# Mysql的整体结构：

Client

Server

连接器：做认证鉴权

分析器：词法 语法分析

优化器：对语句做优化（选择索引什么的）

执行器：负责和底层的 存储引擎交互，执行命令的

执行器

MyISAM：

InnoDB：

Mem

# Mysql的存储引擎：

MyISAM；

适用于查询多，更新少，不需要事务的场景（不需要ACID）

InnoDB：

适用于需要事务的场景，在查询多，更新少的场景，性能相比于MyISAM差一些。

MyISAM比InnoDB慢：

需要维护自身的事务（ACID）所需要维护的东西（undolog redolog mvcc）



# 索引结构：

Mysql使用的是B+树作为底层数据结构

## 聚簇索引和非聚簇索引：

聚簇索引：数据文件和索引文件在一起

非聚簇索引：数据文件和索引文件分开

## mysql索引类型：

Primary key：唯一，非空索引（聚簇索引）

key/index：普通索引（非聚簇索引）

unique：唯一索引（非聚簇索引）

复合索引：多个key组成的

### InnoDB：

支持支持聚簇索引 和 非聚簇索引

#### 聚簇索引：

聚簇索引默认是主键，如果表中没有定义主键，InnoDB 会选择一个唯一的非空索引代替。如果没有这样的索引，InnoDB 会隐式定义一个主键来作为聚簇索引。

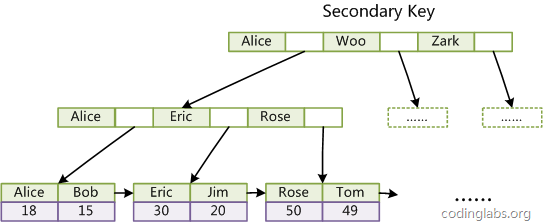
特点：非叶子节点保存的是索引信息，叶子节点保存的是 **整条记录**



#### 非聚簇索引：

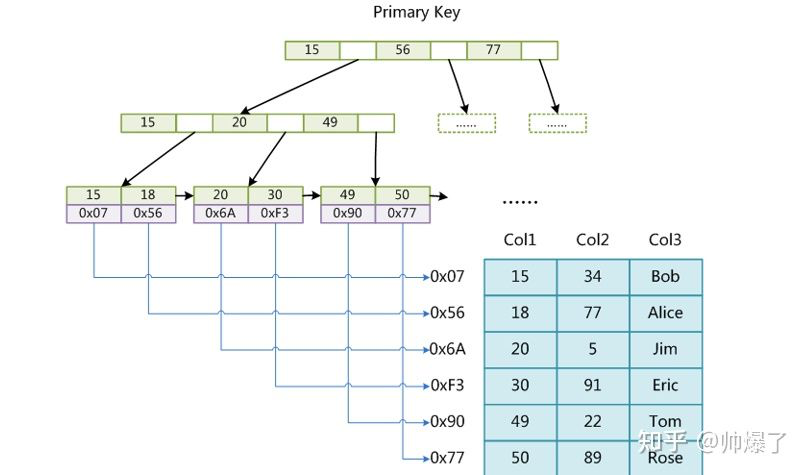
数据和索引不存在一起，根节点只存key值和主键的值（需要拿主键值到聚簇索引去读值。注意是值，不是地址，为了防止频繁更新问题）

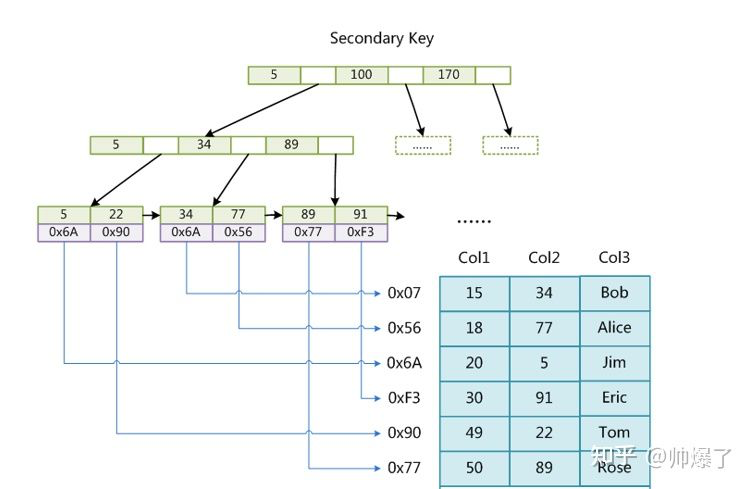
特点：非叶子节点保存的是索引信息，根节点只存**索引值**和**主键的值**



## MyISAM：

MyISAM的索引都是非聚簇的，主键索引和非主键索引原理一样的，唯一区别是主键索引是唯一非空的。辅助索引是可以重复的





# 数据库的锁：

## 全局锁：

**注意：只有MyISAM下逻辑备份的时候再考虑该锁！！！！**

加锁：Flush tables with read lock (FTWRL)

释放锁：unlock tables/连接释放

作用：让整个数据库处于只读

使用场景：MyISAM做全库的逻辑备份（也就是把整库每个表都 select 出来存成文本。）InnoDB使用的是mysqldump -single-transaction，使用mvcc，不影响业务使用。

缺点：业务停摆，主从复制停摆

知识点：全局锁 和 set readonly有相同的作用，但是不要选择readonly。

## 表锁：

**注意：只有MyISAM场景下才考虑该锁。**

**1：表锁（系统自动加，也可以手动加）（Server层加）**

加锁：lock tables … read/write

释放锁：unlock tables/连接释放

使用场景：一般只有MyISAM下使用，读锁和写锁都会自动加，且表锁不会思锁

**2：MDL（元数据锁）（系统自动加）（Server层加）**

**注意：存在修改语句的时候要考虑该锁！！！**

加锁：执行增删改查语句的使用自动加 MDL读锁，执行DDL的时候自动加MDL写锁。

释放锁：事务结束（事务场景下）

解释：实现DDL和DML语句的隔离，增删改查时不能修改表，修改表时不能增删改查；增删改查是读锁，内部可以并发。



**3：意向锁**

**innodb的意向锁主要用户多粒度的锁并存的情况**。比如事务A要在一个表上加S锁，如果表中的一行已被事务B加了X锁，那么该锁的申请也应被阻塞。如果表中的数据很多，逐行检查锁标志的开销将很大，系统的性能将会受到影响。为了解决这个问题，可以在表级上引入新的锁类型来表示其所属行的加锁情况，这就引出了“意向锁”的概念。

举个例子，如果表中记录1亿，事务A把其中有几条记录上了行锁了，这时事务B需要给这个表加表级锁，如果没有意向锁的话，那就要去表中查找这一亿条记录是否上锁了。如果存在意向锁，那么假如事务Ａ在更新一条记录之前，先加意向锁，再加Ｘ锁，事务B先检查该表上是否存在意向锁，存在的意向锁是否与自己准备加的锁冲突，如果有冲突，则等待直到事务Ａ释放，而无须逐条记录去检测。事务Ｂ更新表时，其实无须知道到底哪一行被锁了，它只要知道反正有一行被锁了就行了。

**主要作用是处理行锁和表锁之间的矛盾，能够显示“某个事务正在某一行上持有了锁，或者准备去持有锁”**

## 行锁/间隙锁/nextkey lock：

#### 特点：

InnoDB模式默认使用是行锁；

行锁基于索引进行加锁，如果相关语句中没有使用到 索引，那么此次 DML语句**升级为表锁**；（<https://time.geekbang.org/column/article/75173> 如何解决幻读后半部分有提到）

