RDD：基于内存的集群计算容错抽象 · 读书笔记

1.引言

+补充知识点

数据重用的方法

1.Map Reduce

> 写本地磁盘或写HDFS文中系施

开销:序列化，disk IO，数据复制。
在作业运行的总时间里，这个开销今占很大的比例。

2.

Progel 交互式图计算+HaLoop迭代MapReduce接口----------将中间数据继在在内在里以便重用。
但，中间数据的缓存是系统自动做的，用户没办法指定哪竺
数据应该浮右,应该怎掉经在。

3.

RDD为物据重用建立了一个通用抽象。允许用户将好几个数据集一同装进内存，按零查海其中的数据(不必全表扫描)

1️⃣RDD是容错、并行的数据结构，
2️⃣用户可以显式地将中间结果持久化在内存中，以便重用。控制划分第略、优化数据布局。用丰富的算子操纵数据。

> 怎样容错后文有详细说明

RDD的表达能力

RDD基于粗粒度转换的transform能够有效的表示的集群编程模型包括
Mapreduce\DryadLINQ\SQL\Pregel\HaLooP\interactive data mining

这些模型会对许多元素执行相同的操作

集群，内存数据挖掘方面，spark是第一个可以满足交互式查询速度需求的系统

RDD的性能

1️⃣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。

2️⃣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。

3️⃣Spark能够在5-7s延时范围内，交互式地查询1TB大小的数据集。

4️⃣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。

5️⃣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），说明早分布式数据分析领域，通用RDD编程模型的表达能力足够。以库的形式实现（分别使用了100和200行Scala代码）编程不复杂。

> 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。

2.弹性分布式数据集（RDD）

RDD抽象

RDD是只读的、分区记录的集合

RDD只能基于1️⃣在稳定物理存储中的数据集**和**2️⃣其他已有的RDD上执行确定性操作来创建。

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

🌟部分任务失败导致数据丢失，可以作业可以重建RDD丢失的数据

用户可以对RDD实施控制

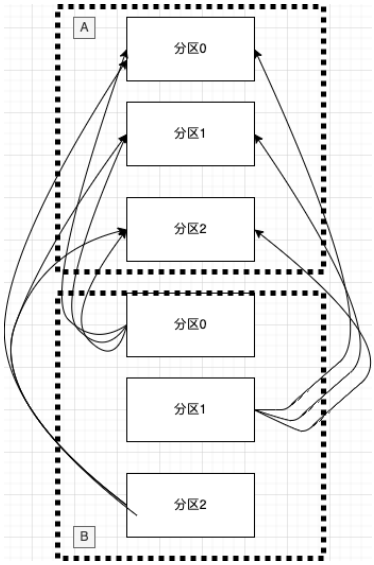
> 可以从两个方面控制RDD，即缓存和分区

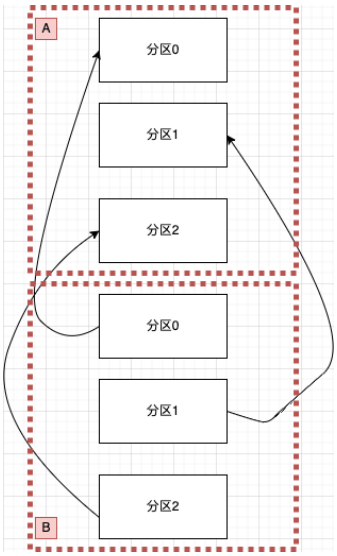
- **用户可以请求将RDD缓存**。这样运行时将已经计算好的RDD分区存储起来，以加速后期的重用。缓存的RDD一般存储在内存中，但如果内存不够，可以写到磁盘上。
- **RDD还允许用户根据关键字（key）指定分区顺序**

目前支持hash分区和range分区。用户可以对记录中的key进行hash，将记录存放在key的hash值对应的分区，这样，当我们需要对两个数据集进行join操作时，属于同一个key的记录，不可能出现两个不同的分区。这种叫做hash-partitioned RDD,join的时候可以节省很多计算量和数据传输量，这是我们在placement optimization方面做的工作。

