

# 数据库 Project Part1报告

11911839 聂雨荷

11910931 段心童

## 第一部分：小组信息以及贡献

姓名	贡献	占比
聂雨荷	数据库设计 数据整理与导入 数据库索引 文件I/O 用户权限设计以及用户需求设计 Redis设计	50%
段心童	先修课表数据处理 数据导入数据库以及性能测试 不同数据库之间对比 文件系统与数据库系统对比 数据库性能测试 数据库高并发压力测试 jdbc查询ORM结构	50%

## 机器配置参考

- **Macbook Pro 2016 (Processor: 2.9 GHz Dual-Core Intel Core i5)    Python 3.7.3    maven**
- **Windows10 Intel(R) Core(TM) i7-7500U CPU    Python 3.8.3    conda 4.9.2**

## 编程语言

- **Java**
- **Python**

## 第二部分：数据库设计

### 1. 原始数据

#### 1.1 select\_course.csv

select\_course.csv 数据如下：

- name: 姓名
- gender: 性别
- college: 书院名称（中英文）
- number: 学号
- course 1-6: 选课课程id

#### 1.2 course\_info.json

course\_info.json 数据如下：

- totalCapacity: 班级容量
- courseid: 课程id
- prerequisite: 先修课
- courseHour: 课时
- courseCredit: 学分
- courseName: 课程名称
- className: 班级名称
- courseDept: 开课院系
- teacher: 任课老师
- weeklist 1-3: 上课时间（第几周）
- location 1-3: 上课地点
- classtime 1-3: 上课时间（第几节课）
- weekday 1-3: 上课时间（星期几）

### 2. 表设计

## 2.1 原始分析

我们从原始数据中获得的关键信息如下：

- 每个学生会选择 1-6 门课程
- 每个课程会分配不同的班级
- 每个班级的上课具体分配（时间、地点）会有 1-3 种分配情况
- 先修课存在 (课程1 OR 课程2 OR 课程3) AND (课程4 OR 课程5) 的与或关系情况
- 体育课 GE232 与其它课程存在差异，一门体育课课程下存在不同类型性质的课程（篮球/足球/乒乓球等），每个课程下还会有班级，每个班级有自己分配
- 每个班级的任课老师一个及以上
- 教学周的种类非常多，尽管只存在 1-16 周，但是种类非常多

这些信息导致表不满足三大范式，为了让数据库设计满足三大范式，且可扩展性强，我们设计了以下表。

## 2.2 表结构介绍

### college 书院表

- **id**: 书院标识符，无实意义（自增主键）
- **chinese\_name**: 书院中文名
- **english\_name**: 书院英文名

UNIQUE(chinese\_name, english\_name)

### department 院系表

- **id**: 院系标识符，无实意义（自增主键）
- **name**: 院系名称

UNIQUE/NOTNULL(name)

### week 教学周表

- **id**: 教学周标识符，无实意义（自增主键）
- **w1-w16**: 第一周至第十六周（共16列）
- **comment**: 备注

UNIQUE/NOTNULL(w1-w16)

## location 地点表

- **id**: 地点标识符, 无实意义 (自增主键)
- **area**: 划分片区
- **building**: 教学楼
- **room**: 房间号
- **function**: 功能
- **comment**: 备注

UNIQUE/NOTNULL(area, building, room)

## course 课程表

- **id**: 课程标识符, 无实意义 (自增主键)
- **department\_id**: 开课院系id (外键, 指向 **department** 表)
- **course\_id**: 课程编号
- **name**: 课程名称
- **hour**: 课时
- **credit**: 课程学分

UNIQUE/NOTNULL(course\_id)

## class 班级表

- **id**: 班级标识符, 无实意义 (自增主键)
- **course\_id**: 课程编号 (外键, 指向 **course** 表)
- **class\_number**: 第几班 (若为0, 则代表本课程不分班级)
- **language**: 班级使用语言 (中文/英文/中英双语)
- **teaching\_object**: 授课对象 (所有学生/仅限留学生)
- **property**: 班级性质 (正课/习题课/实验课/体育课)
- **makeup**: 是否提供补课
- **capacity**: 班级容量
- **class\_name**: 班级名称
- **comment**: 备注

UNIQUE/NOTNULL(class\_number, course\_id, language)

## **sport** 体育课表

- **id**: 体育课标识符, 无实意义 (自增主键)
- **class\_number**: 第几班 (若为0, 则代表本课程不分班级)
- **course\_id**: 课程编号 (外键, 指向 **course** 表)
- **class\_type**: 课程性质 (篮球/足球/羽毛球...)
- **capacity**: 课程容量
- **comment**: 备注

UNIQUE/NOTNULL(class\_type, class\_numbr)

## **distribution** 分配表

- **id**: 分配标识符, 无实意义 (自增主键)
- **class\_id**: 班级编号 (外键, 指向 **class** 表, 体育课置为0)
- **sport\_id**: 体育课编号 (外键, 指向 **sport** 表, 非体育课的分配为空)
- **week\_id**: 教学周编号 (外键, 指向 **week** 表)
- **location\_id**: 地点编号 (外键, 指向 **location** 表)
- **clsss\_time**: 上课时间 (第几节课)
- **week\_day**: 上课时间 (星期几)

UNIQUE/NOTNULL(class\_id, week\_id, location\_id, class\_time, week\_day)

## **prerequisite** 先修课表

- **id**: 先修课标识符, 无实意义 (自增主键)
- **course\_id\_b**: 先修课程编号 (外键, 指向 **course** 表)
- **course\_id\_a**: 后修课程编号 (外键, 指向 **course** 表)
- **group\_id**: 组编号 (同时属于一组的先修课程可以只修其中一门)

UNIQUE/NOTNULL(course\_id\_a, course\_id\_b, group\_id)

## **student** 学生表

- **school\_number**: 学号 (主键)
- **name**: 姓名
- **gender**: 性别
- **college\_id**: 书院编号 (外键, 指向 **college** 表)

UNIQUE/NOTNULL(school\_number)

### **student\_course** 学生-课程表

- **student\_id**: 学号（外键，指向 **student** 表）
- **course\_id**: 课程编号（外键，指向 **course** 表）

UNIQUE/NOTNULL(student\_id, course\_id)

### **teacher** 教师表

- **id**: 教师标识符，无实意义（自增主键）
- **name**: 姓名

UNIQUE/NOTNULL(name)

