数据库原理（H）Project 2 报告

吴梦轩

12212006

1 数据库设计

1.1 实体关系图

我使用的是 crow's foot 符号，并将实体表的表头用单独的颜色标出，关系表的表头统一用紫色标出。

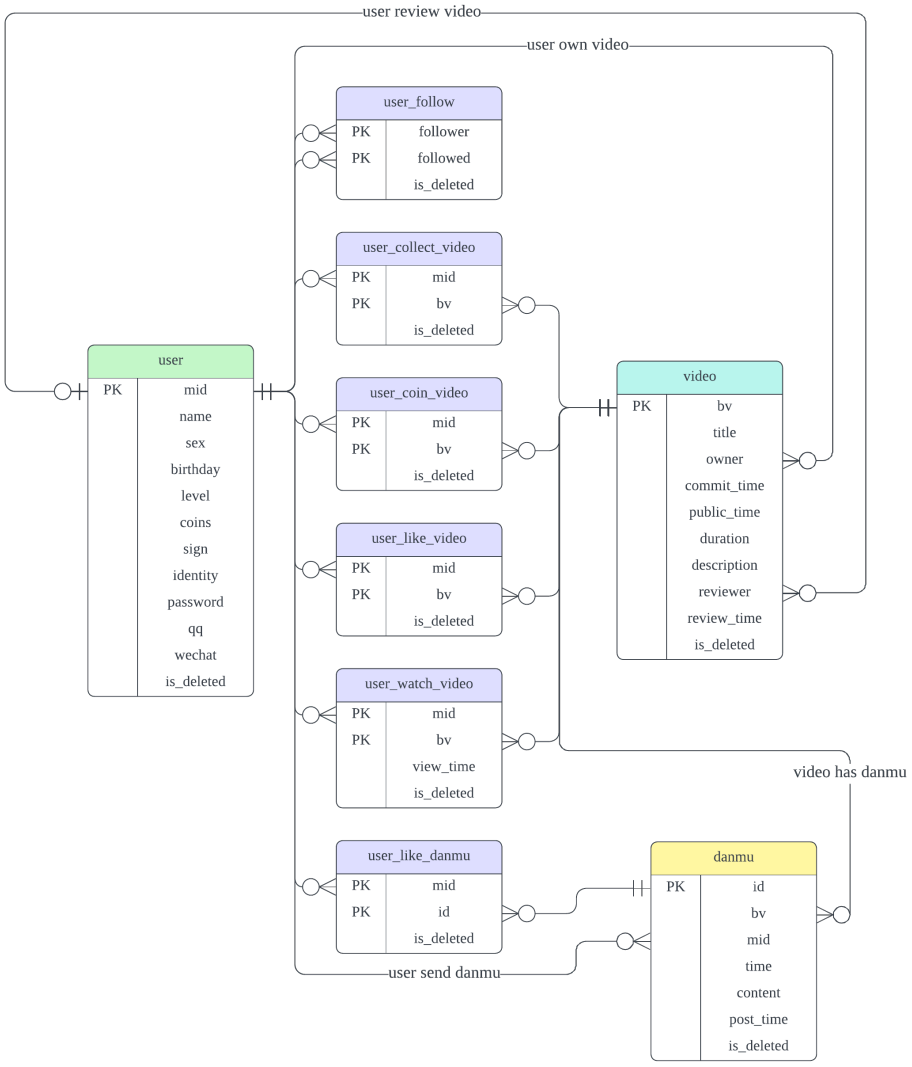

图 1: 实体关系图

1.2 数据库可视化

由于设计的数据库不使用外键，所以在可视化时，我将实体表和关系表分开显示。

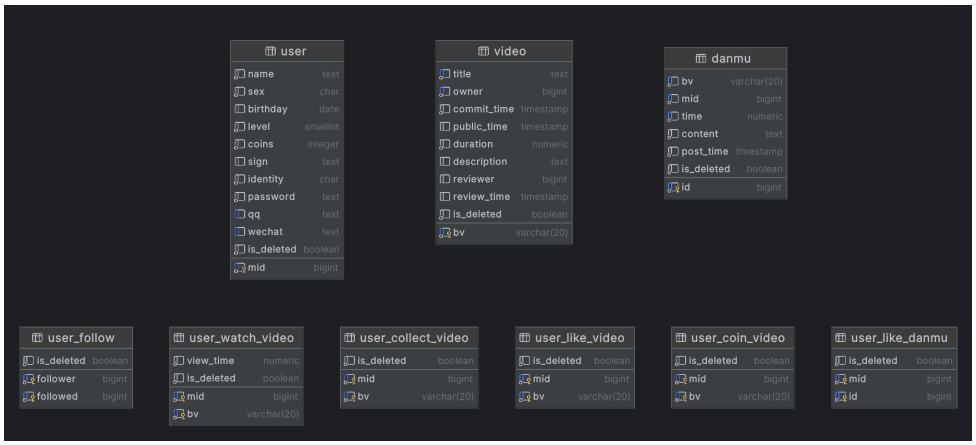

图 2: 数据库可视化

1.3 数据库设计简述

本次设计中，我查阅并尽可能遵守了阿里巴巴 Java 开发手册[1]中的规范，具体见下文。

本数据库共有 3 个实体表和 6 个关系表,其中实体表为 `user`、`video`、`danmu`,关系表为 `user_like_danmu`、`user_watch_video`、`user_coin_video`、`user_like_video`、`user_collect_video` 和 `user_follow`，通过表名表示了实体之间的关系。此设计遵守阿里巴巴 Java 开发手册中的规范有：

