




MySQL

Contents

- Contents
-  概述
 - 关系的三个范式是什么？
 - MySQL 中 varchar 和 char 的区别是什么？
 - join 和 left join 的区别？
 - SQL 怎么实现模糊查询？
 - select 的执行过程？
 - update 的执行过程？
 - count 性能比较？
 - drop、truncate 和 delete 的区别？
 - MySQL 会出现死锁吗，如何避免死锁？
 - 什么情况发生死锁
 - 解决办法：
-  事务
 - MySQL 之事务的四大特性(ACID)？
 - 并发事务会出现什么问题？
 - MySQL 的事务隔离级别？
 - 在不同事务隔离级别下会发生什么现象？
 - MVCC 实现原理？
 - 幻读是如何解决的？
 - 读提交是怎么实现的？
 - 可重复读是怎么实现的？
-  索引
 - MySQL 为什么使用 B+ 树来作索引，它的优势什么？
 - 特性和定义
 - 优势
 - 对比
 - 索引有哪些种类？
 - 什么是最左匹配原则？
 - 索引区分度？

- 联合索引如何进行排序?
- 使用索引会有哪些缺陷?
- 什么时候需要/不需要创建索引?
 - 需要创建索引
 - 不需要创建索引
- 索引的优化(使用索引的注意事项)?
- WHERE 语句索引使用的注意事项?
- 索引什么时候会失效?
-  锁
 - MySQL 的全局锁有什么作用?
 - MySQL 的表级锁有哪些? 作用是什么?
 - 元数据锁
 - 意向锁
 - AUTO-INC 锁
 - MySQL 的行级锁有哪些? 作用是什么?
 - 记录锁(Record Lock)
 - 间隙锁(Gap Lock)
 - 临键锁(Next-Key Lock)
 - 插入意向锁
 - MySQL 如何加锁
 - 唯一索引等值查询
 - 非唯一索引等值查询
 - 唯一索引范围查询
 - 非唯一索引范围查询
-  存储引擎
 - MySQL 的执行引擎有哪些?
 - MyISAM 和 InnoDB 存储引擎的区别?
 - 存储引擎如何选择?
 - MyISAM 索引与 InnoDB 索引的区别?
-  日志
 - MySQL 三种日志?
 - UNDO LOG(回滚日志)
 - REDO LOG(重做日志)
 - BINLOG(归档日志)
 - redo log 与 bin log 的区别?
 - redo log 和 undo log 区别?
 - undo log 是如何实现 MVCC 的?
 - 为什么有了 binlog, 还要有 redo log?

- 被修改 Undo 页面，需要记录对应 redo log 吗？
- binlog 的三种格式？
- redo log 容灾恢复过程？
- redo log 是直接写入硬盘的吗？
- redo log 比直接落盘的优点？
- redo log buffer 什么时候刷盘？
- redo log 文件文件格式和写入过程？
- binlog 什么时候刷盘？
- binlog 什么时候刷盘频率？
- 主从复制是怎么实现？
- MySQL 主从复制模型？
- 为什么需要两阶段提交？
- 两阶段提交的过程是怎样的？
- 异常重启会出现什么现象？
- 事务没提交 redo log 会被持久化到硬盘吗？
- 两阶段提交有什么问题？
- 组提交是什么意思？
- Buffer Pool 有什么作用？
- Buffer Pool 缓存内容？
- ● 优化
 - 慢查询的原因？
 - MySQL 硬盘 I/O 很高有什么优化的方

● 概述

关系的三个范式是什么？

- **第一范式(1NF)**：用来确保每列的原子性，要求每列(或者每个属性值)都是不可再分的最小数据单元(也称为最小的原子单元)
- **第二范式(2NF)**：在第一范式的基础上更进一层，要求表中的每列都和主键相关，即要求实体的唯一性。如果一个表满足第一范式，并且除了主键以外的其他列全部都依赖于该主键，那么该表满足第二范式
- **第三范式(3NF)**：在第二范式的基础上更进一层，第三范式是确保每列都和主键列直接相关，而不是间接相关，即限制列的冗余性。如果一个关系满足第二范式，并且除了主键以外的其他列都依赖于主键列，列和列之间不存在相互依赖关系，则满足第三范式

MySQL 中 varchar 和 char 的区别是什么？

- char 字段的最大长度为 255 字符，varchar 字段的最大长度为 65535 个字符
- char 类型如果存的数据量小于最大长度，剩余的空间会使用空格填充，因此可能会浪费空间，所以 char 类型适合存储长度固定的数据，这样不会浪费空间，效率还比 varchar 略高；varchar 类型如果存到数据量小于最大长度，剩余的空间会留给别的数据使用，所以 varchar 类型适合存储长度不固定的数据，这样虽然没有 char 存储效率高，但至少不会浪费空间
- char 类型的查找效率高，varchar 类型的查找效率较低

join 和 left join 的区别？

- join 等价于 inner join 内连接，是返回两个表中都有的符合条件的行
- left join 左连接，是返回左表中所有的行及右表中符合条件的行
- right join 右连接，是返回右表中所有的行及左表中符合条件的行

SQL 怎么实现模糊查询？

索引 B+ 树是按照索引值有序排列存储的，只能根据前缀进行比较。每一次按照模糊匹配的前缀字典序来进行比较。

select 的执行过程？

- **连接：**首先客户端和 MySQL 通过三次握手建立连接，MySQL 是基于 TCP 进行传输的。MySQL 服务如果没有启动就会报错。MySQL 正常运行的话就去校验用户名和密码，如果认证信息错误也会报错。检验通过之后连接器会获取用户权限并且保存起来，后续的任何操作都会基于开始的读到权限进行判断，即便创建连接之后更改了权限也不会影响已连接的权限

如何断开长连接

- **定期断开长连接**
- **客户端主动重置连接：**当客户端执行了一个很大的操作后，在代码里调用 `mysql_reset_connection` 函数来重置连接，达到释放内存的效果。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
- **查询缓存：**连接成功后会向 MySQL 服务发送 SQL 语句，MySQL 服务收到语句之后会进行解析判断 SQL 语句的类型。如果是 select 语句的话就去缓存中查询，看看之前有没有执行过这条 select 语句。缓存是以 k-v 形式保存在内存中的，key 是 SQL 语句，value 是 SQL 查询结果。如果缓存中有结果就直接返回给客户端，如果没有命中就继续向下执行。执行完成后的结果会被放入缓存中

缓存缺点

对于更新比较频繁的表，查询缓存的命中率很低，只要一个表有更新操作，那么这个表的查询缓存就会被清空。如果刚缓存了一个查询结果很大的数据，还没被使用的时候，刚好这个表有更新操作，查询缓冲就被清空了。

MySQL 8.0 版本直接将查询缓存删掉了，执行一条 SQL 查询语句，不会有查询缓存这个阶段了

- **SQL 解析**：词法分析和语法分析，词法分析是把 SQL 语句的字符串识别出关键字，方便后续优化，语法分析根据语法规则判断 SQL 语句是否满足要求。如果 SQL 语法不正确就会报错
- **SQL 执行**：主要是 prepare 预处理、optimize 优化和 execute 执行阶段
 - **预处理器检查 SQL 查询的表或者字段是否存在**，如果有就将它扩展为 SQL 的所有的列
 - **优化器是确定 SQL 语句的执行方案**，比方说有索引会选择走了哪个索引
 - **执行器会与存储引擎交互**，如果走索引了就将相应索引条件交给存储引擎
 - **存储引擎通过 B+ 树定位数据**，如果数据不存在就像执行器返回错误，然后查询结束；找到了就将记录返回给执行数，执行器读到数据之后判断记录是否满足要求，如果满足要求就将数据返回给客户端，否则跳过该数据
 - **全表扫描优化器**和存储引擎交互之后存储引擎会访问第一条表中数据，执行器会判断这条数据是否满足条件，满足就发给客户端。
 - **执行器查询是一个 while 循环**，继续取下一条记录重复判断，直到读完表中所有记录退出循环。如果使用联合索引，会在存储引擎层分别判断每个索引是否满足条件，而不先执行回表，所有索引有一个不成立就跳过；否则就返回给 Server 层回表，这是一个**索引下推**的过程

update 的执行过程？

执行器负责具体执行，会调用存储引擎的接口，通过主键索引树搜索获取一行记录：

- 如果记录所在的数据页本来就在 buffer pool 中，就直接返回给执行器更新
- 如果记录不在 buffer pool，将数据页从硬盘读入到 buffer pool，返回记录给执行器

