摘要

本文提出了分布式内存抽象的概念——弹性分布式数据集（RDD，Resilient Distributed Datasets），它具备像MapReduce等数据流模型的容错特性，并且允许开发人员在大型集群上执行基于内存的计算。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20多倍，同时还可以在5-7秒内交互式地查询1TB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超大数据集。

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。从稳定的物理存储（如分布式文件系统）中加载记录，记录被传入由一组确定性操作构成的DAG，然后写回稳定存储。DAG数据流图能够在运行时自动实现任务调度和故障恢复。

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。这类应用包括：

- 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）；
- 交互式数据挖掘工具（用户反复查询一个数据子集）。基于数据流的框架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，这带来较大的开销。

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）。它支持基于工作集的应用，同时具有数据流模型的特点：自动容错、位置感知调度和可伸缩性。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，后续的查询能够重用工作集，这极大地提升了查询速度。

RDD提供了一种高度受限的共享内存模型，即RDD是只读的记录分区的集合，只能通过在其他RDD执行确定的转换操作（如map、join和group by）而创建，然而这些限制使得实现容错的开销很低。与分布式共享内存系统需要付出高昂代价的检查点和回滚机制不同，RDD通过Lineage来重建丢失的分区：一个RDD中包含了如何从其他RDD衍生所必需的相关信息，从而不需要检查点操作就可以重构丢失的数据分区。尽管RDD不是一个通用的共享内存抽象，但却具备了良好的描述能力、可伸缩性和可靠性，但却能够广泛适用于数据并行类应用。

第一个指出非循环数据流存在不足的并非是我们，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，对于一些特定类型的应用，这些系统提供了一个受限的通信模型。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，控制其分区，还能执行用户选择的特定操作（而不是在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（用户将数据集装入内存，然后执行ad-hoc查询）。

Spark是我们实现的RDD系统，在我们内部能够被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，使用户可以非常容易地编写并行任务。此外，随着Scala新版本解释器的完善，Spark还能够用于交互式查询大数据集。我们相信Spark会是第一个能够使用有效、通用编程语言，并在集群上对大数据集进行交互式分析的系统。

我们通过微基准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，计算数据分析类报表的性能提高了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），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部分介绍了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部分总结。

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），讨论Spark的编程模型（2.3），并给出一个示例（2.4），最后对比RDD与分布式共享内存（2.5）。

2.1 目标和概述

我们的目标是为基于工作集的应用（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（在内存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，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、分区记录的集合。RDD只能基于在稳定物理存储中的数据集和其他已有的RDD上执行确定性操作来创建。这些确定性操作称之为转换，如map、filter、groupBy、join（转换不是程开发人员在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。

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。

定义RDD之后，程序员就可以在动作中使用RDD了。动作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，只有在动作第一次使用RDD时，才会计算RDD（即延迟计算）。这样在构建RDD的时候，运行时通过管道的方式传输多个转换。

程序员还可以从两个方面控制RDD，即缓存和分区。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，可以写到磁盘上。

