

CS307 Principles of Database Systems

Project 1 Report

成员信息

	姓名 (Student Name)	学号 (Student ID)
1	施米乐 (Shi Mile)	12212921
2	张伟祎 (Zhang Weiyi)	12210653
3	敖恺 (Ao Kai)	12211617

任务分配与贡献比

任务分配

任务1 E-R图: 敖恺、施米乐

任务2 数据库设计: 施米乐、张伟祎、敖恺

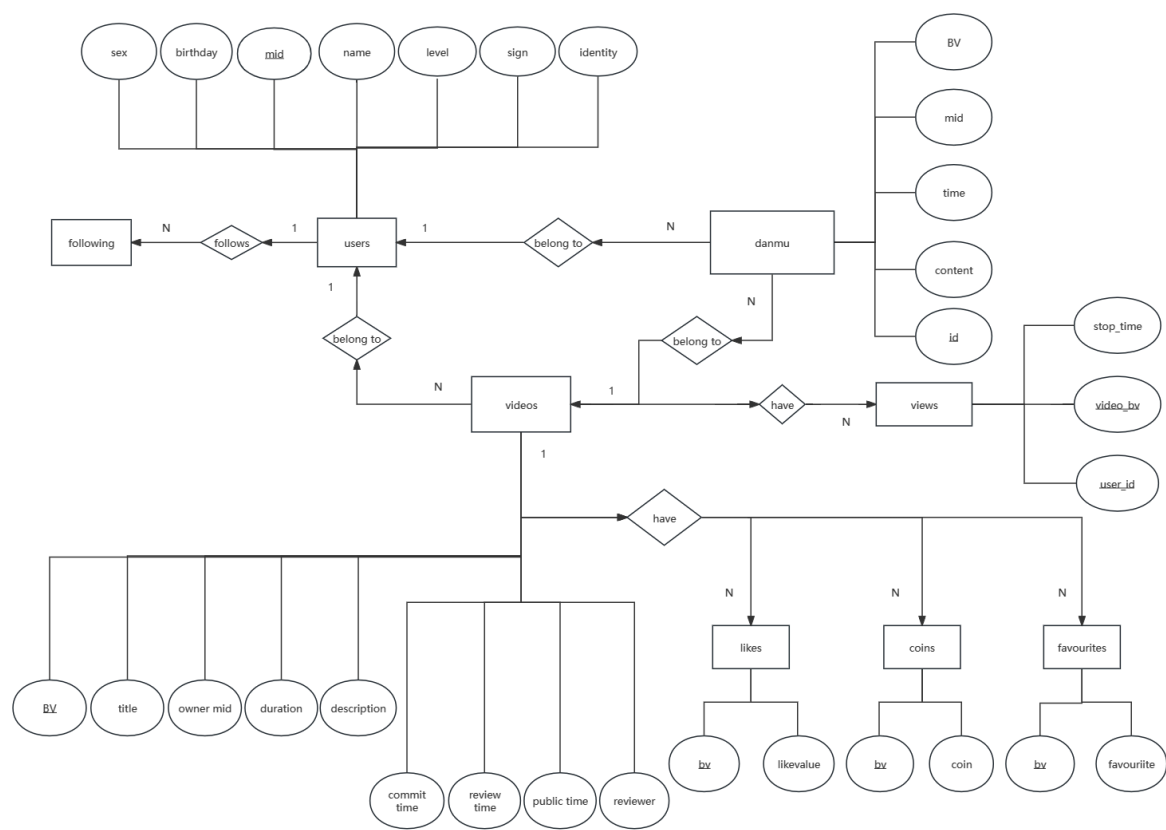
任务3 数据导入: 张伟祎

任务4 比较DBMS与文件I/O: 施米乐

贡献比

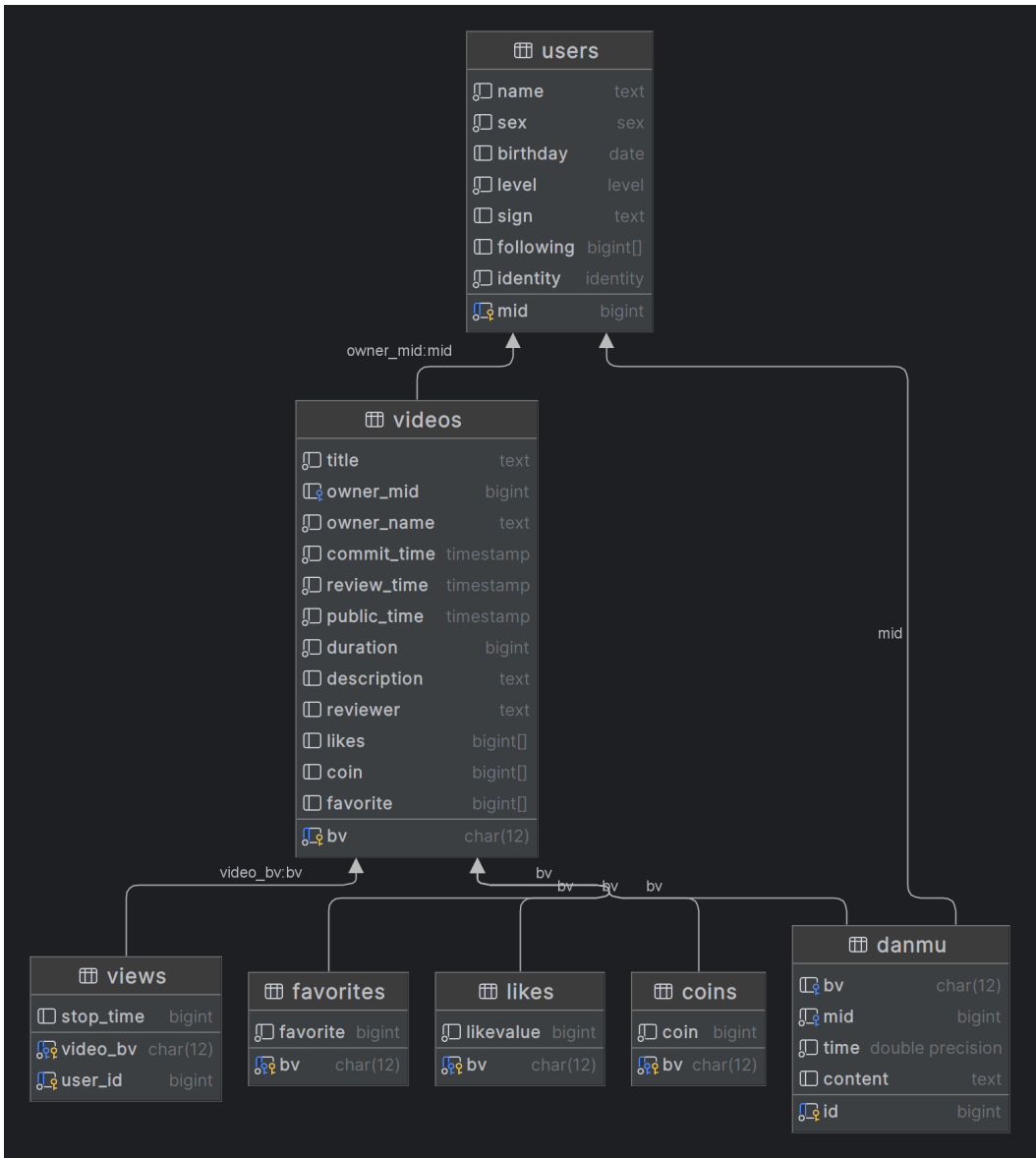
	姓名	贡献比
1	施米乐 (Shi Mile)	33.3%
2	张伟祎 (Zhang Weiyi)	33.3%
3	敖恺 (Ao Kai)	33.3%

任务1：E-R图



绘图工具：ProcessOn

任务2：数据库设计



设计关系型数据库，用于存储用户、视频、弹幕、点赞、投币、收藏、观看等信息。

首先，为了方便表格的设计，我们定义了几个枚举类型 `sex`、`level` 和 `identity`。经过对数据的筛选与观察，我们发现用户的 `sex` 类型有“保密”，“男”，“女”，用户的级别为“0”到“6”，用户身份分为“user”和“superuser”。

```
CREATE TYPE sex AS ENUM ('保密', '男', '女');
CREATE TYPE level AS ENUM ('0', '1', '2', '3', '4', '5', '6');
CREATE TYPE identity AS ENUM ('user', 'superuser');
```

然后，我们开始建表，并利用ER图建立表之间的链接关系，以下是对每个表的解释：

1. User 表 (users):

- 主键 (Primary Key): `Mid` (bigint)
- 列:
 - `Mid`: 用户的唯一标识号，bigint 类型。
 - `Name`: 用户创建的名称，text 类型，不能为空。

- **Sex**: 包括但不限于生物性别, 使用 ENUM 类型 sex 定义, 包括 '保密'、'男'、'女'。
- **Birthday**: 用户的生日, date 类型, 可以为空。
- **Level**: 用户等级 (根据系统决策标准评估的用户参与度), 使用 ENUM 类型 level 定义。
- **Sign**: 用户创建的个人描述, text 类型, 可以为空。
- **Following**: 用户关注列表, bigint 数组类型。
- **Identity**: 用户角色, 使用 ENUM 类型 identity 定义, 包括 'user' 和 'superuser'。

