

# CS307 Project1 Report

Course ID: CS307

Course Name: Database Principle(数据库原理)

Semester: 2021Spring

Lecture Teacher: Shiqi Yu(于仕琪)

Lab Teacher: Yueming Zhu(朱悦铭)

Lab: Wednesday 5-6

Student Assistances: Ziqin Wang(王子勤), Tiancheng Yu(余添诚)

## Part 0. Group Info and Contribution-小组分工与贡献

小组成员数: 2

小组成员列表:

姓名	学号	专业	书院	操作系统
王奕童	11910104	计算机科学与技术	致新书院	Windows 10
张彤	11911831	计算机科学与技术	树德书院	Mac Book

小组内部分工与贡献:

### 【王奕童】

1. 数据库表设计;
2. 数据库小组内同步工作;
3. 数据库数据导入的实现, 比较与优化;
4. 数据库数据全部导入的验证;
5. 数据库的多线程与高并发操作;
6. 数据库事务管理;
7. 数据库索引功能探索;
8. 数据库与 Java 的效能测试;
9. 数据库在 Windows 系统下的性能测试;

### 【张彤】

1. 数据库表设计;
2. 数据库小组内不同系统下的数据库同步与协调问题;
3. 数据库数据导入的实现与优化;
4. 数据库数据全部导入的验证;
5. 与小组成员相互协调达成数据的同步更新;
6. 针对数据导入不符合预期的情况进行调整, 确保结果符合预期且不影响表与表之间的关系;
7. 实现文件存储到 RAM
8. 用户管理权限的实验分析
9. 在 Mac 系统上跑实验进行性能测试
10. 在 DataGrip 上进行数据库表的简单分析

## Part 1. Task 1-建表设计

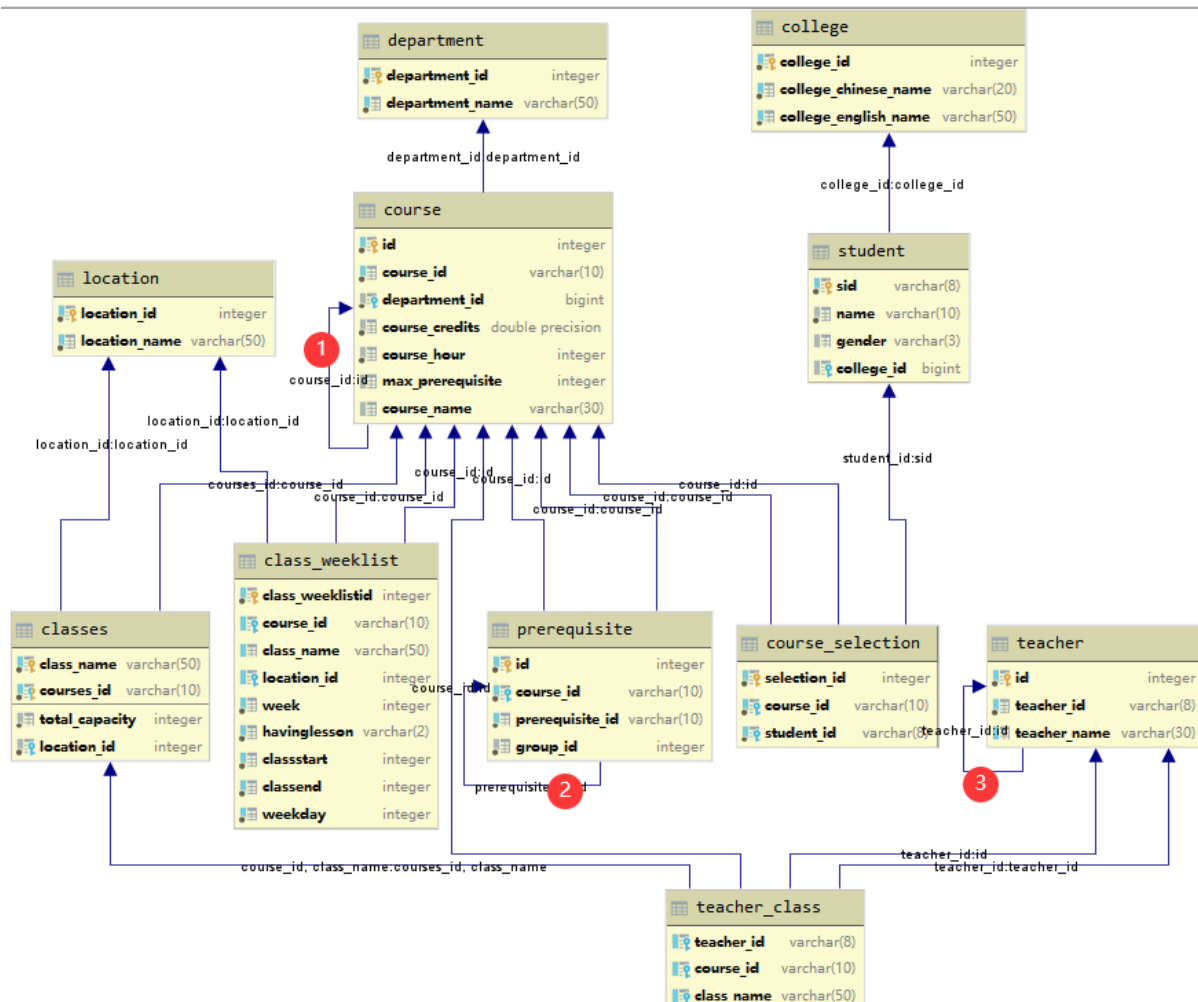
### 【任务需求】

1. 用 PostgreSQL 设计一个数据库, 能够表示在 course\_info.json 和 select\_course.csv 文件中的所有信息
2. 表的设计需要符合三大范式:

Every non key attribute must provide a fact about the key, the whole key, and nothing but the key.

3. 使用 primary key 和 foreign key 指向数据中重要属性和数据之间的关系  
--表中每一行的信息能够被唯一的 primary key 映射到
4. 每个表要有外键，或者其他表的外键指向；
5. 表之间的外键关系不能有环；
6. 每个表之间至少要不有一列 not null 列（包括 primary key 列，但是不包括 serial-id 列）；
7. 对于不同种类的数据使用合适的数据类型；
8. 设计要尽可能地易于扩展（对于三人小组的要求）

## 【表之间关系图】



【NOTICE】在设计图中，1,2,3 并非自己指向自己的外键关系，因此并没有形成外键关系的环。

## 【关于表设计的解释】

本次数据库 project 经过不断的修改和更迭，最终形成了一共有 11 个表格的设计。这 11 个表格总体来看呈树状结构，因此具有结构清晰，易于插入数据的优点。

这个环节主要分为以下几个部分的介绍：

## 【三大范式体现】

数据库三大范式：（部分截取自于仕琪老师 Lecture1 的课件）

### Simple attributes

attributes depend on  
the **full** key

non-key attributes

not depend on each other

【第一范式】要求数据库表的每一列都是不可分割的原子数据项。

【第二范式】在第一范式的基础上，非码属性必须完全依赖于候选码（在第一范式基础上消除非主属性对主码的部分函数依赖）

==>第二范式需要确保数据库表中的每一列都和主键相关，而不能只与主键的某一部分相关（主要针对联合主键而言）。

【第三范式】在第二范式基础上，任何非主属性不依赖于其它非主属性（在第二范式基础上消除传递依赖）

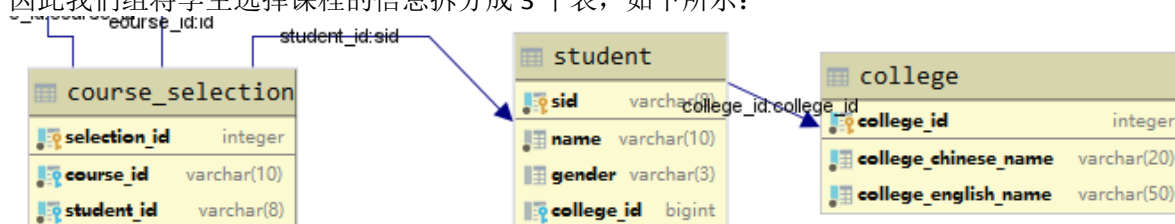
==>第三范式需要确保数据表中的每一列数据都和主键直接相关，而不能间接相关。

通过对原始 CSV 数据格式的分析（如下所示），我们很容易发现 CSV 数据**违反了第一范式和第二范式**，不能直接插入数据库中（违反第一范式因为后续的课程是可以拆分的非原子域，违反第二范式是因为书院中英文名不是直接与学生相关）。

