CS307Project

12210401 龚浚哲　　12210403 王赵融杰

November 2023

1　任务分工与主要贡献

	成员	贡献比
成员及贡献比：	龚浚哲	50%
	王赵融杰	50%

具体分工：

- 数据库设计及建表设计：龚浚哲 & 王赵融杰
- E-R 图绘制：龚浚哲
- 利用 SQL 语句建表：王赵融杰
- 导入数据及优化分析：王赵融杰
- 对比分析 DBMS 和文件 I/O：龚浚哲 & 王赵融杰
- 分析数据库高并发及对比 Mysql 和 PostgreSQL：王赵融杰
- Task4 其他高级部分分析对比：龚浚哲

2 E-R 图

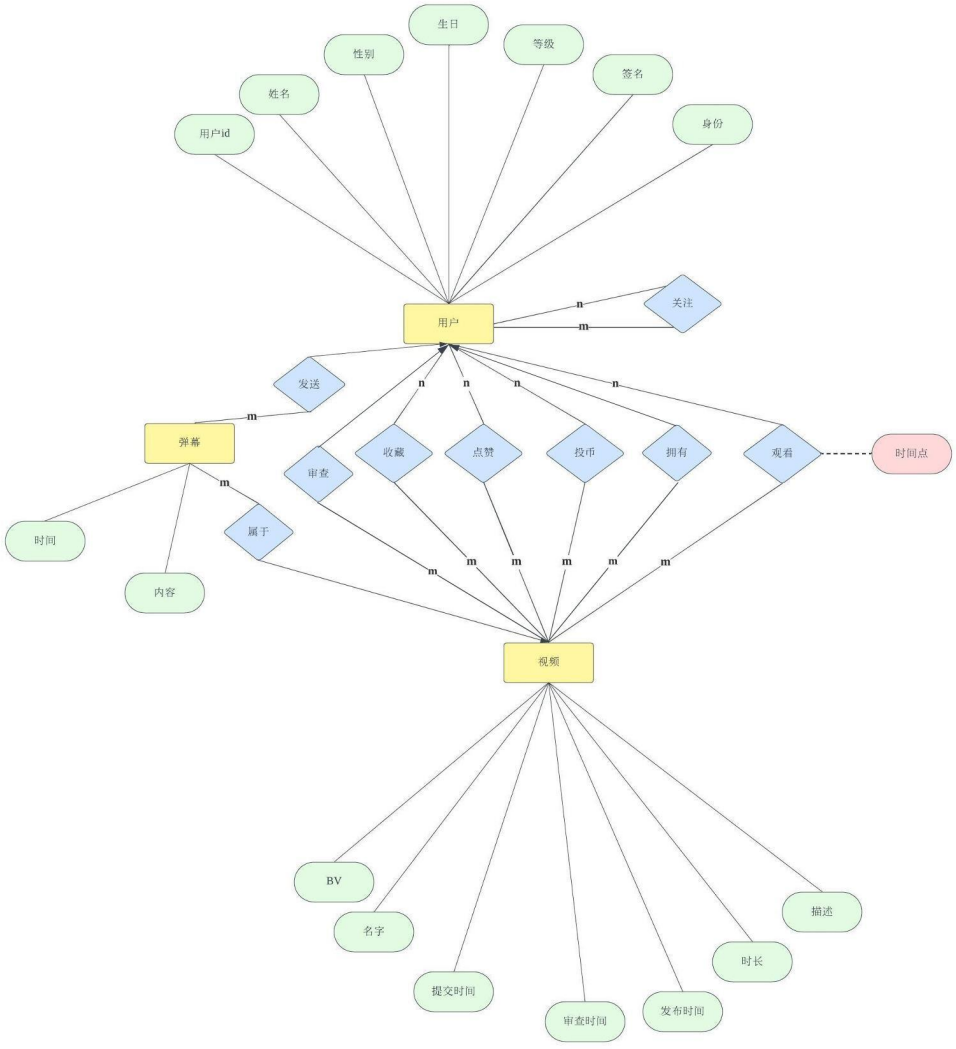

图 1: E-R 图

我们通过一个在线协作白板应用程序 Lucidspark 来制作 E-R 图。图中的矩形框代表实体集，菱形代表关系，椭圆形代表属性，m 和箭头的组合代表一对多，n m 的组合代表多对多。

3 数据库设计

3.1 数据库设计

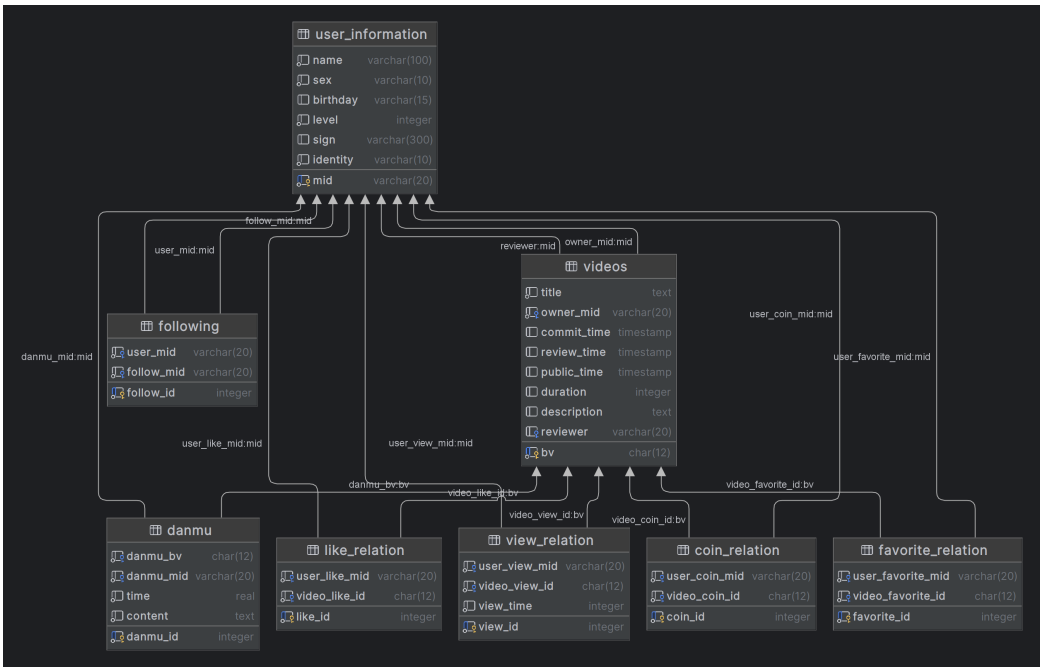

图 2: 数据库可视化

3.2 数据库描述

根据文档信息和分析，我们一共设计了 8 个表格:

- **User_information**: 用户信息表，存放用户除 following 以外的所有信息，并设置 `Mid` 为主键，每一列与 csv 文件中除了 following 的每一列对应。
- **Following**: 关注信息表，每一行代表一组关注和被关注，列 1 为自增的主键用来唯一标识每一组关注信息，列二为用户的 `Mid`，列三为被关注者的 `Mid`。
- **Videos**: 视频表，存放每一个视频的信息，`BV` 为主键，不包括点赞，投币，喜爱，观看时间等信息。

- **关系表**: `like`, `coin`, `favorite`, `view` 分别构建了四个表, 每个表分别对应列为视频 id，和进行操作的用户 `Mid`，当然，在 `view_relation` 表中还有观看时间，同时我们将每个表的第一列设置为主键，为自增的 id，我们认为通过这种关系表的方式可以更好地处理多对多的关系。

- **danmu**: 弹幕表，存放弹幕的各项信息，第一列设置成自增的 id，且为主键来作为唯一标识，其余列与 csv 文件中一一对应。

当然，在所有上述表格中出现的用户的 `Mid` 和视频的 `BV` 都被设计成外键关联到视频表和用户信息表，以确保我们能正确表示数据与数据，表与表之间的关系并进行处理和分析。

4 数据导入

4.1 Java 脚本介绍

1. **Step 1:** 首先，我们下载 PostgreSQL 和 commons-csv 库对应的 Jar 包（commons-csv 为稍后我们会用到的对 csv 文件进行操作的第三方库），并将它们添加到项目的 libraries 中。

2. **Step 2:** 先定义好 JDBC（Java Database Connectivity）连接字符串（`jdbc:postgresql://localhost:5432/postgres`），包括数据库类型，ip 地址和端口，数据库名称；用户名和密码以及文件地址（绝对路径）。

3. **Step 3:** 建立数据库连接，并打开指向 CSV 文件的阅读器以及用于解析 CSV 内容的 CSV 解析器。

4. **Step 4:** 准备好需要的 SQL 语句，以向数据库中插入用户信息为例：”INSERT INTO user_information (Mid, Name, Sex, Birthday, Level, Sign, Identity) VALUES (?, ?, ?, ?, ?, ?, ?)”

