Mysql 插入一条数据的过程

插入流程：编译器解析-> 优化器根据采样和统计涉及字段的cardinality确定是走索引还是全表扫描->**写undo log**，记录版本和回滚指针-> 如果不是唯一索引，记录到change buffer等待刷入磁盘，否则是从磁盘读入内存校验是否已存在->执行器拿到，行版本+1，**写入redo log**->**执行器生成bin log buffer**->**两阶段提交prepare阶段redo刷盘到磁盘**，**commit阶段bin log刷盘（binlog cache-> binlog）**；因为是插入操作，事务结束以后可以删除undo log，否则更新和删除的话需要放到待清理列表判断是否上undo上一个版本还有事务在使用

### 2pc redolog +binlog 在 update 时如何操作

[mysql2阶段提交具体实现\_Mysql两阶段提交\_叶洛曦的博客-CSDN博客](https://blog.csdn.net/weixin_33150295/article/details/113021023)

[mysql的两阶段提交协议 - zengkefu - 博客园 (cnblogs.com)](https://www.cnblogs.com/zengkefu/p/5674863.html)

两阶段提交

update test set age = age +1 where id = 1 当执行这条语句的时候，我们看看它的内部流程

* 执行器先去引擎找id=1的这一行，id是主键引擎直接用树搜索找到这一行。如果这一行的数据页在内存中，就直接返回给执行器，否则就需要从磁盘读取然后在返回。
* 执行器拿到引擎提供的行数据，给age字段加1，比如原来是9，现在就是10，得到新的一行数据，在调用引擎接口写入这行数据
* 引擎将新的数据行更新到内存中，同时将这个更新操作记录到redo log，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
* 执行器生成这个操作的binlog,并把binlog写入磁盘
* 执行器调用引擎接口，引擎把刚刚写的redo log改成提交commit状态，更新完成。
* 最后三步比较绕，将redo log的写入拆成了两个步骤prepare和commit,这就是两阶段提交

### 手动实现事务提交

### 

**Set autocommit = 0; 设置禁止自动提交事务**

**Begin; 事物开始**

**Update inseert delete ...**

**Commit; 事物结束**

### 事务是怎么实现的、机制

事务：<https://www.cnblogs.com/superming/p/13368771.html>

Redolog 和 binlog： <https://blog.csdn.net/lzhcoder/article/details/88814364>

Redolog写入磁盘： <https://blog.csdn.net/weihaodong0557/article/details/109747629>

Binlog写入：

https://blog.csdn.net/weihaodong0557/article/details/109747541?utm\_medium=distribute.pc\_feed\_404.none-task-blog-2~default~BlogCommendFromMachineLearnPai2~default-1.nonecase&depth\_1-utm\_source=distribute.pc\_feed\_404.none-task-blog-2~default~BlogCommendFromMachineLearnPai2~default-1.nonecas

## **undo log 原子性**

在说明原子性原理之前，首先介绍一下MySQL的事务日志。MySQL的日志有很多种，如二进制日志、错误日志、查询日志、慢查询日志等，此外InnoDB存储引擎还提供了两种事务日志：redo log(重做日志)和undo log(回滚日志)。其中redo log用于保证事务持久性；undo log则是事务原子性和隔离性实现的基础。

下面说回undo log。实现原子性的关键，是当事务回滚时能够撤销所有已经成功执行的sql语句。**InnoDB实现回滚，靠的是undo log：当事务对数据库进行修改时，InnoDB会生成对应的undo log；如果事务执行失败或调用了rollback，导致事务需要回滚，便可以利用undo log中的信息将数据回滚到修改之前的样子。**

undo log属于逻辑日志，它记录的是sql执行相关的信息。当发生回滚时，InnoDB会根据undo log的内容做与之前相反的工作：对于每个insert，回滚时会执行delete；对于每个delete，回滚时会执行insert；对于每个update，回滚时会执行一个相反的update，把数据改回去。

以update操作为例：当事务执行update时，其生成的undo log中会包含被修改行的主键(以便知道修改了哪些行)、修改了哪些列、这些列在修改前后的值等信息，回滚时便可以使用这些信息将数据还原到update之前的状态。

### redo log 持久性

redo log和undo log都属于InnoDB的事务日志。下面先聊一下redo log存在的背景。

InnoDB作为MySQL的存储引擎，数据是存放在磁盘中的，但如果每次读写数据都需要磁盘IO，效率会很低。为此，InnoDB提供了缓存(Buffer Pool)，Buffer Pool中包含了磁盘中部分数据页的映射，作为访问数据库的缓冲：当从数据库读取数据时，会首先从Buffer Pool中读取，如果Buffer Pool中没有，则从磁盘读取后放入Buffer Pool；当向数据库写入数据时，会首先写入Buffer Pool，Buffer Pool中修改的数据会定期刷新到磁盘中（这一过程称为刷脏）。

Buffer Pool的使用大大提高了读写数据的效率，但是也带了新的问题：如果MySQL宕机，而此时Buffer Pool中修改的数据还没有刷新到磁盘，就会导致数据的丢失，事务的持久性无法保证。

于是，redo log被引入来解决这个问题：当数据修改时，除了修改Buffer Pool中的数据，还会在redo log记录这次操作；当事务提交时，会调用fsync接口对redo log进行刷盘。如果MySQL宕机，重启时可以读取redo log中的数据，对数据库进行恢复。**redo log采用的是WAL（Write-ahead logging，预写式日志），所有修改先写入日志，再更新到Buffer Pool**，保证了数据不会因MySQL宕机而丢失，从而满足了持久性要求。

既然redo log也需要在事务提交时将日志写入磁盘，为什么它比直接将Buffer Pool中修改的数据写入磁盘(即刷脏)要快呢？主要有以下两方面的原因：

（1）刷脏是随机IO，因为每次修改的数据位置随机，但写redo log是追加操作，属于顺序IO。

（2）刷脏是以数据页（Page）为单位的，MySQL默认页大小是16KB，一个Page上一个小修改都要整页写入；而redo log中只包含真正需要写入的部分，无效IO大大减少。

