```
○ 2.非主键列完全依赖于主键,而不能是依赖于主键的一部分
                                  3.: 非主键列只依赖于主键,不依赖于其他非主键.
                      MySQL 查询过程
                       MySQL 更新过程
                                              Atomicity: 要么成功要么失败。
                                              Consistency: 数据从一个状态转移到另外一个状态,不存在中间状态。
                                              Isolation: 通常来说,一个事务在完全提交之前,对其他事务是不可见的。
                                              Durability: 一旦事务提交,数据的状态便不可更改。
                                               脏读: 脏读就是指当一个事务正在访问数据,并且对数据进行了修改,而这种修改还没有提交到数据
                                               库中,这时,另外一个事务也访问这个数据,然后使用了这个数据。
                                     事务并发问题 😑 幻读: 所谓幻读是指同一个事务内多次查询返回的结果集不一样(比如增加了或者减少了行记录)。
                                                不可重复读: 所谓不可重复读是指在一个事务内根据同一个条件对行记录进行多次查询,但是搜出来
                                                的结果却不一致。发生不可重复读的原因是在多次搜索期间查询条件覆盖的数据被其他事务修改了。
                                                          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
                                                可重复读(RR)
                                     事务隔离级别
                                                          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
                                                读提交(RC)
                                                          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
                      MySQL 事务
                                                          避免并行访问。
                                               串行化
                                                一致性读 😑 一致性视图
                                               当前读
                                     事务的可见性
                                               行锁
                                              事务嵌套执行
                                              挂起父事务,新建事务执行
                                              挂起父事务,以非事务执行
                                              没有父事务,新建事务执行
                                              没有父事务,抛出异常
                                              没有父事务,以非事务执行
                                              以非事务方式执行,如果当前存在事务,则抛出异常。
                                                                            乐观锁会"乐观地"假定大概率不会发生并发更新冲突,访问、处理数据过程中不加锁,只在更新数
                                                                            据时再根据版本号或时间戳判断是否有冲突, 有则处理, 无则提交事务;
                                                   乐观锁与悲观锁是两种并发控制的思想
                                            悲观锁
                                                                            悲观锁会"悲观地"假定大概率会发生并发更新冲突,访问、处理数据前就加排他锁,在整个数据处
                                                                            理过程中锁定数据,事务提交或回滚后才释放锁;
                                            共享锁(S锁) 🖯 允许持锁事务读取一行
                                   锁兼容性
                                                                           共享锁与排他锁是InnoDB实现的两种标准的行锁
                                            排它锁(X锁) 🕒 允许持锁事务更新或者删除一行
                                                                              Innodb 可以采用 MVCC 方式( - single-transaction 参数)
                                            全局锁 😑 全局锁的典型使用场景是,做全库逻辑备份
                                                                              其他采用FTWRL(Flush tables with read lock) 方式
                                                                            与 FTWRL 类似,可以用 unlock tables 主动释放锁,也可以在
                                                                         ○ 客户端断开的时候自动释放。需要注意, lock tables 语法除了会
                                                                                                                  表锁一般是在数据库引擎不支持行锁的时候才会被用到的。
                                                  表锁的语法是 lock tables … read/write
                                                                            限制别的线程的读写外,也限定了本线程接下来的操作对象。
                                                                                                                                                    1.kill 长事务,让DDL语句获取到MDL写锁。但是这种方式不一定管用,如果是一个热点表新的请求可能
                                                                                                                                           解决措施 😑 2.比较理想的机制是,在 alter table 语句里面设定等待时间,如果在这个指定的等待时间里面能够拿到
                                                                                                                                                     MDL 写锁最好,拿不到也不要阻塞后面的业务语句,先放弃。之后开发人员或者 DBA 再通过重试命令重
                                                                                                                                                    复这个过程。
                                                                1. 当对一个表做增删改查操作的时候,加 MDL 读锁;
                                                                                                        在 MySQL 5.5 版本中引入了 MDL,MDL 不需要显式使用,
