概述

一、简介

MySQL是开源，免费的关系型数据库，默认端口号3306

二、底层结构

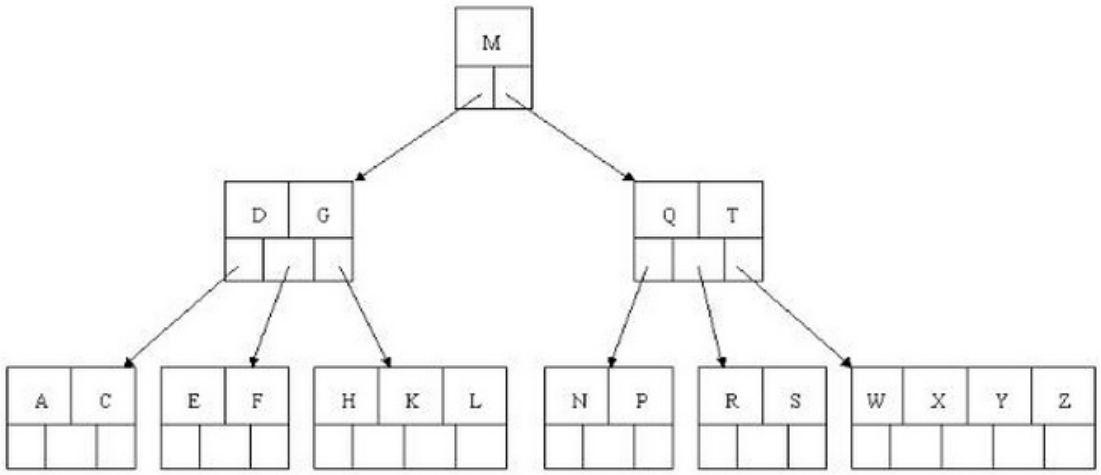
1. 什么是B树 B+树

1.1 B树

- 相当于多叉树，每个节点可以有多个数据，每个节点可以有多个分叉

> 非叶子结点的子节点数`> 1` `<= M` M为B树的阶数，代表每个节点最多有几个子节点

- **所有叶子节点位于同一层**
- 每层元素**从小到大排列**（还要遵守左小右大）



> 充分利用磁盘块（4K，IO可以将整个磁盘块读出来）将节点大小限制在充分利用磁盘块大小的范围

1.2 B+树

B树的优化版

- 非叶子节点不存储**指针**，只保存**索引**（使得一个磁盘块能保存的索引数大大增加，m上限提高，树高下降）（B树每个节点都有**索引**和**指针**）
- 只有叶子节点保存了指向数据的指针
- 叶子结点关键字从小到大排序，相邻叶子结点从左到右有指针相连

优点：

- **速度稳定**：因为数据都在叶子结点，叶子结点在同一层，所以每次查找所花费时间相同
- 树高更低，磁盘IO次数减少
- **全数据遍历更快**只需要遍历叶子结点（叶子结点相连），不需要遍历整个树

缺点：

- 如果经常访问的数据在B树中，离根节点近，则访问速度可能超过B+树

2. 为什么使用 B+树

2.1 如果用其他的

哈希表：

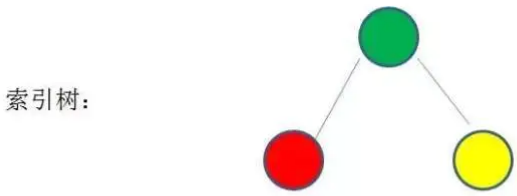
- `select * from sanguo where name='鸡蛋'` 顺利查到
- `select * from sanguo where name>'鸡蛋'` **范围查询**无法实现
- 适合只有k v的情况，例如redis