2. Video 表 (videos):

- 主键: **BV** (char(12))
- 外键 (Foreign Keys):
 - **Owner_Mid**: 对应users表中Mid
- 列:
 - **BV**: 视频唯一标识, char(12) 类型。
 - **Title**: 视频标题, text 类型, 不能为空。
 - **Owner_Mid**: 视频拥有者的唯一标识, bigint 类型, 外键关联 users 表
 - **Owner_Name**: 拥有者用户名, text 类型, 不能为空。
 - **Commit_Time**: 提交时间, timestamp 类型, 不能为空。
 - **Review_Time**: 审核时间, timestamp 类型, 不能为空。
 - **Public_Time**: 公开时间, timestamp 类型, 不能为空。
 - **Duration**: 视频时长, bigint 类型, 不能为空。
 - **Description**: 视频描述, text 类型, 可以为空。
 - **Reviewer**: 审核人, text 类型。
 - **likes**、**coin**、**favorite**: 分别是点赞、投币、收藏的用户列表, bigint 数组类型。

3. Danmu 表 (danmu):

- 外键 (Foreign Keys):
 - **BV** 是对应 Video 表 BV 的外键。
 - **Mid** 是对应 User 表 Mid 的外键。
- 列:
 - **BV**: 视频唯一标识, char(12) 类型, 外键关联 Video 表。
 - **Mid**: 用户唯一标识, bigint 类型, 外键关联 User 表。
 - **Time**: 弹幕发送时间, float 类型, 不能为空。
 - **Content**: 弹幕内容, text 类型。

4. Likes 表 (likes)、Coins 表 (coins)、Favorites 表 (favorites):

- 外键:
 - **BV** 是对应 Video 表 BV 的外键。
- 列:
 - **BV**: 视频唯一标识, char(12) 类型, 外键关联 Video 表。

- `likeValue`、`coin`、`favorite`：用户点赞、投币、收藏的值，`bigint` 类型，不能为空。

5. Views 表 (views):

- 复合主键 (Composite Primary Key): 由 `video_BV` 和 `user_id` 组成。
- 外键:
 - `video_BV` 是对应 Video 表 `BV` 的外键。
- 列:
 - `video_BV`：视频唯一标识，`char(12)` 类型，外键关联 Video 表。
 - `user_id`：用户唯一标识，`bigint` 类型。
 - `stop_time`：停止观看时间，`bigint` 类型。

总结：

这个数据库可以存储用户信息、视频信息、弹幕、点赞、投币、收藏、观看等多个方面的数据，通过外键连接关联起各个表，形成了一个相对完整的多表数据库结构。

任务3：数据导入

使用Java语言编写了数据导入脚本，将数据导入了所设计的表 `users`、`danmu`、`videos`、`likes`、`coins`、`favourites`、`views` 中，并在控制台打印数据导入条数与数据导入的速度，确保所有数据成功导入并方便评估数据导入的效率，并对导入数据速度进行了评估和优化，比较了不同数据导入方式的导入速度。

正确导入数据所需步骤：

- 1、建立导入数据所需java文件项目，安装导入数据所需库；连接好PostgreSQL数据库并完成建表；
- 2、确保配置文件 `loader.cnf` 在数据导入程序 `DataLoader.java` 所在文件夹下，并保证该配置文件包含正确的数据库连接信息；
- 3、将要导入的数据文件 `users.csv` 放在数据导入程序所在根目录下，并根据数据文件第一行所展示出的数据格式特点——包括数据内容、数据类型、分割符号信息，设计读取文件相关代码；
- 4、执行程序，数据文件将会被读取并将数据插入到数据库中，`[-v]`用于启动详细输出（verbose mode）；
- 5、通过调整 `BATCH_SIZE` 值的大小控制批处理大小，以减小内存消耗，同时保证数据的一致性。通过在控制台打印导入数据的加载速度，不断优化批处理大小，以达到在性能、内存使用、一致性和数据库负担的维度间的平衡。

必要的前提条件：

- 1、安装Java运行环境、安装并配置PostgreSQL数据库，确保数据库正常运行；
- 2、数据库连接参数和配置文件 `loader.cnf` 正确配置，能够正确连接到数据库。

注意事项：

- 1、考虑数据文件的特殊情况：

(1) 某些数据信息（如表 `users` 中的 `Following`；表 `videos` 中的 `Like`, `Coin`, `Favourite`, `View`）是数组格式，在数据导入过程中需要将读入的字符串去除两侧的双引号、中括号、逗号、空格等，根据逗号分割为字符串数组并去掉每个元素两端的单引号，将数据类型转为所需数据类型存入数据库；

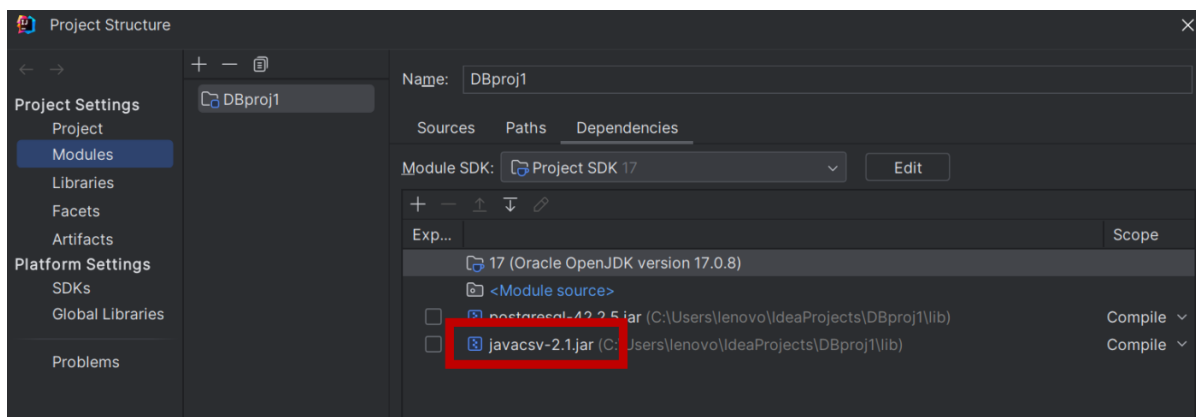
(2) 字段之间使用逗号分割，但需要特殊考虑字段值包含逗号的情况，如 `users` 第876条数据，在 `sign` 这一字符串数据内部有回车符；`danmu` 的 `BV` 和 `Mid` 都不是 `unique` 的，对此我们使用了正则表达式来进行处理：

```
String[] array = csvReader.get("view").substring(1, csvReader.get("view").length() - 1).split(", (?![^()]*\\))");
```

但后续又发现存在字符串内有 `,\` 的情况，于是调用了 `csvReader` 包来进行统一处理：

将 `javacsv` 的 jar 包放置在 `project` 根目录下的 `lib` 文件夹中，并在 `project` 中添加该 module

File -> Project Structure -> Modules -> '+' -> Jar or directories -> browse your driver path



(3) 在videos读取时, 因为Like、Coin、Favorite、View这四列由用户唯一标识号组成的列表数据长度过长, 产生如下报错:

Maximum column length of 100,000 exceeded in column 10 in record 8. Set the SafetySwitch property to false if you're expecting column lengths greater than 100,000 characters to avoid this error.

为了取消对列长度的限制, 将safetySwitch的属性设为了false以禁用安全开关

```
CsvReader csvReader = new CsvReader(new FileReader(fileName));
csvReader.setSafetySwitch(false);
```

但这样实际读取了后四列的数据而并未存入数据库, 是对开销的浪费, 一定程度上对导入数据速度产生了负面影响。

(4) csv文件中列名Like同时也是postgres中的关键词 故要在建表与导入数据时设一个新的名字从而避免冲突。

2、由于表之间的外键约束关系, 导入数据的顺序为 users --> videos --> likes&coins&favourites&views&danmu

每个实体表中的记录数

注: 以下数据导入过程均为首次尝试, 仅用于展示实体表中的记录数, 后文会对导入速度进行进一步优化

1、users 导入

```
37881 records successfully loaded
Loading speed : 4289 records/s
```

2、videos 导入

(1) 非数组类型数据

```
7865 records successfully loaded
Loading speed : 351 records/s
```

(2) Like 导入

```
86757948 records successfully loaded
Loading speed : 65770 records/s
```

(3) Coin 导入

```
80571520 records successfully loaded
Loading speed : 48865 records/s
```

(4) Favorite 导入

```
79181895 records successfully loaded
Loading speed : 42904 records/s
```

(5) View 导入

```
163997974 records successfully loaded
Loading speed : 13538 records/s
```

3、danmu 导入

```
12478996 records successfully loaded
Loading speed : 26744 records/s
```

总结：3个csv的数据导入数据库后，users 是37881条，videos 是7865条，danmu 是12478996条，各实体表记录数均达到预期，csv文件完整导入到数据库中。尽管在首次导入中，我们已经使用了下述导入数据优化策略的1-3条，但由于在导入数据时已经连接了danmu与users、videos之间的外键、videos.csv产生的子表与母表之间的外键等多重因素，导入速度需要进一步优化。

下文中将会介绍我们在提高导入速度方面进行的尝试及获得的成果。

对导入数据的优化

1、在loadData外部定义了预编译的SQL语句stmt，而非动态构建SQL语句，动态构建SQL语句会在插入数据的循环中每次都创建一个新的statement对象，导致执行效率的降低；且存在注入的潜在风险。

```
stmt = con.prepareStatement("insert into likes(BV,likevalue)" + " values(?,?)");
```

2、数据加载使用了事务和批量插入，关闭了自动提交模式，将插入操作包装在事务中，在循环中多次记录并一次性插入数据库，这减少了数据库通信开销，提高了性能；同时所有的插入要么都提交要么都回滚，这保证了数据的一致性。

```
try {
    stmt = con.prepareStatement("insert into students(studentid,name)"
                                + " values(?,?)");
} catch (SQLException e) {
    System.err.println("Insert statement failed");
    System.err.println(e.getMessage());
    closeDB();
    System.exit(1);
}
```



```
37881 records successfully loaded
Loading speed : 3712 records/s
```

