MYSQL面试

基础知识

1、为什么要使用数据库

优点：

- 存取速度快
- 数据永久保存，使用SQL语句，查询方便效率高，管理数据方便

缺点：

- 数据不能永久保存
- 速度比内存操作慢，频繁的IO操作

2、MySQL的binlog有有几种录入格式？分别有什么区别?

有三种格式：statement，row和mixed

- statement模式下，每一条会修改数据的sql都会记录在binlog中。不需要记录每一行的变化，减少了binlog日志量，节约了IO，提高性能。由于sql的执行是有上下文的，因此在保存的时候需要保存相关的信息，同时还有一些使用了函数之类的语句无法被记录复制
- row级别下，不记录sql语句上下文相关信息，仅保存哪条记录被修改。记录单元为每一行的改动，基本是可以全部记下来但是由于很多操作，会导致大量行的改动(比如alter table)，因此这种模式的文件保存的信息太多，日志量太大
- mixed，一种折中的方案，普通操作使用statement记录，当无法使用statement的时候使用row。

> 此外新版的MySQL中对row级别也做了一些优化，当表结构发生变化的时候，会记录语句而不是逐行记录，Mysql默认是用的是row的格式，可以通过show variables like 'binlog_format'命令查看binlog的格式

3、Mysql有哪些数据类型

- 整数类型，包括tinyint、smallint、mediumint、int、bigint，分别表示1字节、2字节、3字节、4字节、8字节。1个字节长度等于8位。长度分别为4、6、9、11、20。
- 实数类型，包括flot、double、decimal。
  decimal可以用于存储比bigint还大的整型，能存储精确的小数。
  而flot和double是有取值范围的，并支持使用标准的浮点进行近似计算。
  计算时flot和double相比decimal效率更高一些，DECIMAL你可以理解成是用字符串进行处理
- 字符串类型，包括varchar、char、text、blobvarchar用于存储可变长字符串，它比定长类型更节省空间。varchar使用额外1或2个字节存储字符串长度。列长度小于255字节时，使用1字节表示，否则使用2字节表示。

4、varchar和char的区别

char的特点

- char表示定长字符串，长度是固定的
- 如果插入数据的长度小于char的固定长度时，则用空格填充

- 因为长度固定，所以存取速度要比varchar快很多，甚至能快50%，但正因为其长度固定，所以会占据多余的空间，是空间换时间的做法
- 对于char来说，最多能存放的字符个数为255，和编码无关

varchar的特点

- varchar表示可变长字符串，长度是可变的
- 插入的数据是多长，就按照多长来存储
- varchar在存取方面与char相反，它存取慢，因为长度不固定，但正因如此，不占据多余的空间，是时间换空间的做法
- 对于varchar来说，最多能存放的字符个数为65532总之，结合性能角度（char更快）和节省磁盘空间角度（varchar更小），具体情况还需具体来设计数据库才是妥当的做法

5、varchar(50)中50的含义?

最多存放50个字符，varchar(50)和(200)存储hello所占空间一样，但后者在排序时会消耗更多内存，因为order by col采用fixed_length计算col长度(memory引擎也一样)。在早期 MySQL 版本中， 50 代表字节数，现在代表字符数

6、float和double的区别是什么?

- float类型数据可以存储至多8位十进制数，并在内存中占4字节
- double类型数据可以存储至多18位十进制数，并在内存中占8字节

7、int(20)中20的涵义

是指显示字符的长度。20表示最大显示宽度为20，但仍占4字节存储，存储范围不变；不影响内部存储，只是影响带 zerofill 定义的 int 时，前面补多少个 0，易于报表展示

8、union和union all的区别

- 如果使用UNION ALL，不会合并重复的记录行
- 效率 UNION 高于 UNION ALL

9、Mysql存储引擎有哪些?

- Innodb引擎
- MyISAM引擎
- MEMORY引擎
- MRG_MYISAM引擎
- BLACKHOLE引擎
- CSV引擎
- ARCHIVE引擎
- PERFORMANCE_SCHEMA引擎
- FEDERATED引擎

常用的引擎有一下三种：

- Innodb引擎：Innodb引擎提供了对数据库ACID事务的支持。并且还提供了行级锁和外键的约束。它的设计的目标就是处理大数据容量的数据库系统。
- MyISAM引擎：不提供事务的支持，也不支持行级锁和外键。
- MEMORY引擎：所有的数据都在内存中，数据的处理速度快，但是安全性不高。