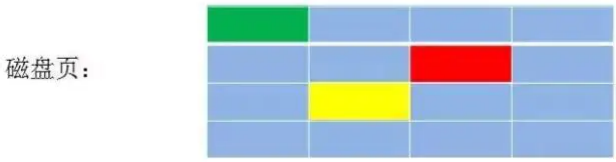
有序数组：

- 增删不方便
- 适合存储不会改变的数据，比如说2019年支付宝账单

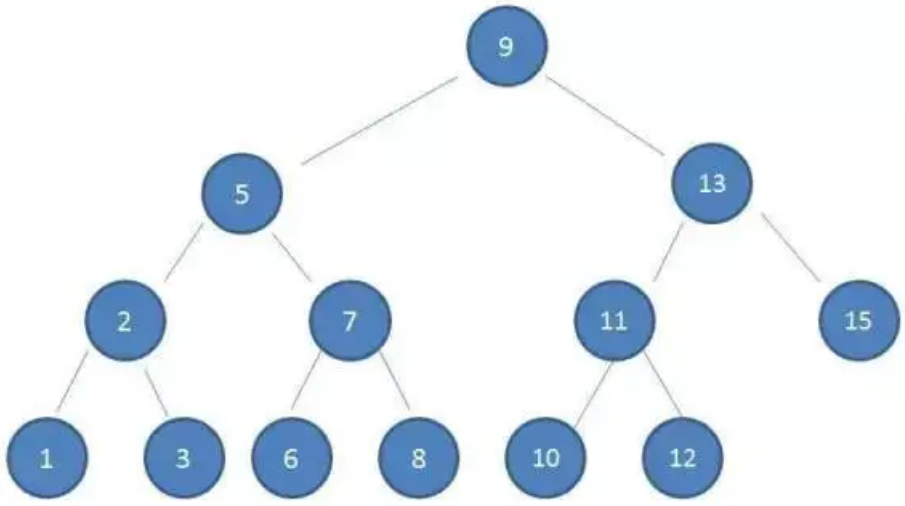
二叉树：

> - 数据库利用索引查找数据（类似于指针，指针可以找到数据，数据存在磁盘中）
> - 数据量很大时，**索引可能很大**(几个G)，所以索引不可能都在内存，也会存在磁盘中
> - 我们查找数据时，如果索引不在内存中，我们需要逐一加载磁盘页去查找
> - 假设我们的索引以**二叉搜索树**存储





> 假设我们从下面的二叉树索引中查找索引10



> - 将9读取到内存，索引10 > 9，所以向右找到13（13这个索引所在的磁盘块可能不在内存中）
> - 将13读取到内存，重复上面步骤，直至找到10
> - 总共进行了4次IO —— **IO次数是树的最大深度**

- 数据量很大时，二叉树深度很大，查询一个数据需要进行多次io，效率低
- 我们需要寻找一个方法**降低最大深度**（B+树）

索引

一、 什么是索引，为什么使用索引

1. 什么是索引

- 索引是一种为了快速查询数据而设计的数据结构
- 常见的索引结构有B树，B+树，Hash

2. 为什么使用索引

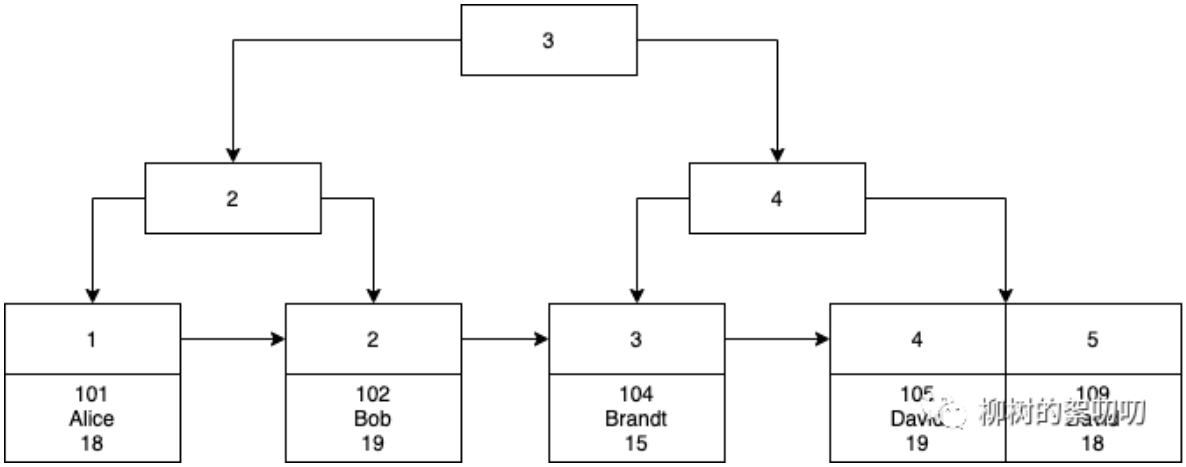
- 通过创建唯一索引，保证每一行数据的唯一性
- 减少数据搜索量，加快检索速度
- 随机IO变为顺序IO
- 但是创建和维护索引需要消耗时间和资源，所以不能无脑创建索引

