（重要）Java面试中常问的数据库方面问题

为什么用自增列作为主键

MySQL

为什么用自增列作为主键

1. @如果我们定义了主键(PRIMARY KEY)，那么InnoDB会选择主键作为聚集索引、@如果没有显式定义主键，则InnoDB会选择第一个不包含有NULL值的唯一索引作为主键索引、@如果也没有这样的唯一索引，则InnoDB会选择内置6字节长的ROWID作为隐含的聚集索引(ROWID随着行记录的写入而主键递增，这个ROWID不像ORACLE的ROWID那样可引用，是隐含的)。
2. 数据记录本身被存于主索引（一颗B+Tree）的叶子节点上。这就要求同一个叶子节点内（大小为一个内存页或磁盘页）的各条数据记录按主键顺序存放，因此每当有一条新的记录插入时，MySQL会根据其主键将其插入适当的节点和位置，如果页面达到装载因子（InnoDB默认为15/16），则开辟一个新的页（节点）
3. 如果表使用自增主键，那么每次插入新的记录，记录就会顺序添加到当前索引节点的后续位置，当一页写满，就会自动开辟一个新的页
4. 如果使用非自增主键（如果身份证号或学号等），由于每次插入主键的值近似于随机，因此每次新纪录都要被插到现有索引页得中间某个位置，此时MySQL不得不为了将新记录插到合适位置而移动数据，甚至目标页面可能已经被回写到磁盘上而从缓存中清掉，此时又要从磁盘上读回来，这增加了很多开销，同时频繁的移动、分页操作造成了大量的碎片，得到了不够紧凑的索引结构，后续不得不通过OPTIMIZE TABLE来重建表并优化填充页面。

为什么使用数据索引能提高效率

1. 数据索引的存储是有序的
2. 在有序的情况下，通过索引查询一个数据是无需遍历索引记录的
3. 极端情况下，数据索引的查询效率为二分法查询效率，趋近于 log2(N)