薛采街,M,格兰芬多(Gryffindor),14999993,FIN206,IPE101,CS202

黄致新,F,阿兹卡班(Azkaban),14999994,EE106,MA403

因此我们组将学生选择课程的信息拆分成 3 个表，如下所示：



这样的三个表就能较好地反映出学生、书院以及课程之间的关系，较好地符合数据库三大范式设计的要求。

再考虑以下 json 文件中的部分内容：

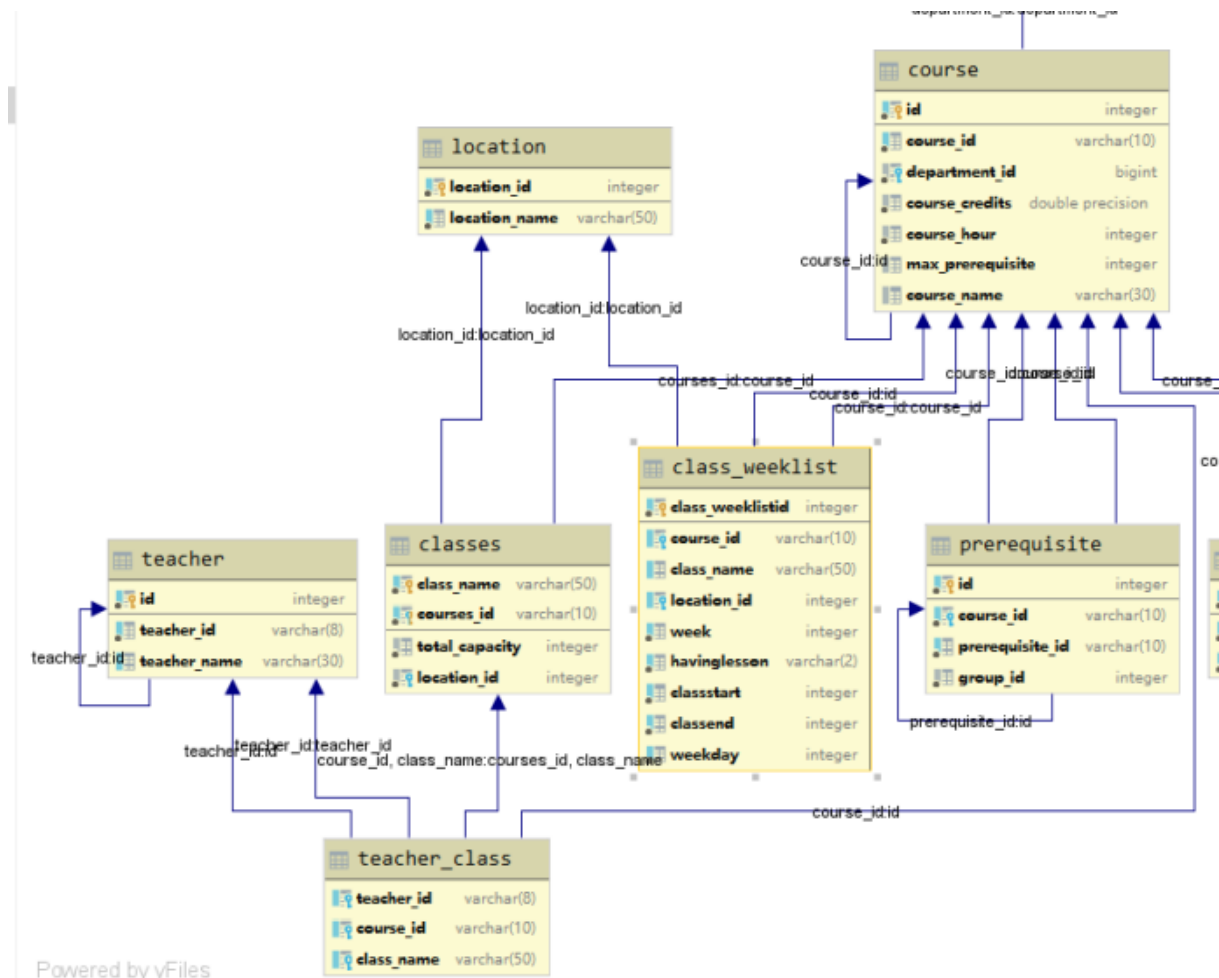
"courseName": "材料学综合实验II",

"className": "英文3班（实验课每次都计入期末成绩，且没有补课）",

"teacher": "章剑波,明静,程化,王海鸥,李艳艳",

{ "weekList": ["1", "2", "3", "4", "5", "6", "7", "8", "9", "10", "11", "12", "13", "14", "15"], "location": "一教406", "classTime": "3-4", "weekday": 1 }],

这个解析出来的 json 文件内容如果插入数据库的话，是违反范式要求的（不是最小原子域，而且有相互关联的非主键成分）。因此我们考虑使用以下几个表来实现以下功能：



如此，先修关系和师生上课关系都能得到较好地通过外键关联的方式实现数据库设计。

### 【表之间的外键关系】

### 【各表的主键】

### 【各表中的非自增 id 的非空列】

以上 3 个部分可以详见于附报告的“表格”文件中。

经过以上部分的介绍，我们可以得出结论：本数据库使用了较为合适的数据类型来表示相应的信息，并且每个表之间均有非自增 id 的非空列元素和相应的主键。不仅如此，表之间的外键关系没有形成闭合的环。因此，我们小组的数据库设计基本符合 Task1 的要求。

## Part 2. Task 2-数据导入

### 【任务需求】

1. 设计能够从两个文件中导入数据库的脚本文件；
2. 寻求能够提高数据导入效率的方法，并且对不同的数据导入方法进行比较；
3. 确保所有数据成功导入数据库；(对于三人小组的要求)

### 【脚本设计】

代码请见于随上传的.java 文件和.sql 文件

## 【困难与解决方案】

一>**遇到困难 1**：字符串表示内容(如课程 id，先修课程，上课地点)的形式不规范，如中英文全半角括号混用、课程代码大小写和空格混乱等等；

**解决方法 1**：将字符串内不规范的内容使用命令强制规范化处理。

一>**遇到困难 2**：先修课程的字符串内部包含有各种信息，为解析成课程之间的先修关系带来了一定困难；

**解决方法 2**：主要分为以下几个步骤↓

以下列先修课程为例：

```
"prerequisite":  
"(计算机程序设计基础 或者 计算机编程基础 或者 计算机程序设计基础A)  
并且 (数据结构与算法分析 或者 数据结构与算法设计)  
并且 计算机组成原理"
```

1. 将先修课程规范化处理：（全角括号全部转化为半角括号等等操作）

```
"prerequisite":  
"(计算机程序设计基础 或者 计算机编程基础 或者 计算机程序设计基础A)  
并且 (数据结构与算法分析 或者 数据结构与算法设计)  
并且 计算机组成原理"
```

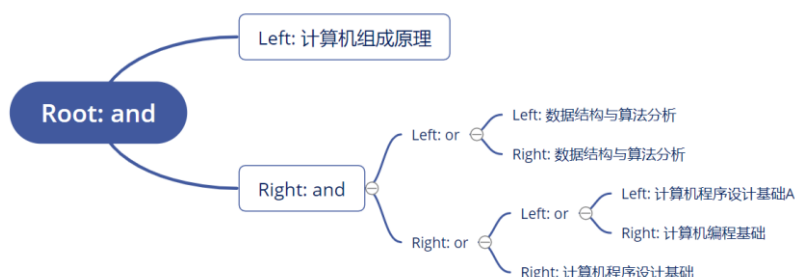
2. 将规范化字符串切割成中缀 Tokens（注意：部分课程名中带有括号，此时需要额外的特判处理）

```
[(, 数据结构与算法分析, 或者, 数据结构与算法设计, ),  
 并且, 计算机组成原理, 并且,  
(, 计算机程序设计基础A, 或者, 计算机程序设计基础,  
 或者, 计算机编程基础, )]
```

3. **利用辅助栈的算法**，扫描切割成的中缀 Tokens，将带有括号的中缀 tokens 表达式转化为不带有括号的后缀 tokens 表达式（注意：双目运算符“并且”的优先等级高于“或者”）

```
[数据结构与算法分析, 数据结构与算法设计, 或者,  
计算机组成原理, 并且, 计算机程序设计基础A,  
计算机程序设计基础, 计算机编程基础, 或者, 或者, 并且]
```

4. 利用辅助栈和树的算法思想，**将后缀 tokens 表达式转化为表达式树**，如下图所示



5. 将创建的树通过 **DFS（深度优先搜索）** 进行赋予所有子节点的 group 值（如下图所示），然后再依次插入数据库即可。（对于非法的数据，如“计算机编程基础”等等，可以参考朱悦铭老师的建议，直接舍弃该数据）



