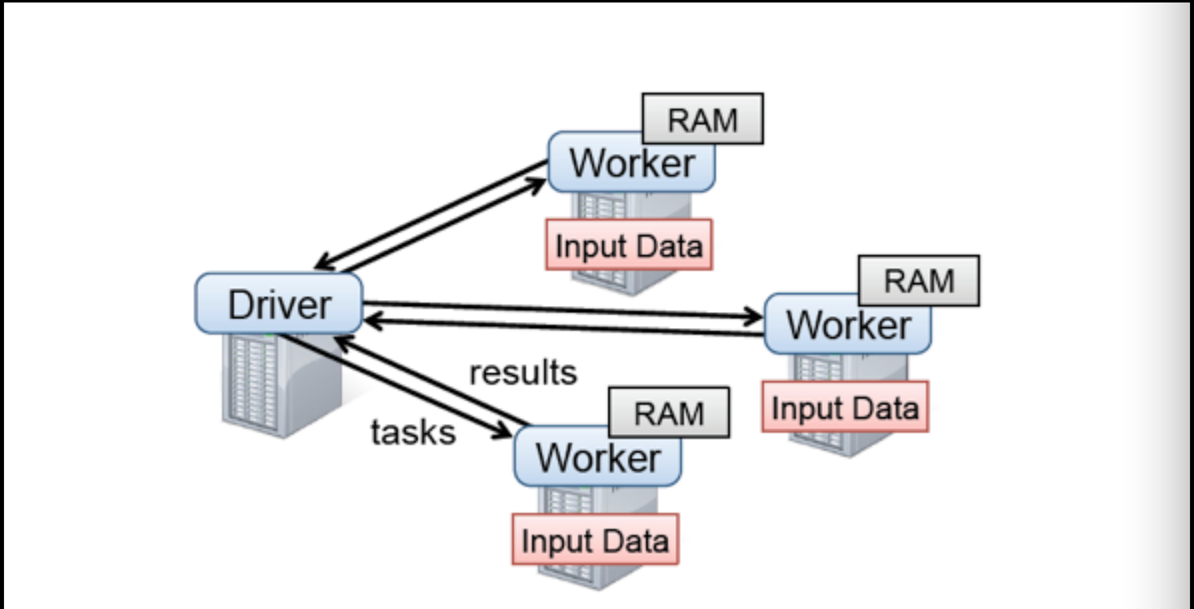
Spark Introduction

@CrazyJvm

什么是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.

就这么简单！



Driver程序启动很多workers，然后workers在(分布式)文件系统中读取数据后转化为RDD，最后对RDD在内存中进行缓存和计算。

为什么Spark的性能高？

1、in-memory computation

2、general computation graphs

可参考 Delay scheduling : A simple technique for achieving locality and fairness in cluster scheduling

提供了哪些语言的API？

Scala

Java

Python

怎样运行Spark？

- Local
- Standalone
- Mesos
- YARN

可参考：@明风Andy 出品的 [基于Spark on Yarn的淘宝数据挖掘平台]

Scala光速入门

玩Spark最好会点Scala。

直接引用Matei-Zaharia之前介绍Scala的示例

Scala语言？

1、基于JVM的FP+OO

2、静态类型

3、和JAVA可以互操作

可以在Scala的命令行里试试！

Scala--变量声明

```
var x : Int = 7

val x = 7 //自动类型推导

val y = “hi” //只读的 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 + 2
x => x + 2
_ +  2
x => {
      val numberToAdd = 2
      x + numberToAdd
}
//如果闭包很长，可以考虑作为参数传入
def addTwo(x : Int) : Int = x + 2
list.map(addTwo)
```

回到Spark

Spark：从RDD说起

RDDs : resilient distributed datasets

1. **immutable** collections of objects spread across a cluster
2. build through parallel transformations(map,filter,groupBy,join etc
3. automatically rebuild on failure (基于lineage和checkpoint)
4. different storage level（可(序列化)存在内存或者硬盘中）

单独看下StorageLevel

```scala
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)
```

来建立几个RDD

sc就是SparkContext,可以看做用户在spark上的入口。

```
scala> val text = sc.textFile("README.md")
13/12/17 14:00:57 INFO storage.MemoryStore: ensureFreeSpace(44405) called with curMem=67016, maxMem=340147568
13/12/17 14:00:57 INFO storage.MemoryStore: Block broadcast_2 stored as values to memory (estimated size 43.4 KB, free 324.3 MB)
text: org.apache.spark.rdd.RDD[String] = MappedRDD[5] at textFile at <console>:12

scala> val logon = sc.textFile("hdfs://server1:9000/data/logon/logon-1.log")
13/12/17 14:01:00 INFO storage.MemoryStore: ensureFreeSpace(44365) called with curMem=111421, maxMem=340147568
13/12/17 14:01:00 INFO storage.MemoryStore: Block broadcast_3 stored as values to memory (estimated size 43.3 KB, free 324.2 MB)
logon: org.apache.spark.rdd.RDD[String] = MappedRDD[7] at textFile at <console>:12
```

RDD到底是啥玩意儿？

internally, each RDD is characterized by five main properties:

- *** - 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)**

进一步看下RDD

- RDDs（可以从一系列数据源创建）
- Transformations (延迟执行)
- Action (def runJob[T, U: ClassManifest]

```
(rdd: RDD[T],
 func: Iterator[T] => U): Array[U])
```

