基于 PostgreSQL 的视频网站设计及性能分析

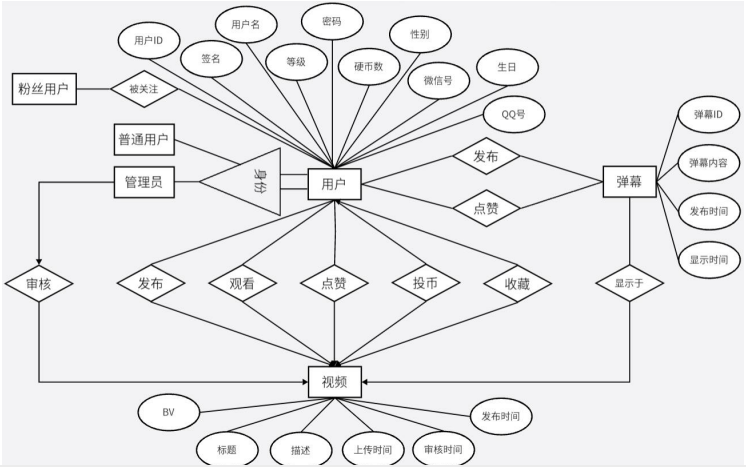
数据库原理（H）课程项目 2 答辩

周圣力 12212232

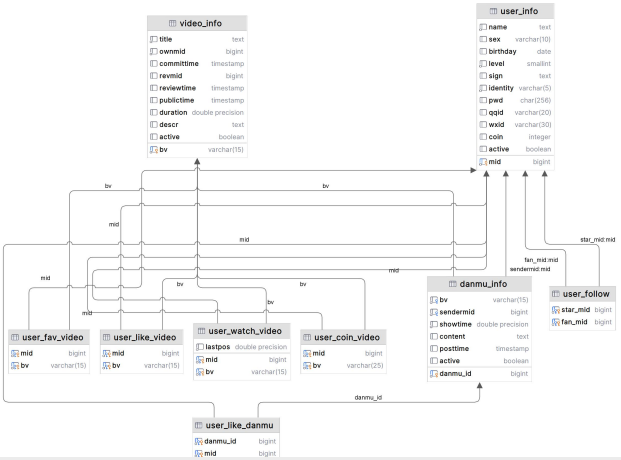
Jan. 5, 2024

章节目录

E-R 图


用户ID
用户名
密码
性别
签名
等级
硬币数
微信号
生日
QQ号
粉丝用户
被关注
普通用户
管理员
身份
用户
发布
点赞
弹幕
弹幕ID
弹幕内容
发布时间
显示时间
审核
发布
观看
点赞
投币
收藏
显示于
视频
BV
发布时间
标题
描述
上传时间
审核时间

数据表设计


video_info
title text
ownmid bigint
committime timestamp
revmid bigint
reviewtime timestamp
publictime timestamp
duration double precision
descr text
active boolean
bv varchar(15)
user_info
name text
sex varchar(10)
birthday date
level smallint
sign text
identity varchar(5)
pwd char(256)
qqid varchar(20)
wxid varchar(30)
coin integer
active boolean
mid bigint
bv
mid
star_mid:mid
fan_mid:mid
sendermid:mid
danmu_info
bv varchar(15)
sendermid bigint
showtime double precision
content text
posttime timestamp
active boolean
danmu_id bigint
user_follow
star_mid bigint
fan_mid bigint
user_fav_video
mid bigint
bv varchar(15)
user_like_video
mid bigint
bv varchar(15)
user_watch_video
lastpos double precision
mid bigint
bv varchar(15)
user_coin_video
mid bigint
bv varchar(25)
danmu_id
user_like_danmu
danmu_id bigint
mid bigint

Foreign Keys: To Add or Not To Add?

外键的加入会直接拖慢所有修改操作的速度。

Foreign Keys: To Add or Not To Add?

外键的加入会直接拖慢所有修改操作的速度。

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。

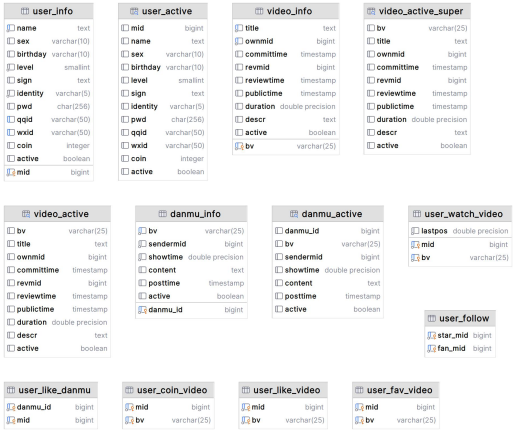
Foreign Keys: To Add or Not To Add?

外键的加入会直接拖慢所有修改操作的速度。

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。

《阿里巴巴 Java 开发手册》：所有外键功能需要在应用层解决。

数据表设计


user_info
name text
sex varchar(10)
birthday varchar(10)
level smallint
sign text
identity varchar(5)
pwd char(256)
qqid varchar(50)
wxid varchar(50)
coin integer
active boolean
mid bigint
user_active
mid bigint
name text
sex varchar(10)
birthday varchar(10)
level smallint
sign text
identity varchar(5)
pwd char(256)
qqid varchar(50)
wxid varchar(50)
coin integer
active boolean
video_info
title text
ownmid bigint
committime timestamp
revmid bigint
reviewtime timestamp
publictime timestamp
duration double precision
descr text
active boolean
bv varchar(25)
video_active_super
bv varchar(25)
title text
ownmid bigint
committime timestamp
revmid bigint
reviewtime timestamp
publictime timestamp
duration double precision
descr text
active boolean
video_active
bv varchar(25)
title text
ownmid bigint
committime timestamp
revmid bigint
reviewtime timestamp
publictime timestamp
duration double precision
descr text
active boolean
danmu_info
bv varchar(25)
sendermid bigint
showtime double precision
content text
posttime timestamp
active boolean
danmu_id bigint
danmu_active
danmu_id bigint
bv varchar(25)
sendermid bigint
showtime double precision
content text
posttime timestamp
active boolean
user_watch_video
lastpos double precision
mid bigint
bv varchar(25)
user_follow
star_mid bigint
fan_mid bigint
user_like_danmu
danmu_id bigint
mid bigint
user_coin_video
mid bigint
bv varchar(25)
user_like_video
mid bigint
bv varchar(25)
user_fav_video
mid bigint
bv varchar(25)

