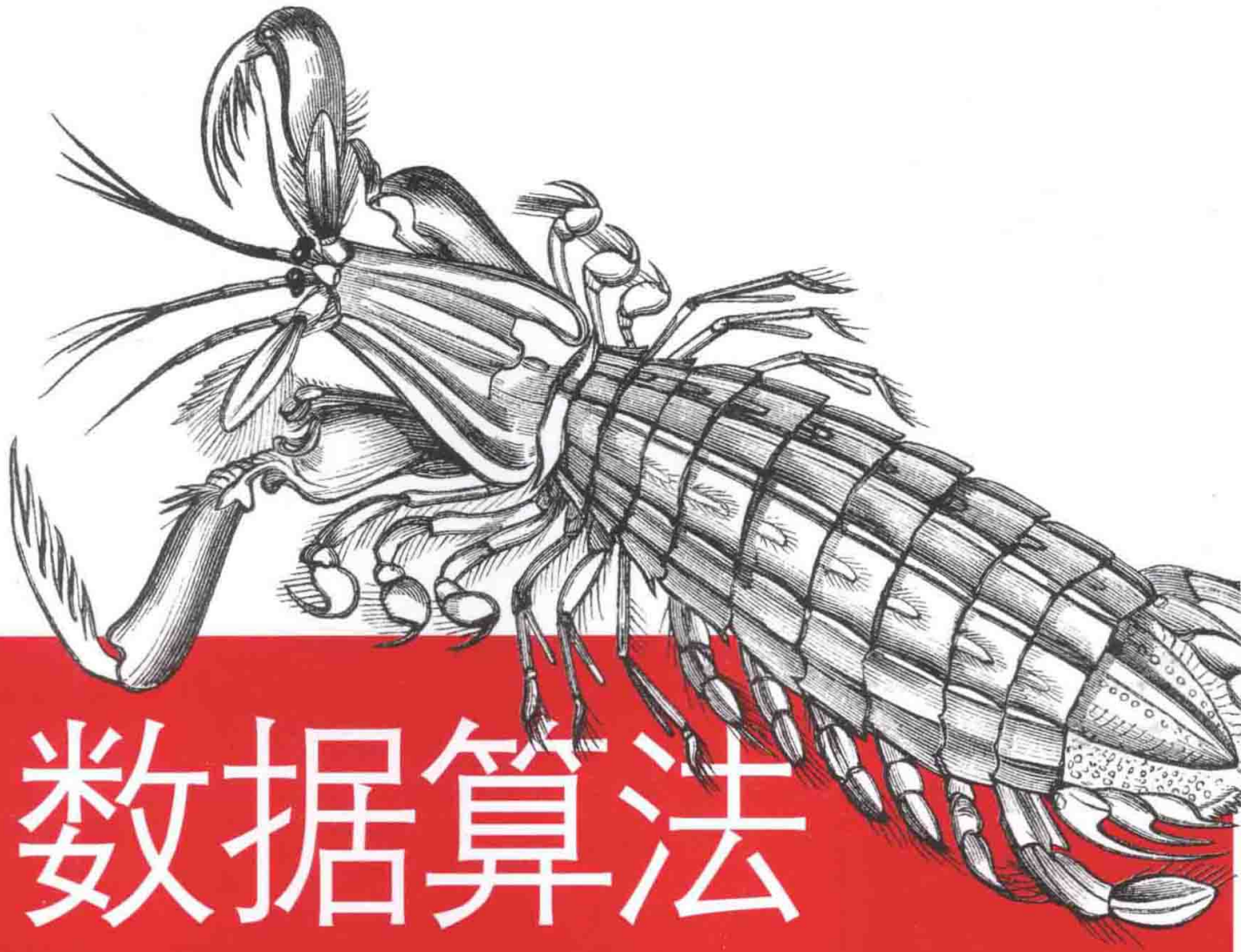
O'REILLY®



# 数据算法

## Hadoop/Spark大数据处理技巧

Data Algorithms

Mahmoud Parsian 著
苏金国 杨健康 等译

中国电力出版社

O'REILLY

# 数据算法：Hadoop/Spark大数据处理技巧

如果你准备深入研究MapReduce框架来处理大数据集，这本书非常实用，通过提供丰富的算法和工具，它会循序渐进地带你探索MapReduce世界，用Apache Hadoop或Apache Spark构建分布式MapReduce应用时通常都需要用到这些算法和工具。每一章分别提供一个实例来解决一个大规模计算问题，如构建推荐系统。你会了解如何用代码实现适当的MapReduce解决方案，而且可以在你的项目中具体应用这些解决方案。

本书介绍了很多基本设计模式、优化技术和数据挖掘及机器学习解决方案，以解决生物信息学、基因组学、统计和社交网络分析等领域的很多问题。这本书还概要介绍了MapReduce、Hadoop和Spark。

**本书的主要内容包括：**

- **完成超大量交易的购物篮分析。**
- **数据挖掘算法（K-均值、KNN和朴素贝叶斯）。**
- **使用超大基因组数据完成DNA和RNA测序。**
- **朴素贝叶斯定理和马尔可夫链实现数据和市场预测。**
- **推荐算法和成对文档相似性。**
- **线性回归、Cox回归和皮尔逊（Pearson）相关分析。**
- **等位基因频率和DNA挖掘。**
- **社交网络分析（推荐系统、三角形计数和情感分析）。**

**Mahmoud Parsian**，计算机科学博士，是一位热衷于实践的软件专家，作为开发人员、设计人员、架构师和作者，他有30多年的软件开发经验。目前领导着Illumina的大数据团队，在过去15年间，他主要从事Java (服务器端)、数据库、MapReduce和分布式计算的有关工作。Mahmoud还著有《JDBC Recipes》和《JDBC Metadata, MySQL,and Oracle Recipes》等书（均由Apress出版）。




DATA/MATH

ISBN 978-7-5123-9594-7



定价：128.00元

O'Reilly Media, Inc.授权中国电力出版社出版

此简体中文版仅限于在中华人民共和国境内（但不允许在中国香港、澳门特别行政区和中国台湾地区）销售发行
This Authorized Edition for sale only in the territory of People's Republic of China (excluding Hong Kong, Macao and Taiwan)

# 数据算法：
## Hadoop/Spark大数据处理技巧

*Mahmoud Parsian* 著
苏金国 杨健康 等译

*Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo* O'REILLY®

O'Reilly Media, Inc.授权中国电力出版社出版

中国电力出版社

**图书在版编目（CIP）数据**

数据算法：Hadoop/Spark大数据处理技巧/（美）马哈默德·帕瑞斯安（Mahmoud Parsian）著；苏金国等译. —北京：中国电力出版社，2016.10

书名原文：Data Algorithms

ISBN 978-7-5123-9594-7

I. ①数… II. ①马… ②苏… III. ①数据处理－算法分析 IV. ①TP274

中国版本图书馆CIP数据核字（2016）第174889号

北京市版权局著作权合同登记

图字：01-2016-4204号

Copyright ©2015 by Mahmoud Parsian. All rights reserved.

Simplified Chinese Edition, jointly published by O'Reilly Media, Inc. and China Electric Power Press, 2016. Authorized translation of the English edition, 2015 O'Reilly Media, Inc., the owner of all rights to publish and sell the same.

All rights reserved including the rights of reproduction in whole or in part in any form.

英文原版由O'Reilly Media, Inc. 出版2015。

简体中文版由中国电力出版社出版2016。英文原版的翻译得到O'Reilly Media, Inc.的授权。此简体中文版的出版和销售得到出版权和销售权的所有者 —— O'Reilly Media, Inc.的许可。

版权所有，未得书面许可，本书的任何部分和全部不得以任何形式重制。

封面设计/ Ellie Volckhausen，张健

出版发行/ 中国电力出版社（http://www.cepp.sgcc.com.cn）

地　　址/ 北京市东城区北京站西街19号（邮政编码100005）

经　　销/ 全国新华书店

印　　刷/ 北京天宇星印刷厂

开　　本/ 787毫米×980毫米　16开本　43.5印张　834千字

版　　次/ 2016年10月第一版　2016年10月第一次印刷

印　　数/ 0001－3000册

定　　价/ 128.00元（册）

**敬 告 读 者**

本书封底贴有防伪标签，刮开涂层可查询真伪

本书如有印装质量问题，我社发行部负责退换

**版 权 专 有　翻 印 必 究**

# O'Reilly Media, Inc.介绍

O'Reilly Media通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自1978年开始，O'Reilly一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O'Reilly的发展充满了对创新的倡导、创造和发扬光大。

O'Reilly为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了Make杂志，从而成为DIY革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O'Reilly的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O'Reilly现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版，在线服务或者面授课程，每一项O'Reilly的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

## 业界评论

“O'Reilly Radar博客有口皆碑。”

——Wired

“O'Reilly凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——Business 2.0

“O'Reilly Conference是聚集关键思想领袖的绝对典范。”

——CRN

“一本O'Reilly的书就代表一个有用、有前途、需要学习的主题。”

——Irish Times

“Tim是位特立独行的商人，他不光放眼于最长远、最广阔的视野并且切实地按照Yogi Berra的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去Tim似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——Linux Journal

本书谨献给我亲爱的家人：

我的妻子Behnaz，

我的女儿Maral，

我的儿子Yaseen。

# 目录


序 ..... 1

前言 ..... 3

第1章 二次排序：简介 ..... 19

二次排序问题解决方案 ..... 21

MapReduce/Hadoop的二次排序解决方案 ..... 25

Spark的二次排序解决方案 ..... 29

第2章 二次排序：详细示例 ..... 42

二次排序技术 ..... 43

二次排序的完整示例 ..... 46

运行示例——老版本Hadoop API ..... 50

运行示例——新版本Hadoop API ..... 52

第3章 Top 10列表 ..... 54

Top N设计模式的形式化描述 ..... 55

MapReduce/Hadoop实现：唯一键 ..... 56

Spark实现：唯一键 ..... 62

Spark实现：非唯一键 ..... 73

使用takeOrdered()的Spark Top 10解决方案 ..... 84

MapReduce/Hadoop Top 10解决方案：非唯一键 ..... 91

第4章 左外连接 ..... 96
左外连接示例 ..... 96
MapReduce左外连接实现 ..... 99
Spark左外连接实现 ..... 105
使用leftOuterJoin()的Spark实现 ..... 117
第5章 反转排序 ..... 127
反转排序模式示例 ..... 128
反转排序模式的MapReduce/Hadoop实现 ..... 129
运行示例 ..... 134
第6章 移动平均 ..... 137
示例1：时间序列数据（股票价格） ..... 137
示例2：时间序列数据（URL访问数） ..... 138
形式定义 ..... 139
POJO移动平均解决方案 ..... 140
MapReduce/Hadoop移动平均解决方案 ..... 143
第7章 购物篮分析 ..... 155
MBA目标 ..... 155
MBA的应用领域 ..... 157
使用MapReduce的购物篮分析 ..... 157
Spark解决方案 ..... 166
运行Spark实现的YARN脚本 ..... 179
第8章 共同好友 ..... 182
输入 ..... 183
POJO共同好友解决方案 ..... 183
MapReduce算法 ..... 184
解决方案1: 使用文本的Hadoop实现 ..... 187
解决方案2: 使用ArrayListOfLongsWritable的Hadoop实现 ..... 189
Spark解决方案 ..... 191

第9章 使用MapReduce实现推荐引擎 ..... 201
购买过该商品的顾客还购买了哪些商品 ..... 202
经常一起购买的商品 ..... 206
推荐连接 ..... 210
第10章 基于内容的电影推荐 ..... 225
输入 ..... 226
MapReduce阶段1 ..... 226
MapReduce阶段2和阶段3 ..... 227
Spark电影推荐实现 ..... 234
第11章 使用马尔可夫模型的智能邮件营销 ..... 253
马尔可夫链基本原理 ..... 254
使用MapReduce的马尔可夫模型 ..... 256
Spark解决方案 ..... 269
第12章 K-均值聚类 ..... 282
什么是K-均值聚类? ..... 285
聚类的应用领域 ..... 285
K-均值聚类方法非形式化描述：分区方法 ..... 286
K-均值距离函数 ..... 286
K-均值聚类形式化描述 ..... 287
K-均值聚类的MapReduce解决方案 ..... 288
K-均值算法Spark实现 ..... 292
第13章 k-近邻 ..... 296
kNN分类 ..... 297
距离函数 ..... 297
kNN示例 ..... 298
kNN算法非形式化描述 ..... 299
kNN算法形式化描述 ..... 299
kNN的类Java非MapReduce 解决方案 ..... 299
Spark的kNN算法实现 ..... 301

第14章 朴素贝叶斯 ..... 315
训练和学习示例 ..... 316
条件概率 ..... 319
深入分析朴素贝叶斯分类器 ..... 319
朴素贝叶斯分类器：符号数据的MapReduce解决方案 ..... 322
朴素贝叶斯分类器Spark实现 ..... 332
使用Spark和Mahout ..... 347
第15章 情感分析 ..... 349
情感示例 ..... 350
情感分数：正面或负面 ..... 350
一个简单的MapReduce情感分析示例 ..... 351
真实世界的情感分析 ..... 353
第16章 查找、统计和列出大图中的所有三角形 ..... 354
基本的图概念 ..... 355
三角形计数的重要性 ..... 356
MapReduce/Hadoop解决方案 ..... 357
Spark解决方案 ..... 364
第17章 K-mer计数 ..... 375
K-mer计数的输入数据 ..... 376
K-mer计数应用 ..... 376
K-mer计数MapReduce/Hadoop解决方案 ..... 377
K-mer计数Spark解决方案 ..... 378
第18章 DNA测序 ..... 390
DNA测序的输入数据 ..... 392
输入数据验证 ..... 393
DNA序列比对 ..... 393
DNA测试的MapReduce算法 ..... 394

第19章 Cox回归 ..... 413
Cox模型剖析 ..... 414
使用R的Cox回归 ..... 415
Cox回归应用 ..... 416
Cox回归 POJO解决方案 ..... 417
MapReduce输入 ..... 418
使用MapReduce的Cox回归 ..... 419
第20章 Cochran-Armitage趋势检验 ..... 426
Cochran-Armitage算法 ..... 427
Cochran-Armitage应用 ..... 432
MapReduce解决方案 ..... 435
第21章 等位基因频率 ..... 443
基本定义 ..... 444
形式化问题描述 ..... 448
等位基因频率分析的MapReduce解决方案 ..... 449
MapReduce解决方案, 阶段1 ..... 449
MapReduce解决方案，阶段2 ..... 459
MapReduce解决方案, 阶段3 ..... 463
染色体X 和Y的特殊处理 ..... 466
第22章 T检验 ..... 468
对bioset完成T检验 ..... 469
MapReduce问题描述 ..... 472
输入 ..... 472
期望输出 ..... 473
MapReduce解决方案 ..... 473
Spark实现 ..... 476
第23章 皮尔逊相关系数 ..... 488
皮尔逊相关系数公式 ..... 489
皮尔逊相关系数示例 ..... 491

皮尔逊相关系数数据集 ..... 492
皮尔逊相关系数POJO 解决方案 ..... 492
皮尔逊相关系数MapReduce解决方案 ..... 493
皮尔逊相关系数的Spark 解决方案 ..... 496
运行Spark程序的YARN脚本 ..... 516
使用Spark计算斯皮尔曼相关系数 ..... 517

第24章 DNA碱基计数 ..... 520
FASTA格式 ..... 521
FASTQ格式 ..... 522
MapReduce解决方案：FASTA格式 ..... 522
运行示例 ..... 524
MapReduce解决方案: FASTQ格式 ..... 528
Spark 解决方案: FASTA格式 ..... 533
Spark解决方案: FASTQ格式 ..... 537

第25章 RNA测序 ..... 543
数据大小和格式 ..... 543
MapReduce工作流 ..... 544
RNA测序分析概述 ..... 544
RNA测序MapReduce算法 ..... 548

第26章 基因聚合 ..... 553
输入 ..... 554
输出 ..... 554
MapReduce解决方案（按单个值过滤和按平均值过滤） ..... 555
基因聚合的Spark解决方案 ..... 567
Spark解决方案：按单个值过滤 ..... 567
Spark解决方案：按平均值过滤 ..... 576

第27章 线性回归 ..... 586
基本定义 ..... 587
简单示例 ..... 587

问题描述 ..... 588
输入数据 ..... 589
期望输出 ..... 590
使用SimpleRegression的MapReduce解决方案 ..... 590
Hadoop实现类 ..... 593
使用R线性模型的MapReduce解决方案 ..... 593

第28章 MapReduce和幺半群 ..... 600
概述 ..... 600
幺半群的定义 ..... 602
幺半群和非幺半群示例 ..... 603
MapReduce示例：非幺半群 ..... 606
MapReduce示例：幺半群 ..... 608
使用幺半群的Spark示例 ..... 612
使用幺半群的结论 ..... 618
函子和幺半群 ..... 619

第29章 小文件问题 ..... 622
解决方案1：在客户端合并小文件 ..... 623
解决方案2：用CombineFileInputFormat解决小文件问题 ..... 629
其他解决方案 ..... 634

第30章 MapReduce的大容量缓存 ..... 635
实现方案 ..... 636
缓存问题形式化描述 ..... 637
一个精巧、可伸缩的解决方案 ..... 637
实现LRUMap缓存 ..... 640
使用LRUMap的MapReduce解决方案 ..... 646

第31章 Bloom过滤器 ..... 651
Bloom过滤器性质 ..... 651
一个简单的Bloom过滤器示例 ..... 653

Guava 库中的Bloom过滤器 ..... 654
MapReduce中使用Bloom过滤器 ..... 655
附录A Bioset ..... 657
附录B Spark RDD ..... 659
参考书目 ..... 677

# 序

破解基因组的奥秘实在是奥妙无穷，它把知识、思维方法和科技能力紧密融合在一起，将带来变革性的发展。不过，这种变革还需要联合和协同，而协同总少不了深层次的协作。从科学家到软件工程师，从学术界到企业界，我们需要通力合作，朝着基因主导的未来稳步前进。

人们开发了大量数据算法来分析大规模基因测序研究生成的庞大信息，这些数据算法的建立正是这个变革的关键。遗传变异形态各异，可能相当复杂，也可能是全新的，这就要求以一种有效的方式将遗传变异与个人的外在表现联系起来，才能建立并适当地应用临床视点。我们需要提升能力，能够针对更大的规模、跨种群地完成这个工作，这一点至关重要。这本书中提供的方法就像一个指南针，可以指导我们在这条路上顺利前行。

MapReduce、Hadoop和Spark是帮助我们大规模使用基因测序以及存储、处理和分析基因组“大数据”的关键技术。Mahmoud的这本书采用一种简明而实用的方式介绍了这些内容。本书就像一盏灯，为数据科学家、软件工程师以及临床医生照亮了破解基因组奥秘的道路，在这本书的帮助下，人类健康将进入一个讲求精确、个性化和转化的新纪元。

——Jay Flatley
Illumina公司CEO

# 前言

随着大规模搜索引擎（如Google和Yahoo!）、基因组分析（DNA测序、RNA测序和生物标志物分析）以及社交网络（如Facebook和Twitter) 的不断发展，需要生成和处理的数据量已经超过了千万亿字节。为了满足如此庞大的计算需求，我们需要高效、可伸缩的并行算法。MapReduce范式就是解决这些问题的一个框架。

MapReduce是一个软件框架，可以采用并行、分布式方式处理GB、TB，甚至PB级的大数据集，同时它也是一个在商用服务器集群之上完成大规模数据处理的执行框架。实现MapReduce的方法有很多，不过这本书中我们主要关注Apache Spark和MapReduce/Hadoop。你将通过简单而具体的示例来了解如何用Spark和Hadoop实现MapReduce。

这本书将为以下领域提供了基本分布式算法（分别用MapReduce、Hadoop和Spark实现），并按照这些领域组织本书的章节：

- 基本设计模式。
- 数据挖掘和机器学习。
- 生物信息、基因组和统计。
- 优化技术。

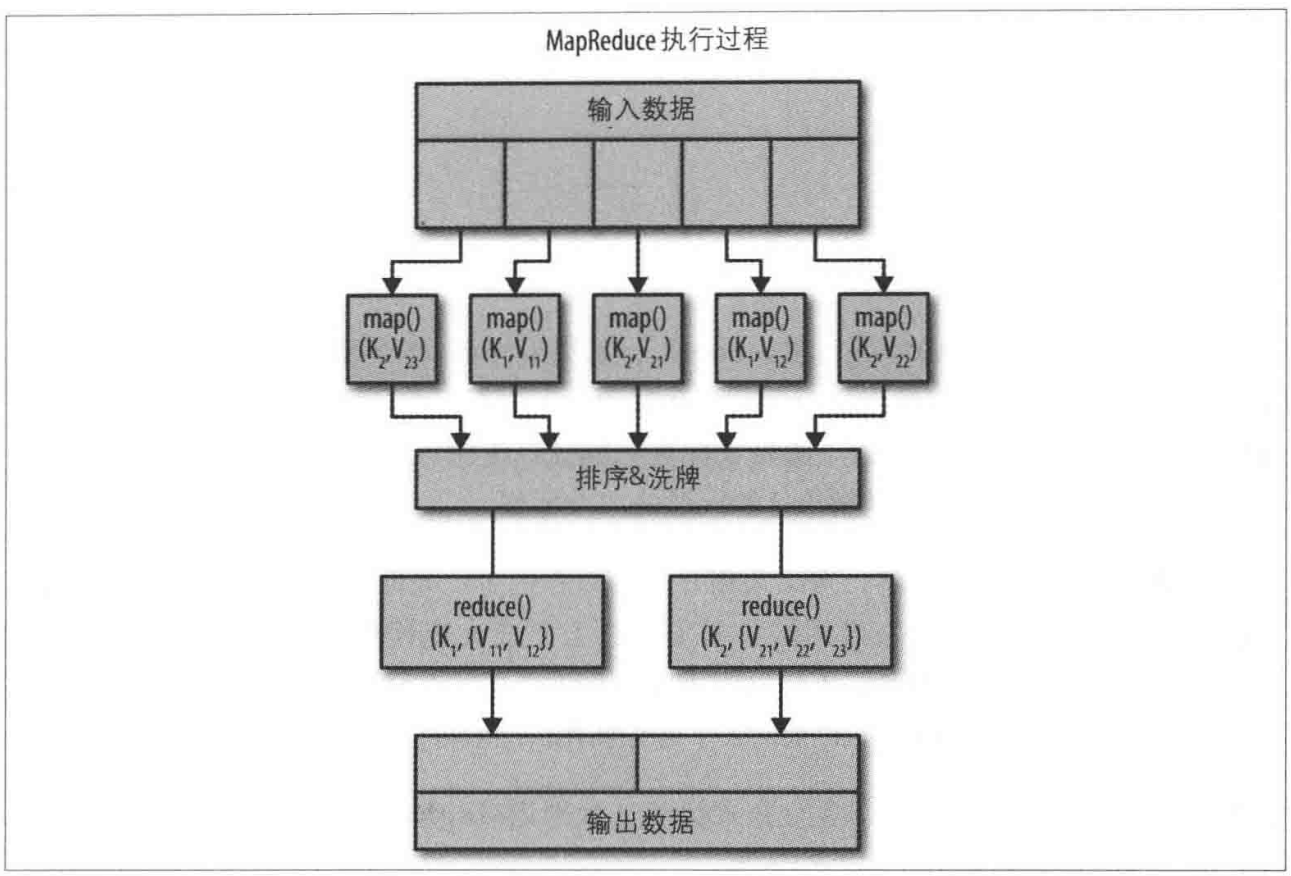
## MapReduce是什么？

MapReduce是一种编程范式，可以利用集群环境的成百上千台服务器实现强大的可伸缩性。MapReduce一词最早源于函数式编程，由Google在一篇名为“MapReduce: Simplified Data Processing on Large Clusters”的文章中率先提出。Google的MapReduce[8]实现是一个专用解决方案，还没有向公众发布。

