Spark最佳实践

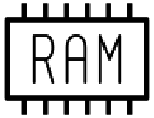
数据平台部--海量计算组
sharkdtu（涂小刚）

Spark

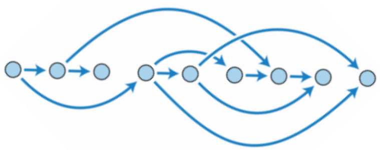
Apache Spark is a **Lightning-fast** and **General** engine for **In-Memory** large-scale data processing.

一个高效通用的内存型分布式计算框架

Why Spark?



Memory-Base



DAG

Functional Programming

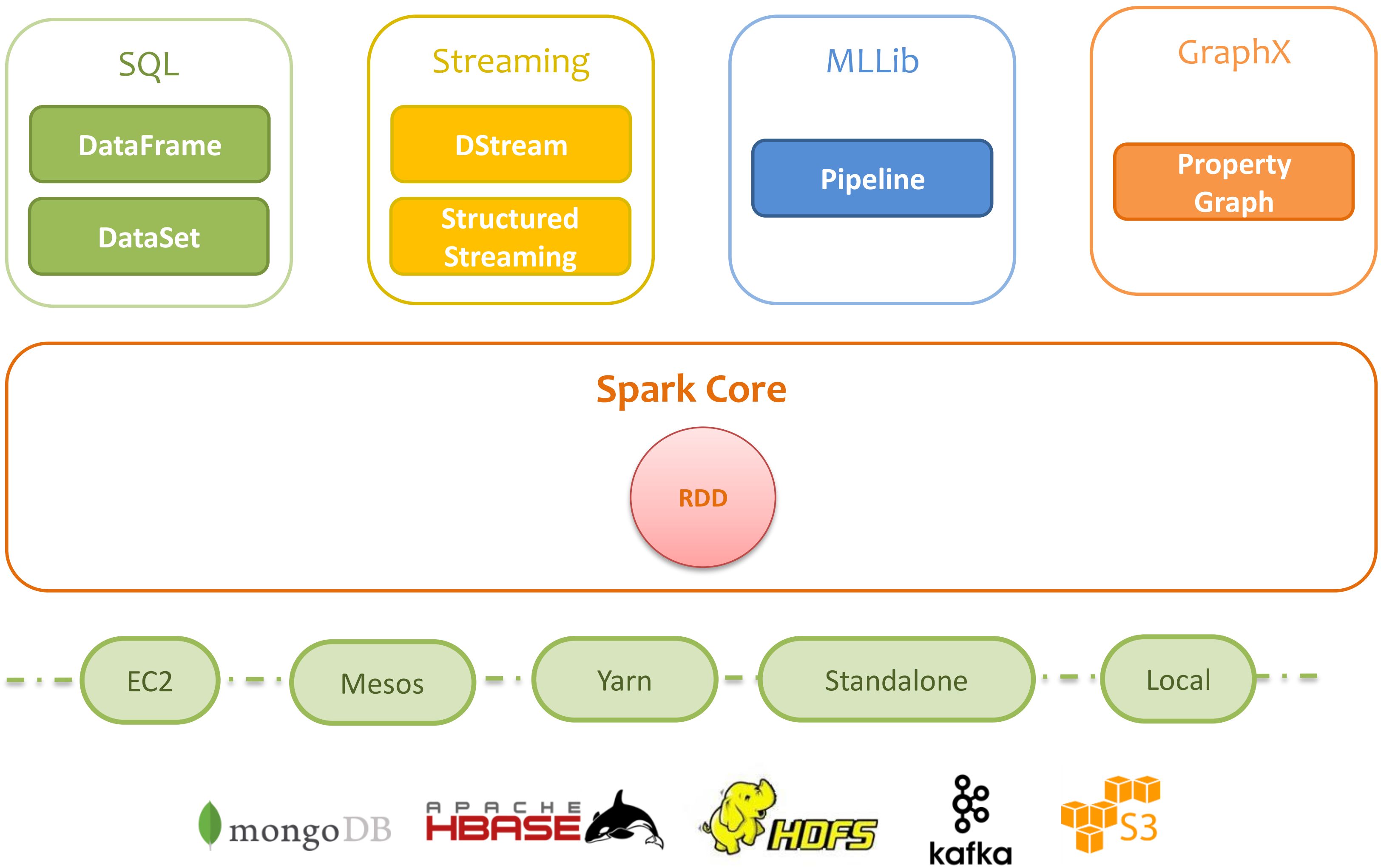
性能

迭代和容错

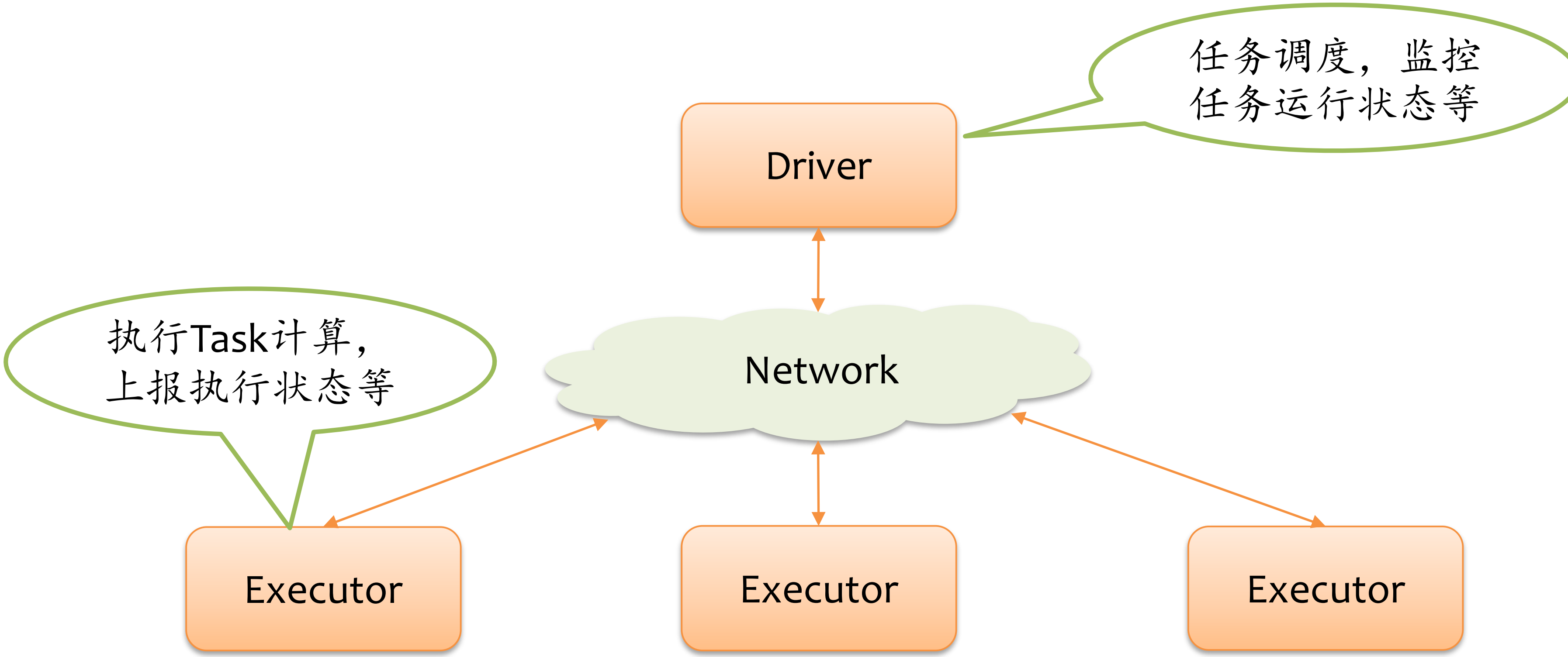
灵活

分布式计算时代的瑞士军刀

Spark体系

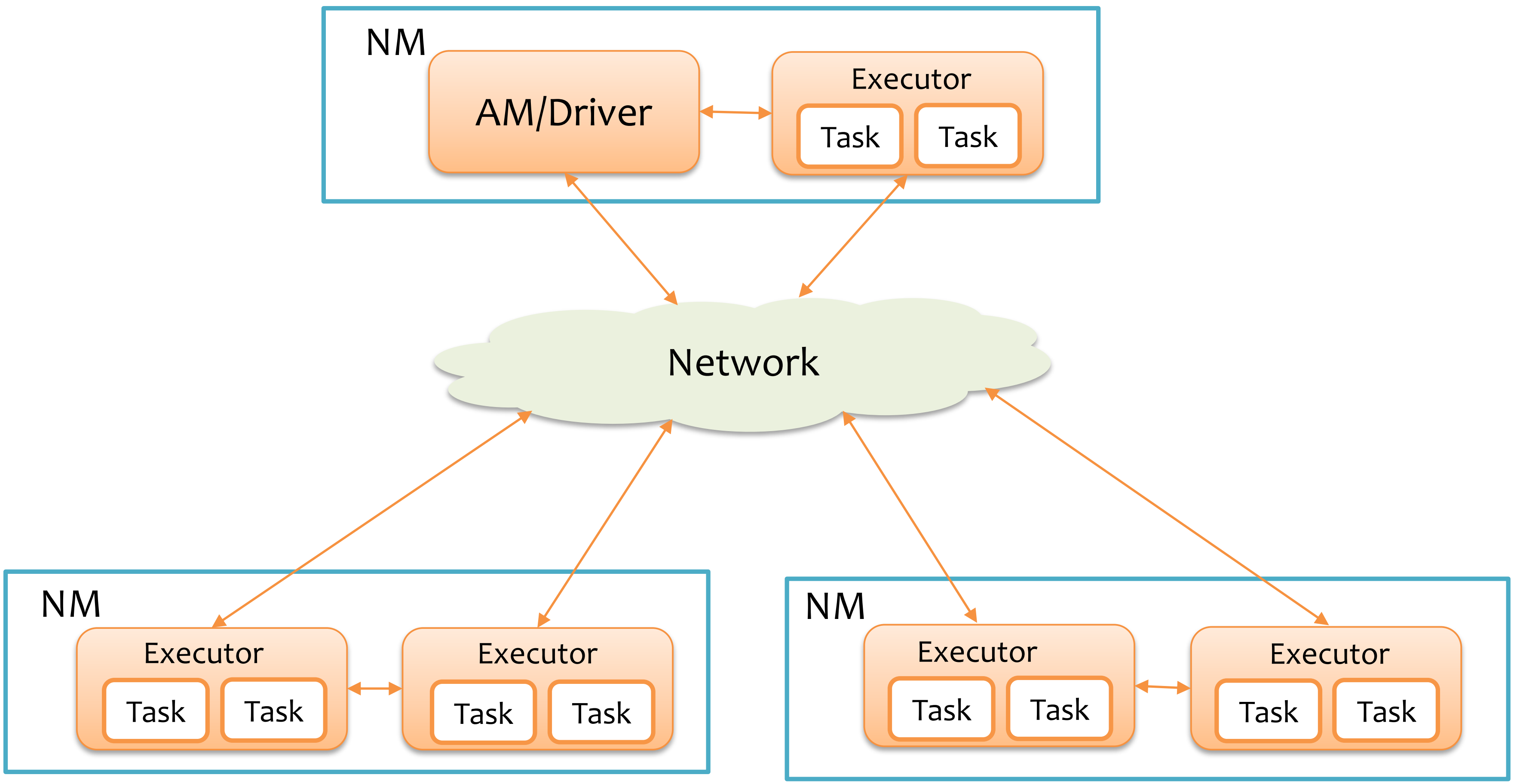

SQL
DataFrame
DataSet
Streaming
DStream
Structured Streaming
MLLib
Pipeline
GraphX
Property Graph
Spark Core
RDD
EC2
Mesos
Yarn
Standalone
Local
mongoDB
APACHE HBASE
HDFS
kafka
S3

Spark分布式运行架构



Master-Slave 主从架构

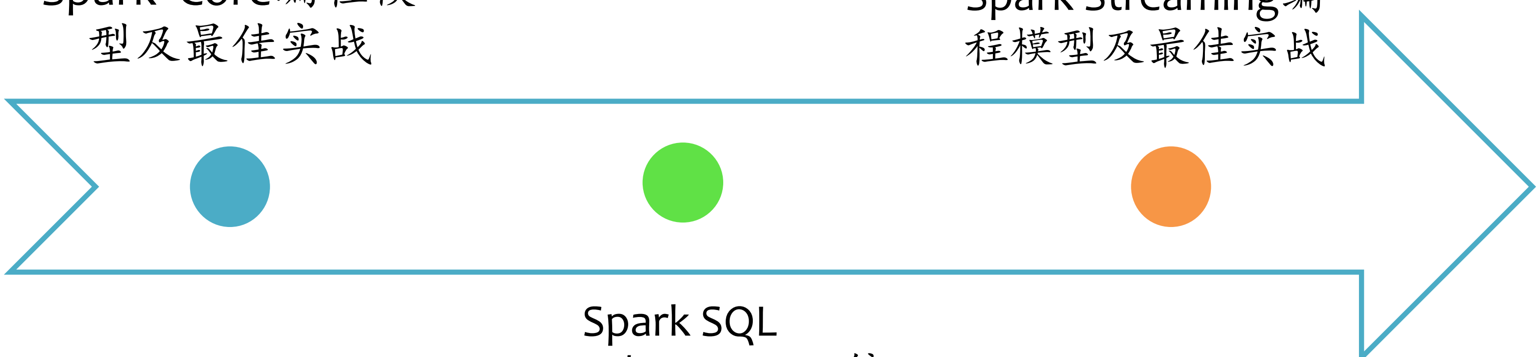
Spark on Yarn运行架构



主要内容

Spark Core编程模型及最佳实战

Spark Streaming编程模型及最佳实战

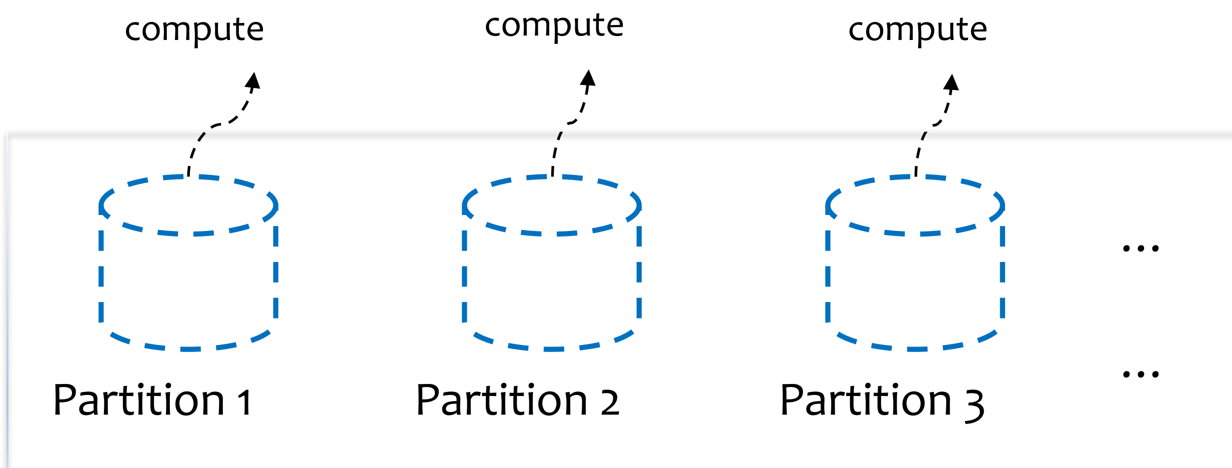


Spark SQL Dataset/DataFrame编程模型及最佳实战

Spark Core编程模型及最佳实战

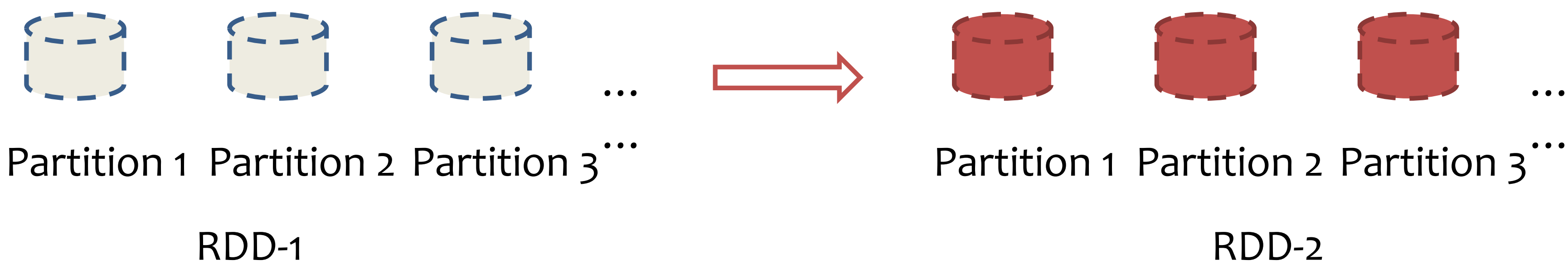
RDD

RDD简介



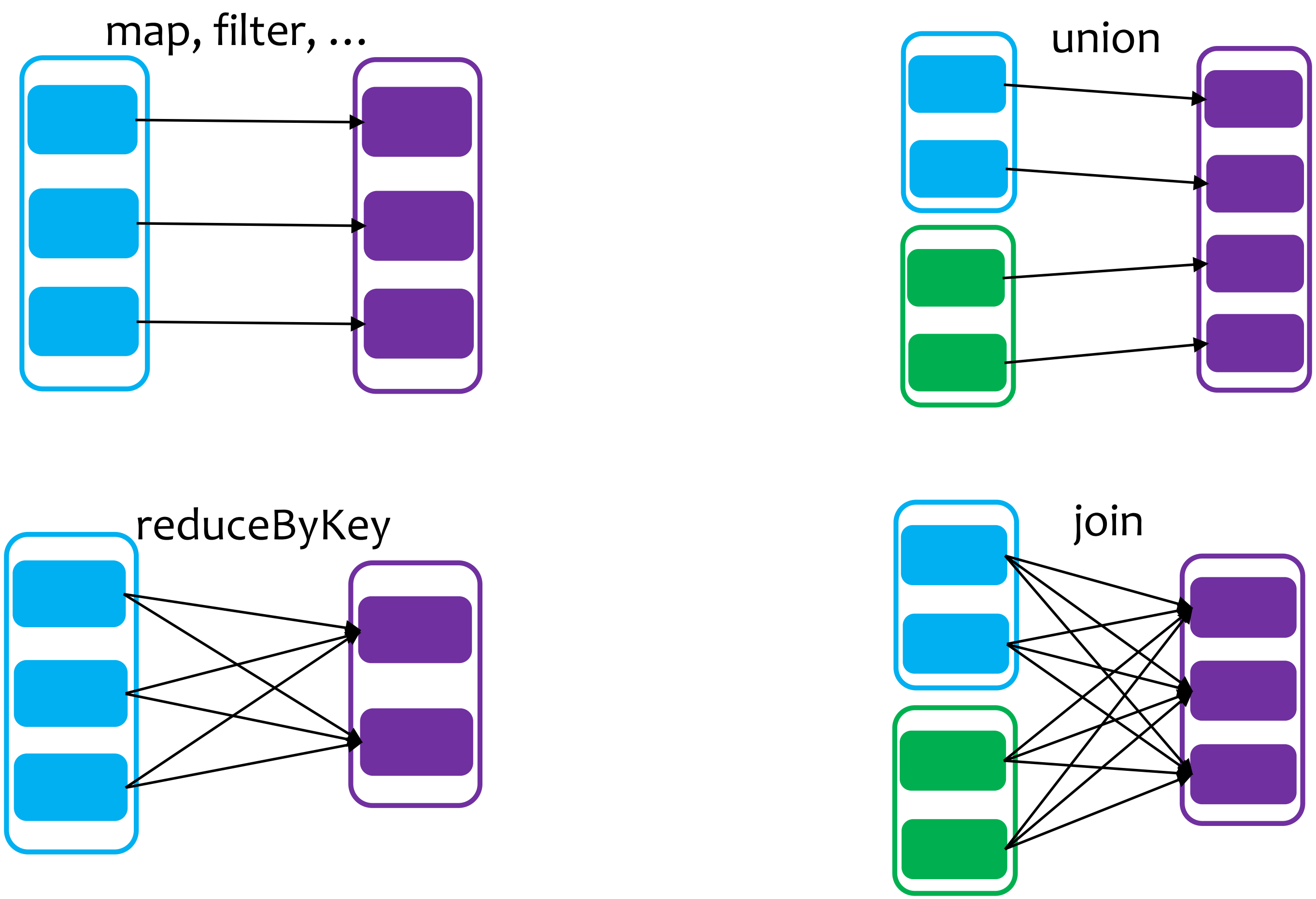
- 分区的
  - 数据是由多个分区组成
  - 可以指定分区方式(Hash...)
- 抽象的
  - 每个分区的数据不一定有物理存储
  - 只能通过compute接口获取分区数据

