

**内部组件**

Mysql的统计信息：

索引的“区分度”（个人理解为散列情况）。一个索引上不同的值越多，这个索引的区分度就越好。一个索引上不同值的个数，称为“基数”。基数越大，区分度越好

**基数**的统计通过采样统计而来，所以并不准确（精确的统计代价太高）

采样过程：InnoDB默认会选择N个数据页，统计这些数据页的不同值，得到一个平均值，然后乘以这个索引的页面数，得到索引的基数；当变更的数据行数超过1/M的时候，会自动触发全新一次的基数采样（索引统计）

索引统计方式

1. innodb\_stats\_persistent 为 on 时，表示统计信息会持久化存储，N=20，M=10
2. innodb\_stats\_persistent 为 off 时，表示统计信息在内存中，N=8，M=16

analyze table T 命令可以用来重新统计索引信息

select \* from t force index(a) where XXXXX 强行使用a索引查询

优化器：选择索引（找到最优的执行方案，用最小的代价去执行语句）是优化器的工作。

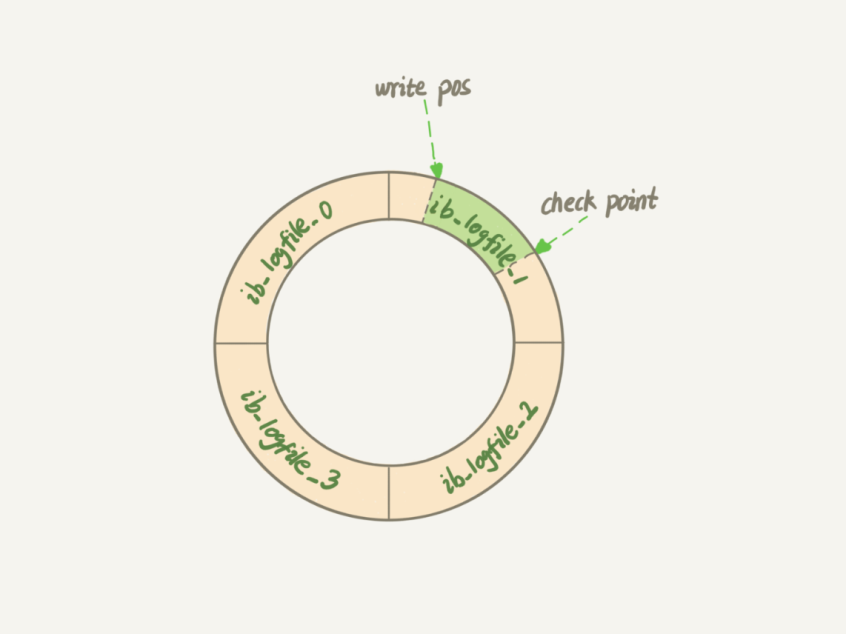
代价：扫描数据的行数；优化器是否会使用临时表；是否排序；等等因素

**RedoLog**

innoDB引擎特有的日志；

1. WAL技术：Write-Ahead Logging，先写日志，再写磁盘。

InnoDB将更改信息写道redoLog中，并更新内存，直到适当的时候将这些记录更新到磁盘中；如果写记录太多（write pos将要追上check point的时候），则会先将一部分记录更新到磁盘中。



1. crash-safe：因为有redolog，保证数据库在异常情况重启后，之前提交的记录不会丢失。

crash-safe保证的是：

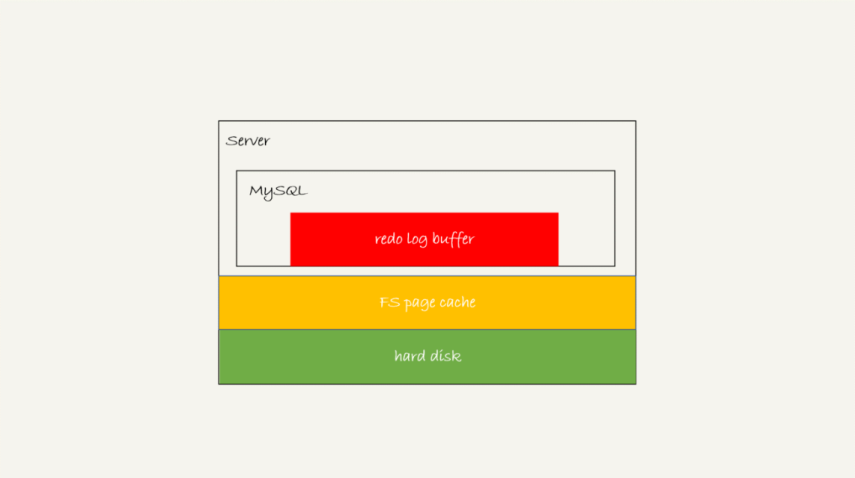
1. 如果客户端收到事务成功的消息，事务就一定持久化了
2. 如果客户端收到事务失败的消息（比如主键冲突，回滚等），事务就一定失败
3. 如果客户端收到“执行异常”的消息，应用需要重连后通过查询当前状态来继续后续的逻辑。此时数据库只需要保证内部（数据与日志之间，主库与备库之间）一致就可以了
4. 参数：

innodb\_flush\_log\_at\_trx\_commit = 1，每次事务的redoLog都持久化到磁盘。保证mysql异常重启后数据不丢失。

1. redo log buffer

内存块，用于先存redo日志；事务执行过程中，生成的redo log先要写在redo log buffer。

全局共用



Mysql redo log 状态

1. 存在redo log buffer中，物理上是在mysql的进程内存中
2. 写到磁盘(write)，但是没有持久化(fsync)，物理上是文件系统的page cache
3. 持久化到磁盘，对应的hard disk

redo log 写入策略：InnoDB提供了innodb\_flush\_log\_at\_trx\_commit 参数

等于0时，每次事务提交都只是把redo log写在redo log buffer中

等于1时，每次事务提交都持久化到磁盘中；（prepare阶段redo log就要持久化一次，因为恢复的时候需要—依赖于 prepare 的 redo log和binlog恢复数据）

等于2时，每次事务提交只是把redo log写到page cache中

Redo log 写入磁盘场景：

1. InnoDB后台有线程，每隔 1 秒，就会把 redo log buffer 中的日志，调用 write 写到文件系统的 page cache，然后调用 fsync 持久化到磁盘。
2. redo log buffer占用的空间达到InnoDB\_log\_buffer\_size一半的时候，后台线程会主动写磁盘（write 只是写到page cache中，并没有fsync）
3. 并行事务提交的时候，一个事务写了一半到buffer中，另一个事务已经提交，并且写入策略等于1（每次事务提交都持久化到磁盘中），那么写到一半的事务的redo log也会一起持久化到磁盘里
4. WAL机制主要得益于：
5. redo log和binlog都是顺序写，磁盘的顺序写比随机写速度要快
6. 组提交机制，可以大幅降低磁盘IOPS消耗

