数据库 Project2 report

12211829 陈峙安

写在前面的话：

这次的大作业作为数据库课程总结性的一次单人 project，我收获颇多，也投入了大量心力尽可能优化、强健我的代码。但随着不断深入和学习交流，我发现能够优化和思考的方向实在太多太多，我现在也不敢说自己在哪一细碎方向达到了前列。

本 report 将从三个方面介绍我的项目工作，分别是数据库架构的设计、API 的实现和优化，以及剩余工作和还能优化的方向。

一、数据库架构

出于快速实现一些功能的想法，几乎所有“主体”之间可能出现的关系种类都以表格的形式被储存至数据库里，具体的原因和思路将会在下文中详述，包括不同权限和角色的建立。

However, in some cases, foreign key can be a problem
- Especially in big data processing applications
- E.g., Alibaba Java Coding Guideline (阿里巴巴Java开发手册)

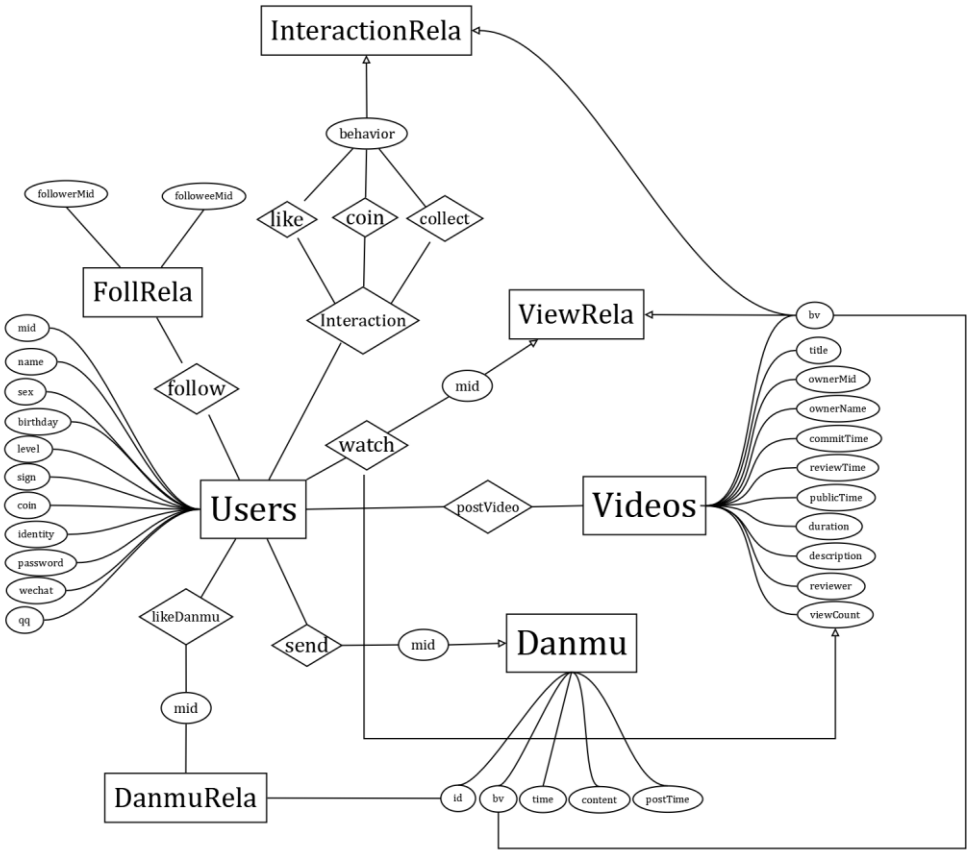
（三）SQL 语句

6. 【强制】不得使用外键与级联，一切外键概念必须在应用层解决。

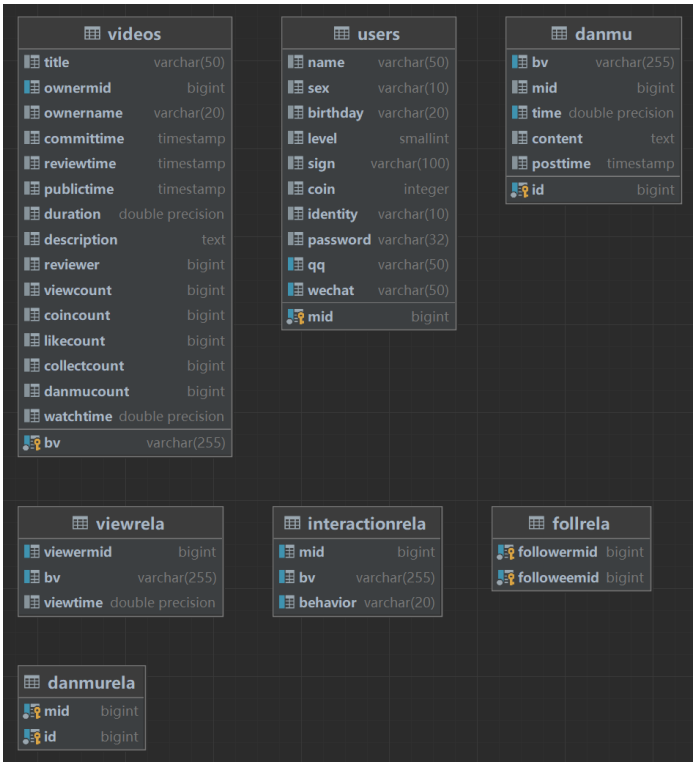
说明：以学生和成绩的关系为例，学生表中的 `student_id` 是主键，那么成绩表中的 `student_id` 则为外键。如果更新学生表中的 `student_id`，同时触发成绩表中的 `student_id` 更新，即为级联更新。外键与级联更新适用于单机低并发，不适合分布式、高并发集群；级联更新是强阻塞，存在数据库更新风暴的风险；外键影响数据库的插入速度。

