现代数据管理理论与技术考点

**课件1**

1.了解：大数据有4个v,各自代表什么含义

20页

业界对大数据归纳出4个层面的特点（4V）：

Volume：数据体量巨大（从TB级别，跃升到PB级别）；

Variety：数据类型繁多（网络日志、视频、图片、地理位置信息等等）；

Velocity：处理速度快（1秒定律，有别于传统的数据挖掘技术）；

Value：价值密度低。

2.大数据中超大规模的图计算，适合跑在什么平台上

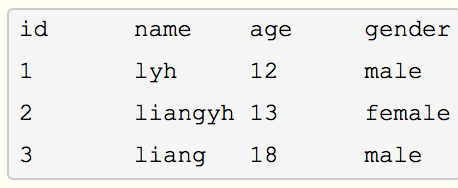
42页

一个高性能的NOSQL图形数据库，是一个嵌入式的、基于磁盘的Java持久化引擎，支持完整的事务。**Neo4j**提供了大规模可扩展性，在一台机器上可以处理数十亿节点/关系/属性的图，可以扩展到多台机器并行运行。

相对于关系数据库来说，图数据库善于处理大量复杂、互连接、低结构化的数据，Neo4j重点解决了拥有大量连接的传统RDBMS在查询时出现的性能衰退问题。（这些数据的频繁查询在关系数据库中会导致大量的表连接，因此会产生性能上的问题。）Neo4j提供了高效的图搜索和图遍历算法（例如查找二度人脉、甚至三度人脉及多度人脉，查找两个用户间最短路径），这在目前的RDBMS系统中都是难以实现的。

3.结构化与非结构化？以及典型的例子

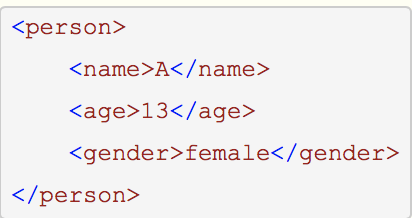
**结构化的数据**是指可以使用关系型数据库表示和存储，表现为二维形式的数据。一般特点是：数据以行为单位，一行数据表示一个实体的信息，每一列数据的属性是相同的。



**半结构化数据**是结构化数据的一种形式，它并不符合关系型数据库或其他数据表的形式关联起来的数据模型结构，但包含相关标记，用来分隔语义元素以及对记录和字段进行分层。因此，它也被称为自描述的结构。

半结构化数据，属于同一类实体可以有不同的属性，即使他们被组合在一起，这些属性的顺序并不重要。

常见的半结构数据有XML和JSON。



**非结构化数据**，顾名思义，就是没有固定结构的数据。各种文档、图片、视频/音频等都属于非结构化数据。对于这类数据，我们一般直接整体进行存储，而且一般存储为二进制的数据格式。

4.NOSQL基本含义、概念、背景

NoSQL指的是非关系型数据库，而我们常用的都是关系型数据库。就像我们常用的mysql，sqlserver一样，这些数据库一般用来存储重要信息，应对普通的业务是没有问题的。但是，随着互联网的高速发展，传统的关系型数据库在应付超大规模，超大流量以及高并发的时候力不从心。而就在这个时候，NoSQL得到了高速的发展。

NoSQL，指的是非关系型的数据库，该术语在 2009 年初得到了广泛认同。

应用最多的是以“键-值”对存储的形式，结构不固定。每一个元组可以有不一样的字段，每个元组可以根据需要增加一些自己的键值对。

不局限于固定的结构，可以减少一些时间和空间的开销。

其它类型的NoSQL还包括文档型的、列存储、图型数据库、xml数据库等。

|  |  |  |
| --- | --- | --- |
|  | 关系数据库 | NOSQL |
| 存储方式 | 表格式的；易关联协作存储；提取方便 | 大块组合在一起；存储在数据集中，就像文档、键值对和图结构 |
| 存储结构 | 结构化数据；修改关系数据库的数据难 | 非结构化数据 |
| 存储规范 | 数据分割到最小的关系表避免冗余（范式）；管理思路清晰但数据管理麻烦 | 数据可能重复；存储为一个整体，整块数据更加适合读写 |
| 存储扩展 | 纵向扩展：想要提高数据处理能力，需要更快的计算机。 | 横向扩展：存储天然就是分布式的，可以通过给资源池添加更多的普通数据库服务器来分担负载。 |
| 查询方式 | 结构化查询语言：SQL语句 | 使用非结构化查询语言：UnQI，它是没有标准的；更简单更精确的数据访问模式 |
| 事务 | 关系型数据库遵循ACID规则（原子性(Atomicity)、一致性(Consistency)、隔离性(Isolation)、持久性(Durability)）； 关系型数据库强一致性，对事务支持很好，易于回滚事务 | Nosql数据库遵循BASE原则（基本可用（Basically Availble）、软/柔性事务（Soft-state ）、最终一致性（Eventual Consistency））；Nosql数据库是在CAP（一致性、可用性、分区容忍度）中任选两项，因为基于节点的分布式系统中，很难全部满足，所以对事务的支持不是很好，虽然也可以使用事务，但是并不是Nosql的闪光点。 |
| 性能 | 关系数据库为了维护数据一致性付出了巨大代价读写能力较差，面对高并发时读写能力非常差，面对海量数据时效率非常低。 | NOSQL存储格式都是KEY-VALUE型，并且存储在内存中，非常易于存储，并对一致性是弱要求，无需SQL解析，提高了读写能力。 |

CAP原则又称CAP定理，指的是在一个分布式系统中，Consistency（一致性）、 Availability（可用性）、Partition tolerance（分区容错性），三者不可兼得。

分布式系统的CAP理论：理论首先把分布式系统中的三个特性进行了如下归纳：

● 一致性（C）：在分布式系统中的所有数据备份，在同一时刻是否同样的值。（等同于所有节点访问同一份最新的数据副本）

● 可用性（A）：在集群中一部分节点故障后，集群整体是否还能响应客户端的读写请求。（对数据更新具备高可用性）

● 分区容错性（P）：以实际效果而言，分区相当于对通信的时限要求。系统如果不能在时限内达成数据一致性，就意味着发生了分区的情况，必须就当前操作在C和A之间做出选择。

