## 实验

二分查找mid = high+low/2 这里会有int范围一处问题,要用

Mid = low + 1/2(high-low)

<https://zhuanlan.zhihu.com/p/142165678>

整体流程

navicat亲测,开两个事务,对同一行forupdate,后开的哪个事务会阻塞在这一行,直到另一个事务commit或者后面这个事务超时错误.

SELECT \* from information\_schema.INNODB\_TRX 我开启一个事务后,执行这一句,确实可以看到有一条记录,如果事务提交了,这条记录也就没了,这条记录里面有很多内容,包括事务的开始时间,事务的状态,这个事务如果被锁阻塞了还会告诉你是被哪个事务阻塞了.

SELECT \* from information\_schema.INNODB\_LOCKS可以查看有那些锁.

## TIPS

锁和事务的关系就是锁可以对你的某些操作特殊的提供一些隔离性的加持,但是事务不是通过锁实现的.

什么叫undolog,其实就是没有被事务操作时候的数据的备份,注意undo的语句.

**缓冲区目前有两个东西是周期性的往硬盘刷的，一个是redolog，注意哦，redolog也是有自己的缓存的，然后再刷到硬盘里成为真正的redolog。还有一个就是刷脏页。**

insert into users\_wx set wx\_tel='12345434566',owner = 'csc', wx\_id = 3;这样insert也是可以的

即使在页中的数据也不是一行行从上到下排列的，也是按照堆的形式排列的

B+树只有子叶节点才存数据,上面的节点都是索引节点,只有索引作用.

主从复制的主库肯定要加一些配置,而且从库也要输一些配置,其中一个配置就是主库的一个账户和密码这样才行.

一个区1M 一个页16KB 一个区有64个页

查询学生5到10名的成绩 从第五个开始,步长是5

SELECT \* FROM student ORDER BY score DESC LIMIT 5,5;

查询排在第4名的成绩. 从第4个开始,步长是5

SELECT \* FROM student ORDER BY score DESC LIMIT 4,1;

注意limit 的第一个参数就是索引是从0开始的.

两次写,东西解决了是刷脏页过程中出了问题,导致刷脏页不完整,而redolog解决的是一个事务的完整性问题,解决的问题是不一样的.

刷新邻接页的意思就是再刷新一个脏页的时候，检查这个脏页所在区还有没有脏页，有的话一起把脏页刷了就行。这样的话可以减少对硬盘的操作

两次写的的意思就是缓冲池里先有个2M大小的doublewritebuffer,然后在磁盘里有个连续2M大小的共享表空间.刷页的时候先把内容刷到共享表空间里,这个过程由于是连续空间基本不会出问题,然后再往真正的数据页刷,如果过程失败了也就是没刷完整,我们可以用共享空间的内容去恢复数据页为完整的数据了.

Explain select的时候查出来的type是all就是没用索引,是index或者ref就是用了索引 而且是如果是索引查询的话 key 也会是你的索引列的!!!!!

慢查询日志不仅可以记录那些查询的比较慢的操作,也可以记录那些命中缓冲池比较低的查询.

自适应hash索引的意思就是如果你这些数据被访问的很频繁,那么我对缓冲池中这些内存数据建一个hash来加快访问效率.

Checkpoint流程就是刷脏页的流程,同时呢,你刷了脏页redolog也就可以删除了

Checkpoint流程,缓冲池肯定不可能把所有的数据都缓冲住,肯定是有个隔一段时间就刷脏页的流程,checkpoint流程就是这个流程,***刷脏页,而且你一旦刷了脏页,在这个节点之前的redolog也就可以删了***,checkpoint流程是masterthread弄的,主线程才是刷脏页的线程.一般都是缓冲池不够了啊,redolog空间不够了才会触发到checkpoint流程.

使用group by某一列的时候会默认根据分组的字段排个序,效率偏低,可以

Select xx from xx group by xx order by null;这样去除排序效率高的.

>=100的效率要比>101效率低,因为>=要做两次全表扫描.

主线程的操作说明了redolog在缓冲区是有一个buffer的,redolog也是 会有从缓冲刷到redolog页的操作的,而且是每秒中就有一次这个操作的,每秒的这个是刷redolog到磁盘而不是删哈,这个频率要很快才行,一秒刷一次都慢,因为你要靠redolog来恢复啊.

这就解释了为什么事务很大提交也很快,因为redolog每秒都会刷到硬盘里面.

主线程还会触发刷脏页,至多刷100个,合并插入缓冲,主线程有多种状态,如果空闲了就会进入到backup状态.

Mysql用的就是域套接字，那么就会有一个mysql.sock文件

Linux每个进程都会有一个pid文件,避免进程重复!!!,类似于锁文件,mysql也会有自己的pid文件.

那么mysql就会有自己的pid文件和自己的域套接字文件.

同时还会有表空间文件(存数据的地方)和表定义文件.

Mysql默认的就是AIO.

后台线程的作用主要是1,接收外面的请求,修改缓冲池中的数据,保证缓冲池中的数据是最新的,2,刷脏页

Master thread刷脏页 IO thread处理外面的read write请求,用的是AIO

Purge thread 清掉没用的undolog.

缓冲池是有两个作用的,一个是就是对于查询操作的时候从缓冲池里查,而不是从物理文件.还有就是对于insert和update操作是先写缓冲池,然后再把脏页刷新到磁盘文件.

Innodb的核心 内存与后台线程

回表的意思,就是通过辅助索引找到了具体数据的地址或是指针还是啥的,反正就是找到了,然后拿着信息去IOT的聚集索引去找真正的数据.

而我们的覆盖索引由于B+树那里就保存了数据是无需回表的.

Where后面的条件列， order by后面的条件列 join语句后面的条件列尽量都有索引比较好 where order by join都是可以触发到索引的。

MVCC多版本控制的意思,也就是非锁定读的意思,也就是如果被读的数据正在背写,那么正常情况下就不能读了,但是在innodb中,会从undolog里读.

B+树的索引的告诉一般都在2-4层,没有很高的那种B+树.

也就是说查找某一个数据顶多2到4次IO.

需要牢记的是,B+树索引本身并不能找到具体的一条记录,而是能找到该记录所在的页.数据库把页载入到内存,然后通过page directory再进行二叉查找.只不过二叉查找的时候复杂度很低,同时在内存中的查找很快,因此通常忽略这部分查找用的时间.

B+树索引找到的是一个页,然后把这个页加载在内存中,然后利用pagedirectory二叉查找.

LRU算法,最近最少使用算法

经过上面的学习,我们已经知道了页是innodb存储引擎管理数据库的最小的磁盘单位.页类型为B-tree Node的页存放的即是表中行的实际数据了!!!!!!!!

上面我们可以知道了,innodb是IOT,那么就是索引和数据在一起,那么数据存储的方式肯定也和B树有关系,现在我们知道了,其实b树中的一个子叶节点就是一个页 嗯!

**那么数据在页中的user\_record中就是一行行从上到下排列的吗,不是的,是按照堆的形式排列的!!!!!!!!!!!**

SHOW ENGINE INNODB STATUS 查看存储引擎的状态

show variables like 'log\_bin'; 查看是否启用了二进制日志

SHOW VARIABLES xxx 查看数据库中所有的参数。

Show VARIABLES xxx这个参数挺厉害的啊

Select \*,\_rowid from users\_user

\_rowid就是主键列的值，innodb存储引擎每个表都会有一个主键的。即使你没有设置主键

Show variables like ‘datadir’

查看mysql的这些日志的保存位置！！！

Update多个字段这样 update xxx set xx=xx,xx=xx,xx=xx where xxx

中间用，隔开。

Mysql是C++写的。

Select VERSION();查看mysql版本号.

show ENGINES 查看支持的数据库引擎

查询的话少查几列的话会稍微快一点,但是查询主要还是靠索引来提速.

在navicat中修改数据要保存一下才会生效.

数据少的时候用索引也不会提升速度。

Navicat的工具还是视图栏可以打开mysql命令行.

Navicat点击格式可以美化sql。

Where (aa=dd or aa=cc) and mm=xx

看到了吗 or由于是多个组合要加个括号而且是这样的,or前后有两个=的.

SELECT \* FROM main\_course where (grade = 8 or grade = 6) and season = 'q'；

多个or要用括号括起来。

多表查询，select xxx from a,b,c,d where xx=xx and xx=xx这样就可以，我用where也可以，按理说应该用on，用where也是可以的。

如果查询数据的时候你想根据一列排重，group by 列名，效率是不是distinct效率高的。

SELECT DISTINCT user\_id,course\_id,order\_no,is\_refund FROm main\_learn

SELECT DISTINCT user\_id FROm main\_learn

Distinc这种后面有几列就是这几列都相同了才会去掉，而且结果里只会有你distnct这几列。

而且如果有distinct的话，全都要再distinct后面，下面这种玩法是不行的。

SELECT work,DISTINCT name FROM `user\_test`

Group by是以一列分组去重，这样的话是可以看到所有数据的。

而且如果你是group by grade的话，前面select展示的东西是可以有grade的，因为是按照grade分组的，比如select grade count（\*） from xxx group by grade是可以这样的。

如果你是group by course\_id的话，那么你在select 后面是可以展示c.grade,c.subjects的，如果关联的话，因为是以course\_id这个课程表的主键group by的。

多表查询，select xxx from a,b,c,d where xx=xx and xx=xx这样就可以，我用where也可以。

如果查询数据的时候你想根据一列排重，group by 列名。

select count(0) c,tel from users\_user group by tel having c > 1

查询出现次数大于2的。

Having这个条件只配合group by使用，having用来操作聚合操作的结果，比如count（xx） max（xx） avg（xx）

比如我们分库分表的话要查询多个表的集合就可以这么用，还有就是这两个表不一定非得列都一样，不是一样也可以查询

SELECT column\_name(s) FROM table\_name1

UNION

SELECT column\_name(s) FROM table\_name2

还有就是，如果select 后面的查询内容一样的话，用union会把相同的内容去重，如果不想去重 union all

多表关联，注意尽量中间的条件用一对一的那种，不要一对多的当条件。3个表的话要注意3个表之间最好都要有条件，按理说要是都是一对一的那么2个表有

SELECT \* from main\_learn l,main\_course c where l.course\_id = c.product\_id and c.season = 'c'

这种是连接查询，查看报了春季课的人。

SELECT \* from main\_learn where course\_id in(SELECT product\_id FROM main\_course where season = 'c')

这种是子查询配合嵌套查询in 查看报了春季课的人。

In（xx） not in（xxx）

据说连接查询效率高，我这里试的是子查询效率高，可能当数据到一定量后就是连接查询效率高了吧。

SELECT \* FROM main\_course ORDER BY grade,season,course\_type,items

Orderby多个的使用。

多个not like也要用or 有or的话一定要把or的那部分用一个括号括起来。

SELECT \* from users\_orders o,main\_course c where o.course\_id\_id = c.product\_id and c.season = 'd' and ( o.order\_no not like 'dier%' or o.order\_no not like '家长%' or o.order\_no not like 'free' or o.order\_no not like 'gongkaike%' or o.order\_no not like 'kecheng%') and o.status = 1

SELECT \* FROM main\_course WHERE (season = 'q' and subjects = 'shuxue' AND course\_type IN (0, 1, 2) ) or (season = 'x' and subjects = 'shuxue' and course\_type IN (0, 1, 2) )

两套条件，满足哪个条件都可以，也可以把这两套条件用or弄起来。

is null 对于null的判断是is null和is not null系列。

Left join 是做这种报了寒假学生有没有续报春季的经典使用，报了寒假的学生left join 报春季的就是以寒假学生打底。

## 注意!!!!!!!!!!!!!!!!!!!!!!

## B+树索引并不能找到一个给定键值的具体行.B+树索引能找到的只是呗查找数据行所在的页,然后数据库把页读到内存再进行查找,也就是mysql每次查找都最少读一个页到内存中!!!!!!!!!!!!!!!

## Mysql架构

最下面是物理文件,这里往上是插入式存储引擎,最上面是连接池组件,左上角是管理的组件,连接池下面是sql接口组件 查询优化器组件 和缓冲池组件.

