Java超神之路-MySQL

一、基础架构和日志相关

1、一个查询语句是怎么执行的?
2、一条更新语句是怎么执行的?
3、redolog和binlog的区别?
4、为什么日志需要“两阶段提交”?
5、MySQL可以恢复到半个月内任意一秒的状态，怎么做的?
6、什么是buffer pool?
7、buffer pool里包含什么东西?
8、buffer pool中数据页那么多，他怎么知道哪些是空闲的?
9、怎么知道我此次请求对应的数据已经在buffer pool里了?
10、buffer pool中数据页那么多，他怎么知道哪个页是脏页要刷到磁盘上?
11、buffer pool满了的话，需要将部分页刷到磁盘，刷哪些?
    11.1、淘汰哪些数据?
    11.2、说说buffer pool里的lru链表的工作原理
    11.3、lru链表、flush链表什么时候刷盘?
12、什么是脏页?
13、MySQL为什么要有预读功能?
14、buffer pool优化
15、redo log长什么样?
16、undo log长什么样?
17、生产环境下的一次性能抖动

二、存储相关

1、为什么MySQL把一行行数据存储到磁盘的时候要采取这种格式？直接像Java里面的序列化一样把很多行的数据做成一个大对象然后序列化一下写到磁盘文件里，取出来的时候直接反序列化下不香嘛?
2、变长字段列表里的十六进制值为什么要逆序存储?
3、为什么数据里的null值不直接存储为null，要搞个bitmap出来?
4、什么是行溢出?

三、存储引擎

1、MySQL支持哪些存储引擎?
2、Innodb和MyIsam的区别?
3、为什么MyIsam读效率高于Innodb?

四、事务

1、事务的四大特性
2、事务四种隔离级别
3、什么是脏写、脏读、不可重复读、幻读
4、什么是mvcc？原理是什么?
    4.1、什么是undo log版本链
    4.2、什么是read view
5、RC（Read Committed，读提交）如何实现的
6、RR（Repeatable Read，可重复读）如何实现的
7、RR能解决幻读吗?

五、锁

1、锁都有哪几种？什么意思?
2、说说事务隔离级别与锁的关系
3、什么是死锁
4、怎么解决死锁?
5、死锁出现的案列

六、 索引

1、列举一些导致索引失效的场景
2、索引不适合哪些场景?
3、为什么用B+树而不是B树或二叉树或平衡二叉树?
4、聚簇索引与非聚簇索引的区别
5、如何写sql能够有效的使用到复合索引?
6、Hash索引和B+树区别是什么？你在设计索引是怎么抉择的?

7、索引有哪几种类型?
8、创建索引有什么原则呢?
9、什么是最左前缀原则?什么是最左匹配原则?
10、覆盖索引、回表等这些，了解过吗?
11、使用索引查询一定能提高查询的性能吗?为什么?
12、count(1)、count(*) 与 count(列名) 的区别?
13、列值为NULL时，查询是否会用到索引?
14、非聚簇索引的查询都需要回表吗?
15、什么是索引下推?
16、频繁更新索引字段可以吗?为什么?
17、为什么group by 和 order by会使查询变慢
18、怎么优化order by?
19、怎么优化group by?
20、从innodb的索引结构分析，为什么索引的key长度不能太长?

七、主从复制和读写分离

1、主从复制有什么好处?解决了哪些问题?
2、MySQL主从复制原理的是啥?
3、主从复制半同步复制（semi-sync）什么意思?
4、哪些场景可能造成主从延迟?
5、如何解决MySQL主从同步的延时问题?
6、并行复制是什么意思?
7、介绍下主从复制GTID的方式?
8、如何解决互为主备后的循环复制问题?
9、为什么要做读写分离?
10、如何实现MySQL的读写分离?

八、MySQL分库分表

1、为什么要分库分表?
2、分库分表的技术有哪些?
3、分库分表的拆分方式有?他们分别主要解决什么问题?
4、单张大表要进行分库分表怎么办?
5、分库分表有哪些算法?
6、分库分表后面临的问题

九、生产环境实战案例

1、生产环境下的一次性能抖动
2、死锁出现的案列
3、字符串单引号
4、name字段utf8传表情
5、社交app用户信息搜索
6、上千万级别的评论系统深度分页
7、亿级用户表统计流失人数的SQL太慢
8、大事务影响性能
9、强制使用索引

十、分散型题目

1、为什么表数据删掉一半，表文件大小不变?
2、为什么我只查一行的语句，也执行这么慢?
3、怎么最快地复制一张表?
4、MySQL数据库cpu飙升到500%的话他怎么处理?
5、如果某个表有近千万数据，CRUD比较慢，如何优化?
6、数据库自增主键可能遇到什么问题?
7、百万级别或以上的数据，你是如何删除的?
8、关心过业务系统里面的sql耗时吗?统计过慢查询吗?对慢查询都怎么优化过?
9、你进行SQL优化的时候一般步骤是什么?
10、深分页很慢，怎么解决的呢?
11、主键自增ID还是UUID之间选择哪个?为什么?
12、如何排查项目中可优化的SQL?

