【转】弹性分布式数据集：一种基于内存的集群计算的容错性抽象方法

原文出处：http://www.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-82.pdf

译文原处：http://bbs.sciencenet.cn/home.php?mod=space&uid=425672&do=blog&id=520947

摘要：本文提出了弹性分布式数据集（RDD，Resilient Distributed Datasets），这是一种分布式的内存抽象，允许在大型集群上执行基于内存的计算（In-Memory Computing），与此同时还保持了MapReduce等数据流模型的容错特性。现有的数据流系统对两种应用的处理并不高效：一是迭代式算法，这在图应用和机器学习领域很常见；二是交互式数据挖掘工具。这两种情况下，将数据保存在内存中能够极大地提高性能。为了有效地实现容错，RDD提供了一种高度受限的共享内存，即RDD是只读的，并且只能通过其他RDD上的批量操作来创建。尽管如此，RDD仍然足以表示很多类型的计算，包括MapReduce和专用的迭代编程模型（如Pregel）等。我们实现的RDD在迭代计算方面比Hadoop快二十多倍，同时还可以在5-7秒的延时内交互式地查询1TB的数据集。

1. 引言

无论是工业界还是学术界，都已经广泛使用高级集群编程模型来处理日益增长的数据，如MapReduce和Dryad。这些系统将分布式编程简化为自动提供位置感知性（locality-aware）调度、容错以及负载均衡，使得大量用户能够在商用集群上分析庞大的数据集。

大多数现有的集群计算系统都是基于非循环的数据流模型（acyclic data flow model）。从稳定的物理存储（如分布式文件系统）中加载记录，一组确定性操作构成一个DAG，记录被传入这个DAG，然后写回稳定存储。通过这个DAG数据流图，运行时自动完成调度工作及故障恢复。

尽管非循环数据流是一种很强大的抽象方法，但仍然有些应用无法使用这种方式描述。我们就是针对这些不太适合非循环模型的应用，它们的特点是在多个并行操作之间重用工作数据集（working set）。这类应用包括：（1）机器学习和图应用中常用的迭代算法（每一步对数据执行相似的函数）；（2）交互式数据挖掘工具（用户反复查询一个数据子集）。基于数据流的架构并不明确支持工作集，所以需要将数据输出到磁盘然后在每次查询时重新加载，从而带来较大的开销。

我们提出了一种分布式的内存抽象，称为弹性分布式数据集（RDD，Resilient Distributed Datasets），支持基于工作集的应用，同时具有数据流模型的特点：即自动容错、位置感知性调度和可伸缩性。RDD允许用户在执行多个查询时显式地将工作集缓存在内存中，极大地加速了后期的工作集重用。

RDD提供了一种高度受限的共享内存方式，即RDD是只读的记录分区的集合，只能通过对其他RDD执行确定性的转换操作（如map，join和group by）而创建。这些限制保证了低开销的容错性。与分布式共享内存系统需要高成本的检查点（checkpoint）和回滚（rollback）不同，RDD通过血统（lineage）来重建丢失的分区：RDD中包含如何从其他RDD衍生（即计算）出本RDD所需的相关信息，这样不需要检查点操作就可以重新构建丢失的数据分区。尽管RDD不是一个普适的共享内存抽象，但却具备了良好的描述能力、可伸缩性和可靠性，非常适合大多数数据并行型应用。

第一个指出非循环数据流存在不足的并不是我们。例如，Google的Pregel，是一种专门用于迭代式图算法的编程模型；Twister和HaLoop，是两种典型的迭代式MapReduce模型。但是，这些系统都是针对特定类型应用的。相比之下，RDD则为基于工作集的应用提供了更为通用的抽象。用户可以对中间结果进行显式的命名和物化（materialize），控制其分区，然后使用它们执行特定的用户操作（而不是让运行时去循环执行一系列MapReduce步骤）。RDD可以用来描述Pregel、迭代式MapReduce，以及这两种模型无法描述的其他应用，如交互式数据挖掘工具（用户将数据集装入RAM，然后执行ad-hoc查询）。

昵称：vincent_hv
园龄：10个月
粉丝：7
关注：1
+加关注

< 2013年10月 >

日	一	二	三	四	五	六
29	30	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	1	2
3	4	5	6	7	8	9

搜索
找找看
谷歌搜索

常用链接
我的随笔
我的评论
我的参与
最新评论
我的标签
更多链接

最新随笔
1. linux解压zip乱码解决方案
2. 全能系统监控工具dstat
3. 【转】linux sar命令详解
4. 【原】gnome3增加自定义程序快捷方式
5. 【原】Ubuntu13.04安装、卸载Gnome3.8
6. 【原】安装、卸载、查看软件时常用的命令
7. 【原】中文Ubuntu主目录下的文档文件夹改回英文
8. 【原】Ubuntu ATI/Intel双显卡 驱动安装
9. 【原】Ubuntu 12.04 ATI显卡设置双屏显示
10. 【转】Hadoop vs Spark性能对比

