CS307 Project Part I

Basic Information

Member	Student ID	Contribution Rate
袁龙	12211308	33.33%
于斯瑶	12211655	33.33%
赵钊	12110120	33.33%

Contribution of work

袁龙: E-R Diagram, Table Creation(including Report), `postgresql` & `MySQL` Comparison

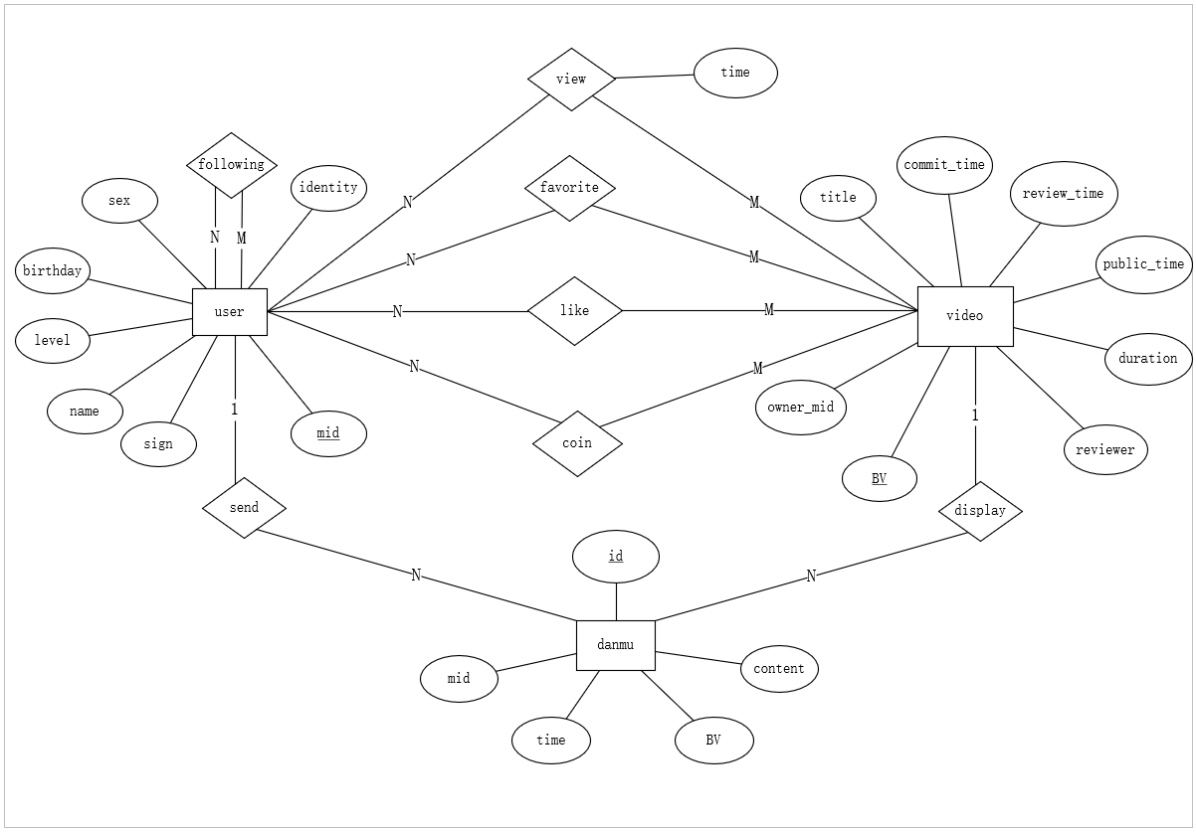
于斯瑶: Data Import and Optimize(including Report), `JDBC` & `PDBC` Comparison

赵钊: Basic Comparison(including Report), Index & `B-Tree` Comparison, Concurrency Test