- 表名、字段名必须使用小写字母或数字，禁止出现数字开头
- 表名不使用复数名词

考虑到本数据库是为大型公司而设计，需要存储大量的数据且有数据审计的需求，我采用了无外键，软删除的设计。其具体做法为：在每个实体表和关系表中都加入了一个 `is_deleted` 字段，默认为 `false`，用于标记该条记录是否被删除。在删除实体时，手动更新该实体及其相关的关系表中的所有记录的 `is_deleted` 字段为 `true`。为保证查询效率，我使用了部分索引，即只对 `is_deleted` 字段为 `false` 的记录建立索引。此设计遵守阿里巴巴 Java 开发手册中的规范有：

- 表达是与否概念的字段，必须使用 `is_xxx` 的方式命名
- 不得使用外键与级联

在数据库权限管理方面，由于我将每个功能都封装成了一个数据库 `function`，所以只需要对于不同的 `role` 赋予对应的 `function` 的 `execute` 权限即可。具体而言，可以考虑以下几种 `role`：

- 访客：查询弹幕 `displayDanmu`、获得推荐视频 `generalRecommendation` 和注册 `register` 权限
- 普通用户：在访客的基础上，获得投稿视频 `postVideo`、更改视频信息 `updateVideoInfo`、投币 `coinVideo`、点赞 `likeVideo`、收藏 `collectVideo`、关注 `follow`、发送弹幕 `sendDanmu`、点赞弹幕 `likeDanmu`、获得个性化推荐视频 `recommendVideoForUser`、获得朋友推荐 `recommendFriend`、删除视频 `deleteVideo` 和删除账号 `deleteAccount` 权限（此处只能删除自己的视频和账号）
- 超级用户：在普通用户的基础上，获得审核视频 `reviewVideo` 权限，且此时可以删除任意视频和任意普通用户的账号

[1]Java 手册界面 - 阿里云开发者社区，`https://developer.aliyun.com/special/tech-java`

- 数据库管理员/数据分析师：在超级用户的基础上，获得检查用户是否合法 `validateUserInfo`、查看视频热点 `getHotSpot` 和查看视频观看率 `getAverageViewRate` 权限，同时拥有所有表的 select 权限

本数据库遵守了第三范式，即每个非主键属性都不传递依赖于主键，每个非主键属性都直接依赖于主键。

2 数据库特殊实现

我的报告主要包含以下几个部分，其主要部分为探索软删除的实现方式以及两种实现方式的优缺点。结构如下：

- 软删除的设计
  - 历史表方法
    - Bloat 膨胀问题
    - Bloat 膨胀的原因：对 PostgreSQL 的 MVCC 机制的探索
  - 标记删除方法
    - 通过部分索引保证查询效率
    - 额外探索：Postgres Query Planner 如何使用索引
- 数据库性能优化
- 数据库安全性优化
- 其他优化与设计

2.1 软删除

软删除 (Soft Delete) 是相对于硬删除 (Hard Delete) 来说的，又称逻辑删除，指的是在数据库中，不是真正地删除数据，而是通过某种标记使得数据被过滤掉，从而达到删除的效果。软删除的存在很多时候不仅是为了数据审计或数据恢复，而是因为真实世界中往往不存在真正的删除，而是数据发生了存在状态的改变，并且在新的状态下可能产生了新的价值。Udi Dahan 曾发表文章 *Don't Delete - Just Don't* [2]来阐述这一观点。

- 订单不是被“删除”的，而是被取消的，并且可能由于过晚取消而产生了违约金
- 员工不是被“删除”的，而是被解雇的，并且公司可能需要为其支付一定的补偿金
- （需要招聘的）职位不是被“删除”的，而是被填补了或者不再需要了

并且，即使被删除的数据不会产生新的价值，也并不能简单的级联删除：

- 问题：如果一个商品被下架了，我们能够直接级联删除它吗？
  - 答：不能。如果我们直接级联删除了该商品，那么所有购买了该商品的订单、仓库中该商品的库存记录以及该商品的进货记录都会被删除，这显然是不合理的。

[2]Udi Dahan, *Don't Delete - Just Don't*, `https://udidahan.com/2009/09/01/dont-delete-just-dont/`

Udi Dahan 最后总结道：**The real world doesn't cascade!** 我们需要更合理的方式来处理数据的删除，也就是软删除。

我对软删除的实现方式进行了探索，主要有两种方法：

1. 为每个表建立一个新的历史表，用于存储被删除的记录，使用 `trigger` 将删除的记录插入到历史表中
2. 在每个表中加入一个新字段，用于标记该条记录是否被删除

接下来我将对这两种方法进行详细的说明。

2.1.1 方法一：历史表

采用历史表的方式是最为简单的，我在最初的设计中也想过采用这种方式。其优点是：

- 逻辑清晰，易于理解
- 软删除的记录和原记录分开存储，不会影响查询未删除记录的效率
- 不需要对 SQL 语句进行修改，每次删除时 `trigger` 自动将记录插入到历史表中

看上去这种方式是完美的，但是我在 lab 课学习了如何查看表和索引的大小后，意外发现了一个问题，并最终让我放弃了这种方式。

为什么删除记录后表的大小没有变化？

对此我进行了一些实验：创建一个 `exp` 表，包含两个字段：主键 `id` 和 `name`，并插入了一百万条记录。

```
CREATE TABLE exp
(
    id   INT PRIMARY KEY,
    name VARCHAR(20)
);

INSERT INTO exp
SELECT i, (floor(random() * 1000))::text
FROM generate_series(1, 1000000) as i;
```

然后我删除了 id 为偶数的记录，并记录了删除前后表与索引的大小。

```
DELETE
FROM exp
WHERE id % 2 = 0;

SELECT pg_size_pretty(pg_relation_size('exp')) AS table_size,
    pg_size_pretty(pg_indexes_size('exp')) AS index_size;
```

结果发现，删除前后表和索引的大小完全没有变化！

操作	表大小	索引大小
删除前	35 MB	21 MB
删除后	35 MB	21 MB

经过搜索，我发现了原因：**PostgreSQL 不会立即回收删除的空间**，已删除的记录仍然被保留在表中，直到 `VACUUM` 命令运行时才会被回收，这导致表的大小不会立刻减少。这种现象被称为 **Bloat**，即膨胀。这是为了 MVCC (Multi-Version Concurrency Control，多版本并发控制) 的实现而做出的妥协。

