CS307 Principle of Database Systems

PROJECT 01 REPORT
Database for Management Department of SUSTC

1 项目基本信息

1.1 项目作者

- 项目作者01
  - 姓名：罗嘉诚（Jiacheng Luo）
  - 学号：12112910
  - 实验课：Lab2
- 项目作者02
  - 姓名：伦天乐（Tianle Lun）
  - 学号：12113019
  - 实验课：Lab2

1.2 各成员项目贡献

根据小组内部讨论并达成一致，建议最终项目贡献比：

- 罗嘉诚：50%
- 伦天乐：50%

各任务详细贡献如下：

- 罗嘉诚

1. 设计与绘制 E-R图
2. 数据库设计与分析
3. 实验结果分析与数据可视化
4. 比较DMBS和文件I/O
5. 报告写作

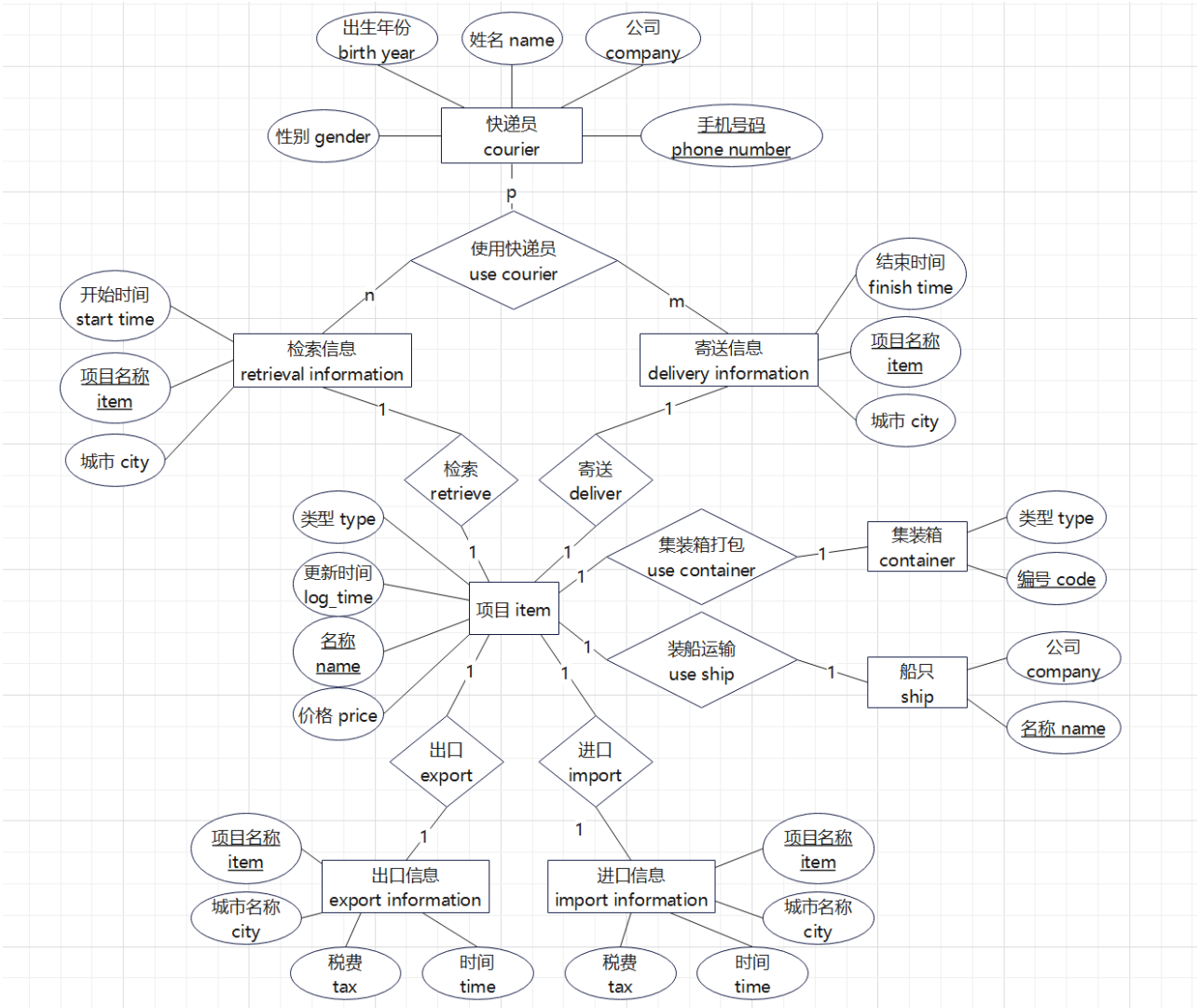
- 伦天乐

1. 编写导入脚本
2. 编写 DDL 和 SQL 文件
3. 比较DMBS和文件I/O
4. 项目测试
5. 报告写作

2 Task 1： E-R 图

根据项目说明中提供的逻辑，迭代多次版本。

使用 亿图图示（EDraw Max） 绘图工具，绘制项目的 E-R 图，提供截图如下：



3 Task 2: 数据库设计

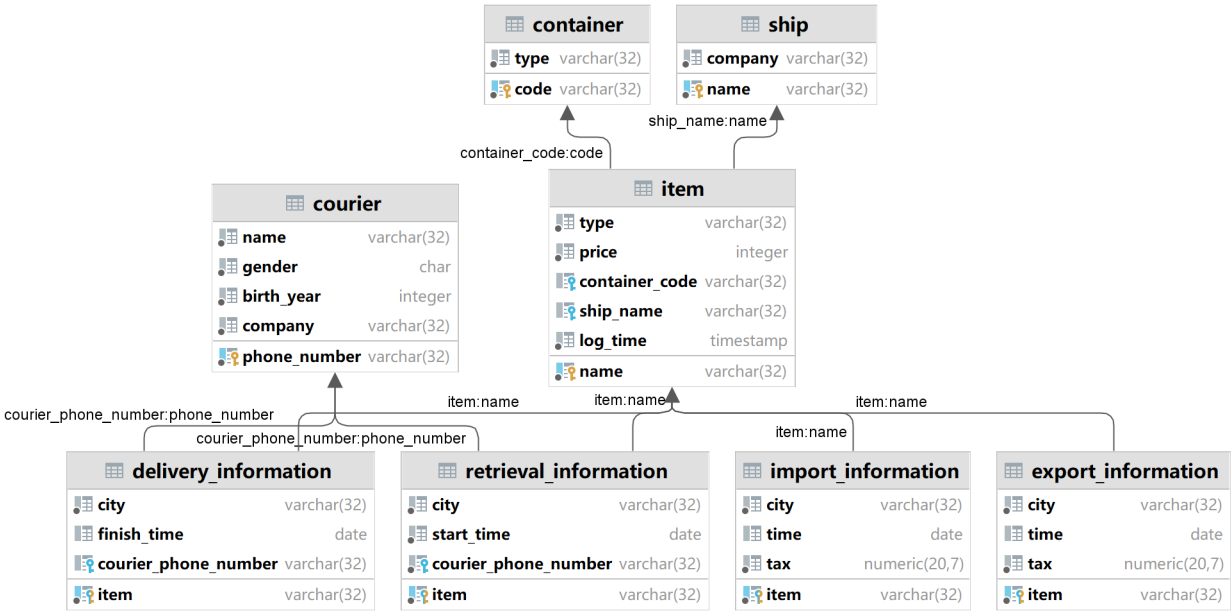
使用附录中 `table_maker.sql` 文件使用 `DDL` 创建数据表。

3.1 数据库设计

使用 `Show Visualization` 显示表设计及关系。

在 `DataGrip` 中创建数据表后并全选，右键 `Diagram > Show Visualization` ，

保存图片如下：



3.2 合理性说明

3.2.1 数据表及其列含义说明

在数据表构建中，我们创建了 8 个数据表，数据表和其中列、外键的含义如下：

- `item` 存储项目信息，包括基本信息`type`（项目类型）、`price`（项目价格）、`name`（项目名称，由于是`unique`的，将其作为主键）、`log_time`（项目最新更新时间）；以及项目进口、出口、寄送、检索、打包、装船的相关信息（分别使用外键`import_information`、`export_information`、`delivery_information`、`retrieval_information`、`container`、`ship`表示）。
- `retrieval_information` 存储检索步骤的信息，包括基本信息`city`（检索城市）、`start_time`（检索时间，即该项目开始时间）、`item`（检索信息所属的项目名称，由于是`unique`的，将其作为主键）；以及快递员的相关信息（使用外键`courier`表示）
- `delivery_information` 存储寄送步骤的信息，包括基本信息`city`（检索城市）、`finish_time`（结束时间，即该项目结束时间）、`item`（寄送信息所属的项目名称，由于是`unique`的，将其作为主键）；以及快递员的相关信息（使用外键`courier`表示）
- `export_information` 存储出口的信息，包括基本信息`city`（出口城市）、`time`（出口时间）、`tax`（出口税费）、`item`（出口信息所属的项目名称，由于是`unique`的，将其作为主键）
- `import_information` 存储进口的信息，包括基本信息`city`（进口城市）、`time`（进口时间）、`tax`（进口税费）、`item`（进口信息所属的项目名称，由于是`unique`的，将其作为主键）
- `ship` 存储装船信息，包括基本信息`company`（管理船公司）、`name`（船名称，由于是`unique`的，将其作为主键）
- `container` 存储打包信息，包括基本信息`type`（集装箱类型）、`code`（集装箱编号，由于是`unique`的，将其作为主键）
- `courier` 存储所有快递员信息，包括基本信息`name`（姓名）、`gender`（性别）、`birth_year`（出生年份，便于计算年龄）、`company`（所属的公司）、`phone_number`（电话号码，我们认为，快递员和其手机号中的用户号码（即除去区号的剩余号码）可以一一对应，所以将其作为主键）

