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）。

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。

（由于粘贴不便，图参见论文原文）

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	取决于应用程序或运行

	的）	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。

（由于粘贴不便，图参见论文原文）

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

表 2（参见论文原文）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)
```

（图参见原文）

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。

（此处涉及 Pregel 实现细节，略去，参见原文）

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。

（图参见原文）

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。

（图参见原文）

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。

（图参见原文）

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。

（具体实验细节参见原文）