RDD简介



- 只读的(不可变的)
  - 一个RDD只能转变为另一个RDD

RDD简介



丰富的操作算子

RDD简介

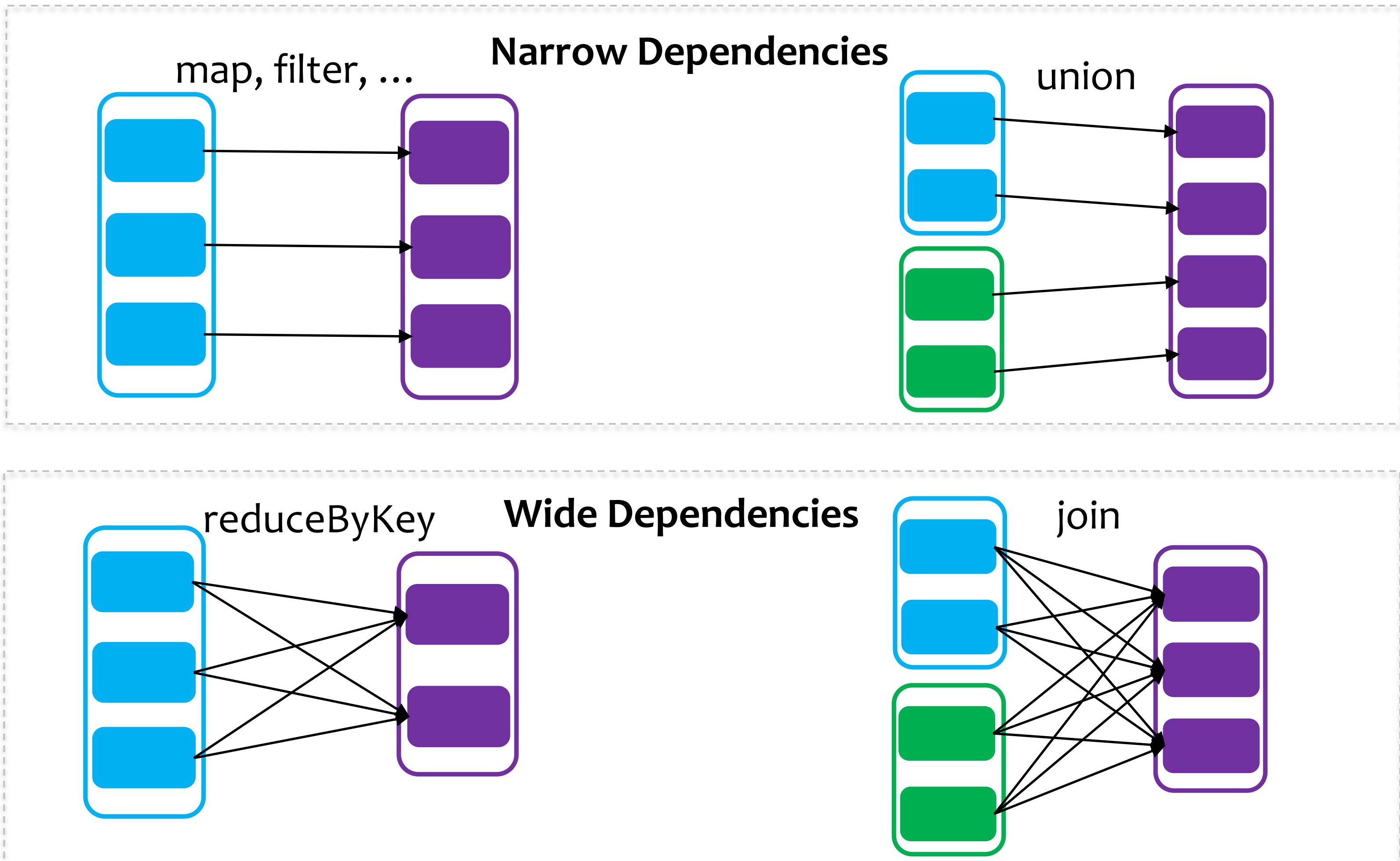
支持的算子列表

Lazy的，不会立即执行，只会记住操作

触发计算，根据RDD记住的操作依次计算

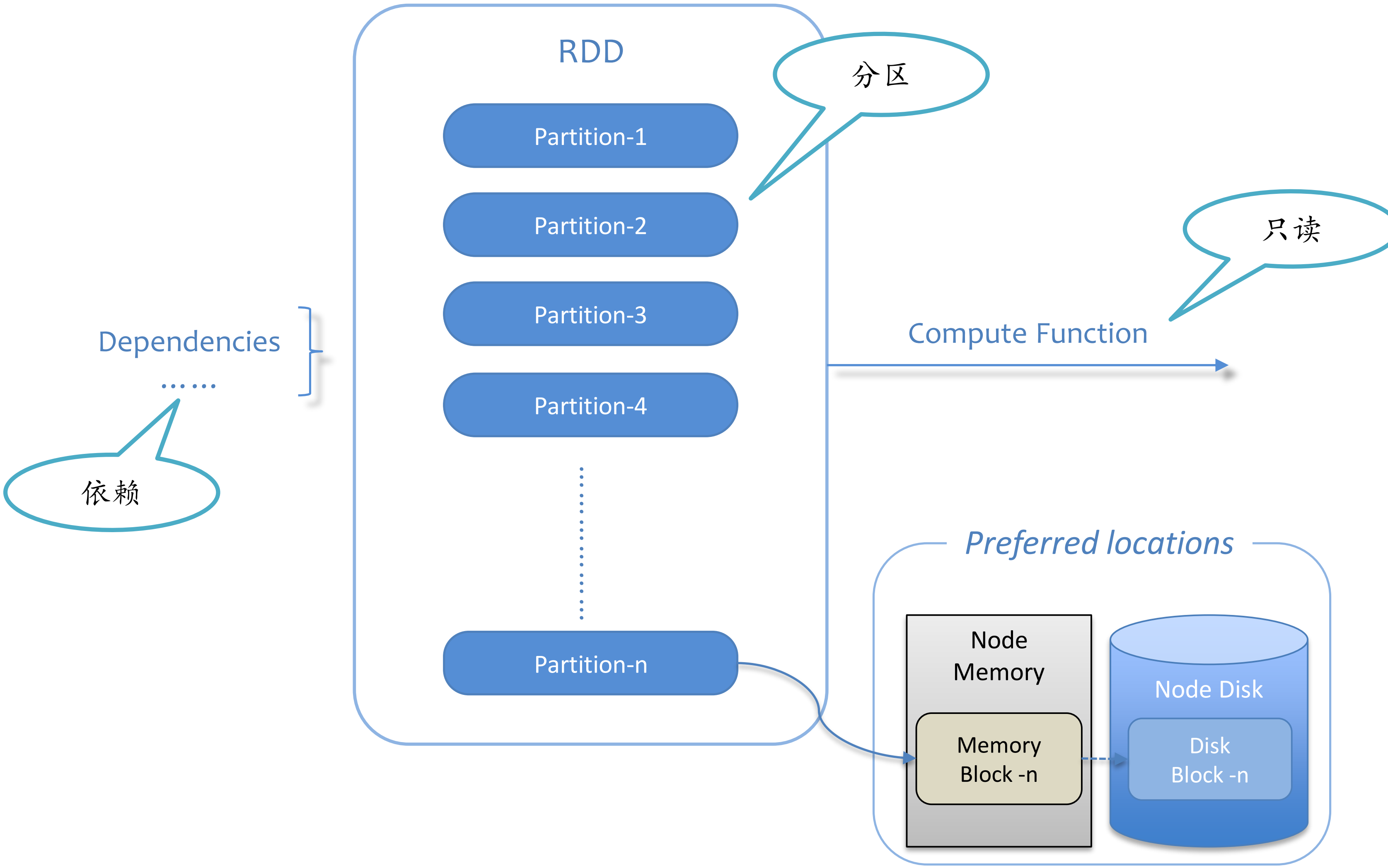
Transformations	$map(f : T \Rightarrow U)$	: RDD[T] ⇒ RDD[U]
	$filter(f : T \Rightarrow Bool)$	: RDD[T] ⇒ RDD[T]
	$flatMap(f : T \Rightarrow Seq[U])$	: RDD[T] ⇒ RDD[U]
	$sample(fraction : Float)$	: RDD[T] ⇒ RDD[T] (Deterministic sampling)
	$groupByKey()$	: RDD[(K, V)] ⇒ RDD[(K, Seq[V])]
	$reduceByKey(f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

RDD简介

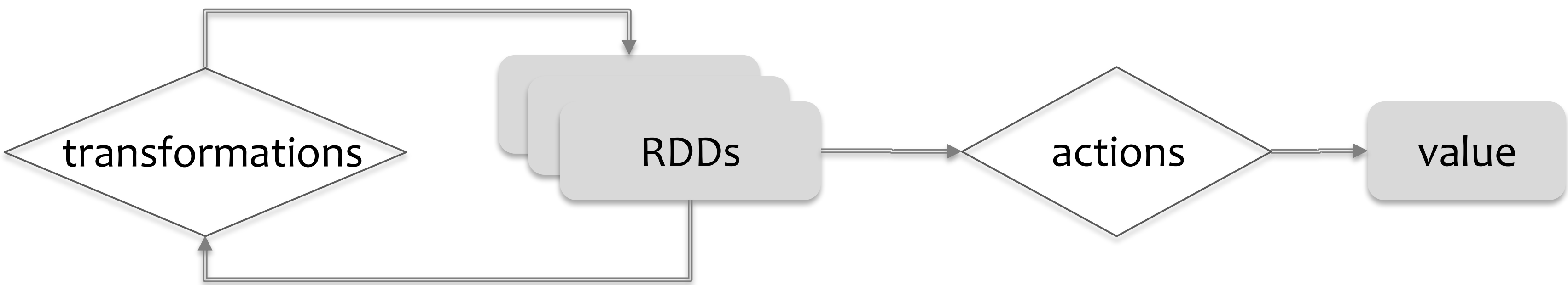


RDD之间存在依赖关系

RDD简介


RDD
Partition-1
Partition-2
Partition-3
Partition-4
Partition-n
分区
只读
Compute Function
Dependencies
……
依赖
Preferred locations
Node
Memory
Memory
Block -n
Node Disk
Disk
Block -n

RDD编程模型



- 编程方式：定义RDD之间的转换，通过action方法得到结果
- 编程语言：支持Scala, Java, Python, R等
- 内存计算：支持内存等多种cache，保存中间结果，避免重计算
- 容错机制：RDD血缘关系恢复，支持checkpoint

RDD编程模型

举例：WordCount简单实例(scala)

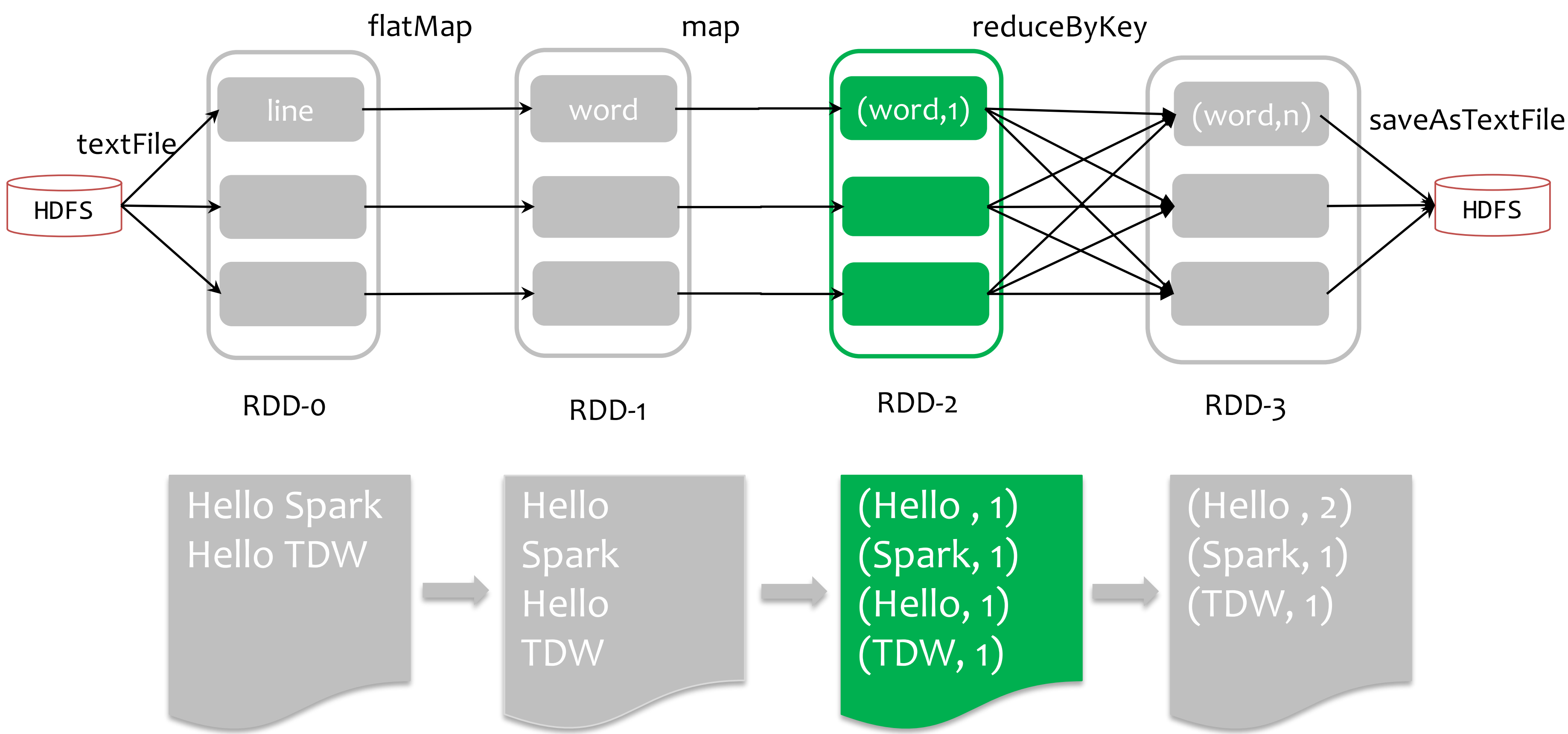
```scala
import org.apache.spark.{SparkContext, SparkConf}

object WordCount {
  def main(args: Array[String]) {
    if (args.length < 2) {
      System.err.println("Usage: WordCount <inputfile> <outputfile>")
      System.exit(1)
    }
    val conf = new SparkConf().setAppName("WordCount")
    val sc = new SparkContext(conf)
    val result = sc.textFile(args(0))
      .flatMap(line => line.split("\\s+"))
      .map(word => (word, 1))
      .reduceByKey(_ + _)
    result.saveAsTextFile(args(1))
    sc.stop()
  }
}
```

