mysql

1.什么是索引，优缺点

3.mysql索引的数据结构类型

3.1 mysql索引的一个发展史

3.2 b+树对应如何用物理模型实现

3.3 索引的类型有那些及其定义

4.如何建索引

4.1 什么情况适合建索引

4.2 什么情况不适合建索引

5.怎么排查索引

6.索引失效的原因

7.如何优化慢sql

7.1 操作步骤，及其分析

7.2 案例分析

8.mysql如何保证事务

8.1 没有水平的回答

8.2 有含金量至少得说工作流程

9.mvcc机制如何实现

10.mysql引擎有那些

11.mysql中有那些锁

1.什么是索引，优缺点

- 索引的概念：是一种排好序的，支持快速查找的，数据结构。
- 索引的优点：支持快速查找，精准定位，减少磁盘IO；支持排序，减少CPU消耗。
- 索引的缺点：索引也许要存储，索引过多会导致存储容量过大，索引我们采用了文件存储索引的形式。

3.mysql索引的数据结构类型

3.1 mysql索引的一个发展史

- mysql，有很多版本，开发的时候，就考虑用链表做存储索引的结构，但这不可能，链表查询最慢，不合适；
- 用二叉树搜索树：发现有一个致命的缺点，如果我顺序增加索引值，会生成一个链表维护索引，不可取；
- 用平衡二叉树：用于解决，二叉搜索树的不平衡问题；但有一个缺点：他的深度太高，查询效率也慢；
- 用b树：b树的特点：是每个节点可以存多个索引值，而且每隔节点有序，b树节点存的是索引和数据；优点是减少了树的深度，但发现数据量大的时候还是会造成深度比较深；
- 用hash表：发现hash表的查找速度远超于树的查找效率，但有两个致命的缺点：hash冲突形成的链表可能太长，不支持范围查找；
- 而b+树，拥有b树的特性，唯一不同的是b+树树的非叶子节点存的是冗余索引；而叶子节点存的是所有的索引和数据。

3.2 b+树对应如何用物理模型实现

1. b+树的模型之前，得先了解一下b+树的特点，这棵树每一个节点可以有多个值，值是排好序的，非叶子节点是存的是冗余的索引，而叶子节点存的是索引和数据，叶子节点之间还有双向指针支持范围查找。
2. b+树实际是如何生成的，我们在操作数据库的时候都会进行磁盘IO，举个列子，我们向数据库中加入 记录的时候，需要先通过磁盘IO读取一页为16kb的数据，这页中分为几个区域，我叫它页模型，页有页头，页头中有前指针和后指针用于指向前一页和下一页；其他分为目录区和用户数据区，用户数据区是一个链表结构，当我们插入一条记录的时候，就会向这链表中插入一条记录，目录是用来方便快速查找的，如果只有链表，我们在查找的时候需要一个一个的遍历，才能找到某一条记录，所以目录中就将链表记录进行分组，每组最小值进行维护在目录结构中；这样查找速度就能提高，譬如一条记录1kb，一页数据16kb也就插入16条记录；当插入满的时候，就会同样维护下一页，页与页之间用页模型中的表头维护成链表，同样链表查询时间复杂度太高b(O(n))，就在链表上建立一层，用一页数据专门存放没页的最小值，和目录一样的作用；数据到达一定量时，就会在建立一层，整体结构看就是一颗b+树的特征。查找时利用二分查找log n时间复杂度。我们算一下b+树最多能存多少条数据，一条1kb 一页就是16条，第二层一个索引按照bigint算8字节，加一个指针4字节共12个字节，16kb/12大致就是1770所以两层b+树可以存1770*16三层就是1770*1770*16五千万条数据最大，当数据达到这个级别，就要考虑分库分表。

