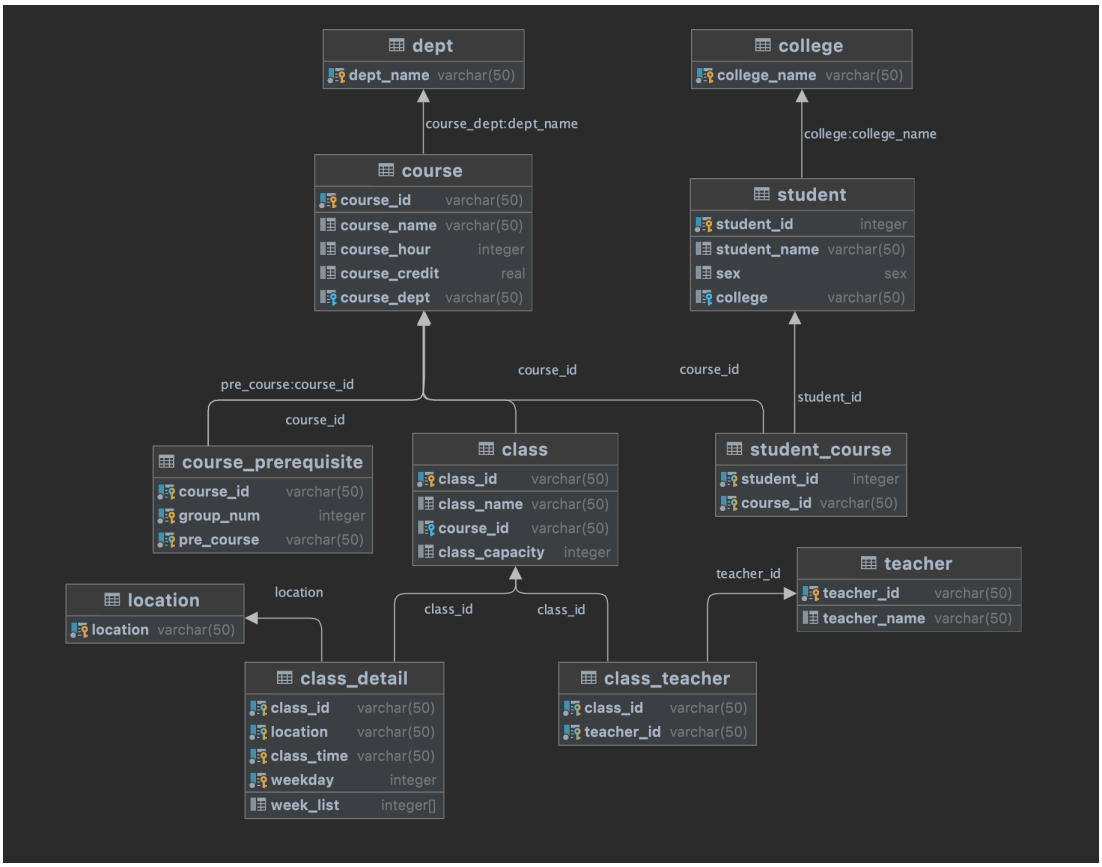
DB_project1（12011517 李子南）

Task 1: Database design



数据库的结构如图所示，总共由11个表组成。

Ideas

设计此数据库时，我将表分类为**实体表和关系表**，并根据数据库设计三大范式对数据进行了分析。若是两个实体表中的属性为一对多的关系，则可以不用单独建立关系表，而是在其中一张表中建立外键。若两个属性为多对多的关系，则需要建立关系表，例如图中的class_teacher表。并且**为了数据库的可拓展性，我对location，dept等仅有一个属性的实体建立了表格，方便以后添加属性。**

实体表： dept, course, student, class, teacher, class_detail

关系表： course_prerequisite, class_teacher, student_course

student中sex一列的数据类型为只包含"F"和"M"enum类

Task 2: Import data

清洗数据

在导入数据之前，需要先对数据进行清洗。数据清洗主要分为：**填充缺失值为空值，去除空格和制表符 ，去除部分数据的重复值。** 对course_info.json文件进行清洗后根据每个表所需的数据，分别导出成json文件。

以dept.json为例：

```
List<Dept> courses = gson.fromJson(content, type);
        System.out.println(courses.size());//清洗前行数
        for (Dept d : courses) {
            d.courseDept = d.courseDept.replaceAll("\\s*", "");
        }//去除空格和制表符
        for (int j = 0; j < courses.size();j++){
            for (int i  = j + 1; i < courses.size();i++){
                if
(courses.get(j).courseDept.equals(courses.get(i).courseDept))courses.remove(i--);
            }
        }//去重
        System.out.println(courses.size());//清洗后行数
```

对prerequisite进行清洗处理的核心代码：

```
for (Pre p : courses) {
    if (p.prerequisite == null) continue;
    int num = 1;
    boolean quote = false;
    boolean check = false;
    for (int i = 0; i < p.prerequisite.length(); i++) {
        if (p.prerequisite.charAt(i) == 44) { /* , */
            if (!quote) num++;
        } else if (p.prerequisite.charAt(i) == 91) { /* [ */
            quote = true;
        } else if (p.prerequisite.charAt(i) == 93) { /* ] */
            quote = false;
            if (!check) num--;
            check = false;
        } else if (p.prerequisite.charAt(i) == 34) { /* " */
            int tmp = i++;
            while (p.prerequisite.charAt(i) != 34) i++;
            if (course_name.contains(p.prerequisite.substring(tmp + 1, i))) {
                //存在这门课，course_name是一个存放了所有课程名的List
                if (quote) check = true;
                pre_courses.add(new Pre_course(p.courseId, num, p.prerequisite.substring(tmp +
1, i)));
            }}}}
```

对于select_course.csv文件，清洗数据用的是python的pandas包，**去除了空格和制表符，删除了选课信息中重复的数据并并且重新调整表格形状以方便导入。**（一开始忘记去空格导致外键一直匹配不上...）

student_course表中数据处理代码：

```
    dataset = []
    with open("select_course.csv", 'r') as f:
        for line in f:
            dataset.append(list(line.strip().split(',')))
        dataframe = pd.DataFrame(dataset,
                                 columns=['name', 'sex', 'college', 'student_id', 'course1',
'course2',
                                          'course3', 'course4', 'course5', 'course6'])
        del dataframe['name']
        del dataframe['sex']
```

```
        del dataframe['college']
        f1 = dataframe[['student_id', 'course1']].rename(columns={'course1': 'course'})
        f2 = dataframe[['student_id', 'course2']].rename(columns={'course2': 'course'})
        f3 = dataframe[['student_id', 'course3']].rename(columns={'course3': 'course'})
        f4 = dataframe[['student_id', 'course4']].rename(columns={'course4': 'course'})
        f5 = dataframe[['student_id', 'course5']].rename(columns={'course5': 'course'})
        f6 = dataframe[['student_id', 'course6']].rename(columns={'course6': 'course'})
        f1 = f1.append(f2).append(f3).append(f4).append(f5).append(f6)
        f1.dropna(inplace=True)
        f1['course'] = f1['course'].str.strip()
        f1.drop_duplicates(inplace=True)
        f1.to_csv('test.csv', encoding='utf-8', index=False)
```

导入数据

json文件可以简单的转化为实例对象，所以先把json文件转成List对象再对其进行操作。这里对于所给样例代码，分别对class_detail和student表中数据做了导入速率测试（因为数据所在文件格式和数据量等级不同）：

AwfulLoader

AwfulLoader在每次insert前后都会打开数据库连接和关闭数据库连接，带来了不必要的开销，同时使用字符串构建sql语句。

	class_detail(records/s)	student(records/s)
1st	113	130
2nd	113	115
3rd	113	113
Average	113	119

VeryBadLoader

相较于AwfulLoader，VeryBadLoader复用了数据库连接。

	class_detail(records/s)	student(records/s)
1st	1134	2215
2nd	1359	2252
3rd	1152	2368
Average	1215	2278

相较于AwfulLoader，class_detail数据导入速度提升了约975%，student数据导入速率提升了约1814%

BadLoader

	class_detail(records/s)	student(records/s)
1st	1688	3781
2nd	1659	3858
3rd	1867	3727
Average	1738	3788