B+树索引和哈希索引的区别

B+树是一个平衡的多叉树，从根节点到每个叶子节点的高度差值不超过1，而且同层级的节点间有指针相互链接，是有序的

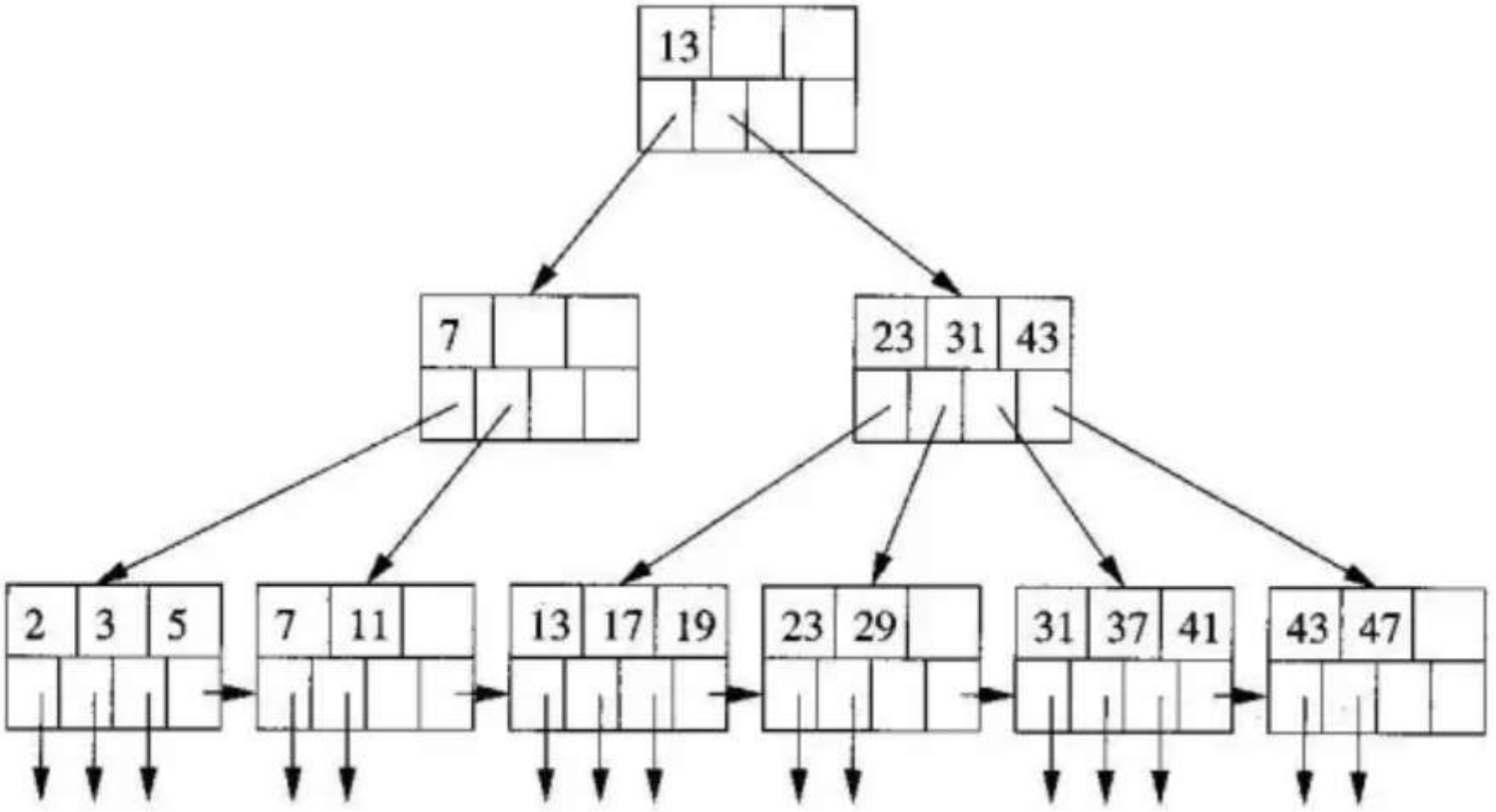
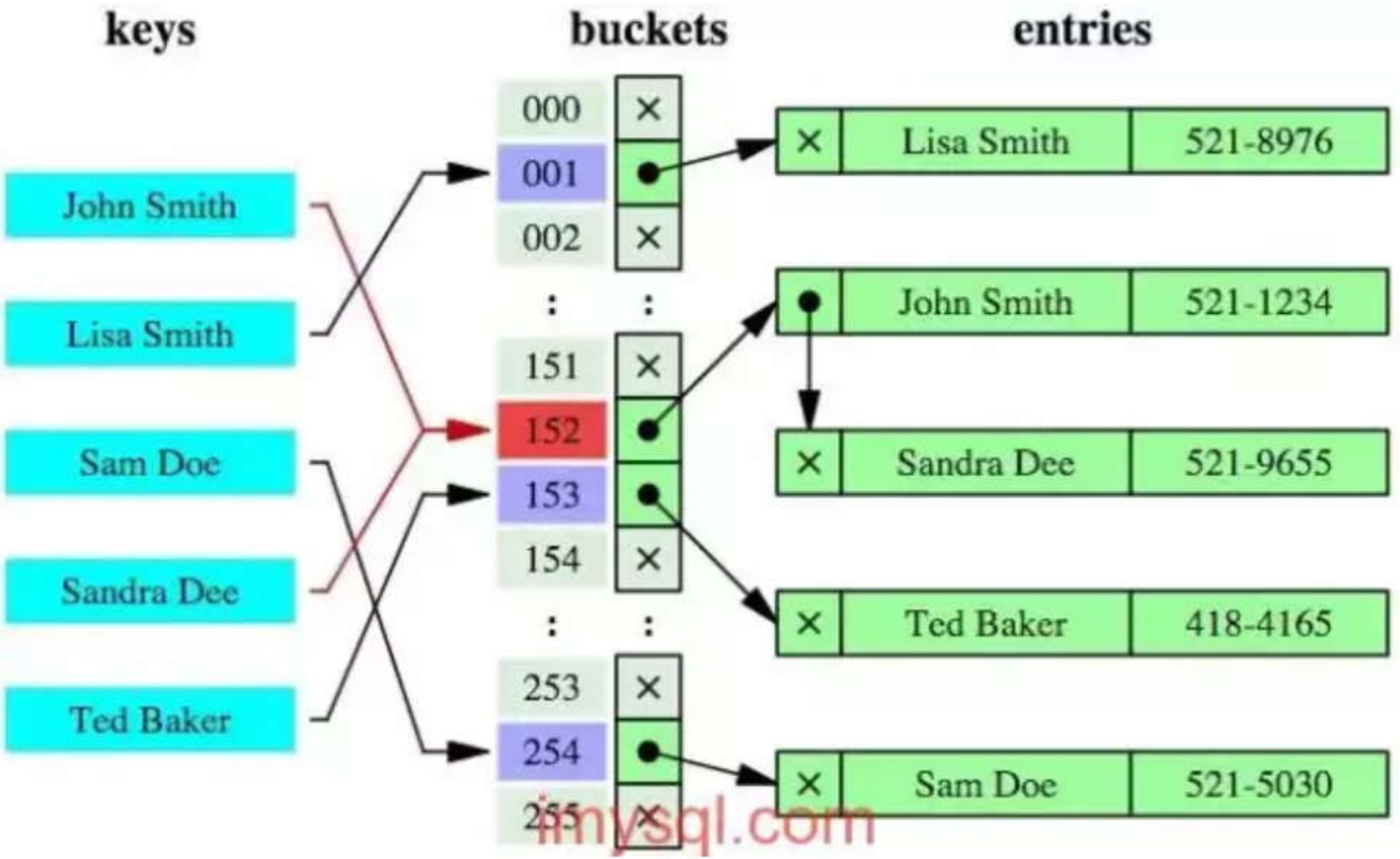