## Sql函数

mc.course\_edate > NOW()

UPDATE user\_test set name = REPLACE(name,'y',1) 替换把name里的y替换成1

UPDATE user\_test set name = SUBSTRING(name,1,2) where work = 'cxy'

切割

UPDATE user\_test set name = CONCAT(name,work,'lalal') where work = 'cxy'

拼接

update main\_chapter set chapter\_start\_time = ADDDATE(chapter\_start\_time,INTERVAL 7 DAY)

日期增加天数

## Mysql的6大日志

Mysql有这几种日志，重做日志redolog，回滚日志undolog，二进制日志binlog，错误日志errorlog，慢查询日志slowquerylog，一般查询日志generalog，中继日志relaylog 这些日志都是会存储在硬盘文件里。

首先redolog是在磁盘里的。

重做日志redolog，主要是保证事务的持久性和数据的一致性，防止到事务的过程中死机了，程序都死了这样的话也就没办法回滚了，但是通过redolog在mysql再次启动的时候做一下redolog的内容把上次被死机打断的事务执行完成，所以redolog的写入是和事务同步的，事务这边执行了一条，redolog也要记录这条不用再执行了，如果事务正常执行完了，那么这个事务的redolog也就没啥用了可以消失了，如果事务执行过程中死机了那么redolog就会记录了这个事务那些语句执行了那些还没执行，等mysql重新启动的话就根据redolog执行剩余还没执行的操作。

看了下面后我发现了还有另一个场景，就是你这个事务提交到了缓存页里，可能事务也跑完了但是并没有存到磁盘里面，缓存页是在内存里面，死机了就没了，所以在这种缓存页还处于脏页的情况下redolog是不能删除的。

所以说redolog并不是事务提交就删了,而是把缓存页中对应的脏页刷新到磁盘中才会删除redolog.

随时注意mysql的缓存页和刷脏页机制!!!!

回滚日志undolog 保存了事务发生之前的数据的一个版本，可以用于回滚，同时可以提供多版本并发控制下的读（MVCC），也即非锁定读。Undolog可以提供MVCC和非锁定读功能。

二进制日志binlog   
　1，用于复制，在主从复制中，从库利用主库上的binlog进行重播，实现主从同步。  
　2，用于数据库的基于时间点的还原。  
　　逻辑格式的日志，可以简单认为就是执行过的事务中的sql语句。

## HOT和IOT

HOT heap table，堆表，为了查询更加快肯定是会有索引的出现，堆表的数据和索引是分开的，数据放在这块，索引放在那块，数据的存放方式是堆，也就是大顶堆小顶堆啥的，这玩意本质上也就是个数组，只不过有了堆的特性，他查询所有的时候肯定是遍历这个数组，那么肯定和插入顺序不一样，肯定是无序的，而有序的索引又不合数据在一起。

HOT 查询慢，增删快，无序。

IOT 索引组织表，查询快，增删慢，有序。这个是把索引和数据都弄在一起弄成一个节点放在主键的位置，这个节点在哪是根据主键的大小来的，比如你的主键是自增的id，那么索引是红黑树的话，你的节点就根据主键大小依次挂在树上，你遍历的话就遍历这个B树，那么肯定是有序的。而且我们发现如果删除了一条数据，那么

自增的id主键是不能再次使用了，就是因为你的主键的B树这个id的位置已经被占了，如果再利用的话就不无序了，后面插入的就跑前面取了。

为啥被删掉的数据的主键自增id不能重复利用.

无序的原因是因为HOT查询全部的时候遍历的是堆，而IOY遍历的是索引的B树。

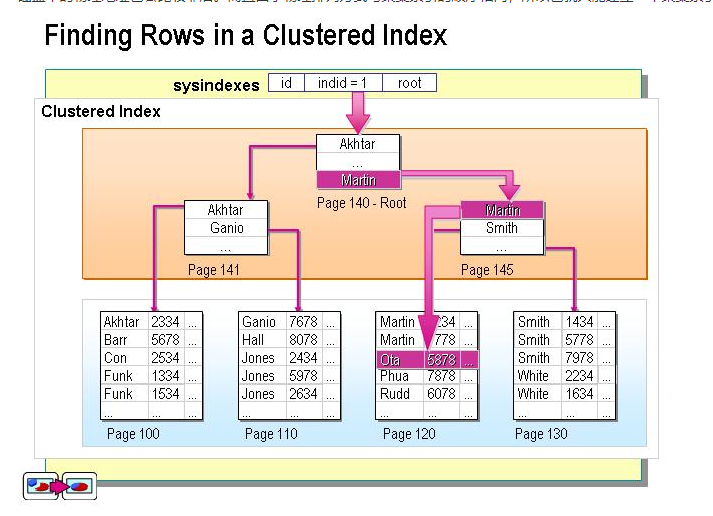
Myisam只支持HOT inoodb只支持IOT oracle这俩都支持

## 聚集索引和非聚集索引

聚集索引常指的是那种id自增的主键索引,而其他比如加在name上或者订单号上的索引就是非聚集索引.

简单来说就是聚集索引的索引顺序和物理的存储顺序是一样的,而非聚集索引的存储顺序是和物理存储的顺序没有关系的.

可以知道,innodb是IOT的,也就是数据和索引是在一起的,也就是B+树的子叶节点包含了索引和数据,可以见下图.



如果你俩的主键id爱着,那么你俩的物理存储位置也是挨着的,存储的前后顺序是和id对应的,因为这个是按照id来存的.可以看到顺序摆放,挨着的都挨个罗列在一个页里.

聚集索引的好处就是由于每一个子叶节点就是一个page里的一行,而且在page里的行都是挨着的.这样的好处就是

你通过索引找到的是全部的行数据,而且如果范围查找的话更快.

聚集索引只能有一个.

非聚集索引的意思就是索引顺序和物理顺序没关系的索引,比如你对name进行了索引.其实这个索引是根据name来的,name的B+树的从上到下的大小排列顺序和物理顺序肯定是没有关系的啊.而且这种索引的子叶节点存的时候数据真实的页和行的地址,所以查到索引后还要二次查询一下通过地址找到真正的数据.但是估计二次查询并不会消耗很大的性能.

后面这个二次查询学名叫做回表

## 数据库的x锁和s锁

在事务里，输入

  SELECT …… FOR UPDATE 这个相当于加了一个x锁，写锁 排斥读和写

  SELECT …… LOCK IN SHARE MODE 这个相当于加了一个s锁，只排斥写不排斥读

当事务提交了这两种锁也就释放了。

## MYSQL技术内幕InooDB

**注意table space segement extent page row这些东西都在磁盘里而不在内存里，内存会有这些东西的缓存。**

### mysql体系结构和存储引擎简介

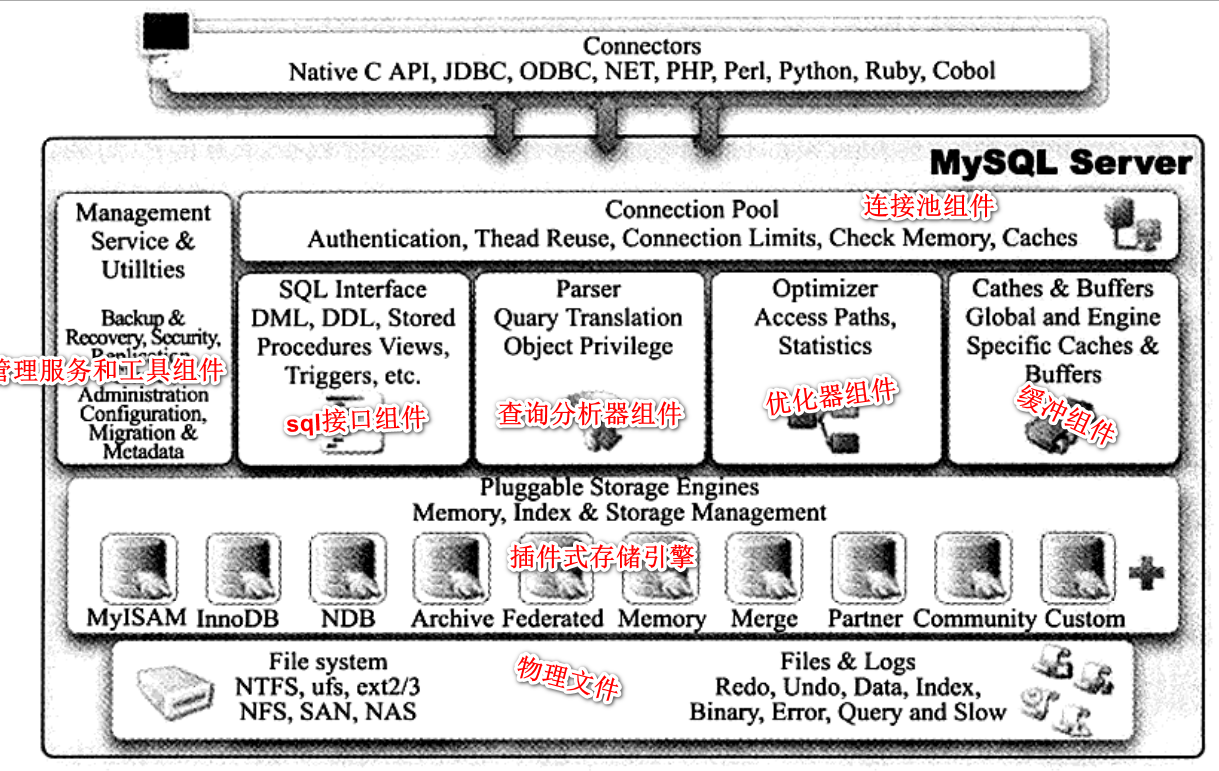
**插入式存储引擎**

Mysql可以在各种操作系统上运行，不同操作系统的线程啊，api啥的区别已经被mysql处理好了。

Mysql是单进程多线程的。oracle的linux版是多进程多线程的，oracle的windows版是单进程多线程的。

数据库：是一个文件的集合，是依照某种数据模型组织起来并存放于二级存储器中的数据集合。

数据库实例：是程序，是位于用户与操作系统之间的一层数据管理软件，用户对数据库的任何操作都是通过数据库实例来完成的。



上图很重要

**也就是同一个库中的表可以自己选择不同的引擎.**

Mysql的插件式的存储引擎架构提供了一系列标准。也就是说，我们可以按照mysql的存储引擎标准开发自己的存储引擎，因为mysql是插件式存储引擎架构嘛，我们就可以把自己按照标准写的存储引擎插进去。

Mysql的存储引擎是基于表的，而不是基于数据库，也就是这个表可以明确自己的存储引擎，而不是整个库确定了哪个就全部用这个存储引擎。

由于mysql数据库独有的插件式体系结构，可以根据不同的应用建立不同的存储引擎的表。各个存储引擎的是有区别的，但是你选择了不同的存储引擎对我们开发来说确是没有任何影响的，存储引擎是透明的。

Mysql是开源的，所以我们可以根据需求修改存储引擎的源码，毕竟开源嘛，随便搞。

Myisam只支持表级锁，用户在操作mysiam表时，select，insert，delete等语句都会锁住表。

Innodb支持事务和行级锁，最大特色就是这个。

#### 1.1 InooDB存储引擎 行级锁和事务

inoodb存储引擎支持事务，他设计的目标主要是面向在线事务处理（OLTP）的应用，特点是行锁设计，支持外键，并支持类似oracle的非锁定读，即默认情况读取操作不会产生锁。从mysql5.5.8版本开始，innodb是默认存储引擎。

#### 1.2 一致性非锁定读和锁定读锁

一致性非锁定读是指InnoDB存储引擎通过多版本控制（multi versioning）的方式来读取当前执行时间数据库中的数据。如果被读的数据行被加了排他锁，在读取这行数据的时候并不会等待锁释放，而是读取该行的一个快照数据。

   之所以称为非锁定读，因为不需要等待被访问行的X锁的释放。快照数据是指改行之前的数据版本，该实现通过undo段来完成。

   非锁定读的方式极大提高了数据库的并发性。在InnoDB存储引擎中，这是默认的读取方式。

非锁定读是默认的也就是可能读到老数据，也就是你在select的时候，那边有人在写，那么这时候你select的时候也就是读的时候读的是以前的，读写同时进行，这和copyonwritearraylist差不多啊。