                                                                                                                                                                                    1. 拿MDL写锁
                                                                当要对表做结构变更操作的时候加 MDL 写锁。
                                                                                                        在访问一个表的时候会被自动加上。MDL 的作用是,保证
                                                                                                                                                                                    2. 降级成MDL读锁
                                                                2.读锁之间不互斥,因此你可以有多个线程同时对一张表增删改查。
                                                                                                        读写的正确性。你可以想象一下,如果一个查询正在遍历
                                                  元数据锁(MDL)
                                                                                                                                                                                                       1、2、4、5如果没有锁冲突,执行时间非常短。第3步占用了DDL绝大部分时间,这期间这个表可以正
                                                                                                        一个表中的数据,而执行期间另一个线程对这个表结构做
                                                                                                                                                                                    3. 真正做DDL(临时表?
                                                                                                                                            5.6以后支持online DDL,不会堵塞增删改查,Online DDL的过程:
                                                                 3.读写锁之间,写锁之间是互斥的,用来保证变更表结构
                                                                                                                                                                                                       常读写数据,是因此称为 "online",上面的问题是在第一步就堵住了。
                                                                                                        变更,删了一列,那么查询线程拿到的结果跟表结构对不
                                                                操作的安全性。因此,如果有两个线程要同时给一个表
                                            表锁
                                                                                                                                                                                    4. 升级成MDL写锁
                                                                                                        上,肯定是不行的。线上修改表结构可能拖垮数据库。
                                                                加字段,其中一个要等另一个执行完才能开始执行。
                                                                                                                                                                                    5.释放MDL锁
                                                                                                                                                  我们知道事务中的 MDL 锁,在语句执行开始时申请,但是语句结束后并不会马上释放,而会等到整个
                                                                                                                                                  事务提交后再释放。如果某个表MDL写锁 被前面的长事务给阻塞了,那么后续的读锁都会被阻塞, 恰好
                                                                                                                                                   这个表上的查询语句频繁,而且客户端有重试机制,也就是说超时后会再起一个新 session再请求的
                                                                                                                                                  话,这个库的线程很快就会爆满,从而导致数据库不可用。
                                                          自增锁是一个特殊的表级锁,事务插入自增列的时候需要获取,最简单情况下如果一个事务插入一
                                                          个值到表中,任何其他事务都要等待,这样第一个事物才能获得连续的主键值。
                                                          基本概念: 为了实现多粒度锁,InnoDB 使用了意向锁(简称 I 锁)。I 锁是表明一个事务稍后要
                                                          获得表中某一行记录的锁(S 锁 或 X 锁), 在对应表上加的表级锁
                                                                                                                 意向锁的主要目的是展示某人正在锁定表中一行,或者将要锁定一行。新请
                                                  意向锁 😑
                                                                 1. 意向排它锁(简称 IX 锁)表明一个事务意图在某个表中设置某些行的 x 锁
                                                                                                                  求的锁只有兼容已有锁才能被允许,否则必须等待不兼容的已有锁被释放。
                                                                2. 意向共享锁(简称 IS 锁)表明一个事务意图在某个表中设置某些行的 s 锁
                                                                                              如果你的事务中需要锁多个行,要把最可能造成锁冲突、
                                                           在InnoDB中,行锁是在需要的时候才加上的,但并不是不
                                                                                              最可能影响并发度的锁尽量往后放。InnoDB 才使用行级锁,