当然，由于对老师上课时提到的这一段话印象过于深刻，我没有在本项目中使用任何外键约束。

E-R 图：



database diagram：



二、各类 API 的实现

由于我在 project 的逐步完善过程中对项目架构进行了多次调整、优化，这里不妨让我跟随我的心路历程对整个 project 进行汇报。

（一）数据导入

```java
/**
 * It's important to mark your implementation class with {@link Service} annotation.
 * As long as the class is annotated and implements the corresponding interface, you can place it under any package.
 */
no usages
@Service
@Slf4j
public class DatabaseServiceImpl implements DatabaseService {

    /**
     * Getting a {@link DataSource} instance from the framework, whose connections are managed by HikariCP.
     * <p>
     * Marking a field with {@link Autowired} annotation enables our framework to automatically
     * provide you a well-configured instance of {@link DataSource}.
     * Learn more: <a href="https://www.baeldung.com/spring-dependency-injection">Dependency Injection</a>
     */
    5 usages
    @Autowired
    public DataSource dataSource;

    12 usages
    private static final int THREAD_POOL_SIZE = 15;
```

作为整个项目最为重要也是唯一给了部分备注和样例实现的部分，我首先着眼于DatabaseServiceImpl。在理解了如上图中各类高亮注释的含义后，我试着进行数据导入。这就不得直面一个问题——

我该如何设计出一个“最佳”的数据库结构?

或者更明白些，如何把 user、danmu、video 之间的复杂关系存下来?
一开始，受 dto 里 record 类的结构和 project 文档的影响，我试着将诸如 follower、likedby 等关系直接作为对应主体的一条属性(通过拼接字符串)存下来。例如下图划线部分:

```
CREATE TABLE IF NOT EXISTS Danmu (
    id BIGSERIAL PRIMARY KEY,
    bv VARCHAR(255),
    mid BIGINT,
    "time" FLOAT,
    content TEXT,
    postTime TIMESTAMP,
    likedby TEXT
);
```

为了节约空间，我打算将其以 base64 编码后储存。为此，我在工具类 Utils 里写了如下方法:

```
no usages
public static String encodeToBase64(long[] longArray) {
    ByteBuffer byteBuffer = ByteBuffer.allocate( capacity: longArray.length * Long.BYTES);
    LongBuffer longBuffer = byteBuffer.asLongBuffer();
    longBuffer.put(longArray);

    return Base64.getEncoder().encodeToString(byteBuffer.array());
}

no usages
public static long[] decodeFromBase64(String base64String) {
    byte[] bytes = Base64.getDecoder().decode(base64String);
    ByteBuffer byteBuffer = ByteBuffer.wrap(bytes);
    LongBuffer longBuffer = byteBuffer.asLongBuffer();
    long[] longArray = new long[bytes.length / Long.BYTES];
    longBuffer.get(longArray);
    return longArray;
}

no usages
public static String modifyBase64String(String base64String, long m) {
    // 解码 Base64 字符串
    byte[] bytes = Base64.getDecoder().decode(base64String);
    ByteBuffer byteBuffer = ByteBuffer.wrap(bytes);
    LongBuffer longBuffer = byteBuffer.asLongBuffer();
    // 将字节转换为 long 数组
    ArrayList<Long> longList = new ArrayList<>();
    while (longBuffer.hasRemaining()) {
        longList.add(longBuffer.get());
    }
    // 检查 m 是否存在，并相应地修改数组
    if (longList.contains(m)) {
        longList.remove(m);
    } else {
        longList.add(m);
    }
    // 创建新的 ByteBuffer 并重新编码为 Base64
    ByteBuffer newByteBuffer = ByteBuffer.allocate( capacity: longList.size() * Long.BYTES);
    LongBuffer newLongBuffer = newByteBuffer.asLongBuffer();
    for (Long l : longList) {
        newLongBuffer.put(l);
    }
    return Base64.getEncoder().encodeToString(newByteBuffer.array());
}
```

encodeToBase()和 decodeFromBase64() 能够对数组编码和解码 base64，同时为了应对后期要求，modifyBase64String()能够快速检测 base64 字符串转成的数组中是否有指定值，并按需“删除”和“增添”。

但是很快，随着对整个项目要求和方法的深入理解，我否决了这一想法——使用拼接字符串将关系存为属性的决定确实节约空间、结构清晰，但是面对频繁的增删改查实在过于臃肿，再加上每次都需要涉及 base64 转码，整个过程是不可想的！

