SUSTech CS213 Project

Student Name: 刘淦

SID：12210729

数据库原理

Database Design

该数据库是以用户为中心设计的数据库结构。数据库整体包含三个主要部分，用户、视频和弹幕。对于用户，member 表格包含了用户的基本信息，我们用 follow 表格来记录关注者和被关注者，联系不同 member 之间的关系。对于视频，video 记录了视频的基本信息，其下的 reviewVideo, coin, viewVide 等表格用于记录观看、收藏、投币等用户和视频的交互信息。对于弹幕，bv 和 mid 信息标记了其所属视频和弹幕作者，id 作为弹幕的唯一标识，弹幕下有一个 likedanmu 表格，用于记录对弹幕的点赞信息。我们用外键连接了各表格，使删除更加方便和高效。

Figure 2 是数据库各个实体之间的关系图。如图，数据库中有三个实体：member, danmu 和 video。每个实体下有多个属性，属性和属性之间的关系也已在图上标识显示。

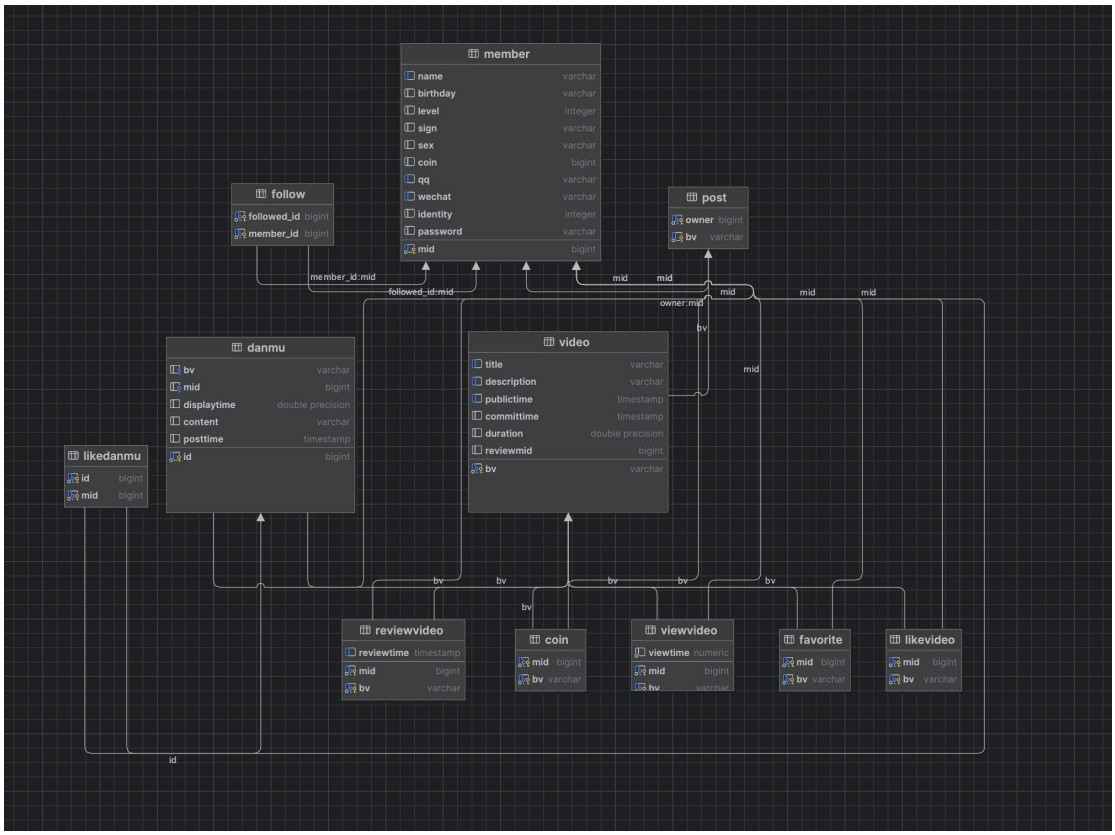

Figure 1: Visualization

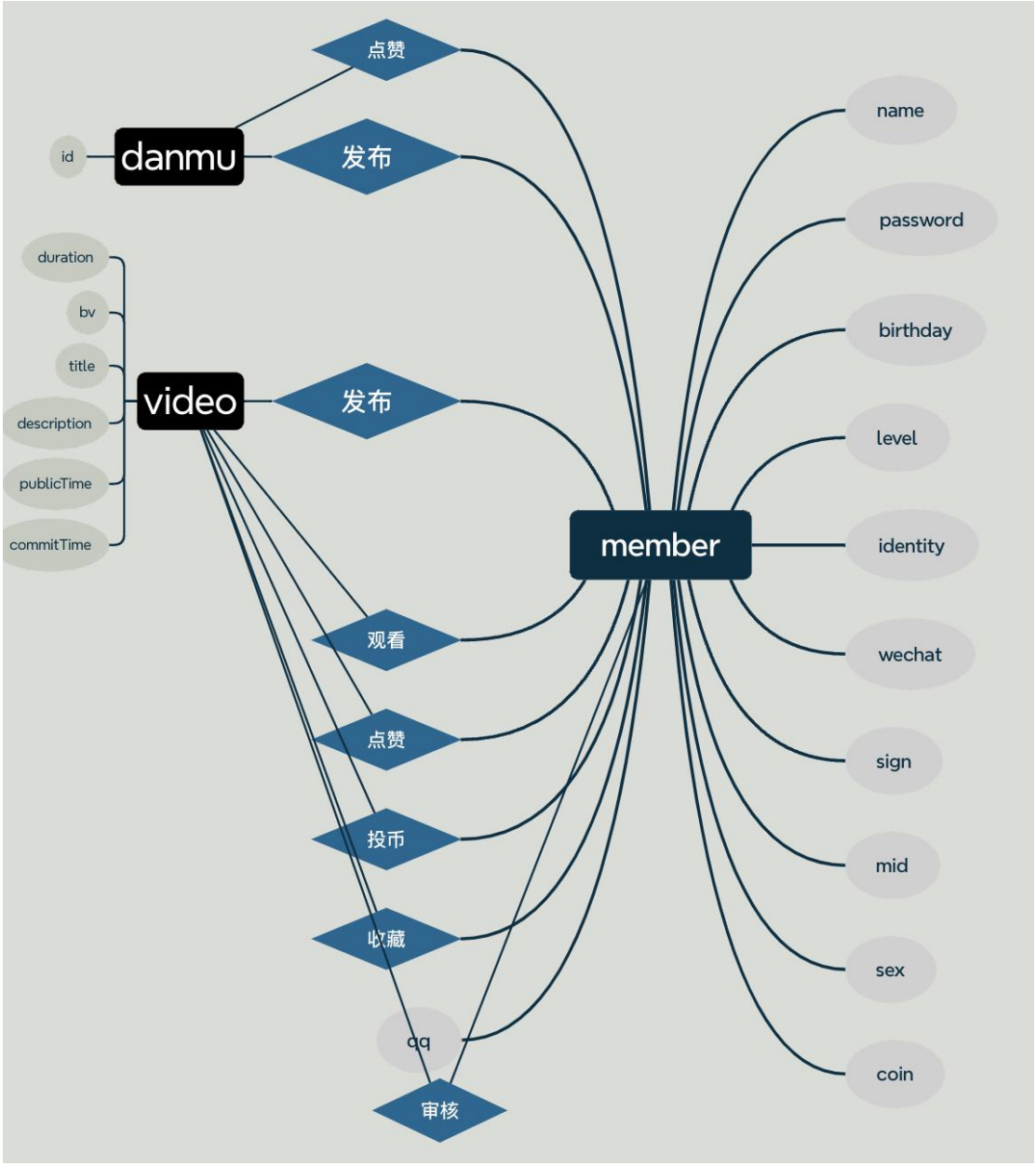

Figure 2：ER 关系图

Optimization

1.缓存

当热点事件发生时，存在热点人物的信息被频繁访问的情况。然而，在没有优化的情况下，用户的每一次访问都需要调取连接，数据库再从磁盘中读取信息，大量的 I/O 会造成严重的卡顿，我们可以在通过缓存，指定某些数据在查询后能够进入缓存中，这样，在下一次查询该数据时，可以直接从缓存中读取该数据。

1.1 实现

缓存的实现利用了 caffeine 本地缓存组件，在 io.sustc.api.config package 中，我们编写了 cacheConfig 类，注解上@EnableCaching 对缓存进行配置，设置好销毁时间。

接着，对于我们想要缓存的数据，例如：userInfo，我们在对应的 get 方法前加上 cacheable 注解，定义好对应的 value 和 key。在下一次对该数据进行读取时，程序会首先从缓存读取判断是否存在含相同 key 值的数据。

```
@Override
@Cacheable(value = "userInfo",key = "#mid")
public UserInfoResp getUserInfo(long mid) {...}
```

