弹性分布式数据集：内存集群计算的容错抽象

Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma,
Murphy McCauley, Michael J. Franklin, Scott Shenker, Ion Stoica
加州大学伯克利分校

摘要

我们提出的弹性分布式数据集（RDDs）是一种分布式内存抽象，可让程序员以容错的方式在大型集群上进行内存计算。目前的计算框架对两类应用的处理效率很低：迭代算法和交互式数据挖掘工具。在这两种情况下，将数据保存在内存中可以将性能提高一个数量级。为了高效地实现容错，RDD 提供了一种受限制的共享内存形式，它基于粗粒度转换，而不是对共享状态的细粒度更新。不过，我们证明 RDDs 的扩展性足以捕获广泛的计算类别，包括最近针对迭代作业的专门编程模型（如 Pregel），以及这些模型无法捕获的新应用。我们在一个名为 Spark 的系统中实现了 RDDs，并通过各种用户应用和基准测试对其进行了评估。

1 导言

MapReduce [10] 和 Dryad [19] 等集群计算框架已被广泛用于大规模数据分析。这些系统允许用户使用一组高级操作符编写并行计算，而无需担心工作分配和容错问题。

尽管当前的框架提供了大量访问集群计算再资源的方法，但它们缺乏利用分布式内存的抽象方法。这使得它们在一类重要的新兴应用中效率低下：那些在多次计算中*重复使用*不同结果的应用。数据重用在许多*迭代式*机器学习和图形算法中很常见，包括 PageRank、K-means 聚类和逻辑回归。另一个引人注目的用例是*交互式*数据挖掘，用户可在同一数据子集上运行多个临时查询。遗憾的是，在当前大多数框架中，在计算之间（*如*两个 MapRe- duce 作业之间）重用数据的唯一方法是将数据写入外部稳定存储系统，*如*分布式文件系统。由于数据复制、磁盘 I/O 和序列化等原因，这会产生大量开销。

这可能会影响应用程序的执行时间。

认识到这个问题后，研究人员为一些需要数据重用的应用开发了专门的框架。例如，Pregel [22] 是一个用于迭代图计算的系统，它将中间数据保存在内存中，而 HaLoop [7] 则提供了一个迭代 MapRe- duce 接口。不过，这些框架只支持特定的计算模式（*如*循环一系列 MapReduce 步骤），并针对这些模式隐式地执行数据共享。它们没有为更普遍的重复使用提供抽象，*例如*，让用户将多个数据集加载到内存中并在它们之间运行临时查询。

在本文中，我们提出了一种名为 "*弹性分布式数据集*"（*RDDs*）的新抽象概念，可在广泛的应用中实现高效的数据重用。RDDs 是一种容错的并行数据结构，用户可以将中间结果永久保存在内存中，控制其分区以优化数据放置，并使用丰富的运算符对其进行管理。

设计 RDD 的主要挑战在于定义一个能*有效*提供容错的编程接口。现有的集群内存存储抽象，如分布式共享内存 [24]、键值存储 [25]、数据库和 Piccolo [27]，提供的接口基于对可变状态（*如*表格中的单元格）的细粒度更新。使用这种接口，提供容错的唯一方法是跨机器复制数据或跨机器记录更新。对于数据密集型工作负载来说，这两种方法的成本都很高，因为它们需要在带宽远低于 RAM 的 Clus- ter 网络上复制大量数据，而且会产生大量存储开销。

与这些系统不同的是，RDD 基于*粗粒度*转换（*如*映射、过滤和连接），将相同的操作应用于许多数据项。这样，它们就可以通过记录用于构建数据集的转换（其*脉络*）而不是实际数据来有效地提供容错。[1]如果 RDD 的一部分丢失，RDD 有足够的信息说明它是如何从其他 RDD 派生的，从而可以重新计算

[1] 不过，当线程链变长时，在某些 RDD 中[检查点]数据可能会有用，我们将在*第5.4 节中*讨论如何做到这一点。

只是该分区。因此，丢失的数据通常可以很快恢复，而不需要昂贵的复制。

虽然基于粗粒度转换的接口乍看起来可能有局限性，但 RDDs 非常适合许多并行*应用，*因为*这些应用自然会对多个数据项应用相同的操作*。事实上，我们已经证明，RDDs 可以高效地表达迄今为止作为独立系统提出的许多集群编程模型，包括 MapReduuce、DryadLINQ、SQL、Pregel 和 HaLoop，以及这些系统无法捕捉的新应用，如交互式数据挖掘。我们认为，RDD 能够满足以前只能通过引入新框架才能满足的计算需求，这是 RDD 抽象强大功能的最可靠证明。

我们在一个名为 Spark 的系统中实现了 RDD，该系统正被用于加州大学伯克利分校和多家公司的研究和生产应用。Spark 在 Scala 编程语言[2]中提供了与 DryadLINQ [31] 类似的方便的语言集成编程界面。此外，Spark 还可用于从 Scala 解释器查询大型数据集。我们相信，Spark 是第一个允许在集群上以极快的速度使用通用编程语言进行内存数据挖掘的系统。

我们通过移动基准测试和用户应用测量来评估 RDD 和 Spark。我们发现，在迭代应用方面，Spark 比 Hadoop 快 20 倍；在实际数据分析报告方面，Spark 比 Hadoop 快 40 倍；在交互式扫描 1 TB 数据集时，Spark 的延迟时间仅为 5-7 秒。更重要的是，为了说明 RDD 的通用性，我们在 Spark 上实现了 Pregel 和 HaLoop 编程模型，包括它们所采用的布局优化，作为相对较小的程序库（各 200 行代码）。本文首先概述了 RDD（第 2 节）和 Spark（第 3 节）。然后，我们讨论 RDDs 的内部表示（§4）、我们的实现（§5）和实验结果（§6）。最后，我们将讨论 RDDs 如何捕捉现有的几种集群编程模型（§7），以及 RDDs 与 Spark 的关系（§8）。回顾相关工作（§8），并得出结论。

2 弹性分布式数据集（RDDs）

本节概述 RDD。我们首先对 RDD 进行细化（§2.1），并介绍其在 Spark 中的编程接口（§2.2）。然后，我们将 RDD 与更细粒度的共享内存抽象进行比较（§2.3）。最后，我们讨论 RDD 模型的局限性（§2.4）。

2.1 RDD 抽象

从形式上看，RDD 是只读、分区的记录集合。RDD 只能通过对 (1) 稳定存储中的数据或 (2) 其他 RDD 进行确定性操作来创建。我们将这些操作称为*转换*

将它们与 RDD 上的其他操作区分开来。转换的例子包括 *map*、*filter* 和 *join*[2]RDD 在任何时候都不需要实体化。相反，一个 RDD 有足够的信息说明它是如何从其他数据集衍生出来的（它的*脉络*），以便从稳定存储中的数据*计算*它的分区。这是一个功能强大的特性：从本质上讲，程序无法引用一个 RDD 数据集。故障后无法重建的 RDD。

最后，用户可以控制 RDD 的另外两个方面：*持久性*和*分区*。用户可以指出他们将重复使用哪些 RDD，并为其选择存储策略（*如*内存存储）。用户还可以要求根据每条记录中的键将 RDD 的元素划分到不同的机器上。这对于布局操作非常有用，例如确保两个要连接在一起的数据集以相同的方式进行哈希分区。

2.2 Spark 编程接口

Spark 通过与 DryadLINQ [31] 和 FlumeJava [8] 类似的语言集成应用程序接口公开 RDD，其中每个数据集都表示为一个对象，并使用这些对象上的方法调用转换。

程序员首先要通过对稳定存储中的数据进行转换（*如映射*和*过滤*）来定义一个或多个 RDD。然后，他们就可以在*操作*中使用这些 RDD，这些操作可将值返回给应用程序或将数据导出到存储系统。操作的例子包括*计数*（返回数据集中元素的数量）、*收集*（返回元素本身）和*保存*（将数据集输出到存储系统）。与 DryadLINQ 一样，Spark 在首次将 RDD 用于操作时也会懒散地计算 RDD，以便进行管道转换。

此外，程序员还可以调用*持久*化方法，指出他们希望在未来的操作中重复使用哪些 RDD。Spark 会将持久化的 RDD 保存在内存中，但如果内存不足，它也会将 RDD 转移到磁盘上。用户还可以通过*持久化*标志（flags to *persist*）请求其他持久化策略，例如只在磁盘上存储 RDD 或跨机器复制 RDD。最后，用户可以为每个 RDD 设置持久化优先级，指定哪些内存数据应优先溢出到磁盘。

2.2.1 实例：控制台日志挖掘

假设一个网络服务出现错误，操作员希望搜索 Hadoop 文件系统 (HDFS) 中数兆字节的日志来查找原因。使用 Spark，操作员可以将日志中的错误信息加载到一组节点的 RAM 中，并以交互方式进行查询。她首先会键入以下 Scala 代码：

