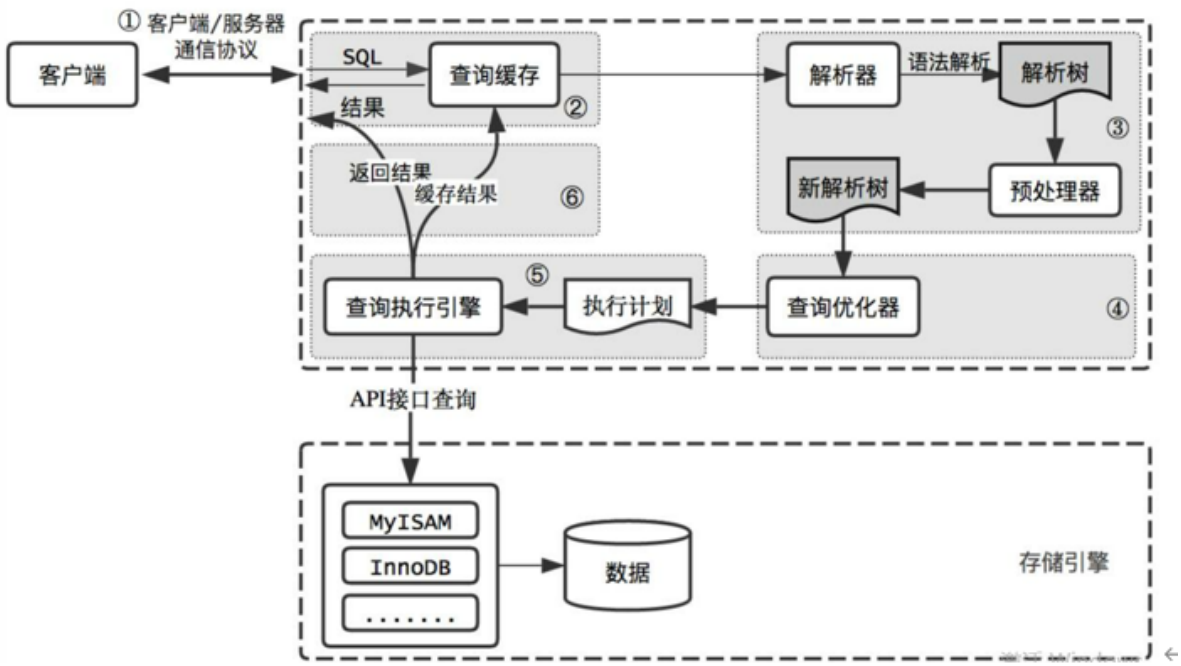
MySql

一、执行流程



（1）首先客户端向MySQL服务器发送一条查询请求

（2）服务器首先检查查询缓存，如果命中缓存，则返回存储在缓存中的结果。否则进入下一阶段。缓存的是语句和对应的查询结构。

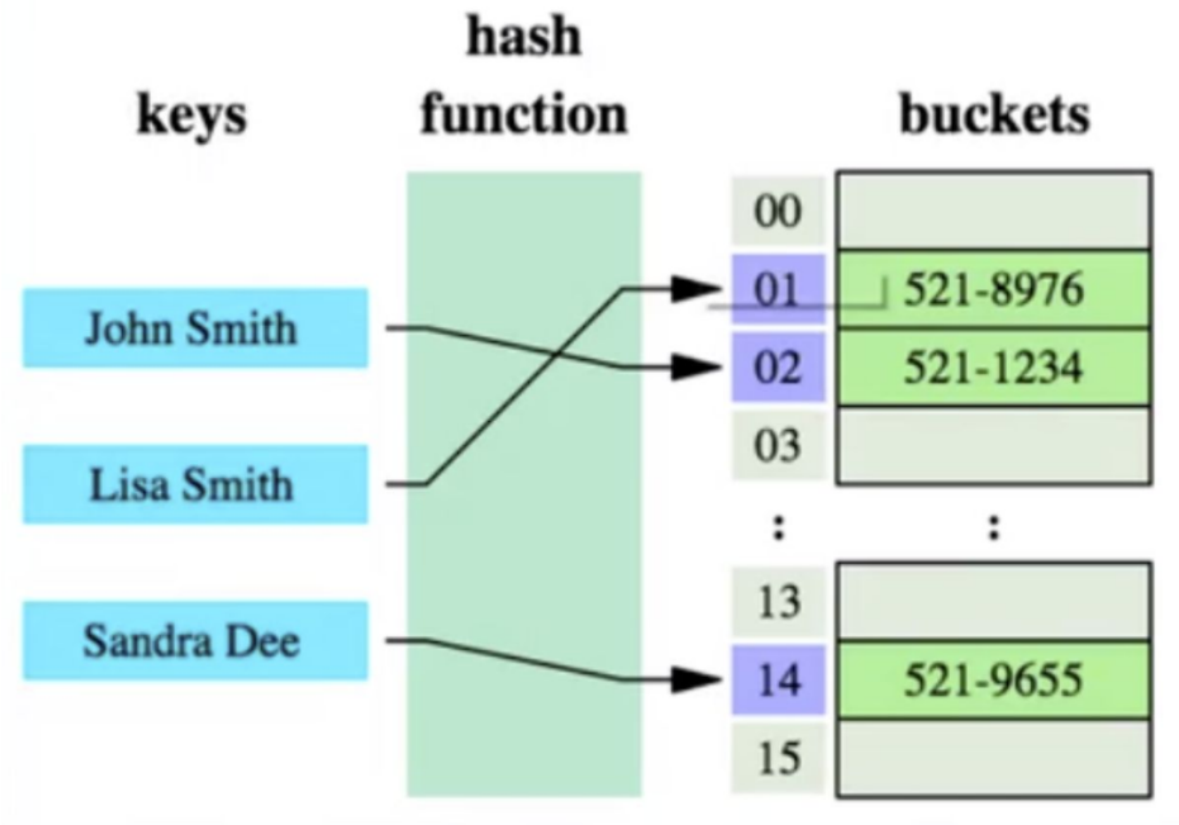
（3）然后服务器进行SQL解析、预处理、再由优化器生成对应的执行计划。解析是指按照SQL自定义的解析规则转换成解析树，预处理则会根据MySQL规则进一步检查解析树是否合法，一条查询可以有多种执行方式，优化器的作用就是找到这其中最好的执行计划。

（4）MySQL根据执行计划，调用存储引擎的API来执行查询

（5）将结果返回给客户端，同时缓存查询结果

二、Mysql 索引底层数据结构选型

1、哈希表



哈希表就是通过哈希函数将一个关键字映射成一个固定的值。

当我们进行查询时，就会将 SQL 语句中待查询字段的值，通过哈希函数映射成一个物理数据的地址，然后根据这个地址去数据文件中拿出数据。

```
select * from user where id=7;
```

哈希算法首先计算存储 id=7 的数据的物理地址 addr=hash(7)=4231，而 4231 映射的物理地址是 0x77，0x77 就是 id=7 存储的是数据的物理地址，通过该独立地址可以找到对应 user_name='g'这个数据。这就是哈希算法快速检索数据的计算过程。

【哈希碰撞】

但这种方式存在数据碰撞的问题，也就是哈希函数可能对不同的 key 会计算出同一个结果，比如 hash(7)可能跟 hash(199)计算出来的结果一样，也就是不同的 key 映射到同一个结果了，这就是碰撞问题。

【链地址法】

解决碰撞问题的一个常见处理方式就是链地址法，即用链表把碰撞的数据接连起来。计算哈希值之后，还需要检查该哈希值是否存在碰撞数据链表，有则一直遍历到链表尾，直达找到真正的 key 对应的数据为止。

【等值查询效率】

哈希算法时间复杂度为 O（1），检索速度非常快。比如进行 id=7 的等值查询时，哈希索引只需要计算一次就可以获取到对应的数据，检索速度非常快。但是 Mysql 并没有采取哈希作为其底层算法，这是为什么呢?

【范围查询】

因为考虑到数据检索有一个常用手段就是范围查找，比如以下这个 SQL 语句

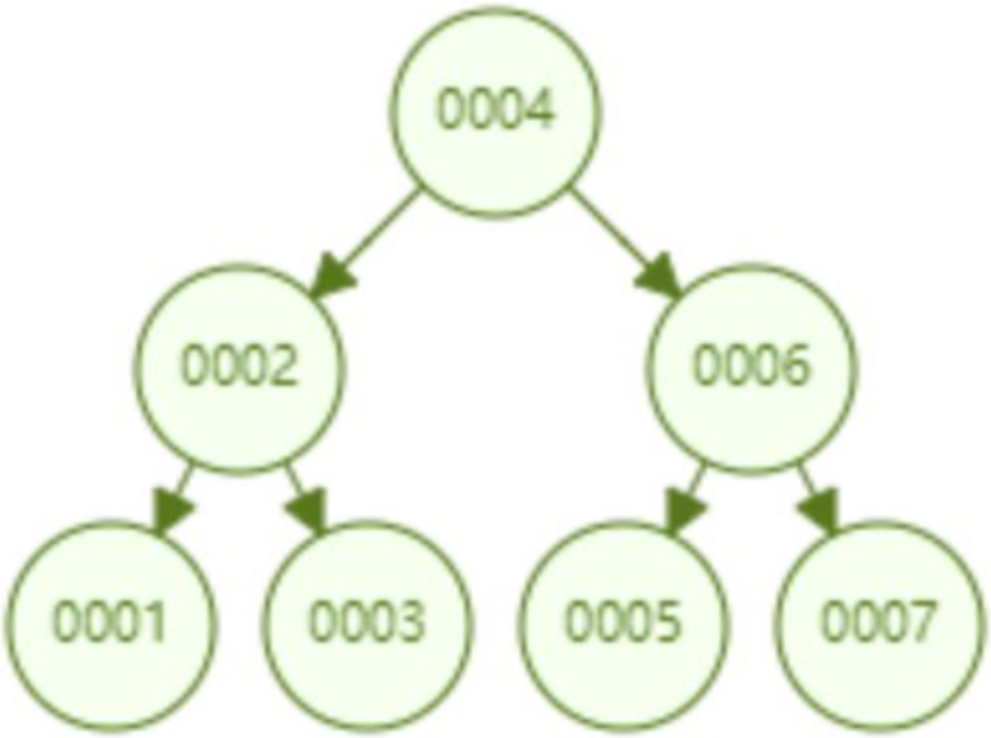
```
select * from user where id>3;
```

因为 hash 结构由于其 hash 函数的不确定性，映射出来的地址分布散列，并不能形成物理与逻辑对应的顺序关系。不像 b 树有严格的定义，就是左子节点一定小于父节点，而右子节点一定大于父节点，这样才是形成了顺序关系，便于范围查询。数据结构的散列性决定的。

而 hash 结构不方便进行范围查询，会一次把所有数据找出来加载到内存，然后再在内存里筛选筛选目标范围内的数据。

> **关键字映射，拿地址取数据**
>
> **哈希碰撞，链地址法**
>
> **等值查询O1**
>
> **范围查询加载所有数据到内存进行筛选**

2、二叉查找树



二叉查找树的时间复杂度是 O(lgn)，检索很高效。但在极端情况下， 二叉查找树会变得很不平衡，从而退化成线性链表的查找效率，时间复杂度为O（n）。比如在自增主键的情况下，会将数据一直添加到右节点，导致出现极端不平衡。

> **左<中<右，方便范围查询，O(lgn)，但会极端不平衡，退化成O(n)**

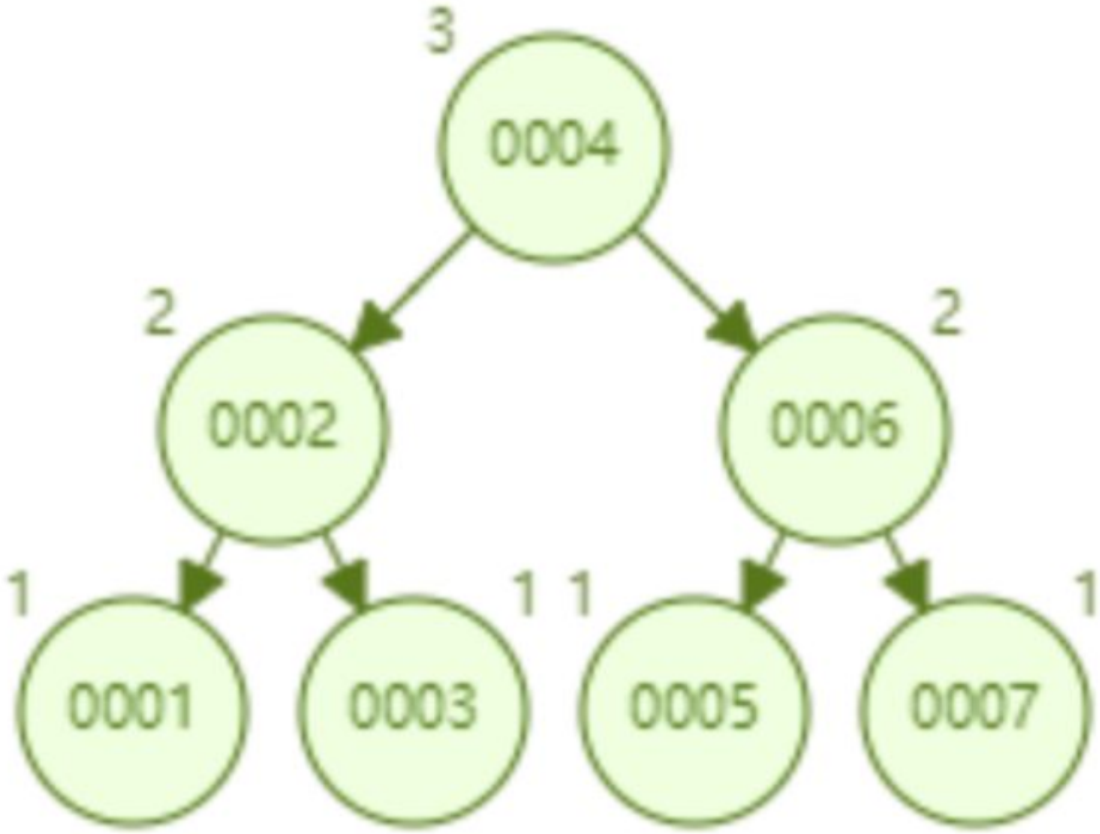
3、红黑树

所以就出现了红黑树，是一颗可以通过节点的左右旋转以及变色来自动调整树从不平衡恢复到基本平衡，恢复时间复杂度到 O（logn），但当数据依然是类似自增主键这样的场景时，树依然还是会呈现出右倾的趋势，只不过其趋势没有二叉平衡树那么明显而已，但依旧会出现不平衡。

> **左右旋转与变色、O(lgn)，也会不平衡，出现右倾**

4、 自平衡二叉树 AVL 树

所以又出现了 AVL 自平衡二叉树，它通过更多的调整来使得树变得更加平衡，这种平衡的标准应该是左右树的高度相差最多为1的意思，同时这种调整是伴随性能的损耗的，但可以解决红黑树出现的调整不够导致的右倾的问题。



【优点】

1. 不错的查找性能（O（logn）），不存在极端的低效查找的情况。
2. 可以实现范围查找、数据排序。

【缺点】

但 AVL 树存在节点承载的数据量少的问题。我们知道，调优数据库一个重要的方式就是减少 IO 操作，而 AVL 树一个节点一次只能存储一个数据， 一次磁盘 IO 只能取出来一个节点上的数据加载到内存里，那比如查询 id=7 这个数据我们就要进行磁盘 IO 三次，这是多么消耗时间的。

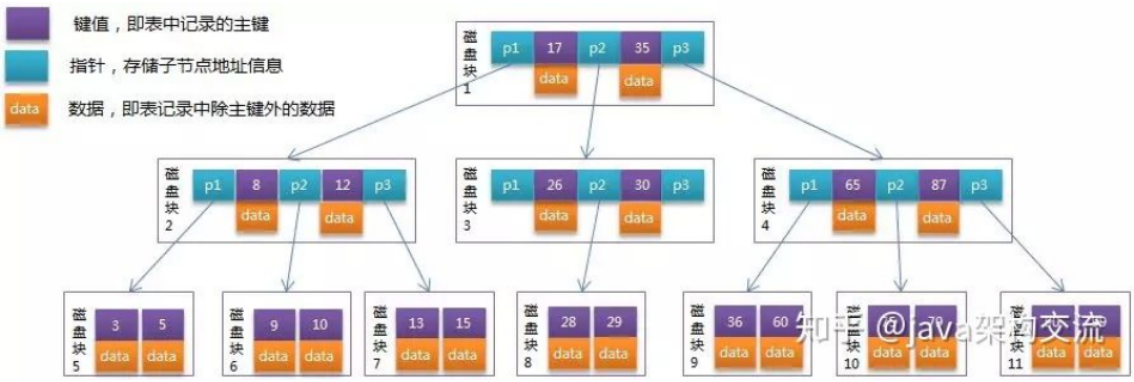
【解决】

而从磁盘读取效率来看，读取 1B 和 读取 1KB 所需要的时间是一样的，根据这个特点，我们就可以让一个节点尽量存储更多的数据，这样一次 IO 就可以读取更多的数据，根据局部性原理和预读机制，相当于缓存了之后经常访问到的数据到节点中，检索的时候就可以提高缓存的命中率，也间接地减少了磁盘的 IO 次数。 而这个就是 B 树和 B+ 树的设计原理。