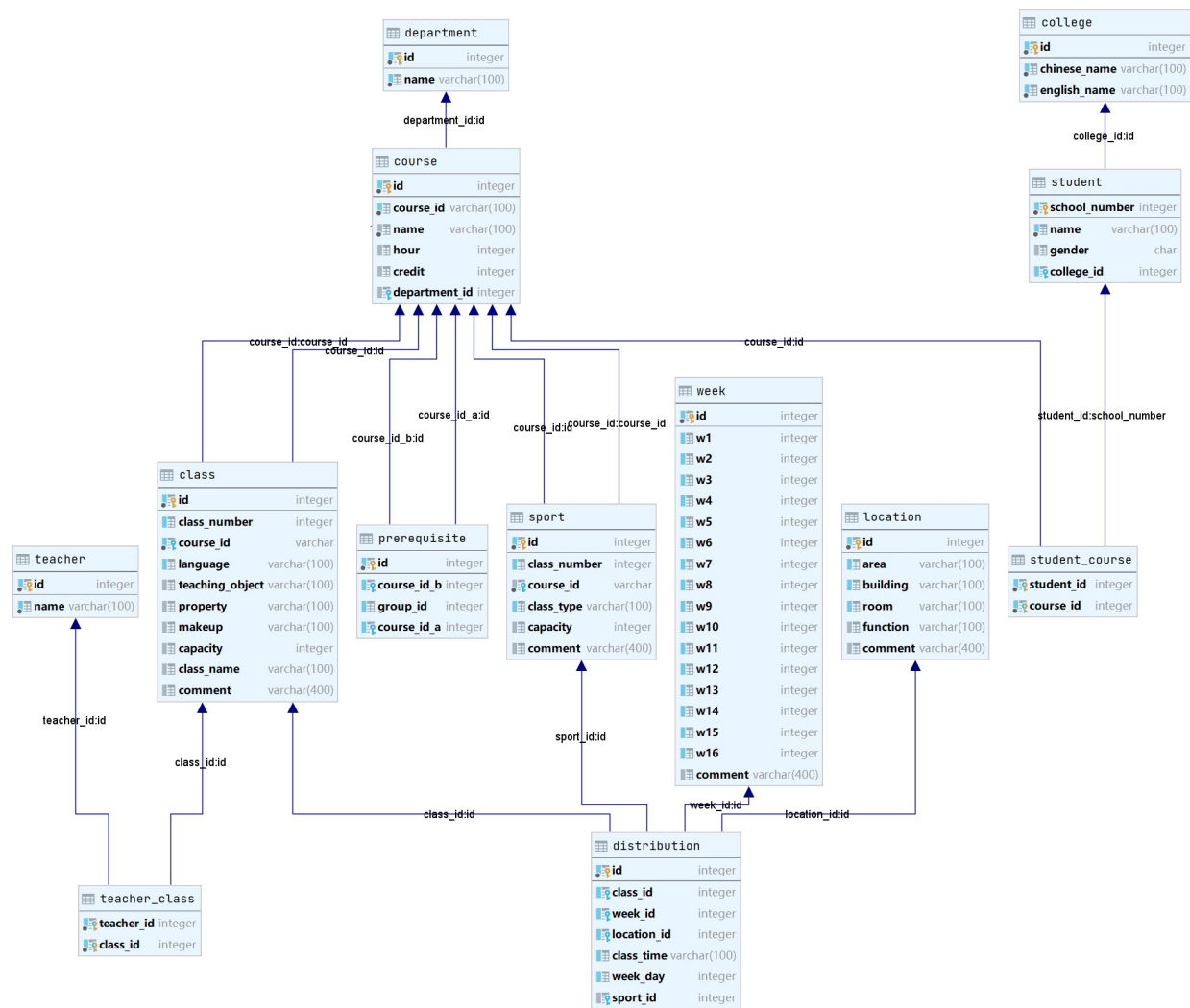
### **teacher\_class** 教师-班级表

- **teacher\_id**: 教师编号（外键，指向 **teacher** 表）
- **class\_id**: 班级编号（外键，指向 **class** 表）

UNIQUE/NOTNULL(teacher\_id, class\_id)

## 2.3 表结构图

由 **DataGrip** 生成的结构图如下：



Powered by yFiles

注意：右上角的两张表是将原始数据导入进了数据库所生成的表

我们的表设计满足以下的设计需求

- 满足三大范式
- 每个表都有主键
- 每个表都有外键/有表的外键指向
- 没有成环
- 每个表都有 NOTNULL 列和 UNIQUE 列
- 对数据使用了正确的类型

## 2.4 可拓展性

我们设计的表具有较强的可拓展性：

- **teacher\_class** 和 **student\_course** 表满足了老师为老师分配班级和为学生分配选课的便捷操作性，在实际选课情况下，只需要更改上述两个表即可调整老师和学生的选择

- college, department, week, location 表可以任意拓展新的情况
- course, class, distribution 具有三层结构，充分描述了任何一门课程的细节和分配，sport 解决了体育课与其它课程不同的需求，当需要加课程、班级、分配的时候，可以通过快速修改以上 4 表的内容
- prerequisite 表解释了与或关系，满足了先修课的复杂关系

## 第三部分：数据导入

### 1. 原始数据导入

我们使用 `jwxt_parser.py` 将 `course_info.json` 文件转化成了 CSV 格式

- `course_info.csv`
- `select_course.csv`

### 2. 数据整合与表导入

#### 2.1 建表

详见：[DDL.sql](#)

#### 2.2 通过数据库语言整合表

部分表使用导入数据库的原始数据，通过执行 SQL 命令筛选出来，再通过 **DataGrip** 直接导入

完全通过 SQL 导入的表有

1. teacher
2. department
3. course
4. college
5. student
6. sport
7. teacher\_class

部分通过 SQL 命令，再手动修改的表有



### 1. location

部分地点需要手动挑出来修改，小的细节需要人工修改

### 2. class

体育课、班级号码的划分需要人工修改

详见：[GenerateData.sql](#)

## 2.3 通过Python处理导入

部分表不能通过 SQL 命令简单执行，由于情况复杂（涉及外键），需要用 Python 处理后再导入

### 1. student\_course

### 2. distribution

详见：

- [create student-course.ipynb](#)
- [create distribution.ipynb](#)

## 2.4 人工处理

剩下的表由于自身原因无法用任何语言处理，只能人工处理

### 1. prerequisite

### 2. week

## 3. 导入性能测试

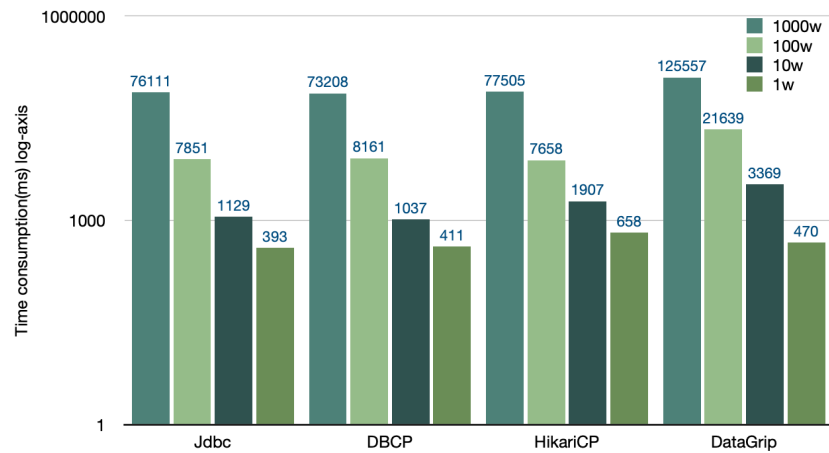
我们首先用 Java 脚本，选择使用 JDBC 作为 benchmark，选择传统数据库连接池代表之一 DBCP 测试传统数据库连接池性能，选择现在号称最快数据库连接池，作为 Spring Boot 内置的连接池的 HikariCP 连接池来对比连接池之间的性能差异。以及使用 DataGrip 内置的导入文件方法来对比脚本效率。Java 代码请见：[task2\\_import\\_java\\_pgsql](#), Python代码请见：[task2\\_import\\_python.ipynb](#)

我们还使用 Python 连接数据库，用了 3 种不同导入方法对比方法之间的效率，再整体对比 Java 脚本与 Python 脚本之间的导入效率。

- Postgresql
  - Java脚本

- 单线程，多线程高效率 JDBC
- 单线程，多线程数据库连接池 DBCP
- 单线程，多线程数据库连接池 HikariCP
- Python脚本
  - Python单线程 `copy_from`, `executemany`, `pd.io.sql.to_sql` 方法
- 命令行
  - 单线程，多线程命令行执行脚本导入
- DataGrip 图形化界面
- Mysql
  - java脚本，与Postgresql性能对比

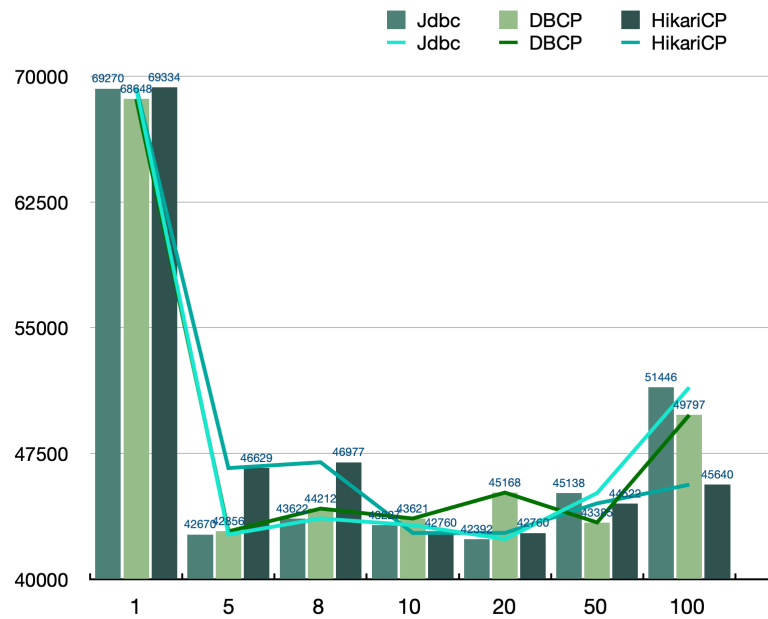
### 3.0 Java 脚本不同数据量单线程导入数据库性能（详细数据见附录时间对比表）



- 由上图，可以看出在单线程下数据库连接池的表现与 JDBC 没有很大差别，我们认为是在单线程的情况下数据库连接池初始化可能需要较多时间，JDBC 只需要初始化一个链接，所以没有明显性能优势。
- 使用 DataGrip 界面导入数据的速度相较 Java 脚本要慢很多。
- 由于纵坐标是 log 坐标刻度，可以看出随着数据量每次增加10倍，时间增加不是线性的。由此可见数据量的增加对于数据库处理时间的增加不是线性的，我们认为这是由索引引起的，将在索引一节详细研究。

