

弹幕视频网站应用项目报告

本项目开源在[这儿](#)。

1、应用需求

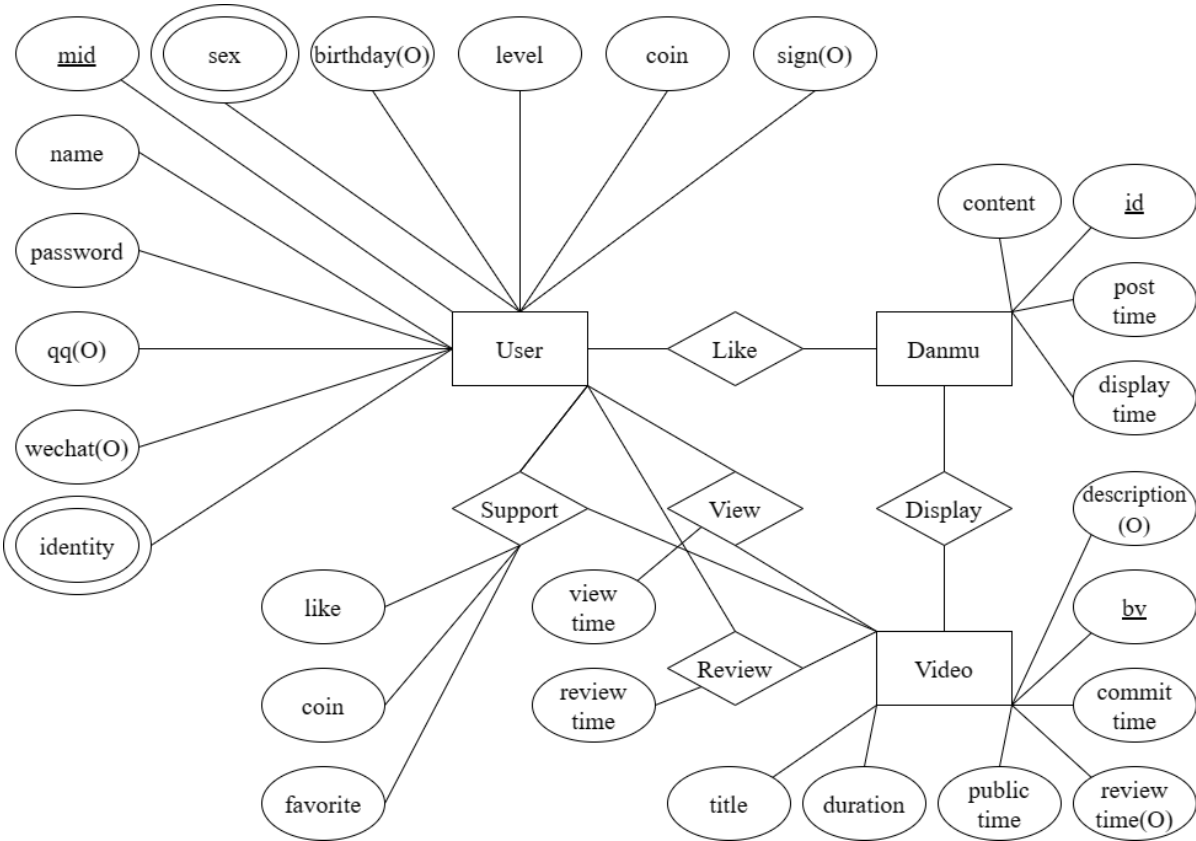
本项目要求使用SpringBoot框架与Postgresql共同实现简单的弹幕视频网站应用后端。该应用要求能处理用户、视频、弹幕三类数据，包括插入、删除、搜索和创建关系。根据现实中弹幕视频网站应用需求和提供的测试数据，用户数量应大于视频数量、弹幕数量应大于用户数量。但在本次项目中，提供的用户数据和弹幕数据均在 30000 左右，视频数据在 400 左右。因此数据库架构与更契合现实的架构略有不同。

在弹幕视频网站应用中，估计查询（如搜索、观看视频、显示弹幕）与修改（如上传视频、修改信息）业务的分布是偏向查询。因此数据库需要对高频查询有辅助表，对少量的修改可以加入触发器以简化业务逻辑。对特别高频的查询，如视频推荐等，还应当加入分层缓存，以进一步提高查询效率。

对于不同数据的关系，一个用户可以对多个用户有“关注”关系，在现实中较少有很高关注量，最高一般不超过 3000。但一个用户对视频的“观看”关系数量可以很高，例如达到 50000 以上。“点赞”、“投币”、“收藏”三个关系与“观看”类似，同样可以达到 10^4 量级。用户对弹幕的“喜爱”关系可以更高，因为可以对一个视频内多个弹幕“喜爱”。因此单用户的“喜爱”关系可以达到 10^5 量级以上。因此，对于这些基于用户的关系，需要对数据表有所分区，以加速对用户和视频的查询。

