大数据处理与编程实践

Name: 傅小龙 Dept: CS

Grade: 3 ID: 191220029

一、并行计算与大数据处理技术简介

1. 为什么需要并行计算

贯穿整个计算机技术发展的核心目标：提高计算性能

1.1 提高计算机性能的主要手段

a. 提高处理器字长 4bits-8bits-16bits-32bits-64bits
b. 提高集成度。摩尔定律：18 24月计算性能提高一倍
c. 流水线等微体系结构技术
d. 提高处理器频率

1.2 单核处理器性能提升接近极限

a. VLSI集成度不可能无限制提高
b. 处理器的指令级并行度(ILP墙:Instruction Level Parallism)提升接近极限
c. 处理器速度和存储器速度差异越来越大(存储墙)
d. 功耗和散热大幅增加超过芯片承受能力(功耗墙)

2. 并行计算技术的分类

a. 按数据和指令处理结构分类：弗林分类
SISD(单指令单数据流，传统单处理器串行处理) SIMD(单指令多数据流，向量机，信号处理系统)
MISD(很少使用) MIMD(最常用)
b. 并行类型分类：位(bit)级并行、指令集并行、线程级并行（数据级并行：划分数据，任务级并行：划分任务）
c. 存储访问结构分类：共享内存、分布共享存储体系结构、分布式内存
d. 系统类型分类：多核/众核并行计算系统MC(Multicore/Manycore)-对称多处理系统SMP(Symmetric Multiprocessing)-大规模并行处理MPP-集群-网格：耦合度紧密-松散，可拓展性低-高，能耗低-高。
e. 计算特征分类：数据密集型并行计算（大规模Web 信息搜索）、计算密集型并行计算（3-D建模与渲染，气象预报，科学计算）、数据密集与计算密集混合型并行计算（3—D电影渲染）
f. 并行程序设计模型/方法分类：共享内存变量、消息传递方式、MapReduce方式

3. 并行计算的主要技术问题

- 多核/多处理器网络互连结构技术

a. 共享总线连接 （Shared Bus）
b. 交叉开关矩阵（Crossbar Switch）

c. 环形结构（Torus）
d. Mesh网络结构（Mesh Network）
e. 片上网络（NOC，Network-on-chip）

- 存储访问体系结构

a. 共享存储器体系结构(Shared Memory)：共享数据访问与同步控制
b. 分布存储体系结构(Distributed Memory)：数据通信控制和节点计算同步控制
c. 分布共享存储结构(Distributed Shared Memory)：Cache的一致性问题,数据访问/通信的时间延迟

- 分布式数据与文件管理

a. RedHat GFS (Global File System)
b. IBM GPFS
c. Sun Lustre
d. Google GFS(Google File System)
e. Hadoop HDFS(Hadoop Distributed File System)

- 并行计算任务分解与算法设计:数据划分、算法分解与设计
- 并行程序设计模型和方法

a. 共享内存式并行程序设计:为共享内存结构并行计算系统提供的程序设计方法,需提供数据访问同步控制机制（Pthread,OpenMP）
b. 消息传递式并行程序设计:为分布内存结构并行计算系统提供的、以消息传递方式完成节点间数据通信的程序设计方法
c. MapReduce并行程序设计:为解决前两者在并行程序设计上的缺陷，提供一个综合的编程框架，为程序员提供了一种简便易用的并行程序设计方法
d. 并行程序设计语言(语言级扩充-OpenMP),并行计算库函数与编程接口(MPI:消息传递接口,CUDA(NVIDIA GPU))
e. 并行编译与优化技术

- 数据同步访问和通信

a. 控制共享数据访问和同步控制L:数据访问的不确定性，用同步机制(互斥信号，条件变 量等)，死锁
b. 分布存储结构下的数据通信和同步控制:计算的同步(同步障)

- 可靠性设计与容错技术

a. 数据失效恢复:备份
b. 系统和任务失效恢复:失效检测和隔离技术

- 并行计算软件框架平台

a. 提供自动化并行处理能力
b. 高可扩展性和系统性能提升:MapReduce并行计算框架保证系统性能几乎随节点的增加线性提升

- 系统性能评价和程序并行度评估

a. 系统性能评估：标准性能评估(Benchmark)方法（浮点运算能力）。High-Performance Linpack Benchmark是最为知名的评估工具。
b. 程序并行度评估。Amdahl定律：$S = \frac{1}{(1-P)+\frac{P}{N}}$, S:加速比, P:程序可并行比, N:处理器数目。一个并行程序可加速程度是有限制的，并非可无限加速，并非处理器越多越好。

3. MPI并行程序设计(Message Passing Interface，基于消息传递的高性能并行计算编程接口)

3.1 MPI主要功能：用常规语言编程方式，所有节点运行同一个程序，但处理不同的数据。提供点对点通信(Point-point communication)：提供同步通信功能（阻塞通信）、提供异步通信功能（非阻塞通信）； 提供节点集合通信(Collective communication) 提供一对多的广播通信、提供多节点计算同步控制、提供对结果的规约(Reduce)计算功能；提供用户自定义的复合数据类型传输。

3.2 基本编程接口

通信组（Communicator）:一个处理器可以同时参加多个通信组；MPI定义了一个最大的缺省通信组: MPI_COMM_WORLD，指明系统中所有的进程都参与通信。一个通信组中的总进程数可以由MPI_Comm_Size调用来确定

进程标识:一个通信组中每个进程标识号由系统自动编号（从0开始）；进程标识号可以由MPI_Comm_Rank调用来确定。

1. MPI_Init (argc, argv)：初始化MPI，开始MPI并行计算程序体
2. MPI_Finalize: 终止MPI并行计算
3. MPI_Comm_Size(comm, size): 确定指定范围内处理器/进程数目
4. MPI_Comm_Rank(comm, rank)：确定一个处理器/进程的标识号

同步（阻塞式）通信：

5. MPI_Send (buf, count, datatype, dest, tag, comm)：发送一个消息
6. MPI_Recv (buf, count, datatype, source, tag, comm, status)：接受消息

size: 进程数，rank：指定进程的ID，omm：指定一个通信组(communicator)，Dest：目标进程号，source：源进程标识号，tag：消息标签

异步通信：

7.MPI_ISend (buf, count, datatype, dest, tag, comm, request)

8.MPI_IRecv (buf, count, datatype, source, tag, comm, status, request)

9.MPI_Wait (request, status)：等待非阻塞数据传输完成

10.MPI_Test (request, flag, status)：检查是否异步数据传输确实完成

编程示例：pptch1-64

11.MPI_Reduce(sendbuf, recvbuf, count, datatype, op, root, comm)

其中规约操作op可设为下表定义的操作之一：MPI_MAX 求最大值 MPI_MIN 求最小值 MPI_SUM 求和 MPI_PROD 求积 MPI_LAND 逻辑与 MPI_BAND 按位与 MPI_LOR 逻辑或 MPI_BOR 按位或 MPI_LXOR 逻辑异或 MPI_BXOR 按位异或 MPI_MAXLOC 最大值和位置 MPI_MINLOC 最小值和位置

3.3 MPI的特点

a. 灵活性好，适合于各种计算密集型的并行计算任务
b. 独立于语言的编程规范，可移植性好
c. 有很多开放机构或厂商实现并支持

3.3 MPI的缺点

a. 无良好的数据和任务划分支持
b. 缺少分布文件系统支持分布数据存储管理
c. 通信开销大，当计算问题复杂、节点数量很大时，难以处理，性能大幅下降
d. 无节点失效恢复机制，一旦有节点失效，可能导致计算过程无效
e. 缺少良好的构架支撑，程序员需要考虑以上所有细节问题，程序设计较为复杂

4. 4.1 大数据并行处理的重要性

a. 大数据及其处理已经成为现实世界的急迫需求
b. 大数据处理已成为公认的重大研究领域
c. 过去10多年数据急剧增长,未来急剧增长的数据将带来极大的技术挑战
d. “大数据研究的科学价值”“未来的新石油”
e. 国内外出现了“数据科学”的概念

4.2 为什么需要大数据并行处理技术?

a. 数据处理、存储、访问能力大幅落后于数据增长速度,传统关系数据库已经无法应对大数据的存储和处理
b. 大数据将带来巨大的技术和商业机遇
c. 大数据处理对未来技术和人才的需求
d. 大数据隐含着更准确的事实

4.3 什么是大数据?

大数据是指无法使用常用的软件工具在一定时间内完成获取、管理和处理的数据集。

4.4 大数据的特点

a. 大容量,TB-ZB
b. 多样性
c. 时效性
d. 准确性
e. 复杂性

4.4 大数据的类型

a. 结构特征:– 结构化数据 - 非结构化/半结构化数据
b. 获取和处理方式 – 动态(流式/增量式/线上)/实时数据 – 静态(线下数据)/非实时数据
c. 关联特征 – 无关联/简单关联数据(键值记录型数据) – 复杂关联数据(图数据)

4.5 大数据问题的基本特点

a. 大数据来自应用行业,具有极强的行业应用需求特性
b. 数据规模极大，达到PB甚至EB量级，超过任何传统数据库系统的处理能力
c. 大数据处理给传统计算技术带来极大挑战,大多数传统算法在面向大数据处理时都面临问题，需要重写

4.6 大数据研究的基本原则

a. 应用需求为导向: 以行业应用问题和需求为出发点
b. 领域交叉为桥梁：行业、IT产业、学术界协同
c. 计算技术为支撑：研究解决涉及的计算技术问题

4.7 大数据研究的基本目标

以有效的信息技术手段和计算方法，获取、处理和分析各种应用行业的大数据，发现和提取数据的内在价值，为行业提供高附加值的应用和服务

- 技术手段：信息技术和计算方法
- 核心目标：价值发现
- 效益目标：形成高附加值行业应用

4.8 大数据研究的挑战

a. 数据规模导致难以应对的存储量
b. 数据规模导致传统算法失效
c. 大数据复杂的数据关联性导致高复杂度的计算

4.9 大数据研究的基本途径

a. 寻找新算法降低计算复杂度

b. 降低大数据尺度，寻找数据尺度无关算法

c. 大数据并行化处理

从信息处理过程 获取 存储 处理 利用

从计算机技术研究与学科视角 学术角度（基础理论 算法） 业界角度（系统 应用）

从系统的观点看大数据研究层面 基础设施 系统平台 算法 应用。 大数据是诸多计算技术的融合，因此需要从信息系统视角来划分大数据研究层面，形成一个基于硬件系统的软件栈。

单一层面的研究往往难以获得理想的综合解决方案，上下层交叉组合、各层计算技术系统化综合集成以构建大数据处理的综合解决方案和平台

4.10 大数据十个典型和热点技术问题

系统层

一、 大数据存储管理和索引查询

二、 Hadoop性能优化和功能增强

三、 并行计算模型和框架

基础算法

四、 并行化机器学习和数据挖掘算法

典型/共性应用算法与技术

五、 社会网络分析

六、 Web信息搜索和排名推荐

七、 媒体分析检索

八、 基于本体的语义分析与挖掘

九、 大数据自然语言处理智能化应用

十、 大数据可视化计算与分析

4.11 2013年之后大数据研究层面内容新变化

a. 大数据查询分析计算

b. 批处理计算

c. 流式计算

e. 迭代计算

f. 图计算

g. 内存计算

5. 5.1 为什么需要MapReduce?

a. 并行计算技术和并行程序设计的复杂性,难以找到统一和易于使用的计算框架和编程模型与工具

b. 大数据处理需要有效的并行处理技术,依靠MPI等并行处理技术难以凑效

c. MapReduce是目前面向大数据处理最为成功的技术

5.2 什么是MapReduce?

a. 基于集群的高性能并行计算平台(Cluster Infrastructure)

b. 并行程序开发与运行框架(Software Framework)

c. 并行程序设计模型与方法(Programming Model & Methodology)

5.3 为什么MapReduce很重要?

a. 它提供了高效的大规模数据处理方法。

b. 改变了大规模尺度上组织计算的方式

c. 第一个不同于冯诺依曼结构的、基于集群而非单机的计算方式的重大突破，这种抽象允许我们组织计算，而不是在单个机器上，而是在整个集群上

d. 目前为止最为成功的基于大规模计算资源的并行计算抽象方法

二、MapReduce简介

1. 大规模数据处理时，MapReduce在三个层面上的基本构思

    1. 如何对付大数据处理：分而治之
    大数据分而治之的并行化计算:不可分拆的计算任务或相互间有依赖关系的数据无法进行并行计算。Master Node划分、分配任务，Worker Node负责数据块的计算。
    大数据任务划分和并行计算模型:大数据计算任务划分为子任务，执行完成后结果合并为计算结果。
    2. 上升到抽象模型：Mapper与Reducer
    主要设计思想： 为大数据处理过程中的两个主要处理操作提供一种抽象机制
    典型的流式大数据问题的特征
    Map和Reduce操作的抽象描述:提供一种抽象机制，把做什么和怎么做分开，程序员仅需要描述做 什么，不需要关心怎么做
    基于Map和Reduce的并行计算模型和计算过程:
    Map: 对一组数据元素进行某种重复式的处理 (k1; v1) → [(k2; v2)]; Reduce: 对Map的中间结果进行某种进一步的结果整理 (k2; [v2]) *rightarrow* [(k3; v3)].各个map函数对所划分的数据并行处理，从不同的输入数据产生不同的中间结果输出,各个reduce也各自并行计算，各自负责处理不同的中间 结果数据集合.进行reduce处理之前,必须等到所有的map函数做完，因此,在进入reduce前需要有一个同步障(barrier);这个阶段也负责对map的中间结果数据进行收集整理(aggregation & shuffle)处理,以便reduce更有效地计算最终结果,最终汇总所有reduce的输出结果即可获得最终结果.
    3. 上升到构架：统一构架，实现自动并行化计算,为程序员隐藏系统层细节
    主要需求、目标和设计思想:实现自动并行化计算,为程序员隐藏系统层细节
    MapReduce提供统一的构架并完成以下的主要功能:
    任务的划分和调度：提交的一个计算作业(job)将被划分为很多个计算任务(tasks),任务调度功能主要负责为这些划分后的计算任务分配和调度计算节点(map节点或reducer节点); 同时负责监控这些节点的执行状态, 并负责map节点执行的同步控制(barrier); 也负责进行一些计算性能优化处理, 如对最慢的计算任务采用多备份执行、选最快完成者作为结果

    数据/代码互定位：为了减少数据通信，一个基本原则是本地化数据处理(locality)，即一个计算节点尽可能处理其本地磁盘上所分布存储的数据，这实现了代码向数据的迁移；当无法进行这种本地化数据处理时，再寻找其它可用节点并将数据从网络上传送给该节点(数据向代码迁移)，但将尽可能从数据所在的本地机架上寻找可用节点以减少通信延迟.

