

Spark 理论基石 —— RDD

分布式系统，数据库，存储，摄影

概述

RDD，学名可伸缩的分布式数据集（Resilient Distributed Dataset）。是一种对数据集形态的抽象，基于此抽象，使用者可以在集群中执行一系列计算，而不用将中间结果落盘。而这正是之前 MR 抽象的一个重要痛点，每一个步骤都需要落盘，使得不必要的开销很高。

对于分布式系统，容错支持是必不可少的。为了支持容错，RDD 只支持粗粒度的变换。即，输入数据集是 immutable（或者说只读）的，每次运算会产生新的输出。不支持对一个数据集中细粒度的更新操作。这种约束，大大简化了容错支持，并且能满足很大一类的计算需求。

初次接触 RDD 的概念的时候，不大能够理解为什么要以数据集为中心做抽象。后来随着不断深入的了解，对数据集的一致性抽象正是计算流水线（pipeline）得以存在和优化的精髓所在。在定义了数据集的基本属性（不可变，分区，依赖关系，存放位置等）后，就可以在此基础上施加各种高阶算子，以构建 DAG 执行引擎，并做适当优化。从这个角度来说，RDD 实在是一种精妙设计。

例行总结一下 RDD 论文的主要设计点有：

1. **显式抽象**。将运算中的数据集进行显式抽象，定义了其接口和属性。由于数据集抽象的统一，从而可以将不同的计算过程组合起来进行统一的 DAG 调度。
2. **基于内存**。相较于 MapReduce 中间结果必须落盘，RDD 通过将结果保存在内存中，从而大大降低了单个算子计算延迟以及不同算子之间的加载延迟。
3. **宽窄依赖**。在进行 DAG 调度时，定义了宽窄依赖的概念，并以此进行阶段划分，优化调度计算。
4. **谱系容错**。主要依赖谱系图计算来进行错误恢复，而非进行冗余备份，因为内存实在是有限，只能以计算换存储了。
5. **交互查询**。修改了 Scala 的解释器，使得可以交互式的查询基于多机内存的大型数据集。进而支持类 SQL 等高阶查询语言。

作者：青藤木鸟 [Muniao's blog](#) 转载请注明出处

小引

Dryad 和 MapReduce 是业已流行的大数据分析工具。它们给用户提供了些高阶算子来使用，而不用去关心底层的分布式和容错细节。但它们都缺少对分布式内存的抽象，不同计算过程之间只能够通过外存来耦合：前驱任务将计算结果写到外存上去，后继任务再将其作为输入加载到内存，然后才能接着执行后继计算任务。这样的设计有两个很大的劣势：复用性差、延迟较高。这对于像 Page-Rand，K-Means，LR 等要求迭代式计算

的机器学习算法（需要数据复用）极其不友好；对于一些随机的交互式查询（要求延迟低）也是个灾难。因为他们将大部分的时间都耗费在数据备份、硬盘 IO 和数据序列化之上。

在 RDD 之前，为了解决数据复用的问题，业界已有诸多尝试。包括将中间结果放在内存中的迭代式图计算系统——Pregel，以及将多个 MR 串在一块，缓存循环不变量的 HaLoop。但这些系统只支持受限的计算模型（比如MR），而且只进行隐式[1]的数据复用。如何进行更通用的数据复用，以支持更复杂的查询计算，仍是一个难题。

RDD 正是为解决问题而设计，高效地复用数据的一个**数据结构抽象**。RDD 支持数据容错、数据并行；在此之上，能够让用户利用多机内存、控制数据分区、构建一系列运算过程。从而解决很多应用中连续计算过程对于数据复用的需求。

其中比较难的一个设计是如何针对内存数据进行**高效的容错**。现有的一些基于集群内存的系统，比如分布式 KV、共享内存、Piccolo 都提供一种可以细粒度的修改的可变数据集抽象。为了支持这种抽象之上的容错，就需要进行数据多机冗余或者操作日志备份。这些操作都会导致多机间大量的数据传输，由于网络带宽远慢于 RAM，使得分布式利用内存这件事失去其优势。

与之相对，RDD 只提供粗粒度的、基于整个数据集的计算接口，即数据集中的所有条目都施加同一种操作。这样一来，为了容错，我们只需要**备份每个操作而非数据本身**（因为是整体更新的）；在某个分区数据出现问题进行错误恢复时，只需要从原始数据集出发，按顺序再算一遍即可。