10、MyISAM和InnoDB区别

- InnoDB主键索引索引是聚簇索引，MyISAM索引是非聚簇索引。
- InnoDB的主键索引的叶子节点存储着数据，因此主键索引非常高效。

- MyISAM索引的叶子节点存储的是行数据地址，需要再寻址一次才能得到数据。
- InnoDB非主键索引的叶子节点存储的是主键和其他带索引的列数据，因此查询时做到覆盖索引会非常高效。
- Innodb支持外键和事务，MyISAM不支持
- InnoDB支持行级锁、表锁、锁定力度小并发能力高，MyISAM只支持表锁
- select MyISAM性能更优，insert、update、delete InnoDB性能更优
- InnoDB支持哈希索引(InnoDB支持的哈希索引是自适应的，InnoDB存储引擎会根据表的使用情况自动为表生成哈希索引，不能认为干预是否在一张表中生成哈希索引)，MyISAM不支持
- MyISAM支持全文索引，InnoDB不支持全文索引

索引(基于Mysql InnoDb讨论)

1、什么是索引?

索引是一种数据结构。数据库索引，是数据库管理系统中一个排序的数据结构，以协助快速查询、更新数据库表中数据。索引的实现通常使用B树及其变种B+树。更通俗的说，索引就相当于目录。为了方便查找书中的内容，通过对内容建立索引形成目录。索引是一个文件，它是要占据物理空间的。

> 只要能回答到索引是一种数据结构，能够协助我们快速查询和更新表中数据就OK

2、索引有哪些优缺点?

索引的有点：

- 可以大大加快数据的检索速度。这也是创建索引的最主要原因。
- 通过使用索引，可以在查询的过程中，使用优化隐藏器，提高系统的性能。

索引的缺点：

- 时间方面：创建索引和维护索引要耗费时间，对表中的数据进行增加、删除和修改的时候，索引也要动态的维护，会降低增/改/删的执行效率
- 空间方面：索引需要占物理空间。

3、索引有哪几种类型?

- 主键索引：数据列不允许重复，不允许为NULL，一个表只能有一个主键。
- 唯一索引：数据列不允许重复，允许为NULL值，一个表允许多个列创建唯一索引。
- 普通索引：基本的索引类型，没有唯一性的限制，允许为NULL值。
- 全文索引：是目前搜索引擎使用的一种关键技术。(InnoDB不支持全文索引)

4、索引的数据结构有哪些?

B+树(多叉树)、Hash索引

5、说一说B+树的数据结构

B+树又名平衡多路查找树(多叉树)。B+树的每个节点可以表示更多的信息，因此树的高度相比二叉树更加矮胖，这在磁盘的查找数据的过程中，可以减少磁盘IO的次数，从而提升查找速度。B+树的数据是存储在叶子节点的，节点上只存索引。因为Mysql是按页存储的，所以节点可以分更多的叉，相比B树相同的高度下能够存更多的数据，也有利于更快的缩小查询找范围。B+树所有叶子节点的数据结构是有序链表，因此当需要进行一次全数据遍历的时候，B+树只需要使用O(logN)时间找到最小的一个节点，然后通过链进行O(N)的顺序遍历即可。而B树则需要对树的每一层进行遍历，这会需要更多的内存置换次数，因此也就需要花费更多的时间

6、说一说Hash索引的数据结构

Hash索引的底层是一个哈希表，哈希表是一种以键-值（key-value）存储数据的结构，我们只要输入待查找的键即 key，就可以找到其对应的值即Value。哈希的思路很简单，把值放在数组里，用一个哈希函数把key换算成一个确定的位置，然后把 value 放在数

组的这个位置。不可避免地，多个key值经过哈希函数的换算，会出现同一个值的情况。处理这种情况的一种方法是，拉出一个链表。

7、B+树索引和Hash索引的区别

哈希表这种结构只适用于等值查询，不支持范围查询、不支持联合索引的最左匹配原则、不支持索引进行排序。由于是一次定位数据，不想B+树索引需要从根节点到叶子节点，需要经过多次IO访问，所以它的检索效率远高于B+树。但是如果Hash冲突严重，性能不一定高于B+树。

8、数据库为什么使用B+树而不是B树

- B树只适合随机检索，而B+树同时支持随机检索和顺序检索
- B+树空间利用率更高，可减少I/O次数，磁盘读写代价更低。一般来说，索引本身也很大，不可能全部存储在内存中，因此索引往往以索引文件的形式存储的磁盘上。这样的话，索引查找过程中就要产生磁盘I/O消耗。B+树节点只存索引，而B树节点上即存索引也存了数据，所以B+树每页能容纳的结点中关键字数量更多，一次性读入内存中可以查找的关键字也就越多，相对的，IO读写次数也就降低了，而IO读写次数是影响索引检索效率的最大因素。同样的B+树因为每页存的索引越多，那么分叉也就越多，在相同的高度下存的数据也就越多
- B+树的查询效率更加稳定。B树搜索有可能会在非叶子结点结束，越靠近根节点的记录查找时间越短，只要找到关键字即可确定记录的存在，其性能等价于在关键字全集内做一次二分查找。而在B+树中，顺序检索比较明显，随机检索时，任何关键字的查找都必须走一条从根节点到叶节点的路，所有关键字的查找路径长度相同，导致每一个关键字的查询效率相当
- B树在提高了磁盘IO性能的同时并没有解决元素遍历的效率低下的问题。B+树的叶子节点使用指针顺序连接在一起，只要遍历叶子节点就可以实现整棵树的遍历。而且在数据库中基于范围的查询是非常频繁的，而B树不支持这样的操作。
- 增删文件（节点）时，效率更高。因为B+树的叶子节点包含所有关键字，并以有序的链表结构存储，这样可很好提高增删效率。