transformations

action

RDD编程模型

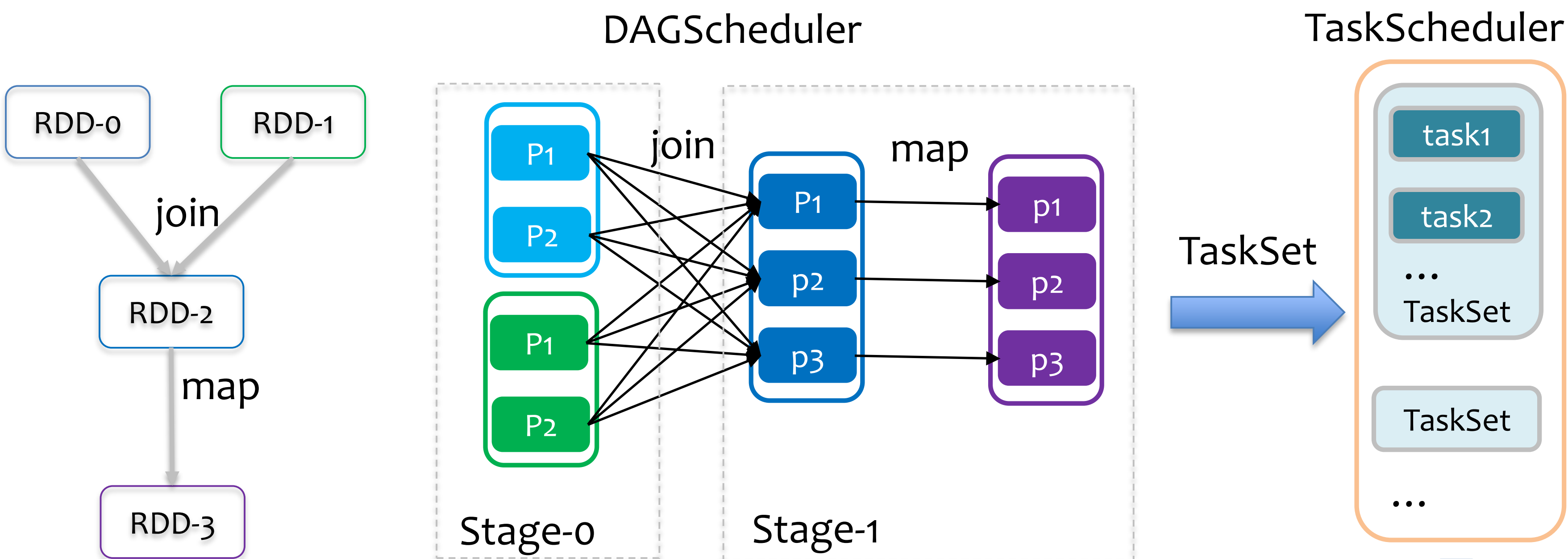
WordCount RDD转换关系



Spark Core编程模型及最佳实战

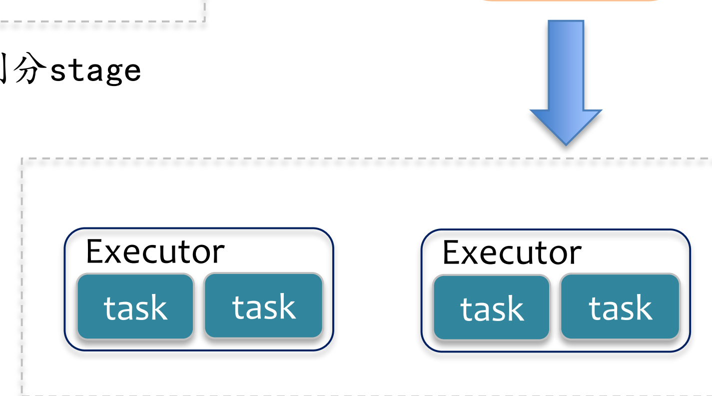
Scheduler

Spark任务调度



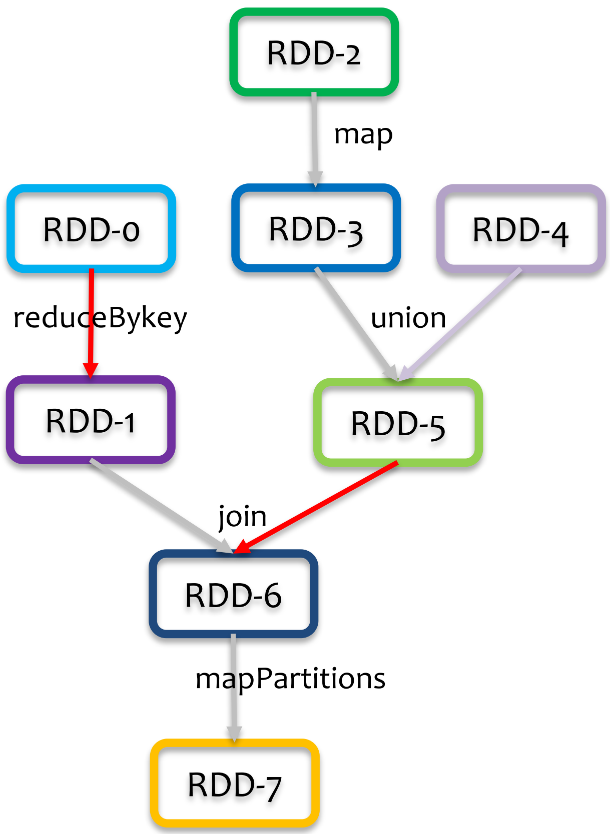
stage级的调度，以Shuffle为界划分stage

- Job：以action方法为界
- Stage：Job的子集，以Shuffle为界
- Task：Stage的子集，以分区数界定

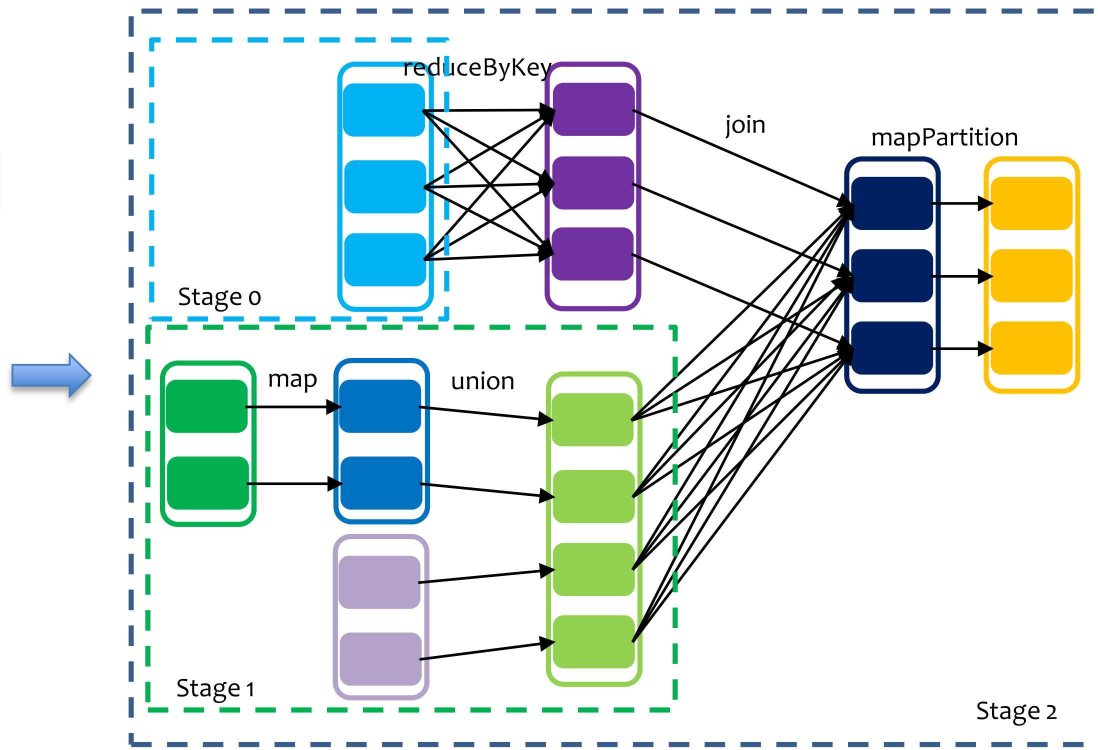


task级的调度，分区数为多少，则有多少个task

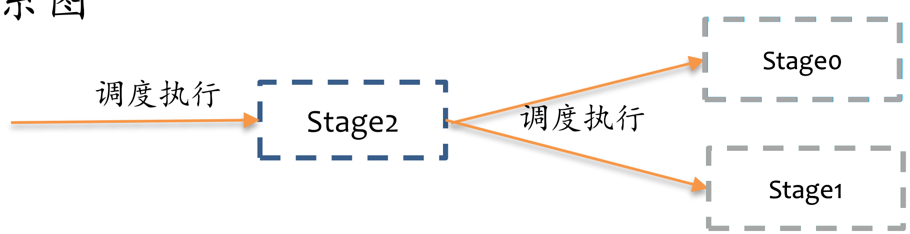
DAG 划分



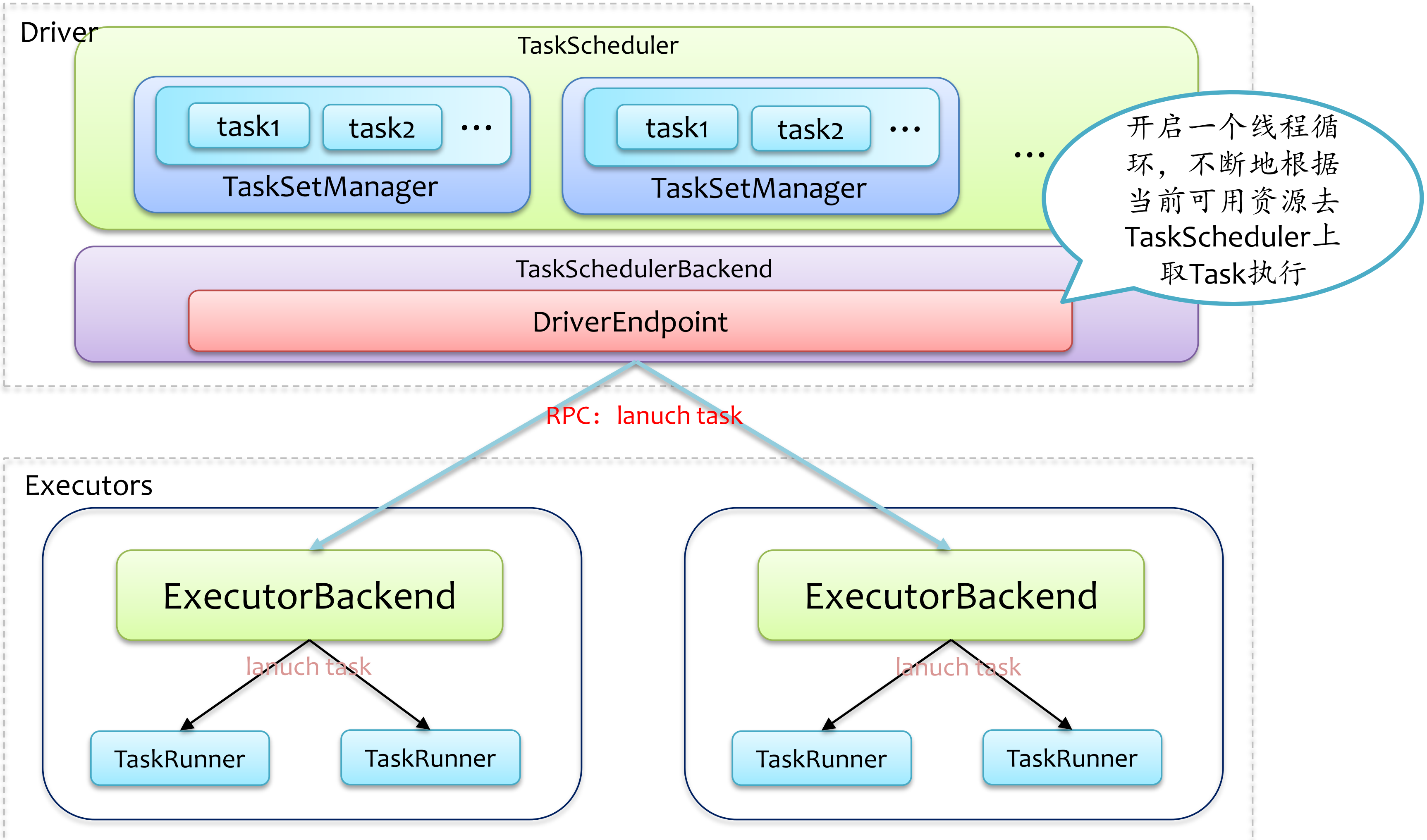
RDD依赖关系图



DAG：以Shuffle为界，划分Stage单元，生成有向无环图

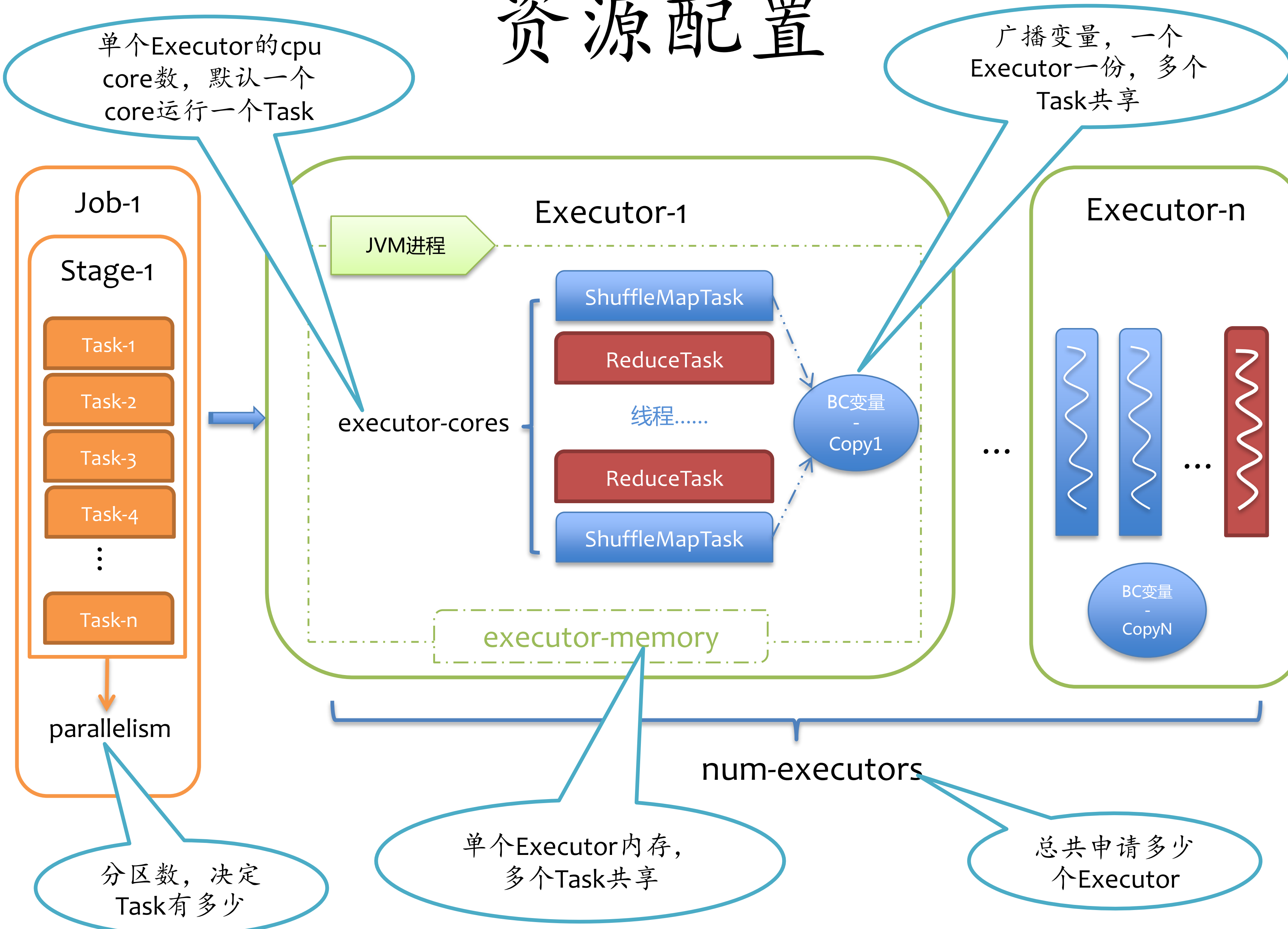


Task调度



Driver
TaskScheduler
task1
task2
…
TaskSetManager
task1
task2
…
TaskSetManager
…
开启一个线程循环，不断地根据当前可用资源去TaskScheduler上取Task执行
TaskSchedulerBackend
DriverEndpoint
RPC：lanuch task
Executors
ExecutorBackend
lanuch task
TaskRunner
TaskRunner
ExecutorBackend
lanuch task
TaskRunner
TaskRunner

资源配置



单个Executor的cpu core数，默认一个core运行一个Task
广播变量，一个Executor一份，多个Task共享
Job-1
Stage-1
Task-1
Task-2
Task-3
Task-4
Task-n
parallelism
Executor-1
JVM进程
executor-cores
ShuffleMapTask
ReduceTask
线程……
ReduceTask
ShuffleMapTask
BC变量 - Copy1
executor-memory
Executor-n
BC变量 - CopyN
num-executors
分区数，决定Task有多少
单个Executor内存，多个Task共享
总共申请多少个Executor

资源配置

- 太大的内存(executor-memory > 15G)
  - 资源浪费，影响其他业务
  - 限制单个Executor的内存不超过14G
  - 大内存需求单独找运维童鞋

- 太大的cpu核数(executor-cores > 5)
  - 丢弃了并发的优势 *例如：num-executor=10 executor-cores=10*
  - 申请等待时间和风险
  - 建议单个executor的core为2~3个

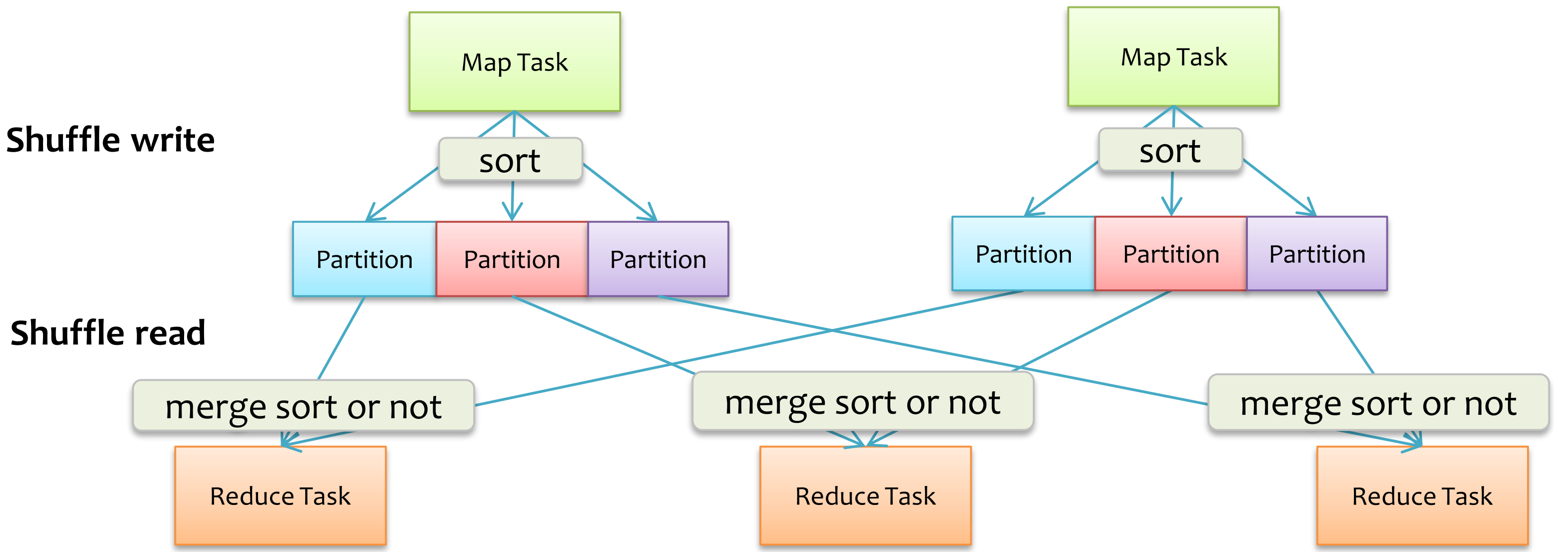
- 太多的executor(num-executor > 500)
  - 申请等待时间和风险
  - 每个executor都是独立的JVM，网络IO成本
  - 丢弃了多任务的优势 *例如：num-executor=1000 executor-cores=1*

- 太多的分区(parallelism > executor * cores * (3-5))
  - 任务过细，轮数太多
  - 增加driver的维护压力

Spark Core编程模型及最佳实战

Shuffle

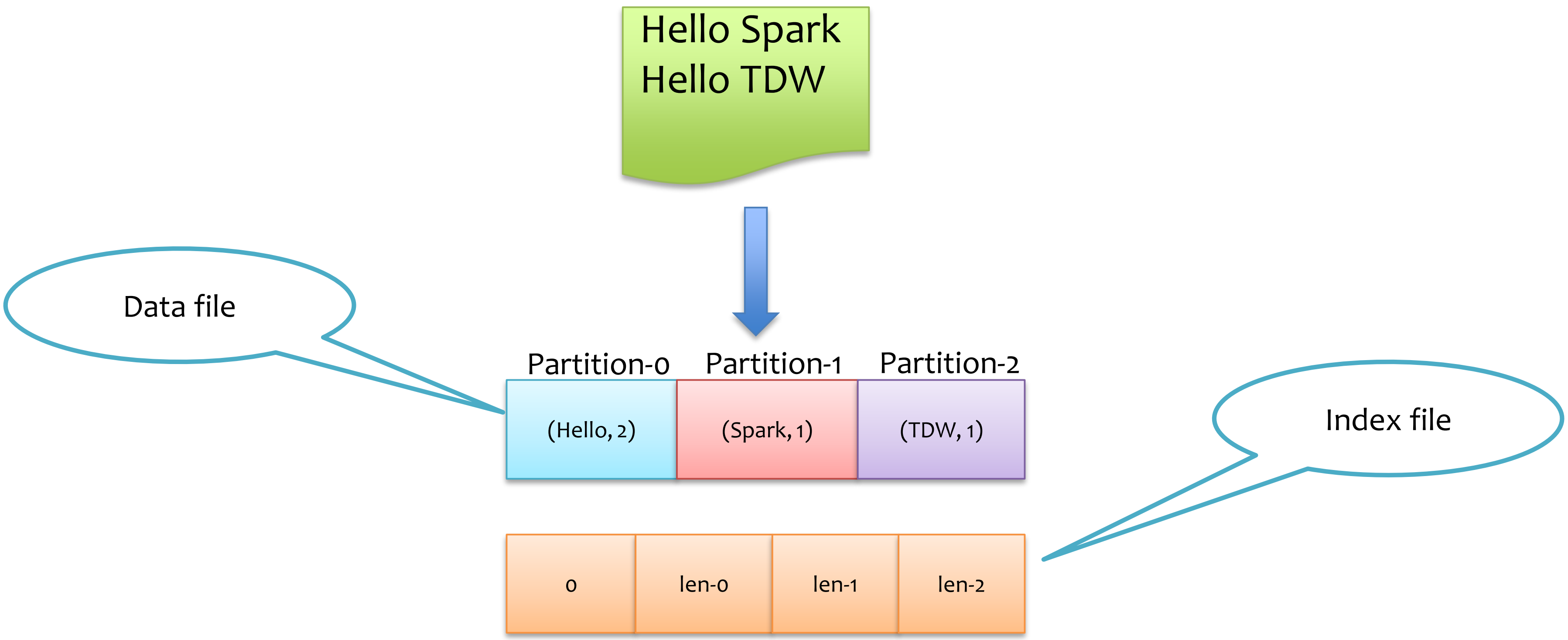
Spark Shuffle内部原理



2 Phrase： Write & Read

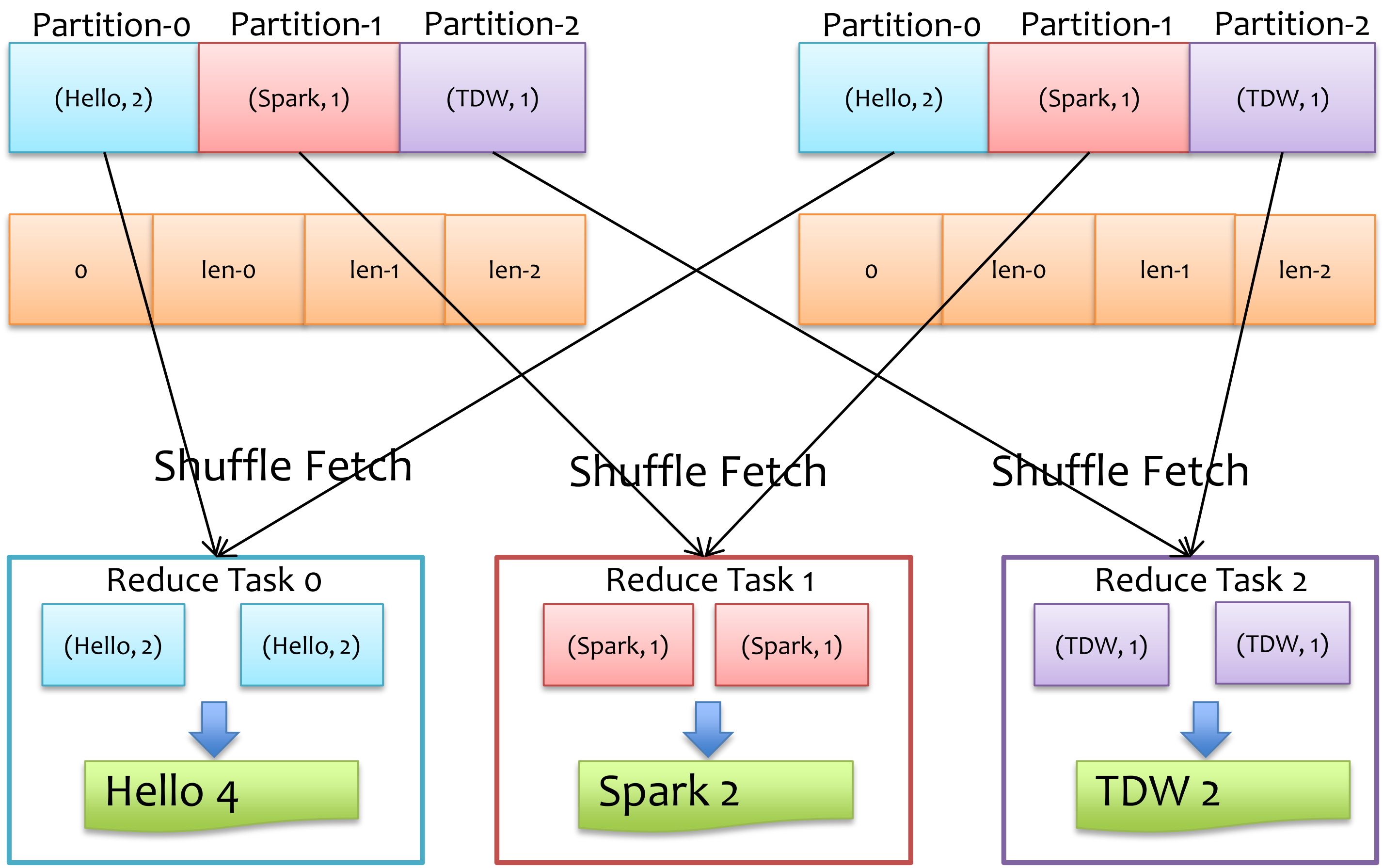
- Shuffle is Expensive
  - 序列化： CPU
  - 跨机器： Network IO
  - 读写文件: Disk IO
- Shuffle操作
  - repartition
  - *ByKey
  - Join & cogroup

Spark Shuffle实例



Shuffle Write: Map Tasks, 分区为3

Spark Shuffle实例



Shuffle Read: Reduce Tasks

Shuffle调优

- 优化数据结构
  - 尽可能使用原生类型(Int, Long, Double等)
  - 尽可能使用对象数组以及原生类型数组以替代Java或者Scala集合类
  - 尽可能避免采用嵌套数据结构来保存小对象
- 数据落地时，partition数不宜过多
  - 在保存数据到hdfs/tdw前，尽量控制partition数，避免落地后小文件较多影响后续加载，落地前调用 rdd.coalesce(num_partition)减少partition数
- 主动Shuffle-repartition
  - 如果分区数较少，可加大分区，将任务细分
  - 提高后续分布式运行的速度
- 调整shuffle read并发度
  - 内存紧俏时，减少shuffle read并发，内存充足时，增加shuffle read并发

```
spark.reducer.maxSizeInFlight=48m  // default: 48m
```

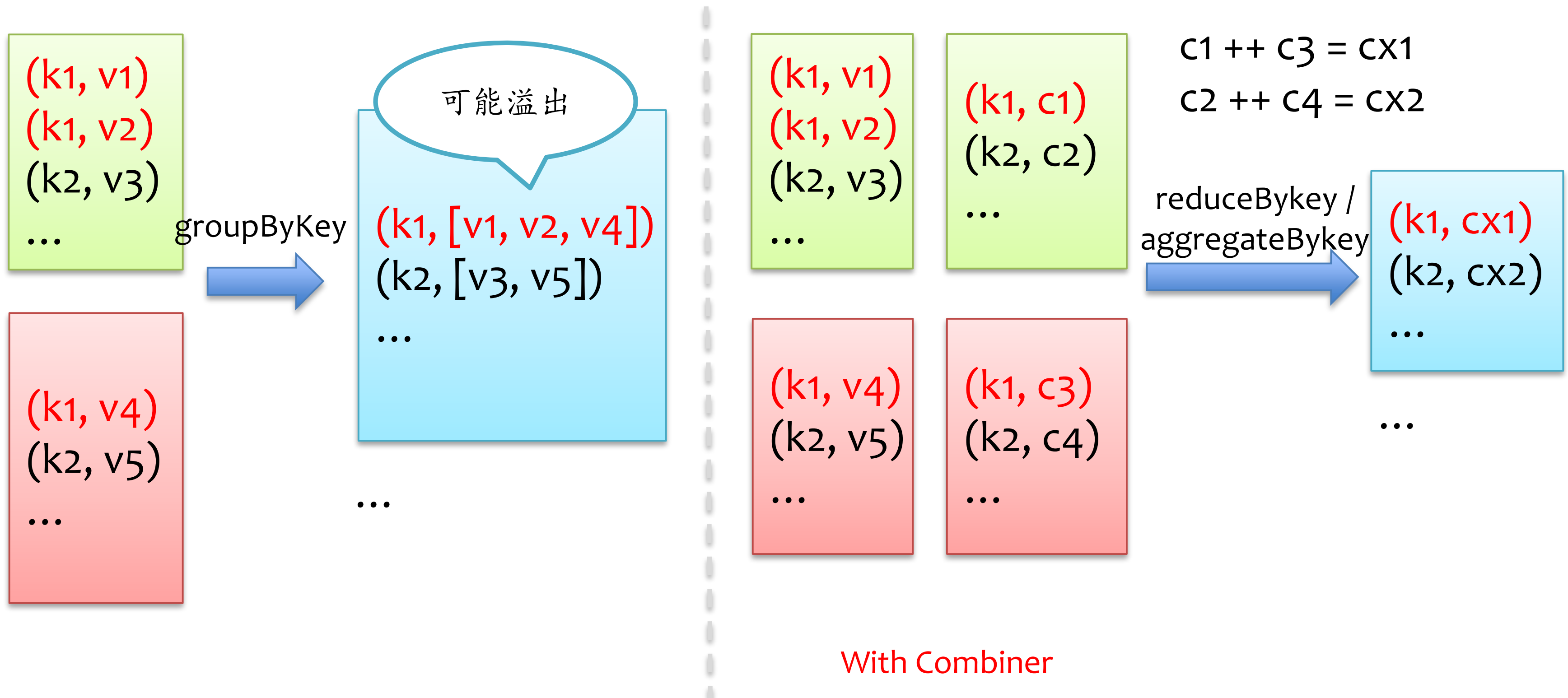
Shuffle调优

- 多表Join
  - 如果要Join多个RDD，请使用cogroup

```
// 多次shuffle，多一个join多一次shuffle
val joinedRdd = rdd1.join(rdd2 ).join(rdd3)

// 使用cogroup join三个rdd
val rdd2Data = rdd1.cogroup(rdd2, rdd3)

// 使用cogroup join四个rdd
val rdd2Data = rdd1.cogroup(rdd2, rdd3, rdd4)
```

