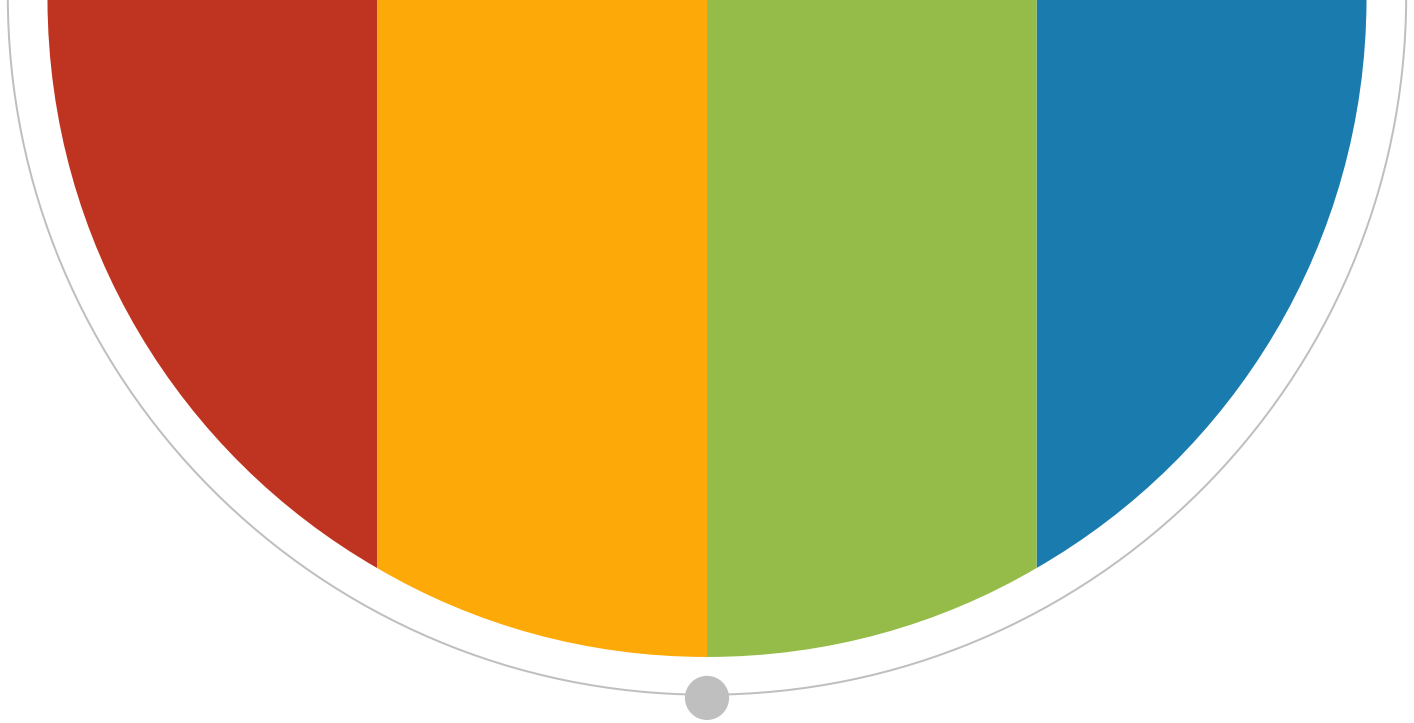


大数据处理与分析：Spark编程实践

王桂玲

北方工业大学信息学院数据工程研究院

大规模流数据集成与分析技术北京市重点实验室

wangguiling@ncut.edu.cn

1 Spark概述

2 Spark生态系统

3 Spark运行架构

4 Spark的部署方式

第三章 Spark设计与运行原理

PART ONE

Spark概述

Knowledge isn't free. You have to pay attention.

3.1 Spark 概述

- 3.1 Spark简介
- 3.2 MapReduce编程模型
- 3.3 Spark和Hadoop的比较

There's no shame in not knowing things! The only shame is to pretend that we know everything.

Richard Feynman

Spark简介

- Spark最初由美国加州大学伯克利分校（UC Berkeley）的AMP实验室于2009年开发，是基于内存计算的大数据并行计算框架，可用于构建大型的、低延迟的数据分析应用程序

- 2013年Spark加入Apache孵化器项目后发展迅猛，如今已成为Apache软件基金会最重要的三大分布式计算系统开源项目之一（Hadoop、Spark、Storm）

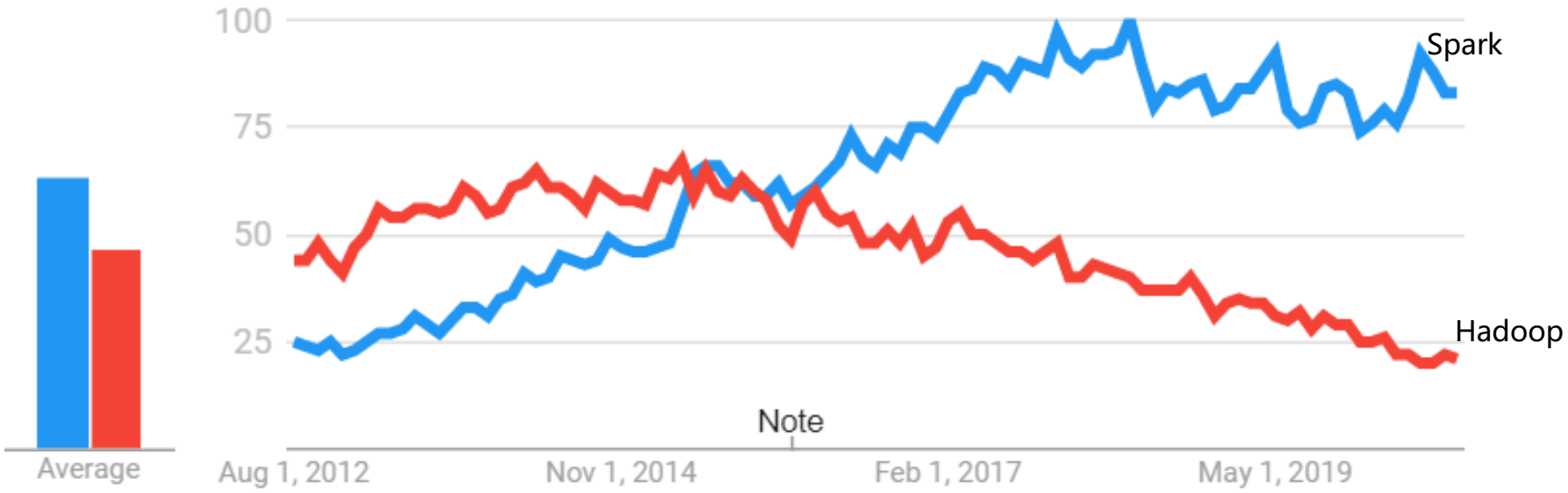
  - Spark在2014年打破了Hadoop保持的基准排序纪录
  - Spark/206个节点/23分钟/100TB数据
  - Hadoop/2000个节点/72分钟/100TB数据
  - Spark用十分之一的计算资源，获得了比Hadoop快3倍的速度

Spark简介

- Spark具有如下几个主要特点：
  - **运行速度快**：使用DAG执行引擎以支持循环数据流与内存计算
  - **容易使用**：支持使用Scala、Java、Python和R语言进行编程，可以通过Spark Shell进行交互式编程
  - **通用性**：Spark提供了完整而强大的技术栈，包括SQL查询、流式计算、机器学习和图算法组件
  - **运行模式多样**：可运行于独立的集群模式中，可运行于Hadoop中，也可运行于Amazon EC2等云环境中，并且可以访问HDFS、Cassandra、HBase、Hive等多种数据源

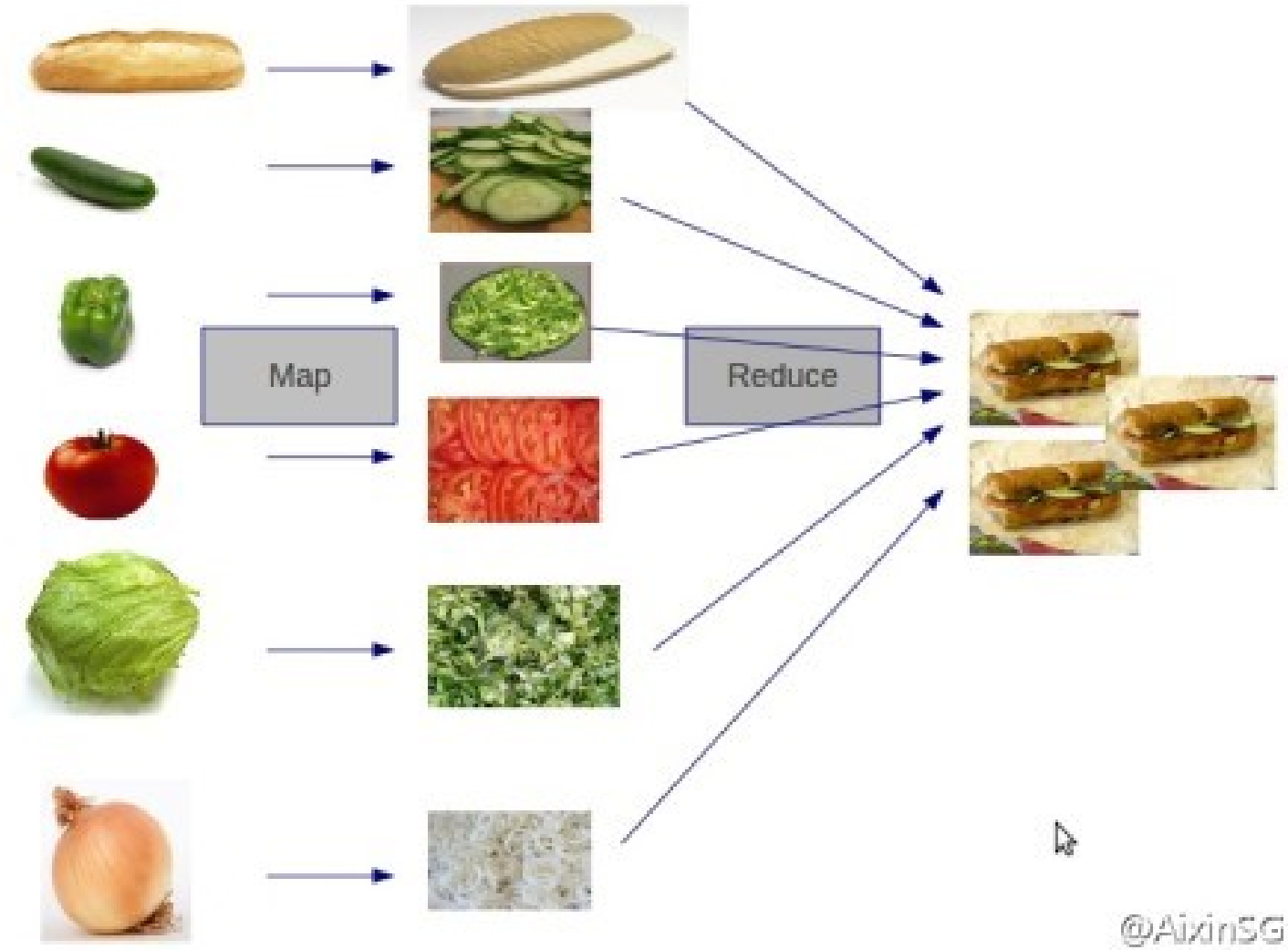
Spark简介

- Spark如今已吸引了国内外各大公司的注意，如腾讯、淘宝、百度、亚马逊等公司均不同程度地使用了Spark来构建大数据分析应用，并应用到实际的生产环境中



MapReduce编程模型

MapReduce编程模型（MapReduce Programming Model）：是一种在集群上进行大规模数据处理的分布式**并行**计算模型，它将数据的处理分成Map和Reduce**两个阶段**，其中Map过程实现数据的过滤和排序等处理，Reduce过程实现数据的汇总和合并。



MapReduce编程模型

- 示例
- 设有200亿个文档，要统计每个独立单词在这些文档里出现的次数
  - 如果每个文档有20K，总数据量为400T（1T=1024G）
  - 一台计算机读入这些数据大约需要1个月（150MB/s）