3.2.2 数据库构建满足三大范式

数据库三大范式的定义如下：

第一范式：数据库表的每一列都是不可分割的基本数据项，同一列中不能有多个值。

第二范式：每个表必须有主关键字,其他数据元素与主关键字一一对应。

第三范式：表中所有数据元素不但唯一地被主关键字所标识,而且它们之间必须相互独立,不存在其他函数关系。

通过示意图可以看到，对于每一个数据表，每一列都是不可分割，在当前应用场景下，同一列没有多个值。每个表都有主关键字，主关键字都是`unique`（独一无二）的，其他数据元素能和主关键字一一对应。通过设计外键连接，我们将同一数据表中具有“传递”关系的数据列设计成不同的表格进行设计。

可以看到，按照以上设计思想设计出的数据库满足三大范式的要求。

3.2.3 数据库构建的其他要求

根据项目要求文档`CS307-2022Fall_Project1.pdf`所指出的详细的注意点，在本报告中依次叙述：

1. 数据库的构建，既依据项目说明又依据`shipment_records.csv`中的数据（参考列名称、数据值得范围），后续通过能完全导入给出得数据，来进一步验证这一点。
2. 数据库的构建，严格地遵循了三大范式得要求。
3. 数据库的构建是用主键描述最重要、最有辨识度的属性，并且通过外键描述数据表之间的关系。
4. 数据库中每个数据表中的每一行数据都由其主键列唯一标识。
5. 没有一个数据表是孤立的。
6. 表和表之间通过外键形成树的形状，不存在外键成环情况。
7. 每个表至少含有一列是非空的，这在数据库构建时使用`not null`约束加以限制。
8. 每个表至少有一列采用`unique`约束。
9. 在数据表中，所有数据都采用合适的数据类型加以存储，如姓名、公司等文本，使用`varchar()`来存储，性别字符采用`char`来存储，进出口税费小数采用`numeric()`来存储，项目价格正整数采用`integer`来存储，项目更新时间精确到时分秒，采用`timestamp`来存储，而其他时间精确到年月日采用`date`来存储。
10. 上述数据库中数据表的构建，具有很强的可扩展性，这在接下来的实验中将有所体现。

4 Task 3： 数据导入

在数据导入部分，我们使用`Java`语言编写脚本，使用`PostgreSQL`和`MySQL`分别进行数据导入，并对这两种`SQL`导入之间的效率进行比较；同时迭代数据导入脚本的版本（迭代三个版本`SimpleDataManager`、`FastDataManager`、`MultiThreadDataManager`），使之在效率方面进行优化。此外，我们在使用`MultiThreadDataManager`脚本的基础上对比采用`导入数据同时进行约束检查`和`预处理-导入数据-约束检查`两种方案的效率，并由此估计出外键约束对导入数据效率的影响。

上述所有内容均在本地进行相关测试。

4.1 测试环境

请见 5.1 节，我们非常详细地给出了本报告中所有测试的运行环境。

在测试中：

- 为了避免实验数据的偶然性误差，我们取多次测试的平均值在报告中呈现。
- 为了保证实验的精确程度，进行测试时，计算机只运行测试进程。

如需使用代码，请参阅 **6 附录：代码结构说明和运行须知**。

4.2 脚本编写

由于不同的 `SQL` 可能对数据导入效率产生影响，我们对比采用 `MySQL` 和 `PostgreSQL` 两种 `SQL` 导入数据的效率。

这两种方式内部，我们都使用了三种不同的脚本迭代版本进行数据导入：

- `SimpleDataManager`：SQL 语句逐条执行导入，对于每个`Item`的每条信息（对应8个表），都向数据库连接递交一次执行链接，若有 500000 个`Item`，则会进行 4000000 次执行链接。
- `FastDataManager`：根据MySQL官方文档 8.2.5.1，在 `SimpleDataManager` 中，由于是单行数据插入，每次递交一次执行链接，都会有 30% 的时间占比在链接数据库， 20% 的时间在发送请求， 20% 的时间在解析请求， 10% 的时间在关闭链接，这样的占比是不可接受的。也就是说，我们需要把时间主要花在插入，而不是链接等额外的时间，因此我们选择**批量执行SQL**语句，也就是一次链接，多行插入。在`FastDataManager`中，我们每5000条 SQL 语句提交一次，并且一个表一个表的进行数据导入。
- `MultiThreadDataManager`：在`FastDataManager`中，我们已经得到较好的数据导入表现，但是我们仍然没有发挥数据库的优势——并发。相较于一般的文件存储，数据库最大的优势是可以多终端多用户同时读写。在`FastDataManager`中，数据多表排队导入导致大量时间被浪费在了一张表上，因此我们采用多线程优化，实现了多表同时导入。

并且在`MultiThreadDataManager`脚本的基础上，因为外键约束的关系，我们同时测试了以下两种方式：

- 导入过程进行的同时同步进行约束检查。
- 先对数据进行预处理使之满足约束、再直接导入数据，最后再进行整体约束检查。

为了追求导入数据的高效性，避免无效的外键检查（因为建表数据导入预处理后可以保证无外键冲突），我们最终采用了忽略外键作为最终表现。

我们在本地进行实验，得到结论：

1. 所有脚本都成功运行，将所有数据导入数据库。
2. 在使用同一 `SQL` 导入数据时，三种迭代版本的脚本运行效率有明显区别，`SimpleDataManager`、`FastDataManager`、`MultiThreadDataManager` 的效率依次递增。
3. 在使用同一版本脚本导入数据时，使用不同的`SQL`，对导入数据效率有一定影响。在使用 `SimpleDataManager` 、`MultiThreadDataManager` 版本导入数据时，`PostgreSQL` 的效率明显优于`MySQL`，而是采用`FastDataManager`版本导入数据时，`MySQL` 的效率明显优于 `PostgreSQL`。
4. 两种数据导入方式运行效率差异对不同的`SQL`有一定区别。对于`PostgreSQL`，后者（`预处理-导入数据-约束检查`）在效率方面明显优于前者（`导入数据同时进行约束检查`）；对于`MySQL`，两者在运行效率方面没有明显区别。

4.3 实验结果与分析

4.3.1 脚本运行情况与数据导入情况

我们在脚本设计中以`[LOG] [year/month/day time] Case Information: Time Cost`的格式在终端输出脚本运行信息。

- 使用 `PostgreSQL` 中 `SimpleDataManager` 脚本 导入 `500000` 条数据的 `3` 次实验

```
[LOG][2022/9/29 04:19:08] PostgreSQL SimpleDataManager 500000 Records Init Cost #0: 388537ms
[LOG][2022/9/29 04:25:39] PostgreSQL SimpleDataManager 500000 Records Init Cost #1: 391026ms
[LOG][2022/9/29 04:32:23] PostgreSQL SimpleDataManager 500000 Records Init Cost #2: 403587ms
```

- 使用 `PostgreSQL` 中 `FastDataManager` 脚本 导入 `500000` 条数据的 `3` 次实验

```
[LOG][2022/9/29 04:33:52] PostgreSQL FastDataManager 500000 Records Init Cost #0: 88551ms
[LOG][2022/9/29 04:35:22] PostgreSQL FastDataManager 500000 Records Init Cost #1: 89845ms
[LOG][2022/9/29 04:36:54] PostgreSQL FastDataManager 500000 Records Init Cost #2: 91893ms
```

- 使用 `PostgreSQL` 中 `MutilThreadDataManager` 脚本 采用 `导入数据同时进行约束检查` 方式导入 `500000` 条数据的 `3` 次实验

```
[LOG][2022/9/29 05:28:51] PostgreSQL MutilThreadDataManager(With Constraint) 500000 Records Init Cost #0: 43009ms
[LOG][2022/9/29 05:29:34] PostgreSQL MutilThreadDataManager(With Constraint) 500000 Records Init Cost #1: 42466ms
[LOG][2022/9/29 05:30:16] PostgreSQL MutilThreadDataManager(With Constraint) 500000 Records Init Cost #2: 42294ms
```

- 使用 `PostgreSQL` 中 `MutilThreadDataManager` 脚本 采用 `预处理-导入数据-约束检查` 方式导入 `500000` 条数据的 `3` 次实验

```
[LOG][2022/9/29 04:37:07] PostgreSQL MutilThreadDataManager(Without Constraint) 500000 Records Init Cost #0: 13686ms
[LOG][2022/9/29 04:37:21] PostgreSQL MutilThreadDataManager(Without Constraint) 500000 Records Init Cost #1: 13080ms
[LOG][2022/9/29 04:37:34] PostgreSQL MutilThreadDataManager(Without Constraint) 500000 Records Init Cost #2: 13436ms
```

- 使用 `MySQL` 中 `SimpleDataManager` 脚本 导入 `500000` 条数据的 `1` 次实验

*由于此项耗时较长，采用每执行 `1000` 条数据，输出当前的平均用时，并以此估计总用时。

```
[LOG][2022/9/29 05:11:33] MySQL SimpleDataManager average cost: 49.507000ms
[LOG][2022/9/29 05:12:16] MySQL SimpleDataManager average cost: 46.408500ms
[LOG][2022/9/29 05:12:57] MySQL SimpleDataManager average cost: 44.633333ms
[LOG][2022/9/29 05:13:38] MySQL SimpleDataManager average cost: 43.616000ms
[LOG][2022/9/29 05:14:18] MySQL SimpleDataManager average cost: 42.964000ms
[LOG][2022/9/29 05:14:59] MySQL SimpleDataManager average cost: 42.517833ms
[LOG][2022/9/29 05:15:39] MySQL SimpleDataManager average cost: 42.213429ms
[LOG][2022/9/29 05:16:20] MySQL SimpleDataManager average cost: 42.052750ms
[LOG][2022/9/29 05:17:00] MySQL SimpleDataManager average cost: 41.815556ms
[LOG][2022/9/29 05:17:40] MySQL SimpleDataManager average cost: 41.651100ms
```

- 使用 `MySQL` 中 `FastDataManager` 脚本 导入 `500000` 条数据的 `3` 次实验

```
[LOG][2022/9/29 04:50:29] MySQL FastDataManager 500000 Records Init Cost #0: 60161ms
[LOG][2022/9/29 04:51:29] MySQL FastDataManager 500000 Records Init Cost #1: 58058ms
[LOG][2022/9/29 04:52:31] MySQL FastDataManager 500000 Records Init Cost #2: 59912ms
```