随笔分类
Android(8)
Hadoop(2)
Java(20)
JVM(3)
Linux(23)
others(1)
Scala(5)
Spark(20)
数据结构与算法(2)

我们实现的RDD称为Spark，用于开发多种并行应用。Spark采用Scala语言实现，提供类似于DryadLINQ的集成语言编程接口，方便用户编写并行任务。此外，通过修改Scala解释器，Spark还可用于交互式查询大数据集。我们相信Spark是第一个允许集群上对大数据集进行交互式分析的有效、通用的编程语言框架。

我们通过微基准（microbenchmark）和用户应用程序来评估RDD。实验表明，在处理迭代式应用上Spark比Hadoop快高达20多倍，数据分析报表的性能提高了40多倍，同时能够在5-7秒的延时内交互式扫描1TB的数据集。此外，我们还在Spark之上实现了Pregel和HaLoop编程模型（包括其位置优化策略，placement optimization），以库的形式实现（分别使用了100和200行Scala代码）。最后，利用RDD内在的确定性特性，我们还创建了一种Spark调试工具rddbg，允许用户在任务期间利用血统关系（lineage）重建RDD，然后像传统调试器那样重新执行任务。

本文首先在第2部分介绍RDD的概念，然后第3部分描述Spark API，第4部分解释如何使用RDD表示几种并行应用（包括Pregel和HaLoop），第5部分讨论Spark中RDD的表示方法以及任务调度器，第6部分描述具体实现和rddbg，第7部分对RDD进行评估，第8部分给出了相关研究工作，最后第9部分小结。

2. 弹性分布式数据集（RDD

本部分描述RDD和编程模型。首先讨论设计目标（2.1），然后定义RDD（2.2），接着讨论Spark的编程模型（2.3），并给出一个示例（2.4），最后将RDD与分布式共享内存进行比较（2.5）。

2.1 目标和概述

我们的目标是为基于工作集（working set）的应用（即多个并行操作重用中间结果的这类应用）提供抽象，同时保持MapReduce及其相关模型的优势特性：即自动容错、位置感知性调度和可伸缩性。RDD比数据流模型更易于编程，同时基于工作集的计算也具有良好的描述能力。

在这些特性中，最难实现的是容错性。一般来说，分布式数据集的容错性有两种方式：即数据检查点和记录数据的更新。我们面向的是大规模数据分析，数据检查点操作成本很高：需要通过数据中心的网络连接在机器之间复制庞大的数据集，而网络带宽往往比内存带宽低得多，同时还需要消耗更多的存储资源（在RAM中复制数据可以减少需要缓存的数据量，而存储到磁盘则会拖慢应用程序）。所以，我们选择记录更新的方式。但是，如果更新太多，那么记录更新成本也不低。因此，RDD只支持粗粒度转换（coarse-grained transformation），即在大量记录上执行的单个操作。将创建RDD的一系列转换记录下来（即lineage），以便恢复丢失的分区。

虽然只支持粗粒度转换限制了编程模型，但我们发现RDD仍然可以很好地适用于很多应用，特别是支持数据并行的批量分析应用，包括数据挖掘，机器学习，图算法等，因为这些程序通常都会在很多记录上执行相同的操作。RDD不太适合那些异步更新共享状态的应用，例如并行web爬行器。因此，我们的目标是为大多数分析型应用提供有效的编程模型，而其他类型的应用交给专门的系统。

2.2 RDD抽象

RDD是只读的记录分区的集合。RDD只能通过在——（1）稳定物理存储中的数据集；（2）其他已有的RDD——上执行确定性（deterministic）操作来创建。这些操作称之为转换（transformation），如map, filter, groupBy, join。（转换不是程序员在RDD上执行的操作。）

RDD不需要物化。RDD含有如何从其他RDD衍生（即计算）出本RDD的相关信息（即lineage），据此可以从物理存储的数据计算出相应的RDD分区（partition）。

2.3 编程模型

在Spark中，RDD被表示为对象，通过这些对象上的方法（或函数）调用转换（transformation）。

定义RDD之后，程序员就可以在行为（action）中使用RDD了。行为（action）是向应用程序返回值，或向存储系统导出数据的那些操作，例如，count（返回RDD中的元素个数），collect（返回元素本身），save（将RDD输出到存储系统）。在Spark中，只有在action第一次使用RDD时，才会计算RDD（即懒计算，lazily evaluated）。这样在构建RDD的时候，运行时以流水线的方式执行（pipeline）多个转换。