相较于VeryBadLoader，class_detail数据导入速度提升了约43%，student数据导入速率提升了约66%

AverageLoader

AverageLoader在打开数据库连接后，关掉了AutoCommit，并在关闭连接前一次性commit所有更改。

	class_detail(records/s)	student(records/s)
1st	2972	11013
2nd	2884	11337
3rd	2945	11640
Average	2933	11330

相较于BadLoader，class_detail数据导入速度提升了约69%，student数据导入速率提升了约199%

GoodLoader

相较于AverageLoader，GoodLoader使用了批量提交(Batch)

	class_detail(records/s)	student(records/s)
1st	5413	23320
2nd	6776	23850
3rd	6211	23784
Average	6133	23651

相较于AverageLoader，class_detail数据导入速度提升了约107%，student数据导入速率提升了约109%

最终核心代码如下：

```
//json文件导入
List<class_name> courses = gson.fromJson(content, type);//创建实例对象
start = System.currentTimeMillis();//起始时间
openDB(prop.getProperty("host"), prop.getProperty("database"),
        prop.getProperty("user"), prop.getProperty("password"));//连接数据库
        try {
            for (class_name c : courses) {
                loadData(c.courseId,c.className,c.teacher,c.classList);//向batch中添加
preparestatement
            }
            stmt.executeBatch();//执行batch中的语句
```

```
            stmt.clearBatch();//清空batch
            con.commit();
            stmt.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
closeDB();//关闭数据库
end = System.currentTimeMillis();//结束时间
System.out.printf("Time : %d", end - start);
closeDB();
```

```
//csv文件导入
openDB(prop.getProperty("host"), prop.getProperty("database"),
        prop.getProperty("user"), prop.getProperty("password"));//连接数据库
        start = System.currentTimeMillis();//起始时间
        try {
            FileReader fw = new FileReader("clean_data.csv");//将csv文件作为文本文件打开
            BufferedReader bf = new BufferedReader(fw);
            bf.readLine();
            String line;
            while((line=bf.readLine())≠null){
                String item[] = line.split(",");//以逗号做分割
                loadData(Integer.parseInt(item[3]), item[0], item[1], item[2]);
            }
            stmt.executeBatch();
            stmt.clearBatch();
            con.commit();
            stmt.close();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        } catch (SQLException throwables) {
            throwables.printStackTrace();
        }
        closeDB();
        end = System.currentTimeMillis();
        System.out.printf("Time : %dms", end - start);
        closeDB();
```

更多提升效率的方法(基于GoodLoader改进)

1.copy from

对于csv文件，可以直接执行sql命令copy from进行导入，测试结果如下表：

```
copy "student"(student_name, sex, college,student_id)  from
'/Users/lee/PycharmProjects/pythonProject/clean_data.csv' delimiter ',' csv header;
```

	student(records/s)
1st	41236
2nd	45454
3rd	38834
Average	41841

相较于GoodLoader，导入速率提升了近77%

2.关闭log，减少io

```
alter table class_detail set unlogged;
```

关闭之后重新导入，记录时间为下表：

	class_detail(records/s)	student(records/s)
1st	7125	25041
2nd	6292	25391
3rd	7021	24948
Average	6812	25127

相较于GoodLoader，class_detail数据导入速度提升了约11%，student数据导入速率提升了约6%。关闭log可以极大的减少事务记录时间，对大量数据插入优化效果比较明显。也可以把数据插入整体作为一个事务执行，也有相同的效果。

3.重建外键和主键

即创建表时不绑定外键，导入数据后再进行绑定。

```
create table student(
    student_id int ,
    student_name varchar(50),
    sex varchar(2),
    college varchar(50)
);
/* after loading */
alter table student add primary key (student_id);
alter table student add foreign key (college) references college(college_name);
create table class_detail(
    class_id varchar(50) ,
    week_list int[],
    location char(50) ,
    class_time varchar(50),
    weekday int
);
/* after loading */
alter table class_detail add primary key (weekday,class_time,class_id,location);
alter table class_detail add foreign key (class_id) references "class"(class_id);
alter table class_detail add foreign key (location) references "location"(location);
```

	class_detail(records/s)	student(records/s)
1st	15885	64845
2nd	16150	65257
3rd	15380	65113
Average	15805	65071

	class_detail(rebuild key)	student(rebuild key)
1st	29ms	5222ms
2nd	34ms	5124ms
3rd	33ms	5087ms
Average	32ms	5144ms

这种方法对于导入速率的提升极大(特别是在大数据量时)，并且重建外键的速度也十分迅速，对于student中3999920条数据，重建外键只用了5s左右的时间。

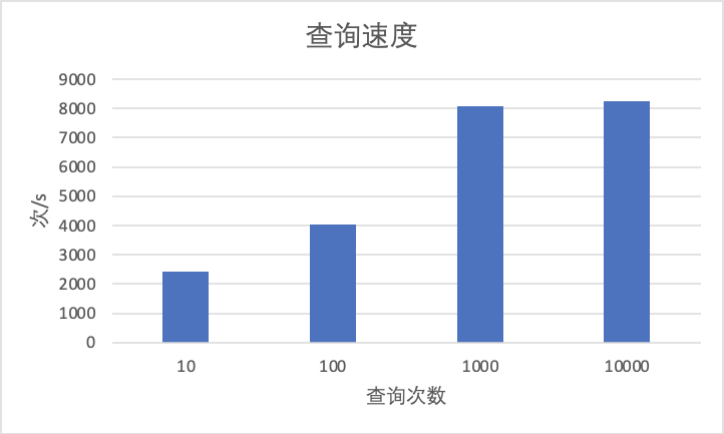
Task 3: Use DML to analyze your database

这部分使用python psycopg2包对数据库进行操作，**分别对数据库执行10，100，1000，10000次操作重复三次取平均值**

查询测试

```
if __name__ == '__main__':
    connection = psycopg2.connect(database='db_project1', user='lee', password='buzz10161',
host='localhost',
                                  port=5432)
    con = connection.cursor()
    sql = 'select * from student where student_id = %s;'
    start = time.time()
    for i in range(11000001, 11100001):
        con.execute(con.mogrify(sql, (i,)))
    conection.commit()
    end = time.time()
    con.close()
    connection.close()
    print(end - start)
```

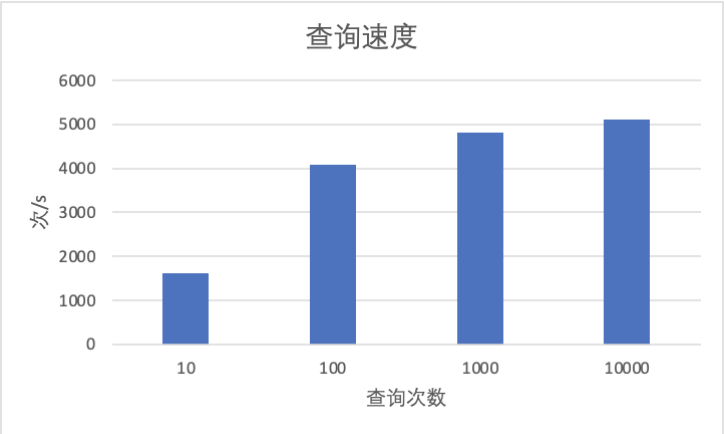
分别对数据库执行10，100，1000，10000次查询操作重复三次取平均值，结果如下表



由图可知，数据库在进行大量查询时会进行优化，查询速度会随着查询次数提升至瓶颈。

此外，为了测试数据库多表连接查询性能，设计了如下实验：

```
sql = 'select * from student s join student_course sc on sc.student_id = s.student_id where
s.student_id = %s;'
```