Shuffle调优

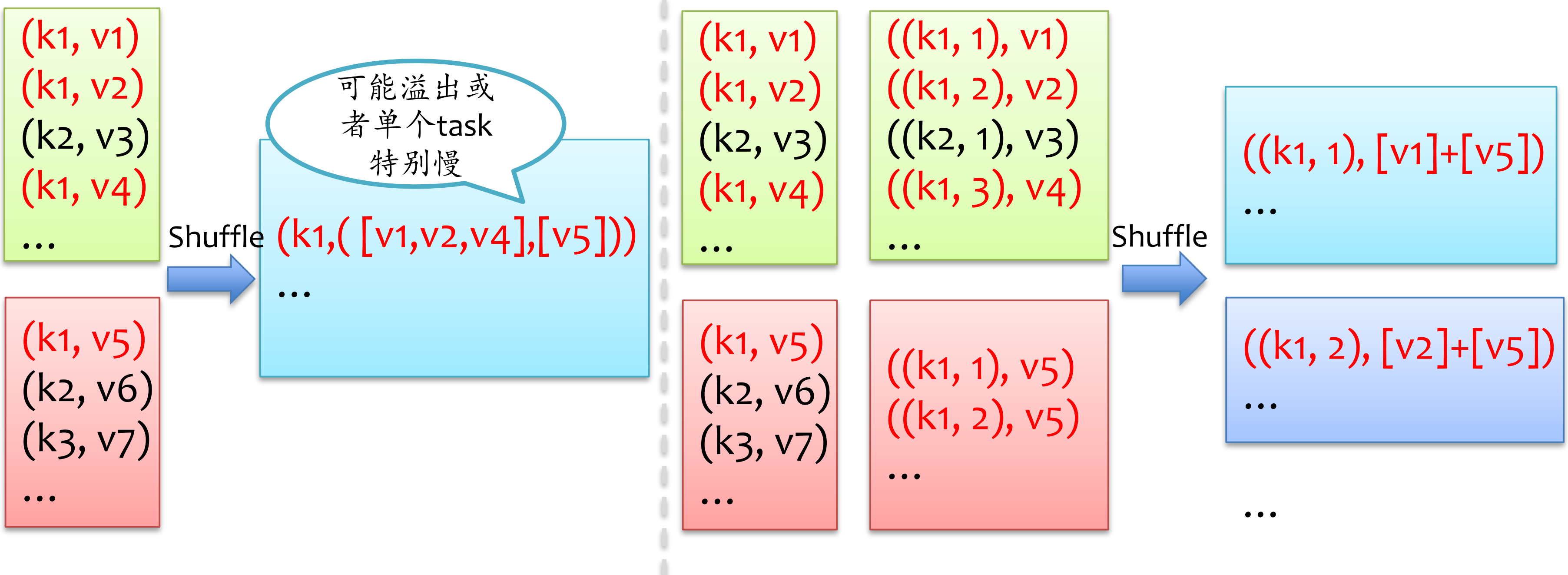
- 避免使用groupByKey做聚合操作
  - 特征：groupByKey.mapValues(_.sum)
  - 改进：rdd.reduceByKey(_ + _)



Shuffle调优

- 数据倾斜处理--join
  - 对key进行分桶　　缺点：小表会膨胀，整体运行较慢

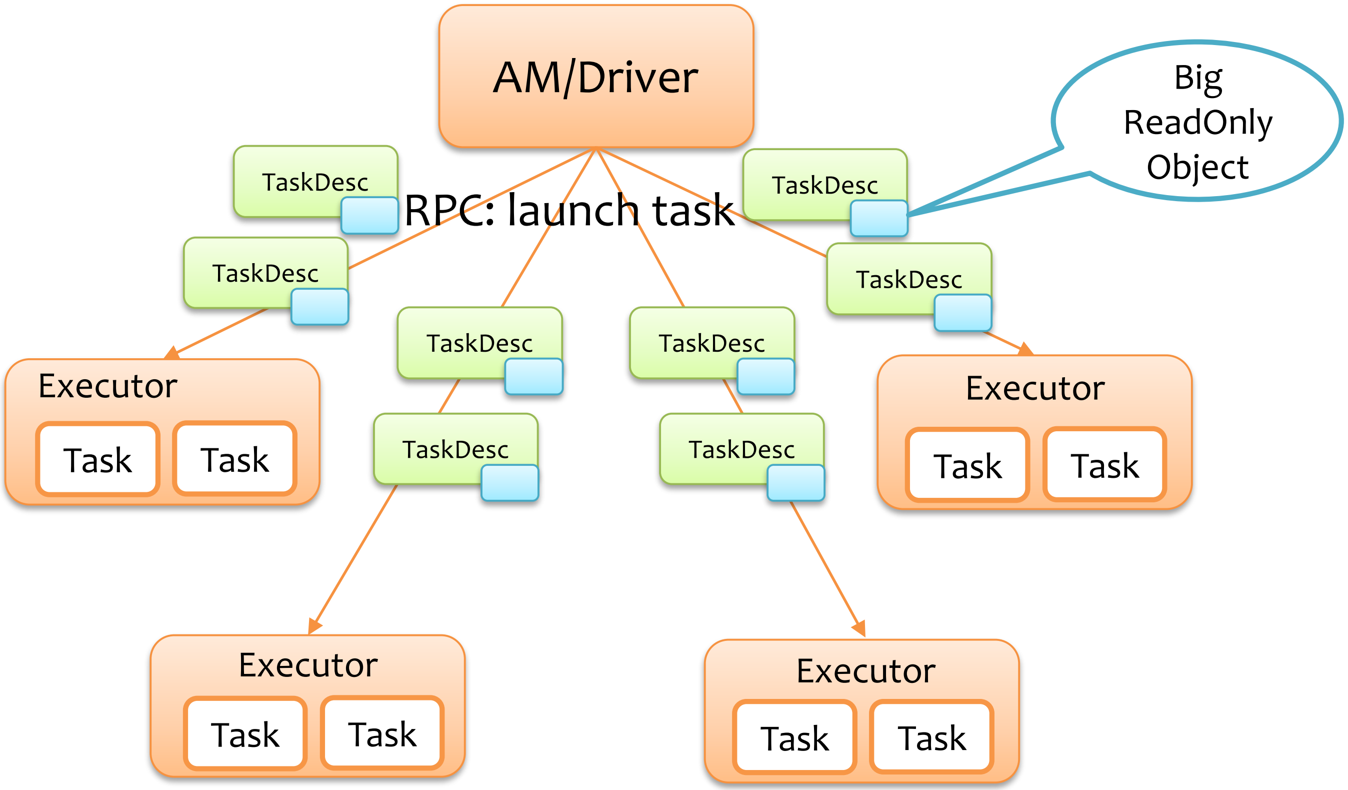
```
leftRdd.map( x => ((x._1, Random.nextInt(100)), x._2)).join(
  rightRdd.flatMap(x => for (i <- 0 until 100) yield((x._1, i), x._2) )
)  // leftRdd is skewed
```



Spark Core编程模型及最佳实战

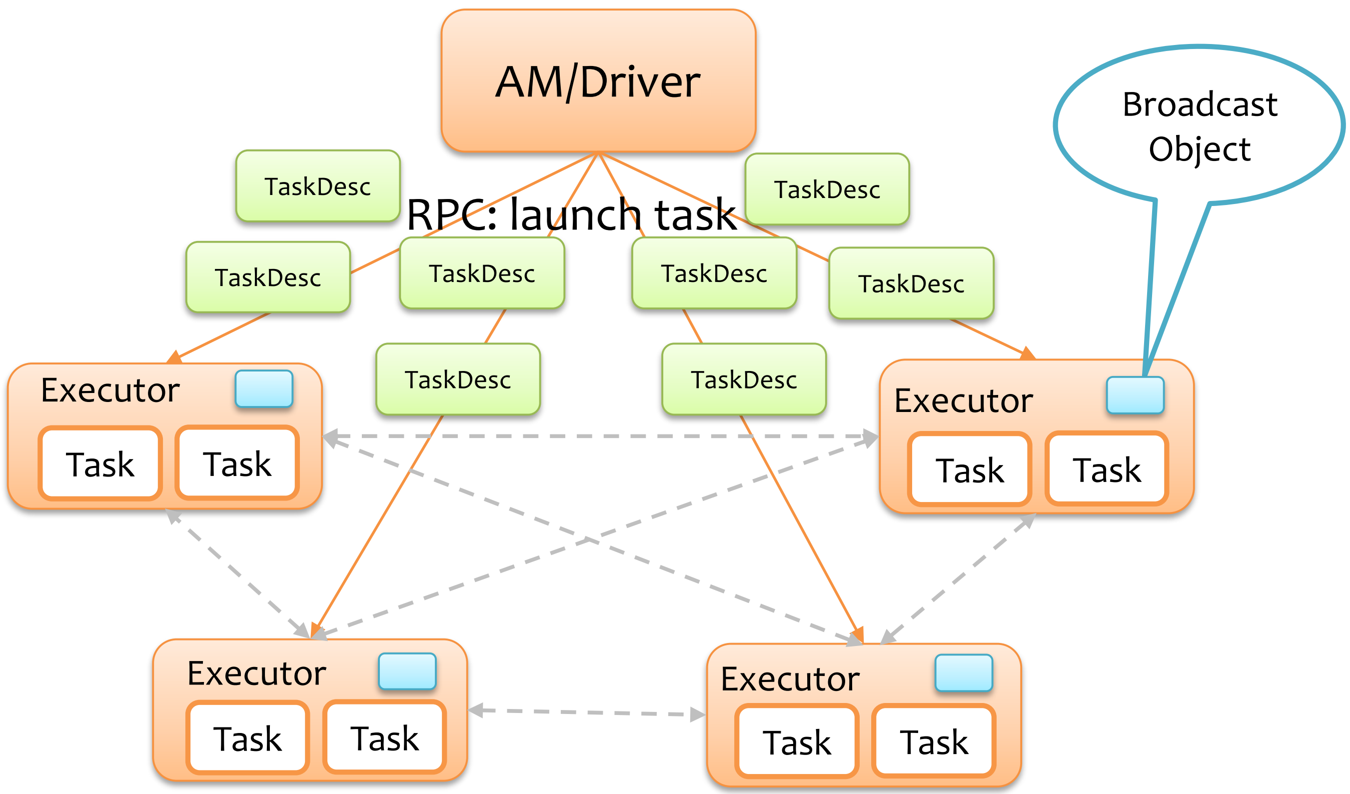
Broadcast

Broadcast基本原理



No broadcast: 每次启动任务都要传输大对象

Broadcast基本原理

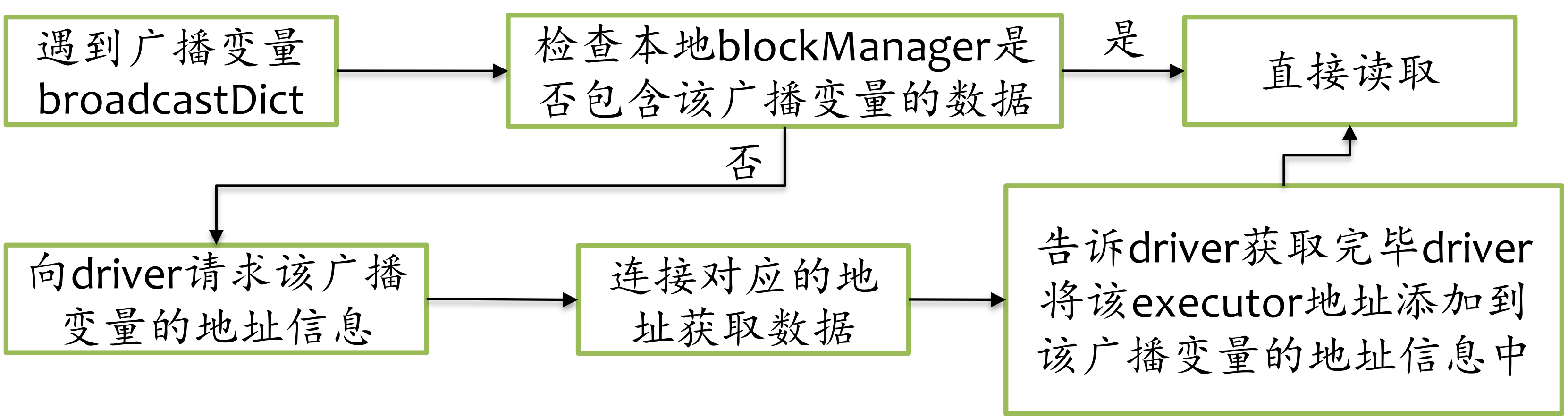


Broadcast: 每个executor传输一次大对象，并使用torrent加快网络传输

Broadcast举例

- 广播大对象
  - 如果任务逻辑中会使用比较大的对象(大于10M)，例如静态查找表，则考虑将其变成广播变量

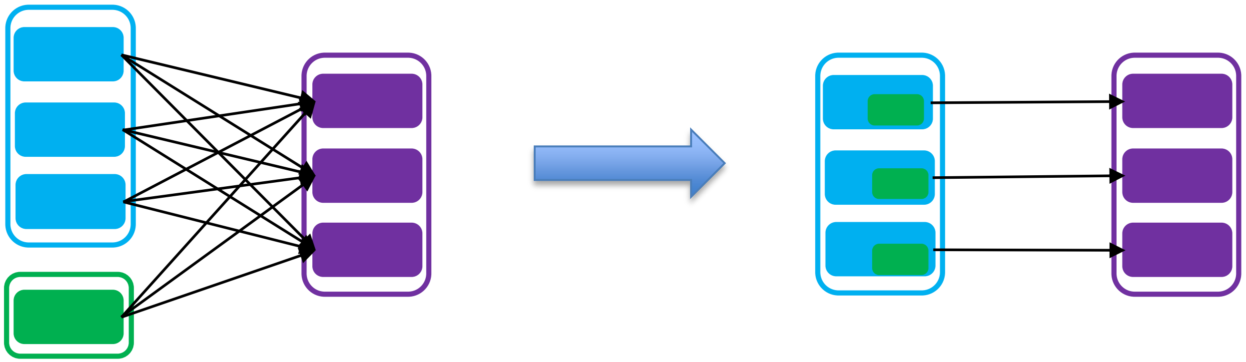
```
val dict: HashMap[Int, String] = ...
val broadcastDict = sc.broadcast(dict)
rdd.map { x =>
  val dict = broadcastDict .value
  ...
}
```



实现MapJoin

- Broadcast+map - 大表join小表
  - 避免shuffle

```
// 传统的join操作会导致shuffle操作
val joinedRdd = rdd1.join(rdd2)

// 使用Broadcast将一个数据量较小的RDD作为广播变量
val rdd2Data = rdd2.collectAsMap()
val rdd2DataBroadcast = sc.broadcast(rdd2Data)
val joinedRdd = rdd1.map{ x =>
  val rdd2Data = rdd2DataBroadcast.value
  …
}
```



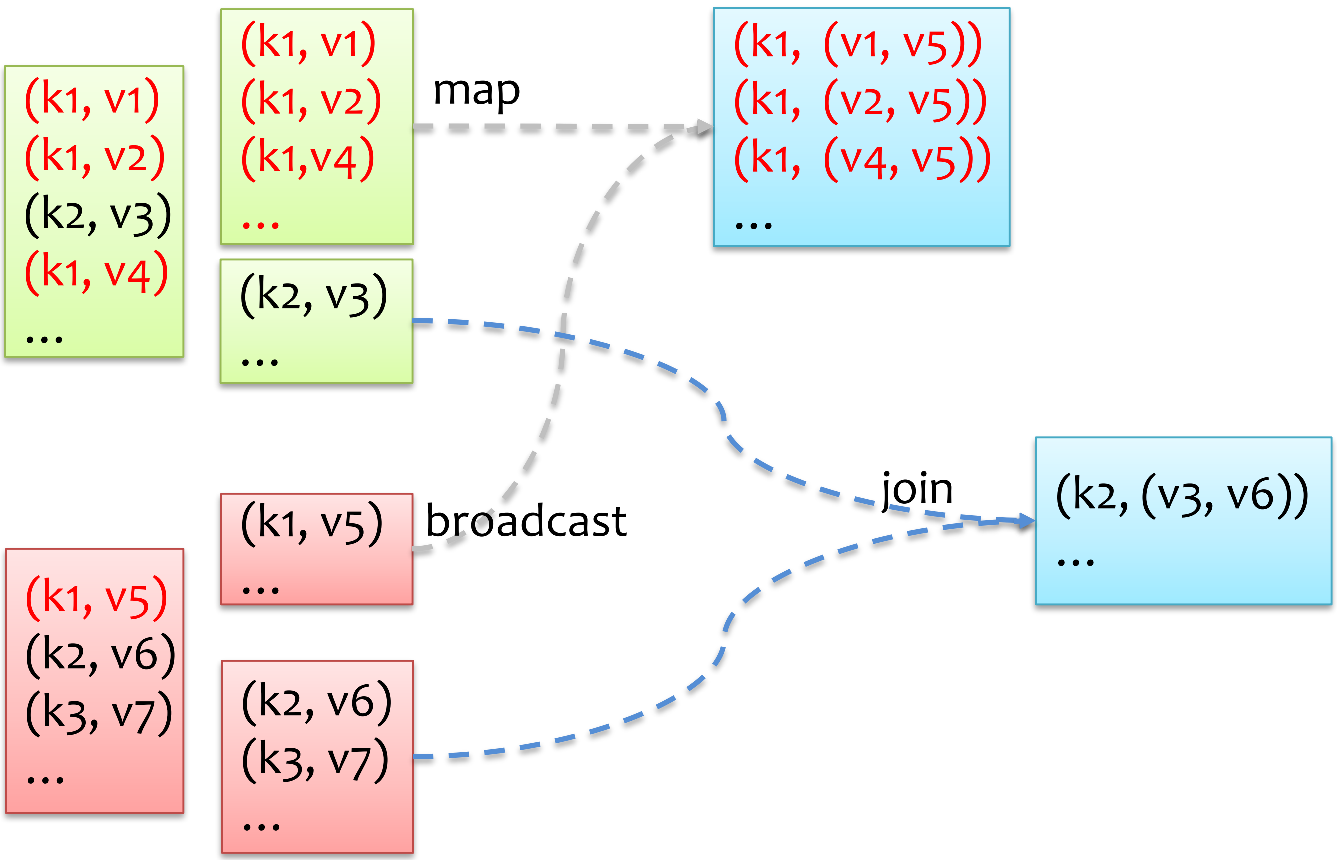
MapJoin处理数据倾斜

- 数据倾斜处理--join
  - 将rdd分割成两部分进行join

```
// leftRdd is skewed
val cnts = leftRdd.mapValues(_ => 1L).reduceByKey(_ + _).persist(storageLevel)
val skewedCnts = cnts.filter(_._2 >= 100000).collectAsMap()

val bcSkewedCnts = sc.broadcast(skewedCnts)
val leftRddSkewedPart = leftRdd.filter{ x => bcSkewedCnts.value.contains(x._1) }
val leftRddNoSkewedPart = leftRdd.filter{ x => !bcSkewedCnts.value.contains(x._1) }

val rightRddBcPart = rightRdd.filter{ x =>
  bcSkewedCnts.value.contains(x._1)
}.collectAsMap()
val rightRddNoBcPart = rightRdd.filter{ x => !bcSkewedCnts.value.contains(x._1) }
```

