

CS213 数据库原理 Project 2 报告

2024 年 1 月 4 日

肖翊成

xiaoyc2022@mail.sustech.edu.cn

12210414

1 数据库设计

数据库设计过程中需要考虑**实体 (Entities)** 和 **关系 (Relations)**。数据库设计的目标是为了减少数据库调用过程中的冗余。

1.1 ER 图设计

我使用了 Chow's foot 的方法去绘制 ER 图，这种 ER 图的好处是在设计过程中就可以明确每个属性在数据库中的类型，并且清晰的表现出关系的多对多，一对多等关系，便于后续表的创建。

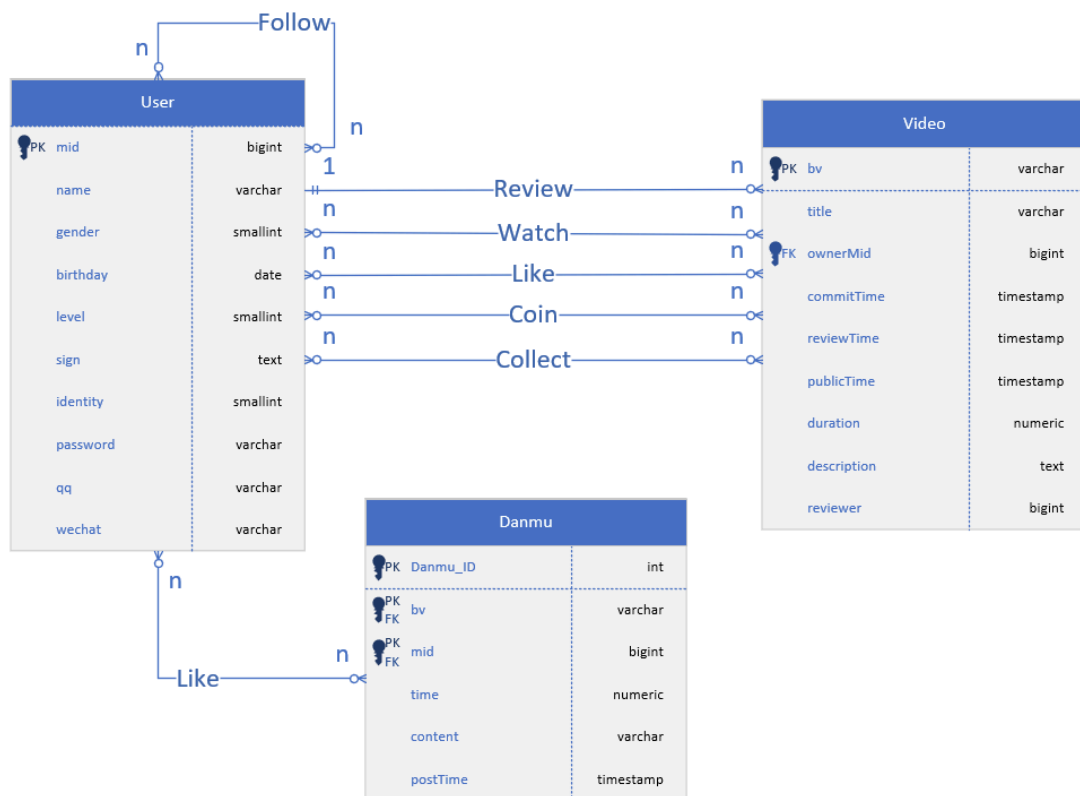


图 1 ER graph

1.2 Datagrip 可视化

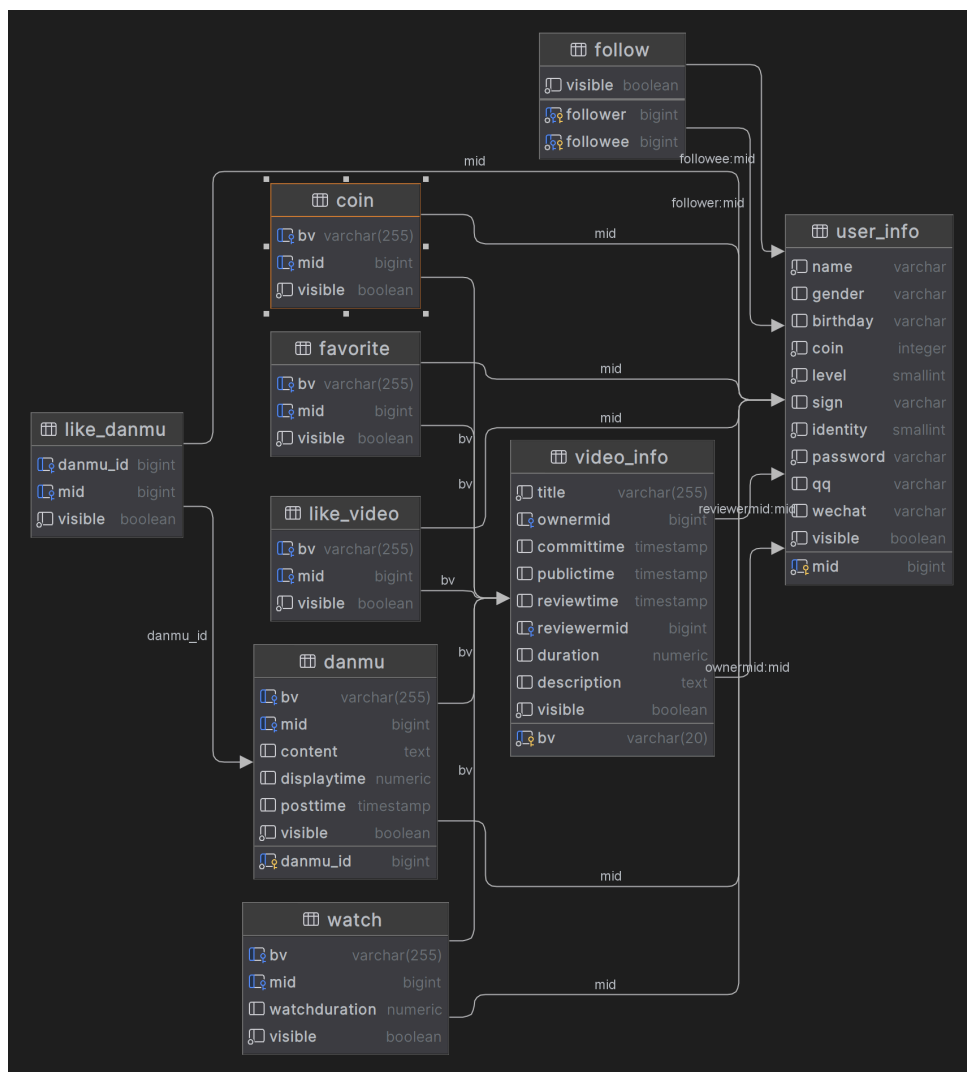


图 2 Data Grip 可视化表图

1.3 设计细节

基于上述目标和陈述，我的数据库分为以下几张表: user_info, video_info, danmu, watch, like_video, coin, favorite, like_danmu, follow. 对于 ER 图中多对多的关系，我都对其建立了单独的表，表的设计过程中严格遵守了范式原理。

1.3.1 第一范式

第一范式要求**最小化数据单元**，不允许在一条记录的一条属性中出现大于一条的内容。在 Java 的 UserRecords, VideoRecords 和 DanmuRecords 类中, following, like, coin, favorite, viewerMids, viewTime 和 likedBy 都是以数组的方式进行存储，我对所有这些数据都新建了关系表来进行存储: follow, like_video, coin, favorite, watch 和 likedBy.

1.3.2 第二范式

第二范式要求键值必须独立，**不能出现键值之间有相互依赖的关系**。在 user_info, video_info 和 danmu 三张表中，每一条记录都是依赖与一个独立的键值 (MID, BV, Danmu_id)。对于其余的 6 张表，键值由 MID 和 BV (在 like_danmu 中还包括 danmu_id) 等外键构成，用于表示这是一个与 video_info 和 user_info 相关的关系表. 由于 BV 和 MID (还有 danmu_id) 都是互相独立的，因此键值之间也不存在依赖关系。