5. **Step 5:** 逐行遍历 csv 文件，对于每一行进行操作。利用 `record.get(i)` 获取这一行中我们需要的信息。在获取完信息后，利用 `addBatch` 将单个命令添加到批处理。

```
for (CSVRecord record : csvParser) {
    String column1 = record.get(0);
    // ...
    statement.setString(1, column1);
    // ...
    statement.addBatch();
}
```

6. **Step 6:** 当批处理操作积累到 300 个时我们一次性提交 SQL 命令到数据库, 最后提交事务，保存所有更改。

注意事项及先决条件：

- 给项目准备好必要的 jar 包。
- 在运行代码时，根据自身数据库的实际情况和导入文件的绝对路径来修改适当位置代码。
- 在进行文件导入前，要先关闭自动提交，不然批处理可能会失效。
- 利用 try-catch 模块进行异常处理。
- 由于文件过大，建议调整 JVM 最大堆内存到 10G。

在实际运行时，需要在 Main 类里面新建一个对象并调用该类中的方法，根据不同的表格我们设计了几个不同的类来存放方法。

4.2 导入结果

表名	行数
$user\_information$	37881
$danmu$	12478996
$videos$	7865
$following$	5996651
$like\_relation$	86765813
$coin\_relation$	80579385
$favorite\_relation$	79189760
$view\_relation$	163997974

4.3 测试环境介绍

高级部分分析和研究基于以下测试环境进行：

- **处理器**：12th Gen Intel(R) Core(TM) i7-12700H @ 2.30 GHz
- **内存 (RAM)**：16.0 GB (15.7 GB 可用)
- **设备 ID**：3943F5C8-9474-4A9A-B103-38492291162D
- **产品 ID**：00342-30567-87083-AAOEM
- **系统类型**：64 位操作系统, 基于 x64 的处理器
- **固态硬盘 (SSD)**：1TB
- **内存容量**：32GB
- **显卡**：RTX3060

Windows 规格：

- 版本：Windows 11 家庭中文版
- 版本号：22H2

- 安装日期：2023/2/26
- 操作系统版本：22621.2428

软件配置

- IntelliJ IDEA Community Edition 2022.2.1
- Programming Language: Java
- postgresql-42.2.5.jar
- 第三方库：commons-csv-1.10.0
- jdk-17.0.2
- DataGrip 2023.2.1
- PostgreSQL 15.4, compiled by Visual C++ build 1914, 64-bit

在复制我们实验时，需要注意我们所使用的程序语言，导入包的版本，数据库类型和版本，同时还需要知晓处理器型号，RAM 以及操作系统。

4.4 比较不同导入方法

在我们的项目中，测试了三种导入方法并比较它们的性能:

- **Java 脚本导入**：利用 Java 编写脚本连接数据库，利用 SQL 语句中的 `INSERT` 进行导入。
- **Datagrip 自带可视化导入**:在 Datagrip 中选择表格,右键选择 `IMPORT` 进行导入。
- **COPY 指令**：利用 PostgreSQL 中的 `COPY` 指令选择表和文件进行导入。

我们对于导入 $user\_information$ 的耗时进行比较，结果如下：

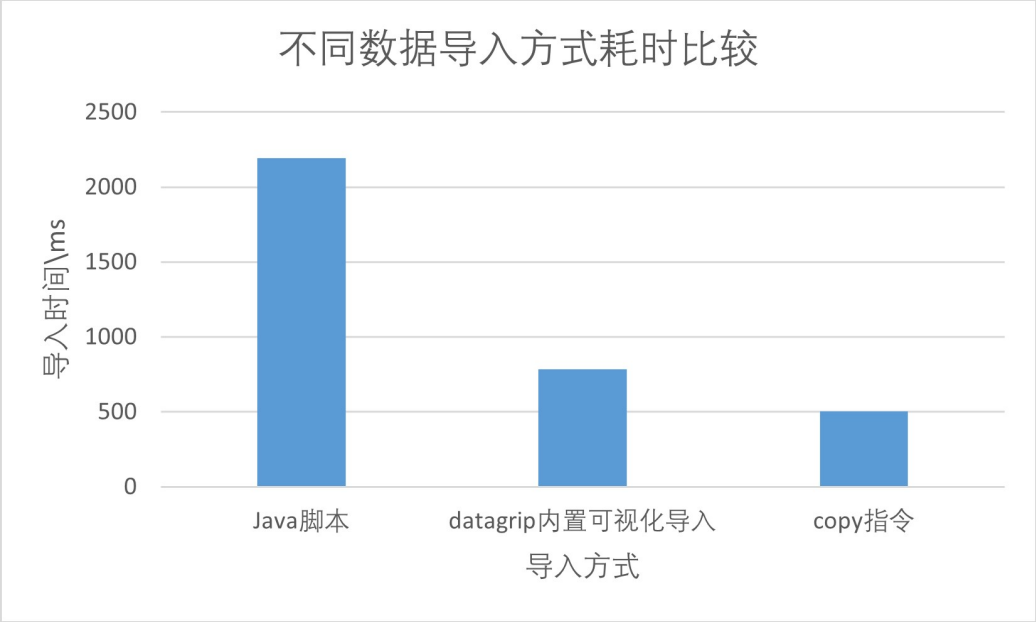

图 3: 导入耗时比较

在我们的测试中，发现 $Java$ 脚本导入最慢，可视化较快，$COPY$ 指令最快。

利用 $Java$ 脚本导入数据在多个方面会带来额外的时间开销，由于 $Java$ 程序运行在 JVM（Java 虚拟机）上，JVM 本身的启动和运行有一定的开销。同时，$Java$ 程序使用 $JDBC$（Java Database Connectivity）驱动来与数据库交互。$JDBC$ 本身有一定的开销，如创建连接、处理 SQL 语句、结果集映射等，这些都会消耗时间。

$DataGrip$ 在导入数据时则会使用批处理技术，将多条 $INSERT$ 语句合并为一次操作，减少网络往返和数据库事务的开销。对内存和系统资源的管理也会更加优化

而 $COPY$ 指令直接与数据库的底层引擎交互，避开了 SQL 解析和常规 DML（数据操纵语言）操作的开销。此外，它还能够高效地处理原生格式数据，它可以迅速地将数据文件中的内容加载到数据库表中，减少了转换和处理数据的开销。同时，通过利用并行处理，相较于没有设置多线程的 $Java$ 操作，能显著提高性能。而且 $COPY$ 操作时会将数据从服务器本地

文件直接导入数据库，可以避免网络传输的开销，这在大量数据处理中尤其有效。

4.5 导入速度优化

在该测试中，我们以导入 danmu.csv 到 danmu 表中为例进行速度测试和优化，总行数为 12,478,996。

在编写脚本中，我们发现有多个方面会影响速度，但在第一次编写脚本时我们已经考虑到了一些可能会影响速度的地方并避免了它们，如：实现准备 SQL 语句，关闭自动提交，设置批处理，预编译 SQL 语句等等，所以在这部分我们将不会对它们进行讨论。

