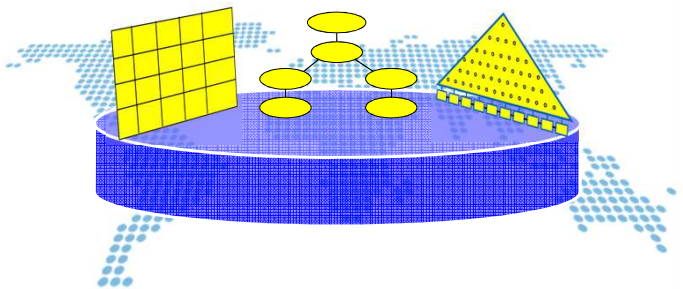
数据库系统

# 大数据系统初步

陈世敏

（中科院计算所）

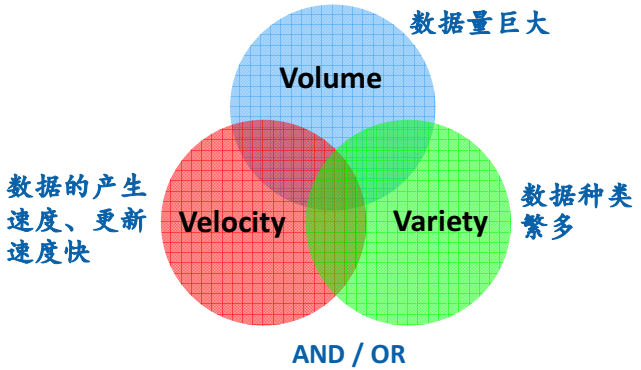


## Outline

- 大数据系统简介
  - 基本概念
  - KV键值对存储系统
  - 图数据库和图计算系统
  - 树状数据（JSON）数据库
- HDFS和MapReduce
  - 分布式文件系统HDFS
  - MapReduce



## 大数据的概念（Big Data）也是三个重要挑战






## 大数据的概念（Big Data）

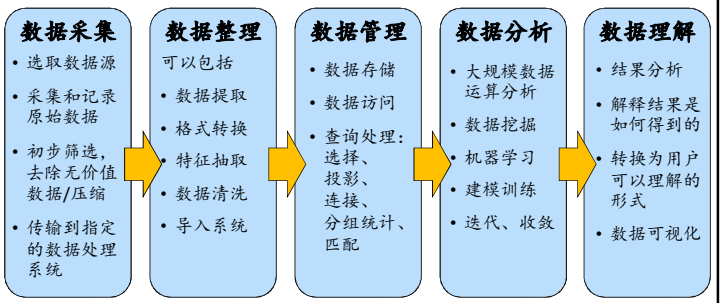
- 与传统的关系型数据相比
  - 传统：大部分情况是先设计系统，再采集数据
  - 大数据：很多时候是先有各种数据，然后需要分析
- 处理需求更加丰富了
  - 不局限于传统的关系运算
  - 各种复杂的机器学习算法、分析统计和迭代算法
- 数据密度降低了
  - 有价值的信息与数据总量相比

## 大数据与云计算是什么关系？

- 云计算(Cloud computing)
  - 数据中心网络环境，大量的服务器节点，虚拟机
  - 基础云服务：Amazon AWS, 阿里云
  - 在基础云服务上提供计算平台、应用平台
- 大数据(Big Data)
  - 更加强调数据的采集、存储、管理、运算和分析
  - 大数据系统很多时候是运行在云上的



## 大数据处理流程




我们集中关注的主要内容是
数据管理系统和部分数据分析系统



## 基本概念

- 时间性能
- 吞吐率
- 空间占用
- 复杂性
- 可扩展性
- 可用性
- 数据冗余
- 故障模型



## 可扩展性（Scalability）

- 两种扩展
- 向上扩展（Scale Up）：增加核数
  - 同一台服务器内部
  - 增加大数据系统采用的CPU核的数量
  - 主要问题：多线程的数据竞争、并发控制
- 向外扩展（Scale Out）：增加机器数
  - 多台服务器
  - 增加大数据系统采用的机器节点数量
  - 主要问题：多机通信、数据传输

## 扩展性（2）

- 两种可扩展性度量
- 强可扩展性（Strong Scalability）
  - 用更多的硬件（CPU核/机器节点）来执行同一个任务
  - 希望处理得更快
  - 线性可扩展：同一任务，n组硬件➔执行时间减少为1/n
- 弱可扩展性
  - 用更多的硬件（CPU核/机器节点）处理更多的数据
  - 改变任务需要处理的数据量
  - 线性可扩展：n组硬件，n倍数据量➔执行时间不变