- 使用 `MySQL` 中 `MutilThreadDataManager` 脚本 采用`导入数据同时进行约束检查` 方式导入 `500000` 条数据的 `3` 次实验

```
[LOG][2022/9/29 05:31:12] MySQL MutilThreadDataManager(With Constraint) 500000 Records Init Cost #0: 53958ms
[LOG][2022/9/29 05:32:08] MySQL MutilThreadDataManager(With Constraint) 500000 Records Init Cost #1: 53663ms
[LOG][2022/9/29 05:33:03] MySQL MutilThreadDataManager(With Constraint) 500000 Records Init Cost #2: 53456ms
```

- 使用 `MySQL` 中 `MutilThreadDataManager` 脚本 采用 `预处理-导入数据-约束检查` 方式导入 `500000` 条数据的 `3` 次实验

```
[LOG][2022/9/29 05:20:36] MySQL MutilThreadDataManager(Without Constraint) 500000 Records Init Cost #0: 58087ms
[LOG][2022/9/29 05:21:35] MySQL MutilThreadDataManager(Without Constraint) 500000 Records Init Cost #1: 56923ms
[LOG][2022/9/29 05:22:34] MySQL MutilThreadDataManager(Without Constraint) 500000 Records Init Cost #2: 56412ms
```

上述各项信息均表明，所有版本的脚本均正确运行，并完成向数据库中导入所有的`500000`条数据的任务。

此外，我们将测试文件生成的脚本信息以`log.txt`格式的文档作为附件一并上传，供查阅日志。

4.3.2 数据结果与分析

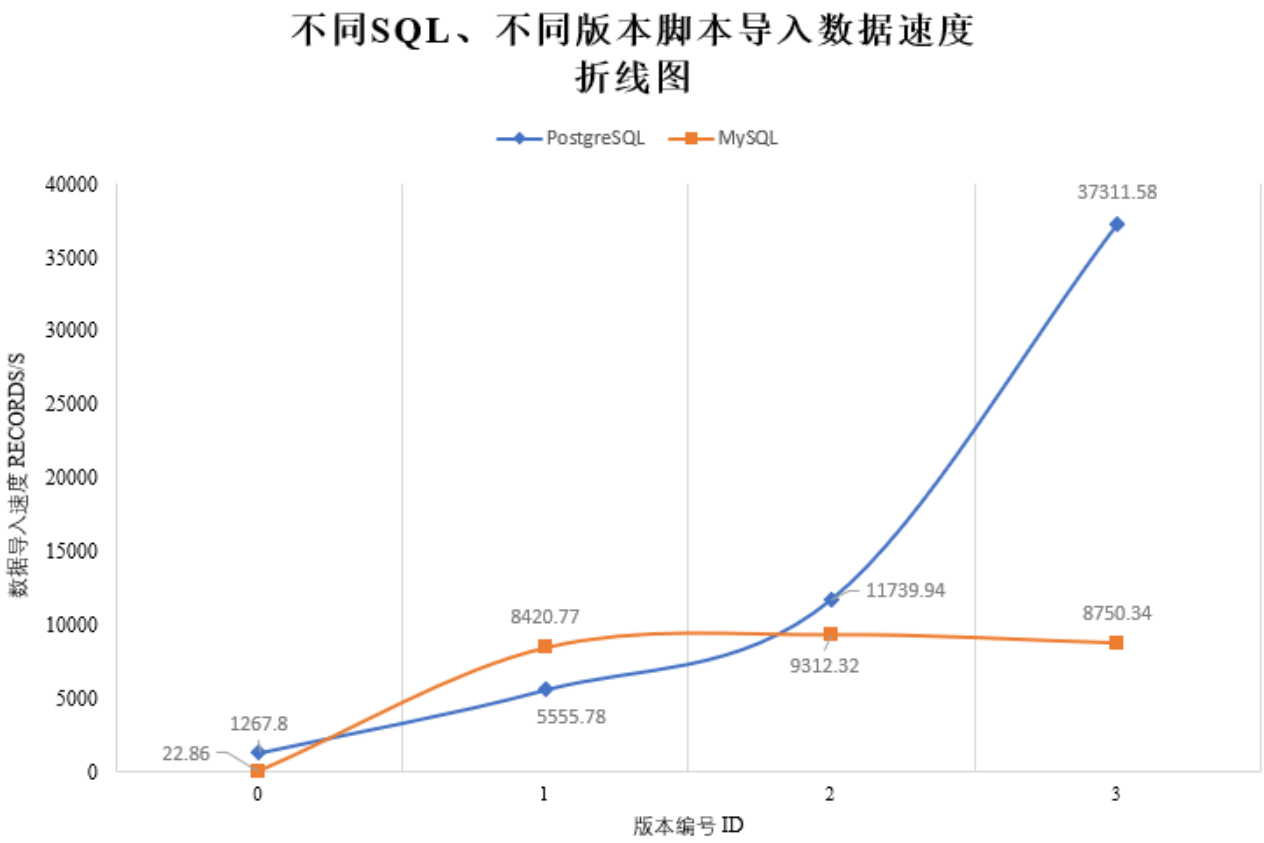
4.3.2.1 不同SQL和版本脚本与导入数据速率关系

我们经过数据处理，使用`每秒平均导入数据条数`(单位：`RECORDS/S`)来衡量导入数据快慢。

使用不同的`SQL`分别编写`4`个不同的版本，用`ID`分别表示：

- `0` `SimpleDataManager`：SQL 语句逐条执行导入。
- `1` `FastDataManager`：关闭AutoCommit，批量执行（batch) SQL 语句导入。
- `2` `MultiThreadDataManager(with constraint)`：多线程优化导入,导入数据同时进行约束检查。
- `3` `MultiThreadDataManager(without constraint)`：多线程优化导入，采用预处理-导入数据-约束检查。

根据导入全部`500000`条数据，获得的结果，绘制折线图如下：



4.3.2.2 优化结果

根据上图，使用`PostgreSQL`编写的脚本优化较为成功，总结如下：

- **Version 0** `SQL`

  语句逐条执行导入，时间效率较低。

- **Version 1** [+338.2%]

  关闭了`AutoCommit`，采用批量执行（batch)的方式进行导入，时间有所提高。

- **Version 2** [+111.3%]

  互相之间无关系的表采用多线程导入，进一步优化。

- **Version 3** [+217.8%]

关系型数据库，采用外键的时候，本身导入会占用大量时间，并且多线程过程中，也必须先执行被`reference`的表，因此我们在导入的过程关闭了所有的约束检查（数据初始导入预处理后保证满足约束）再导入完成后再恢复约束检查，进一步优化。

最终优化导入数据的脚本，较最开始版本效率提高，达 28.43 倍。

也就是说，原来可能需要 5 分钟才能完成的数据导入，最终优化后可能只需要 10s 即可完成导入。

5 Task 4： 比较数据库管理系统 DBMS 和文件输入输出 File I/O

5.1 测试环境

5.1.1 操作系统

Windows 10

5.1.2 硬件

- 内存：16384 MB (2 x DDR4-3192)
- 硬盘：NVMe SAMSUNG MZVL2512HCJQ-00B00(SSD) 1T
- CPU：11th Gen Intel Core i7-11800H 2300 MHz (4389 MHz)
- GPU：NVIDIA GeForce RTX 3060 Laptop GPU (6144 MB) 1000 MHz (900 MHz)

5.1.3 软件

- DataGrip 2022.2.2
- Eclipse IDE for Java Developers Version: 2021-09 (4.21.0)
- Java 1.8.0_351 Javac 1.8.0_351
- MySQL 8.0.29
- PostgreSQL 14.5

5.1.4 Java 外部库

- Fast Json by Alibaba 1.2.47
- PostgreSQL JDBC 42.5.0
- MySQL JDBC Drvier 8.0.25

5.1.5 复现实验说明

您可能需要在复现实验时更需要关注 `软件` 部分的配置，因为绝大多数的硬件配置时可以复现本实验的，而错误的`软件` 部分配置可能导致实验复现无法正确进行。

在测试中：

- 为了避免实验数据的偶然性误差，我们取多次测试的平均值在报告中呈现。
- 为了保证实验的精确程度，进行测试时，计算机只运行测试进程。

如需使用代码，请参阅 **6 附录：代码结构说明和运行须知**。

5.2 DDL 和 表格文件

我们选择在 `retrieval_information` 表格中进行实验。

- DDL

```sql
create table if not exists retrieval_information (
    item varchar(32) primary key,
    city varchar(32) not null,
    start_time date not null,
    courier_phone_number varchar(32) not null references courier(phone_number)
);
```

- 文件格式

使用上述 `DDL` 生成的文件为 `retrieval_information.csv` 该文件包含了`500000`行的大量数据和较为简单的存储结构。该表是理想对象，同时保持了简单性和一般性，数据足以反映比较差异，并且易于操作。

5.3 实验方案

为了对比数据库管理系统 DBMS 和文件输入输出 File I/O在 **增删改查** 的性能，我们对比使用不同`SQL`的情况（即使用`postgreSQL`和`MySQL`）