为了更好的说明这一现象，我将首先介绍 MVCC 的实现原理:

当很多人同时操作数据库时，可能同时存在读写操作。如果同一条记录在被读取时被修改了，那么读取到的数据可能是不完整的，或者是不一致的。

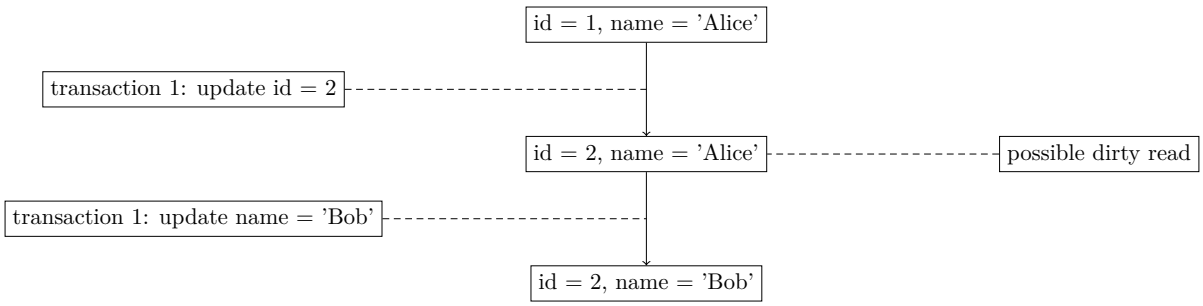

图 3: 不约束并发读写可能导致的问题

为了避免这种情况的发生，数据库需要对读写操作进行并发控制。最简单的解决方式是将正在被修改的记录锁定，不允许读取，直到修改完成后才解除锁定，但是这种方式会导致并发性能极差。因此，PostgreSQL 采用了 MVCC 的方式来实现并发控制：

- 每个 `transaction` 都有一个唯一的 `transaction id` (又称 `xid`)，用于唯一标记该 `transaction`
- 当 `transaction` 对某一行记录进行修改时，将为该行记录创建一个新版本。只对新版本进行修改，旧版本将被保留但标记为已删除
- 在底层，每个版本由一个 tuple 对应存储。每个 tuple 的 header 中都有两个字段：`xmin` 和 `xmax`，分别表示创建该 tuple 的 `xid` 和删除该 tuple 的 `xid`，未被删除的记录的 `xmax` 为 `0`。
- 每个 `transaction` 通过读取 tuple 的 header，可以判断该 tuple 是否在该 `transaction` 中可见

这样，由于任何修改操作都不会直接修改原有的记录，所以读取操作不会被阻塞，从而实现了并发控制。我将通过 `pageinspect` 扩展来查看 tuple 的各项属性，并配合实验阐释 MVCC 的实现原理。

先创建一个 `transaction_test` 表，并插入一条数据 id = 1, name = 'a'。

```
CREATE TABLE transaction_test
(
    id   INT PRIMARY KEY,
    name VARCHAR(20)
);

INSERT INTO transaction_test
VALUES (1, 'a');
```

使用 `pageinspect` 扩展查看该表

```
SELECT lp, t_xmin, t_xmax, t_ctid, t_infomask
FROM heap_page_items(get_raw_page('transaction_test', 0));
```

可得：

lp	t_xmin	t_xmax	t_ctid	t_infomask	id	name
1	71986	0	(0,1)	2050	1	a

我手动加入了 tuple 对应的内容，方便理解

表格解读：

- `lp` 为该 tuple 在当前 page 中的编号，目前为第一个 tuple。
- `t_ctid` 指向了当前 tuple 最新的版本的位置，当前指向第 0 个 page 的第 1 个 tuple，即指向了自己。
- `t_infomask` 的解读较为复杂，该字段可以解读出 `t_xmin` 与 `t_xmin` 各自对应的 `transaction` 是否已经提交或被放弃 (rollback)，具体为：将其转化为二进制后，若第 9 位为 1，则 `t_xmin` 对应的 `transaction` 已经提交，若第 10 位为 1，则 `t_xmin` 对应的 `transaction` 被放弃，若第 11 位为 1，则 `t_xmax` 对应的 `transaction` 已经提交，若第 12 位为 1，则 `t_xmax` 对应的 `transaction` 被放弃。目前值为 2050，可知 `t_xmax` 对应的 `transaction` 被放弃，即该记录未被删除。

接下来，我将在 t_xmin 和 t_xmax 旁用 a 表示该 transaction 被放弃，用 c 表示该 transaction 已经提交，用 r 表示该 transaction 正在运行。

然后我开启两个 `transaction`，并使用 `txid_current()` 函数查看当前 `transaction` 的 `xid`。得知 `transaction 1` 的 `xid` 为 `71989`，`transaction 2` 的 `xid` 为 `71990`。在 `transaction 1` 中，我将 id 为 1 的记录的 name 修改为'b'，此时再次查看该表，可得：

lp	t_xmin	t_xmax	t_ctid	t_infomask	id	name
1	71986(c)	71989(r)	(0,2)	258	1	a
2	71989(r)	0(a)	(0,2)	10242	1	b

可以看到，因为 `transaction 1` 修改了 id 为 1 的记录，所以该记录被复制了一份，放入到了第 2 个 tuple 中。原本的记录已经过时，应该被标记为删除，所以其 `xmax` 被设置为 `71989`，表示该记录被 `transaction 1` 删除。同时，第 1 个 tuple 的 `t_ctid` 被修改为 (0,2)，指向了第 2 个 tuple，表示第 1 个 tuple 代表的该行记录的最新版本为第 2 个 tuple。