9、什么是最左匹配原则

mysql会一直向右匹配直到遇到范围查询(>、<、between、like)就停止匹配，比如a 1 = and b="2" c> 3 and d = 4如果建立(a,b,c,d)顺序的索引，d是用不到索引的，如果建立(a,b,d,c)的索引则都可以用到索引，a,b,d的顺序可以任意调整。=和in可以乱序，比如a = 1 and b = 2 and c = 3 建立(a,b,c)索引可以任意顺序，mysql的查询优化器会帮你优化成索引可以识别的形式

10、什么是聚簇索引?

将数据和索引放在一块，找到了索引也就找到了数据，这就是聚簇索引，在InnoDB中主键就是聚簇索引。

11、什么是非聚簇索引?

在InnoDB中，在聚簇索引之上创建的索引称之为辅助索引(非聚簇索引)，辅助索引访问数据总是需要第二次查找(俗称回表)，像复合索引，前缀索引，唯一索引，叶子节点存储的不再是行的物理位置，而是主键值。

12、非聚簇索引就一定会回表查询吗?

不一定，如果要查询的字段全部命中索引，那么就不必再进行回表查询。这也就是覆盖索引。

事务

1、什么是数据库事务?

事务是一个不可分割的数据操作序列，也是数据库并发控制的基本单位，其执行的结果必须使数据库从一种一致性状态变到另一种一致性状态。事务是逻辑上的一组操作，要么都执行，要么都不执行

2、事务的四大特性(ACID)

- 原子性(A)：事务是最小的执行单位，不允许分割。事务的原子性确保动作要么全部完成，要么完全不起作用
- 一致性(c)：执行事务前后，数据保持一致，多个事务对同一个数据读取的结果是相同的
- 隔离性(I)：并发访问数据库时，一个用户的事务不被其他事务所干扰，各并发事务之间数据库是独立的

- 持久性(D)：一个事务被提交之后。它对数据库中数据的改变是持久的，即使数据库发生故障也不应该对其有任何影响

3、Mysql事务的隔离级别，默认的隔离级别是什么?

Mysql默认隔离级别是：可重复度

- READ-UNCOMMITTED(读取未提交)：最低的隔离级别，允许读取尚未提交的数据变更，可能会导致脏读、幻读或不可重复读
- READ-COMMITTED(读取已提交)：允许读取并发事务已经提交的数据，可以阻止脏读，但是幻读或不可重复读仍有可能发生
- REPEATABLE-READ(可重复读)：对同一字段的多次读取结果都是一致的，除非数据是被本身事务自己所修改，可以阻止脏读和不可重复读，但幻读仍有可能发生
- SERIALIZABLE(可串行化)： 最高的隔离级别，完全服从ACID的隔离级别。所有的事务依次逐个执行，这样事务之间就完全不可能产生干扰，也就是说，该级别可以防止脏读、不可重复读以及幻读

> 在实现上，数据库里面会创建一个视图，访问的时候以视图的逻辑结果为准。在“可重复读”隔离级别下，这个视图是在事务启动时创建的，整个事务存在期间都用这个视图。在“读提交”隔离级别下，这个视图是在每个 SQL 语句开始执行的时候创建的。这里需要注意的是，“读未提交”隔离级别下直接返回记录上的最新值，没有视图概念；而“串行化”隔离级别下直接用加锁的方式来避免并行访问。

4、什么是脏读？幻读？不可重复读?

- 脏读：某个事务已更新了一份数据，另一个事务在此时又读取到了这一份已更新的数据。但是由于某些原因导致前一个事务回滚了，则后一个事务所读取的数据就是不正确的，这就是脏读。
- 不可重复读：在一个事务的两次查询之中数据不一致，这可能是两次查询过程中有其他事务修改了原有的数据
- 幻读：在可重复读隔离级别下，普通的查询是快照读，是不会看到别的事务插入的数据的，因此，幻读在当前读下才会出现。幻读仅指新插入的行。举例说明：幻读并不是说两次读取的结果集不同，幻读侧重的方面是某一次查询操作得到的结果无法支撑后续的业务操作。例如第一次查询时候记录不存在，但是在执行插入的时候却发现记录已经存在，此时就像发生了幻读。

5、Mysql的读已提交和可重复读底层是怎样实现的?