在数据库管理系统 DBMS 增删改查各项操作采用特定格式：

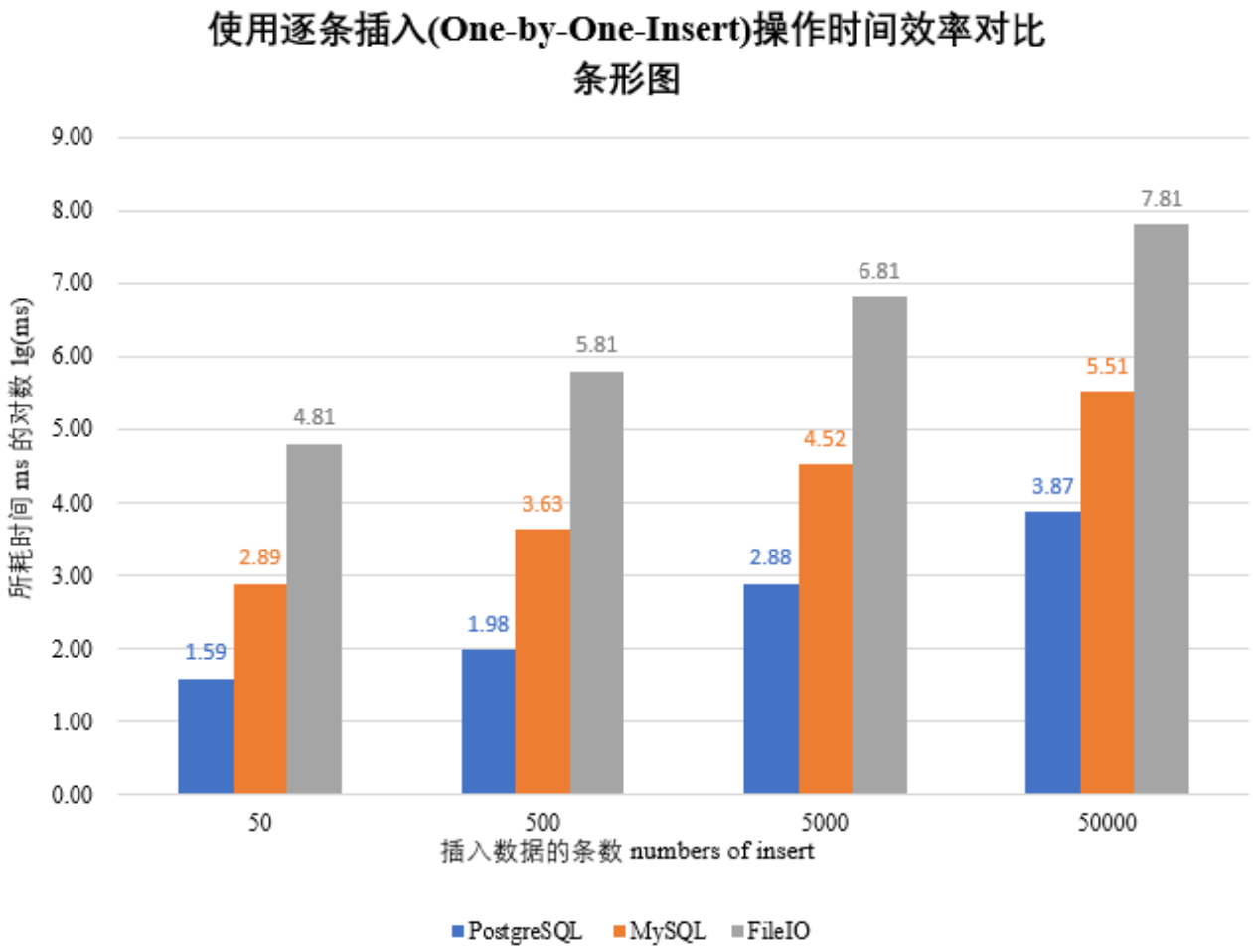
- `Insert` 单条数据依次插入，插入方案同`SimpleDataManager`。
- `Delete` 单条数据依次删除，`DELETE FROM retrieval_information WHERE item = 'item'`
- `Update` 单条数据依次执行，`UPDATE retrieval_information SET courier_phone_number = 'cpn' Where item = 'item'`
- `Select` 单条数据依次查询，`SELECT * FROM retrieval_information WHERE item = 'item'`
- 除此之外，使用 `MultiInsert`、 `MultiDelete`、`MultiUpdate`、`MultiDelete` 采用多条操作同时进行的方式再进行实验。

在文件输入输出 **File I/O**，存储方式采用了 **Json** 格式，使用了外部库 **Fast Json By Alibaba**，增删改查各项操作采用对应文件读写操作，测试脚本使用 `Java` 实现。

此外，我们还对不同数据规模的数据进行了实验。

5.4 实验结果

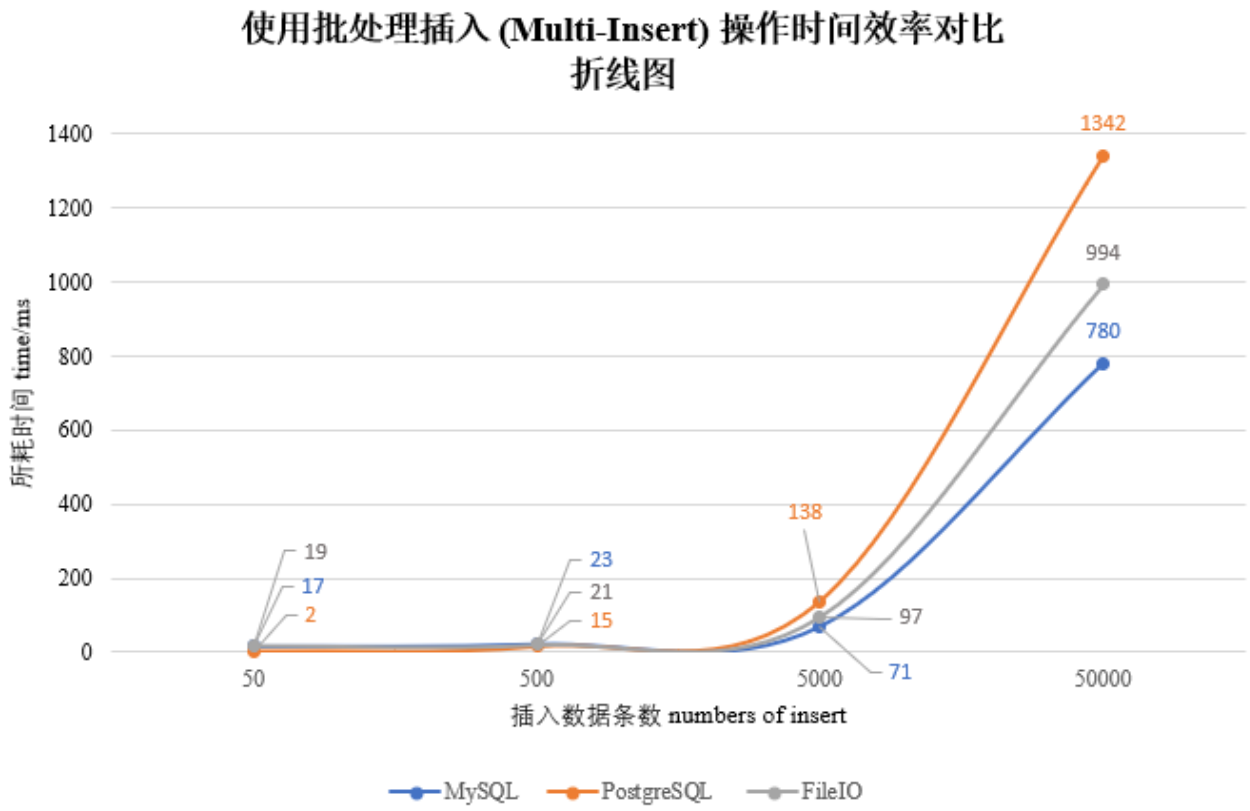
5.4.1 插入操作实验结果及结论



上图中，横坐标是测试插入数据的条数，纵坐标是所花时间 `单位：ms` 的**对数**。

图中数据可以很明显的发现，在使用逐条插入的朴素方法进行插入时：

- 无论总插入数据量如何，插入效率：`PostgreSQL` 远大于 `MySQL` 远大于 `FileIO`。



上图中，横坐标是测试插入数据的条数，纵坐标是所花时间 `单位：ms` 。

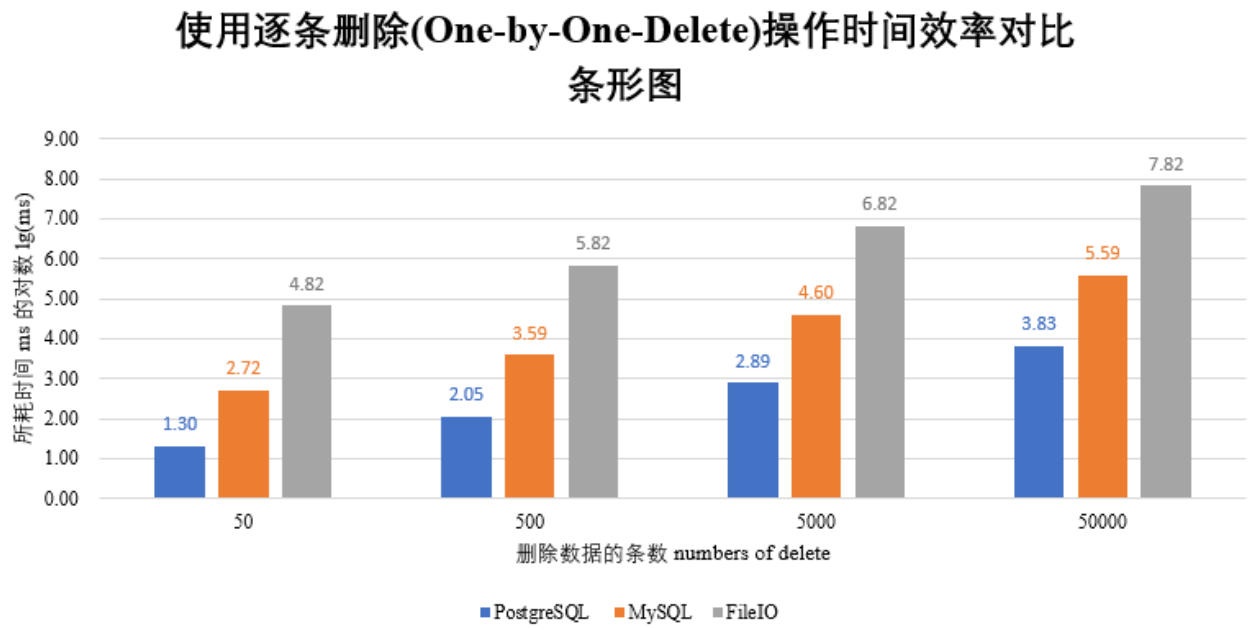
图中数据可以发现，在使用批处理插入方法进行插入时：

- 在小数据规模下，三种插入方法效率相差不大。
- 在较大数据规模下，插入效率：`MySQL` 略大于 `FileIO` 略大于 `PostgreSQL` 。

依据上面两张图，我们可以得出结论：

- 在少量读写（批处理插入数据）的情况下，三种插入方法的效率处于同一数量级，略有区别。
- 在反复读写（逐条插入数据）的情况下，三种插入方法的效率均有所降低，使用`FileIO`方法效率下降尤为明显。