    出错/失效处理：以低端商用服务器构成的大规模MapReduce计算集群中,节点硬件(主机、磁盘、内存等)出错和软件有bug是常态，因此,MapReducer需要能检测并隔离出错节点，并调度分配新的节点接管出错节点的计算任务.
    分布式数据存储与文件管理：海量数据处理需要一个良好的 分布数据存储和文件管理系统支撑,该文件系统能够把海量数 据分布存储在各个节点的本地磁盘上,但保持整个数据在逻辑 上成为一个完整的数据文件；为了提供数据存储容错机制,该 文件系统还要提供数据块的多备份存储管理能力

    结果数据的收集整理:（Combiner和Partitioner（设计目的和作用:为了减少数据通信开销,中间结果数据进入reduce节点前需要进行合并(combine)处理,把具有同样主键的数据合并到一起避免重复传送; 一个reducer节点所处理的数据可能会来自多个map节点, 因此, map节点输出的中间结果需使用一定的策略进行适当的划分(partitioner)处理，保证相关数据发送到同一个reducer节点））

    [处理数据与计算任务的同步,系统通信、负载平衡、计算性能优化处理，处理系统节点出错检测和失效恢复]...

最大亮点：把做什么和怎么做分开。程序员仅需要关心其应用层的具体计算问题。

4. MapReduce的主要设计思想和特征
   a. 向“外”横向扩展，而非向“上”纵向扩展:即MapReduce集群的构筑选用价格便宜、易于扩展的大量低端商用服务器，而非价格昂贵、不易扩展的高端服务器（SMP）.
   b. 失效被认为是常态:一个良好设计、具有容错性的并行计算系统不能因为节点失效而影响计算服务的质量.MapReduce并行计算软件框架使用了多种有效的机制，如节点自动重启技术，使集群和计算框架具有对付节点失效的健壮性，能有效处理失效节点的检测和恢复。
   c. 把处理向数据迁移:大规模数据处理时外存文件数据I/O访问会成为一个制约系统性能的瓶颈。MapReduce采用了数据/代码互定位的技术方法（数据本地化）
   d. 顺序处理数据、避免随机访问数据:MapReduce设计为面向大数据集批处理的并行计算系统，所有计算都被组织成很长的流式操作，以便能利用分布在集群中大量节点上磁盘集合的高传输带宽。
   e. 为应用开发者隐藏系统层细节
   f. 平滑无缝的可扩展性:数据扩展和系统规模扩展:理想的软件算法应当能随着数据规模的扩大而表现出持续的有效性，性能上的下降程度应与数据规模扩大的倍数相当; 在集群规模上，要求算法的计算性能应能随着节点数的增加保持接近线性程度的增长.

三、Google /Hadoop MapReduce基本构架

1. MapReduce的基本模型和处理思想

   同二、1.

2. Google MapReduce的基本工作原理

   2.1 Google MapReduce并行处理的基本过程

   a.有一个待处理的大数据，被划分为大小相同的数据块(如64MB),及与此相应的用户作业程序.

   b.系统中有一个负责调度的主节点(Master),以及数据Map和Reduce工作节点(Worker)

   c.用户作业程序提交给主节点

   d.主节点为作业程序寻找和配备可用的Map节点，并将程序和数据传送给map节点

   e.主节点也为作业程序寻找和配备可用的Reduce节点，并将程序传送给Reduce节点

   f.主节点启动每个Map节点执行程序，每个map节点尽可能读取本地或本机架的数据进行计算

   g.每个Map节点处理读取的数据块,并做一些数据整理工作(combining, sorting等)并将中间结果存放在本地；同时通知主节点计算任务完成并告知中间结果数据存储位置

   h.主节点等所有Map节点计算完成后，开始启动Reduce节点运行；Reduce节点从主节点所掌握的中间结果数据位置信息，远程读取这些数据

   i.Reduce节点计算结果汇总输出到一个结果文件即获得整个处理结果

   2.2 带宽优化（Combiner的设计目的和作用）

   问题：大量的键值对数据在传送给Reduce节点时会引起较大的通信带宽开销

   解决方案：每个Map节点处理完成的中间键值队将由combiner做一个合并压缩，即把那些键名相同的键值对归并为一个键名下的一组数值。

   2.3 用数据分区解决数据相关性问题（Partitioner的设计目的和作用）

   问题：一个Reduce节点上的计算数据可能会来自多个Map节点，因此，为了在进入Reduce节点计算之前，需要把属于一个Reduce节点的数据归并到一起。

   解决方案：在Map阶段进行了Combining以后，可以根据一定的策略对Map输出的中间结果进行分区(partitioning)，这样即可解决以上数据相关性问题避免Reduce计算过程中的数据通信。

   2.4 失效处理

   a. 主节点失效：主节点中会周期性地设置检查点(checkpoint)，检查整个计算作业的执行情况，一旦某个任务失效，可以从最近有效的检查点开始重新执行，避免从头开始计算的时间浪费。

   b. 工作节点失效：工作节点失效是很普遍发生的，主节点会周期性地给工作节点发送心跳检测，如果工作节点没有回应，则认为该工作节点失效，主节点将终止该工作节点的任务并把失效的任务重新调度到其它工作节点上重新执行

   2.5 计算优化

   问题：Reduce节点必须要等到所有Map节点计算结束才能开始执行，因此，如果有一个计算量大、或者由于某个问题导致很慢结束的Map节点，则会成为严重的“拖后腿者”。

   解决方案：把一个Map计算任务让多个Map节点同时做，取最快完成者的计算结果

3. 分布式文件系统GFS的基本工作原理

   基本问题：海量数据怎么存储？数据存储可靠性怎么解决?

   当前主流的分布式文件系统主要用于对硬件设施要求很高的高性能计算或大型数据中心,有：RedHat的GFS IBM的GPFS Sun的Lustre等

   3.1 Google GFS的基本设计原则

   a. 廉价本地磁盘分布存储

b. 多数据自动备份解决可靠性

c. 为上层的MapReduce计算框架提供支撑：负责处理所有的数据自动存储和容错处理.

3.2 Google GFS的基本构架和工作原理

3.2.1 GFS Master的主要作用

Master上保存了GFS文件系统的三种元数据 ：命名空间(Name Space),即整个 分布式文件系统的目录结构; Chunk与文件名的映射表; Chunk副本的位置信息，每一个Chunk默认有3个副本.

前两种元数据可通过操作日志提供容错处理能力；第3个元数据直接保存在ChunkServer上，Master启动或Chunk Server注册时自动完成在Chunk Server上元数据的生成；因此，当Master失效时，只要ChunkServer数据保存完好，可迅速恢复Master上的元数据。

3.2.2 GFS ChunkServer的主要作用

用来保存大量实际数据的数据服务器。GFS中每个数据块划分缺省为64MB.每个数据块会分别在3个(缺省情况下)不同的地方复制副本；对每一个数据块，仅当3个副本都更新成功时，才认为数据保存成功。当某个副本失效时，Master会自动将正确的副本数据进行复制以保证足够的副本数;GFS上存储的数据块副本，在物理上以一个本地的Linux操作系统的文件形式存储，每一个数据块再划分为64KB的子块，每个子快有一个32位的校验和，读数据时会检查校验和以保证使用为失效的数据。

3.3.3 数据访问工作过程

a. 在程序运行前，数据已经存储在GFS文件系统中；程序实行时应用程序会告诉GFS Server所要访问的文件名或者数据块索引是什么

b. GFS Server根据文件名会数据块索引在其文件目录空间中查找和定位该文件或数据块，并找数据块在具体哪些ChunkServer上；将这些位置信息回送给应用程序

c. 应用程序根据GFSServer返回的具体Chunk数据块位置信息，直接访问相应的Chunk Server

d. 应用程序根据GFSServer返回的具体Chunk数据块位置信息直接读取指定位置的数据进行计算处理

特点：应用程序访问具体数据时不需要经过GFS Master，因此，避免了Master成为访问瓶颈;并发访问：由于一个大数据会存储在不同的ChunkServer中，应用程序可实现并发访问

3.3.4 GFS的系统管理技术

a. 大规模集群安装技术：如何在一个成千上万个节点的集群上迅速部署GFS，升级管理和维护等

b. 故障检测技术：GFS是构建在不可靠的廉价计算机之上的文件系统，节点数多，故障频繁，如何快速检测、定位、恢复或隔离故障节点

c. 节点动态加入技术：当新的节点加入时，需要能自动安装和部署GFS

d. 节能技术：服务器的耗电成本大于购买成本，Google为每个节点服务器配置了蓄电池替代UPS，大大节省了能耗

4. BigTable的基本作用和设计思想

GFS是一个文件系统，难以提供对结构化数据的存储和访问管理。为此，Google在GFS之上又设计了一个结构化数据存储和访问管理系统—BigTable，为应用程序提供比单纯的文件系统更方便、更高层的数据操作能力

BigTable提供了一定粒度的结构化数据操作能力，主要解决一些大型媒体数据（Web文档、图片等）的结构化存储问题。但与传统的关系数据库相比，其结构化粒度没有那么高，也没有事务处理等能力，因此，它并不是真正意义上的数据库。

4.1 BigTable设计动机

a. 存储管理海量的结构化半结构化数据

b. 海量的服务请求

c. 商用数据库无法适用

4.2 BigTable设计目标

a. 广泛的适用性

b. 很强的可扩展性

c. 高吞吐量数据访问

d. 高可用性和容错性

e. 自动管理能力

f. 简单性

4.3 BigTable数据模型—多维表

表中的数据通过：行关键字(row key)列关键字(column key)时间戳(time stamp)进行索引和查询定位

(row:string, column:string,time:int64)*rightarrow* 结果数据字节串

表中数据一律视为字节串（bytes）

行(Row):大小不超过64KB的任意字节串。表中的数据都是根据行关键字进行排序的

子表(Tablet)：一个大表可能太大，不利于存储管理，将在水平方向上被分为多个子表

列(Column): BigTable将列关键字组织成为"列族"(column family),每个族中的数据属于同一类别.一个列族下的数据会被压缩在一起存放（按列存放）。因此,一个列关键字可表示为: 族名：列名(family:qualifier)

4.4 BigTable基本构架

4.4.0 主服务器

新子表分配（创建新表、表合并及较大子表的分裂都会产生新的子表）

子表监控：通过Chubby完成。所有子表服务器基本信息被保存在Chubby中的服务器目录中主服务器检测这个目录可获取最新子表服务器的状态信息。当子表服务器出现故障，主服务器将终止该子表服务器，并将其上的全部子表数据移动到其它子表服务器。

负载均衡

4.4.1 子表服务器：BigTable中的数据都以子表形式保存在子表服务器上，客户端程序也直接和子表服务器通信。分配：当一个新子表产生时子表服务器的主要问题包括子表的定位、分配、及子表数据的最终存储。

子表的基本存储结构SSTable:以GFS文件形式存储在GFS文件系统中。一个SSTable实际上对应于GFS中的一个64MB的数据块(Chunk).SSTable中的数据进一步划分为64KB的子块，因此一个SSTable可以有多达1千个这样的子块。为了维护这些子块的位置信息，需要使用一个Index索引.

子表数据格式：概念上子表是整个大表的多行数据划分后构成。而一个子表服务器上的子表将进一步由很多个SSTable构成，每个SSTable构成最终的在底层GFS中的存储单位。一个SSTable还可以为不同的子表所共享，以避免同样数据的重复存储。

子表寻址：子表地址以3级B+树形式进行索引；首先从Chubby服务器中取得根子表，由根子表找到二级索引子表，最后获取最终的SSTable的位置

5. Hadoop 分布式文件系统HDFS

5.1 HDFS的基本特征

模仿Google GFS设计实现. 存储极大数目的信息（terabytes or petabytes），将数据保存到大量的节点当中；支持很大的单个文件. 提供数据的高可靠性和容错能力,通过一定数量的数据复制保证数据存储的可靠性和出错恢复能力。提供对数据的快速访问；并提供良好的可扩展性，通过简单加入更多服务器快速扩充系统容量，服务

更多的客户端。与GFS类似，HDFS是MapReduce的底层数据存储支撑，并使得数据尽可能根据其本地局部性进行访问与计算。

HDFS对顺序读进行了优化，支持大量数据的快速顺序读出，代价是对于随机的访问负载较高。数据支持一次写入，多次读取；不支持已写入数据的更新操作，但允许在文件尾部添加新的数据。数据不进行本地缓存（文件很大，且顺序读没有局部性）。基于块的文件存储，默认的块的大小是64MB：减少元数据的量，有利于顺序读写（在磁盘上数据顺序存放）。多副本数据块形式存储，按照块的方式随机选择存储节点，默认副本数目是3

5.2 HDFS基本构架

NameNode:对等于Google MapReduce 中MasterServer

DataNode:对等于Google MapReduce 中ChunckServer

5.3 HDFS数据分布设计

多副本数据块形式存储，按照块的方式随机选择存储节点.默认副本数目是3.

5.4 HDFS可靠性与出错恢复

DataNode节点的检测: 心跳：NameNode 不断检测DataNode是否有效; 若失效，则寻找新的节点替代，将失效节点数据重新分布.

集群负载均衡

数据一致性: 校验和checksum

主节点元数据失效:NameNode依靠 Multiple FsImage(文件系统状态) and EditLog(没有更新的记录)和Checkpoint.

安全模式: 刚启动的时候，等待每一个DataNode报告情况; 退出安全模式的时候才进行副本复制操作.

5.5 HDFS的安装和启动

5.6 HDFS文件系统操作命令

hdfs dfs -mkdir /user

-put, -ls, -du(disk usage), -mv(move) -cp, -rm, -get(hdfs to local), -cat, -tail(show last 1kb of file)

-safemode enter/leave/get/wait

dfsadmin -...

