iterator, which also verified the conclusion in task3.

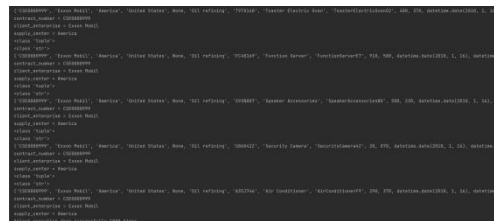
We also try to implement insert, select, update, delete operations using SQLAlchemy, which we have mentioned in task3.

PS: All related operation results are in the last part of this task

Results verified

Select

There are last few search results, after comparing with the file, we can confirm the accuracy of the operation.



CSE0000999	Exxon MobiAmerica	United St	NULL	Oil refini	T9T8140	Toaster E	ToasterEle	400	370	2010/1/16	2010/3/13	2010/1/24
CSE0000999	Exxon MobiAmerica	United St	NULL	Oil refini	FC40169	Function f	FunctionSe	910	580	2010/1/16	2010/2/12	2010/2/26
CSE0000999	Exxon MobiAmerica	United St	NULL	Oil refini	S93N807	Speaker A	SpeakerAcc	580	230	2010/1/16	2010/2/7	2010/1/23
CSE0000999	Exxon MobiAmerica	United St	NULL	Oil refini	S86042Z	Security (SecurityCe	20	870	2010/1/16	2010/3/6	2010/1/16
CSE0000999	Exxon MobiAmerica	United St	NULL	Oil refini	A35J746	Air Condit	AirConditi	290	370	2010/1/16	2010/1/30	2010/1/31

Insert

With *select* statement, the results of the insertion is shown below:

A screenshot of a database table with columns: contract_number, contract_name, contract_status, item, country, city, industry, product_code, product_name, and product_model. The table contains multiple rows of data, including contract numbers like CSE0000999 and CSE0000998, and product names like 'Toaster E' and 'Function f'.

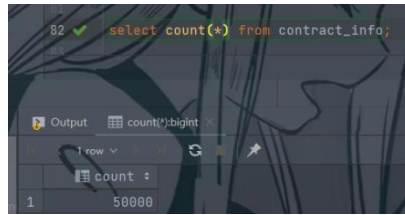
Update

As before, the results of *select* show that the product_model had been correctly updated to "PosterFRANXX".

A screenshot of a database table with columns: contract_number, contract_name, contract_status, item, country, city, industry, product_code, product_name, and product_model. The table contains multiple rows of data, including contract numbers like CSE0000999 and CSE0000998, and product names like 'Toaster E' and 'Function f'. The product_model column is highlighted, showing the updated values.

Delete

After all four kinds of operations executed once, the number of columns in the table should be the same as the beginning, i.e. 50,000 rows .



File IO

Using list functions

In this part, we read the *contract_info.csv* and convert it to the type of *list*, and then all four kinds of operations are based on the list function. However, the results is not so satisfying except inserting.

Using pandas

Insert

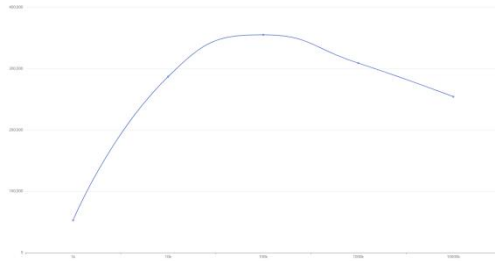
While testing for insert, we can find that the speed is extremely slow(35.16 operations/s), after looking up in the source code of *pandas*, it tell that:

“Iteratively appending rows to a *DataFrame* can be more computationally intensive than a single concatenate. A better solution is to append those rows to a list and then concatenate the list with the original *DataFrame* all at once.”