- 思考：用我们已有的编程知识，如果不考虑等待时间长、内存足够大的因素，程序应该怎么写？

MapReduce编程模型

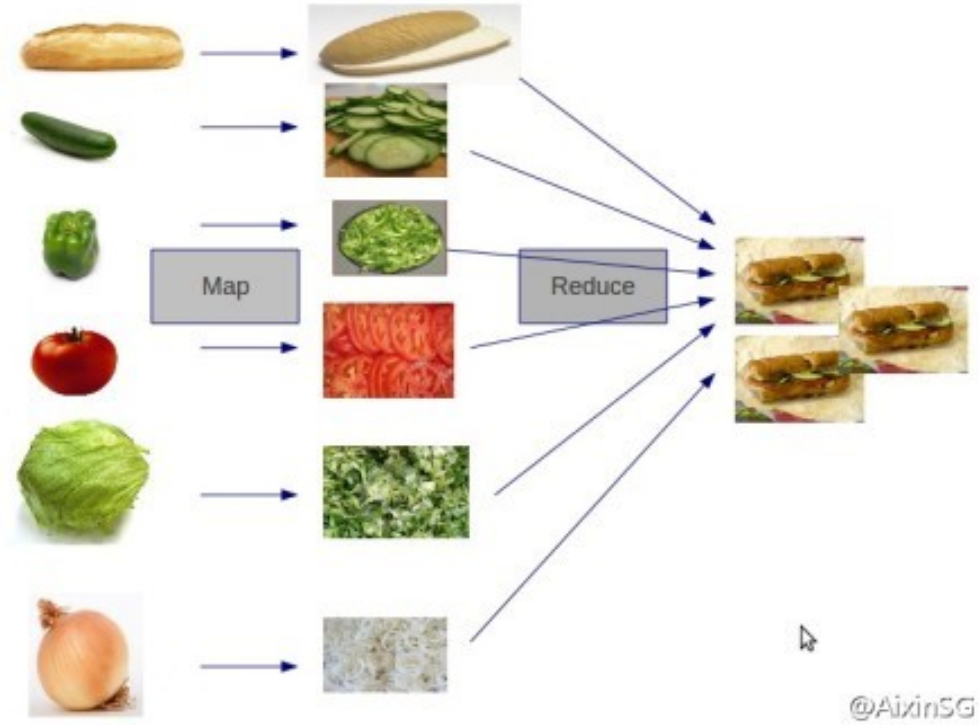
- 如果不怕等的时间长，机器的存储和内存足够大，程序大体逻辑：

```
map<string, int> word_count;  //定义一个计数的map容器
for each document d {
    for each word w in d {
    word_count[w]++;  //循环中遇到一个单词，就将对应的计数值加一
    }
}
... 把统计数据保存到持久性存储器...
```

- 问题：
- 200亿个文档，上述程序运行很慢、一台机器的存储也一次放不下全部文档，如何加速？

如何加速上述处理

- 提高单台机器的处理能力
- 几个线程同时处理、几台机器同时处理



多线程并行化的单词计数

示例2，并行化的单词计数程序

```
Mutex  lock; // 用一个锁保护word_count
map<string, int> word_count;
for each document d in parallel {
    for each word w in d {
       lock.Lock( );
       word_count[w]++;
       lock.Unlock( );
    }
}
```

… 把统计数据保存到持久性存储器…

- 文档输入工作可以很好地并行化
- 实际中启动线程有些麻烦，这里用伪代码，忽略了一些复杂描述
- 主要问题是共享的数据结构**word_count**。用一个数据结构可能造成严重的争用，成为性能瓶颈。可能改进是把数据结构分为很多桶（**buckets**），每个桶用一把锁，根据单词找锁和桶，可缓解这个问题

在几台机器上使用划分的进程进行处理

- 将数据和计算分布到多台机器上
- 在计算机集群上启动许多进程
- 输入进程：读入和处理所有文档的子集
- 输出进程：管理word_count结果数据集

- 假设机器不出故障
  - D是分配给每个输入进程的文档数
  - 开始时启动所有输入和输出进程
  - 一个输入进程里有一组统计表，对应每个输出进程有一个统计表。统计完成后将给各输出进程的表打包送去
  - 输出进程取得各输入进程送来的包，在自己的统计表里综合得到的结果

示例 23-4，并行单词计数程序，使用划分的进程

```
const int M = 1000;   // Number of input processes
const int R = 256;    // Number of output processes
main( ) {
  // Compute the number of documents to assign to each process
  const int D = number of documents / M;
  for (int i = 0; i < M; i++) {  fork InputProcess(i * D, (i + 1) * D);  }
  for (int i = 0; i < R; i++) {  fork OutputProcess(i);  }
  ... wait for all processes to finish ...
}

void InputProcess(int start_doc, int end_doc) {
  map<string, int> word_count[R];     // Separate table per output process
  for each doc d in range [start_doc .. end_doc-1] do {
    for each word w in d {  int b = hash(w) % R;  word_count[b][w]++;    }
  }
  for (int b = 0; b < R; b++) {
    string s = EncodeTable(word_count[b]);   ... send s to output process b ...
  }
}

void OutputProcess(int bucket) {
  map<string, int> word_count;
  for each input process p {
    string s = ... read message from p ...
    map<string, int> partial = DecodeTable(s);
    for each <word, count> in partial do {   word_count[word] += count;   }
  }
  ... save word_count to persistent storage ...
}
```

上述程序的问题

- 上述方法可以工作，但
  - 它比较复杂，难以理解；这里隐藏了打包和解包的细节
  - 仍没有看清楚启动和同步不同进程的细节
  - 没有很好的容错方法
- 处理出故障的机器，可以考虑扩展上述代码
  - 如果某台机器没完成工作，就重做相应工作
  - 处理故障很不容易，费力费时的代码是在并行计算管理的细节中
- 如果能把处理要解决问题的**计算和并行化细节分开**，就可能得到一种很好的**通用并行库或者编程系统**。这种系统不仅适用于做单词计数，还可能适用于完成其他大规模数据处理工作

MapReduce编程模型

- 由Google公司的工程师们提出并实现（2008年），现已经被工业界和学术界广泛接受。
- Hadoop是其开源实现；Spark计算模式也以MapReduce编程模型为基础
- MapReduce将复杂的运行于大规模集群上的并行计算过程高度抽象成了两个计算函数，Map和Reduce
- **向开发者隐藏并行化实现细节**

MapReduce编程模型（MapReduce Programming Model）：是一种在集群上进行大规模数据处理的分布式并行计算模型，它将数据的处理分成Map和Reduce两个阶段，其中Map过程实现数据的过滤和排序等处理，Reduce过程实现数据的汇总和合并。

MapReduce编程模型

- 用两个函数分别做一个文档的单词计数和文档计数归并
- 一个称为映射（map），另一个称为化简（reduce）
- map处理一个文档，对其中每个单词送出一个中间结果
- reduce接到一个单词和一些值（计数），归并计数（对其求和）后送出

MR模型为程序员提供了一个清晰的操作接口抽象描述

示例5　把单词计数问题分解为**Map**和**Reduce**

```
map(String input_key, String input_value):
    // input_key: document name
    // input_value: document contents
    for each word w in input_value:
      EmitIntermediate(w, "1");

reduce(String output_key, Iterator intermediate_values):
    // output_key: a word
    // output_values: a list of counts
    int result = 0;
    for each v in intermediate_values:
      result += ParseInt(v);
    Emit(AsString(result));
```

map: **(k1; v1)** → **[(k2; v2)]**

输入：键值对(k1; v1)表示的数据

输出：键值对[(k2; v2)]表示的一组中间数据

reduce: (k2; [v2]) → [(k3; v3)]

输入：　由map输出的一组键值对[(k2; v2)] 将被进行合并处理将同样主键下的不同数值合并到一个列表[v2]中，故reduce的输入为(k2; [v2])

输出：最终输出结果[(k3; v3)]

MapReduce的主要设计思想和特点

向“外”横向扩展，而非向“上”纵向扩展

Scale “out", not “up”

失效被认为是常态

Assume failures are common

把处理向数据迁移

Moving processing to the data

为应用开发者隐藏系统层细节

Hide system-level details from the application developer

平滑无缝的可扩展性

Seamless scalability

Spark与Hadoop的对比

- Hadoop是MapReduce编程模型的典型实现
- Hadoop存在如下一些缺点：
  - 表达能力有限
  - 磁盘IO开销大
  - 延迟高
    - ⑩ 任务之间的衔接涉及IO开销
    - ⑩ 在前一个任务执行完成之前，其他任务就无法开始，难以胜任复杂、多阶段的计算任务

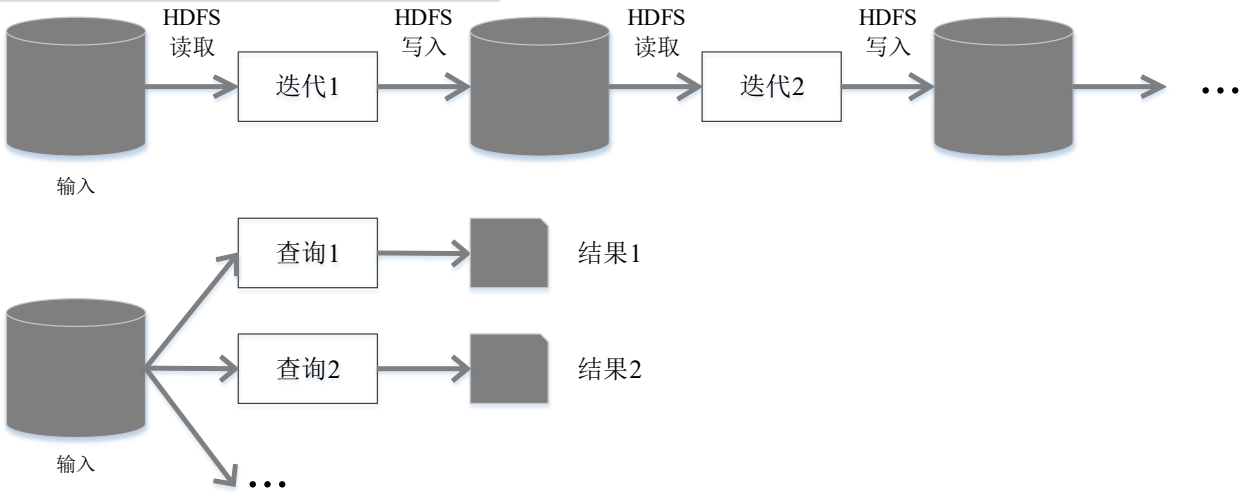
Spark和Hadoop的对比

- Spark在借鉴Hadoop MapReduce优点的同时，很好地解决了MapReduce所面临的问题

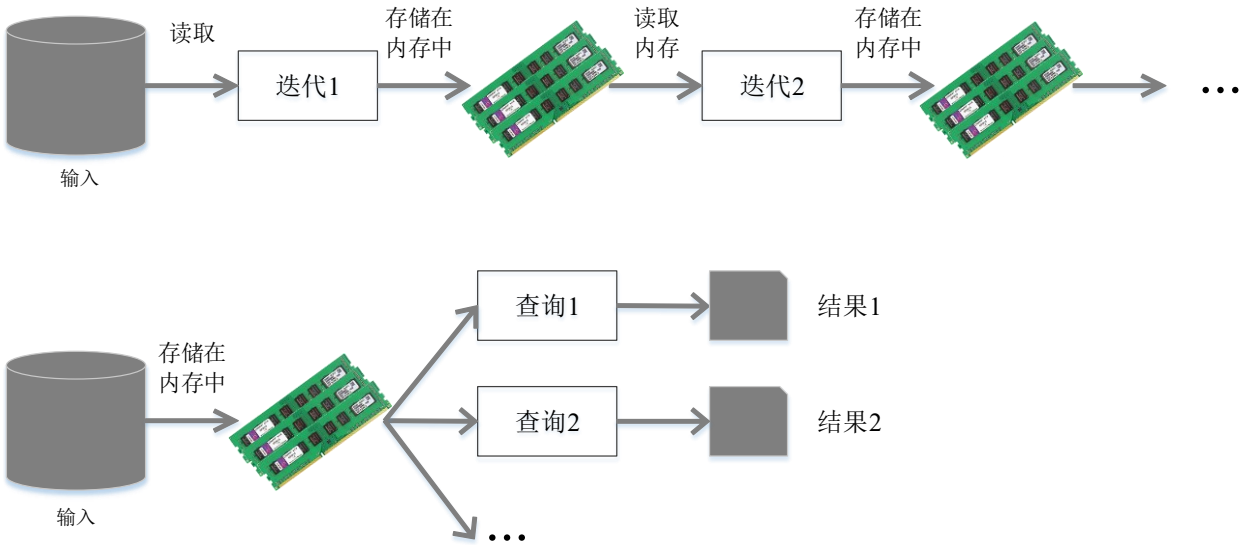
- 相比于Hadoop MapReduce，Spark主要具有如下优点：

  - Spark的计算模式也属于MapReduce，但不局限于Map和Reduce操作，还提供了多种数据集操作类型，编程模型比Hadoop MapReduce更灵活
  - Spark提供了内存计算，可将中间结果放到内存中，对于迭代运算效率更高
  - Spark基于DAG的任务调度执行机制，要优于Hadoop MapReduce的迭代执行机制