执行器得到聚簇索引记录后，会**比较更新前的记录和更新后的记录是否相同**：

- 如果一样就不进行后续更新流程
- 如果不一样就把更新前的记录和更新后的记录都当作参数传给 InnoDB 层，让 InnoDB 真正的执行更新记录的操作
- **开启事务**：首先要记录相应的 undo log，需要把被更新的列的旧值记下来，也就是要生成一条 undo log，undo log 会写入 Buffer Pool 中的 Undo 页面，不过在内存修改该 Undo 页面后，需要记录对应的 redo log

InnoDB 层开始更新记录，会先更新内存(同时标记为脏页)，然后将记录写到 redo log 里面，这个时候更新就算完成了。为了减少硬盘 I/O，不会立即将脏页写入硬盘，后续由后台线程选择一个合适的时机将脏页写入到硬盘。

在一条更新语句执行完成后，然后开始记录该语句对应的 binlog，此时记录的 binlog 会被保存到 binlog cache，在事务提交时才会统一将该事务运行过程中的所有 binlog 刷新到硬盘。

两阶段提交：

- **prepare 阶段：**将 redo log 对应的事务状态设置为 prepare，然后将 redo log 刷新到硬盘；
- **commit 阶段：**将 binlog 刷新到硬盘，接着调用引擎的提交事务接口，将 redo log 状态设置为 commit(将事务设置为 commit 状态后，刷入到硬盘 redo log 文件)

count 性能比较？

count(*) = count(1) > count(主键) > count(字段)

MySQL 会将星号参数转化为参数 0 来处理，所以 count(*) 和 count(1)相等。count(主键)需要判断主键是否为空值；count(字段)会进行全表扫描，效率最差。

drop、truncate 和 delete 的区别？

drop 删除整张表和表结构，以及表的索引、约束和触发器；**truncate** 只删除表数据，表的结构、索引、约束等会被保留；**delete** 只删除表的全部或部分数据，表结构、索引、约束等会被保留。

- **delete 语句为 DML(data maintain Language)**，执行删除操作的过程是每次从表中删除一行，并且同时将该行的删除操作作为事务记录在日志中保存以便进行回滚操作
- **truncate、drop 是 DDL(data define language)**，删除行是不能恢复的，并且在删除的过程中不会激活与表有关的删除触发器，执行速度快，原数据不放到 rollback segment 中，不能回滚。
- **truncate 和 drop 不支持添加 where 条件，而 delete 支持 where 条件**
- 执行速度 **drop > truncate > delete**，delete 是逐行执行的，并且在执行时会把操作日志记录下来，以备日后回滚使用，所以 delete 的执行速度是比较慢的；而 truncate 的操作是先复制一个新的表结构，再把原先的表整体删除，所以它的执行速度居中，而 drop 的执行速度最快。
- truncate 只能对 TABLE；delete 可以是 TABLE 和 VIEW

使用场景

- 如果想删除表用 drop
- 如果想保留表而将所有数据删除，和事务无关，用 truncate 即可
- 如果和事务有关，或者想触发 trigger，用 delete

- 如果是整理表内部的碎片，可以用 truncate 跟上 reuse storage，再重新导入/插入数据

MySQL 会出现死锁吗，如何避免死锁？

什么情况发生死锁

如果 update 语句的 where 条件没有用到索引列，那么就会全表扫描，在一行行扫描的过程中，不仅给行记录加上了行锁，还给行记录两边的空隙也加上了间隙锁，相当于锁住整个表，然后直到事务结束才会释放锁。

行锁会发生死锁，表锁不会。死锁的四个必要条件：**互斥、占有且等待、不可强占用、循环等待**。只要系统发生死锁，这些条件必然成立，但是只要破坏任意一个条件就死锁就不会成立。

解决办法：

- **设置事务等待锁的超时时间**：当一个事务的等待时间超过该值后，就对这个事务进行回滚，于是锁就释放了，另一个事务就可以继续执行了。在 InnoDB 中，参数 `innodb_lock_wait_timeout` 是用来设置超时时间的，默认值是 50 秒
- **应用乐观锁**：在应用层控制并发操作
- **避免长查询**：简化事务，减少复杂查询
- **优化事务设计**：合理顺序获取锁，减少锁持有时间
- **开启主动死锁检测**：主动死锁检测在发现死锁后，主动回滚死锁链条中的某一个事务，让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，默认就开启。当检测到死锁后，就会出现提示

● 事务

MySQL 之事务的四大特性(ACID)?

- **原子性(atomicity)**：一个事务必须视为一个不可分割的最小工作单元，整个事务中的所有操作要么全部提交成功，要么全部失败回滚，对于一个事务来说，不可能只执行其中的一部分操作，这就是事务的原子性
- **一致性(consistency)**：数据库总是从一个一致性的状态转换到另一个一致性的状态
- **隔离性(isolation)**：一个事务所做的修改在最终提交以前，对其他事务是不可见的
- **持久性(durability)**：一旦事务提交，则其所做的修改就会永久保存到数据库中。此时即使系统崩溃，修改的数据也不会丢失

实现

- **原子性**：通过 undo log 来保证
- **一致性**：通过持久性 + 原子性 + 隔离性来保证
- **隔离性**：通过 MVCC 或锁机制来保证
- **持久性**：通过 redo log 来保证

并发事务会出现什么问题？

- **脏读**：读到了其他事务未提交的数据。未提交意味着这些数据可能会回滚，也就是可能最终不会存到数据库中，也就是不存在的数据。读到了不一定最终存在的数据，这就是脏读
- **不可重复读**：对比可重复读，不可重复读指的是在同一事务内，不同的时刻读到的同一批数据可能是不一样的，可能会受到其他事务的影响，比如其他事务改了这批数据并提交。通常针对数据更新操作
- **可重复读**：可重复读指的是在一个事务内，最开始读到的数据和事务结束前的任意时刻读到的同一批数据都是一致的
- **幻读**：幻读是针对数据插入操作来说的。假设事务 A 对某些行的内容作了更改，但是还未提交，此时事务 B 插入了与事务 A 更改前的记录相同的记录行，并且在事务 A 提交之前先提交了，而这时，在事务 A 中查询，会发现好像刚刚的更改对于某些数据未起作用，让用户感觉感觉出现了幻觉，这就叫幻读。简而言之，**当一个事务前后两次查询的结果集不相同，就认为发生幻读**

MySQL 的事务隔离级别？

- **读未提交(read uncommitted)**：指一个事务还没提交时，它做的变更就能被其他事务看到
- **读提交(read committed)**：指一个事务提交之后，它做的变更才能被其他事务看到
- **可重复读(repeatable read)**：指一个事务执行过程中看到的数据，一直跟这个事务启动时看到的数据是一致的，MySQL InnoDB 引擎的默认隔离级别
- **串行化(serializable)**：会对记录加上读写锁，在多个事务对这条记录进行读写操作时，如果发生了读写冲突的时候，后访问的事务必须等前一个事务执行完成，才能继续执行

在不同事务隔离级别下会发生什么现象？

- **读未提交**：可能发生脏读、不可重复读和幻读现象
- **读提交**：可能发生不可重复读和幻读现象，但是不可能发生脏读现象
- **可重复读**：可能发生幻读现象，但是不可能脏读和不可重复读现象
- **串行化**：隔离级别下，脏读、不可重复读和幻读现象都不可能发生

💡提示

解决脏读现象：升级到读提交以上的隔离级别

解决不可重复读：升级到可重复读的隔离级别

解决幻读：不建议将隔离级别升级到串行化，因为这样会导致数据库在**并发事务时性能很差**

MVCC 实现原理？

Read View 有四个重要的字段：

- **m_ids**：指的是在创建 Read View 时，当前数据库中活跃事务的事务 id 列表，活跃事务指的是，启动了但还没提交的事务
- **min_trx_id**：指的是在创建 Read View 时，当前数据库中活跃事务中事务 id 最小的事务，也就是 m_ids 的最小值
- **max_trx_id**：创建 Read View 时当前数据库中应该给下一个事务的 id 值，也就是全局事务中最大的事务 id 值 + 1
- **creator_trx_id**：指的是创建该 Read View 的事务的事务 id