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。目前支持哈希分区和范围分区。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置换策略进行优化，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内存中，然后执行交互式查询。首先，需要在Spark解释器中输入如下Scala命令：

```
lines = spark.textFile("")

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
//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上执行对应的动作，例如统计错误

errors.count()
//用户还可以在RDD上执行更多的转换操作，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，极大地加快了后续计算速度。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Lineage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操作。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。

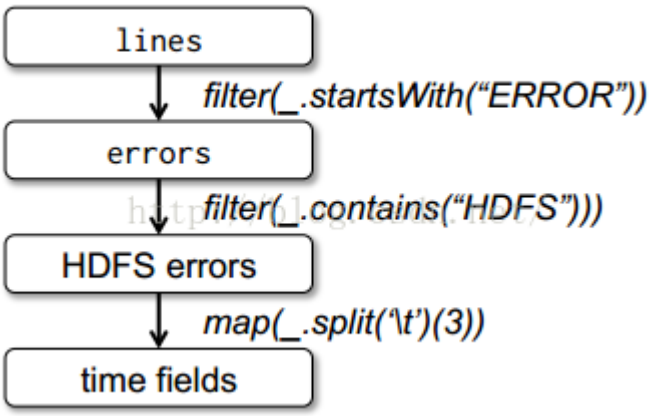

图1 示例中第三个查询的Lineage图。（方框表示RDD，箭头表示转换）

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，而不需要回滚整个程序。

表1 RDD与DSM对比

对比项目	RDD	分布式共享内存（DSM）
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？）

对比项目 RDD 分布式共享内存（DSM）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？）

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上的很多动作（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上的动作。Worker是长时间运行的进程，将RDD分区以Java对象的形式缓存在内存中。

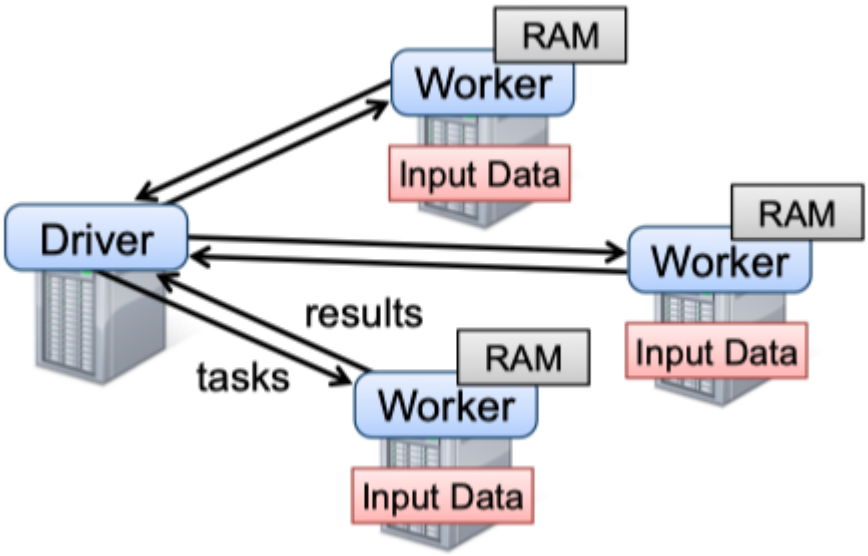

图2 Spark的运行时。用户的driver程序启动多个worker，worker从分布式文件系统中读取数据块，并将计算后的RDD分区缓存在内存中。

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。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。

虽然在概念上使用Scala实现RDD很简单，但还是要处理一些Scala闭包对象的反射问题。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

下边列出了Spark中的RDD转换和动作。每个操作都给出了标识，其中方括号表示类型参数。前面说过转换是延迟操作，用于定义新的RDD；而动作启动计算操作，并向用户程序返回值或向外部存储写数据。

转换

```
1 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[W]))]
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)]
```

动作