Together: Database Design

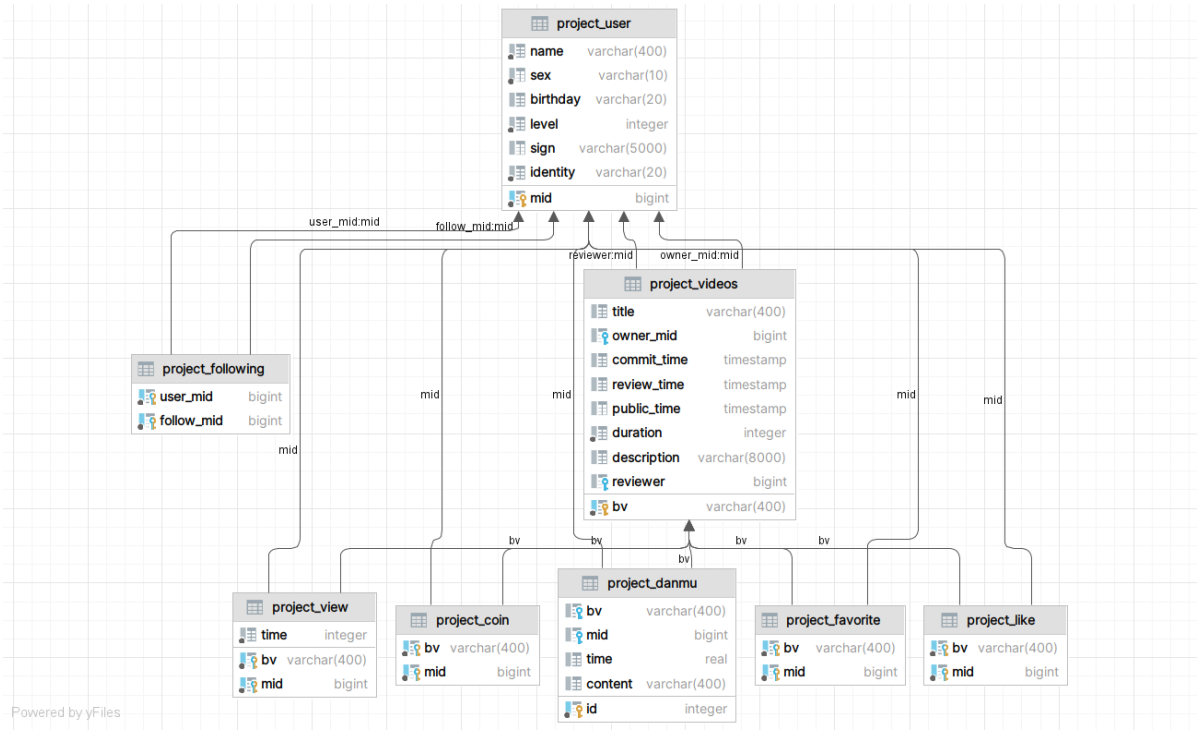
Task 1: E-R Diagram

We use Visio to draw the E-R diagram.



Task 2: Database Design

1. E-R Diagram By DataGrip



2. Table Design Description

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

- Column `following` in `user.csv` usually has multiple mids of up.
- Column `like`, `coin`, `favorite` and `view` in `videos.csv` usually has multiple `mid`.
- In the same video, the same person may send more than one `danmu` at the same time, so we need a self increasing primary key `id`.

These information results in the files do not satisfy the three normal forms.
In order to make the database design meet the three normal forms and be as easy to expand as possible,
we designed the following tables.

project_user

- `mid`: the id of the user which is the primary key of this table
- `name`: the name of the user
- `sex`: the sex of the user
- `birthday`: the birthday of the user
- `level`: the level of the user
- `sign`: the personal description
- `identity`: could only be "user" or "superuser"

project_following

- `user_mid`: the `mid` of the user who followed others
- `follow_mid`: the `mid` of the user who is being followed

project_video

- `BV`: the unique identification string of a video
- `title`: the name of the video
- `owner_mid`: the mid of the video owner
- `commit_time`: the time when the owner committed this video
- `review_time`: the time when the video was inspected by its reviewer
- `public_time`: the time when the video was made public for all users
- `duration`: the video duration
- `description`: the brief text introduction given by the uploader
- `reviewr`: the `mid` of the video reviewer

project_like, project_coin, project_favorite

- `BV`: the video `BV` of the video liked / given the coins / favored
- `mid`: the `mid` of the user who liked / gave the coins / favored the video

project_view

- `BV`: the video `BV` of the video watched
- `mid`: the `mid` of the user who watched the video
- `time`: last watch time duration of the user

project_danmu

- `id`: the tag of `danmu` which is a self increasing primary key
- `BV`: the video `BV` of the video that the `danmu` was sent
- `mid`: the `mid` of the user who sent the `danmu`
- `time`: the time of the video that `danmu` appears
- `content`: the content of the `danmu`

Our design satisfies the needs of following :

- Follow the requirements of the three normal forms.
    - Each property cannot contain multiple values or duplicate values.
    - Every non-key attribute (column) is fully functionally dependent on the entire primary key.
    - Each non-primary key property does not depend on other non-primary key properties.
- Each table has its primary key.
- Every table is included in a link. No isolated tables included.
- Contain no circular links.
- Each table has its own `NOTNULL` column and `UNIQUE` column.

- Use appropriate types for different fields of data.

3. Scalability

The tables we've designed exhibit strong scalability:

- Table `project_following` can store all `mid` of following users. It's also very easy to add or delete new following. Meanwhile, it can easily retrieve the list of users following this user by keyword `group by`.
- Table `project_like`, `project_coin`, `project_favorite` and `project_view` also have the scalability like `project_following` for the similar reason.
- Table `project_danmu` uses self increasing primary key, which can make insert easier.

Task 3: Data Import

1. Script Description

In this task, we have 7 script files. `UserReader.java`, `VideosReader.java`, `DanmuReader.java` are the scripts we used to import data at the beginning. `UserReaderFaster.java`, `VideosReaderFaster.java`, `DanmuReaderFaster.java` are the scripts we after our optimization and `VideosReaderMoreFaster.java` is the second edition over `VideosReaderFaster.java`.

