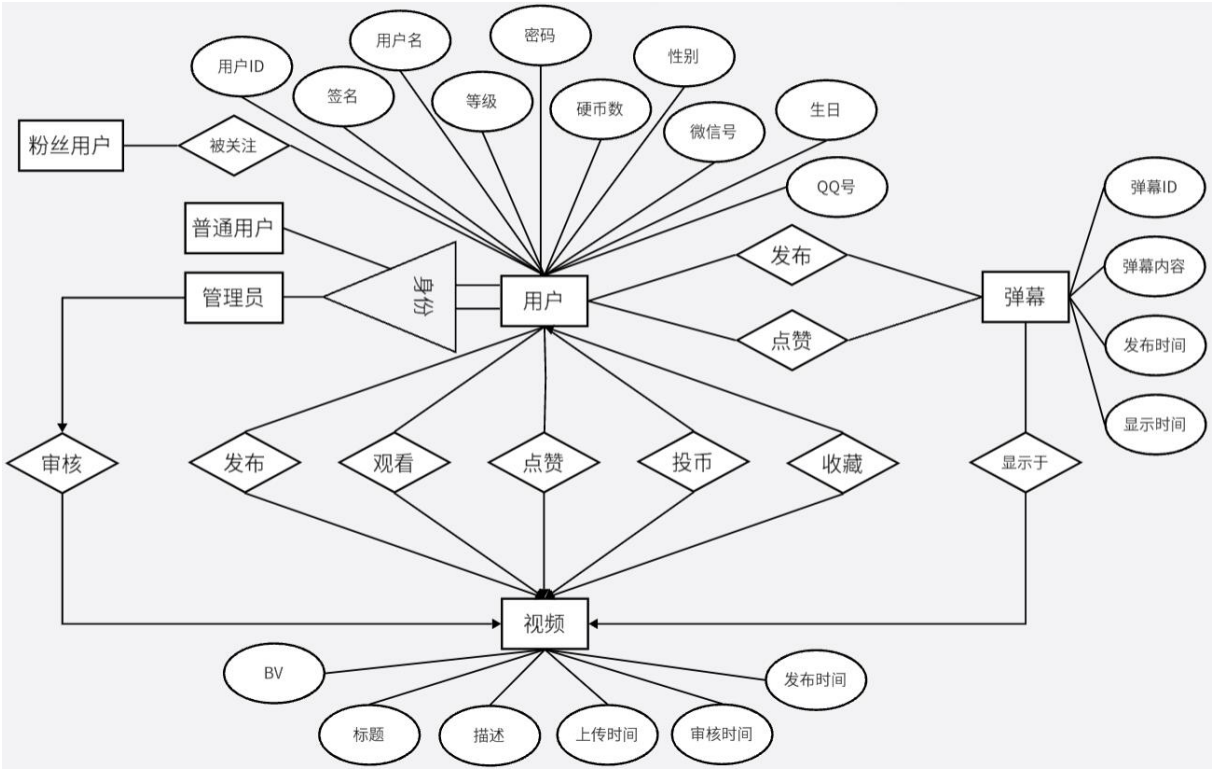
基于 PostgreSQL 的视频网站设计及性能分析

周圣力 12212232

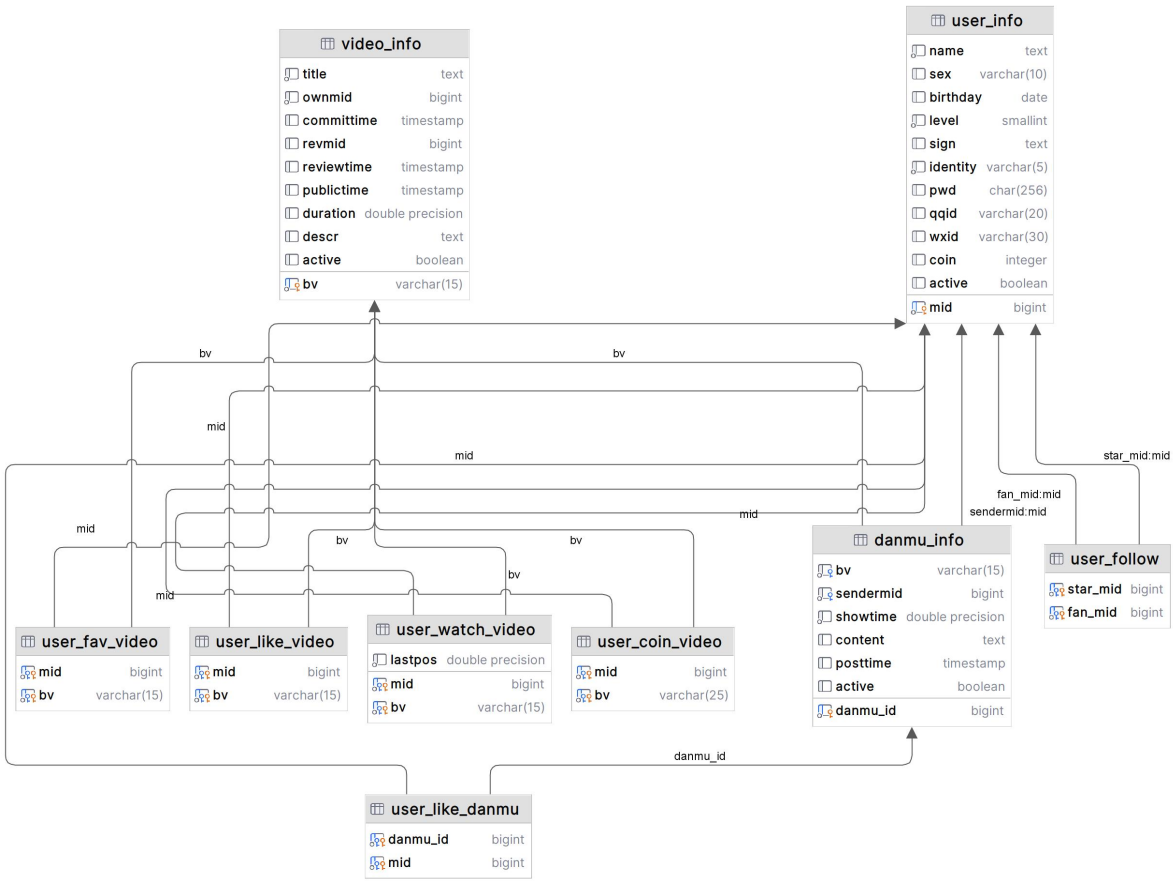
Jan. 4th, 2024

1 数据库设计方案

根据题目要求，我设计了一个由用户（user）、视频（video）和弹幕（danmu）为实体的数据库，其关系如下方 E-R 图所示：

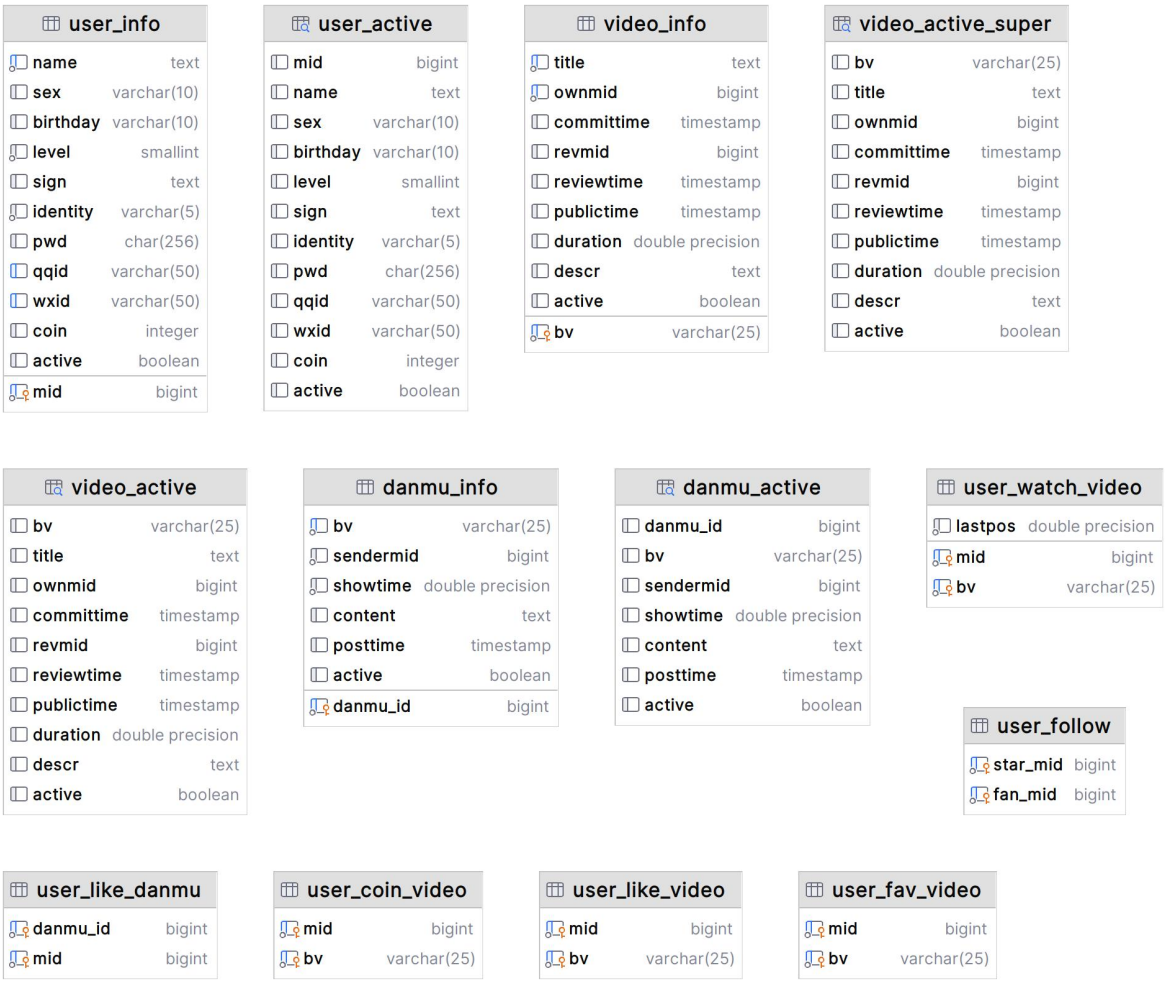


在设计数据表时，我一开始采用了如下方案：



这个方案的特点是具有很多的外键（所有能加的外键都加上了），但这带来了两个很直接的问题：一方面外键的加入会直接**拖慢所有修改操作的速度**（增删改之后都会触发外键的检查），另一方面外键的引入也会给数据库的**安全性**（如级联删除隐患）、**操作灵活性**和数据库的**可迁移性**造成一定的负面影响（个人认为这些问题也是《阿里巴巴 Java 开发手册》(1) 中规定所有外键功能需要在应用层解决的原因）。

在确认外键仅用于测试平台上程序运行结束后级联删除所有数据后，我去掉了所有的外键，把数据库的设计变成了这样（其中 user_active, video_active, video_active_super, danmu_active 为视图，便于查询操作中的权限管理）：



~~看起来清爽多了。~~ 但清爽不是目的，目的是让这张简洁的表让后续的方法跑得更快。一个比较直接的方法是给这些表加上主键和索引。在开始优化建表之前，我先实现了所有的 API，在 benchmark 测试平台上测试后得到的相对运行效率[1]为 0.4（多次实验结果平均值的近似值，此处放一个提交记录的链接以提供更多信息，下同）。此后，我根据不同字段在搜索中的使用次数和数据的区分度对部分字段加了索引：