1.2 实验

测试缓存对于频繁对相同数据进行查询的优化情况，我们编写了一个测试 Benchmark,模拟在短时间内多次访问相同用户信息的情况。我们分别测试了 1000 次查询情况下和 100 次连续查询的情况下启动前后的访问速度对比。最终得出如 Figure 3 的结果，如图所示，无论查询数据量大小，启动缓存的查询速度都要优于未启动时的情况，并且随着查询次数的上升，启动缓存的优势更加明显。

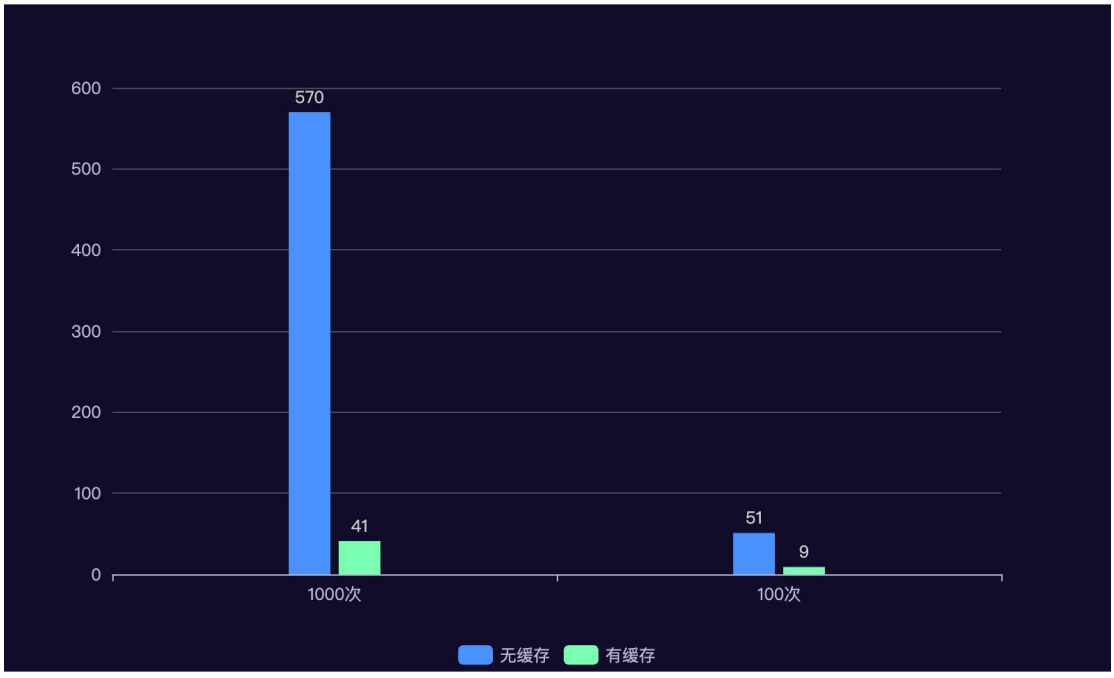

Figure 3：缓存启动前后查询性能情况

2.数据库连接池

由于在 API 内部，需要对传入参数进行检查，涉及到多次和数据库的连接与关闭连接。此时，加入数据库连接池，能够使 API 在调用 connection 时是直接从连接池中调用，减少了连接时间。

在 io.sustc.config package 中，有 DataSourceConfig 类，我们在此处对数据库连接池进行配置。并为 dataSource 注解上@Bean、 @Primary，使得 spring 能够将连接池注入到不同实现类中。

```
@Bean
@Primary
public DataSource dataSource() {
    DruidDataSource datasource = new DruidDataSource();
    datasource.setUrl(url);
    datasource.setUsername(username);
    datasource.setPassword(password);
    datasource.setDriverClassName(driverClassName);
    datasource.setInitialSize(initialSize);
    datasource.setMinIdle(minIdle);
    datasource.setMaxActive(maxActive);
    datasource.setMaxWait(maxWait);
    datasource.setTimeBetweenEvictionRunsMillis(timeBetweenEvictionRunsMillis);
    datasource.setMinEvictableIdleTimeMillis(minEvictableIdleTimeMillis);
    datasource.setValidationQuery(validationQuery);
    datasource.setTestWhileIdle(testWhileIdle);
    datasource.setTestOnBorrow(testOnBorrow);
    datasource.setTestOnReturn(testOnReturn);
    datasource.setPoolPreparedStatements(poolPreparedStatements);
    datasource.setMaxPoolPreparedStatementPerConnectionSize(maxPoolPreparedStatementPerConnectionSize);
    try {
        datasource.setFilters(filters);
    } catch (SQLException e) {
        System.err.println("druid configuration initialization filter: " + e);
    }
    datasource.setConnectionProperties(connectionProperties);


    return datasource;
}
```