字段索引

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。

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。

字段索引

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。

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。

运行效率：0.4 → 0.7。

章节目录

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**。

```
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
-- ...
```

分批插入

Auto commit: 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。

分批插入

Auto commit: 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。

DataGrip：每 1000 条自动打包，一起执行。

分批插入

Auto commit: 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。

DataGrip：每 1000 条自动打包，一起执行。

我也这么干！

分批插入

Auto commit: 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。

DataGrip：每 1000 条自动打包，一起执行。

我也这么干！然后打好的包被 JDBC 的 auto commit 又拆成一条一条执行了。

分批插入

Auto commit: 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。

DataGrip：每 1000 条自动打包，一起执行。

我也这么干！然后打好的包被 JDBC 的 auto commit 又拆成一条一条执行了。

```
conn.setAutoCommit(false);
// add insert commands into a batch
conn.commit();
```

多线程插入

插入时没有外键约束 → 可以多线程同时插入。

多线程插入

插入时没有外键约束 → 可以多线程同时插入。

如何分配每个线程的任务？小表不拆，大表少拆（减少多个线程同时访问一块内存区域的情况）；每个线程任务量尽量平均。

多线程插入

插入时没有外键约束 → 可以多线程同时插入。

如何分配每个线程的任务？小表不拆，大表少拆（减少多个线程同时访问一块内存区域的情况）；每个线程任务量尽量平均。

测试结果：

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

几个线程比较合理?

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：

几个线程比较合理?

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。

几个线程比较合理?

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。

几个线程比较合理?

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**。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。

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。

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。

如何加索引?

方案 1：对所有的记录一视同仁，全部加上同样的索引。
方案 2：使用局部索引（partial index），只对有效的记录加索引。
方案 3：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。

如何加索引?

方案 1：对所有的记录一视同仁，全部加上同样的索引。
方案 2：使用局部索引（partial index），只对有效的记录加索引。
方案 3：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。（×）

如何加索引？

方案 1：对所有的记录一视同仁，全部加上同样的索引。

方案 2：使用局部索引（partial index），只对有效的记录加索引。

方案 3：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。（×）

针对局部索引，官方文档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。

防止视频查找时的注入攻击

视频查找方法是所有方法中唯一一个需要单独处理用户传入的字符串的方法，所以也有较大概率受到用户的注入攻击。对此，我主要分析并处理了以下两种情况：

第一种情况是 **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。

第二种情况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。

章节目录

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。

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。

章节目录

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。

“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**。

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。

实验数据

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

但是……实际上用不了

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**。

章节目录

B 站是怎么做的?

实际场景：大数据、高并发。

B 站是怎么做的?

实际场景：大数据、高并发。
2018 年之前的 B 站：使用 MySQL，需要频繁停机扩容，容易崩。

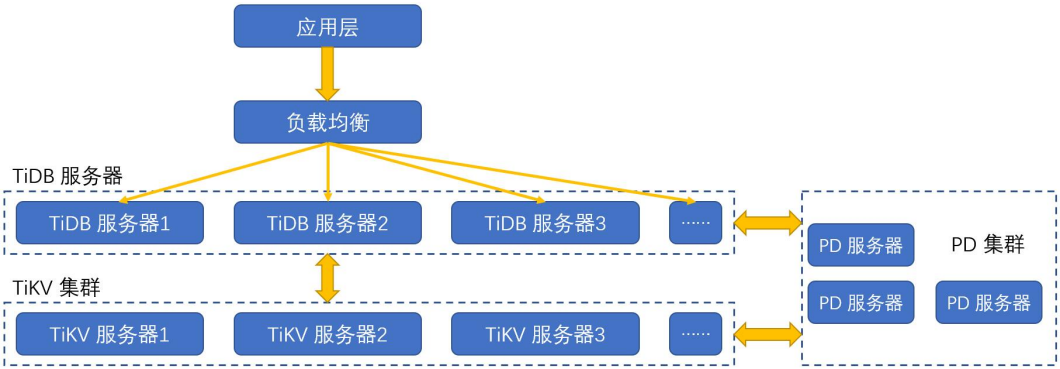
B 站是怎么做的?

实际场景：大数据、高并发。
2018 年之前的 B 站：使用 MySQL，需要频繁停机扩容，容易崩。
2018 年的 B 站：访问越来越多，一直停机扩容不是办法……

B 站是怎么做的?

实际场景：大数据、高并发。
2018 年之前的 B 站：使用 MySQL，需要频繁停机扩容，容易崩。
2018 年的 B 站：访问越来越多，一直停机扩容不是办法……换一个数据库！

TiDB 的结构


应用层
负载均衡
TiDB 服务器
TiDB 服务器1
TiDB 服务器2
TiDB 服务器3
……
TiKV 集群
TiKV 服务器1
TiKV 服务器2
TiKV 服务器3
……
PD 服务器
PD 集群
PD 服务器
PD 服务器

TiDB 的优点

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。

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。

章节目录

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/

谢谢观看！

GitHub 项目地址：https://github.com/fz-zsl/Nilikili