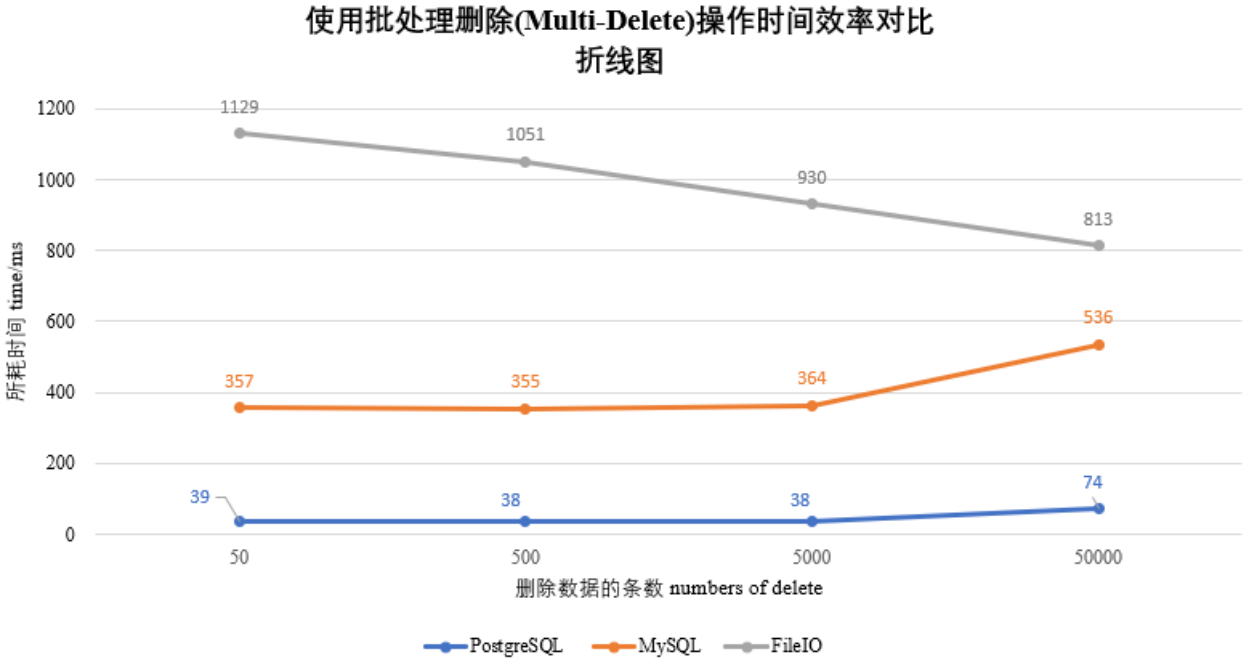
5.4.2 删除操作实验结果和结论



上图中，横坐标是测试删除数据的条数，纵坐标是所花时间 `单位：ms` 的对数。

图中数据可以很明显的发现，在使用逐条删除的朴素方法进行删除时：

- 无论总删除数据量如何，删除效率：`PostgreSQL` 远大于 `MySQL` 远大于 `FileIO`。



图中，横坐标是测试删除数据的条数，纵坐标是所花时间 `单位：ms` 。

图中数据可以发现，在使用批处理删除方法进行数据删除时：

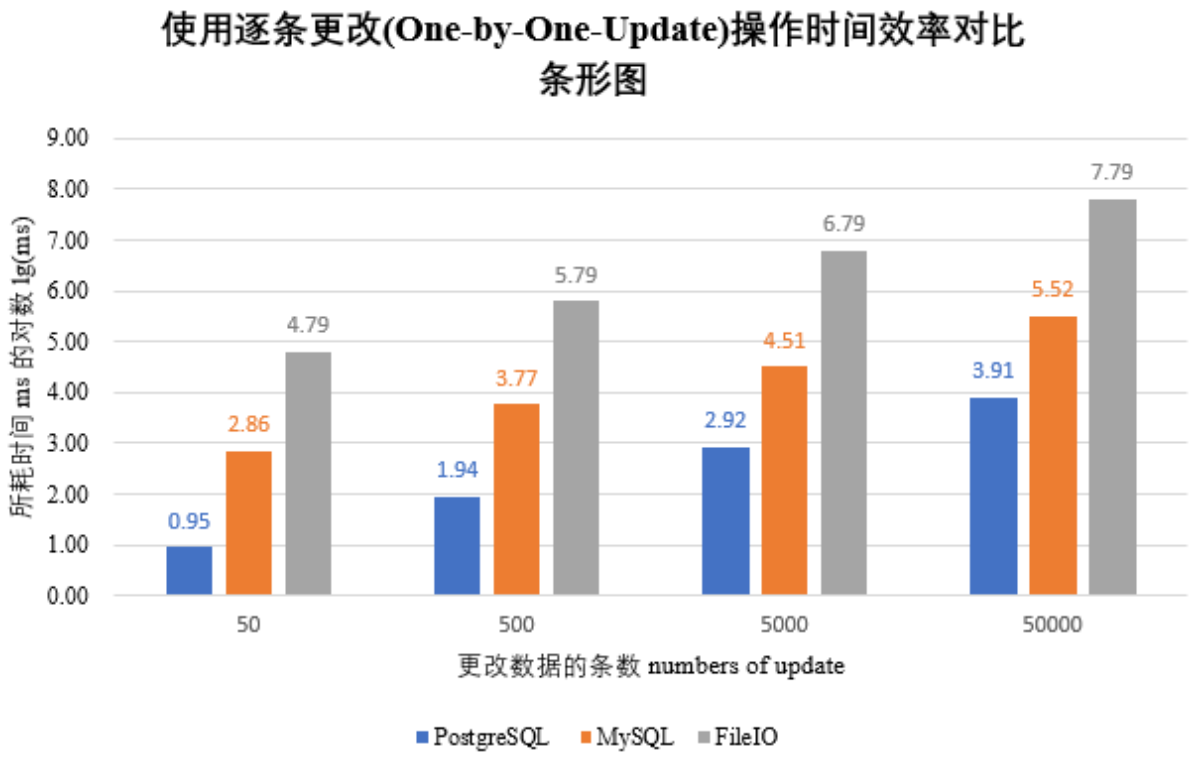
- 删除数据规模对所需删除的时间影响不大。
- 删除效率：`PostgreSQL` 略大于 `MySQL` 大于 `FileIO`。