图P-1给出了MapReduce执行过程的简单视图。简单地说，MapReduce的目标就是实现可伸缩性。使用MapReduce范式时，重点是编写两个函数：

`map()`
: 过滤和聚集数据。

`reduce()`
: 根据`map()`生成的键完成归约、分组和总结。




图P-1：MapReduce执行过程的简单视图

这两个函数定义如下：

`map()`*函数*
: 主节点得到输入，将输入划分为较小的数据块，再将这些小数据块分布到工作节点（从节点）。工作节点对各个数据块应用相同的转换函数，然后将结果传回到主节点。在MapReduce中，程序员要定义一个映射器（mapper），签名如下：

```
map(): (Key1, Value1) → [(Key2, Value2)]
```

`reduce()`*函数*
: 主节点根据唯一的键-值对将接收到的结果进行洗牌和聚集；然后再一次重新分布到工作节点/从节点，通过另一类转换函数组合这些值。在MapReduce中，程序员要定义一个归约器（reducer），签名如下：

```
reduce(): (Key2, [Value2]) → [(Key3, Value3)]
```



这本书中会使用map()函数和reduce()函数的非形式化表示，我会用中括号（[]）表示列表。

在图P-1中，输入数据划分为小块（这里有5个输入分区），每个小块分别发送给一个映射器。各个映射器会生成任意数目的键-值对。映射器的输出如表P-1所示。

**表P-1：映射器的输出**

| 键 | 值 |
|---|---|
| $K_1$ | $V_{11}$ |
| $K_2$ | $V_{21}$ |
| $K_1$ | $V_{12}$ |
| $K_2$ | $V_{22}$ |
| $K_2$ | $V_{23}$ |

在这个例子中，所有映射器只生成两个唯一的键：$\{K_1, K_2\}$。所有映射器的工作完成时，这些键会经过排序、洗牌、分组，然后发送给归约器。最后，归约器将生成所需的输出。对于这个例子，我们有两个归约器，分别用$\{K_1, K_2\}$键标识（见表P-2）。

**表P-2：归约器的输入**

| 键 | 值 |
|---|---|
| $K_1$ | $\{V_{11}, V_{12}\}$ |
| $K_2$ | $\{V_{21}, V_{22}, V_{23}\}$ |

一旦所有映射器的工作完成，归约器就会开始它们的执行过程。每个归约器创建的输出可以包含任意数目的新键-值对（可以0个，也可以多个）。

编写map()函数和reduce()函数时，要确保解决方案是可伸缩的。例如，如果使用了某种数据结构（如List、Array或HashMap），而这种数据结构不能方便地存放在商用服务器的内存中，这个解决方案就不具有可伸缩性。需要说明，map()和reduce()函数都在基础的商用服务器中执行，这些商用服务器可能至多有32GB或64GB RAM（注意，这只是一个例子。如今的服务器可能已经有256GB或512GB RAM，接下来几年中基础服务器可能甚至有1TB的RAM）。因此，可伸缩性是MapReduce的核心。如果你的MapReduce解决方案不能很好的伸缩，就不能称为一个MapReduce解决方案。在这里，我们谈到可伸缩性时，是指“横向扩容”（scaling out），横向扩容表示在系统中增加更多商用节点。MapReduce主要考虑横向扩容（而不是纵向扩容（scaling up），这表示为单个节点增加

资源，比如内存和CPU）。例如，如果DNA测序需要3台服务器用60小时完成，横向扩容到50台类似的服务器时，可以在不到2小时的时间内完成同样的DNA测序。

MapReduce的核心概念是将输入数据集映射到一个键-值对集合，然后对所有包含相同键的键-值对完成归约。尽管基本概念很简单，不过如果考虑以下方面，可以看到这个概念确实很强大、很有效：

- 几乎所有数据都可以映射到键-值对。
- 键和值可以是任意类型：String、Integer、FASTQ（用于DNA测序）、用户自定义的定制类型，当然，也可以是键-值对本身。

MapReduce如何在一组服务器上扩展？MapReduce是如何工作的？关键在于，从概念上讲，MapReduce的输入是一个记录列表（每个记录可以是一行或多行数据）。这些输入记录会划分并传递到集群中的多台服务器，由map()函数使用。map()计算的结果是一个键-值对列表。然后reduce()函数取各个包含相同键的值集，将它们分别组合为一个值（或一个值集）。换句话说，map()函数是由一组数据块生成键-值对，reduce()则是组合map()生成的数据输出，从而能得到所需要的结果，而不是一组键-值对。

MapReduce的主要优点之一是它的“不共享”（shared-nothing）数据处理平台。这意味着所有映射器可以独立地工作，而且映射器完成它们的任务时，归约器也能独立地开始工作（映射器或归约器之间不共享任何数据或临界区。如果有临界区，这会减慢分布式计算的速度）。基于这种“不共享”范式，我们可以很容易地编写map()函数和reduce()函数，而且能轻松、有效地提高并行性。

## MapReduce的简单解释

如何给出MapReduce的一个简单解释？假设我们想要统计一个图书馆的藏书数量，这个图书馆拥有1000个书架，我们要把最后的结果报告给图书管理员。下面给出两种可能的MapReduce解决方案：

- 方案1（使用map()和reduce()）：
  - —map()：聘请1000个工人；每个工人统计一个书架的藏书数量。
  - —reduce()：所有工人集合在一起，把他们各自的统计结果汇总起来（向图书管理员报告他们的统计结果）。
- 方案2（使用map()，combine()和reduce()）：
  - —map()：聘请1110个员工（1000名工人，100位经理，10位主管（每位主管负责管理10个经理，每个经理要管理10名工人）；每个工人负责统计一个书架，将结果报告给他的经理。

—combine()：每个主管负责的10个经理将各自的统计结果汇总报告给这位主管。

—reduce()：所有主管集合起来，把他们各自的统计结果汇总起来（向图书管理员报告他们的统计结果）。

## 什么时候使用MapReduce

MapReduce永远适用吗？答案是否定的。假设我们得到了大数据，如果可以划分这个数据，每个分区能独立地进行处理，就可以考虑使用MapReduce算法。例如，由于图算法采用迭代方法，所以MapReduce并不是很适合。不过，如果要对大量数据完成分组或聚集，MapReduce范式就非常适用。要想使用MapReduce来处理图，可以考虑Apache Giraph（*http://giraph.apache.org/*）和Apache Spark GraphX（*https://spark.apache.org/graphx/*）项目。

下面这些情况也不适合使用MapReduce：

- 一个值的计算依赖于之前计算的值。比如，斐波那契数列就是一个很好的例子，其中每个值都是前两个值之和：

```
F(k + 2) = F(k + 1) + F(k)
```

- 数据集很小，完全可以在一台机器上计算。这种情况下，最好作为一个reduce(map(data))操作来完成，而不需要经过整个MapReduce执行过程。
- 需要同步来处理共享数据。
- 所有输入数据都可以放在内存中。
- 一个操作依赖其他操作。
- 基本计算是处理器敏感型操作。

不过，很多情况下MapReduce都很适用，如：

- 必须处理大量输入数据（例如，对大量数据完成聚集或计算统计结果）。
- 需要利用并行分布式计算、数据存储和数据本地化。
- 可以独立地完成很多任务而无需同步。
- 可以利用排序和洗牌。
- 需要容错性，不能接受作业失败。

## MapReduce不是什么

MapReduce是实现分布式计算的一项开创性技术，不过这个技术还蒙着很多神秘的面纱，这里就来揭开它的一些面纱：

- MapReduce不是一个编程语言，而是一个框架，可以使用Java、Scala和其他编程语言在这个框架上开发分布式应用。
- MapReduce的分布式文件系统不能取代关系型数据库管理系统（如MySQL或Oracle）。一般地，MapReduce的输入是纯文本文件（一个映射器输入记录可以有一行或多行）。
- MapReduce框架主要设计用于批处理，所以不要期望能够在几秒钟内迅速得到结果。不过，如果适当地使用集群，确实可以得到近实时的响应。
- 并不能把MapReduce作为所有软件问题的解决方案。

## 为什么使用MapReduce?

前面我们讨论过，MapReduce的目标是通过增加更多的商用服务器来实现“横向扩容”。这与“纵向扩容”是不同的（纵向扩容是为系统中的单个节点增加更多资源，如内存和CPU）。纵向扩容可能成本很高，有时由于成本以及软件或硬件限制等原因，可能根本无法增加更多的资源。有时人们会提出一些基于主存的新兴算法来解决数据问题，但是由于主存是一个瓶颈，所以这些算法缺乏可伸缩性。例如，在DNA测序分析中，可能需要超过512GB的RAM，这非常昂贵，而且不具有可伸缩性。

如果要提高计算能力，可能需要把计算分布到多个机器上完成。例如，要完成500GB样本数据的DNA测序，可能需要一个服务器计算4天才能完成比对阶段。利用MapReduce，60台服务器可以把计算时间锐减到2小时以内。要处理大量的数据，必须能够将这些数据划分为小块来进行处理，再组合得到最终的结果。MapReduce/Hadoop和Spark/Hadoop可以帮助你提高计算能力，你只需要写两个函数：`map()`和`reduce()`。显然，MapReduce范式为数据分析领域提供了一个强大的新工具，近来，归功于Hadoop等开源解决方案，这个新工具得到了越来越多的关注。

基本说来，MapReduce提供了以下优点：

- 编程模型+基础架构。
- 能够编写在数百甚至上千台机器上运行的程序。
- 自动并行化和分布
- 容错（如果一台服务器宕机，作业可以由其他服务器完成）。
- 程序/作业调度、状态检查和监控。

## Hadoop和Spark

Hadoop（*http://hadoop.apache.org/*）是MapReduce应用实现的事实标准。它由一个或多

个主节点和任意多个从节点组成。Hadoop提出“数据中心就是计算机”，通过提供map()函数和reduce()函数（由程序员定义），允许应用开发人员或程序员利用这些数据中心，从而简化分布式应用。Hadoop高效地实现了MapReduce范式，很容易学习，这是一个可以处理TB甚至PB级大数据的强大工具。

在这本书中，大部分MapReduce算法都采用实例的形式给出（已编译的完整实用解决方案），并且在Java/MapReduce/Hadoop和/或Java/Spark/Hadoop中得到实现。Hadoop和Spark（*http://spark.apache.org/*）框架是开源的，允许我们在分布式环境中完成大数据量的计算和数据处理。

这些框架通过提供“横向扩容”方法来支持可伸缩性，可以采用MapReduce范式在数千台服务器上运行密集型计算。与Hadoop的API相比，Spark的API提供了更高层的抽象。由于这个原因，只用一个Java驱动器类就可以描述Spark解决方案。

Hadoop和Spark是两个不同的分布式软件框架。Hadoop是一个MapReduce框架，在这个框架上可以运行支持map()、combine()和reduce()函数的作业。MapReduce范式很适合单趟计算［先map()，再reduce()］，不过对于多趟算法效率则很低。Spark不是一个MapReduce框架，不过很容易用来支持MapReduce框架的功能，它提供了一个适当的API可以处理map()和reduce()功能。Spark并不限于先完成映射阶段再完成归约阶段。Spark作业可以是由映射和/或归约/洗牌阶段构成的一个任意DAG（有向无环图）。Spark程序可以使用Hadoop运行，也可以不使用Hadoop，另外Spark可以使用Hadoop分布式文件系统（Hadoop Distributed File System，HDFS）或者其他持久存储来实现输入/输出。基本上，对于一个给定的Spark程序或作业，Spark引擎会创建将在集群上完成的任务阶段所构成的一个有向无环图，Hadoop/MapReduce则不同，它会创建由两个预定义阶段（映射和归约）构成的有向无环图。注意，Spark创建的DAG可以包含任意多个阶段。与Hadoop/MapReduce相比，这使得大多数Spark作业都能更快地完成，因为简单的作业只需要一个阶段就可以完成，更复杂的任务也可以一起完成（包括多个阶段），而不需要划分为多个作业。前面已经提到，相比于MapReduce/Hadoop，Spark的API提供了更高层的抽象。例如，Spark的几行代码可能等价于MapReduce/Hadoop的30～40行代码。

尽管Hadoop和Spark等框架建立一个“不共享”范式基础之上，不过它们确实也支持在所有集群节点上共享不可变的数据结构。在Hadoop中，可以通过Hadoop的Configuration对象将这些值传递给映射器和归约器。除了Broadcast只读对象，Spark还支持只写累加器。Hadoop和Spark为大数据处理提供了以下好处：

*可读性*

Hadoop和Spark支持容错（任何节点宕机不会丢失所需的计算结果）。

*可伸缩性*

Hadoop和Spark支持庞大的服务器集群。

*分布式处理*

在Spark和Hadoop中，输入数据和处理是分布式的（可以全面支持大数据）。

*并行化*

可以在节点集群上并行地执行计算。

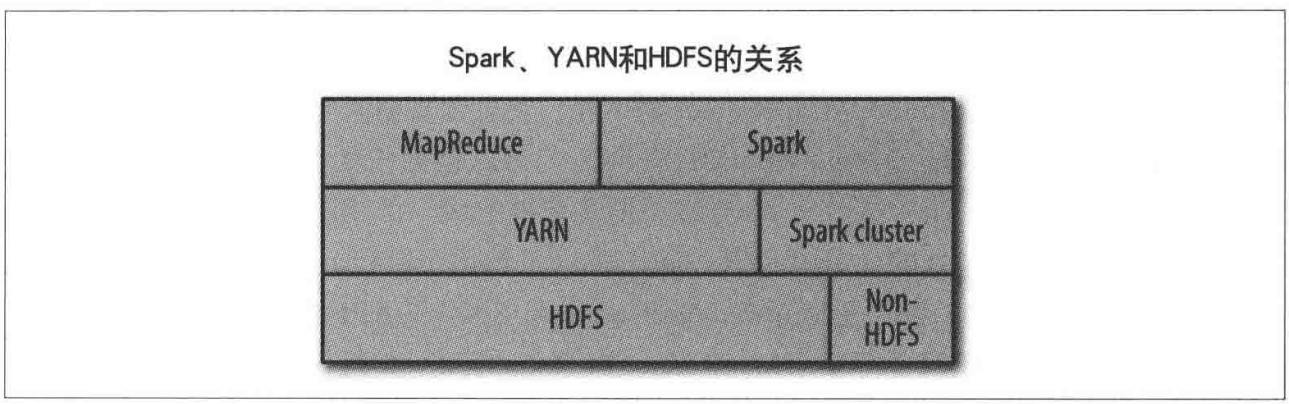
Hadoop主要设计用于批处理，不过如果有足够的内存/RAM，Spark可以用于近实时处理。要了解Spark RDDs（resilient distributed data sets，弹性分布式数据集）的基本用法，可以参考附录B。

那么MapReduce/Hadoop的核心组件有哪些？

- 输入/输出数据包括键-值对。一般地，键是整型、长整型和字符串，而值可以是几乎任何数据类型（字符串、整型、长整型、句子、特殊格式的数据等）。
- 数据划分到商用节点上，填充到数据中心。
- 这个软件会处理故障、重启和其他意外中断。这称为容错（fault tolerance），这也是Hadoop的一个重要特性。

Hadoop和Spark不只是提供了`map()`和`reduce()`功能，还提供了插件模型来实现定制记录读取、二次数据排序及更多其他功能。

图P-2展示了Spark、YARN和Hadoop HDFS之间关系的一个高层视图。




图P-2：MapReduce、Spark和HDFS之间的关系

从这个关系可以看到，使用HDFS （和非HDFS文件系统）运行MapReduce和Spark有很多方法。在这本书中，我会用到以下关键词和术语：

- *MapReduce*表示一般的MapReduce框架范式。
- *MapReduce/Hadoop*表示使用Hadoop的一个特定的MapReduce框架实现。
- *Spark*表示一个特定的Spark实现，它使用HDFS作为持久存储或计算引擎（注意，Spark可以在任何数据存储库上运行，不过这里我们主要强调Hadoop的HDFS）：

— Spark可以脱离Hadoop使用独立的集群节点运行（可以使用HDFS、NFS或其他媒介作为持久数据存储库）。

— Spark也可以结合Hadoop，使用Hadoop的YARN或MapReduce框架运行。

通过这本书，你会循序渐进地学习使用Hadoop构建MapReduce应用所需的算法和工具。MapReduce/Hadoop已经成为处理大数据集（如日志数据、基因组序列、统计应用和社交图谱）的理想编程模型。MapReduce可以用于任何不需要紧耦合并行处理的应用。要记住，Hadoop主要设计用于MapReduce批处理，它并不是一个理想的实时处理解决方案。不要期望在2～5s时间内从Hadoop得到答案，最小的作业也可能需要超过20s的时间。Spark是一个顶级Apache项目，非常适合近实时处理，如果有更多的RAM，它的表现将会更出色。利用Spark，如果使用一个包括100个节点的集群运行一个大数据处理作业（如生物标志物分析或Cox回归），完全有可能在25～35s内处理2亿条记录。一般地，Hadoop作业会有15～20s的延迟，不过这取决于Hadoop集群的大小和配置。

MapReduce实现（如Hadoop）可以在一个庞大的商用计算机集群上运行，具有很好的可伸缩性。例如，一个典型的MapReduce计算过程可以在数百或数千台机器上处理PB或TB级的数据。程序员会发现，MapReduce很容易使用，因为它隐藏了并行化、容错、数据分布和负载平衡等繁杂的细节，使程序员可以集中精力编写两个关键函数`map()`和`reduce()`。

下面是MapReduce/Hadoop/Spark的一些主要应用：

- 查询日志处理。
- 抓取、索引和搜索。
- 分析、文本处理情感分析。
- 机器学习（如马尔可夫链和朴素贝叶斯分类器）。
- 推荐系统。
- 文档聚类和分类。
- 生物信息学（比对、重新校准、生殖细胞采集和DNA/RNA测序）。
- 基因组分析（生物标志物分析以及回归算法，如线性回归和Cox回归）。

# 本书内容

这本书中每一章分别提出一个问题，然后通过一组MapReduce算法加以解决。MapReduce算法/解决方案相当完整（包括MapReduce驱动器、映射器、组合器和归约器程序）。可以在项目中直接使用这些代码（不过，有时可能需要剪切粘贴你需要的部分）。这本书

没有涉及MapReduce框架的底层理论，而是着重于提供使用MapReduce/Hadoop和Spark解决大数据难题的实用算法和示例。这本书的主要内容包括：

- 完成超大量交易的购物篮分析。
- 数据挖掘算法［K-均值、K-近邻（kNN）和朴素贝叶斯］。
- 使用超大量基因组数据完成DNA测序和RNA测序。
- 朴素贝叶斯分类和马尔可夫链实现数据和市场预测。
- 推荐算法和成对文档相似性。
- 线性回归、Cox回归和皮尔逊（Pearson）相关系数。
- 等位基因频率和DNA挖掘。
- 社交网络分析（推荐系统、三角形计数，情感分析）。

你可以复制粘贴这本书中提供的解决方案，使用Hadoop和Spark构建你自己的MapReduce应用和解决方案。所有这些解决方案都经过编译和测试。如果你对Java有一些了解（也就是说，可以读写基本的Java程序），想使用Java/Hadoop/Spark编写和部署MapReduce算法，那么这本书是最适合不过的了。Jimmy Lin和Chris Dyer写过一本好书[16]，其中对MapReduce的一般内容做了详细讨论。重申一次，这本书的目标是使用Hadoop和Spark提供具体的MapReduce算法和解决方案。同样地，这本书也不会详细讨论Hadoop本身。这个内容在Tom White的书[31]中有详细介绍，这也是一本绝妙的好书。

这本书不会介绍如何安装Hadoop或Spark，这里假设你已经安装了这些框架。另外，所有Hadoop命令都相对于Hadoop的安装目录（`$HADOOP_HOME`环境变量）执行。这本书只展示使用MapReduce/Hadoop和Spark的分布式算法。例如，我会讨论API，介绍运行作业的命令行调用，另外会提供完整的实用程序（包括驱动器、映射器、组合器和归约器）。

# 本书重点

这本书的重点是掌握MapReduce范式，并提出一些可以使用MapReduce/Hadoop算法解决的具体问题。对于这里提出的每一个问题，我们会详细介绍`map()`、`combine()`和`reduce()`函数，并提供完整的解决方案，包括：

- 客户端，可以用适当的输入和输出参数调用驱动器。
- 驱动器，明确`map()`和`reduce()`函数，并明确输入和输出。
- 映射器类，实现`map()`函数。
- 组合器类（如果需要），实现`combine()`函数。我们会讨论什么情况下有可能使用组合器。

