



哈爾濱工業大學 海量数据计算研究中心

Massive Data Computing Lab @ HIT

大数据计算基础

第四章 大数据计算系统

王宏志

wangzh@hit.edu.cn

<http://homepage.hit.edu.cn/pages/wang>

- 1 大数据计算系统概述**
- 2 大数据计算框架概述**
- 3 大数据批处理计算框架**
- 4 大数据实时计算框架**
- 5 大图计算框架**
- 6 大数据存储**
- 7 大数据计算的硬件平台**

- 1 大数据计算系统概述
- 2 大数据计算框架概述
- 3 大数据批处理计算框架**
- 4 大数据实时计算框架
- 5 大图计算框架
- 6 大数据存储
- 7 大数据计算的硬件平台

什么是大数据的批处理？

- 批处理操作大容量静态数据集，并在计算过程完成后返回结果。
- 批处理模式中使用的数据集通常符合下列特征：
 - ✓ 有界：批处理数据集代表数据的有限集合。
 - ✓ 持久：数据通常始终存储在某种类型的持久存储位置中。
 - ✓ 大量：批处理操作通常是处理极为海量数据集的唯一方法。
- 批处理适合
 - 需要访问全套记录才能完成的计算工作
 - 对历史数据进行分析
- 批处理不适合
 - 对处理时间要求较高的场合

| | | |
|----|---------------|-------|
| 设计 | 工艺优化、 能效优化 | 流程优化、 |
| 供销 | 成本优化 | |
| 销售 | 需求发现、 | 产量预测 |
| 售后 | 备品供应 | |

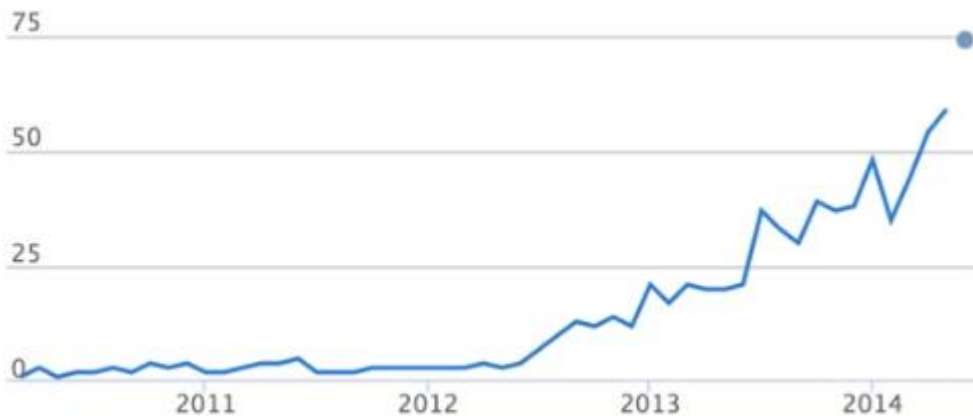
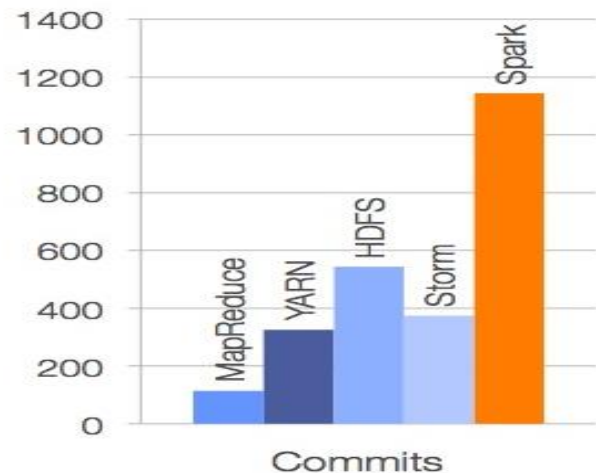
| | | |
|----|-----------------|-------|
| 生产 | 质量监控、 生产状态监测 | 故障预测、 |
| 售后 | 产品运行状态监控、 | 故障预警 |



Spark的发展历程

- 2009: Spark诞生于伯克利大学 AMPLab
- 2010: 开源
- 2013.6: Apache孵化器项目
- 2014.2: Apache顶级项目
- 目前为止, 发布的最新版本为Spark 3.0.1

Spark在最近10年内发展迅速, 相较于其他大数据平台或框架而言, Spark的代码库最为活跃。



Spark代码贡献者每个月的增长曲线

截止2015年6月

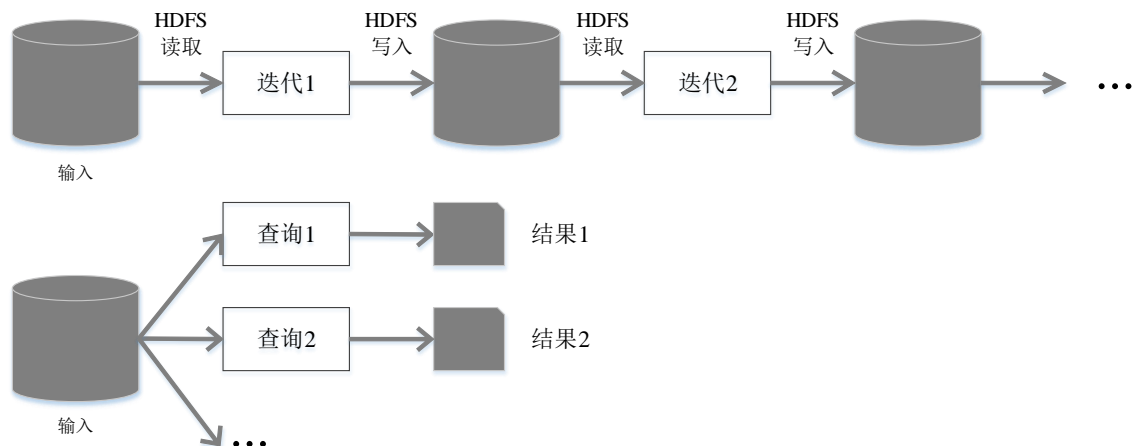
- Spark的Contributor比2014年涨了3倍, 达到730人;
- 总代码行数也比2014年涨了2倍多, 达到40万行
- Spark应用也越来越广泛, 最大的集群来自腾讯——8000个节点, 单个Job最大分别是阿里巴巴和Databricks——1PB

Spark的目标

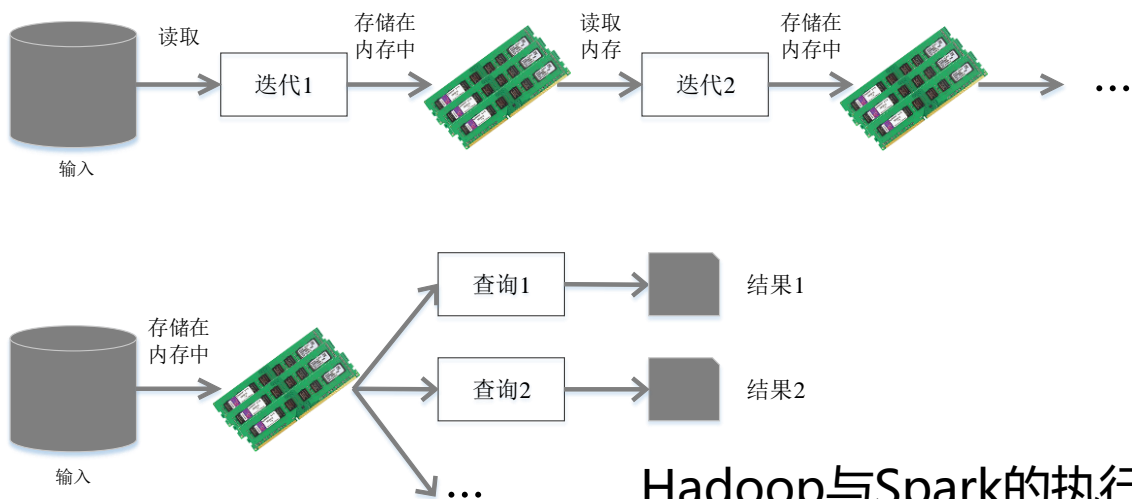
- 为集群运算提供分布式内存抽象，以支持具有工作集的应用程序
- 保留了MapReduce的优秀特性：
 - 容错机制（为崩溃程序和落后程序）
 - 数据局部性
 - 可扩展性

解决方案： 使用 “弹性分布式数据集”
(RDDs) 的增强数据流模型

Spark与Hadoop的对比



(a) Hadoop MapReduce执行流程



(b) Spark执行流程

Hadoop与Spark的执行流程对比图

Hyracks

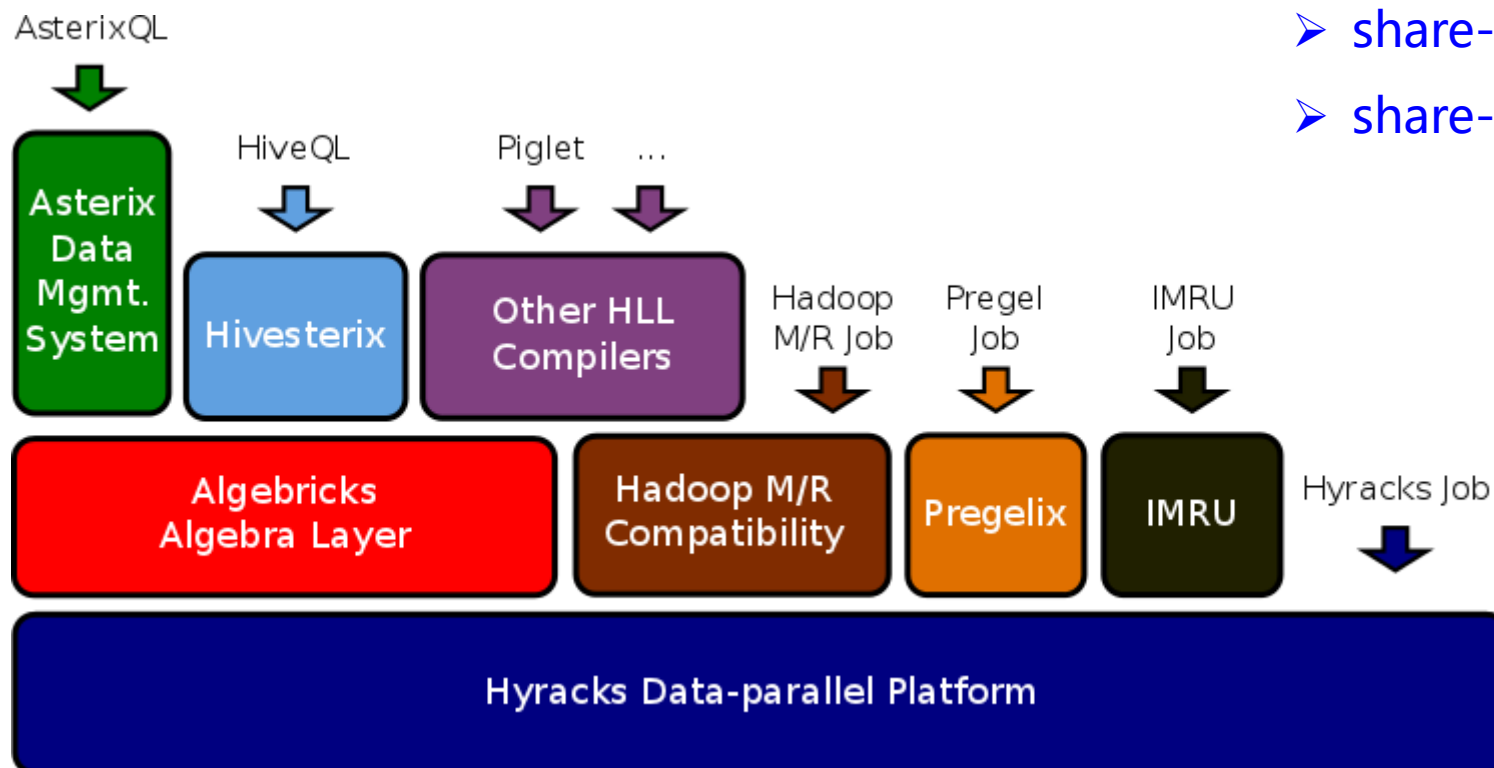


Hyracks简介

- UCI开发, 2013年发布
- 并行数据计算平台, 用于在使用share-nothing的大型集群对大量数据执行数据处理任务

概念区分

- share-nothing
- share-memory



Hyracks开发的原因

- 数据总量成倍增长
 - 社交网络 (Facebook: 5亿活跃用户, 每天超过5000万状态更新)
 - 网络: 几万亿的连接
- 数据类型并不总是统一的
 - 半结构化数据
 - 结构化数据
 - 不同的数据模型 (如数组等)
- 非传统的数据处理
 - 需要使用用户的逻辑来处理数据

- VS. 并行数据库
 - 可扩展的数据模型
 - 容错机制
- VS. Hadoop
 - 更灵活的计算模型
 - 通用的运行时处理模型
 - 更好（透明）的支持调度

需要解决的问题

- 操作的数据形式和结构是什么？
- 应用程序可以对于数据做何种操作？

需要考虑的问题

- 适用的场景/数据是什么？
- 面向的软硬件环境是什么？
- 和支撑应用程序之间的界面在哪里？

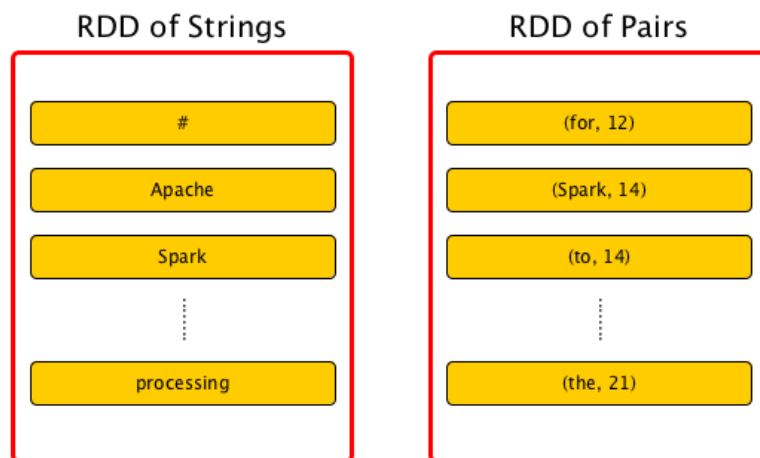


RDD设计背景

- 许多迭代式（比如机器学习、图算法等）和交互式数据挖掘工具的共同之处是，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提供了一个抽象的数据架构，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，可以实现数据流水处理，避免中间数据存储

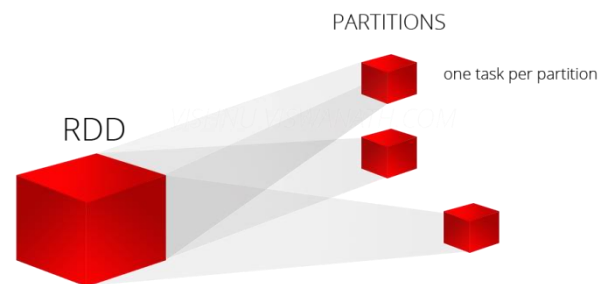
概念区分

- 流水 (pipeline)
- 阻塞 (blocked)



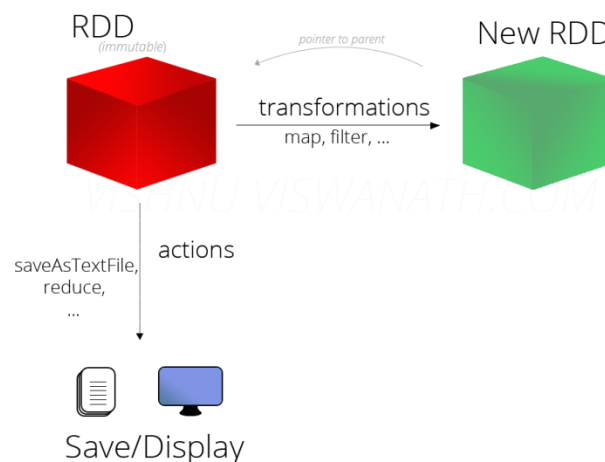
RDD概念

- 一个RDD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
- 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
 - ✓ 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的操作

