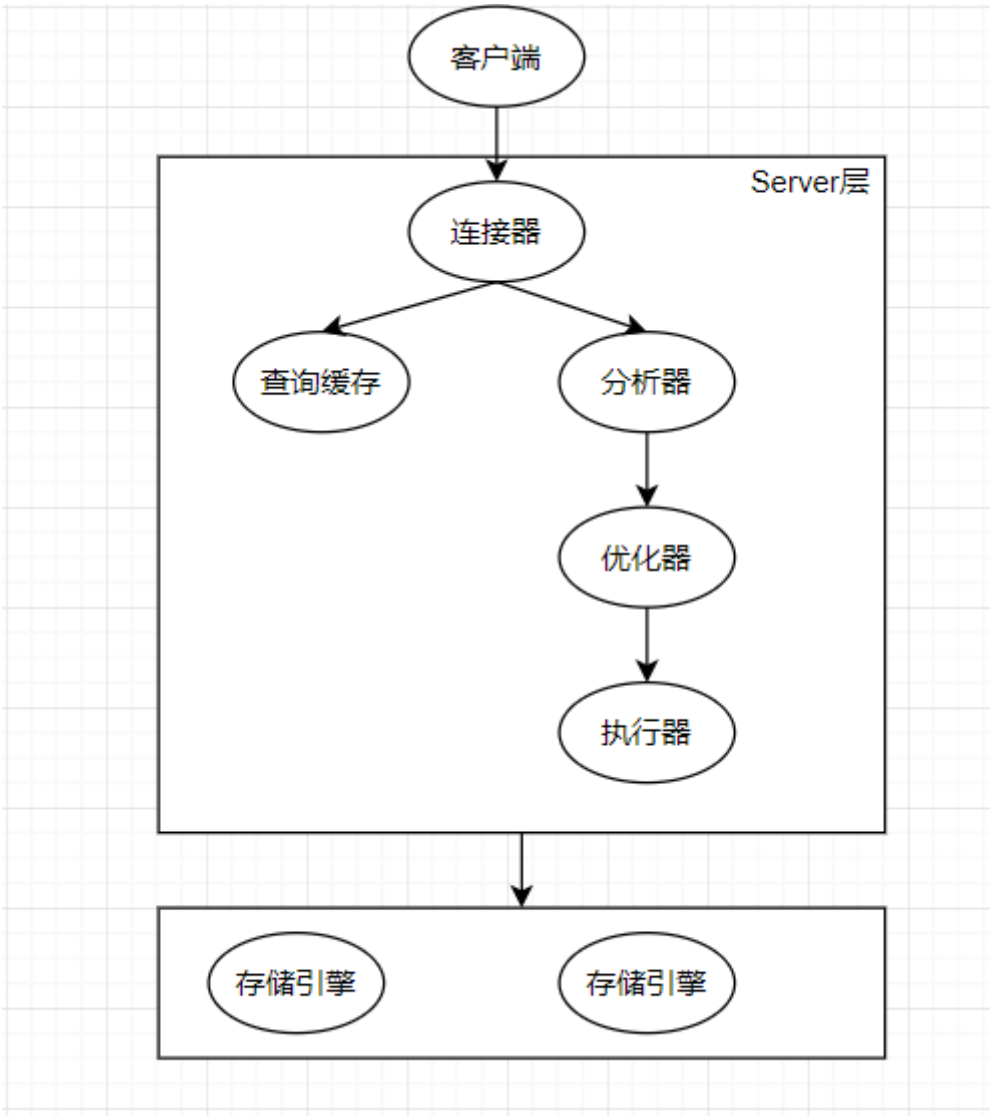
# 一、基础架构：一条SQL查询语句是如何执行的?

一、逻辑架构图




Server层涵盖mysql大多数核心功能，以及所有内置函数、跨存储引擎的功能（存储过程、触发器、视图）

- 连接器：管理连接，权限验证
- 查询缓存：命中缓存直接返回结果
- 分析器：词法分析、语法分析
- 优化器：生成执行计划，选择索引

- 执行器：调用引擎的接口，返回结果
- 存储引擎：存储数据，提供读写接口

二、连接器

1、连接器的职责

- 负责与客户端建立连接
- 权限验证
- 维持和管理连接

2、连接器的特性

- 为避免建立连接的开销，要尽量使用长连接
- 使用长连接，mysql内存占用增长，解决方案
  - 定期断开长连接
  - mysql5.7或以后版本，可在执行一次比较大的操作后，通过执行mysql_reset_connection来重新初始化连接资源。

3、客户端与连接器连接流程

- 客户端连接命令：mysql -h$ip  -P$port -u$user -p
- 客户端与连接器通过tcp协议通信
- 连接建立后，开始验证用户名和密码
  - 密码验证通过后，连接器去权限表读取用户权限，该权限信息在连接的生命周期内有效
- 连接建立后无后续动作，该连接将处于空闲状态。
  - show processlist命令查看连接

- wait_timeout参数，连接断开超时间隔，默认8小时

三、查询缓存

- mysql收到一个查询请求后会先去查询缓存判断之前是否执行过这条语句。执行过的语句会以key-value对的形式，被缓存在内存中。key为查询语句，value为查询结果。
- 若缓存中找不到，继续执行后续阶段。执行完成后，执行结果会被存入查询缓存中。
- 大多情况下不用查询缓存的原因
  - 查询缓存的失效很频繁，只要对一个表的更新，这个表上的所有查询缓存都会被清空。
  - 查询缓存比较适用于不经常变的表，如系统配置表
- 通过参数query_cache_type设为demand，默认不会使用查询缓存了，通过select SQL_CACHE * from T 里显式使用
- mysql8.0版本删除了查询缓存模块

四、分析器

- mysql需要知道你要做什么，因此需要对sql语句做解析
- 词法分析：识别语句中字符串分别是什么，代表着什么
- 语法分析：根据语法规则，判断sql语句是否符合mysql语法

五、优化器

- 优化器是在表中有多个索引的时候，决定使用哪个索引
- 在一个语句有多表关联（join）的时候，决定各个表的连接顺序。

例子：select * from t1 join t2 using(ID) wherr t1.c=10 and t2.d=20

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

两种执行方法的结果是一样的，但执行效率不同，而优化器的作用就是决定使用哪个方案

六、执行器

- 执行之前，权限验证
- 验证通过后，执行器根据表的引擎定义，去使用引擎提供的接口

示例：select * from t where id=10;如果id字段没有索引，执行器的执行流程：

- 调用InnoDB引擎接口取表的第一行，判断id值是否为10，若不是则跳过，若是则将这行存储在结果集中。
- 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
- 对于有索引的表，执行逻辑差不多，第一次调用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，这些接口都是引擎定义好的。

可以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。

七、问题及思考

按理说应该是先到分析器再到查询缓存，为什么连接器到查询缓存直接有一道线?