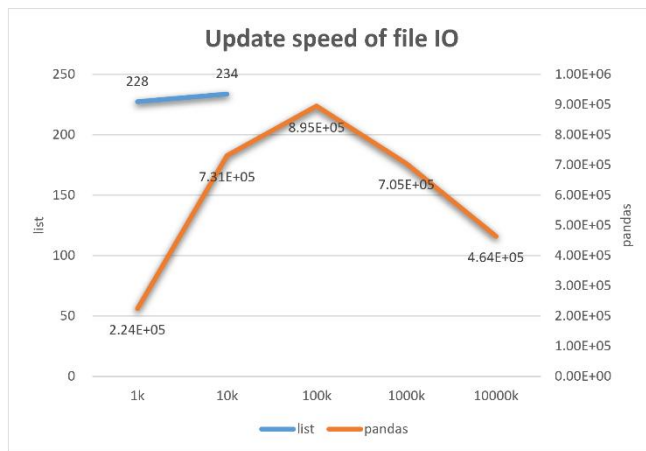
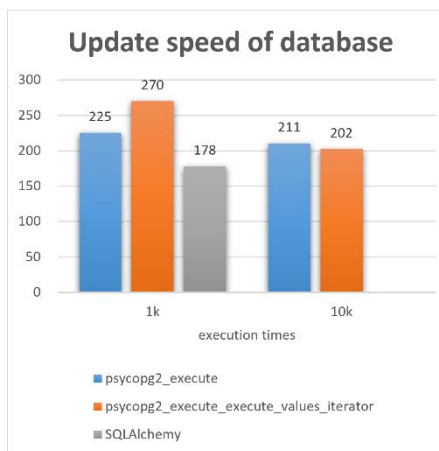
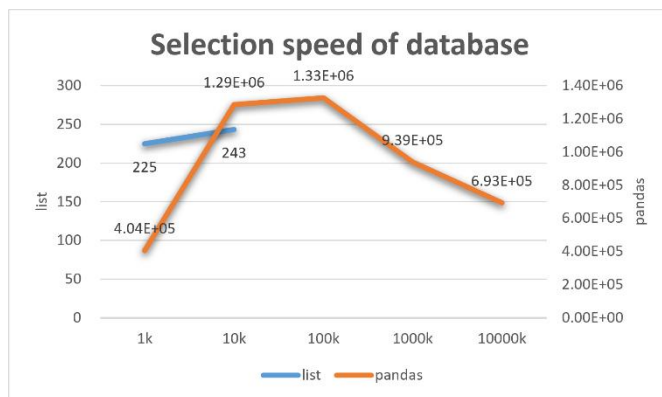
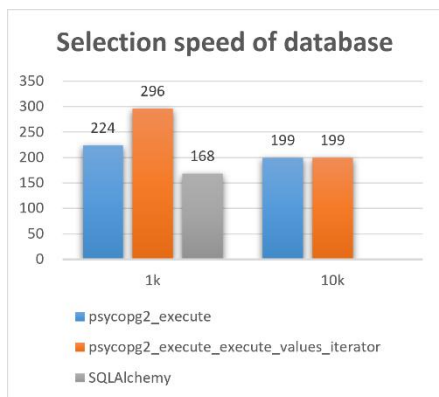
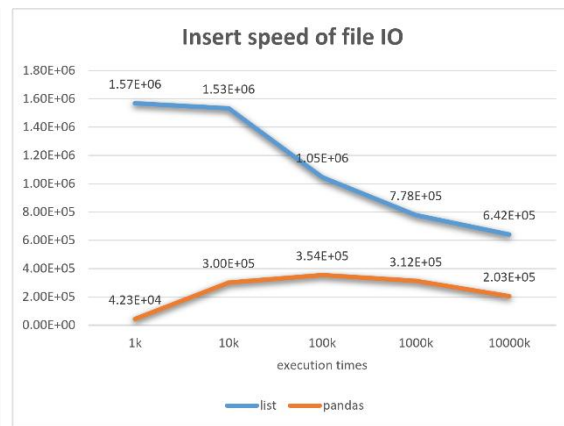
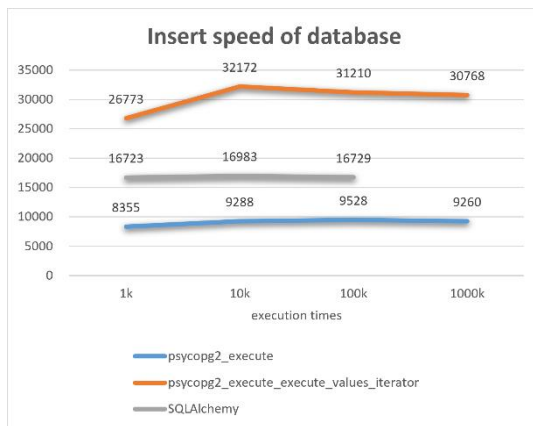
Then, according to the hints, I append the inserted data to a list, and then convert it to *dataframe* and finally concatenate the *dataframe* with original data.