Figure 13.23: A B-tree

哈希索引就是采用一定的哈希算法，把键值换算成新的哈希值，检索时不需要类似B+树那样从根节点到叶子节点逐级查找，只需一次哈希算法即可,是无序的


keys
buckets
entries
John Smith
Lisa Smith
Sam Doe
Sandra Dee
Ted Baker
000
001
002
151
152
153
154
253
254
255
Lisa Smith
521-8976
John Smith
521-1234
Sandra Dee
521-9655
Ted Baker
418-4165
Sam Doe
521-5030
inmysql.com

哈希索引的优势：

1. 等值查询。哈希索引具有绝对优势（前提是：没有大量重复键值，如果大量重复键值时，哈希索引的效率很低，因为存在所谓的哈希碰撞问题。）

哈希索引不适用的场景：

1. 不支持范围查询
2. 不支持索引完成排序
3. 不支持联合索引的最左前缀匹配规则

通常，B+树索引结构适用于绝大多数场景，像下面这种场景用哈希索引才更有优势：

在HEAP表中，如果存储的数据重复度很低（也就是说基数很大），对该列数据以等值查询为主，没有范围查询、没有排序的时候，特别适合采用哈希索引，例如这种SQL：

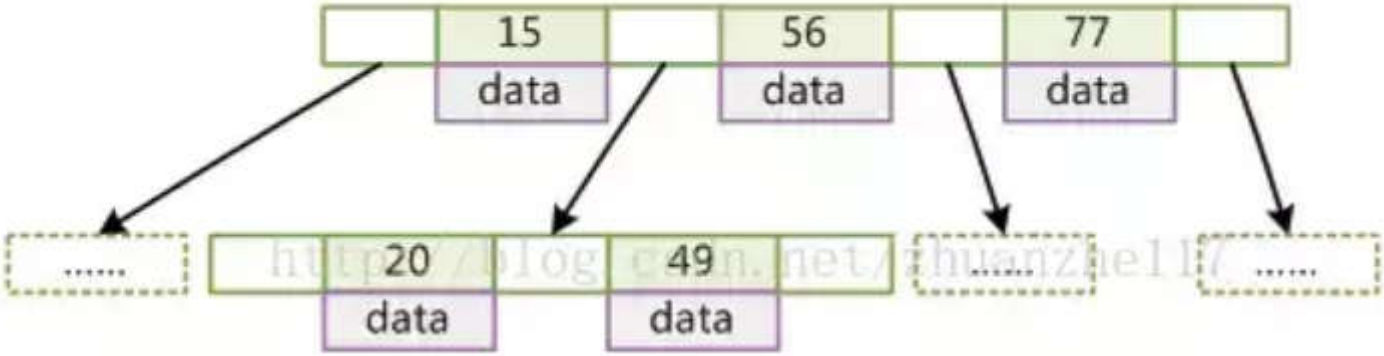
```
select id,name from table where name='李明'; — 仅等值查询
```

而常用的InnoDB引擎中默认使用的是B+树索引，它会实时监控表上索引的使用情况，如果认为建立哈希索引可以提高查询效率，则自动在内存中的“自适应哈希索引缓冲区”建立哈希索引（在InnoDB中默认开启自适应哈希索引），通过观察搜索模式，MySQL会利用index key的前缀建立哈希索引，如果一个表几乎大部分都在缓冲池中，那么建立一个哈希索引能够加快等值查询。

注意：在某些工作负载下，通过哈希索引查找带来的性能提升远大于额外的监控索引搜索情况和保持这个哈希表结构所带来的开销。但某些时候，在负载高的情况下，自适应哈希索引中添加的read/write锁也会带来竞争，比如高并发的join操作。like操作和%的通配符操作也不适用于自适应哈希索引，可能要关闭自适应哈希索引。

