# Mysql

## MyISAM和InnoDB区别

1. Myisam是非聚簇索引，InnoDb是聚簇索引。
2. Myisam只有表级锁，innodb除了表级锁，还支持行级锁(默认)。
3. Myisam不提供事务支持，innodb可以支持事务
4. myisam不支持外键，innodb有外键

## 事务

2.1 事务的4大特性

原子性、一致性、隔离性、持久性

2.2 事务带来的问题

1. 脏读

当A事务正在访问数据并且对数据进行了修改，但是并没有提交到数据库。这时B事务也访问了这个数据。因为A事务还没有提交，这时B事务读取出来的数据还是上一次的数据，导致了脏读。

1. 修改丢失
2. B两个事务同时读取了一个数据，并对这个数据进行修改，这样，事务A修改的数据导致丢失。

事务A读去了一个数据A.age=20，事务B也读取了这个数据B.age=20，事务A修改了数据A.age=age-1，提交了事务，事务B修改了数据，B.age=age-1，也提交了数据，这个时间数据A提交的结果就被覆盖了。

1. 不可重复读

事务A对同一数据进行了多次读取，在事务执行期间，事务B对这一行数据进行了修改，这就导致事务A读取这一行数据前后不一致，导致了不可重复读。

1. 幻读

事务A对同一数据进行了多次读取，在事务执行期间，事务B插入了几行数据，这就导致了在随后的查询中，事务A发现同一个查询条件中多出了几行数据，导致了幻读。

2.3 事务隔离级别

1. READ-UNCOMMITTED(读取未提交)： 最低的隔离级别，允许读取尚未提交的数据变更，可能会导致脏读、幻读或不可重复读。

2. READ-COMMITTED(读取已提交)： 允许读取并发事务已经提交的数据，可以阻止脏读，但是幻读或不可重复读仍有可能发生。

3. REPEATABLE-READ(可重复读)：对同一字段的多次读取结果都是一致的，除非数据是被本身事务自己所修改，可以阻止脏读和不可重复读，但幻读仍有可能发生。默认的。