初看起来，这种计算抽象很受限，但它其实能满足现有的一大类的集群计算需求，包括 MR、DryadLINQ、SQL、Pregel 和 HaLoop。并且能满足一些其他计算需求，比如说交互式计算。RDD 的实现系统 Spark，提供类似 DryadLINQ 的高阶算子，应该是第一个提供交互式的集群运算接口。

RDD

本节首先给出 RDD 的详细定义，然后介绍下 Spark 的中针对 RDD 的操作接口，继而对比了 RDD 与提供细粒度更新接口的共享内存抽象优劣。最后就 RDD 的局限性讨论一下。

RDD 抽象

RDD 是一个基于分区的、只读的数据记录集抽象。RDD 只可以通过对持久存储或其他 RDD 进行确定性运算得来，这种运算被称为**变换**。常用的**变换**算子包括：map，filter 和 join。

RDD 没有选择不断的做检查点以进行容错，而是会记下 RDD 从最初的外存的数据集变化而来的变化路径，也就是其**谱系**（lineage）。理论上所有的 RDD 都可以在出错后从外存中依据谱系图进行重建。一般来说，重建的粒度是分区（Partition）而非整个数据集，一来代价更小，二来不同分区可能在不同机器上。

用户可以对 RDD 的两个方面进行控制：持久化和分区控制。对于前者，如果某些 RDD 需要复用，那么用户可以指示系统按照某种策略将其进行持久化。后者来说，用户可以定制分区路由函数，将数据集中的记录按照某个键值路由到不同分区。比如进行 Join 操作的时候，可以讲待 Join 数据集按照相同的策略进行分区，以并行 Join。

Spark 编程接口

Spark 通过暴露与编程语言集成的算子来提供操作 RDD 的接口。其中 RDD 表现为编程语言中的类，而 RDD 的算子为作用于这些类上的函数。之前的系统如 DryadLINQ 和 FlumeJava 也使用了类似的形式。

用户使用 RDD 时，首先将数据从持久化存储中通过变换 (**Transformations**，如 *map* 或者 *filter*) 将其载入内存，然后可以对 RDD 施加任何系统支持的一系列变换，最后利用动作 (**Action**) 算子，将 RDD 重新持久化到外存中或者将控制权交还用户。和 DryadLINQ 一样，这个加载-变换-落盘的过程是**声明式** (Declarative，或者说是惰式[2]) 的，Spark 在拿到整个拓扑后会利用执行引擎进行执行优化 (比如将并行化、流水线化，之后会进一步讨论)。

此外很重要的一个接口是 *persist*，可以由用户来告诉系统**哪些** RDD 需要持久化，**如何**持久化 (本机硬盘还是跨机器存储)，如果有多个 RDD 需要持久化，那么**优先级**如何确定。Spark 默认将 RDD 保存在内存中，如果内存不够用了会根据用户配置将数据溢出 (spill) 到硬盘上。

举个栗子

假设我们相对存在于 HDFS 上的日志文件，找出错误条目，针对出现 `hdfs` 关键字的具体条目进行分析。利用 Spark 接口，使用 Scala 语言实现，代码如下：

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

// 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()
```

第一行基于某个 `hdfs` 上的文件定义一个 rdd (每一行作为集合中的一个条目)。第二行通过 `filter` 变换生成新的 rdd，第三行请求 spark 将其结果进行暂存。最后一行是链式操作，以一个 `collect` 的动作结尾，求出包含 HDFS 关键字的所有行数的各个字段。

其计算的谱系图 (lineage) 如下：

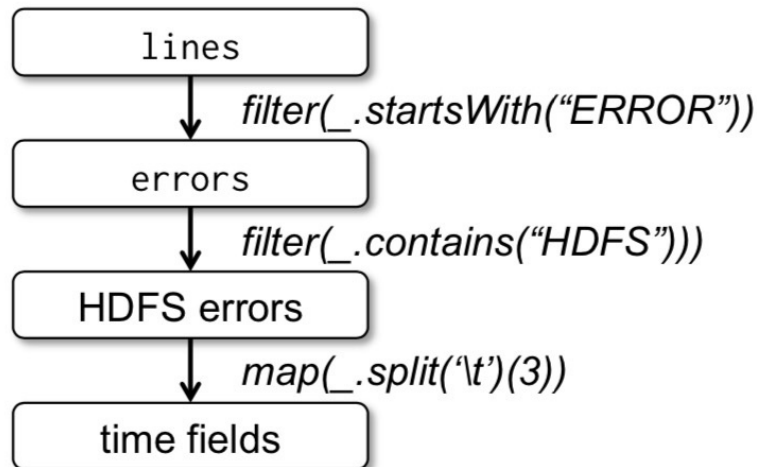


Figure 1: Lineage graph for the third query in our example. Boxes represent RDDs and arrows represent transformations.

知乎 @穆尼奥

有两点需要注意：

1. 直到遇到 `collect` 这个**动作** (Action) 之前都没有发生实际的运算。
2. 链式操作时不保存中间结果；

由于第三行将结果在内存中进行了缓存，因此还可以基于此做其他动作。比如，计算包含 'MySQL' 关键字的错误条数：

