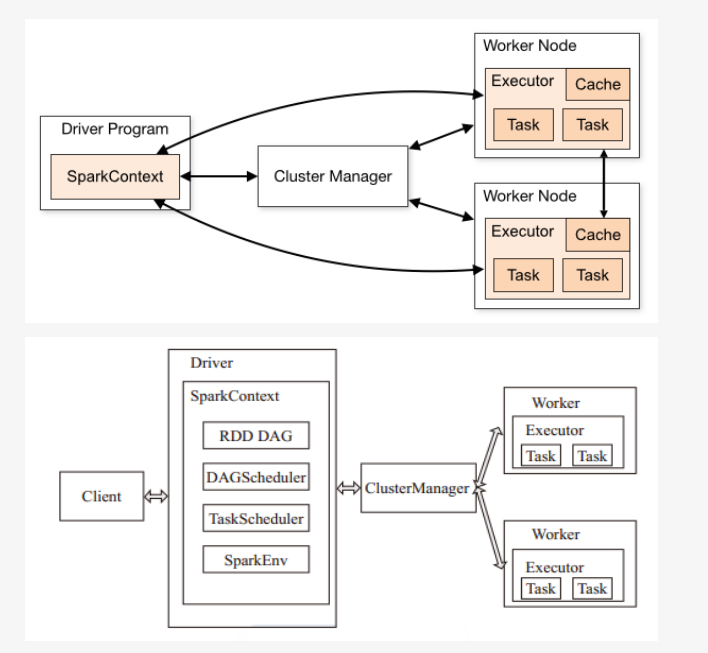
**1,spark basic architect**



2,

**spark 运行流程：**

Spark架构采用了分布式计算中的Master-Slave模型。Master是对应集群中的含有Master进程的节点，Slave是集群中含有Worker进程的节点。

* Master作为整个集群的控制器，负责整个集群的正常运行；
* Worker相当于计算节点，接收主节点命令与进行状态汇报；
* Executor负责任务的执行；
* Client作为用户的客户端负责提交应用；
* Driver负责控制一个应用的执行。

Spark集群部署后，需要在主节点和从节点分别启动Master进程和Worker进程，对整个集群进行控制。在一个Spark应用的执行过程中，Driver和Worker是两个重要角色。Driver 程序是应用逻辑执行的起点，负责作业的调度，即Task任务的分发，而多个Worker用来管理计算节点和创建Executor并行处理任务。在执行阶段，Driver会将Task和Task所依赖的file和jar序列化后传递给对应的Worker机器，同时Executor对相应数据分区的任务进行处理。