4. SERIALIZABLE(可串行化)：最高的隔离级别，完全服从ACID的隔离级别。所有的事务依次逐个执行，这样事务之间就完全不可能产生干扰，也就是说，该级别可以防止脏读、不可重复读以及幻读。分布式事务一般使用。

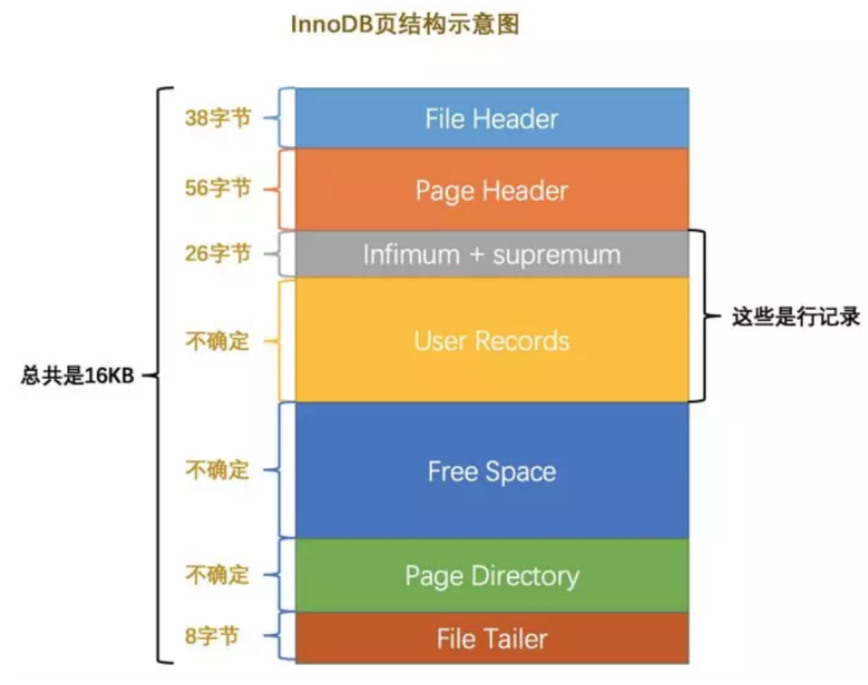


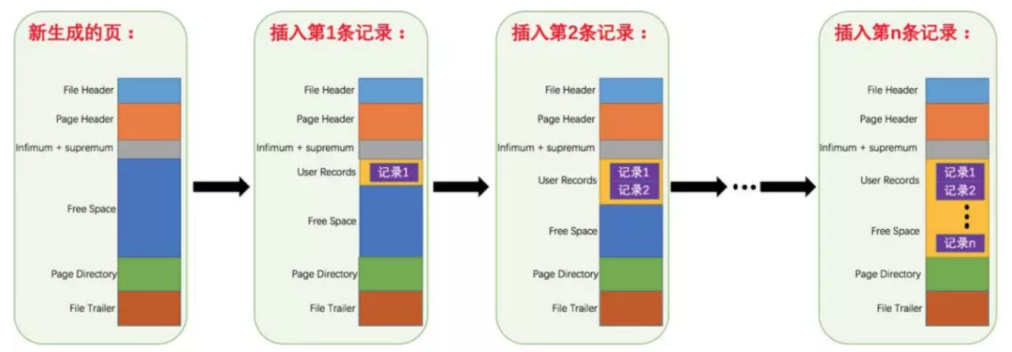
## 索引

### 为什么说索引会加快数据库检索速度

1.1 Mysql的基本结构

Mysql的基本存储单位是页(记录都存在页里面)。





1. 各个数据页可以组成一个双向链表
2. 而每个数据页中的记录又可以组成一个单向链表
   1. 每个数据页都会为存储在它里边儿的记录生成一个页目录，在通过主键查找某条记录的时候可以在页目录中使用二分法快速定位到对应的槽，然后再遍历该槽对应分组中的记录即可快速找到指定的记录
   2. 以其他列(非主键)作为搜索条件：只能从最小记录开始依次遍历单链表中的每条记录。
3. mysql每次从磁盘读取16KB(innodb引擎)，为的是减少磁盘IO，因为根据程序局部性，一个程序在访问了一条数据之后，在之后会有极大的可能再次访问这条数据和访问这条数据的相邻数据，所以索性直接加载4KB的数据到内存中，下次要访问这一页的数据时，直接从内存中找。
4. 为什么页里面的单链表是排好序的，因为单链表的特点也就是增删快，查询慢，所以优化查询的效率是必须的。另外一个作用，是为了页目录查询。

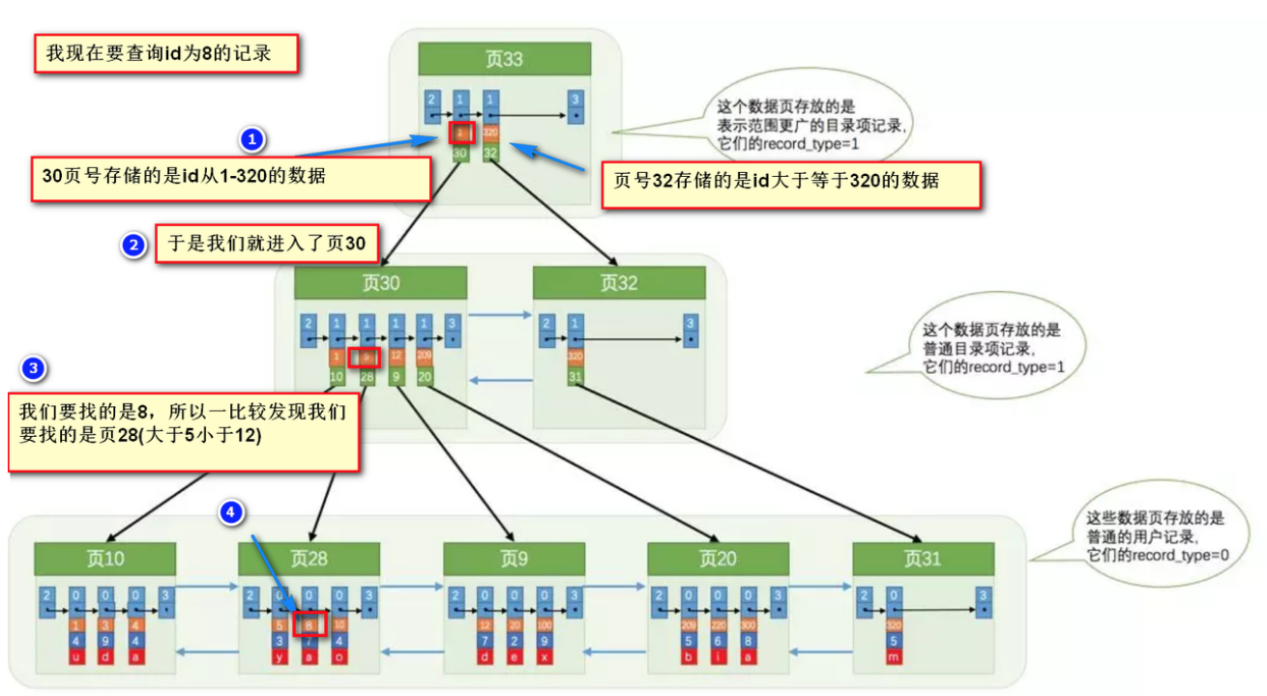
如果我们想要找一个id=5的数据，首先，我们得先从id=1的数据开始读起，然后判断是否是我们需要的数据，如果不是，就再取id=2的数据，再进行判断，循环往复。毋庸置疑，在MySQL帮我们排好序之后，我们需要经历五次磁盘IO，才能将5号数据找到并读出来。