**课件2**

1.memcached简介：

一个高性能的分布式内存对象缓存系统，用于动态Web应用以减轻数据库负载。

基本特征：

1. 在内存中缓存数据和对象，为动态、数据库驱动网站提供更快的运行速度。减少读取数据库的次数，避免使用数据库应对高并发访问时磁盘开销和阻塞的发生。
2. LRU替换策略
3. 分布式缓存，不同主机上的多个用户可同时访问， 解决了单机应用的局限。
4. 使用自己的页块分配器，使用基于存储“键-值”对的hashmap哈希表。
5. 虚拟内存不会产生碎片，虚拟内存分配的时间复杂度可以保证为O(1)。
6. 通过在内存里维护一个统一的巨大的hash表，Memcached能够用来存储各种格式（图像、视频、文件以及数据库检索的结果等）的数据。
7. 具有守护进程（daemon），采用C语言开发。客户端通过memcached协议与守护进程通信，可以用任何语言来编写 。使用libevent（或者在linux下使用epoll）来均衡任何数量的打开链接。
8. 使用非阻塞的网络I/O，对内部对象实现引用计数。
9. 不提供冗余（如复制hashmap条目），当某个服务器S停止运行或崩溃了，所有存放在S上的键-值对都将丢失。

2.Memcached内存分配机制

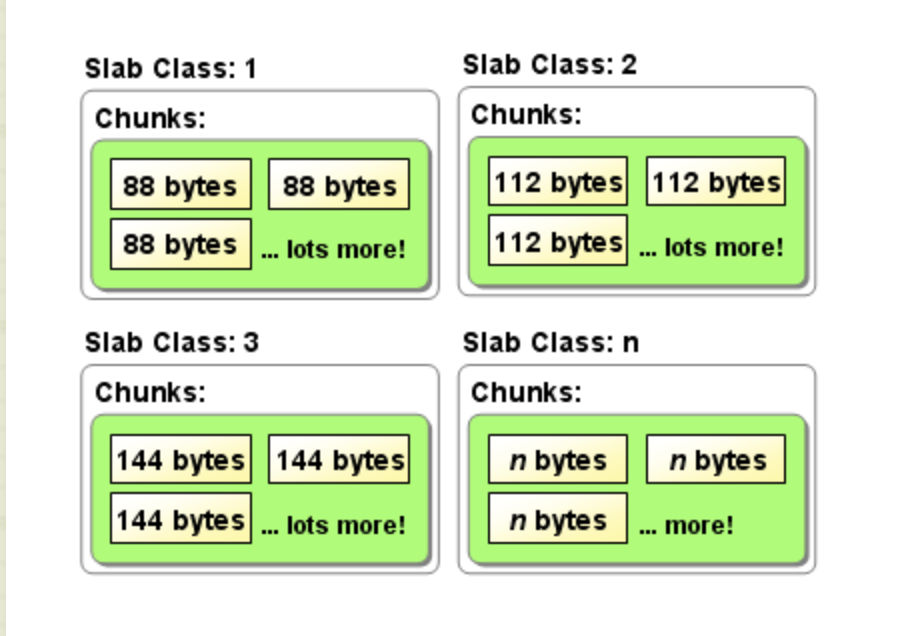
memcached使用了Slab Allocator的内存分配机制：相当于内存池机制，实现从操作系统分配一大块内存，然后memcached自己管理这块内存，负责分配与回收。

Slab Allocator基本原理：

1. Chunk——按照预先规定的大小，将分配的内存分割成特定长度的块
2. slab class——尺寸相同的块分成组（chunk的集合）
3. 分配到的内存不会释放，重复使用已分配的内存

一、首先，像一般的内存池一样，从操作系统分配到一大块内存；

二、将分配的内存分割成各种尺寸的块（chunk），并把尺寸相同的块分成组（slab class，chunk集合），chunk的大小按照一定比例逐渐递增。



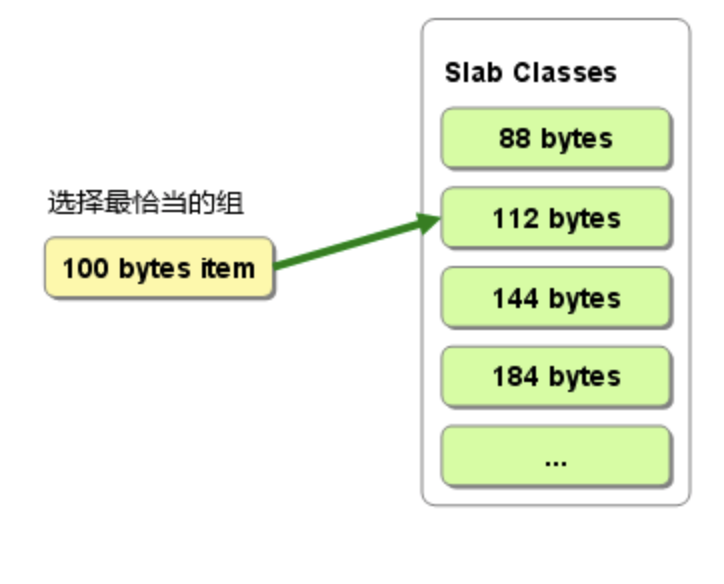
memcached的存储涉及到Slab, page, chunk三个概念

chunk：为固定大小的内存空间，默认为96Byte;