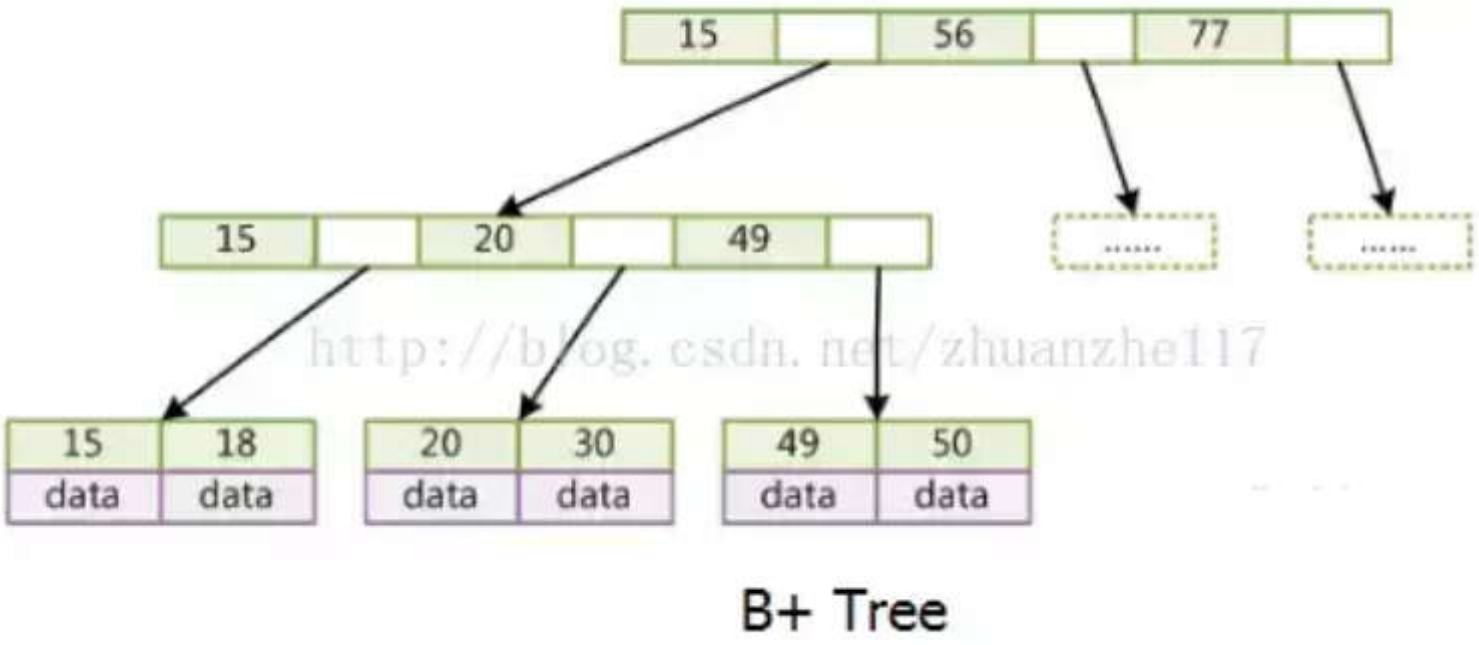
B树和B+树的区别

1. B树，每个节点都存储key和data，所有节点组成这棵树，并且叶子节点指针为nul，叶子结点不包含任何关键字信息。



B Tree

2. B+树，所有的叶子结点中包含了全部关键字的信息，及指向含有这些关键字记录的指针，且叶子结点本身依关键字的大小自小而大的顺序链接，所有的非终端结点可以看成是索引部分，结点中仅含有其子树根结点中最大（或最小）关键字。（而B 树的非终节点也包含需要查找的有效信息）



为什么说B+比B树更适合实际应用中操作系统的文件索引和数据库索引?

1. B+的磁盘读写代价更低B+的内部结点并没有指向关键字具体信息的指针。因此其内部结点相对B树更小。如果把所有同一内部结点的关键字存放在同一盘块中，那么盘块所能容纳的关键字数量也越多。一次性读入内存中的需要查找的关键字也就越多。相对来说IO读写次数也就降低了。
2. B+-tree的查询效率更加稳定由于非终结点并不是最终指向文件内容的结点，而只是叶子结点中关键字的索引。所以任何关键字的查找必须走一条从根结点到叶子结点的路。所有关键字查询的路径长度相同，导致每一个数据的查询效率相当。

