# 一．基础知识

1. 数据库三范式

第一范式：每一列的属性都是原子不可再分的。对于关系型数据库，这是基本要求，只要建的表是合法的，就一定满足。

第二范式：第二范式建立在第一范式的基础上。第二范式要求属性完全依赖于主键，即每一行的数据都可以通过主键唯一区分。因此主键都是唯一非空的。

第三范式：属性不依赖于其他非主属性，而是直接依赖于主键。简而言之就是消除传递依赖。对于一对多，多对多的关系时，我们就需要额外建表，然后通过外键来建立联系。

1. 分别说一下范式和反范式的优缺点

范式就是满足三大范式，列属性原子不可再分，属性依赖于主键，并被主键唯一区分，消除传递依赖。反范式就是不满足数据库的三大范式，比如不满足第二范式，不创建主键，在spring security中存储的token表就没有主键。还是就是不满足第三范式，将数据都放在一张表中。

范式的优点就是数据简洁，没有重复数据。

1. 更新的效率更高，相较于反范式可能要一次更新几条，范式只需要更新一次，
2. 另外由于重复数据少，因此加载到内存也更快，
3. 使用distinct和group by也更快，甚至不需要distinct和group by。

缺点就是分为了多张表，通常需要关联，可能还会导致索引无效，从而效率更低。

反范式的优点就是简单，避免了表之间的关联，当数据比内存大的时候，就避免了随机IO，可能比范式种多个关联更快。缺点就是效率较范式要低，因为更新可能需要同时更新多条数据，并且冗余数据过多，导致表的数据增多，查询可能也会变慢。

现实中我们通常是范式和反范式混用

1. 约束的类型

主键约束，非空约束，唯一约束，外键约束，默认约束。检查约束（mysql不支持,可以设置但是没效果）

1. 关系型数据库和非关系数据库的优缺点
   1. 关系型数据库

**优点：**  
1、**易于维护**：都是使用表结构，**格式一致**；  
2、**使用方便**：**SQL语言通用**，可用于复杂查询；  
3、**复杂操作**：**支持SQL**，可用于一个表以及多个表之间非常复杂的查询。  
**缺点：**  
1、**读写性能比较差**，尤其是海量数据的高效率读写；  
2、固定的表结构，**灵活度稍欠**；  
3、高并发读写需求，传统关系型数据库来说，硬盘I/O是一个很大的瓶颈。

* 1. 非关系型数据库

1、**格式灵活**：存储数据的格式**可以是key,value形式、文档形式、图片形式等等**，文档形式、图片形式等等，使用灵活，应用场景广泛，**而关系型数据库则只支持基础类型**。  
2、**速度快**：nosql可以使用硬盘或者随机存储器作为载体，而关系型数据库只能使用硬盘；  
3、高扩展性；  
4、**成本低**：nosql数据库**部署简单**，**基本都是开源软件**。

**缺点：**  
1、不提供sql支持，**学习和使用成本较高**；  
2、**无事务处理**；  
3、**数据结构相对复杂**，复杂查询方面稍欠。

# 二．引擎

1. MyISAM和InnoDb数据库的区别
   1. 存储结构：MyISAM每张表分为了3个文件，frm（表的结构）,myd（表的数据）,myi（索引），这是由于它把索引和数据是分开存储的。而InnoDb所有表都存储在了一个文件中。
   2. 事物：MyISAM不支持，InnoDb支持
   3. 外键：MyISAM不支持，InnoDb支持
   4. 锁：MyISAM是表级锁，InnoDb是行级锁。
   5. 索引：MyISAM特有全文索引，InnoDb特有哈希索引。
   6. select count(\*):MyISAM更快，因为它内部维护了一个计数器。
   7. 查询：如果查询非主属性且没有索引覆盖的话，MyISAM更快，因为MyISAM查到对应的结点后，直接通过指针就能去文件中取，而InnoDb还需要进行一次回表操作，取主键索引再查一次。
   8. 增删改：InnoDb更快，因为他的锁的粒度更低。
2. MyISAM和InnoDb索引的区别
   1. MyISAM的索引是非聚簇索引，而InnoDb是聚簇索引；
   2. InnoDb索引的叶子结点存储的是主键和其他索引列，因此如果查询时做到覆盖索引，查询效率会十分高，不然的话需要再进行一次寻址。MyISAM的叶子结点存储的是行数据的地址，需要再进行一次寻址操作c才能取到数据。