2、数据库设计

应用ER图



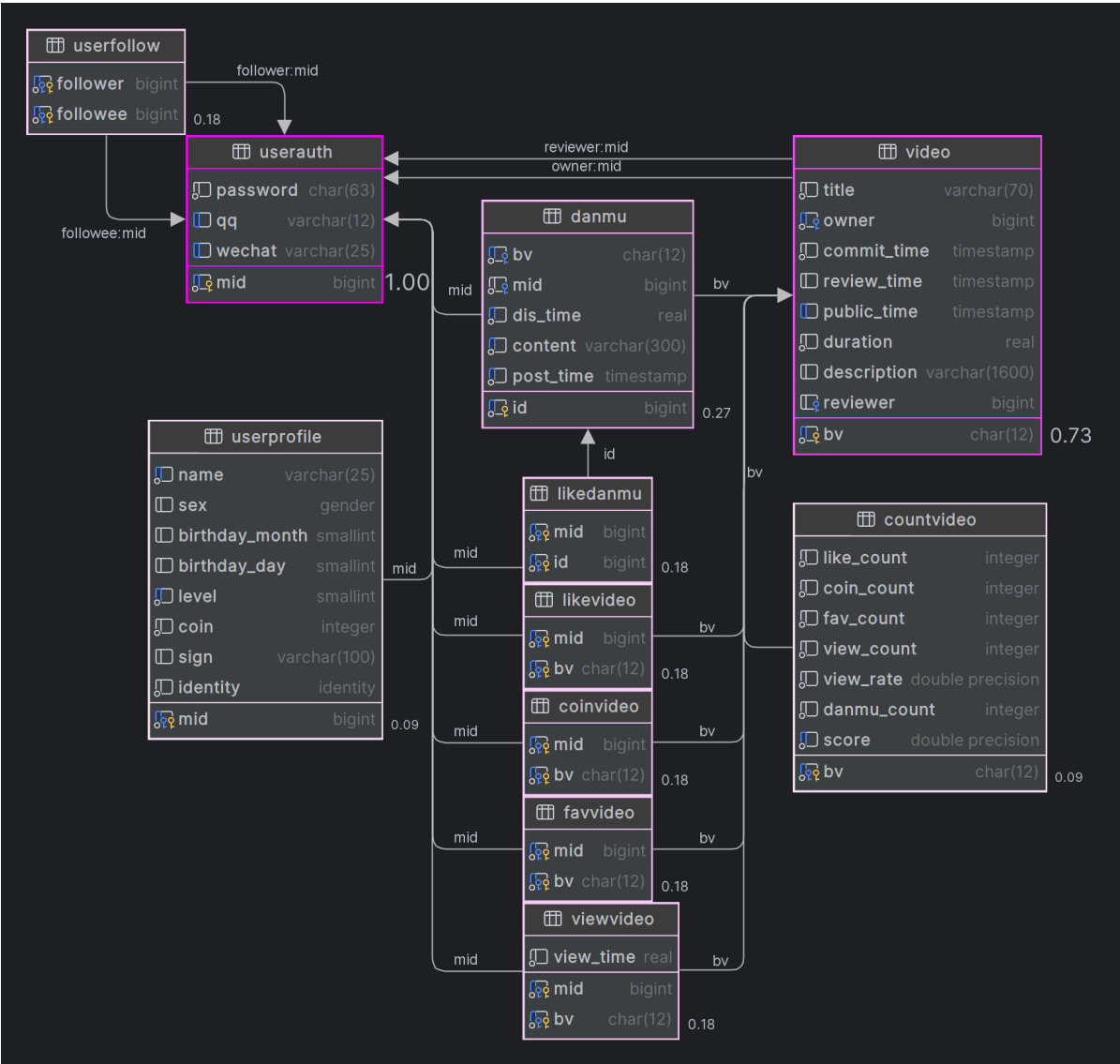
逻辑设计

基于以上考虑，我在本项目中的数据库系统架构设计如下：

注：未注明的表均为普通表。

- 静态信息表
 - 对用户创建两个表 `UserAuth` 与 `UserProfile`，前者存储 `mid`, `password`, `qq`, `wechat`，仅用于用户认证，并作为核心表，提供键值 `mid` 让其他与用户有关数据表依赖。注意，此处的 `password` 字段不存储明文密码，仅存储经过Argon2算法编码后的结果。后者存储 `mid`, `name`, `sex`, `birthday_month`, `birthday_day`, `level`, `coin`, `sign`, `identity` 为个人的所有静态信息。
 - 对视频创建一个表 `video`，存储 `bv`, `title`, `owner`, `commit_time`, `review_time`, `public_time`, `duration`, `description`, `reviewer` 为单个视频除弹幕外的所有静态信息，并作为核心表，提供键值 `bv` 让其他与视频有关数据表依赖。
 - 对弹幕创建一个表 `Danmu`，存储 `id`, `bv`, `mid`, `dis_time`, `content`, `post_name` 为一条弹幕的静态信息，并作为核心表，提供键值 `id` 让其他与弹幕有关数据表依赖。
- 关联表
 - 关联表均有分区，在部分查询中可减少单表数据量提高速度，在另一部分查询中可充分利用多线程在不同表内同时查询提高速度。
 - 对用户创建一个表 `UserFollow`，有键值 `follower`, `followee`，均依赖 `UserAuth.mid`，存储"关注"关系，且以 `follower` 为分区键基于 Hash 四分区。
 - 对视频创建四个表 `Likevideo`, `Coinvideo`, `Favvideo`, `Viewvideo`，其中 `bv` 依赖 `video`，`mid` 依赖 `UserAuth`，存储"点赞""投币""收藏""观看"四种关系，前三个表均以 `mid` 为分区键建立四分区，最后一个表建立八分区。
 - 对弹幕创建一个表 `LikeDanmu`，其中 `id` 依赖 `Danmu`，`mid` 依赖 `UserAuth`，存储"喜爱"关系，以 `id` 为分区键。此处不以 `mid` 为分区键的原因是对喜爱弹幕的统计一般出现在视频中，网页渲染会将高喜爱量的弹幕添加点赞标识。由于 `id` 在 `Danmu` 中为主键，以 `id` 分区可以提高 `Danmu` 与 `LikeDanmu` 的 JOIN 性能。
- 辅助表
 - 对视频的推荐功能创建一个表 `Countvideo`，以统计特定 `bv` 的各项数据，用于提高视频推荐结果生成速度和分页查询速度。当可能存在其他缓存技术如Redis时，应启用并对热点数据缓存。
- 缓存表
 - 视频搜索时会创建 `UNLOGGED` 类型的 `Publicvideo` 表，该表作为查询结果的缓存表，存储上一次查询的结果。
 - 在更新 `Publicvideo` 表时，在会话中会创建 `TEMP` 类型的 `Tempvideo` 表，用于关键词匹配不在缓存中的数据，之后再与缓存表合并供查询。
 - 朋友关系搜索时会创建 `UNLOGGED` 类型的 `UserFirends` 表，该表缓存是朋友关系的用户对，使用触发器更新。