```
// Count errors mentioning MySQL:
errors.filter(_.contains("MySQL")).count()
```

RDD 模型的优点

为了理解 RDD 带来的好处，可以看下面一个表，将 RDD 与 DSM (Distributed Shared Memory) 做了详细对比。DSM 在这里是一个很宽泛的抽象，不仅包括一般的内存共享系统，还包括其他支持细粒度的状态更新的框架，比如说 Piccolo、分布式数据库等。

Aspect	RDDs	Distr. Shared Mem.
Reads	Coarse- or fine-grained	Fine-grained
Writes	Coarse-grained	Fine-grained
Consistency	Trivial (immutable)	Up to app / runtime
Fault recovery	Fine-grained and low-overhead using lineage	Requires checkpoints and program rollback
Straggler mitigation	Possible using backup tasks	Difficult
Work placement	Automatic based on data locality	Up to app (runtimes aim for transparency)
Behavior if not enough RAM	Similar to existing data flow systems	Poor performance (swapping?)

Table 1: Comparison of RDDs with distributed shared memory

首先，DSM 和 RDD 最主要的区别在于，DSM 支持对数据集细粒度的更新。即，可以对任意内存位置进行更新。而 RDD 舍弃了这一点，只允许批量的写入数据，从而提高了容错效率：

1. 使用 lineage 来按需恢复数据，而不用定期 snapshot，减小了不必要开销。
2. 每个 Partition 出错后可以单独进行恢复，而不用进行全数据集的重建。

其次，RDD 的不可变的特点允许系统叫较容易的对某些计算进行迁移。比如说 MR 中的某些 Stragger 任务就可以很方便的迁移到其他计算节点上去，因为其输入数据一定不会被改变，因此不用考虑一致性的问题。

最后还有两个好处值得一提：

1. 由于只支持批量计算，因此调度系统可以比较好的利用数据局部性的特点加快运算速度。
2. 如果集群内存不够的话，只要数据支持迭代，就可以分批加载到内存进行运算，或者分批将结果 spill 到外存。如此一来，在内存不够时能提供很优雅的退化操作，并不太损失性能。

RDD 不适用的场景

如前所述，RDD 适用于针对全数据集统一处理的粗粒度变换的抽象。相对的，就不适用于要求对数据进行细粒度的、异步更新的数据集。比如说 web 应用，再比如说爬虫等等。对于这些引用类型，传统的快照+操作日

志的容错方式可能更适合一些。如数据库 RAMCloud , Percolator 和 Piccolo。 RDD 的目标在于批量分析型应用，而将这些异步应用的需求留给那些专有系统。

Spark 编程接口

Spark 利用 Scala 语言作为 RDD 抽象的接口，因为 Scala 兼顾了精确（其函数式语义适合交互式场景）与高效（使用静态类型）。当然，对于 RDD 本身来说，不限于于任何特定的语言表达。下面从**执行流程**与**代码分发**两个方面来详细说明下 Spark 是如何执行用户代码的。

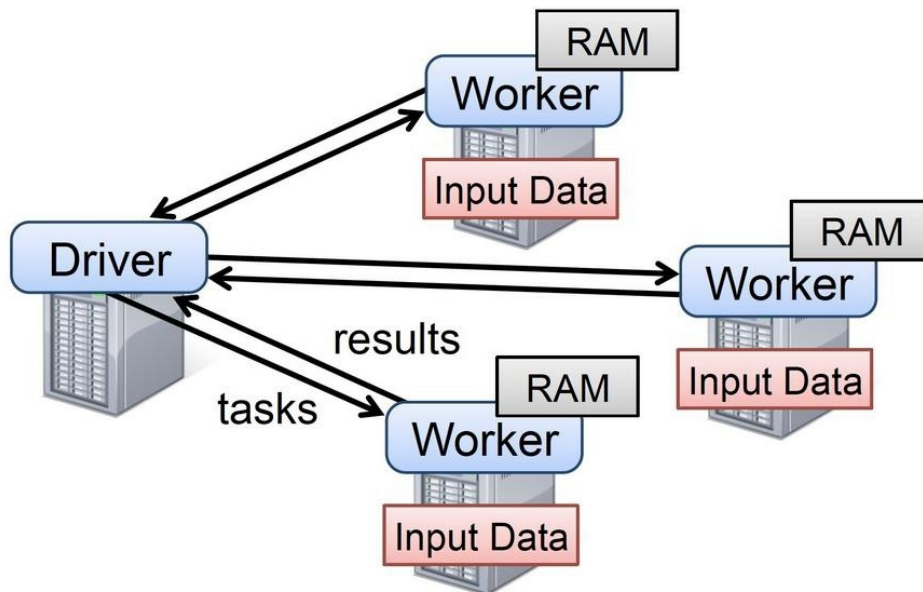