在 `transaction 2` 查看该表时，从编号为 1 的 tuple 的 `t_infomask` 中得知，该记录虽然已经被标记为删除 (`xmax` 不为 0)，但是其 `xmax` 对应的 `transaction` 还在运行，所以该记录仍然可见。从编号为 2 的 tuple 的 `t_infomask` 中得知，该 tuple 的 `xmin` 对应的 `transaction` 还在运行，即该记录还未提交，所以对 `transaction 2` 来说，这条记录不可见。此时就实现了 MVCC 的并发控制：`transaction 1` 修改后，只能看到 tuple 2，但此时 `transaction 2` 仍然可以看到 tuple 1，看不到 tuple 2。

在 `transaction 1` 提交后，再次查看该表，可得：

lp	t_xmin	t_xmax	t_ctid	t_infomask	id	name
1	71986(c)	71989(c)	(0,2)	1282	1	a
2	71989(c)	0(a)	(0,2)	10498	1	b

此时各个 tuple 的 `t_infomask` 被更新，使得编号为 1 的 tuple 不再可见，编号为 2 的 tuple 可见。此时编号为 1 的 tuple 已永远不会被访问，成为了 dead tuple。

删除记录的方式与修改记录的方式类似，只是不会创建新的 tuple，而是直接将 `xmax` 设置为当前 `transaction` 的 `xid`，并将 `t_infomask` 设置为 1282，表示该记录已经被删除。例如以下代码：

```sql
CREATE TABLE transaction_test
(
    id   INT PRIMARY KEY,
    name VARCHAR(20)
);

INSERT INTO transaction_test
VALUES (1, 'a');

DELETE
FROM transaction_test
WHERE id = 1;

SELECT lp, t_xmin, t_xmax, t_ctid, t_infomask
FROM heap_page_items(get_raw_page('transaction_test', 0));
```

其运行结果为：

lp	t_xmin	t_xmax	t_ctid	t_infomask	id	name
1	72004(c)	72004(c)	(0,1)	1282	1	a

可见，由于在删除和修改记录时，都会创建新的版本，所以表的大小会不断增加。对应的，表上的索引因为也需要支持 MVCC，被删除的记录也会被加入到索引中，其大小也会不断增加。

说了这么多 MVCC 的实现原理，现在回到我们的问题：为什么不选择历史表的方式来实现软删除？因为 **Bloat 膨胀**问题在还设计有历史表的情况下只会更加严重，被删除的 tuple 不仅会占用原表的空间，还会占用历史表的空间。更重要的是，索引中也会存在大量的被删除的记录，导致索引的大小不断增加的同时，查询效率也会不断降低。曾有过通过 `VACUUM` 命令删除了超过 70GB 的 dead tuple 和 20GB 的无效索引的例子[3]，如果在该设计中还加入了历史表，那么就会导致额外的 90GB 的空间被占用。

如果仅仅是占用空间过大的问题，那么 `VACUUM` 命令不可以解决吗？我认为，`VACUUM` 操作同样存在一些问题使其不符合视频公司的需求：

- `VACUUM` 命令会删除 dead tuple，但是由于删除的位置是随机的，有效信息将会被分散到磁盘上的各个地方，导致磁盘碎片化，进而影响查询效率
- `VACUUM FULL` 命令会真正释放磁盘空间并重新排列数据，但极为耗时且会阻塞读写操作，并不适合视频公司这种需要实时更新的场景

综上，我放弃了通过创建历史表的方式来实现软删除的想法。

2.1.2 方法二：标记删除

在放弃了历史表的方式后，我开始探索另一种方式：在每个表中加入一个新字段，用于标记该条记录是否被删除。通过以下方法可以快速将一般的数据库设计改为软删除的设计：

- 在每个表中加入一个新字段 `is_deleted`，用于标记该条记录是否被删除

[3]Haki Benita, The Unexpected Find That Freed 20GB of Unused Index Space, `https://hakibenita.com/postgresql-unused-index-size`

- 为每一个表创建一个 `view`，只显示 `is_deleted` 为 `false` 的记录，并将原来的 sql 语句中的表名替换为 `view` 名（如果服务已经发布，则应反向操作，将 `view` 的名字改为原来的表名）
- 在执行删除操作时，手动级联更新所有相关的表中的 `is_deleted` 字段（此处需要将原来的 delete 语句改为 update 语句）

在实践中，标记字段也存在一些变种。如将标记字段改为 `timestamp` 类型，用于记录删除的时间，或者将标记字段改为 `smallint` 类型，可以存储不同的删除状态或者不同等级的可见性。

前面所提到的改动都相对简单。在这种设计中，所遇到的最大挑战来自于主键和索引的设计：如何设计主键和索引，使得查询效率最高？

问题一：唯一性约束如何实现？

主键和其他 unique 约束的唯一性应该只对未被删除的记录进行约束，此时原有的设计已经不满足要求。假设我们设计了一个不允许重复姓名存在的用户表，并只是简单的在表中增加了 `is_deleted` 字段。很明显，这样的设计是不合理的：姓名为 Alice 的用户被删除后，仍然不能再创建一个姓名为 Alice 的用户。

一开始我想到的解决方案是：将 `is_deleted` 字段也加入到 unique 约束中，即：

```
CREATE TABLE exp
(
    id         INT,
    name       VARCHAR(20),
    is_deleted BOOLEAN DEFAULT FALSE,
    PRIMARY KEY (id),
    UNIQUE (name, is_deleted)
);
```

但是这样的设计也存在问题：

- 如果已经有一个姓名为 Alice 的用户被删除了，此时可以创建一个姓名为 Alice 的用户，但是却不能删除这个新创建的用户。
- **unique 约束对应的索引包含了所有的记录，包括已经被删除的记录（完全相同的 Bloat 膨胀问题）**