                                                           需要了就立即释放,而是需要等到事务结束时才释放。
                      MySQL 锁
                                                                                             否则,InnoDB 将使用表锁!
                                                                  记录锁针对索引记录,阻止其他所有事务插入、修改或者删除 被锁住的行。
                                                           记录锁    记录锁总是锁定索引记录,即使表没有索引(这种情况下,InnoDB会创建隐式的索引,并使用这
                                                                   个索引实施记录锁)。
                                  锁的粒度
                                                                  间隙锁(gap)是索引记录之间上的锁,或者说第一个索引记录之前或最后一个索引记录之后的间
                                                                  隔上的锁。一个间隙可能是一个索引值、多个索引值,甚至是空的。
                                                 锁的模式 ⊝ GAP锁 ⊝ I
                                                                  间隙锁只阻止其他事务插入到间隙中,不阻止其他事物在同一个间隙上获得间隙锁,所以 gap x
                                                                  lock 和 gap s lock 有相同的作用。
                                                           NEXT-KEY 锁 — 是记录锁和间隙锁的组合,锁住定行记录和行之间的区间。
                                                                     插入意向锁是在插入一行记录操作之前设置的一种间隙锁,这个锁释放了一种插入方式的信号,亦
                                                                     即多个事务在相同的索引间隙插入时如果不是插入间隙中相同的位置就不需要互相等待。
                                                  锁的兼容性
                                                           1.加锁的基本单位是 next-key lock。
                                                           2. 查找过程中访问到的对象才会加锁。
                                                           3.索引上的等值查询,给唯一索引加锁
                                                           的时候,next-key lock 退化为记录
                                           行锁 🧧
                                                  加锁规则
                                                           4.索引上的等值查询,向右遍历时且
                                                           最后一个值不满足等值条件的时候,
                                                           next-key lock退化为间隙锁。
                                                           5.唯一索引上的范围查询会访问到不
                                                           满足条件的第一个值为止。
                                                                 当并发系统中不同线程出现循环资源依赖,涉及的线程都在等待别的线
                                                                  程释放资源时,就会导致这几个线程都进入无限等待的状态,称为死锁。
                                                                 1.一种策略是,直接进入等待,直到超时。这个超时时
                                                                  间可以通过参数innodb_lock_wait_timeout 来设置
                                                         死锁检测
                                                                 2.另一种策略是,发起死锁检测,发现死锁后,主动回滚死锁链条中的某一个事务,其他事务得以继续
                                                                  执行。将参数 innodb deadlock detect 设置为 on, 表示开启这个逻辑。
                                                  死锁
                                                                                              在 InnoDB 中,innodb_lock_wait_timeout 的默认值是 50s,意味着如果采用第一个策略,当出现
MySQL 必知必会
                                                                                              死锁以后,第一个被锁住的线程要过 50s 才会超时退出,然后其他线程才有可能继续执行。对于在线
                                                                  1.关闭死锁检测,假设业务一定不会出现死锁
                                                                                              服务来说,这个等待时间往往是无法接受的。但是,我们又不可能直接把这个时间设置成一个很小的
                                                                 2.服务端控制统一资源的并发度
                                                                                              值,比如 1s。这样当出现死锁的时候,确实很快就可以解开,但如果不是死锁,而是简单的锁等待
                                                                                              呢?所以,超时时间设置太短的话,会出现很多误伤。所以,正常情况下我们还是要采用第二种策
                                                                  3.业务应用优化,避免死锁发生。
                                                                                              略,即:主动死锁检测,而且innodb_deadlock_detect 的默认值本身就是 on。主动死锁检测在发
                                                                                              生死锁的时候,是能够快速发现并进行处理的,但是它也是有额外负担的
                                    基本概念 😑 索引是存储引擎用于快速找到记录的一种数据结构。其目的是目为了提高数据查询的效率。
                                              减少查询需要扫描的数据量,加快查询速度。
                                    索引的优势    减少查询过程中的排序操作和临时表的创建。
                                              将磁盘的随机IO变成顺序IO,加快磁盘读写的速度。
                                              索引也是数据,因此需要额外的磁盘空间存储。
                                    索引的缺点
                                              在更新数据和插入数据的过程同时也会更新索引,增加了操作的耗时。
                                             哈希表

哈希表这种结构适用于只有等值查询的场景,不适用于区间查询。
                                             有序数组  ○  有序数组在等值查询和范围查询场景中的性能就都非常优秀,但有序数组索引只适用于静态存储引擎。