- 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) 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）
- RDD已经被实践证明可以高效地表达许多框架的编程模型（如MapReduce、SQL、Pregel）
- Spark用Scala语言实现了RDD的API，程序员可以通过调用API实现对RDD的各种操作

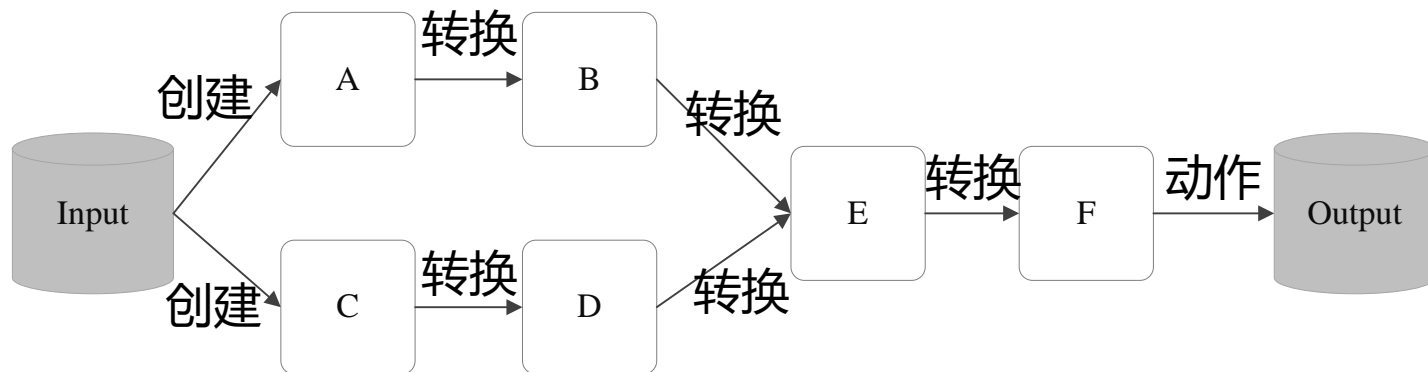


RDD的执行过程

- 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

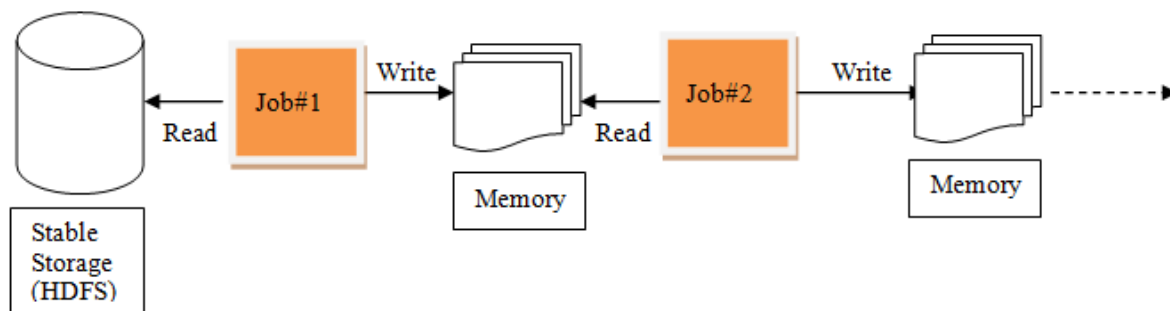
这一系列处理称为一个Lineage（血缘关系），即DAG拓扑排序的结果

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



RDD高效的原因

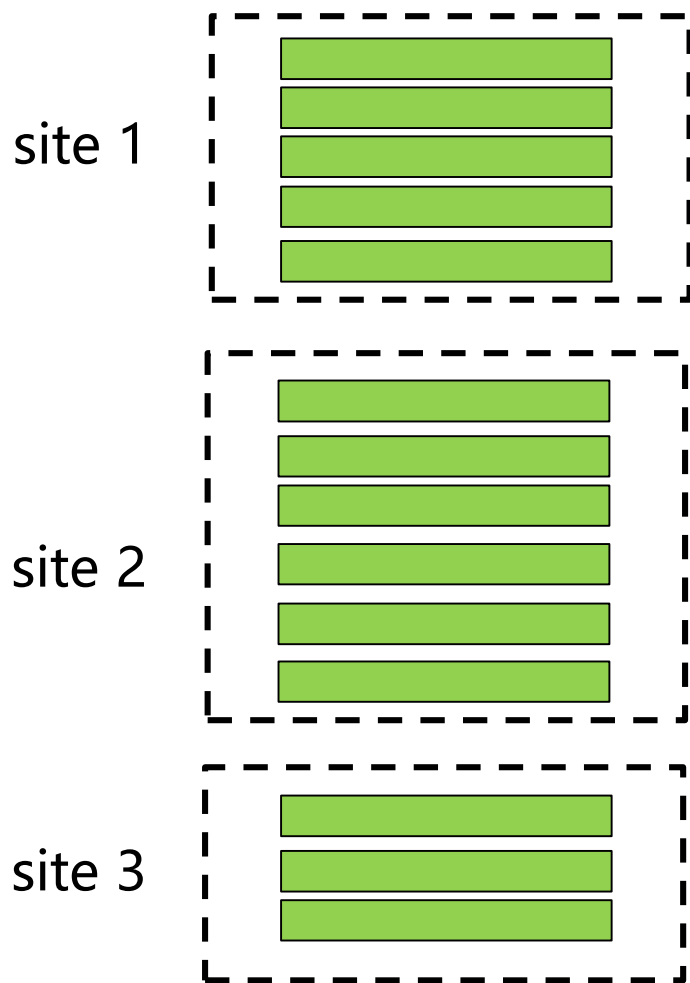
- 高效的容错性
 - ✓ 现有容错机制：数据复制或者记录日志
 - ✓ RDD：血缘关系、重新计算丢失分区、无需回滚系统、重算过程在不同节点之间并行、只记录粗粒度的操作
- 中间结果持久化到内存，数据在内存中的多个RDD操作之间进行传递，避免了不必要的读写磁盘开销
- 存放的数据可以是Java对象，避免了不必要的对象序列化和反序列化



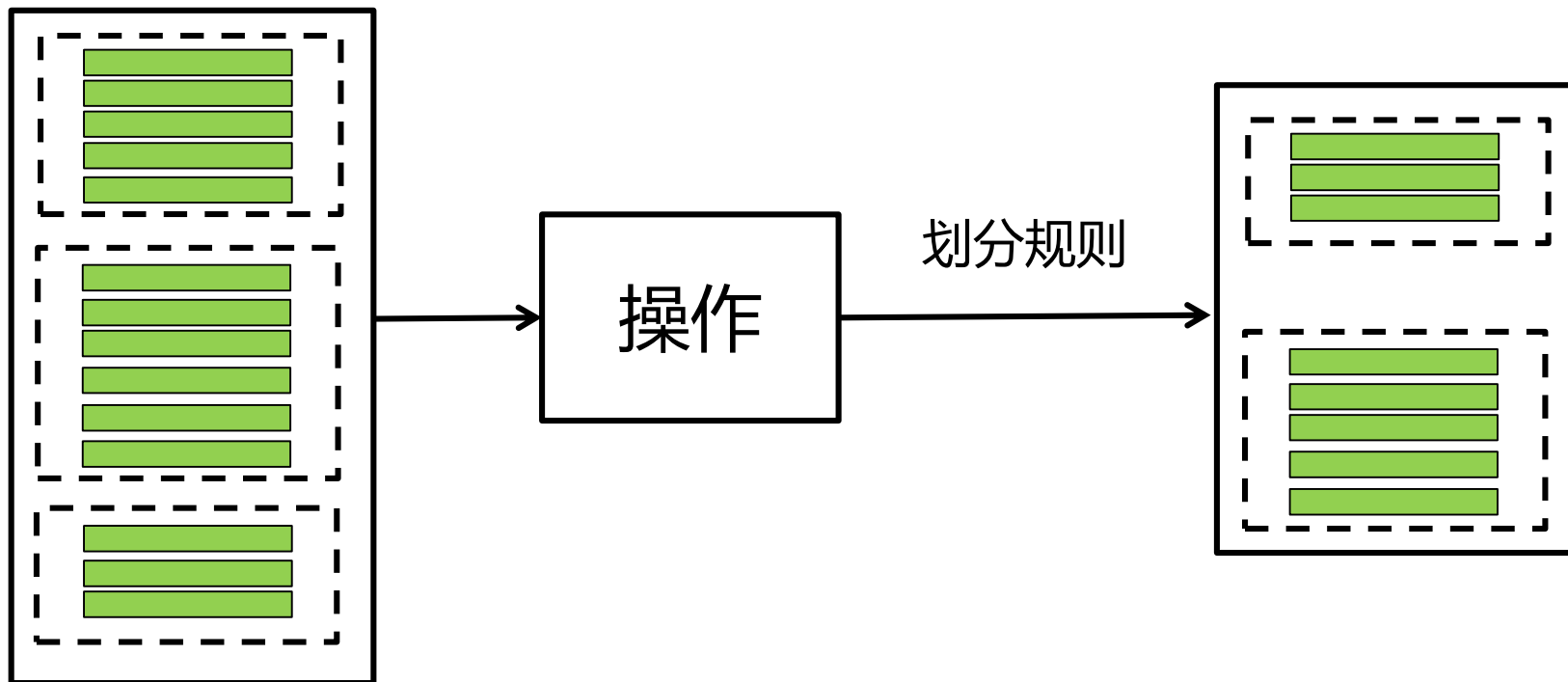
Hyracks



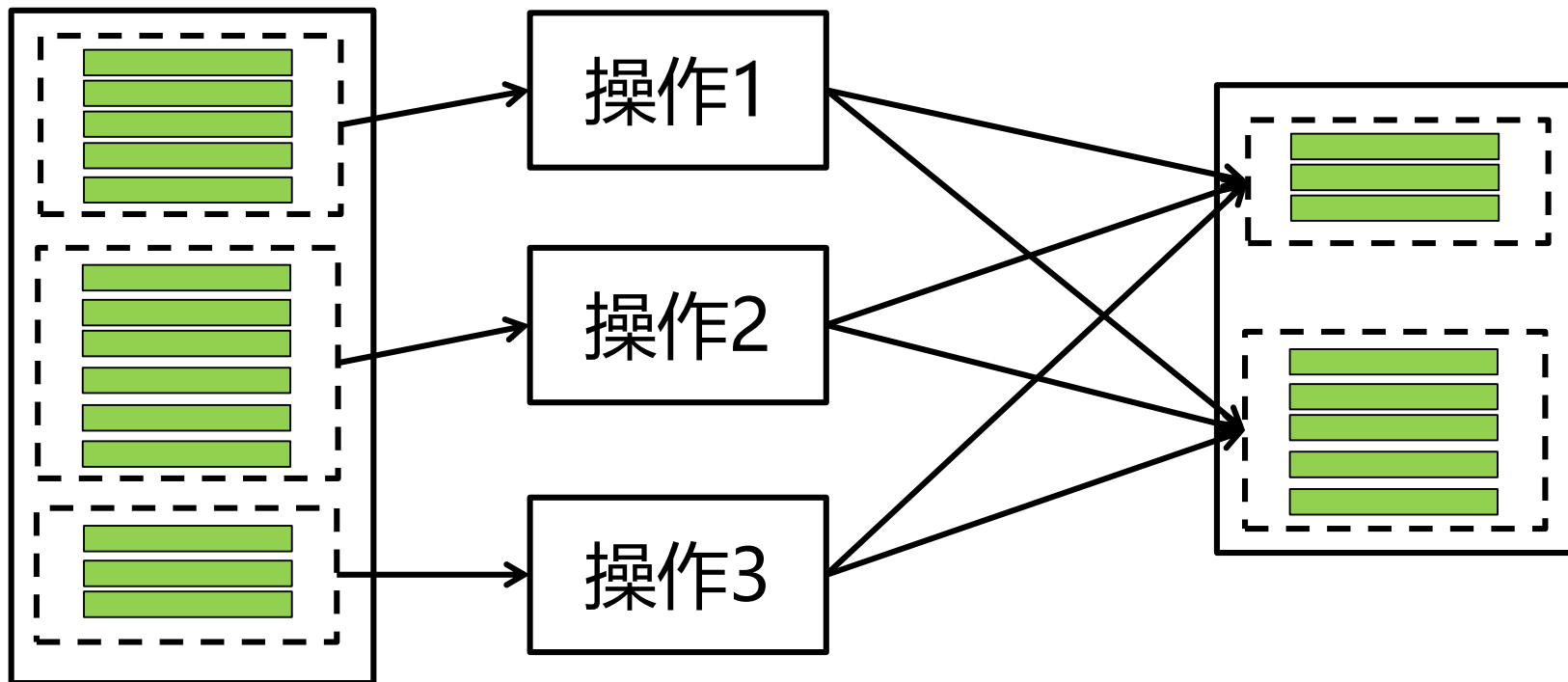
Hyracks 的数据对象



- 在N个站点分区的记录
- 存储有模式的记录（不仅仅是key-value)



操作的并行处理



需要解决的问题

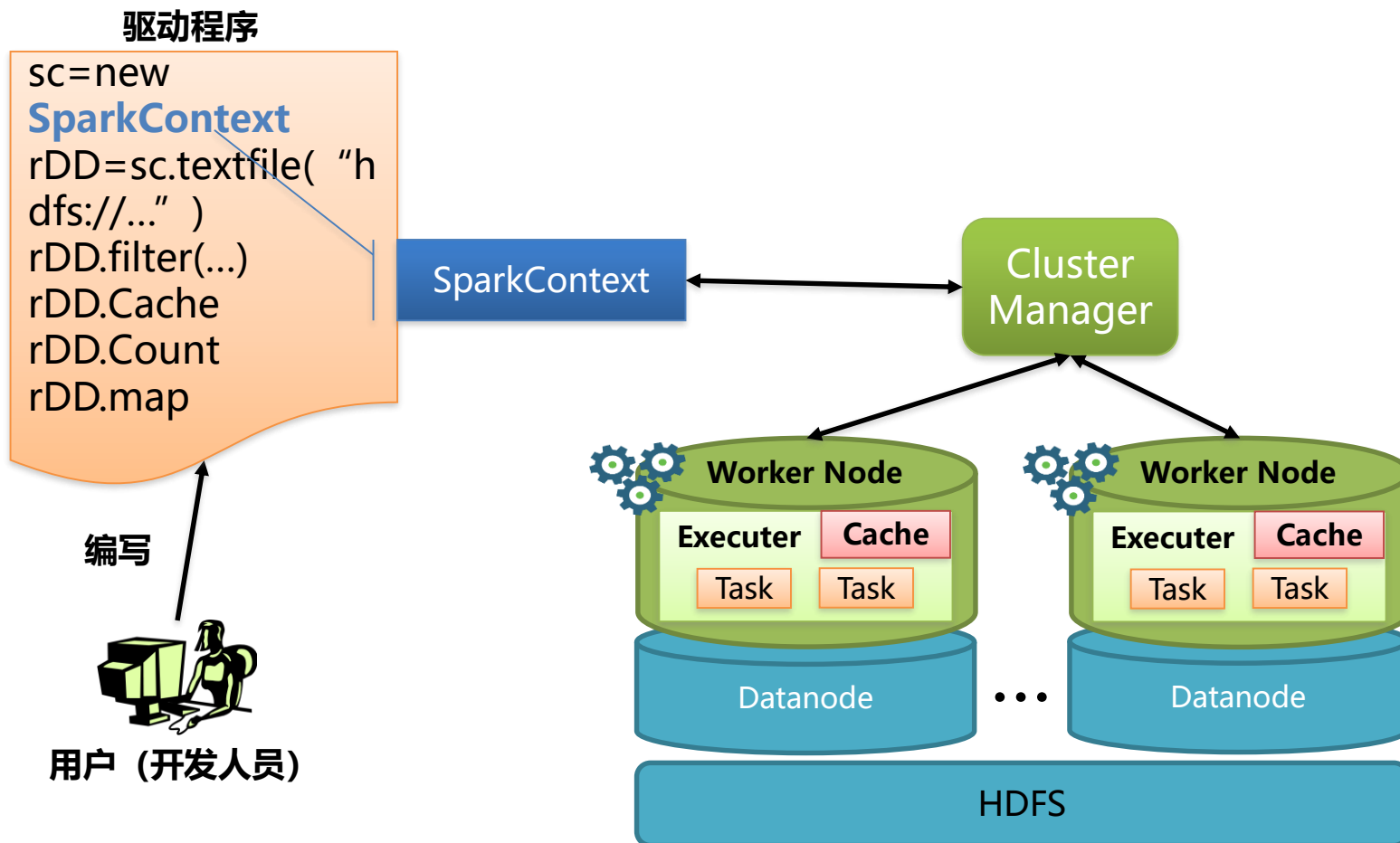
- 程序员如何基于框架编写应用程序？

需要考虑的问题

- 面向什么样的程序员？
- 开发的效率？
- 程序的易读性？
- 程序的美感？
- 是否符合传统开发的习惯？



Spark编程模型



Spark编程模型

驱动程序

```
sc=new  
SparkContext  
rDD=sc.textfile( "  
hdfs://..." )  
rDD.filter(...)  
rDD.Cache  
rDD.Count  
rDD.map
```