If you want to replicate our import process, please **pay attention to** the following tips:

- Your project structure should be like this and run the scripts in `IntelliJ IDEA`, or file path may cause some exceptions.

```
CS307_23F_Project_Part1
|
├── data
|   |   users.csv
|   |   videos.csv
|   └── danmu.csv
|
└── src
    └── task3
        └── *.java
```

- The first three script may run slowly, **please patiently waiting** until the program runs over.
- You should first run `DDL.sql` in task 2 to create tables, then you can run the scripts in this task.
- You should first run `User*.java`, then `Videos*.java` and `Danmu*.java` at last, otherwise there will occur some exceptions relating to violation of foreign key constraints.
- You should modify `host`, `port`, `db_name`, `user`, `pw` to your own information in every script's `openDB()` method.

```java
public void openDB() {
    // Please modify the following strings
    String host = "localhost";
    String port = "5432";
```

```
        String db_name = "project1";
        String user = "postgres";
        String pw = "123456";

        String url = "jdbc:postgresql://" + host + ":" + port + "/" + db_name;
        try {
            Class.forName("org.postgresql.Driver");
            conn = DriverManager.getConnection(url, user, pw);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
```

Follow the above tips and run the `main(String[] args)` method in each script, the data will be imported to your database.

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

1. Open database connection
2. Use file I/O to read the data from `*.csv` files
3. Do **extraordinary complex String operations (we did NOT use any external library and only purely manual cutting and extracting string information which is a huge work, the relating script files are in the folder `task3`)** to get the information of every data
4. Use `Java` to execute `sql` sentence to import data into tables
5. Close database connection

The number of data entries of every table is as follows.

Table	Number of data entries
`project_user`	37881
`project_following`	5958770
`project_videos`	7865
`project_like`	86757948
`project_coin`	80571520
`project_favorite`	79181895
`project_view`	163997974
`project_danmu`	12478996

2. Script Optimization

There are two main optimization directions for us :

1. Pre-compile SQL statements once and execute multiple times
2. Insert data in batches

For the first direction, we put `stmt = conn.prepareStatement(sql);` outside while loop.

For the second direction, we close auto commit using `conn.setAutoCommit(false);` and we add a constant `BATCH_SIZE` which means the number of data we execute and commit one time. Here is an example, the code at the top is the one before optimize and the one at the bottom is after optimize.

For each table, we test the time cost of loading all the data **for 5 times** and calculate the average. We can see the result as follows. Additionally, the test environment information is in Task 4.

Table	Time Cost before Optimize	Time Cost after Optimize	Optimization Rate
`project_user`	6810 ms ≈ 6.8 s	1827 ms ≈ 1.8 s	377%
`project_following`	685435 ms ≈ 11.4 min	103531 ms ≈ 1.73 min	659%
`project_videos`	130712 ms ≈ 2.16 min	47263 ms ≈ 47 s	276%
`project_like` & `project_coin` & `project_favorite`	Too long to wait	Each about 2.2 to 2.5 h	
`project_view`	Too long to wait	About 8.3 h	
`project_danmu`	1549102 ms ≈ 25.8 min	270607 ms ≈ 4.5 min	573%

We find that our script still cost much time to import data into `project_like`, `project_coin`, `project_favorite` and `project_view`. So we continue optimize out script `VideosReaderFaster.java`. We have two main directions :

1. Use external library `opencsv-5.8.jar` instead of out **extremely complex String operation**
2. Use multi-thread to insert a large amount of data at the same time

The `max_connections` of `postgresql` is 100 and the total number of videos is 7865, so we enable 92 threads that each one deal with the data of at most 85 videos. The result we tested is as follows :

Table	Time Cost after the 1st Optimize	Time Cost after the 2nd Optimize	Optimization Rate
`project_like`	About 2.5 h	37 min	405%
`project_coin`	About 2.3 h	33 min	418%
`project_favorite`	About 2.2 h	32 min	413%
`project_view`	About 8.3 h	105 min	474%

We can see that the new optimization is very effective in improving the efficiency of these four tables.

Analysis

- If we put `stmt = conn.prepareStatement(sql);` outside while loop. The pre-compile sentence will be **only run once and use many times**, so the efficiency will highly increased.
- The second optimization utilizes the batch processing mechanism of the database. If each SQL statement needs to be implemented once using `JDBC`, the code will be lengthy and too many database operations will be executed. **If several SQL statements are combined into a batch and transferred to the database for execution at once, it can reduce the number of transfers and improve efficiency**.
- Usually, do some operations by external libraries may cost less time than do them manually.
- **Effectively utilizing the concurrency of databases can greatly improve efficiency.** About the concurrency test, please find detailed information in Task 4.

Task 4: Compare DBMS with File I/O

1. Test Environment

Hardware specification
CPU: 11th Gen Intel(R) Core(TM) i5-11300H @ 3.10GHz
RAM: 16.0 GB
Software specification
DBMS: `postgresql-12.5`, `MySQL-8.0.35.0`
OS: Windows 10 Home Chinese Version
Programing language: `Java 13.0.2`, `python 3.10.9`
IDE: `IntelliJ IDEA 2018.3.3` for Java, `Pycharm Community Edition 2021.3.1` for python
Additional libraries: `postgresql-42.6.0.jar`, `opencsv-5.8.jar`, `commons-lang3-3.13.0.jar`
The result of tests may be different in different test environment.

