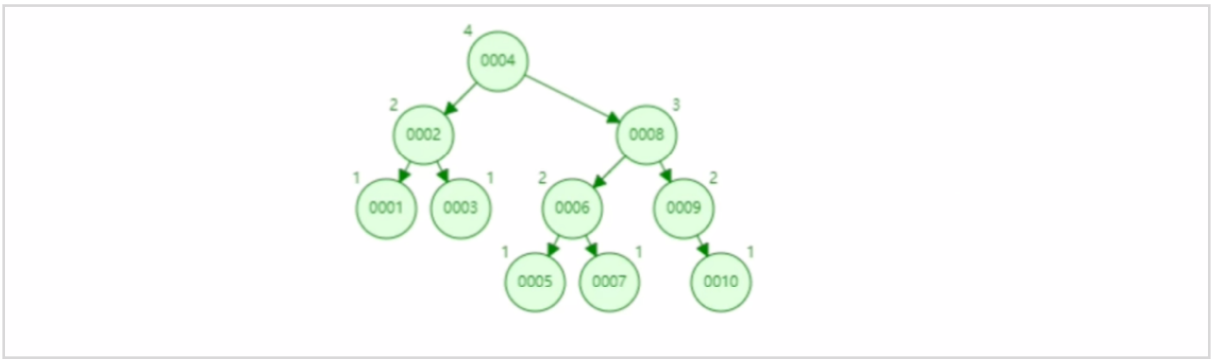
腾讯数据库面试题

1. MySQL为什么使用B+树来作索引，对比B树它的优点和缺点是什么?

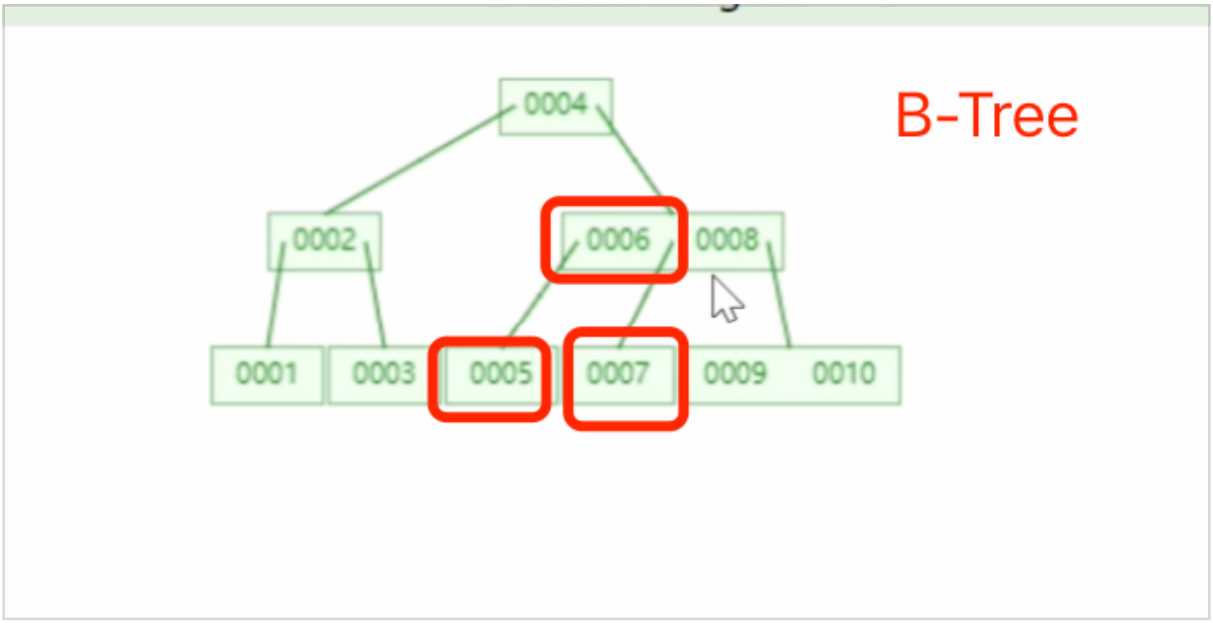
- 引申：
  - 为什么不能用哈希来作索引?