引入页的概念后，MySQL会将多条数据存在一个叫“页”的数据结构中，当MySQL读取id=1的数据时，会将id=1数据所在的页整页读到内存中，然后在内存中进行遍历判断，由于内存的IO速度比磁盘高很多，所以相对于磁盘IO，几乎可以忽略不计，那么我们来看看这样读取数据我们需要经历几次磁盘IO（假设每一页可以存4条数据）。

那么我们第一次会读取id=1的数据，并且将id=1到id=4的数据全部读到内存中，这是第一次磁盘IO，第二次将读取id=5的数据到内存中，这是第二次磁盘IO。所以我们只需要经历2次磁盘IO就可以找到id=5的这条数据。

### 1.2 索引提高检索速度

采用B+树，将无序的数据变成有序。



很明显的是：没有用索引我们是需要遍历双向链表来定位对应的页，现在通过\*\*“目录”\*\*就可以很快地定位到对应的页上了！

## 聚餐索引和非聚餐索引

聚集索引就是以主键创建的索引，非聚集索引就是以非主键创建的索引。非聚集索引也叫做二级索引(因为他要回表查)

所谓聚簇索引，就是将索引和数据放到一起，找到索引也就找到了数据，我们刚才看到的B+树索引就是一种聚簇索引，而非聚簇索引就是将数据和索引分开，查找时需要先查找到索引，然后通过索引回表找到相应的数据。InnoDB有且只有一个聚簇索引，而MyISAM中都是非聚簇索引。

区别：

1. 聚集索引在叶子节点存储的是表中的数据
2. 非聚集索引在叶子节点存储的是主键和索引列
3. 使用非聚集索引查询出数据时，拿到叶子上的主键再去查到想要查找的数据。(拿到主键再查找这个过程叫做回表)。

在创建多列索引中也涉及到了一种特殊的索引-->覆盖索引

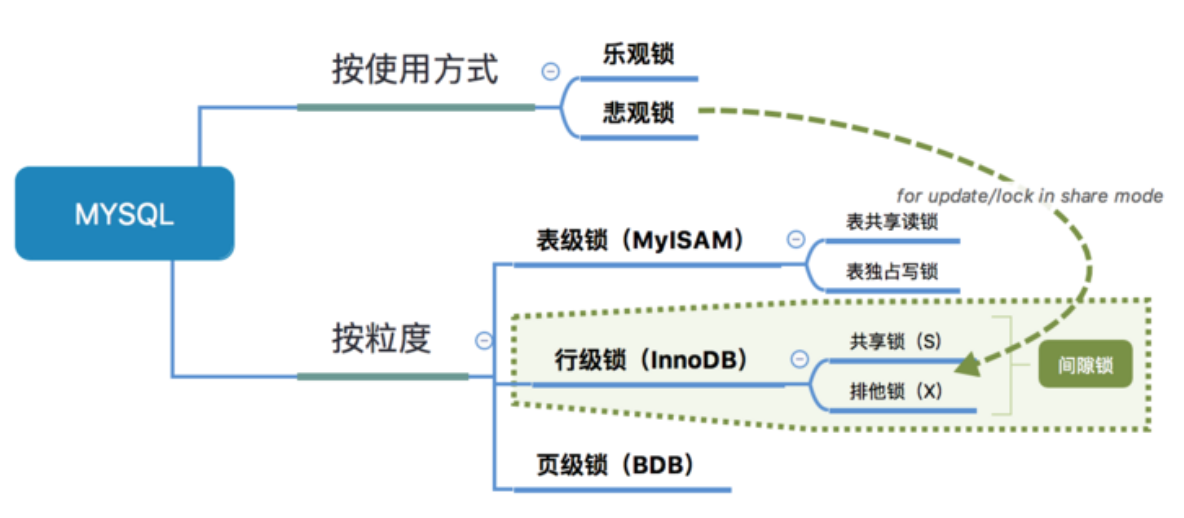
1. 如果不是聚集索引，叶子节点存储的是主键+列值
2. 最终还是要“回表”，也就是要通过主键再查找一次。这样就会比较慢
3. **覆盖索引就是把要查询出的列和索引是对应的，不做回表操作！覆盖索引查出的数据可以从索引中直接取出。**

## 最左匹配原则

(>、<、between、like左匹配)等就不能进一步匹配了，后续退化为线性查找。

# 锁

mysql中使用锁可以按照使用方式、粒度来区分。



使用方式上，可分为

1. 乐观锁

具体体现在在表的设计中填写version列，每次读取更新时按照version的数值来进行操作。

1. 悲观锁

具体体现在sql语法上，在select时，添加for update显示的表明使用悲观锁来锁住数据行。

按照粒度分

1. 表锁 开销小，加锁快；不会出现死锁；锁定力度大，发生锁冲突概率高，并发度最低。
2. 行锁 开销大，加锁慢；会出现死锁；锁定粒度小，发生锁冲突的概率低，并发度高。