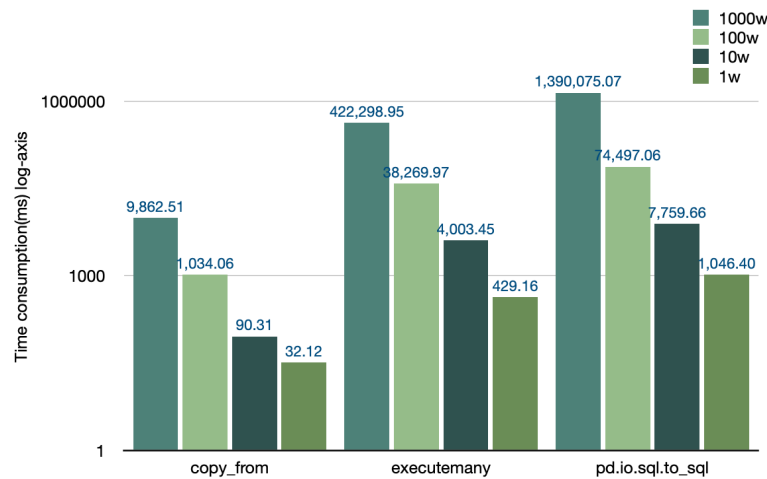
### 3.1 Java脚本不同线程jdbc，DBCP，HikariCP性能对比

电脑CPU数：4，数据量：单表1000万



- 由上图，首先可以看出多线程可以大大降低导入数据所需要的时间
- 由上图数据可以看出随着线程数增加，所需时间先降后增，在最接近笔记本 **CPU** 数时达到最小值。我们相信使用接近电脑 CPU 数量的线程数可以做到真正的并行，所需要花费的时间最短。当线程大于电脑 CPU 数量且逐渐增加时，真正可以并行的线程数量仍然等于 CPU 数量，但线程调度，线程等待需要耗时，导致时间消耗增加。
- 可以看到随着线程数量逐渐增加，**JDBC** 耗时逐渐大于数据库连接池的耗时，且差距逐渐变大。我们分析认为数据库连接池是在需要连接的时候从连接池中取可用连接，连接释放后放回池中，在大量需要连接的时候避免了连接的重复创建和销毁，减小了时间开销。
- 在线程数量较多的情况下，**HikariCP** 插入1000w数据的耗时小于 **DBCP** 小于 **JDBC**，总体结果符合预期

### 3.2 Python脚本导入数据库：



- 由上图数据可以看出 **python** 最快是使用 **copy\_from** 方法，是 PostgreSQL 的一个内置函数，在单表导入 1000w 数据时的速度相较单线程 Java JDBC 快了接近 8 倍。

### 3.3 命令行运行脚本多线程导入测试

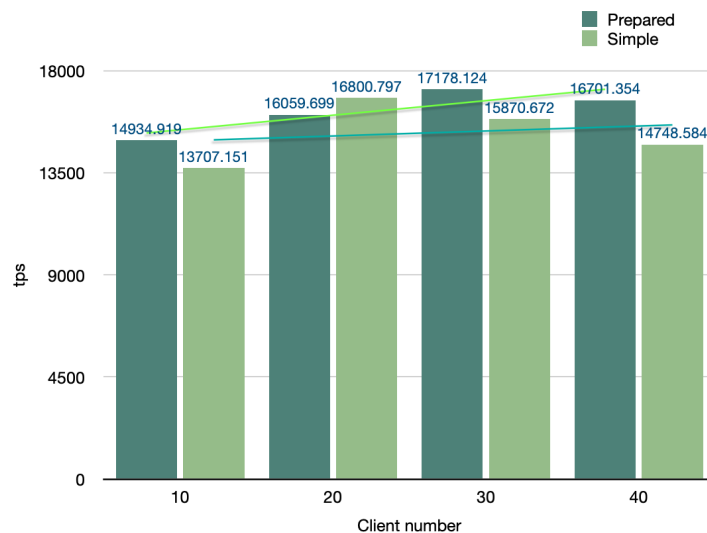
插入数据执行的 sql 脚本（由于命令行读取文件数据比较麻烦，我们采用生成随机数的方式生成插入数据）

```

1 \set aid random(1, 100000 * :scale)
2 \set bid random(1, 1 * :scale)
3 BEGIN;
4 INSERT INTO "Midterm_project".student_course1
  (student_id, course_id) VALUES (:aid, :bid);
5 END;
```

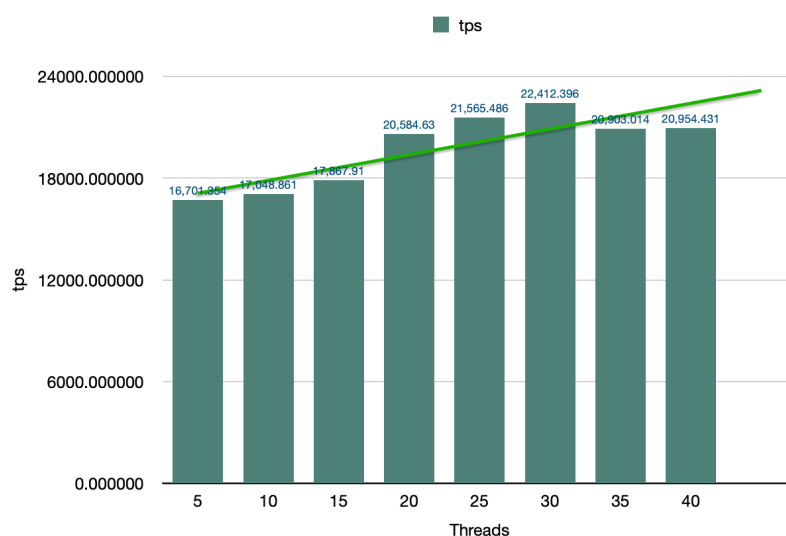
我们分别测试了不同 **client**（具体是指并发运行的数据库 **session** 数量）数量，不同工作线程数量，是否使用 **prepared statement** 情况下的数据库性能。由于此脚本不好控制共执行的事物次数，我们改用计时 120s 执行，将真正执行的事物数/执行时间得到的 **tps(transaction per second)** 作为衡量标准，以下为结果（测试结果详见附录）：

在工作线程数为5的情况下改变 **client number**，分别使用与不使用 **prepared statement** 的结果：



可以看到使用 **Prepared Statement** 执行的效率还是平均要高于不使用的，以及由趋势线可以看到数据库实际 **tps** 随着用户数量的增加有一定幅度的上升。

在 client 数量为 40 时，改变工作线程的数量，得到以下结果：



由趋势线以及柱状图的走向我们可以看到随着工作线程的增加，**tps** 有所增加。但是增加到一定程度后有所下降，可能与前面分析到的线程等待调度等原因有关。

### 3.4 与其他数据库（Mysql）导入数据对比

我们使用传统 JDBC ，对比 Mysql 和 PostgreSQL 在导入不同数据量数据的运行时间，由下表所示：

数据量	MYSQL(MS)	POSTGRE(MS)
-----	-----------	-------------

数据量	MYSQL(MS)	POSTGRE(MS)
1000w	73789	73174
100w	8658	8047
10w	1431	1162
1w	280	244

可以看到，Mysql 数据插入的平均性能不如 PostgreSQL，Mysql 和 PostgreSQL 数据库的其他性能对比将在后面数据库性能对比中详细介绍。

## 4. 确认所有数据导入完毕

我们将所有建好的表和数据导出成 CSV，以证明我们顺利的完成上述任务

详见：[data/GenerateData/](#)

- [college.csv](#)
- [department.csv](#)
- [week.csv](#)
- [location.csv](#)
- [course.csv](#)
- [class.csv](#)
- [sport.csv](#)
- [distribution.csv](#)
- [prerequisite.csv](#)
- [student.csv](#)
- [student\\_course.csv](#)
- [teacher.csv](#)
- [teacher\\_class.csv](#)