Spark与Hadoop的对比



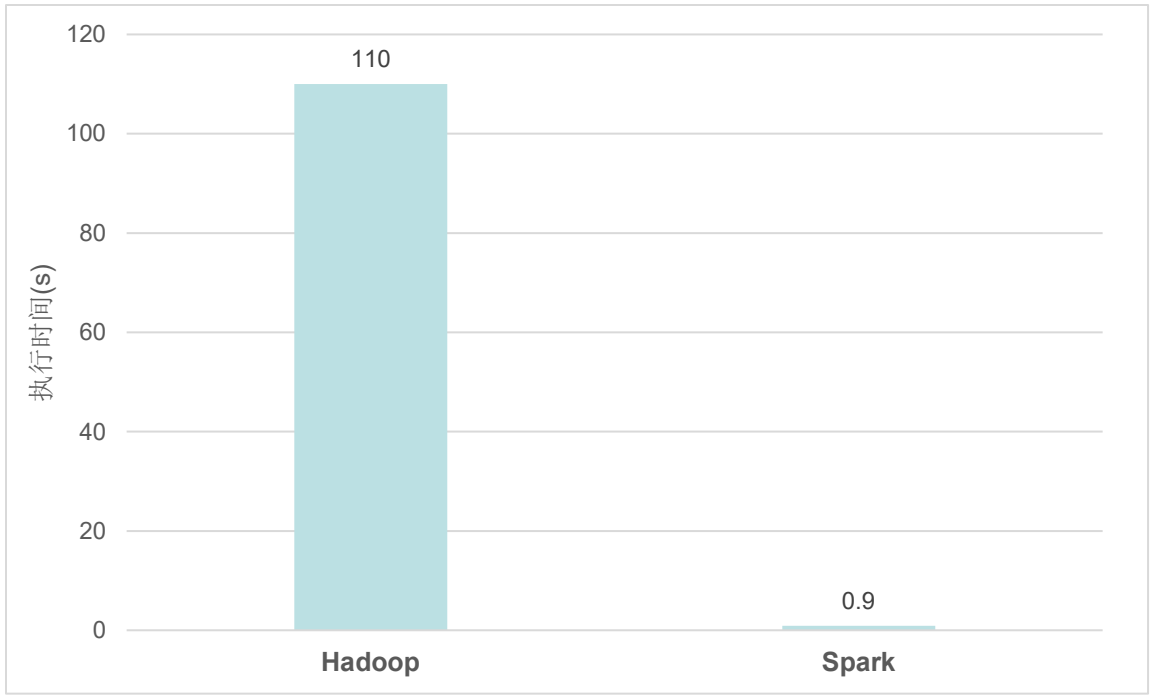
(a) Hadoop MapReduce执行流程



(b) Spark执行流程

Spark与Hadoop的对比

- 使用Hadoop进行迭代计算非常耗资源
- Spark将数据载入内存后，之后的迭代计算都可以直接使用内存中的中间结果作运算，避免了从磁盘中频繁读取数据



Hadoop与Spark执行逻辑回归的时间对比

2

PART TWO

Spark生态系统

Spark生态系统

- 在实际应用中，大数据处理主要包括以下三个类型：
  - 复杂的批量数据处理：通常时间跨度在数十分钟到数小时之间
  - 基于历史数据的交互式查询：通常时间跨度在数十秒到数分钟之间
  - 基于实时数据流的数据处理：通常时间跨度在数百毫秒到数秒之间
- 当同时存在以上三种场景时，就需要同时部署三种不同的软件：比如：MapReduce / Impala / Storm等

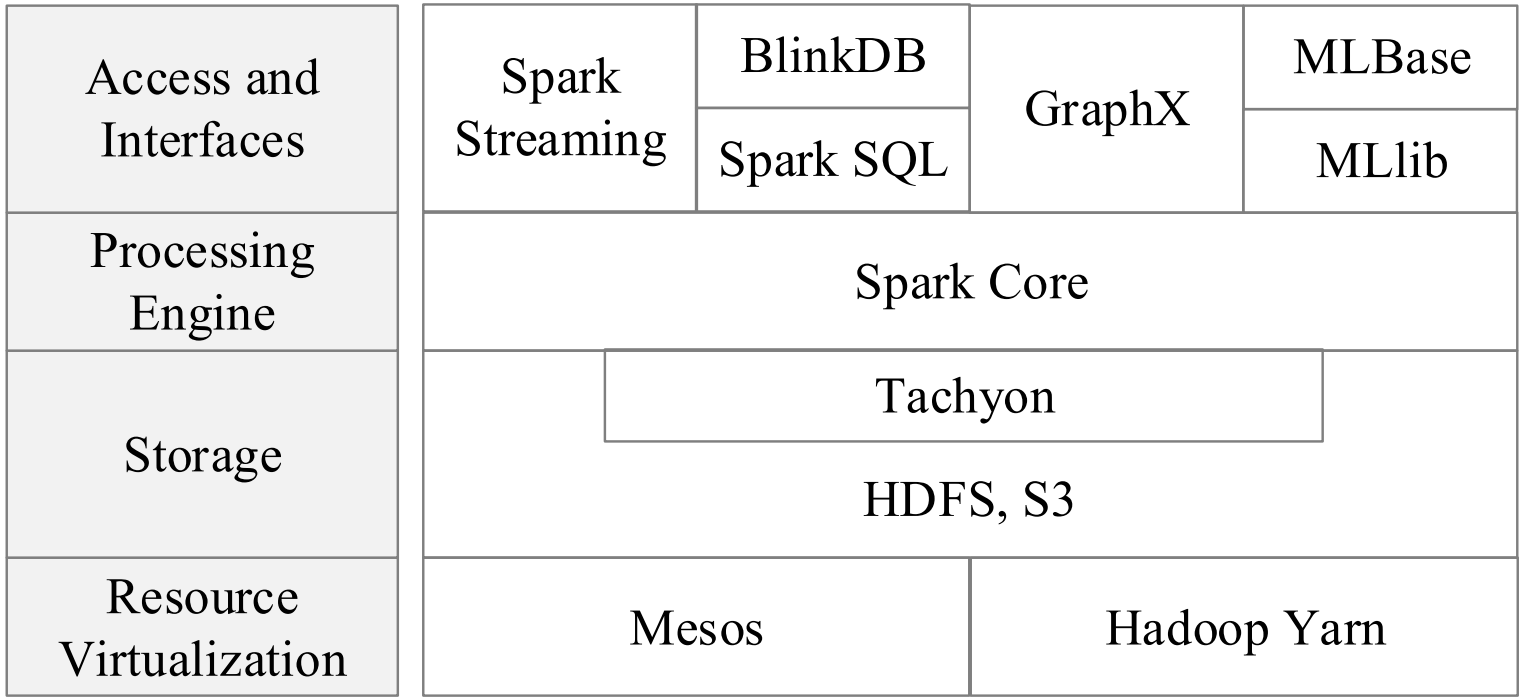
- 这样做难免会带来一些问题：
  - 不同场景之间输入输出数据无法做到无缝共享，通常需要进行数据格式的转换
  - 不同的软件需要不同的开发和维护团队，带来了较高的使用成本
  - 比较难以对同一个集群中的各个系统进行统一的资源协调和分配

Spark生态系统

- Spark的设计遵循“一个软件栈满足不同应用场景”的理念，逐渐形成了一套完整的生态系统
- 既能够提供内存计算框架，也可以支持SQL即席查询、实时流式计算、机器学习和图计算等
- Spark可以部署在资源管理器YARN之上，提供一站式的大数据解决方案
- 因此，Spark所提供的生态系统足以应对上述三种场景，即同时支持批处理、交互式查询和流数据处理

Spark生态系统

- Spark生态系统已经成为伯克利数据分析软件栈BDAS（Berkeley Data Analytics Stack）的重要组成部分
- Spark的生态系统主要包含了Spark Core、Spark SQL、Spark Streaming（ Structured Streaming ）、MLLib和GraphX 等组件



Spark生态系统组件的应用场景

应用场景	时间跨度	其他框架	Spark生态系统中的组件
复杂的批量数据处理	小时级	MapReduce、Hive	Spark
基于历史数据的交互式查询	分钟级、秒级	Impala、Dremel、Drill	Spark SQL
基于实时数据流的数据处理	毫秒、秒级	Storm、S4	Spark Streaming Structured Streaming
基于历史数据的数据挖掘	-	Mahout	MLlib
图结构数据的处理	-	Pregel、Hama	GraphX

课堂习题

- 下述哪些说法是正确的?

A. MR的主要优点是应用逻辑与并行/故障处理等底层细节分离
B. 使用MR模型进行程序开发，开发者仍然需要了解系统层细节
C. MR模型将数据的处理分成Map和Reduce两个阶段
D. 使用MR模型进行集群中并行程序的开发，开发者仍然需要进行机器故障处理

3

PART THREE

Spark运行架构

Spark运行架构

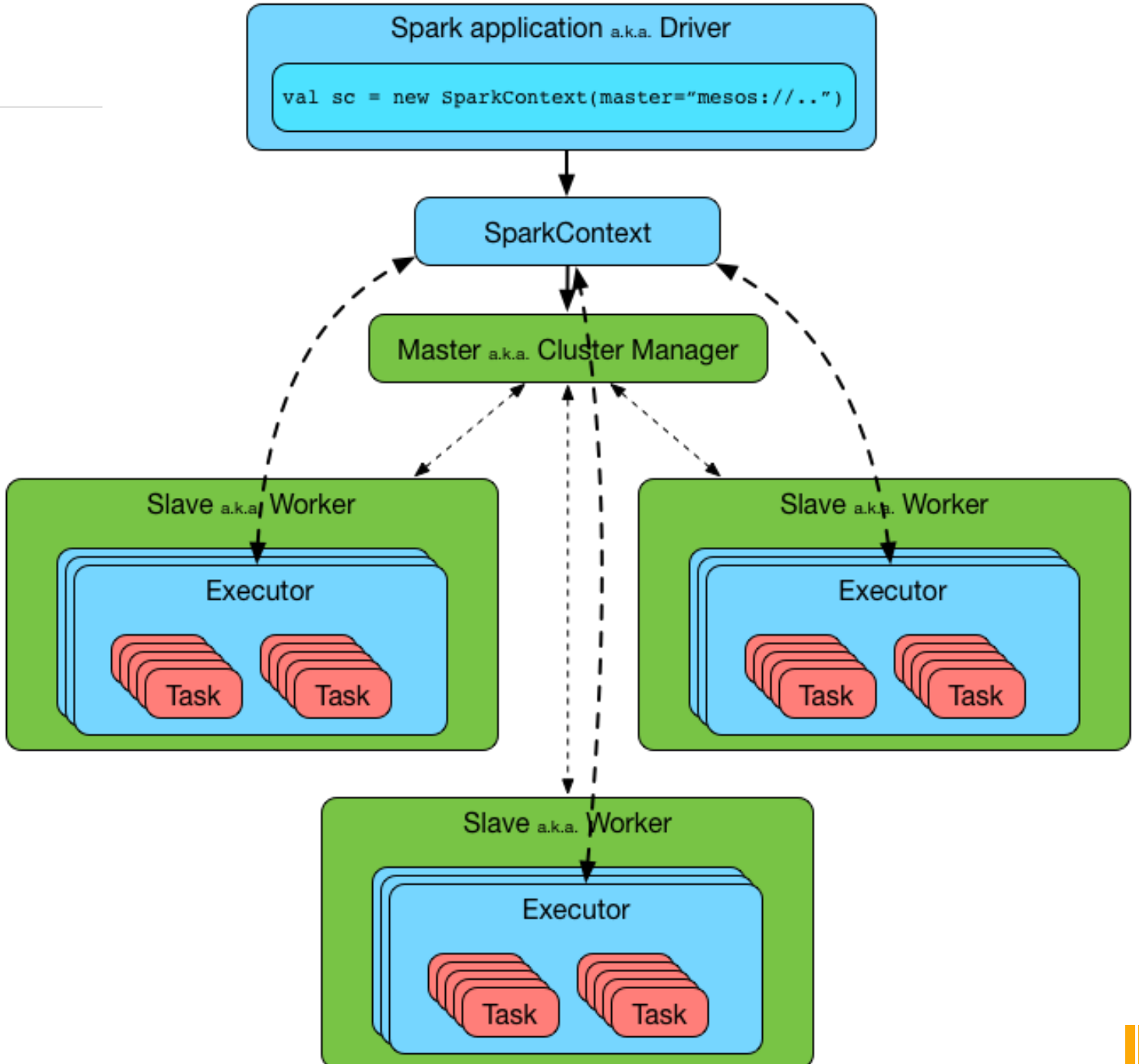
- 3.3.1 基本概念
- 3.3.2 架构设计
- 3.3.3 Spark运行基本流程
- 3.3.4 RDD的设计与运行原理

