MySQL 面试题

1、隔离级别与锁的关系

回答这个问题，可以先阐述四种隔离级别，再阐述它们的实现原理。隔离级别就是依赖锁和 MVCC 实现的。

2、实践中如何优化 MySQL？

最好是按照以下顺序优化：

- SQL 语句及索引的优化
- 数据库表结构的优化
- 系统配置的优化
- 硬件的优化

3、优化子查询

- 用关联查询替代。
- 优化 GROUP BY 和 DISTINCT。
- 这两种查询据可以使用索引来优化，是最有效的优化方法。
- 关联查询中，使用标识列分组的效率更高。

- 如果不需要 ORDER BY，进行 GROUP BY 时加 ORDER BY NULL，MySQL 不会再进行文件排序。
- WITH ROLLUP 超级聚合，可以挪到应用程序处理。

4、前缀索引

- 语法：index(field(10))，使用字段值的前 10 个字符建立索引，默认是使用字段的全部内容建立索引。
- 前提：前缀的标识度高。比如密码就适合建立前缀索引，因为密码几乎各不相同。
- 实操的难度：在于前缀截取的长度。
- 我们可以利用 select count(*)/count(distinct left(password,prefixLen));，通过从调整 prefixLen 的值（从 1 自增）查看不同前缀长度的一个平均匹配度，接近 1 时就可以了（表示一个密码的前 prefixLen 个字符几乎能确定唯一一条记录）。

5、MySQL 5.6 和 MySQL 5.7 对索引做了哪些优化？

- MySQL5.6 引入了索引下推优化，默认是开启的。
- 例子：user 表中（a,b,c）构成一个索引。
- select * from user where a='23' and b like '%eqw%' and c like 'dasd'。
- 解释：如果没有索引下推原则，则 MySQL 会通过 a='23' 先查询出一个对应的数据。然后返回到 MySQL 服务端。MySQL 服务端再基于两个 like 模糊查询来校验 and 查询出的数据是否符合条件。这个过程就设计到回表操作。

- 如果使用了索引下推技术，则 MySQL 会首先返回返回条件 a='23'的数据的索引，然后根据模糊查询的条件来校验索引行数据是否符合条件，如果符合条件，则直接根据索引来定位对应的数据，如果不符合直接 reject 掉。因此，有了索引下推优化，可以在有 like 条件的情况下，减少回表的次数。

6、MySQL 有关权限的表有哪几个呢？

MySQL 服务器通过权限表来控制用户对数据库的访问，权限表存放在 MySQL 数据库里，由 MySQL_install_db 脚本初始化。这些权限表分别 user，db，table_priv，columns_priv 和 host。