二、索引的分类

1. 主键索引

- 一张数据表只有一个主键，且不为null（即使没指定，数据库会先找有没有唯一索引，没有的话数据库帮你建一个自增索引）
- **select * from** student **where** id **=** 5;

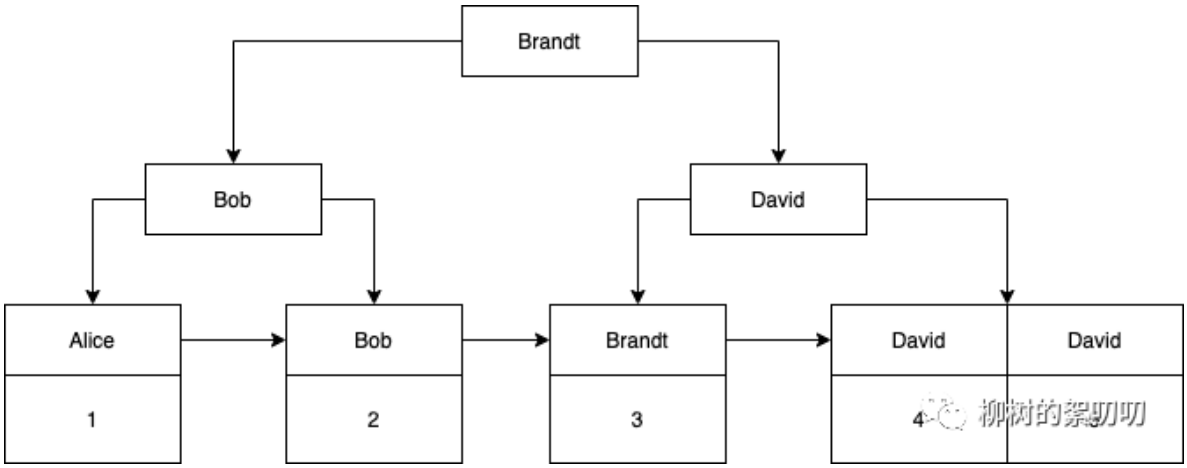
> 先找到3，5比3大，找右节点
>
> 5比4大，找右节点
>
> 叶子结点，递增数组，二分法查找



