

目錄

Introduction	1.1
spark参数介绍	1.2
spark性能调优(官方)	1.3
spark性能调优	1.4
databricks spark知识库	1.5
Cigna优化Spark Streaming实时处理应用	1.6
Spark性能优化指南——基础篇	1.7
Spark性能优化指南——高级篇	1.8

目录

- [spark参数介绍](#)
- [spark性能调优\(官方\)](#)
- [spark性能调优](#)
- [databricks spark知识库](#)
- [Cigna优化Spark Streaming实时处理应用](#)
- [Spark性能优化指南——基础篇](#)
- [Spark性能优化指南——高级篇](#)

spark参数介绍

1 spark on yarn常用属性介绍

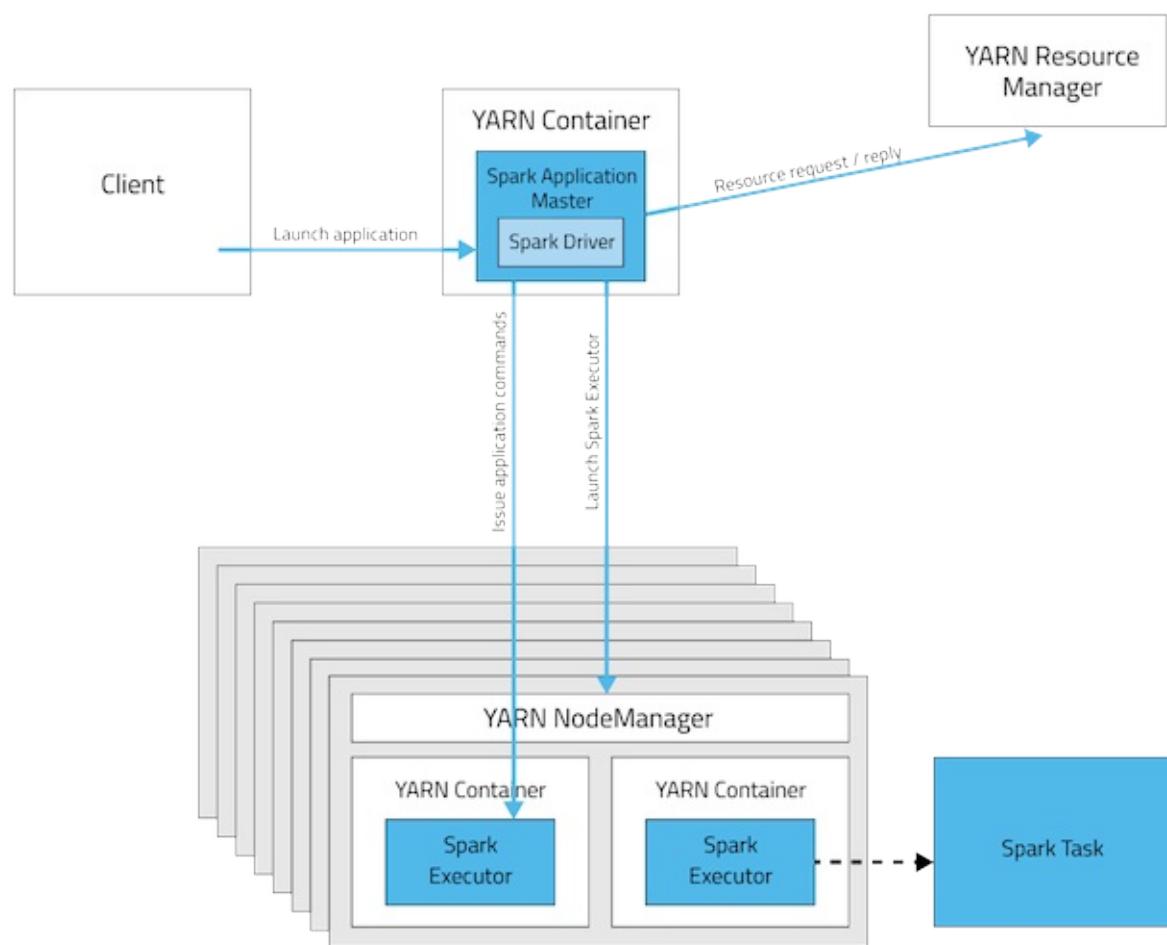
属性名	默认值	属性说明
spark.yarn.am.memory	512m	在客户端模式（client mode）下，yarn 应用 master 使用群模式（cluster mode）。用 spark.driver.memory 代替。
spark.driver.cores	1	在集群模式（cluster mode）下，driver 程序使用的核（cluster mode）下，driver 和 master 运行在同一个 job 中。以 master 控制这个核数。（client mode）下，使用 spark.yarn.am.cores 控制核。
spark.yarn.am.cores	1	在客户端模式（client mode）下，yarn 应用的 master 在群模式下，使用 spark.driver.cores 控制核。
spark.yarn.am.waitTime	100ms	在集群模式（cluster mode）下，用 master 等待 SparkContext 间。在客户端模式（client mode）下，master 等待 driver 端。
spark.yarn.submit.file.replication	3	文件上传到 hdfs 上去的 replication 次数。
spark.yarn.preserve.staging.files	false	设置为 true 时，在 job 结束后保留 staged 文件；否则删除。
spark.yarn.scheduler.heartbeat.interval-ms	3000	Spark 应用 master 与 yarn resourcemanager 之间的心跳间隔时间。
spark.yarn.scheduler.initial-allocation.interval	200ms	当存在挂起的容器分配请求时，用 master 发送心跳给 resourcemanager。它的大小不能大于 spark.yarn.scheduler.heartbeat.interval-ms，如果挂起的请求还存在，则加倍，直到到达 spark.yarn.scheduler.heartbeat.interval-ms 大小。
spark.yarn.max.executor.failures	numExecutors * 2，并且不小于3	在失败应用程序之前，executor 失败次数。
		Executors 的个数。这个配置项只对 yarn 容器有效。

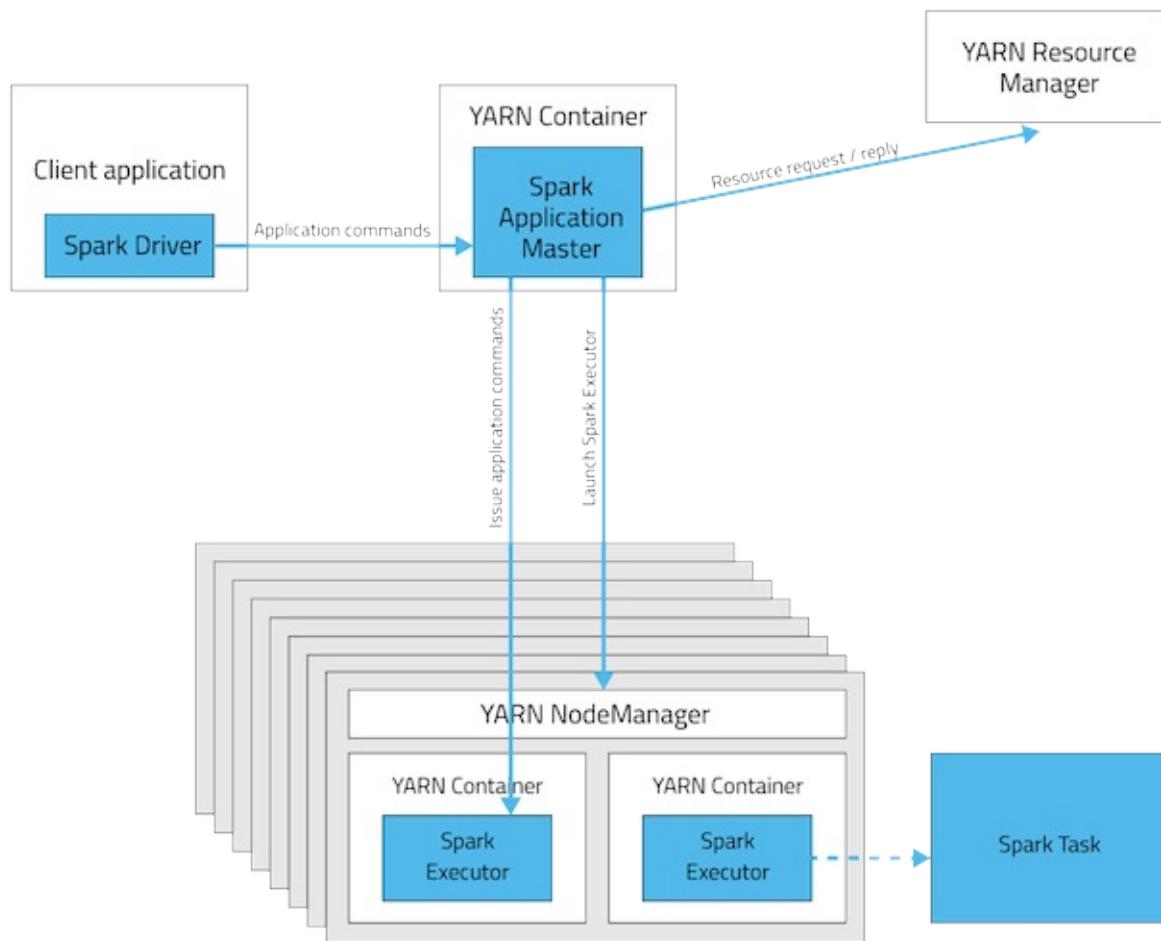
spark.executor.instances	2	和 spark.dynamicAllocation 容。当同时配置这两个配置时，spark.executor.instances 会忽略。
spark.yarn.executor.memoryOverhead	executorMemory * 0.10，并且不小于 384m	每个 executor 分配的堆外内存。
spark.yarn.driver.memoryOverhead	driverMemory * 0.10，并且不小于 384m	在集群模式下，每个 driver 存。
spark.yarn.am.memoryOverhead	AM memory * 0.10，并且不小于 384m	在客户端模式下，每个 driver 内存。
spark.yarn.am.port	随机	Yarn 应用 master 监听的端口。
spark.yarn.queue	default	应用提交的 yarn 队列的名称。
spark.yarn.jar	none	Jar 文件存放的地方。默认 jar 安装在本地，但是 jar 在 hdfs 上，其他机器也可访问。

2 客户端模式和集群模式的区别

这里我们要区分一下什么是客户端模式（client mode），什么是集群模式（cluster mode）。

我们知道，当在 YARN 上运行 Spark 作业时，每个 Spark executor 作为一个 YARN 容器（container）运行。Spark 可以使得多个 Tasks 在同一个容器（container）里面运行。yarn-cluster 和 yarn-client 模式的区别其实就是 Application Master 进程的区别，在 yarn-cluster 模式下，driver 运行在 AM（Application Master）中，它负责向 YARN 申请资源，并监督作业的运行状况。当用户提交了作业之后，就可以关掉 Client，作业会继续在 YARN 上运行。然而 yarn-cluster 模式不适合运行交互类型的作业。在 yarn-client 模式下，Application Master 仅仅向 YARN 请求 executor，client 会和请求的 container 通信来调度他们工作，也就是说 Client 不能离开。下面的图形象表示了两者的区别。





2.1 Spark on YARN集群模式分析

2.1.1 客户端操作

- 1、根据 `yarnConf` 来初始化 `yarnClient`，并启动 `yarnClient`；
- 2、创建客户端 `Application`，并获取 `Application` 的 `ID`，进一步判断集群中的资源是否满足 `executor` 和 `ApplicationMaster` 申请的资源，如果不满足则抛出 `IllegalArgumentException`；
- 3、设置资源、环境变量：其中包括了设置 `Application` 的 `Staging` 目录、准备本地资源（`jar` 文件、`log4j.properties`）、设置 `Application` 其中的环境变量、创建 `Container` 启动的 `Context` 等；
- 4、设置 `Application` 提交的 `Context`，包括设置应用的名字、队列、`AM` 的申请的 `Container`、标记该作业的类型为 `Spark`；
- 5、申请 `Memory`，并最终通过 `yarnClient.submitApplication` 向 `ResourceManager` 提交该 `Application`。

当作业提交到 YARN 上之后，客户端就没事了，甚至在终端关掉那个进程也没事，因为整个作业运行在 YARN 集群上进行，运行的结果将会保存到 HDFS 或者日志中。

2.1.2 提交到YARN集群，YARN操作

- 1、运行 ApplicationMaster 的 run 方法；
- 2、设置好相关的环境变量。
- 3、创建 amClient，并启动；
- 4、在 Spark UI 启动之前设置 Spark UI 的 AmIpFilter；
- 5、在 startUserClass 函数专门启动了一个线程（名称为 Driver 的线程）来启动用户提交的 Application，也就是启动了 Driver。在 Driver 中将会初始化 SparkContext；
- 6、等待 SparkContext 初始化完成，最多等待 spark.yarn.applicationMaster.waitTries 次数（默认为 10），如果等待了的次数超过了配置的，程序将会退出；否则用 SparkContext 初始化 yarnAllocator；
- 7、当 SparkContext、Driver 初始化完成的时候，通过 amClient 向 ResourceManager 注册 ApplicationMaster；
- 8、分配并启动 Executors。在启动 Executors 之前，先要通过 yarnAllocator 获取到 numExecutors 个 Container，然后在 Container 中启动 Executors。如果在启动 Executors 的过程中失败的次数达到了 maxNumExecutorFailures 的次数，maxNumExecutorFailures 的计算规则如下：

```
// Default to numExecutors * 2, with minimum of 3
private val maxNumExecutorFailures = sparkConf.getInt("spark.yarn.max.executor.failures",
  sparkConf.getInt("spark.yarn.max.worker.failures", math.max(args.numExecutors * 2,
  3)))
```

那么这个 Application 将失败，将 Application Status 标明为 FAILED，并将关闭 SparkContext。其实，启动 Executors 是通过 ExecutorRunnable 实现的，而 ExecutorRunnable 内部是启动 CoarseGrainedExecutorBackend 的。

- 9、最后，Task 将在 CoarseGrainedExecutorBackend 里面运行，然后运行状况会通过 Akka 通知 CoarseGrainedScheduler，直到作业运行完成。

2.2 Spark on YARN客户端模式分析

和 `yarn-cluster` 模式一样，整个程序也是通过 `spark-submit` 脚本提交的。但是 `yarn-client` 作业程序的运行不需要通过 `client` 类来封装启动，而是直接通过反射机制调用作业的 `main` 函数。下面是流程。

- 1、通过 `SparkSubmit` 类的 `launch` 的函数直接调用作业的 `main` 函数（通过反射机制实现），如果是集群模式就会调用 `Client` 的 `main` 函数。
- 2、而应用程序的 `main` 函数一定都有个 `SparkContent`，并对其进行初始化；
- 3、在 `SparkContent` 初始化中将会依次做如下的事情：设置相关的配置、注册 `MapOutputTracker`、`BlockManagerMaster`、`BlockManager`，创建 `taskScheduler` 和 `dagScheduler`；
- 4、初始化完 `taskScheduler` 后，将创建 `dagScheduler`，然后通过 `taskScheduler.start()` 启动 `taskScheduler`，而在 `taskScheduler` 启动的过程中也会调用 `SchedulerBackend` 的 `start` 方法。在 `SchedulerBackend` 启动的过程中将会初始化一些参数，封装在 `ClientArguments` 中，并将封装好的 `ClientArguments` 传进 `Client` 类中，并 `client.runApp()` 方法获取 `Application ID`。
- 5、`client.runApp` 里面的做的和上章客户端进行操作那节类似，不同的是在里面启动是 `ExecutorLauncher`（`yarn-cluster` 模式启动的是 `ApplicationMaster`）。
- 6、在 `ExecutorLauncher` 里面会初始化并启动 `amClient`，然后向 `ApplicationMaster` 注册该 `Application`。注册完之后将会等待 `driver` 的启动，当 `driver` 启动完之后，会创建一个 `MonitorActor` 对象用于和 `CoarseGrainedSchedulerBackend` 进行通信（只有事件 `AddWebUIFilter` 他们之间才通信，`Task` 的运行状况不是通过它和 `CoarseGrainedSchedulerBackend` 通信的）。然后就是设置 `addAmIpFilter`，当作业完成的时候，`ExecutorLauncher` 将通过 `amClient` 设置 `Application` 的状态为 `FinalApplicationStatus.SUCCEEDED`。
- 7、分配 `Executors`，这里面的分配逻辑和 `yarn-cluster` 里面类似。
- 8、最后，`Task` 将在 `CoarseGrainedExecutorBackend` 里面运行，然后运行状况会通过 `Akka` 通知 `coarseGrainedScheduler`，直到作业运行完成。
- 9、在作业运行的时候，`YarnClientSchedulerBackend` 会每隔1秒通过 `client` 获取到作业的运行状况，并打印出相应的运行信息，当 `Application` 的状态是 `FINISHED`、`FAILED` 和 `KILLED` 中的一种，那么程序将退出等待。
- 10、最后有个线程会再次确认 `Application` 的状态，当 `Application` 的状态是 `FINISHED`、`FAILED` 和 `KILLED` 中的一种，程序就运行完成，并停止 `SparkContext`。整个过程就结束了。

3 spark submit 和 spark shell参数介绍

参数名	格式	参数说明
--master	MASTER_URL	如spark://host:port
--deploy-mode	DEPLOY_MODE	Client或者master，默认是client
--class	CLASS_NAME	应用程序的主类
--name	NAME	应用程序的名称
--jars	JARS	逗号分隔的本地jar包，包含在driver和executor的classpath下
--packages		包含在driver和executor的classpath下的jar包逗号分隔的"groupId:artifactId : version"列表
--exclude-packages		用逗号分隔的"groupId:artifactId"列表
--repositories		逗号分隔的远程仓库
--py-files	PY_FILES	逗号分隔的".zip",".egg"或者".py"文件，这些文件放在python app的PYTHONPATH下面
--files	FILES	逗号分隔的文件，这些文件放在每个executor的工作目录下面
--conf	PROP=VALUE	固定的spark配置属性
--properties-file	FILE	加载额外属性的文件
--driver-memory	MEM	Driver内存，默认1G
--driver-java-options		传给driver的额外的Java选项
--driver-library-path		传给driver的额外的库路径
--driver-class-path		传给driver的额外的类路径
--executor-memory	MEM	每个executor的内存，默认是1G
--proxy-user	NAME	模拟提交应用程序的用户
--driver-cores	NUM	Driver的核数，默认是1。这个参数仅仅在 standalone 集群 deploy 模式下使用
--supervise		Driver失败时，重启driver。在 mesos 或者 standalone 下使用
--verbose		打印debug信息
--total-		

executor-cores	NUM	所有executor总共的核数。仅仅在 mesos 或者 standalone 下使用
--executor-core	NUM	每个executor的核数。在 yarn 或者 standalone 下使用
--driver-cores	NUM	Driver的核数，默认是1。在 yarn 集群模式下使用
--queue	QUEUE_NAME	队列名称。在 yarn 下使用
--num-executors	NUM	启动的executor数量。默认为2。在 yarn 下使用

你可以通过 `spark-submit --help` 或者 `spark-shell --help` 来查看这些参数。

参考文献

- 【1】[Spark:Yarn-cluster和Yarn-client区别与联系](#)
- 【2】[Spark on YARN客户端模式作业运行全过程分析](#)
- 【3】[Spark on YARN集群模式作业运行全过程分析](#)

Spark 调优

由于大部分 Spark 计算都是在内存中完成的，所以 Spark 程序的瓶颈可能由集群中任意一种资源导致，如：CPU、网络带宽、或者内存等。最常见的情况是，数据能装进内存，而瓶颈是网络带宽；当然，有时候我们也需要做一些优化调整来减少内存占用，例如将 RDD 以序列化格式保存。本文将主要涵盖两个主题：1. 数据序列化（这对于优化网络性能极为重要）；2. 减少内存占用以及内存调优。同时，我们也会提及其他几个比较小的主题。

1 数据序列化

序列化在任何一种分布式应用性能优化时都扮演几位重要的角色。如果序列化格式序列化过程缓慢，或者需要占用字节很多，都会大大拖慢整体的计算效率。通常，序列化都是 Spark 应用优化时首先需要关注的地方。Spark 着眼于便利性（允许你在计算过程中使用任何 Java 类型）和性能的一个平衡。Spark 主要提供了两个序列化库：

- **Java serialization**：默认情况，Spark 使用 Java 自带的 `ObjectOutputStream` 框架来序列化对象，这样任何实现了 `java.io.Serializable` 接口的对象，都能被序列化。同时，你还可以通过扩展 `java.io.Externalizable` 来控制序列化性能。Java 序列化很灵活但性能较差，同时序列化后占用的字节数也较多。
- **Kryo serialization**：Spark 还可以使用 Kryo 库（版本2）提供更高效的序列化格式。Kryo 的序列化速度和字节占用都比 Java 序列化好很多（通常是10倍左右），但 Kryo 不支持所有实现了 `Serializable` 接口的类型，它需要你在程序中 `register` 需要序列化的类型，以得到最佳性能。

要切换使用 Kryo，你可以在 `SparkConf` 初始化的时候调用 `conf.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer")`。这个设置不仅控制各个 worker 节点之间的混洗数据序列化格式，同时还控制 RDD 存到磁盘上的序列化格式。目前，Kryo 不是默认的序列化格式，因为它需要你在使用前注册需要序列化的类型，不过我们还是建议在对网络敏感的应用场景下使用 Kryo。

Spark 对一些常用的 Scala 核心类型，如在 Twitter chill 库的 `AllScalaRegistrar` 中，自动使用 Kryo 序列化格式。

如果你的自定义类型需要使用 Kryo 序列化，可以用 `registerKryoClasses` 方法先注册：

```
val conf = new SparkConf().setMaster(...).setAppName(...)
conf.registerKryoClasses(Array(classOf[MyClass1], classOf[MyClass2]))
val sc = new SparkContext(conf)
```

Kryo 的文档中有详细描述了更多的高级选项，如：自定义序列化代码等。

如果你的对象很大，你可能需要增大 `spark.kryoserializer.buffer` 配置项。其值至少需要大于最大对象的序列化长度。

最后，如果你不注册需要序列化的自定义类型，Kryo 也能工作，不过每一个对象实例的序列化结果都会包含一份完整的类名，这有点浪费空间。

2 内存调优

内存占用调优主要需要考虑3点：数据占用的总内存（你会希望整个数据集都能装进内存）；访问数据集中每个对象的开销；垃圾回收的开销（如果你的数据集中对象周转速度很快的话）。

一般情况下，Java 对象的访问时很快的，但同时 Java 对象会比原始数据（仅包含各个字段值）占用的空间多 2~5 倍。主要原因有：

- 每个 Java 对象都有一个对象头（`object header`），对象头大约占用 16 字节，其中包含像其对应 `class` 的指针这样的信息。对于一些包含较少数据的对象（比如只包含一个 `Int` 字段），这个对象头可能比对象数据本身还大。
- Java 字符串（`String`）有大约 40 字节额外开销（Java String 以 `char` 数据的形式保存原始数据，所以需要一些额外的字段，如数组长度等），并且每个字符都以两字节的 `UTF-16` 编码在内部保存。因此，10 个字符的 `String` 很容易就占了 60 字节。
- 一些常见的集合类，如 `HashMap`、`LinkedList`，使用的是链表类数据结构，因此它们对每项数据都有一个包装器。这些包装器对象不仅自身就有“对象头”，同时还有指向下一个包装器对象的链表指针（通常为 8 字节）。
- 原始类型的集合通常也是以“装箱”的形式包装成对象（如：`java.lang.Integer`）。

本节只是 Spark 内存管理的一个概要，下面我们会更详细地讨论各种 Spark 内存调优的具体策略。特别地，我们会讨论如何评估数据的内存使用量，以及如何改进 – 要么改变你的数据结构，要么以某种序列化格式存储数据。最后，我们还会讨论如何调整 Spark 的缓存大小，以及如何调优 Java 的垃圾回收器。

2.1 内存管理概览

Spark 中内存主要用于两类目的：执行计算和数据存储。执行计算的内存主要用于 `shuffle`、关联（`join`）、排序（`sort`）以及聚合（`aggregation`），而数据存储的内存主要用于缓存和集群内部数据传播。Spark 中执行计算和数据存储都是共享同一个内存区域（`M`）。如果执行计算没有占用内存，那么数据存储可以申请占用所有可用的内存，反之亦然。执行计算可能会抢占数据存储使用的内存，并将存储于内存的数据逐出内存，直到数

据存储占用的内存比例降低到一个指定的比例 (R)。换句话说， R 是 M 基础上的一个子区域，这个区域的内存数据永远不会被逐出内存。然而，数据存储不会抢占执行计算的内存。

这样设计主要有这么几个需要考虑的点。首先，不需要缓存数据的应用可以把整个空间用来执行计算，从而避免频繁地把数据吐到磁盘上。其次，需要缓存数据的应用能够有一个数据存储比例 (R) 的最低保证，也避免这部分缓存数据被全部逐出内存。最后，这个实现方式能够在默认情况下，为大多数使用场景提供合理的性能，而不需要专家级用户来设置内存使用如何划分。

虽然有两个内存划分相关的配置参数，但一般来说，用户不需要设置，因为默认值已经能够适用于绝大部分的使用场景：

- `spark.memory.fraction` : 表示上面 M 的大小，其值为相对于 `JVM` 堆内存的比例（默认 `0.75`）。剩余的 `25%` 是为其他用户的数据结构、`Spark` 内部元数据以及避免 `OOM` 错误的安全预留空间。
- `spark.memory.storageFraction` : 表示上面 R 的大小，其值为相对于 M 的一个比例（默认 `0.5`）。 R 是 M 中专门用于缓存数据块的部分，这部分数据块永远不会因执行计算任务而逐出内存。

2.2 评估内存消耗

确定一个数据集占用内存总量最好的办法就是，创建一个 `RDD`，并缓存到内存中，然后再回到 `web UI` 上“Storage”页面查看。页面上会展示这个 `RDD` 总共占用了多少内存。

要评估一个特定对象的内存占用量，可以用 `SizeEstimator.estimate` 方法。这个方法对试验哪种数据结构能够裁剪内存占用量比较有用，同时，也可以帮助用户了解广播变量在每个执行器堆上占用的内存量。

2.3 数据结构调优

减少内存消耗的首要方法就是避免过多的 `Java` 封装（减少对象头和额外辅助字段），比如基于指针的数据结构和包装对象等。以下有几条建议：

- 设计数据结构的时候，优先使用对象数组和原生类型，减少对复杂集合类型（如：`HashMap`）的使用。`fastutil` 提供了一些很方便的原生类型集合，同时兼容 `Java` 标准库。
- 尽可能避免嵌套大量的小对象和指针。
- 对应键值应尽量使用数值型或枚举型，而不是字符串型。
- 如果内存小于 `32GB`，可以设置 `JVM` 标志参数 `-XX:+UseCompressedOops` 将指针设为4字节而不是8字节。你可以在 `spark-env.sh` 中设置这个参数。

2.4 序列化RDD存储

如果经过上面的调整后，存储的数据对象还是太大，那么你可以试试将这些对象以序列化格式存储，所需要做的只是通过 `RDD persistence API` 设置好存储级别，如：`MEMORY_ONLY_SER`。Spark 会将 `RDD` 的每个分区以一个巨大的字节数组形式存储起来。以序列化格式存储的唯一缺点就是访问数据会变慢一点，因为 Spark 需要反序列化每个被访问的对象。如果你需要序列化缓存数据，我们强烈建议你使用 `Kryo`，和 `Java` 序列化相比，`Kryo` 能大大减少序列化对象占用的空间（当然也比原始 `Java` 对象小很多）。