## 第四部分：比较数据库和文件操作

### 1. 数据与环境

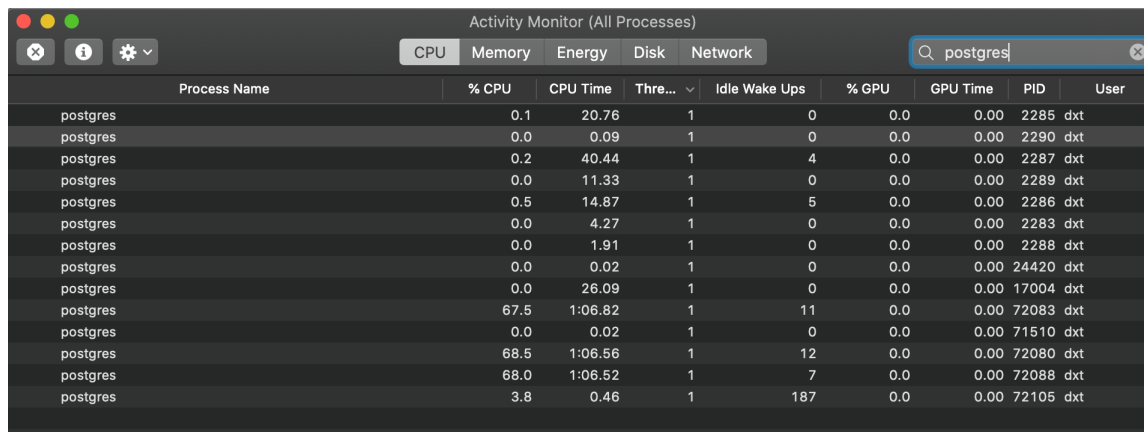
由于此次搭建的数据库多数表数据量都较小，测试没有很大的参考意义，我们除了在这次搭建的数据库中做了和文件系统的测试以外（使用student\_course, student, course 表），还额外建了总数据量大，数据种类多，varchar类型较长的宽表来进行测试。

## 2. 实验设计

与文件系统的对比我们主要是用 **Postgresql**, **Mysql** 数据库的增删改查操作与 **Python** 脚本进行同样操作的时间进行对比，结果如下所示：

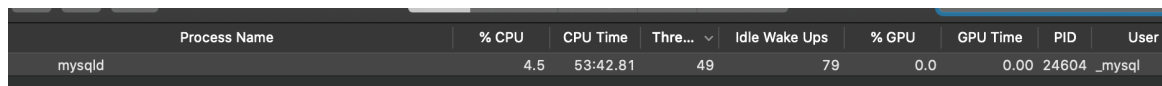
### 2.1 运行方式

首先对于 **Mysql** 和 **PostgreSQL**，**Mysql** 是单进程多线程，而 **PostgreSQL** 是多进程单线程的方式，如下图所示：



The screenshot shows the Activity Monitor window with the 'CPU' tab selected. A search bar at the top right contains 'postgres'. The table below lists 15 postgres processes, each with its own PID and user (dxt).

Process Name	% CPU	CPU Time	Thre...	Idle Wake Ups	% GPU	GPU Time	PID	User
postgres	0.1	20.76	1	0	0.0	0.00	2285	dxt
postgres	0.0	0.09	1	0	0.0	0.00	2290	dxt
postgres	0.2	40.44	1	4	0.0	0.00	2287	dxt
postgres	0.0	11.33	1	0	0.0	0.00	2289	dxt
postgres	0.5	14.87	1	5	0.0	0.00	2286	dxt
postgres	0.0	4.27	1	0	0.0	0.00	2283	dxt
postgres	0.0	1.91	1	0	0.0	0.00	2288	dxt
postgres	0.0	0.02	1	0	0.0	0.00	24420	dxt
postgres	0.0	26.09	1	0	0.0	0.00	17004	dxt
postgres	67.5	1:06.82	1	11	0.0	0.00	72083	dxt
postgres	0.0	0.02	1	0	0.0	0.00	71510	dxt
postgres	68.5	1:06.56	1	12	0.0	0.00	72080	dxt
postgres	68.0	1:06.52	1	7	0.0	0.00	72088	dxt
postgres	3.8	0.46	1	187	0.0	0.00	72105	dxt



The screenshot shows the Activity Monitor window with the 'CPU' tab selected. A search bar at the top right contains 'mysql'. The table below lists a single mysql process with PID 24604 and user \_mysql.

Process Name	% CPU	CPU Time	Thre...	Idle Wake Ups	% GPU	GPU Time	PID	User
mysqld	4.5	53:42.81	49	79	0.0	0.00	24604	_mysql

### 2.2 数据库基本操作性能对比

代码请见：[db compare with file.ipynb](#)

使用 **student\_course**, **student**, **course** 表测试

查询 **student\_course** 表的记录条数

PostgreSQL:

```
1 SELECT count(1)
2 FROM student_course;
```

Mysql:

```
1 SELECT count(1)
2 FROM student_course;
```

Python:

```
1 start_time=time.time()
2 print(len(initial_data))
3 end_time=time.time()
4 print(end_time-start_time)
```

POSTGRESQL	MYSQL	PYTHON
1.218 s	2.08 s	0.000912 s

排序返回学生选课**student\_course**表

PostgreSQL:

```
1 SELECT count(1)
2 FROM student_course
3 ORDER BY course_id;
```

Mysql:

```
1 SELECT count(1)
2 FROM student_course
3 ORDER BY course_id;
```

Python:

```
1 start=time.time()
2 sorted_data=initial_data.sort_values(axis=0,ascending=True,by='course_id')
3 end=time.time()
```

POSTGRESQL	MYSQL	PYTHON
11.61 s	11.04 s	2.35 s

联合查询

PostgreSQL:



```

1 SELECT *
2 FROM student_course s1
3 LEFT JOIN student s ON s.school_number=s1.student_id
4 LEFT JOIN course c2 ON s1.course_id = c2.id;

```

Mysql:

```

1 SELECT *
2 FROM student_course s1
3 LEFT JOIN student s ON s.school_number=s1.student_id
4 LEFT JOIN course c2 ON s1.course_id = c2.id;

```

Python:

```

1 start_merge=time.time()
2 first=pd.merge(student_course,course,how='left')
3 second=pd.merge(first,student,how='left')
4 end_merge=time.time()
5 print(end_merge-start_merge)

```

POSTGRESQL	MYSQL	PYTHON
16.36 s	18.21 s	6.76 s

更新 **Uptate**

PostgreSQL:

```

1 UPDATE student_course
2 SET course_id=course_id+3
3 WHERE student_id%10=5;

```

Mysql:

```

1 UPDATE student_course
2 SET course_id=course_id+3
3 WHERE student_id%10=5;

```

Python:

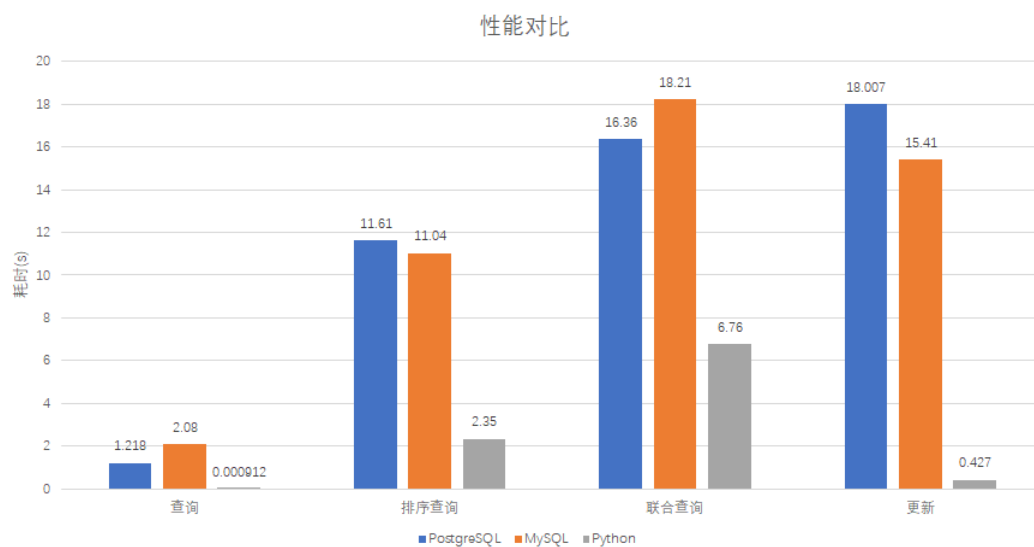
```

1 start_update=time.time()
2 initial_data[initial_data.student_id%10==5].course_id=initial_d
  ata[initial_data.student_id%10==5].course_id+3
3 end_update=time.time()
4 print(end_update-start_update)

```

POSTGRESQL	MYSQL	PYTHON
18.007 s	15.41s	0.427 s

经过统计，性能效果如下：



使用新建大数据宽表测试：

我们使用 sysbench 对 PostgreSQL 和 Mysql 创建了 15 张数据量为100万的表，使用8个线程，运行180s，测试数据库性能，结果如下图所示：

Mysql:

```
apple -- bash -- 85x29
Initializing worker threads...

Threads started!

SQL statistics:
  queries performed:
    read:          1397200
    write:         399200
    other:         199600
    total:         1996000
  transactions:    99800 (554.39 per sec.)
  queries:         1996000 (11087.71 per sec.)
  ignored errors:  0      (0.00 per sec.)
  reconnects:      0      (0.00 per sec.)

General statistics:
  total time:      180.0176s
  total number of events: 99800

Latency (ms):
  min:            2.91
  avg:            14.43
  max:            358.49
  95th percentile: 34.95
  sum:            1439803.80

Threads fairness:
  events (avg/stddev): 12475.0000/47.75
  execution time (avg/stddev): 179.9755/0.00
```

