数据分析与挖掘

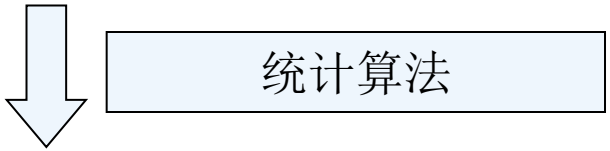
大数据处理常见场景

- 批量处理(Batch Processing)
  - ✓ 侧重于处理海量数据，处理速度可忍受，时间可能在数十分钟到数小时
  - ✓ MR
- 历史数据交互式查询(Interactive Query)
  - ✓ 时间在数秒到数十分钟之间
  - ✓ Presto
- 实时流数据处理(Streaming Processing)
  - ✓ 通常在数十毫秒到数秒之间
  - ✓ Storm

MR编程模型

- 词频统计

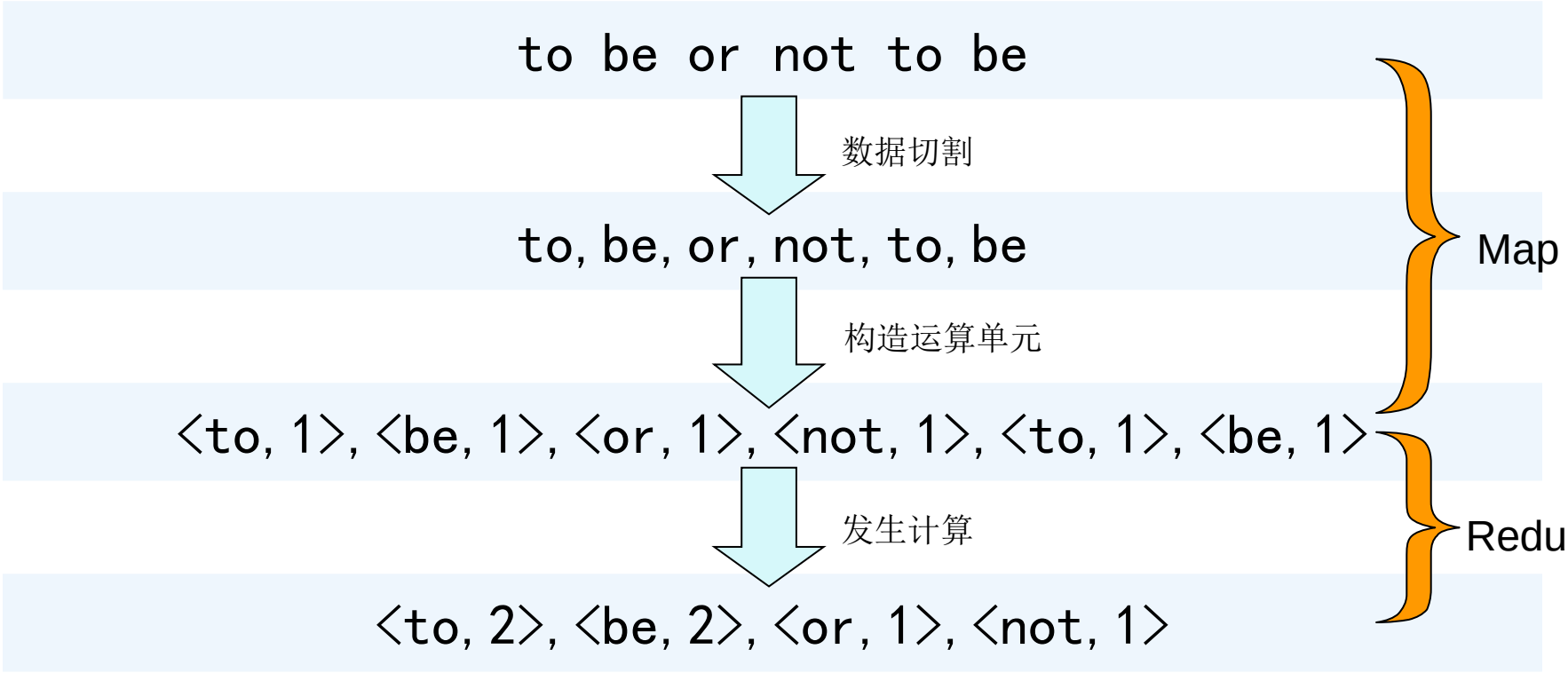
```
to be or not to be
```




```
to: 2
be: 2
or: 1
not:1
```

MR编程模型

- 示例：词频统计



MR代码预览

```
SparkConf sparkConf = new SparkConf().setAppName("JavaWordCount");
JavaSparkContext ctx = new JavaSparkContext(spar
JavaRDD<String> lines = ctx.textFile(args[0], 1)

JavaRDD<String> words = lines.flatMap(new FlatMapFunction<String, String>() {
  @Override
  public Iterable<String> call(String s) {
    return Arrays.asList(SPACE.split(s));
  }
});

JavaPairRDD<String, Integer> ones = words.mapToPair(new PairFunction<String, String, Integer>() {
  @Override
  public Tuple2<String, Integer> call(String s) {
    return new Tuple2<String, Integer>(s, 1);
  }
});

JavaPairRDD<String, Integer> counts = ones.reduceByKey(new Function2<Integer, Integer, Integer>() {
  @Override
  public Integer call(Integer i1, Integer i2) {
    return i1 + i2;
  }
});

List<Tuple2<String, Integer>> output = counts.collect();
for (Tuple2<?,?> tuple : output) {
  System.out.println(tuple._1() + ": " + tuple._2());
}
ctx.stop();
```

to be or not to be

to, be, or, not, to, be

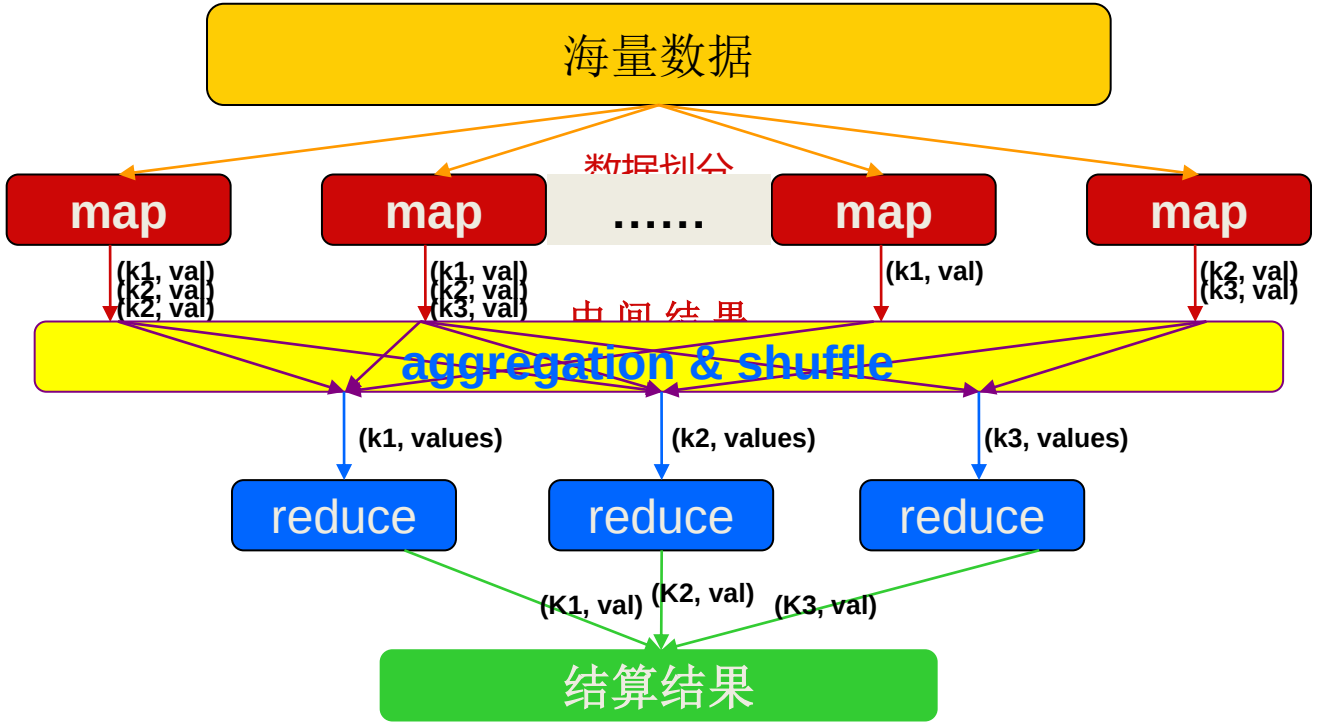
<to, 1>, <be, 1>, <or, 1>, <not, 1>, <to, 1>, <be, 1>

<to, 2>, <be, 2>, <or, 1>, <not, 1>

输出结果

```
SparkConf sparkConf = new SparkConf().setAppName("JavaWordCount");
JavaSparkContext ctx = new JavaSparkContext(sparkConf);
JavaRDD<String> lines = ctx.textFile(args[0], 1);

JavaRDD<String> words = lines.flatMap(new FlatMapFunction<String, String>() {
  @Override
  public Iterable<String> call(String s) {
    return Arrays.asList(SPACE.split(s));
  }
});

JavaPairRDD<String, Integer> ones = words.mapToPair(new PairFunction<String, String, Integer>() {
  @Override
  public Tuple2<String, Integer> call(String s) {
    return new Tuple2<String, Integer>(s, 1);
  }
});

JavaPairRDD<String, Integer> counts = ones.reduceByKey(new Function2<Integer, Integer, Integer>() {
  @Override
  public Integer call(Integer i1, Integer i2) {
    return i1 + i2;
  }
});

List<Tuple2<String, Integer>> output = counts.collect();
for (Tuple2<?,?> tuple : output) {
  System.out.println(tuple._1() + ": " + tuple._2());
}
ctx.stop();
```

MR编程模型


海量数据
map
map
……
map
map
(k1, val)
(k2, val)
(k2, val)
(k1, val)
(k2, val)
(k3, val)
(k1, val)
(k2, val)
(k3, val)
aggregation & shuffle
(k1, values)
(k2, values)
(k3, values)
reduce
reduce
reduce
(K1, val)
(K2, val)
(K3, val)
结算结果

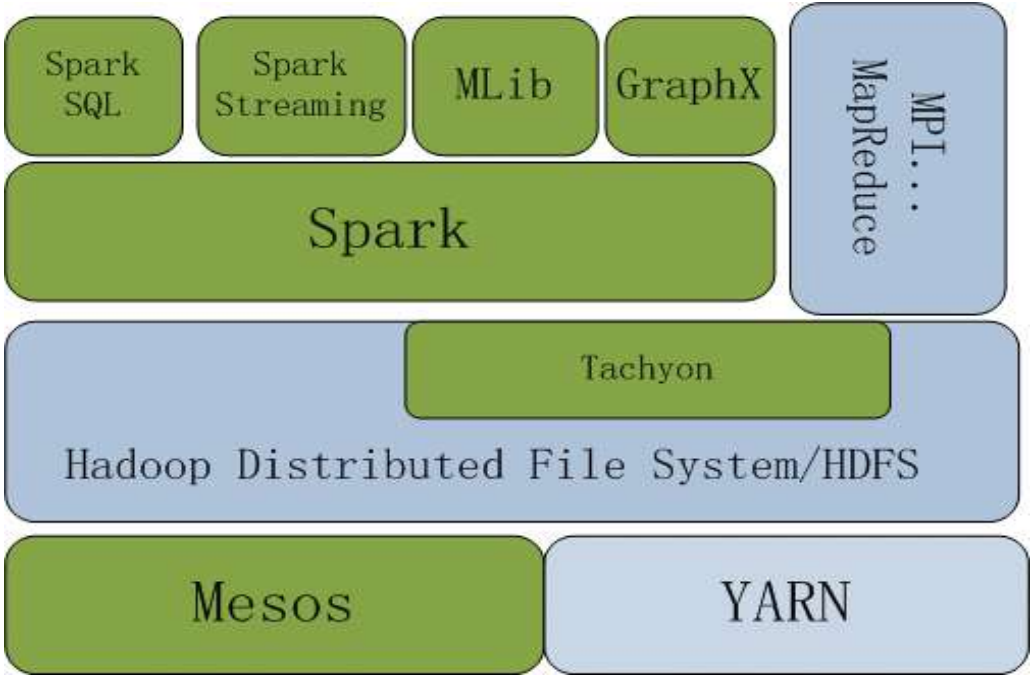
Spark篇

Spark是什么?

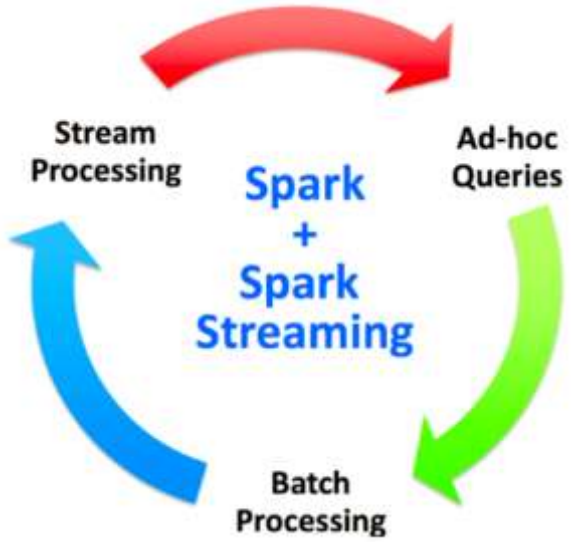
Spark生态圈是伯克利APMLab实验室打造的，力图在算法、机器、人之间通过大规模集成来展现大数据应用的一个平台。

Spark立足内存计算，从多迭代批量处理出发，兼收并蓄数据仓库、流处理和图计算等多种计算范式。

Spark生态圈已经涉及到机器学习、数据挖掘、数据库、信息检索、自然语言处理和语音识别等多个领域。



Spark 是什么?



- 大数据处理一站式解决平台!?
- 基于内存分布式并行计算框架

Spark动机

- ✓ 复杂的批量数据处理（batch data processing），通常的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（interactive query），通常的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（streaming data processing），通常的时间跨度在数百毫秒到数秒之间。

Spark优点

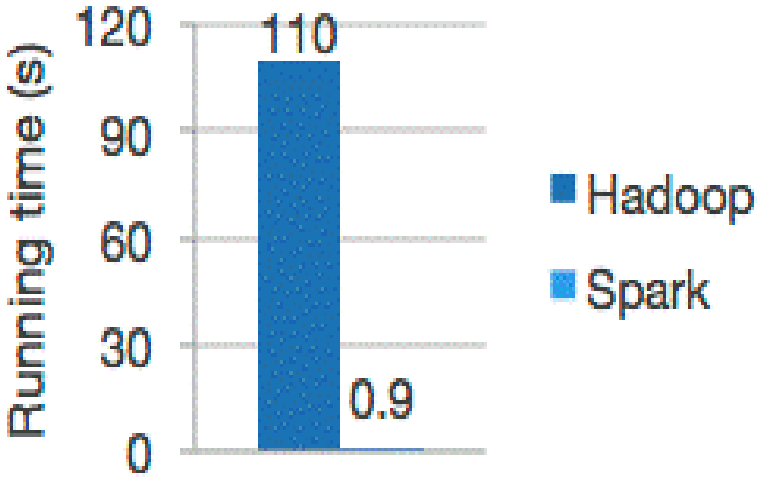
- ✓轻：Spark1.0核心代码3万行，Hadoop1.0 9万行，2.0 22万行。
- ✓快：Spark基于内存的迭代计算框架，适用于需要多次操作特定数据集的应用场合。Spark对小数据集能达到亚秒级的延迟
  - ✓这对于Hadoop MapReduce是无法想象的（由于“心跳”间隔机制，仅任务启动就有数秒的延迟）。
  - ✓就大数据集而言，对典型的迭代机器学习、图计算等应用，Spark版本比基于MapReduce、Hive和Pregel的实现快上十倍到百倍。其中内存计算、数据本地性（locality）和传输优化、调度优化等该居首功。
- ✓灵：Spark提供了不同层面的灵活性。
  - ✓在实现层，可更换的集群调度器、序列化库；
  - ✓在原语（Primitive）层，它允许扩展新的数据算子、新的数据源、新的language（Java和Python）；
  - ✓在范式（Paradigm）层，Spark支持内存计算、多迭代批量处理、即时查询、流处理和图计算等多种范式。
- ✓巧：巧在借势和借力。
  - ✓Spark借Hadoop之势，与Hadoop无缝结合；接着Spark SQL借了Hive的势；
- ✓Spark不适用异步细粒度更新状态的应用，如web服务的存储或者是增量的web爬虫和索引。

Spark特点

运行速度快

Spark对小数据集能达到亚秒级的延迟，对于Hadoop MapReduce是无法想象的（由于”心跳”间隔机制，仅任务启动就有数秒的延迟）。

Spark拥有DAG执行引擎，支持在内存中对数据进行迭代计算。如果数据由磁盘读取，速度是Hadoop MapReduce的10倍以上，如果数据从内存中读取，速度可以高达100多倍。



Spark特点

易用性好

Spark不仅支持Scala编写应用程序，而且支持Java和Python等语言进行编写，特别是Scala是一种高效、可拓展的语言，能够用简洁的代码处理较为复杂的处理工作。

在实现层，它完美演绎了Scala trait动态混入策略（如可更换的集群调度器、序列化库）；

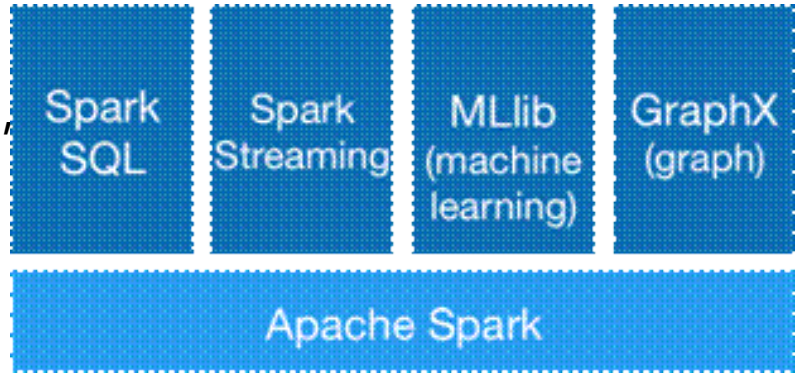
在原语层，它允许扩展新的数据算子、新的数据源、新的language bindings(Java和Python)；

在范式层，Spark支持内存计算、多迭代批星处理、流处理和图计算等多种范式。

Spark特点

通用性强

Spark生态圈即BDAS（伯克利数据分析栈）包含了Spark Core、Spark SQL、Spark Streaming、MLLib和GraphX等组件，这些组件分别处理Spark Core提供内存计算框架、SparkStreaming的实时处理应用、Spark SQL的即席查询、MLlib或MLbase的机器学习和GraphX的图处理，它们都是由AMP实验室提供，能够无缝的集成并提供一站式解决平台。



Spark特点

随处运行

Spark具有很强的适应性，能够读取HDFS、Cassandra、HBase、S3和Techyon为持久层读写原生数据，能够以Mesos、YARN和自身携带的Standalone作为资源管理器调度job，来完成Spark应用程序的计算。

Spark与Hadoop比较

Spark借鉴了MapReduce分布式并行计算的优点并改进了MapReduce明显缺陷：

Spark把中间数据放到内存中，迭代运算效率高。MapReduce中计算结果需要落地，保存到磁盘上，这样势必会影响整体速度，而Spark支持DAG图的分布式并行计算的编程框架，减少了迭代过程中数据的落地，提高了处理效率。

Spark容错性高。Spark引进了弹性分布式数据集RDD（Resilient Distributed Dataset）的抽象，它是分布在一组节点中的只读对象集合，这些集合是弹性的，如果数据集一部分丢失，则可以根据“血统”（即充许基于数据衍生过程）对它们进行重建。另外在RDD计算时可以通过CheckPoint来实现容错，而CheckPoint有两种方式：CheckPoint Data，和Logging The Updates，用户可以控制采用哪种方式来实现容错。

Spark更加通用。不像Hadoop只提供了Map和Reduce两种操作，Spark提供的数据集操作类型有很多种，大致分为：Transformations和Actions两大类。Transformations包括Map、Filter、FlatMap、Sample、GroupByKey、ReduceByKey、Union、Join、Cogroup、MapValues、Sort和PartionBy等多种操作类型，同时还提供Count，Actions包括Collect、Reduce、Lookup和Save等操作。另外各个处理节点之间的通信模型不再像Hadoop只有Shuffle一种模式，用户可以命名、物化，控制中间结果的存储、分区等。

Spark与Hadoop比较

Spark是Apache顶级的开源项目，在迭代计算，交互式查询计算以及批量流计算方面都有相关的子项目，Shark、Spark Streaming、MLbase、GraphX、SparkR等

Spark core

其他主件：

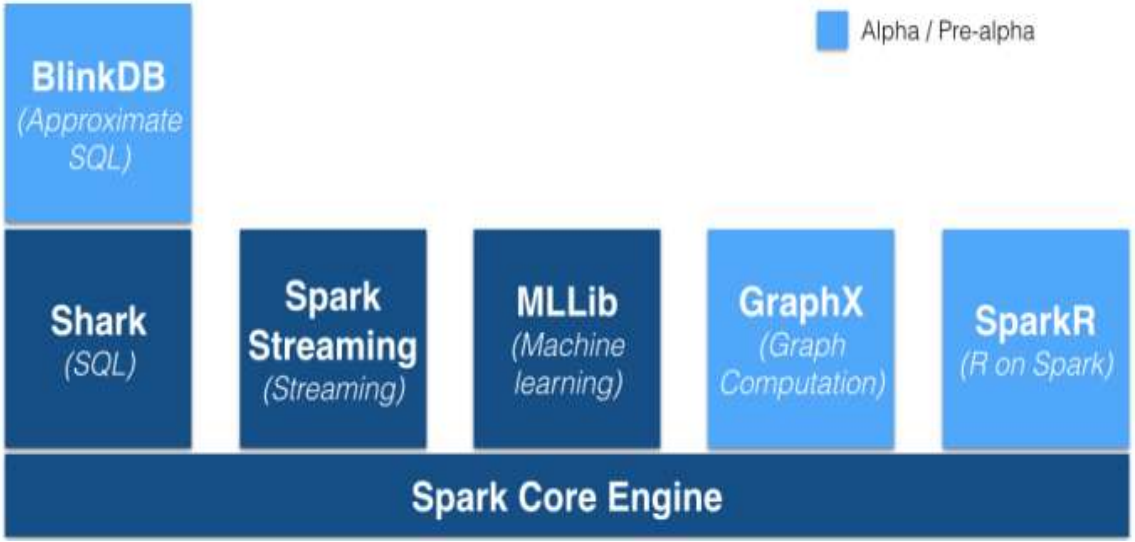
及时查询工具：Shark（sql） spark sql

流式处理工具：spark steaming

机器学习、数据挖掘 Mllib

图处理：Graphx

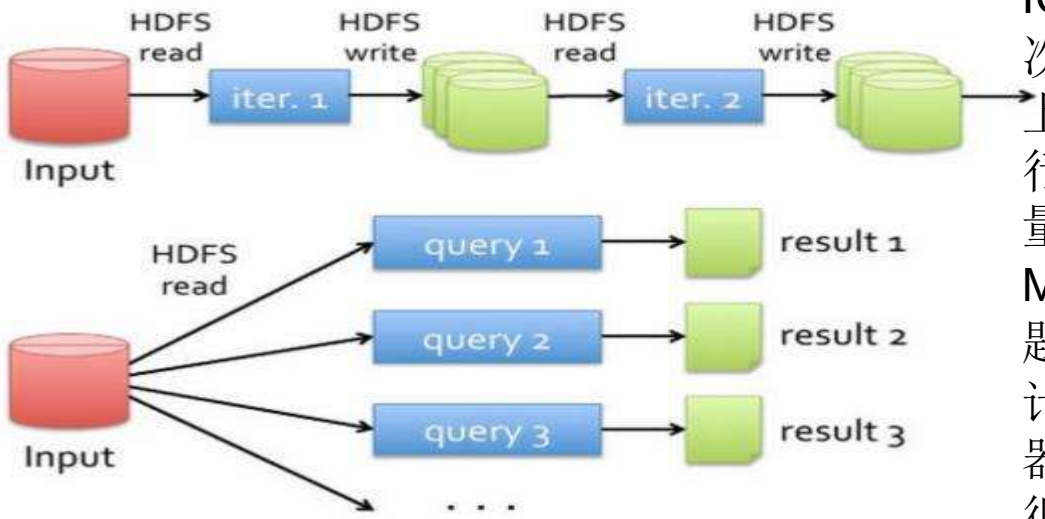
将R扩展成并行运算的SparkR



BlinkDB：在查询速度和查询精度进行权衡的快速查询引擎

Spark VS MapReduce

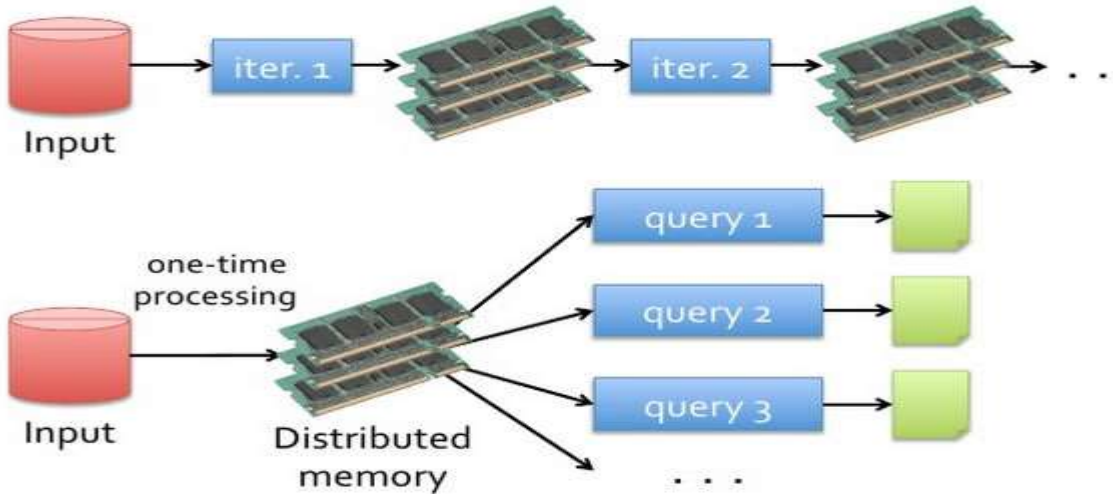
传统Hadoop数据抽取运算模型



数据的抽取运算基于磁盘，中间结果也是存储在磁盘上。MR运算伴随着大量的磁盘IO。Mapreduce在计算时每次的计算结果都会写到硬盘上，每次再从硬盘上读取进行下一次运算，由于造成大量的I/O操作，会有延迟MapReduce自身存在很多问题，包括迭代式计算和DAG计算等类型的数据挖掘与机器学习算法性能低下，不能很好地利用内存资源，编程复杂度较高

Spark VS MapReduce

Spark 则使用内存代替了传统HDFS存储中间结果



第一代Hadoop完全使用Hdfs存储中间结果，第二代Hadoop加入了cache来保存中间结果。而Spark则基于内存的中间数据集存储。可以将Spark理解为Hadoop的升级版本，Spark兼容了Hadoop的API，并且能够读取Hadoop的数据文件格式，包括HDFS，Hbase等。

Spark读写过程不像hadoop溢出写入磁盘，都是基于内存，因此速度很快。DAG作业调度系统的宽窄依赖让Spark可以一次运行多个任务，大大提高速度。

Spark VS MapReduce

- MR只提供Map和Reduce两种操作，Spark提供Transformations和Actions两大类操作。Spark不仅实现了MapReduce的算子map 函数和reduce函数及计算模型，还提供更为丰富的算子，如filter、join、groupByKey等。是一个用来实现快速而同用的集群计算的平台。
  - Transformations操作：map, filter, flatMap, sample, groupByKey, reduceByKey, union, join, cogroup, mapValues,sort,partionBy
  - actions操作：Count,collect, reduce, lookup, save。
  - 处理节点之间通信模型不再像Hadoop，就是唯一的Data Shuffle一种模式。
  - 可以命名，物化，控制中间结果的存储、分区等。
- 容错性 checkpoint缓冲机制，在分布式数据集计算时通过checkpoint来实现容错，而checkpoint有两种方式，一个是checkpoint data，一个是logging the updates。用户可以控制采用哪种方式来实现容错。
- Spark更适合于迭代运算比较多的ML和DM运算。因为在Spark里面，有RDD的抽象概念。
- 对机器学习算法，图计算能力有很好的支持。