#### 注意点：

RC模式下：只会加行锁

RR模式下：行锁+间隙锁+nextkey lock

#### 小的知识点：

1:在select 语句的时候，默认不加任何锁，不参与锁的体系

2:加读锁需要手动加：lock in share mode；读锁会锁住：**使用到的索引中的命中项**；

3:DML语句会默认加写锁，select可以手动加写锁，for update；写锁会锁住：**所用到的索引** 和 **聚簇索引中的命中项**；

4:注意数据更新时，会涉及到索引的更新，如果对应的索引此时在上锁，那么更新会被阻塞。

ps：不手动加锁的select语句，不受任何的锁的影响，肯定可以读数据

#### 验证上述知识点：

**表结构：注意，一定要使用唯一索引去验证，因为普通索引会有间隙锁，不能明显地验证 lock in share mode只给命中到的索引加锁。**

**表结构：**

use test;

DROP TABLE IF EXISTS `t`;

CREATE TABLE `t` (

`id` int NOT NULL,

`a` int not null,

`b` int not null,

`c` int,

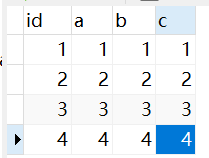
PRIMARY KEY (`id`),

unique INDEX `ka`(`a`) ,

unique INDEX `kb`(`b`)

)