组提交机制知识点：

1. 日志逻辑序列号（log sequence number， LSN）

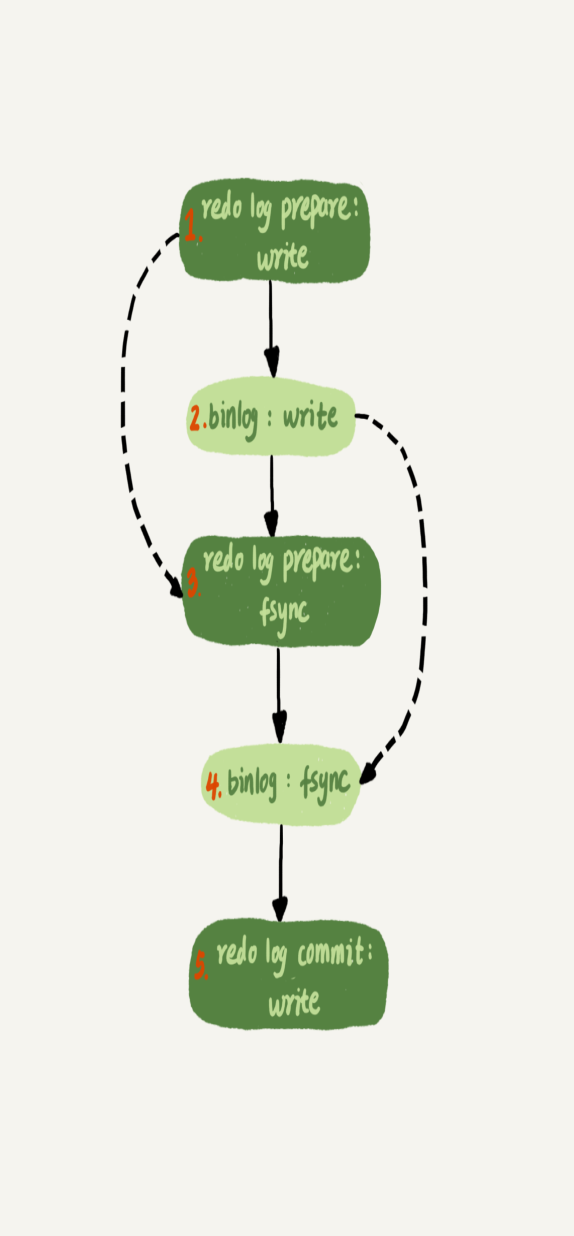
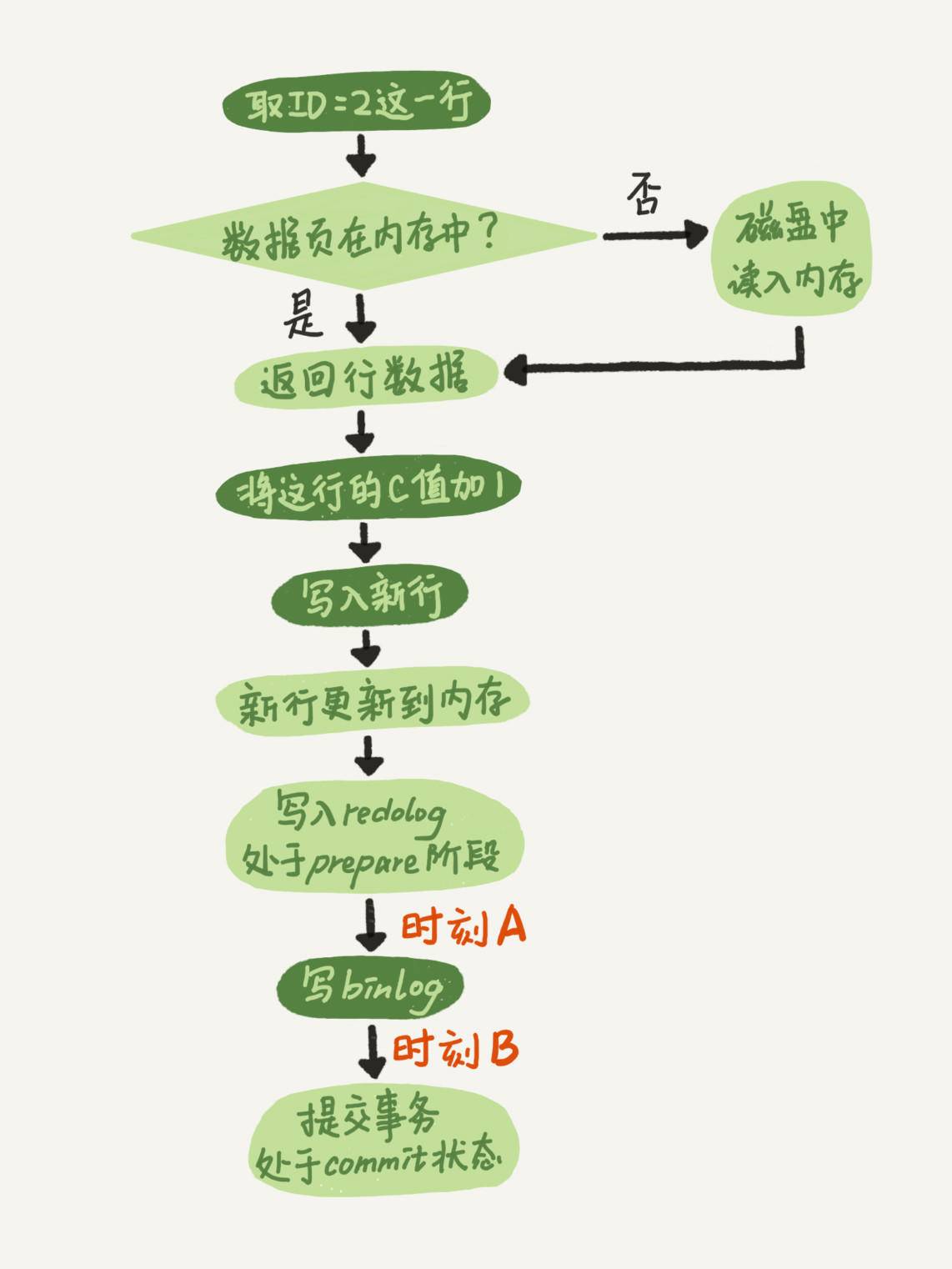
单调递增，用来对应redo log一个个写入点。每次写入长度为length的redo log，LSN就会增加length。LSN也会写到InnoDB的数据页中

并发事务一起（组）提交时，组员越多，效果越好。

**BinLog（归档日志）**

Server层日志，所有引擎都有

1. MyISAM没有crash-safe能力
2. update 操作流程，（从执行器操作开始）（浅色在innoDB引擎中执行，深色在执行器）



1. 两阶段提交：prepare和commit。保证状态和逻辑一致。

异常情况如何处理

1. 时刻A发生了崩溃，此时写了redo log，但是没有写binlog，redo log还没提交，所以崩溃恢复时，数据会回滚；binlog没写，所以也不会传到备库里
2. B时刻，binlog写完，redo log没有提交前发生crash

注：崩溃恢复规则（如果redo log里面的事务是完整的，也就有了commit标识，则直接提交；如果redo log 中事务只有完整的prepare，则判断对应的事务binlog是否存在并完整：如果完整则提交事务，不完整则回滚事务）

1. 参数：

sync\_binlog = 1 ，每次事务的binlog都持久化到磁盘，保证binLog不丢失。

1. MySQL如何知道binlog完整：

Statement格式的binlog，最后会有commit；

Row格式的binlog，最后会有一个XID event

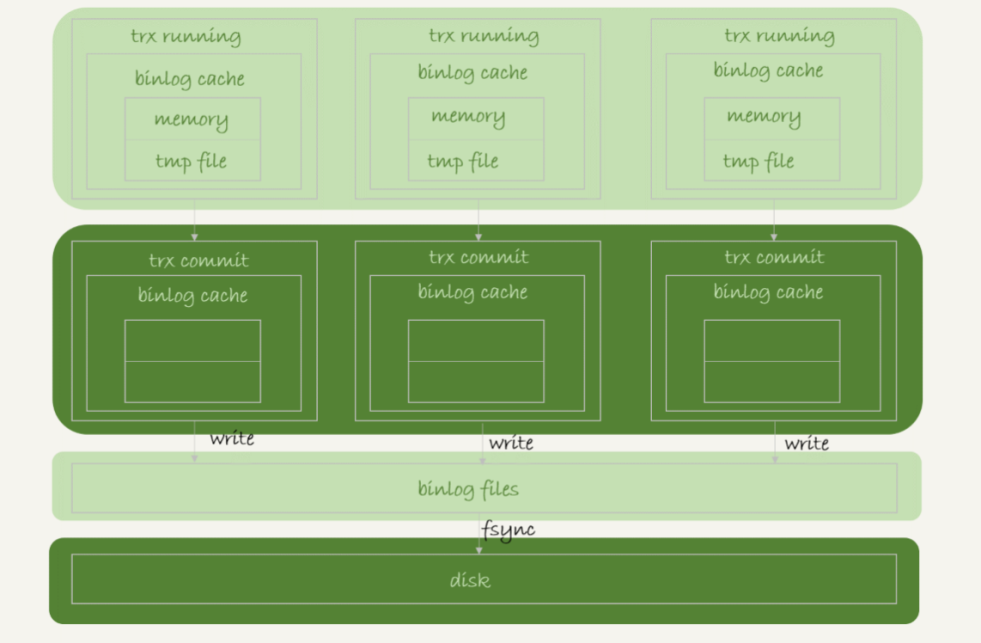
另外：MySQL5.6.2版本以后，还引入了binlog-checksum参数，用来验证binlog的正确性

1. binlog写入机制

逻辑：先写入binlog cache中，事务提交的时候再把binlog cache写到binlog日志文件中。

系统给 binlog cache 分配了一片内存，每个线程一个，参数 binlog\_cache\_size 用于控制单个线程内 binlog cache 所占内存的大小。如果超过了这个参数规定的大小，就要暂存到磁盘。