> **O(lgn)，可维持平衡，但节点只存一个数据**

5、 B 树

所以B 树就是尽可能在一次磁盘 IO 中多读一点数据到内存，让每个节点可以存储更多的索引的 Key。

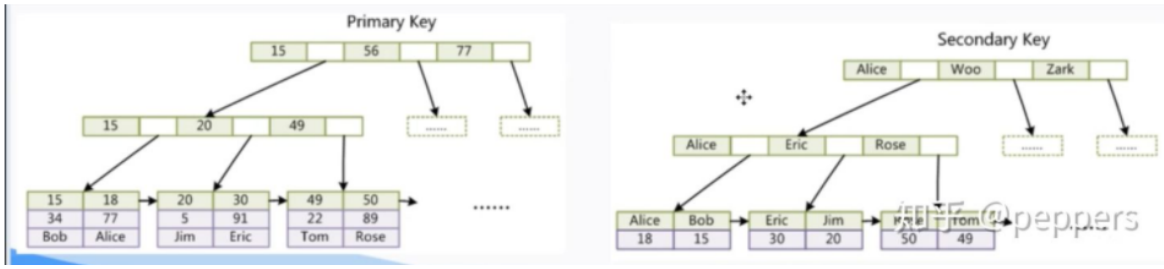


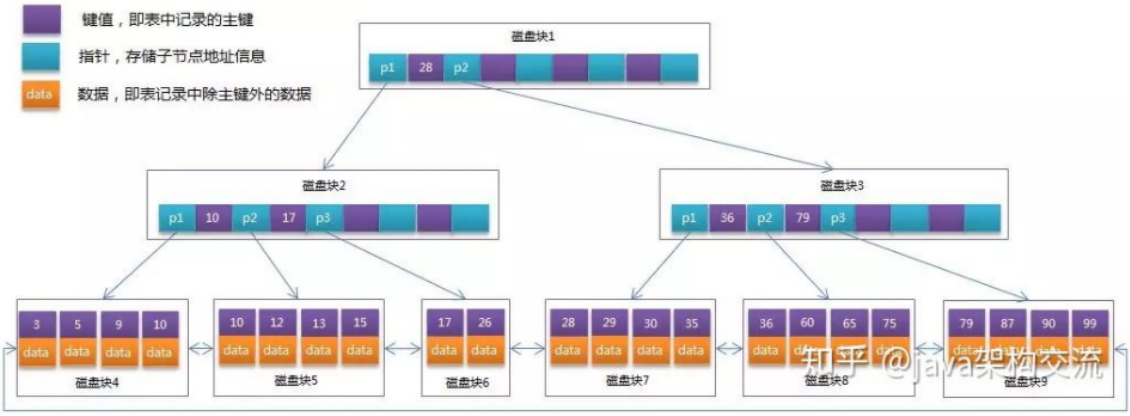
【优点】

1. 优秀检索速度，时间复杂度：B 树的查找性能等于 O（h*logn），其中 h 为树高，n 为每个节点关键词的个数；
2. 尽可能少的磁盘 IO，加快了检索速度；
3. 可以支持范围查找。

> **O（h*logn）、局部性原理预读、节点更多数据、增加缓存命中、降低IO**

6、B+ 树





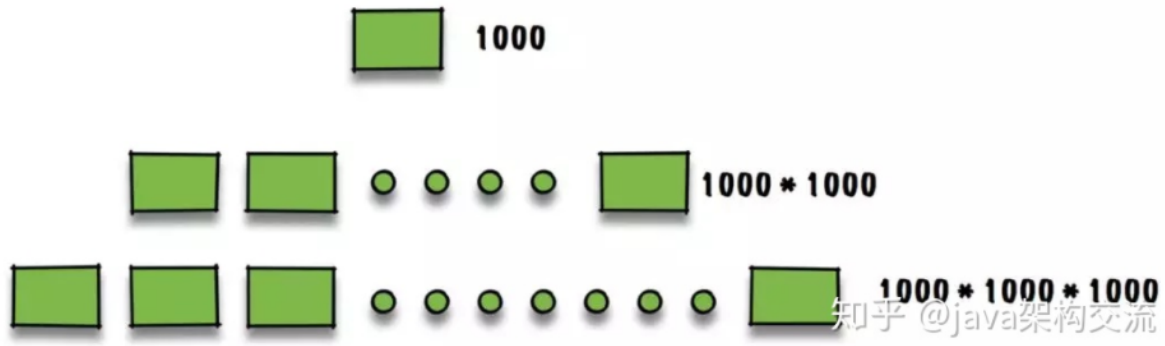
【B+ 树与 B 树的区别】

1）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+树一个节点能存很多索引，使得整个 B+树高度降低，减少了磁盘 IO 。

2）B+树叶子节点存储索引所对应的一行数据或数据的地址，即存储了表的一行完整数据。

3）B+树的相邻的叶子节点使用链表连接起来，提高了相邻区间的访问效率，不像B树还需要重新返回到上一层的父节点再进行遍历。

【一颗 3 层的 B+树能够存储多少索引呢？】



大约能够存储 10 亿个索引。通常 B+ 树的高度在 2-4 层，由于 MySql 在运行时，根节点是常驻内存的，因此每次查找只需要大约 2 -3 次 IO。可以说，B+ 树的设计，就是根据机械磁盘的特性来进行设计的。

【预读】

Mysql 每页为16kb，每个节点存储的正好是一个节点的数据16kb。而操作系统每页4kb，这样设计的好处是，操作系统根据局部性原理，有预读的机制，所以 mysql 每次也会读取多个页，比如读取4页，这样正好就是16kb。

而预读使得b+树的每个节点能够存储更多的数据，树的高度更低，索引能够覆盖的数据更多，缓存的命中率页更好，最终降低了 IO 次数。

> **O（h*logn）、节点存更多索引、可以覆盖更多数据，树更低、IO更少，**
>
> **叶子节点存数据、区间链表**

三、Innodb 引擎和 Myisam 引擎的实现

Innodb 创建表后生成的文件有：

- frm:创建表的语句
- idb:表里面的数据+索引文件

Myisam 创建表后生成的文件有

- frm:创建表的语句
- MYD:表里面的数据文件（myisam data）
- MYI:表里面的索引文件（myisam index）

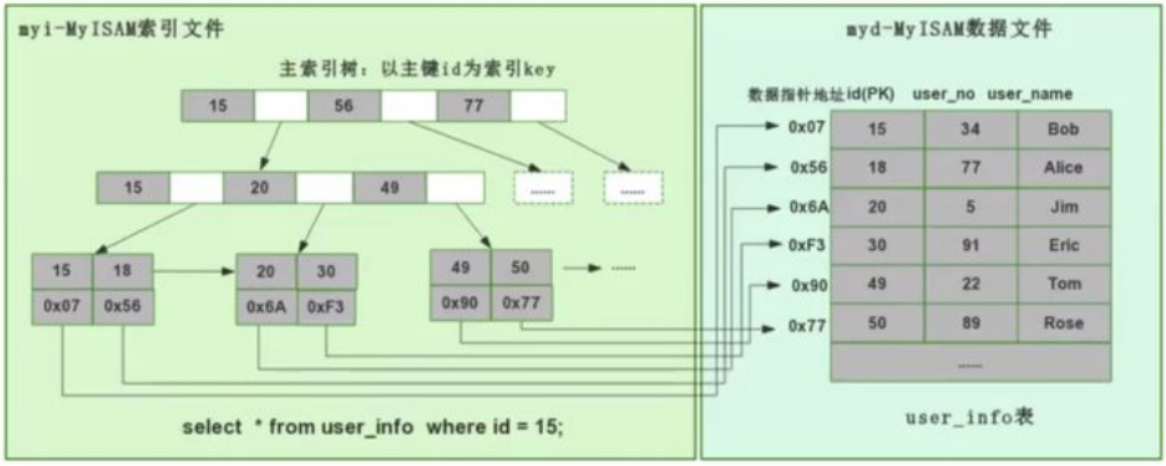
MyISAM 引擎把数据和索引分开了，这叫做非聚集索引方式；Innodb 引擎把数据和索引放在同一个文件里了，这叫做聚集索引方式

> **建表语句文件、数据文件、索引文件。**

> **注意点：小表使用全表扫描更快，中大表才使用索引。超级大表索引基本无效。**

1、MyISAM 引擎的底层实现（非聚集索引方式）

MyISAM 用的是非聚集索引方式，即数据和索引落在不同的两个文件上。MyISAM 在建表时以主键作为 KEY 来建立主索引 B+树，树的叶子节点存的是对应数据的物理地址。我们拿到这个物理地址后，就可以到 MyISAM 数据文件中直接定位到具体的数据记录了。



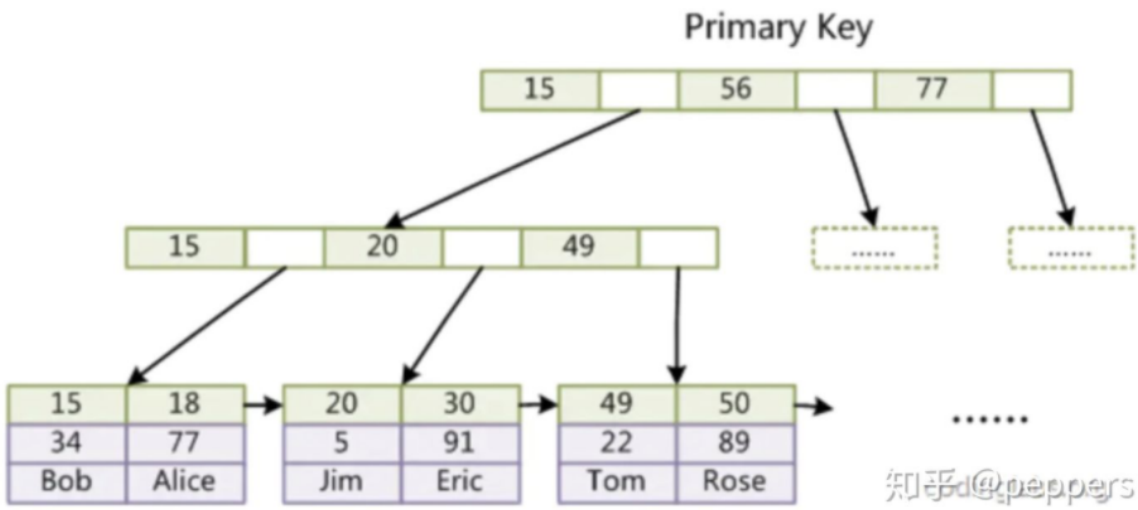
当我们为某个字段添加索引时，我们同样会生成对应字段的索引树，该字段的索引树的叶子节点同样是记录了对应数据的物理地址，然后也是拿着这个物理地址去数据文件里定位到具体的数据记录。

2、Innodb 引擎的底层实现（聚集索引方式）

InnoDB 是聚集索引方式，因此数据和索引都存储在同一个文件里。首先 InnoDB 会根据主键 ID 作为 KEY 建立索引 B+树， 然后检索到叶子节点，叶子节点存储了主键 key 以及 key 所对应的一行表数据。

而当我们为其他字段建立索引时，也会建立一颗索引 B+ 树，这不过这棵树的叶子节点存储的是主键的 Key，这叫辅助索引，然后再通过这个主键 key 回到主键索引的 B+ 树中检索到最终的数据，这个过程，就叫回表。

所以这也是为什么 Innodb 需要有主键的原因。



> 为什么 InnoDB 只在主键索引树的叶子节点存储了具体数据，但是其他索引树却不存具体数据呢，而要多此一举先找到主键，再在主键索引树找到对应的数据呢?

为了节约磁盘空间。因为一个表可能有多个索引，如果每颗索引树都存储了数据的话，就会导致一张表的许多索引树上挂了相同的数据，会占据很大的磁盘空间。

> InnoDB 和 MyISAM 性能对比

MyISAM 查询性能更好 ，MyISAM 直接找到物理地址后就可以直接定位到数据记录，但是 InnoDB 查询到叶子节点后，还需要再查询一次主键索引树，才可以定位到具体数据。等于 MyISAM 一步就查到了数据，但是 InnoDB 要两步，那当然 MyISAM 查询性能更高。

> **MyISAM 非聚集，叶子存地址**
>
> **InnoDB 聚集，叶子存数据，辅助索引叶子存主键，需要回表（主键索引树）**

四、事务

1、概念

数据库事务就是指一组连续的 SQL 组成的执行单元，这个执行单元中的这组 SQL，要么全部执行成功，要么全部执行失败。

举个简单又经典的例子就是转账了，事务A中要进行转账，那么转出的账号要扣钱，转入的账号要加钱，这两个操作都必须同时执行成功，为了确保数据的一致性。

2、ACID

在Mysql中事务的四大特性主要包含：**原子性（Atomicity）**、**一致性（Consistent）**、**隔离性（Isalotion）**、**持久性(Durable)**，

原子性是指事务中对数据的操作，要么全部执行成功，要么全部执行失败，实现事务的原子性，是基于日志的`Redo/Undo`机制。

一致性是指事务执行前后状态的一致性，状态主要指数据的状态，要么都更新成功，要么都更新失败。隔离性侧重指事务之间相互隔离，不受影响，这个与事务设置的隔离级别有密切的关系。

持久性则是指在一个事务提交后，这个事务的状态会被持久化到数据库中，也就是事务提交，对数据的新增、更新将会持久化到数据库中。

原子性、隔离性、持久性都是为了保障一致性而存在的，一致性也是最终的目的。

3、 Redo/Undo机制

Redo/Undo机制比较简单，它们将所有对数据的更新操作都写到日志中。

Redo log用来记录某数据块被修改后的值，可以用来恢复已经提交但未持久化到数据文件的数据；Undo log是用来记录数据更新前的值，保证数据更新失败能够回滚。

假如数据库在执行的过程中，不小心崩了，可以通过该日志的方式，回滚之前已经执行成功的操作，实现事务的一致性。

> 假如某个时刻数据库崩溃，在崩溃之前有事务A和事务B在执行，事务A已经提交，而事务B还未提交。当数据库重启后将进行数据恢复，通过Redo log将已经提交的事务A的更改写到数据文件，而还没有提交的事务B就通过Undo log进行roll back。

> **ACID**
>
> **redo 记录更新后的数据，用于恢复已提交未持久化的数据，undo 记录更新前的数据，用户更新失败时（出异常）的回滚**
>
> **set session transaction isolation level　设置会话交流的隔离级别**

4、事务隔离级别

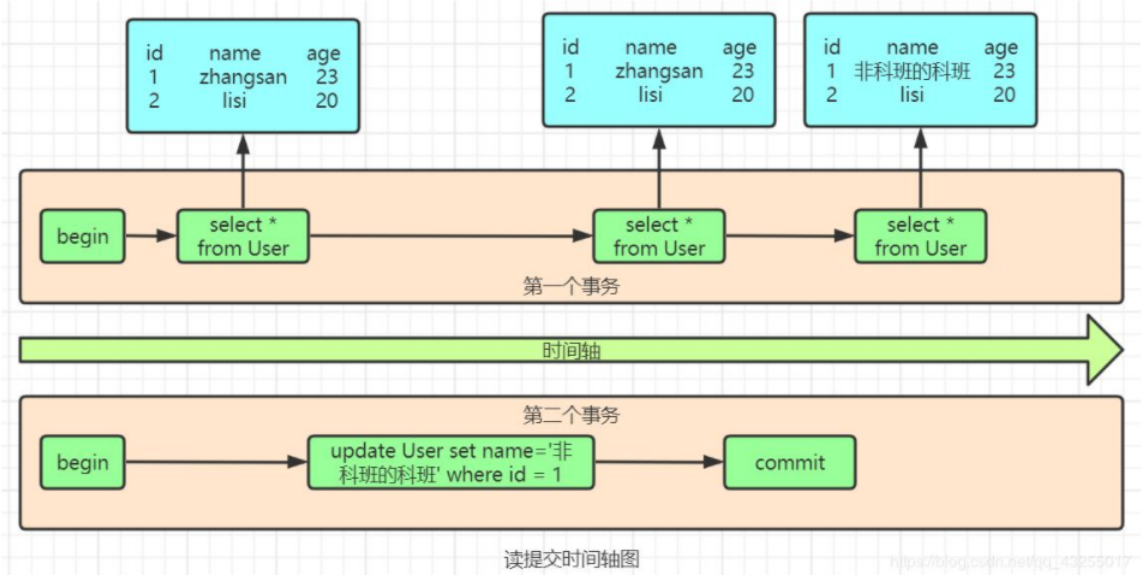
Mysql中事务的隔离级别分为四大等级，**读未提交（READ UNCOMMITTED）、读提交 （READ COMMITTED）、可重复读 （REPEATABLE READ）、串行化 （SERIALIZABLE）**。 通过set session transaction isolation level设置。