**数据集：**

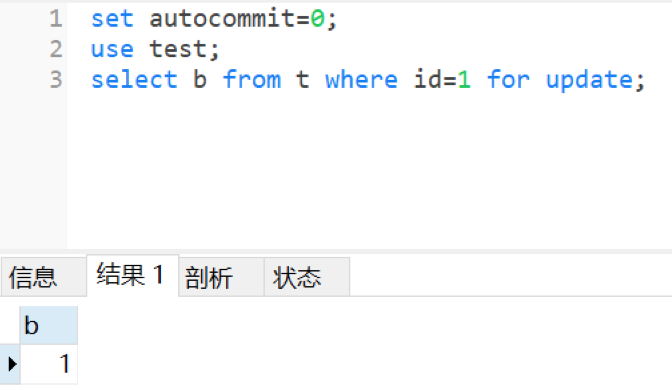
****

##### 1: lock in share mode的时候，只会给命中到的索引行加锁，不会给聚簇索引对应的行加锁

###### 1-1:先给二级索引a加锁



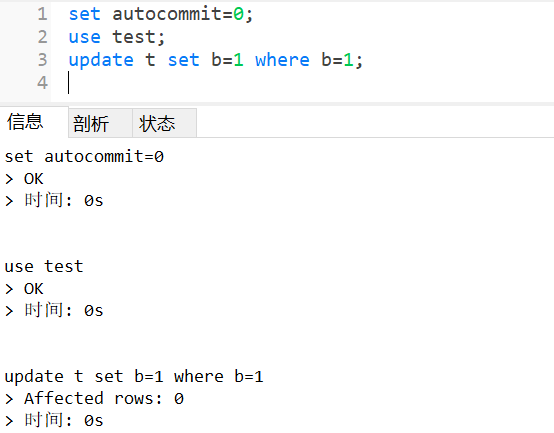
###### 1-2:再给聚簇索引中id=1的行进行读取，读取成功，验证成功。



##### 2:验证update的时候，会自动加写锁，并且会锁住 命中的索引和 对应聚簇索引中的行

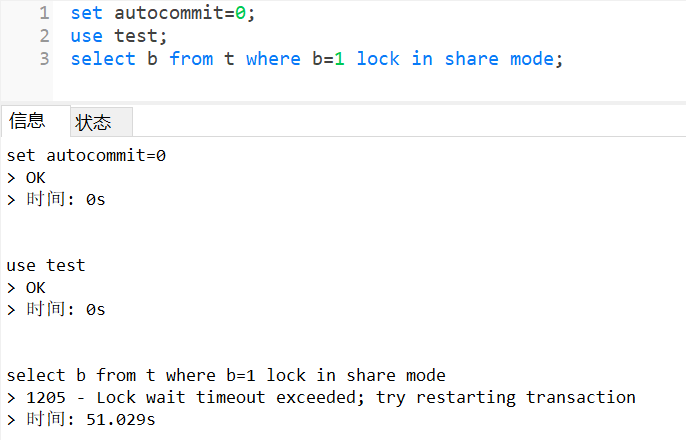
###### 2-1: 先给二级索引b加写锁，更新字段b

涉及：b二级索引更新+主键索引更新

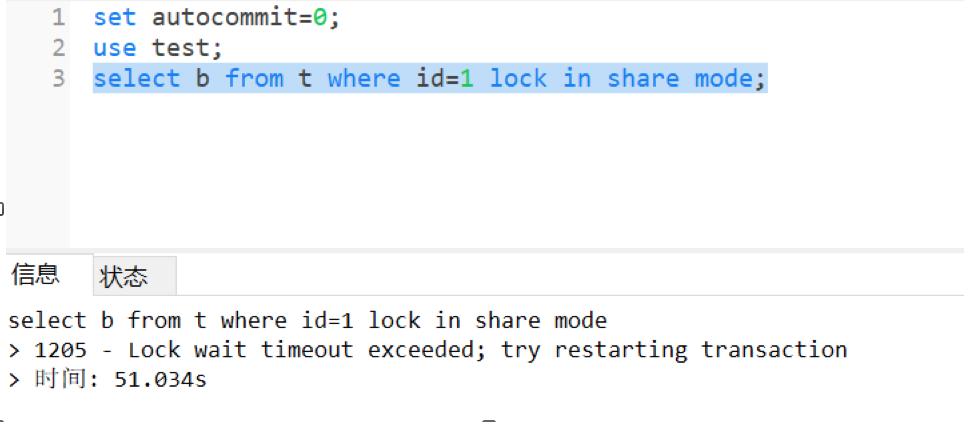


###### 2-2：查询b字段，通过b=1查询（覆盖索引查询 二级索引b）

注意：不涉及回表操作，所以如果阻塞住，那么说明锁住了二级索引

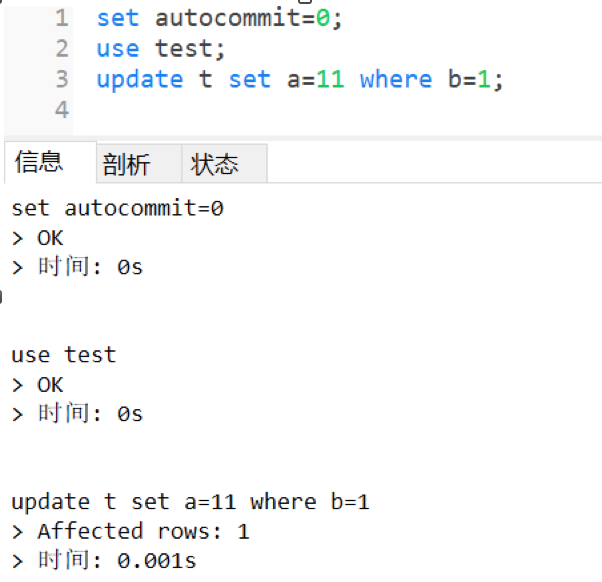


###### 2-3:使用主键进行查询，验证 聚簇索引加上了锁

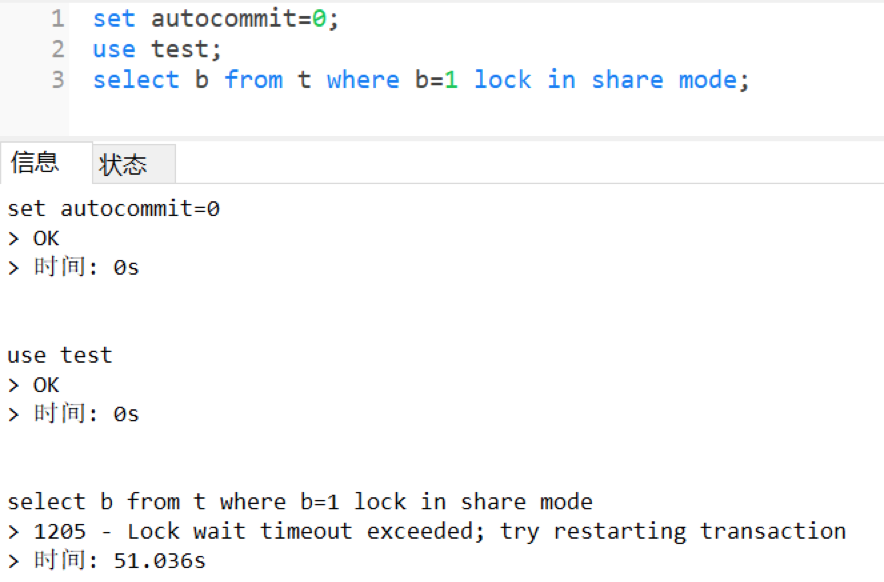


##### 3:验证更新情况时：1会给命中的二级索引加锁；2：命中的主键索引加锁；3：需要更新的字段对应的二级索引加写锁进行更新。

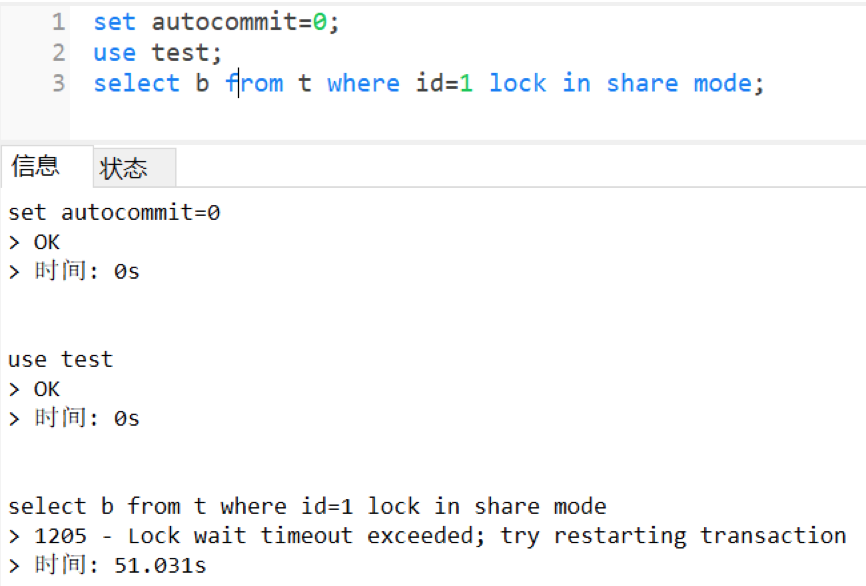
###### 3-1：通过索引b进行过滤，并且更新a字段



###### 3-2: 验证 二级索引b加了行锁

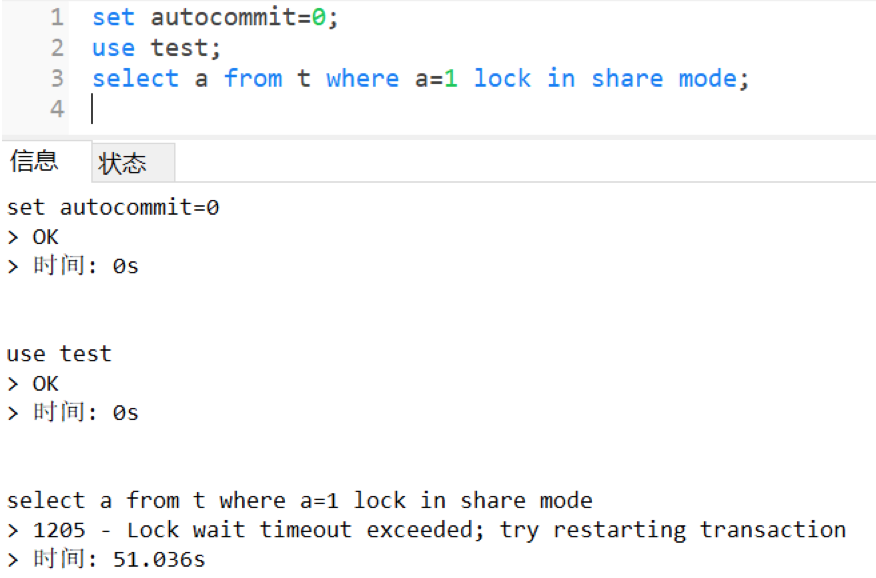


###### 3-3:验证 聚簇索引加了行锁



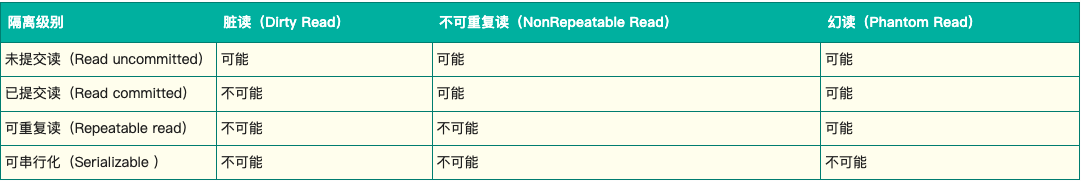
###### 3-4:验证更新字段的a 二级索引加了行锁：

注意：a在更新的时候设置的是11，和原有的值不同，所以可以会更新二级索引，如果a的值不变，不会更新二级索引，也不会加行锁



## 间隙锁和 nextLock key的引入：

#### 我们知道数据库有4个隔离级别（并发的事务之间的隔离级别）



### 事务的隔离存在三个问题：

###### 1:脏读：事务A中可以读到事物B中没有提交的修改数据

###### 2:不可重复读：事务A中，对于同一条数据，在前后两次读取的时候，值被修改过。

###### 3:幻读：在事务A种，读一批数据，前后两次读的数据条数不同，读到了其他事务insert的数据。

### 对应了数据库的四种隔离界别

###### 这四种隔离界别在不同程度上，解决了上述三个问题

###### 1：读未提交：解决了脏读

没有解决任何问题

###### 2：读已提交：解决了脏读

原理：MVCC保证

解决办法：在同一个事务中，每次在**读取之前**，都先创建一个快照，快照中数据是当前时间点，已经commit的事务的数据。

###### 3：可重复读：解决了脏读+不可重复读+幻读

原理：MVCC保证

解决办法：在**事务开始时**，先创建一个快照，这个快照的数据是，当前时间点的已经commit的事务的数据。

注意：这里模式是使用select读取，并没有手动加读锁和写锁的场景，如果加了读锁或者写锁，那么事务中的读就是当前读，读的是最新已提交事务的数据。

###### 4：串行化：

解决了所有问题，把并行改为行；

