一条SQL查询语句是如何执行的?

- 连接器：连接器负责和客户端建立连接，获取用户权限，维持和管理连接
- 查询缓存：MySQL拿到一条查询语句后，会先到缓存中查询之前是不是执行过此语句，之前执行的语句及其结果可能会以Key-value对的形式存储在缓存中，如果存在，直接返回结果。
- 分析器：输入的SQL语句是由字符串和空格构成的，MySQL需要识别出这些字符串分别是什么，代表什么。
- 优化器：当一个表存在多个索引时，优化器会决定使用哪个索引。或者一个语句中由多个表相连时，决定各个表的连接顺序。
- 执行器：通过分析和优化，知道了要做什么，并且还知道可该怎么做。接着就是进行语句的执行。

事务隔离级别有哪些?

- 读未提交：一个事务还没提交时，它的变更就能被其他事务看到。(会发生脏读、不可重复读、幻读)
- 读提交：一个事务提交后，他的变更才能被其他事务看到。(会发生不可重复读、幻读)
- 可重复读：一个事务在执行过程中看到的数据，总是跟在执行过程中看到的数据是一致的。(会发生幻读)
- 串行化：写会加写锁，读会加读锁。当出现读写锁冲突时，后访问的事务必须等前一个事务执行完成，才能继续执行。

按隔离水平高低排序：

串行化 > 可重复读 > 读已提交 > 读未提交

这四种隔离级别是如何实现的呢?

- 对于「读未提交」隔离级别的事务来说，因为可以读到未提交事务修改的数据，所以直接读取最新的数据就好了；
- 对于「串行化」隔离级别的事务来说，通过加读写锁的方式来避免并行访问；
- 对于「读提交」和「可重复读」隔离级别的事务来说，它们是通过 Read View 来实现的，它们的区别在于创建 Read View 的时机不同。可以把 Read View 理解成一个数据快照，就像相机拍照那样，定格某一时刻的风景。「读提交」隔离级别是在「每个语句执行前」都会重新生成一个 Read View，而「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。

事务的四大特性是什么?

- 原子性：事务时不可分割的最小工作单元，一个事务对应于一个完整的业务，一个事务的操作要么全部完成，要么全部没完成，当操作过程中出现问题，会回滚到原始状态。
- 一致性：事务执行之前和执行之后都必须处于一致性状态。
- 隔离性：允许多个并发的事务同时对数据进行修改和读取，执行互不干扰，防止多个事务并发执行由于交叉执行造成数据不一致的情况。
- 持久性：一个事务一旦被提交，那么对数据库中的数据的改变就是永久性的，即便在数据库系统中遇到故障的情况下也不会丢失提交事务的操作。

InnoDB引擎通过什么技术保证事务的四大特性呢?

- 持久性是通过 redo log （重做日志）来保证的；
- 原子性是通过 undo log（回滚日志） 来保证的；

- 隔离性是通过 MVCC（多版本并发控制） 或锁机制来保证的；
- 一致性则是通过持久性+原子性+隔离性来保证；

并行事务会引发什么问题?

在同时处理多个事务时，就可能出现脏读、不可重复读以及幻读。

- 脏读：一个事务「读到」了另一个事务「未提交事务修改过的数据」。
- 不可重复读：在一个事务内多次读取同一个数据，如果出现前后两次读到的数据不一样的情况，就意味着发生了「不可重复读」现象。
- 幻读：在一个事务内多次查询某个符合查询条件的「记录数量」，如果出现前后两次查询到的记录数量不一样的情况，就意味着发生了「幻读」现象。

按严重性排序：

脏读 > 不可重复读 > 幻读

MySQL可重复读隔离级别，完全解决幻读了吗?

- 针对**快照读**（普通 select 语句），是**通过 MVCC 方式解决了幻读**，因为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，是查询不出来这条数据的，所以就很好了避免幻读问题。
- 针对**当前读**，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，所以就很好了避免幻读问题。

请举例可重复读隔离级别下，发生幻读的场景

- 对于快照读，MVCC 并不能完全避免幻读现象。因为当事务 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。

可重复读隔离级别下虽然很大程度上避免了幻读，但是还是没有能完全解决幻读。

如何避免可重复读隔离级别下发生幻读的场景呢?

- 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。

索引有哪些种类?

- 从数据结构维度分类：
  - B+树索引：所有子节点存储数据，适合范围查询。
  - 哈希索引：适合等值查询。
  - 全文索引：MyISAM和InnoDB中支持使用全文索引。
  - R-Tree索引：用来对GIS数据类型创建SPATIAL索引。
- 从物理存储维度分类：
  - （聚集索引）聚簇索引：数据存储与索引一起存放，叶子节点会存储一整行记录，找到索引也就找到了数据。
  - （非聚集）二级索引：数据存储与索引分开存放，叶子节点不存储数据，存储的是数据地址。
- 从逻辑维度分类

- 主键索引：一种特殊的唯一索引，不允许由空值。
- 唯一索引：索引列中的值必须唯一，值允许为空。
- 普通索引：MySQL中的基本索引类型，允许空值和重复值。
- 联合索引：多个字段创建的索引，使用时遵循最左前缀原则。
- 空间索引：MySQL5.7之后支持空间索引，在空间索引这方面遵循OpenGIS几何数据模型规则。

MySQL为什么使用B+树来作索引呢?