1.3.3 第三范式

第三方是要求**非键值之间不能存在相互依赖的关系**。在 `user_info` 中, 属性 `name`, `gender`, `birthday`, `coin`, `level`, `sign`, `identity`, `password`, `qq`, `wechat` 互不依赖, 只依赖于键值 `MID`。在 `video_info` 中, 属性 `title`, `ownerMID`, `commitTime`, `publicTime`, `reviewTime`, `reviewerMID`, `duration` 互不依赖, 只依赖于键值 `BV`。在 `danmu` 中, `BV` 和 `MID` 依赖于 `danmu_id` 因为每一条弹幕只能由一个人在一个视频中发送。其他属性 `content`, `displayTime`, `postTime` 都只依赖于 `danmu_id` 且互不依赖。

1.4 用户创建和权限管理

我的用户创建和权限管理如下:

```
CREATE ROLE sustc LOGIN PASSWORD '000000';
GRANT CONNECT ON DATABASE sustc TO sustc;
GRANT SELECT, UPDATE, INSERT on ALL TABLES IN SCHEMA public to sustc;
GRANT EXECUTE ON ALL FUNCTIONS IN SCHEMA public TO sustc;
ALTER TABLE * OWNER TO sustc;
```

这遵循了最小权限原理, 连接只允许 Java 程序连接 `sustc` 用户并且只有对数据进行 `SELECT`, `UPDATE` 和 `INSERT` 的权限。但是在 Benchmark 测试中, 我给予了 `sustc` 用户更多的权限因为性能需要。

- `DELETE`: 在 `deleteVideo` 和 `deleteAccount` 过程中, 级联删除往往能带来比 `Visible` 更高的效率。
- `CREATE/DROP`: 在数据导入的过程中, 我们可以通过先关闭外键和索引来提高插入速率, 由于提供的数据都是合法的 (仅限在本次 Project 中是这样), 可以在插入过程完成后再添加外键和索引, 这样能提高运行的效率。如图所示:
- 在导入数据完成后添加外键和索引:

Test Step	# Case	# Passed	Std. Time (ms)	Elapsed Time (ms)	Correctness	Performance
1	N/A	N/A	65832	46960	N/A	1.00

- 不删除外键和索引:

Test Step	# Case	# Passed	Std. Time (ms)	Elapsed Time (ms)	Correctness	Performance
1	N/A	N/A	65832	77195	N/A	0.85

如果不适用 `DELETE`, 视图和 `visible` 列可以用于保证数据的完整性, 并防止数据被恶意删除。我们可以通过 `UPDATE ON <table_name> SET visible = false WHERE <condition>` 来设置数据的可见度。并让用户通过访问视图来访问数据。

```
Execute | JSON | Copy
CREATE OR REPLACE VIEW user_info_view AS
| SELECT * FROM user_info WHERE visible = true;

Execute | JSON | Copy
CREATE OR REPLACE VIEW video_info_view AS
| SELECT * FROM video_info WHERE visible = true;

Execute | JSON | Copy
CREATE OR REPLACE VIEW danmu_view AS
| SELECT * FROM danmu WHERE visible = true;
```

图3 视图创建案例 (不完整)

2 API 基础实现

下图是在 Benchmark OJ 中 API 的性能结果。

Test Step	# Case	# Passed	Std. Time (ms)	Elapsed Time (ms)	Correctness	Performance
1	N/A	N/A	65832	46960	N/A	1.00
2	20	20	376	483	1.00	0.78
3	61	61	34	73	1.00	0.47
4	56	56	9	7	1.00	1.00
5	43	43	1729	1719	1.00	1.00
6	11	11	925	574	1.00	1.00
7	27	27	3311	1081	1.00	1.00
8	142	142	29219	6225	1.00	1.00
9	382	382	14	22	1.00	0.64
10	1016	1016	40	62	1.00	0.65
11	500	500	134	306	1.00	0.44
12	500	500	N/A	N/A	1.00	N/A
13	511	511	148	152	1.00	0.97
14	686	686	189	211	1.00	0.90
15	644	644	156	185	1.00	0.84
16	184	184	42	18	1.00	1.00
17	273	273	46	808	1.00	0.06
18	184	184	25	14	1.00	1.00
19	35	35	11	13	1.00	0.85
20	6	6	29	270	1.00	0.11
21	55	55	362	449	1.00	0.81
22	199	199	114	127	1.00	0.90
23	87	87	18	20	1.00	0.90