6. Hadoop MapReduce的基本工作原理

6.1 Hadoop MapReduce主要组件

文件输入格式InputFormat: TextInputFormat KeyValueTextInputFormat SequenceFileInputFormat

输入数据分块InputSplits:InputSplit定义了输入到单个Map任务的输入数据.一个MapReduce程序被统称为一个Job，可能有上百个任务构成.InputSplit将文件分为64MB的大小

数据记录读入RecordReader:nputSplit定义了一个数据分块，但是没有定义如何读取数据记录,RecordReader实际上定义了如何将数据记录转化为一个(key,value)对的详细方法，并将数据记录传给Mapper类.TextInputFormat提供了LineRecordReader，读入一个文本行数据记录

Mapper: 每一个Mapper类的实例生成了一个Java进程，负责处理某一个InputSplit上的数据.有两个额外的参数OutputCollector以及Reporter，前者用来收集中间结果，后者用来获得环境参数以及设置当前执行的状态。

Combiner:合并相同key的键值对，减少partitioning时候的数据通信开销.是在本地执行的一个Reducer，满足一定的条件才能够执行。

Partitioner & Shuffle:在Map工作完成之后，每一个 Map函数会将结果传到对应的Reducer所在的节点，此时，用户可以提供一个Partitioner类，用来决定一个给

定的(key,value)对传给哪个Reduce节点

Sort:传输到每一个Reducer节点上的、将被所有的Reduce函数接收到的Key,value对会被Hadoop自动排序
Reducer:做用户定义的Reduce操作.接收到一个OutputCollector的类作为输出

文件输出格式OutputFormat: TextOutputFormat, SequenceFileOutputFormat, NullOutputFormat.写入到HDFS的所有OutputFormat都继承自FileOutputFormat. 每一个Reducer都写一个文件到一个共同的输出目录，文件名是part-nnnnn，其中nnnnn是与每一个reducer相关的一个号（partition id）

RecordWriter:TextOutputFormat实现了缺省的LineRecordWriter，以"key^value"形式输出一行结果。

4.2 Hadoop MapReduce 工作过程

a.客户端MapReduce程序run job提交给JobClient
b.JobClient获取新的job ID发送给JobTracker
c.JobClient拷贝job资源到HDFS
d.JobClient提交job至JobTracker
e.JobTracker初始化job
f.JobTracker从HDFS获取输入分片(input splits)
g.TaskTracker给JobTracker发送心跳包
h.TaskTracker从HDFS获取job资源
i.TaskTracker启动任务，在JVM上创建子进程
j. 子进程运行MapTask或ReduceTask

6.3 容错处理与计算性能优化

由Hadoop系统自己解决,主要方法是将失败的任务进行再次执行.TaskTracker会把状态信息汇报给JobTracker，最终由JobTracker决定重新执行哪一个任务.为了加快执行的速度，Hadoop也会自动重复执行同一个任务，以最先执行成功的为准（投机执行）

7. Hadoop 分布式文件系统HDFS编程

ppt ch3 77-86

四、Hadoop系统安装运行与程序开发

1. Hadoop安装方式
    a. 单机方式
    b. 单机伪分布方式：用不同java进程模拟分布运行的NameNode, DataNde, JobTracker, TaskTracker等各类结点。
    c. 集群分布模式
2. 单机/伪分布式Hadoop系统安装基本步骤
    a. 安装JDK
    b. 下载安装Hadoop
    c. 配置SSH
    d. 配置Hadoop环境
    e. 格式化HDFS文件系统 hadoop namenode -format
    f. 启动Hadoop环境
    g. 运行程序测试
    h. 查看集群状态
3. 集群Hadoop系统安装基本步骤

    同2. 在完成主节点Hadoop系统安装后将主节点的Hadoop系统目录复制到每一个从节点避免重复安装操作。
4. Hadoop集群远程作业提交与执行

    4.1 程序开发与提交作业基本过程
        a. 在本地编写程序和调试
        b. 上传程序与数据
        c. 在集群上完成计算作业
        d. 查看运行结果

    4.2 集群分布方式下远程提交作业
        a. 在本地编写程序和调试
        b. 创建用户账户
        c. 上传程序与数据到集群
        d. 用SSH命令远程登录到Hadoop集群
        e. 将数据复制到HDFS中
        f. 在集群上用hadoop jar命令完成计算作业
        g. 查看运行结果
5. Hadoop MapReduce程序开发

    ch4 ppt 37-88

五、MapReduce算法设计

1. MapReduce可解决哪些算法问题?

   基本算法：各种全局数据相关性小、能适当划分数据的计算任务，如：分布式排序、分布式GREP(文本匹配查找)、关系代数操作、矩阵向量相乘、矩阵相乘、词频统计、词频重要性分析、单词同现关系分析、文档倒排索引.....

   复杂算法或应用：Web搜索、Web访问日志分析、数据/文本统计分析、图算法聚类、相似性比较分析算法、基于统计的文本处理、机器学习、数据挖掘、统计机器翻译、生物信息处理、广告推送与推荐、基于大数据集的机器学习和自然语言处理算法、机器学习与数据挖掘并行算法、大规模重复文档检测并行算法、大规模长基因序列比对算法

2. 回顾：MapReduce流水线

   a. $map(K1, V1) -> [(K2, V2)]$

   b. shuffle and sort

   c. $reduce(K2, [V2]) -> [(K3, V3)]$

3. MapReduce排序算法

   Hadoop lib 中有一个class TotalOrderPartitioner: 提供全序划分

   避免在某些Reducer上聚集太多的数据？大量key分配到多个partition时如何高效找到每个key所属的partition?

   a. 通过采样获取数据分布：获取数据分布作均匀划分。由于Key分布未知，预读一小部分数据采样，对采样数据排序后划分: $sample[i-1] <= key < sample[i]$的key划分到同一reducer.

   b. 构建高效的划分模型：若Key的数据类型可以按字节比较大小(如Text),可以狗仔三叉树；否则按二分查找确定key的所属区间。

4. MapReduce单词同现分析算法

   矩阵元素M[i, j]代表单词W[i]与单词W[j]在一定范围内同现的次数（一个语句中，一个段落中，一篇文档中，或文本串中一个宽度为M个单词的窗口中，这些都依具体问题而定）

   同现矩阵的空间开销为$O(n^2)$

   1: class Mapper

   2: method Map(docid a, doc d)

   3: for all term w $\in$ doc d do

   4: for all term u $\in$ Neighbors(w) do

   5: //Emit count for each co-occurrence

   Emit(pair (w, u), count 1)

   1: class Reducer

   2: method Reduce(pair p; counts [c1, c2,...])

   3: s $\leftarrow$ 0

   4: for all count c in counts [c1, c2,...] do

   5: s $\leftarrow$ s + c //Sum co-occurrence counts

   6: Emit(pair p, count s)

   根据同现关系的不同，可能需要实现和定制不同的 FileInputFormat和RecordReader.《深入理解大数据》P266

   类似应用问题：零售商通过分析大量的交易记录，识别出关联的商品购买行为（如："啤酒和纸尿裤"的故事）；从生物医学文献中自动挖掘基因交互作用关系

5. MapReduce文档倒排索引算法

```
public class InvertedIndexMapper extends Mapper<Text, Text, Text,
    Text> {
        @Override
        protected void map(Text key, Text value, Context context)
            throws IOException, InterruptedException // default
              RecordReader: LineRecordReader; key: line offset; value: line string
        {
                FileSplit fileSplit = (FileSplit)context.
                    getInputSplit();
                String fileName = fileSplit.getPath().getName();
                Text word = new Text();
                Text fileName_lineOffset = new Text(fileName+"#"+
                    key.toString());
                StringTokenizer itr = new StringTokenizer(value.
                    toString());
                for(; itr.hasMoreTokens(); ) {
                        word.set(itr.nextToken());
                        context.write(word, fileName_lineOffset);

                }
        }
}

public class InvertedIndexReducer extends Reducer<Text, Text,
    Text, Text>
{
        @Override protected void reduce(Text key, Iterable<Text>
            values, Context context) throws IOException,
            InterruptedException
        {
                Iterator<Text> it = values.iterator();
                StringBuilder all = new StringBuilder();
                if(it.hasNext()) all.append(it.next().toString())
                    ;
                for(; it.hasNext(); ) {
                        all.append(";");
                        all.append(it.next().toString());
                }
                context.write(key, new Text(all.toString()));
        }
//最终输出键值对示例：("fish", "doc1#0; doc1#8;doc2#0;doc2#8 ")
}
public class InvertedIndexer {
        public static void main(String[] args) {
        try {
                        Configuration conf = new Configuration();
                        job = new Job(conf, "invert_index");
                        job.setJarByClass(InvertedIndexer.class);
                        job.setInputFormatClass(TextInputFormat.
                            class);
                        job.setMapperClass(InvertedIndexMapper.
                            class);
                        job.setReducerClass(InvertedIndexReducer.
                            class);
                        job.setOutputKeyClass(Text.class);
```

```
42                          job.setOutputValueClass(Text.class);
43                          FileInputFormat.addInputPath(job, new
                                Path(args[0]));
44                          FileOutputFormat.setOutputPath(job, new
                                Path(args[1]));
45                          System.exit(job.waitForCompletion(true) ?
                                0 : 1);
46                  }
47                  catch (Exception e) {
48                          e.printStackTrace();
49                  }
50          }
51  }
```

带词频属性的文档倒排算法

1: class Mapper
2: procedure Map(docid dn, doc d)
3: F ← new AssociativeArray
4: for all term t ∈ doc d do
5: Ft ← Ft + 1
6: for all term t ∈ F do
7: Emit(term t, posting $< dn, Ft >$)
1: class Reducer
2: procedure Reduce(term t, postings $[< dn1, f1 >, < dn2, f2 > \ldots]$)
3: P ← new List
4: for all posting $< dn, f >$ ∈ postings $[< dn1, f1 >, < dn2, f2 > \ldots]$ do
5: Append(P, $< dn, f >$)
6: Sort(P)
7: Emit(term t; postings P)

可扩展的带词频属性的文档倒排算法

1: class Mapper
2: method Map(docid dn; doc d)
3: F ← new AssociativeArray
4: for all term t ∈ doc d do
5: Ft ← Ft + 1
6: for all term t ∈ F do
7: Emit(tuple$< t, dn >$, tf Ft)
1: class Reducer
2: method Setup // 初始化
3: tprev ← Ø;
4: P ← new PostingsList
5: method Reduce(tuple $< t, n >$, tf [f])
6: if t ≠ tprev ∧ tprev ≠ Ø then
7: Emit(tprev, P)
8: P.Reset()

9: P.Add($< n, f >$)

10: tprev ← t

11: method Close

12: Emit(t, P)

问题：当对键值对进行shuffle处理以传送给合适的Reducer时，将按照新的键$< t, dn >$进行排序和选择Reducer，因而同一个term的键 值对可能被分区到不同的Reducer！ 解决方案：定制Partitioner来解决这个问题:

```
Class NewPartitioner extends HashPartitioner<K,V>
    //org.apache.hadoop.mapreduce.lib.partition.HashPartitioner {
// override the method
        getPartition(K key, V value, int numReduceTasks)
        {
                term = key.toString().split(",")[0];     //¡term, docid¿=¿term
                super.getPartition(term, value, numReduceTasks);
        }
}
Job.setPartitionerClass(NewPartitioner)
```

6. 专利文献数据分析

专利被引列表

```
public static class MapClass extends Mapper<LongWritable, Text,
    Text, Text> {
        public void map(LongWritable key, Text value, Context
            context) throws IOException, InterruptedException
        // 输入key: 行偏移值; value: "citing专利号, cited专利号" 数据对
        {
                String[] citation = value.toString().split(",");
                context.write(new Text(citation[1]), new Text(
                    citation[0]));
        } // 输出key: cited 专利号; value: citing专利号
}
public static class ReduceClass extends Reducer<Text, Text, Text,
     Text> {
        public void reduce(Text key, Iterable<Text> values,
            Context context) throws IOException,
            InterruptedException {
                String csv = "";
                for (Text val:values) {
                if (csv.length() > 0)
                        csv += ",";
                csv += val.toString();
                }
                context.write(key, new Text(csv));
        } // 输出key: cited专利号; value: "citing专利号1, citing专利号2,..."
}
```

专利被引次数统计

```
        public static class MapClass extends Mapper<LongWritable,
            Text, Text, IntWritable> {
        IntWritable one = new IntWritable(1);
```

```
3           public void map(LongWritable key, Text value, Context
                context) throws IOException, InterruptedException {
4           // 输入key: 行偏移值; value: "citing专利号, cited专利号" 数据对
5           String[] citation = value.toString().split(",");
6           context.write(new Text(citation[1]), one);
7           } // 输出key: cited 专利号; value: 1
8   }
9   public static class ReduceClass extends Reducer<Text, IntWritable
        , Text, Text> {
10          public void reduce(Text key, Iterable<IntWritable> values
                , Context context) throws IOException,
                InterruptedException {
11                  int count = 0;
12                  while (values.hasNext())
13                          count += values.next().get();
14                  context.write(key, new IntWritable(count));
15          } // 输出key: 被引专利号; value: 被引次数
16  }
```

专利被引次数直方图统计：扫描专利被引次数统计的被引次数统计数据，忽略每一行中的专利号，仅考虑右侧的被引次数，看每种被引次数分别有多少次出现。

```
1   public static class MapClass extends Mapper<Text, Text,
        LongWritable, LongWritable> {
2           private final static IntWritable uno = new IntWritable(1)
                ;
3           private IntWritable citationCount = new IntWritable();
4           public void map(Text key, Text value, Context context)
                throws IOException, InterruptedException {
5                   citationCount.set(Integer.parseInt(value.toString
                        ()));
6                   context.write (citationCount, uno);
7           }
8   }
9   public static class ReduceClass extends Reducer <IntWritable,
        IntWritable,IntWritable,IntWritable> {
10   public void reduce(IntWritable key, Iterable<IntWritable> values
         , Context context) throws IOException, InterruptedException
         {
11          int count = 0;
12          while (values.hasNext())
13                  count += values.next().get();
14          context.write(key, new IntWritable(count));
15          } // 输出key: 被引次数; value: 总出现次数
16  }
17  job.setInputFormat(KeyValueTextInputFormat.class);
```

年份或国家专利数统计:扫描专利描述数据集，根据要统计的 列名(年份或国家等），取出对应列上的年份(col_idx=1)或国家 (col_idx=4)，然后由Map发出（year，1）或（country，1），再由 Reduce累加.