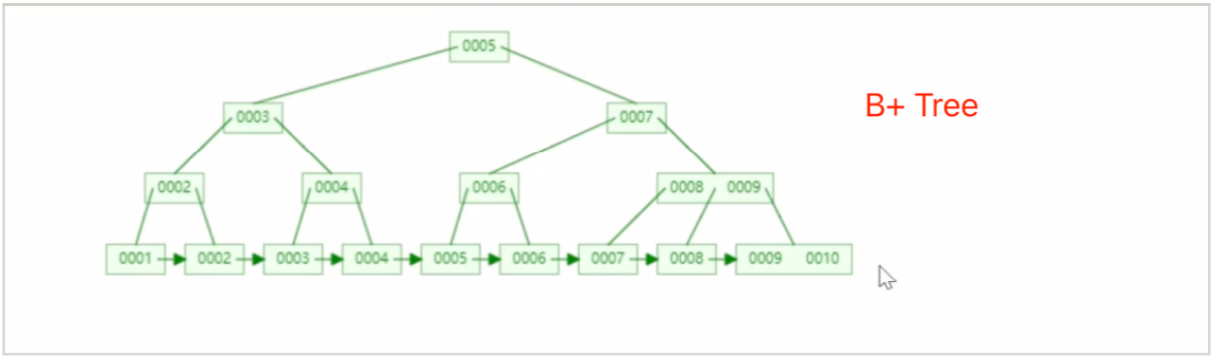
    因为哈希是无序的一个值，如果使用哈希就没法做范围查找，数据库需要很多大于小于的操作。且使用哈希的话就没法对数据进行排序

  - 为什么不用平衡二叉树?
    - 平衡二叉树树越高（数据越来越多的时候），查找速度就越来越慢了
    - 比如查找>5的树，那么查找的时候要首先找到5，然后再不断回旋找到其他>5的数字，如果>5的数据非常多，查找就会特别慢

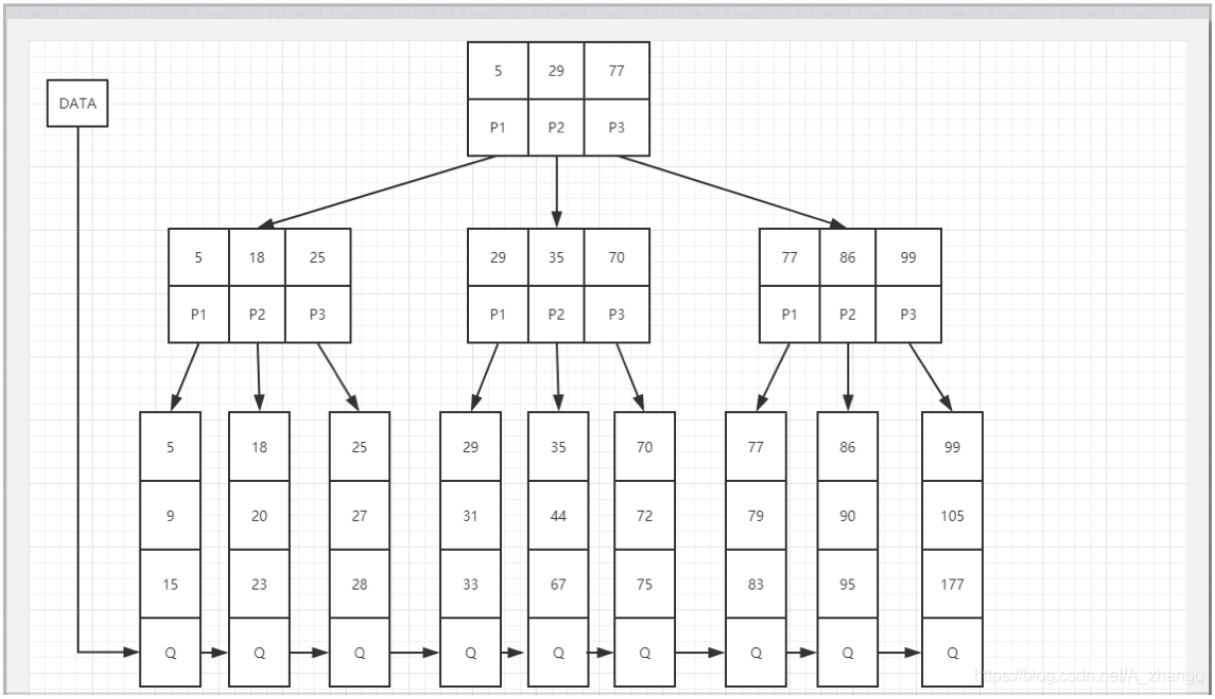


  - B树、B+树





> 因为无论是平衡的二叉搜索树，B树，都存在回旋查找的问题。
> 而B+树的叶子结点变成了一个链表，一个有序的链表，所以**解决了回旋查找的问题**。同时它也B树一样，一个节点可以存储多个值，**树的高度就比较低，所以查找速度也比较快**。
> （B+树的非叶子结点只存key，叶子结点既存key也存value，value就是数据的地址）



总结

- 数据库索引为什么是树?