- 查询底层节点：B+树的非叶子节点不存放实际的记录数据，进存放索引，因此数据量相同的情况下，相比既存储索引又存储记录的B树，B+树可以有更大的空间来存放更多的索引，所以查询底层节点的磁盘I/O次数会更少。
- 插入和删除：B+树有大量的冗余节点，删除一个节点时，可以直接从叶子节点中删除，甚至不用移动非叶子节点，删除非常块。插入也是一样，虽然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，删除的时候非常复杂，可能设计复杂的树的变形。
- 范围查询：B+ 树叶子节点之间用链表连接了起来，有利于范围查询，而 B 树要实现范围查询，因此只能通过树的遍历来完成范围查询，这会涉及多个节点的磁盘 I/O 操作，范围查询效率不如 B+ 树。

B+树和B树的差异有哪些?

- B+树的叶子节点存放实际数据（索引+记录），非叶子节点只会存放索引。
- 所有的索引都会在叶子节点出现，叶子节点之间构成一个有序链表。
- 非叶子节点的索引也会同时存在在子节点中，并且是在子节点中所有索引的最大（或最小）。
- 非叶子节点中有多少个子节点，就有多少个索引;

什么时候需要建立索引?

- 表的主关键字：自动建立唯一索引。
- 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，否则用的是文件排序。
- 唯一性约束列： 如果某列具有唯一性约束，那么为了确保数据的唯一性，可以在这些列上创建唯一索引。
- 大表中的关键列： 在大表中，如果查询的效率变得很低，可以考虑在关键列上创建索引。

什么时候不需要创建索引?

- 小表：对小表创建索引可能会带来额外的开销，因为在小数据集中扫描整个表可能比使用索引更快。
- 频繁的插入、更新和删除：索引的维护成本会随着数据的插入、更新和删除操作而增加。如果表经常被修改，过多的索引可能会影响性能。
- 数据重复且均匀分布：这种字段建索引一般不会提高数据库的查询速度。
- 很少被查询的列：如果某列很少被用于查询条件，那么为它创建索引可能没有明显的性能提升。
- 查询结果总行数很少的列：如果查询的结果集总行数很少，使用索引可能不会有太大的性能提

升。

索引失效的场景有哪些?

- 当我们使用左或者左右模糊匹配的时候，也就是`like %xx`或者`like %xx%`这两种方式都会造成索引失效;
- 表连接中的列类型不匹配： 如果在连接操作中涉及的两个表的列类型不匹配，索引可能会失效。例如，一个表的列是整数，另一个表的列是字符，连接时可能会导致索引失效。
- 当我们在查询条件中对索引列做了计算、函数、类型转换操作，这些情况下都会造成索引失效;
- 不等号条件： 当查询中包含不等号条件（如>,<,>=,<=）时，索引可能会失效。通常情况下，索引只能用于等值比较。
- OR 条件： 当查询中使用多个 OR 条件时，如果这些条件不涉及同一列，索引可能无法有效使用。数据库可能会选择全表扫描而不是使用多个索引。

优化索引的方法有哪些?

- 前缀索引优化
- 覆盖索引优化
- 主键索引最好是自增的
- 防止索引失效

MySQL的执行引擎有哪些?

- 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。

MySQL单表不要超过2000w行，靠谱吗?

2000w只是推荐值，超过这个值可能会导致B+树层级更高，影响查询性能。

count(*)和count(1)有什么区别？哪个性能更好?

count(*)其实等于 count(0)，也就是说，当你使用 count(*) 时，MySQL 会将 * 参数转化为参数 0 来处理。

性能排序

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

count(1)、 count(*)、 count(主键字段)在执行的时候，如果表里存在二级索引，优化器就会选择二级索引进行扫描。如果要执行 count(1)、 count(*)、 count(主键字段) 时，尽量在数据表上建立二级索引，这样优化器会自动采用 key_len 最小的二级索引进行扫描，相比于扫描主键索引效率会高一些。不要使用 count(字段) 来统计记录个数，因为它的效率是最差的，会采用全表扫描的方式来统计。如果你非要统计表中该字段不为 NULL 的记录个数，建议给这个字段建立一个二级索引。

如何优化count(*)?