## Postgresql:

```
apple -- bash -- 94x31
Initializing worker threads...

Threads started!

SQL statistics:
  queries performed:
    read:          806540
    write:         230440
    other:         115220
    total:         1152200
  transactions:    57610 (319.89 per sec.)
  queries:         1152200 (6397.72 per sec.)
  ignored errors:  0      (0.00 per sec.)
  reconnects:      0      (0.00 per sec.)

Throughput:
  events/s (eps):  319.8859
  time elapsed:    180.0955s
  total number of events: 57610

Latency (ms):
  min:            2.21
  avg:            25.00
  max:            655.88
  95th percentile: 102.97
  sum:            1440207.76

Threads fairness:
  events (avg/stddev): 7201.2500/85.86
  execution time (avg/stddev): 180.0260/0.04
```

结果对比（180s）：

DATABASE	READ	WRITE	OTHER	TOTAL	TRANSACTION	TPS
Mysql	1397200	399200	199600	1996000	99800	554.39
PostgreSQL	806540	230440	115220	1152200	57610	319.89

相较而言Mysql在 180s 内的操作数量以及 transaction 数量较高。

我们还创建了三张数据量为 20000000 的表，同样使用 8 个线程运行 180s，得到的结果如下：

## Mysql:

```
apple — bash — 100x33

Initializing worker threads...

Threads started!

SQL statistics:
  queries performed:
    read:          1371888
    write:         391968
    other:         195984
    total:         1959840
  transactions:    97992 (544.33 per sec.)
  queries:         1959840 (10886.56 per sec.)
  ignored errors:  0      (0.00 per sec.)
  reconnects:      0      (0.00 per sec.)

General statistics:
  total time:      180.0223s
  total number of events: 97992

Latency (ms):
  min:             2.78
  avg:             14.69
  max:             663.00
  95th percentile: 38.25
  sum:             1439788.77

Threads fairness:
  events (avg/stddev): 12249.0000/40.35
  execution time (avg/stddev): 179.9736/0.01

(base) dxts-MacBook-Pro-5:~ dxt$
```

## PostgreSQL:

```
apple — bash — 94x31

Threads started!

SQL statistics:
  queries performed:
    read:          162400
    write:         46400
    other:         23200
    total:         232000
  transactions:    11600 (64.41 per sec.)
  queries:         232000 (1288.16 per sec.)
  ignored errors:  0      (0.00 per sec.)
  reconnects:      0      (0.00 per sec.)

Throughput:
  events/s (eps):  64.4081
  time elapsed:    180.1017s
  total number of events: 11600

Latency (ms):
  min:             11.26
  avg:             124.18
  max:             884.24
  95th percentile: 257.95
  sum:             1440462.08

Threads fairness:
  events (avg/stddev): 1450.0000/18.47
  execution time (avg/stddev): 180.0578/0.04
```

结果对比（180s）：

DATABASE	READ	WRITE	OTHER	TOTAL	TRANSACTION	TPS
Mysql	1371888	391968	195984	1959840	97992	544.33
PostgreSQL	162400	46400	23200	232000	11600	64.41

相较而言 Mysql 在 180s 内的操作数量以及 transaction 数量较高。

数据库读取，写入，更改，删除性能单独测试

我们使用不同脚本，分别对 Mysql 数据库对于单

- 只读写(read and write)
- 只读(read only)
- 只更新存在索引的列(update index)
- 只更新不存在索引的列(update non index)
- 只写(Write only)
- 只插入(insert only)
- 只删除(delete only)

进行的单独的性能测试（8个线程），结果如下表所示，详细执行结果请见附录 [detail information on mysql sysbench test.pdf](#)

READ ONLY	UPDATE INDEX	UPDATE NON INDEX	WRITE ONLY	INSERT ONLY	DELETE ONLY
1854.17tps	1554.74tps	4719.05tps	1096.53tps	5374.52tps	4776.66tps

高并发环境下的数据库压力测试

我们除了使用 8 线程以外，还使用了 24 线程，64 线程，84 线程来测试 Mysql 数据库的读写性能，结果如下图所示，详细执行信息请见附录 [detail information on mysql sysbench test.pdf](#)

24线程	64线程	84线程
1235.45tps	1146.52tps	669.20tps

可以明显的看到，随着连接数量的增加，数据库在处理每一个链接的tps有明显的下降。

## 2.3 读写操作性能对比

我们使用 python 和 `student` 表来测试下面的情况，它一共有 3999920 条数据

数据读入RAM

我们以 `student` 表为例测试了三种数据保存类型读入 RAM 所花费的时间，测试文件详见 [Compare.ipynb](#)

其中由于 student 表没有 json 文件，所以我们生成了 [student.json](#)

- CSV 读入 RAM: 使用 `pandas` 包的 `read_csv()` 方法读入
- JSON 读入 RAM: 使用 `json` 包的 `with open()` 方法读入
- PostgreSQL 读入 RAM: 使用 `sqlalchemy` 包与 `pandas` 结合后的 `read_sql_query` 方法读入

进行 10 次测试取平均值，效果如下

CSV 读入 RSM	JSON 读入 RAM	POSTGRESQL 读入 RAM
4.38 s	11.81 s	11.33 s

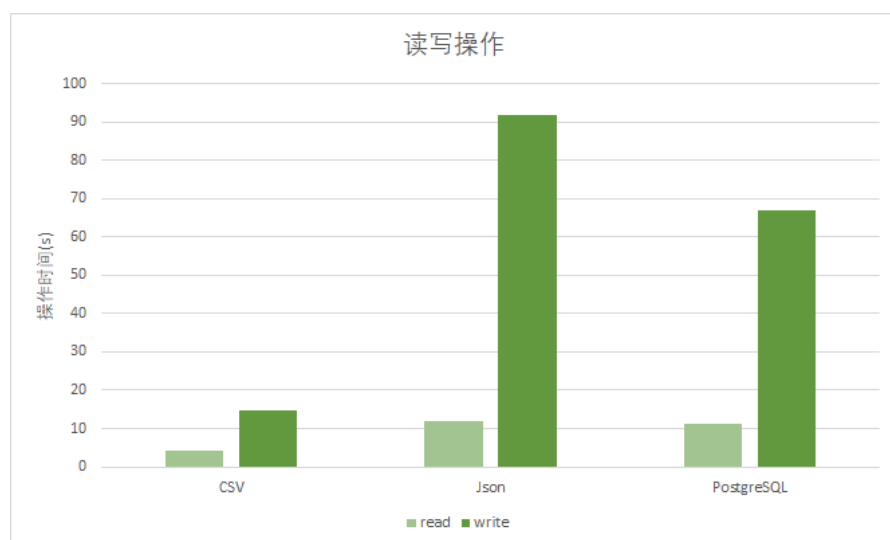
### 从RAM写入数据

我们以 student 表为例测试了从 RAM 保存数据进入三种数据保存类型所花费的时间，测试文件详见 [Compare.ipynb](#)

- RAM 保存 CSV: 使用 `pandas` 包的 `to_csv()` 方法保存
- RAM 保存 JSON: 使用 `json` 包的 `dump()` 方法保存
- RAM 保存 PostgreSQL: 使用 `psycopg2` 包 `copy_from` 方法保存

RAM 保存 CSV	RAM 保存 JSON	RAM 保存 POSTGRESQL
14.88 s	91.72 s	66.77 s

### 总结



通过实验可以看出，数据读写的时间耗费均为  $CSV < PostgreSQL < Json$ ，这与某些情况有关

- CSV 读写使用 `pandas` 操作，`pandas` 底层使用 C 实现，性能高
- Json 读写使用 `json` 进行简单读写，时间相对慢
- PostgreSQL 读写涉及到数据库操作，但由于涉及到与 `python` 交互，以及输出为 `pandas` 的 `Dataframe` 格式数据，时间相对慢

而相同数据占据的内存为：

.CSV	.JSON	POSTGRESQL
87.7 MB	347 MB	199MB

- .csv 占用的内存最小，因为它仅为逗号分割的数据文件，除此之外不表示任何信息
- PostgreSQL 占据的内存中等，因为主键、索引、外键等约束让它除了保存数据以外还有其它的数据结构存储
- .json 占用的内存最大，因为它有键值对关系，比如每行数据都有 "name": 名字等关系