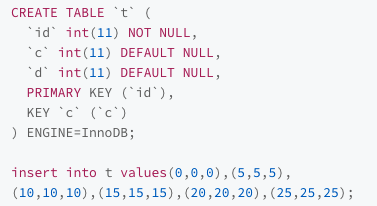
### 问题引入：

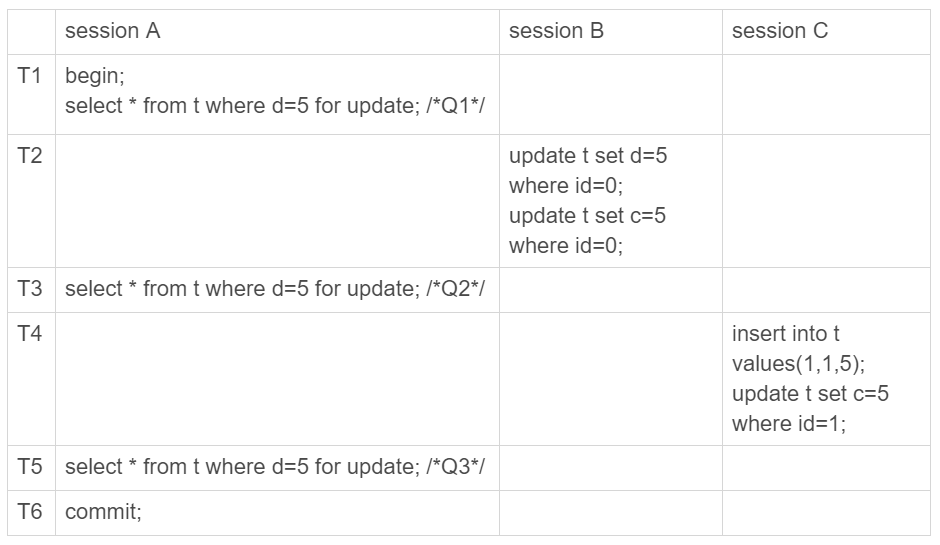
RR模式下MVCC已经解决了上面的三个问题：

因为幻读是指的，当前事务多次读取的 数据，都是事物开启之前，最后提交的事务的数据。并没有限定其他事务不能再insert数据。

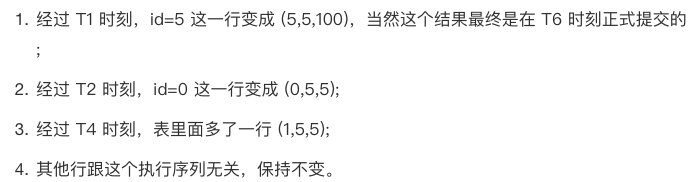
不可重复读指的是，同一条数据，当前事务前后读取都是一样的值。没有限定其他事务不能update数据。

那么如下的场景，只有会带来数据和逻辑日记不一致的问题：





数据逻辑：



日志逻辑：



问题根因就是，加锁时只加了 行锁，不加间隙锁导致的。

所以：间隙锁的引入，就是解决加行锁 不能阻止新数据的插入（对其他事务有影响的行），从而带来的数据和 日志不一致的问题。

**PS：RC模式下只会加行锁,所以可能会有数据与日志逻辑上不一致问题：**

**所以在RC模式下，需要 配合binlog\_format=row使用，不然会有不一致问题。**

### 当前读是会打破MVCC的 不可重复读和 幻读的

### 加锁的原则：

https://time.geekbang.org/column/article/75659

1：扫描到的 数据对象 会加锁

2：加锁都是加的next-key lock（左开右闭）

3：等值查询时：如果**唯一索引**情况下**命中行**，则退化为行锁

3-1:非唯一索引的等值查询，需要找到下一个不满足的 数据为止

4: 索引上的等值查询，向右遍历时且**最后一个值不满足等值条件的时候**，next-key lock 退化为间隙锁。

一个 bug：唯一索引上的范围查询会访问到不满足条件的第一个值为止。（所有索引的范围 + 非唯一索引的等值都是找到不满足的第一个值为止）

5：范围查询：向后访问到最后一个不满足的值

6：>= 可以拆分为> 和=两个操作来分析。

# MVCC

<https://time.geekbang.org/column/article/70562> 林晓斌牛逼！

之前讲到数据库事务的隔离性带来的三个问题，MVCC解决了其中绝大多数的问题，那么MVCC是怎么实现的呢？着章节就是复习MVCC的知识。

在RR隔离模式下，MVCC保证了 事务中的所有的普通读操作，只能读到创建事务前的 已经提交的数据；

在RC隔离模式下，MVCC保证了 事务中的所有的 读操作，只能读到当前读操作前的已经提交的事务；

**本文使用RR隔离模式进行分析MVCC原理**

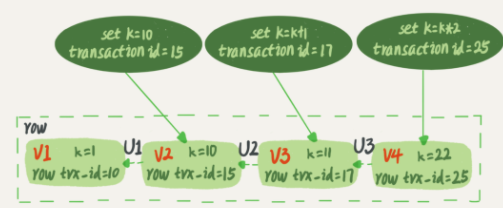
## **前提知识：**

#### 1:事务层面

InnoDB 里面每个事务有一个唯一的事务 ID，叫作 transaction id。它是在事务开始的时候向 InnoDB 的事务系统申请的，是按申请顺序严格递增的。

#### **2:数据层面**

而每行数据也都是有多个版本的。每次事务更新数据的时候，都会生成一个新的数据版本（不commit也会），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，旧的数据版本要保留，并且在新的数据版本中，能够有信息可以直接拿到它。



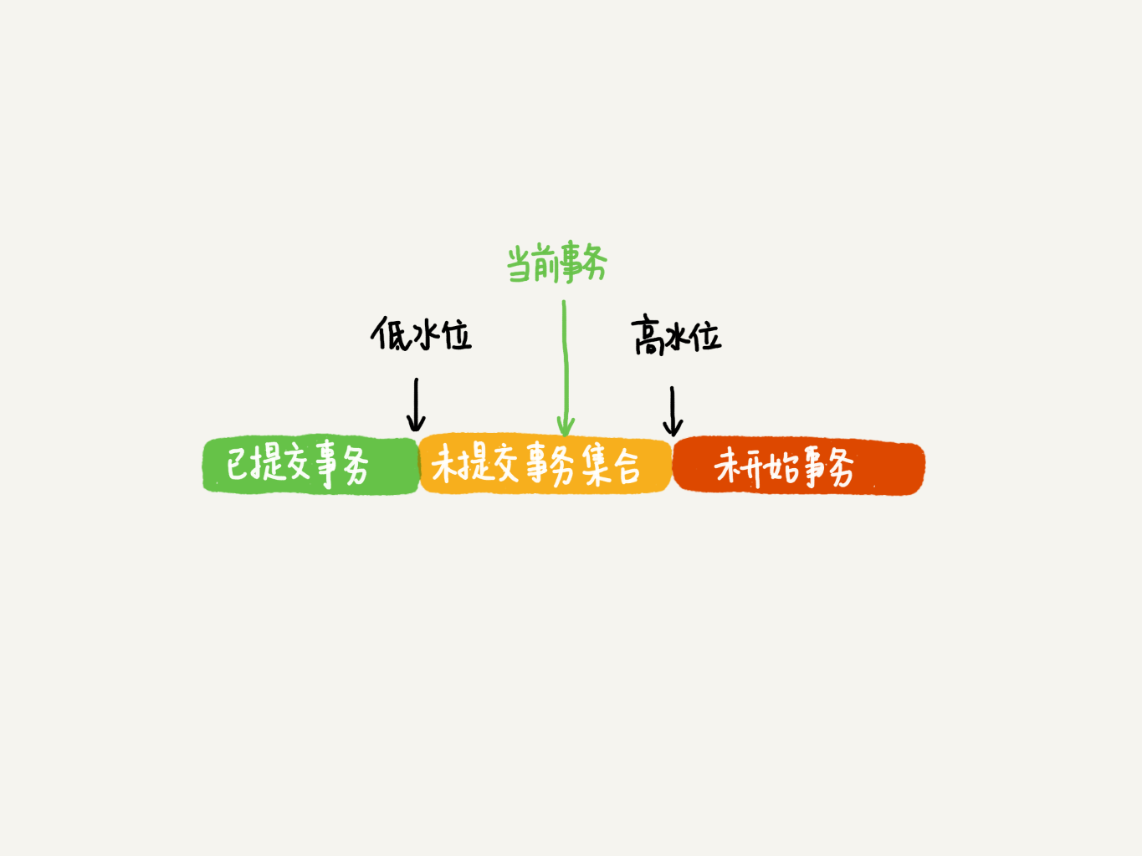
**ps：V1，V2，V3的箭头方向的就是回滚操作：需要undo log的支持，如果一个长事务一直不释放，会导致其他事务的undo log也不能释放，导致内存泄露**