2.5 垃圾回收调优

`JVM` 的垃圾回收在某些情况下可能会造成瓶颈，比如，你的 `RDD` 存储经常需要“换入换出”（新 `RDD` 抢占了老 `RDD` 内存，不过如果你的程序没有这种情况的话那 `JVM` 垃圾回收一般不是问题，比如，你的 `RDD` 只是载入一次，后续只是在这个 `RDD` 上做操作）。当 `Java` 需要把老对象逐出内存的时候，`JVM` 需要跟踪所有的 `Java` 对象，并找出哪些对象已经没有用了。概括起来就是，垃圾回收的开销和对象个数成正比，所以减少对象的个数（比如用 `Int` 数组取代 `LinkedList`），就能大大减少垃圾回收的开销。当然，一个更好的方法就如前面所说的，以序列化形式存储数据，这时每个 `RDD` 分区都只包含有一个对象了（一个巨大的字节数组）。在尝试其他技术方案前，首先可以试试用序列化 `RDD` 的方式（`serialized caching`）评估一下 `GC` 是不是一个瓶颈。

如果你的作业中各个任务需要的工作内存和节点上存储的 `RDD` 缓存占用的内存产生冲突，那么 `GC` 很可能会出现问题。下面我们将讨论一下如何控制好 `RDD` 缓存使用的内存空间，以减少这种冲突。

衡量GC的影响

`GC` 调优的第一步是统计一下，垃圾回收启动的频率以及 `GC` 所使用的总时间。给 `JVM` 设置一下这几个参数（参考 `Spark` 配置指南，查看 `Spark` 作业中的 `Java` 选项参数）：- `verbose:gc -XX:+PrintGCDetails`，就可以在后续 `Spark` 作业的 `worker` 日志中看到每次 `GC` 花费的时间。注意，这些日志是在集群 `worker` 节点上（在各节点的工作目录下 `stdout` 文件中），而不是你的驱动器所在节点。

高级GC调优

为了进一步调优 `GC`，我们就需要对 `JVM` 内存管理有一个基本的了解：

- `Java` 堆内存可分配的空间有两个区域：新生代（`Young generation`）和老年代（`old generation`）。新生代用以保存生存周期短的对象，而老年代则是保存生存周期长的对象。
- 新生代区域被进一步划分为三个子区域：`Eden`，`Survivor1`，`Survivor2`。

- 简要描述一下垃圾回收的过程：如果 `Eden` 区满了，则启动一轮 `minor GC` 回收 `Eden` 中的对象，生存下来（没有被回收掉）的 `Eden` 中的对象和 `survivor1` 区中的对象一并复制到 `survivor2` 中。两个 `survivor` 区域是互相切换使用的（就是说，下次从 `Eden` 和 `survivor2` 中复制到 `survivor1` 中）。如果某个对象的年龄（每次 `GC` 所有生存下来的对象长一岁）超过某个阈值，或者 `survivor2`（下次是 `survivor1`）区域满了，则将对象移到老年代（`old` 区）。最终如果老年代也满了，就会启动 `full GC`。

`Spark GC` 调优的目标就是确保老年代（`old generation`）只保存长生命周期 `RDD`，而同时新生代（`young generation`）的空间又能足够保存短生命周期的对象。这样就能在任务执行期间，避免启动 `full GC`。以下是 `GC` 调优的主要步骤：

- 从 `GC` 的统计日志中观察 `GC` 是否启动太多。如果某个任务结束前，多次启动了 `full GC`，则意味着用以执行该任务的内存不够。
- 如果 `GC` 统计信息中显示，老年代内存空间已经接近存满，可以通过降低 `spark.memory.storageFraction` 来减少 `RDD` 缓存占用的内存；减少缓存对象总比任务执行缓慢要强！
- 如果 `major GC` 比较少，但 `minor GC` 很多的话，可以多分配一些 `Eden` 内存。你可以把 `Eden` 的大小设为高于各个任务执行所需的工作内存。如果要把 `Eden` 大小设为 `E`，则可以这样设置新生代区域大小：`-Xmn=4/3*E`。（放大 $4/3$ 倍，主要是为了给 `survivor` 区域保留空间）
- 举例来说，如果你的任务会从 `HDFS` 上读取数据，那么单个任务的内存需求可以用其所读取的 `HDFS` 数据块的大小来评估。需要特别注意的是，解压后的 `HDFS` 块是解压前的 $2\sim3$ 倍。所以如果我们希望保留 $3\sim4$ 个任务并行的工作内存，并且 `HDFS` 块大小为 `64MB`，那么可以评估 `Eden` 的大小应该设为 `4*3*64MB`。
- 最后，再观察一下垃圾回收的启动频率和总耗时有没有什么变化。

我们的很多经验表明，`GC` 调优的效果和你的程序代码以及可用的总内存相关。网上还有不少调优的选择，但总体来说，就是控制好 `full GC` 的启动频率，就能有效减少垃圾回收开销。

3 其他事项

3.1 并行度

一般来说集群并不会满负荷运转，除非你把每个操作的并行度都设得足够大。`Spark` 会自动根据对应的输入文件大小来设置“`map`”类算子的并行度（当然你可以通过一个 `SparkContext.textFile` 等函数的可选参数来控制并行度），而对于想 `groupByKey` 或 `reduceByKey` 这类“`reduce`”算子，会使用其各父 `RDD` 分区数的最大值。你可以将并行度作

为构建 `RDD` 第二个参数（参考 `spark.PairRDDFunctions`），或者设置 `spark.default.parallelism` 这个默认值。一般来说，评估并行度的时候，我们建议 2~3 个任务共享一个 `CPU`。

3.2 Reduce任务的内存占用

如果 `RDD` 比内存要大，有时候你可能收到一个 `OutOfMemoryError` 错误，其实这是因为你的任务集中的某个任务太大了，如 `reduce` 任务 `groupByKey`。Spark 的 `Shuffle` 算子（`sortByKey`, `groupByKey`, `reduceByKey`, `join` 等）会在每个任务中构建一个哈希表，以便在任务中对数据分组，这个哈希表有时会很大。最简单的修复办法就是增大并行度，以减小单个任务的输入集。Spark 对于 200ms 以内的短任务支持非常好，因为 Spark 可以跨任务复用执行器 `JVM`，任务的启动开销很小，因此把并行度增加到比集群中总 `CPU` 核数没有任何问题。

3.3 广播大变量

使用 `SparkContext` 中的广播变量相关功能（`broadcast functionality`）能大大减少每个任务本身序列化的大小，以及集群中启动作业的开销。如果你的 `Spark` 任务正在使用驱动程序中定义的巨大对象（比如：静态查询表），请考虑使用广播变量替代。Spark 会在 `master` 上将各个任务的序列化后大小打印出来，所以你可以检查一下各个任务是否过大；通常来说，大于 20KB 的任务就值得优化一下。

3.4 数据本地性

数据本地性对 `Spark` 作业往往会有较大的影响。如果代码和其所操作的数据在同一节点上，那么计算速度肯定会更快一些。但如果二者不在一起，那必然需要移动其中之一。一般来说，移动序列化好的代码肯定比挪动一大堆数据要快。Spark 就是基于这个一般性原则来构建数据本地性的调度。

数据本地性是指代码和其所处理的数据的距离。基于数据当前的位置，数据本地性可以划分成以下几个层次（按从近到远排序）：

- `PROCESS_LOCAL` 数据和运行的代码处于同一个 `JVM` 进程内。
- `NODE_LOCAL` 数据和代码处于同一节点。例如，数据处于 `HDFS` 上某个节点，而对应的执行器（`executor`）也在同一个机器节点上。这会比 `PROCESS_LOCAL` 稍微慢一些，因为数据需要跨进程传递。
- `NO_PREF` 数据在任何地方处理都一样，没有本地性偏好。
- `RACK_LOCAL` 数据和代码处于同一个机架上的不同机器。这时，数据和代码处于不同机器上，需要通过网络传递，但还是在同一个机架上，一般也就通过一个交换机传输即可。
- `ANY` 数据在网络中未知，即数据和代码不在同一个机架上。

Spark 倾向于让所有任务都具有最佳的数据本地性，但这并非总是可行的。某些情况下，可能会出现一些空闲的执行器（executor）没有待处理的数据，那么 Spark 可能就会牺牲一些数据本地性。有两种可能的选项：a) 等待已经有任务的 CPU，待其释放后立即在同一台机器上启动一个任务；b) 立即在其他节点上启动新任务，并把所需的数据复制过去。

通常，Spark 会等待一会，看看是否有 CPU 会被释放出来。一旦等待超时，则立即在其他节点上启动并将所需的数据复制过去。数据本地性各个级别之间的回落超时可以单独配置，也可以在统一参数内一起设定；详细请参考 [configuration page](#) 中的 `spark.locality` 相关参数。如果你的任务执行时间比较长并且数据本地性很差，你就应该试试调大这几个参数，不过默认值一般都能适用于大多数场景了。

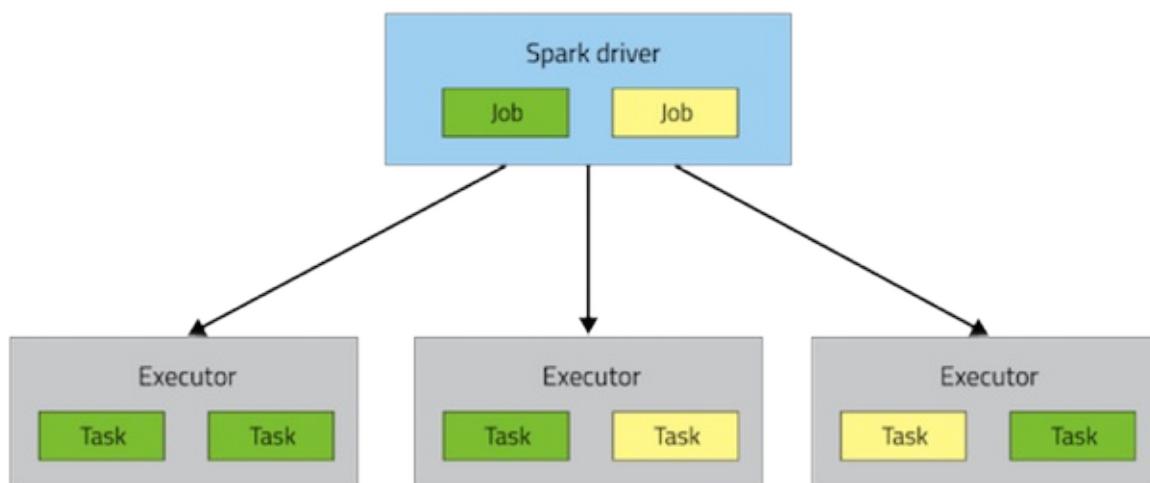
spark性能调优

当你开始编写 Apache Spark 代码或者浏览公开的 API 的时候，你会遇到诸如 transformation, action, RDD 等术语。了解到这些是编写 Spark 代码的基础。同样，当你任务开始失败或者你需要透过 web 界面去了解自己的应用为何如此费时的时候，你需要去了解一些新的名词：job, stage, task。对于这些新术语的理解有助于编写良好 Spark 代码。这里的良好主要指更快的 spark 程序。对于 Spark 底层的执行模型的了解对于写出效率更高的 Spark 程序非常有帮助。

1 Spark 是如何执行程序的

一个 Spark 应用包括一个 driver 进程和若干个分布在集群的各个节点上的 executor 进程。

driver 主要负责调度一些高层次的任务流（flow of work）。executor 负责执行这些任务，这些任务以 task 的形式存在，同时存储用户设置需要 caching 的数据。task 和所有的 executor 的生命周期为程序的整个运行过程（如果使用了 dynamic resource allocation 时可能不是这样的）。一个 executor 可以运行多个任务，任务的运行是并行的。如何调度这些进程是通过集群管理框架完成的（比如 YARN, Mesos, Spark Standalone），任何一个 Spark 程序都会包含一个 driver 和多个 executor 进程。



如上图，在执行层次结构的最上方是一系列 Job。调用一个 Spark 内部的 action 会产生一个 Spark job 来完成它。为了确定这些 job 的实际内容，Spark 检查 RDD 的 DAG 再计算出执行 plan。这个 plan 以最远端的 RDD 为起点（最远端指的是对外没有依赖的 RDD 或者数据已经缓存下来的 RDD），以产生结果 RDD 的 action 为结束。

执行的 plan 由一系列 stage 组成，stage 是 job 的 transformation 的组合，stage 对应于一系列 task，task 指的是对于不同的数据集执行的相同代码。每个 stage 包含不需要 shuffle 所有数据的 transformation 的序列。

什么决定数据是否需要 shuffle 呢？RDD 包含固定数目的 partition，每个 partition 包含若干的 record。对于那些通过 narrow transformation（窄依赖，比如 map 和 filter）返回的 RDD，一个 partition 中的 record 只需要从父 RDD 对应的 partition 中的 record 计算得到。每个对象只依赖于父 RDD 的一个对象。有些操作（比如 coalesce）可能导致一个 task 处理多个输入 partition，但是这种 transformation 仍然被认为是窄的，因为用于计算的多个输入 record 始终是来自有限个数的 partition。

然而 Spark 也支持宽依赖的 transformation，比如 groupByKey，reduceByKey。在这种依赖中，计算得到一个 partition 中的数据需要从父 RDD 中的多个 partition 中读取数据。所有拥有相同 key 的元组最终会被聚合到同一个 partition 中，被同一个 stage 处理。为了完成这种操作，Spark 需要对数据进行 shuffle，意味着数据需要在集群内传递，最终生成由新的 partition 集合组成的新 stage。

在下面的代码中，只有一个 action 以及一系列处理文本的 transformation，这些代码就只有一个 stage，因为没有哪个操作需要从不同的 partition 里面读取数据。

```
sc.textFile("someFile.txt").
  map(mapFunc).
  flatMap(flatMapFunc).
  filter(filterFunc).
  count()
```

跟上面的代码不同，下面一段代码需要统计总共出现超过1000次的字母。

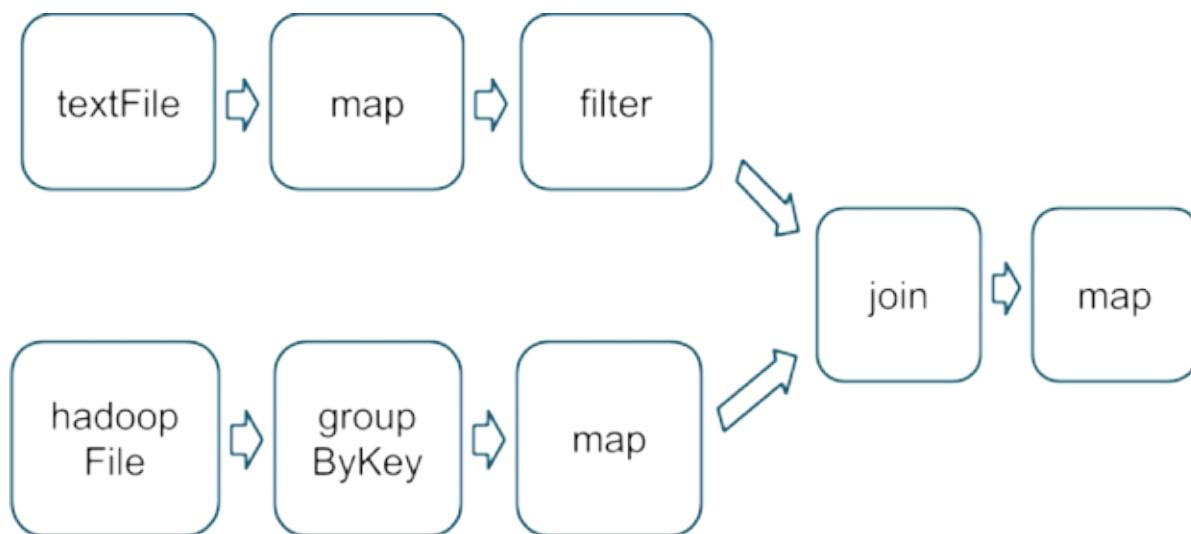
```

val tokenized = sc.textFile(args(0)).flatMap(_.split(' '))
val wordCounts = tokenized.map((_, 1)).reduceByKey(_ + _)
val filtered = wordCounts.filter(_.value >= 1000)
val charCounts = filtered.flatMap(_.value.toCharArray).map((_, 1)).
    reduceByKey(_ + _)
charCounts.collect()

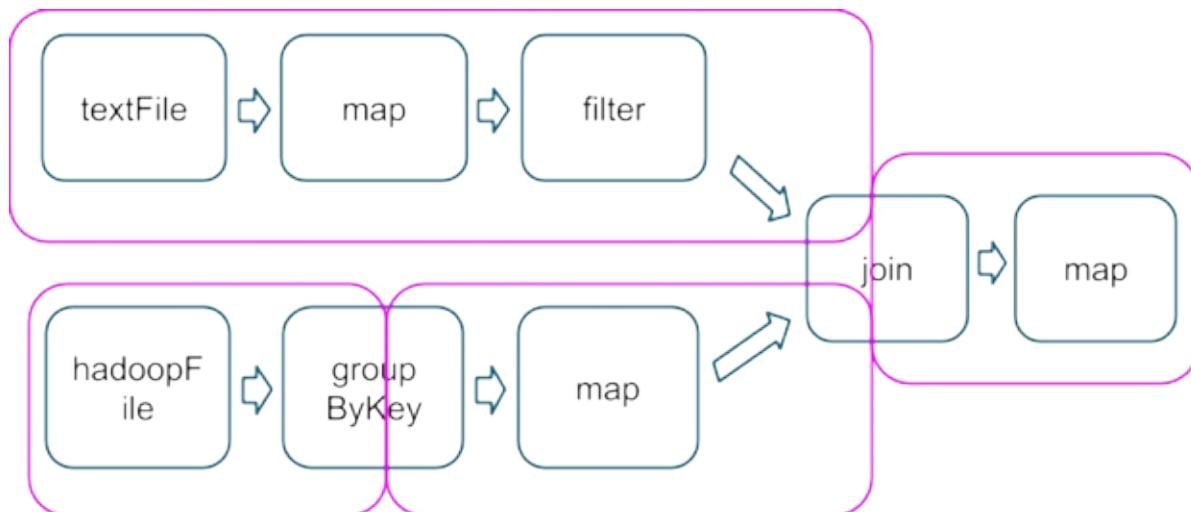
```

这段代码可以分成三个 stage 。 `reduceByKey` 操作是各 stage 之间的分界，因为计算 `reduceByKey` 的输出需要按照可以重新分配 partition 。

这里还有一个更加复杂的 transformation 图，包含一个有多路依赖的 `join` transformation 。



粉红色的框框展示了运行时使用的 stage 图。



运行到每个 `stage` 的边界时，数据在父 `stage` 中通过 `task` 写到磁盘上，而在子 `stage` 中经过网络通过 `task` 去读取数据。这些操作会导致很重的网络以及磁盘的 I/O，所以 `stage` 的边界是非常占资源的，在编写 `Spark` 程序的时候需要尽量避免的。父 `stage` 中 `partition` 个数与子 `stage` 的 `partition` 个数可能不同，所以那些产生 `stage` 边界的 `transformation` 常常需要接受一个 `numPartition` 的参数来决定子 `stage` 中的数据将被切分为多少个 `partition`。

正如在调试 `MapReduce` 是选择 `reducor` 的个数是一项非常重要的参数，调整在 `stage` 边界时的 `partition` 个数经常可以很大程度上影响程序的执行效率。我们会在后面的章节中讨论如何调整这些值。

2 选择正确的 Operator

当需要使用 `Spark` 完成某项功能时，程序员需要从不同的 `action` 和 `transformation` 中选择不同的方案以获得相同的结果。但是不同的方案，最后执行的效率可能有云泥之别。避免常见的陷阱，选择正确的方案可以使得最后的表现有巨大的不同。一些规则和深入的理解可以帮助你做出更好的选择。

选择 `Operator` 方案的主要目标是减少 `shuffle` 的次数以及被 `shuffle` 的文件的大小。因为 `shuffle` 是最耗资源的操作，所有 `shuffle` 的数据都需要写到磁盘并且通过网络传递。`repartition`，`join`，`cogroup`，以及任何 `*By` 或者 `*ByKey` 的 `transformation` 都需要 `shuffle` 数据。这些 `operator` 不是所有都是平等的，但是有些常见的性能陷阱是需要注意的。

- 当进行联合规约操作时，避免使用 `groupByKey`。举个例子，`rdd.groupByKey().mapValues(_.sum)` 与 `rdd.reduceByKey(_ + _)` 执行的结果是一样的，但是前者需要把全部的数据通过网络传递一遍，而后者只需要根据每个 `key` 局部的 `partition` 累积结果，在 `shuffle` 的之后把局部的累积值相加后得到结果。
- 当输入和输入的类型不一致时，避免使用 `reduceByKey`。举个例子，我们需要实现为每一个 `key` 查找所有不相同的 `string`。一个方法是利用 `map` 把每个元素的转换成一个 `Set`，再使用 `reduceByKey` 将这些 `Set` 合并起来

```
rdd.map(kv => (kv._1, new Set[String]() + kv._2))
    .reduceByKey(_ ++ _)
```

这段代码生成了无数的非必须的对象，因为需要为每个 `record` 新建一个 `Set`。这里使用 `aggregateByKey` 更加适合，因为这个操作是在 `map` 阶段做聚合。

```

val zero = new collection.mutable.Set[String]()
rdd.aggregateByKey(zero)(
  (set, v) => set += v,
  (set1, set2) => set1 ++= set2)

```

- 避免 **flatMap-join-groupBy** 的模式。当有两个已经按照 `key` 分组的数据集，你希望将两个数据集合并，并且保持分组，这种情况可以使用 `cogroup`。这样可以避免对 `group` 进行装箱拆箱的开销。

3 什么时候不发生 Shuffle

当然了解在哪些 `transformation` 上不会发生 `shuffle` 也是非常重要的。当前一个 `transformation` 已经用相同的 `partitioner` 把数据分区了，`spark` 知道如何避免 `shuffle`。参考以下代码：

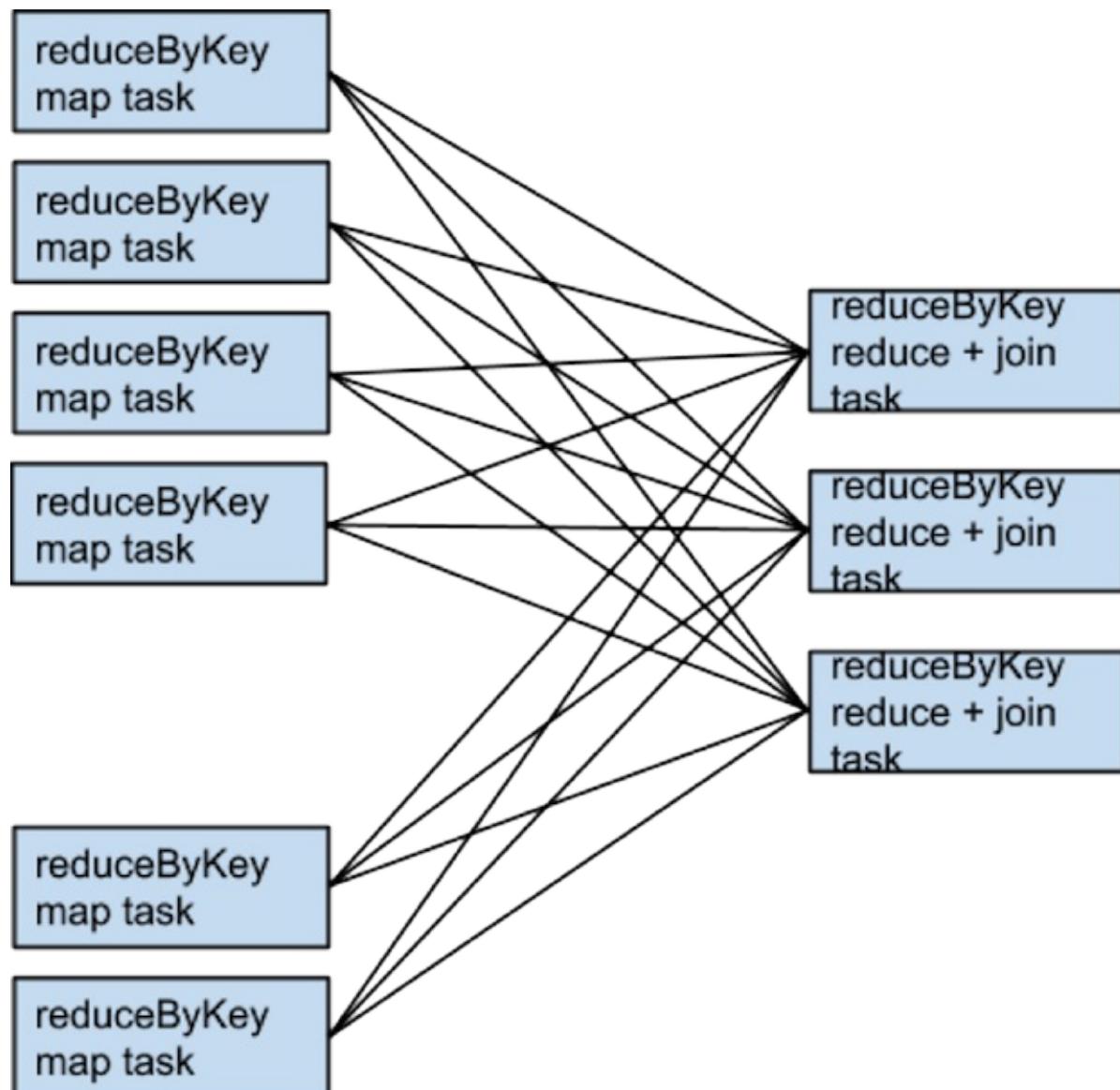
```

rdd1 = someRdd.reduceByKey(...)
rdd2 = someOtherRdd.reduceByKey(...)
rdd3 = rdd1.join(rdd2)