首先，我们来探究不同批处理次数对速度影响。

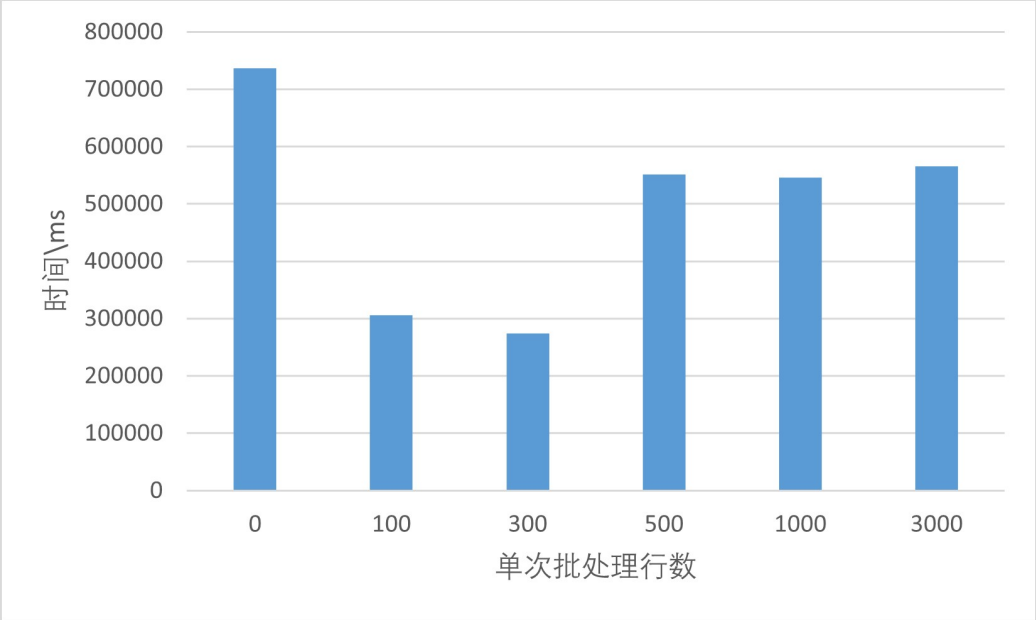

图 4: 单次批处理处理的语句个数对导入速度的影响

开启批处理后，可以将多个语句合并成一个事务，减少事务开销。同时也可以减少网络往返次数，以此来减少导入时间，但是，如果批处理的语句过多，这个处理事务会占用大量的磁盘内存，影响处理性能。因此，我们在实际情况中需要最佳批处理次数。

对于导入弹幕而言，我们测试了不同批处理的性能，发现批处理一次处理 300 条语句时性能较好，当然，我们还测试了一次处理较多语句时的情况，发现达到 500 后再增加对整体速度影响不大。

在下表中，我们展示了批处理大小对速度的优化程度，基准为 1000 条做批处理时的时间。(最开始进行导入时设置的批处理大小)

单次处理的 SQL 语句数	优化程度
0	-35%
100	43%
300	50%
500	-1%
3000	-4%

在对批处理次数进行调整后，我们接下来采取了多线程的方式来加快导入。在开启多个线程时，导入速度有明显提升，我们比较了不同线程的导入速度，结果如下：

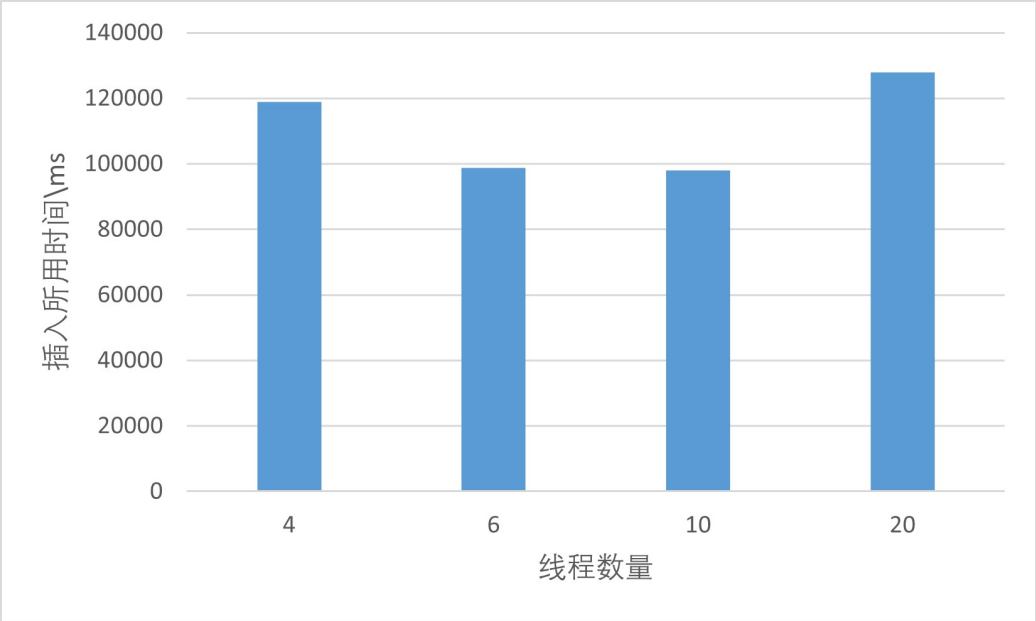

图 5: 不同线程数对导入速度的影响

多线程意味着我们可以同时操作更多的 SQL 语句，处理更多的事物，执行多个导入操作。因此开启多线程时导入时间会明显减少。然而，受到电脑性能的限制，开启过多线程时会消耗过多的内存资源。同时受到磁盘读写速度限制，过多的线程会导致磁盘读写出现瓶颈，反而速度变慢。如果并发的线程数超过数据库能处理的并发数大小，也会导致导入速度下降。根据测试，我们发现在线程为 6 时效果最好，继续增加线程对速度的增加效果不够显著。

以下为我们最终的优化结果：

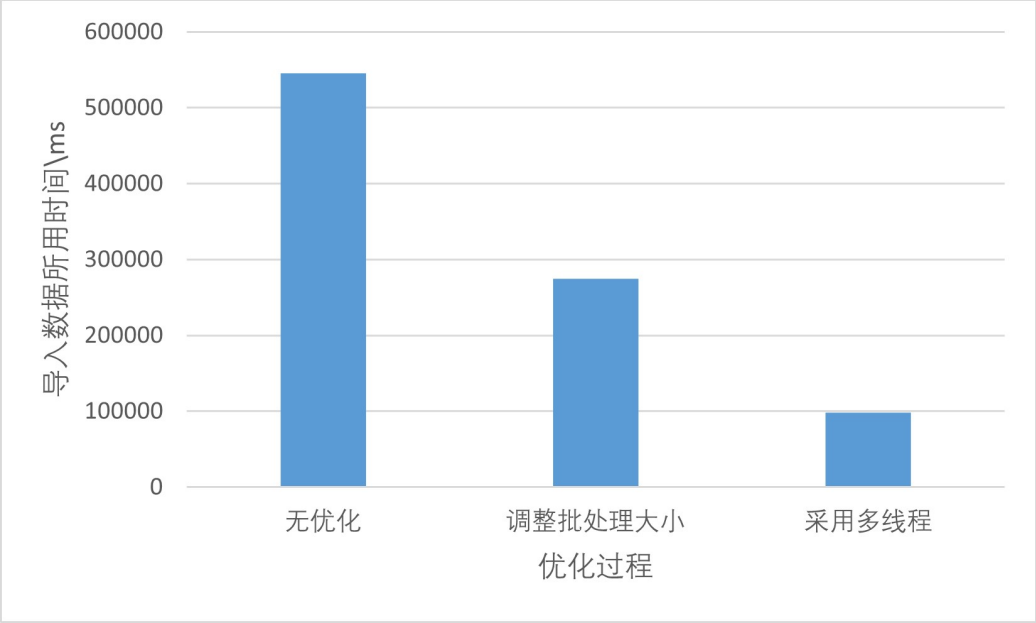

图 6: 优化结果

最终，我们实现了在 99 秒内导入一千多万条数据，所用时间是我们一开始的 18%.