In order to replicate the experiment, you may need to construct the project structural components as follows.

```
CS307_23F_Project_Part1
|
├── data
|   |   users.csv
|   |   videos.csv
|   └── danmu.csv
|
├── properties
|   └── jdbc.properties
|
└── src
    └── task4
        └── *.java
        └── *.py
```

2. Test Description

In this task, we mainly use the data in file `user.csv`, so we mainly focus on table `project_user` and `project_follow`. In the basic part, we will test some DQL and some DML including insert, update, delete operations then compare them with `java.io`. In the advanced task, we will compare DBMS with indexes with Java implemented `B-tree`; using multi-user and multi-threading to test concurrency; comparing `postgresql` with `MySQL`; comparing `JDBC` with `Python Database Connection`.

3. Script Description

There are mainly 4 java file we used in finishing the basic part.

DataHandler.java

This is an interface which contains some methods that will be implemented in class `FileHandler` and `DatabaseHandler`.The methods are listed below. There are very detailed annotation in the source code that explained every method.

```
public interface DataHandler {
    public List<String> queryAllUsersName();
    public List<String> queryNameByMIDRange(long min, long max);
    public List<String> queryNameByLevel(int level);
    public List<String> queryNameByMID2Digits(char c1, char c2);
    public List<String> queryLongestName();
    public int queryDistinctBirthdayCnt();
    public List<String> queryMostFollowersUserName();
    public void insertUser(long mid, String name, String sex, String birth,
                           int lv, String sign, List<Long> fol, String idt);
    public void insertFollowersByMID(long up_mid, long fans_mid);
    public void updateSexByMID(long mid, String sex);
    public void deleteUserByMID(long mid);
}
```

FileHandler.java

This is a script only use `java.io` and `java.util` to implements the method in the interface `DataHandler`. Instead of `DBMS` operations, we use a lot of String operations such as `split()`, `substring()` and `equals()` and some collections such as `Map<>` (implemented by `HashMap<>`), `HashSet<>` (implemented by `HashSet<>`) and `List<>` (implemented by `ArrayList<>`).

DatabaseHandler.java

Despite the methods implemented from the interface `DataHandler`, there are two additional method called `openDB()` and `closeDB()` which means get and close the connection between program and database.

Client.java

This is the class that we run our script so it is called `Client`. We input our operation id (an integer between `1` and `11`) and if necessary, the script will generate random data, if not, it will run and calculate the time cost using file I/O or DBMS then print it out. **The time is not counted when `openDB()` and `closeDB()` is running. Similarly, the time of File I/O is only counted when doing operations on file.**

jdbc.properties

A property file that contains some necessary information to connect to DBMS. Information contains

- `driver`: The class name for `postgresql` driver
- `ip`: The user ip which default value is `localhost`
- `port`: The port of `postgresql`
- `db_name`: The database name we will connect
- `user`: The user's name which default value is `postgres`
- `password`: The password of the user

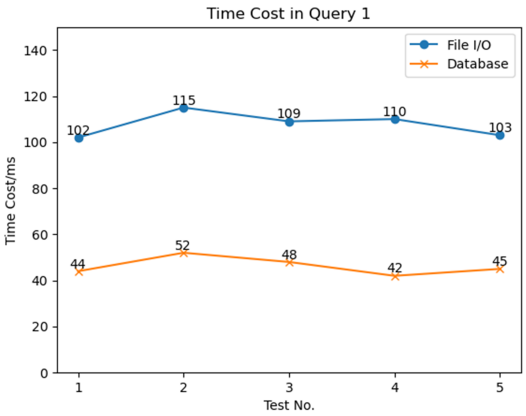
4. Basic Comparison

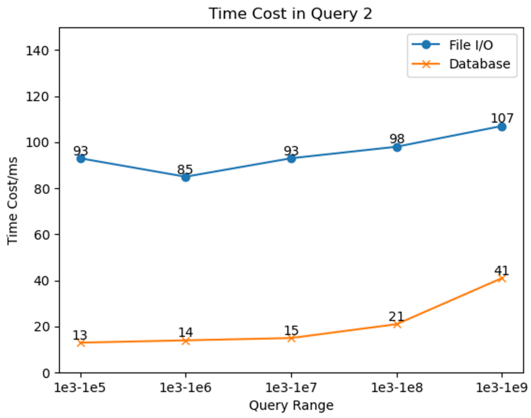
Basic Queries

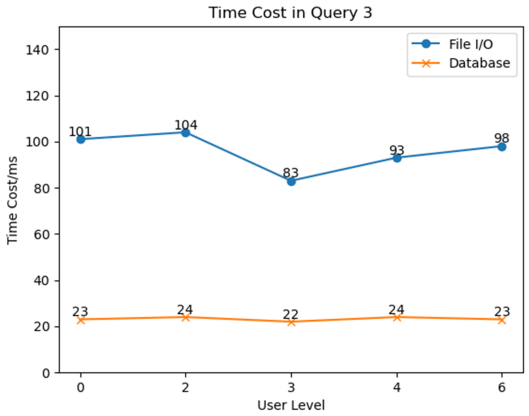
We compare 4 queries in basic query, containing query all, range query, specific query and vague query. The query sentences are below.

