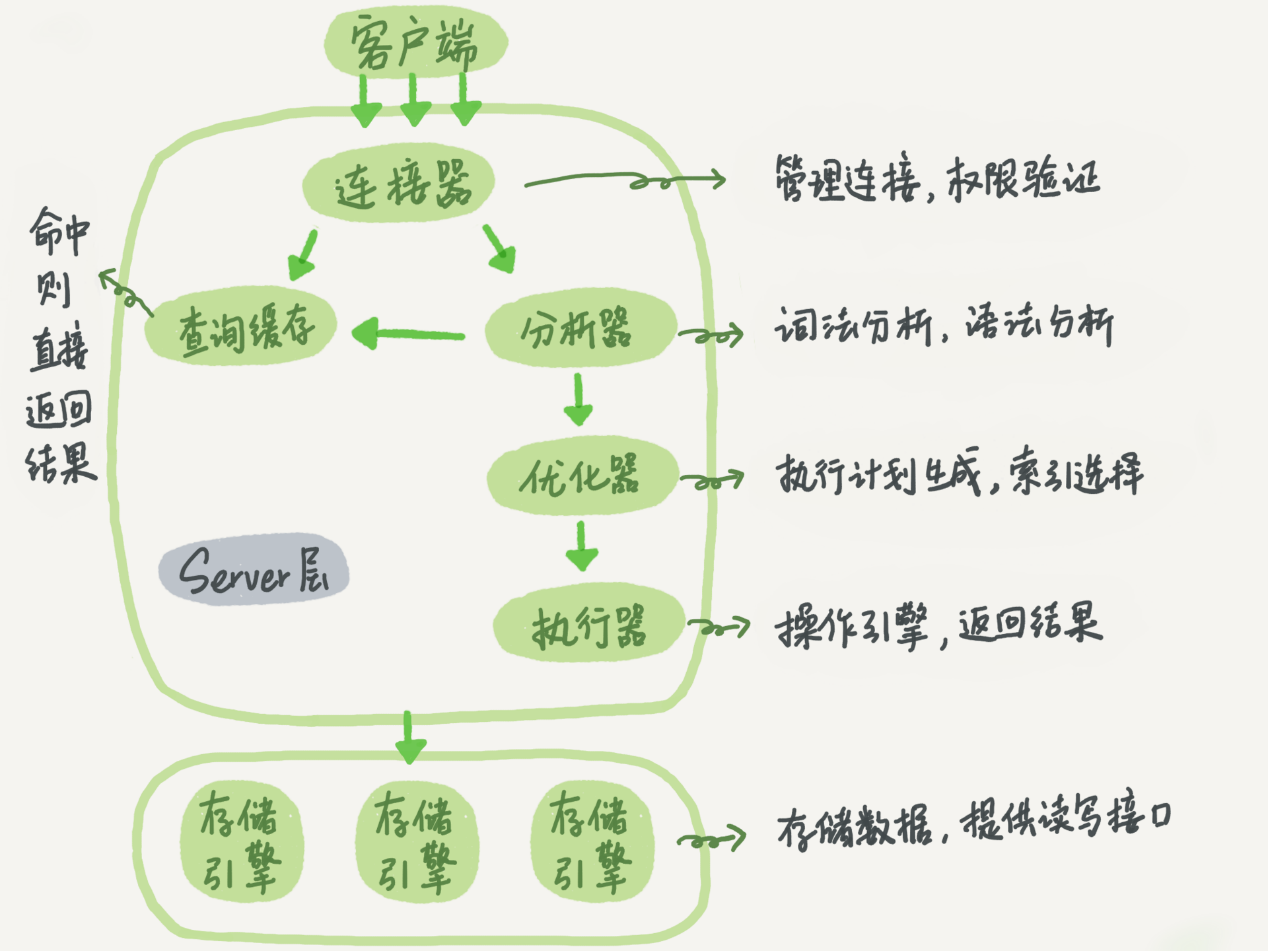
# Mysql基础知识

# 零 什么是mysql？

Mysql是一个多线程多用户健壮性SQL数据库服务器

# 一 SQL语句是如何执行的?



## 1.1 MySQL分层

MySQL 可以分为 Server 层和存储引擎层两部分

Server层： 连接器、查询缓存、分析器、优化器、执行器。核心服务功能，以及所有内置函数(如日期、时间、数学和加密函数等)，跨引擎功能都在这一层实现，比如存储过程、触发器、视图等。

存储引擎层：负责数据存储和提取。架构模式是插件式，支持MyISAM、InnoDB。默认使用InnoDB存储引擎，可以指定存储引擎类型来进行选择，比如通过create table语句中使用engine=memory。

不同存储引擎层共用一个Server层

## 1.2 连接器

负责与客户端建立链接、获取权限、维持和管理链接。

通过连接命令

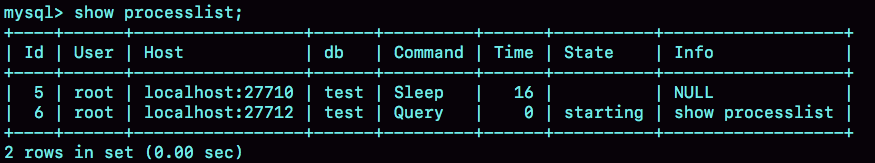
mysql -h$ip –p $port -u $user -p

建议不要输入在命令后面直接输入密码，有泄漏风险。

客户端与mysql服务器完成TCP握手后，连接器就要进行身份认证，验证用户名和密码

1. 如果用户名和密码不正确，则会收到一个“Access denied for user”的错误，然后客户端程序结束运行
2. 如果用户名密码认证通过，连接器会到权限表里面查出拥有的权限，然后这个连接里面的权限判断逻辑，都会依赖此时读到的权限。这意味着如果此时管理员修改了这个用户权限，也不会印象已经存在连接的权限，只会对新建的连接才会使用新的权限设置。

连接完成后，如没有后续动作，连接处于空闲状态，可以在show processlist命令看到它。



默认空闲断开时间是8小时

长连接的优势是减少了建立连接的动作，

劣势是MySQL占用内存涨的太快，这是因为MySQL在执行过程中使用的内存是管理在连接对象中的，这些资源会在连接断开时才进行释放，长期积累下来，会导致内存占用太大。被系统强行杀死（OOM），就像MySQL异常重启了

解决方法：

1. 定期断开长连接
2. MySQL5.7或更新版本，每执行比较大操作后，通过执行mysql\_reset\_connection来重新初始化连接资源，无需重连和重做权限验证。

## 1.3 查询缓存

Mysql拿到查询请求后，会先到查询缓存中查看，之前是否执行过这条语句。执行语句及其结果会以key-value对形式缓存到内存中。如果能在缓存中找到这个key，value就会直接返回给客户端。

如果语句不在查询缓存中，就会继续后面的执行阶段。执行完成后，执行结果会被存入查询缓存中。

**查询缓存不建立使用**：

1. 多数情况不会总是查询同一sql语句
2. 查询缓存失效非常频繁，只要有一个表更新，所有查询缓存都会失效。

## 1.4 分析器

SQL语句词法分析、语法分析

## 1.5 优化器

确定执行方案

表有多个索引，决定使用哪个索引。

一个语句有多表关联，决定表连接顺序。

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。

## 1.6 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：

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；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。

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。

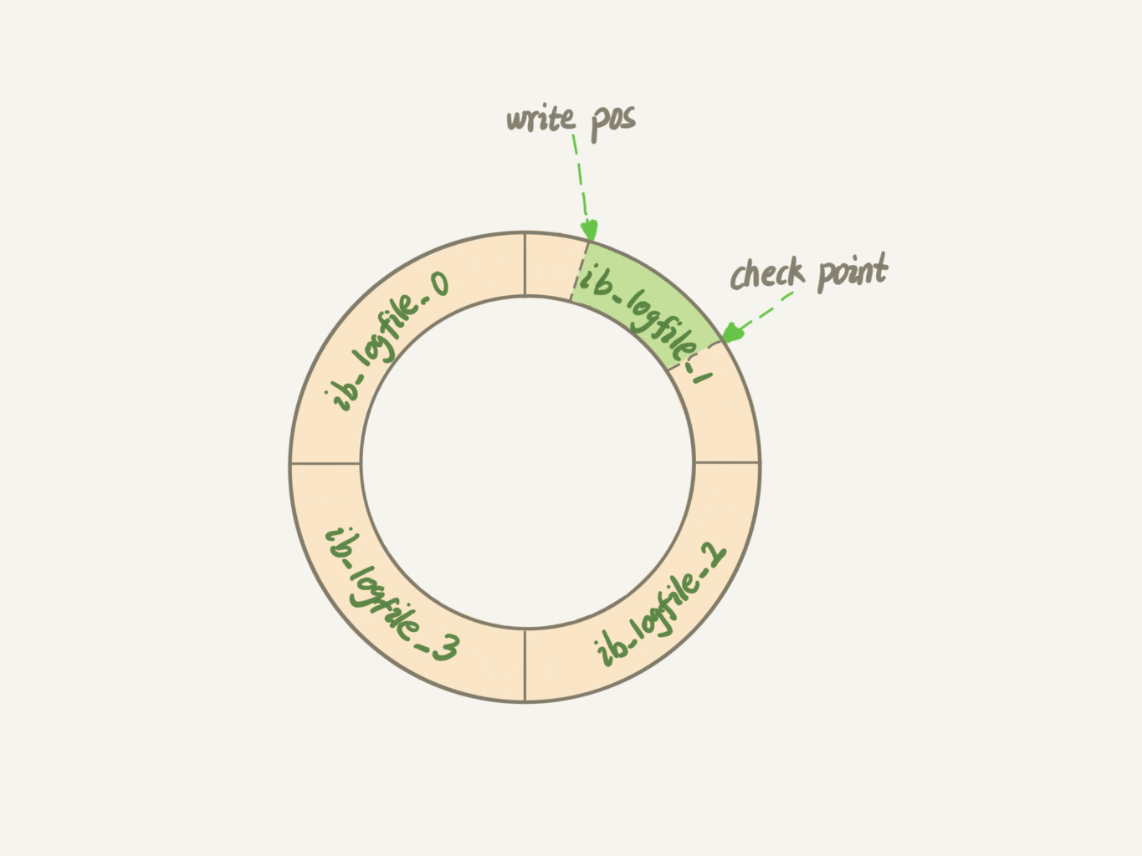
至此，这个语句就执行完成了。对于有索引的表，执行的逻辑也差不多。第一次调用的是“取满足条件的第一行”这个接口，之后循环取“满足条件的下一行”这个接口，这些接口都是引擎中已经定义好的。你会在数据库的慢查询日志中看到一个 rows\_examined 的字段，表示这个语句执行过程中扫描了多少行。这个值就是在执行器每次调用引擎获取数据行的时候累加的。

# 二 日志系统：一条SQL更新语句是如何执行的？

更新语句涉及日志模块，redo log和binlog

## redo log

redo log:固定大小，可配置一组4个文件，每个文件4GB，从头开始写，写到末尾就回到开头循环写。



write pos是当前记录位置，一边写一遍后移，写到三号文件末尾就回到0号文件开头，checkpoint是当前要擦除位置，也是往后推移并循环的，擦除记录前要把记录更新到数据文件。

write pos和checkpoint之间的空白部分，可以用来记录新操作，如果write pos追上checkpoint，则不能继续执行新更新，需要停下擦掉一些记录，把checkpoint推进一下

redo log保证InnoDB即使数据库发生异常重启，之前提交记录也不会丢失，这就是**crash-safe.**

## binlog

与redo log是InnoDB所特有不同，binlog是Server层的日志模块

MySQL自带引擎是MyISAM，但是没有crash-safe能力，binlog日志只能用于归档。redo log能够实现crash-safe能力

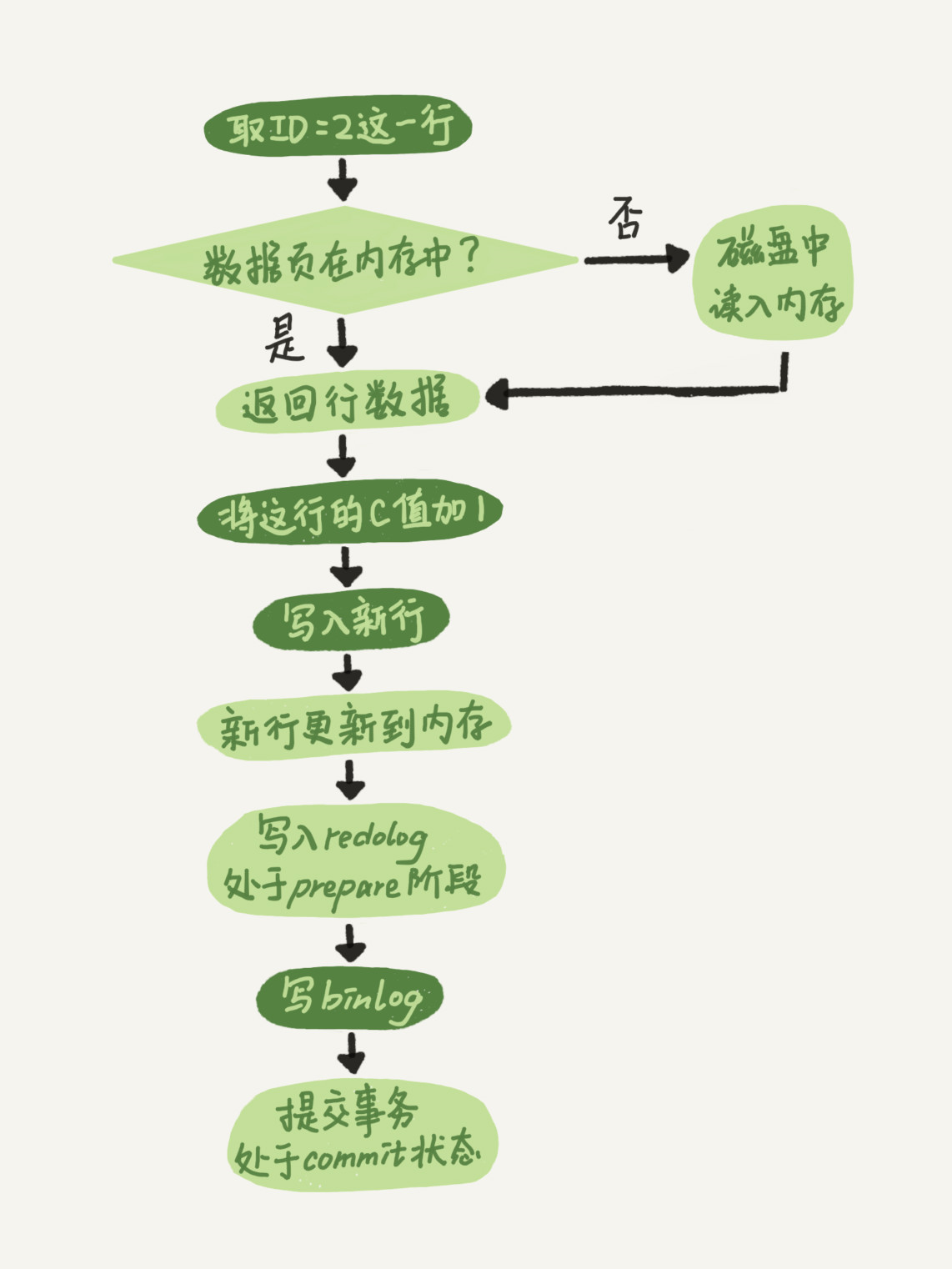
binlog 与 redo log不同：

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。

有了对这两个日志的概念性理解，我们再来看执行器和 InnoDB 引擎在执行这个简单的 update 语句时的内部流程。

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。

这里我给出这个 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。



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

两阶段提交的原因是：数据库恢复到半月前任意一秒状态，需要使用到全量备份和binlog日志。如果不用两阶段提交，在数据恢复的过程中，无论是采用先写redo log后写binlog，或者是先写binlog后写redo log，在数据库系统崩溃时，都会导致数据库恢复出现数据不一致的状况，尤其是需要扩容时，需要增加备库增加读能力，会使主从数据库出现不一致的情况。