```

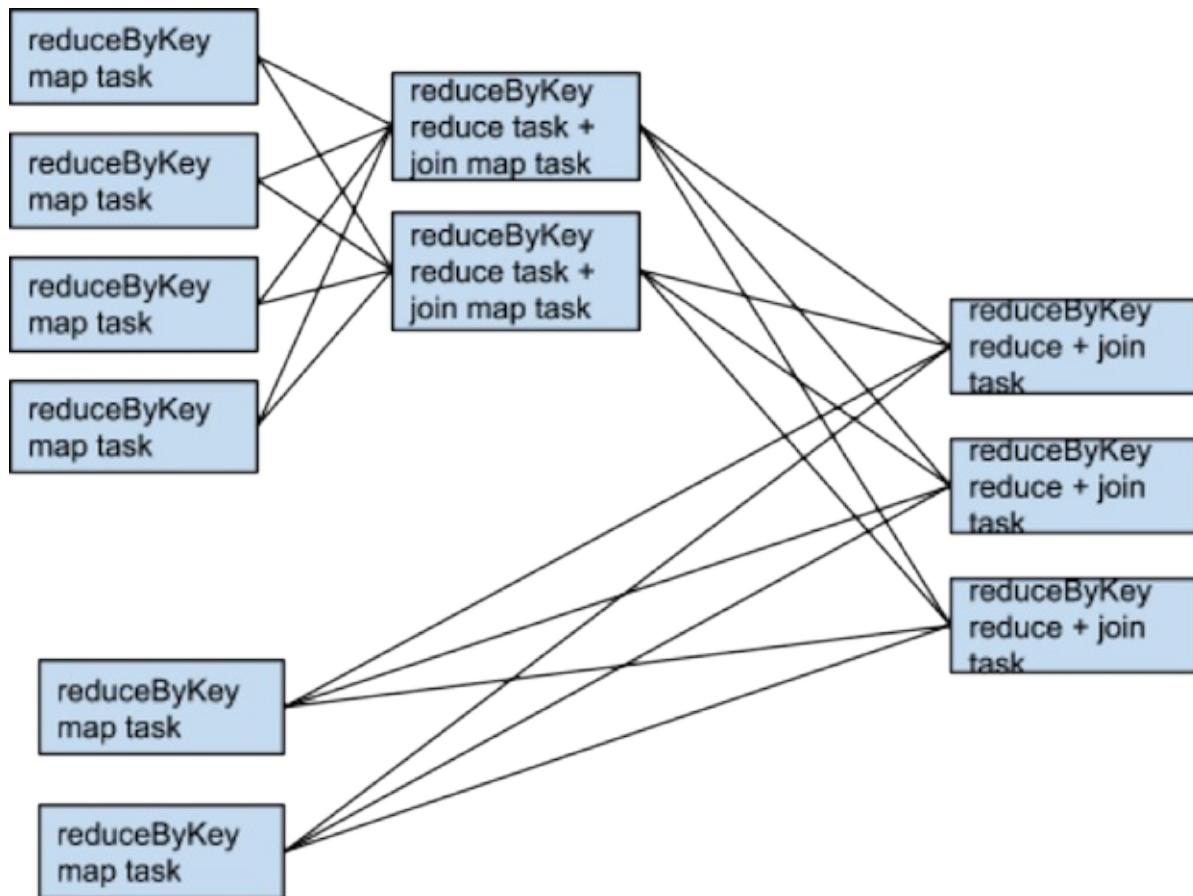
因为没有 `partitioner` 传递给 `reduceByKey`，所以系统使用默认的 `partitioner`，所以 `rdd1` 和 `rdd2` 都会使用 `hash` 进行分区。代码中的两个 `reduceByKey` 会发生两次 `shuffle`。如果 `RDD` 包含相同个数的 `partition`，`join` 的时候将不会发生额外的 `shuffle`。因为这里的 `RDD` 使用相同的 `hash` 方式进行 `partition`，所以全部 `RDD` 中同一个 `partition` 中的 `key` 的集合都是相同的。因此，`rdd3` 中一个 `partition` 的输出只依赖 `rdd2` 和 `rdd1` 的同一个对应的 `partition`，所以第三次 `shuffle` 是不必要的。

举个例子说，当 `someRdd` 有4个 `partition`，`someOtherRdd` 有两个 `partition`，两个 `reduceByKey` 都使用3个 `partition`，所有的 `task` 会按照如下的方式执行：



如果 `rdd1` 和 `rdd2` 在 `reduceByKey` 时使用不同的 `partitioner` 或者使用默认的 `partitioner`，但是 `partition` 的个数不同，那么在 `join` 时只用一个 `RDD` (`partition` 数更少的那个)需要重新 `shuffle`。

相同的 `transformation`，相同的输入，不同的 `partition` 个数的情况：



当两个数据集需要 `join` 时，避免 `shuffle` 的一个方法是使用 `broadcast variables`。如果一个数据集小到能够塞进一个 `executor` 的内存中，那么它就可以在 `driver` 中写入到一个 `hash table` 中，然后 `broadcast` 到所有的 `executor` 中。然后 `map transformation` 可以引用这个 `hash table` 作查询。

4 什么情况下 Shuffle 越多越好

尽可能减少 `shuffle` 的准则也有例外的场合。如果额外的 `shuffle` 能够增加并发那么这也能够提高性能。比如当你的数据保存在几个没有切分过的大文件中时，那么使用 `InputFormat` 产生分 `partition` 可能会导致每个 `partition` 中聚集了大量的 `record`，如果 `partition` 不够，导致没有启动足够的并发。在这种情况下，我们需要在数据载入之后使用 `repartition`（会导致 `shuffle`）提高 `partition` 的个数，这样能够充分使用集群的 `CPU`。

另外一种例外情况是在使用 `reduce` 或者 `aggregate action` 聚集数据到 `driver` 时，如果把 `partition` 个数很多的数据进行聚合时，单进程执行的 `driver` `merge` 所有 `partition` 的输出时很容易成为计算的瓶颈。为了缓解 `driver` 的计算压力，可以使用 `reduceByKey` 或者 `aggregateByKey` 执行分布式的 `aggregate` 操作把数据分布到更少的

`partition` 上。每个 `partition` 中的数据并行的进行 `merge`，再把 `merge` 的结果发给 `driver` 以进行最后一轮 `aggregation`。查看 `treeReduce` 和 `treeAggregate` 查看如何这么使用的例子。

这个技巧在已经按照 `key` 聚集的数据集上格外有效，比如当一个应用是需要统计一个语料库中每个单词出现的次数，并且把结果输出到一个 `map` 中。一个实现的方式是使用 `aggregation`，在每个 `partition` 中本地计算一个 `map`，然后在 `driver` 中把各个 `partition` 中计算的 `map merge` 起来。另一种方式是通过 `aggregateByKey` 把 `merge` 的操作分布到各个 `partiton` 中计算，然后在简单地通过 `collectAsMap` 把结果输出到 `driver` 中。

5 二次排序

还有一个重要的技能是了解接口 `repartitionAndSortWithinPartitions`。这是一个听起来很晦涩的 `transformation`，但是却能涵盖各种奇怪情况下的排序，这个 `transformation` 把排序推迟到 `shuffle` 操作中，这使大量的数据有效的输出，排序操作可以和其他操作合并。

例如，`Apache Hive on Spark` 在 `join` 的实现中，使用了这个 `transformation`。而且这个操作在 `secondary sort` 模式中扮演着至关重要的角色。`secondary sort` 模式是指用户期望数据按照 `key` 分组，并且希望按照特定的顺序遍历 `value`。使用 `repartitionAndSortWithinPartitions` 再加上一部分用户的额外的工作可以实现 `secondary sort`。

6 调试资源分配

在本章中，你将学会压榨出你集群的每一分资源。`spark` 推荐的配置将根据不同的集群管理系统（`YARN`、`Mesos`、`Spark Standalone`）而有所不同，我们将主要集中在 `YARN` 上，因为这个是 `cloudera` 推荐的方式。

`Spark`（以及 `YARN`）需要关心的两项主要的资源是 `CPU` 和 `内存`，`磁盘`和 `IO` 当然也影响着 `Spark` 的性能，但是不管是 `Spark` 还是 `Yarn` 目前都没法对他们做实时有效的管理。

在一个 `Spark` 应用中，每个 `spark executor` 拥有固定个数的 `core` 以及固定大小的堆大小。`core` 的个数可以在执行 `spark-submit` 或者 `spark-shell` 时，通过参数 `--executor-cores` 指定，也可以在 `spark-defaults.conf` 配置文件或者 `SparkConf` 对象中设置

`spark.executor.cores` 参数。同样地，堆的大小可以通过 `--executor-memory` 参数或者 `spark.executor.memory` 配置项配置。`core` 配置项控制一个 `executor` 中 `task` 的并发数。`--executor-cores 5` 意味着每个 `executor` 中最多同时可以有5个 `task` 运行。`memory` 参数影响 `Spark` 可以缓存的数据的大小，也就是在 `group,aggregate` 以及 `join` 操作时 `shuffle` 的数据结构的最大值。

`--num-executors` 命令行参数或者 `spark.executor.instances` 配置项控制需要的 `executor` 个数。从 `Spark 1.3` 开始，你可以避免使用这个参数，只要你通过设置 `spark.dynamicAllocation.enabled` 参数打开动态分配。动态分配可以使的 `Spark` 的应用在有积压的等待 `task` 时请求 `executor`，并且在空闲时释放这些 `executor`。

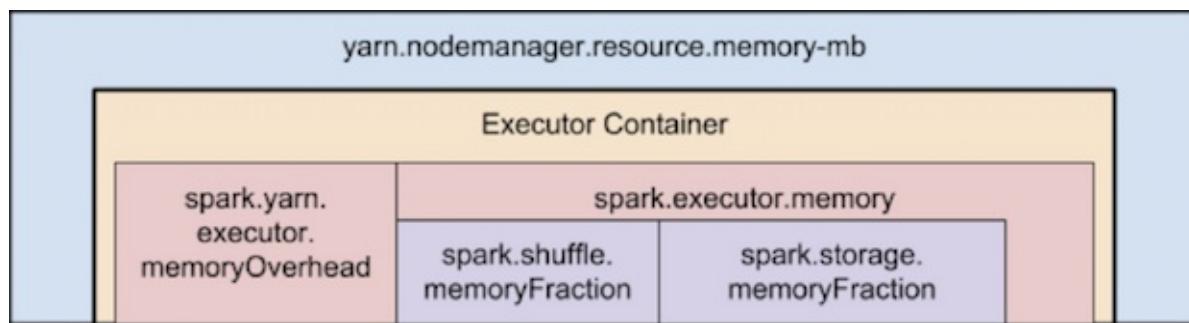
同时 `Spark` 需求的资源如何跟 `YARN` 中可用的资源配置也是需要着重考虑的，`YARN` 相关的参数有：

- `yarn.nodemanager.resource.memory-mb` 控制在每个节点上 `container` 能够使用的最大内存；
- `yarn.nodemanager.resource.cpu-vcores` 控制在每个节点上 `container` 能够使用的最大 `core` 个数；

请求5个 `core` 会生成向 `YARN` 要5个虚拟 `core` 的请求。但是从 `YARN` 请求内存相对比较复杂，因为以下的一些原因：

- `--executor-memory/spark.executor.memory` 控制 `executor` 的堆的大小，但是 `JVM` 本身也会占用一定的堆空间，比如 `interned String` 或者 `direct byte buffer`，`spark.yarn.executor.memoryOverhead` 属性决定了向 `YARN` 请求的每个 `executor` 的内存大小，默认值为 `max(384, 0.7 * spark.executor.memory)`；
- `YARN` 可能会比请求的内存高一点，`YARN` 的 `yarn.scheduler.minimum-allocation-mb` 和 `yarn.scheduler.increment-allocation-mb` 属性控制请求的最小值和增加量。

下面展示的是 `Spark on YARN` 内存结构：



如果以上这些还不够决定 Spark executor 个数，那么可以考虑下面一些概念：

- 应用程序的 master，是一个非 executor 的容器，它拥有从 YARN 请求资源的能力，它自己本身所占的资源也需要被计算在内。在 yarn-client 模式下，它默认请求 1024MB 和 1个 core。在 yarn-cluster 模式中，应用的 master 运行 driver，所以使用参数 --driver-memory 和 --driver-cores 配置它的资源常常很有用。
- 在 executor 执行的时候配置过大的 memory 经常会导致过长的 GC 延时，64G 是推荐的一个 executor 内存大小的上限。
- 我们注意到 HDFS client 在大量并发线程时会有性能问题。大概的估计是每个 executor 中最多5个并行的 task 就可以占满的写入带宽。
- 运行微型 executor（比如只有一个core而且只有够执行一个task的内存）会抛弃在一个 JVM 上同时运行多个 task 的好处。比如 broadcast 变量需要为每个 executor 复制一遍，这么多小 executor 会导致更多的数据拷贝。

为了让上面的说明更具体一点。我们举例子说明如何完全用满整个集群的资源。

假设一个集群中 NodeManager 运行在6个节点上，每个节点有16个 core 以及 64GB 的内存。那么 NodeManager 的容量为：yarn.nodemanager.resource.memory-mb 和 yarn.nodemanager.resource.cpu-vcores 可以设为 $6 * 16 * 1024 = 64512$ (MB) 和 15。我们避免使用 100% 的 YARN container 资源，因为还要为 os 和 hadoop 的 Daemon 留一部分资源。在上面的场景中，我们预留了1个 core 和1G的内存给这些进程。

所以看起来我们最先想到的配置会是这样的：--num-executors 6 --executor-cores 15 --executor-memory 63G。但是这个配置可能无法达到我们的需求，因为：

- 63GB+ 的 executor memory 塞不进只有63GB容量的 NodeManager；
- 应用程序的 master 也需要占用一个 core，意味着在某个节点上，没有15个 core 给

`executor` 使用；

- 15个 `core` 会影响 HDFS IO 的吞吐量。

配置成 `--num-executors 17 --executor-cores 5 --executor-memory 19G` 可能会效果更好，因为：

- 这个配置会在每个节点上生成3个 `executor`，除了应用的 `master` 运行的机器，这台机器上只会运行2个 `executor`；
- `--executor-memory` 被分成3份 ($63G/\text{每个节点3个executor}) = 21$ 。 $21 * (1 - 0.07) \sim 19$ 。

7 调试并发

我们知道 `Spark` 是一套数据并行处理的引擎。但是 `Spark` 并不是神奇得能够将所有计算并行化，它没办法从所有的并行化方案中找出最优的那个。每个 `Spark stage` 中包含若干个 `task`，每个 `task` 串行地处理数据。在调试 `Spark` 的 `job` 时，`task` 的个数可能是决定程序性能的最重要的参数。

那么这个数字是由什么决定的呢？前文介绍了 `Spark` 如何将 `RDD` 转换成一组 `stage`。`task` 的个数与 `stage` 中上一个 `RDD` 的 `partition` 个数相同。而一个 `RDD` 的 `partition` 个数与被它依赖的 `RDD` 的 `partition` 个数相同。除了以下的情况：`coalesce transformation` 可以创建一个具有更少 `partition` 个数的 `RDD`，`union transformation` 产出的 `RDD` 的 `partition` 个数是它父 `RDD` 的 `partition` 个数之和，`cartesian` 返回的 `RDD` 的 `partition` 个数是它们的积。

如果一个 `RDD` 没有父 `RDD` 呢？由 `textFile` 或者 `hadoopFile` 生成的 `RDD` 的 `partition` 个数由它们底层使用的 `MapReduce InputFormat` 决定的。一般情况下，每读到的一个 `HDFS block` 会生成一个 `partition`。通过 `parallelize` 接口生成的 `RDD` 的 `partition` 个数由用户指定，如果用户没有指定则由参数 `spark.default.parallelism` 决定。

要想知道 `partition` 的个数，可以通过接口 `rdd.partitions().size()` 获得。

这里最需要关心的问题在于 `task` 的个数太小。如果运行时 `task` 的个数比实际可用的 `slot` 还少，那么程序没法使用到所有的 `CPU` 资源。

过少的 `task` 个数可能会导致在一些聚集操作时，每个 `task` 的内存压力会很大。任何 `join`, `cogroup`, `*ByKey` 操作都会在内存生成一个 `hash-map` 或者 `buffer` 用于分组或者排序。`join`, `cogroup`, `groupByKey` 会在 `shuffle` 时在 `fetching` 端使用这些数据结构，`reduceByKey`, `aggregateByKey` 会在 `shuffle` 时在两端都会使用这些数据结构。

当需要进行这个聚集操作的 `record` 不能完全轻易塞进内存中时，一些问题就会暴露出来。首先，在内存中保有大量这些数据的 `record` 会增加 `GC` 的压力，可能会导致流程停顿下来。其次，如果数据不能完全载入内存，`Spark` 会将这些数据写到磁盘，这会引起磁盘 `IO` 和排序。这可能是导致 `Spark Job` 慢的首要原因。

那么如何增加你的 `partition` 的个数呢？如果你的 `stage` 是从 `Hadoop` 读取数据，你可以做以下的选项：

- 使用 `repartition` 选项，会引发 `shuffle`；
- 配置 `InputFormat`，将文件分得更小；
- 写入 `HDFS` 文件时使用更小的 `block`。

如果 `stage` 从其他 `stage` 中获得输入，引发 `stage` 边界的操作会接受一个 `numPartitions` 的参数，比如

```
val rdd2 = rdd1.reduceByKey(_ + _, numPartitions = x)
```

`x` 应该取什么值？最直接的方法就是做实验。不停的将 `partition` 的个数从上次实验的 `partition` 个数乘以 `1.5`，直到性能不再提升为止。

同时也有一些原则用于计算 `x`，但是也不是非常的有效是因为有些参数是很难计算的。这里写到不是因为它们很实用，而是可以帮助理解。这里主要的目标是启动足够的 `task` 可以使得每个 `task` 接受的数据能够都塞进它所分配到的内存中。

每个 `task` 可用的内存通过这个公式计算：`spark.executor.memory * spark.shuffle.memoryFraction * spark.shuffle.safetyFraction) / spark.executor.cores`。
`memoryFraction` 和 `safetyFraction` 默认值分别 `0.2` 和 `0.8`。

在内存中所有 `shuffle` 数据的大小很难确定。最可行的是找出一个 `stage` 运行的 `Shuffle Spill (memory)` 和 `Shuffle Spill(Disk)` 之间的比例。再用所有 `shuffle` 写乘以这个比例。但是如果这个 `stage` 是 `reduce` 时，可能会有点复杂：

$$\frac{(\text{observed shuffle write}) * (\text{observed shuffle spill memory}) * (\text{spark.executor.cores})}{(\text{observed shuffle spill disk}) * (\text{spark.executor.memory}) * (\text{spark.shuffle.memoryFraction}) * (\text{spark.shuffle.safetyFraction})}$$

在有所疑虑的时候，使用更多的 `task` 数（也就是 `partition` 数）通常效果会更好，这与 `MapRecuce` 中建议 `task` 数目选择尽量保守的建议相反。这个因为 `MapReduce` 在启动 `task` 时相比需要更大的代价。

8 压缩数据结构

`Spark` 的数据流由一组 `record` 构成。一个 `record` 有两种表达形式：一种是反序列化的 `Java` 对象，另外一种是序列化的二进制形式。通常情况下，`Spark` 对内存中的 `record` 使用反序列化之后的形式，对要存到磁盘上或者需要通过网络传输的 `record` 使用序列化之后的形式。也有计划在内存中存储序列化之后的 `record`。

`spark.serializer` 控制这两种形式之间的转换的方式。`Kryo serializer`，`org.apache.spark.serializer.KryoSerializer` 是推荐的选择。但不幸的是它不是默认的配置，因为 `KryoSerializer` 在早期的 `Spark` 版本中不稳定，而 `Spark` 不想打破版本的兼容性，所以没有把 `KryoSerializer` 作为默认配置，但是 `KryoSerializer` 应该在任何情况下都是第一的选择。

`record` 在这两种形式切换的频率对 `Spark` 应用的运行效率具有很大的影响。检查传递的数据的类型，看看能否改进是非常值得一试的。

过多的反序列化 `record` 可能会导致数据 `spill` 到磁盘更加频繁，减少缓存的对象的个数。

过多的序列化 `record` 导致更多的磁盘和网络 `IO`，同样也会使得能够 `cache` 在内存中的 `record` 个数减少，这里主要的解决方案是把所有的用户自定义的 `class` 都通过 `SparkConf#registerKryoClasses` 的 `API` 定义和传递。

9 数据格式

任何时候你都可以决定你的数据以怎样的格式保存在磁盘上，可以使用可扩展的二进制格式比如：`Avro`，`Parquet`，`Thrift` 或者 `Protobuf`。当人们在谈论在 `Hadoop` 上使用 `Avro`，`Thrift` 或者 `Protobuf` 时，意味着每个 `record` 是一个 `Avro/Thrift/Protobuf` 结构，并保存成 `sequence file`。而不是 `JSON` 格式。

参考文献

[【1】How-to: Tune Your Apache Spark Jobs \(Part 1\)](#)

[【2】How-to: Tune Your Apache Spark Jobs \(Part 2\)](#)

[【3】Tuning Spark](#)

Databricks Spark 知识库

1 最佳实践

1.1 避免使用 GroupByKey

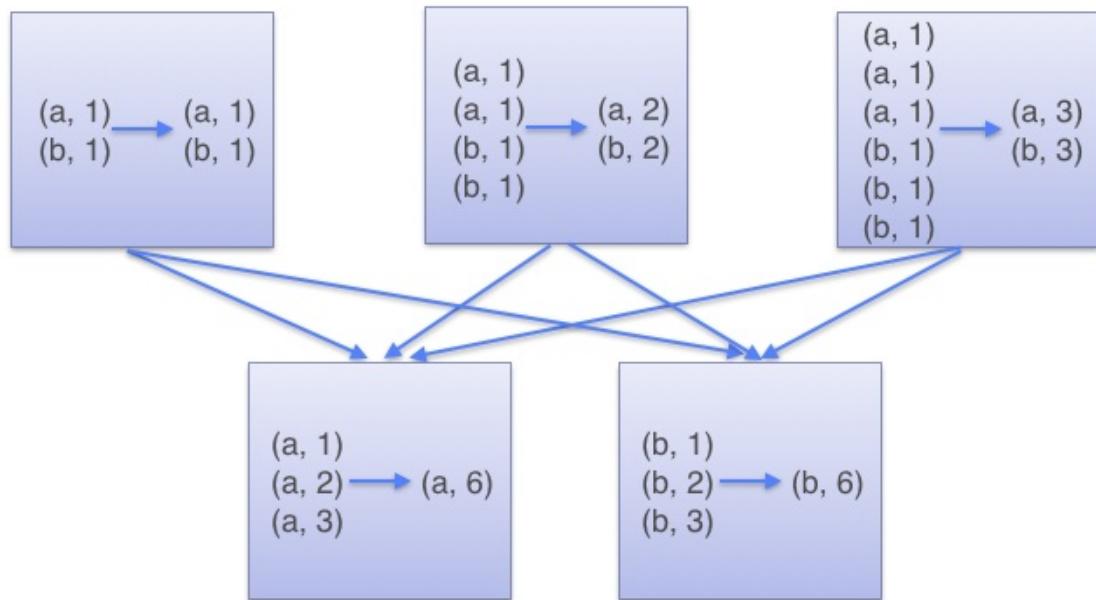
让我们看一下使用两种不同的方式去计算单词的个数，第一种方式使用 `reduceByKey`，另外一种方式使用 `groupByKey`：

```
val words = Array("one", "two", "two", "three", "three", "three")
val wordPairsRDD = sc.parallelize(words).map(word => (word, 1))
//reduce
val wordCountsWithReduce = wordPairsRDD
  .reduceByKey(_ + _)
  .collect()
//group
val wordCountsWithGroup = wordPairsRDD
  .groupByKey()
  .map(t => (t._1, t._2.sum))
  .collect()
```

虽然两个函数都能得出正确的结果，`reduceByKey` 更适合使用在大数据集上。这是因为 Spark 知道它可以在每个分区 `shuffle` 数据之前，聚合 `key` 值相同的数据。

借助下图可以理解在 `reduceByKey` 里发生了什么。注意在数据对被 `shuffle` 前同一机器上同样 `key` 的数据是怎样被组合的(`reduceByKey` 中的 `lambda` 函数)。然后 `lambda` 函数在每个区上被再次调用来将所有值 `reduce` 成一个最终结果。

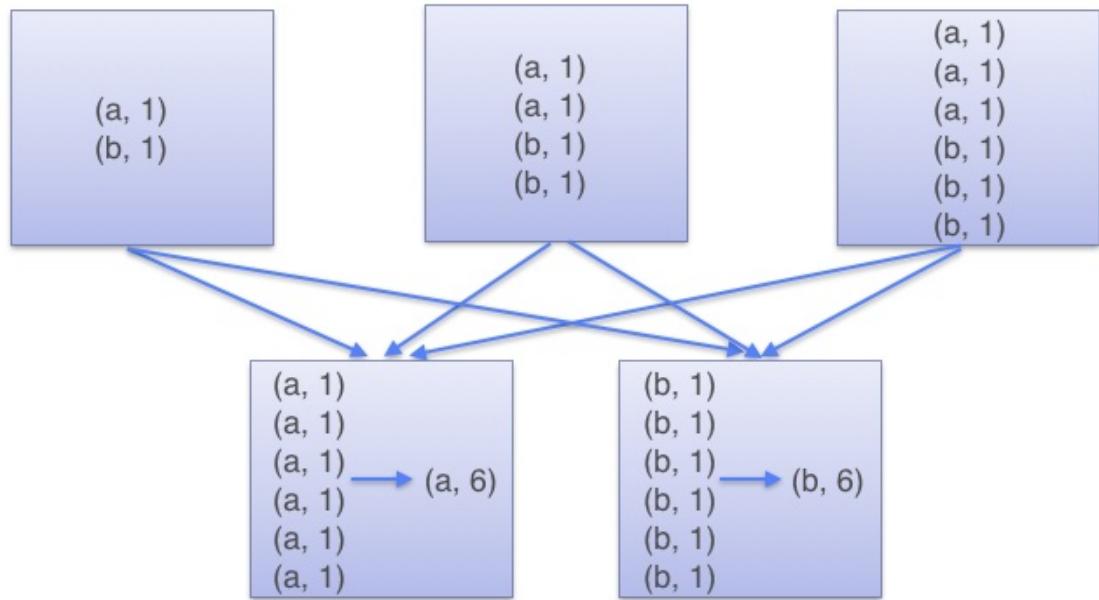
ReduceByKey



但是，当调用 `groupByKey` 时，所有的键值对(key-value pair)都会被 `shuffle` 。在网络上传输这些数据非常没有必要。

为了确定将数据对 `shuffle` 到哪台主机，`Spark` 会对数据对的 `key` 调用一个分区函数。当 `shuffle` 的数据量大于单台执行机器内存总量时，`Spark` 会把数据保存到磁盘上。不过在保存时每次只会处理一个 `key` 的数据，所以当单个 `key` 的键值对超过内存容量会存在内存溢出的可能。我们应避免将数据保存到磁盘上，这会严重影响性能。

GroupByKey



你可以想象一个非常大的数据集，在使用 `reduceByKey` 和 `groupByKey` 时他们的差别会被放大更多倍。

以下函数应该优先于 `groupByKey` :

- `combineByKey` 组合数据，但是组合之后的数据类型与输入时值的类型不一样。
- `foldByKey` 合并每一个 `key` 的所有值，在级联函数和“零值”中使用。

1.2 不要将大型 **RDD** 的所有元素拷贝到**driver**

如果你的 `driver` 内存容量不能容纳一个大型 `RDD` 里面的所有数据，不要做以下操作：

```
val values = myVeryLargeRDD.collect()
```

`Collect` 操作会试图将 `RDD` 里面的每一条数据复制到 `driver` 上，这时候会发生内存溢出和崩溃。相反，你可以调用 `take` 或者 `takeSample` 来确保数据大小的上限。或者在你的 `RDD` 中使用过滤或抽样。同样，要谨慎使用下面的操作，除非你能确保数据集小到足以存储在内存中：

- `countByKey`
- `countByValue`
- `collectAsMap`

如果你确实需要将 `RDD` 里面的大量数据保存在内存中，你可以将 `RDD` 写成一个文件或者把 `RDD` 导出到一个容量足够大的数据库中。

1.3 优雅地处理坏的输入数据

当处理大量的数据的时候，一个常见的问题是有些数据格式不对或者内容有误。使用 `filter` 方法可以很容易丢弃坏的输入或者使用 `map` 方法可以修复可能修复的坏的数据。当你尝试着修复坏的数据，但是丢弃无法被修复的数据时，`flatMap` 函数是最好的选择。让我们考虑下面的输入 `json` 串。

```
input_rdd = sc.parallelize([
    "{\"value\": 1}", # Good
    "bad_json", # Bad
    "{\"value\": 2}", # Good
    "{\"value\": 3}" # Missing an ending brace.
])
```

当我们尝试着在 `SQLContext` 中使用这个输入串时，很明显它会因为格式不对而报错。

```
sqlContext.jsonRDD(input_rdd).registerTempTable("valueTable")
# The above command will throw an error.
```