数据库图



- 注：
- 1. 中间列五个相连的关联表均有对应分区表，在图中未体现。
 - 2. 各数据类型综合依照数据集和BiliBili网站限制设定。

3、业务实现

- 在任何修改数据库的操作中，均以事务包装，对任何查询命令，均不加入事务。
- 由于该应用侧重于查询，因此性能分析只针对必要的业务，省略简单语句分析。
- 对于每个业务的详细查询语句均位于 DatabaseServiceImpl 中，因此报告内省略具体查询语句内容。

数据导入

对于业务中实际调用次数极少的本模块（一般数据库初始化有且仅调用一次），我为了减少数据表间的耦合，选择对每个表单独进行初始化。注意，在输入数据集中，UserAuth.password 为明文，因此需要对其进行Argon2编码以提高安全性。由于性能测试时，编码时间也会被计入，因此在本次实现中，编码模块的参数为 saltLength = 8, hashLength = 16, parallelism = 1, memory = 128, iterations = 1。这个编码参数是不安全的，但编码速度相对较快，而默认的编码参数为 saltLength = 16, hashLength = 32, parallelism = 1, memory = 4096, iterations = 3。盐值和散列长度

减小是为了减少数据表的体积，内存使用和迭代次数减少会极大提高编码速度，但对安全性影响较高。尽管如此，在Profiling中，编码时长仍占据大量程序CPU运行时间。

此外，在导入时我使用 `COPY` 命令以加速数据插入性能，在数据表创建完毕后再创建索引和相关约束。对于每次 `COPY` 命令传输的数据量根据表的不同有所区别，静态信息表选择 1000 个数据作为一次提交，关联表选择 10000 个数据作为一次提交。分批大小的选择会一定程度影响导入性能。对于关联表，从较小的 1000 到较大的 400000 均有抽样测试，最大优化幅度是 17% 左右，体现为接近 10 秒的时间优化。

导入数据基本信息

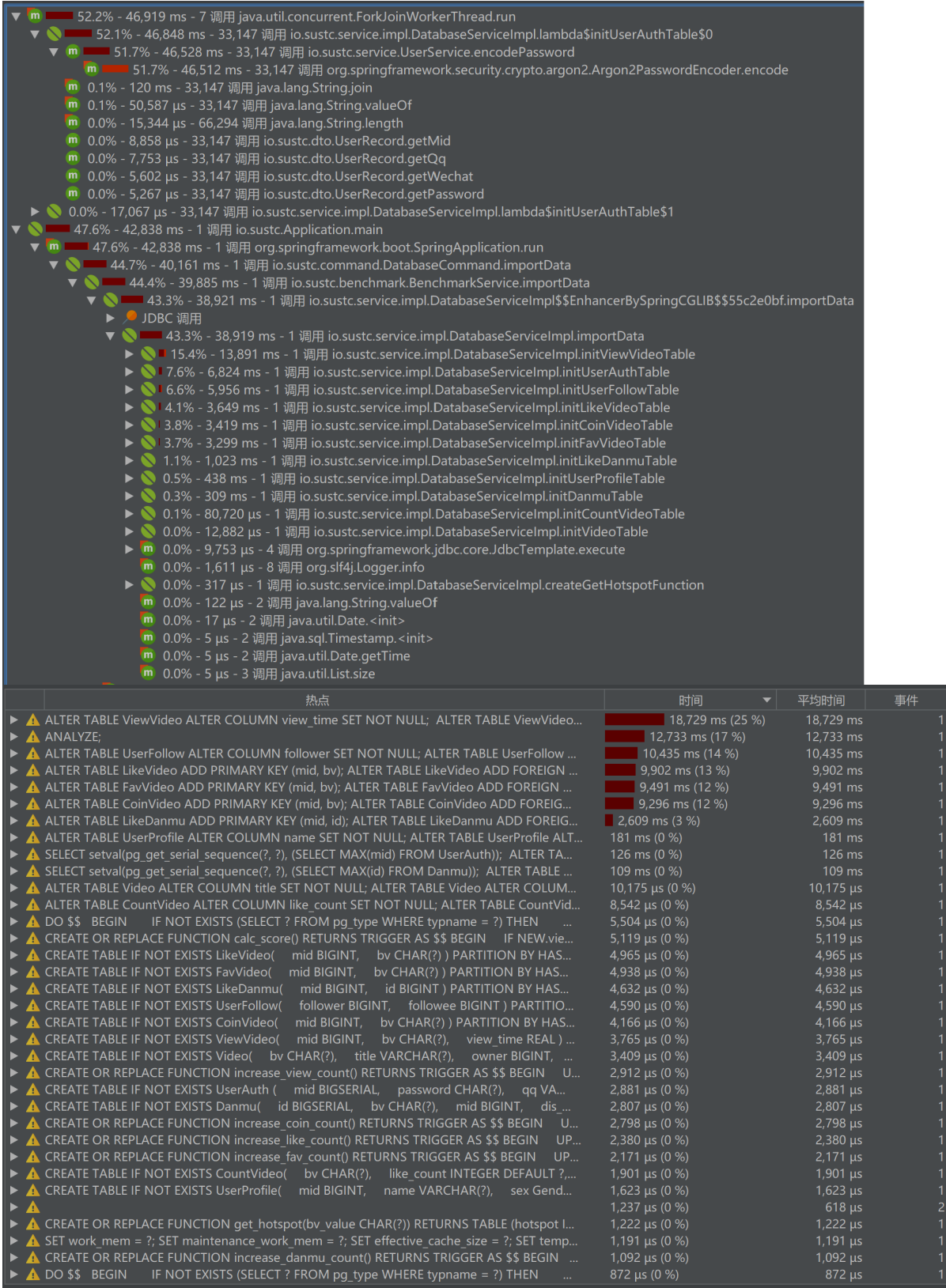
性能分析快照位于仓库中 `snapshot` 目录下。`public` 所有表空间大小（包括索引表）：2691MB。各数据表行数据量：