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。

# 三 事务隔离：为什么你改了我还是看不见

事务：保证一组数据库操作，要么全部成功，要么全部失败。

InnoDb支持事务，MyISAM不支持事务，这是MyISAM被InnoDB取代的重要原因之一。

## 事务隔离性与隔离级别

数据库上有多个事务同时执行时，就可能出现脏读(dirty read)、不可重复度(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。

隔离级别越高，效率越低。

事务隔离级别包括：

读未提交(read uncommitted)、读提交(read committed)、可重复读(repeatable read)和串行化(serializable)

**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。当然在可重复读隔离级别下，未提交变更对其他事务也是不可见的。

**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

## Mysql锁基础知识

### 3.2.1 行锁与表锁

**行锁**:只锁住某一行记录，其他记录不受影响

**表锁**:锁住整张表，所有对于该表操作都会受到影响。

### 3.2.2 S锁与X锁

传统锁有两种:

**共享锁**(Shared Locks): 简称S锁，事务对一条记录进行读操作时，需要先获取该记录的共享锁。

**排他锁**(Exclusive Locks): 简称X锁，事务对一条记录进行写操作时，需要先获取该记录的排他锁。

S锁与S锁兼容，S锁与X锁不兼容，X锁与X锁也是不兼容的。

读取时对记录加S锁:

SELECT … LOCK IN SHARE MODE;

如果事务执行了该语句，就会在读取的记录上加S锁，这样就允许其他事务也能获取到该记录的S锁；如果其他事务需要获取该记录的X锁，那么就需要等待当前事务提交后释放S锁。

读取时对记录加X锁：

SELECT … FOR UPDATE;

如果事务执行了该语句，则会在读取的时候加上X锁。这样其他事务想要获取该记录的S锁或者X锁，就需要等待当前事务提交后释放掉X锁。

### 3.2.3 基于锁实现隔离级别

在基于锁实现方式下，四种隔离级别的区别就在于加锁方式的区别：

**读未提交**： 读操作不加锁，读读、读写、写读并行。写操作加X锁且直到事务提交后才进行释放

**读已提交**: 读操作加S锁，写操作加X锁且知道事务提交后才释放；读操作不会阻塞其他事务读或写，写操作会阻塞其他事务写或读，因此可以防止脏读问题。

**可重复读**：写操作加S锁且知道事务提交后才释放，写操作加X锁且直到事务提交后才释放；读操作不会阻塞其他事务读但是会阻塞其他事务写，写操作会阻塞其他事务读或写，因此可以防止脏读、不可重复读。

**串行化**: 读操作和写操作都会加X锁且知道事务提交后才释放，粒度为表锁，也就是严格串行。

### 3.2.4 长锁与短锁

**长锁**：如果锁获取后直到事务提交后才释放，这种锁称之为长锁

**短锁**: 如果锁在操作完成后就被释放，这种锁称之为短锁。

例如在读已提交隔离级别下，读操作所加S锁为短锁，写操作所加X锁为长锁。

### 3.2.5 两阶段锁

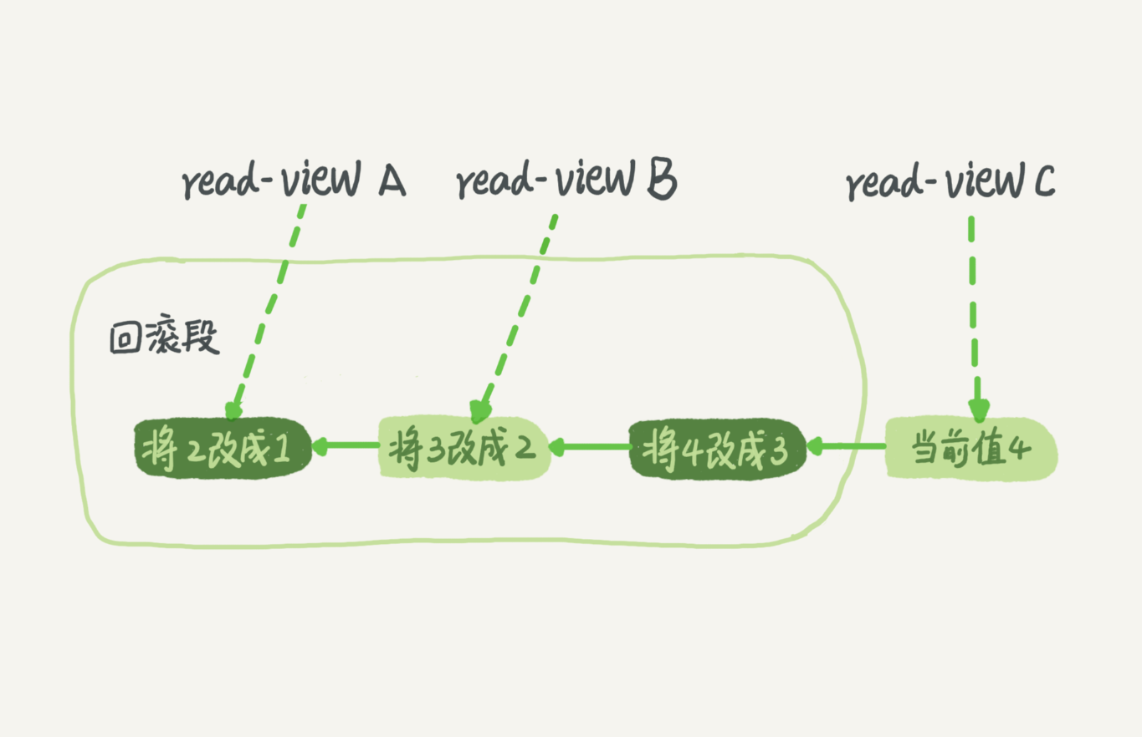
**两阶段锁**：将获取锁和释放所分为两个不同的阶段称之为两阶段锁协议，例如可重复读和串行化隔离级别，读操作所加S锁和写操作所加X锁均为长锁，即事务获取锁之后直到事务提交后才能释放。

两阶段锁规定在加锁阶段，一个事务可以获取锁但是不能释放锁；而在解锁阶段事务可以释放锁，并不能获得新的锁。两阶段锁协议能够保证事务串行化执行，解决事务并发问题，但也会导致死锁发生概率大大提升。

## 多版本并发控制（MVCC）

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，但是查询这条记录时，不同时刻启动的事务会有不同的read-view。

多版本并发控制(MVCC):同一条记录在系统中可以存在多个版本。

回滚日志的删除时机：当系统里没有比这个回滚日志更早的read-view时，就可以删除。

## 事务启动方式

长事务意味着系统会有很老的事务视图。由于这些事务随时可以访问数据库里的任何数据。所以事务提交以前，数据库里它肯呢个用到的回滚记录都必须保留，这就会导致占用大量存储空间。回滚日志跟数据字典一起放在ibdata文件里的。

MySQL事务启动方式两种：

1. 显式启动事务，begin或start transaction。提交语句commit，回滚语句rollback。
2. set autocommit=0，这个命令会将这个线程的自动提交关掉。以为只要执行一个select语句，事务就启动了，并且不会自动提交。事务持续到执行commit或rollback语句，或者断开连接。

建议使用set autocommit=1，通过显式语句来启动事务。避免长连接下的长事务。

### 长事务查询方法

select \* from information\_schema.innodb\_trx where TIME\_TO\_SEC(timediff(now(),trx\_started))>60

# 四MySQL数据库索引

## 4.1 索引常见模型

索引的目的是为了提高数据查询效率，实现索引的方式有很多种。三种常见的索引数据结构有：哈希表、有序数组和搜索树。

### 4.1.1 哈希表

哈希表是一种键-值(key-value)存储数据的结构，只需要输入带查找的键即key，就可以查找对应的值即value。哈希的思路是，将值放在数组里面，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。

解决哈希冲突的三种方法：

1. 拉链法（链地址法）

哈希冲突的位置将节点放入数组链表中。

2 开放地址法

* 1. 线性探查

发生冲突时，往数组后一个位置查看有没有位置可以放，如果可以放就放进去如果不行继续往后探查空位置，直到往后找不到为止。

* 1. 二次探查

冲突发生时，在表的左右进行跳跃式探查。

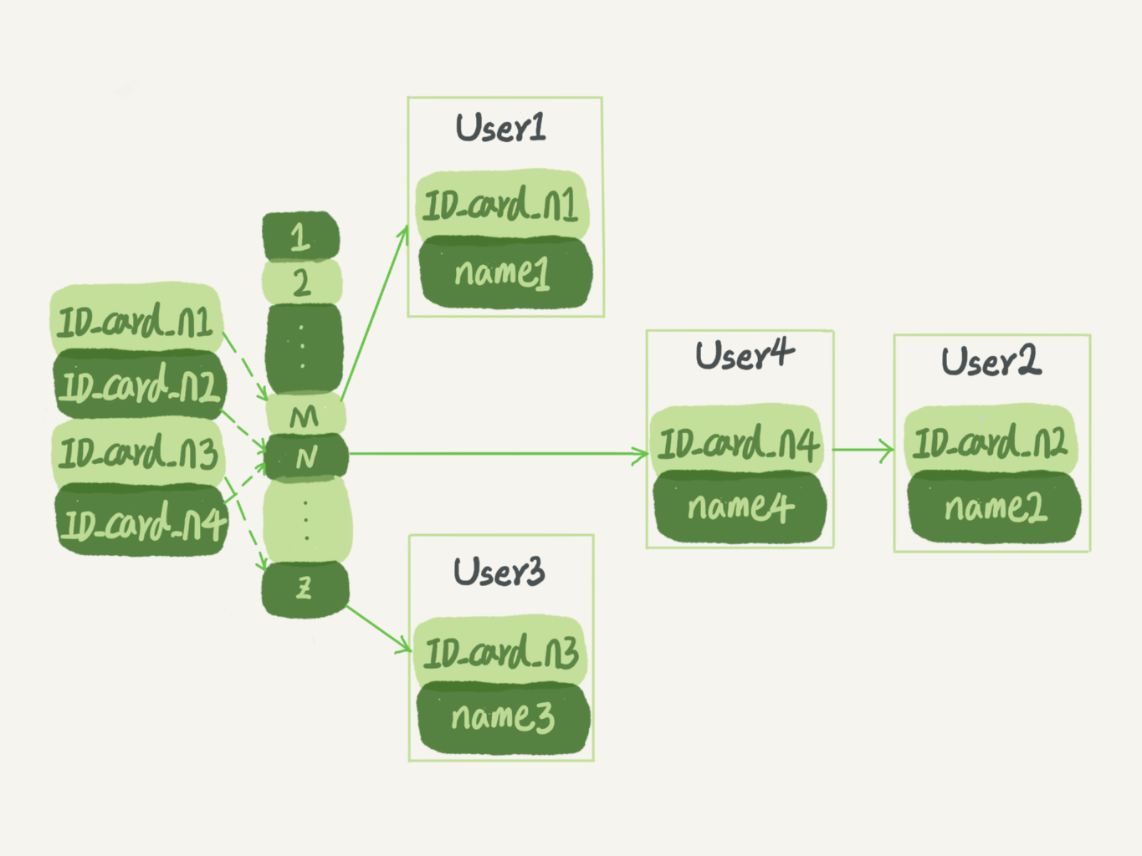
* 1. 伪随机探查

冲突发生时，以伪随机数发生器，（如i=(i+p)%m）进行探查，p为随机数。

3 再散列法

发生冲突时，再次应用散列法。直到找到不发生冲突位置。

维护一个身份证信息和姓名的表，需要根据身份证号查找对应的姓名，这个时候对应的哈希索引示意图如下所示：



哈希表作为索引的优点是做等值查询，非常快。

但是做范围查询会比较慢，因为需要全表扫描。

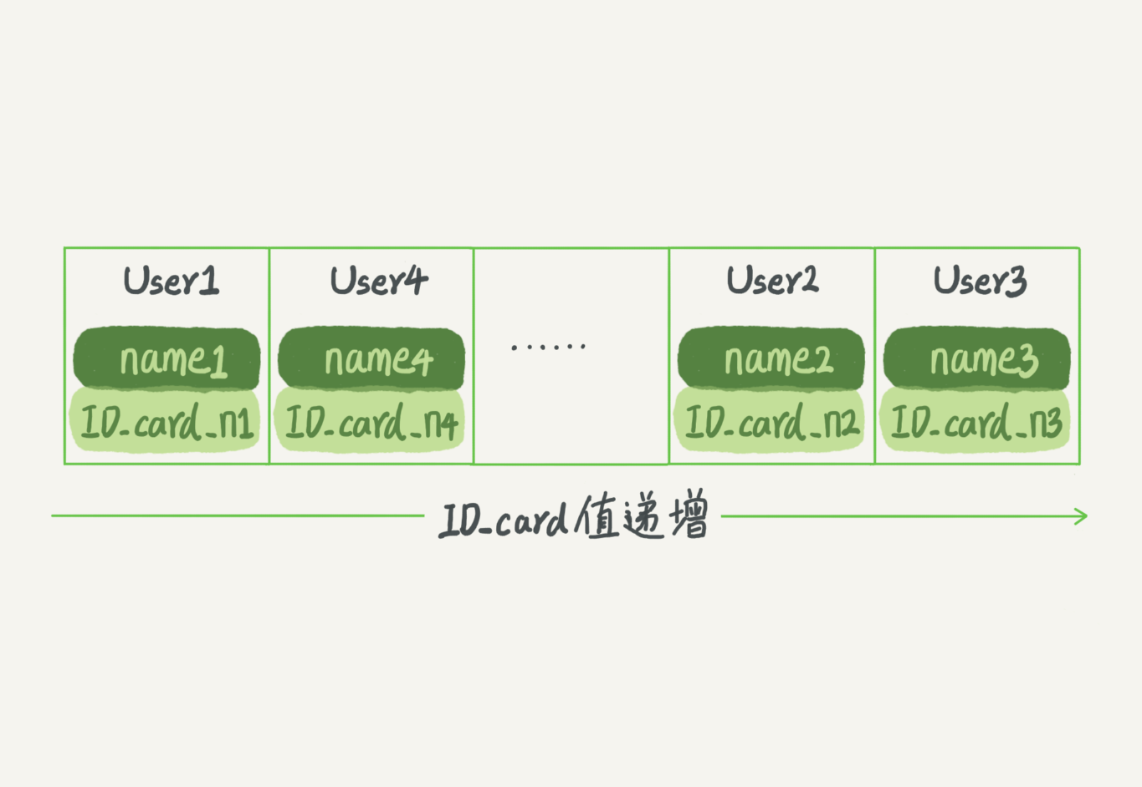
**如何使用hash索引？**

InnoDB和MyISAM都不支持hash索引。只有MEMORY和NDB引擎支持，使用的方式是在创建索引的时候，使用using hash，比如：create index index\_t using hash on test2(id);

### 4.1.2 有序数组

有序数组在等值查询和范围查询中的性能都是非常优秀的。

根据证查询姓名，示意图如下：



如果需要查询ID\_card\_n2对应名字。只需要使用二分法就可以快速得到，时间复杂度是O(log(N)).