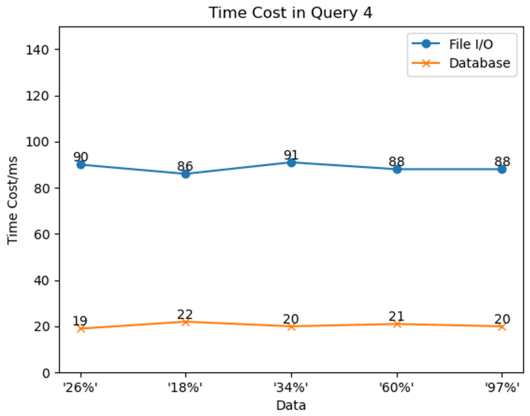
```
1. select name from project_user
2. select name from project_user where ? <= mid and mid <= ?
3. select name from project_user where level = ?
4. select name from project_user where mid = ? -- ? is like '24%', '53%' and so
on
```

We test each sentence for 5 times, record the result and visualize them with graph below. It is worth mentioning that in the second query, each time we change the range to see whether the program work good or not. For query 2 to 4, each time we change the value of `?` to wider the testing range. And the result of this 4 basic query are as follows.









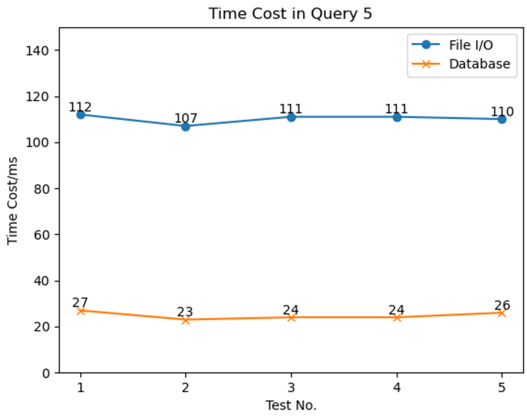
We have found that in all four queries, DBMS does better than File I/O. In addition, we observed that in range query, the larger the range is, the more time both File I/O and DBMS cost. However, the growth percentage of File I/O is larger. This can be explain from the underlying implementation. Every time, File I/O will iterate every line, which complexity is `O(N)`. But DBMS may use B-Tree to manage data, with a complexity of `O(logN)`.

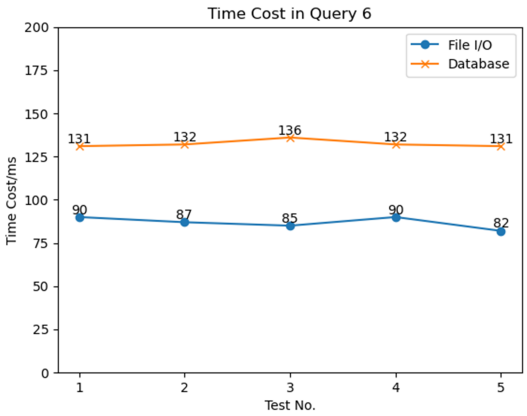
Queries Relating to Functions

We compare 2 queries in basic query, containing `max(unknown)` and `count(unknown)` in addition keyword `distinct`. The query sentences are below.

```
1. select name from project_user
   where length(name) = (select max(length(name)) from project_user)
2. select count(distinct name) from project_user
```

Also, we test each sentence for 5 times, and the result are in the below graph. **In the implement of function `max()`, DBMS is better than File I/O. But in the implement of keyword `distinct`, File I/O perform better. This may because `HashSet<>` is very efficiency since its complexity in single operation is `O(1)`.**



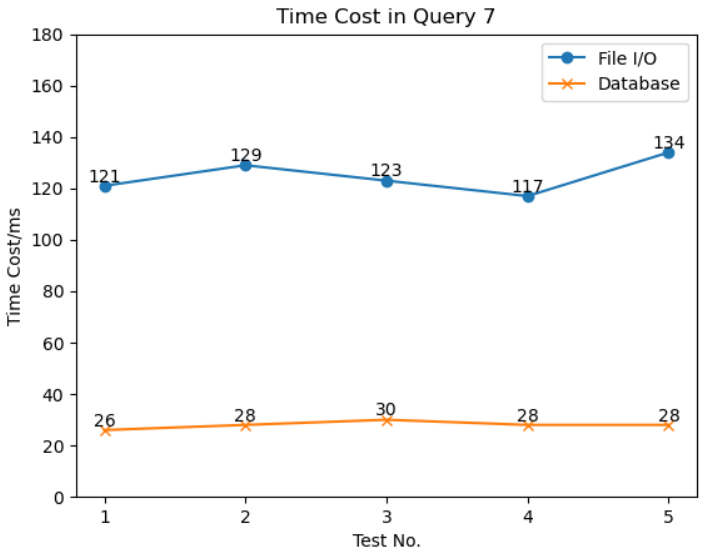


Multi-table Query

There is a query relating to table `project_user` and `project_following` that return the `name` of user who has the most followers. And here is the query sentence.

```
select name from project_user
group by mid order by count(mid) desc limit 1
```

The result is as follows. We can find that: **The efficiency of DBMS is not affected by the increase in the number of data tables.** Maybe DBMS has efficient way to join multi-table together.