```
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
```

注意，有些操作只对键值对可用，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能够放入内存，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).persist()

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行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[12]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BSP范式的编程模型。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 α/N + (1 - α) * Σci，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Shuffle通信（即cogroup操作）。然后对每个顶点ID上的状态和消息应用用户函数（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。

4.3.1 Pregel容错

当前，Pregel基于检查点机制来为顶点状态及其消息实现容错[21]。然而作者是这样描述的：通过在其它的节点上记录已发消息日志，然后单独重建丢失的分区，只需要恢复局部数据即可。上面提到这两种方式，RDD都能够很好地支持。

通过4.3小节的实现，Spark总是能够基于Lineage实现顶点和消息RDD的重建，但是由于过长的Lineage链，恢复可能会付出高昂的代价。因为迭代RDD依赖于上一个RDD，对于部分分区来说，节点故障可能会导致这些分区状态的所有迭代版本丢失，这就要求使用一种“级联-重新执行”的方式去依次重建每一个丢失的分区。为了避免这个问题，用户可以周期性地在顶点和消息RDD上执行save操作，将状态信息保存到持久存储中。然后，Spark能够在失败的时候自动地重新计算这些丢失的分区（而不是回滚整个程序）。

最后，我们意识到，RDD也能够实现检查点数据的reduce操作，这要求通过一种高效的检查点方案来表达检查点数据。在很多Pregel作业中，顶点状态都包括可变与不可变的组件，例如，在PageRank中，与一个顶点相邻的顶点列表是不可变的，但是它们的排名是可变的，在这种情况下，我们可以使用一个来自可变数据的单独RDD来替换不可变RDD，基于这样一个较短的Lineage链，检查点仅仅是可变状态，图4解释了这种方式。

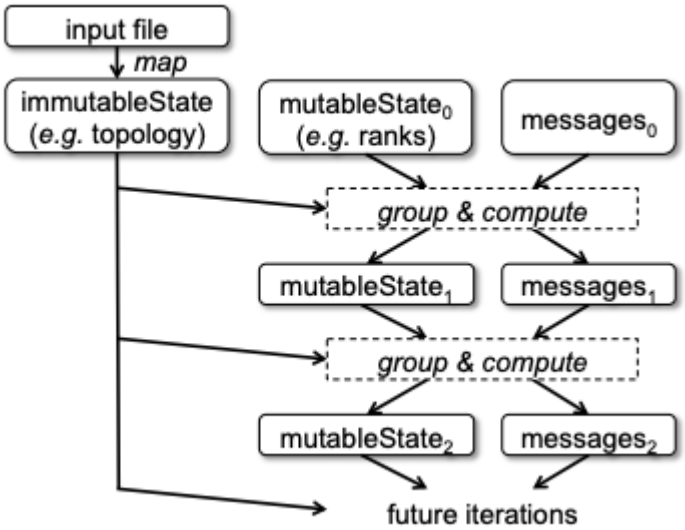

图4 经过优化的Pregel使用RDD的数据流。可变状态RDD必须设置检查点，不可变状态才可被快速重建。

在PageRank中，不可变状态（相邻顶点列表）远大于可变状态（浮点值），所以这种方式能够极大地降低开销。

4.4 使用RDD实现HaLoop

HaLoop[8]是Hadoop的一个扩展版本，它能够改善具有迭代特性的MapReduce程序的性能。基于HaLoop编程模型的应用，使用reduce阶段的输出作为map阶段下一轮迭代的输入。它的循环感知任务调度器能够保证，在每一轮迭代中处理同一个分区数据的连续map和reduce任务，一定能够在同一台物理机上执行。确保迭代间locality特性，reduce数据在物理节点之间传输，并且允许数据缓存在本地磁盘而能够被后续迭代重用。

使用RDD来优化HaLoop，我们在Spark上实现了一个类似HaLoop的API，这个库只使用了200行Scala代码。通过partitionBy能够保证跨迭代的分区的一致性，每一个阶段的输入和输出被缓存以用于后续迭代。

4.5 不适合使用RDD的应用