表名	行数据量
ViewVideo	8515433
UserFollow	5958770
LikeVideo	4501094
CoinVideo	4151635
FavVideo	4079928
LikeDanmu	1257908
UserAuth	37881
UserProfile	37881
Danmu	32672
Video	407
CountVideo	407

单线程导入性能分析

1. 根据基于JVM的插桩分析，Argon2编码时长占到总体时长 50% 以上。而且在Argon2编码实现中，我已经使用Java自带的多线程库ForkJoinWorker。在未开启采样时，受本机所限，并行库能以 6 线程执行，将CPU负载提升到 95% 以上。因此这段耗时是无法再优化，是数据导入过程的主要瓶颈。
2. 除编码外，在数据库导入时，我使用 `COPY` 命令导入数据。数据导入被捕获在显示为空白的行内，因为该命令通过jdbc的回调函数实现，无法被探针捕获。探针只能基于 `jdbc.execute()` 汇总性能分析信息，因此位于空白分组内。在此前提下，对 `viewVideo` 的限制建立过程耗时极高，`UserFollow` 和 `LikeVideo` 紧随其后。这是由于这三个表的数据量最高，需要较长时间建立索引。但由于不同表之间有层级依赖关系，例如 `viewVideo` 和 `UserFollow` 在依赖关系中处于同层。因此可以考虑部分并行，以提高数据库系统使用率，加快整体数据导入速度。
3. 观察到 `viewVideo` 的数据表过大，四分区时单表仍有 2×10^6 的数据量。因此在改进中会将该表的分区调整为八分区，以减少单表数据量，提高查询效率。

注1：性能分析基于JProfiler的插桩模式，对性能有一定影响，应关注各参数的比率而非绝对数值。
注2：此时对 `viewVideo` 的分区策略为四分区，根据性能分析结果后调整为八分区。



多线程导入性能分析

可以观察到，在根据拓扑关系使用多线程导入数据后，与数据库传输部分CPU耗时相对下降 30%，但由于数据库处理能力有限，执行命令的等待时间占比提升。在本机测试中，总运行时间为 55.741 秒。相比于导入部分全串行的运行时间 111.522 秒，加速比为 2.001，加速效率为 33.35%，实际加速效率与性能分析的估计加速性能接近，符合分析预期。其中，`ANALYZE` 命令不计入数据统计结果中，在提交代码中是不执行该语句的。提交代码的性能分析报告也在仓库目录中，由于主要瓶颈基本未发生变化，因此不重复说明。