3.索引和联合索引

由于涉及大量的查询，有良好的索引结构能够将查询时间从 O(n)的时间复杂度降低到 O(logn)。经过对常用查询和数据表的结构分析，我构建了如下的索引。然而，单列索引有时的效率仍然比较低效，例如对 follow 表格的查询，即使对 followed_id 列创建了索引，数据库可能仍然要对 member_id 进行序列扫描，因此，我选择了建立联合索引，加快了包括 follow 在内的多个表格的查询速度。

3.1 实现

根据实际的业务情况，构建了如下若干个索引及联合索引。

```
create index mid_post on post(owner);
create index bv_post on post(bv);
create index bv_review on reviewVideo(bv);
create index time_review on reviewVideo(reviewTime);
create index title on video(title);
create index des on video(description);
create index na on member(name);
create index time_video on video(publicTime);
create index bv_video on video(bv);
create index bv_fa on favorite(bv);
create index bv_coin on viewVideo(bv);
create index bv_vv on coin(bv);
create index mid_vv on viewVideo(mid);
create index mid_followed on follow(followed_id);
create index mid_member on follow(member_id);
create index cluster_index on follow(member_id,followed_id);
create index cluster_index1 on viewVideo(bv,mid);
create index cluster_index2 on likeVideo(bv,mid);
create index cluster_index3 on viewVideo(bv,mid);
create index mid_mm on member(mid);
create index wechat_mm on member(wechat);
create index qq_mm on member(qq);
```

3.2 实验

我们测试在无索引、有索引和有二级索引的情况下，业务的查询性能情况。我们对 recommendFriend 服务进行测试，对于这一服务，由于 following 的表数据量极大，无索引情况下，需要进行全表扫描，时长极大。

