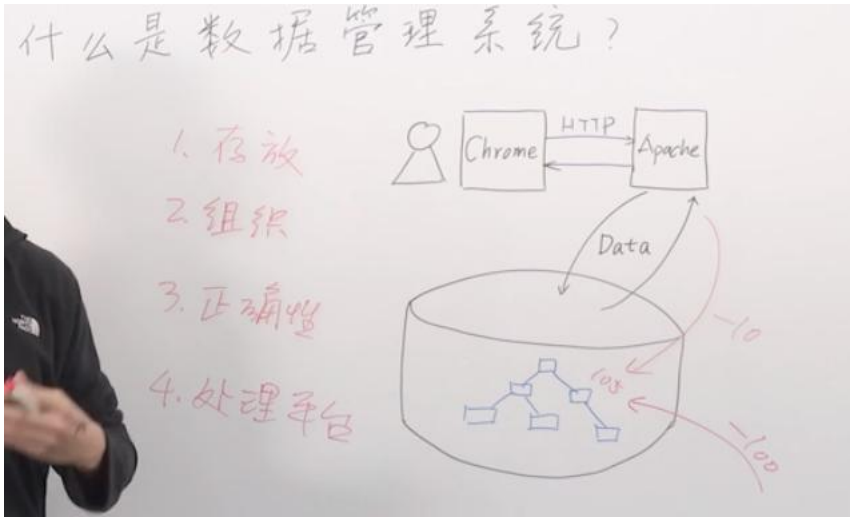
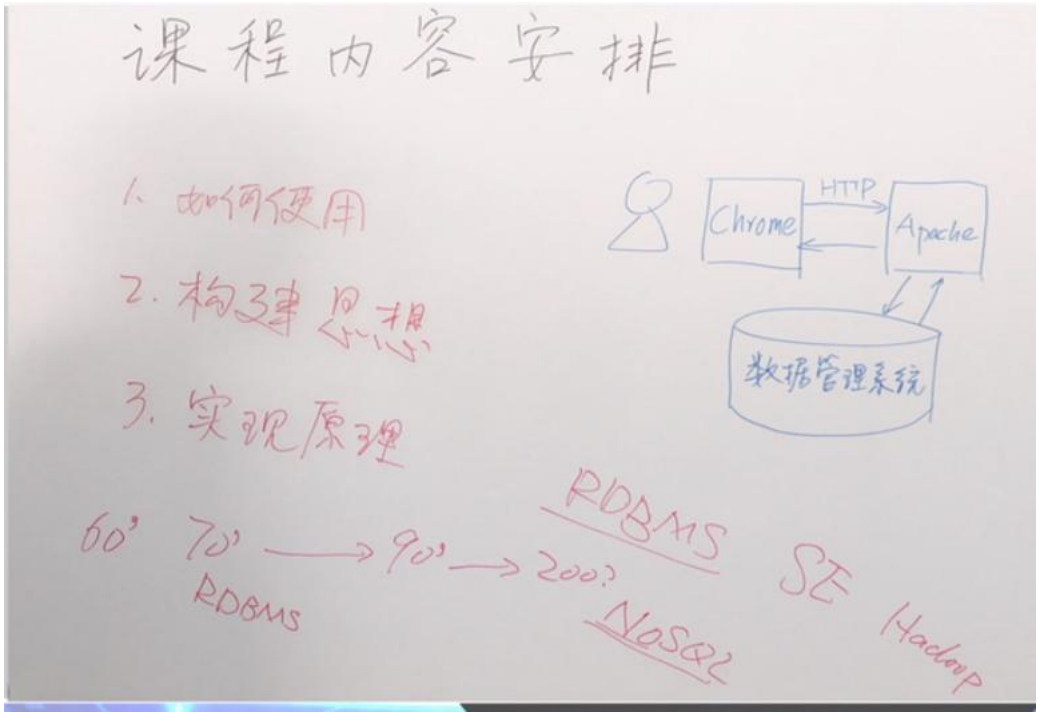
