Spark: Cluster Computing wi th Working Sets

Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, Ion Stoica
University of California, Berkeley

CONTENT

目录

01

Motivation

文中主要介绍了当时解决大规模数据的分布式框架存在的局限性，并针对这些问题提出了Spark的解决方案

摘要

MapReduce及其各种变种，在商业集群，实现大规模数据密集型应用方面取得了巨大成功。然而，这些系统大多都是围绕非迭代数据模型构建的，不适合其他主流应用。本文侧重于此类应用：可以并行操作重用一组工作数据集的应用。包括许多机器学习迭代算法，以及交互式数据分析等。我们提出了一个名为Spark的新框架，它支持这类应用，同时保留MapReduce 的可扩展性和容错性。为了实现这些目标，Spark 引入了弹性分布式数据集 (RDD) 。RDD 是在一组可分区的只读对象集合，支持分区数据丢失重建。

介绍

现有的大规模数据解决方案（主要指MapReduce）针对如下两类问题时，存在着局限性：

> 1）迭代式作业：虽然每次迭代可以都可以表示一个MR任务，但是每一次迭代必须从磁盘加载数据；

> 2）交互式数据分析：SQL虽然也可以转换成MR任务，但是每一次MR任务都要从磁盘加载数据。

针对上述问题，本文提出了一种新的大规模数据计算方案Spark，弹性分布式数据集（RDD）可以用来解决迭代式作业的问题；而Spark是基于Scala进行构建的，而Scala可以提供交互式的操作，可以很好的解决交互式的数据分析。

02

Programming Model

1. 弹性分布式数据集（RDD）
2. 共享变量

Framework of Spark

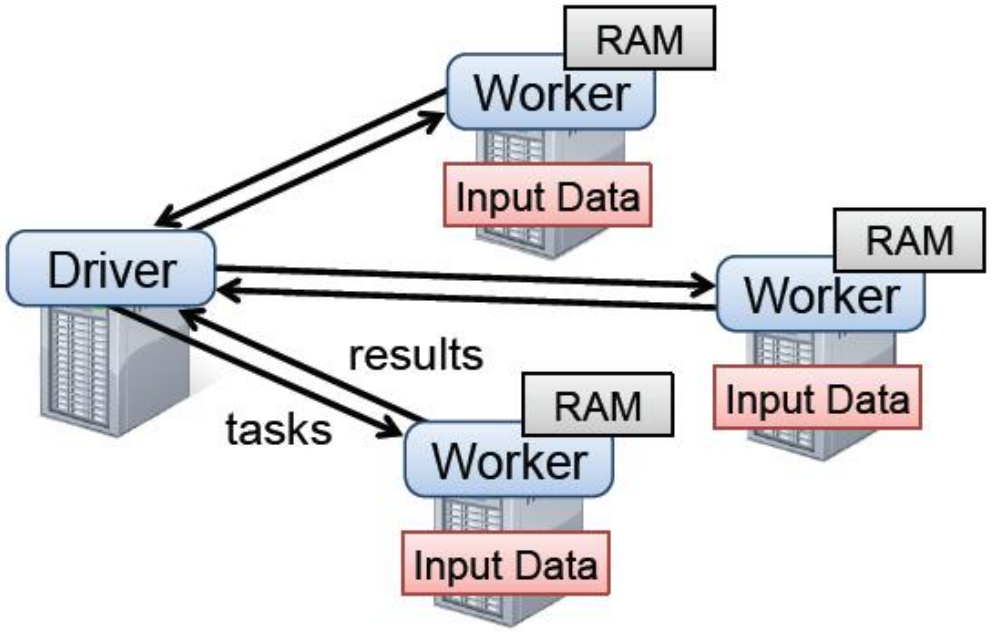

Figure 2: Spark runtime. The user's driver program launches multiple workers, which read data blocks from a distributed file system and can persist computed RDD partitions in memory.