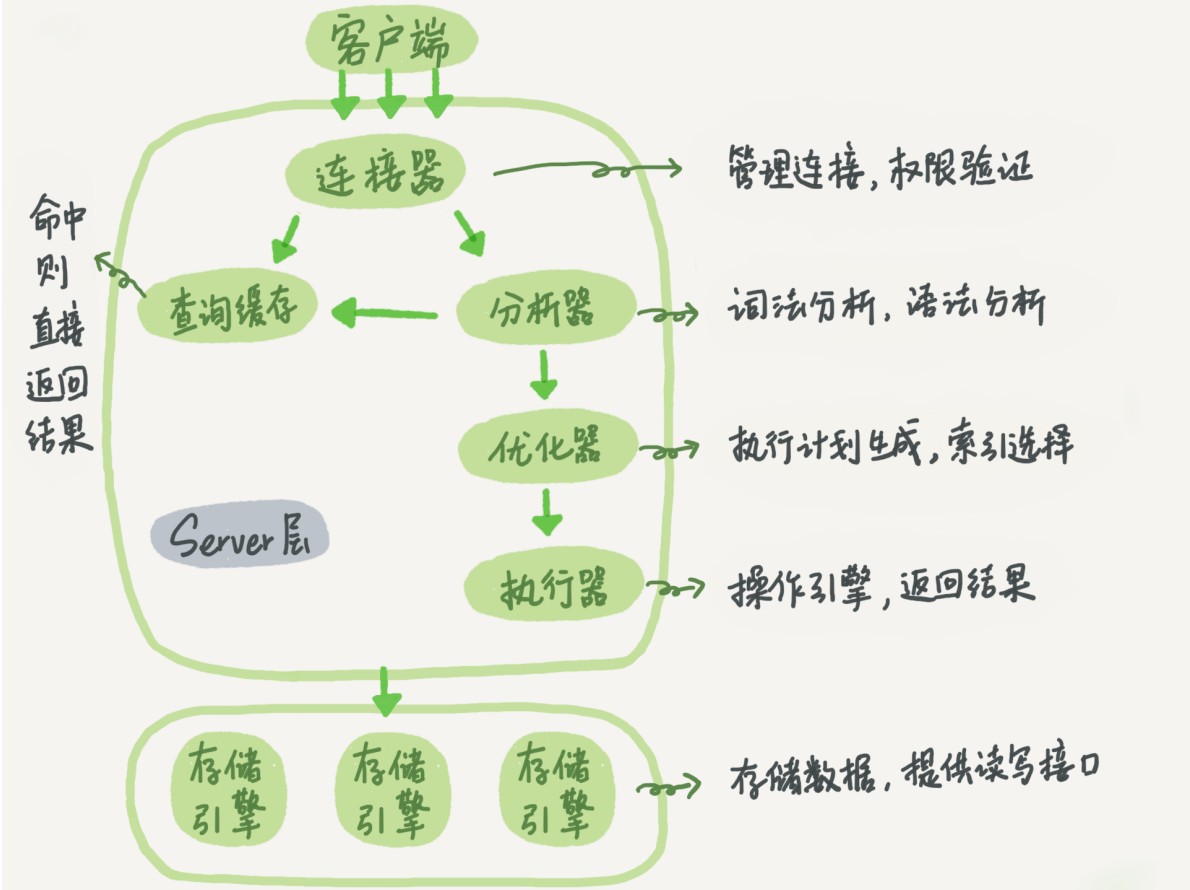
Java超神之路-MySQL

> author：编程界的小学生
>
> date：2021/01/29

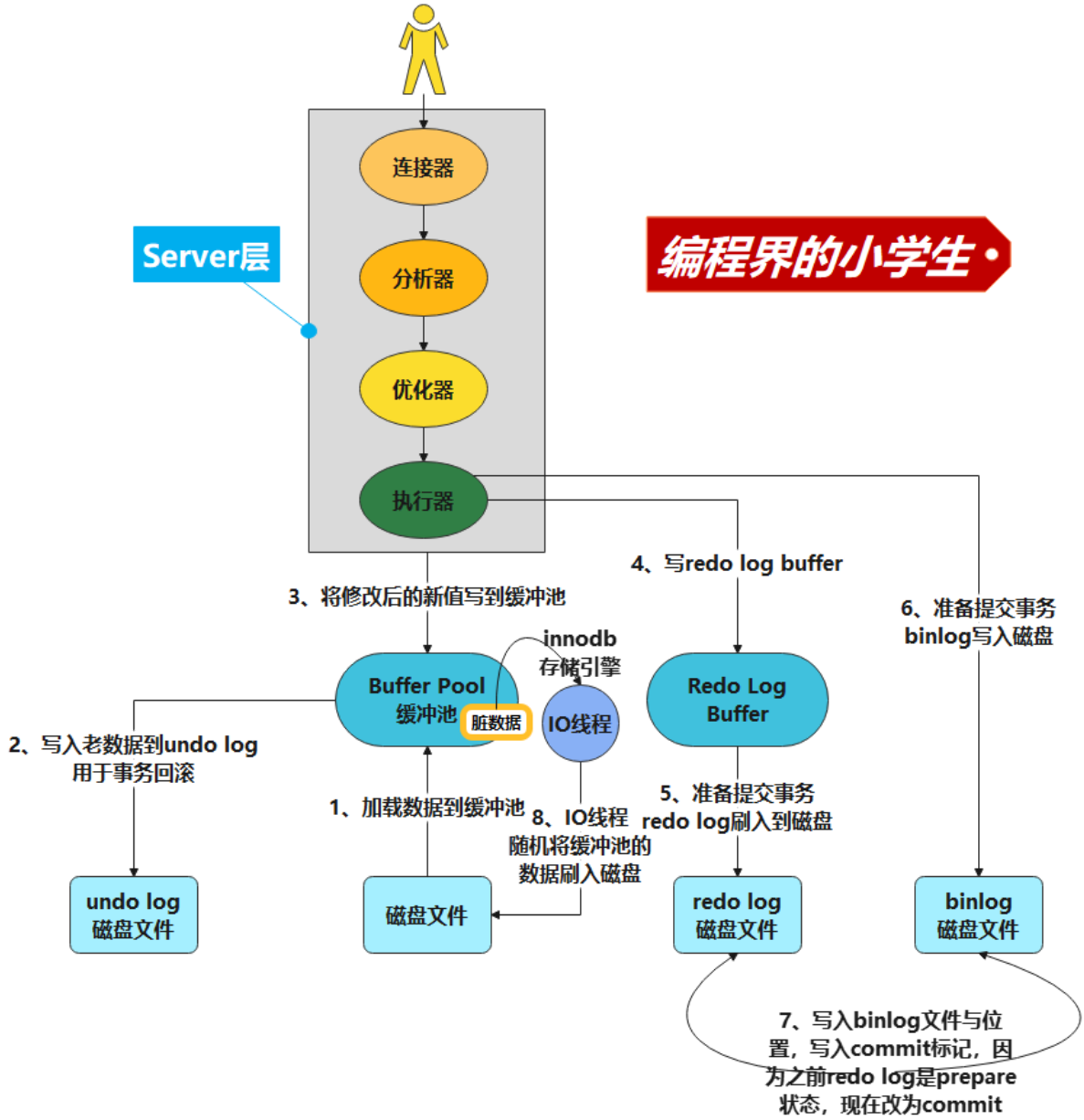
一、基础架构和日志相关

1、一个查询语句是怎么执行的?

> 查询缓存弊大于利，因为更新操作会让缓存失效，所以MySQL8.0将此部分彻底移除了。



2、一条更新语句是怎么执行的?



3、redolog和binlog的区别?

- redo log 是 InnoDB 引擎特有的，binlog 是 MySQL 的 Server 层实现的，所有引擎都可以使用。
- redo log 是物理日志，记录的是“在某个数据页上做了什么修改”，binlog 是逻辑日志，记录的是这个语句的原始逻辑，比如“给 ID=2 这一行的 c 字段加 1 ”
- redo log 是循环写的，空间固定会用完，binlog 是可以追加写入的。“追加写”是指 binlog 文件写到一定大小后会切换到下一个，并不会覆盖以前的日志。

4、为什么日志需要“两阶段提交”?

什么是"两阶段提交？"

写入redolog的状态处于prepare阶段，然后写binlog，写完binlog后提交事务，并将redolog改为commit状态。

为什么要"两阶段提交？"

防止写完redolog成功，还未写入binlog的时候宕机了的情况，这样重启后redolog里的数据还存在，可以恢复，但是从库都是拉的binlog，binlog里却丢失数据了。造成主从不一致，所以需要"两阶段提交"。

5、MySQL可以恢复到半个月内任意一秒的状态，怎么做的?

首先dba会定期全库备份的，当需要恢复到指定的某一秒时，比如某天下午两点发现中午十二点有一次误删表，需要找回数据，那你可以这么做：

- 首先，找到最近的一次全量备份，从这个备份恢复到临时库
- 然后，从备份的时间点开始，将备份的binlog依次取出来，重放到中午误删表之前的那个时刻。
- 这样临时库就跟误删之前的线上库一样了，然后可以把表数据从临时库取出来，按需要恢复到线上库去。

6、什么是buffer pool?

增删改操作首先就要对内存的Buffer Pool里的数据执行相应增删改操作，因为直接写盘太慢了，所以有了Buffer Pool。他就是数据库的一个内存组件，里面缓存了磁盘的真实数据，然后我们增删改的时候直接操作Buffer Pool，最后日志文件都写完且commit之后会由线程随机将Buffer Pool里的数据刷到磁盘上。

7、buffer pool里包含什么东西?

buffer pool里存的是缓存页（从数据页（磁盘上的）加载进来的）和描述数据。一页16kb，数据页里是一行行数据，描述数据包含比如所属的表空间、数据页的编号、这个缓存页在buffer pool中的地址等描述信息。

8、buffer pool中数据页那么多，他怎么知道哪些是空闲的?

> 换种问法：请求进来，发现buffer pool中没有当前请求对应的数据页缓存，buffer pool又那么大，里面无数个数据页，他怎么知道我此次请求的数据放到哪个数据页上？如果放的是一个已存在数据的数据页，那不覆盖了吗?

buffer pool里维护了一份free链表，free链表是个双向链表，里面放的是描述数据，描述数据里面包含页的位置，free链表里的节点都是空闲可用的。所以请求进来后将free链表里最后的node节点移除就好了，就代表此页已经被占用。等释放页的时候在加回free链表中。

9、怎么知道我此次请求对应的数据已经在buffer pool里了?

> 如果没在，那么从磁盘上拿走放到缓冲区里缓存，但是他怎么知道我此次请求的数据有没有对应的数据页在缓存区里了呢?

数据库有一个哈希表的数据结构，key是表空间号+数据页号，value是缓存页的地址。每次读取一个数据页到buffer pool之后，都会在这个 哈希表中写入一个key-value，下次再用这个数据页的时候就可以先从哈希表中直接读取出来他是不是已经放到buffer pool里了。

10、buffer pool中数据页那么多，他怎么知道哪个页是脏页要刷到磁盘上?

buffer pool里维护了一份flush链表，只要是被修改过的数据都会将其对应的描述数据加入到flush双向链表中，flush链表里的数据都是脏页，都是要被刷到磁盘上去的。刷到磁盘上后将此块描述数据从flush链表中移除，加入到free链表中，让其重新可用 。

11、buffer pool满了的话，需要将部分页刷到磁盘，刷哪些?

> 如果Buffer Pool中的缓存页不够了怎么办？也就是说free链表的空闲页不够了怎么办?

那肯定要淘汰一部分数据，然后释放缓存页，加入到free链表中，使其成为空闲页。buffer pool里维护了一份lru双向链表，用于淘汰数据。

11.1、淘汰哪些数据?

淘汰哪些缓存命中率很低的数据，也就是不常用的数据页，比如在100次请求中，数据页A被访问或修改了50次，数据页B只被访问或修改了1次，那么没有空闲缓存页的时候肯定是会淘汰B的，让其重新空闲可用。

11.2、说说buffer pool里的lru链表的工作原理

首先lru链表分为冷热数据两部分区域。这样做也是为了把性能做到极致。叼的很。

从磁盘加载一个数据页到缓存页的时候，就把这个缓存页的描述数据块放到LRU链表冷数据区头部去，然后在1s之后你修改冷数据区的数据页后才会被挪动到热数据区链表的头部。（根据MySQL的`innodb_old_blocks_times`参数（默认1s）），若修改的是lru链表热数据区的缓存页，则需要判断是不是lru链表热数据区前1/4的缓存页被访问，如果是的话，则不动，如果不是的话（也就是访问的 后面的3/4），则给他挪动到lru热数据区链表头部。这样的话尽可能的减少链表中的节点移动了（因为热数据区操作本来就很频繁），然后若buffer pool的缓存页不够了，则就把lru冷数据区尾部的节点刷到磁盘上且移除lru链表和flush链表且加入到free链表中去。

> MySQL设计者把lru这部分的性能做到了极致，佩服的五体投地！！！
>
> MySQL设计者把lru这部分的性能做到了极致，佩服的五体投地！！！
>
> MySQL设计者把lru这部分的性能做到了极致，佩服的五体投地！！！

扩展补充

为什么搞得这么复杂，不直接：只要有数据的缓存页他都在lru里了，而且最近被修改过的数据缓存页他都会挪到lru链表头部去，当空间不够的时候就把lru尾部的节点刷到磁盘上且移除lru链表和flush链表且加入到free链表中去?