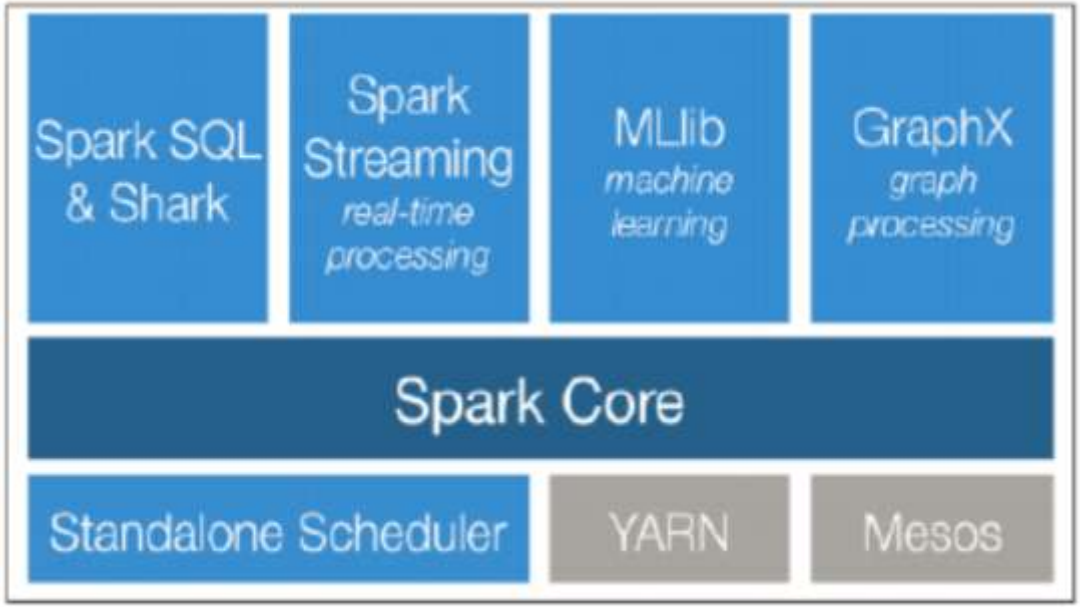
Spark VS MapReduce

	Spark	Hadoop MapRecuce
架构	Spark+ RDD RDD： 由 Spark 内部维护的、基于内存的分布式数据集	MapReduce + HDFS HDFS：分布式文件系统
工作量	面向函数编程 需要提供Map/Reduce函数。	面向对象编程 需要提供Map/Reduce类。
数据处理	RDD 保存Map操作的结果，支持多次 Map 迭代。 Map 计算懒加载，用到时才发生计算	Map、Reduce成对出现。 Reduce 结果落地后才能被下次 Map 使用
故障处理	多主多备 集成HDFS不会有数据丢失，其他情况会有丢失情况； standalone 启动模式 Driver 节点不能自动恢复，任务需要重新提交；	依赖 HDFS 能快速恢复计算节点

Spark生态圈

以Spark Core为核心从HDFS、Amazon S3和HBase等持久层读取数据，Spark和yarn结合容易，以MESOS、YARN和自身携带的Standalone为资源管理器调度Job完成Spark应用程序的计算。

应用程序可以来自于不同的组件，如Spark Shell/Spark Submit批处理、Spark Streaming实时处理应用、Spark SQL即席查询、BlinkDB权衡查询、MLlib/MLbase机器学习、GraphX图处理和SparkR的数学计算等等。



Spark 对于资源管理与作业调度可以使用Standalone(独立模式)，Apache Mesos及Hadoop YARN来实现。Spark on Yarn遵循YARN的官方规范实现，得益于Spark天生支持多种Scheduler和Executor的良好设计，对YARN的支持也就非常容易。让Spark运行于YARN上与Hadoop共用集群资源可以提高资源利用率。

Spark在hadoop生态体系中的位置

Spark应用及应用场景

- ✓ Spark是基于内存的迭代计算框架，适用于需要多次操作特定数据集的应用场合。需要反复操作的次数越多，所需读取的数据量越大，受益越大，数据量小但是计算密集度较大的场合，受益就相对较小。
- ✓ 由于RDD的特性，Spark不适用那种异步细粒度更新状态的应用，例如web服务的存储或者是增量的web爬虫和索引。就是对于那种增量修改的应用模型不适合。

Spark应用及应用场景

- ✓ 腾讯
- ✓ Yahoo
- ✓ 淘宝
- ✓ 优酷土豆
- ✓ 迭代算法，包括大部分机器学习算法Machine Learning和比如PageRank的图形算法。
- ✓ 交互式数据挖掘，用户大部分情况都会大量重复的使用导入RAM的数据（R、Excel、python）
- ✓ 需要持续长时间维护状态聚合的流式计算。

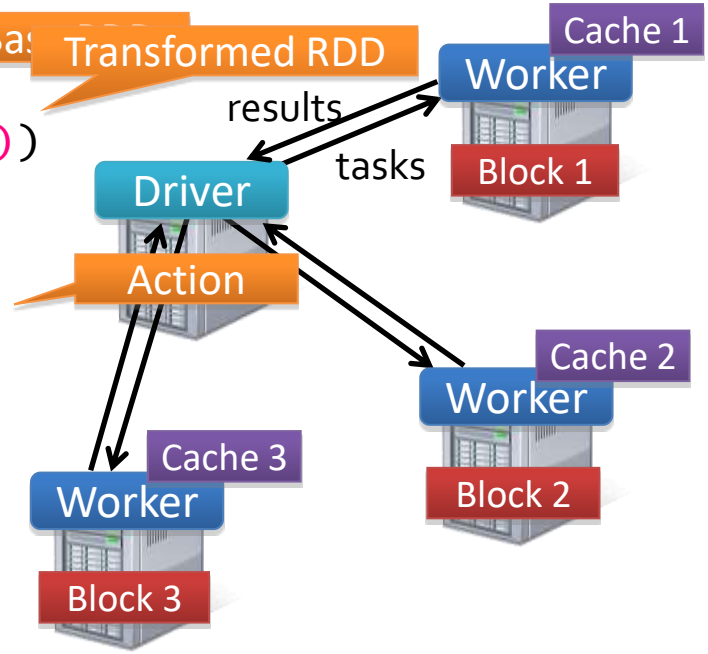
Spark架构

- 基于Hadoop的HDFS，Spark采用Driver、Worker的主从结构，由Driver节点调度，负责任务分配、资源安排、结果汇总、容错等处理。Worker节点主要是存放数据和进行计算。
- 第一次从外设读取数据，之后主要在内存计算。

Spark架构

- 案例中涉及到RDD、Transformation、Action等操作，从中可以发现，处理方式是MR+SQL语法，包括map、reduce、count、groupby、join、union等

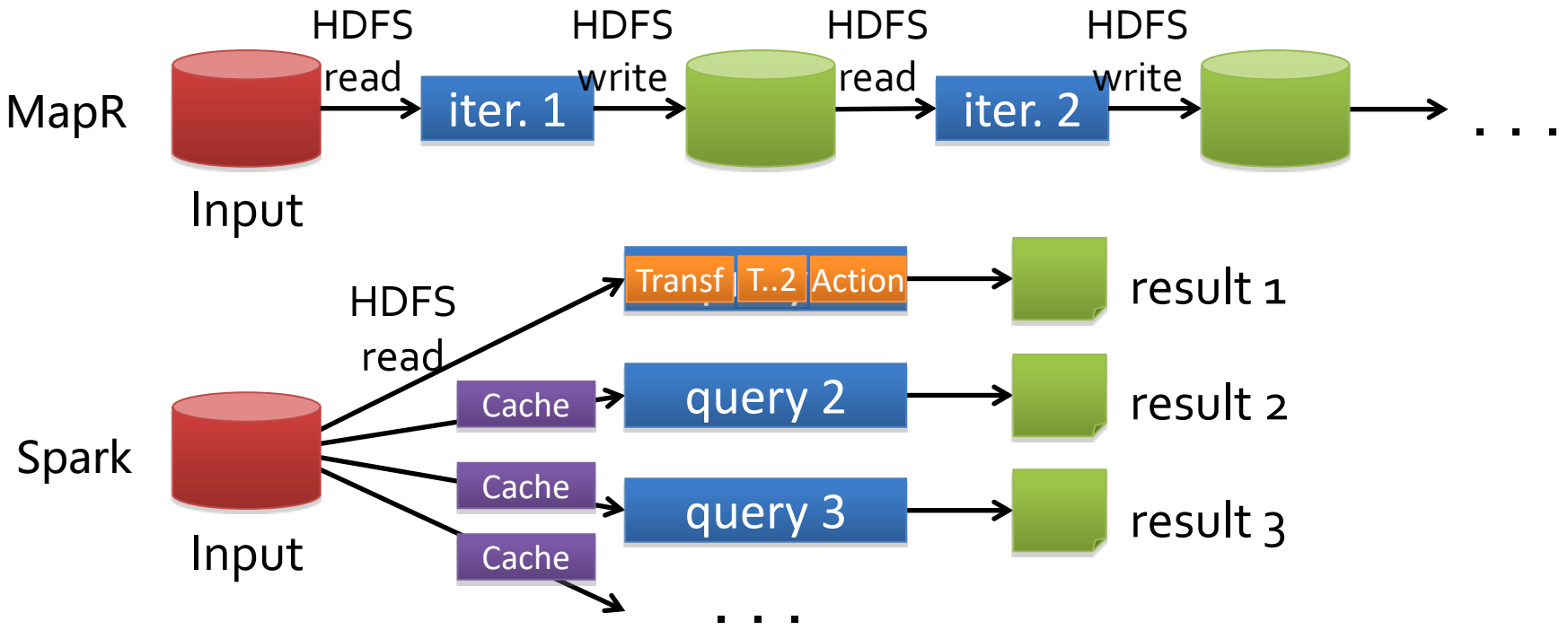
```
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
messages = errors.map(_.split('\t')(2))
cachedMsgs = messages.cache()
cachedMsgs.filter(_.contains("foo")).count
cachedMsgs.filter(_.contains("bar")).count
. . .
```



Result: scaled to 1 TB data in 5-7 sec
(vs 170 sec for on-disk data)

Spark架构

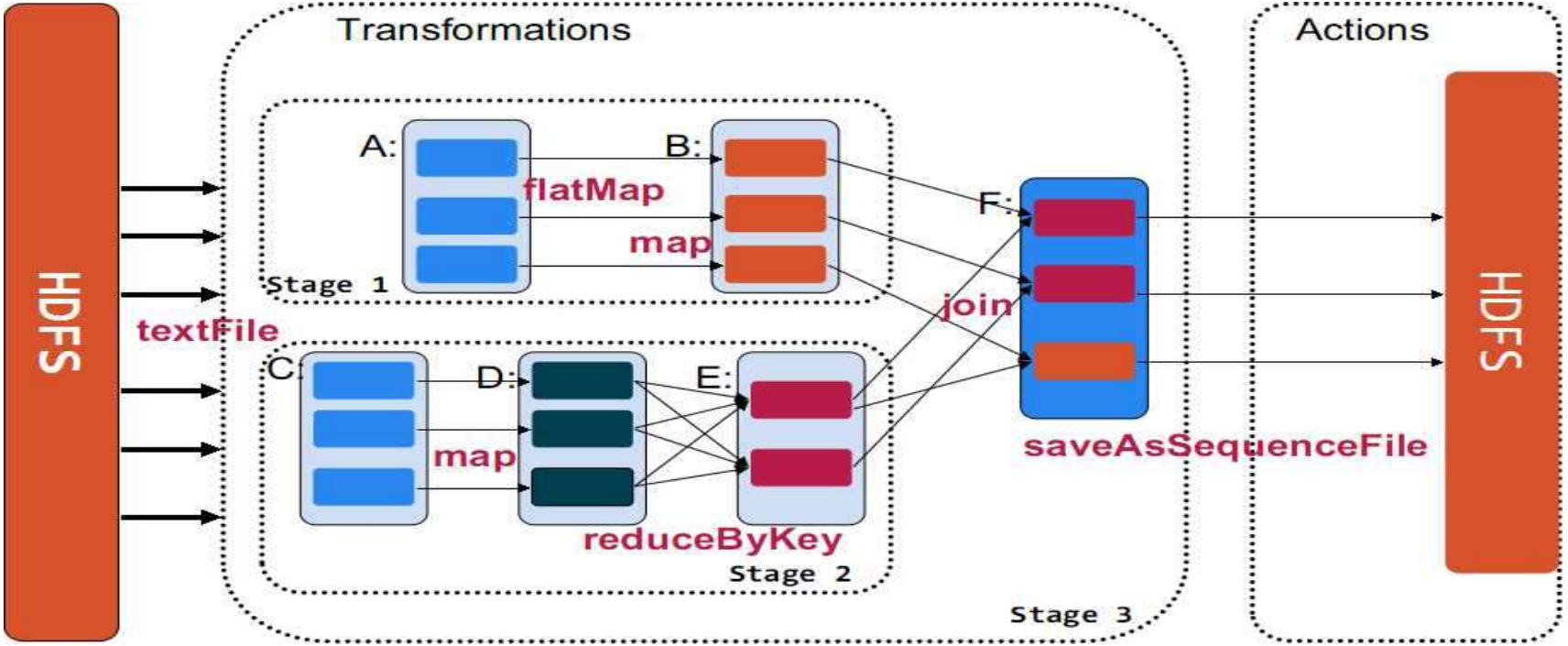
- MapReduce每次读写，都需要序列化到磁盘。一个复杂任务，需要多次处理，几十次磁盘读写。
- Spark只需要一次磁盘读写，大部分处理在内存中进行。



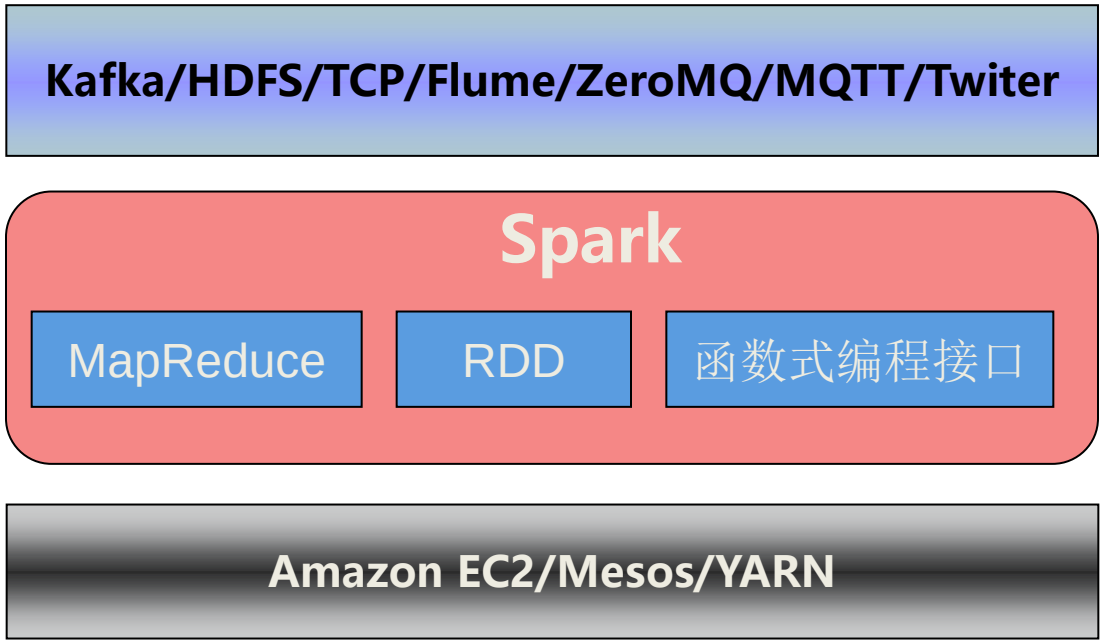
I/O and serialization can take **90%** of the time

Spark架构

- ✓ 弹性分布式RDD
- ✓ 提供了支持DAG图的分布式并行计算框架，减少多次计算之间中间结果IO开销
- ✓ 提供Cache机制来支持多次迭代计算或者数据共享，减少IO开销
- ✓ RDD之间维护了血统关系，一旦RDD fail掉了，能通过父RDD自动重建，保证了容错性
- ✓ 移动计算而非移动数据，RDD Partition可以就近读取分布式文件系统中的数据块到各个节点内存中进行计算
- ✓ 使用多线程池模型来减少task启动开稍
- ✓ shuffle过程中避免不必要的sort操作
- ✓ 采用容错的、高可伸缩性的akka作为通讯框架



Spark架构



由Scala编写，支持函数式编程。

支持多种数据源接入。

RDD-弹性分布式数据集，Spark将数据分布到多台机器的内存中进行并行计算。

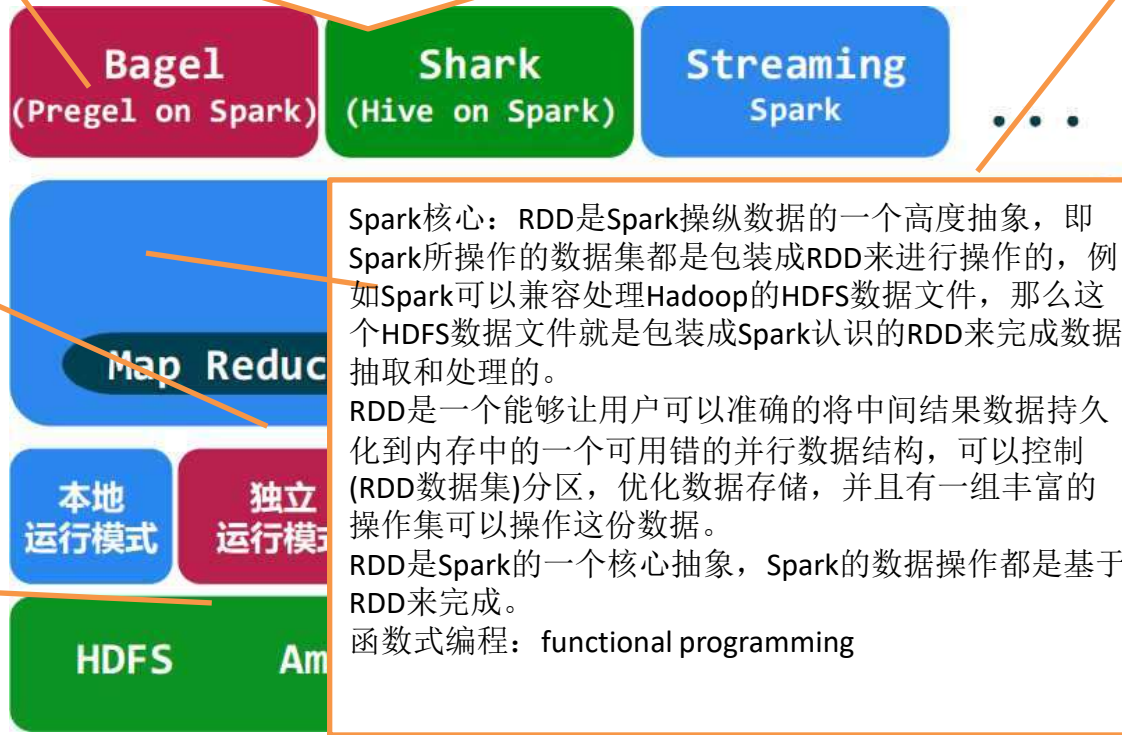
Spark 不具备集群管理能力，需要别的软件进行管理。

支持流式运算，可以从kafka等数据源不断的获取数据，并按时间切片处理。

Bagel(pregel on spark)：Bagel是基于Spark的轻量级的Pregel(Pregel是Google鼎鼎有名的图计算框架)的实现

Shark是基于Spark计算框架之上且兼容Hive语法的SQL执行引擎，Shark基本上就是在Spark的框架基础上提供和Hive一样的HiveQL命令接口。可以理解为Shark On Spark，就是Hive On Hadoop，两者地位是一样的。Shark可以通过UDF用户自定义函数实现特定的数据分析学习算法，使得SQL数据查询和运算分析功能结合在一起（最大化RDD的重复使用）。由于底层的计算采用了Spark，性能比MapReduce的Hive普遍快2倍以上，如果是纯内存计算的SQL，要快5倍以上，当数据全部load在内存的话，将快10倍以上，因此Shark可以作为交互式查询应用服务来使用。

Spark Streaming是构建在Spark上的处理实时数据的框架。其基本原理是将Stream数据分成小的时间片段（几秒），以类似batch批处理的方式来处理小部分数据。



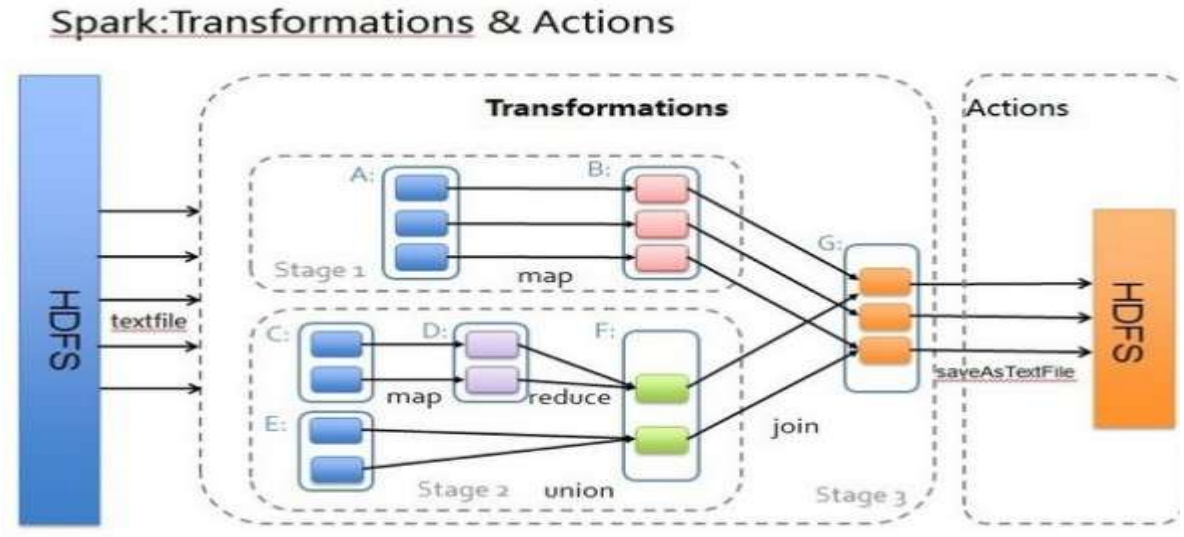
Apace spark分布式部署方式

Spark核心：RDD是Spark操纵数据的一个高度抽象，即Spark所操作的数据集都是包装成RDD来进行操作的，例如Spark可以兼容处理Hadoop的HDFS数据文件，那么这个HDFS数据文件就是包装成Spark认识的RDD来完成数据抽取和处理的。
RDD是一个能够让用户可以准确的将中间结果数据持久化到内存中的一个可用错的并行数据结构，可以控制(RDD数据集)分区，优化数据存储，并且有一组丰富的操作集可以操作这份数据。
RDD是Spark的一个核心抽象，Spark的数据操作都是基于RDD来完成。
函数式编程：functional programming

数据存储方式

Spark架构

Map Reduce：MR 是Spark可以支撑的运算模式，比传统的Hadoop MR的性能更好，并且操作集更加丰富。Spark的MR计算引擎的架构图：



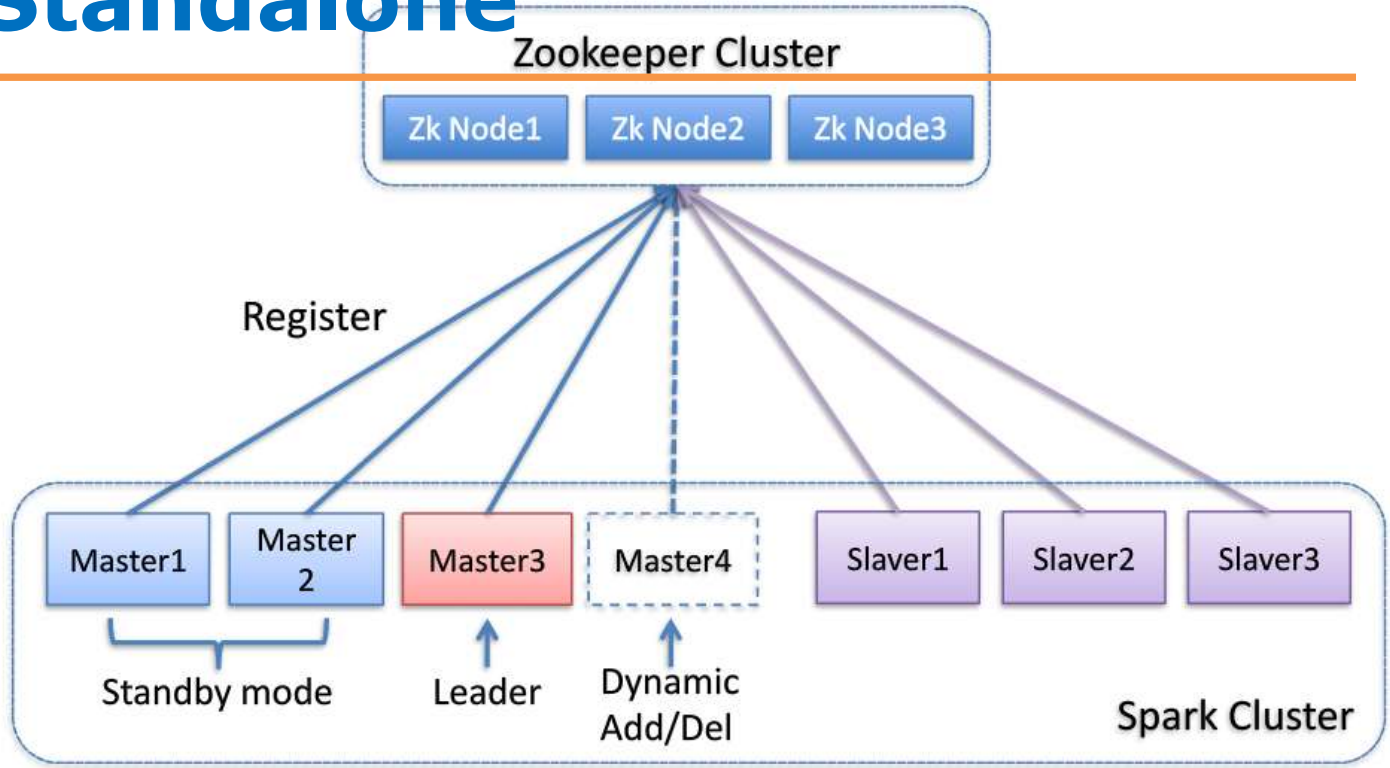
Spark的运行模式

- ✓ Standalone模式
  - ✧ 独立模式，自己负责资源调度。单点故障借助 zookeeper实现。
- ✓ Yarn模式
  - ✧ 利用yarn，资源由Yarn负责管理，最有前景的部署模式，支持动态添加资源。但是限于YARN自身发展，目前仅支持粗粒度模式。
- ✓ Mesos模式
  - ✧ Spark运行在Mesos上，支持 CPU 非独占，资源由Mesos负责管理。Mesos模式调度方式有两种：粗粒度和细粒度
- ✓ Cloud模式
  - ✧ 如 AWS的EC2，使用这种模式，访问Amazon的S3很方便。

Spark on Standalone

- ✓ 独立模式，自带完整的服务，可单独部署到一个集群中，无需依赖任何其他资源管理系统。从一定程度上说，该模式是其他两种的基础。
- ✓ 各个节点上的资源被抽象成粗粒度的slot，有多少slot就能同时运行多少task。
- ✓ 不同的是，MapReduce将slot分为map slot和reduce slot，它们分别只能供Map Task和Reduce Task使用，而不能共享，这是MapReduce资源利率低效的原因之一，而Spark则更优化一些，它不区分slot类型，只有一种slot，可以供各种类型的Task使用，这种方式可以提高资源利用率，但是不够灵活，不能为不同类型的Task定制slot资源。总之，这两种方式各有优缺点。