Transformation和Action

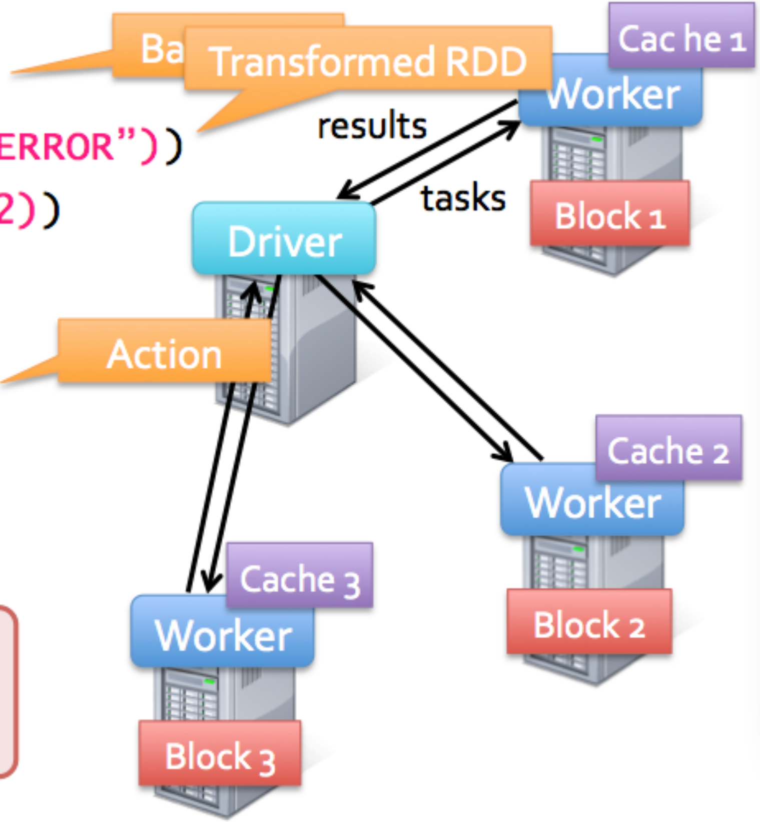
Transformations	$map(f : \mathrm{T} \Rightarrow \mathrm{U})$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{RDD[U]}$
	$filter(f : \mathrm{T} \Rightarrow \mathrm{Bool})$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{RDD[T]}$
	$flatMap(f : \mathrm{T} \Rightarrow \mathrm{Seq[U]})$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{RDD[U]}$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

举个经典的例子

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

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

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



可以略微简化下代码

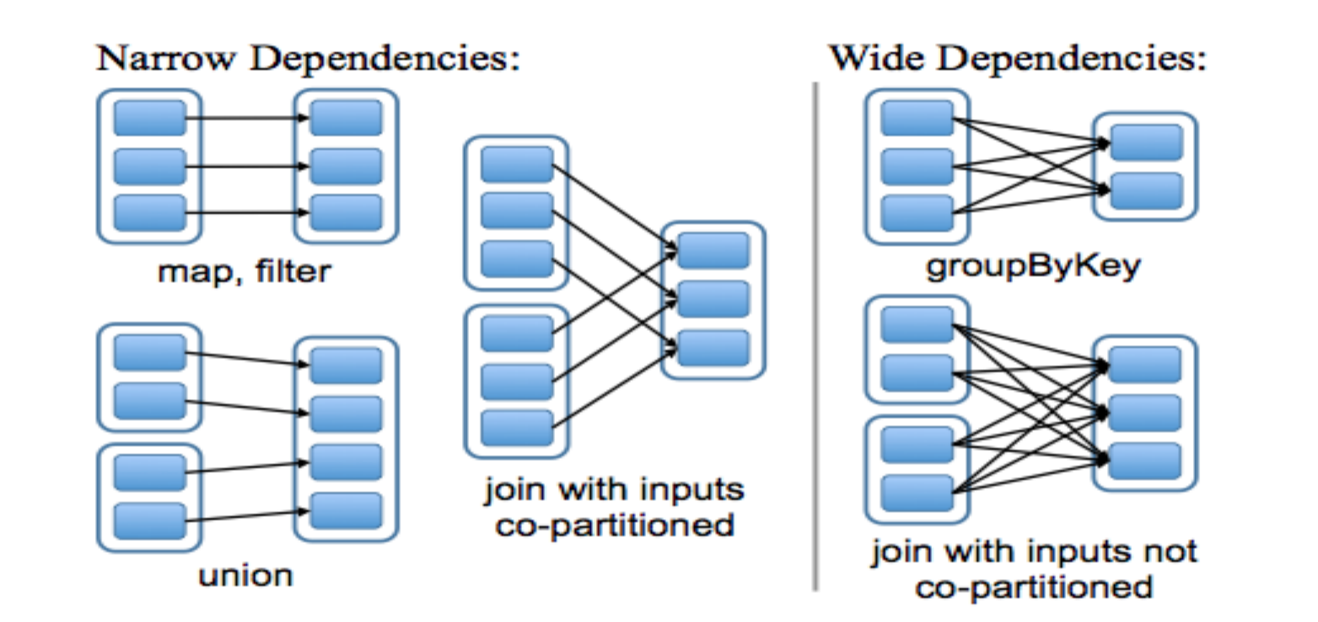
```
val message=spark.textFile("hdfs://...").filter(_.startsWith("ERROR")).map(_.split('\t')(2)).cache
def  getStatistic(keyword) : Long = message.filter(_.contains(keyword)).count
```