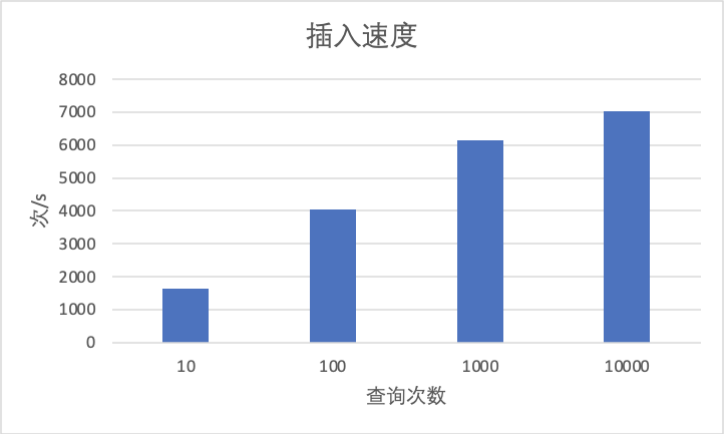


由上表可知，多表连结查询速率要明显低于单表查询。

插入测试

```
if __name__ == '__main__':
    connection = psycopg2.connect(database='db_project1', user='lee', password='buzz10161',
host='localhost',
                                  port=5432)
    con = connection.cursor()
    sql = "INSERT INTO student VALUES (%s, %s, %s, %s);"
    start = time.time()
    for i in range(1, 11):
        con.execute(sql, (i, '李子南', 'M', '格兰芬多(Gryffindor)'))
    connection.commit()
    end = time.time()
    con.execute("DELETE FROM student WHERE student_name like '%李子南%'")
    connection.commit()
    con.close()
    connection.close()
    print(end - start)
```

分别对数据库执行10，100，1000，10000次插入操作重复三次取平均值，结果如下表：



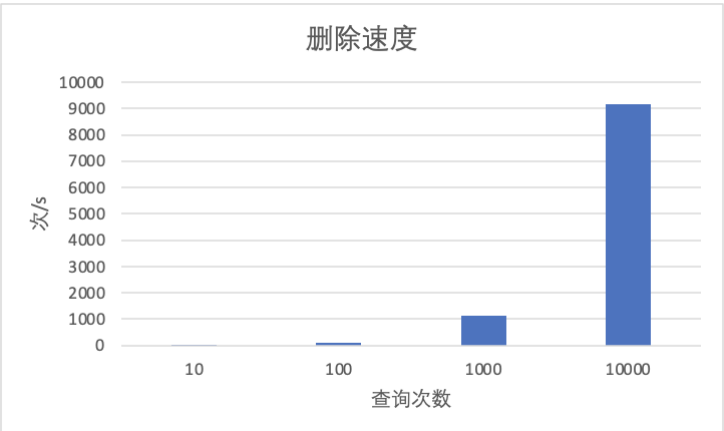
由表可知，插入速度随插入条数逐渐递增。

删除测试

单次删除多条数据

```
if __name__ == '__main__':
    connection = psycopg2.connect(database='db_project1', user='lee', password='buzz10161',
host='localhost',
                                  port=5432)
    con = connection.cursor()
    sql = "INSERT INTO student VALUES (%s, %s, %s, %s);"
    for i in range(1, 10001):
        con.execute(sql, (i, '李子南', 'M', '格兰芬多(Gryffindor)'))
    connection.commit()
    start = time.time()
    con.execute("DELETE FROM student WHERE student_name like '%李子南%'")
    connection.commit()
    end = time.time()
    con.close()
    connection.close()
    print(end - start)
```

结果如下表：



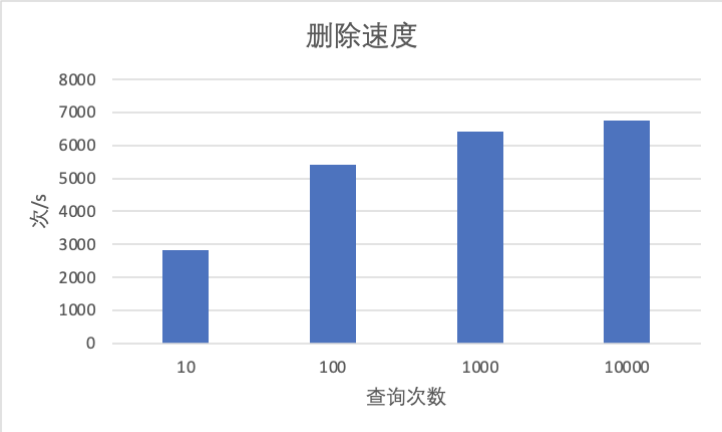
删除速度随条数提升，且提升幅度较大，推测是因为少量条数时查找占用时间较长导致。

多次删除单条数据

```
if __name__ == '__main__':
```

```
    connection = psycopg2.connect(database='db_project1', user='lee', password='buzz10161',
host='localhost',
                                  port=5432)
    con = connection.cursor()
    sql = "INSERT INTO student VALUES (%s, %s, %s, %s);"
    delete = 'DELETE FROM student WHERE student_id = %s'
    for i in range(1, 10001):
        con.execute(sql, (i, '李子南', 'M', '格兰芬多(Gryffindor)'))
    connection.commit()
    start = time.time()
    for i in range(1, 10001):
        con.execute(con.mogrify(delete, (i,)))
    connection.commit()
    end = time.time()
    con.close()
    connection.close()
    print(end - start)
```

结果如下图：



由图可知，删除速度随条数增加，且增加速率逐渐下降。

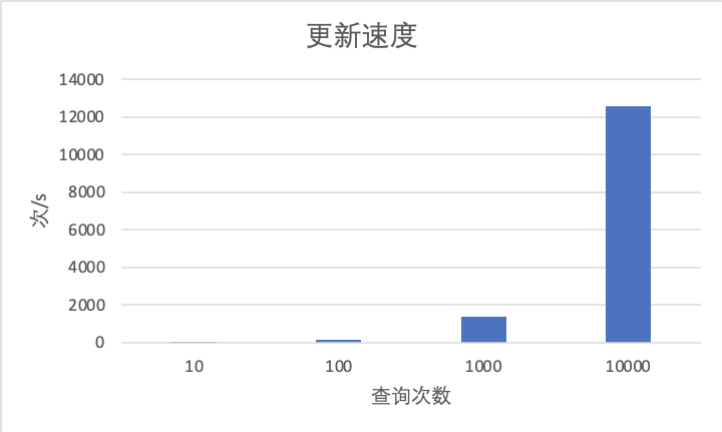
更新操作

一次更新多条数据

```
if __name__ == '__main__':
    connection = psycopg2.connect(database='db_project1', user='lee', password='buzz10161',
host='localhost',
                                  port=5432)
    con = connection.cursor()
    sql = "INSERT INTO student VALUES (%s, %s, %s, %s);"
    delete = 'DELETE FROM student WHERE student_id = %s;'
    update = "update student set student_name = 'LZN' where student_name = '李子南';"
    for i in range(1, 11):
        con.execute(sql, (i, '李子南', 'M', '格兰芬多(Gryffindor)'))
    connection.commit()
    start = time.time()
    con.execute(update)
    connection.commit()
    end = time.time()
    for i in range(1, 11):
        con.execute(con.mogrify(delete, (i,)))
```

```
    connection.commit()
    con.close()
    connection.close()
    print(end - start)
```

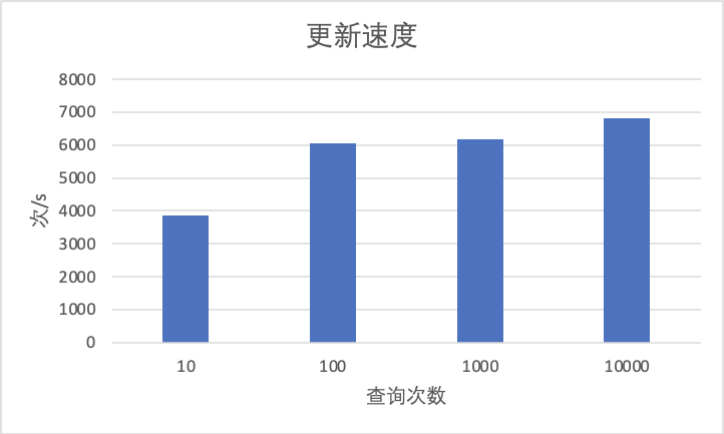
结果如下图：



由图可知，更新速度随更新次数上升。

多次更新一条数据

```
if __name__ == '__main__':
    connection = psycopg2.connect(database='db_project1', user='lee', password='buzz10161',
host='localhost',
                                  port=5432)
    con = connection.cursor()
    sql = "INSERT INTO student VALUES (%s, %s, %s, %s);"
    delete = 'DELETE FROM student WHERE student_id = %s;'
    update = "update student set student_name = 'LZN' where student_id = %s;"
    for i in range(1, 10001):
        con.execute(sql, (i, '李子南', 'M', '格兰芬多(Gryffindor)'))
    connection.commit()
    start = time.time()
    for i in range(1, 10001):
        con.execute(con.mogrify(update, (i,)))
    connection.commit()
    end = time.time()
    for i in range(1, 10001):
        con.execute(con.mogrify(delete, (i,)))
    connection.commit()
    con.close()
    connection.close()
    print(end - start)
```