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。

例：集合AB的join操作





- 左图表示没有hash partition的集合
  - join操作，3个字任务运行在A的3个分区所在的服务口?
  - 每个子任务需要收集B所有记录
  - 3*3次分区数据传输
- 右图表示含有hash partition的集合
  - AB存在相同的key，都是hash partition集合
  - AB分区0 key集合相同&key不会出现在其他分区
  - join操作：B分区0 -> A分区0
  - 三次分区数据传输就可以，计算量也对应小很多
- 半hash partition的集合 （A有B无
  - 数据传输图和左图相同，但是数据传输量是左图的1/3
  - 我们过滤RDD B中的记录，属于RDD分区的key，记录只传给RDDA分区0，不必送RDDA分区1和2

RDD与分布式共享内存

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[28]）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的优点

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？）

不适合使用RDD的应用

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。

3. Spark编程接口

RDD的实现

RDD是object。transformation 是object里面的方法

action的定义

action是程序，也可以说是代码或者operation，调用RDD中的method，运算结束之后给application返回一个值，或者把数据传入storage system

RDD的惰性计算

1个rdd ,一个action 引用他的时候系统计算其中的数据，所以多个transformation可以串成管道。

RDD的持久化

程序员调用persist()或者cache()的方法告诉系统这个RDD会重用。

然后spark就会把这个RDD的数据保存在内存里面，如果内存不够，就spill到磁盘上

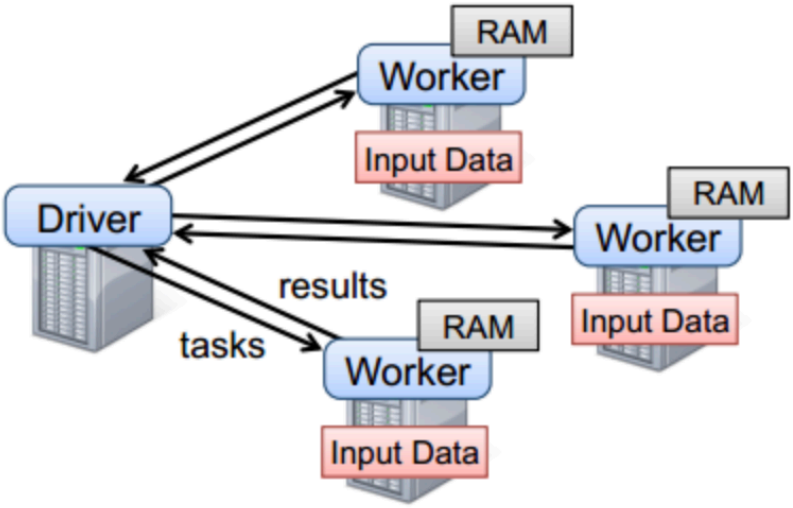
最后用户为每个RDD设置persistance priority，从而告诉系统内存紧张的时候，哪块数据应该先写盘/

控制台日志挖掘(例子)

Web service 出问题，我们要搜索hdfs文件系统中好几个T的 日志找原因。

使用spark，操作员可以只把error消息读入内存，然后对他们进行交互式查询。

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。



Spark的运行时。用户的driver程序启动多个worker

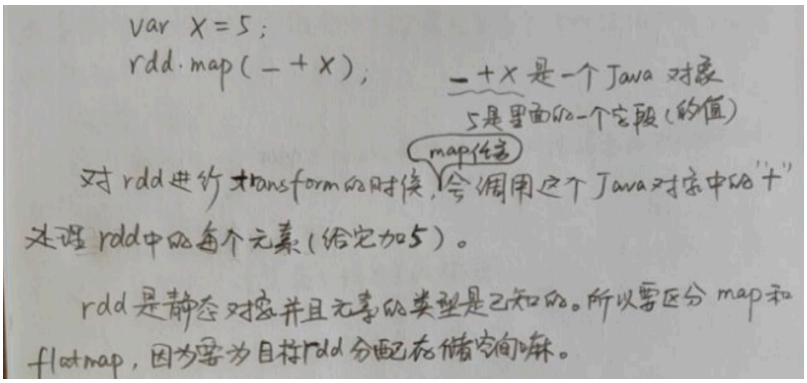
worker从分布式文件系统中读取数据块（对RDD上的数据执行action操作）