关于Cache多说几句

目前RDD的storage level一旦设定就不能再改变。以后可能会允许改变。(当然你可以使用unpersist去删除cache)

```scala
/**
 * Set this RDD's storage level to persist its values across operations after the first time
 * it is computed. This can only be used to assign a new storage level if the RDD does not
 * have a storage level set yet..
 */
def persist(newLevel: StorageLevel): RDD[T] = {
  // TODO: Handle changes of StorageLevel
  if (storageLevel != StorageLevel.NONE && newLevel != storageLevel) {
    throw new UnsupportedOperationException(
      "Cannot change storage level of an RDD after it was already assigned a level")
  }
  storageLevel = newLevel
  // Register the RDD with the SparkContext
  sc.persistentRdds(id) = this
  this
}
```

容错



基于lineage(子RDD挂了可以从父RDD再次计算得来)和checkpoint(物化)

接下来，举几个RDD的例子。

RDD的创建

1、直接从集合转化

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

2、从各种(分布式)文件系统来

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

3、从现存的任何Hadoop InputFormat而来

```
sc.hadoopFile(keyClass, valClass, inputFormat,conf)
```

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 style 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 style RDD 续

```
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)
```

会写wordcount程序了吗？

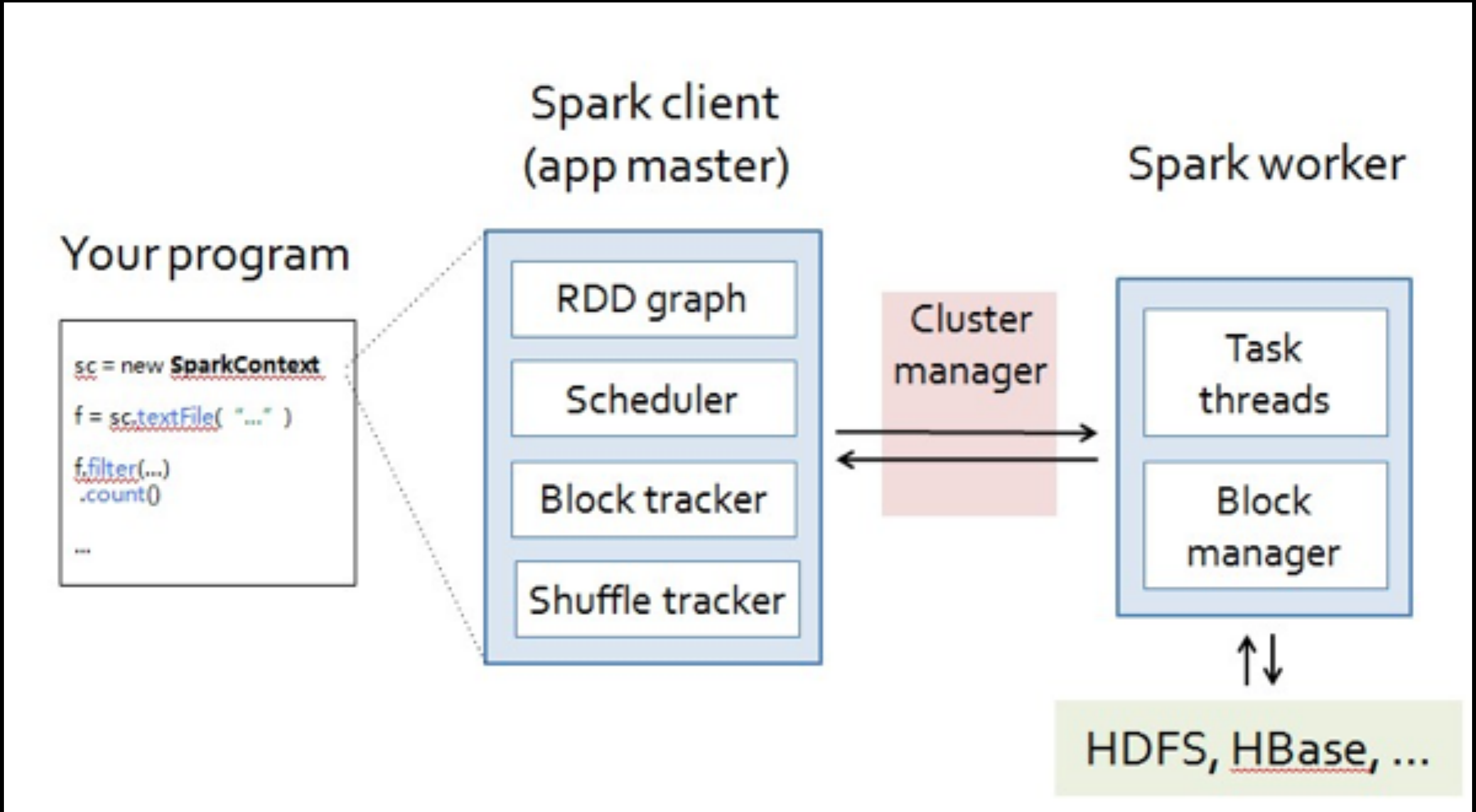
word count

```
sc.textFile("README.md").flatMap(_.split(' ')).map((_,1)).reduceByKey(_+_)
```

