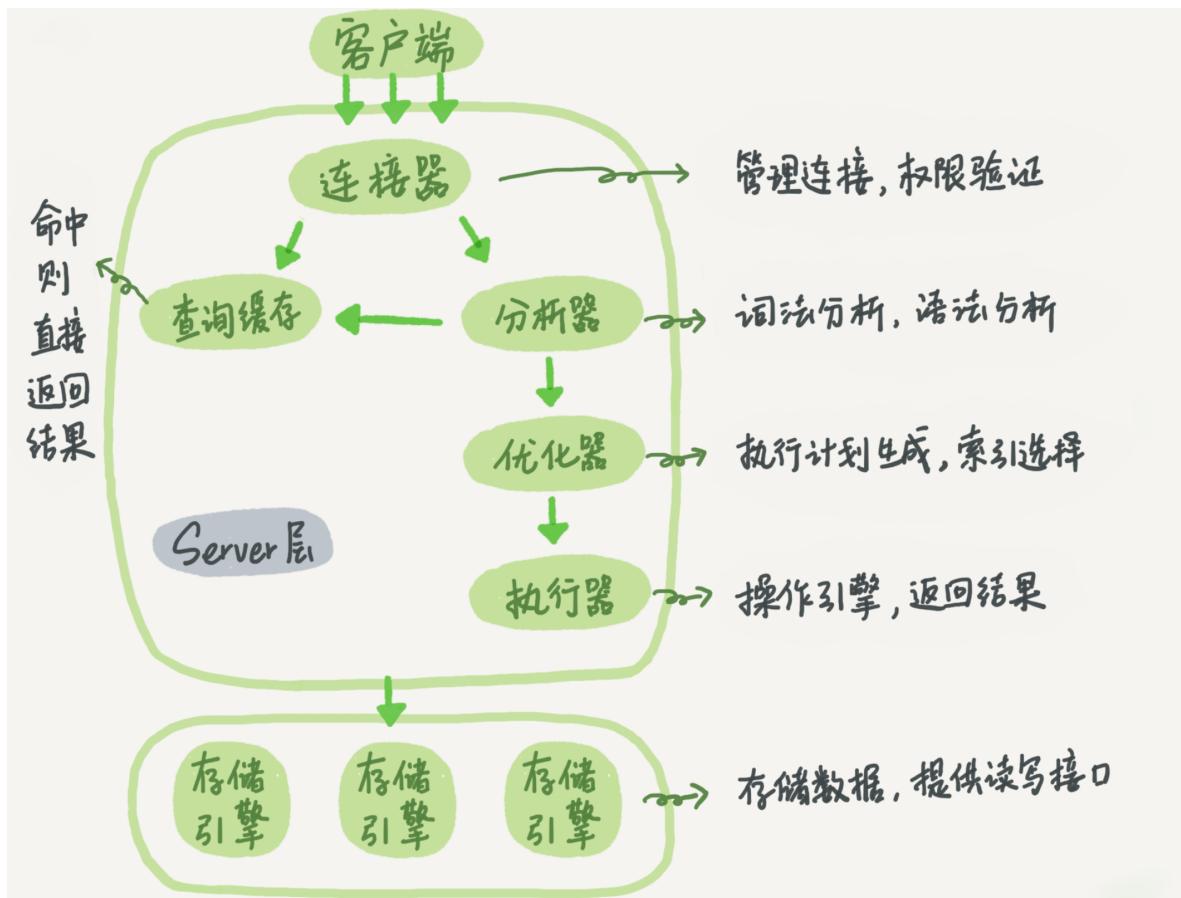


# MySQL

## 01 | 基础架构：一条SQL查询语句是如何执行的？

MySQL 的基础架构

```
select * from T where ID=10;
```



### 连接器

```
mysql -h$ip -P$port -u$user -p
```

```
mysql> show processlist;
+----+----+-----+----+-----+----+----+-----+
| Id | User | Host           | db | Command | Time | State   | Info      |
+----+----+-----+----+-----+----+----+-----+
| 5  | root | localhost:27710 | test | Sleep   |    16 | NULL    | NULL
| 6  | root | localhost:27712 | test | Query   |     0 | starting | show processlist
+----+----+-----+----+-----+----+----+-----+
2 rows in set (0.00 sec)
```

客户端如果太长时间没动静，连接器就会自动将它断开。这个时间是由参数 `wait_timeout` 控制的，默认值是 8 小时

尽量使用长连接 减少建立连接的动作

但是全部使用长连接后，你可能会发现，有些时候 MySQL 占用内存涨得特别快，这是因为 MySQL 在执行过程中临时使用的内存是管理在连接对象里面的。这些资源会在连接断开的时候才释放。所以如果长连接累积下来，可能导致内存占用太大，被系统强行杀掉（OOM）

两种解决方案。

1. 定期断开长连接。使用一段时间，或者程序里面判断执行过一个占用内存的大查询后，断开连接，之后要查询再重连。
2. 如果你用的是 MySQL 5.7 或更新版本，可以在每次执行一个比较大的操作后，通过执行 `mysql_reset_connection` 来重新初始化连接资源。这个过程不需要重连和重新做权限验证，但是会将连接恢复到刚刚创建完时的状态。

## 查询缓存

MySQL 拿到一个查询请求后，会先到查询缓存看看，之前是不是执行过这条语句。之前执行过的语句及其结果可能会以 key-value 对的形式，被直接缓存在内存中。`key` 是查询的语句，`value` 是查询的结果

但是大多数情况下我会建议你不要使用查询缓存，为什么呢？因为查询缓存往往弊大于利

查询缓存的失效非常频繁，只要有对一个表的更新，这个表上所有的查询缓存都会被清空。因此很可能你费劲地把结果存起来，还没使用呢，就被一个更新全清空了。对于更新压力大的数据库来说，查询缓存的命中率会非常低。除非你的业务就是有一张静态表，很长时间才会更新一次。比如，一个系统配置表，那这张表上的查询才适合使用查询缓存

将参数 `query_cache_type` 设置成 `DEMAND`，这样对于默认的 SQL 语句都不使用查询缓存。而对于你确定要使用查询缓存的语句，可以用 `SQL_CACHE` 显式指定

```
mysql> select SQL_CACHE * from T where ID=10
```

MySQL 8.0 版本直接将查询缓存的整块功能删掉了

## 分析器

对 SQL 语句做解析

分析器先会做“词法分析”。你输入的是由多个字符串和空格组成的一条 SQL 语句，MySQL 需要识别出里面的字符串分别是什么，代表什么。

MySQL 从你输入的“select”这个关键字识别出来，这是一个查询语句。它也要把字符串“T”识别成“表名 T”，把字符串“ID”识别成“列 ID”。

做完了这些识别以后，就要做“语法分析”。根据词法分析的结果，语法分析器会根据语法规则，判断你输入的这个 SQL 语句是否满足 MySQL 语法。

如果你的语句不对，就会收到“You have an error in your SQL syntax”的错误提醒，比如下面这个语句 `select` 少打了开头的字母“s”。

```
mysql> elect * from t where ID=1;  
  
ERROR 1064 (42000): You have an error in your SQL syntax; check the manual that  
corresponds to your MySQL server version for the right syntax to use near 'elect  
* from t where ID=1' at line 1
```

## 优化器

优化器是在表里面有多个索引的时候，决定使用哪个索引；或者在一个语句有多表关联（join）的时候，决定各个表的连接顺序

```
mysql> select * from t1 join t2 using(ID) where t1.c=10 and t2.d=20;
```

- 既可以先从表 t1 里面取出 c=10 的记录的 ID 值，再根据 ID 值关联到表 t2，再判断 t2 里面 d 的值是否等于 20。
- 也可以先从表 t2 里面取出 d=20 的记录的 ID 值，再根据 ID 值关联到 t1，再判断 t1 里面 c 的值是否等于 10。

这两种执行方法的逻辑结果是一样的，但是执行的效率会有不同，而优化器的作用就是决定选择使用哪一个方案。

## 执行器

MySQL 通过分析器知道了你要做什么，通过优化器知道了该怎么做，于是就进入了执行器阶段，开始执行语句。

开始执行的时候，要先判断一下你对这个表 T 有没有执行查询的权限，如果没有，就会返回没有权限的错误，如下所示（在工程实现上，如果命中查询缓存，会在查询缓存返回结果的时候，做权限验证。查询也会在优化器之前调用 precheck 验证权限）

```
mysql> select * from T where ID=10;  
ERROR 1142 (42000): SELECT command denied to user 'b'@'localhost' for table 'T'
```