3、使用了 `BATCH_SIZE`，批次大小需要在性能、内存使用、一致性和数据库负担的维度间进行权衡。合适的批次大小可以更好的控制内存使用同时提高处理速度，是对事务边界的更好控制，可以确保在出现错误时只回滚一小部分操作而非全部数据。

```
37881 records successfully loaded
Loading speed : 4289 records/s
```

注：到此为首次数据导入所实现的策略

在首次数据导入过程中，`BATCH_SIZE` 都取成了500，但 `BATCH_SIZE` 过大可能会导致内存消耗过高，因为需要在内存中维护较大的数据批次，还会导致单个事务包含大量的数据操作：如果出现错误一同回滚操作时可能产生较大消耗甚至数据丢失增加。而过小的 `BATCH_SIZE` 会导致频繁的提交操作，对数据导入速度的优化无法产生明显作用。综上所述，我们以500为基础，对 `BATCH_SIZE` 的大小进行了改变，又尝试了1000、5000、100、50四组并计算了平均的导入速度，发现 `BATCH_SIZE` 为100的时候，导入速度在五组中最大，如下：

```
37881 records successfully loaded
Loading speed : 4600 records/s
```

4、导入数据之前先删除相关表上的外键，导入完成后再重新创建。

在 `openDB()` 方法中加入下述代码，在导入数据之前执行sql语句以删除 `users` 与 `danmu` 之间的外键关系

```
try (Statement dropFkStmt = con.createStatement()) {
    String dropFkSQL = "ALTER TABLE danmu DROP CONSTRAINT
fk_danmu_users";
    dropFkStmt.execute(dropFkSQL);
}
```

在 `main()` 方法中导入数据后加入下述代码，重新添加外键约束

```
try (Statement addFkStmt = con.createStatement()) {
    String addFkSQL = "ALTER TABLE danmu ADD CONSTRAINT fk_danmu_users
FOREIGN KEY (Mid) REFERENCES users (mid)";
    addFkStmt.execute(addFkSQL);
}
```

导入速度如下：

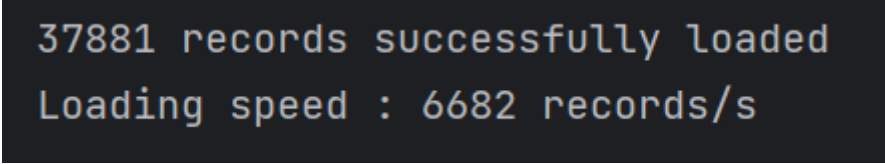
```
37881 records successfully loaded
Loading speed : 6590 records/s
```

和对照组的4289records/s相比较，考虑每次运行存在一定随机性，可以说速度基本上没有得到什么优化。这是因为在删除与重新添加外键均在计时区间内部，从sql语句执行的底层逻辑来看，增加了JDBC驱动程序（Java用于与数据库通信的标准API）与数据库建立通信并

执行sql语句的耗时以及数据库引擎处理sql语句的耗时。

于是我们恢复了 DataLoader 中的代码，将删除与重建外键约束放到了 DataGrip 中手动操作，仅计算数据导入本身的速度。

```
ALTER TABLE danmu
DROP CONSTRAINT fk_danmu_users;
```



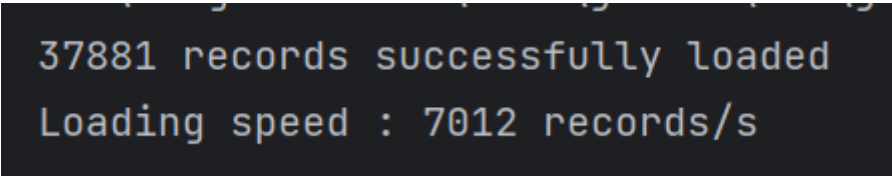
```
37881 records successfully loaded
Loading speed : 6682 records/s
```

这次的数据导入速度有了较为明显的提升。

5、导入数据之前先把表改成UNLOGGED模式，导入完成后改回LOGGED模式。导入信息不记录WAL日志，极大减少io，提升导入速度。

```
ALTER TABLE danmu SET UNLOGGED;
ALTER TABLE users SET UNLOGGED;

CREATE TABLE logged_users AS SELECT * FROM users;
DROP TABLE users;
ALTER TABLE logged_users RENAME TO users;
```



```
37881 records successfully loaded
Loading speed : 7012 records/s
```

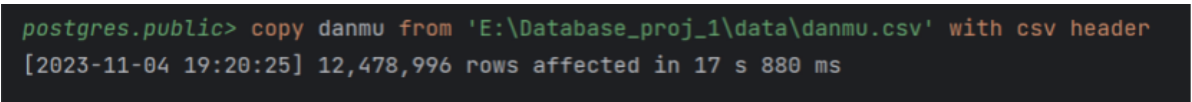
导入数据的多种方法及执行效率的比较

注：由于 users 表中日期需要将string进行标准格式转换，为了方便调用copy指令，且为了避免因为外键链接而导致在重复运行时多次删表重建，在这一部分我们使用 danmu 表进行多种导入数据方法的比较。

1、使用java脚本和csvreader包导入

由上文首次导入记录可知，导入速度为 26744 records/s，用时466s

2、使用 copy 命令执行postgresql导入csv



```
postgres.public> copy danmu from 'E:\Database_proj_1\data\danmu.csv' with csv header
[2023-11-04 19:20:25] 12,478,996 rows affected in 17 s 880 ms
```

3、使用其他编程语言导入csv文件

在java之外，我们又尝试使用了python进行csv文件的数据导入。

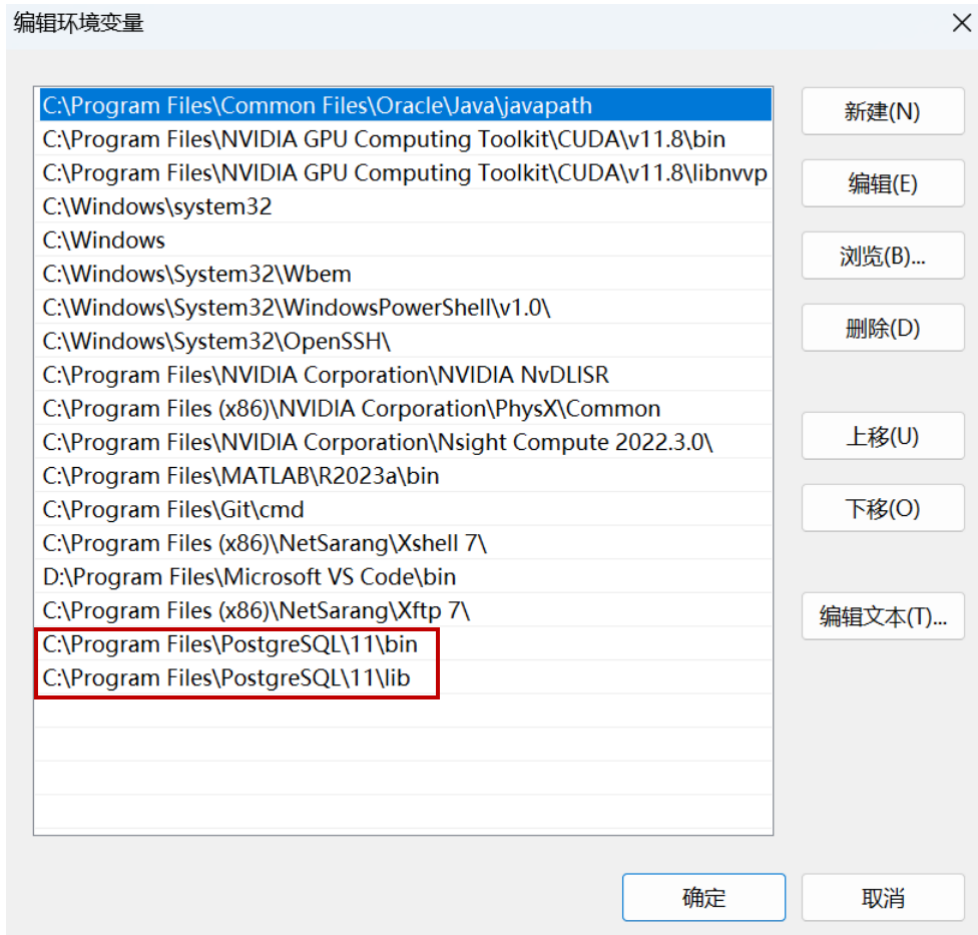
首先打开anaconda prompt，在所需环境下安装 psycopg2 库。

```
pip install psycopg2
```

```
(usual) (base) C:\Users\lenovo> python.exe -m pip install --upgrade pip
Looking in indexes: https://pypi.tuna.tsinghua.edu.cn/simple
Requirement already satisfied: pip in c:\users\lenovo\usual\lib\site-packages (22.3.1)
Collecting pip
  Downloading https://pypi.tuna.tsinghua.edu.cn/packages/47/6a/453160888fab7c6a432a6e25f8afe6256d0d9f2cbd25971021da6491d899/pip-23.3.1-py3-none-any.whl (2.1 MB)
  2.1/2.1 MB 6.4 MB/s eta 0:00:00
Installing collected packages: pip
  Attempting uninstall: pip
    Found existing installation: pip 22.3.1
    Uninstalling pip-22.3.1:
      Successfully uninstalled pip-22.3.1
  Successfully installed pip-23.3.1
```