#### Redolog写入

1. redo log不是随着事务的提交才写入的，而是在事务的执行过程中，便开始写入redo 中。具体 的落盘策略可以进行配置 。防止在发生故障的时间点，尚有脏页未写入磁盘，在重启mysql服务的时候，根据redo log进行重做，从而达到事务的未入磁盘数据进行持久化这一特性。RedoLog是为了实现事务的持久性而出现的产物

一旦事务成功提交且数据持久化落盘之后，此时Redo log中的对应事务数据记录就失去了意义，所 以Redo log的写入是日志文件循环写入的·

Redolog什么时候写入磁盘(事务提交第一阶段 preparement 阶段)

https://blog.csdn.net/weihaodong0557/article/details/109747629

**Redolog binlog 区别**

* redo log是属于innoDB层面，binlog属于MySQL Server层面的，这样在数据库用别的存储引擎时可以达到一致性的要求。
* redo log是物理日志，记录该数据页更新的内容；binlog是逻辑日志，记录的是这个更新语句的原始逻辑 （redolog 记录值 binlog记录sql语句）
* redo log是循环写，日志空间大小固定；binlog是追加写，是指一份写到一定大小的时候会更换下一个文件，不会覆盖。
* binlog可以作为恢复数据使用，主从复制搭建，redo log作为异常宕机或者介质故障后的数据恢复使用。

**隔离性：**

**锁+MVCC**

* 原子性：语句要么全执行，要么全不执行，是事务最核心的特性，事务本身就是以原子性来定义的；实现主要基于undo log
* 持久性：保证事务提交后不会因为宕机等原因导致数据丢失；实现主要基于redo log
* 隔离性：保证事务执行尽可能不受其他事务影响；InnoDB默认的隔离级别是RR，RR的实现主要基于锁机制（包含next-key lock）、MVCC（包括数据的隐藏列、基于undo log的版本链、ReadView）
* 一致性：事务追求的最终目标，一致性的实现既需要数据库层面的保障，也需要应用层面的保障

### 索引的原理，索引的分类

原理

B 树和 B+树有什么不同呢？

第一，B 树一个节点里存的是数据，而 B+树存储的是索引（地址），所以 B 树里一个节点存不了很多个数据，但是 B+树一个节点能存很多索引，B+树叶子节点存所有的数据。

第二，B+树的叶子节点是数据阶段用了一个链表串联起来，便于范围查找。

# 数据库三范式

数据库表的每一列都是不可分割的基本数据项

**第二范式(确保表中的每列都和主键相关)**

**第三范式(确保每列都和主键列直接相关,而不是间接相关)**

Mysql 插入一条数据的过程

插入流程：编译器解析-> 优化器根据采样和统计涉及字段的cardinality确定是走索引还是全表扫描->写undo log，记录版本和回滚指针-> 如果不是唯一索引，记录到change buffer等待刷入磁盘，否则是从磁盘读入内存校验是否已存在->执行器拿到，行版本+1，写入redo log->执行器生成bin log， commit->**两阶段提交**prepare阶段redo刷盘，commit阶段bin log刷盘；因为是插入操作，事务结束以后可以删除undo log，否则更新和删除的话需要放到待清理列表判断是否上undo上一个版本还有事务在使用

1. **Sql 语句：只有一列 name，里面有重复的，怎么把重复的名字找出来**
2. **（Group by...Having）**

-- 有readerid，bookid，tagid，查出指定bookid最热门的10个标签，readerid越多，越热门

SELECT TAGID,COUNT(READID) FROM BYTEDANCE\_READ

WHERE BOOKID = 1

GROUP BY TAGID

ORDER BY COUNT(READID) DESC

LIMIT 0,3;

MVCC

解决 读-写的问题 会有事务隔离级别相关问题脏读幻读等

**通过无锁解决读写并发问题**。

https://www.jianshu.com/p/8845ddca3b23

MVcc中

Undolog 有 insertundolog 可以随时删 updateundolog 不能轻易删除，因为有老版本需要这个回滚

Readview 获取快照度的快照在记录一下最活跃的事务号

几个版本号： 创建，最后一次操作，指向上个的指针，

Mysql锁相关问题

<https://zhuanlan.zhihu.com/p/29150809/>

https://www.cnblogs.com/wintersoft/p/10787474.html

行锁

（innodb独有，走索引才会生效，开销大，加锁慢，并发度高会出现死锁）

适合并发量大的情况下 行锁是存储引擎级别的实现，Mysql没有

行锁实现：

共享锁（读，可以多个共享锁进来） **lock on share mode**

排它锁（写(读也可以加)，只能有一个排它锁存在） **for update**

意向共享锁/意向排它锁

事务加共享锁/排它锁前需要获取到该表的意向共享锁/意向排它锁

表锁

（基于mysql的锁不是基于存储引擎的锁，innodb中没有使用索引的行锁都会走表锁）

开销小，加锁快，并发度小，适合查询为主，并发量小的情况

MySQL行锁是针对于索引进行加锁，所以说一旦索引失效就不会走行锁就会默认变为表锁

### **锁类型**

**共享锁**(S锁):假设事务T1对数据A加上共享锁，那么事务T2**可以**读数据A，**不能**修改数据A。  
**排他锁**(X锁):假设事务T1对数据A加上共享锁，那么事务T2**不能**读数据A，**不能**修改数据A。  
我们通过update、delete等语句加上的锁都是行级别的锁。只有LOCK TABLE … READ和LOCK TABLE … WRITE才能申请表级别的锁。  
**意向共享锁**(IS锁):一个事务在获取（任何一行/或者全表）S锁之前，一定会先在所在的表上加IS锁。  
**意向排他锁**(IX锁):一个事务在获取（任何一行/或者全表）X锁之前，一定会先在所在的表上加IX锁。

**意向锁存在的目的?**

OK，这里说一下意向锁存在的目的。假设事务T1，用X锁来锁住了表上的几条记录，那么此时表上存在IX锁，即意向排他锁。那么此时事务T2要进行LOCK TABLE … WRITE的表级别锁的请求**，可以直接根据意向锁是否存在而判断是否有锁冲突**。