对于使用 InnoDB 存储引擎的数据库表，它的聚簇索引记录中都包含下面两个隐藏列：

- **trx_id**：当一个事务对某条聚簇索引记录进行改动时，就会把该事务的事务 id 记录在 trx_id 隐藏列里。
- **roll_pointer**：每次对某条聚簇索引记录进行改动时，都会把旧版本的记录写入到 undo 日志中，然后这个隐藏列是个指针，指向每一个旧版本记录，于是就可以通过它找到修改前的记录

一个事务去访问记录的时候，除了自己的更新记录总是可见之外，还有这几种情况：

- 如果记录的 trx_id 值小于 Read View 中的 min_trx_id 值，表示这个版本的记录是在创建 Read View 前已经提交的事务生成的，所以该版本的记录对当前事务可见。
- 如果记录的 trx_id 值大于等于 Read View 中的 max_trx_id 值，表示这个版本的记录是在创建 Read View 后才启动的事务生成的，所以该版本的记录对当前事务不可见。
- 如果记录的 trx_id 值在 Read View 的 min trx id 和 max trx id 之间，需要判断 trx_id 是否在 m_ids 列表中：
 - 如果记录的 trx_id 在 m_ids 列表中，表示生成该版本记录的活跃事务依然活跃着(还没提交事务)，所以该版本的记录对当前事务不可见。
 - = 如果记录的 trx_id 不在 m_ids 列表中，表示生成该版本记录的活跃事务已经被提交，所以该版本的记录对当前事务可见

幻读是如何解决的？

- **快照读(普通 select 语句)**：是通过 MVCC 方式解决了幻读，可重复读隔离级别下，事务执行过程中看到的数据，一直跟这个事务启动时看到的数据是一致的，即使中途有其他事务插入了一条数据，查询不出来这条数据的
- **当前读(select ... for update 等语句)**：是通过 next-key lock(记录锁+间隙锁)方式解决了幻读，因为当执行 select ... for update 语句的时候会加上 next-key lock，如果有其他事务在 next-key lock 锁范围内插入了一条记录，那么这个插入语句就会被阻塞，无法成功插入

失败的情况

对于快照读：MVCC 并不能完全避免幻读现象。当事务 A 更新了一条事务 B 插入的记录，那么事务 A 前后两次查询的记录条目就不一样了，所以就发生幻读

对于当前读：如果事务开启后，并没有执行当前读，而是先快照读，然后这期间如果其他事务插入了一条记录，那么事务后续使用当前读进行查询的时候，就会发现两次查询的记录条目就不一样了，所以就发生幻读

即 MySQL 可重复读隔离级别并没有彻底解决幻读，只是很大程度上避免了幻读现象的发生

尽量在开启事务之后，马上执行 select ... for update 这类当前读的语句，因为它会对记录加 next-key lock，从而避免其他事务插入一条新记录

读提交是怎么实现的？

读提交隔离级别是在**每次读取数据时，都会生成一个新的 Read View**。事务期间的多次读取同一条数据，前后两次读的数据可能会出现不一致，因为可能这期间**另外一个事务修改了该记录，并提交了事务**。

可重复读是怎么实现的？

可重复读隔离级别是在**启动事务时生成一个 Read View，然后整个事务期间都在用这个 Read View**。事务期间多次读取同一条数据，前后两次读的数据若被修改过，则会因为记录中的 **trx_id** 仍然在 **m_ids** 范围内，导致程序通过 **roll_pointer** 获取所指向的前一个版本对应的记录，以此类推，最终找到不在 m_ids 队列中的事务对应的记录。不会导致正在执行事务在同一位置读到不同的记录。

索引

MySQL 为什么使用 B+ 树来作索引，它的优势什么？

特性和定义

B+ Tree 是一种多叉树，叶子节点才存放数据，非叶子节点只存放索引，每个节点里的数据是按主键顺序存放的。在叶子节点中，包括了所有的索引值信息，并且每一个叶子节点都指向下一个叶子节点，形成一个链表。B+ Tree 存储千万级的数据只需要 3-4 层高度就可以满足，千万级的表查询目标数据最多需要 3-4 次硬盘 I/O。

B+ 树和 B 树相比：

- B+ 树**所有关键码都存放在叶节点中**，上层的非叶节点的关键码是其子树中最小关键码的复写
- B+ 树**叶节点包含了全部关键码及指向相应数据记录存放地址的指针**，且叶节点本身按关键码从小到大顺序连接
- B+ 树在搜索过程中，**如果查询和内部节点的关键字一致，那么搜索过程不停止，而是继续向下搜索这个分支**

优势

- **单点查询**：B 树进行单个索引查询时，最快可以在 $O(1)$ 的时间代价内就查到。从平均时间代价来看，会比 B+ 树稍快一些。但是 B 树的查询波动会比较大，因为每个节点即存索引又存记录，所以有时候访问到了非叶子节点就可以找到索引，而有时需要访问到叶子节点才能找到索引。B+ 树的非叶子节点不存放实际的记录数据，仅存放索引，数据量相同的情况下，B+ 树的非叶子节点可以存放更多的索引，查询底层节点的硬盘 I/O 次数会更少
- **插入和删除效率**：B+ 树有大量的冗余节点，删除一个节点的时候，可以直接从叶子节点中删除，甚至可以不动非叶子节点，删除非常快。B+ 树的插入也是一样，有冗余节点，插入可能存在节点的分裂(如果节点饱和)，但是最多只涉及树的一条路径。B 树没有冗余节点，删除节点的时候非常复杂，可能涉及复杂的树的变形
- **范围查询**：B+ 树所有叶子节点间有一个链表进行连接，而 B 树没有将所有叶子节点用链表串联起来的结构，因此只能通过树的遍历来完成范围查询，范围查询效率不如 B+ 树。B+ 树的插入和删除效率更高。存在大量范围检索的场景，适合使用 B+ 树，比如数据库。而对于大量的单个索引查询的场景，可以考虑 B 树，比如 nosql 的 MongoDB

对比

- **B+ Tree 对比 B Tree**: B+Tree 只在叶子节点存储数据, 而 B 树 的非叶子节点也要存储数据, 所以 B+Tree 的单个节点的数据量更小, 在相同的硬盘 I/O 次数下, 就能查询更多的节点。B+ Tree 叶子节点采用的是双链表连接, 适合 MySQL 中常见的基于范围的顺序查找, 而 B 树无法做到这一点
- **B+ Tree 对比 二叉树**: 对于有 N 个叶子节点的 B+ Tree, 其搜索复杂度为 $O(\log_d N)$, 其中 d 表示节点允许的最大子节点个数。在实际的应用当中, d 值是大于 100 的, 即使数据达到千万级别时, B+ Tree 的高度依然维持在 3~4 层左右, 一次数据查询操作只需要做 3~4 次的硬盘 I/O 操作就能查询到。二叉树的每个父节点的儿子节点个数是 2 个, 意味着其搜索复杂度为 $O(\log_2 N)$, 二叉树检索到目标数据所经历的硬盘 I/O 次数要更多
- **B+ Tree 对比 Hash**: Hash 在做等值查询的时候效率高, 搜索复杂度为 $O(1)$ 。但是 Hash 表不适合做范围查询

索引有哪些种类?