RDD
(Resilient
Distributed
Dataset)

- 不可变的数据结构
- 存储在内存中（显式）
- 容错
- 并行数据结构
- 可控的分区以优化数据存储
- 可以使用多种的运算符操作

编写



用户（开发人员）

操作RDD的API

- 编程接口：程序员可以执行3种类型的操作

Transformations

- 从现有的数据集中创建新的数据集.
- 惰性计算。它们仅在action执行时才执行。
- 例子：
 - Map(func)
 - Filter(func)
 - Distinct()

Actions

- 返回一个数据到驱动程序或者计算之后向一个存储系统输出数据
- 例子：
 - Count()
 - Reduce(func)
 - Collect
 - Take()

Persistence

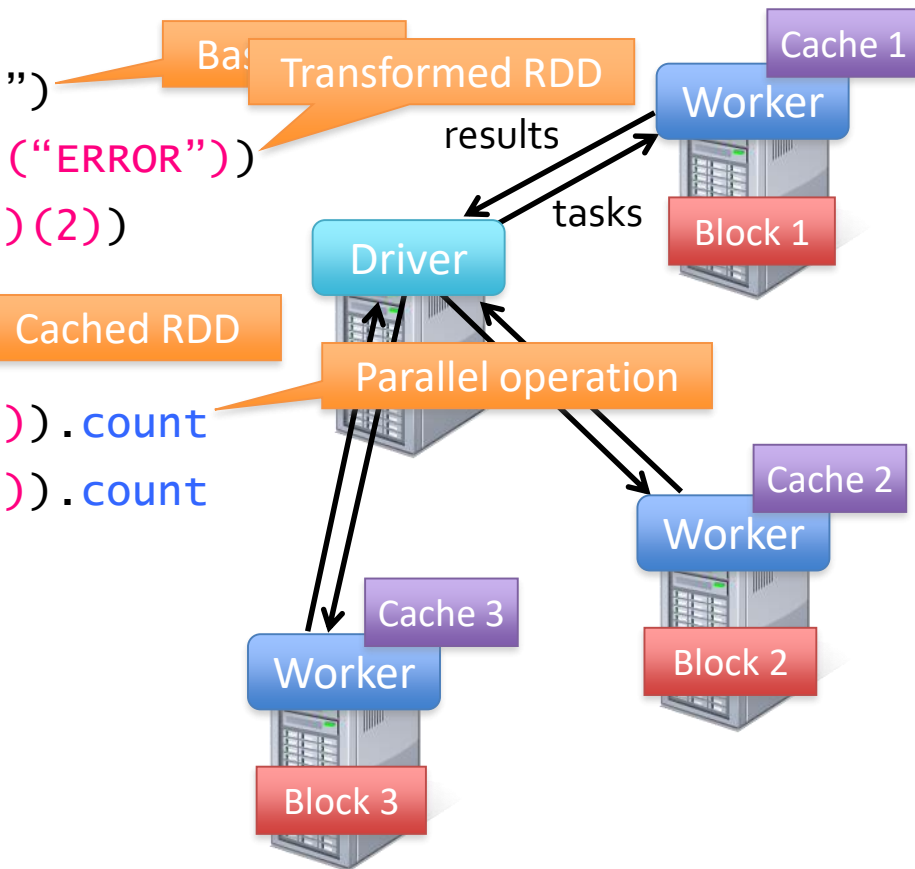
- 为后续操作在内存中缓存数据集
- 可选存在磁盘上或者RAM上, 或混合存储 (存储级别)
- 例子：
 - Persist()
 - Cache()

例1：日志挖掘

- 从日志中加载错误信息到内存，接着交互式地搜索不同的模式

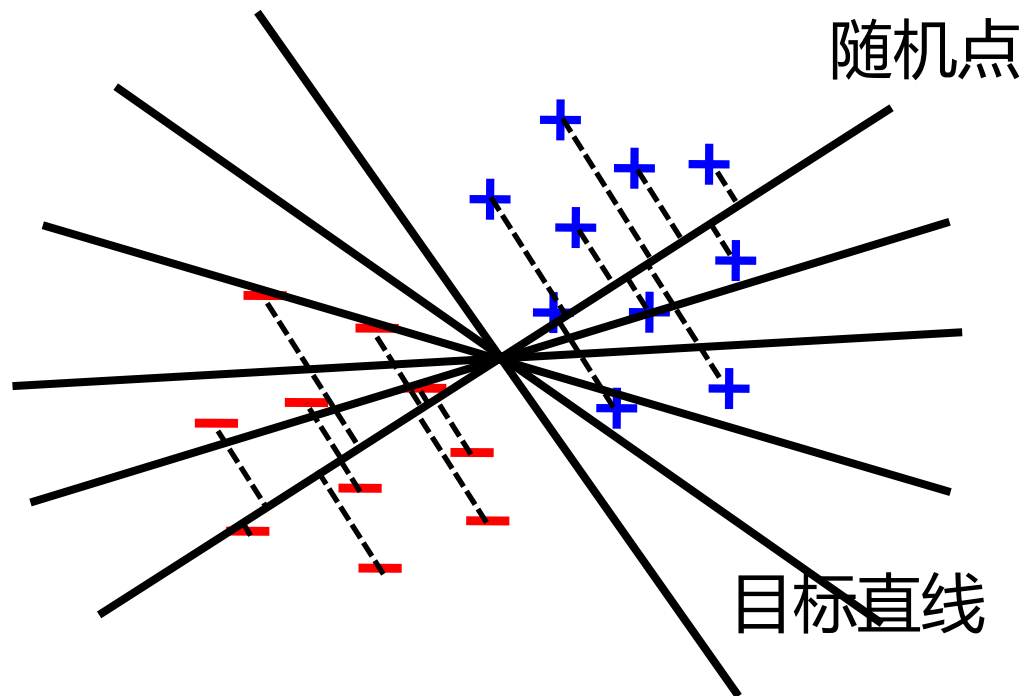
```
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
messages = errors.map(_.split('\t')(2))
cachedMsgs = messages.cache()

cachedMsgs.filter(_.contains("foo")).count
cachedMsgs.filter(_.contains("bar")).count
. . .
```



例2：逻辑回归

- 目的：找到最好的分割两组点的直线



逻辑回归代码

- `val data = spark.textFile(...).map(readPoint).cache()`
- `var w = Vector.random(D)`
- `for (i <- 1 to ITERATIONS) {`
- `val gradient = data.map(p =>`
 - `(1 / (1 + exp(-p.y*(w dot p.x))) - 1) * p.y * p.x`
 - `).reduce(_ + _)`
 - `w -= gradient`
- `}`
- `println("Final w: " + w)`

例3: MapReduce

MapReduce可以用RDD转换来表示

```
res = data.flatMap(rec => myMapFunc(rec))  
          .groupByKey()  
          .map((key, vals) => myReduceFunc(key, vals))
```

或者使用Combiner

```
res = data.flatMap(rec => myMapFunc(rec))  
          .reduceByKey(myCombiner)  
          .map((key, val) => myReduceFunc(key, val))
```


例4：字数统计

```
1 public class WordCount {
2     public static class TokenizerMapper
3         extends Mapper<Object, Text, Text, IntWritable>{
4
5         private final static IntWritable one = new IntWritable(1);
6         private Text word = new Text();
7
8         public void map(Object key, Text value, Context context
9             ) throws IOException, InterruptedException {
10             StringTokenizer itr = new StringTokenizer(value.toString());
11             while (itr.hasMoreTokens()) {
12                 word.set(itr.nextToken());
13                 context.write(word, one);
14             }
15         }
16     }
17
18     public static class IntSumReducer
19         extends Reducer<Text, IntWritable, Text, IntWritable> {
20         private IntWritable result = new IntWritable();
21
22         public void reduce(Text key, Iterable<IntWritable> values,
23             Context context
24             ) throws IOException, InterruptedException {
25             int sum = 0;
26             for (IntWritable val : values) {
27                 sum += val.get();
28             }
29             result.set(sum);
30             context.write(key, result);
31         }
32     }
33
34     public static void main(String[] args) throws Exception {
35         Configuration conf = new Configuration();
36         String[] otherArgs = new GenericOptionsParser(conf, args).getRemainingArgs();
37         if (otherArgs.length < 2) {
38             System.err.println("Usage: wordcount <in> <in>... <out>");
39             System.exit(2);
40         }
41         Job job = new Job(conf, "word count");
42         job.setJarByClass(WordCount.class);
43         job.setMapperClass(TokenizerMapper.class);
44         job.setCombinerClass(IntSumReducer.class);
45         job.setReducerClass(IntSumReducer.class);
46         job.setOutputKeyClass(Text.class);
47         job.setOutputValueClass(IntWritable.class);
48         for (int i = 0; i < otherArgs.length - 1; ++i) {
49             FileInputFormat.addInputPath(job, new Path(otherArgs[i]));
50         }
51         FileOutputFormat.setOutputPath(job,
52             new Path(otherArgs[otherArgs.length - 1]));
53         System.exit(job.waitForCompletion(true) ? 0 : 1);
54     }
55 }
```

```
1 val f = sc.textFile(inputPath)
2 val w = f.flatMap(l => l.split(" ")).map(word => (word, 1)).cache()
3 w.reduceByKey(_ + _).saveAsText(outputPath)
```

WordCount in 3 lines of Spark

WordCount in 50+ lines of Java MR

字数统计的代码

Scala:

```
val f = sc.textFile("README.md")
val wc = f.flatMap(l => l.split(" ")).map(word => (word, 1)).reduceByKey(_ + _)
wc.saveAsTextFile("wc_out")
```

Python:

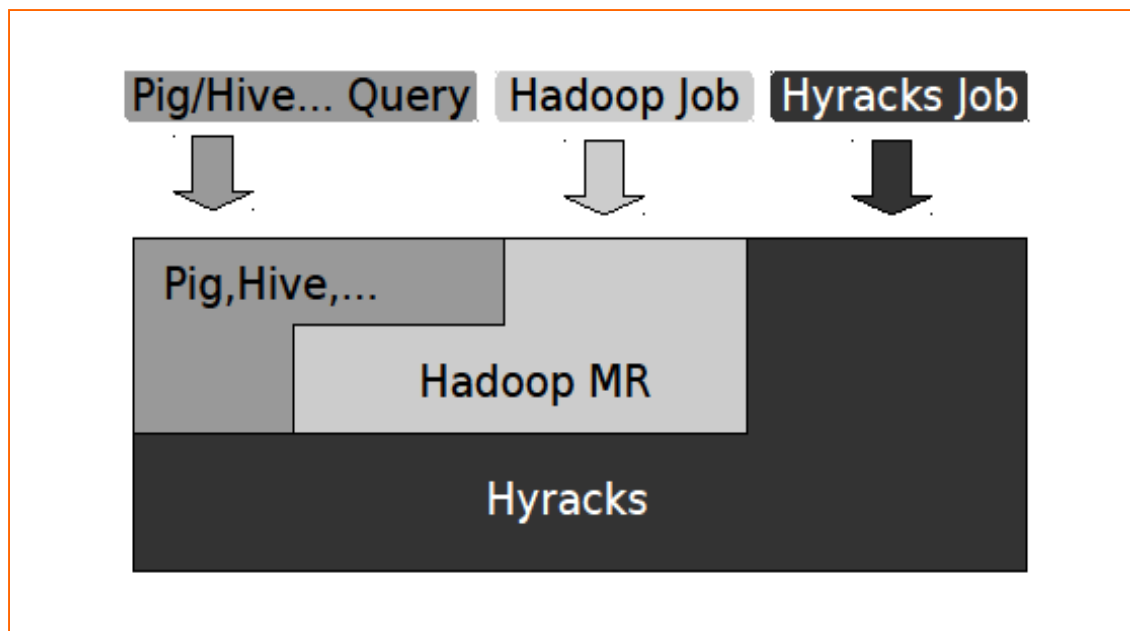
```
from operator import add
f = sc.textFile("README.md")
wc = f.flatMap(lambda x: x.split(' ')).map(lambda x: (x, 1)).reduceByKey(add)
wc.saveAsTextFile("wc_out")
```

Hyracks



Hyracks – 使用场景

- 设计时考虑了三种可能的用途
 - ✓ 以高级数据语言为编译目标
 - 直接编译为原生Hyracks工程
 - 编译Hadoop工程并使用兼容层运行
 - ✓ 从现有的Hadoop客户端运行Hadoop工程
 - ✓ 运行原生的Hyracks工程



- 接受ByteBuffer对象的统一的push-style 迭代器接口

```
public interface IFrameWriter {  
    public void open() ...;  
    public void close() ...;  
    public void nextFrame(ByteBuffer frame)...;  
}
```