一>**遇到困难 3**：先修课程信息中，有一些课程是 json 信息文件中不存在的课程，为处理先修关系带来了困难；

**解决方法 3**：根据朱悦铭老师的提示，如果出现查询失败的情形，那么可以直接将非法数据进行舍弃。

一>**遇到困难 4**：课程中的上课时间是以数组存储，如果使用数组存储到数据库中，那必然违反数据库设计第一范式（因为数组是可分割的元素）；

**解决方法 4**：考虑到课程信息的数据量较小，可以采用每个课程的每个周次都放置一条记录进行存储，从而将上课周次划分为最小的原子单元。

一>**遇到困难 5**：有可能出现一门课程由多名老师来上，而 teacher 就放了所有老师的信息（包括’，’），也需要额外的处理工作；

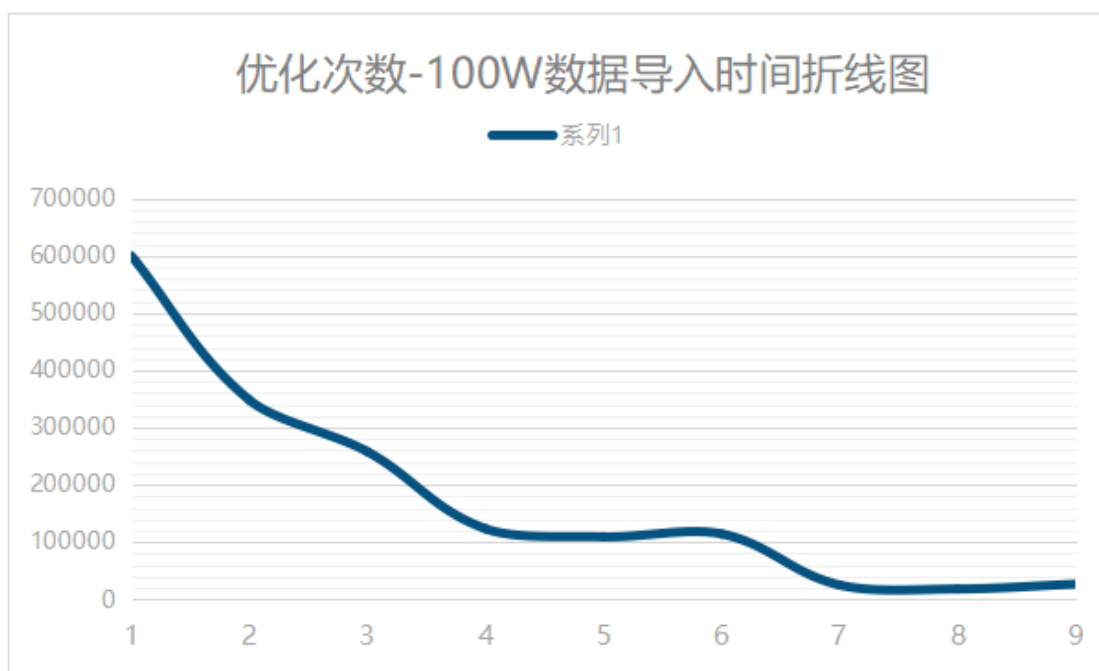
**解决方法 5**：将表示 teacher 的字符串信息进行 split 操作，从而得到分割后的结果。

## 【效率提升】

总体数据导入效率的变化折线图如下图所示（数据规模：100W，纵轴单位：毫秒）

（测试系统：Windows 10）

**前后效率提升：4873%**



用下面一张于老师的表情包进行简要概括：



## 【优化策略分析】

【数据规模】100W

【计时方法】采用 Java 库当中的 `System.currentTimeMillis()` 方法实现程序的计时。

【最初测试】文件全部原始转化为 college 和 student 对象，使用 Statement 进行逐一导入

【测试用时】600000 毫秒左右

### 【优化策略 1】利用 Java 语言的哈希表数据结构，避免创建 college 重复对象

【测试用时】347950 毫秒

【速度提升】+70%

【原因分析】Java 创建重复对象需要花费大量时间和空间。

### 【优化策略 2】将 Statement 更换为 PreparedStatement

【测试用时】289657 毫秒

【速度提升】+17%

【原因分析】相较于 Statement 来说，PreparedStatement 能够得到预编译，减轻数据库的负担。

### 【优化策略 3】将 PreparedStatement 对象声明放置于循环体之外

【测试用时】257742 毫秒

【速度提升】+12%

【原因分析】放置于循环体外可以只需要预编译一次，减少 Java 与数据库的通讯时间。

### 【优化策略 4】合并 insert 语句+多条 PreparedStatement (如 5000 条) 批量执行

【测试用时】123148 毫秒

【速度提升】+110%

【原因分析】批量执行多条 SQL 语句可以进一步减少 Java 与数据库的通讯时间。(如对于 5000 条语句来说，原先要通讯 5000 次，现在只需要 1 次)

### 【优化策略 5】先移除尝试插入表的约束和索引，在完成插入后再将索引和约束进行恢复

【测试用时】108991 毫秒

【速度提升】+12%

【原因分析】在保证插入数据正确的情况下，移除表的索引和约束可以减少表在外键引用表的



约束检查消耗。

#### 【优化策略 6】增加 college 表的索引

【测试用时】114056 毫秒

【速度提升】-4%

【原因分析】优化不成功，源于 college 本身是一个小数据量的表，增加数据库索引并不能显著提升运行速度，反而导致负优化。

#### 【优化策略 7】针对要插入的 collegeid，将子查询操作预先处理完成，移除每一次插入可能需要的子查询操作

【测试用时】24737 毫秒

【速度提升】+340%

【原因分析】原先针对每一次 collegeid 都要访问一次数据库进行子查询操作，大量浪费了时间。移除子查询操作可以显著提升性能。

#### 【优化策略 8】将笔记本电脑从外部连接电源

【测试用时】15917 毫秒

【速度提升】+55%

【原因分析】可移动笔记本对于有无连接电源会采用不同的功耗策略，从而影响 CPU 下的工作频率，进而影响 Java 与数据库程序运行的效率。（有无电源对于可移动笔记本的 Java 与 PostgreSQL 的通讯影响较大，会随着通讯趋于频繁而影响不断增大，例如有子查询时速度提升比率更大，接近 100%）

（策略 8 参考来源：11912824 2020Fall C/C++程序设计期中报告）

#### 【优化策略 9】开启多线程同时插入数据

【测试用时】12066 毫秒

【速度提升】+31%

【原因分析】多线程并发插入数据可以充分调度 CPU 的执行能力，从而提升插入数据的效率

#### 【插入效果】

本次导入数据的过程中，可以通过以下方法证明对于有效合法数据来说，导入数据是 100%成功的。

#### 【课程信息】

【Java 解析】共 597 个课程班级

```
✓ courses = {ArrayList@1683} size = 597
  > 0 = {Course@1872}
  > 1 = {Course@1873}
  > 2 = {Course@1874}
  > 3 = {Course@1875}
  > 4 = {Course@1876}
```

【数据库】共 597 个课程班级信息



select distinct count(*) from classes;	
count :	
1	597

## 【学生信息】

【csv 文件列表】共 399920 行，一行一条学生信息，共 399920 条学生信息。

select\_course.csv - 记事本

文件(E) 编辑(E) 格式(O) 查看(V) 帮助(H)

第 3999920 行, 第 37 列 100% Windows (CRLF) UTF-8

【数据库】共 3999920 条学生信息

63 ✓	select distinct count(*) from student;	
alhost	count :	
ms	1	3999920
78 ms		
e 528 ms		

## 【课程信息】

【csv 文件列表】共 15504 个课程的上课周次信息（不足 16 周的自动补为 16，与数据库的结构一致）

```
int totalWeek=0; totalWeek: 15504
for (Course c:courses){ courses: size = 597
    for (ClassList cl:c.getClassList()){
        totalWeek+=16; totalWeek: 15504
    }
}
```

【数据库】共 15504 个课程的上课周次信息

select count(*) from class_weekList;	
count(*) bigint ×	
count :	
1	15504

## Part 3. Task 3-文件与数据库的比较

### 【任务需求】

1. 将数据存储到数据库的表中，然后使用 SQL 的数据操作语言对数据库做简单分析。记录分析算法的运行时间；
2. 保存数据到一个文件中，然后加载到 RAM 中（数据格式任意）。设计一个算法来做简要的文件分析。在这个任务中，小组可以将数据重构为更快的检索形式。记录分析算法的运行时间。
3. 数据库高级功能探索（以下部分三人小组应该做得更好）
  - a. 高并发与事务管理
  - b. 用户权限管理
  - c. 数据库索引与文件 IO
  - d. 比较不同系统上多个数据库的表现性能

