常见问题总结

什么是MySQL

一种关系型数据库，常用与Java企业级开发，因为MySql是开源免费的，并且方便扩展。阿里巴巴使用大量的MySQL，默认端口是3306；

存储引擎

查看MySQL提供的所有存储引擎；

MySQL默认的存储的引擎是InnoDB，在5.7版本中所有的存储引擎中只有InnoDB是事务性的存储引擎，也就是只有InnoDB支持事务；

查看MySQL当前默认的存储引擎；

```
show variables like '%storage_engine%';
```

查看表的存储引擎

```
show table status like "table_name";
```

MyISAM和InnoDB区别

MyISAM是MySQL以前的默认引擎，5.5之前。虽然性能极佳，提供了大量的特性，包括全文索引、压缩、空间函数等，但是MyISAM不支持事务和行级锁，而且最大缺陷是崩溃之后无法安全恢复。不过5.5之后引入了InnoDB事务性数据库引擎，MySQL5.5版本之后默认的存储引擎是InnoDB。

大部分情况使用InnoDB存储引擎库，某些情况下是可以使用MyISAM合适比如密集读的情况。

两者的对比： **1. 是否支持行级锁**： MyISAM只有表级锁，而InnoDB支持行级锁和表级锁，默认为行级锁；

2. 是否支持事务和崩溃后的安全恢复： MyISAM强调的是性能，每次查询之后具有原子性，骑执行速度比InnoDB类型更快，但是不提供事务支持。但是InnoDB提供事务支持事务，外部键等高级数据库功能。具有事务commit，回滚roolback和崩溃修复能力crash recovery capabilities的事务安全型表；

3. 是否支持外键：MyISAM不支持，而InnoDB支持；

4. 是否支持MVCC： 仅InnoDB支持MVCC，应对高并发事务，MVCC比单纯的加锁更高效；MVCC只在READ COMMITTED和REPEATABLE READ两个隔离级别下工作，MVCC可以使用乐观锁和悲观锁来实现，个数据库中MVCC实现并不统一；

字符集和校对规则 字符集是指的一种从二进制编码到某类字符符号的映射。校对规则是指某种字符集下的排序规则。MySQL中每一种字符集都会对应一系列的校对规则；

MySQL采用的是类似集成的方式，指定字符集的默认值，没个数据库以及每张数据表都有自己的默认值，他们逐层继承。比如：某个库中所有的表的默认字符集将是该数据库所指定的字符集。没有指定字符集的情况下采用默认字符集；

索引

分为 **Btree索引和哈希索引**。对于哈希索引来说，底层的数据结构就是哈希表，因此绝大多数需求是单条记录查询的时候使用哈希索引。但是绝大多数树场景中使用BTree索引；

MyISAM：B+Tree叶节点中大data域存放的数据地址，索引检索的时候，如果指定key存在，则取出其data域中的值，然后以data域中的值读取相应的数据记录；

InnoDB：其数据文件本身就是索引文件。相比于MyISAM，索引和数据文件分离，其表数据文件本身就是按照B+Tree组织的一个索引结构，叶子结点的data域保存了完整的数据记录，这个索引的key是数据表的主键，因此InnoDB文件本身就是一个主索引，其余的索引使辅助索引，辅助索引的data域存储相应记录主键的值而不是地址。

在根据主索引搜索时，直接找到key所在的节点即可取出数据；在根据辅助索引查找时，则需要先取出主键的值，再走一遍主索引。 因此，在设计表的时候，不建议使用过长的字段作为主键，也不建议使用非单调的字段作为主键，这样会造成主索引频繁分裂；

查询缓存的使用

执行查询语句的时候，会先查询缓存，不过8.0之后移除，功能不太实用；

开启查询换成后在同样的查询条件及数据情况下，会直接在缓存中返回结构。查询条件包括查询本身，当前查询的数据库，客户算协议版本号等一些可能影响结果的信息。任何两个查询在任何字符上的不同都不会命中；

缓存虽然能够提升数据库的查询效率但是缓存也带来了额外的开销，每次查询之后都要进行一次缓存操作，失败后还要销毁。

什么是事务?