正确率的平均值为 1.00，速率的平均值为 0.79. 可以在 [这里](#)查看这次结果。

3 进阶 API 实现

3.1 更快速

3.1.1 Batch 批处理

在 Java 中，通过 `addBatch()` 和 `executeBatch()` 可以对数据进行批处理，提高运行的效率。但是我在测试的过程当中发现，如果 batch 过大，可能会导致栈内资源不足，以及在最后 `executeBatch()` 的过程中效率也会非常低，因此需要手动设置 count，在到达一定的 count 之后就执行一次 `executeBatch()`，这样可以进一步提高数据导入的效率，示例如下：

```

int count = 0;
try(PreparedStatement follow_stmt = conn_1.prepareStatement(sql_follow)){
    for (int i = 0; i < userRecords.size() / 4; i++) {
        for (int j = 0; j < userRecords.get(i).getFollowing().length; j++) {
            follow_stmt.setLong(parameterIndex:1, userRecords.get(i).getMid());
            follow_stmt.setLong(parameterIndex:2, userRecords.get(i).getFollowing()[j]);
            follow_stmt.addBatch();
            count++;
            if(count % batch == 0){
                follow_stmt.executeBatch();
                conn_1.commit();
            }
        }
    }
    follow_stmt.executeBatch();
    conn_1.commit();
    follow_stmt.close();
    System.out.println(x:"[Thread 1] Follow inserted");
}

```

图 4 批处理案例（不完整）

Batch 参数大小	5000	10000	5000	50000	100000	1000000
运行时间(ms)	60799	56091	66366	67725	70960	76387

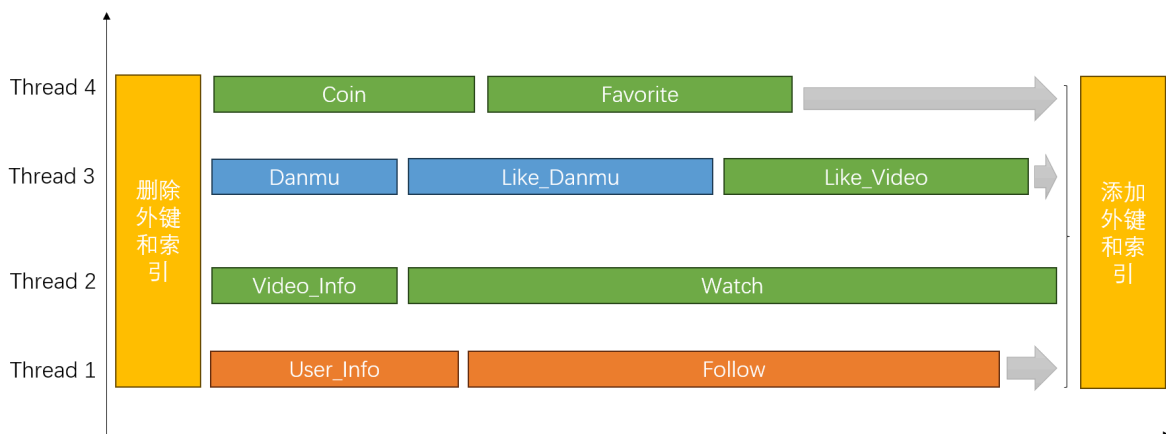
表 1 本地运行运行时间测试

不难看出将 Batch 的大小设置太大和太小均会导致批处理的性能变弱，因此选择合适的批处理大小非常重要，以下测试均是在 Batch 大小为 50000 的情况下进行运行。