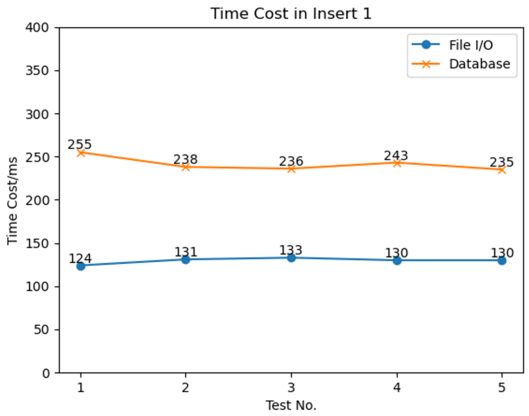
Insert & Update & Delete

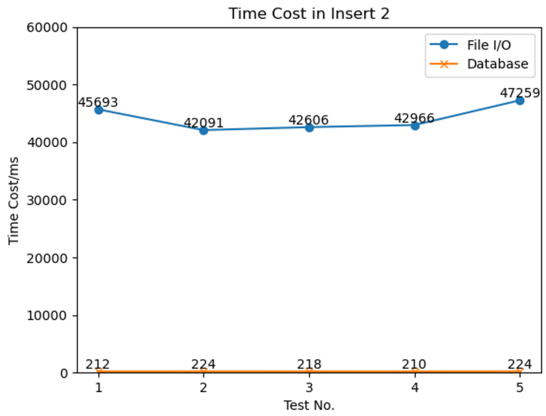
There are two insert sentence tested in this part, including insert an user and insert a follower to an user. **Only when there is no data in buffer stream, the time of File I/O stops.** We use the third `DML` sentence to test update operation. The two `?` respectively representing `sex` and `mid` this two `varchar` field. We use the fourth `DML` sentence to test delete operation. The `?` in the sentence means `mid` this field.

```
1. insert into project_user values(?, ?, ?, ?, ?, ?, ?)
2. insert into project_following values(?, ?)
3. update project_user set sex = ? where mid = ?
4. delete from project_user where mid = ?
```

All inserted data is generate randomly by methods in class `Client`. **The data size of this two test are both 1000**, that is, we will test 5 times and each time we insert 1000 random data into the corresponding table. The result of this two insert sentence test is below.

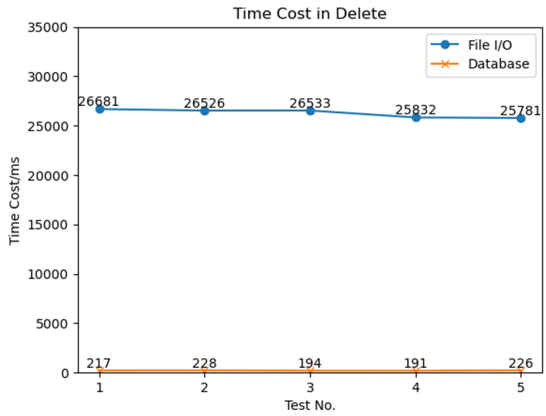
We find that in the first inert operation, File I/O does better than DBMS. This may because if we want to insert an `user`, we just need to append a single line at the end of the file and this is very fast. But if we want to add a follower to one user, we need to rewrite the whole file. And in the two insert operations, DBMS takes about the same amount of time which reflect DBMS Has good stability.

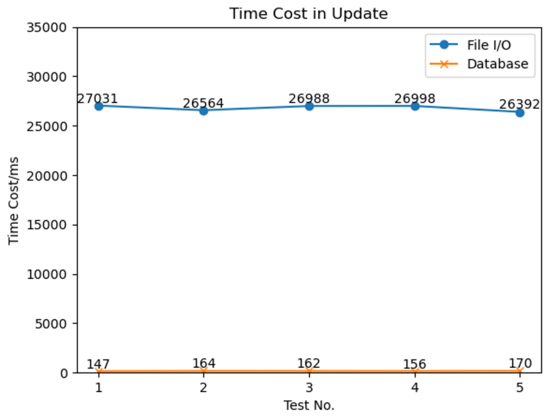




All updated data is generate randomly by methods in class `Client`. **Each time we update 500 users `sex` and we test for 5 times.** The result of this test is below. **File I/O cost much more time than DBMS. This is similarly to the second insert operation, if we want to update an user's `sex` we need to rewrite the whole file which cost more time.**

The user we delete each time is chosen randomly and the users deleted in file and database in one test are the same. **Each time we delete 500 users and we test for 5 times.** The result of this test is as follows. **The same as insert and update, the delete operation of File I/O is far more slower than DBMS.** This is because if we want to delete a data with File I/O, we need to rewrite the whole file, which cost a large amount of time.





5. Comparison between DBMS with Indexes and File I/O with B-Tree

According to the `postgresql` docs, `postgresql` has an index system implemented by B-tree, which can increase the efficiency of some operations. So we can compare DBMS indexes with `Java` implemented B-tree.

In this part, we compare the index of the primary key in `project_user` and File I/O with B-Tree. Since the the primary key of a table has native index, we do not need to add extra index to the table.