让我妈用下面的 `python` 代码修复输入数据。

```
def try_correct_json(json_string):
    try:
        # First check if the json is okay.
        json.loads(json_string)
        return [json_string]
    except ValueError:
        try:
            # If not, try correcting it by adding a ending brace.
            try_to_correct_json = json_string + "}"
            json.loads(try_to_correct_json)
            return [try_to_correct_json]
        except ValueError:
            # The malformed json input can't be recovered, drop this input.
            return []
```

经过上面函数的处理之后，我们就可以使用这些数据了。

```
corrected_input_rdd = input_rdd.flatMap(try_correct_json)
sqlContext.jsonRDD(corrected_input_rdd).registerTempTable("valueTable")
sqlContext.sql("select * from valueTable").collect()
# Returns [Row(value=1), Row(value=2), Row(value=3)]
```

2 常规故障处理

2.1 Job aborted due to stage failure: Task not serializable

如果你看到以下错误：

```
org.apache.spark.SparkException: Job aborted due to stage failure:  
Task not serializable: java.io.NotSerializableException: ...
```

上述的错误在这种情况下会发生：当你在 `master` 上初始化一个变量，但是试图在 `worker` 上使用。在这个示例中，`Spark Streaming` 试图将对象序列化之后发送到 `worker` 上，如果这个对象不能被序列化就会失败。思考下面的代码片段：

```
NotSerializable notSerializable = new NotSerializable();  
JavaRDD<String> rdd = sc.textFile("/tmp/myfile");  
rdd.map(s -> notSerializable.doSomething(s)).collect();
```

这段代码会触发上面的错误。这里有一些建议修复这个错误：

- 让 `class` 实现序列化
- 在作为参数传递给 `map` 方法的 `lambda` 表达式内部声明实例
- 在每一台机器上创建一个 `NotSerializable` 的静态实例
- 调用 `rdd.foreachPartition` 并且像下面这样创建 `NotSerializable` 对象：

```
rdd.foreachPartition(iter -> {  
    NotSerializable notSerializable = new NotSerializable();  
    // ...Now process iter  
});
```

2.2 缺失依赖

在默认状态下，`Maven` 在 `build` 的时候不会包含所依赖的 `jar` 包。当运行一个 `spark` 任务时，如果 `Spark worker` 机器上没有包含所依赖的 `jar` 包会发生类无法找到的错误(`ClassNotFoundException`)。

有一个简单的方式，在 `Maven` 打包的时候创建 `shaded` 或 `uber` 任务可以让那些依赖的 `jar` 包很好地打包进去。

使用 `<scope>provided</scope>` 可以排除那些没有必要打包进去的依赖，对 `Spark` 的依赖必须使用 `provided` 标记，因为这些依赖已经包含在 `Spark cluster` 中。在你的 `worker` 机器上已经安装的 `jar` 包你同样需要排除掉它们。

下面是一个 `Maven pom.xml` 的例子，工程了包含了一些需要的依赖，但是 `Spark` 的 `libraries` 不会被打包进去，因为它使用了 `provided`：

```
<project>
  <groupId>com.databricks.apps.logs</groupId>
  <artifactId>log-analyzer</artifactId>
  <modelVersion>4.0.0</modelVersion>
  <name>Databricks Spark Logs Analyzer</name>
  <packaging>jar</packaging>
  <version>1.0</version>
  <repositories>
    <repository>
      <id>Akka repository</id>
      <url>http://repo.akka.io/releases</url>
    </repository>
  </repositories>
  <dependencies>
    <dependency> <!-- Spark -->
      <groupId>org.apache.spark</groupId>
      <artifactId>spark-core_2.10</artifactId>
      <version>1.1.0</version>
      <scope>provided</scope>
    </dependency>
    <dependency> <!-- Spark SQL -->
      <groupId>org.apache.spark</groupId>
      <artifactId>spark-sql_2.10</artifactId>
      <version>1.1.0</version>
      <scope>provided</scope>
    </dependency>
    <dependency> <!-- Spark Streaming -->
      <groupId>org.apache.spark</groupId>
      <artifactId>spark-streaming_2.10</artifactId>
      <version>1.1.0</version>
      <scope>provided</scope>
    </dependency>
    <dependency> <!-- Command Line Parsing -->
      <groupId>commons-cli</groupId>
      <artifactId>commons-cli</artifactId>
      <version>1.2</version>
    </dependency>
  </dependencies>
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>2.3.2</version>
        <configuration>
          <source>1.8</source>
          <target>1.8</target>
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>
```

```

<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-shade-plugin</artifactId>
    <version>2.3</version>
    <executions>
        <execution>
            <phase>package</phase>
            <goals>
                <goal>shade</goal>
            </goals>
        </execution>
    </executions>
    <configuration>
        <filters>
            <filter>
                <artifact>*:*</artifact>
                <excludes>
                    <exclude>META-INF/*.SF</exclude>
                    <exclude>META-INF/*.DSA</exclude>
                    <exclude>META-INF/*.RSA</exclude>
                </excludes>
            </filter>
        </filters>
        <finalName>uber-${project.artifactId}-${project.version}</finalName>
    >
        </configuration>
    </plugin>
</plugins>
</build>
</project>

```

2.3 执行 start-all.sh 错误: Connection refused

如果是使用 Mac 操作系统运行 start-all.sh 发生下面错误时：

```
% sh start-all.sh
starting org.apache.spark.deploy.master.Master, logging to ...
localhost: ssh: connect to host localhost port 22: Connection refused
```

你需要在你的电脑上打开“远程登录”功能。进入 系统偏好设置 ---> 共享 勾选打开 远程登录。

2.4 Spark 组件之间的网络连接问题

Spark 组件之间的网络连接问题会导致各式各样的警告或错误：

- **SparkContext <-> Spark Standalone Master**

如果 `SparkContext` 不能连接到 `Spark standalone master`，会显示下面的错误：

```
ERROR AppClient$ClientActor: All masters are unresponsive! Giving up.
ERROR SparkDeploySchedulerBackend: Spark cluster looks dead, giving up.
ERROR TaskSchedulerImpl: Exiting due to error from cluster scheduler:
Spark cluster looks down
```

如果 `driver` 能够连接到 `master` 但是 `master` 不能回连到 `driver`，这时 `Master` 的日志会记录多次尝试连接 `driver` 失败并且会报告不能连接：

```
INFO Master: Registering app SparkPi
INFO Master: Registered app SparkPi with ID app-XXX-0000
INFO: Master: Removing app app-app-XXX-0000
[...]
INFO Master: Registering app SparkPi
INFO Master: Registered app SparkPi with ID app-YYY-0000
INFO: Master: Removing app app-YYY-0000
[...]
```

在这样的情况下，`master` 报告应用已经被成功地注册了。但是注册成功的通知 `driver` 接收失败了，这时 `driver` 会自动尝试几次重新连接直到失败的次数太多而放弃重试。其结果是 `Master web UI` 会报告多个失败的应用，即使只有一个 `SparkContext` 被创建。

如果你遇到上述的错误，有两条可以遵循的建议：

- 检查 `workers` 和 `drivers` 配置的 `Spark master` 的地址
- 设置 `driver`, `master`, `worker` 的 `SPARK_LOCAL_IP` 为集群的可寻地址主机名。

配置 hostname/port

这节将描述我们如何绑定 `Spark` 组件的网络接口和端口。在每节里，配置会按照优先级降序的方式排列。如果前面所有配置没有提供则使用最后一条作为默认配置。

SparkContext actor system:

Hostname:

- `spark.driver.host` 属性
- 如果 `SPARK_LOCAL_IP` 环境变量的设置是主机名(hostname)，就会使用设置时的主机名。如果 `SPARK_LOCAL_IP` 设置的是一个 IP 地址，这个 IP 地址会被解析为主机名。
- 使用默认的 IP 地址，这个 IP 地址是 Java 接口 `InetAddress.getLocalHost` 方法的返回值。

Port:

- `spark.driver.port` 属性。

- 从操作系统(OS)选择一个临时端口。

Spark Standalone Master / Worker actor systems:

Hostname:

- 当 Master 或 Worker 进程启动时使用 --host 或 -h 选项(或是过期的选项 --ip 或 -i)。
- SPARK_MASTER_HOST 环境变量(仅应用在 Master 上)。
- 如果 SPARK_LOCAL_IP 环境变量的设置是主机名(hostname)，就会使用设置时的主机名。如果 SPARK_LOCAL_IP 设置的是一个 IP 地址，这个 IP 地址会被解析为主机名。
- 使用默认的 IP 地址，这个 IP 地址是Java 接口 InetAddress.getLocalHost 方法的返回值.

Port:

- 当 Master 或 Worker 进程启动时使用 --port 或 -p 选项。
- SPARK_MASTER_PORT 或 SPARK_WORKER_PORT 环境变量(分别应用到 Master 和 Worker 上)。
- 从操作系统(OS)选择一个临时端口。

3 性能和优化

3.1 一个 RDD 有多少分区

在调试和故障处理的时候，我们通常有必要知道 `RDD` 有多少个分区。这里有几个方法可以找到这些信息：

使用 UI 查看在分区上执行的任务数

当 `stage` 执行的时候，你可以在 `Spark UI` 上看到这个 `stage` 上的分区数。下面的例子中的简单任务在 4 个分区上创建了共 100 个元素的 `RDD`，然后在这些元素被收集到 `driver` 之前分发一个 `map` 任务：

```
scala> val someRDD = sc.parallelize(1 to 100, 4)
someRDD: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[0] at parallelize at <console>:12

scala> someRDD.map(x => x).collect
res1: Array[Int] = Array(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18,
 19, 20, 21, 22, 23,
24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45,
46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59,
60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81
, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100)
```

在 Spark 的应用 UI 里，从下面截图上看到的 "Total Tasks" 代表了分区数。

The screenshot shows the Spark Shell application UI at `localhost:4040/stages/`. The main title is "Spark Stages". Below it, the following metrics are displayed:

- Total Duration: 5.9 min
- Scheduling Mode: FIFO
- Active Stages: 0
- Completed Stages: 1
- Failed Stages: 0

Under "Completed Stages (1)", there is a table with the following data:

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Shuffle Read	Shuffle Write
0	collect at <console>:15 +details	2014/09/17 14:49:51	71 ms	4/4			

The "Tasks: Succeeded/Total" column for stage 0 is highlighted with a red box.

使用 UI 查看分区缓存

持久化 RDD 时通常需要知道有多少个分区被存储。下面的这个例子和之前的一样，除了现在我们要对 RDD 做缓存处理。操作完成之后，我们可以在 UI 上看到这个操作导致什么被我们存储了。

```
scala> someRDD.setName("toy").cache
res2: someRDD.type = toy ParallelCollectionRDD[0] at parallelize at <console>:12

scala> someRDD.map(x => x).collect
res3: Array[Int] = Array(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18,
, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37,
38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59
,
60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81
, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100)
```

注意：下面的截图有 4 个分区被缓存。

RDD Name	Storage Level	Cached Partitions	Fraction Cached	Size in Memory	Size in Tachyon	Size on Disk
toy	Memory Deserialized 1x Replicated	4	100%	2.8 KB	0.0 B	0.0 B

Storage

RDD Name	Storage Level	Cached Partitions	Fraction Cached	Size in Memory	Size in Tachyon	Size on Disk
toy	Memory Deserialized 1x Replicated	4	100%	2.8 KB	0.0 B	0.0 B

编程查看 RDD 分区

在 Scala API 里，RDD 持有一个分区数组的引用，你可以使用它找到有多少个分区：

```
scala> val someRDD = sc.parallelize(1 to 100, 30)
someRDD: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[0] at parallelize at <console>:12

scala> someRDD.partitions.size
res0: Int = 30
```

在 Python API 里，有一个方法可以明确地列出有多少个分区：

```
In [1]: someRDD = sc.parallelize(range(101), 30)
In [2]: someRDD.getNumPartitions()
Out[2]: 30
```

3.2 数据本地性

Spark 是一个并行数据处理框架，这意味着任务应该在离数据尽可能近的地方执行(即最少的数据传输)。

检查本地性

检查任务是否在本地运行的最好方式是在 Spark UI 上查看 stage 信息，注意下面截图中的 Locality Level 列显示任务运行在哪个地方。

Details for Stage 360

Total task time across all tasks: 0.1 s

Summary Metrics for 8 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Result serialization time	0 ms	0 ms	0 ms	0 ms	0 ms
Duration	1 ms	2 ms	2 ms	3 ms	0.1 s
Time spent fetching task results	0 ms	0 ms	0 ms	0 ms	0 ms
Scheduler delay	17 ms	17 ms	18 ms	18 ms	19 ms

Aggregated Metrics by Executor

Executor ID	Address	Task Time	Total Tasks	Failed Tasks	Succeeded Tasks	Input	Shuffle Read	Shuffle Write	Shuffle Spill (Memory)	Shuffle Spill (Disk)
0	ip-10-0-236-90.us-west-2.compute.internal:38951	0.3 s	8	0	8	0.0 B	0.0 B	0.0 B	0.0 B	0.0 B

Tasks

Index	ID	Attempt	Status	Locality Level	Executor	Launch Time	Duration	GC Time	Accumulators	Errors
2	2748	0	SUCCESS	PROCESS_LOCAL	10.2.145.133:38951	2014/09/18 00:09:56	2 ms			
1	2747	0	SUCCESS	PROCESS_LOCAL	10.2.145.133:38951	2014/09/18 00:09:56	2 ms			
0	2746	0	SUCCESS	PROCESS_LOCAL	10.2.145.133:38951	2014/09/18 00:09:56	3 ms			
4	2750	0	SUCCESS	PROCESS_LOCAL	10.2.145.133:38951	2014/09/18 00:09:56	1 ms			
7	2753	0	SUCCESS	PROCESS_LOCAL		2014/09/18 00:09:56	0.1 s			

调整本地性配置

你可以调整 Spark 在每个数据本地性 level (data local --> process local --> node local --> rack local --> Any)上等待的时长。更多详细的参数信息请查看程序配置文档的 Scheduling 章节里类似于 spark.locality.* 的配置。

4 Spark Streaming

ERROR OneForOneStrategy

如果你在 `Spark Streaming` 里启用 `checkpointing`，`foreachRDD` 函数使用的对象都应该可以被序列化(`Serializable`)。否则会出现这样的异常 "ERROR OneForOneStrategy: ...
`java.io.NotSerializableException:`"

```
JavaStreamingContext jssc = new JavaStreamingContext(sc, INTERVAL);

// This enables checkpointing.
jssc.checkpoint("/tmp/checkpoint_test");

JavaDStream<String> dStream = jssc.socketTextStream("localhost", 9999);

NotSerializable notSerializable = new NotSerializable();
dStream.foreachRDD(rdd -> {
    if (rdd.count() == 0) {
        return null;
    }
    String first = rdd.first();

    notSerializable.doSomething(first);
    return null;
});
// This does not work!!!!
```

按照下面的方式之一进行修改，上面的代码才能正常运行：

- 在配置文件里面删除 `jssc.checkpoint` 这一行关闭 `checkpointing`。
- 让对象能被序列化。
- 在 `foreachRDD` 函数里面声明 `NotSerializable`，下面的示例代码是可以正常运行的：

```
JavaStreamingContext jssc = new JavaStreamingContext(sc, INTERVAL);

jssc.checkpoint("/tmp/checkpoint_test");

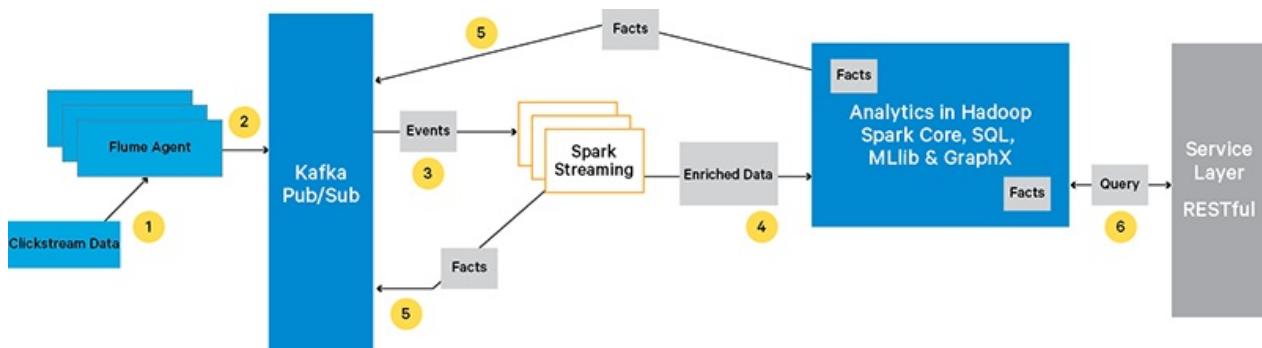
JavaDStream<String> dStream = jssc.socketTextStream("localhost", 9999);

dStream.foreachRDD(rdd -> {
    if (rdd.count() == 0) {
        return null;
    }
    String first = rdd.first();
    NotSerializable notSerializable = new NotSerializable();
    notSerializable.doSomething(first);
    return null;
});
// This code snippet is fine since the NotSerializable object
// is declared and only used within the forEachRDD function.
```

Cigna优化Spark Streaming实时处理应用

1 框架一览

事件处理的架构图如下所示。



2 优化总结

当我们第一次部署整个方案时， kafka 和 flume 组件都执行得非常好，但是 spark streaming 应用需要花费4-8分钟来处理单个 batch 。这个延迟的原因有两点，一是我们使用 DataFrame 来强化数据，而强化数据需要从 hive 中读取大量的数据；二是我们的参数配置不理想。

为了优化我们的处理时间，我们从两方面着手改进：第一，缓存合适的数据和分区；第二，改变配置参数优化spark应用。运行spark应用的 spark-submit 命令如下所示。通过参数优化和代码改进，我们显著减少了处理时间，处理时间从4-8分钟降到了低于25秒。

```

./opt/app/dev/spark-1.5.2/bin/spark-submit \
--jars \
/opt/cloudera/parcels/CDH/jars/zkclient-0.3.jar,/opt/cloudera/parcels/CDH/jars/kafka_2
.10-0.8.1.1.jar,\
./opt/app/dev/jars/datanucleus-core-3.2.2.jar,/opt/app/dev/jars/datanucleus-api-jdo-3.2
.1.jar,/opt/app/dev/jars/datanucleus-rdbms-3.2.1.jar \
--files /opt/app/dev/spark-1.5.2/conf/hive-site.xml,/opt/app/dev/jars/log4j-eir.proper
ties \
--queue spark_service_pool \
--master yarn \
--deploy-mode cluster \
--conf "spark.ui.showConsoleProgress=false" \
--conf "spark.driver.extraJavaOptions=-XX:MaxPermSize=6G -XX:+UseConcMarkSweepGC -Dlog
4j.configuration=log4j-eir.properties" \
--conf "spark.sql.tungsten.enabled=false" \
--conf "spark.eventLog.dir=hdfs://nameservice1/user/spark/applicationHistory" \
--conf "spark.eventLog.enabled=true" \
--conf "spark.sqlcodegen=false" \
--conf "spark.sql.unsafe.enabled=false" \
--conf "spark.executor.extraJavaOptions=-XX:+UseConcMarkSweepGC -Dlog4j.configuration=
log4j-eir.properties" \
--conf "spark.streaming.backpressure.enabled=true" \
--conf "spark.locality.wait=1s" \
--conf "spark.streaming.blockInterval=1500ms" \
--conf "spark.shuffle.consolidateFiles=true" \
--driver-memory 10G \
--executor-memory 8G \
--executor-cores 20 \
--num-executors 20 \
--class com.bigdata.streaming.OurApp \
./opt/app/dev/jars/OurStreamingApplication.jar e
xternal_props.conf

```

下面我们将详细介绍这些改变的参数。

2.1 driver选项

这里需要注意的是，`driver`运行在`spark on yarn`的集群模式下。因为`spark streaming`应用是一个长期运行的任务，生成的日志文件会很大。为了解决这个问题，我们限制了写入日志的消息的条数，并且用`RollingFileAppender`限制了它们的大小。我们也关闭了`spark.ui.showConsoleProgress`选项来禁用控制台日志消息。

通过测试，我们的`driver`因为永久代空间填满而频繁发生内存耗尽（永久代空间是类、方法等存储的地方，不会被重新分配）。将永久代空间的大小升高到6G可以解决这个问题。

```
spark.driver.extraJavaOptions=-XX:MaxPermSize=6G
```

2.2 垃圾回收

因为我们的 spark streaming 应用程序是一个长期运行的进程，在处理一段时间之后，我们注意到 gc 暂停时间过长，我们想在后台减少或者保持这个时间。调整 UseConcMarkSweepGC 参数是一个技巧。

```
--conf "spark.executor.extraJavaOptions=-XX:+UseConcMarkSweepGC -Dlog4j.configuration=log4j-eir.properties" \
```

2.3 禁用Tungsten

Tungsten 是 spark 执行引擎主要的改进。但是它的第一个版本是有问题的，所以我们暂时禁用它。

```
spark.sql.tungsten.enabled=false  
spark.sqlcodegen=false  
spark.sql.unsafe.enabled=false
```

2.4 启用反压

Spark Streaming 在批处理时间大于批间隔时间时会出现问题。换一句话说，就是 spark 读取数据的速度慢于 kafka 数据到达的速度。如果按照这个吞吐量执行过长的时间，它会造成不稳定的情况。即接收 executor 的内存溢出。设置下面的参数解决这个问题。

```
spark.streaming.backpressure.enabled=true
```

2.5 调整本地化和块配置

下面的两个参数是互补的。一个决定了数据本地化到 task 或者 executor 等待的时间，另外一个被 spark streaming receiver 使用对数据进行组块。块越大越好，但是如果数据没有本地化到 executor，它将会通过网络移动到任务执行的地方。我们必须在这两个参数间找到一个好的平衡，因为我们不想数据块太大，并且也不想等待本地化太长时间。我们希望所有的任务都在几秒内完成。

因此，我们改变本地化选项从3s到1s，我们也改变块间隔为1.5s。

```
--conf "spark.locality.wait=1s" \  
--conf "spark.streaming.blockInterval=1500ms" \
```

2.6 合并临时文件

在 ext4 文件系统中，推荐开启这个功能。因为这会产生更少的临时文件。

```
--conf "spark.shuffle.consolidateFiles=true" \
```

2.7 开启 executor 配置

在你配置 kafka Dstream 时，你能够指定并发消费线程的数量。然而， kafka Dstream 的消费者会运行在相同的 spark driver 节点上面。因此，为了从多台机器上面并行消费 kafka topic ，我们必须实例化多个 Dstream 。虽然可以在处理之前合并相应的 RDD ，但是运行多个应用程序实例，把它们都作为相同 kafka consumer group 的一部分。

为了达到这个目的，我们设置20个 executor，并且每个 executor 有20个核。

```
--executor-memory 8G  
--executor-cores 20  
--num-executors 20
```

2.8 缓存方法

使用 RDD 之前缓存 RDD ，但是记住在下次迭代之前从缓存中删除它。缓存那些需要使用多次的数据非常有用。然而，不要使分区数目过大。保持分区数目较低可以减少，最小化调度延迟。下面的公式是我们使用的分区数的计算公式。

```
# of executors * # of cores = # of partitions
```

参考文献

[【1】How Cigna Tuned Its Spark Streaming App for Real-time Processing with Apache Kafka](#)

Spark性能优化指南——基础篇

1 前言

在大数据计算领域，Spark已经成为了越来越流行、越来越受欢迎的计算平台之一。Spark的功能涵盖了大数据领域的离线批处理、SQL类处理、流式/实时计算、机器学习、图计算等各种不同类型的计算操作，应用范围与前景非常广泛。大多数同学（包括笔者在内），最初开始尝试使用Spark的原因很简单，主要就是为了让大数据计算作业的执行速度更快、性能更高。

然而，通过Spark开发出高性能的大数据计算作业，并不是那么简单的。如果没有对Spark作业进行合理的调优，Spark作业的执行速度可能会很慢，这样就完全体现不出Spark作为一种快速大数据计算引擎的优势来。因此，想要用好Spark，就必须对其进行合理的性能优化。

Spark的性能调优实际上是由很多部分组成的，不是调节几个参数就可以立竿见影提升作业性能的。我们需要根据不同的业务场景以及数据情况，对Spark作业进行综合性的分析，然后进行多个方面的调节和优化，才能获得最佳性能。

笔者根据之前的Spark作业开发经验以及实践积累，总结出了一套Spark作业的性能优化方案。整套方案主要分为开发调优、资源调优、数据倾斜调优、shuffle调优几个部分。开发调优和资源调优是所有Spark作业都需要注意和遵循的一些基本原则，是高性能Spark作业的基础；数据倾斜调优，主要讲解了一套完整的用来解决Spark作业数据倾斜的解决方案；shuffle调优，面向的是对Spark的原理有较深层次掌握和研究的同学，主要讲解了如何对Spark作业的shuffle运行过程以及细节进行调优。

本文作为Spark性能优化指南的基础篇，主要讲解开发调优以及资源调优。

2 开发调优

2.1 调优概述

Spark性能优化的第一步，就是要在开发Spark作业的过程中注意和应用一些性能优化的基本原则。开发调优，就是要让大家了解以下一些Spark基本开发原则，包括：RDD lineage设计、算子的合理使用、特殊操作的优化等。在开发过程中，时时刻刻都应该注意以上原则，并将这些原则根据具体的业务以及实际的应用场景，灵活地运用到自己的Spark作业中。

2.2 原则一：避免创建重复的RDD

通常来说，我们在开发一个Spark作业时，首先是基于某个数据源（比如Hive表或HDFS文件）创建一个初始的RDD；接着对这个RDD执行某个算子操作，然后得到下一个RDD；以此类推，循环往复，直到计算出最终我们需要的结果。在这个过程中，多个RDD会通过不同的算子操作（比如map、reduce等）串起来，这个“RDD串”，就是RDD lineage，也就是“RDD的血缘关系链”。

我们在开发过程中要注意：对于同一份数据，只应该创建一个RDD，不能创建多个RDD来代表同一份数据。

一些Spark初学者在刚开始开发Spark作业时，或者是有经验的工程师在开发RDD lineage极其冗长的Spark作业时，可能会忘了自己之前对于某一份数据已经创建过一个RDD了，从而导致对于同一份数据，创建了多个RDD。这就意味着，我们的Spark作业会进行多次重复计算来创建多个代表相同数据的RDD，进而增加了作业的性能开销。

一个简单的例子

```
//需要对名为"hello.txt"的HDFS文件进行一次map操作，再进行一次reduce操作。也就是说，需要对一份数据执行两次算子操作。
//错误的做法：对于同一份数据执行多次算子操作时，创建多个RDD。
//这里执行了两次textFile方法，针对同一个HDFS文件，创建了两个RDD出来，然后分别对每个RDD都执行了一个算子操作。
//这种情况下，Spark需要从HDFS上两次加载hello.txt文件的内容，并创建两个单独的RDD；
//第二次加载HDFS文件以及创建RDD的性能开销，很明显是白白浪费掉的。
val rdd1 = sc.textFile("hdfs://192.168.0.1:9000/hello.txt")
rdd1.map(...)
val rdd2 = sc.textFile("hdfs://192.168.0.1:9000/hello.txt")
rdd2.reduce(...)

//正确的用法：对于一份数据执行多次算子操作时，只使用一个RDD。
//这种写法很明显比上一种写法要好多了，因为我们对于同一份数据只创建了一个RDD，然后对这一个RDD执行了多次算子操作。
//但是要注意到这里为止优化还没有结束，由于rdd1被执行了两次算子操作，第二次执行reduce操作的时候，
//还会再次从源头处重新计算一次rdd1的数据，因此还是会有重复计算的性能开销。
//要彻底解决这个问题，必须结合“原则三：对多次使用的RDD进行持久化”，才能保证一个RDD被多次使用时只被计算一次。
val rdd1 = sc.textFile("hdfs://192.168.0.1:9000/hello.txt")
rdd1.map(...)
rdd1.reduce(...)
```