如果有权限，就打开表继续执行。打开表的时候，执行器就会根据表的引擎定义，去使用这个引擎提供的接口。

比如我们这个例子中的表 T 中，ID 字段没有索引，那么执行器的执行流程是这样的：

1. 调用 InnoDB 引擎接口取这个表的第一行，判断 ID 值是不是 10，如果不是则跳过，如果是则将这行存在结果集中；
2. 调用引擎接口取“下一行”，重复相同的判断逻辑，直到取到这个表的最后一行。
3. 执行器将上述遍历过程中所有满足条件的行组成的记录集作为结果集返回给客户端。

至此，这个语句就执行完成了

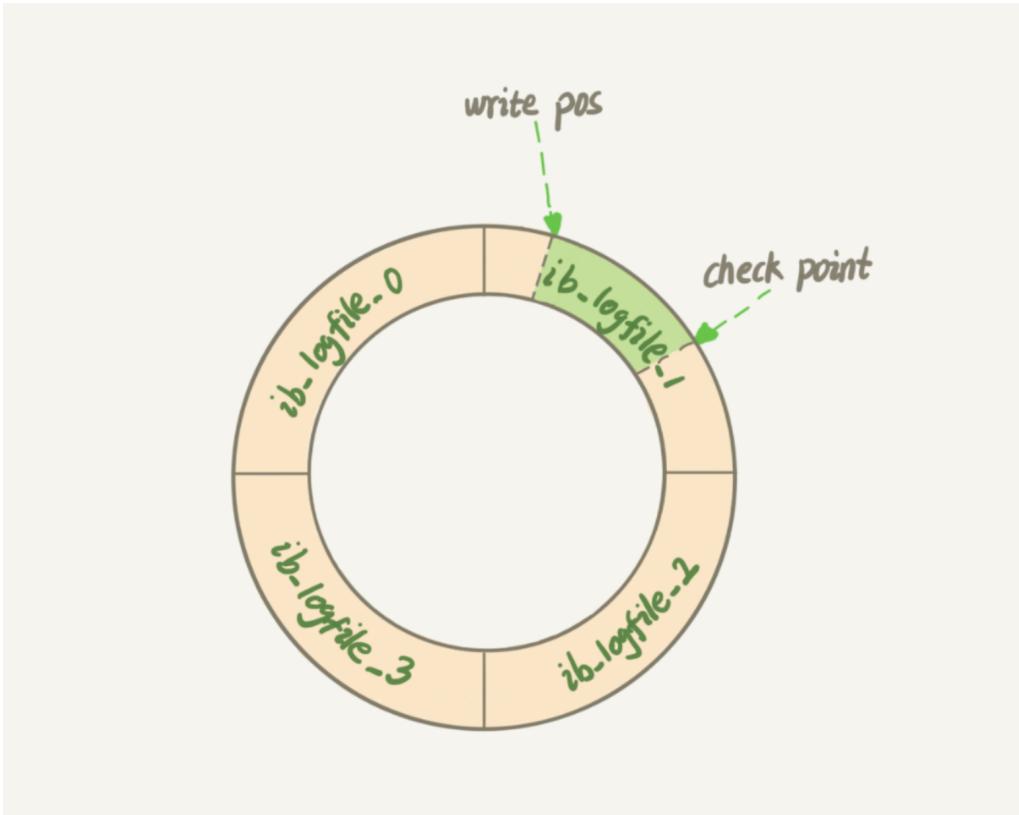
如果表 T 中没有字段 k，而你执行了这个语句 `select * from T where k=1`，那肯定是会报“不存在这个列”的错误：“Unknown column ‘k’ in ‘where clause’”。你觉得这个错误是在我们上面提到的哪个阶段报出来的呢？

## 02 | 日志系统：一条SQL更新语句是如何执行的？

如果每一次的更新操作都需要写进磁盘，然后磁盘也要找到对应的那条记录，然后再更新，整个过程 IO 成本、查找成本都很高

WAL 技术全称是 Write-Ahead Logging，它的关键点就是先写日志，再写磁盘

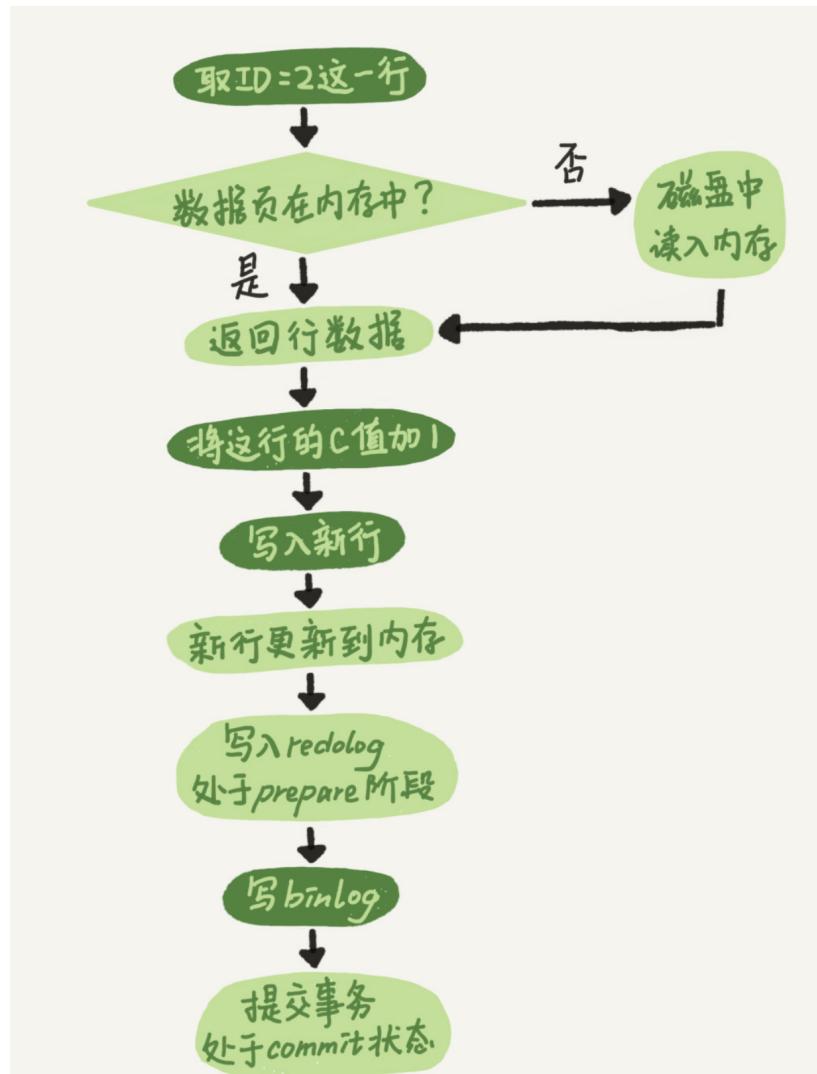
当有一条记录需要更新的时候，InnoDB 引擎就会先把记录写到 redo log 里面，并更新内存，这个时候更新就算完成了。同时，InnoDB 引擎会在适当的时候，将这个操作记录更新到磁盘里面



MySQL 整体来看，其实就有两块：一块是 Server 层，它主要做的是 MySQL 功能层面的事情；还有一块是引擎层，负责存储相关的具体事宜。 redo log 是 InnoDB 引擎特有的日志，而 Server 层也有自己的日志，称为 binlog（归档日志）

这两种日志有以下三点不同。

1. redo log 是 InnoDB 引擎特有的；binlog 是 MySQL 的 Server 层实现的，所有引擎都可以使用。
2. redo log 是物理日志，记录的是“在某个数据页上做了什么修改”；binlog 是逻辑日志，记录的是这个语句的原始逻辑，比如“给 ID=2 这一行的 c 字段加 1”。
3. redo log 是循环写的，空间固定会用完；binlog 是可以追加写入的。“追加写”是指 binlog 文件写到一定大小后会切换到下一个，并不会覆盖以前的日志。



update 语句的执行流程图，图中浅色框表示是在 InnoDB 内部执行的，深色框表示是在执行器中执行的

1. 执行器先找引擎取 ID=2 这一行。ID 是主键，引擎直接用树搜索找到这一行。如果 ID=2 这一行所在的数据页本来就在内存中，就直接返回给执行器；否则，需要先从磁盘读入内存，然后再返回。
2. 执行器拿到引擎给的行数据，把这个值加上 1，比如原来是 N，现在就是 N+1，得到新的一行数据，再调用引擎接口写入这行新数据。
3. 引擎将这行新数据更新到内存中，同时将这个更新操作记录到 redo log 里面，此时 redo log 处于 prepare 状态。然后告知执行器执行完成了，随时可以提交事务。
4. 执行器生成这个操作的 binlog，并把 binlog 写入磁盘。
5. 执行器调用引擎的提交事务接口，引擎把刚刚写入的 redo log 改成提交（commit）状态，更新完成

