简述数据库三大范式

第一范式是最基本的范式。如果数据库表中的所有字段值都是不可分解的原子值，就说明该数据库表满足了第一范式。

数据库第二范式：关系模式必须满足第一范式，并且所有非主属性都完全依赖于主码。注意，符合第二范式的关系模型可能还存在数据冗余、更新异常等问题。关系模型（学号，姓名，专业编号，专业名称）中，学号->姓名，而专业编号->专业名称，不满足数据库第二范式

数据库第三范式：关系模型满足第二范式，所有非主属性对任何候选关键字都不存在传递依赖。即每个属性都跟主键有直接关系而不是间接关系。接着以学生表举例，对于关系模型（学号，姓名，年龄，性别，所在院校，院校地址，院校电话）院校地址，院校电话和学号不存在直接关系，因此不满足第三范式。

简述MySQL的架构

MySQL可以分为应用层,逻辑层,数据库引擎层,物理层。

应用层：负责和客户端，响应客户端请求，建立连接，返回数据。

逻辑层：包括SQK接口，解析器，优化器，Cache与buffer。

数据库引擎层：有常见的MyISAM,InnoDB等等。

物理层：负责文件存储，日志等等。

简述执行SQL语言的过程

1. 客户端首先通过连接器进行身份认证和权限相关
2. 如果是执行查询语句的时候，会先查询缓存，但MySQL 8.0 版本后该步骤移除。
3. 没有命中缓存的话，SQL 语句就会经过解析器，分析语句，包括语法检查等等。
4. 通过优化器，将用户的SQL语句按照 MySQL 认为最优的方案去执行。
5. 执行语句，并从存储引擎返回数据。

简述MySQL的共享锁排它锁

共享锁也称为读锁，相互不阻塞，多个客户在同一时刻可以同时读取同一个资源而不相互干扰。排他锁也称为写锁，会阻塞其他的写锁和读锁，确保在给定时间内只有一个用户能执行写入并防止其他用户读

取正在写入的同一资源。

简述MySQL中的按粒度的锁分类

表级锁: 对当前操作的整张表加锁,实现简单，加锁快，但并发能力低。

行锁: 锁住某一行，如果表存在索引，那么记录锁是锁在索引上的，如果表没有索引，那么 InnoDB 会创建一个隐藏的聚簇索引加锁。行级锁能大大减少数据库操作的冲突。其加锁粒度最小，并发度高，但加锁的开销也最大，加锁慢，会出现死锁。

Gap 锁：也称为间隙锁: 锁定一个范围但不包括记录本身。其目的是为了防止同一事物的两次当前读出现幻读的情况。

Next-key Lock： 行锁+gap锁。

如何解决数据库死锁

1. 预先检测到死锁的循环依赖，并立即返回一个错误。
2. 当查询的时间达到锁等待超时的设定后放弃锁请求。

简述乐观锁和悲观锁

乐观锁：对于数据冲突保持一种乐观态度，操作数据时不会对操作的数据进行加锁，只有到数据提交的时候才通过一种机制来验证数据是否存在冲突。

悲观锁：对于数据冲突保持一种悲观态度，在修改数据之前把数据锁住，然后再对数据进行读写，在它释放锁之前任何人都不能对其数据进行操作，直到前面一个人把锁释放后下一个人数据加锁才可对数据进行加锁，然后才可以对数据进行操作，一般数据库本身锁的机制都是基于悲观锁的机制实现的。

简述InnoDB存储引擎

InnoDB 是 MySQL 的默认事务型引擎，支持事务，表是基于聚簇索引建立的。支持表级锁和行级锁，支持外键，适合数据增删改查都频繁的情况。

InnoDB 采用 MVCC 来支持高并发，并且实现了四个标准的隔离级别。其默认级别是 REPEATABLE READ，并通过间隙锁策略防止幻读，间隙锁使 InnoDB 不仅仅锁定查询涉及的行，还会对索引中的间隙进行锁定防止幻行的插入。

简述MyISAM存储引擎

MySQL5.1及之前，MyISAM 是默认存储引擎。MyISAM不支持事务，Myisam支持表级锁，不支持行级锁，表不支持外键，该存储引擎存有表的行数，count运算会更快。适合查询频繁，不适合对于增删改要求高的情况

简述Memory存储引擎

Memory存储引擎将所有数据都保存在内存，不需要磁盘 IO。支持哈希索引，因此查找速度极快。Memory 表使用表级锁，因此并发写入的性能较低。

索引是什么?

索引是存储引擎中用于快速找到记录的一种数据结构。在关系型数据库中，索引具体是一种对数据库中一列或多列的值进行排序的存储结构。

为什么引入索引?