- **单值索引**: 即一个索引只包含单个列, 一个表可以有多个单列索引
 - 建表时加上 **key(列名)** 指定
 - 单独创建: **create index 索引名 on 表名(列名)**
 - 单独创建: **alter table 表名 add index 索引名(列名)**
- **唯一索引**: 索引列的值必须唯一, 但允许有 null 且 null 可以出现多次
 - 建表时加上 **unique(列名)** 指定
 - 单独创建: **create unique index idx 表名(列名) on 表名(列名)**
 - 单独创建: **alter table 表名 add unique 索引名(列名)**
- **主键索引**: 设定为主键后数据库会自动建立索引, innodb 为聚簇索引, 值**必须唯一且不能为 null**
 - 建表时加上 **primary key(列名)** 指定
- **复合索引**: 即一个索引包含多个列
 - 建表时加上 **key(列名列表)** 指定
 - 单独创建: **create index 索引名 on 表名(列名列表)**
 - 单独创建: **alter table 表名 add index 索引名(列名列表)**
- **前缀索引**: 对字符类型字段的前几个字符建立的索引, 而不是在整个字段上建立的索引, 前缀索引可以建立在字段类型为 **char**、**varchar**、**binary**、**varbinary** 的列上。使用前缀索引的目的是为了减少索引占用的存储空间, 提升查询效率
 - 单独创建: **alter table 表名 add 索引名(column_name(索引长度))**

什么是最左匹配原则？

使用联合索引时，存在最左匹配原则，也就是按照最左优先的方式进行索引的匹配。使用联合索引进行查询的时候，如果不遵循最左匹配原则，联合索引会失效。

以联合索引(a, b, c)为例

因为 (a, b, c) 联合索引是先按 a 排序；在 a 相同的情况再按 b 排序；在 b 相同的情况再按 c 排序。所以 **b 和 c 是全局无序，局部相对有序的**，这样在没有遵循最左匹配原则的情况下，是无法利用到索引的。**利用索引的前提是索引里的 key 是有序的**

联合索引的最左匹配原则在遇到范围查询(如 >、<)的时候停止匹配，范围查询的字段可以用到联合索引，在范围查询字段的后面的字段无法用到联合索引。

注意：对于 >=、<=、BETWEEN、like 前缀匹配的范围查询并不会停止匹配

索引区分度？

查询优化器发现某个值出现在表的数据行中的百分比(惯用的百分比界线是"30%")很高的时候会忽略索引，进行全表扫描。

联合索引如何进行排序？

给索引列和排序列建立一个联合索引，在查询时，查到一个索引之后，还要对 create_time 排序，用到文件排序 filesort，在 SQL 执行计划中，Extra 列会出现 Using filesort。

可以利用索引的有序性，在排序列建立联合索引，这样根据 status 筛选后的数据就是按照 create_time 排好序的，避免在文件排序，提高了查询效率。

使用索引会有哪些缺陷？

虽然索引大大提高了查询速度，同时却会**降低更新表的速度**，如对表进行 INSERT、UPDATE 和 DELETE。

因为更新表时，MySQL **不仅要保存数据，还要保存索引文件**每次更新添加了索引列的字段，都会调整因为更新所带来的键值变化后的索引信息。

实际上索引也是一张表，该表保存了主键与索引字段，并指向实体表的记录，所以索引列也是要占用空间的。

什么时候需要/不需要创建索引?

需要创建索引

- 表的主关键字：自动建立唯一索引
- 表的字段唯一约束：利用索引来保证数据的完整
- 直接条件查询的字段：经常用于 WHERE 查询条件的字段，这样能够提高整个表的查询速度
- 查询中与其它表关联的字段：例如字段建立了外键关系
- 查询中排序的字段：排序的字段如果通过索引去访问将大大提高排序速度
- 查询中统计或分组统计的字段：经常用于 GROUP BY 和 ORDER BY 的字段，可以创建联合索引

不需要创建索引

- 表记录太少：表数据太少的时候，不需要创建索引
- 经常插入、删除、修改的字段：经常更新的字段不用创建索引，索引字段频繁修改，由于要维护 B+ Tree 的有序性，那么就需要频繁的重建索引，会影响数据库性能
- 数据重复且分布平均的表字段：假如一个表有10万行记录，性别只有男和女两种值，且每个值的分布概率大约为50%，那么对这种字段建索引一般不会提高数据库的查询速度。
- 经常和主字段一块查询但主字段索引值比较多的表字段

索引的优化(使用索引的注意事项)?

- like 语句的前导模糊查询不能使用索引

```
select * from doc where title like '%XX';    --不能使用索引
select * from doc where title like 'XX%';    --非前导模糊查询，可以使用索引
```

union、in、or 都能够命中索引，建议使用 in，因为 in 的综合效率最高。

- 负向条件查询不能使用索引

负向条件有：!=、<>、not in、not exists、not like 等，优化案例：

```
select * from doc where status != 1 and status != 2;    --优化前
select * from doc where status in (0,3,4);              --优化后
```

- 联合索引最左前缀原则

如果在 (a, b, c) 三个字段上建立联合索引，那么他会自动建立 a| (a,b) | (a,b,c) 组索引。

- 建立联合索引的时候，区分度最高的字段在最左边
- 存在非等号和等号混合判断条件时，在建立索引时，把等号条件的列前置。如 where a=? and b=?，那么即使 a 的区分度更高，也必须把 b 放在索引的最前列

- 最左前缀查询时，并不是指 SQL 语句的 where 顺序要和联合索引一致
- **不能使用索引中范围条件右边的列(范围列可以用到索引)，范围列之后列的索引全失效**
 - 范围条件有：<、<=、>、>=、between 等
 - 索引最多用于一个范围列，如果查询条件中有两个范围列则无法全用到索引
 - 假如有联合索引 (empno、title、fromdate)，那么下面的 SQL 中 emp_no 可以用到索引，而 title 和 from_date 则使用不到索引

```
select * from employees.titles where emp_no < 10010' and title='Senior Engineer'and from_
```

- **不要在索引列上面做任何操作(计算、函数)，否则会导致索引失效而转向全表扫描**

```
select * from doc where YEAR(create_time) <= '2016'; --优化前`
select * from doc where create_time <= '2016-01-01'; --优化后
```

- **强制类型转换会全表扫描**

```
select * from doc where YEAR(create_time) <= '2016'; --优化前
select * from doc where create_time <= '2016-01-01'; --优化后
```

- **更新十分频繁、数据区分度不高的列不宜建立索引**

- 更新会变更 B+ 树，更新频繁的字段建立索引会大大降低数据库性能
- "性别"这种区分度不大的属性，建立索引是没有什么意义的，不能有效过滤数据，性能与全表扫描类似
- 一般区分度在 80% 以上的时候就可以建立索引，区分度可以使用 **count(distinct(列名)) / count(*)** 来计算

- **利用覆盖索引来进行查询操作，避免回表，减少 select * 的使用**

- **覆盖索引**：查询的列和所建立的索引的列个数相同，字段相同
- **被查询的列**：数据能从索引中取得，而不用通过行定位符 row-locator 再到 row 上获取，即“被查询列要被所建的索引覆盖”，这能够加速查询速度
- 可以建立 (login_name, passwd, login_time) 的联合索引，由于 login_time 已经建立在索引中了，被查询的 uid 和 login_time 就不用去 row 上获取数据了，从而加速查询

```
select uid, login_time from user where login_name=? and passwd=?;
```

- **索引不会包含有 NULL 值的列，IS NULL，IS NOT NULL 无法使用索引**

- 只要列中包含有 NULL 值都不会被包含在索引中，复合索引中只要有一列含有 NULL 值，那么这一列对于此复合索引就是无效的。所以在数据库设计时，尽量使用 NOT NULL 约束以及默认值

- **如果有 order by、group by 的场景，利用索引的有序性**

- order by 最后的字段是组合索引的一部分，并且放在索引组合顺序的最后，避免出现 file_sort 的情况，影响查询性能
- 使用短索引(前缀索引)
 - 对列进行索引，如果可能应该**指定一个前缀长度**。例如，如果有一个 CHAR(255) 的列，如果该列在前 10 个或 20 个字符内，可以做到既使得前缀索引的区分度接近全列索引，那么就**不要对整个列进行索引**。因为短索引不仅可以提高查询速度而且可以节省硬盘空间和 I/O 操作，减少索引文件的维护开销。可以使用 `count(distinct leftIndex(列名, 索引长度)) / count(*)` 来计算前缀索引的区分度
 - 缺点是不能用于 ORDER BY 和 GROUP BY 操作，也不能用于覆盖索引
 - 很多时候没必要对全字段建立索引，根据实际文本区分度决定索引长度即可
- 利用延迟关联或者子查询优化超多分页场景
 - MySQL 并不是跳过 offset 行，而是取 **offset + N** 行，然后返回放弃前 offset 行，返回 N 行，那当 offset 特别大的时候，效率就非常的低下，要么控制返回的总页数，要么对超过特定阈值的页数进行 SQL 改写
- 如果明确知道只有一条结果返回，limit 1 能够提高效率

```
select * from user where login_name=?;  
// 可以优化为  
select * from user where login_name=? limit 1
```