- user_info 表：mid **（加主键）**，name **（加索引）**，birthday（搜索太少，不加），level（搜索太少，区分度小，不加），sign（搜索太少，不加），identity（区分度小，不加），pwd（搜索较少，不加），qqid **（加索引）**，wxid **（加索引）**，coin（搜索太少，不加），active（区分度小，不加）。
- video_info 表：bv **（加主键）**，title **（加索引）**，ownMid **（加索引）**，其它字段基本不用于搜索，不加索引。

[1]各项功能与标准程序的耗时之比的倒数（最大为 1）的平均值，约接近 1 说明效率越高。

- danmu_info 表：danmu_id**（加主键）**, bv**（加索引）**, senderMid**（加索引）**, showTime**（加索引）**，其它字段基本不用于搜索，不加索引。

在一顿操作之后，相对运行效率提高到了 0.7。

2 部分 API 实现方案及相关优化

2.1 Database Service**（数据导入）**

数据导入看似一个简单的操作，但要以较快的速度完成数据的插入，仍需要在操作方式上进行一定的优化。在所有优化之前，在本地导入所有数据的平均时间为 19 分 23 秒（这一部分的测试时间较早，平台还未上线，所以 2.1.1 到 2.1.4 都是在本地用较大的数据集测试的结果）。

2.1.1 **推迟主键和索引的创建**

在创建主键和索引之后，所有的增删改操作都会触发主键的重复性查询和索引的修正，产生额外的时间开销。考虑到**插入的数据是保证有效的**，我**将所有主键和索引的建立放到了数据插入之后**。在这一优化之后，插入数据的平均时间缩短到 13 分 25 秒。

```
create table table_name ( ... )
-- ...
insert into table_name values ( ... )
-- ...
alter table table_name add constraint ... -- add primary keys
-- ...
create index ... on table_name (...) -- create indexes
-- ...
create view ... as select ... from table_name -- create views
-- ...
create or replace function ... -- create functions
-- ...
```

2.1.2 **弹幕编号的获取**

在所有需要插入的数据中，有一张表的插入较为特殊——插入弹幕后需要手动生成弹幕的编号并返回给测试器。一种很“老实”的办法是插入弹幕，让数据库生成编号，最后再从数据库获取编号。这种方法看似合理，但意味着每插入一条弹幕就会多一次查

询操作，而且多条弹幕的信息不能同时插入，大大降低了效率。所以我将方案改为：直接在 Java 中给每一个弹幕提前预设好编号，然后直接插入数据库中。在这一优化后，插入的时间缩短到了 9 分 40 秒。

