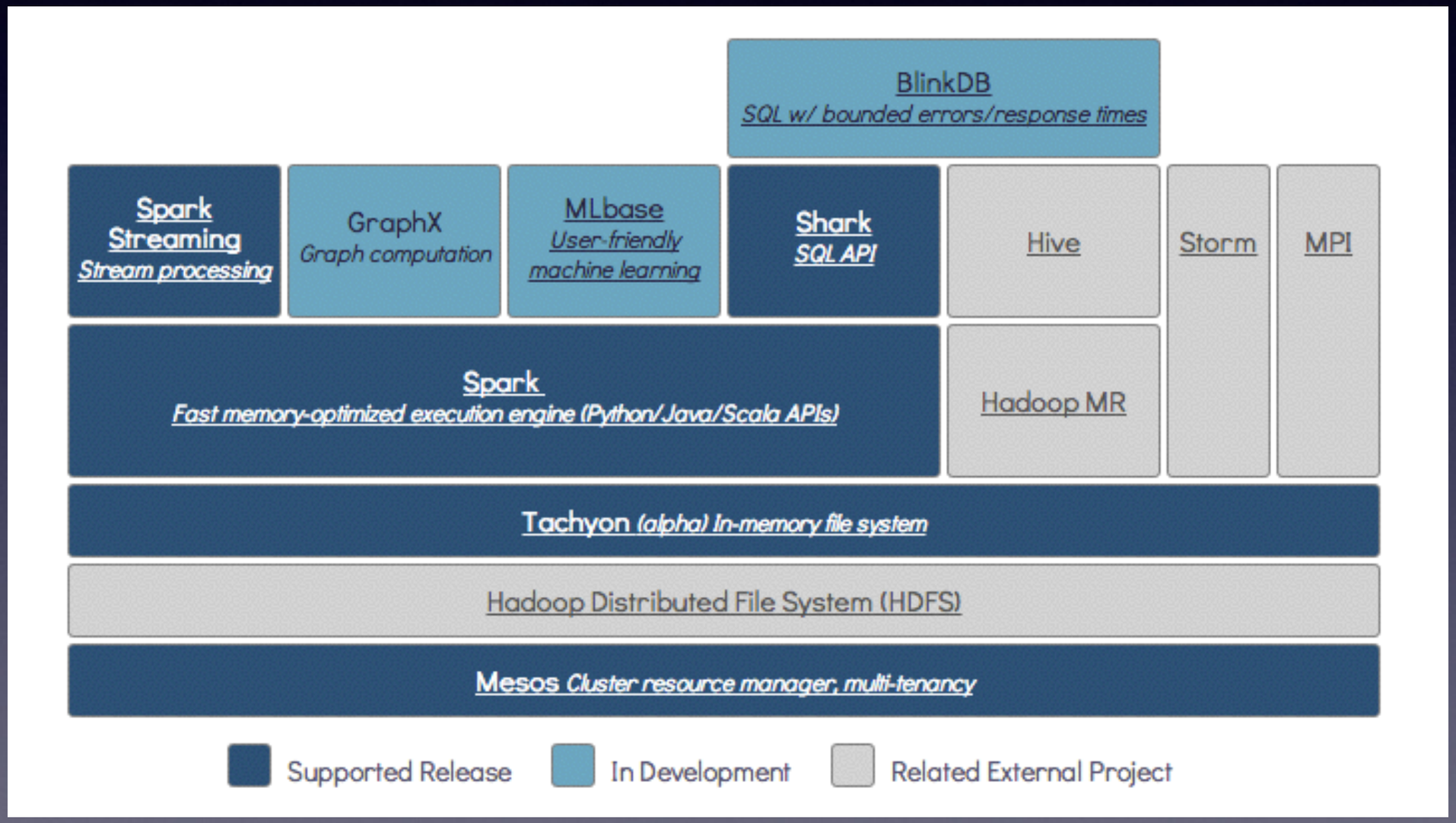
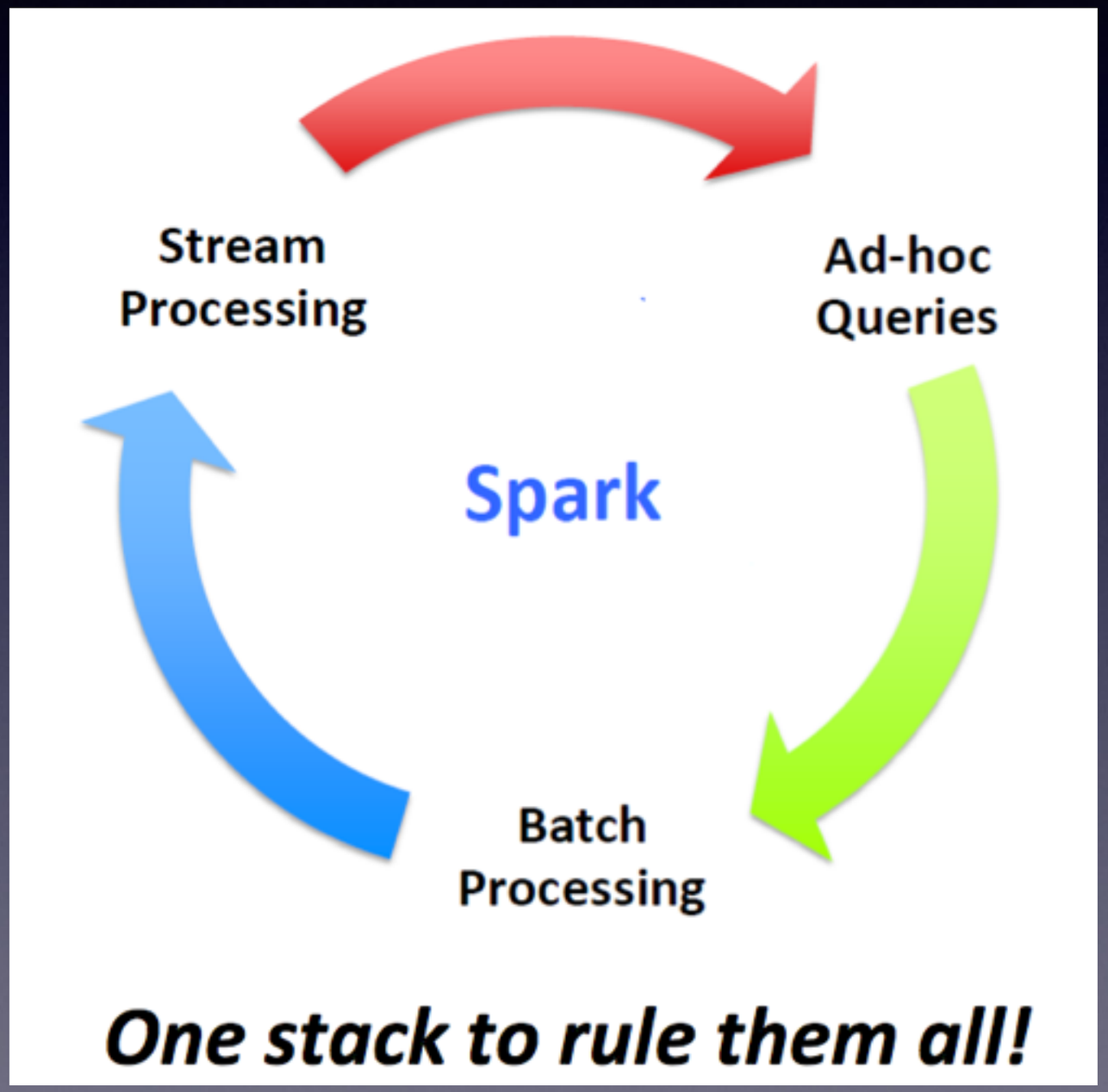
深入浅出Spark