依据上面两张图，我们可以得出结论：

- 在少量读写（批处理删除数据）的情况下，三种插入方法的效率处于同一数量级，略有区别。

- 在反复读写（逐条删除数据）的情况下，三种插入方法的效率均有所降低，使用`FileIO`方法效率下降尤为明显。

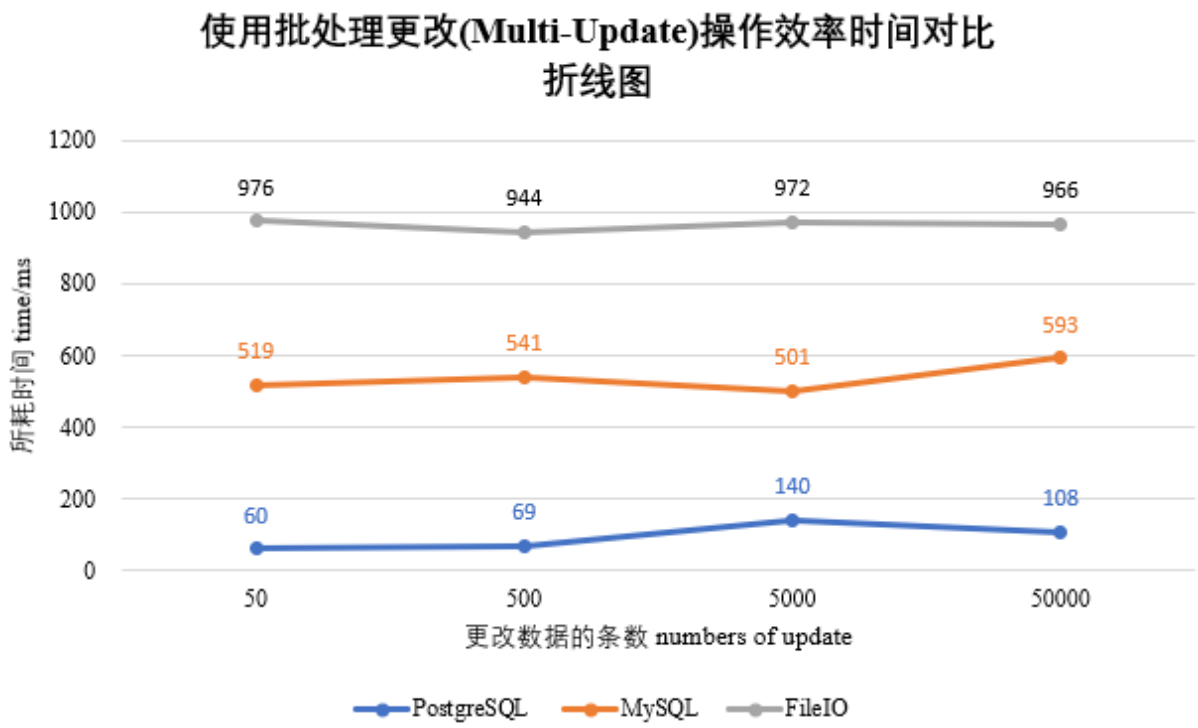
5.4.3 修改操作实验结果和结论



上图中，横坐标是测试更改数据的条数，纵坐标是所花时间 `单位：ms` 的对数。

图中数据可以很明显的发现，在使用逐条更改的朴素方法进行更改时：

- 无论总更改数据量如何，更改效率：`PostgreSQL` 远大于 `MySQL` 远大于 `FileIO`。



图中，横坐标是测试更改数据的条数，纵坐标是所花时间 `单位：ms` 。

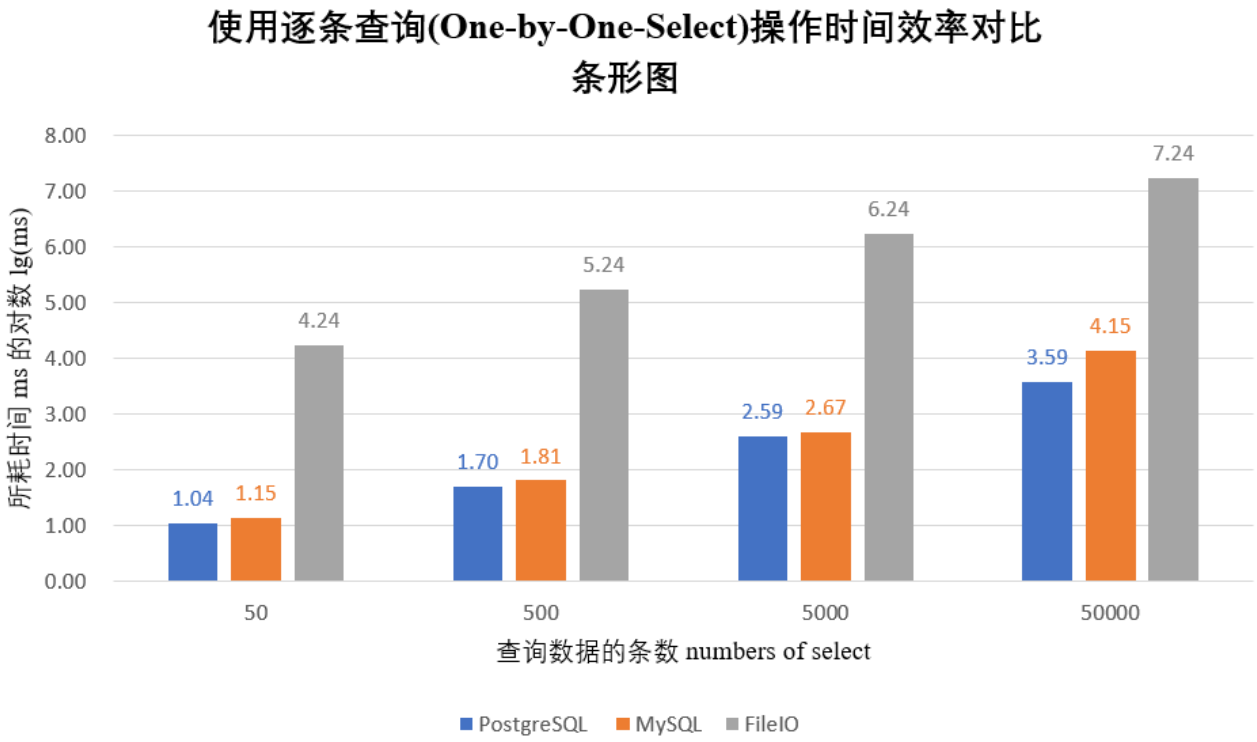
图中数据可以发现，在使用批处理更改方法进行数据更改时：

- 更改数据规模对所需更改的时间影响不大。
- 更改效率：`PostgreSQL` 略大于 `MySQL` 略大于 `FileIO`。

依据上面两张图，我们可以得出结论：

- 在少量读写（批处理更改数据）的情况下，三种插入方法的效率处于同一数量级，略有区别。
- 在反复读写（逐条更改数据）的情况下，三种插入方法的效率均有所降低，使用`FileIO`方法效率下降尤为明显。