- 操作可以被连接在同一个线程或是不同的线程中运行
- 直接调用/改写底层操作

需要解决的问题

- 系统有哪些模块？
- 模块之间如何交互？

需要考虑的问题

- 有效配合硬件
- 扩展性高
- 效率高
- 弹性好
- 开发容易
- 维护方便
- 升级简单

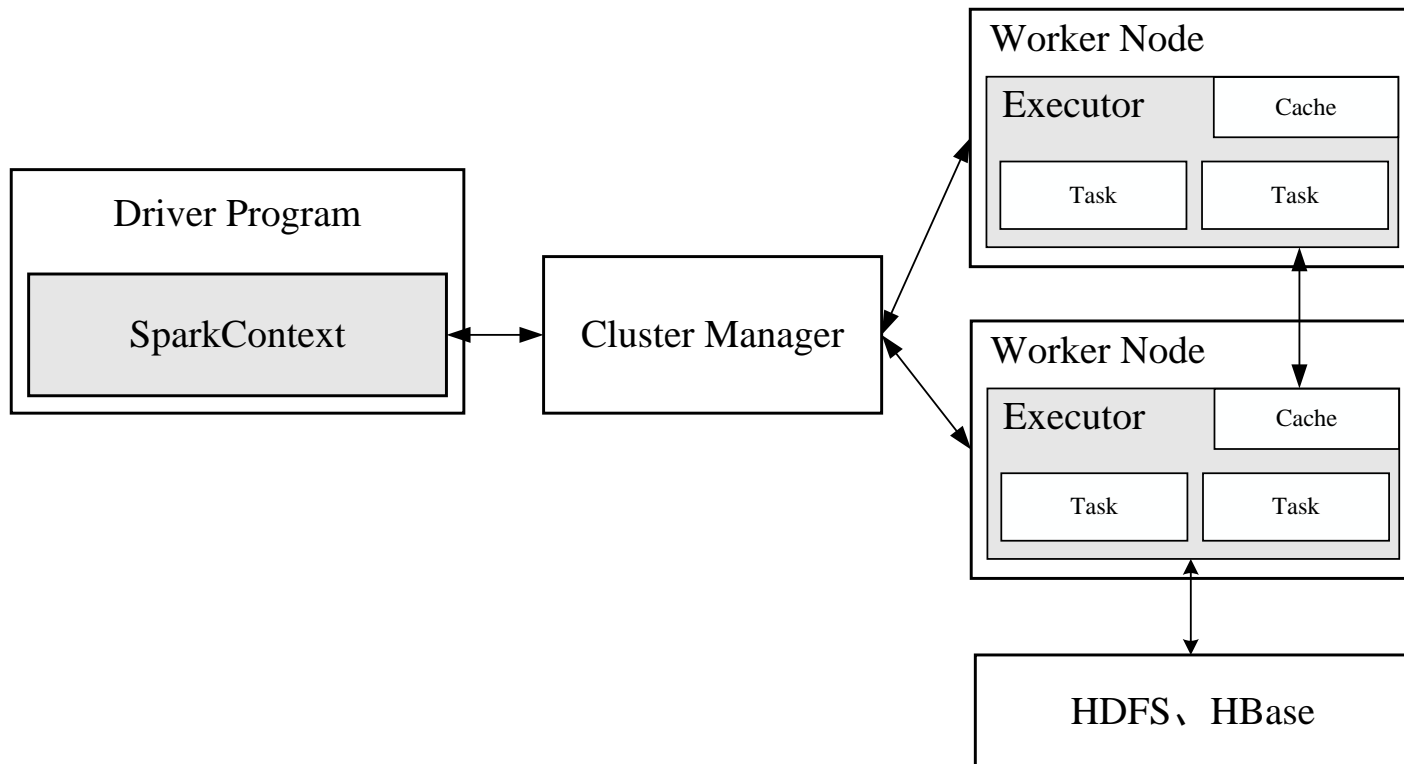


基本概念

- 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，代表了一组关联的、相互之间没有Shuffle依赖关系的任务组成的任务集

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Mesos或YARN

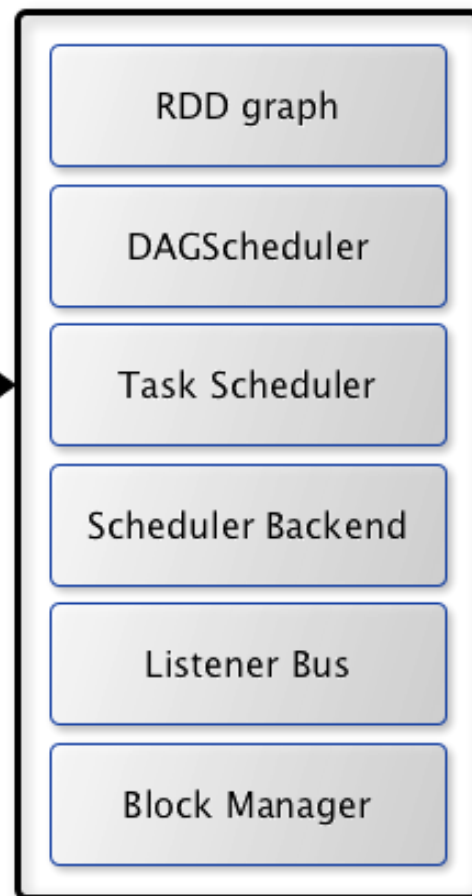


SparkContext

SparkContext处于DriverProgram核心位置，所有与Cluster、Worker Node交互的操作都需要SparkContext来完成

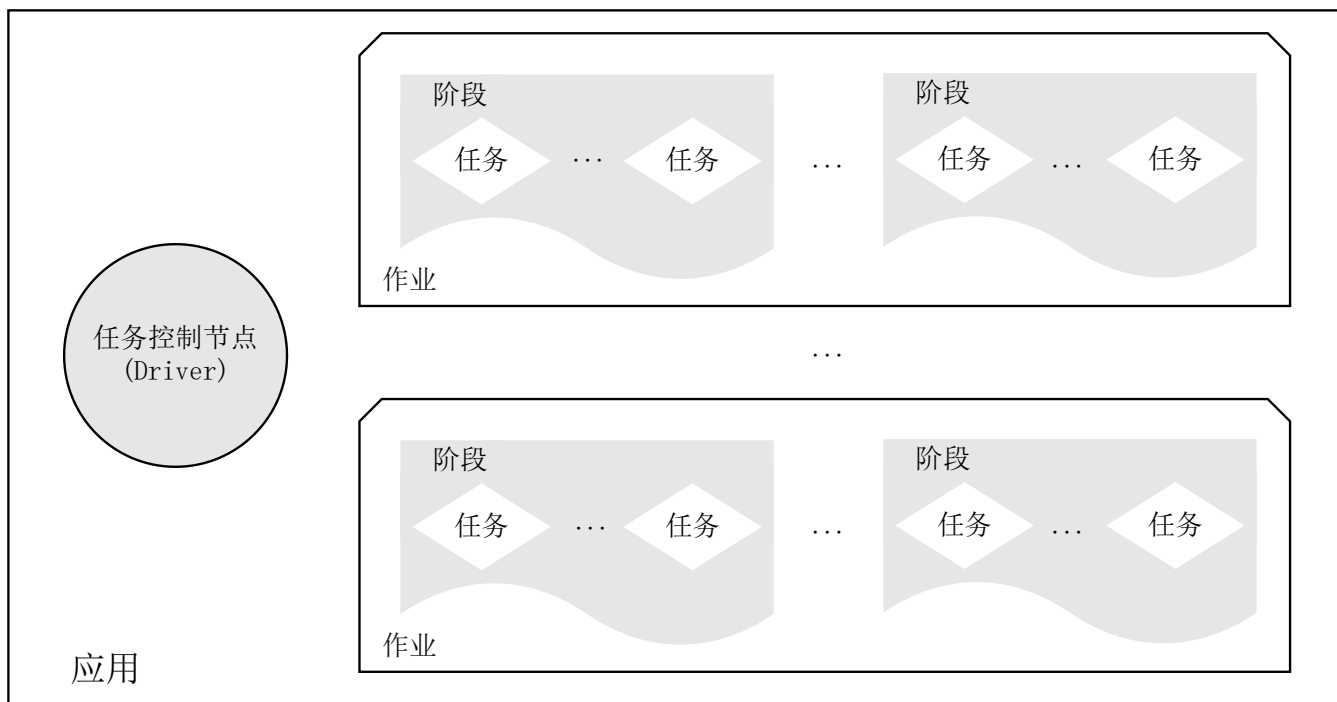
```
val sc = new SparkContext(master="local[*]",  
                           appName="SparkMe App", new SparkConf)  
val lines = sc.textFile(...).cache()  
val c = lines.count()  
println(s"There are $c lines in $fileName")
```

Spark context



架构设计

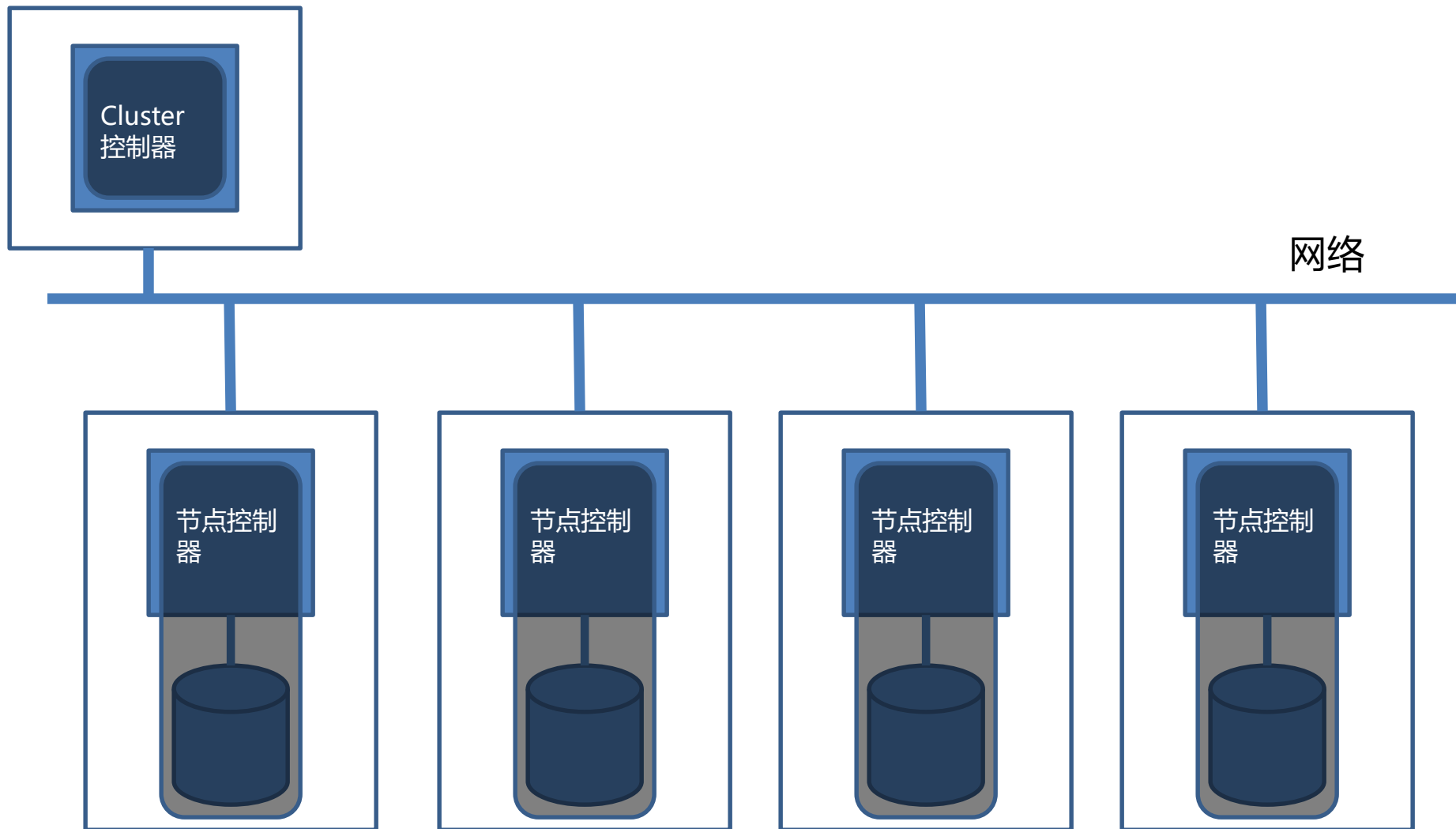
- 一个应用由一个Driver和若干个作业构成，一个作业由多个阶段构成，一个阶段由多个没有Shuffle关系的任务组成
- 当执行一个应用时，Driver会向集群管理器申请资源，启动Executor，并向Executor发送应用程序代码和文件，然后在Executor上执行任务，运行结束后，执行结果会返回给Driver，或者写到HDFS或者其他数据库中