## 可用性（Availability）

- 系统正常运行的时间占总时间的比例


- 可用性=$\frac{\text{正常时间}}{\text{总时间}}$
- 高可用性（High Availability）
  - 一些说法：24x7，意思是24小时，7天都正常运行
  - 99.9%：每年有8.76小时宕机/离线维护
  - 99.99%：每年有52.6分钟宕机/离线维护
  - 99.999%：每年有5.26分钟宕机/离线维护
  - 99.9999%：每年有31.5秒宕机/离线维护

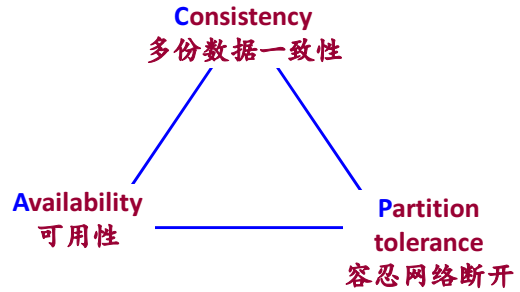


## 分布式系统类型

- Client / Server
  - 客户端发送请求，服务器完成操作，发回响应
  - 例如：3-tier web architecture
    - Presentation: web server
    - Business Logic: application server
    - Data: database server
- P2P (Peer-to-peer)
  - 分布式系统中每个节点都执行相似的功能
  - 整个系统功能完全是分布式完成的
  - 没有中心控制节点
- Master / workers
  - 有一个/一组节点为主，进行中心控制协调
  - 其它多个节点为workers，完成具体工作

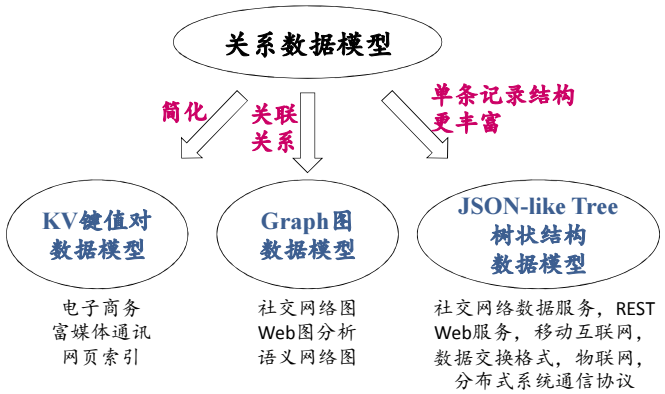


## CAP定理




三者不可得兼

## 大数据常见的数据类型






## SQL 与 No-SQL

- SQL就是指关系型数据库
- NoSQL
  - 简化关系型数据库的能力
  - 支持非关系的数据模型
- 包括
  - 上述KV、图、树状结构数据模型



## Key-Value Store

- Key-Value store是一种分布式数据存储系统
  - 简而言之，数据形式为<key, value>，支持Get/Put操作
  - 实际上，多种不同的系统的数据模型和操作各有差异
- 现有系统举例
  - Dynamo: 由Amazon公司研发
  - Bigtable / HBase: Bigtable起源于Google公司, Hbase是开源实现
  - Cassandra：由Facebook研发，后成为Apache开源项目
  - Memcached：基于内存的单机键值对系统
  - Redis：基于内存的分布式键值对系统



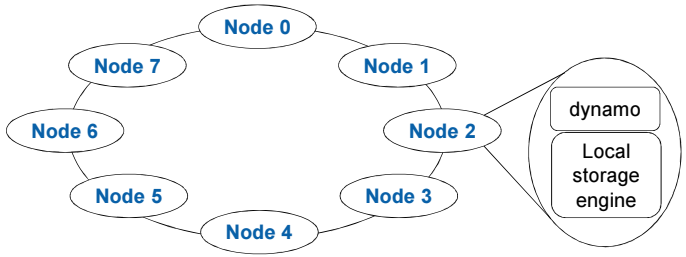
## Dynamo数据模型和操作

- 最简单的<key, value>
  - key = primary key：唯一地确定这个记录
  - value：大小通常小于1MB
- 操作
  - Put(key, version, value)
  - Get(key) ➔ (value, version)
- ACID?
  - 没有Transaction概念
  - 仅支持单个<key,value>操作的一致性