并将计算后的RDD分区缓存在内存中。

> driver上的spark代码还会跟踪rdd的lineage

> worker是一个一直运行的进程，担任RDD分区保存在RAM里

控制台日志挖掘(例子)中，用户给rdd提供的参数是clause scale，每个closure是一个java对象。可以序列化，装在其他node上运行。closure内部的任何变量是java对象的字段(field创建的时候赋值)



Spark中的RDD操作

表3 Spark中支持的RDD转换和动作

转换	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

> - 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。
> - 方括号表示类型参数

1️⃣有些操作只对键值对可用，比如join。

> key-value对形式的RDD可以用join操作

2️⃣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）。

3️⃣transformation定义一个新的RDD是惰性操作

4️⃣persist(cache)持久化数据

5️⃣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（get RDD partition order）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，然后看看如何使用RDD描述几种已有的集群编程模型。

4.1逻辑回归

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[15]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。

> map:计算每个样本点的梯度
>
> reduce：对梯度进行累加
>
> 用梯度修正模型参数w...

在内存中缓存points，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。

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()//如果任务包含combiner，则相应的代码为  .reduceByKey(myCombiner)
    .map((k, vs) => myReduce(k, vs))
```

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）。

```
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)
```

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。

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。

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。

表3 Spark中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

> Preferred location (data placement)
>
> 比如如果一个RDD表示一个HDFS文件，他的每个partition就是文件中的一块数据，placement就是存放数据的机器。
>
> 如果一个RDD是某个map操作的结果，一个map任务的结果存在同一个partition，是map任务对父RDD每个element的处理结果。

如何表示RDD之间的依赖

设计接口的一个关键问题就是，如何表示RDD之间的依赖。

RDD之间的依赖关系可以分为两类，即：

- 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）；