---

2 虽然单个 RDD 是不可变的，但可以通过多个 RDD 来表示数据集的多个版本，从而实现可变状态。我们将 RDD 设为不可变，是为了更容易描绘线性图，但如果将我们的抽象抽象为版本化数据集，并在线性图中跟踪版本，效果也是一样的。

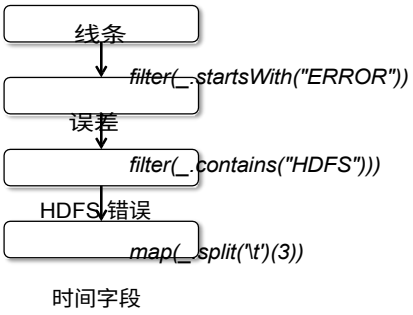


图 1：示例中第三个查询的脉络图。方框代表 RDD，箭头代表转换。

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
errors.persist()
```

第 1 行定义了一个由 HDFS 文件（作为文本行的集合）支持的 RDD，而第 2 行则从中导出了一个经过过滤的 RDD。然后，第 3 行要求将错误持久化在内存中，以便在查询时共享。请注意，*filter 的*参数是闭包的 Scala 语法。

在这一点上，并没有对 Clus- ter 执行任何工作。不过，用户现在可以在操作中使用 RDD，*例如*计算消息的数量：

```
errors.count()
```

用户还可以对 RDD 进行进一步转换，并使用转换结果，如下面几行所示：

```
// 对提及 MySQL 的错误进行计数：
errors.filter(_.contains("MySQL")).count()

// 返回错误的时间字段
// HDFS 作为数组（假设时间为字段
// 数字 3（以制表符分隔）：
errors.filter(_.contains("HDFS"))
      .map(_.split('\t')(3))
      .collect()
```

在涉及错误的第一个操作运行后，Spark 会将错误分区存储在内存中，从而大大加快后续计算速度。请注意，基础 RDD（行）*不会*加载到 RAM 中。这样做是可取的，因为错误信息可能只是数据的一小部分（小到足以装入内存）。

最后，为了说明我们的模型是如何实现容错的，我们在图 1 中展示了第三个查询中 RDD 的行图。

方面	RDDs	Distr.共享备忘录
读数	粗粒或细粒	细粒度
写道	粗粒	细粒度
一致性	琐碎（不可改变）	最多应用程序/运行时间
故障恢复	使用 lineage 实现精细化和低开销	需要检查点和程序回滚
减少落伍者	可能使用备份任务	困难
工作实习	根据数据位置自动	最多可用于应用程序（运行时力求透明）
内存不足时的行为	与现有数据流系统类似	性能不佳（交换？）

在这个查询中，我们以错误为起点，对行进行过滤，并应用了进一步的分析。

表 1：RDD 与分布式共享内存的比较。

2.3 RDD 模式的优势

为了了解 RDD 作为分布式内存抽象的优势，我们在表 1 中将其与分布式共享内存（DSM ）进行了比较。在 DSM 系统中，应用程序读写全局地址空间中的任意位置。请注意，根据这一定义，我们不仅包括传统的共享内存系统[24] ，还包括应用程序对共享状态进行细粒度写入的其他系统，包括提供共享 DHT 的 Piccolo [27] 和分布式数据库。DSM 是一个非常通用的抽象概念，但这种通用性使其更难在商品集群上以高效、容错的方式实现。

RDD 与 DSM 的主要区别在于，RDD 只能通在运行*采集*之前，请检查过滤器和地图。星火调度器将对后两种转换进行管道化处理，并向缓存错误分区的节点发送一组任务来计算这两种转换。此外，如果某个错误分区丢失，Spark 会仅对相应的行分区应用过滤器来重建该分区。

过粗粒度转换来创建（" 写入" ），而 DSM 允许对每个内存位置进行读写。[3] 这使得 RDDs 只能用于执行批量写入的应用程序，但却能实现更高效的容错。特别是，RDD 不需要承担检查点的开销，因为它们可以使用 lineage 恢复。[4] 此外，故障发生时只需重新计算 RDD 中丢失的分区，而且可以在不同节点上并行重新计算，无需回滚整个程序。

RDD 的第二个好处是，其不可变的结构允许系统通过运行慢速任务的备份（如 MapReduce [10] ）来缓解慢速节点（滞后者）的问题。使用 DSM 很难实现备份任务，因为任务的两个副本会访问相同的内存位置，并干扰彼此的更新。

最后，与 DSM 相比，RDD 还具有另外两个优势。首先，在对 RDD 进行批量操作时，运行时可以安排

[3]注意，对 RDD 的*读取*仍然可以是细粒度的。例如，应用程序可将 RDD 视为大型只读查找表。

[4] 正如我们在第 5.4 节中所讨论的，在某些应用中，对具有较长数据链的 RDD 进行检查点仍有帮助。不过，这可以在后台完成，因为 RDD 是不可变的，不需要像 DSM 那样对*整个*应用进行快照。

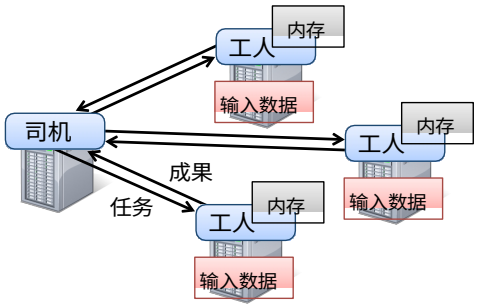


图 2：Spark 运行时。用户的驱动程序会启动多个 Worker，这些 Worker 会从分布式文件系统读取数据块，并将计算出的 RDD 分区保存在内存中。

根据数据的位置性来安排任务，以提高性能。其次，只要 RDD 仅用于基于扫描的操作，当内存不足以存储 RDD 时，RDD 会优雅地退化。无法在 RAM 中存储的分区可以存储在磁盘上，并提供与当前数据并行系统类似的性能。

2.4 不适合 RDD 的应用

如导言所述，RDD 最适合对数据集的所有元素应用相同操作的批处理应用。在这种情况下，RDD 可以将每次转换作为线性图中的一个步骤进行有效记忆，并且可以恢复丢失的分区，而无需记录大量数据。RDD 不太适合对共享状态进行异步细粒度更新的应用，例如网络应用的存储系统或增量网络爬虫。对于这些应用，使用执行传统更新日志和数据检查指导的系统会更有效，如数据库、RAMCloud [25]、Percolator 等。

[26] 和 Piccolo [27]。我们的目标是为批量分析提供高效的编程模型，并将这些异步应用留给专门的系统。

3 Spark 编程接口

Spark 通过与 Scala [2] 中 DryadLINQ [31] 类似的语言集成应用程序接口提供 RDD 抽象，Scala 是一种用于 Java VM 的静态类型函数式编程语言。我们选择 Scala 是因为它兼具简洁性（方便交互式使用）和高效性（静态类型）。但是，RDD 抽象并不需要函数式语言。

如图 2 所示，要使用 Spark，开发人员需要编写一个连接到 *Worker* 集群的*驱动程序*。驱动程序定义一个或多个 RDD，并对其进行调用。驱动程序上的 Spark 代码还能跟踪 RDD 的运行轨迹。Worker 是长寿命进程，可以在跨操作时将 RDD 分区存储在 RAM 中。正如我们在第 2.2.1 节的日志挖掘示例中所展示的，用户为 RDD 操作提供参数。

通过传递闭包（函数字面量），Scala 可以*将每个闭包表示为 Java 对象*，这些对象可以序列化并加载到另一个节点上。Scala 将每个闭包表示为一个 Java 对象，这些对象可以序列化并加载到另一个节点上，从而在网络上传递闭包。Scala 还会将闭包中绑定的任何变量保存为 Java 对象中的字段。例如，我们可以编写 `var x = 5; rdd.map(_ + x)` 这样的代码，为 RDD 的每个元素添加 5。[5]

RDD 本身是由元素类型参数化的静态类型对象。例如，RDD[Int] 是一个整数 RDD。不过，由于 Scala 支持类型推断，我们的大多数示例都省略了类型。虽然我们在 Scala 中公开 RDD 的方法在概念上很简单，但我们不得不使用反射来解决 Scala 闭包对象的问题 [33]。我们还需要做更多的工作，才能让 Spark 在 Scala 解释器中可用，这一点我们将在第 5.2 节中讨论。不过因此，我们无需修改 Scala 编译器。

3.1 Spark 中的 RDD 操作

表 2 列出了 Spark 中可用的主要 RDD 转换和操作。我们给出了每个操作的签名，并在方括号中显示了类型参数。我们再次指出，*转换*是定义新 RDD 的懒惰操作，而*操作*则是启动计算，向程序返回值或向外部存储写入数据。

请注意，某些操作（如 *join*）只能用于键值对的 RDD。例如，*map* 是一对一的映射，而 *flatMap* 则是将每个输入值映射到一个或多个输出（类似于 MapReduce 中的 map）。

除了这些操作符，用户还可以要求 RDD 持久化。此外，用户还可以获取 RDD 的分区顺序（由 Partitioner 类表示），并根据该顺序对另一个数据集进行分区。诸如 *groupByKey*、*reduceByKey* 和 *sort* 等操作会自动生成散列或范围分区的 RDD。

3.2 应用实例

我们将补充第 2.2.1 中的两个迭代应用：逻辑回归和 PageRank。后者还展示了对 RDD 分割的控制如何提高性能。

3.2.1 逻辑回归

许多机器学习算法都是迭代性质的，因为它们运行迭代优化程序（如梯度下降）来最大化某个函数。因此，通过将数据保存在内存中，它们的运行速度会更快。

举例来说，下面的程序实现了一种常见的分类算法--lo-gistic 回归[14]。

[5] 我们会在创建时保存每个闭包，因此即使 `x` 发生变化，本例中的*地图也*会始终加上 5。

变革	*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] （确定性采样） *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)] (保留分区) *sort*(*c* : Comparator[K]) : RDD[(K, V)] ⇒ RDD[(K, V)] *partitionBy*(*p* : Partitioner[K]) : RDD[(K, V)] ⇒ RDD[(K, V)]
行动	*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]（在散列/范围分割的 RDD 上） *save*(*path*：字符串)：将 RDD 输出到存储系统，*如* HDFS

表 2：Spark 中 RDD 上可用的转换和操作。Seq[T] 表示 T 类型元素的序列。

的超平面 *w*，它能将两组点（*如*垃圾邮件和非垃圾邮件）最好地分开。超平面

该算法使用梯度下降法：它将 *w* *设*为一个随机值，每次迭代时，它都会通过数据求和 *w* 的一个函数，使 *w* 向着能改善它的方向移动。