3.3 索引的类型有那些及其定义

索引分为好多类型：单值索引（唯一，主键）,多值索引（覆盖，复合），聚集索引和非聚集索引

1. 主键索引：我们的主键索引，对于inndb而言，他的b+树存储形式叶子节点存的是数据和索引。主键索引我们在建立时希望是数字类型的，自动增长的，整性的；为什么，数字整数类型相对于其他数据类型而言标记效率比较高，在二分查找的时候可以高效执行;自动增长是方便b+树生成的时候能搞效率更高；
2. 唯一索引：就是我们在维护索引的时候维护的是一个b+树的形式，而叶子节点存的是主键和唯一索引，所以这个就有了一个回表操作。
3. 复合索引：就时多个字段建立一个索引，我们在查询时一定要复合建立索引的顺序规则，最左前缀法则。最左前缀法则，假如给你a_b_c三个字段建立索引，给的顺序为bca就是不符合了，mysql在执行的时候，底层会对sql优化，优化成索引的顺序；所以这不是最左前缀法则标准；最左前缀法则指的是三个字段在缺失的前提下，带头大哥不能少，中间兄弟不能断的标准。
4. 覆盖索引：覆盖索引就是当我们对字段建立了索引，当我们查询的时候希望查询的字段就是索引字段，这叫覆盖索引。
5. 聚集索引：指的时在b+树结构中，叶子节点的数据和索引在一起，实际上是复合inndb的索引数据结构标准的，存放的文件形式是frm和idb文件持久化存储的。
6. 非聚集索引：与之相反，叶子节点村的是索引和数据的引用地址，索引可以通过引用地址找到实际数据，整体看非聚簇索引比聚簇索引多了一次查询，效率会慢；存储文件是frm myi myd三个文件是mysiam引擎的实现方式。

4.如何建索引

4.1 什么情况适合建索引

- where频繁查询的字段
- 排序分组字段
- 连接字段: 有讲究：如果查询为left join 索引建在右边；right join 索引建立在左表上效率更高
- 主键字段自动建立索引

4.2 什么情况不适合建索引

- where用不到的字段
- 频繁增删改的字段：索引是排好序的数据结构；增删改需要数据库维护索引；
- 频繁重复的字段：建立索引效率得不到明显提示，还占用内存空间；

5.怎么排查索引

1. 恢复慢sql现场

2. 开启慢sql日志
3. explain+sql查询

- explain是一个查看一个sql语句，mysql到底是如何执行的
- id 用于代表查询的编号，如果id相同，我们的执行顺序如上到下依次执行，比如连接查询；如果id不同，我们的执行顺序是id大的先执行，然后id相同遵循上述顺序；如连接查询和子查询的联合使用。
- select_type 这个指的是5查询类型，如主查询，子查询，union等查询。
- table 代表的是执行的表名，有时候也可能是虚表，常常id,select_type,table配合在一起看来确定表的执行顺序。
- type 这个字段有意思，代表的是索引走的类型：最差的是all全表扫描，比好一点的就是扫描索引index,在下面的就是走索引的范围查找：range;走索引的固定查找：ref:用于匹配命中的固定字段的所有记录，eq_ref：用于匹配命中的固定字段的一条记录，因为复合条件的只有一条；更完美的是const：用来区分普通索引和主键索引的，就是会不会表的问题；system 最高级的：要求表中只有一条记录。
- select_keys: 代表mysql认为你需要走的索引；
- key : 代表你实际命中的索引;
- key_len: 代表索引的长度;
- ref: 这个ref是与id.....是同级的用于体现是那个索引：那个库，那张表，那个索引字段
- rows: 代表扫描索引过程中扫描了多少条记录；
- extra: 这个字段是很关键的；有几个特别重要的信息：好的是using index,using where :代表走了索引和索引查找；坏的是 using filesort 和 using temporary 没有走索引搞了个文件临时排序；另一个是搞了个临时表，出现这两个字段，切记我们一定要去优化我们的sql，不然会带来严重的线上问题；