# 三．索引

1. 什么是索引

索引其实就是一种数据结构，用来方便数据的查找。常见的有B+树和哈希索引，其中InnoDB默认的是B+树，也是我们平时使用最多的。另外MyISAM还支持R树索引和全文索引。

1. 索引的优缺点

索引的好处：

1. 大大加快了查询的效率，通过B树等数据结构，减少了查询的次数。
2. 避免服务器的排序和临时表。因为B树的叶子结点本身有序，可以直接返回。
3. 将随机I/O变为顺序I/O

索引的缺点：

1. 创建额外的空间来存储索引。
2. 在增删改的时候还需要额外的时间来维护索引。

因此不是所有的情况都需要用上索引，如果说使用索引带来效率的提升大于维护索引的成本，那是可以的。在数据量比较少的时候，全表扫描可能比索引更快。在数据量比较多的时候，索引才能发挥它的优势。

1. 索引的类型

唯一索引，主键索引，普通索引，前缀索引，哈希索引(innodb)，全文索引(myisam)，

1. B+ 树和 B 树的区别，为什么是B+树

B+树和B树都是一种平衡多叉树，他们的最大区别就是B树的非叶子结点存储了数据，而B+树的非叶子结点是不存储数据的，数据全部存储在叶子结点，并且叶子结点是对他的父结点的一个拷贝，并且多了一个指针指向下一个叶子结点。

因为B+树的能够很好的解决范围查询，并且由于B+树的非叶子结点没有存储数据，一次也能够加载更多的数据到内存中去，从而查询更快。不过他的缺点就是每次查询都必须要到叶子结点才能拿到数据。而B树可以在非叶子结点就能够拿到数据并返回。

1. B+树的叶子结点存储的是什么数据

如果是主键的话，存储了整行的数据，如果不是主键的话，存储的是主键值以及其他有索引列的数据。

1. 聚簇索引和非聚簇索引

聚簇索引：叶子结点放的是行数据

非聚簇索引：叶子结点放的是主键或者行数据地址，需要再次查询或者寻址才能拿到数据，因此聚簇索引效率要比非聚簇索引高很多，所以要利用好聚簇索引

1. 覆盖索引

覆盖索引就是所查询的列都建立了索引，由于索引的叶子结点存储的是主键值和索引值，因此覆盖索引后能够减少一次回表操作，直接在索引上就能够拿到数据。因此平时写的时候尽量只写必要的查询列，不写select \* ,增加覆盖索引的概率。

1. 对于联合（多列）索引，哪些情况下能用到索引，哪些情况用不到索引

联合索引通常在使用and条件的时候使用，在5.0版本之前，即使建立了索引，但是无法使用，会进行全表扫面，5.0开始进行了优化，能够使用索引，但是需要合并索引，就是将分别用单个索引查询出来的结果进行合并，这样会消耗大量的资源，并且查询优化器也不会把这些成本计入。从而导致还不如进行全表的顺序扫描，因此多条件的时候最好建立联合索引。

对于联合索引的建立，我们通常把使用频率最高的放在前面，因为这样能够增大使用到索引的概率，比如建立了(key1,key2,key3)的联合索引，相当于建立了(key1)的索引，(key1,key2)的索引还有(key1,key2,key3)的索引。而(key2,key3)这样的查询就使用不到索引。因此要把使用频率最高的放前面。

另外，还可以把区分度最高的放在前面，对于联合索引，有一个最左匹配原则，即在检索的时候从最左边开始匹配。比如（a,b,c）的联合索引，会先通过a的值匹配，然后再是B,然后再是c。

另外，如果在联合索引中使用范围查询就会使范围查询后面的列失效。如果使用表达式，函数的话，索引也会是失效的。

最后还要注意索引失效的情况，比如范围查询会使当前字段后面的字段无效，虽然a字段经常用，且区分度高，但是可能使索引直接无效了，这个时候就考虑把a字段放到后面。

因此联合索引的顺序选择十分重要，通常把使用频率最高的索引放在最前面。

1. 什么时候用不到索引
2. 虽然条件中有索引，但是加入了表达式就不行，比如where price + 1 = 5;
3. 使用了通配符开头的like操作，直接like可以
4. 字符串不加单引号会失效，因为mysql会有一个隐式的转换，相当于使用了函数。
5. or条件会使索引失效
6. 查询优化器经过计算后，可能使用索引的成本更高，从而不使用索引。
7. Mysql 索引重建