程序员还可以从两个方面控制RDD，即缓存（caching）和分区（partitioning）。用户可以请求将RDD缓存，这样运行时将已经计算好的RDD分区存储起来，以加速后期的重用。缓存的RDD一般存储在内存中，但如果内存不够，可以溢出（spill）到磁盘。

另一方面，RDD还允许用户根据关键字（key）指定分区顺序，这是一个可选的功能。目前支持哈希分区（hash partition）和范围分区（range partition）。例如，应用程序请求将两个RDD按照同样的哈希分区方式进行分区（将同一机器上具有相同关键字的记录放在一个分区），以加速它们之间的join操作。在Pregel和HaLoop中，多次迭代之间采用一致性的分区放置策略（Consistent partition placement）进行优化，我们同样也允许用户指定这种优化。

2.4 示例：控制台日志挖掘

本部分我们通过一个具体示例来阐述RDD。假定有一个大型网站出错，操作员想要检查Hadoop文件系统（HDFS）中的日志文件（TB级大小）来找出原因。通过使用Spark，操作员只需将日志中的错误信息装载到一组节点的RAM中，然后执行交互式查询。首先她需要在Spark解释器中敲入以下Scala命令：

```
lines = spark.textFile("hdfs://...")

errors = lines.filter(_.startsWith("ERROR"))

errors.cache()
```

第1行从HDFS文件定义了一个RDD（即一个文本行集合），第2行获得一个过滤后的RDD，第3行请求将errors缓存起来。注意在Scala语法中filter的参数是一个闭集（closure）。

这时集群还没有开始执行任何任务。但是，用户已经可以在这个RDD上执行action操作了，例如统计错误消息的数目：

```
errors.count()
```

用户还可以在RDD上执行更多的转换（transformation）操作，并使用转换结果，如：

```
// Count errors mentioning MySQL:

errors.filter(_.contains("MySQL")).count()

// Return the time fields of errors mentioning

// HDFS as an array (assuming time is field

// number 3 in a tab-separated format):

errors.filter(_.contains("HDFS"))

.map(_.split('\t')(3))

.collect()
```

使用errors的第一个action运行以后，Spark会把errors的分区缓存在内存中，极大地加速了后续计算。注意，最初的RDD lines不会被缓存。因为错误信息可能只占原数据集的很小一部分（小到足以放入内存）。

最后，为了说明模型的容错性，图1给出了第3个查询的血统（lineage）关系图。在lines RDD上执行filter操作，得到errors，然后再filter、map后得到新的RDD，在这个RDD上执行collect行为。Spark调度器以流水线的方式执行后两个转换，向拥有errors分区缓存的节点发送一组任务。此外，如果某个errors分区丢失，Spark只在相应的lines分区上执行filter操作来重建该errors分区。

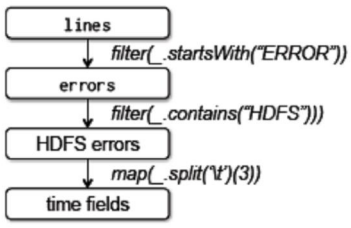


图1 示例中第三个查询的血统关系图。（方框表示RDD，箭头表示转换）

2.5 RDD与分布式共享内存

为了进一步理解RDD是一种分布式的内存抽象，表1列出了RDD与分布式共享内存（DSM，distributed shared memory）的对比。在DSM系统中，应用可以向全局地址空间的任意位置进行读写操作。（注意这里的DSM，不仅指传统的共享内存系统，还包括那些通过分布式哈希表或分布式文件系统进行数据共享的系统，比如Piccolo。）DSM是一种通用的抽象，但这种通用性同时也使得在商用集群上实现有效的容错性更加困难。

RDD与DSM主要区别在于，不仅可以通过批量转换创建（即"写"）RDD，还可以对任意内存位置读写。也就是说，RDD限制应用执行批量写操作，这样有利于实现有效的容错。特别地，RDD没有检查点开销，因为可以使用lineage来恢复RDD。而且，失效时只需要重新计算丢失的那些RDD分区，可以在不同节点上并行执行，而不需要回滚（roll back）整个程序。

对比项目	**RDD**	**分布式共享内存**
读	批量或细粒度操作	细粒度操作
写	批量转换操作	细粒度操作
一致性	不重要（RDD是不可更改的）	取决于应用程序或运行时
容错性	细粒度，低开销（使用lineage）	需要检查点操作和程序回滚
落后任务的处理	任务备份	很难处理
任务安排	基于数据存放的位置自动实现	取决于应用程序（通过运行时实现透明性）
如果内存不够	与已有的数据流系统类似	性能较差（交换？）