- 查询缓存存储的是key-value，key为查询语句，不需要进行分析就能判断有无

# 二、日志系统：一条SQL更新语句是如何执行的?

一、更新语句执行流程

- 连接器-->分析器-->清空查询缓存-->优化器--》执行器-->redo log（重做日志） 和 binlog（归档） 。

二、redo log

设计思想：酒店赊账

如果有人要赊账或者还账，有两种做法：

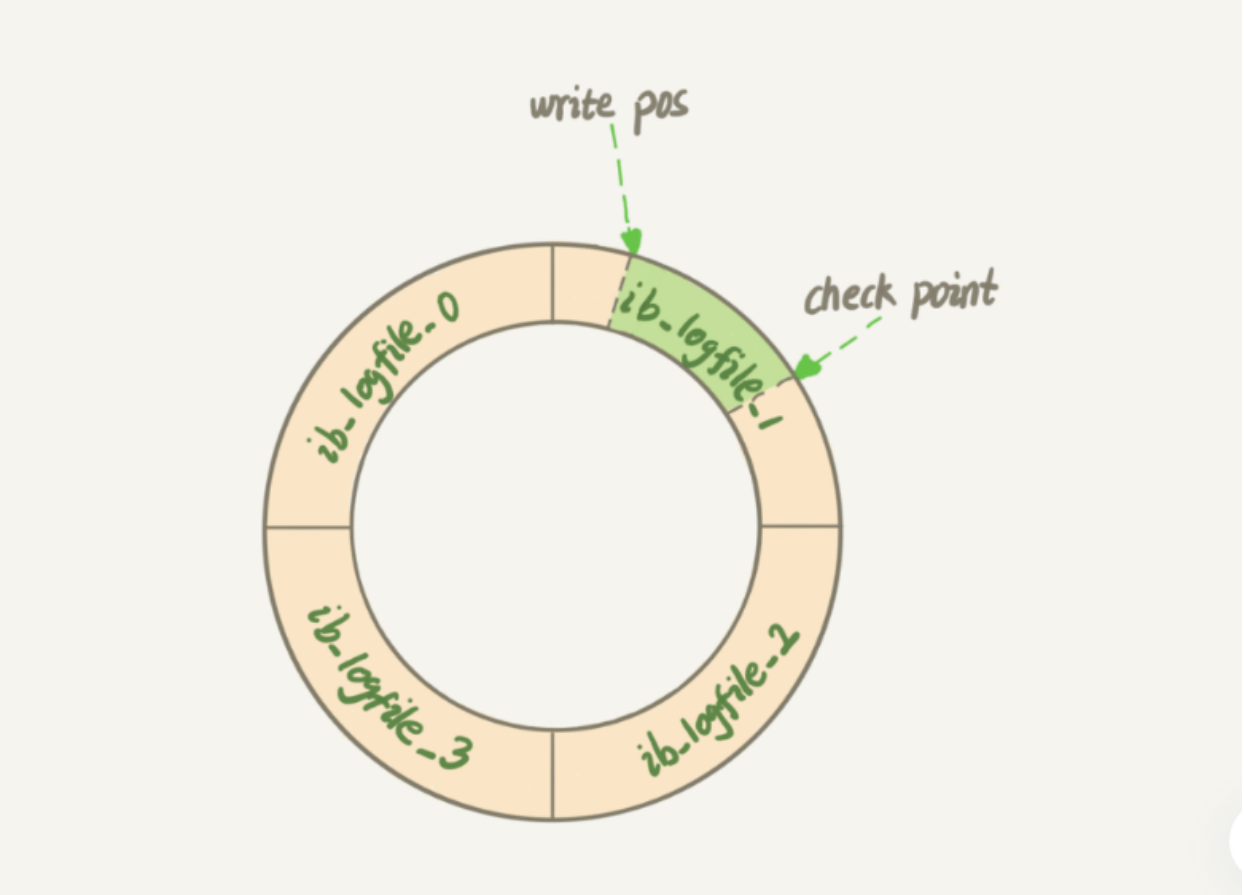
- 把账本翻开，找到相应页，把这次的账加上去
- 先在粉板上记下这次账，等打烊以后再往账本上记（效率更高）

mysql中也有相似场景：如果每次更新操作都需要写进磁盘，然后在磁盘中也要找到相应记录，然后更新，整个过程磁盘IO成本很高。mysql采用粉板记账的思想来提升写的效率。

粉板和账本配合的过程：mysql中常说的WAL技术，WAL的全称是Write-Ahead Logging，它的关键点就是先写日志，再写磁盘，即先写粉板，等不忙的时候再写账本。

- 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。
- innoDB引擎会在适当的时候，将这个操作记录更新到磁盘里面。
- redo log 满的时候，会触发磁盘的写操作

redo log是固定大小的，比如可配置为一组4个文件，每个文件大小是1GB，那么粉板总共就可以记录4GB的操作。从头开始写，写到末尾就又回到开头循环写。




- write pos是当前记录的位置，一边写一遍后移，写到第3号文件末尾就写0号文件的开头。
- checkpoint是当前要擦除的位置，也是往后推移并且循环，擦除记录前要把记录更新到数据文件
- write pos和checkpoint之间的是空闲部分。
- write pos追上checkpoint就表示粉板满了。
- 有了redo log ，InnoDB可以保证即使数据库发生异常重启，之间提交的记录也不会丢失，这个能力称为crash-safe。

三、binlog

redo log是InnoDB引擎特有的日志，而server层也有自己的日志，称为binlog。

为什么会有两份日志?

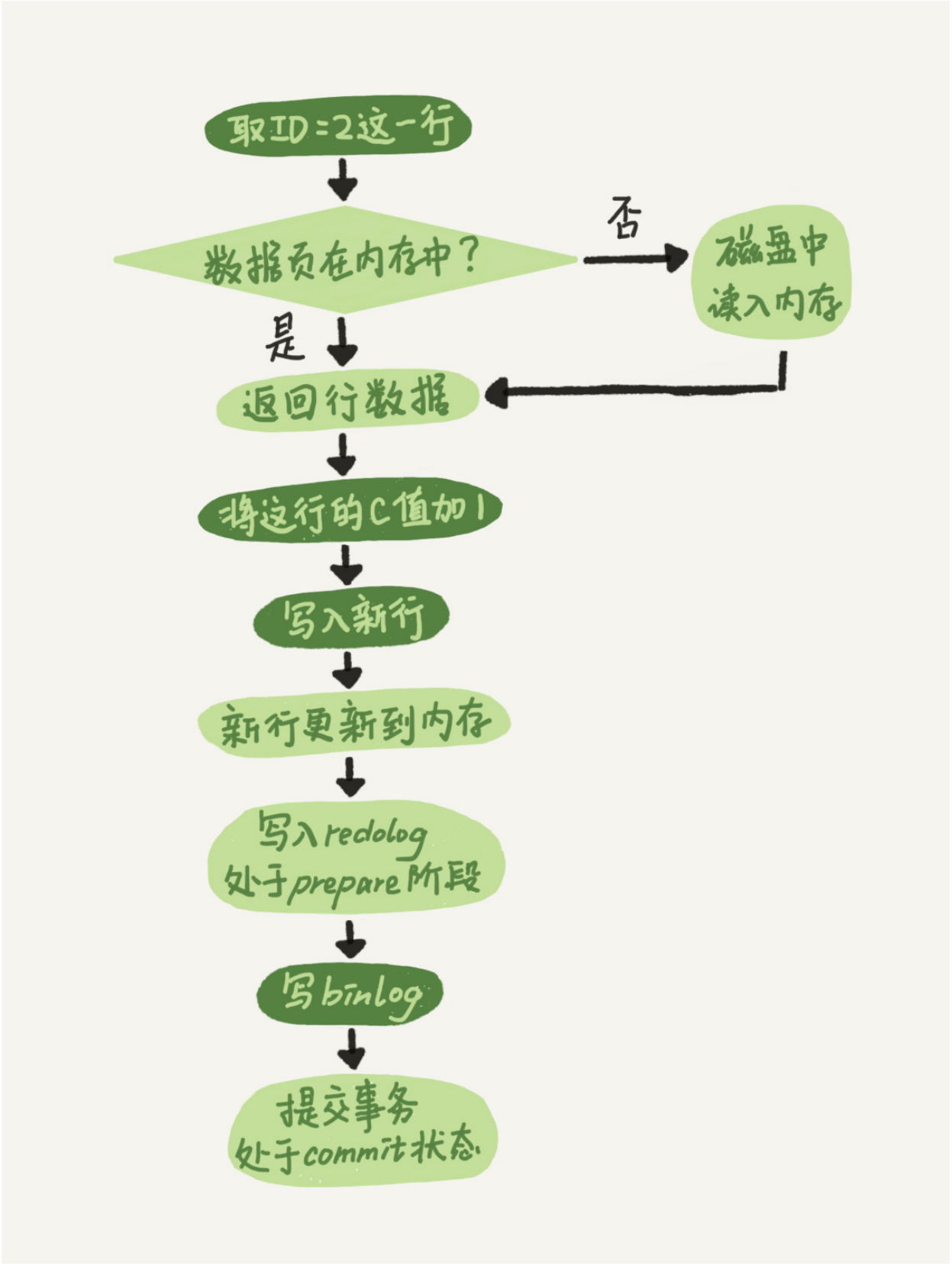
- mysql一开始没有InnoDB引擎，而是自带的MyISAM。MyISAM没有crash-safe能力，binlog日志只能用于归档。
- InnoDB以插件形式引入到Mysql中，使用redo log来实现crash-safe能力

两种日志的不同点：

- redo log属于引擎，binlog属于server层，所有引擎都可以使用。
- redo log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”
- redo log是循环写的，空间固定会用完；binlog是可以追加写入的，即binlog文件写到一定大小后会切换到写一个，并不会覆盖以前的日志。

执行器和InnoDB引擎执行update语句的内部流程：

- 执行器先调用引擎接口取id=2的记录。id是主键，引擎通过索引找到这一行。若该条记录在内存中，就直接返回给执行器。否则，需要先从磁盘读到内存再返回。
- 执行器拿到该记录，给值加1，得到新的记录，再调用引擎接口写入新记录。
- 引擎将新记录更新到内存，同时将更新操作记录到redolog，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。
- 执行器生成该操作的binlog，并把binlog写入磁盘
- 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

四、两阶段提交

为什么要有两阶段提交：为了让两份日志之间的逻辑一致。

怎样让数据库恢复到半个月内任意一秒的状态?