重建索引在常规的数据库维护操作中经常使用。在数据库运行了较长时间后，索引都有损坏的可能，这时就需要重建。对数据重建索引可以起到提高检索效率。REPAIR TABLE `table\_name` QUICK;

另外，要经常清理索引的碎片。optimze table或者重建表()

1. 索引下推

在不使用索引下推的时候，在使用非主键索引查询时，存储引擎会把查询到的数据返回给数据库，数据库再根据条件判断数据是否符合条件。

而使用索引下推的时候，mysql服务器会把判断条件传递给存储引擎，在存储引擎就能够通过索引判断是否符合条件了，然后把符合条件的数据返回给数据库。

他的好处就是减少了数据库的回表操作，另外在一定程度上也打破了最左前缀的原则。

# 四．事务

1. 介绍一下数据库的事务

事物有四大特性：原子性，隔离性，一致性和持久性。

原子性是：事物要么全部失败，要么全部成功。保证原子性是通过redo/undo log保证的。

隔离性：其他事务不能访问当前事务。

一致性：事务都是从一个一致性状态转换到另外一个一致性状态。

持久性：事务操作成功了之后都是永久生效的。

上面4个特性加起来就是事务了。

1. Mysql 有哪些隔离级别

读未提交：可以读取到还没有提交到的事务，如果事务回滚了，会造成脏读

读已提交：只能看到已经提交的事务所作的改变，可以解决脏读，但是还有不可重复读，幻读的问题。

可重复读：保证多个事务在并发读取的时候，可以看到相同的行，即解决了不可重复读，但是还是有幻读。主要是通过行级锁来实现的。

可串行化：这是MySql的最高隔离级别，不允许并发操作，事务只能串行化执行，这样就不会有并发带来的问题，从而幻读也解决了。

InnoDb的默认隔离级别是可重复读。

1. Binlog 和 Redolog ，undolog?

redolog记录着更新成功后的值。如果事务成功提交后，还没来得及写入data file就挂了的话，可以通过redolog进行恢复。从而保证持久性

undolog记录着更新之前的值。如果事务失败了，可以通过undolog进行回滚。从而保证了一致性

binlog是二进制日志，记录着数据库表结构的表更还有表数据的修改的二进制文件。当我们需要复制数据库的时候就用到了binlog

1. Mysql 什么情况会造成脏读、不可重复度、幻读？如何解决

脏读：由于读取到了进行了回滚的事务，从而读取到了不正确的数据。将隔离级别设置为读未提交及以上，

不可重复度：在一个事务中，前后两次读取同一行的数据不一致。就是由于在两次读取中间进行了一次更新的操作，从而导致前后两次读取的结果不一致。将隔离级别设置为可重复度及以上。

幻读：在一个事务中，前后两次读取的数据的行数不一致。就是由于在两次读取中间进行了一次插入操作，从而前后两次读取的结果不一致。将隔离级别设置为可串行话可以解决。

1. 谈一谈 MVCC 多版本并发控制

MVCC就是多版本并发控制。MVCC解决的问题是读写互相不阻塞的问题，每次更新都产生一个新的版本，读的话可以读历史版本。MVCC 是一种并发控制的方法，一般在数据库管理系统中，实现对数据库的并发访问。MVCC是行级锁的一个变种，但是它在很多情况下避免了加锁操作，因此开销更低。虽然实现机制有所不同，但大都实现了非阻塞的读操作，写操作也只锁定必要的行。

在Mysql的InnoDB引擎中就是指在读已提交(READ COMMITTD)和可重复读(REPEATABLE READ)这两种隔离级别下的事务对于SELECT操作会访问版本链中的记录的过程。

这就使得别的事务可以修改这条记录，反正每次修改都会在版本链中记录。SELECT可以去版本链中拿记录，这就实现了读-写，写-读的并发执行，提升了系统的性能。 InnoDB只查找版本(DB\_TRX\_ID)早于当前事务版本的数据行。

**版本链**

在InnoDB引擎表中，它的聚簇索引记录中有两个必要的隐藏列：

trx\_id这个id用来存储的每次对某条聚簇索引记录进行修改的时候的事务id。

roll\_pointer每次对哪条聚簇索引记录有修改的时候，都会把老版本写入undo log**中**。这个roll\_pointer就是存了一个指针，它指向这条聚簇索引记录的上一个版本的位置，通过它来获得上一个版本的记录信息。