一致性锁定读不是默认的那么就需要开启了，开启的方法就是，在事务里，输入

  SELECT …… FOR UPDATE 这个相当于加了一个x锁，写锁 排斥读和写

  SELECT …… LOCK IN SHARE MODE 这个相当于加了一个s锁，只排斥写不排斥读

当事务提交了这两种锁也就释放了。

Inoodb使用了MVCC 多版本并发控制来获得高并发性，并且实现了sql标准的四种隔离级别。默认是repeatable read级别。

这个mvcc就是有快照，如果读写冲突了读不阻塞而是读快照里的。

#### 1.3 MyISAM存储引擎 无事务是表锁

MYISAM存储引擎不支持事务，不支持行锁，支持表锁，支持全文索引。mysql5.5之前默认的存储引擎就是MyISAM。

MyISAM春初引擎表由MYD和MYI组成，MYD用来存放数据文件，MYI用来存放索引文件。从mysql5。0开始MyISAM默认支持256TB的表单数据。

#### 1.4 NDB存储引擎 集群的

NDB存储引擎是一个集群存储引擎，由于是集群的可用性更高。NDB的特点是数据全放在内存中，因此主键查找速度极快，而且由于是集群的，你添加了集群节点那么性能就会提高啦。

注意的是NDB的连接操作是在mysql数据库层完成的，而不是存储引擎层完成的，这意味着连接操作需要走很多的节点，势必有巨大的网络开销，如果解决了这个，这种集群的高可用高性能的优势就很大了。

#### 1.5 MEMORY存储引擎

Memeory存储引擎将数据都存放在内存中，一旦数据库重启或者崩溃表中数据都没了，适合那种临时表。

#### 1.6 Archive存储引擎（档案文件）

Archive存储引擎只支持insert和select操作archive存储引擎使用zlib算法将数据进行压缩后存储，压缩比可以达到1：10，非常适合存储归档数据，比如日志信息本身不是事务安全的，主要目标是高速的插入和压缩功能。

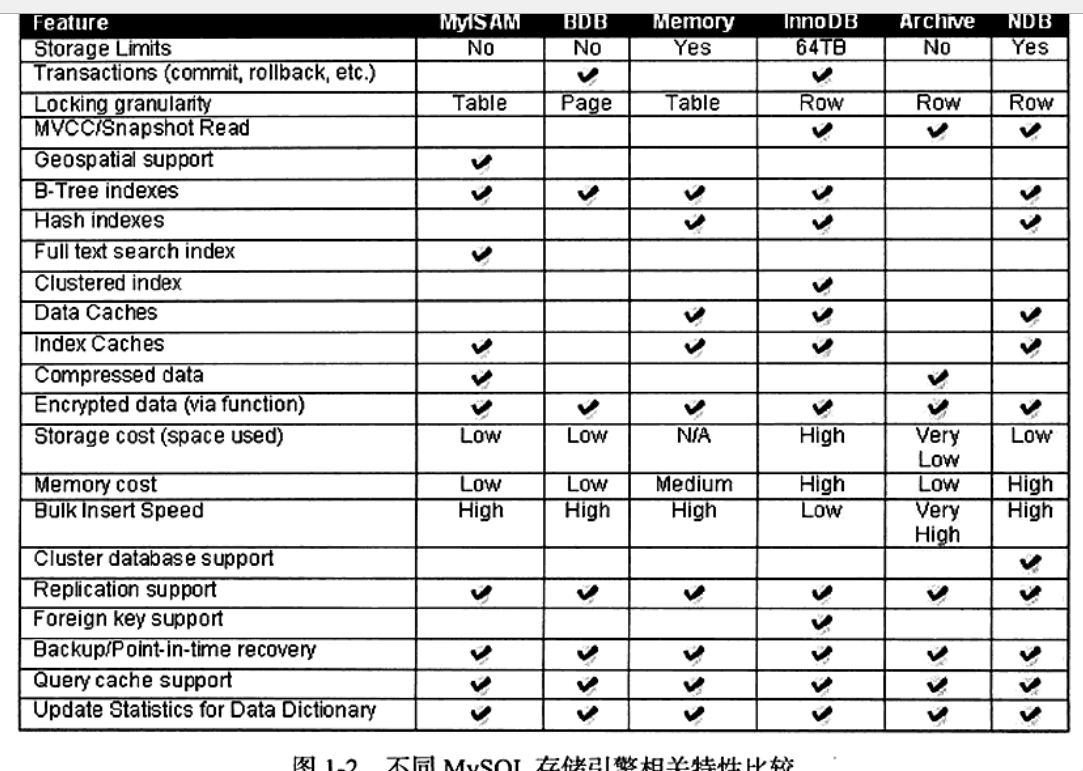
#### 1.7 Maria存储引擎 mysiam二代

新开发的存储引擎，mysiam的二代，行锁设计，MVCC功能，支持事务，也可以选中不支持。

#### 1.8 总结

Mysql数据库支持全文索引，myisam innodb都是支持的。

当表的数据超过1000万时mysql的性能会大幅下降吗？不会的，mysql是数据库，不是文件，随着数据行数的增加，性能当然会有所下降，但是这些下降都是线性的，如果用户选择了正确的引擎，正确的配置多大的数据量mysql也是可以的。



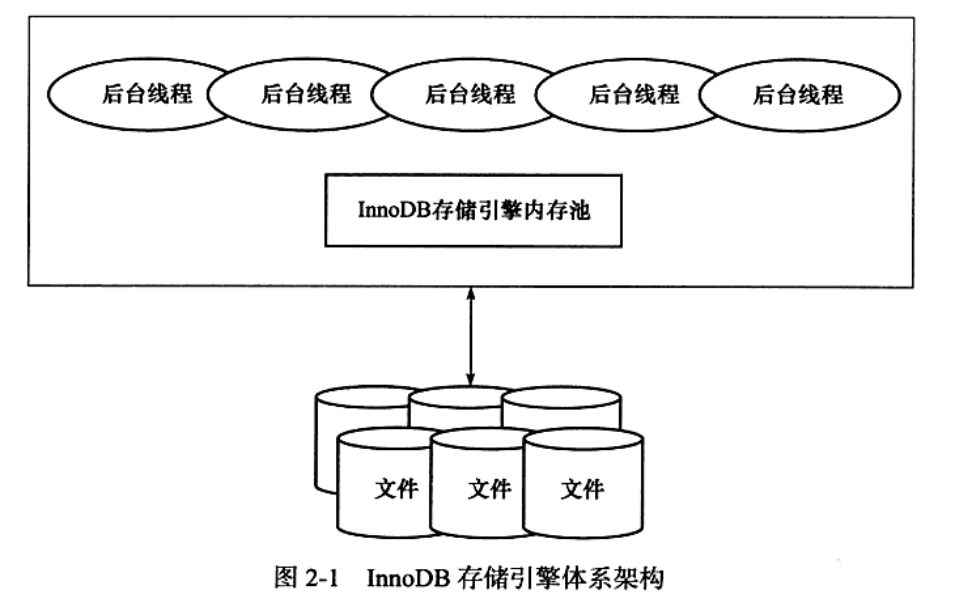
以前有人说过数据库和传统的文件系统最大区别在于数据库是支持事务的，但是mysql的myisam不就不支持事务吗。

连接到mysql是tcp连接。

### InooDB存储引擎

#### 2.1 架构

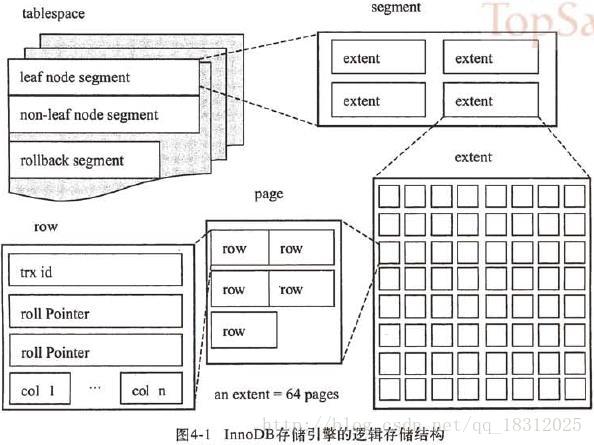
该存储引擎是第一个完整支持acid的引擎，行锁，MVCC，一致性非锁定读，有效的利用cpu和内存。从mysql5.1版本开始，mysql允许开发商动态的加载引擎，这样存储引擎的更新就可以不会mysql绑定了，类似于动态连接了。



InnoDB存储引擎的逻辑存储结构和 Oracle大致相同 ，所有数据都被逻辑地存放在一个空间中 ，我们称之为表空间 （ tablespace ） 。表空间又由段 （ segment ） 、区 （ extent ） 、页 ( page ） 组成 。页在一些文档中有时也称为块（block)

注意table space segement extent page这些东西都在磁盘里而不在内存里，内存会有这些东西的缓存。

一个页大约是16kb，一个区大约是64个页，一个区差不多1M.



如上图 tableplace segement extend page row

#### 2.2 内存与后台线程

Inoodb引擎里面有很多内存块，这些内存块组成了类似内存池的东西，这些内存维护了上面的多后台线程运行的需要的空间，缓存了一些磁盘上的数据，方便快速的读取，在修改之前在这里缓存（这特么不跟redis差不多了嘛）

后台线程的主要作用是负责刷新内存池中的数据，保证缓冲池中的内存缓存的是最新的数据。并将已修改的数据刷新到磁盘文件里，同时保证数据库发送一次的时候也能恢复到正常状态。

后面发现master thread线程的purge功能和刷新脏页都移到了其他线程，purge和 page cleanerthread

##### 2.2.1后台线程

1. master thread 是一个非常核心的后台线程，主要负责将缓冲池中的而数据异步刷新到磁盘，保证数据的一致性，包括脏页的刷新，合并插入缓冲等。

Masterthread是用来刷内存池数据到磁盘文件的。

1. IO Thread inoodb存储引擎中大量使用了AIO，来处理IO请求，这样可以极大的提高数据库性能。而IO Thread的工作主要是负责这些IO请求的回调(因为是AIO所以这里是回调)处理。IO Tread有四种，write，read （读写）insert buffer和log IO Thread. IO线程是和外界交互的估计操作表空间也是这个
2. Purge Thread事务被提交后，其所使用的undolog可能不再需要，就需要一个线程来回收那些不用的undo页，这个过程叫做purge操作，本来这些操作是在master thread中的，现在弄一个新的线程purge thread来做分担主线程压力。purgethread线程是用来回收东西
3. Page cleaner thread 这个是1.2版本中引入的，其作用是将之前版本中脏页的刷新操作放入到单独的线程中来完成。目的是减少mater thrad的压力。

##### 2.2.2内存

**缓冲池看起来是属于innodb的.**

读请求是先从缓冲池里读的,如果读到缓冲池就称为命中缓冲池.