Mysql 读操作除了串行化(serializable)加共享锁(lock in share mode)，其他情况下都不会默认加锁，如果手动加行锁(共享或排他 lock in share mode/ for update)，走索引行锁生效，没走行锁，行锁升级为表锁。

InnoDB使用间隙锁的目的：

防止幻读，以满足相关隔离级别的要求；

满足恢复和复制的需要：

**间隙锁**在 RU RC失效 在RR S生效

For update 加锁

**InnoDB有三种行锁的算法：**

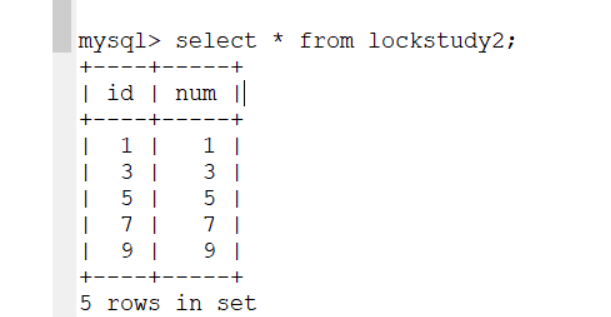
1，Record Lock：单个行记录上的锁。

2，Gap Lock：间隙锁，锁定一个范围，但不包括记录本身。GAP锁的目的，是为了防止同一事务的两次当前读，出现幻读的情况。

3，Next-Key Lock：1+2，锁定一个范围，并且锁定记录本身。对于行的查询，都是采用该方法，主要目的是解决幻读的问题。

<https://www.cnblogs.com/zhoujinyi/p/3435982.html>

Forupdate 测试：



**如果表中是有唯一索引，主键索引 primary key id 就锁单行 Record Lock**

1客户端： select \* from lockstudy2 where id = 5 for uodate;

锁的范围只是 5这个索引 如果执行 insert 4 6都可以执行

**如果表中是二级索引 锁的是范围 上一个索引到下一个索引的范围 Next-Key Lock**

锁的是 [3,7] 所有的数据 如果执行 delete 3 / 7 insert 4 6 都会被阻塞

Mysql 索引失效

https://www.jb51.net/article/176633.htm

1. 使用or左右两边不是一个索引，索引会失效
2. 使用 is null/is not null 在单个索引情况下会生效，在复合索引情况下会使索引失效
3. 使用 != <! >! <> 会使索引失效
4. 使用like %M 失效 M% 生效 %M% 需要使用覆盖索引才生效

Like %m% 时间复杂度不是 logN 是 N ，需要从二级索引树中华便利所有

Like m% 是logN ，遵循最佳左前缀原则，符合索引B+树查询结构.

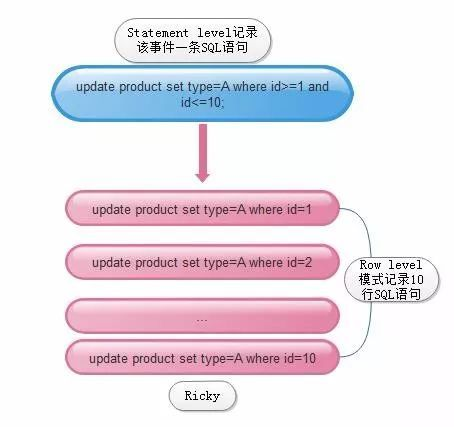
1. Order by 中 多个索引 asc desc 这种hi使索引失效
2. +不遵循最左前缀原则，跳跃使用索引会使部分索引失效
3. 索引使用内置函数 sum avg 等一定会失效
4. 索引使用算术操作 where a = 10-1 一定会失效
5. 隐士类型转换 比如 char类型不加双引号或单引号索引会失效
6. 索引列数据离散性不高，innodb会认为走索引还不如走全表（离散度： 数据不同种类越多，离散度越高）

Binlog 默认关闭

**binlog日志三种模式**

**ROW Level**

记录的方式是行，即如果批量修改数据，记录的不是批量修改的SQL语句事件，而是每条记录被更改的SQL语句，因此，ROW模式的binlog日志文件会变得很“重”。



优点：row level的binlog日志内容会非常清楚的记录下每一行数据被修改的细节。而且不会出现某些特定情况下存储过程或function，以及trigger的调用和触发器无法被正确复制的问题。

缺点：row level下，所有执行的语句当记录到日志中的时候，都以每行记录的修改来记录，这样可能会产生大量的日志内容，产生的binlog日志量是惊人的。批量修改几百万条数据，那么记录几百万行……

**Statement level(默认)**

记录每一条修改数据的SQL语句（批量修改时，记录的不是单条SQL语句，而是批量修改的SQL语句事件）。看上面的图解可以很好的理解row level和statement level两种模式的区别。

优点：statement模式记录的更改的SQ语句事件，并非每条更改记录，所以大大减少了binlog日志量，节约磁盘IO，提高性能。

缺点：statement level下对一些特殊功能的复制效果不是很好，比如：函数、存储过程的复制。由于row level是基于每一行的变化来记录的，所以不会出现类似问题

**Mixed**

实际上就是前两种模式的结合。在Mixed模式下，MySQL会根据执行的每一条具体的sql语句来区分对待记录的日志形式，也就是在Statement和Row之间选择一种。

企业场景如何选择binlog的模式

1、 如果生产中使用MySQL的特殊功能相对少（存储过程、触发器、函数）。选择默认的语句模式，Statement Level。

2、 如果生产中使用MySQL的特殊功能较多的，可以选择Mixed模式。

3、 如果生产中使用MySQL的特殊功能较多，又希望数据最大化一致，此时最好Row level模式；但是要注意，该模式的binlog非常“沉重”。

**查看binlog模式**

mysql> show global variables like "%binlog\_format%"; +---------------+-----------+

| Variable\_name | Value |

+---------------+-----------+

| binlog\_format | STATEMENT |

+---------------+-----------+

**配置binlog日志模式**

vim my.cnf（在[mysqld]模块中配置）

log-bin = /data/3306/mysql-bin

binlog\_format="STATEMENT"

#binlog\_format="ROW"

#binlog\_format="MIXED"