因为MySQL有预读功能。预读就是当你从磁盘上加载一个数据页到缓存页的时候，他可能连带着把这个数据页相邻的其他页也一起加载到缓存中，比如你查了100w条数据，连着写了5个数据页了，MySQL认为你这次操作这么大，第6个页也需要，所以直接提前给你加载到缓存页了。那么这种情况如果按照上面斜体字的逻辑来看就是第6个页永远没人用，他却跑到了lru链表第6个位置（实际没人访问，顺带着被加载进来的），然后清理的时候从lru尾部开始，不合适呀！所以lru链表有了冷热数据两部分。不得不说MySQL设计者，真TM牛逼。这个冷热让我莫名其妙的想到了并发大师dog 李的读写锁设计，一个int 高低位代表读写两种不同的锁，都是神人!

11.3、lru链表、flush链表什么时候刷盘?

LRU链表

- buffer pool没空闲缓存页的时候
- 有个后台线程每隔一段时间就会把lru冷数据区的尾部一些缓存页刷到磁盘，然后移除lru链表和flush链表且加入到free链表

flush链表

- buffer pool没空闲缓存页的时候从lru冷数据区移除数据且移除flush链表的数据

- 有个后台线程每隔一段时间就会把flush链表中的缓存页都刷到磁盘，然后移除lru链表和flush链表且加入到free链表

总结

后台线程不停的把flush链表和lru链表冷数据区的缓存页刷到磁盘上去，然后free链表中的缓存页不停的增加。这是个动态过程，并非要free不够了才刷盘释放缓存页。

12、什么是脏页?

flush链表里的缓存页都是脏页，也就是说凡是被修改过的且还没刷到磁盘上的缓存页都属于脏页。

13、MySQL为什么要有预读功能?

预读就是当你从磁盘上加载一个数据页到缓存页的时候，他可能连带着把这个数据页相邻的其他页页一起加载到缓存中，比如你查了100w条数据，连着写了5个数据页了，MySQL认为你这次操作这么大，第6个页也需要，所以直接提前给你加载到缓存页了。提前预读进行缓存。在某些场景下可以达到性能优化的效果。

14、buffer pool优化

增大buffer pool内存，然后设置多个buffer pool，比如 buffer pool设置8GB，实例个数设置为4，相当于每个buffer pool 2GB左右。因为buffer pool默认是1个，操作数据页的时候会上锁，所以可以增加buffer pool个数。

一般建议buffer pool大小设置为MySQL内存的50%~60%。

15、redo log长什么样?

- 表空间ID
- 数据页号
- 数据页中的偏移量
- 修改数据长度
- 具体修改的数据

16、undo log长什么样?

- undo log的开始位置
- 主键的各列长度和值
- 表的id
- undo log日志编号
- undo log日志类型
- undo log的结束位置

17、生产环境下的一次性能抖动

背景：线上运行的服务，出现了一次性能抖动。导致了告警。

经查询发现是如下原因：

平时只需要几百毫秒的语句这一下执行了几秒钟了，经查询发现某个SQL需要查询大量数据（不属于慢SQL的范畴），然后需要将大量的数据缓存到缓存页中去，此时就可能导致内存里大量的脏页需要淘汰出去刷到磁盘上，要不然没空间缓存这批查询的数据到buffer pool中。所以大量的刷盘操作是很慢的，导致了抖动。

补充：

还有一种情况也可能造成性能抖动，就是大量的update/insert/delete语句导致redo log buffer快写满的时候，造成大批量的redo log buffer的数据刷盘操作，这时候你再进行update/insert/delete的时候会发现性能极差，比如单表update/insert/delete发现1s都没完成，这时候可以看下是不是大量的redo log在刷盘。

解决办法：

调整`innodb_io_capacity`参数，这个参数是告诉数据库采用多大的IO速率把缓存页flush到磁盘里，先采取压测看下当前磁盘的io速率支持多少然后针对性调整大小。

调整`innodb_flush_neighbors`参数，这个参数是说flush缓存页到磁盘的时候，可能会控制把缓存页临近的其他缓存页也一起刷到磁盘，但是这样有时候会导致刷磁盘的页太多了，因为临近的页也刷到磁盘来了，所以调整成0。

二、存储相关

1、为什么MySQL把一行行数据存储到磁盘的时候要采取这种格式？直接像Java里面的序列化一样把很多行的数据做成一个大对象然后序列化一下写到磁盘文件里，取出来的时候直接反序列化下不香嘛？

原因很多，比如：

（1）因为不需要每次都获取全部数据，只获取其中一两个字段的话，按照现在的序列化方式，就可以计算出字段对应的偏移量来获取。

（2）序列化方式也占用较大空间。MySQL的格式会节省很多空间占用。

2、变长字段列表里的十六进制值为什么要逆序存储？

答：这样可以使记录中位置靠前的字段和它们对应的字段长度信息加载到内存中时，位置距离更近，可能会提高高速缓存的命中率。

> 不得不承认写这些软件的人真他娘的不是人，这种小细节都不放过。

3、为什么数据里的null值不直接存储为null，要搞个bitmap出来？

举个最简单的例子：在UTF-8编码下，一个英文字符是1个字节，"NULL"是四个字节 = 32位。字符串占用的空间是bit位存储占用的空间32倍。

4、什么是行溢出？

一个数据页大小16kb，若遇到varchar(65535)、text等这种大文本类型的，大小远大于16kb了，所以一个数据页放不下，需要多个数据页存储一行数据，缓存到buffer pool的缓存页里的时候也是多个缓存页，这就是行溢出。

三、存储引擎

1、MySQL支持哪些存储引擎？

- Innodb
- MyIsam
- memory

- archive

2、Innodb和MyIsam的区别?

- Innodb支持事务，myisam不支持事务
- innodb支持外键，myisam不支持外键
- innodb支持mvcc，myisam不支持mvcc
- count(*) 时，myisam会很快，因为他保存了一个变量记录总数，直接获取，innodb需要遍历全表进行统计
- innodb支持表锁、行锁、间隙锁等，myisam只支持表锁
- innodb表必须有主键，即使没有的话，innodb也会以rowid做为主键，myisam可以没主键
- innodb按主键大小有序插入，而myisam按顺序插入

3、为什么MyIsam读效率高于Innodb?

- innodb支持事务，所以会有一个mvcc的比较过程。
- 查询的时候，如果走了索引的话，innodb是聚簇索引，会有一个回表过程，即：先去非聚簇索引树中查询数据，找到数据对应的key后，再通过key回表到聚簇索引树找到最终的数据。而myisam直接就是非聚簇索引，查询的时候查到的最后结果不是聚簇索引树的key，而是磁盘地址，所以直接回查询磁盘的完整数据，无需回表。
- innodb支持行锁，在检查锁的时候不仅要检查表锁，还要看行锁。

四、事务

1、事务的四大特性

ACID

- 原子性

> A Atomicity：要么都成功，要么都失败。

- 一致性

> C Consistent：事务开始之前和完成之后的数据都必须保持一致的状态，必须保证数据库的完整性。

- 隔离性

> I Isolation：数据库允许多个并发事务同事对数据进行操作，隔离性保证各个事务相互独立，事务处理时的中间状态对其它事务是不可见的，以此防止出现数据不一致状态。可通过事务隔离级别设置：包括读未提交（Read uncommitted）、读提交（read committed）、可重复读（repeatable read）和串行化（Serializable）。

- 持久性

> D Durable：一个事务处理结束后，其对数据库的修改就是永久性的，即使宕机了，数据也不会丢失。redolog+binlog

2、事务四种隔离级别

- 读未提交

> READ-UNCOMMITTED：一个事务没提交但是他修改的数据却可以被其他事务看到，可能会导致脏读、幻读、不可重复读。

- 读已提交

> READ-COMMITTED：一个事务没提交，那么他修改的数据其他事务看不到，只能读到已经提交的数据。可能导致幻读和不可重复读。

- 可重复读（innodb默认）

> REPEATABLE-READ：MySQL INNODB默认的隔离级别。一个事务内读到的数据永远是一样的，不管其他事务做了什么更改，是否提交，我都不知道，我只活在我自己的事务里。可以阻止脏读和不可重复读和幻读。

- 串行化

> SERIALIZABLE：让所有事务串行化执行，就好比单线程了，一个一个的排队执行，效率低下，可以避免脏读、幻读、不可重复读，因为单线程了嘛。

3、什么是脏写、脏读、不可重复读、幻读

- 脏写

> A和B两个事务都更新同一条数据， 数据原始值为1，A将数据改为2，B将数据改为3，这时候A回滚了，把B写的3也给抹掉了。这就是脏写。
>
> 就是两个事务都更新一个数据，结果其中一个事务回滚了，这波操作把另一个事务刚写入的值也给抹掉了。