4. show prefile

6.索引失效的原因

索引失效的概念就是，我们建了索引，并没有用到索引的现象：索引失效；

1. 最左前缀不匹配（前面我介绍过了，请看标题3.3）
2. 在索引上做了手动计算和自动计算：手动就是我们不要在索引上做计算；自动计算：我举个例子我们在查询某个字段为字符串的时候，我们给了个数字，mysql默认只会将字符串类型转换为数字类型；"2000"---2000;"df"---->0这样就发生了隐式转换，导致索引失效；
3. 使用了特殊的关键字：!= <> is not null is null like like中有特殊情况："%abc%","%ab"都会失效，"ab%"是不会失效的；

4. 范围查找会导致后面的索引失效
5. 查找了不必要的字段：导致全表扫描比走索引还高效；举个例子：a_b_c是个复合索引，查a>1的字段无疑是查全部记录，但查询字段为* 二分+数据全部回表；你若查的是abc字段ok我们直接返回无需回表，高效；

7.如何优化慢sql

7.1 操作步骤，及其分析

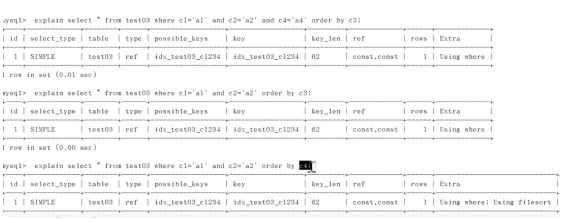
1. 分析sql出错原因

- 线上找出慢sql
- explain分析
- 根据对索引的优化规则，写出高效sql

2. 如何建立sql

- 对于单表操作：我们去尝试建立适合的索引字段;
- 对于联表操作：left join我们索引建右边；反之左边，我测试过的;
- 对于order by： 排序字段要走索引;
- 对于group by: group by的前提是先排序在分组，最好走索引;

7.2 案例分析



```
mysql> explain select * from test03 where c1='a1' and c2='a2' and c4='a4' order by c3;
+----+-------------+--------+------+---------------+-------------+---------+-------------+------+-------------+
| id | select_type | table  | type | possible_keys | key         | key_len | ref         | rows | Extra       |
+----+-------------+--------+------+---------------+-------------+---------+-------------+------+-------------+
|  1 | SIMPLE      | test03 | ref  | idx_test03_c1234 | idx_test03_c1234 | 62      | const,const |    1 | Using where |
+----+-------------+--------+------+---------------+-------------+---------+-------------+------+-------------+
1 row in set (0.01 sec)

mysql> explain select * from test03 where c1='a1' and c2='a2' order by c3;
+----+-------------+--------+------+---------------+-------------+---------+-------------+------+-------------+
| id | select_type | table  | type | possible_keys | key         | key_len | ref         | rows | Extra       |
+----+-------------+--------+------+---------------+-------------+---------+-------------+------+-------------+
|  1 | SIMPLE      | test03 | ref  | idx_test03_c1234 | idx_test03_c1234 | 62      | const,const |    1 | Using where |
+----+-------------+--------+------+---------------+-------------+---------+-------------+------+-------------+
1 row in set (0.00 sec)

mysql> explain select * from test03 where c1='a1' and c2='a2' order by c4;
+----+-------------+--------+------+---------------+-------------+---------+-------------+------+-----------------------------+
| id | select_type | table  | type | possible_keys | key         | key_len | ref         | rows | Extra                       |
+----+-------------+--------+------+---------------+-------------+---------+-------------+------+-----------------------------+
|  1 | SIMPLE      | test03 | ref  | idx_test03_c1234 | idx_test03_c1234 | 62      | const,const |    1 | Using where; Using filesort |
+----+-------------+--------+------+---------------+-------------+---------+-------------+------+-----------------------------+
```

5）
explain select * from test03 where c1='a1' and c2='a2' and c4='a4' order by c3;
c3作用在排序而不是查找
6）
explain select * from test03 where c1='a1' and c2='a2' order by c3;
7）
explain select * from test03 where c1='a1' and c2='a2' order by c4;
出现了filesort