Spark on Standalone



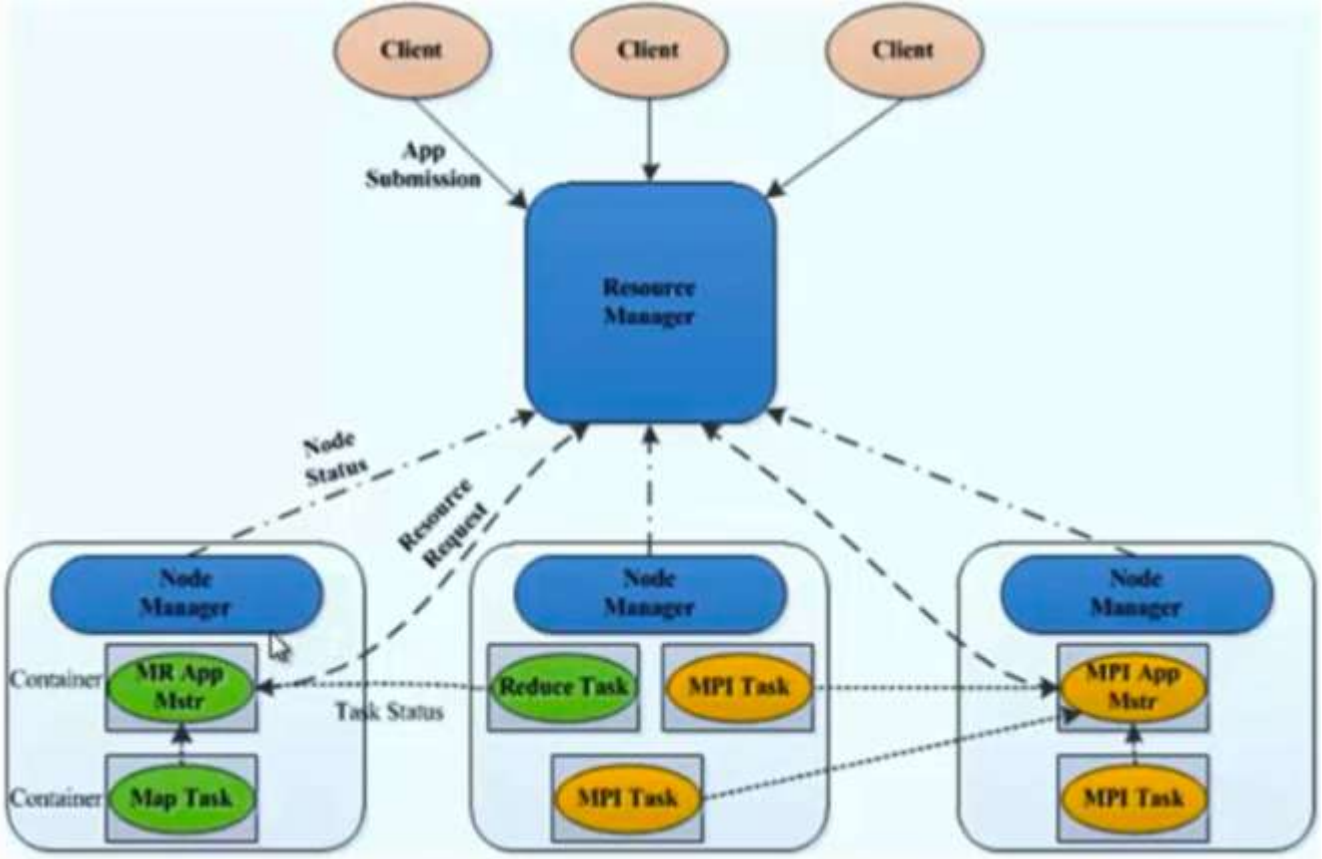
启动多个Master并注册到 Zookeeper 集群中，并保存状态。其中一个会被选为Leader，其余的保持Standby模式，当Leader故障，则选择另一个 Master 为Leader,并从Zookeeper中读取状态恢复。Master节点可动态添加或删除

Spark on Mesos

- ✓ 官方推荐这种模式。正是由于Spark开发之初就考虑到支持Mesos，因此，目前而言，Spark运行在Mesos上会比运行在YARN上更加灵活，更加自然。

Spark on Yarn

- ✓ 一种最有前景的部署模式。但限于YARN自身的发展，目前仅支持粗粒度模式。这是由于YARN上的Container资源是不可以动态伸缩的，一旦Container启动之后，可使用的资源不能再发生变化。



Spark的运行模式

local	本地模式	常用于本地开发测试，本地还分为local和local-cluster
standalone	集群模式	典型的Master/Slave模式，不过也能看出Master是有单点故障的，Spark支持Zookeeper来实现HA
On YARN	集群模式	运行在Yarn资源管理器框架之上，由Yarn负责资源管理，Spark负责任务调度和计算
On Mesos	集群模式	运行在mesos资源管理器框架之上，由mesos负责资源管理，Spark负责任务调度和计算
On cloud	集群模式	比如Aws的EC2，使用这个模式能很方便地访问Amazon的S3；Spark支持多种分布式存储系统，hdfs和S3、hbase等

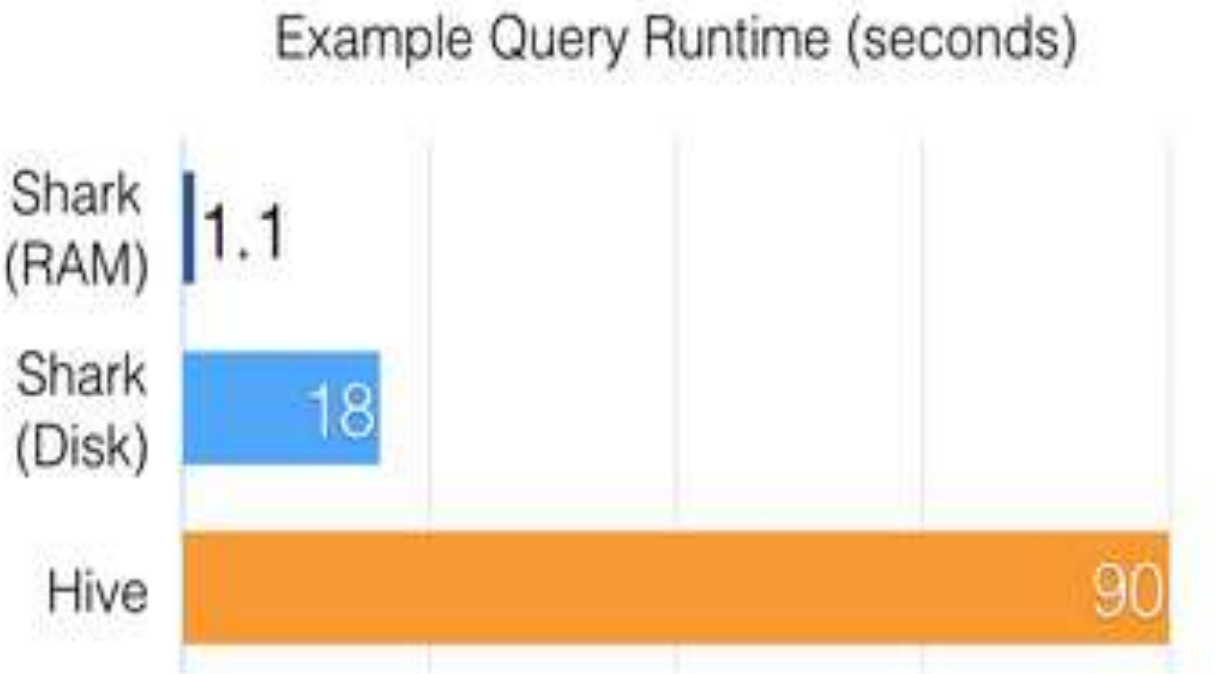
Spark的运行模式

apache Mesos和YARN计算两套资源管理框架，Spark最初设计就是跑在这两个资源管理框架之上的，至于Spark的本地运行模式和独立运行模式则是方便了调试。YARN资源管理框架也是Hadoop2.0的产物，大大优化了传统Hadoop通过JobTracker和TaskTracker来调度计算任务的方式，使集群更加平台化，可以部署多中计算引擎，比如传统的Hadoop MR和Spark都可以跑在同一个集群上，YARN这类资源管理框架出现之前是做不到的。

这几种分布式部署方式各有利弊，通常需要根据公司情况决定采用哪种方案。进行方案选择时，往往要考虑公司的技术路线（采用Hadoop生态系统还是其他生态系统）、服务器资源（资源有限的话就不要考虑standalone模式了）、相关技术人才储备等。

shark与hive对比

- Shark是在Spark的框架基础上提供和Hive一样的HiveQL命令接口，Shark可以自动在内存中缓存特定的RDD，实现数据重用，进而加快特定数据集的检索。
- Shark通过UDF实现特定的数据分析算法，使得SQL数据查询和运算分析能结合在一起，最大化RDD的重复使用



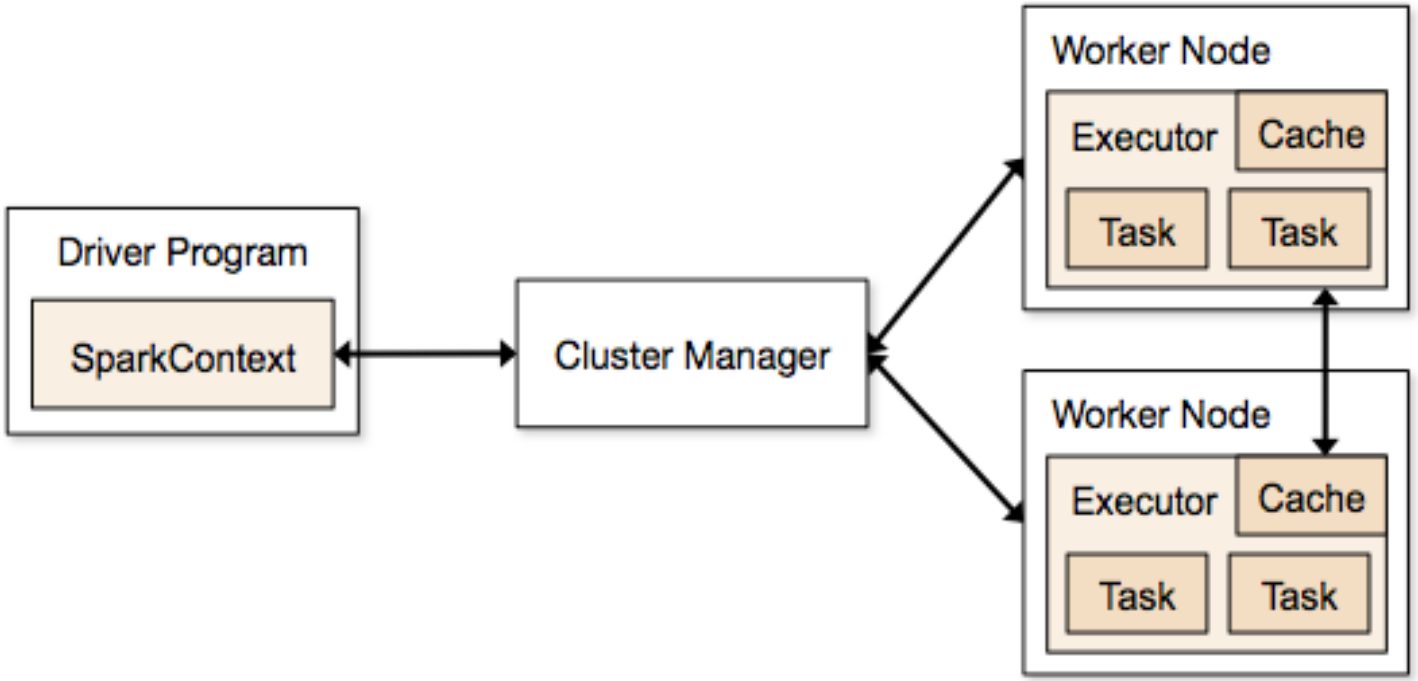
Spark数据的存储

- ✓ Spark支持多种数据底层存储，这点比Hadoop支持的数据文件格式广泛的多。Spark可以兼容HDFS，Hbase,Amazon S3等多种数据集，将这些数据集封装成RDD进行操作

Spark集群的运行方式

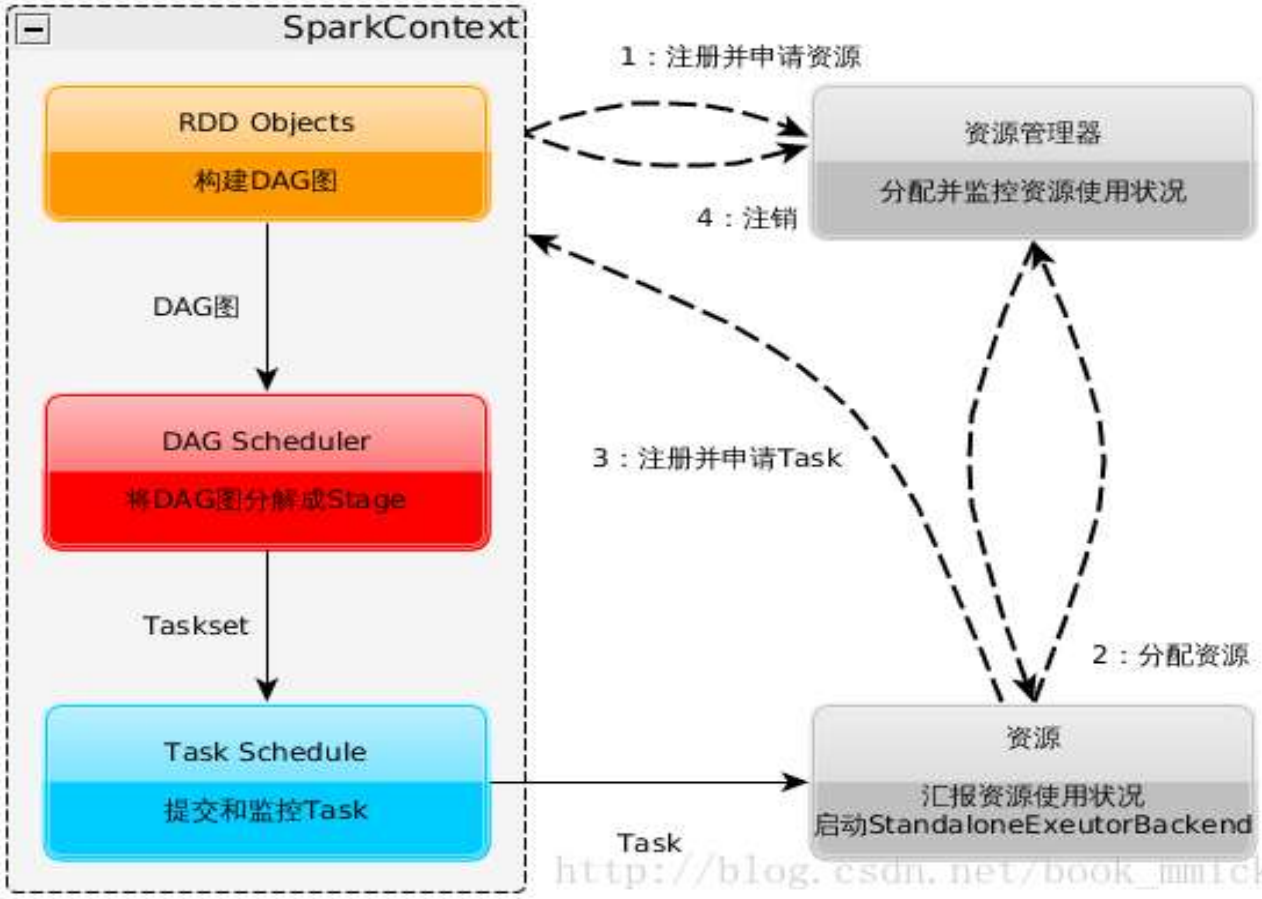
- Client 可以在集群内也可以在集群外
- 1个 application 包含1个 driver 和多个 executor
- 1个 executor 里面运行的 tasks 都属于同1个application
- Spark应用程序都离不开SparkContext和Executor两部分，Executor负责执行任务，运行Executor的机器称为Worker节点，SparkContext由用户程序启动，通过资源调度模块和Executor通信。SparkContext和Executor这两部分的核心代码实现在各种运行模式中都是公用的，在它们之上，根据运行部署模式的不同，包装了不同调度模块以及相关的适配代码。具体来说，以SparkContext为程序运行的总入口，在SparkContext的初始化过程中，Spark会分别创建DAGScheduler作业调度和TaskScheduler任务调度两级调度模块。
- 其中作业调度模块是基于任务阶段的高层调度模块，它为每个Spark作业计算具有依赖关系的多个调度阶段（通常根据shuffle来划分），然后为每个阶段构建出一组具体的任务（通常会考虑数据的本地性等），然后以TaskSets（任务组）的形式提交给任务调度模块来具体执行。而任务调度模块则负责具体启动任务、监控和汇报任务运行情况。
-

Spark集群的运行方式



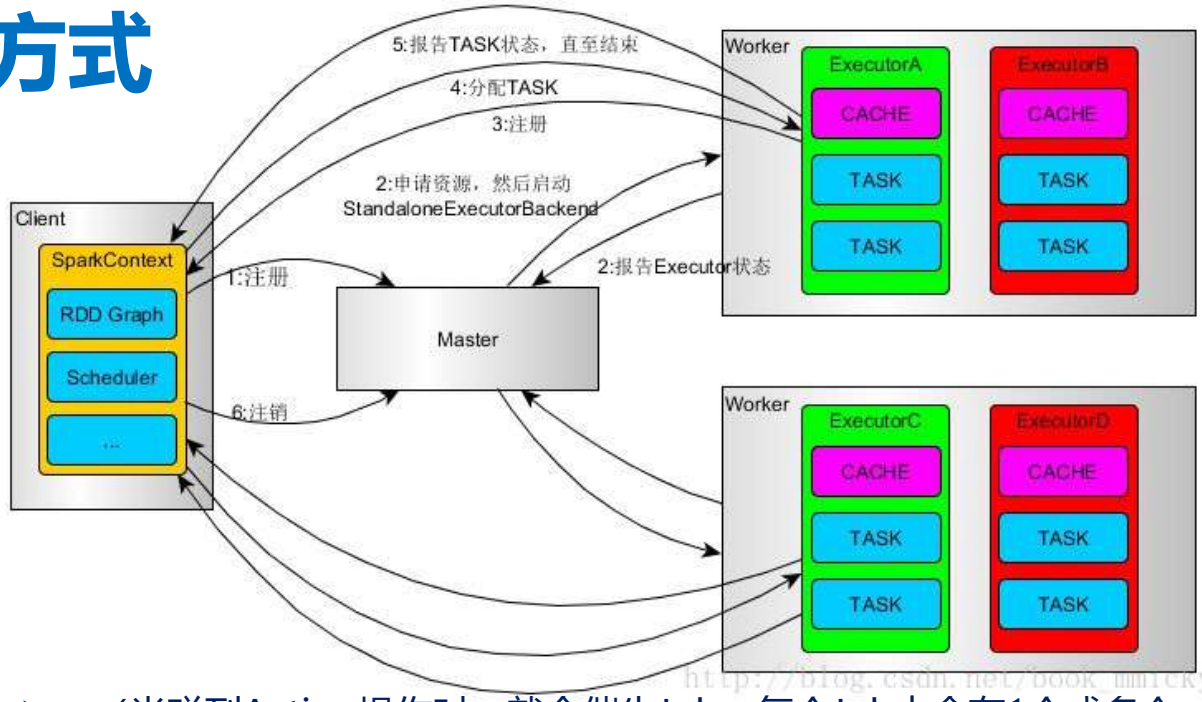
Spark集群的运行方式

- 构建Spark Application的运行环境（启动SparkContext）
- SparkContext向资源管理器（可以Standalone、Mesos、Yarn）申请运行Executor资源，并启动StandaloneExecutorBackend，executor向SparkContext申请Task。
- SparkContext将应用程序代码发放给executor
- SparkContext构建成DAG图、将DAG图分解成Stage、将Taskset发送给Task Scheduler、最后由Task Scheduler将Task发放给Executor运行。
- Task在Executor上运行，运行完毕释放所有资源。



Spark集群的运行方式

- Spark on Standalone运行过程（client模式）
- SparkContext连接到Master，向Master注册并申请资源（CPU Core 和Memory）
- Master根据SparkContext的资源申请要求和worker心跳周期内报告的信息决定在哪个worker上分配资源，然后在该worker上获取资源，然后启动StandaloneExecutorBackend。
- StandaloneExecutorBackend向SparkContext注册
- SparkContext将Applicaiton代码发送给StandaloneExecutorBackend；并且SparkContext解析Applicaiton代码，构建DAG图，并提交给DAG Scheduler分解成Stage



- （当碰到Action操作时，就会催生Job；每个Job中含有1个或多个Stage，Stage一般在获取外部数据和shu f f le之前产生），然后以Stage（或者称为TaskSet）提交给Task Scheduler，Task Scheduler负责将Task分配到相应的worker，最后提交给StandaloneExecutorBackend执行；
- StandaloneExecutorBackend会建立executor 线程池，开始执行Task，并向SparkContext报告，直至Task完成。
- 所有Task完成后，SparkContext向Master注销，释放资源。

Spark目录

- core Spark核心代码都在此目录下
- Sql Spark sql相关的代码
- Streaming Spark Streaming（实时计算）代码
- mlib MLib（机器学习）相关代码
- Graphx GraphX（图计算）相关代码
- yarn 支持Spark运行在Yarn上的模块
- Example 各种spark作业的例子
- Assembly 组装spark项目的地方
- ec2 提交spark集群到Amazon EC2
- External 与一些外部系统的依赖
- extra 包含了spark默认不构建的组件
- repl Spark shell功能模块
- tools 工具包

Spark集群的运行方式

1、用户通过bin/spark-submit或bin/spark-class 向YARN提交Application。

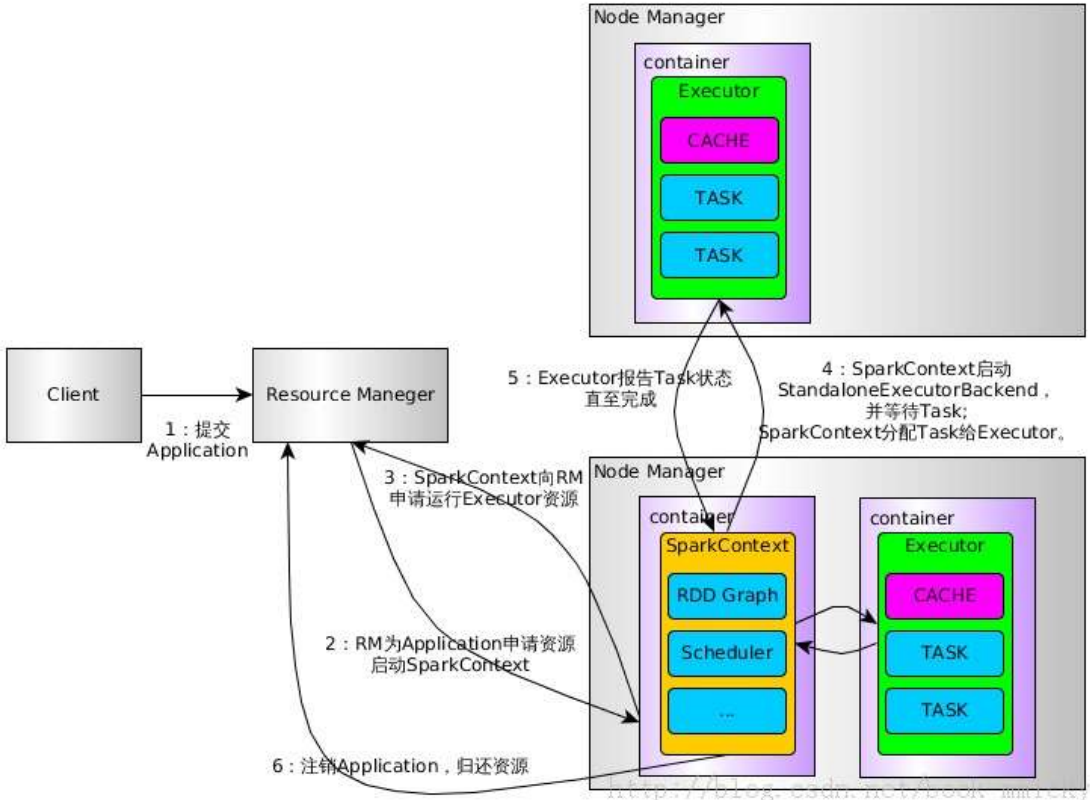
2、RM为Application分配第一个Container，并在指定节点的container上启动SparkContext。

3、SparkContext向RM申请资源以运行Executor。

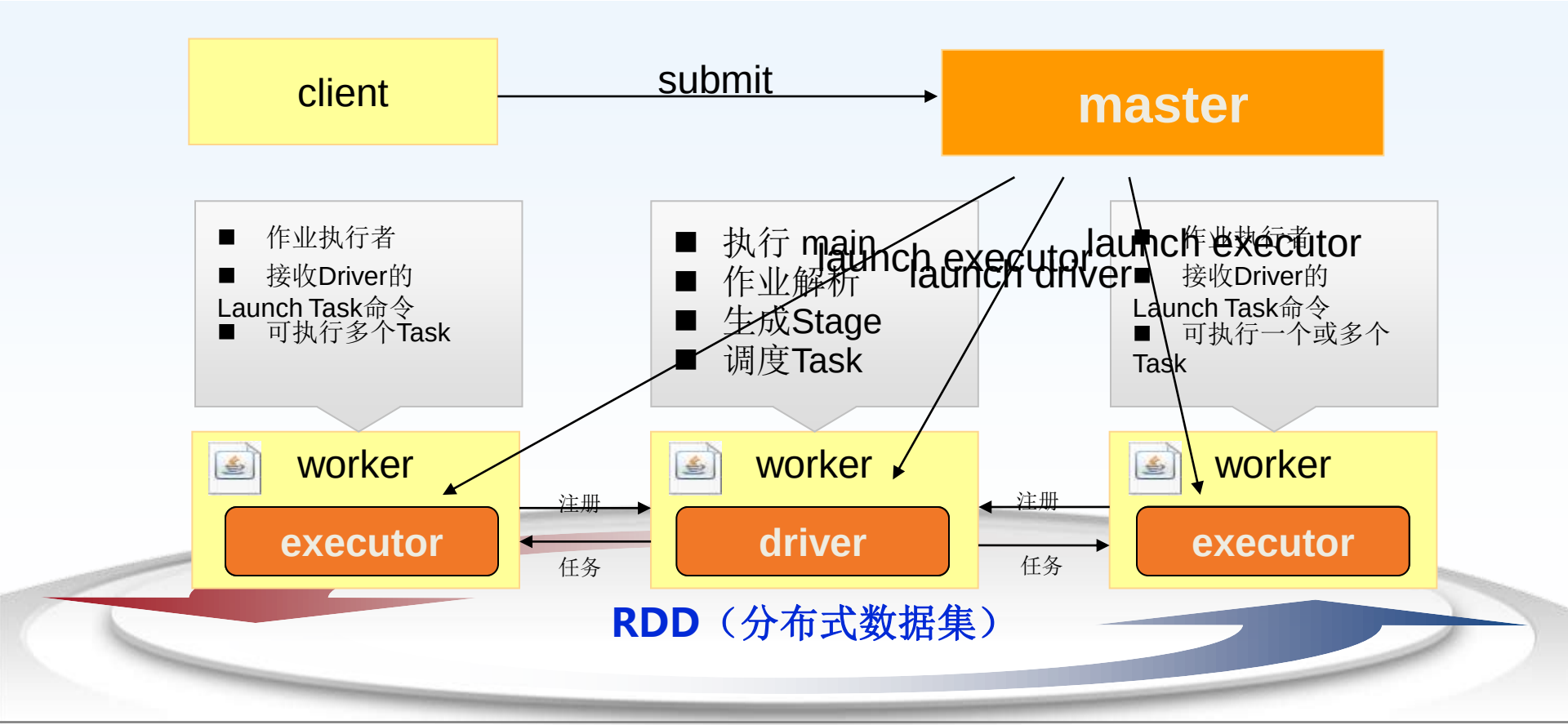
4、RM分配Container给SparkContext，SparkContext和相关的NM通讯，在获得的Container上启动StandaloneExecutorBackend，StandaloneExecutorBackend启动后，开始向SparkContext注册并申请Task。

5、SparkContext分配Task给StandaloneExecutorBackend执行StandaloneExecutorBackend**执行Task**并向SparkContext汇报运行状况

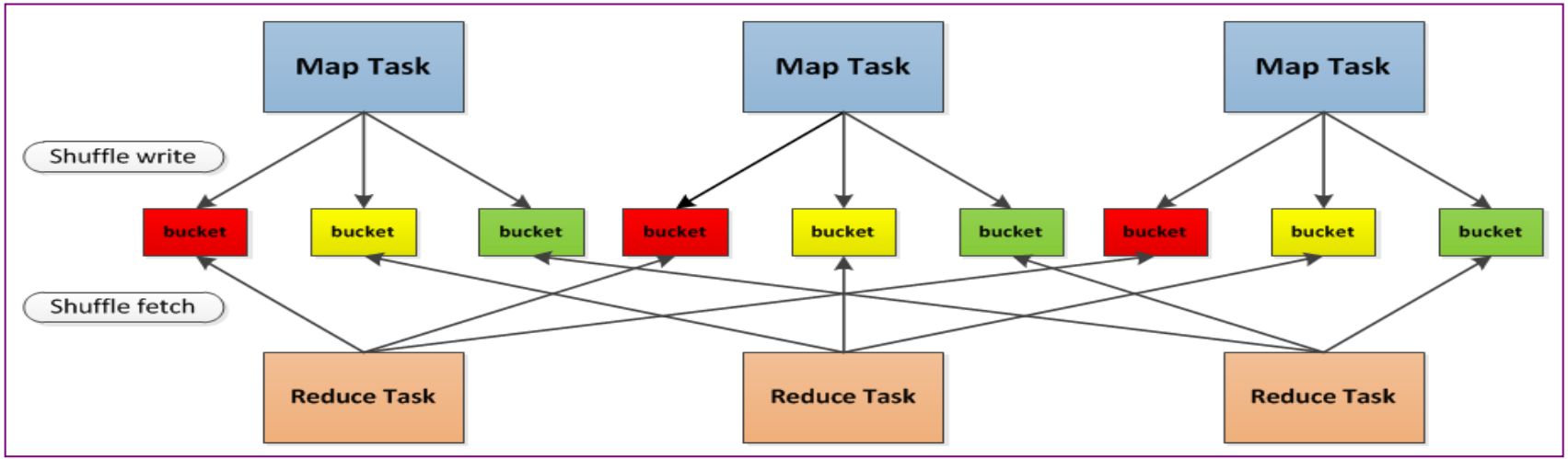
6、Task运行完毕，SparkContext归还资源给RM，并注销退出。。



Spark集群的运行方式



Spark Shuffle



1、每一个Mapper会根据Reducer的数量创建出相应的bucket，bucket的数量是M×R，其中M是Map的个数，R是Reduce的个数。
2、Mapper产生的结果会根据设置的partition算法填充到每个bucket中去。这里的partition算法是可以自定义的，当然默认的算法是根据key哈希到不同的bucket中去。
3、当Reducer启动时，它会根据自己task的id和所依赖的Mapper的id从远端或是本地的block manager中取得相应的bucket作为Reducer的输入进行处理。