1. Excecutor /Task 每个程序自有，不同程序互相隔离，task多线程并行
2. 集群对Spark透明，Spark只要能获取相关节点和进程
3. Driver 与Executor保持通信，协作处理

**三种集群模式：**

1.Standalone 独立集群

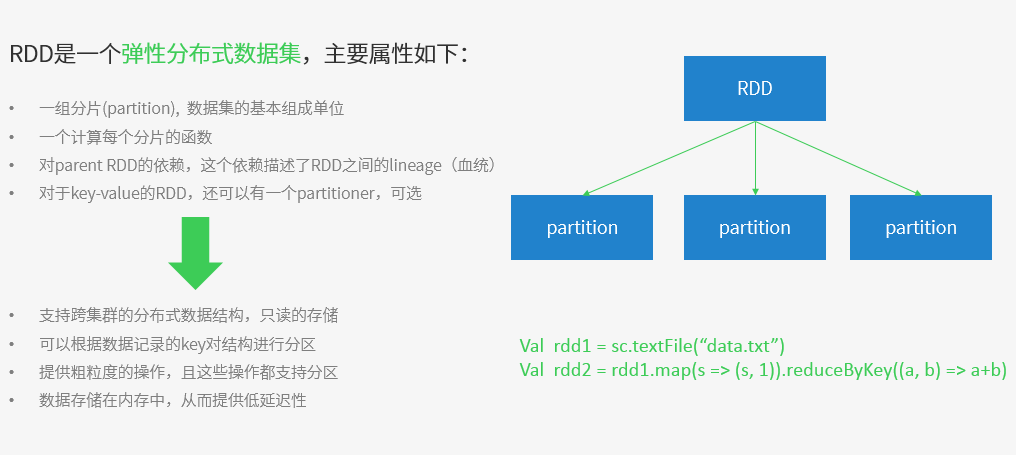
2.Mesos, apache mesos

3.Yarn, hadoop yarn

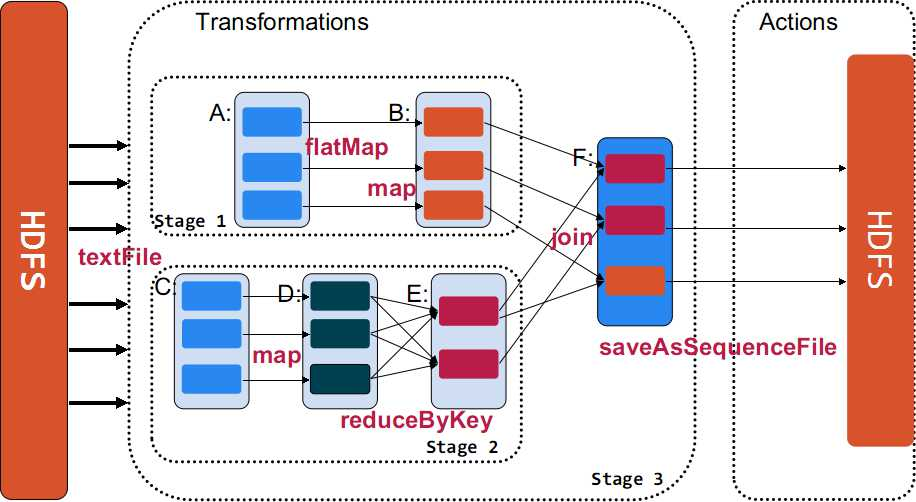
**基本概念：**

* Application =>Spark的应用程序，包含一个Driver program和若干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 => 是为Application运行在Worker node上的一个进程，该进程负责运行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Executor来处理任务
* Cluster Manager =>在集群上获取资源的外部服务 (例如：Standalone、Mesos、Yarn)
* Worker Node => 集群中任何可以运行Application代码的节点，运行一个或多个Executor进程
* 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task，每组任务被称为Stage，也称TaskSet
* RDD => 是Resilient distributed datasets的简称，中文为弹性分布式数据集;是Spark最核心的模块和类
* DAGScheduler => 根据Job构建基于Stage的DAG，并提交Stage给TaskScheduler
* TaskScheduler => 将Taskset提交给Worker node集群运行并返回结果
* Transformations => 是Spark API的一种类型，Transformation返回值还是一个RDD，所有的Transformation采用的都是懒策略，如果只是将Transformation提交是不会执行计算的
* Action => 是Spark API的一种类型，Action返回值不是一个RDD，而是一个scala集合；计算只有在Action被提交的时候计算才被触发。

**Spark核心概念之RDD**



**Spark核心概念之Transformations / Actions**



Transformation返回值还是一个RDD。它使用了链式调用的设计模式，对一个RDD进行计算后，变换成另外一个RDD，然后这个RDD又可以进行另外一次转换。这个过程是分布式的。 Action返回值不是一个RDD。它要么是一个Scala的普通集合，要么是一个值，要么是空，最终或返回到Driver程序，或把RDD写入到文件系统中。