六、HBase与Hive程序设计

1. HBase基本工作原理

1.1 关系数据库的理论局限性

1.2 RDBMS的实现局限性

1.3 HBase的设计目标和功能特点

a. 针对HDFS缺少结构化半结构化数据存储访问能力的缺陷，提供一个分布式数据管理系统，解决大规模的结构化和半结构化数据存储访问问题.

b. 提供基于列存储模式的大数据表管理能力

c. 可存储管理数十亿以上的数据记录，每个记录可包含百万以上的数据列

d. HBase试图提供随机和实时的数据读写访问能力

e. 具有高可扩展性、高可用性、容错处理能力、负载平衡能力、以及实时数据查询能力

f. 可与MapReduce协同工作，为MapReduce提供数据输入输出，以完成数据的并行化处理

1.4 HBase数据模型

a. 逻辑数据模型
数据存储逻辑模型与BigTable类似,但实现上有一些不同之处。是一个分布式多维表，表中的数据通过：一个行关键字(row key)一个列关键字(column key)一个时间戳(time stamp)进行索引和查询定位的。

b. 物理存储格式
按照列存储的稀疏行/列矩阵。物理存储格式上按逻辑模型中的行进行分割，并按照列族存储。值为空的列不予存储，节省存储空间

1.5 HBase的基本构架

由一个MasterServer和由一组子表数据区服务器RegionServer构成，分别存储逻辑大表中的部分数据.大表中的底层数据存于HDFS中

1.6 HBase数据存储管理

HBase子表数据存储与子表服务器：每个子表中的数据区Region由很多个数据存储块Store构成，而每个Store数据块又由存放在内存中的memStore和存放在文件中的StoreFile构成

HBase数据的访问：当客户端需要进行数据更新时，先查到子表服务器,然后向子表提交数据更新请求。提交的数据并不直接存储到磁盘上的数据文件中，而是添加到一个基于内存的子表数据对象 memStore中，当memStore中的数据达到一定大小时，系统将 自动将数据写入到文件数据块StoreFile中。每个文件数据块StoreFile最后都写入到底层基于HDFS的文件中。需要查询数据时，子表先查memStore。如果没有，则再查磁盘上的StoreFile。每个StoreFile都有类似B树的结构，允许进行快速的数据查询。StoreFile将定时压缩，多个压缩为一个。两个小的子表可以进行合并。子表大到超过某个指定值时，子表服务器就需要把它分割为两个新的子表。

HBase子表服务器与主服务器：

HBase主服务器HServer:与BigTable类似，HBase使用主服务器HServer来管理所有子表服务器。主服务器维护所有子表服务器在任何时刻的状态。当一个新的子表服务器注册时，主服务器让新的子表服务器装载子表。若主服务器与子表服务器连接超时，那么子表服务器将自动停止，并重新启动；而主服务器则假定该子表服务器已死机，将其上的数据转移至其它子表服务器，将其上的子表标注为空闲，并在重新启动后另行分配使用。

HBase数据记录的查询定位：描述所有子表和子表中数据块的元数据都存放在专门的元数据表中,并存储在特殊的子表中。子表元数据会不断增长，因此会使用多个子表来保存。所有元数据子表的元数据都保存在根子表中。主服务器会扫描根子表，从

而得到所有的元数据子表位置，再进一步扫描这些元数据子表即可获得所寻找子表的位置。

HBase使用三层类似B+树的结构来保存region位置：第一层是保存zookeeper里面的文件，它持有root region的位置。第二层root region是.META.表的第一个region，其中保存了.META.表其它region的位置。通过root region，我们就可以访问.META.表的数据。.META.是第三层，它是一个特殊的表，保存了HBase中所有数据表的region位置信息。

HBase数据记录的查询定位：元数据子表采用三级索引结构：根子表fl >用户表的元数据表fl >用户表

2. HBase基本操作与编程方法

2.1 创建表：create '(*namespace* = default):*tableName*' [,'*columnFamilyName*']

2.2 插入数据：put '(*namespace* = default):*tableName*', '*rowkey*', '*columnFamliy*:*columnName*', '*value*'

2.3 显示描述表信息：discribe '*tableName*'

2.4 扫描数据：scan '*tableName*'

2.5 限制列进行扫描：scan '*tableName*', {COLUMNS=>'*ColumnName*'}

2.6 HBase中的disable和enable：disable和enable都是HBase中比较常见的操作，很多对table的修改都需要表在disable的状态下才能进行。disable 'students' 将表students的状态更改为disable的时候，HBase会在zookeeper中的table结点下做记录。在zookeeper记录下修改该表的同时，还会将表的region全部下线，region为offline状态。enable的过程和disable相反，会把表的所有region上线，并删除zookeeper下的标志。如果在enable前，META中有region的server信息，那么此时会在该server上将该region上线；如果没有server的信息，那么此时还要随机选择一台机器作为该region的server。

3. HBase的Java编程

3.1 创建表：HTableDescriptor代表的是表的schema，HColumnDescriptor代表的是column的schema

```
1  HBaseAdmin hAdmin = new HBaseAdmin(hbaseConfig);
2  HTableDescriptor t = new HTableDescriptor(tableName);
3  t.addFamily(new HColumnDescriptor("f1"));
4  t.addFamily(new HColumnDescriptor("f2"));
5  t.addFamily(new HColumnDescriptor("f3"));
6  t.addFamily(new HColumnDescriptor("f4"));
7  hAdmin.createTable(t);
```

3.2 插入数据：Put对象或者List put对象分别实现单条插入和批量插入

```
1  public static void addData (String tableName, String rowKey,
       String family, String qualifier, String value) throws
       Exception {
2          try {
3                  HTable table = new HTable(conf, tableName);
4                  Put put = new Put(Bytes.toBytes(rowKey));
5                  put.add(Bytes.toBytes(family), Bytes.toBytes(
                       qualifier), Bytes.toBytes(value));
6                  table.put(put);
7                  System.out.println("insert recored success!")
                       ;
8          }
9          catch (IOException e) { e.printStackTrace(); }
10 }
```

3.3 删除表：

```
HBaseAdmin hAdmin = new HBaseAdmin(hbaseConfig);
if (hAdmin.tableExists(tableName)) {
        hAdmin.disableTable(tableName);
        hAdmin.deleteTable(tableName);
}
```

3.4 查询数据：单条查询是通过rowkey在table中查询某一行的数据。HTable 提供了get方法来完成单条查询。批量查询是通过制定一段rowkey的范围来查询。HTable提 供了个getScanner方法来完成批量查询。

```
Scan s = new Scan();
s.setMaxVersions();
ResultScanner ss = table.getScanner(s);
for(Result r:ss){
        System.out.println(new String(r.getRow()));
        for(KeyValue kv:r.raw())
                System.out.println(new String(kv.getColumn())
                    );
}
```

3.5 删除数据：

```
HTable table = new HTable(hbaseConfig, "mytest");
Delete d = new Delete("row1".getBytes());
table.delete(d)
```

3.6 切分表：hbase.hregion.max.filesize指示在当前ReigonServer上单个Reigon的最大存储空间，单个Region超过该值时，这个 Region会被自动split成更小的region。HBaseAdmin提供split方法来将table 进行手工split：public void split(final String tableNameOrRegionName)。如果提供tableName，那么会将table所有region进行split;如果提供 region Name，那么只会split这个region.由于split是一个异步操作，并不能确切地控制region的个数。

4. Hive基本工作原理

使用Hadoop进行数据分析：MapReduce是一个底层的编程接口，对于数据分析人员来说，这个编程接口并不是十分友好，还需要进行大量的 编程以及调试工作

Hive可以被认为是一种数据仓库，包括数据的存储以及查询.Hive包括一个高层语言的执行引擎，类似于SQL的执行引擎.Hive建立在Hadoop的其它组成部分之上，包括Hive依赖于HDFS进行数据保存，依赖于MapReduce完成查询操作.最初的分析是通过手工的python脚本形式进行.

Hive的应用范围举例：日志分析，数据挖掘，文档索引，商业智能信息处理，即时查询以及数据验证

4.1 Hive的组成模块

HiveQL：这是Hive的数据查询语言，与SQL非常类似。Hive提供了这个数据查询语言与用户的接口，包括一个shell的接口， 可以进行用户的交互以及网络接口与JDBC接口。JDBC使得程序可以直接使用Hive功能而无需更改.

Driver: 执行驱动程序，用以将各个组成部分形成一个有机的执行系统，包括会话的处理，查询获取以及执行驱动

Compiler：编译器，将HiveQL语言编译成中间表示，包括对于HiveQL语言的分析，执行计划的生成以及优化 等工作

Execution Engine：执行引擎，在Driver的驱动下，具体完成执行操作，包括MapReduce执行，或者HDFS操作，或者元数据操作

Metastore：用以存储元数据：存储操作的数据对象的格式信息，在HDFS中的存储位置的信息以及其他的用于数据转换 的信息SerDe等

4.2 Hive的系统结构:见Figure 1.

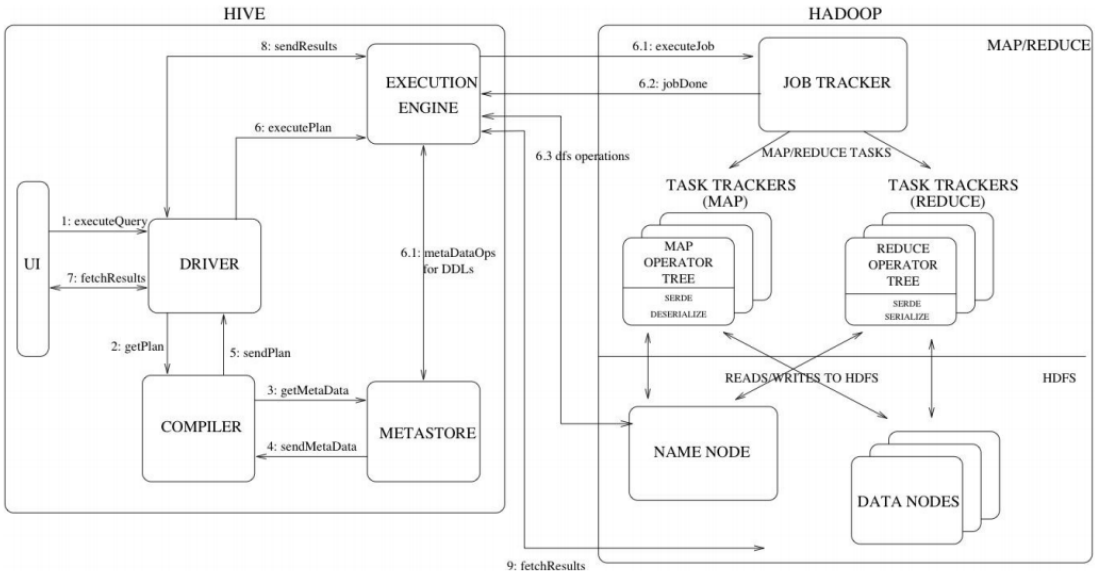

Figure 1: Hive的系统结构

4.3 Hive的数据模型

Tables：Hive的数据模型由数据表组成. 数据表中的列是有类型的（int, float, string, data, boolean） 也可以是复合的类型，如list: map （类似于JSON形式的数据）

Partitions：数据表可以按照一定的规则进行划分Partition. 例如：日期

Buckets：数据存储的桶。在一定范围内的数据按照Hash的方式进行划分（这对于数据的抽样以及对于join的优化很有意义）

4.4 元数据存储：Metastore

在Hive中由一系列的数据表格组成一个命名空间，这个命名空间的描述信息会保存在Metastore的空间中.

元数据使用SQL的形式存储在传统的关系数据库中，因此可以使用任意一种关系数据库，例如Derby(apache的关系数据 库实现)，MySQL以及其他的多种关系数据库存储方法.

在数据库中，保存最重要的信息是有关数据库中的数据表格描述，包括每一个表的格式定义，列的类型，物理的分布情况，数据划分情况等

4.5 数据的物理分布情况

Hive在HDFS中有固定的位置，通常被放置在HDFS的如下目录中：/home/hive/warehouse

每个数据表被存放在warehouse的子目录中. 数据划分Partition、数据桶Buckets形成了数据表的子目录.数据可能以任意一种形式存储，例如：使用分隔符的文本文件，或者是SequenceFile. 使用用户自定义的SerDe，则可以定义任意格式的文件.

4.6 Hive系统的配置

首先，需要创建Hive的配置文件hive-site.xml.Hive安装包中包含一个配置文件模板－hive-default.xml.template.

fs.default.name用以告知Hive对应的HDFS文件系统的名字节点在什么位置

mapred.job.tracker用以告知Hive对应的MapReduce处理系统的任务管理器在什么位置

通过上面两点的配置，就可以让Hive系统与对应的HDFS以及 MapReduce进行交互，完成数据的存取以及查询的操作

5. Hive基本操作实例

5.1 启动Hive的命令行界面shell:

进入hive安装目录执行 ./hive

5.2 创建数据表的命令

显示所有的数据表：show tables;

创建一个表，这个表包括两列，分别是整数类型以及字符串类型，使用文本文件表达，数据域之间的分隔符为tab：create table shakespeare (freq int, word string) row format delimited fields terminated by '\t' stored as textfile;

显示所创建的数据表的描述，即创建时候对于数据表的定义：describe shakespeare;

5.3 装入数据

load data inpath "shakespeare_freq" into table shakespeare;

5.4 SELECTS and FILTERS

select * from shakespeare limit 10;

select * from shakespeare where freq > 100 sort by freq asc limit 10;

SELECT a.foo FROM invites a WHERE a.ds='2008-08-15';

INSERT OVERWRITE DIRECTORY '/tmp/hdfs_out' SELECT a.* FROM invites a WHERE a.ds='2008-08-15';

INSERT OVERWRITE LOCAL DIRECTORY '/tmp/local_out' SELECT a.* FROM pokes a;

5.5 Group By

INSERT OVERWRITE TABLE events SELECT a.bar, count(*) FROM invites a WHERE a.foo > 0 GROUP BY a.bar;

5.6 Join

SELECT t1.bar, t1.foo, t2.foo FROM pokes t1 JOIN invites t2 ON t1.bar = t2.bar;

七、高级MapReduce编程技术计

1. 复合键值对的使用

   1.1 用复合键让系统完成排序

   带频率的倒排索引：为了能利用系统自动对docid进行排序，解决方法是：代之以生成（term，$< docid, tf >$)键值对，map时将term和docid组合起来形成复合键$< term, docid >$。需要实现一个新的Partitioner：从$< term, docid >$中取出term，以term作为key进行partition。

   1.2 把小的键值对合并成大的键值对

   单词同现矩阵算法：一个Map可能会产生单词a与其它单词间的多个键值对，这些键值对可以在Map过程中合并成一个大的键值对a → { b: 1, c: 2, d: 5, e: 3, f: 2 },然后，在Reduce阶段，把每个单词a的键值对(条)进行累加.

2. 用户自定义数据类型

   2.1 Hadoop内置的数据类型: 这些数据类型都实现了WritableComparable接口，以便进行网络传输和文件存储，以及进行大小比较。BooleanWritable, ByteWritable, DoubleWritable, FloatWritable, IntWritable, LongWritable, Text, NullWritable.

   2.2 用户自定义数据类型

   需要实现Writable接口，作为key或者需要比较大小时则需要implements实现WritableComparable接口

```java
public class Edge implements WritableComparable<Edge>{
        private String departureNode;
        private String arrivalNode;
        public String getDepartureNode() {
                return departureNode;
        }
        @Override
        public void readFields(DataInput in) throws
            IOException
        {
                departureNode = in.readUTF();
                arrivalNode = in.readUTF();
        }
        @Override
        public void write(DataOutput out) throws IOException
        {
                out.writeUTF(departureNode);
                out.writeUTF(arrivalNode);
        }
        @Override
        public int compareTo(Edge o)
        {
                return (departureNode.compareTo(o.
                    departureNode)!=0)
                ?departureNode.compareTo(o.departureNode):
                    arrivalNode.compareTo(o.arrivalNode);
        }
}
```

3. 用户自定义输入输出格式

3.1 Hadoop内置的文件输入格式:AutoInputFormat, CombineFileInutFormat, CompositeInputFormat, DBInputFormat, FileInputFormat, **KeyValueTextInputFormat**, LineDocInputFormat, MultiFileInputFormat, NLineInputFormat, SequenceFileAsBinaryInputFormat, SequenceFileAsTextInputFormat, SequenceFileInputFilter, **SequenceFileInputFormat**, StreamInputFormat, **TextInputFormat**.