## **MVCC原理：**

#### 事务的 一致性视图：开始事务时的活跃事务的数组 + 高低水位

**每次事务开始时，事务会创建一个数组，这个数组保存的是事务开始时，活跃的事务（还没有提交的事务）。数组中最大的值就是高水位，最小的值就是低水位。**

**数据版本的可见性，就是基于： 一致性视图 + 数据的 row trxId**



#### **读规则：**

1: row trx\_id在绿色区域，普通读可以读到（当前事务启动前提交）

2: row trx\_id在红色区域，普通读肯定读不到（当前事务启动后提交）

3: row trx\_id在黄色区域，包含两种情况

a. 若 row trx\_id 在数组中，表示这个版本是由还没提交的事务生成的，不可见；

b. 若 row trx\_id 不在数组中，表示这个版本是已经提交了的事务生成的，可见。

**一句话总结；就是普通读，只能读到 当前事务开启前，已经提交的事务的数据或者自己事务的数据。**

**原理：每次读取的时候，查看当前数据的版本，不满足一致性读，就往前回滚，直到回滚到满足的版本。**

#### **更新规则：**

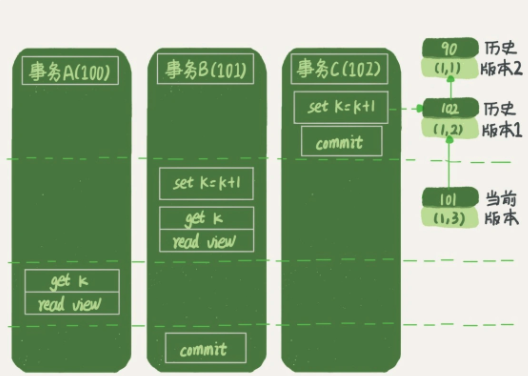
更新时，update语句时当前读，读到的是最新提交的版本的数据。称作当前读。

同样的，select 如果加读锁 或者 写锁，同样是当前读；

一句话总结：更新操作时，是当前读。/ 加锁时，是当前读。

#### 重要的点：

举例：事务A B C开启之前，不存在活跃的事务，A\_trxId = 100, B\_trxId = 101, C\_trxId = 102;



可以看到90是老的历史版本，在ABC开启之前，就已经提交；

###### 引入问题 ：对于101来说，102可以是没有提交的么？

回答：不可能，因为102如果没有提交，那就说明这行数据对应的锁还没有释放，事务B是不可能对这行数据进行操作生成101版本的数据的。

**所以：行数据的版本数据，只有最后一个版本可以是没有提交的。**

# 相关日志：

## RedoLog：

https://time.geekbang.org/column/article/68633

#### 涉及文件：RedoLog文件，表数据文件

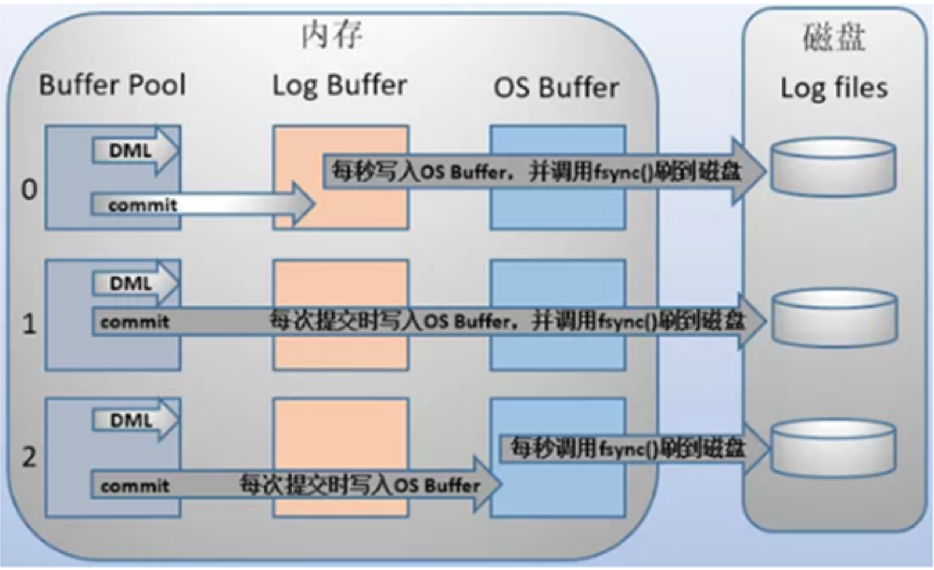
#### 背景：

数据提交后，为了性能，表数据文件并没有实时落盘，因为表数据的落盘时随机读写，非常的慢。这样就引来的问题，数据库时ACID的，如果没有落盘，数据库重启了，那么就会导致数据丢失，不满足D。所以引入了 RedoLog。

#### redoLog：

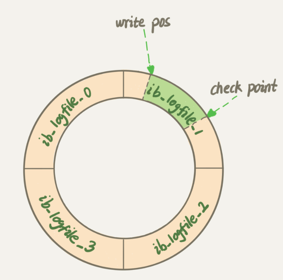
在mysql45讲中，举的例子很好，redoLog相当于小黑板，这个小黑板是实时记录的，并且每次commit时落盘。

配置成1:redoLog实时落盘，保证数据肯定不丢失



Redolog记录的就是数据文件的物理修改，rodolog是个环形文件追加写的形式，数据是顺序读写，所以每次cmmit后直接落盘，不会太费时间。数据库重启之后，可以通过RedoLog进行数据的恢复。