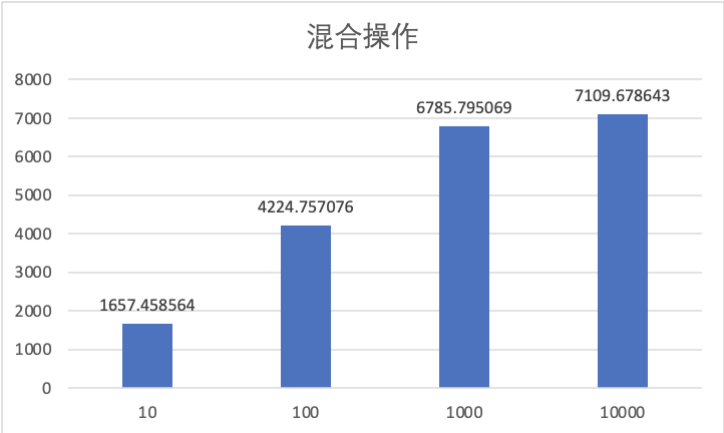


由上图可知，更新速度随条数增加，且增加速率逐渐下降。

混合操作测试

```
if __name__ == '__main__':
    connection = psycopg2.connect(database='db_project1', user='lee', password='buzz10161',
host='localhost',
                                  port=5432)
    con = connection.cursor()
    select = 'select * from student where student_id = %s;'
    insert = "INSERT INTO student VALUES (%s, %s, %s, %s);"
    delete = 'DELETE FROM student WHERE student_id = %s;'
    update = "UPDATE student SET student_name = 'LZN' WHERE student_id = %s;"
    start = time.time()
    for i in range(1, 10001):
        rand = random.randint(1, 5)
        if rand == 1:
            con.execute(con.mogrify(select, (i,)))
        elif rand == 2:
            con.execute(insert, (i, '李子南', 'M', '格兰芬多(Gryffindor)'))
        elif rand == 3:
            con.execute(con.mogrify(delete, (i - 4)))
        else:
            con.execute(con.mogrify(update, (i - 4)))
    connection.commit()
    end = time.time()
    con.execute("DELETE FROM student WHERE student_name LIKE '%李子南%' OR student_name LIKE
'%LZN%';")
    connection.commit()
    con.close()
    connection.close()
    print(time.time())
```

随机进行10，100，1000，10000次操作测试速度，结果如下图



总结

数据库的查询等操作有一定的性能瓶颈，且数据库在执行大量操作时会自动进行优化。比起多次少量操作，一次操作大量数据效率更高，在操作数据库时应考虑这一点。

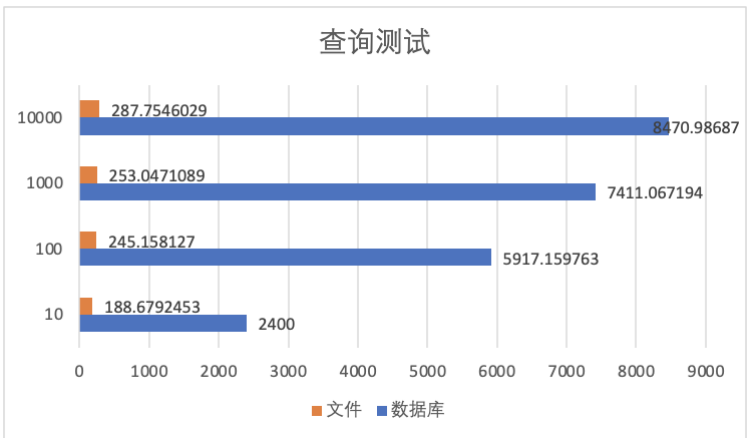
Task 4: Compare database and file

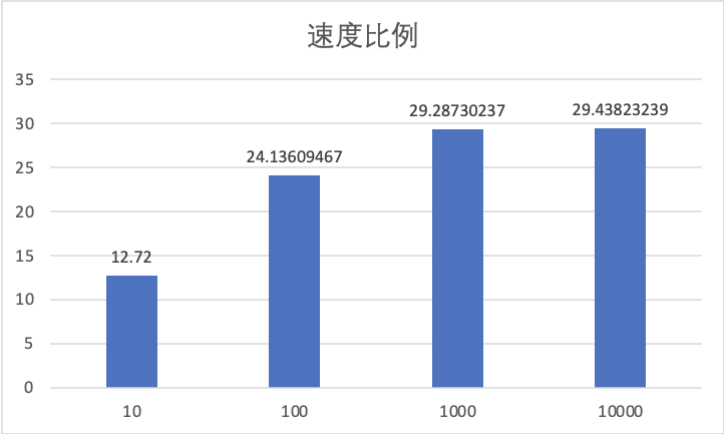
为了控制变量，这里选取与数据库表格结构更为相似的csv文件进行测试。这部分将使用python对文件和数据库进行相同操作，**分别对数据库和文件执行10，100，1000，10000次操作重复三次取平均值**，并记录文件的执行时间且和数据库执行速度相比较。

查询测试

```
    file = pandas.read_csv('clean_data.csv')
    result = []
    start2 = time.time()
    for i in range(11000001, 11000011):
        result.append(file[file['student_id'] == i])
    end2 = time.time()
    print(end2 - start2)
```

结果如下图



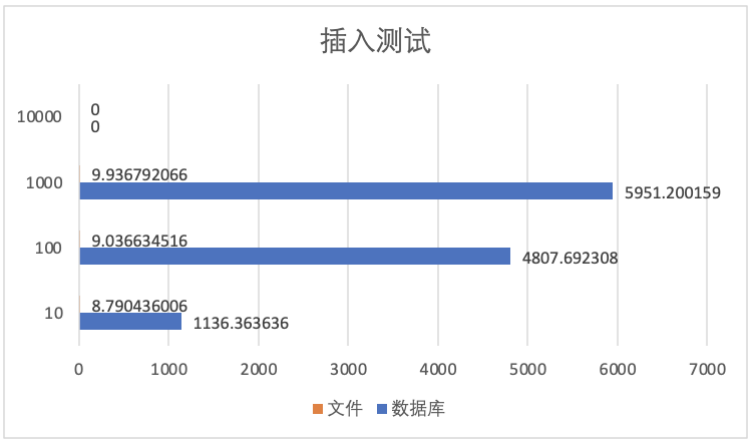


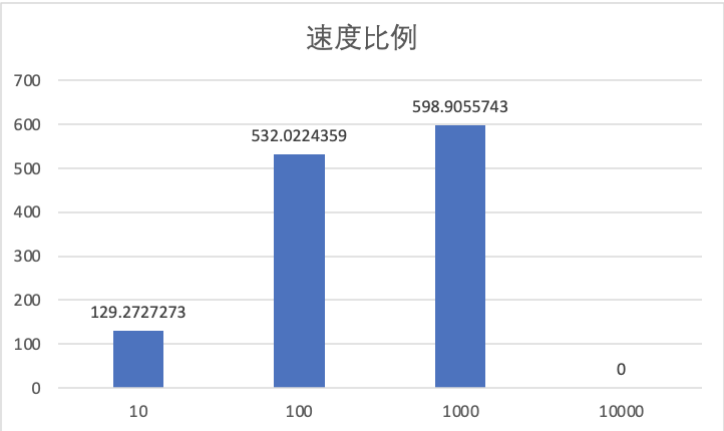
由图可知，文件的查询效率远远低于数据库，且查询数据量越大数据库优势越明显。

插入测试

```
    file = pandas.read_csv('clean_data.csv')
    start2 = time.time()
    for i in range(1, 11):
        file.append([{'name': '李子南', 'sex': 'M', 'college': '格兰芬多(Gryffindor)',
'student_id': i}],
                    ignore_index=True)
    end2 = time.time()
    file = file[~file['name'].isin(['李子南'])]
    print(end2 - start2)
```