5 对比 DBMS 和文件 I/O

测试环境参照 4.3

5.1 SQL 语句准备以及文件类型

为了比较不同类型操作的性能差异，我们设计了四种 SQL 语句来进行比较分析：分别是 `insert`, `delete`, `update` 和 `select`。

- 对于 `insert` 语句，我们设计为向已有弹幕表中插入弹幕，插入的弹幕为弹幕 csv 文件中的前任意行。
- 对于 `delete` 语句，我们利用了 `between-and` 语句，实现了将弹幕表中任意一行开始删除 n 行的操作。
- 对于 `update` 语句，我们针对 `user_information` 表，对于为空的行进行替换。
- 对于 `select` 语句，我们假设了一个查询一个用户发送了多少弹幕和一个视频一共有多少弹幕的场景，利用聚合函数 `count` 实现查询。

我们所使用的数据储存在 CSV 文件当中，每一行数据对应了我们建立的表中的一行，每行的数据通过逗号分隔。

5.2 测试脚本描述

对于 SQL 语句的测试脚本，大体跟上文提到的导入类似。
首先建立起数据库连接，接着利用准备好的 SQL 语句进行 executeUpdate 操作，最后 commit 提交事务。当然，对于 select 操作我们还利用了一个 ResultSet 来接收结果。在每个脚本中我们还添加了计时器来进行测试比较。

对于文件操作的测试脚本，我们使用 BufferedReader 方法读入 csv 文件，逐行对数据进行筛选，当遇到满足要求的行，我们就进行删除、插入、查找操作，对于更新操作，我们遍历每个数据，发现数据为空时我们就更新该数据为“project”

5.3 打开 csv 文件以及建立数据库连接耗时分析

首先，我们通过删除操作来测试 Java 打开 csv 文件所需要的时间以及 Java 连接数据库所需要的时间。先是在打开 csv 文件和连接数据库之前开

始计时，记录耗时，然后再在打开 csv 文件和连接数据库之后开始计时，记录耗时。最后相减取平均，得到所需时间。

打开 csv 文件耗时分析

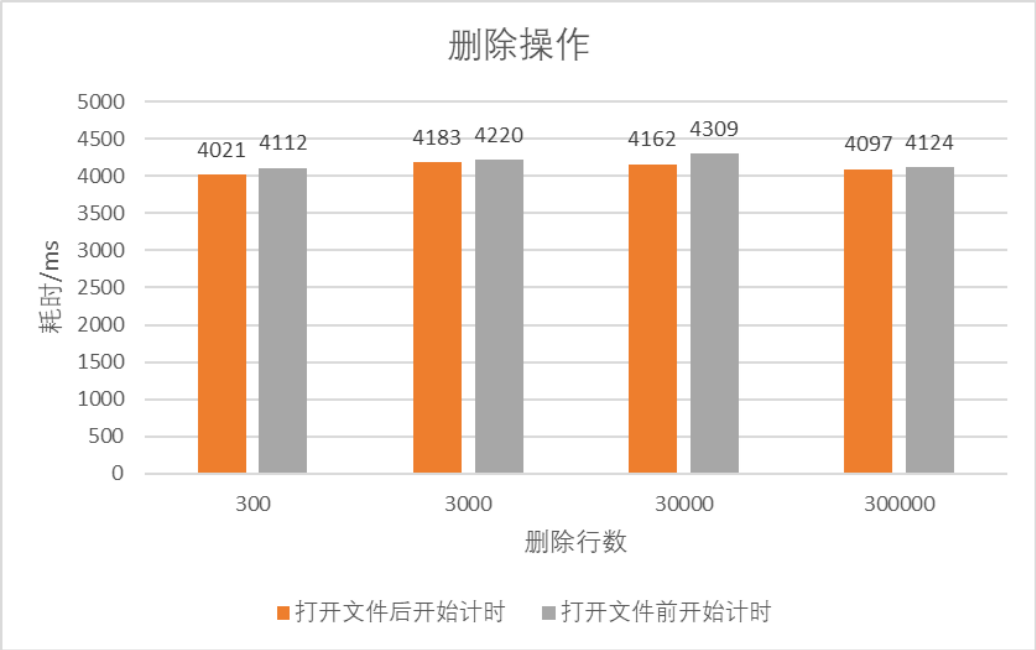

图 7: 打开 csv 文件耗时

相减取平均得出打开文件的平均耗时为 75.5ms。

连接数据库耗时分析

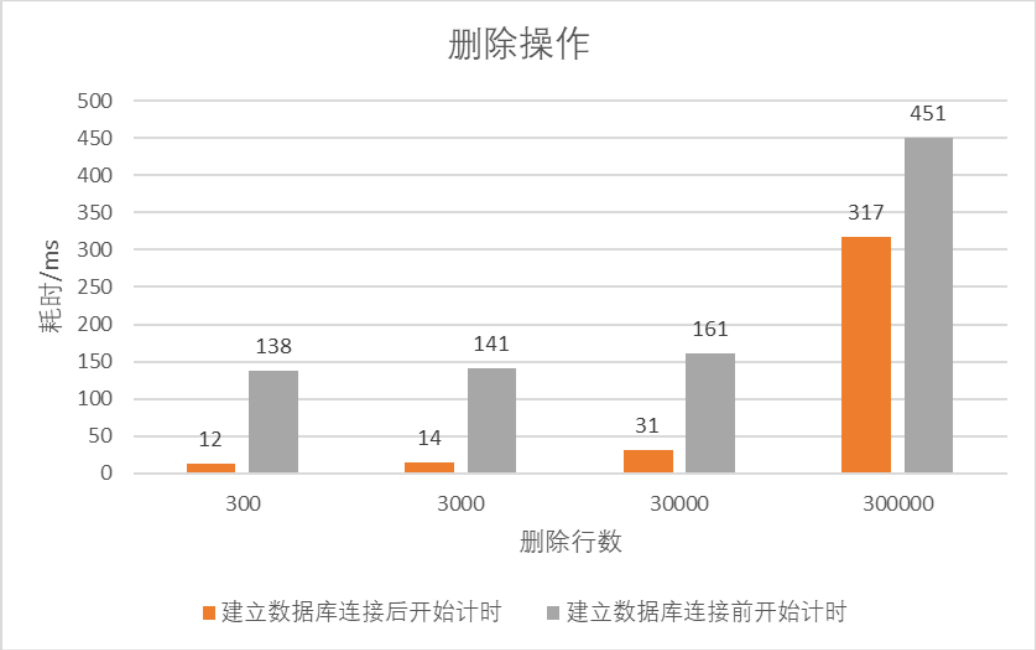

图 8: 连接数据库耗时

相减取平均得出建立数据库连接的平均耗时为 129.25ms。

速度差异分析

打开 csv 文件速度较快原因

1. 本地磁盘访问通常非常快。
2. 如果文件已经被操作系统缓存，连续的读取操作会很快。
3. Java 中的文件 I/O 操作通常很高效。

建立数据库连接速度较慢的原因

1. 建立数据库连接包括了多个步骤：加载 JDBC 驱动、与数据库建立 TCP 连接、进行身份验证和初始化连接对象。
2. 在本地数据库服务负载较高的情况下可能会增加连接的响应时间。

二者有时间差距的原因

1. 数据库连接是一个比简单的文件 I/O 更复杂的过程。它不仅包括磁盘访问，还包括网络协议处理、身份验证等。

2. 如果数据库系统正在服务于其他应用或者执行其他操作，它需要在多个任务之间共享资源，这可能导致连接建立时出现延迟。

3. 数据库连接的初始化包括准备连接的各种参数（比如：Host：数据库服务器的主机名或 IP 地址；Port：数据库服务器监听的端口号；Database Name：需要连接的数据库名称，等），可能还会加载数据库的一些初始设置（Client Encoding：设置客户端字符编码，以确保客户端和数据库之间正确地交换文本数据；Date/Timezone Settings：配置日期和时间的本地化设置，以匹配应用程序的时区），这些都是打开 CSV 文件不需要做的。