**修改多个<key, value>可能出现什么问题？**

各种不一致情况，要求上层应用设计时考虑这点

## Dynamo系统结构




- 多个nodes互连形成分布式系统
- 每个node上由local storage engine + dynamo软件层组成
  - Local storage engine: Berkeley DB, 或MySQL, etc.
  - 用于存储<key, value>



## 最终一致性(Eventual Consistency)

- Put操作并没有等待所有N个节点写完成
  - 可以提高写效率
  - 可以避免访问出错/下线的节点，提高系统可用性
- 系统总会最终保证每个<key,value>的N个副本都写成功，都变得一致
  - 但并不保证能够在短时间内达到一致
  - 最终可能需要很长时间才能达到
- 这种“最终”达到的一致性就是eventual consistency



## 图(Graph)的概念

- G=(V, E)
  - V：顶点(Vertex)的集合
  - E：边(Edge)的集合
    - 边 e=(u,v), u∈V, v∈V
- 有向图
  - 边有方向
- 无向图
  - 边没有方向
  - 可以用有向图表达无向图：每条无向边➔2条有向边



## 与图相关的系统

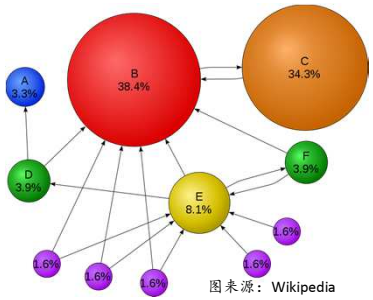
- 图数据存储系统
  - 存储图顶点和边
  - 提供顶点和边的查询
  - 例如：Neo4j，JanusGraph
- 图运算系统
  - 以图为基础运行大规模算法
  - 例如：Pregel, Giraph, PowerGraph, Spark GraphX, GraphLite等

## Neo4j

- 单机图数据库系统
  - 采用自定义的结构在本地硬盘存储图的顶点和边
- 开源Java实现
- Neo4j存储
  - 顶点：称为node
  - 边：称为relationship
  - 顶点和边上可以存储多个key-value值：称为property
- Neo4j使用
  - Cypher：Declarative query language
  - Traversal: Embedded Java lib



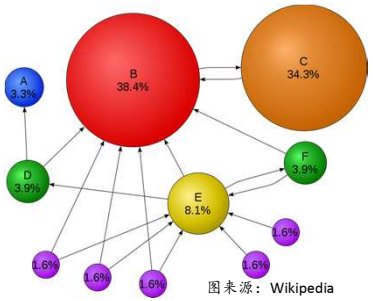
## 图计算举例：PageRank




- Google用于对网页重要性打分的算法
- 上图简单示意了PageRank在一个图上的运行结果
  - 顶点：网页
  - 边：超链接



## 图算法举例：PageRank



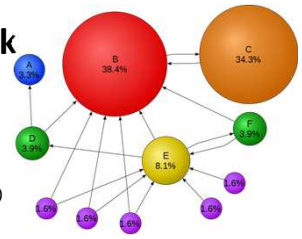

**如果没有这种随机跳转，进入A,B,C后就出不来了**

- 用户浏览一个网页时，有85%的可能性点击网页中的超链接，有15%的可能性转向任意的网页
  - PageRank算法就是模拟这种行为
  - d=85%（damping factor）



## 图算法举例：PageRank

- $R_u = \frac{1-d}{N} + d\sum_{v \in B(u)} \frac{R_v}{L_v}$
  - $R_v$: 顶点v的PageRank
  - $L_v$: 顶点v的出度（出边的条数）
  - $B(u)$: 顶点u的入邻居集合
  - d: damping factor
  - N: 总顶点个数




- 计算方法
  - 初始化：所有的顶点的PageRank为$\frac{1}{N}$
  - 迭代：用上述公式迭代直至收敛

## 图算法举例：PageRank

- $R_u = \frac{1-d}{N} + d\sum_{v\in B(u)}\frac{R_v}{L_v}$

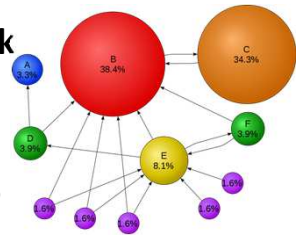
**问题：N非常大时，数据精度可能不够？**

- $NR_u = 1 - d + d\sum_{v\in B(u)}\frac{NR_v}{L_v}$
- 设$R'_u = NR_u$
- $R'_u$初始化为1
- $R'_u = 1 - d + d\sum_{v\in B(u)}\frac{R'_v}{L_v}$