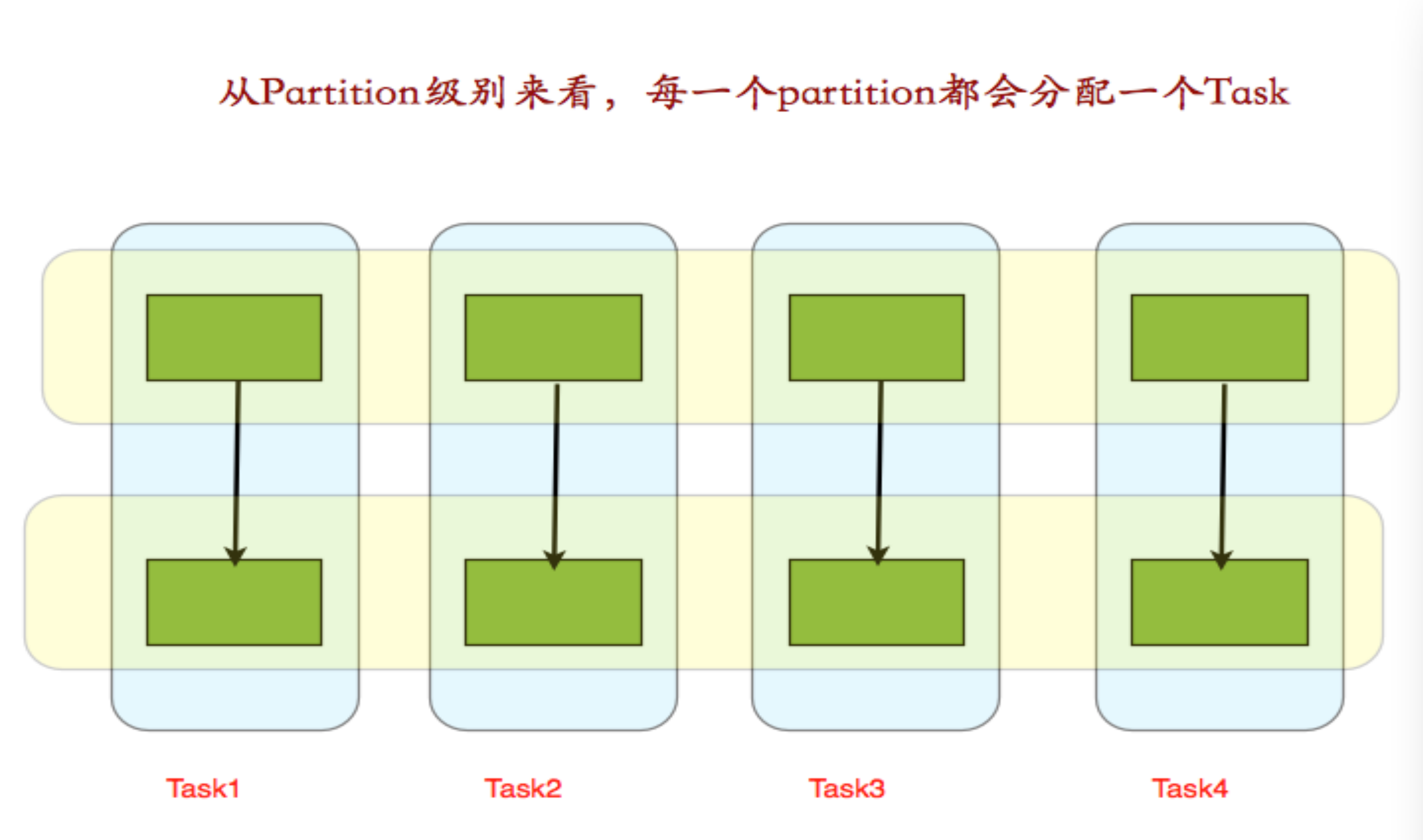
感受一下 ╭(╯^╰)╮

接下来看下作业调度相关内容

Spark组件


Spark client
(app master)
Spark worker
Your program
sc = new SparkContext
f = sc.textFile("…")
f.filter(…)
.count()
...
RDD graph
Scheduler
Block tracker
Shuffle tracker
Cluster manager
Task threads
Block manager
HDFS, HBase, …