- 超过三个表最好不要 join
 - 需要 join 的字段，数据类型必须一致，多表关联查询时，保证被关联的字段需要有索引
 - 例如：left join 是由左边决定的，左边的数据一定都有，所以右边是我们的关键点，建立索引要建右边的。当然如果索引在左边，可以用 right join
- 单表索引建议控制在 5 个以内
- SQL 性能优化 explain 中的 type：至少要达到 range 级别，要求是 ref 级别，如果可以是 consts 最好
 - **consts**：单表中最多只有一个匹配行(主键或者唯一索引)，在优化阶段即可读取到数据
 - **ref**：使用普通的索引(Normal Index)
 - **range**：对索引进行范围检索
 - 当 type=index 时，索引物理文件全扫，速度非常慢
- 业务上具有唯一特性的字段，即使是多个字段的组合，也必须建成唯一索引
 - 不要以为唯一索引影响了 insert 速度，这个速度损耗可以忽略，但提高查找速度是明显的

WHERE 语句索引使用的注意事项？

- where 子句使用的所有字段，都必须建立索引

- 确保 **MySQL 版本 5.0 以上**，且查询优化器开启了 **index_merge_union = on**，也就是变量 optimizer_switch 里存在 index_merge_union 且为 on

索引什么时候会失效？

- 查询条件中**带有 or**，除非所有的查询条件都建有索引，
- **like 查询是以 % 开头**，索引会失效
- 如果列类型是字符串，那在查询条件中需要将数据用引号引用起来，否则索引失效
- 索引列上参与计算，索引失效
- **违背最左匹配原则**，索引失效
- 如果 **MySQL 估计全表扫描要比使用索引要快**，索引失效

锁

MySQL 的全局锁有什么作用？

- **作用**：让整个数据库处于**只读状态**，增删改会被阻塞
- **使用场景**：全局锁**主要应用于做全库逻辑备份**，不会因为数据或表结构的更新而出现备份文件的数据与预期的不一样
- **缺陷**：数据库里有很多数据，备份会花费很多的时间。备份期间，业务**只能读数据而不能更新数据**，这样会造成业务停滞
- **改进**：可重复读的隔离级别。在备份数据库之前先开启事务，会先创建 Read View，然后整个事务执行期间都在用这个 Read View，而且由于 MVCC 的支持，备份期间业务依然可以对数据进行更新操作。即使其他事务更新了表的数据，也不会影响备份数据库时的 Read View，这样备份期间备份的数据一直是在开启事务时的数据

MySQL 的表级锁有哪些？作用是什么？

元数据锁

- **作用**：对数据库表进行操作时**自动**给这个表加上元数据锁。可以保证当用户对表执行 CRUD 操作时其他线程对这个表结构做了变更。元数据锁在事务提交后才会释放

意向锁

作用：对某些记录加上「共享锁」之前，需要先在表级别加上一个「意向共享锁」，对某些记录加上「独占锁」之前，需要先在表级别加上一个「意向独占锁」。普通的 select 是不会加行级锁的，普通的 select 语句是利用 MVCC 实现一致性读，是无锁的

- 意向共享锁和意向独占锁是表级锁，不会和行级的共享锁和独占锁发生冲突，意向锁之间也不会发生冲突，只会和共享表锁和独占表锁发生冲突。意向锁的目的是为了快速判断表里是否有记录被加锁

Tips

select 也是可以对记录加共享锁和独占锁的

AUTO-INC 锁

- **作用：**表里的主键通常在设置成自增后可以在插入数据时不指定主键的值，数据库会自动给主键递增赋值。**递增值是通过 AUTO-INC 锁实现的。**在插入数据时会加一个表级别的 AUTO-INC 锁并将 AUTO_INCREMENT 修饰的字段递增赋值，等插入语句执行完成后才会把 AUTO-INC 锁释放掉。在此期间其他事务向该表插入语句都会被阻塞，从而保证插入数据时字段的值是连续递增的
- **缺陷：**对大量数据进行插入的时会影响插入性能，因为其他事务中的插入会被阻塞
- **改进：**InnoDB 存储引擎提供了一种轻量级的锁来实现自增。在插入数据的时候，会为被 AUTO_INCREMENT 修饰的字段加上轻量级锁，然后给该字段赋值一个自增的值，就把这个轻量级锁释放了，而不需要等待整个插入语句执行完后才释放锁

AUTO-INC 锁控制

设置 innodb_autoinc_lock_mode 的系统变量，是用来控制选择用 AUTO-INC 锁

- 当 innodb_autoinc_lock_mode = 0，采用 AUTO-INC 锁，语句执行结束后才释放锁
- 当 innodb_autoinc_lock_mode = 2，采用轻量级锁，申请自增主键后就释放锁，并不需要等语句执行后才释放
- 当 innodb_autoinc_lock_mode = 1
 - 普通 insert 语句，自增锁在申请之后就马上释放
 - 类似 insert ... select 这样的批量插入数据的语句，自增锁要等语句结束后才被释放

innodb_autoinc_lock_mode = 2 是性能最高的方式，但是当搭配 binlog 的日志格式是 statement 时，在主从复制的场景中会发生**数据不一致**的问题

binlog 日志格式要设置为 row，这样在 binlog 里面记录的是主库分配的自增值。到备库执行的时候，主库的自增值是什么，从库的自增值就是什么

所以，当 `innodb_autoinc_lock_mode = 2` 并且 `binlog_format = row` 时既能提升并发性，又不会出现数据一致性问题

MySQL 的行级锁有哪些？作用是什么？

记录锁(Record Lock)

- **作用：**锁住的是一条记录，记录锁分为**共享锁(S 锁)**和**排他锁(X 锁)**

间隙锁(Gap Lock)

- **作用：**只存在于可重复读隔离级别，目的是为了**解决可重复读隔离级别下幻读的现象**。间隙锁之间是兼容的，两个事务可以同时持有包含共同间隙范围的间隙锁，并不存在互斥关系

临键锁(Next-Key Lock)

Next-Key Lock 临键锁是 **Record Lock + Gap Lock** 的组合

- **作用：**锁定一个范围，并且锁定记录本身。next-key lock 既能保护该记录，又能阻止其他事务将新纪录插入到被保护记录前面的间隙中

插入意向锁

一个事务在插入一条记录的时候，需要判断插入位置是否已被其他事务加了间隙锁(next-key lock 也包含间隙锁)。如果有的话，插入操作就会发生阻塞，直到拥有间隙锁的那个事务提交为止，在此期间会生成一个插入意向锁，表明有事务想在某个区间插入新记录，但是现在处于等待状态。

MySQL 如何加锁

唯一索引等值查询

- 当查询的记录**存在**，在索引树上定位到对应记录后，将该记录的 **next-key lock** 退化成**记录锁**
- 当查询的记录**不存在**，在索引树上定位到第一条大于该查询的记录后，将该记录的 **next-key lock** 退化成**间隙锁**

非唯一索引等值查询

- 当查询的记录**存在**，由于不是唯一索引，所以有可能存在索引值相同的记录。在扫描的过程中对扫描到的二级索引记录加 **next-key lock**；对于第一个不符合条件的二级索引记录退化成**间隙锁**；符