```
val points = spark.textFile(...)
                  .map(parsePoint).persist()
var w = // 随机初始向量
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 -= 梯度
}
```

我们首先定义一个名为 points 的持久化 RDD，作为对文本文件进行 *map* 转换的结果，该转换将每行文本解析为一个点对象。正如我们在第 6.1 节中所展示的，在迭代过程中将点保留在内存中可以提高 20 倍的速度。

3.2.2 网页排名

PageRank [6]中的数据共享模式更为复杂。该算法通过将链接到该文档的文档的贡献值相加，迭代更新每个文档的*排名*。每次迭代时，每个文档都会向其邻居发送 $\frac{r}{n}$ 的贡献，其中 r 是其等级和 n 是它的邻居数量。然后它更新

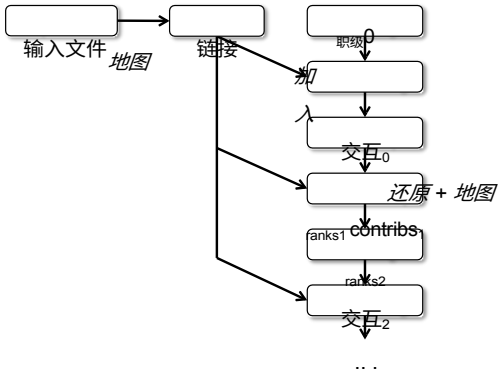


图 3：PageRank 数据集的世系图。

```
val links =
spark.textFile(...).map(...).persist() var ranks
= // RDD of (URL, rank) pairs
for (i <- 1 to ITERATIONS) {
  // 建立一个包含（targetURL、浮动）对的 RDD
  // 包含每个页面发送的贡献 val contribs =
  links.join(ranks).flatMap {
    (url, (links, rank)) =>
      links.map(dest => (dest, rank/links.size))
  }
  // 按 URL 对贡献求和，得到新的排名 ranks =
  contribs.reduceByKey((x,y) => x+y)
              .mapValues(sum => a/N + (1-a)*sum)
}
```

该程序可生成图中的 RDD 系谱图。

图 3。每次迭代，我们都会创建一个新的行列数据集根据上一次迭代的内容和等级

其等级为 $\alpha/N + (1 - \alpha) \sum c_i$ ，其中，总和为其收到的捐款，N 为其收到的捐款总数。

文件。我们可以用 Spark 将 PageRank 写成下面这样：

```
// 将图形加载为（URL、外链）对的 RDD
```

和静态链接数据集。[6]该图的一个有趣特点是，它随着链接数量的增加而变长。

[6]注意，虽然 RDD 是不可变的，但变量 `ranks` 和程序中的 `contribs` 在每次迭代中指向不同的 RDD。

的迭代次数。因此，在迭代次数较多的作业中，可能有必要可靠地复制行列的某些版本，以减少故障恢复时间[20]。用户可以调用带有 RELIABLE 标志的 *persist* 来实现这一点。不过，需要注意的是，链接数据集并不需要复制，因为可以通过在输入文件的块上重新运行*映射*，高效地重建链接数据集的分区。这个数据集通常比 rank 大得多，因为每个文件都有许多链接，但只有一个数字作为它的 rank，所以使用 lineage 恢复它比检查程序整个内存状态的系统更省时。

最后，我们可以通过控制 RDD 的*分区来*优化 PageRank 中的通信。如果我们为链接指定一个分区（*例如，*按 URL 对链接列表进行跨节点哈希分区），我们就可以用同样的方法对等级进行分区，并确保链接和等级之间的*连接*操作无需通信（因为每个 URL 的等级与其链接列表在同一台机器上）。我们还可以编写一个自定义的分区器类，将相互链接的页面分组在一起（*例如，*按域名对 URL 进行分区）。这两种优化方法都可以通过在定义链接时调用 partitionBy 来实现：