MySQL联合索引

1. 联合索引是两个或更多个列上的索引。对于联合索引:Mysql从左到右的使用索引中的字段，一个查询可以只使用索引中的一部份，但只能是最左侧部分。例如索引是key index (a,b,c). 可以支持a 、 a,b 、 a,b,c 3种组合进行查找，但不支持 b,c进行查找 . 当最左侧字段是常量引用时，索引就十分有效。
2. 利用索引中的附加列，您可以缩小搜索的范围，但使用一个具有两列的索引 不同于使用两个单独的索引。复合索引的结构与电话簿类似，人名由姓和名构成，电话簿首先按姓氏对进行排序，然后按名字对有相同姓氏的人进行排序。如果您知 道姓，电话簿将非常有用；如果您知道姓和名，电话簿则更为有用，但如果您只知道名不姓，电话簿将没有用处。

什么情况下应不建或少建索引

1. 表记录太少
2. 经常插入、删除、修改的表
3. 数据重复且分布平均的表字段，假如一个表有10万行记录，有一个字段A只有T和F两种值，且每个值的分布概率大约为50%，那么对这种表A字段建索引一般不会提高数据库的查询速度。
4. 经常和主字段一块查询但主字段索引值比较多的表字段

MySQL分区

一. 什么是表分区?

表分区，是指根据一定规则，将数据库中的一张表分解成多个更小的，容易管理的部分。从逻辑上看，只有一张表，但是底层却是由多个物理分区组成。

二. 表分区与分表的区别

分表：指的是通过一定规则，将一张表分解成多张不同的表。比如将用户订单记录根据时间成多个表。

分表与分区的区别在于：分区从逻辑上来讲只有一张表，而分表则是将一张表分解成多张表。

三. 表分区有什么好处?

1. 分区表的数据可以分布在不同的物理设备上，从而高效地利用多个硬件设备。 2. 和单个磁盘或者文件系统相比，可以存储更多数据
2. 优化查询。在where语句中包含分区条件时，可以只扫描一个或多个分区表来提高查询效率；涉及sum和count语句时，也可以在多个分区上并行处理，最后汇总结果。
3. 分区表更容易维护。例如：想批量删除大量数据可以清除整个分区。
4. 可以使用分区表来避免某些特殊的瓶颈，例如InnoDB的单个索引的互斥访问，ext3问价你系统的inode锁竞争等。

四. 分区表的限制因素

1. 一个表最多只能有1024个分区
2. MySQL5.1中，分区表达式必须是整数，或者返回整数的表达式。在MySQL5.5中提供了非整数表达式分区的支持。
3. 如果分区字段中有主键或者唯一索引的列，那么多有主键列和唯一索引列都必须包含进来。即：分区字段要么不包含主键或者索引列，要么包含全部主键和索引列。
4. 分区表中无法使用外键约束
5. MySQL的分区适用于一个表的所有数据和索引，不能只对表数据分区而不对索引分区，也不能只对索引分区而不对表分区，也不能只对表的一部分数据分区。

五. 如何判断当前MySQL是否支持分区?

命令：show variables like '%partition%' 运行结果:
mysql> show variables like '%partition%';+------------------+-------+| Variable_name | Value |+------------------+-------+| have_partitioning | YES |+------------------+-------+1 row in set (0.00 sec)have_partintioning 的值为YES，表示支持分区。

六. MySQL支持的分区类型有哪些?

1. RANGE分区： 这种模式允许将数据划分不同范围。例如可以将一个表通过年份划分成若干个分区

2. LIST分区： 这种模式允许系统通过预定义的列表的值来对数据进行分割。按照List中的值分区，与RANGE的区别是，range分区的区间范围值是连续的。
3. HASH分区 ： 这中模式允许通过对表的一个或多个列的Hash Key进行计算，最后通过这个Hash码不同数值对应的数据区域进行分区。例如可以建立一个对表主键进行分区的表。
4. KEY分区 ： 上面Hash模式的一种延伸，这里的Hash Key是MySQL系统产生的。

四种隔离级别

1. Serializable (串行化)：可避免脏读、不可重复读、幻读的发生。
2. Repeatable read (可重复读)：可避免脏读、不可重复读的发生。
3. Read committed (读已提交)：可避免脏读的发生。
4. Read uncommitted (读未提交)：最低级别，任何情况都无法保证。

关于MVVC

MySQL InnoDB存储引擎，实现的是基于多版本的并发控制协议——MVCC (Multi-Version Concurrency Control) (注：与MVCC相对的，是基于锁的并发控制，Lock-Based Concurrency Control)。MVCC最大的好处：读不加锁，读写不冲突。在读多写少的OLTP应用中，读写不冲突是非常重要的，极大的增加了系统的并发性能，现阶段几乎所有的RDBMS，都支持了MVCC。

1. LBCC：Lock-Based Concurrency Control，基于锁的并发控制。
2. MVCC：Multi-Version Concurrency Control，基于多版本的并发控制协议。纯粹基于锁的并发机制并发量低，MVCC是在基于锁的并发控制上的改进，主要是在读操作上提高了并发量。