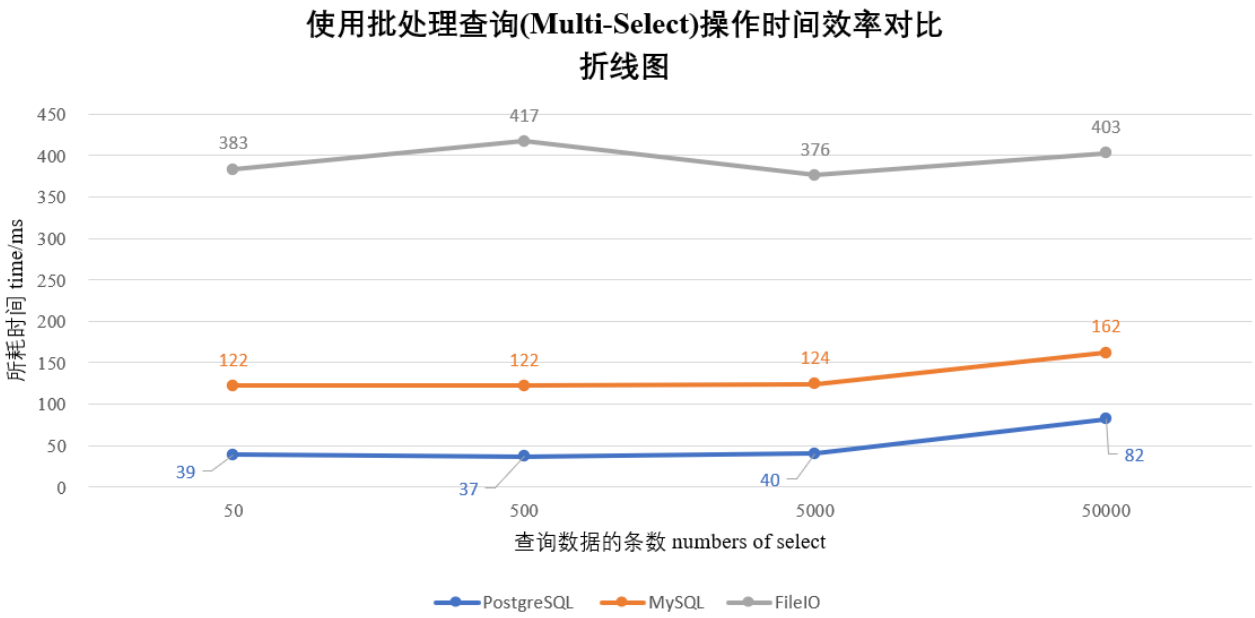
5.4.4 查询操作实验结果和结论



上图中，横坐标是测试更改数据的条数，纵坐标是所花时间 `单位：ms` 的对数。

图中数据可以很明显的发现，在使用逐条查询的朴素方法进行查询时：

- 无论总查询数据量如何，查询效率：`PostgreSQL` 略大于 `MySQL` 远大于 `FileIO`。



图中，横坐标是测试查询数据的条数，纵坐标是所花时间 `单位：ms` 。

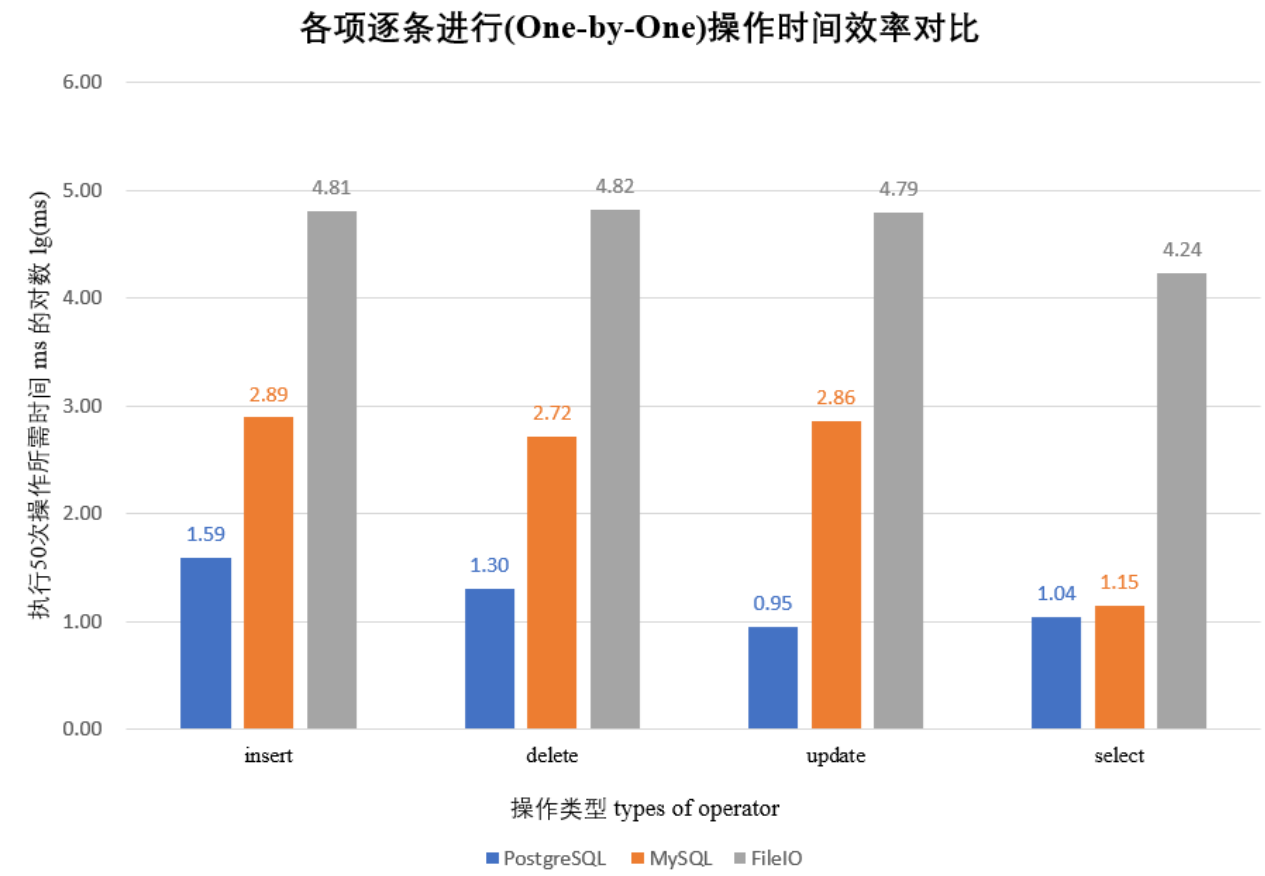
图中数据可以发现，在使用批处理更改方法进行数据查询时：

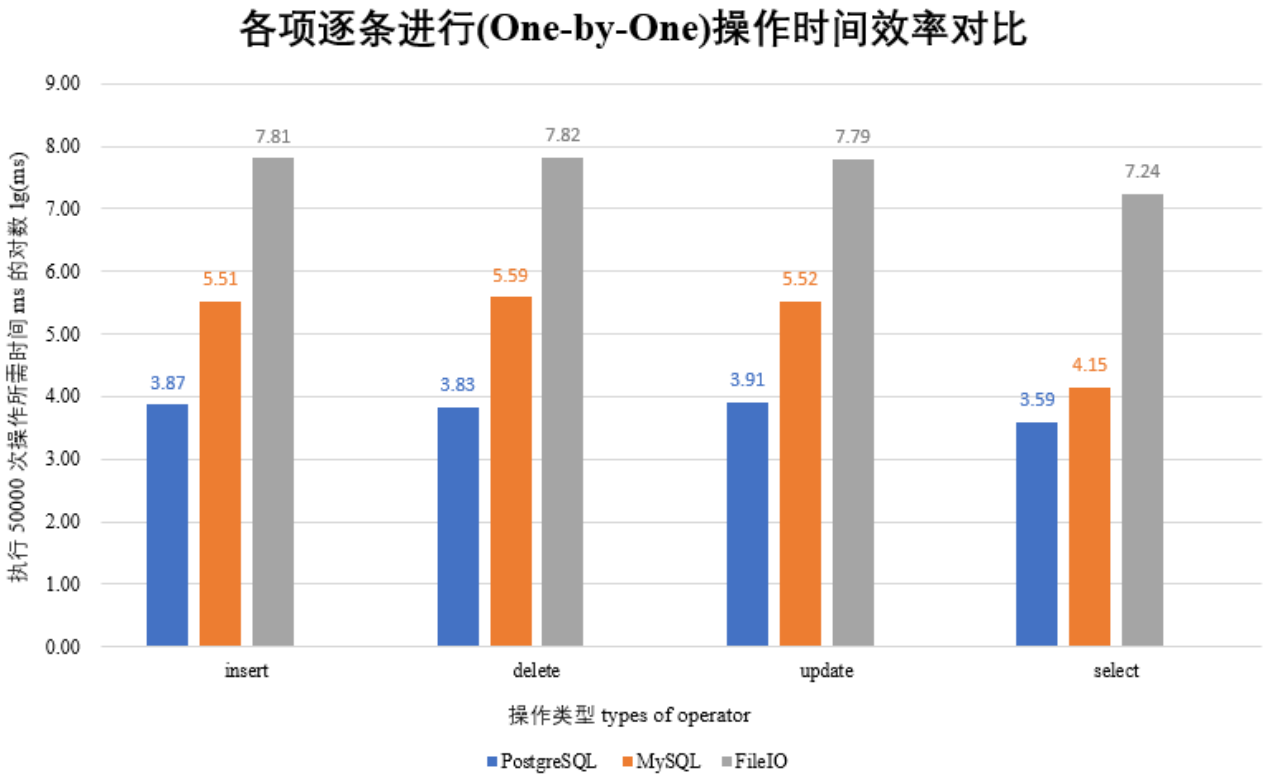
- 查询数据规模对所需更改的时间影响不大。
- 查询效率：`PostgreSQL` 略大于 `MySQL` 大于 `FileIO` 。

依据上面两张图，我们可以得出结论：

- 在少量读写（批处理查询数据）的情况下，三种插入方法的效率处于同一数量级，略有区别。
- 在反复读写（逐条查询数据）的情况下，三种插入方法的效率均有所降低，使用`FileIO`方法效率下降尤为明显。
- 使用`PostgreSQL` 与 `MySQL` 效率在少量读写和反复读写时虽然有区别，但整体效率高且稳定。

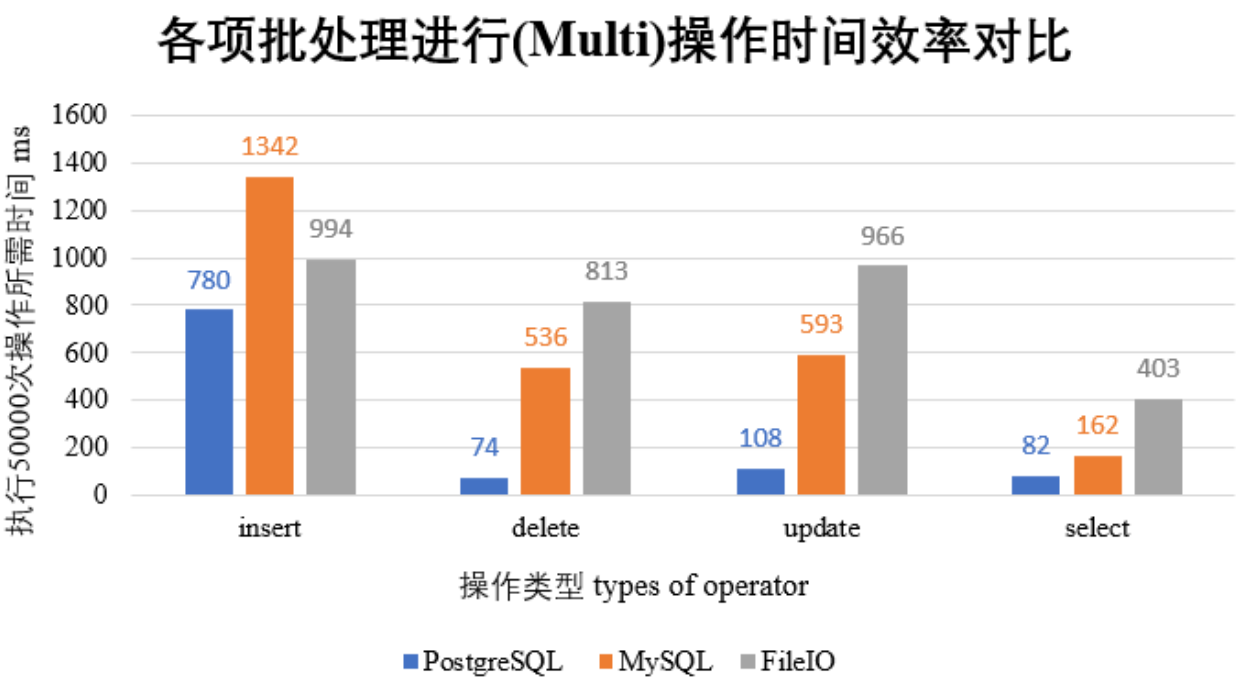
5.4.5 各个操作（增删改查）使用 `SQL` 和 文件读写耗时区别与结论

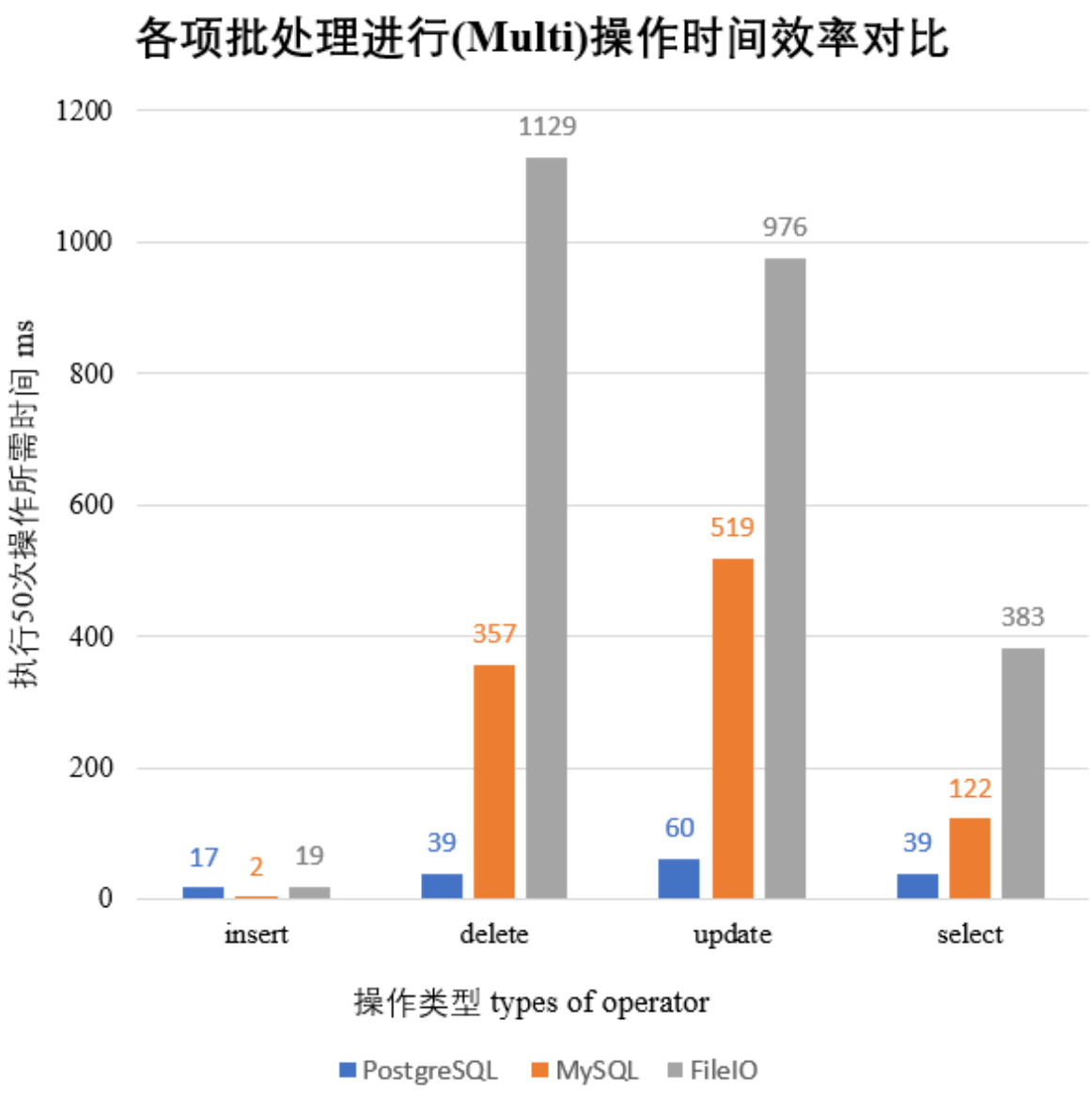




由上图我们可以得到如下结论：

- 逐次操作时，使用`PostgreSQL`、`MySQL`、`FileIO`进行各项操作，使用`SQL`进行操作无论是在大规模数据还是小规模数据下都能获得比`FileIO`远高的效率。
- 逐次操作时，使用`PostgreSQL`进行操作比使用`MySQL`进行操作效率略高。
- 逐次操作时，`insert`、`delete`、`update`三种操作效率低于`select`操作。