   为了防止打开 *csv* 文件和连接数据库在时间上的差异影响后续测试结果，接下来的所有操作都是在打开 *csv* 文件后，或是建立数据库连接后开始计时

5.4 删除操作耗时分析

在此操作中，我们将删除弹幕表中的 n 行数据，并记录耗时。

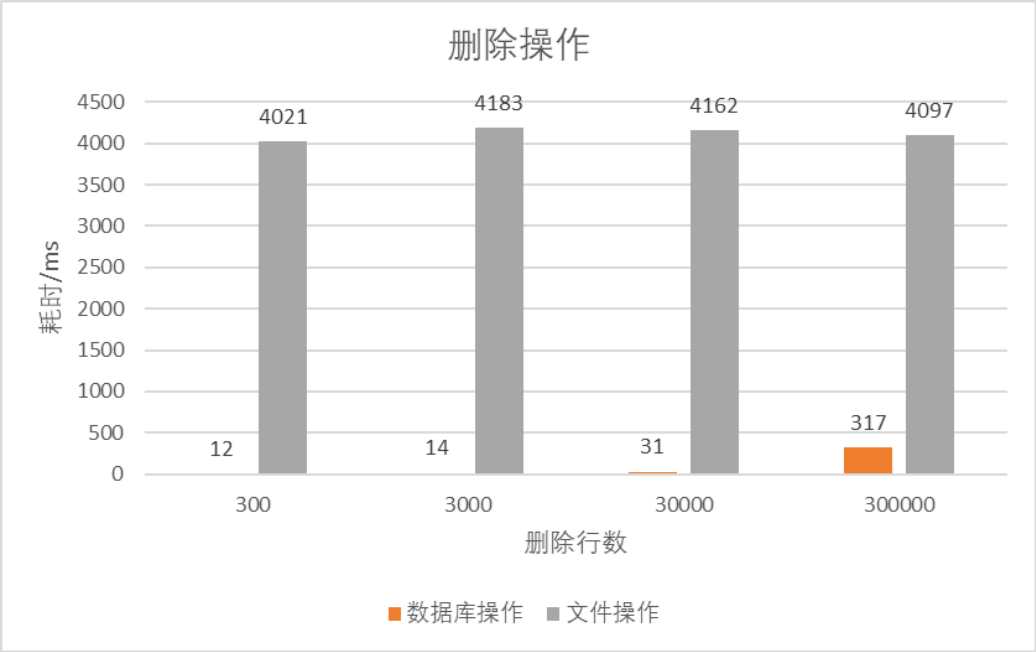

图 9: 两种删除操作的耗时对比

从图中我们可以得到，数据库操作耗时明显短于文件操作耗时。

原因分析

数据库操作耗时短的原因

1. 数据库操作通过直接与数据库通信来删除记录。数据库系统通常被优化以快速执行这类操作（如：索引；事务日志；数据页管理，自动清理，等），特别是当涉及到批量删除时。

2. 删除操作是在服务器端进行的，只需要发送命令和接收确认，网络上传输的数据量很小。

文件操作耗时长的原因

1. 读取和写入文件操作通常比内存操作慢。

2. 每一行都需要进行 I/O 操作，这在大文件中特别耗时。

3. 行号的判断需要在 Java 程序中进行，增加了 CPU 的计算负担。

4. 写入新文件需要额外的磁盘空间，并且在写入时还需磁盘 I/O。

5. 处理 CSV 文件的方式是在客户端进行的，涉及到处理每一行的数据，并且在删除行之后还需要重新写入数据到一个新文件，这在数据量大时会非常缓慢。

5.5 更新操作耗时分析

在此操作中,我们将“user_information”表中的所有空值更新为“project”,并记录耗时。

	数据库操作	**文件操作**
耗时/ms	171	98

表 1: 数据库与文件操作耗时对比

从图表中我们可以得出，数据库操作耗时略高于文件操作耗时。

原因分析

数据库操作耗时长的原因

1. 数据库的写操作通常需要更多的磁盘访问，比如更新索引、写回日志等。

2. 数据库操作通常是为了确保 ACID 特性（原子性、一致性、隔离性和持久性）而进行一系列优化的。这些优化确保了数据的安全性，但是会牺牲一定的速度。

3. 在数据库操作代码中设置了 `connection.setAutoCommit(false);` 以批量提交事务，但是事务提交的过程仍然需要时间。如果修改了大量数据，这个时间可能会非常长。

4. 数据库在更新数据时可能会锁定表或行,这是为了维护数据的一致性。这样的锁定有可能会延长操作时间，尤其是在并发访问的环境下，在

高并发的环境中，多个事务可能试图同时修改相同的数据。如果一个事务已经在一个数据项上持有锁，其他事务必须等待，直到锁被释放才能继续，这会导致等待时间增加。

5. 数据库需要解析 SQL 语句并生成执行计划，这在复杂的查询和更新操作中可能会消耗很多时间。

6. 数据库的更新操作通常需要更多的 CPU 资源，尤其是计算涉及大量数据行的时候。

文件操作耗时短的原因

1. 处理 CSV 文件时，虽然有磁盘 I/O 操作，但是它是一个相对简单的读和写操作，没有涉及到复杂的数据结构和事务管理。

2. 文件处理不需要额外的时间去处理 SQL 语句、事务日志和锁定机制。

3. 使用缓冲读写（`BufferedReader` 和 `BufferedWriter`）可以显著提高文件操作的效率，因为它减少了磁盘 I/O 次数。

5.6 插入操作耗时分析

在此操作中，我们将在弹幕表中插入 n 行数据，并记录耗时。

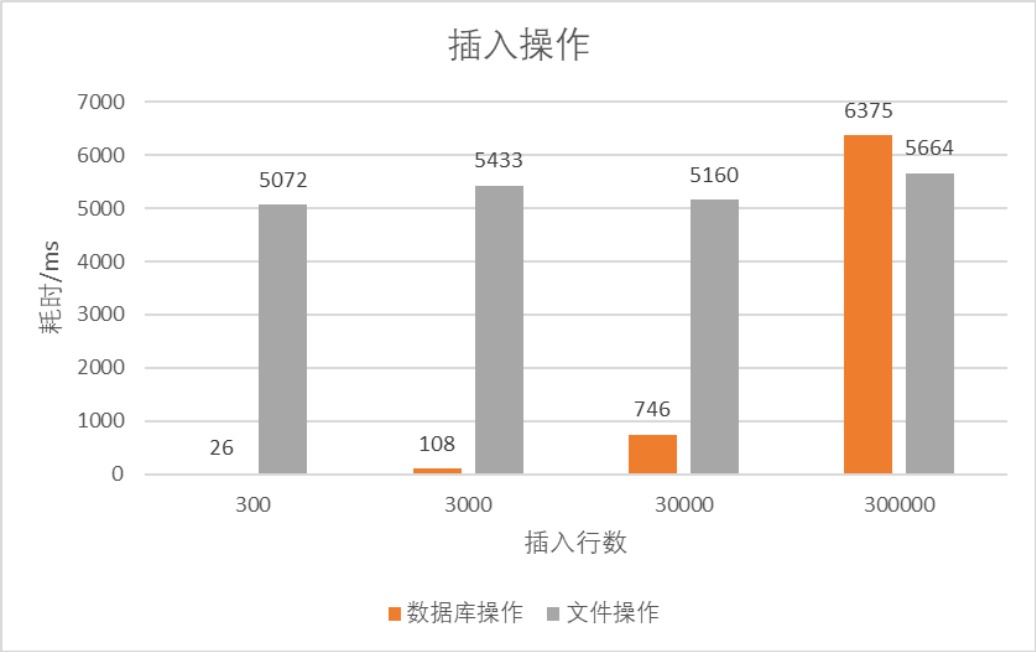

图 10: 两种插入操作的耗时对比

从图中我们可以看出，在插入行数少时，数据库操作耗时明显短于文件操作耗时，但当插入行数过大时，数据库操作耗时会超过文件操作。

原因分析

数据库操作：

1. 连接开销：数据库连接包含初始化连接、认证等开销，但这在整个操作过程中只做一次。

2. 预处理和批处理：使用预编译的 SQL 语句和批处理可以显著提高插入效率。预编译减少了每次插入时解析 SQL 的时间，批处理减少了与数据库的往返次数。

3. I/O 和日志：数据库操作通常伴随着磁盘 I/O，因为要更新索引、写事务日志等，以确保数据的完整性和持久性。

行数少时：

- 批处理操作可以在少数几次往返中快速完成，磁盘 I/O 和索引更新的时间也相对较短。

行数多时：

- 数据库需要处理更多的数据，这意味着更多的磁盘 I/O、更复杂的索引更新和更长的事务日志写入时间。随着数据量的增加，这些操作的时间开销会越来越大。

CSV 文件操作：

1. 文件读写是相对直接的磁盘操作，没有数据库那样的复杂性。没有事务日志、索引更新等开销。
2. 读取和修改 CSV 文件依赖于内存，如果文件很大，可能需要大量内存来存储文件内容。
3. 大量的文件操作（尤其是插入操作）可能涉及数据的移动，这会增加 I/O 的开销。