事务是逻辑上的一组操作，要么都执行要么都不执行； 转账举例，小明转给小红1000元，两个操作，小明减1000，小红加1000，如果发生错误比如系统崩溃，导致小明减1000，小红却未加1000，这样就不对了。事务保证了两个关键操作要么都成功，要么都失败；

1. 原子性：事务是最小的执行单位，不允许分割，事务的原子性保证，动作要么全部完成，要么全部不完成；
2. 一致性：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； **3. 隔离性**：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时独立的； **4. 持久性**: 一个事务被提交之后，对数据库中数据改变是持久的，即使数据库发生故障也应该不受影响；

并发事务带来的问题

脏读：一个事务访问数据库并修改了数据，还没有提交到数据库，此时另外一个事务访问读取了这个数据，然后使用了这个数据，此时这个没有提交的数据可能导致之后的操作是错误的； **丢失修改**：一个事务读取一个数据时，另外一个事务也访问了数据，第一个事务修改了这个数据后，第二个事务也修改了这个数据，这样第一个事务的修改结果就发生了丢失，称之为丢失修改； **不可重复读**：一个事务多次读同一个数据，在这个事务没有结束时，另一个事务也访问了改数据，那么两次读之间，第二个事务可能导致第一个事务两次读取数据不一样； **幻读**：发生在一个事务读取了几行数据，接着另一个事务插入了一些数据，在随后的查询中第一个事务就发现多了一些原本不存在的数据，好像幻觉一样；

不可重复读，主要是第二个事务在第一个事务执行期间对数据的修改，造成第一个事务两次读取之间发生不一样。 幻读：主要强调的是数据的新增和删除；

事务隔离级别有哪些？MySQL的默认隔离级别是？

四个隔离级别：

READ-UNCOMMITTED(读取未提交):最低的隔离级别，允许读取未提交的数据变更，可能导致脏读，幻读和不可重复读； **READ-COMMITTED(读取已提交)**：允许读取并发事务已经提交的的数据，可以阻止脏读，不可阻止幻读不可重复读； **REPEATABLE-READ(可重复读)**：对同一个字段读取的结果是一样的，除非数据是被本身事务自己修改的，可以阻止脏读和不可重复读，幻读仍然可以发生； **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，这样的事务之间就完全不可能发生干扰，**该级别防止脏读，不可重复读和幻读；**

隔离级别	脏读	不可重复读	幻影读
READ-UNCOMMITTED	√	√	√
READ-COMMITTED	×	√	√
REPEATABLE-READ	×	×	√
SERIALIZABLE	×	×	×

InnoDB存储引擎默认支持的是REPEATABLE-READ(可重读),

与SQL标准不同的地方在于InnoDB存储引擎在**REPEATABLE-READ(可重复读)**,事务隔离级别下使用的是Next-key lock锁算法，因此可以避免幻读的产生，这与其他数据库（SQL server）是不同的。所以说InnoDB存储引擎的默认支持的隔离级别是REPETABLE-READ可重读，已经可以完全保证事务的隔离性要求，即InnoDB默认的REPETABLE-READ隔离级别已经完全保证了事务的隔离性要求，达到了SQL标准的SERIALIZABLE隔离级别。事务隔离级别越低，请求的锁越少，所以大部分数据库系统的隔离水平都是READ-COMITTED，但是InnoDB默认使用REPEATABLE-READ并不会有任何性能损失；

InnoDB存储引擎在分布式事务中，一般情况使用SERIALIZABLE隔离级别；

锁机制与InnoDB锁算法

MyISAM：采用的是表级锁； **InnoDB**：支持行级锁和表级锁，默认为行级锁；

表级锁和行级锁对比

表级锁：MySQL中锁定粒度最大的一种锁，对当前操作的整张表加锁，实现简单，消耗资源比较少，加锁快，不会出现死锁。其锁定粒度最大，触发锁冲突的概率最高，并发度最低，MyISAM和InnoDB引擎都支持表级锁。 **行级锁**：MySQL中粒度最小的一种锁，只针对当前操作的行进行加锁。行级锁能大大减少数据库的冲突。其加锁粒度最小，并发度高，但加锁的开销最大，加锁慢，会出现死锁；

锁分类

可以划分为共享锁和排它锁： **共享锁Share Lock简称S**：又被称之为读锁，其他用户可以并发的读取数据，但任何事务都不能获取数据上的排它锁，知道释放所有的共享锁；

共享锁称之为读锁，若事务T对A加S锁，其他事务只能再对A加S锁，而不能加X锁，直到T释放S锁。这就保证了其他事务可以读A，但在T释放A的S锁之前不能对A进行修改；

排他锁Exclusive lock简称X锁：又称之为写锁，若事务对A加上X锁，则只允许T读取和修改A，其他任何事务都不能再对A加任何类型的锁，知道T释放A上锁。他防止任何其他事务获取资源上的锁，知道事务T最后释放A上面的锁。在更新操作上始终应用排它锁；