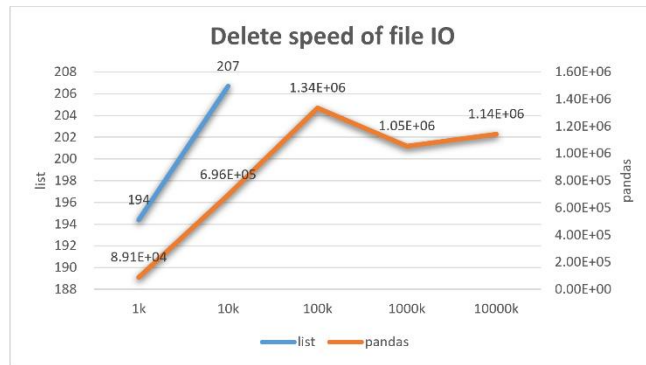
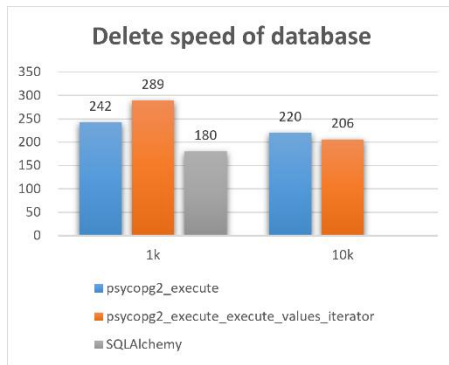
```
1. list = []
2. for i in range(0, total_cnt):
3.     contract_number = "CSE" + str(i + 5000).zfill(7)
4.     list.append([contract_number, 'Studio Trigger', 'Asia',
5.                 'Japan', 'NULL', 'Internet', 'BJS0822', 'Poster', 'PosterKIK', 50, 1000, '2013-10-3',
6.                 '2013-11-10', '2013-11-11', 'Steven Edwards', 'Theo White', '12140327', 'Male', 25,
7.                 '13986643179'])
8. newdata = pd.DataFrame(list, columns=column_name)
9. data = data.append(newdata, ignore_index=True)
```

The speed up is tremendous(54060.19 operations/s). What's more, the speed advantage of manipulating files increases as the number of operations increases, that is, when operations increases from 1k to 1000k, the speed is accelerated by 5.4 times.



Results of comparing DBMS with File I/O in Python





Extra explanation:

In the line graphs above, we can see that the operation speed of pandas is escalating while the counts of operation increase and reach a peak approximately at operation count = 100k. After then, it shows a declining trend. A guess to explain this phenomenon is that as the number of operations increases, the amount of commands executed by the ALU also increases abruptly leading to the CPU overheating and eventually results in the downclock. Another conjecture is the decrease of memory during the operations.

Java

Comparison

In database API, we imitate Lab6 to construct the main structure of the program. And we also use methods in video to elevate running speed: execute by batches, using prepareStatement to decrease compile time, open and close connection only one time for 1000 operations in the same type (To make comparison fair, below I/O also open and close file connection for only one time in the same type). The main steps are below:

1. Start timer. Open connections to database
2. Prepare statements will be used
3. Execute one sql statement, insert some values into prepared statement if necessary.
Repeat this step for 1000 times.
4. Close the connection with database
5. Stop timer, calculate running speed of this type of operation.

In I/O part, we use BufferedReader and BufferedWriter to read and write information from and into target file. All four operations INSERT, DELETE, UPDATE, SELECT have the same structure. The main steps are below:

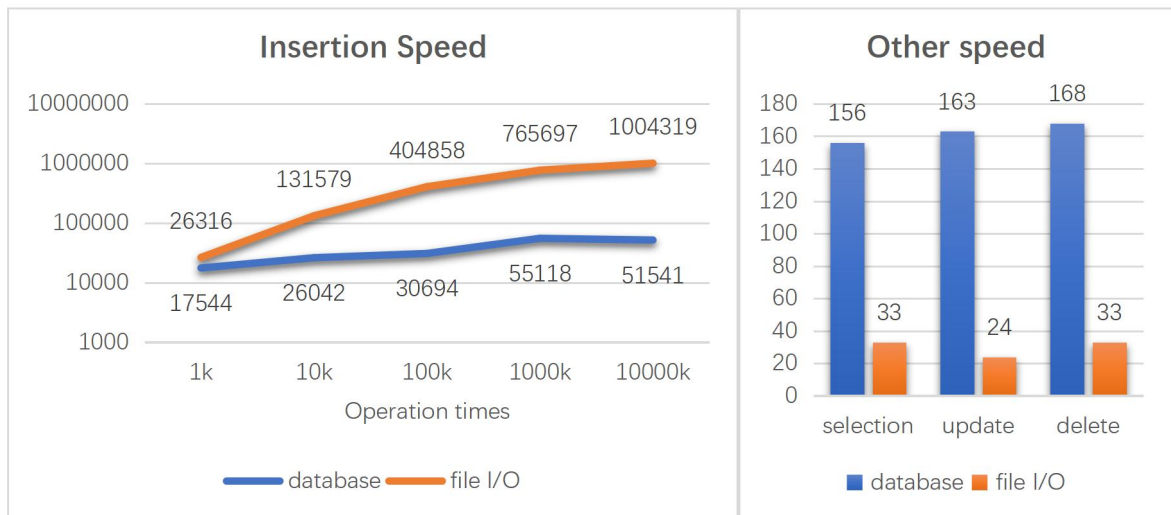
1. Start timer, initialize the reader and writer.
2. Read all data into a list. The datatype of list may be different in different options.
In selection, list is only used for getting data rather than edition, so we use ArrayList, which has $O(1)$ time complexity to get element.
In delete and update, list is used for edition, so we use LinkedList which has $O(1)$ time complexity to change element.
3. Travel over the list and do one operation. Repeat this step for 1000 times.
4. Close reader and writer.
5. Stop timer, calculate running speed of this type of operation.

In process, we do not write optimal algorithm at first. We change the structure of I/O code for three times:

version 1	Code	Open and close reader for only one time in each type of operation. In one travel of data, perform all requests for 1000 time.
	Shortage	This is equal to “select xx from xx where xx is between xx and xx” for one time, not “select xx from xx where xx = xx” for 1000 times. Do not satisfy our request at all.
version 2	Code	Only perform one request in one travel of data. Open and close reader and writer in every operation of the same type.
	Shortage	Unfair in comparison, because database only open and close connection for one time in each type of operation.
Version 3	Code	Open and close reader and writer for only one time.

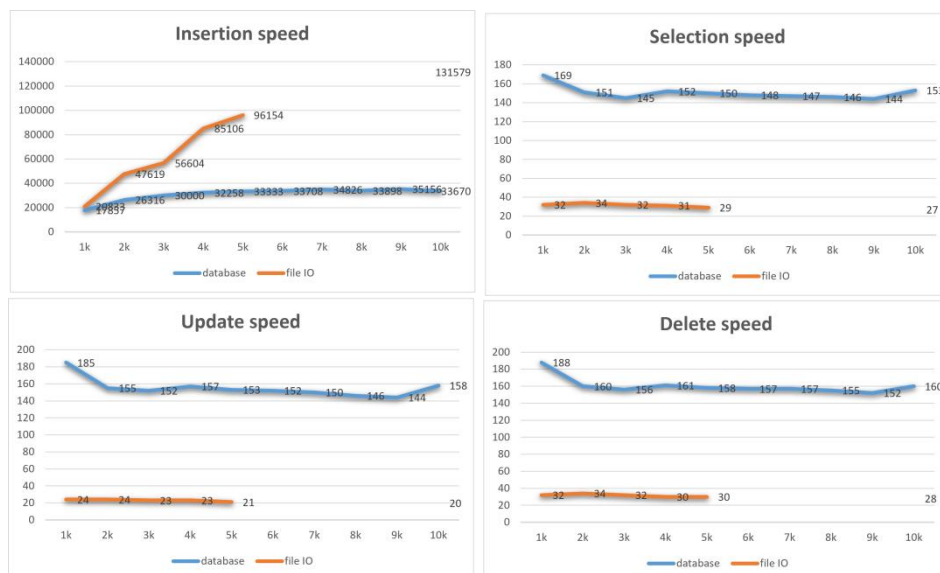
Among all four types, insertion is the fastest, so we do multiple test on insertion. Because other operation is much slower than insertion, so we only perform one test for them.