于是，我便走上了另一个极端——将所有常用的主体间关系都存成一个个特定表格、每个表格各司其职，储存较少的信息。如下图：

```sql
-- 创建 DanmuRela 表

CREATE TABLE IF NOT EXISTS DanmuRela
(
    mid BIGINT,
    id  BIGINT,
    PRIMARY KEY (mid, id)
);

-- 创建 FollRela 表
CREATE TABLE IF NOT EXISTS FollRela
(
    followerMid BIGINT,
    followeeMid BIGINT,
    PRIMARY KEY (followerMid, followeeMid)
);

-- 创建 ViewRela 表
CREATE TABLE IF NOT EXISTS ViewRela
(
    viewerMid BIGINT,
    bv        VARCHAR(255),
    viewTime  FLOAT
);

-- 创建 InteractionRela 表
CREATE TABLE IF NOT EXISTS InteractionRela
(
    mid      BIGINT,
    bv       VARCHAR(255),
    behavior VARCHAR(20) --CHECK (behavior IN ('like', 'coin', 'collect'))
);
```

同时，为了再次加速检索速度，几乎所有需要用到 WHERE 筛选的属性列都被设置了 INDEX 索引。

既然提到了 index，就绕不开一个想法——到底是先导数据、再建 index 快；还是先建 index、再导数据快？这里我进行了多次实验，截取了其中两次，分别对应前后两者的代表。

```
Step 1: Import data
BenchmarkResult(id=1, passCnt=null, elapsedTime=86921)
```

```
Step 1: Import data
BenchmarkResult(id=1, passCnt=null, elapsedTime=109006)
```

不难看出，在导入小型数据时，最后再建 index 可以有效减低时间成本。猜测原因如下：虽然按照“建立 B-tree”的 index 结构来看，二者都是 O(nlogn)的复杂度，但是先建立 index 意味着之后的每次查询都需要在原表和 index 中进行 O(logn)的查找，复杂度可达到上界。而对于已有数据集建立 index 虽然对数据库逻辑和性能有一定要求，但确实可以优化处理速度。

然而，无论是哪一种方法，现在的导入速度都太慢了，于是我打算用多线程（都用 java 了还不狠狠调线程池），代码如下。

```java
public void importUsersExecutorService(List<UserRecord> userRecords) throws InterruptedException {
    int chunkSize = userRecords.size() / THREAD_POOL_SIZE;

    ExecutorService executor = Executors.newFixedThreadPool(THREAD_POOL_SIZE);

    for (int i = 0; i < THREAD_POOL_SIZE; i++) {
        int start = i * chunkSize;
        int end = (i == THREAD_POOL_SIZE - 1) ? userRecords.size() : (start + chunkSize);

        Runnable task = () -> {
            try (Connection connection = dataSource.getConnection()) {
                importUsers(connection, userRecords.subList(start, end));
            } catch (SQLException e) {
                e.printStackTrace();
            }
        };
        executor.submit(task);
    }

    executor.shutdown();
```

即在正常 import 之外套个线程框架，并将数据按线程数分好（我这里用的 15 线程）。
通过 Runnable task 调用真正的导入函数。

```java
private void importVideos(Connection connection, List<VideoRecord> videoRecords) throws SQLException {
    String insertVideos = "INSERT INTO Videos (bv,title,ownerMid,ownerName,commitTime,reviewTime,publicTime,duration,de
    String insertViewRela = "INSERT INTO ViewRela (viewerMid,bv,viewTime) VALUES (?,?,?);";
    String insertInteractionRela = "INSERT INTO InteractionRela (mid,bv,behavior) VALUES (?,?,?);";
    long count=0;
    try(PreparedStatement ps = connection.prepareStatement(insertVideos);
        PreparedStatement ps2 = connection.prepareStatement(insertViewRela);
        PreparedStatement ps3 = connection.prepareStatement(insertInteractionRela)) {...}
}
```

运行后效果显著——原来导入数据 IDEA 只能吃 0.8%左右 CPU，现在能够达到 8—10%的占用率。

```
Step 1: Import data
BenchmarkResult(id=1, passCnt=null, elapsedTime=19492)
```

运行时间如上图，导入数据仅用 20s 左右，提速明显。

有趣的是，当我试图将线程数设置为 30 时，耗时没有明显减少；而将线程数设置成 10 时，耗时进一步缩小，稳定在 15s。

```
Step 1: Import data
BenchmarkResult(id=1, passCnt=null, elapsedTime=15146)
```

猜测原因：

1. 多线程在 java 端，本地 Postgres 数据库依旧有交互上限瓶颈。
2. 建立多线程以及对每个线程新建 Connection 有一定时间成本——这从耗时与线程数非线性递减即可看出。
3. 在多线程执行插入命令时，由于 postgres 需要同步更新 index，可能导致一定排队阻塞等。

总结：在我自己的电脑上，在“平衡”能耗下，经多次实验后线程数设置为 8-10 对 import data 的提速最为明显。

Ps：addBatch()、对批容量的设置等基础优化操作就不过多赘述了。

（二）插入新数据时主键的自动生成和获取。

提到主键，首先想到的就是这句“关系数据库理论要求每一个表都要有一个主键”，于是开始我将希望寄予于 Postgres”强大”的主键生成能力，将 User 表里 mid 设置为 BIGSERIAL。然而，现实很骨感，一句“出现重复键违反唯一约束”很快将我将我打回原形。检测后发现——BIGSERIAL 不具有检测初值的能力，即如果手动在导入数据时插入了离散的 mid 值，那么后续 BIGSERIAL 在自增长的过程中就会与已有值碰撞产生报错。