为了让 Python 能够找到 PostgreSQL 的可执行文件，需要确保 PostgreSQL 的 `bin` 目录已经包含在系统的 PATH 环境变量中：

1. 使用快捷键 `win + R` 打开运行对话框，输入 `sysdm.cpl` 打开系统属性窗口；
2. 在系统属性窗口中，选择 "高级" 选项卡，在环境变量窗口中，找到系统变量部分，然后在变量名列中找到 "Path"。
3. 点击 "Path" 变量，添加 `C:\Program Files\PostgreSQL\11\bin` 到新建的路径中，保存更改。



在安装了库的 `usual` 环境下运行 `DBproj1.py` 文件，运行时间如下：

```
(usual) (base) C:\Users\lenovo\PycharmProjects\pythonProject4>python DBproj1.py
The time cost is: 2422.3563356399536 s
```

总结：

在java与python两种编程语言在大数据集 `danmu.csv` 文件数据导入性能方面的对比中，java 的性能优于 python。经过调研我们发现，java在数据导入，特别是在处理大型数据集时，处理速度会优于 python。这是因为 Java 是编译型语言，更接近底层硬件，执行效率更高；同时，Java 数据库驱动程序通常也经过高度优化，能够更有效地处理数据导入。

而在数据库自带copy命令与使用其他编程语言编写的脚本文件之间比较，显然copy命令在数据导入速度上大幅度领先。这是因为copy命令是由数据库管理系统提供的高度优化的数据导入工具，它能够直接将数据流式传输到数据库，而无需在应用程序和数据库之间来回传递数据，这无疑避免了其他编程语言脚本在于数据库服务器建立网络通信时产生的网络延迟；数据库引擎可以在内部进行高效的数据写入操作，例如并行处理、磁盘优化等，这也提高了数据导入性能。

任务4：比较 DBMS 与文件 I/O

test environment

1、hardware environment

- CPU: i7-12700H 4.70GHz 8核心
- RAM: 16G 4800HZ(在任务管理器中, postgres最多只用了2G)
- SSD:samsung 970 EVO NVMe
- 顺序读取速度 3500MB/s
- 顺序写入速度 2500MB/s
- 4k随机读 500K IOPS
- 4k随机写 480K IOPS

2、software environment

- DBMS: Postgresql 15.4.1
- OS:Windows 11 家庭中文版
- 版本: 22H2
- 操作系统版本: 22621.2428
- Language: Java
- Java version "17.0.4.1" 2022-08-18 LTS
- Java(TM) SE Runtime Environment (build 17.0.4.1+1-LTS-2)
- Java HotSpot(TM) 64-Bit Server VM (build 17.0.4.1+1-LTS-2, mixed mode, sharing)
- Jdk - 17.0.4.1

3、如果其他人要复制您的实验, 我们应该为他/她提供哪些必要信息:

- 数据库的版本, 使用编程语言的版本以及外接library;
- 运行环境的内存大小以及频率, cpu主频, 存储数据的硬盘读写速度, 以及如果远程连接服务器的网络带宽;
- 操作系统的位数以及版本。

Test data

我们直接选择了 `danmu` 数据作为测试数据。在这组数据中, 共有12,478,996 行数据, 每行数据都由四列组成。挑选这组数据的原因是, 数据构成较简单, 便于导入和select, 通过选择不同的限制, 可以进行一定范围内的对比。

同时, 我们也通过 `pgbench` 创建了一组随机数据来测试。

SQL code

在对数据库进行测试时, 我们选择了用copy导入数据, 并对 `insert` `select` `update` `delete` 等数据库最基础的增删改查操作以及一些较为高级的聚合函数进行了计时。

在计时操作中, 原本使用的是postgres自带的计时, 不仅提供的信息不足, 只能提供大体上的总时间, 还会计算多余的时间。所以, 我们使用了 `explain analyse` 来更加准确的计时。

```
postgres.public> explain analyse
select *from danmu where mid > 100000000 limit 10000
[2023-11-03 16:54:29] 6 rows retrieved starting from 1 in 69 ms (execution: 34 ms, fetching: 35 ms)
```

```
QUERY PLAN
1 Limit (cost=0.00..5374.56 rows=10000 width=54) (actual time=0.016..25.492 rows=10000 loops=1)
2   -> Seq Scan on danmu (cost=0.00..292523.11 rows=544274 width=54) (actual time=0.014..25.136 rows=10000 loops=1)
3     Filter: (mid > 100000000)
4     Rows Removed by Filter: 217926
5 Planning Time: 0.069 ms
6 Execution Time: 25.697 ms
```

如上图所示，两者相差约10ms，对于总时间就35ms的过程来说，是有比较大误差的。根据分析，这种误差可能是自带的计时多计算了一些连接服务器，编译语句的时间。

同时，该语句可以提供语句执行时间的分步计时，更加实用。

用一个例子来解释一下该语句的返回值。

```
Limit (cost=0.00..2095.81 rows=100000 width=54) (actual time=0.017..19.047
rows=100000 loops=1)
-> Seq Scan on danmu (cost=0.00..261348.09 rows=12470009 width=54) (actual
time=0.016..15.599 rows=100000 loops=1)
Planning Time: 0.082 ms
Execution Time: 20.932 ms
```

在这个例子中，我进行的操作是从 danmu 表里选了前100000个数据，接下来逐一解释返回值。

(1) Limit (cost=0.00..2095.81 rows=100000 width=54) (actual time=0.017..19.047 rows=100000 loops=1)

Cost 是postgres估计的运行时间，是通过postgres内置的统计模型计算的，虽然有参考意义，但价值不大，对于本次project可忽略。

Actual time 是实际运行时间，也是我们的目标数据。格式为“____..____”，在“..”前的是找到第一个符合标准的数据的时间，而之后就是总的执行时间（在该节点中，此概念稍后提及）。

(2) -> Seq Scan on danmu (cost=0.00..261348.09 rows=12470009 width=54) (actual time=0.016..15.599 rows=100000 loops=1)

Explain返回值是以节点的方式返回的，由于例子中的查询语句是对全表查询，故只有一个节点，也就是seq scan节点。若有多级查询，结果应该如下：

```
Sort
├── Hash Join
│   ├── Seq Scan
│   └── Hash
│       ├── Bitmap Heap Scan
│       └── Bitmap Index Scan
```

通过这样的方式，我们可以得知在语句执行过程中，每一步所需的执行时间

(3) Planning Time: 0.082 ms

Execution Time: 20.932 ms

毋须多言，就是语句计划的时间和执行的时间。计划时间是指postgres服务器在接收到 sql script 后花费了一点时间优化，但是也可以看到，对于较大数据查询时，可以忽略不计。

相应语句/操作的运行时间的比较研究

(一) DBMS的分析

在开始比较执行时间之前，我们先对DBMS的执行顺序以及他们的时间分析一下：

在postgres中，有一种JIT (just in time) 机制，可以意译为即时编译机制，也就是对执行的语句编译后，如果遇到相似的语句，就不用全部重新编译，节约一点时间。

什么叫相似的语句呢？经过实验，相似的范畴还是很大的。

首先，我们是怎样去判断有没有重新编译呢？我们选择是，观察planning time，也就是postgres优化语句所花的时间。如下两张图，都是执行同一个select语句，第一张图是第1次执行，第二张图是第2次执行。虽然execution time也有区别，但是运行了几次，确定这是在随机误差的范围内的。但对于planning time，第一次是0.393ms，而经过JIT机制后，只需要0.049ms，少了一个数量级。故我们可以发现，JIT预编译机制最大的优化是对于planning time的。

QUERY PLAN
1 Limit (cost=0.00..5374.56 rows=10000 width=54) (actual time=0.019..16.607 rows=10000 loops=1)
2 -> Seq Scan on danmu (cost=0.00..292523.11 rows=544274 width=54) (actual time=0.017..16.255 rows=10000 loops=1)
3 Filter: (mid > 100000000)
4 Rows Removed by Filter: 219333
5 Planning Time: 0.393 ms
6 Execution Time: 16.812 ms

QUERY PLAN
1 Limit (cost=0.00..5374.56 rows=10000 width=54) (actual time=0.012..18.799 rows=10000 loops=1)
2 -> Seq Scan on danmu (cost=0.00..292523.11 rows=544274 width=54) (actual time=0.011..18.419 rows=10000 loops=1)
3 Filter: (mid > 100000000)
4 Rows Removed by Filter: 219650
5 Planning Time: 0.049 ms
6 Execution Time: 19.000 ms

接下来，我们讨论JIT会对相似到什么程度的语句进行预编译。首先，我们重新连接一遍postgres数据库，以免过往的操作有影响。由于本部分与project没有太大关联，我们只探索一下三种情况。

- **select *from tablename & select * from table name where ...**

```
explain analyse
select * from danmu limit 10000 ;
```

	Planning time(ms)
First time	0.404
Second time	0.038

```
explain analyse
select * from danmu where mid > 100000000 limit 10000;
```

	Planning time (ms)
First time	0.051
Second time	0.045

很明显，只有第一次运行时才有编译。所以，where 限定条件对于JIT是相似的。

- **Select * from tablename where condition1& Select * from tablename where condition2**

```
explain analyse
select * from danmu where time < 10 limit 10000;
```

	Planning time (ms)
First time	0.442
Second time	0.053

```
explain analyse
select * from danmu where mid > 100000000 limit 10000 ;
```

	Planning time (ms)
First time	0.051
Second time	0.046

同样的，where条件的改变也不需要重新编译。（感觉这个对比有点不必要）

- **Select aggregate1() from tablename &select aggregate2 () from tablename**

```
explain analyse select count(*) from danmu limit 10000;
```