Figure 2: Spark runtime. The user's driver program launches multiple workers, which read data blocks from a distributed file system and can persist computed RDD partitions in memory.

开发者利用 Spark 提供的库编写**驱动程序**（driver programme）以使用 Spark。驱动程序会定义一到多个 RDD，并对其进行各种变换。Spark 提供的库会连接 Spark 集群，生成计算拓扑，并将拓扑分散到多个 workers 上去进行执行，同时记下变换的谱系（lineage）。这些 workers 是分散在 Spark 集群内各个机器上的常驻进程，它们在内存里保存计算过程中生成的 RDD 的各个分区。

像前面举的例子一样，开发者需要将函数作为参数传给 map 等 Spark 算子。Spark 会将这些函数（或者说**闭包**）序列化为 Java 对象，然后分发给执行节点进行加载。闭包所涉及的变量会被当做上述生成对象的字段值。RDD 本身会被包装成静态类型的参数进行传递。由于 Scala 支持类型推断，大部分例子都省掉了 RDD 数据类型。

尽管 Spark 暴露的 Scala 的 RDD 接口在概念上看起来很简单，但实在实现上有一些很脏的角落，比如说 Scala 的闭包得使用反射，比如说尽量避免修改 Scala 的解释器。

Spark 中的 RDD 操作

下表列出了 Spark 中支持的 RDD 操作。如前面所说，**变换** (transformations) 是生成新 RDD 的惰性算子，而**动作** (actions) 是触发调度的算子，它会返回一个结果或者将数据写到外存中。

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

Table 2: Transformations and actions available on RDDs in Spark. Seq[T] denotes a sequence of elements of type T.

需要注意的是：

1. 有些操作如 `join`，要求操作数 RDD 必须是键值对 (key value pairs)。
2. `map` 是一一对一的映射，而 `flatMap` 是类似于 MapReduce 中一对一或者一对多的映射。
3. `save` 会将 RDD 进行持久化。
4. `groupByKey`，`reduceByKey` 和 `sort` 都会导致 RDD 中不同分区进行再哈希或者重排。

RDD 的表示

提供 RDD 抽象的一个难点在于，如何高效的**跟踪谱系**并能提供丰富的**变换支持**。最后我们选用了基于图的调度模型，将调度和算子进行了解耦。从而能够在不改变调度模块逻辑的前提下，很方便的增加算子支持。具体来说，RDD 抽象的核心组成主要有以下五个部分：

1. **分区集** (partition set)。分区是每个 RDD 的最小构成单元。
2. **依赖集** (dependencies set)。主要是 RDD 间的父子依赖关系。
3. **变换函数** (compute function)。作用于分区上的变换函数，可以由几个父分区计算得到一个子分区。
4. **分区模式** (partition scheme)。该 RDD 的分区是基于**哈希分片**的还是**直接切分**的。
5. **数据放置** (data placement)。知道分区的存放位置可以进行计算优化。

Operation	Meaning
<code>partitions()</code>	Return a list of Partition objects