结果如下图





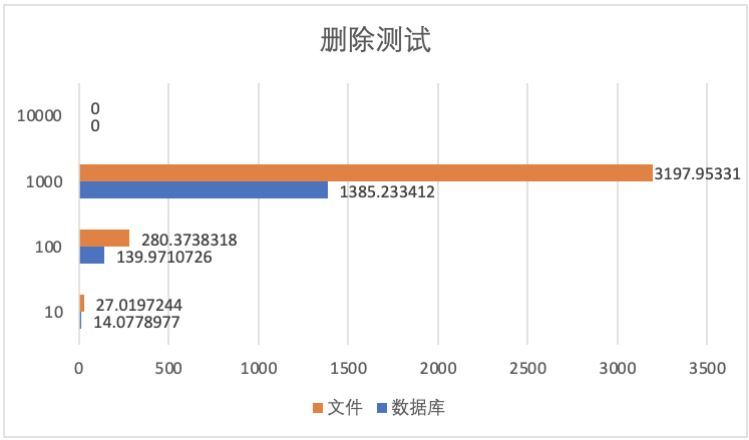
由图可知，文件的写入速率慢得令人发指，其比例达到了将近600倍，因为写入速率实在太慢故不测试10000条数据插入。

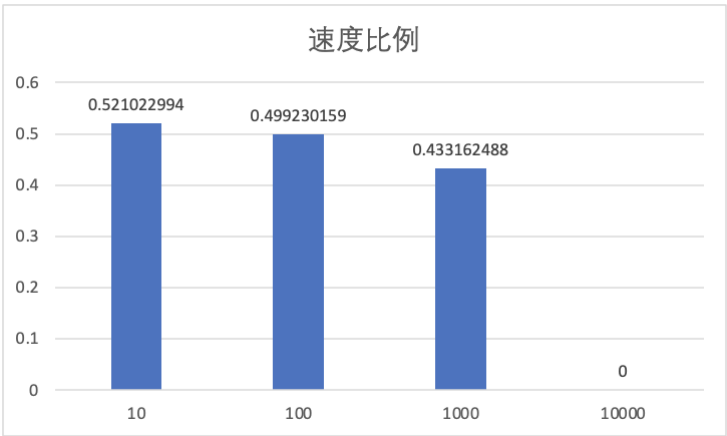
删除测试

一次删除多条

```
    file = pandas.read_csv('clean_data.csv')
    for i in range(1, 1001):
        file.append([{'name': '李子南', 'sex': 'M', 'college': '格兰芬多(Gryffindor)',
'student_id': i}],
                    ignore_index=True)
    start2 = time.time()
    file = file[~file['name'].isin(['李子南'])]
    end2 = time.time()
    print(end2 - start2)
```

结果如下图

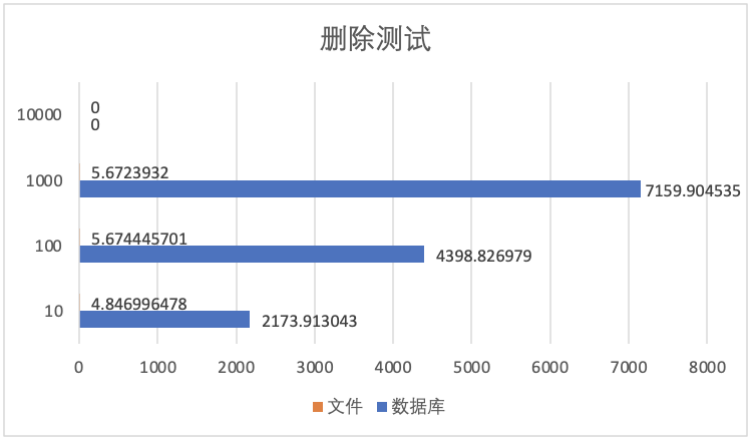


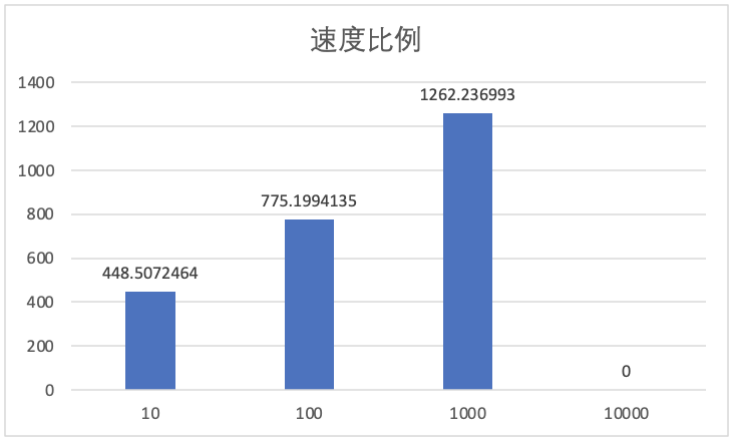


结果出乎意料，文件的删除速度比数据库要更快，且快了近一倍。

多次删除一条数据

```
    file = pandas.read_csv('clean_data.csv')
    for i in range(1, 101):
        file.append([{'name': '李子南', 'sex': 'M', 'college': '格兰芬多(Gryffindor)',
'student_id': i}],
                    ignore_index=True)
    start2 = time.time()
    for i in range(1, 101):
        file = file[~file['student_id'].isin([i])]
    end2 = time.time()
    print(end2 - start2)
```





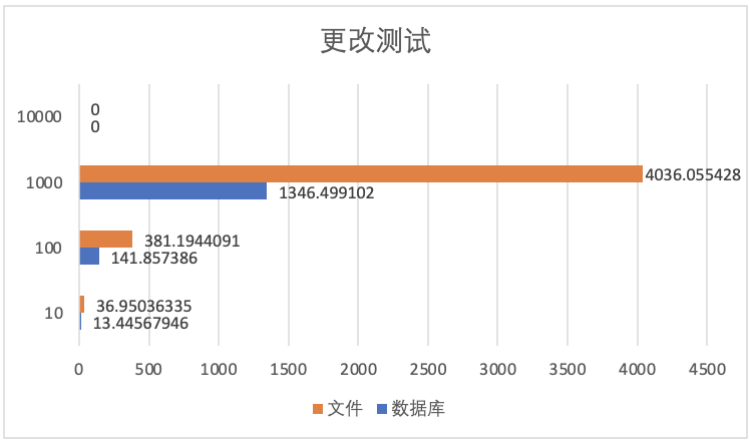
文件在应对大量操作时的性能显然远远低于数据库。

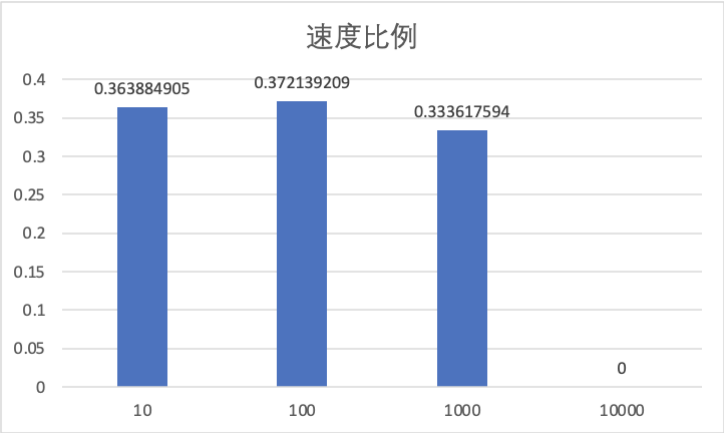
更新测试

一次更新多条数据

```
    file = pandas.read_csv('clean_data.csv')
    for i in range(1, 1001):
        file.append([{'name': '李子南', 'sex': 'M', 'college': '格兰芬多(Gryffindor)',
'student_id': i}],
                    ignore_index=True)
    start2 = time.time()
    file['name'][file['name'] == '李子南'] = 'LZN'
    end2 = time.time()
    file = file[~file['student_id'].isin(['LZN'])]
    print(end2 - start2)
```