	Planning time (ms)
First time	0.307
Second time	0.054

```
explain analyse select **max**(Mid) from danmu limit 10000;
```

	Planning time (ms)
First time	0.225
Second time	0.064

终于发现了一个不相似的了 orz..... 不同的聚合函数，需要重新编译。

一条postgres语句，在你按下执行之后，先经过编译，然后被优化，最后再是执行。说了这么多，应该如何来计算DBMS的执行速度呢？主要的时间花费还是在于数据库对于数据的处理（也就是增删改查的时间），然后附加的有相似语句第一次的编译时间和连接到数据库的网络时间（此项通过explain analyse消除了误差）。

为了模拟真实应用的工作负载，从而更好地了解数据库在实际应用中的表现，接下来我们选择了Pgbench 来做测试。Pgbench 是postgres自带的数据库测试软件，可以模拟多用户并发操作来测试数据库的性能，测试数据库在不同负载下的吞吐量、响应时间等性能指标。

先在 powershell （注意不是psql）里切换到C:\Program Files\PostgreSQL\11\bin也就是postgres的地址。

```
cd C:\Program Files\PostgreSQL\11\bin
```

在这个工作目录下，我们可以输入

```
pgbench --help
```

来获取pgbench的指令模版（ pgbench [OPTION]... [DBNAME]）。以下是在本次project中主要使用的命令：

option	usage
-i	invokes initialization mode
-s(lower case) [number]	scaling factor (in intialization) report this scale factor in output(in benchmarking) (we set this as 10,which represents that pgbench has 10*100000 test data)
-c(lower case!)	number of concurrent database clients (default: 1)
-r	report average latency per command
-S(upper case)	perform SELECT-only transactions
-j [number]	number of threads (default: 1)
-t [number]	number of transactions each client runs (default: 10)

在实验中，主要改变的是-j -t和-c，也就是数据库服务器的线程，执行测试语句的次数以及并发查询的客户端数量。

然后执行初始化pgbench，并连接本地数据库“postgres”。

```
pgbench -U postgres -s 10 -i postgres
```

之后会弹出Password申请，输入数据库密码登录postgres用户即可。

```
C:\Program Files\PostgreSQL\11\bin>pgbench -U postgres -s 10 -i postgres
Password:
dropping old tables...
creating tables...
generating data...
100000 of 1000000 tuples (10%) done (elapsed 0.01 s, remaining 0.10 s)
200000 of 1000000 tuples (20%) done (elapsed 0.02 s, remaining 0.09 s)
300000 of 1000000 tuples (30%) done (elapsed 0.12 s, remaining 0.29 s)
400000 of 1000000 tuples (40%) done (elapsed 0.14 s, remaining 0.20 s)
500000 of 1000000 tuples (50%) done (elapsed 0.25 s, remaining 0.25 s)
600000 of 1000000 tuples (60%) done (elapsed 0.38 s, remaining 0.25 s)
700000 of 1000000 tuples (70%) done (elapsed 0.39 s, remaining 0.17 s)
800000 of 1000000 tuples (80%) done (elapsed 0.50 s, remaining 0.12 s)
900000 of 1000000 tuples (90%) done (elapsed 0.51 s, remaining 0.06 s)
1000000 of 1000000 tuples (100%) done (elapsed 0.62 s, remaining 0.00 s)
vacuuming...
creating primary keys...
done.
```

在pgbench初始化后，会生成四张表，pgbench_accounts，pgbench_branches，pgbench_history，pgbench_tellers，但我们主要关注的是pgbench_accounts，也就是pgbench存储数据的表。

在psql中，登录数据库用户信息，进而输入下述指令查看pgbench_accounts的结构。

```
Server [localhost]: 127.0.0.1
Database [postgres]: postgres
Port [5432]: 5432
Username [postgres]: postgres
用户 postgres 的口令:
psql (11.21)
```

```
pgbench=# \d+ pgbench_accounts
```

```
postgres=# \d+ pgbench_accounts
          数据表 "public.pgbench_accounts"
  栏位  |  类型  |  校对规则  |  可空的  |  预设  |  存储  |  统计目标  |  描述
-----+-----+-----+-----+-----+-----+-----+-----
 aid    | integer |              | not null |        | plain |              |
 bid    | integer |              |          |        | plain |              |
 abalance | integer |              |          |        | plain |              |
 filler | character(84) |              |          |        | extended |              |
索引:
 "pgbench_accounts_pkey" PRIMARY KEY, btree (aid)
选项: fillfactor=100
```

由类型可知pgbench_accounts也就是integer和char组成的表，基本满足我们的测试需求。

根据返回值发现，对于select测试，pgbench执行的是如下语句：

```
SELECT abalance FROM pgbench_accounts WHERE aid = :aid;
```

注：需要指出的是，在pgbench测试中，我们选择是simple query mode，也就是每次都需要重新编译，没有jit机制。

接下来开始正式实验.....

1、对照组（一个客户端，一个线程，50000次查询）

```
pgbench -U postgres -t 50000 -j 1 -r -S postgres
```

```
C:\Program Files\PostgreSQL\11\bin>pgbench -U postgres -t 50000 -j 1 -r -S postgres
Password:
starting vacuum...end.
transaction type: <builtin: select only>
scaling factor: 10
query mode: simple
number of clients: 1
number of threads: 1
number of transactions per client: 50000
number of transactions actually processed: 50000/50000
latency average = 0.050 ms
tps = 19929.076403 (including connections establishing)
tps = 20344.290497 (excluding connections establishing)
statement latencies in milliseconds:
    0.001  \set aid random(1, 100000 * :scale)
    0.048  SELECT abalance FROM pgbench_accounts WHERE aid = :aid;
```

结果为：单次事务（即执行一次语句）的时间是0.050ms，除去连接时间的TPS（每秒处理事务数量）是约20300次。

2、多线程（一个客户端，八个线程，50000次查询）

```
pgbench -c 1 -j 8 -U postgres -t 50000 -r -S postgres
```

```
C:\Program Files\PostgreSQL\11\bin>pgbench -c 1 -j 8 -U postgres -t 50000 -r -S postgres
Password:
starting vacuum...end.
transaction type: <builtin: select only>
scaling factor: 10
query mode: simple
number of clients: 1
number of threads: 8
number of transactions per client: 50000
number of transactions actually processed: 50000/50000
latency average = 0.050 ms
tps = 20120.539739 (including connections establishing)
tps = 20541.363527 (excluding connections establishing)
statement latencies in milliseconds:
    0.001  \set aid random(1, 100000 * :scale)
    0.048  SELECT abalance FROM pgbench_accounts WHERE aid = :aid;
```

TPS和单次事务时间都没有发生较大变化，有点违背常理。但是我们看到了，number of thread仍然是1。这说明一个客户端只能进行一个线程。

3、多线程2（八个客户端，八个线程&一个线程，50000次查询）

```
pgbench -U postgres -t 50000 -j 8 -c 8 -r -S postgres
```

```
C:\Program Files\PostgreSQL\11\bin>pgbench -U postgres -t 50000 -j 8 -c 8 -r -S postgres
Password:
starting vacuum...end.
transaction type: <builtin: select only>
scaling factor: 10
query mode: simple
number of clients: 8
number of threads: 8
number of transactions per client: 50000
number of transactions actually processed: 400000/400000
latency average = 0.095 ms
tps = 84286.646917 (including connections establishing)
tps = 87052.014724 (excluding connections establishing)
statement latencies in milliseconds:
    0.001  \set aid random(1, 100000 * :scale)
    0.089  SELECT abalance FROM pgbench_accounts WHERE aid = :aid;
```

```
pgbench -U postgres -t 50000 -j 1 -c 8 -r -S postgres
```

```
C:\Program Files\PostgreSQL\11\bin>pgbench -U postgres -t 50000 -j 1 -c 8 -r -S postgres
Password:
starting vacuum...end.
transaction type: <builtin: select only>
scaling factor: 10
query mode: simple
number of clients: 8
number of threads: 1
number of transactions per client: 50000
number of transactions actually processed: 400000/400000
latency average = 0.125 ms
tps = 64099.375570 (including connections establishing)
tps = 64615.491896 (excluding connections establishing)
statement latencies in milliseconds:
    0.000  \set aid random(1, 100000 * :scale)
    0.105  SELECT abalance FROM pgbench_accounts WHERE aid = :aid;
```

很明显，在控制变量之后，在线程数小于等于客户端数之后，线程数的增加对于TPS是由积极作用的。

4、多客户端（50个客户端&20个客户端&10个客户端，8个线程，50000次查询）

```
pgbench -U postgres -t 50000 -j 8 -c 20 -r -S postgres
```

```
C:\Program Files\PostgreSQL\11\bin>pgbench -U postgres -t 50000 -j 8 -c 20 -r -S postgres
Password:
starting vacuum...end.
transaction type: <builtin: select only>
scaling factor: 10
query mode: simple
number of clients: 20
number of threads: 8
number of transactions per client: 50000
number of transactions actually processed: 1000000/1000000
latency average = 0.134 ms
tps = 148887.530241 (including connections establishing)
tps = 152575.289474 (excluding connections establishing)
statement latencies in milliseconds:
    0.001  \set aid random(1, 100000 * :scale)
    0.119  SELECT abalance FROM pgbench_accounts WHERE aid = :aid;
```

```
pgbench -U postgres -t 50000 -j 8 -c 50 -r -S postgres
```

```
C:\Program Files\PostgreSQL\11\bin>pgbench -U postgres -t 50000 -j 8 -c 50 -r -S postgres
Password:
starting vacuum...end.
transaction type: <builtin: select only>
scaling factor: 10
query mode: simple
number of clients: 50
number of threads: 8
number of transactions per client: 50000
number of transactions actually processed: 2500000/2500000
latency average = 0.327 ms
tps = 152858.961512 (including connections establishing)
tps = 155005.629455 (excluding connections establishing)
statement latencies in milliseconds:
    0.001  \set aid random(1, 100000 * :scale)
    0.271  SELECT abalance FROM pgbench_accounts WHERE aid = :aid;
```