```
links = spark.textFile(...).map(...)
             .partitionBy(myPartFunc).persist()
```

首次调用后，链接和排名之间的*连接*操作将自动汇总每个 URL 对其链接列表所在机器的贡献，计算其在该机器上的新排名，并将其与链接连接起来。这种跨迭代的一致分区是 Pregel 等专业框架的主要优化之一。RDD 可以让用户直接表达这一目标。

4 表示 RDD

将 RDDs 作为一种抽象概念提供时，面临的挑战之一是为 RDDs 选择一种能在各种转换中跟踪其脉络的表示方法。理想情况下，实现 RDD 的系统应提供尽可能丰富的转换操作符集（*如*表 2 中的操作符），并允许用户以任意方式对其进行组合。我们为 RDDs 提出了一种简单的基于图的表示法，它有助于实现这些目标。我们在 Spark 中使用这种表示法来支持各种转换形式，而无需为每种转换形式在调度程序中添加特殊逻辑，从而大大简化了系统设计。

简而言之，我们建议通过一个通用接口来表示每个 RDD，该接口会公开五项信息：一组*分区，*即数据集的原子片段；一组对父 RDD 的*依赖关系*；一个根据分区计算数据集的函数；以及有关分区方案和数据放置的元数据。例如，代表 HDFS 文件的 RDD 为文件的每个块都有一个分区，并知道每个块在哪些机器上。同时，结果

运行	意义
分区()	返回分区对象列表
首选地点（*p*）	列出因数据位置性而可以更快访问分区 *p* 的节点
依赖关系()	返回依赖项列表
迭代器（*p*，*parentIters*）	计算分区 *p* 的元素 其父分区的迭代器
partitioner()	返回元数据，说明 RDD 是否进行了散列/范围分割

表 3：Spark 中用于表示 RDD 的接口。

的*映射*具有相同的分区，但在计算其元素时会将映射功能应用于父数据。我们在表 3 中总结了这一接口。

在设计这一界面时，最有趣的问题是如何表示 RDD 之间的依赖关系。我们发现，将依赖关系分为两种类型既充分又有用：*狭义*依赖关系，即父 RDD 的每个分区最多被子 RDD 的一个分区使用；*广义依赖关系*，即多个子分区可能依赖于父 RDD。例如，*map* 会导致窄依赖关系，而 *join* 会导致宽依赖关系（除非父 RDD 是散列分区）。图 4 显示了其他示例。

这种区分之所以有用，有两个原因。首先，窄依赖关系允许在一个 clus- ter 节点上流水线执行，该节点可以计算所有父分区。例如，可以在逐个元素的基础上应用*映射*，然后再应用*过滤器*。相比之下，宽依赖性要求所有父分区的数据都可用，并使用类似 MapReduce 的操作在各节点之间进行洗牌。其次，采用窄依赖关系时，节点故障后的恢复效率更高，因为只需要重新计算丢失的父分区，而且可以在不同节点上并行重新计算。相比之下，在具有广泛依赖性的系谱图中，单个节点故障可能会导致 RDD 所有祖先的某些分区丢失，从而需要重新执行整个过程。RDD 的这种通用接口使得在 Spark 中实现大多数转换只需不到 20 行代码。事实上，即使是新的 Spark 用户，也能在不知道调度的细节的情况下实现新的转换（*如*采样和各种类型的连接）。

我们将在下文中简要介绍一些 RDD 的实现方法。下面我们将简要介绍一些 RDD 的实现方法。

HDFS 文件：我们样本中的输入 RDD 都是 HDFS 中的文件。对于这些 RDD，*partitions* 会为文件的每个块返回一个分区（块的偏移量存储在每个 Partition 对象中），*preferredLocations* *会*给出块所在的节点，而 *iterator* *会*读取块。

***map*：**在任何 RDD 上调用 *map* *都会*返回一个 MappedRDD 对象。该对象具有与其父对象相同的分区和首选位置，但会应用传递给

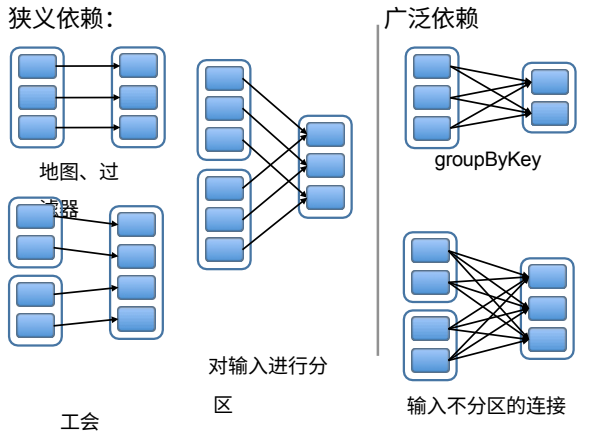


图 4：狭义和广义依赖关系示例。每个方框都是一个 RDD，分区显示为阴影矩形。

*映射*到父记录的*迭代器*方法。

union：在两个 RDD 上调用 *union* 会返回一个 RDD，其分区是父分区的联合。每个子分区都是通过相应父分区的窄延迟计算得出的。[7]

采样采样与映射类似，只是 RDD 为每个分区存储了一个随机数生成器种子，以确定性地采样父记录

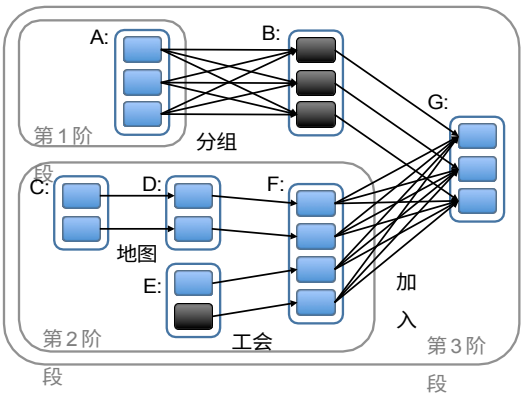
。

连接：将两个 RDD 连接起来可能会产生两个分支依赖关系（如果它们都使用相同的分区器进行哈希/范围分区）、两个宽依赖关系或混合依赖关系（如果一个父级 RDD 有分区器，而另一个没有）。无论哪种情况，输出 RDD 都有一个分区器（从父节点继承的分区器或默认哈希分区器）。

5 实施情况

我们用大约 14000 行 Scala 语言实现了 Spark。该系统在 Mesos 集群管理器[17] 上运行，允许它与 Hadoop、MPI 和其他应用程序共享资源。每个 Spark 程序都作为独立的 Mesos 应用程序运行，有自己的驱动程序（mas- ter）和工作者，这些应用程序之间的资源共享由 Mesos 处理。

Spark 可以使用 Hadoop 现有的输入插件 API 从任何 Hadoop 输入源（*如*HDFS 或 HBase）读取数据



，并在未修改的 Scala 版本上运行。现在，我们将简要介绍该系统在技术上令人感兴趣的几个部分：我们的作业调度器（§5.1）、允许交互式使用的 Spark 解释器（§5.2）、内存管理器（§5.3）、Hadoop 数据库（§5.4）、Scala 数据库（§5.5）。管理（§5.3），并支持检查点（§5.4）。

5.1 工作调度

Spark 的调度程序使用我们的 RDD 表示法，第 4 节对此进行了描述。

总的来说，我们的调度程序与 Dryad[19] 类似，但它还考虑到了每个数据包的分区。

[7]注意，我们的*合并*操作不会删除重复值。

图 5：Spark 如何计算作业阶段的示例。带有实线轮廓的方框是 RDD。分区为阴影矩形，如果已在内存中，则为黑色。要在 RDD G 上运行一个操作，我们要在广泛的依赖关系上建立构建阶段，并在每个阶段内进行管道转换。在本例中，阶段 1 的输出 RDD 已在 RAM 中，因此我们先运行阶段 2，然后再运行阶段 3。

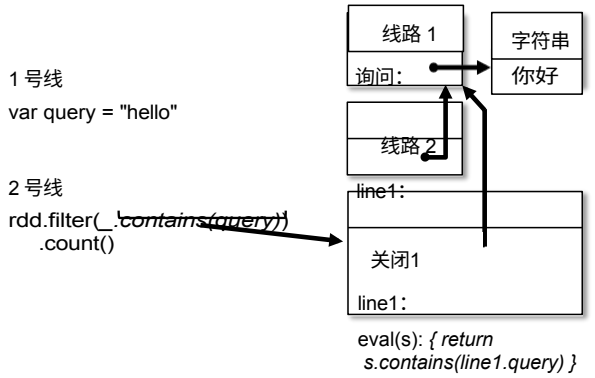
内存中可用的 RDD 是一致的。如图 5 所示，每当用户在 RDD 上运行一个操作（*如计数*或*保存*）时，调度程序就会检查该 RDD 的线程图，以建立一个要执行的*阶段* DAG。每个阶段都包含尽可能多的流水线转换，且依赖关系越窄越好。阶段的边界是宽依赖关系所需的洗牌操作，或任何已经计算过的分区，这些分区可以缩短父 RDD 的计算时间。然后，调度程序会启动任务，计算每个阶段中缺失的分区，直到计算出目标 RDD。

我们的调度器使用延迟调度技术（delay scheduling）[32]，根据数据的位置性将任务分配给机器。如果任务需要处理的分区在某个节点的内存中可用，我们就会将其发送到该节点。否则，如果任务要处理的分区包含的 RDD 提供了预置位置（*如* HDFS 文件），我们就会将其发送到这些节点。

对于广泛的依赖关系（*即*洗牌依赖关系），我们目前在持有父分区的节点上具体化中间记录，以简化故障恢复，就像 MapReduce 具体化地图输出一样。

如果某个任务失败，只要其阶段的父节点仍然可用，我们就会在另一个节点上重新运行该任务。如果某些阶段已经不可用（*例如，*由于洗牌的 "地图侧 "输出丢失），我们会重新提交任务，并行计算丢失的分区。我们目前还不能容忍调度程序出现故障，不过复制 RDD 系谱图将会非常简单。

最后，尽管目前 Spark 中的所有计算都是响应驱动程序中调用的操作而运行的，但我们也在尝试让 Clus- ter 上的任务（*如*地图）调用*查找*操作，该操作可按键随机访问散列分区 RDD 中的元素。在这种情况下，如果缺少所需的分区，任务需要告诉调度程序计算该分区。



a) 用户输入的行 b) 结果对象图

图 6：示例显示 Spark 解释器如何将用户输入的两行内容翻译成 Java 对象。

5.2 口译一体化

Scala 包含一个类似于 Ruby 和 Python 的交互式 shell。鉴于内存数据的低延迟性，我们希望让用户在解释器中主动运行 Spark，以查询大型数据集。

Scala 解释器通常为用户键入的每一行编译一个类，将其加载到 JVM 中，并在其上调用一个函数。该类包含一个单例对象，其中包含该行的变量或函数，并在 ini- tialize 方法中运行该行的代码。例如，如果用户输入 `var x = 5`，然后输入 `println(x)`，解释器就会定义一个包含 `x` 的类 `Line1`，并使第二行编译为 `println(Line1.getInstance().x)`。