MapJoin处理数据倾斜

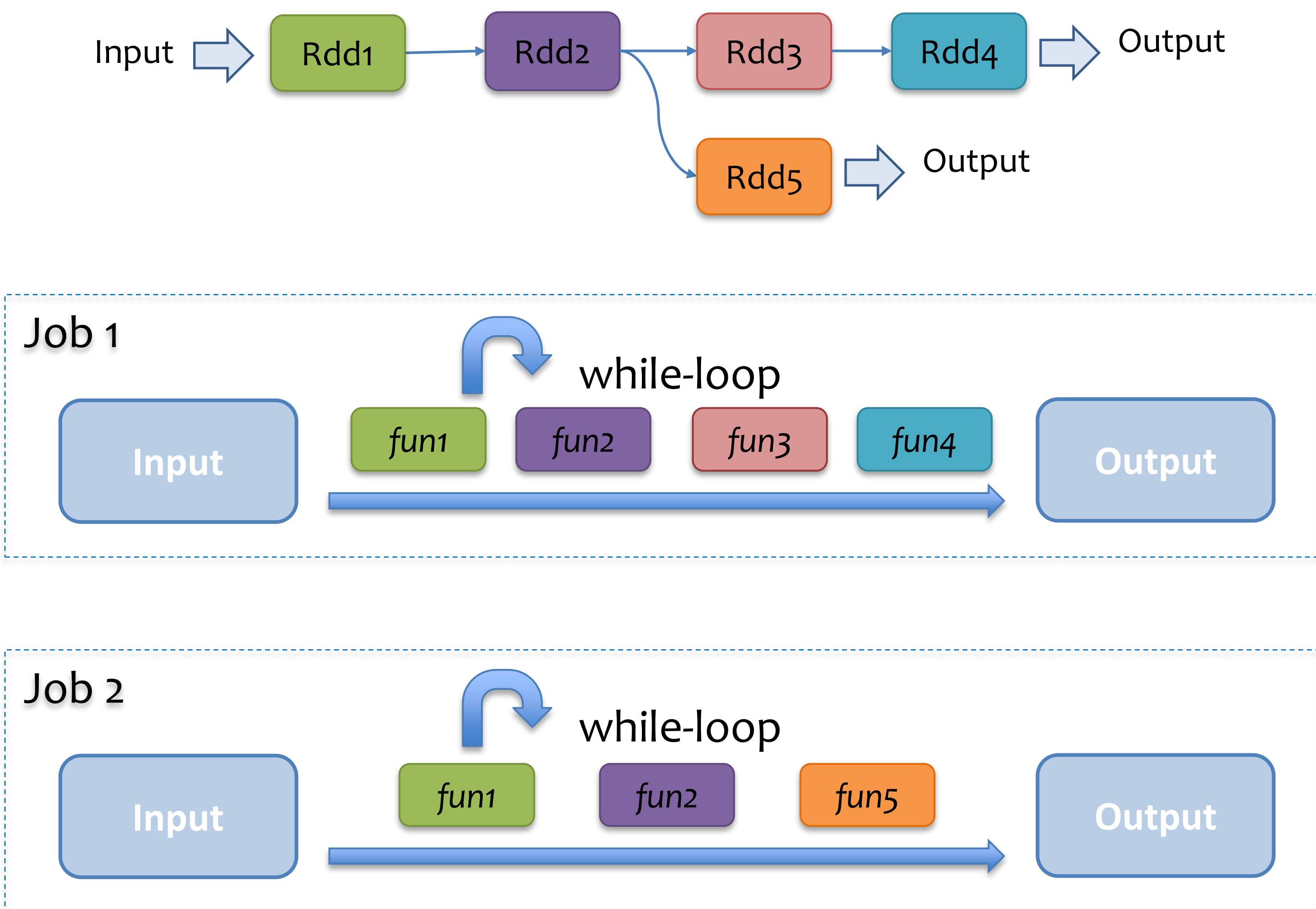
- 数据倾斜处理--join
  - 将rdd分割成两部分进行join



Spark Core编程模型及最佳实战

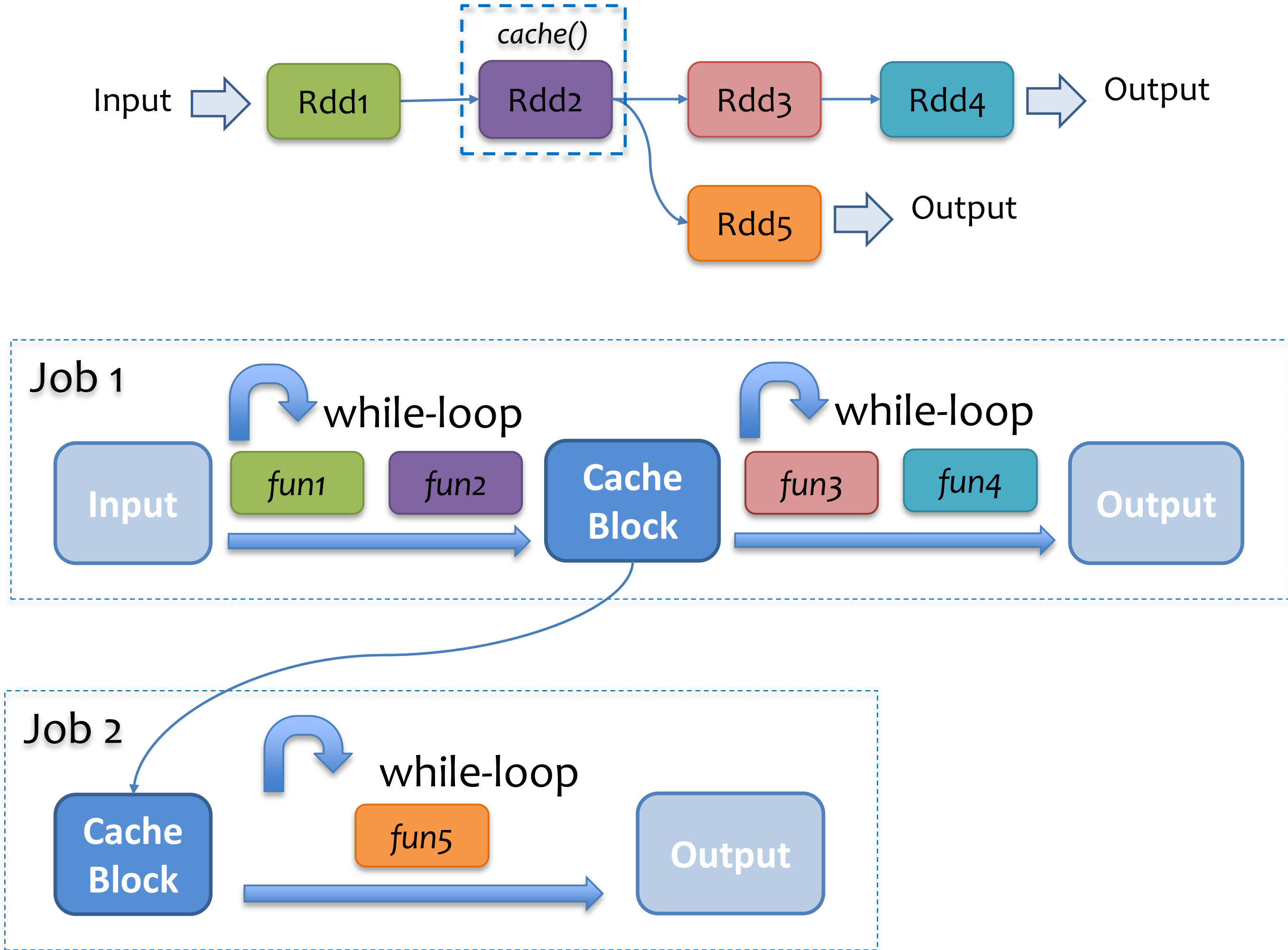
Cache

Cache基本原理



No Cache

Cache基本原理



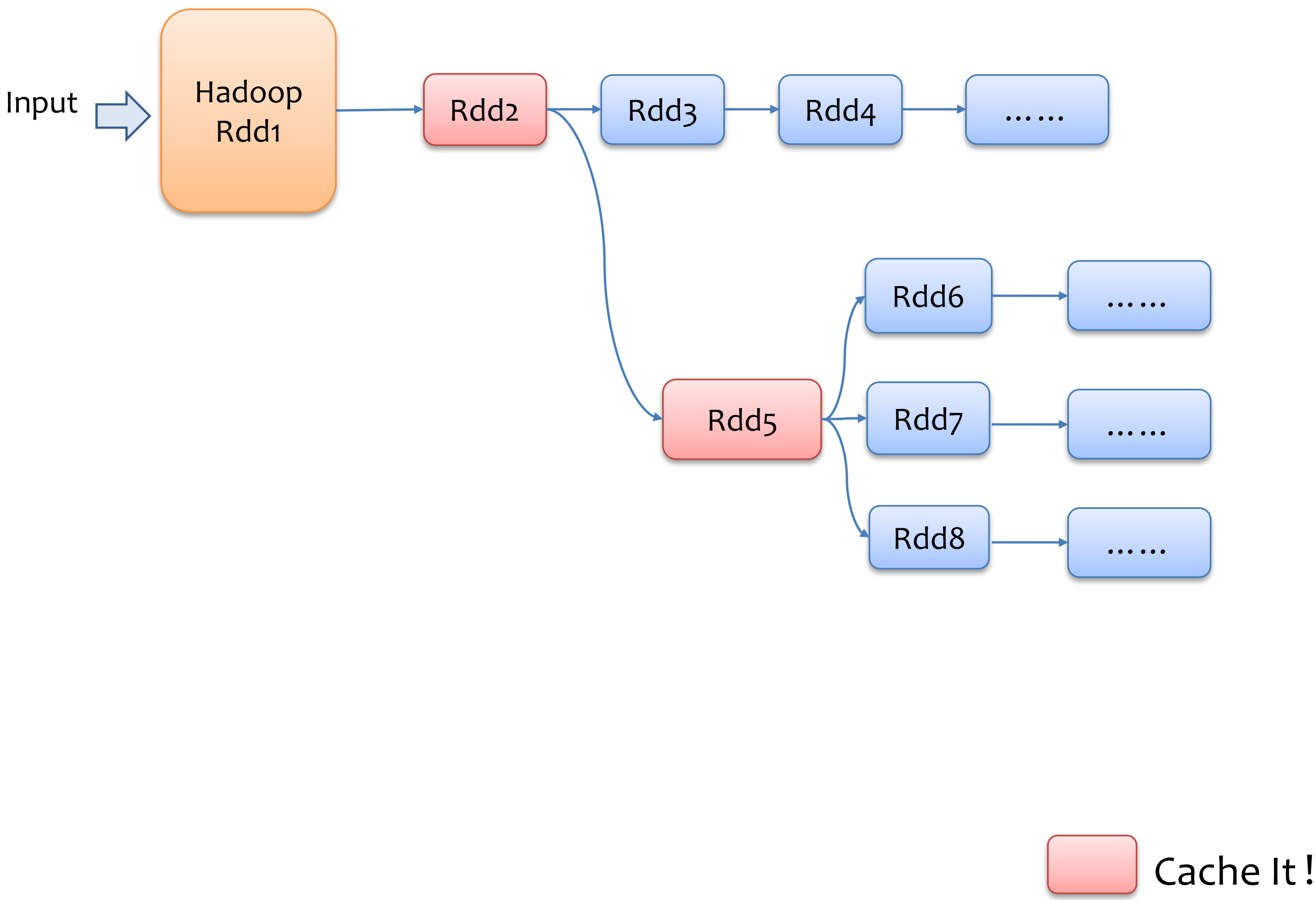
Cache

Cache原则

- Persist原则(*Cache =Persist(StorageLevel.MEMORY_ONLY)*)
  - 长链型的RDD，每个都不需要Cache，Spark会链式执行
  - 树型的RDD，分叉处的RDD，尽量Cache
  - 多次迭代中，会变化的RDD不需要Cache，不变的RDD要Cache

- Unpersist原则
  - 不使用时尽快显示地调用rdd.unpersist(blocking = false)清除缓存
  - unpersist的RDD，依然能够被使用，只是需要被重新计算

Cache 时机


Input
Hadoop
Rdd1
Rdd2
Rdd3
Rdd4
……
Rdd5
Rdd6
……
Rdd7
……
Rdd8
……
Cache It！

Cache性能

Kmeans算法Cache性能测试

输入大小	driver_memory	num_executor	executor_cores	分区数	聚类个数	迭代次数
9.3g	4g	10	2	100	6	10

缓存方式	executor内存使用总量	cache比例	训练时间/s	GC时间占比
MEMORY_ONLY	20g	33%	1558	12%
MEMORY_ONLY_SER	20g	85%	984	7%
MEMORY_ONLY_SER + COMPRESS	20g	100%	534	5%
MEMORY_ONLY	40g	90%	986	7%
MEMORY_ONLY	60g	100%	463	4.7%
MEMORY_AND_DISK	20g	100%	1182	16.9%
DISK_ONLY	20g	100%	514	3.2%

磁盘是多个Application
共享，存在不稳定性，
所以尽量少用

内存充足时推荐：rdd.persist(StorageLevel. MEMORY_ONLY)
内存稀缺时推荐：rdd.persist(StorageLevel. MEMORY_ONLY_SER)
// spark.rdd.compress=true

Spark Core编程模型及最佳实战

Checkpoint

Checkpoint

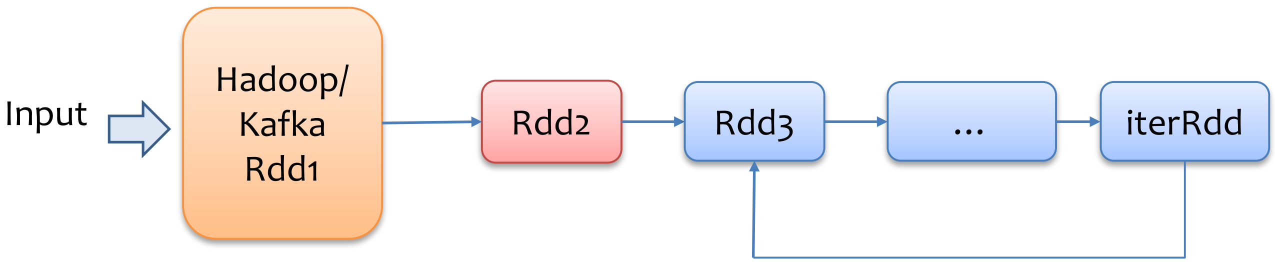
- **Checkpointing** is a process of truncating RDD lineage graph and saving it to a reliable distributed (HDFS) or local file system.

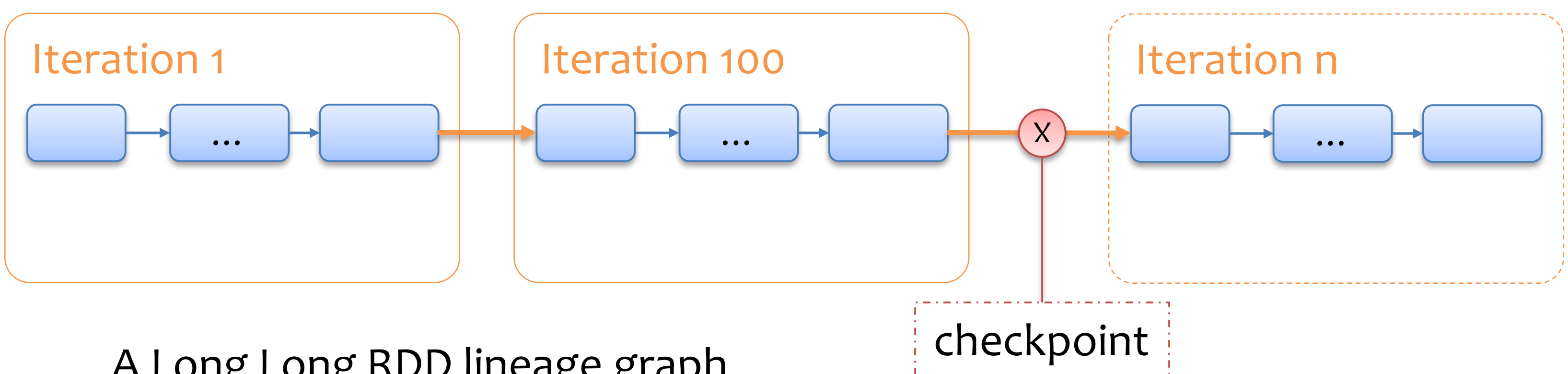
- 可靠性
  - sparkContext.setCheckpointDir -> cache->checkpoint

防止计算两次

断链(Truncate RDD lineage)

GraphX or MLLib





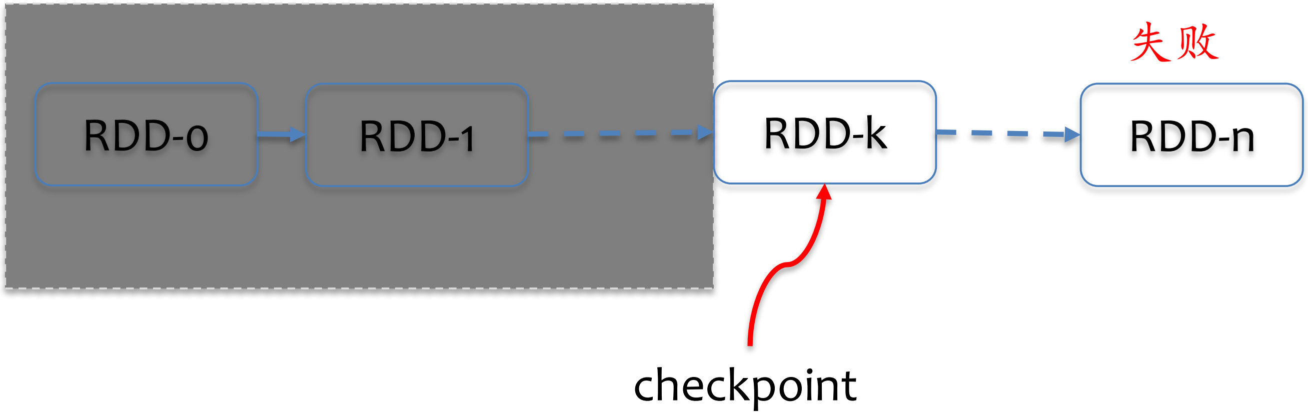
A Long Long RDD lineage graph

Checkpoint举例

- 随着迭代的进行，RDD依赖关系越来越长，driver维护压力变大，很可能driver OOM

```
var curRdd = init(...).cache()
while(i < maxTimes) {
  val updatedRdd = iteration(curRdd).cache()
  curRdd.unpersist(false)
  curRdd = updatedRdd
  if (i % 20 == 0) curRdd.checkpoint()
  i += 1
}
```

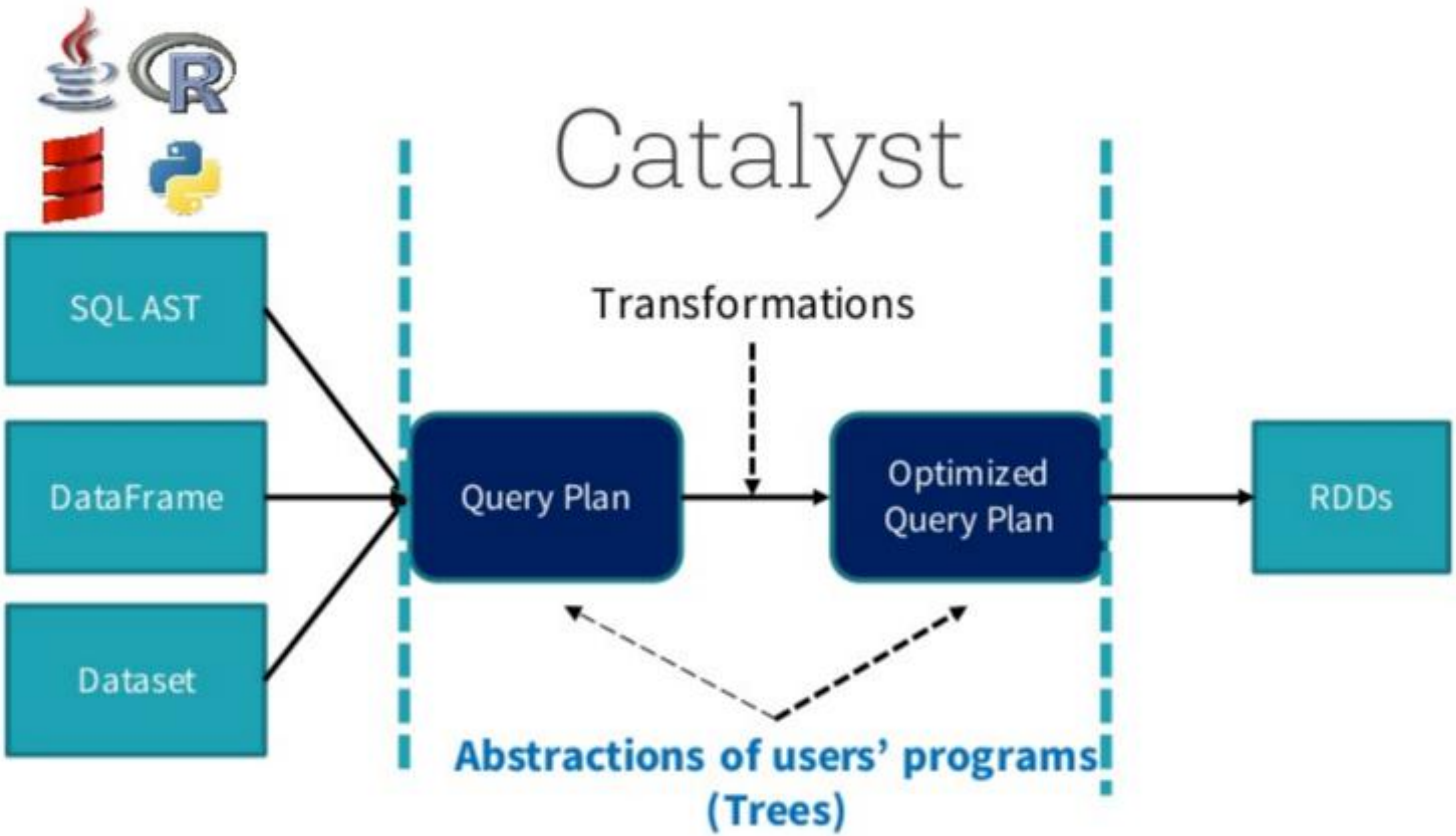
注意：需要做checkpoint的rdd必须是缓存过的，否则会重计算



Spark SQL Dataset/DataFrame编程模型及最佳实战

Dataset/DataFrame

Spark SQL概述



- 引入更高级别的API--Dataset/DataFrame
- 充分利用SparkSQL的优化执行策略，用户可省去复杂的优化过程
- 充分利用SparkSQL的二进制编码、内存管理以及Codegen等优化技术

Dataset/DataFrame简介

id	name	gender	age	hobbies
1	Jim	0	19	Football
2	David	0	18	Basketball
3	Joy	0	20	null
4	Linda	1	20	drawing
…	…	…	…	…

```
root
 |-- id: integer (nullable = false)
 |-- name: string (nullable = false)
 |-- gender: byte (nullable = false)
 |-- age: integer (nullable = false)
 |-- hobbies: string(nullable = true)
```

- DataFrame
  - 分布式数据集
    (可以理解为一张分布式表/视图)
  - 包含schema信息
  - 丰富的sql语义算子
  - 弱类型，不支持编译期类型检查

```
type DataFrame = Dataset[Row]
```