注：多线程插桩对性能影响更高，再次强调绝对数值的意义不大。



线程状态: 线程选择: 聚合级别: 方法

	热点	时间	平均时间	事件
▶ ⚠ ALTER TABLE LikeVideo ADD PRIMARY KEY (mid, bv); ALTER TABLE Li...		27,999 ms (19 %)	27,999 ms	1
▶ ⚠ ALTER TABLE UserFollow ALTER COLUMN follower SET NOT NULL; AL...		27,770 ms (19 %)	27,770 ms	1
▶ ⚠ ALTER TABLE ViewVideo ALTER COLUMN view_time SET NOT NULL; ...		27,331 ms (19 %)	27,331 ms	1
▶ ⚠ ALTER TABLE CoinVideo ADD PRIMARY KEY (mid, bv); ALTER TABLE C...		22,418 ms (15 %)	22,418 ms	1
▶ ⚠ ALTER TABLE FavVideo ADD PRIMARY KEY (mid, bv); ALTER TABLE Fa...		16,483 ms (11 %)	16,483 ms	1
▶ ⚠ ANALYZE;		15,239 ms (10 %)	15,239 ms	1
▶ ⚠ ALTER TABLE LikeDanmu ADD PRIMARY KEY (mid, id); ALTER TABLE Li...		4,171 ms (2 %)	4,171 ms	1
▶ ⚠ SELECT setval(pg_get_serial_sequence(?, ?), (SELECT MAX(id) FROM D...		308 ms (0 %)	308 ms	1
▶ ⚠ ALTER TABLE UserProfile ALTER COLUMN name SET NOT NULL; ALTE...		204 ms (0 %)	204 ms	1
▶ ⚠ SELECT setval(pg_get_serial_sequence(?, ?), (SELECT MAX(mid) FROM ...		93,494 μs (0 %)	93,494 μs	1
▶ ⚠ CREATE OR REPLACE FUNCTION increase_like_count() RETURNS TRIG...		17,307 μs (0 %)	17,307 μs	1
▶ ⚠ ALTER TABLE Video ALTER COLUMN title SET NOT NULL; ALTER TABL...		16,176 μs (0 %)	16,176 μs	1
▶ ⚠ ALTER TABLE CountVideo ALTER COLUMN like_count SET NOT NULL;...		16,072 μs (0 %)	16,072 μs	1
▶ ⚠ CREATE TABLE IF NOT EXISTS ViewVideo(mid BIGINT, bv CHAR...		11,580 μs (0 %)	11,580 μs	1
▶ ⚠ DO \$\$ BEGIN IF NOT EXISTS (SELECT ? FROM pg_type WHERE t...		10,916 μs (0 %)	10,916 μs	1
▶ ⚠ CREATE OR REPLACE FUNCTION increase_coin_count() RETURNS TRI...		9,433 μs (0 %)	9,433 μs	1
▶ ⚠ CREATE OR REPLACE FUNCTION calc_score() RETURNS TRIGGER AS \$...		8,968 μs (0 %)	8,968 μs	1
▶ ⚠ CREATE TABLE IF NOT EXISTS LikeVideo(mid BIGINT, bv CHAR(...		8,172 μs (0 %)	8,172 μs	1
▶ ⚠ CREATE TABLE IF NOT EXISTS CoinVideo(mid BIGINT, bv CHAR(...		7,773 μs (0 %)	7,773 μs	1
▶ ⚠ CREATE TABLE IF NOT EXISTS FavVideo(mid BIGINT, bv CHAR(?...		7,623 μs (0 %)	7,623 μs	1
▶ ⚠		6,518 μs (0 %)	407 μs	16
▶ ⚠ CREATE TABLE IF NOT EXISTS LikeDanmu(mid BIGINT, id BIGIN...		6,275 μs (0 %)	6,275 μs	1
▶ ⚠ CREATE OR REPLACE FUNCTION increase_fav_count() RETURNS TRIG...		5,483 μs (0 %)	5,483 μs	1
▶ ⚠ CREATE OR REPLACE FUNCTION increase_view_count() RETURNS TRI...		5,151 μs (0 %)	5,151 μs	1
▶ ⚠ CREATE TABLE IF NOT EXISTS UserFollow(follower BIGINT, follo...		4,942 μs (0 %)	4,942 μs	1
▶ ⚠ CREATE TABLE IF NOT EXISTS UserAuth (mid BIGSERIAL, passw...		4,300 μs (0 %)	4,300 μs	1
▶ ⚠ CREATE TABLE IF NOT EXISTS Video(bv CHAR(?), title VARCHA...		4,053 μs (0 %)	4,053 μs	1
▶ ⚠ CREATE OR REPLACE FUNCTION increase_danmu_count() RETURNS T...		3,939 μs (0 %)	3,939 μs	1
▶ ⚠ CREATE TABLE IF NOT EXISTS UserProfile(mid BIGINT, name VA...		3,269 μs (0 %)	3,269 μs	1
▶ ⚠ CREATE TABLE IF NOT EXISTS Danmu(id BIGSERIAL, bv CHAR(?...		3,243 μs (0 %)	3,243 μs	1
▶ ⚠ CREATE TABLE IF NOT EXISTS CountVideo(bv CHAR(?), like_cou...		3,003 μs (0 %)	3,003 μs	1
▶ ⚠ CREATE OR REPLACE FUNCTION get_hotspot(bv_value CHAR(?)) RET...		1,760 μs (0 %)	1,760 μs	1
▶ ⚠ SET work_mem = ?; SET maintenance_work_mem = ?; SET effective_c...		1,296 μs (0 %)	1,296 μs	1

用户服务

- 注册
 - 应用层校验各项数据符合要求后，返回序列值 mid
 - 数据库操作三次 UserAuth：查询 qq 和 wechat 是否存在、查询 name 是否存在、插入新用户
- 删除
 - 身份校验
 - 数据库获取一次 UserAuth 对应字段后应用层校验
 - 身份校验后对比用户权限，返回删除结果
 - 数据库操作四次 UserAuth：查询待删除用户是否存在、查询当前权限、查询待删除用户权限、删除用户
- 获取信息
 - 应用层调用多线程查询，返回用户信息
 - 数据库操作八次
 - 一次 UserAuth 查询指定 mid 是否存在
 - 五次关联表中查询指定 mid 的所有信息（异步）
 - 一次 Video 查询指定 mid 上传的视频（异步）
 - 一次 UserProfile 查询指定 mid 硬币数量
- 关注
 - 身份校验和目标检查后，返回关注结果
 - 数据库操作三次：UserAuth 查询被关注者 mid 是否存在、UserFollow 查询当前关注关系、修改 UserFollow 为指定关系（插入或删除）

弹幕服务

- 发送弹幕
 - 身份校验、视频存在校验、观看校验
 - 数据库操作三次：`video` 获取视频持续时间、`viewvideo` 获取观看记录、插入 `Danmu`
- 显示弹幕
 - 身份校验、视频存在校验、获取弹幕信息
 - 数据库操作两次
 - `video` 获取视频持续时间
 - 查询 `Danmu`
 - 对弹幕总量最多的视频 `BV19e4y1y7HV` 执行全区间过滤查询的分析简要结果如下

```
Unique
-> Sort
"    Sort Key: content, post_time"
    Sort Method: quicksort Memory: 59kB
-> Bitmap Heap Scan on danmu
    Recheck Cond: ((bv = 'BV19e4y1y7HV'::bpchar) AND (dis_time >=
'0'::double precision) AND (dis_time <= '6688'::double precision))
    Heap Blocks: exact=7
-> Bitmap Index Scan on danmubvdistimeindex
    Index Cond: ((bv = 'BV19e4y1y7HV'::bpchar) AND (dis_time >=
'0'::double precision) AND (dis_time <= '6688'::double precision))
Planning Time: 0.215 ms
Execution Time: 0.677 ms
```