- 脏读

> 一个事务中访问到了另外一个事务未提交的数据，读未提交隔离级别会造成脏读。

- 不可重复读

> 同一个事务里，读取同一条记录多次可能读到不一致的结果。比如A和B两个事务，A负责读取两次，B负责修改数据，A第一次读取到1，B给修改成2且提交了。A在读取的时候发现变成了2，同一个事物里读到的结果不一致了。可重复读策略可以避免此类情况。

- 幻读

> 比如范围查询，同一个事物里执行两次 `where id > 10;` 第一次结果是100条记录，但是其他事务在执行第二遍的时候insert了一条，那么回到原事务里再次执行的时候出现了101条记录。

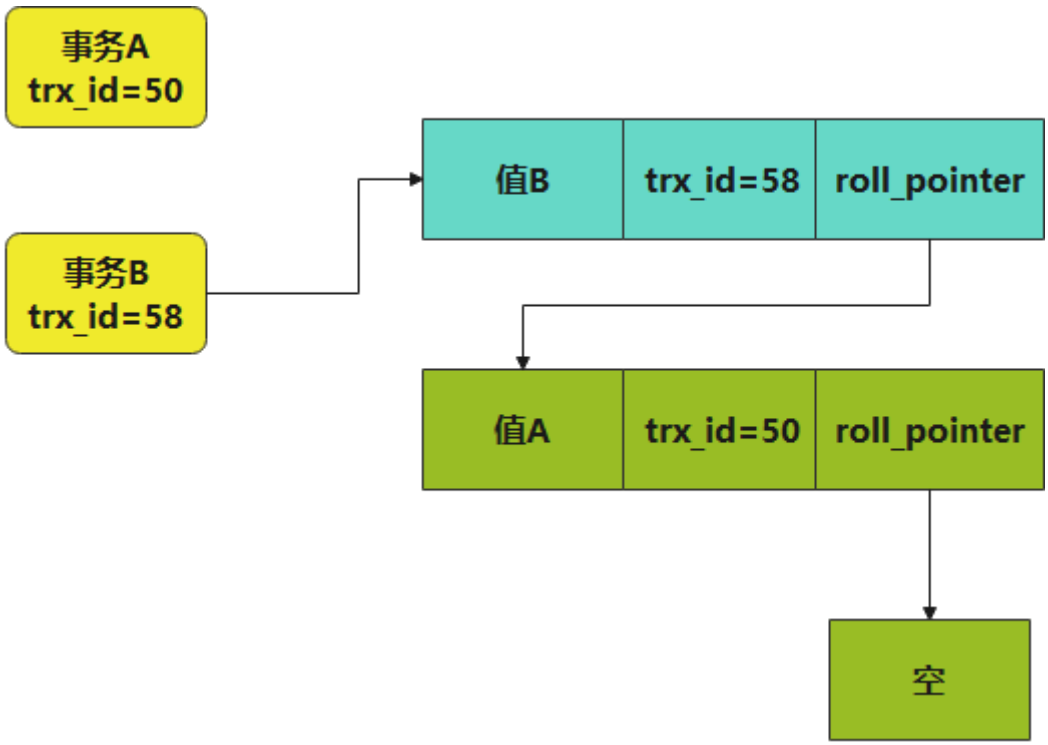
4、什么是mvcc？原理是什么？

Multi-Version Concurrency Control，多版本并发控制。用于实现提交读和可重复读这两种隔离级别。主要原理是undo log版本链+read view。

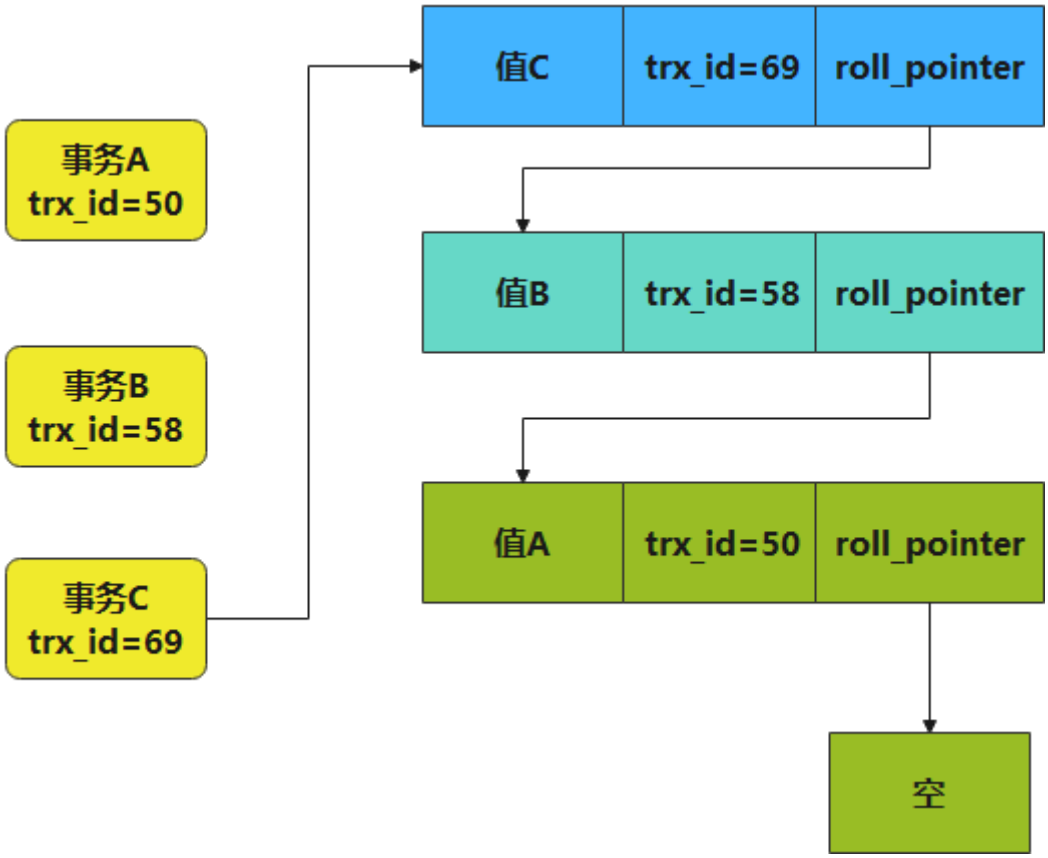
4.1、什么是undo log版本链

每条数据都有两个隐藏的字段：trx_id和roll_pointer，trx_id是最近一次更新这一条数据的事务 id，roll_pointer指向了这条数据之前改动所生成的undo log。

比如：事务A（事务id=50）插入了一条数据，然后事务B（事务id=58）修改了一下这条数据，将值改为了B，那么此时事务B的roll_pointer会指向事务A的undolog，事务A的roll_pointer会指向null（因为他是insert的，之前这条数据没undolog），如下图：



接着事务C（事务id=69）又把这条数据的值改为C，此时如下图：



这种链式就叫redolog链，所以每条数据的redolog都有一条版本链。 总结一句话：redo log版本链就是针对每个事务对同条数据的修改都记录下来形成一个链式结构，链由roll_pointer这个隐藏字段来连接。

4.2、什么是read view

你执行一个事务的时候就会给你生成一个ReadView，ReadView可以理解成快照，里面包含四个关键信息：

- m_ids：**此时此刻**有哪些事务还是未提交的（也就是说和我并发了，我开启事务的时候，还有哪些事务和我一起并发且未提交的）。

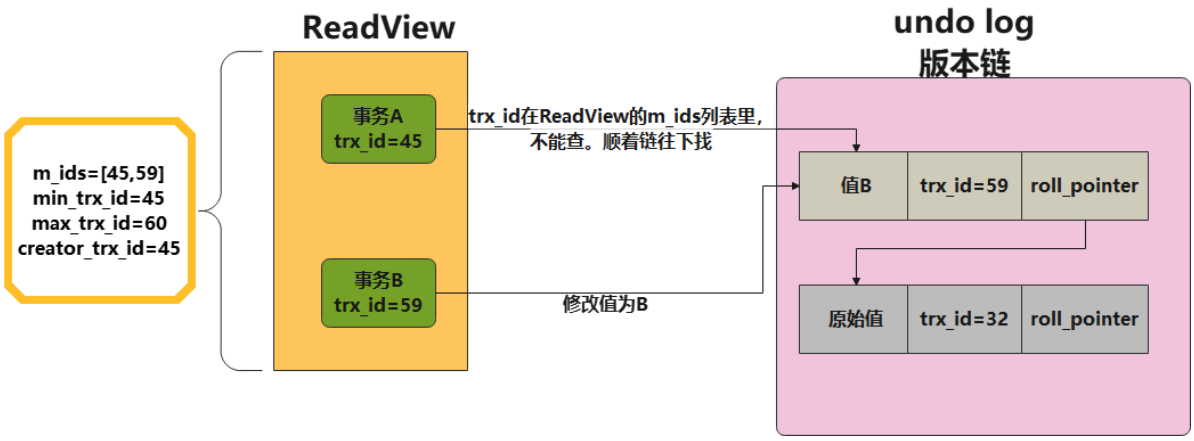
- min_trx_id：m_ids里的最小值。
- max_trx_id：MySQL**下一个**要生成的事务id，比如m_ids里有[3,12]这两个id，那max_trx_id就是13。
- creator_trx_id：本事务id。