RDD Graph


从Partition级别来看，每一个partition都会分配一个Task
Task1
Task2
Task3
Task4

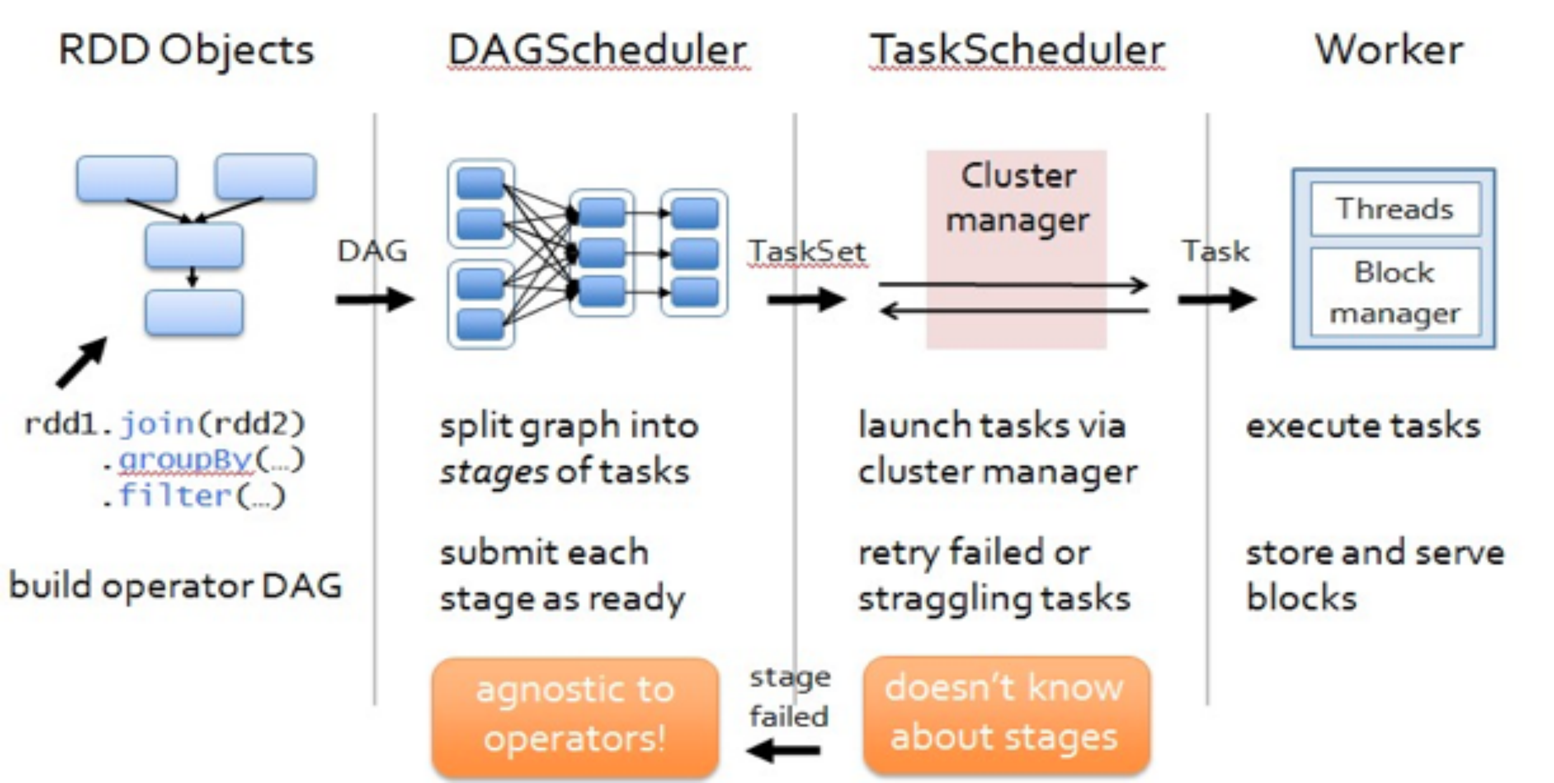
数据本地性

1、文件系统的本地性

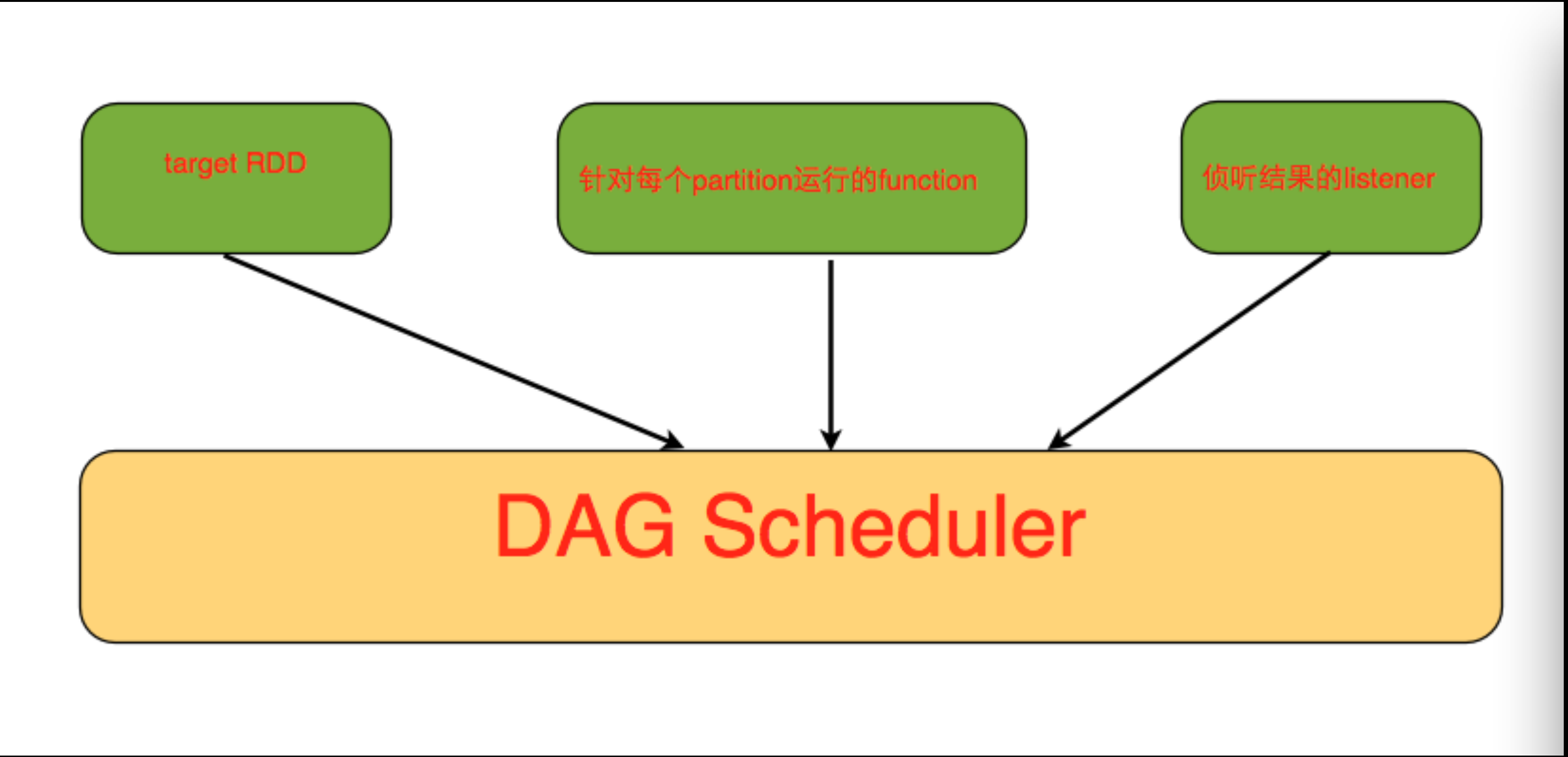
2、内存本地性

3、内存假如被evict掉后退化为文件系统的本地性

作业与任务调度


RDD Objects
DAGScheduler
TaskScheduler
Worker
DAG
TaskSet
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
launch tasks via cluster manager
retry failed or straggling tasks
execute tasks
store and serve blocks
agnostic to operators!
stage failed
doesn't know about stages