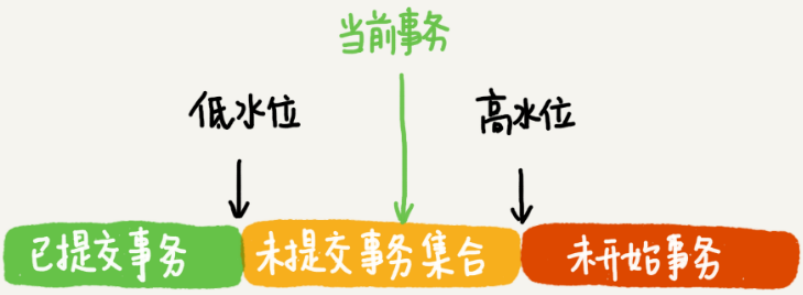
主要是通过多版本并发控制(MVCC)实现，在实现MVCC时用到了一致性读视图。

6、说一说MVCC是如何工作的?

InnoDB 里面每个事务有一个唯一的事务 ID，叫作 transaction id。它是在事务开始的时候向 InnoDB 的事务系统申请的，是按申请顺序严格递增的。而每行数据也都是有多个版本的。每次事务更新数据的时候，都会生成一个新的数据版本，并且把 transaction id 赋值给这个数据版本的事务 ID，记为 row trx_id。同时，旧的数据版本要保留，并且在新的数据版本中，能够有信息可以直接拿到它。也就是说，数据表中的一行记录，其实可能有多个版本 (row)，每个版本有自己的 row trx_id。

按照可重复读的定义，一个事务启动的时候，能够看到所有已经提交的事务结果。但是之后，这个事务执行期间，其他事务的更新对它不可见。因此，一个事务只需要在启动的时候声明说，“以我启动的时刻为准，如果一个数据版本是在我启动之前生成的，就认；如果是我启动以后才生成的，我就不认，我必须要找到它的上一个版本”。当然，如果“上一个版本”也不可见，那就得继续往前找。还有，如果是这个事务自己更新的数据，它自己还是要认的。

在实现上， InnoDB 为每个事务构造了一个数组，用来保存这个事务启动瞬间，当前正在“活跃”的所有事务 ID。“活跃”指的就是，启动了但还没提交。数组里面事务 ID 的最小值记为低水位，当前系统里面已经创建过的事务 ID 的最大值加 1 记为高水位。`这个视图数组和高水位，就组成了当前事务的一致性视图（read-view）。`而数据版本的可见性规则，就是基于数据的 row trx_id 和这个一致性视图的对比结果得到的。这个视图数组把所有的 row trx_id 分成了几种不同的情况。



这样，对于当前事务的启动瞬间来说，一个数据版本的 row trx_id，有以下几种可能：

- 如果落在绿色部分，表示这个版本是已提交的事务或者是当前事务自己生成的，这个数据是可见的；
- 如果落在红色部分，表示这个版本是由将来启动的事务生成的，是肯定不可见的；
- 如果落在黄色部分，那就包括两种情况
  - 若 row trx_id 在数组中，表示这个版本是由还没提交的事务生成的，不可见
  - 若 row trx_id 不在数组中，表示这个版本是已经提交了的事务生成的，可见。

下面举一个例子来说明，下面是一个只有两行数据的表的初始化语句：

```
CREATE TABLE `t` (
    `id` int(11) NOT NULL,
    `k` int(11) DEFAULT NULL,
    PRIMARY KEY (`id`)
  ) ENGINE=InnoDB;
  insert into t(id, k) values(1,1),(2,2);
```

事务A、B、C的执行流程如下

事务A	事务B	事务C
start transaction with consistent snapshot;		
	start transaction with consistent snapshot;	
		update t set k=k+1 where id=1;
	update t set k=k+1 where id=1; select k from t where id=1;	
select k from t where id=1; commit;		
	commit;	