提交的时候执行器把cache完整的事务写到binlog中，并清空binlog cache



Write：日志写到文件系统的page cache中，并没有数据持久化到磁盘中，速度较快

Fsyne：数据持久化到磁盘中

Write和fsyne的时机由参数 **sync\_binlog**控制：

sync\_binlog =0，每次事务提交只write，不fsync

sync\_binlog =1，每次事务都会执行fsync

sync\_binlog =N，每次事务都write，累计N个事务后再fsync

所以，将sync\_binlog设置比较大，会提升性能；风险是如果主机发生异常重启，那么会丢掉最近的N个事务的binlog日志

**redoLog和binLog区别**

1. redoLog是innoDB引擎特有，binLog所有引擎都能用
2. redoLog是物理日志，记录的是“在某数据页做什么修改”，binLog是逻辑日志，记录“给id=2的数据的C字段+1的操作”
3. redoLog循环写，固定空间会用完；binLog追加写，当达到一定大小后会切换下一个，不会覆盖以前的日志
4. binlog cache是每个线程自己维护，而redo log buffer是全局共用。因为binlog cache不能被打断，一个事务的binlog是要连续写；而redo log 不需要，中间生成的日志写到redo log buffer中，里面的内容还可以跟随某一事务的提交而顺便写到磁盘里

**redoLog和binLog关联**

redolog和binlog有共同的数据字段：XID，崩溃后恢复的时候，会按顺序扫描redolog

1. 如果碰到既有prepare、又有commit 的redolog，就直接提交
2. 如果碰到只有prepare、而没有commit 的redolog，就根据XID去binlog找对应的事务

处于prepare阶段的redolog加上完整的binlog，重启就能恢复，mysql为什么这样设计？

数据与备份一致性有关；如果在B时刻也就是写完binlog后崩溃，但此binlog已经写入并且被从库使用。所以在主库上也要提交这个事务。采用此策略，保证主从数据一致。

事务执行期间，没有到commit阶段，如果发生crash的话，redo log丢失，但不会导致主备库数据不一致：因为这时候binlog还在binlog cache中，没有发给从库。Crash后redo log和binlog都没有，所以事务也没有提交，数据是一致的。

事务（原子性，一致性，隔离性，持久性）  
在引擎层实现（InnoDB）

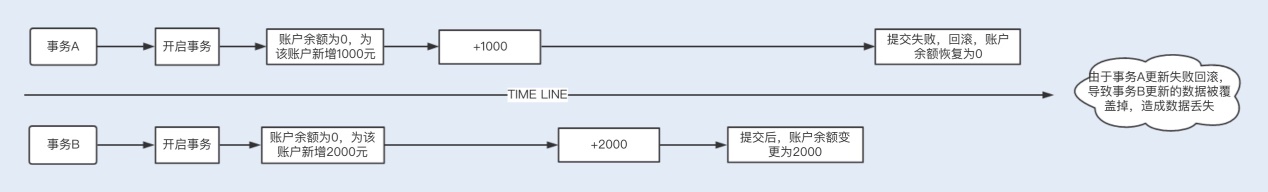
Begin/start transaction 命令并不是开启事务的起点，在执行到后面的第一个操作innodb表的语句，事务才真正启动

启动事务方式：

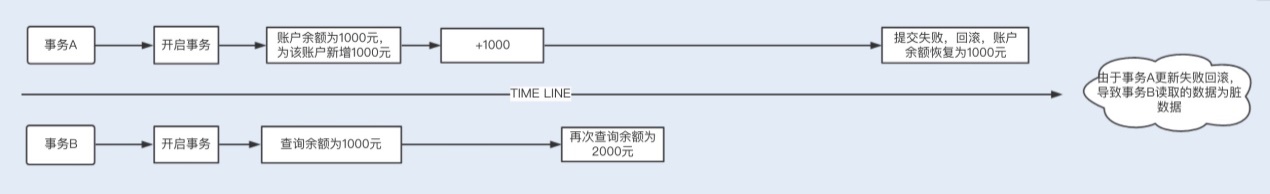
1. 一致性视图在执行第一个快照读语句时创建的
2. 一致性视图在执行 start transaction with consistent snapshot 时创建的

并发事务带来的问题

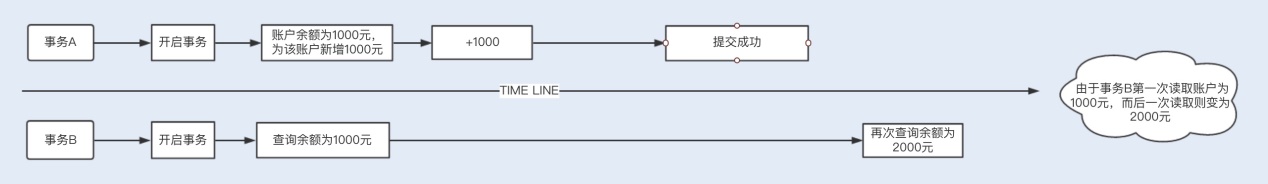
1. 数据丢失



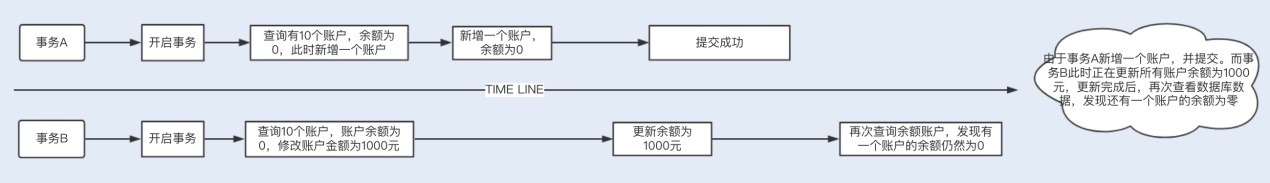
1. 脏读



1. 不可重复读



1. 幻读



隔离级别：

1. 读未提交：在事务A读取数据时，事务B读取数据加了共享锁，修改数据加了排他锁。这种隔离级别会导致脏读、不可重复读和幻读
2. 读已提交：
3. 可重复读
4. 可串行化

视图：

1. view。它是查询语句定义的虚拟表，create view
2. InnoDB在实现MVCC时用到的一致性视图，即consistent read view，用于支持RC（读提交），和RR（Repeatable Read，可重复读）隔离级别的实现

**索引**

覆盖索引：索引上的信息足够满足查询的需求，不需要在主键索引上取数据

最左匹配：通过修改联合索引的顺序，减少索引的创建

索引下推：通过索引和非索引字段查询，mysql5.6之前，查到索引数据后需要回表，再对比非索引字段；5.6版本后引入索引下推优化：在索引遍历过程中，对索引中包含的字段会先做判断，直接过滤掉不满足的条件，减少回表次数（减少存储引擎查询基础表的次数，也可以减少MySQL服务器从存储引擎接收数据的次数）

唯一索引：能用普通索引尽量用普通索引，因为唯一索引无法使用change buffer

实际问题：

1. 在字符串上创建索引
2. 直接创建完整索引，这样比较占用时间
3. 创建前缀索引，节省空间，但会增加查询扫描次数，并且不能使用覆盖索引
4. 倒序存储，再创建前缀索引，用于绕过字符串本身前缀的区分度不大的问题
5. 创建hash字段索引，查询性能稳定，有额外计算消耗；与C一样都不支持范围搜索
6. 索引失效

函数操作：

1. 条件字段函数操作

对索引字段做函数处理，可能会破坏索引值的有序性，因此优化器就决定放弃走树搜索功能

1. 隐式类型转换  
   隐式对索引字段做函数处理  
   （不会走索引，因为将varchar类型的tradeid 转换为int）  
   select \* from tradelog where tradeid=110717;  
   select \* from tradelog where CAST(tradid AS signed int) = 110717;
2. 隐式字符集编码转换  
   隐式对索引字段做函数处理

例如：当utf8转化为utf8mb4

utf8与utf8mb4区别：

utf8最多只只支持3个字节，而utf8mb4可以支持4个字节的字符。utf8mb4是utf8的超集。

**Change Buffer**

（在内存中有拷贝，也会被写入磁盘） change buffer 会写在redo log中，所以数据库崩溃，在恢复的时候也能找回来

读数据：InnoDB的数据是按照数据页为单位读取的。读取数据时，是以页为单位，将数据从磁盘中整体读到内存。每个数据页默认为16k

更新数据：如果要更新的数据页在内存中，那么直接更新内存；如果数据页不在内存中，在不影响数据一致性的前提下，InnoDB会将更新操作缓存在change buffer 中，这样就不用从磁盘读取数据到内存中；在下次查询需要访问这个数据页的时候，再将数据页读入内存，然后执行change buffer中与此数据页相关的操作

Merge：将change buffer中的操作应用到原数据页，得到最新结果的过程

Merge触发：

1. 访问查询数据页到内存
2. 后台线程系统定期merge
3. 数据库正常关闭（shutdown）过程中merge

Merge流程：

1. 从磁盘读数据页到内存（老版本的数据页）
2. 从change buffer中找到这个数据页的change buffer 记录（可能是多个），一次应用，得到最新版的数据页
3. 写redo log。redo log 包含数据的变换和change buffer的变换
4. Merge结束。此时，数据页和内存中change buffer 对应的磁盘位置都没有修改，属于脏页，之后各自刷自己的物理数据（其他过程）