- 喜爱弹幕
 - 身份校验、弹幕存在校验、观看校验
 - 数据库操作四次：`Danmu` 查询 `id` 对应 `bv`、`viewvideo` 查询是否观看、`LikeDanmu` 查询喜爱状态、修改 `LikeDanmu` 为指定关系（喜爱或不喜爱）

视频服务

- 上传视频
 - 用户校验、视频存在校验
 - 数据库操作两次：`video` 查询指定 `title` 是否存在、插入 `video`
- 删除视频
 - 用户校验、权限检查
 - 数据库操作三次：`video` 查询 `bv` 视频拥有者、`UserProfile` 查询 `mid` 当前权限、如果允许，删除 `video` 指定 `bv`
- 更新视频
 - 用户校验、视频信息检查
 - 数据库操作三次：`video` 查询 `bv` 视频拥有者、`video` 查询 `bv` 视频原始信息、更新 `video` 指定 `bv` 如果允许
- 查询平均观看比

- 用户校验、视频观看校验
- 数据库操作两次: `video` 查询 `bv` 是否允许观看、`CountVideo` 查询 `view_rate` 和 `view_count`
- 查询视频热点
 - 用户校验、弹幕存在校验
 - 数据库操作两次
 - `Danmu` 查询指定 `bv` 是否有弹幕
 - `get_hotspot()` 自定义函数查询, 对弹幕总量最多的视频执行查询分析简要结果如下

```
CTE Scan on bvcount
Filter: (count = $1)
Rows Removed by Filter: 289
CTE bvcount
-> HashAggregate
  Group Key: (floor((danmu.dis_time / '10'::double precision))::integer
  Batches: 1 Memory Usage: 73kB
  -> Bitmap Heap Scan on danmu
    Recheck Cond: (bv = 'BV19e4y1y7HV'::bpchar)
    Heap Blocks: exact=7
    -> Bitmap Index Scan on danmubvdistimeindex
      Index Cond: (bv = 'BV19e4y1y7HV'::bpchar)
InitPlan 2 (returns $1)
-> Aggregate
  -> CTE Scan on bvcount bvcount_1
Planning Time: 0.435 ms
Execution Time: 0.543 ms
```

- 审核视频
 - 用户校验、权限校验、相关检查
 - 数据库操作四次: `UserProfile` 查询 `mid` 权限、`video` 查询 `bv` 的拥有者、`video` 查询 `bv` 是否已被审核、修改 `video` 审核指定 `bv`
- 三连
 - 用户校验、视频可三连检查、当前状态检查
 - 数据库操作四次 (投币六次)
 - `video` 查询 `bv` 拥有者
 - `video` 查询 `bv` 审核状态
 - (`UserProfile` 查询当前硬币数量)
 - 关系表查询当前状态
 - 修改关系表
 - (修改 `UserProfile` 硬币数量)
- 搜索视频
 - 对指定搜索词的查询结果建立 `UNLOGGED` 缓存表并统计
 - 数据库操作四次以上

- 创建或更新 `Publicvideo` 表，该表根据 `mid` 的可观看视频创建，简要性能分析是基于新建数据表

```
Hash Join
Hash Cond: (video.bv = countvideo.bv)
-> Nested Loop
    -> Hash Anti Join
        Hash Cond: (video.bv = publicvideo.bv)
        -> Seq Scan on video
        -> Hash
            Buckets: 1024 Batches: 1 Memory Usage: 8kB
            -> Seq Scan on publicvideo
    -> Index Scan using userprofile_pkey on userprofile
        Index Cond: (mid = video.owner)
-> Hash
    Buckets: 1024 Batches: 1 Memory Usage: 28kB
    -> Seq Scan on countvideo
Planning Time: 2.952 ms
Execution Time: 7.303 ms
```

- 若创建表则添加限制和索引，若更新表则删去不可观看、更新观看数、加入数据

- 全表更新两次，以全 `video` 表更新为例分析结果如下，结果存储于临时表 `Tempvideo` 中

```
Update on tempvideo
-> Seq Scan on tempvideo
Planning Time: 0.076 ms
Execution Time: 3.472 ms
```

- 当关键词相同时，在会话内使用临时表存储中间结果，减少对 `Publicvideo` 表的修改次数，但仍需要再将新数据插入 `Publicvideo` 中

```
Insert on publicvideo
-> Seq Scan on tempvideo
Planning Time: 0.050 ms
Trigger for constraint publicvideo_bv_fkey: time=3.721 calls=407
Execution Time: 8.367 ms
```

- 查询分页结果

```
Sort
" Sort Key: relevance DESC, view_count DESC"
Sort Method: quicksort Memory: 31kB
-> Bitmap Heap Scan on publicvideo
    Recheck Cond: (relevance > 0)
    Heap Blocks: exact=8
-> Bitmap Index Scan on publicvideoindex
    Index Cond: (relevance > 0)
Planning Time: 0.092 ms
Execution Time: 0.099 ms
```