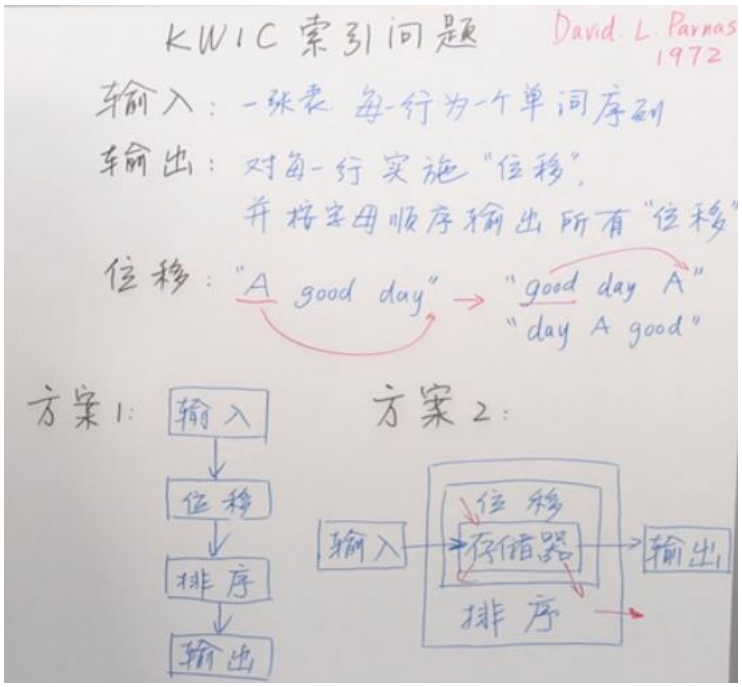
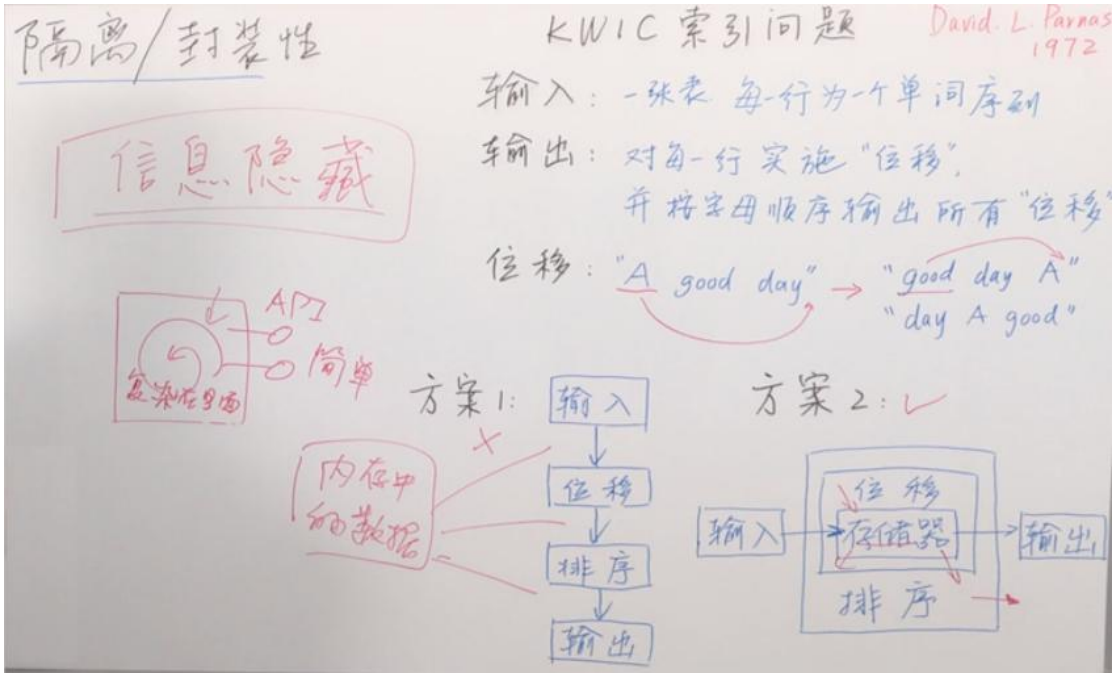