RedoLog有两个指针，写指针，和check指针，只有记录落盘之后，check才可以往前走。



## UndoLog：

#### 背景：

因为事务的A特性，要么全部执行，要么全部不执行，所以有一个undo的操作。每个DML语句都会记录一个UndoLog

#### 用途：

MVCC中，一致性视图的数据回滚

## BinLog：

#### 背景：

用于数据的备份，恢复

#### 原理：

每次commit的时候，都会记录事务中的DML语句（statment模式）/ 数据行（row模式）

## 两阶段提交：

注意区别两阶段锁：两阶段锁是指 手动加锁或者DML自动加锁 🡪 事务结束时释放锁。

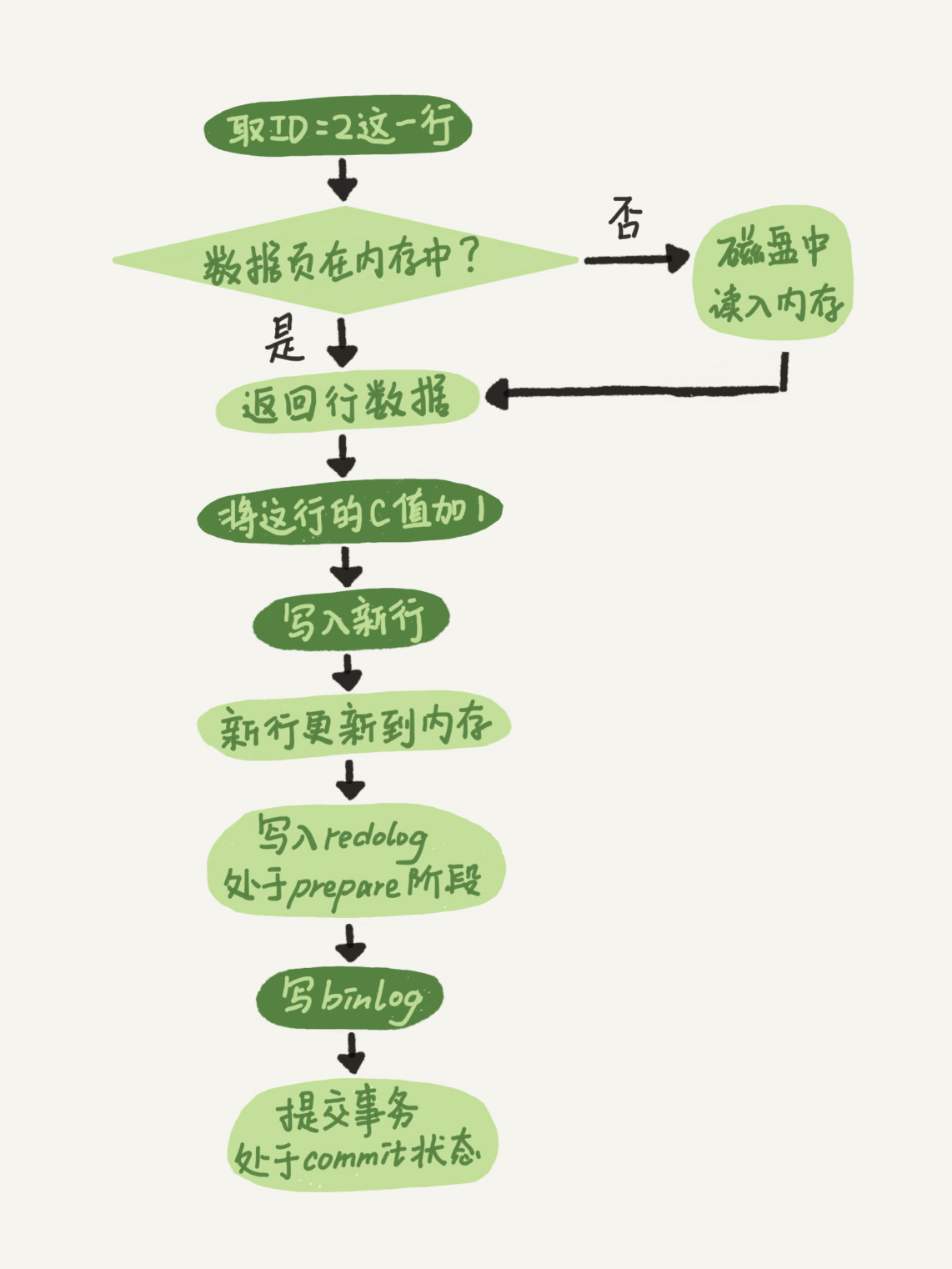
#### 背景：

在事务commit时，会记录redoLog和binlog，这两个log需要两阶段进行提交，不然就会出现 表数据 和 binlog不一致的情况；

比如先写redoLog再写binlog，写完redoLog后此时数据库重启，那么redoLog会恢复表数据，但是binlog没有记录。

或者先写binlog，再写redoLog，写完binlog后，数据库重启，那么redolog没有写，且表数据没有落盘，就会导致表数据文件没有记录而 binlog中有。

#### 解决办法：两阶段提交



##### 流程：

先写redolog，并设置 prepare状态，再写binlog，最后提交

崩溃恢复的时候，会按顺序扫描 redo log：

1:如果碰到既有 prepare、又有 commit 的 redo log，就直接提交；

2:如果碰到只有 parepare、而没有 commit 的 redo log，就拿着 XID 去 binlog 找对应的事务

2-1:如果binlog完整，就提交

2-2:如果binlog不完整，那么放弃

###### 问题1：MySQL 怎么知道 binlog 是完整的?

一个事务的 binlog 是有完整格式的：

statement 格式的 binlog，最后会有 COMMIT；

row 格式的 binlog，最后会有一个 XID event。

###### 问题2：redo log 和 binlog 是怎么关联起来的?

回答：它们有一个共同的数据字段，叫 XID。

# Mysql中的几个名词

<https://time.geekbang.org/column/article/69636>

回表：

在执行语句的时候，如果使用到了索引，为了避免在聚簇索引上全表扫描，会先去二级索引上 找到对应的索引值的主键，然后再拿着主键去聚簇索引去找自己需要的所有值，这个拿主键 主聚簇索引找数据的操作，就是回表；

PS：如果是范围操作，找到一个回一次，找到一个回一次。

索引覆盖：

如果我们需要的数据，在二级索引中就已经存在，那么就不需要进行回表操作，这个就叫索引覆盖；

最左前缀匹配：

使用索引的时候，不一定要和索引完全匹配，在单个索引的时候，可以只匹配前缀，通过可以定位到开始位置，然后进行扫描。

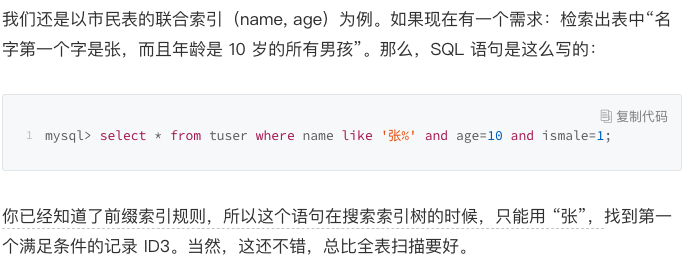
或者在使用复合索引的时候，没必要条件中包含 所有的子索引，包含前缀就可以使用这个覆盖索引；比如索引为ABC，我们A B或者A同样可以使用该索引。或者A B的前部分，也可以使用。

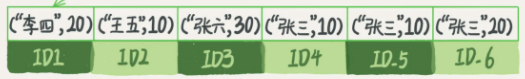
索引下推：

上一段我们说到满足**最左前缀原则**的时候，最左前缀可以用于在索引中定位记录。这时，你可能要问，那些不符合最左前缀的部分有啥用？

答：可以做条件过滤，减少回表的次数！

举例：



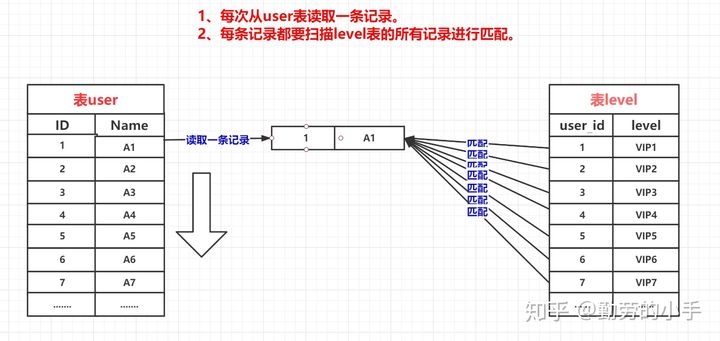


可以在索引遍历过程中，对索引中包含的字段先做判断，直接过滤掉不满足条件的记录，减少回表次数。

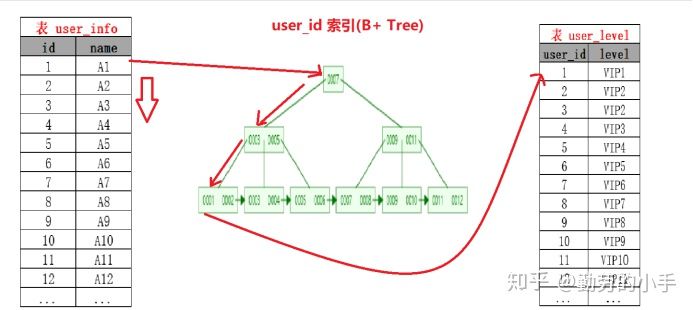
# Join原理

[数据库基础（七）Mysql Join算法原理 - 知乎 (zhihu.com)](https://zhuanlan.zhihu.com/p/54275505)