page：分配给Slab的内存空间，默认为1MB.然后在将1MB切分成chunk

slab：同样大小的chunk组称为slab

（例如：如果要存的数据为100，那么这个数据就会加入112byte的slab里边）



Slab Allocator存在的问题：

由于分配特定长度内存，可能无法有效利用分配的内存。

（还拿上边的例子举例，100的放到了112的chunk中，但是还有12byte没有用上，那么这12字节的空间就浪费了.）

3.memcached一致性分布的算法

Memcached的标准的分布式方法（对键的存储根据服务器台数的余数进行分散）：

1）求得键的整数哈希值；

2）除以服务器台数，根据其余数来选择服务器。

3）当选择的服务器无法连接时，rehash——将连接次数添加到键之后再次计算哈希值并尝试连接。

优点：方法简单，数据的分散性一般较好。

缺点：当添加或移除服务器时，缓存重组的代价大。

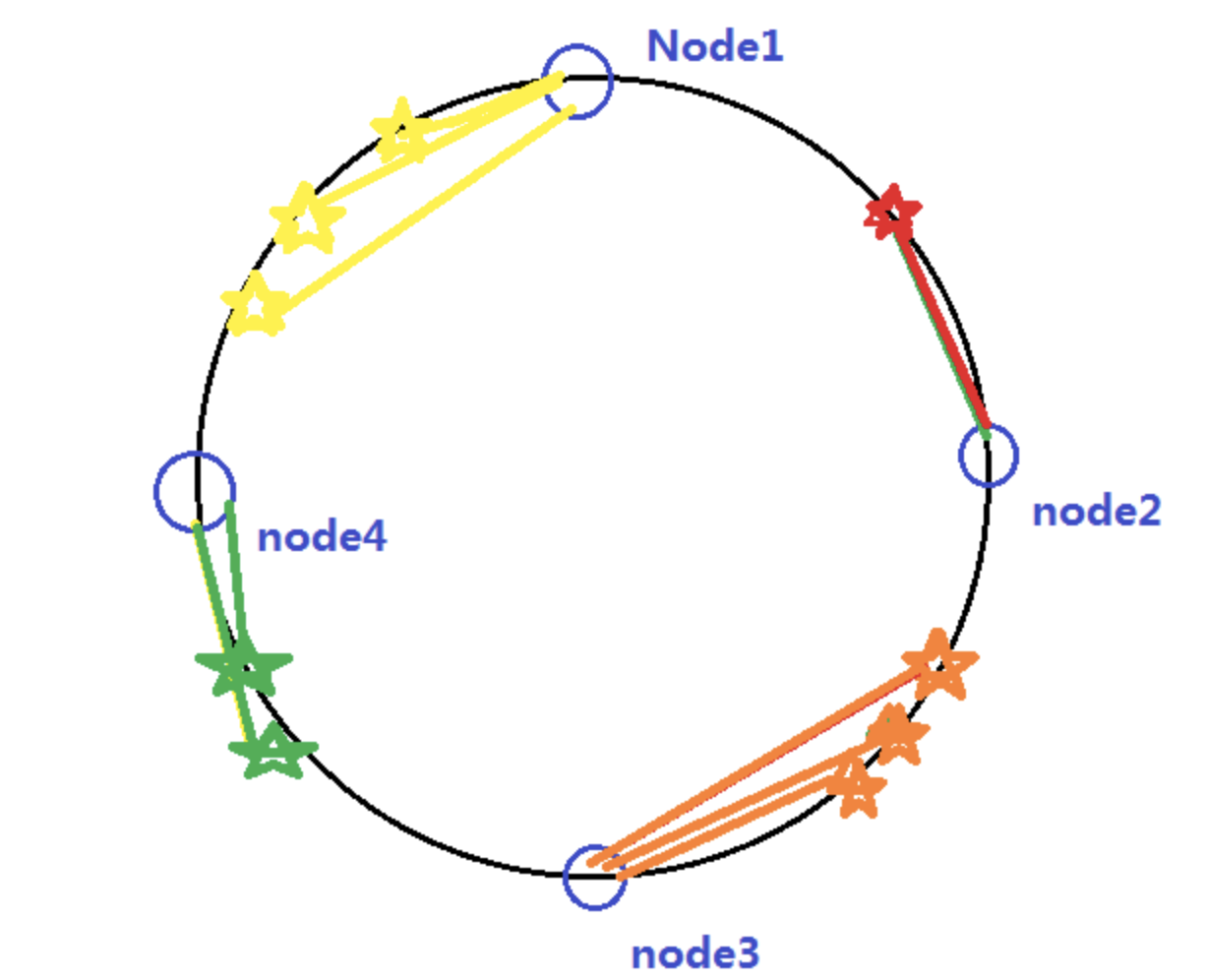
改进的分布式方法：一致性hash（Consistent Hashing）：

1）求出服务器节点的哈希值， 将其配置到0～2^32的圆上；

2）用同样的方法求出存储数据的键的哈希值并映射到圆上；

3）从数据映射到的位置开始顺时针查找，将数据保存到找到的第一台服务器上；

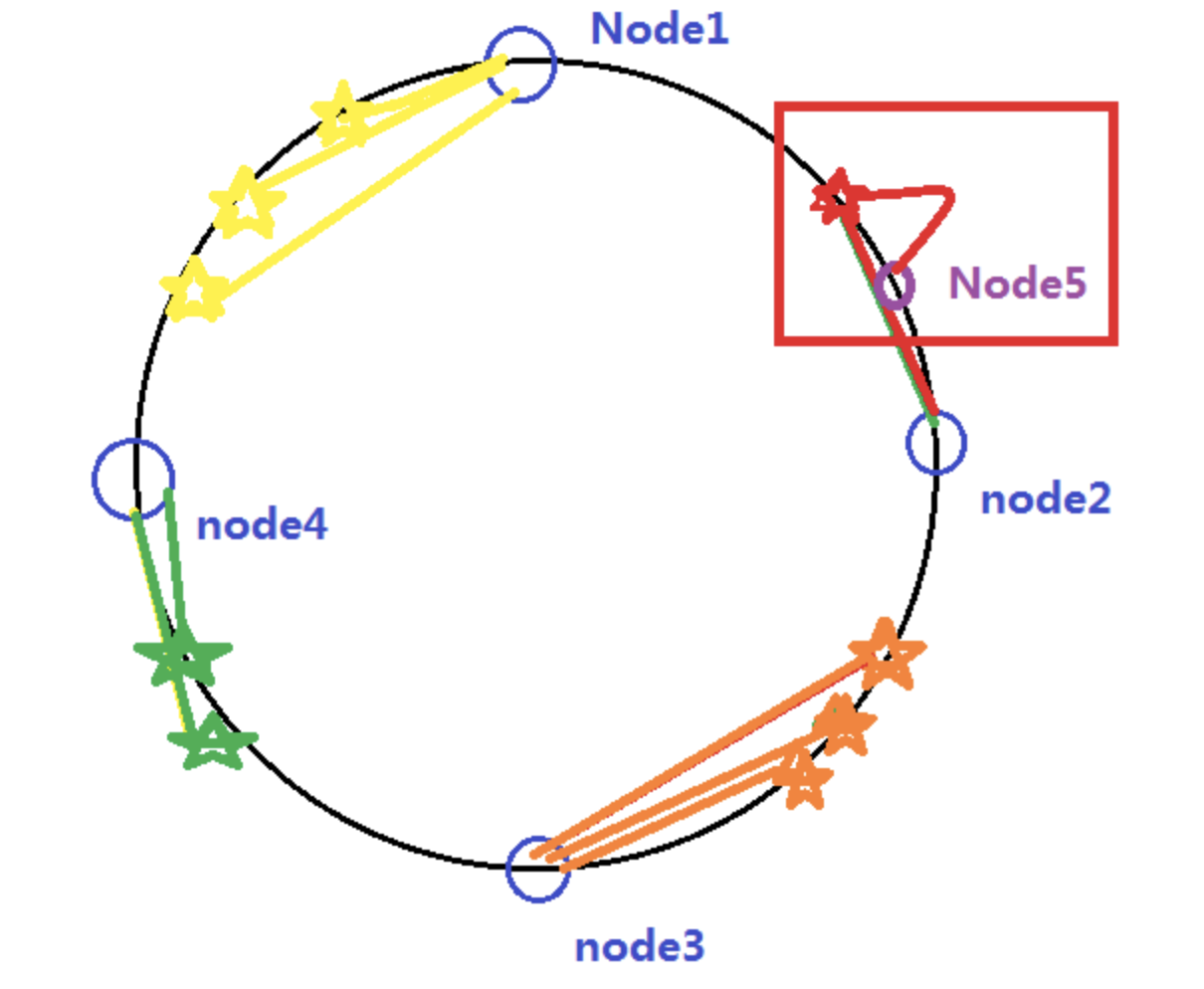
4）如果超过2^32仍然找不到服务器，就保存到第一台服务器上。



添加一台服务器？

只有在环上增加服务器位置的逆时针方向第一台服务器之间的键会受到影响，有效地抑制了键的重新分布。

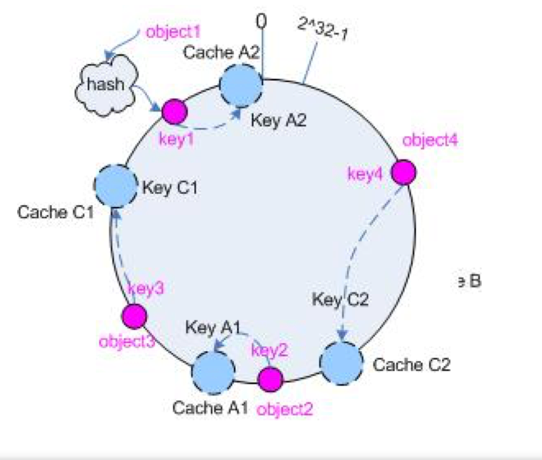
如下图所示，只有红色星星的位置到Node5之间的数据收到了影响，别的地方命中率是没有变化的，可以发现，我们布置的node越多，受影响的概率越小



使用一般的hash函数，服务器的映射地点的分布可能出现不均匀的情况。

解决方案——虚拟节点：

为每个物理节点（服务器）在圆环上分配100～200个点，从而抑制分布不均匀，最大限度地减小服务器增减时的缓存重新分布。



4.memcached内存替换策略

1）优先使用已超时的记录的空间。

2）如果还存在追加新记录时空间不足的情况， 使用最近最少使用（LRU）机制来替换已有缓存内容。 （引用计数非零则不替换）