DAG Scheduler



DAG Scheduler的角色

1、为每个Job分割stage，同时会决定最佳路径，并且DAG Sheduler会记录哪个RDD或者stage的输出被物化，从而来找到一个最优调度方案。

2、将TaskSet传给TaskScheduler

3、重新提交那些输出lost的stage

DAG Scheduler优化

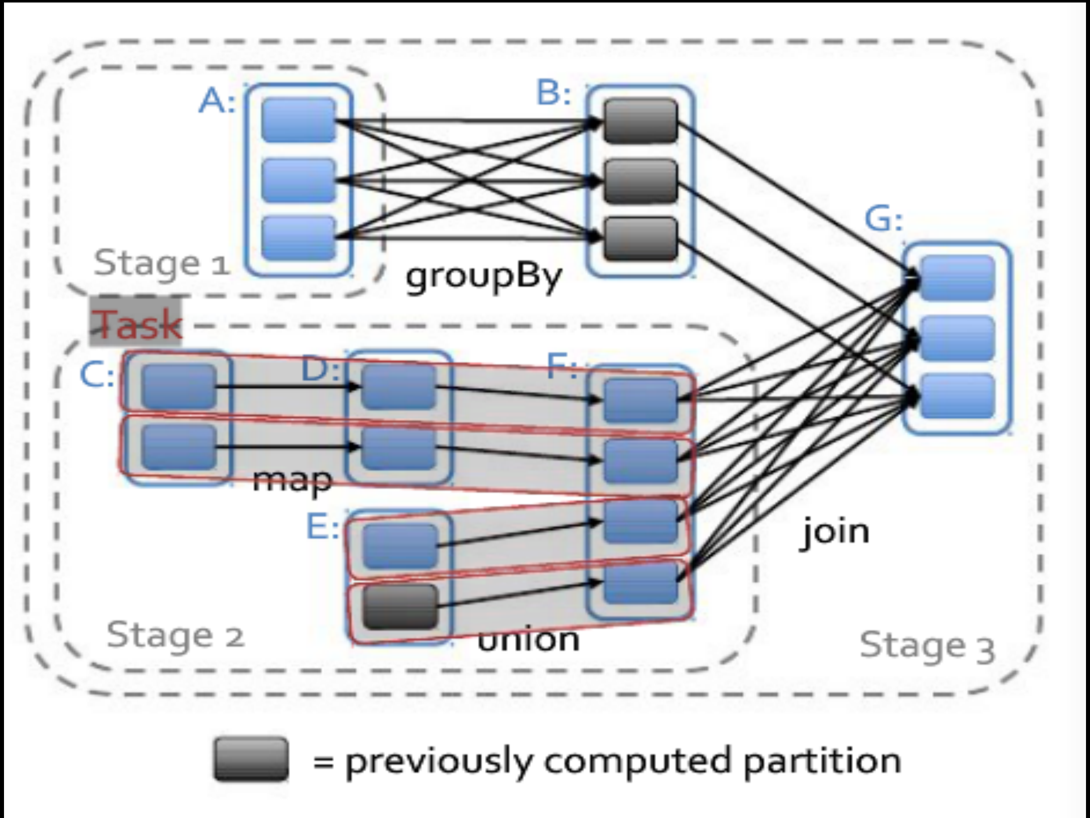
1、某个stage内pipeline执行

对比：1+1+1+1+1=5（1）

1+1=2 2+1=3 3+1=4 4+1=5（2）

2、基于partition选择合适的join算法最小化shuffle

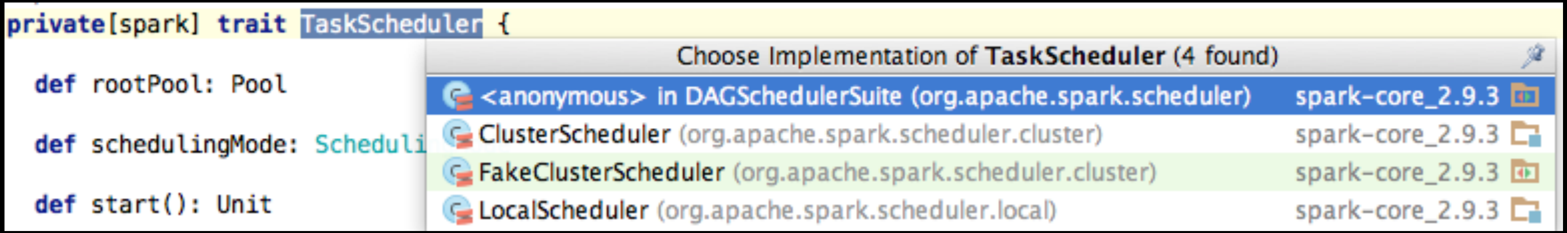
3、重用已经cache过的数据


A:
B:
groupBy
G:
Stage 1
Task
C:
D:
F:
map
E:
join
Stage 2
union
Stage 3
= previously computed partition