- 需要保留最近半个月的所有binlog，同时系统会定期做整库备份
- 首先找到最近一次的全量备份，从这个备份恢复到临时库
- 然后从备份的时间点开始，将备份的binlog依次取出，重放到误操作之前的那个时刻。此时临时库就与原来相同了。

为什么日志需要两阶段提交?

反证法

redo log和binlog是两个独立的逻辑，如果不用两阶段提交，要么是先写完redo log 要么先写完binlog。

- 若redo log写完，binlog还没写的时候，mysql进程异常。
  - redo log写后可恢复记录，记录恢复后c的值为1.
  - 但是binlog中没有该语句
  - 如果使用binlog恢复时，会出现恢复后的库与原库值不同
- 若binlog写完，redo log还没写时，mysql进程异常。
  - binlog已记录该操作
  - redo log还没写，崩溃恢复后这个事务无效，c的值还是0，
  - 此时利用binlog恢复库，就与原库不同了

# 三、 多版本并发控制（MVCC)

一、MVCC是什么？作用是什么?

- 用来解决数据库读写冲突的一种乐观并发控制
- 使得读不阻塞写，写也不阻塞读，同时解决脏读和不可重复读的问题

二、MVCC的特性?

- MVCC可以结合锁来解决写写冲突，也可以结合乐观锁来解决写写冲突。
- MVCC只有在可重复读和提交读两种隔离级别下，才会起作用。

三、InnoDB中MVCC的实现原理?

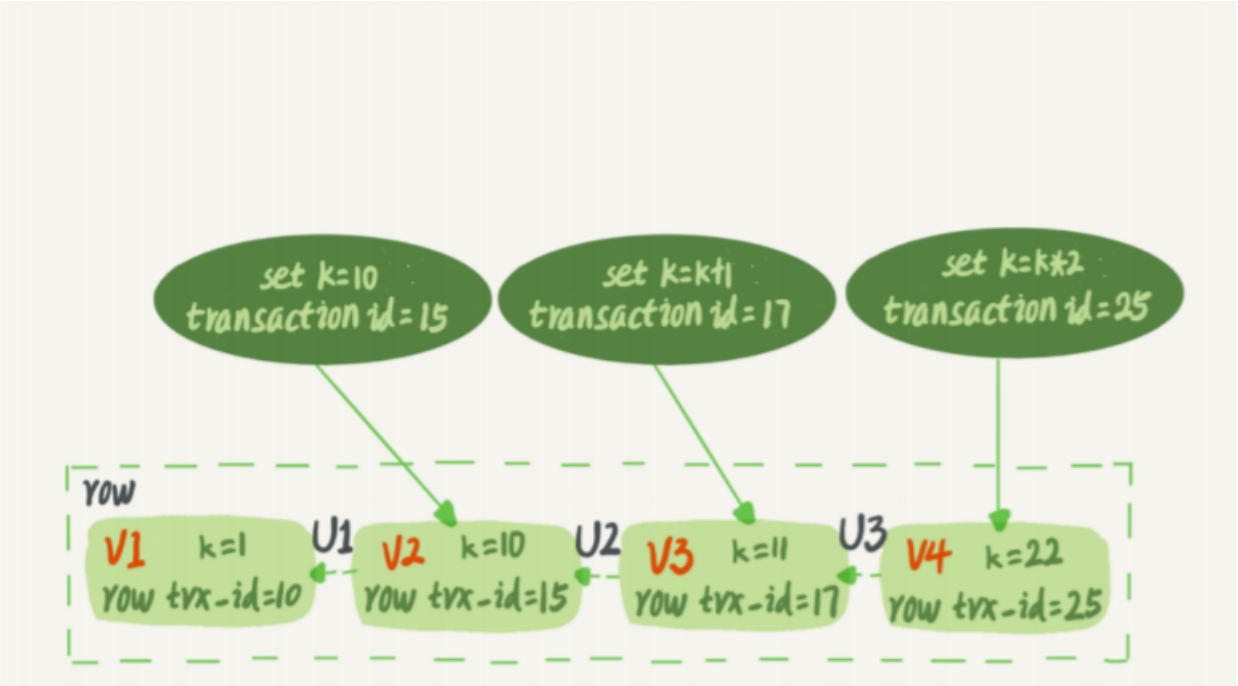
1、引入的概念

事务id：

- 每个事务启动时会申请一个事务id，该id是严格递增的。

数据版本：

- 每行记录都有多个版本，版本id，row trx_id，可根据当前版本和undo log直接得旧的数据版本。
- 通过在每行记录后保存两个隐藏的列来实现，一个列保存了row trx_id，一个列保存行的过期时间。




事务数组：

- 事务启动时，会创建一个数组，保存那些已经启动的，但还没有提交的事务id

低高水位：

- 低水位：事务数组中最小的事务id
- 高水位：系统中已创建过的事务id的最大值加1.

一致性视图：

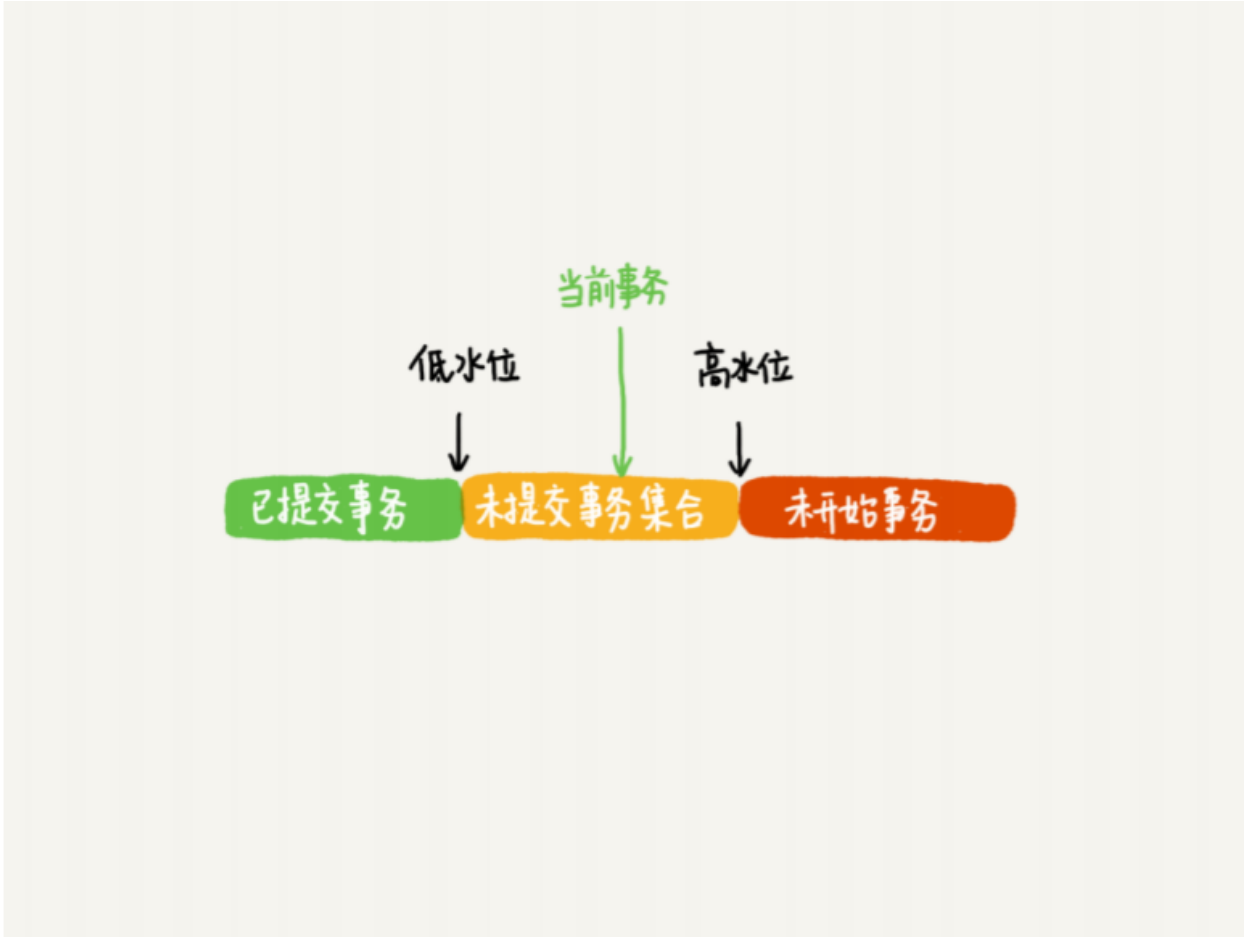
- 是InnoDB为了实现MVCC提出的概念，用于支持读提交和可重复读隔离级别的实现。
- 在这两种隔离级别下，事务执行期间能看到什么数据取决于一致性视图。
- begin、start transaction并不会立马建立一致性视图，而是等第一个快照读语句才创建。
- start transaction with consistent snapshot会立马创建一致性视图。