- 归约器类，实现reduce()函数。

这本书的一个目标是提供一个循序渐进的指南，介绍如何使用Spark和Hadoop作为MapReduce算法的解决方案。另一个目标是展示如何将一个MapReduce作业的输出作为另一个作业的输入（这称为MapReduce作业链或流水线）。

# 本书面向的读者

这本书面向了解Java基础知识并且想使用Hadoop和Spark开发MapReduce算法（数据挖掘、机器学习、生物信息技术、基因组和统计领域）和解决方案的软件工程师、软件架构师、数据科学家和应用开发人员。前面已经提到，这里假设你已经了解Java编程语言的基础知识（例如，知道如何编写类、由一个现有的类定义一个新类，以及使用while循环和if-then-else等基本控制结构）。

更具体地，这本书主要面向以下读者：

- 希望完成大数据分析（分类、回归算法）的数据科学工程师和专业人员。这本书会采用一种实例的形式给出使用大数据应用分类和回归算法的基本步骤。书中会详细介绍map()和reduce()函数，展示如何将它们应用到实际数据，并说明在哪里应用基本设计模式来解决MapReduce问题。对这些MapReduce算法稍做修改就可以很容易地应用于其他行业（例如，修改输入格式）。所有解决方案都已经在Apache Hadoop/Spark中实现，可以根据实际情况调整这些示例。
- 希望设计机器学习算法（如朴素贝叶斯和马尔可夫链算法）的软件工程师和软件架构师。这本书会展示如何构建模型，然后使用MapReduce设计模式将模型应用到新的数据集。
- 希望利用MapReduce使用数据挖掘算法（如K-均值聚类和k-近邻算法）的软件工程师和软件架构师。这里会给出详细的示例，可以指导专业人员实现类似的算法。
- 希望对生物医疗数据应用MapReduce算法（如DNA测序和RNA测序）的数据科学工程师。这本书会清楚地解释生物信息学家和医疗人员可以采用的实用算法。这里提供了适用不同生物数据类型的主要回归/分析算法。这些算法大多都已经部署在实际的生产系统中。
- 希望在MapReduce/分布式环境中应用最重要优化的软件架构师。

这本书假设你已经对Java和Hadoop HDFS有基本的了解。如果需要进一步熟悉Hadoop和Spark，下面这些书可以提供你需要的背景知识：

- *Hadoop: The Definitive Guide*，Tom White著 (O'Reilly)
- *Hadoop in Action*，Chuck Lam著 (Manning Publications)

- *Hadoop in Practice*，Alex Holmes著（Manning Publications）
- *Learning Spark*，Holden Karau, Andy Konwinski, Patrick Wendell和Matei Zaharia著（O'Reilly）

## 在线资源

这本书有两个配套网站：

*https://github.com/mahmoudparsian/data-algorithms-book/*

在这个GitHub网站上，可以找到源代码（按章节组织）、shell脚本（用于运行MapReduce/Hadoop和Spark程序）、用于测试的示例输入文件以及书中未涵盖的一些额外内容（包括附加的两章）的相应链接。

*http://mapreduce4hackers.com*

在这个网站上可以找到额外的源文件（书中未提到）以及书中未涉及的另外一些内容的链接。将来还会更全面地介绍MapReduce/Hadoop/Spark主题。

# 本书中使用的软件

开发这本书中介绍的解决方案和示例时，我使用了表P-3列出的软件和编程环境。

**表P-3：本书中使用的软件/编程环境**

| 软件 | 版本 |
|---|---|
| Java编程语言 (JDK7) | 1.7.0_67 |
| 操作系统：Linux CentOS | 6.3 |
| 操作系统：Mac OS X | 10.9 |
| Apache Hadoop | 2.5.0, 2.6.0 |
| Apache Spark | 1.1.0, 1.3.0, 1.4.0 |
| Eclipse IDE | Luna |

这本书中的所有程序都使用Java/JDK7、Hadoop 2.5.0和Spark (1.1.0，1.3.0, 1.4.0)完成了测试。我们给出了不同操作系统环境（Linux和OS X）下的例子。对于所有示例和解决方案，我都使用了基本的文本编辑器（如vi、vim和TextWrangler），然后使用Java命令行编译器（javac）来完成编译。

这本书中，我使用shell脚本（如bash脚本）来运行示例MapReduce/Hadoop和Spark程序。以$或#字符开头的代码行表示必须在终端提示窗口（如bash）输入这些命令。

# 本书使用约定

以下是本书中使用的排版约定：

*斜体（Italic）*

指示新术语、URL、email地址、文件名和文件扩展名。

等宽字体（`Constant width`）

用于程序代码清单，以及在段落中用来指示程序元素，如变量或函数名、数据库、数据类型、环境变量、语句和关键字。



这表示一个一般说明。

## 使用代码示例

前面已经提到，可以从以下网址下载补充材料（包括代码示例、练习等），*https://github.com/mahmoudparsian/data-algorithms-book/*和*http://www.mapreduce4hackers.com*。

这本书的目的就是要帮助你完成实际工作。一般来讲，如果本书提供了示例代码，完全可以在你的程序和文档中使用这些代码，不需要联系我们来得到许可，除非你直接复制了大部分的代码。例如，如果你在编写一个程序，使用了本书中的多段代码，这并不需要得到许可。但是出售或发行O'Reilly示例代码光盘则需要得到许可。回答问题时如果引用了这本书的文字和示例代码，这不需要得到许可。但是如果你的产品的文档借用了本书中的大量示例代码，则需要得到许可。

我们希望但不严格要求标明引用出处。引用信息通常包括书名、作者、出版商和ISBN。例如，“Data Algorithms by Mahmoud Parsian (O'Reilly). Copyright 2015 Mahmoud Parsian, 978-1-491-90618-7”。

如果你认为在使用代码示例时超出了合理使用范围或者上述许可范围，可以随时联系我们：*permissions@oreilly.com*。

# Safari®图书在线

Safari图书在线（*www.safaribooksonline.com*）是一个应需而变的数字图书馆，通过图书和视频方式提供世界顶尖作者在技术和商业领域积累的专家经验。

技术专家、软件开发人员、Web设计人员和企业以及有创意的专业人员都使用Safari图书在线作为其主要资源来完成研究、解决问题、深入学习和资质培训。

Safari图书在线为机构、政府部门和个人提供了多种产品组合和定价程序。

订阅者可以在一个可以快捷搜索的数据库中访问多家出版社提供的成千上万种图书、培训视频和正式出版前手稿，如O'Reilly Media、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology以及其他数十家出版公司。关于Safari图书在线的更多信息，请访问我们的在线网站。

# 如何联系我们

美国：

O'Reilly Media，Inc.
1005 Gravenstein Highway North
Sebastopol，CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）
奥莱利技术咨询（北京）有限公司

我们为本书提供了网页，该网页上面列出了勘误表、范例和任何其他附加的信息。您可以访问如下网页获得：

*http://oreil.ly/data_algorithms*

要询问技术问题或对本书提出建议，请发送电子邮件至：

*bookquestions@oreilly.com*

要获得更多关于我们的书籍、会议、资源中心和O'Reilly网络的信息，请参见我们的网站：

*http://www.oreilly.com.cn*
*http://www.oreilly.com*

# 致谢

致各位读者：非常感谢你们读这本书。衷心希望这本书对你有用，能为你提供帮助。

感谢我的O'Reilly编辑Ann Spencer，感谢你在这个图书项目中对我的信任，重新组织章节时提供的莫大支持，还要感谢对这个新书名的建议（原先我的书名是《MapReduce for Hackers》）。另外，还要感谢Mike Loukides（O'Reilly Media内容策略部副总裁）对这个项目的信任和支持。

此外，非常感谢我的编辑Marie Beaugureau，作为O'Reilly的数据和开发编辑，长久以来他一直耐心地与我共同战斗，在这个项目的每个阶段都无私地支持着我。Marie的意见和建议非常有用，也很有建设性。

还要特别感谢我的文字编辑Rachel Monaghan，感谢她在图书编辑方面的卓越见识和极为中肯的意见和建议。正是因为有她，这本书才更为通俗易懂。另外，还要感谢产品编辑Matthew Hacker，因为他出色的工作，这本书才得以顺利制作发行。感谢Rebecca Demarest（O'Reilly绘图）和Dan Fauxsmith （O'Reilly 发行服务主管）对这本书的美化。另外，我还要感谢Rachel Head（ 校对）、Judith McConville（索引）、David Futato（内文设计）和Ellie Volckhausen（封面设计）。

感谢我的技术审校Cody Koeninger、Kun Lu、Neera Vats、Dr. Phanendra Babu、Willy Bruns和Mohan Reddy。你们的意见非常中肯，我尽可能地采纳了你们的建议。特别感谢Cody提供的详细反馈。

特别要感谢Jay Flatley（Illumina首席执行官），你提供了一个无与伦比的机会和环境来解开基因组的奥秘。感谢我亲爱的朋友Saeid Akhtari（NextBio首席执行官）和Satnam Alag博士（Illumina工程部副总裁），感谢你们过去5年对我的信任和支持。

感谢我永远的朋友Ramachandran Krishnaswamy博士（我的博士导师），感谢他悉心的指导和为我的计算机科学研究工作提供的良好环境。

感谢我的父母（妈妈Monireh Azemoun和爸爸Bagher Parsian）把教育当作他们的第一要务。感谢他们一直以来的默默支持。感谢我的哥哥Ahmad Parsian博士帮助我理解数学要义。感谢我的妹妹Nayer Azam Parsian让我体会到友爱和关怀。

最后，但绝不是最不重要，要感谢我亲爱的家人（Behnaz、Maral和Yaseen）在写这本书的过程中，你们的鼓励和支持对我的意义绝对无以言表。

# 对本书的建议和问题

如果你对这本书中描述的问题和解决方案有想法，我非常乐于收到有关的反馈和建议。

请把关于这本书的建议和问题通过email发送到*mahmoud.parsian@yahoo.com*。另外也可以通过*http://www.mapreduce4hackers.com*与我联系。

——Mahmoud Parsian
加利福尼亚州森尼韦尔

# 第1章

# 二次排序：简介

二次排序（secondary sort）问题是指在归约阶段对与某个键关联的值排序。有时这也称为值键转换（value-to-key conversion）。利用二次排序技术，可以对传入各个归约器的值完成（升序或降序）排序。我将给出一些具体例子来说明如何按升序或降序实现二次排序。

这一章的目标是用MapReduce/Hadoop和Spark实现二次排序（Secondary Sort）设计模式。在软件设计和编程领域，设计模式是用于解决常见问题的可重用的算法。一般地，设计模式并不限定于采用某种特定的编程语言来表示，完全可以用很多不同的编程语言加以实现。

MapReduce框架会自动对映射器生成的键完成排序。这说明，在启动归约器之前，映射器生成的所有中间键-值对必然是按键有序的（而不是按值有序）。传入各个归约器的值并不是有序的，它们可能有任意的顺序。不过，如果想要对归约器的值进行排序该怎么做呢？MapReduce/Hadoop和Spark不会为归约器完成值排序。所以，如果某些应用（如时间序列数据）希望对归约器数据排序，二次排序设计模式可以为这些应用实现这个目标。

首先，我们会重点介绍MapReduce/Hadoop解决方案。下面来看MapReduce范式，然后详细分析二次排序的概念：

$$\text{map}(key_1, value_1) \rightarrow \text{list}(key_2, value_2)$$
$$\text{reduce}(key_2, \text{list}(value_2)) \rightarrow \text{list}(key_3, value_3)$$

首先，map()函数接收一个键-值对输入（$key_1$, $value_1$）。然后它会输出任意数目的键-值对（$key_2$, $value_2$）。接下来，reduce()函数接收另一个键-值对（$key_2$, list($value_2$)）作为输入，并输出任意数目的键-值对（$key_3$, $value_3$）。

现在考虑下面的键-值对（$key_2$, list($value_2$)），这会作为归约器的输入：

list(value2) = ($V_1$, $V_2$, ..., $V_n$)

这里，归约器值（$V_1$, $V_2$, ..., $V_n$)是无序的。

二次排序模式的目标就是让归约器接收的值有某种顺序。这样一来，对MapReduce范式应用这种模式时，就可以得到：

SORT($V_1$, $V_2$, ..., $V_n$) = ($S_1$, $S_2$, ..., $S_n$)

list($value_2$) = ($S_1$, $S_2$, ..., $S_n$)

在这里：

- $S_1 < S_2 < ... < S_n$ （升序），或
- $S_1 > S_2 > ... > S_n$ （降序）

下面是一个二次排序问题的例子：考虑一个科学试验得到的温度数据。这些温度数据可能如下所示（各个列分别为年（year）、月（month）、日（day）和当天温度（daily temperature））：

```
2012, 01, 01, 5
2012, 01, 02, 45
2012, 01, 03, 35
2012, 01, 04, 10
...
2001, 11, 01, 46
2001, 11, 02, 47
2001, 11, 03, 48
2001, 11, 04, 40
...
2005, 08, 20, 50
2005, 08, 21, 52
2005, 08, 22, 38
2005, 08, 23, 70
```

假设我们希望输出每一个“年-月”（year-month）的温度，而且值按升序排序。实际上，这里希望对归约器值的迭代器排序。因此，我们想要生成类似下面的输出（第一列是“年-月”（year-month），第二列是已排序的温度）：

```
2012-01: 5, 10, 35, 45, ...
2001-11: 40, 46, 47, 48, ...
2005-08: 38, 50, 52, 70, ...
```

# 二次排序问题解决方案

归约器值排序至少有两种方案。这些解决方案在MapReduce/Hadoop和Spark框架中都可以应用：

- 第一种方案是让归约器读取和缓存给定键的所有值（例如，缓存到一个数组数据结构中），然后对这些值完成一个归约器中（in-reducer）排序。这种方法不具有可伸缩性，因为归约器要接收一个给定键的所有值，这种方法可能导致归约器耗尽内存（`java.lang.OutOfMemoryError`）。另一方面，如果值数量很少，不会导致内存溢出错误，那么这种方法就是适用的。
- 第二种方案是使用MapReduce框架对归约器值排序（这样一来，就不再需要对传入归约器的值完成归约器中排序）。这种方法“会为自然键增加部分或整个值来创建一个组合键以实现排序目标”。关于这种方法的详细内容，可以参考“Java Code Geeks”（*http://bit.ly/secondary_sorting*）。这种方法是可伸缩的，不会产生内存溢出错误。在这里，排序工作基本上交由MapReduce框架来完成（实际上排序正是MapReduce/Hadoop框架的一个突出特性）。

  下面对第二种方法做个小结：

  1. 使用值键转换设计模式：构造一个组合中间键（K，$V_1$），其中$V_1$是次键（secondary key）。在这里，K称为自然键（natural key）。要在归约器键中注入一个值（即$V_1$），只需要创建一个组合键（详细内容参见`DateTemperaturePair`类）。在我们的例子中，$V_1$就是温度数据（temperature）。
  2. 让MapReduce执行框架完成排序（而不是在内存中排序，要让框架使用集群节点来完成排序）。
  3. 保留多个键-值对的状态来完成处理，可以利用适当的映射器输出分区器来实现这一点（例如，我们会按自然键对映射器的输出分区）。

## 实现细节

要实现二次排序特性，还需要另外一些Java插件类。我们要告诉MapReduce/Hadoop框架：

- 如何对归约器键排序。
- 如何对传入归约器的键分区（定制分区器）。
- 如何对到达各个归约器的数据分组。

## 中间键的排序顺序

要实现二次排序，我们需要控制中间键的排序顺序，以及归约器处理键的顺序。首先，要在组合键中注入一个值（temperature数据），然后控制中间键的排序顺序。自然键、组合键和键-值对之间的关系如图1-1所示。

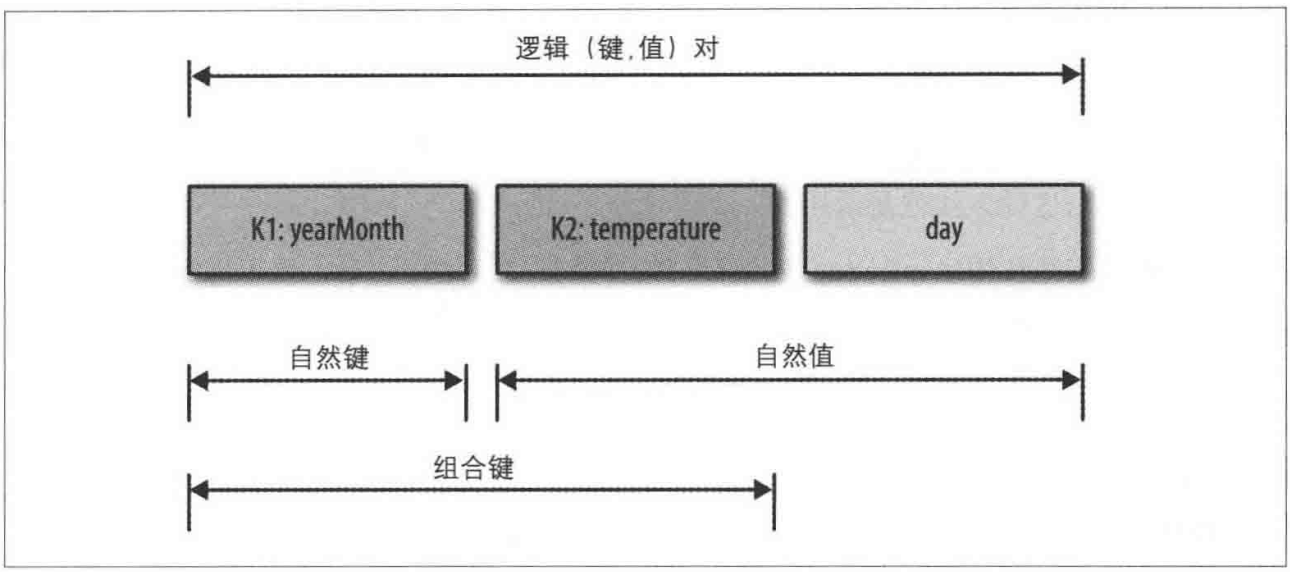



图1-1：二次排序键

主要问题是，要为自然键增加什么值来实现二次排序。在这里，答案是temperature数据字段（因为我们希望归约器值按temperature排序）。所以，要用compareTo()方法指出如何对DateTemperaturePair对象排序。要定义一个适当的数据结构保存键和值，另外还需要提供中间键的排序顺序。在Hadoop中，如果需要持久存储定制数据类型（如DateTemperaturePair），则必须实现Writable接口；如果要比较定制数据类型，它们还必须实现另外一个接口WritableComparable（参见示例1-1）。

示例1-1：DateTemperaturePair类

```
1 import org.apache.hadoop.io.Writable;
2 import org.apache.hadoop.io.WritableComparable;
3 ...
4 public class DateTemperaturePair
5     implements Writable, WritableComparable<DateTemperaturePair> {
6
7      private Text yearMonth = new Text(); // 自然键
8      private Text day = new Text();
9      private IntWritable temperature = new IntWritable(); // 次键
10
11     ...
12
13     @Override
14     /**
15      * 这个比较器将控制键的排序顺序
16      */
17     public int compareTo(DateTemperaturePair pair) {
```

```
18          int compareValue = this.yearMonth.compareTo(pair.getYearMonth());
19          if (compareValue == 0) {
20              compareValue = temperature.compareTo(pair.getTemperature());
21          }
22          //return compareValue; // 升序排序
23          return -1*compareValue; // 降序排序
24      }
25      ...
26  }
```

## 定制分区器

基本说来，分区器会根据映射器的输出键来决定哪个映射器的输出发送到哪个归约器。为此，我们需要两个插件类：首先需要一个定制分区器控制哪个归约器处理哪些键，另外还需要一个定制比较器（Comparator）对归约器值排序。这个定制分区器可以确保有相同键（自然键，而不是包含temperature值的组合键）的所有数据都将发送给同一个归约器。定制比较器会完成排序，保证一旦数据到达归约器，就会按自然键（year-month）对数据分组。

示例1-2：DateTemperaturePartitioner类

```
1 import org.apache.hadoop.io.Text;
2 import org.apache.hadoop.mapreduce.Partitioner;
3
4 public class DateTemperaturePartitioner
5    extends Partitioner<DateTemperaturePair, Text> {
6
7     @Override
8     public int getPartition(DateTemperaturePair pair,
9                             Text text,
10                            int numberOfPartitions) {
11        // 确保分区数非负
12        return Math.abs(pair.getYearMonth().hashCode() % numberOfPartitions);
13     }
14 }
```

Hadoop提供了一个插件体系结构，允许在框架中注入定制分区器代码。我们在驱动器类中完成这个工作（驱动器将把MapReduce作业提交给Hadoop），如下所示：

```
import org.apache.hadoop.mapreduce.Job;
...
Job job = ...;
...
job.setPartitionerClass(TemperaturePartitioner.class);
```

## 分组比较器

在示例1-3中，我们定义了比较器（DateTemperatureGroupingComparator类），它会控制哪些键要分组到一个Reducer.reduce()函数调用。

示例1-3：DateTemperatureGroupingComparator类

```
1 import org.apache.hadoop.io.WritableComparable;
2 import org.apache.hadoop.io.WritableComparator;
3
4 public class DateTemperatureGroupingComparator
5    extends WritableComparator {
6
7     public DateTemperatureGroupingComparator() {
8         super(DateTemperaturePair.class, true);
9     }
10
11    @Override
12    /**
13     * 这个比较器控制哪些键要
14     * 分组到一个reduce()方法调用
15     */
16    public int compare(WritableComparable wc1, WritableComparable wc2) {
17        DateTemperaturePair pair = (DateTemperaturePair) wc1;
18        DateTemperaturePair pair2 = (DateTemperaturePair) wc2;
19        return pair.getYearMonth().compareTo(pair2.getYearMonth());
20    }
21 }
```

Hadoop提供了一个插件体系结构，允许在框架中注入定制比较器。我们在驱动器类中完成这个工作（驱动器将把MapReduce作业提交给Hadoop），如下所示：

```
job.setGroupingComparatorClass(YearMonthGroupingComparator.class);
```