## 第五部分：Bouns

### 1. 用户权限管理

PostgreSQL 对于数据库用户安排了一些权限，通过阅读 PostgreSQL 官方手册，不同用户具有的权限大致如下。

#### 1.1 用户权限

	创建数据库	角色修改	流复制	SELECT	CREATE	DELETE	ALTER
SUPERUSER	✓	✓	✓	✓	✓	✓	✓
CREATEDB	✓						
CREATROLE		✓					
REPLICATION			✓				
GRANT SELECT				✓			
GRANT CREATE					✓		
GRANT DELETE						✓	
GRANT ALTER							✓

## 1.2 赋予用户权限的命令

### 超级用户

```
1 CREATE USER super_user WITH PASSWORD 'password' SUPERUSER;
```

### 可创建数据库用户

```
1 CREATE USER createdb_user WITH PASSWORD 'password' CREATEDB;
```

### 普通用户

```
1 CREATE USER user1 WITH PASSWORD 'password' LOGIN;
2 GRANT ALL PRIVILEGES ON DATABASE postgres TO user1;
3 GRANT ALL PRIVILEGES ON TABLE project1.class TO user1;
4 # 授权SELECT
5 GRANT SELECT ON project1.class TO user2;
6 # 取消授权SELECT
7 REVOKE SELECT ON project1.class FROM user2;
```

## 1.3 用户使用场景设计

结合本 Project 可能存在的现实场景设计，我们提出了以下的目标用户以及他们可以获得的权限，场景如下：

	创建数 据库	角色 修改	流 复制	SELECT	CREATE	DELETE	ALTER
创建者+管理者 (最高层)	✓	✓	✓	✓ all	✓ all	✓ all	✓ all
数据库管理员			✓	✓ all	✓ all	✓ all	✓ all



	创建数 据库	角色 修改	流 复 制	SELECT	CREATE	DELETE	ALTER
教务系统管理员				✓ all	✓ course class sport distribution teacher- class student- course week prerequisite	✓ course class sport distribution teacher- class student- course week prerequisite	✓ course class sport distribution teacher- class student- course week prerequisite
老师				✓ all	✓ class course sport teacher- class distribution prerequisite		✓ class course sport teacher- distribution prerequisite
辅导员				✓ all			
学生				✓ all	✓ student- course	✓ student- course	✓ student- course

## 1.4 用户需求

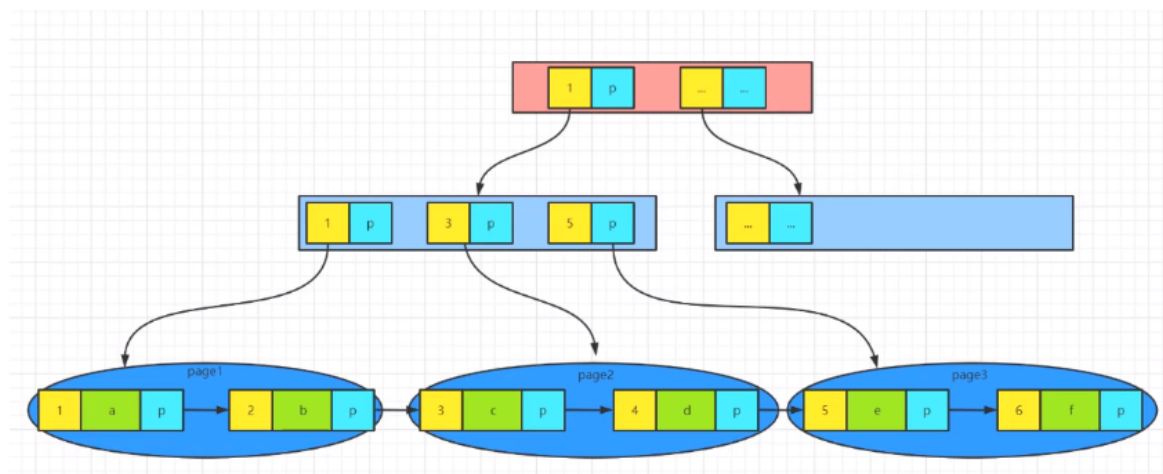
根据用户设定，我们初步为不同的用户设计了查询的需求。

用户需求的 SQL 样例详见 [user.sql](#)

## 2. 索引

## 2.1 原理解释

索引是一种在数据库中使用的**减少查询时间**的数据结构，通常使用 **B+** 树作为底层实现原理，使用它将会减少大数据量的查询时间。



数据库使用B+树的示意图

但是，由于数据存储在了一种新的数据结构中，索引也增加了存储量，且对于增删改都会增加时间，如何平衡查询和增删改的时间是设计索引需要注意的：

推荐下列情况创建索引

- 频繁搜索的列
- 经常用作查询选择的列
- 经常排序、分组的列
- 经常用作连接的列（主键/外键）

不推荐下面情况创建索引

- 仅包含几个不同值的列
- 表中仅包含几行

数据库会自动满足以下条件之一的列创建索引

- 主键列
- UNIQUE约束列

## 2.2 建立索引的命令

在PostgreSQL中建立索引的命令如下：

### 创建索引

```
1 CREATE [UNIQUE|FULLTEXT|SPATIAL] INDEX <索引名> ON <表名> <列名1,列名2,...>
```

### 删除索引

```
1 DROP INDEX <索引名> ON <表名>
```

### 查看索引

```
1 SHOW INDEX FROM <表名>
```

## 2.3 实验

### 索引设计

在我们的表中，除了 `student` 和 `select_course` 以外，其它的表数据都相当小，使用索引与否不会很影响查询效率。

`student_course` 作为关联表，在设计表的时候具有 `UNIQUE( studnt_id, course_id )` 约束，所以本就作为联合索引，效果不明显。

`student` 表中 `gender` 和 `college` 仅包含个位数的值，不适合创建索引，所以我们仅仅能使用 `name` 字段创建索引并测试（索引创建耗时7378.423 ms）。

```
1 CREATE INDEX name_index ON student(name);
```

### 测试

我们测试如下 SQL 语句设置 `name` 为索引前后的执行时间

索引测试文件详见：[index.sql](#)

### 测试1

```
1 SELECT * FROM student
2 WHERE name = '谈凡走' OR name = '伏右段';
```

## 测试2

```
1 SELECT s.name, sc.course_id, c.course_id FROM student s
2 INNER JOIN student_course sc ON s.school_number = sc.student_id
3 INNER JOIN course c ON sc.course_id = c.id
4 WHERE s.name = '谈凡走' OR s.name = '伏右段';
```

## 测试3

```
1 SELECT c.course_id, c.name, c.hour, c.credit, d.name AS
   department FROM student s
2 INNER JOIN student_course sc ON s.school_number = sc.student_id
3 INNER JOIN course c ON c.id = sc.course_id
4 INNER JOIN department d ON c.department_id = d.id
5 WHERE s.name = '谈凡走';
```

## 测试4

```
1 SELECT course.course_id AS course_ID, course.name AS name,
2        course.hour AS hour, course.credit AS credit,
3        c.class_name AS class_name, c.language AS language,
4        t.name,
5        c.teaching_object AS teaching_object, c.property AS
   property,
6        c.capacity AS capacity, d.class_time AS time,
7        d.week_day AS week_day, l.comment AS location
8 FROM course
9 INNER JOIN class c ON course.course_id = c.course_id
10 INNER JOIN distribution d ON c.id = d.class_id
11 INNER JOIN location l ON d.location_id = l.id
12 INNER JOIN teacher_class tc ON c.id = tc.class_id
13 INNER JOIN teacher t ON tc.teacher_id = t.id
14 INNER JOIN student_course sc ON course.id = sc.course_id
15 INNER JOIN student s ON sc.student_id = s.school_number
16 WHERE s.name = '谈凡走';
```

## 测试5

```

1 SELECT sum(c.credit) AS credit_sum FROM student s
2 INNER JOIN student_course sc on s.school_number = sc.student_id
3 INNER JOIN course c on c.id = sc.course_id
4 WHERE s.name = '谈凡走'
5 GROUP BY s.school_number;

```

## 测试6

```

1 SELECT s.school_number FROM student s
2 INNER JOIN student_course sc on s.school_number = sc.student_id
3 INNER JOIN course c on c.id = sc.course_id
4 WHERE college_id = 1
5     AND s.name IN(
6         SELECT temp.name FROM student temp)
7 GROUP BY s.school_number
8 HAVING sum(credit) < 10;

```

## 性能比较

测试	BEFORE	AFTER
1	1964.559 ms	2.342 ms
2	3493.012 ms	3.650 ms
3	232.814 ms	1.041 ms
4	223.301 ms	11.386 ms
5	219.799 ms	1.302 ms
6	6823.459 ms	4735.753 ms

可以看出来创建了索引后数据库性能有了很大幅度的提升。

在使用 **WHERE** 后谓词与建立索引的列有关的时候，索引会极大的优化查询时间。

## 3. 文件IO

我们使用 **python** 来演示文件I/O操作，仍然以 **student** 表作为测试表

代码文件详见：[IO\\_test.ipynb](#)