                                                               二叉树是搜索效率最高的,但是实际上大多数的数据库存储却并不使用二叉树。其原因是,索引不止存在内存中,
                                                               还要写到磁盘上。为了让一个查询尽量少地读磁盘,就必须让查询过程访问尽量少的数据块。那么,我们就不应该
                                                              使用二叉树,而是要使用"N叉"树。这里,"N叉"树中的"N"取决于数据块的大小。5.6以后可以通过page大小来间
                                                               接控制"N"的取值。此外,二叉树结构每次指针查找很大概率是触发随机磁盘读。
                                                                     主键索引是一种特殊的唯一索引,一个表只能有一个主键,不允许有空值。
                                                                     如果没有主键,InnoDB会选择一个唯一键来作为聚簇索引,如果没有唯一键,会生成一个隐式的主键。
                                                        主键索引
                                                                      主键索引的叶子节点存的是整行的数据(聚簇索引),非主键索引的叶子节点内容是主键的值(二级索引)
                                                        普通索引
                                                                      主键索引只要搜索ID这个B+Tree即可拿到数据。普通索引先搜索索引拿到主键值,再到主键索引树搜索一次
                                               索引分类
                      MySQL 索引
                                                                      (回表)。一般情况下我会建议创建一个自增主键,这样非主键索引占用的空间最小。
                                                        唯一索引
                                                        联合索引
                                                        全文索引
                                    Innodb 索引
                                                        覆盖索引
                                               索引查询 😑 最左前缀原则
                                                        索引下推(MySQL 5.6 及以上)
                                                                                     1.中间节点不保存数据,那么就可以保存更多的索引,减少数据库磁盘IO的次数.
                                               每一个索引在 InnoDB 里面对应一棵 B+ 树
                                                                                     2.因为中间节点不保存数据,所以每一次的查找都会命中到叶子节点,而叶子节点是处在同一层的,因
                                               。B+ 树能够很好地配合磁盘的读写特性,减
                                                                           B+ 的优点
                                                                                     此查询的性能更加的稳定.
                                               少单次查询的磁盘访问次数。
                                                                                     3.所有的叶子节点按顺序链接成了链表,因此可以方便的话进行范围查询.
                                                     一个数据页满了,按照B+Tree算法,新增加一个数据页,叫做页分裂,会导致性能下降。空间利用率降
                                             页分裂 🖯 低大概50%。当相邻的两个数据页利用率很低的时候会做数据页合并,合并的过程是分裂过程的逆过
                                                      索引可能因为删除,或者页分裂等原因,导致数据页有空洞,重建索引的过程会创建一个新的索引,把
                                                      数据按顺序插入,这样页面的利用率最高,也就是索引更紧凑、更省空间。
                                                       范围查询优化
                                             select 优化
                                                      索引合并
                                   SQL 优化 ⊝/
                                                       索引下推
                                             子查询优化
                      Msyql 优化
                                              索引选择
                                              外键优化
                                   优化与索引
                                              列索引
                                             符合索引
                                             索引下推
                                                                      建立连接的过程通常是比较复杂的,所以我建议你在使用中要尽量减少建立连接的动作,也就是尽量使用长连
                                                                      接。但是全部使用长连接后,你可能会发现,有些时候 MySQL 占用内存涨得特别快,这是因为 MySQL 在执
                                                                      行过程中临时使用的内存是管理在连接对象里面的。这些资源会在连接断开的时候才释放。所以如果长连接累
                                                                      积下来,可能导致内存占用太大,被系统强行杀掉(OOM),从现象看就是 MySQL 异常重启了。解决方
                                                             1.短连接
                                                管理客户端连接
                                                                      1、定期断开长连接。使用一段时间,或者程序里面判断执行过一个占用内存的大查询后,断开连接,之后要
                                                            2.长连接
                                                                      2、 如果你用的是 MySQL 5.7 或更新版本,可以在每次执行一个比较大的操作后,通过执行
                                                                      mysql_reset_connection 来重新初始化连接资源。这个过程不需要重连和重新做权限验证,但是会将连接恢
                                                                      复到刚刚创建完时的状态
                                       Server 层
                      MySQL 逻辑结构
                                                但是大多数情况下我会建议你不要使用查询缓存,查询缓存失效往往很频繁,利大于弊。需要注意的是,
                                                MySQL 8.0 版本直接将查询缓存的整块功能删掉了,也就是说 8.0 开始彻底没有这个功能了。