## 使用插件类的数据流

为了帮助理解map()和reduce()函数和定制插件类，图1-2展示了部分输入的数据流。

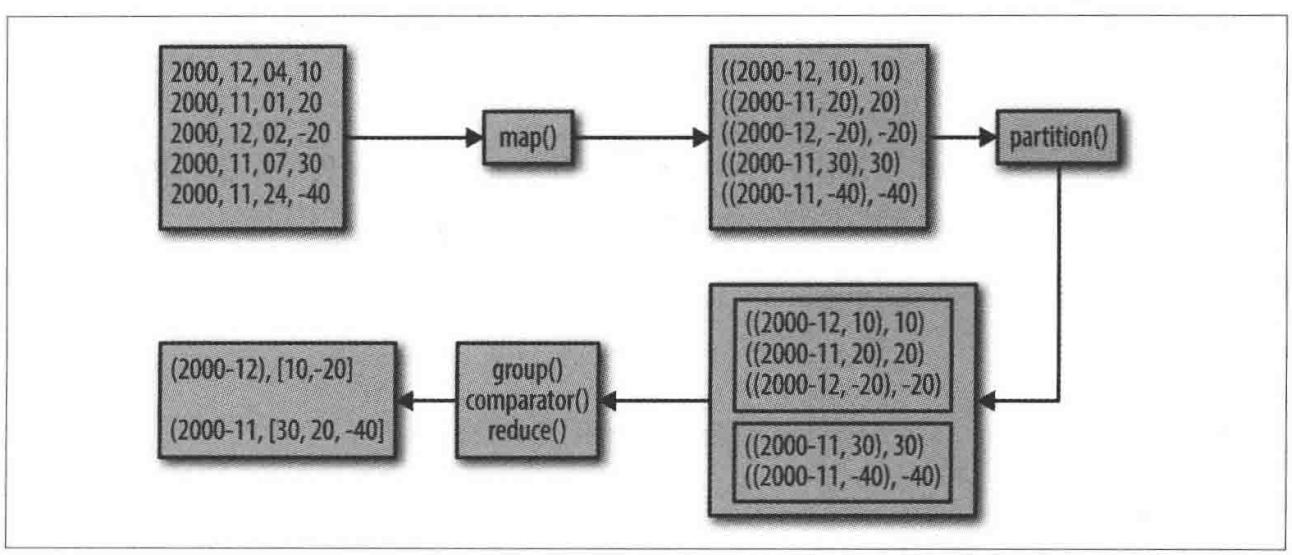



图1-2：二次排序数据流

映射器创建（*K*,*V*)对，这里K是组合键（year,month,temperature），*V*是temperature。

组合键的（year,month）部分是自然键。利用分区器插件类，可以把所有自然键发送给同一个归约器，分组比较器类则使得温度会按有序的顺序到达归约器。二次排序设计模式使用MapReduce的框架对归约器值排序，而不是先把它们收集在一起再在内存中完成排序。二次排序设计模式允许我们“横向扩容”，而不论有多少归约器值需要排序。

# MapReduce/Hadoop的二次排序解决方案

这一节使用Hadoop框架为二次排序问题提供一个完整的MapReduce实现。

## 输入

输入是一组文件，其中每个记录（行）的格式如下：

```
格式：
     <year><,><month><,><day><,><temperature>

示例：
2012, 01, 01, 35
2011, 12, 23, -4
```

## 期望输出

期望输出有以下格式：

```
格式：
      <year><->><month>: <temperature1><,><temperature2><,> ...
      其中temperature1 <= temperature2 <= ...

示例：
      2012-01: 5, 10, 35, 45, ...
      2001-11: 40, 46, 47, 48, ...
      2005-08: 38, 50, 52, 70, ...
```

## map()函数

map()函数对输入完成解析和词法分析，然后将值（temperature）注入到归约器键，如示例1-4所示。

示例1-4：二次排序：map()

```
1 /**
2  * @param key由Hadoop生成（在这里忽略）
3  * @param value有以下格式："YYYY,MM,DD,temperature"
4  */
5 map(key, value) {
6    String[]tokens = value.split(",");
7    // YYYY = tokens[0]
8    // MM = tokens[1]
```

```
    // DD = tokens[2]
    // temperature = tokens[3]
    String yearMonth = tokens[0] + tokens[1];
    String day = tokens[2];
    int temperature = Integer.parseInt(tokens[3]);
    // 准备归约器键
    DateTemperaturePair reducerKey = new DateTemperaturePair();
    reducerKey.setYearMonth(yearMonth);
    reducerKey.setDay(day);
    reducerKey.setTemperature(temperature); // 将值注入到键
    // 发送到归约器
    emit(reducerKey, temperature);
}
```

## reduce()函数

归约器的主函数将值连接在一起（已经利用二次排序设计模式对这些值完成了排序），然后作为输出发出。reduce()函数如示例1-5所示。

示例1-5：二次排序：reduce()

```
/**
 * @param key是一个DateTemperaturePair对象
 * @param value是一个温度列表
 */
reduce(key, value) {
   StringBuilder sortedTemperatureList = new StringBuilder();
   for (Integer temperature : value) {
      sortedTemperatureList.append(temperature);
      sortedTemperatureList.append(",");
   }
   emit(key, sortedTemperatureList);
}
```

## Hadoop实现类

我们使用表1-1所示的类来解决这个问题。

**表1-1：MapReduce/Hadoop解决方案中使用的类**

| 类名 | 类描述 |
|---|---|
| SecondarySortDriver | 驱动器类，定义输入/输出，并注册插件类 |
| SecondarySortMapper | 定义map()函数 |
| SecondarySortReducer | 定义reduce()函数 |
| DateTemperatureGroupingComparator | 定义如何对键分组 |
| DateTemperaturePair | 将日期和温度对定义为Java对象 |
| DateTemperaturePartitioner | 定义定制分区器 |

如何将值注入到键中？第一个比较器（DateTemperaturePair.compareTo()方法）会控制键的排序顺序，第二个比较器（DateTemperatureGroupingComparator.compare()方法）会控制哪些键要分组到一个reduce()方法调用。通过结合这两个比较器，建立作业时就好像为值定义了顺序一样。

SecondarySortDriver是驱动器类，它会向MapReduce/Hadoop框架注册定制插件类（DateTemperaturePartitioner和DateTemperatureGroupingComparator）。这个驱动器类如示例1-6所示。

示例1-6：SecondarySortDriver类

```
public class SecondarySortDriver extends Configured implements Tool {
    public int run(String[] args) throws Exception {
        Configuration conf = getConf();
        Job job = new Job(conf);
        job.setJarByClass(SecondarySortDriver.class);
        job.setJobName("SecondarySortDriver");

        Path inputPath = new Path(args[0]);
        Path outputPath = new Path(args[1]);
        FileInputFormat.setInputPaths(job, inputPath);
        FileOutputFormat.setOutputPath(job, outputPath);

        job.setOutputKeyClass(TemperaturePair.class);
        job.setOutputValueClass(NullWritable.class);

        job.setMapperClass(SecondarySortingTemperatureMapper.class);
        job.setReducerClass(SecondarySortingTemperatureReducer.class);
        job.setPartitionerClass(TemperaturePartitioner.class);
        job.setGroupingComparatorClass(YearMonthGroupingComparator.class);

        boolean status = job.waitForCompletion(true);
        theLogger.info("run(): status="+status);
        return status ? 0 : 1;
    }

    /**
     * 二次排序MapReduce程序的主驱动器。
     * 调用这个方法提交MapReduce作业。
     * @throws Exception，如果与作业跟踪器
     * 存在通信问题，会抛出异常。
     */
     public static void main(String[] args) throws Exception {
         // 确保有2个参数
         if (args.length != 2) {
             throw new IllegalArgumentException("Usage: SecondarySortDriver" +
                                                " <input-path> <output-path>");
         }

         //String inputPath = args[0];
         //String outputPath = args[1];
         int returnStatus = ToolRunner.run(new SecondarySortDriver(), args);
```

```
42          System.exit(returnStatus);
43      }
44
45  }
```

# Hadoop实现运行示例

## 输入

```
# cat sample_input.txt
2000,12,04, 10
2000,11,01,20
2000,12,02,-20
2000,11,07,30
2000,11,24,-40
2012,12,21,30
2012,12,22,-20
2012,12,23,60
2012,12,24,70
2012,12,25,10
2013,01,22,80
2013,01,23,90
2013,01,24,70
2013,01,20,-10
```

## HDFS输入

```
# hadoop fs -mkdir /secondary_sort
# hadoop fs -mkdir /secondary_sort/input
# hadoop fs -mkdir /secondary_sort/output
# hadoop fs -put sample_input.txt /secondary_sort/input/
# hadoop fs -ls /secondary_sort/input/
Found 1 items
-rw-r--r-- 1 ... 128 ... /secondary_sort/input/sample_input.txt
```

## 脚本

```
# cat run.sh
export JAVA_HOME=/usr/java/jdk7
export BOOK_HOME=/home/mp/data-algorithms-book
export APP_JAR=$BOOK_HOME/dist/data_algorithms_book.jar
INPUT=/secondary_sort/input
OUTPUT=/secondary_sort/output
$HADOOP_HOME/bin/hadoop fs -rmr $OUTPUT
PROG=org.dataalgorithms.chap01.mapreduce.SecondarySortDriver
$HADOOP_HOME/bin/hadoop jar $APP_JAR $PROG $INPUT $OUTPUT
```

## 运行示例日志

```
# ./run.sh
...
Deleted hdfs://localhost:9000/secondary_sort/output
```

```
13/02/27 19:39:54 INFO input.FileInputFormat: Total input paths to process : 1
...
13/02/27 19:39:54 INFO mapred.JobClient: Running job: job_201302271939_0001
13/02/27 19:39:55 INFO mapred.JobClient:  map 0% reduce 0%
13/02/27 19:40:10 INFO mapred.JobClient:  map 100% reduce 0%
13/02/27 19:40:22 INFO mapred.JobClient:  map 100% reduce 10%
...
13/02/27 19:41:10 INFO mapred.JobClient:  map 100% reduce 90%
13/02/27 19:41:16 INFO mapred.JobClient:  map 100% reduce 100%
13/02/27 19:41:21 INFO mapred.JobClient: Job complete: job_201302271939_0001
...
13/02/27 19:41:21 INFO mapred.JobClient: Map-Reduce Framework
...
13/02/27 19:41:21 INFO mapred.JobClient:   Reduce input records=14
13/02/27 19:41:21 INFO mapred.JobClient:   Reduce input groups=4
13/02/27 19:41:21 INFO mapred.JobClient:   Combine output records=0
13/02/27 19:41:21 INFO mapred.JobClient:   Reduce output records=4
13/02/27 19:41:21 INFO mapred.JobClient:   Map output records=14
13/02/27 19:41:21 INFO SecondarySortDriver: run(): status=true
13/02/27 19:41:21 INFO SecondarySortDriver: returnStatus=0
```

### 检查输出

```
# hadoop fs -cat /secondary_sort/output/p*
2013-01  90,80,70,-10
2000-12  10,-20
2000-11  30,20,-40
2012-12  70,60,30,10,-20
```

## 如何按升序或降序排序

通过使用DateTemperaturePair.compareTo()方法可以很容易地控制值的排序顺序（升序或降序），如下所示：

```
1 public int compareTo(DateTemperaturePair pair) {
2     int compareValue = this.yearMonth.compareTo(pair.getYearMonth());
3     if (compareValue == 0) {
4         compareValue = temperature.compareTo(pair.getTemperature());
5     }
6     //return compareValue; // 升序排序
7     return -1*compareValue; // 降序排序
8 }
```

# Spark的二次排序解决方案

要在Spark中解决二次排序问题，至少有两种方案：

*方案1*

将一个给定键的所有值读取并缓存到一个List数组（Array）数据结构中，然后对这

些值完成归约器中排序。如果每个归约器键的值集很小（可以完全放在内存中），这种解决方案就是适用的。

*方案2*

使用Spark框架对归约器值排序（这种做法不需要对传入归约器的值完成归约器中排序）。这种方法“会为自然键增加部分或整个值来创建一个组合键以实现排序目标”。这种方法是可伸缩的（因为不会受商用服务器内存的限制）。

## 时间序列作为输入

为了展示二次排序，下面使用时间序列数据作为输入：

```
name time value
  x    2    9
  y    2    5
  x    1    3
  y    1    7
  y    3    1
  x    3    6
  z    1    4
  z    2    8
  z    3    7
  z    4    0
  p    2    6
  p    4    7
  p    1    9
  p    6    0
  p    7    3
```

## 期望输出

期望输出如下所示。需要说明，归约器值按名分组，并按时间排序：

```
name  t1   t2   t3   t4   t5 ...
x => [3,   9,   6]
y => [7,   5,   1]
z => [4,   8,   7,   0]
p => [9,   6,   7,   0,   3]
```

## 方案1:内存中实现二次排序

由于Spark有一个非常强大的高层API，这里将用一个Java类提供整个解决方案。Spark API建立在弹性分布式数据集（resilient distributed data set，RDD）基本抽象概念基础上。要充分利用Spark的API，首先必须理解RDD。`RDD<T>`（也就是一个类型为T的RDD）对象表示一个不可变的元素分区集合（元素类型为T），这些元素可以并行处理。`RDD<T>`类包含所有RDD可用的基本MapReduce操作，如`map()`、`filter()`和

persist()，JavaPairRDD<K,V>类则包含mapToPair()、flatMapToPair()和groupByKey()等MapReduce操作。除此以外，Spark的PairRDDFunctions包含只适用于键-值对RDD的操作，如reduce()、groupByKey()和join()（关于RDD的详细内容，可以参考Spark API（*http://bit.ly/spark_rdd*）和本书附录B）。因此，JavaRDD<T>是一个T类型对象列表，JavaPairRDD<K,V>是一个Tuple2<K,V>类型对象列表（其中每个元组分别表示一个键-值对）。

下面给出基于Spark的算法。尽管这里有10个步骤，不过其中大部分步骤都很简单，有些步骤只是用来调试：

1. 导入所需的Java/Spark类。MapReduce的主Java类在org.apache.spark.api.java包中。这个包包括以下类和接口：
   - JavaRDDLike（接口）
   - JavaDoubleRDD
   - JavaPairRDD
   - JavaRDD
   - JavaSparkContext
   - StorageLevels
2. 将输入数据作为参数传入并验证。
3. 创建一个JavaSparkContext对象，连接到Spark master，这个对象将用来创建新的RDD。
4. 使用（第3步中创建的）上下文对象为输入文件创建一个RDD，得到的RDD将是一个JavaRDD<String>。这个RDD中的各个元素分别是一个时间序列数据记录：<name><,><time><,><value>。
5. 接下来，从一个JavaRDD<String>创建键-值对，其中键是name，值是一个（time, value）对。得到的RDD将是一个JavaPairRDD<String, Tuple2<Integer, Integer>>。
6. 为了验证第5步，从JavaPairRDD<>收集所有值，并打印。
7. 按键（name）对JavaPairRDD<>元素分组。为实现这一步，我们要使用groupByKey()方法。

   结果将是以下RDD：

   ```
   JavaPairRDD<String, Iterable<Tuple2<Integer, Integer>>>
   ```

   注意结果列表（Iterable<Tuple2<Integer, Integer>>）并不是有序的。一般地，出于性能方面的原因，更优先使用Spark的reduceByKey()而不是groupByKey()，不

过这里除了groupByKey()我们没有其他选择（因为reduceByKey()不允许对给定键的值完成原地排序）。

8. 为了验证第7步，从JavaPairRDD<String, Iterable<Tuple2<Integer, Integer>>>收集所有值，并打印。

9. 对归约器值排序来得到最终输出。我们通过编写一个mapValues()方法来完成这一步。这里只对值排序（键保持不变）。

10. 为了验证最终结果，从已排序的JavaPairRDD<>收集所有值，并打印。

方案1可以用一个驱动器类来实现：SecondarySorting （参见示例1-7）。所有步骤（1~10）都放在这个类定义中，我们会在后面的小节中分别说明。一般地，Spark应用包括一个驱动器程序，它会运行用户的main()函数，并在一个集群上执行各个并行操作。并行操作要通过大量使用RDD来实现。关于RDD的更多详细信息，参见附录B。

示例1-7：SecondarySort类总结构

```
// 步骤1: 导入必要的Java/Spark类
public class SecondarySort {
  public static void main(String[] args) throws Exception {
    // 步骤2: 读取输入参数并验证
    // 步骤3: 通过创建一个JavaSparkContext对象 (ctx)
    // 连接到Spark master
    // 步骤4: 使用ctx创建JavaRDD<String>
    // 步骤5: 由JavaRDD<String>创建键-值对，其中
    // 键是{name} ，值是 (time, value)对
    // 步骤6: 验证步骤5，收集JavaPairRDD<>的所有值
    // 并打印
    // 步骤7: 按键 ({name})对JavaPairRDD<>元素分组
    // 步骤8: 验证步骤7，收集JavaPairRDD<>的所有值
    // 并打印
    // 步骤9: 对归约器值排序；将得到最终输出
    // 步骤10: 验证步骤9，收集JavaPairRDD<>的所有值
    // 并打印

    // 完成
    ctx.close();
    System.exit(0);
  }
}
```

## 步骤1：导入必要的类

如示例1-8所示，Spark的主Java API包是org.apache.spark.api.java，其中包括JavaRDD、JavaPairRDD和JavaSparkContext类。JavaSparkContext是一个工厂类，用来创建新的RDD（如JavaRDD和JavaPairRDD对象）。

示例1-8：步骤1：导入必要的类

```
// 步骤1：导入必要的Java/Spark类
import scala.Tuple2;
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.api.java.JavaPairRDD;
import org.apache.spark.api.java.JavaSparkContext;
import org.apache.spark.api.java.function.Function;
import org.apache.spark.api.java.function.Function2;
import org.apache.spark.api.java.function.PairFunction;

import java.util.List;
import java.util.ArrayList;
import java.util.Map;
import java.util.Collections;
import java.util.Comparator;
```

## 步骤2：读取输入参数

这一步（如示例1-9所示）要读取HDFS输入文件（Spark可以从HDFS和其他持久存储库（如Linux文件系统）读取数据），例如*/dir1/dir2/myfile.txt*。

示例1-9：步骤2：读取输入参数

```
    // 步骤2：读取输入参数并验证
    if (args.length < 1) {
       System.err.println("Usage: SecondarySort <file>");
       System.exit(1);
    }
    String inputPath = args[0];
    System.out.println("args[0]: <file>="+args[0]);
```

## 步骤3：连接Spark master

要处理RDD，首先需要创建一个JavaSparkContext对象（如示例1-10所示），这是一个创建JavaRDD和JavaPairRDD对象的工厂。也可以在JavaSparkContext的类构造函数中注入一个SparkConf对象来创建JavaSparkContext对象。从XML文件读取集群配置信息时这种方法很有用。基本说来，JavaSparkContext对象有以下职责：

- 初始化应用驱动器。
- 向集群管理器注册这个应用驱动器（如果使用Spark集群，集群管理器就是Spark master；如果使用YARN，则是YARN的资源管理器）。
- 获得一个执行器列表来执行应用驱动器。

示例1-10：步骤3：连接Spark master

```
    // 步骤3：通过创建一个JavaSparkContext连接到Spark master对象
    final JavaSparkContext ctx = new JavaSparkContext();
```

### 步骤4：使用JavaSparkContext创建JavaRDD

这一步（如示例1-11所示）读取一个HDFS文件，并创建一个JavaRDD<String>（表示一个记录集合，其中每个记录分别是一个String对象）。根据定义，Spark的RDD是不可变的（也就是说，它们不能修改或改变）。需要说明，Spark的RDD是并行执行的基本抽象。另外还要指出，可以使用textFile()读取HDFS或非HDFS文件。

示例1-11：步骤4：创建JavaRDD

```
// 步骤4：使用ctx创建JavaRDD<String>
// 输入记录格式：<name><,><time><,><value>
JavaRDD<String> lines = ctx.textFile(inputPath, 1);
```

### 步骤5：从JavaRDD创建键-值对

这一步（如示例1-12所示）会实现一个映射器。每个记录（来自JavaRDD<String>，包括<name><,><time><,><value>）分别转换为一个键-值对，其中键是name，值是一个Tuple2(time, value)。

示例1-12：步骤5：从JavaRDD创建键-值对

```
// 步骤5：由JavaRDD<String>创建键-值对，其中
// 键是{name}，值是（time, value)对。
// 得到的RDD将是一个JavaPairRDD<String, Tuple2<Integer, Integer>>。
// 将各个记录转换为Tuple2(name, time, value)。
// PairFunction<T, K, V>
//   T => Tuple2(K, V) 其中T是输入（String),
//   K=String
//   V=Tuple2<Integer, Integer>
JavaPairRDD<String, Tuple2<Integer, Integer>> pairs =
      lines.mapToPair(new PairFunction<
                                       String,                   // T
                                       String,                   // K
                                       Tuple2<Integer, Integer> // V
                                      >() {
 public Tuple2<String, Tuple2<Integer, Integer>> call(String s) {
   String[] tokens = s.split(","); // x,2,5
   System.out.println(tokens[0] + "," + tokens[1] + "," + tokens[2]);
   Integer time = new Integer(tokens[1]);
   Integer value = new Integer(tokens[2]);
   Tuple2<Integer, Integer> timevalue =
   new Tuple2<Integer, Integer>(time, value);
   return new Tuple2<String, Tuple2<Integer, Integer>>(tokens[0], timevalue);
 }
});
```

### 步骤6：验证步骤5

要在Spark中调试和验证已完成的步骤（如示例1-13所示），可以使用JavaRDD.collect()和JavaPairRDD.collect()。需要指出，collect()只是用于调试和展示（而

在生产集群中要避免为了调试而使用collect()，这会影响性能）。另外，可以使用JavaRDD.saveAsTextFile()完成调试并创建所需的输出。

示例1-13：步骤6：验证步骤5

```
// 步骤6：验证步骤5-收集JavaPairRDD<>的所有值
// 并打印
List<Tuple2<String, Tuple2<Integer, Integer>>> output = pairs.collect();
for (Tuple2 t : output) {
   Tuple2<Integer, Integer> timevalue = (Tuple2<Integer, Integer>) t._2;
   System.out.println(t._1 + "," + timevalue._1 + "," + timevalue._1);
}
```