2.3 原则二：尽可能复用同一个RDD

除了要避免在开发过程中对一份完全相同的数据创建多个RDD之外，在对不同的数据执行算子操作时还要尽可能地复用一个RDD。比如说，有一个RDD的数据格式是key-value类型的，另一个是单value类型的，这两个RDD的value数据是完全一样的。那么此时我们可以只

使用key-value类型的那个RDD，因为其中已经包含了另一个的数据。对于类似这种多个RDD的数据有重叠或者包含的情况，我们应该尽量复用一个RDD，这样可以尽可能地减少RDD的数量，从而尽可能减少算子执行的次数。

一个简单的例子

```
//错误的做法。

//有一个<Long, String>格式的RDD，即rdd1。
//接着由于业务需要，对rdd1执行了一个map操作，创建了一个rdd2，而rdd2中的数据仅仅是rdd1中的value值而已，
//也就是说，rdd2是rdd1的子集。
JavaPairRDD<Long, String> rdd1 = ...
JavaRDD<String> rdd2 = rdd1.map(...)

// 分别对rdd1和rdd2执行了不同的算子操作。
rdd1.reduceByKey(...)
rdd2.map(...)

//正确的做法。

//上面这个case中，其实rdd1和rdd2的区别无非就是数据格式不同而已，rdd2的数据完全就是rdd1的子集而已，
//却创建了两个rdd，并对两个rdd都执行了一次算子操作。
//此时会因为对rdd1执行map算子来创建rdd2，而多执行一次算子操作，进而增加性能开销。

//其实在这种情况下完全可以复用同一个RDD。
//我们可以使用rdd1，既做reduceByKey操作，也做map操作。
//在进行第二个map操作时，只使用每个数据的tuple._2，也就是rdd1中的value值，即可。
JavaPairRDD<Long, String> rdd1 = ...
rdd1.reduceByKey(...)
rdd1.map(tuple._2...)

//第二种方式相较于第一种方式而言，很明显减少了一次rdd2的计算开销。
//但是到这里为止，优化还没有结束，对rdd1我们还是执行了两次算子操作，rdd1实际上还是会计算两次。
//因此还需要配合“原则三：对多次使用的RDD进行持久化”进行使用，才能保证一个RDD被多次使用时只被计算一次。
```

2.4 原则三：对多次使用的RDD进行持久化

当你在Spark代码中多次对一个RDD做了算子操作后，恭喜，你已经实现Spark作业第一步的优化了，也就是尽可能复用RDD。此时就该在这个基础之上，进行第二步优化了，也就是要保证对一个RDD执行多次算子操作时，这个RDD本身仅仅被计算一次。

Spark中对于一个RDD执行多次算子的默认原理是这样的：每次你对一个RDD执行一个算子操作时，都会重新从源头处计算一遍，计算出那个RDD来，然后再对这个RDD执行你的算子操作。这种方式的性能是很差的。

因此对于这种情况，我们的建议是：对多次使用的RDD进行持久化。此时Spark就会根据你的持久化策略，将RDD中的数据保存到内存或者磁盘中。以后每次对这个RDD进行算子操作时，都会直接从内存或磁盘中提取持久化的RDD数据，然后执行算子，而不会从源头处重新计算一遍这个RDD，再执行算子操作。

```
//如果要对一个RDD进行持久化，只要对这个RDD调用cache()和persist()即可。  
  
//正确的做法。  
//cache()方法表示：使用非序列化的方式将RDD中的数据全部尝试持久化到内存中。  
//此时再对rdd1执行两次算子操作时，只有在第一次执行map算子时，才会将这个rdd1从源头处计算一次。  
//第二次执行reduce算子时，就会直接从内存中提取数据进行计算，不会重复计算一个rdd。  
val rdd1 = sc.textFile("hdfs://192.168.0.1:9000/hello.txt").cache()  
rdd1.map(...)  
rdd1.reduce(...)  
  
//persist()方法表示：手动选择持久化级别，并使用指定的方式进行持久化。  
//比如说，StorageLevel.MEMORY_AND_DISK_SER表示，内存充足时优先持久化到内存中，内存不充足时持久化到磁盘文件中。  
//而且其中的_SER后缀表示，使用序列化的方式来保存RDD数据，此时RDD中的每个partition都会序列化成一个大的字节数组，  
//然后再持久化到内存或磁盘中。  
// 序列化的方式可以减少持久化的数据对内存/磁盘的占用量，进而避免内存被持久化数据占用过多，从而发生频繁GC。  
val rdd1 = sc.textFile("hdfs://192.168.0.1:9000/hello.txt").persist(StorageLevel.MEMORY_AND_DISK_SER)  
rdd1.map(...)  
rdd1.reduce(...)
```

对于persist()方法而言，我们可以根据不同的业务场景选择不同的持久化级别。

Spark的持久化级别

持久化级别	含义解释
MEMORY_ONLY	使用未序列化的Java对象格式，将数据保存在内存中。如果内存不够存放所有的数据，则数据可能就不会进行持久化。那么下次对这个RDD执行算子操作时，那些没有被持久化的数据，需要从源头处重新计算一遍。这是默认的持久化策略，使用cache()方法时，实际就是使用的这种持久化策略。
MEMORY_AND_DISK	使用未序列化的Java对象格式，优先尝试将数据保存在内存中。如果内存不够存放所有的数据，会将数据写入磁盘文件中，下次对这个RDD执行算子时，持久化在磁盘文件中的数据会被读取出来使用。
MEMORY_ONLY_SER	基本含义同MEMORY_ONLY。唯一的区别是，会将RDD中的数据进行序列化，RDD的每个partition会被序列化成一个字节数组。这种方式更加节省内存，从而可以避免持久化的数据占用过多内存导致频繁GC。
MEMORY_AND_DISK_SER	基本含义同MEMORY_AND_DISK。唯一的区别是，会将RDD中的数据进行序列化，RDD的每个partition会被序列化成一个字节数组。这种方式更加节省内存，从而可以避免持久化的数据占用过多内存导致频繁GC。
DISK_ONLY	使用未序列化的Java对象格式，将数据全部写入磁盘文件中。
MEMORY_ONLY_2, MEMORY_AND_DISK_2	对于上述任意一种持久化策略，如果加上后缀_2，代表的是将每个持久化的数据，都复制一份副本，并将副本保存到其他节点上。这种基于副本的持久化机制主要用于进行容错。假如某个节点挂掉，节点的内存或磁盘中的持久化数据丢失了，那么后续对RDD计算时还可以使用该数据在其他节点上的副本。如果没有副本的话，就只能将这些数据从源头处重新计算一遍了

如何选择一种最合适的持久化策略

- 默认情况下，性能最高的当然是MEMORY_ONLY，但前提是你的内存必须足够足够大，可以绰绰有余地存放下整个RDD的所有数据。因为不进行序列化与反序列化操作，就避免了这部分的性能开销；对这个RDD的后续算子操作，都是基于纯内存中的数据的操作，不需要从磁盘文件中读取数据，性能也很高；而且不需要复制一份数据副本，并远程传送到其他节点上。但是这里必须要注意的是，在实际的生产环境中，恐怕能够直接用这种策略的场景还是有限的，如果RDD中数据比较多时（比如几十亿），直接用这种持久化级别，会导致JVM的OOM内存溢出异常。
- 如果使用MEMORY_ONLY级别时发生了内存溢出，那么建议尝试使用MEMORY_ONLY_SER级别。该级别会将RDD数据序列化后再保存在内存中，此时每个partition仅仅是一个字节数组而已，大大减少了对象数量，并降低了内存占用。这种级别

比MEMORY_ONLY多出来的性能开销，主要就是序列化与反序列化的开销。但是后续算子可以基于纯内存进行操作，因此性能总体还是比较高的。此外，可能发生的问题同上，如果RDD中的数据量过多的话，还是可能会导致OOM内存溢出的异常。

- 如果纯内存的级别都无法使用，那么建议使用MEMORY_AND_DISK_SER策略，而不是MEMORY_AND_DISK策略。因为既然到了这一步，就说明RDD的数据量很大，内存无法完全放下。序列化后的数据比较少，可以节省内存和磁盘的空间开销。同时该策略会优先尽量尝试将数据缓存在内存中，内存缓存不下才会写入磁盘。
- 通常不建议使用DISK_ONLY和后缀为_2的级别：因为完全基于磁盘文件进行数据的读写，会导致性能急剧降低，有时还不如重新计算一次所有RDD。后缀为_2的级别，必须将所有数据都复制一份副本，并发送到其他节点上，数据复制以及网络传输会导致较大的性能开销，除非是要求作业的高可用性，否则不建议使用。

2.5 原则四：尽量避免使用shuffle类算子

如果有可能的话，要尽量避免使用shuffle类算子。因为Spark作业运行过程中，最消耗性能的地方就是shuffle过程。shuffle过程，简单来说，就是将分布在集群中多个节点上的同一个key，拉取到同一个节点上，进行聚合或join等操作。比如reduceByKey、join等算子，都会触发shuffle操作。

shuffle过程中，各个节点上的相同key都会先写入本地磁盘文件中，然后其他节点需要通过网络传输拉取各个节点上的磁盘文件中的相同key。而且相同key都拉取到同一个节点进行聚合操作时，还有可能会因为一个节点上处理的key过多，导致内存不够存放，进而溢写到磁盘文件中。因此在shuffle过程中，可能会发生大量的磁盘文件读写的IO操作，以及数据的网络传输操作。磁盘IO和网络数据传输也是shuffle性能较差的主要原因。

因此在我们的开发过程中，能避免则尽可能避免使用reduceByKey、join、distinct、repartition等会进行shuffle的算子，尽量使用map类的非shuffle算子。这样的话，没有shuffle操作或者仅有较少shuffle操作的Spark作业，可以大大减少性能开销。

Broadcast与map进行join代码示例

```

//传统的join操作会导致shuffle操作。
//因为两个RDD中，相同的key都需要通过网络拉取到一个节点上，由一个task进行join操作。
val rdd3 = rdd1.join(rdd2)

//Broadcast+map的join操作，不会导致shuffle操作。
//使用Broadcast将一个数据量较小的RDD作为广播变量。
val rdd2Data = rdd2.collect()
val rdd2DataBroadcast = sc.broadcast(rdd2Data)

//在rdd1.map算子中，可以从rdd2DataBroadcast中，获取rdd2的所有数据。
//然后进行遍历，如果发现rdd2中某条数据的key与rdd1的当前数据的key是相同的，那么就判定可以进行join。
//此时就可以根据自己需要的方式，将rdd1当前数据与rdd2中可以连接的数据，拼接在一起（String或Tuple）。
val rdd3 = rdd1.map(rdd2DataBroadcast...)

//注意，以上操作，建议仅仅在rdd2的数据量比较少（比如几百M，或者一两G）的情况下使用。
//因为每个Executor的内存中，都会驻留一份rdd2的全量数据。

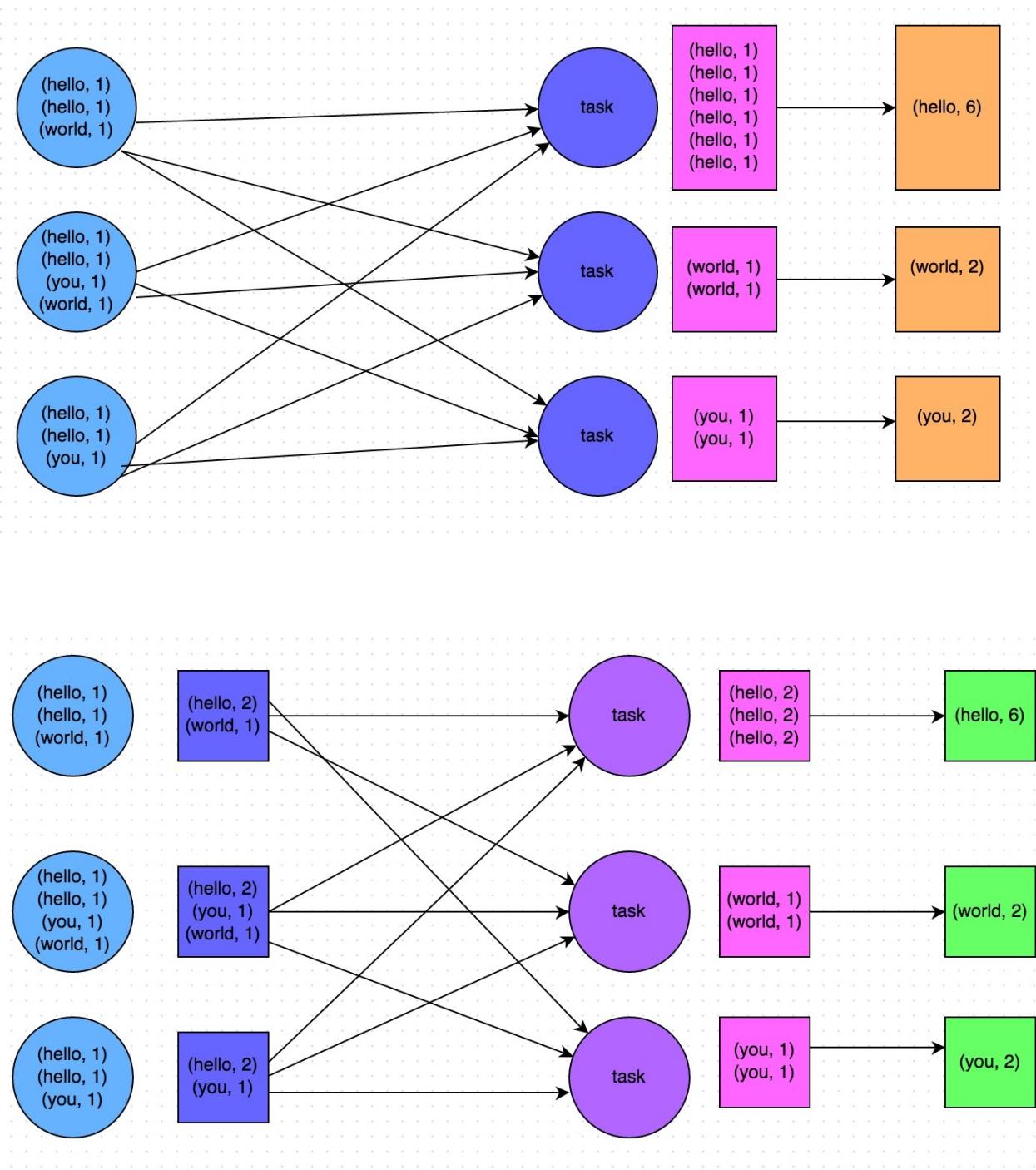
```

2.6 原则五：使用**map-side**预聚合的**shuffle**操作

如果因为业务需要，一定要使用**shuffle**操作，无法用**map**类的算子来替代，那么尽量使用可以**map-side**预聚合的算子。

所谓的**map-side**预聚合，说的是在每个节点本地对相同的**key**进行一次聚合操作，类似于**MapReduce**中的本地**combiner**。**map-side**预聚合之后，每个节点本地就只会有一条相同的**key**，因为多条相同的**key**都被聚合起来了。其他节点在拉取所有节点上的相同**key**时，就会大大减少需要拉取的数据数量，从而也就减少了磁盘IO以及网络传输开销。通常来说，在可能的情况下，建议使用**reduceByKey**或者**aggregateByKey**算子来替代掉**groupByKey**算子。因为**reduceByKey**和**aggregateByKey**算子都会使用用户自定义的函数对每个节点本地的相同**key**进行预聚合。而**groupByKey**算子是不会进行预聚合的，全量的数据会在集群的各个节点之间分发和传输，性能相对来说比较差。

比如如下两幅图，就是典型的例子，分别基于**reduceByKey**和**groupByKey**进行单词计数。其中第一张图是**groupByKey**的原理图，可以看到，没有进行任何本地聚合时，所有数据都会在集群节点之间传输；第二张图是**reduceByKey**的原理图，可以看到，每个节点本地的相同**key**数据，都进行了预聚合，然后才传输到其他节点上进行全局聚合。



2.7 原则六：使用高性能的算子

除了shuffle相关的算子有优化原则之外，其他的算子也都有着相应的优化原则。

使用**reduceByKey/aggregateByKey**替代**groupByKey**

使用**mapPartitions**替代普通**map**

mapPartitions类的算子，一次函数调用会处理一个partition所有的数据，而不是一次函数调用处理一条，性能相对来说会高一些。但是有的时候，使用**mapPartitions**会出现OOM（内存溢出）的问题。因为单次函数调用就要处理掉一个partition所有的数据，如果内存不够，垃圾回收时是无法回收掉太多对象的，很可能出现OOM异常。所以使用这类操作时要慎重！

使用**foreachPartitions**替代**foreach**

原理类似于“使用**mapPartitions**替代**map**”，也是一次函数调用处理一个partition的所有数据，而不是一次函数调用处理一条数据。在实践中发现，**foreachPartitions**类的算子，对性能的提升还是很有帮助的。比如在**foreach**函数中，将RDD中所有数据写MySQL，那么如果是普通的**foreach**算子，就会一条数据一条数据地写，每次函数调用可能就会创建一个数据库连接，此时就势必会频繁地创建和销毁数据库连接，性能是非常低下；但是如果用**foreachPartitions**算子一次性处理一个partition的数据，那么对于每个partition，只要创建一个数据库连接即可，然后执行批量插入操作，此时性能是比较高的。实践中发现，对于1万条左右的数据量写MySQL，性能可以提升30%以上。

使用**filter**之后进行**coalesce**操作

通常对一个RDD执行**filter**算子过滤掉RDD中较多数据后（比如30%以上的数据），建议使用**coalesce**算子，手动减少RDD的partition数量，将RDD中的数据压缩到更少的partition中去。因为**filter**之后，RDD的每个partition中都会有很多数据被过滤掉，此时如果照常进行后续的计算，其实每个task处理的partition中的数据量并不是很多，有一点资源浪费，而且此时处理的task越多，可能速度反而越慢。因此用**coalesce**减少partition数量，将RDD中的数据压缩到更少的partition之后，只要使用更少的task即可处理完所有的partition。在某些场景下，对于性能的提升会有一定的帮助。

使用**repartitionAndSortWithinPartitions**替代**repartition**与**sort**类操作

repartitionAndSortWithinPartitions是Spark官网推荐的一个算子，官方建议，如果需要在**repartition**重分区之后，还要进行排序，建议直接使用**repartitionAndSortWithinPartitions**算子。因为该算子可以一边进行重分区的**shuffle**操作，一边进行排序。**shuffle**与**sort**两个操作同时进行，比先**shuffle**再**sort**来说，性能可能是要高的。

2.8 原则七：广播大变量

有时在开发过程中，会遇到需要在算子函数中使用外部变量的场景（尤其是大变量，比如100M以上的大集合），那么此时就应该使用Spark的广播（Broadcast）功能来提升性能。

在算子函数中使用到外部变量时，默认情况下，Spark会将该变量复制多个副本，通过网络传输到task中，此时每个task都有一个变量副本。如果变量本身比较大的话（比如100M，甚至1G），那么大量的变量副本在网络中传输的性能开销，以及在各个节点的Executor中占用过多内存导致的频繁GC，都会极大地影响性能。

因此对于上述情况，如果使用的外部变量比较大，建议使用Spark的广播功能，对该变量进行广播。广播后的变量，会保证每个Executor的内存中，只驻留一份变量副本，而Executor中的task执行时共享该Executor中的那份变量副本。这样的话，可以大大减少变量副本的数量，从而减少网络传输的性能开销，并减少对Executor内存的占用开销，降低GC的频率。

广播大变量的代码示例

```
//以下代码在算子函数中，使用了外部的变量。
//此时没有做任何特殊操作，每个task都会有一份list1的副本。
val list1 = ...
rdd1.map(list1...)

//以下代码将list1封装成了Broadcast类型的广播变量。
//在算子函数中，使用广播变量时，首先会判断当前task所在Executor内存中，是否有变量副本。
//如果有则直接使用；如果没有则从Driver或者其他Executor节点上远程拉取一份放到本地Executor内存中。
//每个Executor内存中，就只会驻留一份广播变量副本。
val list1 = ...
val list1Broadcast = sc.broadcast(list1)
rdd1.map(list1Broadcast...)
```

2.9 原则八：使用Kryo优化序列化性能

在Spark中，主要有三个地方涉及到了序列化：

- 在算子函数中使用到外部变量时，该变量会被序列化后进行网络传输（见“原则七：广播大变量”中的讲解）。
- 将自定义的类型作为RDD的泛型类型时（比如JavaRDD，Student是自定义类型），所有自定义类型对象，都会进行序列化。因此这种情况下，也要求自定义的类必须实现Serializable接口。
- 使用可序列化的持久化策略时（比如MEMORY_ONLY_SER），Spark会将RDD中的每个partition都序列化成一个大的字节数组。

对于这三种出现序列化的地方，我们都可以通过使用Kryo序列化类库，来优化序列化和反序列化的性能。Spark默认使用的是Java的序列化机制，也就是ObjectOutputStream/ObjectInputStream API来进行序列化和反序列化。但是Spark同时支持使用Kryo序列化库，Kryo序列化类库的性能比Java序列化类库的性能要高很多。官方介绍，Kryo序列化机制比Java序列化机制，性能高10倍左右。Spark之所以默认没有使用Kryo作为序列化类库，是因为Kryo要求最好要注册所有需要进行序列化的自定义类型，因此对于开发者来说，这种方式比较麻烦。

以下是使用Kryo的代码示例，我们只要设置序列化类，再注册要序列化的自定义类型即可（比如算子函数中使用到的外部变量类型、作为RDD泛型类型的自定义类型等）：

```
//创建SparkConf对象。
val conf = new SparkConf().setMaster(...).setAppName(...)
//设置序列化器为KryoSerializer。
conf.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer")
//注册要序列化的自定义类型。
conf.registerKryoClasses(Array(classOf[MyClass1], classOf[MyClass2]))
```

2.10 原则九：优化数据结构

Java中，有三种类型比较耗费内存：

- 对象，每个Java对象都有对象头、引用等额外的信息，因此比较占用内存空间。
- 字符串，每个字符串内部都有一个字符数组以及长度等额外信息。
- 集合类型，比如HashMap、LinkedList等，因为集合类型内部通常会使用一些内部类来封装集合元素，比如Map.Entry。

因此Spark官方建议，在Spark编码实现中，特别是对于算子函数中的代码，尽量不要使用上述三种数据结构，尽量使用字符串替代对象，使用原始类型（比如Int、Long）替代字符串，使用数组替代集合类型，这样尽可能地减少内存占用，从而降低GC频率，提升性能。

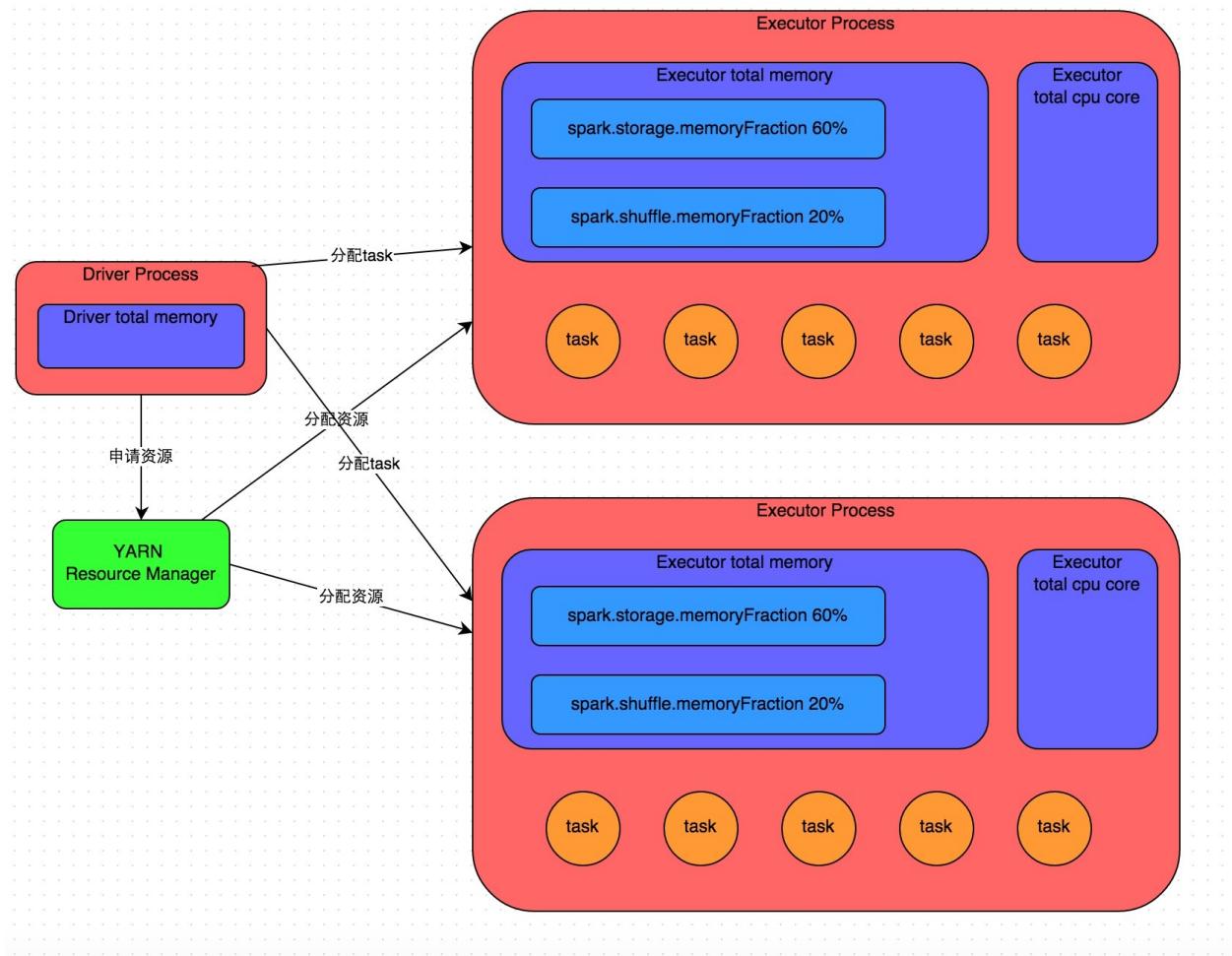
但是在笔者的编码实践中发现，要做到该原则其实不容易。因为我们同时要考虑到代码的可维护性，如果一个代码中，完全没有任何对象抽象，全部是字符串拼接的方式，那么对于后续的代码维护和修改，无疑是一场巨大的灾难。同理，如果所有操作都基于数组实现，而不使用HashMap、LinkedList等集合类型，那么对于我们的编码难度以及代码可维护性，也是一个极大的挑战。因此笔者建议，在可能以及合适的情况下，使用占用内存较少的数据结构，但是前提是保证代码的可维护性。

3 资源调优

3.1 调优概述

在开发完Spark作业之后，就该为作业配置合适的资源了。Spark的资源参数，基本都可以在spark-submit命令中作为参数设置。很多Spark初学者，通常不知道该设置哪些必要的参数，以及如何设置这些参数，最后就只能胡乱设置，甚至压根儿不设置。资源参数设置的不合理，可能会导致没有充分利用集群资源，作业运行会极其缓慢；或者设置的资源过大，队列没有足够的资源来提供，进而导致各种异常。总之，无论是哪种情况，都会导致Spark作业的运行效率低下，甚至根本无法运行。因此我们必须对Spark作业的资源使用原理有一个清晰的认识，并知道在Spark作业运行过程中，有哪些资源参数是可以设置的，以及如何设置合适的参数值。