基本概念

- RDD：是Resillient Distributed Dataset（弹性分布式数据集）的简称，是分布式内存的一个抽象概念，提供了一种高度受限的共享内存模型
- DAG：是Directed Acyclic Graph（有向无环图）的简称，反映RDD之间的依赖关系
- Executor：是运行在工作节点（WorkerNode）的一个进程，负责运行Task
- 应用（Application）：用户编写的Spark应用程序
- 任务（Task）：运行在Executor上的工作单元
- 作业（Job）：一个作业包含多个RDD及作用于相应RDD上的各种操作
- 阶段（Stage）：是作业的基本调度单位，一个作业会分为多组任务，每组任务被称为阶段，或者也被称为任务集合

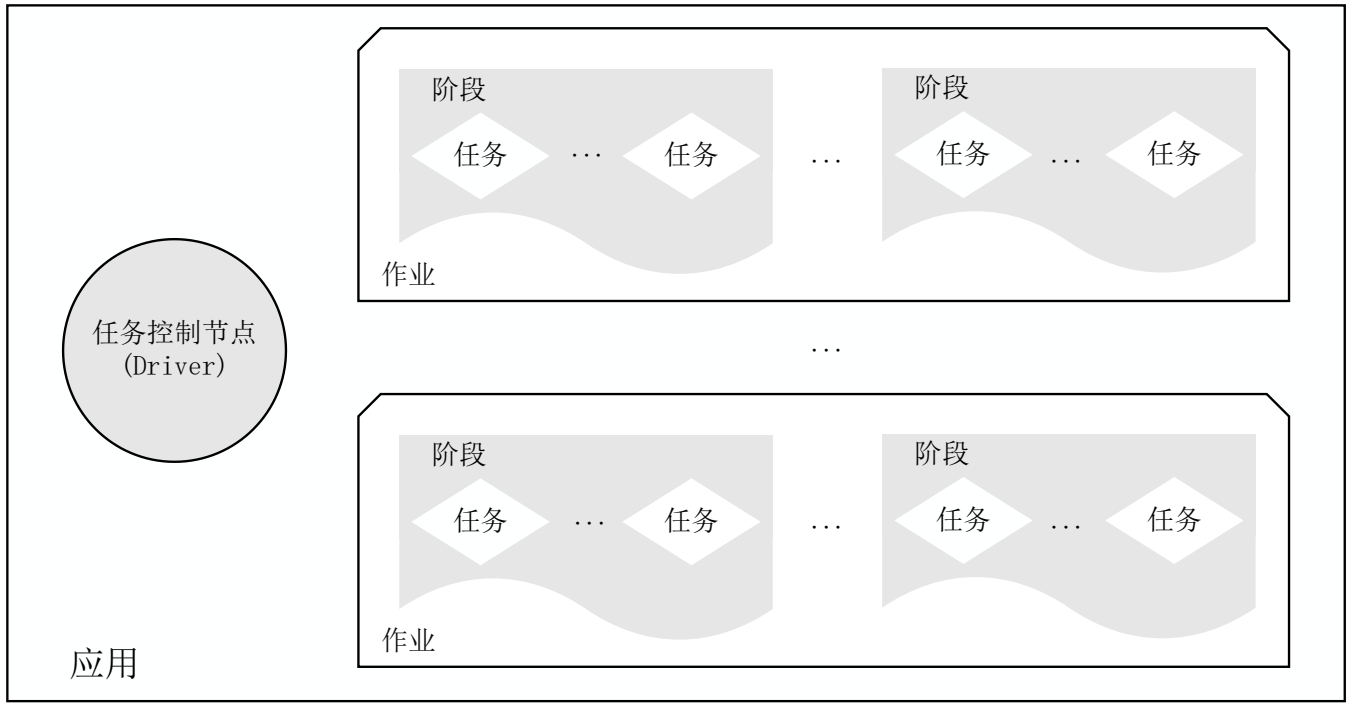
架构设计

- Spark运行架构包括集群资源管理器（Cluster Manager）、运行作业任务的工作节点（Worker Node）、每个应用的**任务控制节点（Driver）**和每个工作节点上负责具体任务的执行进程（Executor）
- 资源管理器可以自带或Mesos或YARN

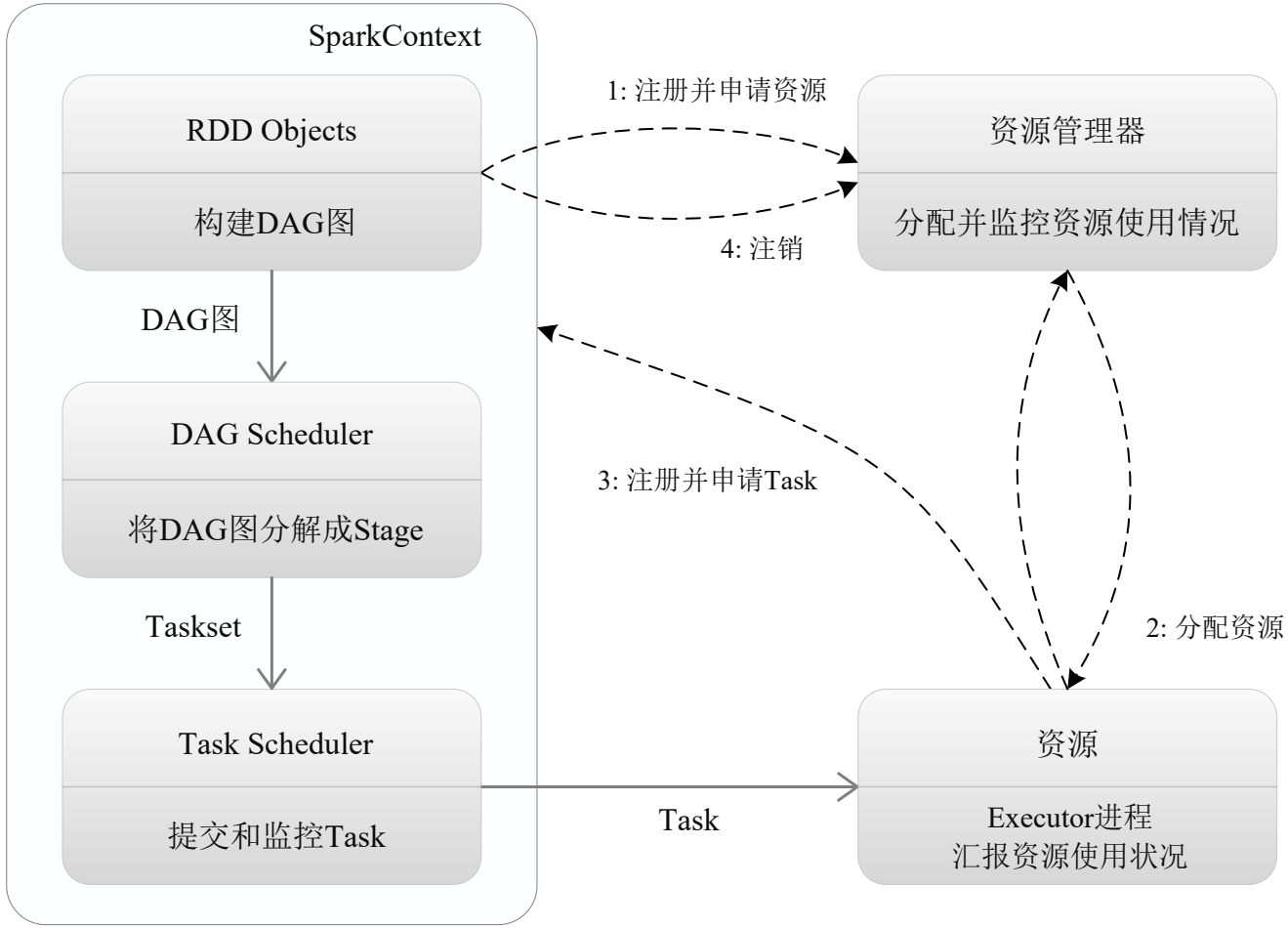


架构设计

- 一个应用由一个Driver和若干个作业构成，一个**作业**由多个**阶段**构成，一个阶段由多个没有Shuffle关系的**任务**组成
- 当执行一个应用时，Driver会向集群管理器申请资源，启动Executor，并向Executor发送应用程序代码和文件，然后在Executor上执行任务，运行结束后，执行结果会返回给Driver，或者写到HDFS或者其他数据库中



运行基本流程图



SparkContext对象代表了和一个集群的连接

- （1）首先为应用构建起基本的运行环境，即由Driver创建一个SparkContext，进行资源的申请、任务的分配和监控
- （2）资源管理器为Executor分配资源，并启动Executor进程
- （3）SparkContext根据RDD的依赖关系构建DAG图，DAG图提交给DAGScheduler解析成Stage，然后把一个个TaskSet提交给底层调度器TaskScheduler处理；Executor向SparkContext申请Task，Task Scheduler将Task发放给Executor运行，并提供应用程序代码
- （4）Task在Executor上运行，把执行结果反馈给TaskScheduler，然后反馈给DAGScheduler，运行完毕后写入数据并释放所有资源

RDD设计与运行原理

- 1.RDD设计背景
- 2.RDD概念
- 3.RDD特性
- 4.RDD之间的依赖关系
- 5.阶段的划分
- 6.RDD运行过程

RDD设计背景

- 许多迭代式算法（比如机器学习、图算法等）和交互式数据挖掘工具，共同之处是，不同计算阶段之间会重用中间结果
- 目前的MapReduce框架都是把中间结果写入到稳定存储（比如磁盘）中，带来了大量的数据复制、磁盘IO和序列化开销
- RDD就是为了满足这种需求而出现的，它提供了一个抽象的数据架构，我们不必担心底层数据的分布式特性，只需将具体的应用逻辑表达为一系列转换处理，不同RDD之间的转换操作形成依赖关系，可以实现管道化，避免中间数据存储

什么是RDD

- 一个RDD就是一个分布式对象集合，本质上是一个只读的分区记录集合，每个RDD可分成多个分区，每个分区就是一个数据集片段，并且一个RDD的不同分区可以被保存到集群中不同的节点上，从而可以在集群中的不同节点上进行并行计算

- RDD提供了一种高度受限的共享内存模型，即RDD是只读的记录分区的集合，不能直接修改，只能基于稳定的物理存储中的数据集创建RDD，或者通过在其他RDD上执行确定的转换操作（如map、join和group by）而创建得到新的RDD

RDD概念

- RDD提供了一组丰富的操作以支持常见的数据运算，分为“动作”（Action，有时翻译为活动、行动）和“转换”（Transformation）两种类型
- RDD提供的转换接口都非常简单，都是类似map、filter、groupBy、join等粗粒度的数据转换操作，而不是针对某个数据项的细粒度修改（不适合网页爬虫）
- 表面上RDD的功能很受限、不够强大，实际上RDD已经被实践证明可以高效地表达许多框架的编程模型（比如MapReduce、SQL、Pregel）
- Spark提供了RDD的API，程序员可以通过调用API实现对RDD的各种操作

RDD概念

- RDD典型的执行过程如下：
  - RDD读入外部数据源进行创建
  - RDD经过一系列的转换（Transformation）操作，每一次都会产生不同的RDD，供给下一个转换操作使用
  - 最后一个RDD经过“动作”操作进行转换，并输出到外部数据源
- 这一系列处理称为一个Lineage（血缘关系），即DAG拓扑排序的结果
- **优点：惰性调用、管道化、避免同步等待、不需要保存中间结果、每次操作变得简单**

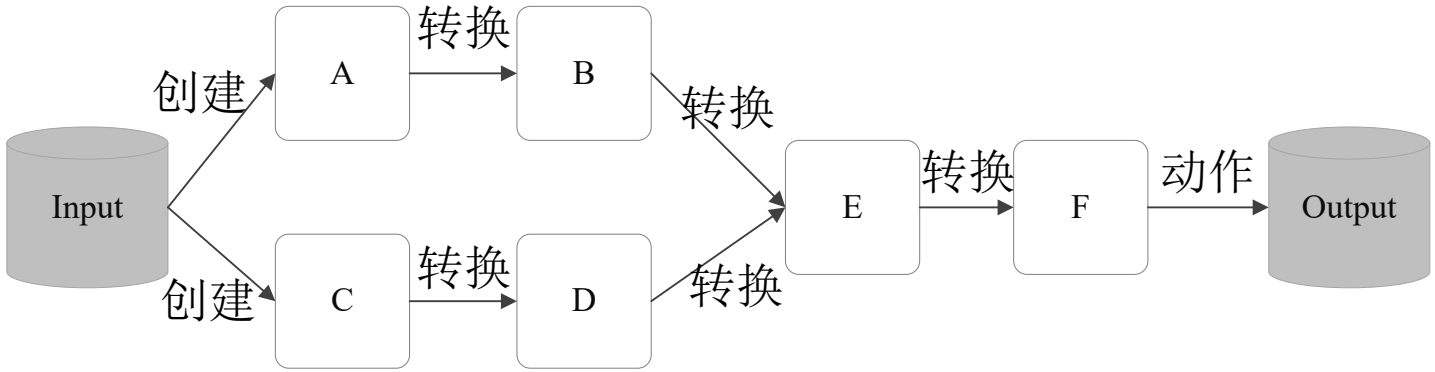


图 RDD执行过程的一个实例

RDD特性

- Spark采用RDD以后能够实现高效计算的原因主要在于：
- （1）高效的容错性
  - 现有容错机制：数据复制或者记录日志
  - RDD：血缘关系、重新计算丢失分区、无需回滚系统、重算过程在不同节点之间并行、只记录粗粒度的操作
- （2）中间结果持久化到内存，数据在内存中的多个RDD操作之间进行传递，避免了不必要的读写磁盘开销
- （3）存放的数据可以是Java对象，避免了不必要的对象序列化和反序列化