原理如下：

去undolog版本链上找，看看trx_id是不是自己或者是不是小于min_trx_id，如果是，则代表这条数据是历史就被修改过的，不是与此次并发的，就可以满足条件，如果大于等于max_trx_id，则代表我此次事务还没执行完，其他事务先我一步把数据更新了，我是快照，我当然不能看到。

举个例子：

假设很早之前事务id=32插入了一条数据且提交了事务。然后现在事务A（trx_id=45）和事务B（trx_id=59），事务A想要读取这行数据，事务B想要更新这行数据，那么事务A和事务B会分别开一个ReadView出来，那么这个事务A的ReadView里的m_ids是[45, 59]，min_trx_id=45，max_trx_id=60，creator_trx_id=45，假设事务B先更新了这行数据然后事务A开始读取，然后事务A查询的时候会发现trx_id大于ReadView的min_trx_id（也就是45），同时小于max_trx_id，说明这行undo log版本链上的数据是被一个跟我并发执行的事务修改的，于是看下trx_id=59是不是在m_ids列表里，结果发现是在的，就确定这行数据是并发执行的，不能读取的，顺着roll_pointer往下继续找undolog日志链，继续走上面的流程，发现trx_id=32这条数据吻合，拿出值。如下图：



如果再来个事务C（trx_id=100），那么事务A查询的时候会看事务C的 trx_id是不是满足条件，发现100比max_trx_id都大了，肯定是我的事务都没执行完，其他事务却先一步update了，这数据肯定不能要，就顺着undolog版本链继续往下找。

所以ReadView结合undo log版本链才会有奇效，所以undo log版本链配合read view是mvcc的核心原理。

5、RC（Read Committed，读提交）如何实现的

主要原理是基于undolog版本链和ReadView来实现。RC是可以造成不可重复读的问题的，所以比如事务A查询两次，事务B在事务A两次查询期间update了一条且commit了，那么事务A第一次查询的值和第二次查询的值是不同的。但是ReadView不是快照吗？不是一个事务一个ReadView吗？怎么还能读到新值？是因为**RC隔离级别不是针对每个事务开启一个ReadView，而是每个数据库操作都开启一个新的ReadView**，所以事务A的两次查询对应两个ReadView，第二个查询的时候对应的ReadView里m_ids已经不包含事务B了，因为他已经提交完了，事务A就认为你是历史数据可以正常读取。

6、RR（Repeatable Read，可重复读）如何实现的

主要原理是基于undolog版本链和ReadView来实现。每个事务开启一个ReadView快照，所以不管你外界怎么提交，我只管我此次事务的快照里的内容，不受外界影响，所以避免了脏读 、不可重复读的问题。

7、RR能解决幻读吗?

能解决。采取的以下两种方式解决的：

- 快照读

> 也就是普通读，比如普通的select，所以是不会加锁的，RR快照读解决幻读的方式采取的是MVCC。

- 当前读

> 指的是加锁的select，比如select...for update，以及insert/delete/update语句，RR当前读解决幻读的方式采取的是next-key locks。

五、锁

1、锁都有哪几种？什么意思?

- 全局锁

> 对整个数据库实例加锁，开启全局锁后整个数据库的更新语句，修改表结构语句都会被阻塞，多用于全库逻辑备份的时候，就需要开启全局锁让整个库处于只读状态，防止数据不一致的情况。

- 表锁

> 对整张表进行加锁，表锁被大部分的mysql引擎支持，MyISAM和InnoDB都支持表级锁。

- 行锁

> 只锁某一行，比如id=3的这行数据，其他的比如id=4的数据不会被锁。InnoDB支持。行级锁分为共享锁 和 排他锁。

- 共享锁

> `select ... lock in share mode;`
>
> 就是多个事务对于同一数据可以共享一把锁，都能访问到数据，但是只能读不能修改。

- 排他锁

> `select ... for update;`
>
> 就是不能与其他锁并存，如一个事务获取了一个数据行的排他锁，其他事务就不能再获取该行的其他锁。包括读也不行。

- 间隙锁

> - **作用于非唯一索引上（当记录不存在的时候，唯一索引中也会产生间隙锁。【主键也是这样，记录不存在会产生间隙锁，否则不会。】）**
> - **可重复读RR级别下才会有间隙锁！如果把事务的隔离级别降级为读提交(Read Committed, RC)，间隙锁则会自动失效。也就是仅仅在RR上生效。**
> - **间隙锁之间互不影响，也就是说不同事务可以在锁定的区间都添加间隙锁。**
> - **间隙锁会导致死锁的发生。**

> 当我们使用范围条件查询而不是等值条件查询的时候，InnoDB就会给符合条件的范围索引加锁，在条件范围内并不存的记录就叫做"间隙（GAP）"，主要目的，就是为了防止其他事务在间隔中插入数据，以导致幻读。
>
> 比如有数据：1，2，3，4，5，6，7，8
>
> `SELECT * FROM t WHERE key > 4 AND key < 7;`
>
> 那么就会给4-7这个范围加间隙锁，意味着不能在这段修改数据。

- Next-key Lock

> 是行锁和间隙锁的结合。可重复读RR级别下才会有next-key锁！ 也就是说，next key锁不是一个单独的锁，就我理解，它其实是一个概念，这个概念是由上面两个锁的概念组合而来的，主要用来解决当前读产生的幻读问题的。

- 死锁

> 死锁是指两个或多个事务在同一资源上相互占用，并请求锁定对方的资源，从而导致恶性循环的现象。比如
>
> | session1 | session2 |
> | --- | --- |
> | begin;
update t set name='xx' where id=1; | begin |
> | | update t set name='xx' where id=2; |
> | update t set name='xx' where id=2; | |
> | 等待..... | update t set name='xx' where id=1; |
> | 等待..... | 等待..... |

2、说说事务隔离级别与锁的关系

- 读未提交：不需要加共享锁，这样就不会跟被修改的数据的排他锁冲突。
- 读已提交：需要加共享锁，但是语句执行完后要释放锁。
- 可重复读：读操作需要加共享锁，但是在事务提交之前并不释放共享锁，也就是必须等待事务执行完毕以后才释放共享锁。
- 串行化：该级别锁定整个范围的键，并一直持有锁，直到事务完成。

3、什么是死锁

死锁是指两个或多个事务在同一资源上相互占用，并请求锁定对方的资源，从而导致恶性循环的现象。死锁在InnoDB中才会出现死锁，MyISAM是不会出现死锁，因为MyISAM支持的是表锁，一次性获取了所有得锁，其它的线程只能排队等候。

4、怎么解决死锁?

- 等待事务超时，主动回滚。
- 进行死锁检查，主动回滚某条事务，让别的事务能继续走下去。

下面提供一种方法，解决死锁的状态:

```
-- 查看正在被锁的事务
SELECT * FROM INFORMATION_SCHEMA.INNODB_TRX;
```

下面提供三种一定概率避免死锁的方法：

- 如果不同程序会并发存取多个表，尽量约定以**相同的顺序**访问表，可以大大降低死锁概率。
- 在同一个事务中尽可能做到一次锁定所需要的所有资源，减少死锁产生概率。
- 对于非常容易产生死锁的业务部分，可以尝试使用升级锁定颗粒度，通过表级锁定来减少死锁产生的概率。

5、死锁出现的案列

- 间隙锁导致死锁

> 表结构和数据：
>
> ```
> CREATE TABLE `child` (
>   `id` int(11) unsigned NOT NULL AUTO_INCREMENT,
>   `age` int(11) unsigned DEFAULT NULL,
>   PRIMARY KEY (`id`),
>   KEY `age` (`age`) USING BTREE
> ) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8mb4;
>
> INSERT INTO `child` (`id`, `age`) VALUES ('10', '10');
> INSERT INTO `child` (`id`, `age`) VALUES ('20', '20');
> INSERT INTO `child` (`id`, `age`) VALUES ('40', '40');
> ```
>
> 然后如下步骤复现死锁

Session A	Session B
begin	
11:29:31>>select * from child where age=15 for update; Empty set (0.00 sec)	
	begin
	11:29:31>>select * from child where age=15 for update; Empty set (0.00 sec)
11:31:27>>insert into child values (15,15);	
Query OK, 1 row affected (2.39 sec)	11:31:31>>insert into child values (15,15); ERROR 1213 (40001): Deadlock found when trying to get lock; try restarting transaction
commit	commit

当我们锁定age=15的值的时候，由于这个值不存在，所以锁定了区间(10,20)，而在session B上也锁定了这个区间，由于间隙锁不存在冲突，所以session B这个语句执行成功。也就是说，两个会话都锁定了这个区间，此时我们在session A上插入age=15的记录，可以发现锁住了，迟迟没有操作结果，而在session B上插入age=15的记录，则直接提示deadlock found，此时session A上的语句执行成功，也就是说这两个会话上的insert操作互相相成了死锁。

- 批量入库，存在则更新不存在则插入，`insert into t1(xxx,xxx) on duplicate key update name = 'xxx';`，具体原因可以自行Google搜下：

`insert into … on duplicate key update 死锁`，也可以看下下面这篇文章，其实也是间隙锁在捣鬼。

https://zhuanlan.zhihu.com/p/29349080

六、 索引

1、列举一些导致索引失效的场景

- 查询条件包含or导致后面的索引失效
- like左模糊

- 字符串类型没加单引号
- 联合索引，查询时的条件列不是联合索引中的第一个列
- 在索引列上使用mysql的内置函数
- 对索引列运算（如，+、-、*、/）
- 索引字段上使用！= 或者 < >，not in
- 发生隐式类型转换会导致索引失效
- mysql估计使用全表扫描要比使用索引快,则不使用索引。这个是mysql自己决定的，发现后我们可以使用强制索引的语法让MySQL强制走索引

2、索引不适合哪些场景?