> 数据结构中，`hash` 和 `tree` 相比，明显 `hash` 更占优势，`hash` 平均时间复杂度为 `O(1)`；`tree` 比如平衡二叉查找树，平均时间复杂度为 `O(lg(n))`；
>
> 但是我们需要数据库更多的是 `范围查询` 而不是单条操作。树的“有序”，可以提高效率；
>
> > InnoDB 不支持 hash 索引。

- MySQL为什么使用B+树来作索引：

mysql索引为什么用B+树_大米饭-CSDN博客

1. 节点为m叉，不是二叉，大大降低了树高度，可以减少IO次数
2. 可以充分利用磁盘，存储更多的key
3. 可以很快的遍历，范围查询，排序
4. 和B树相比，通过给叶子节点加链指针，解决了回旋查找的问题

- B+树对比B树它的优点和缺点是什么？
  - 优点：

1. 在B树的基础上进行了改进，只有叶子节点存储数据（B树的节点都存储数据，在数据库中**定位**的时候，会比B+树多出不必要的读取）
2. 叶子节点上增加了链表设计，更加适合**范围查询**（数据库常用），而且是一个有序链表，遍历不需要再中序进行
3. 因为非叶子节点不存储数据，所以**可以存储更多关键信息** 。比如存储索引（key），一个节点可以存储更多的key，查询定位会大大加快

> 范围查询，排序，可以存储更多的关键信息，这都一一命中了我们数据库的需求

  - 缺点：

1. **无论查找成功与否，都要走到叶子结点**。因为B+树的非叶子结点只是索引部分，这些结点中只含有子树的最大or最小关键字，当非叶子结点的关键字等于要查询的值时，查询也不会终止，而是会继续往下走到叶子结点

2. **B+树最大的性能问题是会产生大量的随机IO**

> 随着新数据的插入，叶子结点会慢慢分裂，逻辑上连续的叶子节点在物理上往往不连续，甚至分离的很远，但做范围查询时，会产生大量读随机IO；对于大量的随机写也一样，举一个插入key跨度很大的例子，如7->1000->3->2000 … 新插入的数据存储在磁盘上相隔很远，会产生大量的随机写IO

2. 数据库的事务隔离级别有哪些？各有哪些优缺点？

> https://blog.csdn.net/fuzhongmin05/article/details/91126936
> 事务的四大特性：ACID，其中Isolation隔离性又有多种隔离级别
>
> 但如果不考虑事务的隔离性，会发生的几种问题：

脏读	（针对未提交数据）事务1第二次读取的时候，读取到了事务2未提交的数据；若事务2回滚，则事务1第二次读取到的就是脏数据
不可重复读	（针对其他提交前后，读取数据的不同）事务2在事务1第二次读取时，提交了数据；导致事务1两次读取的结果不同。
幻读	（针对其他提交前后，读取数据**条数**的对比）当用户读取某一范围的数据行时，另一个事务又在该范围内插入了新行，当用户再读取该范围的数据行时，会发现有新的“幻影”行

不可重复读和幻读的区别：不可重复读的重点在修改，幻读的重点在新增或者删除

“脏读”、“不可重复读”和“幻读”，其实都是数据库读一致性问题，必须由数据库提供一定的事务隔离机制来解决。数据库实现事务隔离的方式，基本上可分为以下两种：

- 一种是在读取数据前，对其加锁，阻止其他事务对数据进行修改。
- 另一种是不用加任何锁，通过一定机制生成一个数据请求时间点的一致性数据快照（Snapshot），并用这个快照来提供一定级别（语句级或事务级）的一致性读取。从用户的角度来看，好像是数据库可以提供同一数据的多个版本，因此，这种技术叫做数据多版本并发控制（MultiVersion Concurrency Control，简称MVCC或MCC），也经常称为多版本数据库。

数据库的事务隔离越严格，并发副作用越小，但付出的代价也就越大，因为事务隔离实质上就是使事务在一定程度上 “串行化”进行，这显然与“并发”是矛盾的。同时，不同的应用对读一致性和事务隔离程度的要求也是不同的，比如许多应用对“不可重复读”和“幻读”并不敏感，可能更关心数据并发访问的能力