- 近似值：如果业务对于统计个数不需要很精确，可以使用show table status或者explain命令来进行估算。

- 额外表保存计数值：想精确的获取表的记录总数，我们可以将这个计数值保存到单独的一张计数表中。当我们在数据表插入一条记录的同时，将计数表中的计数字段 + 1。也就是说，在新增和删除操作时，我们需要额外维护这个计数表。

MySQL使用 like "%x"，索引一定会失效吗?

不一定

因为如果这张表的字段都是索引字段，那么查询的时候就会使用覆盖索引，**查询的数据都在二级索引的 B+ 树，因为二级索引的 B+ 树的叶子节点包含「索引值+主键值」，所以查二级索引的 B+ 树就能查到全部结果了**

但是如果查询字段中含有非索引字段，那么索引就会失效了，要查询的数据就不能只在二级索引树里找了，得需要回表操作才能完成查询的工作，再加上是左模糊匹配，无法利用索引树的有序性来快速定位数据，所以得在二级索引树逐一遍历，获取主键值后，再到聚簇索引树检索到对应的数据行，优化器认为上面这样的查询过程的成本实在太高了，所以直接选择全表扫描的方式来查询数据。

MySQL有哪些锁呢?

- 全局锁：主要应用于做**全库逻辑备份**，这样在备份数据库期间，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。
  - 全局锁有什么缺点呢?
    - 加上全局锁，意味着整个数据库都是只读状态。那么如果数据库里有很多数据，备份就会花费很多的时间，关键是备份期间，业务只能读数据，而不能更新数据，这样会造成业务停滞。
  - 使用全局锁会影响业务，那有什么其他方式可以避免?
    - 如果数据库的引擎支持的事务支持**可重复读的隔离级别**，那么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。
- 表级锁
  - 表锁：表锁除了会限制别的线程的读写外，也会限制本线程接下来的读写操作。
  - 元数据锁：对数据库表进行操作时，会自动给这个表加上元数据锁，为了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：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」；对某些纪录加上「独占锁」之前，需要先在表级别加上一个「意向独占锁」；当执行插入、更新、删除操作，需要先对表加上「意向独占锁」，然后对该记录加独占锁。

    而普通的 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。
    - **意向锁的目的是为了快速判断表里是否有记录被加锁。**
  - AUTO-INC锁：表里的主键通常都会设置成自增的，之后可以在插入数据时，可以不指定主键的值，数据库会自动给主键赋值递增的值通过 AUTO-INC 锁实现的。在插入数据时，会加一个表级别的 AUTO-INC 锁，然后为被 AUTO_INCREMENT 修饰的字段赋值递增的值，等插入语句执行完成后，才会把 AUTO-INC 锁释放掉。其他事务的如果要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。
- 行级锁
  - Record Lock：记录锁，也就是仅仅把一条记录锁上；记录锁分为排他锁和共享锁。
    - 只有S和S兼容
    - 共享锁（S锁）满足读读共享，读写互斥。

        - 独占锁（X锁）满足写写互斥、读写互斥。
    - Gap Lock：间隙锁，锁定一个范围，但是不包含记录本身；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，即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**
    - Next-Key Lock：Record Lock + Gap Lock 的组合，锁定一个范围，并且锁定记录本身。next-key lock 即能保护该记录，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是如何加锁的?

update没加索引会锁全表?

是的，会引起锁全表！**在 update 语句的 where 条件没有使用索引，就会全表扫描，于是就会对所有记录加上 next-key 锁（记录锁 + 间隙锁），相当于把整个表锁住了。**

- 执行 update 语句的时候，确保 where 条件中带上了索引列，并且在测试机确认该语句是否走的是索引扫描，防止因为扫描全表，而对表中的所有记录加上锁。
- 打开 MySQL sql_safe_updates 参数，这样可以预防 update 操作时 where 条件没有带上索引列。
- 如果发现即使在 where 条件中带上了列索引列，优化器走的还是全标扫描，这时我们就要使用 `force index([index_name])` 可以告诉优化器使用哪个索引。

MySQL记录锁+间隙锁可以防止删除操作而导致幻读吗?

可以防止此类事情发生！

- 在 MySQL 的可重复读隔离级别下，针对当前读的语句会对**索引**加记录锁+间隙锁，这样可以避免其他事务执行增、删、改时导致幻读的问题。
- 在执行 update、delete、select ... for update 等具有加锁性质的语句，一定要检查语句是否走了索引，如果是全表扫描的话，会对每一个索引加 next-key 锁，相当于把整个表锁住了，这是挺严重的问题。

MySQL死锁了，怎么办?

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。**

两种策略通过打破循环等待条件来解除死锁状态：

- 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。
- 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。

MySQL日志文件有哪几种？有什么用?

- undo log：回滚日志，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。**

    - 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，MySQL 在执行快照读（普通 select 语句）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，是 Innodb 存储引擎层生成的日志，实现了事务中的**持久性**，主要**用于掉电等故障恢复。**
- binlog：归档日志，是 Server 层生成的日志，主要**用于数据备份和主从复制。**