3.2 Hadoop内置的RecordReader: CombineFileRecordReader, DBInputFormat.DBRecordReader, InnerJoinRecordReader, JoinRecordReader, **KeyValueLineRecordReader**, LineDocRecordReader, MultiFilterRecordReader, OuterJoinRecordReader, OverrideRecordReader,**LineRecordReader**, SequenceFileAsBinaryInputFormat.SequenceFileAsBinaryRecordReade SequenceFileAsTextRecordReader, **SequenceFileRecordReader**, StreamBaseRecordReader, StreamXmlRecordReader, WrappedRecordReader

3.3 自定义InputFormat和RecordReader:简单的文档倒排索引

```
1  public class FileNameOffsetInputFormat extends
       FileInputFormat<Text, Text>
2          {
3          @Override
4          public RecordReader<Text, Text> createRecordReader(
               InputSplit split, TaskAttemptContext context)
5          {
6                  FileNameOffsetRecordReader fnrr = new
                       FileNameOffsetRecordReader();
7          try
8          {
9                  fnrr.initialize(split, context); }
10                 catch (IOException e) { e.printStackTrace();
                       }
11                 catch (InterruptedException e) { e.
                       printStackTrace(); }
12                 return fnrr;
13         }
14 }
15 public class FileNameOffsetRecordReader extends RecordReader<
       Text, Text>
16 {
17         String fileName;
18         LineRecordReader lrr = new LineRecordReader();
19         ...
20         @Override
21         public Text getCurrentKey() throws IOException,
               InterruptedException
22         {
23                 return new Text("(" + fileName + "#" + lrr.
                       getCurrentKey() + ")");
24         }
25         @Override
26         public Text getCurrentValue() throws IOException,
               InterruptedException
27         { return lrr.getCurrentValue(); }
28         @Override
29         public void initialize(InputSplit arg0,
               TaskAttemptContext arg1)
30         throws IOException, InterruptedException
31         {
32                 lrr.initialize(arg0, arg1);
```

```
33                      fileName = ((FileSplit)arg0).getPath().
                            getName();
34              }
35      }
```

3.4 Hadoop内置的OutputFormat

TextOutputFormat, **SequenceFileOutputFormat**, **NullOutputFormat**, DBOutputFormat, FileOutputFormat, FilterOutputFormat, IndexUpdateOutputFormat, LazyOutputFormat, MapFileOutputFormat, MultipleOutputFormat, MultipleSequenceFileOutputFormat, MultipleTextOutputFormat, NullOutputFormat, SequenceFileAsBinaryOutputFormat, SequenceFileOutputFormat, TextOutputFormat

3.5 Hadoop内置的RecordWriter

LineRecordWriter, DBOutputFormat.DBRecordWriter, FilterOutputFormat.FilterRecordWriter, TextOutputFormat.LineRecordWriter

4. 用户自定义Partitioner和Combiner

4.1 Partitioner

```
1   Class NewPartitioner extends HashPartitioner<K,V>
2   {
3           // override the method
4           getPartition(K key, V value, int numReduceTasks)
5           {
6                   term = key.toString().split(",")[0]; //¡term,
                        docid¿=¿term
7                   super.getPartition(term, value,
                        numReduceTasks);
8           }
9   }
10  //在Job中设置新的Partitioner:
11  Job.setPartitionerClass(NewPartitioner)
```

4.2 Combiner 申请美国专利的国家数统计：Reduce中对每一个($< year, country >$, [1, 1,1,...])输入，忽略后部的出现次数，仅考虑key部分：$< year, country >$

```
1   public static class NewCombiner extends Reducer< Text,
        IntWritable, Text, IntWritable >
2   {
3           public void reduce(Text key, Iterable<IntWritable>
                values, Context context) throws IOException,
                InterruptedException
4           { // 忽略($< year, country >$, [1, 1,1,...])后部很长的数据串，
5             // 归并为$< year, country >$的1次出现
6                   context.write(key, new IntWritable(1));
7           } // 输出key: $< year, country >$; value: 1
8   }
```

5. 迭代MapReduce计算 页面排序算法PageRank(第8章介绍)

6. 组合式MapReduce程序设计 专利文献引用直方图统计，需要先进行被引次数统计，然后在被引次数上再进行被引直方图统计

链式MapReduce中的每个子任务需要提供独立的jobconf，并按照前后子任务间的输入输出关系设置输入输出路径,而任务完成后所有中间过程的输出结果路径都可以删除掉。

具有数据依赖关系的MapReduce子任务的执行:Hadoop通过Job和JobControl类提供了一种管理这种具有数据依赖关系的子任务的MapReduce作业的执行。Job除了维护Conf信息外，还能维护job间的依赖关系:

obx = new Job(jobxconf, “Jobx”);

……

joby = new Job(jobyconf, “Joby”);

……

jobz = new Job(jobzconf, “Jobz”);

jobz.addDependingJob(jobx); // jobz将等待jobx执行完毕

jobz.addDependingJob(joby); // jobz将等待joby执行完毕

JobControl JC = new JobControl（“XYZJob”）；

JC.addJob(jobx);

JC.addJob(joby);

JC.addJob(jobz);

JC.run();

MapReduce前处理和后处理步骤的链式执行：多个独立的MapReduce作业的执行开销较大，且增加很多I/O操作，因而效率不高。一个办法是在核心的Map和Reduce过程之外，把这些前后处理步骤实现为一些辅助的Map和Reduce过程，将这些辅助Map和Reduce过程与核心Map和Reduce过程合并为一个过程链，从而完成整个作业的执行。Hadoop提供了链式Mapper(ChainMapper)和链式Reducer (ChainReducer)来完成这种处理。ChainMapper和ChainReducer别提供了addMapper方法加入一系列Mapper：ChainMapper.addMapper（……）;ChainReducer.addMapper

public static void addMapper

(Job job, // 主作业

Class<?*extendsMapper* > mclass, // 待加入的map class

Class<? > inputKeyClass, //待加入的map输入键class

Class<? > inputValueClass, //待加入的map输入键值class

Class<? > outputKeyClass, //待加入的map输出键class

Class<? > outputValueClass, //待加入的map输出键值class

org.apache.hadoop.conf.Configuration mapperConf // 待加入的map的conf

) throws IOException

示例：JobConf map1Conf = new JobConf(false);
ChainMapper.addMapper(job, Map1.class, LongWritable.class, Text.class, Text.class, Text.class, true, map1Conf);

示例：JobConf map3Conf = new JobConf(false);
ChainReducer.addMapper(job, Map3.class, Text.class, Text.class, LongWritable.class, Text.class, true, map3Conf);

7. 多数据源的连接

Hadoop系统没有关系数据库中那样强大的join处理功能，因此多数据源的连接处理比关系数据库中要复杂一些。根据不同的需要和权衡，可以有几种不同的连接方法。

7.1 用DataJoin类实现Reduce端Join

Hadoop提供了一个实现多数据源连接的基本框架DataJoin类，帮助程序员完成一些必要的连接处理；DataJoin框架包含以下三个抽象类，使用DataJoin框架后，程序员仅需要实现几个抽象方法：

DataJoinMapperBase: 由程序员实现的Mapper类继承,该基类已实现了map()方法用以完成标签化数据记录的生成和输出,因此程序员仅需实现三个产生数据源标签、GroupKey(key)和标签化记录(value)所需要的抽象方法

DataJoinReducerBase: 由程序员实现的Reducer类继承,该基类已实现了reduce()方法用以完成多数据源记录的叉积组合生成.

TaggedMapOutput: 描述一个标签化数据记录，实现了getTag(), setTag()方法；作为Mapper的key-value输出中的value的数据类型，由于需要进行I/O，程序员需要继承并实现Writable接口,并实现抽象的getData()方法用以读取记录数据。

Mapper需要实现的抽象方法:

```
protected Text generateInputTag(String inputFile)
{
        return new Text(inputFile);
}
protected TaggedMapOutput generateTaggedMapOutput(Object
    value)
{
        // 设程序员继承实现的TaggedMapOutput子类为TaggedWritable
        TaggedWritable retv = new TaggedWritable((Text) value
            );
        // 将generateInputTag()方法计算出来存储在Mapper.inputTag中的标签
            设为数据源标签
        retv.setTag(this.inputTag);
        return retv;
}
```

Reducer需要实现的抽象方法：abstract TaggedMapOutput combine(Object[] tags, Object[] values) 实现笛卡尔积操作。

实现Customers和Orders连接处理的程序实现:

```
public static class MapClass extends DataJoinMapperBase{
        protected Text generateInputTag(String inputFile) {
                String datasource = inputFile.split("-")[0];
                    // 使用输入文件名作为标
                    签
                return new Text(datasource); // 该数据源标签将
                    被map()保存在inputTag中
        }
        protected Text generateGroupKey(TaggedMapOutput
            aRecord) {
                String line = ((Text) aRecord.getData()).
                    toString();
                String[] tokens = line.split(",");
                String groupKey = tokens[0];
                return new Text(groupKey); // 取CustomerID作
                    为GroupKey（key）
        }
        protected TaggedMapOutput generateTaggedMapOutput(
            Object value) {
                TaggedWritable retv = new TaggedWritable((
                    Text) value);
                retv.setTag(this.inputTag); // 给一个原始数据记录
                    贴上指定的标签
                return retv;
        }
}
public static class ReduceClass extends DataJoinReducerBase
{
        protected TaggedMapOutput combine(Object[] tags,
            Object[] values)
```

```
21          {
22                  if (tags.length < 2) return null; // 一个以下数
                        据源，没有需连接的数据记录
23                  String joinedStr = "";
24                  for (int i=0; i<values.length; i++)
25          {
26                  if (i > 0) joinedStr += ",";
27                  TaggedWritable tw = (TaggedWritable) values[i
                        ];
28                  String line = ((Text) tw.getData()).toString
                        ();
29                  String[] tokens = line.split(",", 2); //
                        把CustomerID与后部的字段分为两段
30                  if(i==0) joinedStr += tokens[0]; // 拼接一
                        次CustomerID
31                  joinedStr += tokens[1]; // 拼接每个数据源记录后部
                        的字段
32          }
33                  TaggedWritable retv = new TaggedWritable(new
                        Text(joinedStr));
34                  retv.setTag((Text) tags[0]); // 把第一个数据源标
                        签设为join后记录的标签
35                  return retv; // join后的该数据记录将在reduce()中
                        与GroupKey一起输出
36          }
37  }
38  public static class TaggedWritable extends TaggedMapOutput
39  {
40          private Writable data;
41          public TaggedWritable(Writable data)
42          {
43                  this.tag = new Text("");
44                  this.data = data;
45          }
46          @Override
47          public Writable getData()
48          {
49                  return data;
50          }
51          public void write(DataOutput out) throws IOException
52          {
53                  this.tag.write(out);
54                  this.data.write(out);
55          }
56          public void readFields(DataInput in) throws
                IOException
57          {
58                  this.tag.readFields(in);
59                  this.data.readFields(in);
60          }
61  }
62
63  public Class DataJoin
64  {
65          public static void main(String[] args) throws
                Exception
```

```
66          {
67                  Configuration conf = getConf();
68                  JobConf job = new JobConf(conf, DataJoin.
                        class);
69                  Path in = new Path(args[0]);
70                  Path out = new Path(args[1]);
71                  FileInputFormat.setInputPaths(job, in);
72                  FileOutputFormat.setOutputPath(job, out);
73                  job.setJobName("DataJoin");
74                  job.setMapperClass(MapClass.class);
75                  job.setReducerClass(Reduce.class);
76                  job.setInputFormat(TextInputFormat.class);
77                  job.setOutputFormat(TextOutputFormat.class);
78                  job.setOutputKeyClass(Text.class);
79                  job.setOutputValueClass(TaggedWritable.class)
                        ;
80                  job.set("mapred.textoutputformat.separator",
                        ",");
81                  Job.waitForCompletion(true);
82          }
83  }
```

Join操作直到Reduce阶段才能处理，很多无效的连接数据组合在Reduce阶段才能去除，而这时这些数据已经通过网络从Map阶段传送到了Reduce阶段，占据了很多的通信带宽。因此这个方法的效率不是很高。

7.2 用DataJoin类实现Reduce端Join

Hadoop提供了一个distributed cache机制用于将一个或多个文件复制到所有节点上。Job类中：public void void addCacheFile(URI uri); Mapper或Reducer的context类中：public Path[] getLocalCacheFiles():

```
1   Configuration conf = getConf();
2   Job job = new Job(conf, DataJoinDC.class);
3   // 将第一个数据源(假定是较小的那个)放置到distributed cache file中
4   Job.addCacheFile(new Path(args[0]).toUri());
5   Path in = new Path(args[1]);
6   Path out = new Path(args[2]);
7   FileInputFormat.setInputPaths(job, in);
8   FileOutputFormat.setOutputPath(job, out);
9   job.setJobName("DataJoin with DistributedCache");
10  job.setMapperClass(MapClass.class);
11  job.setNumReduceTasks(0);
12  job.setInputFormat(KeyValueTextInputFormat.class);
13  job.setOutputFormat(TextOutputFormat.class);
14  job.set("key.value.separator.in.input.line", ",");
15  Job.waitForCompletion(true);
16
17  public static class MapClass extends Mapper<Text, Text, Text,
        Text>
18  {
19    private Hashtable<String, String> joinData = new Hashtable<
          String, String>();
20    public void setup(Mapper.Context context) // override setup()
21    {
22      try // 将distributed cache file装入各个Map节点本地的内存数据joinData中
23      {
```

```
24          Path [] cacheFiles = context.getLocalCacheFiles();
25          if (cacheFiles != null && cacheFiles.length > 0)
26          {
27            String line;
28            String[] tokens;
29            BufferedReader joinReader = new BufferedReader(
30            new FileReader(cacheFiles[0].toString()));
31            try // 以CustomerID作为key，将后部的字段数据存入一个
32            // Hashtable，以便后面使用
33            {
34              while ((line = joinReader.readLine()) != null)
35              {
36                tokens = line.split(",", 2);
37                joinData.put(tokens[0], tokens[1]);
38              }
39            }
40            finally
41            {
42              joinReader.close();
43            }
44          }
45        }
46      catch (IOException e)
47      {
48        System.err.println("Exception reading DistributedCache: "
              + e);
49      }
50      public void map(Text key, Text value, Context context)
          throws IOException, InterruptedException
51      { // 第二个数据源的数据记录将作为Map方法的输入键值对
52        // 将value与joinData中的相应记录进行join
53        String joinValue = joinData.get(key);
54        if (joinValue != null)
55        {
56          output.collect(key,
57          new Text(value.toString() + "," + joinValue));
58        }
59      }
60  }
61  //由于在Map端即实现了join，因而不再需要实现Reducer。
```

带Map端过滤的Reduce端Join：Mapper端同上。额外实现一个方法，根据一定的条件过滤Customers数据记录、并保存为一个临时文件。

7.3 多数据源连接

以上的多数据源Join只能是具有相同主键/外键的数据源间的连接，如果数据源两两之间具有多个不同的主键/外键的连接，则需要使用多次MapReduce过程完成不同主/外键间的连接。例如，三个表的连接需要两次MapReduce作业完成。

8. **全局参数/数据文件的传递** Configuartion类专门提供以下用于保存和获取属性的方法 mapper/reducer类初始化方法setup()从configuration对象中读出属性