Action是返回值返回给driver或者存储到文件，是RDD到result的变换，Transformation是RDD到RDD的变换。

只有action执行时，rdd才会被计算生成，这是rdd懒惰执行的根本所在。

**Spark核心概念之Jobs / Stage**

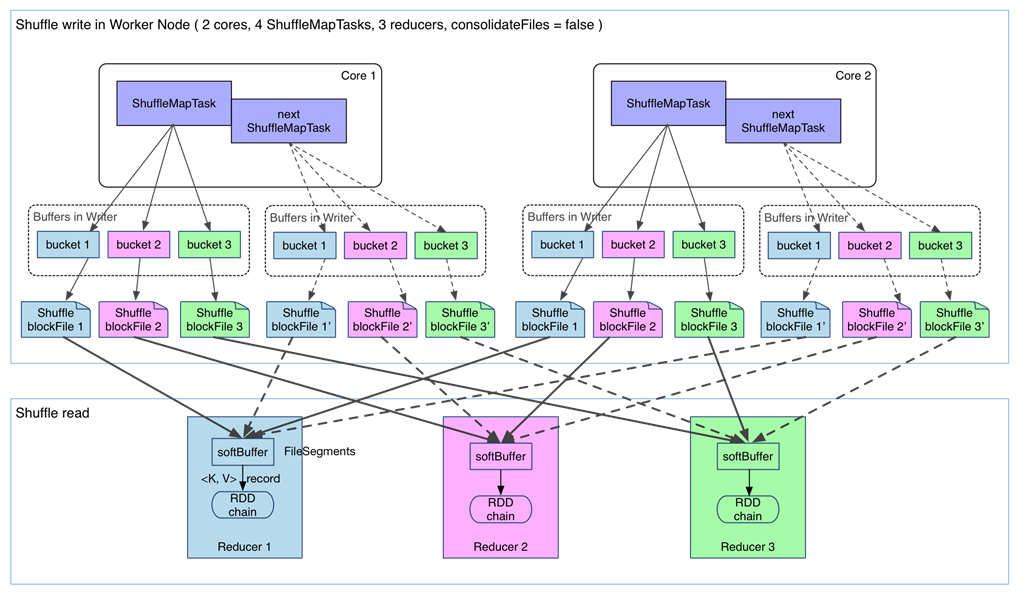
Job => 包含多个task的并行计算，一个action触发一个job

stage => 一个job会被拆为多组task，每组任务称为一个stage，以shuffle进行划分

**Spark核心概念之Shuffle**

以reduceByKey为例解释shuffle过程。

**在没有task的文件分片合并下的shuffle过程如下**：（spark.shuffle.consolidateFiles=false）



**fetch 来的数据存放到哪里？**

刚 fetch 来的 FileSegment 存放在 softBuffer 缓冲区，经过处理后的数据放在内存 + 磁盘上。这里我们主要讨论处理后的数据，可以灵活设置这些数据是“只用内存”还是“内存＋磁盘”。如果spark.shuffle.spill = false就只用内存。由于不要求数据有序，shuffle write 的任务很简单：将数据 partition 好，并持久化。之所以要持久化，一方面是要减少内存存储空间压力，另一方面也是为了 fault-tolerance。

shuffle之所以需要把中间结果放到磁盘文件中，是因为虽然上一批task结束了，下一批task还需要使用内存。如果全部放在内存中，内存会不够。另外一方面为了容错，防止任务挂掉。

**存在问题如下：**

1. 产生的 FileSegment 过多。每个 ShuffleMapTask 产生 R（reducer 个数）个 FileSegment，M 个 ShuffleMapTask 就会产生 M \* R 个文件。一般 Spark job 的 M 和 R 都很大，因此磁盘上会存在大量的数据文件。
2. 缓冲区占用内存空间大。每个 ShuffleMapTask 需要开 R 个 bucket，M 个 ShuffleMapTask 就会产生 MR 个 bucket。虽然一个 ShuffleMapTask 结束后，对应的缓冲区可以被回收，但一个 worker node 上同时存在的 bucket 个数可以达到 cores R 个（一般 worker 同时可以运行 cores 个 ShuffleMapTask），占用的内存空间也就达到了cores× R × 32 KB。对于 8 核 1000 个 reducer 来说，占用内存就是 256MB。

为了解决上述问题，我们可以使用文件合并的功能。

**在进行task的文件分片合并下的shuffle过程如下**：（spark.shuffle.consolidateFiles=true）

可以明显看出，在一个 core 上连续执行的 ShuffleMapTasks 可以共用一个输出文件 ShuffleFile。先执行完的 ShuffleMapTask 形成 ShuffleBlock i，后执行的 ShuffleMapTask 可以将输出数据直接追加到 ShuffleBlock i 后面，形成 ShuffleBlock i'，每个 ShuffleBlock 被称为 FileSegment。下一个 stage 的 reducer 只需要 fetch 整个 ShuffleFile 就行了。这样，每个 worker 持有的文件数降为 cores× R。FileConsolidation 功能可以通过spark.shuffle.consolidateFiles=true来开启。

**Spark核心概念之Cache**

val rdd1 = ... // 读取hdfs数据，加载成RDD  
rdd1.cache  
​  
val rdd2 = rdd1.map(...)  
val rdd3 = rdd1.filter(...)  
​  
rdd2.take(10).foreach(println)  
rdd3.take(10).foreach(println)  
​  
rdd1.unpersist

cache和unpersisit两个操作比较特殊，他们既不是action也不是transformation。cache会将标记需要缓存的rdd，真正缓存是在第一次被相关action调用后才缓存；unpersisit是抹掉该标记，并且立刻释放内存。只有action执行时，rdd1才会开始创建并进行后续的rdd变换计算。

cache其实也是调用的persist持久化函数，只是选择的持久化级别为MEMORY\_ONLY。

persist支持的RDD持久化级别如下：

需要注意的问题：

Cache或shuffle场景序列化时， spark序列化不支持protobuf message，需要java 可以serializable的对象。一旦在序列化用到不支持java serializable的对象就会出现上述错误。

Spark只要写磁盘，就会用到序列化。除了shuffle阶段和persist会序列化，其他时候RDD处理都在内存中，不会用到序列化。

**Spark Streaming运行原理**

spark程序是使用一个spark应用实例一次性对一批历史数据进行处理，spark streaming是将持续不断输入的数据流转换成多个batch分片，使用一批spark应用实例进行处理。