Mysql的性能瓶颈也是硬盘的速度，所以mysql会有一个缓冲池，就是跟缓存似的一个东西。

在数据库中进行读取页的操作时，首先将从磁盘读到的页存放到缓冲池中，称为fix到缓存池中，下一次如果读相同的就命中该缓存页。

对于数据库中页的修改首先要修改缓冲池中的页，然后再以一定的频率刷新到磁盘里。 缓存池大小可以设置 个数也可以配置。

缓冲池里面除了缓存了数据页还缓存了索引页，也还有锁信息，字典信息等。

很多对于这种缓存数据的管理比如redis和mysql缓存池的管理都是用的LRU算法（latest recent used）最少使用算法。

就是把经常使用的也就是经常被命中的缓存放在前面，不经常使用的放在后面，然后当内存不够的时候就从后面开始删除咯。

#### 2.3 checkpoint

如果页被改了，磁盘还没来得及刷新的时候，这个时候页和磁盘文件的数据就是不一致的，那么这个页就暂时被称为脏页。这些就需要把页里的内容刷到磁盘文件。但是呢如果页没变一次就刷一次改一次磁盘文件，这样的开销是非常大的。这样的还有一个问题呢，就是如果还没刷呢就死机了这不就完蛋了么。

为了解决第二个问题就有了redolog，事务先写redo日志然后再修改页。

Redolog是为了保证持久性的。

Checkpoint技术，这个是啥目的呢，因为redolog占用的空间和缓存占用的空间肯定是要小于数据库的容量的，redolog如果有几T那恢复数据那恢复起来太慢了肯定是隔一段时间就处理一下，如果死机恢复的话就恢复那段时间没有处理的。

还有就是缓存页也要按时刷到磁盘空间然后这些缓存页再循环使用来缓存别的东西，不然数据库好几T的容量你用内存肯定缓存不住，到底啥时候刷呢？

Checkpoint流程做的事情就是把脏页刷新到磁盘，这样的话以前的redolog就不用保存了，空间就可以释放了，因为你的redolog是为了防止出意外而事务无法写到磁盘里的，现在脏页已经在checkpoint流程中写到了磁盘里，所以对应的redolog就可以不需要了。

所以当数据库重启后只需要上次checkpoint节点之后的redolog就可以了，这个节点就标志了啥时候的redolog要恢复，所以他叫point！！

啥时候发送checkpoint呢，很复杂，比如缓存池不够了把脏页刷到磁盘里释放空间，或者redolog不够用了页把脏页刷到磁盘里，这样以前的redolog就可以回收了这样就会释放redolog的空间，也会释放缓存池的空间。

根据功能来看checkpoint流程是masterthread线程完成的，因为主线程是把缓存池刷磁盘的线程。

Iothread看来是收到外面的请求改缓存池的线程。

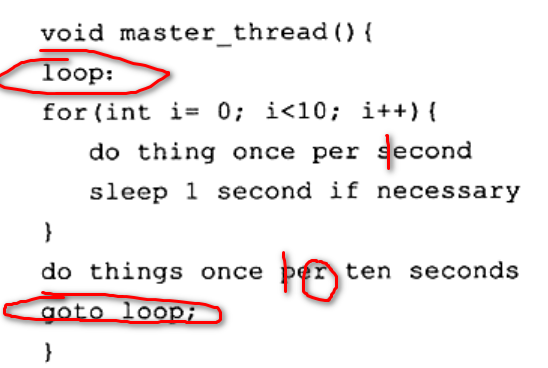
#### 2.4 MasterThread

主线程的操作说明了redolog在缓冲区是有一个buffer的,redolog也是 会有从缓冲刷到redolog页的操作的.

注意这里讲的masterthread里面是有删除undolog的操作的，后面版本purge是有专门的线程来删除的。

Masterthread线程是具有最高优先级别的线程，其内部有多个循环，主循环（loop），后台循环（backgroup loop），刷新循环（flush loop）,暂停循环（suspend loop）这几个状态，会在这几个状态里面来回切换。

主线程的操作是这样的



上面的代码用了goto，意思就是先执行一个循环每秒执行一种操作，执行10次后，就执行外面的东西然后再goto回去就是一直循环，外面的东西相当于是10秒执行一回，但是时间都不会太精确，因为代码执行需要时间啊，用sleep（）一秒还实现的。

其中每一秒的操作包括：

日志缓冲刷新到redolog（），即使这个事务还没有提交（总是）

合并插入缓冲（可能）

至多刷新100个innodb的缓存池中的脏页到磁盘（可能）

如果当前没有用户活动，切换到backgroudlood（可能）

从这里可以发现一个事务 是先写undolog作为备份，然后写缓冲池，数据到数据缓冲，重做日志到重做日志缓冲，然后每秒就把重做日志缓冲往重做日志文件里写，然后1秒或10秒都会看情况把数据缓冲的脏页写到磁盘

挨个分析，1，每一秒都会把重做日志缓冲的内容刷到redolog，即使这个事务还没有提交，对的，因为redolog是用来恢复一致性的，所以必须要保持比较高的刷新频率， 而且这样也可以解释为什么操作再多的事务commit的时候也是很短的，因为masterthread每秒都在刷到redolog，你很大前面的已经再前几秒已经刷到redolog里面了。

2合并插入缓冲，并不是每秒都发生的，innodb存储引擎会判断当前一秒内发送的io次数是否小于5次，如果小于5次，innodb认为当前的io压力很小，可以执行合并插入缓冲的操作。

3刷新100个脏页也不是美妙都会发送的。innodb存储引擎会通过判断当前缓冲脏页的比例，是否超过了配置文件中的innodb\_max\_dirty\_pages\_pct这个参数，默认为90即90%，如果超过了这个阈值，innodb存储引擎就会认为需要做磁盘同步操作，将100个脏页写入磁盘中。

接下来执行完10秒循环后执行的操作，

刷新100个脏页到磁盘（可能）

合并至多5个插入缓冲（总是）

将日志缓冲刷新到磁盘（总是）这个也是把重做日志缓存刷新到重做日志文件

删除无用的undo页（总是）

删除无用的undo页就是full purge操作！！！

Undolog就是比如必要delete或者update啥东西的时候，保存一下未操作前行数据的备份。。insert就啥也不用保存了。

若果没有用户活动（数据库空闲时）就会切换到backgroudloop循环，backgroundloop会执行以下操作。

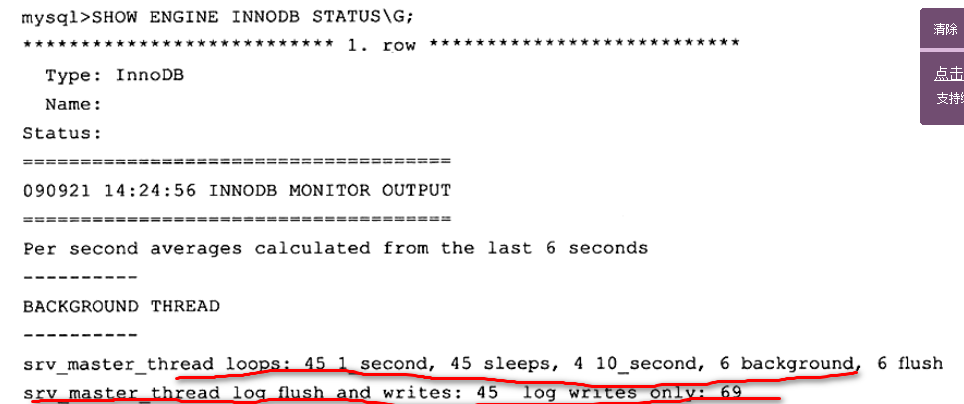
删除无用的undo页 总是

合并20个插入缓冲 总是

跳回主循环 总是

不同版本的master thread的操作有区别的。

SHOW ENGINE INNODB STATUS;查看innodb情况。

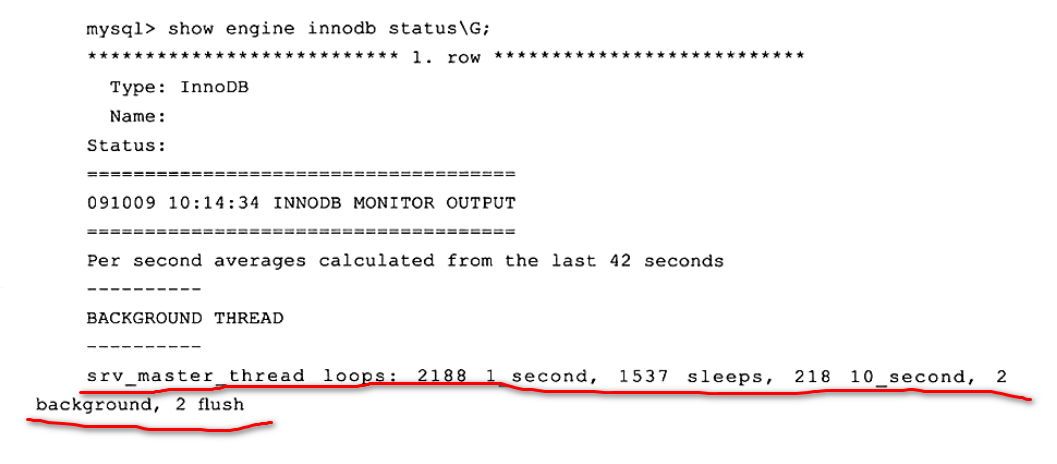


上面可以看到1秒的操作做了45次，sleep了45次，10秒的操作做了10次，

1：10的比例差不多，切换到了6次backgroudloop，有6 次flush操作。

这样看数据库的压力是比较小的。

如果服务器压力比较大的话是这样，



可以看到这里master thread loops做了2188次，而sleep只有1537次，这是为什么呢，因为现在压力比较大，所有减少了些sleep的次数，所以可以通过masterthread的次数和sleep的比例来判断数据库是不是很忙。

#### 2.5 innodb关键特性

##### 2.5.1 合并插入缓冲insertbuffer

Insert buff(插入缓冲)innodb存储引擎的关键特性之一,他并不是在内存中的缓存池的一部分,而是物理页的一部分,和一般的数据页一样.

这个主要要明白上面的聚集索引和非聚集索引的知识,在innodb中,若主键(这个就是一个聚集索引 primary key)自增,那么在页中存储时,会按照主键的顺序存放所以说数据是集中的,大部分在同一个页或几个页中,这样会减少磁盘的离散读取.数据都连续的保存在一个或几个页中这样的话就不会有离散读取了.要是数据这个页有几条,那个页有几条读的话就会很离散,很麻烦.这就是聚集索引数据和索引顺序一样的好处.

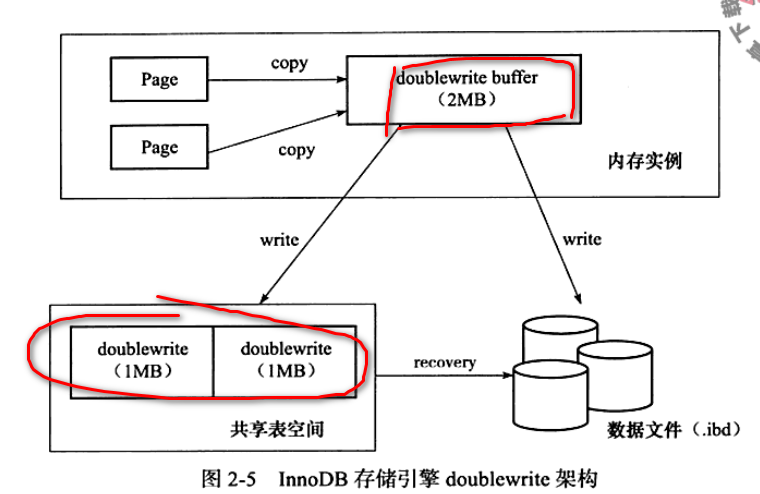
当然一个表是可以有其他的索引的,比如name订单号啊,这些为了加快查找速度也就是有索引的,而数据呢是按照聚集索引的顺序存储的,那么肯定就和非聚集索引顺序不一样了,如果不是和索引顺序的,如果你插入一条数据,那么索引数据肯定也要变化,索引数据也会多,但是聚集索引的增加很简单,因为本来就是IOT索引就是和数据在一起的,而且还是聚集索引,索引顺序还和物理存储一样还不是离散的,所以聚集索引的维护十分简单.

但是由于你插入了数据,比如你这个user表,name列也是有索引的,那么name的索引页数据肯定要更新,你插入一个就去更新下索引肯定不太好,这个合并插入缓冲的意思就是我并不是来一个插入的数据就修改辅助索引页,而是先修改insert buff把这个修改索引页的先弄到insert buffer插入缓冲里面,然后当聚集的比较多的时候再去插入.

##### 2.5.2 两次写

问题是这样，比如你在把脏页往磁盘刷的时候会有这种问题，一个16k的页刷到了一半mysql进程死了，这样就只刷新了一半，你可能想说可以用redolog来恢复，但是页是连续的一块内存往里面刷，redolog只是一个命令其实，如果刚才的磁盘的页被刷了4k，这4k并不是正好完整的sql命令。这样你用redolog这种命令集合去刷新就是错的，那么怎么才能正确呢。

Mysql用的就是doublewrite技术。



如上图，有两个脏页page，他们要刷到磁盘中，他是先这样的，现在内存里面复制这个page，弄出个doublewritebuffer内的空间，这个时候先把doublewritebuffer刷到磁盘的一个共享空间表里面，这个共享空间表是个数组，也是2m但是由于是数据内存是连续的，因为你只刷2m，而且磁盘是连续的，这是很快的，这个其实就当作磁盘数据页的备份，然后把doublewritebuff的内容刷到磁盘，如果刷的过程中mysql死机了，我们就用磁盘里的doublewritebuffer还原数据页.