综上，我最后选择使用 trigger+函数为我生成主键，下面我将以 BV 号的生成举例。

```
DO
$$
    BEGIN
        IF NOT EXISTS (SELECT 1 FROM pg_trigger WHERE tgname = 'trigger_set_bv') THEN
            CREATE TRIGGER trigger_set_bv
                BEFORE INSERT
                ON Videos
                FOR EACH ROW
            EXECUTE FUNCTION set_bv();
        END IF;
    END
$$;
```

首先，创建 BV 触发器，对 BV 插入指令调用 set_bv( )进行编辑。

在 set_bv( )中对 bv 做非空检查，
并对空值执行 generate_unique_bv( )

```
-- 创建设置 BV 的触发器函数
CREATE OR REPLACE FUNCTION set_bv()
    RETURNS TRIGGER AS
$$
BEGIN
    IF NEW.bv IS NULL THEN
        NEW.bv := generate_unique_bv();
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

```sql
-- 创建自动生成 BV 的函数
CREATE OR REPLACE FUNCTION generate_unique_bv()
    RETURNS VARCHAR AS
$$
DECLARE
    new_bv VARCHAR(13);
    i      INT;
    ch     CHAR;
BEGIN
    LOOP
        new_bv := 'BV';
        FOR i IN 1..10
            LOOP
                ch := SUBSTRING(MD5(RANDOM()::TEXT) FROM i FOR 1);
                IF RANDOM() < 0.5 THEN
                    new_bv := new_bv || UPPER(ch);
                ELSE
                    new_bv := new_bv || ch;
                END IF;
            END LOOP;
        IF NOT EXISTS (SELECT 1 FROM Videos WHERE bv = new_bv) THEN
            RETURN new_bv;
        END IF;
    END LOOP;
END;
$$ LANGUAGE plpgsql;
```

上图是生成 BV 的主体函数。首先，使用 MD5 和 RANDOM 函数生成随机 10 个大小写敏感字符；接着，在前方拼接上"BV"开头；最后，对该 BV 号进行查重，确认唯一性后返回。

之后，便是对生成的各种主键的获取了，这里以在 import data 阶段唯一不在 record 里提供主键 id 的 DanmuRecord 举例。

```sql
CREATE TABLE IF NOT EXISTS Danmu
(
    id       BIGSERIAL PRIMARY KEY,
    bv       VARCHAR(255),
    mid      BIGINT,
    "time"   FLOAT,
    content  TEXT,
    postTime TIMESTAMP
);
```

对于 BIGSERIAL PRIMARY KEY，如果插入时无该列，则会顺序自动生成，并通过 RETURN_GENERATED_KEYS 返回。

```java
try (PreparedStatement ps = connection.prepareStatement(insertDanmu, Statement.RETURN_GENERATED_KEYS);
```

经试验，即使在有多线程、addBatch 的情况下，执行该指令后返回的 ResultSet getGeneratedKeys()仍是顺序且不会引起主键冲突的多行 id，这里 postgres 的功劳很大。

至此，关于数据库数据的导入、插入等已解析完成。

(三) “有趣”的 API

在本项目中，要求我们实现了很多的方法，这里我将挑选出最好玩的几个进行分析。

1. isValidAuth ()方法：

作为一个视频网站，检测用户身份的合法性自然是重中之重，不仅需要复杂的匹配检查逻辑，还需与数据库的交互，因而我做了大量优化。
考虑到建立和执行 PreparedStatement 所需的大量开销，我限制整个方法与数据库的交互次数不超过一次，并且将所有可能多次使用的变量比较存为布尔值以供调用。

```
15 usages
public static UserRecord isValidAuth(Connection connection, AuthInfo auth) {
    if(auth.getPassword()!=null){
        if(auth.getMid()==0)return null;
        try (PreparedStatement ps = connection.prepareStatement( sql: "SELECT password,mid,name,coin,identity FROM Users WHERE
            ps.setLong( parameterIndex: 1, auth.getMid());
            try(ResultSet rs = ps.executeQuery();) {
                if (!rs.next()) return null;
                if (!rs.getString( columnLabel: "password").equals(auth.getPassword())) return null;
```

（共 72 行）
最终，考虑到 auth 的参数本身可能是残缺的，我创新性地将 isValidAuth()方法的返回值改为了 UserRecord 类，如果 auth 不合法就返回 null，这样就可通过一次查询同时得到合法性和用户的所有信息，大大节约了后续所有方法的时间开销。

2. likeVideo()，coinVideo()，collectVideo()方法。（点赞、投币、收藏）：

作为某站的招牌功能，三连并没有看上去那么好完成。原因有三：

① 三连需要满足很多前置条件（视频是否存在？是否可见？是否是视频拥有者？）
② 如果之前做过该操作，需要特殊处理（回退操作 or 不做操作）
③ 在复杂的各类储存关系的表中增删改查

我们一个个慢慢道来。

```
public boolean likeVideo(AuthInfo auth, String bv) {
```

首先，检测用户合法性以及一堆前置条件势必会用到多次查询，但可以利用上个方法中 isValidAuth ()的优化思路，让一次查询返回尽可能多有用信息并做处理，落到实处就是下方的过程：