### 【数据内容与结构分析】

#### Part1. json 文件：

-->名称：course\_info.json

-->大小：312Kb

-->特点：**数据量较小，结构较为复杂**

-->内部结构：由多个 course 对象组成，course 及其下属结构如下图所示：

```
▼ courses = {ArrayList@781} size = 597
  ▼ 0 = {Course@862}
    ▼ classList = {ClassList[1]@967}
      ▼ 0 = {ClassList@971}
        > f weekList = {int[15]@972} [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
        > f location = "乒乓球馆"
        > f classTime = "5-6"
        > f weekday = 3
        > f totalCapacity = 28
      > f courseId = "GE232"
      > f prerequisite = null
      > f courseHour = 32
      > f courseCredit = 1.0
      > f courseName = "体育IV"
      > f className = "乒乓球3班"
      > f teacher = "何紫琳"
      > f courseDept = "体育中心"
      > f preSplit = null
    > 1 = {Course@863}
```

#### Part2.csv 文件：

-->名称：select\_course.csv

-->大小：267MB

→特点：**存储信息量大，存储结构简单**，部分 attributes 比如 College\_id 与 json 文件内数据相关性较大

→内部结构：每一行表示一个学生信息，信息结构为：

姓名,性别,书院中文名(书院英文名),学号,修学课程 ID1,修学课程 ID2,...

→存在问题 1：数据量过于庞大，导入数据需要花很长的时间：在 Macbook 上导入 1h+

→思考点 1:

在 Java 中使用 BufferedReader 类对 csv 文件进行一行一行的读取并调用 String 类的 split 方法进行每个字段的分割；

分割后的第三个数据(书院名称)：中文（英文）是不符合第一范式的，所以在这里我们又对 data[2]进行了又一次的分割采用 String[] names = data[2].split("/")进行了二次分割，然后在存入数据库时，names[0] 可以直接存入数据库，但是 names[1]会多英文名字后面的一个“)”” 这里我们没有采用再分割，通过 substring 的方法对该字符串进行截取之后直接添加进入数据库当中。

```
BufferedReader reader = new BufferedReader(new FileReader
    ( fileName: "/Users/lukcyone/Desktop/AlgorithmDesign/DB/select_course.csv"));
```

→思考点 2: 为了避免字符串切割拼接再切割的操作，节省耗时，我们在导入数据时定义了三个类 class，分别是 College、Student、course\_selection（考虑到外键约束按照插入顺序排列），由于没有 json 的解析方法，在读取 csv 文件时，我们直接在这三个定义的类中进行了所需数据的提取与分割。

c Student		
f	name	String
f	gender	String
f	college_id	long
f	college_name	String
f	sid	String
f	college_chinese_name	String

c course_selection_info		
f	courses_id	String[]
f	student_id	String

→思考点 3:从 csv 文件中需要插入的三个表各有特点:

- 第一个表 College: 书院数据在 csv 文本中重复出现，表格中的数据量极少(10 以内)；
- 第二个表 Student: 学生数据在 csv 文本中每一行均有所不同，数据量较大(400W 左右)；
- 第三个表 course\_selection: 同一学生可以选择多门课程，因此数据量极大(1000W 左右);

插入第一个表解析数据为 Java 对象:

【最初做法】

权限为 public 的构造方法，每读取一行即 new 一个 College 对象

```

public College(String line) {
    String[] data;
    data = line.split( regex: ",");
    String name = data[2];
    String[] names;
    names = name.split( regex: "\\(");
    this.college_chinese_name = names[0];
    this.college_english_name = names[1].substring(0,names.length-1);
}

```

【遇到问题】大量对象被重复创建，浪费大量的时间和空间，效率低下，而且容易引发内存溢出异常使得程序直接挂掉，如下图所示

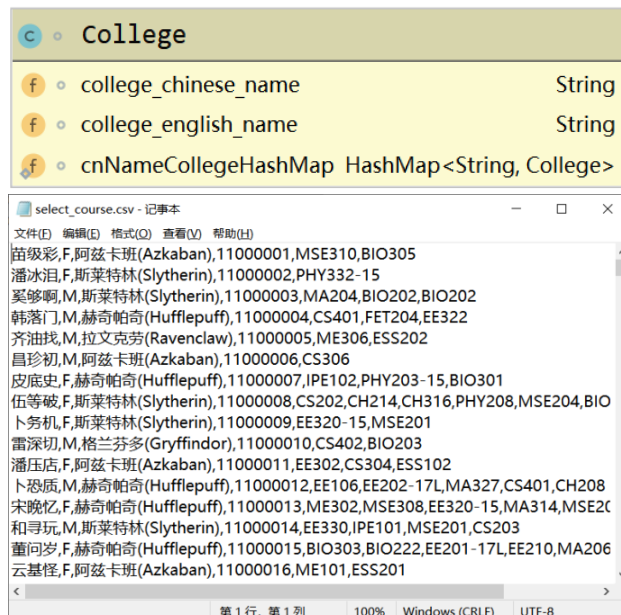
Exception in thread "main" java.lang.OutOfMemoryError Create breakpoint : Java heap space at Main.main(Main.java:6)

Process finished with exit code 1

【改进方法】参考类设计模式的"懒汉思想"和哈希思想，现采取的策略如下：

1. 控制构造方法的访问权限为 private;
2. 构造存储字符串和实例对象的哈希表：
3. 如果对应的对象已经创建，那么就不再创造对象；否则就创建对象并加入哈希表。

如此，对于相同的字符串表示的对象只会存在一次，大量节约时间和空间，避免了 OutOfMemoryError 异常。



```

static HashMap<String, College> cnNameCollegeHashMap = new HashMap<>();
private College(String cn,String en){
    this.college_chinese_name=cn;
    this.college_english_name=en;
}

public static College newInstance(String line){
    String[] data;
    data = line.split( regex: ",");
    String name = data[2];
    String[] names;
    names = name.split( regex: "\\(");
    if(!cnNameCollegeHashMap.containsKey(names[0])){
        College c=new College(names[0],names[1]);
        cnNameCollegeHashMap.put(names[0],c);
        return c;
    }else {
        return cnNameCollegeHashMap.get(names[0]);
    }
}

```

第二个表 Student,数据量太大，导入时间较慢，后来在优化过程中采用例如多线程的方式提高了插入效率（详情可见于 Part 2 的优化策略）。

## 【测试环境】

【操作系统】Windows 10，Mac Book

【编程语言】Java，PostgreSQL

## 【实验设计】

### 【多线程高并发】

本次 project 中利用多线程进行了高并发量的测试。

测试表：考虑 student 表，要插入的数据规模在 100W 左右（999980 条）

考虑将插入的 100W 数据分配给 8000 个线程同时执行插入操作。

```

v threads2 = {Thread[8000]@9549}
> 0 = {Thread@1540} "Thread[Thread-1,5]"
> 1 = {Thread@1546} "Thread[Thread-2,5]"
> 2 = {Thread@1547} "Thread[Thread-3,5]"
> 3 = {Thread@1548} "Thread[Thread-4,5]"

```

但最终全部执行完成后数据可内只有 999176 条，出现了数据的丢失情况。

[2021-04-10 09:33:15] 999,176 rows affected in 806 ms

【原因分析】存储数据的容器是 ArrayList，是线程不安全的。在多线程访问时极有可能出现访问冲突问题。另外，这 8000 个线程共用同一个 Connection 对象，在 commit 的时候极易引起线程冲突导致数据插入失败。

```

private static ArrayList<String>allInput=new ArrayList<>();

```

【解决方法 1】将存储的容器修改为线程安全的 Vector 或者 CopyOnWriteArrayList，避免线程冲突问题。

```
private static CopyOnWriteArrayList<String> allInput=new CopyOnWriteArrayList<>();
```

如此，再次测试插入时，就可以完成 999980 条的数据的全部插入。

[2021-04-10 09:52:22] 999,980 rows affected in 816 ms

【解决方法 2】增加 commit 方法的线程锁，保证任何时间至多有 1 个线程执行 Connection 对象的 commit 方法，避免多个线程竞争 Connection 对象的情况。

```
private static synchronized void myCommit(){
    try{
        con.commit();
    }catch (SQLException e){
        e.printStackTrace();
    }
}
```