## 步骤7：按键（name）对JavaPairRDD元素分组

我们将使用groupByKey()实现归约器操作。如示例1-14所示，与MapReduce/Hadoop相比，使用Spark实现归约器要容易得多。需要说明，在Spark中，一般来讲reduceByKey()比groupByKey()更高效。不过，这里不能使用reduceByKey()。

示例1-14：步骤7：JavaPairRDD元素分组

```
// 步骤7：按键（{name}）对JavaPairRDD<>元素分组
JavaPairRDD<String, Iterable<Tuple2<Integer, Integer>>> groups =
      pairs.groupByKey();
```

## 步骤8：验证步骤7

这一步（如示例1-15所示）使用collect()函数验证上一步，它会从groups RDD获得所有值。

示例1-15：步骤8：验证步骤7

```
// 步骤8：验证步骤7，收集JavaPairRDD<>的所有值
// 并打印
System.out.println("===DEBUG1===");
List<Tuple2<String, Iterable<Tuple2<Integer, Integer>>>> output2 =
      groups.collect();
for (Tuple2<String, Iterable<Tuple2<Integer, Integer>>> t : output2) {
   Iterable<Tuple2<Integer, Integer>> list = t._2;
   System.out.println(t._1);
   for (Tuple2<Integer, Integer> t2 : list) {
      System.out.println(t2._1 + "," + t2._2);
   }
   System.out.println("=====");
 }
```

下面给出这一步的输出。可以看到，归约器值还没有排序：

```
y
2,5
1,7
3,1
```

```
=====
x
2,9
1,3
3,6
=====
z
1,4
2,8
3,7
4,0
=====
p
2,6
4,7
6,0
7,3
1,9
=====
```

## 步骤9：在内存中对归约器值排序

这一步（如示例1-16所示）使用另一个很强大的Spark方法mapValues()对归约器生成的值排序。利用mapValues()方法，我们可以将（$K$, $V_1$)转换为（$K$, $V_2$)，这里$V_2$是排序后的$V_1$。Spark的RDD是不可变的，无法以任何方式修改/更新，这一点非常重要。例如，在这一步中，为了对值进行排序，首先必须把它们复制到另一个列表。RDD本身以及其中包含的元素都具有这种不可变性。

示例1-16：步骤9：在内存中对归约器值排序

```
// 步骤9：将归约器值排序；将得到最终输出。
// 方案1：可行
// mapValues[U](f: (V) => U): JavaPairRDD[K, U]
// 通过一个映射函数将各个值传入键-值对RDD，
// 而不改变键；
// 还会保留原来的RDD分区。
JavaPairRDD<String, Iterable<Tuple2<Integer, Integer>>> sorted =
       groups.mapValues(
         new Function<Iterable<Tuple2<Integer, Integer>>, // 输入
                      Iterable<Tuple2<Integer, Integer>>  // 输出
                     >() {
    public Iterable<Tuple2<Integer, Integer>> call(Iterable<Tuple2<Integer,
                                                              Integer>> s) {
      List<Tuple2<Integer, Integer>> newList = new ArrayList<Tuple2<Integer,
                                                              Integer>>(s);
      Collections.sort(newList, new TupleComparator());
      return newList;
    }
});
```

### 步骤10：输出最终结果

collect()方法将RDD的所有元素收集到一个java.util.List对象中。然后迭代处理这个List，得到所有最终元素（参见示例1-17）。

示例1-17：步骤10：输出最终结果

```
// 步骤10：验证步骤9，收集JavaPairRDD<>的所有值
// 并打印
System.out.println("===DEBUG2=");
List<Tuple2<String, Iterable<Tuple2<Integer, Integer>>>> output3 =
      sorted.collect();
for (Tuple2<String, Iterable<Tuple2<Integer, Integer>>> t : output3) {
   Iterable<Tuple2<Integer, Integer>> list = t._2;
   System.out.println(t._1);
   for (Tuple2<Integer, Integer> t2 : list) {
     System.out.println(t2._1 + "," + t2._2);
   }
   System.out.println("=====");
}
```

## Spark运行示例

使用Spark/Hadoop时，可以在3种不同模式下运行Spark应用[注1]：

*独立模式*

这是默认设置。在一个主节点上启动Spark master，并在每个从节点上分别启动一个“worker”，然后将Spark应用提交给Spark master。

*YARN客户端模式*

在这种模式下，不用启动Spark主节点或工作节点。实际上，需要把Spark应用提交给YARN，它会在Spark客户端进程中运行Spark驱动器来提交应用。

*YARN集群模式*

采用这种模式，不需要启动Spark主节点或工作节点。实际上，需要把Spark应用提交给YARN，它会在YARN中的ApplicationMaster中运行Spark驱动器。

接下来，我们来介绍如何在独立模式和YARN集群模式下提交二次排序应用。

### 独立模式下运行Spark

下面各小节分别给出Spark独立模式下二次排序应用运行示例的输入、脚本和日志输出。

### HDFS 输入

```
# hadoop fs -cat /mp/timeseries.txt
x,2,9
```

注1：有关的详细信息可以参考Spark文档（*http://bit.ly/spark_on_yarn*）。

```
y,2,5
x,1,3
y,1,7
y,3,1
x,3,6
z,1,4
z,2,8
z,3,7
z,4,0
p,2,6
p,4,7
p,1,9
p,6,0
p,7,3
```

## 脚本

```
# cat run_secondarysorting.sh
#!/bin/bash
export JAVA_HOME=/usr/java/jdk7
export SPARK_HOME=/home/hadoop/spark-1.1.0
export SPARK_MASTER=spark://myserver100:7077
BOOK_HOME=/home/mp/data-algorithms-book
APP_JAR=$BOOK_HOME/dist/data_algorithms_book.jar
INPUT=/home/hadoop/testspark/timeseries.txt
# 在Spark独立集群上运行
prog=org.dataalgorithms.chap01.spark.SparkSecondarySort
$SPARK_HOME/bin/spark-submit \
  --class $prog \
  --master $SPARK_MASTER \
  --executor-memory 2G \
  --total-executor-cores 20 \
  $APP_JAR \
  $INPUT
```

## 运行日志

```
# ./run_secondarysorting.sh
args[0]: <file>=/mp/timeseries.txt
...
=== DEBUG STEP 5 ===
...
x,2,2
y,2,2
x,1,1
y,1,1
y,3,3
x,3,3
z,1,1
z,2,2
z,3,3
z,4,4
p,2,2
p,4,4
```

```
p,1,1
p,6,6
p,7,7
=== DEBUG STEP 7 ===
14/06/04 08:42:54 INFO spark.SparkContext: Starting job: collect
  at SecondarySort.java:96
14/06/04 08:42:54 INFO scheduler.DAGScheduler: Registering RDD 2
  (mapToPair at SecondarySort.java:75)
...
14/06/04 08:42:55 INFO scheduler.DAGScheduler: Stage 1
  (collect at SecondarySort.java:96) finished in 0.273 s
14/06/04 08:42:55 INFO spark.SparkContext: Job finished:
  collect at SecondarySort.java:96, took 1.587001929 s
z
1,4
2,8
3,7
4,0
=====
p
2,6
4,7
1,9
6,0
7,3
=====
x
2,9
1,3
3,6
=====
y
2,5
1,7
3,1
=====
=== DEBUG STEP 9 ===
14/06/04 08:42:55 INFO spark.SparkContext: Starting job: collect
  at SecondarySort.java:158
...
14/06/04 08:42:55 INFO scheduler.TaskSchedulerImpl: Removed TaskSet 3.0,
  whose tasks have all completed, from pool
14/06/04 08:42:55 INFO spark.SparkContext: Job finished: collect at
  SecondarySort.java:158, took 0.074271723 s
z
1,4
2,8
3,7
4,0
=====
p
1,9
2,6
4,7
6,0
```

```
7,3
=====
x
1,3
2,9
3,6
=====
y
1,7
2,5
3,1
=====
```

一般地，会把最终结果保存到HDFS。创建“有序”RDD之后，可以增加下面这行代码来保存最终结果：

```
sorted.saveAsTextFile("/mp/output");
```

可能会看到如下的输出：

```
# hadoop fs -ls /mp/output/
Found 2 items
-rw-r--r-- 3 hadoop root,hadoop   0 2014-06-04 10:49 /mp/output/_SUCCESS
-rw-r--r-- 3 hadoop root,hadoop 125 2014-06-04 10:49 /mp/output/part-00000

# hadoop fs -cat /mp/output/part-00000
(z,[(1,4), (2,8), (3,7), (4,0)])
(p,[(1,9), (2,6), (4,7), (6,0), (7,3)])
(x,[(1,3), (2,9), (3,6)])
(y,[(1,7), (2,5), (3,1)])
```

**YARN集群模式下运行Spark。**YARN集群模式下提交这个Spark应用的脚本如下：

```
# cat run_secondarysorting_yarn.sh
#!/bin/bash
export JAVA_HOME=/usr/java/jdk7
export HADOOP_HOME=/usr/local/hadoop-2.5.0
export HADOOP_CONF_DIR=$HADOOP_HOME/etc/hadoop
export YARN_CONF_DIR=$HADOOP_HOME/etc/hadoop
export SPARK_HOME=/home/hadoop/spark-1.1.0
BOOK_HOME=/home/mp/data-algorithms-book
APP_JAR=$BOOK_HOME/dist/data_algorithms_book.jar
INPUT=/mp/timeseries.txt
prog=org.dataalgorithms.chap01.spark.SparkSecondarySort
$SPARK_HOME/bin/spark-submit \
  --class $prog \
  --master yarn-cluster \
  --executor-memory 2G \
  --num-executors 10 \
  $APP_JAR \
  $INPUT
```

## 方案2：使用Spark框架实现二次排序

在方案1中，我们在内存中（使用Java的`Collections.sort()`方法）对归约器值排序，如果归约器值无法完全放在商用服务器的内存中，这种方案就不具有可伸缩性。接下来我们会为MapReduce/Hadoop框架实现方案2。这在当前的Spark（Spark-1.1.0）框架下还无法实现，因为目前Spark的洗牌过程基于散列完成，这与MapReduce基于排序的洗牌过程有所不同。所以，必须使用一个RDD操作符显式地实现排序。如果有一个按自然键（`name`）分区的分区器，能保留RDD的顺序，这将是一个可行的解决方案。例如，如果按（`name, time`）排序，可以得到：

```
(p,1),(1,9)
(p,4),(4,7)
(p,6),(6,0)
(p,7),(7,3)

(x,1),(1,3)
(x,2),(2,9)
(x,3),(3,6)

(y,1),(1,7)
(y,2),(2,5)
(y,3),(3,1)

(z,1),(1,4)
(z,2),(2,8)
(z,3),(3,7)
(z,4),(4,0)
```

这里有一个分区器（表示为一个抽象类`org.apache.spark.Partitioner`），不过它没有保留原RDD元素的顺序。因此，使用当前版本的Spark (1.1.0)无法实现方案2。

## 有关二次排序的更多资源

要在Spark中支持二次排序，可以扩展JavaPairRDD类，并增加一些额外的方法，如`groupByKeyAndSortValues()`。可以参考以下资源了解有关这个主题的更多研究工作：

- 支持除了键以外的值排序（即二次排序）（*http://bit.ly/secondary_sort_ticket*）。
- *https://github.com/tresata/spark-sorted*。

第2章会给出使用MapReduce和Spark框架的二次排序设计模式的具体实现。

# 第2章

# 二次排序：详细示例

MapReduce框架会按键对归约器的输入排序，不过归约器值的顺序是任意的。这意味着，对于key = K，如果所有映射器生成以下键-值对：

$$(K, V_1), (K, V_2), \ldots, (K, V_n)$$

那么所有这些值$\{V_1, V_2, \ldots, V_n\}$会由同一个归约器处理（对应key = K），不过，各个$V_i$之间并不是有序的（不能保证升序或降序）。在第1章已经了解到，二次排序是一个设计模式，可以用来对值应用或提供一个顺序（如“升序”或“降序”）。如何做到呢？假设我们希望对归约器值应用某种顺序：

$$S_1 \le S_2 \le \ldots \le S_n$$

或：

$$S_1 \ge S_2 \ge \ldots \ge S_n$$

这里$S_i \in \{V_1, V_2, \ldots, V_n\}$，$i = \{1, 2, \ldots, n\}$。注意，每个$V_i$可能是一个简单数据类型，如String或Integer，也可以是一个元组（不只是单个值，也就是说，是一个组合对象）。

对归约器值排序有两种方法：

*方案1*

将归约器值缓存在内存中，然后排序。（对于每一个归约器）如果归约器值数量很少，完全可以放在内存中，这个解决方案就是可行的。不过，如果归约器值数目很大，可能无法放在内存中（这就不是一个可行的解决方案了）。这个方案的实现很简单，我们已经在第1章给出了这种方案的具体实现，这一章不再讨论。

*方案2*

使用MapReduce框架的二次排序设计模式，归约器值到达时就是有序的（也就是说，不再需要在内存中对值进行排序）。这种技术使用了MapReduce框架的洗牌和排序技术来完成归约器值的排序。这种解决方案比方案1更可取，因为采用这种方案时，不再依赖内存来完成排序（重申一次，如果有太多的值，方案1可能就不是一个可行的选择了）。这一章接下来会重点介绍方案2。

我们将使用Hadoop实现方案2，这里会用到：

- 老版本的Hadoop API（使用`org.apache.hadoop.mapred.JobConf`和`org.apache.hadoop.mapred.*`）。这里我特意使用了这个API，因为考虑到你有可能还没有移植到新版本的Hadoop API。
- 新版本的Hadoop API（使用`org.apache.hadoop.mapreduce.Job`和`org.apache.hadoop.mapreduce.lib.*`）。

# 二次排序技术

假设对应key = K有以下值：

$(K, V_1), (K, V_2), \ldots, (K, V_n)$

另外假设每个$V_i$是包含m个属性的一个元组，如下所示：

$(a_{i1}, a_{i2}, \ldots, a_{im})$

在这里，我们希望按$a_{i1}$对归约器的元组值进行排序。我们将用r表示（$a_{i2}$, ..., $a_{im}$）（其余的属性）。因此，可以把归约器值表示为：

$(K, (a_1, r_1)), (K, (a_2, r_2)), \ldots, (K, (a_n, r_n))$

要按$a_i$对归约器值排序，需要创建一个组合键：（K, $a_i$）。新映射器将发出对应key = K的键-值对，如表2-1所示。

**表2-1：映射器发出的键-值对**

| 键 | 值 |
|---|---|
| $(K, a_1)$ | $(a_1, r_1)$ |
| $(K, a_2)$ | $(a_2, r_2)$ |
| ... | ... |
| $(K, a_n)$ | $(a_n, r_n)$ |

所以，组合键为（K, $a_i$），自然键为K。通过定义组合键（即为自然键增加属性$a_i$），我

们可以使用MapReduce框架对归约器值排序，不过如果希望对键分区，则要按自然键（K）来完成分区。组合键和自然键如图2-1所示。

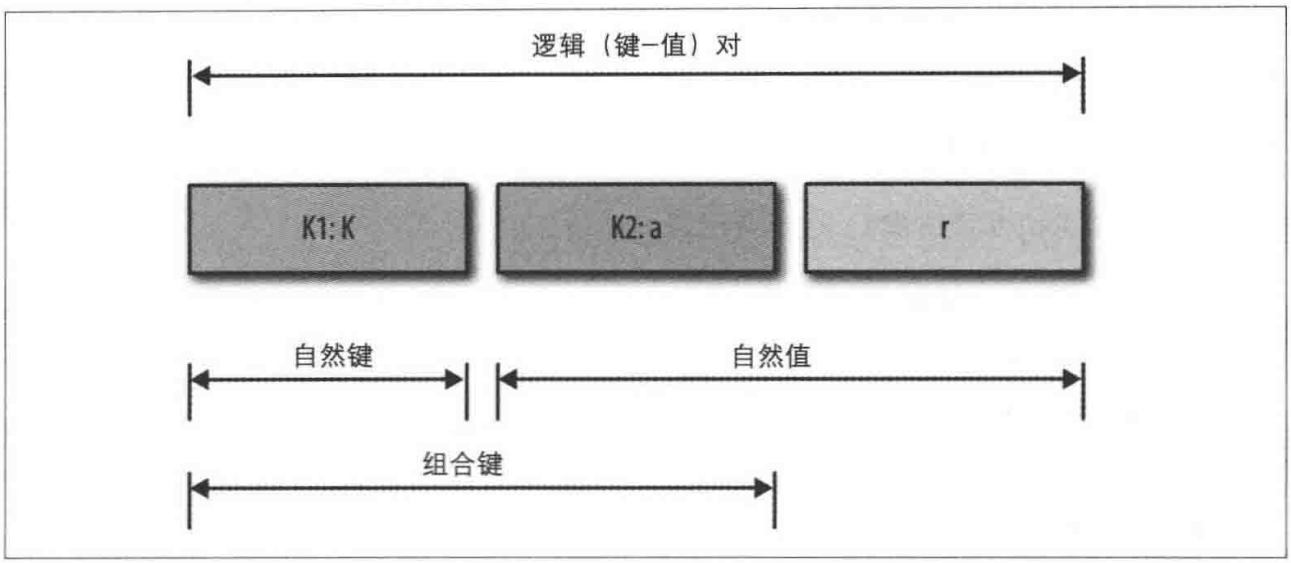



图2-1：二次排序键

必须告诉MapReduce框架如何使用组合键（包括两个字段：K和$a_i$）对键排序。为此，我们要定义一个排序插件类CompositeKeyComparator，将用这个插件类对组合键排序。示例2-1展示了如何在MapReduce框架中插入这个比较器类。

示例2-1：插入比较器类

```
1 import org.apache.hadoop.mapred.JobConf;
2 ...
3 JobConf conf = new JobConf(getConf(), <your-mapreduce-driver-class>.class);
4 ...
5 // map()创建键－值对
6 // (CompositeKey, NaturalValue)
7 conf.setMapOutputKeyClass(CompositeKey.class);
8 conf.setMapOutputValueClass(NaturalValue.class);
9 ...
10 // 比较器插件类:
11 // CompositeKey对象如何排序
12 conf.setOutputKeyComparatorClass(CompositeKeyComparator.class);
```

CompositeKeyComparator类告诉MapReduce框架如何对组合键排序。示例2-2中的实现会完成两个WritableComparable对象（表示CompositeKey对象）的比较。

示例2-2：比较器类：CompositeKeyComparator

```
1 import org.apache.hadoop.io.WritableComparable;
2 import org.apache.hadoop.io.WritableComparator;
3
4 public class CompositeKeyComparator extends WritableComparator {
5
6    protected CompositeKeyComparator() {
```

```
7        super(CompositeKey.class, true);
8     }
9
10    @Override
11    public int compare(WritableComparable k1, WritableComparable k2) {
12       CompositeKey ck1 = (CompositeKey) k1;
13       CompositeKey ck2 = (CompositeKey) k2;
14
15       // 比较ck1和ck2，并返回
16       // 0, 如果ck1和ck2相等
17       // 1, 如果ck1 > ck2
18       // -1,如果ck1 < ck2
19
20       // 实现细节在后面的小节中给出
21    }
22 }
```

下一个要插入的类是一个“自然键分区器”（我们把它命名为NaturalKeyPartitioner），这个类要实现Partitioner[注1]接口。示例2-3展示了如何将这个类插入到MapReduce框架。

示例2-3：插入NaturalKeyPartitioner

```
1 import org.apache.hadoop.mapred.JobConf;
2 ...
3 JobConf conf = new JobConf(getConf(), <your-mapreduce-driver-class>.class);
4 ...
5 conf.setPartitionerClass(NaturalKeyPartitioner.class);
```

接下来定义NaturalKeyPartitioner类，如示例2-4所示。

示例2-4：定义NaturalKeyPartitioner类

```
 1 import org.apache.hadoop.mapred.JobConf;
 2 import org.apache.hadoop.mapred.Partitioner;
 3
 4 /**
 5  * 映射阶段的数据发送到洗牌阶段之前，
 6  * NaturalKeyPartitioner对输出分区。
 7  *
 8  * getPartition()对映射器生成的数据分区。
 9  * 这个函数将按自然键对数据分区。
10  *
11  */
12 public class NaturalKeyPartitioner implements
13     Partitioner<CompositeKey, NaturalValue> {
14
15     @Override
16     public int getPartition(CompositeKey key,
17                             NaturalValue value,
```

注1：org.apache.hadoop.mapred.Partitioner。

```
18                              int numberOfPartitions) {
19          return <number-based-on-composite-key> % numberOfPartitions;
20      }
21
22      @Override
23      public void configure(JobConf arg) {
24      }
25 }
```

最后要插入的是NaturalKeyGroupingComparator，这个类用来比较两个自然键。示例2-5展示了如何将这个类插入到MapReduce框架。

示例2-5：插入NaturalKeyGroupingComparator

```
1 import org.apache.hadoop.mapred.JobConf;
2 ...
3 JobConf conf = new JobConf(getConf(), <your-mapreduce-driver-class>.class);
4 ...
5 conf.setOutputValueGroupingComparator(NaturalKeyGroupingComparator.class);
```

接下来，如示例2-6所示，我们如下定义NaturalKeyGroupingComparator类。

示例2-6：定义NaturalKeyGroupingComparator类

```
 1 import org.apache.hadoop.io.WritableComparable;
 2 import org.apache.hadoop.io.WritableComparator;
 3
 4 /**
 5  *
 6  * NaturalKeyGroupingComparator
 7  *
 8  * 这个类将在Hadoop的洗牌阶段
 9  * 根据键的第一部分（即自然键）对组合键分组。
10  */
11 public class NaturalKeyGroupingComparator extends WritableComparator {
12
13    protected NaturalKeyGroupingComparator() {
14        super(NaturalKey.class, true);
15    }
16
17    @Override
18    public int compare(WritableComparable o1, WritableComparable o2) {
19        NaturalKey nk1 = (NaturalKey) o1;
20        NaturalKey nk2 = (NaturalKey) o2;
21        return nk1.getNaturalKey().compareTo(nk2.getNaturalKey());
22    }
23 }
```