```
public boolean likeVideo(AuthInfo auth, String bv) {
    try(Connection connection= dataSource.getConnection();){
        UserRecord userRecord=isValidAuth(connection,auth);
        if(userRecord==null)return false;
        if(canBvReach(connection,bv,userRecord)){
            if(isDone(connection,bv, userRecord.getMid(), behavior: "like")){
```

```java
public static boolean canBvReach(Connection connection,String bv,UserRecord userRecord){
    if (bv==null||bv.isEmpty())return false;
    try(PreparedStatement ps = connection.prepareStatement(
            sql: "SELECT ownerMid,reviewTime,publicTime FROM Videos WHERE bv = ?")){
        ps.setString( parameterIndex: 1,bv);
        try(ResultSet rs1 = ps.executeQuery()) {
            if (rs1.next()) {
                if(userRecord.getMid() == rs1.getLong( columnLabel: "ownerMid"))return false;
                if (String.valueOf(userRecord.getIdentity()).equals("SUPERUSER")) return true;
                if (rs1.getTimestamp( columnLabel: "reviewTime") == null) return false;
                if(rs1.getTimestamp( columnLabel: "publicTime")==null)return true;
                if (rs1.getTimestamp( columnLabel: "publicTime") != null && rs1.getTimestamp( columnLabel: "publicTime").
                    return false;
            } else return false;
            return true;
        }
    } catch (SQLException e) {
        e.printStackTrace();
    }
    return false;
}
```

在 canBvReach( )中，通过一次查询即对①中的所有前置条件进行了筛查和判断，全部满足才返回 true；加上 isDone( )检测是否需特殊处理的一次查询，共计两次查询就理清了所有情况。

但接下来我又犯了难——插入\删除三连记录、更新视频信息怎么都需要两次执行，似乎没有优化空间了……于是，我做了一个“违背祖宗”的决定：

```java
redStatement ps = connection.prepareStatement(
l: "INSERT INTO InteractionRela (mid,bv,behavior) VALUES (?,?,?);UPDATE Videos SET collectCount=collectCount+1 where bv = ?;")
```

将多条 SQL 操作写进了一个 PreparedStatement 里！既然 Postgres 支持这样的操作，不用白不用。（当然，类似的复合语句操作在我的程序优化中几乎随处可见，之后就不过多赘述了：
其中的代表就是下图中删除用户时做的超级大更新……）

```java
        return false;
    }
    try (Connection connection = dataSource.getConnection();
         PreparedStatement ps = connection.prepareStatement( sql: "SELECT identity,mid FROM Users WHERE mid = ? ;");
         PreparedStatement ps2 = connection.prepareStatement( sql: "DELETE FROM Users WHERE mid = ? ;\n" +
                 "DELETE FROM FollRela WHERE followerMid = ? ;\n" +
                 "DELETE FROM FollRela WHERE followeeMid = ? ;\n" +
                 "WITH DeletedRecords AS (\n" +
                 "    DELETE FROM ViewRela\n" +
                 "        WHERE viewerMid = ?\n" +
                 "        RETURNING bv\n" +
                 ")\n" +
                 "UPDATE Videos\n" +
                 "SET viewCount = viewCount - 1\n" +
                 "WHERE bv IN (SELECT bv FROM DeletedRecords);\n" +
                 "\n" +
                 "WITH DeletedLikeRecords AS (\n" +
                 "    DELETE FROM InteractionRela\n" +
                 "        WHERE mid = ? AND behavior = ?\n" +
                 "        RETURNING bv\n" +
                 ")\n" +
                 "UPDATE Videos\n" +
                 "SET likeCount = likeCount - 1\n" +
                 "WHERE bv IN (SELECT bv FROM DeletedLikeRecords);\n" +
                 "\n" +
                 "WITH DeletedCoinRecords AS (\n" +
                 "    DELETE FROM InteractionRela\n" +
                 "        WHERE mid = ? AND behavior = ?\n" +
                 "        RETURNING bv\n" +
                 ")\n" +
                 "UPDATE Videos\n" +
                 "SET coinCount = coinCount - 1\n" +
                 "WHERE bv IN (SELECT bv FROM DeletedCoinRecords);\n" +
                 "\n" +
                 "WITH DeletedCollectRecords AS (\n" +
                 "    DELETE FROM InteractionRela\n" +
                 "        WHERE mid = ? AND behavior = ?\n" +
                 "        RETURNING bv\n" +
                 ")\n" +
                 "UPDATE Videos\n" +
                 "SET collectCount = collectCount - 1\n" +
                 "WHERE bv IN (SELECT bv FROM DeletedCollectRecords);")
    ) {UserRecord rs2=isValidAuth(connection,auth);
        if(rs2==null)return false;
        ps.setLong( parameterIndex: 1, mid);
```

5072	1.00	0.03
6365	1.00	0.03
5783	1.00	0.03

综上经过一轮优化后，三连方法的 benchmark 表现分从上方的一开始 0.03 分跃升至下方的全部过半，耗时也大幅缩短，可喜可贺可喜可贺。（第一次如此之慢其实还有忘设 index 的原因）

229	1.00	0.65
333	1.00	0.57
291	1.00	0.54

3. generalRecommendations( ) “推荐最火视频”

既然提到了 videos，那就不得不细说下这个方法，作为一个逆天的推荐算法，它几乎以一己之力改变了多少人的数据库架构。

方法要求算出每个视频的三连比例、弹幕比例、观看时长比例并排序，看似朴实极了，实则确实朴实，但如果真用 join 或者查询来计算则会时间复杂度爆炸。经测试，面对较大数据时，单个测试样例就需要上百毫秒，这显然不符合期望。

单纯的建立 index 或者优化查询次数面对庞大数据杯水车薪，有没有更加直接的方法？

只有一个——改变表的属性结构。如下图，我在 Videos 表中加入 viewCount、coinCount、likeCount、collectCount、danmuCount、watchTime 列，专门储存推荐算法中用到的值。同时，DatabaseServiceImpl 的 importData 方法也改为了最后导入 VideoRecord，力求用最小的代价将 Videos 的各列统计完全。