合查询条件的记录的主键索引加**记录锁**

- 当查询的记录**不存在**，扫描的第一条不符合条件的二级索引记录退化成**间隙锁**；因为不存在满足条件的查询记录，因此不会对主键索引加

唯一索引范围查询

- 当查询条件为 $>$ ：对大于当前条件的所有索引加 **next-key lock**；对最大索引后面的伪记录加 **next-key lock**。相当于在区间 $(value, +\infty)$ 加锁
- 当查询条件为 \geq ：对等于当前条件的索引加**记录锁**；对大于当前条件的所有索引加 **next-key lock**；对最大索引后面的伪记录加 **next-key lock**。相当于在区间 $(value, +\infty)$ 加锁
- 当查询条件为 $<$ ：对小于当前条件的所有索引加 **next-key lock**；对第一个不满足小于条件的记录左端加 **间隙锁**。相当于在区间 $(-\infty, value)$ 加 **next-key lock**；在区间 $(pre-value, value]$ 加**间隙锁**
- 当查询条件为 \leq ：
 - 如果限制条件对应的记录存在：在区间 $(-\infty, value]$ 加 **next-key lock**
 - 如果限制条件对应的记录不存在：对小于当前条件的所有索引加 **next-key lock**；对第一个不满足小于条件的记录左端加 **间隙锁**。相当于在区间 $(-\infty, value)$ 加 **next-key lock**；在区间 $(pre-value, next-value)$ 加**间隙锁**

非唯一索引范围查询

- 索引的 **next-key lock** 不会有退化为**间隙锁**和**记录锁**的情况
- 非唯一索引进行范围查询时，对二级索引记录加锁**都是加 next-key 锁**

● 存储引擎

MySQL 的执行引擎有哪些？

主要有 MyISAM、InnoDB、Memery 等引擎：

- InnoDB 引擎提供了对事务 ACID 的支持，还提供了行级锁和外键的约束
- MyISAM 引擎不支持事务，也不支持行级锁和外键约束
- Memery 就是将数据放在内存中，数据处理速度很快，但是安全性不高

MyISAM 和 InnoDB 存储引擎的区别？

- **锁的细粒度不同**：InnoDB 比 MyISAM 更好的支持并发，因为 InnoDB 的支持行锁，而 MyISAM 支持表锁，行锁对每一条记录上锁，所以开销更大，但是可以解决脏读和不可重复读的问题，相对来

说也更容易发生死锁

- **可恢复性**：InnoDB 有事务日志，数据库崩溃后可以通过日志进行恢复，MyISAM 没有日志支持
- **查询性能**：MyISAM 要好于 InnoDB，因为 InnoDB 在查询过程中是在维护数据缓存。并且先要定位到所在数据块，然后从数据块定位到数据内存地址来查找数据
- **表结构文件**：MyISAM 的表结构文件包括 .frm(表结构定义)，.MYI(索引)、.MYD(数据)；而 InnoDB 的表数据文件为 .ibd(数据和索引集中存储)和 .frm(表结构定义)
- **记录存储顺序**：MyISAM 按照记录插入顺序，InnoDB 按照主键大小顺序有序插入
- **外键和事务**：MyISAM 均不支持，InnoDB 支持。对于 InnoDB 每一条 SQL 语言都默认封装成事务，自动提交，这样会影响速度，所以最好把多条 SQL 语言放在 begin 和 commit 之间，组成一个事务。对一个包含外键的 InnoDB 表转为 MYISAM 会失败
- **操作速度**：对于 SELECT 前者更优，INSERT、UPDATE、DELETE 后者更优。select count(*)使用 MyISAM 更快，因为内部维护了一个计数器，可以直接调度
- **存储空间**：MyISAM 可被压缩，存储空间较小，InnoDB 的表需要更多的内存和存储，会在主内存中建立专用的缓冲池用于高速缓存数据和索引
- **索引方式**：二者都是 B+ 树索引，前者是堆表，后者是索引组织表

为什么 InnoDB 没有计数器变量？

因为 InnoDB 的事务特性，同一时刻表中的行数对于不同事务而言是不同的，因此计数器统计的是当前事务对应的行数，而不是总行数

存储引擎如何选择？

如果没有特别的需求，使用默认的 InnoDB 即可。

要支持事务选择 InnoDB，如果不需要可以考虑 MyISAM；如果表中绝大多数都只是读查询考虑 MyISAM，如果既有读也有写使用 InnoDB 存储引擎。

系统崩溃后，MyISAM 恢复起来更困难，能否接受系统崩溃的程度；MySQL5.5 版本开始 InnoDB 已经成为 MySQL 的默认引擎(之前是 MyISAM)，说明其优势是有目共睹的。

MyISAM 索引与 InnoDB 索引的区别？

- InnoDB 是聚簇索引，MyISAM 是非聚簇索引
- InnoDB 的主键索引的叶子节点存储着行数据，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



MySQL 三种日志？

UNDO LOG(回滚日志)

undo log 是 InnoDB 存储引擎层生成的日志，实现了事务中的原子性，主要用于事务回滚和 MVCC。

在事务没提交之前，InnoDB 会先记录更新前的数据到 undo log 中，回滚时利用 undo log 来进行回滚。每当进行一条记录进行操作(修改、删除、新增)时，要把回滚时需要的信息都记录到 undo log 里：原理是执行一条相反的操作。undo log 有两个参数：roll_pointer 指针和一个 trx_id 事务 id，通过 trx_id 可以知道该记录是被哪个事务修改的；通过 roll_pointer 指针可以将这些 undo log 串成一个链表，形成版本链。

InnoDB 存储引擎也通过 ReadView + undo log 实现 MVCC(多版本并发控制)。

UNDO LOG 的作用

实现事务回滚，保障事务的原子性：如果出现了错误或者用户执行了 ROLLBACK 语句，可以利用 undo log 中的历史数据将数据恢复到事务开始之前的状态

实现 MVCC 关键因素之一：MVCC 是通过 ReadView + undo log 实现的。undo log 为每条记录保存多份历史数据，在执行快照读的时候，会根据事务的 Read View 里的信息，顺着 undo log 的版本链找到满足其可见性的记录

REDO LOG(重做日志)

redo log 是物理日志，记录了某个数据页做了什么修改，每当执行一个事务就会产生一条或者多条物理日志。在事务提交时，先将 redo log 持久化到硬盘即可，不需要等到将缓存在 Buffer Pool 里的脏页数据持久化到硬盘。当系统崩溃时，虽然脏页数据没有持久化但是 redo log 已经持久化，可以根据 redo log 的内容，将所有数据恢复到最新的状态。

redo log 实现了事务中的持久性，主要用于掉电等故障恢复。发生更新的时候，InnoDB 会先更新内存，同时标记为脏页，然后将本次对这个页的修改以 redo log 的形式记录下来。InnoDB 引擎会在适当的时候，由后台线程将缓存在 Buffer Pool 的脏页刷新到硬盘里，实现 WAL 技术。

什么是 WAL 技术？

WAL 技术指的是 MySQL 的写操作并不是立刻写到硬盘上，而是先写日志，然后在合适的时间再写到硬盘上

什么是 crash-safe?

redo log + WAL 技术，InnoDB 就可以保证即使数据库发生异常重启，之前已提交的记录都不会丢失

BINLOG(归档日志)

Server 层生成的日志，主要用于数据备份和主从复制。

在完成一条更新操作后，Server 层会生成一条 binlog，等之后事务提交的时候，会将该事物执行过程中产生的所有 binlog 统一写入 binlog 文件。binlog 文件是记录了所有数据库表结构变更和表数据修改的日志，不会记录查询类的操作。

redo log 与 bin log 的区别?

- **适用对象不同**：binlog 是 MySQL 的 Server 层实现的，所有存储引擎都可以使用；redo log 是 InnoDB 存储引擎实现的日志
- **文件格式不同**：redo log 是物理日志，记录的是在某个数据页做了什么修改，比如对 XXX 表空间中的 YYY 数据页 ZZZ 偏移量的地方做了 AAA 更新
- **写入方式不同**：binlog 是追加写，写满一个文件，就创建一个新的文件继续写，不会覆盖以前的日志，保存的是全量的日志。redo log 是循环写，日志空间大小是固定，全部写满就从头开始，保存未被刷入硬盘的脏页日志
- **用途不同**：binlog 用于备份恢复、主从复制；redo log 用于掉电等故障恢复

redo log 和 undo log 区别?

redo log 记录了此次事务完成后的数据状态，undo log 记录了此次事务开始前的数据状态。

undo log 是如何实现 MVCC 的?