8）
8.1
explain select * from test03 where c1='a1' and c5='a5' order by c2,c3;

只用c1一个字段索引，但是c2、c3用于排序,无filesort
8.2
explain select * from test03 where c1='a1' and c5='a5' order by c3,c2;

出现了filesort，我们建的索引是1234，它没有按照顺序来，3 2 颠倒了
9）
explain select * from test03 where c1='a1' and c2='a2' order by c2,c3;
10）
explain select * from test03 where c1='a1' and c2='a2' and c5='a5' order by c2,c3;
用c1、c2两个字段索引，但是c2、c3用于排序,无filesort

explain select * from test03 where c1='a1' and c2='a2' and c5='a5' order by c3,c2;

为排序使用索引

- MySql两种排序方式：文件排序或扫描有序索引排序
- MySql能为排序与查询使用相同的索引

KEY a_b_c (a, b, c)

order by 能使用索引最左前缀
- ORDER BY a
- ORDER BY a,b
- ORDER BY a, b, c
- ORDER BY a DESC, b DESC, c DESC

如果WHERE使用索引的最左前缀定义为常量，则order by能使用索引
- WHERE a = const ORDER BY b, c
- WHERE a = const AND b = const ORDER BY c
- WHERE a = const ORDER BY b, c
- WHERE a = const AND b > const ORDER BY b, c

不能使用索引进行排序
- ORDER BY a ASC, b DESC, c DESC /* 排序不一致 */
- WHERE g = const ORDER BY b, c /* 丢失a索引 */
- WHERE a = const ORDER BY c /* 丢失b索引 */
- WHERE a = const ORDER BY a, d /* d 不是索引的一部分 */
- WHERE a in (...) ORDER BY b, c /* 对于排序来说，多个相等条件也是范围查询 */

8.mysql如何保证事务

8.1 没有水平的回答

mysql如何保证事务的回滚与撤销：单纯而言：日志保证的：undo.log日志是用来保证事务的撤销和回滚的，redo.log日志是用来恢复数据库的，bin.log是用来为数据库主从架构来同步数据服务的；

8.2 有含金量至少得说工作流程

- 上面的回答已经说出了作用
- 工作机制：我们在对mysql进行操作的时候，比如更新语句，我们的mysql会先进行磁盘io，从磁盘读取一页的16kb的页数据，在修改的时候mysql会将这页数据先放在buff pool中，怎么放呢，buffpool缓冲区就相当于一个数组一样，每个数组元素存的是页数据，怎么找呢？我们会维护两个链表，一个叫flush链表，一个叫free链表，链表的每一个元素都叫一个控制块，对于free链表而言，他的每一个控制块连接到对应的一个空闲页，然后我们的页数据会根据链表去找对应的空闲位置进行存放，存放完成之后，我们的flush链表表示待同步的链表，每一个控制块连接的是对应的要更新的页数据，这样我们就能够快速根据链表找到那些页是要同步到磁盘io的。
- 接着上面的说：如果当我们的buff pool已经存满了，还需要向buffpool中存也数据，怎么办，我们的buff pool是由自己的淘汰算法的,这个时候buffpool会维护一个叫lru的链表，听名字就最近最少用的淘汰算法，怎么工作，这个链表也是有控制块的，每个控制块代表的是最近最常用的页数据，如果有新的页数来时，就用头插法插入控制块，淘汰的时候就是链表最尾部的，但其实是有问题的，如果这个控制块存的都是我们经常用的业务数据，结果不好，因为一条全表扫描的sql导致我们大量磁盘io页数据，如果是那样，我们链表的热点数据直接被大换血，所以实际上lru链表分为两个区域：一个叫热数据区域，一个叫冷数据区域，热数据区域针对于业务数据的淘汰，冷数据区域针对于非业务的数据，怎么判断呢是有一个判断算法，当控制块添加的速度小于1ms的时候，会被认为你是一个全表扫描的恶意sql反之，认为是业务数据。
- 说了一大堆，终于到了正题，下来就是要将buff pool中修改的数据同步到磁盘，当然不是直接同步的，是先会持久化一个redo.log日志文件，这个文件中记录的是我们对那一页数据那一条记录操作的命令，用户防止直接同步到磁盘页数据，mysql宕机或者其他故障，引起数据丢失，还提高效率，在redo.log而言，在之前还有undo.log日志，用于存储反向操作语句，保证事务恢复的，在后面还有bin.log日志，用于存放执行的命令同步主从文件的。