# 二次排序的完整示例

考虑下面的数据：

```
Stock-Symbol Date Closed-Price
```

另外假设希望对每个股票代码生成以下输出数据：

$$\text{Stock-Symbo}_1\text{: (Date}_1\text{, Price}_1\text{)(Date}_2\text{, Price}_2\text{)...(Date}_n\text{, Price}_n\text{)}$$

在这里：

$$\text{Date}_1 \leqslant \text{Date}_2 \leqslant ... \leqslant \text{Date}_n$$

我们希望归约器值按收盘价日期排序。可以使用二次排序完成这个任务。

## 输入格式

假设输入数据采用CSV格式：

```
Stock-Symbol,Date,Closed-Price
```

例如：

```
ILMN,2013-12-05,97.65
GOOG,2013-12-09,1078.14
IBM,2013-12-09,177.46
ILMN,2013-12-09,101.33
ILMN,2013-12-06,99.25
GOOG,2013-12-06,1069.87
IBM,2013-12-06,177.67
GOOG,2013-12-05,1057.34
```

## 输出格式

希望输出按收盘价日期排序，所以对于上面的示例输入，相应的输出如下：

```
ILMN: (2013-12-05,97.65)(2013-12-06,99.25)(2013-12-09,101.33)
GOOG: (2013-12-05,1057.34)(2013-12-06,1069.87)(2013-12-09,1078.14)
IBM: (2013-12-06,177.67)(2013-12-09,177.46)
```

## 组合键

自然键是股票代码，组合键是（Stock-Symbol,Date ）对。这里Date字段必须是组合键的一部分，因为我们希望归约器值按Date排序。自然键和组合键如图2-2所示。

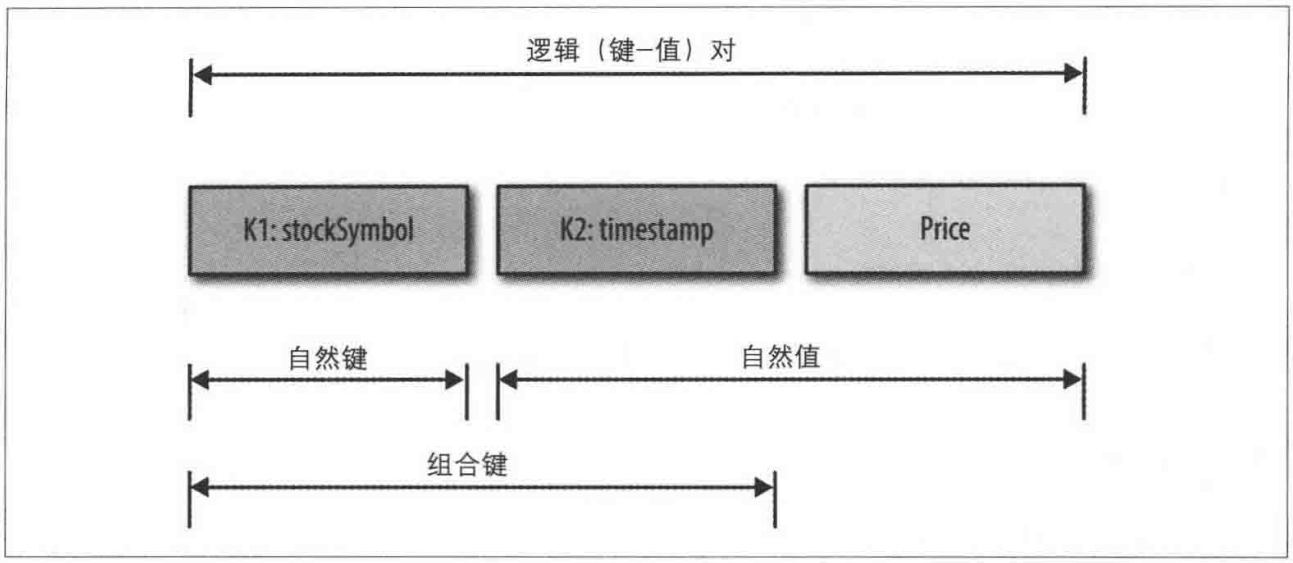



图2-2：二次排序：组合键和自然键

可以将组合键类定义为CompositeKey，与它关联的比较器类定义为CompositeKeyComparator（这个类告诉MapReduce如何对CompositeKey对象排序）。

## 组合键定义

在示例2-7中，组合键定义为CompositeKey类，它实现了WritableComparable<CompositeKey>接口[注2]。

示例2-7：定义组合键

```
import java.io.DataInput;
import java.io.DataOutput;
import java.io.IOException;
import org.apache.hadoop.io.WritableComparable;
import org.apache.hadoop.io.WritableComparator;

/**
 *
 * CompositeKey: 表示一个（String stockSymbol, long timestamp)对。
 * 需要说明，timestamp表示Date。
 *
 * 在stockSymbol字段上完成一次分组，
 * 将相同类型的所有数据分为一组，然后洗牌阶段的二次排序
 * 使用timestamp long分量对数据点排序，
 * 使得它们到达归约器时已经分区而且是有序的（按日期有序）。
 *
 */
public class CompositeKey implements WritableComparable<CompositeKey> {
    // 自然键是（stockSymbol)
```

注2：WritableComparable可以相互比较，通常利用Comparator来完成比较。Hadoop/MapReduce框架中任何作为键的类型都应当实现这个接口。

```
20      // 组合键是一个（stockSymbol, timestamp)对
21      private String stockSymbol; // 股票代码
22      private long timestamp;     // 日期
23
24      public CompositeKey(String stockSymbol, long timestamp) {
25          set(stockSymbol, timestamp);
26      }
27
28      public CompositeKey() {
29      }
30
31      public void set(String stockSymbol, long timestamp) {
32         this.stockSymbol = stockSymbol;
33         this.timestamp = timestamp;
34      }
35
36      public String getStockSymbol() {
37         return this.stockSymbol;
38      }
39
40      public long getTimestamp() {
41         return this.timestamp;
42      }
43
44      @Override
45      public void readFields(DataInput in) throws IOException {
46          this.stockSymbol = in.readUTF();
47          this.timestamp = in.readLong();
48      }
49
50      @Override
51      public void write(DataOutput out) throws IOException {
52          out.writeUTF(this.stockSymbol);
53          out.writeLong(this.timestamp);
54      }
55
56      @Override
57      public int compareTo(CompositeKey other) {
58          if (this.stockSymbol.compareTo(other.stockSymbol) != 0) {
59              return this.stockSymbol.compareTo(other.stockSymbol);
60          }
61          else if (this.timestamp != other.timestamp) {
62              return timestamp < other.timestamp ? -1 : 1;
63          }
64          else {
65             return 0;
66          }
67      }
68
69  }
```

## 组合键比较器定义

示例2-8将组合键比较器定义为CompositeKeyComparator类，通过实现compare()方法完成

两个CompositeKey对象的比较。如果两个对象相等，compare()方法返回0，如果第一个组合键小于第二个组合键，则返回-1，否则这个方法会返回+1。

示例2-8：定义组合键比较器

```
import org.apache.hadoop.io.WritableComparable;
import org.apache.hadoop.io.WritableComparator;

/**
 * CompositeKeyComparator
 *
 * 这个类的作用是比较两个CompositeKey。
 *
 */
public class CompositeKeyComparator extends WritableComparator {

   protected CompositeKeyComparator() {
      super(CompositeKey.class, true);
   }

   @Override
   public int compare(WritableComparable wc1, WritableComparable wc2) {
       CompositeKey ck1 = (CompositeKey) wc1;
       CompositeKey ck2 = (CompositeKey) wc2;

       int comparison = ck1.getStockSymbol().compareTo(ck2.getStockSymbol());
       if (comparison == 0) {
       // 在这里，股票代码相同
       if (ck1.getTimestamp() == ck2.getTimestamp()) {
               return 0;
           }
           else if (ck1.getTimestamp() < ck2.getTimestamp()) {
               return -1;
           }
           else {
               return 1;
           }
       }
       else {
           return comparison;
       }
   }
}
```

# 运行示例——老版本Hadoop API

表2-2所示的类使用老版本Hadoop API实现二次排序设计模式。

表2-2：使用老版本Hadoop API的实现类

| 类名 | 描述 |
| --- | --- |
| CompositeKey | 定义一个组合键 |
| CompositeKeyComparator | 实现组合键排序 |
| DateUtil | 定义一些有用的日期处理方法 |
| HadoopUtil | 定义工具函数 |
| NaturalKeyGroupingComparator | 定义自然键如何分组 |
| NaturalKeyPartitioner | 实现自然键分区 |
| NaturalValue | 定义一个自然值 |
| SecondarySortDriver | 向Hadoop提交一个作业 |
| SecondarySortMapper | 定义map() |
| SecondarySortReducer | 定义reduce() |

# 输入

```
# hadoop fs -ls /secondary_sort_chapter/input/
Found 1 items
-rw-r--r--   ... /secondary_sort_chapter/input/sample_input.txt

# hadoop fs -cat /secondary_sort_chapter/input/sample_input.txt
ILMN,2013-12-05,97.65
GOOG,2013-12-09,1078.14
IBM,2013-12-09,177.46
ILMN,2013-12-09,101.33
ILMN,2013-12-06,99.25
GOOG,2013-12-06,1069.87
IBM,2013-12-06,177.67
GOOG,2013-12-05,1057.34
```

# 运行MapReduce作业

```
# ./run.sh
...
13/12/12 21:13:20 INFO mapred.FileInputFormat: Total input paths to process : 1
13/12/12 21:13:21 INFO mapred.JobClient: Running job: job_201312122109_0002
13/12/12 21:13:22 INFO mapred.JobClient: map 0% reduce 0%
...
13/12/12 21:14:25 INFO mapred.JobClient: map 100% reduce 100%
...
13/12/12 21:14:26 INFO mapred.JobClient: Map-Reduce Framework
13/12/12 21:14:26 INFO mapred.JobClient: Map input records=8
13/12/12 21:14:26 INFO mapred.JobClient: Combine input records=0
13/12/12 21:14:26 INFO mapred.JobClient: Reduce input records=8
13/12/12 21:14:26 INFO mapred.JobClient: Reduce input groups=3
13/12/12 21:14:26 INFO mapred.JobClient: Combine output records=0
13/12/12 21:14:26 INFO mapred.JobClient: Reduce output records=3
```

```
13/12/12 21:14:26 INFO mapred.JobClient: Map output records=8
```

## 输出

```
# hadoop fs -ls /secondary_sort_chapter/output/
-rw-r--r-- 1 ... 0 2013-12-12 21:14 /secondary_sort_chapter/output/_SUCCESS
drwxr-xr-x - ... 0 2013-12-12 21:13 /secondary_sort_chapter/output/_logs
-rw-r--r-- 1 ... 0 2013-12-12 21:13 /secondary_sort_chapter/output/part-00000
-rw-r--r-- 1 ... 66 2013-12-12 21:13 /secondary_sort_chapter/output/part-00001
...
-rw-r--r-- 1 ... 0 2013-12-12 21:14 /secondary_sort_chapter/output/part-00008
-rw-r--r-- 1 ... 43 2013-12-12 21:14 /secondary_sort_chapter/output/part-00009
# hadoop fs -cat /secondary_sort_chapter/output/part*
GOOG (2013-12-05,1057.34)(2013-12-06,1069.87)(2013-12-09,1078.14)
ILMN (2013-12-05,97.65)(2013-12-06,99.25)(2013-12-09,101.33)
IBM (2013-12-06,177.67)(2013-12-09,177.46)
```

# 运行示例——新版本Hadoop API

表2-3所示的类使用新版本Hadoop API实现二次排序设计模式。

**表2-3：使用新版本Hadoop API的实现类**

| 类名 | 描述 |
|---|---|
| CompositeKey | 定义一个组合键 |
| CompositeKeyComparator | 实现组合键排序 |
| DateUtil | 定义一些有用的日期处理方法 |
| HadoopUtil | 定义工具函数 |
| NaturalKeyGroupingComparator | 定义自然键如何分组 |
| NaturalKeyPartitioner | 实现自然键分区 |
| NaturalValue | 定义一个自然值 |
| SecondarySortDriver | 向Hadoop提交一个作业 |
| SecondarySortMapper | 定义map() |
| SecondarySortReducer | 定义reduce() |

## 输入

```
# hadoop fs -ls /secondary_sort_chapter_new_api/input/
Found 1 items
-rw-r--r-- ... /secondary_sort_chapter_new_api/input/sample_input.txt
# hadoop fs -cat /secondary_sort_chapter_new_api/input/sample_input.txt
ILMN,2013-12-05,97.65
GOOG,2013-12-09,1078.14
IBM,2013-12-09,177.46
ILMN,2013-12-09,101.33
```

```
ILMN,2013-12-06,99.25
GOOG,2013-12-06,1069.87
IBM,2013-12-06,177.67
GOOG,2013-12-05,1057.34
```

## 运行MapReduce作业

```
# ./run.sh
...
13/12/14 21:18:25 INFO ... Total input paths to process : 1
...
13/12/14 21:18:25 INFO mapred.JobClient: Running job: job_201312142112_0002
13/12/14 21:18:26 INFO mapred.JobClient: map 0% reduce 0%

13/12/14 21:19:15 INFO mapred.JobClient: map 100% reduce 100%
13/12/14 21:19:16 INFO mapred.JobClient: Job complete: job_201312142112_0002
...
13/12/14 21:19:16 INFO mapred.JobClient: Map-Reduce Framework
13/12/14 21:19:16 INFO mapred.JobClient: Map input records=8
13/12/14 21:19:16 INFO mapred.JobClient: Spilled Records=16
13/12/14 21:19:16 INFO mapred.JobClient: Combine input records=0
13/12/14 21:19:16 INFO mapred.JobClient: Reduce input records=8
13/12/14 21:19:16 INFO mapred.JobClient: Reduce input groups=3
13/12/14 21:19:16 INFO mapred.JobClient: Combine output records=0
13/12/14 21:19:16 INFO mapred.JobClient: Reduce output records=3
13/12/14 21:19:16 INFO mapred.JobClient: Map output records=8
```

## 输出

```
# hadoop fs -cat /secondary_sort_chapter_new_api/output/part*
GOOG (2013-12-05,1057.34)(2013-12-06,1069.87)(2013-12-09,1078.14)
ILMN (2013-12-05,97.65)(2013-12-06,99.25)(2013-12-09,101.33)
IBM (2013-12-06,177.67)(2013-12-09,177.46)
```

这一章和前一章提供了实现二次排序设计模式的具体解决方案。下一章将介绍如何使用MapReduce/Hadoop和Spark实现Top N设计模式。

# 第3章

# Top 10列表

给定一组（key-as-string，value-as-integer）对，假设我们想要创建一个top N列表（其中N > 0）。Top N是一种设计模式（第1章解释过，设计模式是针对一个常见问题的独立于语言的可重用解决方案，利用设计模式，可以生成可重用的代码）。例如，如果key-as-string是一个URL，value-as-integer是访问这个URL的次数，你可能会问这样一个问题：上星期访问次数最多的10个URL（top 10）是哪些？对于这些键-值对类型，这种问题相当常见。可以把找出一个top 10列表的问题归为一种过滤模式（也就是说，需要过滤数据，找出top 10列表）。关于Top N设计模式的详细信息，可以参考Donald Miner和Adam Shook编写的《MapReduce Design Patterns》[18]。

这一章会为Top N设计模式提供5个完整的MapReduce解决方案，另外还会给出使用Apache Hadoop（使用传统MapReduce的map()和reduce()函数）和Apache Spark（使用弹性分布式数据集）的相关实现：

- 使用MapReduce/Hadoop的Top 10解决方案。我们假设所有输入键都是唯一的。也就是说，对于一个给定的输入集合{(*K*, *V*)}，所有*K*都是唯一的。
- 使用Spark的Top 10解决方案。我们假设所有输入键都是唯一的。也就是说，对于一个给定的输入集{(*K*, *V*)}，所有*K*都是唯一的。在这个解决方案中，我们不使用Spark的排序函数，如top()或takeOrdered()。
- 使用Spark的Top 10解决方案。我们假设所有输入键不是唯一的。也就是说，对于一个给定的输入集 {(*K*, *V*)}，所有*K*并不唯一。在这个解决方案中，我们不使用Spark的排序函数，如top()或takeOrdered()。
- 使用Spark的Top 10解决方案。我们假设所有输入键不是唯一的。也就是说，对于一个给定的输入集 {(*K*, *V*)}，所有*K*并不唯一。在这个解决方案中，我们要使用Spark强

大的排序函数takeOrdered()。

- 使用MapReduce/Hadoop的Top 10解决方案。我们假设所有输入键不是唯一的。也就是说，对于一个给定的输入集合{(*K*, *V*)}，所有*K*并不唯一。

我们的MapReduce解决方案不仅能生成top 10列表，另外还能找出top *N*列表（$N > 0$）。例如，我们可以找出“top 10喵星人”、“top 50最常访问的网站”或者“一个搜索引擎的top 100热门搜索查询”。

# Top N设计模式的形式化描述

令N是一个整数，而且$N > 0$。令L是一个List<Tuple2<T,Integer>>，其中T可以是任意类型（如字符串或URL）；L.size() = S; S > N; L的元素为：

$$\{(K_i, V_i), 1 \leqslant i \leqslant S\}$$

其中$K_i$类型为T，$V_i$为Integer类型（这是$K_i$的频度）。令sort(L)返回已排序的L值，这里使用频度作为键，如下所示：

$$\{(A_j, B_j), 1 \leqslant j \leqslant S, B_1 \geqslant B_2 \geqslant \ldots \geqslant B_S\}$$

其中$(A_j, B_j) \in$ L。则L的top N可以定义如下：

$$\text{topN}(L) = \{(A_j, B_j), 1 \leqslant j \leqslant N, B_1 \geqslant B_2 \geqslant \ldots \geqslant B_N \geqslant B_N+1 \geqslant \ldots \geqslant B_S\}$$

为了实现Top N，我们需要一个散列表数据结构，从而可以得到键的全序（键表示频度）。在Java中实现Top N有一种简单方法，可以使用SortedMap<K,V>[注1]（作为接口）和TreeMap<K,V>（作为SortedMap的一个实现类），然后将L的所有元素增加到topN，如果topN.size() > N（也就是说，任何时候都只保留N项），就要删除topN中的第一个元素（频度最小的元素）。示例3-1展示了列表L的Top N算法。

示例3-1：Top N算法

```
1 import scala.Tuple2;
2 import java.util.List;
3 import java.util.TreeMap;     // 类，实现SortedMap
4 import java.util.SortedMap;   // 接口，  维护有序映射
5 import <your-package>.T;      // 期望类型T
6 ...
7 static SortedMap<Integer, T> topN(List<Tuple2<T, Integer>> L, int N) {
8     if ( (L == null) || (L.isEmpty()) ) {
9        return null;
```

注1：这是一个映射，它会提供映射键的一个全序。这个映射根据键的自然顺序排序，或者根据一个比较器排序，比较器通常在创建有序映射时指定（资料来源：Java SE 7文档（*http://bit.ly/sortedmap*））。

```
10      }
11      SortedMap<Integer, T> topN = new TreeMap<Integer, T>();
12      for (Tuple2<T,Integer> element : L) {
13         // element._1 类型为T
14         // element._2 是频度，类型为Integer
15         topN.put(element._2, element._1);
16         // 只保留top N
17         if (topN.size() > N) {
18            // 删除频度最小的元素
19            topN.remove(topN.firstKey());
20         }
21      }
22      return topN;
23  }
```

# MapReduce/Hadoop实现：唯一键

对于我们的第一个MapReduce解决方案，令cats是一个包含3个属性（cat_id, cat_name和 cat_weight）的关系。如表3-1所示。假设我们有几十亿只猫（大数据）。

**表3-1：Cat属性**

| 属性名 | 属性类型 |
|---|---|
| cat_id | String |
| cat_name | String |
| cat_weight | Double |

令$N$为整数，而且$N$>0，假设我们希望根据cat_weight找出top $N$猫列表。在具体分析这个MapReduce解决方案之前，下面来看在SQL中如何表示一个top 10猫列表：

```
SELECT cat_id, cat_name, cat_weight
   FROM cats
      ORDER BY cat_weight DESC LIMIT 10;
```

SQL的解决方案非常直接，只需要对整个cats表排序。所以你可能想知道为什么我们不直接使用一个关系数据库和SQL来完成这个任务。简单地说，这是因为，大多数情况下我们的大数据并没有建构为关系数据库和数据库表，而且很多情况下需要解析半结构化数据（如日志文件或其他类型的数据）来完成排序。如果数据规模非常庞大，关系数据库可能会停止响应，而且不能很好地伸缩。

MapReduce解决方案相当简单：每个映射器找出一个本地top $N$列表（$N > 0$），然后把它传递到一个归约器。这个归约器从映射器传送来的所有本地top $N$列表中找出最终的top $N$列表。一般来讲，对于大多数MapReduce算法，如果只有一个归约器，这可能会有问题，有可能带来一个性能瓶颈（因为将由一个服务器上的一个归约器接收所有数据，数据量可能相当大，而所有其他集群节点什么也不做，所有压力和负载全部都在这一个

节点上）。不过在这里，我们的归约器不会带来性能问题。为什么呢？假设有1000个映射，每个映射器只会生成10个键-值对。因此，这个归约器只会得到10000 (10 × 1000) 个记录，这个数据量还不至于导致性能瓶颈！

Top *N*算法如图3-1所示。首先将输入分区为小块，每个小块发送到一个映射器。前面已经解释过，每个映射器会创建一个本地top 10列表，然后将这个本地top 10列表发送到归约器。发出映射器输出时，我们使用了一个归约器键，这样所有映射器的输出都将由一个归约器处理。