1、 user 权限表：记录允许连接到服务器的用户帐号信息，里面的权限是全局级的。

2、 db 权限表：记录各个帐号在各个数据库上的操作权限。

3、 table_priv 权限表：记录数据表级的操作权限。

4、 columns_priv 权限表：记录数据列级的操作权限。

5、 host 权限表：配合 db 权限表对给定主机上数据库级操作权限作更细致的控制。这个权限表不受 GRANT 和 REVOKE 语句的影响。

7、MySQL 中都有哪些触发器？

MySQL 数据库中有六种触发器：

- Before Insert
- After Insert

- Before Update
- After Update
- Before Delete
- After Delete

8、大表怎么优化？分库分表了是怎么做的？分表分库了有什么问题？有用到中间件么？他们的原理知道么？

当 MySQL 单表记录数过大时，数据库的 CRUD 性能会明显下降，一些常见的优化措施如下：

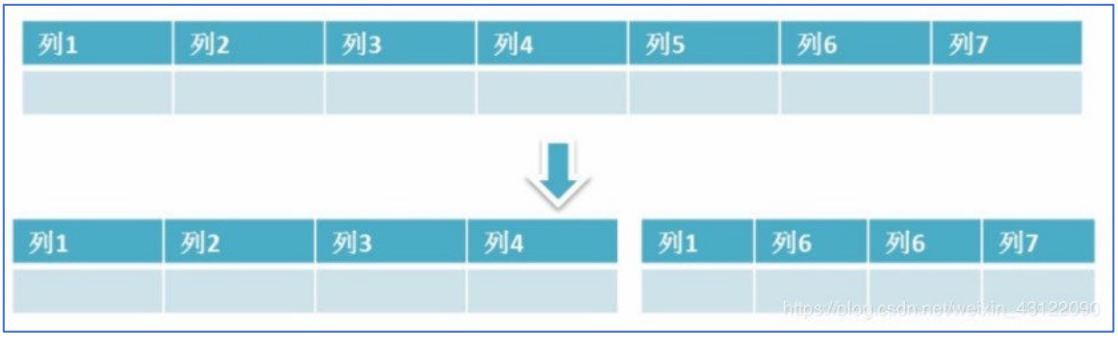
- 限定数据的范围： 务必禁止不带任何限制数据范围条件的查询语句。比如：我们当用户在查询订单历史的时候，我们可以控制在一个月的范围内。；
- 读/写分离： 经典的数据库拆分方案，主库负责写，从库负责读；
- 缓存： 使用 MySQL 的缓存，另外对重量级、更新少的数据可以考虑使用应用级别的缓存；

还有就是通过分库分表的方式进行优化。主要有垂直分区、垂直分表、水平分区、水平分表

垂直分区

1、 根据数据库里面数据表的相关性进行拆分。 例如，用户表中既有用户的登录信息又有用户的基本信息，可以将用户表拆分成两个单独的表，甚至放到单独的库做分库。

2、 简单来说垂直拆分是指数据表列的拆分，把一张列比较多的表拆分为多张表。 如下图所示，这样来说大家应该就更容易理解了。



垂直拆分的优点：

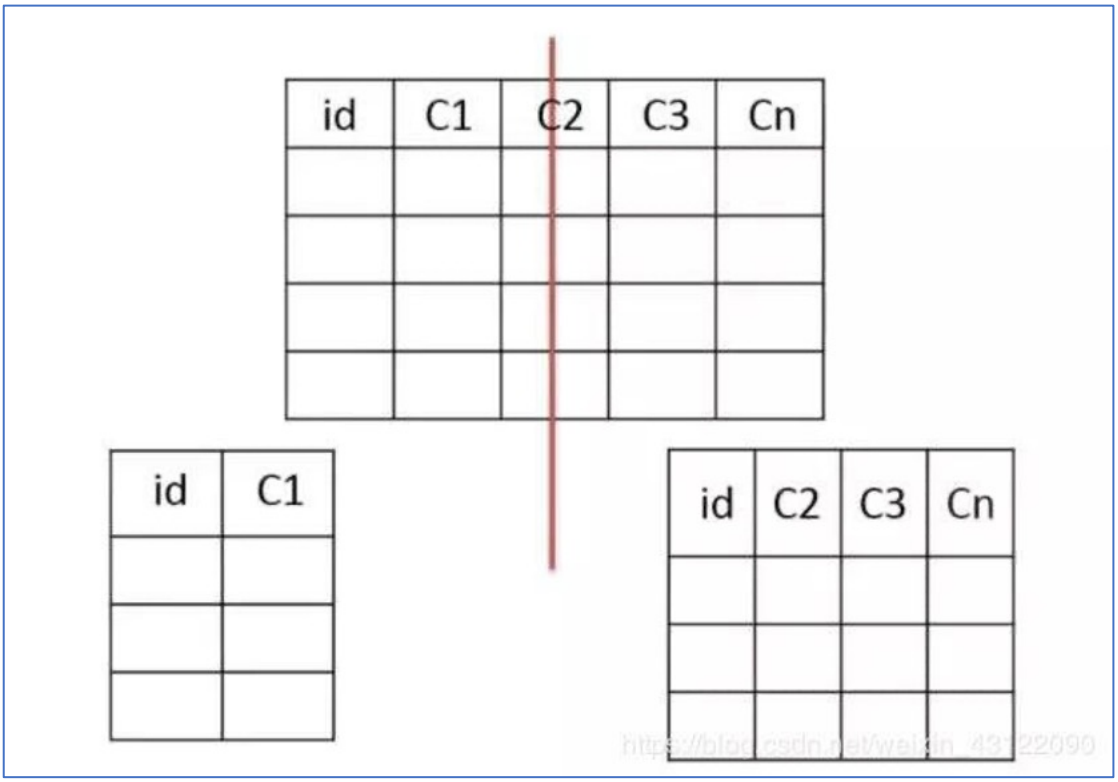
可以使得行数据变小，在查询时减少读取的 Block 数，减少 I/O 次数。此外，垂直分区可以简化表的结构，易于维护。