数据管理系统的四大功能：存放、组织、正确性和处理平台







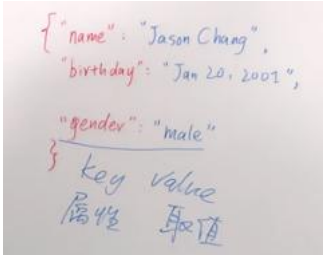
系统模块的来源是经验
库函数 library 调用
文件系统 数据管理系统 独立的系统

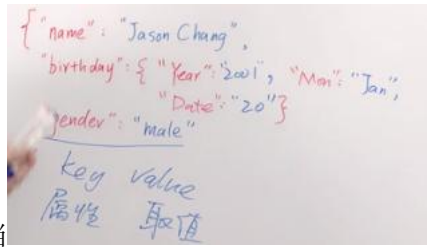
程序和系统的交互：
1.框架 framework：Spring Express 不同的前端设计语言都有其框架，很多应用通过一种模式构建，所以不同重复写，其实就是留空，空的东西决定逻辑
2.API（库）

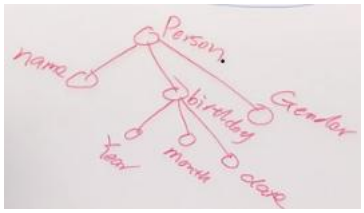
例 1.为什么是独立运行的系统，不是库的形式（相当于别人写好了一部分的程序，把它拉过