### 3.1 打开操作 open

文件在打开的时候有多种操作

常用的模式有：

- **r**：以只读方式打开文件。文件的指针将会放在文件的开头
- **r+**：打开一个文件用于读写。文件指针将会放在文件的开头
- **w**：打开一个文件只用于写入。如果该文件已存在则打开文件，并从开头开始编辑，即原有内容会被删除。如果该文件不存在，创建新文件。
- **w+**：打开一个文件用于读写。如果该文件已存在则打开文件，并从开头开始编辑，即原有内容会被删除。如果该文件不存在，创建新文件。
- **a**：打开一个文件用于追加。如果该文件已存在，文件指针将会放在文件的结尾。也就是说，新的内容将会被写入到已有内容之后。如果该文件不存在，创建新文件进行写入。
- **a+**：打开一个文件用于读写。如果该文件已存在，文件指针将会放在文件的结尾。文件打开时会追加模式。如果该文件不存在，创建新文件用于读写。

#### .CSV 打开操作

使用基本的打开文件操作打开 .csv 文件

```
1 file = open('CSV文件路径', '打开方式', encoding='编码')
```

使用 pandas 包打开 .csv 文件

```
1 import pandas as pd
2 pd.read_csv('CSV文件路径')
```

使用 csv 包打开 .csv 文件

```
1 file = open('CSV文件路径', '打开方式', encoding='编码')
2 csv_file = csv.reader(file)
```

我们取 10 次计算三种情况所花费的时间，平均值如下：

OPEN	PANDAS	CSV
0.0003 s	3.6810 s	0.0009 s

## .Json 打开操作

由于 .json 文件的存储格式不同，我们使用另一种方式打开 .json 文件

```
1 import json
2
3 file = open('Json文件地址', '打开方式', encoding='编码')
4 json_data = json.load(file)
```

我们取 10 次计算三种情况所花费的时间，平均值为 9.57s

## 3.2 读操作 read

### .CSV 读操作

使用基本文件读取 .csv 文件

```
1 for i in range(100):
2     print(file.readline())
```

使用 pandas 包读取 .csv 文件

```
1 print(df[0:100])
```

使用 csv 包读取 .csv 文件

```
1 for i in range(100):
2     print(next(csv_file))
```

我们取 10 次计算三种情况所花费的时间，平均值如下：

OPEN	PANDAS	CSV
0.0039 s	0.0129 s	0.0050 s

## .Json 读操作

```
1 print(json_data[0:100])
```

我们取 10 次计算三种情况所花费的时间，平均值为 0 s

## 3.3 写 write

### .CSV 写操作

使用基本文件写入 .csv 文件

```
1 file = open('../data/Test Data/student.csv', 'r', encoding='UTF-8')
2
3 # 1. 创建文件对象
4 f = open('test.csv', 'w', encoding='utf-8')
5
6
7 # 4. 写入csv文件内容
8 for i in range(3999920):
9     w_str = str(file.readline()) + '\n'
10    f.write(w_str)
11
12 # 5. 关闭文件
13 f.close()
```

使用 pandas 包写入 .csv 文件

```
1 df.to_csv('test.csv')
```

使用 csv 包写入 .csv 文件

```
1 file = open('../data/Test Data/student.csv', 'r', encoding='UTF-8')
2 csv_file = csv.reader(file)
3
4 # 1. 创建文件对象
5 f = open('test.csv', 'w', encoding='utf-8')
6
```



```

7  # 2. 基于文件对象构建 csv写入对象
8  csv_writer = csv.writer(f)
9
10 # 3. 构建列表头
11 csv_writer.writerow(["id", "name", "gender", "college"])
12
13 # 4. 写入csv文件内容
14 for i in range(3999920):
15     csv_writer.writerow(next(csv_file))
16
17 # 5. 关闭文件
18 f.close()

```

我们取 10 次计算三种情况所花费的时间，平均值如下：

OPEN	PANDAS	CSV
7.5211 s	10.9593 s	10.0678 s

### .Json 写操作

```

1  with open(filename, 'w') as file_obj:
2      json.dump(message, file_obj)

```

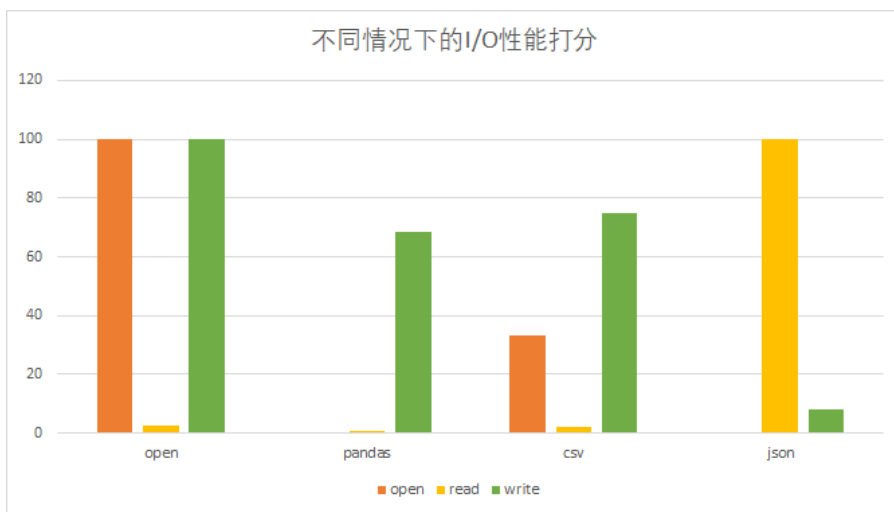
我们取 10 次计算三种情况所花费的时间，平均值为 91.7163 s

## 3.4 性能评估

我们在每个操作中耗时最短的为 100 分（性能最好），其它的满足

$$score = 100 * \frac{this.attribute}{min(attribute)}$$

作为评分标准，那么根据I/O操作评判如下图：



可以看到，在纯 I/O 操作上，直接时候用 `with open` 的效果最好，使用 `pandas` 反而速度更慢，而 `json` 在打开和写入的效果都很差，实时上也不适用于以 `.json` 的格式提取表的信息

## 4. 使用非关系型数据库 Redis 存储

### 4.1 Redis介绍

近年来非关系型数据库 NoSQL 使用趋势增大，非关系型数据库不再使用关系表的结构存储数据，转而有更多丰富的存储方法。

Redis 作为非关系型数据库的代表之一，主要以键值对的方式存储各类数据（当然还包括 list、set、hashSet、zset 等丰富的数据结构）。

`python` 中 `rejson` 包为 Redis 进一步提供了支持 `json` 形式数据的保存、查看和求改，它在基于 Redis 的基本命令功能的基础上进一步封装了由 `.json` 的一系列命令。

### 4.2 Redis数据库设计

获得更多 `rejson` 的信息，可以访问 <https://pypi.org/project/rejson/0.4.0/>

基于 `course_info.json` 与 `select_course.csv` 两张表本身就 `.json` 对象非常相似，我们的数据库设计几乎不需要很大的工作，下面我们主要阐述设计原理：

#### `student_info`

- `key`: 学号
- `value`: json 对象封装的数据

```

1  {'id': '<学号>',
2   'name': '<姓名>',
3   'gender': 'F',
4   'college': '<书院名称>',
5   'course': ['<课程1>', '<课程2>' ... 最多6门 ]}

```

示例

```

1  key: 11000100
2  value:
3  {'id': '11000100',
4   'name': 'Shi chechao ',
5   'gender': 'F',
6   'college': 'Gryffindor',
7   'course': ['CS318', 'IPE102', 'CS208', 'CH214', 'ESE212',
8             'BMEB324']}

```

## course\_info

- **key**: 课程编号（如CS305）
- **value**: json 对象封装的数据

```

1  {'courseId': '<课程编号>',
2   'department': '<开课院系>',
3   'prerequisite': ['先修课1', '先修课2' ...],
4   'courseHour': <课时>,
5   'courseCredit': <学分>,
6   'courseName': '<课程名称>',
7   'class': [
8       {'totalCapacity': <班级容量>,
9        'className': '<班级名称>',
10       'teacher': ['教师1'...],
11       'classList': [
12           {'weekList': ['1','2','3','4','5','6','7','8',
13                        '9','10','11','12','13','14','15'],
14            'location': '<班级上课地点>',
15            'classTime': '<上课时间（第几节课）>',
16            'weekday': <星期几>},
17
18           {<分配2>},
19           {<分配3>}...