使用change buffer好处：

更新操作在change buffer，减少读磁盘，数据从磁盘读到内存涉及随机IO访问，成本很高；数据读入内存需要占用buffer pool，所以还能避免占用内存，提高内存利用率；

使用change buffer 条件：

唯一索引不使用：因为其所有的操作都要先判断此操作是否违反唯一约束条件，而此操作必须将数据页读入内存才能判断。既然读到了内存，那么更新内存会更快，就没必要使用change buffer了

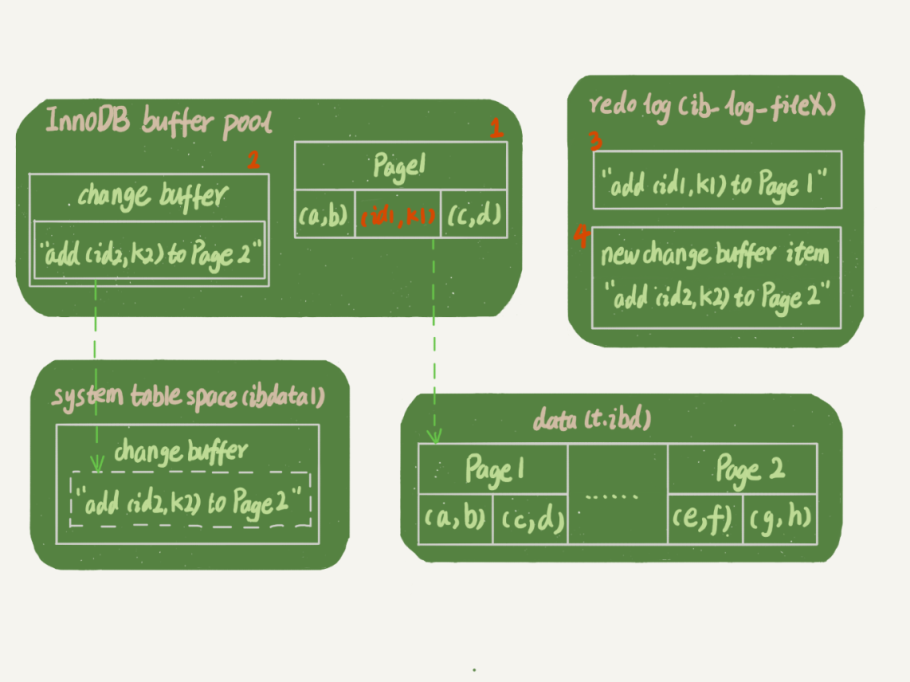
普通索引能够使用change buffer

change buffer使用的是buffer pool中的内存，因此不能无限扩大。

大小参数设置 innodb\_change\_buffer\_max\_size 50 （表示change buffer最多只能占用buffer pool的50%）

使用场景：写多读少；因为 在merge前，change buffer记录的变更数据越多，那么意味着减少访问磁盘的次数就越多，也就越有收益；如果写完直接读，那么还要将数据读取到内存，change buffer 作用不大，而且还增加维护change buffer的成本

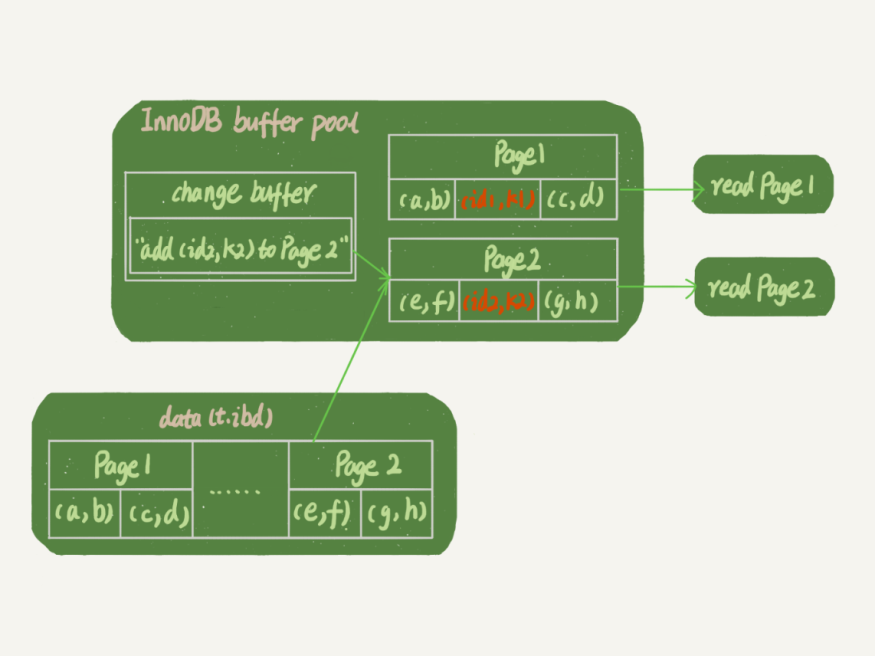
**Change buffer 和 redo log**



假设insert k1所在的数据页在内存中，k2所在的数据页不在内存中，步骤如下：

1. Page 1在内存中，直接更新；
2. Page 2 不在内存中，写入change buffer中；
3. 将两个insert语句写到redo log中

以上做完，事务完成。操作成本很低，写两处内存，一次磁盘（两个insert合并一起写到redo log中），而且是顺序写

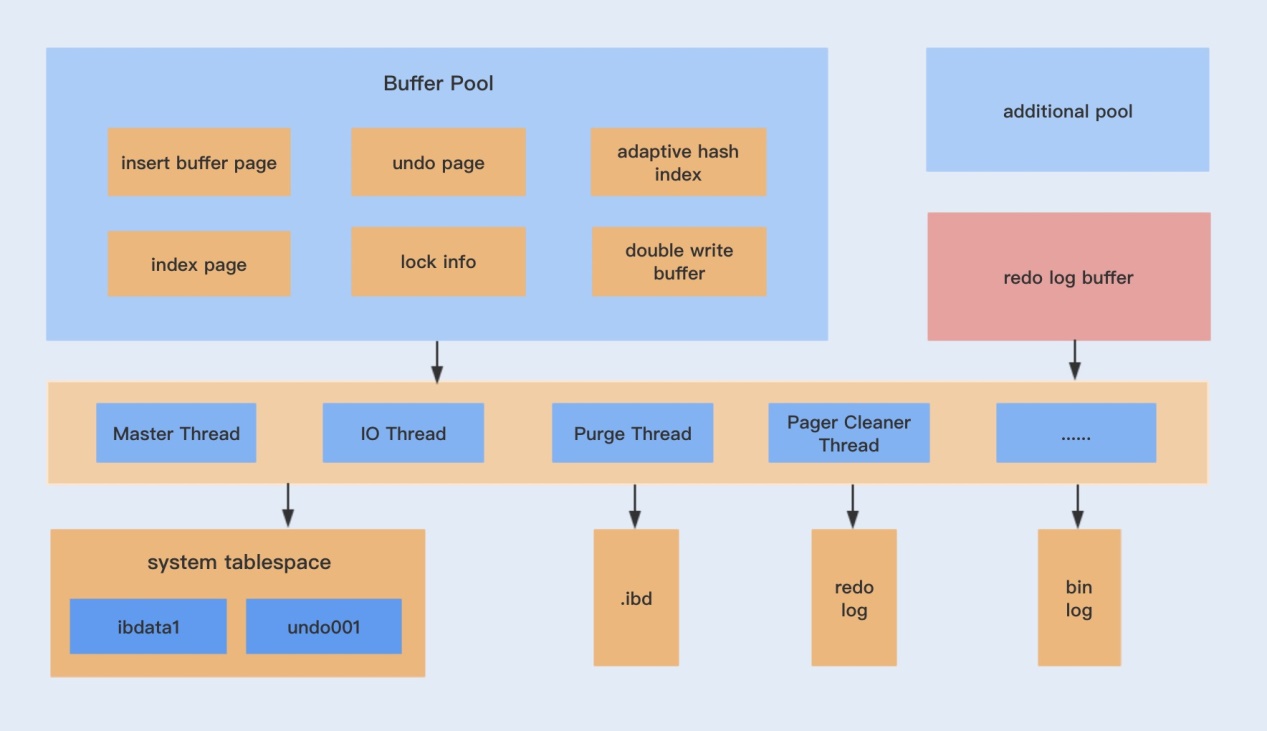


读取上述数据的时候（k1，k2）

1. Page 1 原本在内存中，直接读取
2. Page 2 读取时要将page2 从磁盘中读到内存，然后应用change buffer的操作日志，生成正确的数据版本，再返回

综上：redo log 主要节省的是随机写磁盘的IO消耗（改为顺序写）；change buffer主要节省的是随机读磁盘的IO消耗

**InnoDB**



1. 内存池（缓冲池）：提高整个数据库的读写能力

缓冲池中有索引页、数据页、undo log页、changebuffer缓存、自适应hash索引、锁信息等

1. 后台线程

Master Thread：将缓冲池中的数据异步刷新到磁盘中，change buffer和undo 页的回收；

IO Thread：负责读写IO

Purge Thread：回收事务已经提交的undo log

Pager Cleaner Thread用于协助Master Thread刷新脏页到磁盘；减小Master的工作压力

1. 存储文件

.frm文件：存储表元信息，主要是表结构

**锁**

全局锁：对整个数据库实例加锁

命令：Flush tables with read lock (FTWRL)，让整个库只读

使用场景：全库逻辑备份

表级锁：表锁、元数据锁

表锁：lock tables … read/write，unlock tables主动释放锁，也可以在客户端断开的时候自动释放表锁。除了限制其他线程的读写外，也会限制本线程的操作。