结果如下图



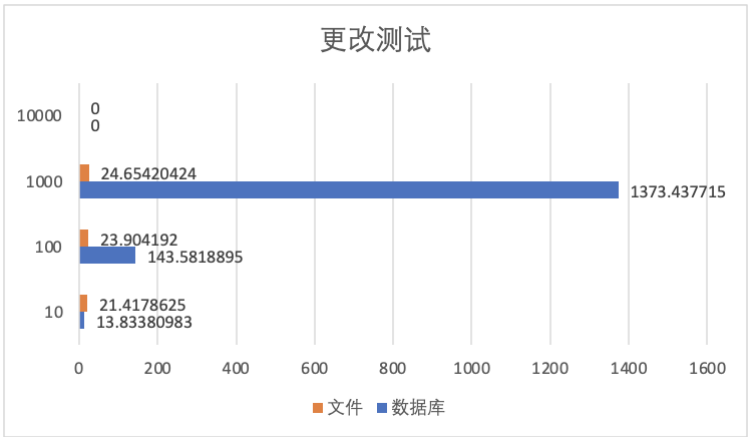


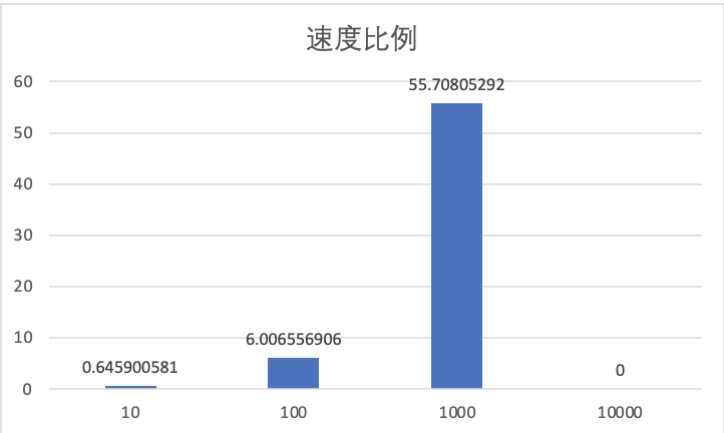
由上表可以看出，文件在单次操作时的速度要高于数据库。

多次更新单条数据

```
file = pandas.read_csv('clean_data.csv')
    for i in range(1, 11):
        file.append([{'name': '李子南', 'sex': 'M', 'college': '格兰芬多(Gryffindor)',
'student_id': i}],
                    ignore_index=True)
    start2 = time.time()
    for i in range(1, 11):
        file['name'][file['student_id'] == i] = 'LZN'
    end2 = time.time()
    file = file[~file['student_id'].isin(['LZN'])]
    print(end2 - start2)
```

结果如下图





总结

由上述测试可以发现，文件的各种操作的耗时几乎和次数成正比，即执行效率一定。而数据库的耗时与次数成对数关系，在数据量较大时，效率增长趋于平滑。数据库在应对大量操作时的表现远远优于数据库，推测是因为数据库使用了B+Tree等数据结构，将原本O(n)的复杂度优化成了O(logn)，大大提升了操作效率。

文件的优点：

1. 分布式方案使得文件的数据储存能力要优于数据库。
2. 数据的冗余保证了高可用性。
3. 在单次大量删除和更改时性能优于数据库。

文件的缺点：

1. 在执行大量操作时速率极低。
2. 统一数据可能会以不同的格式存储，造成数据冗余。
3. 事务管理困难。

数据库的优点

1. 数据库对事务的支持较好，能够保证事务的原子性，保证数据安全。
2. DBMS使得用户可以简单的操作数据。
3. 支持高并发访问。
4. 数据库的日志管理更为完善。
5. 权限管理系统

数据库的缺点

1. 数据没有冗余，高可用性不如文件。

Task 5: Bonus part

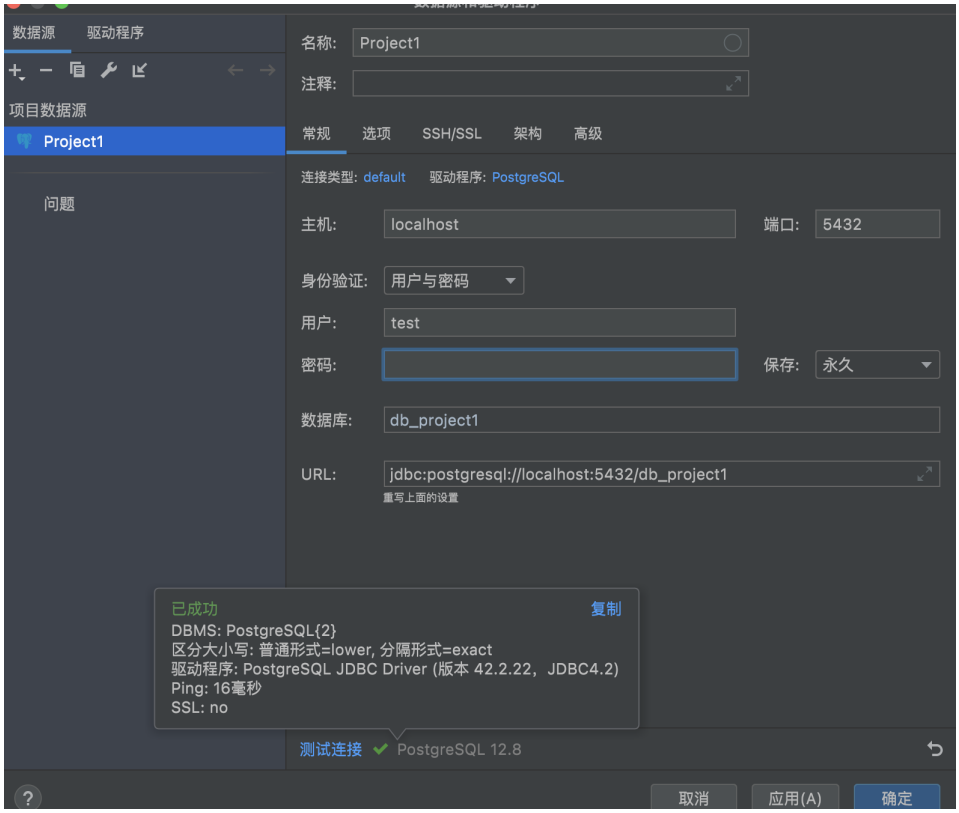
用户权限管理

为了方便数据库的维护和更新，用户权限应该总共有四个等级：学生，老师，管理员和超级管理员。其中超级管理员拥有数据库的所有权限，老师拥有student_course表的insert和delete权限，以及所有表的查询权限。**老师拥有对选课名单进行修改的权限，但是开课或者修改课程信息仍然需要向教务处提交申请，申请通过后由管理员进行修改。（老师只能修改自己课程的学生这个条件由前端判断）**学生具有student_course表的insert和delete权限(选课退课)，**以及除了student_course，student和college之外所有表的查询权限(限制学生查看别的学生的选课情况)。管理员具有老师和学生的所有权限且可以修改老师和学生的密码**，可以直接执行sql命令(前端)。

创建一个测试账户，赋予学生权限：

```
create user test with password '123456';
grant insert, delete on student_course to test;
grant select on
course,class,class_detail,class_teacher,course_prerequisite,dept,location,teacher to test;
```

用此用户连接数据库：



测试权限：

```
db_project1.public> truncate table dept
[2021-11-08 23:00:22] [42501] ERROR: permission denied for table dept
db_project1.public> delete
                    from student
                    where sex = 'F'
[2021-11-08 23:00:29] [42501] ERROR: permission denied for table student
db_project1.public> update teacher
                    set teacher_id = '666666666'
                    where teacher_name = 'C'
[2021-11-08 23:00:40] [42501] ERROR: permission denied for table teacher
db_project1.public> select *
                    from course c
                             join student_course sc on c.course_id = sc.course_id
                             join student s on sc.student_id = s.student_id
                    where sex = 'F'
                      and sc.course_id = 'CS307'
[2021-11-08 23:00:44] [42501] ERROR: permission denied for table student_course
db_project1.public> select *
                    from course c
                             join class c2 on c.course_id = c2.course_id
                             join class_detail cd on c2.class_id = cd.class_id
                    where location = '排球场'
[2021-11-08 23:00:49] 在 71 ms (execution: 11 ms, fetching: 60 ms) 内检索到从 1 开始的 2 行
```

```
db_project1.public> insert into student_course values (11000009,'ESE216')
[2021-11-09 00:00:36] 4 ms 中有 1 行受到影响
```