共享锁S锁：如果事务T对数据A加上共享锁之后，则其他事务只能对A加共享锁，不能加排它锁，获取共享锁的事务只能读数据不能修改数据； 排它锁X锁：如果事务T对数据A加上排它锁，则其他事务不能再对A加任何类型的锁，获取排它锁的事务技能读数据也能修改数据；

当事务自己想要加锁的数据对象正在被其他事务加排他锁占用的时候，该事物可以在需要锁定的行的表上添加一个合适的意向锁，如果自己需要一个共享锁，那么就在表上添加一个意向共享锁，而如果自己需要是某行上面添加一个排它锁的话，则在表上面添加一个意向排它锁，意向共享锁可以同时并存多个，但是意向排他锁同时只能有一个存在； **意向共享锁IS**：表示事物准备给数据行计入共享锁，事物在一个数据行加共享锁之前必须先取得该表的IS锁。 **意向排它锁IX**：表示事物准备给数据加入排它锁，事务在一个数据行加排它锁前必须先取得该表的IX锁；

意向锁是表级锁，表示的是一种意向，仅仅表示事物正在读取和记录某一行记录，在真正加行锁时才会判断是否冲突，意向锁是InnoDB自动加的，不需要用户干预；

IX IS是表级锁不会和行级的X\S锁发生冲突，只会和表级的X\S发生冲突；

死锁和避免死锁

InnoDB行级锁是基于索引实现的，如果查询语句未命中任何索引，那么InnoDB会使用表级锁；此外InnoDB的行级锁是针对索引加的锁，不针对数据记录。因此即使在访问不同行的记录，如果使用了相同的索引键仍然会出现锁冲突；

不同于MyISAM总是一次性的获得所有锁，InnoDB是逐步获得的，当两个事务都需要获取对方持有的锁，导致双方产生了死锁，发生死锁之后，InnoDB一般都可以检查到，并使其中一个事务释放锁回退，另一个可以获取锁完成事务。

避免死锁： 通过表级锁来减少死锁的概率； 多个程序尽量约定以相同的顺序访问表，（哲学家就餐） 同一个事务尽可能一次性的获取所有所需的资源的锁；

InnoDB存储引擎的算法有三种： Record lock：单个行记录的锁； Gap lock：间隙锁，锁定一个范围。不包括记录本身； Next-key lock：record+gap锁定一个范围，包含记录本身；

相关知识点：

1. innodb对于行的查询使用next-key lock
2. Next-locking keying为了解决幻读问题；
3. 当查询的索引含有唯一属性的时候，将next-key lock降级为record key
4. Gap锁设计的目的是为了阻止多个事务将记录插入同一个范围之中，而这会导致幻读问题的产生；
5. 两种方式显式关闭gap锁，除了外键约束和唯一性检查外，其余情况仅适用record lock；A. 将事务隔离级别设置为RC B. 将参数innodb_locks_unsafe_for_binlog设置为1；