BDAS



the Berkeley Data Analytics Stack

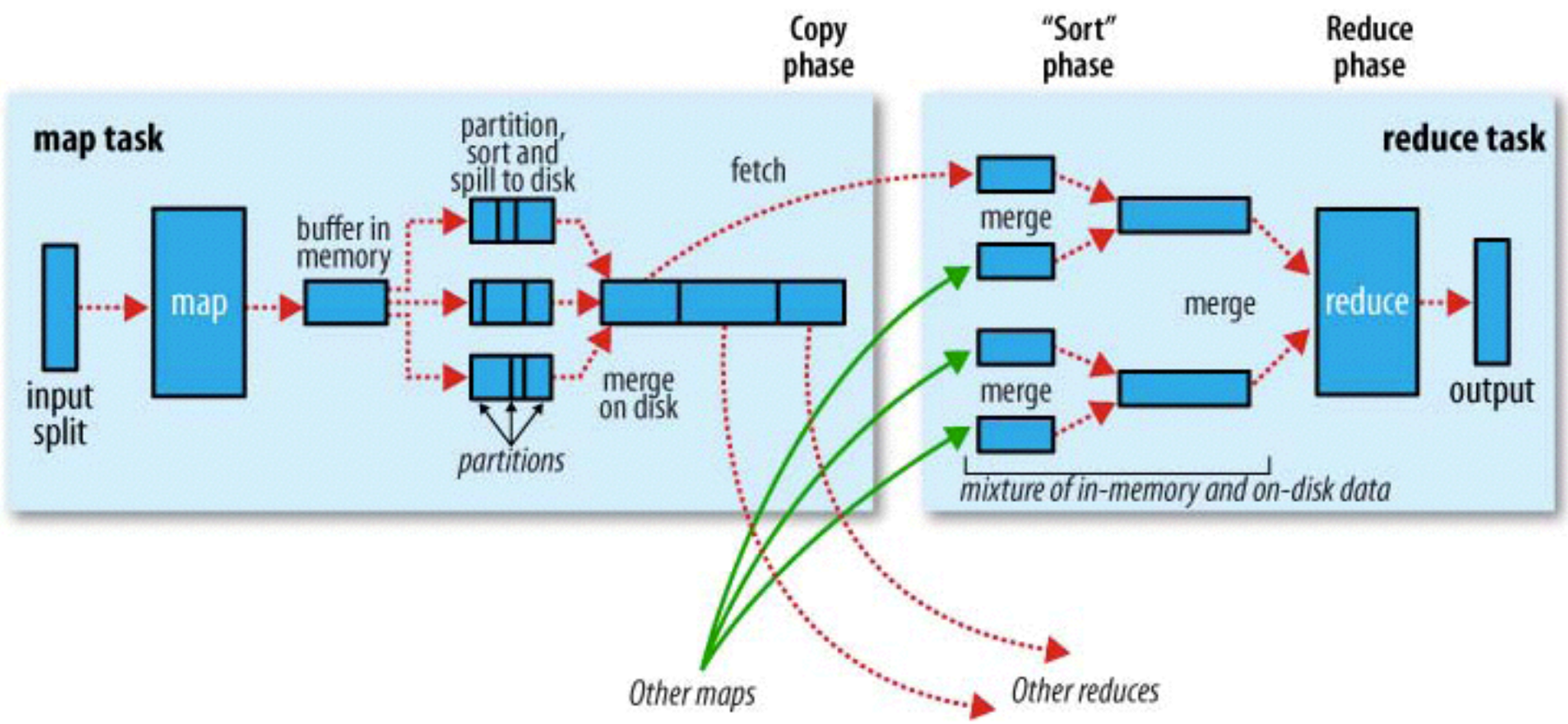
搞定所有！



What's Spark

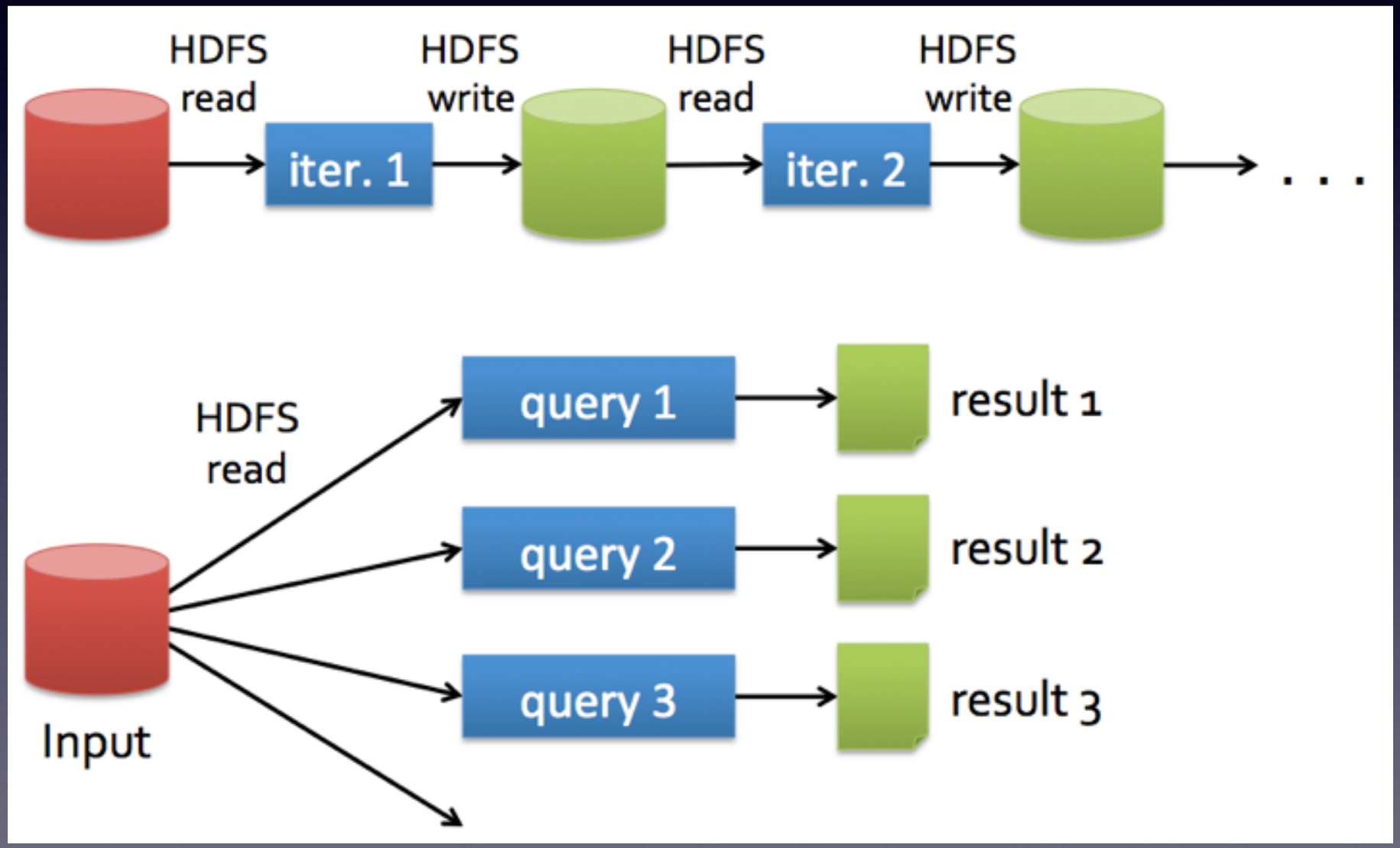
- Apache Spark is an open source cluster computing system that aims to make data analytics fast — both fast to run and fast to write