显然范围查询的话

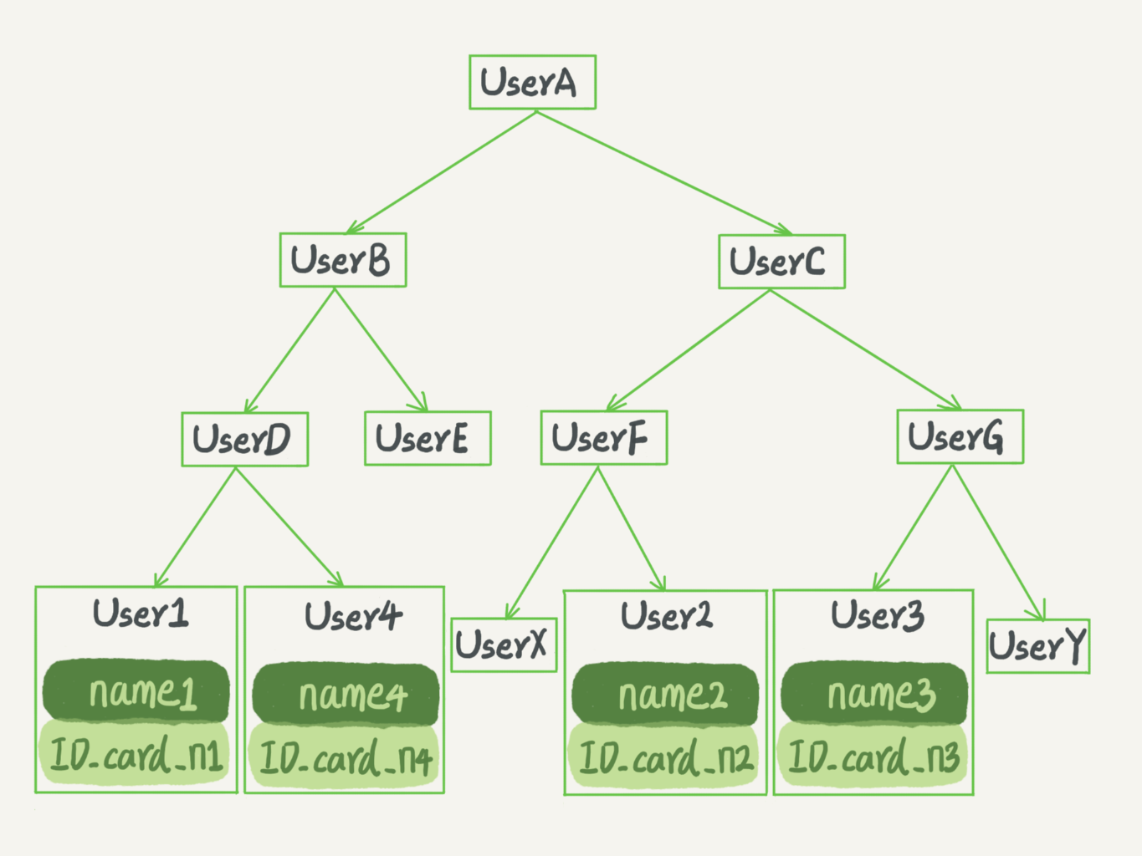
先用二分法找到ID\_card\_x(如果不存在ID\_card\_X,就找到大于ID\_card\_X的第一个User)，然后往后遍历，直到查到第一个大于ID\_card\_Y的身份证号，退出循环。

有序数组查询效率会比较高，但是插入记录的效率，成本太高。在数组中间插入一个记录就必须往后挪动所有记录，成本太高。

有序数组索引只适用于静态存储引擎。

### 4.1.2 二叉搜索树

二叉搜索树是一个特殊的二叉树，它保证了父节点的值比左孩子节点大，比右孩子节点值要小。应用于根据身份证查询名字例子。示意图如下：



图n 二叉搜索树示意图

如果要查找ID\_card\_n2的话，按照图中的搜索顺序是按照UserA->UserC->UserF->User2这个路径得到，时间复杂度是O(log(N))。

二叉树搜索效率最高，但是实际情况下，数据库存储却不使用二叉树，其原因是，索引不止存在内存中，还要写到磁盘上。

如果一颗100万节点的平衡二叉树，树高20.一次查询需要访问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10ms左右的寻址时间。也就是说，对于一个100万行表，如果使用二叉树存储，单独访问一个行可能需要20个10ms时间，查询会显得很慢。

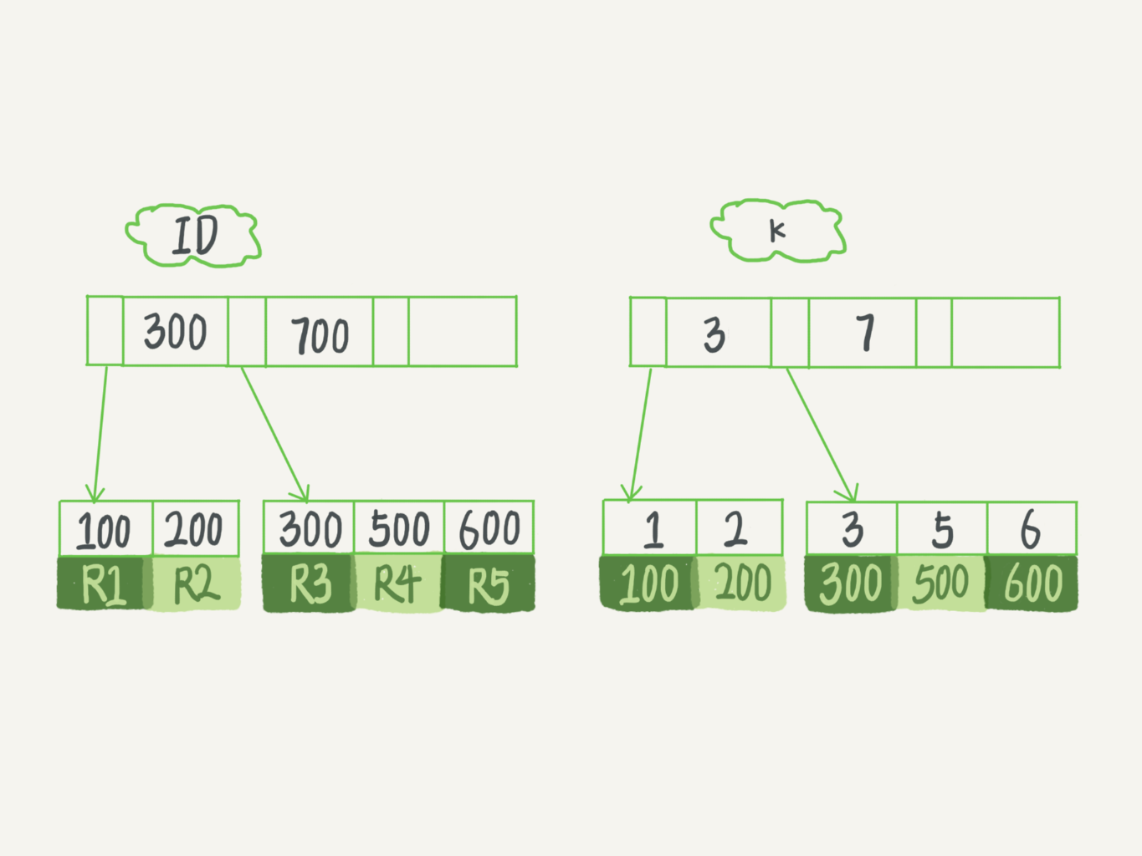
为了尽量少读磁盘，就必须让查询过程访问尽量少的数据块。那么，应该使用”N叉”树。这里，“N叉”树中的”N”取决于数据块的大小。

以InnoDB的一个整数字段索引为例，这个N差不多是1200。树高为4时，可以存储1200的3次方个值，达到17亿级别。酷哦率树根总是在内存中，一个10亿行表的整数字段索引，查找一个值最多需要访问3次磁盘。树的第二层也有很大概率存储在内存中，那么访问磁盘次数平均次数就更少了。

扩展：跳表、LSM树数据结构也被用于引擎设计中。

## 4.2 InnoDB索引模型

InnoDB中，表都是根据顺序以索引形式存放的，存储方式的表称为索引组织表。InnoDB使用B+树索引模型，所以数据都是存储到B+树中的。



**每一个索引在InnoDB中对应一颗B+树。**

**索引分为主键索引和非主键索引**

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

主键索引和普通索引的区别：

1. 如果语句是select \* from T where ID=500，即主键查询方式，则只需要搜索ID这颗B+树
2. 如果语句是select \* from T where k=5，即普通索引查询方式，则需要先搜索k索引树，得到ID值为500，再到ID搜索树搜索一次，这个过程称之为回表。

## 4.3 索引维护

B+树为了维护索引有序行，插入新值需要必要维护。

### 4.3.1 腾挪位置

以上图为例，插入新行ID为700，则只需要在R5记录后面插入一个新记录。如果需要插入的ID值为400，就比较麻烦了。需要逻辑上挪动后面的位置，空出位置。

### 4.3.2 页分裂

更糟的是，如果R5所在数据页已经满了，根据B+树算法。需要申请一个新的数据页，然后挪动部分数据进去。这个过程称之为页分裂，性能自然会受到影响。页分裂也会使空间利用率下降，整体空间利用率降低大约50%。

### 4.3.3 页合并

相邻两个页由于删除数据，利用率变低后，会将数据页做合并。合并过程，可以认为是分裂过程的逆过程。

### 4.3.4 自增主键

自增主键的插入数据模式，符合递增插入擦汗给你精，不涉及到挪动其他记录，也不会触发叶子节点的分裂。

主键长度越小，普通索引叶子节点就越小，普通索引占用空间也就越小。从性能和存储空间上考虑，自增主键往往是最为合理的选择。

业务字段直接做主键的场景？

1. 只有一个索引
2. 该索引必须是唯一索引

典型的KV场景。

## 4.4 索引优化手段

### 4.4.1 覆盖索引

在mysql数据库中执行下列语句

mysql> create table T (

ID int primary key,

k int NOT NULL DEFAULT 0,

s varchar(16) NOT NULL DEFAULT '',

index k(k))

engine=InnoDB;

insert into T values(100,1, 'aa'),(200,2,'bb'),(300,3,'cc'),(500,5,'ee'),(600,6,'ff'),(700,7,'gg');

可以得到如下两颗索引树

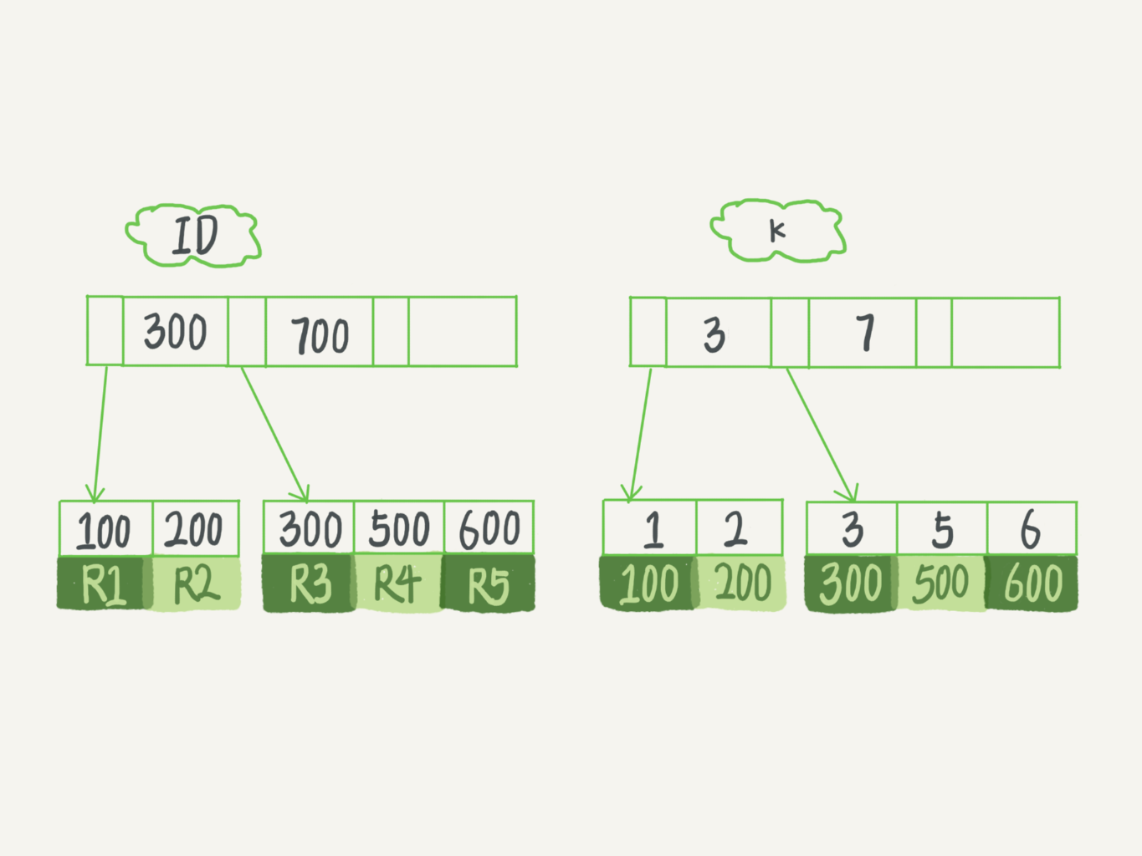


图4-N InnoDB索引树结构

在表T中执行 select \* from T where k between 3 and 5, 需要执行几次树搜索操作，扫描多少行？

这条范围查询语句执行流程是：

1. 在k索引树上找到k=3的记录，取得ID=300；
2. 再到ID索引数查到ID=300对应R3；
3. 在k索引树取下一个值k=5，取得ID=500；
4. 再回到ID索引数查到ID=500对应R4；
5. 在k索引树取下一个值k=6,不满足条件，循环结束。