在2.1节我们讨论过，RDD适用于具有批量转换需求的应用，并且相同的操作作用于数据集的每一个元素上。在这种情况下，RDD能够记住每个转换操作，对应于Lineage图中的一个步骤，恢复丢失分区数据时不需要写日志记录大量数据。RDD不适合那些通过异步细粒度地更新来共享状态的应用，例如Web应用中的存储系统，或者增量抓取和索引Web数据的系统，这样的应用更适合使用一些传统的方法，例如数据库、RAMCloud[26]、Percolator[27]和Piccolo[28]。我们的目标是，面向批量分析应用的这类特定系统，提供一种高效的编程模型，而不是一些异步应用程序。

5. RDD的描述及作业调度

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§ 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

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。

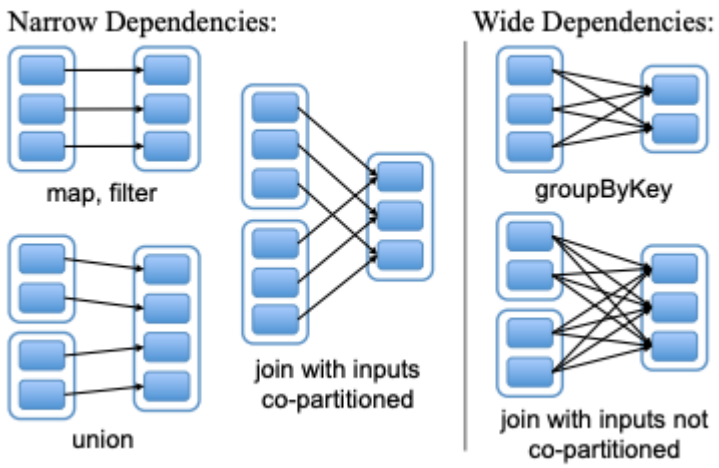

图5 窄依赖和宽依赖的例子。（方框表示RDD，实心矩形表示分区）

区分这两种依赖很有用。

- 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；
- 宽依赖则需要首先计算好所有父分区数据，然后在节点之间进行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的结构信息为每个动作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中的所有动作（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Lineage图创建一个由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。

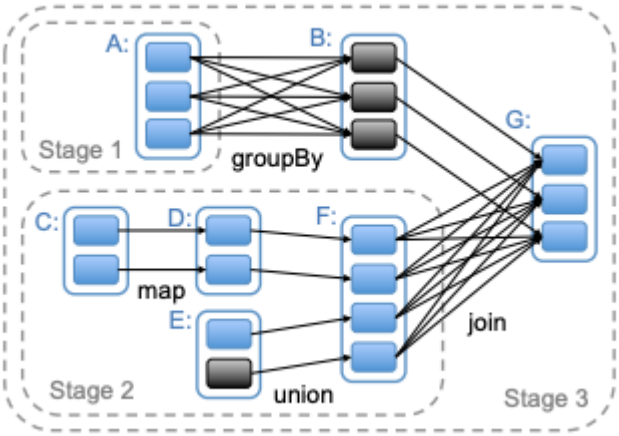

图6 Spark怎样划分任务阶段（stage）的例子。实线方框表示RDD，实心矩形表示分区（黑色表示该分区被缓存）。要在RDD G上执行一个动作，调度器根据宽依赖创建一组stage，并在每个stage内部将具有窄依赖的转换流水线化（pipeline）。 本例不用再执行stage 1，因为B已经存在于缓存中了，所以只需要运行2和3。

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Shuffle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，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，Shuffle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动作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p4.3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（checkpoint 机制），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 Lineage图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。

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. 实现

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。

Scala解释器通常根据将用户输入的代码行，来对类进行编译，接着装载到JVM中，然后调用类的函数。这个类是一个包含输入行变量或函数的单例对象，并在一个初始化函数中运行这行代码。例如，如果用户输入代码var x = 5，接着又输入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：

- 类传输：解释器能够支持基于HTTP传输类字节码，这样worker节点就能获取输入每行代码对应的类的字节码。