从原理上看，把传统的spark批处理程序变成streaming程序，spark需要构建什么？

需要构建4个东西：

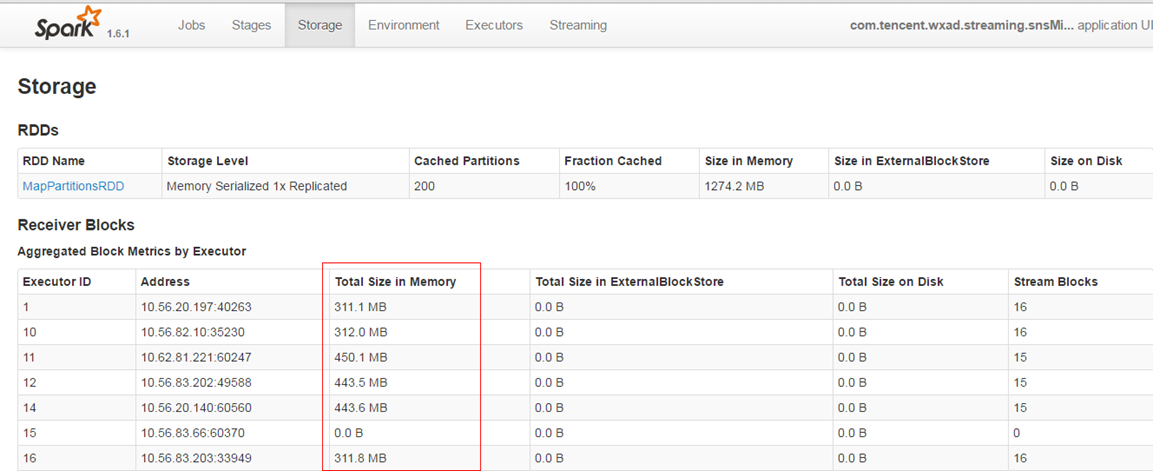
1. 一个静态的 RDD DAG 的模板，来表示处理逻辑；
2. 一个动态的工作控制器，将连续的 streaming data 切分数据片段，并按照模板复制出新的 RDD ；
3. DAG 的实例，对数据片段进行处理；
4. Receiver进行原始数据的产生和导入；Receiver将接收到的数据合并为数据块并存到内存或硬盘中，供后续batch RDD进行消费；
5. 对长时运行任务的保障，包括输入数据的失效后的重构，处理任务的失败后的重调。

具体streaming的详细原理可以参考广点通出品的源码解析文章：

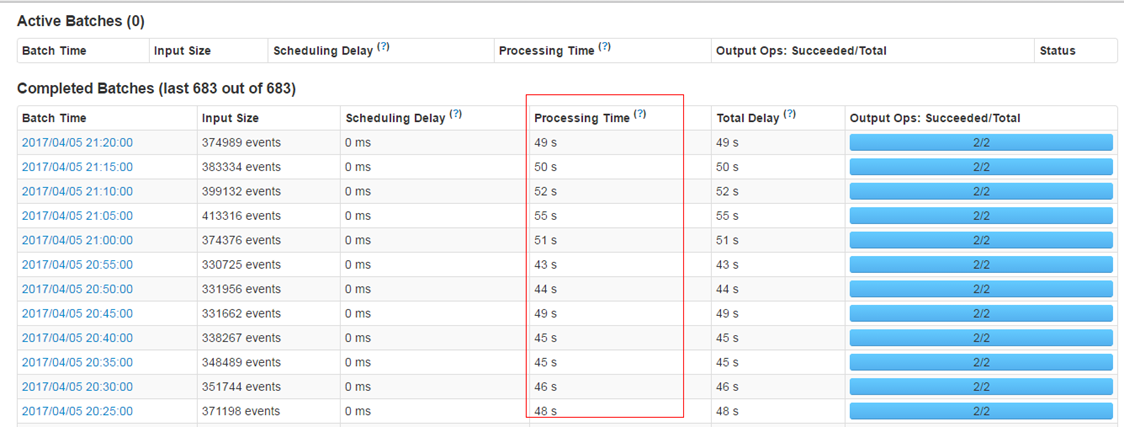
<https://github.com/lw-lin/CoolplaySpark/blob/master/Spark%20Streaming%20%E6%BA%90%E7%A0%81%E8%A7%A3%E6%9E%90%E7%B3%BB%E5%88%97/0.1%20Spark%20Streaming%20%E5%AE%9E%E7%8E%B0%E6%80%9D%E8%B7%AF%E4%B8%8E%E6%A8%A1%E5%9D%97%E6%A6%82%E8%BF%B0.md#24>