**回到主键索引数搜索的过程，称之为回表。**

**可以看到，这个查询过程读了k索引树3条记录，回表了两次。**

若执行的语句是select ID from T where k between 3 and 5,这时只需要查ID值，而ID值已经在k索引数上了，因此可以直接查询结果，不需要回表查询。

什么是覆盖索引？

**索引覆盖了查询需要的内容，称之为覆盖索引。**

**覆盖索引可以减少树的搜索次数，显著提升查询性能，所以使用覆盖索引是一个常用的性能优化手段。**

### 4.4.2 最左前缀原则

B+树索引结构可以利用索引的”最左前缀”.来定位记录。

以(name,age)联合索引分析

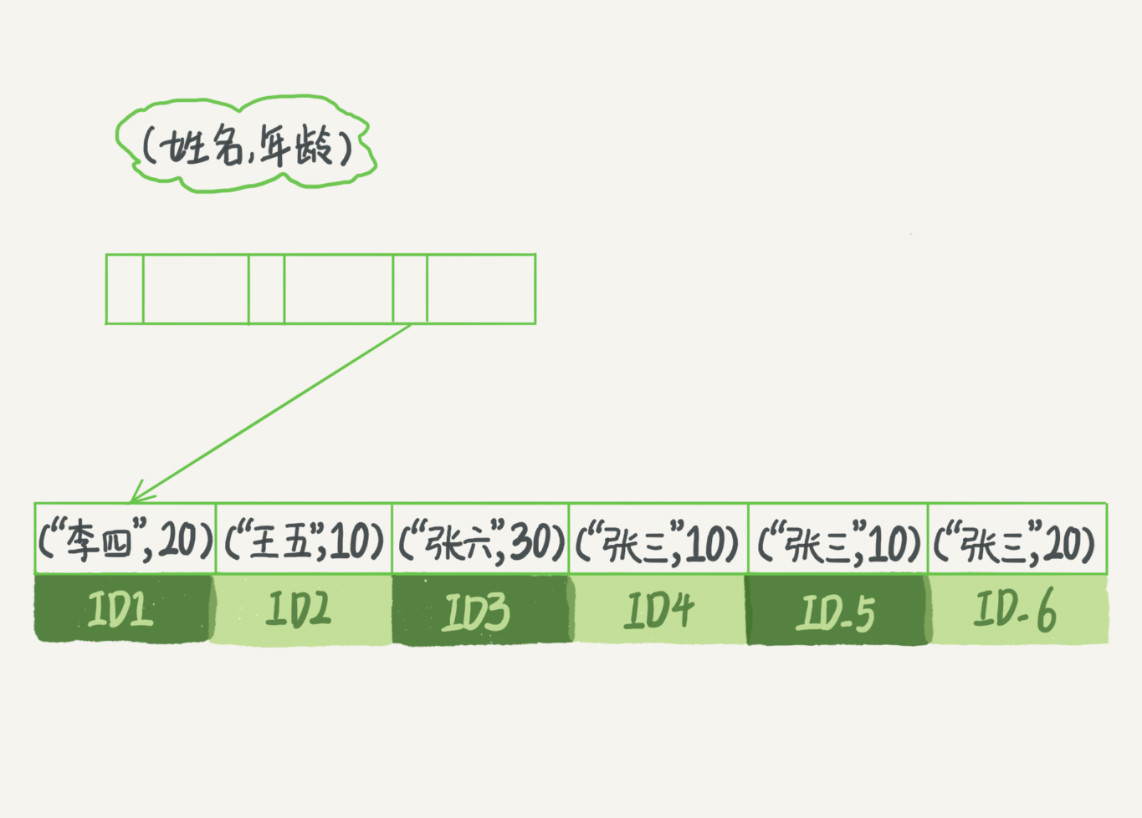


图4-N (name,age)索引示意图

只要满足索引的最左前缀，就可以利用索引来加速检索，这个最左前缀可以是联合索引的最左N个字段，也可以是字符串索引的最左M个字符。

最左前缀原则目的是为了减少索引的创建。

建立联合索引时，如何安排索引内的顺序？

第一原则，如果能够调整顺序，可以少维护一个索引，优先考虑。

第二原则，如果非要建立多个索引，以空间占用作为考虑。

### 4.4.3 索引下推

对查询条件中不满足索引字段进行判断，过滤不满足条件的记录，减少回表次数。（MySQL5.6）

思考题：

DBA 小吕在入职新公司的时候，就发现自己接手维护的库里面，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;

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？

表记录–a--|–b--|–c--|–d—

1 2 3 d

1 3 2 d

1 4 3 d

2 1 3 d

2 2 2 d

2 3 4 d

主键 a，b 的聚簇索引组织顺序相当于 order by a,b ，也就是先按 a 排序，再按 b 排序，c 无序。

索引 ca 的组织是先按 c 排序，再按 a 排序，同

时记录主键–c--|–a--|–主键部分b-- （

注意，这里不是 ab，而是只有 b）

2 1 3

2 2 2

3 1 2

3 1 4

3 2 1

4 2 3这个跟索引 c 的数据是一模一样的。

索引 cb 的组织是先按 c 排序，在按 b 排序，

同时记录主键–c--|–b--|–主键部分a—

（同上）

2 2 2

2 3 1

3 1 2

3 2 1

3 4 1

4 3 2所以，

结论是 ca 可以去掉，cb 需要保留。

# 五 MySQL全局锁和表锁

锁：合理的控制数据库资源的访问规则

根据加锁范围MySQL里面的锁大致可分为全局锁、表级锁和行锁三类。

## 5.1 全局锁

全局锁对整个数据库实例加锁。MySQL提供了一个加全局锁的方法，命令是Flush tables with read lock (FTWRL)

这个命令使整个库处于只读状态。之后其他线程的 数据更新语句(数据增删改)、数据定义语句(包括建表、修改表结构)和更新类事务提交语句。

全局锁使用场景是全库逻辑备份

数据备份中出现数据不一致的问题：

假设你现在要维护“极客时间”的购买系统，关注的是用户账户余额表和用户课程表。现在发起一个逻辑备份。假设备份期间，有一个用户，他购买了一门课程，业务逻辑里就要扣掉他的余额，然后往已购课程里面加上一门课。如果时间顺序上是先备份账户余额表 (u\_account)，然后用户购买，然后备份用户课程表 (u\_course)，会怎么样呢？你可以看一下这个图：

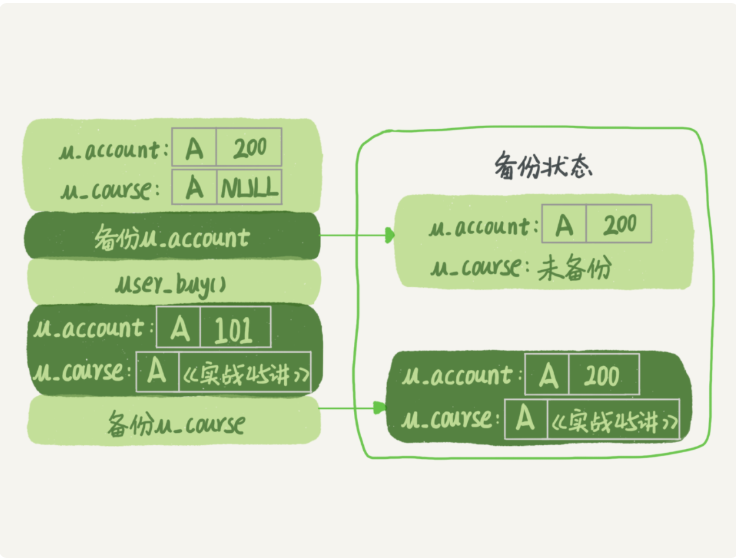


图 1 业务和备份状态图

可以看到，这个备份结果里，用户 A 的数据状态是“账户余额没扣，但是用户课程表里面已经多了一门课”。如果后面用这个备份来恢复数据的话，用户 A 就发现，自己赚了

导致这一问题的原因是备份过程中看到了部分旧数据，部分新数据导致，数据不一致。解决方法时取到一致性视图，或者加全局锁避免备份过程中出现更新。

加了全局锁会出现的问题:

1. 在主库上备份，备份期间不能执行更新，业务停摆。
2. 从库上备份，备份期间从库不能执行主库同步过来的binlog，导致主从延迟。

备份数据的方法

1. 使用官方自带的逻辑备份工具mysqldump。当mysqldump使用参数-single-transaction的时候，导数据之前就启动一个事务，确保拿到一致性视图。由于MVCC支持，这个过程数据可以正常更新。但是前提是引擎要支持这个隔离级别，MyISAM就不支持事务，如果备份中有更新，就总能取到最新数据，破坏了备份一致性。就只能使用FTWRL命令
2. FTWRL命令，让整个库只读
3. set global readonly=true,让全库进入只读状态，但是这个方法有两个缺点:

一是,有些系统中readonly被用来做其他逻辑，比如判断一个库是主库还是备库。修改global便理那个影响面更大。

二是,发生数据库异常时，FTWRL命令会释放全局锁，整个库回到正常更新状态。将整个库设置为readonly后，若发生异常，数据库会长时间处于不可读状态，风险较高。

## 表级锁

表级锁有两种:一种是表锁，一种是元数据锁(meta data lock ,MDL)

### 5.2.1 表锁

**表锁语法是lock table ... read/write。与FTWRL类似，可使用unlock tables主动释放锁，也可以在客户端断开时候自动释放锁。**

使用了lock tables语法后虽然会限制其他线程的读写，同时也会限制本线程的其他操作。

比如，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。

### 元数据锁(MDL)

MDL不需要显示使用，在访问一个表的时候会自动加上，MDL作用是，保证读写正确性。例如，如果一个查询正在遍历一个表中的数据，而执行期间另一个线程对这个表结构做变更，删了一列，那么查询线程拿到的结果跟表结构对不上，肯定是不行的。

MySQL5.5版本引入MDL，当对一个表做增删改查，加MDL读锁；当要对表结果变更操作时，加MDL写锁。

读锁之间不互斥，因此可以多个线程同时对一张表增删改查。

读写锁、写锁之间是互斥的，用来保证表结构操作安全性。

事务中的MDL锁在语句执行开始时申请，语句结束后并不马上释放，而会等到整个事务提交后再释放。

如何安全的给小表加字段？

1. 优先解决长事务。在MySQL的information\_schema库的innodb\_trx表中，可以查询到当前执行中的事务，如果要做DDL变更刚好有长事务，要考虑暂停DDL，或者kill长事务。
2. 在alter table语句设置等待时间，如果指定等待时间能够拿到MDL写锁最好，拿不到也不要阻塞后面的业务语句，先放弃。之后开发人员或者 DBA 再通过重试命令重复这个过程。

已经合并了 AliSQL 的这个功能，所以这两个开源分支目前都支持 DDL NOWAIT/WAIT n 这个语法。

ALTER TABLE tbl\_name NOWAIT add column ...

ALTER TABLE tbl\_name WAIT N add column ...

最后，我给你留一个问题吧。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：

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。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；

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。

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。

# 六 行锁

MySQL行锁是由引擎层实现的，并不是所有数据库引擎都支持行锁，比如MyISAM引擎就不支持行锁。不支持行锁会导致并发度降低。

## 6.1 两阶段锁协议

两阶段锁协议: InnoDB事务，行所是需要时加上，执行完并不马上释放，而需要等到事务结束后才释放。

优化建议：事务中需要锁多行。把最可能造成锁冲突，最可能影响并发度的锁尽量往后放。

举个例子。假设你负责实现一个电影票在线交易业务，顾客 A 要在影院 B 购买电影票。我们简化一点，这个业务需要涉及到以下操作：

从顾客 A 账户余额中扣除电影票价；

给影院 B 的账户余额增加这张电影票价；

记录一条交易日志。

也就是说，要完成这个交易，我们需要 update 两条记录，并 insert 一条记录。当然，为了保证交易的原子性，

我们要把这三个操作放在一个事务中。那么，你会怎样安排这三个语句在事务中的顺序呢？试想如果同时有另外一个顾客 C 要在影院 B 买票，那么这两个事务冲突的部分就是语句 2 了。因为它们要更新同一个影院账户的余额，需要修改同一行数据。根据两阶段锁协议，不论你怎样安排语句顺序，所有的操作需要的行锁都是在事务提交的时候才释放的。

所以，如果你把语句 2 安排在最后，

比如按照 3、1、2 这样的顺序，那么影院账户余额这一行的锁时间就最少。这就最大程度地减少了事务之间的锁等待，提升了并发度。

## 6.2 死锁和死锁检测

死锁： 数据库中两个事务或者多个事务在保持某条记录锁的同时，又去申请其他记录锁，造成双方都得不到锁而无线等待的情况。



事务 A 在等待事务 B 释放 id=2 的行锁，而事务 B 在等待事务 A 释放 id=1 的行锁。 事务 A 和事务 B 在互相等待对方的资源释放，就是进入了死锁状态

出现死锁，两种解决方法：

1. 设置死锁等待超市时间。这个超时时间可以通过参数innodb\_lock\_wait\_timeout设置。InnoDB默认值是50s，对于线上系统来说，往往无法接受。设置时间太小，容易误伤执行时间长的sql。一般不推荐使用。
2. 设置死锁检测，发现死锁后，主动回滚死锁链条的某个事物，让其他事务可以继续执行

。设置参数innodb\_deadlock\_detect为on，表示开启死锁检测。死锁检测在并发量大时，会占用大量CPU资源，一种解决方法是关闭死锁检测，但是可能会出现大量超时，业务有损。另一种方法是，在数据库源码层面或者中间件层面，对于同行更新，进入引擎前排队。在设计层面上，可以把一行改成多行，减少冲突发生的可能性。