LRU替换策略（Least Recently Used）：即最近最少使用，常用于页面置换算法，是为虚拟页式存储管理服务的。(把最近没用过的换了)

5.memcached数据删除、回收

记录超时不会释放已分配的内存，只是客户端无法再看见该记录，其存储空间此时可重复使用。

内部不实时监视记录是否过期，而是在get时查看记录的时间戳是否过期。

Memcached为每个item设置了一个过期时间，但不是到期就把item从内存中删除，而是访问item时如果到了有效期，才把item删除，使用do\_item\_get\_notedeleted函数在memcached中查找指定的item，从上面代码可知，当item过期时间早于当前时间时，便会删除此item。

6.memcached为什么要用惰性回收

**不在过期监视上耗费CPU时间。**

7. DynamoDB技术特征

（1）数据分布基本策略：

按哈希算法切分数据分布在不同node上，读数据依据key的哈希值寻找对应的node。使用Consistent Hashing算法（node对应的不再是一个确定的hash值，而是一个hash值范围）。另外，还设计了虚拟节点的大小与位置分配策略。

（2）支持数据的多副本写操作

节点临时性失效处理技术（sloppy Quorum & hinted handoff，一些副本不可用时，提供高可用和持久性的保证）；

节点永久性失效恢复技术（反熵 & Merkle trees，实现后台副本恢复）。

（3）节点成员关系和失效检测

基于Gossip的成员协议和失效检测，避免用中心节点管理节点成员关系。

8. DynamoDB对一致性hash的改进

原策略：设置虚拟结点的主要缺点：虚拟节点位置和大小随机，如果有新节点加入，可能导致扫描所有节点上的所有数据对象，判断全部数据对象是否需要迁移，这种全局的扫描造成很大的开销。

改进：固定所有虚拟节点的大小和位置，只改变虚拟节点和节点的对应关系。

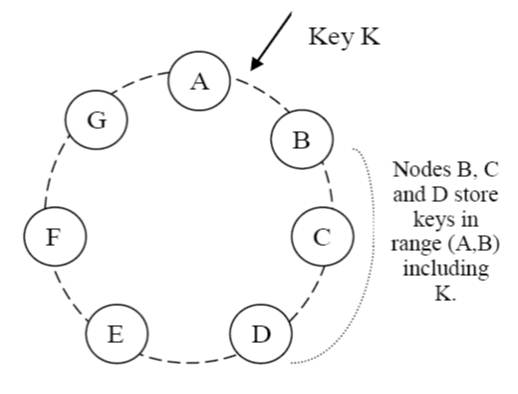
将整个地址空间平均分成Q个虚拟节点，每个物理节点（假设有S个）分配Q/S个虚拟节点。当有节点加入时，从现有节点每个拿出等量的虚拟节点分给新节点；当有节点离开时，将此节点的所有虚拟节点平均分配给余下的节点，保证系统中每个节点始终都有Q/S个虚拟节点。