## 图算法举例：PageRank

- $R_u = 1 - d + d\sum_{v\in B(u)}\frac{R_v}{L_v}$
  - $R_v$: 顶点v的PageRank*N
  - $L_v$: 顶点v的出度（出边的条数）
  - $B(u)$: 顶点u的入邻居集合
  - d: damping factor
  - N: 总顶点个数

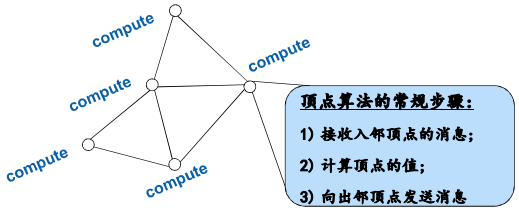


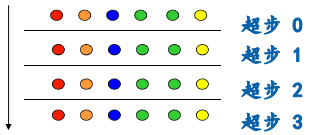

图来源：Wikipedia

- 计算方法
  - 初始化：所有的顶点的PageRank为1
  - 迭代：用上述公式迭代直至收敛



## Pregel分布式图计算模型




BSP模型



## JSON

- JavaScript Object Notation
  - 是一个低成本的数据交换格式
  - 是Javascript程序语言标准(1999年)的子集
- JSON对应于程序语言中的结构与数组
- 举例：

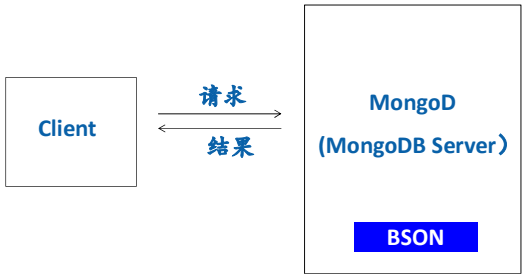
```
{"id" : 131234, "name" : "张飞", "major" : "计算机", "year": 2013, "course" :
  [ {"course name" : "体系结构", "year" : "2014", "grade" : 85},
    {"course name" : "操作系统", "year" : "2014", "grade" : 90},
    {"course name" : "英语", "year" : "2013", "grade" : 87} ],
  "address": {"州" : "幽州",
              "郡" : "涿郡",
              "街道" : "张家庄",
              "邮编" : "072750"}
}
```

## Document Store


- Document store
  - JSON是基本数据类型，存储为BSON二进制表示
- 基于C++实现
- 名词
  - Database ~ 关系型中的数据库概念
  - Collection ~ 关系型中的table概念
  - Document ~ 关系型中的记录概念
- 一个database包含多个collections，每个collection包含多个documents
  - document < 16MB



## 单机MongoDB

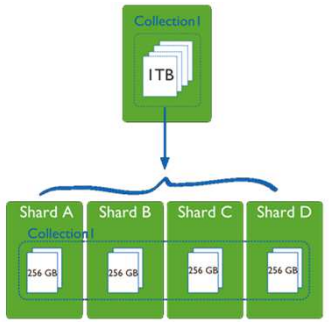



- 与RDBMS非常相似
- BSON：一种JSON二进制表示
  - 行式，key都是字符串，value根据具体的类型存储二进制值



## 分布式MongoDB

- Sharding = horizontal partitioning




MongoDB Manual

- Shard key
- range partitioning
- hash partitioning



## Outline

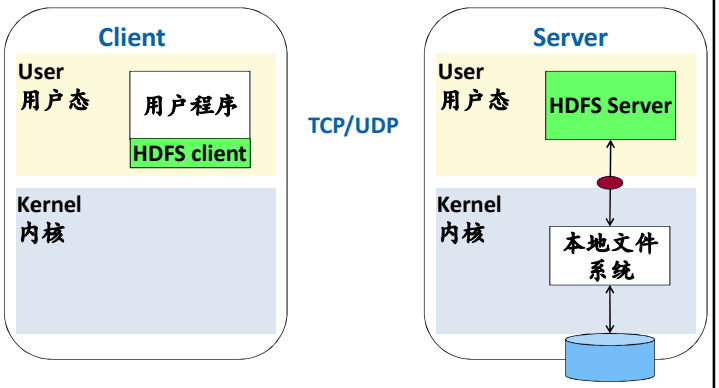
- 大数据系统简介
  - 基本概念
  - KV键值对存储系统
  - 图数据库和图计算系统
  - 树状数据（JSON）数据库
- HDFS和MapReduce
  - 分布式文件系统HDFS
  - MapReduce