再看Hadoop MR

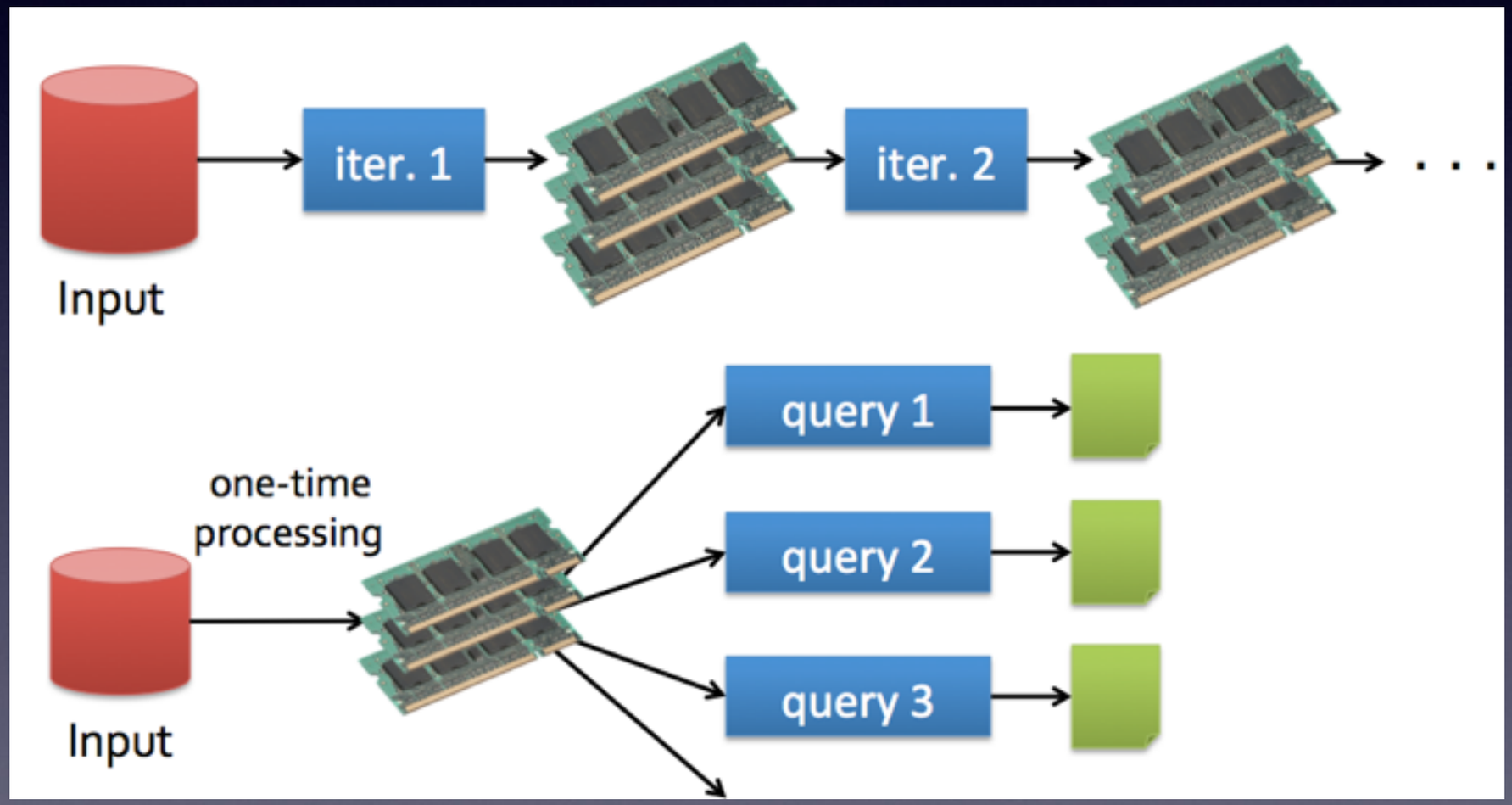


Hadoop的数据共享？慢！

- 为什么慢？？？ 额外的复制，序列化和磁盘IO开销。



Spark的共享数据？快！



Spark快在哪里?

- 内存计算
- DAG

很多优化措施其实都是相通的，譬如说**delay scheduling.**

Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling

Spark API

- 支持3种语言的API
  - Scala(2.10.x)
  - Python(pyspark,推荐Python2.7)
  - Java(请用Java8)

通过哪些模式运行Spark呢?

- 有4种模式可以运行
  - local(多用于测试)
  - Standalone
  - Mesos
  - YARN

Scala快速入门

- Scala语言?
  - 基于JVM的FP+OO
  - 静态类型
  - 和Java可以互操作

Scala变量声明

- var x : Int = 7
- val x = 7 //自动类型推导
- val y = “hi” //只读，相当于Java里的final

Scala函数

```