为了提高数据查询的效率。索引对数据库查询良好的性能非常关键，当表中数据量越来越大，索引对性能的影响越重要。

Mysql有哪些常见索引类型?

- 数据结构角度
  > B-Tree索引
  > 哈希索引
  > R-Tree索引
  > 全文索引
- 物理存储角度
  > 主键索引（聚簇索引）：叶子节点存的是整行的数据
  > 非主键索引（二级索引）：叶子节点存的主键的值

简述B-Tree与B+树

B-Tree 是一种自平衡的多叉树。每个节点都存储关键字值。其左子节点的关键字值小于该节点关键字值，且右子节点的关键字值大于或等于该节点关键字值。

B+树也是是一种自平衡的多叉树。其基本定义与B树相同，不同点在于数据只出现在叶子节点，所有叶子节点增加了一个链指针，方便进行范围查询。

B+树中间节点不存放数据，所以同样大小的磁盘页上可以容纳更多节点元素，访问叶子节点上关联的数据也具有更好的缓存命中率。并且数据顺序排列并且相连，所以便于区间查找和搜索。

B树每一个节点都包含key和value，查询效率比B+树高。

简述Hash索引

哈希索引对于每一行数据计算一个哈希码，并将所有的哈希码存储在索引中，同时在哈希表中保存指向每个数据行的指针。只有 Memory 引擎显式支持哈希索引。

Hash索引不支持范围查询，无法用于排序，也不支持部分索引列匹配查找。

简述自适应Hash索引

InnoDB对于频繁使用的某些索引值，会在内存中基于 B-Tree 索引之上再创键一个哈希索引，这也被称为自适应Hash索引。

简述聚集索引和稀疏索引

聚集索引按每张表的主键构建一棵B+树，数据库中的每个搜索键值都有一个索引记录，每个数据页通过双向链表连接。表数据访问更快，但表更新代价高。

稀疏索引不会为每个搜索关键字创建索引记录。搜索过程需要，我们首先按索引记录进行操作，并按顺序搜索，直到找到所需的数据为止。

简述辅助索引与回表查询

辅助索引是非聚集索引，叶子节点不包含记录的全部数据，包含了一个书签用来告诉InnoDB哪里可以找到与索引相对应的行数据。

通过辅助索引查询，先通过书签查到聚集索引，再根据聚集索引查对应的值，需要两次，也称为回表查询。

简述联合索引和最左匹配原则

联合索引是指对表上的多个列的关键词进行索引。

对于联合索引的查询，如果精确匹配联合索引的左边连续一列或者多列，则mysql会一直向右匹配直到遇到范围查询（>,<,between,like）就停止匹配。Mysql会对第一个索引字段数据进行排序，在第一个字段基础上，再对第二个字段排序。

简述覆盖索引

覆盖索引指一个索引包含或覆盖了所有需要查询的字段的值，不需要回表查询，即索引本身存了对应的值。

为什么数据库不用红黑树用B+树

红黑树的出度为 2，而 B Tree 的出度一般都非常大。红黑树的树高 h 很明显比 B Tree 大非常多，IO次数很多，导致会比较慢，因此检索的次数也就更多。

B+Tree 相比于 B-Tree 更适合外存索引，拥有更大的出度，IO次数较少，检索效率会更高。

基于主键索引的查询和非主键索引的查询有什么区别?

对于select * from 主键=XX，基于主键的普通查询仅查找主键这棵树，对于select * from 非主键=XX，基于非主键的查询有可能存在回表过程（回到主键索引树搜索的过程称为回表），因为非主键索引叶子节点仅存主键值，无整行全部信息。

非主键索引的查询一定会回表吗?

不一定，当查询语句的要求字段全部命中索引，不用回表查询。如select 主键 from 非主键=XX，此时非主键索引叶子节点即可拿到主键信息，不用回表。

简述MySQL使用EXPLAIN 的关键字段

explain关键字用于分析sql语句的执行情况，可以通过他进行sql语句的性能分析。

type：表示连接类型，从好到差的类型排序为

- system：系统表，数据已经加载到内存里。
- const：常量连接，通过索引一次就找到。
- eq_ref：唯一性索引扫描，返回所有匹配某个单独值的行。

- ref：非主键非唯一索引等值扫描，const或eq_ref改为普通非唯一索引。
- range：范围扫描，在索引上扫码特定范围内的值。
- index：索引树扫描，扫描索引上的全部数据。
- all：全表扫描。

key：显示MySQL实际决定使用的键。

key_len：显示MySQL决定使用的键长度，长度越短越好

Extra：额外信息

- Using filesort：MySQL使用外部的索引排序，很慢需要优化。
- Using temporary：使用了临时表保存中间结果，很慢需要优化。
- Using index：使用了覆盖索引。
- Using where：使用了where。