来去运行）？
复杂的功能不一定要变成系统，数据管理系统是要用各种硬件去处理这些数据，调用 cpu 去维护，去调度处理，保证数据完整，是要自己取管理 cpu，硬盘，要管理的话就不是 library，是随时随地； 多个数据要共享，被若干个共享的话必须是系统，没法协调，不能是 library ；如果只是嵌在应用中的一段程序，如果不设立屏障的话，容易被篡改，系统可以随时备份，保证本身的完好。
例 2：什么叫好的模块化？
并不是为了提高程序运行效率，基本目的是为了提高开发的效率，是人在制作应用程序的效率，模块之间的交互尽量简单，以至于模块外部看不到细节。
开发是把一件复杂事情变成一步步，分工需要模块化，做完一个再做一个
在维护程序的过程中，有人要去改，做的好的模块化有利于别人去理解，可读性会增加。
例 3：数据管理系统要变成好的模块应该满足的条件？
DBMS 职责明确，一件事情要么全部给他做，要么就不做
不明确的结果：程序和系统就有很多复杂交互，不利于模块化，如数据存进去就行了，而不用担心会不会丢失
数据访问简单，接口问题不是那么容易的，对于文档数据库与 SQL 数据库
尽量所有数据的交给 DBMS 管理
运行在独立的机器上是为了提高整个系统的可靠性，如集群

Week2:
基本概念：
数据库是 DBMS 的最早原型，数据放进去，需要时去查找，更新修改，不丢失，一致性和准确性
功能：CRUD
Create 创建数据项
Read 读取数据
Update 修改
Delete 删除
增删改查
OOP object oriented programming
数据看成对象 对象如何描述，如所有人放在一张表；每个人的信息打包在一起变成文档
描述对象的结构--数据模型 根据不同数据模型，便又不同系统，如关系、文档数据库系统

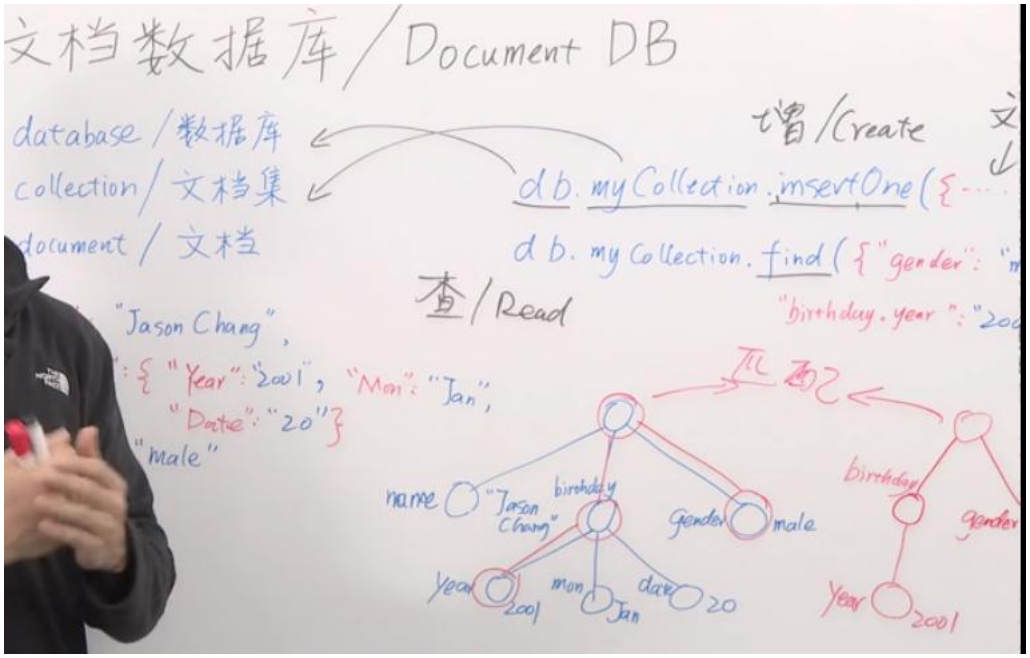
文档模型：
数据库，每一个库对应一个应用
Collection 文档集
Document 文档 