def square(x : Int) : Int = x * x
```

```
def square(x : Int) : Int = {x*x} //在block中的最后一个值将被返回
```

```
def announce(text : String) {println(text)}
```

Scala泛型

```
var arr = new Array[Int](8)
var lst = List(1,2,3) //1st的类型是List[Int]

//索引访问
arr(5) = 7
println(lst(1))
```

Scala—FP的方式处理集合

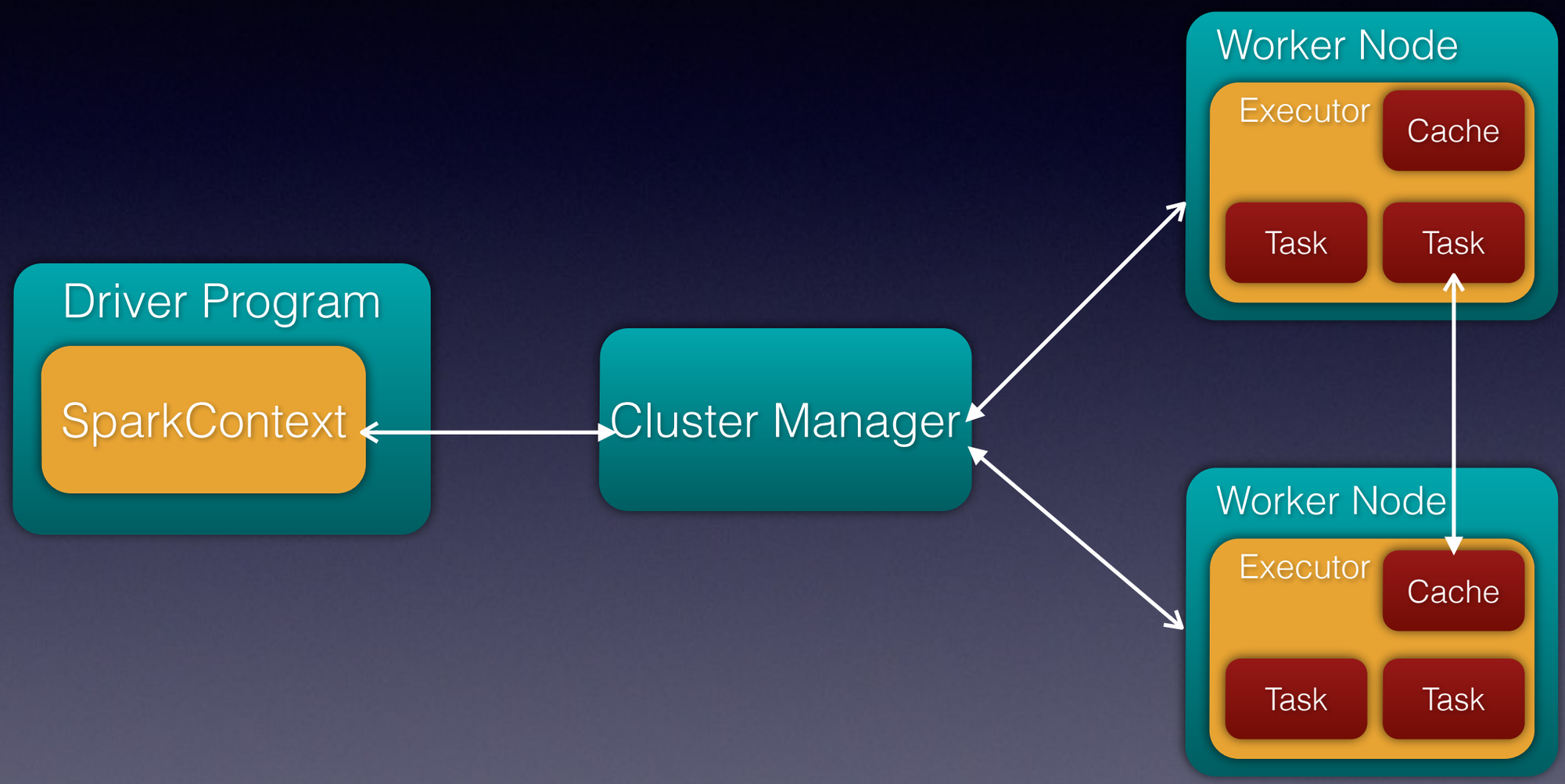
```
val list = List(1,2,3)
list.foreach(x=>println(x))//打印出1,2,3
list.foreach(println)

list.map(x => x + 2) //List(3,4,5)
list.map(_+2)

list.filter(x => x % 2 == 1) //List(1,3)
list.filter(_ % 2 == 1)

list.reduce((x,y) => x + y) //6
list.reduce(_+_)
```