简述MySQL优化流程

1. 通过慢日志定位执行较慢的SQL语句
2. 利用explain对这些关键字段进行分析
3. 根据分析结果进行优化

简述MySQL中的日志log

redo log: 存储引擎级别的log（InnoDB有，MyISAM没有），该log关注于事务的恢复.在重启mysql服务的时候，根据redo log进行重做，从而使事务有持久性。

undo log：是存储引擎级别的log（InnoDB有，MyISAM没有）保证数据的原子性，该log保存了事务发生之前的数据的一个版本，可以用于回滚，是MVCC的重要实现方法之一。

bin log：数据库级别的log，关注恢复数据库的数据。

简述事务

事务内的语句要么全部执行成功，要么全部执行失败。

事务满足如下几个特性：

- 原子性（Atomicity）：
  > 一个事务中的所有操作要么全部完成，要么全部不完成。

- 一致性（Consistency）：
  > 事务执行前后数据库的状态保存一致。
- 隔离性（Isolation）
  > 多个并发事务对数据库进行操作，事务间互不干扰。
- 持久性（Durability）
  > 事务执行完毕，对数据的修改是永久的，即使系统故障也不会丢失

数据库中多个事务同时进行可能会出现什么问题?

- 丢失修改
- 脏读：当前事务可以查看到别的事务未提交的数据。
- 不可重读：在同一事务中，使用相同的查询语句，同一数据资源莫名改变了。
- 幻读：在同一事务中，使用相同的查询语句，莫名多出了一些之前不存在的数据，或莫名少了一些原先存在的数据。

SQL的事务隔离级别有哪些?

- 读未提交：
  > 一个事务还没提交，它做的变更就能被别的事务看到。
- 读提交：
  > 一个事务提交后，它做的变更才能被别的事务看到。
- 可重复读：
  > 一个事务执行过程中看到的数据总是和事务启动时看到的数据是一致的。在这个级别下事务未提交，做出的变更其它事务也看不到。
- 串行化：
  > 对于同一行记录进行读写会分别加读写锁，当发生读写锁冲突，后面执行的事务需等前面执行的事务完成才能继续执行。

什么是MVCC?

MVCC为多版本并发控制，即同一条记录在系统中存在多个版本。其存在目的是在保证数据一致性的前提下提供一种高并发的访问性能。对数据读写在不加读写锁的情况下实现互不干扰,从而实现数据库的隔离性,在事务隔离级别为读提交和可重复读中使用到。

在InnoDB中，事务在开始前会向事务系统申请一个事务ID，该ID是按申请顺序严格递增的。每行数据具有多个版本，每次事务更新数据都会生成新的数据版本，而不会直接覆盖旧的数据版本。数据的行结构中包含多个信息字段。其中实现MVCC的主要涉及最近更改该行数据的事务ID（DB_TRX_ID）和可以找到历史数据版本的指针（DB_ROLL_PTR）。InnoDB在每个事务开启瞬间会为其构造一个记录当前已经开启但未提交的事务ID的视图数组。通过比较链表中的事务ID与该行数据的值与对应的DB_TRX_ID，并通过DB_ROLL_PTR找到历史数据的值以及对应的DB_TRX_ID来决定当前版本的数据是否应该被当前事务所见。最终实现在不加锁的情况下保证数据的一致性。

读提交和可重复读都基于MVCC实现，有什么区别?

在可重复读级别下，只会在事务开始前创建视图，事务中后续的查询共用一个视图。而读提交级别下每个语句执行前都会创建新的视图。因此对于可重复读，查询只能看到事务创建前就已经提交的数据。而对于读提交，查询能看到每个语句启动前已经提交的数据。

InnoDB如何保证事务的原子性、持久性和一致性?

利用undo log保障原子性。该log保存了事务发生之前的数据的一个版本，可以用于回滚，从而保证事务原子性。

利用redo log保证事务的持久性，该log关注于事务的恢复.在重启mysql服务的时候，根据redo log进行重做，从而使事务有持久性。

利用undo log+redo log保障一致性。事务中的执行需要redo log，如果执行失败，需要undo log 回滚。

MySQL是如何保证主备一致的?