例如：某个线程 A 中执行 lock tables t1 read, t2 write。其他线程写 t1、读写 t2 的语句都会被阻塞。同时，线程 A 在执行 unlock tables 之前，也只能执行读 t1、读写 t2 的操作。连写 t1 都不允许，自然也不能访问其他表。

元数据锁（Metadata Lock）：不需要显示操作。在alter等语句执行前申请锁，等到整个事务提交后才会释放锁，这是为了避免数据库挂掉。

目的：保证读写正确性，保证在增删改查等操作时，被其他线程修改表结构导致报错。

用法：当对表做怎删改查的时候，加MDL；当对表结构做修改的时候，加MDL

读锁之间不互斥，多个线程可同时增删改查；

读写之间，写写之间互斥，用来保证变更表结构操作的安全性。因此，如果有两个线程要同时给一个表加字段，其中一个要等另一个执行完才能开始执行。

行锁：引擎实现，InnoDB支持。

事务A要更新一行数据，事务B也要更新同一行，则事务B必须等A的操作完成后才能更新

1，两阶段锁协议：在InnoDB引擎中，行锁是在需要的时候才加上的，如果有行锁，在事务结束后才会释放；（启发：如果事务中要锁多行数据，要把最可能造成冲突、最可能影响性能、影响并发的的锁尽量往后放）

2，死锁：并发系统中不同线程出现循环资源依赖，涉及的线程都在等待别的线程释放资源，导致进入无限等待的状态，出现死锁



解决死锁策略：

1. 进入等待，直到超时。超时时间：innodb\_lock\_wait\_timeout 设置
2. 死锁检测：发现死锁后，主动回滚死锁链中的某一个事务，让其他事务继续执行，参数 innodb\_deadlock\_detect 设置为 on，表示开启此逻辑

问题：可能会导致性能问题，因为要消耗大量CPU资源：每个被堵住的线程要判断是不是自己导致了死锁，O(n)的算法，也就是并发1000个线程更新同一行，那么死锁检测就达到100万的数量级

如何解决：

1. 如果确保业务不会发生死锁，临时关闭死锁检测；但有可能出现大量超时
2. 控制并发量

避免死锁：

1. 在编程中尽量用固定的顺序处理数据库记录
2. 在允许幻读和不可重复读的情况下，尽量使用RC事务隔离级别，可以避免gap lock导致的死锁问题
3. 更新表时，尽量使用主键更新
4. 避免长事务
5. 设置超时时间 innodb\_lock\_wait\_timeout

间隙锁（在可重复读级别）

Select \*from t where XX = 22 for update

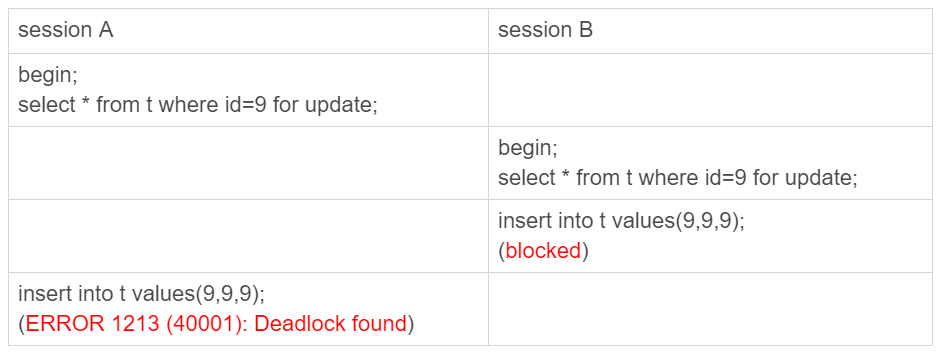
以上语句，在一行行扫描的过程中，不仅将给行加上行锁，还给行两边的空隙加锁。

与间隙锁冲突的是，往间隙中插入数据的操作。间隙锁之间不存在冲突。

**间隙锁和行锁**合称**next-key lock**（解决幻读问题）

间隙锁的引入的缺点：

1. 造成死锁。



2，可能会导致同样的语句锁住更大的范围：影响并发度

当隔离级别不是可重复读，则间隙锁就不会生效。降低隔离级别为读提交，没有间隙锁，但需要解决出现的数据与日志不一样的问题，此时需要把binlog格式设置为row。这也是很多公司使用的配置组合。

排他锁：select XX for update

**集群/分库/读写/主从**

面对海量数据，单表操作其性能出现明显下降；单库中的数据库连接数、磁盘IO以及网络吞吐等资源有限，也会影响性能

分库分表：

垂直拆分：常常根据不用业务进行使用不用的数据库；或者主表和关联表

水平拆分：庞大的数据量

**性能**

Mysql抖动（SQL变慢）

InnoDB在处理更新语句时，只做了一个写日志redo log（磁盘）的操作，将内存里的数据写入磁盘中的过程叫，flush

脏页：当内存数据页和磁盘数据页内容不一致的时候，称此内存页为脏页

干净页：当内存数据写到磁盘后，内存与磁盘数据一致，成为干净页

抖动原因

1. 刷脏页（flush）
2. redo log 写满了，系统会停止所有更新操作，将checkpoint往前推进，redo log留出空间可以继续写（redo log 可以理解为环形），checkpoint推进的部分就是flush到磁盘上的信息（影响性能，InnoDB应尽量避免，此时系统会停止更新）
3. 系统内存不足。当需要一些新的数据页，而内存页不够的时候，就要淘汰掉一些数据页。如果淘汰掉的是脏页，那么还要进行flush（应该控制脏页比例，如果脏页写满或者太多，会影响性能）
4. MySQL自认为空闲的时候，自动flush
5. MySQL正常关闭，会将内存中的脏页flush，这样下次启动的时候，就可以直接从磁盘读数据，启动速度会很快