Scala — 闭包

```
(x : Int) => x + 1
x => x + 1
_+1
x =>{
    val numberToAdd = 1
    x + numberToAdd
}
//如果闭包很长，可以考虑作为参数传入
def addOne(x : Int) : Int = x + 1
list.map(addOne)
```

OK，回到Spark~

Spark Runtime



核心抽象:RDD

- Resilient Distributed Dataset
  - A list of partitions
  - A function for computing each split
  - A list of dependencies on other RDDs
  - Optionally, a Partitioner for key-value RDDs (e.g. to say that the RDD is hash-partitioned)
  - Optionally, a list of preferred locations to compute each split on (e.g. block locations for an HDFS file)

如何创建RDD

- 直接从集合转化

  sc.parallelize(List(1,2,3,4,5,6,7,8,9,10))

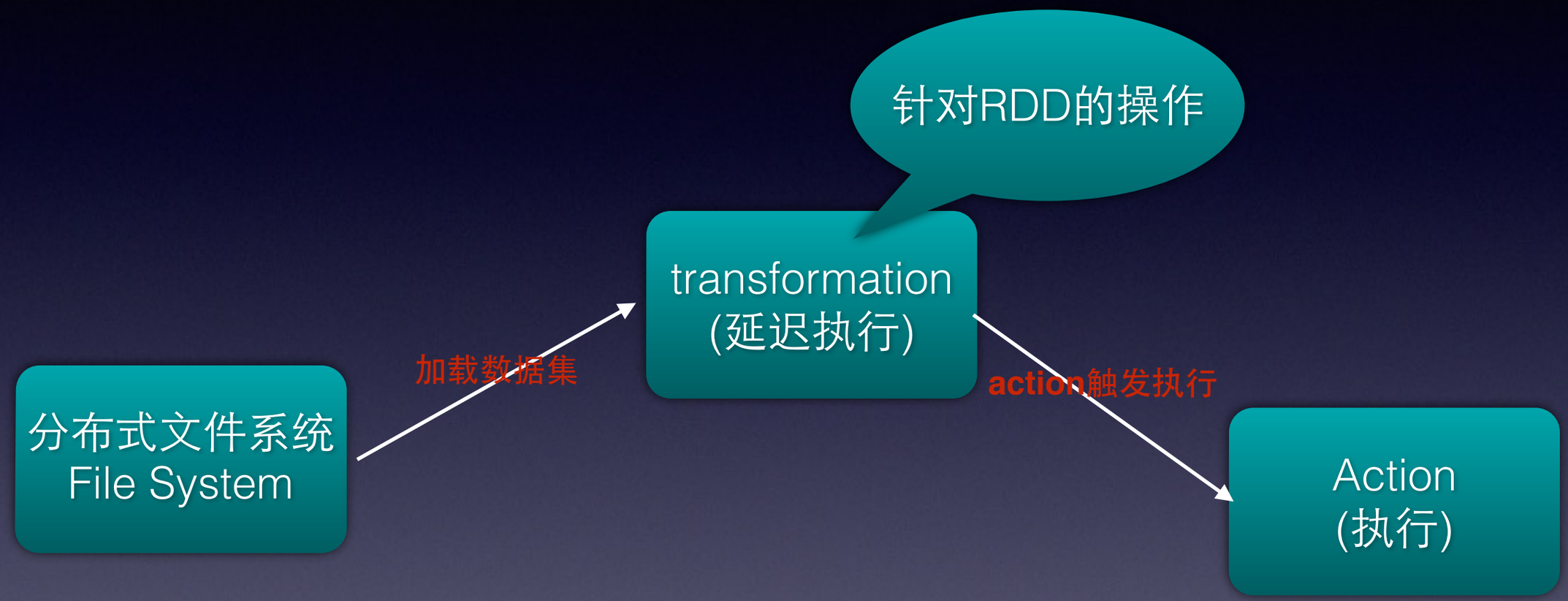
- 从各种(分布式)文件系统来

  sc.textFile(“README.md”)　　sc.textFile(“hdfs://xxx”)