在MVCC并发控制中，读操作可以分成两类：

1. 快照读 (snapshot read)：读取的是记录的可见版本 (有可能是历史版本)，不用加锁（共享读锁s锁也不加，所以不会阻塞其他事务的写）。

2. 当前读 (current read)：读取的是记录的最新版本，并且，当前读返回的记录，都会加上锁，保证其他事务不会再并发修改这条记录。

行级锁定的优点：

1. 当在许多线程中访问不同的行时只存在少量锁定冲突。
2. 回滚时只有少量的更改
3. 可以长时间锁定单一的行。

行级锁定的缺点：

1. 比页级或表级锁定占用更多的内存。
2. 当在表的大部分中使用时，比页级或表级锁定速度慢，因为你必须获取更多的锁。
3. 如果你在大部分数据上经常进行GROUP BY操作或者必须经常扫描整个表，比其它锁定明显慢很多。
4. 用高级别锁定，通过支持不同的类型锁定，你也可以很容易地调节应用程序，因为其锁成本小于行级锁定。

MySQL触发器简单实例

1. CREATE TRIGGER <触发器名称> --触发器必须有名字，最多64个字符，可能后面会附有分隔符.它和MySQL中其他对象的命名方式基本相象.
2. { BEFORE | AFTER } --触发器有执行的时间设置：可以设置为事件发生前或后。
3. { INSERT | UPDATE | DELETE } --同样也能设定触发的事件：它们可以在执行insert、update或delete的过程中触发。
4. ON <表名称> --触发器是属于某一个表的:当在这个表上执行插入、 更新或删除操作的时候就导致触发器的激活. 我们不能给同一张表的同一个事件安排两个触发器。
5. FOR EACH ROW --触发器的执行间隔：FOR EACH ROW子句通知触发器 每隔一行执行一次动作，而不是对整个表执行一次。
6. <触发器SQL语句> --触发器包含所要触发的SQL语句：这里的语句可以是任何合法的语句， 包括复合语句，但是这里的语句受的限制和函数的一样。

什么是存储过程

简单的说，就是一组SQL语句集，功能强大，可以实现一些比较复杂的逻辑功能，类似于JAVA语言中的方法；

ps:存储过程跟触发器有点类似，都是一组SQL集，但是存储过程是主动调用的，且功能比触发器更加强大，触发器是某件事触发后自动调用；

有哪些特性

1. 有输入输出参数，可以声明变量，有if/else, case,while等控制语句，通过编写存储过程，可以实现复杂的逻辑功能；
2. 函数的普遍特性：模块化，封装，代码复用；
3. 速度快，只有首次执行需经过编译和优化步骤，后续被调用可以直接执行，省去以上步骤；

```
DROP PROCEDURE IF EXISTS `proc_adder`;
DELIMITER ;;
CREATE DEFINER=`root`@`localhost` PROCEDURE `proc_adder`(IN a int, IN b int, OUT sum int)
BEGIN
  #Routine body goes here...

  DECLARE c int;
  if a is null then set a = 0;
  end if;

  if b is null then set b = 0;
  end if;

  set sum  = a + b;
```

```
END
;;
DELIMITER ;

set @b=5;
call proc_adder(0,@b,@s);
SELECT @s as sum;



create table tab2(
  tab2_id varchar(11)
);

DROP TRIGGER if EXISTS t_ai_on_tab1;
create TRAILING t_ai_on_tab1
AFTER INSERT ON tab1
for EACH ROW
BEGIN
  INSERT INTO tab2(tab2_id) values(new.tab1_id);
end;

INSERT INTO tab1(tab1_id) values('0001');

SELECT * FROM tab2;
```

MySQL优化

1. 开启查询缓存，优化查询
2. explain你的select查询，这可以帮你分析你的查询语句或是表结构的性能瓶颈。EXPLAIN 的查询结果还会告诉你你的索引主键被如何利用的，你的数据表是如何被搜索和排序的
3. 当只要一行数据时使用limit 1，MySQL数据库引擎会在找到一条数据后停止搜索，而不是继续往后查少下一条符合记录的数据
4. 为搜索字段建索引
5. 使用 ENUM 而不是 VARCHAR，如果你有一个字段，比如“性别”，“国家”，“民族”，“状态”或“部门”，你知道这些字段的取值是有限而且固定的，那么，你应该使用 ENUM 而不是VARCHAR。
6. Prepared StatementsPrepared Statements很像存储过程，是一种运行在后台的SQL语句集合，我们可以从使用 prepared statements 获得很多好处，无论是性能问题还是安全问题。Prepared Statements 可以检查一些你绑定好的变量，这样可以保护你的程序不会受到“SQL注入式”攻击
7. 垂直分表
8. 选择正确的存储引擎

key和index的区别

1. key 是数据库的物理结构，它包含两层意义和作用，一是约束（偏重于约束和规范数据库的结构完整性），二是索引（辅助查询用的）。包括primary key, unique key, foreign key 等
2. index是数据库的物理结构，它只是辅助查询的，它创建时会在另外的表空间（mysql中的innodb表空间）以一个类似目录的结构存储。索引要分类的话，分为前缀索引、全文本索引等；

Mysql 中 MyISAM 和 InnoDB 的区别有哪些?