3.2 Spark作业基本运行原理



详细原理见上图。我们使用 `spark-submit` 提交一个 Spark 作业之后，这个作业就会启动一个对应的 Driver 进程。根据你使用的部署模式（deploy-mode）不同，Driver 进程可能在本地启动，也可能在集群中某个工作节点上启动。Driver 进程本身会根据我们设置的参数，占有一定数量的内存和 CPU core。而 Driver 进程要做的第一件事情，就是向集群管理器（可以是 Spark Standalone 集群，也可以是其他的资源管理集群）申请运行 Spark 作业需要使用的资源，这里的资源指的就是 Executor 进程。YARN 集群管理器会根据我们为 Spark 作业设置的资源参数，在各个工作节点上，启动一定数量的 Executor 进程，每个 Executor 进程都占有一定数量的内存和 CPU core。

在申请到了作业执行所需的资源之后，Driver 进程就会开始调度和执行我们编写的作业代码了。Driver 进程会将我们编写的 Spark 作业代码拆分为多个 stage，每个 stage 执行一部分代码片段，并为每个 stage 创建一批 task，然后将这些 task 分配到各个 Executor 进程中执行。task 是最小的计算单元，负责执行一模一样的计算逻辑（也就是我们自己编写的某个代码片段），只是每个 task 处理的数据不同而已。一个 stage 的所有 task 都执行完毕之后，会在各个

节点本地的磁盘文件中写入计算中间结果，然后Driver就会调度运行下一个stage。下一个stage的task的输入数据就是上一个stage输出的中间结果。如此循环往复，直到将我们自己编写的代码逻辑全部执行完，并且计算完所有的数据，得到我们想要的结果为止。

Spark是根据shuffle类算子来进行stage的划分。如果我们的代码中执行了某个shuffle类算子（比如reduceByKey、join等），那么就会在该算子处，划分出一个stage界限来。可以大致理解为，shuffle算子执行之前的代码会被划分为一个stage，shuffle算子执行以及之后的代码会被划分为下一个stage。因此一个stage刚开始执行的时候，它的每个task可能都会从上一个stage的task所在的节点，去通过网络传输拉取需要自己处理的所有key，然后对拉取到的所有相同的key使用我们自己编写的算子函数执行聚合操作（比如reduceByKey()算子接收的函数）。这个过程就是shuffle。

当我们在代码中执行了cache/persist等持久化操作时，根据我们选择的持久化级别的不同，每个task计算出来的数据也会保存到Executor进程的内存或者所在节点的磁盘文件中。

因此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%。

task的执行速度是跟每个Executor进程的CPU core数量有直接关系的。一个CPU core同一时间只能执行一个线程。而每个Executor进程上分配到的多个task，都是以每个task一条线程的方式，多线程并发运行的。如果CPU core数量比较充足，而且分配到的task数量比较合理，那么通常来说，可以比较快速和高效地执行完这些task线程。

以上就是Spark作业的基本运行原理的说明，大家可以结合上图来理解。理解作业基本原理，是我们进行资源参数调优的基本前提。

3.3 资源参数调优

了解完了Spark作业运行的基本原理之后，对资源相关的参数就容易理解了。所谓的Spark资源参数调优，其实主要就是对Spark运行过程中各个使用资源的地方，通过调节各种参数，来优化资源使用的效率，从而提升Spark作业的执行性能。以下参数就是Spark中主要的资源参数，每个参数都对应着作业运行原理中的某个部分，我们同时也给出了一个调优的参考值。

num-executors

- 参数说明：该参数用于设置Spark作业总共要用多少个Executor进程来执行。Driver在向YARN集群管理器申请资源时，YARN集群管理器会尽可能按照你的设置来在集群的各个工作节点上，启动相应数量的Executor进程。这个参数非常之重要，如果不设置的话，默认只会给你启动少量的Executor进程，此时你的Spark作业的运行速度是非常慢的。

- 参数调优建议：每个Spark作业的运行一般设置50-100个左右的Executor进程比较合适，设置太少或太多的Executor进程都不好。设置的太少，无法充分利用集群资源；设置的太多的话，大部分队列可能无法给予充分的资源。

executor-memory

- 参数说明：该参数用于设置每个Executor进程的内存。Executor内存的大小，很多时候直接决定了Spark作业的性能，而且跟常见的JVM OOM异常，也有直接的关联。
- 参数调优建议：每个Executor进程的内存设置4G-8G较为合适。但是这只是一个参考值，具体的设置还是得根据不同部门的资源队列来定。可以看看自己团队的资源队列的最大内存限制是多少，`num-executors`乘以`executor-memory`，就代表了你的Spark作业申请到的总内存量（也就是所有Executor进程的内存总和），这个量是不能超过队列的最大内存量的。此外，如果你是跟团队里其他人共享这个资源队列，那么申请的总内存量最好不要超过资源队列最大总内存的1/3~1/2，避免你自己的Spark作业占用了队列所有的资源，导致别的同学的作业无法运行。

executor-cores

- 参数说明：该参数用于设置每个Executor进程的CPU core数量。这个参数决定了每个Executor进程并行执行task线程的能力。因为每个CPU core同一时间只能执行一个task线程，因此每个Executor进程的CPU core数量越多，越能够快速地执行完分配给自己的所有task线程。
- 参数调优建议：Executor的CPU core数量设置为2-4个较为合适。同样得根据不同部门的资源队列来定，可以看看自己的资源队列的最大CPU core限制是多少，再依据设置的Executor数量，来决定每个Executor进程可以分配到几个CPU core。同样建议，如果是跟他人共享这个队列，那么`num-executors * executor-cores`不要超过队列总CPU core的1/3~1/2左右比较合适，也是避免影响其他同学的作业运行。

driver-memory

- 参数说明：该参数用于设置Driver进程的内存。
- 参数调优建议：Driver的内存通常来说不设置，或者设置1G左右应该就够了。唯一需要注意的一点是，如果需要使用`collect`算子将RDD的数据全部拉取到Driver上进行处理，那么必须确保Driver的内存足够大，否则会出现OOM内存溢出的问题。

spark.default.parallelism

- 参数说明：该参数用于设置每个stage的默认task数量。这个参数极为重要，如果不设置可能会直接影响你的Spark作业性能。

- 参数调优建议：Spark作业的默认task数量为500-1000个较为合适。很多同学常犯的一个错误就是不去设置这个参数，那么此时就会导致Spark自己根据底层HDFS的block数量来设置task的数量，默认是一个HDFS block对应一个task。通常来说，Spark默认设置的数量是偏少的（比如就几十个task），如果task数量偏少的话，就会导致你前面设置好的Executor的参数都前功尽弃。试想一下，无论你的Executor进程有多少个，内存和CPU有多大，但是task只有1个或者10个，那么90%的Executor进程可能根本就没有task执行，也就是白白浪费了资源！因此Spark官网建议的设置原则是，设置该参数为num-executors * executor-cores的2~3倍较为合适，比如Executor的总CPU core数量为300个，那么设置1000个task是可以的，此时可以充分地利用Spark集群的资源。

spark.storage.memoryFraction

- 参数说明：该参数用于设置RDD持久化数据在Executor内存中能占的比例，默认是0.6。也就是说，默认Executor 60%的内存，可以用来保存持久化的RDD数据。根据你选择的不同的持久化策略，如果内存不够时，可能数据就不会持久化，或者数据会写入磁盘。
- 参数调优建议：如果Spark作业中，有较多的RDD持久化操作，该参数的值可以适当提高一些，保证持久化的数据能够容纳在内存中。避免内存不够缓存所有的数据，导致数据只能写入磁盘中，降低了性能。但是如果Spark作业中的shuffle类操作比较多，而持久化操作比较少，那么这个参数的值适当降低一些比较合适。此外，如果发现作业由于频繁的gc导致运行缓慢（通过spark web ui可以观察到作业的gc耗时），意味着task执行用户代码的内存不够用，那么同样建议调低这个参数的值。

spark.shuffle.memoryFraction

- 参数说明：该参数用于设置shuffle过程中一个task拉取到上个stage的task的输出后，进行聚合操作时能够使用的Executor内存的比例，默认是0.2。也就是说，Executor默认只有20%的内存用来进行该操作。shuffle操作在进行聚合时，如果发现使用的内存超出了这个20%的限制，那么多余的数据就会溢写到磁盘文件中去，此时就会极大地降低性能。
- 参数调优建议：如果Spark作业中的RDD持久化操作较少，shuffle操作较多时，建议降低持久化操作的内存占比，提高shuffle操作的内存占比比例，避免shuffle过程中数据过多时内存不够用，必须溢写到磁盘上，降低了性能。此外，如果发现作业由于频繁的gc导致运行缓慢，意味着task执行用户代码的内存不够用，那么同样建议调低这个参数的值。

资源参数的调优，没有一个固定的值，需要同学们根据自己的实际情况（包括Spark作业中的shuffle操作数量、RDD持久化操作数量以及spark web ui中显示的作业gc情况），同时参考本篇文章中给出的原理以及调优建议，合理地设置上述参数。

引用文档

Spark性能优化指南——基础篇

Spark性能优化指南——高级篇

1 数据倾斜调优

1.1 调优概述

有的时候，我们可能会遇到大数据计算中一个最棘手的问题——数据倾斜，此时Spark作业的性能会比期望差很多。数据倾斜调优，就是使用各种技术方案解决不同类型的数据倾斜问题，以保证Spark作业的性能。

2.2 数据倾斜发生时的现象

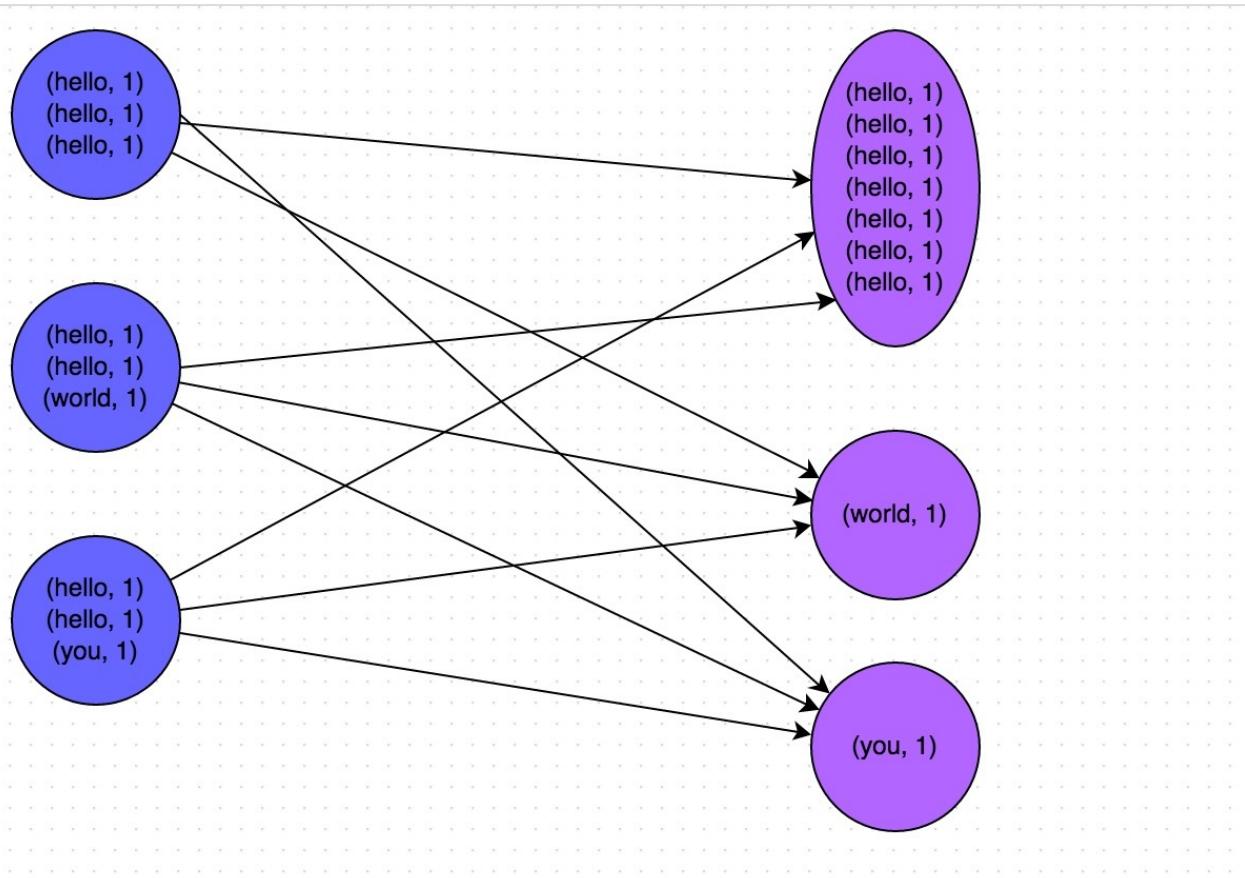
- 绝大多数task执行得都非常快，但个别task执行极慢。比如，总共有1000个task，997个task都在1分钟之内执行完了，但是剩余两三个task却要一两个小时。这种情况很常见。
- 原本能够正常执行的Spark作业，某天突然报出OOM（内存溢出）异常，观察异常栈，是我们写的业务代码造成的。这种情况比较少见。

2.3 数据倾斜发生的原理

数据倾斜的原理很简单：在进行shuffle的时候，必须将各个节点上相同的key拉取到某个节点上的一个task来进行处理，比如按照key进行聚合或join等操作。此时如果某个key对应的数据量特别大的话，就会发生数据倾斜。比如大部分key对应10条数据，但是个别key却对应了100万条数据，那么大部分task可能就只会分配到10条数据，然后1秒钟就运行完了；但是个别task可能分配到了100万数据，要运行一两个小时。因此，整个Spark作业的运行进度是由运行时间最长的那个task决定的。

因此出现数据倾斜的时候，Spark作业看起来会运行得非常缓慢，甚至可能因为某个task处理的数据量过大导致内存溢出。

下图就是一个很清晰的例子：hello这个key，在三个节点上对应了总共7条数据，这些数据都会被拉取到同一个task中进行处理；而world和you这两个key分别才对应1条数据，所以另外两个task只要分别处理1条数据即可。此时第一个task的运行时间可能是另外两个task的7倍，而整个stage的运行速度也由运行最慢的那个task所决定。



2.4 如何定位导致数据倾斜的代码

数据倾斜只会发生在shuffle过程中。这里给大家罗列一些常用的并且可能会触发shuffle操作的算子：distinct、groupByKey、reduceByKey、aggregateByKey、join、cogroup、repartition等。出现数据倾斜时，可能就是你的代码中使用了这些算子中的某一个所导致的。

某个task执行特别慢的情况

首先要看的，就是数据倾斜发生在第几个stage中。

如果是用yarn-client模式提交，那么本地是直接可以看到log的，可以在log中找到当前运行到了第几个stage；如果是用yarn-cluster模式提交，则可以通过Spark Web UI来查看当前运行到了第几个stage。此外，无论是使用yarn-client模式还是yarn-cluster模式，我们都可以在Spark Web UI上深入看一下当前这个stage各个task分配的数据量，从而进一步确定是不是task分配的数据不均匀导致了数据倾斜。

比如下图中，倒数第三列显示了每个task的运行时间。明显可以看到，有的task运行特别快，只需要几秒钟就可以运行完；而有的task运行特别慢，需要几分钟才能运行完，此时单从运行时间上看就已经能够确定发生数据倾斜了。此外，倒数第一列显示了每个task处理的数据

量，明显可以看到，运行时间特别短的task只需要处理几百KB的数据即可，而运行时间特别长的task需要处理几千KB的数据，处理的数据量差了10倍。此时更加能够确定是发生了数据倾斜。

85	154	0	SUCCESS	PROCESS_LOCAL	3 / rz-data-hdp-dn0912.rz.sankuai.com	2016/01/29 13:42:02	3.2 min	0.9 s	807.7 KB / 8691
86	155	0	SUCCESS	PROCESS_LOCAL	46 / rz-data-hdp-dn0890.rz.sankuai.com	2016/01/29 13:42:02	49 s	0.5 s	531.4 KB / 5309
87	156	0	SUCCESS	PROCESS_LOCAL	92 / rz-data-hdp-dn1275.rz.sankuai.com	2016/01/29 13:42:02	31 s	0.6 s	360.7 KB / 3696
88	157	0	SUCCESS	PROCESS_LOCAL	64 / rz-data-hdp-dn0121.rz.sankuai.com	2016/01/29 13:42:02	27 s	0.4 s	406.1 KB / 4104
89	158	0	SUCCESS	PROCESS_LOCAL	13 / rz-data-hdp-dn1184.rz.sankuai.com	2016/01/29 13:42:02	14 s	0.4 s	347.3 KB / 3561
90	159	0	SUCCESS	PROCESS_LOCAL	5 / rz-data-hdp-dn0912.rz.sankuai.com	2016/01/29 13:42:02	13 s	0.3 s	351.4 KB / 3622
91	160	0	RUNNING	PROCESS_LOCAL	90 / rz-data-hdp-dn0059.rz.sankuai.com	2016/01/29 13:42:02	3.9 min	0.8 s	1617.0 KB / 18545
92	161	0	SUCCESS	PROCESS_LOCAL	87 / rz-data-hdp-dn0879.rz.sankuai.com	2016/01/29 13:42:02	26 s	0.4 s	318.1 KB / 3081
93	162	0	SUCCESS	PROCESS_LOCAL	55 / rz-data-hdp-dn0875.rz.sankuai.com	2016/01/29 13:42:02	19 s	0.5 s	359.6 KB / 3574
94	163	0	RUNNING	PROCESS_LOCAL	82 / rz-data-hdp-dn0430.rz.sankuai.com	2016/01/29 13:42:02	3.9 min	0.8 s	2023.4 KB / 22812
95	164	0	SUCCESS	PROCESS_LOCAL	99 / rz-data-hdp-dn0817.rz.sankuai.com	2016/01/29 13:42:02	5 s	0.2 s	188.1 KB / 1426
96	165	0	SUCCESS	PROCESS_LOCAL	56 / rz-data-hdp-dn0875.rz.sankuai.com	2016/01/29 13:42:02	10 s	0.3 s	214.5 KB / 1683
97	166	0	SUCCESS	PROCESS_LOCAL	71 / rz-data-hdp-dn0576.rz.sankuai.com	2016/01/29 13:42:02	2.9 min	0.4 s	673.8 KB / 6932
98	167	0	SUCCESS	PROCESS_LOCAL	77 / rz-data-hdp-dn0242.rz.sankuai.com	2016/01/29 13:42:02	13 s	0.3 s	276.3 KB / 2349
99	168	0	RUNNING	PROCESS_LOCAL	58 / rz-data-hdp-dn0491.rz.sankuai.com	2016/01/29 13:42:02	3.9 min	1 s	1321.0 KB / 14508

知道数据倾斜发生在哪一个stage之后，接着我们就需要根据stage划分原理，推算出来发生倾斜的那个stage对应代码中的哪一部分，这部分代码中肯定会有一个shuffle类算子。精准推算stage与代码的对应关系，需要对Spark的源码有深入的理解，这里我们可以介绍一个相对简单实用的推算方法：只要看到Spark代码中出现了一个shuffle类算子或者是Spark SQL的SQL语句中出现了会导致shuffle的语句（比如group by语句），那么就可以判定，以那个地方为界限划分出了前后两个stage。

这里我们就以Spark最基础的入门程序——单词计数来举例，如何用最简单的方法大致推算出一个stage对应的代码。如下示例，在整个代码中，只有一个reduceByKey是会发生shuffle的算子，因此就可以认为，以这个算子为界限，会划分出前后两个stage。

- stage0，主要是执行从textFile到map操作，以及执行shuffle write操作。shuffle write操作，我们可以简单理解为对pairs RDD中的数据进行分区操作，每个task处理的数据中，相同的key会写入同一个磁盘文件内。
- stage1，主要是执行从reduceByKey到collect操作，stage1的各个task一开始运行，就会首先执行shuffle read操作。执行shuffle read操作的task，会从stage0的各个task所在节点拉取属于自己处理的那些key，然后对同一个key进行全局性的聚合或join等操作，在这里就是对key的value值进行累加。stage1在执行完reduceByKey算子之后，就计算出了最终的wordCounts RDD，然后会执行collect算子，将所有数据拉取到Driver上，供我们遍历和打印输出。

```

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

val lines = sc.textFile("hdfs://...")
val words = lines.flatMap(_.split(" "))
val pairs = words.map((_, 1))
val wordCounts = pairs.reduceByKey(_ + _)

wordCounts.collect().foreach(println(_))

```

通过对单词计数程序的分析，希望能够让大家了解最基本的stage划分的原理，以及stage划分后shuffle操作是如何在两个stage的边界处执行的。然后我们就知道如何快速定位出发生数据倾斜的stage对应代码的哪一个部分了。比如我们在Spark Web UI或者本地log中发现，stage1的某几个task执行得特别慢，判定stage1出现了数据倾斜，那么就可以回到代码中定位出stage1主要包括了reduceByKey这个shuffle类算子，此时基本就可以确定是由reduceByKey算子导致的数据倾斜问题。比如某个单词出现了100万次，其他单词才出现10次，那么stage1的某个task就要处理100万数据，整个stage的速度就会被这个task拖慢。

某个**task**莫名其妙内存溢出的情况

这种情况下去定位出问题的代码就比较容易了。我们建议直接看yarn-client模式下本地log的异常栈，或者是通过YARN查看yarn-cluster模式下的log中的异常栈。一般来说，通过异常栈信息就可以定位到你的代码中哪一行发生了内存溢出。然后在那行代码附近找找，一般也会有shuffle类算子，此时很可能就是这个算子导致了数据倾斜。

但是大家要注意的是，不能单纯靠偶然的内存溢出就判定发生了数据倾斜。因为自己编写的代码的bug，以及偶然出现的数据异常，也可能导致内存溢出。因此还是要按照上面所讲的方法，通过Spark Web UI查看报错的那个stage的各个task的运行时间以及分配的数据量，才能确定是否是由于数据倾斜才导致了这次内存溢出。

2.5 查看导致数据倾斜的**key**的数据分布情况

知道了数据倾斜发生在哪之后，通常需要分析一下那个执行了shuffle操作并且导致了数据倾斜的RDD/Hive表，查看一下其中key的分布情况。这主要是为之后选择哪一种技术方案提供依据。针对不同的key分布与不同的shuffle算子组合起来的各种情况，可能需要选择不同的技术方案来解决。

此时根据你执行操作的情况不同，可以有很多种查看key分布的方式：

- 如果是Spark SQL中的group by、join语句导致的数据倾斜，那么就查询一下SQL中使用的表的key分布情况。

- 如果是对Spark RDD执行shuffle算子导致的数据倾斜，那么可以在Spark作业中加入查看key分布的代码，比如RDD.countByKey()。然后对统计出来的各个key出现的次数，collect/take到客户端打印一下，就可以看到key的分布情况。

举例来说，对于上面所说的单词计数程序，如果确定了是stage1的reduceByKey算子导致了数据倾斜，那么就应该看看进行reduceByKey操作的RDD中的key分布情况，在这个例子中指的就是pairs RDD。如下示例，我们可以先对pairs采样10%的样本数据，然后使用countByKey算子统计出每个key出现的次数，最后在客户端遍历和打印样本数据中各个key的出现次数。

```
val sampledPairs = pairs.sample(false, 0.1)
val sampledWordCounts = sampledPairs.countByKey()
sampledWordCounts.foreach(println(_))
```

2.6 数据倾斜的解决方案

解决方案一：使用Hive ETL预处理数据

方案适用场景：导致数据倾斜的是Hive表。如果该Hive表中的数据本身很不均匀（比如某个key对应了100万数据，其他key才对应了10条数据），而且业务场景需要频繁使用Spark对Hive表执行某个分析操作，那么比较适合使用这种技术方案。

方案实现思路：此时可以评估一下，是否可以通过Hive来进行数据预处理（即通过Hive ETL预先对数据按照key进行聚合，或者是预先和其他表进行join），然后在Spark作业中针对的数据源就不是原来的Hive表了，而是预处理后的Hive表。此时由于数据已经预先进行过聚合或join操作了，那么在Spark作业中也就不需要使用原先的shuffle类算子执行这类操作了。

方案实现原理：这种方案从根源上解决了数据倾斜，因为彻底避免了在Spark中执行shuffle类算子，那么肯定就不会有数据倾斜的问题了。但是这里也要提醒一下大家，这种方式属于治标不治本。因为毕竟数据本身就存在分布不均匀的问题，所以Hive ETL中进行group by或者join等shuffle操作时，还是会出数据倾斜，导致Hive ETL的速度很慢。我们只是把数据倾斜的发生提前到了Hive ETL中，避免Spark程序发生数据倾斜而已。

方案优点：实现起来简单便捷，效果还非常好，完全规避掉了数据倾斜，Spark作业的性能会大幅度提升。

方案缺点：治标不治本，Hive ETL中还是会发生数据倾斜。

方案实践经验：在一些Java系统与Spark结合使用的项目中，会出现Java代码频繁调用Spark作业的场景，而且对Spark作业的执行性能要求很高，就比较适合使用这种方案。将数据倾斜提前到上游的Hive ETL，每天仅执行一次，只有那一次是比较慢的，而之后每次Java调用Spark作业时，执行速度都会很快，能够提供更好的用户体验。

项目实践经验：在美团·点评的交互式用户行为分析系统中使用了这种方案，该系统主要是允许用户通过Java Web系统提交数据分析统计任务，后端通过Java提交Spark作业进行数据分析统计。要求Spark作业速度必须要快，尽量在10分钟以内，否则速度太慢，用户体验会很差。所以我们将有些Spark作业的shuffle操作提前到了Hive ETL中，从而让Spark直接使用预处理的Hive中间表，尽可能地减少Spark的shuffle操作，大幅度提升了性能，将部分作业的性能提升了6倍以上。

解决方案二：过滤少数导致倾斜的key

方案适用场景：如果发现导致倾斜的key就少数几个，而且对计算本身的影响并不大的话，那么很适合使用这种方案。比如99%的key就对应10条数据，但是只有一个key对应了100万数据，从而导致了数据倾斜。

方案实现思路：如果我们判断那少数几个数据量特别多的key，对作业的执行和计算结果不是特别重要的话，那么干脆就直接过滤掉那少数几个key。比如，在Spark SQL中可以使用where子句过滤掉这些key或者在Spark Core中对RDD执行filter算子过滤掉这些key。如果需要每次作业执行时，动态判定哪些key的数据量最多然后再进行过滤，那么可以使用sample算子对RDD进行采样，然后计算出每个key的数量，取数据量最多的key过滤掉即可。

方案实现原理：将导致数据倾斜的key给过滤掉之后，这些key就不会参与计算了，自然不可能产生数据倾斜。

方案优点：实现简单，而且效果也很好，可以完全规避掉数据倾斜。

方案缺点：适用场景不多，大多数情况下，导致倾斜的key还是很多的，并不是只有少数几个。

方案实践经验：在项目中我们也采用过这种方案解决数据倾斜。有一次发现某一天Spark作业在运行的时候突然OOM了，追查之后发现，是Hive表中的某一个key在那天数据异常，导致数据量暴增。因此就采取每次执行前先进行采样，计算出样本中数据量最大的几个key之后，直接在程序中将那些key给过滤掉。