（这里的意思是必须先写完磁盘的doublewritebuff数组，才会刷数据页 同步关系）

（redobuff刷到redologfile磁盘里是masterthread触发的，脏页刷新到磁盘是masterthread和backthread触发的）（上面说的doublewriterbuff脏页的内容不一定是一个完整的事务，可能是是个事务的一部分，然后触发了刷盘，所以redolog是有用的）

如果从缓冲池写doublewrite磁盘数组的时候失败了，那么就不会触发到后面的流程，磁盘的数据还是完整的，就可以直接用redolog来恢复，doublewrite只是为了解决刷脏页过程中死机导致磁盘页不完整不连续的问题。

Doublewrite可以配置不开启，但是最好还是开启。

##### 2.5.3 自适应哈希索引

首先这个是优化查询的。

自适应hash索引（adaptive hash index, AHI）的意思，数据库的索引是B+树，不用红黑树的目的就是减少对磁盘的操作嘛，自适应hash索引呢，就是在某些情况下，对缓存池的这个表的页数据建造hash的索引，b+树的索引很快但是呢如果数量不是很大的话还是hash快，hash的效率几乎是1，而且自适应hash索引是对缓冲池的内容建索引都不走磁盘的，那肯定快啊，但是为他构建自适应索引是有前提的，前提就是，对这个页的访问模式必须是多次一样的，

模式1，where a = xx 模式2 where a=xx and b = xxx

如果一直是1这种查询，就会触发自适应索引，如果是ab交替就不会触发。

触发条件： 该模式访问了100次 页通过该模式访问了N次，其中N = 也在记录数\*1/1 6

速度提升了几倍，也有开关的。

##### 2.5.4 异步IO

异步IO就是AIO，同步IO就是发了一个请求，必须等这个请求弄完响应了才能继续下面的流程发下一个。

AIO就是嘎嘎的发，异步IO，用回调来弄。

用户可以在发出一个IO请求后立即再发出另一个IO请求，当全部IO请求发送完毕后，等待所有的IO操作的完成。

AIO的另一个优势是可以进行IO Merge操作。

AIO需要操作系统的api来支持，只有windows和linux支持，mac是不支持AIO的。

默认mysql就是AIO的，可以通过配置来修改。

##### 2.5.5 刷新邻接页

意思就是再刷新一个脏页的时候，检查这个脏页所在区还有没有脏页，有的话一起把脏页刷了就行。这样的话可以减少对硬盘的操作，因为你一起刷脏页刷的比以前多了。这个如果是机械硬盘的话最好打开，要是固态硬盘的话最好关闭。

##### 2.5.6 启动关闭与恢复？？？？

### 3 文件

下面几个文件都是mysql数据库自己的，和插入式存储引擎innodb是没有关系的。

#### 3.1 参数文件

告诉mysql实例启动时在那里可以找到数据库文件，并且指定某些初始化参数，这些参数定义了某些内存结构的大小等设置，还会介绍各种参数的类型。oracel实例在启动时若找不到参数文件，是不能进行装载的。mysql实例在启动的时候如果找不到参数文件，这是参数会取决于编译时的默认值，如果默认值里也找不到就启动失败了。

在哪里找到数据库文件是定义在参数文件里面。

数据库的参数看起来就像是很多的键值对，键就是参数类型，值就是参数的值，

类似于innodb\_buffer\_pool\_size = 1G

参数分为静态参数和动态参数，动态参数可以用 SET XXX=XXX来随时修改，静态参数用这个命令修改就会出问题。

#### 3.2 日志文件

注意这里说的日志就给人看的，展示mysql数据库信息的，而上面说的redolog和undolog和二进制日志是mysql存储引擎自己用的，用来完成功能不可缺少的一部分。

用来记录mysql实例对某种条件做出响应时写入的文件，如错误日志文件，二级制日志文件，慢查询日志文件，查询日志文件等。

这些日志文件可以帮助DBA对mysql数据库的运行状态进行诊断，从而更好的对数据库层面的优化。

错误日志就是数据库出错了的日志，哪里错了。

慢查询日志就是一些比较慢的查询，数据库会记录下来这样DBA就可以找到慢的日志进行优化。

可以配置参数设定每分钟插入的slow log的条数，避免给slow log插入太多的数据。

Slow log文件可能很大导致不好分析，mysql提供了mysqldumpslow命令比如查询最慢的10条的命令

Mysqldumpslow -s al -n 10 david.log

Mysql5.1开始可以将慢查询日志放到一个表中，这个表叫做slow\_log

所以查看慢查询的命令就是select \* from mysql.slow\_log

注意慢查询日志并不是默认开启的，需要自己来开！！！

慢查询日志可以查看有多少查询是走的磁盘，有那些是走的缓冲池。

慢查询日志不仅可以保存那些查询的慢的，还有那些走磁盘太频繁的，还可以这两个都有的，可以自己在slow\_query\_type这个参数设置，如果设置为0，那么slow\_log就只会保存查询慢的，如果为1就是只保存IO次数多的。

#### 3.3查询日志

SELECT \* from mysql.general\_log 查看查询日志，这里会保存所有的查询操作。

#### 3.4二进制日志

首先二进制日志只保存update insert这种操作，查询相关的日志都在上面的查询日志里面。

还有就是 即使这个update语句没有修改数据库（where条件匹配不到）也是会记录到二进制日志。

那么二进制就可以用来恢复 审计和复制数据库了！！！

show variables like 'log\_bin'; 查看是否启用了二进制日志

Show variables like ‘datadir’ 这里就可以查看出MySQL的文件在哪里，这个命令的结果是一个路径，二进制文件就在这里。

二进制文件默认不是开的，要自己手动开。开了的话性能影响都不超过1%的。

#### 3.5 套接字文件

##### 3.5.1域套接字

这里讲的是域套接字 域域域

UNIX域套接字用于在同一台机器上运行的进程之间的通信。虽然因特网域套接字可用于同一目的，但UNIX域套接字的效率更高。UNIX域套接字仅仅复制数据；它们并不执行协议处理，不需要添加或删除网络报头，无需计算检验和，不要产生顺序号，无需发送确认报文。

UNIX域套接字提供流和数据报两种接口。UNIX域数据报服务是可靠的，既不会丢失消息也不会传递出错。UNIX域套接字是套接字和管道之间的混合物。为了创建一对非命名的、相互连接的UNIX域套接字，用户可以使用它们面向网络的域套接字接口，也可使用socketpair函数。

域套接字网络编程最不一样的地方是服务器端bind的时候用的是sockaddr\_un结构，客户端connect的时候用的也是sockaddr\_un结构，而不是sockaddr\_in或sockaddr。而对于sockaddr\_un结构，重点是给它提供一个bind()函数生成的socket类型文件的路径，即sockaddr\_un.sun\_path的值。并且客户端与服务器端的这个sockaddr\_un结构的sun\_path是一致的，通常这个路径是众所众知的，就像百度的域名那样。

意思就是域套接字是会产生一个文件的。

##### 3.5.2 mysql的套接字文件

Mysql用的就是域套接字，那么就会有一个mysql.sock文件。

#### 3.6 pid文件

当mysql实例启动时，会将自己的进程id写入一个文件中，该文件就是pid文件。默认为主机名.pid

Show variables like ‘pid\_file’\G

#### 3.7表结构定义文件

mysql的数据的存储是根据表进行的，每个表都会有与之对应的表结构定义文件，不管你用的是什么存储引擎，后缀名是.frm

#### 3.8 innodb存储引擎文件 在磁盘里！！！！

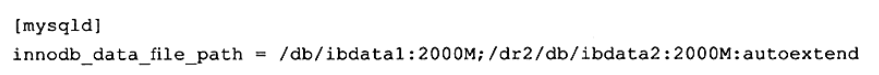
上面介绍的都是mysql数据库本身的文件，和存储引擎无关。

下面介绍存储引擎自己的文件。

##### 3.8.1表空间文件 在磁盘里！！！！

Innodb采用将存储的数据按表空间（tablespace）进行存放的设计。在默认配置下会有一个初始大小为10MB，名为ibdata1的文件。该文件就是默认的tablespace文件。可以设置大小。通过innodb\_data\_file\_path属性进行设置。

用户可以通过多个文件组成一个表空间，同时指定文件的属性。

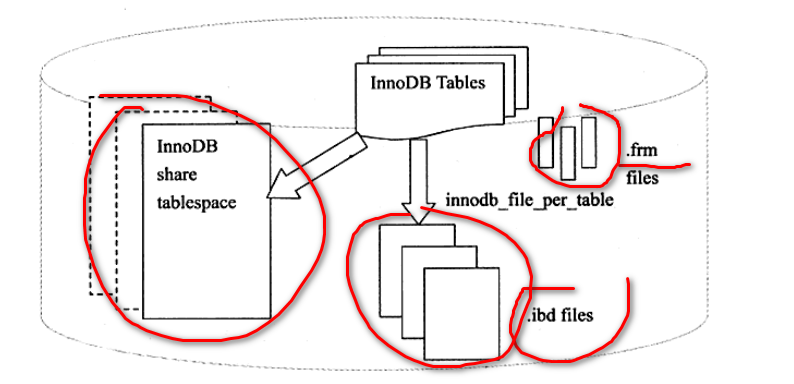


这里就是将/db/ibdata1和/dr2/db/ibdata2两个文件用来组成表空间。若这两个文件在不同的硬盘上，磁盘的负载就会被平均，这样就会提高数据库的整体性能。这两个文件虽然设置大小是2000m但是如果用完了是会自动增长的。

设置完innodb\_data\_file\_path参数后，所有基于innodb存储引擎的表的数据都会记录到该共享空间表中。若设置了参数innodb\_file\_per\_table（看看明白就知道啥意思了）则用户可以将每个基于innodb存储引擎的表产生一个独立表空间。独立表空间的命令规则为：表名.ibd。

所以说innodb存储引擎的表空间是个文件。如果不设置的话就是只有一个表文件，如果设置了上面的参数的话就是一个表有一个自己的表空间文件。

但是呢并不是就没有默认表空间了，还是有的，单独的表空间文件只存自己的，和这个表无关的数据还是存放在默认表空间里。

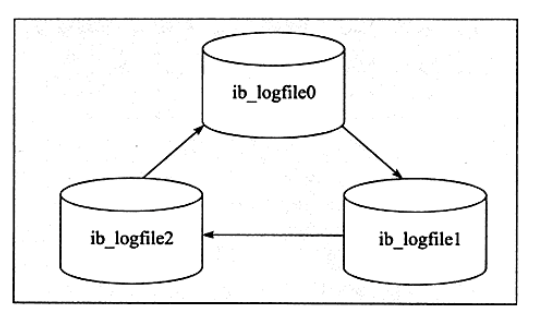


##### 3.8.2重做日志文件

Innodb存储引擎目录下会有两个名为ib\_logfile0和ib\_logfile1的文件。在mysql官方手册中将其称为innodb存储引擎的日志文件。

Innodb的日志文件就是redolog，可见这个很重要

每个innodb存储引擎至少有1个重做日志组，每个文件组下至少有2个重做日志文件。为了提供可靠性，可以设置多个镜像日志组。将不同的文件组放在不同的磁盘上，以提高可靠性。 循环使用，我也不知道啥意思。



参数innodb\_log\_file\_size指定每个重做日志的大小。在innodb1.2版本之前，重做日志文件总和不能大于4GB，之后的扩大到了512GB 这个东西是很大的。innodb\_log\_files\_in\_groups指定了日志文件组中镜像文件的数量。默认为2. innodb\_mirrored\_log\_groups指定了有多少文件组，默认为1，这样这个文件组就没有镜像了，如果磁盘已经有磁盘阵列功能了，就可以不开启重做日志的镜像功能。最后参数innodb\_log\_group\_home\_dir指定了日志文件的路径。