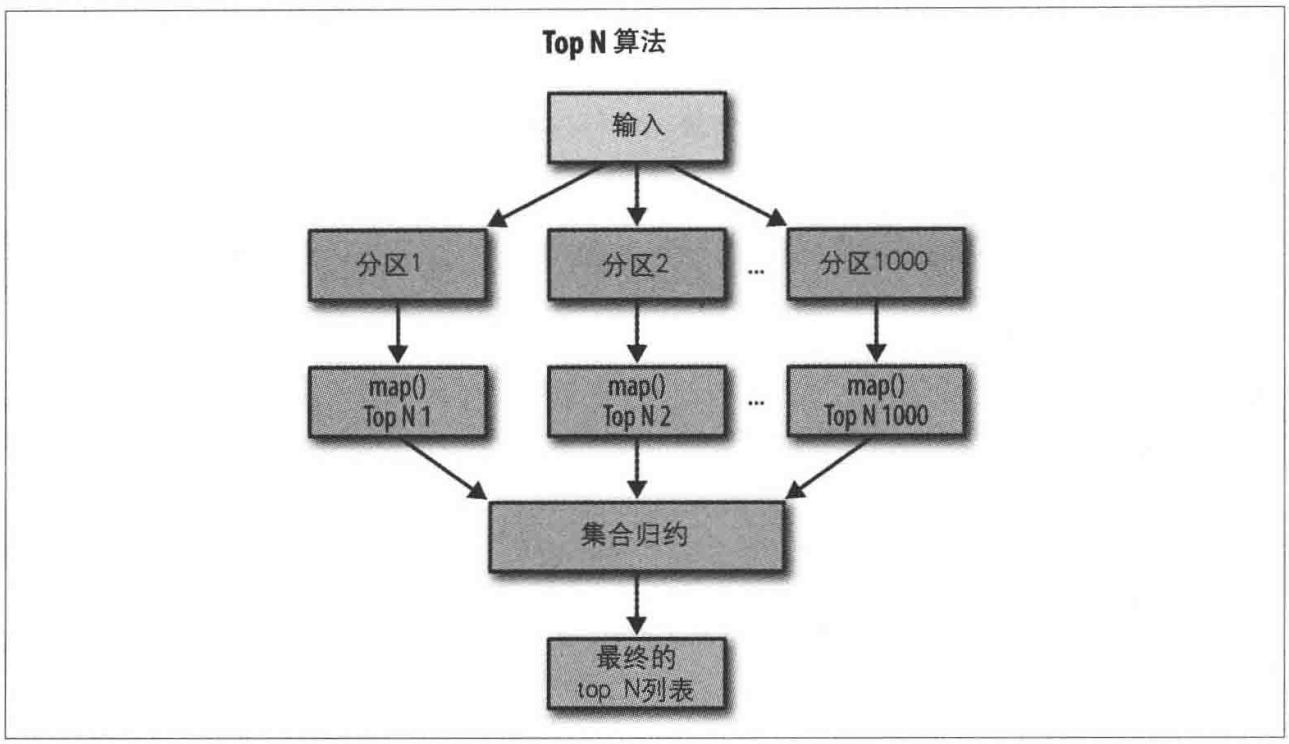



图3-1：Top N MapReduce算法：唯一键

要为top *N*列表指定参数，只需要使用MapReduce Configuration对象[注2]将N从驱动器（启动MapReduce作业的程序）传递到`map()`和`reduce()`函数。这个驱动器会设置`"top.n"`参数，`map()`和`reduce()`会在它们的`setup()`函数中读取这个参数。

下面重点分析如何找出top *N*猫列表。映射器类的结构如示例3-2所示。

示例3-2：查找top *N*猫列表的映射器类结构

```
1 // imports ...
2 public class TopN_Mapper {
3    // 定义本地top 10所需的数据结构
4    private SortedMap<Double, Text> top10cats = new TreeMap<Double, Text>();
5    private int N = 10; // 默认为top 10
6
```

注2： `org.apache.hadoop.conf.configuration`

```
   // 对应每个映射器执行一次setup()函数
   setup(Context context) {
     ...
   }

   map(key, value) {
     ... process (key, value) pair
   }

   // 对应每个映射器执行一次cleanup()函数
   cleanup(Context context) {
     ...
   }
}
```

接下来定义setup()函数，对应每个映射器会执行一次这个函数（参见示例3-3）。

示例3-3：top *N*列表的setup()

```
public class TopN_Mapper {
   ...
   private SortedMap<Double, Text> top10cats = new TreeMap<Double, Text>();
   private int N = 10; // 默认为top 10

/**
 * 对应每个映射器执行一次setup()函数
 * 这里建立 " top N猫列表"（top10cats）
 */
setup(Context context) {
   // "top.n" 由作业的驱动器设置
   Configuration conf = context.getConfiguration();
   N = conf.get("top.n");
}
```

map()函数接收一个输入块，生成一个本地top 10列表（参见示例3-4）。我们将使用不同的分隔符优化映射器和归约器的输入解析（避免不必要的字符串连接）。

示例3-4：top *N*列表的map()

```
/**
 * @param key由MapReduce框架生成，在这里忽略
 * @param value是一个String，有以下格式：
 * <cat_weight><,><cat_id><;><cat_name>
 */
map(key, value) {
   String[] tokens = value.split(",");
   // cat_weight = tokens[0];
   // <cat_id><;><cat_name> = tokens[1]
   Double weight = Double.parseDouble(tokens[0]);
   top10cats.put(weight, value);

   // 只保留top N
   if (top10cats.size() > N) {
      // 删除频度最小的元素
      top10cats.remove(top10cats.firstKey());
```

```
17    }
18 }
```

每个映射器接收一个猫分区。映射器会创建一个top 10列表（SortedMap<Double, Text>），完成之后，cleanup()方法发出这个列表（参见示例3-5）。需要说明，这里使用了一个键（通过调用NullWritable.get()来获取），可以保证所有映射器的输出都将由一个归约器处理。

示例3-5：top *N*列表的cleanup()

```
1 /**
2  * 在各个映射器的最后执行一次cleanup()函数
3  * 在这里建立"top N猫列表"（top10cats）
4  */
5 cleanup(Context context) {
6    // 现在从这个映射器发出top N元素
7    for (String catAttributes : top10cats.values() ) {
8        context.write(NullWritable.get(), catAttributes);
9    }
10 }
```

这个归约器得到所有本地top 10列表，然后创建一个最终的top 10列表，如示例3-6所示。

示例3-6：top *N*列表的reduce()

```
1 /**
2  * @param key 为null(单个归约器)
3  * @param values是一个String列表，列表中的每个元素
4  * 有以下格式： <cat_weight>,<cat_id>;<cat_name>
5  */
6 reduce(key, values) {
7    SortedMap<Double, Text> finaltop10 = new TreeMap< Double, Text>();
8
9    // 聚集所有本地top 10列表
10   for (Text catRecord : values) {
11      String[] tokens = catRecord.split(",");
12      Double weight = Double.parseDouble(tokens[0]);
13      finaltop10.put(weight, value);
14
15      if (finaltop10.size() > N) {
16         // 删除频度最小的元素
17         finaltop10.remove(finaltop10.firstKey());
18      }
19   }
20
21   // 发出最终的top 10列表
22   for (Text text : finaltop10.values()) {
23      context.write(NullWritable.get(), text);
24   }
25 }
```

## MapReduce/Hadoop中的实现类

MapReduce/Hadoop实现包括表3-2所示的类。需要说明，这个解决方案假设所有猫（也就是键）是唯一的，这意味着不会对输入完成聚集。

表3-2：Hadoop中的实现类

| 类名 | 类描述 |
| --- | --- |
| TopN_Driver | 提交作业的驱动器 |
| TopN_Mapper | 定义map() |
| TopN_Reducer | 定义reduce() |

TopN_Driver类从命令行读取N，在Hadoop Configuration对象中设置这个参数，这个对象将由map()函数读取。

## Top 10运行示例

### 输入

```
# hadoop fs -cat /top10list/input/sample_input.txt
12,cat1,cat1
13,cat2,cat2
14,cat3,cat3
15,cat4,cat4
10,cat5,cat5
100,cat100,cat100
200,cat200,cat200
300,cat300,cat300
1,cat001,cat001
67,cat67,cat67
22,cat22,cat22
23,cat23,cat23
1000,cat1000,cat1000
2000,cat2000,cat2000
```

### 脚本

```
# cat run.sh
#/bin/bash
export JAVA_HOME=/usr/java/jdk6
export HADOOP_HOME=/usr/local/hadoop-1.0.3
export HADOOP_HOME_WARN_SUPPRESS=true
export BOOK_HOME=/mp/data-algorithms-book
export APP_JAR=$BOOK_HOME/dist/data_algorithms_book.jar
export PATH=$PATH:$JAVA_HOME/bin:$HADOOP_HOME/bin
#
INPUT="/top10list/input"
OUTPUT="/top10list/output"
HADOOP_HOME/bin/hadoop fs -rmr $OUTPUT
```

```
HADOOP_HOME/bin/hadoop jar $APP_JAR TopN_Driver $INPUT $OUTPUT
```

### 示例运行日志

```
# ./run.sh
...
added manifest
adding: TopN_Driver.class(in = 3481) (out= 1660)(deflated 52%)
adding: TopN_Mapper.class(in = 3537) (out= 1505)(deflated 57%)
adding: TopN_Reducer.class(in = 3693) (out= 1599)(deflated 56%)
rmr: cannot remove /top10list/output: No such file or directory.
13/03/06 15:17:54 INFO TopN_Driver: inputDir=/top10list/input
13/03/06 15:17:54 INFO TopN_Driver: outputDir=/top10list/output
13/03/06 15:17:54 INFO TopN_Driver: top.n=10
13/03/06 15:17:54 INFO input.FileInputFormat: Total input paths to process : 1
...
13/03/06 15:17:55 INFO mapred.JobClient: Running job: job_201303061200_0022
13/03/06 15:17:56 INFO mapred.JobClient:  map 0% reduce 0%
13/03/06 15:18:10 INFO mapred.JobClient:  map 100% reduce 0%
13/03/06 15:18:22 INFO mapred.JobClient:  map 100% reduce 100%
13/03/06 15:18:27 INFO mapred.JobClient: Job complete: job_201303061200_0022
...
13/03/06 15:18:27 INFO mapred.JobClient: Map-Reduce Framework
13/03/06 15:18:27 INFO mapred.JobClient:   Map output materialized bytes=193
13/03/06 15:18:27 INFO mapred.JobClient:   Map input records=14
13/03/06 15:18:27 INFO mapred.JobClient:   Reduce shuffle bytes=0
...
13/03/06 15:18:27 INFO mapred.JobClient:   Reduce input records=10
13/03/06 15:18:27 INFO mapred.JobClient:   Reduce input groups=1
13/03/06 15:18:27 INFO mapred.JobClient:   Combine output records=0
13/03/06 15:18:27 INFO mapred.JobClient:   Reduce output records=10
13/03/06 15:18:27 INFO mapred.JobClient:   Map output records=10
13/03/06 15:18:27 INFO TopN_Driver: run(): status=true
13/03/06 15:18:27 INFO TopN_Driver: returnStatus=0
```

### 输出

```
# hadoop fs -cat /top10list/output/part*
14.0    14,cat3,cat3
15.0    15,cat4,cat4
22.0    22,cat22,cat22
23.0    23,cat23,cat23
67.0    67,cat67,cat67
100.0   100,cat100,cat100
200.0   200,cat200,cat200
300.0   300,cat300,cat300
1000.0  1000,cat1000,cat1000
2000.0  2000,cat2000,cat2000
```

## 查找Top 5

默认会返回一个top 10列表，不过如果想得到一个top 5列表，只需要传入另一个参数：

```
$ INPUT="/top10list/input"
$ OUTPUT="/top10list/output"
$ $HADOOP_HOME/bin/hadoop jar $APP_JAR TopN_Driver $INPUT $OUTPUT 5
$ hadoop fs -cat /top10list/output/*
100.0  100,cat100,cat100
200.0  200,cat200,cat200
300.0  300,cat300,cat300
1000.0 1000,cat1000,cat1000
2000.0 2000,cat2000,cat2000
```

## 查找Bottom 10

要查找末尾10项（bottom 10）而不是前10项（top 10），只需要修改一行代码。

将下面的代码：

```
// 查找top 10
if (top10cats.size() > N) {
     // 删除频度最小的元素
     top10cats.remove(top10cats.firstKey());
}
```

替换为：

```
// 查找bottom 10
if (top10cats.size() > N) {
     // 删除频度最大的元素
     top10cats.remove(top10cats.lastKey());
}
```

# Spark实现：唯一键

在这个Spark实现中，我们假设对于所有给定的输入（*K*, *V*)对，*K*是唯一的。与传统的MapReduce/Hadoop相比，Spark提供了更高层的抽象，它提供了一组丰富的函数式编程API，使得MapReduce编程非常容易。Spark可以从本地文件系统以及Hadoop的HDFS读/写数据。它使用Hadoop客户端库读写HDFS和其他支持Hadoop的存储系统。

这里假设你已经在Hadoop上运行了一个Spark集群（或者也可以在YARN上运行Spark，这样就无需启动Spark集群）。要了解如何安装Spark/Hadoop，可以参考：

- Apache Spark网站（*http://spark.apache.org/*）。
- Holden Karau的《Fast Data Processing with Spark》（Packt Publishing出版）。

## RDD刷新器

要掌握Spark，首先来回顾RDD（弹性分布式数据集）的概念。在第1章中我们已经了解

到，RDD是Spark中的一个基本抽象：它表示一个不可变的元素分区集合，这些分区集合可以并行地处理。无需处理不同类型的输入/输出，你只需要处理一个RDD，它可以表示不同类型的输入/输出。例如，下面的代码段表示两个RDD（lines和words）：

```
1 JavaSparkContext ctx = new JavaSparkContext();
2 JavaRDD<String> lines = ctx.textFile(args[1], 1);
3 JavaRDD<String> words = lines.flatMap(new FlatMapFunction<String, String>() {
4    public Iterable<String> call(String s) {
5       return Arrays.asList(s.split(" "));
6    }
7 });
```

表3-3对这个代码给出了解释。

**表3-3：Spark RDD解释**

| 行 | 描述 |
|---|---|
| 1 | 创建一个Java Spark上下文对象（JavaSparkContext）。这表示与一个Spark集群的连接，可以用来创建RDD、累加器以及这个集群上的广播变量 |
| 2 | 创建一个新的RDD（JavaRDD<String>），它将一个文本文件表示为一个行集（每一行/记录分别是一个String对象） |
| 3~7 | 从一个现有的RDD创建一个新的RDD（JavaRDD<String>），这表示词法分析得到的一个单词集。FlatMapFunction<T, R>包含一个类型为T => Iterable<R>的函数 |

Spark非常擅长从现有的RDD创建新的RDD。例如，下面我们使用lines创建一个新的RDD（JavaPairRDD<String, Integer> pairs）。

```
1 JavaPairRDD<String,Integer> pairs =
2    lines.map(new PairFunction<String, String, Integer>() {
3    public Tuple2<String,Integer> call(String s) {
4       String[] tokens = s.split(",");
5       return new Tuple2<String,Integer>(tokens[0],
6                                         Integer.parseInt(tokens[1])
7                                        );
8    }
9 });
```

JavaPairRDD<String,Integer>中的每一项分别表示一个Tuple2<String,Integer>。在这里我们假设每个输入记录有两个token：<String><,><Integer>。

## Spark的函数类

Java不支持匿名或首类函数（JDK8中实现了这样一些函数），所以我们必须通过扩

展Function[注3]、Function2[注4]等类来实现函数。Spark的RDD方法（如collect()和countByKey()）会返回Java集合数据类型，如java.util.List和java.util.Map。要表示键-值对，可以使用scala.Tuple2，这是一个Scala类，另外可以使用new Tuple2<K, V>(key, value)创建一个新的键－值对。

Spark的强大之处就体现在Java API使用的函数类（见表3-4），其中每个类都有一个抽象方法call()，程序员必须实现这个方法。

**表3-4：Java API使用的Spark函数类**

| Spark Java类 | 函数类型 |
|---|---|
| Function<T, R> | T => R |
| DoubleFunction<T> | T => Double |
| PairFunction<T, K, V> | T => Tuple2<K, V> |
| FlatMapFunction<T, R> | T => Iterable<R> |
| DoubleFlatMapFunction<T> | T => Iterable<Double> |
| PairFlatMapFunction<T, K, V> | T => Iterable<Tuple2<K, V>> |
| Function2<T1, T2, R> | T1, T2 => R （两参数函数） |

## Spark Top N模式回顾

在Spark中，可以在一个驱动器中编写整个大数据处理作业，这要归功于Spark提供的丰富的MapReduce范式高层抽象。在用Spark提供完整的top 10解决方案之前，我们先来回顾Top 10算法。这里假设输入记录有以下格式：

```
<Integer><,><String>
```

我们的目标是为给定的输入找出相应的top 10列表。首先，将输入分区为区段（segment）（假设要将输入分区到1000个映射器，每个映射器独立地处理一个区段）：

```
1 class mapper :
2     setup(): initialize top10 SortedMap<Integer, String>
3
4     map(key, inputRecord):
5          key is system generated and ignored here
6          insert inputRecord into top10 SortedMap
7          if (length of top10 is greater than 10) {
8              truncate list to a length of 10
9          }
10
11    cleanup(): emit top10 as SortedMap<Integer, String>
```

注3：org.apache.spark.api.java.function.Function。

注4：org.apache.spark.api.java.funtion.Funtion2。

由于Spark在映射器或归约器中不支持传统MapReduce/Hadoop的setup()和cleanup()函数，我们要使用JavaPairRDD.mapPartitions()方法实现同样的功能。

不过，如何在Spark中实现与setup()和cleanup()函数等价的函数呢？在传统MapReduce/Hadoop框架中，我们会如下使用setup()和cleanup()：

```
1 public class ClassicMapper extends Mapper<K,V,K2,V2> {
2   private ExpensiveConnection expensiveConn;
3   @Override
4   protected void setup(Context context) {
5     expensiveConn = ...;
6   }
7   ...
8   @Override
9   protected void cleanup(Context context) {
10    expensiveConn.close();
11  }
12  ...
13  // map()功能: 使用expensiveConn
14 }
```

在Spark中，可以用mapPartitions()得到同样的功能：

```
1 JavaRDD<Tuple2<K2,V2>> partitions = pairRDD.mapPartitions(
2     new FlatMapFunction<Iterator<Tuple2<K,V>>, Tuple2<K2,V2>>() {
3     @Override
4     public Iterable<Tuple2<K2,V2>> call(Iterator<Tuple2<K,V>> iter) {
5        setup();
6        while (iter.hasNext()) {
7           // map()功能
8        }
9        cleanup();
10       return <the-result>;
11    }
12 });
```

归约器的任务与映射器类似：它会从一个给定集合（包含映射器生成的所有top 10列表）中找出最终的top 10列表。归约器得到一个SortedMap<Integer, String>对象集合作为输入，然后创建一个最终的SortedMap<Integer, String>作为输出：

```
1 class reducer:
2     setup(): initialize finaltop10 SortedMap<Integer, String>
3
4     reduce(key, List<SortedMap<Integer, String>>):
5             build finaltop10 from List<SortedMap<Integer, String>>
6             emit finaltop10
```

## 完整的Spark Top 10解决方案

首先给出所有主要步骤（参见示例3-7），然后详细描述各个步骤。我们将在Spark中用一个Java类（表示为Top10.java）查找top 10列表。

示例3-7：Spark中的top 10程序

```
1 // 步骤1：导入必要的类
2 public class Top10 {
3    public static void main(String[] args) throws Exception {
4      // 步骤2：确保有正确的输入参数
5      // 步骤3：创建与Spark master的连接
6      // 步骤4：从HDFS读取输入文件并创建第一个RDD
7      // 步骤5：创建一组Tuple2<Integer, String>
8      // 步骤6：为各个输入分区创建一个本地top 10
9      // 步骤7：收集所有本地top 10并创建最终的top 10列表
10     // 步骤8：输出最终的top 10列表
11     System.exit(0);
12   }
13 }
```

### 步骤1：导入必要的类

示例3-8展示了第一步需要导入的类。

示例3-8：步骤1：导入必要的类

```
1 // 步骤1：导入必要的类
2 import scala.Tuple2;
3 import org.apache.spark.api.java.JavaRDD;
4 import org.apache.spark.api.java.JavaPairRDD;
5 import org.apache.spark.api.java.JavaSparkContext;
6 import org.apache.spark.api.java.function.FlatMapFunction;
7 import org.apache.spark.api.java.function.Function2;
8 import org.apache.spark.api.java.function.PairFunction;
9
10 import java.util.Arrays;
11 import java.util.List;
12 import java.util.Map;
13 import java.util.TreeMap;
14 import java.util.SortedMap;
15 import java.util.Iterator;
16 import java.util.Collections;
```

### 步骤2：处理输入参数

这一步（参见示例3-9）要确保有一个输入参数：HDFS输入文件。例如：

```
args[0]: /top10/input/top10data.txt
```

示例3-9：步骤2：处理输入参数

```
1    // 步骤2：确保有正确的输入参数
```

```
2    if (args.length < 1) {
3        System.err.println("Usage: Top10 <hdfs-file>");
4        System.exit(1);
5    }
6    String inputPath = args[0];
7    System.out.println("inputPath: <hdfs-file>="+inputPath);
```

### 步骤3：连接Spark master

在这一步中（参见示例3-10），我们要为Spark master创建一个连接对象（JavaSparkContext）。后面将用ctx对象创建RDD。

示例3-10：步骤3：连接Spark master

```
1    // 步骤3：创建与Spark master的连接
2    JavaSparkContext ctx = new JavaSparkContext();
```

### 步骤4：从HDFS读取输入文件并创建RDD

在这一步中（参见示例3-11），将从HDFS读取一个文件，并创建一个RDD（JavaRDD<String>）。

示例3-11：步骤4：从HDFS读取输入文件

```
1    // 步骤4：从HDFS读取输入文件并创建第一个RDD
2    // 输入记录格式：<string-key><,><integer-value>
3    JavaRDD<String> lines = ctx.textFile(inputPath, 1);
```

### 步骤5：Top10类

在这一步中（参见示例3-12），我们将从现有的RDD（JavaRDD<String>）创建一个新的RDD（JavaPairRDD<Integer, String>）。这个PairFunction类有3个参数：第一个参数（T）是一个输入，后两个参数（K, V）是输出：