不重启，使配置在msyql中生效

SET global binlog\_format='STATEMENT';

Mysql 执行的顺序

1.from计算笛卡尔积。

2.on根据join\_condition过滤数据。

3.join添加相关外部表格数据。

4.where根据where\_condition过滤数据。

5.group by分组。

6.having根据having\_condition过滤数据。

7.Select 选择指定的列。

8.distinct指定列去重。

9.orderby按order\_by\_condition排序。

10.Limit 取出指定的记录量。

DDL DML DCL DQL

DQL : 查询 select

DCL: 事务

DML: 修改 insert delete update

DDL: 定义 create

主从复制 只记录 DML DDL

MVCC 2,3

间隙锁 3,4

面试题

**在数据库增删改查四种操作中，insert、delete和update都是会加排它锁(Exclusive Locks)的，而select只有显式声明才会加锁:**

table test

int a 1 字段 A 存一个1

int b 1 字段 B 存一个1

事务1

select \* from test; ------ 1 执行顺序 第一个执行

update set a = 2 b = 2 where a = 1; ---- 3

事务2

select \* from test; ------2

update set a = 3 b = 3 where a = 1; ----4

最后 a变成了几？

a变成了2

普通select都是快照，update时是当前读，所以改 a = 2时就会加锁， update a = 3就会阻塞 不能操作这行数据

**1.当前读, 读取的是最新版本, 并且**对读取的记录加锁, 阻塞其他事务同时改动相同记录，避免出现安全问题****

**2.当前读的实现方式：next-key锁(行记录锁+Gap间隙锁)**

1. **快照读，为了避免同一个事物不同时间读取的同一个数据不一样（不可重复读），所以一次事务中只在**第一次select**时生成快照版本，后续的查询都是在这个版本上进行，从而实现了**可重复读，说白了一个表 字段a = 0 这个时候把版本记录下来，然后后来我自己把a变为1，变为100 ，我读取还是读取 a=0的那个版本，就是Mvcc****

****图解：****

****白话文****

[mysql mvcc底层详解+图解\_promisessh的博客-CSDN博客](https://blog.csdn.net/promisessh/article/details/115385685)

****Readview + undolog 版本连 复杂****

[图解MySQL 8.0 中的MVCC原理\_尼尧Nier的博客-CSDN博客](https://blog.csdn.net/h380115990/article/details/107638370)

# **深入理解mysql四种隔离级别及底层实现原理（MVCC和锁）**

**这两个url 是重点**

**https://blog.csdn.net/weixin\_42130471/article/details/89142983**

https://www.cnblogs.com/wwcom123/p/10727194.html

读不可提交： 一个事务可以读到另一个事务没提交的

读可提交： 一个事务可以读到另一个事务提交的

可重复度， 通过快照读，可以重复读取一个事务中不同时候的操作，读出来的结果是一样的

序列化： 像序列化一样，所有的都加共享锁，读写互斥

# 为什么要二阶段提交mysql prepare commit

##### MySQL使用两阶段提交主要解决 binlog 和 InnoDB redo log 的数据一致性问题

## Mysql 查询慢的原因

[腾讯面试：一条SQL语句执行得很慢的原因有哪些？---不看后悔系列 - 知乎 (zhihu.com)](https://zhuanlan.zhihu.com/p/407400857?utm_source=wechat_timeline&utm_medium=social&utm_oi=1058644653334773760&utm_campaign=shareopn)