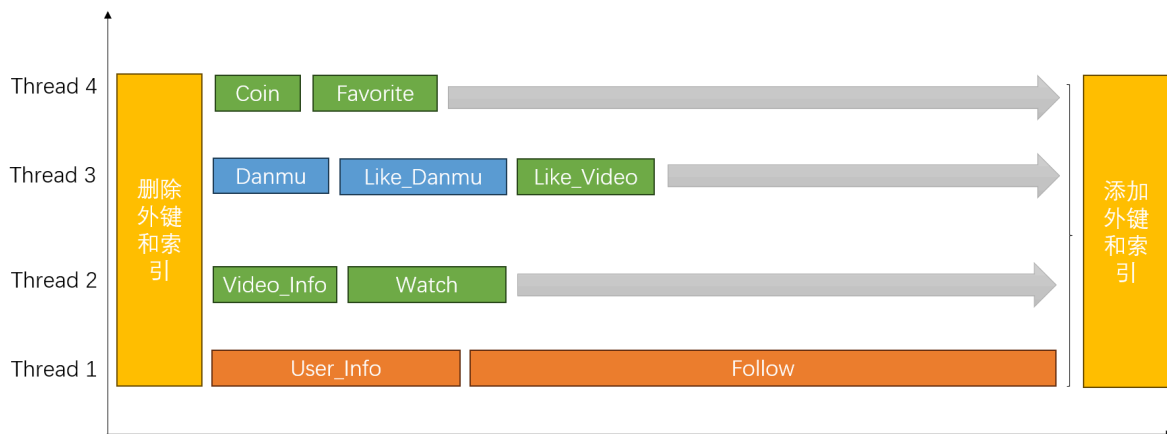
3.1.2 Thread 多线程

在数据导入过程中，顺序插入数据花费了大量时间，因为最大的表包含将近 600 万条记录（小数据集）和大约 800 万条记录（大数据集）。在我的第一个实现版本中（很抱歉，我忘记保存记录了，因此没有可运行的例子可以展示），插入大数据集的所有数据大约需要 4 分钟，插入小数据集的时间大约是 1 分半钟。在用 4 个线程实现导入过程之后，大数据集的耗时可以降低到 1 分钟（没有索引），小数据集的耗时降低到 50 秒（在插入后建立索引）。我的实现方式也随着时间变化：

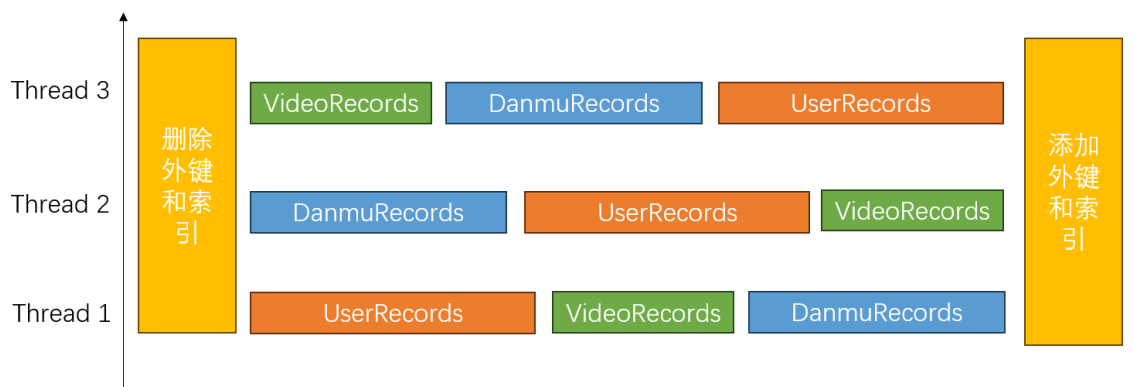
- 在最初多线程的设计中，我是基于数据大小进行设计的。我提前预估了每张表的大小，按照大小将不同表的插入任务分配到了不同的线程，基于大数据集我的设计如下：



但是显然这样的表存在很大的问题，其中一个就是复用性问题，在小数据集中，只有 follow 表仍然保持比较大的数据量，这就会让导入数据的过程变得不平衡。如下图所示：



- 第二次插入过程中，我将 follow 表的插入过程分别插入到了四个线程之中，需要注意的是，Postgresql 有在插入过程中的 lock 机制，如果在同一时间对数据进行插入（执行 Batch 指令）会导致数据丢失。因此最后为了防止数据冲突，我将 userRecords、videoRecords 和 danmuRecords 均平分分到三个线程，通过调控不同线程的插入顺序来避免数据冲突，提高了插入过程的复用性。



在 Benchmark OJ 上，最高的插入效率可以提高到平均时间 50s 左右。（如果除去索引和外键的建立时间大约平均时间 30s 左右）

3.1.3 索引建立

索引的建立对于性能的提升是非常显著的，简单来说，索引的建立是用空间来换取时间的一种平衡手段。创建索引会占据大量空间。

	user info	video info	danmu	coin	like video	follow	like danmu	favorite	watch
table size	6336 kB	64 kB	72 kB	26 MB	28 MB	297 MB	600 kB	25 MB	50 MB
index size	3728 kB	48 kB	96 kB	21 MB	23 MB	464 MB	616 kB	20 MB	40 MB
total size	10064 kB	112 kB	168 kB	47 MB	50 MB	760 MB	1216 kB	45 MB	91 MB