- 表里数据量很少
- 更新频繁的字段
- 区分度很低的字段

3、为什么用B+树而不是B树或二叉树或平衡二叉树?

为什么不是一般二叉树?

如果二叉树特殊化为一个链表，相当于全表扫描。平衡二叉树相比于二叉查找树来说，查找效率更稳定，总体的查找速度也更快。

为什么不是平衡二叉树呢?

我们知道，在内存比在磁盘的数据，查询效率快得多。如果树这种数据结构作为索引，那我们每查找一次数据就需要从磁盘中读取一个节点，也就是我们说的一个磁盘块，但是平衡二叉树可是每个节点只存储一个键值和数据的，如果是B树，可以存储更多的节点数据，树的高度也会降低，因此读取磁盘的次数就降下来了，查询效率就快了。

那为什么不是B树而是B+树呢?

1）B+树非叶子节点上是不存储数据的，仅存储键值，而B树节点中不仅存储键值，也会存储数据。innodb中页的默认大小是16KB，如果不存储数据，那么就会存储更多的键值，相应的树的阶数（节点的子节点树）就会更大，树就会更矮更胖，如此一来我们查找数据进行磁盘的IO次数有会再次减少，数据查询的效率也会更快。

2）B+树索引的所有数据均存储在叶子节点，而且数据是按照顺序排列的，链表连着的。那么B+树使得范围查找，排序查找，分组查找以及去重查找变得异常简单。

4、聚簇索引与非聚簇索引的区别

- 聚簇索引：索引的叶节点就是数据节点。所以不需要回表。主键id就是聚簇索引，有且仅有一个。
- 非聚簇索引：叶节点仍然是索引节点，只不过有一个指针指向对应的数据块。要想更多的数据需要回表查，可以有多个非聚簇索引。

5、如何写sql能够有效的使用到复合索引?

确保最左匹配原则有效。

6、Hash索引和B+树区别是什么？你在设计索引是怎么抉择的?

- B+树可以范围查询，Hash不行
- B+树可以复合索引的最左侧原则，Hash不行
- B+树可以支持order by排序，Hash不行
- B+树在等值查询上的效率不如Hash

- B+树支持右模糊查询走索引，hash索引根本无法进行模糊查询

7、索引有哪几种类型?

- 主键索引：也是聚簇索引，一张表只能有一个 ，不允许为null且唯一，查询的时候不需要回表。
- 唯一索引：不允许为null且唯一。一张表可以有多个。
- 普通索引：普通索引。
- 全文索引：全文检索用，对文本内容进行分词搜索。
- 覆盖索引：select的字段正好是索引字段，这时候不需要回表操作。
- 组合索引：多个列值组成一个索引，使用的时候要遵循最左匹配原则。

8、创建索引有什么原则呢?

- 最左前缀匹配
- 频繁作为查询条件的字段
- 频繁更新的字段不适合
- 索引列不能参与计算和使用一些函数
- 优先考虑索引组合，而不是每次都新建索引
- 在order by或group by字句中，使用索引要遵循最左前缀匹配
- 区分度低的列不适合
- 大数据类型的列不适合（比如text等）

9、什么是最左前缀原则？什么是最左匹配原则?

就是最左优先，在创建多列索引时，要根据业务需求，where子句中使用最频繁的一列放在最左边。比如有组合索引（a,b），那么使用的时候只写where b = xxx会导致索引失效，因为a在前面丢了，但是写成这样where b = x and a = x;这样索引是可以生效的，因为优化器阶段会给优化成where a = x and b = x，不会让索引失效。

10、覆盖索引、回表等这些，了解过吗?

- 覆盖索引：查询列就是所使用的索引列，这样不需要回表查询。
- 回表：二级索引无法直接查询所有列的数据，所以通过二级索引查询到聚簇索引后，再查询到想要的数据，这种通过二级索引查询出来的过程，就叫做回表。

11、使用索引查询一定能提高查询的性能吗？为什么?

通常情况下是可以的，但是也有特例，比如你在区分度不高的字段上使用索引（比如性别），那就未必能提升性能，因为索引也需要物理存储空间的。

12、count(1)、count(*) 与 count(列名) 的区别?

- count(*)包括了所有的列，相当于行数，在统计结果的时候，不会忽略列值为NULL
- count(1)包括了忽略所有列，用1代表代码行，在统计结果的时候，不会忽略列值为NULL
- count(列名)只包括列名那一列，在统计结果的时候，会忽略列值为空（这里的空不是只空字符串或者0，而是表示null）的计数，即某个字段值为NULL时，不统计。

效率上MySQL对count(*)做过优化， count(1)≈count(*)>count(列名)

13、列值为NULL时，查询是否会用到索引?

列值为NULL也是可以走索引的。但是执行计划对列进行索引，应尽量避免把它设置为可空，因为这会让MySQL 难以优化引用了可空列的查询，同时增加了引擎的复杂度。

14、非聚簇索引的查询都需要回表吗?

不一定，如果查询语句的字段全部命中了索引，那么就不必再进行回表查询，比如覆盖索引。

15、什么是索引下推?

MySQL5.6新做的优化。索引下推可以在索引遍历过程中，对索引中包含的字段先做判断，直接过滤掉不满足条件的记录，减少回表次数，比如：

```
-- (name,is_del)是组合索引
select * from t_user where name like '张%' and is_del=1;
```

在MySQL 5.6之前，只能从匹配的位置一个个回表。到主键索引上找出数据行，再对比字段值是不是张开头且is_del=1。

在MySQL 5.6之后，索引下推可以在索引遍历过程中，对索引中包含的字段先做判断，直接过滤掉不满足条件的记录，也就是回表查询的时候只回表查询有效结果集，也就是姓张的且is_del=1的这些数据进行回表，简单来说就是先过滤好需要的结果，然后去回表查询全部字段。

16、频繁更新索引字段可以吗？为什么?

可以但不推荐，因为每个索引都是一棵B+树，是按照大小排序的，更新索引字段的话可能会造成多个数据页的数据之间挪动。

17、为什么group by 和 order by会使查询变慢

group by 和 order by操作通常需要创建一个临时表来处理查询的结果，所以如果查询结果很多的话会严重影响性能。

18、怎么优化order by?

order by 里的字段可以组合索引，但是要么都加升序要么都加降序，不能部分字段升序，部分字段降序，这样走不了索引。

19、怎么优化group by?

遵循最左前缀匹配就行。

20、从innodb的索引结构分析，为什么索引的key长度不能太长?

key 太长会导致一个页当中能够存放的key的数目变少，间接导致索引树的页数目变多，索引层次增加，从而影响整体查询变更的效率。

七、主从复制和读写分离

1、主从复制有什么好处？解决了哪些问题?

- 做备份数据库，主库宕机后，从库可以切换为主库继续工作。
- 提高并发能力

2、MySQL主从复制原理的是啥?

- 主库将在每个事务提交之前将变更写入binlog

- 然后从库起一个I/O线程连接到master，master机器会为slave开启binlog dump线程。当master的 binlog发生变化的时候，binlog dump线程会将binlog的内容发送给该I/O线程。该I/O线程接收到binlog内容后，再将内容写入到本地的relay log。
- 若读取的进度已经跟上了主库，那么就进入睡眠状态并等待主库产生新的事件。
- 最后从库中有一个SQL线程会从relay log里顺序读取日志内容并在从库中执行一遍，从而与主库的数据保持一致

3、主从复制半同步复制（semi-sync）什么意思?

> 因为MySQL默认主从方式会造成数据丢失，他是执行完SQL后就提交了事务，此时从库还没拉取变更的binlog，主库挂了，那么从库升级为主库，从库上没有主库刚提交的变更记录，数据丢失。而半同步复制很好的解决主库数据丢失的问题。

主库将变更写入binlog后，就会将数据同步到从库，然后从库同步完binlog且将日志写入自己本地的relay log后会返回给主库一个ack，主库收到ack后才会提交事务。这样完美避免了数据丢失的情况。

4、哪些场景可能造成主从延迟?