经常： 索引问题

偶尔：

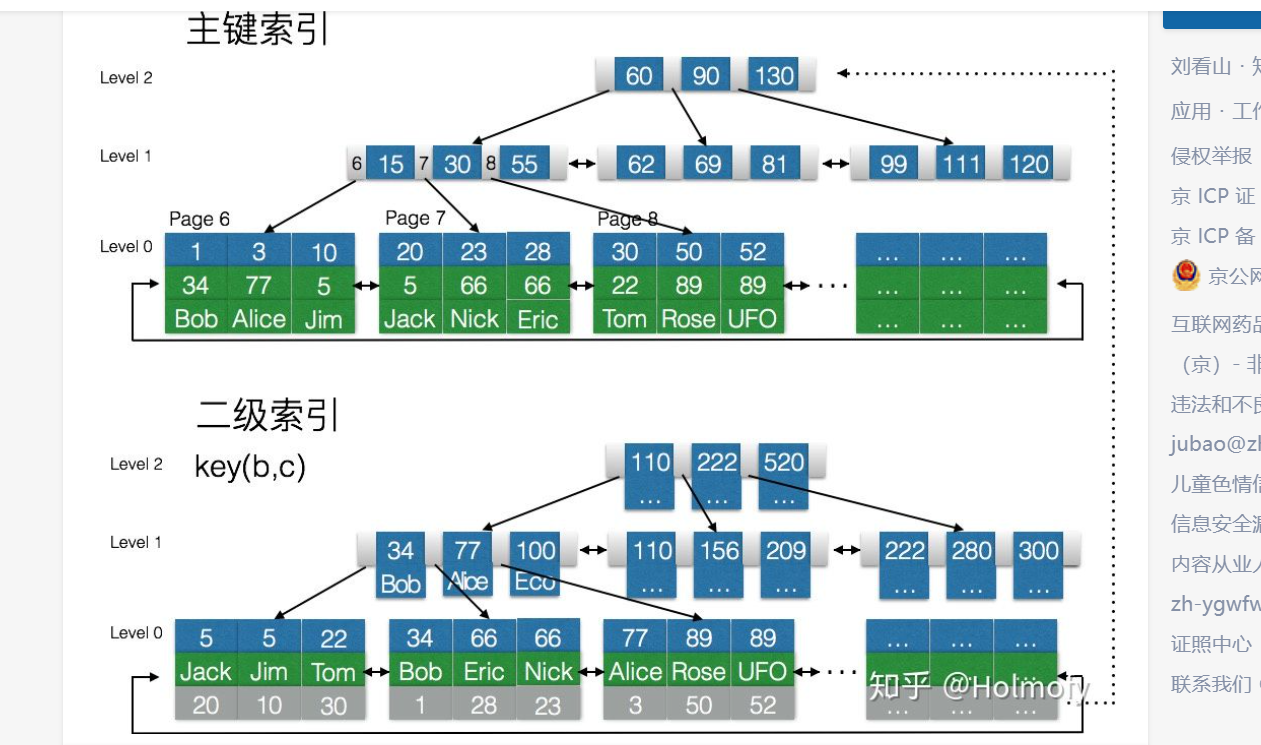
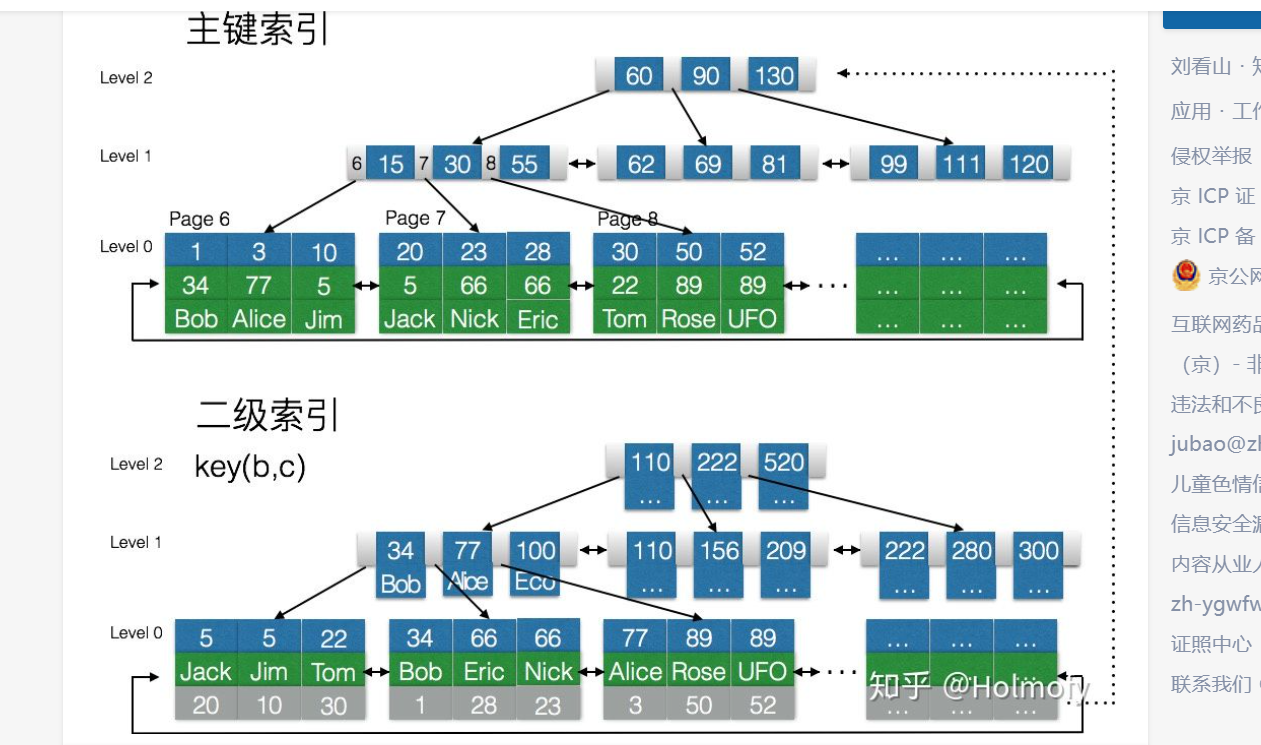
1. redolog 满了正在脏刷