将 redo log 的写入拆成了两个步骤：prepare 和 commit，这就是“两阶段提交”。

## 两阶段提交

为什么必须有“两阶段提交”呢？这是为了让两份日志之间的逻辑一致。要说明这个问题，我们得从文章开头的那个问题说起：怎样让数据库恢复到半个月内任意一秒的状态？

简单说，redo log 和 binlog 都可以用于表示事务的提交状态，而两阶段提交就是让这两个状态保持逻辑上的一致。

物理日志 redo log 和逻辑日志 binlog。

`redo log` 用于保证 `crash-safe` 能力。`innodb_flush_log_at_trx_commit` 这个参数设置成 1 的时候，表示每次事务的 `redo log` 都直接持久化到磁盘。这个参数我建议你设置成 1，这样可以保证 MySQL 异常重启之后数据不丢失。

`sync_binlog` 这个参数设置成 1 的时候，表示每次事务的 `binlog` 都持久化到磁盘。这个参数我也建议你设置成 1，这样可以保证 MySQL 异常重启之后 `binlog` 不丢失。

两阶段提交是跨系统维持数据逻辑一致性时常用的一个方案

`Redo log` 不是记录数据页“更新之后的状态”，而是记录这个页“做了什么改动”。

`Binlog` 有两种模式，`statement` 格式的话是记 SQL 语句，`row` 格式会记录行的内容，记两条，更新前和更新后都有。

`binlog` 还不能去掉：原因是，`redolog` 只有 InnoDB 有，别的引擎没有。另一个原因是，`redolog` 是循环写的，不持久保存，`binlog` 的“归档”这个功能，`redolog` 是不具备的。

## 03 | 事务隔离：为什么你改了我还看不见？

---

简单来说，事务就是要保证一组操作，要么全部成功，要么全部失败

MySQL 原生的 MyISAM 引擎就不支持事务，这也是 MyISAM 被 InnoDB 取代的重要原因

ACID (Atomicity、Consistency、Isolation、Durability，即原子性、一致性、隔离性、持久性)

隔离性

当数据库上有多个事务同时执行的时候，就可能出现脏读 ( dirty read )、不可重复读 ( non-repeatable read )、幻读 ( phantom read ) 的问题，为了解决这些问题，就有了“隔离级别”的概念。

|                |        |
|----------------|--------|
| 事务A            | 事务B    |
| 启动事务<br>查询得到值1 | 启动事务   |
|                | 查询得到值1 |
|                | 将1改成2  |
| 查询得到值V1        |        |
|                | 提交事务B  |
| 查询得到值V2        |        |
| 提交事务A          |        |
| 查询得到值V3        |        |

- 读未提交是指，一个事务还没提交时，它做的变更就能被别的事务看到。
- 读提交是指，一个事务提交之后，它做的变更才会被其他事务看到。
- 可重复读是指，一个事务执行过程中看到的数据，总是跟这个事务在启动时看到的数据是一致的
- 串行化，顾名思义是对于同一行记录，“写”会加“写锁”，“读”会加“读锁”。当出现读写锁冲突的时候，后访问的事务必须等前一个事务执行完成，才能继续执行。

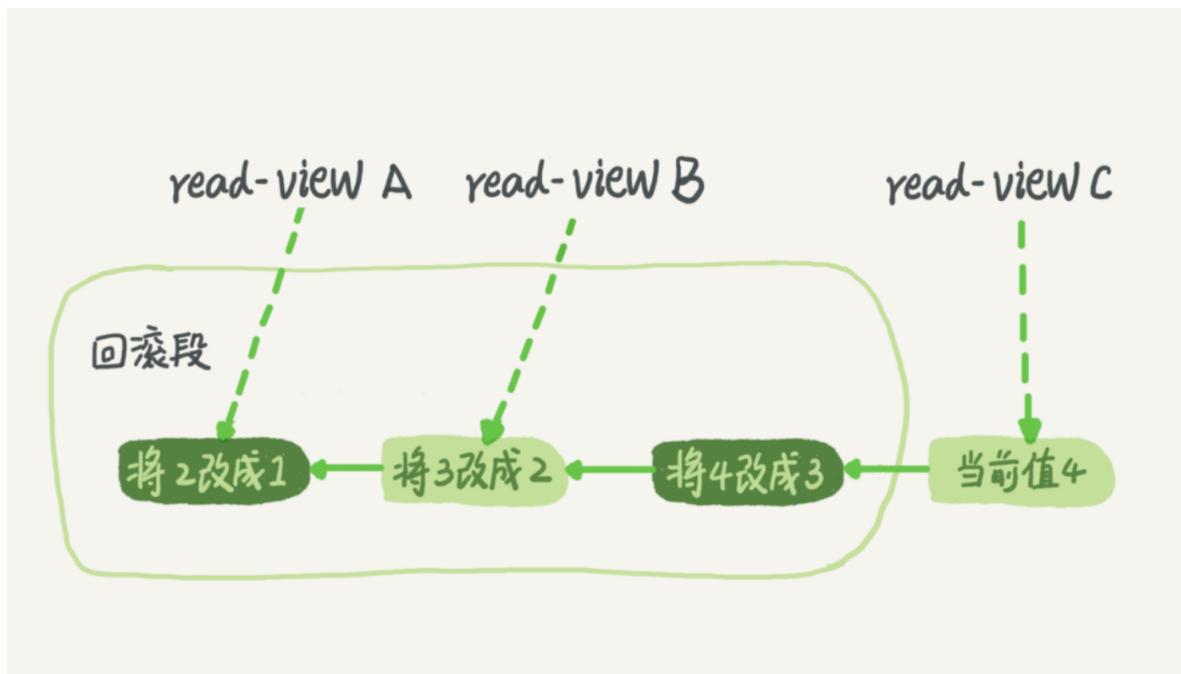
配置的方式是，将启动参数 transaction-isolation 的值设置成 READ-COMMITTED。你可以用 show variables 来查看当前的值

```
mysql> show variables like 'transaction_isolation';
```

|                       |                |
|-----------------------|----------------|
| variable_name         | value          |
| transaction_isolation | READ-COMMITTED |

在 MySQL 中，实际上每条记录在更新的时候都会同时记录一条回滚操作。记录上的最新值，通过回滚操作，都可以得到前一个状态的值。

假设一个值从 1 被按顺序改成了 2、3、4，在回滚日志里面就会有类似下面的记录



当前值是 4，但是在查询这条记录的时候，不同时刻启动的事务会有不同的 read-view。如图中看到的，在视图 A、B、C 里面，这一个记录的值分别是 1、2、4，同一条记录在系统中可以存在多个版本，就是数据库的多版本并发控制 (MVCC) con-currency。

对于 read-view A，要得到 1，就必须将当前值依次执行图中所有的回滚操作得到。

同时你会发现，即使现在有另外一个事务正在将 4 改成 5，这个事务跟 read-view A、B、C 对应的事务是不会冲突的。

## 事务的启动方式

MySQL 的事务启动方式有以下几种：

1. 显式启动事务语句，`begin` / `start transaction`。提交语句是 `commit`，回滚语句是 `rollback`。
2. `set autocommit=0`，将这个线程的自动提交关掉。意味着如果你只执行一个 `select` 语句，这个事务就启动了，而且并不会自动提交。这个事务持续存在直到你主动执行 `commit` 或 `rollback` 语句，或者断开连接。