### 表锁

不同的存储引擎支持的锁粒度是不一样的：

InnoDB行锁和表锁都支持！

MyISAM只支持表锁！

但是：InnoDB的行锁是基于索引的！也就是说，InnoDB只有通过索引条件检索数据才使用行级锁，否则，InnoDB将使用表锁。

表锁分为两种形式：

1. 表读锁
2. 表写锁

在表读锁和表写锁的环境下：读读不阻塞，读写阻塞，写写阻塞！

1. 读读不阻塞：当前用户在读数据，其他的用户也在读数据，不会加锁
2. 读写阻塞：当前用户在读数据，其他的用户不能修改当前用户读的数据，会加锁！
3. 写写阻塞：当前用户在修改数据，其他的用户不能修改当前用户正在修改的数据，会加锁！

### 行锁

innodb实现了两种类型的行锁：

1. 共享锁，允许一个事务去读一行，阻止其他事务获得相同数据集的排他锁。
   1. 也叫做读锁：读锁是共享的，多个客户可以同时读取同一个资源，但不允许其他客户修改。
2. 排他锁（X锁)：允许获得排他锁的事务更新数据，阻止其他事务取得相同数据集的共享读锁和排他写锁。
   1. 也叫做写锁：写锁是排他的，写锁会阻塞其他的写锁和读锁。

**innodb基于行锁还实现了MVCC多版本并发控制，MVCC在隔离级别下的Read committed和Repeatable read下工作。MVCC能够实现读写不阻塞！**

### MVCC

MVCC指的是，基于innodb引擎，在已提交读(read committed)和可重复读(repeatable read)两种隔离级别下的事务，通过对版本链中的记录进行判断来操作select。

这就使得别的事务可以修改这条记录，在每次修改都会在版本链中记录。SELECT可以去版本链中拿记录，这就实现了读-写，写-读的并发执行，提升了系统的性能。

### 具体实现

其具体的实现依赖两个数据结构，一个是版本链、一个是read view

### 版本链：

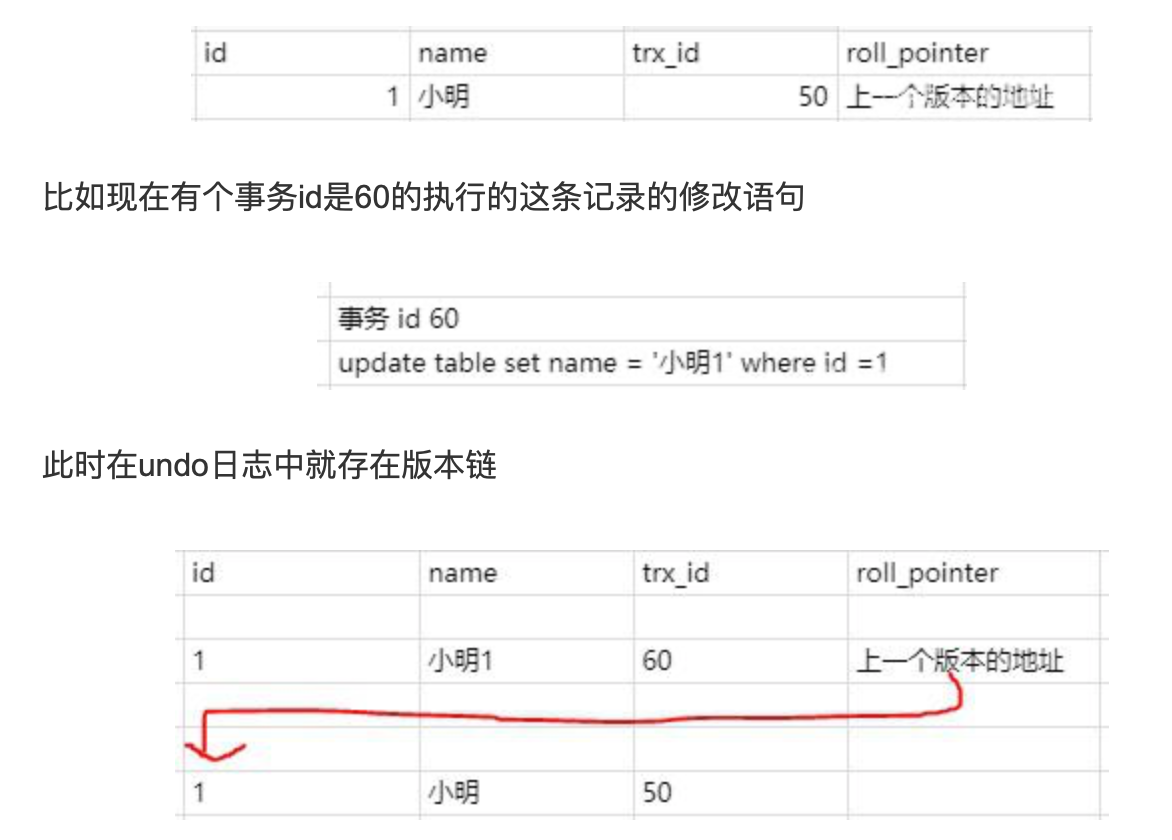
在innodb中，聚簇索引的记录中有两个隐藏的字段：

1. trx\_id

这个id用来存储的每次对某条聚簇索引记录进行修改的时候的事务id。

1. roll\_pointer

每次对数据进行修改时，都会将老数据放到undo\_log中，在这个roll\_pointer存放的就是undo\_log中老数据的指针。通过它可以获得上一个版本的数据。