1. 待操作行加锁了  **show processlist**这个命令来查看当前锁的状态

### Mysql索引底层

https://blog.csdn.net/weixin\_29737443/article/details/113190776

### Mysql联合索引底层

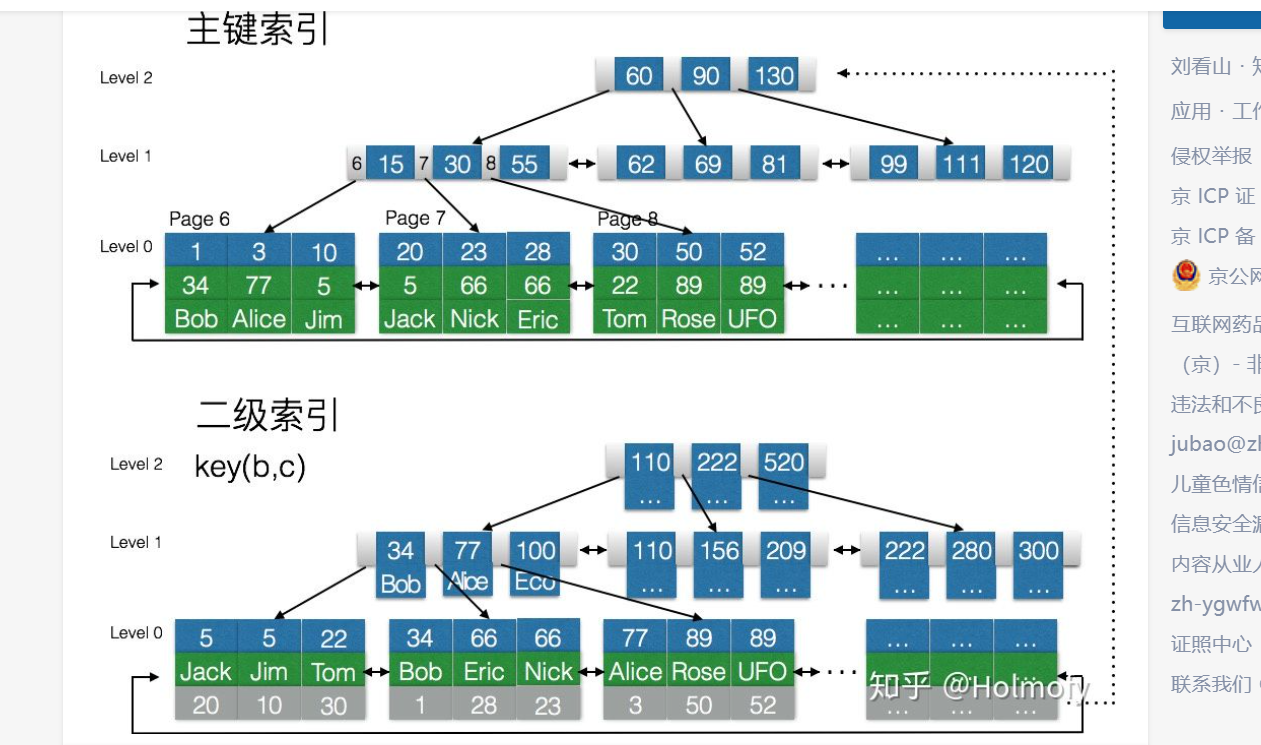


[mysql联合索引的数据结构\_cristianoxm的博客-CSDN博客\_mysql联合索引存储结构](https://blog.csdn.net/cristianoxm/article/details/107818084) 这个链接讲的详细

所以使用覆盖索引，可以直接命中联合索引所在b+树，不用回表到主索引树上

所以联合索引如果不遵循最佳左前缀原则，在联合索引B+树上，根本无法定位，因为每个索引的顺序就是 b,c

### %Like失效原因



由于B+树查找索引是按照值查找，你搞一个 %M，根本无法定位到具体哪一个索引，所以必然失效

# 1.2.2 高并发条件下的行锁表锁

既然聊到了索引，就不得不聊一聊锁，在InnoDB存储引擎中的行锁就是基于索引来实现的。而MySQL默认的锁是在MySQL Server层面的表锁。这种表锁如果在安全的角度看的确可以很好地解决高并发下数据一致性的问题。但是如果从执行效率的角度看就不是那么得当了。因此MySQL InnoDB引入了基于索引的行锁，这种行锁在索引失效或者没有索引的情况下会自动更替为MySQL Server层面的表锁。并且行锁也会根据索引类型的不同而锁定不同的行。在PRIMARY INDEX / UNIQUE INDEX情况下会执行RECORD LOCK算法，只会锁当前数据行，而如果是普通的INDEX则会执行NEXT-KEY LOCK算法，NEXT-KEY LOCK算法则相当于是GAP LOCK算法与RECORD LOCK算法的结合，他不仅仅会把当前索引所在的数据行进行上锁操作，而且还会对上一条数据到下一条数据的范围内的范围进行加锁操作。而这种行锁的根本作用一方面是为了解决幻读的问题，另一方便是为了解决并发UPDATE/INSERT/DELETE的问题。下面降将通过具体的例子论证这三种行锁算法的作用。

这里将通过Student表来进行验证，首先来查看一下Student表内的数据及索引情况。如下图1-4所示

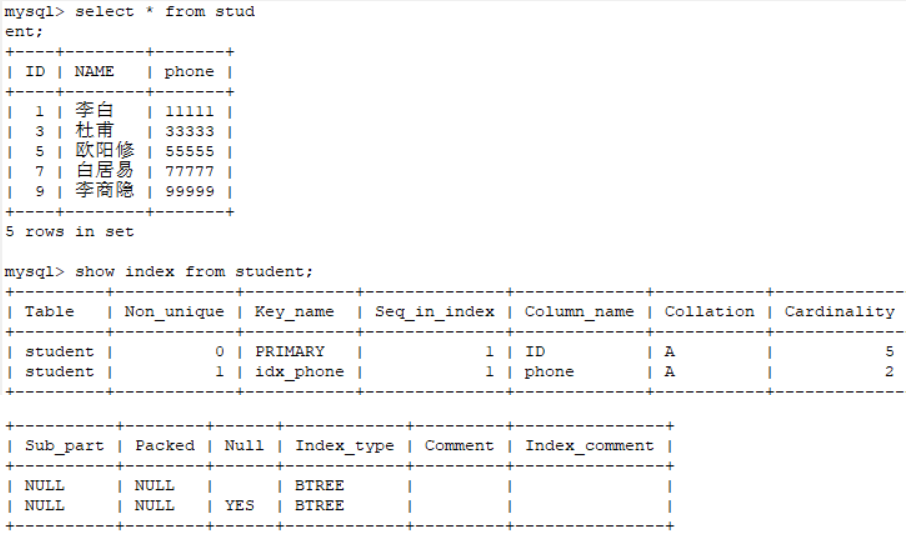


图1-4 Student表信息及索引信息

Fig.1-4 Student table information and index information

我们选择开启三条线程进行测试，首先进行PRIMARY KEY的测试。如下图1-5所示

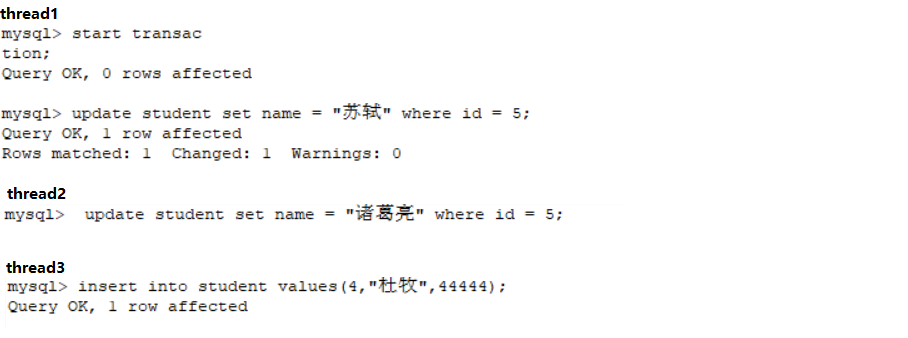


图1-5 主键索引下行锁的测试

Fig.1-5 Tests for primary key index downlocks

可以看到，线程1开启事务，执行UPDATA后并没有进行COMMIT操作，这个时候行锁执行的算法就是RECORD LOCK算法，在这种情况下，只有索引所在行会被上锁，其他行则不受影响。所以线程2想执行修改操作却一直没阻塞没有返回Query Ok的字样，线程3则可以正常的进行修改操作因为线程3没有对上锁的行进行操作。