创建教师权限账户：

```
create user test_teacher with password '654321';
grant select on
class,class_detail,course,college,course_prerequisite,dept,location,student_course,student,teac
her to test_teacher;
grant insert,delete on student_course to test_teacher;
```

测试权限：

```
db_project1.public> truncate table dept
[2021-11-09 00:10:26] [42501] ERROR: permission denied for table dept
db_project1.public> delete
                    from student
                    where sex = 'F'
[2021-11-09 00:10:30] [42501] ERROR: permission denied for table student
db_project1.public> update teacher
                    set teacher_id = '666666666'
                    where teacher_name = 'C'
[2021-11-09 00:10:34] [42501] ERROR: permission denied for table teacher
db_project1.public> select *
                    from course c
                             join student_course sc on c.course_id = sc.course_id
                             join student s on sc.student_id = s.student_id
                    where sex = 'F'
                      and sc.course_id = 'CS307'
[2021-11-09 00:10:40] 在 2 s 726 ms (execution: 2 s 644 ms, fetching: 82 ms) 内检索到从 1 开始的 500 行
db_project1.public> select *
                    from course c
                             join class c2 on c.course_id = c2.course_id
                             join class_detail cd on c2.class_id = cd.class_id
                    where location = '排球场'
[2021-11-09 00:10:47] 在 83 ms (execution: 7 ms, fetching: 76 ms) 内检索到从 1 开始的 2 行
db_project1.public> insert into student_course values (11000009,'ESE216')
[2021-11-09 00:10:52] 4 ms 中有 1 行受到影响
db_project1.public> delete from student_course where course_id = 'ESE216' and student_id = 11000009
[2021-11-09 00:10:55] 3 ms 中有 1 行受到影响
```

测试过程中发现若是不给予学生student_course表的select权限，学生账户就无法删除student_course中的数据，**推测是因为select权限涉及到数据库信息筛选，导致delete时where语句后面的条件请求被数据库拒绝。**这样若是要限制学生用户读取student_course表中的数据则需要在前端进行限制。

事务管理

对于一个数据库来说，事务管理是不可或缺的部分，正确的事务管理能让数据库内存储的数据从失败中恢复到正常，同时提供了数据库即使在异常状态下仍能保持一致性的方法。数据库事务的隔离级别如下表：

Isolation Level	Dirty Read	Nonrepeatable Read	Phantom Read	Serialization Anomaly
Read uncommitted	not in PG	possible	possible	possible
Read committed	not possible	possible	possible	possible
Repeatable read	not possible	not possible	not in PG	possible
Serializable	not possible	not possible	not possible	not possible

隔离级别由上到下依次递增，并且隔离级别越高，并发性能越差，但越能保证数据的完整性。pg数据库的默认隔离级别为read committed。

测试（分别开启两个事务）

1. **以默认隔离级别执行**

```
/* phantom read */
begin transaction;
insert into student values (666,'李子南','M','格兰芬多(Gryffindor)');
commit;
```

```
/* phantom read */
begin transaction ;
select * from student where student_name like '%李子南%';
/*after other transaction commit*/
select * from student where student_name like '%李子南%';
end transaction ;
```

运行结果：

```
db_project1.public> begin transaction
[2021-11-24 00:57:24] 在 3 ms 内完成
db_project1.public> select * from student where student_name like '%李子南%'
[2021-11-24 00:57:28] 在 824 ms (execution: 808 ms, fetching: 16 ms) 内检索到 0 行
db_project1.public> begin transaction
[2021-11-24 00:57:17] 在 1 ms 内完成
db_project1.public> insert into student values (666,'李子南','M','格兰芬多(Gryffindor)')
[2021-11-24 00:57:34] 2 ms 中有 1 行受到影响
db_project1.public> select * from student where student_name like '%李子南%'
[2021-11-24 00:57:41] 在 804 ms (execution: 787 ms, fetching: 17 ms) 内检索到 0 行
db_project1.public> commit
[2021-11-24 00:57:51] 在 3 ms 内完成
db_project1.public> select * from student where student_name like '%李子南%'
[2021-11-24 00:57:57] 在 813 ms (execution: 795 ms, fetching: 18 ms) 内检索到从 1 开始的 1 行
db_project1.public> end transaction
[2021-11-24 00:58:01] 在 1 ms 内完成
```

由运行结果可知，出现了幻读的情况，但脏读被避免了，pgsql的默认隔离等级应该是**Read committed**。

2. 以Read uncommitted等级执行（测试脏读）

```
/* dirty read*/
begin transaction;
set transaction isolation level read uncommitted;
insert into student values (666,'李子南','M','格兰芬多(Gryffindor)');
rollback;
```

```
/* dirty read */
begin transaction;
set transaction isolation level read uncommitted;
select * from student where student_name like '%李子南%';
end transaction ;
```

运行结果：

```
db_project1.public> begin transaction
[2021-11-24 11:18:07] 在 3 ms 内完成
db_project1.public> set transaction isolation level read uncommitted
[2021-11-24 11:18:07] 在 2 ms 内完成
db_project1.public> begin transaction
[2021-11-24 11:18:17] 在 4 ms 内完成
db_project1.public> set transaction isolation level read uncommitted
[2021-11-24 11:18:17] 在 1 ms 内完成
db_project1.public> select * from student where student_name like '%李子南%'
[2021-11-24 11:18:21] 在 783 ms (execution: 769 ms, fetching: 14 ms) 内检索到 0 行
db_project1.public> insert into student values (666,'李子南','M','格兰芬多(Gryffindor)')
[2021-11-24 11:18:29] 2 ms 中有 1 行受到影响
db_project1.public> select * from student where student_name like '%李子南%'
[2021-11-24 11:18:35] 在 795 ms (execution: 777 ms, fetching: 18 ms) 内检索到 0 行
```

由运行结果可知，在readuncommitted等级下，pgsql也避免了脏读的出现，但其他数据库在read uncommitted下允许脏读。

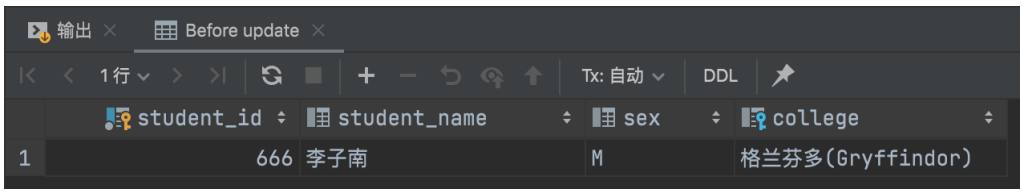
3. 以Repeatable read等级执行（测试不可重复读和幻读）

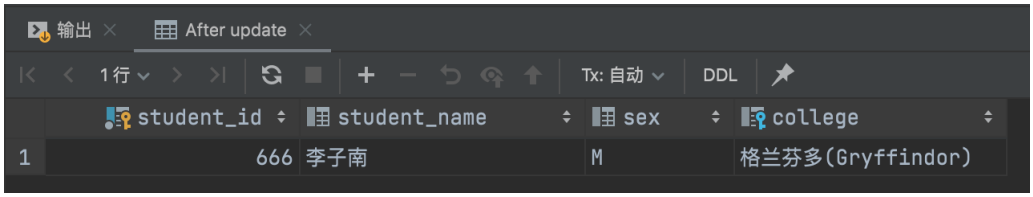
   不可重复读

```sql
/* Nonrepeatable Read */
begin transaction;
set transaction isolation level repeatable read;
/*Before update*/
select * from student where student_name like '%李子南%';
/* After update */
select * from student where student_name like '%李子南%';
commit;
```

```sql
/* Nonrepeatable Read */
begin transaction;
set transaction isolation level repeatable read;
update student set student_id = 777 where student_name like '李子南';
commit;
```