### ReadView

ReadView中主要就是有个列表来存储我们系统中当前活跃着的读写事务，也就是begin了还未提交的事务。通过这个列表来判断记录的某个版本是否对当前事务可见。

假设当前列表里的事务id为[80,100]。

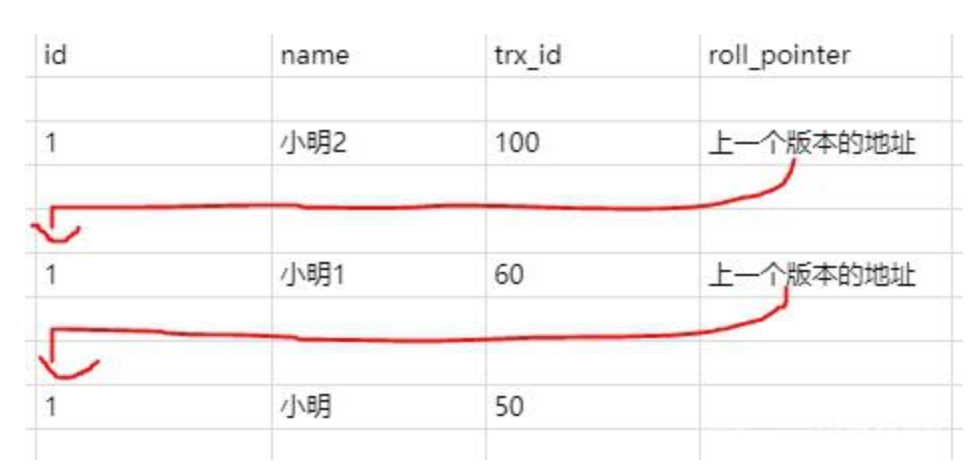
1. 如果你要访问的记录版本的事务id为50，比当前列表最小的id80小，那说明这个事务在之前就提交了，所以对当前活动的事务来说是可访问的。
2. 如果你要访问的记录版本的事务id为70,发现此事务在列表id最大值和最小值之间，那就再判断一下是否在列表内，如果在那就说明此事务还未提交，所以版本不能被访问。如果不在那说明事务已经提交，所以版本可以被访问。
3. 如果你要访问的记录版本的事务id为110，那比事务列表最大id100都大，那说明这个版本是在ReadView生成之后才发生的，所以不能被访问。

这些记录都是去版本链里面找的，先找最近记录，如果最近这一条记录事务id不符合条件，不可见的话，再去找上一个版本再比较当前事务的id和这个版本事务id看能不能访问，以此类推直到返回可见的版本或者结束。

**已提交读和可重复读的区别就在于它们生成ReadView的策略不同。**

### 已提交读readview策略

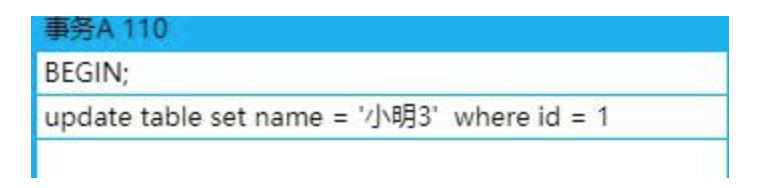
比如此时有一个事务id为100的事务，修改了name,使得的name等于小明2，但是事务还没提交。则此时的版本链是



那此时另一个事务发起了select 语句要查询id为1的记录，那此时生成的ReadView 列表只有[100]。那就去版本链去找了，首先肯定找最近的一条，发现trx\_id是100,也就是name为小明2的那条记录，发现在列表内，所以不能访问。