2.1.3 分批插入

在说这个优化之前，要先提及 JDBC 中的一个设置开关——auto commit。JDBC 的使用者大多数对于这个 API 的底层结构没有很深的了解，导致往往会出现事务（transaction）没有启动或在不正常的地方被启动的情况发生~~（然后开发者就又被投诉了）~~。为了解决这个问题，JDBC 的开发者就想了一个“土方法”——你不 commit，我来帮你！于是就有了 auto commit 这个开关：在 auto commit 开启后，JDBC 会自动帮忙管理事务，在自己的事务中单独执行每一个语句。

通过 Project 1 的测试，我发现在处理大量的 insert 语句时，DataGrip 会聪明地将 1000 条语句打包成一个整体发给数据库执行，这类似于下面代码中的 addBatch。但单纯的分批发送插入指令碰到 auto commit 就没辙了，原来打好的包过了 JDBC 又会被拆成一条一条命令单独执行，造成了大量的时间浪费。

所以解决方案到这里就很清晰了，直接使用 conn.setAutoCommit(false) 关掉这个设置即可。当然最后不能忘了要自己手动加上 conn.commit()。

在这一优化后，插入的时间缩短到 3 分 27 秒。

```
// load user_coin_video
String insertUserCoinVideoSQL = "insert into user_coin_video values (?, ?);";
try (Connection conn = dataSource.getConnection();
     PreparedStatement stmt = conn.prepareStatement(insertUserCoinVideoSQL)) {
    conn.setAutoCommit(false);
    for (VideoRecord videoRecord : videoRecords) {
        stmt.setString( parameterIndex: 2, videoRecord.getBv());
        for (Long mid : videoRecord.getCoin()) {
            stmt.setLong( parameterIndex: 1, mid);
            stmt.addBatch();
        }
    }
    stmt.executeBatch();
    conn.commit();
    conn.setAutoCommit(true);
} catch (SQLException e) {
    System.out.println("[ERROR] Fail to insert user_coin_video records, " + e.getMessage());
}
```

这里再说一句题外话，之前在 Project 1 中，由于 DataGrip 是直接链接 PostgreSQL 的，并不会经过 JDBC，所以不存在 auto commit 的问题。一开始我做这个测试的时候

以为 addBatch 就已经可以保证所有命令是被一起执行的了，直到关闭了 auto commit 才发现居然还能优化。

2.1.4 减少信息量

考虑到插入的所有信息都需要从 Java 程序传到数据库服务器上运行，所以提升传输的效率也是提高插入速度的一种方法。需要插入的数据量本身是无法减少的，所以我选择了减少插入语句的长度，从 insert into 表名 字段名 values 数值 变为 insert into 表名 values (...)，进一步还可以将多个 insert 语句合并，即 insert into 表名 values (...), (...), ...。不过由于要插入的数据内容在数量级上多于 SQL 命令的内容，所以这种优化的效果并不明显。