TaskScheduler

1、提交tasks到集群并执行，假如出错就重试。
2、假如shuffle输出lost就报告fetch failed错误
3、遇到straggle task需要放到别的node上重试

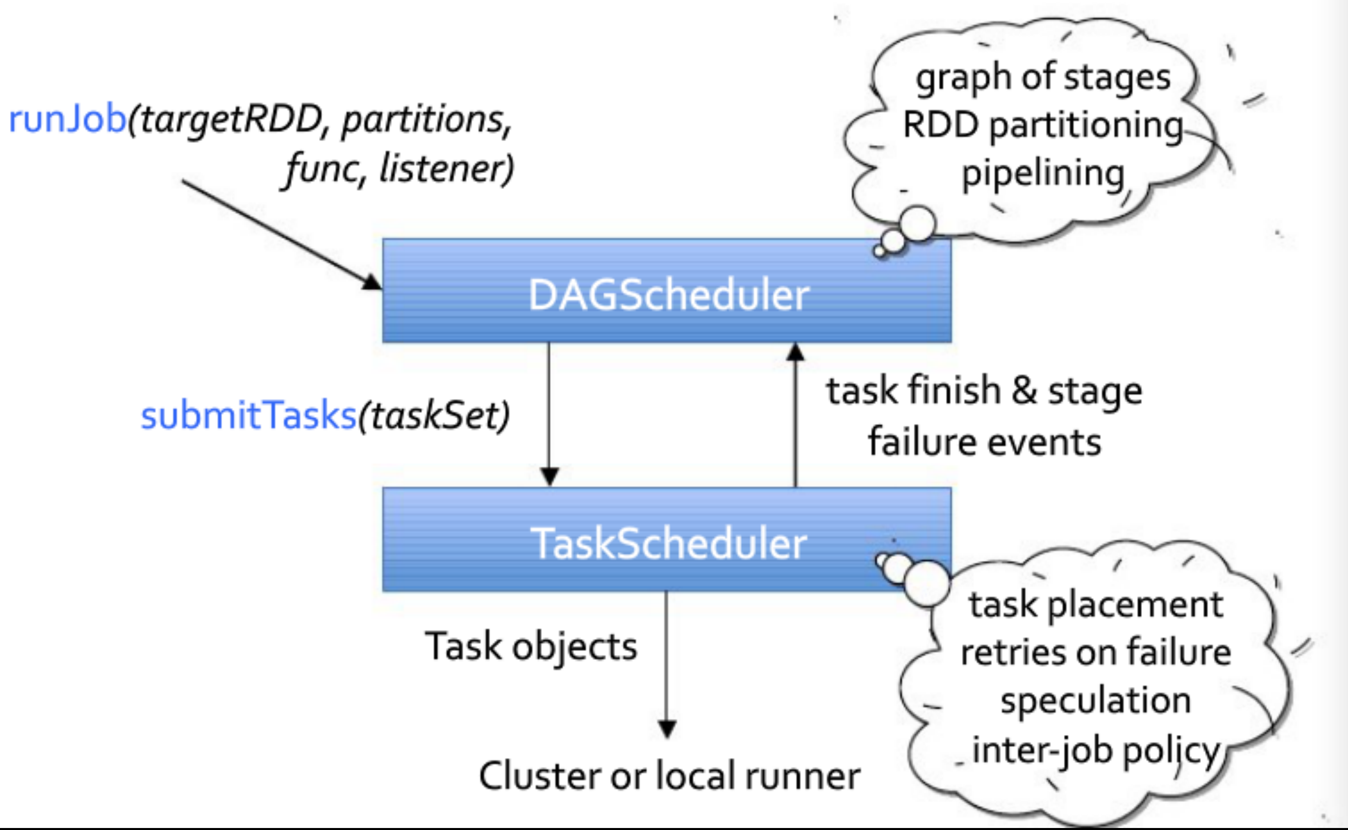
实现：



Task细节

物化Shuffle输出到内存或者硬盘以便重试

Job调度执行流程


runJob(targetRDD, partitions, func, listener)
graph of stages
RDD partitioning
pipelining
DAGScheduler
submitTasks(taskSet)
task finish & stage failure events
TaskScheduler
task placement
retries on failure
speculation
inter-job policy
Task objects
Cluster or local runner

Broadcast variables

BT style的变量广播

场景：lookup表，mapside join

参考：Performance and Scalability of Broadcast in Spark

Accumulators

类似于MapReduce中的counter

最后谈下Spark(性能)的优化

CASE 1

问题：序列化Task太大

解决：大对象使用broadcast variables

ps: spark 0.9版本中task序列化大小大于100k将会打印警告日志

参考：https://spark-project.atlassian.net/browse/SPARK-590

CASE 2

问题: val rdd = xx.map(..).filter(..).filter(..) ①

rdd.map(xx).reduceBy… ②

经过语句①, ②可能会产生很多小任务

解决: 使用coalesce或者repartition

CASE 3

问题:每个record的开销较大

典型场景 rdd.map{x=>conn=getDBConn;conn.write(x.toString);conn.close}

解决方案:

rdd.mapPartitions(records=>conn=getDBConn;for(item <- records) write(item.toString);conn.close)

CASE 4

问题:任务执行速度倾斜

解决:

1、数据倾斜(一般是partition key取的不好)

考虑其它的并行处理方式

中间可以加入一步aggregation

2、Worker倾斜(在某些worker上的executor不给力)