虽然20个客户端比10个客户端的速度快一点，但50个居然和20个速度差异不大。这可能是我们设计的数据库对高并发的请求没法很好的满足，但暂且不论。虽然客户端的数量增加可以增大TPS，但并不成线性关系，可能在客户端数量增多的情况下，该测试环境的CPU达到了性能上限。这也体现在每个select语句执行的时间随着客户端数目的增加而增加。

在做完这些对比实验后，我们可以汇总一下数据（执行次数均为50000）。

客户端数目	线程数目	TPS	语句单次执行时间 (ms)
1	1	20300	0.048
1	8	20500	0.048
8	8	87000	0.089
8	1	64600	0.105
50	8	152000	0.271
20	8	155000	0.119
10	8	97000	0.098

由上表可以得出结论，对于TPS影响最大的是客户端数目，但是在本次测试环境下，基本在20个达到了上限。而对语句执行时间影响较大的同样是客户端数目，但这次是负面影响，导致语句执行变慢。但这似乎相矛盾，为什么TPS越大反而执行速度越慢呢？我们想到，测试环境下CPU的计算核心只有8个，可能在8个客户端左右有最好的表现。

```
pgbench -U postgres -t 50000 -j 8 -c 8 -r -S postgres
```

```
C:\Program Files\PostgreSQL\11\bin>pgbench -U postgres -t 50000 -j 8 -c 8 -r -S postgres
Password:
starting vacuum...end.
transaction type: <builtin: select only>
scaling factor: 10
query mode: simple
number of clients: 8
number of threads: 8
number of transactions per client: 50000
number of transactions actually processed: 400000/400000
latency average = 0.097 ms
tps = 82410.468289 (including connections establishing)
tps = 85123.567978 (excluding connections establishing)
statement latencies in milliseconds:
    0.001  \set aid random(1, 100000 * :scale)
    0.091  SELECT abalance FROM pgbench_accounts WHERE aid = :aid;
```

TPS = 85123

latency avg = 0.091ms

这组数据在这两个上都表现得比较均衡，也就作为DBMS的代表去和文件读写比较。DBMS结果总结

数据源	执行操作	TPS	执行时间(ms)
pgbench	Select	85123	0.091

(二) 与文件读写系统的比较

在本部分中，讨论了DBMS在导入数据，select，insert和aggregate函数方面与文件读写系统的比较。对于每种语句，我们都是进行多次实验，取平均值。

1、Data import

对于DBMS，我们使用了copy方法直接导入数据。但是此语句无法用explain来计时，我们只能用自带的时间。

```
copy danmu from 'E:\Database_proj_1\data\danmu.csv' with csv header;
```

```
postgres.public> copy danmu from 'E:\Database_proj_1\data\danmu.csv' with csv header
[2023-11-04 19:20:01] 12,478,996 rows affected in 16 s 418 ms
```

平均结果是17s 149ms。

接下来使用java csvreader实现数据读入：

```
String filename = "E:\\Database_proj_1\\data\\danmu.csv";

int cnt = 0;
long start = System.currentTimeMillis();
try {
    csvReader = new CsvReader(new FileReader(filename));
    // csvWriter = new CsvWriter(filename);
    csvReader.readHeaders();
    while (csvReader.readRecord()) {
        bv[cnt] = csvReader.get(0);
        mid[cnt] = Long.parseLong(csvReader.get(1));
        time[cnt] = Float.parseFloat(csvReader.get(2));
        content[cnt] = csvReader.get(3);
        cnt++;
    }
    csvReader.close();
} catch (FileNotFoundException e) {
    throw new RuntimeException(e);
} catch (IOException e) {
    throw new RuntimeException(e);
}
long end = System.currentTimeMillis();
System.out.printf("Runtime of import data is: %dms and has %d rows\n", (end - start), cnt);
```

```
Runtime of import data is: 6106ms and has 12478996 rows
```

结果是6s 106ms。

很让人惊讶的是，居然文件读写导入数据的速度快了将近三倍。对导入DBMS的方法尝试了一些改进，比如关闭日志之类，但都没有显著的提升。猜测可能DBMS需要连接数据库，而java是本地操作。

2、Select

首先是没有限定条件的select，也就是选取一整列。避免时间太久，只选取前100000行。

```
--select experiment 1
explain analyse
select bv
from danmu
limit 100000;
```

实验次数	运行时间 (ms)
1	14.431
2	14.943
3	14.410
4	14.111

实验次数	运行时间 (ms)
5	14.294
Avg	14.437ms

接下来是文件读写：

```
private static List<String> select() {
    List<String> ans = new ArrayList<String>();
    long start = System.currentTimeMillis();
    for (int i = 0; i < 100000; i++) {
        ans.add(content[i]);
    }
    long end = System.currentTimeMillis();
    System.out.printf("Runtime of select is %d ms\n", (end - start) );
    return ans;
}
```

Runtime of select is 2 ms

这次居然又是文件读写胜出了。但这是最基础的功能，我们接下来看带有限定条件的select。

```
--select experiment 2
explain analyse
select Content
from danmu
where Content like '%原神%';
```

实验次数	运行时间 (ms)
1	721.323
2	724.669
3	723.987
4	727.279
5	705.752
Avg	720.602

同理，接下来是文件读写的表现：


```
private static List<String> select() {
    long start = System.currentTimeMillis();
    int cnt = 0;
    List<String> ans = new ArrayList<String>();
    for (int i = 0; i < 12478997; i++) {
        if (content[i] != null && content[i].contains("原神")) {
            ans.add(content[i]);
            cnt++;
        }
    }
    long end = System.currentTimeMillis();
    System.out.printf("Runtime of select is %d ms and %d rows\n", (end - start), cnt);
    return ans;
}
```

Runtime of select is 176 ms and 10146 rows

在有了限定条件后，还是文件读写比较快。

在danmu数据下，文件读写很成功，我们让他对pgbench生成的随机数据进行select测试。通过datagrip的导出功能，我们将pgbench_accounts导出为一个一百万行的csv文件。

```
private static List<String> select() {
    Random random = new Random();
    int _aid = random.nextInt( bound: 1000000);
    long start = System.nanoTime();
    int cnt = 0;
    List<String> ans = new ArrayList<String>();
    for (int i = 0; i < 1000000; i++) {
        //conditional select
        // if (content[i] != null && content[i].contains("原神")) {
        //     ans.add(content[i]);
        //     cnt++;
        // }
        if (aid[i] == _aid)
            ans.add(String.valueOf(ambalance[i]));
        cnt++;
    }
    long end = System.nanoTime();
    System.out.printf("Runtime of select is %d ns and %d rows\n", (end - start), cnt);
    return ans;
}
```

Runtime of select is 60852000 ns and 1000000 rows

经过计算，一次select操作是60.85ns 也就是0.00006085ms，所以TPS达到了一千六百万级别，也就是比pgbench算出来大了快2000倍。

在本次实验中，文件读写在select上比DBMS更胜一筹。

3. Insert

DBMS表现如下：


```
explain analyse
insert into danmu (bv, mid, time, content)
values ('BVhfkhaj', 123456, 0.1, 'nothing');
```

实验次数	运行时间 (ms)
1	0.015
2	0.023
3	0.017
4	0.014
5	0.014
Avg	0.016

文件读写表现如下:

```
BufferedWriter bufferedWriter = new BufferedWriter(new FileWriter(filename, append: true));
csvWriter = new CsvWriter(bufferedWriter, c: ',');

private static void insert() {
    long start = System.nanoTime();
    String[] comment = {"BVhfkhaj", "123456", "0.1", "nothing"};
    try {
        for (int i = 0; i < 1000; i++) {
            csvWriter.writeRecord(comment);
        }
    } catch (IOException e) {
        throw new RuntimeException(e);
    }
    long end = System.nanoTime();
    csvWriter.close();
    System.out.printf("Runtime of insert is %d microsecond\n", (end - start) / 1000);
}
```

在这次文件读写中, 我们因为单次插入浮动较大, 选择了插入1000次然后除以一千的操作。

实验次数	运行时间 (ms)
1	0.003
2	0.002
3	0.002
4	0.002
5	0.002
Avg	0.002

又是文件读写取得了很大优势。

接下来我们看 aggregate 函数

4、Aggregate

根据前两个实验的分析，我们发现，在仅仅对数据进行遍历的操作上，似乎由于网络连接等因素的影响，文件读写似乎比较快。毕竟文件读写只需要进行磁盘擦除和写入的操作，而数据库还有优化和连接服务器，创建查询计划的时间。接下来，我们选择postgres中一个比较特殊的group by函数来测试。

```
--aggregate function
explain analyse
select BV
from danmu
group by BV;
```

实验次数	运行时间 (ms)
1	405.559
2	380.788
3	382.945
4	378.399
5	383.715
Avg	386.28

以下是文件读写的表现：

```
public static String[] group_by(String[] array) {
    long start = System.currentTimeMillis();

    HashMap<String, Integer> hashMap = new HashMap<>();

    // 遍历数组并将元素插入HashMap
    for (String element : array) {
        hashMap.put(element, 0);
    }

    // 获取去重后的元素
    HashSet<String> uniqueElements = new HashSet<>(hashMap.keySet());
    long end = System.currentTimeMillis();
    System.out.printf("Runtime of group by is %dms and %d rows", end - start, uniqueElements.size());
    // 将去重后的元素转换为数组并返回
    return uniqueElements.toArray(new String[0]);
}
```