```
CREATE TABLE IF NOT EXISTS Videos
(
    bv           VARCHAR(255) PRIMARY KEY,
    title        VARCHAR(50),
    ownerMid     BIGINT,
    ownerName    VARCHAR(20),
    commitTime   TIMESTAMP,
    reviewTime   TIMESTAMP,
    publicTime   TIMESTAMP,
    duration     FLOAT,
    description  TEXT,
    reviewer     BIGINT,
    viewCount    BIGINT,
    coinCount    BIGINT,
    likeCount    BIGINT,
    collectCount BIGINT,
    danmuCount   BIGINT,
    watchTime    DOUBLE PRECISION
);
```

```
PreparedStatement pstmt = conn.prepareStatement( sql: "WITH VideoStats AS (\n" +
        "    SELECT\n" +
        "        v.bv,\n" +
        "        (LEAST(v.coinCount::DOUBLE PRECISION/v.viewCount,1)+LEAST(v.likeCount::DOUBLE PRECISION/v.viewCount
        "        v.danmuCount::DOUBLE PRECISION/v.viewCount AS danmu_avg,\n" +
        "        v.watchTime/v.viewCount/v.duration AS finish_avg\n" +
        "    FROM Videos v\n" +
        "    group by v.bv)\n" +
        "SELECT bv,interaction_rate,danmu_avg,finish_avg\n" +
        "FROM VideoStats\n" +
        "ORDER BY (interaction_rate + danmu_avg + finish_avg) DESC\n" +
        "LIMIT ? OFFSET ?;")) {
```

需注意各比率不能超过 1（有些操作不需要观看视频）以及计算时需先转为 double。
这下，推荐算法就成了一个单纯的求和排序了。

Test Step	# Case	# Passed	Std. Time (ms)	Elapsed Time (ms)	Correctness	Performance
6	11	11	925	3	1.00	1.00

当然，这么做也有副作用，即每次对互动记录做增删改都要更新对应视频的各项属性，不过考虑到可以应用上文中的“复合语句”优化，以及使视频记录更加丰富充实，这么做瑕不掩瑜。

4. searchVideo( )　按关键词查询、排序视频

这个方法恐怕是所有方法中最“难”的了，涉及大批量的字符串匹配和一些分割方面的 corner case。最初我使用的是 DSAA 课程里教的字符串匹配算法，但由于需要主动维护一个循环指针，效率不太好。之后，我发现 Postgres 手册里有一个名为 regexp_count（）的函数，使用正则表达式匹配子串数量，完整函数如下：

```
CREATE OR REPLACE FUNCTION count_non_overlapping_substrings(main_str text, sub_str text) RETURNS integer AS $$
DECLARE
    count integer := 0;
    sub_str_array text[];
    current_sub_str text;
BEGIN
    IF sub_str = '' OR main_str = '' THEN
        RETURN 0;
    END IF;
    sub_str_array := string_to_array(sub_str, ' ');
    FOREACH current_sub_str IN ARRAY sub_str_array
        LOOP
            if current_sub_str =''THEN
                CONTINUE;
            END IF;
            count := count + regexp_count(main_str, '(?i)' || regexp_replace(current_sub_str, '([\[\].^$*+?|{}()])', '\\\1', 'g'));
        END LOOP;
    RETURN count;
END;
$$ LANGUAGE plpgsql;
```

首先将查询字符串 sub_str 以“ ”分隔开，对每个分隔后的 current_sub_str 进行非空检查以避免多个空格作为一个分隔符的情况。之后，对 sub_str 中可能影响正则匹配的特殊字符进行转义，最后加上(?!)的前缀表示大小写不敏感，累加 regexp_count 函数的结果，即可得到所有 sub_str 在 main_str 中的出现次数了。
下图为 searchVideo( )的 SQL，其中调用了上述的计数方法 count_non_overlapping_substrings

```
if(String.valueOf(rs1.getIdentity()).equals("SUPERUSER")) {
    try(PreparedStatement ps = connection.prepareStatement( sql: "SELECT bv, relevance\n" +
            "FROM (\n" +
            "         SELECT bv,\n" +
            "                (count_non_overlapping_substrings(title, ?) +\n" +
            "                 count_non_overlapping_substrings(ownerName, ?) +\n" +
            "                 count_non_overlapping_substrings(description, ?)) AS relevance,viewCount\n" +
            "         FROM Videos\n" +
            "     ) AS sub\n" +
            "WHERE sub.relevance > 0\n" +
            "ORDER BY relevance DESC, viewCount DESC\n" +
            "LIMIT ? OFFSET ?;")) {
```

特别地，对于一般用户，需筛选公开视频。