固定所有虚拟节点的大小和位置的策略下，为实现结点加入和离开系统时影响较小，要求Q>>S。如果业务快速发展，使得主机不断增加，从而导致Q不再满足Q>>S，该策略将不断退化。(实际应用中还有存储备份的因素，假设每个数据存储N个备份，则要满足Q>>S\*N)。

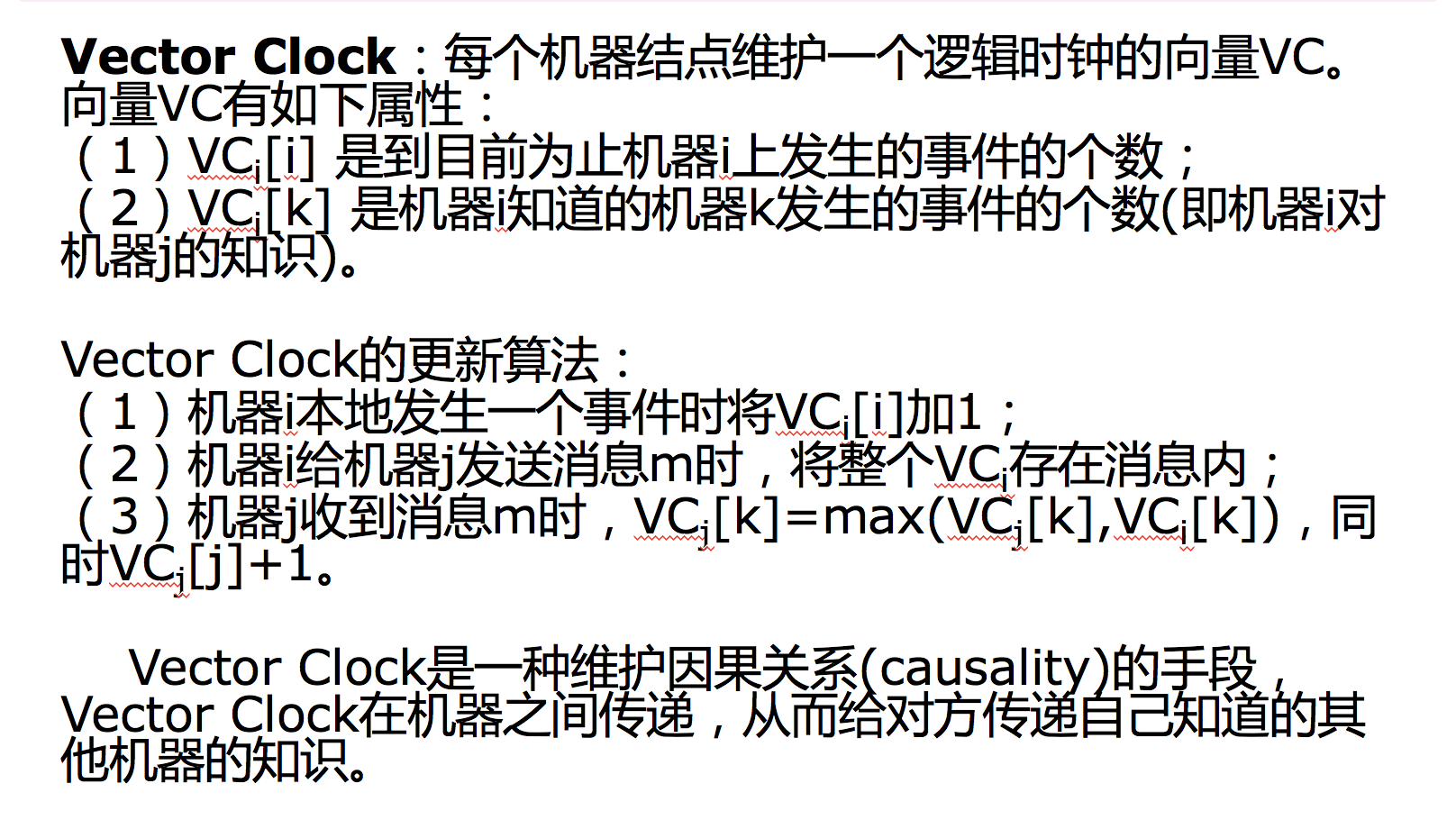
1. DynamoDB多副本机制：sloppy Quorum

Sloppy quarum（草率的法定人数，草率仲裁）

每个数据对象有N个副本，分别存放在N个不同的节点上（该数据对象在地址环上顺时针找到N个不同的节点），这N个节点称为该数据的preference list。下图中key为k的数据对象的preference list为节点B、C、D。



10. Vector Clock简介



通过比较这些向量的大小，来确定事件发生的顺序：假如一个向量的所有分向量的count值都小于或等于另一个向量，可以认为后者并前者更“新”；否则，存在冲突。

例：

对于每一个分布式存储的对象副本都有这样一个时间戳，则存在以下几种关系：

1. 本机{0，0，1}， 消息{0，1，2}。消息的每个节点的count都大于等于本机的，因此舍弃本机，同步消息。
2. 本机{0，1，2} ，消息{0，1，1}。消息的每个节点的count都小于等于本机的，那么舍弃消息，保留本机。

（3）本机{0，3，1}， 消息{0，1，2}。有的大，有的小，出现冲突，无法判断谁是最新版本，需要进行冲突仲裁。

Vector Clock实例：

设有3个节点，NRW协议中W=1,则R=N=3。

假设一个写请求，第一次被节点A处理了。节点A会增加一个版本信息(A，1)，把这个时候的数据记做D1(A，1)。然后另外一个对同样key的请求又被A处理了，于是有D2(A，2)。