垂直拆分的缺点：

主键会出现冗余，需要管理冗余列，并会引起 Join 操作，可以通过在应用层进行 Join 来解决。此外，垂直分区会让事务变得更加复杂。

垂直分表

把主键和一些列放在一个表，然后把主键和另外的列放在另一个表中



适用场景

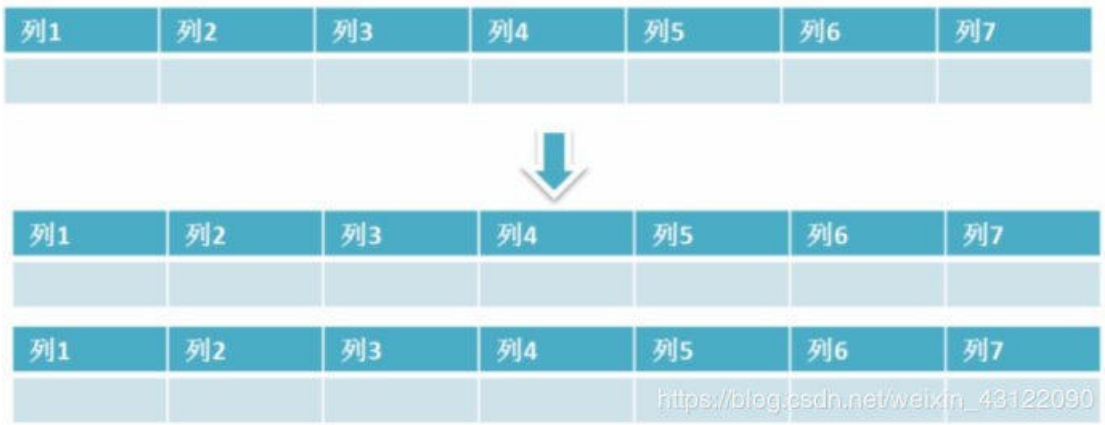
- 如果一个表中某些列常用，另外一些列不常用
- 可以使数据行变小，一个数据页能存储更多数据，查询时减少 I/O 次数

缺点

- 有些分表的策略基于应用层的逻辑算法，一旦逻辑算法改变，整个分表逻辑都会改变，扩展性较差
- 对于应用层来说，逻辑算法增加开发成本
- 管理冗余列，查询所有数据需要 join 操作

水平分区

- 保持数据表结构不变，通过某种策略存储数据分片。这样每一片数据分散到不同的表或者库中，达到了分布式的目的。 水平拆分可以支撑非常大的数据量。
- 水平拆分是指数据表行的拆分，表的行数超过 200 万行时，就会变慢，这时可以把一张的表的数据拆成多张表来存放。举个例子：我们可以将用户信息表拆分成多个用户信息表，这样就可以避免单一表数据量过大对性能造成影响。

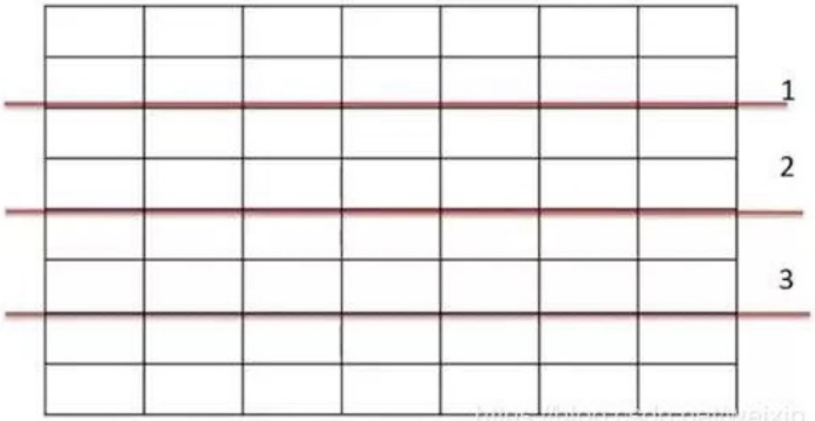


- 水品拆分可以支持非常大的数据量。需要注意的一点是:分表仅仅是解决了单一表数据过大的问题，但由于表的数据还是在同一台机器上，其实对于提升 MySQL 并发能力没有什么意义，所以 水平拆分最好分库 。