刷脏页控制策略：

Innodb\_io\_capacity 会告诉我们InnoDB的磁盘能力

1. 刷相邻数据页（脏页）

Innodb\_flush\_neighbors

0：不找邻居，只刷自己的脏页

1：如果相邻数据页是脏页，则连着“邻居”一起刷，并且回继续蔓延

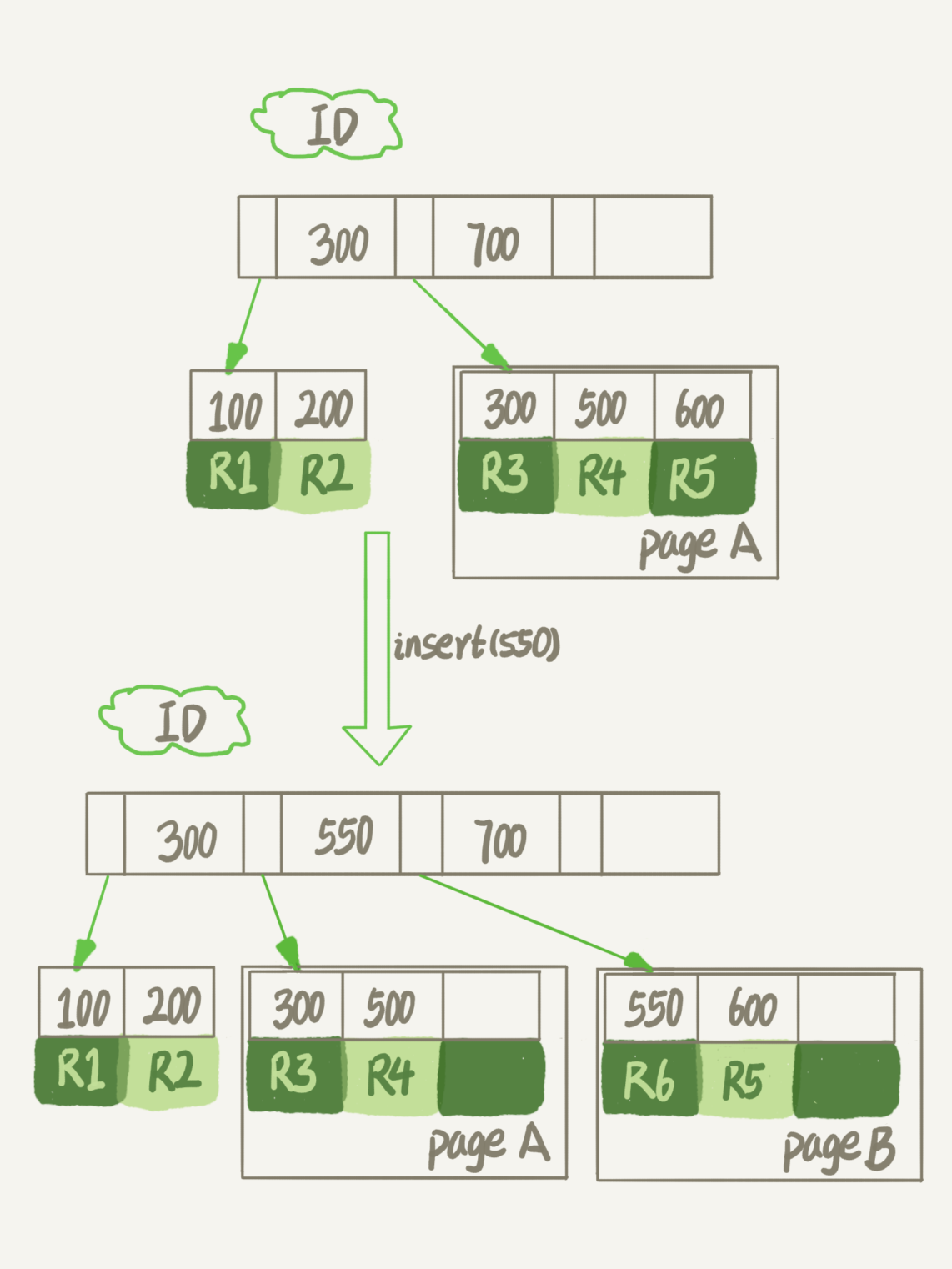
1. 一个高配机器，但是redo log太小

每次事务提交都是写redo log，设置太小，很快被写满，现象就是，磁盘压力小，但是数据库回出现间接性的性能下降

删除数据表文件大小不变

删除一条数据或一个数据页的数据，则被删除的空间会被复用

用delete命令其实就是把记录的位置、或者数据页标记为“可复用”，delete操作不能回收表空间，这会造成数据“空洞”，随机插入也会造成，如下图；收缩表使用alter table t engine = InnoDB



Count性能

引擎：

1. MyIsam，把表的总行数存在磁盘上，count(\*)会直接返回个数，效率很高
2. InnoDB，需要都磁盘，然后累计给出数据

注：如果where后面有条件，MyIsam也不会那么快

InnoDB不存行个数的原因：版本并发控制MVCC。由于不同会话中，对同一个表的count个数不一样。

利用表存行数（利用MySQL事务）

Count用法：

1. count(主键) ：InnoDB 引擎会遍历整张表，把每行的Id取出来，返回给server层；server层判断不为空后，就按行累加
2. count(1) ：遍历整张表，但不取值。判断不为空，按行累加。很明显count(1)比count(id)快，因为从引擎返回id会涉及解析数据行，以及拷贝字段值等操作
3. count(字段)：要判断这个字段是否是null，一行行取出，不为空则累加；
4. count(\*)：例外。不会取出所有字段，专门做了优化，不取值。count(\*)按行累加

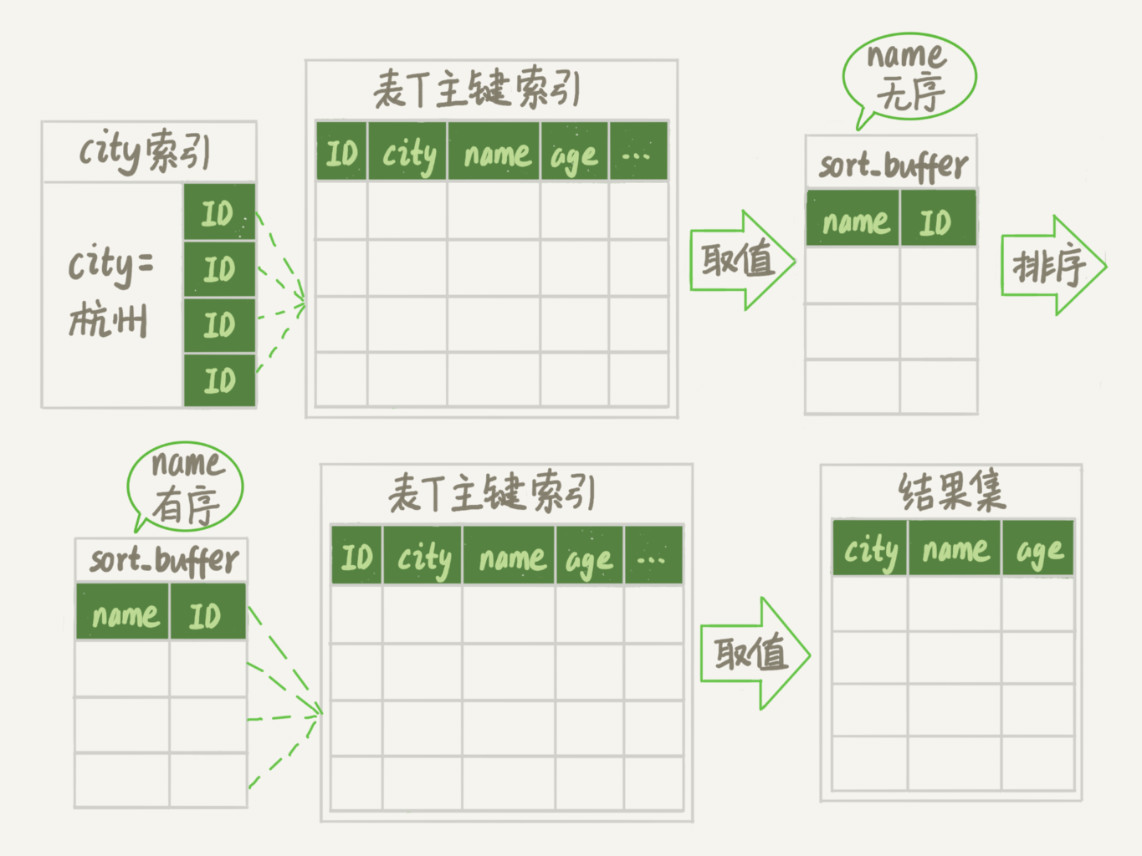
效率：4>=2>1>3

order by 排序

当explain查询语句，extra里含有using filesort，表示需要排序，MySQL会给每个线程分配一个内存块用于排序，称为sort\_buffer

排序可能在内存中排序，也可能需要使用外部排序，这取决于排序所需的内存参数sort\_buffer\_size（MySQL为排序开辟内存sort\_buffer大小）。如果排序内存不够，则需要利用磁盘文件辅助排序

max\_length\_for\_sort\_data 控制用于排序的行数据的参数。如果单行数据超过这个值，mysql就要换一个算法。（现在sort\_buffer中排序，然后通过id再回表查询行数据返回，增加查询的时间，降低效率），这个过程叫 rowid 排序，如下图



