MYSQL索引面试题

如何建立和删除索引

说一说你对MySQL索引的理解

怎么判断要不要加索引

select in 语句中如何使用索引

MySQL的索引为什么用B+树?

联合索引的存储结构是什么，它的有效方式是什么?

如何建立和删除索引

```Java
Create TABLE table_name{
    各列的信息
    (KEY|INDEX) index_name (需要被建立索引的单个或多个列)
    //(HASH) hash_index_name (需要被建立索引的单个或多个列), InnoDB不支持手动创建哈希索引
}

ALTER TABLE table_name ADD (KEY|INDEX) index_name (需要被建立索引的单个或多个列);

ALTER TABLE table_name DROP (KEY|INDEX) index_name
```

说一说你对MySQL索引的理解

在我的理解里，索引是对数据排序及快速定位的解决方案，本质上是一类特定的数据结构，比如最常见的B+树，哈希表(InnoDB不支持手动创建哈希索引)等。以B+树为例，在使用InnoDB引擎的情况下，mysql找到数据所在的数据页所进行的IO操作次数，等于B+树的高度，总的来说就是可以明显的提高查询效率。

在创建表时，Mysql会自动为表的主键建立聚簇索引，在有需要的情况下，我们也可以为其他列创建二级索引。两者的不同点在于，聚簇索引保存了完整的记录，但二级索引中只保存了其列属性以及对应的主键，想要获得其他数据还需要进行一次回表操作。

除此之外，索引的使用是有空间代价和时间代价的，因为二级索引需要占据额外的空间，在建立一些字符串类型的索引时，可以选择只使用其前缀来建立二级索引，时间代价上，对某个表进行增删改记录时，可能需要对基于该表的所有索引都要进行调整。

而且并不是所有列都适合建立索引，建立索引后，sql语句写得不好，可能会用不上索引

怎么判断要不要加索引

首先，添加的索引应该要满足某些业务需求的，比如频繁作为查找条件的字段，经常用于排序和分段的字段等。

与此同时，这里有个非常重要的判断指标Cardinality，表示该索引不重复的值的数量，优化器会根据这个值来判断是否使用这个索引，虽然这个值并不是精确的。假如这个值与表的总记录数相差不大，

Cardinality统计信息的更新发生在两个操作中：insert和update。InnoDB存储引擎内部对更新Cardinality信息的策略为：①表中1/16的数据已发生了改变；②stat_modified_counter>2000 000 000

第一种策略为自从上次统计Cardinality信息后，表中的1/16的数据已经发生过变化，这是需要更新Cardinality信息

第二种情况考虑的是，如果对表中某一行数据频繁地进行更新操作，这时表中的数据实际并没有增加，实际发生变化的还是这一行数据，则第一种更新策略就无法适用这种情况，故在InnoDB存储引擎内部有一个计数器start_modified_counter,用来表示发生变化的 次数,当start_modified_counter>2 000 000 000 时，则同样更新Cardinality信息

select in 语句中如何使用索引

在这种情况下，使用索引执行查询语句很容易会产生许多单点扫描区间，如果所使用的索引不是唯一二级索引，那么就会获取索引对应B+树每个区间的最左记录与最右记录，然后在计算两条记录之间存在多少条记录（记录少时精确计算，记录多时估算），然后计算这种方式执行查询的成本。

为了避免In中的参数过多，导致成本计算的消耗过大，mysql中提供了一个系统变量eq_range_index_dive_limit，默认值为200，当in中的参数大于这个数时就会使用索引统计数据进行估算，使用SHOW TABLE STATUS显示出来的ROWS(表中的记录数)，除以使用SHOW INDEX显示出来的Cardinality（基数，表示该列中不重复的值的个数），从而推测出该列的某个值的平均重复次数，然后将这个平均重复次数✖In中的参数个数，从而估算出总共需要回表的记录数。这种估算的方式最大的问题就是不准确，可能会出现估算出来的查询成本与实际执行的成本相差甚远。

所以explain select in语句时，发现没有使用索引执行查询，可以尝试调整eq_range_index_dive_limit系统参数。

MySQL的索引为什么用B+树?

日常中对数据库的主要操作便是对数据的增删查改，链表的随机访问性能差，数组增删数据的效率低，因此使用有序的树形结构便是一个比较合适的选择。

与此同时，为了提高Mysql的性能，对数据的操作过程中要尽可能减少性能损耗较大的IO操作（频繁的IO是阻碍提高性能的瓶颈），二叉查找树容易产生高层数的情况导致需要大量的I/O操作，B+树与B树（B-树，二叉平衡树）是平衡的，树的高度会相对较低，而B+树的数据全保存于叶子节点，因此B+树的内节点能够保存的数据就大大增加，因此同等数据量下，B+树的高度会低于B树。

除此之外，B+树的叶子节点，也就是数据页，是由双向链表串联起来的，叶子节点的内部数据的关键字是递增的，更有利于数据的遍历与范围查找，这也是B+树结构索引与哈希索引的重要区别。

联合索引的存储结构是什么，它的有效方式是什么?

联合索引的存储结构是B+树，其本质上也是一个二级索引，叶结点中不存储完整数据，但是会存储对应的主键值，~~使用该类索引查询索引中不含有的列时会产生回表操作，~~但与普通二级索引不同之处在于，联合索引以多个列的大小作为排序规则，也就是同时为多个列创造一个索引。

联合索引的有效方式为，进行条件过滤时，先对第一列进行条件判断，