最后，我给你留下一个问题吧。如果你要删除一个表里面的前 10000 行数据，有以下三种方法可以做到：第一种，直接执行 delete from T limit 10000;第二种，在一个连接中循环执行 20 次 delete from T limit 500;第三种，在 20 个连接中同时执行 delete from T limit 500。你会选择哪一种方法呢？为什么呢？

第一种方式（即：直接执行 delete from T limit 10000）里面，单个语句占用时间长，锁的时间也比较长；而且大事务还会导致主从延迟。

第三种方式（即：在 20 个连接中同时执行 delete from T limit 500），会人为造成锁冲突。

第二种方式是相对较好的

# 七 事务隔离与readview

## 7.1 readview

read-view:在可重复读隔离级别，事务T启动时会创建一个视图read-view，之后事务T执行期间，即使其他事务修改了数据，事务T看到的数据仍然和启动时看到的一样。在读提交级别上，事务T读取数据时每次都会新生成一个视图。

## 7.2 start transaction with consistent snapshot与begin/start transaction

两者区别：

1. 第一种方式，一致性视图实在执行start transaction with consistent snapshot时创建的
2. 第二种方式，一致性视图实在执行第一个快照语句时创建的。

## 7.3 视图

MySQL中有两个视图概念：

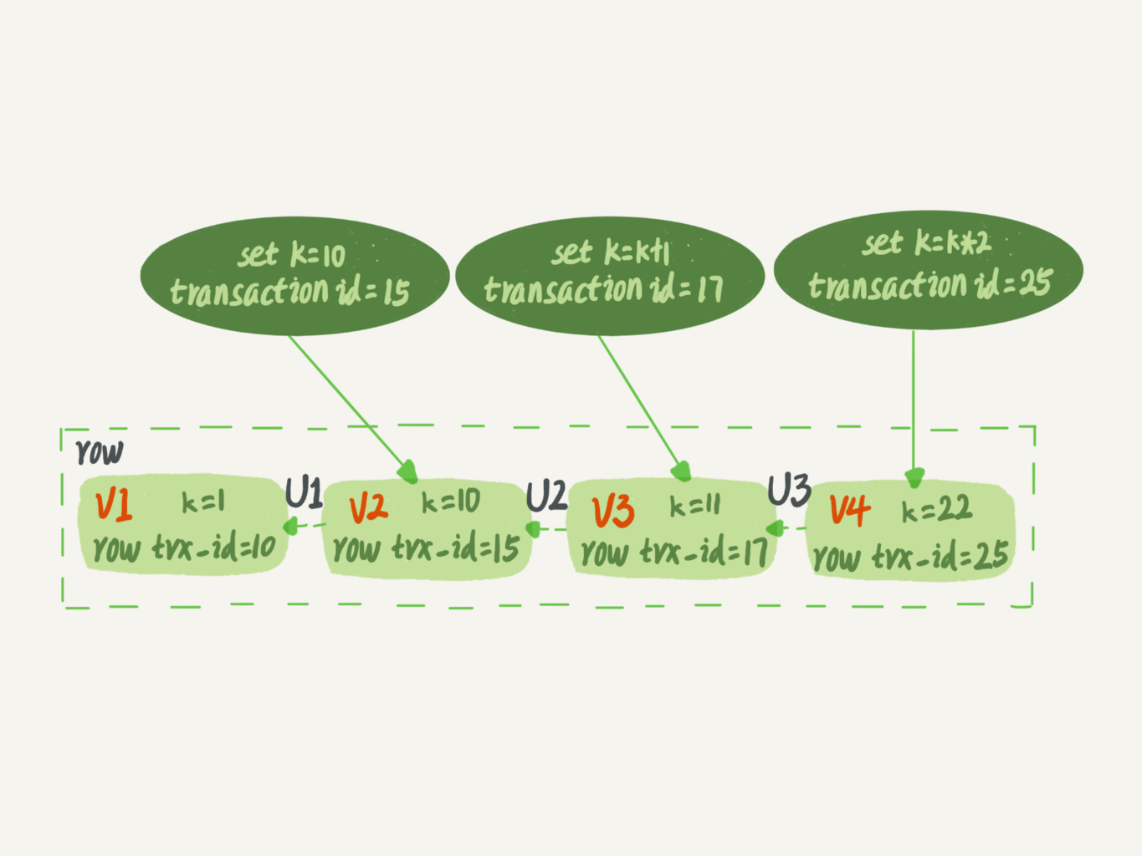
1. view，用查询语句定义的虚拟表，调用时候执行查询语句并生成结果。创建视图语法是create view…,查询方式和表一样。
2. InnoDB在实现MVCC时用到的一致性视图，即consistent read view，用于支持RC(read committed)和RR(Repeatable Read,可重复读)隔离级别的实现。

## 7.4 快照在MVCC实现方式

InnoDB里面每个事物都有唯一的事务ID，叫做transaction id。它是在事务开始时向InnoDB的事务系统申请的，是按照申请顺序严格递增的。

每行数据也都有多个版本。每次事务更新数据时，都会生成一个新的数据版本，并且把transaction id赋值给这个数据版本的事务ID，记为row trx\_id。同时，旧数据版本保留保留在undo log(回滚日志中).通过undo log和最新版本数据，就可以算出历史版本的数据。

一个记录被多个事务连续更新后的状态。



V1、V2、V3 并不是物理上真实存在的，而是每次需要的时候根据当前版本和 undo log 计算出来的。比如，需要 V2 的时候，就是通过 V4 依次执行 U3、U2 算出来。

## 7.5 数据版本可见性规则

下面是只有两行表的初始化语句。

mysql> CREATE TABLE `t` (

`id` int(11) NOT NULL,

`k` int(11) DEFAULT NULL,

PRIMARY KEY (`id`)

) ENGINE=InnoDB;

insert into t(id, k) values(1,1),(2,2);



事务A、B、C执行流程

我们不妨做如下假设：事务 A 开始前，系统里面只有一个活跃事务 ID 是 99；事务 A、B、C 的版本号分别是 100、101、102，且当前系统里只有这四个事务；三个事务开始前，(1,1）这一行数据的 row trx\_id 是 90。这样，事务 A 的视图数组就是[99,100], 事务 B 的视图数组是[99,100,101], 事务 C 的视图数组是[99,100,101,102]。

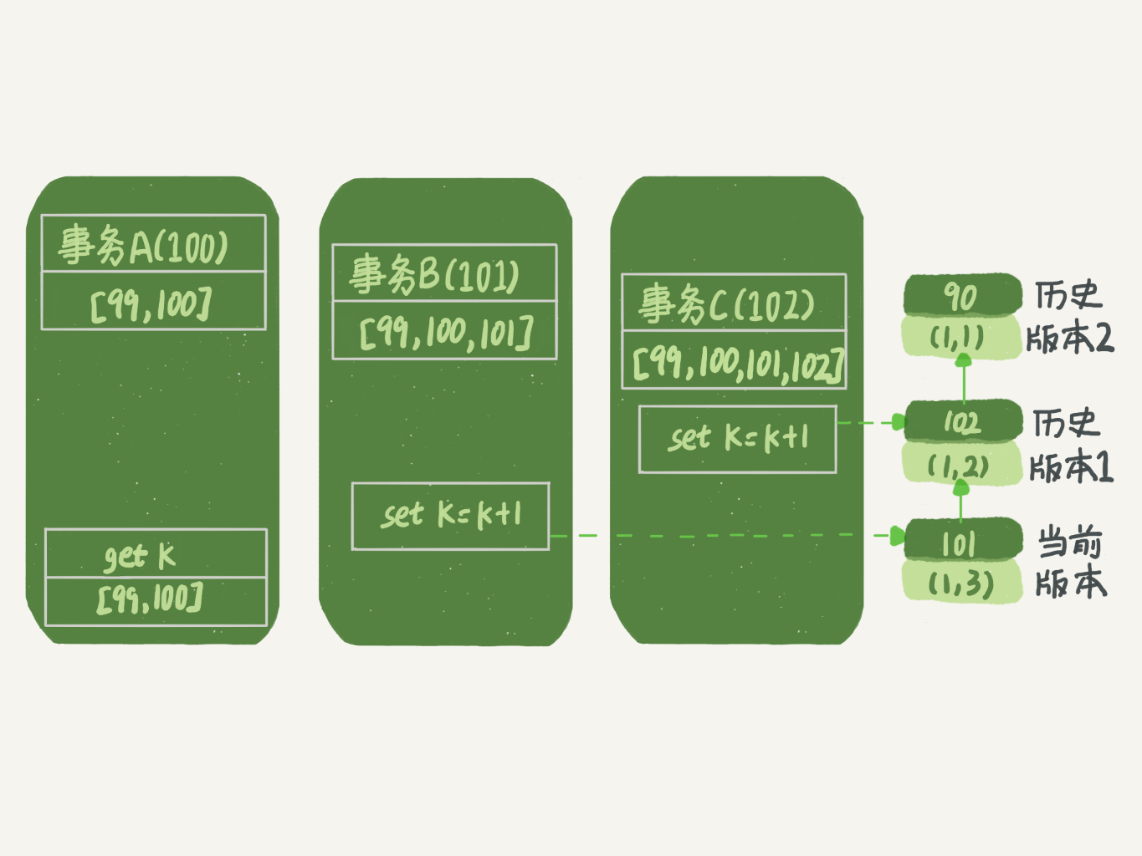


图 4 事务 A 查询数据逻辑图

### 7.5.1 数据版本可见性规则：

一个珊瑚橘版本，对于一个视图事务来说，除了自己的更新总是可见外，有三种情况

1. **版本未提交，不可见**
2. **版本已经提交，但是在视图创建后提交的，不可见。**
3. **版本已经提交，而且是在视图创建前提交的，可见。**

现在，我们用这个规则来判断图 4 中的查询结果，事务 A 的查询语句的视图数组是在事务 A 启动的时候生成的，这时候：(1,3) 还没提交，属于情况 1，不可见；(1,2) 虽然提交了，但是是在视图数组创建之后提交的，属于情况 2，不可见；(1,1) 是在视图数组创建之前提交的，可见。

### 7.5.2 数据版本更新可见性规则

事务 B 的 update 语句，如果按照一致性读，好像结果不对

图 5 中，事务 B 的视图数组是先生成的，之后事务 C 才提交，不是应该看不见 (1,2) 吗，怎么能算出 (1,3) 来？

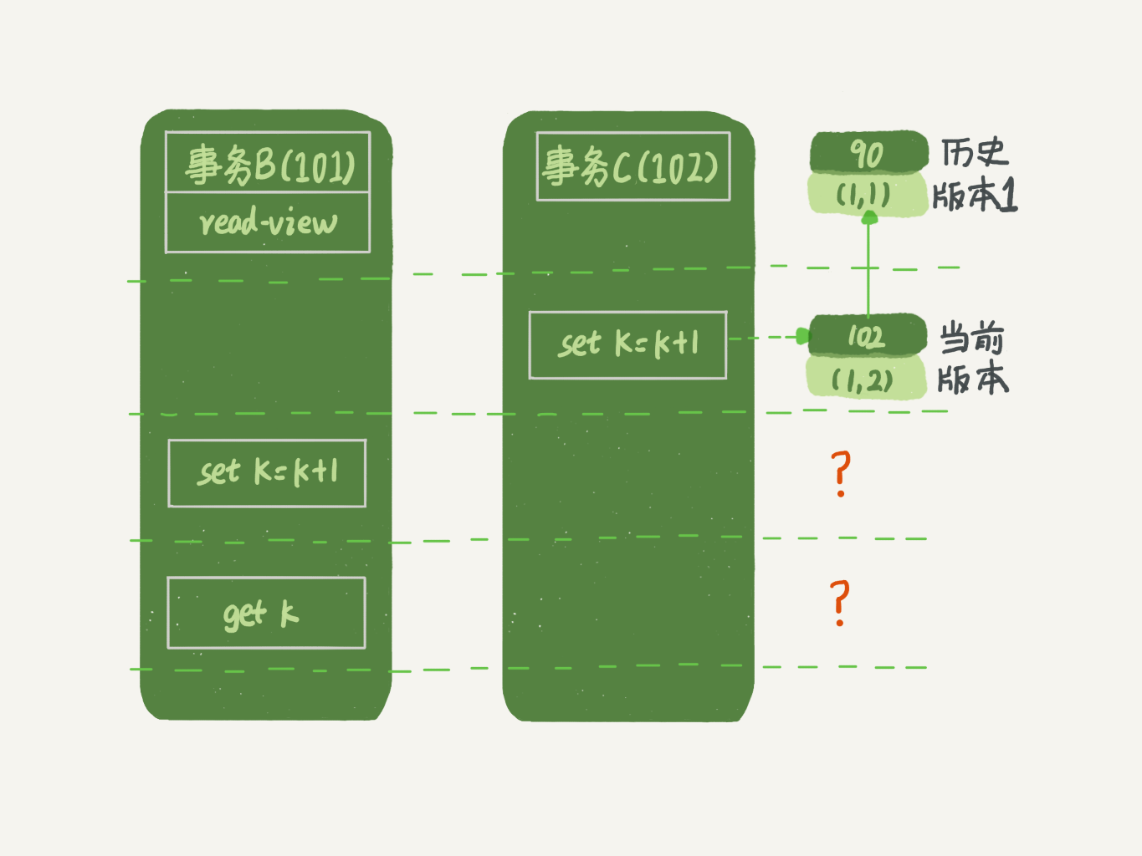


图 5 事务 B 更新逻辑图

这里就有，更新数据的可见性规则：

**更新数据都是先读后写的，而这个读，只能读当前的值，称为“当前读”（current read）。**

因此，在更新的时候，当前读拿到的数据是 (1,2)，更新后生成了新版本的数据 (1,3)，这个新版本的 row trx\_id 是 101。所以，在执行事务 B 查询语句的时候，一看自己的版本号是 101，最新数据的版本号也是 101，是自己的更新，可以直接使用，所以查询得到的 k 的值是 3。

### 7.5.3 MySQL中如何使用当前读

把事务 A 的查询语句 select \* from t where id=1 修改一下，加上 lock in share mode 或 for update，也都可以读到版本号是 101 的数据，返回的 k 的值是 3。下面这两个 select 语句，就是分别加了读锁（S 锁，共享锁）和写锁（X 锁，排他锁）。

mysql> select k from t where id=1 lock in share mode;

mysql> select k from t where id=1 for update;