### 【事务管理】

本次 project 中针对事务管理中的问题进行了测试，并给出了合适的解决方案。

【参考链接】<https://blog.csdn.net/cuichunchi/article/details/89535677>

### 【不可重复读】

本次 project 中进行了以下操作，出现了不可重复读现象。

1. 对于本地库，同时开启两个终端终端 Task3 和 Task3\_1;

```
Task3 [postgres@localhost]
Task3_1 [postgres@localhost]
```

2. 按照以下表格进行两个终端的操作

步骤	终端 Task3	终端 Task3_1
1	begin;	begin transaction;
2	select * from college where college_chinese_name='阿兹卡班';	
3		update college set college_chinese_name='阿兹卡班 2' where college_id=10;
4		commit transaction ;

5	<code>select * from college where college_chinese_name='阿兹卡班';</code>	
6	<code>commit;</code>	

在 Step2 中，可以读取 ‘阿兹卡班’ 的对应信息，如下：

	college_id	college_chinese_name	college_english_name
1	10	阿兹卡班	Azkaban

而在 Step5 中，就不再能够查询到 ‘阿兹卡班’ 的信息，如下：

	college_id	college_chinese_name	college_english_name
0 rows			

如此我们可以看到出现了不可重复读现象。

### 【幻读】

本次 project 中进行了以下操作，导致了幻读现象的出现。

- 对于本地库，同时开启两个终端 Task3 和 Task3\_1;

Task3 [postgres@localhost]

Task3\_1 [postgres@localhost]

- 按照以下表格进行两个终端的操作

步骤	终端 Task3	终端 Task3_1
1	<code>set default_transaction_isolation = 'read committed';</code>	<code>set default_transaction_isolation = 'read committed';</code>
2	<code>begin;</code>	<code>begin;</code>
3	<code>insert into college (college_chinese_name, college_english_name) values ('致新书院', 'ZhiXinCollege');</code>	
4		<code>select * from college;</code>
5	<code>rollback;</code>	
6		<code>select * from college;</code>



7		commit;
---	--	---------

在 step4 中，我们在终端 3\_1 可以看到回滚前的数据，如下所示：

	college_id	college_chinese_name	college_english_name
1	2	斯莱特林	Slytherin
2	3	赫奇帕奇	Hufflepuff
3	4	阿兹卡班	Azkaban
4	6	格兰芬多	Gryffindor
5	15	拉文克劳	Ravenclaw
6	2007224	致新书院	ZhiXin College

在 Step5 时，终端 3 执行了 rollback 操作，使得加入的数据无效；因此在终端 3\_1 中的第二次查询就查询不到信息，也就是出现了幻读的现象。

### 【解决方案 1】设置隔离级别为最高的隔离级别 serializable（不推荐）

存在问题：虽然避免了所有的错误情形，但在高并发量的时候容易读写阻塞，有较大机会出现死锁问题，效率较低。

### 【解决方案 2】修改表为多版本并发控制（推荐）

操作方法：

1. 每条数据都增加隐藏的两列（创建和删除），使得每个事务在开始时都会有一个递增的版本号
2. 增删改查的方法如下：

【增加】直接按照当前数据列插入即可

【删除】将数据的版本号更新为当前事务的版本号

【修改】数据的修改采用“除旧迎新”的方法。

1. 对需要更新的数据执行上面所述的“删除操作”
2. 使用更新的版本号新增一条更新好的数据

【查询】为了避免查询到“旧数据”，查询返回的结果必须满足以下条件：

1. 当前事务版本号 >= 创建版本号（保证结果已经正确创建）
2. 当前事务版本号 <= 删除版本号或者删除版本号 is null（保证结果是最新的版本）

### 【数据库用户权限管理】

#### 【默认声明】

1.create user 和 create role 除了 user 默认有登陆权限之外 user 和 role 没有其他差别

也就是说在这种情况下 user B 和 role A 是完全等价的,所以在以下的实验操作中，我们对 user 和 role 没有进行特别的区分，特此说明

```
create role A password 'xxxxxx' login
create user B password 'xxxxxx'
```

2.为了保证数据库内数据的安全性和可恢复性，以下操作都会使用 **begin**，**rollback** 来保障数据不丢失。

### 【权限】

数据库系统中的权限包括 **select, insert, update, delete, rule, references, trigger, create, temporary, execute, usage**，我们使用 **grant** 命令给普通用户开启权限，使用 **revoke** 撤销给普通用户赋予的权限。具体语句如下所示：

**注意：**要在 Data Source Properties 里面将 User 配置为 name\_login,否则单纯更改 owner 为 name\_login 没有仍然可以以 superuser 的身份访问

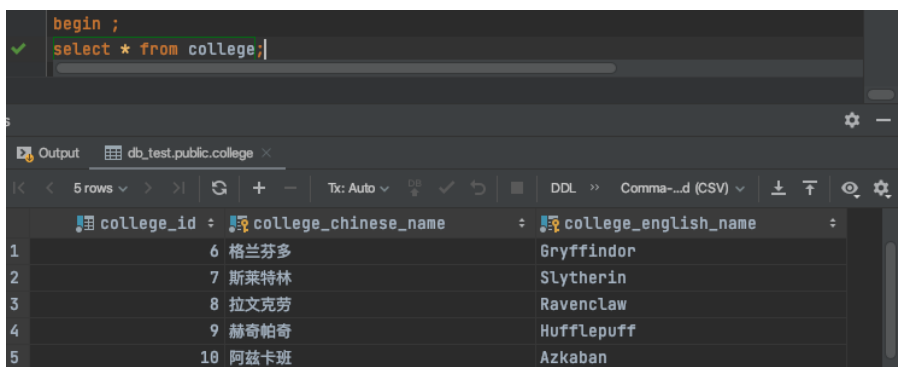
User:

### 【创建普通用户&&为普通用户开权限】

```
-- 创建一个普通用户
create user name_login password 'xxxxxx';
-- 将 college 的 owner 设置为用户 name_login
alter table college owner to name_login;
-- 给 name_login 赋予在 college 表上的 update 权限
grant update on college to name_login;
-- 给 name_login 撤销在 college 表上的 update 权限
revoke update on college from name_login;
```

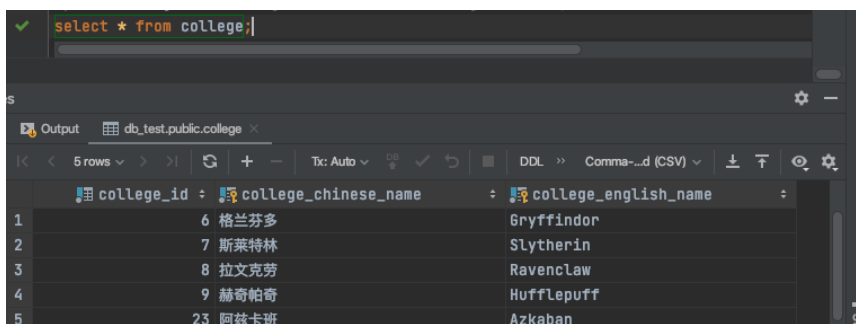
以下是对应实验截图

如图所示，最初 college 表



college_id	college_chinese_name	college_english_name
6	格兰芬多	Gryffindor
7	斯莱特林	Slytherin
8	拉文克劳	Ravenclaw
9	赫奇帕奇	Hufflepuff
10	阿兹卡班	Azkaban

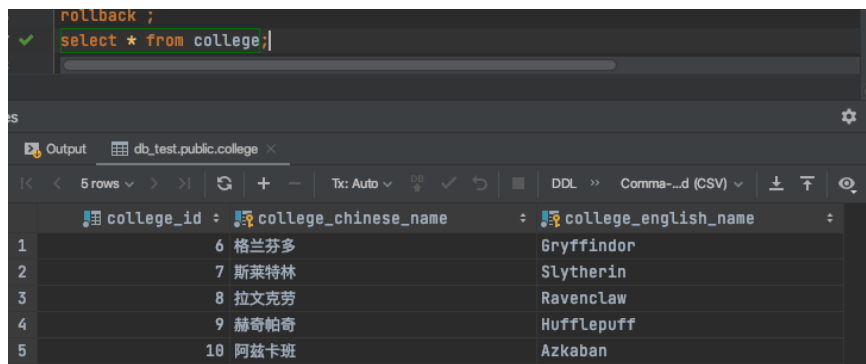
如图所示这是进行完 update 后的 college 表



college_id	college_chinese_name	college_english_name
6	格兰芬多	Gryffindor
7	斯莱特林	Slytherin
8	拉文克劳	Ravenclaw
9	赫奇帕奇	Hufflepuff
23	阿兹卡班	Azkaban