9.mvcc机制如何实现

mvcc机制的介绍

1. 事务请求和非事务请求：事务请求（更改数据的请求）--> 事务会产生事务id,非事务不会产生事务id

2. 版本链：mvvc底层会维护一个版本链：三部分：数据部分，事务id部分, 指针部分,只有事务请求记录才会在版本链中维护，对象是全库
3. readview: 由非事务请求产生，也就是查询语句,他的构成是由未提交数据的事务id数组和已经提交的最大事务id组成
4. mvcc机制的规则：

- readview中的数组最小事务id叫 min_id,最大已提交的事务id叫 max_id
- min_id和max_id将事务分为三个区域：<min_id的叫已提交事务；大于max_id叫未开始事务；两者之间叫有已提交和未提交部分
- 未开始事务和未提交事务都是不可见的，已提交是可见的
- 怎么比：我举个例子：我们执行一条查询的sql会产生一个readview,遍历版本链的事务id,id落在 未开始代表不可见继续遍历，落在已经提交区域，直接返回这条记录，如果在中间区域，需要查事务id在不在readview的未提交事务id数组中，在不可见，遍历下一个，不在就返回记录
- 删除语句：我们会在版本链的记录中维护一个标识，true,遍历先判断是否为true如果是，就忽略这条记录，遍历下一条。
- readview它是和隔离级别有关的：mysql是可重复读，所以在一次中，会沿用上一次的readview

10.mysql引擎有那些

mysql的引擎其实有很多种，我最熟悉的就是inndb和mysiam两种；

inndb和mysiam的区别：

- inndb 行级锁，并发高，粒度小，支持事务，数据结构方面存储为两个文件 frm（结构） 和 idb（索引和数据），统计的时候会全表扫描
- mysiam 表级锁 ，并发小，粒度大，不支持事务，数据结构方面存储为三个文件 frm(结构) 和 myi 和myd ,统计很快，因为他维护了一个表记录总条数的变量。

11.mysql中有那些锁

1. mysql锁的分类

- 按照类型划分：共享锁（读锁），排他锁 （写锁）
- 按照粒度划分：表级锁，行级锁，页级锁，间隙锁

2. mysql锁的详细介绍

- 表级锁：在mysiam引擎中，得到了实现，粒度大，并发低，速度慢；假如我们在使用mysiam引擎作为案例说：用两个终端：一个终端我们给某一张表上加个读锁；对于加锁的终端而言：他只能对当前表进行读取，不可写，不可操作其他表；对于其他终端而言，可以读，可以操作其他表，但如果对

锁的表进行写操作会被阻塞，只有等加锁终端释放掉，然后自动执行；如果加锁终端加的是写锁，自己可对其进行读写操作，不得操作其他表；其他终端读写此表都会被阻塞；总而言之：读锁就是读读可以共享，不可写；写锁是写读互斥，写写互斥；

- 行级锁：默认在innodb引擎中体现了，粒度小，锁的对象是行级别，并发高，效率快；支持事务；行级锁而言，就是我对当前sql进行事务操作的时候，其他进程或者终端只能被阻塞；案例演示的时候：我们第一步要关闭自动提交，然后通过两个终端做演示是最好的；