解决方案三：提高shuffle操作的并行度

方案适用场景：如果我们必须要对数据倾斜迎难而上，那么建议优先使用这种方案，因为这是处理数据倾斜最简单的一种方案。

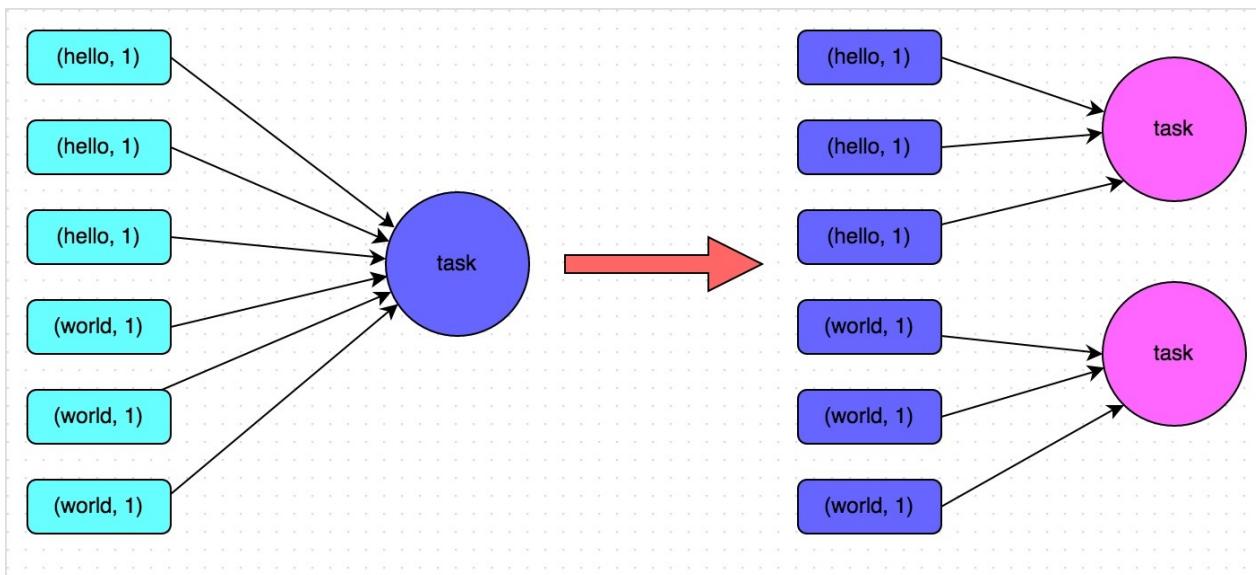
方案实现思路：在对RDD执行shuffle算子时，给shuffle算子传入一个参数，比如reduceByKey(1000)，该参数就设置了这个shuffle算子执行时shuffle read task的数量。对于Spark SQL中的shuffle类语句，比如group by、join等，需要设置一个参数，即spark.sql.shuffle.partitions，该参数代表了shuffle read task的并行度，该值默认是200，对于很多场景来说都有点过小。

方案实现原理：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。举例来说，如果原本有5个key，每个key对应10条数据，这5个key都是分配给一个task的，那么这个task就要处理50条数据。而增加了shuffle read task以后，每个task就分配到一个key，即每个task就处理10条数据，那么自然每个task的执行时间都会变短了。具体原理如下图所示。

方案优点：实现起来比较简单，可以有效缓解和减轻数据倾斜的影响。

方案缺点：只是缓解了数据倾斜而已，没有彻底根除问题，根据实践经验来看，其效果有限。

方案实践经验：该方案通常无法彻底解决数据倾斜，因为如果出现一些极端情况，比如某个key对应的数据量有100万，那么无论你的task数量增加到多少，这个对应着100万数据的key肯定还是会分配到一个task中去处理，因此注定还是会发生在数据倾斜的。所以这种方案只能说是在发现数据倾斜时尝试使用的第一种手段，尝试去用简单的方法缓解数据倾斜而已，或者是和其他方案结合起来使用。



解决方案四：两阶段聚合（局部聚合+全局聚合）

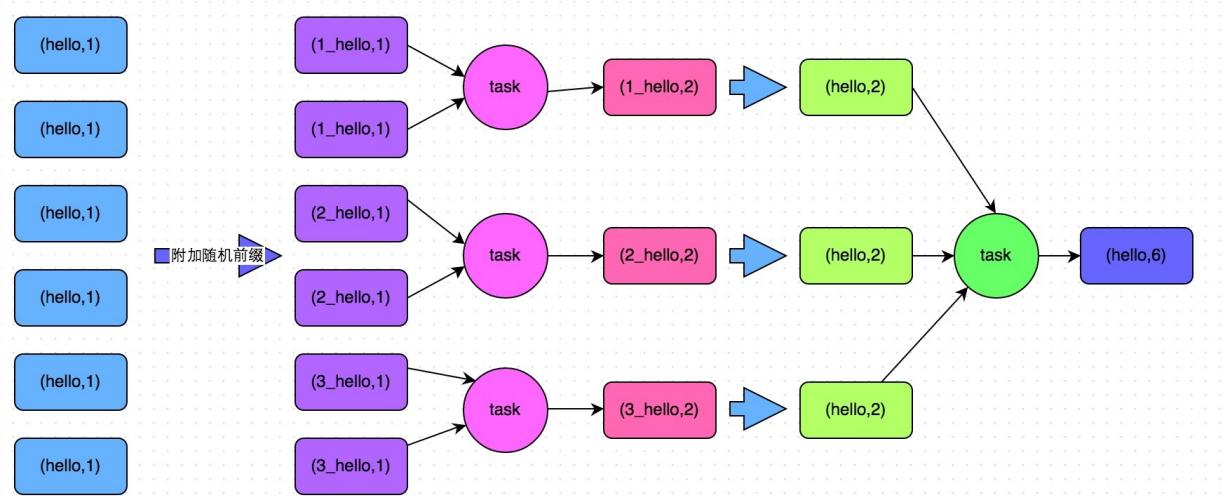
方案适用场景：对RDD执行reduceByKey等聚合类shuffle算子或者在Spark SQL中使用group by语句进行分组聚合时，比较适用这种方案。

方案实现思路：这个方案的核心实现思路就是进行两阶段聚合。第一次是局部聚合，先给每个key都打上一个随机数，比如10以内的随机数，此时原先一样的key就变成不一样的了，比如(hello, 1) (hello, 1) (hello, 1) (hello, 1)，就会变成(1_hello, 1) (1_hello, 1) (2_hello, 1) (2_hello, 1)。接着对打上随机数后的数据，执行reduceByKey等聚合操作，进行局部聚合，那么局部聚合结果，就会变成了(1_hello, 2) (2_hello, 2)。然后将各个key的前缀给去掉，就会变成(hello, 2)(hello, 2)，再次进行全局聚合操作，就可以得到最终结果了，比如(hello, 4)。

方案实现原理：将原本相同的key通过附加随机前缀的方式，变成多个不同的key，就可以让原本被一个task处理的数据分散到多个task上去做局部聚合，进而解决单个task处理数据量过多的问题。接着去除掉随机前缀，再次进行全局聚合，就可以得到最终的结果。具体原理见下图。

方案优点：对于聚合类的shuffle操作导致的数据倾斜，效果是非常不错的。通常都可以解决掉数据倾斜，或者至少是大幅度缓解数据倾斜，将Spark作业的性能提升数倍以上。

方案缺点：仅仅适用于聚合类的shuffle操作，适用范围相对较窄。如果是join类的shuffle操作，还得用其他的解决方案。



```

// 第一步，给RDD中的每个key都打上一个随机前缀。
JavaPairRDD<String, Long> randomPrefixRdd = rdd.mapToPair(
    new PairFunction<Tuple2<Long, Long>, String, Long>() {
        private static final long serialVersionUID = 1L;
        @Override
        public Tuple2<String, Long> call(Tuple2<Long, Long> tuple)
            throws Exception {
            Random random = new Random();
            int prefix = random.nextInt(10);
            return new Tuple2<String, Long>(prefix + "_" + tuple._1, tuple._2);
        }
    });
}

// 第二步，对打上随机前缀的key进行局部聚合。
JavaPairRDD<String, Long> localAggrRdd = randomPrefixRdd.reduceByKey(
    new Function2<Long, Long, Long>() {
        private static final long serialVersionUID = 1L;
        @Override
        public Long call(Long v1, Long v2) throws Exception {
            return v1 + v2;
        }
    });
}

// 第三步，去除RDD中每个key的随机前缀。
JavaPairRDD<Long, Long> removedRandomPrefixRdd = localAggrRdd.mapToPair(
    new PairFunction<Tuple2<String, Long>, Long, Long>() {
        private static final long serialVersionUID = 1L;
        @Override
        public Tuple2<Long, Long> call(Tuple2<String, Long> tuple)
            throws Exception {
            long originalKey = Long.valueOf(tuple._1.split("_")[1]);
            return new Tuple2<Long, Long>(originalKey, tuple._2);
        }
    });
}

// 第四步，对去除了随机前缀的RDD进行全局聚合。
JavaPairRDD<Long, Long> globalAggrRdd = removedRandomPrefixRdd.reduceByKey(
    new Function2<Long, Long, Long>() {
        private static final long serialVersionUID = 1L;
        @Override
        public Long call(Long v1, Long v2) throws Exception {
            return v1 + v2;
        }
    });
}

```

解决方案五：将reduce join转为map join

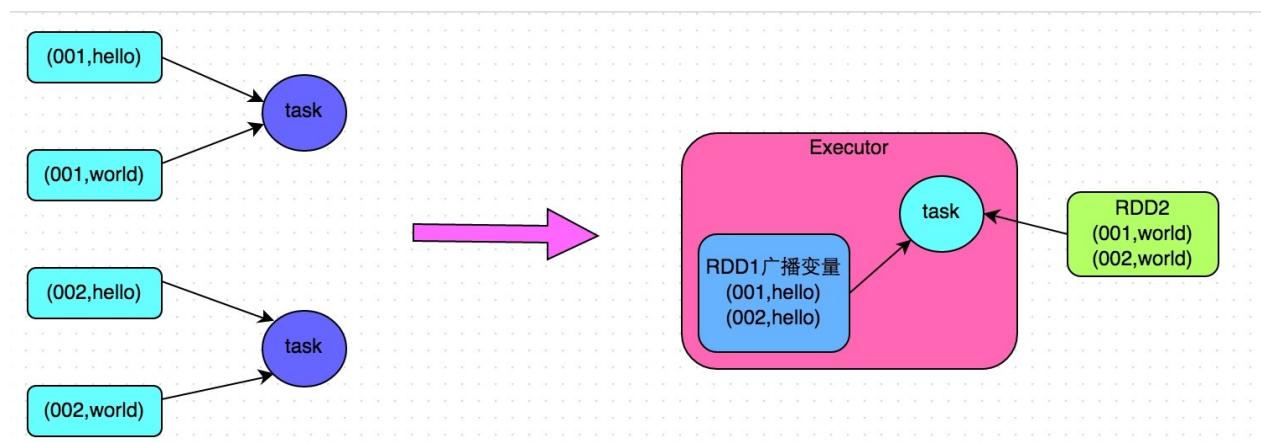
方案适用场景：在对RDD使用join类操作，或者是在Spark SQL中使用join语句时，而且join操作中的一个RDD或表的数据量比较小（比如几百M或者一两G），比较适用此方案。

方案实现思路：不使用join算子进行连接操作，而使用Broadcast变量与map类算子实现join操作，进而完全规避掉shuffle类的操作，彻底避免数据倾斜的发生和出现。将较小RDD中的数据直接通过collect算子拉取到Driver端的内存中来，然后对其创建一个Broadcast变量；接着对另外一个RDD执行map类算子，在算子函数内，从Broadcast变量中获取较小RDD的全量数据，与当前RDD的每一条数据按照连接key进行比对，如果连接key相同的话，那么就将两个RDD的数据用你需要的方式连接起来。

方案实现原理：普通的join是会走shuffle过程的，而一旦shuffle，就相当于会将相同key的数据拉取到一个shuffle read task中再进行join，此时就是reduce join。但是如果一个RDD是比较小的，则可以采用广播小RDD全量数据+map算子来实现与join同样的效果，也就是map join，此时就不会发生shuffle操作，也就不会发生数据倾斜。具体原理如下图所示。

方案优点：对join操作导致的数据倾斜，效果非常好，因为根本就不会发生shuffle，也就根本不会发生数据倾斜。

方案缺点：适用场景较少，因为这个方案只适用于一个大表和一个小表的情况。毕竟我们需要将小表进行广播，此时会比较消耗内存资源，driver和每个Executor内存中都会驻留一份小RDD的全量数据。如果我们广播出去的RDD数据比较大，比如10G以上，那么就可能发生内存溢出了。因此并不适合两个都是大表的情况。



```

// 首先将数据量比较小的RDD的数据，collect到Driver中来。
List<Tuple2<Long, Row>> rdd1Data = rdd1.collect()
// 然后使用Spark的广播功能，将小RDD的数据转换成广播变量，这样每个Executor就只有一份RDD的数据。
// 可以尽可能节省内存空间，并且减少网络传输性能开销。
final Broadcast<List<Tuple2<Long, Row>>> rdd1DataBroadcast = sc.broadcast(rdd1Data);

// 对另外一个RDD执行map类操作，而不再是join类操作。
JavaPairRDD<String, Tuple2<String, Row>> joinedRdd = rdd2.mapToPair(
    new PairFunction<Tuple2<Long, String>, String, Tuple2<String, Row>>() {
        private static final long serialVersionUID = 1L;
        @Override
        public Tuple2<String, Tuple2<String, Row>> call(Tuple2<Long, String> tuple)
    )
        throws Exception {
    // 在算子函数中，通过广播变量，获取到本地Executor中的rdd1数据。
    List<Tuple2<Long, Row>> rdd1Data = rdd1DataBroadcast.value();
    // 可以将rdd1的数据转换为一个Map，便于后面进行join操作。
    Map<Long, Row> rdd1DataMap = new HashMap<Long, Row>();
    for(Tuple2<Long, Row> data : rdd1Data) {
        rdd1DataMap.put(data._1, data._2);
    }
    // 获取当前RDD数据的key以及value。
    String key = tuple._1;
    String value = tuple._2;
    // 从rdd1数据Map中，根据key获取到可以join到的数据。
    Row rdd1Value = rdd1DataMap.get(key);
    return new Tuple2<String, String>(key, new Tuple2<String, Row>(value,
    rdd1Value));
}
});

// 这里得提示一下。
// 上面的做法，仅仅适用于rdd1中的key没有重复，全部是唯一的场景。
// 如果rdd1中有多个相同的key，那么就得用flatMap类的操作，
// 在进行join的时候不能用map，而是得遍历rdd1所有数据进行join。
// rdd2中每条数据都可能会返回多条join后的数据。

```

解决方案六：采样倾斜key并分拆join操作

方案适用场景：两个RDD/Hive表进行join的时候，如果数据量都比较大，无法采用“解决方案五”，那么此时可以看一下两个RDD/Hive表中的key分布情况。如果出现数据倾斜，是因为其中某一个RDD/Hive表中的少数几个key的数据量过大，而另一个RDD/Hive表中的所有key都分布比较均匀，那么采用这个解决方案是比较合适的。

方案实现思路：

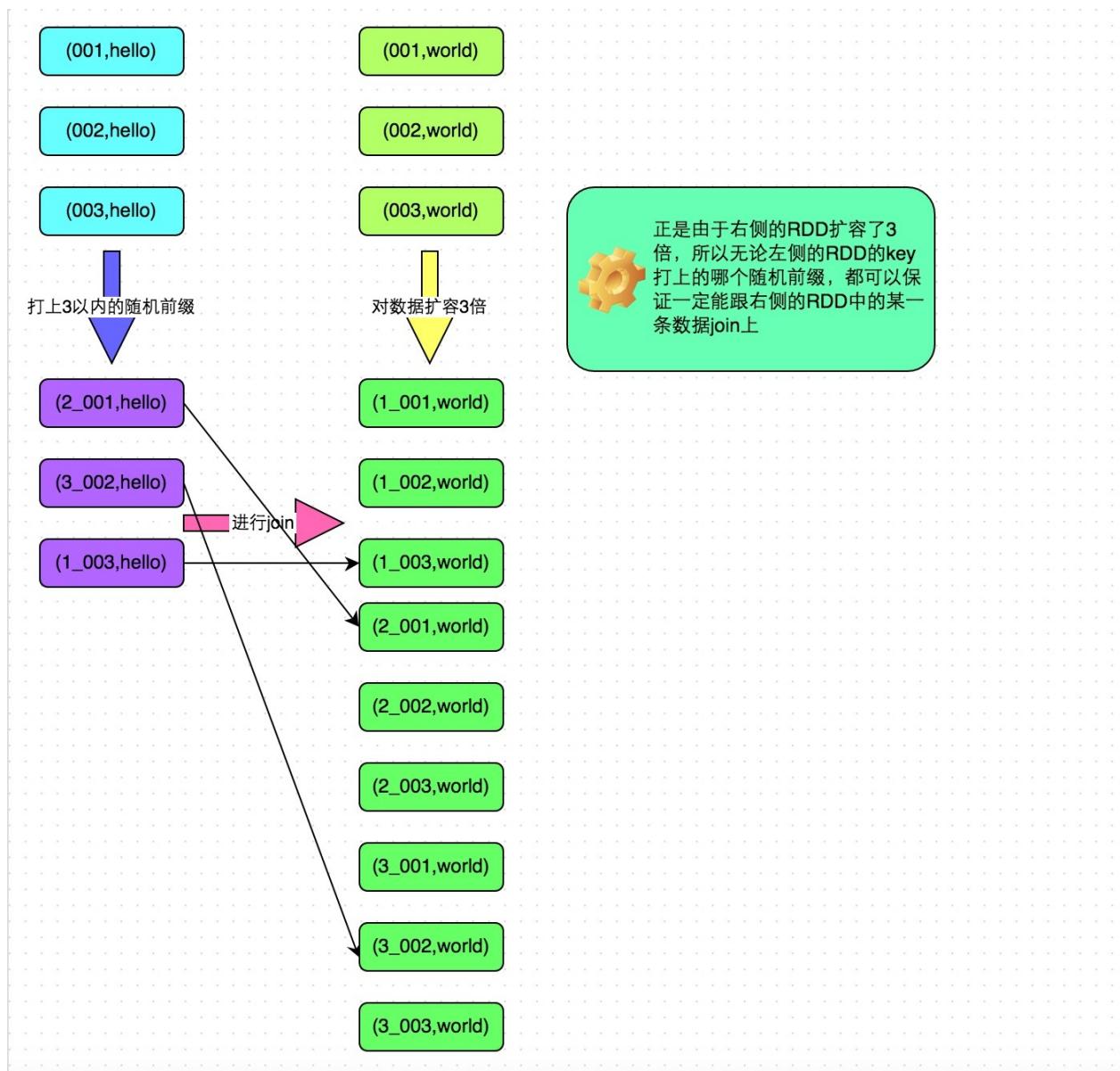
- 对包含少数几个数据量过大的key的那个RDD，通过sample算子采样出一份样本来，然后统计一下每个key的数量，计算出来数据量最大的是哪几个key。

- 然后将这几个key对应的数据从原来的RDD中拆分出来，形成一个单独的RDD，并给每个key都打上n以内的随机数作为前缀，而不会导致倾斜的大部分key形成另外一个RDD。
- 接着将需要join的另一个RDD，也过滤出来那几个倾斜key对应的数据并形成一个单独的RDD，将每条数据膨胀成n条数据，这n条数据都按顺序附加一个0~n的前缀，不会导致倾斜的大部分key也形成另外一个RDD。
- 再将附加了随机前缀的独立RDD与另一个膨胀n倍的独立RDD进行join，此时就可以将原先相同的key打散成n份，分散到多个task中去进行join了。
- 而另外两个普通的RDD就照常join即可。
- 最后将两次join的结果使用union算子合并起来即可，就是最终的join结果。

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。具体原理见下图。

方案优点：对于join导致的数据倾斜，如果只是某几个key导致了倾斜，采用该方式可以用最有效的方式打散key进行join。而且只需要针对少数倾斜key对应的数据进行扩容n倍，不需要对全量数据进行扩容。避免了占用过多内存。

方案缺点：如果导致倾斜的key特别多的话，比如成千上万个key都导致数据倾斜，那么这种方式也不适合。



```
// 首先从包含了少数几个导致数据倾斜key的rdd1中，采样10%的样本数据。
JavaPairRDD<Long, String> sampledRDD = rdd1.sample(false, 0.1);

// 对样本数据RDD统计出每个key的出现次数，并按出现次数降序排序。
// 对降序排序后的数据，取出top 1或者top 100的数据，也就是key最多的前n个数据。
// 具体取出多少个数据量最多的key，由大家自己决定，我们这里就取1个作为示范。
JavaPairRDD<Long, Long> mappedSampledRDD = sampledRDD.mapToPair(
    new PairFunction<Tuple2<Long, String>, Long, Long>() {
        private static final long serialVersionUID = 1L;
        @Override
        public Tuple2<Long, Long> call(Tuple2<Long, String> tuple)
            throws Exception {
            return new Tuple2<Long, Long>(tuple._1, 1L);
        }
    });
JavaPairRDD<Long, Long> countedSampledRDD = mappedSampledRDD.reduceByKey(
```

```

        new Function2<Long, Long, Long>() {
            private static final long serialVersionUID = 1L;
            @Override
            public Long call(Long v1, Long v2) throws Exception {
                return v1 + v2;
            }
        });
    JavaPairRDD<Long, Long> reversedSampledRDD = countedSampledRDD.mapToPair(
        new PairFunction<Tuple2<Long, Long>, Long, Long>() {
            private static final long serialVersionUID = 1L;
            @Override
            public Tuple2<Long, Long> call(Tuple2<Long, Long> tuple)
                throws Exception {
                return new Tuple2<Long, Long>(tuple._2, tuple._1);
            }
        });
    final Long skewedUserId = reversedSampledRDD.sortByKey(false).take(1).get(0)._2;

    // 从rdd1中分拆出导致数据倾斜的key，形成独立的RDD。
    JavaPairRDD<Long, String> skewedRDD = rdd1.filter(
        new Function<Tuple2<Long, String>, Boolean>() {
            private static final long serialVersionUID = 1L;
            @Override
            public Boolean call(Tuple2<Long, String> tuple) throws Exception {
                return tuple._1.equals(skewedUserId);
            }
        });
    // 从rdd1中分拆出不导致数据倾斜的普通key，形成独立的RDD。
    JavaPairRDD<Long, String> commonRDD = rdd1.filter(
        new Function<Tuple2<Long, String>, Boolean>() {
            private static final long serialVersionUID = 1L;
            @Override
            public Boolean call(Tuple2<Long, String> tuple) throws Exception {
                return !tuple._1.equals(skewedUserId);
            }
        });
}

// rdd2，就是那个所有key的分布相对较为均匀的rdd。
// 这里将rdd2中，前面获取到的key对应的数据，过滤出来，分拆成单独的rdd，
// 并对rdd中的数据使用flatMap算子都扩容100倍。
// 对扩容的每条数据，都打上0~100的前缀。
JavaPairRDD<String, Row> skewedRdd2 = rdd2.filter(
    new Function<Tuple2<Long, Row>, Boolean>() {
        private static final long serialVersionUID = 1L;
        @Override
        public Boolean call(Tuple2<Long, Row> tuple) throws Exception {
            return tuple._1.equals(skewedUserId);
        }
    }).flatMapToPair(new PairFlatMapFunction<Tuple2<Long, Row>, String, Row>() {
        private static final long serialVersionUID = 1L;
        @Override
        public Iterable<Tuple2<String, Row>> call(
            Tuple2<Long, Row> tuple) throws Exception {
    
```

```

        Random random = new Random();
        List<Tuple2<String, Row>> list = new ArrayList<Tuple2<String, Row>>();
        for(int i = 0; i < 100; i++) {
            list.add(new Tuple2<String, Row>(i + "_" + tuple._1, tuple._2));
        }
        return list;
    }

});

// 将rdd1中分拆出来的导致倾斜的key的独立rdd，每条数据都打上100以内的随机前缀。
// 然后将这个rdd1中分拆出来的独立rdd，与上面rdd2中分拆出来的独立rdd，进行join。
JavaPairRDD<Long, Tuple2<String, Row>> joinedRDD1 = skewedRDD.mapToPair(
    new PairFunction<Tuple2<Long, String>, String, String>() {
        private static final long serialVersionUID = 1L;
        @Override
        public Tuple2<String, String> call(Tuple2<Long, String> tuple)
            throws Exception {
            Random random = new Random();
            int prefix = random.nextInt(100);
            return new Tuple2<String, String>(prefix + "_" + tuple._1, tuple._2);
        }
    })
    .join(skewedUserId2infoRDD)
    .mapToPair(new PairFunction<Tuple2<String, Tuple2<String, Row>>,
        Long, Tuple2<String, Row>>() {
        private static final long serialVersionUID = 1L;
        @Override
        public Tuple2<Long, Tuple2<String, Row>> call(
            Tuple2<String, Tuple2<String, Row>> tuple)
            throws Exception {
            long key = Long.valueOf(tuple._1.split("_")[1]);
            return new Tuple2<Long, Tuple2<String, Row>>(key, tuple._2);
        }
    });
}

// 将rdd1中分拆出来的包含普通key的独立rdd，直接与rdd2进行join。
JavaPairRDD<Long, Tuple2<String, Row>> joinedRDD2 = commonRDD.join(rdd2);

// 将倾斜key join后的结果与普通key join后的结果，union起来。
// 就是最终的join结果。
JavaPairRDD<Long, Tuple2<String, Row>> joinedRDD = joinedRDD1.union(joinedRDD2);

```

解决方案七：使用随机前缀和扩容**RDD**进行join

方案适用场景：如果在进行join操作时，RDD中有大量的key导致数据倾斜，那么进行分拆key也没什么意义，此时就只能使用最后一种方案来解决问题了。

方案实现思路：

- 该方案的实现思路基本和“解决方案六”类似，首先查看RDD/Hive表中的数据分布情况，找到那个造成数据倾斜的RDD/Hive表，比如有多个key都对应了超过1万条数据。
- 然后将该RDD的每条数据都打上一个n以内的随机前缀。
- 同时对另外一个正常的RDD进行扩容，将每条数据都扩容成n条数据，扩容出来的每条数据都依次打上一个0~n的前缀。
- 最后将两个处理后的RDD进行join即可。

方案实现原理：将原先一样的key通过附加随机前缀变成不一样的key，然后就可以将这些处理后的“不同key”分散到多个task中去处理，而不是让一个task处理大量的相同key。该方案与“解决方案六”的不同之处就在于，上一种方案是尽量只对少数倾斜key对应的数据进行特殊处理，由于处理过程需要扩容RDD，因此上一种方案扩容RDD后对内存的占用并不大；而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。