- Dataset
  - 强类型，拥有RDD和DataFrame共同优点
  - 可以像RDD一样支持编译期类型检查
  - 也可以像DataFrame一样调用sql语义的算子

Dataset/DataFrame编程模型

- 完全类似于RDD的编程模型
- 新增sql语义算子(select、where、groupBy等)，灵活方便
- Dataset/DataFrame依赖图会被转化为SQL算子，通过SparkSQL引擎执行

Dataset/DataFrame编程模型

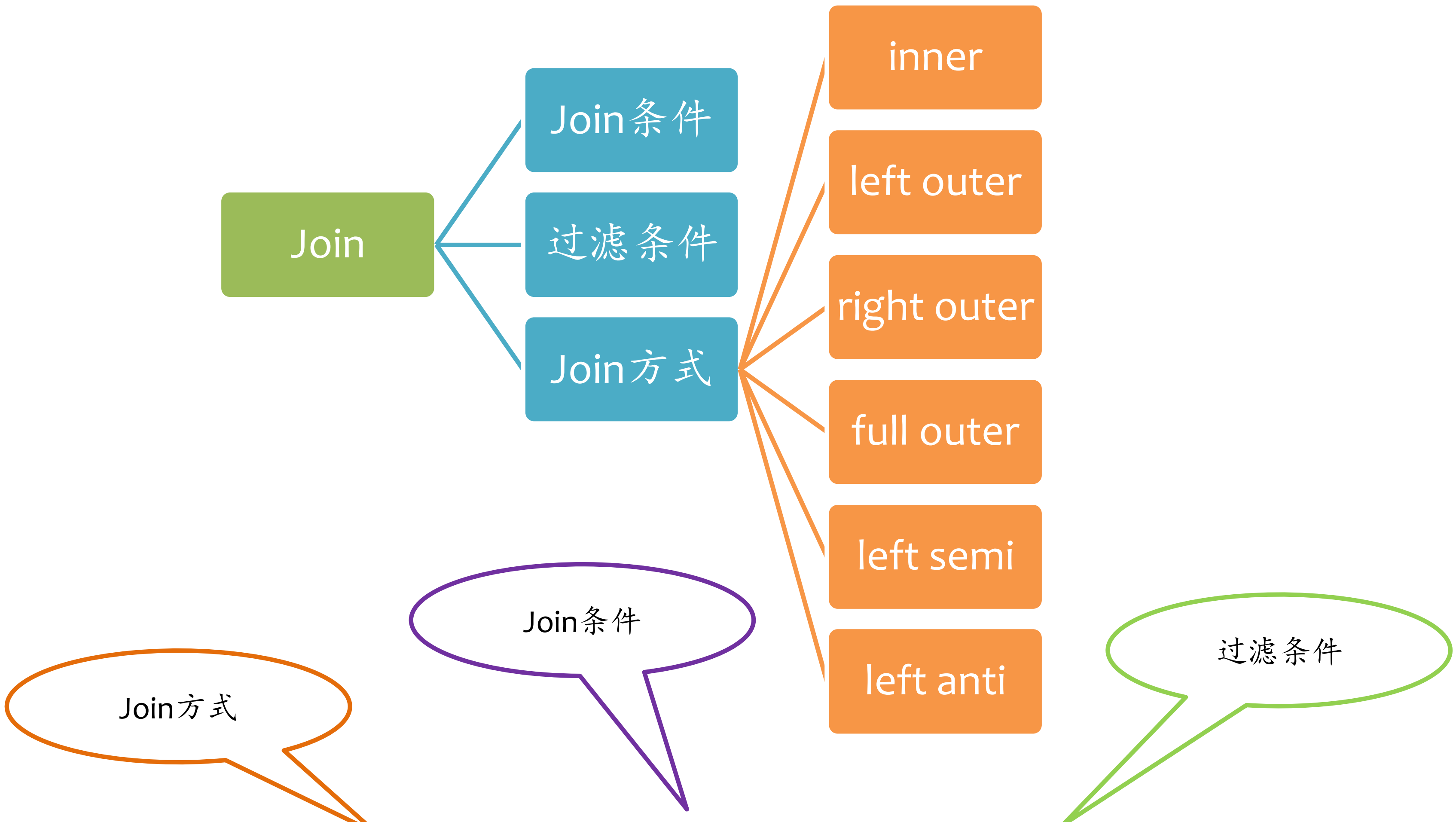
举例

```
import org.apache.spark.sql.SparkSession

val spark = SparkSession.builder().appName("Spark-SQL-Example").getOrCreate()

// For implicit conversions like converting RDDs to DataFrames
import spark.implicits._
// you can use custom classes that implement the Product interface
case class Person(name: String, age: Long)

// Encoders are created for case classes
val caseClassDS = Seq(Person("Jim", 43)).toDS()
// val caseClassDS = spark.createDataset(Seq(Person("Jim", 43)))
caseClassDS.show()
// +----+---+
// |name|age|
// +----+---+
// |Jim |43 |
// +----+---+

// Encoders for most common types are automatically provided by importing spark.implicits._
val primitiveDS = Seq(1, 2, 3).toDS()
primitiveDS.map(_ + 1).collect() // Returns: Array(2, 3, 4)
```

Spark SQL Dataset/DataFrame编程模型及最佳实战

Join

Join 概述




```
// SELECT * FROM A INNER JOIN B ON A.id=B.id WHERE  A.score > 60
dfA.join(dfB,  $"dfAKey" === $"df2Key ", " inner") .where($"dfAScore" > 60)
```

Join 基本流程

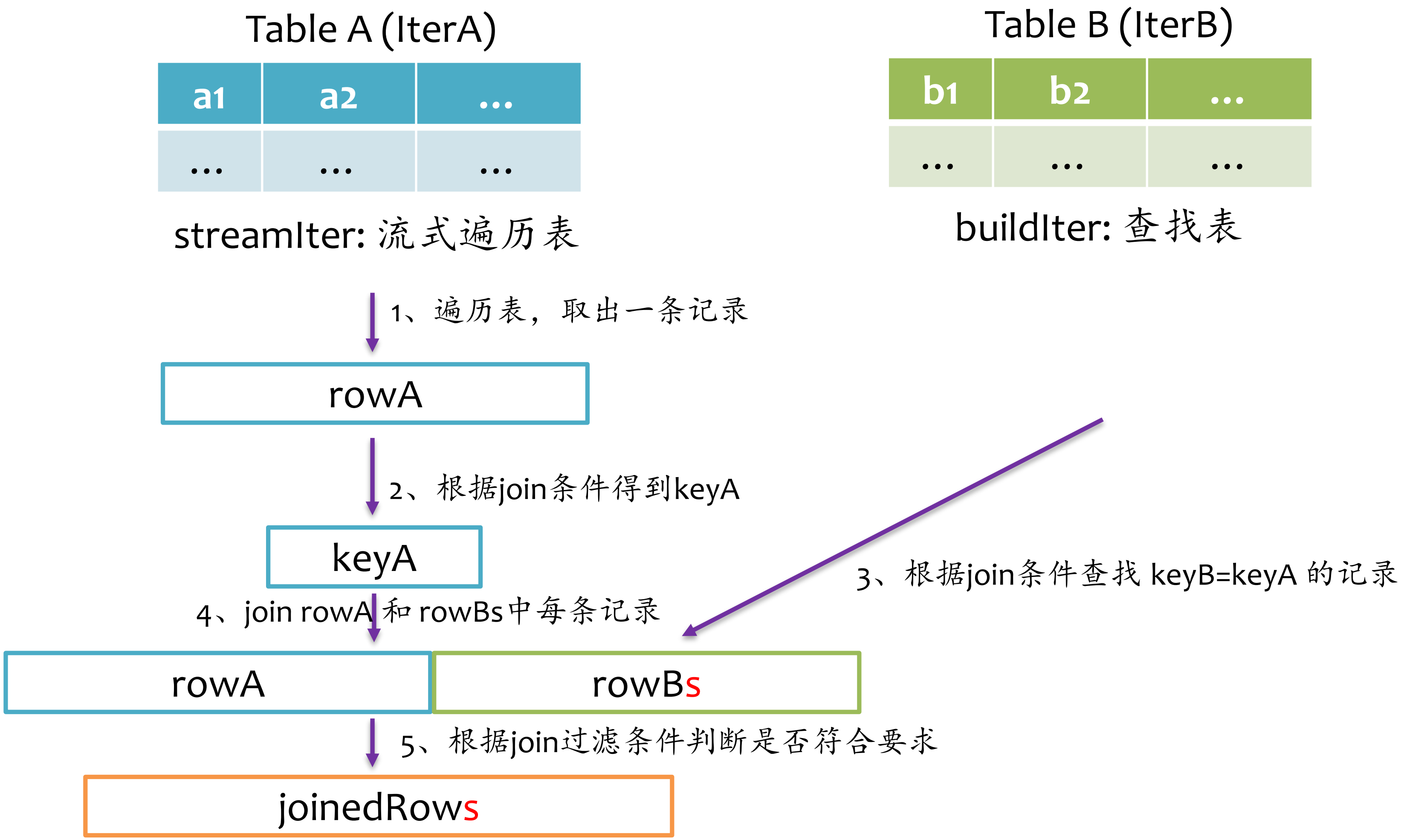
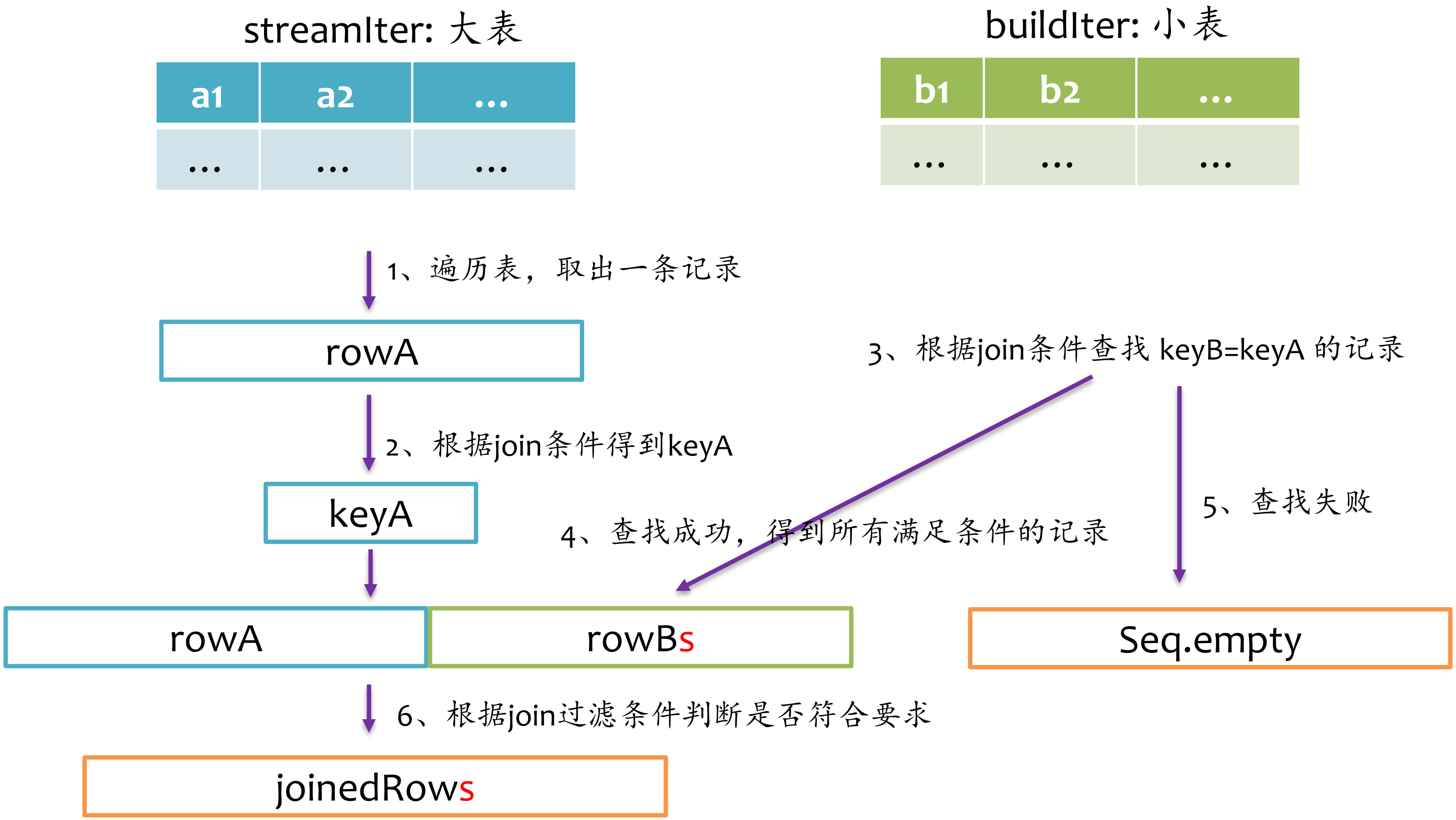


Table A (IterA)
a1
a2
...
...
...
...
streamIter: 流式遍历表
Table B (IterB)
b1
b2
...
...
...
...
buildIter: 查找表
1、遍历表，取出一条记录
rowA
2、根据join条件得到keyA
keyA
3、根据join条件查找 keyB=keyA 的记录
4、join rowA 和 rowBs中每条记录
rowA
rowBs
5、根据join过滤条件判断是否符合要求
joinedRows