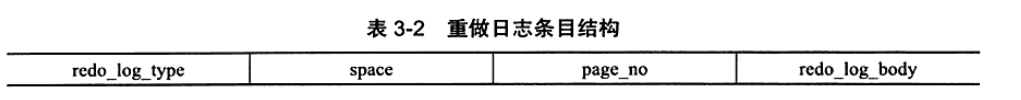
重做日志的大小设置对于innodb存储引擎的性能你有着非常大的影响。设置很大，恢复需要时间太久。设置太小，可能导致一个事务的日志需要多次切换重做日志，而且重做日志文件太小会导致频繁的发生async checkpoint，导致性能的抖动。

Redolog和二进制日志有啥不同呢？肯定不同的很多啊

首先二进制日志是mysql数据库的，redolog是inndodb存储引擎的。

二是内容不同，二进制日志记录的是一个事务的具体操作内容，即逻辑日志，update了什么，insert了什么，而redolog是物理日志，是记录了每个页要做什么样的操作。

三是写入的时间页不同，二进制日志文件仅在事务提交前进行写入，也就是不管事务有多少条语句，不论事务有多大，只写一次。而redolog是在事务执行过程中不断写入（MasterThread触发写redolog）



重做日志的结构 log\_type日志的类型。

Space表示表空间的ID

Page\_no表示页的偏移量

Redo\_log\_body表示每个重做日志的数据部分，恢复时愮调用对应的函数解析这个body的内容进行恢复。

写入重做日志文件的操作不是直接写，而是先进入一个redologbuffer，然后按照一定的顺序写入到日志文件（redo log file 包括ib\_logfile1 和ib\_logfile2）

重做日志缓存往磁盘写入时，是写512字节写一次的，也就是一个扇区的大小进行写入的最小单位，因此可以保证写入必定是成功的。因此重做日志写入过程中不需要有doublewrite。

Doublewrite的能生效的保证也是：从缓存池中的这块内存 doublewritebuff往磁盘的共享页数组写的2m的数据是完好的。

前面提到了从日志缓冲写入磁盘上的重做日志文件是这几种情况。

1. 是masterthread每秒都会讲重做日志缓冲的内容往磁盘刷一次。
2. 另一个触发刷缓冲重做日志到磁盘的情况是事务提交的过程会触发到redolog缓冲写磁盘中的redologfile。这个有一个参数设置，要设置成1，才行，不然有问题，设置成1了事务提交就会触发刷redolog缓存。

redolog的存在保证了事务的持久性。

### 4 表

表就是关于特定实体的数据的集合，这也是关系型数据库模型的核心。

#### 4.1 索引组织表

在innodb存储引擎表中，每张表都会有个主键 primary key，如果在创建表时没有显式的定义主键，则innodb存储引擎会按如下方式选择或者创建主键。

1. 首先判断表中是否有非空的唯一索引（unique not null）如果有，那么就把该列当作主键。。。废话么
2. 如果不符合上述条件，innodb存储引擎会自动创建一个6字节大小的指针。

当表中有多个非空唯一索引式，那么就会选择建表时候的第一个非空唯一索引当作主键，后来alter的不算的。

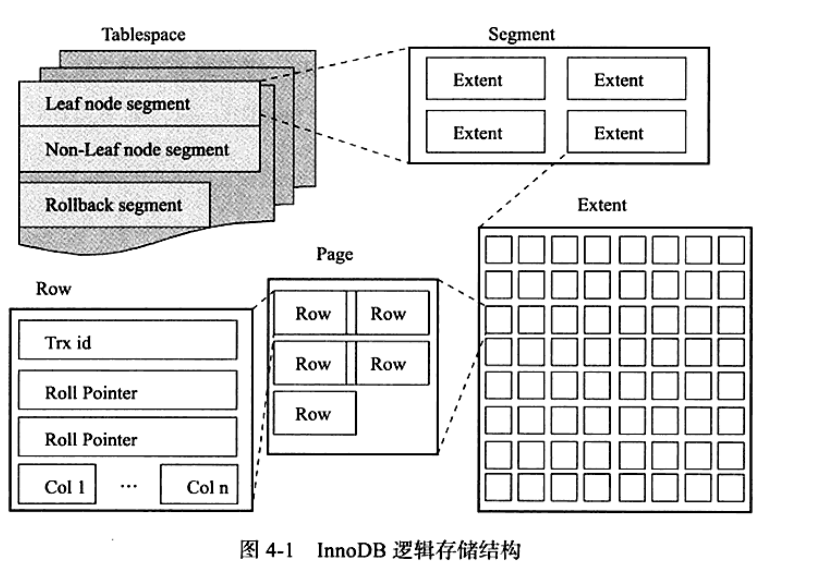
Select \*,\_rowid from users\_user \_rowid就是主键列的值。但是如果把主键变成那种两列主键这个\_rowid就不好用了。

Create table a（a int,b int,primary key(a,b)）engine=innodb

这样就是把两个列的组合当作主键。

#### 4.2 innodb逻辑存储结构

从innodb存储引擎的逻辑存储结构看，所有数据都被逻辑地存放在一个空间里，称为表空间。表空间又由段，区，页组成。页在一些文档中有时也称为块。



注意上面的段是分为也就是tablespace里面的segment是分为rollback segment和leaf segment和non-leaf segment 回滚节点子叶和非子叶节点三个的。

##### 4.2.1 表空间 tablespace

上面说了如果不设置的话是会有一个共享表空间的，然后所有表的数据 索引 回滚信息 二次写 啥的都放在这个共享表空间里。如果设置了那个参数，就是每一个表都有自己的表空间文件，但是还是有这个共享表空间文件存在的。

这样的话每张表的表空间内存放的只是数据，索引和插入缓冲bitmap页，系统事务信息。

插入缓冲索引页，系统事务信息，二次写缓冲，undolog等还是存放在原来的共享表空间内。

通过show variliable命令可以查看到一开始共享表空间的大小仅有58M，后来我们通过开启一个事务不停的update操作不提交，这样再看共享表空间大小就会发现已经是114M了，生成了大量的uodolog，但是呢这个表的表空间确没有明显变大。如果此时你rollback了，共享表空间并不会缩小，也就是undolog并不会删掉，而是会标记成没用了，以后这块空间就可以被再利用了。

上面说过master thread会有个full purge操作，即删除没用的undolog。10秒后上面的undolog就会被删除。

也就是full purge操作和rollback操作都会触发把空间标记成可用。而full purge则会删除这些没用的空间。

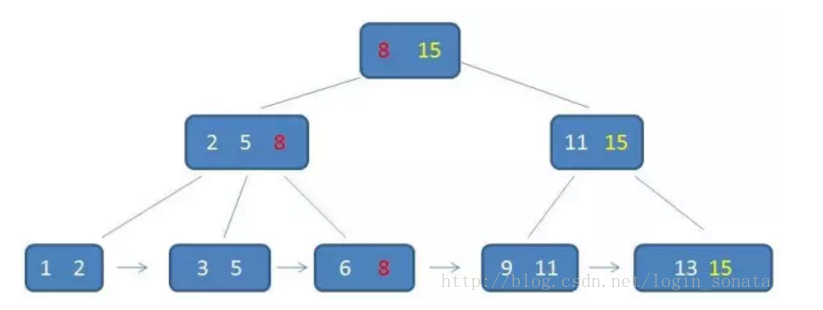
看看这个undolog也赶上啥咯，这俩都会起效。

Undolog就是比如必要delete或者update啥东西的时候，保存一下未操作前行数据的备份。。insert就啥也不用保存了。fullpurge确定之前备份的undolog没用了就会删除。

##### 4.2.2段 segment

上面介绍了IOT，innodb只支持IOT，也就是innodb只支持索引组织表。

表空间是由各个段组成的，常见的段有数据段，索引段，回滚段等。



先看下B+树，mysql的索引就是B+树。B+树的一个好处就是中间的节点不存数据，只有索引值，所有的数据都存在最下面一行，都在子叶节点。而且最下面还用链表串了起来，这样用索引进行范围查询的时候十分方便。

IOT的话，索引即是数据，数据也是索引。

那么也就是我们的mysql的数据和索引（现在合并称为数据）都在B+树的子叶节点中，也就是最下面那排。但是注意。表空间的数据段存的就是这些子叶节点（包括索引与数据），表空间的索引段存的就是B+树上面的那些索引。

对段的管理都是innodb自己维护的，没啥参数让你改的。

##### 4.2.3 区 大小1MB

区是由连续页组成的空间，在任何情况下每个区的大小都为1MB，一个页的大小是16KB，也就是一个区有64个页。

Innodb后来有了压缩页，也就是通过参数可以改变页的带下，为2 4 8k，那么一个区中页的个数页会成倍增加。无论怎么变化区的大小总是1M.

上面说了一个区最小是1M，但是数据库创建的时候默认表空间是96KB，连表都不够1M.因为是在每个段开始时，先用32个页大小的碎片页来存放数据。在使用在这些页之后才是64个连续页的申请。这样做的目的是，对于一些小表或者undo这类的字段，可以在开始时申请较少的空间，节省磁盘的开销。

##### 4.2.4页

页是最小的数据库单位。

常见的页类型有：数据页（B+TREE NODE）undo页（undolog page）系统页（System page）事务数据页（Transaction system page）插入缓冲位图页（insert buffer bitmap）插入缓冲空闲列表页（insert buffer free list）未压缩的二进制大对象页（uncompressed BLOB page）压缩的二进制大对象页（compressed BLOB page）

##### 4.2.5行

一个页是16KB，一个页最多存(16KB/2)-200条记录，即最多存7992条记录。innodb是面向行的，也就是数据是一行行存的。面向列的数据库也是现在一个很好的发展方向.

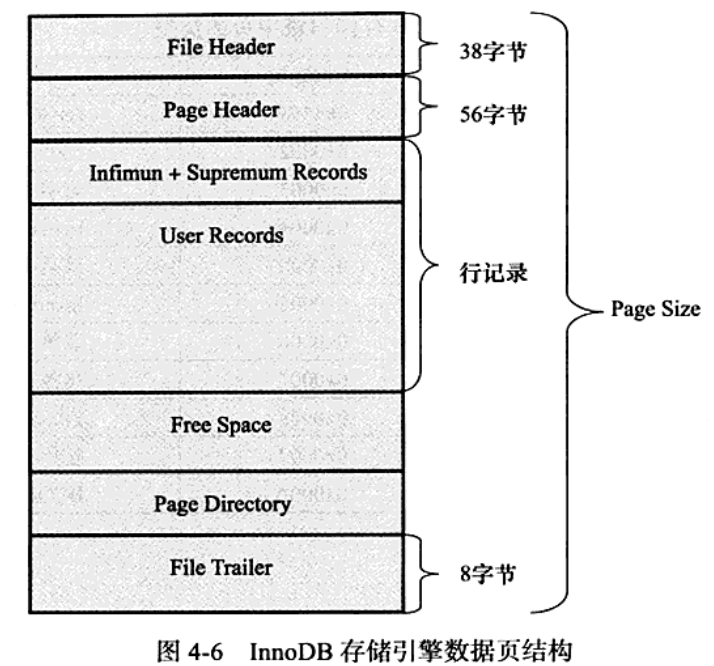
#### 4.3 innodb行记录格式 ?

#### 4.4 innodb数据页结构

经过上面的学习,我们已经知道了页是innodb存储引擎管理数据库的最小的磁盘单位.页类型为B-tree Node的页存放的即是表中行的实际数据了!!!!!!!!

上面我们可以知道了,innodb是IOT,那么就是索引和数据在一起,那么数据存储的方式肯定也和B树有关系,现在我们知道了,其实b树中的一个子叶节点就是一个页 嗯!

那么数据在页中的user\_record中就是一行行从上到下排列的吗,不是的,是按照堆的形式排列的!!!!!!!!!!!



其中的user\_record就是一行行的数据,其他的都是用来标记这个页的一些属性,这些属性的类的大小都是固定的.

##### 4.4.1 file header

有8个部分就,共占38字节,有checksum的值,表示这个页在整个表空间中位置的东西.存储引擎的类型.表示该页属性哪个表空间.表示这个页的前后的btree node页的位置的东西.