```
 WHERE  ownerMid = ? OR(reviewTime is not null AND publicTime<now())\n" +
```

5.其余 API

剩余未在本报告中提到的 API 要么大多为体力劳动、要么其亮点已在上文方法中提过、要么是我没找出好的优化思路而羞于展示（例如 displayDanmu、sendDanmu、getUserInfo 等）。
我希望让读者知晓，虽然我试图尽己所能提速程序，但始终保留着严谨性，没有所谓“在测试样例中找优化”，也没有因为 benchmark 的规范传参而放弃方法开头的参数检查。

三、一些额外操作与思考

1.敏感信息加密和解密。

加密作为信息储存最主要的一把锁，我采取的是 XOR+base64 编码的加密方式，简单有效。
具体实现如下：

```
CREATE TABLE IF NOT EXISTS PasswordKey (
    id BIGSERIAL,
    key TEXT
);
INSERT INTO PasswordKey (key) VALUES (substr(md5(random()::text), 1, 20));
```

首先，在数据库层面随机生成 20 长度大小写不敏感字符串，并存进 PasswordKey 表中。
之后，在 java 中读取该密钥并存进 String Key 中。这样，后续导入各类数据就可以先调用下方的方法加密了。

```java
private static String encryptDecryptXOR(String input, String key) {
    StringBuilder output = new StringBuilder();
    for (int i = 0; i < input.length(); i++) {
        output.append((char) (input.charAt(i) ^ key.charAt(i % key.length())));
    }
    return output.toString();
}


// 加密函数，使用 XOR 加密后转为 Base64
2 usages
public static String encrypt(String input, String key) {
    String encrypted = encryptDecryptXOR(input, key);
    return Base64.getEncoder().encodeToString(encrypted.getBytes());
}

// 解密函数，先将 Base64 解码，然后使用 XOR 解密
2 usages
public static String decrypt(String input, String key) throws Exception {
    byte[] decodedBytes = Base64.getDecoder().decode(input);
    return encryptDecryptXOR(new String(decodedBytes), key);
}
```

如图，使用抑或进行加密，之后进行 base64 编码。
解密也同理，由于该方法仅在导入数据和检测 auth 合法性中用到，该过程并不会过多拖慢时间复杂度。

2.额外 API 的实现

在完成整个项目的过程中，我意识到很多方法都需判断“该用户是否看过某视频”，即在 ViewRela 中查找。然而，除了导入数据时 VideoRecord 里自带的 long[ ]记录了初始观看用户外，整个项目并无方法能添加观看记录了。于是，我便设计完成了下方的 watchVideos 方法。

```java
public boolean watchVideos(AuthInfo auth, String bv,float watchTime) {
    try(Connection connection= dataSource.getConnection()){
        UserRecord userRecord=isValidAuth(connection,auth);
        if(userRecord==null)return false;
        if(canBvReach(connection,bv,userRecord)){
            if(!isView(connection,bv,userRecord)){
                try(PreparedStatement ps=connection.prepareStatement( sql: "INSERT INTO ViewRela (viewTime,bv,viewerMid) VALUES (?,?,?);UPDATE Videos SET watchTime = watchTime + ? WHERE
                    ps.setFloat( parameterIndex: 1,watchTime);
                    ps.setString( parameterIndex: 2,bv);
                    ps.setLong( parameterIndex: 3,userRecord.getMid());
                    ps.setDouble( parameterIndex: 4,(double)watchTime);
                    ps.setString( parameterIndex: 5,bv);
                    ps.execute();
                    return true;
                }
            }else {
```

该方法也会更新 Videos 表的 watchTime 列。

3. 角色建立与权限管理

```
CREATE ROLE ordinary_user LOGIN PASSWORD '123456';
GRANT SELECT, INSERT, DELETE ON FollRela TO ordinary_user;
GRANT SELECT, INSERT, DELETE ON InteractionRela TO ordinary_user;
GRANT SELECT, INSERT, DELETE ON DanmuRela TO ordinary_user;
GRANT SELECT, INSERT, UPDATE, DELETE ON ViewRela TO ordinary_user;
GRANT SELECT, INSERT, DELETE ON Danmu TO ordinary_user;
GRANT SELECT, INSERT, UPDATE, DELETE ON Videos TO ordinary_user;
GRANT SELECT, INSERT, UPDATE, DELETE ON Users TO ordinary_user;
CREATE ROLE administrator LOGIN PASSWORD '123456';
GRANT SELECT, INSERT, UPDATE, DELETE ON FollRela TO administrator;
GRANT SELECT, INSERT, UPDATE, DELETE ON InteractionRela TO administrator;
GRANT SELECT, INSERT, UPDATE, DELETE ON DanmuRela TO administrator;
GRANT SELECT, INSERT, UPDATE, DELETE ON ViewRela TO administrator;
GRANT SELECT, INSERT, UPDATE, DELETE ON Danmu TO administrator;
GRANT SELECT, INSERT, UPDATE, DELETE ON Videos TO administrator;
GRANT SELECT, INSERT, UPDATE, DELETE ON Users TO administrator;
```

如上图，建立普通用户和管理员两类角色，分别赋予一定方法。