Hyracks



Hyracks的结构



需要解决的问题

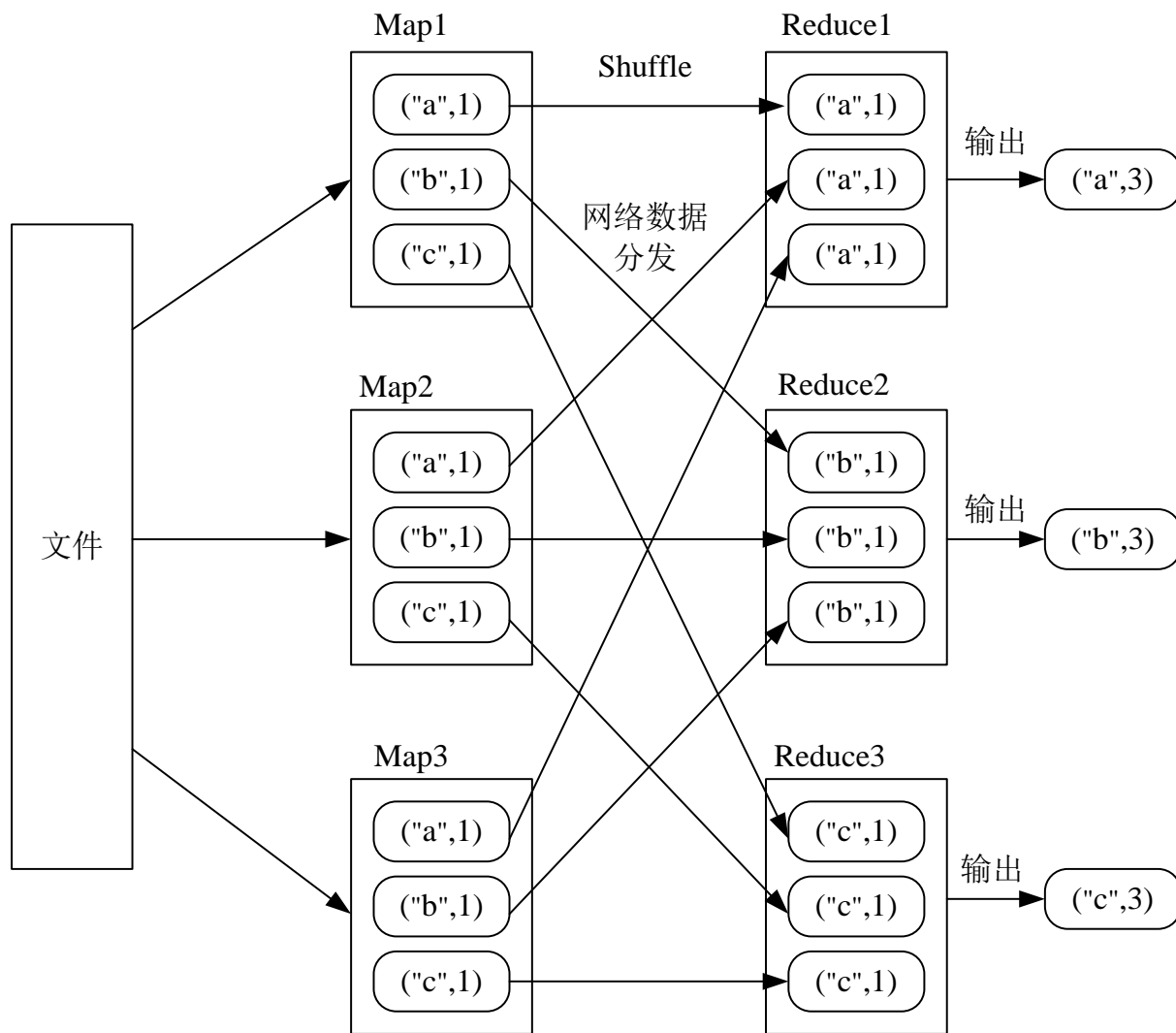
- 包括哪些基本数据操作？
- 并行还是串行实现？
- 高效实现算法？

需要考虑的问题

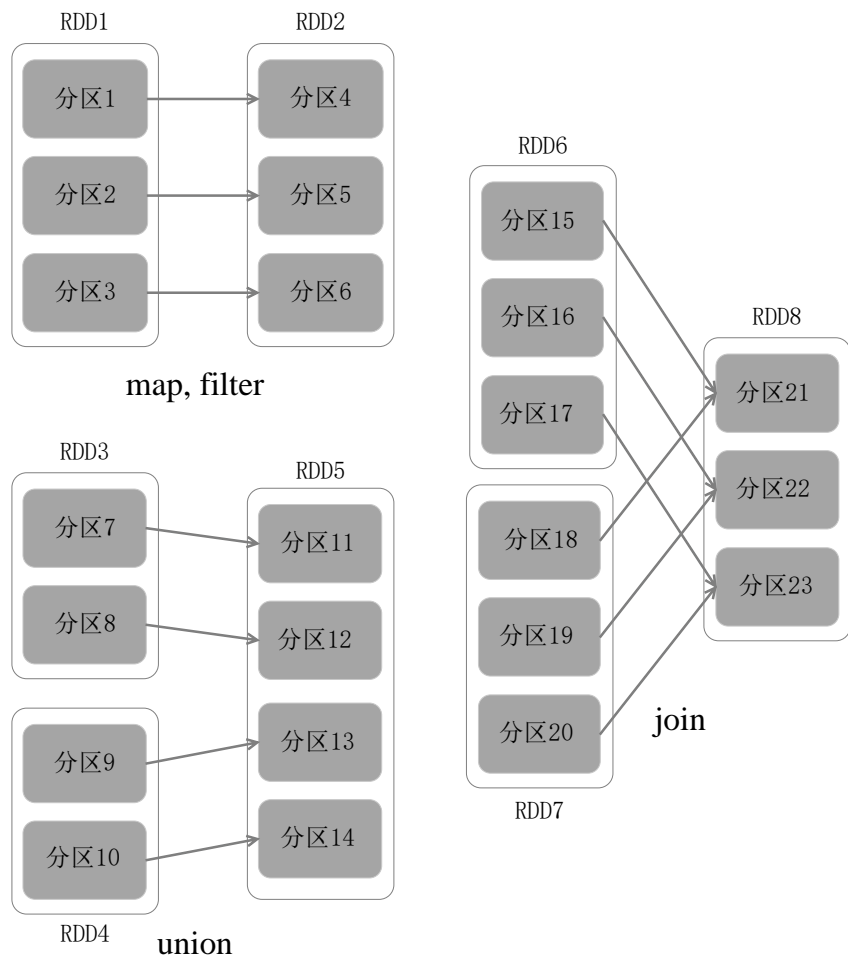
- 有效支撑API
- 容易理解
- 扩展性高
- 效率高



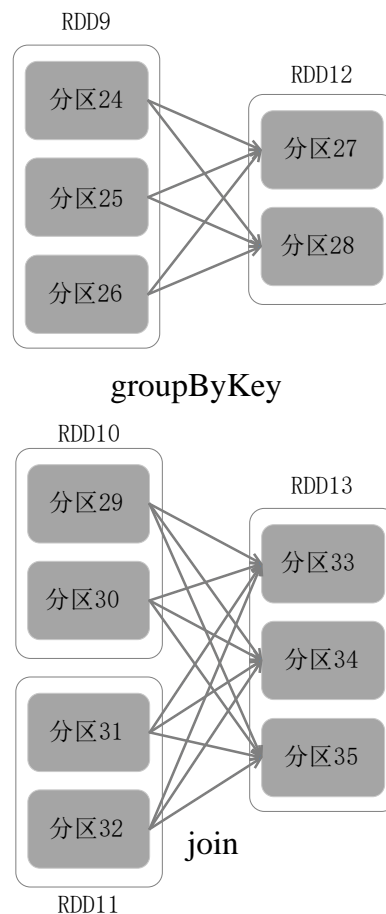
Shuffle操作



窄依赖和宽依赖



(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
- ✓ **宽依赖:** 存在一个父RDD的一个分区对应一个子RDD的多个分区

Hyracks



- 对抽象数据类型的集合的操作
- 实例化期间提供数据模型操作
- 如：
 - 排序操作接受一系列比较操作
 - 基于散列的分组操作接受一系列散列方法和比较操作
 - 散列分区的连接接受散列方法

- File reader/writers
- Mappers
- Sorters
- Joiners (各种类型)
- Aggregators
- 更多...

- N:M 哈希划分
- N:M 哈希划分合并 (输入有序)
- N:M 分区划分(使用分区向量)
- N:M 复制
- 1:1
- 更多...

- 修复用以传输数据的内存块
- 用于迭代序列化数据的游标
- 普通的操作（比较/散列/映射）可以直接在框架中规定数据上执行
- 目的：最小化垃圾回收次数以及最小化数据副本
 - 有趣的是：执行约60分钟的请求，总的GC时间约为1.8秒。

需要解决的问题

- 基本数据操作和用户自定义逻辑的步骤划分
- 基本数据操作和用户自定义逻辑的执行顺序

需要考虑的问题

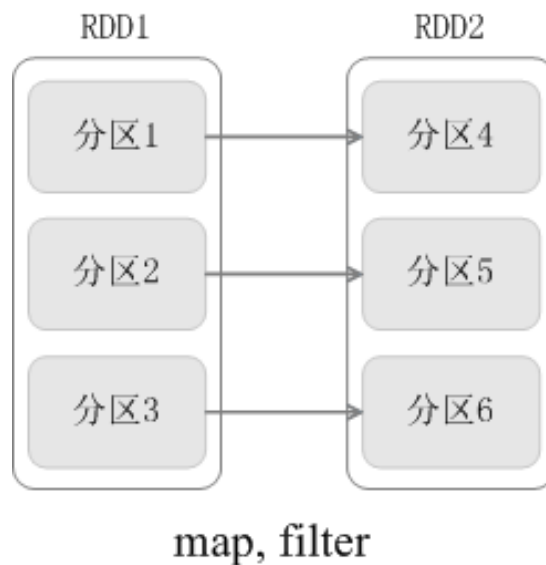
- 可用性高
- 效率高
- 容易理解
- 容易支持调试工具



阶段的划分

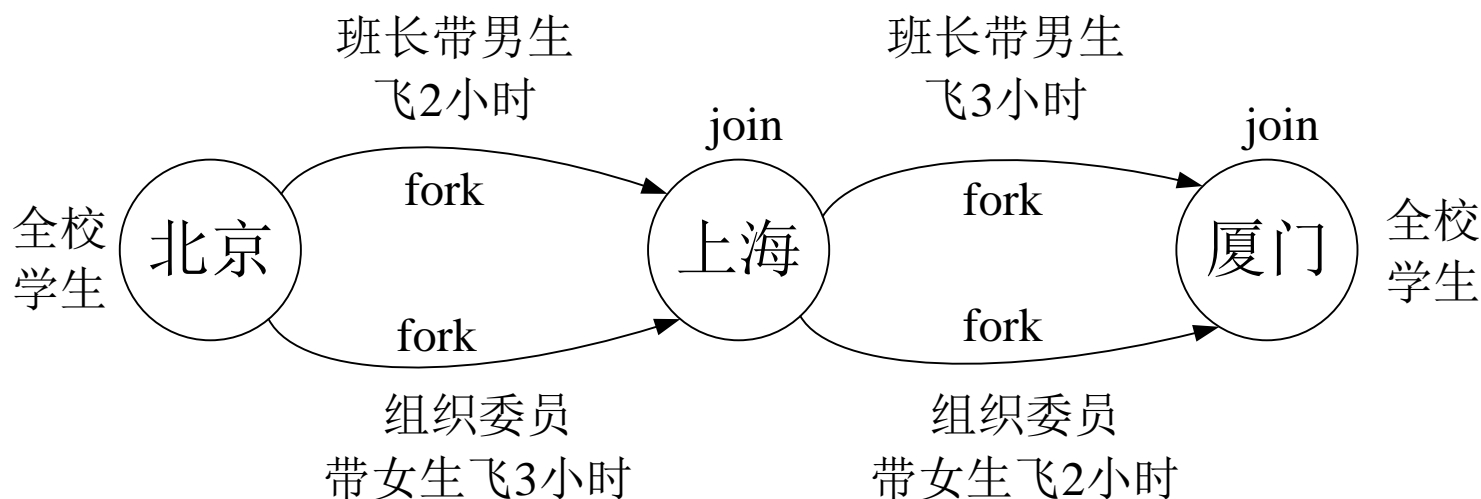
- 根据DAG 图中的RDD 依赖关系，把一个作业分成多个阶段。
- 阶段划分的依据是窄依赖和宽依赖：窄依赖对于作业优化很有利，宽依赖无法优化
- 逻辑上，每个RDD 操作都是一个fork/join（一种用于并行执行任务的框架），把计算fork 到每个RDD 分区，完成计算后对各个分区得到的结果进行join 操作，然后fork/join 下一个RDD 操作

- ✓ **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
- ✓ **宽依赖**：存在一个父RDD的一个分区对应一个子RDD的多个分区



fork/join的优化原理

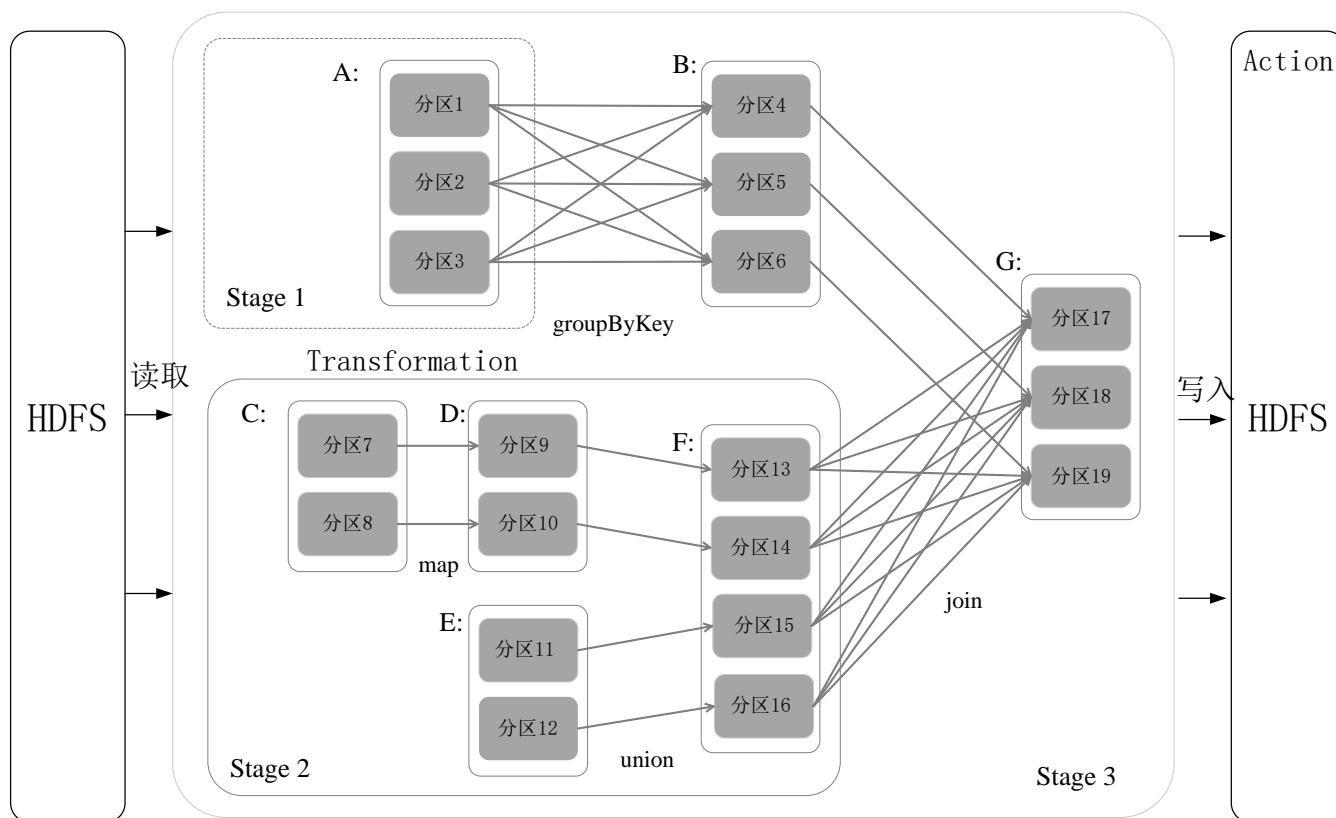
举例：一个学校（含2个班级）完成从北京到厦门的旅行



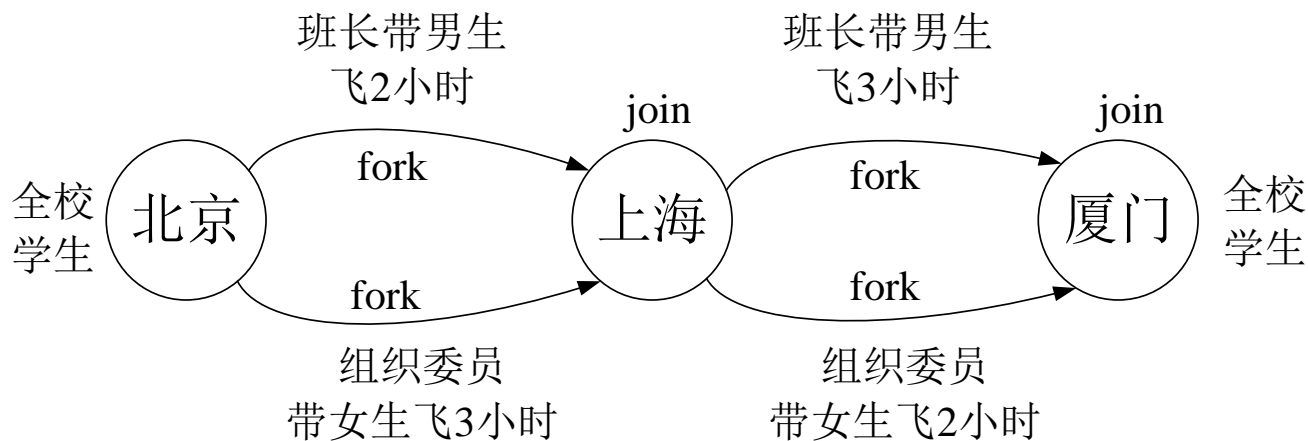
阶段的划分

宽依赖往往对应着shuffle操作(多对一,汇总,多节点), 需要在运行过程中将同一个父RDD的分区传入到不同的子RDD分区中, 中间可能涉及多个节点之间的数据传输; 而窄依赖的每个父RDD的分区只会传入到一个子RDD分区中, 通常可以在一个节点内完成转换