方案优点：对join类型的数据倾斜基本都可以处理，而且效果也相对比较显著，性能提升效果非常不错。

方案缺点：该方案更多的是缓解数据倾斜，而不是彻底避免数据倾斜。而且需要对整个RDD进行扩容，对内存资源要求很高。

方案实践经验：曾经开发一个数据需求的时候，发现一个join导致了数据倾斜。优化之前，作业的执行时间大约是60分钟左右；使用该方案优化之后，执行时间缩短到10分钟左右，性能提升了6倍。

```

// 首先将其中一个key分布相对较为均匀的RDD膨胀100倍。
JavaPairRDD<String, Row> expandedRDD = rdd1.flatMapToPair(
    new PairFlatMapFunction<Tuple2<Long, Row>, String, Row>() {
        private static final long serialVersionUID = 1L;
        @Override
        public Iterable<Tuple2<String, Row>> call(Tuple2<Long, Row> tuple)
            throws Exception {
            List<Tuple2<String, Row>> list = new ArrayList<Tuple2<String, Row>>();
            for(int i = 0; i < 100; i++) {
                list.add(new Tuple2<String, Row>(0 + "_" + tuple._1, tuple._2));
            }
            return list;
        }
    });
}

// 其次，将另一个有数据倾斜key的RDD，每条数据都打上100以内的随机前缀。
JavaPairRDD<String, String> mappedRDD = rdd2.mapToPair(
    new PairFunction<Tuple2<Long, String>, String, String>() {
        private static final long serialVersionUID = 1L;
        @Override
        public Tuple2<String, String> call(Tuple2<Long, String> tuple)
            throws Exception {
            Random random = new Random();
            int prefix = random.nextInt(100);
            return new Tuple2<String, String>(prefix + "_" + tuple._1, tuple._2);
        }
    });
}

// 将两个处理后的RDD进行join即可。
JavaPairRDD<String, Tuple2<String, Row>> joinedRDD = mappedRDD.join(expandedRDD);

```

解决方案八：多种方案组合使用

在实践中发现，很多情况下，如果只是处理较为简单的数据倾斜场景，那么使用上述方案中的某一种基本就可以解决。但是如果要处理一个较为复杂的数据倾斜场景，那么可能需要将多种方案组合起来使用。比如说，我们针对出现了多个数据倾斜环节的Spark作业，可以先运用解决方案一和二，预处理一部分数据，并过滤一部分数据来缓解；其次可以对某些shuffle操作提升并行度，优化其性能；最后还可以针对不同的聚合或join操作，选择一种方案来优化其性能。大家需要对这些方案的思路和原理都透彻理解之后，在实践中根据各种不同的情况，灵活运用多种方案，来解决自己的数据倾斜问题。

2 shuffle调优

2.1 调优概述

大多数Spark作业的性能主要就是消耗在了shuffle环节，因为该环节包含了大量的磁盘IO、序列化、网络数据传输等操作。因此，如果要让作业的性能更上一层楼，就有必要对shuffle过程进行调优。但是也必须提醒大家的是，影响一个Spark作业性能的因素，主要还是代码开发、资源参数以及数据倾斜，shuffle调优只能在整个Spark的性能调优中占到一小部分而已。因此大家务必把握住调优的基本原则，千万不要舍本逐末。下面我们就给大家详细讲解shuffle的原理，以及相关参数的说明，同时给出各个参数的调优建议。

2.2 ShuffleManager发展概述

在Spark的源码中，负责shuffle过程的执行、计算和处理的组件主要就是ShuffleManager，也即shuffle管理器。而随着Spark的版本的发展，ShuffleManager也在不断迭代，变得越来越先进。

在Spark 1.2以前，默认的shuffle计算引擎是HashShuffleManager。该ShuffleManager而HashShuffleManager有着一个非常严重的弊端，就是会产生大量的中间磁盘文件，进而由大量的磁盘IO操作影响了性能。

因此在Spark 1.2以后的版本中，默认的ShuffleManager改成了SortShuffleManager。SortShuffleManager相较于HashShuffleManager来说，有了一定的改进。主要就在于，每个Task在进行shuffle操作时，虽然也会产生较多的临时磁盘文件，但是最后会将所有的临时文件合并（merge）成一个磁盘文件，因此每个Task就只有一个磁盘文件。在下一个stage的shuffle read task拉取自己的数据时，只要根据索引读取每个磁盘文件中的部分数据即可。

2.3 HashShuffleManager运行原理

未经优化的HashShuffleManager

下图说明了未经优化的HashShuffleManager的原理。这里我们先明确一个假设前提：每个Executor只有1个CPU core，也就是说，无论这个Executor上分配多少个task线程，同一时间都只能执行一个task线程。

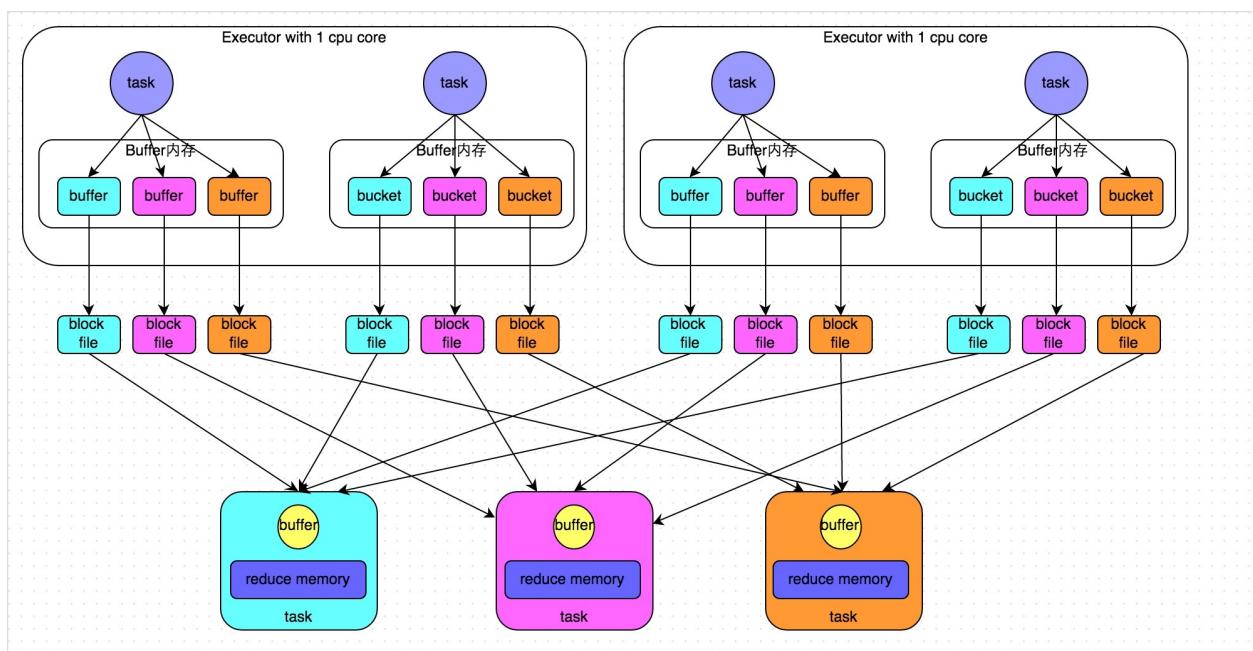
我们先从shuffle write开始说起。shuffle write阶段，主要就是在在一个stage结束计算之后，为了下一个stage可以执行shuffle类的算子（比如reduceByKey），而将每个task处理的数据按key进行“分类”。所谓“分类”，就是对相同的key执行hash算法，从而将相同key都写入同一个磁盘文件中，而每一个磁盘文件都只属于下游stage的一个task。在将数据写入磁盘之前，会先将数据写入内存缓冲中，当内存缓冲填满之后，才会溢写到磁盘文件中去。

那么每个执行shuffle write的task，要为下一个stage创建多少个磁盘文件呢？很简单，下一个stage的task有多少个，当前stage的每个task就要创建多少份磁盘文件。比如下一个stage总共有100个task，那么当前stage的每个task都要创建100份磁盘文件。如果当前stage

有50个task，总共有10个Executor，每个Executor执行5个Task，那么每个Executor上总共就要创建500个磁盘文件，所有Executor上会创建5000个磁盘文件。由此可见，未经优化的shuffle write操作所产生的磁盘文件的数量是极其惊人的。

接着我们来说说shuffle read。shuffle read，通常就是一个stage刚开始时要做的事情。此时该stage的每一个task就需要将上一个stage的计算结果中的所有相同key，从各个节点上通过网络都拉取到自己所在的节点上，然后进行key的聚合或连接等操作。由于shuffle write的过程中，task给下游stage的每个task都创建了一个磁盘文件，因此shuffle read的过程中，每个task只要从上游stage的所有task所在节点上，拉取属于自己的那一个磁盘文件即可。

shuffle read的拉取过程是一边拉取一边进行聚合的。每个shuffle read task都会有一个自己的buffer缓冲，每次都只能拉取与buffer缓冲相同大小的数据，然后通过内存中的一个Map进行聚合等操作。聚合完一批数据后，再拉取下一批数据，并放到buffer缓冲中进行聚合操作。以此类推，直到最后将所有数据拉取完，并得到最终的结果。



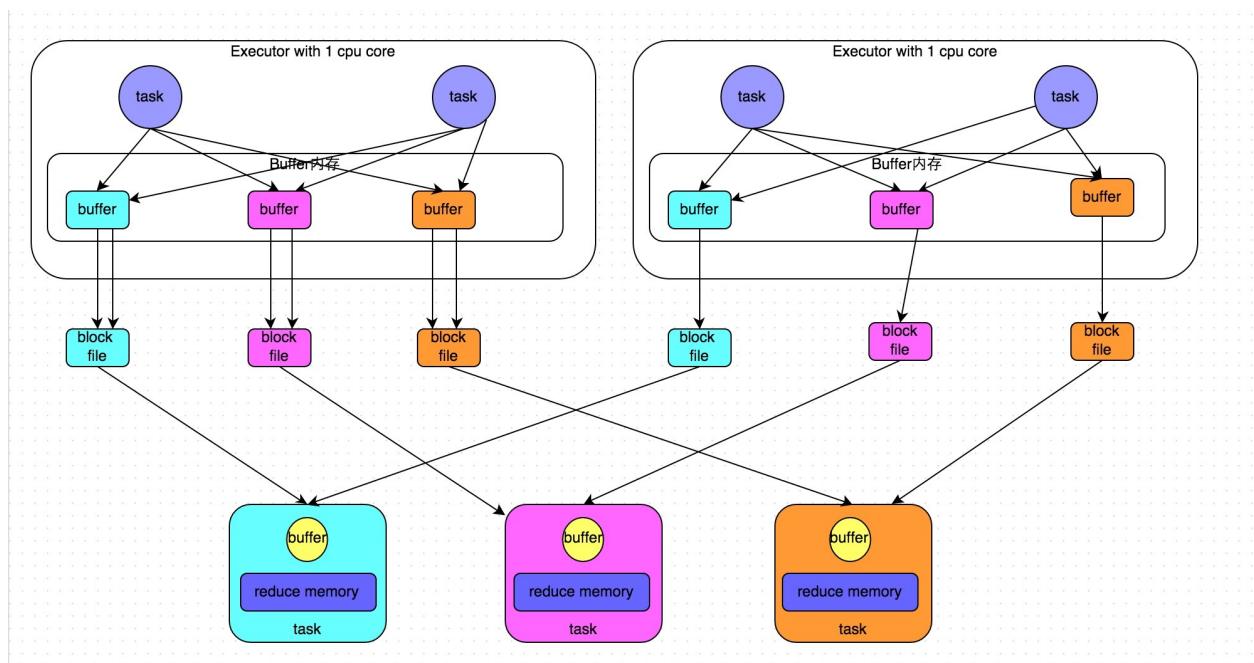
优化后的HashShuffleManager

下图说明了优化后的HashShuffleManager的原理。这里说的优化，是指我们可以设置一个参数，`spark.shuffle.consolidateFiles`。该参数默认值为false，将其设置为true即可开启优化机制。通常来说，如果我们使用HashShuffleManager，那么都建议开启这个选项。

开启consolidate机制之后，在shuffle write过程中，task就不是为下游stage的每个task创建一个磁盘文件了。此时会出现shuffleFileGroup的概念，每个shuffleFileGroup会对应一批磁盘文件，磁盘文件的数量与下游stage的task数量是相同的。一个Executor上有多少个CPU core，就可以并行执行多少个task。而第一批并行执行的每个task都会创建一个shuffleFileGroup，并将数据写入对应的磁盘文件内。

当Executor的CPU core执行完一批task，接着执行下一批task时，下一批task就会复用之前已有的shuffleFileGroup，包括其中的磁盘文件。也就是说，此时task会将数据写入已有的磁盘文件中，而不会写入新的磁盘文件中。因此，consolidate机制允许不同的task复用同一批磁盘文件，这样就可以有效将多个task的磁盘文件进行一定程度上的合并，从而大幅度减少磁盘文件的数量，进而提升shuffle write的性能。

假设第二个stage有100个task，第一个stage有50个task，总共还是有10个Executor，每个Executor执行5个task。那么原本使用未经优化的HashShuffleManager时，每个Executor会产生500个磁盘文件，所有Executor会产生5000个磁盘文件的。但是此时经过优化之后，每个Executor创建的磁盘文件的数量的计算公式为：CPU core的数量 * 下一个stage的task数量。也就是说，每个Executor此时只会创建100个磁盘文件，所有Executor只会创建1000个磁盘文件。



2.4 SortShuffleManager运行原理

SortShuffleManager的运行机制主要分成两种，一种是普通运行机制，另一种是bypass运行机制。当shuffle read task的数量小于等于spark.shuffle.sort.bypassMergeThreshold参数的值时（默认为200），就会启用bypass机制。

普通运行机制

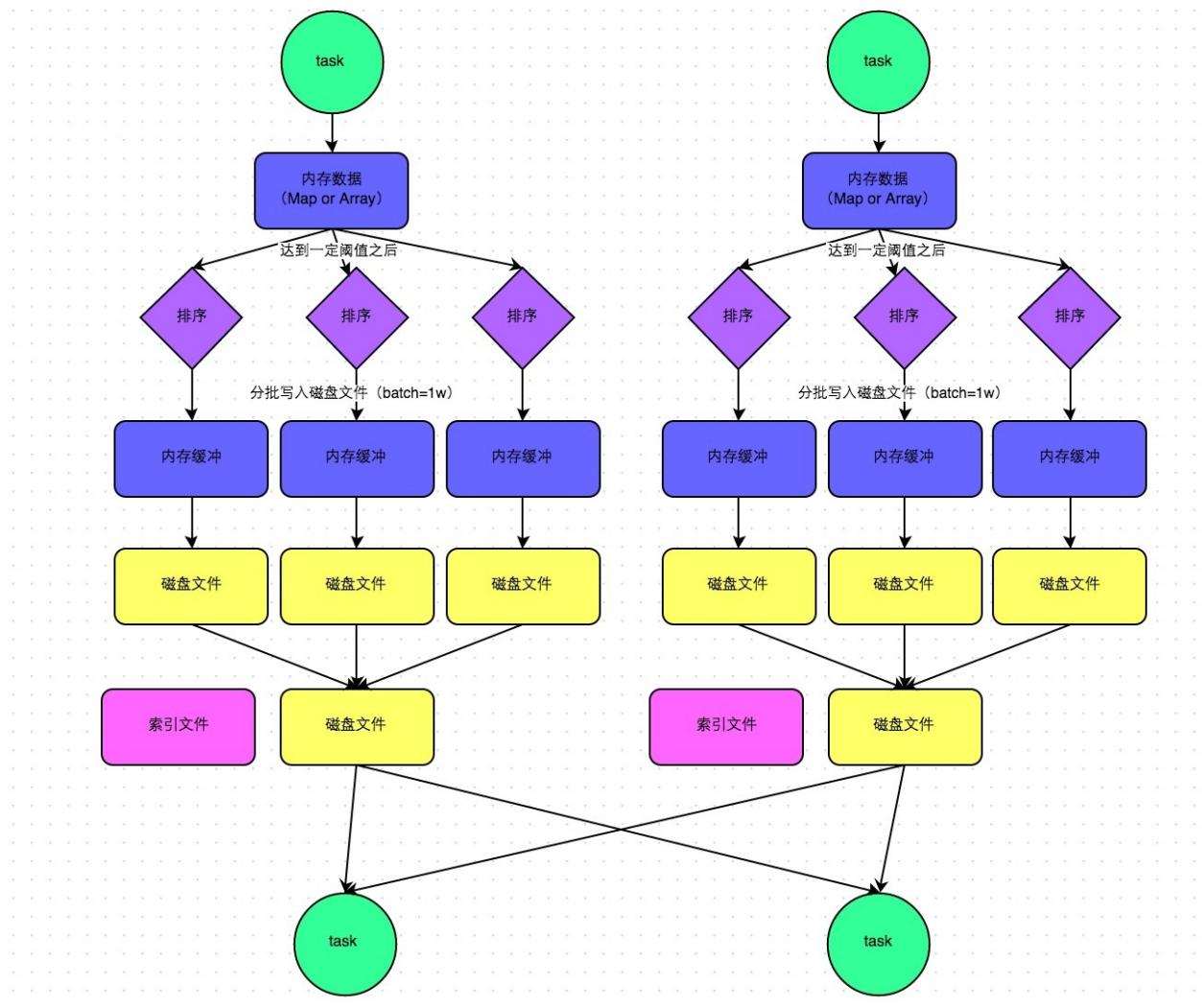
下图说明了普通的SortShuffleManager的原理。在该模式下，数据会先写入一个内存数据结构中，此时根据不同的shuffle算子，可能选用不同的数据结构。如果是reduceByKey这种聚合类的shuffle算子，那么会选用Map数据结构，一边通过Map进行聚合，一边写入内存；如果

是join这种普通的shuffle算子，那么会选用Array数据结构，直接写入内存。接着，每写一条数据进入内存数据结构之后，就会判断一下，是否达到了某个临界阈值。如果达到临界阈值的话，那么就会尝试将内存数据结构中的数据溢写到磁盘，然后清空内存数据结构。

在溢写到磁盘文件之前，会先根据key对内存数据结构中已有的数据进行排序。排序过后，会分批将数据写入磁盘文件。默认的batch数量是10000条，也就是说，排序好的数据，会以每批1万条数据的形式分批写入磁盘文件。写入磁盘文件是通过Java的BufferedOutputStream实现的。BufferedOutputStream是Java的缓冲输出流，首先会将数据缓冲在内存中，当内存缓冲满溢之后再一次写入磁盘文件中，这样可以减少磁盘IO次数，提升性能。

一个task将所有数据写入内存数据结构的过程中，会发生多次磁盘溢写操作，也就会产生多个临时文件。最后会将之前所有的临时磁盘文件都进行合并，这就是merge过程，此时会将之前所有临时磁盘文件中的数据读取出来，然后依次写入最终的磁盘文件之中。此外，由于一个task就只对应一个磁盘文件，也就意味着该task为下游stage的task准备的数据都在这一个文件中，因此还会单独写一份索引文件，其中标识了下游各个task的数据在文件中的start offset与end offset。

SortShuffleManager由于有一个磁盘文件merge的过程，因此大大减少了文件数量。比如第一个stage有50个task，总共有10个Executor，每个Executor执行5个task，而第二个stage有100个task。由于每个task最终只有一个磁盘文件，因此此时每个Executor上只有5个磁盘文件，所有Executor只有50个磁盘文件。



bypass 运行机制

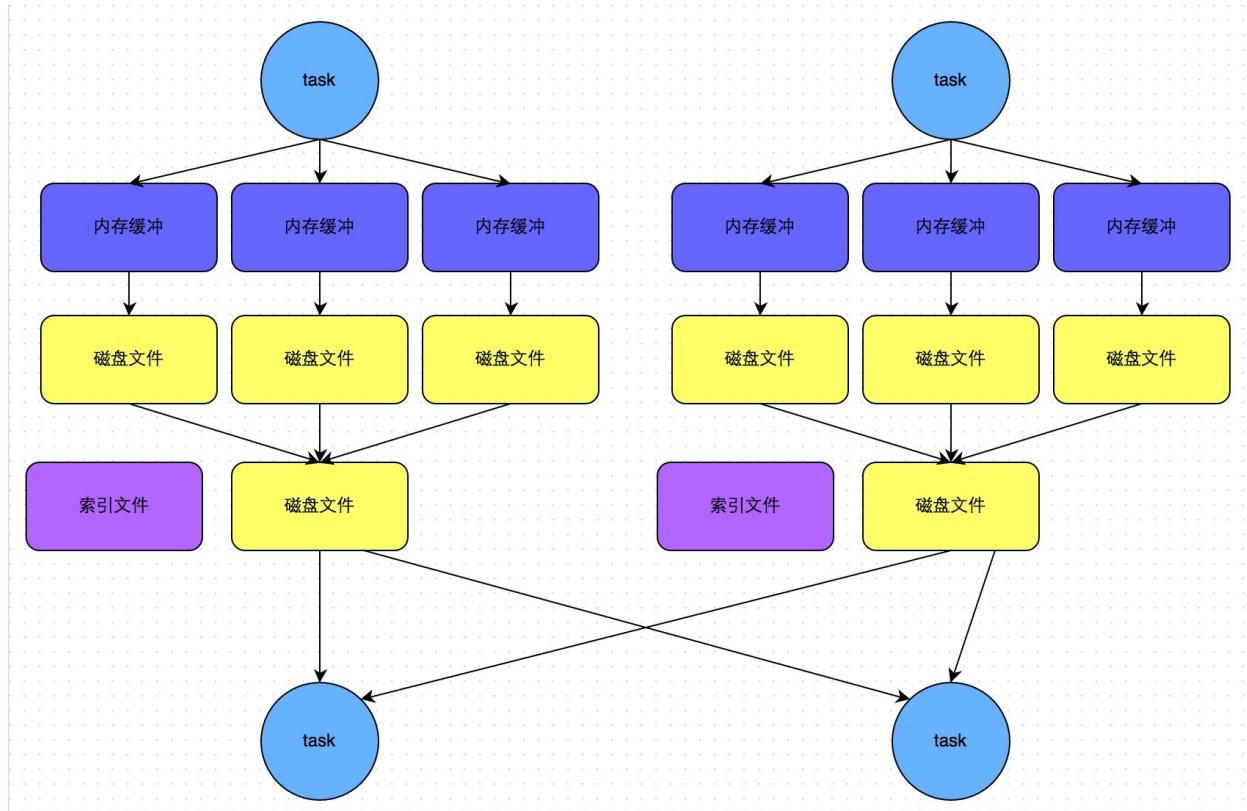
下图说明了bypass SortShuffleManager的原理。bypass运行机制的触发条件如下：

- shuffle map task数量小于spark.shuffle.sort.bypassMergeThreshold参数的值。
- 不是聚合类的shuffle算子（比如reduceByKey）。

此时task会为每个下游task都创建一个临时磁盘文件，并将数据按key进行hash然后根据key的hash值，将key写入对应的磁盘文件之中。当然，写入磁盘文件时也是先写入内存缓冲，缓冲写满之后再溢写到磁盘文件的。最后，同样会将所有临时磁盘文件都合并成一个磁盘文件，并创建一个单独的索引文件。

该过程的磁盘写机制其实跟未经优化的HashShuffleManager是一模一样的，因为都要创建数量惊人的磁盘文件，只是在最后会做一个磁盘文件的合并而已。因此少量的最终磁盘文件，也让该机制相对未经优化的HashShuffleManager来说，shuffle read的性能会更好。

而该机制与普通SortShuffleManager运行机制的不同在于：第一，磁盘写机制不同；第二，不会进行排序。也就是说，启用该机制的最大好处在于，shuffle write过程中，不需要进行数据的排序操作，也就节省掉了这部分的性能开销。



2.5 shuffle相关参数调优

spark.shuffle.file.buffer

- 默认值：32k
- 参数说明：该参数用于设置shuffle write task的BufferedOutputStream的buffer缓冲大小。将数据写到磁盘文件之前，会先写入buffer缓冲中，待缓冲写满之后，才会溢写到磁盘。
- 调优建议：如果作业可用的内存资源较为充足的话，可以适当增加这个参数的大小（比如64k），从而减少shuffle write过程中溢写磁盘文件的次数，也可以减少磁盘IO次数，进而提升性能。在实践中发现，合理调节该参数，性能会有1%~5%的提升。

spark.reducer.maxSizeInFlight

- 默认值：48m
- 参数说明：该参数用于设置shuffle read task的buffer缓冲大小，而这个buffer缓冲决定了每次能够拉取多少数据。
- 调优建议：如果作业可用的内存资源较为充足的话，可以适当增加这个参数的大小（比

如96m），从而减少拉取数据的次数，也就可以减少网络传输的次数，进而提升性能。在实践中发现，合理调节该参数，性能会有1%~5%的提升。

spark.shuffle.io.maxRetries

- 默认值：3
- 参数说明：shuffle read task从shuffle write task所在节点拉取属于自己的数据时，如果因为网络异常导致拉取失败，是会自动进行重试的。该参数就代表了可以重试的最大次数。如果在指定次数之内拉取还是没有成功，就可能会导致作业执行失败。
- 调优建议：对于那些包含了特别耗时的shuffle操作的作业，建议增加重试最大次数（比如60次），以避免由于JVM的full gc或者网络不稳定等因素导致的数据拉取失败。在实践中发现，对于针对超大数据量（数十亿~上百亿）的shuffle过程，调节该参数可以大幅度提升稳定性。

spark.shuffle.io.retryWait

- 默认值：5s
- 参数说明：具体解释同上，该参数代表了每次重试拉取数据的等待间隔，默认是5s。
- 调优建议：建议加大间隔时长（比如60s），以增加shuffle操作的稳定性。

spark.shuffle.memoryFraction

- 默认值：0.2
- 参数说明：该参数代表了Executor内存中，分配给shuffle read task进行聚合操作的内存比例，默认是20%。
- 调优建议：在资源参数调优中讲解过这个参数。如果内存充足，而且很少使用持久化操作，建议调高这个比例，给shuffle read的聚合操作更多内存，以避免由于内存不足导致聚合过程中频繁读写磁盘。在实践中发现，合理调节该参数可以将性能提升10%左右。

spark.shuffle.manager

- 默认值：sort
- 参数说明：该参数用于设置ShuffleManager的类型。Spark 1.5以后，有三个可选项：hash、sort和tungsten-sort。HashShuffleManager是Spark 1.2以前的默认选项，但是Spark 1.2以及之后的版本默认都是SortShuffleManager了。tungsten-sort与sort类似，但是使用了tungsten计划中的堆外内存管理机制，内存使用效率更高。
- 调优建议：由于SortShuffleManager默认会对数据进行排序，因此如果你的业务逻辑中需要该排序机制的话，则使用默认的SortShuffleManager就可以；而如果你的业务逻辑不需要对数据进行排序，那么建议参考后面的几个参数调优，通过bypass机制或优化的HashShuffleManager来避免排序操作，同时提供较好的磁盘读写性能。这里要注意的是，tungsten-sort要慎用，因为之前发现了一些相应的bug。

spark.shuffle.sort.bypassMergeThreshold

- 默认值：200
- 参数说明：当ShuffleManager为SortShuffleManager时，如果shuffle read task的数量小于这个阈值（默认是200），则shuffle write过程中不会进行排序操作，而是直接按照未经优化的HashShuffleManager的方式去写数据，但是最后会将每个task产生的所有临时磁盘文件都合并成一个文件，并会创建单独的索引文件。
- 调优建议：当你使用SortShuffleManager时，如果的确不需要排序操作，那么建议将这个参数调大一些，大于shuffle read task的数量。那么此时就会自动启用bypass机制，map-side就不会进行排序了，减少了排序的性能开销。但是这种方式下，依然会产生大量的磁盘文件，因此shuffle write性能有待提高。

spark.shuffle.consolidateFiles

- 默认值：false
- 参数说明：如果使用HashShuffleManager，该参数有效。如果设置为true，那么就会开启consolidate机制，会大幅度合并shuffle write的输出文件，对于shuffle read task数量特别多的情况下，这种方法可以极大地减少磁盘IO开销，提升性能。
- 调优建议：如果的确不需要SortShuffleManager的排序机制，那么除了使用bypass机制，还可以尝试将spark.shuffle.manager参数手动指定为hash，使用HashShuffleManager，同时开启consolidate机制。在实践中尝试过，发现其性能比开启了bypass机制的SortShuffleManager要高出10%~30%。

引用

[Spark性能优化指南——高级篇](#)