由上图我们可以得到如下结论：

- 批处理各项操作时，使用`PostgreSQL`、`MySQL`、`FileIO`进行各项操作，使用`SQL`进行操作无论是在大规模数据还是小规模数据下都能获得比`FileIO`较高的效率（数量级相差不大）。
- 批处理各项操作时，使用`PostgreSQL`进行操作比使用`MySQL`进行操作效率略高（在小数据规模时，可能由于测试时间较短，产生误差原因导致测得`MySQL`效率在`insert`操作高于`postgreSQL`）。
- 批处理各项操作时，在小数据规模下，`delete`、`update`、`select`三种操作效率低于`insert`操作。
- 批处理各项操作时，在大数据规模下，`insert`、`delete`、`update`三种操作效率低于`select`操作。

5.5 关于高级任务需求

5.5.1 高级任务需求 1

本数据库可以实现高效率并发查找出所有`container`的服务时间，且对于File IO也同样可以并发高效实现需求。

以`Java`语言为例，首先导出所有在`import_information`、`export_information`、`item`三个表数据到内存中，即采用`select * from table`语句（以用`Map`的数据结构进行存储为例），使用`Java`进行分析。

对于每一个`item`，获取其`export`时间和`import`时间，若`export`时间为空，则说明尚未进行出口，忽略该`item`。若`import`时间为空且有`export`时间，则设`import`时间为当前时间，用`import`时间减去`export`时间得到该`item`的海上时间，并累计到`item`的`container_code`。

最后，对每一个`container_code`，从`container`表中可获取到对应的`type`，以此实现需求。

值得一提的是，本数据库可以实现`import_information`、`export_information`、`item`三个表数据进行多线程并发查询，此在`MultiDataManager`的`init`方法中足以体现可实现性。

对于`File IO`，我们`File IO`采用的是`Json`文件类型，因此只需要读取在上述表对应的文件，采用类似的方式替代数据库在其中的作用即可实现，在这里，因为文件之间的独立性，我们同样可以采用多线程并发读取的方式。

对于`File IO`和数据库效率的对比，根据前文中，对于批处理`Select`操作的调查，我们不难估计，在这里数据库的效率会高于`File IO`。

5.5.2 高级任务需求 2

本数据库可以实现高效率并发查找出收检和发放包裹最多的快递员，且对于File IO也同样可以并发高效实现需求。

与上一个需求类似，我们导出所有在`retrieval_information`、`delivery_information`中的数据到内存中，同样使用`Java`进行分析。

对于每一个信息，我们调出其中的`courier_phone_number`，紧接着将其累计到该快递员所属公司，该信息所在城市，该快递员的收检和发放次数中，例如采用`Map`的数据结构进行处理，每次处理更新到该公司在该城市的快递员业绩最大值变量当中即可。同样，对于导出数据，我们也可以采用多线程并发的方式。

对于`File IO`，与上个需求类似，需求同样可以实现，此处不再过多赘述。

对于效率对比，与上个需求类似，数据库的效率会高于`File IO`，此处不再赘述。

5.5.3 高级任务需求 3

本数据库可以实现高效率并发查找出各个类型物品的出口税最低的城市，且对于`File IO`也同样可以并发高效实现需求。

与前两个需求不同的是，这个需求要求针对特定公司进行处理。我们导出所有在`item`表中，公司名为所选公司的数据，即使用`select * from item where company='company'`语句。

接着对于每个物品类型，选出在`export_information`中每个分类的所有数据，采用`select * from export_information where item like 'class-%'`语句。

然后对于每个导出的物品类型，以`Map`数据结构为例，将该项目的出口税累计到对应城市中，并且更新最小值，这样即可找出该公司每种物品类型的最小出口税城市。

对于`File IO`，与前两个需求类似，操作完全可以模仿并替代数据库实现需求。

对于效率对比，同样的，根据前文的分析，批处理`Select`操作，都是数据库的效率会高于`File IO`。

5.5.4 高级任务需求 4

本数据库具有高并发的特点。

`MultiThreadDataManager`中的多线程并发导入和样板多线程导出数据体现了这一特点。在前文中，我们已经尝试并分析过了高数据量的并发数据导入，此处不再赘述。

5.5.5 高级任务需求 5

本文中，对于每一个任务，我们基本上都有对比`PostgreSQL`和`MySQL`的性能。

在实验过程中，我们发现`MySQL`的功能更加繁杂，限制也更多，比如在链接方面，`MySQL`在连接时默认禁止了公钥访问方式，需要在`url`中设置对应的属性。`PostgreSQL`中，可以单独关闭某个表的外键等约束的触发器，而`MySQL`并不能直接实现此功能，只能通过设置变量`foreign_key_checks`实现。

但因为版本较高的原因，`MySQL`体现出了不稳定性，我们在实验中为了保证实验稳定，曾多次更换实验版本。同时我们也分析认定，`MySQL`因其复杂的功能，以及与`PostgreSQL`语法上的不同，在各类数据导入中，效率存在一定的测量误差。最终的结果是较新版本的`MySQL`的性能在大多数情况下不如当前最新版本的`PostgreSQL`。值得注意是，我们没有特别设置`MySQL`的数据库引擎，即没有采用比较常用的`InnoDB`，而是采用了其默认设置的引擎，这是因为我们想要测试两种数据库在设计者给出的默认环境中的区别。

5.5.6 常见问题及解答

写在最后，我们想要阐述一些此文读者可能产生的问题。

Q：为什么只使用数据库的`select`功能，而不使用数据库进行处理?

A：我们认为，数据库本该是存储数据的功能，我们应该更加的着重于存储和导出，而不是在数据库进行运算，对于运算，我们应该在更适合运算的使用端系统内存环境中进行，从而减少数据库无意义的性能消耗，以及对其它用户访问的影响。

Q：你们的数据库是否只能解决上述业务?

A：我们认为，我们的数据库在建表设计上，基本满足了所有并发查询和修改需求，实现了高效性，对于任何实际有需要的业务，只要数据取了出来，都可以使用各类编程语言解决。

Q：你们是如何实现多线程并发查询数据库的?

A：在`MultiDataManager`中，我们写了一个方法`sampleMultiThreadInput`，里面有着并发导出`export_information`、`import_information`表数据的样例代码。

6 附录：代码结构说明和运行须知

6.1 代码结构

`cn.sustech.edu.cs307` 包

- `Main.java`主类，负责生成加载`config.properties`配置文件、输出`debug`、`log`信息、生成`log.txt`以及导入`data.csv`。
- `PerformanceAnalysis.java` 用于脚本测试、数据采集、耗时分析。
- `datamanager`包 含有全部的导入数据脚本。
    1. `DataManager.java` 抽象类所有的脚本类继承于此，所有脚本包含方法`init(List<DataRecord>)`表示导入数据。
    2. `SimpleDataManager.java`基本脚本，具体内容在报告中有所说明。
    3. `FastDataManager.java` 快速导入脚本，具体内容在报告中有所说明。
    4. `MultiThreadDataManager.java` 多线程并发导入脚本，具体内容在报告中有所说明。
    5. `FileDataManager.java` 实现`File IO`导入的脚本，使用`Alibaba`的`Fast Json`库实现`Json`数据存储。
    6. `DataRecord.java`用于存储单条`item`信息的类
- `sqlconnector`包 含有所有的数据库连接工具
    1. `SQLUtils.java` 生成数据库连接器实例的工厂类
    2. `SQLConnector.java` 抽象类，所有的数据库链接器都继承与该类
    3. `MySQLConnector.java` `MySQL`数据库连接器
    4. `PostgreSQLConnector.java` `PostgreSQL`数据库连接器
- `utils`包 一些工具类
    1. `CalendarUtils.java`负责`debug`、`log`信息等日期相关处理需求的工具类

`pom.xml` `Maven`项目的构建文件

`config.properties`在第一次运行程序后会自动生成的配置文件，包含运行所需要的一些信息，具体在运行须知有所说明。

6.2 运行须知

- 本代码项目为`Apache Maven`项目，需要开发者提前预配好`Apache Maven`环境，并利用`pom.xml`文件构建项目，下载依赖，以及导入`src`源码包。
- 本代码项目包含`config.properties`文件，若无生成，会在第一次执行时自动生成该文件，`config.properties`文件包含有如下词条：
    1. `postgresql-host`，`postgresql-port`，`postgresql-user`，`postgresql-password`，`postgresql-database`，分别对应`PostgreSQL`的主机名、端口，用户名称以及密码，数据库名
    2. `mysql-host`，`mysql-port`，`mysql-user`，`mysql-password`，`mysql-database`，分别对应`MySQL`的主机名、端口，用户名称以及密码，数据库名
    3. `data-file-path` 数据文件绝对路径
    4. `table-maker-file-path` 建表`SQL`文件的绝对路径
    5. `table-dropper-file-path` 删除表的sql文件的绝对路径（用于反复测试时自动建表删表）
    6. `file-storage-directory` `File IO`的文件存储目录，也就是该文件夹的绝对路径
- 本代码在测试完成后，会自动生成`log.txt`文件，包含数据的测试结果

- 本代码在运行过程中，遇到数据库链接出错时，请检查环境配置是否准确，数据库是否正常开启，必要时重启数据库服务。遇到其它报错时，请根据`Java`的报错栈信息溯源，并且分析出问题，如文件缺失，程序不具有访问权限等。必要时可自行修改代码。

6.3 引用

1. MySQL官方文档 8.2.5.1：https://dev.mysql.com/doc/refman/8.0/en/insert-optimization.html。