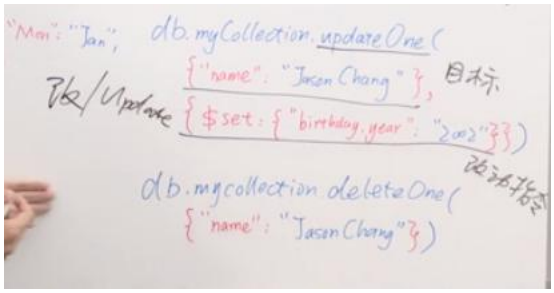
文档里面还可以嵌套文档 

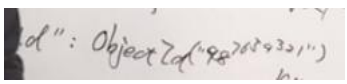
其实就是一个树的结构： 

每一个对象都是通过一个文档中的键值对来描述，同一类型的文档集合起来形成文档集，多个文档集被同一个应用去使用，即数据库。

MongoDB



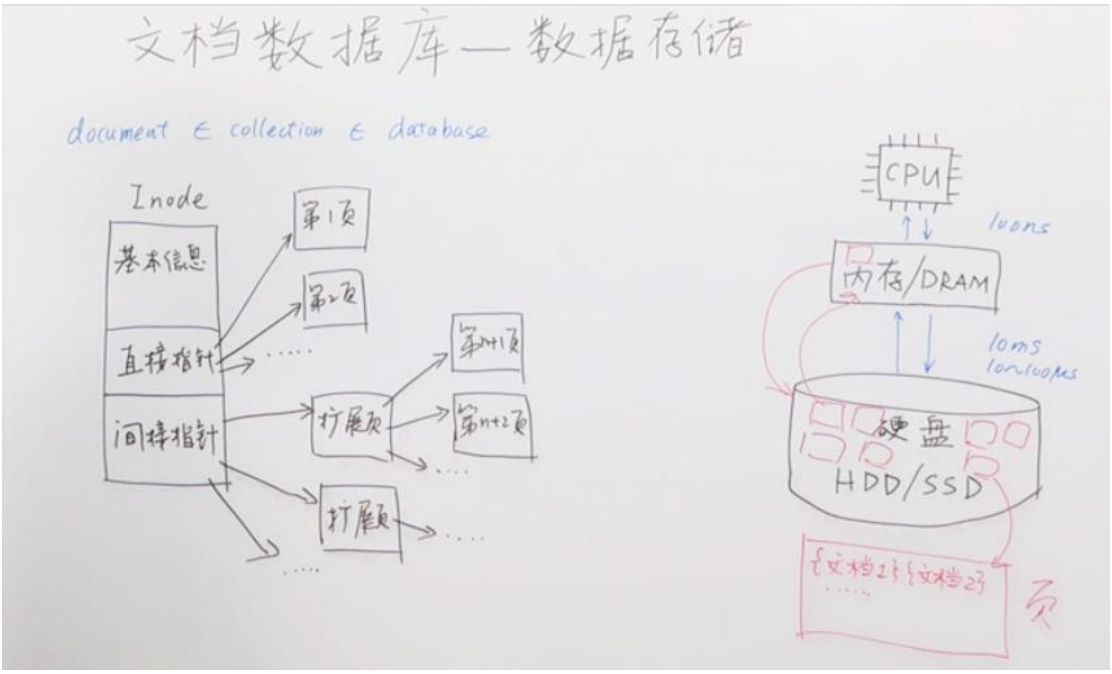


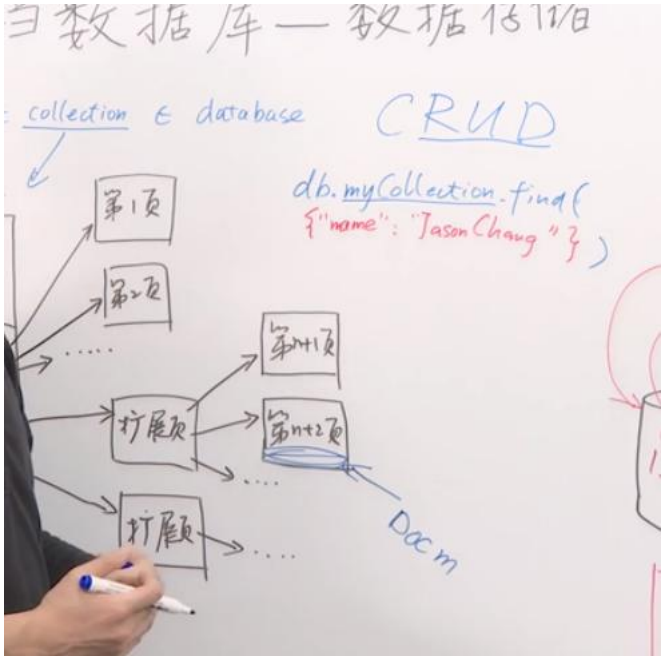


文档的存储

HHD 磁盘 SSD 闪存 内存必须带电才不会丢失，硬盘不会

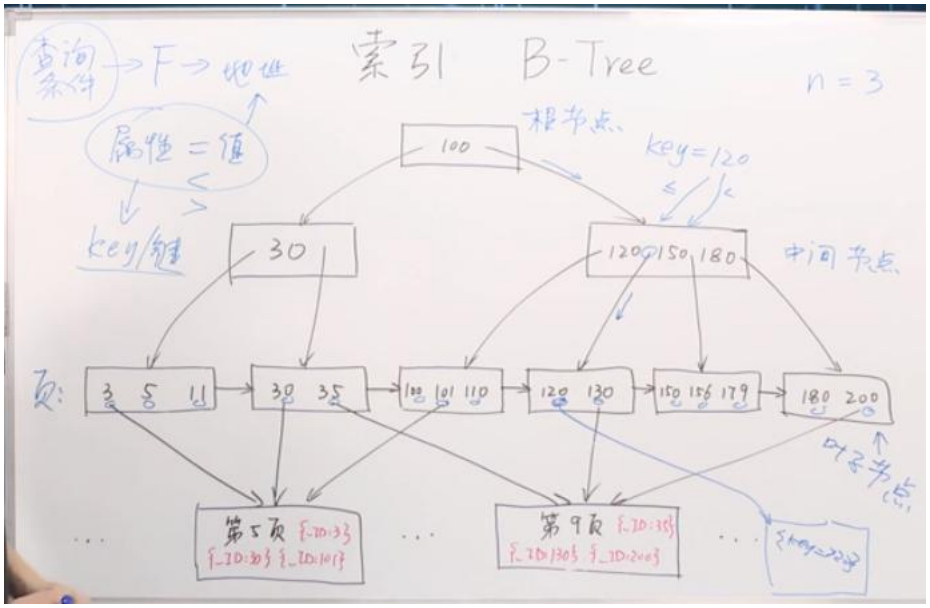

document
database
{ "_Id"
"name"
"bir
"g
}
CPU
内存/DRAM
硬盘
HDD/SSD


文档数据库—数据存储
document ∈ collection ∈ database
Inode
基本信息
直接指针
间接指针
CPU
内存/DRAM
硬盘
HDD/SSD