对于读提交和可重复读隔离级别，快照读是通过 Read View + undo log 来实现的，区别在于创建 Read View 的时机不同：

- 读提交在每个 select 都会生成一个新的 Read View，事务期间的多次读取同一条数据，前后两次读的数据可能会出现不一致，因为可能这期间另外一个事务修改了该记录，并提交了事务
- 可重复读隔离级别是启动事务时生成一个 Read View，然后整个事务期间都在用这个 Read View，这样就保证了在事务期间读到的数据都是事务启动前的记录

通过事务的 **Read View** 里的字段和记录中的两个隐藏列(**trx_id** 和 **roll_pointer**) 的比对, 如果不满足可见行, 就会顺着 undo log 版本链里找到满足其可见性的记录, 从而控制并发事务访问同一个记录时的行为。

为什么有了 binlog, 还要有 redo log?

早期版本 MySQL 里没有 InnoDB 引擎, MySQL 自带的 MyISAM 引擎没有 crash-safe 的能力, binlog 日志只能用于归档。InnoDB 是另一个公司以插件形式引入 MySQL 的, 所以 InnoDB 使用 redo log 来实现 crash-safe 能力。

被修改 Undo 页面, 需要记录对应 redo log 吗?

需要。开启事务后, InnoDB 更新记录前, 首先要记录相应的 undo log, 如果是更新操作, 也就是要生成一条 undo log, undo log 会写入 Buffer Pool 中的 Undo 页面。在内存修改该 Undo 页面后, 需要记录对应的 redo log。

binlog 的三种格式?

STATEMENT(默认格式)、ROW、MIXED:

- **STATEMENT**: 每一条修改数据的 SQL 都会被记录到 binlog 中, 主从复制中 slave 端再根据 SQL 语句重现
- **ROW**: 记录行数据最终被修改成什么样了, 不会出现 STATEMENT 下动态函数的问题
- **MIXED**: 包含了 STATEMENT 和 ROW 模式, 它会根据不同的情况自动使用 ROW 模式和 STATEMENT 模式

redo log 容灾恢复过程?

- 如果 redo log 是完整(commit 状态)的, 直接用 redo log 恢复
- 如果 redo log 是预提交 prepare 但不是 commit 状态, 此时要去判断 binlog 是否完整, 如果完整那就提交 redo log, 再用 redo log 恢复, 不完整就回滚事务

redo log 是直接写入硬盘的吗?

不是。直接写入硬盘会产生大量的 I/O 操作, redo log 会写入 redo log buffer, 每当产生一条 redo log 时, 会先写入到 redo log buffer, 后续在持久化到硬盘。

redo log 比直接落盘的优点？

redo log 的写方式使用了追加，日志操作是顺序写，硬盘操作是随机写，MySQL 的写操作从硬盘的**随机写变成了顺序写**，提升语句的执行性能。

- **实现事务的持久性**，让 MySQL 有 crash-safe 的能力，能够保证 MySQL 在任何时间段突然崩溃，重启后之前已提交的记录都不会丢失
- **将写操作从随机写变成了顺序写**，提升 MySQL 写入硬盘的性能

redo log buffer 什么时候刷盘？

- MySQL 正常关闭时，会触发落盘
- 当 redo log buffer 中记录的写入量大于 redo log buffer 内存空间的一半时，会触发落盘
- InnoDB 的后台线程每隔 1 秒，将 redo log buffer 持久化到硬盘
- 每次事务提交时都将缓存在 redo log buffer 里的 redo log 直接持久化到硬盘

redo log 文件文件格式和写入过程？

InnoDB 有 1 个 redo log 组，由有 2 个 redo log 文件组成：logfile0 和 logfile1。

redo log 组中每个 redo log 的大小是固定且相同的，redo log 组是以循环写的方式工作的，从头开始写，写到末尾就又回到开头，相当于一个环形。

先写 logfile0 文件，当 logfile0 文件被写满的时候，会切换至 logfile1 文件，当 logfile1 文件也被写满时，会切换回 ib_logfile0 文件。随着系统运行，Buffer Pool 的脏页刷新到了硬盘中，redo log 对应的记录没用了，会腾出空间记录新的更新操作。redo log 是循环写的方式相当于一个环形，用 write pos 表示 redo log 当前记录写到的位置，用 checkpoint 表示当前要擦除的位置。

write pos 追上了 checkpoint，说明 redo log 文件满了，MySQL 会被阻塞，会停下来将 Buffer Pool 中的脏页刷新到硬盘中，然后标记 redo log 哪些记录可以被擦除，接着对旧的 redo log 记录进行擦除，等擦除完旧记录腾出了空间，checkpoint 就会往后移动，然后 MySQL 恢复正常运行，继续执行新的更新操作。

binlog 什么时候刷盘？

事务执行过程中，先把日志写到 binlog cache(Server 层的 cache)，事务提交的时候，再把 binlog cache 写到 binlog 文件中。

binlog 什么时候刷盘频率？

MySQL 提供一个 `sync_binlog` 参数来控制数据库的 binlog 刷到硬盘上的频率：

- **`sync_binlog = 0` 时**：表示每次提交事务都只 write，不 fsync，后续交由操作系统决定何时将数据持久化到硬盘
- **`sync_binlog = 1` 时**：表示每次提交事务都会 write，然后马上执行 fsync
- **`sync_binlog = N(N > 1)` 时**：表示每次提交事务都 write，但累积 N 个事务后才 fsync

系统默认的设置是 `sync_binlog = 0`，也就是不做任何强制性的硬盘刷新指令，这时候的性能是最好的，但是风险也是最大的，因为一旦主机发生异常重启，还没持久化到硬盘的数据就会丢失。

当 `sync_binlog` 设置为 1 的时候，是最安全但是性能损耗最大的设置。因为当设置为 1 的时候，即使主机发生异常重启，最多丢失一个事务的 binlog，而已经持久化到硬盘的数据就不会有影响，不过就是对写入性能影响太大。

如果能容少量事务的 binlog 日志丢失的风险，为了提高写入的性能，一般会 `sync_binlog` 设置为 100~1000 中的某个数值。

主从复制是怎么实现？

binlog 记录 MySQL 上的所有变化并以二进制形式保存在硬盘上。复制的过程就是将 binlog 中的数据从主库传输到从库上。

- 主库在收到提交事务的请求之后会先写入 binlog，再提交事务，更新存储引擎中的数据，事务提交完成后，返回“操作成功”的响应
- 从库会创建一个专门的 I/O 线程，连接主库的 log dump 线程，来接收主库的 binlog 日志，再把 binlog 信息写入 relay log 的中继日志里，再返回给主库“复制成功”的响应
- 从库会创建一个用于回放 binlog 的线程，去读 relay log 中继日志，然后回放 binlog 更新存储引擎中的数据，最终实现主从的数据一致性

MySQL 主从复制模型？

- **同步复制**：MySQL 主库提交事务的线程要等待所有从库的复制成功响应，才返回客户端结果。这种方式在实际项目中，基本上没法用，原因有两个：一是性能很差，因为要复制到所有节点才返回响应；二是可用性也很差，主库和所有从库任何一个数据库出问题，都会影响业务。
- **异步复制(默认)**：MySQL 主库提交事务的线程并不会等待 binlog 同步到各从库，就返回客户端结果。这种模式一旦主库宕机，数据就会发生丢失。

- **半同步复制**：介于两者之间，事务线程不用等待所有的从库复制成功响应，只要一部分复制成功响应回来就行，比如一主二从的集群，只要数据成功复制到任意一个从库上，主库的事务线程就可以返回给客户端。这种半同步复制的方式，兼顾了异步复制和同步复制的优点，即使出现主库宕机，至少还有一个从库有最新的数据，不存在数据丢失的风险