<code>preferredLocations(<i>p</i>)</code>	List nodes where partition <i>p</i> can be accessed faster due to data locality
<code>dependencies()</code>	Return a list of dependencies
<code>iterator(<i>p</i>, <i>parentIters</i>)</code>	Compute the elements of partition <i>p</i> given iterators for its parent partitions
<code>partitioner()</code>	Return metadata specifying whether the RDD is hash/range partitioned

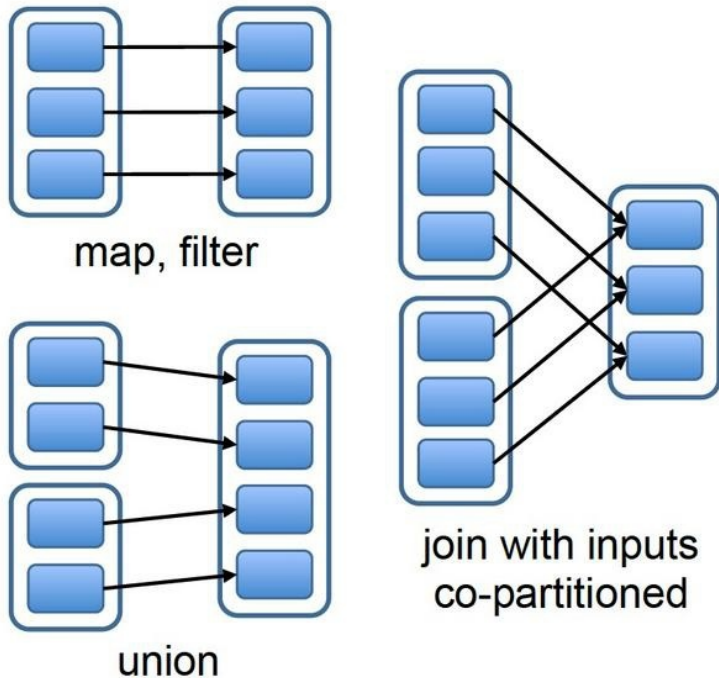
Table 3: Interface used to represent RDDs in Spark.

知乎 @穆尼奥

在 RDD 的接口设计中最有趣的一个点是如何对 RDD 间的**依赖关系**进行规约。最后发现可以将所有依赖归纳为两种类型：

1. **窄依赖** (narrow dependencies)：父 RDD 的分区最多被一个子 RDD 的分区所依赖，比如 *map*。
2. **宽依赖** (wide dependencies)：父 RDD 的分区可能被多个子 RDD 的分区所依赖，比如 *join*。

Narrow Dependencies:



Wide Dependencies:

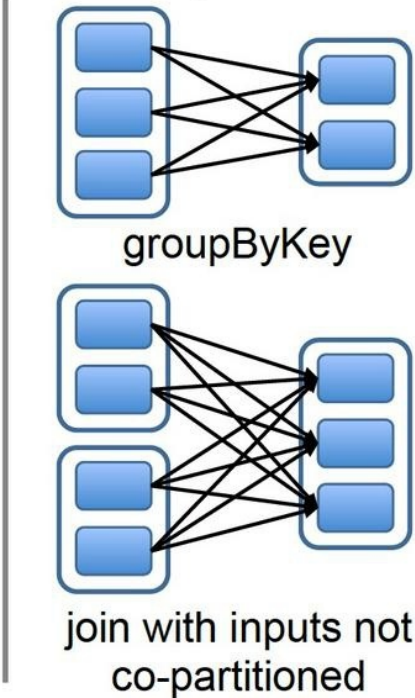


Figure 4: Examples of narrow and wide dependencies. Each box is an RDD, with partitions shown as shaded rectangles.

如此归纳的原因主要有两点。

调度优化。对于窄依赖，可以对分区间进行并行**流水化**调度，先计完成某个窄依赖算子（比如说 map）的分区不用等待其他分区而直接进行下一个窄依赖算子（比如 filter）的运算。与之相对，宽依赖的要求父 RDD 的所有分区就绪，并进行跨节点的传送后，才能进行计算。类似于 MapReduce 中的 shuffle。

数据恢复。在某个分区出现错误或者丢失时，窄依赖的恢复更为高效。因为涉及到的父分区相对较少，并且可以并行恢复。而对于宽依赖，由于依赖复杂（如上图，子 RDD 的每个分区都会依赖父 RDD 的所有分区），一个分区的丢失可能就会引起全盘的重新计算。

这样将调度和算子解耦的设计大大简化了变换的实现，大部分变换都可以用20余行代码来实现。由于不需要了解调度细节，任何人都可以很快的上手实现一个新的变换。试举几例：

HDFS 文件：*partitions* 函数返回 HDFS 文件的所有 block，每个 block 被当做一个 partition。*preferredLocations* 返回每个 block 所在的位置，*iterator* 会对每个 block 进行读取。

map：在任意 RDD 上调用 map 会返回一个 MappedRDD 对象，该对象的 *partitions* 函数和 *preferredLocations* 与父 RDD 保持一致。对于 *iterator*，只需要将传给 map 算子的函数以此作用到其父 RDD 的各个分区即可。

union: 在两个 RDD 上调用 union 会返回一个新的 RDD，该 RDD 的每个分区由对应的两个父 RDD 通过窄依赖计算而来。

sample：抽样函数和 map 大体一致。但该函数会给每个分区保存一个随机数种子来决定父 RDD 的每个记录是否保留。