Programming Model

驱动程序（Driver）：为了使用Spark，开发人员需要编写驱动程序，它的作用是控制应用程序的执行流程并在并行的环境中执行一系列的并行操作

弹性分布式数据集（RDD）

并行操作：可以在RDD上执行一系列的并行操作

共享变量：可以在集群上运行的函数中使用

2.1 弹性分布式数据集（RDD）

RDD是什么：分布式内存资源抽象

RDD（Resilient Distributed Dataset，弹性分布式数据集）本质上是一种只读、分片的记录集合，只能由所支持的数据源或是由其他 RDD 经过一定的转换（Transformation）来产生。通过由用户构建 RDD 间组成的产生关系图，每个 RDD 都能记录到自己是如何由还位于持久化存储中的源数据计算得出的，即其血统（Lineage）。

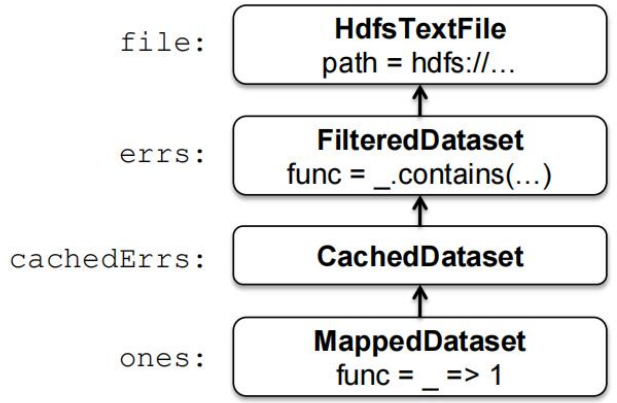

Figure 1: Lineage chain for the distributed dataset objects defined in the example in Section 4.

2.1 弹性分布式数据集（RDD）

一个RDD由以下部分组成：

1）其分片集合

2）其父 RDD 集合；

3）计算产生该 RDD 的方式

4）描述该 RDD 所包含数据的模式、分片方式、存储位置偏好等信息的元数据

2.1 弹性分布式数据集（RDD）

RDD与分布式共享内存的不同

相比于 RDD 只能通过粗粒度的“转换”来创建（或是说写入数据），分布式共享内存（Distributed Shared Memory，DSM）是另一种分布式系统常用的分布式内存抽象模型：应用在使用分布式共享内存时可以在一个全局可见的地址空间中进行随机的读写操作。类似的系统包括了一些常见的分布式内存数据库（如 Redis、Memcached）。

2.1 弹性分布式数据集（RDD）

RDD的特点：

1）跨计算机间的可分区的只读对象集合；

2）分区丢失之后可以重建（因为RDD不需要物化在物理存储上，相反可以通过物理存储上的数据来构建RDD）；

3）可以持久化RDD，供后续计算来使用。

2.1 弹性分布式数据集（RDD）

如何创建RDD？

1）从HDFS这样的分布式文件系统创建；

2）通过并行的读取Scala集合来创建；

3）从另一个RDD转化而来：Spark提供了map、flatMap等一系列的转换操作；

4）改变现有RDD的持久性。

2.1 弹性分布式数据集（RDD）

RDD默认是惰性并且临时的，但是可以通过特定的操作来改变其持久性：

1）Cache action：将数据保存在内存中，以便后期重用时，可以快速的使用。

2）Save action：将数据持久化到像HDFS这样的分布式文件存统上，这个被保存的版本也可以在后期的操作中重用。

2.2 共享变量

Spark提供了两种共享变量：

1）广播变量：这种变量只会被广播到每一个Worker一次；

2）累加器：可以在Worker节点间共享该变量，可以用来作为计数器。

03

Implementation

RDD 具体实现与计算调度

RDD 在物理形式上是分片的，其完整数据被分散在集群内若干机器的内存上。当用户创建出新的 RDD 后，新的 RDD 与原本的 RDD 便形成了依赖关系。根据用户所选 操作的不同，RDD 间的依赖关系可以被分为两种：