1）读未提交

有两个事务A和B，事务A先读到一个字段的值是1，事务B紧接着修改了这个字段的值为2，此时事务B还没有提交，而事务A接着又读了该字段，发现变成了2。这就是由于隔离机制设置为都未提交导致的，事务A的执行受到了事务B未提交的数据的影响，会产生脏读问题。

2）读已提交、

同样的道理，此时隔离机制设置为读已提交，那么事务B未提交的数据对事务A不可见，但是当事务B提交了之后，事务A再读取相同字段的值的时候，该值就会变成事务B修改并提交后的值2。所以不能进行重复读（update、delete）。



用到了 MVCC 多版本并发控制中一致性视图的原理。就是在事务中，每个 sql 执行前都会生成一个快照，这样当其他事务已经提交了数据时，相当于对于当前事务执行 sql 时是可以看见其他事务的修改的，这就实现了读已提交。

3）可重复读

如果希望在一个事务中，重复读取某个值的时候，该值只会受到当前事务的影响，而不会受到其他事务的影响的话，就需要设置隔离机制为可重复读。 可重复读解决了之前不可重复读和脏读的问题。

用到了 MVCC 多版本并发控制中一致性视图的原理。就是在事务开始前创建一个视图，也称为快照，然后在这个事务执行的过程中，多个 sql 都共享这个快照，也就是说其他的事务对于源数据的更改不会影响到当前的事务，这就实现了可重复读的效果。

幻读

幻读一般针对插入操作，对某个事物从无到有的幻象。比如事务A查询某个字段值，发现记录不存在。此时事务B插入了一条新记录，接着事务A同样也要插入这条数据，但是由于记录已经存在，所以报了主键冲突，然后事务A一查询，发现记录已经存在。对于同一个事务来说，记录的出现对自己不可见，这就是幻读。

MVCC 解决了读数据情况下的幻读问题 ,对于修改的操作依旧存在幻读问题，就是说MVCC对于幻读的解决时不彻底的。

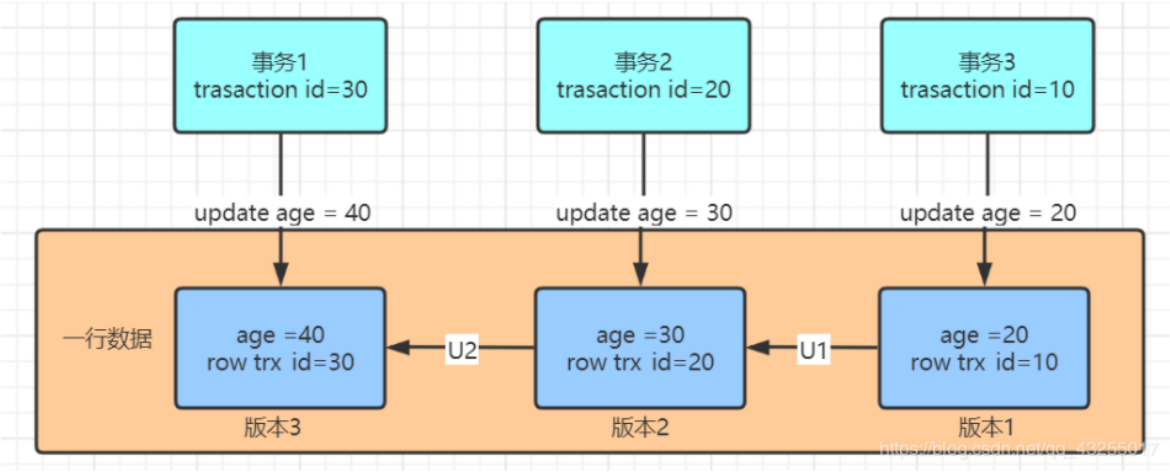
4）串行化

串行化是通过添加读写锁来保证，各个事务之间按照顺序串行执行，效率最低。

> **读未提交（产生脏读）**

> **读已提交（sql视图，不可重复读）**
>
> **可重复读（事务视图，解决了脏读和不可重复读、解决读操作下的幻读，不能解决修改操作update和delete的幻读，不彻底）**
>
> **串行化（加更多的读写锁排队执行，性能最差）**

【MVCC 的底层原理】



MVCC 多版本并发控制主要是基于一个版本机制来控制不同事务的执行逻辑的。每个事务执行前会分配一个事务id，然后事务的每个操作都会产生一个执行版本，比如对于同一行数据而言，多个事务就会存在多种不同的执行逻辑，也就是多种版本，而且版本和事务id绑定。

当事务执行时，就会根据事务id获得对应的操作版本，执行对应的逻辑。

> **事务id、执行版本、绑定**

5、数据库锁

1）读锁即共享锁，对于同一份数据，多个事务可以同时加锁，即可以并发读，但不可以写，查询语句。

2）写锁即排他锁，对于同一份数据，只要有一个事务加了锁，其他的事务不能进行读写，只能阻塞等待锁的释放，select ..for update 就是加写锁。

3）行锁，锁的粒度是行，锁住一行或者某些行，锁冲突较少，适合并发，但上锁比较慢，innodb 可以支持到行级锁。

4）表锁，锁的粒度是表，锁的冲突较大，不适合并发，但上锁的效率高， MyISAM 利用 SQL SERVER 本身的表级锁来支持表级锁。 MySQL的表级锁有两种模式：表共享读锁（Table Read Lock）和表独占写锁（Table Write
Lock） 。

5）间隙锁，就是对某个字段的值的区间来加锁，有两种，一种是区间的开区间加锁，一种是闭区间加锁。

表级锁更适合于以查询为主，只有少量按索引条件更新数据的应用，如Web应用；而行级锁则更适合于有大量按索引条件并发更新少量不同数据，同时又有并发查询的应用

> Mysql中什么时候会加锁呢?

在数据库的增、删、改、查中，只有增、删、改才会加上排它锁，而只是查询并不会加锁，只能通过在select语句后显式加lock in share mode或者for update来加共享锁或者排它锁。

> 假如两个事务执行写操作，又怎么保证并发呢?

假如事务1和事务2都要执行update操作，事务1先update数据行的时候，先回获取行锁，锁定数据，当事务2要进行update操作的时候，也会取获取该数据行的行锁，但是已经被事务1占有，事务2只能wait。若是事务1长时间没有释放锁，事务2就会出现超时异常 。

> 那没有索引的条件下呢？没办法快速定位到数据行呢？

如果没有索引的话，首先会获取所有行，将所有行上锁，然后再根据条件筛选掉一些行，只保留符合条件的行进行加锁。

> **读写锁、行锁、表锁、间隙锁**

五、日志同步

1、redo log

【脏页】

当我们想要修改DB上某一行数据的时候，InnoDB是把数据从磁盘读取到内存的缓冲池上进行修改。这个时候数据在内存中被修改，与磁盘中相比就存在了差异，我们称这种有差异的数据为**脏页**。InnoDB对脏页的处理不是每次生成脏页就将脏页刷新回磁盘，这样会产生海量的IO操作，严重影响InnoDB的处理性能。

【背景】

假设我们有一条sql语句：

```
update user_table set name='java3y' where id = '3'
```

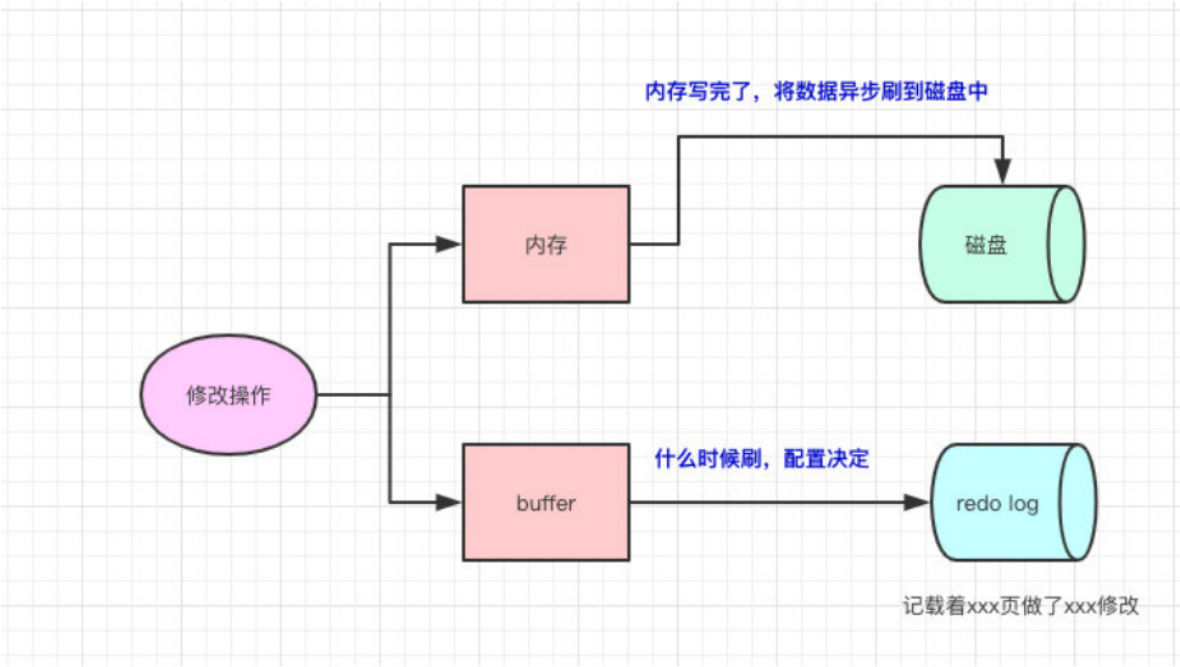
实际上Mysql的基本存储结构是**页**(记录都存在页里边)，所以MySQL是先把这条记录所在的**页**找到，然后把该页加载到内存中，将对应记录进行修改。

> 现在就可能存在一个问题：如果在内存中把数据改了，还没来得及落磁盘，而此时的数据库挂了怎么办？显然这次更改就丢了。

如果每个请求都需要将数据**立马**落磁盘之后，那速度会很慢，MySQL可能也顶不住。所以MySQL是怎么做的呢？

【工作原理】

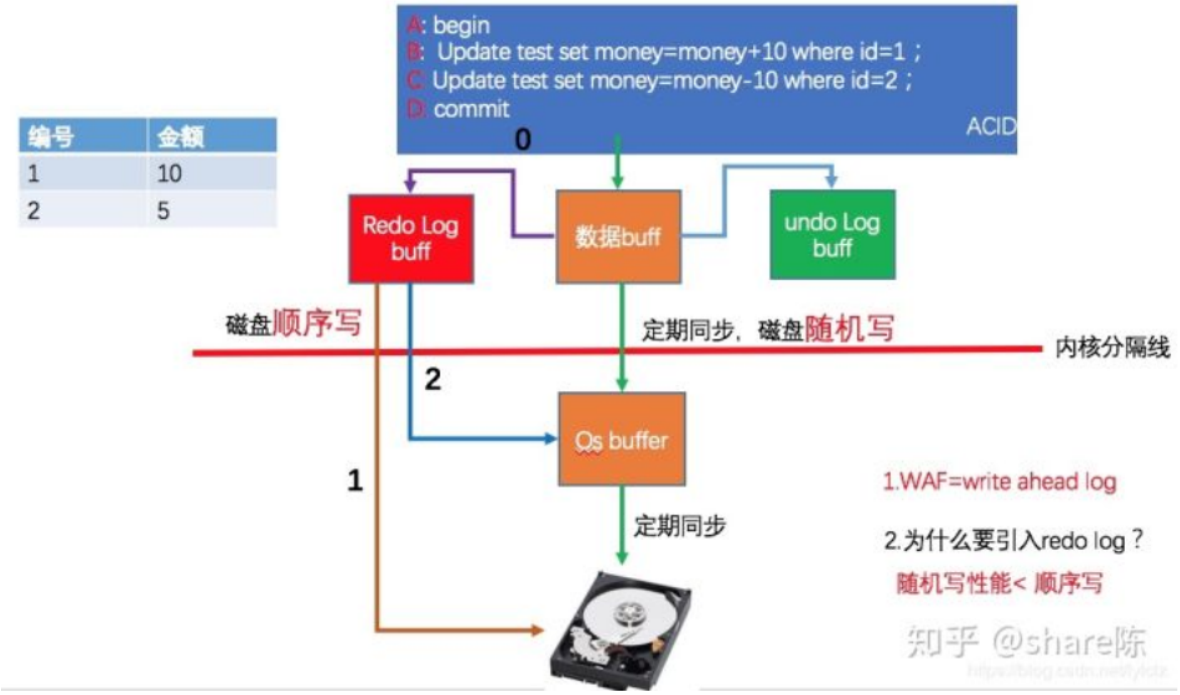
MySQL引入了`redo log`，内存写完了，然后会写一份`redo log`，这份`redo log`记载着这次**在某个页上做了什么修改**。其实写`redo log`的时候，也会有`buffer`，是先写`buffer`，再真正落到磁盘中的。至于从`buffer`什么时候落磁盘，会有配置供我们配置。



写`redo log`也是需要写磁盘的，但它的好处就是`顺序IO`（我们都知道顺序IO比随机IO快非常多）。

所以，`redo log`的存在为了：当我们修改的时候，写完内存了，但数据还没真正写到磁盘的时候。此时我们的数据库挂了，我们可以根据`redo log`来对数据进行恢复。因为`redo log`是顺序IO，所以**写入的速度很快**，并且`redo log`记载的是物理变化（xxxx页做了xxx修改），文件的体积很小，**恢复速度很快**。当对应事务的脏页写入到磁盘之后，redo log的使命也就完成了，重做日志占用的空间就可以重用（被覆盖）。

我们看这个事务，很简单，一个金额加，一个金额减 。



首先修改在内存里面修改金额，因为要支持回滚，所以修改前的数据要放到一个地方存起来，这个地方就是undo log，就是右边绿色的部分，当执行回滚操作时，直接把undo log的覆盖回来就行。修改后，数据buffer的内容定期同步到磁盘，落地的路径：数据buffer - os bufer - 硬盘。

> **一个问题： 修改数据之后，数据buffer直接刷到os buferr ,然后调用磁盘同步函数fsync，不就可以直接实现持久化吗？**

是的，可以这么做，mysql为啥没有这么做，主要原因是数据buffer直接落地的过程磁盘是一个随机写的过程。而是采取写 redo log 的方式：

1.在修改数据buffer前，把更新后的值写入到redo log中(重做日志)中

2.redo log落地后，再去修改数据buffer的值

3.当突然断电后，数据以redolog中的值为准来恢复

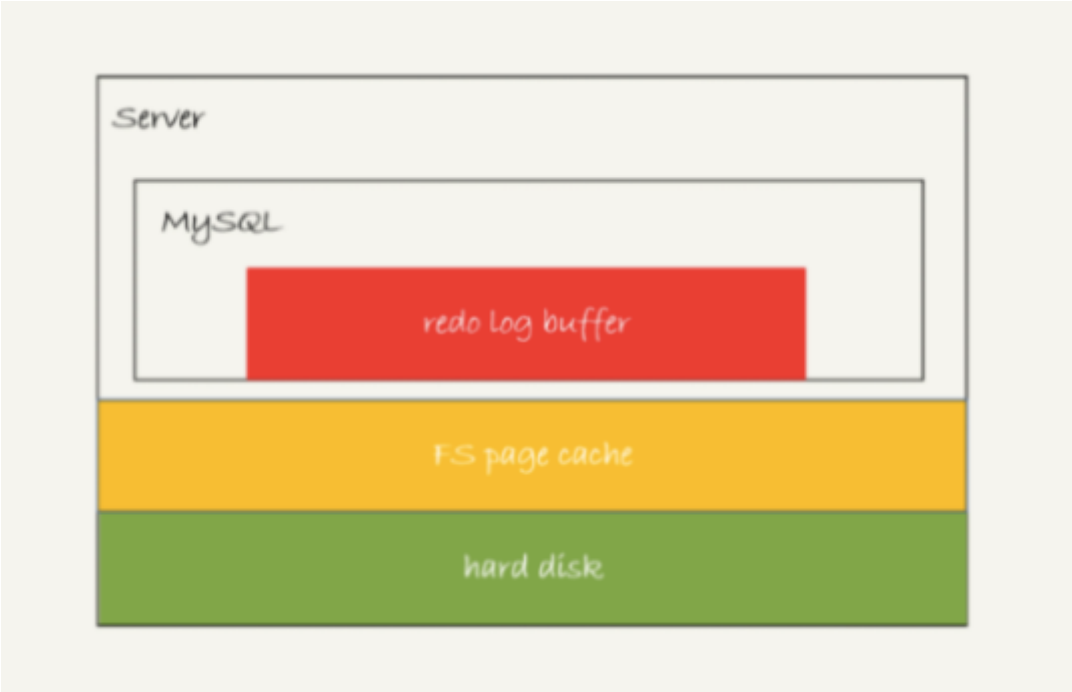
> **根据分析redo log还是要落地才能保证持久化，跟在数据buffer修改后，直接持久化有啥不同呢？**