这时候就通过指针继续找下一条，name为小明1的记录，发现trx\_id是60，小于列表中的最小id,所以可以访问，直接访问结果为小明1。

那这时候我们把事务id为100的事务提交了，并且新建了一个事务id为110也修改id为1的记录，并且不提交事务。



这个时间版本链变为了：



这时候之前那个select事务又执行了一次查询,要查询id为1的记录。

如果你是已提交读隔离级别，这时候你会重新一个ReadView，那你的活动事务列表中的值就变了，变成了[110]。

按照上的说法，你去版本链通过trx\_id对比查找到合适的结果就是小明2。

### 可重复读隔离级别readview策略

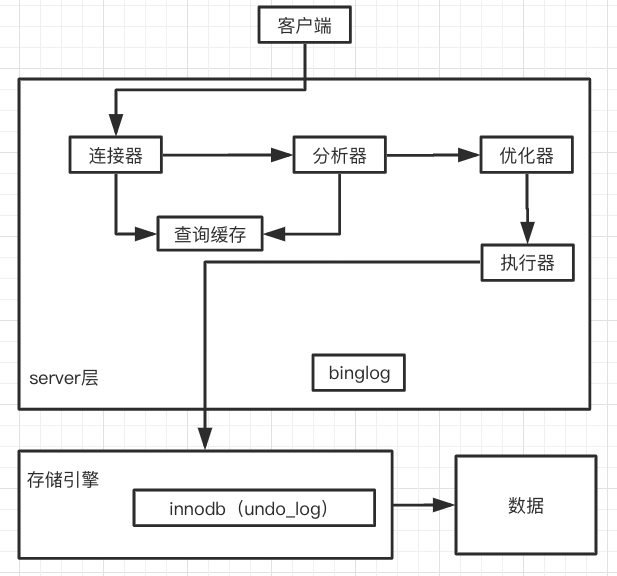
这时候你的ReadView还是第一次select时候生成的ReadView,也就是列表的值还是[100]。所以select的结果是小明1。所以第二次select结果和第一次一样，所以叫可重复读！

也就是说已提交读隔离级别下的事务在每次查询的开始都会生成一个独立的ReadView,而可重复读隔离级别则在第一次读的时候生成一个ReadView，之后的读都复用之前的ReadView。

这就是Mysql的MVCC,通过版本链，实现多版本，可并发读-写，写-读。通过ReadView生成策略的不同实现不同的隔离级别。

# sql在mysql中是如何执行的

mysql简要架构：



Server层：主要包括连接器、查询缓存、分析器、优化器、执行器等，所有跨存储引擎的功能都在这一层实现，比如存储过程、触发器、视图，函数等，还有一个通用的日志模块 binglog日志模块。

存储引擎： 主要负责数据的存储和读取，采用可以替换的插件式架构，支持InnoDB、MyISAM、Memory等多个存储引擎，其中InnoDB引擎有自有的日志模块redolog 模块。

### 连接器

主要负责用户登录，进行用户身份认证，包括校验账户密码、权限等操作。

### 查询缓存

此部分musql8.0之后就去掉了，实际应用场景比较少。

连接建立后，执行查询语句的时候，会先查询缓存，Mysql会先校验这个sql是否执行过，以Key-Value的形式缓存在内存中，Key是查询预计，Value是结果集。如果缓存key被命中，就会直接返回给客户端，如果没有命中，就会执行后续的操作，完成后也会把结果缓存起来，方便下一次调用。

Mysql 查询不建议使用缓存，因为对于经常更新的数据来说，缓存的有效时间太短了，往往带来的效果并不好，对于不经常更新的数据来说，使用缓存还是可以的。

### 分析器

1. sql语句提取

一条SQL语句有多个字符串组成，首先要提取关键字，比如select，提出查询的表，提出字段名，提出查询条件等等。

2. 语法分析

主要就是判断你输入的sql是否正确，是否符合mysql的语法。

### 优化器

优化器的作用就是它认为的最优的执行方案去执行（虽然有时候也不是最优），比如多个索引的时候该如何选择索引，多表查询的时候如何选择关联顺序等。

### 执行器

当选择了执行方案后，就会去调用搜索引擎接口，返回执行接口。

## 具体执行步骤

### 查询过程

Select \* from db\_table A where A.age = 19 and A.name = “张三”;

1. 校验该语句是否有前线，如果没有直接返回。
2. 通过分析器进行语句拆分，提取sql语句的关键元素，
   1. 比如：查询select，
   2. 提取需要查询的表为db\_table，
   3. 需要查询的列是什么。
   4. 查询条件是什么

判断有无语法错误。

1. 优化器需要进行确认执行方案，上面这个sql有两种执行方案：
   1. 先查询A.name=”张三”的学生，然后在判断A.age = 19
   2. 先找出A.age=19的学生，然后在过滤出A.name=“张三”的学生

那么优化器根据自己的优化算法进行选择执行效率最好的一个方案。

1. 进行权限校验，如果没有权限就会返回错误信息，如果有权限就会调用数据库引擎接口，返回引擎的执行结果。

### 更新语句

update tb\_student A set A.age='19' where A.name='张三';

前面的3步和查询一致，只是在更新的时候要记录日志。具体流程：

1. 先查询到张三这条记录
2. 然后拿到查询的数据，把age改成19，然后调用搜索引擎的API，写入这一行数据，innodb会将此数据记录到undo\_log中，此时undo\_log进入prepare状态，然后告诉执行器，执行完成了，随时可以提交。
3. 执行器收到通知后，记录binlog，然后调用搜索引擎API，提交undo\_log为提交状态。
4. 更新完成。