数据库提供四种隔离级别：

1. 读未提交：（最低级别，任何情况都会发生）
2. 读已提交：（可避免脏读的发生）
3. 可重复读：（可避免脏读、不可重复读的发生）MySQL的默认隔离级别，确保事务可以多次从一个字段中读取相同的值，在此事务持续期间，禁止其他事务对此字段的更新，可以避免脏读和不可重复读，仍会出现幻读问题
4. 串行化：（可避免脏读、不可重复读、幻读情况的发生）最严格的事务隔离级别，要求所有事务被串行执行，不能并发执行

事务隔离级别	脏读	不可重复读	幻读
读未提交（read-uncommitted）	是	是	是
不可重复读（read-committed）	否	是	是
可重复读（repeatable-read）	否	否	是
串行化（serializable）	否	否	否

四种隔离级别的优缺点：

	优点	缺点
读未提交	仅保证不读取物理损坏的数据	脏读、不可重复读、幻读都有可能出现
读已提交	避免了脏读	可能出现不可重复读、幻读
可重复读	在事务持续期间，静止其他事务对此字段进行更新，避免了脏读和不可重复读的问题	可能会出现幻读
串行化	可以避免脏读、不可重复读、幻读的问题	但是数据库中的所有事务都是串联的，在这个隔离级别中，没有并发事务

补充：串行化的缺点还有代价花费最高，性能最低

3. 什么是数据库事务，MySQL为什么会使用InnoDB作为默认选项?

什么是数据库事务

指作为**单个逻辑工作单元**执行的**一系列操作**，要么完全地执行，要么完全地不执行

简单的说，事务就是**并发控制的单位**，是**用户定义的一个操作序列**

而一个逻辑工作单元要成为事务，就必须满足ACID属性

```
A: 原子性（Atomicity）
    事务中的操作要么都不做，要么就全做。
C: 一致性（Consistency）
    事务执行的结果必须是从数据库从一个一致性状态转换到另一个一致性状态。
I: 隔离性（Isolation）
    一个事务的执行不能被其他事务干扰
D: 持久性（Durability）
    一个事务一旦提交，它对数据库中数据的改变就应该是永久性的
```

事务运行的三种模式：

```
1.自动提交事务
    每条单独的语句都是一个事务，每个语句都隐含一个commit
2.显式事务
    以begin transaction 开始，以commit 或 rollback 结束。
3.隐性事务
    在前一个事务完成时，新事务隐式启动，但每个事务仍以commit或rollback显示结束
```