2. 辅助索引

- 如果你想按照姓名查找呢？没有索引只能全表扫描，如果不想全表扫描

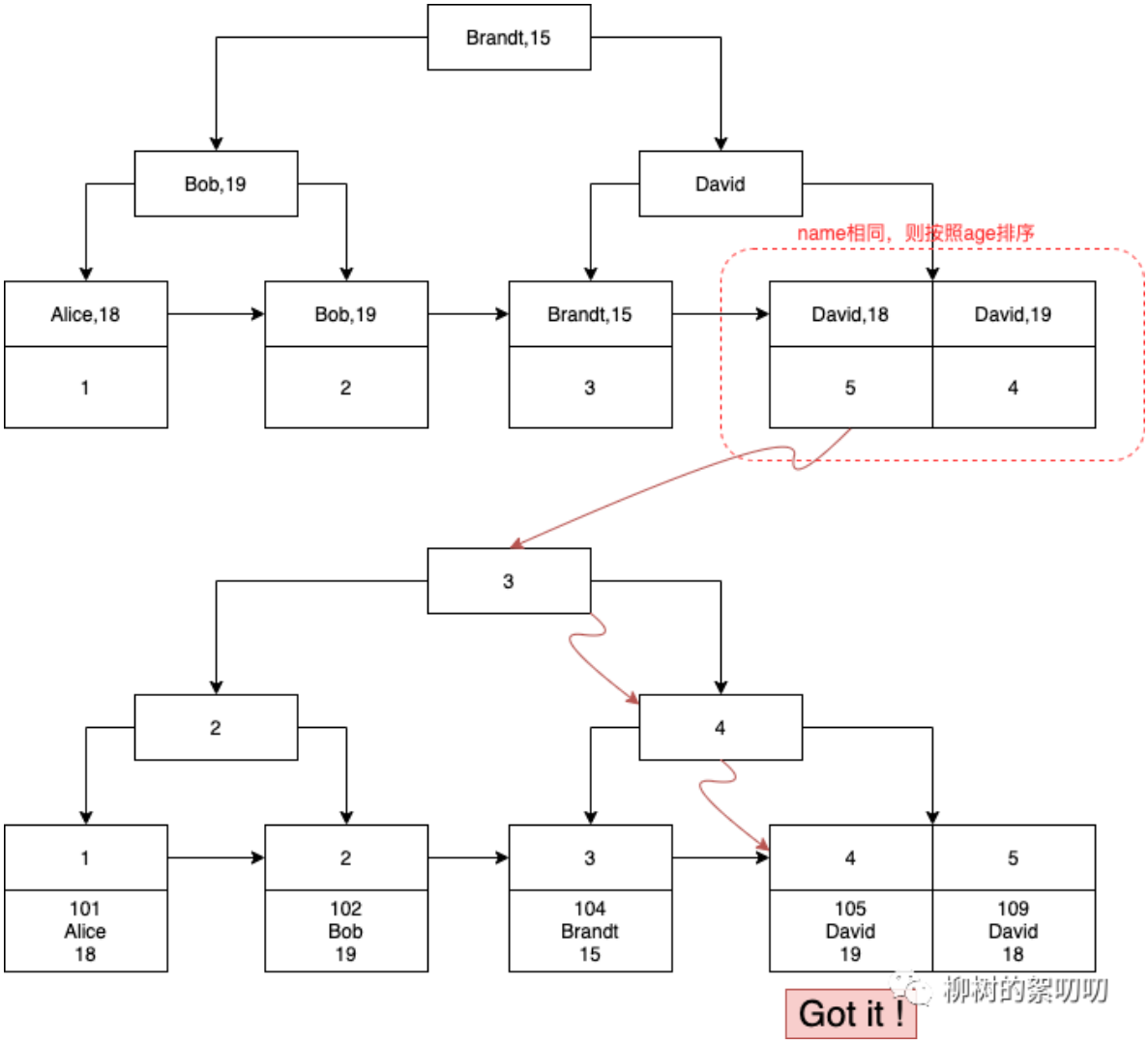
- **create index** idx_name **on** student(name); 创建索引，MySQL会生成一棵新的B + 树



- 叶子结点没有完整的数据，只有**主键id**和**姓名**

3. 复合索引

- 根据姓名和年龄同时查询
- **create index** idx_name_age **on** student(name,age); 继续新建一棵树
- **select * from** student **where** name = "David" **and** age = 18;



> - 我们建立索引时，name写在前面，所以先根据name大小建索引，name相同时再看年龄
> - 所以查询时我们也要将name放在前边（**最左匹配原则**），优先匹配