> 窄依赖：父 RDD 的每个分片至多被子 RDD 中的一个分片所依赖

> 宽依赖：父 RDD 中的分片可能被子 RDD 中的多个分片所依赖

RDD 具体实现与计算调度

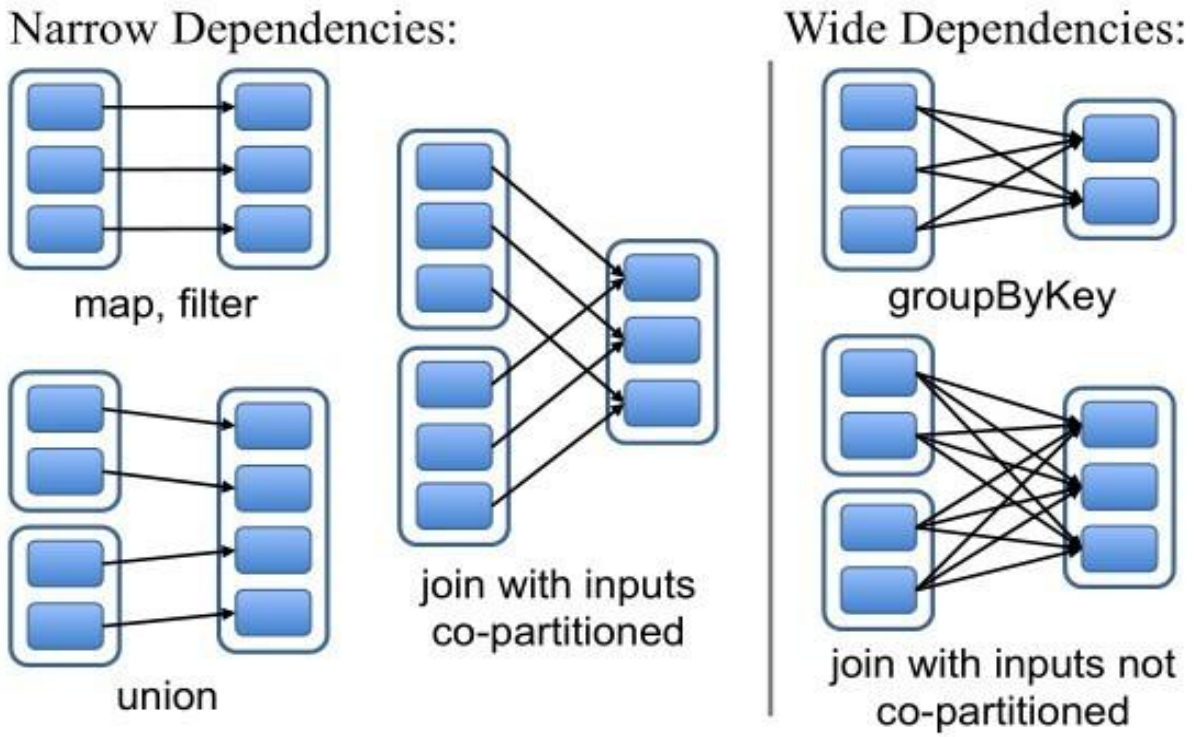

Figure 4: Examples of narrow and wide dependencies. Each box is an RDD, with partitions shown as shaded rectangles.

RDD 具体实现与计算调度

通过将窄依赖从宽依赖中区分出来，Spark 便可以针对 RDD 窄依赖进行一定的优化。首先，窄依赖使得位于该依赖链上的 RDD 计算操作可以被安排到同一个集群节点上流水线进行；其次，在节点失效需要恢复 RDD 时，Spark 只需要恢复父 RDD 中的对应分片即可，恢复父分片时还能将不同父分片的恢复任务调度到不同的节点上并发进行。

RDD 具体实现与计算调度

在用户调用 Action 方法触发 RDD 计算时，Spark 会按照定义好的 RDD 依赖关系绘制出完整的 RDD 血统图，并根据图中各节点间依赖关系的不同对计算过程进行切分：