                                                         1.词法分析,识别SQL语句中各字符串是什么,代表什么
                                                        2.语法分析,根据词法分析的结果进行语法分析,判断SQL语句是否满足Mysql的语法
                                                 索引的选取,各表的连接顺序等,优化器阶段完成后,语句的执行方案就已经确定了下来,而后进入执行
                                                执行 SQL 语句。判断表权限,调用执行引擎接口查询语句。
                                       存储引擎层    存储数据,提供读写接口
                                                         1.在 MySQL 里存在这样的一个问题,如果每一次的更新操作都需要写进磁盘,磁盘也要找到对应的那条记录,然后再更新,整个
                                                         过程 IO 成本、查找成本都很高。为了解决这个问题,MySQL 采用 WAL 技术来解决这个问题。WAL 的全称是 Write-Ahead
                                                         Logging,它的关键点就是先写日志,再写磁盘。
                                                                                                                                最开始 MySQL 里并没有 InnoDB 引擎。MySQL 自带的引擎是 MyISAM,但是 MyISAM 没有 crash-safe
                                                         2.具体来说,当有一条记录需要更新的时候,InnoDB 引擎就会先把记录写到 redo log(粉板)里面,并更新内存,这个时候更
                                                         新就算完成了。同时,InnoDB 引擎会在适当的时候,将这个操作记录更新到磁盘里面,而这个更新往往是在系统比较空闲的时
                                                                                                                                的能力,binlog 日志只能用于归档。而 InnoDB 是另一个公司以插件形式引入MySQL 的,既然只依靠
                                                redo log
                                                                                                                                binlog 是没有 crash-safe 能力的,所以 InnoDB 使用另外一套日志系统——也就是 redo log 来实现
                                                                                                                                crash-safe 能力。这两种日志有以下三点不同:
                                                         3.InnoDB 的 redo log 是固定大小的,比如可以配置为一组 4 个文件,每个文件的大小是 1GB,那么这块"粉板"总共就可以记
                                                                                                                                1、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。
                                                         录 4GB 的操作。从头开始写,写到末尾就又回到开头循环写。
                                       基本概念
                                                                                                                                2、 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
                                                         4.有了 redo log,InnoDB 就可以保证即使数据库发生异常重启,之前提交的记录都不会丢失,这个能力称为crash-safe。
                                                                                                                                语句的原始逻辑,比如"给 ID=2 这一行的 c 字段加 1"。
                                                        基本概念: Mysql Server 层日志,称为 binlog(归档日志)。记录的是SQl语句原始的逻辑,采用追加的方式写入日志文件。
                                                                                                                                3、redo log 是循环写的,空间固定会用完;binlog 是可以追加写入的。"追写"是指 binlog文件写到一定
                                                                    1.基于SQL语句的复制(statement-based replication,SBR)每一条会修改数据的sql都会记录在binlog中
                                                                                                                                大小后会切换到下一个,并不会覆盖以前的日志。
                                                                   2.基于行的复制(row-based replication,RBR), 5.1.5版本才支持。仅保存哪条记录被修改
                      MySQL 日志系统
                                                                   3.混合模式复制(mixed-based replication,MBR),从5.1.8版本开始,MySQL提供了Mixed格式,实际上就
                                                                   是Statement与Row的结合
                                                                1.执行器先找引擎取 ID=2 这一行。ID 是主键,引擎直接用树搜索找到这一行。如果 ID=2 这一行所在的数据页本来就在内存中,就直接返回给执行器;否则,需要先从磁盘读入内存,然后再返回
                                                               2.执行器拿到引擎给的行数据,把这个值加上 1,比如原来是 N,现在就是 N+1,得到新的一行数据,再调用引擎接口写入这行新数据。
                                                               3.引擎将这行新数据更新到内存中,同时将这个更新操作记录到 redo log 里面,此时 redolog 处于 prepare 状态。然后告知执行器执行完成了,随时可以提交事务
                                       sql: update T set c=c+1 where ID=2;
                                                               4. 执行器生成这个操作的 binlog,并把 binlog 写入磁盘。
                                                               5.执行器调用引擎的提交事务接口,引擎把刚刚写入的 redo log 改成提交(commit)状态,更新完成。
                                                  基本概念 🔾 redo log 的写入被拆成了两个步骤: prepare 和 commit, 这就是"两阶段提交"。
```

1.每个列都不可以再拆分

两阶段提交 😑

1.确保 redo log 和 binlog 一致

2.在MySQL 异常重启或者扩容时,确保数据的一致性。

varchar(10)中的10代表字符串最长宽度为10,存进库里的字符串长度超过10将报错无法进行存储

int(10),存储为int类型,占4个字节,非负存值为0到2的32次方,并不会因为10的限制存储

范围为0到2的10次方,表示的是存储数字宽度,如果宽度不够10就会按照0进行填充。

1.Null 值问题?

常见问题

2.超大分页问题?

3.MyBatis中的#和\$有什么区别?

5.varchar(10)和int(10)代表什么含义?

4.varchar和char的区别?