这里的undo\_log主要是为了解决事务的问题。

为什么引入prepare状态，如果不用prepare状态的话，有两种提交方式

1. 先写undo\_log直接提交，然后写binlog

假设写完redo log 后，机器挂了，binlog日志没有被写入，那么机器重启后，这台机器会通过redo log恢复数据，但是这个时候bingog并没有记录该数据，后续进行机器备份的时候，就会丢失这一条数据，同时主从同步也会丢失这一条数据。

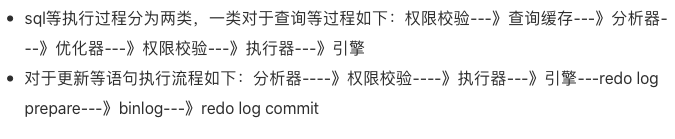
1. 先写binlog，然后写undo\_log

假设写完了binlog，机器异常重启了，由于没有redo log，本机是无法恢复这一条记录的，但是binlog又有记录，那么和上面同样的道理，就会产生数据不一致的情况。

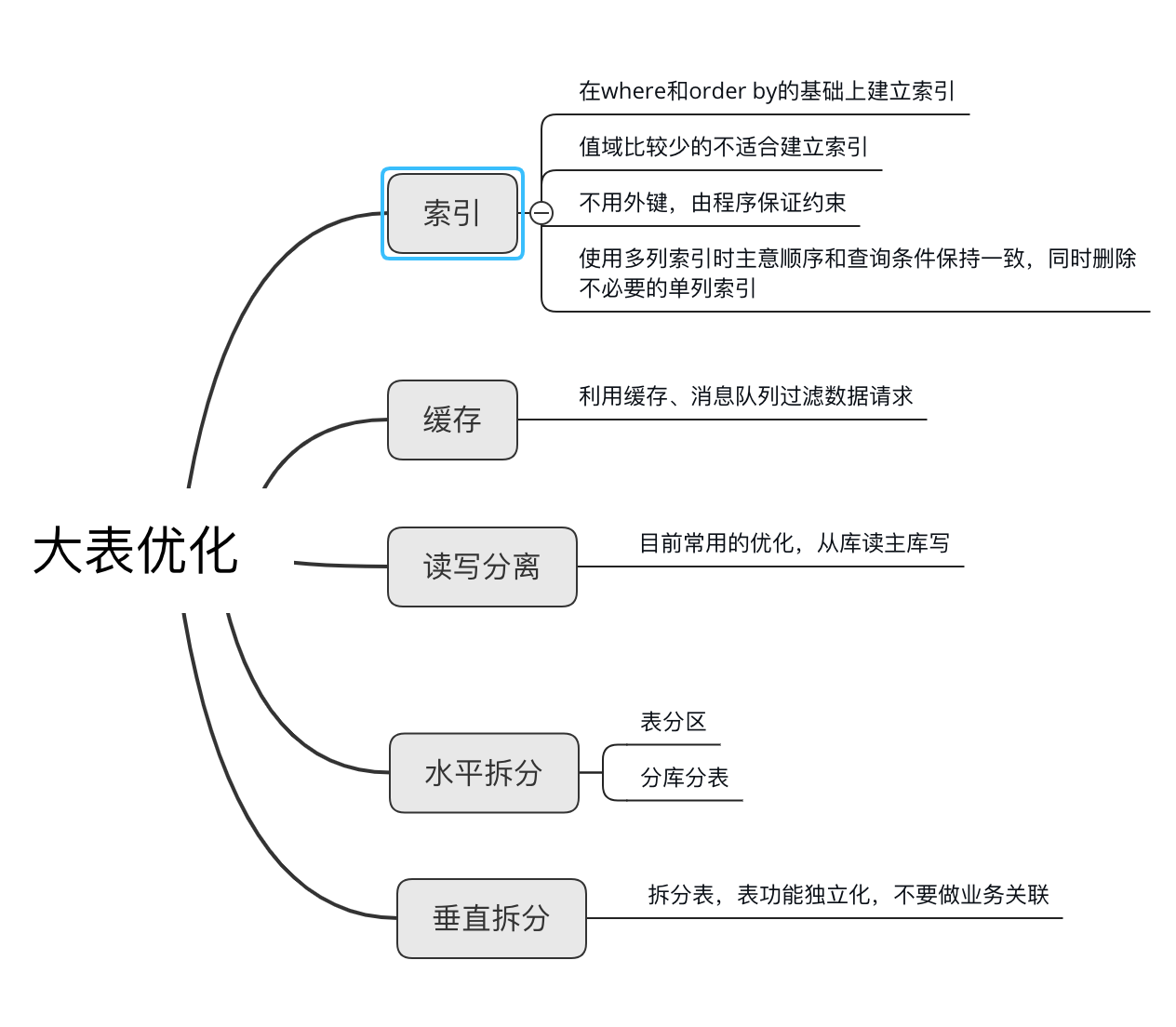
因此主要解决的是数据一致性问题。

假设redo log 处于预提交状态，binglog也已经写完了，这个时候发生了异常重启会怎么样呢？ 这个就要依赖于mysql的处理机制了，mysql的处理过程如下：