数据buffer直接落地，磁盘是随机写，Redo log 是顺序写。

【redo log 持久化策略】

所以mysql是否持久化的关键要看redo log的持久化策略,mysql提供了三种redo log持久化策略

- 0：只写redo log，每秒落地一次，性能最高，数据一致性最差，如果mysql奔溃可能丢失一秒的数据。
- 1：表示每次事务提交时都将 redo log 直接持久化到磁盘。性能最差，一致性最高。
- 2：写redo log buffer 同时写入到os buffer，性能好，安全性也高，只要os不宕机就能保证数据一致性。当DB发生故障能恢复数据。但如果操作系统也出现宕机，那么文件系统没有及时写入磁盘的数据就会丢失。
- redo log buffer 占用的空间达到 innodb_log_buffer_size 一半的时候，后台线程会主动写盘。由于这个事务并没有提交，所以这个写盘动作只是 write，而没有调用 fsync，也就是只留在了文件系统的 page cache。



1）存在redo log buffer中,物理上是存在mysql进程内存中,就是图中红色部分

2）写到磁盘(write),但是没有持久化(fsync),物理上是在文件系统的page cache里面,也就是图中得黄色部分

3）持久化到磁盘,对应的是hard disk,也就是图中得绿色部分

日志写到 redo log buffer 是很快的，wirte 到 page cache 也差不多，但是持久化到磁盘的速度就慢多了。无论是写redo log，还是写binlog，都是先写缓存，再fsync到磁盘。

> 注意，事务执行中间过程的 redo log 也是直接写在 redo log buffer 中的，这些 redo log 也会被后台线程一起持久化到磁盘。也就是说，一个没有提交的事务的 redo log，也 是可能已经持久化到磁盘的。因此重做日志的写盘，并不一定是随着事务的提交才写入重做日志文件的，而是随着事务的开始，逐步开始的。

【适用场景】

所以数据使用哪种持久化方式要看应用具体场景

- 支付类场景数据一致性要求高，适合1
- 文章评论等场景，适合0
- 订单类个人认为，适合2

【写文件】

redo 日志是循环写入的，从头开始写，写到末尾就又回到开头循环写 。redo log 的大小是固定的，日志上的记录修改落盘后，日志会被覆盖掉，无法用于数据回滚/数据恢复等操作。

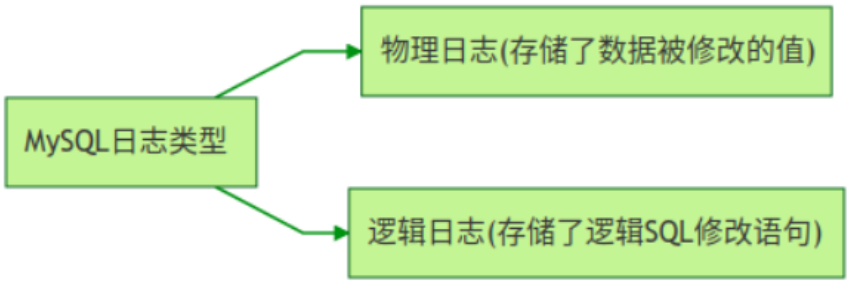
checkpoint

check point 表示将日志记录的修改写进磁盘，完成数据落盘，数据落盘后checkpoint 会将日志上的相关记录擦除掉。所以，当DB重启恢复时，只需要恢复checkpoint之后的数据。这样就能大大缩短恢复时间。

LSN(Log Sequence Number)

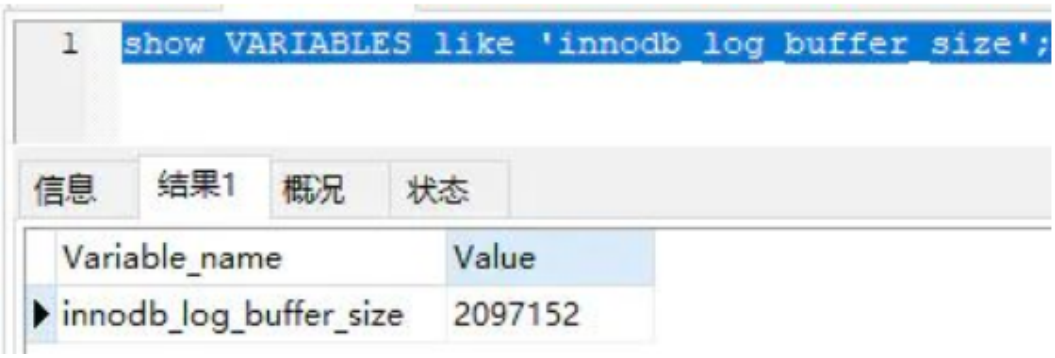
LSN实际上就是InnoDB使用的一个版本标记的计数，它是一个单调递增的值。数据页和redo log都有各自的LSN。我们可以根据数据页中的LSN值和redo log中LSN的值判断需要恢复的redo log的位置和大小。

日志类型



redo log在数据库重启恢复的时候被使用，因为其属于物理日志的特性，恢复速度远快于逻辑日志。而我们经常使用的binlog就属于典型的逻辑日志。

我们可以通过`show VARIABLES like 'innodb_log_buffer_size'`命令来查看当前的日志缓存大小，下图为线上的大小。



总结

- Mysql由于cpu速度与磁盘速度的鸿沟，引入缓存池的概念，数据的写都是先在内存中操作，然后异步刷新到此磁盘
- Mysql buffer刷新到磁盘的过程是一个随机写的过程
- Mysql通过redo log 顺序写的优势来保证内存与磁盘的数据一致性
- Redo log的刷盘策略决定了mysql事务的强一致性，还是弱一致性
- 事务的设置强一致性还是弱一致性要看具体使用场景

2、binlog

【存储的内容】

`binlog`记载的是`update/delete/insert`这样的SQL语句，而redo log记录的是"物理级别"上的页修改操作，比如页号xxx、偏移量yyy写入了'zzz'数据。`redo log` 记录的是数据的**物理变化**，`binlog` 记录的是数据的**逻辑变化**。

【功能】

`redo log`的作用是为**持久化**而生的。写完内存，如果数据库挂了，那我们可以通过`redo log`来恢复内存还没来得及刷到磁盘的数据，将`redo log`加载到内存里边，那内存就能恢复到挂掉之前的数据了。

`binlog`的作用是复制和恢复而生的。

- 主从服务器需要保持数据的一致性，通过`binlog`来同步数据。
- 如果整个数据库的数据都被删除了，`binlog`存储着所有的数据变更情况，那么可以通过`binlog`来对数据进行恢复。用于数据库的基于时间点的还原。

【工作原理】

其实，binlog 的写入逻辑比较简单。事务执行过程中，先把日志写到 binlog cache，事务提交的时候，再把 binlog cache 写到 binlog 文件中。系统给 binlog cache 分配了一片内存，如果超过了这个参数规定的大小，就要暂存到磁盘。

> **又看到这里，你会想：”如果整个数据库的数据都被删除了，那我可以用`redo log`的记录来恢复吗？“不能**

因为功能的不同，`redo log` 存储的是物理数据的变更，如果我们内存的数据已经刷到了磁盘了，那 `redo log` 的数据就无效了。所以 `redo log` 不会存储着**历史**所有数据的变更，**文件的内容会被覆盖的。**

【binlog 二进制日志包括两类文件】

- 索引文件（文件名后缀为.index）用于记录哪些日志文件正在被使用
- 日志文件（文件名后缀为.00000*）记录数据库所有的DDL和DML(除了数据查询语句)语句事件。

说到日志文件。在 `innodb` 里其实又可以分为两部分，一部分在缓存中，一部分在磁盘上。这里业内有一个词叫做**刷盘**，就是指将缓存中的日志刷到磁盘上。跟**刷盘**有关的参数有两个: `sync_binlog` 和 `binlog_cache_size` 。这两个参数作用如下

```
binlog_cache_size: 二进制日志缓存部分的大小，默认值32k

sync_binlog=[N]: 表示写缓冲多少次，刷一次盘,默认值为0
```

注意两点:

- (1) `binlog_cache_size` 设过大，会造成内存浪费。 `binlog_cache_size` 设置过小，会频繁将缓冲日志写入临时文件。
- (2) `sync_binlog=0` :表示刷盘的频率和时机，操作系统自身会每隔一段时间就会刷新缓存数据到磁盘，这个性能最好。 `sync_binlog=1` ，代表每次事务提交时就会刷新binlog到磁盘。 `sync_binlog=N` ,代表每N个事务提交会进行一次binlog刷新。

【数据丢失问题】

另外，这里存在一个一致性问题， `sync_binlog=N` ，数据库在操作系统宕机的时候，可能数据并没有同步到磁盘，于是再次重启数据库，会带来数据丢失问题。
当 `sync_binlog=1` ，**事务在 `commit` 的时候，数据写入 `binlog` ，但是还没写入事务日志**(`redo log` 和 `undo log`)。此时宕机，重启数据库，**数据被回滚。但是 `binlog` 里已经记录**，这里存在不一致问题。这个事务日志和 `binlog` 一致性的问题，大家可以查询mysql的内部**XA协议**，该协议就是解决这个一致性问题的。

> **索引文件、日志文件、binlog_cache_size、sync_binlog（刷盘频率）、XA**

【常见格式】

STATMENT模式：基于SQL语句的复制(statement-based replication, SBR)，**每一条会修改数据的sql语句会记录到binlog中。**

ROW模式：不记录每一条SQL语句的上下文信息，**仅需记录哪条数据被修改了，修改成了什么样子了。**

业内目前推荐使用的是 `row` 模式，准确性高，虽然说文件大，但是现在有SSD和万兆光纤网络，磁盘 IO 和网络 IO 负载能力大，这些磁盘IO和网络IO都是可以接受的。

> **语句模式（对数据修改的每条sql都会记录，文件小、节约磁盘和网络IO、准确性差一些）**
>
> **数据行模式（记录哪行数据被修改了，改成什么样子了，文件大、磁盘和网络IO大、准确性强）**

3、undo log

`undo log`主要有两个作用：回滚和多版本控制(MVCC)

在数据修改的时候，不仅记录了`redo log`，还记录`undo log`，如果因为某些原因导致事务失败或回滚了，可以用`undo log`进行回滚。

`undo log`主要存储的也是逻辑日志，比如我们要`insert`一条数据了，那`undo log`会记录的一条对应的`delete`日志。我们要`update`一条记录时，它会记录一条对应**相反**的update记录。

这也应该容易理解，毕竟回滚嘛，跟需要修改的操作相反就好，这样就能达到回滚的目的。因为支持回滚操作，所以我们就能保证：“**一个事务包含多个操作，这些操作要么全部执行，要么全都不执行**”。【原子性】

因为`undo log`存储着修改之前的数据，相当于一个**前版本**，MVCC实现的是读写不阻塞，读的时候只要返回前一个版本的数据就行了。

4、宕机恢复

DB宕机后重启，InnoDB会首先去数据库查看数据页中的LSN的数值。这个值代表数据页被刷新回磁盘的LSN的大小。然后再去查看redo log的LSN的大小。如果数据页中的LSN值大说明数据页领先于redo log刷新回磁盘，不需要进行恢复。反之，可通过redo log将未落盘的数据恢复，即保证已经提交的事务记录不会丢失。

【binlog 和 redo log 区别】

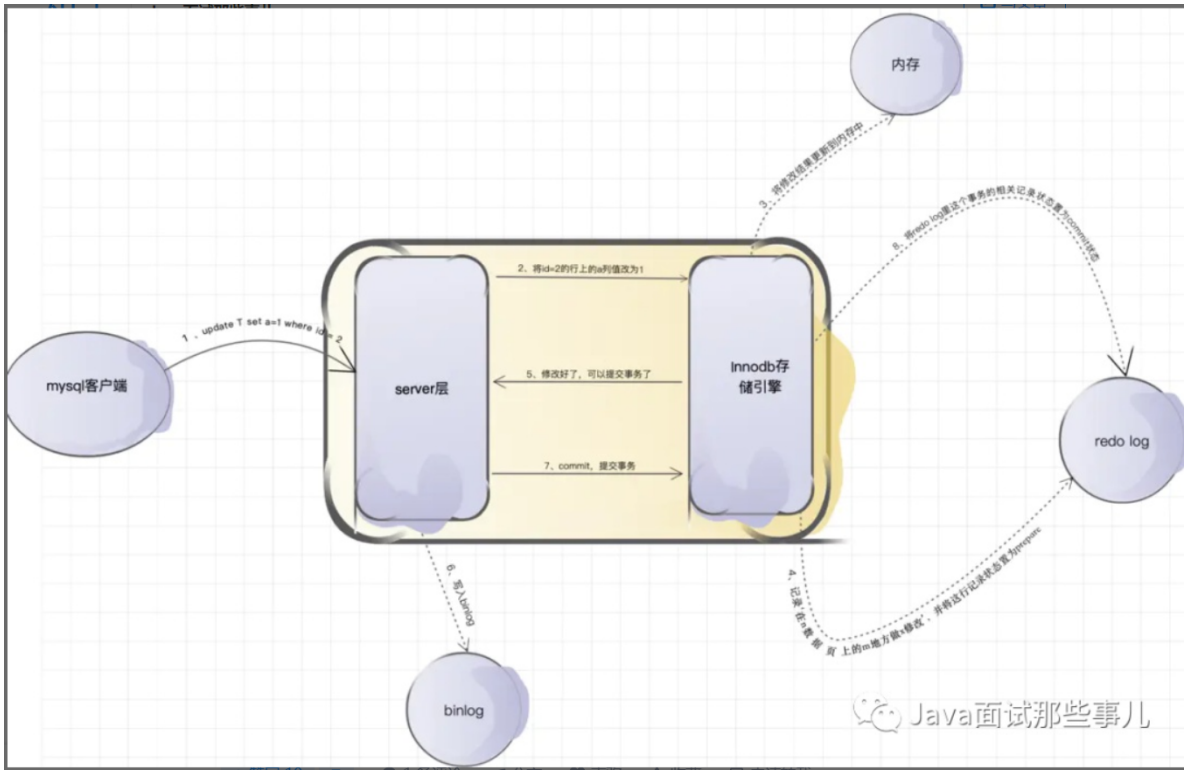
1）角色不同。redo log 是 innodb 引擎层实现的，并不是所有引擎都有。binlog 是 server 层实现的，意味着所有引擎都可以使用 binlog 日志。

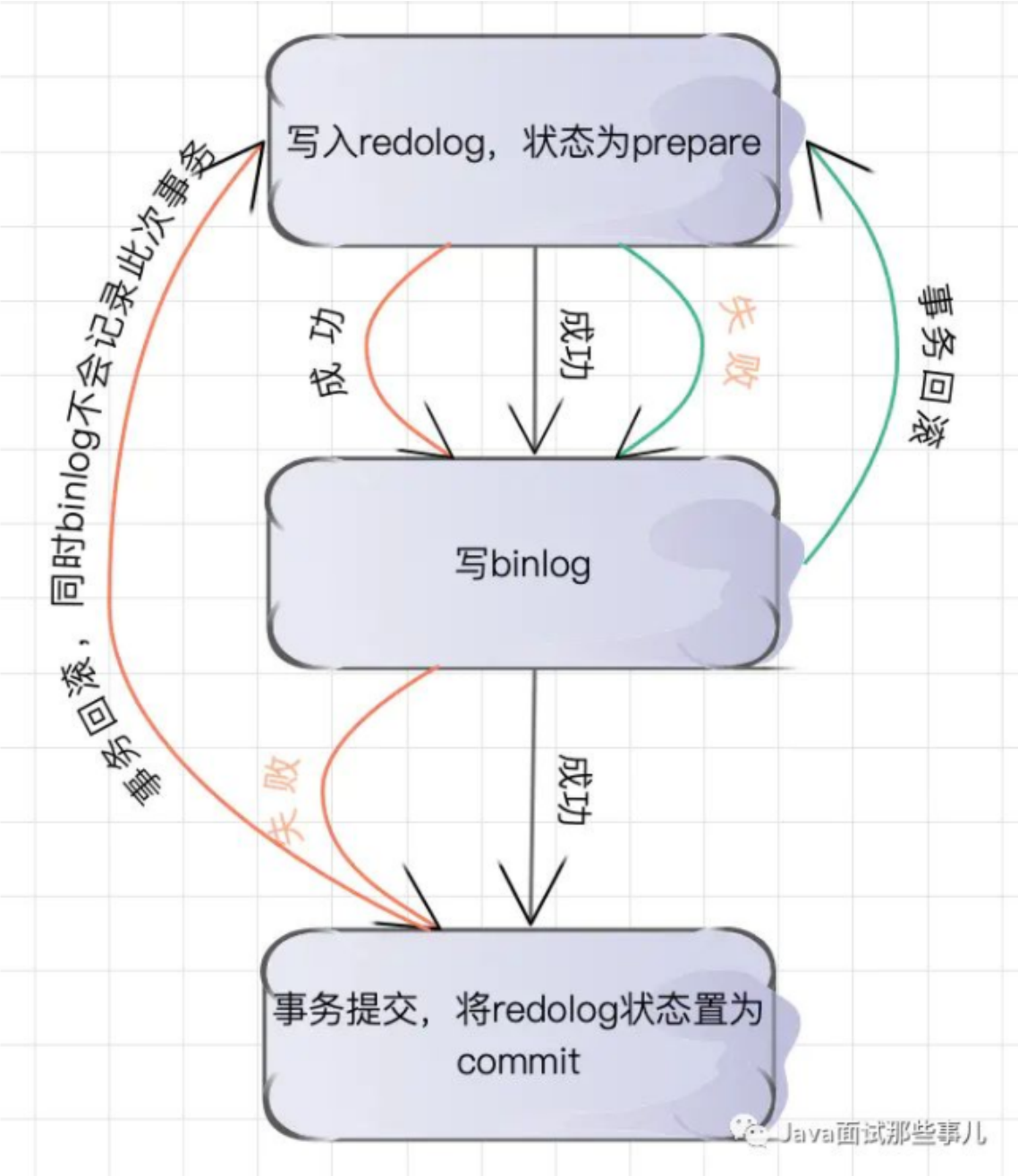
2）恢复数据的侧重点不同。redo log 侧重于将内存中已提交更新的，由于数据库奔溃而没有来得及刷新到磁盘中的数据，在重启后可以重新刷到磁盘中。binlog 一方面侧重于当整个数据库数据丢失时，可以根据 binlog 文件的某个时间点来重放执行语句，达到数据恢复的作用，另一方面用作主从同步。