问题二：查询效率如何保证？

在只需要检索一条记录时，原有的主键索引似乎完全可以满足要求。借用上面的例子，如果我们需要查询 id 为 1 的用户的姓名，使用 `EXPLAIN` 给出的执行计划如下表示，PostgreSQL 会先使用主键索引找到 id 为 1 的记录，然后再检查该记录的 `is_deleted` 字段是否为 `false`，最后返回 `name` 字段。

但是，如果检索结果有多条记录，那么查询效率就会大大降低：比如在本次设计中，有一个 `get_user_info` 的方法，需要返回某个用户观看的所有视频的 bv 号。如果我们不改动原有设计，即将 `user_watch_video` 表中的用户编号 `mid` 和视频编号 `bv` 作为复合主键，那么在查询某个用户观看的所有视频时，PostgreSQL 需要通过主键先找到该用户的所有观看记录，再对每条记录进行一次检查，判断该记录是否被删除，最后返回所有未被删除的记录。

```
EXPLAIN
```

```
SELECT bv
FROM user_watch_video
WHERE mid = 1 AND is_deleted = FALSE;

QUERY PLAN
Bitmap Heap Scan on user_watch_video (cost=4.60..92.76 rows=23 width=13)
    Recheck Cond: (mid = 1)
    Filter: (NOT is_deleted)
    -> Bitmap Index Scan on user_watch_video_pk (cost=0.00..4.60 rows=23 width=0)
        Index Cond: (mid = 1)
```

很明显，在已经删除的记录较多的情况下，这样的查询效率是很低的。更重要的是，**这么做的效率一定低于原先硬删除的设计**，因为硬删除的设计中，表中的记录都是未被删除的，不需要进行额外的检查。

经过搜索，我发现了一个解决方案：**部分索引 (Partial Index)**。

部分索引是指只对满足某一条件的记录建立索引，不满足条件的记录不会被加入到索引中。如果对 `is_deleted` 为 `false` 的数据建立部分索引，则可以完全排除掉已经被删除的记录，既减小了索引的大小，又可以实现只对未被删除的记录建立 unique 约束。例如在本次设计中，`user_watch_video` 表中的索引即为部分索引：

```
CREATE INDEX idx_user_watch_video_mid_bv
ON user_watch_video (mid, bv)
WHERE is_deleted = FALSE;
```

如果此时再次查询某个用户观看的所有视频，PostgreSQL 会直接使用部分索引，而不是先使用主键索引再进行检查：

```
EXPLAIN
SELECT bv
FROM user_watch_video
WHERE mid = 1;

QUERY PLAN
Index Only Scan using idx_user_watch_video_mid_bv on user_watch_video (cost=0.42..4.83
    rows=23 width=13)
    Index Cond: (mid = 1)
```

可以从两次查询的执行计划的 `cost` 字段看出，使用部分索引的查询效率是原来的二十倍以上，而实际测试时更能达到数百倍！在设置和不设置部分索引的情况下，分别执行不同次数的 `get_user_info` 方法，得到的结果如下：

记录数量	不使用部分索引（单位：毫秒）	使用部分索引（单位：毫秒）	提升效率
100	18465	199	97 倍
1000	198102	363	545 倍
10000	1948565	2206	883 倍

此时，我们已经得到了很好的标记删除解决方案。

额外探索：索引

在使用部分索引的前提下，我发现了一些奇怪的现象：PostgreSQL 在使用了部分索引进行 `Bitmap Index Scan` 的前提下，仍然执行了 recheck 操作，而且 recheck 操作的代价很大。例如，在一次 update 操作中，使用 `EXPLAIN ANALYZE` 查看执行计划如下：

```
EXPLAIN ANALYZE
UPDATE user_watch_video
SET is_deleted = true
WHERE mid = 75
  AND is_deleted = false;

QUERY PLAN
Update on user_watch_video (cost=4.60..92.76 rows=0 width=0) (actual time=2.635..2.636
    rows=0 loops=1)
->  Bitmap Heap Scan on user_watch_video (cost=4.60..92.76 rows=23 width=7) (actual
    time=0.046..0.320 rows=26 loops=1)
    Recheck Cond: ((mid = 75) AND (NOT is_deleted))
    Filter: (NOT is_deleted)
    Heap Blocks: exact=26
    ->  Bitmap Index Scan on idx_user_watch_video_mid_bv (cost=0.00..4.60 rows=23
        width=0) (actual time=0.023..0.023 rows=26 loops=1)
        Index Cond: (mid = 75)
```

在这个例子中，我希望删除用户 75 的所有观看记录（在软删除中转化为 update 操作）。可以看到，PostgreSQL 首先使用了部分索引 `idx_user_watch_video_mid_bv`，定位到了用户 75 的所有未被删除的观看记录。按理来说，此时索引返回的记录中 `is_deleted` 一定为 `false`，但是 PostgreSQL 仍然在上一级的节点中进行了 recheck 操作，再次检查 `is_deleted` 字段是否为 `false`。并且 recheck 操作的代价很大，占用了绝大部分的执行时间（预计 cost 中占 95%，实际时间中占 85%）

我产生了两个疑问：

1. 为什么有可以直接使用的索引，却不执行 `Index Only Scan`，而是执行 `Bitmap Index Scan` 和 `Bitmap Heap Scan`?
2. 在部分索引保证了条件满足的情况下，为什么仍然需要进行 recheck 操作?