此时D2是可以覆盖 D1的，没有冲突产生。现在假设D2传播到了所有节点(B和C)，所以B和C都持有数据D2(A，2)。

接下来又一个请求被B处理了，生成数据D3(A，2;B，1) 。（因为这是一个新版本的数据，被B 处理，所以要增加B的版本信息）

假设D3还未传播到C时，又一个请求被C处理记做，产生D4(A，2;C，1)。

如果在这些版本没有传播开以前，发生读取操作，因为R=3，所以R会从所有三个节点上读，读到三个版本。A上的D2(A，2)、B上的D3(A，2;B，1)、C上的D4(A，2; C，1)。依据Vector Clock算法，D2已经是旧版本（其B、C分量均为0），可以舍弃，但是D3和D4都是新版本，需要应用自己去合并。

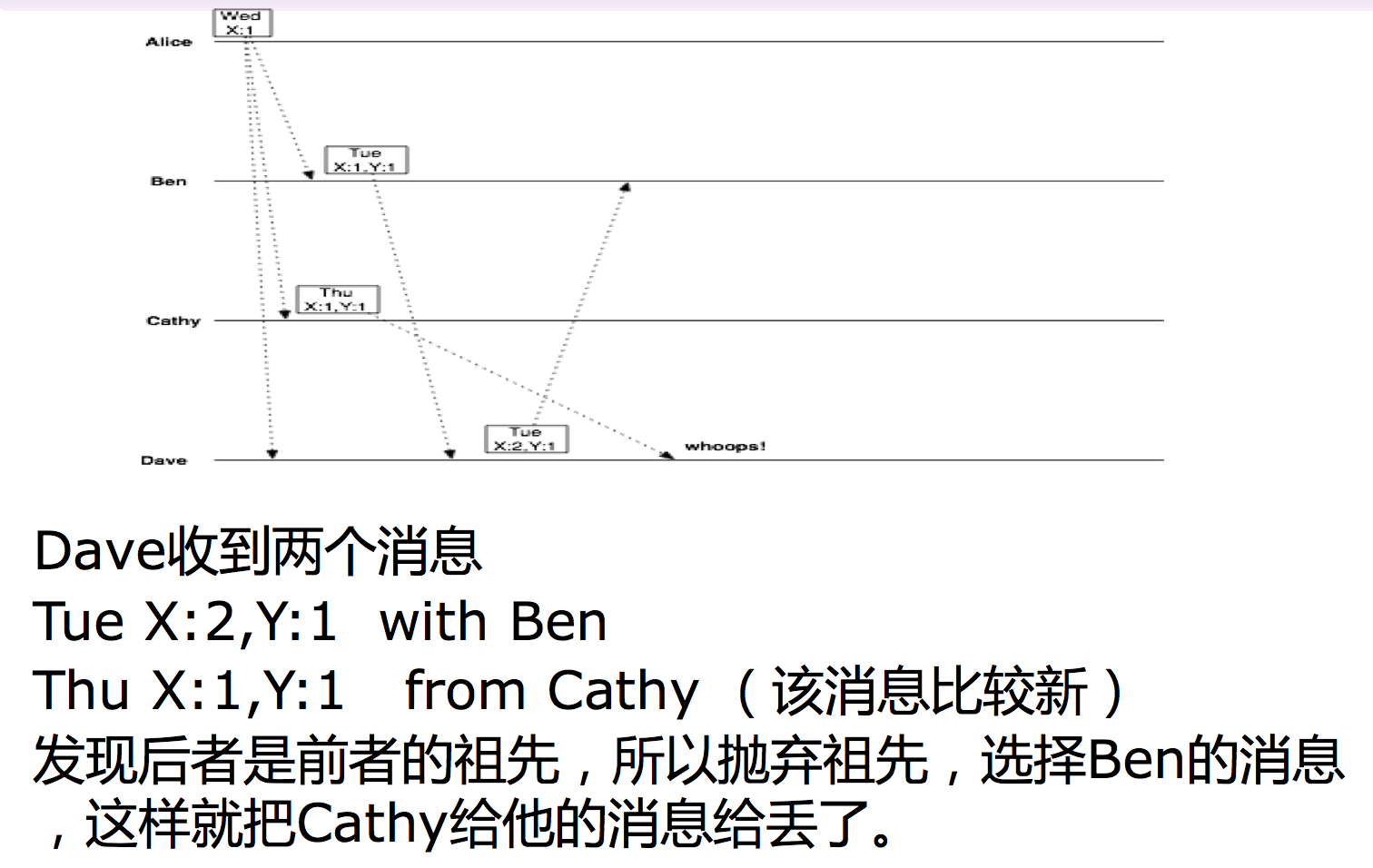
11. DynamoDB引入Vector Clock的更新算法与带来问题与解决方案

Vector Clock向量维度的无限增长问题：

现实生活中，如果有很多的决策者，相当于有很多的客户端，整个向量时钟的长度就无限制增长了，这对于存储系统来说，需要想办法解决。

解决方案（1）：采用服务器向量：server的数量是可控的，可以不用client来标识向量空间，而是用server来标识向量空间——向量标签不再是客户端，而是server标识。