区别：

1. InnoDB支持事务，MyISAM不支持，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；
2. InnoDB支持外键，而MyISAM不支持。对一个包含外键的InnoDB表转为MYISAM会失败；
3. InnoDB是聚集索引，数据文件是和索引绑在一起的，必须要有主键，通过主键索引效率很高。但是辅助索引需要两次查询，先查询到主键，然后再通过主键查询到数据。因此，主键不应该过大，因为主键太大，其他索引也都会很大。而MyISAM是非聚集索引，数据文件是分离的，索引保存的是数据文件的指针。主键索引和辅助索引是独立的。
4. InnoDB不保存表的具体行数，执行select count(*) from table时需要全表扫描。而MyISAM用一个变量保存了整个表的行数，执行上述语句时只需要读出该变量即可，速度很快；
5. Innodb不支持全文索引，而MyISAM支持全文索引，查询效率上MyISAM要高；

如何选择：

1. 是否要支持事务，如果要请选择innodb，如果不需要可以考虑MyISAM；
2. 如果表中绝大多数都只是读查询，可以考虑MyISAM，如果既有读写也挺频繁，请使用InnoDB。
3. 系统奔溃后，MyISAM恢复起来更困难，能否接受；
4. 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，如果你不知道用什么，那就用InnoDB，至少不会差。

数据库表创建注意事项

一、字段名及字段配制合理性

1. 剔除关系不密切的字段
2. 字段命名要有规则及相对应的含义（不要一部分英文，一部分拼音，还有类似a.b.c这样不明含义的字段）
3. 字段命名尽量不要使用缩写（大多数缩写都不能明确字段含义）
4. 字段不要大小写混用（想要具有可读性，多个英文单词可使用下划线形式连接）
5. 字段名不要使用保留字或者关键字
6. 保持字段名和类型的一致性

7. 慎重选择数字类型
8. 给文本字段留足余量

二、系统特殊字段处理及建成后建议

1. 添加删除标记（例如操作人、删除时间）
2. 建立版本机制

三、表结构合理性配置

1. 多型字段的处理，就是表中是否存在字段能够分解成更小独立的几部分（例如：人可以分为男人和女人）
2. 多值字段的处理，可以将表分为三张表，这样使得检索和排序更加有调理，且保证数据的完整性！

四、其它建议

1. 对于大数据字段，独立表进行存储，以便影响性能（例如：简介字段）
2. 使用varchar类型代替char，因为varchar会动态分配长度，char指定长度是固定的。
3. 给表创建主键，对于没有主键的表，在查询和索引定义上有一定的影响。
4. 避免表字段运行为null，建议设置默认值（例如：int类型设置默认值为0）在索引查询上，效率立显！
5. 建立索引，最好建立在唯一和非空的字段上，建立太多的索引对后期插入、更新都存在一定的影响（考虑实际情况来创建）。

Redis

Redis单线程问题

单线程指的是网络请求模块使用了一个线程（所以不需考虑并发安全性），即一个线程处理所有网络请求，其他模块仍用了多个线程。

为什么说Redis能够快速执行

1. 绝大部分请求是纯粹的内存操作（非常快速）
2. 采用单线程,避免了不必要的上下文切换和竞争条件
3. 非阻塞IO - IO多路复用

Redis的内部实现

内部实现采用epoll，采用了epoll+自己实现的简单的事件框架。epoll中的读、写、关闭、连接都转化成了事件，然后利用epoll的多路复用特性，不在io上浪费一点时间 这3个条件不是相互独立的，特别是第一条，如果请求都是耗时的，采用单线程吞吐量及性能很差。redis为特殊的场景选择了合适的技术方案。

Redis关于线程安全问题

redis实际上是采用了线程封闭的观念，把任务封闭在一个线程，自然避免了线程安全问题，不过对于需要依赖多个redis操作的复合操作来说，依然需要锁，而且有可能是分布式锁。

使用Redis有哪些好处?