# 04 | 深入浅出索引 (上)

索引的出现其实就是为了提高数据查询的效率，就像书的目录一样

## 索引的常见模型

提高读写效率的数据结构很多，三种常见的数据结构，它们分别是哈希表、有序数组和搜索树

哈希的思路很简单，把值放在数组里，用一个哈希函数把 key 换算成一个确定的位置，然后把 value 放在数组的这个位置

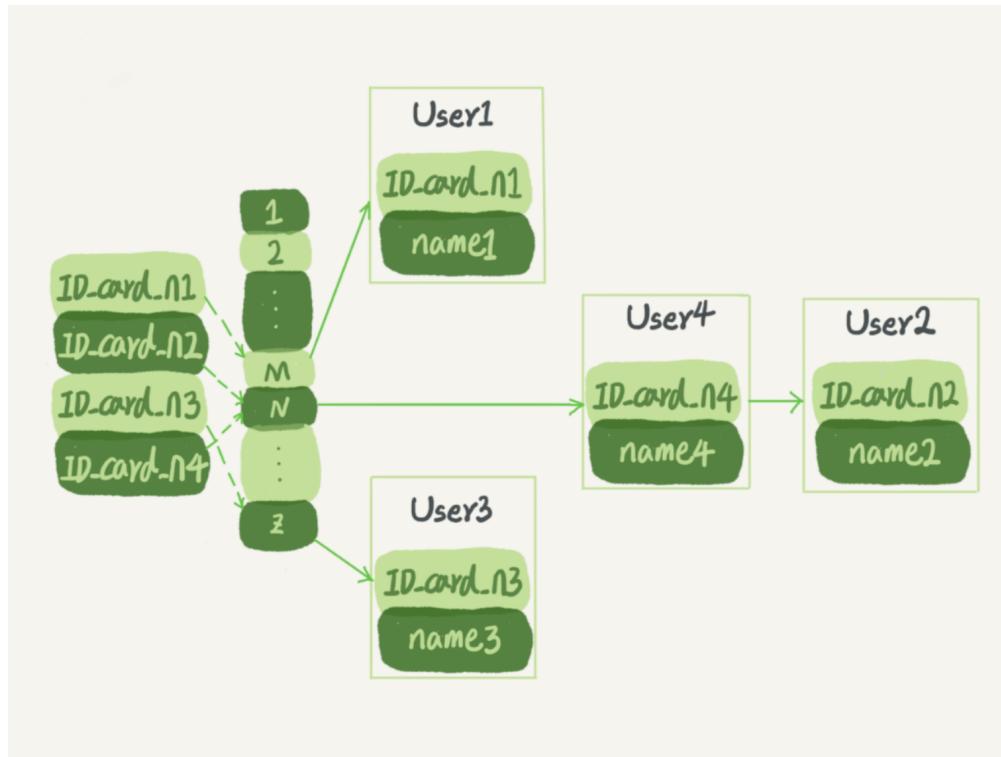


图 1 哈希表示意图

图中四个 ID\_card\_n 的值并不是递增的，这样做好处是增加新的 User 时速度会很快，只需要往后追加。

缺点是，因为不是有序的，所以哈希索引做区间查询的速度是很慢的。

如果你现在要找身份证号在 [ID\_card\_X, ID\_card\_Y] 这个区间的用户，就必须全部扫描一遍了。

所以，哈希表这种结构适用于只有等值查询的场景，比如 Memcached 及其他一些 NoSQL 引擎。

有序数组在等值查询和范围查询场景中的性能就都非常优秀

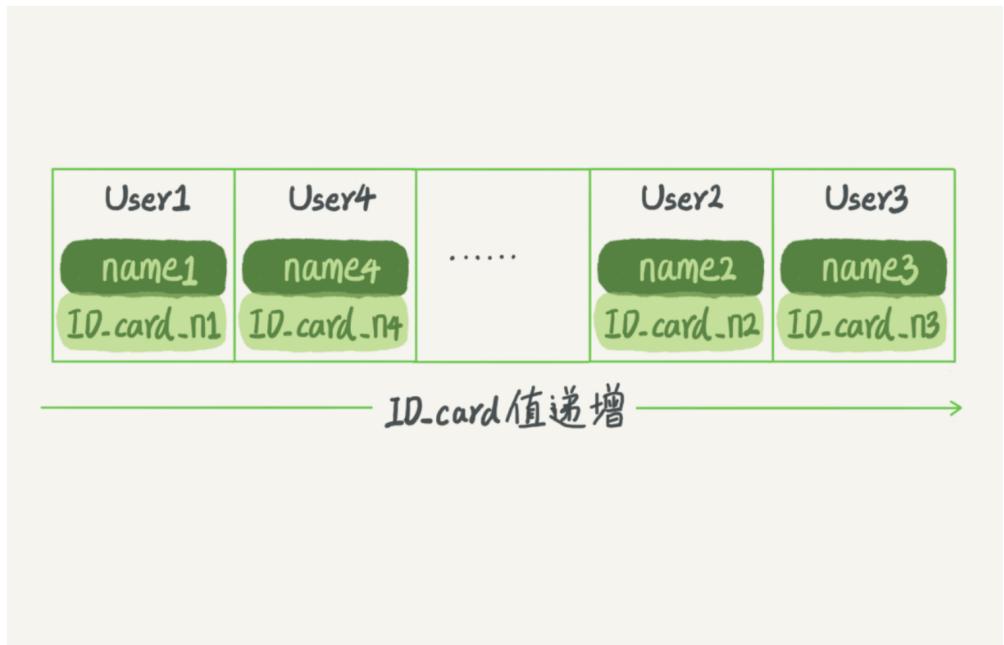


图 2 有序数组示意图

如果仅仅看查询效率，有序数组就是最好的数据结构了。但是，在需要更新数据的时候就麻烦了，你往中间插入一个记录就必须得挪动后面所有的记录，成本太高。

所以，**有序数组索引只适用于静态存储引擎**，比如你要保存的是 2017 年某个城市的所有人口信息，这类不会再修改的数据。

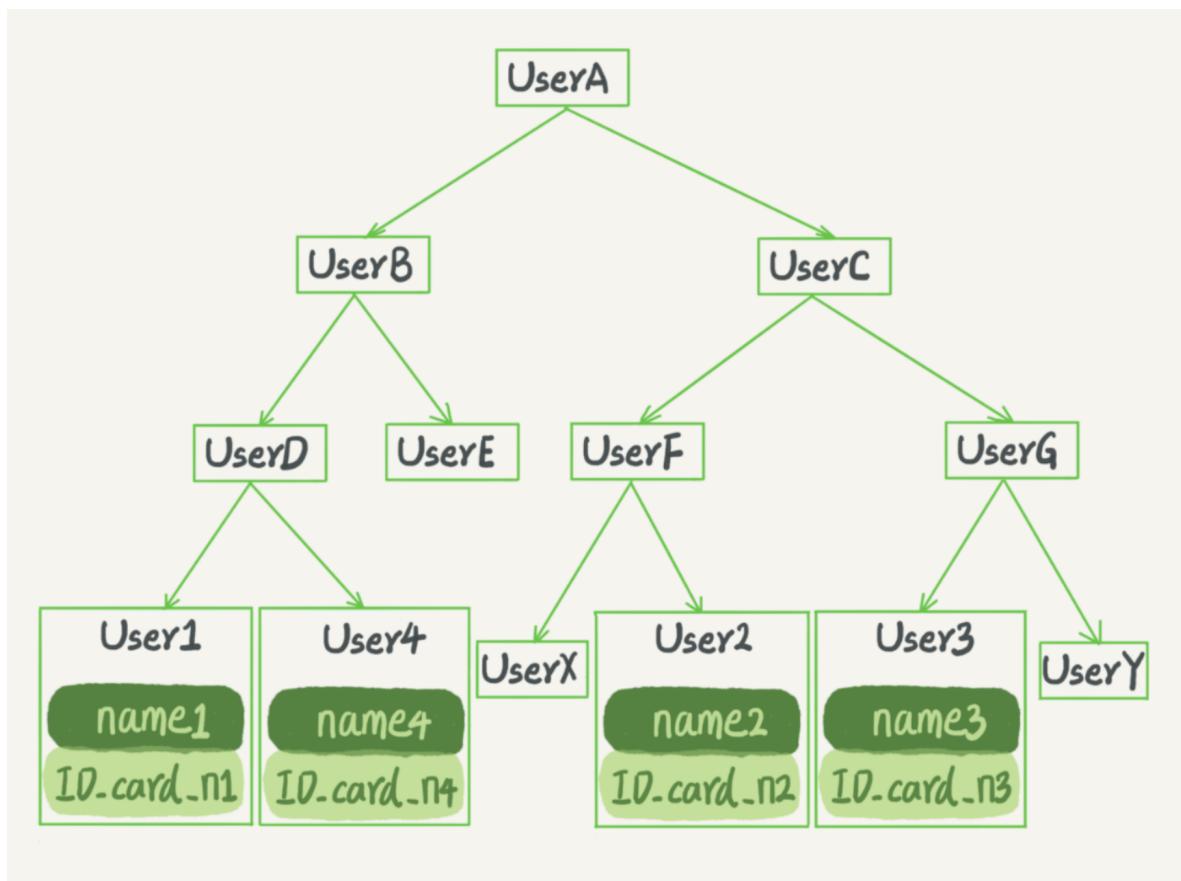


图 3 二叉搜索树示意图

二叉搜索树的特点是：每个节点的左儿子小于父节点，父节点又小于右儿子。

时间复杂度是  $O(\log(N))$ 。为了维持  $O(\log(N))$  的查询复杂度，你就需要保持这棵树是平衡二叉树。为了做这个保证，更新的时间复杂度也是  $O(\log(N))$ 。