当前读：

- 执行修改时，都是先读后写。而读的是记录的当前版本
- 因为不读当前版本，可能会导致上一个事务的修改丢失

2、查询时，MVCC运作过程

查询到的记录，其row trx_id可能存在的情况：




- 绿色部分：小于低水位，说明当前事务启动时已提交，可见。
- 红色部分：当前事务之后的事务做的修改，不可见。
    - 为什么能出现未开始的事务：未开始指的是当前事务启动时未开始，当前事务启动后和执行查询之间的间隙，可能有后续事务启动。

- 黄色部分：（大于低水位，但小于当前事务的）
  - 若在当前事务数组中，表示当前事务启动时，还未提交的。不可见
  - 若不在事务数组中，表示当前事务启动时，已提交，可见。

示例：

```
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);
```

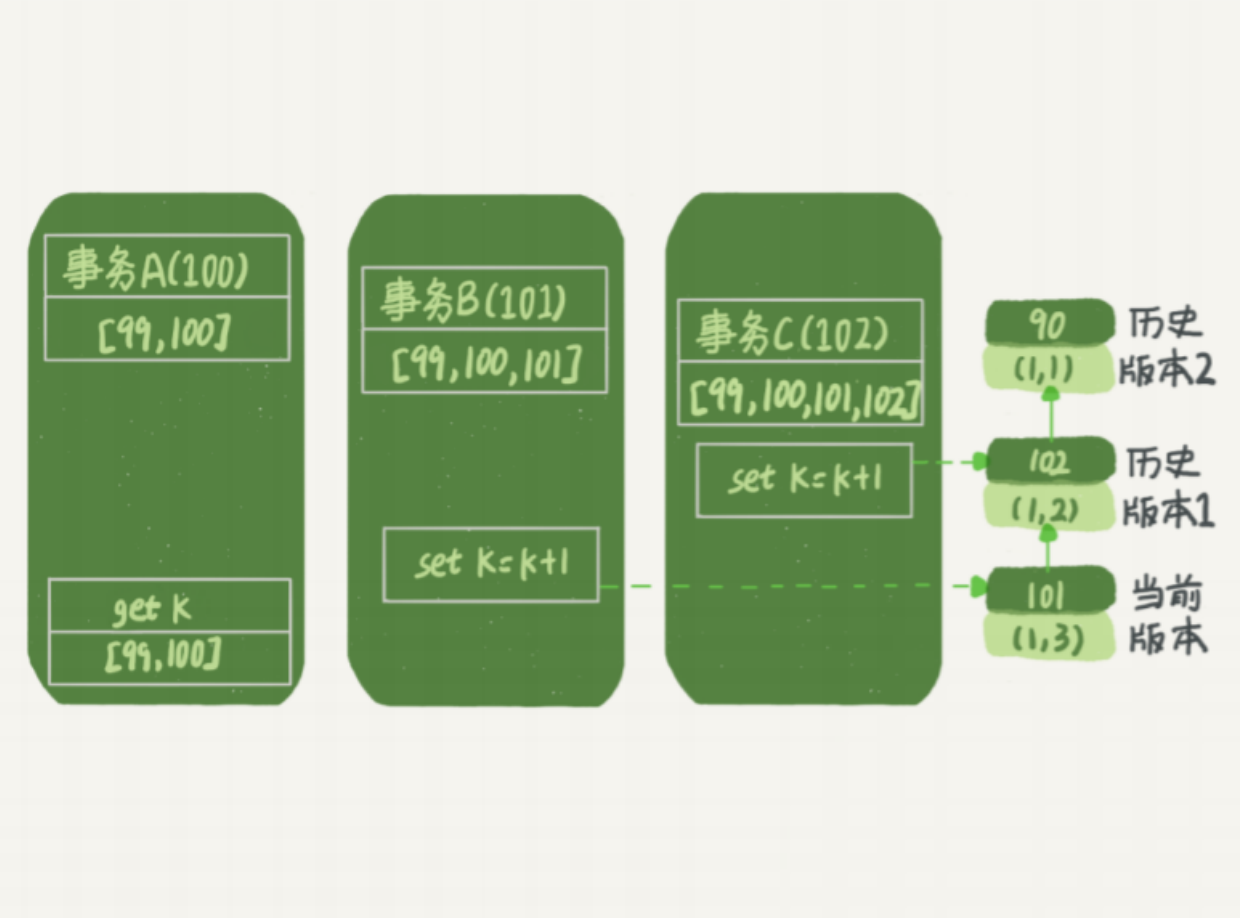
| 事务A | 事务B | 事务C |
| --- | --- | --- |
| start transaction with consistent snapshot; | | |
| | start transaction with consistent snapshot; | |
| | | update t set k=k+1 where id=1; |
| | update t set k=k+1 where id=1;<br>select k from t where id=1; | |
| select k from t where id=1;<br>commit; | | |
| | commit; | |