表1 RDD与分布式共享内存的对比

注意，通过备份任务的拷贝，RDD还可以处理落后任务（即运行很慢的节点），这点与MapReduce类似。而DSM则难以实现备份任务，因为任务及其副本都需要读写同一个内存位置。

与DSM相比，RDD模型有两个好处。第一，对于RDD中的批量操作，运行时将根据数据存放的位置来调度任务，从而提高性能。第二，对于基于扫描的操作，如果内存不足以缓存整个RDD，就进行部分缓存。把内存放不下的分区存储到磁盘上，此时性能与现有的数据流系统差不多。

最后看一下读操作的粒度。RDD上的很多行为（action，如count和collect）都是批量读操作，即扫描整个数据集，可以将任务分配到距离数据最近的节点上。同时，RDD也支持细粒度操作，即在哈希或范围分区的RDD上执行关键字查找。

3. Spark编程接口

Spark用Scala语言实现了RDD的API。Scala是一种基于JVM的静态类型、函数式、面向对象的语言。我们选择Scala是因为它简洁（特别适合交互式使用）、有效（因为是静态类型）。但是，RDD抽象并不局限于函数式语言，也可以使用其他语言来实现RDD，比如像Hadoop那样用类表示用户函数。

要使用Spark，开发者需要编写一个driver程序，连接到集群以运行worker，如图2所示。Driver定义了一个或多个RDD，并调用RDD上的行为（action）。Worker是长时间运行（long-lived）的进程，将RDD分区以Java对象的形式缓存在RAM中。

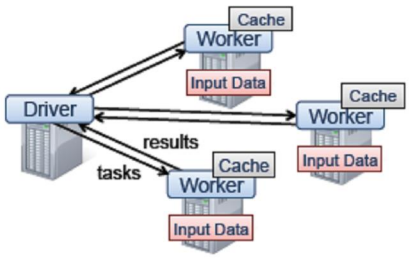


图2 Spark的运行时。用户的driver程序启动多个worker，worker从分布式文件系统中读取数据块（block），并将计算后的RDD分区（partition）缓存在内存中。

再看看2.4中的例子，用户执行RDD操作时会提供参数，比如map传递一个闭包（closure，函数式编程中的概念）。Scala将闭包表示为Java对象，如果传递的参数是闭包，则这些对象被序列化，

通过网络传输到其他节点上进行装载。Scala将闭包内的变量保存为Java对象的字段（field）。例如，var x = 5; rdd.map(_ + x) 这段代码将RDD中的每个元素加5。总的来说，Spark的语言集成类似于DryadLINQ。

RDD本身是静态类型对象，由参数指定其元素类型。例如，RDD[int]是一个整型RDD。不过，我们举的例子几乎都省略了这个类型参数，因为Scala支持类型推断。

虽然使用Scala实现RDD概念上很简单，但还是要处理一些Scala闭包对象的反射（reflection）问题。如何通过Scala解释器来使用Spark还需要更多工作，这点我们将在第6部分讨论。不管怎样，我们都不需要修改Scala编译器。

3.1 Spark中的RDD操作

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

表2列出了Spark中的RDD转换（transformation）和行为（action）。每个操作都给出了标识，其中方括号表示类型参数。前面说过转换是懒操作，用于定义新的RDD；而行为启动计算操作，并向用户程序返回值或向外部存储写数据。

注意，有些操作只对键值对（key-value pairs）可用，比如join。另外，函数名与Scala及其他函数式语言中的API匹配，例如map是一对一的映射，而flatMap是将每个输入映射为一个或多个输出（与MapReduce中的map类似）。

除了这些操作以外，用户还可以请求将RDD缓存起来。而且，用户还可以通过Partitioner类获取RDD的分区顺序，然后将另一个RDD按照同样的方式分区。有些操作会自动产生一个哈希或范围分区的RDD，像groupByKey，reduceByKey和sort等。

4. 应用程序示例

现在我们讲述如何使用RDD表示几种基于数据并行的应用。首先讨论一些迭代式机器学习应用（4.1），然后看看如何使用RDD描述几种已有的集群编程模型，即MapReduce（4.2），Pregel（4.3），和Hadoop（4.4）。最后讨论一下RDD不适合哪些应用（4.5）。

4.1 迭代式机器学习

很多机器学习算法都具有迭代特性，运行迭代优化方法来优化某个目标函数，例如梯度下降方法。如果这些算法的工作集（working set）能够放入RAM，将极大地加速程序运行。而且，这些算法通常采用批量操作，例如映射和求和，这样更容易使用RDD来表示。