在索引上（联合索引）排序，可以防止file\_sort

order by rand()：随机排序

通过explain分析，extra字段显示 using temporary，表示需要临时表；using filesort，表示需要执行排序操作

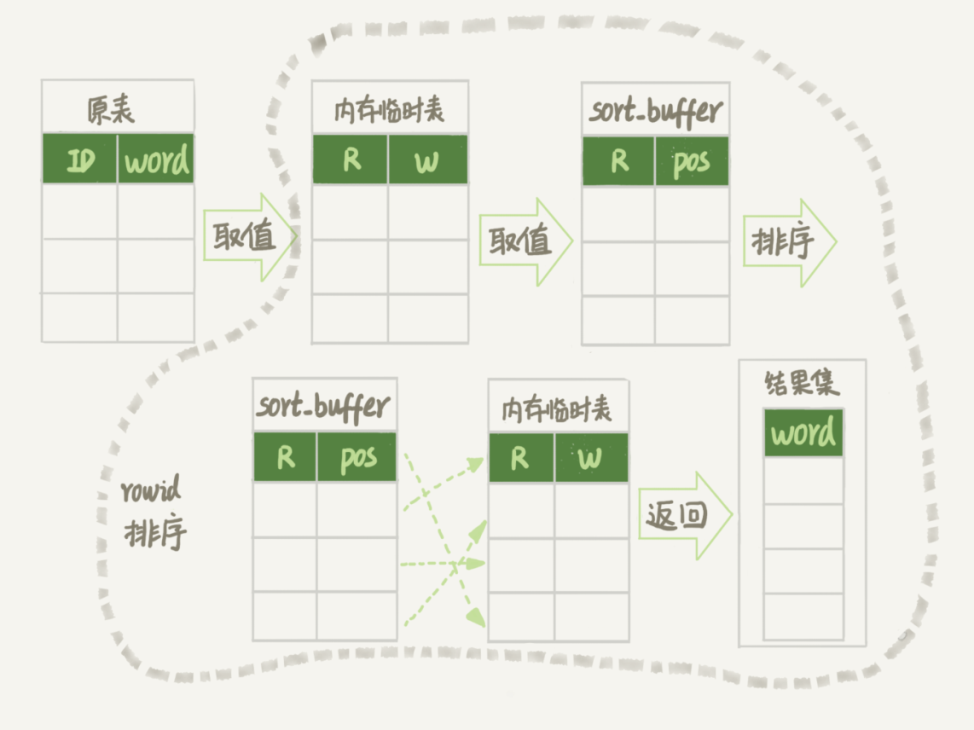
执行过程：

1. 创建临时表（没有索引），使用memory引擎。有两个字段，一个double类型，一个varchar类型
2. 从表中按照主键顺序去取所有的数据，通过rand()生成大于0小于1的随机小数，把这个随机数和数据 ，存在临时表中。随机数对应double
3. 初始化sort\_buffer
4. 从内存临时表中读取随机数和位置信息，存入sort\_buffer中（又扫描一遍）
5. 根据随机数排序
6. 排序完成后取前几个数据

注1：内存表临时排序使用的是rowid排序。

有主键的InnoDB表，rowid就是主键；

没有主键的InnoDB，系统会生成长度为6字节的rowid



注2：不是所有的临时表，都是内存，参数tmp\_table\_size限制了内存临时表的内存大小，默认是16M。如果临时表大小超过此值，那么内存临时表会转换为磁盘临时表。

磁盘临时表使用的引擎是InnoDB，由参数internal\_tmp\_disk\_storage\_engine控制。

注3：优先队列排序算法

查询一行语句，执行特别慢

第一类：查询长时间不返回，如 select \* from t where id = 2

通常情况下：表t被锁

等MDL锁（version5.7之后不复现（修改了MDL加锁策略））

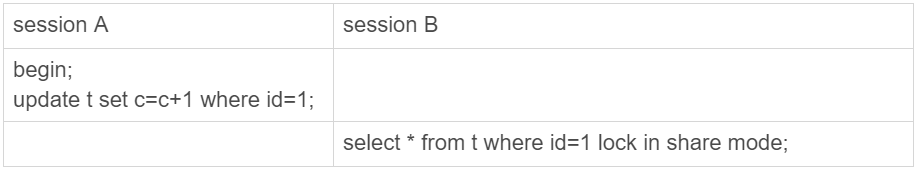
查询 sys.schema\_table\_lock\_waits 表，可以找到造成阻塞的processid，然后kill就好。

等flush

一般flush很快，原因很可能是flush被其他命令堵住了



等行锁



第二类，查询慢

慢sql原因

1. 无索引、索引失效
2. 锁等待

例如：InnoDB中行锁升级为表锁，行锁是基于索引加的锁，所以如果我们在做更新操作时，如果索引失效，那么行锁会升级为表锁；死锁；表锁（MyIsam）

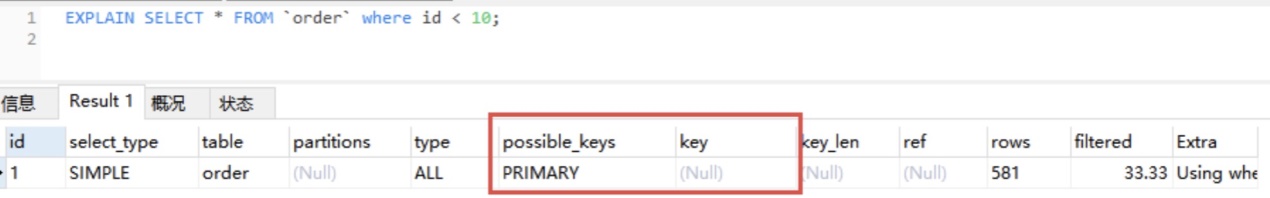
1. 不恰当的sql语句

Select \* ，在大数据表中会用limit M,N，对非索引字段排序

优化SQL

分析阶段：

1. 通过explain分析SQL执行计划



select\_type：simple（普通查询）、primary（主查询）、union（union后面的查询）、subquery（子查询）等

table：当前执行计划的表，当前表名或自定义别名

partitions：访问的区分表信息

type：表示从表中查询到行所执行的方式，结果值从好到坏依次是 system>const>eq\_ref>ref>range>index>all

system/const：表中只有一行数据匹配

eq\_ref：使用唯一索引

ref：非唯一索引扫描

range：扫因范围扫描

index：索引全表扫描

All：全表扫描

possible\_keys：可能使用到的索引

key：实际使用到的索引

key\_len：当前使用的索引的长度

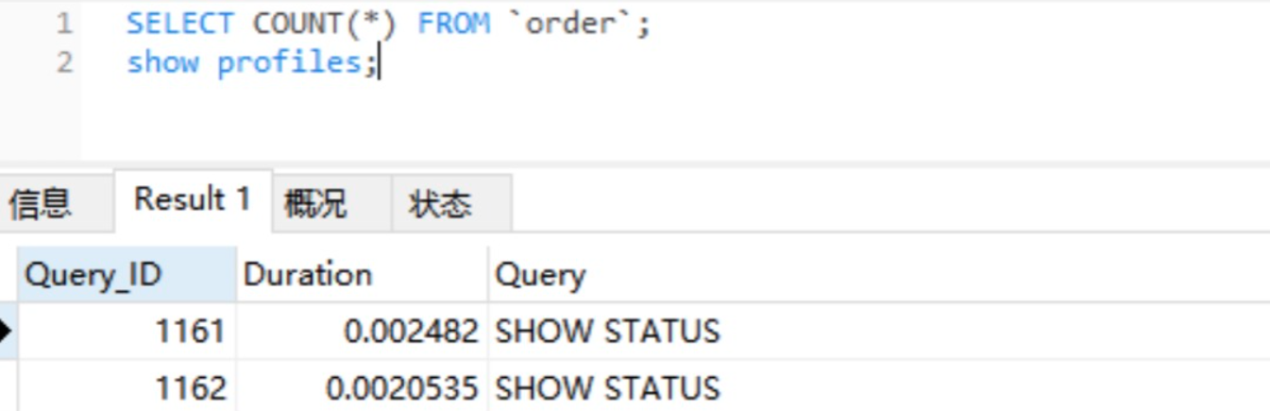
ref：关联id等信息

rows：扫面的行数

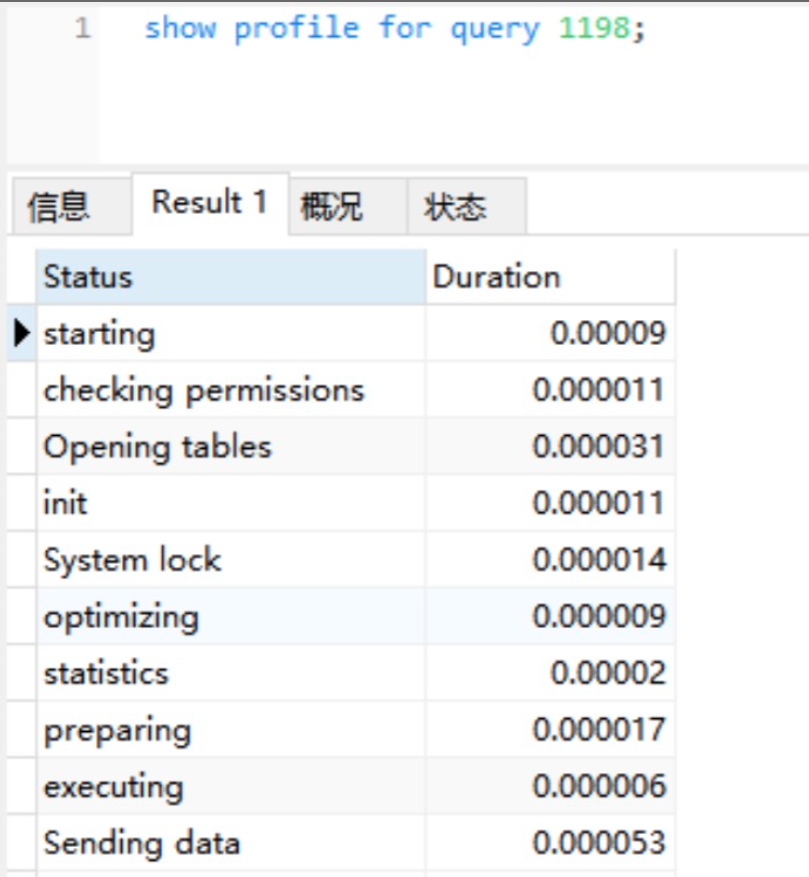
filtered：查找到所需记录占总扫描数的比例

extra：额外的信息

1. 通过show profile 分析SQL执行性能
2. Show Profiles 查询最近执行的记录，记录个数由profiling\_history\_size参数决定，最大100