## GFS/HDFS

- Google File System
  - SOSP 2003，C/C++实现
  - Google MapReduce系统的基础
- Hadoop Distributed File System
  - Google File System的开源实现
  - 基于Java
  - 与Hadoop捆绑在一起
  - 是一种应用层的文件系统

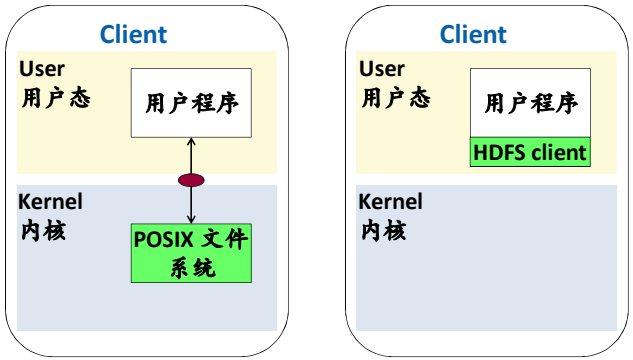



## GFS/HDFS是应用层文件系统






## POSIX文件系统 vs. 应用层文件系统



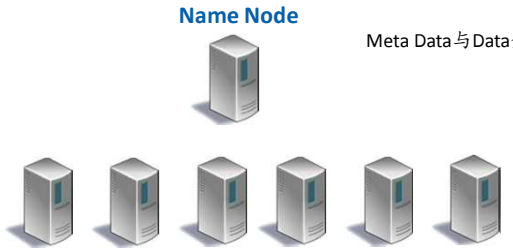

所有程序不改变就可以用　　必须链接HDFS client库才能使用



## GFS设计目标

- 优化
  - 大块数据的顺序读
  - 并行追加(append)
- 不支持
  - 不支持文件修改(overwrite)操作
  - 所以，consistency的实现可以大大简化！

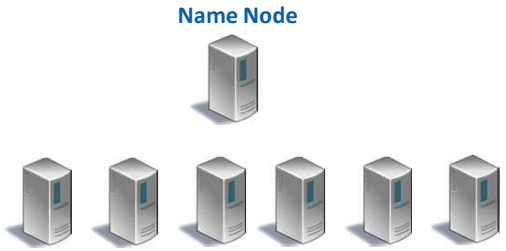
## HDFS/GFS系统架构




- Name Node：存储文件的metadata(元数据)
  - 文件名，长度，分成多少数据块，每个数据块分布在哪些Data Node上
- Data Node: 存储数据块



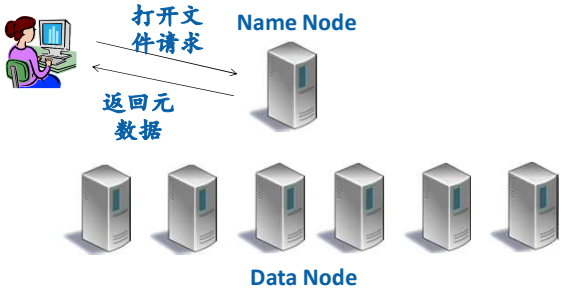
## HDFS/GFS系统架构




- 文件切分成定长的数据块（默认为64MB大小的数据块）
- 每个数据块独立地分布存储在Data Node上
- 默认每个数据块存储3份，在3个不同的data node上
  - Rack-aware