表 2 数据库表和索引大小统计

对于每一张表，根据主要查询索引的不同，我建立了多个索引：

```

CREATE index user_index ON user_info (MID);
CREATE index user_index_qq ON user_info (QQ);
CREATE index user_index_wechat ON user_info (Wechat);
CREATE index follow_index ON follow (follower, followee);
CREATE index follow_index_followee ON follow (followee);
CREATE index video_index ON video_info (BV);
CREATE index video_index_owner ON video_info (ownerMID);
CREATE index danmu_index ON danmu (Danmu_ID, BV, MID);
CREATE index danmu_index_bv ON danmu (BV, MID);
CREATE index danmu_index_MID ON danmu (MID);
CREATE index watch_index ON watch (BV, MID);
CREATE index watch_index_mid ON watch (MID);
CREATE index like_video_index ON like_video (BV, MID);
CREATE index like_video_index_mid ON like_video (MID);
CREATE index coin_index ON coin (BV, MID);
CREATE index coin_index_mid ON coin (MID);
CREATE index favorite_index ON favorite (BV, MID);
CREATE index favorite_index_mid ON favorite (MID);
CREATE index like_danmu_index ON like_danmu (Danmu_id, MID);
CREATE index like_danmu_index_mid ON like_danmu (MID);

```

以最大表 follow 为例，我们可以看到建立索引对于信息的查询的速率优化效果。我们测试

```

EXPLAIN ANALYSE SELECT followee FROM follow WHERE follower = <mid_input>
INTERSECT
SELECT follower FROM follow WHERE followee = <mid_input>;

```

运行时间从原来的 200ms 左右快速下降到只需要 1ms 左右的查询时间。

0	Filter	0	Filter
1	HashSetOp Intersect (cost=1000.00.70037.90 rows=156 width=12) (actual time=273.602.277.671 rows=3 loops=1)	1	HashSetOp Intersect (cost=5.65.630.26 rows=156 width=12) (actual time=1.797.1.801 rows=3 loops=1)
2	-> Append (cost=1000.00.70036.52 rows=550 width=12) (actual time=6.620.277.494 rows=473 loops=1)	2	-> Append (cost=5.65.628.88 rows=550 width=12) (actual time=0.105.1.743 rows=473 loops=1)
3	-> Subquery Scan on "SELECT* 2" (cost=1000.00.70010.53 rows=157 width=12) (actual time=6.618.277.233 rows=147 loops=1)	3	-> Subquery Scan on "SELECT* 2" (cost=5.65.606.89 rows=157 width=12) (actual time=0.104.1.490 rows=147 loops=1)
4	-> Gather (cost=1000.00.70008.96 rows=157 width=8) (actual time=6.618.277.188 rows=147 loops=1)	4	-> Bitmap Heap Scan on follow (cost=5.65.605.32 rows=157 width=8) (actual time=0.103.1.472 rows=147 loops=1)
5	Workers Planned: 2	5	Recheck Cond: (follower = 917516)
6	Workers Launched: 2	6	Heap Blocks: exact=147
7	-> Parallel Seq Scan on follow (cost=0.00.68993.26 rows=65 width=8) (actual time=3.572.165.797 rows=49 loops=1)	7	-> Bitmap Index Scan on follow_index_followee (cost=0.00.5.61 rows=157 width=0) (actual time=0.058.0.058 rows=1)
8	Filter: (followee = 917516)	8	Index Cond: (followee = 917516)
9	Rows Removed by Filter: 1986208	9	-> Subquery Scan on "SELECT* 1" (cost=0.43.19.24 rows=393 width=12) (actual time=0.154.0.222 rows=326 loops=1)
10	-> Subquery Scan on "SELECT* 1" (cost=0.43.23.24 rows=393 width=12) (actual time=0.021.0.164 rows=326 loops=1)	10	-> Index Only Scan using follow_index on follow_follow_1 (cost=0.43.15.31 rows=393 width=8) (actual time=0.153.0.1)
11	-> Index Only Scan using pk_follow_always on follow_follow_1 (cost=0.43.19.31 rows=393 width=8) (actual time=0.020)	11	Index Cond: (follower = 917516)
12	Index Cond: (follower = 917516)	12	Heap Fetches: 0
13	Heap Fetches: 0	13	Planning Time: 0.678 ms
14	Planning Time: 0.172 ms	14	Execution Time: 1.858 ms
15	Execution Time: 277.722 ms		