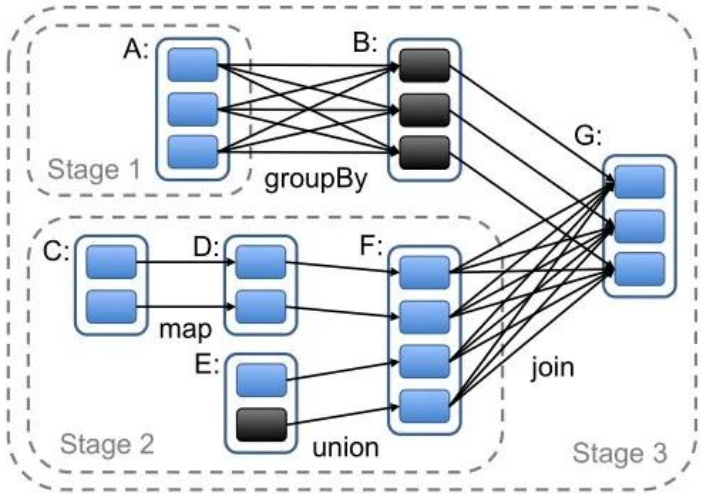

Figure 5: Example of how Spark computes job stages. Boxes with solid outlines are RDDs. Partitions are shaded rectangles, in black if they are already in memory. To run an action on RDD G, we build build stages at wide dependencies and pipeline narrow transformations inside each stage. In this case, stage 1's output RDD is already in RAM, so we run stage 2 and then 3.

RDD 具体实现与计算调度

Spark会把尽可能多的可以流水线执行的窄依赖 Transformation 放到同一个 Stage 中，而 Stage 之间则要求集群对数据进行 Shuffle。Job Stage 划分完毕后， Spark 便会为每个 Partition 生成计算任务（Task）并调度到集群节点上运行。

在调度 Task 时，Spark 也会考虑计算该 Partition 所需的数据的位置：例如，如果 RDD 是从 HDFS 中读出数据，那么 Partition 的计算就会尽可能被分配到持有对应 HDFS Block 的节点上；或者，如果 Spark 已经将父 RDD 持有在内存中，子 Partition 的计算也会被尽可能分配到持有对应父 Partition 的节点上。对于不同 Job Stage 之间的 Data Shuffle，目前 Spark 采取与 MapReduce 相同的策略，会把中间结果持久化到节点的本地存储中，以简化失效恢复的过程。

当 Task 所在的节点失效时，只要该 Task 所属 Job Stage 的父 Job Stage 数据仍可用，Spark 只要将该 Task 调度到另一个节点上重新运行即可。如果父 Job Stage 的数据也已经不可用了，那么 Spark 就会重新提交一个计算父 Job Stage 数据的 Task，以完成恢复。

04

Examples and Comparisons

文本搜索

```
val file = spark.textFile("hdfs://...")
val errs = file.filter(_.contains("ERROR"))
val cachedErrs = errs.cache()
val ones = cachedErrs.map(_ => 1)
val count = ones.reduce(_+_)
```

假设需要对存储在HDFS中的大型日志文件中包含的错误行进行统计。上面的代码示例使用Spark的方式实现了MapReduce操作。与MapReduce的操作不同的是，Spark可以保存中间数据。如果我们想保存errs数据，就可以使用val cachedErrs = errs.cache()这种方式创建一个缓存的RDD：这样如果后续我们需要读errs数据进行更多的操作，就会大大的提高执行效率了。

Spark相比于MR的优点

MapReduce	Spark
数据存储结构：磁盘HDFS文件系统的split	使用内存构建弹性分布式数据集RDD 对数据进行运算和cache
编程范式：Map + Reduce	DAG: Transformation + Action
计算中间结果落到磁盘，IO及序列化、反序列化代价大	计算中间结果在内存中维护 存取速度比磁盘高几个数量级
Task以进程的方式维护，需要数秒时间才能启动任务	Task以线程的方式维护 对于小数据集读取能够达到亚秒级的延迟

Spark相比于Hadoop的优点

- 减少磁盘I/O：HadoopMapReduce的map端将中间输出和结果存储在磁盘中，reduce端又需要从磁盘读写中间结果，势必造成磁盘IO成为瓶颈。Spark允许将map端的中间输出和结果存储在内存中，reduce端在拉取中间结果时避免了大量的磁盘I/O。
- 增加并行度：由于将中间结果写到磁盘与从磁盘读取中间结果属于不同的环节，Hadoop将它们简单的通过串行执行衔接起来。Spark把不同的环节抽象为Stage，允许多个Stage既可以串行执行，又可以并行执行。
- 可选的Shuffle排序：HadoopMapReduce在Shuffle之前有着固定的排序操作，而Spark则可以根据不同场景选择在map端排序或者reduce端排序。
- 灵活的内存管理策略：Spark将内存分为堆上的存储内存、堆外的存储内存、堆上的执行内存、堆外的执行内存4个部分。最大限度的提高资源的利用率，减少对资源的浪费。