- 从现存的任何Hadoop InputFormat而来

  sc .hadoopFile(keyClass,valueClass,inputFormat,conf)

流程示意



ps:RDD可以从集合直接转换而来，也可以由从现存的任何Hadoop InputFormat而来,亦或者HBase等等。

缓存策略

```
class StorageLevel private(
    private var useDisk_ : Boolean,
    private var useMemory_ : Boolean,
    private var deserialized_ : Boolean,
    private var replication_ : Int = 1)
```

```
val NONE = new StorageLevel(false, false, false)
val DISK_ONLY = new StorageLevel(true, false, false)
val DISK_ONLY_2 = new StorageLevel(true, false, false, 2)
val MEMORY_ONLY = new StorageLevel(false, true, true)
val MEMORY_ONLY_2 = new StorageLevel(false, true, true, 2)
val MEMORY_ONLY_SER = new StorageLevel(false, true, false)
val MEMORY_ONLY_SER_2 = new StorageLevel(false, true, false, 2)
val MEMORY_AND_DISK = new StorageLevel(true, true, true)
val MEMORY_AND_DISK_2 = new StorageLevel(true, true, true, 2)
val MEMORY_AND_DISK_SER = new StorageLevel(true, true, false)
val MEMORY_AND_DISK_SER_2 = new StorageLevel(true, true, false, 2)
```

cache默认

transformation & action

Transformations	$map(f : T \Rightarrow U)$	:	$RDD[T] \Rightarrow RDD[U]$
	$filter(f : T \Rightarrow Bool)$	:	$RDD[T] \Rightarrow RDD[T]$
	$flatMap(f : T \Rightarrow Seq[U])$	:	$RDD[T] \Rightarrow RDD[U]$
	$sample(fraction : Float)$	:	$RDD[T] \Rightarrow RDD[T]$ (Deterministic sampling)
	$groupByKey()$	:	$RDD[(K, V)] \Rightarrow RDD[(K, Seq[V])]$
	$reduceByKey(f : (V, V) \Rightarrow V)$	:	$RDD[(K, V)] \Rightarrow RDD[(K, V)]$
	$union()$	:	$(RDD[T], RDD[T]) \Rightarrow RDD[T]$
	$join()$	:	$(RDD[(K, V)], RDD[(K, W)]) \Rightarrow RDD[(K, (V, W))]$
	$cogroup()$	:	$(RDD[(K, V)], RDD[(K, W)]) \Rightarrow RDD[(K, (Seq[V], Seq[W]))]$
	$crossProduct()$	:	$(RDD[T], RDD[U]) \Rightarrow RDD[(T, U)]$
	$mapValues(f : V \Rightarrow W)$	:	$RDD[(K, V)] \Rightarrow RDD[(K, W)]$ (Preserves partitioning)
	$sort(c : Comparator[K])$	:	$RDD[(K, V)] \Rightarrow RDD[(K, V)]$
	$partitionBy(p : Partitioner[K])$	:	$RDD[(K, V)] \Rightarrow RDD[(K, V)]$
Actions	$count()$	:	$RDD[T] \Rightarrow Long$
	$collect()$	:	$RDD[T] \Rightarrow Seq[T]$
	$reduce(f : (T, T) \Rightarrow T)$	:	$RDD[T] \Rightarrow T$
	$lookup(k : K)$	:	$RDD[(K, V)] \Rightarrow Seq[V]$ (On hash/range partitioned RDDs)
	$save(path : String)$	:	Outputs RDD to a storage system, *e.g.*, HDFS

Transformation

```
val nums = sc.parallelize(List(1,2,3,4,5,6,7,8,9))
```

nums: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[0] at parallelize at <console>:12

```
val squares = nums.map(x => x*x)
```

squares: org.apache.spark.rdd.RDD[Int] = MappedRDD[1] at map at <console>:14

```
val even = nums.filter(_ % 2 == 0)
```

even: org.apache.spark.rdd.RDD[Int] = FilteredRDD[2] at filter at <console>:14

Action

squares.collect

res0: Array[Int] = Array(1, 4, 9, 16, 25, 36, 49, 64, 81)

even collect

res1: Array[Int] = Array(2, 4, 6, 8)

nums.reduce (_ + _)

45

nums take 5

Array(1, 2, 3, 4, 5)

nums.count

9

K-V类型的RDD

val rdd = sc.parallelize(List(("A",1), ("B",2), ("C",3), ("A",4), ("B",5)))

val rbk = rdd.reduceByKey(_+_).collect

Array((A,5), (B,7), (C,3))

val gbk = rdd.groupByKey.collect

Array((A,ArrayBuffer(1, 4)), (B,ArrayBuffer(2, 5)), (C,ArrayBuffer(3)))

val sbk = rdd.sortByKey().collect //注意这里sortByKey的小括号不能省。

Array((A,1), (A,4), (B,2), (B,5), (C,3))

K-V类型的RDD

```
val player = sc.parallelize(List(("ACMILAN","KAKA"),("ACMILAN","BT"),("GUANGZHOU","ZHENGZHI")))

val team = sc.parallelize(List(("ACMILAN",5),("GUANGZHOU",3)))
```

player.join(team)

```
Array((GUANGZHOU,(ZHENGZHI,3)), (ACMILAN,(KAKA,5)), (ACMILAN,(BT,5)))
```

player.cogroup(team)

```
Array((GUANGZHOU,(ArrayBuffer(ZHENGZHI),ArrayBuffer(3))), (ACMILAN,(ArrayBuffer(KAKA, BT),ArrayBuffer(5))))
```

注意怎样控制reduce task的数量：

```
xx.reduceByKey(_+_,10)      xx.groupByKey(5)
```

试一下！

加载进来成为RDD

```
lines = sc.textFile("hdfs://...")
```

transformation(延迟执行)

```
errors = lines.filter(_.startsWith("ERROR"))
```

缓存RDD(延迟执行)

```
errors.persist()
```

action(把errors放进缓存)

```
mysql_errors = errors.filter(_.contains("MySQL")).count

http_errors = errors.filter(_.contains("Http")).count
```

数据本地性如何?