## 恢复数据

```
rollback ;
select * from college;
```



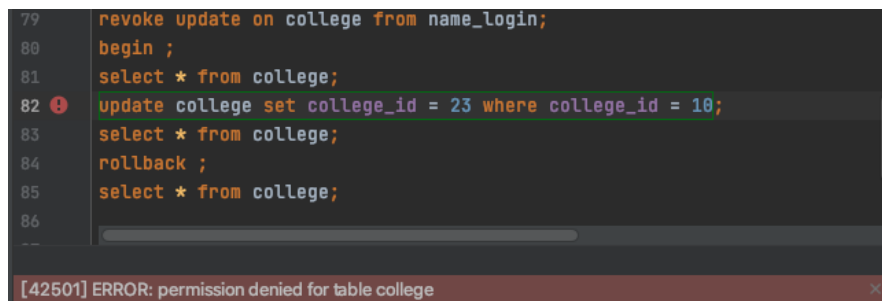
college_id	college_chinese_name	college_english_name
6	格兰芬多	Gryffindor
7	斯莱特林	Slytherin
8	拉文克劳	Ravenclaw
9	赫奇帕奇	Hufflepuff
10	阿兹卡班	Azkaban

### 【撤销普通用户的权限】

当我们撤销了用户 name\_login 的更新权限后再使用可以看到它会报错：

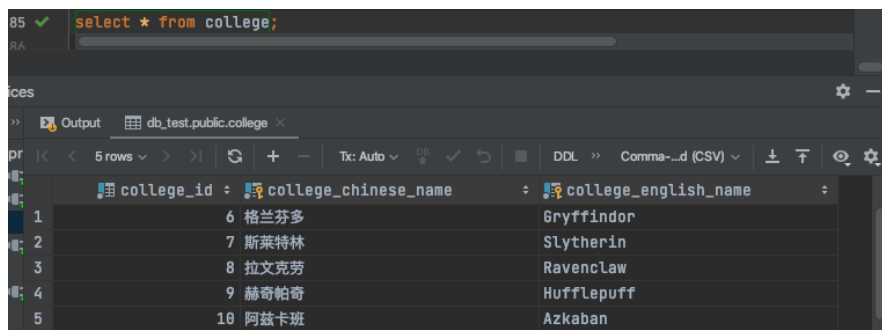
permission denied for table college，即该用户没有进行对 college 表进行更新操作的权限

```
79 revoke update on college from name_login;
80 begin ;
81 select * from college;
82 ❸ update college set college_id = 23 where college_id = 10;
83 select * from college;
84 rollback ;
85 select * from college;
86 --
```



但是不管是普通用户还是超级用户都有对他所管理（own）的表有查看属性

```
85 select * from college;
```



college_id	college_chinese_name	college_english_name
6	格兰芬多	Gryffindor
7	斯莱特林	Slytherin
8	拉文克劳	Ravenclaw
9	赫奇帕奇	Hufflepuff
10	阿兹卡班	Azkaban

但如果该表的 owner 不是 name\_login，比如我们拿 student 为例，可以看到该用户也是没有权限查看的，因为 student 的 owner 目前是超级用户 checker 而不是普通用户 name\_login

```
alter table student
owner to checker;
```

```
71 select * from student;
72
```



此外普通用户也没有创建角色/用户的权限

```
90 create role test;
```



以下以 superuser : checker 为例对用户管理数据库的操作进行分析

先查看 user checker 的属性，可以看到 rolsuper 为 true，因此为 superuser。

```
131 ✓ select * from pg_roles where rolname = 'checker';
```

rolname	rolsuper	rolinherit	rolcreatorole	rolcreatedb	rolcanlogin	rolreplication	rolconnlimit	rolpassword	rolvalidunt
1 checker	true	true	false	false	true	false	-1	*****	<null>

### 【创建用户/角色】

对于数据库说，只有超级用户 superuser 或者被赋予了创建用户/角色属性的普通用户/角色才可以创建用户/角色

```
90 ✓ create role test;
```

### 【数据添加】

Q: <Filter Criteria>				
	sid	name	gender	college_id
1	11519087	谈藏整	F	10
2	11519088	于乡党	M	6
3	11519089	韦散祖	M	10
4	11519090	韩而若	M	8
5	11519091	蒋爸解	F	6
6	11519092	贺况险	M	6
7	11519093	邬子的	M	9
8	11519094	蒋文恶	M	9
9	11519095	赵登黄	M	6
10	11519096	岑单难	F	9
11	11519097	汤婿医	F	8
12	11519098	李读族	F	7
13	11519099	葛太本	F	9
14	11519100	钱各起	F	8
15	11519101	滕皮担	F	7
16	11519102	宋至消	M	9
17	11519103	鄢德了	M	7

```
private String host = "localhost";
private String dbname = "db_test";
private String user = "checker";
private String pwd = "5432";
private String port = "5432";
```

### 【查询数据】

使用 superuser 的身份可以进行数据库的查询

124 ✓ `select * from college;`

Services

Output db\_test.public.college x

college_id	college_chinese_name	college_english_name
1	2 斯莱特林	Slytherin
2	3 赫奇帕奇	Hufflepuff
3	4 阿兹卡班	Azkaban
4	6 格兰芬多	Gryffindor
5	15 拉文克劳	Ravenclaw
6	6174551 SHUDE College	树德书院

User: checker

### 【删除数据】

使用 superuser 可以对数据库中的数据进行删除

如图所示为删除后的查询结果，可以看到 college\_english\_name 为树德书院的记录被删除

✓ `select * from college;`  
`delete from college where college_english_name = '树德书院';`

Output db\_test.public.college x

college_id	college_chinese_name	college_english_name
1	2 斯莱特林	Slytherin
2	3 赫奇帕奇	Hufflepuff
3	4 阿兹卡班	Azkaban
4	6 格兰芬多	Gryffindor
5	15 拉文克劳	Ravenclaw

### 【更改数据】

superuser 有更改数据库的权限，比方说 drop 掉 College 的一个 attributes 列

为了避免对数据库的破坏，采用 transaction 的方式更改完数据之后，再把数据 rollback 恢复到更改前的数据，如第二幅图所示，rollback 之后再次查询该表，可看到表与删除之前一致

126 `begin;`  
127 `alter table college drop column college_english_name;`  
128 `rollback ;`

Services

Output db\_test.public.college x

college_id	college_chinese_name
1	2 斯莱特林
2	3 赫奇帕奇
3	4 阿兹卡班
4	6 格兰芬多
5	15 拉文克劳

```
rollback ;
select * from college;
```

	college_id	college_chinese_name	college_english_name
1	2	斯莱特林	Slytherin
2	3	赫奇帕奇	Hufflepuff
3	4	阿兹卡班	Azkaban
4	6	格兰芬多	Gryffindor
5	15	拉文克劳	Ravenclaw

## 【创建用户属性】

我们也可以在用超级用户创建角色/用户时为该用户/角色赋予某些特定的属性比如：

createdb(创建数据库), createrole (创建角色), replication(同步数据库)

-- 创建一个可以创建数据库的角色

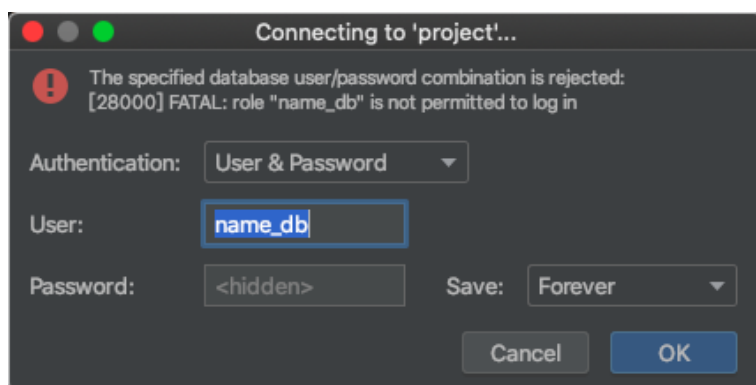
```
create role name_db createdb password 'xxxxxx';
```

-- 创建一个可以创建角色的角色

```
create role name_role createrole password 'xxxxxx';
```

实验结果如下图所示：

注意：因为我们在用超级用户创建这一角色的时候并没有赋予它登陆的权限，角色不像用户默认有登陆权限，所以我们需要给该角色添加一个登陆权限才能进行使用



-- 创建一个可以创建数据库的角色

```
create role name_db createdb login password 'xxxxxx';
```

-- 创建一个可以创建角色的角色

```
create role name_role login createrole password 'xxxxxx';
```

如图所示，当允许登入时，User 设置为 name\_db 时可以创建数据库 test

```
db_test.public> create database test
[2021-04-10 13:43:56] completed in 549 ms
```