以下陈列了一些在加入索引之后效率提升较大的 API。

Benchmark ID	Benchmark Content	Performan Without Index	Performance With Index (avg)	Performance With Index (Best)
10	sendDanmu	0	0.47	0.65
11	getUserInfo	0	0.33	0.44
13	coinVideo	0.13	0.74	0.97
14	likeVideo	0.12	0.67	0.9
15	collectVideo	0.1	0.74	0.84
18	updateVideoInfo	0.28	1	1
19	reviewVideo	0.42	0.73	0.85
22	deleteAccount	0	0.81	0.9
23	follow	0.08	0.67	0.9
Average		0.415	0.675909091	0.787272727

表 3 Performance Comparison Table

3.1.4 视图创建

在使用 API 的过程中, 对常用表格建立视图可以提高同时大量查询的效率, 因为不需要对表格进行反复 join 和筛选的操作。我根据 API 调用时的需要创建了以下视图:

```
--用于展示已经经过审核的并且发布过的视频, 在很多 API 中比如 generalRecommendation() 等均有调用到
CREATE OR REPLACE VIEW public_video AS
    SELECT video_info.bv AS bv
    FROM video_info
    where reviewtime IS NOT NULL
    AND publictime < NOW()
    AND reviewtime < NOW();
--用于展示每一个视频的播放量, 在计算一个视频的 Popularity 得分时需要反复调用到每个视频的播放量, 而在数据库
--中 watch 不是以每个视频的总数存储, 而是陈列了观看的具体数据
CREATE OR REPLACE VIEW count_watch AS
    SELECT watch.bv AS bv, count(watch.bv) AS watch_count
    FROM watch
    GROUP BY watch.bv;
--用于计算完播率, 在 Recommendation 中常用到
CREATE OR REPLACE VIEW watch_rate AS
SELECT video_info.bv AS bv, avg(watch.watchduration/video_info.duration) AS watch_rate
    FROM video_info, watch
    WHERE video_info.bv = watch.bv
    GROUP BY video_info.bv;
```

视图的创建提高了像 Recommendation 中的推荐指数计算的效率, 能够有更好的性能呈现。

3.2 易维护

3.2.1 在数据库中实现方法

对比在 Java 中单独写出 SQL 指令, 我通过 SQL 文件在数据库中实现方法, 这样在 JAVA 中只需要输入指令 SELECT function_name(input variable(s))。在对 API 进行维护的时候非常方便, 因为不需要对后端 Java 程序进行频繁的修改, 而仅需要修改数据库中的函数内容即可。通过 CREATE OR REPLACE FUNCTION function_name(input variable(s)) 可以对函数进行快速修改和调整。同时数据库会对反复调用的函数进行优化, 以提高查询的效率。

3.2.2 将常用方法单独建立

通过将常用方法单独写出, 可以有更好的复用性。比如对于用户 AuthInfo 的认证, 对于视频 BV 的验证, 以及 MID 和 BV 的生成函数, 我均单独列出, 这样当我需要对其中的部分进行的调整, 可以单独调整常用的方法。我此次创建的常用函数有:

```
--Function 1: To authenticate whether a user is valid and return the mid
CREATE OR REPLACE FUNCTION is_auth_valid(mid BIGINT, password VARCHAR, qq VARCHAR,
wechat VARCHAR) RETURNS BIGINT;
--Function 2: To authenticate whether a bv is available in public and return the bv.
CREATE OR REPLACE FUNCTION is_bv_valid_for_users (search_bv VARCHAR) RETURNS VARCHAR;
CREATE OR REPLACE FUNCTION is_bv_valid_for_superusers (search_bv VARCHAR) RETURNS
VARCHAR;
--Function 3: To determine whether a insertion time for danmu is valid for a video.
CREATE OR REPLACE FUNCTION time_valid(bv varchar, timeStart numeric, timeEnd numeric)
RETURNS INTEGER;
--Function 4: Is SuperUser
```



```

CREATE OR REPLACE FUNCTION is_super_user(mid BIGINT) RETURNS BOOLEAN;
--Function 5: Is video VALID
CREATE OR REPLACE FUNCTION is_video_valid(mid BIGINT, title VARCHAR) RETURNS BOOLEAN;
--Function 6: Do user exist?
CREATE OR REPLACE FUNCTION is_user_exist(mid BIGINT) RETURNS BOOLEAN;
--Function 7: is register info valid?
CREATE OR REPLACE FUNCTION is_reg_valid(name varchar, qq varchar, wechat varchar)
RETURNS INTEGER;
--Function 8: Generate the MID
CREATE OR REPLACE FUNCTION generate_mid () RETURNS BIGINT;
--Function 9: Generate BV function:
CREATE OR REPLACE FUNCTION generate_bv () RETURNS VARCHAR;