第一次运行时数据不在内存中，所以从HDFS上取，任务最好运行在数据所在的节点上!

文件系统本地性

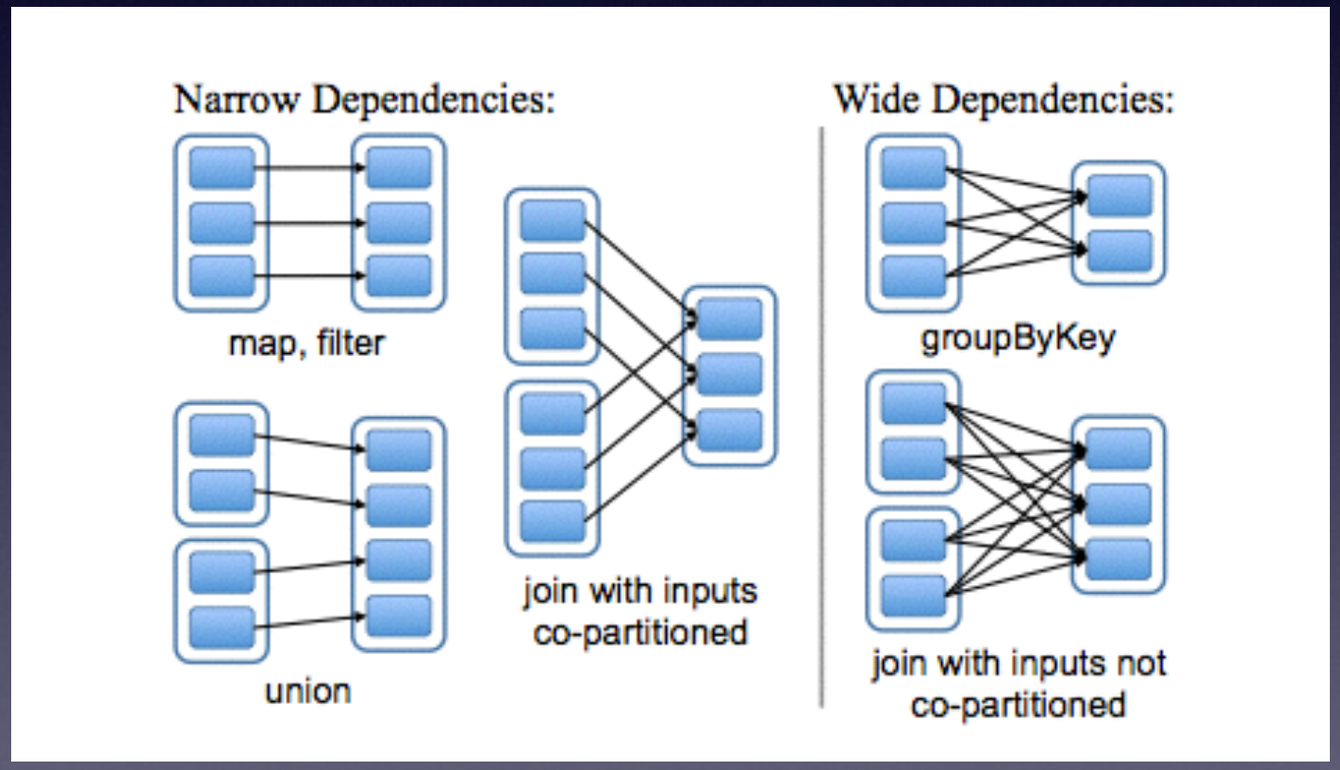
第二次运行，数据已经在内存中，所以任务最好运行在该数据所在内存的节点上。

内存本地性

万一有数据被置换出内存，则仍然从HDFS上取。

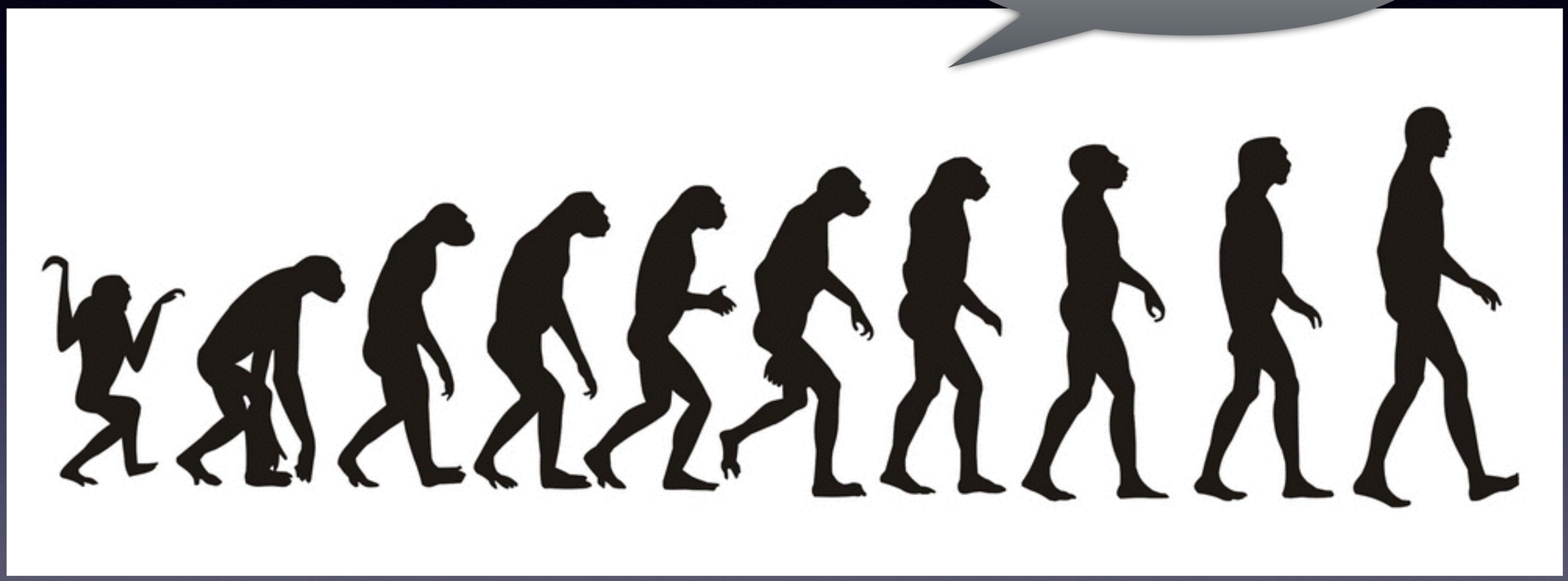
LRU置换

Dependency


Narrow Dependencies:
Wide Dependencies:
map, filter
union
join with inputs
co-partitioned
groupByKey
join with inputs not
co-partitioned

Lineage

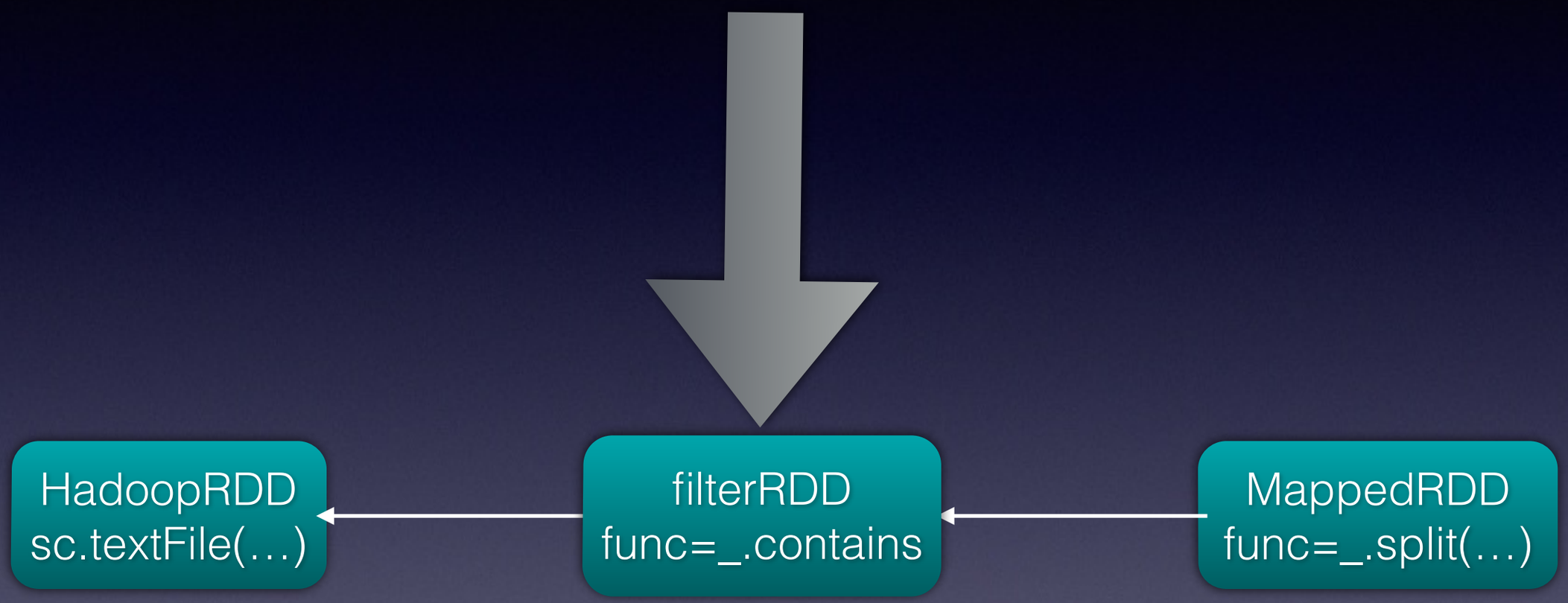
每一个都看做RDD



但是假如每次都快到进化完的时候就挂了，那岂不是每次都要从头进化？何不在中间制作个拷贝呢？！

容错

val logs = sc.textFile(…).filter(_.contains(“spark”)).map(_.split(‘\t’)(1))



每个RDD都会记录自己依赖于哪个(哪些)RDD，万一某个RDD的某些partition挂了，可以通过其它RDD并行计算迅速恢复出来。

版本选择?

- 自己编译对应版本
- pre-built版本

术语解释

术语	解释
Application	基于Spark的用户程序，包含了driver程序和集群上的executor
Driver Program	运行main函数并且新建SparkContext的程序
Cluster Manager	在集群上获取资源的外部服务(例如:standalone,Mesos,Yarn)
Worker Node	集群中任何可以运行应用代码的节点
Executor	是在一个worker node上为某应用启动的一个进程，该进程负责运行任务，并且负责将数据存在内存或者磁盘上。每个应用都有各自独立的executors
Task	被送到某个executor上的工作单元
Job	包含很多任务的并行计算，可以看做和Spark的action对应
Stage	一个Job会被拆分很多组任务，每组任务被称为Stage(就像Mapreduce分map任务和reduce任务一样)

谢谢大家