3.1 最左匹配原则

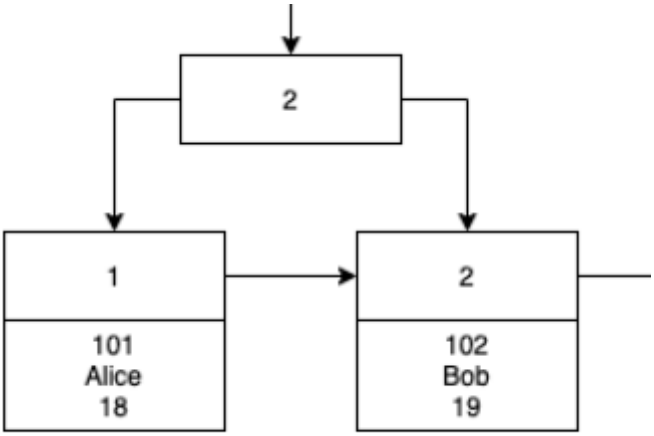
- 索引有多个列（a, b, c, d），key也由多个列组成
- 将**区分度**最高的索引放到最左侧
- 查询时，如果遇到范围查询（> < between like），停止匹配

> 如有索引 (a,b,c,d)，查询条件 a=1 and b=2 and c>3 and d=4，则会在每个节点依次命中a、b、c，无法命中d。(c已经是范围查询了，d肯定是排不了序了)

三、其他的一些索引

1. 聚簇索引

- 聚簇索引是指：叶子节点存放**完整的数据**（而不是指针）
- InnoDB的主键索引是聚簇索引



> 叶子结点存放着id，name，age

- **优点**：**查询速度快**，定位到叶子节点，就定位到了数据
- **缺点**：**依赖有序数据**，**更新代价大**

> - 如果索引乱序插入，叶子结点带有数据的话，排序插入速度慢（数据移动）
> - 如果索引列的数据修改，需要重新排序，修改代价大（所以一般主键不允许修改）
> - 所以**主键自增**的话，**顺序插入**，一页满直接放到下一页，避免了数据的移动，效率高

2. 非聚簇索引

- MyISAM是非聚簇索引，辅助索引也是非聚簇索引
- **优点**：更新代价小
- **缺点**：**回表**（查到对应的索引后，还需要根据指针再次查询主键表或者磁盘）

> 一定会回表吗?
>
> 不一定，只要查找的内容就是索引就不需要回表
>
> - `SELECT name FROM table WHERE username='guang19';` 如果name有索引，就不需要回表，直接返回
> - 同样的`SELECT id FROM table WHERE id=1;`也不需要回表，查找主键

3. 覆盖索引

- 一个索引包含所有需要查询的字段的值，我们称为覆盖索引
- 比如在上面的联合索引中`select age from student where name = 'Bob'`，不需要回表
- **减少搜索数据量**，提升性能，很多联合索引的创建就是为了支持覆盖索引，提高效率