# 锁

1. Innodb 的默认加锁方式是什么，是怎么实现的
2. 按锁的粒度划分，可分为表级锁、行级锁、页级锁（mysql）
   1. 表级锁：开销小，加锁块；不会出现死锁；但是锁的粒度大，并发度低
   2. 行级锁：开销大，加锁慢；会出现死锁；锁的粒度小，并发度最高
   3. 页级锁：介于行级锁和表级锁之间的一种锁
3. 按锁级别划分，可分为共享锁、排他锁
   1. 共享锁：也称为读锁，多个事务对于同一数据都能获取同一把锁，但是只能读不能写，即有共享锁的时候，只能进行读，但是不能写。用法：select …lock in share mode
   2. 排他锁：也称为写锁，排他锁只有一个事务能够获取，获取了排他锁的事务能够对改行数据进行读写。
4. 按使用方式划分，可分为乐观锁、悲观锁
   1. 乐观锁：乐观锁就是认为数据在一般情况下不会出现冲突的问题，因此并不会加锁。而只有在更新的时候才进行检查是否冲突，如果冲突了再返回用户，让用户决定如何去做。实现乐观锁有两种方式，比较版本号和比较时间戳，他们的内核都是CAS的原子更新操作。就是提交的时候，用拿到的版本号或者时间戳对数据库种的进行比较，如果和数据库的一致，则进行更新，如果不一致，就返回冲突。
   2. 悲观锁：悲观锁就是悲观的认为数据访问总是会发生冲突，因此在处理数据之前都会加锁。这个锁就是数据库本身的一个锁，比如读写锁。
5. **间隙锁讲解一下**？

当我们使用范围查询进行加锁的时候，mysql不仅会对相应的索引记录加锁，还会对在范围内不存在的数据进行加锁。 因此导致了某些无辜的数据也被锁定了，从而导致无法操作在范围内的所有数据，这对性能上有一定的损害

1. Mysql 中 varchar 和 char 的区别

char是固定长度，varchar长度可变。varchar：**如果原先存储的位置无法满足其存储的需求**，就需要一些额外的操作，根据存储引擎的不同，有的会采用**拆分机制**，有的采用**分页机制**。

char和varchar的存储字节由**具体的字符集**来决定;

char是固定长度，长度不够的情况下，用空格代替。varchar表示的是实际长度的数据类型

1. 数据库外键的优缺点

优点：

设置外键可以保证数据的完整性与关联性，杜绝数据冗余

       级联操作方便，比如涉及一个离职用户的所有相关信息，可以直接删除这个用户就能做到所有信息一起删除的操作

缺点：

 主表进行修改或变更，会波及一大片从表，顿时会使公司部分业务处于不可用的状态，不利中小公司的设计，一般中小型公司都没有dba，开发人员对数据库了解的不够深，容易误操作，所以建议不用使用过多的外键来操作相关业务表

      而且对于insert,   update,   delete等操作时都会先检查外键的约束条件再操作，性能有所下降

1. Mysql 什么情况会造成慢查，如何查看慢查询

造成慢查询：数据量很大的情况下，没有用到索引；使用了临时表；

查看慢查询：开启慢查询日志，如果出现慢查询会记录在慢查询日志中，然后通过explain，来查看可能造成慢查询的原因。

1. **数据库问题，说一下从你打开命令行到发送请求，mysql服务器的整个相应流程? (当问到需要介绍数据库底层时可以这样回答)**

MySQL 主要分为 Server 层和引擎层。

Server 层主要包括连接器、查询缓存、分析器、优化器、执行器，同时还有一个日志模块（binlog），这个日志模块所有执行引擎都可以共用, redolog 只有 InnoDB 有。

引擎层是插件式的，目前主要包括，MyISAM,InnoDB,Memory 等。

**连接器：** 身份认证和权限相关(登录 MySQL 的时候)。

**查询缓存:** 执行查询语句的时候，会先查询缓存（MySQL 8.0 版本后移除，因为这个功能不太实用）。

**分析器:** 没有命中缓存的话，SQL 语句就会经过分析器，分析器说白了就是要先看你的 SQL 语句要干嘛，再检查你的 SQL 语句语法是否正确。先词法分析，再语法分析

**优化器：** 按照 MySQL 认为最优的方案去执行。

**执行器:** 执行语句，然后从存储引擎返回数据。

查询语句：

select \* from tb\_student A where A.age='18' and A.name=' 张三 ';

先检查该语句是否有权限，如果没有权限，直接返回错误信息，如果有权限，在 MySQL8.0 版本以前，会先查询缓存，以这条 sql 语句为 key 在内存中查询是否有结果，如果有直接缓存，如果没有，执行下一步。