- 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）。

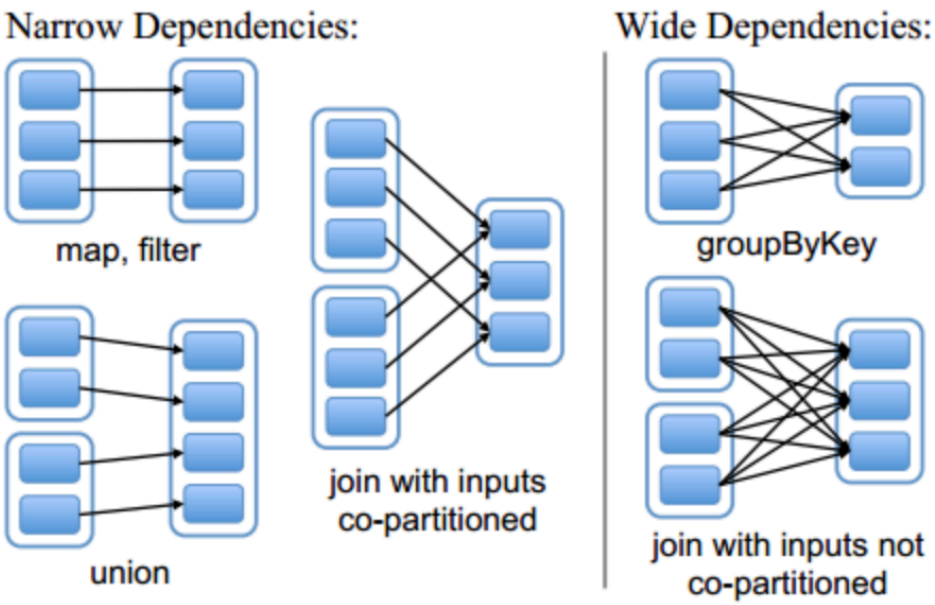


图5 窄依赖和宽依赖的例子。（方框表示RDD，实心矩形表示分区）

区分这两种依赖的意义：

- 第一，runtime对窄宽依赖处理不同。
  - 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：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，失效节点的恢复。
  - 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。

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）。

RDD实现举例

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，下面是 RDD实现实例

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。

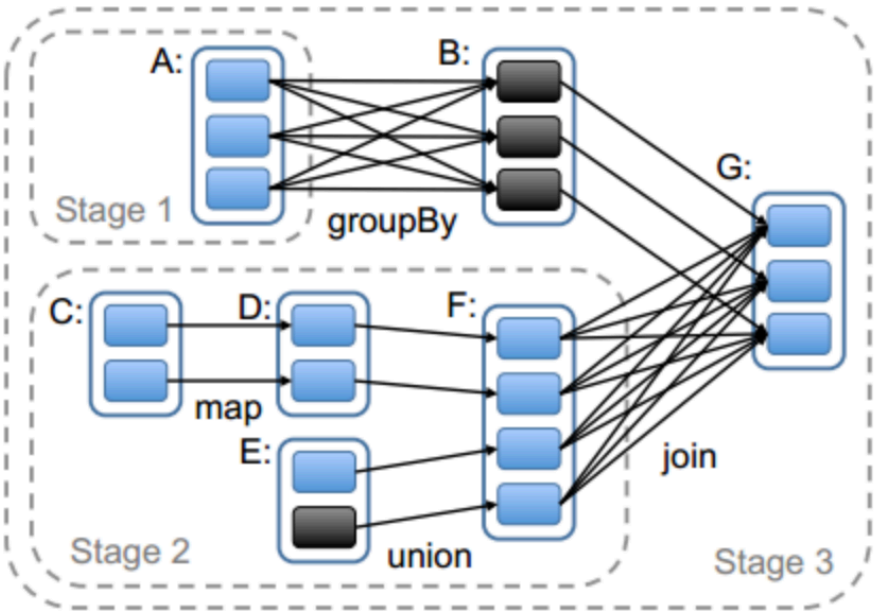
join：对两个RDD执行join操作可能产生窄依赖（如果这两个RDD拥有相同的哈希分区或范围分区），可能是宽依赖，也可能两种依赖都有（比如一个父RDD有分区，而另一父RDD没有）。

Spark任务调度器

怎样使用RDD接口进行调度：

调度器跟Dryad类似，但还考虑了哪些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。

1 调度器根据数据存放的位置分配任务（图片 TOP 一行解释），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。

2 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。

3 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区。

> 调度器容错，可能可以复制RDD的lineage图

4️⃣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。

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。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。对比来看，对稳定存储中数据的具有窄依赖的RDD，例如，在我们逻辑回归的例子中的点集合，以及在Pregel优化变体中不可变的顶点状态，可能永远都不需要执行检查点操作。

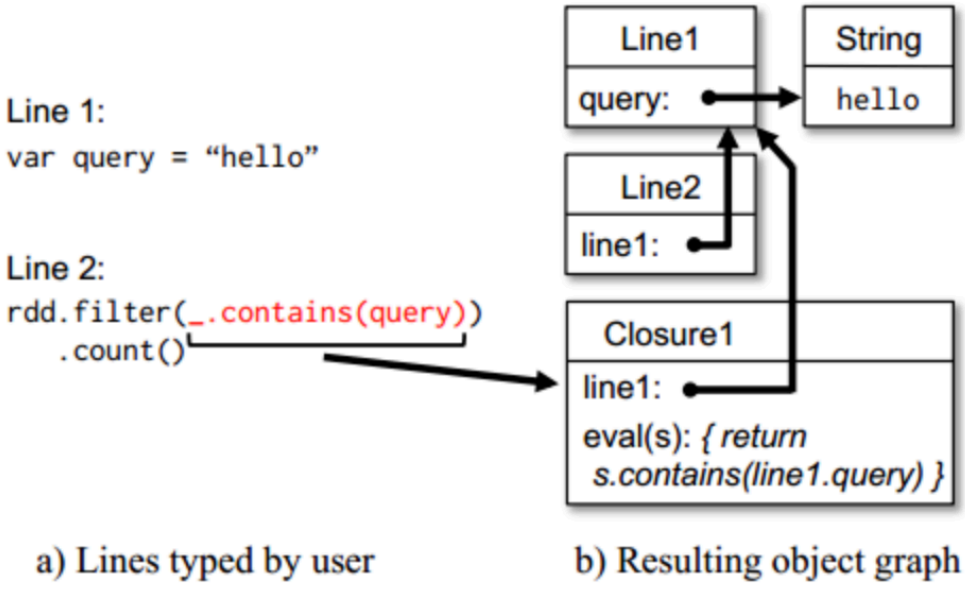
当前Spark版本提供检查点API，但由用户决定是否需要执行检查点操作。今后我们将实现自动检查点，根据成本效益分析确定RDD Lineage图中的最佳检查点位置。

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

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)。



7 Spark解释器如何将用户输入的两行代码解释为Java对象

在Spark中我们对解释器做了两点改动：

1. 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。

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。

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。

7. 评估--RDD可以高效表示的大数据编程模型

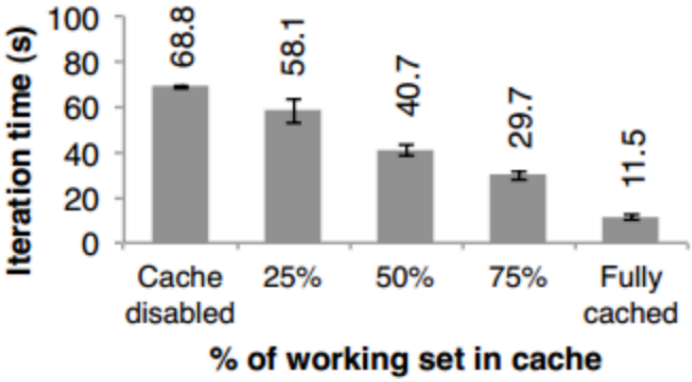