问题：会因为网络传输延时而丢失数据。如下图：



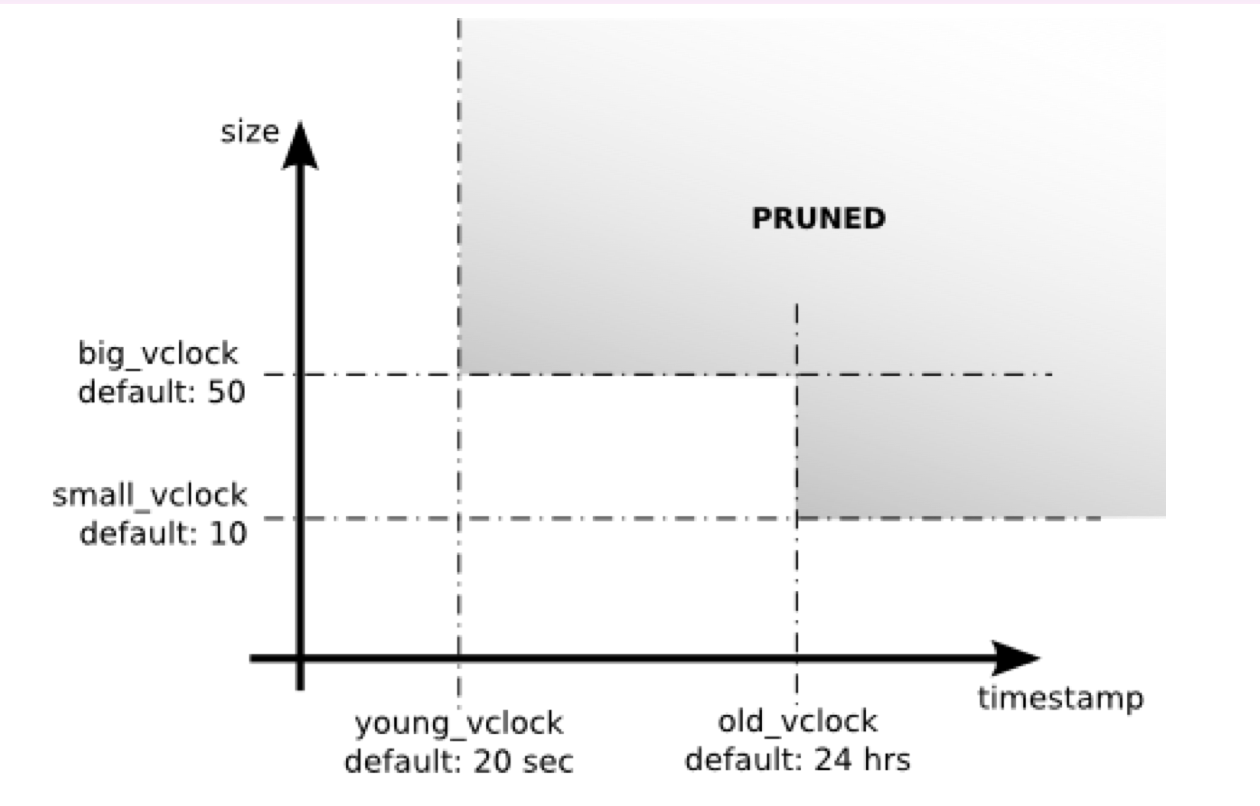
解决方案（2）：向量时钟的剪枝

Riak剪枝系统用四个参数来避免向量时钟空间的无限增长：

* small\_vclock
* big\_vclock
* young\_vclock
* old\_vclock

small\_vclock和big\_vclock参数标识向量时钟的长度，如果长度小于small\_vclock就不会被剪枝掉，如果长度大于big\_vclock就会被剪枝掉。

young\_vclock和old\_vclock参数标识存储这个向量时钟时的时间戳，剪枝策略同理，大于old\_vclock的才会被剪枝掉。



向量剪枝尽量只丢掉一些向量时钟的信息，而不是丢掉实实在在的数据。

特殊情况带来的问题：一个客户端保持了一个很久之前的向量时钟，然后继承该向量时钟并提交了一个数据，此时会产生问题（因为服务器已经没有这个很久之前的向量时钟信息了，可能已经被剪枝掉了），所以客户端提交的此次数据，在服务端无法找到一个祖先，此时Riak会创建一个sibling。

剪枝策略是一个权衡方案，一方面是无限增长的向量时钟的空间，另一方面是偶尔的会有“false merge”，产生兄弟数据，但不会丢失数据。

从这个意义上看，防止向量时钟空间的无限增长，剪枝策略优于用server标识向量时钟的策略。

12. DynamoDB节点临时失效处理：hinted handoff技术（暗示接力）

为了保证每次都能写到W个副本，读到R个副本，我们每次读和写都是发送给N个节点。如果这N个节点有节点失效，**那么往后继续找一个不同的节点，暂时的代替失效的节点，**当该后续节点定期监测到故障节点恢复，则将暂时代为保管的数据写回复活节点。

例：N=3，某数据的preference list是节点A、B、C。若A节点失效，则对该数据的写请求将发送到节点B、C、D上。D暂时取代A的角色，那些原本应该写到A上的数据存放在D中的**一个特定的文件夹**中（意味着这些数据不是D本该拥有的，而是别的节点的）。**D上会启动一个线程定期检查A的状态，当发现A恢复后，就将D上存放的这些A的数据写回到A。**

该策略保证了节点失效时系统的高可用和数据持久性。

13. Dynamo节点永久性失效恢复技术

Dynamo实现了反熵协议(anti-entropy，基于Gossip的一致同步) 来保持副本同步。当故障发生或者有节点加入、离开集群时，都涉及分片的拷贝和传输，因此希望能够快速检查分片中内容是否相同，并通过仅发送不同的部分来减少数据传输量。为了更快地检测副本之间的不一致性，并且减少传输的数据量和反熵过程中磁盘读取的次数，Dynamo采用 Merkle Tree技术。

14. Merkle tree

原理：

每个叶子节点对应一个数据项，并记录其hash值 ；

每个非叶子节点记录其所有子节点的hash值。

Dynamo为每一个分片维护一个Merkle Tree，需要比较分片是否相同时，自根向下的比较两个Merkle Tree的对应节点，可以快速发现并定位差异所在。

优点：

1. 树的每个分支可以独立地检查，而不需要下载整个树或整个数据集；
2. 有助于减少为检查副本间不一致而传输的数据的大小；
3. 通过部分hash就能校验整个文件的完整性。

例：如果两树的根哈希值相等，且树的叶节点值也相等，那么节点不需要同步。如果不相等，则意味着一些副本的值不同。在这种情况下，节点可以交换子节点的哈希值，该处理一直进行到树的叶子，此时主机可以识别出“不同步”的 key，通过这种二叉树的结构可以在log(N)的复杂度快速定位到出错的数据块。

典型应用：文件校验（BitCommit，BitTorrent种子）、副本同步（DynamoDB）、可信计算、区块链

**课件3**

了解：淘宝做TFS部署的数据规模

存储区域网的特点、通讯模式

了解：oceanbase对事务的支持能力

**课件4（GFS）**

1. 设计思想与基本概念

组件失效被视作是常态而不是意外事件，放松了对一致性模型的要求，引入了原子性的记录追加操作，从而保证多个客户端能同时进行追加操作，不需要额外的同步操作来保证数据的一致性。

Chunk（大块）：文件被分割成固定大小的Chunk。每一个Chunk创建时，Master服务器给其分配一个64位全球唯一且不变的Chunk标识。

Namespace(名称空间)：GFS提供了一套类似传统文件系统API的接口函数（没有严格按照POSIX等标准API的形式实现），文件以分层目录的形式组织，用路径名来标识。在逻辑上，GFS的名称空间就是一个全路径和元数据映射关系的查找表。