```
CREATE ROLE the_boss2 LOGIN PASSWORD '123456';
GRANT CONNECT ON DATABASE sustc TO the_boss2;
ALTER DEFAULT PRIVILEGES FOR ROLE the_boss2 IN SCHEMA public
    GRANT ALL PRIVILEGES ON TABLES TO the_boss2;
GRANT ALL PRIVILEGES ON ALL TABLES IN SCHEMA public TO the_boss2;
ALTER schema public OWNER TO the_boss2;
GRANT TRUNCATE ON ALL TABLES IN SCHEMA public TO the_boss2;
GRANT INSERT ON ALL TABLES IN SCHEMA public TO the_boss2;
```

如上图，建立了 the_boss2 角色，并设置密码“123456”，并给予其 public 里当前及未来可能出现的所有表格的权限。

有趣的是，权限的赋予和调用十分严格，在后续的操作中我发现函数也只能由建立这个函数的角色调用；甚至如果表格的某列是 BIGSERIAL 也是如此。如下图，拥有较高权限的 the_boss2 甚至能直接删除 Danmu 表，却无法调用 Danmu 的“id”自增。

```
sustc.public> INSERT INTO Danmu (bv,mid,time,content,postTime) VALUES ('BV12mG4y1z761',204644,729.579,'聊爆了呀  随便查阿','2023-07-29 18:09:02.514+08'::timestamp)
[2024-01-04 17:05:00] [42501] 错误: 对于序列 danmu_id_seq, 权限不够
```

只能手动插入 id 才不至报错（Danmu 表由 superuser 创建）

```
sustc.public> INSERT INTO Danmu (id,bv,mid,time,content,postTime) VALUES (1234,'BV12mG4y1z761',204644,729.579,'聊爆了呀  随便查阿','2023-07-29 18:09:02.514+08'::timestamp)
[2024-01-04 17:04:30] 1 row affected in 13 ms
```

4.启用审计功能

为了复原恶意操作和不当操作，我试图启用数据库的审计功能

```
                    # bind-parameter values to N bytes;
                    # -1 means print in full, 0 disables
log_statement = 'mod'           # none, ddl, mod, all
#log_replication_commands = off
#log_temp_files = -1            # log temporary files equal or
```

启用后，导入数据速度明显变慢，如下图。

```
: Step 1: Import data
: BenchmarkResult(id=1, passCnt=null, elapsedTime=214479)
```

同时，postgresSQL 文件夹下多出了 1.65G 的日志文件。
当以'mod'模式启用审计时，将会记录所有 INSERT、UPDATE、DELETE、COPY FROM/TO 信息，这似乎有些浪费了，建议只记录必要信息或使用其他 postgres 审计软件。

四、总结

感谢你阅读到这里，正如 report 开头所说，本次的数据库 project 对我而言收获颇丰，体验到了设计、推进、完成、优化的全过程，就连埋头 debug 的几天现在回想起来也格外"有趣"，相信经历了这次作业，我对数据库本身机器作用都有了更深的理解。
最后，附上第一次跑线上测试和最后一次测试的对比。

Benchmark Result

This benchmark is based on the test data with ETAG 69d03da0c45e786250b1c3dd586e1ee9.

Test Step	# Case	# Passed	Std. Time (ms)	Elapsed Time (ms)	Correctness	Performance
1	N/A	N/A	65832	28888	N/A	1.00
2	20	20	376	17	1.00	1.00
3	61	61	34	4	1.00	1.00
4	56	56	9	6	1.00	1.00
5	43	43	1729	2265	1.00	0.76
6	11	11	925	3	1.00	1.00
7	27	27	3311	1303	1.00	1.00
8	142	142	29219	12069	1.00	1.00
9	382	382	14	46	1.00	0.30
10	1016	1016	40	118	1.00	0.34
11	500	500	134	266	1.00	0.50
12	500	500	N/A	N/A	1.00	N/A
13	511	511	148	229	1.00	0.65
14	686	686	189	333	1.00	0.57
15	644	644	156	291	1.00	0.54
16	184	184	42	37	1.00	1.00
17	273	273	46	57	1.00	0.81
18	184	184	25	29	1.00	0.86
19	35	35	11	14	1.00	0.79
20	6	6	29	4	1.00	1.00
21	55	55	362	21	1.00	1.00
22	199	199	114	164	1.00	0.70
23	87	87	18	25	1.00	0.72

Copy Benchmark Report

Elapsed Time (ms)	Correctness	Performance
116160	N/A	0.57
56	1.00	1.00
9	1.00	1.00
20	1.00	0.45
2792	0.67	0.62
5	1.00	1.00
56	1.00	1.00
4152	1.00	1.00
43	1.00	0.33
1657	1.00	0.02
267	1.00	0.50
N/A	1.00	N/A
6345	1.00	0.02
9012	1.00	0.02
7174	1.00	0.02
40	1.00	1.00
861	1.00	0.05
1130	0.97	0.02
72	1.00	0.15
25	0.83	1.00
311	1.00	1.00
895	1.00	0.13
730	1.00	0.02

Copy Benchmark Report