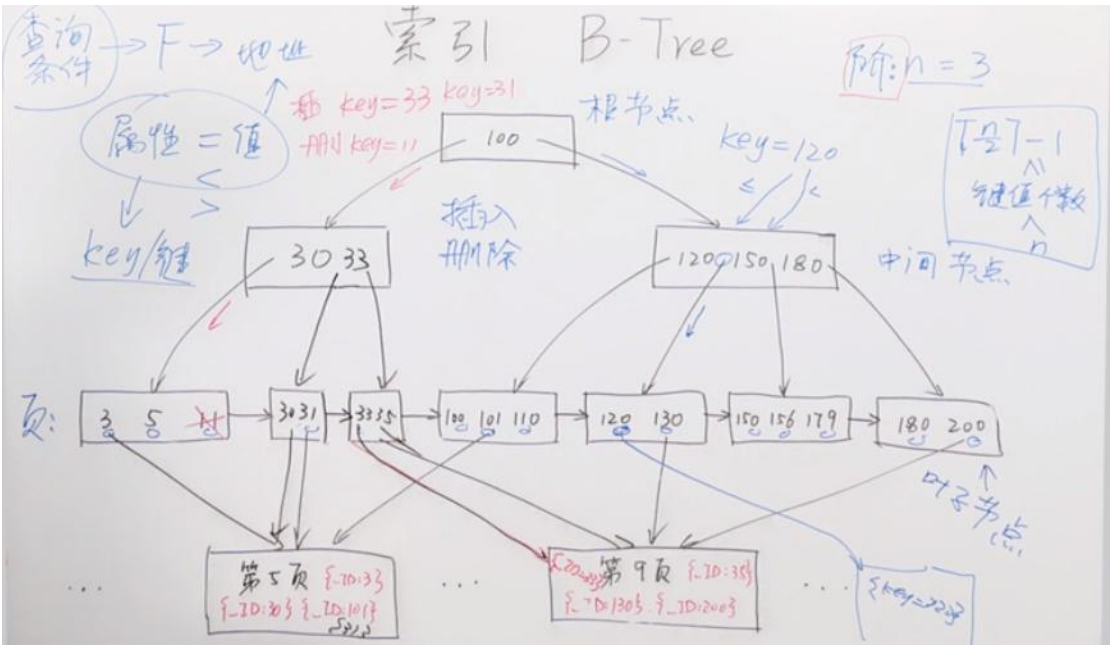


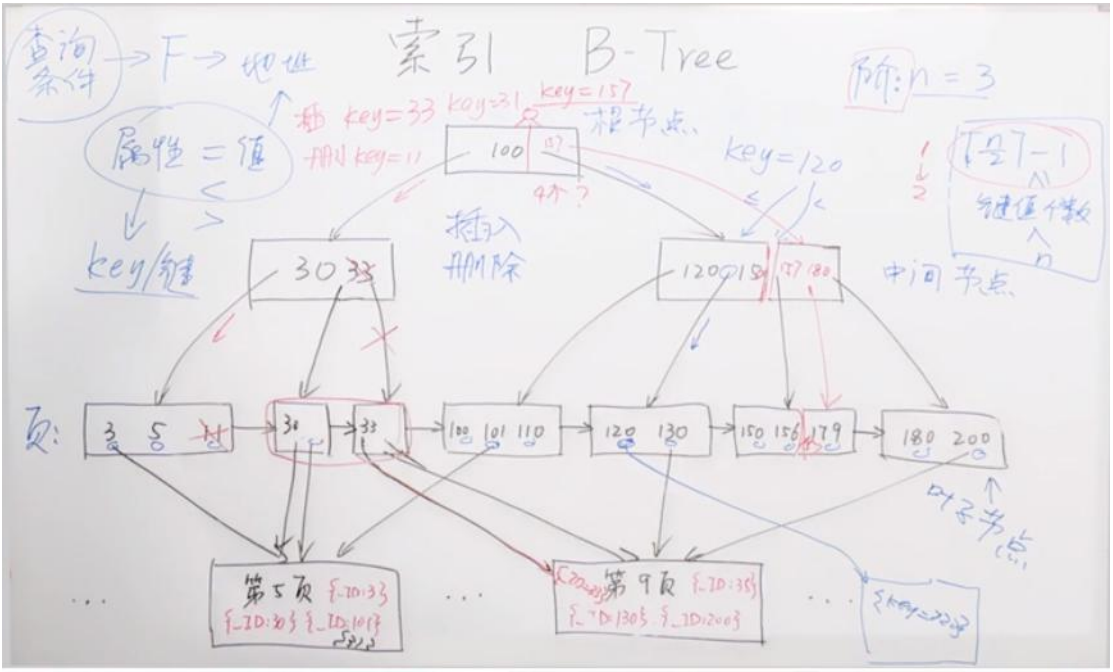
光页面堆积的结构对于创建友好，但是对于其他不友好，代价比较大，每次都要找到，扫描整个磁盘的空间，时间长。所以需要索引，可以看成函数，不用遍历

B 树索引，点查询，给了一个文档的属性=/>/<值，对一个单一的属性就可以用该结构。



任何一个结点的键值个数范围  n/2-1　　n 保证树的平衡，执行的效率



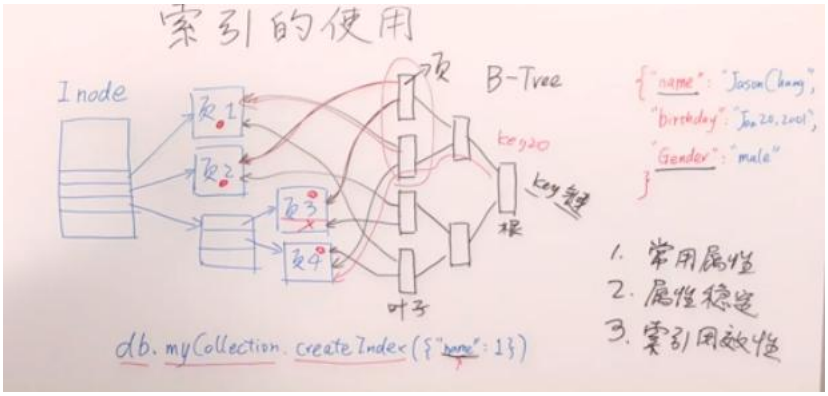


必须要用户自己指定要在哪一类文档创建哪一类索引，

并不需要为每一个属性都创建一个索引，索引是复杂的，所使用的存储空间可能和 collection 差不多；更新数据后需要更新索引。

1. 常用属性
2. 不常被修改，属性稳定，修改是先要删除再插入
3. 索引的有效性 如性别，一半

多值属性上的索引，关系数据库会讲



如果用 ID 作为索引，那么 inode 结构可以去掉，既可以用来访问，也可以用来串联。

第 1 题：一个文档数据库里面有两类对象，书（book）和人（person）。书和人之间存在一种写作的关系，即某个人是某本书的作者。请问这种写作关系的信息应该如何存放？
一个对象是另一个对象的父亲，用程序里就用引用 reference、指针（C）
引用属性必须要去唯一识别另外一个对象，名字没有办法去唯一识别一个对象，在文档数据库里有一个特殊的属性 ID，虽然这个值没有任何的意义
第 2 题：查阅 mongoDB 手册
第 3 题：对于程序开发者来说，程序写起来复杂；不是原子的