- 从库所在机器的性能比主库的差很多
- 做了读写分离，从库压力过大导致
- 大事务

> 因为主库上必须等事务执行完成才会写入 binlog，再传给备库。所以，如果一个主库上的语句执行 10 分钟，那这个事务很可能就会导致从库延迟 10 分钟。所以不要一次性地用 delete 语句删除太多数据。其实，这就是一个典型的大事务场景。

5、如何解决MySQL主从同步的延时问题?

> 也可以被问：读写分离有哪些问题？ 那就是主从同步的延时问题，可能读到旧数据。

- 并行复制

> 提升主从同步效率的一种手段，可以尽量避免主从同步的延迟问题，从库开启多个线程并行去读取relay log中不同库的日志，然后并行重放不同库的 日志，着是库级别的并行。MySQL5.7支持的。

- 强制走主库方案

> 将请求分成两类：允许读到旧数据的和不允许读到旧数据的，把允许读到的那些场景走读库，不允许的那些SQL强制去读主库。

- 判断主备无延迟方案

> 1.等主库位点方案
>
> 2.GTID 方案

6、并行复制是什么意思?

提升主从同步效率的一种手段，可以尽量避免主从同步的延迟问题，从库开启多个线程并行去读取relay log中不同库的日志，然后并行重放不同库的 日志，着是库级别的并行。MySQL5.7支持的。

7、介绍下主从复制GTID的方式?

以前 MySQL 的主从复制是基于复制点的，slave 从 master 二进制日志的某个位置开始复制

有了 GTID 之后，就多了一种复制方式，MySQL 在每个事务操作时都会分配一个全局唯一的ID，slave就可以基于这个ID进行复制，只要是自己没有复制过的事务，就拿过来进行复制，可以不用关心具体的复制位置了

优点：

- 可以更方便的故障转移，出现问题时，多个slave不用根据新master的二进制偏移量来同步了
- 主从配置更简单

8、如何解决互为主备后的循环复制问题?

互为主备，也就是A和B两个节点互为主从，A既是B的主节点又是B的从节点，反过来B既是A的主节点又是A的从节点。也是双M架构。那么就产生个问题：

A节点变更了一条记录将binlog同步给了B节点，B节点执行完这条语句后也产生了一个binlog又将日志传给A，这不是死循环吗？这就是互为主备的循环复制问题。

我们可以用下面的逻辑，来解决两个节点间的循环复制的问题：

- 规定两个库的 server id 必须不同，如果相同，则它们之间不能设定为主备关系
- 一个备库接到 binlog 并在重放的过程中，生成与原 binlog 的 server id 相同的新的 binlog
- 每个库在收到从自己的主库发过来的日志后，先判断 server id，如果跟自己的相同，表示这个日志是自己生成的，就直接丢弃这个日志。

按照这个逻辑，如果我们设置了双 M 结构，日志的执行流就会变成这样：

- 从节点 A 更新的事务，binlog 里面记的都是 A 的 server id
- 传到节点 B 执行一次以后，节点 B 生成的 binlog 的 server id 也是 A 的 server id
- 再传回给节点 A，A 判断到这个 server id 与自己的相同，就不会再处理这个日志。所以，死循环在这里就断掉了。

但依然有风险，如下：

- 在一个主库更新事务后，用命令 set global server_id=x 修改了 server_id。等日志再传回来的时候，发现 server_id 跟自己的 server_id 不同，就只能执行了。

9、为什么要做读写分离?

- 分摊服务器压力，提高机器的系统处理效率。读写分离适用于读远比写多的场景
- 增加冗余，提高服务可用性，当一台数据库服务器宕机后可以调整另外一台从库以最快速度恢复服务

10、如何实现MySQL的读写分离?

- mycat中间件
- sharding-jdbc这种在代码业务层自己控制

八、MySQL分库分表

1、为什么要分库分表?

单机的存储能力、连接数、QPS是有限的，分库分表是一种很好的优化手段，将大表拆分到不同库不同表，减轻数据量，提高查询性能。

2、分库分表的技术有哪些?

- mycat

> 需要部署，自己运维一套中间件，运维成本高，但是好处在于对于各个项目是透明的，如果遇到升级之类的都是自己中间件那里搞就行了。

- sharding-jdbc

> 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，但是如果遇到升级啥的需要各个系统都重新升级版本再发布，各个系统都需要耦合sharding-jdbc 的依赖。

3、分库分表的拆分方式有？他们分别主要解决什么问题？

- 垂直拆分

> 把一个有很多字段的表给拆分成多个表，或者是多个库上去，每个库表的结构都不一样，每个库都都包含部分字段。一般来说会将较少的访问频率很高的字段放到一个表里去，然后将较多的访问频率很低的字段放到另外一个表里去。因为数据库是有缓存的，你访问频率高的行字段越少，就可以在缓存里缓存更多的行，性能就越好。这个一般在表层面做的较多一些。

- 水平拆分

> 表结构相同，拆分到多张表，这多张表可以在同一个库也可以在不同库，然后采取算法（比如哈希）将数据分散到多张表里，减轻单表的压力。

4、单张大表要进行分库分表怎么办？

> 大表怎么优化？某个表有近千万数据，CRUD比较慢，如何优化？
>
> 分库分表。

(1) 停机迁移方案

停机迁移方案中，就是把系统在凌晨 12 点开始运维，系统停掉，然后提前写好一个导数据的一次性工具，此时直接跑起来，然后将单库单表的数据写到分库分表里面去。 导入数据完成了之后，修改系统的数据库连接配置，然后直接启动连到新的分库分表上去。

(2) 双写迁移方案

此方案不用停机，比较常用。 简单来说，就是在线上系统里面，之前所有写库的地方，增删改操作，都除了对老库增删改，都加上对新库的增删改，这就是所谓的双写。同时写两个库，老库和新库。然后系统部署之后，新库数据差太远，用之前说的导数据工具，跑起来读老库数据写新库，写的时候要根据，判断下读出来的数据在新库里没有，或者是根据更新时间对比，比新库的数据新才会写。

5、分库分表有哪些算法？

比如：

- 数据分段

> 比如：user_id属于[0, 100万]为0库，属于[100万, 200万]为2库。以此类推。优点：简单、数据均衡、扩容简单。缺点：因为未知最大值，所以无法用时间戳作为key，这个方法不能用表的自增主键，因为每个表都自增数量不是统一维护。所以需要有一个发号器或发号系统做统一维护key自增的地方。
>
> 推荐日志表可以按照此方式，按照时间来分，比如一个月一张表。

- hash取模

> 比如：user_id%2=0为0库，user_id%2=1为1库。优点：简单、数据均衡、负载均衡。缺点：扩容困难，要迁移数据，%2变%3麻烦。

- 一致性哈希

> hash取模简单粗暴，但是涉及到数据迁移的很多，几乎全部挪动。一致性哈希可以解决这个问题，但是也无法避免数据不迁移，只是迁移的少了而已，一般分布式缓存用一致性哈希居多，具体如下：

> 假设分为4个库，每个库16张表。
>
> 库：一致性哈希算法(库的标识 [比如ip]) % 2^32
>
> 数据：一致性哈希算法(key) % 2^32，然后对比四个库的哈希值，看看哪个最小就放到哪个库里，也就是顺时针找最近的节点。然后再取模16找到具体在哪个表里。
>
> 优点：四个库不够用了，多加一个库，那么只需要迁移四分之一的数据即可，不需要全部迁移。

6、分库分表后面临的问题

- 分布式事务问题

> TCC、LCN、Seata、本地消息表、RocketMQ

- 分页、排序、跨库聚合统计等问题

> 一般做法是分N次查询，然后业务代码里自己进行分页、排序或聚合

- 跨库join

> 一般做法是分N次查询，然后业务代码里自己组装数据

- 全局主键

> 分布式id。
>
> - UUID
> - 数据库自增id
> - Redis
> - 号段模式
> - 雪花算法
> - 美团Leaf：支持号段模式+雪花算法

九、生产环境实战案例

1、生产环境下的一次性能抖动

生产环境下的一次性能抖动

2、死锁出现的案列

死锁出现的案列

3、字符串单引号

字符串类型的唯一索引字段，select的时候没有加单引号('')导致索引失效了，走了全表扫描。血淋淋的泪，很常见的低级错误。

4、name字段utf8传表情

name字段是utf8类型的，前端根据name查用户的时候发来个表情，直接数据库告警，因为表情需要utf8mb4才支持，我们字段确是utf8类型的，隐式类型转换 了，索引失效。

5、社交app用户信息搜索

背景：类似陌陌、探探等社交APP，需要按照省市、性别、年龄范围这三个字段查找用户。如下SQL：

`SELECT * FROM user_info WHERE province = xx AND city = xx AND sex = 1 AND age > xx AND age < xx;`

很简单，我们设计个复合索引就完事了，(province, city, sex, age)

但是用户可以不选择性别，只按照省市和年龄范围选，这时候咋办呢？如果sql条件里不带上sex字段的话，那么索引失效了，因为不符合最左匹配原则，那么我们可以将SQL写成如下：

`SELECT * FROM user_info WHERE province = xx AND city = xx AND sex = -1 AND age > xx AND age < xx;`