设置spark.speculation=true

把那些持续不给力的node去掉

CASE 5

问题：不设置spark.local.dir

这是spark写shuffle输出的地方

解决:可以设置一组disk

spark.local.dir=/mn1/spark, /mnt2/spar, /mnt3/spark

CASE 6

问题：reducer数量不合适

解决：看实际情况(请忽略这句废话！)

太多的reducer，造成很多的小任务，以此产生很多启动任务的开销。

太少的reducer，慢！并且可能造成OOM。

CASE 7

问题：collect输出大量结果慢

```scala
/**
 * Return an array that contains all of the elements in this RDD.
 */
def collect(): Array[T] = {
  val results = sc.runJob(this, (iter: Iterator[T]) => iter.toArray)
  Array.concat(results: _*)
}
```

解决:

不要用collect，直接输出到分布式文件系统。

CASE 8

问题：序列化不给力！

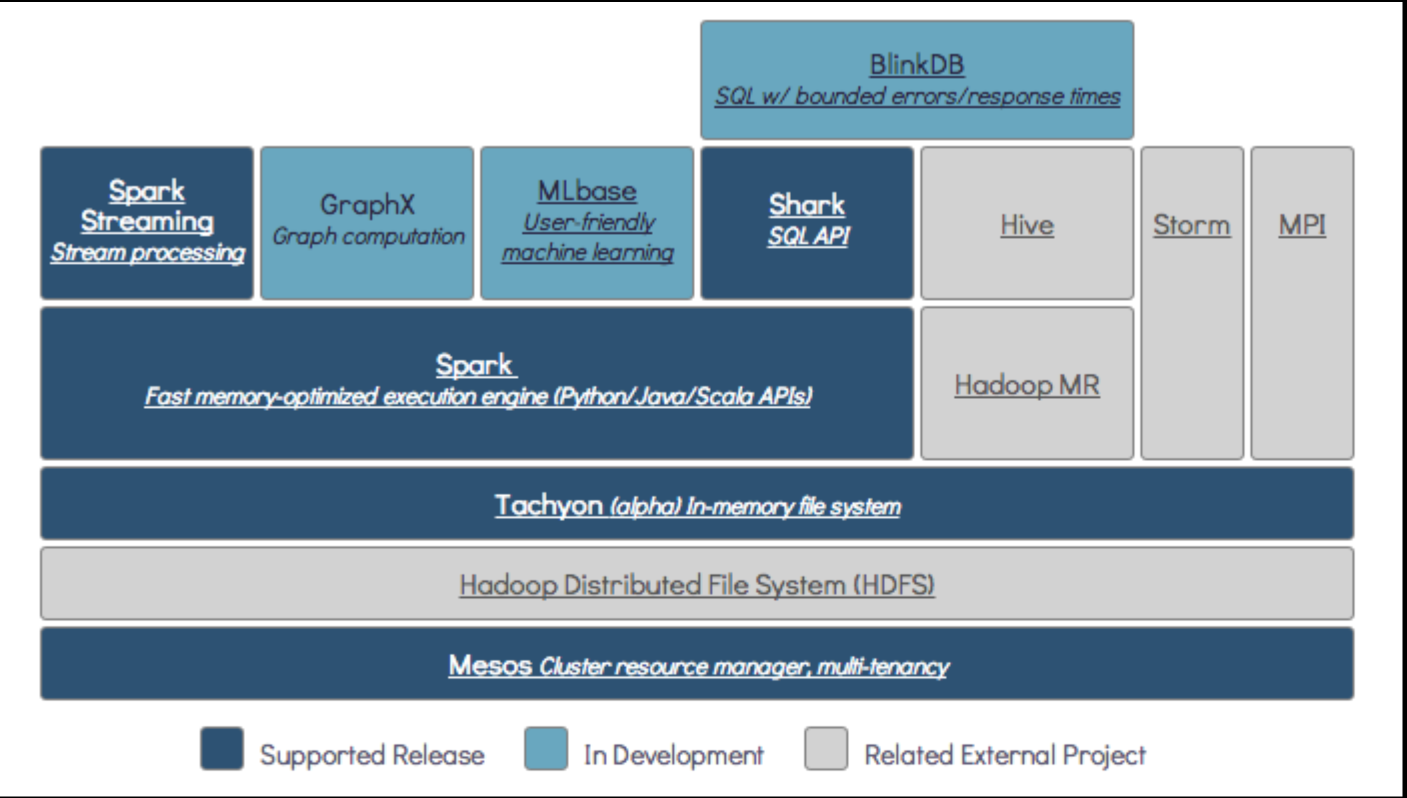
spark默认的序列化机制是JAVA的ObjectOutputStream（兼容性好。但！又大又慢）

解决：

使用Kyro serialization(兼容性略差。但！又快又小)

SPARK就介绍到这，但，这其实只是个开始！

BDAS


BlinkDB
SQL w/ bounded errors/response times
Spark Streaming
Stream processing
GraphX
Graph computation
MLbase
User-friendly machine learning
Shark
SQL API
Hive
Storm
MPI
Spark
Fast memory-optimized execution engine (Python/Java/Scala APIs)
Hadoop MR
Tachyon (alpha) In-memory file system
Hadoop Distributed File System (HDFS)
Mesos Cluster resource manager, multi-tenancy
Supported Release
In Development
Related External Project

Q & A

推荐关注的新浪微博

@hashjoin

@李浩源HY

@明风Andy

@Andrew-Xia

@连城404

@吴甘沙

@尹绪森

还有谁我漏掉了？

thanks all, enjoy spark!

@CrazyJvm