第 4 题：大部分系统都是对存储空间进行分页管理的。请问，分页模式的优势不包括

A：有利于减少存储空间的碎片化，提升空间利用率。

B：有利于提升数据访问的性能。 ✗

C：有利于提升内存缓存的效率。 ✓

D：有利于减少空间管理的成本（即减少空间管理对CPU和内存资源的消耗）。

1.碎片化，直接存储文档，插入再删除会逐渐产生一个个过小而无法利用的空洞
处理方法：回收 把洞给扒出来，数据塞在一起，空洞塞在一起，代价高，可能 80%的数据要移动
如果把空间分成页：如果空间很小，就不需要整，也就是局部化，整个管理的代价就会降低
2. 有利于提升数据访问的性能：连续去访问的性能比跳着去访问的性能要好很多？分页的有利于把要访问的数据放在同一页里
3. 不一定，缓存是用来把经常被访问的数据放在里面，比普通的访问快，但小，热数据放在缓存中，可能当页比较大时，有可能有一部分是热数据，大部分是冷数据，那么冷数据侵蚀了空间，所以不能。。
4. 有利于减少空间管理的成本，以页为单位，否则是要以数据为单位，大小不统一，会变得简单

第 5 题：课程中提到，内存由于比较昂贵且无法持久地保存数据，通常只作为数据缓存。那么，什么数据不适合被放在缓存中

A：经常被修改的数据

B：像Inode这样的组织结构数据 ✗

C：刚被插入的数据

D：刚被删除的数据 ✓

你预期接下来不会被再次使用
A. 但是还是要落实到硬盘，会断电，节省的代价会有限
经常去读的，但很少去改的，如 Inode
防误操作，会有一个缓存区，而不是缓存（是稀缺的资源，加速访问）

思考题 1：当我们对存储空间进行分页管理的时候，页的大小通常是一个设计要点。有的数据管理系统选择使用比较小的页，如 2KB 或 4KB。而另一些系统会使用比较大的页，比如 4MB 或 8MB。请问：小页面对什么情况有利？大页又对什么情况有利？我们确定页的大小时应该考虑哪些因素？
小页面：可以解决碎片的问题；大页面：有利于提升数据访问的性能
页面太大容易产生空间浪费，程序假如只使用了 1 个字节却被分配了 10M 的页面，这岂不是极大的浪费，页面太小会导致页表占用空间过大，所以页面需要折中选择合适的大小

思考题 2：如课程中提到的，内存通常被数据管理系统作为缓存使用。缓存的数据单元可以有不同的选择；可以是页，即当访问完一页后，将整个页继续保留在内存中，以期后面再次访问该页就无需再从硬盘获取；也可以是文档，即当访问完一页中的某个文档后，将这个文档继续保留在内存中，而将页移除，以期后面再次访问该文档时无需再从硬盘获取。请问：页缓存和文档缓存各自的优势和劣势是什么？什么情况下，我们可以考虑使用文档缓存
页缓存所占的内存空间会较小

1ns=10-9s　CPU 访问内存 100ns 即内存墙 cache 也分指令、数据的 cache
10ms 硬盘，大部分用闪存
105IOPS 读取 随机读与顺序读差异大
存储系统中最明显的差异就是缓存，就是在 ram 有没有命中，那什么时候容易在缓存命中？数据访问的局部性（挨着一个一个去访问即空间局部性）好比较容易在缓存上命中，尽量在金字塔上面，不用了再扔掉。

一页包含多个文档
文档放在缓存了，要单独组织这个文档，每个文档都有 ID，有利于减少缓存的空间，但管理成本高
如果是一个个挨着去访问，用页去做缓存是有效的

真正的应用系统都是面向对象的，把数据成一个个对象的信息
很多时候要找的对象就是一个，所以用 findone 就可以了
Update,delete 先找到，即查询条件，然后在做

内存管理
文件系统
两者之间有相似的地方，文件系统有一个存储空间管理
内存也有空间管理，C 语言的空间管理是显性的要 malloc，free 一个空间；使用面向对象的时候都对应一个个相应的空间　虚拟内存：对空间管理的设计
都分成一页一页，4KB，一般不会调
页表：对应于物理空间　一个指针就有 8byte，页表就会很大，页表是用来管理的，当页越小的时候，管理成本越高；页太大，就会形成浪费

Oracle 以 8KB 分页，可能有很多随机的访问
MongoDB　64KB 到 1/4MB 都是可以调的
64KB 而不是 8KB，
访问数据的性能变好
连续访问比跳着好，一个文档由很多页构成，这样页的个数变少
当页变大也会造成性能变差：
1. 访问的数据本来就很少，本身访问的内容就变多了，读（写）放大
2. 当页很大的时候要 cache 起来，不利于提高 cache 的利用率
当一次性要访问很多数据的时候，如播放电影，顺序连续访问，页大的时候是有利的，不用跳来跳去，但需要跳来跳去即随机访问，I/O 和 cache 的资源
文件系统一页是 64KB 整体的性能是最好的，所以可能提高页的大小。。