3. MySQL为什么使用InnoDB作为默认引擎？：前言

mysql为什么采用innodb做默认引擎？_大米饭-CSDN博客

MyISAM的索引与行记录是分开存储的，叫做**非聚集索引**（UnClustered Index）

其主键索引与普通索引没有本质差异：

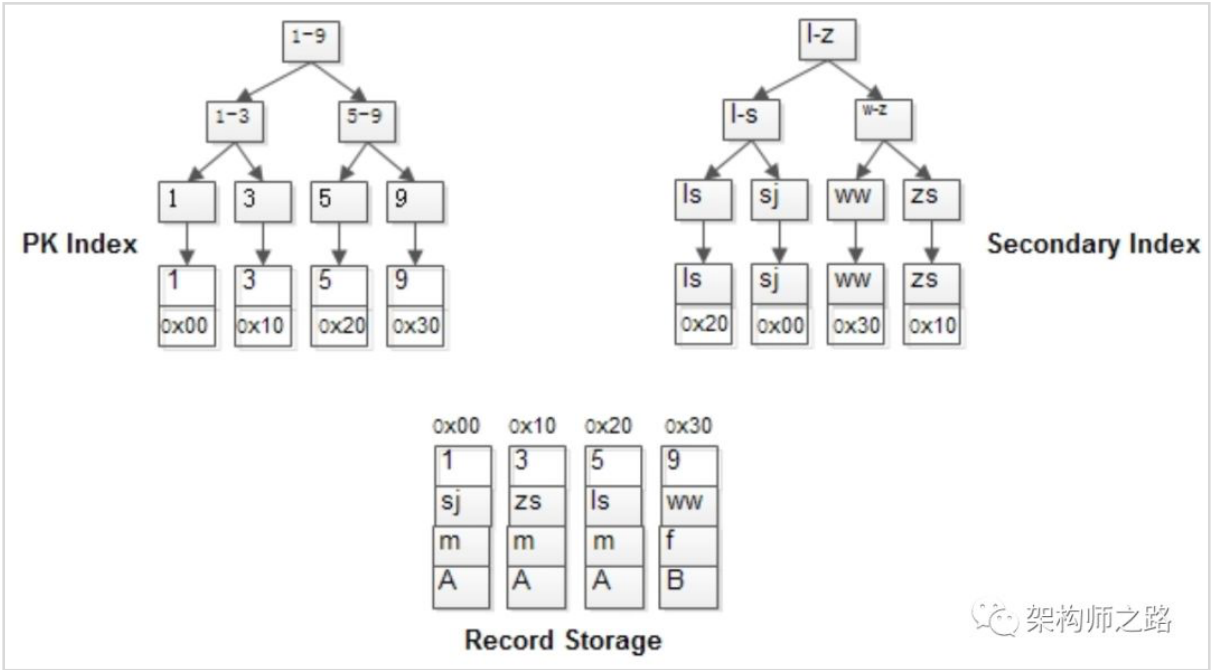
有连续聚集的区域单独存储行记录

主键索引的叶子节点，存储主键，与对应行记录的指针

普通索引的叶子结点，存储索引列，与对应行记录的指针

画外音：*MyISAM*的表可以没有主键

主键索引与普通索引是两棵独立的索引B+树，通过索引列查找时，先定位到B+树的叶子节点，再通过指针定位到行记录



InnoDB的**主键索引与**行记录是存储在一起的，故叫做**聚集索引**（Clustered Index）：

没有单独区域存储行记录

主键索引的叶子节点，存储主键，与对应行记录（而不是指针）

画外音：因此，*InnoDB*的*PK*查询是非常快的

因为这个特性，InnoDB的表必须要有聚集索引：

(1) 如果表定义了PK，则PK就是聚集索引；

(2) 如果表没有定义PK，则第一个非空unique列是聚集索引；

(3) 否则，InnoDB会创建一个隐藏的row-id作为聚集索引；

聚集索引，也只能够有一个，因为数据行在物理磁盘上只能有一份聚集存储