##### 4.4.2 page header

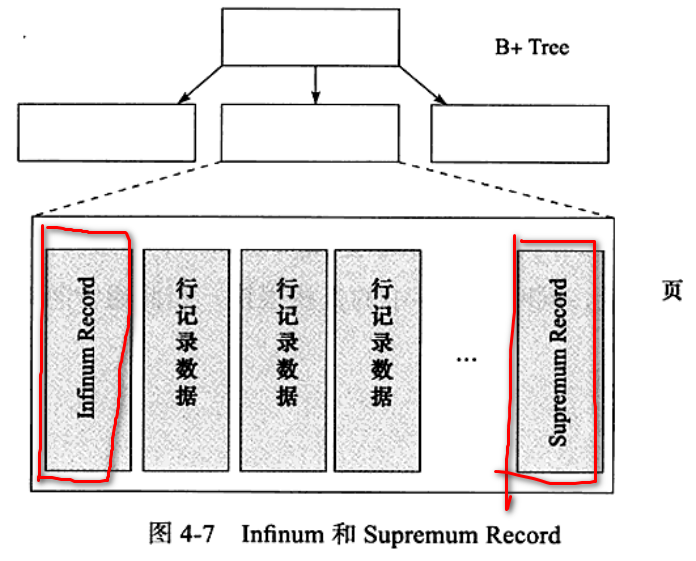
接着file header的部分是page header,该部分用来记录数据页的状态信息,由14个部分组成,共占用56个字节.

分为槽数,堆中第一个记录的指针,数据在页中是根据堆的形式存放的!!!!!!!!

堆中的记录数 指向可重用空间的首指针 已删除记录的字节数,即行记录结构中deleteflag为1的记录大小的总数.最后插入记录的位置.最后插入的方向,一个方向连续插入记录的数量,该页中记录的数量 修改当前页的最大事务id 当前页在索引树中的位置,索引id,表示当前页属于哪个索引.B+树数据集页非叶节点所在段的segment header.B+树数据页所在段的segment header.

##### 4.4.3 infimum和supremum record

在innodb存储引擎中,每个数据页中的user\_record中有两个虚拟的行记录,用来限定记录的边界.infimum记录是比该页中任何主键值都要小的值,supermum是比任何可能大的值还要大的值.这两个值在页创建时被建立,并且在任何情况下不会被删除.



4.4.4 user record和free space

User record就是之前讨论过的部分,即实际存储行记录的内容.再次强调,innodb存储引擎表总是B+树索引组织表的.

free space很明显的指的就是空闲空间,是个链表结构,在一条记录被删除后,该空间就会被加入到空闲链表中.

##### 4.4.5 page directory

Page directory(页目录)中存放了记录的相对位置,有些时候这些记录指针称为slots槽.与其他数据库系统不同的是,在innodb中并不是每个记录拥有一个槽,innodb存储引擎的槽式一个稀疏目录.

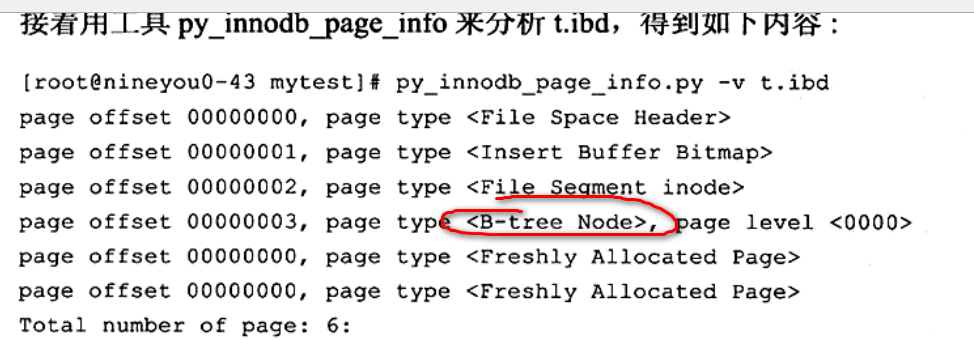
好像是这个意思,行数据在一个页中确实是按照堆的形式排列的,还有一个东西叫做槽,槽记录着这些数据的指针,但是呢并不是一个行数据就有一个槽,而是一个数据对应几个槽,你先确定哪个页后,再在page directory里面二叉查找(为啥是二叉查找呢,因为是个堆)是哪个槽,然后确定是槽里的哪个指针(由于在innodb存储引擎中必须通过recorder header中的next\_record中的next\_record来继续查找相同记录,同事page directory很好的解释了recorder header中nowned值的含义,这个玩意好像就是标记了这个槽里面有几个指针)

需要牢记的是,B+树索引本身并不能找到具体的一条记录,而是能找到该记录所在的页.数据库把页载入到内存,然后通过page directory再进行二叉查找.只不过二叉查找的时候复杂度很低,同时在内存中的查找很快,因此通常忽略这部分查找用的时间.

##### 4.4.6 file trailer

为了检测页是否已经完整的写入磁盘(如可能发送的写入过程中磁盘损坏,机器关机等),innodb存储引擎的页中设置了filetrailer部分.

就是通过checksum实现的.



这个btreenode就是数据页,因为类型是btreenode.

#### 4.5 约束 主键 外键 唯一 default等保证数据完整性

##### 4.5.1 数据完整性

关系型数据库系统和文件系统的一个不同点是,关系数据库本身能够保证存储数据的完整性,不需要应用程序的控制,而文件系统一般需要在程序端进行控制.几乎所有的关系型数据库都提供了约束机制,该机制提供了一条强大而简单的途径来保证数据库中数据的完整性.

数据完整性有一下三种形式.

实体完整性保证表中有一个主键.在innodb存储引擎表中,用户可以通过定义primary key或者unique key约束来保证实体的完整性.

域完整性保证数据每列的值满足特定的条件,在innodb存储引擎表中,域完整性可以通过以下几种途径来保证. 合适的数据类型,外键约束,触发器,default约束强制,外键.

Innodb存储引擎提供了以下几种约束,

Primary key unique key foreign key 也就是说主键,唯一约束 外键约束,default都是为了保证完整性的了.

对于不符合约束的数据,你可以配置,是让mysql报错失败,还是报个warning,还是mysql自己转换成一个正确的数帮你插入.

ENUM约束可以让数据只有那几个合法.

触发器也可以完成约束功能.

#### 4.6 视图

在mysql数据库中,视图是一个命名的虚表,他由一个sql查询来定义,可以当作表使用.与持久表不同的是,视图中的数据没有实际的物理存储.

视图是由一条查询语句定义的!!!!!!!!!!!

基本表的改变,视图也会改变.

如果你改变了视图,那么基本表也会对应的改变.

作用,方便操作,方便查询语句的使用,这是最重要的作用.

更加安全,数据库授权命令不能限定到特定行和特定列,但是通过合理创建视图,可以把权限限定到行列级别.

咋使用呢,比如用户表课程表订单表三个表,你要查询一个东西,需要三个表联合查询,每次都需要这个搞个长sql,可以弄个视图,

创建视图

CREATE ALGORITHM = UNDEFINED

DEFINER = `root`@`localhost`   
SQL SECURITY DEFINER VIEW `view\_user\_course` AS (

看到了吗 as里面是个select语句!!!!!!!!!!!!!!

SELECT

`uc`.`id` AS `id`,

`u`.`name` AS `username`,

`c`.`name` AS `coursename`

FROM

(

(

`user` `u`

LEFT JOIN `user\_course` `uc` ON ((`u`.`id` = `uc`.`userid`))

)

LEFT JOIN `course` `c` ON ((`uc`.`courseid` = `c`.`id`))

)

);

通过视图查询,看起来和从表里查询没有什么区别

SELECT

vuc.username,

vuc.coursename FROM

view\_user\_course vuc WHERE

vuc.username = '小张'

视图呢作为一种方便查询的工具,肯定要把性能调到最优.

视图呢,如果是多表弄出来的视图,那么你是不能同时修改两个表的(意思就是你修改了视图的几列,这几列在多个表里)

但是你只修改一列如果不冲突的话是没问题的.

#### 4.7 分区表

##### 4.7.1 分区概述

分区功能并不是在存储引擎层完成的,因此不是只有innodb存储引擎支持分区,常见的存储引擎myisam ndb等都支持分区.但是有的存储引擎就不支持.

Mysql数据库在5.1版本时添加了对分区的支持.分区的过程是将一个表或索引分解为多个更小更可管理的部分.就访问数据库的应用而言,从逻辑上讲,应用只看到一个表或者一个索引,但是在物理上这个表或索引可能由数十个物理分区组成.每个分区都是一个独立处理的单元,也可以组成一个大的单元.

Mysql数据库支持水平分区(按行分区),不支持垂直分区(按列分区)

Musql数据库支持以下几种类型的分区.

Range分区,按照行一个连续的区间分区.

List分区,离散的分区.

Hash分区,用户根据自定义的hash表达式进行分区.

Key分区,根据mysql数据库提供的hash函数分区

##### 4.7.2 分区和性能 OLTP和OLAP

对表做个分区,数据库的查询就会加快了,是这样吗?不是的,实际上可能根本高级不到查询速度的提升,甚至会发现查询速度急剧下降!!!

数据库的应用分为两类:一类是OLTP(在线事务处理)电子商务,网游等大多数都是OLTP,另一类是OLAP(在线分析处理),如数据仓库,数据集市.在一个实际的应用,比如网游,玩家操作引起的数据库变动就是OLTP,而游戏厂商对玩家日志的分析就似乎OLAP类型.

对于OLAP的应用,分区的确是可以很好地提高查询的性能,因为OLAP应用大多数查询需要频繁的扫描很大的一张.假设有一张1亿行的表,其中有一个时间戳,用户的查询需要从这种表中获取某一年的数据,如果按时间戳进行分区,则只需要扫描对应的分区即可.(是哦,分区的好处就是这样呢)

但是对于OLTP的,其实他每次只会读取几条数据,B+树读几条已经很快了,2,3次IO就出来了,你分区还得确定去哪个区,麻烦.

### 5索引与算法

#### 5.1什么样的列加索引

加索引的列得是那种该列数据很多,并且数据大部分都不一样的,比如这个列的数据只有男或女 这两个值,你还加个毛线的索引,但是如果这列是orderno这种每一列都不一样的,那么肯定要加B+树索引的.

索引应该开发的时候就加上,别让DBA去加,因为开发才会更明白的知道索引的使用.

索引不是添加越多越好,占用空间挺大的,要找到一个合适的点.

#### 5.2 索引的分类

支持常见的B+树索引 全文索引 自适应哈希索引 (非空,唯一,主键等)

自适应Hash索引上面说过了,不是人为产生的,而是mysql感觉这个东西访问太多了给搞的一个索引.

#### 5.3 B+索引

B+索引就是普遍意义的索引.

注意!!!!!!!!!!!!!!!!!!B+树索引并不能找到一个给定键值的具体行.B+树索引能找到的只是呗查找数据行所在的页,然后数据库把页读到内存再进行查找.

B+树一般是2-4层,所以查找到一个正确的页基本只需要2-4次IO即可.

聚集索引和辅助索引(非聚集索引)

B+树的分裂,就是这个意思,比如一个B+树节点,设定他的一个节点可以容纳10个数据,当11个数据想插入这个节点的时候就插入不进去了,那么这个节点就会分裂,变成两个,或者三个节点,分裂规则有很多,具体的可以自己设置.

#### 5.4 Cardinality选择性值

Cardinality值,这个的意思就是一个列中的不重复的值有多少个,比如一个列是订单号,那么订单号的去重值/列数,那么就肯定是1,那么这种就要加索引了,如果一个列是性别,整个表有1000条数据,但是选择性值只有2个.

这个值在查看索引属性的命令中可以看到.

并不是所有的列都适合加索引,对于性别,地区,类型这些可取值的范围很小的列,称为低选择性

像手机号啊,主键id这种各个都不一样的就是高选择性.

如何查看索引是否具有高选择性呢,可以通过show index结果中的cardinality列来观察.cardinality是一个预估值,并不是一个准确的值.

在实际应用中,Cardinality/n\_rows\_in\_table应尽可能接近1.如果非常小,那么用户需要考虑是否还有必要创建这个索引.

Cardinality的统计是在存储引擎中进行的,而且这个不是每次操作都统计的,而是采样统计的.