文件系统中一页一页，Linux 页大大小不一定 Ext3 4KB，可以调
有的文件系统：
分布式文件系统 HDFS 一个块（页）的大小 block　128MB，很大，由应用场景决定的，分布式是存储一个个巨大的文件，大片大片去读写，很连续的访问模式，整个页的调度成本降低，数据访问成本降低
为什么要分页？
如果没有一个个的页，是全局的，分页就可以局部，页的调度也比较方便

第 1 题：以下哪个因素不会显著影响B树的访问性能？

- A：树的高度
- B：树的阶
- C：节点的空间大小（通常一个节点为存储空间中的一页，因此可理解为页的大小）
- D：节点内部的数据充满度 ✓

第 2 题：B树的平衡性主要由哪条性质保证？

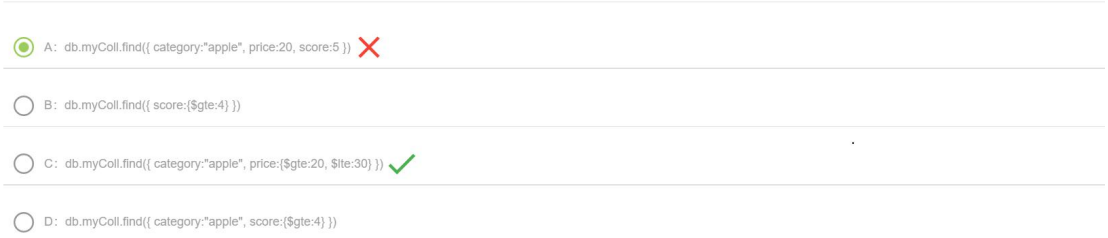
- A：每个节点的大小固定
- B：每个节点的充满度都超过1/2 ✓
- C：叶子节点上的数据是有序的
- D：以上性质都不能

第 3 题：如果我们在属性price上创建一个索引（比如使用指令db.myColl.createIndex({ price: 1 }) ），那么以下哪个查询可以无法从这个索引获益？

- A：db.myColl.findone({ category:"apple", price:20 })
- B：db.myColl.findone({ category:"apple" }) ✓
- C：db.myColl.findone({ price:{$gte:20, $lte:30} })
- D：db.myColl.findone({ category:"apple", price:{$gte:20, $lte:30} })

先找到满足 price 的例子，再去跟前面条件去对比

第 4 题：如果我在多个属性上创建一个复合索引，例如db.myColl.createIndex({ score: 1, price: 1, category: 1 })，那么以下哪个查询无法从索引获益?



Score price category 很多时候会用，适用范围更广
C 是可能散落在空间的不同地方，因为没有 score
D category 没办法用，price 不确定，因为 price 可能散落到任何一个点，但是使用到 score 就可以用到
A-B-C-D
有 A;AB;ABC 包含前缀就可以用，如果不包含，只是有后面的属性，如 BCD 就没办法用
比单独 A 好的原因：AB 更好？
代价：键值变长，k 变大，B 树里面能够容纳的个数变小，这个树高度会增加，查询的性能会恶化

C 是可以考虑的，但数据被改了都要对 a 操作
A 不带来很多要维护的代价
区分效果不好的如性别，几种颜色，洲
Selectivity=k/x>=1% 通常不需要创建索引，无效　属性是随机分布，除了 id

B 树相对于平衡二叉树，其好处为：一个结点的散出效果越大，高度越低 fan-out
如果是比较次数，B 树比 AVL 更大，把一棵树变矮，并不是为了减少计算量，而是减少了 I/O 的次数　每一次比较是 CPU 访问内存的代价，但是真正要减少的应该是内存到硬盘中取东西的代价，即 I/O 代价，每次加载的是一 block，最小也有 512byte，里面可以装很多数据
树的高度决定 I/O 代价，而这个是最大的代价，目的是为了减少 I/O 代价，每个结点尽量要装更多，每一个结点应该要把一页用得充分。
所以 AVL 的计算代价并不差，但访存的代价比较高

重要的是树的散出多少，影响散出的 ABC　一个结点用完一页，页的大小决定可以放多少指针
B 树的平衡性，阶 k->k/2-k 即指针（键值）的个数，这个性质决定

B 树
Hash　key->Address(key) 构建在硬盘上面
block 代价 N data k buckets　N/k 即访问桶的代价
点查询，确定 k 值，去查，无法应对范围查询

“##apple##”应该构建什么样的索引
R-Tree
LSM-Tree

用 id 查和用属性查 用很多数据