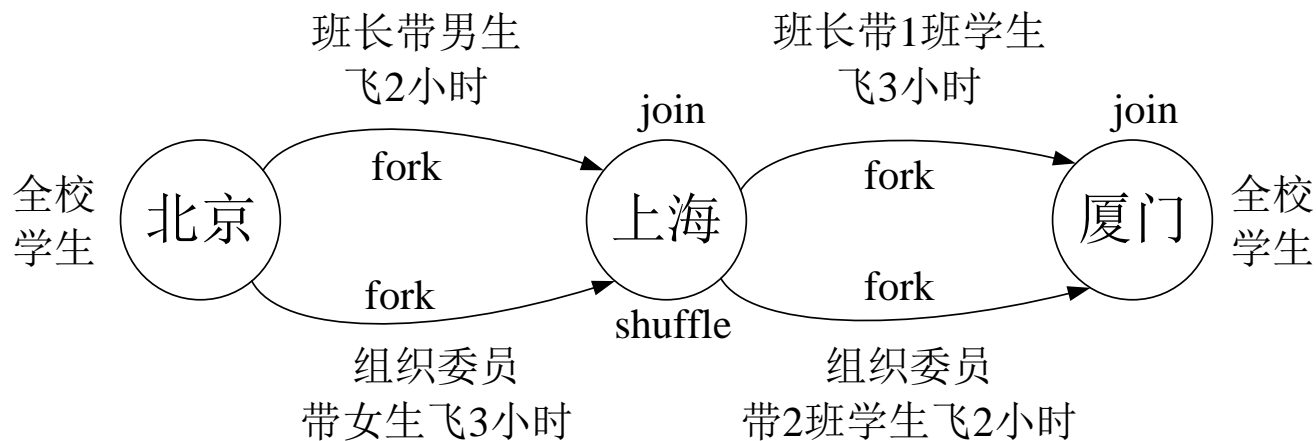
窄依赖可以实现“流水线”优化
宽依赖无法实现“流水线”优化



阶段的划分



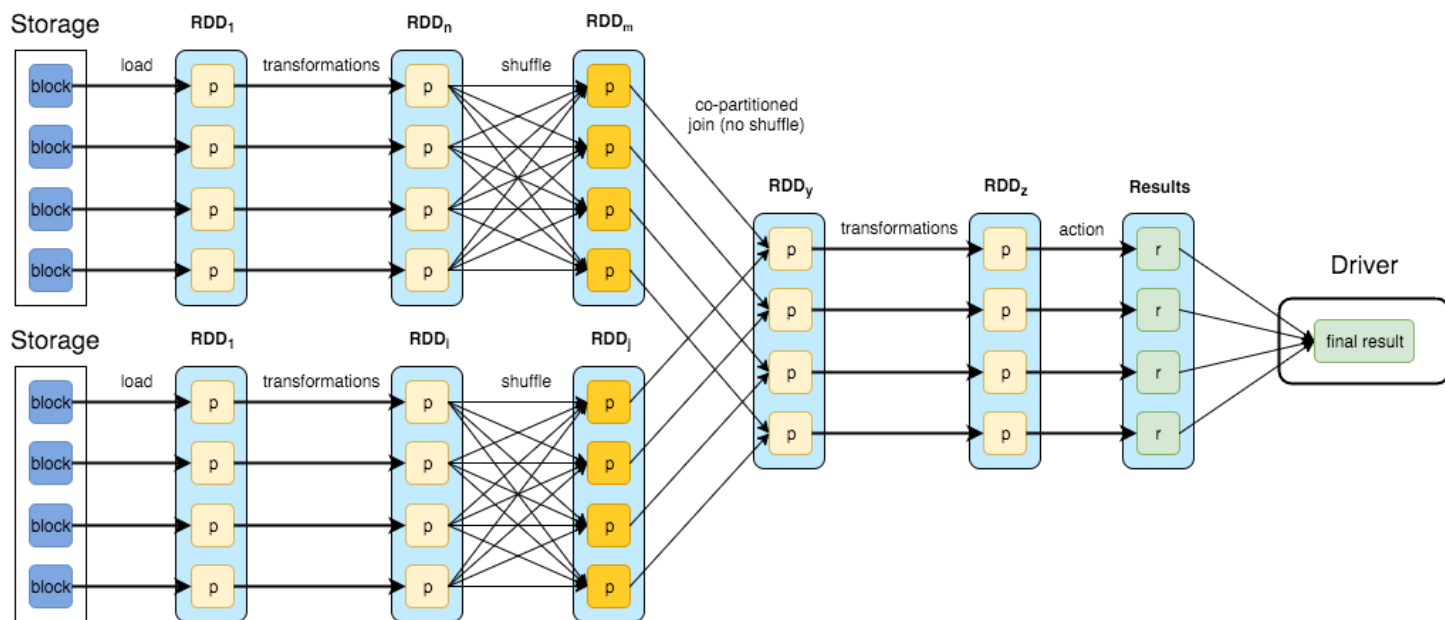
(a) 窄依赖



(a) 宽依赖

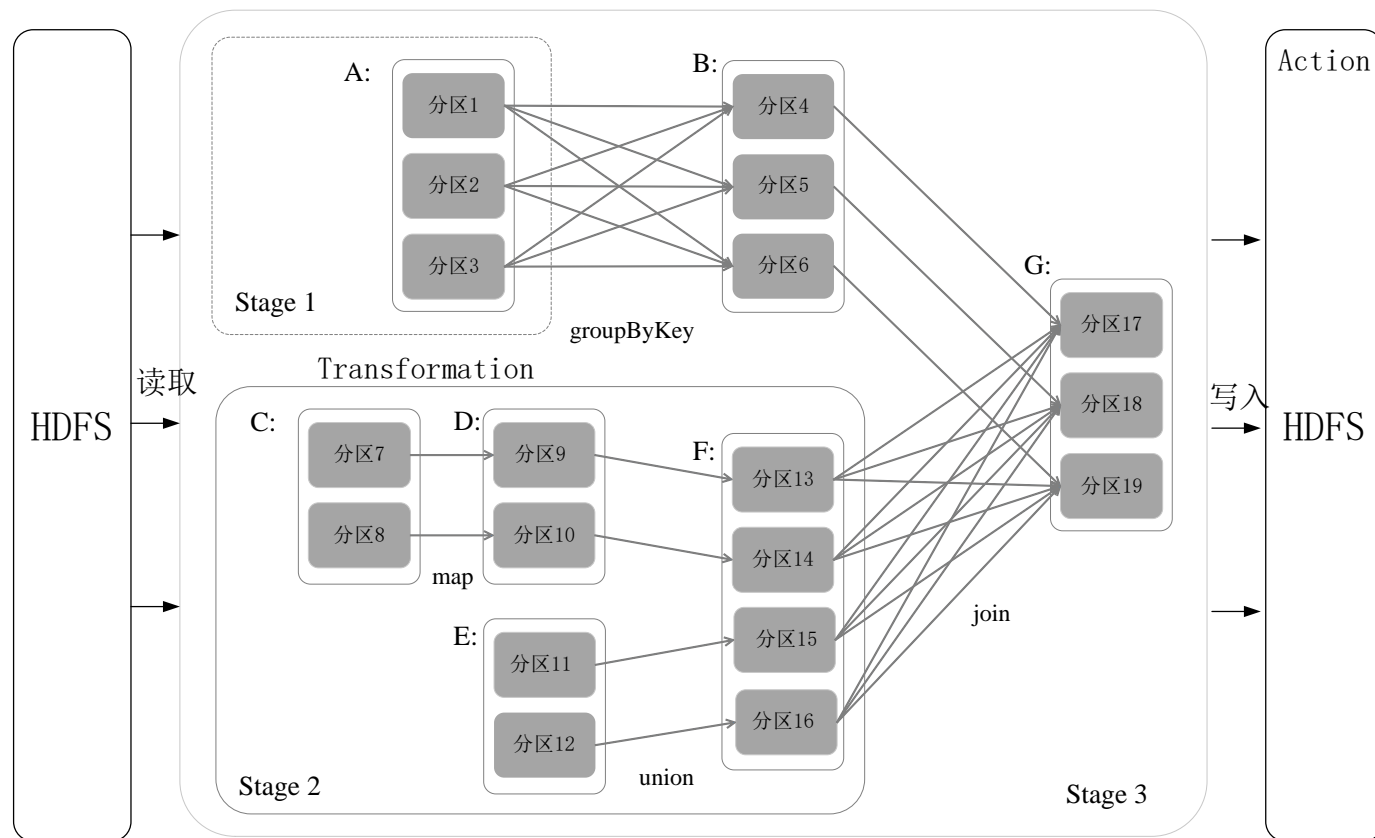
划分的方法

- 通过分析各个RDD的依赖关系生成了DAG，再通过分析各个RDD中的分区之间的依赖关系来决定如何划分Stage
- 具体步骤(贪心法):
 - ✓ 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



划分的实例

任务被分成三个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

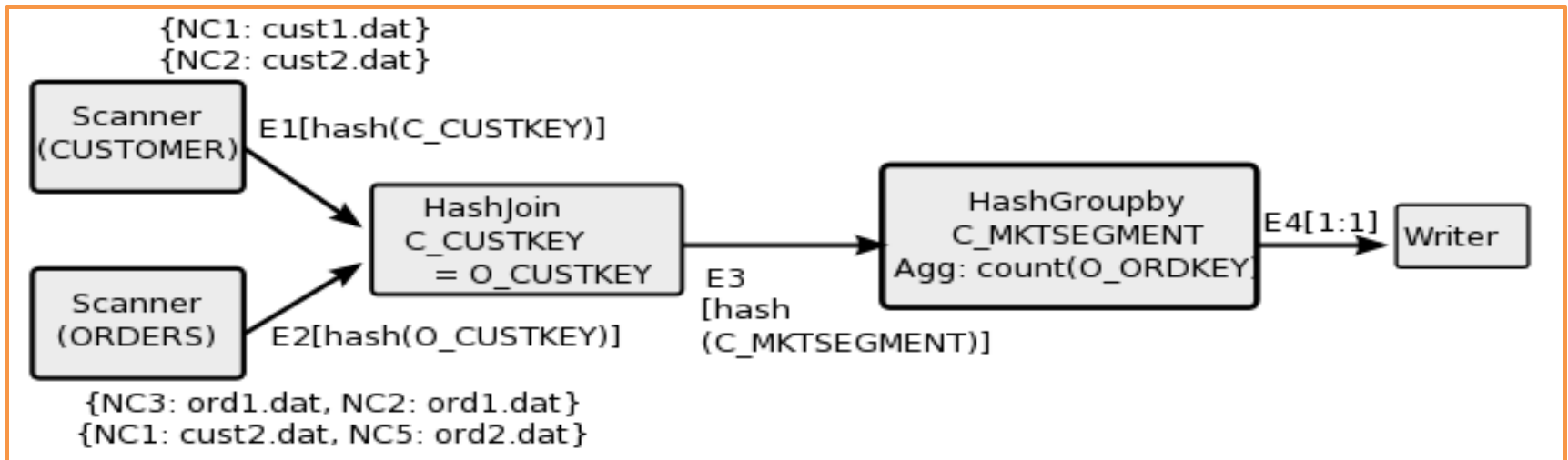
分区7通过map操作生成的分区9，可以不用等待分区8到分区10这个map操作的计算结束，而是继续进行union操作，得到分区13，这样流水线执行大大提高了计算的效率

Hyracks

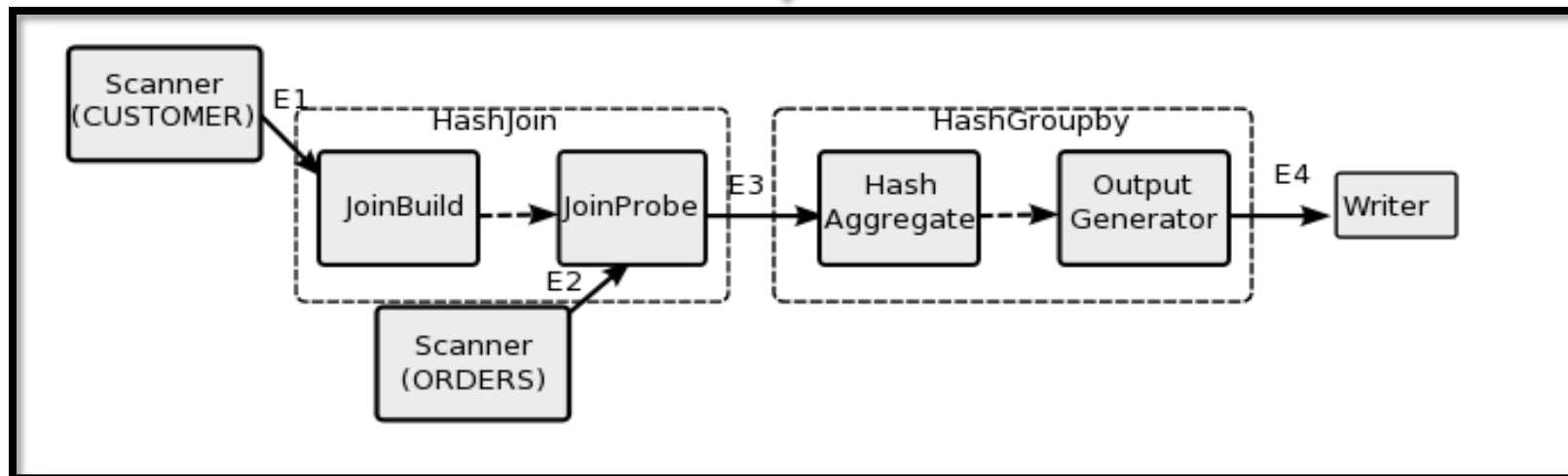
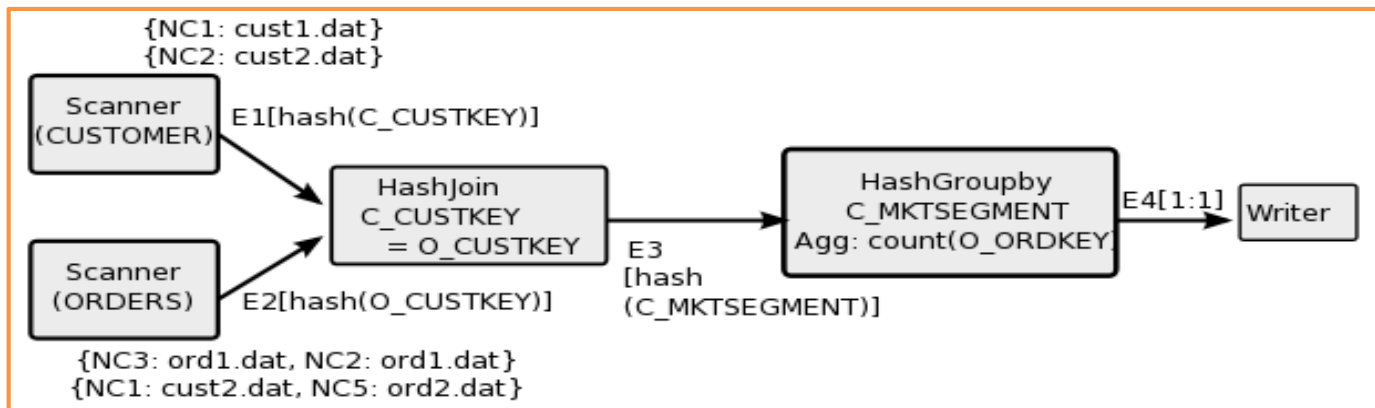


Hyracks中的Jobs

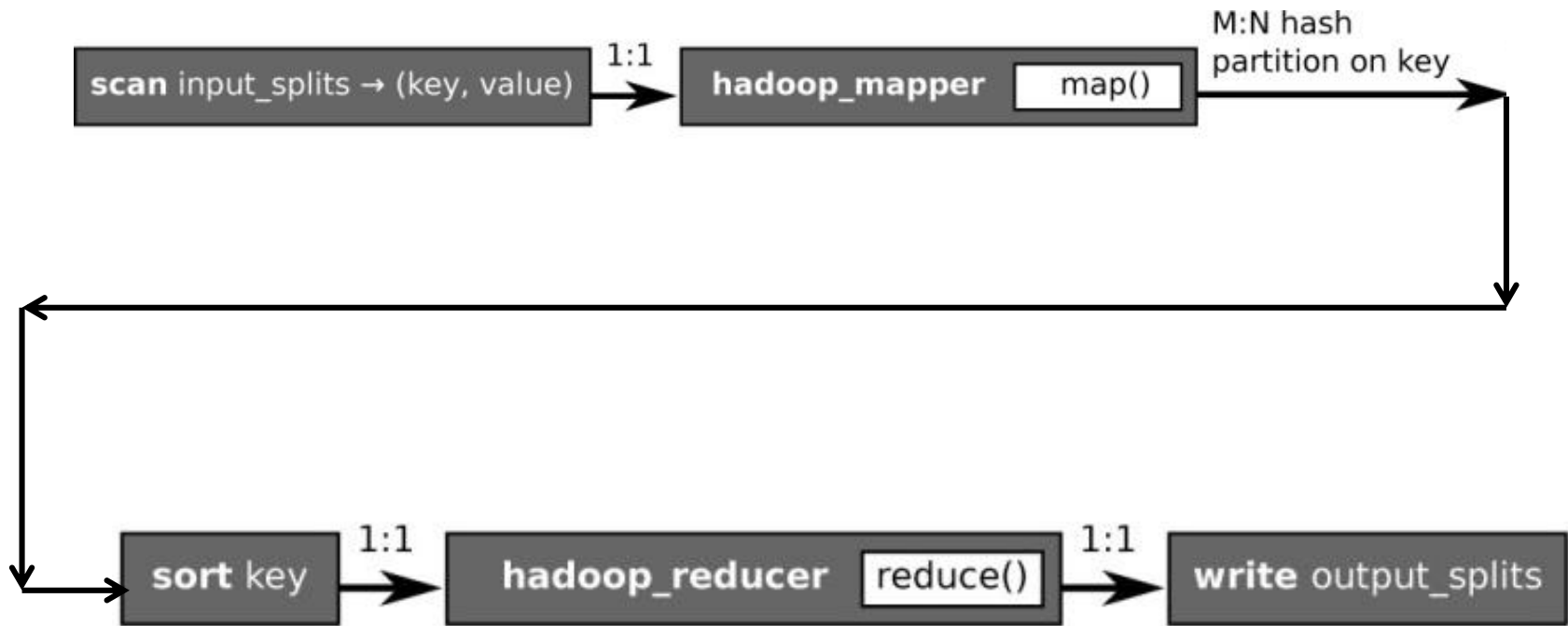
- Job是Hyracks客户端提交的操作单位
- Job是operator(操作)和connector(连接)的有向无环数据流
 - operator使用/产生数据块
 - connector在operator之间再分配/传递数据