#### 一、Simple Nested-Loop Join（简单的嵌套循环连接）

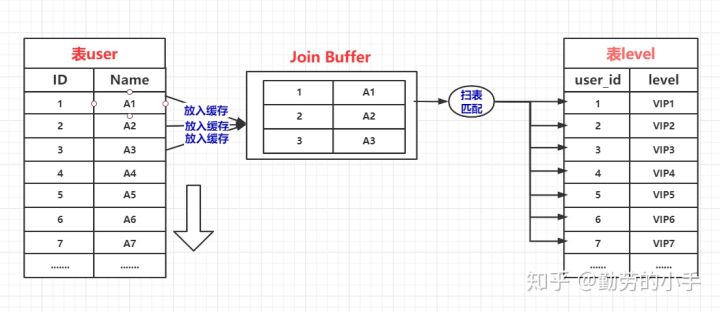


#### 二、Index Nested-Loop Join（索引嵌套循环连接）



**注意：使用Index Nested-Loop Join** **算法的前提是匹配的字段必须建立了索引。**

#### 三Block Nested-Loop Join（缓存块嵌套循环连接）



# 排序原理

详细看mysql45讲orderby原理

**using filesort**

Order by排序是在内存中进行排序的，需要使用到sort\_buffer，如果sort\_buffer不够，就需要使用文件来辅助排序

内存中排序的流程：

1：初始话sort\_buffer

2：从二级索引中取出第一个满足的数据，拿到主键id

3：使用主键id 回表取出需要查的字段，放入sort\_buffer中

4：从二级索引取下一个记录的主键id

5：重复3，4，指导取出所有符合查询条件的数据

6：对sort\_buffer中的数据按照需要排序的字段进行排序

7：将排序后的结果返回给客户端

 如果sort\_buffer的容量不够的话（sort\_buffer\_size），那此时就需要使用临时文件辅助排序；

外部文件排序使用 归并排序

如果单行长度太大，那么sort\_buffer就会因为太大放不下太多行，导致使用文件排序，那么mysql有优化使用**rowid排序**，也就是只取出排序字段和 主键放到sort\_buffer中，排序完成后，回表补全结果集；

# 临时表

MySQL在以下几种情况会创建临时表：

**1、UNION查询；**  
**7、FROM中的子查询；**  
4、join 包括not in、exist等

临时表分为磁盘临时表，和内存临时表，磁盘临时表如果很大，会把IO打满。

# 基础：

## 1:数值类型

tinyInt：1Byte

smallInt：2Byte

mediumInt：3Byte

int：4Byte

bigInt：8Byte

注意：tinyint(M),int(M)中的M没有实际意义，因为范围不受M控制，只有在zerofill时才能有点用。

## 2:浮点数

|  |  |  |
| --- | --- | --- |
| **类型名称** | **说明** | **存储需求** |
| FLOAT | 单精度浮点数 | 4 个字节 |
| DOUBLE | 双精度浮点数 | 8 个字节 |
| DECIMAL (M, D)，DEC | 压缩的“严格”定点数 | M+2 个字节 |

浮点类型和定点类型都可以用(M, D)来表示，其中M称为精度，表示总共的位数；D称为标度，表示小数的位数。

浮点数类型的取值范围为 M（1～255）和 D（1～30，且不能大于 M-2），分别表示显示宽度和小数位数。M 和 D 在 FLOAT 和DOUBLE 中是可选的，FLOAT 和 DOUBLE 类型将被保存为硬件所支持的最大精度。DECIMAL 的默认 D 值为 0、M 值为 10。

DECIMAL 类型不同于 FLOAT 和 DOUBLE。DOUBLE 实际上是以字符串的形式存放的，DECIMAL 可能的最大取值范围与 DOUBLE 相同，但是有效的取值范围由 M 和 D 决定。如果改变 M 而固定 D，则取值范围将随 M 的变大而变大。

从上表中可以看到，DECIMAL 的存储空间并不是固定的，而由精度值 M 决定，占用 M+2 个字节。

FLOAT 类型的取值范围如下：

* 有符号的取值范围：-3.402823466E+38～-1.175494351E-38。
* 无符号的取值范围：0 和 -1.175494351E-38～-3.402823466E+38。

DOUBLE 类型的取值范围如下：

* 有符号的取值范围：-1.7976931348623157E+308～-2.2250738585072014E-308。
* 无符号的取值范围：0 和 -2.2250738585072014E-308～-1.7976931348623157E+308。

提示：不论是定点还是浮点类型，如果用户指定的精度超出精度范围，则会四舍五入进行处理。

最后再强调一下：在 MySQL 中，定点数以字符串形式存储，在对精度要求比较高的时候（如货币、科学数据），使用 DECIMAL 的类型比较好，另外两个浮点数进行减法和比较运算时也容易出问题，所以在使用浮点数时需要注意，并尽量避免做浮点数比较。

## 2:字符类型

char(M)：固定长度，M代表最大字符数，M最大255个**字符**

varchar(M)：变长，M代表最大字符数，varchar最大支持65535-2个**字节**，其中2Byte个需要存储字符长度（字符小于255时，使用1Byte）

text：变长，最大支持65535-2个**字节，**2Byte个需要存储字符长度

差别点：

char存储数据会把 字符串之后的空格截取掉。

varchar需要额外的1-2个字节保存字符的个数

char适合固定长度数据存储：身份证，手机号，md5等

varchar适合变长数据存储。

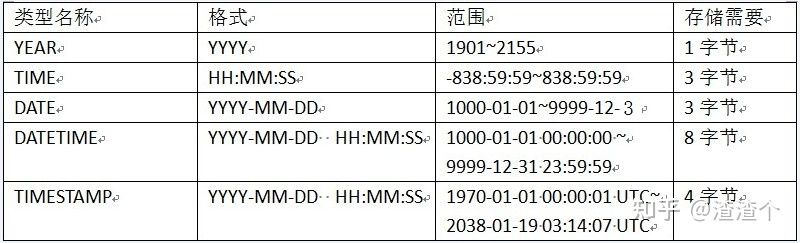
按照查询速度： char最快， varchar次之，text最慢。

一个面试题：

我们倘若用VARCHAR(5)和VARCHAR(200)来存储'hello'，我们知道这两者的空间开销是一样的。那么我们可以让VARCHAR的长度始终保持很大吗？使用更短的列有什么优势吗？

事实证明有很大的优势。更长的列会消耗更多的内存，因为MySQL通常会分配固定大小的内存块来保存内部值（varchar200 就会大小为200的内存，varchar 5就会分配大小为5的内存）。尤其是使用内存临时表进行排序或操作时会特别糟糕。在利用磁盘临时表进行排序时也同样糟糕。

## 时间类型：



date保存类似生日的数据，精确到日

timestamp和datetime区别：

1：timestamp和datetime表示的范围不同，占用的字节大小不同

2：timestamp数据会随着mysql配置的时区变化而变化，datatime时间不会变化，存进去的值不会变

3：timestamp更轻量，索引相对datetime更快。

4：存进去的是NULL，timestamp会自动储存当前时间，而 datetime会储存 NULL。

重要！：其实还可以使用int/bigint去存储时间戳，占用4/8Byte，效率高

结论 在InnoDB存储引擎下，通过时间排序，性能bigint > timestamp > datetime

# 范式

<https://www.zhihu.com/question/24696366>

第一范式：

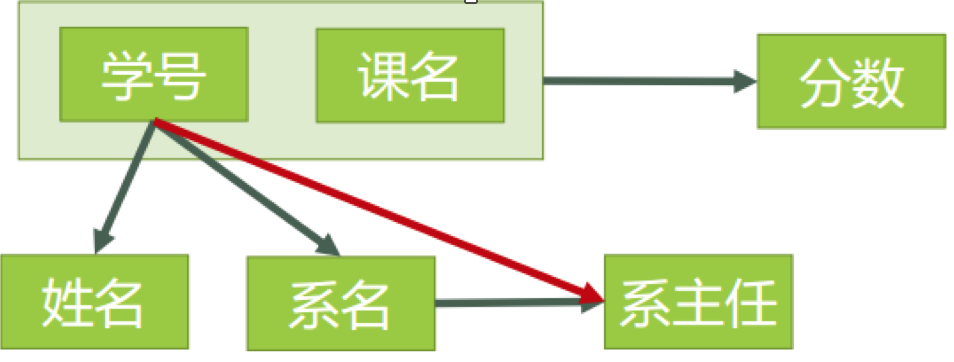
数据项不可分割



第二范式：

满足第一范式，且不存在 非主属性对于 码的部分函数依赖





主码为 （学号+课名）：存在 非主属性（姓名）（系名）部分函数依赖于 主属性（学号）

第三范式：

满足第二范式，且不存在 非主属性对于码的传递依赖



主码：（学号，姓名）

完全依赖：（学号，姓名）-> 系名

完全依赖：系名 -> 系主任

所以存在 非主属性对于 码的传递依赖，不符合3NF

BCNF：

满足第三范式，且不存在 主属性对于码的部分+传递函数依赖



码：（仓库，物品）（管理员，物品）

主属性：仓库，管理员，物品

非主属性：数量

已知函数依赖集：仓库名 → 管理员，管理员 → 仓库名，（仓库名，物品名）→ 数量

## 基础知识：

码，主属性，函数依赖，部分函数依赖

函数依赖：X 可以唯一确认一个 Y，那么X ->Y; Y函数依赖X

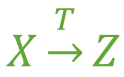
完全函数依赖：X是组合的（A，B），X少了任何一个，依赖不成立，那就是完全函数依赖