为什么需要两阶段提交？

事务提交后，redo log 和 binlog 都要持久化到硬盘，但是这两个是独立的逻辑，可能出现半成功的状态，造成两份日志之间的逻辑不一致。

- **如果在将 redo log 刷入到硬盘之后，MySQL 突然宕机了，而 binlog 还没有来得及写入**：MySQL 重启后，通过 redo log 能将 Buffer Pool 恢复到新值，但是 binlog 里面没有记录这条更新语句，在主从架构中，binlog 会被复制到从库，由于 binlog 丢失了这条更新语句，从库的这一行是旧值，主从不一致。
- **如果在将 binlog 刷入到硬盘之后，MySQL 突然宕机了，而 redo log 还没有来得及写入**：由于 redo log 还没写，崩溃恢复以后这个事务无效，数据是旧值，而 binlog 里面记录了这条更新语句，在主从架构中，binlog 会被复制到从库，从库执行了这条更新语句，这一行字段是新值，与主库的值不一致性。

所以会造成主从环境的数据不一致性。因为 redo log 影响主库的数据，binlog 影响从库的数据，redo log 和 binlog 必须保持一致。

两阶段提交把单个事务的提交拆分成了 2 个阶段，分别是准备(Prepare)阶段和提交(Commit)阶段。每个阶段都由协调者(Coordinator)和参与者(Participant)共同完成。

两阶段提交的过程是怎样的？

在 MySQL 的 InnoDB 存储引擎中，开启 binlog 的情况下，MySQL 会同时维护 binlog 日志与 InnoDB 的 redo log，为了保证这两个日志的一致性，MySQL 使用了内部 XA 事务，内部 XA 事务由 binlog 作为协调者，存储引擎是参与者。

当客户端执行 commit 语句或者在自动提交的情况下，MySQL 内部开启一个 XA 事务，分两阶段来完成 XA 事务的提交。

事务的提交过程有两个阶段，将 redo log 的写入拆成了两个步骤：prepare 和 commit，中间再穿插写入 binlog：

- **prepare 阶段**：将内部 XA 事务的 ID 写入到 redo log，同时将 redo log 对应的事务状态设置为 prepare，然后将 redo log 持久化到硬盘。

- **commit 阶段**：把 内部 XA 事务的 ID 写入到 binlog，然后将 binlog 持久化到硬盘，接着调用引擎的提交事务接口，将 redo log 状态设置为 commit，此时该状态并不需要持久化到硬盘，只需要 write 到文件系统的 page cache 成功，只要 binlog 写硬盘成功，redo log 的状态还是 prepare 没有关系，一样会被认为事务已经执行成功

异常重启会出现什么现象？

在 MySQL 重启后会按顺序扫描 redo log 文件，碰到处于 prepare 状态的 redo log，就拿着 redo log 中的 XID 去 binlog 查看是否存在此 XID：

- 如果 binlog 中没有当前内部 XA 事务的 XID，说明 redolog 完成刷盘，但是 binlog 还没有刷盘，则回滚事务。对应时刻 A 崩溃恢复的情况
- 如果 binlog 中有当前内部 XA 事务的 XID，说明 redolog 和 binlog 都已经完成了刷盘，则提交事务。对应时刻 B 崩溃恢复的情况

对于处于 prepare 阶段的 redo log，即可以提交事务，也可以回滚事务，这取决于是否能在 binlog 中找到与 redo log 相同的 XID，如果有就提交事务，如果没有就回滚事务。这样就可以保证 redo log 和 binlog 这两份日志的一致性了。

事务没提交 redo log 会被持久化到硬盘吗？

会。事务执行中间过程的 redo log 也是直接写在 redo log buffer 中的，这些缓存在 redo log buffer 里的 redo log 也会被后台线程每隔一秒一起持久化到硬盘。

两阶段提交有什么问题？

- **硬盘 I/O 次数高**：每个事务提交都会进行两次 fsync(刷盘)，一次是 redo log 刷盘，另一次是 binlog 刷盘
- **锁竞争激烈**：两阶段提交虽然能够保证单事务两个日志的内容一致，但在多事务的情况下，却不能保证两者的提交顺序一致。在两阶段提交的流程基础上，还需要加一个锁来保证提交的原子性，从而保证多事务的情况下，两个日志的提交顺序一致

组提交是什么意思？

有多个事务提交的时候，会将多个 binlog 刷盘操作合并成一个，从而减少硬盘 I/O 的次数。组提交机制后，prepare 阶段不变，**将 commit 阶段拆分为三个过程**：

- **flush 阶段**：多个事务按进入的顺序将 binlog 从 cache 写入文件(不刷盘)；

- **sync 阶段**：对 binlog 文件做 fsync 操作(多个事务的 binlog 合并一次刷盘)；
- **commit 阶段**：各个事务按顺序做 InnoDB commit 操作；

上面的每个阶段都有一个队列，每个阶段有锁进行保护，因此保证了事务写入的顺序，第一个进入队列的事务会成为 leader，leader 领导所在队列的所有事务，全权负责整队的操作，完成后通知队内其他事务操作结束。对每个阶段引入了队列后，锁就只针对每个队列进行保护，不再锁住提交事务的整个过程，锁粒度减小了，这样就使得多个阶段可以并发执行，从而提升效率

Buffer Pool 有什么作用？

主要的作用是实现缓存：

- **当读取数据时**：如果数据存在于 Buffer Pool 中，会直接读取 Buffer Pool 中的数据
- **当修改数据时**：如果数据存在于 Buffer Pool 中，那直接修改 Buffer Pool 中数据所在的页，然后将其页设置为脏页(该页的内存数据和硬盘上的数据已经不一致)；不会立即将脏页写入硬盘，后续由后台线程选择一个合适的时机将脏页写入到硬盘

Buffer Pool 缓存内容？

InnoDB 会为 Buffer Pool 申请一片连续的内存空间，然后按照默认的大小划分出一个个的页，Buffer Pool 中的页就叫做缓存页。此时这些缓存页都是空闲的，之后随着程序的运行，才会有硬盘上的页被缓存到 Buffer Pool 中。

Buffer Pool 除了缓存索引页和数据页，还包括了 Undo 页，插入缓存、自适应哈希索引、锁信息等等。

开启事务后，InnoDB 层更新记录前，首先要记录相应的 undo log，undo log 会写入 Buffer Pool 中的 Undo 页面。

优化

慢查询的原因？

- **索引不足**：如果查询的表没有合适的索引，MySQL 需要遍历整个表才能找到匹配的记录，这会导致查询变慢。可以通过添加索引来优化查询性能
- **数据库设计问题**：如果数据库设计不合理，例如表过于庞大、列过多等，查询时可能需要耗费大量时间。这时可以通过优化数据库设计来解决问题

- **数据库服务器负载过高**：如果 MySQL 服务器上同时运行了太多的查询，会导致服务器负载过高，从而导致查询变慢。可以通过增加服务器硬件配置或分散查询负载来解决问题
- **查询语句复杂**：复杂的查询语句可能需要耗费更多的时间才能完成。可以尝试简化查询语句或将查询分解成多个较简单的查询语句来提高性能
- **数据库统计信息不准确**：如果数据库统计信息不准确，MySQL 可能会选择不合适的查询计划，从而导致查询变慢。可以通过更新数据库统计信息来解决问题
- **MySQL 版本过低**：较老版本的 MySQL 可能性能较差，升级到较新版本的 MySQL 可能会提高查询性能

MySQL 硬盘 I/O 很高有什么优化的方

- **设置组提交的两个参数**：binlog_group_commit_sync_delay 和 binlog_group_commit_sync_no_delay_count 参数，延迟 binlog 刷盘的时机，从而减少 binlog 的刷盘次数
- **将 sync_binlog 设置为大于 1 的值(比较常见是 100~1000)**：表示每次提交事务都 write，但累积 N 个事务后才 fsync，相当于延迟了 binlog 刷盘的时机。但是这样做的风险是，主机掉电时会丢 N 个事务的 binlog 日志
- **将 innodb_flush_log_at_trx_commit 设置为 2**：表示每次事务提交时，都只是缓存在 redo log buffer 里的 redo log 写到 redo log 文件，注意写入到 redo log 文件并不意味着写入到了硬盘，因为操作系统的文件系统中有个 Page Cache，专门用来缓存文件数据的，所以写入 redo log 文件意味着写入到了操作系统的文件缓存，然后交由操作系统控制持久化到硬盘的时机。但是这样做的风险是，主机掉电的时候会丢数据