4. 哈希索引

- 以O(1)速度查询，但是失去了有序性，不能排序，分组
- 无法范围查找，只能精确查找
- 底层是哈希表

> InnoDB会在某一个索引被频繁使用时，会在B+Tree索引上创建一个哈希索引

四、使用索引需要注意的点

1. 最左匹配原则

2. 索引不参与计算

3. 尽量扩展索引而不是新建索引

4. 覆盖索引

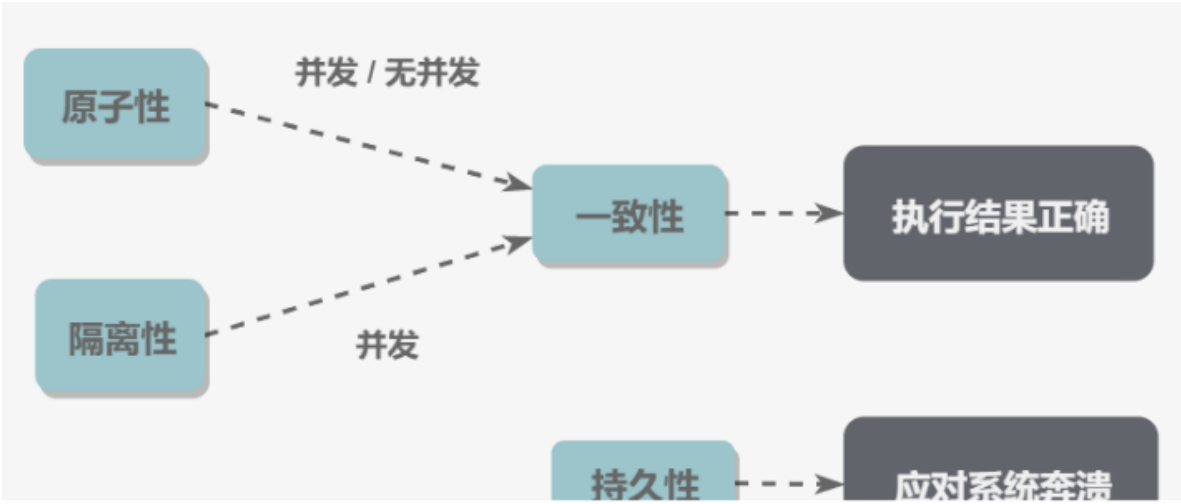
二、事务

1. 什么是事务

逻辑上的一组操作，要么都执行，要么都不执行

2. 四大特性（ACID）

- 原子性：不允许分割，要么都完成，要么都不执行
- 一致性： 事务前后数据保持一致，多个事务对同一个数据读取结果一样
- 隔离性： 事务独立，不受其他事务影响（一个事务所做的修改在最终提交以前，对其它事务是不可见的。）
- 持久性： 事务对数据库的改变是持久的，即使崩溃事务执行的结果也不能丢失



3. 并发事务带来的问题

- 脏读：事务A修改数据到一半，还没提交，事务B就读取数据，这个数据就是**脏数据**
- 丢失修改：事务A B先后对数据修改，A修改还没提交，B又修改了，A的修改就丢失了（隔离性）
- 不可重复读：事务A连续读取数据两次，两次中间事务B**修改数据**，则两次读取的不一致（一致性）

- 幻读：事务A读取数据到一半，B又**添加或删除**了很多数据，A就发现数据凭空产生或消失

4. 事务隔离级别（解决并发事务带来的问题）

- **读取未提交**：可以读取未提交的数据——脏读，不可重复读，幻读
- **读取已提交**：可以读取已提交的数据——不可重复读，幻读
- **可重复读**：对同一字段多次读取都一致——幻读
- **可串行化**：最高隔离级别，消除并发，串行执行

5. 如何实现事务隔离

5.1 锁机制（机制）

MySQL提供两种粒度（级别）的锁：**行级锁**以及**表级锁**，默认使用行级锁

- 锁定的数据量越小，并发程度越高
- 锁定的数据量越小，系统开销越大（会使用更多的锁）

锁类型

读写锁

加了X锁，就独享读写权限；加了S锁就可以读，其他人此时也可以加S锁来读，但不能加X锁来NTR。**行和表**都可以用X和S锁

- 互斥锁（Exclusive），简写为 X 锁，又称写锁。
- 共享锁（Shared），简写为 S 锁，又称读锁。

意向锁

因为有行级锁和表级锁存在，所以如果事务想对表加X锁，就需要扫描表上有没有锁，再扫描**每一项**有没有锁，过于耗时。引入IS IX锁，都是**表级锁**，目的就是标识这个**表**是否可以上X或S锁。

> 为什么叫意向锁，因为这只是一种意向，**并不是真正加锁**，例如A和B分别想对第1，2行加X锁，则都先对表加IX（IX/IS都相互兼容），都成功，然后对1,2行加X锁。如果C相对表加X锁，就要等到表上没有IX锁和IS锁才可以进行加锁。