问题一：为什么不执行 `Index Only Scan`？

由于这个问题和读写性能有关，需要回到数据库底层的存储结构上来解释。

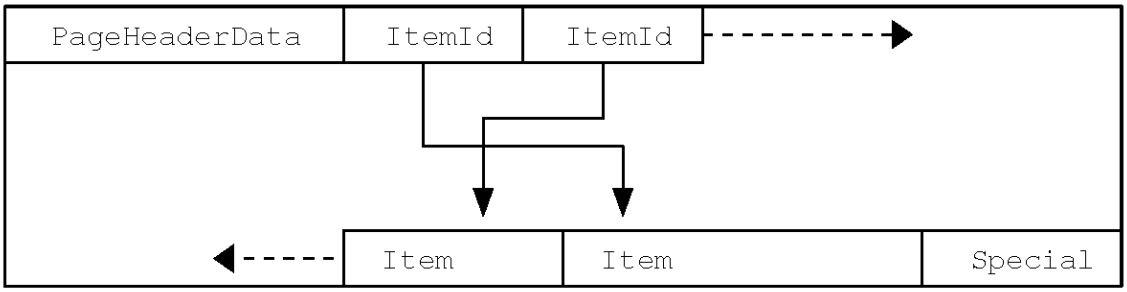

图 4: Page 结构图[4]

在 PostgreSQL 中，每个表都被分为多个固定大小为 8KB 的 Page，每个 Page 中包含了多个 tuple（即图中的 item）。每个 tuple 被分为 header 和数据两部分，header 中存有一个指针指向数据。索引中存放的是指向 tuple 的指针。

此处需要介绍另外一个概念：correlation，即相关性。索引采用的是 B+ 树的结构，但是在树中相邻的两个节点在磁盘上并不一定相邻。相关性反应了数据在树上的顺序和在磁盘上的顺序之间的差异，其值在-1 到 1 之间，值越大则顺序越接近，具体值可以从 `pg_stats` 视图中查看。

为何需要引入相关性这一概念呢？因为机械硬盘的顺序读写速度远大于随机读写速度，一般来说可以达到 100 倍以上（随机读写时机械硬盘需要反复移动磁头）。使用 `Index Only Scan` 时，PostgreSQL 会直接按照索引给出的顺序依次读取 tuple。如果相关性很大，则按照索引给出的 tuple 顺序进行读取几乎就是顺序读取，速度很快。

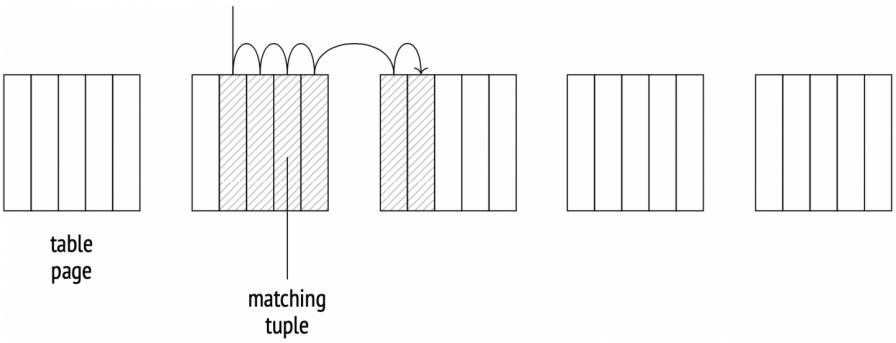

图 5: 顺序读取[5]

但是当相关性很小时，索引给出的 tuple 顺序和磁盘上的顺序差异很大，此时几乎变为了随机读取：

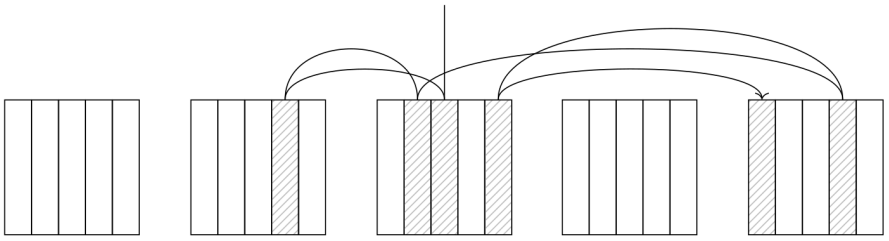

图 6: 随机读取

[4]PostgreSQL Documentation, *storage-page-layout*, `https://www.postgresql.org/docs/current/storage-page-layout.html`

[5]Habr, *Queries in PostgreSQL. Index scan*, `https://habr.com/en/companies/postgrespro/articles/666974/`

如果相关性过小，PostgreSQL 会放弃使用 `Index Only Scan`，而是使用 `Bitmap Index Scan` 和 `Bitmap Heap Scan`。其原理为：首先将所有的 tuple 指针所在的 page 读取到内存中，然后生成一个 `Bitmap`，其中每个 bit 代表相应该位置的 tuple 是否满足条件。最后，根据 `Bitmap` 中的信息，从内存中读取相应的 tuple。这样做的好处是：由于 page 本身是连续存储的 8KB 数据，读取单独的 page 时可以使用顺序读取。并且生成 `Bitmap` 的过程相当于对 tuple 进行了一次排序，使得读取 tuple 时的顺序更加接近顺序读取。

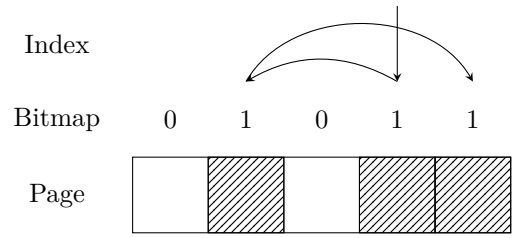

图 7: Bitmap 示意图

附：PostgreSQL 在估算随机读取和顺序读取的代价时，使用的参数 `random_page_cost` 和 `seq_page_cost` 分别为 4 和 1。此数值可以由用户自行修改，例如在随机读写和顺序读写性能差异不大的固态硬盘上，可以将其修改为相同的值。

问题二：为什么需要进行 recheck 操作？

由问题一可知，进行 `Bitmap Index Scan` 需要将目标 tuple 所在的 page 读取到内存中。当 page 数量过多时，内存开销会很大。如果 PostgreSQL 预估使用内存会超过 `work_mem` 的值（默认为 4 MB），将转为 lossy 模式。在此模式下，原先 `Bitmap` 中每个 bit 对应一个 tuple，现在每个 bit 对应一个 page，即改为找出所有包含满足条件的 tuple 的 page。这样做的好处是：内存开销大大减小，但是获得的 page 中可能存在一些不满足条件的 tuple，需要进行 recheck 操作。