MySQL通过binlog（二进制日志）实现主备一致。binlog记录了所有修改了数据库或可能修改数据库的语句，而不会记录select、show这种不会修改数据库的语句。在备份的过程中，主库A会有一个专门的线程将主库A的binlog发送给
备库B进行备份。其中binlog有三种记录格式：

1. statement:记录对数据库进行修改的语句本身，有可能会记录一些额外的相关信息。优点是binlog日志量少，IO压力小，性能较高。缺点是由于记录的信息相对较少，在不同库执行时由于上下文的环境不同可能导致主备不一致。
2. row:记录对数据库做出修改的语句所影响到的数据行以及对这些行的修改。比如当修改涉及多行数据，会把涉及的每行数据都记录到binlog。优点是能够完全的还原或者复制日志被记录时的操作。缺点是日志量占用空间较大，IO压力大，性能消耗较大。
3. mixed:混合使用上述两种模式，一般的语句使用statment方式进行保存，如果遇到一些特殊的函数，则使用row模式进行记录。MySQL自己会判断这条SQL语句是否可能引起主备不一致，如果有

可能，就用row格式，
否则就用statement格式。但是在生产环境中，一般会使用row模式。

redo log与binlog的区别?

1. redo log是InnoDB引擎特有的，只记录该引擎中表的修改记录。binlog是MySQL的Server层实现的，会记录所有引擎对数据库的修改。
2. redo log是物理日志，记录的是在具体某个数据页上做了什么修改；binlog是逻辑日志，记录的是这个语句的原始逻辑。
3. redo log是循环写的，空间固定会用完；binlog是可以追加写入的，binlog文件写到一定大小后会切换到下一个，并不会覆盖以前的日志。

crash-safe能力是什么?

InnoDB通过redo log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。

WAL技术是什么?

WAL的全称是Write-Ahead Logging，它的关键点就是先写日志，再写磁盘。事务在提交写入磁盘前，会先写到redo log里面去。如果直接写入磁盘涉及磁盘的随机I/O访问，涉及磁盘随机I/O访问是非常消耗时间的一个过程，相比之下先写入redo log，后面再找合适的时机批量刷盘能提升性能。

两阶段提交是什么?

为了保证binlog和redo log两份日志的逻辑一致，最终保证恢复到主备数据库的数据是一致的，采用两阶段提交的机制。

1. 执行器调用存储引擎接口，存储引擎将修改更新到内存中后，将修改操作记录redo log中，此时redo log处于prepare状态。
2. 存储引擎告知执行器执行完毕，执行器生成这个操作对应的binlog，并把binlog写入磁盘。
3. 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。

只靠binlog可以支持数据库崩溃恢复吗?

不可以。
历史原因：

1. InnoDB在作为MySQL的插件加入MySQL引擎家族之前，就已经是一个提供了崩溃恢复和事务支持的引擎了。InnoDB接入了MySQL后，发现既然binlog没有崩溃恢复的能力，那引入InnoDB原有的redo log来保证崩溃恢复能力。
   实现原因：
2. binlog没有记录数据页修改的详细信息，不具备恢复数据页的能力。binlog记录着数据行的增删改，但是不记录事务对数据页的改动，这样细致的改动只记录在redo log中。当一个事务做增删改时，其实涉及到的数据页改动非常细致和复杂，包括行的字段改动以及行头部以及数据页头部的改动，甚至b+tree会因为插入一行而发生若干次页面分裂，那么事务也会把所有这些改动记录下来到redo log中。因为数据库系统进程crash时刻，磁盘上面页面镜像可以非常混乱，其中有些页面含有一些正在运行着的事务的改动，而一些已提交的事务的改动并没有刷上磁盘。事务恢复过程可以理解为是要把没有提交的事务的页面改动都去掉，并把已经提交的事务的页面改动都加上去这样一个过程。这些信息，都是binlog中没有记录的，只记录在了存储引擎的redo log中。
3. 操作写入binlog可细分为write和fsync两个过程，write指的就是指把日志写入到文件系统的page cache，并没有把数据持久化到磁盘,fsync才是将数据持久化到磁盘的操作。通过参数设置sync_binlog为0的时候，表示每次提交事务都只write，不fsync。此时数据库崩溃可能导致部分提交的事务以及binlog日志由于没有持久化而丢失。

简述MySQL主从复制

MySQL提供主从复制功能，可以方便的实现数据的多处自动备份，不仅能增加数据库的安全性，还能进行读写分离，提升数据库负载性能。

主从复制流程：

1. 在事务完成之前，主库在binlog上记录这些改变，完成binlog写入过程后，主库通知存储引擎提交事物
2. 从库将主库的binlog复制到对应的中继日志，即开辟一个I/O工作线程，I/O线程在主库上打开一个普通的连接，然后开始binlog dump process，将这些事件写入中继日志。从主库的binlog中读取事件，如果已经读到最新了，线程进入睡眠并等待ma主库产生新的事件。

读写分离：即只在MySQL主库上写，只在MySQL从库上读，以减少数据库压力，提高性能。

我是小牛，非科班转行的微软程序员新人一枚。

我会在这里分享一下自己的转行学习路线、个人经历、内推信息等等！欢迎大家转载我的原创文章，可在公众号下留言！