课堂习题

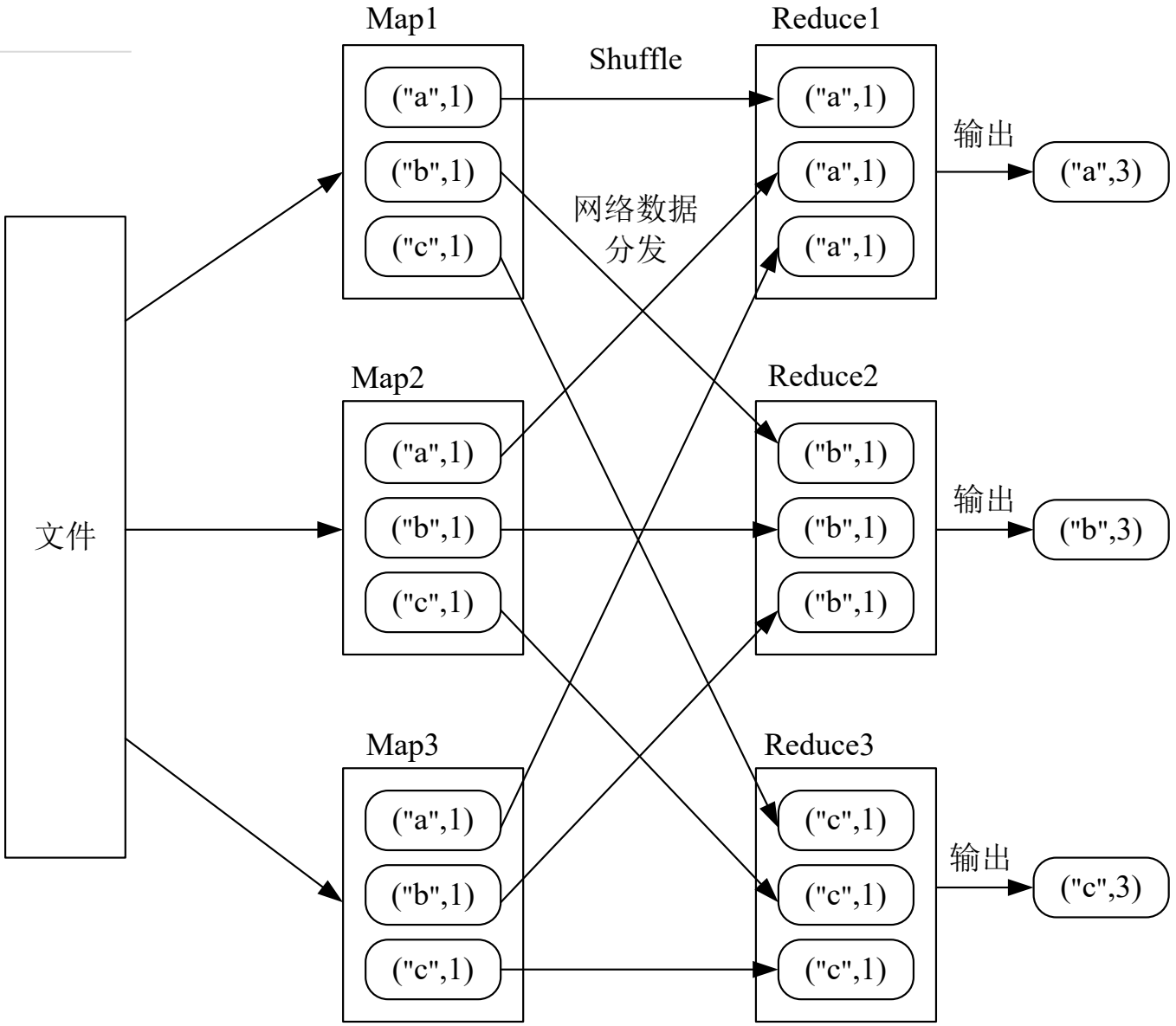
- 下列哪些说法是正确的：
  - A. RDD是Resillient Distributed Dataset（弹性分布式数据集）的简称，是分布式内存的一个抽象概念，提供了一种高度受限的共享内存模型
  - B. 一个Spark作业（Job）只能包含一个RDD及作用于其上的各种操作
  - C. RDD上的操作分为“动作”（Action，有时翻译为活动、行动）和“转换”（Transformation）两种类型
  - D. RDD操作都是把中间结果写入到稳定存储（比如磁盘）中

RDD之间的依赖关系

- Shuffle操作
  - 什么是Shuffle操作
- 窄依赖和宽依赖
  - 是否包含Shuffle操作是区分窄依赖和宽依赖的根据

Shuffle操作

- Shuffle是把Map输出的中间结果分发到Reduce任务所在的及其的过程。这个过程会产生大量的网络数据分发，带来高昂的网络传输开销和I/O开销。

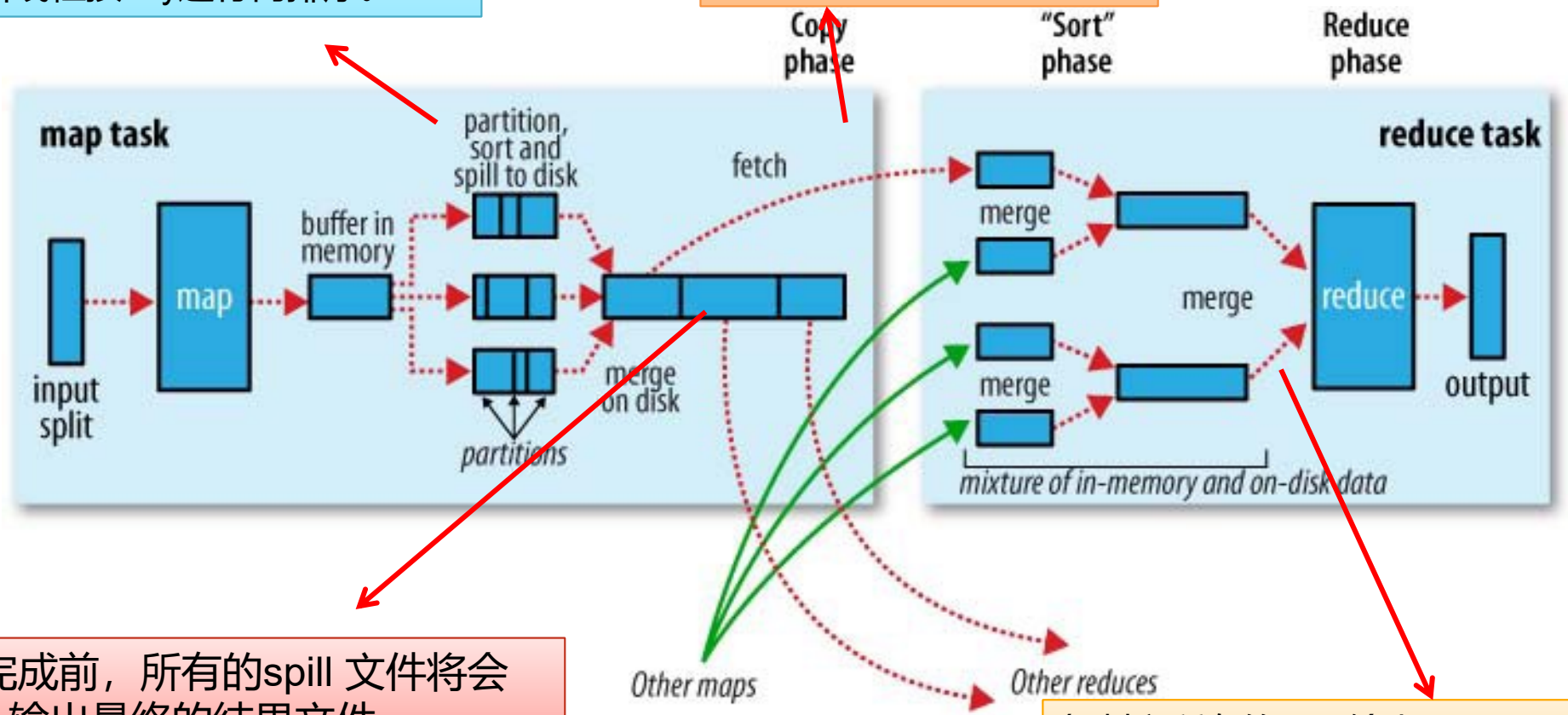


一个关于Shuffle 操作的简单实例

写到内存缓冲区，每当达到阈值后，启动后台spill线程开始写磁盘(spill文件)。在写磁盘前，线程根据数据最终要传送到的reducer把数据划分成相应的分区，在每个分区中，后台线程按key进行内排序。

Reduce任务需要集群上若干个map任务的map输出，每个map任务的完成时间不同，只要有一个map任务完成，reduce任务就开始复制。

课外思考：Spark的Shuffle过程仍然需要把数据写入磁盘，为什么？

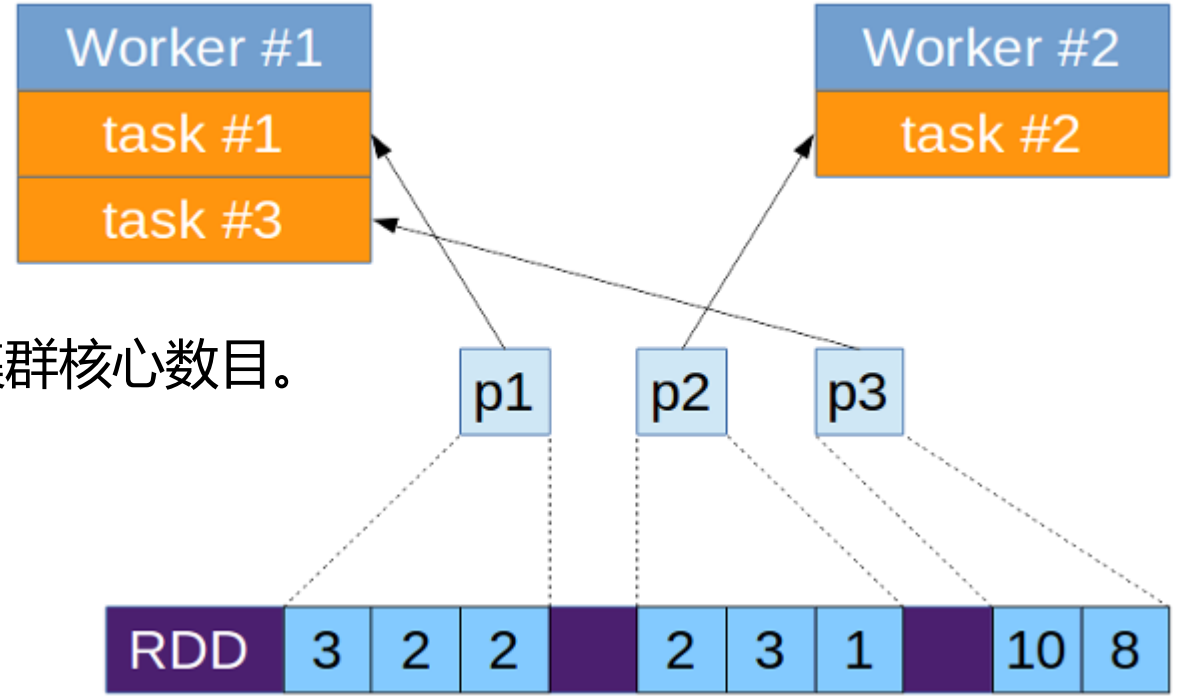


在Map 任务完成前，所有的spill 文件将会被归并排序，输出最终的结果文件

复制完所有的map输出，**reduce 任务将合并map输出，默认维持其顺序进行排序[Spark进行了改进，这里不排序]**

RDD的分区

- RDD 内部，如何表示并行计算的一个计算单元?
  - 使用**分区（Partition）**
  - 每个RDD可分成多个分区，每个分区就是一个数据集片段，并且一个RDD的不同分区可以被保存到集群中不同的节点上，从而可以在集群中的不同节点上进行并行计算
  - 一个task对应一个分区
  - 一个task对应worker节点上的Executor进程的一个线程
  - RDD 分区的一个分配原则是尽可能使得分区的个数，等于集群核心数目。