（1）、可重复读级别下

假设条件：

- 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[99,100,101,102]。

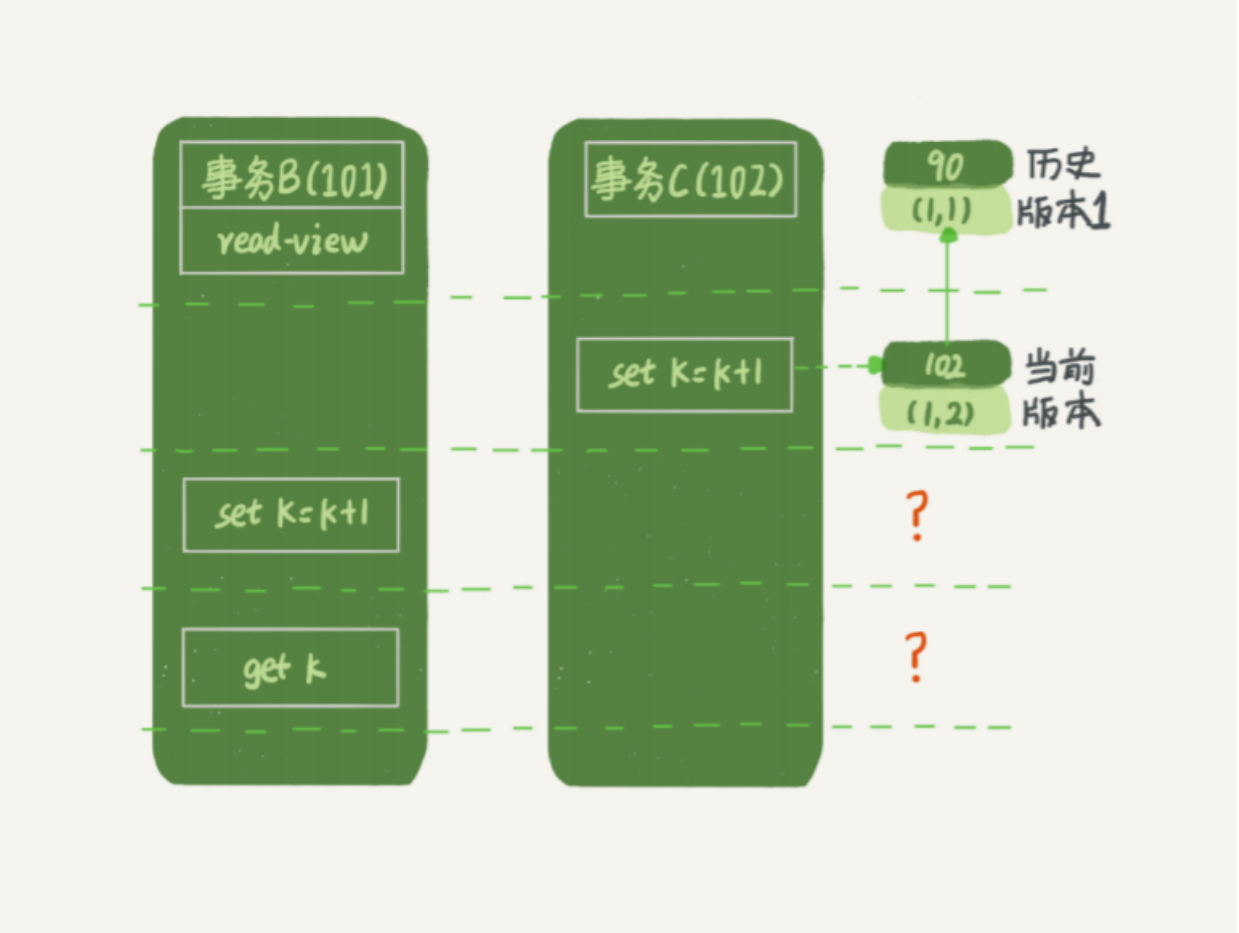
对事务A的查询结果分析：




- 找到（1,3）时，row trx_id=101,比高水位大，处于红色区域，不可见。
- 找到上一历史版本，row trx_id=102，比高水位大，不可见
- 再往前找，找到（1,1），它的row trx_id=90，比低水位低，可见。
- 事务A查询结果：k=1 ，事务B查询结果：k=3

3、更新（update）时，MVCC的运作过程

（1）、可重复读级别下：




事务B更新数据时，不能再历史版本上更新，否则事务C的更新就丢失了，因此事务B此时set k=k+1是在（1,2）基础上操作的。

- 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
- 除了update语句，select语句如果加锁，也是当前读。

| 事务A | 事务B | 事务C' |
| --- | --- | --- |
| start transaction with consistent snapshot; | | |
| | start transaction with consistent snapshot; | |
| | | start transaction with consistent snapshot;<br><br>update t set k=k+1 where id=1; |
| | update t set k=k+1 where id=1;<br>select k from t where id=1; | |
| select k from t where id=1;<br>commit; | | commit; |
| | commit; | |

结果分析：

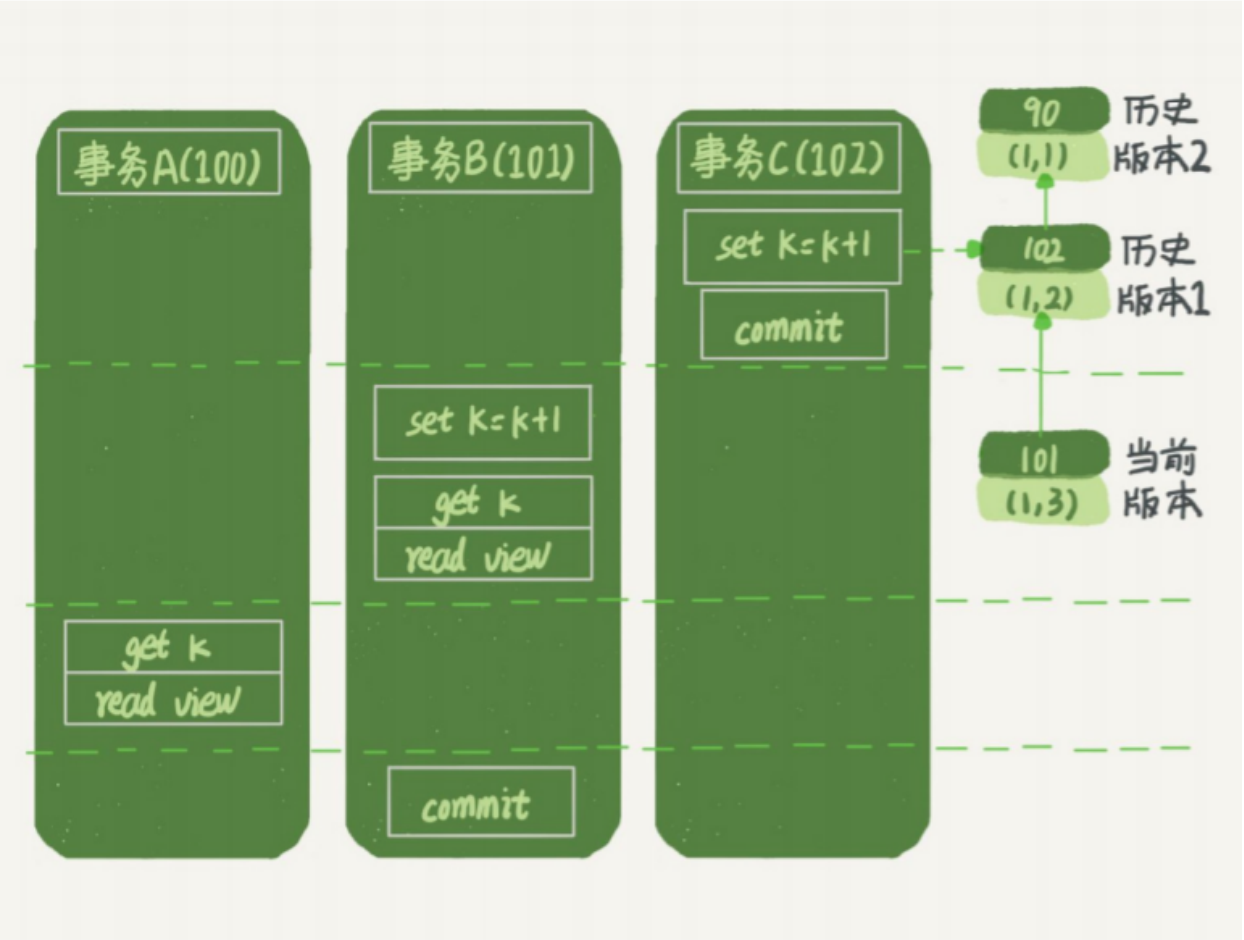
事务A：k=1

事务B：k=3

事务C没提交，对（1,2）版本上的写锁还没有释放。事务B是当前读，必须要读最新版本，而且必须加锁，所以要等待事务C释放这个锁，才能继续它的当前读。

- 由此可看出，涉及到写时，还是要锁参与的。锁加MVCC。

（2）、读提交级别下




事务A： k=2

事务B：k=3

四、MVCC与锁的关系?