部分函数依赖：X是组合的（A，B），X少了一个，依赖依然成立，那就是部分函数依赖



传递函数依赖：Z依赖于Y，Y依赖于X，那么Z传递依赖



码：

设 K 为某表中的一个属性或属性组，若除 K 之外的所有属性都**完全函数依赖**于 K（这个“完全”不要漏了），那么我们称 K 为**候选码**，简称为**码**。

主属性：

包含任何一个 码中的属性，称为主属性

# 问题定位

### 慢查询：

<https://www.51cto.com/article/618217.html>

###### 1:日志：

1:开启慢查询日志

2:查看日志，找到时间比较长的 SQL

3:使用explain分析SQL

###### 2: show processlist

此办法可以看到正在连接的时间比较久的，且当前状态活跃的，查看一下状态，然后再用explain命令分析

### CPU高：

<https://cloud.tencent.com/developer/article/1677706>

###### 1:数据量大

show full processlist 很显眼：执行时间比较久+活跃状态的连接都比较可疑

###### 2:高并发

show full processlist中活跃的 线程数量很高

<https://segmentfault.com/a/1190000023449302>

###### 3:通过top –Hp pid直接找到cpu占用高的 pid

通过select 相关表，过滤OS\_id，kill 对应的processlist中的id

### IO高：

<https://blog.csdn.net/a315157973/article/details/104324009>

###### 1:数据量大，IO过多

###### 2:回表过多

###### 3:使用了磁盘临时表

###### 4:使用了文件排序

# 执行计划

<https://juejin.cn/post/6844903545607553037>

<https://www.cnblogs.com/ggjucheng/archive/2012/11/11/2765237.html>

对sql语句进行分析，从而知道开发人员进行优化



### Id：

id相同时，从上到下按顺序执行

id不同时，**越大的越先执行**

### select\_type:

作用：标记查询的类型

sample:简单的查询

primary：查询包含子查询，外层查询为primary

derived: from后面的子查询

subquery：select 字段中，或者 where中使用的 子查询

union：被union的查询

dependent union：会收到外部查询影响的union语句

dependent subquery：会收到外部影响的subquery

###### type：

作用：表示Mysql找到所需行的方式（访问类型）

https://pic002.cnblogs.com/images/2012/360373/2012111116263147.png

由左至右，由最差到最好

ALL：需要所有的行（没有用到索引）

index：扫描了所有的 索引的行

range：范围扫描了索引的行

ref：非唯一索引的 等值的所有行

eq\_ref：join查询中使用 唯一索引进行关联

const：唯一索引 等值查询一条行

system：访问常量

### possible\_keys，key， key\_len：

possible\_keys：可能会用到的索引

key：用到的索引

key\_len索引的长度

### ref：

连接时，等号右边的类型

### rows：

返回数据量的估算值

### extra：

作用：额外显示信息

using filesort：使用了排序，可能是内部排序，**或者文件排序，耗时！需要优化**

using temporary：使用到了临时表，尽量要避免

using index：使用到了索引覆盖，好事

using index condition：使用到了索引下推，好事

using where：server层使用where条件来过滤结果集

# Mysql优化

### 1:减少存储数据的量，尽可能的使用小的数据

能用thinyint不用smallint/mediumint/int/bigint

能用char不用varchar

能用enum不用string

控制索引的数量，以及索引的大小

### 2: 减少返回数据量

select 时需要的字段，而不是select \*

### 3:使用索引加快查询速度

1:不要对 条件使用表达式，函数，强制类型转化，这样会使得索引失效；

2:关联查时最好使用索引项进行连接

3:避免较少回表次数，尽量要做到覆盖索引，索引下推

3:不要对没有索引的字段做排序，对于where和orderby不同的字段，可以做联合索引

5:like要符合最左前缀匹配原则，or前后要同时使用索引

其他：

1:当知道返回的个数的时候，使用limit可以提升速度

2:能使用union all 不使用union，避免临时表的产生

2:表连接最好不要超过三张表，笛卡尔积很耗时

4:连接时小表驱动大表。

# 以前的疑惑点

一次查询中可以使用多个 索引么

<https://blog.csdn.net/liujun19921020/article/details/103215701>

如果没有使用索引，会锁全表么

会的

使用了非聚簇索引，锁的是啥？非聚簇索引的叶子节点？还是对应的聚簇索引？

<https://blog.csdn.net/qq_44766883/article/details/105879308>

# 好的文章

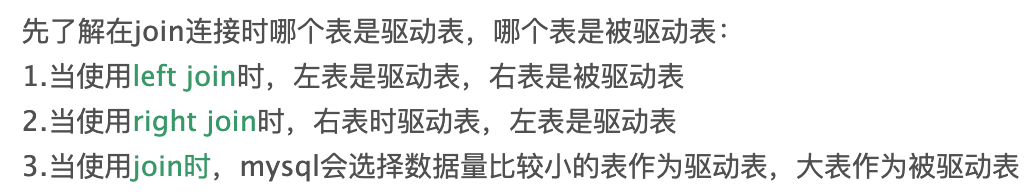
<https://tech.meituan.com/2014/08/20/innodb-lock.html>

# 刷题：

<https://cloud.tencent.com/developer/article/1814683>

# 补充：

### 表连接时 一定要小表驱动大表。



### 多个表join：

SELECT \*

FROM (aa LEFT JOIN bb ON aa.a = bb.a )

LEFT JOIN cc ON cc.a = bb.a;

<https://blog.51cto.com/u_15080022/4368104>

### Having的使用：

配合groupBy聚合函数，然后根绝 聚合内容过滤的

Having sum(a) > 10

### Limit的优化问题

Delete 批量删除1000000条数据，limit 和 orderby的使用（删除最老的）

Select 大量数据:

Select \* from table limit 90000, 100 order by id；这样会查出所有数据，然后取出90000开始的100条，其他的全删掉

优化，使用索引，避免全表扫描：

Select \* From table Where ID>=(

Select ID From table order by ID limit 90000,1

) order by ID limit 100;

### 层高的计算

Mysql的B+树中节点的大小16K

非叶子节点的 数据为：指针和 索引的交替值

指针6Byte，索引如果是bigint，那么是8Byte；所以一个节点的 叶子节点个数为 16k/14 = 1170

聚簇索引 叶子节点都是数据，假设一条数据1k，那么一个节点有16条；

那么两层的聚簇索引，可以有1170\*16条数据，大约2w条

三层的聚簇索引，可以用1170\*1170\*16 = 2000W条

### 对于索引的创建

如果区分度不大的字段，比如说性别，不要加索引。因为思考一下，性别的索引，需要扫描一半的 B+树的叶子，然后一半的回表次数。也就是说，命中率不高。