```

```

20     ]
21
22     {<班级情况2>},
23     {<班级情况3>},...
24 ]
25 }

```

## 示例

```

1  key: 'CS203'
2  value:
3  {'courseId': 'CS203',
4   'department': '计算机科学与工程',
5   'prerequisite': ['CS102A'],
6   'courseHour': 64,
7   'courseCredit': 3,
8   'courseName': '数据结构与算法分析',
9   'class': [
10    {'totalCapacity': 50,
11     'className': '英文班',
12     'teacher': ['程然'],
13     'classList': [{'weekList':
14 ['1','2','3','4','5','6','7','8',
15 '9','10','11','12','13','14','15'],
16     'location': '荔园6栋404机房',
17     'classTime': '9-10',
18     'weekday': 1},
19    {'weekList':
20 ['1','2','3','4','5','6','7','8',
21 '9','10','11','12','13','14','15'],
22     'location': '荔园2栋201',
23     'classTime': '7-8',
24     'weekday': 1}]}

```

## 注意

- 由于购买的云服务器不支持中文编码，所以我们传入的数据为 JSONEncoder 编码好的 jaon 字符串，从 Redis 获得后，再用 JSONDecoder 解码获得指定的 json 数据

- 在 Linux 调试下 Redis 是可以支持中文的，但是由于时间关系我们没有调试

导入包

```
1 from json import JSONEncoder, JSONDecoder
```

使用 JSONDecoder 获得解码后的中文信息

```
1 JSONDecoder().decode(rj.jsonget('BIO320'))
```

```
1 {'courseId': 'BIO320',
2  'department': '生物系',
3  'prerequisite': ['BIO102A'],
4  'courseHour': 48,
5  'courseCredit': 3,
6  'courseName': '分子生物学',
7  'class': [{ 'totalCapacity': 50,
8              'className': '中英双语班',
9              'teacher': '李瑞熙',
10             'classList': [{ 'weekList': ['1', '3', '5', '7', '9', '11',
11                                     '13', '15'],
12                             'location': '一教305',
13                             'classTime': '3-4',
14                             'weekday': 2},
15                             { 'weekList': ['1', '2', '3', '4', '5', '6', '7', '8', '9', '10', '11', '12', '13', '14', '15'],
16                             'location': '一教305',
17                             'classTime': '1-2',
18                             'weekday': 5}]}],
19  'totalCapacity': 50,
20  'className': '英文班',
21  'teacher': '邓烽',
22  'classList': [{ 'weekList': ['2', '4', '6', '8', '10', '12', '14'],
23                  'location': '一教302',
24                  'classTime': '3-4',
25                  'weekday': 2},
26                  { 'weekList': ['1', '2', '3', '4', '5', '6', '7', '8', '9', '10', '11', '12', '13', '14', '15'],
27                  'location': '一教302',
28                  'classTime': '3-4',
29                  'weekday': 2}]}]
```

```

26     'location': '一教302',
27     'classTime': '3-4',
28     'weekday': 4}}]]}]
29 message["class"][1]
30 {'totalCapacity': 50,
31  'className': '英文班',
32  'teacher': '邓怿',
33  'classList': [{ 'weekList': ['2', '4', '6', '8', '10', '12',
34                        '14'],
35                  'location': '一教302',
36                  'classTime': '3-4',
37                  'weekday': 2},
38                { 'weekList':
39                  ['1', '2', '3', '4', '5', '6', '7', '8', '9', '10', '11', '12', '13', '14', '1
40                  5'],
41                  'location': '一教302',
42                  'classTime': '3-4',
43                  'weekday': 4}]]}]

```

## 4.3 数据导入

我们使用 python 直接处理 `course_info.json` 与 `select_course.csv` 并把它们导入进 Redis 数据库。

导入脚本详见: [RedisImportData.ipynb](#)

## 4.4 查询方法

我们设置了一些使用 Redis 查询和修改的样例。

详见: [NoSQL.ipynb](#)

# 5. 云端部署数据库以及访问

## 5.1 MySQL

我们将我们的Mysql数据库部署到了云服务器上

- Host: `cdb-omb6qd8s.cd.tencentcdb.com`

- Port: 10090
- User: user01

不需要密码可以访问我们的云端数据库，且我们给user01只赋予了查看cs307\_midterm schema 中的表的查询权限，不能查看其他schema或者修改任何数据，

## 5.2 Redis

可以通过使用 python 访问我们的云服务器来查看我们保存的数据，其中

- select\_course 部分因为数据库存储空间限制，只上传了100条数据作为测试
- course\_info 上传了所有的数据

下面是我们连接我们云数据库需要的操作：

```
1 # 导入包
2 from rejson import Client, Path
3
4 # 建立连接
5 rj = Client(
6     host='redis-16015.c258.us-east-1-4.ec2.cloud.redislabs.com',
7     port=16015,
8     decode_responses=True,
9     password='aTuwZ3vHob6s6Geoy8ZlniDgJf06CQ7R')
```

## 6. ORM与jdbc查询返回

ORM是Java对象与关系型数据库的映射，通过建立对应每一张表的Java类来为后续业务逻辑处理提供更多方法。每一个类中有对应表中所有的列的私有属性，构造方法，get，set方法，toString方法，方便将数据库返回的数据按需求取出。我们包装了一个泛型方法，可以通过传入不同的sql语句以及参数来查询不同内容，其中核心代码如下

- 返回单行记录的泛型方法：

```
1 public <T> T getQuerySingle(Class<T> clazz, String sql,
2     Object... args) {
3     Connection conn = null;
4     PreparedStatement ps = null;
```

```

4      ResultSet rs = null;
5      try {
6
7          conn = JDBCUtils.getConnection();
8          ps = conn.prepareStatement(sql);
9          for (int i = 0; i < args.length; i++) {
10             ps.setObject(i + 1, args[i]);
11         }
12         rs = ps.executeQuery();
13         ResultSetMetaData rsmd = rs.getMetaData();
14         int columnCount = rsmd.getColumnCount();
15
16         if (rs.next()) {
17             T t = clazz.newInstance();
18             for (int i = 0; i < columnCount; i++) {
19                 Object value = rs.getObject(i + 1);
20                 String columnLabel = rsmd.getColumnLabel(i +
21 1);
22                 Field field =
23                 clazz.getDeclaredField(columnLabel);
24                 field.setAccessible(true);
25                 field.set(t, value);
26             }
27             return t;
28         }
29     } catch (Exception e) {
30         e.printStackTrace();
31     } finally {
32         JDBCUtils.closeResource(conn, ps, rs);
33     }
34     return null;
35 }

```

- 返回多行记录的泛型方法:

```

1  public <T> List<T> getQueryList(Class<T> clazz, String sql,
2  Object... args){
3      Connection conn = null;
4      PreparedStatement ps = null;
5      ResultSet rs = null;
6      try {

```



```

7         conn = JDBCUtils.getConnection();
8         ps = conn.prepareStatement(sql);
9         for (int i = 0; i < args.length; i++) {
10             ps.setObject(i + 1, args[i]);
11         }
12         rs = ps.executeQuery();
13         ResultSetMetaData rsmd = rs.getMetaData();
14         int columnCount = rsmd.getColumnCount();
15         ArrayList<T> list = new ArrayList<>();
16
17         while (rs.next()) {
18             T t = clazz.newInstance();
19             for (int i = 0; i < columnCount; i++) {
20                 Object value = rs.getObject(i + 1);
21                 String columnLabel = rsmd.getColumnLabel(i +
22 1);
23                 Field field =
24 clazz.getDeclaredField(columnLabel);
25                 field.setAccessible(true);
26                 field.set(t, value);
27             }
28             list.add(t);
29         }
30         return list;
31     } catch (Exception e) {
32         e.printStackTrace();
33     } finally {
34         JDBCUtils.closeResource(conn, ps, rs);
35     }
36     return null;
37 }

```

- 使用方法案例:

```

1  /**
2      *
3      * @Description 通过老师名字查询老师上的course
4      *
5      */
6
7      @Test

```

```
8      public void getCourseFromTeacherTest(){
9          getCourseFromTeacher("朱悦铭");
10     }
11
12     public List<Course> getCourseFromTeacher(String
teacher_name){
13         String sql="select c2.id,\n" +
14             "        c2.course_id,\n" +
15             "        c2.name,\n" +
16             "        c2.hour,\n" +
17             "        c2.credit,\n" +
18             "        c2.department_id\n" +
19             "from class c\n" +
20             "        left join teacher_class tc on c.id =
tc.class_id\n" +
21             "        inner join teacher t on tc.teacher_id =
t.id\n" +
22             "        inner join course c2 on c.course_id =
c2.course_id\n" +
23             "where t.name=?";
24         List<Course>
courses=getQueryList(Course.class,sql,teacher_name);
25         System.out.println(courses);
26         return courses;
27     }
```

详情请见代码 [java\\_orm\\_code](#)