但是因为只给 name\_db 用户开了可以创建数据库的权限，因此没有办法创建角色

```
93 create role test;
```

test

[42501] ERROR: permission denied to create role

同理对于用户 name\_role，在使用 superuser 创建这一角色时赋予了这一角色 createrole 的

角色，因此预期的实验结果应为：在这一用户下，创建用户时可以进行的但是创建数据库时会失败。

```
db_test.public> create role test_user
[2021-04-10 14:08:49] completed in 10 ms

57 create database test_user;
58
test_user
[42501] ERROR: permission denied to create database
```

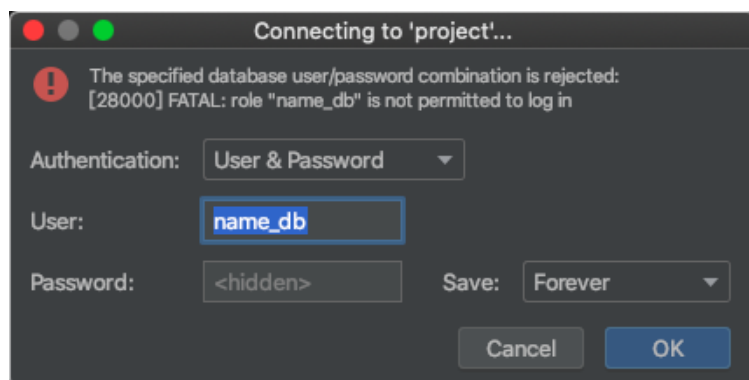
### 【修改用户权限】

在数据库中，可以通过 alter 对用户/角色进行权限的修改

如图所示为锁定用户 name\_db 不允许其登陆的权限修改

```
-- 修改用户权限
alter role name_db with nologin ;
```

如图所示在改变其可登入权限之后，该用户无法登陆数据库



### 【设置访问权限】

如图所示，我们在 superuser checker 下创建了一个用户 test1，然后将 update 这一权限赋予该用户，然后在该用户下，当我们使用 select 时，会失败，这是合理的以为我们给普通用户只开了更新数据的权限

```
109 -- 设置访问权限
110 create user test1;
111 grant update on college to test1;
112 select * from college;
```

如果我们想让该用户可以查看当前数据库下所有的表，那么可以通过这条命令  
grand select on all tables in schema public to test1

**实验结果如下：此时当用户为 test1 时，可以对所有的表进行访问**



```

grant select on all tables in schema public to test1;
✓ select * from college;
✓ select * from student;
✓ select * from course_selection;
✓ select * from classes;
✓ select * from teacher;
✓ select * from course;
✓ select * from location;
✓ select * from department;
✓ select * from prerequisite;

```

## 【用户组】

在实际问题中，一般我们是给一个组赋予一个权限，然后将可以有该权限的人分到这个组里，用户组它本身也是一个 role，包含其他 role 的 role 就是一个用户组

```

135 -- 用户组
136 ✓ create role temporary_users;
137 ✓ grant temporary_users to test1;
138 ✓ grant temporary_users to name_login;
139 ✓ alter role test1 inherit;

```

其中 temporary\_users 是一个用户组

语句 137&138 表示赋予 test1 和 name\_login 关于 temporary\_users 的 membership。

语句 139 表示让 test1 继承用户组 temporary\_users 的所有权限

## 【修改用户名】

```

success 136 ✓ alter user test1 rename to success;

```

## 【设置用户连接数】

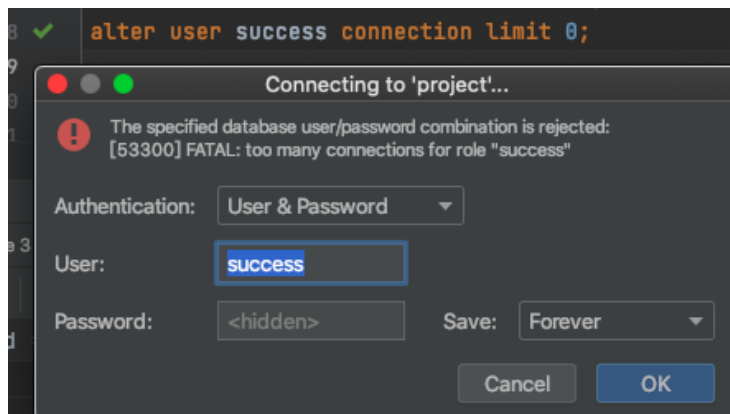
作用是在多个用户同时访问时，只能同时能有最大连接数的用户同时访问数据库，如果不设置最大连接数的话，那么数据库在被访问时很有可能挂机，之所以可以用设置最大连接数的方法来保证数据库的稳定访问，是因为数据库连接是可以并发访问的，这一最大并发连接数受服务器配置，网络环境约束等因素的影响，具体来说则包括服务器的 CPU 以及内存配置网络的带宽，因此对于一个数据库设置其最大连接数还是要根据实际情况进行调整。

```

137 ✓ alter user success connection limit 10;

```

其中如果 limit 设置为 0，则表示不允许登陆，如图所示：



如果 limit 设置为-1 的话，则表示对用户数没有限制

```
✓ alter user success connection limit -1;
```

### 【数据库索引优劣势】

本次 Project 尝试了数据库索引对于数据搜索与更新的效果的实验。

#### 【数据搜索】

测试表：考虑 course\_selection 表，数据规模在 1000W 左右；

1. 尝试直接搜索 course\_selection 表中学生 id 为 11000028 的所有选课信息，用时 1505ms：

```
project1db.public> select * from course_selection where student_id='11000028'
[2021-04-09 22:34:37] 3 rows retrieved starting from 1 in 1 s 505 ms (execution: 1 s 474 ms, fetching: 31 ms)
project1db.public> create index sid_index on course_selection(student_id)
```

2. 添加对于学生 id 的索引，用时 44045ms：

```
project1db.public> create index sid_index on course_selection(student_id)
[2021-04-09 22:33:45] completed in 44 s 45 ms
```

3. 再次搜索 course\_selection 表中学生 id 为 11000028 的所有选课信息，用时 31ms，相较于 1 中的搜索效率提升了 **4754%**！

```
project1db.public> select * from course_selection where student_id='11000028'
[2021-04-09 22:43:13] 3 rows retrieved starting from 1 in 31 ms (execution: 0 ms, fetching: 31 ms)
```

#### 【数据更新】

测试表：考虑 student 表，数据规模在 100W 左右；

1. 尝试在一个能够 rollback 的事务中直接将所有的 SID 都修改为 99999999，用时 2996ms

```
[2021-04-09 23:24:14] Unsafe query: 'Update' statement without 'where' updates all table rows at once
Project1Test.public> update student set SID='99999999'
[2021-04-09 23:24:19] 999,968 rows affected in 2 s 996 ms
```

2. 添加对于学生 SID 的索引，用时 2019ms

```
Project1Test.public> create index sindex on student(SID)
[2021-04-09 23:24:43] completed in 2 s 19 ms
```

3. 再次尝试在一个能够 rollback 的事务中将所有 SID 都修改为 99999999，用时 4350ms，**效率反而发生了下降**

```
2021-04-09 23:24:47] Unsafe query: 'Update' statement without 'where' updates all table rows at once
'project1Test.public> update student set SID='99999999'
2021-04-09 23:24:52] 999,968 rows affected in 4 s 350 ms
```

#### 【数据删除】

测试表：考虑 student 表，数据规模在 100W 左右；

1. 尝试在一个能够 rollback 的事务中直接将所有的姓名大于等于'王'的学生全部删除，用时 524ms

```
Project1Test.public> begin transaction
[2021-04-09 23:33:54] completed in 0 ms
Project1Test.public> delete from student where name >='王'
[2021-04-09 23:33:54] 289,605 rows affected in 524 ms
Project1Test.public> rollback transaction
[2021-04-09 23:33:54] completed in 0 ms
```

2. 添加对于学生 SID 的索引，用时 1859ms

```
Project1Test.public> create index sindex on student(SID)
[2021-04-09 23:33:12] completed in 1 s 859 ms
```

3. 再次尝试在一个能够 rollback 的事务中将所有的姓名大于等于'王'的学生全部删除，用时 625ms

```
Project1Test.public> begin transaction
[2021-04-09 23:33:43] completed in 0 ms
Project1Test.public> delete from student where name >='王'
[2021-04-09 23:33:44] 289,605 rows affected in 625 ms
Project1Test.public> rollback transaction
[2021-04-09 23:33:44] completed in 0 ms
```

从以上两个实验来看，我们可以发现虽然建立索引耗费了比较长的时间，但是在建立索引之后，数据库查询的 execution time 几乎为 0，大大缩短了检索时间；但是对于数据库元素进行删除或者修改的时间则会有所延长。