RDD是什么

Resilient Distributed Dataset (RDD):

- 分布式数据集(分布在集群、已分区的)
- 存储在内存或磁盘上
- 只读的，不可变
- 自动重建(容错)
- 有两类操作
  - transformations
  - actions

RDD是什么

RDD是Spark操纵数据的一个高度抽象，Spark操作的数据集都是包装成RDD来进行操作的，例如Spark可以兼容处理Hadoop的HDFS数据文件，这个HDFS数据文件就是包装成Spark认识的RDD来完成数据抽取和处理

RDD是一个只读的分区存储集合。只能基于稳定物理存储中的数据集或在已有的RDD上执行转换命令（Transformation）来创建。

RDD不需要物化。在创建 RDD 时Spark会维护转换算法。需要使用时，可以从物理存储的数据计算出最终的 RDD。

Spark操纵数据的一个高度抽象，是数据抽取和处理的基础。

Transformations 是一个懒加载过程，在需要用来计算的时候，才会触发一系列的 RDD转换逻辑

RDD是什么

有一个分片列表。就是能被切分，和hadoop一样的能够切分的数据才能并行计算。

有一个函数计算每一个分片， 每一个split调用一次map函数

对其他的RDD的依赖列表，也就是rdd演变，进化。依赖分为宽依赖和窄依赖，并不是所有的RDD都有依赖。

可选：key-value型的RDD是根据哈希来分区的，类似于mapreduce当中的Paritioner接口，控制key分到哪个reduce。

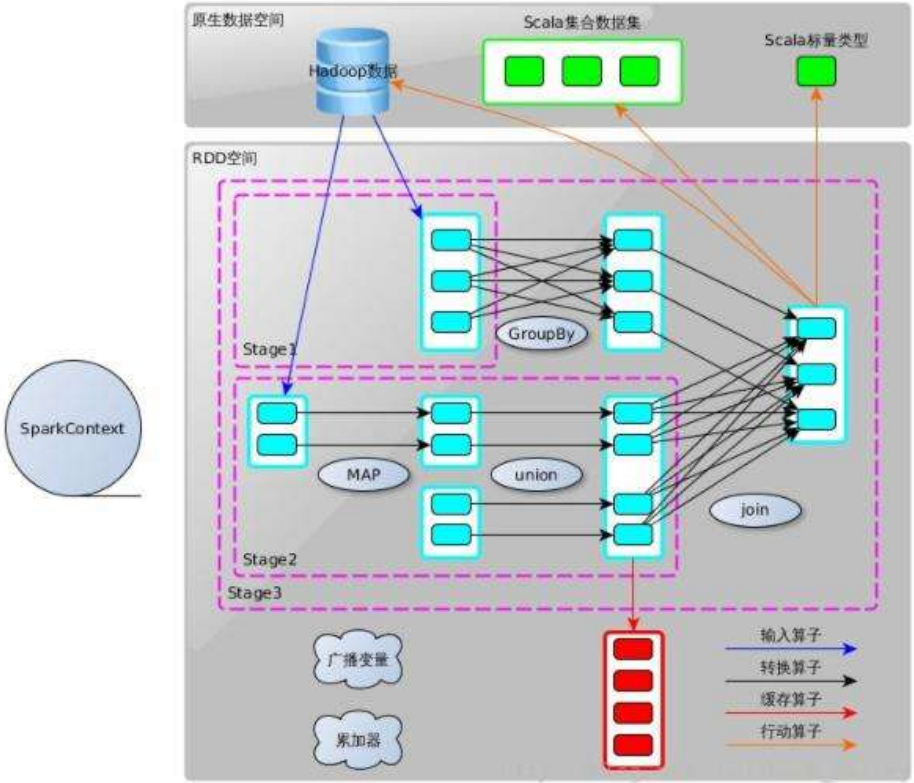
可选：为每一个分片的分配最优先计算位置（preferred locations），比如HDFS的block的所在位置应该是优先计算的位置。

RDD创建

- 由于RDD提供的是只读的共享内存，所以不能直接创建。有两种方法可以创建RDDS
- （1）并行　驱动程序为现有的集合
- （2）引用在外部存储系统，数据集，如共享文件系统，HDFS，HBase的，或任何数据源提供的Hadoop的InputFormat。

RDD特点

- 在集群节点上是不可变的、已分区的集合对象。
- 通过并行转换的方式来创建，如map, filter, join等。
- 失败自动重建。
- 可以控制存储级别（内存、磁盘等）来进行重用。
- 必须是可序列化的。
- 是静态类型的。
- 不变的数据结构存储
- 支持跨集群的分布式数据结构
- 可以根据数据记录的key对结构进行分区
- 提供了粗粒度的操作，这些操作支持分区
- 将数据存储在内存中，提供了低延迟性



RDD特点

- RDD本质是一个内存数据集，在访问RDD时，指针只会指向与操作相关的部分。例如存在一个面向列的数据结构，其中一个实现为Int的数组，另一个实现为Float的数组。如果只需要访问Int字段，RDD的指针可以只访问Int数组，避免了对整个数据结构的扫描。
- RDD将操作分为两类：transformation与action。无论执行了多少次transformation操作，RDD都不会真正执行运算，只有当action操作被执行时，运算才会触发。而在RDD的内部实现机制中，底层接口则是基于迭代器的，从而使得数据访问变得更高效，也避免了大量中间结果对内存的消耗。
  数据复制或日志记录对于以数据为中心的系统而言，这两种方式都非常昂贵
- RDD天生是支持容错的。首先，它自身是一个不变的数据集，其次，它能够记住构建它的操作图，因此当执行任务的Worker失败时，完全可以通过操作图获得之前执行的操作，进行重新计算。由于无需采用replication方式支持容错，很好地降低了跨网络的数据传输成本。
- 在某些场景下，Spark也需要利用记录日志的方式来支持容错。例如，在Spark Streaming中，针对数据进行update操作，或者调用Streaming提供的window操作时，就需要恢复执行过程的中间状态。此时，需要通过Spark提供的checkpoint机制，以支持操作能够从checkpoint得到恢复。

RDD特点

- RDD本质是一个内存数据集，在访问RDD时，指针只会指向与操作相关的部分。例如存在一个面向列的数据结构，其中一个实现为Int的数组，另一个实现为Float的数组。如果只需要访问Int字段，RDD的指针可以只访问Int数组，避免了对整个数据结构的扫描。
- RDD将操作分为两类：transformation与action。无论执行了多少次transformation操作，RDD都不会真正执行运算，只有当action操作被执行时，运算才会触发。而在RDD的内部实现机制中，底层接口则是基于迭代器的，从而使得数据访问变得更高效，也避免了大量中间结果对内存的消耗。
  数据复制或日志记录对于以数据为中心的系统而言，这两种方式都非常昂贵
- RDD天生是支持容错的。首先，它自身是一个不变的数据集，其次，它能够记住构建它的操作图，因此当执行任务的Worker失败时，完全可以通过操作图获得之前执行的操作，进行重新计算。由于无需采用replication方式支持容错，很好地降低了跨网络的数据传输成本。
- 在某些场景下，Spark也需要利用记录日志的方式来支持容错。例如，在Spark Streaming中，针对数据进行update操作，或者调用Streaming提供的window操作时，就需要恢复执行过程的中间状态。此时，需要通过Spark提供的checkpoint机制，以支持操作能够从checkpoint得到恢复。

RDD提供四种算子

RDD 是Spark进行并行运算的基本单位。RDD提供了四种算子：

1、输入算子：将原生数据转换成RDD，如parallelize、txtFile等

2、转换算子：最主要的算子，是Spark生成DAG图的对象。

转换算子并不立即执行，在触发行动算子后再提交给driver处理，生成DAG图–> Stage –> Task –> Worker执行。

按转化算子在DAG图中作用，可以分成两种：

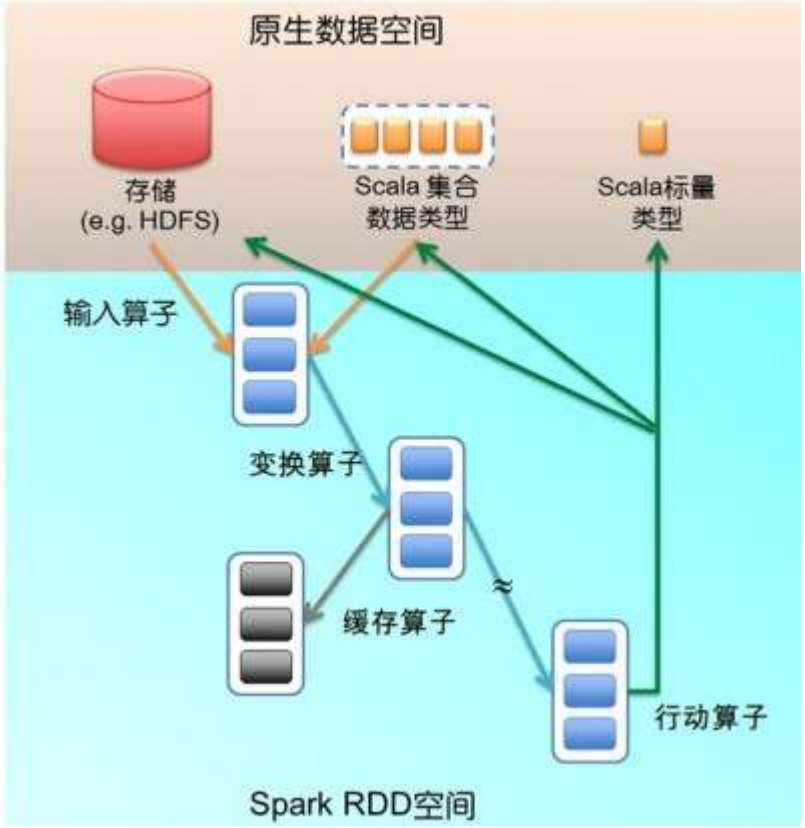
窄依赖算子：输入输出一对一的算子，且结果RDD的分区结构不变，主要是map、flatMap； 输入输出一对一的算子，但结果RDD的分区结构发生了变化，如union、coalesce； 从输入中选择部分元素的算子，如filter、distinct、subtract、sample。

宽依赖算子

宽依赖会涉及shuffle类，在DAG图解析时以此为边界产生Stage。对单个RDD基于key进行重组和reduce，如groupByKey、reduceByKey； 对两个RDD基于key进行join和重组，如join、cogroup。

3、缓存算子：对于要多次使用的RDD，可以缓冲加快运行速度，对重要数据可以采用多备份缓存。

4、行动算子：将运算结果RDD转换成原生数据，如count、reduce、collect、saveAsTextFile等。



RDD支持两种操作

转换：从现有的数据集创建一个新的数据集 。
动作：在数据集上运行计算后，返回一个值给驱动程序
map是一种转换，将数据集每一个元素都传递给函数，并返回一个新的分布数据集表示结果。reduce是一种动作，通过一些函数将所有的元素叠加起来，并将最终结果返回给Driver程序。（不过还有一个并行的reduceByKey，能返回一个分布式数据集）
Spark中的所有转换都是惰性的，他们并不会直接计算结果。相反的，他们只是记住应用到基础数据集（例如一个文件）上的这些转换动作。只有当发生一个要求返回结果给Driver的动作时，这些转换才会真正运行。这个设计让Spark更加有效率的运行。
例如可以实现：通过map创建的一个新数据集，并在reduce中使用，最终只返回reduce的结果给driver，而不是整个大的新数据集。

转换	map(f:T=>U)	: RDD[T] => RDD[U]
	filter(f:T=>Bool)	: RDD[T] => RDD[T]
	flatMap(f:T=>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)=>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) W)	: RDD[(K,V)] => RDD[(K,W)](Preserver partitioning)
	sort(c:Comparator[K])	: RDD[(K,V)] => RDD[(K,V)]
	partitionBy(p:Partitioner[K])	: RDD[(K,V)] => RDD[(K,V)]
动作	count()	: RDD[T] => Long
	collect()	: RDD[T] => Seq[T]
	Reduce(f:(T,T)=>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

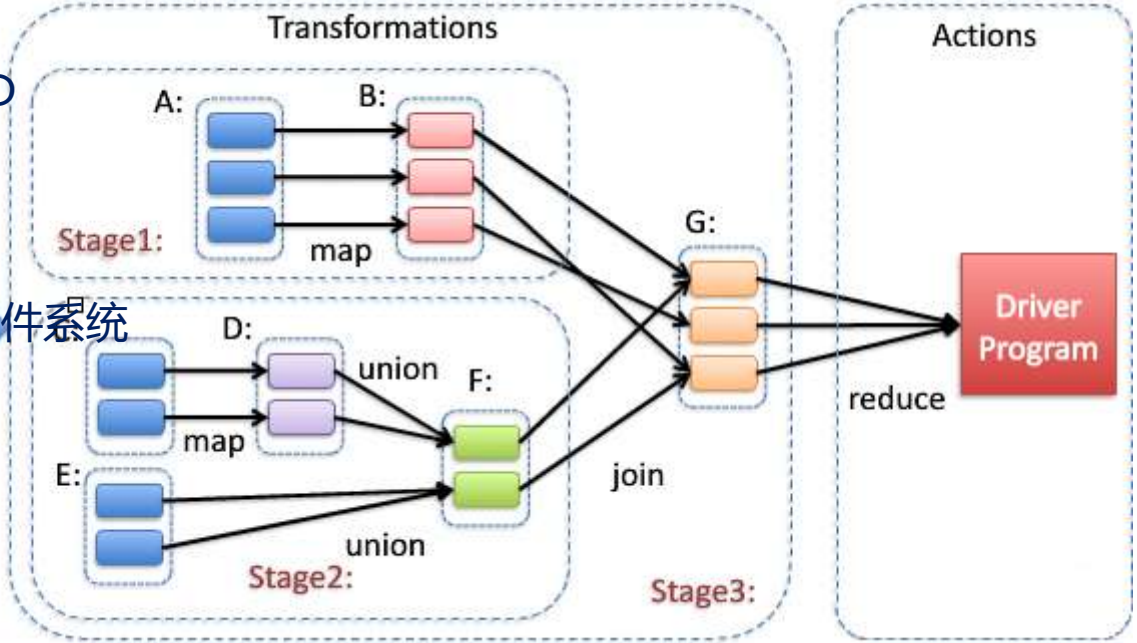
默认下，每一个转换过的RDD都会在它之上执行一个动作时被重新计算。不过，也可以使用persist(或者cache)方法，持久化一个RDD在内存中。在这种情况下，Spark将会在集群中，保存相关元素，下次查询这个RDD时，它将能更快速访问。在磁盘上持久化数据集或在集群间复制数据集也是支持的。

RDD支持两种操作

为了减少网络 IO，Spark 会尽量把连续的任务放在同一台机器上执行。

Transformations	将一个已经存在的RDD中转换成一个新的RDD,所有的转换操作都是lazy执行的。
Actions	一般用于对RDD中的元素进行实际的计算,然后返回相应的值。

- 转换
  - 通过其他RDD构建新的RDD
  - map, filter, join
  - lazy operation
- 动作
  - 返回计算结果或者保存到文件系统
  - count, collect, save
  - triggers execution



RDD转换

Basic RDD transformations on an RDD containing {1, 2, 3, 3}

Function Name	Purpose	Example	Result
map	Apply a function to each element in the RDD and return an RDD of the result	rdd.map(x => x + 1)	{2, 3, 4, 4}
flatMap	Apply a function to each element in the RDD and return an RDD of the contents of the iterators returned. Often used to extract words.	rdd.flatMap(x => x.to(3))	{1, 2, 3, 2, 3, 3, 3}
filter	Return an RDD consisting of only elements which pass the condition passed to filter	rdd.filter(x => x != 1)	{2, 3, 3}
distinct	Remove duplicates	rdd.distinct()	{1, 2, 3}
sample(withReplacement, fraction, [seed])	Sample an RDD	rdd.sample(false, 0.5)	non-deterministic

RDD转换

Two-RDD transformations on RDDs containing {1, 2, 3} and {3, 4, 5}

Function Name	Purpose	Example	Result
union	Produce an RDD contain elements from both RDDs	rdd.union(other)	{1, 2, 3, 3, 4, 5}
intersection	RDD containing only elements found in both RDDs	rdd.intersection(other)	{3}
subtract	Remove the contents of one RDD (e.g. remove training data)	rdd.subtract(other)	{1, 2}
cartesian	Cartesian product with the other RDD	rdd.cartesian(other)	{(1, 3), (1, 4), ... (3,5)}

RDD动作

Basic actions on an RDD containing {1, 2, 3, 3}

Function Name	Purpose	Example	Result
collect()	Return all elements from the RDD	rdd.collect()	{1, 2, 3, 3}
count()	Number of elements in the RDD	rdd.count()	4
take(num)	Return num elements from the RDD	rdd.take(2)	{1, 2}

Pair RDD转换

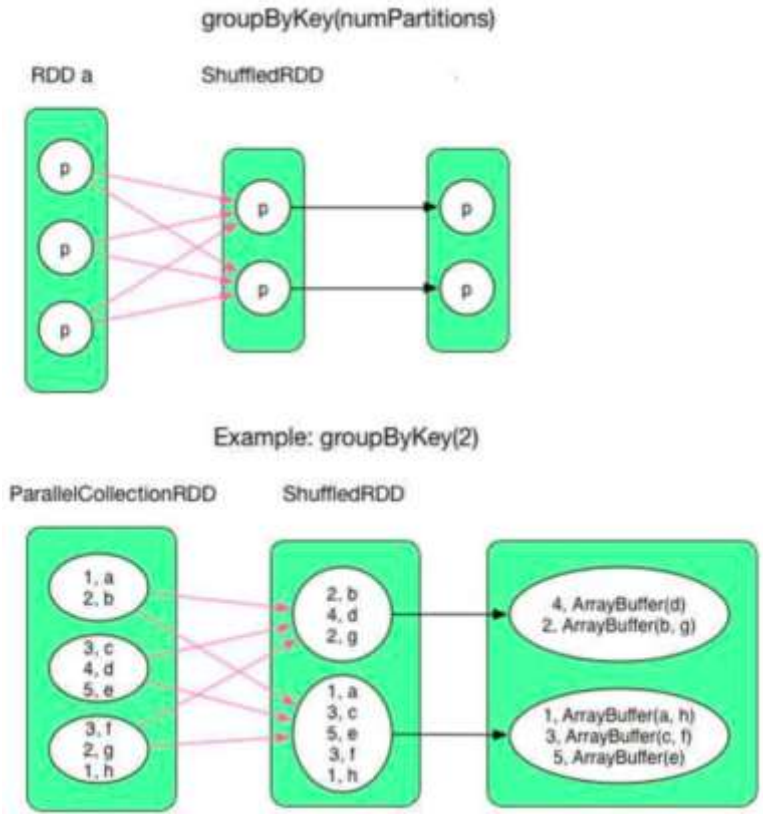
Transformations on one Pair RDD (example ({(1, 2), (3, 4), (3, 6)}))

Function Name	Purpose	Example	Result
combineByKey(createCombiner, mergeValue, mergeCombiners, partitioner)	Combine values with the same key together		
groupByKey()	Group together values with the same key	rdd.groupByKey()	{(1, [2]), (3, [4, 6])}
reduceByKey(func)	Combine values with the same key together	rdd.reduceByKey((x, y) => x + y)	{(1, 2), (3, 10)}
mapValues(func)	Apply a function to each value of a Pair RDD without changing the key	rdd.mapValues(x => x+1)	{(1, 3), (3, 5), (3, 7)}
flatMapValues(func)	Apply a function which returns an iterator to each value of a Pair RDD and for each element returned produce a key-value entry	rdd.flatMapValues(x => x.to(5))	{(1,2), (1,3), (1,4), (1,5), (3, 4),

Pair RDD转换

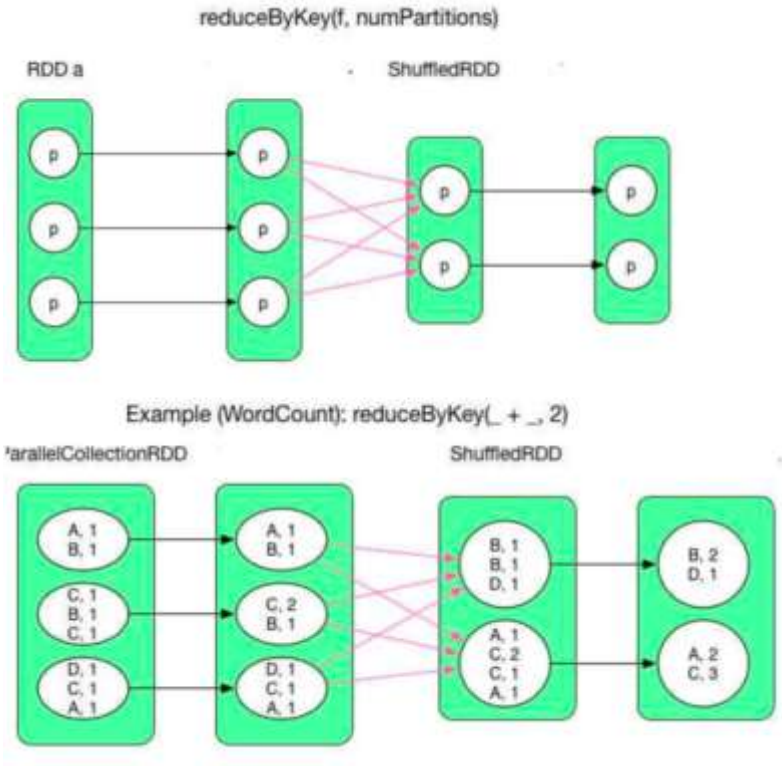
Function Name	Purpose	Example	Result
	with the old key. Often used for tokenization.		(3,5)}
keys()	Return an RDD of just the keys	rdd.keys()	{1, 3, 3}
values()	Return an RDD of just the values	rdd.values()	{2, 4, 6}
sortByKey()	Returns an RDD sorted by the key	rdd.sortByKey()	{(1, 2), (3, 4), (3, 6)}

典型RDD依赖——groupByKey



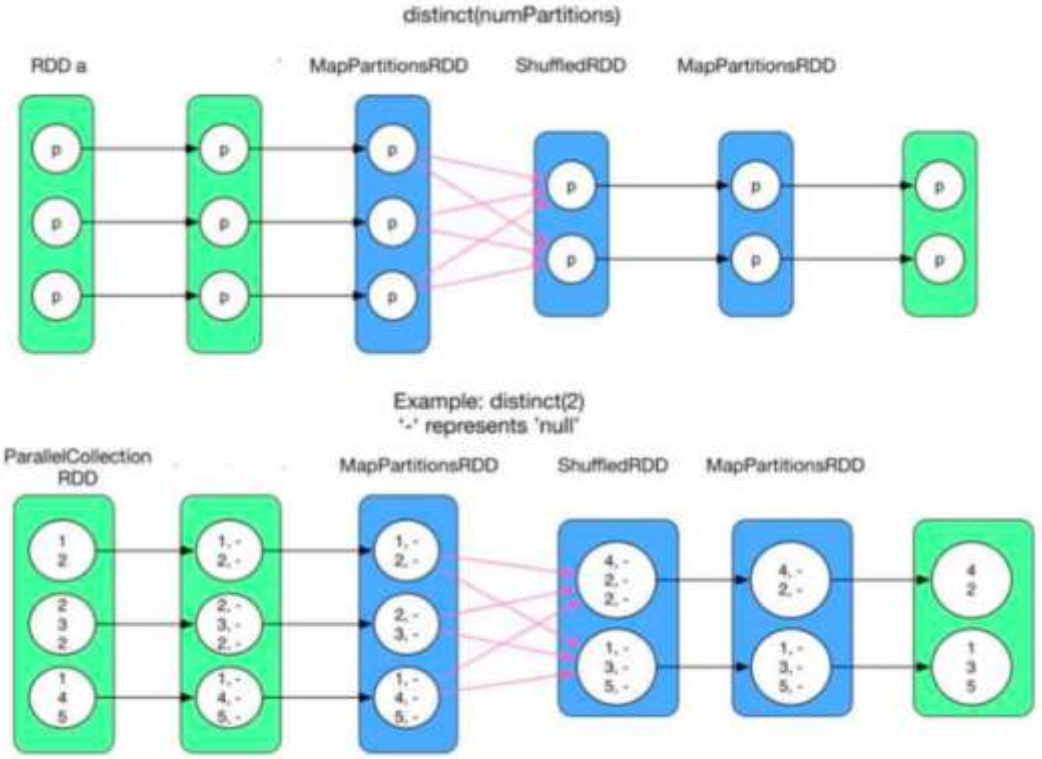
- val rdd = sc.parallelize(List((1, "a"),(2,"b"),(3,"c"),(4, "d"),(5,"e"),(3,"f"),(2," g"),(1,"h")), 3)
- val rdd2 = rdd.groupByKey(2)
- groupByKey通过 shuffle将 Key 相同的 records 聚合在一起
- groupByKey 没有在 map 端进行 combine

典型RDD依赖——reduceByKey



- val rdd = sc.parallelize(List(("A",1),("B",1),("C",1),("B",1),("C",1),("D",1), ("C",1),("A",1)), 3)
- val rdd2 = rdd.reduceByKey(_+ _, 2)
- reduceyByKey相当 于MR中的reduce， 数据流一样
- reduceyByKey在 map 端开启combine

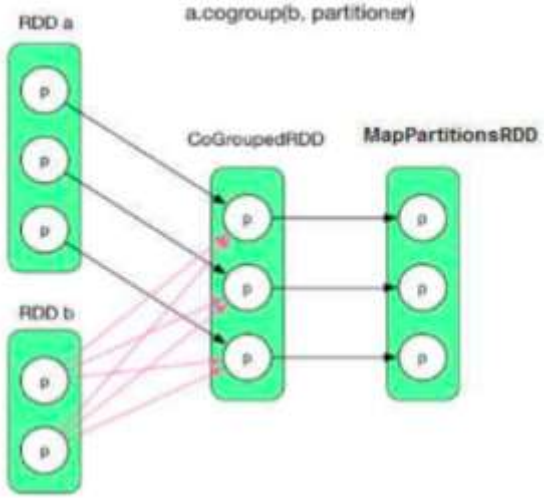
典型RDD依赖——distinct

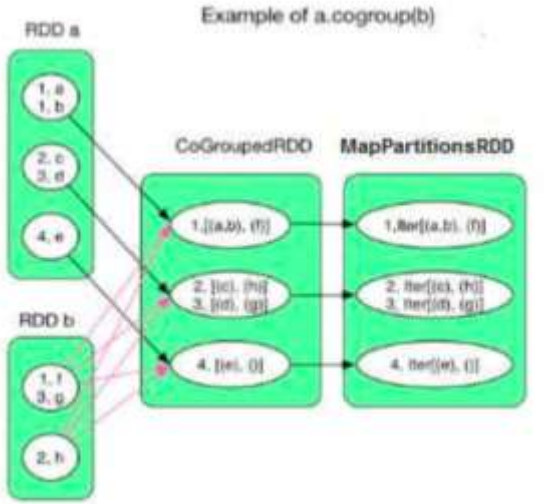


- sc.parallelize(List(1,2,2,3,2,1,4,5),3).distinct(2)

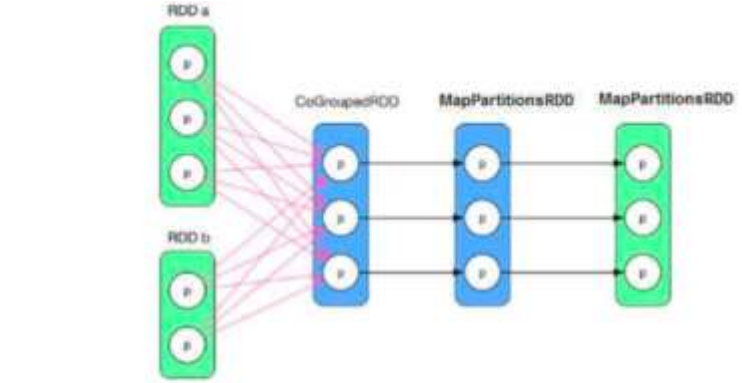
典型RDD依赖——cogroup

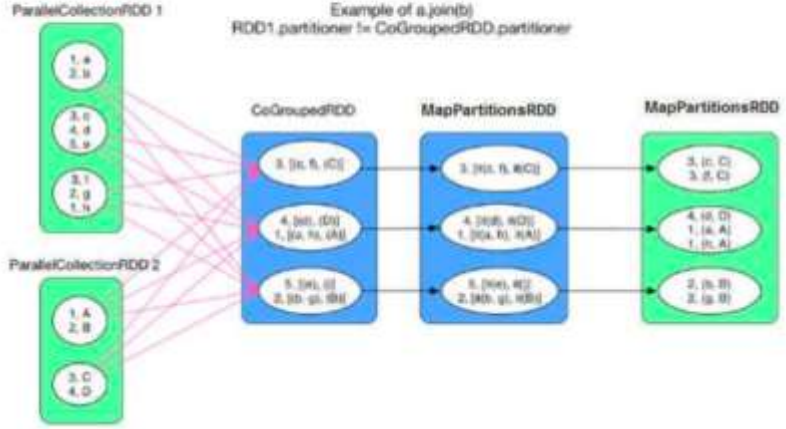
- val rdd1 = sc.parallelize(List((1,"a"),(1,"b"),(2,"c"),(3,"d"), (4,"e")),3)
- val rdd2 = sc.parallelize(List((1,"f"),(3,"g"),(2,"h")),2)
- val rdd3 = rdd1.cogroup(rdd2, 3)