树可以有二叉，也可以有多叉。多叉树就是每个节点有多个儿子，儿子之间的大小保证从左到右递增。二叉树是搜索效率最高的，但是实际上大多数的数据存储却并不使用二叉树。

其原因是，索引不止存在内存中，还要写到磁盘上。你可以想象一下一棵 100 万节点的平衡二叉树，树高 20。一次查询可能需要访问 20 个数据块。在机械硬盘时代，从磁盘随机读一个数据块需要 10 ms 左右的寻址时间。也就是说，对于一个 100 万行的表，如果使用二叉树来存储，单独访问一个行可能需要 20 个 10 ms 的时间，这个查询可真够慢的。

为了让一个查询尽量少地读磁盘，就必须让查询过程访问尽量少的数据块。那么，我们就不应该使用二叉树，而是要使用“N 叉”树。这里，“N 叉”树中的“N”取决于数据块的大小。

以 InnoDB 的一个整数字段索引为例，这个 N 差不多是 1200(16k/13B)。这棵树高是 4 的时候，就可以存 1200 的 3 次方个值，这已经 17 亿了

数据库技术发展到今天，跳表、LSM 树等数据结构也被用于引擎设计中

在 MySQL 中，索引是在存储引擎层实现的，所以并没有统一的索引标准，即不同存储引擎的索引的工作方式并不一样。而即使多个存储引擎支持同一种类型的索引，其底层的实现也可能不同

## InnoDB 的索引模型

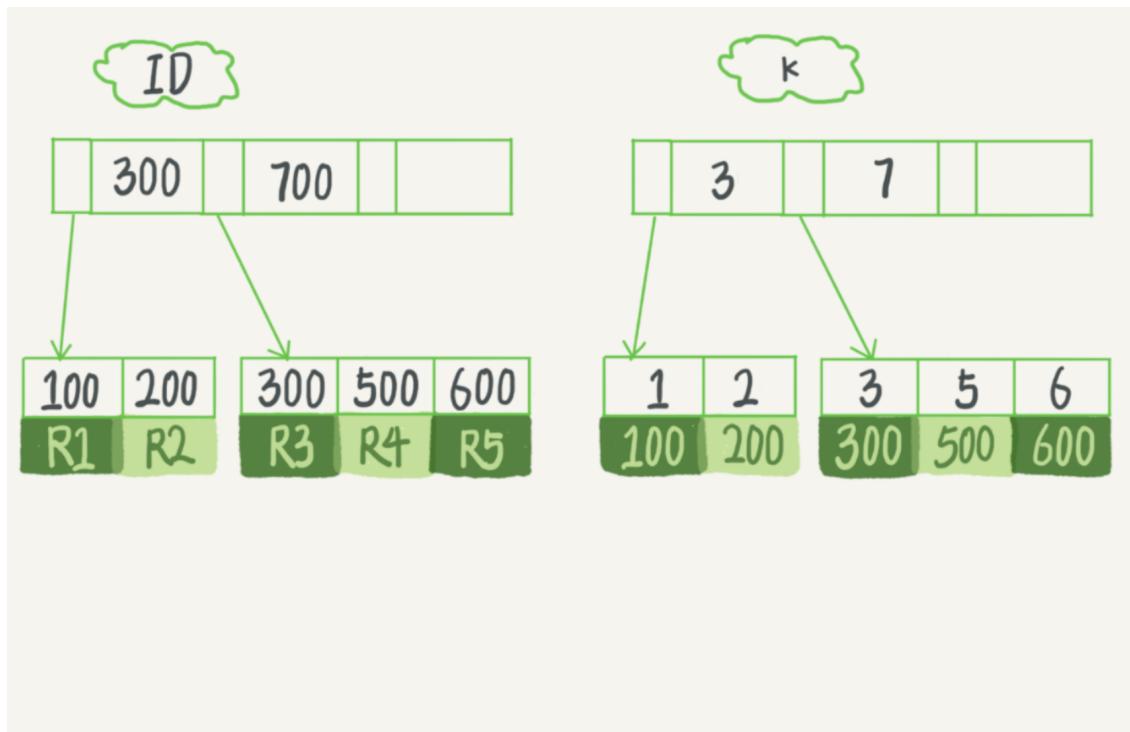


图 4 InnoDB 的索引组织结构

```
mysql> create table t(
id int primary key,
k int not null,
name varchar(16),
index (k))engine=InnoDB;
```

R1~R5 的 (ID,k) 值分别为 (100,1)、(200,2)、(300,3)、(500,5) 和 (600,6)

根据叶子节点的内容，索引类型分为主键索引和非主键索引。

主键索引的叶子节点存的是整行数据。在 InnoDB 里，主键索引也被称为聚簇索引（clustered index）。

非主键索引的叶子节点内容是主键的值。在 InnoDB 里，非主键索引也被称为二级索引（secondary index）。

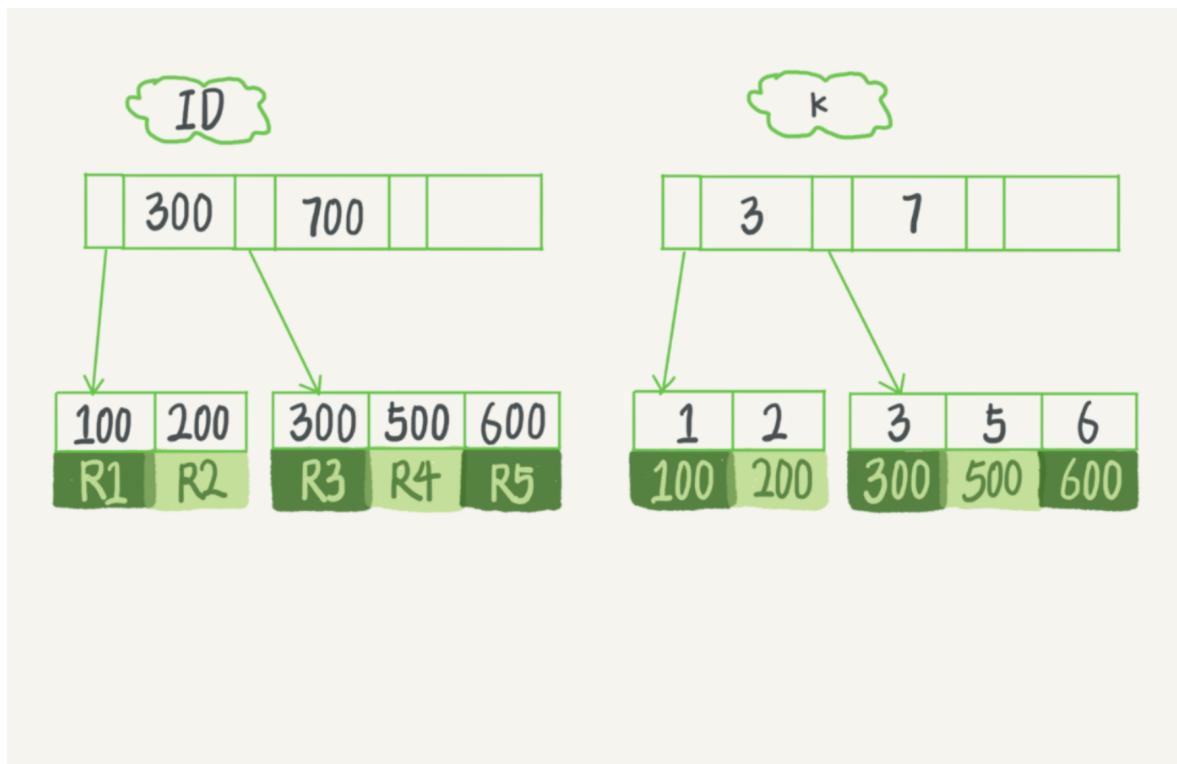
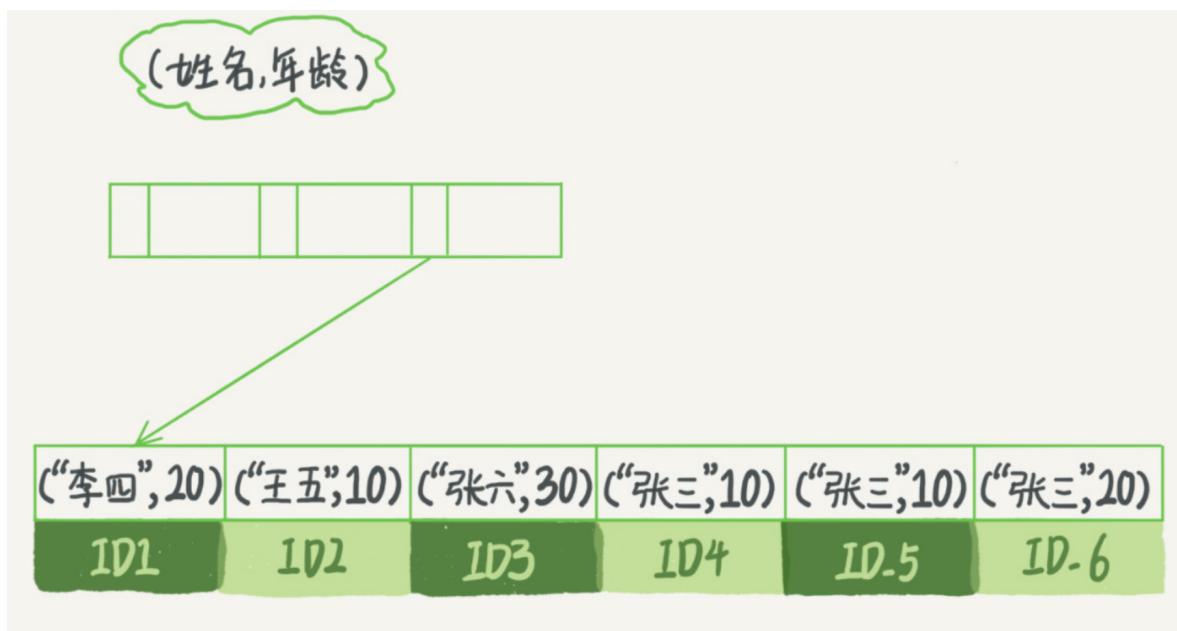