例如下面的程序是逻辑回归的实现。逻辑回归是一种常见的分类算法，即寻找一个最佳分割两组点（即垃圾邮件和非垃圾邮件）的超平面w。算法采用梯度下降的方法：开始时w为随机值，在每一次迭代的过程中，对w的函数求和，然后朝着优化的方向移动w。

```
val points = spark.textFile(...)

.map(parsePoint).cache()

    var w = // random initial vector

for (i <- 1 to ITERATIONS) {

val gradient = points.map{ p =>

p.x * (1/(1+exp(-p.y*(w dot p.x)))-1)*p.y
```

```
}.reduce((a,b) => a+b)
w -= gradient
}
```

首先定义一个名为points的缓存RDD，这是在文本文件上执map转换之后得到的，即将每个文本行解析为一个Point对象。然后在points上反复执行map和reduce操作，每次迭代时通过对当前w的函数进行求和来计算梯度。7.1小节我们将看到这种在内存中缓存points的方式，比每次迭代都从磁盘文件装载数据并进行解析要快得多。

已经在Spark中实现的迭代式机器学习算法还有：kmeans（像逻辑回归一样每次迭代时执行一对map和reduce操作），期望最大化算法（EM，两个不同的map/reduce步骤交替执行），交替最小二乘矩阵分解和协同过滤算法。Chu等人提出迭代式MapReduce也可以用来实现常用的学习算法。

4.2 使用RDD实现MapReduce

MapReduce模型很容易使用RDD进行描述。假设有一个输入数据集（其元素类型为T），和两个函数myMap: T => List[(Ki, Vi)] 和 myReduce: (Ki; List[Vi])) List[R]，代码如下：

```
data.flatMap(myMap)
.groupByKey()
.map((k, vs) => myReduce(k, vs))
```

如果任务包含combiner，则相应的代码为：

```
data.flatMap(myMap)
.reduceByKey(myCombiner)
.map((k, v) => myReduce(k, v))
```

ReduceByKey操作在mapper节点上执行部分聚集，与MapReduce的combiner类似。

4.3 使用RDD实现Pregel

Pregel是面向图算法的基于批量同步并行模型（Bulk Synchronous Parallel paradigm）的编程模型。程序由一系列超步（superstep）协调迭代运行。在每个超步中，各个顶点执行用户函数，并更新相应的顶点状态，变异图拓扑，然后向下一个超步的顶点集发送消息。这种模型能够描述很多图算法，包括最短路径，双边匹配和PageRank等。

以PageRank为例介绍一下Pregel的实现。当前PageRank记为r，顶点表示状态。在每个超步中，各个顶点向其所有邻居发送贡献值（contribution）r/n，这里n是邻居的数目。下一个超步开始时，每个顶点将其分值（rank）更新为（公式无法显示，参见原文），这里的求和是各个顶点收到的所有贡献值的和，N是顶点的总数。

Pregel将输入的图划分（partition）到各个worker上，并存储在其内存中。在每个超步中，各个worker通过一种类似MapReduce的混排（shuffle）操作交换消息。

Pregel的通信模式可以用RDD来描述，如图3。主要思想是：将每个超步中的顶点状态和要发送的消息存储为RDD，然后根据顶点ID分组，进行混排通信（即cogroup操作）。然后对每个顶点ID上的状态和消息应用（apply）用户函数（即mapValues操作），产生一个新的RDD，即(VertexID, (NewState, OutgoingMessages))。然后执行map操作分离出下一次迭代的顶点状态和消息（即mapValues和flatMap操作）。代码如下：

```
val vertices = // RDD of (ID, State) pairs
val messages = // RDD of (ID, Message) pairs
val grouped = vertices.cogroup(messages)
val newData = grouped.mapValues {
(vert, msgs) => userFunc(vert, msgs)
// returns (newState, outgoingMsgs)
}.cache()
val newVerts = newData.mapValues((v,ms) => v)
val newMsgs = newData.flatMap((id,(v,ms)) => ms)
```

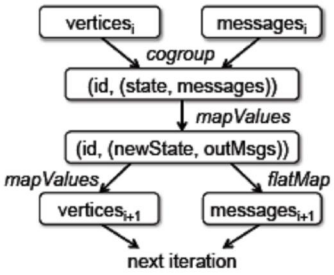
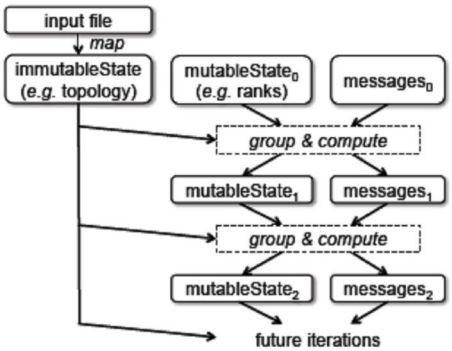