行数少时：

- 对于少量的行，文件操作可能会涉及复制少量的数据并重新写入，这可以迅速完成，但是相比数据库操作，即使是小量数据，也可能因为没有批处理而导致性能较差。

行数多时：

- 尽管文件的读写操作随着数据量的增加而变慢，但是这种增长往往是线性的。此外，没有额外的索引更新或事务日志写入等开销。因此，在处理大量数据时，相较于数据库操作，文件操作可能表现出更好的扩展性。

5.7 查找操作耗时分析

在此操作中，我们分别查找 n 个用户所发的弹幕总和以及 n 个视频所拥有的弹幕总和，并记录耗时。

查找用户

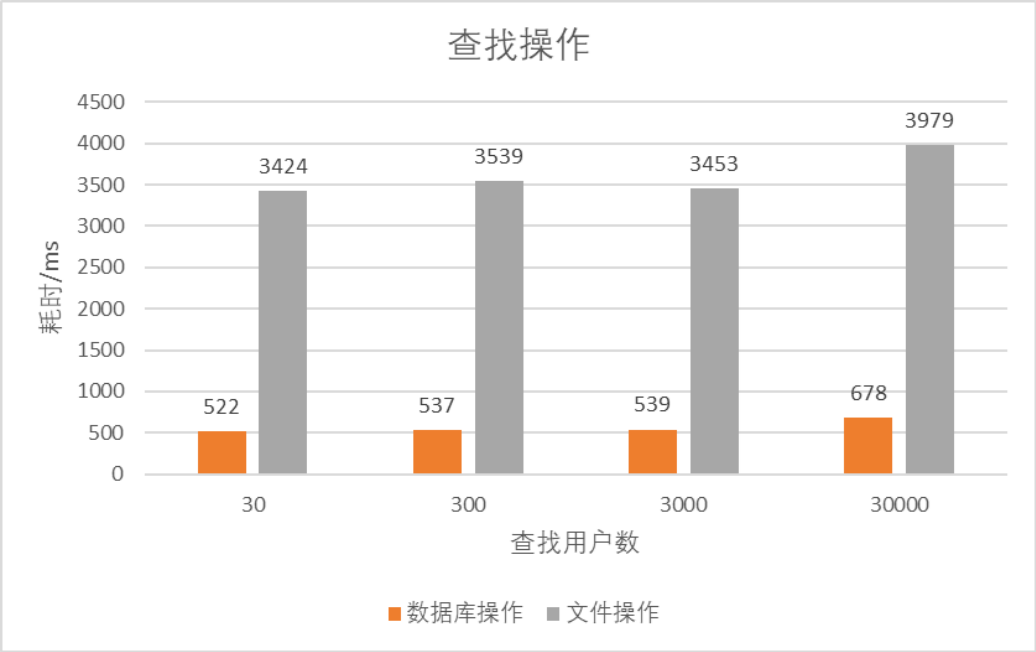

图 11: 两种查找操作的耗时对比

从图中我们可以看到，数据库操作耗时明显短于文件操作耗时。

原因分析

数据库操作耗时比文件操作耗时短的原因：

1. 数据库索引优化：数据库系统通常对查询操作进行了高度优化（如：预读取，根据查询模式预测接下来的 I/O 请求，提前读取数据；写入策略，如使用写入前日志（WAL）来确保数据一致性的同时优化写入性能；查询缓存，重复的查询结果可以从缓存中直接读取，减少计算开销，等）。如果 danmu_mid 字段被索引，数据库能够非常迅速地定位和计算结果。相对于逐行检查，数据库的查询引擎可以更快地执行这种操作。

2. 单次批量操作 vs. 多次迭代：s 数据库操作通过一次 SQL 查询来检索结果，而文件操作则通过逐行检查每行弹幕的数据来计算总数。在

CPU 密集型任务中，一次性完成大批量数据的操作通常比逐条处理数据要高效。

3. 内存与 CPU 使用：在数据库操作中，内存主要用于暂存 CSV 文件的 ID 列表，并将它们传递给数据库查询。文件操作则需要加载两个完整的文件进内存，并且在检查用户 ID 时占用更多 CPU 资源。

4. I/O 操作：文件操作需要读取两个文件，这涉及到更多的磁盘 I/O 操作。相比之下，数据库操作仅仅读取一个 CSV 文件，并执行一个内存到数据库的操作。

5. 语言和平台效率：数据库引擎通常是用 C/C++ 这类更接近系统层面的语言编写，对性能进行了优化，而 Java 虽然非常高效，但是在进行大量数据处理时，可能不如数据库引擎优化得那么好。C/C++ 这些语言编译为机器码，这意味着编译后的代码可以直接由硬件执行，没有额外的抽象层。而 Java 代码编译为字节码，运行时需要 JVM 解释或 JIT 编译为机器码。这增加了额外的转换步骤，可能会导致更高的耗时。此外，C/C++ 可以执行更多的系统级操作，如直接的内存访问和硬件中断处理，这使得它们可以更好地优化与硬件的交互。而 Java 运行在 JVM 上，提供了一个抽象层，为了保证跨平台兼容性和安全性，限制了直接的系统级访问。

6. 错误处理：数据库操作中的错误处理相对简单。而文件操作在处理文件读取错误时没有相应的中断，这意味着即便出错，它也会继续尝试读取整个文件。

7. 内存泄漏和垃圾回收：Java 程序可能会受到垃圾回收的影响，尤其是在处理大量对象（比如文件中的每一行数据）时。而数据库查询不太受这些因素的影响。

对于为什么随着查找的用户数量上升，数据库操作的耗时一定上升，而文件操作的耗时不一定上升：

对于数据库操作：

1. 当查找的用户数增加时，生成的 IN 子句变得更长，这导致了数据库查询语句变得更复杂。每增加一个参数，数据库查询优化器需要做更多的工作来准备执行计划，并且查询本身需要比较更多的键值。

2. 通常情况下，数据库对于较小的 IN 列表效率更高，因为索引可以快速排除大量不相关的记录。但当 IN 列表非常长时，数据库优化器使用索引或者查询的效率可能降低，因为要检查的潜在匹配项增多了。

对于文件操作：

1. 文件操作使用了一个 HashSet 来存储用户 ID，这个集合的查找时间复杂度是常数级别的（O(1)）。这意味着不管 HashSet 包含多少用户 ID，查找一个元素所需的时间都大致相同。

2. 由于文件只被读取了一次，与用户数量无关，所以这部分的时间消耗保持不变。

3. 如果增加的用户 ID 很少或根本没有对应的弹幕记录，文件操作的耗时可能实际上并不会增加，甚至可能会因为某些缓存效果而减少。这是因为它的性能更多地依赖于弹幕文件中有多少行是与给定用户 ID 集合匹配的，而不是用户 ID 集合的大小本身。

查找视频

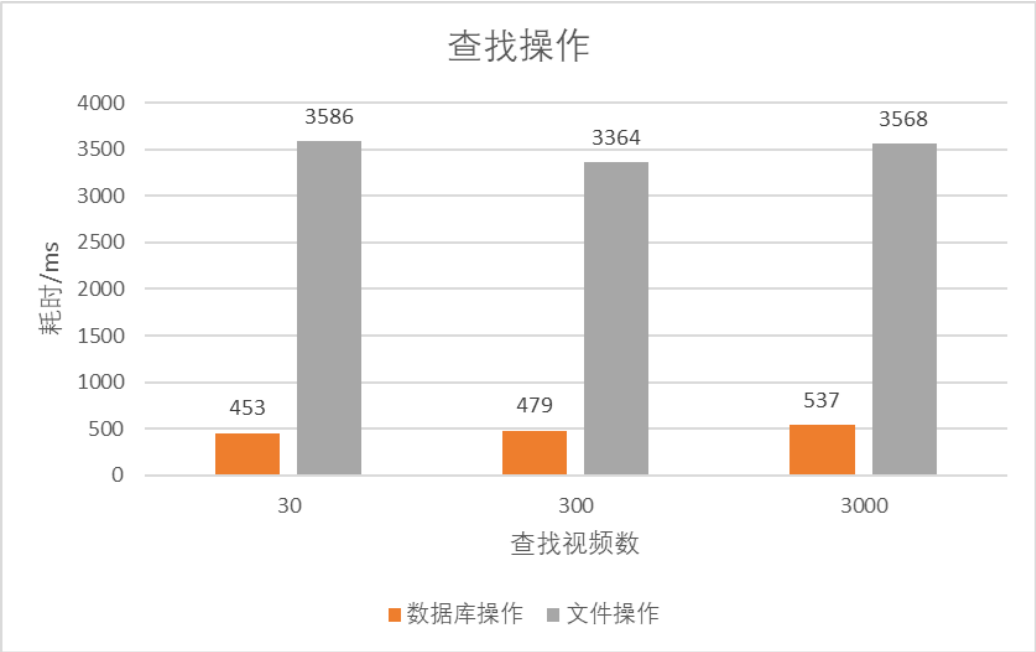

图 12: 两种查找操作的耗时对比

经过对比发现，查找 n 个视频的弹幕总和的结果与查找 n 个用户的弹幕总和的结果类似。并且执行的数据库操作和文件操作也类似。因此，二者在查找 n 个视频的弹幕总和的耗时上有差异的原因与二者在查找 n 个用户的弹幕总和的耗时上有差异的原因相同。