图 1 InnoDB 的索引组织结构



## 索引维护

B+ 树为了维护索引有序性，在插入新值的时候需要做必要的维护

整型做主键，那么每个二级索引的叶子节点则只要 4 个字节

以市民表的联合索引 ( name, age ) 为例。如果现在有一个需求：检索出表中“名字第一个字是张，而且年龄是 10 岁的所有男孩”。那么，SQL 语句是这么写的：

```
mysql> select * from tuser where name like '张 %' and age=10 and ismale=1;
```

这个语句在搜索索引树的时候，只能用“张”，找到第一个满足条件的记录 ID3。

然后呢？在 MySQL 5.6 之前，只能从 ID3 开始一个个回表。到主键索引上找出数据行，再对比字段值。

索引下推优化 ( index condition pushdown)，可以在索引遍历过程中，对索引中包含的字段先做判断，直接过滤掉不满足条件的记录，减少回表次数

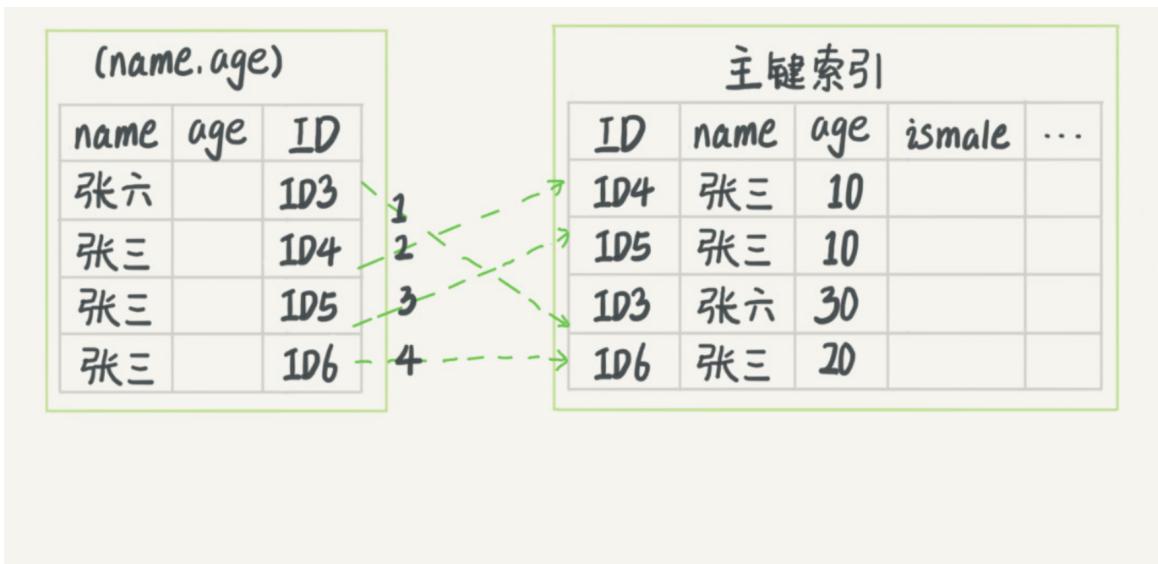


图 3 无索引下推执行流程

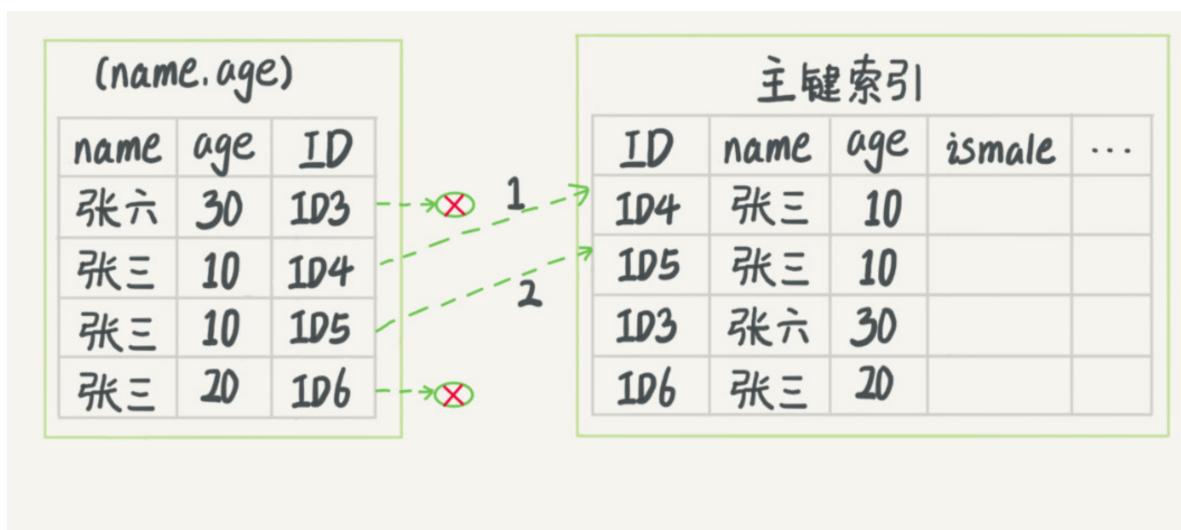


图 4 索引下推执行流程

覆盖索引、前缀索引、索引下推。你可以看到，在满足语句需求的情况下，尽量少地访问资源是数据库设计的重要原则之一

```
CREATE TABLE `geek` (
  `a` int(11) NOT NULL,
  `b` int(11) NOT NULL,
  `c` int(11) NOT NULL,
  `d` int(11) NOT NULL,
  PRIMARY KEY (`a`,`b`),
  KEY `c` (`c`),
  KEY `ca` (`c`,`a`),
  KEY `cb` (`c`,`b`)
) ENGINE=InnoDB;
```

问题：

建表语句是：

```
mysql> create table T(
  id int primary key,
  k int not null,
  name varchar(16),
  index (k))engine=InnoDB;
```

对于上面例子中的 InnoDB 表 T，如果你要重建索引 k，你的两个 SQL 语句可以这么写：

```
alter table T drop index k;
alter table T add index(k);
```

如果你要重建主键索引，也可以这么写：

```
alter table T drop primary key;
alter table T add primary key(id);
```

我的问题是，对于上面这两个重建索引的作法，说出你的理解。如果有不合适的，为什么，更好的方法是什么？

在评论区，有同学问到为什么要重建索引。我们文章里面有提到，索引可能因为删除，或者页分裂等原因，导致数据页有空洞，重建索引的过程会创建一个新的索引，把数据按顺序插入，这样页面的利用率最高，也就是索引更紧凑、更省空间。