#### 5.5联合索引

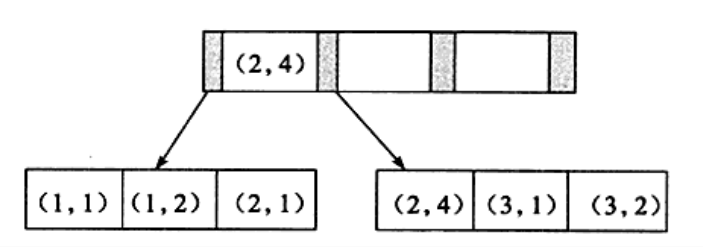
联合索引比如a b c d 四列联合索引可以使用

a,ab,abc,abcd 这四列列组合成索引的原因

其实一个联合索引并不是产生了多个B+树,一个多列的联合索引还是只有一个B+树.这个B+树的键不仅是一个列的值了,而是多个列的值的一个组合,按照从左到右的顺序排列.如上图两个整数列的联合索引.

B+树中的节点我们知道是根据键值的大小从上到下排下来的,那么现在键里面是多个数据了,该怎么排呢,其实就是这样的,先比第一个,第一个小了就小,如果第一个一样就比第二个,第二个小了就小,前两个都一样了就比第三个,前三个小了就小.

这就可以解释了为什么左一 左二 左三可以用了 因为B+树的结构无论是按照左一 左二 左三 左四来算都是拟合的.



覆盖索引 innodb存储引擎支持覆盖索引,即从辅助索引中就可以查询到想要的数据,而不用再去聚集索引那里去查数据了.

使用覆盖索引的一个好处是辅助索引不包含整行记录的所有信息,故其大小要远小于聚集索引,就是一个有数据的索引呗,专业点就是覆盖索引不需要回表.

有的时候,通过explain命令去分析sql语句执行过程的时候,会发现优化器没有选择索引去查找数据,而是通过扫描聚集索引,扫描全表来得到数据,这种情况多发送于范围查找,join连接等情况下.了

### 6 锁

消除一个关于行级锁的一个神话,人们总是认为行级锁会增加开销,实际上,只有当实现本身会增加开销的时候,行级锁才会增加开销,innodb存储引擎不需要锁升级,所以一个锁和多个锁的开销是相同的.

数据库系统使用锁是为了支持对共享资源的并发访问,提供数据完整性和一致性.不同的引擎不同的数据库之间对锁的实现可能是完全不同的.

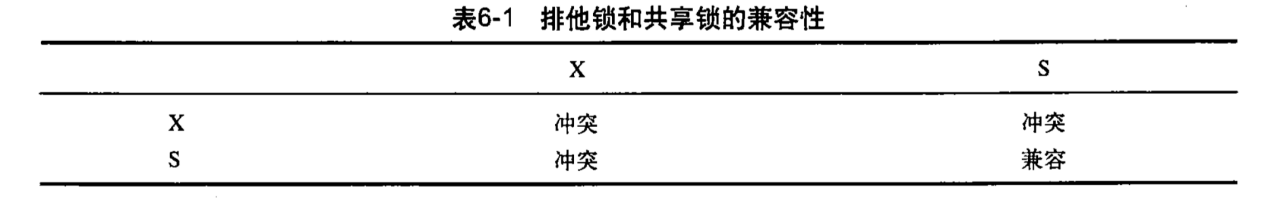
对于myisam来说,锁是表锁,并发情况下读是没有什么问题的,但是并发插入的时候,性能就不行了. 对于sqlserver来说,是页锁,想对于表锁来说是好一点的,这个会有锁升级,会升级到表锁,并发性能是不行的.

innodb的存储引擎锁的实现和oracle非常相似,提供一致性的非锁定读,行级锁,行级锁没有相关的开销,可以同时得到并发现和一致性.

#### 6.1 innodb实现的锁

共享锁 s lock 允许事务读一行数据 lock in share mode

排他锁 x lock 允许事务删除或者更新一行数据 for update



意向锁其实不会阻塞除全表扫之外的任何请求.

SELECT \* from information\_schema.INNODB\_LOCKS

SELECT \* from information\_schema.INNODB\_TRX

SELECT \* from information\_schema.INNODB\_LOCKS\_WAIT

这三个可以查看当前的事务和锁的状态等信息

#### 6.2 一致性风非锁定读

如果读取的行正在执行delete update操作,这是读取操作不会因此儿等到行上锁的释放,而是回去读取一个快照数据,该实现是通过undolog来实现的,undolog本来就是回滚要用的,因此快照数据本身是没有额外的开销的.

mysql默认的隔离级别是rr,这是就是默认用的非锁定读.



我们姑且这样理解,undolog是快照的一部分,因为快照的一个字段就是指向这个undolog的指针,这样就可以用于回滚,我们也认为快照会有个状态一个是已提交,一个是未提交,rr和rc读的都是已提交的快照,ru读的是未提交的快照.

undolog是不能随便删除的,因为可能有陈年老事务还要用你的老版本的快照,要想删除,得确保的老的事务都已经提交了.

rr读的时候如果这个数据没有被其他事务使用,那么直接读,如果被其他事务操作中就读小于自己的最新版本,而rc读的就是最新版本.

可以看出,如果我们一个写一个读,可以接收读到老数据,这样mvcc就可以解决问题.如果我们不能接收读到老数据,那么就是s锁(测试可知,如果吧一列加上了s锁,这一列我们是不能修改的,不用事务普通的修改也不行,但是是可以读的)

我们认为普通的写操作即使不在事务里面也是要申请x锁的.

比如吧一个列的值\*10后加100,就必须得枷锁,不然可能少操作几次,会覆盖.

所以这种要想让写必须有先后顺序挨个来的就得加x锁

比如改名,其实加不加锁都那样,因为update是原子操作,但是如果你的操作很多的,流程比较长的,比如对用户发快递,发快递流程跑完才会最后该数据库的状态,这样你连点两次就可能会发两次,代码是你得加锁,枷锁也只能保持操作是挨个顺序的,你获得锁之后第一件事就是验证状态对不对.!!!!,不然你获得到了锁就直接操作也是有问题的.

#### 6.3 自增锁

如果自增锁弄的不好,高并发插入的时候就可能爆出这个错误,Deadlock found when trying to get lock

最老的方式是auto-inc locking这个的意思就是,这个自增锁不会占用很久,只要sql语句执行完了自增锁就释放了,不用等到事务提交,但是还是要问题,还是要等到insert语句执行完才可以,这样并发isnert还是会有问题.

不用等事务提交后就释放锁的后果就是事务中insert了一个语句占用了一个主键,即使你后来rollback了这个主键id也被跳过了.

意思就是自增id这个东西,是mysql自己保持自增且唯一的,有三种方式,通过参数innodb\_autoinc\_lock\_mode来控制,值有三个, 0,1,2

第一个就是最老的auto-inc locking

第二个就是对于普通的insert,还有就是已经知道要插入多少个的时候,不是在sql执行完就释放锁了,而是申请到自增id就释放,自增id通过互斥量来维护,这样肯定是比较块的,但是对于那种不知道有多少行的还是用auto-inc locking来操作.

第三个就是2 这样所有的方式都是用互斥量,但是这样自增id可能不是连续的.

#### 6.3 锁的算法

innodb中有三种行锁的算法设计

Recod lock 记录锁

Gap lock 间隙锁 锁定一个范围,但不包含记录本身

Next-key lock 锁定一个范围,并且锁定记录本身

Nextkey lock是结合了gaplock和recordlock的一种锁定算法,在nextkeylock算法下,innodb对于行的查询都是采用这个锁定算法.对于不同的sql查询,可能设置共享nextkeylock或者排他的nextkeylock.

rr模式下, nextkeylock是默认的行记录锁定算法.

比如你一个语句

begin

Select \* from user where age < 6 lock in share model

(如果不加in share model的话就不会有哪个效果,但是就会产生幻读的问题了)

另一个地方如果你去insert age 《=6的,就会被阻塞 =6也会被阻塞的,因为nextkeylock就是包含边界的,这样就不会有幻读问题了,

所以解决幻读问题也可以用lock in share model配合nextkeylock

#### 6.4 锁的问题

丢失更新

就是你select的余额, +500后然后update回去,同时另一个事务也在做同样的操作,如果你们的流程交叉了的话就会导致第一个update丢失了,解决这个case的办法就是update 余额= 余额 + 500

对于一些复杂的场景,就只能加锁来保证了.

不可重复读

为啥叫这个名字呢,因为你前后两次读的同一个数据的值不一样,就叫不可重复读了.

阻塞

比如锁冲突了不就阻塞了,默认的阻塞时间是50s.

死锁

就是死锁咯.

锁升级

是指将当前锁的粒度降低.数据库可以把一个1000个行锁升级成一个页锁,这样1000个锁对象就变成了一个页锁对象了, 操作这一页的请求都会互相排队,其他页的不会阻塞,页锁同理也可以升级成表锁.目的是减少锁占用过多的内存.比如sqlserver就十分珍贵锁资源,不会让你搞很多细粒度的锁的.

但是呢,innodb是不会有锁升级的,因为在这个引擎里面,1个锁和1000个锁是一样的.为啥呢,,,因为innodb会给记录搞一个类似余bitmap位图的东西,加锁了就是1,不加锁就是0, 反正bitmap不怎么占空间,而且实现空间都分配好了,所以加不加锁都不会影响什么资源问题.

### 7 事务

Acid 原子性 一致性 隔离性 持久性

对于innodb引擎来说,在rr级别的时候就通过nextkeylock解决了幻读问题,也就是innodb在rr级别下,就可以实现了序列化级别的功能呢. 但是你得自己去用lock in share model来操作,你不加这个s锁的话是不会解决幻读问题的.

#### 7.1 redolog

通过redolog可以保持事务的持久性和完整性,原理前面应该醒了

redolog是物理日志,binlog是逻辑日志.

#### 7.2 undolog

功能就不说了,redolog是放在重做日志中(持久化的file),undolog是放在数据库内部的一个特殊segment中,undo段.

我们通常对于undo有误解,以为undo用户物理的恢复数据,恢复到执行之前的样子,不是的,undolog只是逻辑的恢复成之前的样子,如果你是insert语句,undolog里面就是delete,如果你是delete,undo里面就是一个相反的全量insert,如果你是一个update,那么undo就是一个相反的update

我们可以通过命令行查看undo页的大小.事务提交并不会马上删除undo页,而是主线程周期的删.

这里我们基本可以确定undolog和多版本的版本并不是一个意思.

多版本控制的一部分功能是用了undolog.

#### 7.3 事务控制语句、

反正就是begin setautocommit=0这些,注意普通的东西也是有事务的,不过人家是autocommit的.

还有就是事务还可以有个savepoint和savepointback的操作,就是你在事务的某一个阶段打一个点,然后pointback的话就会回滚到这个点,这个点之前的不会被回滚.

#### 7.4 分布式事务2pc xa seata

<https://blog.csdn.net/zhao1299002788/article/details/102999879>

#### 7.5 事务的问题

不要循环commit, 循环commit一是效率低,因为你每commit一下都会触发一次写redolog到磁盘的,如果你只commit一次性能会高很多.,不过最好也别弄太长的事务,不然会阻塞资源啊.

### 8 备份与恢复

#### 8.1分类

Hot backup热备

指在数据库运行中直接备份,对正在运行的数据库没有任何影响,onlinebackup

Cold backup 冷备

数据库地址下进行备份,简单,一般只需要拷贝相关的 物理文件就可以了.

offlinebackup

Warm backup 温备

同样是在数据库运行的时候执行,但是会对数据的操作有影响

逻辑备份

文件内容是可读的,一般是文本,里面是sql语句 mysqldump就是逻辑备份

物理备份

物理备份就是拷贝数据库的物理文件,这种备份的恢复时间会很短.

#### 8.2 冷备

定期搞一个脚本来备份mysql的各种文件

#### 8.3 binlog备份

#### 8.4 热备

ibbackuo是innodb官方提供的热备工具.

1,记录备份开始时,innodb的redolog的lsn

2,拷贝共享表空间文件以及独立表空间文件

3,记录拷贝完表空间文件后,innodb存储引擎reodlog检查点的lsn

4,拷贝在备份过程中产生的redolog