join：在两个 RDD 上调用 join 操作可能会导致两个窄依赖（比如其分区都是按待 join 的key 哈希的），两个宽依赖，或者混合依赖。每种情况下，子 RDD 都会有一个 *partitioner* 函数，或继承自父分区，或是默认的 hash 分区函数。

实现

Spark 最初版本（论文里提到的），只有 1.4w 行 Scala 代码，由 mesos 管理资源分配，可以和 Hadoop 生态共用资源，并从 Hadoop/Hbase 中加载数据。对于 Spark 的实现，有几个值得一说的点：Job 调度，交互式解释器，内存管理和检查点机制（checkpointing）。

Job 调度

Spark 调度设计依赖于上一节提到的 RDD 的抽象。它的调度策略和 Dryad 有点像，但又不尽相同。在用户在某个 RDD 上调用 Action 类型（count，save 等等）的算子时，调度器就会根据用户代码中调用算子的顺序生成计算拓扑。我们把每一个变换前后的 RDD 当做点，算子产生的 RDD 间的依赖/父子关系当做边，如此构成一个有向无环图（DAG）。为了减小传输，调度器会将几个连续的计算进行归并，称为阶段（Stage），进行阶段归并的依据为是否需要 shuffle，也即是否为宽依赖。这样，会形成一个新的由阶段组成的更精简的 DAG。

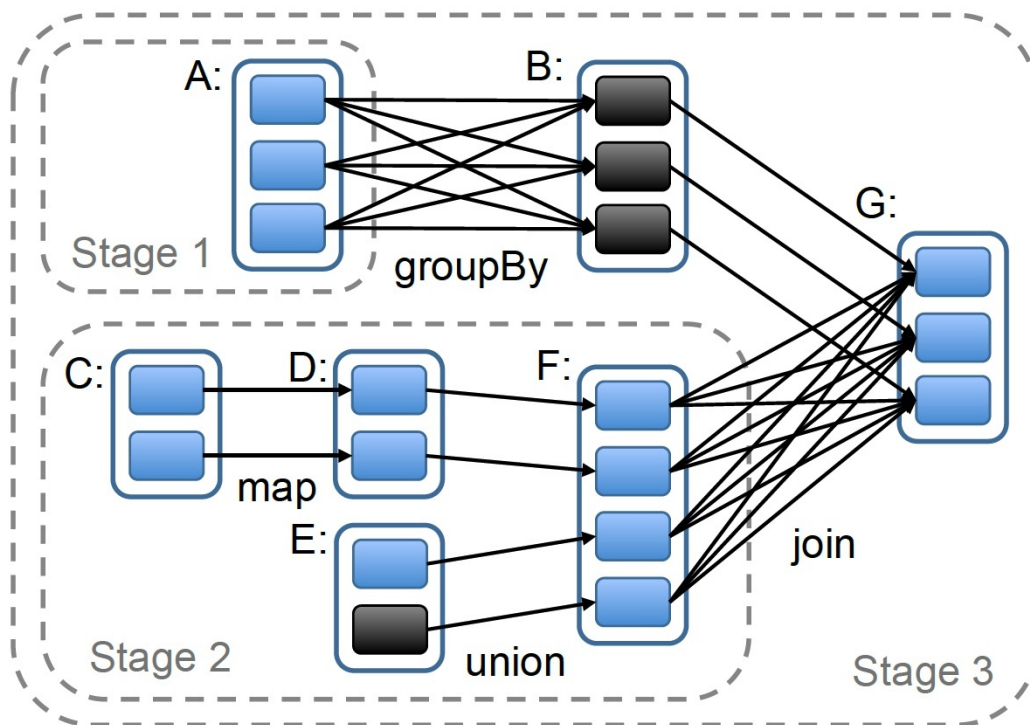


Figure 5: Example of how Spark computes job stages. Boxes with solid outlines are RDDs. Partitions are shaded rectangles, in black if they are already in memory. To run an action on RDD G, we build stages at wide dependencies and pipeline narrow transformations inside each stage. In this case, stage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

之后，调度器会从目标 RDD 出发，沿着 DAG 图中的边往前遍历，对每个不在内存中的分区进行计算。如果需要计算的分区已经在内存中了，则直接利用结果即可，如上图所示。

然后，调度器会将任务调度到离其依赖 RDD 的 Partition 近的地方去：

1. 如果 Partition 在某节点的内存中，则将任务的调度到该节点上。
2. 如果 Partition 还在硬盘上，则将任务调度到 preferredLocations 函数返回的地方去（如 HDFS 文件）。

对于宽依赖，Spark 和 MR 一样，会将其中间结果输出持久化起来，以简化容错。如果某个 Stage 的父 RDD 不可用，调度器就会新提交一些并行运行的任务，来生成这些缺失的分区。不过现在 Spark 还不能对调度器本身故障进行恢复，虽然看起来对 RDD 的谱系图进行冗余备份或许是一个简单可行的方案。

最后，现在仍是由用户 Driver 程序调用 Action 算子来触发调度任务。但我们正在探索维持一些周期性的检查性任务，对 RDD 中某些缺失的分区进行补足。

解释器集成

像 Python 和 Ruby 一样，Scala 提供交互式的 shell 环境。由于 Spark 将数据保存在内存中，我们希望可以借助 Scala 的这个交互式环境让用户对大数据集进行交互式实时的查询。

Scala 的解释器对用户代码进行解释执行的通常做法是，将用户键入的每一行 Scala 命令编译成一个 Java Class 字节码，然后将其加载到 JVM 中。该类包含一个初始化过的单例实例，实例中包含用户定义的变量和函数。比如，用户输入：