通过分析器进行词法分析，提取 sql 语句的关键元素，比如提取上面这个语句是查询 select，提取需要查询的表名为 tb\_student,需要查询所有的列，查询条件是这个表的 id='1'。然后判断这个 sql 语句是否有语法错误，比如关键词是否正确等等，如果检查没问题就执行下一步。

接下来就是优化器进行确定执行方案，上面的 sql 语句，可以有两种执行方案：

a.先查询学生表中姓名为“张三”的学生，然后判断是否年龄是 18。

b.先找出学生中年龄 18 岁的学生，然后再查询姓名为“张三”的学生。

那么优化器根据自己的优化算法进行选择执行效率最好的一个方案（优化器认为，有时候不一定最好）。那么确认了执行计划后就准备开始执行了。

进行权限校验，如果没有权限就会返回错误信息，如果有权限就会调用数据库引擎接口，返回引擎的执行结果。

更新语句：

update tb\_student A set A.age='19' where A.name=' 张三 ';

我们来给张三修改下年龄，在实际数据库肯定不会设置年龄这个字段的，不然要被技术负责人打的。其实条语句也基本上会沿着上一个查询的流程走，只不过执行更新的时候肯定要记录日志啦，这就会引入日志模块了，MySQL 自带的日志模块式 **bin log（归档日志）** ，所有的存储引擎都可以使用，我们常用的 InnoDB 引擎还自带了一个日志模块 **redo log（重做日志）**，我们就以 InnoDB 模式下来探讨这个语句的执行流程。流程如下：

先查询到张三这一条数据，如果有缓存，也是会用到缓存。

然后拿到查询的语句，把 age 改为 19，然后调用引擎 API 接口，写入这一行数据，InnoDB 引擎把数据保存在内存中，同时记录 redo log，此时 redo log 进入 prepare 状态，然后告诉执行器，执行完成了，随时可以提交。

执行器收到通知后记录 binlog，然后调用引擎接口，提交 redo log 为提交状态。

更新完成。

**这里肯定有同学会问，为什么要用两个日志模块，用一个日志模块不行吗?**

这是因为最开始 MySQL 并没与 InnoDB 引擎( InnoDB 引擎是其他公司以插件形式插入 MySQL 的) ，MySQL 自带的引擎是 MyISAM，但是我们知道 redo log 是 InnoDB 引擎特有的，其他存储引擎都没有，这就导致会没有 crash-safe 的能力(crash-safe 的能力即使数据库发生异常重启，之前提交的记录都不会丢失)，binlog 日志只能用来归档。

并不是说只用一个日志模块不可以，只是 InnoDB 引擎就是通过 redo log 来支持事务的。那么，又会有同学问，我用两个日志模块，但是不要这么复杂行不行，为什么 redo log 要引入 prepare 预提交状态？这里我们用反证法来说明下为什么要这么做？

**先写 redo log 直接提交，然后写 binlog**，假设写完 redo log 后，机器挂了，binlog 日志没有被写入，那么机器重启后，这台机器会通过 redo log 恢复数据，但是这个时候 binlog 并没有记录该数据，后续进行机器备份的时候，就会丢失这一条数据，同时主从同步也会丢失这一条数据。

**先写 binlog，然后写 redo log**，假设写完了 bin log，机器异常重启了，由于没有 redo log，本机是无法恢复这一条记录的，但是 bin log 又有记录，那么和上面同样的道理，就会产生数据不一致的情况。

如果采用 redo log 两阶段提交的方式就不一样了，写完 binglog 后，然后再提交 redo log 就会防止出现上述的问题，从而保证了数据的一致性。那么问题来了，有没有一个极端的情况呢？假设 redo log 处于预提交状态，binglog 也已经写完了，这个时候发生了异常重启会怎么样呢？ 这个就要依赖于 MySQL 的处理机制了，MySQL 的处理过程如下：

判断 redo log 是否完整，如果判断是完整的，就立即提交。

如果 redo log 只是预提交但不是 commit 状态，这个时候就会去判断 binlog 是否完整，如果完整就提交 redo log, 不完整就回滚事务。

这样就解决了数据一致性的问题。

查询语句的执行流程如下：权限校验（如果命中缓存）--> 查询缓存 --> 分析器 --> 优化器 --> 权限校验 --> 执行器 --> 引擎

更新语句执行流程如下：分析器 --> 权限校验 --> 执行器 --> 引擎 -- redo log(prepare 状态) --> binlog --> redo log(commit状态)