### 【实验结论】

PostgreSQL 数据库使用索引的优势：

（注：优劣势会随着表的增大而逐步凸显出来）

1. 查询语句中 where 使用索引列可以大大加速查询速度；
2. 索引可以加强约束条件；
3. 索引可以加速表的连接功能；

劣势：

1. 索引会消耗额外的内存空间；
2. 索引表的修改的代价比较高，当对表进行插入或者删除操作时，所有的索引都需要发生变化；

### 【数据库与编程语言的效能测试】

（共做了 Java 和 C++ 两种语言，本报告中以 Java 语言为例）

测试表 1：course\_selection\_info 学生选课信息表，数据量在 1400W 左右

测试表 2：student 学生信息表，数据量在 400W 左右

### 【测试 1】精准查询测试

通过以下语句尝试搜索学生 11000028 的所有课程信息：

```
79 ✓ select * from course_selection where student_id='11000028';  
  
for (Course_selection_info course_selection_info : arrayList) {  
    if (course_selection_info.getStudent_id().equals("11000028")) {  
        System.out.println(course_selection_info);  
    }  
}
```

【PostgreSQL 运行时间】 31ms

【Java 运行时间】 188ms

### 【测试 2】修改测试

通过以下语句将所有学生 ID 改为大写

```
update course_selection set student_id=upper(student_id);  
  
for (Course_selection_info info:arrayList){  
    info.toUpper();  
}
```

【PostgreSQL 运行时间】 401ms

【Java 运行时间】 47676ms

### 【测试 3】排序测试

通过以下语句对学生选课的内容进行排序

```
79 ✓ select * from course_selection order by student_id ;  
  
arrayList.sort(Comparator.comparing(Course_selection_info::getStudent_id));
```

【PostgreSQL 运行时间】 40ms

【Java 运行时间】 200ms

### 【测试 4】多表合并测试

通过以下语句对两个表的内容进行合并再输出

```
79 ✓ select * from course_selection  
80 inner join student s on course_selection.student_id = s.sid;  
for (Course_selection_info info:arrayList){  
    int index=Collections.binarySearch(students, new Student(info.getStudent_id(), b: false),  
        Comparator.comparing(o -> o.sid));  
    System.out.println(info+" "+students.get(index));  
    /*for (Student s:students){  
        if(s.sid.equals(info.getStudent_id())){  
            //System.out.println(info+" "+s);  
        }  
    }*/  
}
```

【PostgreSQL 运行时间】 1140ms

【Java 运行时间】2311ms ( 针对有序数据的二分查找时间，如果数据无序则时间会成倍增长 )

### 【测试 5】删除测试

尝试删除表中学生 11000028 的所有课程信息：

```
delete from course_selection where student_id='11000028';
```

```
for (int i = 0; i < arrayList.size(); i++) {  
    if(arrayList.get(i).getStudent_id().equals("11000028")){  
        arrayList.remove(i);  
        i--;  
    }  
}
```

【PostgreSQL 运行时间】206ms

【Java 运行时间】1450ms

### 【测试 6】插入测试

尝试向表中插入 1000 条数据

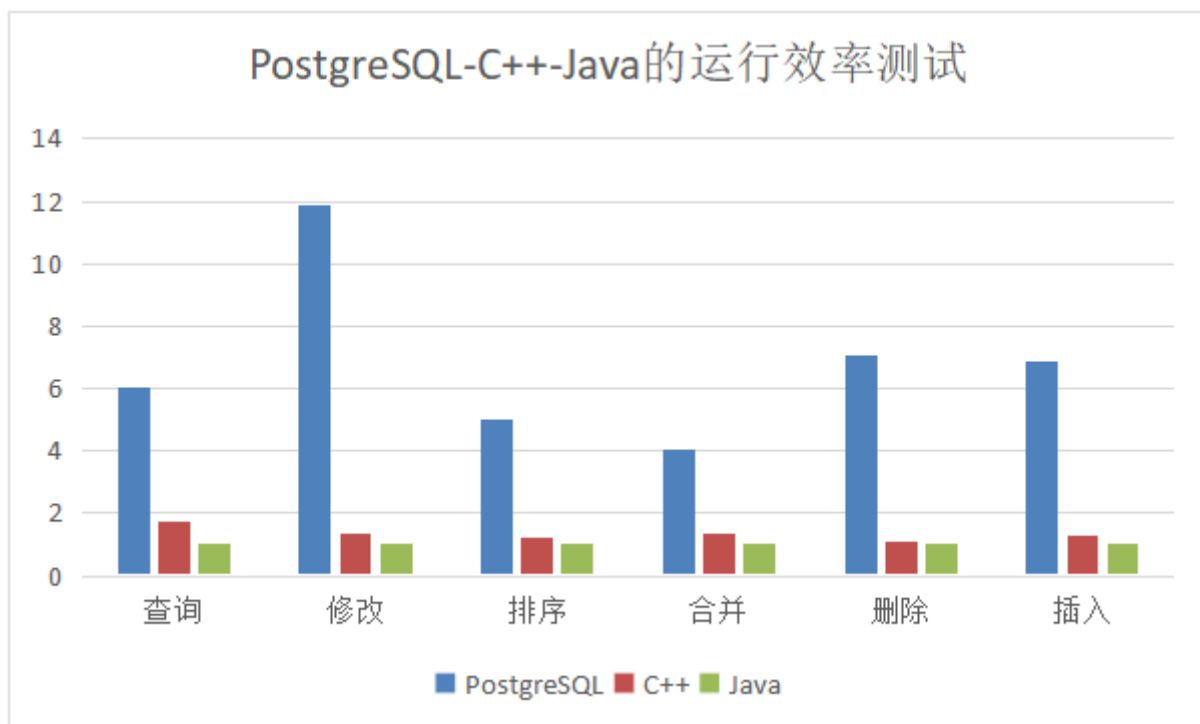
```
84 ✓ insert into course_selection(course_id, student_id)  
85 values ('CS102A', '11910104');  
86 /*...*/  
  
for (int i = 0; i < 1000; i++) {  
    arrayList.add(new Course_selection_info( student_id: "11910104", course_id: "CS102A"));  
}
```

【PostgreSQL 运行时间】10ms

【Java 运行时间】684ms

### 【实验结论】

通过以上测试，我们可以得出结论：PostgreSQL 字各种方面的性能都超过基础编程语言。



## 【数据库系统在不同操作系统的效能测试】

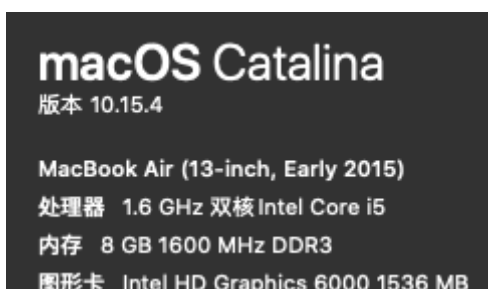
### 【设备信息】

#### 【设备 1】

XiaoXinAir-14iL 2020

设备名称	LAPTOP-Q2EITDTR
处理器	Intel(R) Core(TM) i5-1035G1 CPU @ 1.00GHz 1.19 GHz
机带 RAM	16.0 GB (15.8 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

#### 【设备 2】



### 【实验测试】

考虑插入 100W 条数据的运行时间：

	Windows	Mac
1	347950	2974721
2	114056	3958973
3	24737	53519
4	17917	21056
5	12066	16912

其中 1-5 依次对于插入数据的流程进行了 Task2 中的优化流程，两台设备运行的代码完全一致。

其中 Mac 设备运行效率低于 Windows 系统，这是源于两台设备的**可用内存有所差异** (Windows 设备的可用内存较大，Mac 设备的可用内存较小，对于数据运行的效率带来了较大的影响)

## Part 4.总结

本次 Project 由 2 人组队完成。在这个 project 的制作当中，团队成员都学习了许多的知识与技能：

1. 对于数据库表设计的三大范式的理解；
2. Java 与 PostgreSQL 数据库连接的基本操作；
3. Java 对于不同数据格式文件的读取的方法；
4. Java 哈希数据结构的理解和应用；
5. 提升 PostgreSQL 插入效率的方法；
6. 验证文本文件数据全部插入数据库的方法；
7. PostgreSQL 与 Java 编程语言的效率差异；
8. 数据库高并发访问的问题解决策略；
9. 数据库的事务管理，以及多客户端访问时出现的错误读取现象；
10. 数据库访问的用户权限管理；
11. 数据库索引对于数据增删改查的效率的影响；
12. 数据库在不同操作系统下的性能差异