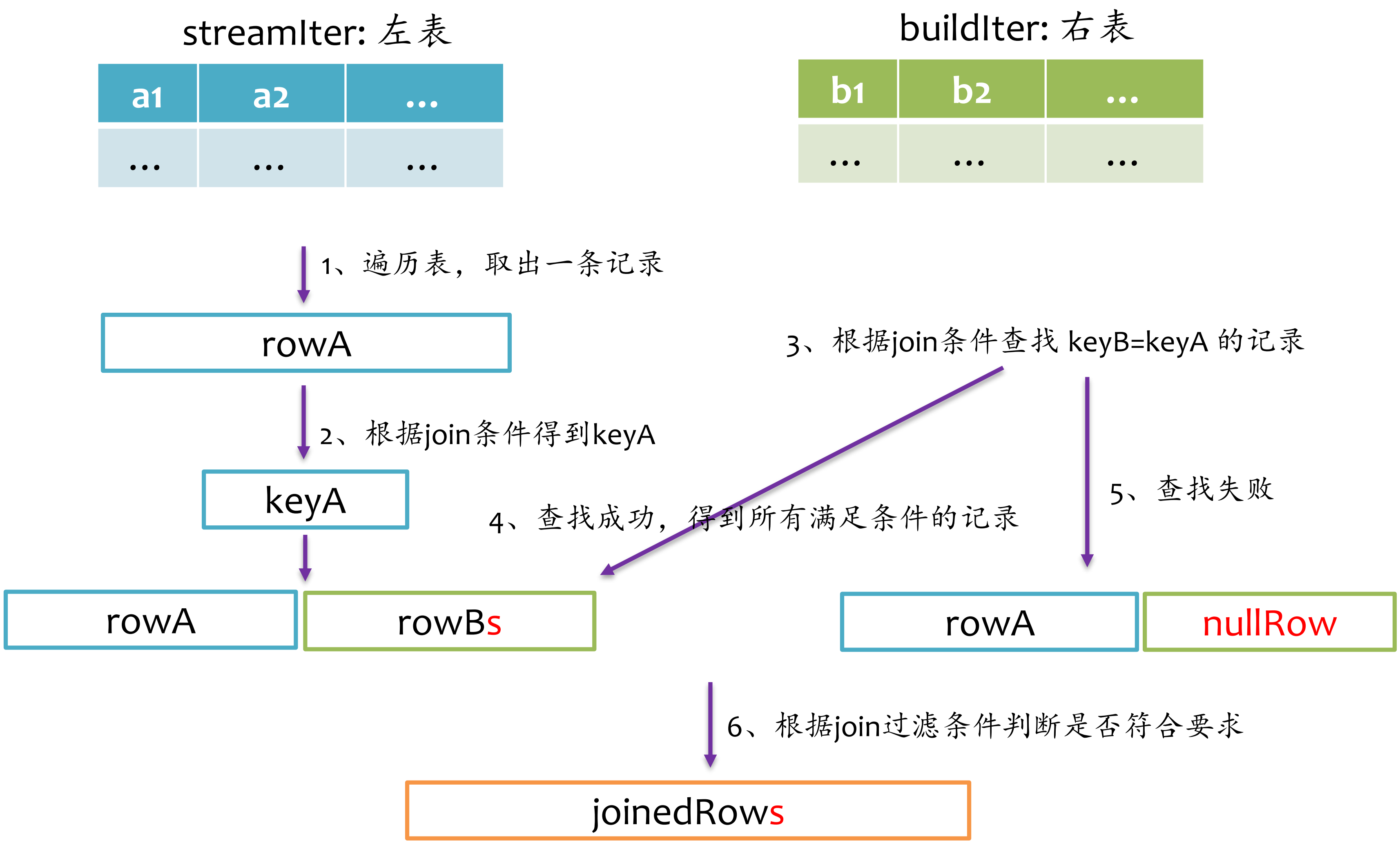
Join 方式

inner join



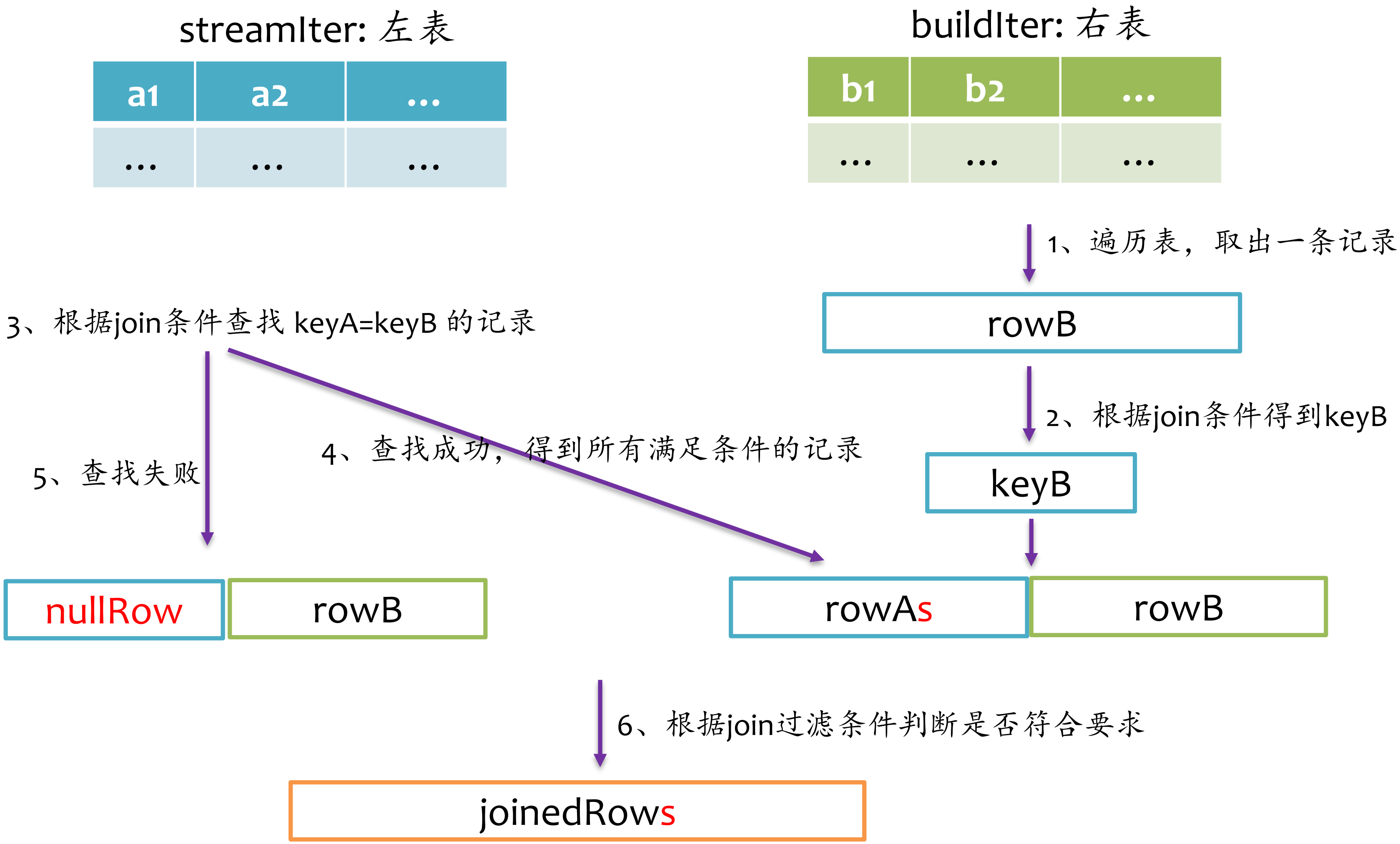
Join 方式

left outer join



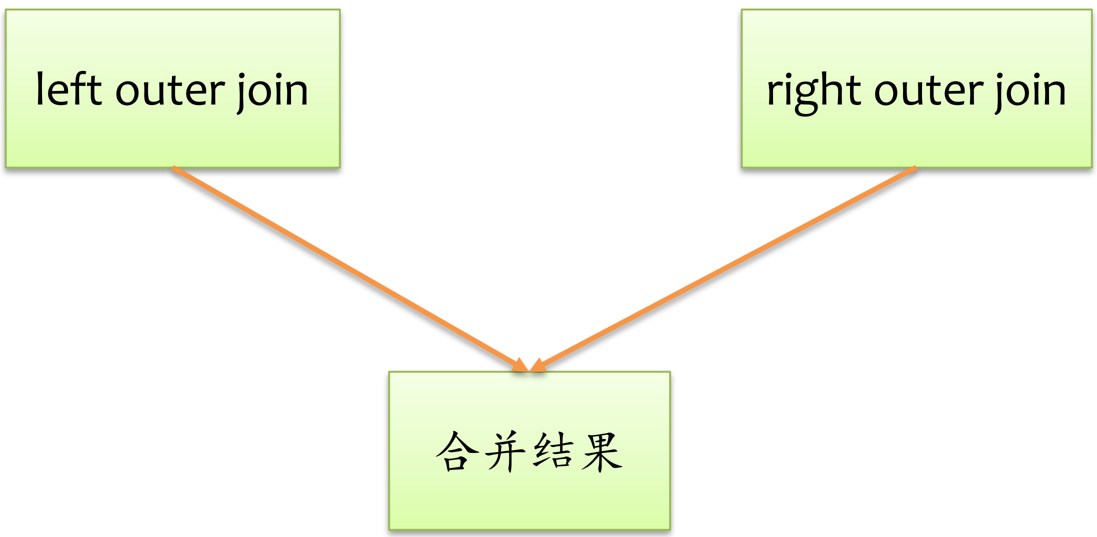
Join 方式

right outer join



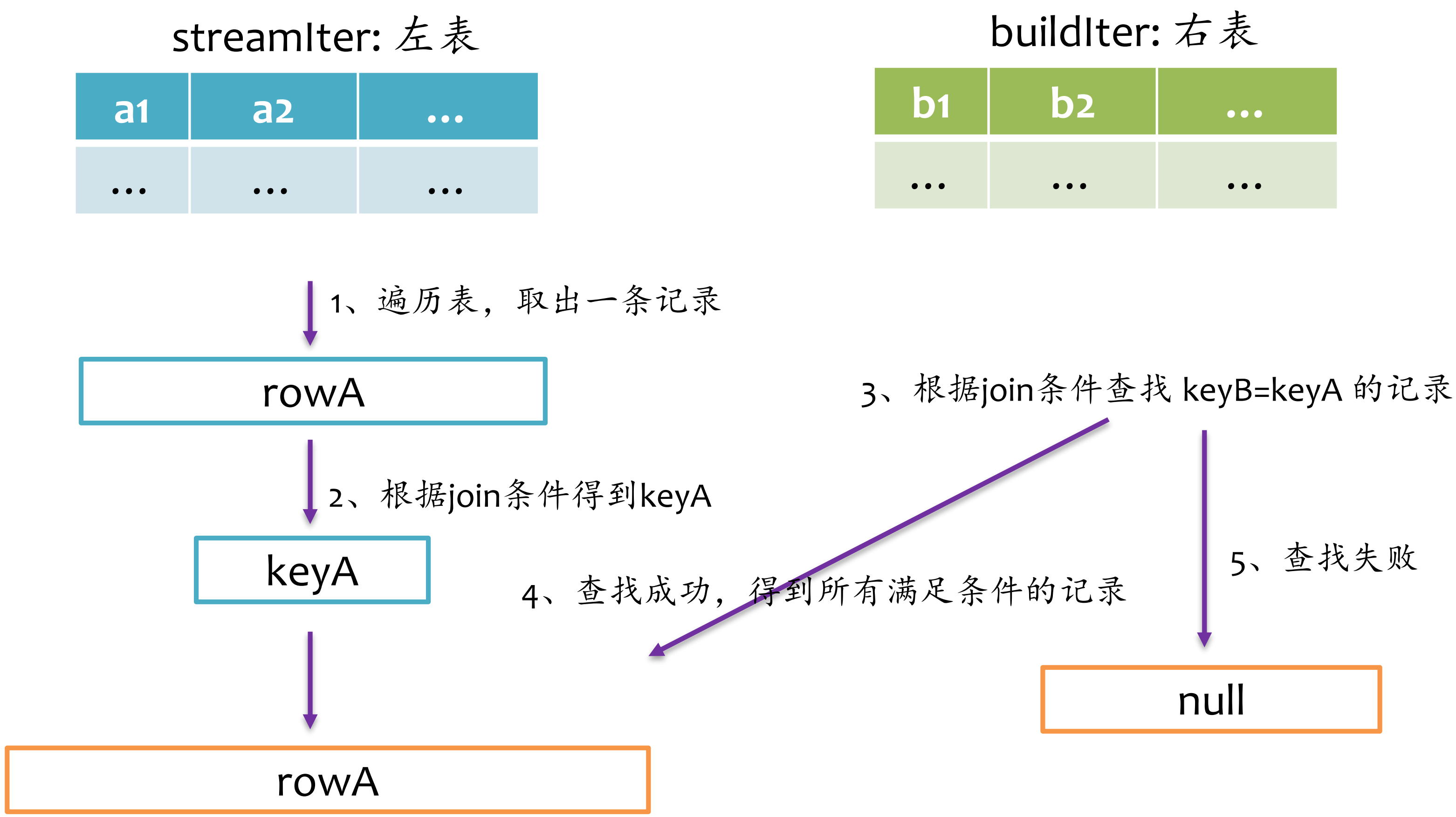
Spark SQL 之 Join

full outer join



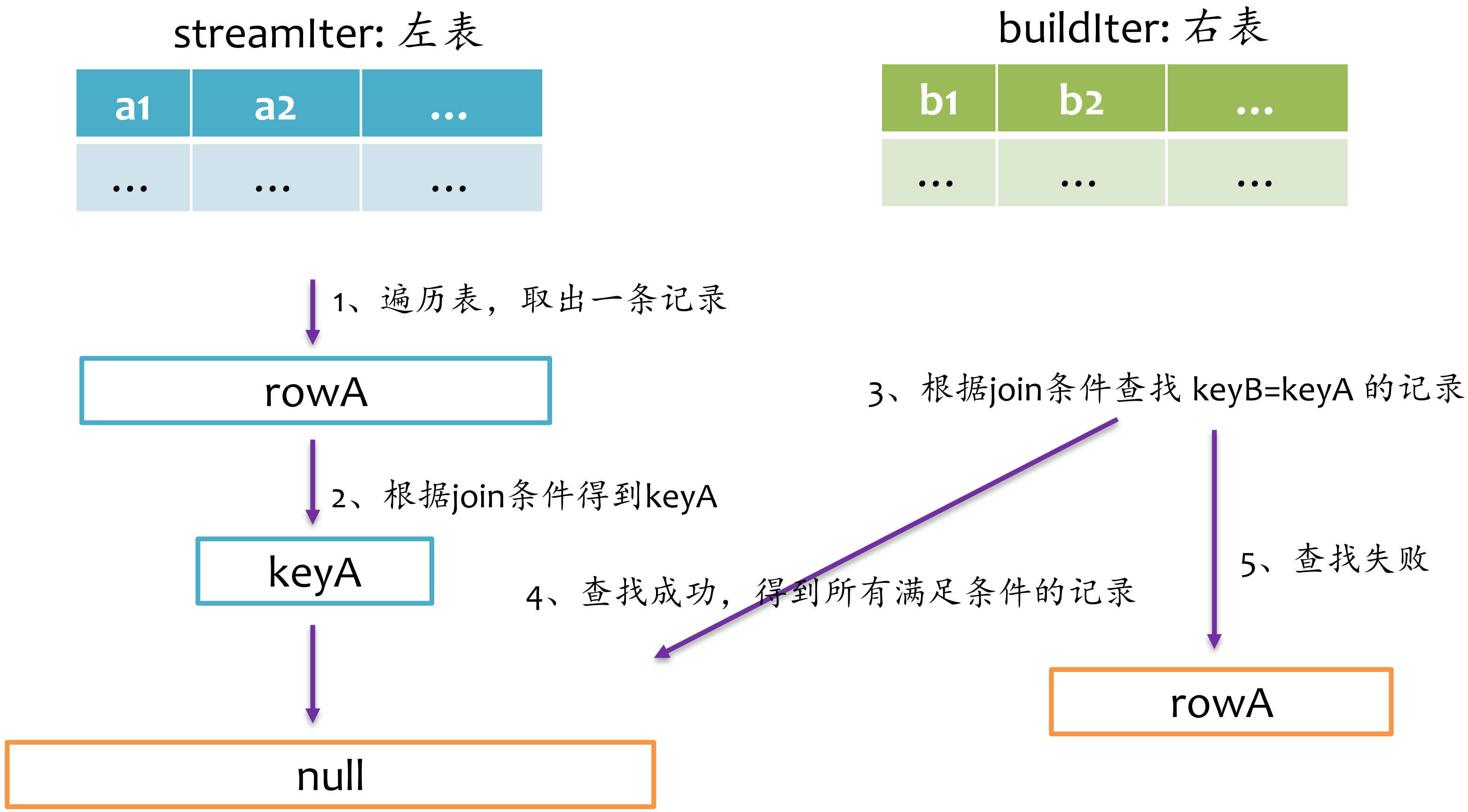
Join 方式

left semi join



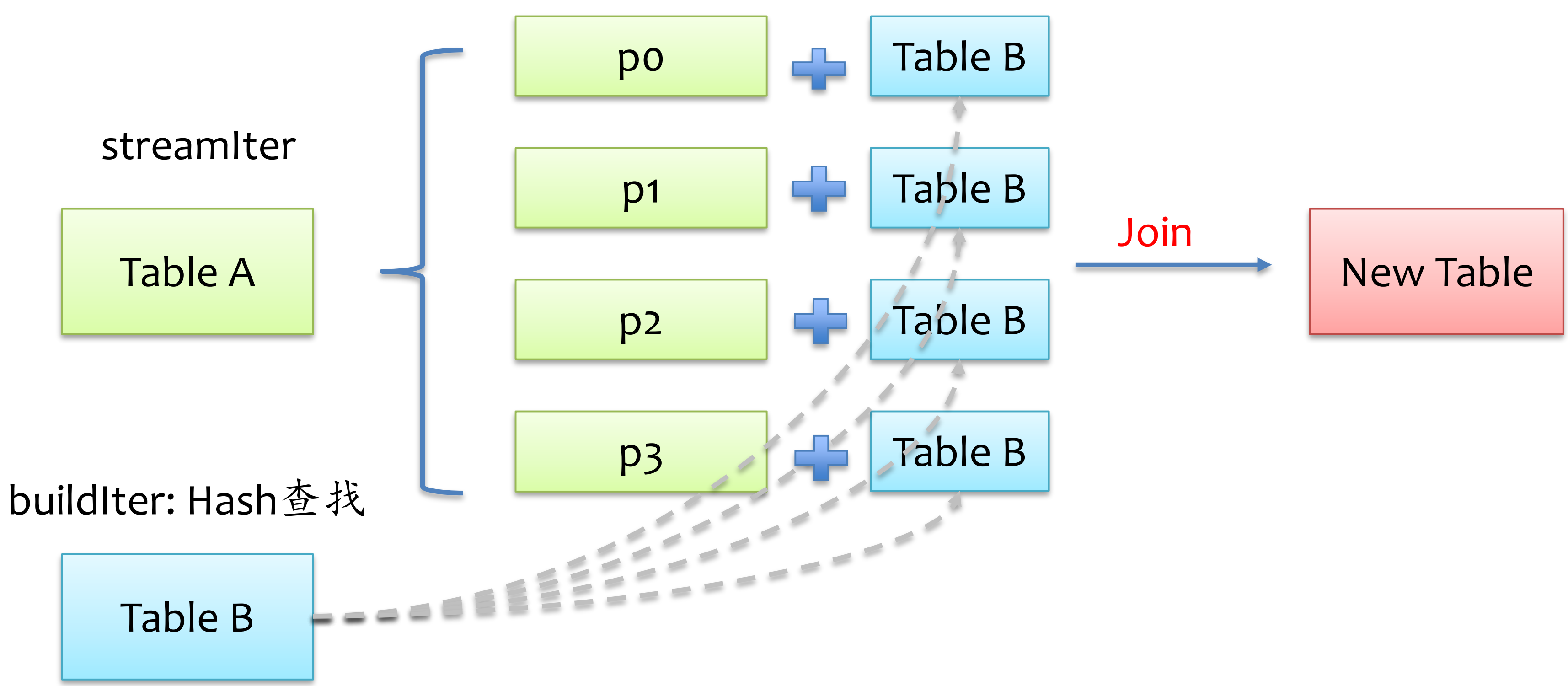
Join 方式

left anti join



Join 实现

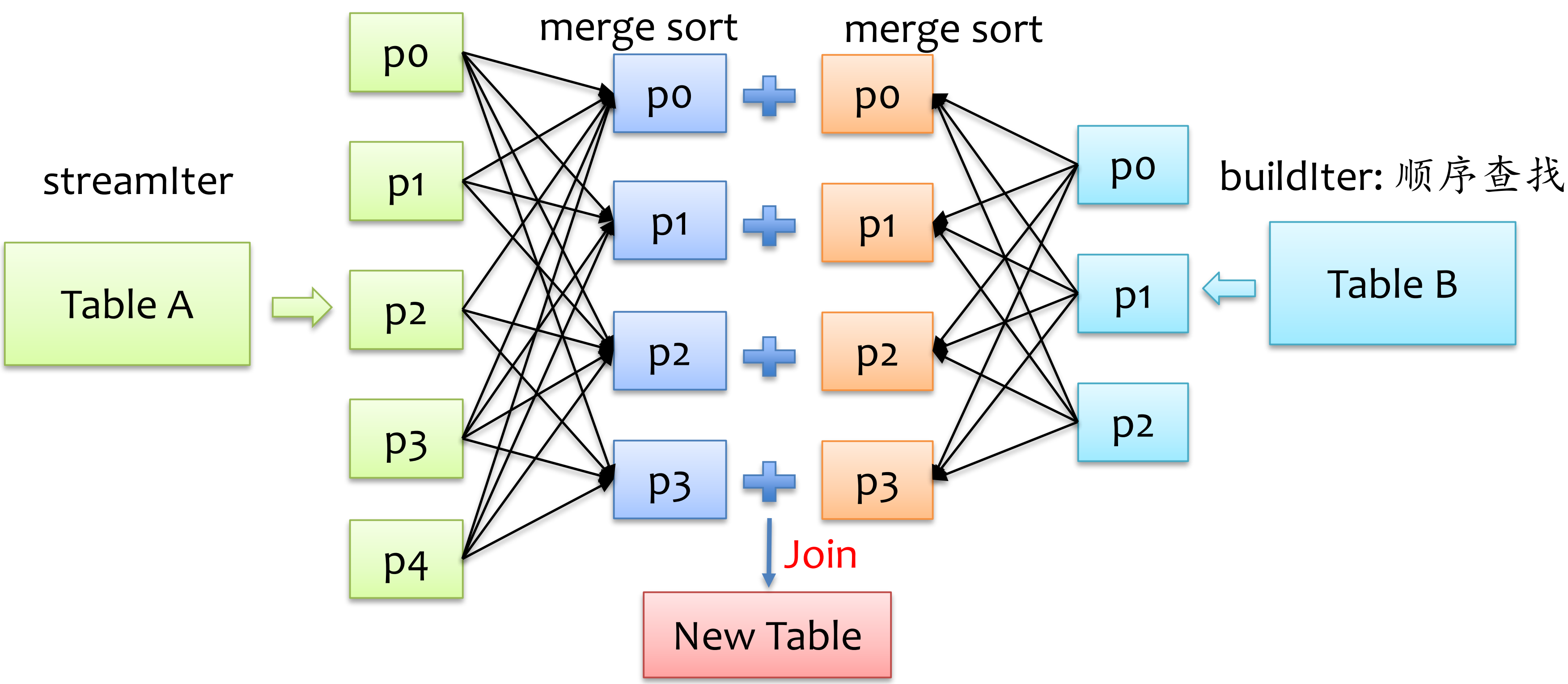
broadcast join 实现



- 无Shuffle，在map端完成
- Table B 不能超过 spark.sql.autoBroadcastJoinThreshold=10M

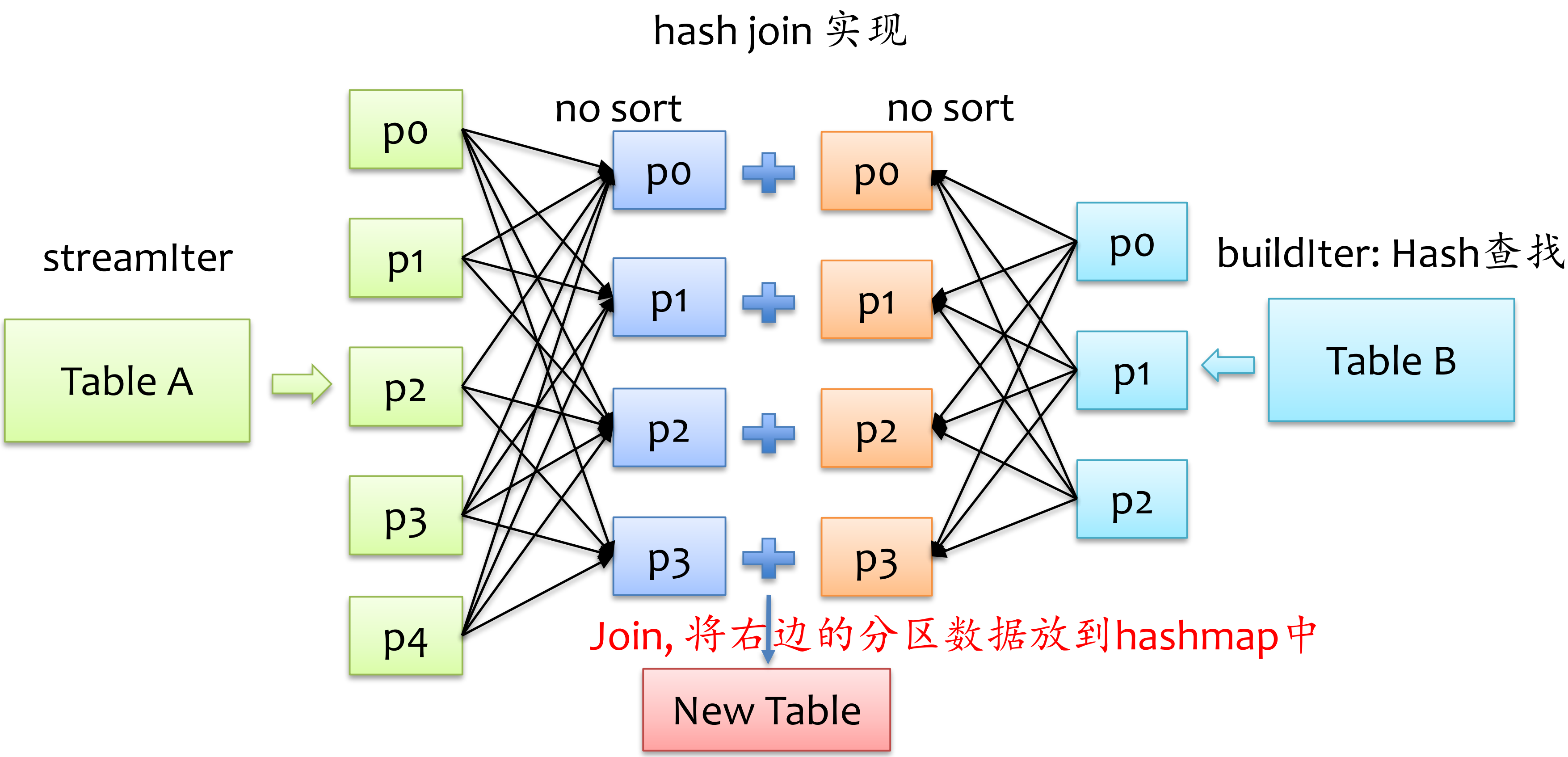
Join 实现

sort join 实现



不满足broadcast join，默认都采用sort join

Join 实现



- Table B 超过10M，即不满足broadcast join条件
- spark.sql.join.preferSortMergeJoin=false
- 分区的平均大小不超过spark.sql.autoBroadcastJoinThreshold=10M
- Table A 大小是 Table B 的三倍以上