3）数据内容不同。redo log 是物理日志，记录某个页面某条数据被修改成什么样子。而 binlog 是逻辑日志，可以简单理解为记录修改的 Sql 语句。

4）相同点，都需要先写入缓存，再持久化到硬盘。

5、binlog 与 redo log 同步


内存
3、将修改结果更新到内存中
8、将redo log里这个事务的相关记录状态置为commit状态
mysql客户端
1、update T set a=1 where id=2
server层
2、将id=2的行上的a列值改为1
5、修改好了，可以提交事务了
7、commit，提交事务
Innodb存
储引擎
redo log
6、写入binlog
4、记录在redo log上的哪个地方做了修改，并将这行记录状态置为prepare
binlog
Java面试那些事儿



MySQL通过两阶段提交过程来完成事务的一致性的，也即redo log和binlog的一致性的。理论上是先写redo log，再写binlog，两个日志都提交成功（刷入磁盘），事务才算真正的完成。

准备：数据已在内存中更新；提交：数据要写入磁盘

【两阶段提交】

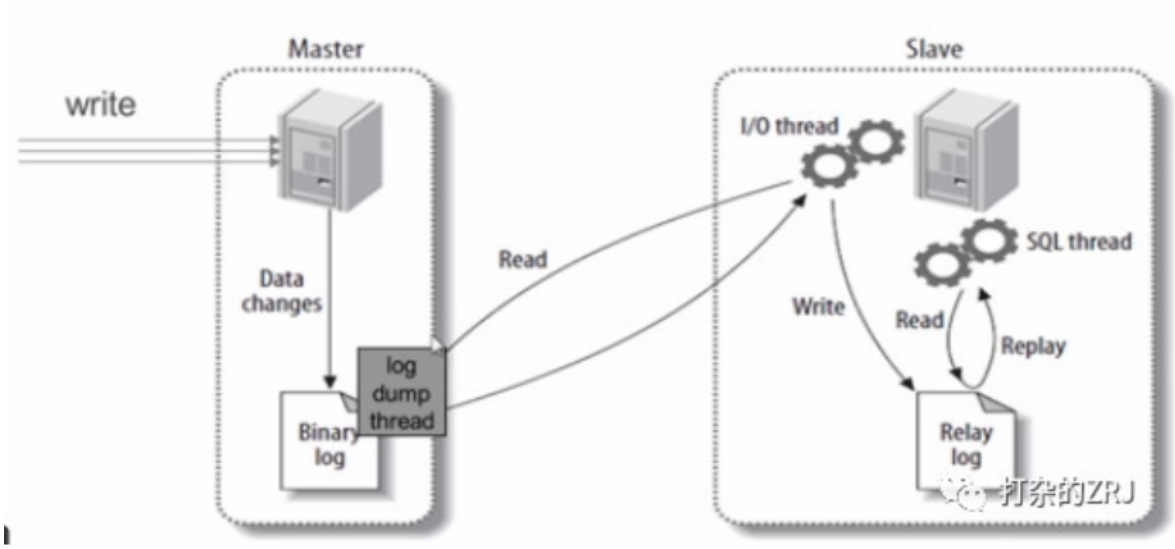
`redo log`**事务开始**的时候，就开始记录每次的变更信息，而`binlog`是在**事务提交**的时候才记录。

于是新有的问题又出现了：我写其中的某一个`log`，失败了，那会怎么办？现在我们的前提是先写`redo log`，再写`binlog`，我们来看看：

- 如果写`redo log`失败了，那我们就认为这次事务有问题，回滚，不再写`binlog`。
- 如果写`redo log`成功了，写`binlog`，写`binlog`写一半了，但失败了怎么办？我们还是会对这次的**事务回滚**，将无效的`binlog`给删除（因为`binlog`会影响从库的数据，所以需要做删除操作）
- 如果写`redo log`和`binlog`都成功了，那这次算是事务才会真正成功。

简单来说：MySQL需要保证`redo log`和`binlog`的**数据是一致**的，如果不一致，那就乱套了。

6、主从同步



主库有一个 `log dump` 线程，将 `binlog` 传给从库。
从库有两个线程，一个I/O线程，一个SQL线程，I/O线程读取主库传过来的 `binlog` 内容并写入到 `relay log` 重放日志中，SQL线程从 `relay log` 里面读取内容进行 SQL 语句的重放，写入从库的数据库。

> **binlog、重放日志、执行重放**

7、同步延迟

定位：

通过监控 show slave status 命令输出的 Seconds_Behind_Master 参数的值：

- NULL，表示io_thread或是sql_thread有任何一个发生故障
- 0，该值为零，表示主从复制良好
- 正值，表示主从已经出现延时，数字越大表示从库延迟越严重

```
      Relay_Master_Log_File: mysql-bin.004014
           Slave_IO_Running: Yes
          Slave_SQL_Running: Yes
            Replicate_Do_DB:
        Replicate_Ignore_DB: mysql,xunlei_s38_50hero_combats
         Replicate_Do_Table:
     Replicate_Ignore_Table:
    Replicate_Wild_Do_Table:
Replicate_Wild_Ignore_Table:
                 Last_Errno: 0
                 Last_Error:
               Skip_Counter: 0
        Exec_Master_Log_Pos: 72560192
            Relay_Log_Space: 72560662
            Until_Condition: None
             Until_Log_File:
              Until_Log_Pos: 0
         Master_SSL_Allowed: No
         Master_SSL_CA_File:
         Master_SSL_CA_Path:
            Master_SSL_Cert:
          Master_SSL_Cipher:
             Master_SSL_Key:
      Seconds_Behind_Master: 0
```

原因：

1）因为从节点的同步都是用单线程进行同步的，没有办法利用并发的优势，所以当主节点的并发请求较高时，产生的sql 语句数量（建表、建索引）超过从节点的 sql 线程的处理能力时，主节点这边的 sql 就需要堆积等待同步和重放，那么延时就产生了。

2）从节点的 sql 线程在重放语句的时候，在磁盘上查找这些语句是随机读的，这样一来 IO 的开销相对较多，而且在执行的时候，还有可能和其他的查询产生竞争锁的情况，造成了锁的等待，如果等待时间长，也会造成延时。

3）从节点有慢 sql，或者在执行比较大的事务等，都会造成 sql 重放的速度变慢，造成等待。

4）从节点的读写压力比较大，且本身的硬件设施的性能比较差，导致 CPU、内存、磁盘、网络等资源在高并发同步和访问时都比较紧张，所以执行 sql 重放的每条语句时，速度会下降，周期变长，吞吐下降，也会造成延时。

解决：

具体问题具体分析。

1）先解决慢查询语句，排查慢的原因，进行优化；以及优化大的事务，将大的事务拆分成多个小事物进行提交；

2）然后再进行架构的优化，进行读写分离，部署适当多的从数据库（3-5个），主数据库负责写，从数据库负责读，分担每个节点的负载压力。进行分库，把一个主库通过路由规则拆分成多个主库，降低每个主库的并发写压力。如果从数据库的读压力大，还可以考虑分表。节点增加缓存层等。

3）升级设备硬件性能，可以用各种命令排查下性能的瓶颈是出在哪里，先解决问题，如果没问题了确定是硬件本身性能太差的话，如果是内存的话就增加内存，如果是磁盘的话就升级SSD，网卡的话看能否增加带宽等。

4）设置数据库的日志同步参数。可以把 binlog 日志的同步参数设置为1，表示每次事务提交都进行数据的同步刷盘；在 innodb 引擎中也有参数可以设置为1，表示每次事务提交都把 redo log buffer 中的数据同步刷盘。但这种具体还是要看业务场景，如果是对数据安全和一致性比较敏感的业务领域可以考虑，但是在高并发的场景下对吞吐量有要求的这种就不合适了。

5）升级数据库，用最新的多线程并行复制。

8、增量备份

在MySQL数据库中，备份支持增量备份的。如果每次都使用全量备份，很消耗性能不说，还占用磁盘空间。

首先在进行增量备份之前需要查看一下配置文件，查看 log_bin 是否开启，因为要做增量备份首先要开启 log_bin 。首先，进入到 myslq 命令行，输入如下命令：

show variables like '%log_bin%';

查看 binlog 文件

```
mysqlbinlog mysql-bin.000001
```

增量备份是依靠mysql-bin日志生成增量数据，因此，需要打开mysql-bin日志

```
vi /etc/my.cnf
```

log-bin=mysql-bin #添加这一行就ok、binlog-format=ROW #选择row模式

> **配置文件打开 log-bin、show variables like '%log_bin%'**

9、总结

背景

redo log 的核心是基于顺序写磁盘的机制，极大降低了内存数据的丢失量。在没有 redo log 之前，修改一条数据，首先需要将数据从磁盘读取到内存的 buffer 中进行修改，然后再根据预设的刷盘策略，进行定时的刷盘，或者每次提交或者每n次提交就会进行一次刷盘。这种刷盘的方式是随机查找的方式，需要很多的磁头寻道的时间，如果在这段时间内，数据库奔溃了，那么就会造成这段相当长的时间内在内存中的数据的丢失。

redo log 工作机制

那么这种丢数据的本质原因是寻道时间太长，如果这个寻道时间很短，那么意味着刷盘时间很快，即使有数据丢失，也只会丢很少的数据量。所以就产生了基于顺序写磁盘机制的 redo log 机制。它的工作原理首先还是先将修改后的数据写入内存的 redo log buffer 中，这点和 binlog 都是一样的，因为 CPU 直接与低速的磁盘 IO 交互开销大，所以数据先放入缓存中。接着，redo log buffer 就可以直接同步到磁盘中的 redo log 文件了，由于它是顺序写的，磁头寻道时间很短，所以很多内存的数据都可以及时地同步到磁盘，相比于 binlog 来说，即使数据库在某个时刻奔溃了，redo log 也只会丢很少的数据。

刷盘策略与应用

而决定 redo log 具体要丢多少数据，和三种 redo log 的刷盘策略有关。我简单介绍一下，有个刷盘参数，如果设置为0，则会有一个后台线程每秒将 redo log buffer 进行刷盘，这种策略性能最高，但一致性最差，如果 mysql 奔溃可能会丢失一秒的数据。适用于文章、评论类等弱一致性的场景。如果参数是1，则表示每次事务提交都进行刷盘，这种性能最差，但一致性最高，适用于金融类或者支付等强一致性的场景。最后一种参数是2，,是在性能和一致性中间取个平衡， 数据写入 redo log buffer 的同时也会写入操作系统的页缓存中，然后等提交的时候再统一将页缓存的数据进行刷盘，这种机制只要操作系统不奔溃，即使 mysql 奔溃了，由于数据在页缓存中，还能进行恢复，而且不用每次都同步刷盘，这样的话性能和可靠性都得到保障，适用于订单类等中等一致性的业务场景。除此以外，如果redo log buffer 的容量达到一半，也会主动将 redo log buffer 刷新到页缓存中，但这种不是主动选择的。

流程与恢复

所以当有了 redo log 机制以后，数据就先写 redo log buffer，然后 redo log 再刷盘，接着才去修改内存的数据，然后再根据 binlog 的刷盘策略去刷盘。当数据库奔溃的时候，InnoDB 会对比一下数据库中数据的偏移量以及redo log 文件中数据的偏移量，如果数据库的偏移量小于 redo log 的偏移量，说明有部分数据还没有刷盘，则根据 redo log 的数据从偏移量开始处进行刷盘，这里的偏移量有个专业的名称叫 LSN。还有另一种数据恢复的场景，就是当数据库的数据不小心被删除了，这个时候就只能用 binlog 文件，根据某个时间点进行一个数据重放。这里不能用 redo log，原因是 redo log 强调的是从内存缓存到数据库的刷盘，当刷盘结束后，redo log 会在文件中提交一个检查点，表示检查点之前的数据已经刷盘了，然后新添加到 redo log 的数据将会覆盖检查点之前的数据，所以 redo log 文件是循环写入的，数据会被覆盖，不能用于这种数据库数据的全面恢复。

而在第一种数据恢复的场景中，还有一个点需要注意就是要保证 redo log 和 bin log 的数据一致性，在 Innodb 中是通过两阶段提交来实现的。首先第一阶段数据先写入 redo log buffer，置状态位为 prepared，然后修改内存数据，第二个阶段是数据写入 binlog buffer，然后开始提交，redo log buffer 的状态位变成 commit，然后两个文件都进行提交，只有两个文件都成功了，才能说明数据保持了一致，否则就会进行回滚。

比较

最后再简单说一下这两个文件的几点不同。

1）角色不同。redo log 是 innodb 引擎层实现的，并不是所有引擎都有。binlog 是 server 层实现的，意味着所有引擎都可以使用 binlog 日志。

2）恢复数据的侧重点不同。redo log 侧重于将内存中已提交更新的，由于数据库奔溃而没有来得及刷新到磁盘中的数据，在重启后可以重新刷到磁盘中。binlog 一方面侧重于当整个数据库数据丢失时，可以根据 binlog 文件的某个时间点来重放执行语句，达到数据恢复的作用。

3）数据内容不同。redo log 是物理日志，记录某个页面某条数据被修改成什么样子。而 binlog 是逻辑日志，可以简单理解为记录修改的 Sql 语句。

4）相同点，都需要先写入缓存，再持久化到硬盘。

还有一个 undo log，记录的是数据修改前的状态版本，主要用于事务执行失败时的一个回滚，存储的是修改语句的反向语句，比如 insert 插入一条数据，那么相对应的 undo log 会生成一条 delete 删除语句。

六、语句执行性能排查

1、一条SQL语句执行得很慢的原因有哪些?

【分类讨论】

一条 SQL 语句执行的很慢，那是每次执行都很慢呢？还是大多数情况下是正常的，偶尔出现很慢呢？所以我觉得，我们还得分以下两种情况来讨论。

1、大多数情况是正常的，只是偶尔会出现很慢的情况。

2、在数据量不变的情况下，这条SQL语句一直以来都执行的很慢。

【针对偶尔很慢的情况】

1）数据库在刷新脏页

当我们要往数据库插入一条数据、或者要更新一条数据的时候，我们知道数据库会在**内存**中把对应字段的数据更新了，但是更新之后，这些更新的字段并不会马上同步持久化到**磁盘**中去，而是把这些更新的记录写入到日志缓冲中，然后当缓冲区满或者让后台线程刷新到磁盘中的 redo log 文件中，那么在刷盘的时候，mysql 就会投入很多资源来做这个事情，**就会导致我们平时正常的SQL语句突然执行的很慢**，所以说，数据库在在同步数据到磁盘的时候，就有可能导致我们的SQL语句执行的很慢了。

2）数据被锁住

我们要执行的这条语句，刚好这条语句涉及到的**表**，别人在用，并且加锁了，我们拿不到锁，只能慢慢等待别人释放锁了。或者，表没有加锁，但要使用到的某个一行被加锁了。

以上两种情况都可以通过show processlist这个命令来查看当前的状态。

> **刷新脏页、锁数据、show processlist**

【针对一直都这么慢的情况】

1）字段没有加索引

2）字段加了索引，但是没有走索引

系统是通过**索引的识别度**来判断的，一个索引上不同的值越多，意味着索引的识别度越高。 意味着走索引查询越有优势。

索引系统是通过遍历部分数据，也就是通过**采样**的方式，来预测索引的区分度， **由于统计的失误，导致系统没有走索引，而是走了全表扫描。**

【针对这种情况可以通过 explain 来查看 sql 的执行计划以及查看慢查询日志】

通过强制走索引的方式来查询，例如

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

我们也可以通过

```
show index from t;
```

来查询索引的基数和实际是否符合，如果和实际很不符合的话，我们可以重新来统计索引的基数，可以用这条命令

```
analyze table t;
```

来重新统计分析。

2、优化器的成本计算

【成本】

1.I/O成本。将某个数据从磁盘加载到内存中，这个过程的时间被称为I/O成本。默认是1。

2.CPU成本。在内存对结果集进行排序的时间被称为CPU成本。默认是0.2。

先来建一个用户表dev_user，里面包括主键id，用户名username，密码password，外键user_info_id，状态status，外键main_station_id，是否外网访问visit，这七个字段。索引有两个，一个是主键的聚簇索引，另一个是显式添加的以username为字段的唯一索引uname_unique。

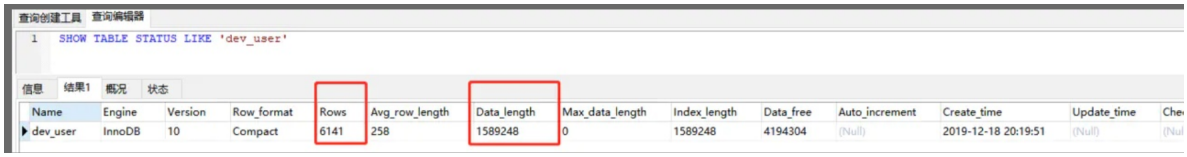
> 如果搜索条件是select * from dev_user where username='XXX'，那么MySQL是如何选择相关索引呢?