- 单纯的利用锁实现并发控制，性能是不高的。
- 所以提出MVCC，在某些场景下利用MVCC来解决读写冲突问题，避免了加锁操作，来提高性能。
- 锁机制+MVCC构成了mysql的并发控制机制

# 四、深入浅出索引（上）

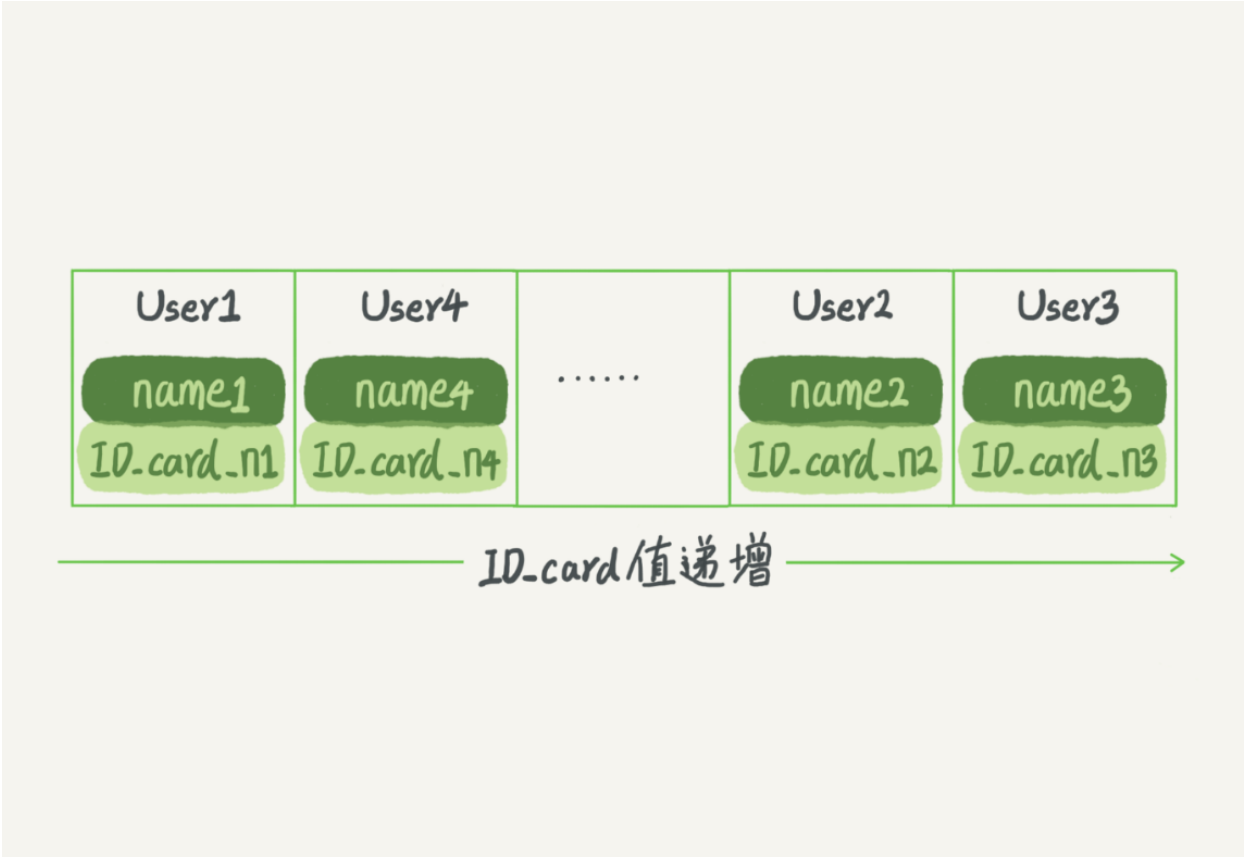
一、索引的常见模型

实现索引的方式有很多，比较常见的有哈希表、有序数组、搜索树。

1、哈希表作为索引

- 哈希表存储key-value形式的数据
- 只要输入key，经过时间复杂度O(1)的计算，就能找到value。
- 原理：把key-value作为元素存储在数组中，用一个哈希函数将key换算为一个数组的下标。
- 作为索引的优点：查询速度快，新增也快
- 作为索引的缺点：数组中的值是无序的，区间查询速度慢，用空间换时间。

2、有序数组作为索引




- 二分法查找，时间复杂度是O(log(N))，等值查找和范围查找都很快
- 更新数据时，代价太大。
- 适用于静态存储引擎

3、二叉搜索树作为索引

- 查找和插入的时间复杂度都是O(log(N))
- 缺点：高度不可控，容器形成单边数，层数过高

4、多路搜索树

示例：以InnoDB的一个整数字段索引为例，差不多是1200个分支，树高是4时，就可存储1200^3个值，约定于17亿，索引树根节点总是在内存中，一个10亿行的表一个整数字段的查找，最多只需要访问3次磁盘。

## 二、InnoDB的索引模型

### 1、概述

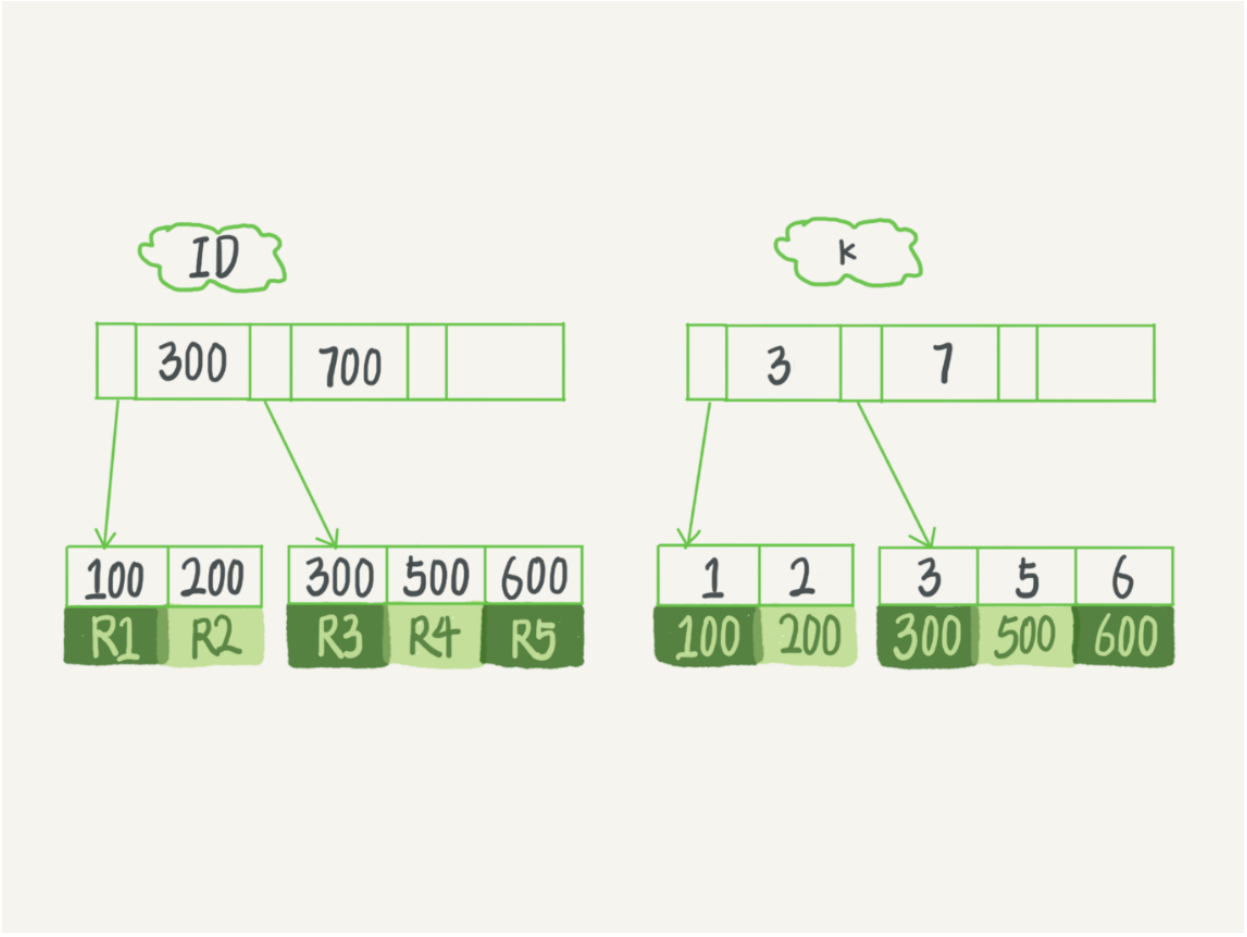
- 索引是在存储引擎层实现的，不同存储引擎的索引的工作方式不一样。
- 在InnoDB中，表都是根据主键顺序以索引的形式存放的（即主键索引树叶子节点存放的是数据，不是地址，靠谱不？），这种存储方式的表称为索引组织表。？？？
- InnoDB使用了B+数索引模型，所以数据都是存储在B+树中。？？
- 每个索引在InnoDB里面对应一个B+树

### 2、示例

有一个主键列为ID的表，表中有字段k，并且在k上有索引。

建表语句：

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




3、索引类型分类

- 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中，主键索引也被称为聚簇索引
- 非主键索引的叶子节点的内容是主键的值。在InnoDB中，非主键索引也被称为二级索引

基于主键索引和普通索引的查询有什么区别?