- 水平拆分能够支持非常大的数据量存储，应用端改造也少，但 分片事务难以解决 ，跨界点 Join 性能较差，逻辑复杂。

水平分表：

表很大，分割后可以降低在查询时需要读的数据和索引的页数，同时也降低了索引的层数，提高查询次数。



适用场景

- 表中的数据本身就有独立性，例如表中分表记录各个地区的数据或者不同时期的数据，特别是有些数据常用，有些不常用。
- 需要把数据存放在多个介质上。

水平切分的缺点

- 给应用增加复杂度，通常查询时需要多个表名，查询所有数据都需 UNION 操作。
- 在许多数据库应用中，这种复杂度会超过它带来的优点，查询时会增加读一个索引层的磁盘次数。

数据库分片的两种常见方案：

客户端代理：

分片逻辑在应用端，封装在 jar 包中，通过修改或者封装 JDBC 层来实现。当当网的 Sharding-JDBC 、阿里的 TDDL 是两种比较常用的实现。

中间件代理：

在应用和数据中间加了一个代理层。分片逻辑统一维护在中间件服务中。** 我们现在谈的 Mycat** 、360 的 Atlas、网易的 DDB 等等都是这种架构的实现。

分库分表后面临的问题

事务支持

分库分表后，就成了分布式事务了。如果依赖数据库本身的分布式事务管理功能去执行事务，将付出高昂的性能代价； 如果由应用程序去协助控制，形成程序逻辑上的事务，又会造成编程方面的负担。

跨库 join

只要是进行切分，跨节点 Join 的问题是不可避免的。但是良好的设计和切分却可以减少此类情况的发生。解决这一问题的普遍做法是分两次查询实现。在第一次查询的结果集中找出关联数据的 id,根据这些 id 发起第二次请求得到关联数据。

数据迁移，容量规划，扩容等问题

来自淘宝综合业务平台团队，它利用对 2 的倍数取余具有向前兼容的特性（如对 4 取余得 1 的数对 2 取余也是 1）来分配数据，避免了行级别的数据迁移，但是依然需要进行表级别的迁移，同时对扩容规模和分表数量都有限制。总得来说，这些方案都不是十分的理