Runtime of group by is 216ms and 7866 rows

文件读写在聚合函数上也比DBMS快。

总结：综上所述，似乎文件读写在各个方面都比DBMS要快。但是，我们思考了几点导致这样似乎不符合常理的结论。

- 1. 数据库存储数据的数据结构与我们不同。

在我们实现的java程序中，我们直接使用了数组去存储输入的数据，但在实际应用中，postgres使用的不是线性数据结构，是一个个堆文件，在访问它们的过程中，是需要用到对应的键值的，这可能会需要更多的时间。

2. 数据库对于select以及group by的算法不同

在上一点中，我们提到数据结构的不同，所以数据库实现算法里，可能需要嵌套循环来查询数据。在group by的算法里，我们只实现了去重的操作，在数据库中，对于该条记录的其他部分也需要聚合。

3. 数据库是对服务器的远程操作，文件读写是在本地的操作。

在一定范围内，链接本机数据库也是需要时间的，但我们通过explain analyse 已经基本去除了这部分误差。同样，本地读写文件，不需要更新日志或者通过中间级的内存，也节约了一部分时间。

Postgres高并发探究

在对postgres高并发探究时，虽然了解了例如postgres是如何利用MVCC机制处理高并发的高深知识，但很遗憾，没能利用底层机制来提高数据库的并发表现。所以，我们只能为了性能牺牲一点数据的安全性，通过调整postgres运行的参数来提高高并发下的表现，同样，测试在上述测试环境中进行，测试软件为pgbench。

根据一篇讲述postgres参数调整的文章（[PostgreSQL 参数调整(性能优化) - VicLW - 博客园 (cnblogs.com)](<https://www.cnblogs.com/VicLiu/p/11854730.html>))，我们尝试了一下几个参数的调整。

```
--optimized state
alter system set full_page_writes = off;
```

本参数是防止写入数据时崩溃产生较大的影响，类似于增加检查点，自然会增加计算消耗。

```
alter system set wal_buffers = 4096;
```

PostgreSQL将其WAL（[预写日志](#)）记录写入缓冲区，然后将这些缓冲区刷新到磁盘。由wal_buffers定义的缓冲区的默认大小为16MB，如果有大量并发连接的话，则设置为一个较高的值可以提供更好的性能。

```
alter system set synchronous_commit = off;
```

此参数的作用为在向客户端返回成功状态之前，强制提交等待WAL被写入磁盘。这是性能和可靠性之间的权衡。如果应用程序被设计为性能比可靠性更重要，那么关闭 synchronous_commit。这意味着成功状态与保证写入磁盘之间会存在时间差。在服务器崩溃的情况下，即使客户端在提交时收到成功消息，数据也可能丢失。

```
alter system set shared_buffers = 524288;
```

PostgreSQL既使用自身的缓冲区，也使用内核缓冲IO。这意味着数据会在内存中存储两次，首先是存入PostgreSQL缓冲区，然后是内核缓冲区。这被称为双重缓冲区处理。对大多数操作系统来说，这个参数是最有效的用于调优的参数。此参数的作用是设PostgreSQL中用于缓存的专用内存量。

shared_buffers 的默认值设置得非常低，因为某些机器和操作系统不支持使用更高的值。但在大多数现代设备中，通常需要增大此参数的值才能获得最佳性能。

建议的设置值为机器总内存大小的25%，但是也可以根据实际情况尝试设置更低和更高的值。实际值取决于机器的具体配置和工作的数据量大小。举个例子，如果工作数据集可以很容易地放入内存中，那么可以增加shared_buffers的值来包含整个数据库，以便整个工作数据集可以保留在缓存中。

在生产环境中，将 `shared_buffers` 设置为较大的值通常可以提供非常好的性能，但应当时刻注意找到平衡点。

```
alter system set work_mem = 8192;
```

此配置用于复合排序。内存中的排序比溢出到磁盘的排序快得多，设置非常高的值可能会导致部署环境出现内存瓶颈，因为此参数是按用户排序操作。如果有多个用户尝试执行排序操作，则系统将为所有用户分配大小为 `work_mem * 总排序操作数` 的空间。全局设置此参数可能会导致内存使用率过高，因此强烈建议在会话级别修改此参数值。默认值为4MB。

```
alter system set maintenance_work_mem = 131072;
```

`maintenance_work_mem` 是用于维护任务的内存设置。默认值为64MB。设置较大的值对于 `VACUUM`，`RESTORE`，`CREATE INDEX`，`ADD FOREIGN KEY` 和 `ALTER TABLE` 等操作的性能提升效果显著。

除此之外，我们还更改了几个无法在终端修改的参数，也就是在 `postgres` 安装目录下的 `postgres.conf` 文件中手动更改。

[PostgreSQL并行查询相关配置参数 - 知乎\(zhihu.com\)](https://www.zhihu.com/question/20131111)

根据这篇文章，我们测试了这几个参数的效果如何。

```
\# - Asynchronous Behavior -  
\#effective_io_concurrency = 0      # 1-1000; 0 disables prefetching
```

根据前人的测试结果，对于本机测试环境下的SSD来说，关闭硬盘预读得不偿失，故没有调试。

最大并行进程数，本机CPU可以有20个进程，我们可以开启到16，以便后续参数的调整。

```
\#max_parallel_maintenance_workers = 2 # taken from max_parallel_workers  
\#max_parallel_workers_per_gather = 8  # taken from max_parallel_workers  
\#max_parallel_workers = 2            # maximum number of max_worker_processes that  
    \# can be used in parallel operations
```

在这三个参数中，我们都把他们调大了两倍，来查看结果，也就是：

```
\#max_parallel_maintenance_workers = 4 # taken from max_parallel_workers  
\#max_parallel_workers_per_gather = 16 # taken from max_parallel_workers  
\#max_parallel_workers = 4            # maximum number of max_worker_processes that  
    \# can be used in parallel operations
```

注意，在配置文件 (`postgres.conf`) 中，需取消注释，并重启数据库服务器。

接下来看在20个客户端查询下的表现对比。注意，这次对比不只是 `select-only mode`，对于 `update` 和 `insert` 都做了实验。

```
pgbench -c 20 -j 16 -U postgres -t 1000 -r postgres
```

原始表现如下图

```
C:\Program Files\PostgreSQL\11\bin>pgbench -c 20 -j 16 -U postgres -t 1000 -r postgres
Password:
starting vacuum...end.
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 1
query mode: simple
number of clients: 20
number of threads: 16
number of transactions per client: 1000
number of transactions actually processed: 20000/20000
latency average = 4.068 ms
tps = 4916.690127 (including connections establishing)
tps = 5254.463127 (excluding connections establishing)
statement latencies in milliseconds:
  0.002  \set aid random(1, 100000 * :scale)
  0.000  \set bid random(1, 1 * :scale)
  0.000  \set tid random(1, 10 * :scale)
  0.000  \set delta random(-5000, 5000)
  0.045  BEGIN;
  0.100  UPDATE pgbench_accounts SET abalance = abalance + :delta WHERE aid = :aid;
  0.079  SELECT abalance FROM pgbench_accounts WHERE aid = :aid;
  1.916  UPDATE pgbench_tellers SET tbalance = tbalance + :delta WHERE tid = :tid;
  1.105  UPDATE pgbench_branches SET bbalance = bbalance + :delta WHERE bid = :bid;
  0.069  INSERT INTO pgbench_history (tid, bid, aid, delta, mtime) VALUES (:tid, :bid, :aid, :delta, CURRENT_TIME
STAMP);
  0.111  END;
```

在改变运行参数后：

```
C:\Program Files\PostgreSQL\11\bin>pgbench -c 20 -j 16 -U postgres -t 1000 -r postgres
Password:
starting vacuum...end.
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 1
query mode: simple
number of clients: 20
number of threads: 16
number of transactions per client: 1000
number of transactions actually processed: 20000/20000
latency average = 3.107 ms
tps = 6437.805472 (including connections establishing)
tps = 7089.951771 (excluding connections establishing)
statement latencies in milliseconds:
  0.002  \set aid random(1, 100000 * :scale)
  0.000  \set bid random(1, 1 * :scale)
  0.000  \set tid random(1, 10 * :scale)
  0.000  \set delta random(-5000, 5000)
  0.047  BEGIN;
  0.104  UPDATE pgbench_accounts SET abalance = abalance + :delta WHERE aid = :aid;
  0.085  SELECT abalance FROM pgbench_accounts WHERE aid = :aid;
  1.287  UPDATE pgbench_tellers SET tbalance = tbalance + :delta WHERE tid = :tid;
  0.790  UPDATE pgbench_branches SET bbalance = bbalance + :delta WHERE bid = :bid;
  0.071  INSERT INTO pgbench_history (tid, bid, aid, delta, mtime) VALUES (:tid, :bid, :aid, :delta, CURRENT_TIME
STAMP);
  0.056  END;
```

有很明显的提升，观察个部分语句的运行时间变化，发现主要是update和end语句变化较大。根据我们分析，主要是

wal_buffers work_mem maintenance_work_mem

这三个缓存变化导致的。我们可以尝试一下，在更大客户端规模下，这些参数变化的提升会不会更大。

```
pgbench -c 50 -j 16 -U postgres -t 1000 -r postgres
```