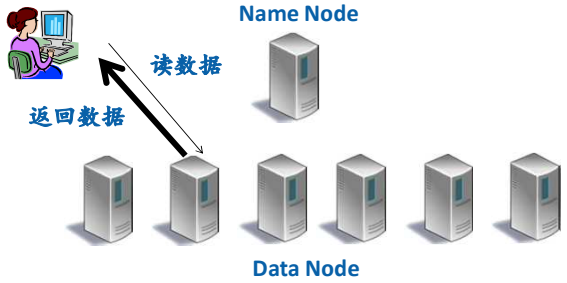
## HDFS/GFS文件操作：open




- 打开文件时，与Name Node通信一次

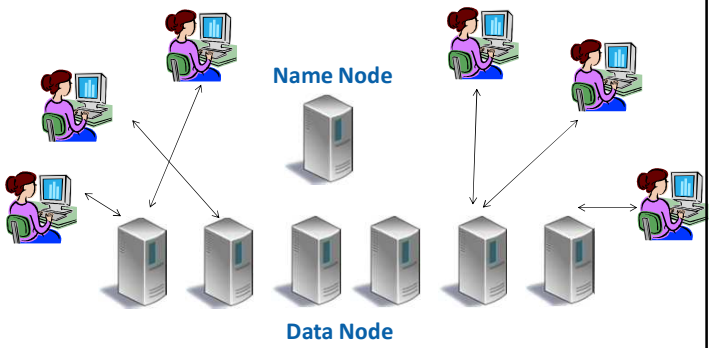


## HDFS/GFS文件操作：read




- 之后的读操作，直接与Data Node通信，绕过了Name Node
- 可以从多个副本中选择最佳的Data Node读取数据

## HDFS/GFS文件操作：read



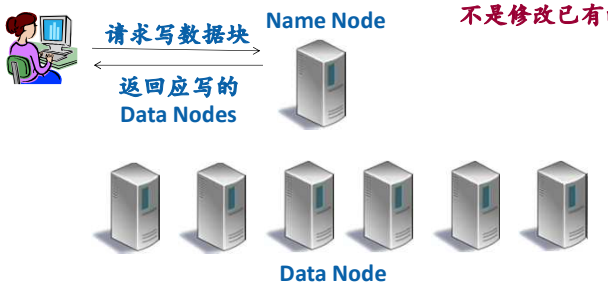

- 可以支持很多并发的读请求



## HDFS/GFS文件操作：write(1)

Client 可能是在一个Data Node 所在的机器上

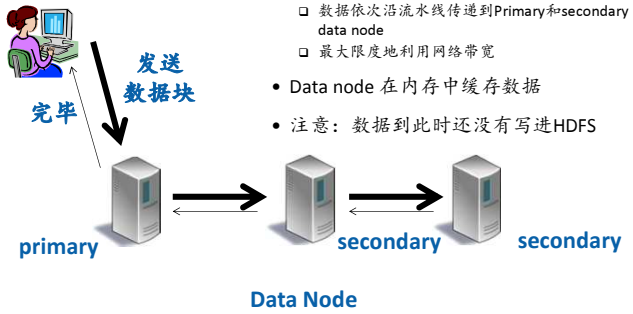
写新创建的文件
不是修改已有的文件！




- Name Node决定应该写到哪些Data Nodes
  - Rack-aware, load balancing
  - 3个副本：本机、本机柜、其它机柜



## HDFS/GFS文件操作：write(2)

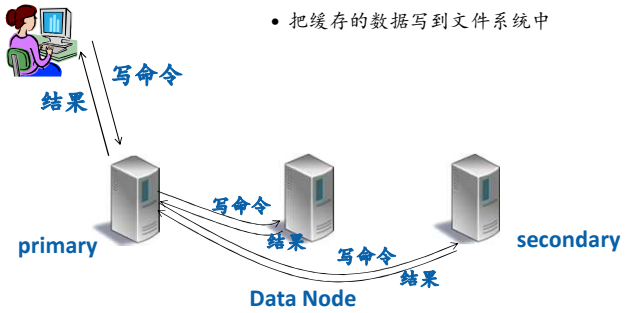



- 形成一个数据传递的pipeline
  - 数据依次沿流水线传递到Primary和secondary data node
  - 最大限度地利用网络带宽
- Data node 在内存中缓存数据
- 注意：数据到此时还没有写进HDFS

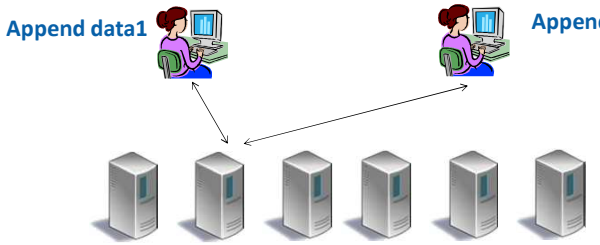


## HDFS/GFS文件操作：write(3)

- 收到写命令时才进行真正地写操作
- 把缓存的数据写到文件系统中



## HDFS/GFS文件操作：并发Append




- 文件最后一个数据块在同一个primary data node
- 可以在单机上完成concurrency control
- 保证并行append成功，但是不保证append的顺序



## HDFS/GFS小结

- 分布式文件系统
- 很好的顺序读性能
  - 为大块数据的顺序读优化
- 不支持并行的写操作：不需要distributed transaction
- 支持并行的append