2.1.5 多线程插入

由于我设计的数据库不存在外键约束，所以插入的顺序是没有影响的。进一步可以将所有的插入任务分入多个线程，通过并行的方法加速数据的导入（以下为一个 4 线程数据导入的示例）。

```
ImportThread1 importThread1 = new ImportThread1(videoRecords);
ImportThread2 importThread2 = new ImportThread2(videoRecords);
ImportThread3 importThread3 = new ImportThread3(userRecords);
ImportThread4 importThread4 = new ImportThread4(userRecords, videoRecords, danmuRecords);
importThread1.start();
importThread2.start();
importThread3.start();
importThread4.start();
try {
    importThread1.join();
    importThread2.join();
    importThread3.join();
    importThread4.join();
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

在此基础上，还需通过确定线程数量及每个线程的工作量（数据插入量）进一步优化数据插入的效率。

由于导入的数据看起来像是来自于真实数据，所以可以一定程度反映需要导入的数据规模比例（至少数量级的比例是相近的）。我统计了每一个表的数据量：danmu_info 681 条，user_coin_video 45 万条，user_fav_video 43 万条，user_follow 595 万条，user_info

4 万条，user_like_danmu 1 万条，user_like_video 48 万条，user_watch_video 87 万条，video_info 76 条，并以此作为分割各个线程工作量的依据。

在切分线程的过程中，我主要考虑了两个因素：

- 尽量减少对于数据表的拆分，即“小表不拆，大表少拆”，从而减少多个线程同时访问一块内存区域的情况。
- 尽量平均各个线程的工作量。

根据以上数据，我设计了以下几个方案（为方便表述，以下使用数据插入量表示工作量，实际情况下按比例放大即可）：

- 单线程插入。
- 4 线程插入：

  1 号线程：user_follow 200 万条；

  2 号线程：user_follow 200 万条；

  3 号线程：user_follow 195 万条，danmu_info 681 条，user_info 4 万条，user_like_danmu 1 万条，video_info 76 条；

  4 号线程：user_coin_video 45 万条，user_fav_video 43 万条，user_like_video 48 万条，user_watch_video 87 万条。
- 8 线程插入：

  1 号线程：user_follow 120 万条；

  2 号线程：user_follow 120 万条；

  3 号线程：user_follow 120 万条；

  4 号线程：user_follow 120 万条；

  5 号线程：user_follow 115 万条；

  6 号线程：user_coin_video 45 万条，user_fav_video 43 万条；

  7 号线程：user_like_video 48 万条，danmu_info 681 条，user_info 4 万条，user_like_ danmu 1 万条，video_info 76 条；

  8 号线程：user_watch_video 87 万条。

- 12 线程插入：

  1 号线程：user_follow 70 万条；

  2 号线程：user_follow 70 万条；

  3 号线程：user_follow 70 万条；

  4 号线程：user_follow 70 万条；

  5 号线程：user_follow 70 万条；

  6 号线程：user_follow 70 万条；

  7 号线程：user_follow 70 万条；

  8 号线程：user_follow 70 万条；

  9 号线程：user_follow 35 万条；

  10 号线程：user_coin_video 45 万条，danmu_info 681 条，user_info 4 万条，user_like_ danmu 1 万条，video_info 76 条；

  11 号线程：user_fav_video 43 万条，user_like_video 48 万条；

  12 号线程：user_watch_video 87 万条。

测试结果如下：

线程数	耗时 1 (ms)	耗时 2 (ms)
1	59438	88863
4	59652	57697
8	27327	33708
12	29427	40304

其中耗时 1 对应使用在线评测机插入数据集的耗时，耗时 2 对应使用本机插入数据集的耗时。

多线程插入的大厦已然筑城，但上空仍飘着两朵“乌云”：第一朵“乌云”是 4 线程的数据导入速度并没有较单线程有明显提高；第二朵“乌云”则来源于 12 线程数据导入时速度的反常下降。通过查找资料，我摸索出了当线程数量逐渐增加时 CPU 行为发生的变化：

- 从单线程到 4 线程，虽然每个线程的任务量减小了，但由于**线程启动、数据分配和线程合并**需要时间，所以优化效果不明显。
- 从 4 线程到 8 线程，线程启动、数据分配和线程合并的时间相近，同时由于每个线程的**任务量降低**，总耗时有较为明显的减少。
- 从 8 线程到 12 线程，或者更多线程呢？从理论上分析，**最佳的线程数量等于** CPU **的逻辑处理器数量**。(2) 当线程数超过逻辑处理器数量时，CPU 会通过同时**多线程技术（**SMT**）和超线程技术（**HT**）**支持更多线程的同时运行，但此时切分线程所增加的时间开销已经超过了单个线程任务量下降节省下的时间，所以效果不会优于 12 线程的情况。进一步由于 12 线程时**通信成本**已经开始明显拖慢程序的总体运行速度了，所以在线程更多的情况下，程序的运行速度只会更慢。故此处不再测试更多线程的方案。

2.2 其它功能的 API

2.2.1 函数的迁移

在文档中，大多数 API 的实现要求可以被拆分为：“判断是否出现不合规输入的情况”和“若数据合规则执行相应操作”两部分，这意味着大多数的 API 都会涉及到超过 1 次（实际大约有 5 次左右）的操作，会在连接数据库的过程中产生大量的时间开销。受 Assignment 4 的启发，我**将所有实际执行参数的函数内容都挪到了数据库的函数中**，而 Java 中函数只需要做两件事——将相关的参数传给数据库的函数并从数据库的函数获取返回值，这样就可以将每个操作的数据库连接次数减到 1 次。

2.2.2 用户及其它实体的删除

『张三在平台上发布了一些 ** 言论，在被发现后“删号跑路”，因为相关的数据也被删除了，所以这件事情从此无法追责……』

作为一个应用于实际场景的数据库，一个基本的要求是里面的数据不能被用户随意地删除（这样在必要的时候才能将记录再次调出），所以一个合理的设计是：在删除用户或其它实体时仅作**软删除**，即仅将对应数据标记为无效数据。于是我将用户、视频、弹幕的表都加了一个 active 列，表示这个实体是否仍有效。

进一步，我又设置了视图 user_active（有效的用户）、video_active（普通用户可见的视频）、video_active_super（管理员可见的视频）和 danmu_active（有效的弹幕），

在简化查询语句的同时，也方便了其它 API 的接入。

这个设计到这里就可以比较好地保证程序运行的准确性和安全性了，但“天下没有免费的午餐”，由于每张实体表中都多记录了一个字段和一堆的无效数据，所以肯定对运行效率有一定的影响。针对这个问题，我考虑了如下建立索引的方法：

- 对所有的记录一视同仁，全部加上同样的索引。
- 使用**局部索引**（partial index），只对有效的记录加索引。
- 将无效的数据单独放到一张表中存起来，对存有效数据的那张表建索引。

首先不难看出，方案 3 一定是劣于方案 2 的。因为方案 2 在软删除一个元素之后，只会将这个元素“剔出”索引对应的 B+ 树；但方案 3 在这一操作之后，还要把被删除的数据插入到新的表中，一定会造成更大的时间开销。

针对局部索引，官方文档 (3) 中对于局部索引的相关表述是：通过**减少冗余的数据，缩减索引的大小**，从而提高索引的运行效率。但与此同时，使用局部索引也意味着**每当一个用户被软删除时，索引都会发生更新**，造成更大的时间开销。一棵有 n 个节点的 B+ 树的高度为 $\Theta(\log_2 n)$，所以在删除元素较少时，B+ 树的层数并不会发生变化，即用两种索引时查询操作的时间开销是相同的。此时又因为局部索引额外需要时间删除不满足条件的元素，所以耗时更长。但在这个项目所考虑的场景和数据中，由于**删除的记录条数较多**，所以使用局部索引可以**有效减少 B+ 树的层数**，在效率上更胜一筹，所以我在提交时也使用了局部索引加速相关操作。

2.2.3 查找视频

视频查找函数的核心是对相关度的计算。根据题目描述，可以通过 PostgreSQL 中的函数 regexp_split_to_table(keywords, E'\\s+') 按空格分割字符串 keywords，通过函数 regexp_count(str1, str2, 1, 'i')) 查找字符串 str1 中 str2 出现的次数。

在处理字符串时，除了要保证查询的正确性外，还需要保证用户不能通过字符串对系统的正常运行造成影响。在这个函数中，用户有可能从两个角度对系统进行攻击：

第一个角度是 SQL **注入攻击**。针对这种情况，我首先使用了 prepared statements 将参数从 Java 传给数据库。一旦用户传入的参数中包括了数据库中的“敏感词”，相关的字符串就会被做**转义处理**，保证不会出现 SQL 注入攻击。其次，在分割单词之后，即使函数受到 SQL 注入攻击，相关的语句也会被切成若干单词，无法保持整个攻击语句的作用，达到防止注入攻击的效果。

第二个角度是通过插入反斜杠等字符，**注入正则表达式**，从而**影响搜索的正确性**。针对这一问题，我在传入参数时**手动转义了正则表达式中的标识符**，从而避免了这种情况的发生。