In test, the original size of database is 50000 lines. Insertion is operated for 1k, 10k, 100k, 1000k, 10000k times respectively. Selection, update and delete are operated for 1k times respectively.



At first glance, it seems wired that database system is slower than file system. But all results are reasonable. The main reason is database originally doesn't guarantee to be faster than file system. Its advantages are in other aspects, which will be described in next part.

To have a clear view of whole variation, I run the test for more times, and draw the figures below:



In insertion figure, the file IO speed keep increasing and database speed also keep increasing in general but in a relatively slower rate, which is the same as the property in speed test above (from 1k to 10000k).

In other operations, the file IO speed is flat, only slight fluctuation, which can be seen as allowable error caused by uncontrolled random factors. The database speed first decrease and then increase. This happens may because the speed is constrained by both threshold of data system and generating speed of sql statements. When operation times are low, since sql statements is familiar, Java can use information in cache to construct sql statements, so the start point is high. Then as operation times increase, the cache is unable to maintain all information, so Java need to construct sql in other way which is slower than from cache, so the speed goes down after start point.

To our surprise, the speeds of selection, update and delete go up at 10k scales. Although we really want to figure out the reason, the running time is too large for us to get the result, so we can not get the reason why it increase at larger scales.

Difference between database and file IO

We summarize the advantages of database system over file system as follow table. The references are these three websites:

[Advantage of database management system over file system](#)

[Advantages of Database Management System](#)

[Characteristics and benefits of a database](#)

Advantages	Description
Content	self-describing: contains both database itself and metadata which defines and describes relationships between tables
Consistency	one change is all needed to change the structure of a file
View	can have subtables, different user could see different views
Concurrency	concurrency control strategies: 1. data sharing: many users can access the same database at same time 2. and all data accessed are correct, data integrity is maintained 3. integrity: data remains consistent and valid even if several users update the same information
Redundancy	only have one database, any change in it will reflected immediately, so no chance of duplicate data
Added tuples	only tuple following rules can be added
Authority	1. easy to control: different users can have different authority 2. privacy: only authorized users can access a database
Independence	data independence: system data descriptions are separated from application programs because data structure is handled by DBMS rather than application program itself
Not know location	users do not need to know where the data is stored
Management tools	backup and recovery: can recover from backup when data is corrupted data replication: faster availability load balancing: reduce the load on a particular database by sharing a load among all replicated databases monitoring and security: prevent unwanted data access
Query set	have a huge query set to manipulation contents
Index	each entry has unique index, which makes search quicker

Transaction ACID	transaction: operations to be carried out in a single logical work unit Atomicity: ensures all or none of operations will be execute in a single logical unit of work Consistency: state of database system should be the same before and after transaction Isolation: multiple transaction executing, one shouldn't interfere others Durability: all changes need to be saved after transaction
Remote access	database can be accessed by online websites or apps remotely
Client supportive	can be accessed whatever programming language is used
Scalability	availability from small scale to large scale

All advantage above put emphasis on management rather than speed of execution.

We think intuitively that database should be faster than file IO is because we have been exposed to file IO in grade one and learn database recently, and database is invented after file data system. We think the later an event is invented, the more advanced it will be. But this is not guaranteed. Database is invented later than file data system, so its functions and main structure is likely to be better than file IO, and have solved some problems which file system still has, like concurrency. But these improvements do not include running speed.

File IO and database query both have unique characteristics. File IO is based on the file in computer, so it can use memory and cache to store data temporarily. Database is based on a separated system, it needs to use connection to query and can not use cache on computer. And reading cache in computer may be faster than the total time of “send query – do query – get result” process. In addition, the information we insert into file is almost the same, only differ from “CSEXXXXXXX”, the rest of which is all the same. Since this string is not huge, it can be stored in cache totally. So file IO read from cache for most of time, while database need to use connection to do query. This is the reason why file IO insertion is faster than database, and become faster and faster when execution time increase, even keep becoming faster when database reached threshold.

In selection, update and delete, file IO can not use cache, so it become much slower than database.

Test in distinct operate system

In addition, we also test insert with psycopg2 by function: execute_execute_values_iterator in mac system. However, there is no so much difference. As for the java program, we gave up testing it due to its cross-platform nature.