## MapReduce/Hadoop简介

- MapReduce是目前云计算中最广泛使用的计算模型
  - 由Google于2004年提出
  - "*MapReduce: Simplied Data Processing on Large Clusters*". Jeffrey Dean and Sanjay Ghemawat (Google). **OSDI 2004**.
- Hadoop是MapReduce的一个开源实现
  - 2005年由Doug Cutting and Mike Cafarella开始了Hadoop项目
  - 2006年Cutting成为Yahoo Lab的员工
  - Hadoop的开发主要由Yahoo Lab推动，后来成为Apache开源项目
  - 基于Java
  - 已经被广泛使用：Yahoo, Facebook, Twitter, Linkedin, Ebay, AOL, Hulu, 百度, 腾讯, 阿里, 天涯社区, ……



## MapReduce的数据模型

- <key, value>
  - 数据由一条一条的记录组成
  - 记录之间是无序的
  - 每一条记录有一个key，和一个value
  - key: 可以不唯一
  - key与value的具体类型和内部结构由程序员决定，系统基本上把它们看作黑匣

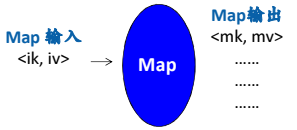
## MapReduce

Map(ik, iv) ➔{<mk, mv>}

Reduce(mk, {mv}) ➔{<ok, ov>}



## Map函数

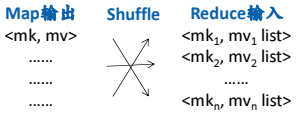



Map(ik, iv) ➔{<mk, mv>}

- 输入是一个key-value记录：<ik, iv>
  - 我们用'i'代表input
- 输出是0~多个key-value记录：<mk, mv>
  - 我们用'm'代表intermediate
- 注意：mk与ik很可能完全不同



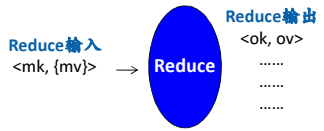
## Shuffle (由系统完成)




- Shuffle = group by mk
- 对于所有的map函数的输出，进行group by
- 将相同mk的所有mv都一起提供给Reduce



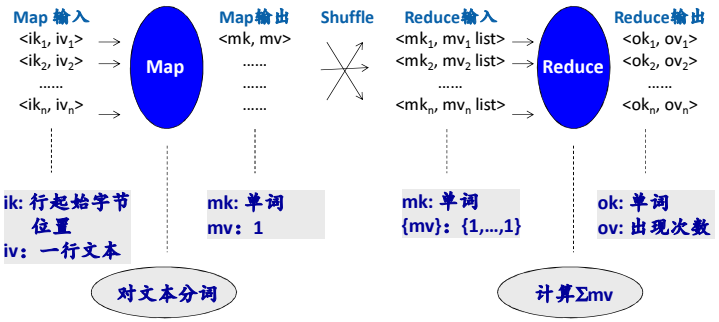
## Reduce函数




Reduce(mk, {mv}) ➔{<ok, ov>}

- 输入是一个mk和与之对应的所有mv
- 输出是0~多个key-value记录：<ok, ov>
  - 我们用'o'代表output
- 注意：ok与mk可能不同

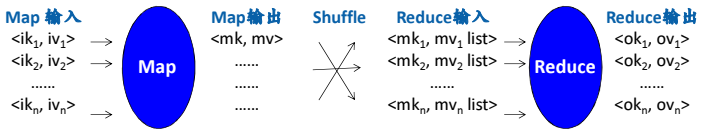
## Map-shuffle-Reduce：word count 举例




Word count: 统计文本中每个单词出现的次数



## Map-shuffle-Reduce




- 程序员编制串行的Map函数和Reduce函数
- 系统完成shuffle功能
  - shuffle = group by mk



## 比较

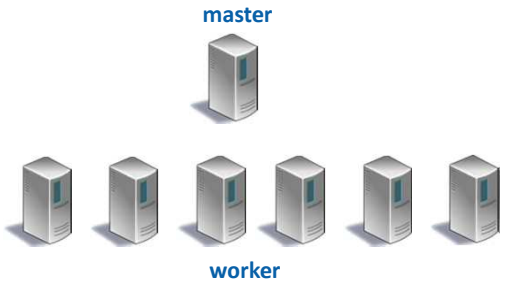
**MapReduce**
- Map
- Shuffle
- Reduce
- 选择的功能更加丰富
  - 程序实现的
  - 类似最简单的SQL select
  - 但不支持join