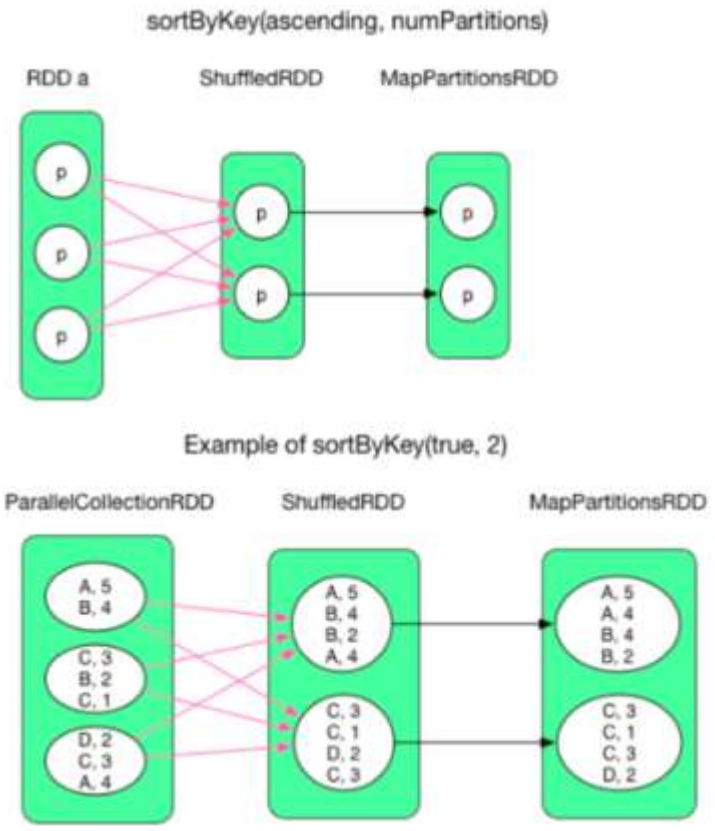


典型RDD依赖——join



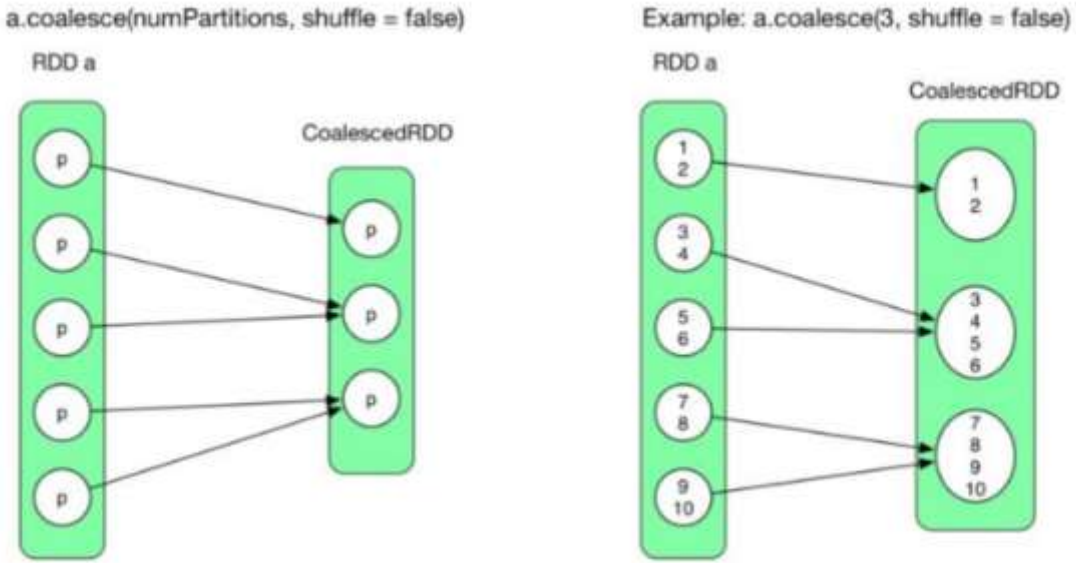


典型RDD依赖——sortByKey



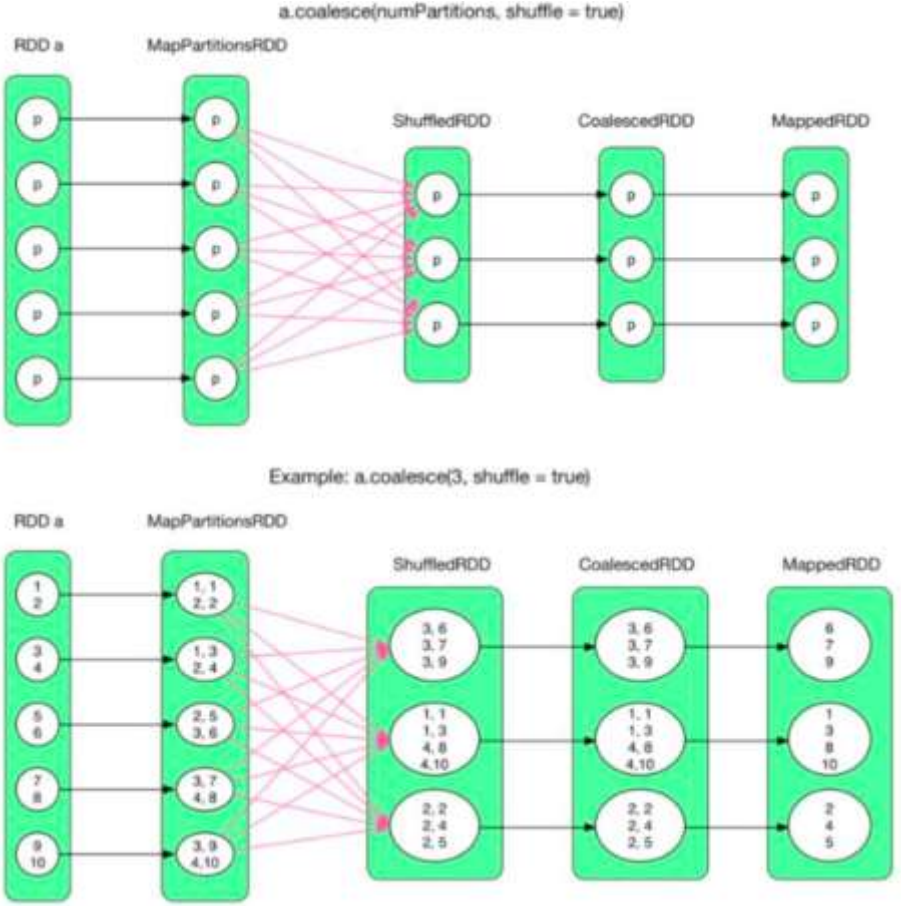
- val rdd = sc.parallelize(List(("A", 5),("B",4),("C",3),("B",2),("C", 1),("D",2),("C",3) ,("A",4)), 3)
- val rdd2 = rdd.sortByKey(true, 2)

典型RDD依赖——coalesce



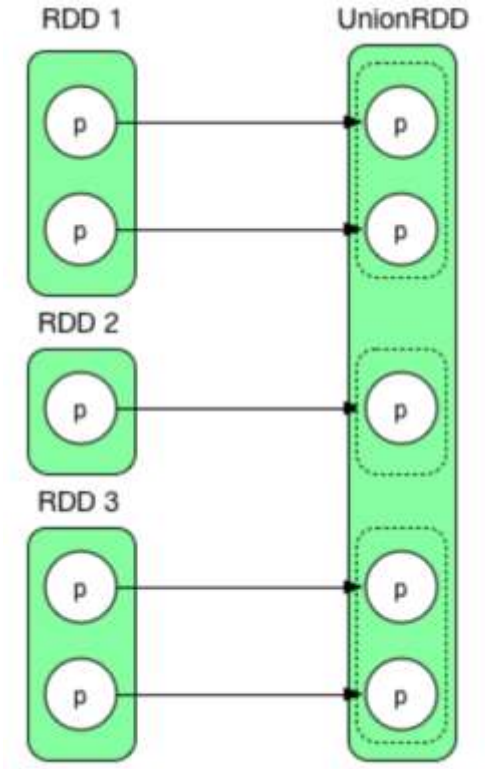
- val rdd = sc.parallelize(List(1,2,3,4,5,6,7,8,9,10), 5)
- rdd.coalesce(3, false)

典型RDD依赖——coalesce



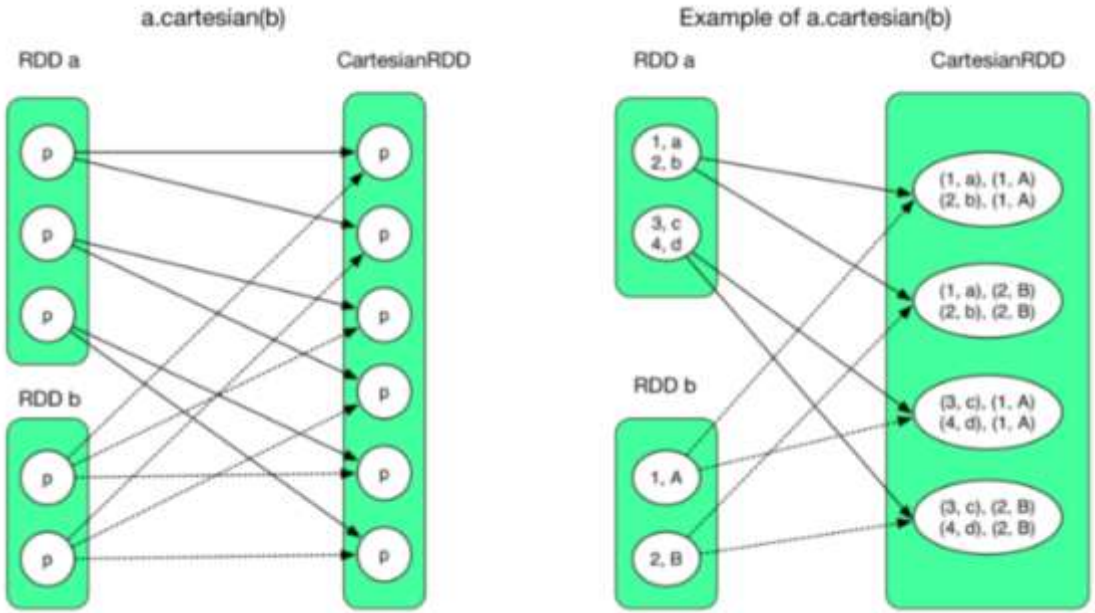
- val rdd = sc.parallelize(List(1,2,3,4,5,6,7,8,9,10), 5)
- rdd.coalesce(3, true)
- repartition(nu mPartitions) 等价于 coalesce(nu mPartitions,tr ue)

典型RDD依赖——union



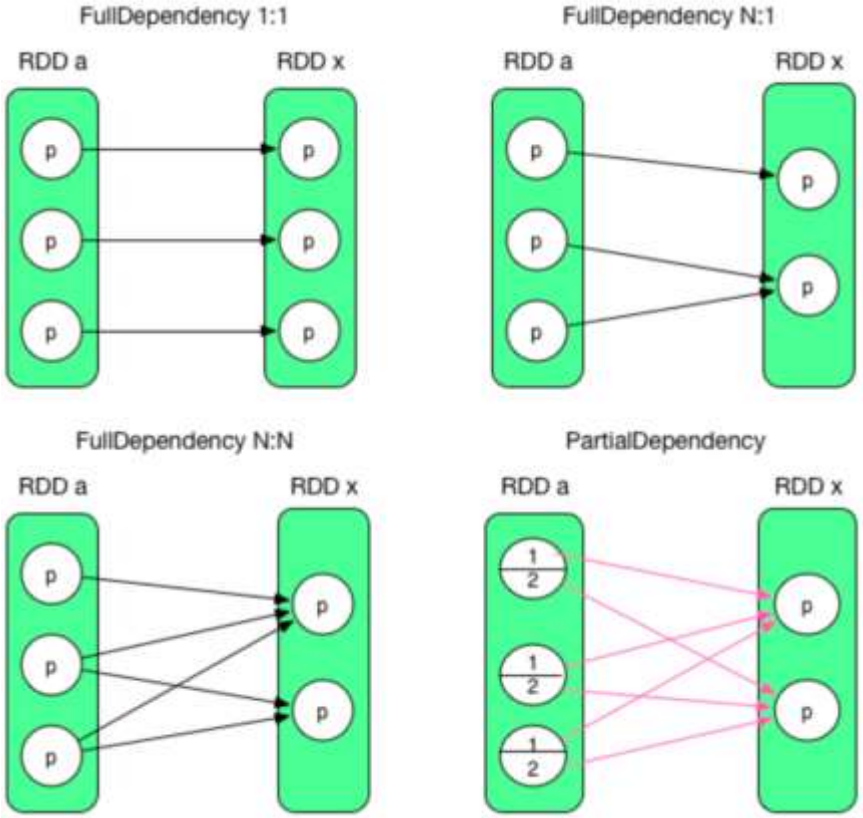
- union将两个 RDD 简单 合并在一起，不改变 partition 里面的数据

典型RDD依赖——cartesian



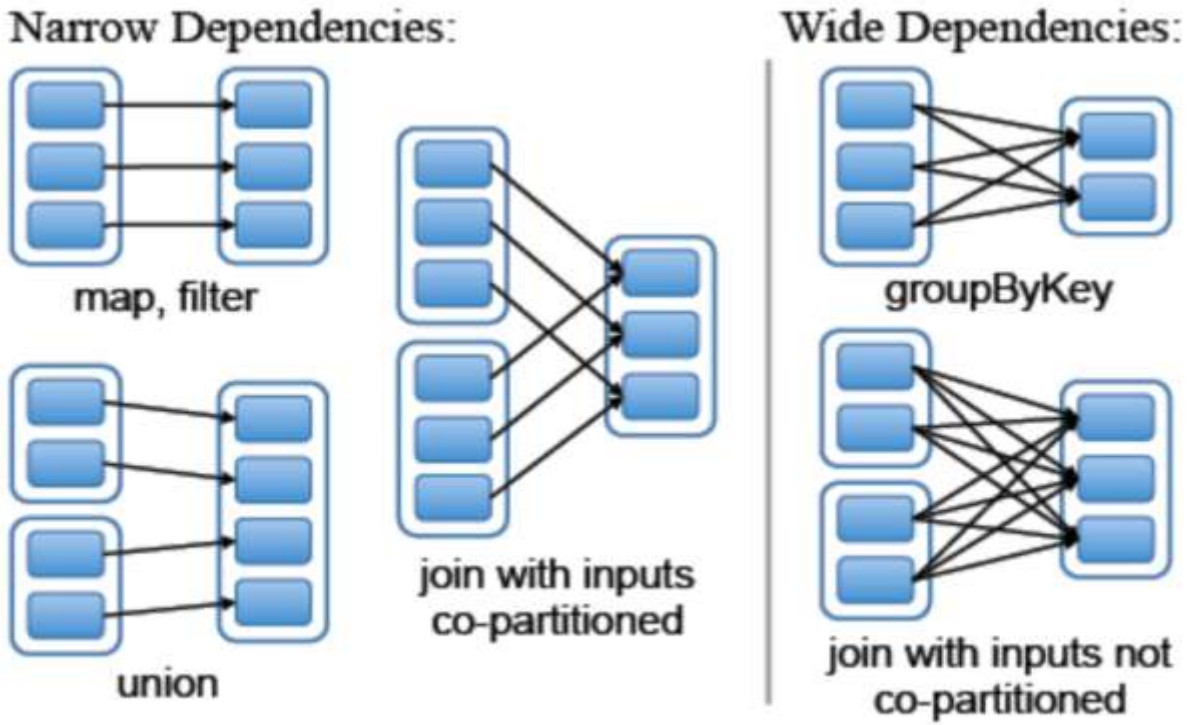
- CartesianRDD中partition个数=RDD a partition个数 * RDD b partition个数
- 一般情况是:max(RDD 1 partition个数，...，RDD n partition个数)

RDD依赖总结



- 1:1 map, filter
- N:1 co-partitioned join
- N:N 很少
- 宽依赖: reduceByKey
- 区分窄依赖和 宽依赖是为了 生成物理执行 计划

RDD依赖总结

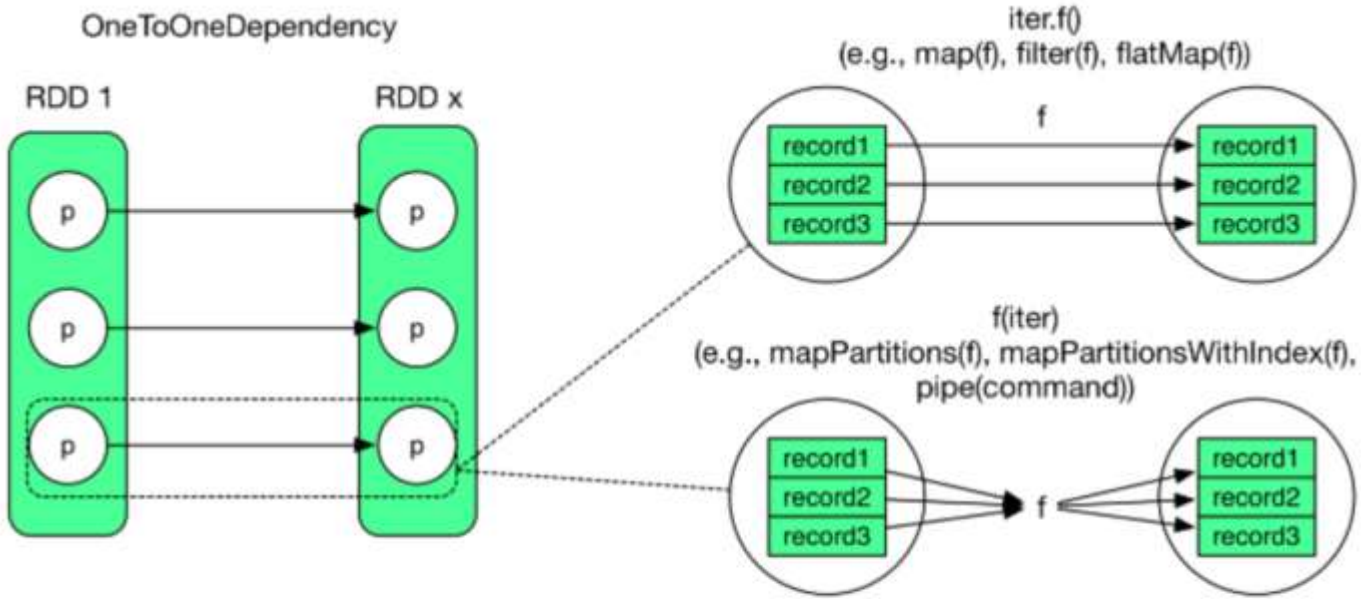


Each box is an RDD, with partitions shown as shaded rectangles

RDD依赖总结

- 窄依赖(narrow dependencies)
  - 子RDD的每个分区依赖于常数个父分区(即与数据规模无关)
  - 允许父分区以流水线的方式找到子分区
  - map产生窄依赖
- 宽依赖(wide dependencies)
  - 子RDD的每个分区依赖于所有父RDD分区
  - 类似于MR的shuffle过程
  - Join产生宽依赖(除非父RDD被哈希分区)

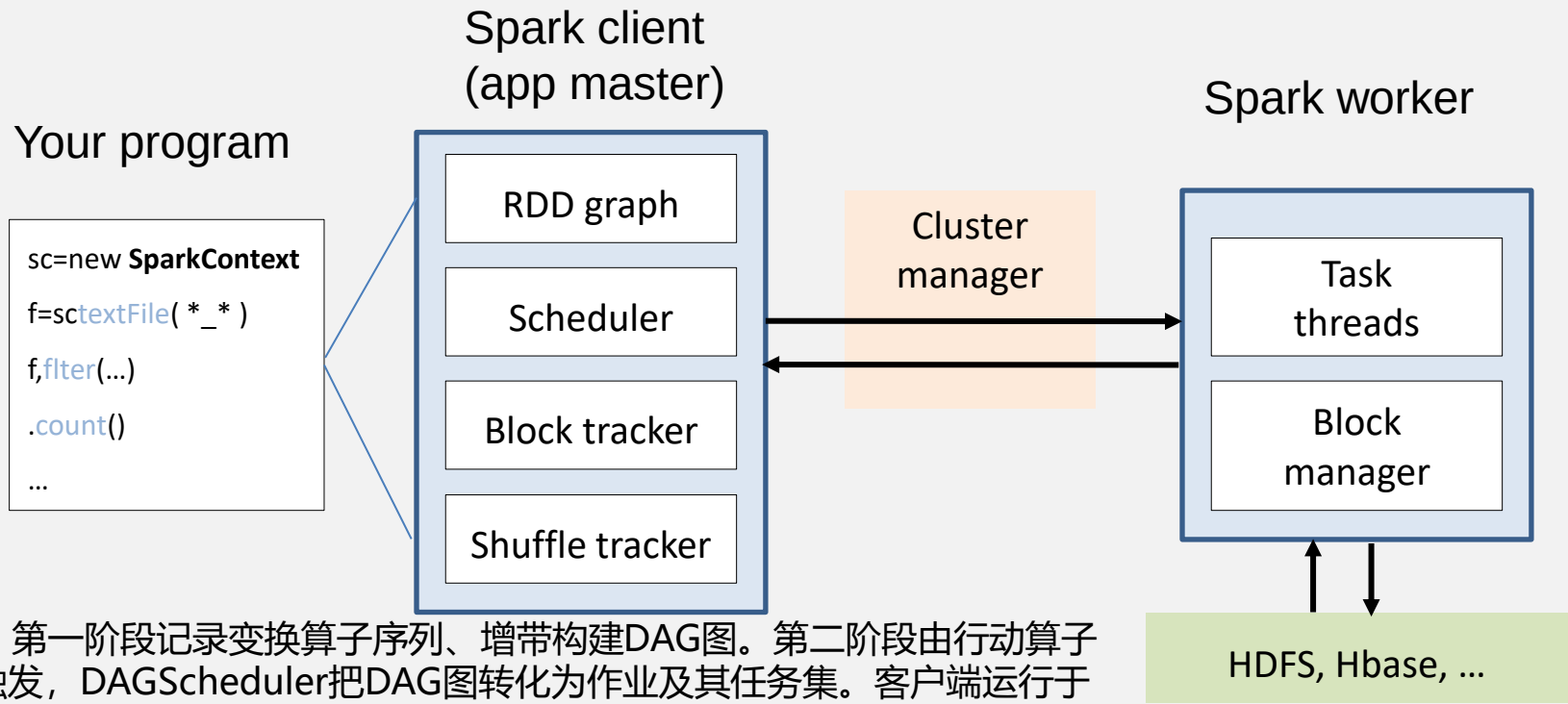
RDD依赖总结




```
int[] array = {1, 2, 3, 4, 5}
for(int i = 0; i < array.length; i++)
    f(array[i])
```

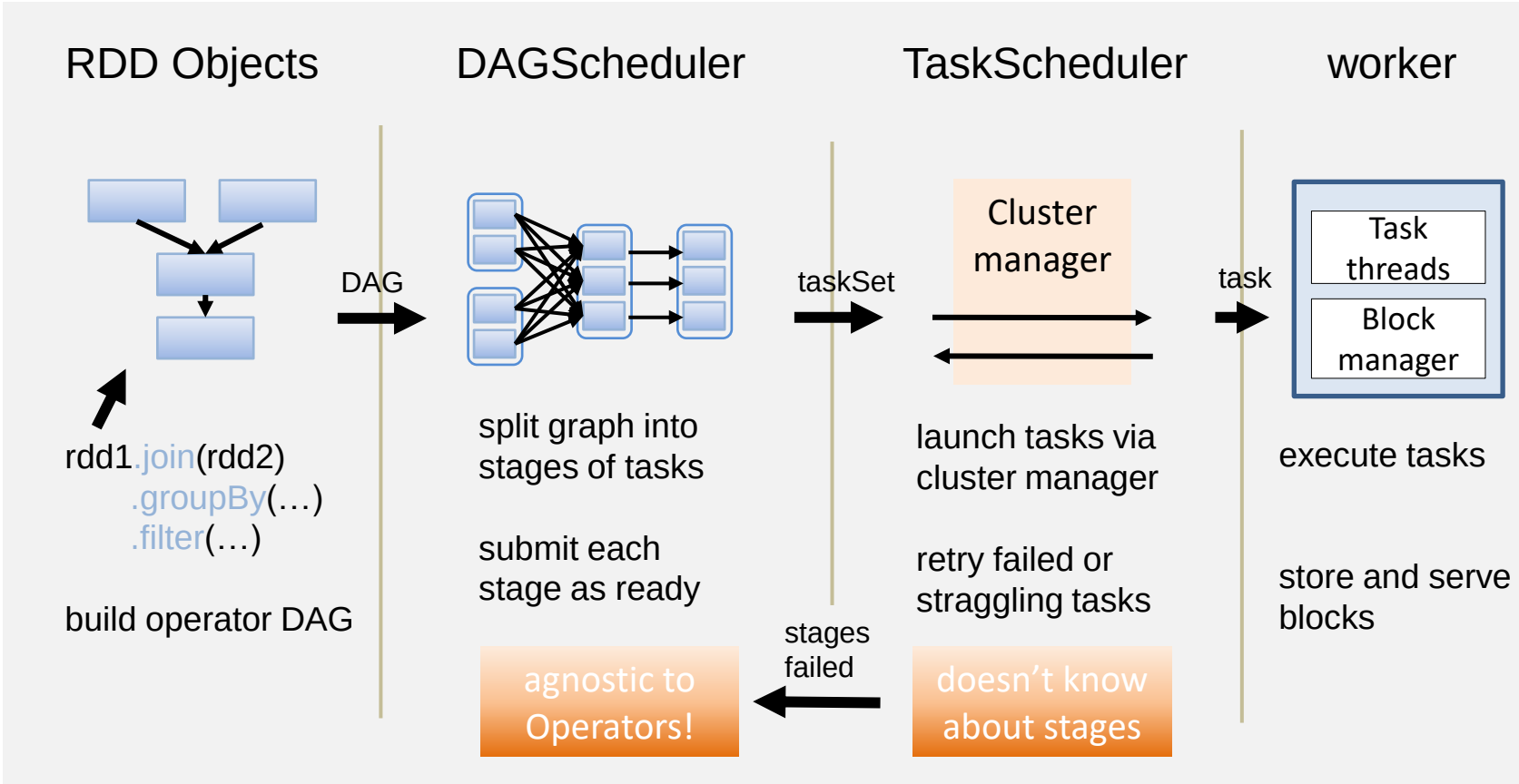
```
int[] array = {1, 2, 3, 4, 5}
f(array)
```

Spark运行和调度



第一阶段记录变换算子序列、增带构建DAG图。第二阶段由行动算子触发，DAGScheduler把DAG图转化为作业及其任务集。客户端运行于master节点上，通过Cluster manager把划分好分区的任务集发送到集群的worker/slave节点上执行。

Spark运行和调度


RDD Objects
DAGScheduler
TaskScheduler
worker
DAG
taskSet
task
Cluster manager
Task threads
Block manager
rdd1.join(rdd2)
.groupBy(…)
.filter(…)
build operator DAG
split graph into stages of tasks
submit each stage as ready
agnostic to Operators!
launch tasks via cluster manager
retry failed or straggling tasks
doesn't know about stages
stages failed
execute tasks
store and serve blocks

Spark配置

- Spark-env.sh配置文件
- Export JAVA_HOME=/usr/local/jdk
- Export SPARK_MASTER_IP=192.168.1.177
- Export SPARK_WORKER_CORES=1
- Export SPARK_WORKER_INSTANCES=1
- Export WORK_MEMORY=8g
- Export SPARK_MASTER_PORT=7077
- Export SPARK_JAVA_OPTS="-verbose:gc –XX:-PrintGCDetails –XX:+PrintGCTimeStamp"