- 注：在更新数据时，还需要对数据库操作关键词数量次，操作对象是 `Publicvideo` 或 `Tempvideo`，取决于是第一次建表或者增量更新

推荐服务

- 通用视频推荐
 - 根据辅助表 `CountVideo` 的结构，可直接根据 `score` 字段返回结果。当辅助表无变化时，未来可以结合 `WHERE` 子句过滤加速分页结果，但由于应用性质和已有的索引，该筛选不能显著提高效率
 - 数据库查询一次，使用 `LIMIT` 和 `OFFSET` 完成分页
- 相关视频推荐
 - 在 `viewvideo` 表中获取关联 `bv` 后，根据 `CountVideo` 的 `view_count` 排序返回结果
 - 数据库查询一次，对数据集播放量最高的 `BV1jL4y1e7Uz` 查询分析简要结果如下

```
Limit
InitPlan 2 (returns $1)
-> Append
    InitPlan 1 (returns $0)
        -> Append
            -> Index Scan using viewvideo_1_bv_idx on viewvideo_1
                Index Cond: (bv = 'BV1jL4y1e7Uz'::bpchar)
                此处省略其余7个分区表的记录
            -> Index Only Scan using viewvideo_1_pkey on viewvideo_1 vv_1
                Index Cond: (mid = ANY ($0))
                Heap Fetches: 0
                此处省略其余7个分区表的记录
        -> Sort
            Sort Key: cv.view_count DESC
            Sort Method: top-N heapsort Memory: 25kB
        -> Seq Scan on countvideo cv
            Filter: (bv = ANY ($1))
Planning Time: 0.803 ms
Execution Time: 2925.670 ms
```

- 用户视频推荐
 - 鉴权后先使用一次查询判断是否有感兴趣的视频，作为方法调用条件。第二次查询使用两个子查询，分别求出 `friends` 列表和 `excluded_videos` 列表，再将其与 `ViewVideo` 连接查询合法视频 `bv`。最后将该中间结果与 `Video` 和 `UserProfile` 连接查询以获取排序依据，使用 `LIMIT` 和 `OFFSET` 组合输出结果
 - 数据库查询两次，测试 `mid` 参数是关注量最多的用户之一 `362870`，`OFFSET` 参数是根据结果取最后部分为 `170`，查询分析简要结果如下：

```
Limit
-> Sort Sort Key: (count(vv.mid)) DESC, up.level DESC, v.public_time DESC
    Sort Method: quicksort Memory: 37kB
-> HashAggregate Group Key: vv.bv, up.level, v.public_time
    Filter: (count(vv.mid) > 0) Batches: 1 Memory Usage: 97kB
-> Nested Loop
    -> Hash Join Hash Cond: (vv.bv = v.bv)
        -> Hash Anti Join Hash Cond: (vv.bv = viewvideo.bv)
            -> Nested Loop
                -> Merge Join Merge Cond: (uf1.followee = uf2.follower)
```

```

-> Index Only Scan using userfollow_2_pkey on
userfollow_2 uf1 Index Cond: (follower = 362870)
-> Sort Sort Key: uf2.follower
Sort Method: quicksort Memory: 32kB
-> Append
-> Bitmap Heap Scan on userfollow_1 uf2_1
Recheck Cond: (followee = 362870)
Heap Blocks: exact=40
-> Bitmap Index Scan on
userfollow_1_followee_idx Index Cond: (followee = 362870)
此处省略其余3个分区表的记录
-> Append
-> Index Only Scan using viewvideo_1_pkey on viewvideo_1
vv_1 Index Cond: (mid = uf1.followee)
此处省略其余7个分区表的记录
-> Hash Buckets: 1024 Batches: 1 Memory Usage: 19kB
-> Index Only Scan using viewvideo_6_pkey on viewvideo_6
viewvideo Index Cond: (mid = 362870)
-> Hash Buckets: 1024 Batches: 1 Memory Usage: 34kB
-> Seq Scan on video v
Filter: ((public_time IS NULL) OR (public_time <
LOCALTIMESTAMP))
-> Memoize Cache Key: v.owner Cache Mode: logical
Hits: 1063 Misses: 153 Evictions: 0 Overflows: 0 Memory Usage: 18kB
-> Index Scan using userprofile_pkey on userprofile up
Index Cond: (mid = v.owner)
Planning Time: 3.992 ms
Execution Time: 3.517 ms

```

- 用户关注推荐

- 鉴权后，对 `UserFollow` 表查询特定关注关系，满足关注者不是指定 `mid` 的关注对象，且关注对象是 `mid` 的关注对象。对筛选出的关注关系分组统计排序，使用 `LIMIT` 和 `OFFSET` 组合输出结果
- 数据库操作一次，测试参数 `mid` 是关注量最多的用户之一 `362870`，分页选择 `OFFSET` 为 `20000`，查询分析简要结果如下：

```

Limit
InitPlan 1 (returns $0)
-> Index Only Scan using userfollow_2_pkey on userfollow_2 userfollow
Index Cond: (follower = 362870)
-> Sort Sort Key: (count(uf.followee)) DESC, up.level DESC
Sort Method: quicksort Memory: 2195kB
-> GroupAggregate Group Key: uf.follower, up.level
Filter: (count(uf.followee) > 1)
Rows Removed by Filter: 2187
-> Sort Sort Key: uf.follower, up.level DESC
Sort Method: quicksort Memory: 32691kB
-> Hash Join Hash Cond: (uf.follower = up.mid)
-> Append
-> Bitmap Heap Scan on userfollow_1 uf_1
Recheck Cond: (followee = ANY ($0))

```

Filter: ((follower <> 362870) AND (NOT (hashed SubPlan 2)))

Rows Removed by Filter: 8804

Heap Blocks: exact=7964

-> Bitmap Index Scan on userfollow_1_followee_idx

Index Cond: (followee = ANY (\$0))