8.1 全局作业参数的传递

Configuartion类专门提供以下用于保存和获取属性的方法：

public void set(String name, String value) //设置字符串属性

public String get(String name) // 读取字符串属性

public String get(String name, String defaultValue) // 读取字符串属性
public void setBoolean(String name, boolean value) //设置布尔属性
public boolean getBoolean(String name, boolean defaultValue) //读取布尔属性
public void setInt(String name, int value) //设置整数属性
public int getInt(String name, int defaultValue) // 读取整数属性
public void setLong(String name, long value) //设置长整数属性
public long getLong(String name, long defaultValue) // 读取长整数属性
public void setFloat(String name, float value) //设置浮点数属性
public float getFloat(String name, float defaultValue) //读取浮点数属性
public void setStrings(String name, String... values) //设置一组字符串属性
public String[] getStrings(String name, String... defaultValue) //读取一组字符串属性
setStrings方法将把一组字符串转换为用“，”隔开的一个长字符串，然后getStrings时自动再根据“,”split成一组字符串，因此，在该组中的每个字符串都不能包含“，”，否则会出错。
在mapper/reducer类的初始化方法setup()中从configuration对象中读出属性. eg:obconf.tInt(“JoinKeyColI
-1);

8.2 全局数据文件的传递

传递一些较小的并且需要复制到各个节点的数据文件：DistributedCache
Job类中：public void addCacheFile(URI uri)：将一个文件件放到distributed cache file中
Mapper或Reducer的context类中：public Path[] getLocalCacheFiles(): 获取设置在distributed cache files中的文件路径，以便能将这些文件读入到每个节点内存中

9. 其他处理技术

9.1 查询任务相关信息

可以通过Configuration对象，使用预定义的属性名称查询计算作业相关的信息。
mapred.job.id
mapred.jar
job.local.jar
mapred.tip.id
mapred.task.id
mapred.task.is.map (flag that whether this is a map task)
mapred.task.partition (id of task within this job)
map.input.file
map.input.start
map.input.length
map.work.output.dir

9.2 划分多个输出文件集合

Hadoop提供了MultipleOutputFormat类.例如：将专利描述文件数据集按照国家进行多文件集合输出

```
1 public static class MapClass extends Mapper<LongWritable,
      Text, NullWritable, Text>
2 {
3         public void map(LongWritable key, Text value, Context
              context)
4         throws IOException, InterruptedException
5         {
6                 context.write (NullWritable.get(), value);
```

```
7           } // NullWritable.get() 返回singleton单一实例
8   }
9   public static class SaveByCountryOutputFormat extends
        MultipleTextOutputFormat<NullWritable,Text> {
10          protected String generateFileNameForKeyValue(
                NullWritable key, Text value, String filename)
11          {
12                  String[] arr = value.toString().split(",",
                        -1);
13                  String country = arr[4].substring(1,3);
14                  return country + "/" + filename;
15          }
16  }
17
18  public class MultiFileDemo
19  {
20          public static void main(String[] args) throws
                Exception
21          {
22                  Configuration conf = new Configuration();
23                  Job job = new Job(conf, MultiFileDemo.class);
24                  Path in = new Path(args[0]);
25                  Path out = new Path(args[1]);
26                  FileInputFormat.setInputPaths(job, in);
27                  FileOutputFormat.setOutputPath(job, out);
28                  job.setJobName("MultiFileDemo");
29                  job.setMapperClass(MapClass.class);
30                  job.setInputFormat(TextInputFormat.class);
31                  job.setOutputFormat(SaveByCountryOutputFormat
                        .class);
32                  job.setOutputKeyClass(NullWritable.class);
33                  job.setOutputValueClass(Text.class);
34                  job.setNumReduceTasks(0);
35                  Job.waitForCompletion(true);
36          }
37  }
```

9.3 输入输出到关系数据库

OLTP (online transaction processing)联机事务处理：主要是关系数据库应用系统中前台常规的各 种事务处理

OLAP (online analytical processing)联机分析处理：主要是进行基于数据仓库的后台数据分析和 挖掘，提供优化的客户服务和运营决策支持

OLTP与OLAP一般采用分离的数据库,前者数据库负责大量的常规的事务处理,后者用数据仓库应对大量的数据分析处理负载。

问题：OLAP端基于关系数据库的数据仓库解决方案，在数据量巨大的情况下，复杂数据分析和挖掘处理的负载很大，速度性能跟不上

解决方案：提供基于MapReduce大规模数据并行处理的OLAP！

问题：如何从MapReduce访问关系数据库?

从数据库中输入数据：DBInputFormat：提供从数据库读取数据的格式；DBRecordReader：提供读取数据记录的接口。处理效率不理想，因此，仅适合读取小量数据记录的计算和应用，不适合OLAP数据仓库大量数据 的读取处理。读取大量数据记录一个更好的解决办法是，用数据库中的Dump工具将大量待分析数据输出为文本数据文件，

并上载到HDFS中进行处理。

向数据库中输出计算结果：DBOutputFormat：提供向数据库输出数据的格式；DBRecordWriter：提供向数据库写入数据记录的接口；DBConfiguration提供数据库配置和创建连接的接口。

创建数据库连接：public static void configureDB(Job job, String driverClass, String dbUrl, String userName, String passwd)

指定写入的数据表和字段：public static void setOutput(Job job, String tableName, String... fieldNames)

示例：

Configuration conf = new Configuration();

Job job = new Job(conf, JobClass.class);

job.setOutputFormat(DBOutputFormat.class);

DBConfiguration.configureDB(job, “com.mysql.jdbc.Driver”,

“jdbc:mysql://db.host.com/mydb”, “username”, “password”)

DBOutputFormat.setOutput(job, “Events”, “event_id”, “time”);

// 向Events表输出event_id和time字段

向数据库中输出计算结果：为了实际完成向数据库中数据写入，程序员要实现DBWritable：

```
1  public class EventsDBWritable implements Writable, DBWritable
2  {
3    private int id;
4    private long timestamp;
5    public void write(DataOutput out) throws IOException
6    {
7      out.writeInt(id); out.writeLong(timestamp);
8    }
9    public void readFields(DataInput in) throws IOException
10   {
11     id = in.readInt(); timestamp = in.readLong();
12   }
13   public void write(PreparedStatement statement) throws
       SQLException
14   {
15     statement.setInt(1, id); statement.setLong(2, timestamp);
           }
16     public void readFields(ResultSet resultSet) throws
         SQLException
17   {
18     id = resultSet.getInt(1);
19     timestamp = resultSet.getLong(2);
20   }
21 // 除非使用DBInputFormat直接从数据库输入数据,否则readFields方法不会被调用
22 }
```

八、基于MapReduce的搜索引擎算法

1. 网页排名图算法PageRank

1.1 PageRank的基本设计思想和设计原则

PageRank是一种在搜索引擎中根据网页之间相互的链接关系计算网页排名的技术。PR值越高说明该网页越受欢迎（越重要）。

被许多优质网页所链接的网页，多半也是优质网页.一个网页要想拥有较高的PR值的条件：有很多网页链接到它；有高质量的网页链接到它

1.2 PageRank的简化模型及其缺陷

可以把互联网上的各个网页之间的链接关系看成一个有向图。

对于任意网页Pi，它的PageRank值可表示为：$R(P_i) = \sum_{P_j \in B_i} \frac{R(P_j)}{L_j}$.其中Bi为所有链接到网页i的网页集合,Lj为网页j的对外链接数

缺陷：Rank leak：一个独立的网页如果没有外出的链接就产生排名泄漏。多次迭代后所有网页的Rank变为。解决办法：将无出度的节点递归地从图中去掉，待其他节点计算完毕后再加上；对无出度的节点添加一条边，指向那些指向它的顶点。

缺陷：Rank Sink：整个网页图中若有网页没有入度链接，其所产生的贡献会被与之相邻的强连通分量吞噬，没有入度结点的PR值在多次迭代后变为0.

1.3 PageRank的随机浏览模型

设定任意两个网页之间都有直接链接，在每个网页以概率d访问自己，以概率1- d访问其他网页。

$R = H^{'} * R$ 满足马尔可夫链的性质，如果马尔可夫链收敛，则R存在唯一解。

$H^{'} = d * H + (1-d) * [1/N]_{N*N}$

H为转移矩阵

随机浏览模型的PageRank公式：

$PR(p_i) = \frac{1-d}{N} + d\sum_{p_j \in M(p_i)} \frac{PR(p_j)}{L(p_j)}$

优点：更加符合用户的行为；一定程度上解决了rank leak和rank sink问题；保证PageRank存在唯一值。

1.4 用MapReduce实现PageRank

Phase1：GraphBuilder: 建立网页之间的超链接图.逐行分析原始数据, Map输出<URL ,(PR_init, link_list)>, Reduce不做处理

Phase2: PageRankIter: 迭代计算各个网页的PageRank值. Map对上阶段的 <URL, (cur_rank, link_list)>产生两种<key, value> 对：

1. For each u in link_list, 输出 <u, cur_rank/—link_list—>
2. 同时为了完成迭代过程，需要传递每个网页的链接信息 <URL, link_list>

Reducer对 Map输出的<URL, url_list> 和多个 <URL, val>做如下处理:

<URL, val>为当前URL的链入网页对其贡献的PageRank值 计算所有val的和，并乘上d，在加上常数$(1-d)/N$得到 new_rank。 输出 (URL, (new_rank, url_list))。

Phase3: RankViewer: 按PageRank值从大到小输出. PageRankViewer从最后一次迭代的结果读出文件，并将文件名和其PR值读出，并以PR值为key网页名为value，并且以PR值从大到小的顺序输出。排序过程中可以采用框架自身的排序处理，重载key的比较函数，使其经过shuffle和sort后反序（从大到小）输出.

```
public static class DecFloatWritable extends FloatWritable {
 ...
 @Override
 public int compareTo(Object o) {
   return -super.compareTo(o);
 }
```

}
PageRank迭代终止条件：各网页的PageRank值不再改变；各网页的PageRank值排序不再变化；迭代至固定次数；

九、基于MapReduce的数据挖掘算法

1. 基于MapReduce的K-Means并行化算法

1.1 数据挖掘并行算法研究的重要性

数据挖掘是通过对大规模观测数据集的分析，寻找这些数据集中有用信息和事实的过程。

数据挖掘的特征之一：海量数据

研究发现大数据隐含着更为准确的事实:大数据集上的简单算法能比小数据集上复杂算法产生更好的结果。

海量数据挖掘是并行计算中值得研究的一个领域

1.2 K-Means聚类算法

定义：将给定的多个对象分K干组，组内的各个对象是相似的，组间的对象时不相似的。进行划分的过程就是聚类过程，划分后的组称为簇(cluster).

聚类方法：基于划分、层次、密度...

数据点的类型可分为：欧式、非欧式空间

Cluster的表示：欧式空间：取数据点平均值(centroid)；非欧式空间：取某些最靠近中间的点、取若干最具代表性的点(clustroid)

相似度(距离)的计算：欧式空间：坐标分量差的平方和开根；非欧式空间：通常不能直接简单计算，Jaccard距离($1-|S\cap T|/|S\cup T|$)、Cosine距离(向量夹角大小)、Edit距离(String 数据间距离).

K-Means算法：初始选取K个点作为聚类中心，遍历输入的每个点p, 选取最近的聚类中心进行分类。重新计算聚类中心。如果聚类中心不再变化，结束。

局限性：①初始聚类中心的选取会影响最终聚类结果；②能够得到局部最优解，不保证得到全局最优解；③相似度计算和比较时的计算量较大。

复杂度$O(nkt)$, n为样本规模，k为聚类数，t为迭代次数。将相似度计算分摊到不同机器上并行运算能够提高计算速度。

1.3 基于MapReduce的K-Means并行算法设计

Map结点读取聚类中心(setup 中完成)，对要计算的数据点分类得到<ClusterID,(p,1)>

Reducer结点计算当前聚类的新的中心. 用Combiner优化，合并相同ClusterID下所有数据点并取这些点的均值以及数据点个数n，输出<ClusterID, pm>.

需要全局共享的数据：当前迭代次数；K个聚类数据：聚类id, 聚类中心, 该聚类中心数据点的个数

```
class Mapper
 setup(...)
 {
  读出全局的聚类中心数据 →Centers
 }
 map(key, p) // p为一个数据点
 {
  minDis = Double.MAX VALUE;
  index = -1;
  for i=0 to Centers.length
  {
   dis= ComputeDist(p, Centers[i]);
   if dis < minDis
    { minDis = dis;
    index = i;
```

```
    }
   }
  emit(Centers[i].ClusterID, (p,1));
}
class Combiner
  reduce(ClusterID, [(p1,1), (p2,1), ...]) {
    pm = 0.0;
    n = 数据点列表[(p1,1), (p2,1), ...]中数据点的总个数;
    for i=0 to n
    pm += p[i];
    pm = pm / n; // 求得这些数据点的平均值
    emit(ClusterID, (pm, n));
  }
class Reducer
  reduce(ClusterID, value = [(pm1,n1),(pm2,n2) ...]) {
    pm = 0.0;  n=0;
    k = 数据点列表中数据项的总个数;
    for i=0 to k
    { pm += pm[i]*n[i]; n+= n[i]; }
    pm = pm / n; // 求得所有属于ClusterID的数据点的均值
    emit(ClusterID, (pm,n)); // 输出新的聚类中心的数据值
  }
```

终止条件：固定迭代次数；均方差变化；每个点固定地属于某个聚类；...