**对于spark streaming需要注意以下三点：**

1. 尽量保证每个work节点中的数据不要落盘，以提升执行效率。

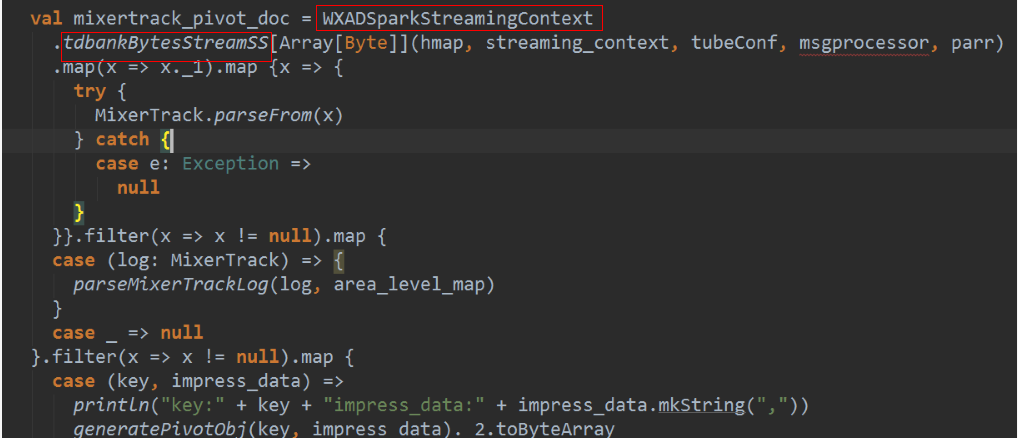


1. 保证每个batch的数据能够在batch interval时间内处理完毕，以免造成数据堆积。



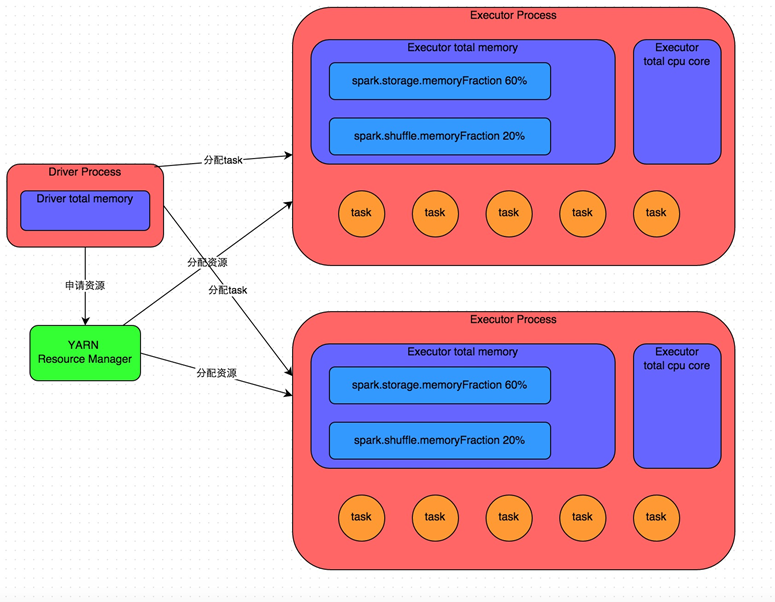
1. 使用steven提供的框架进行数据接收时的预处理，减少不必要数据的存储和传输。从tdbank中接收后转储前进行过滤，而不是在task具体处理时才进行过滤。





**Spark 资源调优**

**内存管理：**



Executor的内存主要分为三块：

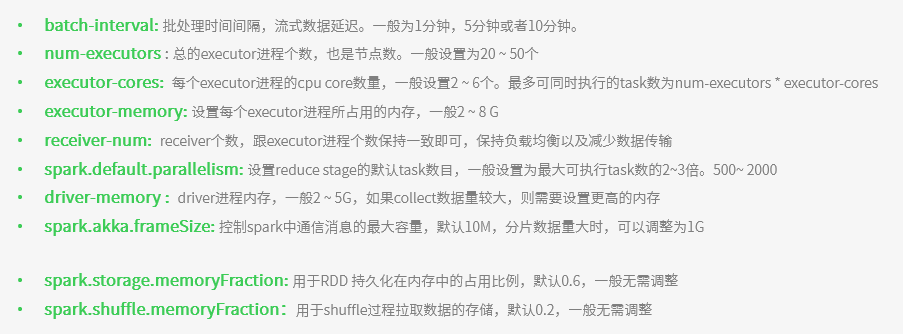
第一块是让task执行我们自己编写的代码时使用，默认是占Executor总内存的20%；