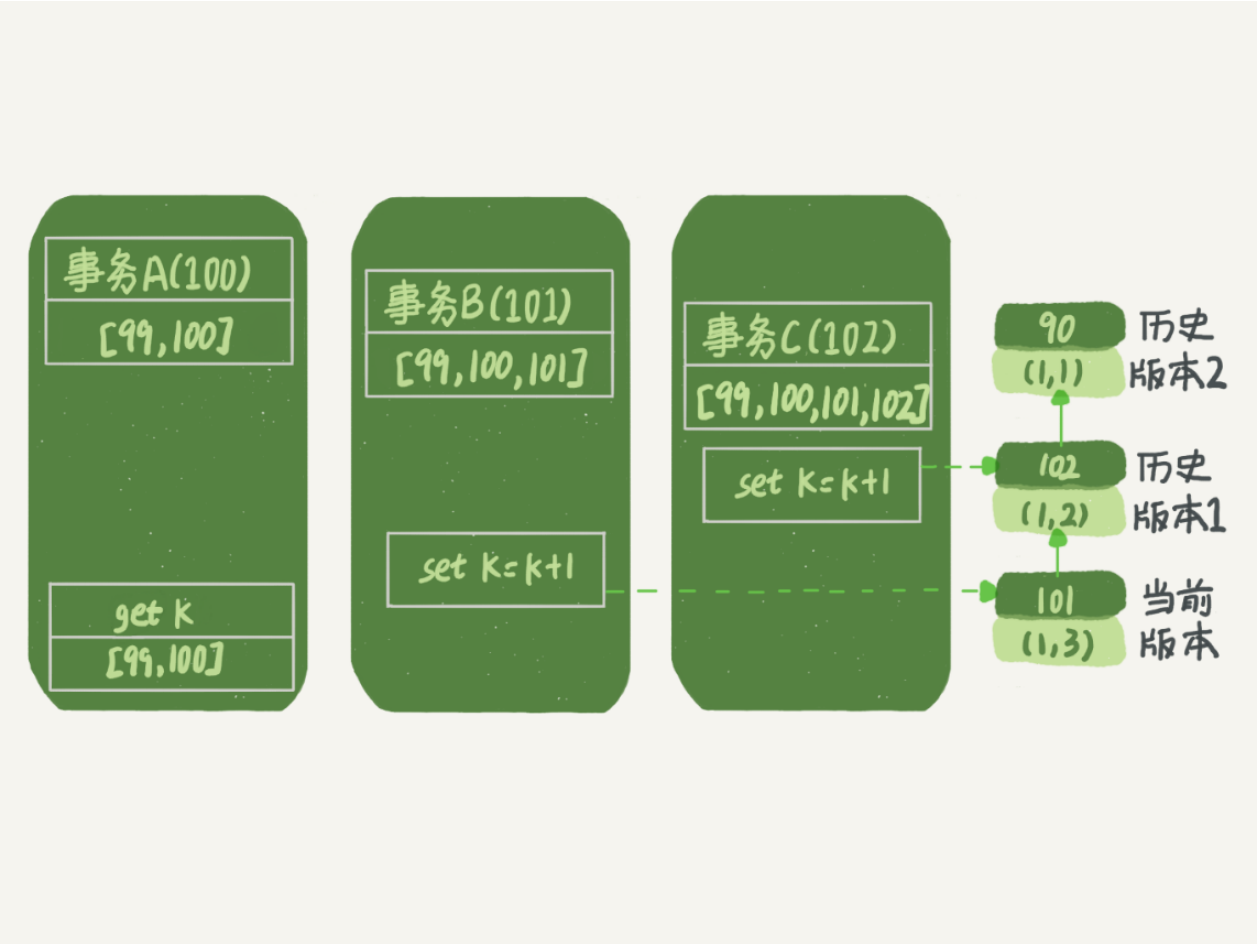
> 这里事务的启动时机需要注意一下：begin/start transaction 命令并不是一个事务的起点，在执行到它们之后的第一个操作 InnoDB 表的语句，事务才真正启动。如果你想要马上启动一个事务，可以使用 start transaction with consistent snapshot 这个命令

假设：

- 1、事务 A 开始前，系统里面只有一个活跃事务 ID 是 99；
- 2、事务 A、B、C 的版本号分别是 100、101、102，且当前系统里只有这四个事务；
- 3、三个事务开始前，(1,1）这一行数据的 row trx_id 是 90。

这样，事务 A 的视图数组就是[99,100], 事务 B 的视图数组是[99,100,101], 事务 C 的视图数组是[99,100,101,102]

通过一下逻辑图来分析：



从图中可以看到，第一个有效更新是事务 C，把数据从 (1,1) 改成了 (1,2)。这时候，这个数据的最新版本的 row trx_id 是 102，而 90 这个版本已经成为了历史版本。

第二个有效更新是事务 B，把数据从 (1,2) 改成了 (1,3)。这时候，这个数据的最新版本（即 row trx_id）是 101，而 102 又成为了历史版本。

你可能注意到了，在事务 A 查询的时候，其实事务 B 还没有提交，但是它生成的 (1,3) 这个版本已经变成当前版本了。但这个版本对事务 A 必须是不可见的，否则就变成脏读了。