Hyracks: Operator的活动



Hyracks实现MapReduce



需要解决的问题

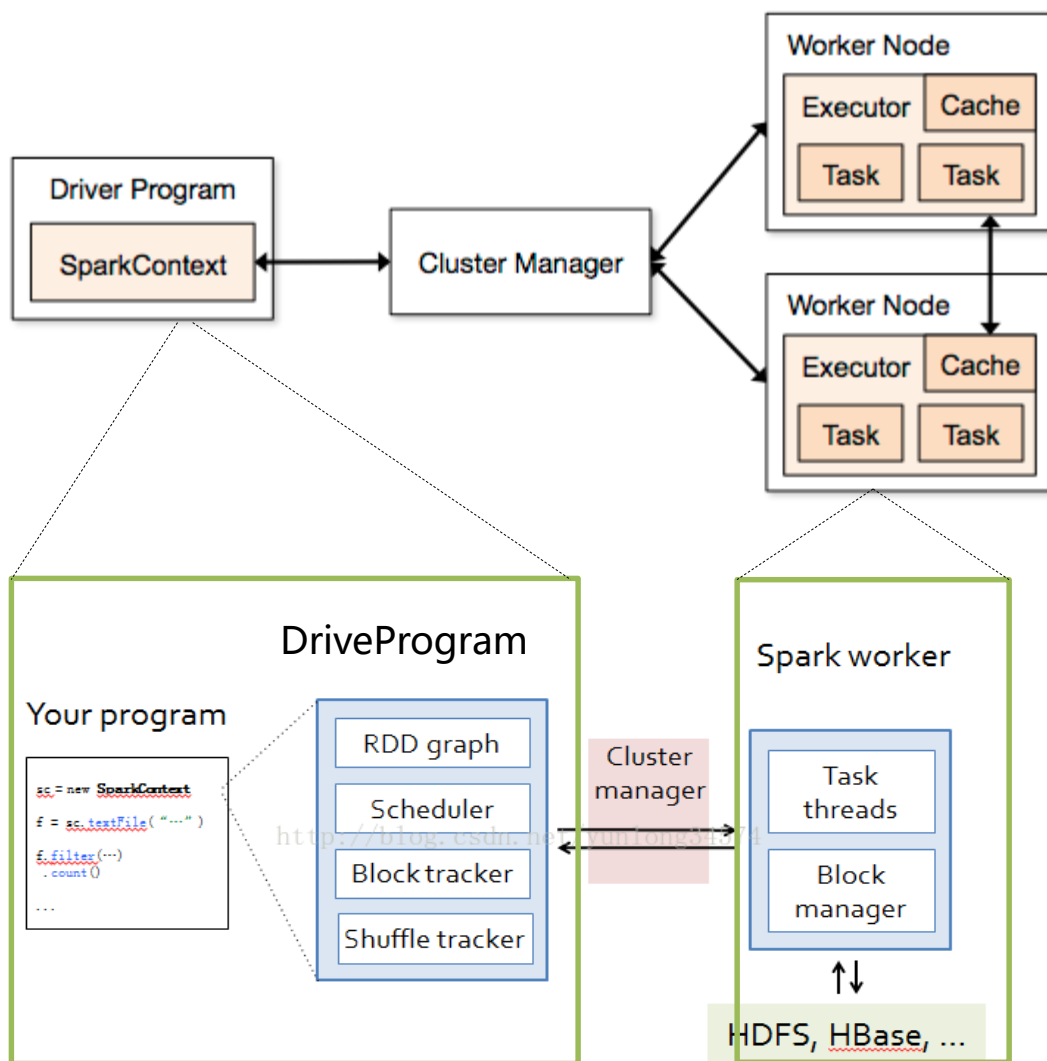
- 执行过程中如何执行操作？

需要考虑的问题

- 可扩展性高
- 效率高



Spark端到端流程

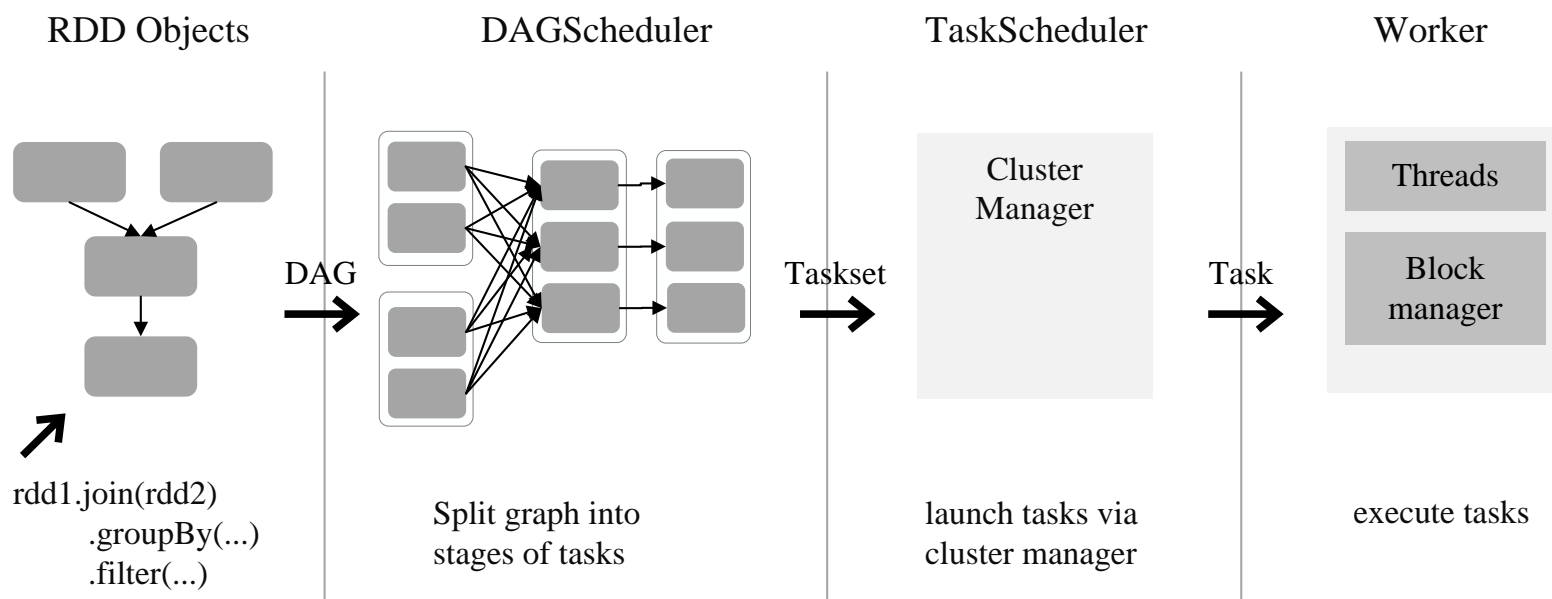


整体流程

- Spark的Driver Program (简称Driver)包含用户的应用程序
- Driver完成task的解析和生成
- Driver向Cluster Manager (集群资源管理器) 申请运行task需要的资源。
- 集群资源管理器为task分配满足要求的节点, 并在节点按照要求创建Executor
- 创建的Executor向Driver注册。
- Driver将spark应用程序的代码和文件传送给分配的executor
- executor运行task, 运行完之后将结果返回给Driver或者写入HDFS或其他介质。

RDD运行过程

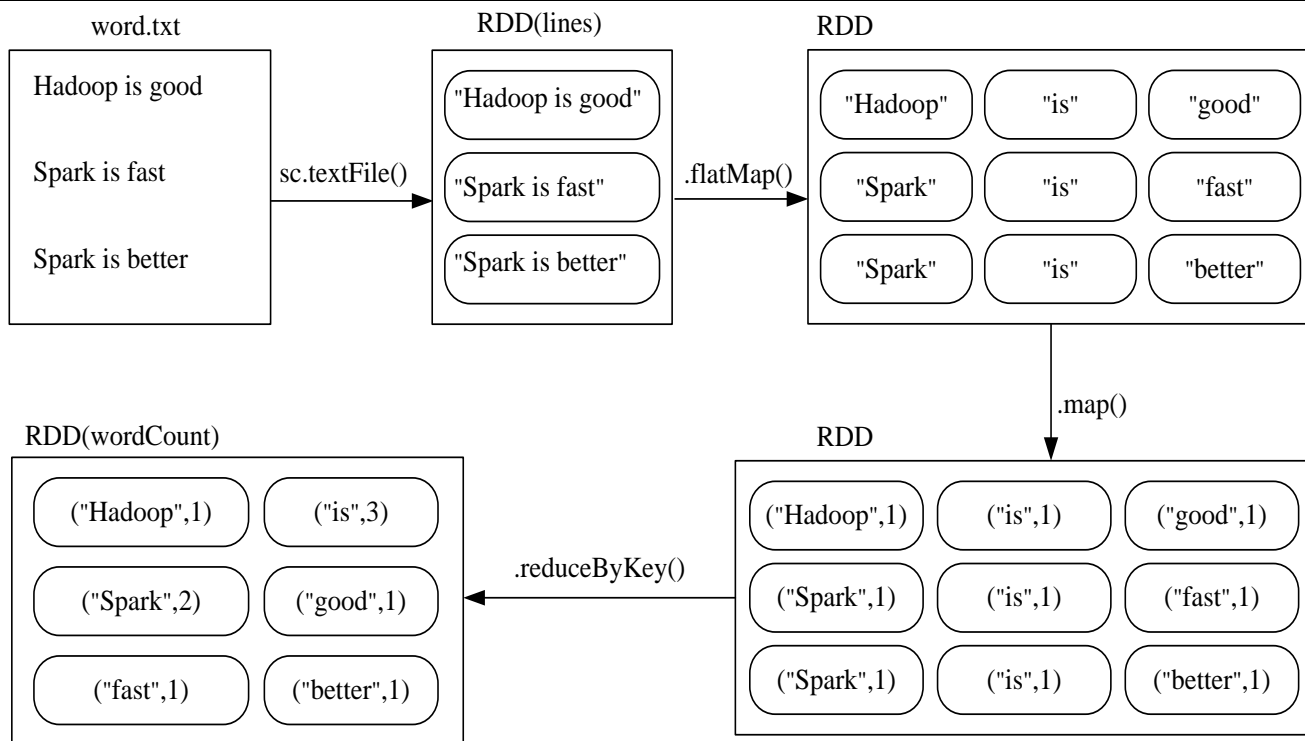
- (1) 创建RDD对象;
- (2) SparkContext负责计算RDD之间的依赖关系, 构建DAG;
- (3) DAGScheduler负责把DAG图分解成多个Stage, 每个Stage中包含了多个Task, 每个Task会被TaskScheduler分发给各个WorkerNode上的Executor去执行。



综合案例

本地文件word.txt包含很多行文本，每行文本由多个单词构成，单词之间用空格分隔。可以使用如下语句进行词频统计（即统计每个单词出现的次数）：

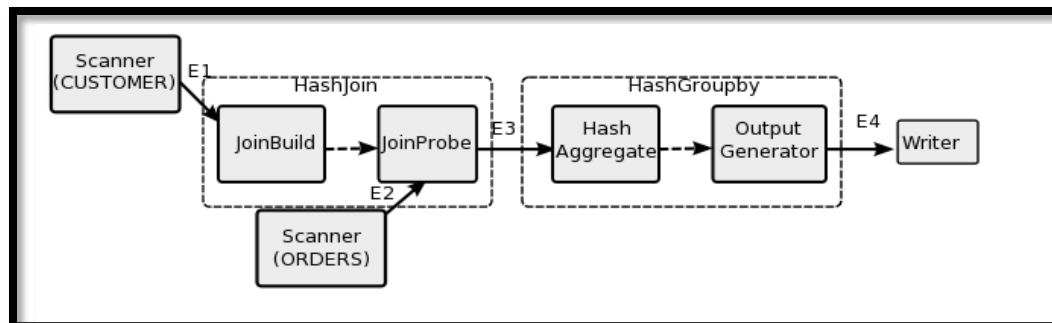
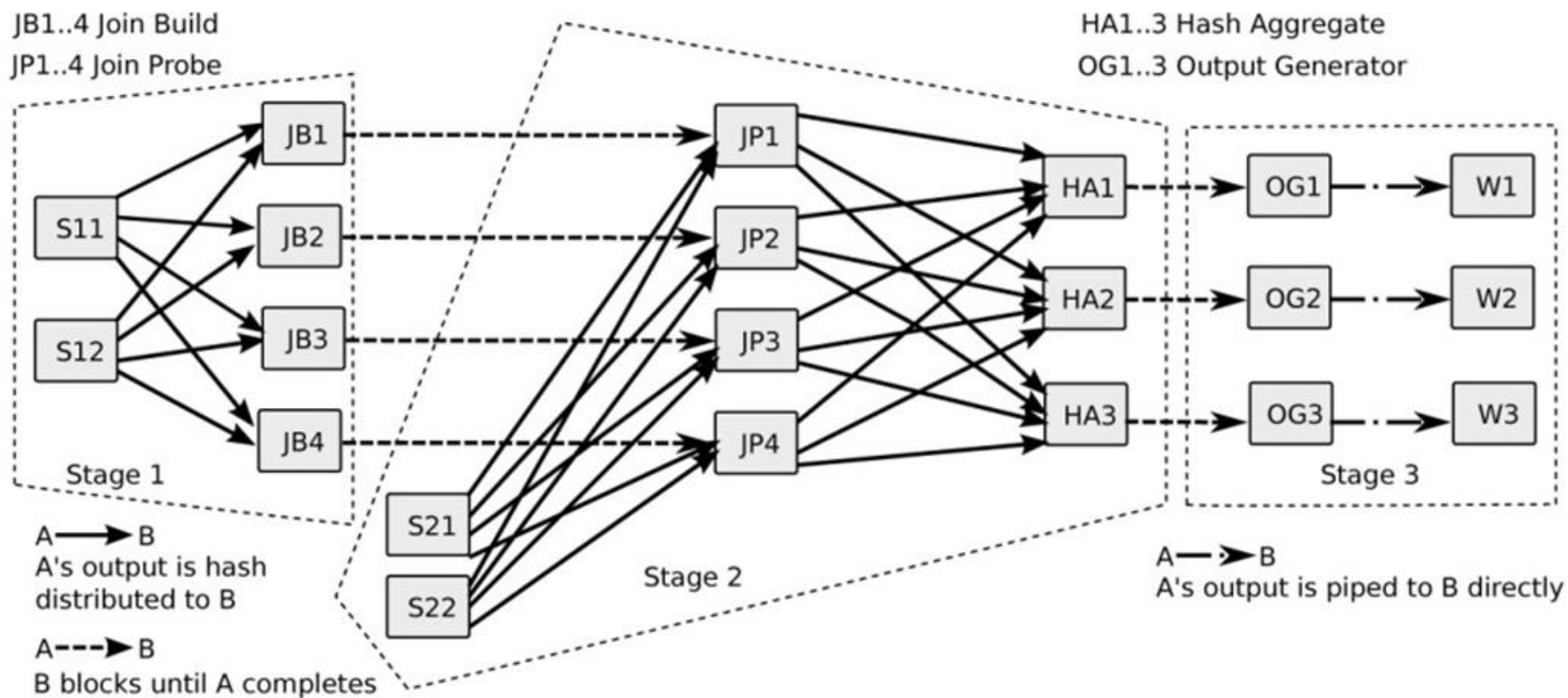
```
scala> val lines = sc. //代码一行放不下，可以在圆点后回车，在下行继续输入
| textFile("file:///usr/local/spark/mycode/wordcount/word.txt")
scala> val wordCount = lines.flatMap(line => line.split(" ")).
| map(word => (word, 1)).reduceByKey((a, b) => a + b)
scala> wordCount.collect()
scala> wordCount.foreach(println)
```