SubPlan 2

-> Index Only Scan using userfollow_2_pkey on userfollow_2

userfollow_1 Index Cond: (follower = 362870)

此处省略其余3个分区表的记录

-> Hash Buckets: 65536 Batches: 1 Memory Usage: 2288kB

-> Seq Scan on userprofile up

Planning Time: 0.750 ms

Execution Time: 772.373 ms

4、样例测试

Test Step		# Case	# Passed	Std. Time (ms)	Elapsed Time (ms)
1	数据导入	N/A	N/A	65832	13351
2	视频搜索	20	20	376	346
3	平均播放率	61	61	34	19
4	视频弹幕热点	56	56	9	10
5	推荐下个视频	43	43	1729	569
6	通用推荐视频	11	11	925	3
7	用户视频推荐	27	27	3311	560
8	推荐关注用户	142	142	29219	1837
9	显示弹幕	382	382	14	58
10	发送弹幕	1016	1016	40	226
11	获取用户信息	500	500	134	338
12	收藏弹幕	500	500	N/A	N/A
13	投币	511	511	148	706
14	点赞	686	686	189	675
15	收藏	644	644	156	518
16	上传视频	184	184	42	104
17	注册用户	273	273	46	186
18	更新视频	184	184	25	81
19	审核视频	35	35	11	41
20	复查17和18	6	6	29	61
21	删除视频	55	55	362	306
22	删除用户	199	199	114	164
23	关注用户	87	87	18	40

根据样例测试的结果，我们可以得到以下表格

业务	标准平均耗时 (ms)	测试平均耗时 (ms)
数据导入	65832	13351
推荐关注用户	205.77	12.94
用户视频推荐	122.63	20.74
通用推荐视频	84.09	0.27

业务	标准平均耗时 (ms)	测试平均耗时 (ms)
推荐下个视频	40.21	13.23
视频搜索	18.8	17.3
删除视频	6.58	5.56
删除用户	0.57	0.82
平均播放率	0.55	0.31
审核视频	0.31	1.17
投币	0.29	1.38
点赞	0.27	0.98
获取用户信息	0.27	0.68
收藏	0.24	0.8
上传视频	0.23	0.57
关注用户	0.21	0.46
注册用户	0.17	0.68
视频弹幕热点	0.16	0.18
更新视频	0.14	0.44
发送弹幕	0.04	0.22
显示弹幕	0.04	0.15

在实际应用中，网络延迟通常在 $30ms$ 量级，具体波动取决于网络环境。对于标准平均耗时在 $20ms$ 以上的业务，在该数据集上有较大程度优化，符合在前文分析中的目标场景（重查询轻修改）优化分析。需要注意由于性能测试的环境设置问题，获取用户信息被迫使用串行方式，在本地测试中加入并行优化后能达到平均 $0.22ms$ 左右，与标准平均 $0.27ms$ 接近。

加速比率前五：通用推荐视频（307.33）、推荐关注用户（14.91）、用户视频推荐（4.91）、数据导入（3.93）、推荐下个视频（2.04）。

加速比率倒五：发送弹幕（-0.82）、投币（-0.79）、显示弹幕（-0.76）、注册用户（-0.75）、审核视频（-0.72）。

5、项目总结展望

1. 本项目特性

- 用户安全性高
 - 使用 `Argon2` 编码用户口令，不在数据库中以明文存储，且有较高可扩展性，能简单通过调整参数提高口令安全性。
- 多线程导入
 - 使用异步机制结合 `COPY` 命令完成数据的多线程导入，仅计算导入过程性能提升 30%。
- 各业务查询优化

- 最慢业务是相关视频推荐，最大数据平均运行时间 2 秒，冷查询时间 2.7 秒，热查询运行时间 2.3 秒，缓存内运行时间 1.6 秒。该业务可以在视频播放时同步计算，因此用户的体感等待时间（指无事可做等待系统刷新）可忽略。
- 次慢业务是用户关注推荐，最大数据平均运行时间 0.7 秒，冷查询时间 0.8 秒，缓存内运行时间 0.6 秒。该业务运行时间与用户加载正常大小图片时间接近，略有等待感。
- 其余业务运行时间均在 0.1 秒以内，在实际业务中不会出现由数据库服务导致的卡顿。
- 应用架构清晰
 - 与数据库连接的有关服务均在 `DatabaseService` 类中，限制应用访问数据库的范围。异步方法在独立类中实现，提高可维护性和可调试性。
 - 对可能修改数据库的语句，均使用事务包裹，保证数据一致性和完整性。所有服务均有完整的输入校验和日志记录。

2. 本项目总结

- 本次项目对弹幕视频网站应用搭建了基于 `Postgresql` 的示例，对数据库系统有更深入的实操和学习。在业务的实现中，缓存和并行化是两个效果立竿见影的优化策略。在数据库查询语句的编写中，查询计划能帮助判断语句的准确性、分析性能瓶颈。
- 在视频搜索业务中，我认为关系型数据库不能很好处理文本匹配功能，在效率上不如其他系统，例如 `ElasticSearch` 等。
- 本次实现的应用偏向优化查询，如何优化修改并没有深入思考，这是另一个可能的学习方向。

3. 对应用的展望

- 对于现实的弹幕视频网站，庞大的视频、弹幕、评论数据量是比用户信息更难处理的部分。这需要借助大数据系统处理非结构化数据，并对其进行数据分析。大量数据还会带来高并发需求，数据系统要有良好的处理性能、稳定性、容错性、水平扩展性。同时，业务的实时性和准确性不可能均满足，实际需要对其有更深入的研究，在不影响用户体验的前提下找到平衡点。