- 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，如果要序列化一个闭包，它引用了前面代码行中变量，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。

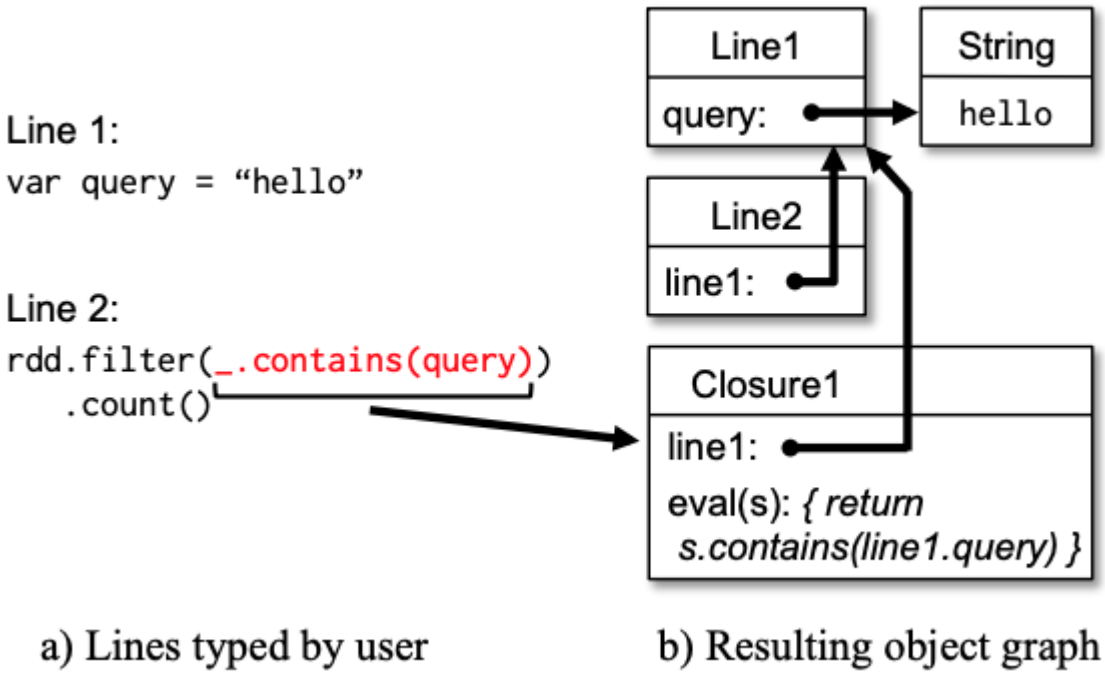

图7 Spark解释器如何将用户输入的两行代码解释为Java对象

Spark解释器便于跟踪处理大量对象关系引用，并且便利了HDFS数据集的研究。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，比如类似SQL和Matlab。

6.2 缓存管理

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。

8. 相关工作

分布式共享内存（DSM）。RDD可以看成是一个基于DSM研究得到的抽象。在2.5节我们讨论过，RDD提供了一个比DSM限制更严格的编程模型，并能在节点失效时高效地重建数据集。DSM通过检查点实现容错，而Spark使用Lineage重建RDD分区，这些分区可以在不同的节点上重新并行处理，而不需要将整个程序回退到检查点再重新运行。RDD能够像MapReduce一样将计算推向数据，并通过推测执行来解决某些任务计算进度落后的问题，推测执行在一般的DSM系统上是很难实现的。

In-Memory集群计算。Piccolo是一个基于可变的、In-Memory的分布式表的集群编程模型。因为Piccolo允许读写表中的记录，它具有与DSM类似的恢复机制，需要检查点和回滚，但是不能推测执行，也没有提供类似groupBy、sort等更高级别的数据流算子，用户只能直接读取表单元数据来实现。可见，Piccolo是比Spark更低级别的编程模型，但是比DSM要高级。

RAMClouds适合作为Web应用的存储系统，它同样提供了细粒度读写操作，所以需要通过记录日志来实现容错。

数据流系统。RDD借鉴了DryadLINQ、Pig和FlumeJava的“并行收集”编程模型，通过允许用户显式地将未序列化的对象保存在内存中，以此来控制分区和基于key随机查找，从而有效地支持基于工作集的应用。RDD保留了那些数据流系统更高级别的编程特性，这对那些开发人员来说也比较熟悉，而且，RDD也能够支持更多类型的应

用。RDD新增的扩展，从概念上看很简单，其中Spark是第一个使用了这些特性的系统，类似DryadLINQ编程模型，能够有效地支持基于工作集的应用。

面向基于工作集的应用，已经开发了一些专用系统，像Twister、HaLoop实现了一个支持迭代的MapReduce模型；Pregel，支持图应用的BSP计算模型。RDD是一个更通用的抽象，它能够描述支持迭代的MapReduce、Pregel，还有现有一些系统未能处理的应用，如交互式数据挖掘。特别地，它能够让开发人员动态地选择操作来运行在RDD上（如查看查询的结果以决定下一步运行哪个查询），而不是提供一系列固定的步骤去执行迭代，RDD还支持更多类型的转换。

最后，Dremel是一个低延迟查询引擎，它面向基于磁盘存储的大数据集，这类数据集是把嵌套记录数据生成基于列的格式。这种格式的数据也能够保存为RDD并在Spark系统中使用，但Spark也具备将数据加载到内存来实现快速查询的能力。

关系数据库。从概念上看，RDD类似于数据库中的视图，缓存RDD类似于物化视图。然而，数据库像DSM系统一样，允许典型地读写所有记录，通过记录操作和数据的日志来实现容错，还需要花费额外的开销来维护一致性。RDD编程模型通过增加更多限制来避免这些开销。

9. 总结

我们提出的RDD是一个面向，运行在普通商用机集群之上并行数据处理应用的分布式内存抽象。RDD广泛支持基于工作集的应用，包括迭代式机器学习和图算法，还有交互式数据挖掘，然而它保留了数据流模型中引人注目的特点，如自动容错恢复，处理执行进度落后的任务，以及感知调度。它是通过限制编程模型，进而允许高效地重建RDD分区来实现的。RDD实现处理迭代式作业的速度超过Hadoop大约20倍，而且还能够交互式查询数百G数据。