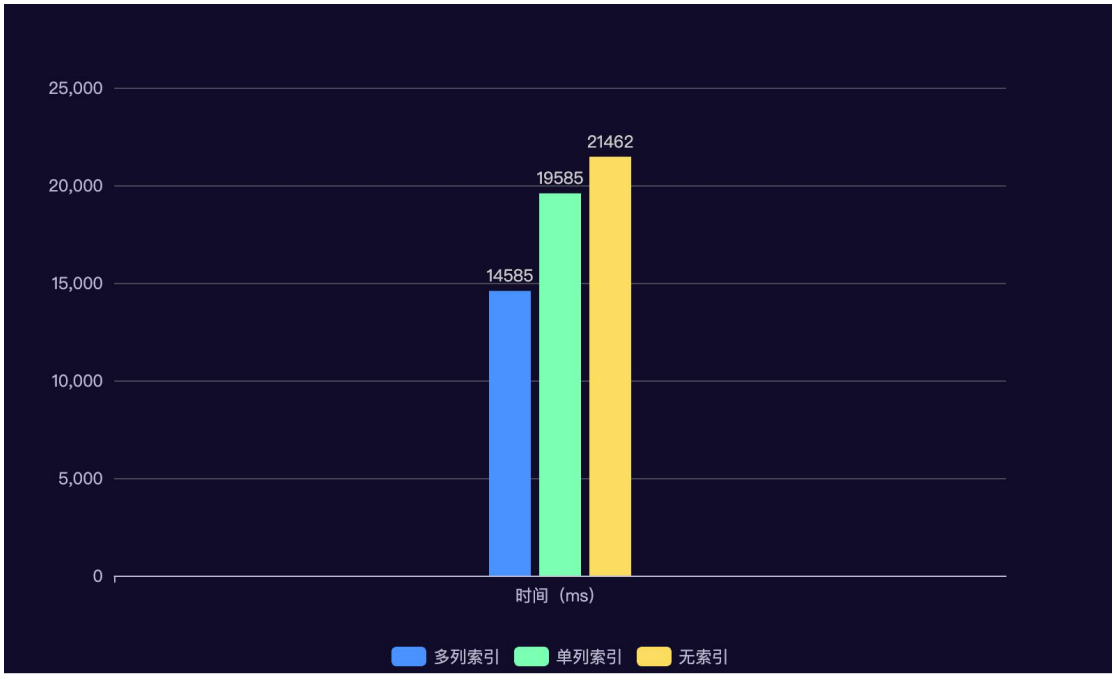

Figure 4：索引对性能的影响

4.表分区

根据实际情况，following 表格和 viewVideo 表格的单表数据量都比较大，即便有索引，查询性能也比较低，因此我们根据 mid 的大小对 viewVideo 表格进行划分。为了保证子表和父表的一致性，需要建立一个 trigger function，在每次插入到父表前，将数据插入子表。触发器如下：

```
CREATE OR REPLACE FUNCTION partition_trigger()
  RETURNS TRIGGER AS $$
BEGIN
  IF NEW.mid < 334268
  THEN
    INSERT INTO tb11_partition(mid, bv, viewTime) VALUES (new.mid,new.bv,new.viewTime);
  ELSIF NEW.mid >=334268 AND NEW.mid < 502028
    THEN
      INSERT INTO tb10_partition (mid, bv, viewTime) VALUES (new.mid,new.bv,new.viewTime);
  ELSIF NEW.mid >=  502028 AND NEW.mid < 672112
    THEN
      INSERT INTO tb9_partition (mid, bv, viewTime) VALUES (new.mid,new.bv,new.viewTime);
 ELSIF NEW.mid >=  672112 AND NEW.mid < 866382
    THEN
      INSERT INTO tb8_partition (mid, bv, viewTime) VALUES (new.mid,new.bv,new.viewTime);
  ELSIF NEW.mid >=866382 and new.mid < 1340442
    THEN
      INSERT INTO tb7_partition (mid, bv, viewTime) VALUES (new.mid,new.bv,new.viewTime);
ELSIF NEW.mid >=1340442 AND NEW.mid < 1504758
    THEN
      INSERT INTO tb6_partition (mid, bv, viewTime) VALUES (new.mid,new.bv,new.viewTime);
  ELSIF NEW.mid >=  1504758 AND NEW.mid < 1654938
    THEN
      INSERT INTO tb5_partition (mid, bv, viewTime) VALUES (new.mid,new.bv,new.viewTime);
 ELSIF NEW.mid >=  1654938 AND NEW.mid < 1805451
    THEN
      INSERT INTO tb4_partition (mid, bv, viewTime) VALUES (new.mid,new.bv,new.viewTime);
  ELSIF NEW.mid >=1805451 and new.mid <1969303
    THEN
      INSERT INTO tb2_partition (mid, bv, viewTime) VALUES (new.mid,new.bv,new.viewTime);
  elsif new.mid   >=1969303 then
     insert into tb1_partition(mid, bv, viewTime) VALUES (new.mid,new.bv,new.viewTime);
END IF;
  RETURN NULL;
END;
```

```
CREATE TRIGGER insert_trigger
BEFORE INSERT ON viewVideo
FOR EACH ROW EXECUTE PROCEDURE partition_trigger();
```

5.多线程

在导入数据时，用户和视频的信息量的比较大，异步地导入会延长导入的时间，为了充分利用好计算机资源，我们多开若干线程，使得信息能够同时导入。如下所示：

```
Thread thread8 = new Thread(()-> importV1(videoRecords));
Thread thread9 = new Thread(()-> importV2(videoRecords));
Thread thread10 = new Thread(()-> importV3(videoRecords));
Thread thread11 = new Thread(()-> importV4(videoRecords));
Thread thread12 = new Thread(()->importV5(videoRecords));
Thread thread13 = new Thread(()->importV6(videoRecords));
Thread thread14 = new Thread(() -> importDanmu(danmuRecords));
Thread thread15 = new Thread(()->importV21(videoRecords));
Thread thread16 = new Thread(()->importV11(videoRecords));

thread8.start();
thread9.start();
thread10.start();
thread11.start();
thread12.start();
thread13.start();

thread15.start();
thread16.start();
```

由于各个方法能够独立并行运行，最终 import data 的时间长度等于所有方法中运行时长最长的方法，我们将导入时间从 90s 提升到 45s。

6.外键

外键能够更好地标识不同表格之间的关系，同时，能够加快删除速度。例如，在删除用户账号时，我们只需一条 SQL 语句：

```
delete from member where mid = ?;
```

由于外键盘的存在和 ON DELETE CASCADE 的约束，数据库能够自动将其他关联数据同时删除。

然而，外键的存在也会极大降低大量插入数据时的效率，所以，在大量导入数据时，我们首先将所有的外建约束删除，在完成插入后，我们再重新建立约束。