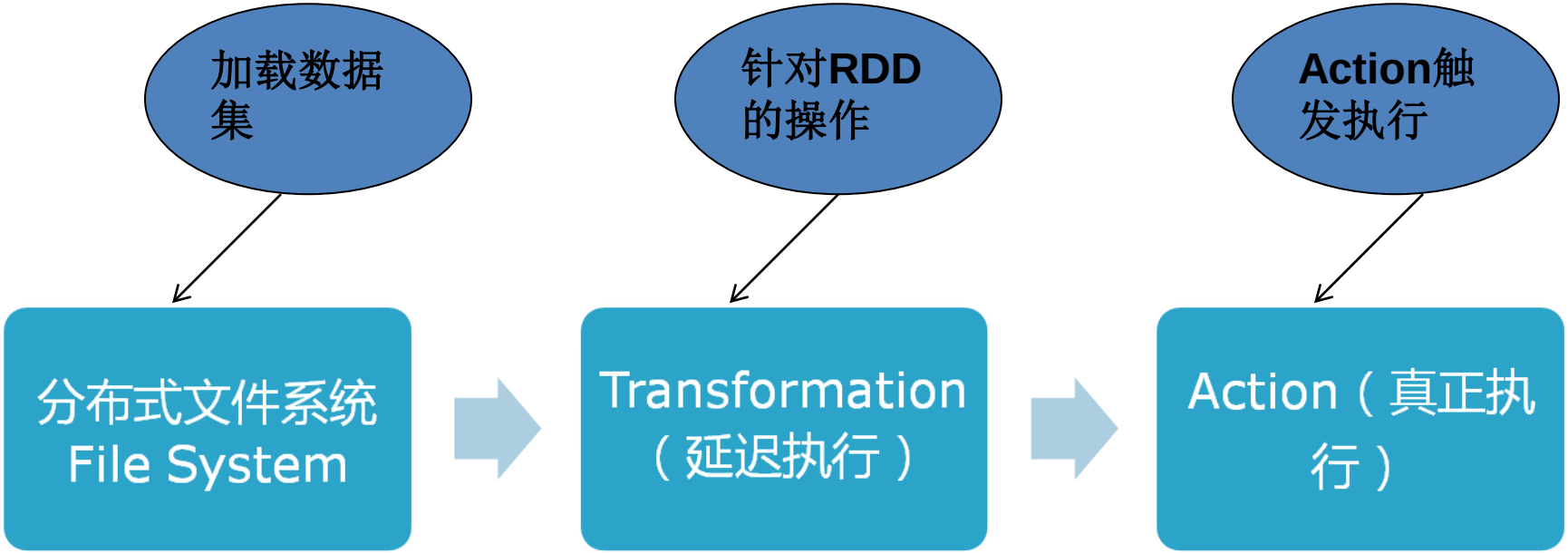
Spark配置

- Slaves
- Xx.xx.xx.2
- Xx.xx.xx.3
- Xx.xx.xx.4
- Xx.xx.xx.5

Spark配置

项目	要求
磁盘	[官方推荐] 4-8块普通磁盘，不需要RAID。
内存	[官方推荐] > 8GB即可。 Spark建议需要提供至少75%的内存空间分配给Spark，至于其余的内存空间，则分配给操作系统与buffer cache。
网络	建议使用10G及以上的网络带宽
CPU	Spark可以支持一台机器扩展至数十个CPU core，它实现的是线程之间最小共享。 若内存足够大，则制约运算性能的就是网络带宽与CPU数。

流程示意



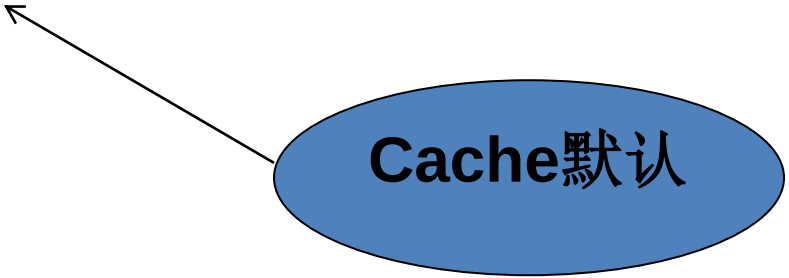
Ps：RDD可以从集合直接转换而来，也可以从现存的任何Hadoop InputFormat而来，亦或者是Hbase等等。

缓存策略

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

缓存策略

- 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)



Spark 关键术语

Client：客户端进程，负责提交作业到Master。
Master：Standalone模式中主控节点，负责接收Client提交的作业，管理Worker，并命令Worker启动分配Driver的资源和启动Executor的资源。
Worker：Standalone模式中slave节点上的守护进程，负责管理本节点的资源，定期向Master汇报心跳，接收Master的命令，启动Driver和Executor。
Driver： 一个Spark作业运行时包括一个Driver进程，也是作业的主进程，负责作业的解析、生成Stage并调度Task到Executor上。包括DAGScheduler，TaskScheduler。
Executor：即真正执行作业的地方，一个集群一般包含多个Executor，每个Executor接收Driver的命令Launch Task，一个Executor可以执行一到多个Task。

Spark 关键术语

Stage：一个Spark作业一般包含一到多个Stage。

Task：一个Stage包含一到多个Task，通过多个Task实现并行运行的功能。

DAGScheduler： 实现将Spark作业分解成一到多个Stage，每个Stage根据RDD的Partition个数决定Task的个数，然后生成相应的Task set放到TaskScheduler中。

TaskScheduler：实现Task分配到Executor上执行。

Spark 关键术语

Application：Spark的应用程序，包含一个Driver program和若干个Executor。
SparkContext：Spark应用程序的入口，负责调度各个运算资源，协调各个Worker Node上的Executor
Driver program：运行Application的main()函数并且创建SparkContext
Executor：是Application运行在Work node上的一个进程，该进场负责运行Task，并且负责将数据存在内存或者磁盘上；每个Application都会申请各自的Executors来处理认为
Cluster Manager：在集群上获取资源的外部服务(例如：Standalone、Mesos、Yarn)
Work Node：集群中任何可以运行Application代码的节点，运行一个或者多个Executor进程

Spark 关键术语

Task：运行在Executor上的工作单元
Job：SparkContext提交的具体Action操作，常和Action对应
Stage：每个Job会被拆分很多组任务（task），每组任务被称为Stage，也称TaskSet
RDD：Resilient Distributed Datasets的简称，弹性分布式数据集，是Spark最核心的模块和类
DAGScheduler：根据Job构建基于Stage的DAG(Directed Acyclic Graph,有向无环图)，并提交Stage给TaskSchedule
TaskSchedule：将TaskSet提交给Worker node集群运行并返回结果
Transformation/Action：Spark API的两种类型；Transformation返回值还是一个RDD，Action返回值不少一个RDD，而是一个Scala的集合；所有的Transformation都是采用的懒策略，如果只是将Transformation提交是不会执行计算的，计算只有在Action被提交时才会被触发。

Spark执行计划

- MR中，用户直接面对 task，mapper 和 reducer 的职责分明:一个进行分块处理， 一个进行 aggregate
- MR中数据依赖和task执行流程是一致且固定 的，只需实现 map 和 reduce函数即可
- Spark中数据依赖可以非常灵活
- Spark将数据依赖和具体task的执行流程分开，通过将逻辑执行计划转换成物理执行计划

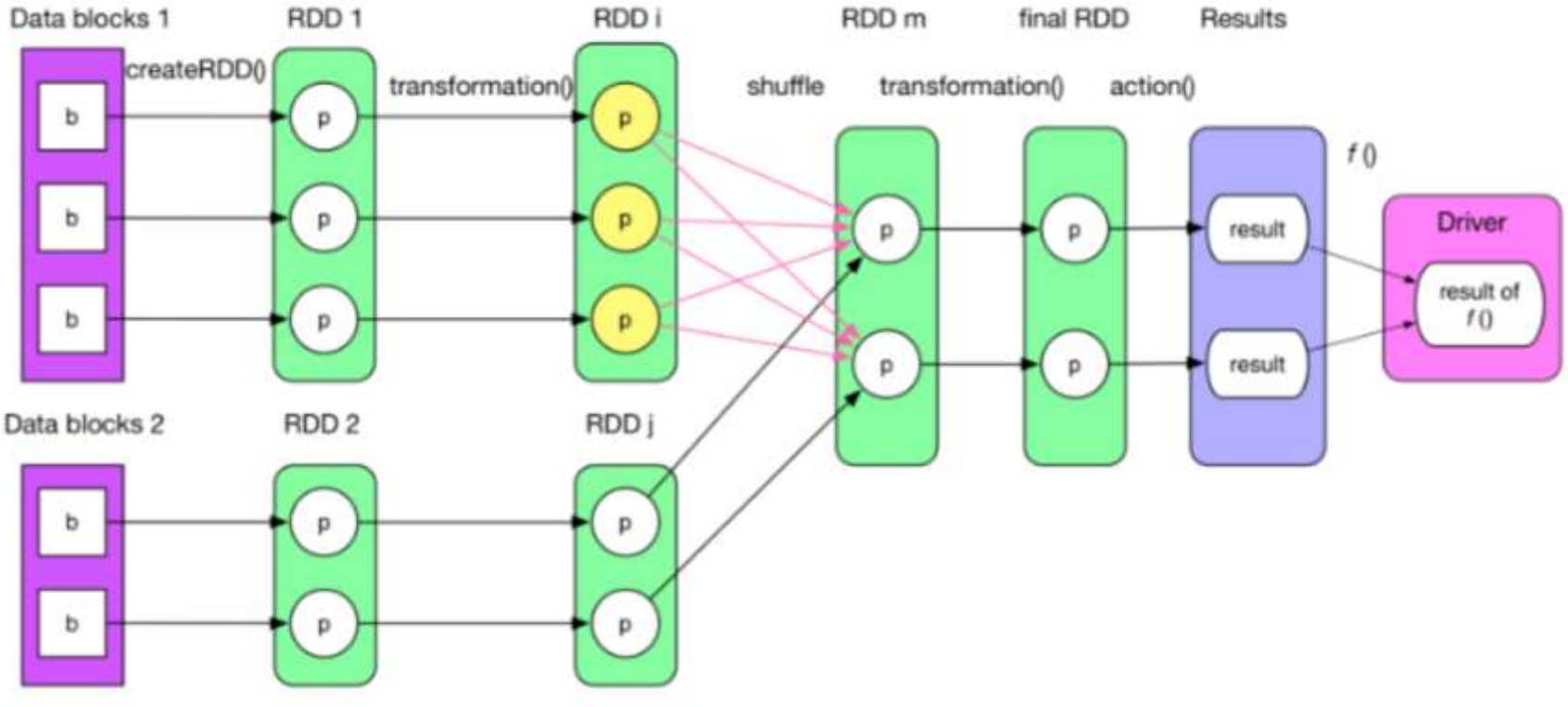
Spark执行计划

- 首先建立逻辑执行计划(RDD依赖关系)
- 把逻辑执行计划转换为物理执行计划(划分 Job，形成Stage和Task)
- 执行Task

Spark逻辑执行计划

- 逻辑执行计划描述了数据流：经过哪些 transformation，中间生成哪些 RDD 及 RDD 之间的依赖关系
- 逻辑执行计划表示数据上的依赖关系，不是 task的执行流程

Spark逻辑执行计划



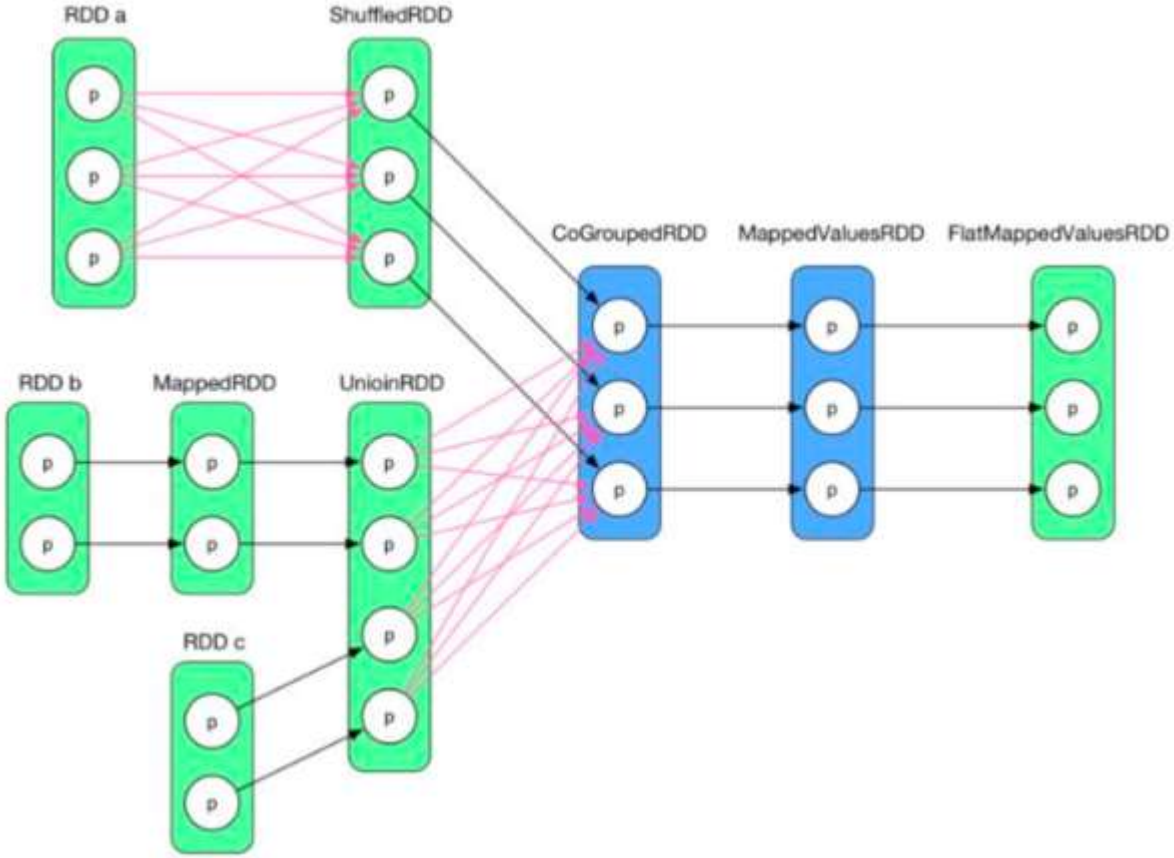
Spark逻辑执行计划

- 从数据源读取数据创建最初的 RDD，如本地 file、 内存数据结构(parallelize)、 HDFS、HBase等
- 对 RDD 进行一系列的 transformation操作
- 对final RDD 进行 action 操作，每个 partition 计算后产生结果 result
- 将 result 回送到 driver 端，进行最后的 f(list[result]) 计算
- 如count() 实际包含了action() 和 sum() 两步

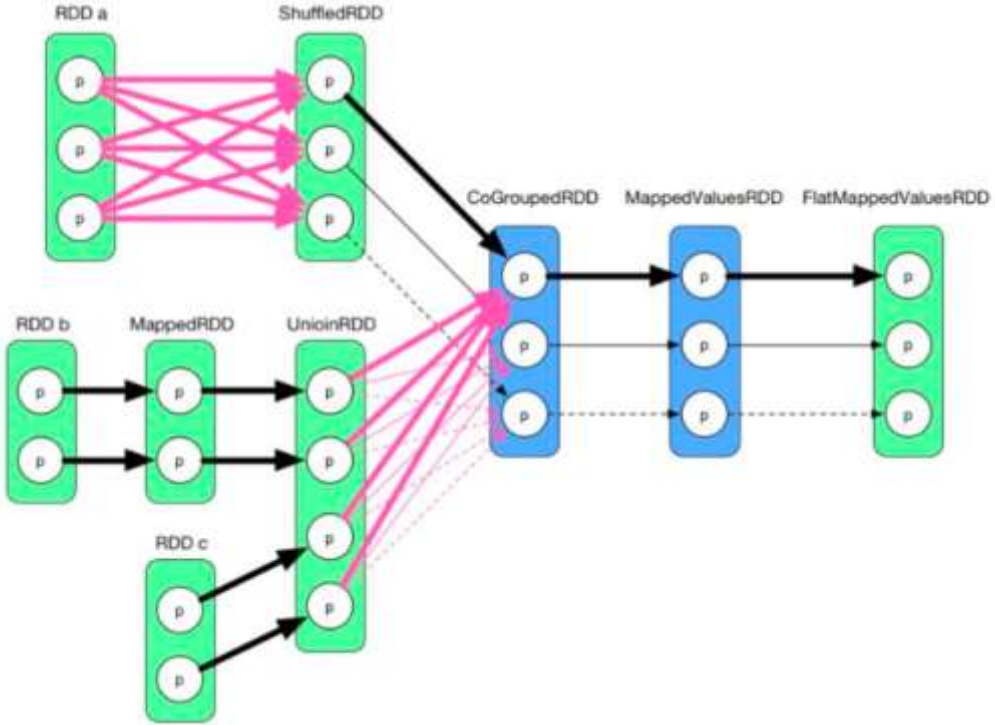
Spark物理执行计划

```
import org.apache.spark.HashPartitioner
val rddA = sc.parallelize(List((5,"e"),(3,"a"),(2,"b"),(1,"c")),3) val
rddAHash = rddA.partitionBy(new HashPartitioner(3))
val rddB = sc.parallelize(List((1,"A"),(2,"B"),(3,"C"),(4,"D")),2) val
rddBMap = rddB.map(x => (x._1, x._2 + "1"))
val rddC = sc.parallelize(List((1,"X"),(2,"Y")),2) val rddBUnionC =
rddBMap.union(rddC)
val result = rddAHash.join(rddBUnionC ,3) result.toDebugString
```

Spark物理执行计划

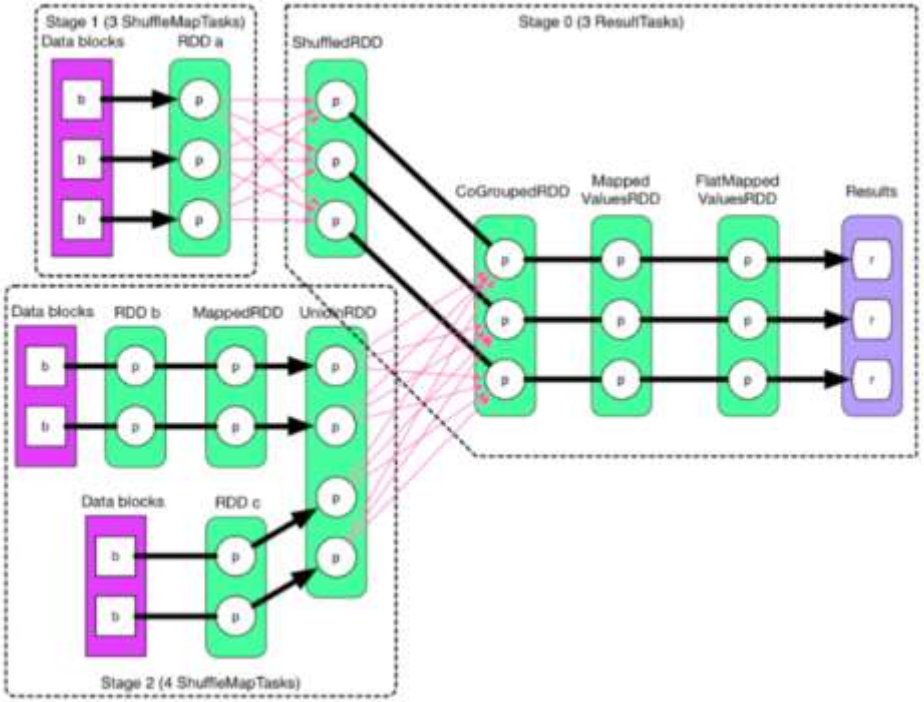


Spark物理执行计划



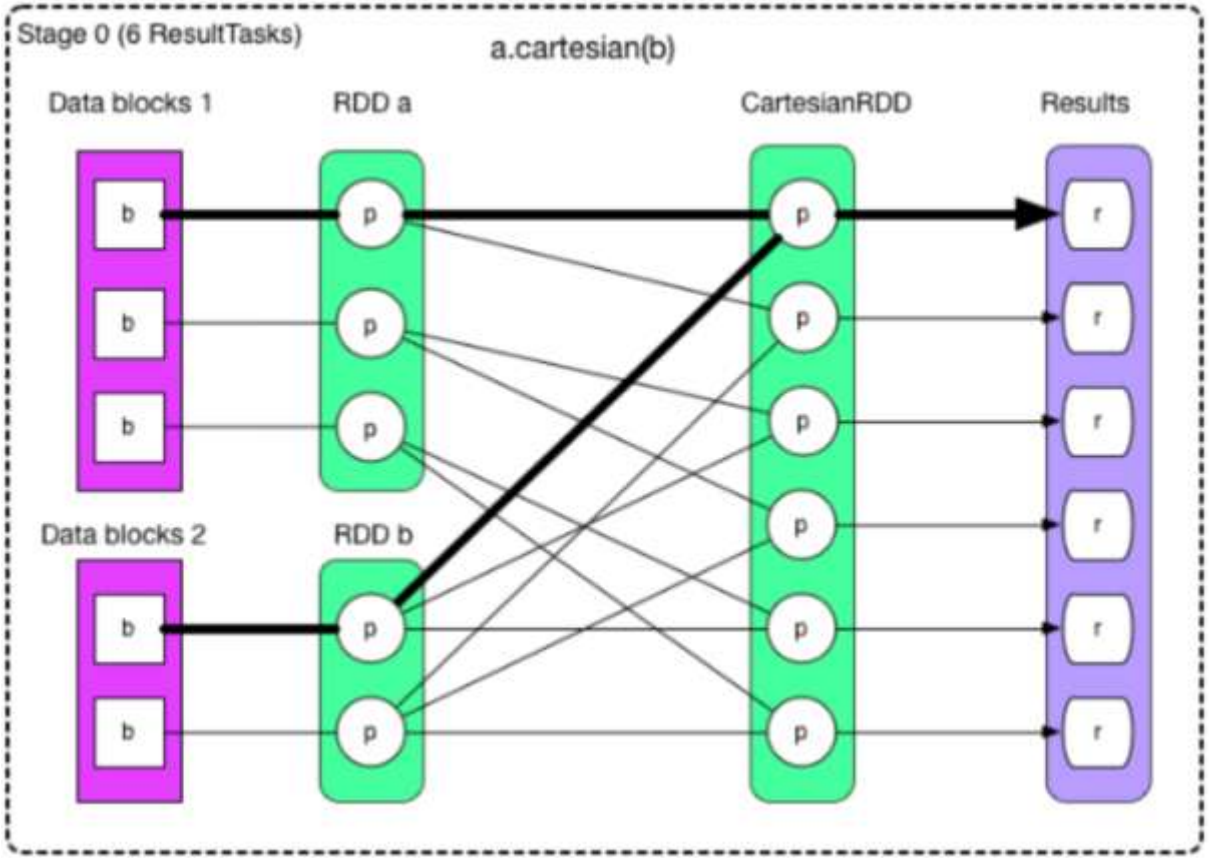
- RDD中每个Partition是独立的

Spark物理执行计划



- 从后往前计算，根据窄依赖和宽依赖划分stage
- stage 里task 数目取决于最后一个 RDD 中的 partition 个数

Spark物理执行计划



Spark物理执行计划

- 1个 Application 可以包含多个 job 、
- 1个 job可以包含多个 stage
- 1个 stage可以包含多个 task

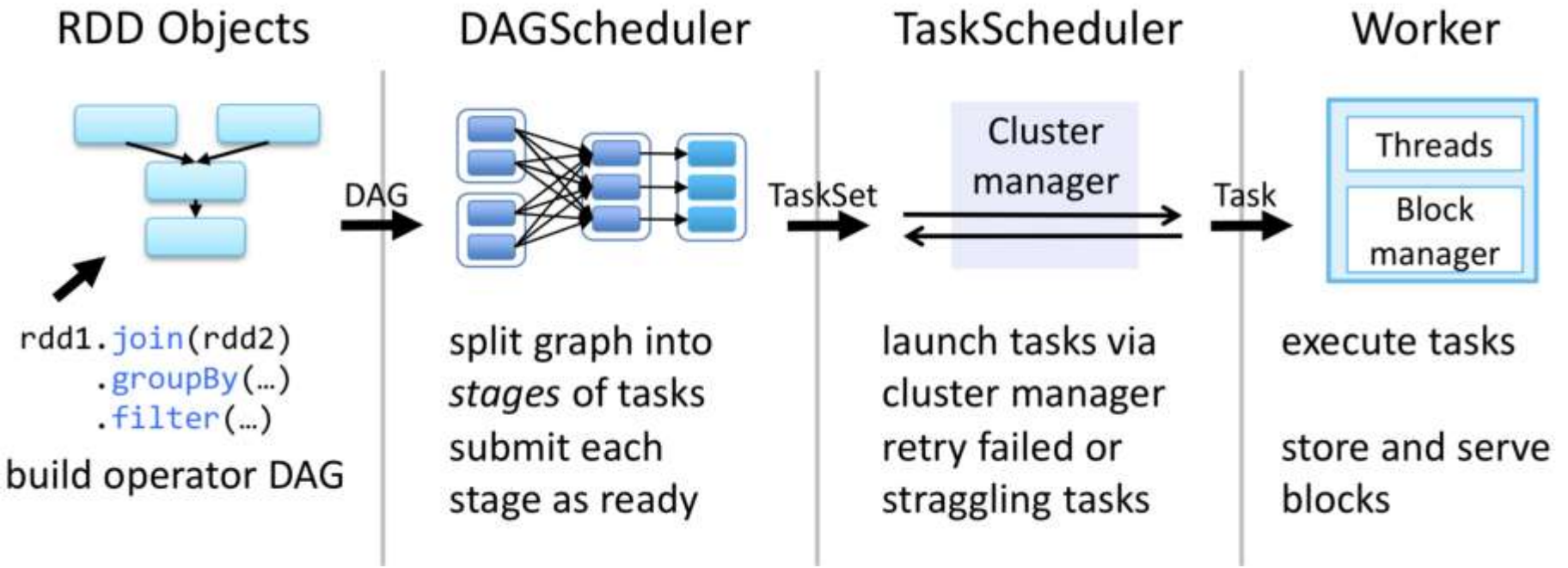
Spark物理执行计划

- 从后往前推，需要哪个 partition 就计算哪个 partition，如果 partition 里面没有数据，就继续向前推
- 对于没有 parent stage 的 stage，该 stage 最左边的 RDD 是可以立即计算的
- 对于有 parent stage 的 stage，先等着所有 parent stages 中 final RDD 中数据计算好，然后shuffle
- 实际计算时数据从前到后流动，计算出第一个 record 流动到不能再流动后，再计算下一个 record
- 并不是当前 RDD 的 partition 中所有 records 计算后再整体向后流动

Spark物理执行计划

- 执行count，先在每个 partition上执行 count， 然后执行结果被发送到 driver，最后在 driver 端进行 sum
- 执行take类似

Spark应用调度



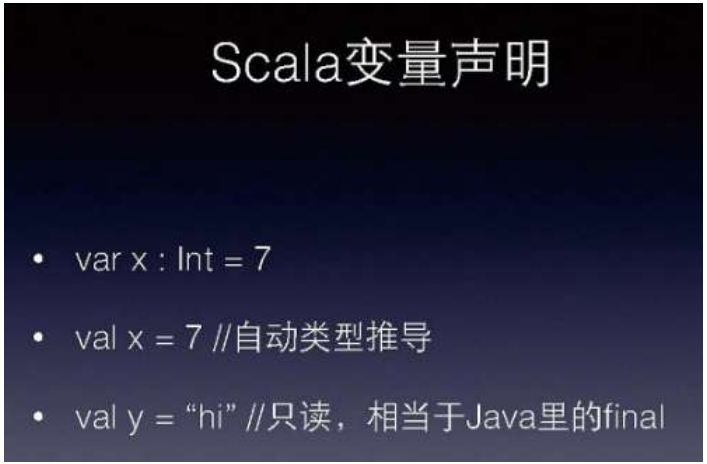
- DAGScheduler:根据Job构建基于Stage的DAG，并提交 Stage给TaskScheduler
- TaskScheduler:将Task提交给Worker节点运行并返回结果

Spark api

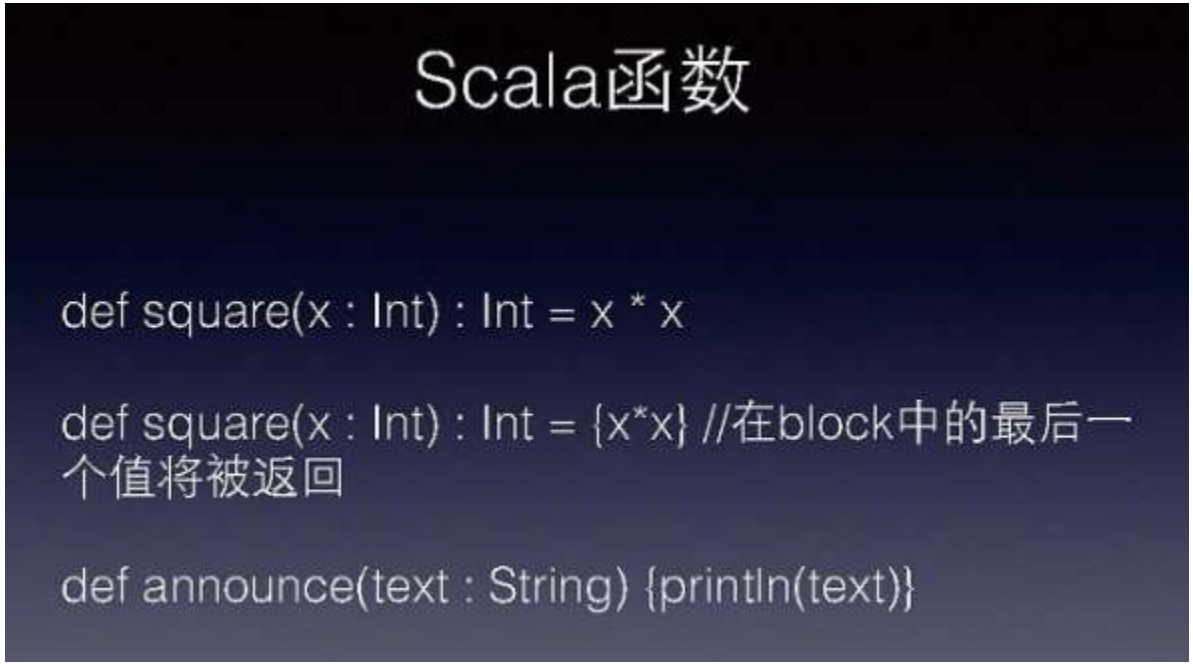
- Scala
- Java
- Python
- R

Scala

- 基于JVM的函数式和面向对象结合
- 静态语言，python动态语言
- 和Java可以互操作



Scala


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


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


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


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)