接下来我们来测试普通索引下的行锁的情况。如下图1-6所示

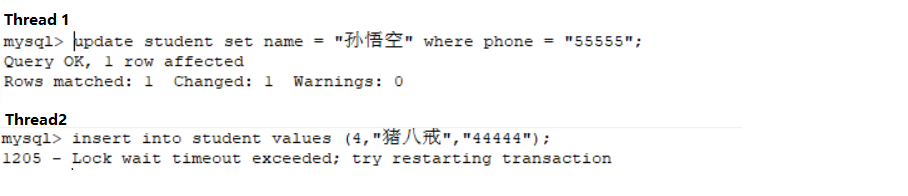


图1-6 普通索引下行锁的测试

Fig.1-5 Tests for normal key index downlocks

在这次测试中线程1对普通索引phone进行操作，这个时候则执行个NEXT-KEY LOCK算法，这个时候会对4,5,6,位置都进行一个上锁操作，所以线程2会返回超时

MySQL由于锁及索引的存在，保证了并发数据安全的问题，所以被广泛使用。

* 1. MySQL架构

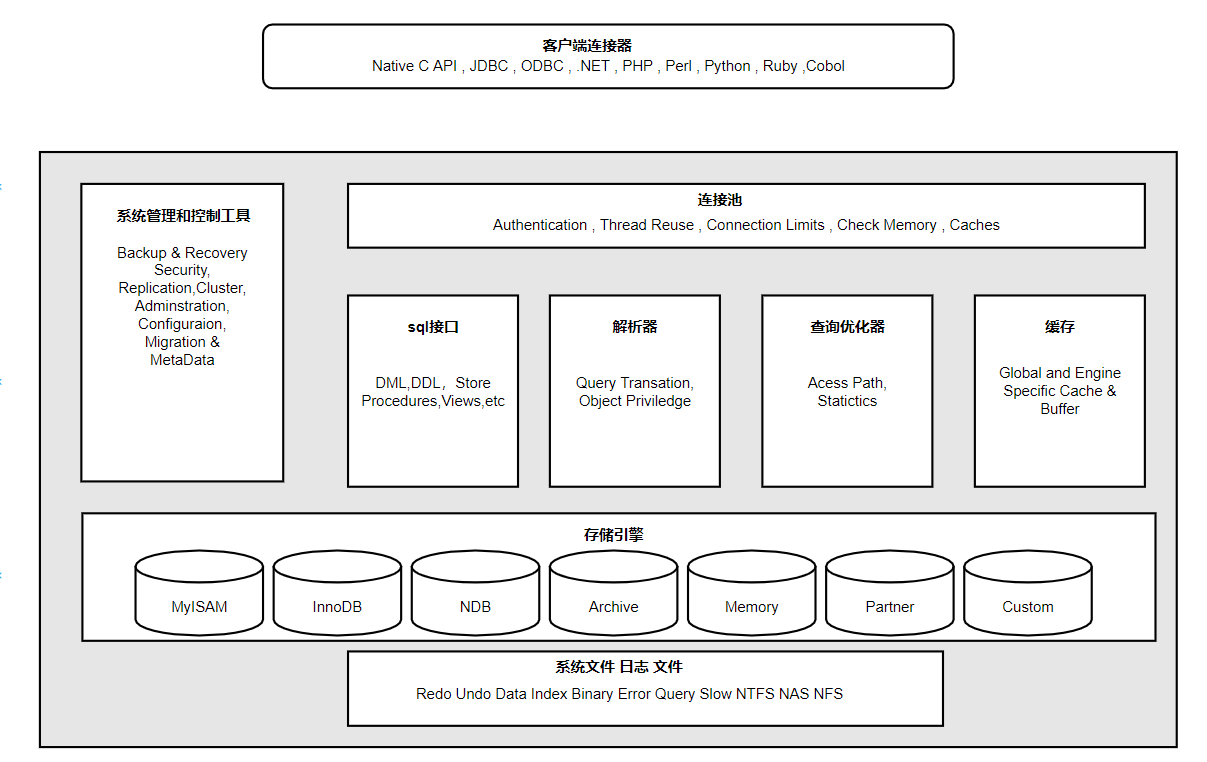


图1-1MySQL架构图

Fig.1-1 MySQL architecture diagram

在聊Innodb存储引擎之前，我们需要先来了解一下MySQL的整体架构，图1-1中可以看到，MySQL的架构层次可以看做三层。最上层是进行连接处理的地方，可以对MySQL进行授权认证，连接处理等功能。第二层主要负责处理MySQL大部分的服务功能，一般包括查询解析，查询优化，缓存，及一些内置函数比如日期，时间等，以及一些跨存储引擎的功能都会在这一层里进行实现。包括一些触发器，存储过程等。第三层就是存储引擎，包含了MyISAM,InnoDB等存储引擎。存储引擎主要负责MySQL数据的存储和提取，并且还有一些日志文件，通过这些可以实现数据的安全性。

* 1. Innodb存储引擎

1.2.1 B+树及聚簇索引

在本文中及本学习系统所研究使用的MySQL都是基于InnoDB存储引擎下的MySQL,采用的版本为5.6版本，其他的存储引擎如MyISAM本文不进行讨论。

InnoDB作为当下最主流的存储引擎，摒弃了MyISAM的Hash索引二采用了B+树作为存储的数据结构，B+树的数据结构如下图1-2所示。图中的为一个高度为二的B+树。B+树可以看做是平衡二叉树演变而来。B+树是采用二分查找(Binary Search)的算法思想来实现的一种数据结构，所以它每次查询的时间复杂度为O(Log n),b并且由于在叶子结点上都是一个双向链表组成，这种链表可以实现快速的范围查找。在B+树中，所有记录节点都是按照键值大小顺序放在当层的叶子结点上，所有非叶子结点上都只是索引，只有叶子结点上在存储了所有的值。