父RDD和子RDD

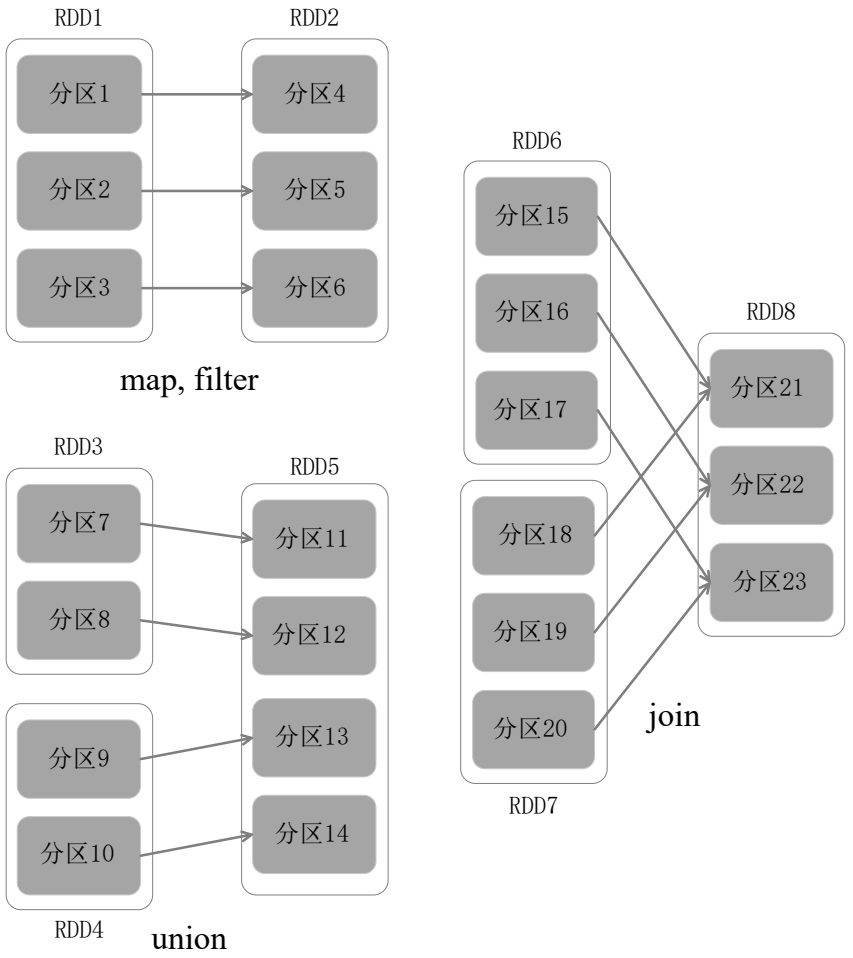
- 在一次转换操作中，创建得到的新 RDD 称为子 RDD，提供数据的 RDD 称为父 RDD，父 RDD 可能会存在多个，我们把子 RDD 与父 RDD 之间的关系称为依赖关系，或者可以说是子 RDD 依赖于父 RDD 。
- 例如
- RDD1：
  - Hadoop is good.
  - Spark is fast.
- filter()转换操作过滤其中不包含spark的行
- 生成RDD2：
  - Hadoop is good
- 那么，RDD2是RDD1的子RDD，而RDD1是父RDD

父RDD和子RDD

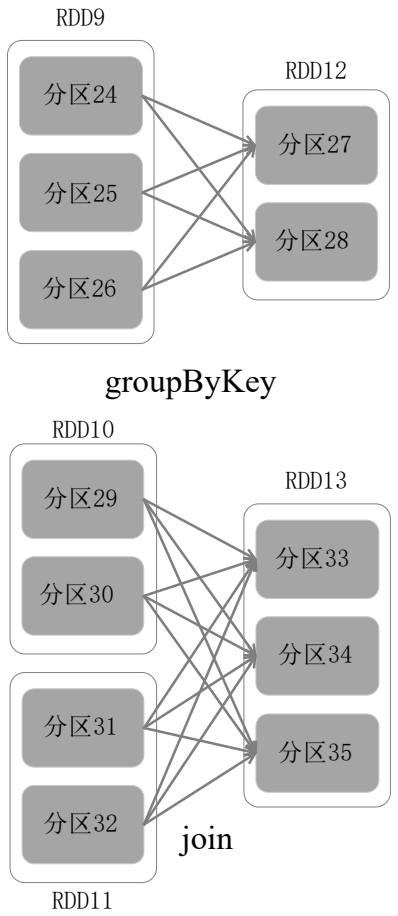
- 再例如：
- RDD1（学号，姓名，成绩）
  - 101, wang, 80
  - 102, li, 90
- RDD2（学号，姓名，家庭住址）
  - 101, wang, Beijing
  - 102, li, Shandong
- Join转换操作，将RDD1和RDD2中学号相同的记录连接为一行
- 生成RDD3（学号，姓名，成绩，家庭住址）
  - 101, wang, 80, Beijing
  - 102, li, 90, Shandong
- RDD3是RDD1和RDD2的子RDD，它有两个父RDD

RDD之间的依赖关系——窄依赖和宽依赖

- 窄依赖表现为一个父RDD的分区对应于一个子RDD的分区或多个父RDD的分区对应于一个子RDD的分区
- 宽依赖则表现为存在一个父RDD的一个分区对应一个子RDD的多个分区



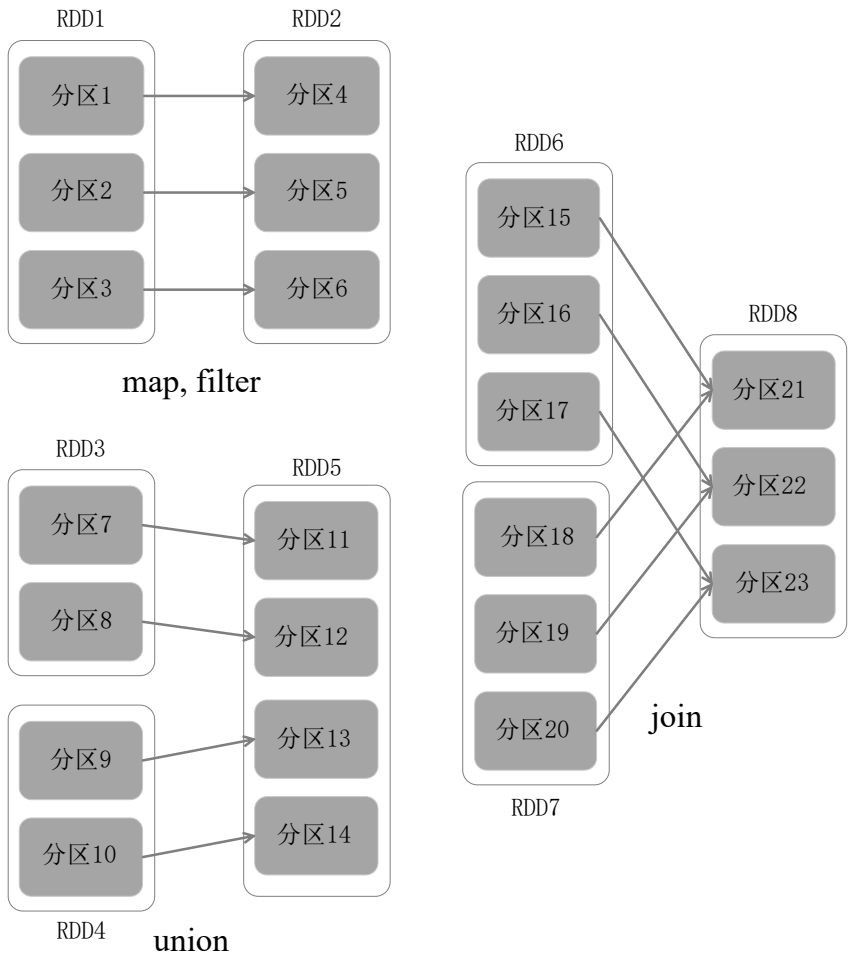
(a)窄依赖



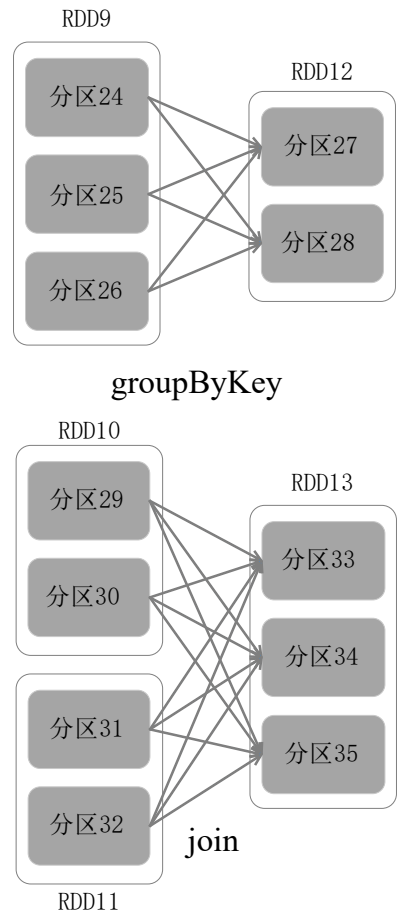
(b)宽依赖

窄依赖

- 一个父RDD的分区对应于一个子RDD的分区或多个父RDD的分区对应于一个子RDD的分区
- 父 RDD 中的一个分区最多只会被子 RDD 中的一个分区使用
- 父 RDD 中，一个分区内的数据是不能被分割的，必须整个交付给子 RDD 中的一个分区