想，多多少少都存在一些缺点，这也从一个侧面反映出了 Sharding 扩容的难度。

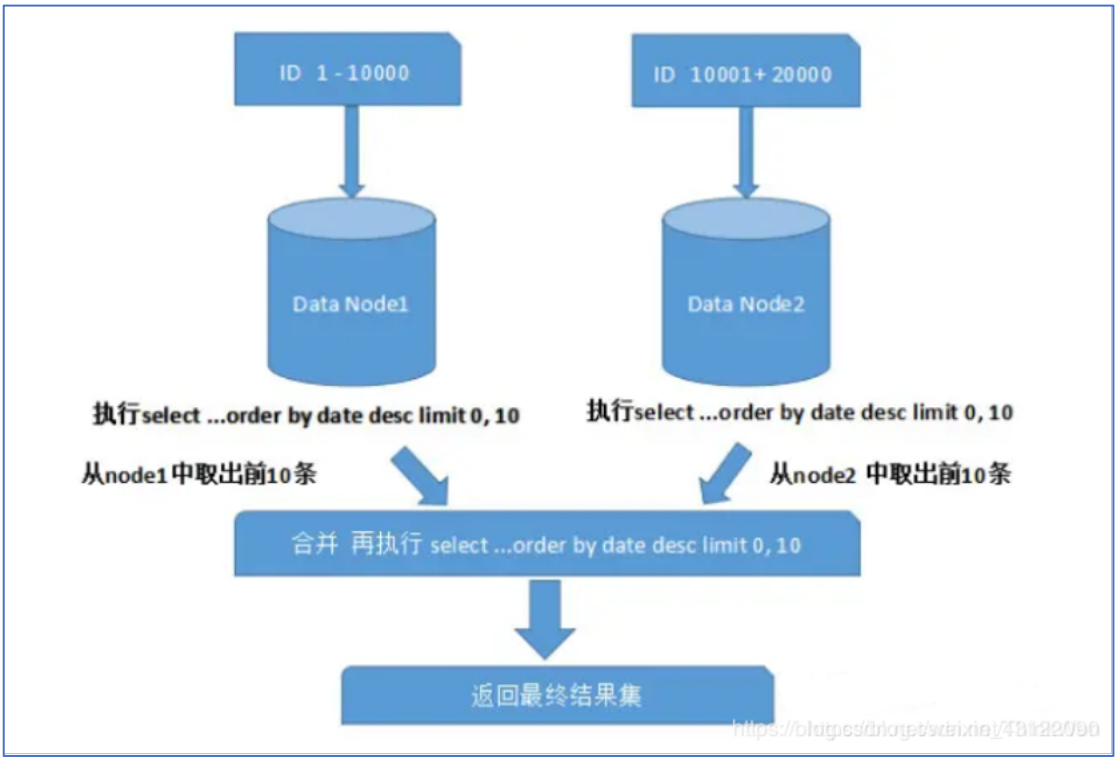
ID 问题

一旦数据库被切分到多个物理结点上，我们将不能再依赖数据库自身的主键生成机制。一方面，某个分区数据库自生成的 ID 无法保证在全局上是唯一的；另一方面，应用程序在插入数据之前需要先获得 ID,以便进行 SQL 路由、一些常见的主键生成策略

UUID 使用 UUID 作主键是最简单的方案，但是缺点也是非常明显的。由于 UUID 非常的长，除占用大量存储空间外，最主要的问题是在索引上，在建立索引和基于索引进行查询时都存在性能问题。 Twitter 的分布式自增 ID 算法 Snowflake 在分布式系统中，需要生成全局 UID 的场合还是比较多的，twitter 的 snowflake 解决了这种需求，实现也还是很简单的，除去配置信息，核心代码就是毫秒级时间 41 位 机器 ID 10 位 毫秒内序列 12 位。

跨分片的排序分页问题

一般来讲，分页时需要按照指定字段进行排序。当排序字段就是分片字段的时候，我们通过分片规则可以比较容易定位到指定的分片，而当排序字段非分片字段的时候，情况就会变得比较复杂了。为了最终结果的准确性，我们需要在不同的分片节点中将数据进行排序并返回，并将不同分片返回的结果集进行汇总和再次排序，最后再返回给用户。如下图所示：



9、B+ Tree 索引和 Hash 索引区别？

- hash 索引适合等值查询，但是无法进行范围查询。
- hash 索引没办法利用索引完成排序。
- hash 索引不支持多列联合索引的最左匹配规则。
- 如果有大量重复健值得情况下，hash 索引的效率会很低，因为哈希碰撞问题。

10、数据库索引的原理，为什么要用 B+树，为什么不用二叉树？

可以从几个维度去看这个问题，查询是否够快，效率是否稳定，存储数据多少，以及查找磁盘次数，为什么不是二叉树，为什么不是平衡二叉树，为什么不是 B 树，而偏偏是 B+

树呢？

为什么不是一般二叉树？

如果二叉树特殊化为一个链表，相当于全表扫描。平衡二叉树相比于二叉查找树来说，查找效率更稳定，总体的查找速度也更快。

为什么不是平衡二叉树呢？

我们知道，在内存比在磁盘的数据，查询效率快得多。如果树这种数据结构作为索引，那我们每查找一次数据就需要从磁盘中读取一个节点，也就是我们说的一个磁盘块，但是平衡二叉树可是每个节点只存储一个键值和数据的，如果是 B 树，可以存储更多的节点数据，树的高度也会降低，因此读取磁盘的次数就降下来啦，查询效率就快啦。

那为什么不是 B 树而是 B+树呢？

1）B+树非叶子节点上是不存储数据的，仅存储键值，而 B 树节点中不仅存储键值，也会存储数据。innodb 中页的默认大小是 16KB，如果不存储数据，那么就会存储更多的键值，相应的树的阶数（节点的子节点树）就会更大，树就会更矮更胖，如此一来我们查找数据进行磁盘的 IO 次数有会再次减少，数据查询的效率也会更快。

2）B+树索引的所有数据均存储在叶子节点，而且数据是按照顺序排列的，链表连着的。那么 B+树使得范围查找，排序查找，分组查找以及去重查找变得异常简单。