图3 使用RDD实现Pregel时，一步迭代的数据流。（方框表示RDD，箭头表示转换）

需要注意的是，这种实现方法中， RDD grouped，newData和newVerts的分区方法与输入RDD vertices一样。所以，顶点状态一直存在于它们开始执行的机器上，这跟原Pregel一样，这样就减少了通信成本。因为cogroup和mapValues保持了与输入RDD相同的分区方法，所以分区是自动进行的。

完整的Pregel编程模型还包括其他工具，比如combiner，附录A讨论了它们的实现。下面将讨论Pregel的容错性，以及如何在实现相同容错性的同时减少需要执行检查点操作的数据量。

我们差不多用了100行Scala代码在Spark上实现了一个类Pregel的API。7.2小节将使用PageRank算法评估它的性能。



5. RDD的描述及任务调度

我们希望在不修改调度器的前提下，支持RDD上的各种转换（transformation）操作，同时能够从这些转换获取lineage信息。为此，我们为RDD设计了一组小型通用的内部接口。

简单地说，每个RDD都包含：（1）一组RDD分区（partition，即数据集的原子组成部分）；（2）对父RDD的一组依赖，这些依赖描述了RDD的血统（lineage）；（3）一个函数，即在父RDD上执行何种计算；（4）元数据，描述分区模式和数据存放的位置。例如，一个表示HDFS文件的RDD包含：各个数据块（block）的一个分区，并知道各个数据块放在哪些节点上。而且这个RDD上的map操作结果也具有同样的分区，map函数是在父数据上执行的。表3总结了RDD的内部接口。

操作	**含义**
partitions()	返回一组Partition对象
preferredLocations(p)	根据数据存放的位置，返回分区p在哪些节点访问更快
dependencies()	返回一组依赖
iterator(p, parentIters)	按照父分区的迭代器，逐个计算分区p的元素
partitioner()	返回RDD是否hash/range分区的元数据信息

表3 Spark中RDD的内部接口

设计接口的一个关键问题就是，如何表示RDD之间的依赖。我们发现RDD之间的依赖关系可以分为两类，即：（1）窄依赖（narrow dependencies）：子RDD的每个分区依赖于常数个父分区（即

与数据规模无关）；（2）宽依赖（wide dependencies）：子RDD的每个分区依赖于所有父RDD分区。例如，map产生窄依赖，而join则是宽依赖（除非父RDD被哈希分区）。另一个例子见图5。

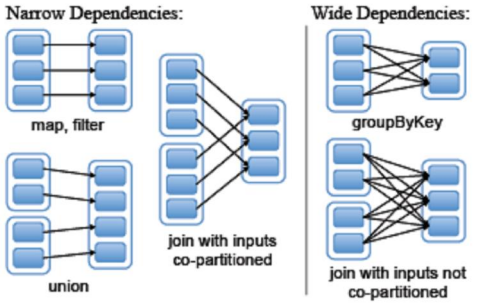


图5 窄依赖和宽依赖的例子。（方框表示RDD，实心矩形表示分区）

区分这两种依赖很有用。首先，窄依赖允许在一个集群节点上以流水线的方式（pipeline）计算所有父分区。例如，逐个元素地执行map、然后filter操作；而宽依赖则需要首先计算好所有父分区数据，然后在节点之间混排（shuffle），这与MapReduce类似。第二，窄依赖能够更有效地进行失效节点的恢复，即只需重新计算丢失RDD分区的父分区，而且不同节点之间可以并行计算；而对于一个宽依赖关系的血统（lineage）图，单个节点失效可能导致这个RDD的所有祖先丢失部分分区，因而需要整体重新计算。

通过RDD接口，Spark只需要不超过20行代码实现便可以实现大多数转换。5.1小节给出了例子，然后我们讨论了怎样使用RDD接口进行调度（5.2），最后讨论一下基于RDD的程序何时需要数据检查点操作（5.3）。

5.1 RDD实现举例

HDFS文件：目前为止我们给的例子中输入RDD都是HDFS文件，对这些RDD可以执行：partitions操作返回各个数据块的一个分区（每个Partition对象中保存数据块的偏移），preferredLocations操作返回数据块所在的节点列表，iterator操作对数据块进行读取。

Map：任何RDD上都可以执行map操作，返回一个MappedRDD对象。该操作传递一个函数参数给map，对父RDD上的记录按照iterator的方式执行这个函数，并返回一组符合条件的父RDD分区及其位置。