好，现在事务 A 要来读数据了，它的视图数组是[99,100]。当然了，读数据都是从当前版本读起的。所以，事务 A 查询语句的读数据流程是这样的：

- 找到 (1,3) 的时候，判断出 row trx_id=101，比高水位大，处于红色区域，不可见；
- 接着，找到上一个历史版本，一看 row trx_id=102，比高水位大，处于红色区域，不可见；
- 再往前找，终于找到了（1,1），它的 row trx_id=90，比低水位小，处于绿色区域，可见。

这样执行下来，虽然期间这一行数据被修改过，但是事务 A 不论在什么时候查询，看到这行数据的结果都是一致的，所以我们称之为一致性读。

一个数据版本，对于一个事务视图来说，除了自己的更新总是可见以外，还有三种情况：

- 1、版本未提交，不可见；
- 2、版本已提交，但是是在视图创建后提交的，不可见；
- 3、版本已提交，而且是在视图创建前提交的，可见。

现在，我们用这个规则来判断上图中的查询结果，事务 A 的查询语句的视图数组是在事务 A 启动的时候生成的，这时候：

- (1,3) 还没提交，属于情况 1，不可见；
- (1,2) 虽然提交了，但是是在视图数组创建之后提交的，属于情况 2，不可见；

- (1,1) 是在视图数组创建之前提交的，可见。

> InnoDB 的行数据有多个版本，每个数据版本有自己的 row trx_id，每个事务或者语句有自己的一致性视图。普通查询语句是一致性读，一致性读会根据 row trx_id 和一致性视图确定数据版本的可见性。
>
> - 对于可重复读，查询只承认在事务启动前就已经提交完成的数据；
> - 对于读提交，查询只承认在语句启动前就已经提交完成的数据；

Mysql锁

1、对Mysql的锁有了解吗?

当数据库有并发事务的时候，可能会产生数据的不一致，这个时候就需要一些机制来保证访问的次序，像Mysql的锁就是这样一种机制。

2、你知道Mysql有哪些锁吗(按照锁的粒度分)？它们有什么区别?

在Mysql中，可以按照锁的粒度把数据锁分为：行级锁、表级锁、页级锁

- 行级锁：行级锁是Mysql中锁定粒度最细的一种锁，表示只针对当前操作的行进行加锁。行级锁能大大减少数据库操作的冲突。其加锁粒度最小，但加锁的开销也最大。
  特点：开销大，加锁慢；会出现死锁；锁定粒度最小，发生锁冲突的概率最低，
  并发度也最高
- 表级锁：表级锁是MySQL中锁定粒度最大的一种锁，表示对当前操作的整张表加锁，它实现简单，资源消耗较少，被大部分MySQL引擎支持。最常使用的MyISAM与InnoDB都支持表级锁定
  特点：开销小，加锁快；不会出现死锁；锁定粒度大，发出锁冲突的概率最高，并发度最低
- 页级锁：页级锁是MySQL中锁定粒度介于行级锁和表级锁中间的一种锁。表级锁速度快，但冲突多，行级冲突少，但速度慢。所以取了折衷的页级，一次锁定相邻的一组记录
  特点：开销和加锁时间界于表锁和行锁之间；会出现死锁；锁定粒度界于表锁和行锁之间，并发度一般
- 全局锁：全局锁就是对整个数据库实例枷锁。当你需要让真个库处于只读状态的时候，可以通过命令Flush tables with read lock (FTWRL)加锁，之后其他线程的语句会被阻塞：数据更新语句（数据的增删改）、数据定义语句（包括建表、修改表结构等）和更新类事务的提交语句。
  全局锁的典型使用场景是做全库逻辑备份

3、按照锁的类别分有哪些锁?

从锁的类别上分，有共享锁和排它锁

- 共享锁(S)：又叫做读锁。 当用户要进行数据的读取时，对数据加上共享锁。共享锁可以同时加上多个
- 排他锁(X：又叫做写锁。 当用户要进行数据的写入时，对数据加上排他锁。排他锁只可以加一个，他和其他的排他锁，共享锁都相斥
- 意向共享锁(IS)：事务打算给数据行加共享锁(S)，此前必须先取得该表的意向共享锁(IS)
- 意向排他锁(IX)：事务打算给数据行加排他锁(X)，此前必须先取得该表的意向排他锁(IX)

> 如果能回答到意向锁，应该对锁的了解还是比较深入了

	读锁	写锁
读锁	兼容	冲突
写锁	冲突	冲突

4、Mysql中InnoDB引擎的行锁如何手动加?

InnoDB是基于索引来完成行锁

例如：select * from table where id = 1 for update; for update 可以根据条件来完成行锁锁定，并且是有索引键的列，如果id不是索引键，那么InnoDB将完成表锁。for update 加的是排他锁。

5、Mysql中如果想手动加一个共享锁，应该如何实现?

例如：select * from table where id lock in share mode;

6、InnoDB对记录加锁方的式有哪些?