我们对 Spark 的解释器做了两处修改：

1. *类运输：*为了让工作节点获取在每一行创建的类的字节码，我们让解释器通过 HTTP 为这些类提供服务。
2. *修改代码生成：*通常，为每行代码创建的单例对象都是通过其对应类上的静态方法访问的。这意味着，当我们序列化一个引用了前一行定义的变量的闭包（如上例中的 `Line1.x`）时，Java 不会通过对象图追踪到包裹 `x` 的 `Line1` 实例。

图 6 显示了解释器是如何将用户键入的一组行翻译成经过我们修改后的 Java 对象的。我们发现，Spark 解释器在处理研究过程中获得的大型跟踪数据和存储在 HDFS 中的数据集时非常有用。我们还计划用它来交互运行更高级别的查询语言，*例如* SQL。

作为序列化数据的内存存储和磁盘存储。第一种方案性能最快，因为 Java 虚拟机可以直接访问每个 RDD 元素。第二个选项允许用户在空间有限的情况下选择比 Java 对象图更节省内存的表示方法，但代价是性能降低。[8]第三个选项适用于 RDD 过大，无法保存在 RAM 中，但每次使用时重新计算成本较高的情况。

为了管理有限的可用内存，我们在 RDD 层面使用了 LRU 驱逐策略。当一个新的

5.3 内存管理

Spark 为持久化 RDD 的存储提供了三种选择：作为反序列化 Java 对象的内存存储、

当计算出 RDD 分区但没有足够空间存储时，我们会从最近获取最少的 RDD 中剔除一个分区，除非该 RDD 与新分区所在的 RDD 相同。在这种情况下，我们会将旧分区保留在内存中，以防止同一 RDD 中的分区循环进出。这一点很重要，因为大多数操作都会在整个 RDD 上运行任务，因此将来很可能需要内存中的分区。迄今为止，我们发现这一默认策略在所有应用中都运行良好，但我们也为每个 RDD 设置了 "持久性优先级"，让用户可以进一步控制。

最后，集群上的每个 Spark 实例目前都有自己独立的内存空间。在未来的工作中，我们计划研究如何通过统一的内存管理器在 Spark 实例间共享 RDD。

5.4 支持检查点

虽然在故障发生后，可以始终使用线程来恢复 RDD，但对于线程链较长的 RDD 来说，这种恢复可能会很耗时。因此，将某些 RDD 检查点到稳定存储区可能会有所帮助。

一般来说，检查点功能适用于具有包含广泛依赖关系的长行图的 RDD，例如 PageRank 示例中的等级数据集（第 3.2.2 节）。在这种情况下，集群中的节点故障可能会导致每个父 RDD 的某些数据片段丢失，从而需要重新进行全面计算[20]。相反，对于与稳定存储中的数据依赖性较小的 RDD，例如我们的逻辑回归示例（3.2.1 节）中的点和 PageRank 中的链接列表，检查点可能永远不值得。如果某个节点发生故障，这些 RDD 中丢失的分区可以在其他节点上并行重新计算，而成本仅为复制整个 RDD 的一小部分。

Spark 目前提供了用于检查点的 API（用于*持久化的*`REPLICATE` 标志），但由用户决定对哪些数据进行检查点。不过，我们也在研究如何自动执行检查点。由于我们的调度程序知道每个数据集的大小以及首次计算所花费的时间，因此它应该能够选择一组最佳的 RDD 进行检查点，从而最大限度地减少系统恢复时间[30]。

最后，请注意 RDD 的只读性质使得

[8] 成本取决于应用程序对每字节数据的计算量，但轻量级处理的成本可能高达 2 倍。

与一般共享内存相比，RDD 的检查点更简单。由于不需要考虑一致性问题，RDD 可以在后台写出，而不需要程序暂停或分布式快照方案。

6 评估

我们通过亚马逊 EC2 上的一系列实验以及用户应用程序的基准测试，对 Spark 和 RDD 进行了评估。总的来说，我们的结果如下：

- 在迭代式机器学习和图形应用中，Spark 的性能比 Hadoop 高出 20 倍。速度的提升来自于将数据以 Java 对象的形式存储在内存中，从而避免了 I/O 和反序列化成本。
- 我们的用户编写的应用程序性能和扩展性都很好。特别是，我们使用 Spark 将在 Hadoop 上运行的一份分析报告的速度提高了 40 倍。
- 当节点发生故障时，Spark 只需重新构建丢失的 RDD 分区，即可快速恢复。
- Spark 可用于查询 1 TB 数据集，间隔时间为 5-7 秒。

我们首先介绍了迭代计算的基准。
和 PageRank（第 6.2 节）。
与 Hadoop 进行比较。然后，我们评估了 Spark 中的故障恢复（§6.3）和数据集不适合内存时的行为（§6.4）。最后，我们讨论了用户应用程序（§6.5）和交互式数据挖掘（§6.6）的结果。

除非另有说明，我们的测试使用的是 4 核、15 GB 内存的 `m1.xlarge` EC2 节点。我们使用 256 MB 区块的 HDFS 进行存储。每次测试前，我们都会清除操作系统缓冲缓存，以便准确测量 IO 成本。

6.1 迭代式机器学习应用

我们实施了两种迭代式机器学习应用--逻辑回归和 K-均值，以比较以下系统的性能：

- *Hadoop*Hadoop 0.20.2 稳定版。
- *HadoopBinMem*：一种 Hadoop 部署，在第一次迭代中将输入数据转换成低开销的二进制格式，

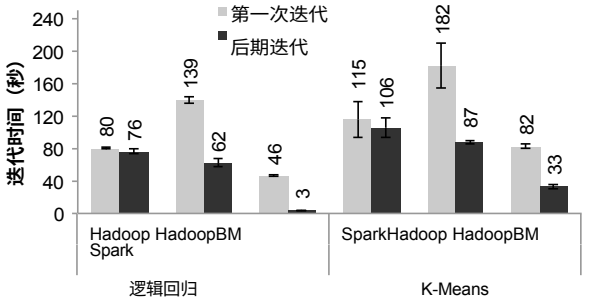
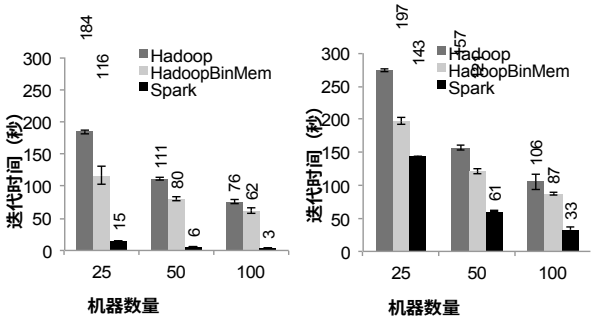


图 7：Hadoop、HadoopBinMem 和 Spark 在 100 节点集群上使用 100 GB 数据进行逻辑回归和 k-means 的第一次迭代和以后迭代的持续时间。



(a) 逻辑回归　　(b) K-Means

以消除后面的文本解析，并将其存储在内存中的 HDFS 实例中。

- *火花*我们的 RDDs 实现。

我们使用 25-100 台机器在 100 GB 数据集上对两种算法进行了 10 次迭代。这两种应用的主要区别在于每字节数据的计算量。K means 的迭代时间主要是计算时间，而 Logistic Re gression 的计算密集度较低，因此对反序列化和 I/O 所花费的时间更为敏感。

由于典型的学习算法需要数十次迭代才能收敛，我们分别报告了第一次迭代和后续迭代的时间。我们发现，通过 RDD 共享数据可大大加快未来的迭代速度。

图 8 ： Hadoop 、HadoopBinMem 和 Spark 中第一次迭代后的运行时间。所有作业的处理量均为 100 GB。

首次迭代 所有三个系统都在首次迭代中从 HDFS 读取文本输入。如图 7 中的浅色条所示，在所有实验中，Spark 比 Hadoop 稍微快一些。造成这一差异的原因是 Hadoop 的主程序和工作程序之间的心跳协议存在信号开销。HadoopBinMem 的速度最慢，因为它运行了一个额外的 MapReduce 作业将数据转换为二进制数据，并必须将这些数据通过网络写入复制的内存 HDFS 实例。

后续迭代 图 7 还显示了后续迭代的平均运行时间，而图 8 则显示了这些时间是如何随集群规模而缩放的。对于逻辑回归，在 100 台计算机上，Spark 分别比 Hadoop 和 HadoopBinMem 快 25.3 倍和 20.7 倍。对于计算量更大的 k-means 应用程序，Spark 仍然实现了 1.9 倍到 3.2 倍的速度提升。

了解速度提升 我们惊讶地发现，Spark 的性能甚至比使用二进制数据内存存储（HadoopBinMem ）的 Hadoop 还要高出 20 倍。在 HadoopBinMem 中，我们使用了 Hadoop 的标准二进制格式（SequenceFile ）和 256 MB 的大块大小，并强制 HDFS 的数据存储在内存文件系统中。然而，由于多种因素，Hadoop 运行速度仍然较慢：

1. 将 Hadoop 软件栈的开销降至最低、
2. HDFS 在提供数据时的开销，以及

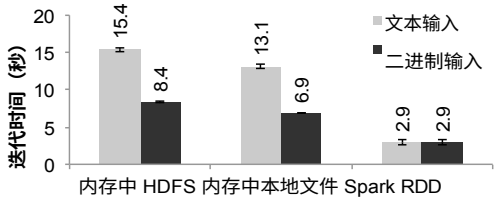


图 9：不同输入源在单机上使用 256 MB 数据进行逻辑回归的迭代时间。

3. 将二进制记录转换为可用内存 Java 对象的反序列化成本。

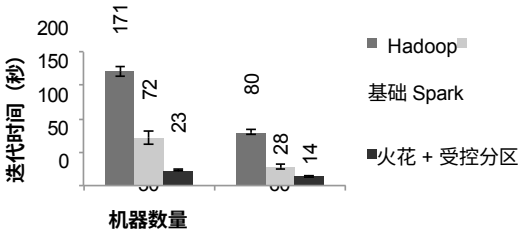
我们依次研究了这些因素。为了确定(1)，我们运行了无操作的 Hadoop 作业，发现这些作业至少需要 25 秒的开销才能完成作业设置、启动任务和清理等最低要求。关于 (2)，我们发现 HDFS 在为每个数据块提供服务时会执行多次内存复制和校验。

最后，为了测量（3），我们在单台机器上运行了微基准测试，对 256 MB 的各种格式输入进行逻辑回归计算。特别是，我们比较了处理来自 HDFS（HDFS 堆栈中的开销会体现出来）和内存中本地文件（内核可以非常高效地将数据传递给程序）的文本和二进制输入所需的时间。