第二块是让task通过shuffle过程拉取了上一个stage的task的输出后，进行聚合等操作时使用，默认也是占Executor总内存的20%；

第三块是让RDD持久化时使用，默认占Executor总内存的60%。

每个task以及每个executor占用的内存需要分析一下。每个task处理一个partiiton的数据，分片太少，会造成内存不够。

其他资源配置：



具体调优可以参考美团点评出品的调优文章：

<http://tech.meituan.com/spark-tuning-basic.html>

<http://tech.meituan.com/spark-tuning-pro.html>

**Spark 环境搭建**

spark tdw以及tdbank api文档：

<http://git.code.oa.com/tdw/tdw-spark-common/wikis/api>

其他学习资料：

<http://km.oa.com/group/2430/articles/show/257492>

<https://blog.51cto.com/zero01/2091635>

原文链接：<https://www.qcloud.com/community/article/770164>

来源：<https://www.cnblogs.com/liuliliuli2017/p/6809094.html>

**18 Spark作业提交流程是怎么样的**

1. spark-submit 提交代码，执行 new SparkContext()，在 SparkContext 里构造 DAGScheduler 和 TaskScheduler。
2. TaskScheduler 会通过后台的一个进程，连接 Master，向 Master 注册 Application。
3. Master 接收到 Application 请求后，会使用相应的资源调度算法，在 Worker 上为这个 Application 启动多个 Executer。
4. Executor 启动后，会自己反向注册到 TaskScheduler 中。 所有 Executor 都注册到 Driver 上之后，SparkContext 结束初始化，接下来往下执行我们自己的代码。
5. 每执行到一个 Action，就会创建一个 Job。Job 会提交给 DAGScheduler。
6. DAGScheduler 会将 Job划分为多个 stage，然后每个 stage 创建一个 TaskSet。
7. TaskScheduler 会把每一个 TaskSet 里的 Task，提交到 Executor 上执行。
8. Executor 上有线程池，每接收到一个 Task，就用 TaskRunner 封装，然后从线程池里取出一个线程执行这个 task。(TaskRunner 将我们编写的代码，拷贝，反序列化，执行 Task，每个 Task 执行 RDD 里的一个 partition)

**数据倾斜：**

### （1）数据倾斜的介绍

#### 1）数据分区的策略：

     - **随机分区**：每一个数据分配的任意一个分区的概率是均等的  
     - **Hash分区**：使用数据的Hash分区值，%分区数。（导致数据倾斜的原因）  
     - **范围分区**：将数据范围划分，数据分配到不同的范围中（分布式的全局排序）

#### 2）数据倾斜的原因：

  Shuffle数据之后导致数据分布不均匀，但是所有节点的机器的性能都是一样的，程序也是一样的，就是数据量不一致，所以决定了task的执行时长就被数据量决定了。

#### 3）定位数据倾斜的代码：

  数据倾斜发生在shuffle过程，有shuffle过程的算子有：distinct、groupByKey、reduceByKey、aggregateByKey、join、cogroup、repartition等。或者查看哪一个task执行缓慢、内存溢出...

#### 4）查看数据倾斜的key的分布情况：

*//使用spark中的抽样算子sample，查看相应的key的分布*

val sampledPairs = pairs.sample(false, 0.1) *//抽样*

val sampledWordCounts = sampledPairs.countByKey()

sampledWordCounts.**foreach**(println(\_))

### （2）数据倾斜的解决方案

#### 1）过滤掉少数数据倾斜的key：

  如果发现导致数据倾斜的key是极少数，并且对计算本身影响不大，那么这种方案比较适用。  
   实现原理：通过spark的sample算子，定位到数据倾斜的key，然后使用filter算子将其过滤即可。

#### 2）提高shuffle的并行度：

   这是一种尝试性策略：就是提高增加shuffle read task的数量，可以让原本分配给一个task的多个key分配给多个task，从而让每个task处理比原来更少的数据。

#### 3）两阶段的聚合（局部聚合和全局聚合）：

**适用场景**：对RDD执行reduceByKey等这类有聚合操作的shuffle算子或者spark SQL使用使用group by语句进行分组聚合，比较适用。  
  
原理：将原本相同的key通过附加随机前缀的方式，变成多个不同key，就可以让原本被一个task处理的数据分散到多个task上做局部聚合，进行解决单个task处理数据量过多的问题。接着去除随机前缀，再次进行全局的聚合，就可以得到最终的结果。  
代码实现：

object \_01SparkDataSkewTwoStageOps {

def main(args: **Array**[String]): Unit = {

Logger.getLogger("org.apache.hadoop").setLevel(Level.WARN)

Logger.getLogger("org.apache.spark").setLevel(Level.WARN)

Logger.getLogger("org.project-spark").setLevel(Level.WARN)

val sc = SparkUtil.sparkContext("local[2]", "\_01SparkDataSkewTwoStageOps")

val **list** = **List**(

"hello you hello hello me",

"hello you hello hello shit",

"oh hello she study"

)

val listRDD = sc.parallelize(**list**)

val pairsRDD = listRDD.flatMap(line => line.split("\\s+")).map((\_, 1))

*//step 1 找到发生数据倾斜key*

val sampleRDD = pairsRDD.sample(**false**, 0.6)

val cbk= sampleRDD.countByKey()

*// cbkRDD.foreach(println)*

val sortedInfo = cbk.toBuffer.sortWith((t1, t2) => t1.\_2 > t2.\_2)

val dataSkewKey = sortedInfo.head.\_1

*// sortedInfo.foreach(println)*

println("发生了数据倾斜的Key：" + dataSkewKey)