1. 获取query\_id后，show profile for query id查询消耗时间



常用SQL优化

1. 优化分页查询
   1. 利用子查询优化分页查询

select \* from `demo`.`order` where id> (select id from `demo`.`order` order by order\_no limit 10000, 1) limit 20;

1. 优化count(\*) 看版本
   1. 使用近似值
   2. 通过单独创建表统计行数
2. 优化select \*

IO瓶颈

1. 将sync\_binlog设置为大于1的整数（比较常见的是100~1000），减少写磁盘频率。风险：主机崩溃断电后会丢失binlog日志
2. 将innodb\_flush\_log\_at\_trx\_commit设置成2，每次事务完成提交到page cache中。风险：主机崩溃断电会丢失部分数据
3. 设置binlog\_group\_commit\_sync\_delay 和 binlog\_group\_commit\_sync\_no\_delay\_count 参数（组提交知识点），减少 binlog 的写盘次数。可能会增加语句的响应时间，但没有丢失数据的风险。

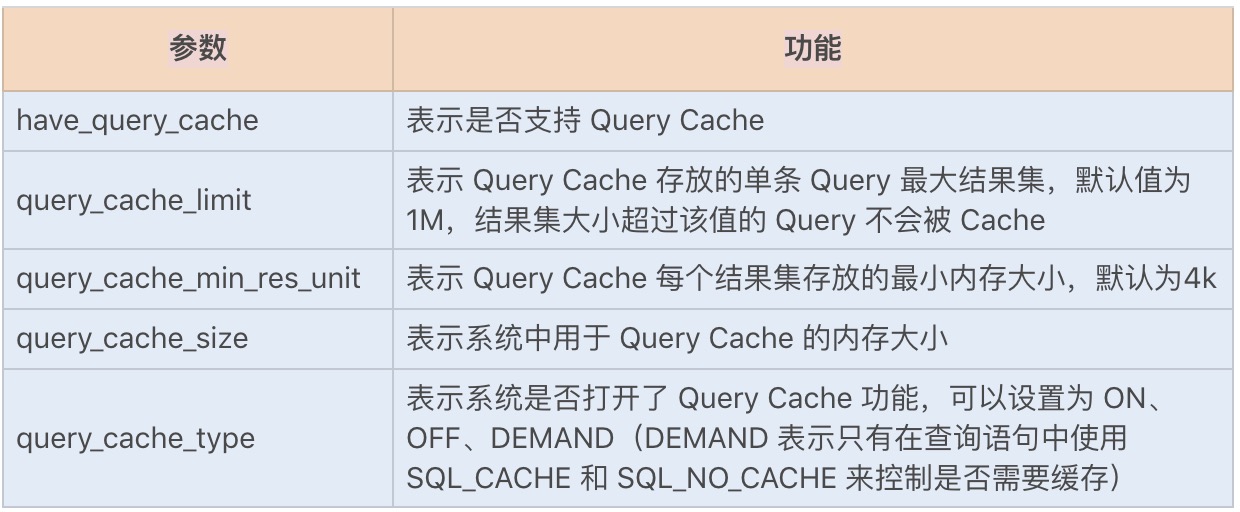
实际应用

MySQL执行顺序：

1. from 、join：笛卡尔生成临时表temp1
2. on 通过条件筛选生成临时表temp2
3. where 对于temp2过滤生成temp3
4. group by 生成temp4（分组）
5. having 分组数据进行聚合生成temp5
6. select 选择指定的列，生成temp6
7. distinct 去重，生成temp7
8. order by 排序，生成temp8
9. limit 截取

参数调优

1. 内存调优



查询缓存（高版本去掉）可以增加查询速度；若删除修改大量反而降低性能

1. InnoDB存储引擎参数设置调优

Innodb\_buffer\_pool\_size，默认128M；越大性能越好，但是过大会造成频繁的SWAP，推荐为服务器物理内存的80%

Innodb\_buffer\_pool\_instances：IBP缓冲池中被划分多个实例；只有ibp设置为1G或更大时才会生效

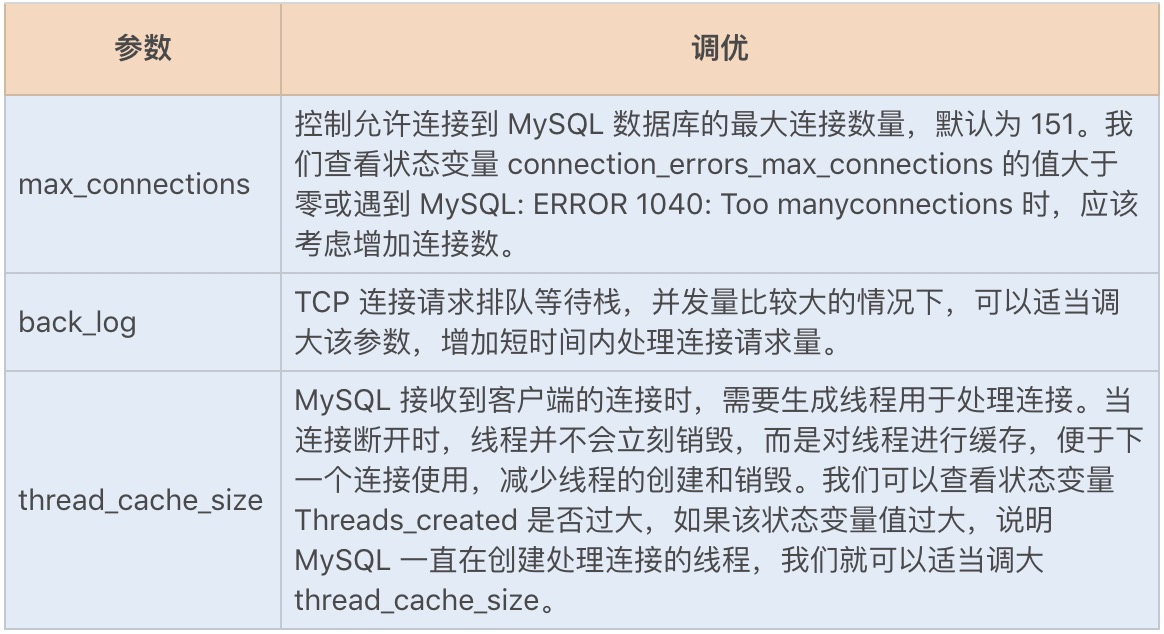
Innodb\_read\_io\_threads/ Innodb\_write\_io\_threads：默认各为4；如果读大于写，可以读多写少；反之写多读少

Innodb\_log\_file\_size：理论上来说，此参数越大，刷脏页的频率就越少，减少IO，增加系统性能；但如果太大，则会导致重启恢复时间过长；一般1G就够了

Innodb\_log\_buffer\_size：日志缓冲池大小，默认8M；高并发中，参数过小会增加磁盘IO操作

Innodb\_flush\_logat\_trx\_commit：控制重做日志从缓存写入文件刷新到磁盘中的策略，默认1，表示每次事务的redolog都会直接持久化到磁盘中，这样保证不丢失数据；

1. 其他



LRU：

Mysql在内存中维护一个链表，通过LRU算法（最近最少使用）淘汰非热点数据

MySQL新增了一个midpoint insertion startegy策略，就是默认情况下，读取到的新页并不是直接放入的LRU列表的首部，而是LRU列表长度的5/8处，目的是为了避免由于一些不常查询SQL偶尔一次查询就把之前热点数据淘汰的情况。