- 记录锁(Record Lock)：锁定一行记录
- 间隙锁(Gap Lock)：锁定一个区间
- 记录锁+间隙锁(Next Lock)：锁定行记录+区间

> 相关知识点：
>
> - InnoDB对行的查询使用的是间隙锁
> - 间隙锁是为了解决幻读问题
> - 当查询的索引包含有唯一属性时，间隙锁降级为行锁
> - 有两种关闭间隙锁的方式：1、将事务隔离级别设置成读已提交 2、将数innodb_locks_unsafe_for_binlog设置为1

7、什么是死锁？怎么解决?

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，从而导致的互相等待。

常见的解决死锁的方法：

- 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，可以大大降低死锁机会
- 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

> Mysql默认有死锁检测，默认为50s

8、如何查看死锁?

执行 show engine innodb status 命令得到的部分输出。这个命令会输出很多信息，有一节 LATESTDETECTED DEADLOCK，就是记录的最后一次死锁信息

系统

1、什么是存储过程？有哪些优缺点?

存储过程是一个预编译的SQL语句，优点是允许模块化的设计，就是说只需要创建一次，以后在该程序中就可以调用多次。如果某次操作需要执行多次SQL，使用存储过程比单纯SQL语句执行要快。

`优点`：

- 存储过程是预编译过的，执行效率高
- 存储过程的代码直接存放于数据库中，通过存储过程名直接调用，减少网络通讯
- 安全性高，执行存储过程需要有一定权限的用户
- 存储过程可以重复使用，减少数据库开发人员的工作量

`缺点`

- 调试麻烦，但是用 PL/SQL Developer 调试很方便！弥补这个缺点
- 移植问题，数据库端代码当然是与数据库相关的。但是如果是做工程型项目，基本不存在移植问题

- 如果在一个程序系统中大量的使用存储过程，到程序交付使用的时候随着用户需求的增加会导致数据结构的变化，接着就是系统的相关问题了，最后如果用户想维护该系统可以说是很难很难、而且代价是空前的，维护起来更麻烦

2、Mysql有哪些触发器?

- Before Insert
- After Insert
- Before Update
- After Update
- Before Delete
- After Delete

3、Sql语句主要分为哪几类?

- 数据定义语言DDL（Data Ddefinition Language）CREATE，DROP，ALTER主要为以上操作 即对逻辑结构等有操作的，其中包括表结构，视图和索引
- 数据查询语言DQL（Data Query Language）SELECT这个较为好理解 即查询操作，以select关键字。各种简单查询，连接查询等 都属于DQL
- 数据操纵语言DML（Data Manipulation Language）INSERT，UPDATE，DELETE
- 数据控制功能DCL（Data Control Language）GRANT，REVOKE，COMMIT，ROLLBACK

4、Sql约束有哪几种?

- NOT NULL: 用于控制字段的内容一定不能为空（NULL）
- UNIQUE: 控件字段内容不能重复，一个表允许有多个 Unique 约束
- PRIMARY KEY: 也是用于控件字段内容不能重复，但它在一个表只允许出现一个
- FOREIGN KEY: 用于预防破坏表之间连接的动作，也能防止非法数据插入外键列，因为它必须是它指向的那个表中的值之一
- CHECK: 用于控制字段的值范围

5、关联查询有哪些?

- 交叉连接（CROSS JOIN）
- 内连接（INNER JOIN）
- 外连接（LEFT JOIN/RIGHT JOIN）
- 联合查询（UNION与UNION ALL）
- 全连接（FULL JOIN）

常见问题

1、如何定位及优化Sql语句性能问题？创建的索引是否使用到？或者怎么才可以早点这条语句运行很慢的原因?

对于低性能的sql语句的定位，最重要的也是最有效的就是使用执行计划。Mysql提供explain命令来查看执行计划。对于查询语句，最重要的优化方式就使用索引。而执行计划就是现实数据库引擎对于Sql语句的执行情况，其中包含了是否使用索引，使用了什么索引，以及使用索引的相关信息。如下图所示：

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	s	(NULL)	index	goods_id,idx_storeid_tagcategory_valid	goods_id	24	(NULL)	12	100.00	Using index
1	SIMPLE	b	(NULL)	eq_ref	PRIMARY	PRIMARY	8	biaoguoworks.s.tag_id	1	100.00	Using index

执行计划包含的信息 `id` 有一组数字组成，表示一个查询中各个子查询的执行顺序

- id相同，则执行顺序由上至下
- id不同，id值越大优先级越高，越先被执行
- id为null时表示一个结果集，不需要使用它查询，常出现在包含union等查询语句中