```

3.3 更安全

3.3.1 用户权限设置

更详细的内容可以参考 小节 1.4 中的内容，用户权限设置可以防止数据被恶意删除，以及防止用户对数据库的物理存储设置进行修改。

3.3.2 对密码进行加密

本文使用了拓展包 pgcrypto 对数据的密码进行了加密处理，防止用户的密码被窃取，增强了数据的安全性。通过 SHA256 的哈希算法对数据密码进行加密映射。本加密方式的嵌入也体现了在小节 3.2.2 中提及的易维护性，对于数据的加密和验证只需要添加下面代码以及在 is_auth_valid 方法中对密码数据进行一次映射即可。

```

CREATE EXTENSION IF NOT EXISTS pgcrypto;

CREATE OR REPLACE FUNCTION encrypt_password()
RETURNS TRIGGER AS $$
BEGIN
    -- Encrypt the password using SHA256
    NEW.password := encode(digest(NEW.password, 'sha256'), 'hex');
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER before_user_insert
BEFORE INSERT ON user_info
FOR EACH ROW
EXECUTE FUNCTION encrypt_password();

```

3.3.3 审计

审计是在设置 visible 表之外保证数据安全的另外一个方式，可以通过以下方式建立审计表格：

```

CREATE TABLE audit_log (
    audit_log_id SERIAL PRIMARY KEY,
    timestamp TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    username VARCHAR(255),
    action_type VARCHAR(50), -- e.g.UPDATE, DELETE
    table_name VARCHAR(255),
    query_text TEXT,
    client_ip VARCHAR(50),

```

```

        user_agent VARCHAR(255)
    );

CREATE OR REPLACE FUNCTION audit_trigger_function()
RETURNS TRIGGER AS $$
BEGIN
    INSERT INTO audit_log (username, action_type, table_name, query_text, client_ip,
user_agent)
    VALUES (
        current_user,
        TG_OP,
        TG_TABLE_NAME,
        current_query(),
        inet_client_addr(),
        inet_client_port()
    );
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER audit_trigger_user_info
AFTER INSERT OR SELECT OR UPDATE OR DELETE ON user_info
FOR EACH ROW EXECUTE FUNCTION audit_trigger_function();

```

通过这样的方式，可以记录用户对数据库的更新，删除操作。但是由于加入审计表后数据性能会差距较大，并且由于使用了 PreparedStatement 后，current_query()函数没有办法获取到每次操作的具体数据（会有对应的 query 信息但是关键信息由? 代替），审计表的方式并没有最后采用，只是在此陈列出可以通过该方式增强数据库的安全性。

4 总结

在本次项目中我接触了 SpringBoot, Gradle , Hikari Pool 等等以前从来没用过的框架、工具，实现了以前觉得很遥远的各种 API 接口，也算是很有成就感，收获颇丰。

有点遗憾的就是感觉对数据库底层原理仍然不甚了解（可能只是知道用了什么 B+树啊什么的），只能说现在大概了解了数据库使用层面的各种设置有些什么。（数据库还是真的是神奇的东西，怎么做到优化这么好的...）

但此次项目耗时时间比较长，很多时候是因为对于 API 的需求阅读不仔细，所以花了很多时间在正确率的提高上面，最后留给优化的时间非常有限，也有很多东西没有来得及探索：比如有没有可能对 Query 进行进一步优化，有没有可能使用 GPU 加速，线程池的工作原理，如何能建立一种更加稳定的并行插入方式，而不会导致数据被吞（尝试使用 parallel stream 来插入数据，但是结果就是 90% 以上的战损率于是作罢了），还有不同数据结构的 index 在不同表的性能表现有何不同，只能说最后把能想到的优化都尝试了一下。¹

¹完结撒花❀！感谢于老师和助教们这一个学期以来的付出！