- 主键查询方式，只需要搜索主键索引树
- 普通索引查询方式，需要先搜索k索引树，得到主键值，再到主键索引树中搜索一次。这个过程称为回表。

4、自增主键

什么是自增主键?

- 自增列上定义的主键，在建表语句中的定义： not null primary key auto_increment
- 插入新记录时可以不指定id，系统会获取当前id最大值加1作为下一条记录的id值。
- 自增主键插入在后面追加，不会引起叶子节点的分裂。有业务逻辑的字段做主键，则往往不容器保证有序插入，这样写数据的成本较高。

假设表中有一唯一字段，比如字符串形式的身份证号，那么应该用身份证号做主键，还是有自增字段做主键呢?

- 由于每个非主键索引的叶子节点存储的是主键的值。
- 如果用身份证号做主键，每个二级索引的叶子节点占用约20个字节，而如果用整型作为主键，则只需要4字节。
- 主键长度越小，普通索引的叶子节点就越小，普通索引占用的空间也就越小
- 从性能和存储空间方面考量，自增主键往往是更合理的选择。

有没有什么场景适合用业务字段直接作为主键?

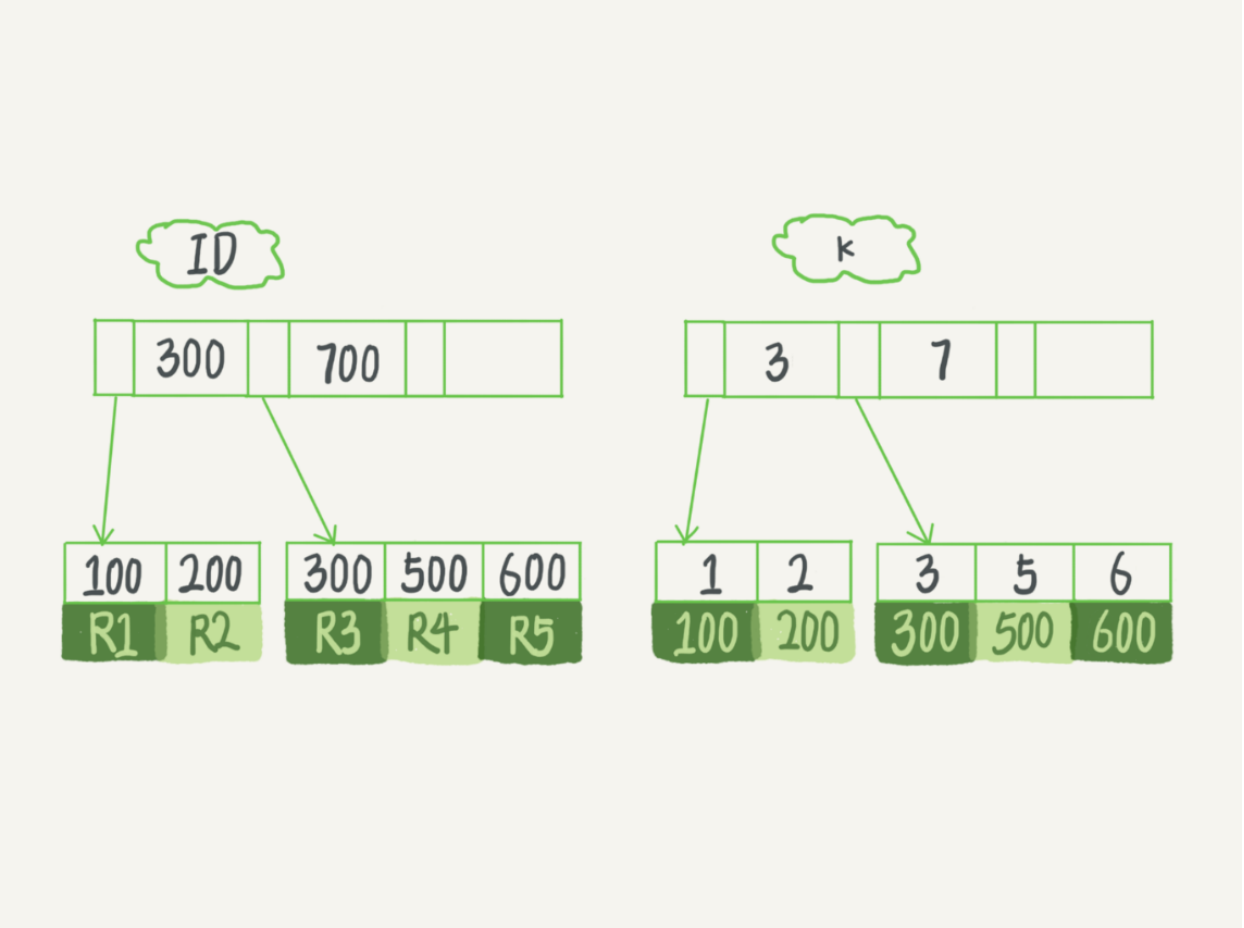
- 有些业务的场景需求是：只有一个索引，该索引必须为唯一索引。

# 五、深入浅出索引（下）

一、select * from T where k between 3 and 5，需要执行几次树的搜索操作，会扫描多少行?

```
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');
```




上述sql查询语句的执行流程：

- 在k索引树上找到k=3的记录，获得主键值id=300

- 再到主键索引树上查到id=300对应的记录R3；
- 在k索引树上取下一个值k=5，取得id=500
- 再回到主键索引树查到id=500对应的R4
- 在k索引树取下一个值k=6，不满足条件，循环结束。

在这个过程中，回到主键索引树搜索的过程，称为回表。整个过程读了k索引树的3条记录，回表了两次。

故应该考虑索引优化，避免回表的过程。

二、覆盖索引

1、什么是覆盖索引?

- 如果执行的是select ID from T where k between 3 and 5;这时只需要查id的值，而id的值已经在k索引树上了，因此不需要回表。
- 即这个查询中，索引k已经覆盖了我们的查询需求，我们称为覆盖索引。
- 使用覆盖索引是一个常用的性能优化手段
- 在引擎内部使用覆盖索引在索引k上其实读了三个记录，但是对于mysql的server层来说，它就是找引擎拿到了两条记录，因此mysql认为扫描行数是2。

2、在一个市民信息表上，是否有必要将身份证号和名字建立联合索引?

```
CREATE TABLE `tuser` (
  `id` int(11) NOT NULL,
  `id_card` varchar(32) DEFAULT NULL,
  `name` varchar(32) DEFAULT NULL,
```

```
  `age` int(11) DEFAULT NULL,
  `ismale` tinyint(1) DEFAULT NULL,
  PRIMARY KEY (`id`),
  KEY `id_card` (`id_card`),
  KEY `name_age` (`name`,`age`)
) ENGINE=InnoDB
```