```
var x = 5
println(x)
```

Scala 解释器会针对第一行生成一个叫做 `Line1` 的类，其中有一个 `x` 的字段，并且将第二行编译为：
`println(Line1.getInstance().x)`

为了让 Scala 解释器能在分布式环境运行，我们在 Spark 中对其进行了以下修改：

1. **类代码传输 (Class shipping)**：为了让工作节点 (Worker Nodes) 可以拉取驱动节点 (Driver Node) 上解释器用户输入编译成的字节码，我们让解释器可以通过 HTTP 将每个类的访问开放出来。
2. **代码生成修改 (Modified code generation)**：Scala 解释器在处理不同行的访问时，会通过一个静态方法来获取其初始化后单例，进而访问上一行的变量 `Line.x`。但是我们只能通过 HTTP 传字节码而没有将初始化后实例（即 `x` 已经被赋值）传过来，因此工作节点不能访问到 `x`。因此我们改变了代码生成逻辑，使得不同行之间能够直接引用实例。

下图反映了我们修改后的 Scala 解释器生成 Java 对象的过程：

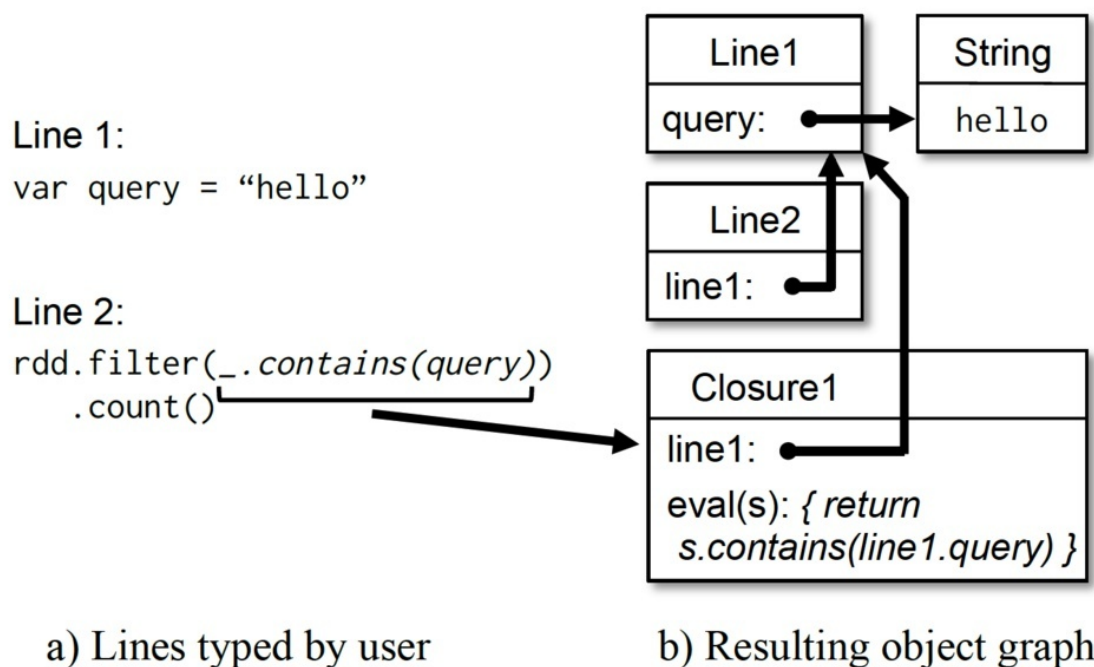


Figure 6: Example showing how the Spark interpreter translates two lines entered by the user into Java objects. 知乎 @穆尼奧

我们发现解释器在对大型数据集进行交互式查询时很有帮助，我们计划对更高级的查询语言进行支持，如 SQL。

内存管理

Spark 提供了三种存储 RDD 的方式：

1. 内存中没有序列化过的 Java 对象
2. 内存中序列化过的数据
3. 磁盘

由于 Spark 跑在 JVM 上，因此第一种存储方式访问最快，第二种允许用户牺牲一点性能以换取更高效的内存利用。当数据尺度太大以至于内存装不下的时候，第三种方式很有用。