运行结果：





在Repeatable read等级下，不可重复读被有效避免了。

幻读测试：

```
db_project1.public> begin transaction
[2021-11-24 11:51:08] 在 1 ms 内完成
db_project1.public> set transaction isolation level repeatable read
[2021-11-24 11:51:08] 在 2 ms 内完成
db_project1.public> select * from student where student_name like '%李子南%'
[2021-11-24 11:51:12] 在 781 ms (execution: 768 ms, fetching: 13 ms) 内检索到 0 行
```

```
db_project1.public> begin transaction
[2021-11-24 11:50:59] 在 2 ms 内完成
db_project1.public> set transaction isolation level repeatable read
[2021-11-24 11:50:59] 在 1 ms 内完成
db_project1.public> insert into student values (666,'李子南','M','格兰芬多(Gryffindor)')
[2021-11-24 11:51:34] 3 ms 中有 1 行受到影响
db_project1.public> commit
[2021-11-24 11:51:34] 在 3 ms 内完成
```

```
db_project1.public> select * from student where student_name like '%李子南%'
[2021-11-24 11:51:42] 在 791 ms (execution: 774 ms, fetching: 17 ms) 内检索到 0 行
```

由测试结果可知，在pgsql中，Repeatable read中幻读也不可能出现。

不同隔离等级下高并发查询速率

详见高并发部分

索引

对student表中的student_name列创建索引。

```
create index student_name_index on student(student_name);
```

用python多线程模拟高并发环境，测试查找速率，并与创建索引前进行对比

```
tmp = []
def select():
    s = "select * from student where student_name = '李谢恩';"
    connection = psycopg2.connect(database='db_project1', user='lee', password='buzz10161',
host='localhost',
                                  port=5432)
    con = connection.cursor()
    start = time.time()
    con.execute(s)
    connection.commit()
    con.fetchone()
    end = time.time()
    con.close()
    connection.close()
    tmp.append((end - start))
if __name__ == '__main__':
    threads = []
    for i in range(1, 100):
```

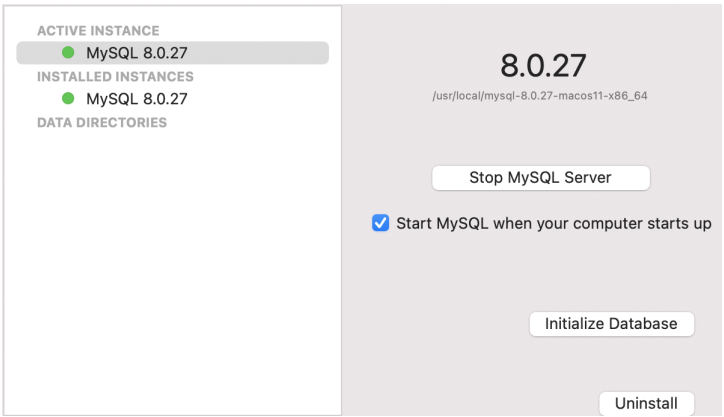
```
        threads.append(threading.Thread(target=select()))
    for t in threads:
        t.start()
    tt = 0
    for i in tmp:
        tt = tt + i
    print('Average time: ' + str(round((tt / 100), 4)))
```

测试结果：

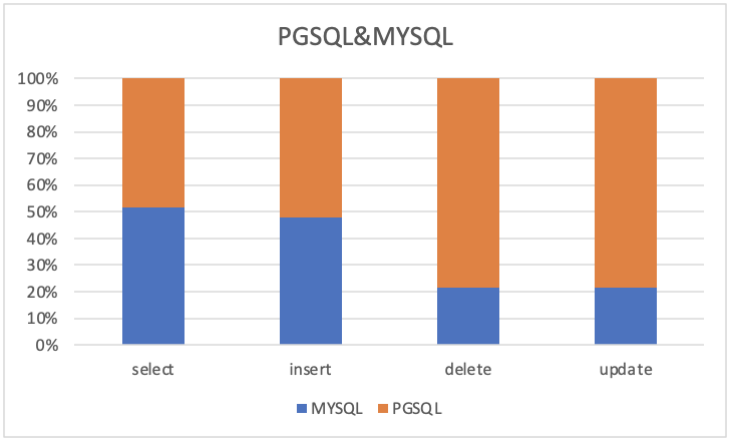
	Time(with index)	Time(without index)
1st	0.003s	0.3013
2nd	0.0025s	0.2622
3rd	0.0026s	0.2519
Average	0.0027	0.2718

由测试结果可以看出，加入索引后检索速度提升了约100倍

pg数据库与mysql数据库对比



这里使用mysql8.0.27版本进行对比测试，为了避免python中不同包对执行时间的影响（尝试的时候mysql的包慢的离谱），这部分直接使用sql命令行进行测试，连续测试三次求平均。（测试的时候还发现第一次执行的时间要略长，推测是第一次执行时恢复数据库连接占用了时间）对比结果如下：



图中数据所占比例越大代表效率越高，由图可知PGSQL的部分性能与MYSQL接近，delete和update性能要高于MYSQL

高并发测试

使用python多线程同时对数据库发出请求，模拟高并发情况，测试数据库执行效率

```
tmp = []
def select():
    s = "select * from student where student_name = '李谢恩';"
    connection = psycopg2.connect(database='db_project1', user='lee', password='buzz10161',
host='localhost',
                                  port=5432)
    con = connection.cursor()
    start = time.time()
    con.execute(s)
    connection.commit()
    con.fetchone()
    end = time.time()
    con.close()
    connection.close()
    tmp.append((end - start))

if __name__ == '__main__':
    threads = []
    for i in range(1, 100):
        threads.append(threading.Thread(target=select()))
    for t in threads:
        t.start()
    tt = 0
    for i in tmp:
        tt = tt + i
    print('Average time: ' + str(round((tt / 100), 4)))
```

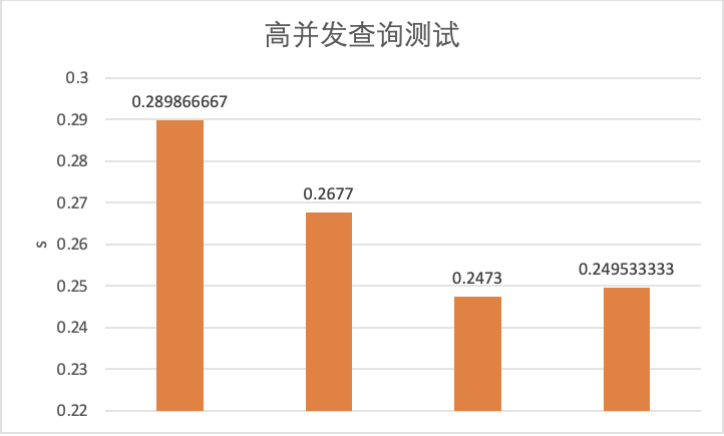
测试三次求平均，平均时间为0.26s，相比于单客户端直接查询100次稍慢，但还不足以看出明显区别。

故第二次测试使用500个线程模拟，平均时间0.24378s，单客户端500次查询平均0.24s。差别还不是很大。

不同隔离等级对并发性能的影响

```
begin transaction;
set transaction isolation level ----;
```

在执行前设置隔离等级进行测试，使用100个线程模拟，结果如下。



上图从左到右分别为serializable，repeatable read，read committed，read uncommitted。由图可知随着隔离等级上升，并发性能确实有所下降。read committed和read uncommitted等级的查询速度基本持平，推测是因为在pgsql中这两种隔离等级的限制基本一样导致的。

索引对并发性能的影响

详见上文索引部分。

其他提升高并发性能的方法

1. 延迟修改，高并发状况下可以把多次修改请求先保存到缓存中，之后再定时执行缓存中的请求。
2. 批量提交，可以把多次请求合并为单次，减少链接数据库开销。
3. 分表，可以把请求次数较多的数据单独建表，比如大一通识课查询量较大，则可以为这些数据单独建表，先在此表查询，如果没有找到再去主表找。

总结

本项目设计实现了一个包含课程数据，学生，老师选课任课情况的数据库，并对先修课等复杂关系进行了一定的处理。并在导入数据的时候总共实现了包括批量提交，重建外键和主键等7种提高效率的方法。在完成数据的导入之后，使用python程序对数据库和文件进行操作，分别测试了文件和数据库的各种性能，总结了数据库和文件分别的优点和缺点。此外，还对用户权限管理，数据库事务管理，索引和高并发进行了一定程度上的探索。最后还将PGSQL和MYSQL做了横向对比，比较了两个数据库的性能。

附github链接（目前私有）：https://github.com/Buzzy0423/SUSTECH_2021_Fall