*//step 2 给对应的key打上N以内的随机前缀*

val prefixPairsRDD = pairsRDD.map{**case** (word, count) => {

**if**(word.equals(dataSkewKey)) {

val random = **new** Random()

val prefix = random.nextInt(2)*//0 1*

(s"${prefix}\_${word}", count)

} **else** {

(word, count)

}

}}

prefixPairsRDD.**foreach**(println)

*//step 3 局部聚合*

val partAggrInfo = prefixPairsRDD.reduceByKey(\_+\_)

println("===============>局部聚合之后的结果：")

partAggrInfo.**foreach**(println)

*//step 4 全局聚合*

*//step 4.1 去掉前缀*

val unPrefixPairRDD = partAggrInfo.map{**case** (word, count) => {

**if**(word.contains("\_")) {

(word.substring(word.indexOf("\_") + 1), count)

} **else** {

(word, count)

}

}}

println("================>去掉随机前缀之后的结果：")

unPrefixPairRDD.**foreach**(println)

*// step 4.2 全局聚合*

val fullAggrInfo = unPrefixPairRDD.reduceByKey(\_+\_)

println("===============>全局聚合之后的结果：")

fullAggrInfo.**foreach**(println)

sc.stop()

}

}

#### 4）将reduce join 转换为map join（大小表）：

   适用场景：在对RDD使用join操作，或者是在sparksql 中使用join语句的时候，而且join操作中的一个RDD或者表的数据量比较小，此方法适用  
   实现原理：有reduce join的过程一定有shuffle，有shuffle就可能出现数据的倾斜，所以将reduce join使用map join 代替。如果一个RDD是比较小的，那么可以使用广播变量的方式，将小RDD发送到各个worker的executor中，实现本地的连接  
代码实现：

object \_02SparkRDDBroadcastOps {

def main(args: Array[String]): Unit = {

Logger.getLogger("org.apache.hadoop").setLevel(Level.WARN)

Logger.getLogger("org.apache.spark").setLevel(Level.WARN)

Logger.getLogger("org.project-spark").setLevel(Level.WARN)

val conf = **new** SparkConf()

.setMaster("local[2]")

.setAppName(s"${\_02SparkRDDBroadcastOps.getClass.getSimpleName}")

val sc = **new** SparkContext(conf)

val stu = List(

"1 郑祥楷 1",

"2 王佳豪 1",

"3 刘鹰 2",

"4 宋志华 3",

"5 刘帆 4",

"6 OLDLi 5"

)

val cls = List(

"1 1807bd-bj",

"2 1807bd-sz",

"3 1807bd-wh",

"4 1807bd-xa",

"7 1805bd-bj"

)

*/\**

*使用广播变量来完成上述操作*

*一般用户表都比较大，而班级表相对很小，符合我们在共享变量中提出的第一个假设*

*所以我们可以尝试使用广播变量来进行解决*

*\*/*

val stuRDD = sc.parallelize(stu)

*//cls-->map---->*

val map = cls.map{**case** line => {

(line.substring(0, line.indexOf(" ")), line.substring(line.indexOf(" ")).trim)

}}.toMap

*//map--->broadcast*

val clsMapBC:Broadcast[Map[String, String]] = sc.broadcast(map)

stuRDD.map{**case** line => {

val map = clsMapBC.value

val fields = line.split("\\s+")

val cid = fields(2)

*// map.get(cid)*

val className = map.getOrElse(cid, "UnKnown")

s"${fields(0)}\t${fields(1)}\t${className}"*//在mr中学习到的map join*

}}.foreach(println)

sc.stop()

}

#### 5）采样倾斜的key并拆分join操作（大大表）：

  适用场景：在hive两张表进行join的时候，如果两张表的数据都很大，并且，一张表的数据很均匀，但是另一张表的数据有少量的key数据量过大，此时使用这个解决方案  
  实现原理：对于join导致的数据倾斜，如果只是某几个key导致了倾斜，可以将少数几个key分拆成独立RDD，并附加随机前缀打散成n份去进行join，此时这几个key对应的数据就不会集中在少数几个task上，而是分散到多个task进行join了。  
代码实现：

一张表中：

Id num

1 100W

2 10

3 10

4 10

5 10

可以使用union的方式来启动多个job并行执行：

//通过分离数据量大key来解决数据倾斜

**select** **count**(\*) **from** t\_test **where** **id** !=1 **group** **by** **id**

**union**

**select** **count**(\*) **from** t\_test **where** **id** ==1 **group** **by** **id**

#### 6）使用随机前缀和扩容RDD进行join（大量key的数据倾斜）：

  适用场景：如果进行join操作时，RDD中有大量的key导致数据倾斜，那么进行拆分可以也没有意义，此时使用这种方法  
  
  实现原理：这一种方案是针对有大量倾斜key的情况，没法将部分key拆分出来进行单独处理，因此只能对整个RDD进行数据扩容，对内存资源要求很高。  
代码实现：  
  
左表的连接条件的值，可以在某个范围内进行随机，并且这个随机值有多少个，那么右表的数据就要复制多少份。

object \_03SparkJoinDataSkewOps {

def main(args: Array[String]): Unit = {

Logger.getLogger("org.apache.hadoop").setLevel(Level.WARN)