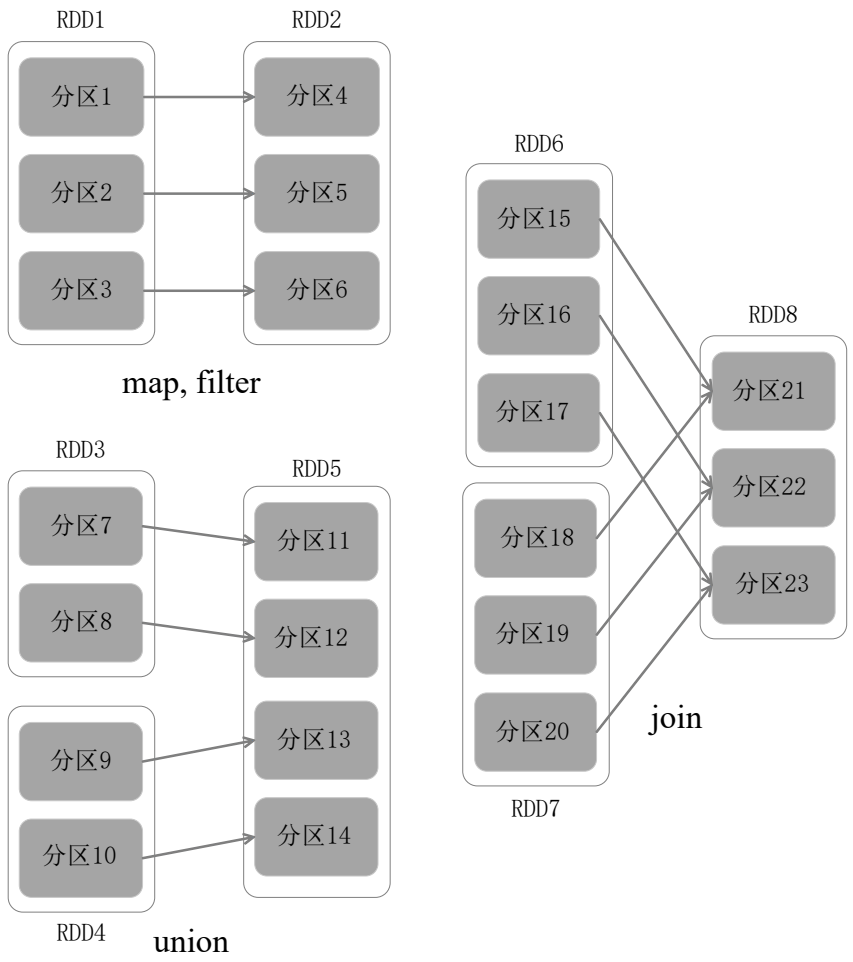
(a)窄依赖



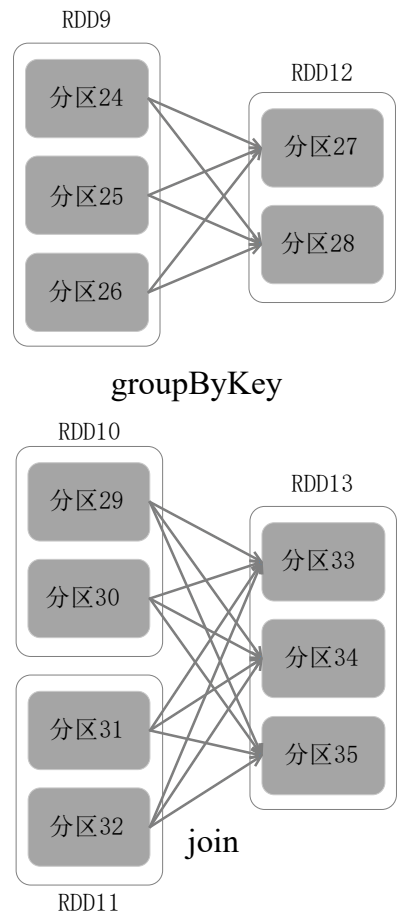
(b)宽依赖

宽依赖

- 宽依赖则表现为存在一个父RDD的一个分区对应一个子RDD的多个分区
- 父 RDD 中一个分区内的数据会被分割，发送给子 RDD 的多个分区



(a)窄依赖



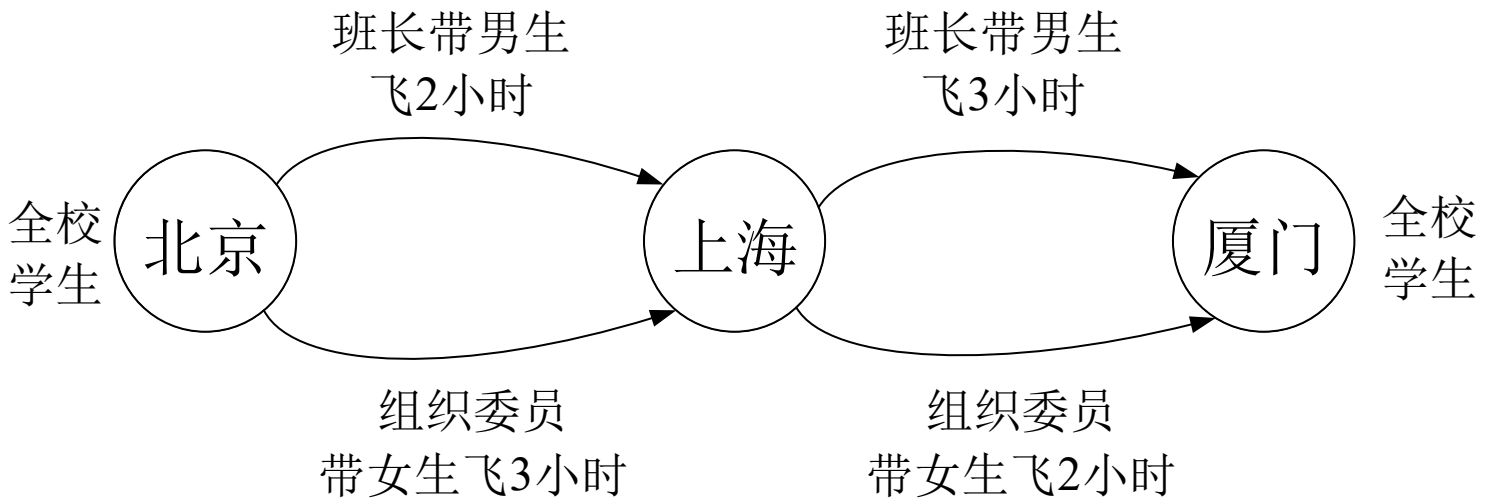
(b)宽依赖

阶段的划分

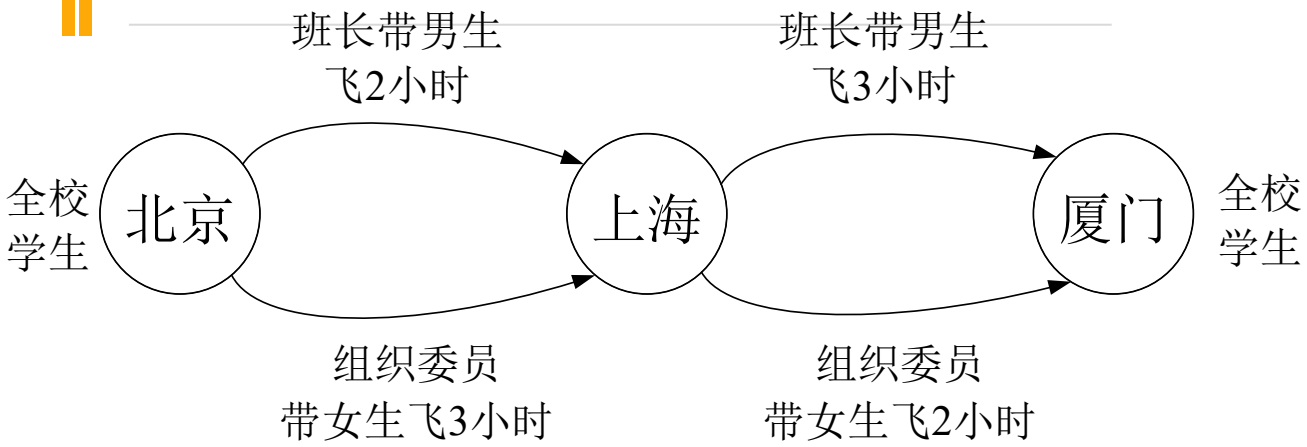
- 窄依赖可以实现“流水线”优化
- 宽依赖无法实现“流水线”优化

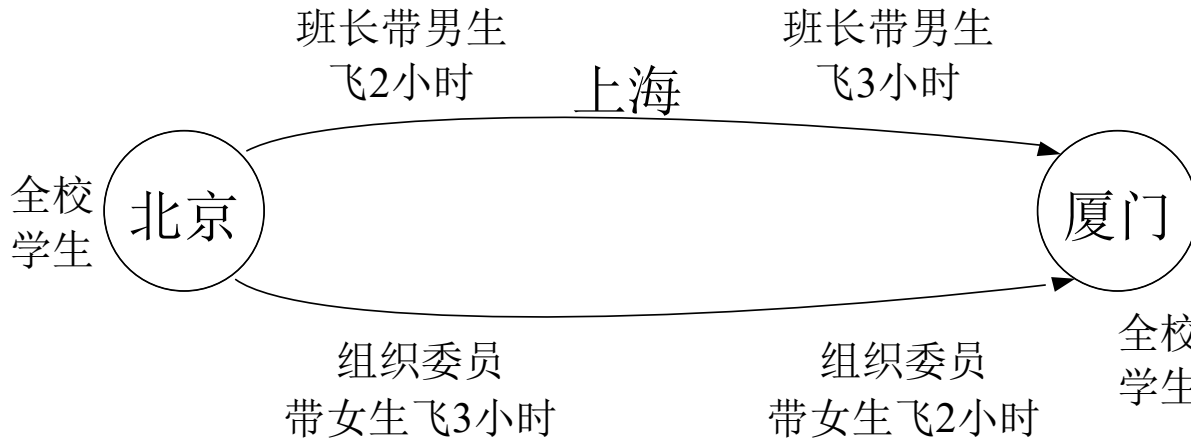
阶段的划分

- 什么是“流水线”优化？(pipelined execution)
- 把一个分区上的多个操作放在一个Task里进行
- 举例：一个学校（含2个班级）完成从北京到厦门的长征



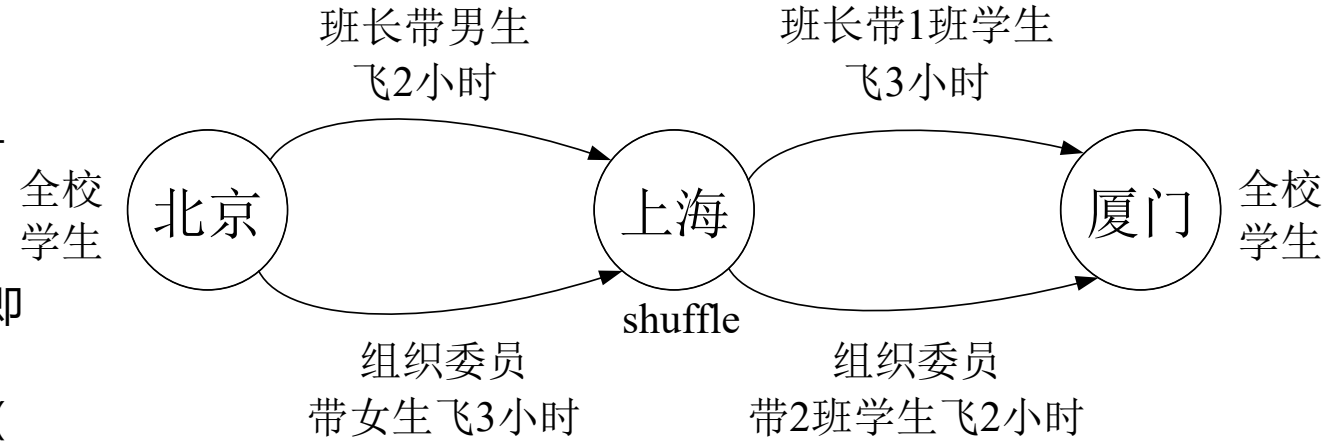
阶段的划分





（a）窄依赖

- a班长带男生从北京-上海和上海-厦门可以放到同一task中，和组织委员带女生从北京-上海-厦门可以放到两个task中并行进行、两个任务都是流水线作业
- b班长带1班学生必须等待男生北京-上海和女生北京-上海完成后再进行，在上海产生等待【人员在不同队伍中的交换（shuffle）】无法流水线作业
- 类比
  - 父RDD：学生（北京-上海）；子RDD：学生（上海-厦门）
- 将男生/女生看作不同分区；1班/2班看作不同分区
- (a) 男生（上海-厦门）对应男生（北京-上海），也即父RDD的一个分区对应子RDD的一个分区
- (b) 1班 (上海-厦门)对应男生（北京-上海）和女生（北京-上海），也即父RDD的一个分区对应子RDD的多个分区

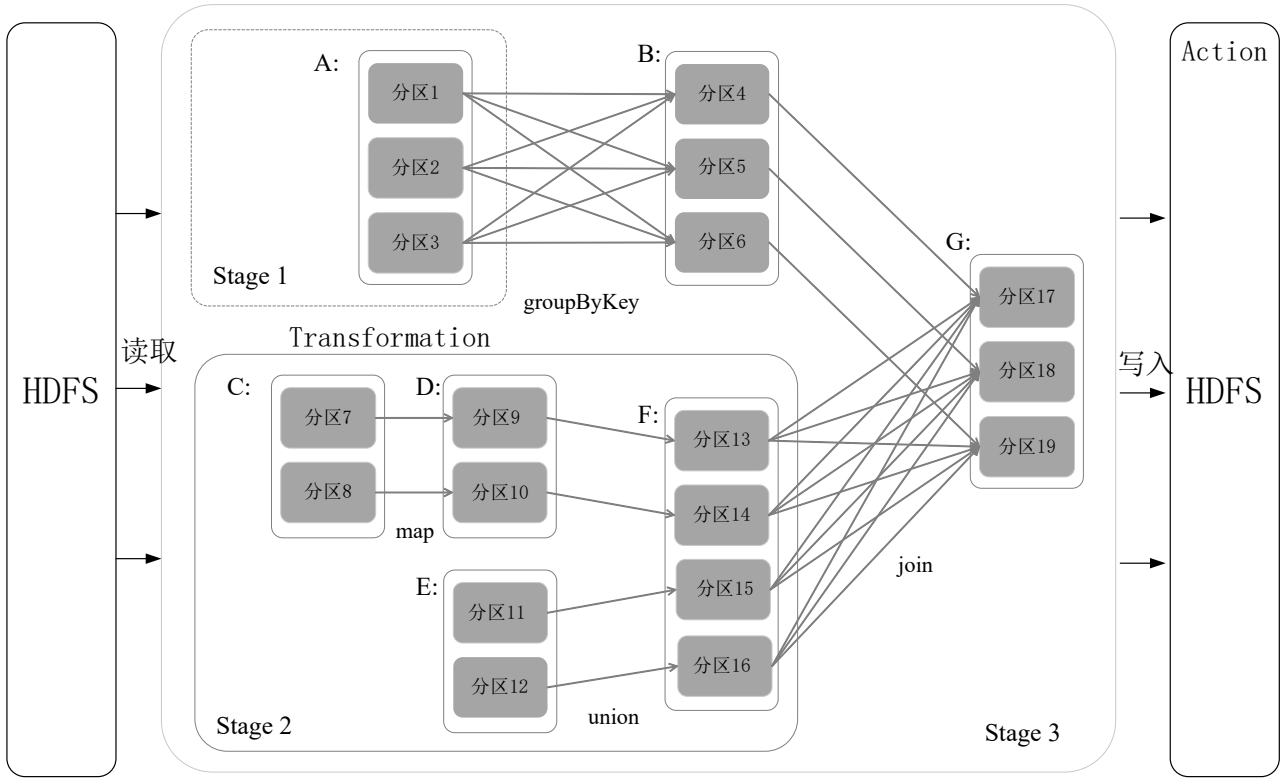


（a）宽依赖

- 窄依赖可以实现“流水线”优化==同一分区的多个操作放在同一task
  - =>没有数据在多个任务task或节点executor之间的分发
  - =>无shuffle
- 宽依赖无法实现“流水线”优化==父RDD分区的操作结果要给子RDD的多个分区使用
  - 由于一个分区对应一个task，父RDD分区对应的task和子RDD的多个分区对应的多个task必然无法合并
  - =>存在数据在多个task之间、以及多个节点executor之间的分发
  - =>有shuffle