The `Java` implemented B-tree is in `BTree.java`. **The implementation is refer to Algorithms, 4th Edition by Robert Sedgewick and Kevin Wayne.** Two inner class `Node` and `Entry` are used to construct the B-Tree. There are two `public` access permission method as follows which we can call them to insert and find data.

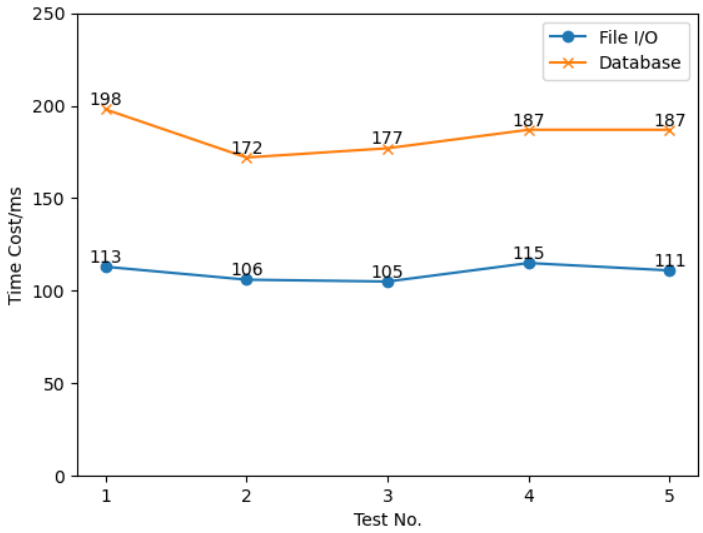
```
public class BTree<Key extends Comparable<Key>, Value>  {
    private static final class Node {
        private int m;
        private Entry[] children;
    }

    private static class Entry {
        private Comparable key;
        private Object val;
        private Node next;
    }

    public Value get(Key key);
    public void put(Key key, Value val);
}
```

In this test, we compare query between DBMS and File I/O. **We test for 5 times and each time we will randomly query 1000 users' `mid` to get their `name`.** Here are the query sentence and the result is as follows.

```
select * from project_user where mid = ?
```



We may find that in this case File I/O with B-Tree cost much less time than without B-Tree and **it is even fast than DBMS**. We conclude that although the build of B-Tree may cost some time, when the number of query operations increase, the advantage of B-Tree will be greater.

6. Comparison between Different DBMS

To comparison the different DBMS, we created 100, 1000 and 10000 pieces of insert sql statements respectively based on the data given.

In this part, we just use table `project_user` as testing object.

We inserted 100, 1000, and 10000 sets of data into `MySQL` and `postgresql` respectively (these data are all from the data given), and we compared the running time of `postgresql` and `MySQL` in inserting different data quantities, as shown in the following table:

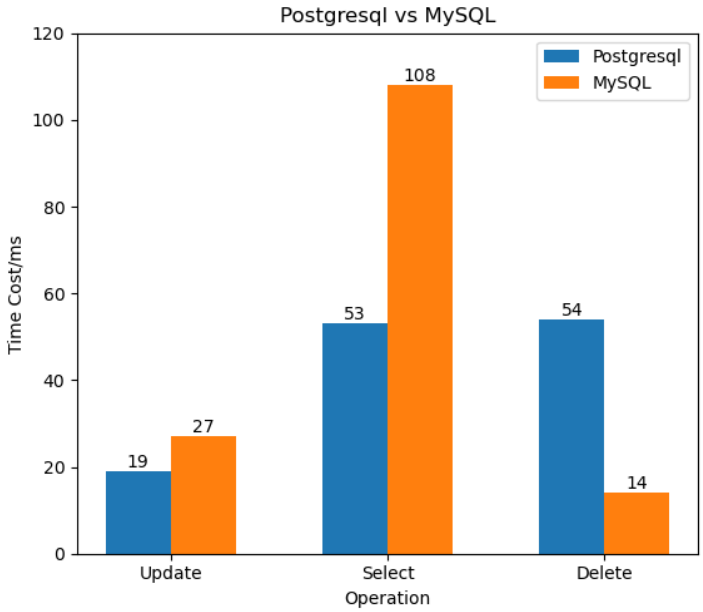
DBMS	Insert 100	Insert 1000	Insert 10000
`postgresql` (ms)	703	4270	848745
`MySQL` (ms)	1414	10476	2126432

Based on the 1000 sets of data, we carried out the update selection and delete operations respectively. The result is in the following chart.

```
update project_user
set birthday = '未知'
where birthday is null;

select mid, name, sex, birthday, sign, level, identity
from project_user
where level > 3;

delete from project_user
where level < 6;
```



By analyzing the above table and line chart, it is not difficult to see that `postgresql` **is significantly better than** `MySQL` **to operate these DQL except for the deletion operation**.

7. Comparison between Different Language Connecting Database

We will mainly compare the efficiency of JDBC and PDBC in this part including connecting, disconnecting, insert, query this four part.

In this part , we use table `project_user` as the test case.

Connecting & disconnecting comparison

We tested connecting and disconnecting 10 times, and tested each 5 times.