这道题目，我给你的“参考答案”是：

重建索引 k 的做法是合理的，可以达到省空间的目的。但是，重建主键的过程不合理。不论是删除主键还是创建主键，都会将整个表重建。所以连着执行这两个语句的话，第一个语句就白做了。这两个语句，你可以用这个语句代替： alter table T engine=InnoDB

问题

实际上主键索引也是可以使用多个字段的。有这么一个表，表结构定义类似这样的：

```
CREATE TABLE `geek` (
  `a` int(11) NOT NULL,
  `b` int(11) NOT NULL,
  `c` int(11) NOT NULL,
  `d` int(11) NOT NULL,
  PRIMARY KEY (`a`,`b`),
  KEY `c` (`c`),
  KEY `ca` (`c`,`a`),
  KEY `cb` (`c`,`b`)
) ENGINE=InnoDB;
```

公司的同事告诉他说，由于历史原因，这个表需要 a、b 做联合主键，这个小吕理解了。

但是，学过本章内容的小吕又纳闷了，既然主键包含了 a、b 这两个字段，那意味着单独在字段 c 上创建一个索引，就已经包含了三个字段了呀，为什么要创建“ca”“cb”这两个索引？

同事告诉他，是因为他们的业务里面有这样的两种语句：

```
select * from geek where c=N order by a limit 1;
select * from geek where c=N order by b limit 1;
```

我给你的问题是，这位同事的解释对吗，为了这两个查询模式，这两个索引是否都是必须的？为什么呢？

## 06 | 全局锁和表锁：给表加个字段怎么有这么多阻碍？

数据库锁设计的初衷是处理并发问题

根据加锁的范围，MySQL 里面的锁大致可以分成全局锁、表级锁和行锁三类

全局锁就是对整个数据库实例加锁。MySQL 提供了一个加全局读锁的方法，

命令是 **Flush tables with read lock (FTWRL)**

其他线程的以下语句会被阻塞：数据更新语句（数据的增删改）、数据定义语句（包括建表、修改表结构等）和更新类事务的提交语句

DML(Data Manipulation Language)

DDL(Data Definition Language) create\drop\alter

全局锁的典型使用场景是，做全库逻辑备份

在可重复读隔离级别下开启一个事务，官方自带的逻辑备份工具是 mysqldump。使用参数`--single-transaction`，导数据之前就会启动一个事务，来确保拿到一致性视图

**single-transaction** 方法只适用于所有的表使用事务引擎的库。如果有的表使用了不支持事务的引擎，那么备份就只能通过 FTWRL 方法。这往往是 DBA 要求业务开发人员使用 InnoDB 替代 MyISAM 的原因之一。

有了这个功能，为什么还需要 FTWRL 呢？一致性读是好，但前提是引擎要支持这个隔离级别。比如，对于 MyISAM 这种不支持事务的引擎，如果备份过程中有更新，总是只能取到最新的数据，那么就破坏了备份的一致性。这时，我们就需要使用 FTWRL 命令了。

你也许会问，既然要全库只读，为什么不使用 **set global readonly=true** 的方式呢？确实 readonly 方式也可以让全库进入只读状态，但我还是会建议你用 FTWRL 方式，主要有两个原因：

- 一是，在有些系统中，readonly 的值会被用来做其他逻辑，比如用来判断一个库是主库还是备库。因此，修改 global 变量的方式影响面更大，我不建议你使用。
- 二是，在异常处理机制上有差异。如果执行 FTWRL 命令之后由于客户端发生异常断开，那么 MySQL 会自动释放这个全局锁，整个库回到可以正常更新的状态。而将整个库设置为 readonly 之后，如果客户端发生异常，则数据库就会一直保持 readonly 状态，这样会导致整个库长时间处于不可写状态，风险较高。

## 表级锁

MySQL 里面表级别的锁有两种：一种是表锁，一种是元数据锁（meta data lock，MDL）。

表锁的语法是 **lock tables ... read/write**。与 FTWRL 类似，可以用 **unlock tables** 主动释放锁，也可以在客户端断开的时候自动释放。

举个例子，如果在某个线程 A 中执行 **lock tables t1 read, t2 write;**

这个语句，则其他线程写 t1、读写 t2 的语句都会被阻塞。

同时，线程 A 在执行 **unlock tables** 之前，也只能执行读 t1、读写 t2 的操作。连写 t1 都不允许，自然也不能访问其他表。

另一类表级的锁是 **MDL (metadata lock)**。MDL 不需要显式使用，在访问一个表的时候会被自动加上。

MDL 的作用是，保证读写的正确性，当对一个表做增删改查操作的时候，加 MDL 读锁；当要对表做结构变更操作的时候，加 MDL 写锁。

- 读锁之间不互斥，因此你可以有多个线程同时对一张表增删改查。
- 读写锁之间、写锁之间是互斥的，用来保证变更表结构操作的安全性。因此，如果有两个线程要同时给一个表加字段，其中一个要等另一个执行完才能开始执行。

## 如何安全地给小表加字段？

给一个小表加个字段，导致整个库挂了：

| session A                   | session B                   | session C                                | session D                                |
|-----------------------------|-----------------------------|------------------------------------------|------------------------------------------|
| begin;                      |                             |                                          |                                          |
| select *<br>from t limit 1; |                             |                                          |                                          |
|                             | select *<br>from t limit 1; |                                          |                                          |
|                             |                             | alter table t<br>add f int;<br>(blocked) |                                          |
|                             |                             |                                          | select *<br>from t limit 1;<br>(blocked) |

我们可以看到 session A 先启动，这时候会对表 t 加一个 MDL 读锁。由于 session B 需要的也是 MDL 读锁，因此可以正常执行。

之后 session C 会被 blocked，是因为 session A 的 MDL 读锁还没有释放，而 session C 需要 MDL 写锁，因此只能被阻塞。

如果只有 session C 自己被阻塞还没什么关系，但是之后所有要在表 t 上新申请 MDL 读锁的请求也会被 session C 阻塞。前面我们说了，所有对表的增删改查操作都需要先申请 MDL 读锁，就都被锁住，等于这个表现在完全不可读写了。

解决方案：首先我们要解决长事务，事务不提交，就会一直占着 MDL 锁，在 alter table 语句里面设定等待时间，如果在这个指定的等待时间里面能够拿到 MDL 写锁最好，拿不到也不要阻塞后面的业务语句，先放弃

问题：备份一般都会在备库上执行，你在用-single-transaction 方法做逻辑备份的过程中，如果主库上的一个小表做了一个 DDL，比如给一个表上加了一列。这时候，从备库上会看到什么现象呢？

假设这个 DDL 是针对表 t1 的，这里我把备份过程中几个关键的语句列出来：

```

Q1:SET SESSION TRANSACTION ISOLATION LEVEL REPEATABLE READ;
Q2:START TRANSACTION WITH CONSISTENT SNAPSHOT ;
/* other tables */
Q3:SAVEPOINT sp;
/* 时刻 1 */
Q4:show create table `t1`;
/* 时刻 2 */
Q5:SELECT * FROM `t1`;
/* 时刻 3 */
Q6:ROLLBACK TO SAVEPOINT sp;
/* 时刻 4 */
/* other tables */

```

在备份开始的时候，为了确保 RR ( 可重复读 ) 隔离级别，再设置一次 RR 隔离级别 (Q1)；

启动事务，这里用 WITH CONSISTENT SNAPSHOT 确保这个语句执行完就可以得到一个一致性视图 (Q2)；

设置一个保存点，这个很重要（Q3）；

show create 是为了拿到表结构（Q4），然后正式导数据（Q5），回滚到 SAVEPOINT sp，在这里的作  
用是释放 t1 的 MDL 锁（Q6）。当然这部分属于“超纲”，上文正文里面都没提到。

DDL 从主库传过来的时间按照效果不同，我打了四个时刻。题目设定为小表，我们假定到达后，如果开  
始执行，则很快能够执行完成。

参考答案如下：

1. 如果在 Q4 语句执行之前到达，现象：没有影响，备份拿到的是 DDL 后的表结构。
2. 如果在“时刻 2”到达，则表结构被改过，Q5 执行的时候，报 Table definition has changed,  
please retry transaction，现象：mysqldump 终止；
3. 如果在“时刻 2”和“时刻 3”之间到达，mysqldump 占着 t1 的 MDL 读锁，binlog 被阻塞，现象：  
主从延迟，直到 Q6 执行完成。
4. 从“时刻 4”开始，mysqldump 释放了 MDL 读锁，现象：没有影响，备份拿到的是 DDL 前的表结  
构。