阶段的划分

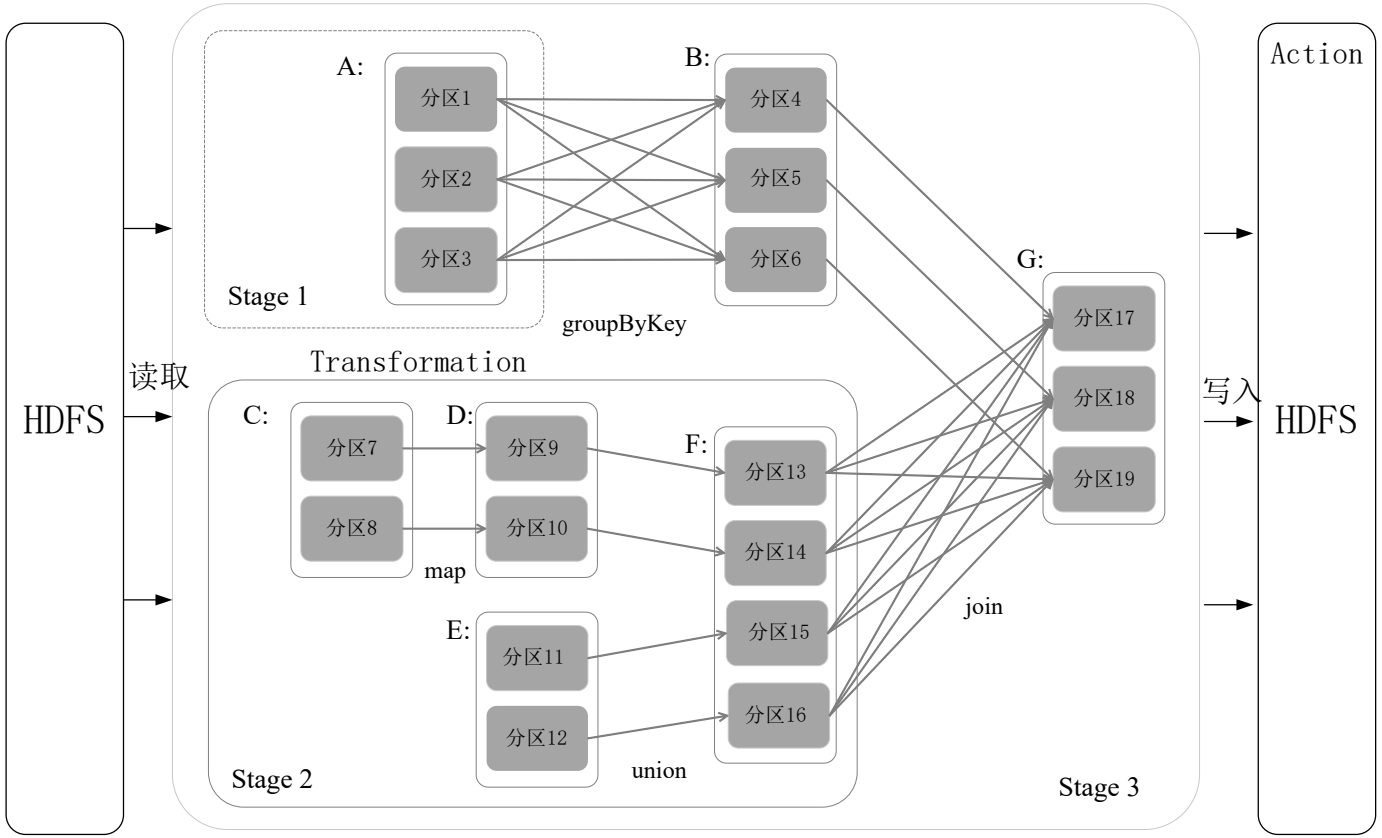
- Spark 根据DAG 图中的RDD 依赖关系，把一个作业分成多个阶段。阶段划分的依据是窄依赖和宽依赖。具体划分方法是：
  - 在DAG中进行反向解析，遇到宽依赖就断开
  - 遇到窄依赖就把当前的RDD加入到Stage中
  - 将窄依赖尽量划分在同一个Stage中，可以实现流水线计算



阶段的划分

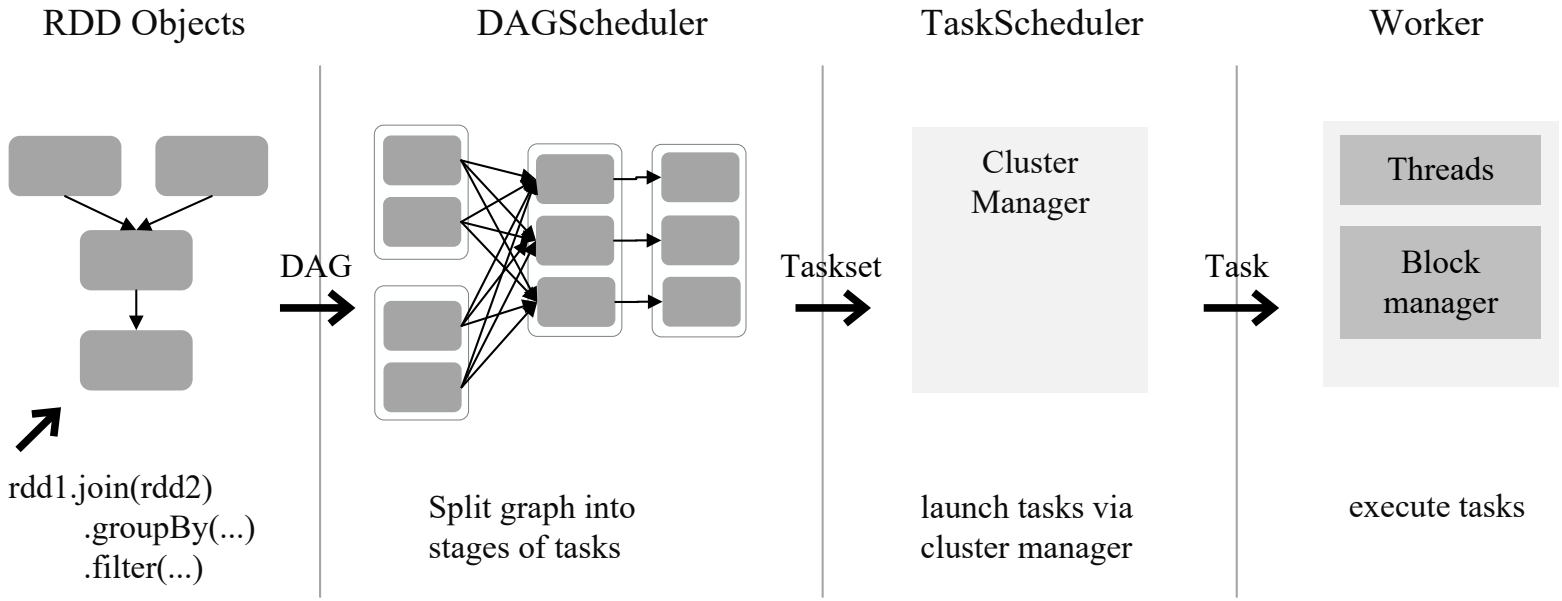
- 被分成三个Stage，在Stage2中，从map到union都是窄依赖，这两步操作可以形成一个流水线操作

- **流水线操作实例**
- 分区7通过map操作生成的分区9，可以不用等待分区8到分区10这个map操作的计算结束，而是继续进行union操作，得到分区13，这样流水线执行大大提高了计算的效率



RDD运行过程

- 通过上述对RDD概念、依赖关系和Stage划分的介绍，结合之前介绍的Spark运行基本流程，再总结一下RDD在Spark架构中的运行过程：
- （1）创建RDD对象；
- （2）SparkContext负责计算RDD之间的依赖关系，构建DAG；
- （3）DAGScheduler负责把DAG图分解成多个Stage，每个Stage中包含了多个Task，每个Task会被TaskScheduler分发给各个WorkerNode上的Executor去执行。



课堂习题

- Spark 根据DAG 图中的RDD 依赖关系，把一个作业分成多个阶段。阶段划分的依据说法正确的是：
- A. 阶段划分的依据是窄依赖和宽依赖，遇到宽依赖就断开，遇到窄依赖就把当前的RDD加入到Stage中
- B. 阶段划分的依据是RDD的操作类型（Action行动和Transformation转换）,行动和转换在不同的阶段
- C. 在DAG中将窄依赖尽量划分在同一个Stage中，可以实现流水线计算
- D. 每个阶段的任务集是由相互之间没有Shuffle依赖关系的任务组成的
- E. 窄依赖无法实现“流水线”优化，宽依赖可以实现“流水线”优化

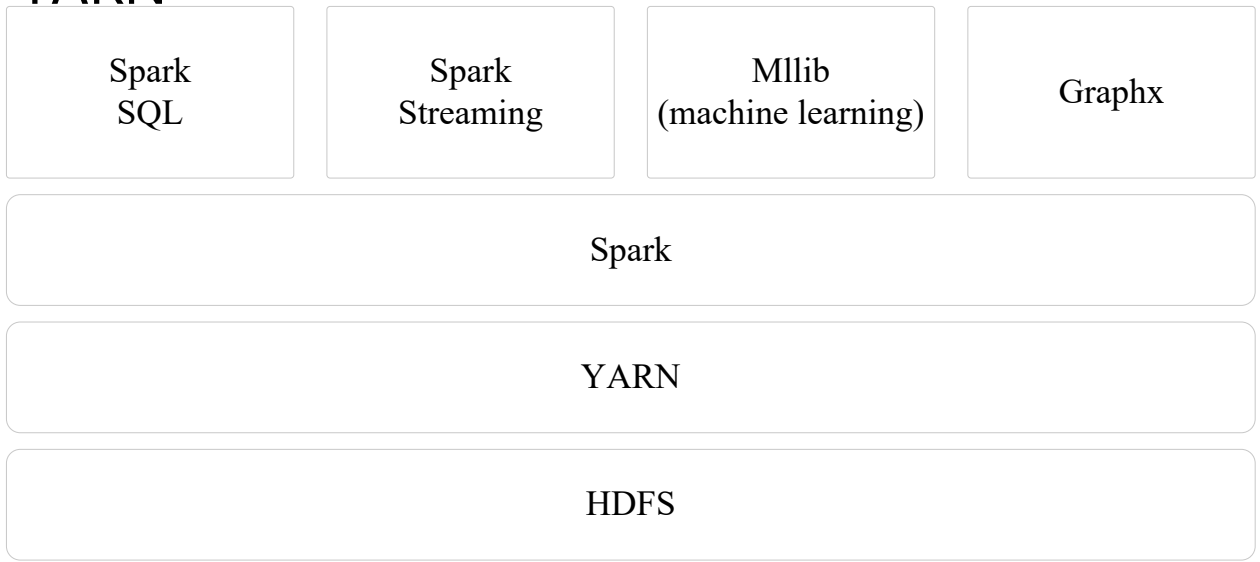
4

PART FOUR

Spark部署方式

Spark的部署方式

- Spark支持四种不同类型的部署方式，包括：
  - Local： 单机模式
  - Standalone：使用Spark自带的简单集群管理器
  - Spark on Mesos（和Spark有血缘关系，更好支持Mesos）
  - Spark on YARN



Spark on YARN架构

讨论：Spark和Hadoop

- 虽然Spark很快，但现在在生产环境中仍然不尽人意，无论扩展性、稳定性、管理性等方面都需要进一步增强
- 同时，Spark在流处理领域能力有限，如果要实现亚秒级或大容量的数据获取或处理需要其他流处理产品。Cloudera宣布旨在让Spark流数据技术适用于80%的使用场合，就考虑到了这一缺陷。我们确实看到实时分析（而非简单数据过滤或分发）场景中，很多以前使用S4或Storm等流式处理引擎的实现已经逐渐被Kafka+Spark Streaming代替
- Spark的流行将逐渐让MapReduce、Tez走进博物馆
- Hadoop现在分三块HDFS/MR/YARN，Spark比Hadoop性能好，只是Spark作为一个计算引擎，比MR的性能要好。但它的存储和调度框架还是依赖于HDFS/YARN，Spark也有自己的调度框架，但仍然非常不成熟，基本不可商用

Thank you