- 对行加S锁前，必须对表加IS锁
- 对行加X锁前，必须对表加IX锁
- InnoDB自加，不须用户干预
- 其实并没有锁定任何资源，只是为了减少查询的次数

5.2 多版本并发控制 MVCC（具体实现）

MySQL的InnoDB引擎实现隔离级别的具体方式，实现了**读取已提交**和**可重复读**两种隔离级别。**可串行化**需要对所有读取的行加锁，MVCC无法实现

基本思想

- 实际使用数据库时，读操作多于写操作，读写锁，二者互斥
- MVCC利用多版本思想，读操作就在旧版快照（原来数据）上读，写操作（DELETE INSERT UPDATE）修改最新版快照，二者不再互斥
- 脏读和不可重复读本质都是A读到了B未提交的数据，为了解决，MVCC规定**只能读取已经提交的快照**

具体实现

- SELECT 不再需要加锁，因为读取的是旧版快照的数据
- DELETE INSERT UPDATE 仍需要加锁，为了保证修改的是最新版的快照
- MVCC只是避免了SELECT的加锁操作

5.3 Next-Key Locks（具体实现）

- Next-Key Locks 是 MySQL 的 InnoDB 存储引擎的一种锁实现。
- 在可重复读的条件下，通过MVCC + Next-Key Locks解决幻影读
- 需要除了主索引外还有

Record Locks

- 锁定一个记录上的索引，而不是记录本身
- 如果没有设置索引，则会自动在主键上创建聚簇索引

Gap Locks

- 锁定索引之间的间隙

Next-Key Locks

- Record Locks 和 Gap Locks 的结合，锁定一个记录上的索引，还锁定这个索引左右两侧的间隙
- 前开后闭
- 对于单一索引，退化为Record Locks，只有多索引查询才用这个

三、查询性能优化

MySQL数据项过多时性能下降，常见的优化措施：

1. 分析方法

使用explain分析， 几个关键的参数

- select_type: 查询的类型，简单查询，联合查询，子查询等
- key: 使用的索引
- rows: 扫描的行数

2. 从查询语句入手

2.1 限定数据范围，减少请求的数据量

- 查询近一个月的订单（限制行数）
- 尽量不用SELECT *，只返回必要的列（限制列数）
- 使用缓存

2.2 减少服务端扫描的行数

覆盖查询（参考2.3索引优化）

3. 从数据库结构入手

3.1 读写分离

主库负责写，从库负责读。使用代理服务器，请求来时，它决定转发到哪个服务器（主库/从库）

3.2 垂直分区

列太多的时候，拆成多个表或者多个数据库

- 优点：数据项数据减少，每个磁盘块放的数据变多，减少IO次数
- 缺点：主键冗余，查询变复杂

3.3 水平分区

拆分到多个相同结构的表中，需要有一个全局id

四、存储引擎

5.5版本之前，MyISAM是MySQL的默认数据库引擎，5.5版本之后引入InnoDB，事务性数据库索引，作为默认引擎。

MyISAM和InnoDB区别

- 是否支持行级锁： InnoDB支持，MyISAM不支持
- 是否支持事务：InnoDB支持，MyISAM不支持
- 是否支持外键：InnoDB支持，MyISAM不支持
- 是否支持MVCC： InnoDB支持
- MyISAM 非聚簇索引，InnoDB 聚簇索引
- InnoDB不支持全文索引，MyISAM支持

五、琐碎知识

1. delete和truncate 的区别

- delete更灵活，可以删除一行数据
- delete删除表示一行一行删除，truncate是删除表后重建
- delete可以回滚，truncate不可以
- delete后，主键延续，truncate主键重置

2. 悲观锁与乐观锁

- 悲观锁，总是假设最坏的情况，每次有人来拿数据，都假设他会修改数据，都必须上锁。比如说，读写锁，表级锁等
- 乐观锁，总是假设最好的情况，认为别人拿走数据不会修改，所以不加锁。