1. 速度快，因为数据存在内存中，类似于HashMap，HashMap的优势就是查找和操作的时间复杂度都是O(1)
2. 支持丰富数据类型，支持string，list，set，sorted set，hash
3. 支持事务，操作都是原子性，所谓的原子性就是对数据的更改要么全部执行，要么全部不执行
4. 丰富的特性：可用于缓存，消息，按key设置过期时间，过期后将会自动删除

redis相比memcached有哪些优势?

1. memcached所有的值均是简单的字符串，redis作为其替代者，支持更为丰富的数据类型
2. redis的速度比memcached快很多
3. redis可以持久化其数据
4. Redis支持数据的备份，即master-slave模式的数据备份。

5. 使用底层模型不同，它们之间底层实现方式 以及与客户端之间通信的应用协议不一样。Redis直接自己构建了VM 机制 ，因为一般的系统调用系统函数的话，会浪费一定的时间去移动和请求。
6. value大小：redis最大可以达到1GB，而memcache只有1MB

Redis主从复制

过程原理：

1. 当从库和主库建立MS关系后,会向主数据库发送SYNC命令
2. 主库接收到SYNC命令后会开始在后台保存快照(RDB持久化过程),并将期间接收到的写命令缓存起来
3. 当快照完成后,主Redis会将快照文件和所有缓存的写命令发送给从Redis
4. 从Redis接收到后,会载入快照文件并且执行收到的缓存的命令
5. 之后,主Redis每当接收到写命令时就会将命令发送从Redis，从而保证数据的一致

缺点：所有的slave节点数据的复制和同步都由master节点来处理,会照成master节点压力太大,使用主从从结构来解决

Redis两种持久化方式的优缺点

1. RDB 持久化可以在指定的时间间隔内生成数据集的时间点快照（point-in-time snapshot）
2. AOF 持久化记录服务器执行的所有写操作命令，并在服务器启动时，通过重新执行这些命令来还原数据集。
3. Redis 还可以同时使用 AOF 持久化和 RDB 持久化。当redis重启时,它会有限使用AOF文件来还原数据集,因为AOF文件保存的数据集通常比RDB文件所保存的数据集更加完整

RDB的优点：

1. RDB 是一个非常紧凑（compact）的文件，它保存了 Redis 在某个时间点上的数据集。 这种文件非常适合用于进行备份： 比如说，你可以在最近的 24 小时内，每小时备份一次 RDB 文件，并且在每个月的每一天，也备份一个 RDB 文件。 这样的话，即使遇上问题，也可以随时将数据集还原到不同的版本。

2. RDB 非常适用于灾难恢复（disaster recovery）：它只有一个文件，并且内容都非常紧凑，可以（在加密后）将它传送到别的数据中心，或者亚马逊 S3 中。
3. RDB 可以最大化 Redis 的性能：父进程在保存 RDB 文件时唯一要做的就是 fork 出一个子进程，然后这个子进程就会处理接下来的所有保存工作，父进程无须执行任何磁盘 I/O 操作。
4. RDB 在恢复大数据集时的速度比 AOF 的恢复速度要快

Redis常见的性能问题都有哪些？如何解决？

1. Master写内存快照，save命令调度rdbSave函数，会阻塞主线程的工作，当快照比较大时对性能影响是非常大的，会间断性暂停服务，所以Master最好不要写内存快照。
2. Master AOF持久化，如果不重写AOF文件，这个持久化方式对性能的影响是最小的，但是AOF文件会不断增大，AOF文件过大会影响Master重启的恢复速度。Master最好不要做任何持久化工作，包括内存快照和AOF日志文件，特别是不要启用内存快照做持久化,如果数据比较关键，某个Slave开启AOF备份数据，策略为每秒同步一次。
3. Master调用BGREWRITEAOF重写AOF文件，AOF在重写的时候会占大量的CPU和内存资源，导致服务load过高，出现短暂服务暂停现象。
4. Redis主从复制的性能问题，为了主从复制的速度和连接的稳定性，Slave和Master最好在同一个局域网内

Redis提供6种数据淘汰策略

1. volatile-lru：从已设置过期时间的数据集（server.db[i].expires）中挑选最近最少使用的数据淘汰
2. volatile-ttl：从已设置过期时间的数据集（server.db[i].expires）中挑选将要过期的数据淘汰
3. volatile-random：从已设置过期时间的数据集（server.db[i].expires）中任意选择数据淘汰
4. allkeys-lru：从数据集（server.db[i].dict）中挑选最近最少使用的数据淘汰
5. allkeys-random：从数据集（server.db[i].dict）中任意选择数据淘汰
6. no-enviction（驱逐）：禁止驱逐数据