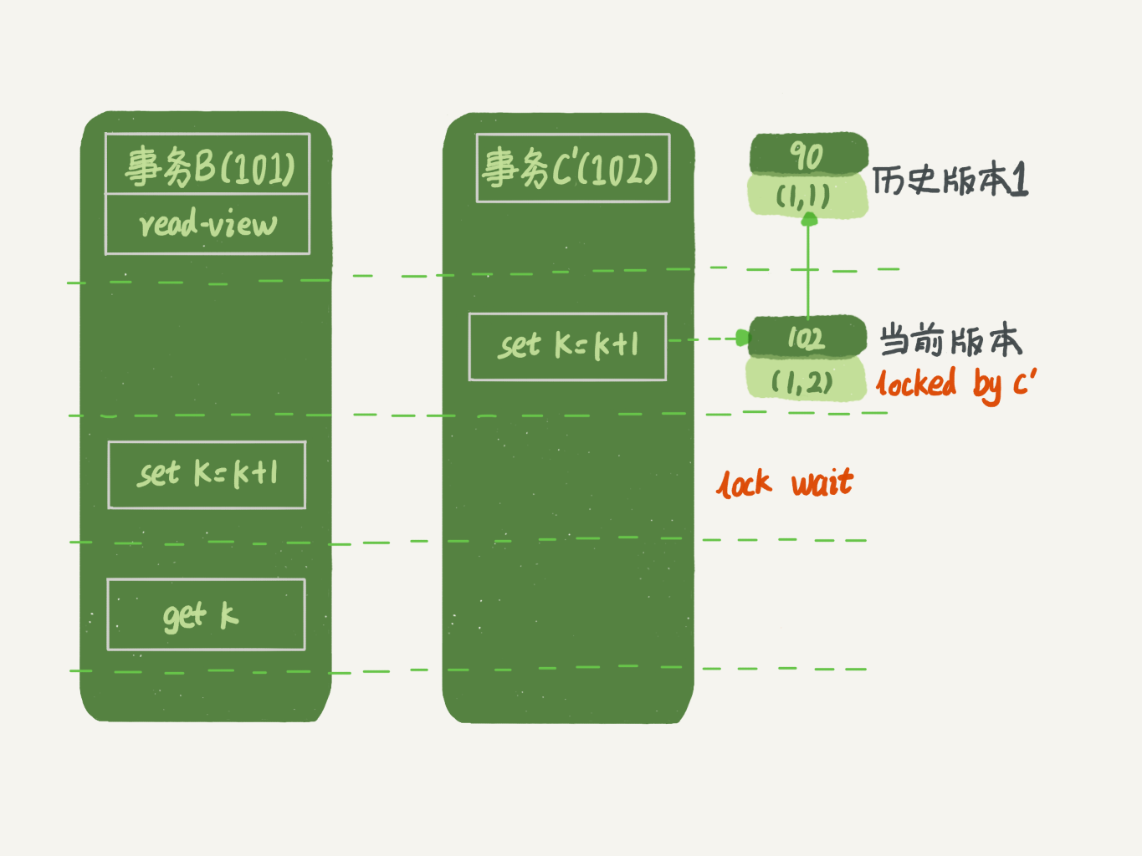
### 7.5.4 两阶段锁对更新数据的影响

事务 C 不是马上提交的，而是变成了下面的事务 C’



图 6 事务 A、B、C'的执行流程

事务C‘更新后并没有马上提交，它提交前，事务B更新语句发起。根据两阶段锁协议，事务C’没有提交，(1,2)版本上的写锁没有释放，事务B是当前读，必须要读最新版本，而且必须加锁，因此被锁住了，必须等到事务C‘释放锁，才能继续当前读。



事务 B 更新逻辑图（配合事务 C'）

### 7.5.5 读提交隔离级别数据可见性

“start transaction with consistent snapshot; ”的意思是从这个语句开始，创建一个持续整个事务的一致性快照。所以，在读提交隔离级别下，这个用法就没意义了

下面是读提交时的状态图，可以看到这两个查询语句的创建视图数组的时机发生了变化

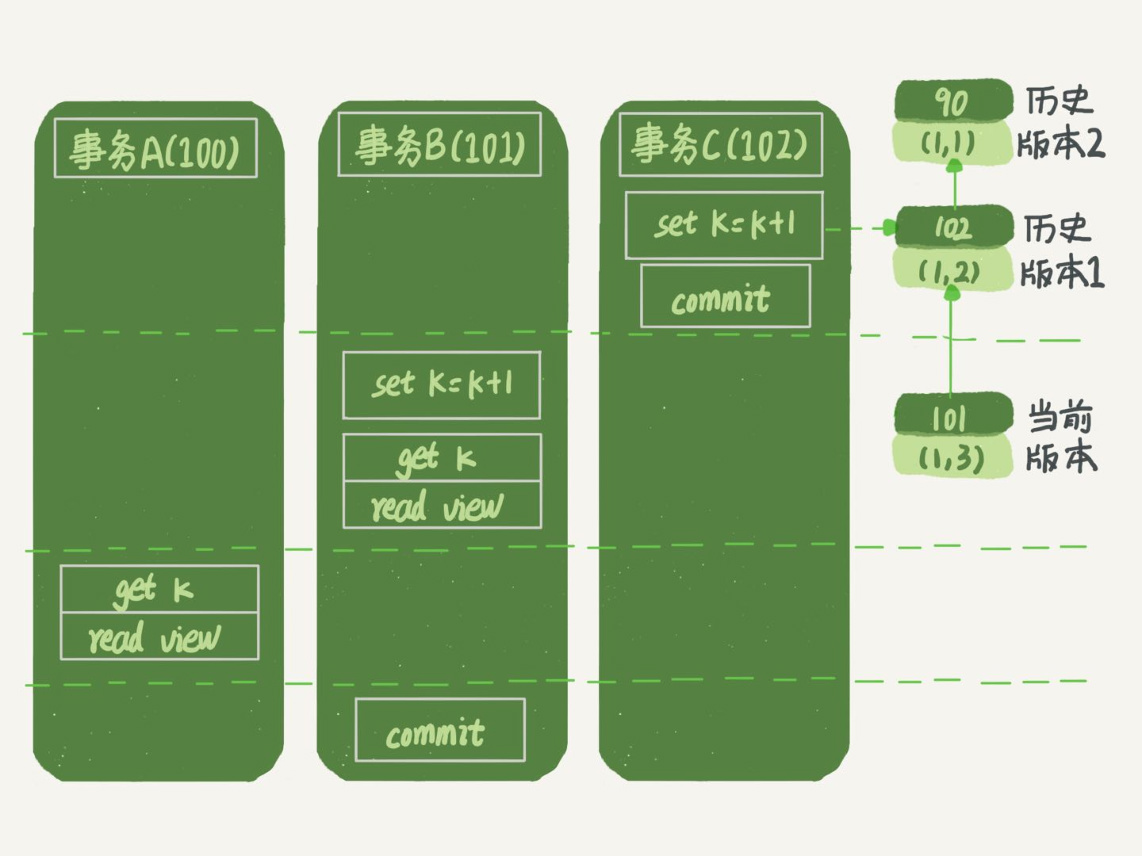


图 8 读提交隔离级别下的事务状态图

对于事务A的查询。

(1,3) 还没提交，属于情况 1，不可见；

(1,2) 提交了，属于情况 3，可见。

这时候事务 A 查询语句返回的是 k=2。

事务 B 查询结果 k=3。

下面的表结构和初始化语句作为试验环境，事务隔离级别是可重复读。现在，我要把所有“字段 c 和 id 值相等的行”的 c 值清零，但是却发现了一个“诡异”的、改不掉的情况。请你构造出这种情况，并说明其原理。

mysql> CREATE TABLE `t` (

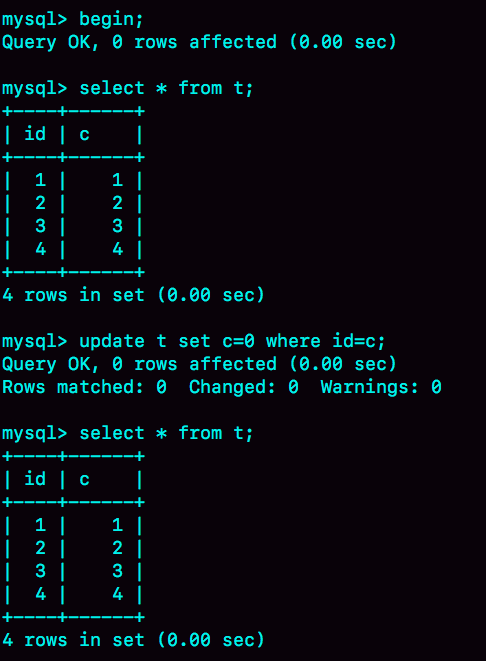
`id` int(11) NOT NULL,

`c` int(11) DEFAULT NULL,

PRIMARY KEY (`id`)

) ENGINE=InnoDB;

insert into t(id, c) values(1,1),(2,2),(3,3),(4,4);



答案：

一种场景



第二种：



思路主要是，事务A可重复读获取一致性读视图时，其他事务还没有更新这个数据。其他事务在事务A使用update语句更新前，更新了数据，让事务A没有更新到数据，这样表上数据没有事务A的row trx\_id。事务A就只看到它自己启动时表数据，以为更新数据了，但是查出来偏偏没有更新的假象。

# 八 changebuffer对唯一索引和普通索引选取的影响

## 8.1 changebuffer

**change buffer是一种特殊的数据结构，当要修改的页不在buffer pool中时，用来缓存对辅助页的修改。对索引页的操作可能是insert、update和delete操作。等到相关的索引页被读入buffer pool中后，才会使用change buffer中的内容对索引页进行修改(即merge操作)。**

**为何使用changebuffer？**

如果没有changebuffer，那么对数据库数据的更新、增加、删除，都需要从磁盘读取页数据，效率会严重下降。

Changebuffer相关原理。

对change buffer的merge可能需要好几个小时，如果被更新的辅助索引行比较多。在merge过程中，磁盘的I/O会增加，可能会引起其他查询的性能的降低。  
merge操作也可能发生在事务提交后。事实上，即使在实例重启后，还会可能发生merge操作。

在内存中，change buffer会占用buffer pool的空间；在物理磁盘上，change buffer是system tablespace的一部分，所以对索引的修改在数据库重启后仍然存在change buffer中。

change buffer包含的特性也叫作change buffering，包含insert buffering、delete buffering、purge buffering。

change buffer中数据类型和总量由参数innodb\_change\_buffering和innodb\_chagne\_buffer\_max\_size配置。查看change buffer中数据的信息，可以通过show engine innodb status查看。

普通索引和唯一索引区别：

1. 对于查询操作，普通索引在查到满足条件的记录后就需要往后继续查询直到碰到第一个不满足条件的记录，而唯一索引在查询到满足条件记录后，就会停止继续检索。
2. 对于更新操作，唯一索引使用不到changebuffer，因为当需要更新一个数据页时，唯一索引需要保证更新操作是否违反唯一性约束，所以必须将对应的数据页读入到内存当中才能判断。如果需要更新的目标页在内存中，唯一索引需要判断是否有冲突，在进行更新。而普通索引直接更新就可以了。如果需要更新的目标页不在内存中，唯一索引由于不能使用changebuffer，性能会比使用changebuffer进行更新的普通索引要差很多。

## 8.2 changebuffer使用场景：

适用于写多读少的业务，页面在写完以后马上被访问到的概率比较小，此时 change buffer 的使用效果最好。这种业务模型常见的就是账单类、日志类的系统。

反之一个业务的更新模式是写入之后马上会做查询，那么即使满足了条件，将更新先记录在 change buffer，但之后由于马上要访问这个数据页，会立即触发 merge 过程。这样随机访问 IO 的次数不会减少，反而增加了 change buffer 的维护代价。所以，对于这种业务模式来说，change buffer 反而起到了副作用。

## 8.3 change buffer搭配redo log

现在，我们要在表上执行这个插入语句：

mysql> insert into t(id,k) values(id1,k1),(id2,k2);

这里，我们假设当前 k 索引树的状态，查找到位置后，k1 所在的数据页在内存 (InnoDB buffer pool) 中，k2 所在的数据页不在内存中。如图 2 所示是带 change buffer 的更新状态图。

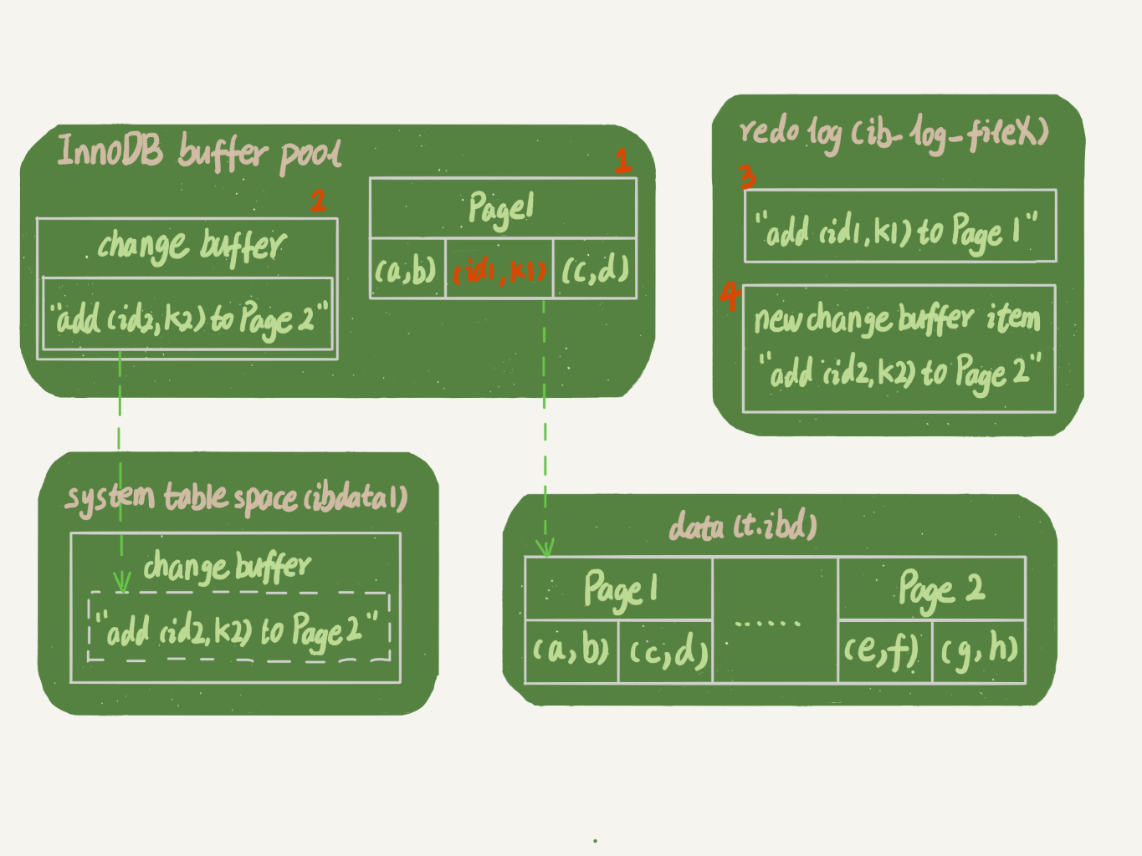


图 2 带 change buffer 的更新过程

涉及了四个部分：内存、redo log（ib\_log\_fileX）、 数据表空间（t.ibd）、系统表空间（ibdata1）。

这条更新语句做了如下的操作（按照图中的数字顺序）：Page 1 在内存中，直接更新内存；Page 2 没有在内存中，就在内存的 change buffer 区域，记录下“我要往 Page 2 插入一行”这个信息将上述两个动作记入 redo log 中（图中 3 和 4）。