图 9 显示了这些测试的结果。内存中的 HDFS 和本地文件之间的差异表明，通过 HDFS 读取数据会产生 2 秒钟的超时，即使数据在本地计算机的内存中。文本和二进制数据之间的差异表明，解析开销为 7 秒。最后，即使从内存文件中读取数据，将预先解析的二进制数据转换为 Java 对象也需要 3 秒钟，这几乎与逻辑回归本身一样昂贵。通过将 RDD 元素直接存储为内存中的 Java 对象，Spark 避免了所有这些开销。

6.2 网页排名

我们使用 54 GB 的维基百科数据集比较了 Spark 和 Hadoop 在 PageRank 方面的性能。我们运行了 10 次 PageRank 算法迭代，处理了约 400 万条文章的链接图。图 10 显示，与 30 个节点上的 Hadoop 相比，仅内存存储就为 Spark 带来了 2.4 倍的速度提升。



此外，如第 3.2.2 节所述，控制 RDD 的分区使其在各迭代中保持一致，可将速度提高到 7.4 倍。结果也几乎线性地扩展到了 60 个节点。

我们还评估了使用 Spark 上的 Pregel 实现编写的 PageRank 版本，我们将在第 7.1 节中介绍该版本。迭代时间与图 10 中的时间相似，但要长 4 秒左右，这是因为 Pregel 会在每次迭代时执行额外操作，让顶点 "投票 "决定是否完成工作。

图 10：PageRank 在 Hadoop 和 Spark 上的性能。

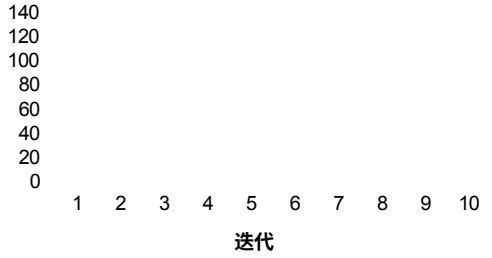


图 11：出现故障时 k-means 的迭代时间。其中一台机器在第 6 次迭代开始时发生故障，导致使用 lineage 重建了部分 RDD。

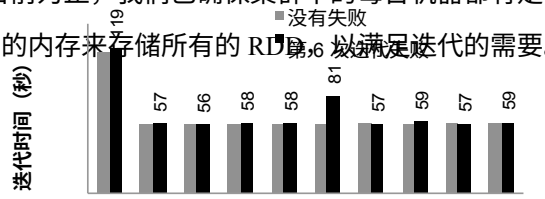
6.3 故障恢复

我们评估了在 k-means 应用程序中出现节点故障后使用 lineage 重建 RDD 分区的成本。图 11 比较了在正常运行情况下 75 节点集群上 10 次 k-means 迭代的运行时间，以及在第 6 次迭代开始时一个节点发生故障的运行时间。在没有任何故障的情况下，每次迭代由 400 个任务组成，处理 100 GB 的数据。

直到第 5 次迭代结束，迭代时间约为 58 秒。在第 6 次迭代中，其中一台机器被杀死，导致在该机器上运行的任务和存储在该机器上的 RDD 分区丢失。Spark 在其他机器上并行重新运行了这些任务，重新读取了相应的输入数据，并通过 lineage 重构了 RDD，从而将迭代时间延长到了 80 秒。一旦丢失的 RDD 分区得到重建，迭代时间就回落到 58 秒。需要注意的是，如果采用基于检查点的故障恢复机制，根据检查点的频率，恢复可能需要重新运行至少几次迭代。此外，系统还需要在 网络上复制应用程序的 100 GB 工作集（将文本输入数据转换为二进制数据），这将消耗两倍于 Spark 在 RAM 中复制的内存，或者需要等待将 100 GB 数据写入磁盘。相比之下，RDD 的线性图在我们的示例中，它们的大小都小于 10 KB。

6.4 记忆力不足时的行为

到目前为止，我们已确保集群中的每台机器都有足够的内存来存储所有的 RDD，以满足迭代的需要。



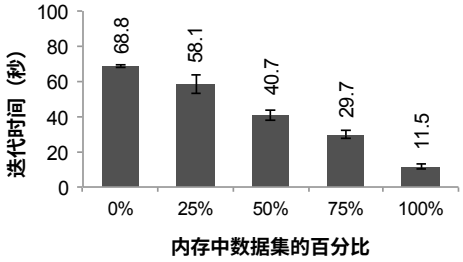
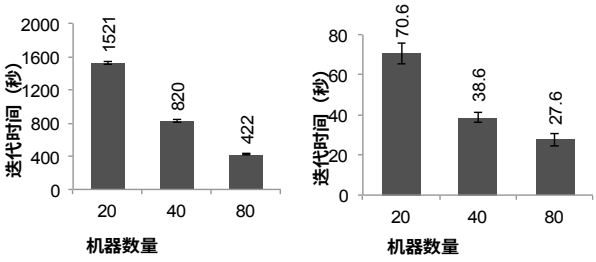


图 12：在 25 台机器上使用 100 GB 数据进行逻辑回归的性能，内存中的数据量各不相同。



(a) 交通建模　　(b) 垃圾邮件分类

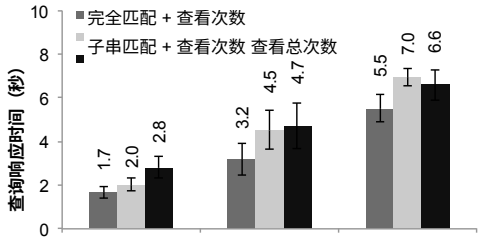
图 13：使用 Spark 实现的两个用户应用程序的每次迭代运行时间。误差条表示标准偏差。

问题。一个很自然的问题是，如果没有足够的内存来存储作业数据，Spark 该如何运行。在本实验中，我们配置 Spark 在每台机器上存储 RDD 时不使用超过一定百分比的内存。我们在图 12 中展示了逻辑回归使用不同存储空间的结果。我们可以看到，空间越小，性能下降得越慢。

6.5 使用 Spark 构建的用户应用程序

Conviva Inc 是一家视频分发公司，它使用 Spark 加速了许多以前在 Hadoop 上运行的数据分析报告。例如，一份报告以一系列 Hive [1] 查询的形式运行，计算客户的各种统计数据。这些查询都针对相同的数据子集（与客户提供的过滤器相匹配的记录），但在不同的分组字段上执行了分组（平均值、百分位数和 COUNT DISTINCT），需要单独的 MapReduce 作业。通过在 Spark 中执行查询并将它们之间共享的数据子集加载到 RDD 中，该公司能够将报告速度提高 40 倍。在 Hadoop 集群上需要 20 个小时才能完成的 200 GB 压缩数据报告，现在只需两台 Spark 机器就能在 30 分钟内完成。此外，Spark 程序只需要 96 GB 内存，因为它只在 RDD 中存储与客户文件相匹配的行和列，而不是整个解压缩文件。

交通建模 伯克利移动千年项目的研究人员[18]并行化了一种学习算法，用于根据汽车 GPS 测量数据推断道路交通拥堵情况。源数据是一个大都市地区的 10,000 条连接道路网络，以及装有 GPS 的汽车的 600,000 个点到点行程时间样本（每条路径的行程时间可能包括多条连接道路）。利用交通模型，该系统可以估算出单条道路的行程时间。研究人员使用期望最大化（EM）算法对该模型进行了训练，该算法反复进行两个*地图*和*减少*关键步骤。如图 13(a) 所示，该应用在 20 到 80 个节点（每个节点 4 个内核）之间几乎呈线性扩展。



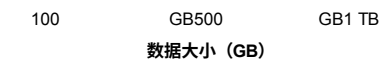


图 14：Spark 上交互式查询的响应时间，在 100 台机器上扫描越来越大的输入数据集。

Twitter 垃圾邮件分类 伯克利的 Monarch 项目 [29] 使用 Spark 识别 Twitter 消息中的垃圾链接。他们在 Spark 上实现了一个逻辑回归分类器，与第 6.1 节中的例子类似，但他们使用了分布式 *reduceByKey* 来并行求和梯度向量。图 13(b) 显示了在 50 GB 数据子集上训练分类器的扩展结果：250,000 个 URL 和 10^7 个与每个 URL 上网页的网络和内容属性相关的特征/维度。由于每次迭代的固定通信成本较高，因此扩展并不接近线性。

6.6 交互式数据挖掘

为了展示 Spark 交互式查询大型数据集的能力，我们用它分析了 1TB 的维基百科页面浏览日志（2 年的数据）。在该实验中，我们使用了 100 个 `m2.4xlarge` EC2 实例，每个实例有 8 个内核和 68 GB 内存。我们运行查询来查找以下页面的总浏览量：(1) 所有页面；(2) 标题与给定单词完全匹配的页面；(3) 标题与单词部分匹配的页面。每个查询都会扫描整个输入数据。

图 14 显示了对全部数据集以及一半和十分之一数据的查询响应时间。即使是 1 TB 的数据，在 Spark 上查询也需要 5-7 秒。这比使用磁盘数据快了不止一个数量级；例如，从磁盘查询 1 TB 的文件需要 170 秒。这说明，RDD 使 Spark 成为交互式数据挖掘的强大工具。

7 讨论

尽管 RDD 因其不可改变的性质和粗粒度转换而似乎提供了一个有限的编程界面，但我们发现它们适用于广泛的应用类别。特别是，RDD 可以表达数量惊人的集群编程模型，而这些模型迄今为止都是作为独立框架提出的，这样用户就可以在一个程序中*组合*这些模型（*例如，*运行 MapReduce 操作来构建一个图，然后在其上运行 Pregel），并在它们之间共享数据。在本节中，我们将讨论 RDD 可以表达哪些编程模型，以及它们为何如此广泛适用（第 7.1 节）。此外，我们还将讨论我们正在追求的 RDD 中的 lineage 信息的另一个好处，即便于在这些模型之间进行调试（§7.2）。

7.1 表达现有编程模型

RDD 可以高效*地*表达迄今为止独立提出的多种集群编程模型。所谓 "高效"，我们指的是 RDD 不仅可以用来产生与用这些模型编写的程序相同的输出，而且 RDD 还可以捕捉这些框架所执行的*优化，*例如将特定数据保留在内存中、对数据进行分区以尽量减少通信，以及高效地从故障中恢复。可使用 RDDs 表达的模型包括

MapReduce该模型可以使用 Spark 中的 *flatMap* 和 *groupByKey* 操作来表示，如果有组合器，也可以使用 *reduce- ByKey 来*表示。

DryadLINQ：与 MapReduce 相比，DryadLINQ 系统通过更通用的 Dryad 运行时提供了更广泛的操作符，但这些操作符都是与 Spark 中可用的 RDD 转换（*map*、*groupByKey*、*join* 等）直接对应的批量操作符。