1） 使用所有可能用到的索引

可能走 username 的字段索引，可能走主键id 的聚集索引，也可能走全表扫描。

2） 计算全表扫描代价

我们通过`show table status like ‘dev_user’`命令知道`rows`和`data_length`字段



rows：表示表中的记录条数，但是这个数据不准确，是个估计值。

data_length:表示表占用的存储空间字节数。

```
data_length=聚簇索引的页面数量X每个页面的大小
```

反推出页面数量=1589248÷16÷1024=97

I/O成本：97X1=97

CPU成本：6141X0.2=1228

总成本：97+1228=1325

3） 计算使用不同索引执行查询的代价

4） 对比各种执行方案的代价，找出成本最低的那个

> **没有加索引、没有走索引、语句优化、explain、慢查询日志**
>
> **计算全表查询和不同索引的查询成本**
>
> **show table status like 成本：IO（页面 x 单位成本1）+ CPU成本（表行总数 x 单位成本0.2)**

3、show processlist

show processlist 命令非常实用，有时候mysql经常跑到50%以上或更多，就需要用这个命令看哪个sql语句占用资源比较多，就知道哪个网站的程序问题了。

show processlist 命令的输出结果显示了有哪些线程在运行，可以帮助识别出有问题的查询语句；如果有SUPER权限，则可以看到全部的线程，否则，只能看到自己发起的线程（这是指当前对应的MySQL帐户运行的线程）。

参数含义：

```
mysql> show processlist;
+-----+------+-----------------+----------+---------+-------+-------+------------------+
| Id  | User | Host            | db       | Command | Time  | State | Info             |
+-----+------+-----------------+----------+---------+-------+-------+------------------+
|  50 | root | localhost:3995  | NULL     | Sleep   | 20114 |       | NULL             |
|  51 | root | localhost:3996  | dss_data | Sleep   | 10884 |       | NULL             |
|  52 | root | localhost:3997  | dss_game | Sleep   | 20111 |       | NULL             |
| 659 | root | localhost:14237 | NULL     | Query   |     0 | NULL  | show processlist |
+-----+------+-----------------+----------+---------+-------+-------+------------------+
4 rows in set (0.00 sec)
```

- user列，显示单前用户，如果不是root，这个命令就只显示你权限范围内的sql语句。
- **host**列，显示这个语句是从哪个ip的哪个端口上发出的。呵呵，可以用来追踪出问题语句的用户。
- **db**列，显示这个进程目前连接的是哪个数据库。
- command列，显示当前连接的执行的命令，一般就是休眠（sleep），查询（query），连接（connect）。
- **time**列，此这个状态持续的时间，单位是秒。
- **state**列，显示使用当前连接的sql语句的状态，很重要的列，后续会有所有的状态的描述，请注意，state只是语句执行中的某一个状态，一个sql语句，已查询为例，可能需要经过copying to tmp table，Sorting result，Sending data等状态才可以完成

大部分状态对应很快的操作，只要有一个线程保持同一个状态好几秒钟，那么可能是有问题发生了。

> Closing tables

正在将表中修改的数据刷新到磁盘中，同时正在关闭已经用完的表。这是一个很快的操作，如果不是这样的话，就应该确认磁盘空间是否已经满了或者磁盘是否正处于重负中。

> Flushing tables

正在执行FLUSH TABLES，等待其他线程关闭数据表。

> Locked

被其他查询锁住了。

> Sleeping

> 正在等待客户端发送新请求.

> **查看有哪些线程在运行，以及线程对应的语句的执行状态和持续的时间等**
>
> **host、db、time、status（flushing tables、closing tables、locked、sleeping）**

4、慢查询日志

MySQL通过慢查询日志定位那些执行效率较低的SQL 语句，mysqld 会将一个包含所有执行时间超过long_query_time 秒的SQL语句的日志文件，通过查看这个日志文件定位效率较低的SQL 。

1，slow_query_log

这个参数设置为ON，可以捕获执行时间超过一定数值的SQL语句。

2，long_query_time

当SQL语句执行时间超过此数值时，就会被记录到日志中，建议设置为1或者更短。

```
SET timestamp=1543979328;
SHOW FULL TABLES FROM `engine_gp` WHERE Table_type = 'VIEW';
# Time: 2018-12-05T03:08:48.630972Z
# User@Host: root[root] @  [192.168.8.45]  Id:    41
# Query_time: 0.001323  Lock_time: 0.000279 Rows_sent: 0  Rows_examined: 29
SET timestamp=1543979328;
SHOW PROCEDURE STATUS WHERE Db='engine_gp';
# Time: 2018-12-05T03:08:48.641976Z
# User@Host: root[root] @  [192.168.8.45]  Id:    41
# Query_time: 0.001444  Lock_time: 0.000183 Rows_sent: 1  Rows_examined: 27
SET timestamp=1543979328;
SHOW FUNCTION STATUS WHERE Db='engine_gp';
# Time: 2018-12-05T05:49:16.444578Z
# User@Host: root[root] @ localhost []  Id:    42
# Query_time: 0.000182  Lock_time: 0.000000 Rows_sent: 1  Rows_examined: 0
SET timestamp=1543988956;
select @@version_comment limit 1;
/usr/sbin/mysqld, Version: 5.7.23 (MySQL Community Server (GPL)). started with:
Tcp port: 3306  Unix socket: /var/lib/mysql/mysql.sock
Time                 Id Command    Argument
# Time: 2018-12-05T14:40:02.506931Z
# User@Host: root[root] @ localhost []  Id:     6
# Query_time: 0.000479  Lock_time: 0.000192 Rows_sent: 1  Rows_examined: 1
use gp-mysql;
SET timestamp=1544020802;
select * from users limit 1;
"/var/lib/mysql/gupaoedu-slow.log" 925L, 47634C
```

Time： 日志记录的时间
User@Host：执行的用户及主机
Query_time：查询耗费时间 Lock_time 锁表时间 Rows_sent 发送给请求方的记录条数 Rows_examined 语句扫描的记录条数
SET timestamp 语句执行的时间点
select .... 执行的具体语句

> **slow_query_log、log_query_time、用户及主机、查询时间、锁表时间**

5、explain 查看执行计划

查询创建工具 查询编辑器

```
1 explain select * from dev_user where username='admin'
```

信息 结果1 概况 状态

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	dev_user	const	UNAME_UNIQUE	UNAME	302	const	1	(Null)

各参数详解

id

一般来说一个select一个唯一id

select_type

- simple：不包括union和子查询的查询都算simple类型。
- primary：包括union，union all，其中最左边的查询即为primary。
- union：包括union，union all，除了最左边的查询，其他的查询类型都为union。

select_type	类型说明
SIMPLE	简单SELECT(不使用UNION或子查询)
PRIMARY	最外层的SELECT
UNION	UNION中第二个或之后的SELECT语句
DEPENDENT UNION	UNION中第二个或之后的SELECT语句取决于外面的查询
UNION RESULT	UNION的结果
SUBQUERY	子查询中的第一个SELECT
DEPENDENT SUBQUERY	子查询中的第一个SELECT，取决于外面的查询
DERIVED	衍生表(FROM子句中的子查询)
MATERIALIZED	物化子查询
UNCACHEABLE SUBQUERY	结果集无法缓存的子查询，必须重新评估外部查询的每一行
UNCACHEABLE UNION	UNION中第二个或之后的SELECT，属于无法缓存的子查询

知乎 @jsava

table

显示这一行是关于哪张表的。

type：

这是**最重要的字段之一**，显示查询使用了何种类型。从最好到最差的连接类型依次为：

> system，const，eq_ref，ref，fulltext，ref_or_null，index_merge，unique_subquery，index_subquery，range，index，ALL

ref

对于来自前面表的每一行，在此表的索引中可以匹配到多行。若联接只用到索引的最左前缀或索引不是主键或唯一索引时，使用ref类型（也就是说，此联接能够匹配多行记录）。

index_merge

表示查询使用了两个以上的索引

fulltext

使用全文索引的时候是这个类型。要注意，**全文索引的优先级很高**，若全文索引和普通索引同时存在时，mysql不管代价，优先选择使用全文索引

range

索引范围查询，常见于使用 =, <>, >, >=, <, <=, IS NULL, <=>, BETWEEN, IN()或者like等运算符的查询中。

index

索引全表扫描，把索引从头到尾扫一遍。这里包含两种情况： 一种是查询使用了覆盖索引，那么它只需要扫描索引就可以获得数据，这个效率要比全表扫描要快，因为索引通常比数据表小，而且还能避免二次查询。在extra中显示Using index，反之，如果在索引上进行全表扫描，没有Using index的提示。

all

全表扫描，性能最差。

possible_keys

对某表进行单表查询时可能用到的索引

key

经过查询优化器计算不同索引的成本，最终选择成本最低的索引

rows

- 如果使用全表扫描，那么rows就代表需要扫描的行数
- 如果使用索引，那么rows就代表预计扫描的行数

> **type、possible_keys、key**

七、相关语句、命令、参数详解

如何查询最后一行记录?

```
select * from table_name order by id desc limit 1;
```

如何随机获取一条记录?

```
SELECT * FROM table_name ORDER BY rand() LIMIT 1;
```

查看当前表有哪些索引?

```
show index from table_name;
```

创建索引

建表时创建

```
CREATE TABLE mytable(
    ID INT NOT NULL,
    username VARCHAR(16) NOT NULL,
    INDEX [indexName] (username(length))
);
```

建表后创建

```
ALTER TABLE my_table ADD [UNIQUE] INDEX index_name(column_name);
或者
CREATE INDEX index_name ON my_table(column_name);
```

注意：

1、索引需要占用**磁盘空间**，因此在创建索引时要考虑到磁盘空间是否足够

2、创建索引时需要**对表加锁**，因此实际操作中需要在业务空闲期间进行

删除索引

```
DROP INDEX my_index ON tablename;
或者
ALTER TABLE table_name DROP INDEX index_name;
```

查看表中的索引

```
SHOW INDEX FROM tablename
```

查看查询语句使用索引的情况

```
//explain 加查询语句
explain SELECT * FROM table_name WHERE column_1='123';
```

Q&A

1、什么是索引?

索引是一种数据结构,可以帮助我们快速的进行数据的查找。

> **数据结构**

2、索引的好处

1.提高数据检索速度，磁盘预读增大命中率，降低数据库IO成本。就是通过缩小表中需要查询的记录的数目从而加快搜索速度。据索引分组和排序，可以加快分组和排序。

2.降低数据排序的成本，降低CPU消耗。之所以查的快，是因为先将数据排好序了。

> **检索速度、IO成本、CPU消耗**

3、索引坏处

1.占用存储空间：索引实际上是一张表，记录了主键和索引字段，一般以索引文件的形式存储在磁盘上。一般来说，索引表占用的空间的数据表的1.5倍；

2.降低更新（删除，添加，修改）表的速度：表的数据发生了变化，对应的索引也需要一起变更。

> **存储空间、更新索引**

4、索引的类型

索引都是实现在存储引擎层的，主要有6中类型

1.普通索引：最基本的索引，没有任何约束

ALTER TABLE 'table_name' ADD **INDEX** ('column')

2.唯一索引：与普通索引类似，但具有唯一性约束

ALTER TABLE 'table_name' ADD **UNIQUE**('column')

3.主键索引：特殊的唯一索引，不允许有空值

ALTER TABLE 'table_name' ADD **PRIMARY KEY**('column')

4.复合索引：将多个组合在一起创建索引，可以覆盖多个列

ALTER TABLE 'table_name' ADD INDEX('colimn1','column2')

5.外键索引：只有InnoDB类型的表才可以使用外建索引，保证数据的一致性，完整性和实现级联操作

6.全文索引：MySQL自带的全文索引只能用于InnoDB,MyISAM，并且只能对英文进行全文检索，一般使用全文索引引擎

> **普唯主复**

5、索引是个什么样的数据结构呢?

索引的数据结构和具体存储引擎的实现有关, 在MySQL中使用较多的索引有Hash索引,B+树索引等,**而我们经常使用的InnoDB存储引擎的默认索引实现为:B+树索引.**

6、什么时候需要给你的表里的字段加索引

- 频繁作为查询条件的字段需要建立索引
- 作为排序和分组的列要建立索引;
- 高并发条件下倾向组合索引;

> **频繁、排序**

7、什么时候不要使用索引?

- 唯一性差（大量重复）的、以及频繁更新的字段不适合建立索引
- 如果需要建立联合索引的话,还需要考虑联合索引中的顺序，根据最左前缀，需要把识别度最高的即最经常用于检索的字段放在最左边
- 只创建必要的索引，避免不必要的开销

唯一性差的，表示重复的行数据很多，即使加了索引，检索出来的数据也会很多，不利于我们快速定位数据；而字段的索引也是需要维护的，进行更新操作的同时，需要调整索引树，会有其他的性能开销。

MySQL可以使用多个字段同时建立一个索引,叫做联合索引.联合索引最左前缀原理，比如联合索引abc，B+ 树检索时，首先根据 a 进行排序，在得到的结果中再根据b进行排序，以此类推。所以一般都是将识别度最高的字段放最左边，以便 b+ 树在最短时间内帮我们进行最高质量的排序。在联合索引中,如果想要命中索引,需要按照建立索引时的字段顺序挨个使用,否则无法命中索引.

> **唯一性差、经常修改**

8、Hash索引和B+树有什么区别或者说优劣呢?

哈希表就是通过哈希函数将一个关键字映射成一个固定的值。当我们进行查询时，就会将 SQL 语句中待查询字段的值，通过哈希函数映射成一个物理数据的地址，然后根据这个地址去数据文件中拿出数据。

但这种方式存在数据碰撞的问题，也就是哈希函数可能对不同的 key 会计算出同一个结果，需要通过链表来解决冲突问题。

哈希算法时间复杂度为 O（1），进行等值查询时效率很高，但不利于范围查询。因为 hash 结构由于其 hash 函数的不确定性，映射出来的地址分布散列，并不能形成物理与逻辑对应的顺序关系。进行范围查询时，会一次把所有数据找出来加载到内存，然后再在内存里筛选筛选目标范围内的数据。 对于区间查询是无法直接通过索引查询的，就需要全表扫描。

而 b+ 树叶子节点之间通过指针关联起来，使得相邻区间的访问效率很高，很适合范围查询。其次是 b+ 树天然有序，可以基于二分法很快定位到数据。而且 b+ 树的叶子节点中，如果是聚集索引，就已经存储了索引对应的数据了，如果使用了索引覆盖，那么直接就可以通过索引，在叶子节点中获得数据，而不用再回表查询，如果这个索引树是主键索引的话，直接就可以在叶子界面中拿到完整的一行数据。

其他的区别还有：

- hash索引不支持使用索引进行排序，原理同上，就是因为它的结构不保证顺序。
- hash索引不支持模糊查询以及多列索引的最左前缀匹配原则。

> **数据碰撞、等值范围查询、聚集索引、索引覆盖、排序和模糊查询**

9、上面提到了B+树在满足聚簇索引和覆盖索引的时候不需要回表查询数据,什么是聚簇索引?

在B+树的索引中,叶子节点可能存储了当前的key值,也可能存储了当前的key值以及整行的数据,这就是聚簇索引和非聚簇索引. 在InnoDB中,只有主键索引是聚簇索引,如果没有主键,则挑选一个唯一键建立聚簇索引.如果没有唯一键,则隐式的生成一个键来建立聚簇索引。

当查询使用聚簇索引时,在对应的叶子节点,可以获取到整行数据,因此不用再次进行回表查询。

10、非聚簇索引一定会回表查询吗?

不一定,这涉及到查询语句所要求的字段是否全部命中了索引,有没有被索引覆盖，如果全部命中了索引,那么就不必再进行回表查询.

举个简单的例子,假设我们在员工表的年龄上建立了索引,那么当进行`select age from employee where age < 20`的查询时,在索引的叶子节点上,已经包含了age信息,不会再次进行回表查询.

> **索引覆盖**

11、创建的索引有没有被使用到?或者说怎么才可以知道这条语句运行很慢的原因?

MySQL提供了explain命令来查看语句的执行计划,MySQL在执行某个语句之前,会将该语句过一遍查询优化器,进行语句执行性能开销的分析，然后产生一个执行计划,其中包含了许多信息. 可以通过其中和索引有关的信息来分析是否命中了索引,例如possilbe_key,key,key_len等字段,分别说明了此语句可能会使用的索引,实际使用的索引以及使用的索引长度.