```
create or replace function search_video(
    auth_mid bigint,
    auth_pwd varchar(260),
    auth_qqid varchar(50),
    auth_wxid varchar(50),
    keywords text,
    page_size int,
    page_num int
)
    returns varchar(25)[] as $$
    declare
        real_mid bigint;
    begin
        real_mid := (select verify_auth( _mid: auth_mid,  _pwd: auth_pwd,  _qqid: auth_qqid,  _wxid: auth_wxid));
        if real_mid < 0 then
            -- raise notice 'Authentication failed.';
            return null;
        end if;
        return (
            with word_set as (
                select regexp_split_to_table(keywords, E'\\s+') as word
            )


            select array_agg(ans) from (
                select tmp4.bv as ans from
                    (select user_watch_video.bv, count(*) as cnt
                        from user_watch_video group by user_watch_video.bv
                    ) as watch_cnt
                join
                    (select tmp1.bv, revMid, publicTime, ownMid, sum(
                            (select regexp_count(tmp1.title, word, 1, 'i')) +
                            (select regexp_count(coalesce(tmp1.descr, ''), word, 1, 'i')) +
                            (select regexp_count(tmp1.name, word, 1, 'i'))
                        ) as relevance
                        from (
                            (select video_active_super.bv, title, descr, ownMid, name, revMid, publicTime
                                from (video_active_super join user_active on ownMid = mid)) tmp2
                            cross join word_set
                        ) as tmp1
                        group by tmp1.bv, revMid, publicTime, ownMid
                    ) tmp4
                on watch_cnt.bv = tmp4.bv
                where relevance > 0 and ((tmp4.revMid is not null and tmp4.publicTime < now())
                    or tmp4.ownMid = real_mid
                    or (select identity from user_active where mid = real_mid) = 'SUPER')