类似

**SQL Select**
- Selection/projection
- Group by
- Aggregation, Having
- 功能由数据类型和SQL语言标准定义
  - 有UDF: user defined function
  - 但支持得不好

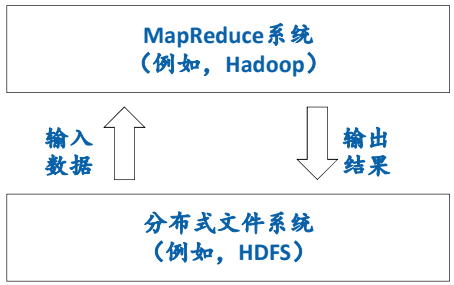


## MapReduce系统架构




在OSDI'04文章中，基本上是1个master对应100~1000数量级的workers

## 系统架构






## MapReduce / Hadoop系统架构


- 注意：JobTracker, TaskTracker, Name Node, Data Node都是进程，所以可以在一台机器上同时运行JobTracker/Name Node, TaskTracker/Data Node
- Hadoop 2.x采用YARN代替了JobTracker，但功能大同小异

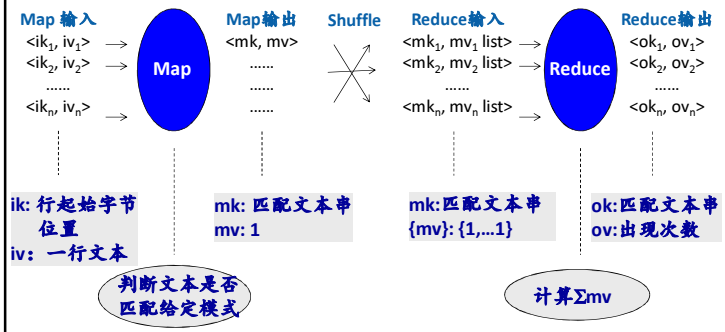


## 典型算法

- Grep
- Sorting
- Join

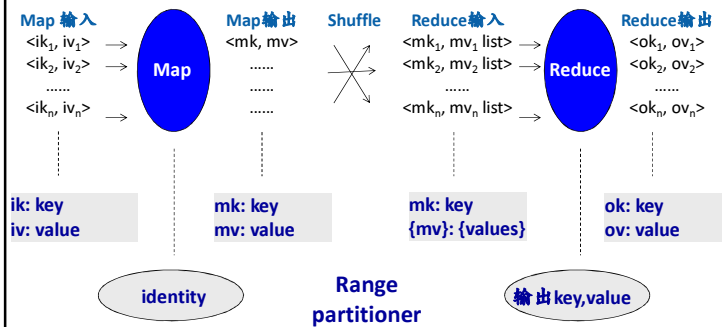


## 举例：Grep (找到符合特定模式的文本)




与word count类似

## 举例：Sorting

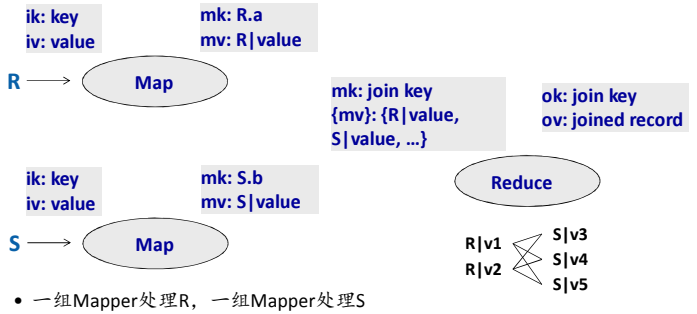



- 利用MapReduce系统的shuffle/sort功能完成sorting
- identity指将输入拷贝到输出



## 举例：Equi-Join

$R \bowtie_{R.a = S.b} S$




- 一组Mapper处理R，一组Mapper处理S
- 利用shuffle/group by把匹配的record放到一起
- Reduce调用时，{mv}包含对应同一个join key的所有匹配的R和S记录，于是产生每一对R记录和S记录的组合（笛卡尔积），并输出
- 需要传输整个R与S会产生比较大的代价



## 小结

- 大数据系统简介
  - 基本概念
  - KV键值对存储系统
  - 图数据库和图计算系统
  - 树状数据（JSON）数据库
- HDFS和MapReduce
  - 分布式文件系统HDFS
  - MapReduce