SQL 查询与 DryadLINQ 表达式一样，SQL 查询也是对记录集进行数据并行操作。

Pregel：谷歌的 Pregel [22] 是一种专门用于迭代图应用程序的模型，乍看起来与其他系统中面向集合的编程模型大相径庭。在 Pregel 中，程序的运行是一系列协调的 "超步"。在每个超级步骤中，图中的每个顶点都会运行一个用户函数，该函数可以更新与顶点相关的状态、改变图的拓扑结构，并向其他顶点发送信息，供 *下一个*超级步骤使用。这个模型可以表达许多图算法，包括最短路径、双侧匹配和 PageRank。

让我们使用 RDDs 实现这一模型的关键是，Pregel 在每次迭代时都会对所有顶点应用*相同*的用户函数。因此，我们可以将 每次迭代的顶点状态存储在一个 RDD 中，然后执行批量转换（*flatMap*）来应用该函数并生成一个 RDD 消息。然后，我们可以将这个

RDD 与顶点状态一起执行信息交换。同样重要的是，RDD 允许我们像 Pregel 那样将顶点状态保存在内存中，通过控制顶点状态的分区将通信量降至最低，并支持故障时的部分恢复。我们在 Spark 上以 200 行库的形式实现了 Pregel，更多详情请读者参阅 [33]。

迭代式 MapReduce：最近提出的几种系统，包括 HaLoop [7] 和 Twister [11]，提供了一种迭代 MapReduce 模型，用户可向系统提供一系列 MapReduce 作业，让系统进行循环。这些系统在迭代过程中保持数据分区的一致性，Twister 还能将数据保存在内存中。这两种优化方式都可以用 RDDs 简单地表达，我们可以使用 Spark 将 HaLoop 作为一个 200 行的库来实现。

分批流处理：研究人员最近为定期用新数据更新结果的应用程序提出了几种增量处理系统[21, 15, 4]。例如，一个每 15 分钟更新一次广告点击统计数据的应用程序，应该能够将前 15 分钟窗口中的中间状态与来自新日志的数据结合起来。这些系统执行与 Dryad 类似的批量操作，但将应用状态存储在分布式文件系统中。将中间状态放在 RDD 中可以加快处理速度。

解释 RDD 的可表达性 为什么 RDD 能够表达这些不同的编程模型？原因在于对 RDD 的限制在许多并行应用中影响甚微。特别是，虽然 RDD 只能通过批量转换创建，但许多并行程序自然会*对许多记录应用相同的操作*，这使得 RDD 易于表达。同样，RDD 的不变性也不是障碍，因为我们可以创建多个 RDD 来表示同一数据集的不同版本。事实上，当今的许多 MapReduce 应用程序都在不允许更新文件的文件系统（如 HDFS）上运行。

最后一个问题是，为什么以前的框架没有提供相同程度的通用性。我们认为，这是因为这些系统探索了 MapReduce 和 Dryad 不能很好处理的特定问题，如迭代，却没有发现这些问题的*共同原因*是缺乏数据共享抽象。

7.2 利用 RDD 进行调试

虽然我们最初设计 RDD 是为了实现容错而确定性地重新计算，但这一特性也为调试提供了便利。特别是，通过记录作业期间创建的 RDD 的流程，我们可以 (1) 在以后重建这些 RDD，并让用户以交互方式查询它们，以及 (2) 通过记录作业期间创建的 RDD 的流程，让用户以交互方式查询它们。

(2) 通过重新计算它所依赖的 RDD 分区，在单进程除错器中重新运行作业中的任何任务。与传统的重放调试器不同，这种调试器适用于一般的故障排除。

这种方法几乎不增加记录开销，因为只需要记录 RDD 行序图。[9]我们目前正在基于这些想法开发一个 Spark 调试器[33]。

8 相关工作

集群编程模型：集群编程模型方面的相关工作分为几类。首先，数据流模型，如 MapReduce [10]、Dryad [19] 和 Ciel [23]，支持丰富的数据处理操作，但通过稳定的存储系统共享数据。RDD 是一种比稳定存储更*有效的*数据共享抽象，因为它们避免了数据复制、I/O 和序列化的成本。[10]

其次，一些数据流系统的高级编程接口，包括 DryadLINQ [31] 和 FlumeJava [8]，提供了语言集成的应用程序接口，用户可以通过 *map* 和 *join* 等操作符来操作 "并行集合"。不过，在这些系统中，并行集合代表磁盘上的文件或用于表达查询计划的短暂数据集。尽管这些系统会在同一查询中跨操作符（*例如，*一个*映射*后接另一个*映射*）进行数据管道化，但它们无法*跨*查询有效地共享数据。由于并行集合模型的便利性，我们将 Spark 的 API 基于并行集合模型，并不声称语言集成接口的新颖性，但通过提供 RDD 作为该接口背后的存储抽象，我们使其能够支持更广泛的应用类别。

第三类系统为需要数据共享的*特定*类别应用提供高级接口。例如，Pregel [22] 支持迭代图应用，而 Twister [11] 和 HaLoop [7] 则是迭代 MapReduce 运行时。不过，这些框架都是针对其支持的计算模式隐式地实现数据共享的，并没有提供用户可用于在自己选择的操作中共享自己选择的数据的通用方法。例如，用户无法使用 Pregel 或 Twister 将数据集加载到内存中，*然后*决定在上面运行什么查询。RDD 明确提供了 分布式存储抽象，因此可以支持这些专业系统无法捕捉的应用，如交互式数据挖掘。

最后，有些系统会公开共享的可变状态， 允许用户执行内存计算。例如，Piccolo[27] 允许用户运行并行函数，读取和更新分布式哈希表中的单元。分布式共享内存（DSM）系统[24]

[9] 与这些系统不同的是，基于 RDD 的调试器不会重放用户函数中的非确定性行为（*如*非确定性*映射*），但至少可以通过校验数据来报告这些行为。

[10] 请注意，在 RAMCloud [25] 等内存数据存储上运行 MapReduce/Dryad 仍需要数据复制和序列化，这对某些应用来说成本很高，如*第6.1 节*所示。

和 RAMCloud [25] 等键值存储系统提供了类似的模型。RDD 与这些系统有两点不同。首先，RDD 提供了基于*映射*、*排序*和*连接*等操作符的更高级编程接口，而 Piccolo 和 DSM 的接口只是对表单元的读取和更新。其次，Piccolo 和 DSM 系统通过检查点和回滚实现恢复，在许多应用中，这比 RDDs 基于行的策略更昂贵。最后，如第 2.3 节所述，与 DSM 相比，RDD 还具有其他优势，如减少散兵游勇。

缓存系统：Nectar[12]可以通过程序分析[16]识别共同的子表达式，从而在DryadLINQ作业中重复使用中间结果。将这种功能添加到基于 RDD 的系统中将非常有吸引力。不过，Nectar 并不提供内存缓存（它将数据放在分布式文件系统中），也不允许用户明确控制要持久化的数据集以及如何对数据集进行分区。Ciel [23] 和 FlumeJava [8] 同样可以缓存任务结果，但不提供内存缓存或对缓存数据的显式控制。

Ananthanarayanan 等人建议在分布式文件系统中添加内存缓存，以利用数据访问的时间和空间位置性[3]。虽然这种解决方案能更快地访问文件系统中的数据，但它并不像 RDD 那样能高效地共享应用程序中的*中间*结果，因为它仍然需要应用程序在各个阶段之间将这些结果写入文件系统。

脉络：捕捉数据的源流或出处信息一直是科学计算和数据库领域的研究课题，其应用领域包括解释结果、允许他人复制结果，以及在工作流中发现错误或数据集丢失时重新计算数据。读者可参阅 [5] 和 [9]，了解这方面的工作。RDDs 提供了一种并行编程模型，在这种模型中，捕捉细粒度的脉络并不昂贵，因此可用于故障恢复。

我们基于行系的恢复机制也类似于 MapReduce 和 Dryad 中*在*计算（作业）中使用的恢复机制，后者跟踪任务 DAG 之间的依赖关系。不过，在这些系统中，作业结束后，线序信息就会丢失，这就需要使用复制存储系统来*跨*计算共享数据。相比之下，RDD 应用线性关系，在计算过程中有效地保存内存中的数据，而不需要复制和磁盘 I/O 的成本。

关系数据库：RDD 在概念上类似于数据库中的视图，持久的 RDD 类似于物化视图 [28]。然而，与 DSM 系统一样，数据库通常允许对所有记录进行细粒度读写访问，需要记录操作和数据以实现容错，并需要额外的开销来维护

一致性。RDD 的粗粒度转换模型不需要这些开销。

9 结论

我们提出了弹性分布式数据集（RDDs），它是集群应用中共享数据的一种高效、通用、容错的表述方式。RDD 可以表达广泛的并行应用，包括为迭代计算提出的许多专门编程模型，以及这些模型无法捕捉的新应用。现有的集群存储抽象需要通过数据复制来实现容错，而 RDD 与之不同，它提供了一种基于粗粒度转换的应用程序接口（API），使其能够使用 lineage 高效地恢复数据。我们在一个名为 Spark 的系统中实现了 RDDs，该系统在迭代应用中的性能比 Hadoop 高出 20 倍，可用于交互查询数百 GB 的数据。