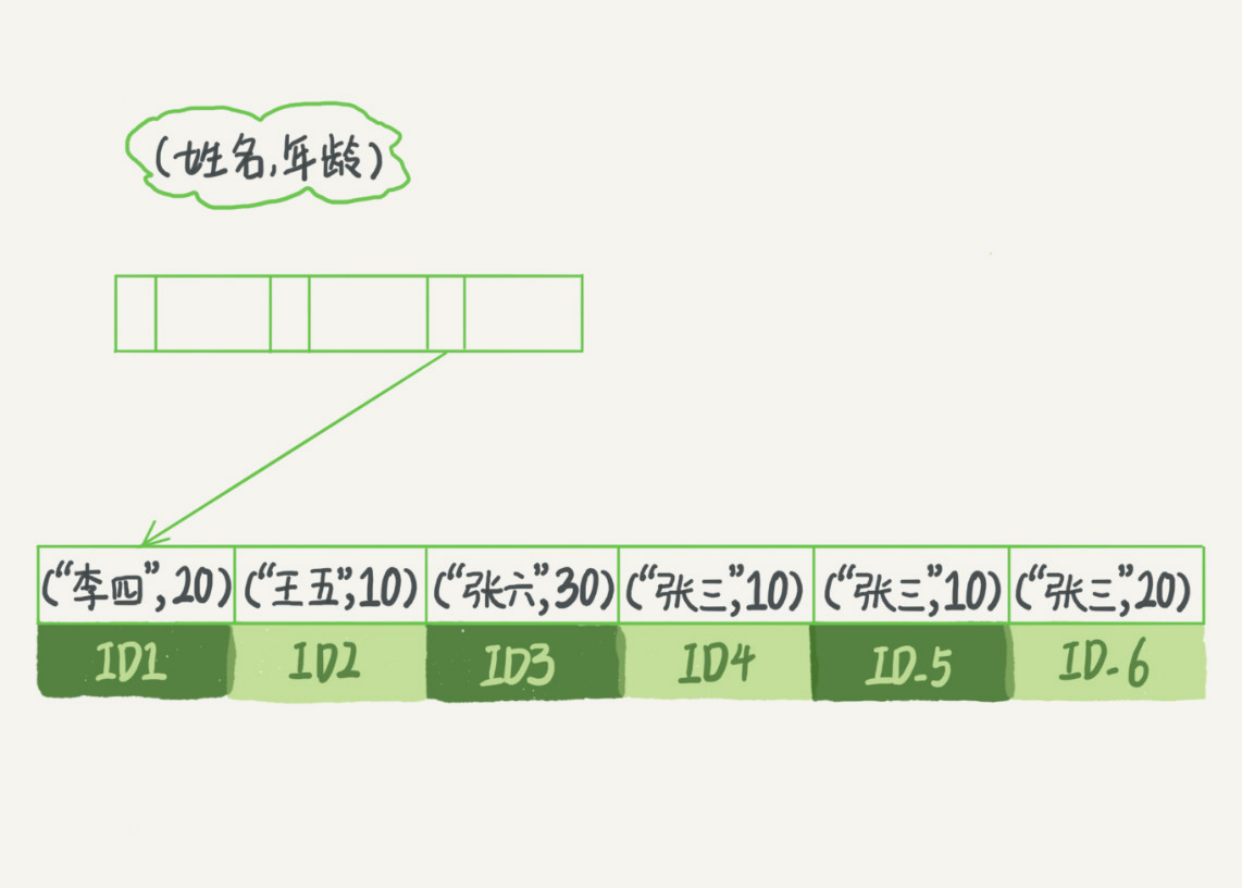
- 要根据身份证号查询市民信息的高频请求，联合索引就有意义了
- 可以在这个高频请求上用到覆盖索引，不需要再回表操作了，减少语句的执行时间
- 当然索引字段的维护是有代价的，在建立冗余索引时要权衡考虑。

三、最左前缀原则

- 存在的问题：
  - 为每一种查询都设计一个索引，索引会太多
  - 虽然有些查询需求出现的概率不高，但总不能走全表扫描。
- B+树可以利用索引最左前缀，来定位记录

1、示例

用（name，age）联合索引来分析




- 索引项是按照索引定义里面出现的字段顺序来排序的
- 若查询“张三”，可以快速定位到id4，然后向后遍历得到所有需要的结果
- where name like '张%',也能用上这个索引
- 不只是索引的全部定义，只要满足最左前缀，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

2、在建立联合索引时，如何安排索引内的字段顺序?

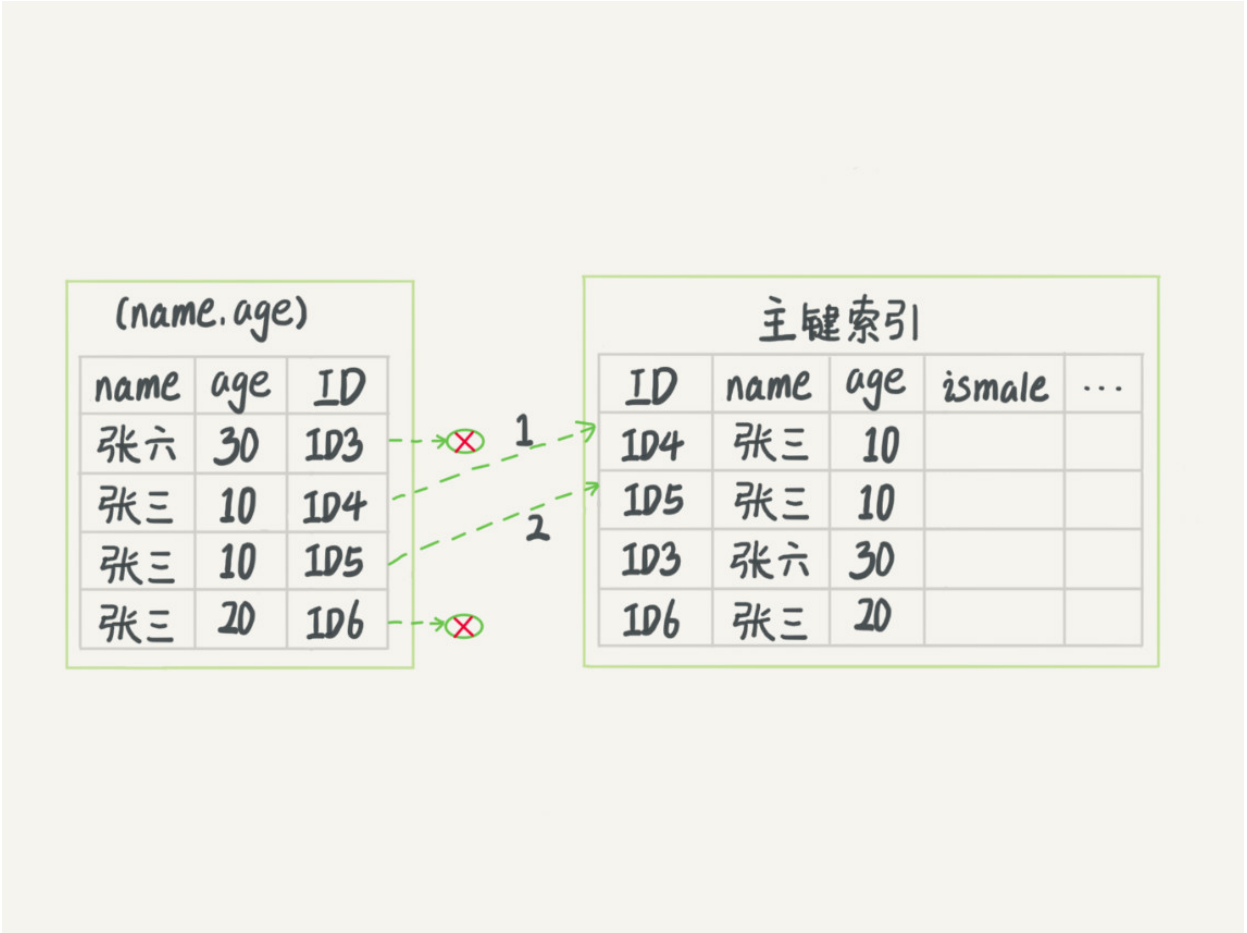
- 索引的复用能力，若已经有（a，b）这个联合索引后，一般不需要单独在a上建立索引了。
- 第一原则是，如果通过调整顺序，可以少维护一个索引，那么这个顺序往往就是需要优先考虑的。
- 在有些场景下，需要同时维护(a,b),(b)这两个索引。

四、索引下推

1、不符合最左前缀的部分，会怎么样?

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

- 在搜索索引树时，能用到“张”，找到第一个满足条件的记录ID3。
- 然后呢?
  - 在mysql5.6之前，只能从id3开始回表。到主键索引上找出数据行，再对比字段值。
  - mysql5.6以后引入了索引下推优化，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。




五、问题：

```
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;
主键索引（a,b），在字段c上创建一个索引即可包含三个字段，为什么还要创建“ca”，“cb”这两个索引?

解释：因为有以下业务查询：

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

这种解释对吗

InnoDB会把主键字段放到索引定义字段后面，同时会去重。
当主键是(a,b)时，
定义为c的索引,实际上是(c,a,b)
定义为(c,a)的索引，实际上是(c,a,b)
所以索引ca重了

六、