6 探索数据库高并发的处理方案

本实验所用配置见上文 4.3 部分。

在实际情况下面，为了应对高并发我们需要更加合理的设计数据库架构和所用开发代码，我们共探讨了以下几个方面。

- **数据库具体设计方面：**
  - 在可能会遇到大量查询的表中添加索引，但同时也要注意过多的索引会影响新的信息插入，因此要根据实际情况设计索引数量。

- 在数据库内部添加存储过程，让数据库利用本身的优化设计对查询或操作进行优化。

- **代码开发方面：**
  - 利用队列批量操作和批处理，设置合理的线程，并在每一个生产者的线程调用队列来进行操作。
  - 利用 `CALL` 语句调用内部的存储过程，而不是直接用基本的查询语句操作。

首先，我们测试了进行多次 select 操作时所用时间，准备的 SQL 语句为 select count(*) from danmu where danmu_Mid='1703941'，作用是查询用户表中的第一个用户发的弹幕总数.

由于弹幕表数据量较大，在没有添加索引时开启一个线程的情况下进行 10 次 selrct 需要约 4 秒，速度远远低于预期。

在给 danmu_mid 添加索引后，select 速度大大增加，我们测试了不同数量的 select 语句下所耗时间，结果如下：

select 语句数量	所用时间 (ms)
10	11
10000	660
1000000	50662

当同时进行的 select 语句数量在 10000 左右时，我们观察到所用时间在 1 秒内，这是一个可以接受的时间。然而，当数量较大时检索时间可能会比较久。同时，在实际情况下，我们的查询语句往往不会那么简单，单个查询所用时间会大大增加，因此，我们接下来考虑了多线程的方式来优化查询。

对此，我们为每一次查询开启一个线程，测试对电脑性能的占用和耗时。经过实验发现，收限于电脑性能，开启多线程达到 100 后对电脑 cpu 的占用已经达到了 100%。这大大影响了我们测试所用时间。

所以，在实际情况中，我们要根据当前的查询流量合理分配线程。根据研究调查，我们按照生产者 消费者模式进行高并发下的处理。具体思路如下：

首先，我们为每一个生产者线程分配一定数量的消费者，(当然，实际情况下由于每个消费者所做查询所耗时间不同，可能需要设计更好的算法来分配)。为每一个消费者线程添加一个阻塞队列 (可以确保线程安全)，按照分配好的消费者数量将所做查询添加进队列。当分配完后，我们开启消费者线程，从队列中取出查询结果来进行查询。代码见 (PostgresStoredProcedureTest 类)。这大大减少了对电脑 cpu 的占用，进行 100 次查询占用 cpu 大约为 30%，但是，这也在一定程度上影响了我们的查询速度，在实际情况中可能要合理分配线程。

此外，我们还探究了使用存储过程将将逻辑放在数据库服务器上，利用以下 SQL 语句在数据库中创建和定义存储过程。

```
CREATE OR REPLACE FUNCTION get_danmu_count
(danmuMid varchar)
RETURNS INTEGER AS $$
DECLARE
    resultCount INTEGER;
BEGIN
    SELECT COUNT(*) INTO resultCount FROM danmu
    WHERE danmu_Mid = danmuMid;
    RETURN resultCount;
END;
$$ LANGUAGE plpgsql;
```

接下来，我们用 call 语句调用数据库内部的存储过程。由于数据库本身会对该存储过程进行优化，同时会对操作进行封装，减少服务器中间的往返次数，减少通信所用时间。而且，数据库本身对高并发提供了较为成熟的处理机制，在遇到高并发问题时能够更加有效地解决。

7 对比 PostgreSQL 和 MySQL

本实验中使用的 MySQL 版本为 8.1.0。

首先，我们对两种数据库导入 danmu 表数据 (12478996 条) 所用时间进行了比较：

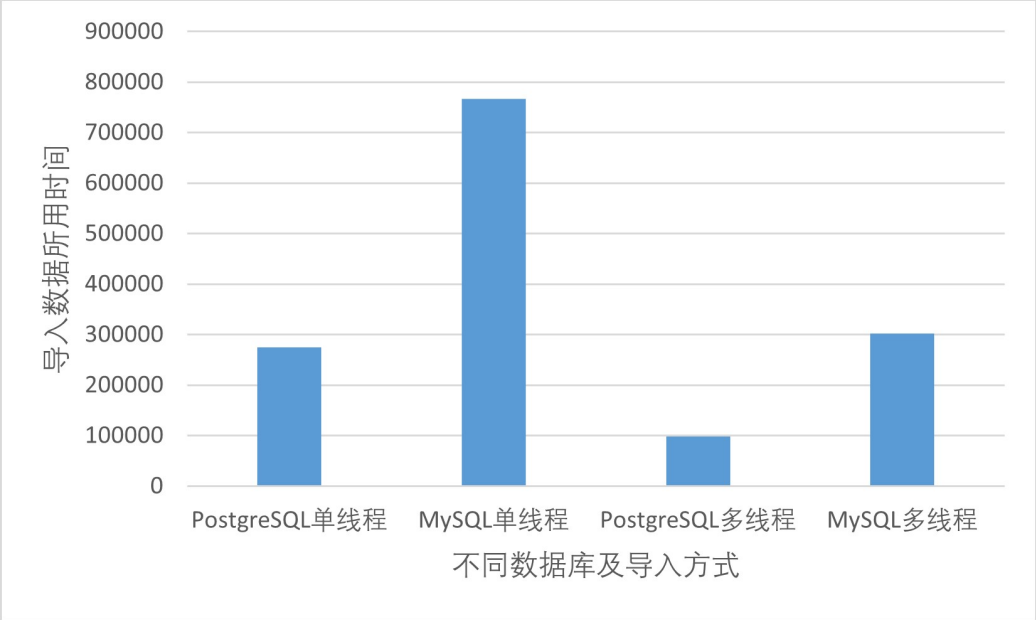

图 13: 不同数据库导入弹幕所用时间比较

可以看到，在相同配置，利用相同脚本进行数据导入时，MySQL 所用时间远远多于 PostgreSQL, 甚至 PostgreSQL 在仅仅开启单线程的情况下导入时间都优于 MySQL 多线程操作。

select 次数	**10**	**10000**	**1000000**
PostgreSQL	11	660	50662
MySQL	22	1958	139042

表 2: 不同数据库 select 耗时对比

可以看到 PostgreSQL 数据库在进行大量操作时性能优于 MySQL 数据库。

经过研究调查，可能有如下原因：

- PostgreSQL 能够较好地处理并发任务，它采用了多版本并发控制(MVCC)机制来处理并发。所以在多条 `SELECT` 同时进行时速度较快。

- PostgreSQL 的 Write-Ahead Logging（WAL）特性使得在对数据进行写入时进行了优化，减少了写入时间。
- 同时，PostgreSQL 可能具有更先进的查询处理器，可以对 `SELECT` 进行优化。
- 我们在进行数据导入时涉及到多种数据类型，PostgreSQL 在处理多数据类型方面性能可能较好。