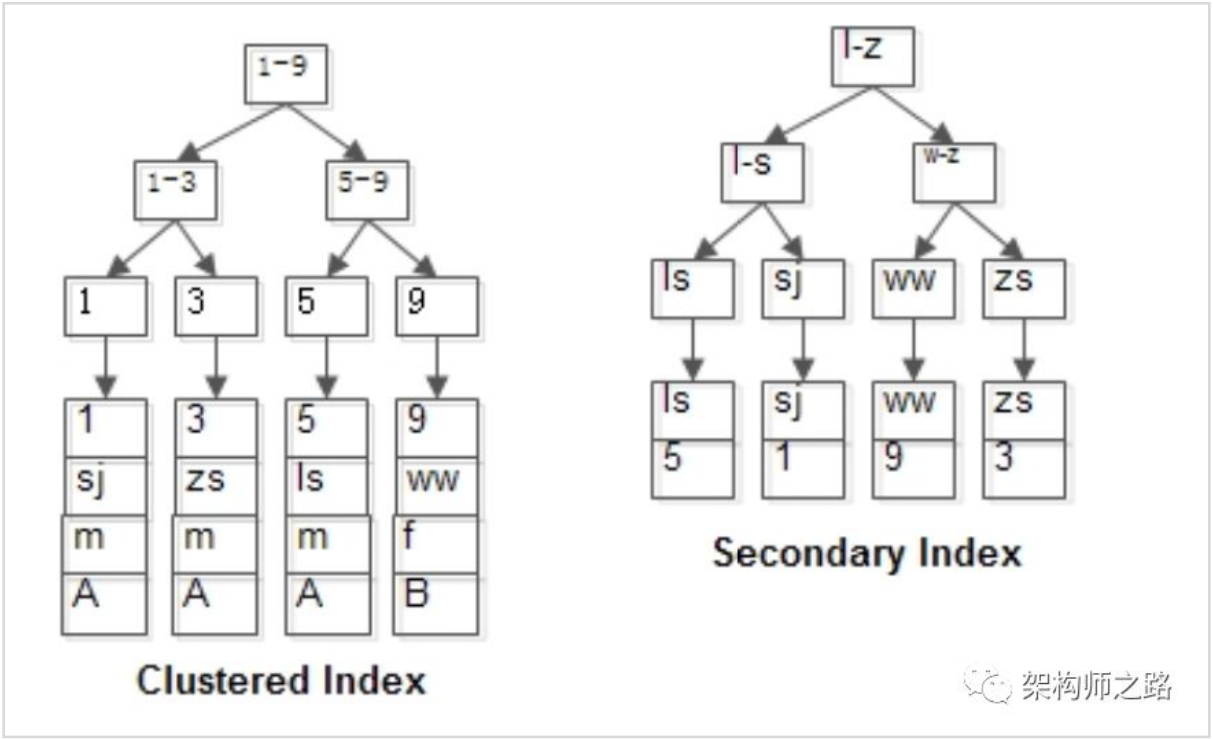
InnoDB的**普通索引**可以有多个，它与聚集索引是不同的：

普通索引的叶子节点，存储主键（也不是指针）

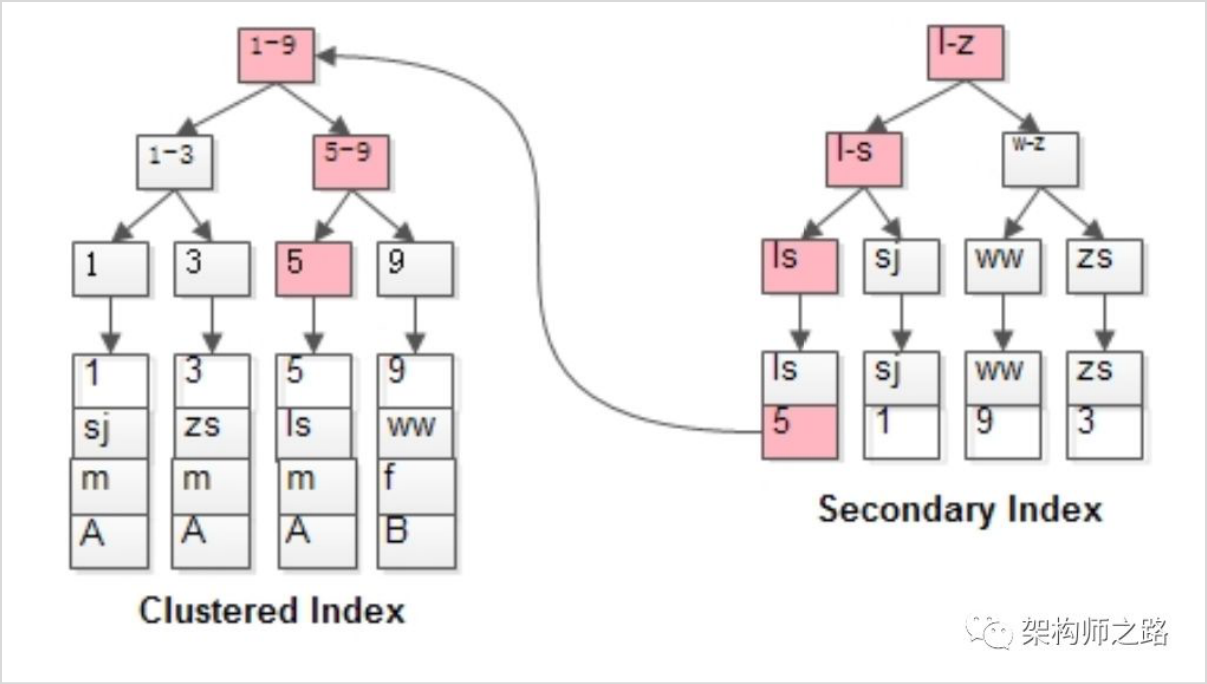
对于InnoDB表，这里的启示是：

(1) 不建议使用较长的列做主键，例如char(64)，因为所有的普通索引都会存储主键，会导致普通索引过于庞大；

(2) 建议使用趋势递增的key做主键，由于数据行与索引一体，这样不至于插入记录时，有大量索引分裂，行记录移动；



select * from t where name='lisi';



会先通过name辅助索引定位到B+树的叶子节点得到id=5，再通过聚集索引定位到行记录。
画外音：所以，其实扫了2遍索引树。

总结：
MyISAM和InnoDB都使用B+树来实现索引：

MyISAM的索引与数据分开存储
MyISAM的索引叶子存储指针，主键索引与普通索引无太大区别
InnoDB的**聚集索引**和数据行统一存储
InnoDB的聚集索引存储数据行本身，**普通索引**存储主键
InnoDB一定有且只有一个聚集索引
InnoDB建议使用趋势递增整数作为PK，而不宜使用较长的列作为PK

MySQL为什么使用InnoDB作为默认引擎?