`select_type` 每个子查询的查询类型

select_type	描述
SIMPLE	不包含任何子查询或union等查询
PRIMARY	包含子查询最外层查询就显示为PRIMARY
SUBQUERY	在select或where语句中包含的查询
DERIVED	from子句中包含的查询
UNION	出现在union后的查询语句中
UNION RESULT	从UNION中获取结果集

`table` 查询的数据表

`partitions` 表分区、表创建的时候可以指定通过哪个列表进行表分区

`type` 访问类型

- ALL 扫描全表数据
- index 遍历索引
- range 索引范围查找
- ref 多表查询时，根据非唯一索引进行查询
- eq_ref 多表关联查询时，根据唯一非空索引行进查询的情况
- system/const 当查询最多匹配一行时，常出现在where条件是=的情况。system是const的一种特殊情况，既表本身只有一行数据的情况

`possible_keys` 可能使用的索引，注意不一定会使用。查询涉及到的字段上若存在索引，则该索引将被列出来。当该列为 NULL 时就要考虑当前的SQL是否需要优化了

`key` 显示Mysql在查询中实际使用的索引，若没有使用索引，显示为NULL

`key_len` 索引长度

`ref` 表示上述表的连接匹配条件，即哪些列或常量被用于查找索引列上的值

`rows` 返回估算的结果集数目，并不是一个准确的值

`extra` 的信息非常丰富，常见的有：

- Using index 使用覆盖索引
- Using where 使用了where子句来过滤结果集
- Using filesort 使用文件排序，使用非索引列进行排序时出现，非常消耗性能、尽量优化
- Using temporary 使用了临时表

2、关心过业务系统里面的sql耗时吗？统计过慢查询吗？对慢查询都是怎么优化的？

在业务系统中，除了使用主键进行的查询，其他的我都会在测试库上测试其耗时，慢查询的统计主要由运维在做，会定期将业务中的慢查询反馈给我们。慢查询的优化首先要搞明白慢的原因是什么？ 是查询条件没有命中索引？是load了不需要的数据列？还是数据量太大？所以优化也是针对这三个方向来的

- 首先分析语句，看看是否load了额外的数据，可能是查询了多余的行并且抛弃掉了，可能是加载了许多结果中并不需要的列，对语句进行分析以及重写
- 分析语句的执行计划，然后获得其使用索引的情况，之后修改语句或者修改索引，使得语句可以尽可能的命中索引
- 如果对语句的优化已经无法进行，可以考虑表中的数据量是否太大，如果是的话可以进行横向或者纵向的分表

3、主键使用自增ID还是UUID？

因为在InnoDB存储引擎中，主键索引是作为聚簇索引存在的，也就是说，主键索引的B+树叶子节点上存储了主键索引以及全部的数据(按照顺序)，如果主键索引是自增ID，那么只需要不断向后排列即可，如果是UUID，由于大小是不确定的，所以在插入的时候，会造成数据的移动或者是也分裂，然后导致产生很多的内存碎片，进而造成插入性能的下降

4、自增主键为什么不是连续的?

InnoDB引擎的自增值，其实是保存在内存中的，并且到了 MySQL 8.0 版本后，才有了“自增值持久化”的能力，也就是才实现了“如果发生重启，表的自增值可以恢复为 MySQL 重启前的值”，具体情况是

- 在 MySQL 5.7 及之前的版本，自增值保存在内存里，并没有持久化。每次重启后，第一次打开表的时候，都会去找自增值的最大值 max(id)，然后将 max(id)+1 作为这个表当前的自增值。举例来说，如果一个表当前数据行里最大的 id 是 10，AUTO_INCREMENT=11。这时候，我们删除 id=10 的行，AUTO_INCREMENT 还是 11。但如果马上重启实例，重启后这个表的 AUTO_INCREMENT 就会变成 10。也就是说，MySQL 重启可能会修改一个表的 AUTO_INCREMENT 的值

5、如何避免长事务?

从应用端开发来看：

- 确认是否使用了手动提交，如果设置了手动提交，先将它改为自动提交
- 确认是否有不必要的只读事务。有些框架会习惯不管什么语句先用 begin/commit 框起来。我见过有些是业务并没有这个需要，但是也把好几个 select 语句放到了事务中。这种只读事务可以去掉
- 业务连接数据库的时候，根据业务本身的预估，通过 SET MAX_EXECUTION_TIME 命令，来控制每个语句执行的最长时间，避免单个语句意外执行太长时间

从数据库端来看：

- 监控 information_schema.Innodb_trx 表，设置长事务阈值，超过就报警 / 或者 kill
- Percona 的 pt-kill 这个工具不错，推荐使用
- 在业务功能测试阶段要求输出所有的 general_log，分析日志行为提前发现问题
- 如果使用的是 MySQL 5.6 或者更新版本，把 innodb_undo_tablespaces 设置成 2