一条SQL语句的查询，可以有不同的执行方案，至于最终选择哪种方案，需要通过优化器进行选择，选择执行成本最低的方案。 在一条单表查询语句真正执行之前，MySQL的查询优化器会找出执行该语句所有可能使用的方案，对比之后找出成本最低的方案。这个成本最低的方案就是所谓的执行计划。 优化过程大致如下： 1、根据搜索条件，找出所有可能使用的索引 2、计算全表扫描的代价 3、计算使用不同索引执行查询的代价 4、对比各种执行方案的代价，找出成本最低的那一个

> **explain、possilbe_key、key、**
>
> **列出索引、计算代价**

12、那么在哪些情况下会发生针对该列创建了索引但是在查询的时候并没有使用呢?(全表扫描)

- 使用不等于查询,使用!=或<>操作符， 优化器无法通过索引来确定要命中的行数，因此需要搜索该表的所有行
- 列参与了数学运算或者用到了函数
- 在模糊查询时，使用 like 进行模糊匹配时，最左边用到了通配符，类似于'%aaa'.
- 当mysql分析全表扫描比使用索引快的时候不使用索引.（表比较小）
- 当使用联合索引但语句不符合最左前缀原则,也无法使用索引.

> **不等于查询、列运算和函数、模糊查询通配符、优化器**

13、同时有多个事务在进行会怎么样呢?

多事务的并发进行一般会造成以下几个问题:根据不同隔离级别

- 脏读: A事务读取到了B事务未提交的内容，造成一个事务中相同的检索会出现不一致的情况
- 不可重复读: 当设置A事务只能读取B事务已经提交的部分,会造成在A事务内的两次查询,结果竟然不一样,因为在此期间B事务进行了提交操作.
- 幻读: A事务读取了一个范围的内容,而同时B事务在此期间插入了一条数据.造成"幻觉".

解决方法：四种隔离机制

> **脏读、不可重复读、幻读**

14、Innodb使用的是哪种隔离级别呢?

InnoDB默认使用的是可重复读隔离级别.

15、为什么要尽量设定一个主键?

因为在辅助索引中，即对非主键建立了索引，那么如果查询时无法做到索引覆盖，那么在这颗非主键索引 b+ 树中，就只能通过叶子节点找到的主键，来进行回表，即通过主键索引树来找到最终的那行数据。所以如果没有主键的话，就办不到。

而且，比如在默认的引擎 innodb 中，即使我们没有主键和唯一键，系统也会帮我们自动生成一个，但有时候系统生成的主键不是很合理，会导致一些问题，这个主键也是隐式的，不可控，还不如我们自己创建一个，也比较方便跟踪问题。

16、主键使用自增ID还是UUID?

推荐使用自增ID,不要使用UUID。因为 B+ 树天然有序，已经根据索引排好顺序了，如果来一个新的自增的索引，就会自动加入到最后一个索引节点中，不需要怎么调整树的结构。但如果不是自增的索引主键值的话，就会引起树结构的调整，会多很多不必要的性能和空间的开销。

17、字段为什么要求定义为not null?

null值会占用更多的字节,不利于用索引进行排序，且会在程序中造成很多与预期不符的情况.

18、InnoDB和MyISAM有什么区别?

- InnoDB是聚集索引、MyISAM是非聚集索引
- InnoDB支持事物，而MyISAM不支持事物
- InnoDB支持行级锁，而MyISAM支持表级锁
- InnoDB支持MVCC, 而MyISAM不支持
- InnoDB支持外键，而MyISAM不支持
- InnoDB不支持全文索引（5.6后支持），而MyISAM支持。

19、MySQL中的varchar和char有什么区别

char是一个定长字段,假如申请了`char(10)`的空间,那么无论实际存储多少内容.该字段都占用10个字符,而varchar是变长的,也就是说申请的只是最大长度,占用的空间为实际字符长度+1,最后一个字符存储使用了多长的空间.

在检索效率上来讲,char > varchar,因此在使用中,如果确定某个字段的值的长度,可以使用char,否则应该尽量使用varchar.例如存储用户MD5加密后的密码,则应该使用char.

20、varchar(10)和int(10)代表什么含义?

varchar的10代表了申请的空间长度,也是可以存储的数据的最大长度,而int的10只是代表了展示的长度,不足10位以0填充.也就是说,int(1)和int(10)所能存储的数字大小以及占用的空间都是相同的,只是在展示时按照长度展示.

21、超大分页怎么处理?

把热点数据加载到 redis 中 ， 命中则返回即可。

sql 优化：建立在已排好序的集合

类似于`select * from table where age > 20 limit 1000000,10`

可以修改为`select * from table where id in (select id from table where age > 20 limit 1000000,10)`

通过建立一个主键，最好是自增主键id，然后利用索引排序将结果集先排好序之后，再进行limit 的话就会沿着已经排好序的结果集来进行 limit ， 这样效率高很多。

> **索引排序、索引覆盖**

22、超键、候选键、主键、外键分别是什么?

- 1、超键：在关系中能唯一标识元组的属性集称为关系模式的超键。一个属性可以为作为一个超键，多个属性组合在一起也可以作为一个超键。超键包含候选键和主键。
- 2、候选键：是最小超键，即没有冗余元素的超键。
- 3、主键： mysql数据库主键，指的是一个列或多列的组合，其值可以唯一定位到某一行，通过它可强制表的实体完整性。主键主要是用于其他表的外键关联，以及本记录的修改与删除。
- 4、外键：在一个表中存在的另一个表的主键称此表的外键。

23、数据库架构调优

【一般步骤】

先集群：主从，且读写分离。
分表：数据量过大导致查询慢，锁和事物等冲突。一般先垂直分，再考虑水平分。
分库：机器的读写压力大，并发量数大，进一步分担单库压力。
分库分表后：降低磁盘使用率，单表数据量少，并发冲突低。

【读写分离】

某个表或者某个行锁住了，其他读不能并发，为了读读可以高并发， 进行读写分离。

【分表】

垂直分表：是指按功能模块拆分，以解决表与表之间的io竞争。比如分为订单库、商品库、用户库...这种方式多个数据库之间的表结构不同。

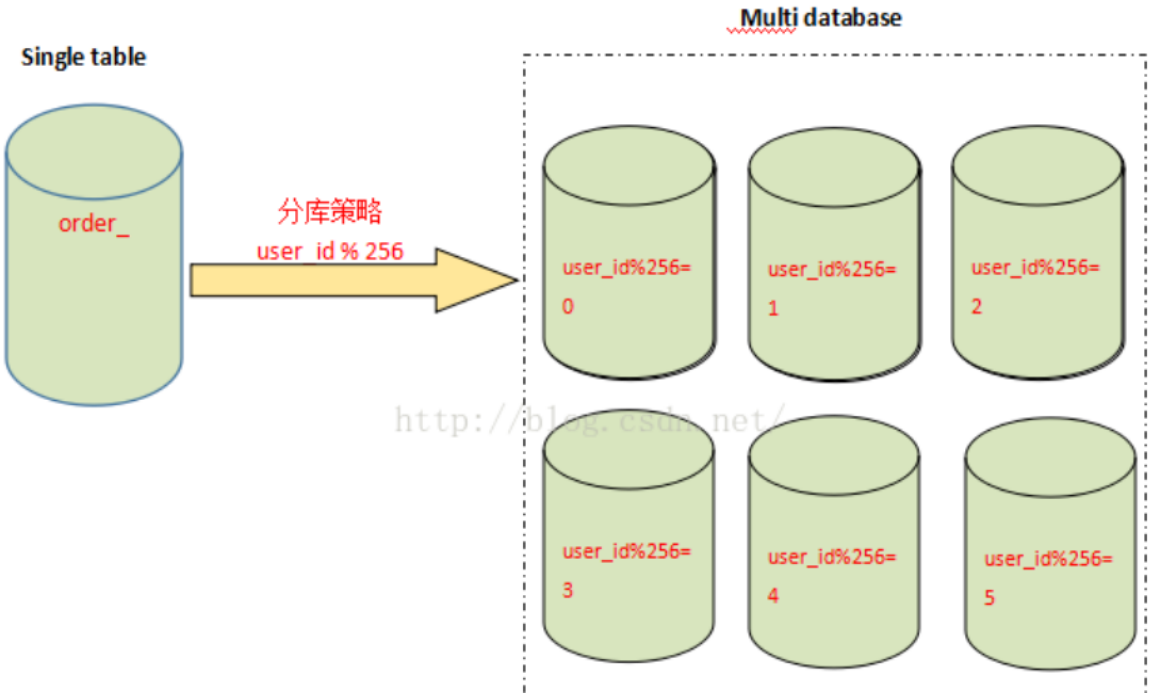
水平分表： 将同一个表的数据进行分块保存到不同的数据库中，来解决单表中数据量增长出现的压力。这些数据库中的表结构完全相同。

【分库】

一个主库通过路由规则拆分为多个主库，降低每个库的写压力，提高写的性能。

分表能够解决单表数据量过大带来的查询效率下降的问题，但是，却无法给数据库的并发处理能力带来质的提升。面对高并发的读写访问，当数据库master服务器无法承载写操作压力时，不管如何扩展slave服务器，此时都没有意义了。因此，我们必须换一种思路，对数据库进行拆分，从而提高数据库写入能力，这就是所谓的分库， 与分表策略相似，分库可以采用通过一个关键字取模的方式，来对数据访问进行路由。

方案
mycat：中间件，代码无需修改。通过改写与转发sql，然后汇总结果后返回给客户端。
shading-jdbc：jar包，代码需要修改。client层方案的优点在于不用部署，运维成本低，不需要代理层的二次转发请求，性能很高，升级的话需要修改系统。
优缺点对比：mycat需要单独部署与运维。sharding-jdbc,需要修改代码。



【其他】

1）解决慢查询语句，排查慢的原因，进行优化；以及优化大的事务，将大的事务拆分成多个小事物进行提交；

2）然后再进行架构的优化，进行读写分离，部署适当多的从数据库（3-5个），主数据库负责写，从数据库负责读，分担每个节点的负载压力。进行分库，把一个主库通过路由规则拆分成多个主库，降低每个主库的并发写压力。如果从数据库的读压力大，还可以考虑分表。节点增加缓存层等。

3）升级设备硬件性能，可以用各种命令排查下性能的瓶颈是出在哪里，先解决问题，如果没问题了确定是硬件本身性能太差的话，如果是内存的话就增加内存，如果是磁盘的话就升级SSD，网卡的话看能否增加带宽等。

4）设置数据库的日志同步参数。可以把 binlog 日志的同步参数设置为0，表示数据的异步刷盘，定时刷盘等；在 innodb 引擎中也有参数可以设置为0，表示每秒把 redo log buffer 中的数据同步刷盘。但这种具体还是要看业务场景，如果是对数据安全和一致性比较敏感的业务领域不可以考虑，但是在高并发的场景下对吞吐量有要求的这种就合适了。

24、什么是存储过程？有哪些优缺点？

存储过程是一些预编译的SQL语句 ，不用再经过 sql 执行过程中的语法分析的过程，提前编译好，在需要的时候调用，提高了执行效率。

> **预编译、语法分析、直接调用**

25、数据库三范式

第一范式（1NF）：字段不可分；

原子性。 字段不可再分,否则就不是关系数据库;

表：字段1， 字段2（字段2.1，字段2.2）， 字段3

第二范式（2NF）：有主键，非主键字段依赖主键；

唯一性 。一个表只说明一个事物；

表：学号， 姓名， 年龄， 课程名称， 成绩， 学分

这个表明显说明了两个事务：学生信息， 课程信息。数据冗余：每条记录都含有相同信息；

第三范式（3NF）：非主键字段不能相互依赖。

每列都与主键有直接关系，不能存在传递依赖。相互依赖则显得关系不够紧密和紧凑。

表：学号， 姓名， 年龄， 所在学院， 学院联系电话

存在依赖传递:：(学号) → (所在学院) → (学院联系电话)

数据库三范式本质要求就是要求表中的数据的凝聚力、内聚力强，主键与非主键形成强相关的关系，要表达的事务更加具体。便于表的维护和管理（增删改查）。

26、索引下推

MySQL 5.6引入了索引下推优化，默认开启，使用SET optimizer_switch = ‘index_condition_pushdown=off’;可以将其关闭。官方文档中给的例子和解释如下： people表中（zipcode，lastname，firstname）构成一个索引
SELECT * FROM people WHERE zipcode=‘95054’ AND lastname LIKE ‘%etrunia%’ AND address LIKE ‘%Main Street%’;
如果没有使用索引下推技术，则MySQL会通过zipcode='95054’从存储引擎中查询对应的数据，返回到MySQL服务端，然后MySQL服务端基于lastname LIKE '%etrunia%'和address LIKE '%Main Street%'来判断数据是否符合条件。 如果使用了索引下推技术，则MYSQL首先会返回符合zipcode='95054’的索引，然后根据lastname LIKE '%etrunia%'和address LIKE '%Main Street%'来判断索引是否符合条件。如果符合条件，则根据该索引来定位对应的数据，如果不符合，则直接reject掉。有了索引下推优化，可以在有like条件查询的情况下，减少回表次数。

> **返回索引代替返回数据、like 查询减少回表**

27、为什么 mongoDB 用b树而 mysql 用b+树

主要是因为b树和b+树的数据结构的差异引起的。b树的每个节点存储索引键的值和数据，而b+树的非叶子节点存储索引和指针，所以当进行单节点的查找时，可能b树在第一层就已经找到并且返回数据了，而b+树则需要一直查找到叶子节点，去取数据，或者回表。

所以 mongodb 是文档型数据库，一个节点一个文档，理论上，单个节点的查找，B树性能可能要高于B+

28、如何选择字符集?

建议在能够完全满足应用的前提下,尽量使用小的字符集。因为更小的字符集意味着能够节省空间、 减少网络传输字节数,同时由于存储空间的较小间接的提高了系统的性能。

有很多字符集可以保存汉字，比如 utf8、gb2312、gbk、latin1 等等，但是常用的是 gb2312 和 gbk。因为 gb2312 字库比 gbk 字库小，有些偏僻字(例如:洺)不能保存，因此 在选择字符集的时候一定要权衡这些偏僻字在应用出现的几率以及造成的影响， 不能做出肯 定答复的话最好选用 gbk。

29、MySQL自增id不连续问题?

- 唯一键冲突
- 事务回滚
- 批量申请自增id的策略

30、NULL和空串判断?

NULL值是没有值,，它不是空串。如果指定''(两个单引号，其间没有字符)，这在NOT NULL列中是允许的。空串是一个有效的值，它不是无值。

判断NULL需要用 IS NULL 或者 IS NOT NULL。

31、死锁?

是指两个或两个以上进程执行过程中，因竞争共享资源造成的相互等待现象。

32、如何处理死锁?

- 设置超时时间。超时后自动释放。
- 发起死锁检测，主动回滚其中一条事务，让其他事务继续执行。

33、Mysql删除表的几种方式？区别?

1.delete : 仅删除表数据，支持条件过滤，支持回滚。记录日志。因此比较慢。

```
delete from table_name;
```

2.truncate: 仅删除所有数据，不支持条件过滤，不支持回滚。不记录日志，效率高于delete。

```
truncate table table_name;
```

3.drop:删除表数据同时删除表结构。将表所占的空间都释放掉。删除效率最高。

```
drop table table_name;
```

> **表数据、条件过滤、回滚、日志、效率**

34、什么是视图?

【概述】

视图是一种虚拟表，不是实际存在于数据库中的，它是由定义和结果集两部分组成的，首先我们定义一个视图，定义的内容就是 sql 语句，然后交给数据库保管，在我们调用视图的时候，就会自动执行预设的 sql 语句，就会返回一个结果集。所以通俗的说，视图就是一个表或多个表的结果集。本质是对基表的操作，会影响到基表。

> **create view view2 as select * from view1 group by view1.cat_id;**

【应用】

1）一个是使操作变得简单，比如我们有时需要写复杂的语句来查询多张表的数据，那么使用视图就可以提前把这些多个表所经常关联查询的数据的查询 sql 先存起来，然后使用的时候这个视图就是这些多个表的一个联合子集，就可以直接在这个结果集中操作，而不用再写复杂的语句。同时也避免使用 join 等关联查询的开销，提升性能。

2）可以屏蔽隐私字段，有些表的某些字段时比较隐私的，需要保密，而其他的字段则可以被访问，此时就可以为这些能够提供查询的字段单独建立视图，从而实现对用户屏蔽隐私字段的功能。

35、什么是临时表?

【概述】

临时表是建立在数据库临时文件夹中的表，但不在数据库中，生命周期是一个连接的会话，如果会话结束，临时表就被删除。

【应用】

1）在短期内有很多增删改的操作，然后这些操作产生的数据只是中间数据，并不会存入数据库时，就可以用临时表，用完就删。比如实现一个购物车列表时，商品的插入和删除，数量的更新等操作，在结算完金钱后这些数据就会被删掉。

2）用于缓存多张关联表的结果，在有多表关联操作时，可以先把相关的部分表关联成临时表，降低表的关联数量，提高速度和效率。

3）比如在导数据时，只想导出部分数据，就可以把数据查询到临时表再导出，导出后再删除临时表。

【对比视图】

1）视图是每次使用时都生成，所以数据比较新，但是每次查询都要建立视图表结构，比较花时间。临时表只需要创建一次，节省了时间，但如果数据源改变了但没有更新临时表，就会有数据不一致的问题。

2）视图主要为了简化复杂的多表查询语句，虚拟表，效率一般；临时表主要存储中间结果集，效率高。

36、MySql最多创建多少列索引?