原始表现

```

C:\Program Files\PostgreSQL\11\bin>pgbench -c 50 -j 16 -U postgres -t 1000 -r postgres
Password:
starting vacuum...end.
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 1
query mode: simple
number of clients: 50
number of threads: 16
number of transactions per client: 1000
number of transactions actually processed: 50000/50000
latency average = 11.830 ms
tps = 4226.637725 (including connections establishing)
tps = 4381.568747 (excluding connections establishing)
statement latencies in milliseconds:
    0.002  \set aid random(1, 100000 * :scale)
    0.000  \set bid random(1, 1 * :scale)
    0.000  \set tid random(1, 10 * :scale)
    0.000  \set delta random(-5000, 5000)
    0.045  BEGIN;
    0.100  UPDATE pgbench_accounts SET abalance = abalance + :delta WHERE aid = :aid;
    0.078  SELECT abalance FROM pgbench_accounts WHERE aid = :aid;
    8.188  UPDATE pgbench_tellers SET tbalance = tbalance + :delta WHERE tid = :tid;
    1.598  UPDATE pgbench_branches SET bbalance = bbalance + :delta WHERE bid = :bid;
    0.070  INSERT INTO pgbench_history (tid, bid, aid, delta, mtime) VALUES (:tid, :bid, :aid, :delta, CURRENT_TIME
STAMP);
    0.115  END;

```

优化后的表现:

```

C:\Program Files\PostgreSQL\11\bin>pgbench -c 50 -j 16 -U postgres -t 1000 -r postgres
Password:
starting vacuum...end.
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 1
query mode: simple
number of clients: 50
number of threads: 16
number of transactions per client: 1000
number of transactions actually processed: 50000/50000
latency average = 9.515 ms
tps = 5254.904210 (including connections establishing)
tps = 5444.980539 (excluding connections establishing)
statement latencies in milliseconds:
    0.002  \set aid random(1, 100000 * :scale)
    0.000  \set bid random(1, 1 * :scale)
    0.000  \set tid random(1, 10 * :scale)
    0.000  \set delta random(-5000, 5000)
    0.051  BEGIN;
    0.108  UPDATE pgbench_accounts SET abalance = abalance + :delta WHERE aid = :aid;
    0.088  SELECT abalance FROM pgbench_accounts WHERE aid = :aid;
    6.408  UPDATE pgbench_tellers SET tbalance = tbalance + :delta WHERE tid = :tid;
    1.302  UPDATE pgbench_branches SET bbalance = bbalance + :delta WHERE bid = :bid;
    0.074  INSERT INTO pgbench_history (tid, bid, aid, delta, mtime) VALUES (:tid, :bid, :aid, :delta, CURRENT_TIME
STAMP);
    0.066  END;

```

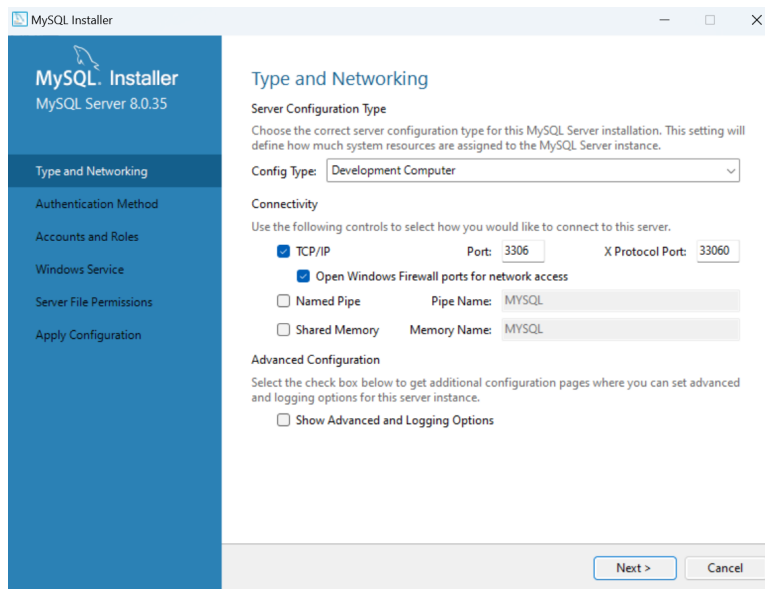
在20个客户端的条件下，提升了40%，但在50个客户端的情况下，提升了25%。说明这些参数的调整也许是比较表面的，需要提升数据库的结构来根本性地提升高并发的表现。这些参数只是把本测试环境下过剩的性能利用了，也就是对于不同的运行环境，需要多次调试来确定这些参数的最佳取值。但postgres使用的MVCC机制已经很好的处理小规模的并发事务处理，在现在这几个阶段，这已经使我们能拿出的最好答案了:。

不同的数据库软件性能比较

我们比较了Mysql，MariaDB和postgresql的各方面性能差距。由于没有找到好用的windows压测软件，所以我们只能用datagrip简单测试他们对于同一个测试文件insert和select的执行效果了。

其他数据库软件安装（以MySQL为例）

1、驱动下载与MySQL安装



2、环境变量配置：与前文 PostgreSQL 相同，把MySQL的 `bin` 目录包含在系统的 PATH 环境变量中；

```
C:\Program Files\PostgreSQL\11\bin
C:\Program Files\PostgreSQL\11\lib
D:\Program Files\MySQL\MySQL Server 8.0\bin
```

完成之后打开 `Powershell`，运行验证

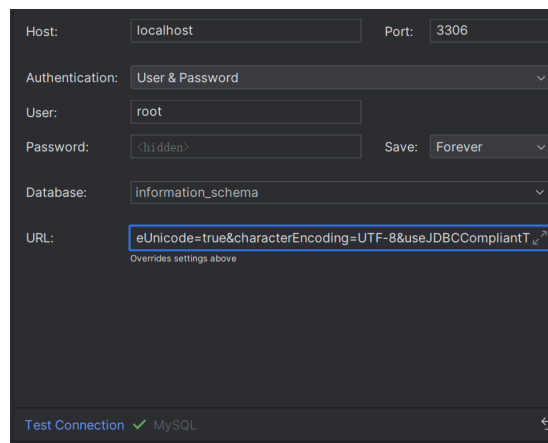
```
PS C:\Windows\system32> mysql -uroot -p
Enter password: *****
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 11
Server version: 8.0.35 MySQL Community Server - GPL

Copyright (c) 2000, 2023, Oracle and/or its affiliates.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.
```

在Datagrip中连接到MySQL数据库



我们先用以下语句生成一个较大的csv文件“test_data”，然后将它分别导入三个数据库。

```
CREATE TABLE test_data
(
  id SERIAL PRIMARY KEY,
  value INTEGER
);
INSERT INTO test_data (value)
SELECT generate_series(1, 10000000);
```

表结构如下：

```
postgres=# \d test_data
                        数据表 "public.test_data"
  栏位  |  类型  |  校对规则  |  可空的  |  预设
-----+-----+-----+-----+-----
 id    | integer |             | not null | nextval('test_data_id_seq'::regclass)
 value | integer |             |          |
索引:
    "test_data_pkey" PRIMARY KEY, btree (id)
```

处于公平起见，我们还是选择五次的平均值，毕竟对于postgres来说，第一次需要编译，而之后就不用，但是mysql和mariadb似乎没有这样的机制。

首先是select操作：

```
Explain analyse
select *
from test_data
where value>1000
Limit 100000;
```

运行时间 (ms) /数据库名	Postgre	Mysql	Mariadb
1	11.65	20.5	43
2	11.365	20.4	41
3	11.041	20.4	38
4	11.621	20.6	29
5	11.648	20.5	64
avg	11.45	20.48	43

分析：Mariadb时间较长可能是没有去除数据库连接时间。

随后我进行了一些稍复杂的查询。

```
select count(*)
from test_data
where value>1000
Limit 500;
```

运行时间 (ms) /数据库名	Postgre	Mysql	Mariadb
1	1112	2767	1994
2	1115	2588	1720
3	1096	2464	1763
4	1011	2617	1783
5	1127	2612	1788

运行时间 (ms) /数据库名	Postgre	Mysql	Mariadb
avg	1092.2	2609.6	1809.6

在聚合函数方面，postgre有很大的优势。而且经过测试，在max和min函数上也有极大的优势。

接下来是对insert语句的对比。

我们插入10000行数据，id是自增的，而value都是1-100 重复五次。由于插入语句过长，故不在报告中展示。

运行时间 (ms) /数据库名	Postgre	Mysql	Mariadb
1	11.313	63	48
2	11.812	65	43
3	10.429	79	31
4	10.848	79	28
5	10.627	70	29
avg	11.001	71.2	35.8

在这次实验中，postgres取得了决定性的优势，即便是算上连接数据库的时间，也是优于其余两者的。

总结

虽然在对比中，postgres都比mysql和mariadb强，但也可能是我对其他两者的优化不到位，并且对比的项目也比较基础，没有对他们的强项（比如mysql的高并发）对比。也受限于windows的测试环境，没有用上压测软件，都是用服务器自带的日志来估测执行时间。

但我们也能看到，各种数据库都是有优缺点的。比如，我们发现，mariadb的执行时一次比一次快。简单探究了一下，由于我们都是执行相同的语句多次，mariadb可能有对结果和查询索引的缓存机制，所以在第一次查询过后，会加速后续的查询。同样的。我们发现，postgres对自增的id查询最大值时，有数量级上的优势，我们推测可能它的查询算法不再是不同的索引查询，而是直接去找到末尾的最大值。

在使用不同数据库时，我们也发现了不同的优化策略。比如在mysql查询时，我们使用“*”和字段名来查询相同内容的执行时间是不同的，但这样的优化对于postgres和mariadb没有太大作用。所以，我们要根据存储的数据类型和主要的查询需求来确定使用的数据库。