我们已在 spark-project.org 上开源了 Spark，作为可扩展数据分析和系统研究的工具。

致谢

我们感谢第一批 Spark 用户，包括蒂姆-亨特（Tim Hunter）、莱斯特-麦基（Lester Mackey）、迪利普-约瑟夫（Dilip Joseph）和詹继斌，他们在实际应用中试用了我们的系统，提出了许多好建议，并指出了一些研究难题。我们还要感谢我们的导师埃德-南丁格尔（Ed Nightingale）和审稿人的反馈意见。本研究部分得到了伯克利 AMP 实验室赞助商谷歌、SAP、亚马逊网络服务、云时代、华为、IBM、英特尔、微软、NEC、NetApp 和 VMWare、美国国防部高级研究计划局（合同编号：FA8650-11-C-7136）、谷歌博士奖学金以及加拿大自然科学与工程研究理事会的支持。

参考资料

[1] Apache Hive。http://hadoop.apache.org/hive。
[2] 斯卡拉。http://www.scala-lang.org。
[3] G. Ananthanarayanan、A. Ghodsi、S. Shenker 和 I. Stoica。数据中心计算中的磁盘位置无关紧要。*HotOS '11*, 2011.
[4] P.Bhatotia, A. Wieder, R. Rodrigues, U. A. Acar, and R.帕斯金Incoop：用于增量计算的 MapReduce。*ACM SOCC '11*, 2011.
[5] R.Bose and J. Frew.科学数据处理的脉络检索：调查。*ACM Computing Surveys*, 37:1-28, 2005.
[6] S.Brin and L. Page.大型超文本网络搜索引擎剖析。*WWW*, 1998.
[7] Y.Bu、B. Howe、M. Balazinska 和 M. D. Ernst。HaLoop：大型集群上的高效迭代数据处理。*Proc.*3:285-296, September 2010.
[8] C.Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry、R.Bradshaw 和 N. Weizenbaum.FlumeJava: 简单、高效的数据并行流水线。In *PLDI '10*.ACM，2010 年。
[9] J.Cheney, L. Chiticariu, and W.-C. Tan.Tan.数据库中的出处：为什么、如何、在哪里。*数据库的基础与趋势》*，1（4）：379-474，2009 年。
[10] J.Dean 和 S. Ghemawat.MapReduce：大型集群上的简化数据处理。*OSDI*, 2004.

[11] J.Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox.Bae, J. Qiu, and G. Fox.Twister: a runtime for iterative mapreduce.在 *HPDC '10 上*，2010 年。

[12] P.K. Gunda、L. Ravindranath、C. A. Thekkath、Y. Yu 和 L.Zhuang.花蜜：数据中心的数据和计算自动管理。2010 年 *OSDI '10*。

[13] Z.Guo、X. Wang、J. Tang、X. Liu、Z. Xu、M. Wu、M. F. Kaashoek 和 Z. Zhang.R2：用于记录和重放的应用级内核。OSDI'08, 2008.

[14] T.Hastie、R. Tibshirani 和 J. Friedman。*统计学习要素：数据挖掘、推理和预测》*。Springer Publishing Company, New York, NY, 2009.

[15] B.He, M. Yang, Z. Guo, R. Chen, B. Su, W. Lin, and L. Zhou.彗星：数据密集型分布式计算的批量流处理。在 *SoCC '10*.

[16] A.Heydon, R. Levin, and Y. Yu.使用精确依赖关系缓存函数调用。*ACM SIGPLAN 公告》*，第 311-320 页，2000 年。

[17] B.Hindman、A. Konwinski、M. Zaharia、A. Ghodsi、A. D. Joseph、R. H. Katz、S. Shenker 和 I. Stoica。Mesos：数据中心细粒度资源共享平台。In *NSDI '11*.

[18] T.Hunter, T. Moldovan, M. Zaharia, S. Merzgui, J. Ma, M. J. Franklin, P. Abbeel, and A. M. Bayen.在云中扩展移动千年系统。In *SOCC '11*, 2011.

[19] M.Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly.Dryad：来自顺序构件的分布式数据并行程序。*欧洲系统07*，2007 年。

[20] S.S. Y. Ko, I. Hoque, B. Cho, and I. Gupta.云计算中中间数据的可用性。In *HotOS '09*, 2009.

[21] D.Logothetis、C. Olston、B. Reed、K. C. Webb 和 K. Yocum。增量分析的有状态批量处理。SoCC '10.

[22] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn、N.Leiser, and G. Czajkowski.Pregel：大规模图处理系统。In *SIGMOD*, 2010.

[23] D.G. 默里、M. 施瓦茨科普夫、C. 斯莫顿、S. 史密斯、A.Madhavapeddy, and S. Hand.Ciel：分布式数据流计算的通用执行引擎。在 *NSDI*，2011 年。

[24] B.Nitzberg and V. Lo.分布式共享内存：问题与算法概览》。*Computer*, 24(8):52 -60, Aug 1991.

[25] J.Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis、J.Leverich, D. Mazie`res, S. Mitra, A. Narayanan, G. Parulkar、M.Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman.RAMClouds 的案例：完全在 DRAM 中的可扩展高性能存储。*SIGOPS Op.Rev.*, 43:92-105, Jan 2010.

[26] D.Peng 和 F. Dabek.使用分布式事务和通知的大规模增量处理。见 *OSDI 2010*。

[27] R.Power 和 J. Li.Piccolo：用分区表构建快速分布式程序。在 *Proc.奥斯迪 2010*，2010 年。

[28] R.Ramakrishnan 和 J. Gehrke.*数据库管理系统》*。麦格劳-希尔公司，2003 年 3 版。

[29] K.Thomas、C. Grier、J. Ma、V. Paxson 和 D. Song。实时 URL 垃圾邮件过滤服务的设计与评估。*IEEE 安全与隐私研讨会*，2011 年。

[30] J.W. Young.最佳检查点间隔的一阶近似值。*Commun.ACM*，17：530-531，1974 年 9 月。

[31] Y.Yu, M. Isard, D. Fetterly, M. Budiu, U´ .Erlingsson, P. K. Gunda, and J. Currey.DryadLINQ: A system for 使用高级语言进行通用分布式数据并行计算。2008 年 *OSDI '08*。

[32] M.M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy、S.申克和 I. 斯托伊卡。延迟调度：集群调度中实现局部性和公平性的简单技术。*欧洲系统10*，2010 年。

[33] M.M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma、M.McCauley、M. Franklin、S. Shenker 和 I. Stoica。弹性分布式数据集：内存集群计算的容错抽象。技术报告 UCB/EECS-2011-82，加州大学伯克利分校电子工程科学系，2011 年。