8 对比 Java 和 Python

我们分别使用 Java 代码和 Python 代码，来对弹幕的 csv 文件进行删除行操作，记录并比较耗时。

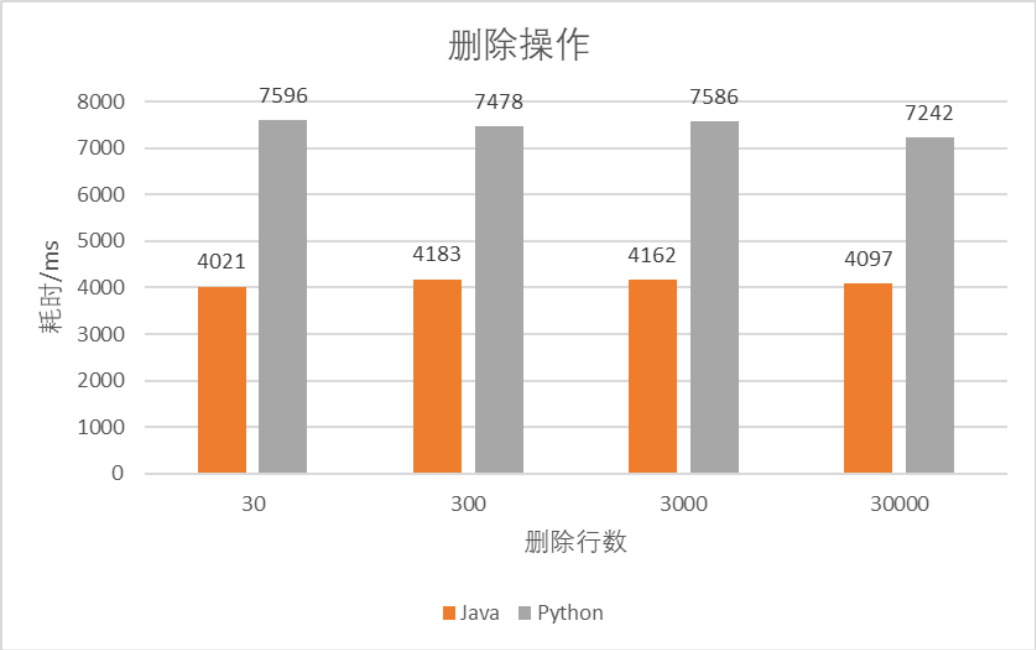

图 14: 两种编程语言的耗时对比

从图中可以得出，Java 的耗时短于 Python 的耗时。

原因分析

Java 代码

1. Java 使用 BufferedReader 和 BufferedWriter，这两个类是为高效读写而设计的。它们使用缓冲区来减少实际的磁盘 I/O 操作次数。
2. Java 代码在编译时进行了优化，JVM 也在运行时提供即时编译（JIT），进一步优化性能。
3. Java 是静态类型的，类型检查在编译时完成，这可以帮助 JVM 优化代码执行。

Python 代码

1. Python 使用内置的 open 函数来创建文件对象。虽然 Python 也有缓冲，但它的实现可能没有 Java 那样经过深度优化。
2. Python 代码是解释执行的，通常没有像编译型语言那样的执行速度优势。
3. Python 是动态类型的，这增加了运行时的某些开销，因为每个操作都需要类型检查和解析。

Java 比 Python 更快的可能原因：

1. JVM 在运行 Java 字节码时进行大量优化，包括热点代码探测、即时编译和多级缓存。
2. 静态类型允许在编译阶段进行更多优化，而这些优化在 Python 中通常不可能实现。
3. Java 的 I/O 类为性能优化提供了更多控制，例如可以指定缓冲区的大小。
4. Python 代码在运行时必须由解释器逐行解释和执行，这会增加开销。
5. Python 的动态类型系统虽然灵活，但是每次操作时都进行类型检查会减慢执行速度。

9 对比不同电脑配置对导入数据速度的影响

我们使用了两台配置不同的电脑，命名为电脑 1 和电脑 2，分别进行弹幕数据的导入操作，记录并比较耗时。

电脑 1 配置

处理器 12th Gen Intel(R) Core(TM) i7-12700H 2.30 GHz
机带 RAM 16.0 GB (15.7 GB 可用)

电脑 2 配置

处理器 12th Gen Intel(R) Core(TM) i9-12900H 2.50 GHz
机带 RAM 16.0 GB (15.7 GB 可用)

导入弹幕数据	**电脑 1**	**电脑 2**
耗时/ms	274574	236210

表 3: 导入操作耗时表格

原因分析

1. Intel Core i9-12900H 比 Core i7-12700H 具有更高的基本频率（2.50 GHz 相比于 2.30 GHz），以及可能有更高的 Turbo Boost 频率，这意味着在需要高性能时，i9 处理器能够提供更高的计算速度。

2. 虽然这里没有提供具体的核心和线程数，但一般来说，i9 系列处理器会拥有比 i7 系列更多的核心和线程。这可能会在多线程应用程序中提供更好的性能。

3. i9 处理器可能拥有比 i7 更大的缓存。CPU 缓存能够存储临时数据，大缓存能够提高处理大数据集时的效率，这在处理 CSV 文件等操作时尤其重要。

4. i9 处理器可能有更高级的热管理和功耗特性，这可以在高负载下维持更高的频率而不会过热降频。

10 对比是否使用库对文件操作速度的影响

我们用两个 Java 程序分别实现删除弹幕 csv 文件的 n 行，其中一个 Java 程序使用库，另一个不使用库，记录并对比耗时。

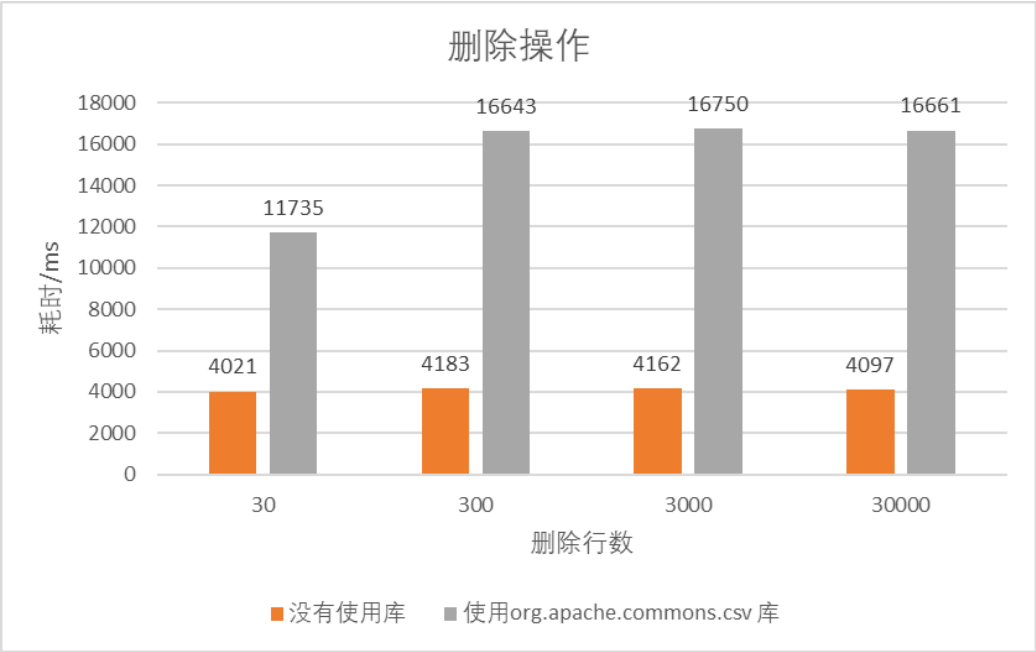

图 15: 两种删除操作的耗时对比

原因分析

1. 第一段代码使用了 CSVParser 和 CSVPrinter 类来处理 CSV 文件的读写操作。这些类提供了额外的功能，例如处理逗号、引号、换行等特殊字符，并支持不同的 CSV 格式。这些额外功能可能会引入额外的处理时间。第二段代码直接使用了 BufferedReader 和 BufferedWriter 类，这些类对文件的处理更接近于原始数据流的处理方式，没有额外的格式处理负担。

2. Apache Commons CSV 库为了提供强大的功能和易用性，它的操作可能会包含更多的方法调用和对象创建。对于大文件，这些操作的累积可能会显著影响性能。