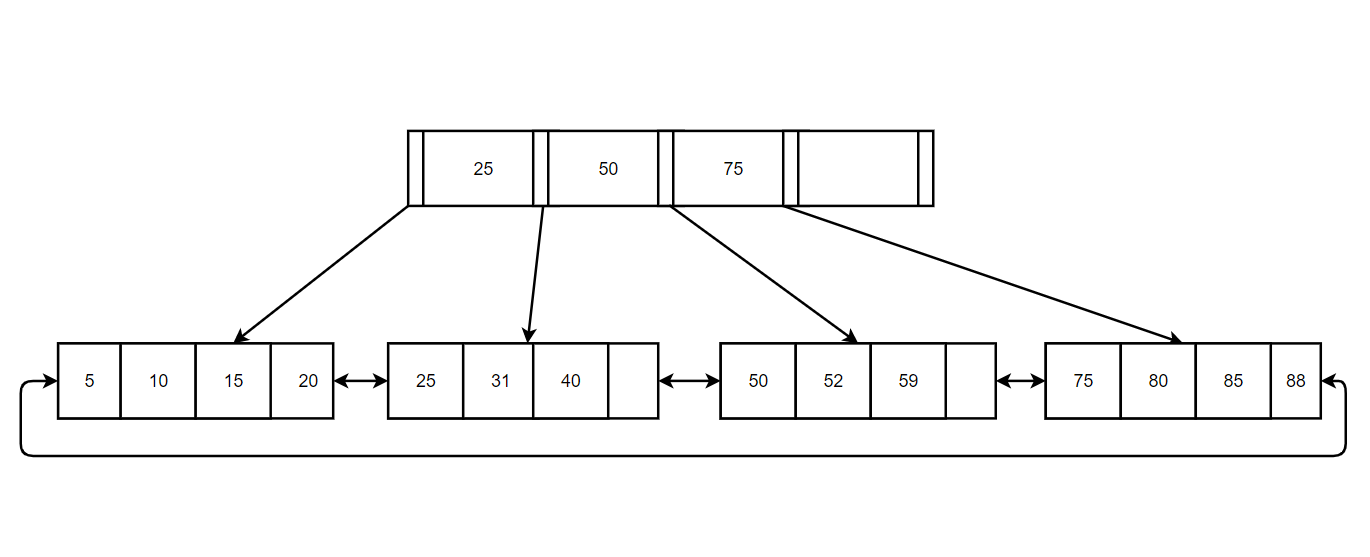


图1-2B+树数据结构

Fig.1-2 B+ Tree data structure

在MySQL InnoDB中B+树作为主要索引形式，并且由于在数据库中独特的高扇出性导致在MySQL数据库中一般的B+树的高度一般都是在2-4层。这也就是说每一次查找数据一般需要进行2-4次磁盘的IO操作。如果按照一般的磁盘，每一秒可以实现100次的IO来计算，2-4次IO操作的时间仅仅需要0.02 - 0.04秒就可以完成。

B+树在InnoDB具体的落地实现为聚簇索引和非聚簇索引，这也是Innodb存储引擎的灵魂所在。通过在聚簇索引叶子结点存储全部数据，其他非叶子节点及非聚簇索引上存放的都是索引的方式，大量的降低单个索引存储数据的数量，增加查询效率。并且在MySQL5.6之后得版本引入了索引下推，使得在使用非聚簇索引时不需要二次IO，通过回表的方式经过一次的IO就可以实现数据的查找。

下图1-3即为一组B+树在MySQL InnoDB存储引擎下的实际例子。

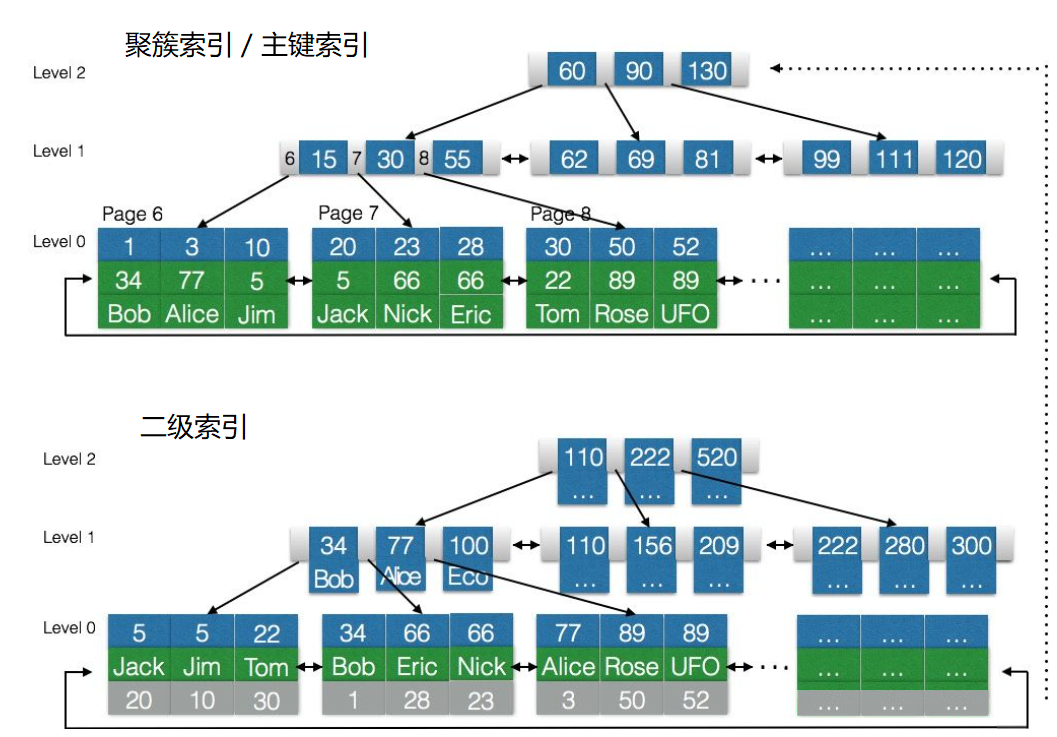


图1-3B+树实际应用

Fig.1-3 B+ Tree pratical application

在这里需要注意一点的是，如上图1-3中的非聚簇索引（二级索引）中，查找的顺序必须按照索引创建的顺序进行查找，即遵循最左前缀原则进行查找，否则会索引失效，从而进行全表查找，效率低下。