**执行这条更新语句的成本很低，就是写了两处内存，然后写了一处磁盘（两次操作合在一起写了一次磁盘），而且还是顺序写的。**

读请求，要怎么处理呢？

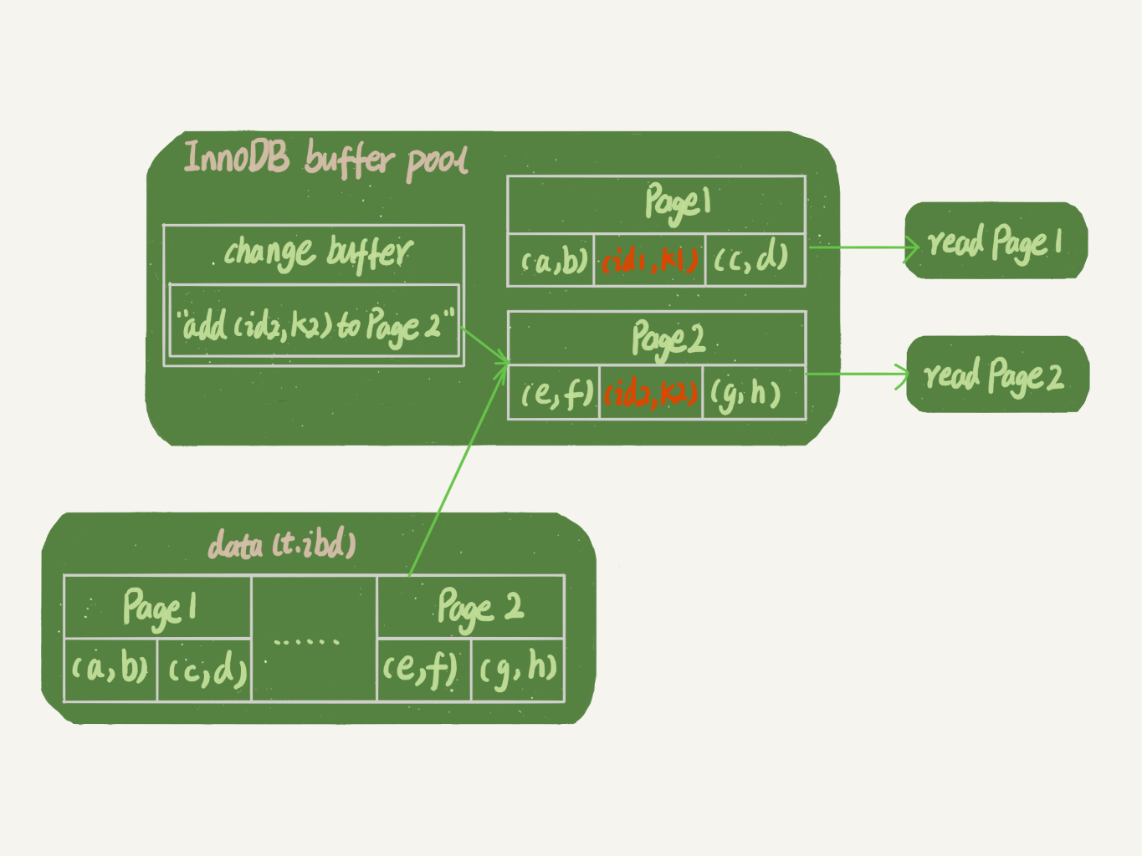


图 3 带 change buffer 的读过程

从图中可以看到：

1. 读 Page 1 的时候，直接从内存返回。
2. 要读 Page 2 的时候，需要把 Page 2 从磁盘读入内存中，然后应用 change buffer 里面的操作日志，生成一个正确的版本并返回结果。

可以看到，直到需要读 Page 2 的时候，这个数据页才会被读入内存。所以，如果要简单地对比这两个机制在提升更新性能上的收益的话，**redo log 主要节省的是随机写磁盘的 IO 消耗（转成顺序写），而 change buffer 主要节省的则是随机读磁盘的 IO 消耗。**

**由于唯一索引用不上 change buffer 的优化机制，因此如果业务可以接受，从性能角度出发我建议你优先考虑非唯一索引。**

## 8.4 思考题

通过图 2 你可以看到，change buffer 一开始是写内存的，那么如果这个时候机器掉电重启，会不会导致 change buffer 丢失呢？change buffer 丢失可不是小事儿，再从磁盘读入数据可就没有了 merge 过程，就等于是数据丢失了。会不会出现这种情况呢？

虽然是只更新内存，但是在事务提交的时候，我们把 change buffer 的操作也记录到 redo log 里了，所以崩溃恢复的时候，change buffer 也能找回来。

在评论区有同学问到，merge 的过程是否会把数据直接写回磁盘，

这里，我再为你分析一下。merge 的执行流程是这样的：

1. 从磁盘读入数据页到内存（老版本的数据页）；
2. 从 change buffer 里找出这个数据页的 change buffer 记录 (可能有多个），依次应用，得到新版数据页；
3. 写 redo log。这个 redo log 包含了数据的变更和 change buffer 的变更。到这里 merge 过程就结束了。

这时候，数据页和内存中 change buffer 对应的磁盘位置都还没有修改，属于脏页，之后各自刷回自己的物理数据，就是另外一个过程了。

# 九 MySQL如何选择索引

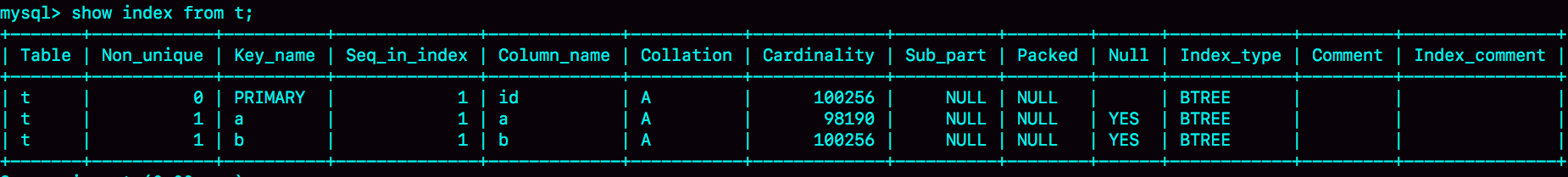
## 9.1 索引扫描行数判断

Mysql执行语句时并不能直到满足条件的语句有多少条，只能通过统计信息来估算记录条数。

索引上不同值越多，区分度越好。

索引上不同值个数，称为基数，也就是说基数越大，区分度越好。

使用show index可以查看索引的基数。但实际上索引基数并不往往准确。



## 9.2 MySQL如何获取索引的基数。

InnoDB 默认会选择 N 个数据页，统计这些页面上的不同值，得到一个平均值，然后乘以这个索引的页面数，就得到了这个索引的基数。而数据表是会持续更新的，索引统计信息也不会固定不变。所以，当变更的数据行数超过 1/M 的时候，会自动触发重新做一次索引统计。

## 9.3 MySQL选择索引考虑的因素

索引扫描行数，使用普通索引回表的代价、是否使用临时表、是否排序等。综合计算得到选择索引的结果。

本质原因还是因为扫描行数估算不对，也就是索引统计信息不对。可以通过analyze table t 命令来重新统计索引信息。

在某些场合，比如

mysql> select \* from t where (a between 1 and 1000) and (b between 50000 and 100000) order by b limit 1;

如果使用索引 a 进行查询，那么就是扫描索引 a 的前 1000 个值，然后取到对应的 id，再到主键索引上去查出每一行，然后根据字段 b 来过滤。显然这样需要扫描 1000 行。

如果使用索引 b 进行查询，那么就是扫描索引 b 的最后 50001 个值，与上面的执行过程相同，也是需要回到主键索引上取值再判断，所以需要扫描 50001 行。

所以你一定会想，如果使用索引 a 的话，执行速度明显会快很多。

mysql> explain select \* from t where (a between 1 and 1000) and (b between 50000 and 100000) order by b limit 1;



可以看到，返回结果中 key 字段显示，这次优化器选择了索引 b，而 rows 字段显示需要扫描的行数是 50198。

从这个结果中，你可以得到两个结论：

1．扫描行数的估计值依然不准确；

2. 这个例子里 MySQL 又选错了索引。

3. 通过字段b排序的代价并没有MySQL想象的大。

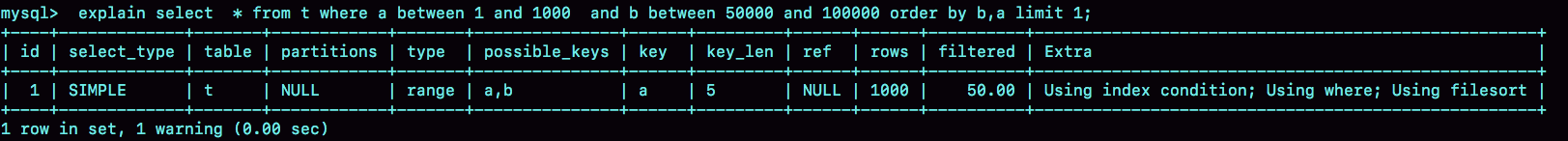
## 9.4 索引选择异常和处理

可以通过如下几种方式，强制或者诱导MySQL选择正确的索引

1. 采用 force index 强行选择一个索引。优点是，MySQL绝对不会选错索引，简单粗暴。

缺点是，由于会修改到代码，修改之后还要测试和发布，对于生产系统来说，这个过程不够敏捷。

1. 所以第二种方法就是，我们可以考虑修改语句，引导 MySQL 使用我们期望的索引。比如，在这个例子里，显然把“order by b limit 1” 改成 “order by b,a limit 1” ，语义的逻辑是相同的。我们来看看改之后的效果：



之前选择索引b是因为order by b，选择索引b就不用重新排序了。现在改成order by b，a。这样就使扫描行数为影响决策主要条件。

1. 在有些场景下，我们可以新建一个更合适的索引，来提供给优化器做选择，或删掉误用的索引。

## 9.5 课后习题

通过 session A 的配合，让 session B 删除数据后又重新插入了一遍数据，然后就发现 explain 结果中，rows 字段从 10001 变成 37000 多。

delete 语句删掉了所有的数据，然后再通过 call idata() 插入了 10 万行数据，看上去是覆盖了原来的 10 万行。但是，session A 开启了事务并没有提交，所以之前插入的 10 万行数据是不能删除的。这样，之前的数据每一行数据都有两个版本，旧版本是 delete 之前的数据，新版本是标记为 deleted 的数据。这样，索引 a 上的数据其实就有两份。

# 十 给字符串加索引

## 10.1 前缀索引

给字符串加索引时，可以选择字符串前几位作为索引，这样的索引称之为前缀索引。

前缀索引的优点是，可以减少长字符串字段的索引占用空间，在字符串前缀具有良好区分度情况下，可以起到减少索引占用的空间。

缺点是，使用了前缀索引就不能使用覆盖索引对查询进行优化，每次前缀索引上找到符合条件记录后，还需要到主键索引上进行回表查询。

10.1.1 前缀索引的选择方式

到底选择多长的前缀是和能接受的区分度损失度关联起来的，比如可以接受的损失比例为5%。那么需要在下面SQL语句统计前缀区分读时，找到总计值大于L\*95%的前缀。

mysql> select

count(distinct left(email,4)）as L4,

count(distinct left(email,5)）as L5,

count(distinct left(email,6)）as L6,

count(distinct left(email,7)）as L7,

from SUser;

## 10.2 倒序存储

在保存身份证等具有特殊含义字段时，由于身份证号的最后 6 位没有地址码这样的重复逻辑，所以最后这 6 位很可能就提供了足够的区分度。可以存储身份证的倒序，再使用前缀索引，相应的查询可以这样使用reverse函数。不要忘记使用 count(distinct) 方法去做个验证。

mysql> select field\_list from t where id\_card = reverse('input\_id\_card\_string');

## 10.3 使用hash字段

给表添加hash字段，在查询的时候使用hash字段和被hash字段两个条件进行查询，

由于hash字段可以用int 字段表示，只占用4个字节，具有很好的区分度，又占用很小空间，这种方法在保证效率的同时，也是很省索引占用空间的。

在表上再创建一个整数字段，来保存身份证的校验码，同时在这个字段上创建索引。

mysql> alter table t add id\_card\_crc int unsigned, add index(id\_card\_crc);

每次插入新记录的时候，都同时用 crc32() 这个函数得到校验码填到这个新字段。由于校验码可能存在冲突，也就是说两个不同的身份证号通过 crc32() 函数得到的结果可能是相同的，所以你的查询语句 where 部分要判断 id\_card 的值是否精确相同。

## 10.4 倒序存储和使用 hash 字段的异同点

它们的区别，主要体现在以下三个方面：

1. 从占用的额外空间来看，倒序存储方式在主键索引上，不会消耗额外的存储空间，而 hash 字段方法需要增加一个字段。
2. 在 CPU 消耗方面，倒序方式每次写和读的时候，都需要额外调用一次 reverse 函数，而 hash 字段的方式需要额外调用一次 crc32() 函数。reverse 函数额外消耗的 CPU 资源会更小些。
3. 从查询效率上看，使用 hash 字段方式的查询性能相对更稳定一些。因为 crc32 算出来的值虽然有冲突的概率，但是概率非常小，可以认为每次查询的平均扫描行数接近 1。而倒序存储方式毕竟还是用的前缀索引的方式，也就是说还是会增加扫描行数