> sex = -1 代表全部的意思，强行给他带上这个sex索引列即可。

如果还支持兴趣爱好等这种固定枚举值（复选框，可选择几个进行查询）的查询方式的话，那么可以重新设计个索引：

(province, city, sex, hobby, age)

`SELECT * FROM user_info WHERE province = xx AND city = xx AND sex = -1 AND hobby IN ('xx', 'xx', 'xx') AND age > xx AND age < xx;`

需要注意的是age索引字段放到最后，因为他是范围查询，放到前面的话会导致后面的索引失效。而其他几个字段都是等值查询，包括IN里面也是固定的几个，可以完美复合最左匹配原则。这样一来所有字段都可以用上索引了。

如果要搜索所有女性，且按照印象分从高往低排序的话，那么可以建立个辅助索引：（sex, score）

`SELECT * FROM user_info WHERE sex = 1 ORDER BY score DESC LIMIT xx,xx;`

总结：

尽量用1-2个复杂的组合索引抗下你80%的查询，然后用1-2个辅助索引抗下剩余的20%查询场景，充分利用索引，别稀里糊涂搞一大堆索引出来。

6、上千万级别的评论系统深度分页

深分页很慢，怎么解决的呢？

7、亿级用户表统计流失人数的SQL太慢

背景：要找到最后登录时间小于xxx的全部用户，然后进行统计看看有多少个。

SQL：

`SELECT COUNT(id) FROM user WHERE id IN (SELECT user_id FROM user_extent_info WHERE last_login_time < xxx);`

几十秒才出结果。太慢了。

调优：

经过explain分析发现先执行了子查询，然后将子查询的结果生成了一个临时表到磁盘（到磁盘，这就慢了），接着他对user表做了一个全表扫描，扫描过程就是把每一条数据都放到临时表里的数据去做全表扫描（亿级用户搞全表扫描！）。

然后通过`show warnings;`命令发现他采取了semi join的方式来给我们执行的sql，而生成临时表到磁盘也是semi join搞的鬼，所以两种办法解决：

- 禁用semi join：`set optimizer_switch=semijoin=off;`但是生产环境谁让你瞎搞？
- 改写SQL如下：

```
`SELECT COUNT(id) FROM user WHERE id IN (SELECT user_id FROM
user_extent_info WHERE last_login_time < xxx) OR id IN (SELECT user_id
FROM user_extent_info WHERE last_login_time < -1);`
```

也就是说where后面跟个or，这个or是根本不可能成立的，这样我们在分析执行计划发现MySQL并没有再进行semi join优化了，查询速度几百毫秒解决了。

8、大事务影响性能

背景：突然告警发现业务系统的SQL很慢。

原因：在一个事务里删除了千万级别的数据，产生了大事务。kill掉这个del语句线程就好了。

风险：

- 锁定太多的数据，造成大量的阻塞和锁超时， 回滚时所需要时间比
- 较长，执行时间长，容易造成主从延迟，
- 如果主库的事务执行了几个小时后再提交，才会写入binlog 里，从库才会读binlog 日志 才开始同步
- innodb 是行级锁，当涉及所有记录时，就会相当于整个表锁住，

如何处理大事务：

- 避免一次处理太多的数据
- 移除不必要在事务中的select 操作

9、强制使用索引

背景：生成环境，同一条sql在不同的从库执行，产生的执行计划不同，一个使用了索引，一个未使用索引。

```
explain SELECT * FROM `database`.`table` FORCE INDEX(create_time) WHERE
create_time >= xxx and create_time <= xxx ORDER BY create_time asc LIMIT xxx,
xxx;
```

原因：分析是索引文件或者表的碎片导致，后咨询阿里DBA给分析是表的碎片问题导致产生的执行计划不正常。

解决：

- 方案1：执行`OPTIMIZE TABLE`修复碎片或者执行`ALTER TABLE foo ENGINE=InnoDB`，以上两种操作都会锁表，对于数据量大，且业务高峰期执行需要慎重。
- 方案2：强制索引，也就是`FORCE index create_time`，强制mysql 引擎使用索引，这个强制语法在这类问题上很有效。

十、分散型题目

1、为什么表数据删掉一半，表文件大小不变?

背景：100GB的日志表，三个月前的数据都迁移走了，可以直接delete掉，然后delete后发现表大小还是100GB。

原因：删除了那么多数据，其实只是代表那些数据所在的数据页变成可复用的，实际磁盘空间并没有减少。所以只是标记了下数据页变为可用。

解决：重建表，命令：`alter table A engine=InnoDB;`

2、为什么我只查一行的语句，也执行这么慢?

背景：`select * from t where id=1;`就这SQL很慢，为啥?

分析：

- 大概率是表被锁住了，用`show processlist;`来分析下是不是存在`Waiting for table metadata lock`的。
- 如果不是锁表的话，那么看下是不是在等flush，在表里执行`select * from information_schema.processlist where id=1;`，看看这个线程的状态是 不是`Waiting for table flush`，这个状态表示的是，现在有一个线程正要对表 t 做 flush 操作。MySQL 里面对表做 flush 操作的用法，一般有以下两个：`flush tables t with read lock;` `flush tables with read lock;`
- 检查下是不是存在行锁：`select * from t sys.innodb_lock_waits where locked_table='test.t'\G`

3、怎么最快地复制一张表?

- mysqldump 方法
- 导出csv，然后load data导入新表

```
select * from db1.t where a>900 into outfile '/server_tmp/t.csv';
load data infile '/server_tmp/t.csv' into table db2.t;
```

4、MySQL数据库cpu飙升到500%的话他怎么处理?

- top命令找到占用cpu最高的进程id。
- 如果是 mysqld 造成的，`show processlist;`看看是不是有消耗资源的 sql 在运行。
- 如果有的话先kill掉这些线程，保证线上稳定运行，然后再分析SQL如何优化。

5、如果某个表有近千万数据，CRUD比较慢，如何优化?

分库分表

某个表有近千万数据，可以考虑优化表结构，分表（水平分表，垂直分表），当然，你这样回答，需要准备好面试官问你的分库分表相关问题呀，如

- 分表方案（水平分表，垂直分表，切分规则hash等）
- 分库分表中间件（Mycat，sharding-jdbc等）
- 分库分表一些问题（事务问题、分布式id、跨节点Join等问题）
- 解决方案（分布式事务等）

索引优化

除了分库分表，优化表结构，当然还有所以索引优化等方案。

6、数据库自增主键可能遇到什么问题?

- 使用自增主键对数据库做分库分表，可能出现诸如主键重复等的问题
- 自增主键会产生表锁，从而引发问题
- 自增主键可能用完问题

7、百万级别或以上的数据，你是如何删除的?

- 我们想要删除百万数据的时候可以先删除索引

- 然后批量删除其中无用数据
- 删除完成后重新创建索引。

```
alter table T engine=InnoDB;
```

8、关心过业务系统里面的sql耗时吗？统计过慢查询吗？对慢查询都怎么优化过？

- 我们平时写Sql时，都要养成用explain分析的习惯。
- 慢查询的统计，运维会定期统计给我们
- 我们业务系统也有监听器统计耗时的SQL到日志文件

优化慢查询：

- 先确定是不是网络、cpu、io、锁、大事务的问题，如果不是的话在分析SQL语句索引方面。
- 分析语句，是否加载了不必要的字段/数据。
- 分析SQl执行句话，是否命中索引等。
- 如果SQL很复杂，优化SQL结构
- 如果表数据量太大，考虑分表

9、你进行SQL优化的时候一般步骤是什么？

- 避免返回不必要的数据
- 加索引，explain进行分析
- 适当分批量进行
- 分库分表
- 读写分离

10、深分页很慢，怎么解决的呢？

利用子查询优化超多分页场景。先快速定位需要获取的id段，然后再关联。这样可以防止回表。

比如：

```
-- 原来SQL
SELECT * from employee WHERE 条件 LIMIT 1000000,10
-- 优化成如下
SELECT a.* FROM employee a, (SELECT id FROM employee WHERE 条件 LIMIT
1000000,10 ) b WHERE a.id=b.
```

11、主键自增ID还是UUID之间选择哪个？为什么？

自增ID，因为主键是聚簇索引，也就是一颗B+树，索引字段在页里是从小到大排序的，所以自增可以避免页分裂，uuid的话会频繁页分裂。因为每次插入uuid都可能比之前的小，就需要插入到前面去，就造成了页分裂，多个数据页之间的数据来回挪动。

12、如何排查项目中可优化的SQL？

- 1.把慢SQL的时间设置为0，慢SQL=0也就是记录了全部SQL到slowlog里。**调整慢SQL的时间参数不需要重启MySQL服务。**
- 2.然后用**工具（pt-query=digest）**分析哪些SQL最慢，哪些SQL执行次数最多等指标，项目上线前2天可以这么配置，分析完后再把慢日志调回原来数值，否则一直写slowlog也会对mysql造成性能压力（10%以内）。
- 3.然后调优那些单次执行时间最长的（几乎都是索引有问题，所以调优）、总访问次数最多的SQL（因为这种SQL肯定是系统核心功能的SQL，做到极致，比如发现某种配置类的SQL，内容长

期不更新的，就可以放到缓存里等）。