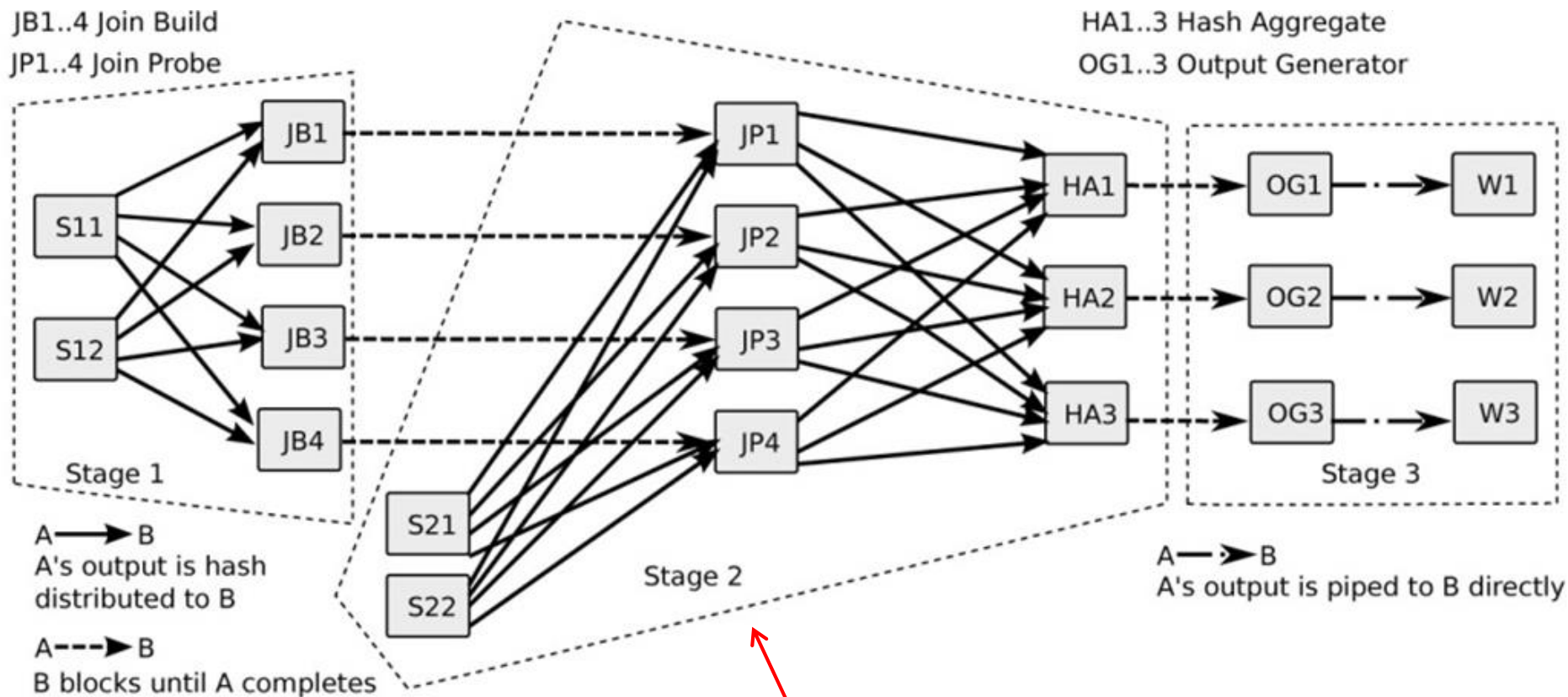
Hyracks



并行执行的实例



并行执行的实例



阶段2
(输入阶段完成后启动)

需要解决的问题

如何保证执行的正确性

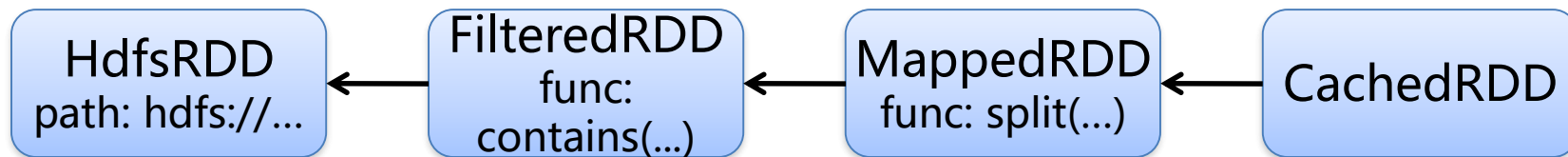
需要考虑的问题

- 结点出错怎么办?
- 分布式系统出现读脏写脏怎么办?



- RDD维护着可以用来重建丢失分区的信息
- 例如:

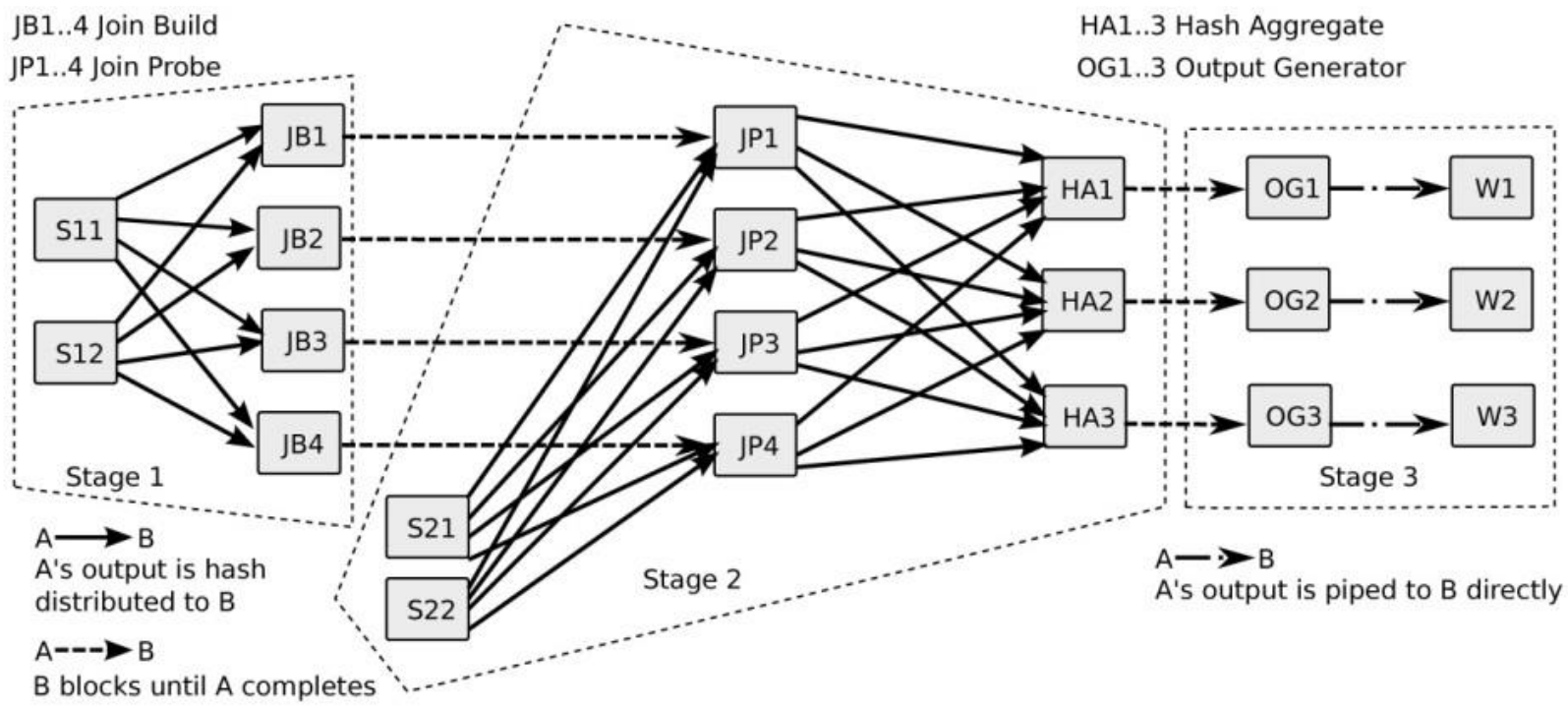
```
cachedMsgs = textFile(...).filter(_.contains( "error" ))  
                        .map(_.split( '\t' )(2))  
                        .cache()
```



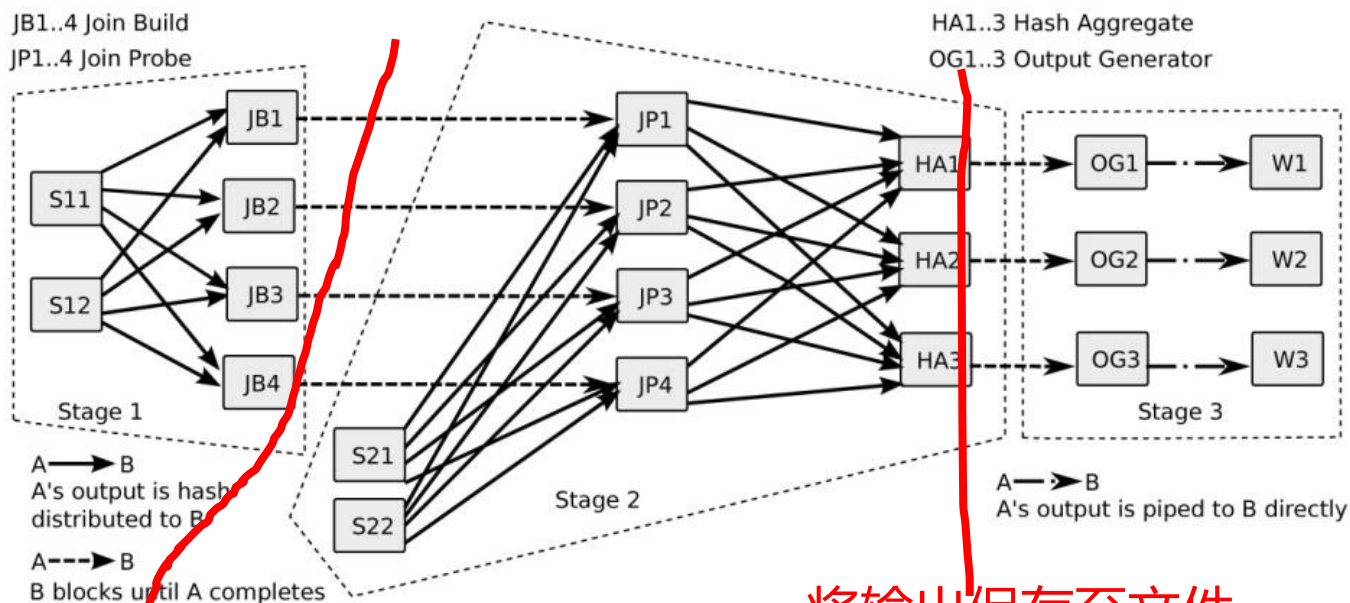
Hyracks



Hyracks 容错: 可否继续工作?

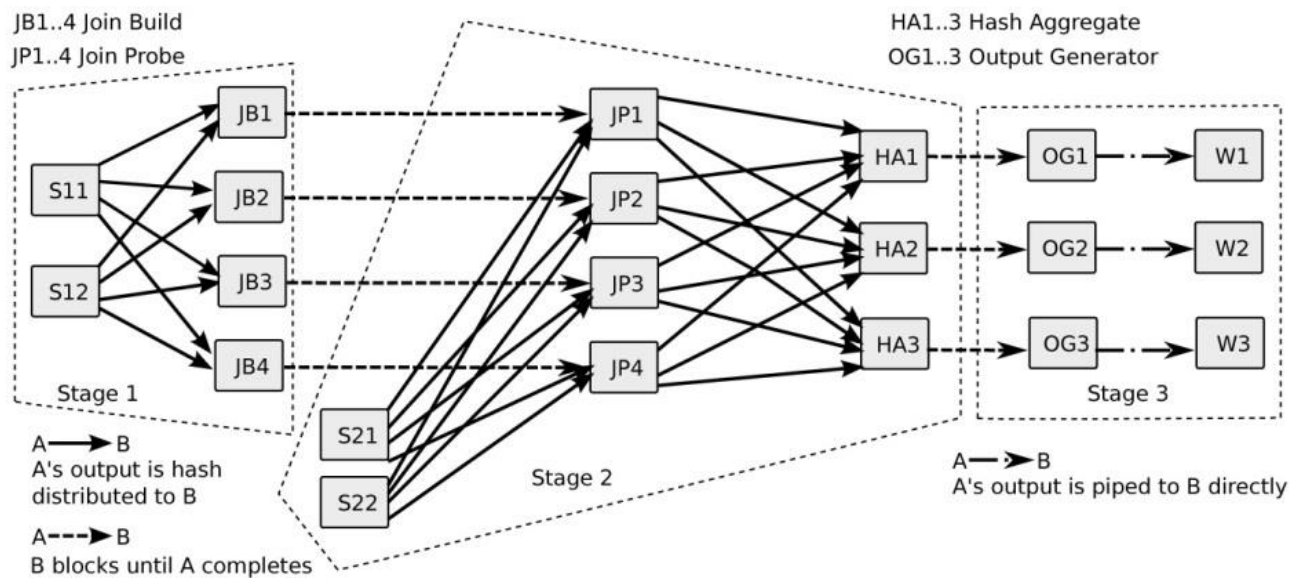


Hyracks 容错: 可否继续工作?



Hadoop/MR 方法: 保存部分结果永久存储;失败后, 重做所有的工作来重建丢失的数据

Hyracks 容错: 可否继续工作?



我们可以做的更好吗?

每个进程保留以前的结果, 直到不再需要

Pi 输出: ~~r1, r2, r3, r4~~, r5, r6, r7, r8

已经找到到最终结果的路径

当前输出

需要解决的问题

- 何时存储数据?
- 数据存储成什么形式?

需要考虑的问题

- 高(时间/存储)效率
- 内存满足要求



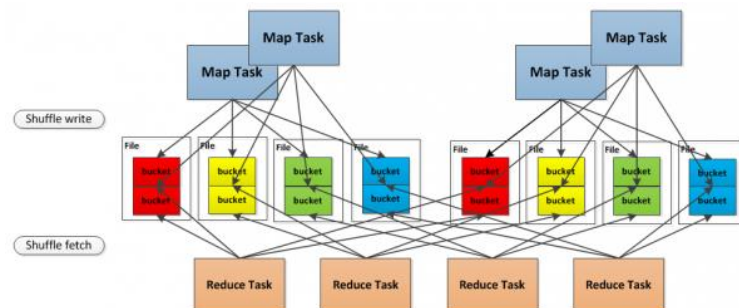
Spark中的存储机制

- RDD缓存：包括基于内存和磁盘的缓存

- ✓ 分区和数据块是一一对应的
- ✓ 在内部建立RDD分区和数据块之间的映射，需要读取缓存的RDD时，根据映射关系取得分区对应的数据块
- ✓ 内存缓存=哈希表+存取策略
- ✓ 一个数据块对应着文件系统中的文件，文件名和块名称的映射关系是通过哈希算法计算所得的

- Shuffle数据的持久化

- Shuffle数据块必须是在磁盘上进行缓存的
- Shuffle数据块的存储有两种方式：
 - 将Shuffle数据块映射成文件
 - 将Shuffle数据块映射成文件中的一段



Hyracks



Database