	1	2	3	4	5
JDBC time cost (ms)	508	561	495	493	492
PDBC time cost (ms)	267	266	282	268	266

It shows that PDBC is faster than JDBC when only connect the database and disconnect it. The possible reasons could be that `JDBC` use more mandatory exception handling and invoke more methods.

Insert comparison

We tested insert all data in `project_user` for 5 times. Both process a 1000 SQL statements in each batch.

In Java, it recognizes `\` as an escape character and escapes the `,` after `\`, making errors. Therefore, we customized it as a character with ASCII code 1, successfully inserting all data. But this problem is not exist in python.

```
CSVParserBuilder csvParserBuilder = new CSVParserBuilder();
char c = (char) 1;
csvParserBuilder.withEscapeChar(c);
CSVReaderBuilder csvReaderBuilder = new CSVReaderBuilder(new
FileReader(filePath));
csvReaderBuilder.withCSVParser(csvParserBuilder.build());
CSVReader in = csvReaderBuilder.build();
```

The code in Java is similar to task 3. The following is main part of Python code.

```
with open('..\\..\\data\\users.csv', 'r', encoding='utf-8') as f:
    reader = csv.reader(f)
    next(reader)
    rows = [(row[0], row[1], row[2], row[3], row[4], row[5], row[7]) for row in
reader]
start = time.time()
for i in range(0, len(rows), 1000):
    batch = rows[i:i + 1000]
    execute_values(cur, "insert into project_user (mid, name, sex, birthday
,level, sign, identity) VALUES %s", batch)
```

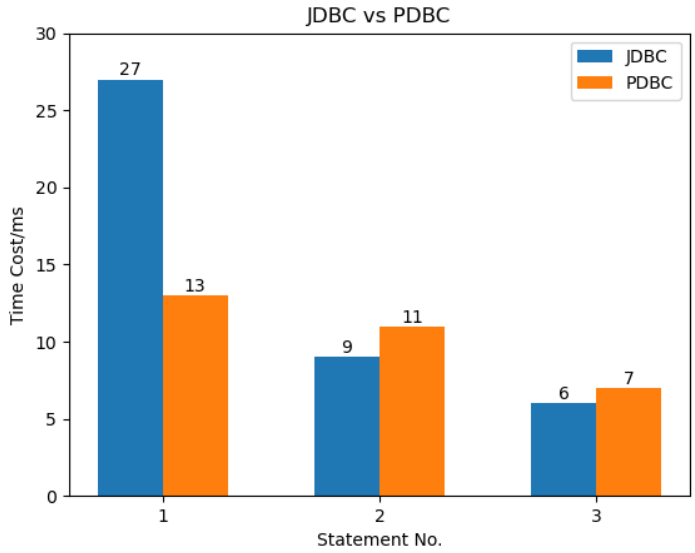
	1	2	3	4	5
JDBC time cost (ms)	1321	1268	1314	1243	1246
PDBC time cost (ms)	651	685	660	639	650

It shows that PDBC is about twice as fast as JDBC. The possible reasons could be that `pandas` and `OpenCSV` have differences in reading, Java invokes more methods, performed more forced exception handling, and different processing and submitting SQL statements.

Query comparison

In this part, we compare 3 query sql statement and each of them test for 5 times. The result is as follows.

```
select mid from project_user  -- query all
select mid from project_user where level > 3  -- range query
select mid from project_user where cast(mid as varchar) like '24%'  -- vague
query
```



We find that for the global query, PDBC is more faster than JDBC and these two perform similar in other queries.

8. Concurrency Test

To test the concurrency of our DBMS, we use multi-thread to simultaneously perform insert operations on the DBMS. There are two method added in class `DatabaseHandler` which are `concurrencyMulti()` and `concurrencySingle()` which have been pasted below, respectively representing multi-thread and single-thread insertion. We use `java.util.concurrent.ExecutorService` to implement concurrent processing. Then a new client is created called `ConcurrencyTest.java` which is used to test the script.

```
public void concurrencyMulti() {
    ExecutorService executorService = Executors.newFixedThreadPool(threadNum);
    for (int i = 0; i < threadNum; i++) {
        UserThread userThread = new UserThread(false);
        executorService.execute(userThread);
    }
    executorService.shutdown();
}

public void concurrencySingle() {
    ExecutorService executorService = Executors.newSingleThreadExecutor();
    UserThread userThread = new UserThread(true);
    executorService.execute(userThread);
    executorService.shutdown();
}
```

We set the total number of data is `100000`, and use `1`, `5`, `10`, `20`, `50` threads to test the time cost. We test each condition for `5` times and the result is the average of them.

The result is as follows. The figure on the left shows time cost of each number of threads (we called this one condition) and the x-axis means the first to the fifth time we tested. It is easy to figure out that **the more threads we use, the less time it will cost**. **So we conclude that the concurrency of the DBMS is good because more threads significantly decrease its time cost.**

The figure on the right is a line chart which shows the time cost of different condition deal with different size of data. From the graph, **all condition are extremely close to linear**, with single thread cost just a bit less time. **This means even more threads and more data, DBMS will not cost extra time so the concurrency of DBMS is good.**