Scala


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)

Spark SQL

- Spark SQL的前生是Shark
- 基于Spark最佳的SQL计算
- 专门用于处理结构化数据
- 支持多种数据源，包括Hive、parquet、orc、JSON、RDBMS、RDD等
- 性能优化上做的很好
- 使用简单，代码易读
- 给熟悉SQL的技术人员提供快速访问大数据的工具

Spark SQL

- 处理结构化数据
- 把结构数据抽象成 DataFrame
- 工作方式：分布式SQL查询引擎

```
// Load a text file and convert each line to a Java Bean.
JavaRDD<Person> people = ctx.textFile("examples/src/main/resources/people.txt")

// SQL can be run over RDDs that have been registered as tables.
DataFrame teenagers = sqlContext.sql("SELECT name FROM people WHERE age >= 13 AND age <= 19");

// The results of SQL queries are DataFrames and support all the normal RDD operations.
// The columns of a row in the result can be accessed by ordinal.
List<String> teenagerNames = teenagers.toJavaRDD().map(new Function<Row, String>() {
  @Override
  public String call(Row row) {
    return "Name: " + row.getString(0);
  }
}).collect();
```

Spark SQL API

- SQL
- 与hive类似
- 有SparkSQL命令行
- 有thriftserver和beeline客户端
- DataFrame

DataFrame

- 可以在spark程序中使用SQL
- 与RDD类似的分布式数据集，但增加了列的概念
- 之前是SchemaRDD
- 将来是DataSet
- 其API支持Scala/Java/Python/R
- SQL与控制语句的结合

DataFrame开发

- 创建SQLContext对象
- 生成DataFrame，可来自RDD或其他数据源
- 执行Transformation，调用DataFrameAPI或者通过函数执行sql
- 执行Action，返回结果或保存到文件中

DataFrame开发

- 创建SQLContext对象

```
// 导入语句，可以隐式地将RDD转化成DataFrame
import sqlContext.implicits._
// 首先用SparkContext对象创建SQLContext对象
valsqlContext= new org.apache.spark.sql.SQLContext(sc)
```

DataFrame开发

- RDD生成DataFrame

```
//参数名即为列名
case class Employee(id: Int, name: String, age: Int)

valrdd= sc.textFile("employee.txt")
valdf= rdd.map(_.split(","))
.map(e => employee(e(0).trim.toInt, e(1), e(2).trim.toInt))
.toDF()
```

DataFrame开发

```
import sqlContext.implicits._
valsqlContext= new org.apache.spark.sql.SQLContext(sc)
case class Employee(id: Int, name: String, age: Int)
valrdd= sc.textFile("employee.txt")
valdf= rdd.map(_.split(","))
.map(e => employee(e(0).trim.toInt, e(1), e(2).trim.toInt))
.toDF()
df.show()
df.printSchema()
df.select("name").show()
df.filter(df("age") > 21).show()
df.groupBy("age").count().show()
```

DataFrame开发

- RDD生成DataFrame
- case class方式：通过定义case Class，使用反射推断Schema
- applySchema方式：通过可编程接口，定义Schema，并应用到RDD上

DataFrame读写多种数据源

- RDD
- JSON
- Parquet
- JDBC

DataFrameSaveMode

SaveMode选择

- SaveMode.Append：将新数据追加在原数
- 据后面
- SaveMode.Overwrite：覆盖原数据
- SaveMode.ErrorIfExists：默认为此状态
- SaveMode.Ignore：如果文件存在就放弃写

Spark SQL示例

- RDD转DataFrame
- JSON转DataFrame
- Parquet转DataFrame
- hive表转DataFrame
- hive表与非hive表混合使用
- cache使用

Spark SQL 优化

选项	默认值	含义
spark.sql.shuffle.partitions	200	设置Shuffle过程中Reduce Task的并行度
spark.sql.autoBroadcastJoinThreshold	10M	广播join表，在内存够用的情况下，增加其大小
spark.sql.inMemoryColumnarStorage.compressed	true	会自动压缩内存中的列式存储
spark.sql.parquet.compression.codec	snappy	可以设置多种编码方式，除了snappy外，还可以设置为uncompressed，gzip和lzo
spark.sql.tungsten.enabled	true	钨丝计划自动管理内存

Spark SQL 优化

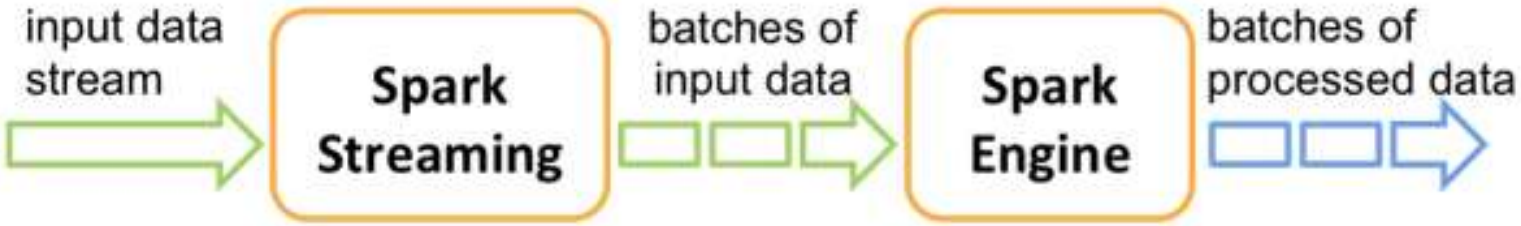
- 通sqlContext.setConf(“spark.sql.shuffle.partitions”，20)设置
- 编写SQL时，给出明确的列名，如select column1，column2 from table，不要写select *的方式
- 对于Spark SQL查询结果，如果数据量比较大，比如超过1000条，那么就不要一次性collect()到Driver
- 对于SQL语句中多次使用到的表，对其进行缓存，使用sqlContext.cacheTable(tableName)，或者
- dataFrame.cache()

Spark Streaming介绍

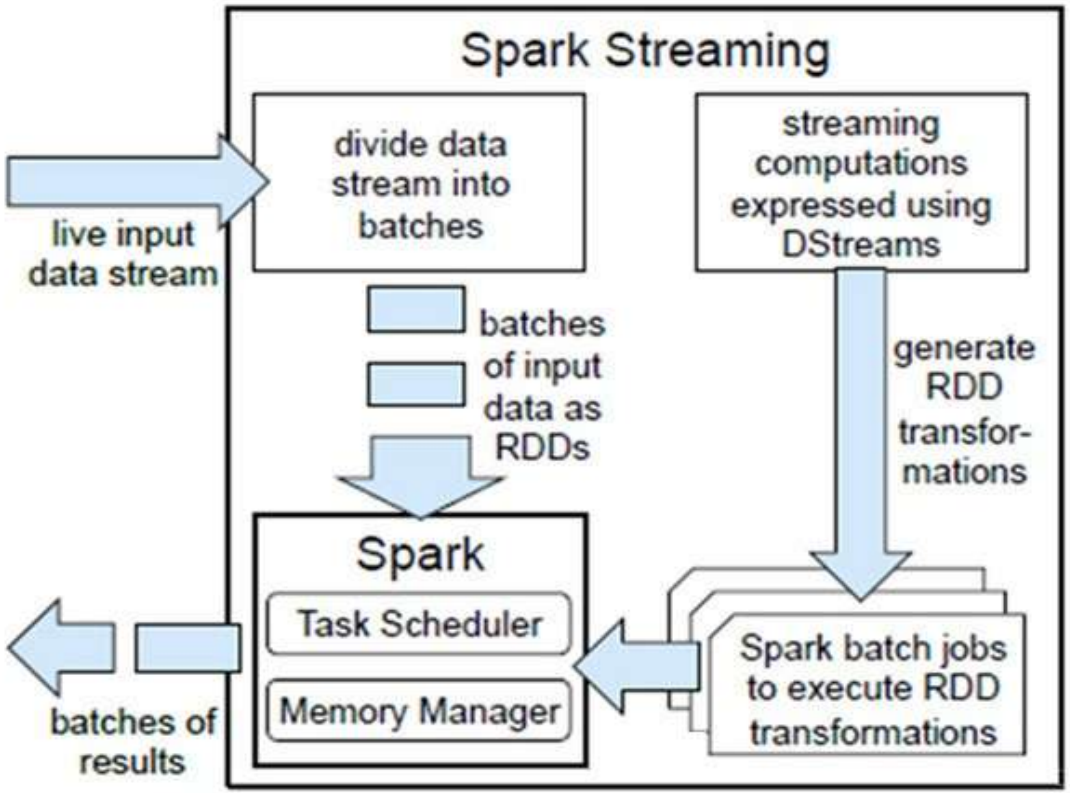
- 近实时流处理计算框架
- 面向大数据，扩展容易
- 秒级延迟
- 提供简单的类似于Spark Core的API
- 支持多种输入和多种输出

Spark Streaming

具有高吞吐与容错性能的在线数据流的实时流处理框架。
构建在Spark上处理Stream数据的框架，基本原理是将Stream数据分成小的时间片断（几秒），以类似batch批量处理的方式来处理这小部分数据。
Spark Streaming构建在Spark上，一方面是因为Spark的低延迟执行引擎（100ms+）可以用于实时计算，另一方面相比基于Record的其它处理框架（如Storm），RDD数据集更容易做高效的容错处理。此外小批量处理的方式使得它可以同时兼容批量和实时数据处理的逻辑和算法。方便了一些需要历史数据和实时数据联合分析的特定应用场合。

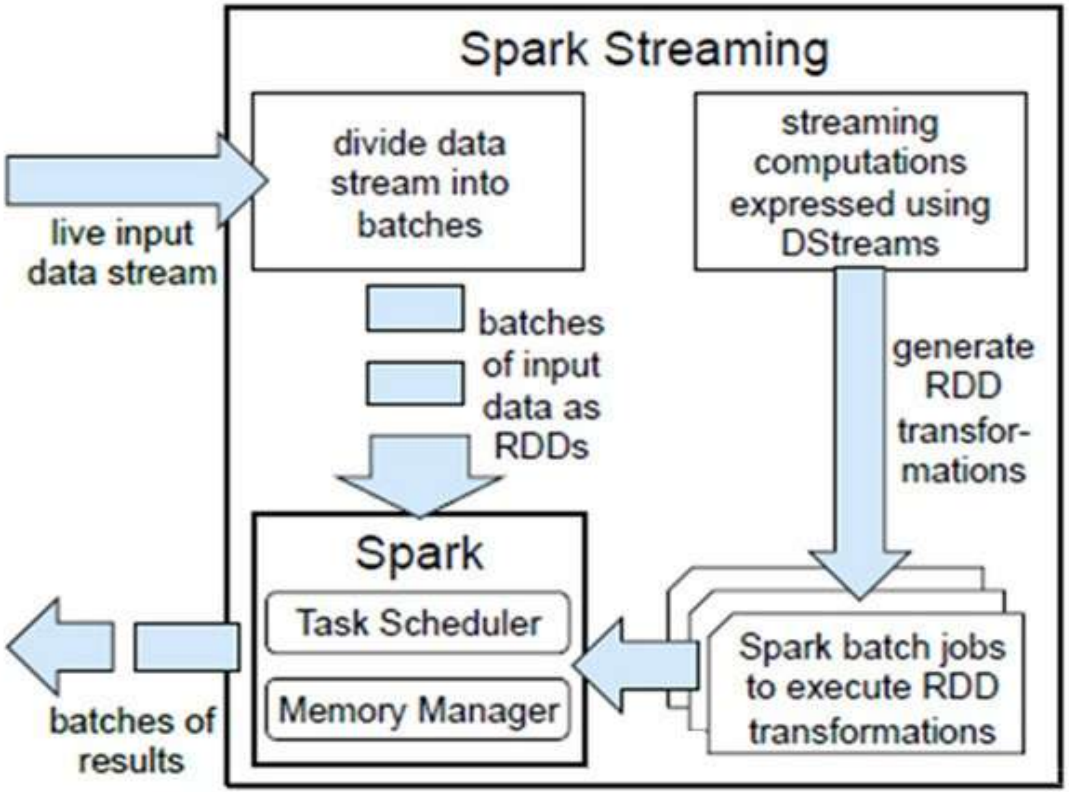


Spark Streaming



- Spark Streaming是将流式计算分解成一系列短小的批处理作业。这里的批处理引擎是Spark，也就是把Spark Streaming的输入数据按照batch size（如1秒）分成一段一段的数据（Discretized Stream），每一段数据都转换成Spark中的RDD（Resilient Distributed Dataset），然后将Spark Streaming中对DStream的Transformation操作变为针对Spark中对RDD的Transformation操作，将RDD经过操作变成中间结果保存在内存中。整个流式计算根据业务的需求可以对中间的结果进行叠加，或者存储到外部设备。

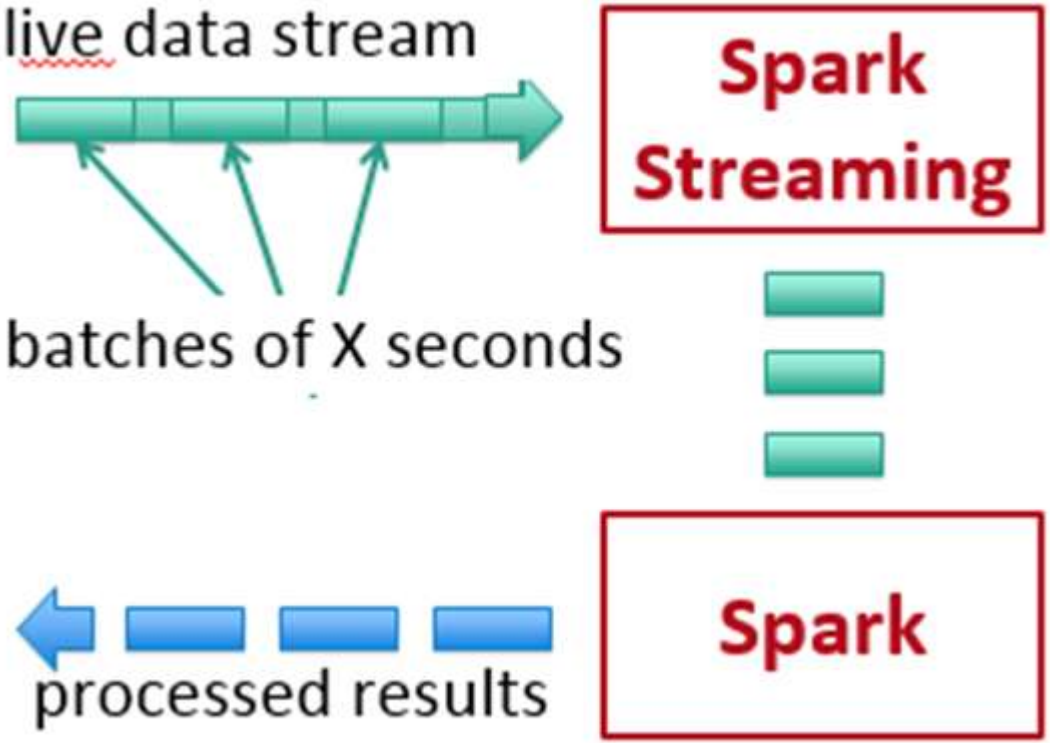
Spark Streaming



- 持续的从输入源读取数据
- 根据数据推送的时间，按时间段切片
- 把切片包装成 RDD，执行 Map + Reduce 计算
- 通过 RDD.collect() 函数收集计算结果

Spark Streaming——输入输出

Spark Streaming和Spark关系

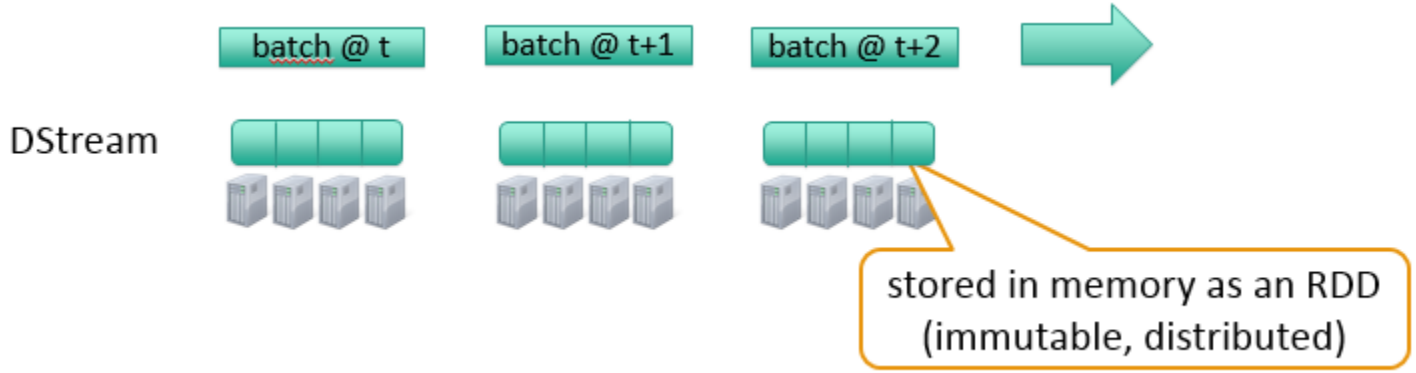


DStream

DStream: RDD序列

valssc= new StreamingContext(conf, Seconds(1))

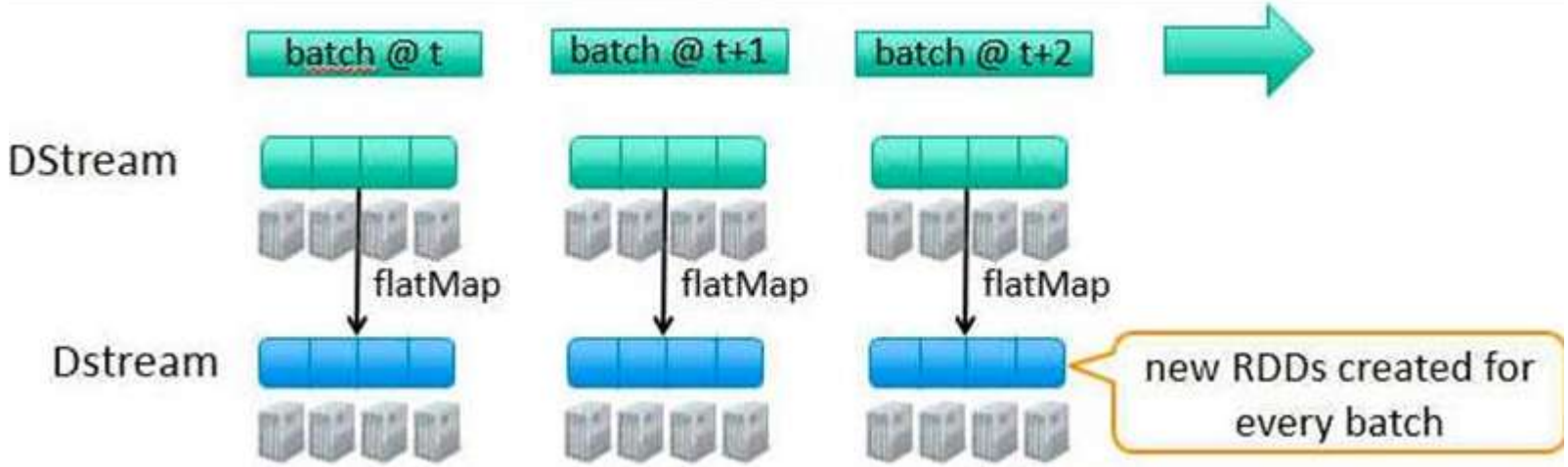
vallines = ssc.socketTextStream("localhost", 9999) ———产生DStream



DStream

- DstreamTransformation: 由1个DStream创建出1个新的DStream

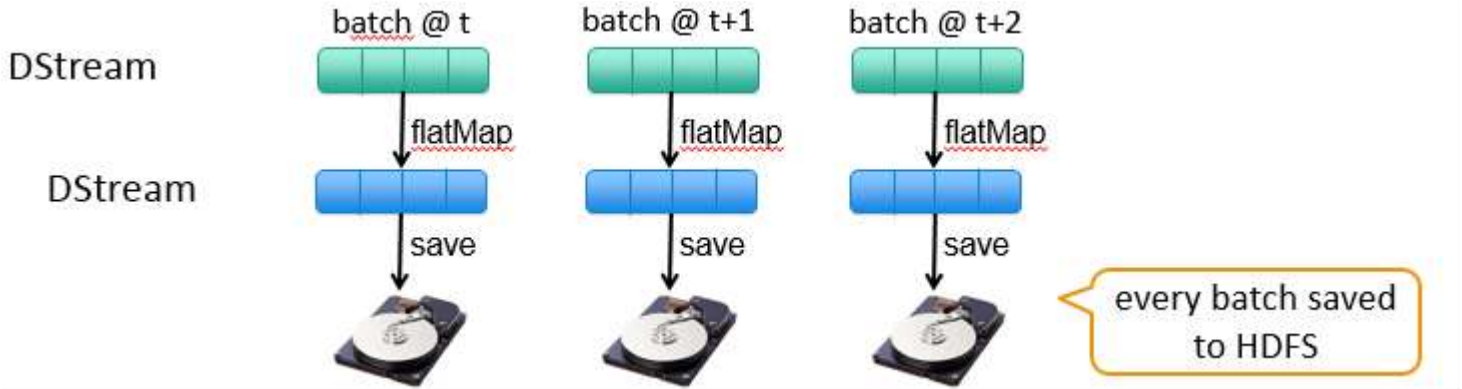
valwords = lines.flatMap(_.split(" "))



DStream

➢DstreamOutput：把数据放入外部存储中

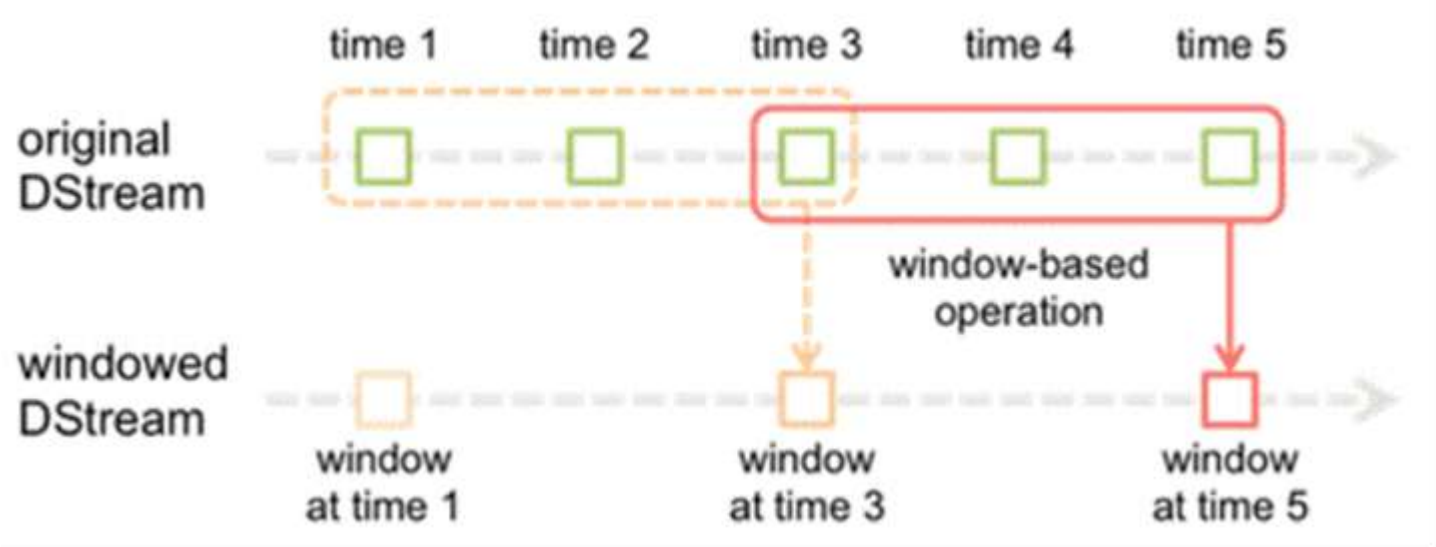
```
valpairs = words.map(word => (word, 1))
valwordCounts= pairs.reduceByKey(_ + _)
.saveAsHadoopFiles("hdfs://...")
```



DStream

- DStreamWindow Operation

saleAmounts.reduceByKeyAndWindow(_+_, Seconds(30), Seconds(20))



Spark Streaming应用场景

- 实时统计
- 实时告警
- 实时处理前端系统请求

Spark Streaming

```
SparkConf sparkConf = new SparkConf().setAppName("JavaKafkaWordCount");
// Create the context with a 1 second batch size
JavaStreamingContext jssc = new JavaStreamingContext(sparkConf,  Durations.milliseconds(1000));

int numThreads = Integer.parseInt(args[3]);
Map<String, Integer> topicMap = new HashMap<String, Integer>();
String[] topics = args[2].split(",");
for (String topic: topics) {
  topicMap.put(topic, numThreads);
}

JavaPairReceiverInputDStream<String, String> messages =
        KafkaUtils.createStream(jssc, args[0], args[1], topicMap);

JavaDStream<String> lines = messages.map(new Function<Tuple2<String, String>, String>() {
  @Override
  public String call(Tuple2<String, String> tuple2) {
    return tuple2._2();
  }
});

JavaDStream<String> words = lines.flatMap(new FlatMapFunction<String, String>() {
  @Override
  public Iterable<String> call(String x) {
    return Lists.newArrayList(SPACE.split(x));
  }
```

设置批量处理频率：1s 一次

打开 kafka 输入

Spark Streaming

```
JavaStreamingContext jssc;
jssc = new JavaStreamingContext(conf, Durations.milliseconds(1000));
```

```
collect ...
提交时间:  2015-11-03 16:21:03.635,  处理时间:  2015-11-03 16:21:04.296,  延时=661ms
提交时间:  2015-11-03 16:21:03.335,  处理时间:  2015-11-03 16:21:04.279,  延时=944ms
提交时间:  2015-11-03 16:21:03.935,  处理时间:  2015-11-03 16:21:04.325,  延时=390ms
collect ...
提交时间:  2015-11-03 16:21:04.235,  处理时间:  2015-11-03 16:21:05.286,  延时=1051ms
提交时间:  2015-11-03 16:21:04.835,  处理时间:  2015-11-03 16:21:05.305,  延时=470ms
提交时间:  2015-11-03 16:21:04.535,  处理时间:  2015-11-03 16:21:05.297,  延时=762ms
collect ...
提交时间:  2015-11-03 16:21:05.735,  处理时间:  2015-11-03 16:21:06.3,  延时=565ms
提交时间:  2015-11-03 16:21:05.135,  处理时间:  2015-11-03 16:21:06.281,  延时=1146ms
提交时间:  2015-11-03 16:21:05.435,  处理时间:  2015-11-03 16:21:06.288,  延时=853ms
提交时间:  2015-11-03 16:21:06.035,  处理时间:  2015-11-03 16:21:06.308,  延时=273ms
```