Spark SQL Dataset/DataFrame编程模型及最佳实战

Aggregate

Aggregate概述

```
ds.groupBy($"department", $"gender").agg(Map(
  "salary" -> "avg",
  "age" -> "max"
)).show()
```

department	gender	age	salary
1	0	26	5000
1	0	28	6500
2	0	26	5500
1	1	25	5200

department	gender	max (age)	avg (salary)
1	0	28	57500
1	1	25	5200
2	0	26	5500

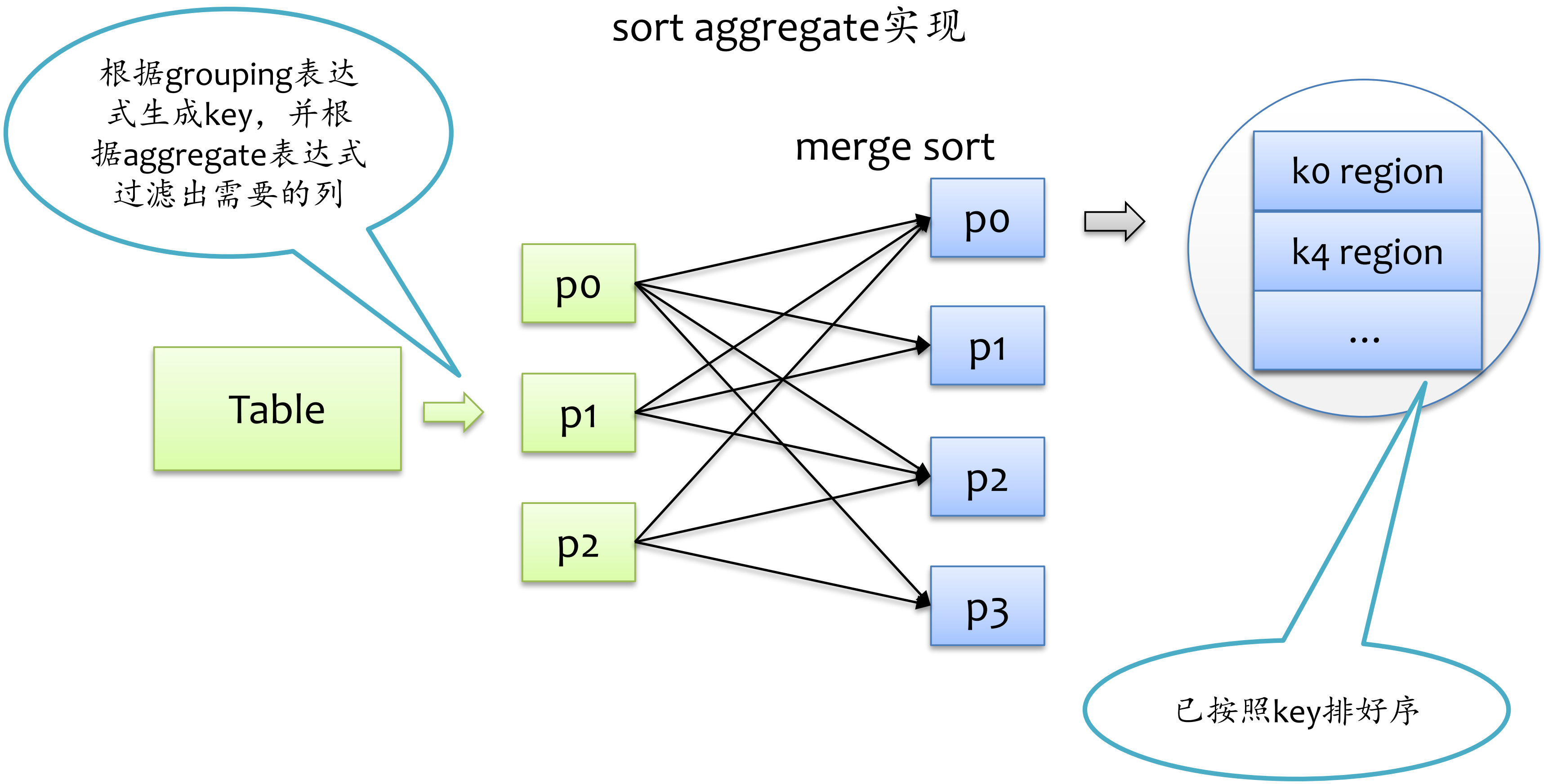
aggregate
表达式，过滤
出需要的列

grouping
表达式，用
于生成key

SELECT max(age), avg(salary) FROM ds GROUP BY department, gender

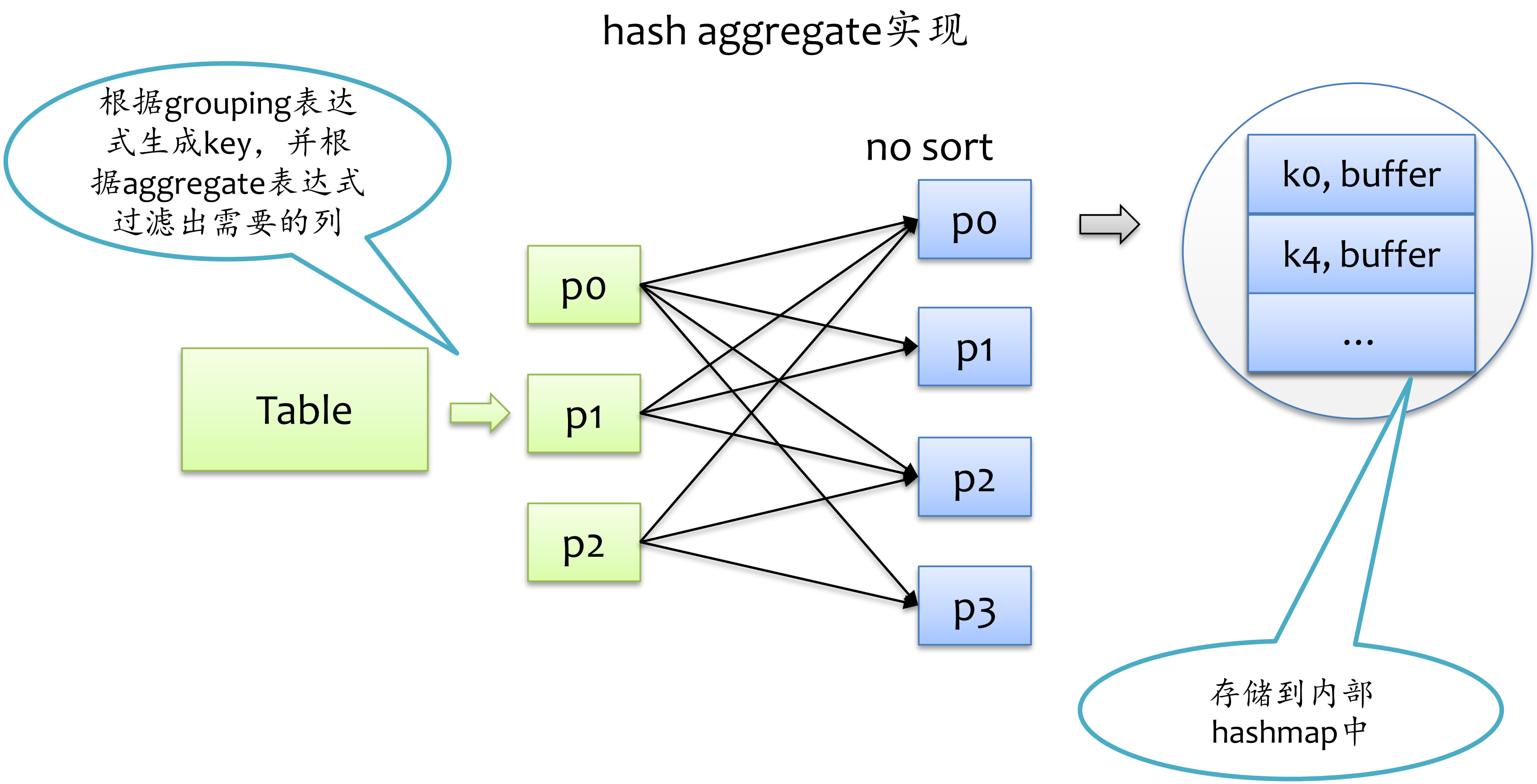
Aggregate实现

sort aggregate实现



在每个分区里按顺序对每个key进行聚合计算

Aggregate实现

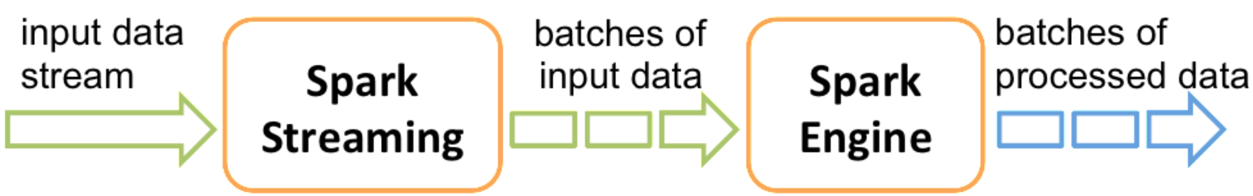


hash aggregate: 只支持原生类型的聚合操作(Boolean, Byte, Short, Int, ...)

Spark Streaming编程模型及最佳实践

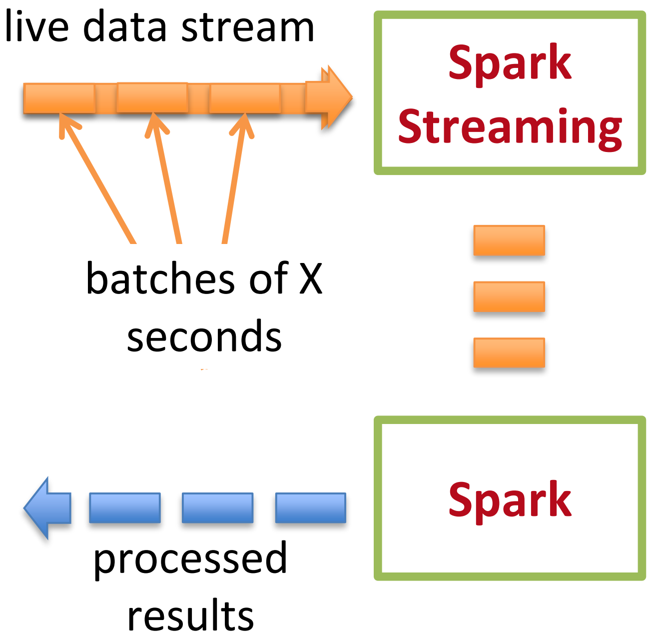
DStream

Spark Streaming概述



DStream 简介

- 按时间将数据流切分为一个个batch
- Spark将每个batch的数据看做一个RDD，Dstream为这些RDD按时间排列的抽象
- Spark按batch时间顺序启动Job处理一个个RDD输出结果



DStream 编程模型

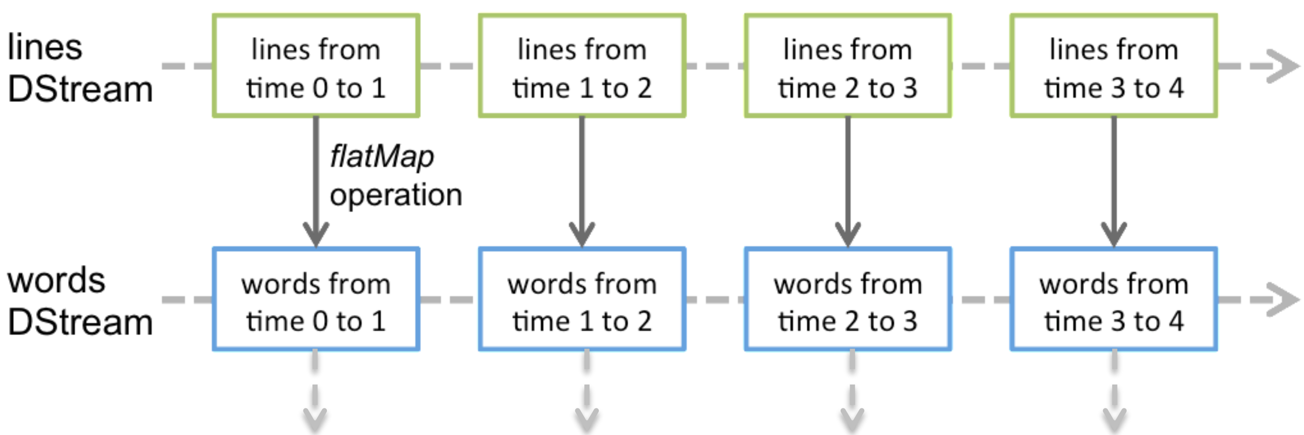
```
val conf = new SparkConf()
val ssc = new StreamingContext(conf, Seconds(60))
val lines = ssc.socketTextStream(“10.215.100.131", 9999)
val words = lines.flatMap(_.split(" "))
val pairs = words.map(word => (word, 1))
val wordCounts = pairs.reduceByKey(_ + _)
wordCounts.print()
ssc.start()
ssc.awaitTermination()
```

1.定义batch
时间间隔

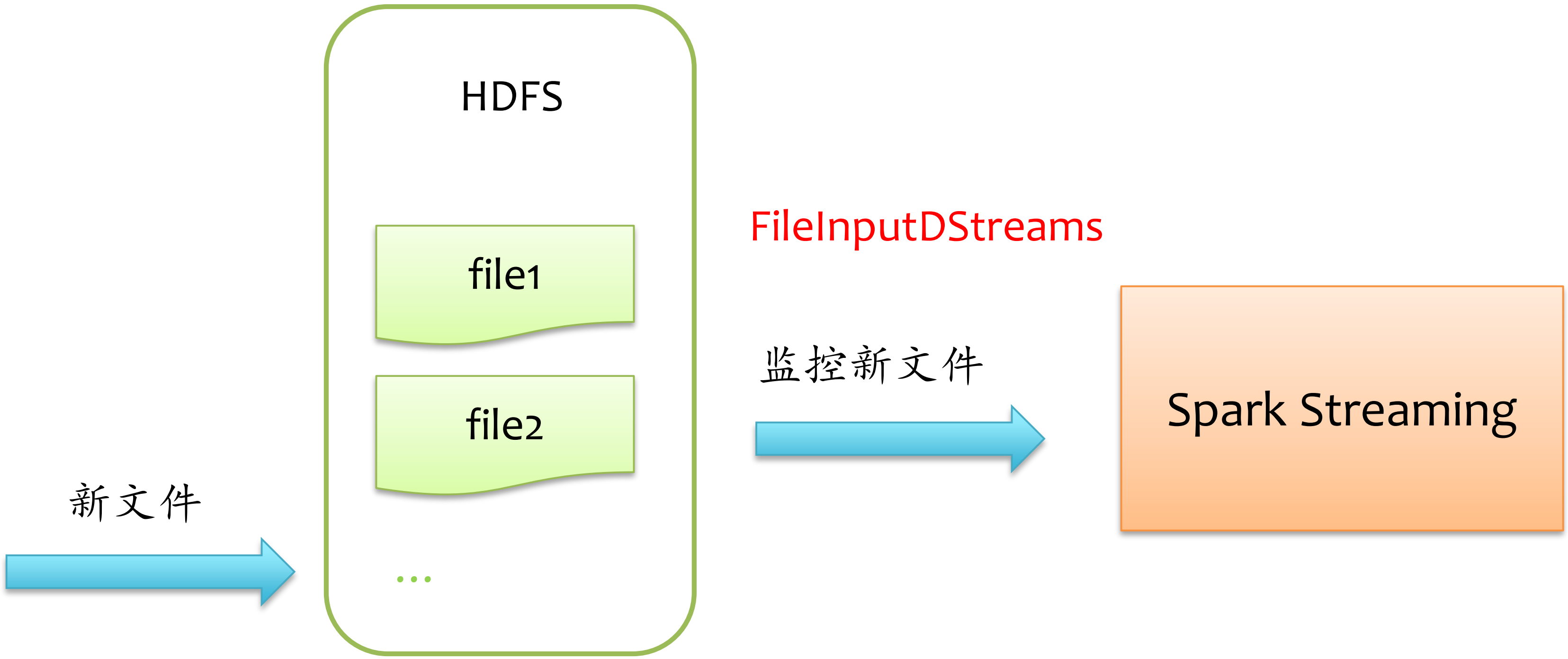
2.指定数据源

3.定义dstream
处理逻辑

4.开始
streaming任务
直到手动终止

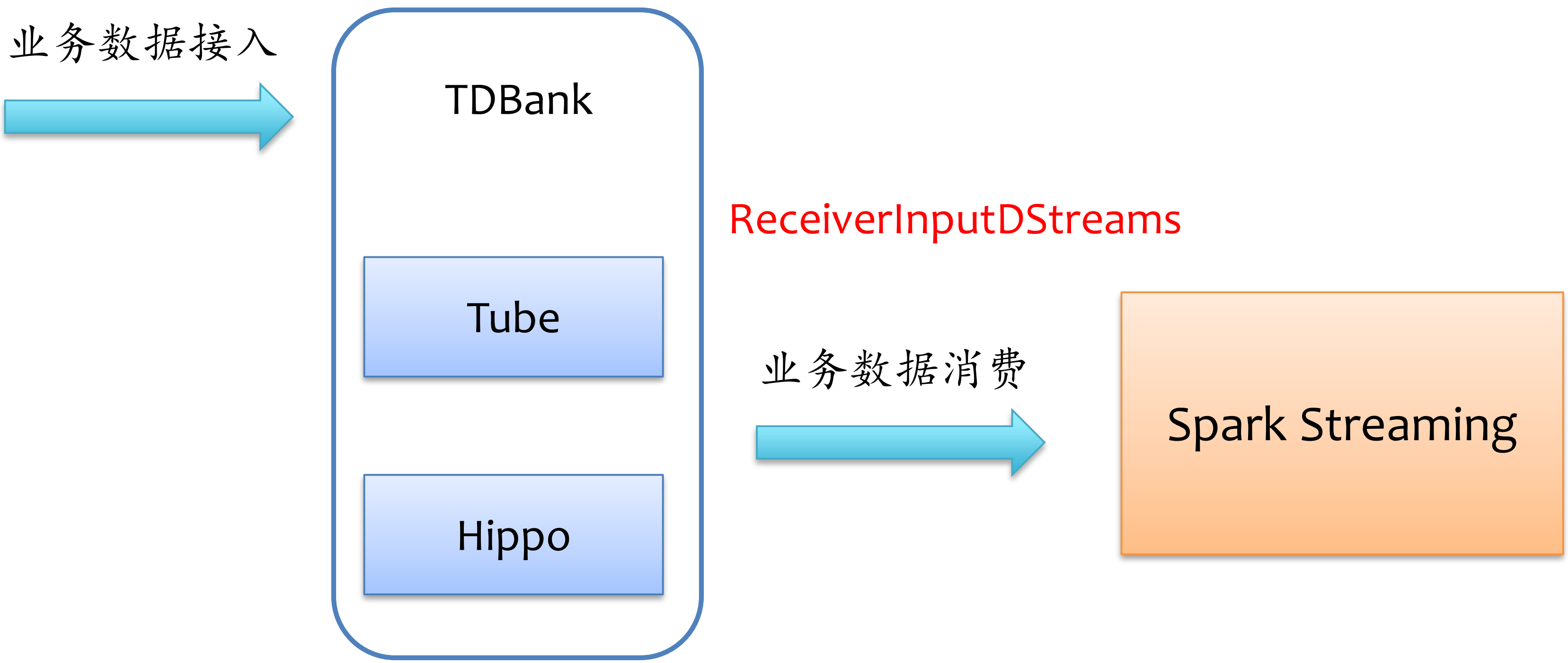


Spark Streaming数据源



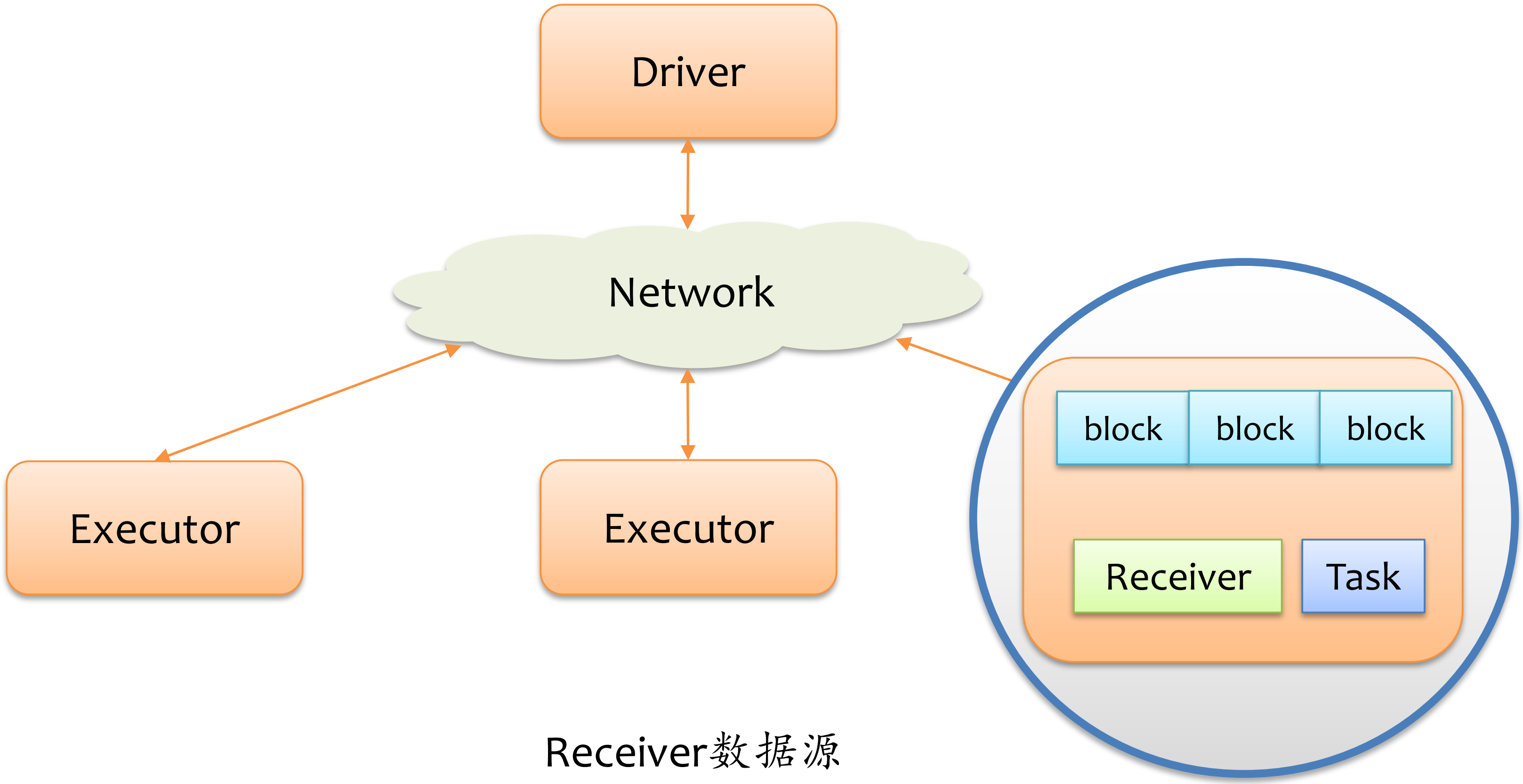
数据源：文件系统

Spark Streaming数据源



数据源：消息中间件(通过Receiver接收)

Spark Streaming数据源



Receiver数据源

Long-running

长时运行保证

Three Factors of Long-running

长时运行保证

- Fault tolerance
  - 增加AM & Spark Driver重试次数以及长时运行保证

```
// tdw-spark default: 1
spark.yarn.maxAppAttempts=4
spark.yarn.am.attemptFailuresValidityInterval=1h
```

  - 增加Executor失败最大容忍次数

```
// default: max(2 * num executors, 3)
spark.yarn.max.executor.failures={8 * num_executors}
spark.yarn.executor.failuresValidityInterval=1h
```

  - 增加Task失败最大容忍次数

```
// default: 4
spark.task.maxFailures=8
```

长时运行保证

- Performance *(保证一个batch的处理时间不大于一个batch的间隔)*
  - 开启推测执行，淘汰那些跑的慢的Task，注意action操作是幂等的

```
spark.speculation=true
```

  - 如果数据源是来自Receiver，建议num_receiver=num_executor，同时控制每个batch的partition数

```
// num_partition =  (batch_interval / blockInterval) * num_receiver
spark.streaming.blockInterval=200ms
```

  - 如果数据源是来自消息中间件，建议控制接收速率

```
spark.streaming.backpressure.enabled=true
//单位:每秒接收记录的条数
spark.streaming.receiver.maxRate=xxx
```

长时运行保证

- Stop gracefully *(保证异常退出或主动kill时不丢数据)*
  - 应用程序异常退出时，可能还有一些数据尚未处理完，在关闭前做好收尾工作

```
spark.streaming.stopGracefullyOnShutdown=true
```

  - 如果主动Kill，上述参数可能并不一定有用

```
yarn application –kill appid
```

Send SIGTREM → Wait 250ms → Send SIGKILL

解决办法：
为应用程序单独开启一个线程监控hdfs文件，当检测到文件被删除后主动stop优雅退出。

谢谢

Q & A