## 10.5 小结

1. 直接创建完整索引，这样可能比较占用空间；

2. 创建前缀索引，节省空间，但会增加查询扫描次数，并且不能使用覆盖索引；

3. 倒序存储，再创建前缀索引，用于绕过字符串本身前缀的区分度不够的问题；

4. 创建 hash 字段索引，查询性能稳定，有额外的存储和计算消耗，跟第三种方式一样，都不支持范围扫描。

## 10.6 思考题

如果你在维护一个学校的学生信息数据库，学生登录名的统一格式是”学号 @gmail.com", 而学号的规则是：十五位的数字，其中前三位是所在城市编号、第四到第六位是学校编号、第七位到第十位是入学年份、最后五位是顺序编号。系统登录的时候都需要学生输入登录名和密码，验证正确后才能继续使用系统。就只考虑登录验证这个行为的话，你会怎么设计这个登录名的索引呢？

由于这个学号的规则，无论是正向还是反向的前缀索引，重复度都比较高。因为维护的只是一个学校的，因此前面 6 位（其中，前三位是所在城市编号、第四到第六位是学校编号）其实是固定的，邮箱后缀都是 @gamil.com，因此可以只存入学年份加顺序编号，它们的长度是 9 位。而其实在此基础上，可以用数字类型来存这 9 位数字。比如 201100001，这样只需要占 4 个字节。其实这个就是一种 hash，只是它用了最简单的转换规则：字符串转数字的规则，而刚好我们设定的这个背景，可以保证这个转换后结果的唯一性。

# 十一 脏页刷新

InnoDB 在处理更新语句的时候，只做了写日志这一个磁盘操作。这个日志叫作 redo log（重做日志）在更新内存写完 redo log 后，就返回给客户端，本次更新成功。

## 11.1 脏页和干净页

**当内存数据页跟磁盘数据页内容不一致的时候，我们称这个内存页为“脏页”。内存数据写入到磁盘后，内存和磁盘上的数据页的内容就一致了，称为“干净页”。**

## 11.2 脏页刷新时机

数据库在刷新脏页时会消耗CPU资源，可能会使数据库正常服务受到影响

下面有四种场景会刷新脏页到磁盘上：

1. redo log 写满了，这时候系统会停止所有更新操作，把checkpoint向前推进，redo log留出空间来继续写。
2. buffer pool内存不足。需要新的内存页的时候要淘汰一些数据页，留出内存给别的数据页使用。如果是脏页，就要先将脏页写到磁盘。为什么不直接淘汰内存页，下次需要请求时，从磁盘读入数据页，然后拿redo log应用？从性能角度来看，如果先将脏页写到磁盘，下次需要读入内存，肯定是正确结果，效率更高。
3. 系统空闲的时候，刷新脏页
4. MySQL正常关闭时，刷新脏页。

## 11.3 脏页刷新控制

尽量避免redo log写满，会使更新数跌为0

刷新脏页的恶劣情况：

1. 一个查询要淘汰的脏页个数太多，会导致查询的响应时间明显变长
2. 日志写满，更新全部堵住，写性能跌为 0，这种情况对敏感业务来说，是不能接受的。

控制策略：

1. 告诉InnoDB所在主机IO能力，使MySQL根据主机能力刷新脏页，使用innodb\_io\_capacity 这个参数，具体如下

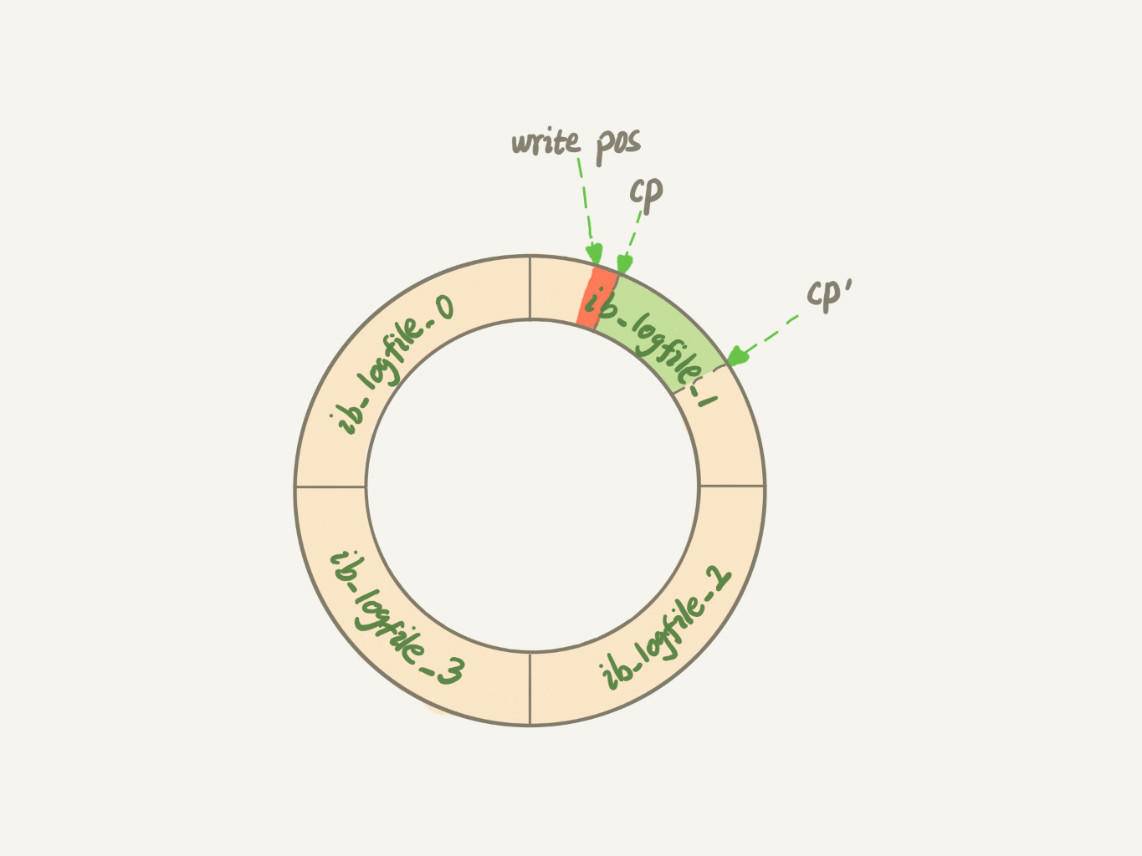
fio -filename=$filename -direct=1 -iodepth 1 -thread -rw=randrw -ioengine=psync -bs=16k -size=500M -numjobs=10 -runtime=10 -group\_reporting -name=mytest

1. MySQL会根据脏页比例，以及redo log写盘速度来控制刷脏页速度。
2. 脏页连坐，顾名思义，当要刷新一个脏页时，如果他的同桌也是脏页，也带着一起刷掉。

可以通过innodb\_flush\_neighbors，对于机械硬盘来说，会减少很多随机IO。

## 11.4 思考题

每次事务提交都要写 redo log，如果设置太小，很快就会被写满，也就是下面这个图的状态，这个“环”将很快被写满，write pos 一直追着 CP。



这时候系统不得不停止所有更新，去推进 checkpoint。这时，你看到的现象就是磁盘压力很小，但是数据库出现间歇性的性能下跌。

# 十二 数据库表空间删除

## 12.1 表文件存放地址

表数据既可以存在共享表空间里，也可以是单独的文件。这个行为是由参数 innodb\_file\_per\_table 控制的：

1. OFF 表示的是，表的数据放在系统共享表空间，也就是跟数据字典放在一起；
2. ON 表示的是，每个 InnoDB 表数据存储在一个以 .ibd 为后缀的文件中。

从 MySQL 5.6.6 版本开始，它的默认值就是 ON 了。

建议不论使用 MySQL 的哪个版本，都将这个值设置为 ON。因为，一个表单独存储为一个文件更容易管理，drop table 命令，系统就会直接删除这个文件。而如果是放在共享表空间中，即使表删掉了，空间也是不会回收的。

## 12.2 数据删除流程

如果我们删掉了一个数据页上的所有记录，会怎么样？答案是，整个数据页就可以被复用了。

记录复用和数据页复用：

1. 记录复用是复用索引页上的部分空洞，必须满足B+树性质
2. 数据页复用的范围会更加大
3. 用 delete 命令把整个表的数据删除，所有的数据页都会被标记为可复用。但是磁盘上，文件不会变小。
4. 插入数据也会造成页分裂，导致部分数据页空洞。

## 12.3 消除页空洞

### 12.3.1 重建表

新建一个与表 A 结构相同的表 B，然后按照主键 ID 递增的顺序，把数据一行一行地从表 A 里读出来再插入到表 B 中。由于表 B 是新建的表，所以表 A 主键索引上的空洞，在表 B 中就都不存在了。

如果我们把表 B 作为临时表，数据从表 A 导入表 B 的操作完成后，用表 B 替换 A，从效果上看，就起到了收缩表 A 空间的作用。

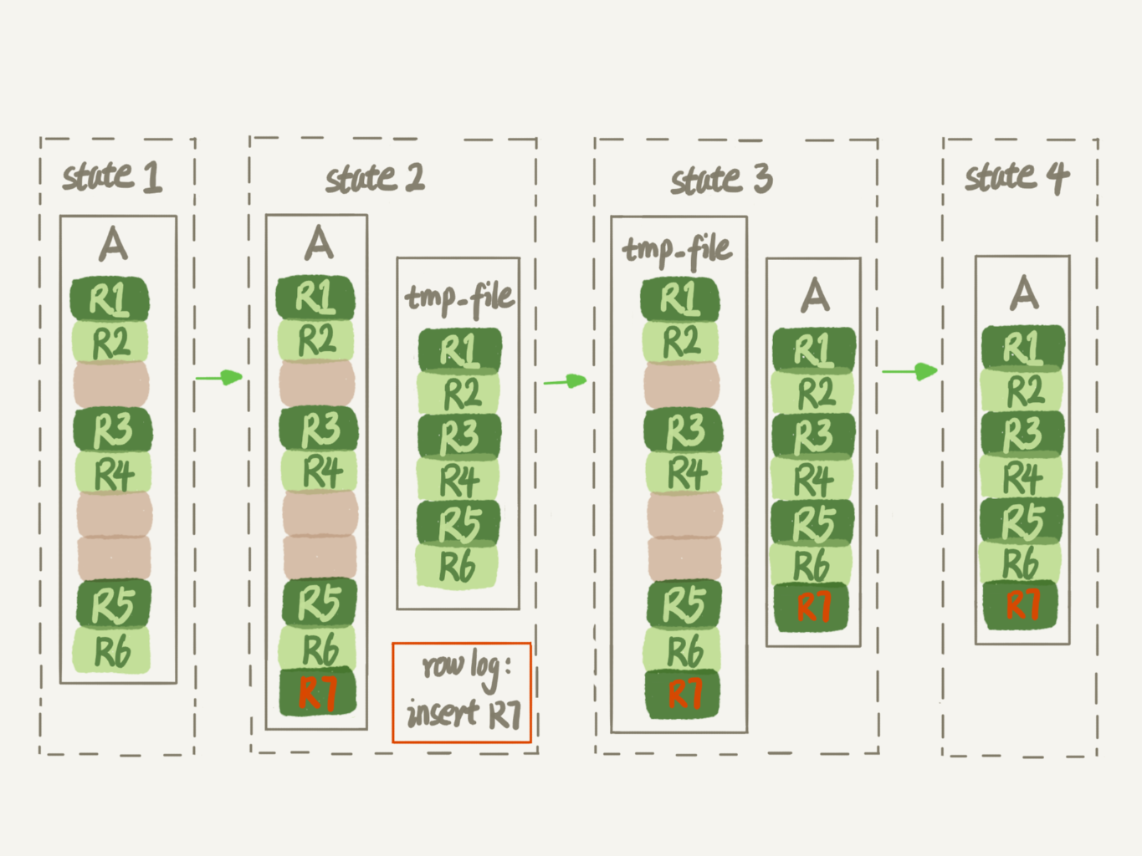
通过**alter table A engine=InnoDB消除空洞**

可以使用 **alter table A engine=InnoDB** 命令来重建表。在 MySQL 5.5 版本之前，这个命令的执行流程跟我们前面描述的差不多，区别只是这个临时表 B 不需要自己创建，MySQL 会自动完成**转存数据、交换表名、删除旧表**的操作。

缺点，整个 DDL 过程中，表 A 中不能有更新。也就是说，这个 DDL 不是 Online 的。

MySQL 5.6 版本开始引入的 Online DDL，对这个操作流程做了优化

1. 建立一个临时文件，扫描表 A 主键的所有数据页；
2. 用数据页中表 A 的记录生成 B+ 树，存储到临时文件中；
3. 生成临时文件的过程中，将所有对 A 的操作记录在一个日志文件（row log）中，对应的是图中 state2 的状态；
4. 临时文件生成后，将日志文件中的操作应用到临时文件，得到一个逻辑数据上与表 A 相同的数据文件，对应的就是图中 state3 的状态；
5. 用临时文件替换表 A 的数据文件。



Online DDL

Online DDL:

在重建表的过程中，允许对表 A 做增删改操作。

alter 语句在启动的时候需要获取 MDL 写锁，但是这个写锁在真正拷贝数据之前就退化成读锁了。MDL 读锁不会阻塞增删改操作。

上述的这些重建方法都会扫描原表数据和构建临时文件。对于很大的表来说，这个操作是很消耗 IO 和 CPU 资源的。因此，对于线上服务，需要小心地控制操作时间。如果想要比较安全的操作的话，推荐使用 GitHub 开源的 gh-ost 来做。

## 12.4 online和inplace区别

1. DDL 过程如果是 Online 的，就一定是 inplace 的；

2. 反过来未必，也就是说 inplace 的 DDL，有可能不是 Online 的。截止到 MySQL 8.0，添加全文索引（FULLTEXT index）和空间索引 (SPATIAL index) 就属于这种情况。

## 12.5 optimize table、analyze table 和 alter table 这三种方式重建表的区别。

1. 从 MySQL 5.6 版本开始，alter table t engine = InnoDB（也就是 recreate），主要过程是复制表索引到临时文件，复制过程中记录更新操作到row log中。复制完成将row log日志应用于临时文件，，得到与原表相同的数据文件，然后用临时文件替换原表数据文件。

2. analyze table t 其实不是重建表，只是对表的索引信息做重新统计，没有修改数据，这个过程中加了 MDL 读锁；

3. optimize table t 等于 recreate+analyze。

## 12.6 思考题

假设现在有人碰到了一个“想要收缩表空间，结果适得其反”的情况，看上去是这样的：

1. 一个表 t 文件大小为 1TB；
2. 对这个表执行 alter table t engine=InnoDB；
3. 发现执行完成后，空间不仅没变小，还稍微大了一点儿，比如变成了 1.01TB。

这张表本没有空洞，可能已经做了alter table t engine = InnoDB操作，再做一次的时候，其他事务引入了新的DML操作，反而增加了数据页空洞，使空间变大。