为了有效的利用有限的内存，我们在 RDD 分区级别上进行 LRU 式的驱逐策略。即，当我们新计算出一个 RDD 的分区时，如果发现内存不够用，就会从内存中驱逐出去一个最久没有使用过的 RDD 的分区。但是，如果这个最久没有使用过的分区和新计算出的分区属于同一个 RDD，我们会接着寻找，直到找到一个和当前分区不属于一个 RDD 并且最久没用过的分区。因为 Spark 的大部分计算会施加于整个 RDD 上，这样做可以防止这些分区被反复的计算-驱逐。这个策略在论文成文时用的很好，不过，我们仍然提供给了用户进行深度控制的接口——指定存储优先级。

现在每个 Spark 实例拥有自己的分立的内存空间，我们计划将来提供跨 Spark 实例的统一的内存管理。

检查点机制

尽管所有失败的 RDD 都可以通过谱系 (lineage) 来重新计算得出，但是对于某些谱系特别长的 RDD 来说，这将是一个很耗时间的操作，因此提供 RDD 级别的外存检查点 (checkpointing) 可能会很有用。

对于具有很长谱系图，并且谱系图中存在很多宽依赖的 RDD，在外存上做检查点会很有帮助，因为某一两个的分区可能会引起全盘的重算，对于这种又臭又长的计算拓扑来说，依据谱系图重算无疑非常浪费时间。而对于只有窄依赖的、不那么长谱系图来说，在外存做检查点可能有些得不偿失，因为他们可以很简单的并行计算出来。

Spark 现阶段提供检查点的 API (给 persist 函数传 REPLICATE 标志)，然后由用户来决定是否对其持久化。但我们在思考，是否可以进行一些自动的检查点计算。由于调度器知道每个数据集的内存占用以及计算使用时间，我们或许可以选择性的对某些关键 RDD 进行持久化以最小化宕机恢复时间。

最后，由于 RDD 的只读特性，我们在做检查点时不用像通用共享内存模型那样过分考虑一致性的问题，因此可以用后台线程默默地干这些事情而不用影响主要工作流，也不用使用复杂的分布式的快照算法来解决一致性问题。

注解

[1] **隐式与显示显式**：在这里可以理解为，显式是把数据集这个概念完整的构造出来，定义他的内涵和边界，并基于其上做一些外延拓展。而隐式只是事实上复用了数据，但并没有定义被复用的数据格式。

[2] **声明式 (Declarative) 语言与命令式 (Imperative) 语言**：前者例子有 SQL，HTML；后者例子最常见的有 Shell，其他的常见编程语言 C，Java，Python 也属于此列。前者的好处在于将"干什么"和"怎么干"这两件事解耦，这样一来就可以开发不同的执行引擎，针对不同场景来优化"怎么干"这件事。而后者会告诉机器以特定的顺序执行特定的操作，与直觉一致，是一般编程语言的路子。