是否执行 lossy 模式可以通过 `EXPLAIN ANALYZE` 得知，如前文所提到的例子中有一行：`Heap Blocks: exact=26`。这表示有 26 个 page 是直接对每个 tuple 生成 `Bitmap`。如果执行了 lossy 模式，将会显示为：`exact = ..., lossy = ...`。即使没有执行 lossy 模式，仍然会在执行计划中显示 recheck 操作，但实际运行时不会进行。

2.2 数据库性能优化

在此次的项目中，针对不同类型的方法，我采用了不同的优化策略。但其核心目标可以被总结为：

- 减少 Java 端与数据库连接的次数
- 减少 Java 端与数据库传输的数据量

2.2.1 数据导入

数据导入部分，我采用的策略有：批处理，多线程插入，多值插入，以及删除与重建索引。

批处理与多值插入

使用批处理功能前，Java 端每发送一条 SQL 语句，都需要等待数据库返回结果，之后才能发送下一条 SQL 语句。而在批处理中，Java 端可以一次性发送多条 SQL 语句，并一次性接收多条 SQL 语句的结果，减少了中间的等待时间。更重要的是，由于使用了 `PreparedStatement`，在同一个 `Batch` 中相同的 SQL 语句只会被编译一次，大大减少了编译的时间。

多值插入则指的是将原来的多条插入语句合并为一条，例如：

```
INSERT INTO user_exp
VALUES (1, 'a'), (2, 'b'), (3, 'c');
```

这样的改写是数据库的优化器自动完成的，只需要在数据库连接的 url 中加入 `rewriteBatchedStatements=true` 即可。

多线程插入

数据库同时可以执行多个事务，所以我们可以利用多线程同时向数据库中插入数据。在本次设计中，了解到服务器的 CPU 核心数为 4，故我使用了 4 个线程，并将每个关系表的数据分为 4 份，由 4 个线程分别插入。测试后得到的结果如下：

单线程插入（单位：毫秒）	多线程插入（单位：毫秒）	提升效率
44808	32755	1.37 倍

删除与重建索引

在导入数据时，相比于在每次插入时都更新索引，直接对完整的表进行一次建立索引的操作更加高效。因此，我采取了先删除索引，再导入数据，最后重建索引的策略。在数据量较大时，这种策略的效率提升非常明显，例如我使用提供的大数据集进行测试时，得到的结果如下：

不删除索引（单位：毫秒）	删除并重建索引（单位：毫秒）	提升效率
446028	270521	1.65 倍

2.2.2 数据查询

数据查询部分，我采用的策略有：将方法封装在数据库端、使用 `PreparedStatement` 和有效利用索引。

将方法封装在数据库端

将方法封装在数据库的 `function` 中，即原来数据库执行 `insert`、`update`、`delete` 等操作后返回结果给 Java 端，逻辑判断在 Java 端完成，现在将逻辑判断的部分一同封装在数据库端，Java 端只需要调用 `function` 即可。这种方法可以很明显的减少连接次数和传输数据量。例如在一次 `searchVideo` 的查询中，如果在 Java 端实现，步骤如下：

- 建立连接 1：数据库找到用户编号对应的用户信息
- Java 端判断密码、qq 等信息是否正确
- 如果身份验证成功，建立连接 2：数据库查询该用户的用户属性
- Java 端根据用户属性，拼接 SQL 查询语句
- 建立连接 3：数据库搜索视频

这样的操作不仅需要建立 3 次连接，并且由于搜索视频的 SQL 语句较长，传输的数据量也较大。如果改为在数据库端封装方法，则 java 端只需要建立一次连接，传输的内容也仅限于用户身份信息、搜索关键词等：

- 建立连接 1：将用户身份信息、搜索关键词等传入数据库端的 `searchVideo function` 中，直接返回结果

这样做的一个好处是：执行 `function` 后，PostgreSQL 会缓存该 `function` 的执行计划，下次再次调用时，不需要再次编译，直接使用缓存中的执行计划，从而提高了查询效率。

另外一个好处是：不同函数间的相互调用更加方便。在 Java 中，不同类别的服务的实现代码被分散在不同的类中，如 `UserServiceImpl` 和 `VideoServiceImpl`。这会导致部分通用的方法需要在不同的类中重复实现，例如查验用户身份信息是否合法的方法。而将这些方法封装在数据库端，可以在不同的函数中直接调用，避免了重复实现。

以下是两种策略的的查询效率（std 程序耗时除以我的程序耗时）对比：

将方法封装在 Java 端的效率	将方法封装在数据库端的效率	提升效率
0.39	0.61	1.56 倍

PreparedStatement

使用 `PreparedStatement` 不仅可以防止 SQL 注入,还可以减少编译的时间。其原理与上文中 `function` 的缓存机制类似，即通过缓存执行计划来提高查询效率。可以从 `pg_catalog.pg_prepared_statements` 中查看缓存的执行计划。与 `function` 的缓存不同的是，每个数据库连接的 `PreparedStatement` 缓存是独立的，并且总缓存数目有限，PostgreSQL 提供了 `max_prepared_transactions` 参数来修改。而 `function` 的缓存有其单独的内存空间，且允许跨连接共享。

有效利用索引

在软删除的设计中，我使用了部分索引，即只对未被删除的记录建立索引，这部分已在前文中介绍。在本次设计中，我还使用了覆盖索引，具体如下：

覆盖索引 (Covering Index) 指的是将部分字段的值加入到索引中，从而不需要实际读取表中的记录，即可返回查询结果。在阿里巴巴的 Java 开发手册中，有一段生动的比喻：