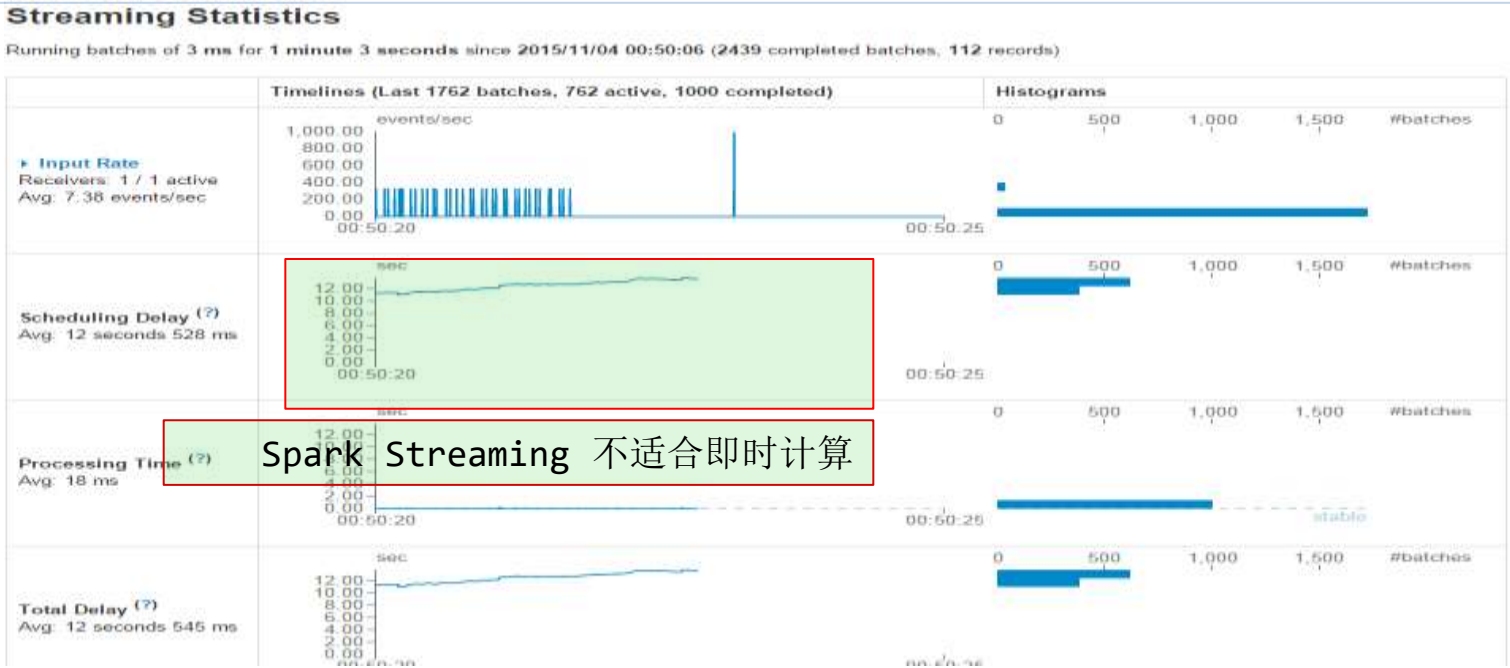
Spark Streaming

```
JavaStreamingContext jssc;
jssc = new JavaStreamingContext(conf, Durations.milliseconds(300));
```

```
提交时间:  2015-11-03 16:24:51.551,  处理时间:  2015-11-03 16:24:51.883,  延时=332ms
collect ...
提交时间:  2015-11-03 16:24:51.851,  处理时间:  2015-11-03 16:24:52.179,  延时=328ms
collect ...
提交时间:  2015-11-03 16:24:52.152,  处理时间:  2015-11-03 16:24:52.485,  延时=333ms
collect ...
提交时间:  2015-11-03 16:24:52.452,  处理时间:  2015-11-03 16:24:52.78,  延时=328ms
collect ...
提交时间:  2015-11-03 16:24:52.752,  处理时间:  2015-11-03 16:24:53.078,  延时=326ms
collect ...
提交时间:  2015-11-03 16:24:53.052,  处理时间:  2015-11-03 16:24:53.38,  延时=328ms
collect ...
提交时间:  2015-11-03 16:24:53.352,  处理时间:  2015-11-03 16:24:53.676,  延时=324ms
collect ...
提交时间:  2015-11-03 16:24:53.652,  处理时间:  2015-11-03 16:24:53.978,  延时=326ms
```

Spark Streaming

```
JavaStreamingContext jssc;
jssc = new JavaStreamingContext(conf, Durations.milliseconds(3));
```



Spark BlinkDB

BlinkDB用于在海量数据上运行交互式SQL查询的大规模并行查询引擎。允许用户通过权衡数据精度来提升查询响应时间，其数据的精度被控制在允许的误差范围内。

为了达到这个目标，BlinkDB 使用两个核心思想:

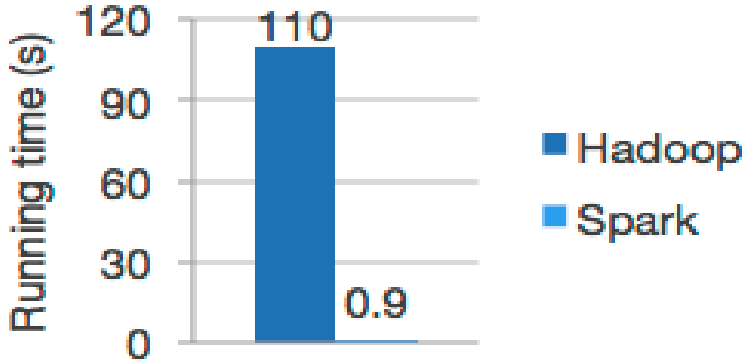
一个自适应优化框架，从原始数据随着时间的推移建立并维护一组多维样本；

一个动态样本选择策略，选择一个适当大小的示例基于查询的准确性和（或）响应时间需求。

BlinkDB 演示了在 Amazon EC2 集群部署了 100 个节点，大约 17TB 的数据中查询不到 2 秒钟，比 Hive 快 200 倍，错误率在 2-10%。

MLlib

- 机器学习库：提供高质量的算法,比MapReduce快100倍
- 高性能：



Logistic regression in Hadoop and Spark

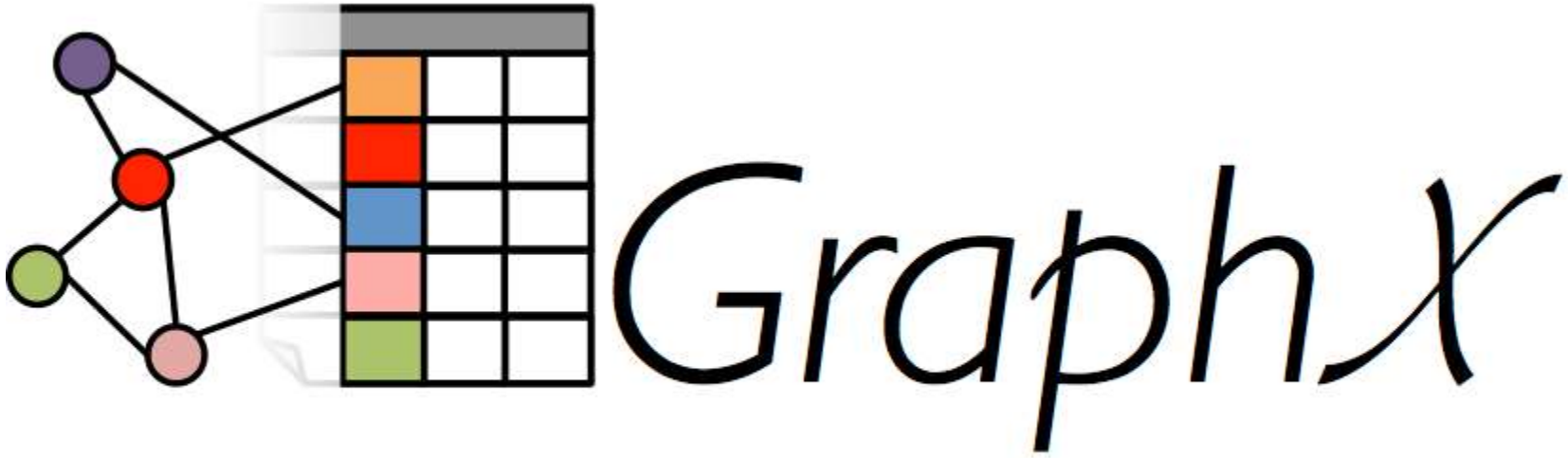
- Spark擅长迭代计算，这可以使mllib运行的更快，另外，MLlib也包含高效的算法，利用spark的迭代优势，从而产生百倍效果

MLlib

- 目标：简化机器学习过程，提供可扩展性
- 提供基本的机器学习算法和功能，包括：
  - 分类、
  - 回归、
  - 聚类、
  - 协同过滤、
  - 降维。
- 提供底层优化
- 提供管道化API

GraphX

- 并行的图计算：alpha版本，已并入spark



Spark例子

wordcount例子：

1、初始化，构建SparkContext。

```
val ssc=new
SparkContext(args(0),"WordCount",System.getenv("SPARK_HOM
E"),Seq(System.getenv("SPARK_EXAMPLES_JAR")))
```

2、输入算子　val lines=ssc.textFlle(args(1))

3、变换算子　val words=lines.flatMap(x=>x.split(" "))

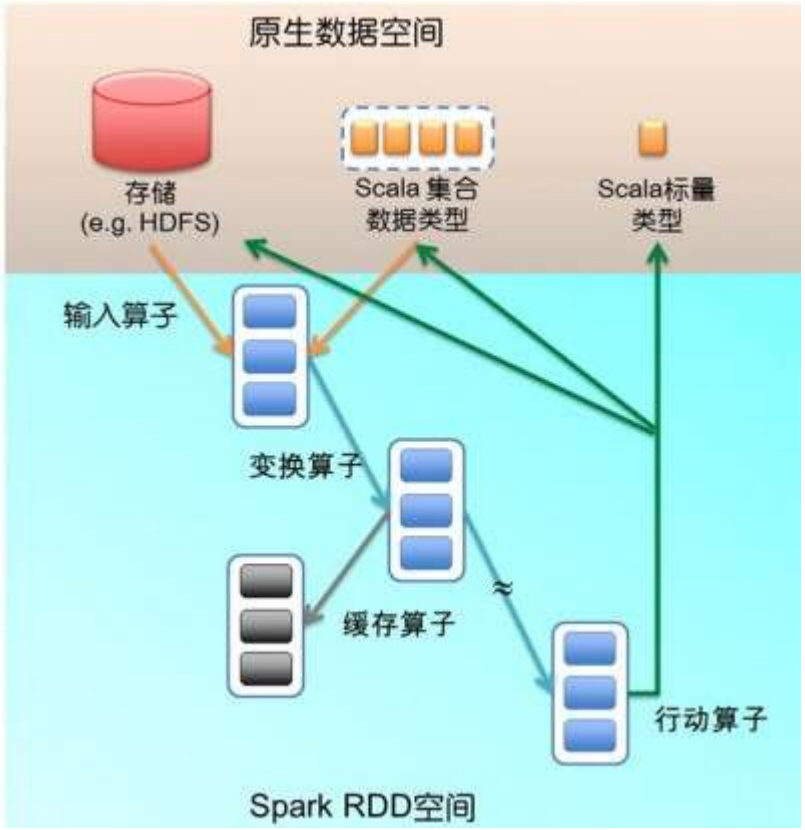
4、缓存算子　words.cache() //缓存

5、变换算子

```
val wordCounts=words.map(x=>(x,1))
```

```
val red=wordCounts.reduceByKey((a,b)=>{a+b})
```

6、行动算子　red.saveAsTextFile("/root/Desktop/out")



Spark例子

- Spark安装
- 交互式开发

./spark-shell --master local

./spark-shell --master yarn-client

- IDE开发——WordCount

Spark例子

```
val textFile = sc.textFile( “/spark/core/README.md” )

textFile.count
textFile.first

val linesWithSpark = textFile.filter(line =>
line.contains("spark"))

linesWithSpark.count
```

Spark 例子

```
./bin/spark-submit
--name <application-name>
--class <main-class>
--master yarn-cluster
--num-executors 2
--executor-memory 4G
--executor-cores 2
--driver-memory 1G
****.jar
<inpuPath1> <inputPath2> ...
<outputPath>
```

- spark-submit提交程序，会读取配置文件conf/spark-defaults.conf
- 命令行中参数优先级高于配置文件
- 程序中SparkConf对象的设置优先级最高

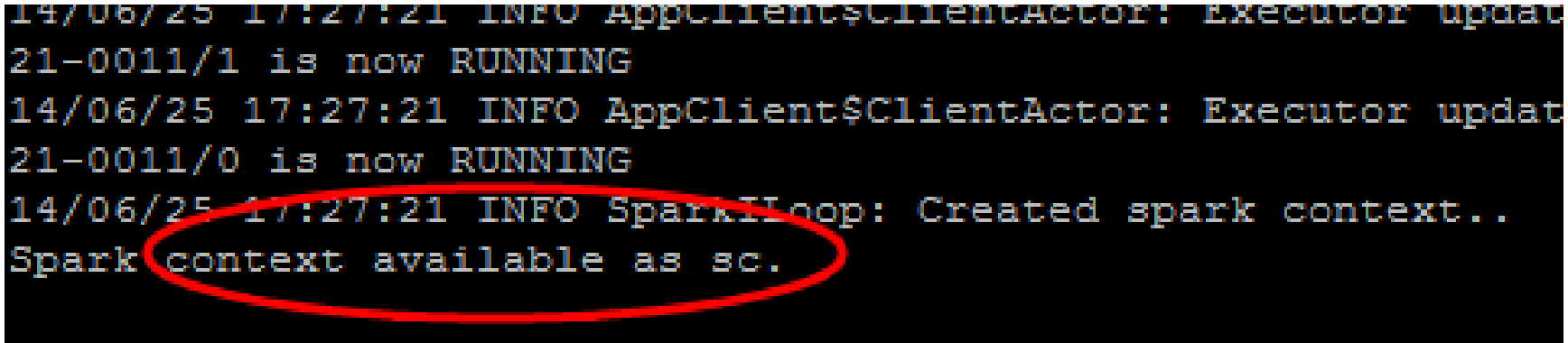
Spark 例子

- Lines = sc.textFile(“hdfs://…”) //加载进来成为RDD
- Errors = lines.filter(_.startsWith(“ERROR”))//transformation
- Errors.persist() //缓存RDD
- Mysql_errors=errors.fileter(_.contains(“MySQL”)).count //action
- http_errors = errors.filter(_.contains(“Http”)).count

Spark 例子

启动spark： spark-shell

- val rdd= sc.parallelize(List(1,2,3,4,5,6).map(2*_）).filter(_>5). collect
- sc： sparkContext，在spark启动时已被实例化

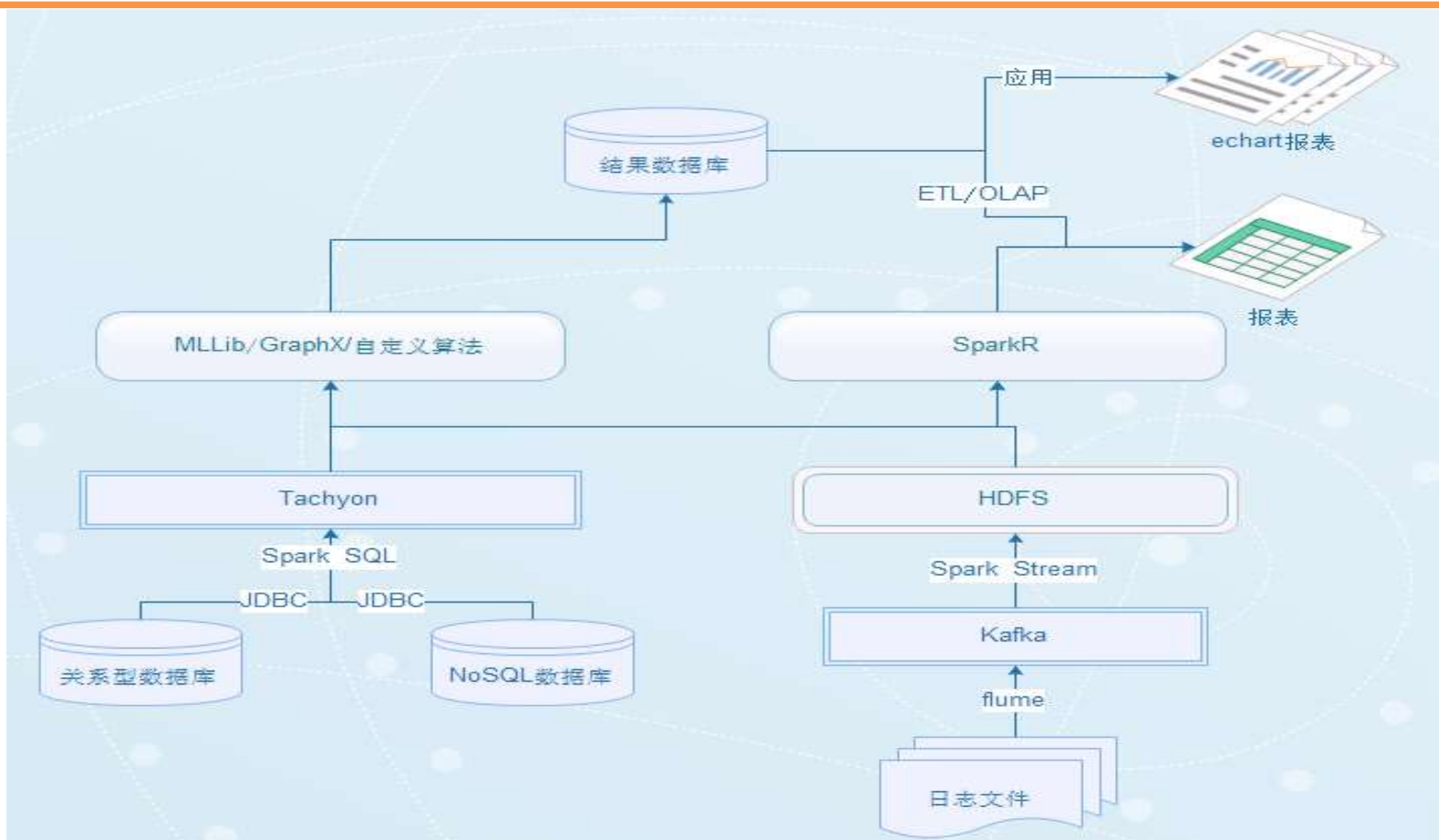


加载数据

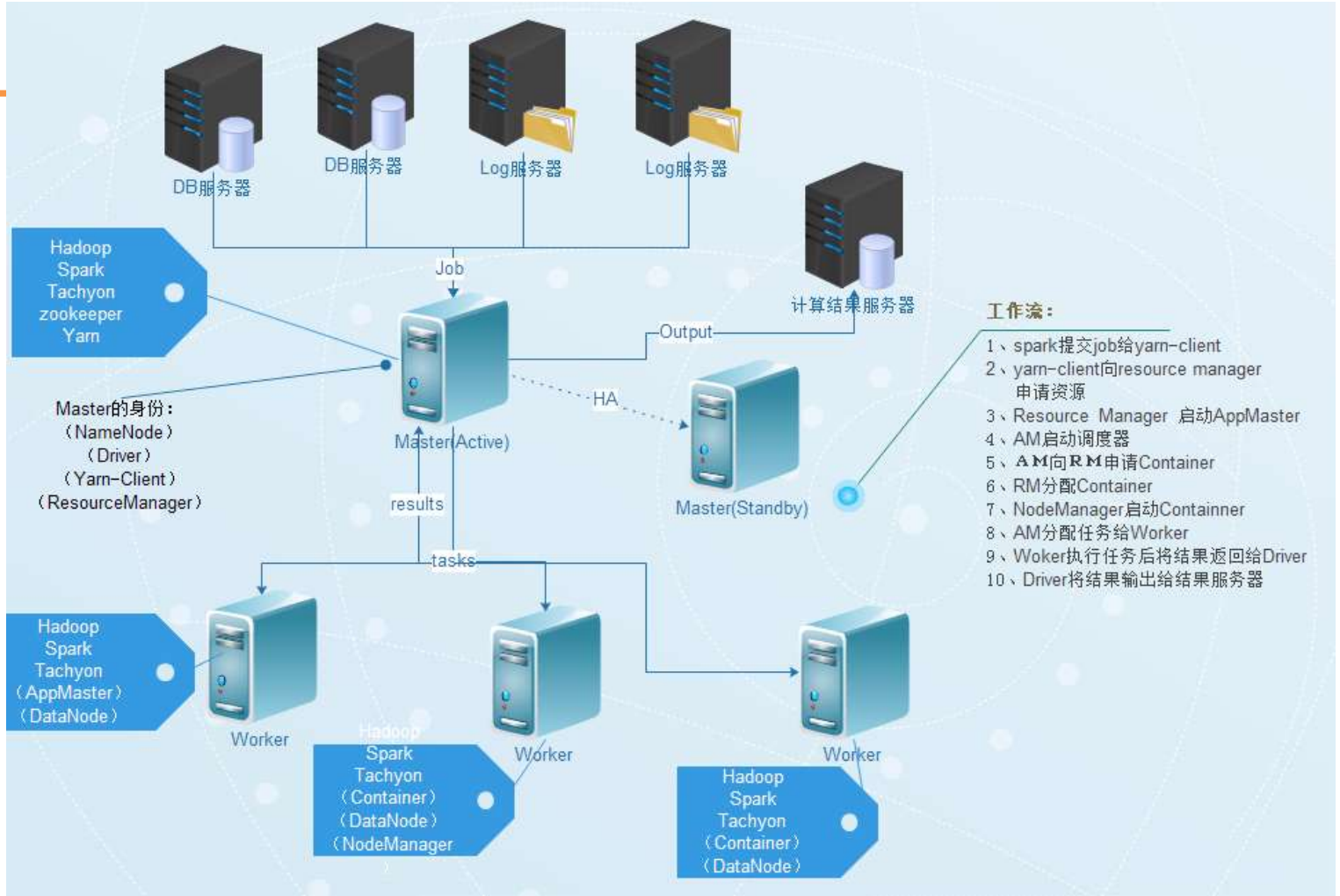
- 加载hdfs上的数据:
  sc.textFile(“hdfs://192.168.1.177:9000/tmp/SogouLabDic.dic”)

- 加载本地数据:val rdd =
  sc.textFile(“/usr/local/SogouLabDic.dic”)

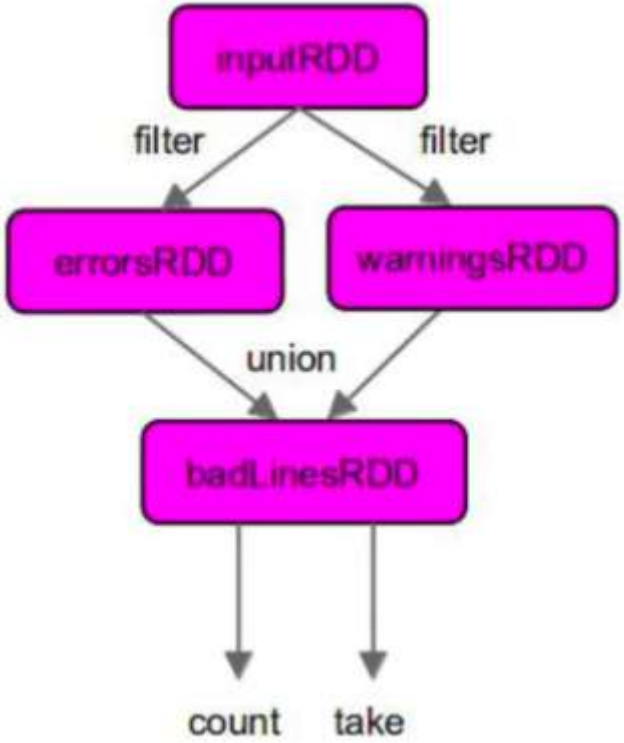
数据流程


应用
echart报表
结果数据库
ETL/OLAP
报表
MLLib/GraphX/自定义算法
SparkR
Tachyon
HDFS
Spark SQL
Spark Stream
JDBC
JDBC
关系型数据库
NoSQL数据库
Kafka
flume
日志文件

部署


DB服务器
DB服务器
Log服务器
Log服务器
计算结果服务器
Hadoop
Spark
Tachyon
zookeeper
Yarn
Job
Output
HA
Master的身份：
（NameNode）
（Driver）
（Yarn-Client）
（ResourceManager）
Master(Active)
Master(Standby)
results
tasks
Hadoop
Spark
Tachyon
（AppMaster）
（DataNode）
Worker
Hadoop
Spark
Tachyon
（Container）
（DataNode）
（NodeManager
Worker
Hadoop
Spark
Tachyon
（Container）
（DataNode）
Worker
工作流：
1、spark提交job给yarn-client
2、yarn-client向resource manager
申请资源
3、Resource Manager 启动AppMaster
4、AM启动调度器
5、AM向RM申请Container
6、RM分配Container
7、NodeManager启动Containner
8、AM分配任务给Worker
9、Woker执行任务后将结果返回给Driver
10、Driver将结果输出给结果服务器

Spark RDD Lineage




```
val inputRDD = sc.textFile("log.txt")

val errorsRDD = inputRDD.filter(line =>
line.contains("error"))

val warningsRDD = inputRDD.filter(line =>
line.contains(“warning"))

badLinesRDD = errorsRDD.union(warningsRDD)

println("Input had " + badLinesRDD.count() + "
concerning lines")
```

An RDD has enough information about how it was derived from other datasets(its lineage)

RDD容错

通过RDD Lineage，重新计算丢失的数据，如

Messages = textFile(...).filter(_.contains("error")) .map(_.split('₩t'))

RDD Persistence

- 缓存级别
  - memory_only (default), memory_and_disk等
- API
  - persist(StorageLevel)
  - cache() 等同于 persist(StorageLevel.MEMORY_ONLY)

RDD Persistence

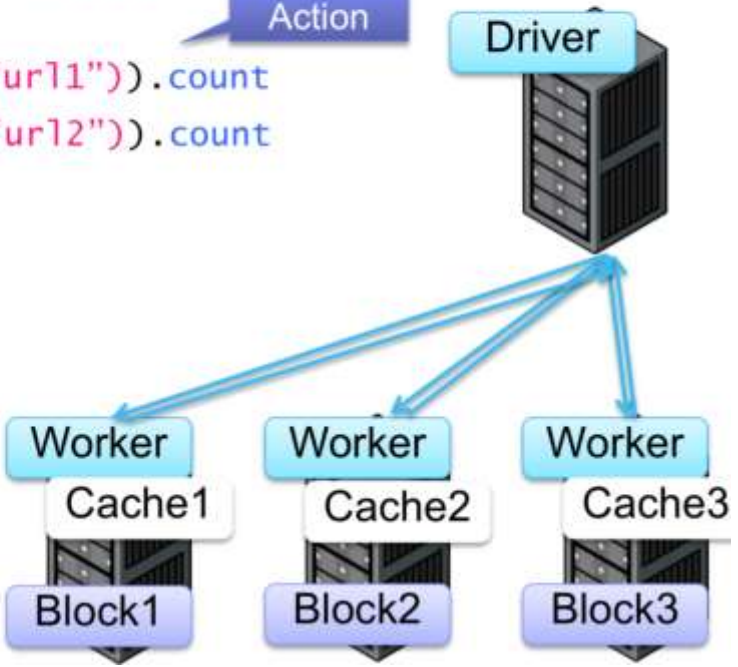
```
lines = sc.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
messages = errors.map(_.split('\t')(2))
cachedMsgs = messages.cache()
cachedMsgs.filter(_.contains("url1")).count
cachedMsgs.filter(_.contains("url2")).count
```

Base RDD

Transformed RDD

Action

缓存错误日志，过滤出各url的个数



RDD Persistence

Level	Space Used	CPU time	In memory	On Disk	Nodes with data	Comments
MEMORY_ONLY	High	Low	Y	N	1	
MEMORY_ONLY_2	High	Low	Y	N	2	
MEMORY_ONLY_SER	Low	High	Y	N	1	
MEMORY_ONLY_SER_2	Low	High	Y	N	2	
MEMORY_AND_DISK	High	Medium	Some	Some	1	Spills to disk if there is too much data to fit in memory.
MEMORY_AND_DISK_2	High	Medium	Some	Some	2	Spills to disk if there is too much data to fit in memory.
MEMORY_AND_DISK_SER	Low	High	Some	Some	1	Spills to disk if there is too much data to fit in memory.
MEMORY_AND_DISK_SER_2	Low	High	Some	Some	2	Spills to disk if there is too much data to fit in memory.

Spark 优化——代码

- RDD重用需要持久化

```
val rdd = sc.textFile("hdfs://***")
rdd.filter(...).count
rdd.filter(...).count

val rdd = sc.textFile("hdfs://***")
rdd.persist(StorageLevel.MEMORY_AND_D ISK_SER)
rdd.filter(...).count
rdd.filter(...).count
```

Spark 优化——代码

合理使用broadcast——map join

```
val rdd3 = rdd1.join(rdd2)

val rdd2Collect = rdd2.collect()
val rdd2Broadcast = sc.broadcast(rdd2Collect)
val rdd3 = rdd1.map(rdd2Broadcast...)
```

Spark 优化——代码

- 使用Kryo进行序列化

```
val conf = new SparkConf().setAppName(...)

// 设置序列化器
conf.set("spark.serializer",
"org.apache.spark.serializer.KryoSerializer")

// 注册自定义类型(要序列化)
conf.registerKryoClasses(Array(classOf[You rClass1],
classOf[YourClass2],...))
```

Spark 优化——代码

- 尽量减少shuffle，如reduceByKey、join
- 能使用reduceByKey，不用groupByKey
- 合理使用coalesce，如filter后

Spark 优化——参数

- Driver进程内存
- Driver进程CPU
- Executor进程内存
  - 运行task，Executor总内存20%
  - shuffle进行聚合，Executor总内存20%
  - RDD持久化，Executor总内存60%
- Executor进程CPU core数影响并行执行的 task数量

Spark 优化——参数

参数	说明
num-executors	Application总共有多少个Executor进程执行，初始可 以设置30-50，根据队列资源分配和资源使用情况设置
executor-cores	每个Executor进程的CPU core数，初始可以设置2-4
executor-memory	每个Executor进程的内存数量，初始可以设置4-8G
driver-memory	Driver进程的内存，一般1G左右
spark.default.par allelism	每个stage的task数量，设置为num-executors * executor-cores的2-3倍，充分利用申请的资源
spark.storage.me moryFraction	RDD持久化数据在Executor内存中能占的比例，持久 化操作较多可适当提高，频繁gc可适当降低
spark.shuffle.me moryFraction	Shuffle操作在Executor内存中能占的比例，shuffle操 作较多可适当提高，频繁gc可适当降低

Spark 优化——参数

```
./bin/spark-submit
--num-executors 50
--executor-cores 4
--executor-memory 8G
--driver-memory 1G
--conf spark.default.parallelism=600 --conf
spark.storage.memoryFraction=0.5 --conf
spark.shuffle.memoryFraction=0.3
```

谢谢！