                group by tmp4.bv, relevance, watch_cnt.cnt
                order by relevance desc, cnt desc
                limit page_size offset ((page_num - 1) * page_size)
            ) as tmpx
        );
    end $$ language plpgsql;
```

2.2.4 基于返回集合的函数的优化

在 Java 中，如果需要获取一个从 PostgreSQL 函数返回的集合，最简单的方法是将符合条件的结果 select 到一张表中，再通过 ResultSet 的 next() 方法逐一加入数组。这个方法存在一个较大的问题——当数组中的元素较多时，逐个用 next() 方法遍历元素会导致很大的时间开销。于是我使用了 PostgreSQL 中的 array_agg 函数，直接将所有的结果放在一个数组中作为一个整体返回，这样只需用调用一次 next() 方法即可获取到所有的元素。

3 其它基于安全性和鲁棒性的探究

3.1 防止 SQL 注入攻击

保障数据库安全的一个重要方面是防止 SQL 注入攻击。这里我将 SQL 注入攻击分为两种：

第一种情况是**直接将注入攻击的语句作为参入传入函数**并与函数一起执行。对于这种情况，由于注入的语句是完整的，可以直接使用 prepared statements 进行过滤。

第二种情况则更为复杂——由于所有的操作都是在 PostgreSQL 的函数中完成的，理论上用户可以通过 SQL 注入攻击的语句片段放入函数中，**通过与函数本身的内容拼接形成具有破坏性的语句**。对此，我采用了以下策略：

- 使用软删除，这样就**避免了在代码中出现** delete、drop **等具有破坏性功能的命令的出现**，一方面保证了所有数据都会留在数据库中，万一发生问题相对可控；一方面如果用户手动将 delete、drop 等关键词传入函数中，则这些关键词会被 prepared statements 识别并做转义处理，有效防止了注入攻击。

- 除此之外，在数据库函数中，对于传入字符串的使用仅包括直接使用、切分后使用和使用空格拼接后使用三种方式，保证了用户也不能通过 SQL 命令分段传入函数并执行。

3.2 信息加密存储

数据库的许多信息（例如本次项目中的密码字段）属于用户个人的隐私。为了保证用户的信息安全，即使数据库数据发生泄露，也应当尽量减少用户信息的暴露。一种处

理方法是**在数据库中使用** pg_crypto **插件里的** digest **函数结合加密算法（如** SHA256**）对用户密码等信息加密后进行存储**。

这一方法看似合理，但仍然存在风险——“黑客”可以通过监听服务器给数据库发送的数据得到加密前的密码，从而获取用户信息。所以**加密过程应当在应用层或客户端进行**，才能真正保障用户隐私的安全。

3.3 处理不同线程之间的影响

虽然这个项目的测试器是逐一对各个模块进行测试的，但在实际应用中，不可避免地会出现多个函数同时调用的情况。如果在其它函数执行的过程中相关数据被其它函数修改，那么有可能导致函数运行结果与预期不符的情况。这种情况在实际应用中对查询相关的函数影响较小（毕竟查到的结果要么是上一个时刻的，要么是下一个时刻的，都说得过去），但是对于增删改相关的函数，就有可能导致修改出错的情况。

针对这一问题，我在实体对应的表只做软删除的基础上，将所有当前存在的关系都存储在表中（即使对应的实体已经被“删除”了），在使用关系表时，仅会查询有效实体之间的关系。通过这样的设计，即使在修改关系时对应的实体有效性发生变化，系统也不会因为这次错误的插入而导致其它问题。

4 其它加速方法

4.1 GPU 加速

GPU **有着强大的任务并行执行能力**，所以常常被用于加速运算。在 PostgreSQL 中，我们可以通过 pg_storm **插件**，将 CPU 的密集型工作负载转移到 GPU 处理，提升计算效率。在进行查询时，pg_storm 会**检测**给定的查询指令是否可以全部或部分地放在 GPU 上运行，若可以则会**创建一个在** GPU **上可以运行的二进制文件源代码**。此后，pg_storm 会提取行集装载入 DMA 缓存，并异步启动 DMA 传输和 GPU 内核进行计算。

虽然 GPU 在理论上可以较大程度地提升 PostgreSQL 的运行速度，但由于其有**较高的局限性**，所以在这个项目的场景中对运行效率的提升并不明显。

4.2 “外挂”开关——full_page_writes

full_page_writes 是 PostgreSQL 的一个配置参数，可以通过命令 alter system set full_page_writes = on / off; 将其开启或关闭。在系统中，这个设置是被默认打开的，此时 PostgreSQL 服务器会**在每个检查点后的页面的第一次修改期间，将每个页面的全部内容写入到预写式日志（WAL）中**。(3)

这样做是为了防止在操作系统崩溃期间，正在进行的页面写入操作可能只完成了一部分，导致一个磁盘页面中混合有新旧数据。在系统崩溃后的恢复期间，通常存储在 WAL 中的行级改变数据可能不足以完全恢复整个页面。所以 full_page_writes 可以很好地避免这种情况的发生。

一般来说，安全性的提升会导致效率的下降，而在这个项目中，我们无需特别考虑意外掉电等会导致部分写入的特殊情况，所以不妨尝试将 full_page_writes 关闭并再次测试运行效果（由于在线测试平台不支持修改 full_page_writes 设置，所以以下测试都在本地完成）：