16

37、count(*)在不同引擎的实现方式?

MyISAM :把一个表的总行数存在了磁盘上，执行 count(*) 的时候会直接返回这个数，效率很高。

InnoDB : 比较麻烦，它执行 count(*) 的时候，需要把数据一行一行地从引擎里面读出来，然后累积计数。

38、乐观锁和悲观锁

乐观锁

【概述】

乐观锁就是假设大部分情况下没有冲突，不用加锁也能执行成功而保证线程安全和数据一致性，那么就省下了大部分时间都去加锁的开销，整体上只需要付出小部分占用 CPU 来自旋尝试的开销即可。

【应用】

1）读多写少的场景，因为读不会产生冲突，而少量的写才需要通过不停自旋尝试。

2）同步代码块短的场景，因为同步代码块执行很快，持有锁的线程可能很快就释放了，所以当前要取锁的线程只要付出轻微的自旋时间就可以立马获得锁，而不用通过状态切换的重量级锁。

3）上述两种情况下的高并发场景，高并发场景下加锁的开销较大，可以考虑使用乐观锁。

【弊端】

不适合高并发下写多读少，而且同步逻辑过长，因为这会在冲突比较大的时候增加了自旋等待的时间，最后还需要通过同步阻塞的方式，相当于不能快速失败而且还要付出锁的开销，反而得不偿失。

【总结】

这里的高并发环境，还需要具体分析是高并发的写比较多，还是高并发的读比较多。

悲观锁

【概述】

悲观锁就是认为每次操作数据都会有冲突，所以为了数据安全就统一都加锁了，让请求顺序串行地操作数据。

【应用】

如果读写都有，写的数量不多，那么用乐观锁。如果写远多读少的场景，因为写多的话意味着冲突的概率就大，通过自旋等待的次数会变多，同时如果是同步代码块较大时，单次自旋等待的时间就长，造成整体等待的周期接近甚至超过悲观锁的开销，在超过自旋等待次数后，就会进入悲观锁，相当于多了自旋的开销以及悲观锁的开销，那不如一开始就用悲观锁。所以它适合于写比较多的场景，因为这意味着使用悲观锁的价值会比较高。

39、InnoDB 行锁实现原理

InnoDB行锁是通过给索引上的索引项加锁来实现的，InnoDB这种行锁实现特点意味着：只有通过索引条件检索数据，InnoDB才使用行级锁，否则，InnoDB将使用表锁！

InnoDB是基于索引来完成行锁

例: select * from tab_with_index where id = 1 for update;

for update 可以根据条件来完成行锁锁定,并且 id 是有索引键的列,

如果 id 不是索引键那么InnoDB将完成表锁,,并发将无从谈起

> **innodb 的主键索引b+树的叶子节点已经存储了对应的一行数据，所以直接通过索引找到某行，再对该行加锁，就是其原理。**

40、如何查看主从一致性

pt-table-checksum是查看主从一致性的工具

```
[root@master ~]# pt-table-checksum --nocheck-replication-filters --no-check-binlog-format --replicate=cnt.checksums --databases=cent h=127.
0.0.1,u=user,p=123456,P=3306
            TS ERRORS  DIFFS     ROWS  CHUNKS SKIPPED    TIME TABLE
06-15T10:24:20      0      0      910       1       0   0.267 cent.ldb_alipay_order
06-15T10:24:21      0      0     1557       1       0   0.283 cent.ldb_charge_history
06-15T10:24:21      0      0     2555       1       0   0.262 cent.ldb_charge_history_detail
06-15T10:24:21      0      0     1488       1       0   0.268 cent.ldb_device
06-15T10:24:21      0      0        7       1       0   0.007 cent.ldb_device_charge
06-15T10:24:21      0      0       34       1       0   0.007 cent.ldb_device_failure
06-15T10:24:21      0      0        0       1       0   0.007 cent.ldb_device_pwd
06-15T10:24:21      0      0        4       1       0   0.007 cent.ldb_failure_info
```

TS： 完成检查的时间
ERRORS： 检查时候发生错误和警告的数量
DIFFS： 0表示一致，1表示不一致。当指定–no-replicate-check时，会一直为0，当指定–replicate-check-only会显示不同的信息
ROWS： 表的行数
CHUNKS： 被划分到表中的块的数目
SKIPPED： 由于错误或警告或过大，则跳过块的数目
TIME： 执行的时间
TABLE： 被检查的表名

> **两个参数：DIFFS 是否一致、table 表名**

41、mysql 备份计划

有增量备份也有全量备份，全量备份是1天一次备份，隔天34点。

42、MySQL的复制原理以及流程

基本原理流程，3个线程以及之间的关联；

主：binlog线程——记录下所有改变了数据库数据的语句，放进master上的binlog中；

从：io线程——在使用start slave 之后，负责从master上拉取 binlog 内容，放进 自己的relay log中；

从：sql执行线程——执行relay log中的语句；

43、innodb引擎的4大特性

预读(read ahead)，插入缓冲（insert buffer)，二次写(double write)，自适应哈希索引(ahi)

44、有多少种日志

事务日志：重做日志redo和回滚日志undo

二进制日志：记录对数据库执行更改的所有操作。

慢查询日志：设置一个阈值，将运行时间超过该值的所有SQL语句都记录到慢查询的日志文件中。

查询日志：记录所有对数据库请求的信息，不论这些请求是否得到了正确的执行。

错误日志：记录出错信息，也记录一些警告信息或者正确的信息。

中继日志：重放日志，中继日志也是二进制日志，用来给slave 库恢复

45、事务是如何通过日志来实现的

对数据的操作首先会存入磁盘的 redo log 日志文件中，但直接存取磁盘速度太慢，所以 mysql 开辟了一个日志缓存，redo日志是先写入log buffer，之后才会被刷新到磁盘的redo日志文件。

46、MySQL数据库cpu飙升到500%的话他怎么处理？

如果已经确定是 mysql 进程的原因的话，那我会

1）先用 show processlist 命令查看线程的运行状况，查看哪些线程的状态已经很久没有变化了，通过状态持续的秒的时间来查看，然后查看 status 属性，看是否数据库正在刷新脏页，或者是某些行被锁住。

2）其次就是看错误日志，以及慢查询日志等，定位到慢查询的具体 sql

3）再用 explain 命令查看慢 Sql 的执行计划，查看字段是否有加索引，或者没有用到索引，或者数据库自己走了全表扫描等，再根据具体情况具体分析来优化 sql。

> **show processlist、日志、explain**

47、500台db，在最快时间之内重启

可以使用批量 ssh 工具 pssh 来对需要重启的机器执行重启命令。 也可以使用 salt（前提是客户端有安装 salt）或者 ansible（ ansible 只需要 ssh 免登通了就行）等多线程工具同时操作多台服务器

> **批量ssh工具来一次性操作多台服务器进行重启**

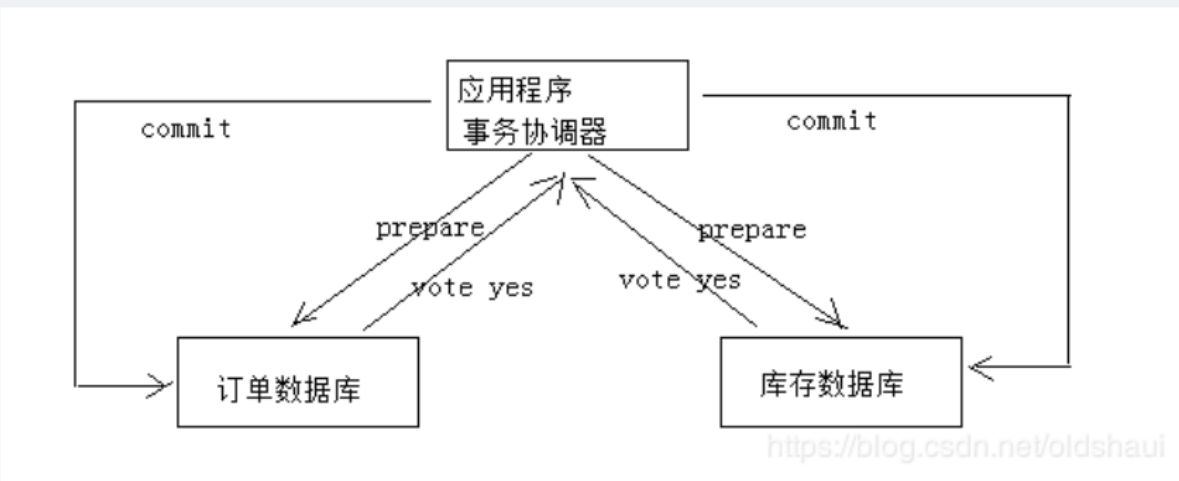
49、分布式事务

1）二阶段提交

比如在订单系统中，有订单和库存两个数据库。

第一阶段：应用程序发送事务消息给两个数据库，消息状态是准备，表示让数据库准备好执行事务。数据库也就是参与者收到后，把事务消息写入本地的事务日志，作为备案，然后向应用程序也就是发起者回复，如果准备好了回复 yes，如果写入事务日志或者其他过程出错了，则回复 no。

第二阶段：如果应用程序收到一个 no，则进行回滚，同时发送消息给参与者，让其删除本地事务日志内容。如果收到 yes，则再发送消息给参与者，消息状态是提交，表示让所有参与者开始提交事务，如果有参与者提交失败则进行回滚，如果提交成功，则分布式事务执行成功。



这种方式的优点是一致性强，但是缺点就是如果采用 http 或者 dubbo rpc 框架的话，多个模块之间或者多个节点之间都需要同步等待这个消息，且有可能因为模块间耦合度较高导致调用链较长，增加了一定的时间和系统资源的开销。

2）基于 RocketMQ 消息队列的事务消息来实现最终一致性

将分布式事务拆分成多个本地事务来执行。比如在转账业务中，有 A 和 B 两个用户，A 向 B 转账100 元，在单机上业务执行的过程就是：锁定 A 账户，锁定 B 账户，检查 A 是否有 100 元，有的话从 A 减去 100，然后 B 增加 100，解锁 B 账户，解锁 A 账户。

但这里对 A 账户的操作和 B 账户的操作是相互独立的，可以拆分成两个子事务分布在两个模块中，比如事务 1 就是锁定账户 A ，检查账户 A 是否有100 元，从 A 减去 100，解锁 A 账户。事务 2 就是锁定账户 B，增加 100 元，解锁账户 B。

然后模块 1 通过 RocketMQ 的事务消息先向 Broker 发送一条消息，状态是准备，表示模块 1 准备好执行事务了，然后 Broker 返回 ok 之后，模块 1 开始执行本地事务 A，锁定账户 A ，检查账户 A 是否有 100 元，从 A 减去 100，解锁 A 账户。执行完后给 Broker 发送消息，状态是提交，表示本地事务已完成，消息状态可以提交，这时 Broker 才会把这条状态为 commit 的消息真正发送到消费者队列中，然后模块 2 去订阅这个模块 1 的主题，去拉取消费这条消息，收到 commit 的通知后，就会执行模块 2 的事务。

模块 2 执行成功后会返回消费成功，那么分布式事务执行成功。如果消费失败，且设置了重试次数，那么就继续重试，一般重试次数内都可以成功的，如果失败了，那么事务就要进行回滚，此时模块 2 也可以发消息给模块 1 令其回滚。如果模块 1 在执行完本地事务后，因为网络抖动等原因没有发送 commit 消息给 Broker 的话，Broker 就会通过模块 1 提供的回调接口去查询模块 1 事务的执行状态，然后模块 1 就会根据其执行的状态回复 Broker 是提交成功还是需要回滚的消息。

这种方式由于基于 RocketMQ 的现有的事务消息，所以编码上比较简单，同时通过这种异步发送的方式，以及消息队列解耦了模块间的关系，提高了吞吐量。

50、主键和唯一索引区别?

本质区别，主键是一种约束，唯一索引是一种索引。

主键不能有空值（非空+唯一），唯一索引可以为空。

主键可以是其他表的外键，唯一索引不可以。

一个表只能有一个主键，唯一索引 可以多个。

主键就是用来唯一查找某行或某些行数据，体现了唯一性。

51、数据库乐观锁实现

即为数据增加一个版本标识，一般是通过为数据库表增加一个数字类型的 “version” 字段来实现。当读取数据时，将version字段的值一同读出，数据每更新一次，对此version值加一。当我们提交更新的时候，判断数据库表对应记录的当前版本信息与第一次取出来的version值进行比对，如果数据库表当前版本号与第一次取出来的version值相等，则予以更新，否则认为是过期数据

总结

【存储结构】

hash表

关键字映射，拿地址取数据

哈希碰撞，链地址法

等值查询O1

范围查询加载所有数据到内存进行筛选

二叉树

左<中<右，方便范围查询，O(lgn)，但会极端不平衡，退化成O(n)

红黑树

左右旋转与变色、O(lgn)，也会不平衡，出现右倾

自平衡二叉树

O(lgn)，可维持平衡，但节点只存一个数据

b树

O（h*logn）、局部性原理预读、节点更多数据、增加缓存命中、降低IO

b+树

O（h*logn）、节点存更多索引、可以覆盖更多数据，树更低、IO更少，叶子节点存数据、区间链表

【存储引擎】

MyISAM 非聚集，叶子存地址

InnoDB 聚集，叶子存数据，辅助索引叶子存主键，需要回表（主键索引树）

【事务】

基础

ACID

redo 记录更新后的数据，用于恢复已提交未持久化的数据，undo 记录更新前的数据，用户更新失败时（出异常）的回滚

set session transaction isolation level 设置会话交流的隔离级别

隔离级别

读未提交（产生脏读）

读已提交（sql视图，不可重复读）

可重复读（事务视图，解决了脏读和不可重复读、解决读操作下的幻读，不能解决修改操作update和delete的幻读，不彻底）

串行化（加更多的读写锁排队执行，性能最差）

MVCC（事务Id、执行版本、绑定）

数据库锁

读写锁、行锁、表锁、间隙锁

【日志数据持久化】

查看缓存：show VARIABLES like 'innodb_log_buffer_size'

工作原理：数据操作 -> log buffer -> 磁盘redo log

刷盘时机：缓存不足、后台定期

binlog：索引文件、日志文件、binlog_cache_size、sync_binlog（刷盘频率）、XA

主从同步：binlog、重放日志、执行重放

binlog格式：语句模式（对数据修改的每条sql都会记录，文件小、一次存储和发送的数据多，节约磁盘和网络IO、准确性差一些），数据行模式（记录哪行数据被修改了，改成什么样子了，文件大、磁盘和网络IO大、准确性强）

增量模式：配置文件打开 log-bin、show variables like '%log_bin%'

【性能排查】

流程

show processlist、刷新脏页、锁数据、

日志（错误日志、慢查询日志）

explain（没有加索引、没有走索引、索引优化）

索引优化

计算全表查询和不同索引的查询成本

show table status like　成本：IO（页面 x 单位成本1）+ CPU成本（表行总数 x 单位成本0.2）

processlist

查看有哪些线程在运行，以及线程对应的语句的执行状态和持续的时间等

host、db、time、status（flushing tables、closing tables、locked、sleeping）

explain

type、possible_keys、key

慢查询日志

slow_query_log、long_query_time、用户及主机、查询时间、锁表时间

参考资料：

深入理解 Mysql 索引底层原理

https://zhuanlan.zhihu.com/p/113917726

3万字总结，Mysql优化之精髓

https://zhuanlan.zhihu.com/p/106061137

把MySQL中的各种锁及其原理都画出来

https://zhuanlan.zhihu.com/p/149228460

老夫整理的1000行MySQL学习笔记传授有缘人

https://zhuanlan.zhihu.com/p/71232689

彻底弄懂MySQL索引

https://zhuanlan.zhihu.com/p/145097959

MySql 三大知识点——索引、锁、事务

https://zhuanlan.zhihu.com/p/59764376

Mysql_为什么是B+树

https://zhuanlan.zhihu.com/p/158205230

【聊聊MySQL】四.MySQL-InnoDB表索引和B+树

https://liweidan.cn/db/mysql/2020/01/17/488/

MySQL之Redo Log
https://zhuanlan.zhihu.com/p/35355751

你还精通MySQL，竟然连bin log、redo log都不知道！
https://zhuanlan.zhihu.com/p/140955634

关于MySQL的重做日志（redo log），回滚日志（undo log）及二进制日志（binlog）
https://www.linuxidc.com/Linux/2018-01/150614.htm

mysql两个重要的日志redolog和binlog
https://www.lagou.com/lgeduarticle/107406.html

通俗易懂-Mysql事务持久化实现之-redolog
https://zhuanlan.zhihu.com/p/93131864