数据间是无关的，但是数据和聚类中心相关的，因此需要全局文件，但不构成性能瓶颈。

1.4 聚类算法应用实例：

NetFlix百万美元大奖赛：影片推荐。

2. 基于MapReduce的分类并行化算法

2.1 分类问题的基本描述

基本作用：从一组已经带有分类标记的训练样本数据集预测一个测试样本的分类结果。

2.2 K-最近邻分类并行化算法

计算测试样本到各训练样本的距离，取其中距离最小的K个，并根据这K个训练样本的标记进行投票得到测试样本的标记。

加权K-近邻分类：在根据K个训练样本的标记进行投票时，根据测试样本与训练样本间的距离决定训练样本标记的作用大小，依次决定最终的测试样本的分类标记。$y' = \frac{\sum S_i * y_i}{\sum S_i}$, S为相似度，$y_i$为标记值

MapReduce设计思路：测试样本分块，训练样本在DistributedCache中作为全局数据。Map阶段计算测试数据与每个训练样本数据相似度，选出投票结果的标记，发射(testId, y)。Reduce阶段直接写出Map的结果。

2.3 朴素贝叶斯分类并行化算法

若朴素贝叶斯分类将未知样本X分配个Y_i, 有$P(Y_i|X) > P(Y_j|X), i \leq j \leq m, X = \{x_1, x_2, ..., x_n\}$， 样本共有m个类别。

由贝叶斯定理$P(Y_i|X) = P(X|Y_i) * P(Y_i)/P(X)$. 对于一个未知的样本X,可以取$P(X|Y_i)*P(Y_i)$的最大概率的样本标记$Y_i$作为分类结果。可由训练集计算$P(X_i|Y_i)$的概率。

MapReduce设计思路：扫描训练集，计算每个分类的$P(Y_i)$以及$P(X_j|Y_i)$.

```
class Mapper
  map(key, tr) // tr为一个训练样本
  {
  emit(y, 1)
  for i=0 to A.lenghth)
  {
    A[i] → 属性名xni和属性值xvi
    emit(<y, xni, xvi>, 1)
  }
  }
class Reducer
  reduce(key, value_list) // key 或为分类标记y，或为¡y, xni, xvi¿
  {
  sum = 0;
  while(value_list.hasNext())
  sum += value_list.next().get();
  emit(key, sum)
  }
```

测试样本分类预测Mapper伪代码

```
class Mapper
  setup()
  {
    读取从训练集得到的频度数据
    分类频度表FY = {(Yi, 每个Yi的频度FYi) }
    属性频度表 FxY = {(<Yi, xnj, xvj>, 出现频度为FxYij,即FxjYi/Fyi }
  }
  map(key, ts) // ts为一个测试样本
  {
    ts → tsid, A
    MaxF = MIN_VALUE; idx = -1;
    for (i=0 to FY.length)
    {
      FXYi = 1.0; Yi = FY[i].Yi; FYi = FY[i].FYi
      for (j = 0 to A.length)
        {
          xnj = A[j].xnj; xvj = A[j].xvj
          根据<Yi, xnj, xvj>扫描FxY表，取得FxYij
          FXYi = FXYi * FxYij;
        }
      if(FXYi* FYi > MaxF) { MaxF = FXYi*FYi; idx = i; }
      }
    emit(tsid, FY[idx].Yi)
  }
```

2.4 SVM短文本多分类并行化算法

思路：高维稀疏空间文本分类问题适合使用SVM训练效果、速度更快，使用Linear SVM处理。训练阶段：对于480多类问题，对每一个类做一个2-Class分类器；预测阶段：用前面的二分类器打分，分类为"是"且最高分超过阈值则分为该类，否则为异类。

训练阶段：Map:每个训练样本Map输出(ClassID, <true/false, 特征向量>). Reduce: 具有相同的ClassID进入同一个Reduce结点，以此训练一个二分类器。

测试阶段：Map(每个Map结点是上一步的二分类器): 将每个测试样本，输出(SampleID, <LableID, 评分Score>). Reduce: 具有相同SampleID的键值进入同一个Reduce结点。根据评分和阈值判断分类结果。

3. 基于MapReduce的频繁项集挖掘算法

3.1 频繁项集挖掘基本问题

一个transaction由id和集合itemset构成. k-itemset：集合大小为k

记I为items的一个子集，N为transactions的数量, M为transaction中itemset包含I的项数量。

M/N为I在D中的权重。频繁项集为transaction中权重大于用户设置阈值的项. k频繁项集表示I的大小为k.

枚举计算的复杂度是$O(2^n Nt)$. n是Item的总数, N是transaction大小, t是每个transaction平均包含的Item数。

3.2 现有算法概述

Apriori算法：多趟处理。每次发现(k)频繁项集. 直到没有频繁项集被发现，结束。

SON算法：把整个数据库分为多个不重合的部分，每部分发现所有部分的频繁项集，合并后去除全局下的非频繁项集得到全局频繁项集。引理：不是局部频繁的一定不是全局频繁的；全局频繁的一定是局部频繁的。

3.3 PSON：基于MapReduce的并行化算法

使用MapReduce的动机：数据库的一个划分和其他划分是无关的；每个划分可以被同步处理；SON算法天然适合并行化。

Map任务：找出每个划分的局部频繁项集(Apriori算法)，按项集的大小划分发送给Reduce结点。

Reduce任务：每个Reduce结点收到的项集大小是一样的。在全局验证收到的项集。

运行两次Mapreduce任务，第一次生成所有的全局候选项集；第二次判断全局候选项集是否是全局频繁项集。

第一次MapReduce: Map阶段：Apriori algorithm生成局部频繁项集，给每个频繁项集F发送<F, 1>. 1表示F是局部的频繁项集。Shuffle and Sort Phase: 相同的局部频繁项集发送给同一个reduce结点。Reduce结点：每个reduce结点给每个F只发送一次<F, 1>. 最后合并所有的<F, 1>的对作为全局候选项集。

第二次MapReduce: Map阶段：输入是上一阶段的输出（全局数据）和该Map节点获取的数据库划分。统计项集在该Map拥有的划分出现的频率。发送<F, v>, F是候选项集，v是出现频率. Shuffle and Sort phase: 同一个F的结果到同一个reduce节点. Reduce: 对同一个F的v求和，判断其是否满足全局频繁项集条件。

3.4 并行化算法实验结果

当数据库的大小达到数百GB的阈值时，PSON可以在可接受的时间内完成运行，从而在扩展中获得良好的性能。PSON可以在加速方面取得良好的表现（线性）。

十、Spark系统及其编程技术

1. Spark系统简介

1.1 Scala语言

Scala语言(JVM上的语言), 面相对象+函数式编程。

val, var 表示变量是不变引用，还是可变引用。 eg. val c : Int = 1

class定义一个类，括号里是构造参数. eg. class A(x: String, y: String).

object对象，同名类的单例对象或者自定义的单实例

方法：由def定义，方法名、参数列表、返回类型、方法体. eg: def addOne(x: Int): Int = x + 1

函数：带有参数的表达式。 eg: val addOne = (x: Int) =¿ x + 1

List.map函数：对List中的每个元素应用一个函数，返回应用后的元素所组成的List

1.2 为什么会有Spark?

MapReduce计算模式的缺陷：最初设计用于高吞吐量批量数据处理，不适合低延迟；需要在作业之间将数据存储到HDFS中，在迭代计算中数据共享效率低下；不是为充分利用内存而设计的，很难实现高性能；MapReduce不适用于复杂的计算问题，如图计算、迭代计算；2阶段固定模式，磁盘计算大量I/O性能低下；

Spark以内存计算为核心、集诸多计算模式之大成. 在速度、易用性(Scala)、广泛新、多处运行上比Hadoop更好。很多复杂的作业和交互式分析的查询都需要MapReduce不具备的一个功能：高效的数据共享。

Spark实现了弹性分布式数据集(横跨集群所有节点进行并行计算的分区元素集合, Hadoop文件创建得来，或已有Scala集合转换得到). Spark使用RDD以及对应的Transform/Action等操作算作进行运算。

根据RDD依赖关系组成世系(lineage),重计算以及checkpoint机制保证容错；只读、可分区，这个数据集的全部或部分可以缓存在内存中，在多次计算间重用；弹性是指内存不够时可以与磁盘进行交换。

RDD形成可执行的有向无环图DAG，构成灵活的计算流图

1.3 Spark的基本构架和组件

主要体系结构和组件：见3.

Spark集群的基本结构：

Master node+Driver：是集群部署时的概念，是整个集群的控制器，负责整个集群的正常运行，管理 Worker node;

Worker node+Executors：是计算节点，接收主节点命令与进行状态汇报; Executors：每个Worker上有一个Executor，负责完成Task程序的执行;

Spark集群部署后，需要在主从节点启动Master进程和Worker进程，对整个集群进行控制

本地线程可以访问HDFS存储。

Spark系统的基本结构：Driver 是应用执行起点，负责作业调度；Worker管理计算节点及创建并行处理任务，Cache存储中间结果等，Input Data为输入数据。

Spark应用程序的基本结构：Application：基于Spark的用户程序，包含了一个Driver Program和多个executor （Worker中）； Job：包含多个Task的并行计算，由Spark action催生 ；Stage：Job拆分成多组Task，每组任务被称为Stage，也可称为TaskSet；Task：基本程序执行单元，在一个executor上执行. 见Figure2.

Spark Driver的组成 pptch10-pg33

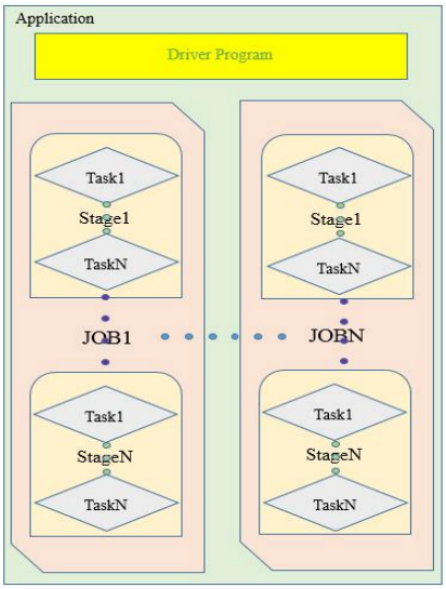

Figure 2: Spark应用程序的基本结构

Worker node的结构 pptch10-pg34
spark程序运行机制： 见Figure3

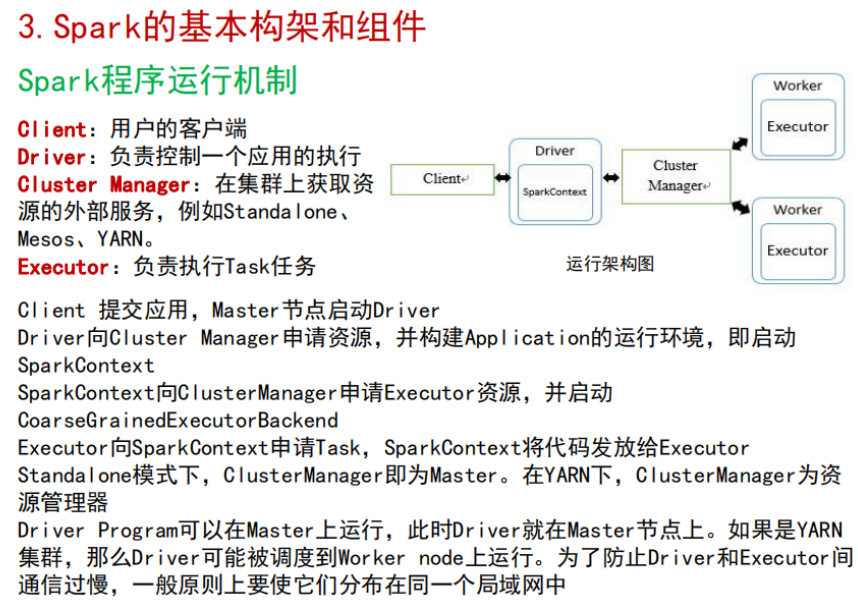

Figure 3: Spark程序运行机制

1.4 Spark程序执行过程

Spark运行框架主节点：
Application：由用户编写的Spark应用程序，其中包括driver program和executor
Driver Program：执行用户代码的main()函数，并创建 SparkContext
Cluster manager：集群当中的资源调度服务选取。 例：standalone manager, Mesos, YARN
Job：由某个RDD的Action算子生成或者提交的一个或者多 个一系列的调度阶段，称之为一个或者多个Job，类似于MapReduce中Job的概念

SparkContext：SparkContext由用户程序启动，是Spark运行的核心模块，它对一个Spark程序进行了必要的初始化过程，其中包括了：创建SparkConf类、SparkEnv类、调度类的实例

Executor：executor负责在子节点上执行Spark任务，每个application都有自身的Executor

Stage：每一个Job被分成一系列的任务的集合，这些集合被称之为Stage，用于Spark阶段的调度 例：在MapReduce作业中，Spark将划分为Map的Stage和Reduce的Stage进行调度

Task：被分发到一个Executor上的最小处理单元

Spark程序执行过程：

1. 用户编写的Spark程序提交到相应的Spark运行框架中

2. Spark创建SparkContext作为本次程序的运行环境

3. SparkContext连接相应的集群配置(Mesos/YARN),来确定程序的资源配置使用情况。

4. 连接集群资源成功后，Spark获取当前集群上存在Executor的 节点，即当前集群中Spark部署的子节点中处于活动并且可 用状态的节点(Spark准备运行你的程序并且确定数据存储)

5. Spark分发程序代码到各个节点。

6. 最终，SparkContext发送tasks到各个运行节点来执行。

一个作业就是一张RDD世系图. 按shuffle划分世系.

一个作业（Job）就是一组 Transformation操作和一个action操作的集合。每执行一次action操作，那么就会提交一个Job。Stage分为两种，Shuffle Stage和final Stage，每个作业必然只有一个final Stage，即每个action操作会生 成一个final stage，如果一个作业中还包含Shuffle操作，那么每进行一次shuffle操作，便会生成一个 Shuffle Stage。huffle操作只有在宽依赖的时候才会触发

任务（Task）作用的单位是Partition，针对同一个 Stage，分发到不同的Partition上进行执行。

构建RDD世系关系的优势：基于RDD世系的并行执行优化；更快速，更细粒度的容错。

1.5 Spark的技术特点：

RDD：Spark提出的弹性分布式数据集，是Spark最核心的分布式数据抽象，Spark的很多特性都和RDD密不可分。

Transformation&Action：Spark通过RDD的两种不同类型的运算实现了惰性计算，即在RDD的Transformation运算时， Spark并没有进行作业的提交；而在RDD的Action操作时才会触发SparkContext提交作业。

Lineage：为了保证RDD中数据的鲁棒性，Spark系统通过世系 关系(lineage)来记录一个RDD是如何通过其他一个或者多个父类RDD转变过来的，当这个RDD的数据丢失时，Spark可以通过它父类的RDD重新计算

Spark调度：Spark采用了事件驱动的Scala库类Akka来完成任务的启动，通过复用线程池的方式来取代MapReduce进程或者线程启动和切换的开销。

API：Spark使用scala语言进行开发，并且默认Scala作为其 编程语言。因此，编写Spark程序比MapReduce程序要简洁得多。同时，Spark系统也支持Java、Python语言进行开发

Spark生态：Spark SQL、Spark Streaming、GraphX等等为 Spark的应用提供了丰富的场景和模型，适合应用于不同的计 算模式和计算任务

Spark部署：Spark拥有Standalone、Mesos、YARN等多种部署方式，可以部署在多种底层平台上

综上所述，Spark是一种基于内存的迭代式分布式计算框架，适合于完成多种计算模式的大数据处理任务

1.6 Spark编程模型与编程接口

RDD:是一种分布式的内存抽象，允许在大型集群上执行基于内存的计算（In-Memory Computing），同时还保持了MapReduce等数据流模型的容错特性。RDD只读、可分区，这个数据集的全部或部分可以缓存在内存中，在多次计算间重用。简单来说，RDD是MapReduce模型的一种简单的扩展和延伸。

Spark基本编程方法示例:二次排序：数据分为两列，先对第一列数字按照升序排列分组，再在每组中按照第二列数据进行升序排序。本程序使用Scala语言编写，这个语言也是Spark开发和 编程的推荐语言。

```
def main(args: Array[String])
{
        //定义一个main函数
        val conf = new SparkConf().setAppName("二次排序")
        //定义一个sparkConf， 提供Spark运行的各种参数，如程序名称、用户名称等
        val sc = new SparkContext(conf)
        //创建Spark的运行环境，并将Spark运行的 参数传入Spark的运行环境中
        val fileRDD =sc.textFile("hdfs:///root/Log")
        //调用Spark的读文件函数，从HDFS 中读取Log文件，输出一个RDD类型的实例：fileRDD。具体类型： RDD[String]
        val rdd = fileRDD.map(x => x.replaceAll("\\s+","␣").split("␣")).map(x => (x(0),x(1)))
        //调用RDD的map函数， 对RDD中每一行执行括号中定义的函数， 第一个map将多个空白符替换成一个空格，并按照空格进行拆分。 第二个map将上一个map输出的每个数组只取前两个元素，转换为二元组。
        .groupByKey()
        //调用RDD的groupByKey函数， 该函数用于将RDD[K,V]中每个K对应的V值，合并到一个集合Iterable[V]中。 本示例中K是第一列数据，V是第二列数据。
        .mapValues(x => x.toList.sortBy(x=>x))
        //调用RDD的mapValues函数， 同基本转换操作中的map，区别是mapValues是针对[K,V]中的V值进行map 操作。本示例中将每一个K对应的集合Iterable[V]进行排序。
        .sortByKey()
        //调用RDD的sortByKey函数， 该函数用于将RDD[K,V]按照K的顺序进行排序，默认是升序。
        .foreach(println)
        //调用RDD的foreach函数， 该函数对RDD中每一行执行括号中定义的函数，和map函数区别在于： map：用于遍历RDD,将函数f应用于每一个元素， 返回新的RDD(transformation算子)。 foreach: 用于遍历RDD,将函数f应用于每一个元素， 无返回值(action算子)。
        sc.stop() //关闭Spark运行环境。
}
```