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）（虽然*MySIAM*强调的是性能，执行速度更快）

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. 如果表中绝大多数都是读查询（有人总结出 读:写比率大于100:1），可以考虑 MyISAM，如果既有读又有写，而且也挺频繁，请使用 InnoDB

4. MyISAM 不支持崩溃后的安全恢复，系统崩溃后比较难恢复；而Innodb支持崩溃后的安全恢复（可利用事务日志进行数据恢复）

5. 锁级别为行锁，行锁优点是适用于高并发的频繁表修改，高并发是性能优于 MyISAM。缺点是系统消耗较大

> MySQL中MyISAM与InnoDB区别及选择 - Lucky小黄人^_^ - 博客园
> Innodb引擎的其他特点：
> （1） 索引不仅缓存自身，也缓存数据，相比 MyISAM 需要更大的内存
> （2） 因为它没有保存表的行数，当使用COUNT统计时会扫描全表（缺点）
>
> MyISAM的优点：
> （1）高性能读取
> （2）因为它保存了表的行数，当使用COUNT统计时不会扫描全表；
>
> MyISAM的缺点：
> （1）锁级别为表锁，表锁优点是开销小，加锁快；缺点是锁粒度大，发生锁冲动概率较高，容纳并发能力低，这个引擎适合查询为主的业务
> （2）此引擎不支持事务，也不支持外键。
> （3）INSERT和UPDATE操作需要锁定整个表；
> （4）它存储表的行数，于是SELECT COUNT(*) FROM TABLE时只需要直接读取已经保存好的值而不需要进行全表扫描。
>
> MyISAM适合：(1) 做很多count 的计算；(2)插入不频繁，查询非常频繁；(3)没有事务。
> InnoDB适合：(1) 可靠性要求比较高，或者要求事务；(2) 表更新和查询都相当的频繁，并且表锁定的机会比较大的情况。

4. 简述乐观锁以及悲观锁的区别以及使用场景

简述乐观锁和悲观锁的区别

- 乐观锁：

  乐观锁对加锁持有一种乐观的态度，每次拿数据的时候都认为别人不会修改，先进行业务操作，等到了最后一步更新数据的时候再进行加锁，并在更新的时候判断一下在此期间有没有人去更新这个数据

- 悲观锁：

  悲观锁对数据加锁持有一种悲观的态度，每次拿去拿数据的时候都认为别人会修改，

因此在整个数据处理过程中，都将数据处于锁定状态

悲观锁的实现，往往依靠数据库提供的锁机制（也只有数据库层提供的锁机制才能真正保证数据访问的排他性，否则，即使在本系统中实现了加锁机制，也无法保证外部系统不会修改数据）

使用场景

- 乐观锁的使用场景：

乐观锁比较适合**读取操作比较频繁的场景**，但如果出现大量的写入操作，数据发生冲突的可能性就会增大，为了保证数据的一致性，应用层需要不断的重新获取数据，这样会增加大量的查询操作，降低了系统的吞吐量，所以多写的场景喜爱使用悲观锁更合适

- 悲观锁的使用场景

悲观锁比较适合**写入操作比较频繁的场景**，如果出现大量的读取操作，每次读取的时候都会进行加锁，这样会增加大量的锁的开销，降低了系统的吞吐量

> **读取频繁使用乐观锁，写入频繁使用悲观锁。乐观锁不能解决脏读的问题。**

5. 什么情况下会发生死锁，如何解决死锁?

> 面试官：什么是死锁？如何解决死锁？写一段死锁的代码吧！_CG国斌的博客-CSDN博客
> 面试官问我知不知道 MySQL 的锁，接下来的 15 分钟让他刮目相看 - 知乎
> MySQL的锁到底有多少内容？再和腾讯大佬的技术面谈，我还是小看锁了！ - SegmentFault 思否

延伸：什么是死锁?

死锁是指两个或两个以上的进程在执行过程中，由于**竞争资源**或者**由于彼此通信**而造成的一种阻塞的现象，若无外力作用，它们都将无法推进下去。此时称系统处于死锁状态或系统产生了死锁，这些永远在互相等待的进程称为死锁进程

> 在数据库中有两种基本的锁类型：排它锁（Exclusive Locks，即X锁）和共享锁（Share Locks，即S锁）
>
> 当数据对象被加上**排它锁**时，其他的事务**不能对它读取和修改**
>
> 加了**共享锁**的数据对象**可以被其他事务读取，但不能修改**
>
> 数据库利用这两种基本的锁类型来对数据库的事务进行并发控制

延伸：产生死锁的原因?