## 07 | 行锁功过：怎么减少行锁对性能的影 响？

MySQL 的行锁是在引擎层由各个引擎自己实现的。但并不是所有的引擎都支持行锁，比如 MyISAM 引  
擎就不支持行锁。

不支持行锁意味着并发控制只能使用表锁，同一张表上任何时刻只能有一个更新在执行，这会影响到  
业务并发度

行锁就是针对数据表中行记录的锁。这很好理解，比如事务 A 更新了一行，而这时候事务 B 也要更新  
同一行，则必须等事务 A 的操作完成后才能进行更新

### 两阶段锁说起

事务 B 的 update 语句会被阻塞，直到事务 A 执行 commit 之后，事务 B 才能继续执行



在 InnoDB 事务中，行锁是在需要的时候才加上的，但并不是不需要了就立刻释放，而是要等到事务结  
束时才释放。这个就是两阶段锁协议

原则：如果你的事务中需要锁多个行，要把最可能造成锁冲突、最可能影响并发度的锁的申请时机尽量往后放。

1. 从顾客 A 账户余额中扣除电影票价；
2. 给影院 B 的账户余额增加这张电影票价；
3. 记录一条交易日志。

### 死锁和死锁检测

循环资源依赖

两种策略：

- 一种策略是，直接进入等待，直到超时。这个超时时间可以通过参数 `innodb_lock_wait_timeout` 来设置。

在 InnoDB 中，`innodb_lock_wait_timeout` 的默认值是 50s，意味着如果采用第一个策略，当出现死锁以后，第一个被锁住的线程要过 50s 才会超时退出

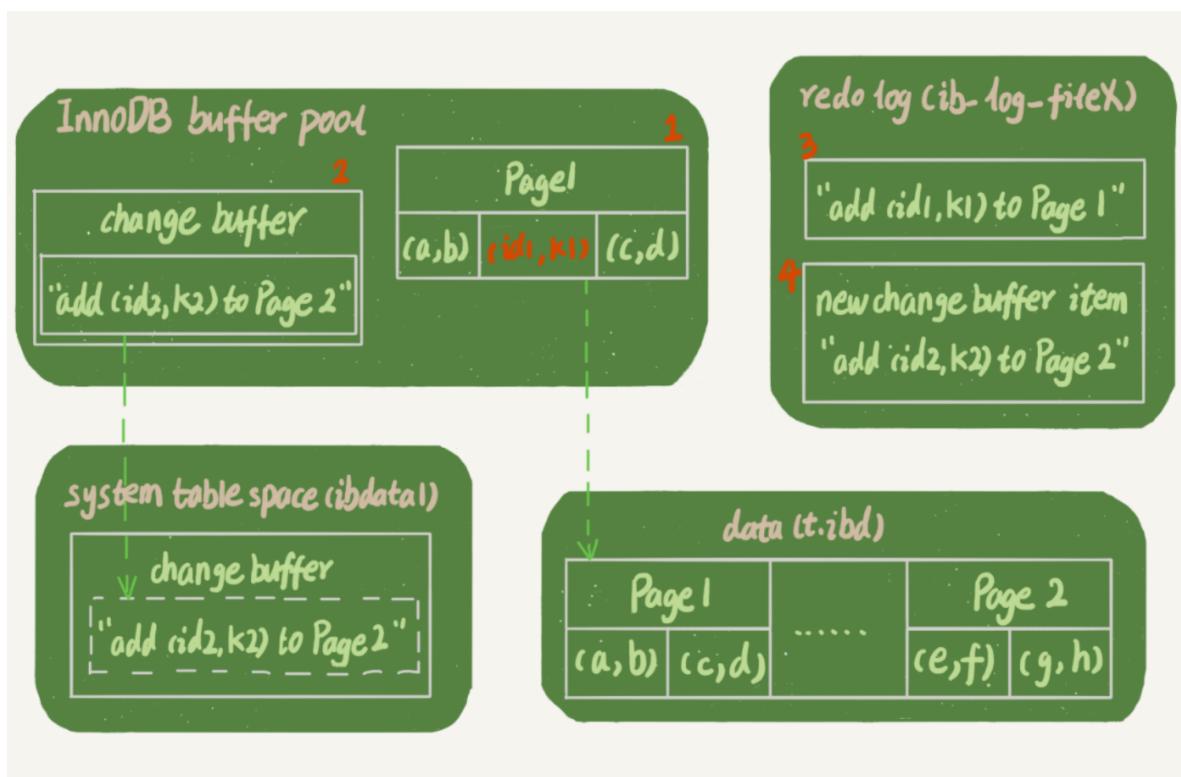
- 另一种策略是，发起死锁检测，发现死锁后，主动回滚死锁链条中的某一个事务，让其他事务得以继续执行。将参数 `innodb_deadlock_detect` 设置为 `on`，表示开启这个逻辑

主动死锁检测，而且 `innodb_deadlock_detect` 的默认值本身就是 `on`

但是，调整语句顺序并不能完全避免死锁。所以我们引入了死锁和死锁检测的概念，以及提供了三个方案，来减少死锁对数据库的影响。减少死锁的主要方向，就是控制访问相同资源的并发事务量。

## 08 | 事务到底是隔离的还是不隔离的？

讲事务隔离级别的时候提到过，如果是可重复读隔离级别，事务 T 启动的时候会创建一个视图 `read-view`，之后事务 T 执行期间，即使有其他事务修改了数据，事务 T 看到的仍然跟在启动时看到的一样。也就是说，一个在可重复读隔离级别下执行的事务，好像与世无争，不受外界影响。



## 20 | 幻读是什么，幻读有什么问题？

InnoDB 的默认事务隔离级别是可重复读

## 28 | 读写分离有哪些坑？

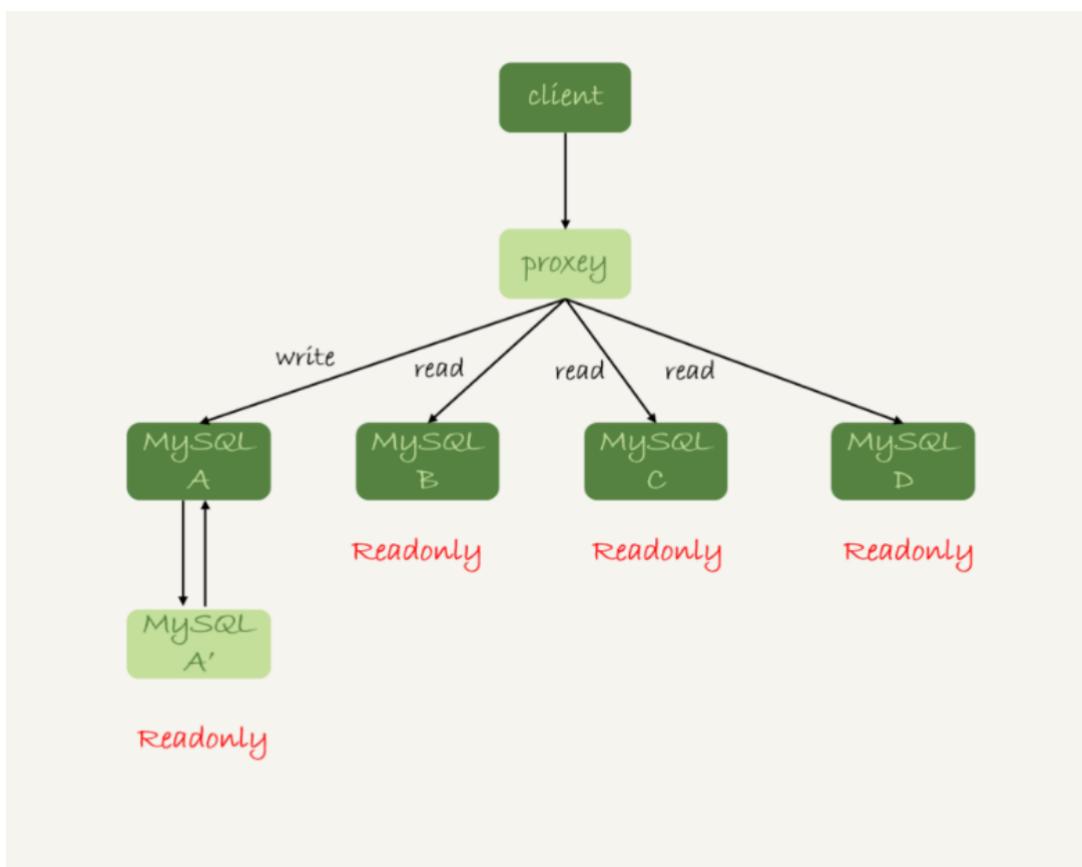


图 2 带 proxy 的读写分离架构

## 29 | 如何判断一个数据库是不是出问题了？