Logger.getLogger("org.apache.spark").setLevel(Level.WARN)

Logger.getLogger("org.project-spark").setLevel(Level.WARN)

val sc = SparkUtil.sparkContext("local[2]", "\_03SparkJoinDataSkewOps")

val list1 = List(

"hello 1",

"hello 2",

"hello 3",

"hello 4",

"you 1",

"me 2"

)

val list2 = List(

"hello zhangsan dfsadfasdfsa",

"hello lisi adfasdfasd",

"you wangwu adfasdfs",

"me zhouqi adfadfa"

)

*//<key, value>*

val listRDD1 = sc.parallelize(list1).map(line => {

val fields = line.split("\\s+")

(fields(0), fields(1))

})

*//<key, value>*

val listRDD2 = sc.parallelize(list2).map(line => {

val fields = line.split("\\s+")

(fields(0), fields(1))

})

val joinRDD: RDD[(String, (String, String))] = dataSkewRDDJoin(sc, listRDD1, listRDD2)

println("最后进行join的结果：")

joinRDD.foreach(println)

sc.stop()

}

private def dataSkewRDDJoin(sc: SparkContext, listRDD1: RDD[(String, String)], listRDD2: RDD[(String, String)]) = {

*//假设listRDD1中的部分key有数据倾斜，所以我在进行join操作的时候，需要进行拆分计算*

*//step 1 找到发生数据倾斜的key*

val dataSkewKeys = listRDD1.sample(false, 0.6).countByKey().toList.sortWith((t1, t2) => t1.\_2 > t2.\_2).take(1).map(t => t.\_1)

println("通过sample算子得到的可能发生数据倾斜的key：" + dataSkewKeys)

*//step 2 对listRDD1和listRDD2中的数据按照dataSkewKeys各拆分成两个部分*

*//step 2.1 讲dataSkewKeys进行广播*

val dskBC = sc.broadcast(dataSkewKeys)

*// step 2.2 进行拆分*

val dataSkewRDD1 = listRDD1.filter { **case** (word, value) => {

*//有数据倾斜的rdd--->dataskewRDD1*

val dsks = dskBC.value

dsks.contains(word)

}

}

val commonRDD1 = listRDD1.filter { **case** (word, value) => {

*//没有数据倾斜的rdd--->commonRDD1*

val dsks = dskBC.value

!dsks.contains(word)

}

}

val dataSkewRDD2 = listRDD2.filter { **case** (word, value) => {

*//有数据倾斜的rdd--->dataskewRDD1*

val dsks = dskBC.value

dsks.contains(word)

}

}

val commonRDD2 = listRDD2.filter { **case** (word, value) => {

*//没有数据倾斜的rdd--->commonRDD1*

val dsks = dskBC.value

!dsks.contains(word)

}

}

}

*//step 3 对dataskewRDD进行添加N以内随机前缀*

*// step 3.1 添加随机前缀*

val prefixDSRDD1:RDD[(String, String)] = dataSkewRDD1.map { case (word, value) => {

val random = **new** Random()

val prefix = random.nextInt(2)

(s"${prefix}\_${word}", value)

}

}

*// step 3.2 另一个rdd进行扩容*

val prefixDSRDD2:RDD[(String, String)] = dataSkewRDD2.flatMap { case (word, value) => {

val ab = ArrayBuffer[(String, String)]()

**for** (i <- 0 until 2) {

ab.append((s"${i}\_${word}", value))

}

ab

}

}

println("---->有数据倾斜RDD1添加前缀成prefixDSRDD1的结果：" + prefixDSRDD1.collect().mkString(","))

println("---->有数据倾斜RDD2扩容之后成prefixDSRDD2的结果：" + prefixDSRDD2.collect().mkString(","))

*// step 4 分步进行join操作*

*// step 4.1 有数据倾斜的prefixDSRDD1和prefixDSRDD2进行join*

val prefixJoinDSRDD = prefixDSRDD1.join(prefixDSRDD2)

*//ste 4.2 无数据倾斜的commonRDD1和commonRDD2进行join*

val commonJoinRDD = commonRDD1.join(commonRDD2)

*// step 4.3 将随机前缀去除*

val dsJionRDD = prefixJoinDSRDD.map { **case** (word, (value1, value2)) => {

(word.substring(2), (value1, value2))

}

}

*//step 5 将拆分进行join之后的结果进行union连接，得到最后的结果 sql union all*

val joinRDD = dsJionRDD.union(commonJoinRDD)

joinRDD

}

}