- 系统资源不足
- 程序执行的顺序有问题
- 资源分配不当等

> 如果系统资源充足，进程的资源请求都能够得到满足，那么死锁出现的可能性就很低；否则，就会因争夺有限的资源而陷入死锁
> 其次，程序执行的顺序与速度不同，也可能产生死锁

产生死锁的四个必要条件：

- 互斥条件：一个资源每次只能被一个进程使用。
- 请求与保持条件：一个进程因请求资源而阻塞时，对已获得的资源保持不放。
- 不剥夺条件：进程已获得的资源，在未使用完之前，不能强行剥夺。
- 循环等待条件：若干进程之间形成一种头尾相接的循环等待资源关系。

> 这四个条件是死锁的必要条件，只要系统发生死锁，这些条件必然成立，而只要上述条件之一不满足，就不会发生死锁

什么情况下会发生死锁，怎么解决？： 常见的死锁情况，如何解决

1. 事务之间 资源访问顺序 的交替

- 出现原因：

一个用户A 访问表A（锁住了表A），然后又访问表B；另一个用户B 访问表B（锁住了表B），然后企图访问表A；这时用户A由于用户B已经锁住表B，它必须等待用户B释放表B才能继续，同样用户B要等用户A释放表A才能继续，这就死锁就产生了

- 解决方法：

这种死锁比较常见，是由于程序的BUG产生的，所以要通过调整程序的逻辑来解决。对于数据库的多表操作时，尽量按照相同的顺序进行处理，尽量避免同时锁定两个资源。如操作A和B两张表时，总是按先A后B的顺序处理， 必须同时锁定两个资源时，要保证在任何时刻都应该按照相同的顺序来锁定资源

2. 并发 修改 同一 记录

- 出现原因：

用户A查询一条纪录，然后修改该条纪录；这时用户B修改该条纪录，这时用户A的事务里锁的性质由查询的共享锁企图上升到独占锁（A要修改）；但是用户B（在修改）的独占锁，因为A的共享锁的存在，所以必须等A释放掉共享锁才能释放。而A由于B的独占锁的存在，无法上升到独占锁，所以也就一直没有释放自己的共享锁，于是出现了死锁。这种死锁由于比较隐蔽，但在稍大点的项目中经常发生

- 解决方法：
  - 使用乐观锁进行控制：

乐观锁大多是基于数据版本（Version）记录机制实现。即为数据增加一个版本标识，在基于数据库表的版本解决方案中，一般是通过为数据库表增加一个“version”字段来实现。读取出数据时，将此版本号一同读出，之后更新时，对此版本号加一。此时，将提交数据的版本数据与数据库表对应记录的当前版本信息进行比对，如果提交的数据版本号大于数据库表当前版本号，则予以更新，否则认为是过期数据。乐观锁机制避免了长事务中的数据库加锁开销（用户A和用户B操作过程中，都没有对数据库数据加锁），大大提升了大并发量下的系统整体性能表现

但需要注意的是，由于乐观锁机制是在我们的系统中实现，来自外部系统的用户更新操作不受我们系统的控制，因此可能会造成脏数据被更新到数据库中

  - 使用悲观锁进行控制：

悲观锁大多数情况下依靠数据库的锁机制实现，如Oracle的Select … for update语句，以保证操作最大程度的独占性。但随之而来的就是数据库性能的大量开销，特别是对长事务而言，这样的开销往往无法承受。如果面对成百上千个并发，这样的情况将导致灾难性的后果。所以，采用悲观锁进行控制时一定要考虑清楚。

3. 索引不当导致全表扫描：

- 出现原因：

如果在事务中**执行了一条不满足条件的语句，执行全表扫描**，把**行级锁**上升为**表级锁**，多个这样的事务执行后，就很容易产生死锁和阻塞。类似的情况还有**当表中的数据量非常庞大而索引建的过少或不合适的时候，使得经常发生全表扫描，最终应用系统会越来越慢，最终发生阻塞或死锁**

- 解决方法：

SQL语句中不要使用太复杂的关联多表的查询；使用“执行计划”对SQL语句进行分析，对于有全表扫描的SQL语句，建立相应的索引进行优化

4. 事务封锁范围大且互相等待

并发插入引发的死锁问题排查_zhoxing-CSDN博客

- 出现原因：

在同一数据库中并发执行多个需要长时间运行的事务时通常会发生死锁。事务运行时间越长，其持有排它锁或更新锁的时间也就越长，从而堵塞了其它活动并可能导致死锁

- 解决方法：
  - 保持事务简短并在一个批处理中：

    保持事务在一个批处理中，可以最小化事务的网络通信往返量，减少完成事务可能的延迟并释放锁
  - 使用低隔离级别

    确定事务是否能在更低的隔离级别上运行。使用较低的隔离级别（例如提交读）而不使用较高的隔离级别（例如可串行读）可以缩短持有共享锁的时间，从而降低了锁定争夺

6. 简述脏读和幻读的发生场景，InnoDB 是如何解决幻读的?