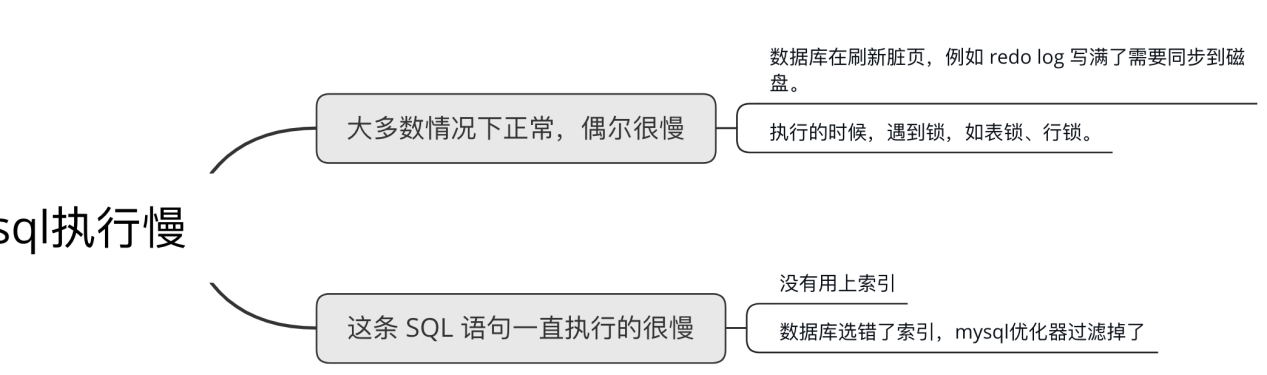
1. 判断redo log 是否完整，如果判断是完整的，就立即提交。
2. 如果redo log 只是预提交但不是commit状态，这个时候就会去判断binlog是否完整，如果完整就提交 redo log, 不完整就回滚事务。



## Mysql大表优化方案



## sql执行慢的原因



### 大多数情况下正常，偶尔很慢

### 数据库在刷新脏页（flush）

向数据库中插入或者更新数据的时候，更新的数据不会立马持久化到磁盘，而是被更新到redo log中，等到空闲的时候，在通过 redo log 里的日记把最新的数据同步到磁盘中去。

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”。

**脏页刷新有4中场景：**

1. redo log写满

redo log里的容量是有限的，如果数据库一直很忙，更新又很频繁，redolog很快就会被写满，此时mysql会把数据同步到磁盘里。这个时候就会导致正常的sql语句会突然执行的很慢。

1. 内存不够用

如果一次查询较多的数据，恰好碰到所查的数据页不在内存中，需要申请内存，而此时内存不足，需要淘汰一部分数据页，如果是干净页，就直接释放，如果是脏页，就需要刷脏页。

1. Mysql正常关闭

在关闭时，mysql会把脏页全部刷新到磁盘上。

### 拿不到锁

在执行sql时，正好碰到这个表被别人锁了，拿不到锁，此时sql执行会变慢，可以用**show processlist来查看。**

## sql一直很慢

## 没有用到索引，查询条件没有设置索引

1. 字段根本就没有索引
2. 字段有索引，但是由于查询条件中用了运算公式或者函数，导致索引没用。

## 查询字段涉及到了索引，但是mysql优化掉了，导致全表扫描

mysql在执行sql时，在最后调用搜索引擎接口前，需要经过优化器处理，目的时为了预测是走索引扫描的行数少，还是全表扫描的行数少。扫描的行数越少，I/O操作就越少。

优化器通过索引区分度来进行判断，如果一个索引上字段越多，说明索引区分度越高，区分度越高，说明走索引就越有优势，扫描的行数就越少。

这个索引区分度如何而来，mysql不可能全表先遍历一遍吧，只能通过采样来预测索引的区分度。

只要涉及到采样，数据越大的表，采样的结果就越难预测。如果采样的时候，采样的区间没有一条数据匹配上了索引，mysql就基本上认为这条sql的索引区分度基本上为0了，就会直接走全表扫描。

最终的结论就是**由于统计的失误，导致系统没有走索引，而是走了全表扫描。**

（系统判断是否走索引，扫描行数的预测其实只是原因之一，这条查询语句是否需要使用使用临时表、是否需要排序等也是会影响系统的选择的）

如何避免：

1. 可以通过强制使用索引的方式来查询，force index(name)

select \* from t **force index**(a) where c < 100 and c < 100000;

可以通过show index 来查看索引的基数(cardinality)，看一下是否和实际相符，如果不相符，可以重新统计索引数，analyze table t;