1. 元数据的存储与访问机制

1）利用前缀压缩，namespace表高效的存储在内存中。

2）在存储名称空间的树型结构上，每个节点（绝对路径的文件名或绝对路径的目录名）都有一个关联的读写锁。

不同于许多传统文件系统，GFS没有针对每个目录实现能够列出目录下所有文件的数据结构，GFS也不支持文件或者目录的链接（即Unix术语中的硬链接或者符号链接）。

1. GFS体系架构

* Chunkserver：Chunk服务器把Chunk以linux文件的形式保存在本地硬盘上，根据指定的Chunk标识和字节范围来读写块数据。

可靠性：每个块会复制到多个chunkserver上（缺省3个存储复制节点）。

复制级别：用户可以为不同的文件命名空间区域设定不同的复制级别。

* Master节点：GFS分布式文件系统中的总控节点。

1. 执行所有的namespace操作。
2. 管理整个系统里所有Chunk的副本：
   1. 创建新Chunk和它的副本
   2. 决定Chunk的存储位置
   3. 在所有的Chunk服务器之间进行负载均衡
   4. 协调各种系统活动以保证Chunk被完全复制
   5. 回收不再使用的存储空间
3. 管理所有的文件系统元数据（保存在内存中，Chunk的数量以及整个系统的承载能力都受限于Master的内存大小）：

（1）Namespace； （2）文件和Chunk的映射信息；

（3）Chunk当前的位置信息；（4）访问控制信息

1. 管理系统范围内的活动

（1）Chunk租约管理；（2）孤儿Chunk的回收；

（3）Chunk在Chunk服务器之间的迁移。

1. Master不持久保存Chunk位置信息，而是使用心跳信息周期地和每个Chunk服务器通讯（轮询）：（1）发送指令到各个Chunk服务器；（2）接收Chunk服务器的状态信息。
2. Master持久化保存操作日志：操作日志记录两种类型的元数据变化（命名空间、文件和Chunk的对应关系）：（1）同时会以操作日志的方式记录在操作系统的系统日志文件中（存储在本地磁盘）；（2）该日志同时被复制到其它的远程Master服务器上（系统能简单可靠的更新Master服务器的状态，不用担心Master服务器崩溃导致数据不一致的风险）。

* Client：GFS客户端实现了GFS文件系统的API接口函数，代码以库的形式被链接到客户程序里。应用程序同时与Master节点和Chunk服务器通讯，对数据进行读写操作：
* Control message：客户端和Master节点的通信只获取元数据；
* Data message：所有的数据操作都是客户端和Chunk服务器进行交互的。

1. 元数据操作日志

* 操作日志记录元数据变更历史的作用：

1）是元数据唯一的持久化存储记录；

2）是判断同步操作顺序的逻辑时间基线；

文件和Chunk（及其版本）都由创建它们的逻辑时间唯一的、永久的标识。

* 写日志机制：

1）确保日志文件是完整的且元数据的变化被持久化后，日志才对客户端是可见的。否则，即使Chunk本身没有出现问题，仍有可能丢失整个文件系统（或者客户端最近的操作）。

2）日志会复制到多台远程机器，只有相应的日志记录写入本地以及远程机器的硬盘后，才会响应客户端的操作请求。

3）Master服务器会收集多个日志记录后批量处理，以减少写入磁盘和复制对系统整体性能的影响。

1. 灾难恢复/日志checkpoint

灾难恢复时，Master通过重演操作日志把文件系统恢复到最近的状态。

解决方法：checkpoint，减少重演操作的日志量。

日志增长到一定量 → Master对系统状态进行一次Checkpoint，将所有的状态数据写入一个Checkpoint文件，并删除之前的日志文件。

系统恢复：Master从磁盘上读取最新的Checkpoint文件，重演Checkpoint之后的有限个日志文件。

Checkpoint文件存储：压缩B树形式。可直接映射到内存，在用于命名空间查询时无需额外的解析，从而大大提高恢复速度，增强可用性。

Checkpoint创建：checkpoint过程不阻塞正在进行的修改操作：

1） Master使用独立的线程切换到新的日志文件和创建新的Checkpoint文件；

2）新的Checkpoint文件包括切换前所有的修改；

3）创建完成后，Checkpoint文件被写入本地和远程硬盘。

重点太多不整理了，直接看PPT：

1. GFS写数据，主副本和二级副本的数据传播方式，更新的数据推送方式
2. GFS是单一master结构（逻辑上唯一的master节点，物理上有master节点复制机制），面向不稳定集群，master管理哪些数据，哪些数据是不持久化的，哪些数据是持久化的
3. GFS特殊操作：原子性操作

**课件5（2个）**

BigTable基本设计思想:MAP实现table

MAP寻址三个关键字：行关键字+列关键字+时间戳

MAP怎样用三层寻址方式实现

Paxos算法基本思想

了解CAP定理以及解读CAP定理的两种应用场景（金融领域、互联网领域）

RAFT算法日志包含哪些东西（经日志复制同步做一致性），复杂流程（机群转换中应用过渡配置）

Big Table：多维度排序Map

\* 寻址+缓存策略

\* 分布式锁Chubby

**课件6**

map/reduce模型

**课件7**

发现迭代过程中有些东西是可以优化的：网络传输开销、计算开销

怎么优化map/reduce执行代码框架，写算法（e.g.如何把优化的思想用在大规模的图中，找多跳邻居 ；用数学的形式，放在map/reduce的框架下，把找多跳邻居描述出来，之后写怎么实现它；包括主函数、map函数、reduce函数怎么写；->综合题）

**课件8**

了解：

storm集群里有特殊节点zookeeper，zookeeper管什么东西

storm用拓扑逻辑把任务分解，分布式的执行，分布式包含两种并行：task之间的并行、先后task之间的流水线并行

job分解为task集合，最终放在物理CPU上跑。逻辑层面的task和物理层面的executor，了解executor以及executor如何共享集群资源

storm高效的消息保障技术怎么做的

**课件9**

spark中Yarn-cluster的部署，job执行过程，资源分配与释放，执行过程，图怎么画的