> 高效：可以实现+能够优化

注：实验在Amazon EC2[1]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进行了对比。结果在"1.引言-RDD的性能"部分给出

> 实验大致步骤浅浅了解一下，具体见paper：
>
> 我们基准测试首先从一个运行在Hadoop上的具有迭代特征的机器学习应用和PageRank开始，然后评估在Spark中当工作集不能适应缓存时系统容错恢复能力，最后讨论用户应用程序和交互式数据挖掘的结果。除非特殊说明，我们的实验使用m1.xlarge EC2 节点，4核15GB内存，使用HDFS作为持久存储，块大小为256M。在每个作业运行执行时，为了保证磁盘读时间更加精确，我们清理了集群中每个节点的操作系统缓存。

内存不足时表现

在实验中，我们通过在每个节点上限制缓存RDD所需要的内存资源来配置Spark，在不同的缓存配置条件下执行Logistic回归，我们可以看出，随着缓存的减小，性能平缓地下降。



Spark上运行Logistic回归的性能表现

8.相关的研究知识

> **分布式共享内存（DSM）**

RDD可以看成是一个基于DSM研究[24]得到的抽象。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[19]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**[12]，并**通过推测执行**来解决某些任务计算进度落后的问题

In-Memory集群计算

Piccolo[28]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[26]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[34]、Pig[25]和FlumeJava[9]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[13]、HaLoop[8]实现了一个支持迭代的MapReduce模型；Pregel[21]，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（交互，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[22]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。

Lineage

我们通过参考[6]到[10]做过调研，在科学计算和数据库领域，对于一些应用，如需要解释结果以及允许被重新生成、工作流中发现了bug或者数据集丢失需要重新处理数据，表示数据的Lineage和原始信息一直以来都是一个研究课题。RDD提供了一个受限的编程模型，在这个模型中**使用细粒度的Lineage来表示**是非常容易的，因此它可以被用于容错。

缓存系统

Nectar[14]能够**通过识别带有程序分析的子表达式，跨DryadLINQ作业重用中间结果**，如果将这种能力加入到基于RDD的系统会非常有趣。但是Nectar并没有提供In-Memory缓存，也不能够让用户显式地控制应该缓存那个数据集，以及如何对其进行分区。Ciel[23]同样能够记住任务结果，但不能提供In-Memory缓存并显式控制它。

语言迭代

> DryadLINQ[34]能够**使用LINQ获取到表达式树然后在集群上运行**，Spark系统的**语言集成与它很类似**。不像DryadLINQ，Spark**允许用户显式地跨查询将RDD存储到内存中，并通过控制分区来优化通信**。Spark支持交互式处理，但DryadLINQ却不支持。
>
> **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[29]。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**。