Task	开启设置 (ms)	关闭设置 (ms)	Task	开启设置 (ms)	关闭设置 (ms)
0	N/A	N/A	12	N/A	N/A
1	59163	49475	13	200	202
2	408	395	14	195	177
3	61	53	15	160	163
4	11	9	16	41	43
5	1628	2042	17	129	58
6	738	732	18	30	22
7	2482	1623	19	10	9
8	28459	26375	20	186	179
9	26	18	21	23	22
10	60	50	22	66	54
11	431	415	23	27	25

通过上述实验可以发现，这个方式确实可以提升各项功能的运行速度。但与此同时，PostgreSQL 的官方文档也明确指出这个操作会让数据导入的安全性降低，**违反了** ACID **原则中的** “D”，所以**在真实场景中，不应该通过关闭** full_page_writes **设置提升运行速度**。

5　项目之外：哔哩哔哩弹幕视频网的数据库设计

哔哩哔哩弹幕视频网（以下简称 B 站）的部分功能与本项目所需要实现的功能具有较高的相似度~~（三者如有相似之处，纯属巧合）~~，所以不妨通过 B 站对于相关功能的实现方案进一步探究项目中相关功能实现方案的局限性及性能提升空间。

作为一个日均 41 亿访问量的弹幕视频网站 (4)，B 站需要面临的两大难题是大量用户数据的存储和高并发的访问。为了处理这两个问题，B 站在 2018 年选择将所有数据从原先的 MySQL 数据库迁移到 TiDB 数据库。

TiDB 是一款结合了 RDBMS 和 NoSQL 的分布式 NewSQL 数据库，其结构图下图所示 (5)：

在 TiDB 中，TiDB 服务器负责接收 SQL 请求并处理 SQL 命令（功能上类似于 PostgreSQL 的服务器），通过 PD 集群获取 TiKV 的地址后与 TiKV 交互获得数据并计算出最终结果。简言之，TiDB 的主要功能为**接收命令、分析、计算，并不具有存储数据的功能**。

PD 集群的主要功能是：**存储 TiKV 集群的元信息**（如哪个键值在哪个地址），TiKV 集群的**调度和负载均衡**，以及**分配全局唯一且递增的事物** ID。

TiKV 集群的主要功能则是**存储数据**。TiKV 集群对于键值的存储**以区域**（Region）**为单位**，每个区域存储一段连续的键值信息，每台 TiKV 服务器又存储多个区域的信息。除了存储自身信息之外，TiKV 服务器还**通过** Raft **协议进行数据备份**，以保证数据的一致性和容灾能力。数据的备份及通过 PD 集群进行的调度也都以区域为单位，保证了数据的相对连续性，提高查询速度。

与 PostgreSQL 相比，TiDB 在数据存储与查询方面最大的优点在于其具有**很好的水平扩展性**——TiDB 服务器可以在几分钟内完成水平扩展，以满足快速增长的访问需求；TiKV 服务器扩展后也可以**通过** PD **集群的调度达到负载的均衡**，满足存储更多数

据的需求。正因为这一特性，B 站可以相对轻松地应对用户和访问量的快速增长，而不需要因为增加服务器而三天两头停服维护 (6)。

除此之外，相较于本项目中使用的单服务器结构，TiDB 自带的 PD 集群可以**有效地平衡不同** TiKV **集群间的负载**，大幅提升了 B 站的**并发性能**，为有效应对大型赛事直播等用户集中访问的场景提供了可靠的方案。

6 总结

在这个项目中，我以 Java 和 PostgreSQL 为基础设计了一款弹幕视频网站的数据管理系统。通过优化表和索引的结构、分批及多线程插入数据、调整 auto_commit 和 full_page_writes 设置等方式探究了优化了程序运行效率的不同方案，同时通过防止 SQL 注入攻击、应用 SHA256 加密算法等方式保障了程序运行的稳定性和安全性。

在完成效率和安全性等客观指标要求的同时，我也考虑了不使用外键、软删除等实际应用中的需求，让这个系统具有更好的实用性。

最后，我还通过 B 站使用 TiDB 管理数据的案例，分析了本项目的局限性和 TiDB 在管理数据等方面的优势，让我对于数据库的设计与应用有了更深的了解。

参考文献

[1] 阿里巴巴集团技术团队, “阿里巴巴 java 开发手册,” 2021. [Online]. Available: https://developer.aliyun.com/ebook/386/92080?spm=a2c6h.26392470.ebook-read.10.43ef21a7xZT3tp

[2] J. He, “Cpu 核心数与线程数,” 2019. [Online]. Available: https://zhuanlan.zhihu.com/p/86855590

[3] T. P. G. D. Group, “Postgresql 16.1 documentation,” p. 624, 2023.

[4] 蓝狮问道, “哔哩哔哩 2023 年用户数据分析,” 2023. [Online]. Available: https://www.bilibili.com/read/cv24819507/

[5] 技术能量站, “15 分钟了解 tidb,” 2020. [Online]. Available: https://zhuanlan.zhihu.com/p/338947811

[6] PingCAP, “Tidb x bilibili | ‘一键三连’背后的数据库,” 2021. [Online]. Available: https://cn.pingcap.com/case/user-case-bilibili/