Union：在两个RDD上执行union操作，返回两个父RDD分区的并集。通过相应父RDD上的窄依赖关系计算每个子RDD分区。（注意union操作不会过滤重复值，相当于SQL中的UNION ALL）

Sample：抽样与映射类似，但是sample操作中，RDD需要存储一个随机数产生器的种子，这样每个分区能够确定哪些父RDD记录被抽样。

Join：对两个RDD执行join操作可能产生窄依赖（如果这两个RDD拥有相同的哈希分区或范围分区），可能是宽依赖，也可能两种依赖都有（比如一个父RDD有分区，而另一父RDD没有）。

5.2 Spark任务调度器

调度器根据RDD的结构信息为每个行为（action）确定有效的执行计划。调度器的接口是runJob函数，参数为RDD及其分区集，和一个RDD分区上的函数。该接口足以表示Spark中的所有行为（action，即count，collect，save等）。

总的来说，我们的调度器跟Dryad类似，但我们还考虑了哪些RDD分区是缓存在内存中的。调度器根据目标RDD的血统关系图（lineage graph）创建一个由stage构成的无回路有向图（DAG）。每个stage内部尽可能多地包含一组具有窄依赖关系的转换，并将它们流水线并行化（pipeline）。stage的边界有两种情况：一是宽依赖上的混排（shuffle）操作；二是已缓存分区，它可以缩短父RDD的计算过程。例如图6。父RDD完成计算后，可以在stage内启动一组任务计算丢失的分区。

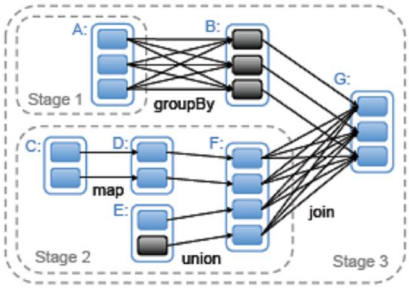


图6 Spark怎样划分任务阶段（stage）的例子。实线方框表示RDD，实心矩形表示分区（黑色表示该分区被缓存）。要在RDD G上执行一个行为（action），调度器根据宽依赖创建一组stage，并在每个stage内部将具有窄依赖的转换流水线化（pipeline）。本例不用再执行stage 1，因为B已经存在于缓存中了，所以只需要运行2和3。

调度器根据数据存放的位置分配任务，以最小化通信开销。如果某个任务需要处理一个已缓存分区，则直接将任务分配给拥有这个分区的节点。否则，如果需要处理的分区位于多个可能的位置（例如，由HDFS的数据存放位置决定），则将任务分配给这一组节点。

对于宽依赖（例如需要混排的依赖），目前的实现方式是，在拥有父分区的节点上将中间结果物化（materialize），简化容错处理，这跟MapReduce中物化map输出很像。

如果某个任务失效，只要stage中的父RDD分区可用，则只需在另一个节点上重新运行这个任务即可。如果某些stage不可用（例如，混排时某个map输出丢失），则需要重新提交这个stage中的所有任务来计算丢失的分区。

最后，lookup行为允许用户从一个哈希或范围分区的RDD上，根据关键字读取一个数据元素。这里有一个设计问题。Driver程序调用lookup时，只需要使用当前调度器接口计算关键字所在的那个分区。当然任务也可以在集群上调用lookup，这时可以将RDD视为一个大的分布式哈希表。这种情况下，任务和被查询的RDD之间的并没有明确的依赖关系（因为worker执行的是lookup），如果所有节点上都没有相应的缓存分区，那么任务需要告诉调度器计算哪些RDD来完成查找操作。

5.3 检查点

尽管RDD中的lineage信息可以用来故障恢复，但对于那些lineage链较长的RDD来说，这种恢复可能很耗时。例如4.3小节中的Pregel任务，每次迭代的顶点状态和消息都跟前一次迭代有关，所以lineage链很长。如果将lineage链存到物理存储中，再定期对RDD执行检查点操作就很有效。

一般来说，lineage链较长、宽依赖的RDD需要采用检查点机制。这种情况下，集群的节点故障可能导致每个父RDD的数据块丢失，因此需要全部重新计算。将窄依赖的RDD数据存到物理存储中可以实现优化，例如前面4.1小节逻辑回归的例子，将数据点和不变的顶点状态存储起来，就不再需要检查点操作。

当前Spark版本提供检查点API，但由用户决定是否需要执行检查点操作。今后我们将实现自动检查点，根据成本效益分析确定RDD 血统关系图（lineage graph）中的最佳检查点位置。