大表优化

MySQL单表数据量过大时，数据库的CRUD性能会明显下降，一下常见的优化措施如下：

1. 限定数据的范围

务必禁止不带任何限制数据范围条件的查询语句，比如我们查询历史订单的时候，一定控制在一个月范围内；

2. 读写分离

经典的数据库拆分方案，主库负责写，从库负责读；

3. 垂直分区

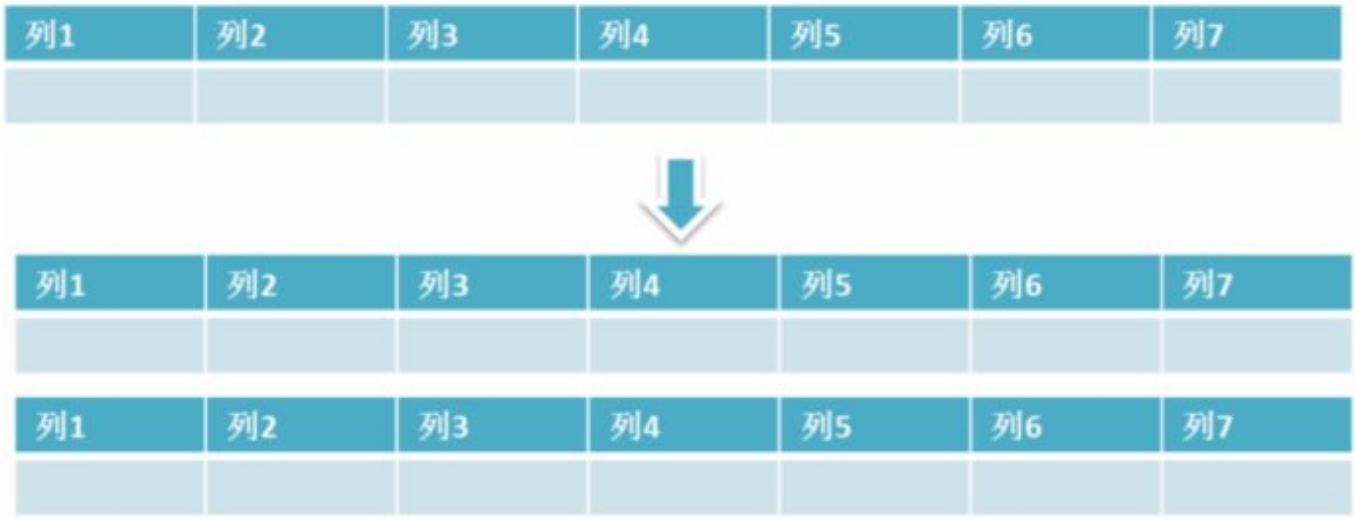
根据数据库中数据表的相关性进行拆分。将用户表拆分为两个单独的表，甚至放到单独的库做分库； 垂直分区就是讲表列的拆分，把一张列比较多的表拆分为多个表，如图所示；

垂直拆分的优点：使列数据变小，在查询时减少读取的Block数，减少IO次数，此外，垂直分区可以简化表的结构，易于维护。 垂直拆分的缺点：主键会出现冗余，需要管理冗余列，并会引起Join操作，可以通过在应用层进行Join来解决。此外垂直分区使得事务变的更加复杂；

4. 水平分区

保持数据表结构不变，通过某种策略存储的数据分片。这样每一片数据分散到不同的表中，达到分布式的目的，水平拆分可以支撑非常大的数据量；

水平拆分是指数据表行的拆分，表的行数超过200万行时，就会变慢，这时可以把一张的表的数据拆成多张表来存放。举个例子：我们可以将用户信息表拆分成多个用户信息表，这样就可以避免单一表数据量过大对性能造成影响。



分表仅仅是解决了单一表数据过大的问题，但由于表的数据还是在同一台机器上，其实对于提升MySQL并发能力没有什么意义，所以 **水平拆分最好分库**

水平拆分能够支持非常大的数据量存储，应用端改造也少，但是分片事务难以解决，跨节点Join性能较差，逻辑复杂，尽量不要对数据进行分片，因为拆分会带来逻辑、部署、运维的各种复杂度

数据库分片的两种常见方案： 客户端代理：分片逻辑在应用端，封装在jar包中，通过修改或者封装JDBC来实现。

中间件代理：在应用和数据中间加了一个代理层。分片逻辑统一维护在中间件服务中。

解释一下什么是池化设计思想，什么是数据库连接池？为什么需要数据库连接池?

这种设计会初始预设资源，解决的问题是抵消每次获取资源的消耗，如创建线程的开销，获取远程连接的开销等等； 除了初始化资源，池化设计还包括如下特征：池子的初始值、池子的活跃度、池子的最大值等，这些特征直接映射到java线程池和数据库连接池的成员属性中，

数据库连接本质上是一个Socket的连接。数据库服务端还要维护一些缓存和用户权限等信息之类的，所以占用了一些内存。我们可以把数据库连接池看做是维护的数据库连接的缓存，一遍将来对数据库的请求时可以重用这些连接。l

在连接池中，创建连接后，将其放置在池中，并再次使用它，因此不必建立新的连接，如果使用了所用连接就会建立一个新的连接将其添加到池中；

分库分表之后，id主键如何处理?

分表之后，每个表都是从1累加的，这样是不对的，需要一个全局唯一的id来支持；

全局id有以下几种方式： UUID：不适合作为主键，太长了，而且无序不可读，查询效率低下，适合用于生成唯一的名字的标示比如文件的名字；

数据库自增ID：两台数据库分别设置不同的步长，生产不重复的ID的策略来实现高可用，这种方式生成的id有序但是需要独立部署数据库实例，成本高，有性能瓶颈；

利用redis生成id：性能比较好，灵活方便，不依赖与数据库。但是引入新的组件造成系统更加复杂，可用性降低编码复杂；

Twitter的snowflake算法：Github 地址：https://github.com/twitter-archive/snowflake。

美团的Leaf分布式ID生成系统：Leaf是美团开源的分布式ID生成器，能保证全局唯一性，趋势递增、单调递增、信息安全等； ……