> 如果一本书需要知道第 11 章是什么标题，会翻开第 11 章对应的那一页吗？目录浏览一下就好，这个目录就是起到覆盖索引的作用。

在 PostgreSQL 中，部分索引可以和覆盖索引合并使用。例如，假设经常需要查找可见用户中某个 id 对应用户的身份信息，可以将索引设计如下：

```
CREATE INDEX idx_user_mid_identity
ON user (mid)
INCLUDE (identity)
WHERE is_deleted = FALSE;
```

此外，我还考虑了一些其他的优化策略和规范

- 将使用率更高的字段置于复合索引的前面
- 所有应该具有唯一性的字段都建立了 unique 索引

2.3 数据库安全性优化

2.3.1 字段加密

为了用户的隐私安全，我希望做到以下几点：

- 用户的密码不应该以明文的形式存储在数据库中，避免被数据库管理员窃取
- 用户的微信，qq 等信息也需要加密，防止数据库被攻破后用户信息泄露

PostgreSQL 提供了 pgcrypto 扩展，可以用于加密和解密数据。与 SA 交流得知，由于目前服务器上的 pgcrypto 出现了版本兼容性问题，暂时无法使用该扩展，故我并未在本次设计中使用该扩展。下面是一个简单的加密解密示例：

```
CREATE TABLE crypto_password(
    id       INT NOT NULL,
    password VARCHAR(255) NOT NULL,
    PRIMARY KEY (id)
);

INSERT INTO crypto_password
VALUES (1, crypt('my password', gen_salt('bf')));
```

此时再查看 crypto_password 表，可以看到密码已经被加密：

id	password
1	$2a$06$U5efxgzPqiEEt/qEjhoFSugeFQUS.4pHzOg/NKibdIZOmZgn9NNfW

若想要验证密码是否正确，可以再次使用 crypt 函数：

```
SELECT password = crypt('my password', password)
FROM crypto_password
WHERE id = 1;
```

结果为 true，说明密码正确。

2.3.2 防止内部逻辑泄露与防止 SQL 注入

在本次设计中，我将几乎所有的 api 都封装在了数据库的 function 中，Java 端代码只使用 PreparedStatement 执行函数调用。这样即使 Java 端代码被泄露，也无法得知数据库内部表的结构和函数的实现细节，从而保证了数据库的安全性。

此外，我还探索了在数据库端防止 SQL 注入的方法，即配合动态 SQL 使用 EXECUTE 和 USING 语句：动态 SQL，即先以字符串拼接的形式构造 SQL 语句，再执行，通常用于查询条件不确定的情况。由于需要将用户输入的字符串拼接到 SQL 语句中，这种方式很容易受到 SQL 注入攻击。此时可以使用 EXECUTE 和 USING 语句，将用户输入的字符串作为参数传入，而不是直接拼接到 SQL 语句中。例如：

```
CREATE OR REPLACE FUNCTION test_execute(name TEXT)
RETURNS TABLE (id INT, name TEXT)
AS $$
BEGIN
    RETURN QUERY EXECUTE 'SELECT id, name FROM test WHERE name = $1' USING name;
END;
$$ LANGUAGE plpgsql;
```

对于输入参数个数可变的情况，动态 SQL 也是很好的解决方案，只需要将 USING 后的参数改为数组形式即可。例如实现一个不定个数的整数加法：

```
CREATE OR REPLACE FUNCTION test_execute(numbers INT[])
RETURNS INT
AS $$
DECLARE
    dynamic_query TEXT := '';
    result        INT;
BEGIN
    FOR i IN 1..array_length(numbers, 1)
        LOOP
            dynamic_query := dynamic_query || '$1[' || i || '] + ';
        END LOOP;
    dynamic_query := 'SELECT ' || substring(dynamic_query, 1, length(dynamic_query) - 3);
    EXECUTE dynamic_query USING numbers INTO result;
    RETURN result;
END
$$ LANGUAGE plpgsql;
```

对于一个长度为 3 的输入数组，`dynamic_query` 被拼接为为 `SELECT $1[1] + $1[2] + $1[3]`，最后将 `numbers` 数组传入，即可得到结果。

需要说明的是，由于动态 SQL 的不确定性，PostgreSQL 无法缓存其执行计划，每次执行时都需要重新编译。

2.4 数据库其他优化与设计

2.4.1 连接池

在本次设计中，由于 SA 提供的模板中已经使用了 HikariCP 连接池，故我并未对使用连接池进行额外的尝试。连接池主要的作用是重复利用已经建立的数据库连接，以减少每次建立连接的开销。并且可以限制连接的数量，防止由于超过数据库连接数上限而导致新的连接无法建立的情况。

2.4.2 使用合适的数据类型减小数据大小

在 `user` 表中，用户的性别和身份信息直接使用单个字符表示以减少数据大小。每个表的 `is_deleted` 字段使用布尔型表示。此处没有使用 `ENUM` 类型，因为 PostgreSQL 中 `ENUM` 类型的数据一旦被创建，就无法添加或删除其中的类别，极其不利于后续的维护。

2.4.3 额外探索：UUID 与雪花算法

在本次设计中，我们需要对新注册的用户和视频生成唯一的编号。在一般的情况下，我们可以使用自增的整数作为编号，但是对于大型公司分库分表的情况，有可能造成不同表之间编号重复的情况。对于这个问题，比较经典的解决方案是使用 UUID 和雪花算法。

PostgreSQL 提供了内置的 `uuid` 类型，可以用于存储 UUID。UUID 虽然能够保证唯一性，但是其内容没有规律，不利于人类阅读或记忆。在此项目中，UUID 由于包含 `-` 字符，违反了视频编号只能包含数字和字母的要求，也违反了用户编号为整数类型的要求，故没有使用。

雪花算法可以生产一个 64 位的整数，其中包含了时间戳、机器编号等信息，利于人类直接阅读，也利于直接分析出数据的分布情况。但是 PostgreSQL 中没有内置的雪花算法，故本次设计中没有使用。