RDD的创建：RDD只能通过两种方式创建：①在驱动程序中并行化一个已经存在的集合，或者引用一个外部存储系统的数据集，例如共享的文件系、HDFS、HBase或其他Hadoop数据格式的数据源。eg:val rdd = sc.parallelize(1 to 100, 2) ////生成一个1到100的数组，并行化成RDD; ②其他RDD的数据上的确定性操作来创建(即Transformation)。eg: val filterRDD=file.filter(line=¿line.contains(“ERROR”))//通过file的filter操作生成一个新的filterRDD

RDD的操作: RDD支持两种类型的操作：①转换(transformation)：这是一种惰性操作，即使用这种方法时，只是定义了一个新的RDD，而并不马上计算新的RDD内部的值。 例：val filterRDD=fileRDD.filter(line=¿line.contains("ERROR")) //上述这个操作对于Spark来说仅仅记录从file这个RDD通过filter操作变换到filterRDD这个RDD的变换，并不计算filterRDD的结果。②动作(action)：立即计算这个RDD的值，并返回结果给程序，或者将结果写入到外存储中。 例：val result = filterRDD.count() 上述操作计算最终的result结果是多少，包括前边 transformation时的变换

Spark 支持的一些常用transformation操作(Figure 4, Figure 5, Figure 6)

Transformation	Meaning
map(func)	Return a new distributed dataset formed by passing each element of the source through a function *func*.
filter(func)	Return a new dataset formed by selecting those elements of the source on which *func* returns true.
flatMap(func)	Similar to map, but each input item can be mapped to 0 or more output items (so *func* should return a Seq rather than a single item).
mapPartitions(func)	Similar to map, but runs separately on each partition (block) of the RDD, so *func* must be of type Iterator<T> => Iterator<U> when running on an RDD of type T.
mapPartitionsWithIndex(func)	Similar to mapPartitions, but also provides *func* with an integer value representing the index of the partition, so *func* must be of type (Int, Iterator<T>) => Iterator<U> when running on an RDD of type T.
sample(withReplacement, fraction, seed)	Sample a fraction *fraction* of the data, with or without replacement, using a given random number generator seed.
union(otherDataset)	Return a new dataset that contains the union of the elements in the source dataset and the argument.

Figure 4: Spark的常用transformation操作-1

Spark支持的一些常用action操作 (Figure 7, Figure 8)
RDD的容错实现：存在两种容错的方式：①Lineage(世系系统、依赖系统)：RDD提供一种基于粗粒度变换的接口，这使得RDD可以 通过记录RDD之间的变换，而不需要存储实际的数据，就可以完成数据的恢复，使得Spark具有高效的容错性； ②CheckPoint(检查点)对于很长的lineage的RDD来说，通过lineage来恢复耗 时较长。因此，在对包含宽依赖的长世系的RDD设置检 查点操作非常有必要；由于RDD的只读特性使得Spark比常用的共享内存更容易完成checkpoint。

RDD之间的依赖关系：在Spark中存在两种类型的依赖：①窄依赖：父RDD中的一个Partition最多被子RDD中的一个Partition所依赖。②宽依赖：父RDD中的一个Partition被子RDD中的多个Partition所依赖。

RDD持久化：Spark提供了三种对持久化RDD的存储策略：①未序列化的Java对象，存于内存中；②序列化的数据，存于内存中(取消RDD对象，减少RDD存储开销，使用时需要反序列化恢复为RDD对象。更有效使用内存，但是降低了性能。)；③磁盘存储(开销大)。

RDD内部设计：每个RDD包含：一组RDD分区（partition），即数据集的原子组成部分；对父RDD的一组依赖，这些依赖描述了RDD的Lineage；一个函数，即在父RDD上执行何种计算；元数据，描述分区模式和数据存放的位置。

Transformation	Meaning
intersection(*otherDataset*)	Return a new RDD that contains the intersection of elements in the source dataset and the argument.
distinct([*numTasks*]))	Return a new dataset that contains the distinct elements of the source dataset.
groupByKey([*numTasks*])	When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs.
reduceByKey(*func*, [*numTasks*])	When called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function *func*, which must be of type (V,V) => V. Like in groupByKey, the number of reduce tasks is configurable through an optional second argument.
aggregateByKey(*zeroValue*)(*seqOp*, *combOp*, [*numTasks*])	When called on a dataset of (K, V) pairs, returns a dataset of (K, U) pairs where the values for each key are aggregated using the given combine functions and a neutral "zero" value. Allows an aggregated value type that is different than the input value type, while avoiding unnecessary allocations. Like in groupByKey, the number of reduce tasks is configurable through an optional second argument.
sortByKey([*ascending*], [*numTasks*])	When called on a dataset of (K, V) pairs where K implements Ordered, returns a dataset of (K, V) pairs sorted by keys in ascending or descending order, as specified in the boolean ascending argument.

Figure 5: Spark的常用transformation操作-2

Spark编程接口：Spark用Scala语言实现了RDD的API。

Spark支持三种语言的API: Scala, Python, Java

1.7 Spark安装运行模式

操作系统：Spark是运行在Java虚拟机上的.工具和脚本（如启动脚本）是针对Linux环境编写的,建议在Linux操作系统上安装运行Spark.

SSH

Java

Scala API / Python API

HDFS

基本安装步骤(Standlone模式)：软件环境准备(操作系统，Scala)； 下载编译好的Spark包；修改Spark配置文件；启动Spark；运行测试程序；查看集群状态。

Spark和集群管理工具：统一资源管理平台和集装箱思想(将应用和依赖“装箱”，一次配置，随处部署)：使用Yarn或Mesos运行Spark, 使用Docker部署Spark. 不同计算引擎各有所长，真实应用中往往需要同时使用不同 的计算框架(Spark, GraphLab, Flink)不同框架和应用会争抢资源，互相影响，使得管理难度和成本增加.应用往往在单机上开发，在小规模集群上测试，在云上运行。在不同环境下部署时总要经历复杂而且痛苦的配置过程。这些管理工具的作用：资源管理；资源共享；资源隔离；提高资源利用效率；扩展和容错。

2. Spark编程

2.1 WordCount

```
val file = spark.textFile("hdfs://...")
val counts = file.flatMap(line => line.split(" ")) //分词
        .map(word => (word, 1)) //对应mapper的工作
        .reduceByKey(_ + _) //相同key的不同value之间进行"+"运算
counts.saveAsTextFile("hdfs://...")
```

Transformation	Meaning
join(*otherDataset*, [*numTasks*])	When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (V, W)) pairs with all pairs of elements for each key. Outer joins are supported through leftOuterJoin,rightOuterJoin, and fullOuterJoin.
cogroup(*otherDataset*, [*numTasks*])	When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (Iterable<V>, Iterable<W>)) tuples. This operation is also called groupWith.
cartesian(*otherDataset*)	When called on datasets of types T and U, returns a dataset of (T, U) pairs (all pairs of elements).
pipe(*command*, [*envVars*])	Pipe each partition of the RDD through a shell command, e.g. a Perl or bash script. RDD elements are written to the process's stdin and lines output to its stdout are returned as an RDD of strings.
coalesce(*numPartitions*)	Decrease the number of partitions in the RDD to numPartitions. Useful for running operations more efficiently after filtering down a large dataset.
repartition(*numPartitions*)	Reshuffle the data in the RDD randomly to create either more or fewer partitions and balance it across them. This always shuffles all data over the network.
repartitionAndSortWithinPartitions(*partitioner*)	Repartition the RDD according to the given partitioner and, within each resulting partition, sort records by their keys. This is more efficient than calling repartition and then sorting within each partition because it can push the sorting down into the shuffle machinery.

Figure 6: Spark的常用transformation操作-3

3. Spark环境中其它功能组件简介

3.1 Spark SQL

用来处理结构化数据的分布式SQL查询引擎，具有以下几个特点：与Spark程序无缝对接(使用集成API和RDD模型查询结构化数据);统一数据访问接口(Hive, Parquet, Json);与Hive高度兼容; 使用标准链接.

3.2 Spark Streaming

对实时数据进行高吞吐量、容错处理的流式数据处理系统. 可以对多种数据源（Kafka、Flume、Twitter、TCP 套接字等）进行各种复杂处理（map、reduce、join、window等），处理结果可以输出到文件系统、数据库或实时显示。

工作机制：对数据流进行分片，使用Spark计算引擎处理分片数据，并返回相应分片的计算结果。

提供的基本流式数据抽象:DStream,由一系列连续的RDD表示.每个数据流分片被表示为一个RDD，对DStream的操作被转换成对相应RDD序列的操作。

3.3 GraphX

对图进行表示和并行处理的组件,把图抽象为：给每个顶点和边附着了属性的有向多重图

提供了一系列基本图操作（比如subgraph、joinVertices(连接顶点)、aggregateMessages(聚合消息)等）和优化了的Pregel API变种，并且各种图算法还在不断丰富中。

GraphX使用高效的点分割存储模式。图的每一条边只会存储在一台机器上，但顶点可能存储在 多台机器上.当点被分割到不同机器上时，是相同的镜像，但是有一个点作为主点(master), 其他的点作为虚点(ghost), 更新数据时先更新主点的数据，然后将数据发送到所有虚点的机器更新虚点。

Routing Table(路由表):顶点表中的一个划分对应着路由表中的一个划分，路由表指示了一个顶点会涉及到哪些边表划分.

Action	Meaning
reduce(*func*)	Aggregate the elements of the dataset using a function *func* (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel.
collect()	Return all the elements of the dataset as an array at the driver program. This is usually useful after a filter or other operation that returns a sufficiently small subset of the data.
count()	Return the number of elements in the dataset.
first()	Return the first element of the dataset (similar to take(1)).
take(*n*)	Return an array with the first *n* elements of the dataset.
takeSample(*withReplacement*,*num*, [*seed*])	Return an array with a random sample of *num* elements of the dataset, with or without replacement, optionally pre-specifying a random number generator seed.
takeOrdered(*n*, [*ordering*])	Return the first *n* elements of the RDD using either their natural order or a custom comparator.

Figure 7: Spark的常用transformation操作-3

3.4 MLlib

MLlib是Spark的分布式机器学习算法库，包含了很多常用机器学习算法和工具类.

Action	Meaning
saveAsTextFile(*path*)	Write the elements of the dataset as a text file (or set of text files) in a given directory in the local filesystem, HDFS or any other Hadoop-supported file system. Spark will call toString on each element to convert it to a line of text in the file.
saveAsSequenceFile(*path*) (Java and Scala)	Write the elements of the dataset as a Hadoop SequenceFile in a given path in the local filesystem, HDFS or any other Hadoop-supported file system. This is available on RDDs of key-value pairs that either implement Hadoop's Writable interface. In Scala, it is also available on types that are implicitly convertible to Writable (Spark includes conversions for basic types like Int, Double, String, etc).
saveAsObjectFile(*path*) (Java and Scala)	Write the elements of the dataset in a simple format using Java serialization, which can then be loaded using SparkContext.objectFile().
countByKey()	Only available on RDDs of type (K, V). Returns a hashmap of (K, Int) pairs with the count of each key.
foreach(*func*)	Run a function *func* on each element of the dataset. This is usually done for side effects such as updating an accumulator variable (see below) or interacting with external storage systems.

Figure 8: Spark的常用transformation操作-3

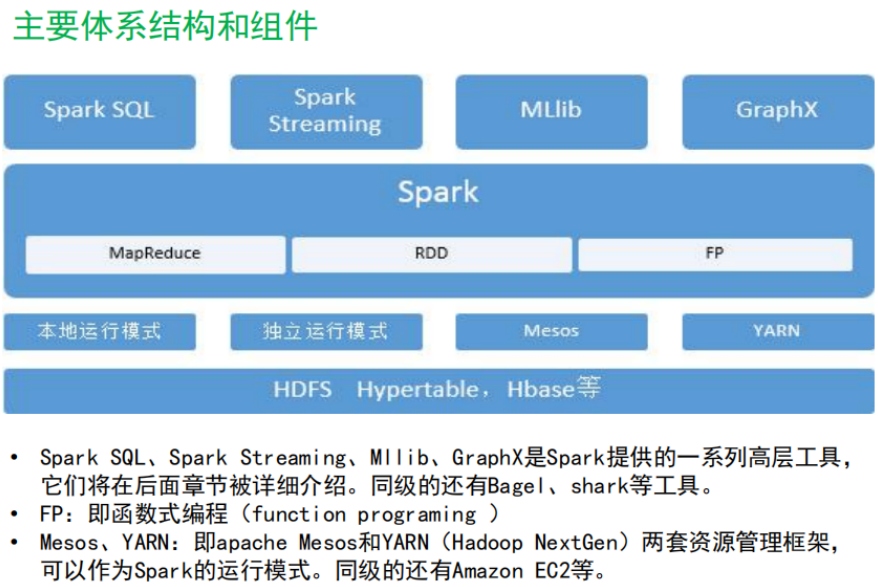

Figure 9: Spark主要体系结构和组件

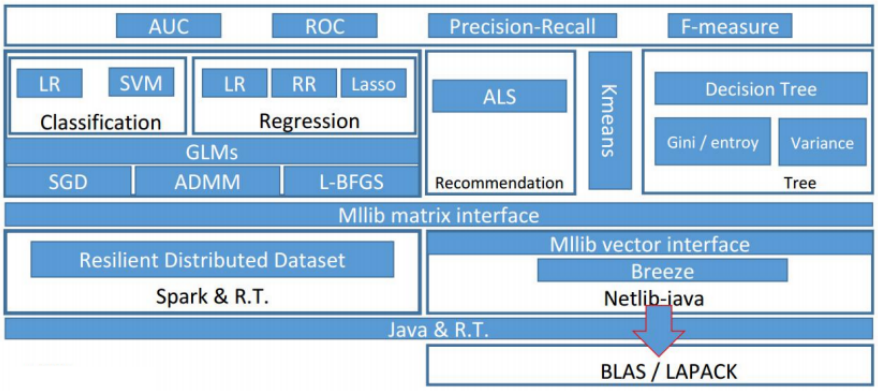

Figure 10: MLlib