值得注意的是，因为RDD是只读的，所以不需要任何一致性维护（例如写复制策略，分布式快照或者程序暂停等）带来的开销，后台执行检查点操作。

6. 实现

我们使用10000行Scala代码实现了Spark。系统可以使用任何Hadoop数据源（如HDFS，Hbase）作为输入，这样很容易与Hadoop环境集成。Spark以库的形式实现，不需要修改Scala编译器。

这里讨论关于实现的三方面问题：（1）修改Scala解释器，允许交互模式使用Spark（6.1）；（2）缓存管理（6.2）；（3）调试工具rddbg（6.3）。

6.1 解释器的集成

像Ruby和Python一样，Scala也有一个交互式shell。基于内存的数据可以实现低延时，我们希望允许用户从解释器交互式地运行Spark，从而在大数据集上实现大规模并行数据挖掘。

通常Scala解释器将用户输入的每一行编译为一个类，装载到JVM中，然后执行函数。这个类是一个包含输入行变量或函数的单态（singleton）对象，并在一个初始化函数中运行这行代码。例如，如果用户敲入代码 var x = 5; println(x)，则解释器会定义一个包含x的Line1类，并将第2行编译为println(Line1.getInstance().x)。

在Spark中我们对解释器做了两点改动：

1. 类传输（class shipping）：解释器支持每行上创建的类以HTTP传输，这样worker节点就能获取这些类的字节码。

2. 改进的代码生成逻辑：通常每行上创建的单态对象通过对应类上的静态方法进行访问。也就是说，如果要序列化一个引用前面代码行变量的闭集（closure），比如上面的例子Line1.x，Java不会根据对象关系传输包含x的Line1实例。所以worker节点不会收到x。我们将这种代码生成逻辑改为直接引用各个行对象的实例。

图7说明了解释器如何将用户输入的一组代码行解释为Java对象。

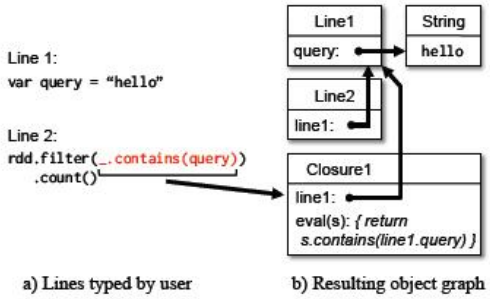


图7 Spark解释器如何将用户输入的两行代码解释为Java对象

Spark解释器便于处理大量对象关系引用（trace），并且有利于HDFS数据集的探究。我们计划以Spark解释器为基础，开发提供高级数据分析语言支持的交互式工具，比如SQL和Matlab的变种。

6.2 Cache管理

Worker节点将RDD分区以Java对象的形式缓存在内存中。由于大部分操作是基于扫描的，采取RDD级的LRU（最近最少使用）替换策略（即不会为了装载一个RDD分区而将同一RDD的其他分区替换出去）。目前这种简单的策略适合大多数用户应用。另外，使用带参数的cache操作可以设定RDD的缓存优先级。

6.3 rddbg：RDD程序的调试工具

RDD的初衷是为了支持容错的确定性再计算（re-computation），这个特性使得调试更容易。我们创建了一个名为rddbg的调试工具，使用程序记录的lineage信息，允许用户：（1）重建任何由程序创建的RDD，并执行交互式查询；（2）使用一个单进程Java调试器（如jdb）传入计算好的RDD分区，重新运行job中的任何任务。

我们强调一下rddbg不是一个完全重放的（replay）调试器：特别是不对非确定性的代码或行为进行重放。但如果某个任务一直运行很慢（比如由于数据分布不均匀或者异常输入等原因），仍然可以用它来帮助找到其中的逻辑错误和性能错误。

Rddbg给程序执行带来的开销很小。程序本来就需要将各个RDD中的所有闭包（closure）序列化并通过网络传送，只不过使用rddbg同时还要将这些闭集记录到磁盘。

7. 实验评估

我们在Amazon EC2上进行了一系列实验来评估Spark及RDD的性能，并与Hadoop及其他应用程序的基准（benchmark）进行了对比。总的说来，结果如下：

（1）对于迭代式机器学习应用，Spark比Hadoop快20多倍。这种加速比是因为：数据存储在内存中，同时Java对象缓存避免了反序列化操作（deserialization）。

（2）用户编写的应用程序执行结果很好。例如，Spark分析报表比Hadoop快40多倍。

（3）如果节点发生失效，通过重建那些丢失的RDD分区，Spark能够实现快速恢复。

（4）Spark能够在5-7s延时范围内，交互式地查询1TB大小的数据集。

分类: Spark

« 上一篇：【转】JVM（Java虚拟机）优化大全和案例实战
» 下一篇：【转】Spark源码分析之-Storage模块