- Spark Java类：PairFunction<T, K, V>
- 函数类型：T => Tuple2<K, V>

示例3-12：步骤5：创建一组Tuple2

```
1    // 步骤5：创建一组Tuple2<String,Integer>
2    // PairFunction<T, K, V>
3    // T => Tuple2<K, V>
4    JavaPairRDD<String,Integer> pairs =
5        lines.mapToPair(new PairFunction<
6                                        String, // input (T)
7                                        String, // K
8                                        Integer // V
9                                       >() {
10     public Tuple2<String,Integer> call(String s) {
11       String[] tokens = s.split(","); // cat24,123
12       return new Tuple2<String,Integer>(tokens[0], Integer.parseInt(tokens[1]));
```

```
13      }
14    });
```

## 步骤6：为各个输入分区创建本地top 10列表

在这一步中，我们又要从现有的RDD（JavaPairRDD<String,Integer>）创建一个新的RDD（JavaRDD<SortedMap<Integer, String>>）。我们已经讨论过，mapPartitions()是一个非常强大的方法，可以用来生成传统MapReduce/Hadoop方法setup()、map()和cleanup()的功能。在示例3-13中可以看到，我们将从各个分区分别创建一个本地top 10列表。

示例3-13：步骤6：为各个输入分区创建一个本地top 10列表

```
1   // 步骤6：为各个输入分区创建一个本地top 10列表
2   JavaRDD<SortedMap<Integer, String>> partitions = pairs.mapPartitions(
3       new FlatMapFunction<
4                           Iterator<Tuple2<String,Integer>>,
5                           SortedMap<Integer, String>
6                       >() {
7       @Override
8       public Iterable<SortedMap<Integer, String>>
9          call(Iterator<Tuple2<String,Integer>> iter) {
10         SortedMap<Integer, String> top10 = new TreeMap<Integer, String>();
11         while (iter.hasNext()) {
12            Tuple2<String,Integer> tuple = iter.next();
13            // tuple._1 : 唯一键，如cat_id
14            // tuple._2 : 项的频度 (cat_weight)
15            top10.put(tuple._2, tuple._1);
16            // 只保留top N
17            if (top10.size() > 10) {
18               // 删除频度最小的元素
19               top10.remove(top10.firstKey());
20            }
21         }
22         return Collections.singletonList(top10);
23      }
24    });
```

第10行将完成与MapReduce/Hadoop setup()等价的功能（也就是说，它会创建和初始化一个本地top 10列表，表示为SortedMap<Integer, String>）。第11～20行模拟map()函数。最后的第22行实现了cleanup()方法的功能（返回一个本地top 10列表的结果）。

## 步骤7：使用collect()创建最终的top 10列表

这一步会迭代处理为每个分区创建的所有本地top 10列表，并创建一个最终的top 10列表。为了得到所有本地top 10列表，我们使用了collect()方法，如示例3-14所示。

示例3-14：步骤7：使用collect()创建最终的top 10列表

```
1   // 步骤7：收集所有本地top 10列表，创建一个最终的top 10列表
2   SortedMap<Integer, String> finaltop10 = new TreeMap<Integer, String>();
```

```
    List<SortedMap<Integer, String>> alltop10 = partitions.collect();
    for (SortedMap<Integer, String> localtop10 : alltop10) {
        // weight = tuple._1 (frequency)
        // catname = tuple._2
        for (Map.Entry<Integer, String> entry : localtop10.entrySet()) {
            // System.out.println(entry.getKey() + "--" + entry.getValue());
            finaltop10.put(entry.getKey(), entry.getValue());
            // 只保留top 10
            if (finaltop10.size() > 10) {
                // 删除频度最小的元素
                finaltop10.remove(finaltop10.firstKey());
            }
         }
     }
```

这一步还有一个替代解决方案：可以使用JavaRDD.reduce()。这个替代方案如示例3-15所示。在查看这个解决方案之前，先来看reduce()的签名：

```
T reduce(Function2<T,T,T> f)
// 使用指定的二元操作符（具有交换性和结合性），
// 归约这个RDD的元素
```

示例3-15：步骤7：使用reduce()创建最终的top 10列表

```
// 步骤7：收集所有本地top 10并创建最终的top 10列表
SortedMap<Integer, String> finaltop10 = partitions.reduce(
    new Function2<
                SortedMap<Integer, String>, // m1 (作为输入)
                SortedMap<Integer, String>, // m2 (作为输入)
                SortedMap<Integer, String>  // 输出：合并m1和m2
            >() {
    @Override
    public SortedMap<Integer, String> call(SortedMap<Integer, String> m1,
                                           SortedMap<Integer, String> m2) {
       // 将m1和m2合并到一个top 10列表
       SortedMap<Integer, String> top10 = new TreeMap<Integer, String>();

       // 处理m1
       for (Map.Entry<Integer, String> entry : m1.entrySet()) {
           top10.put(entry.getKey(), entry.getValue());
           if (top10.size() > 10) {
               // 只保留top 10，删除频度最小的元素
               top10.remove(top10.firstKey());
           }
       }

       // 处理m2
       for (Map.Entry<Integer, String> entry : m2.entrySet()) {
           top10.put(entry.getKey(), entry.getValue());
           if (top10.size() > 10) {
               // 只保留top 10，删除频度最小的元素
               top10.remove(top10.firstKey());
           }
        }
```

```
31
32          return top10;
33      }
34  });
```

### 步骤8：发出最终的top 10列表

这一步（参见示例3-16）会迭代处理SortedMap<Integer, String>，发出最终的top 10列表。

示例3-16：步骤8：发出最终的top 10列表

```
// 步骤8：输出最终的top 10列表
System.out.println("=== top-10 list ===");
for (Map.Entry<Integer, String> entry : finaltop10.entrySet()) {
    System.out.println(entry.getKey() + "--" + entry.getValue());
}
```

## 运行示例：查找Top 10

在一个包括3个节点的集群上运行这个示例，3个节点分别是sparkserver100、sparkserver200和sparkserver300，其中sparkserver100是Spark主节点。

### 输入

对于这个运行示例，我创建了以下示例输入文件：

```
# hadoop fs -ls /top10/input/top10data.txt
Found 1 items
-rw-r--r--   ... 161 2014-04-28 14:22 /top10/input/top10data.txt

# hadoop fs -cat /top10/input/top10data.txt
cat1,12
cat2,13
cat3,14
cat4,15
cat5,10
cat100,100
cat200,200
cat300,300
cat1001,1001
cat67,67
cat22,22
cat23,23
cat1000,1000
cat2000,2000
cat400,400
cat500,500
```

### 脚本

```
# cat run_top10.sh
```

```
export JAVA_HOME=/usr/java/jdk7
export SPARK_HOME=/usr/local/spark-1.1.0
export SPARK_MASTER=spark://myserver100:7077
BOOK_HOME=/mp/data-algorithms-book
APP_JAR=$BOOK_HOME/data_algorithms_book.jar
INPUT=/top10/top10data.txt
prog=org.dataalgorithms.chap03.spark.Top10
# 在Spark独立集群上运行
$SPARK_HOME/bin/spark-submit \
  --class $prog \
  --master $SPARK_MASTER \
  --executor-memory 2G \
  --total-executor-cores 20 \
    $APP_JAR $INPUT
```

## 使用Spark集群运行Top 10示例

下面给出运行示例的日志输出，这里为适应版面对输出格式做了调整：

```
# ./run_top10.sh
inputPath: <file>=/top10/top10data.txt
...
14/06/03 22:42:24 INFO spark.SparkContext: Job finished:
collect at Top10.java:69, took 4.521464436 s
key= cat10 value= 10
key= cat1 value= 1
key= cat10000 value= 10000
key= cat400 value= 400
key= cat500 value= 500
key= cat2 value= 2
key= cat4 value= 4
key= cat6 value= 6
key= cat1200 value= 1200
key= cat10 value= 10
key= cat11 value= 11
key= cat12 value= 12
key= cat13 value= 13
key= cat50 value= 50
key= cat51 value= 51
key= cat45 value= 45
key= cat46 value= 46
key= cat200 value= 200
key= cat234 value= 234
...
collect at Top10.java:116, took 1.15125893 s
45--cat45
46--cat46
50--cat50
51--cat51
200--cat200
234--cat234
400--cat400
500--cat500
1200--cat1200
```

```
10000--cat10000
```

## 指定Top N参数

在MapReduce/Hadoop实现中，我们能够指定*N*作为一个参数，所以可以找出top 10、top 20或top 100。不过在Spark中如何将*N*作为参数？答案是使用Broadcast对象。Broadcast允许将*N*作为全局共享数据，从而可以从任何集群节点访问。我们的做法如下：

```
 1 import org.apache.spark.broadcast.Broadcast;
 2 import org.apache.spark.api.java.JavaSparkContext;
 3 ...
 4 int topN = <any-integer-number-greater-than-zero>;
 5 ...
 6 JavaSparkContext context = new JavaSparkContext();
 7 ...
 8 // broadcastTopN可以从任何集群节点访问
 9 final Broadcast<Integer> broadcastTopN = context.broadcast(topN);
10
11 RDD.map() {
12     ...
13     final int topN = broadcastTopN.value();
14     // 使用topN
15     ...
16 }
17
18 RDD.groupBy() {
19     ...
20     final int topN = broadcastTopN.value();
21     // 使用topN
22     ...
23 }
```

下面来分解介绍这个代码：

- 第1～2行导入必要的类。Broadcast类允许我们定义全局共享数据结构，这样就可以从任意集群节点访问这些数据。要定义一个类型为T的全局共享数据结构，一般格式如下：

  ```
  JavaSparkContext context = <an-instance-of-JavaSparkContext>;
  ...
  T t = <create-data-structure-of-type-T>;
  Broadcast<T> broadcastT = context.broadcast(t);
  ```

  广播这个数据结构broadcastT之后，可以从任何集群节点在映射器、归约器和转换器中读取这个数据。需要说明，在这个例子中，我们只广播了一个整数值（对应top *N*），不过完全可以广播你想要共享的任何数据结构。

- 第4行定义topN，这可以是top 10、top 20或top 100。
- 第6行创建JavaSparkContext的一个实例。

- 第9行为topN定义一个全局共享数据结构（可以是任何值）。
- 第13行和第20行读取并使用对应topN的全局共享数据结构（从任意集群节点）。读取类型为T的全局共享数据结构时，一般格式如下：

```
T t = broadcastT.value();
```

## 查找Bottom N

如果想找出末尾的N项而不是前N项该怎么做呢？可以使用另一个参数来实现，所有集群节点会共享这个参数。这个共享变量名为direction，值可以是"top" 或"bottom"：

```
1 import org.apache.spark.broadcast.Broadcast;
2 import org.apache.spark.api.java.JavaSparkContext;
3 ...
4 final int N = <any-integer-number-greater-than-zero>;
5 ...
6 JavaSparkContext context = <create-a-context-object>;
7 ...
8 String direction = "top"; // 或"bottom"
9 ...
10 final Broadcast<Integer> broadcastN = context.broadcast(N);
11 final Broadcast<String> broadcastDirection = context.broadcast(direction);
```

现在，根据broadcastDirection的值，我们要删除第一项（如果direction等于"top"）或者最后一项（如果direction等于"bottom"）。所有代码中的做法必须一致：

```
1 final int N = broadcastN.value();
2 final String direction = broadcastDirection.value();
3 SortedMap<Integer, String> finalN = new TreeMap<Integer, String>();
4 ...
5 if (finalN.size() > N) {
6    if (direction.equals("top")) {
7        // 删除频度最小的元素
8        finalN.remove(finalN.firstKey());
9    }
10   else {
11       // direction.equals("bottom")
12       // 删除频度最大的元素
13       finalN.remove(finalN.lastKey());
14   }
15 }
```

# Spark实现：非唯一键

在这个实现中，我们假设对于所有给定的（$K$, $V$）对，$K$不唯一。由于$K$不是唯一的，所以必须增加额外的步骤来确保在应用top 10（或bottom 10）算法之前键是唯一的。为了帮助你了解非唯一键的解决方案，这里给出一个简单的例子。下面假设我们想找出一个网

站上最常被访问的10个URL。另外，假设我们有3个Web服务器（Web服务器1、Web服务器2和Web服务器3），每个Web服务器采用以下形式收集URL：

```
(URL, count)
```

下面再假设我们只有7个URL，分别标为*A*、*B*、*C*、*D*、*E*、*F*和*G*。表3-5列出了对应各个Web服务器生成的URL统计情况。

**表3-5：对应各个Web服务器的（URL, count）**

| Web服务器1 | Web服务器2 | Web服务器3 |
|---|---|---|
| (A, 2) | (A, 1) | (A, 2) |
| (B, 2) | (B, 1) | (B, 2) |
| (C, 3) | (C, 3) | (C, 1) |
| (D, 2) | (E, 1) | (D, 2) |
| (E, 1) | (F, 1) | (E, 1) |
| (G, 2) | (G, 2) | (F, 1) |
| | | (G, 2) |

假设我们希望得到最常访问的2个URL。如果先得到各个Web服务器的本地top 2，然后再得到所有3个本地top 2列表中的top 2，这个结果并不正确。原因是，所有Web服务器上的URL并不唯一。要得到正确的结果，首先必须由所有输入创建一组唯一的URL，然后将这些唯一的URL分区到$M$（$> 0$）个分区。接下来得到对应各个分区的本地top 2，最后在所有本地top 2列表中确定最终的top 2。对于这个例子，生成的唯一URL如表3-6所示。

**表3-6：聚集的（URL, count）对**

| 聚集/归约的（URL, count）对 |
|---|
| (A, 5) |
| (B, 5) |
| (C, 7) |
| (D, 4) |
| (E, 3) |
| (F, 2) |
| (G, 6) |

现在假设将所有唯一URL分区到两个分区，如表3-7所示。

表3-7：每个分区的（URL, count）

| 分区1 | 分区2 |
|---|---|
| (A, 5) | (D, 4) |
| (B, 4) | (E, 3) |
| (C, 7) | (F, 2) |
| | (G, 6) |

下面找出所有数据的top 2：

```
top-2(Partition-1) = { (C, 7), (A, 5) }
top-2(Partition-2) = { (G, 6), (D, 4) }
top-2(Partition-1, Partition-2) = { (C, 7), (G, 6) }
```

所以，重点在于，找出一组（*K*, *V*）对的top *N*之前，必须确保所有*K*是唯一的。

这一节会提供Top N设计模式的一个Spark/Hadoop实现，这里假设所有*K*不是唯一的。这个Top 10算法的主要步骤如下：

1. 确保所有*K*是唯一的。要保证*K*是唯一的，我们要把输入映射到`JavaPairRDD<K, V>`对，然后交给`reduceByKey()`。
2. 将所有唯一的（*K*, *V*）对划分为*M*个分区。
3. 找出各个分区的top *N*（我们称为本地top *N*）。
4. 找出所有本地top *N*的最终top *N*。

非唯一键的Top 10算法如图3-2所示。

## 完整的Spark Top 10解决方案

对于top 10的这个Spark解决方案，首先我会在示例3-17中给出所有高层步骤，然后展开介绍各个步骤。我们将在Spark中使用一个Java类（表示为`Top10NonUnique.java`）查找top 10。需要说明，这个解决方案是通用的，尽管其中一个步骤的目的是确保所有键是唯一的，但在实际中键是否唯一并没有影响。

示例3-17：对应非唯一键的Spark Top 10程序

```
1 package org.dataalgorithms.chap03;
2 // 步骤1：导入必要的类和接口
3 /**
4  * 假设：对于所有输入（K, V），K是不唯一的。
5  * 这个类实现了Top N设计模式（N > 0）。
6  * 主要假设为对于所有输入（K, V）对，K
7  * 非唯一。这说明，我们可能会看找到类似
8  * (A, 2), ..., (A, 5),...的项。如果发现重复的K,则
9  * 累加它们对应的值，然后创建一个唯一的K。
```

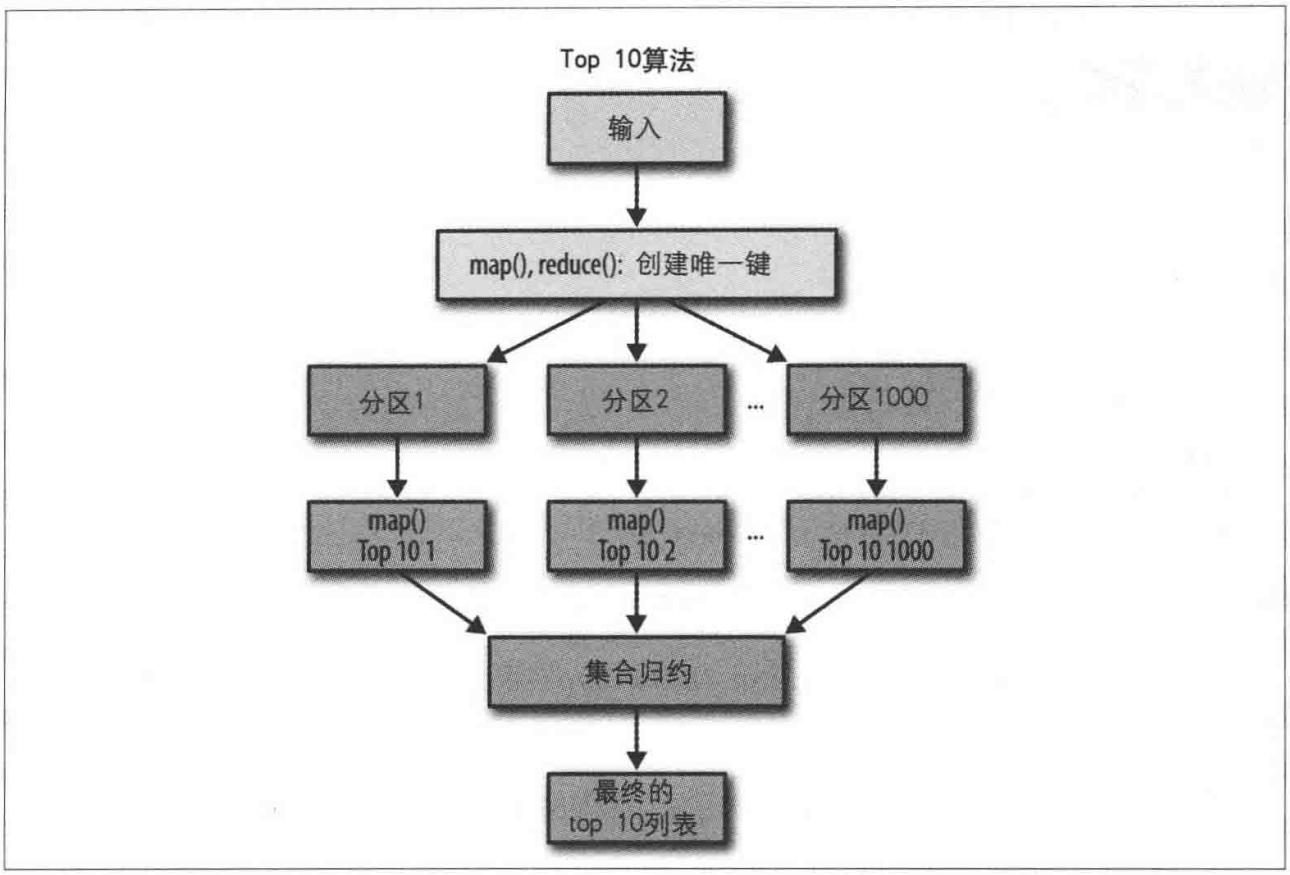



图3-2：Top 10 MapReduce算法：非唯一键

```
 * 如果有（A, 2）和（A, 5），则创建的唯一项为
 * (A, 7)，这里（7=2+5）。
 *
 * 这个类也可以用来找出bottom N
 * （只需要保留集合中最小的N个元素）。
 *
 * Top 10 设计模式：Top 10结构
 *
 * 1. 映射（输入） => (K, V)
 * 2. 归约(K, List<V1, V2, ..., Vn>) => (K, V),
 * 其中V = V1+V2+...+Vn; 现在所有K都是唯一的。
 * 3. 将(K,V)对划分到P个分区。
 * 4. 找出各个分区的top N（我们称为一个本地top N）
 * 5. 从所有本地top N找出最终的top N
 *
 * @author Mahmoud Parsian
 *
 */
public class Top10NonUnique {
   public static void main(String[] args) throws Exception {
      // 步骤2：处理输入参数
      // 步骤3：创建一个Java Spark上下文对象
      // 步骤4：将topN广播到所有集群节点
      // 步骤5：从输入创建一个RDD
      // 步骤6：RDD分区
      // 步骤7：输入（T）映射到（K,V）对
```

```
36        // 步骤8：归约重复的K
37        // 步骤9：创建本地top N
38        // 步骤10：查找最终top N
39        // 步骤11：发出最终top N
40        System.exit(0);
41    }
42 }
```

## 输入

这里提供了示例输入。我们使用这个HDFS输入来打印各个步骤的输出。可以看到，键是不唯一的。在应用Top 10算法之前，首先要生成唯一的键：

```
# hadoop fs -ls /top10/top10input/
Found 3 items
-rw-r--r-- 3 hadoop hadoop 24 2014-08-31 12:50 /top10/top10input/file1.txt
-rw-r--r-- 3 hadoop hadoop 24 2014-08-31 12:50 /top10/top10input/file2.txt
-rw-r--r-- 3 hadoop hadoop 28 2014-08-31 12:50 /top10/top10input/file3.txt

# hadoop fs -cat /top10/top10input/file1.txt
A,2
B,2
C,3
D,2
E,1
G,2

# hadoop fs -cat /top10/top10input/file2.txt
A,1
B,1
C,3
E,1
F,1
G,2

# hadoop fs -cat /top10/top10input/file3.txt
A,2
B,2
C,1
D,2
E,1
F,1
G,2
```

## 步骤1：导入必要的类和接口

示例3-18导入所需的所有Java类和接口。

示例3-18：步骤1：导入必要的类和接口

```
1 // 步骤1：导入必要的类和接口
2 import org.dataalgorithms.util.SparkUtil;
3
4 import scala.Tuple2;
```

```
 5 import org.apache.spark.api.java.JavaRDD;
 6 import org.apache.spark.api.java.JavaPairRDD;
 7 import org.apache.spark.api.java.JavaSparkContext;
 8 import org.apache.spark.api.java.function.FlatMapFunction;
 9 import org.apache.spark.api.java.function.PairFunction;
10 import org.apache.spark.api.java.function.Function2;
11 import org.apache.spark.broadcast.Broadcast;
12
13 import java.util.List;
14 import java.util.Map;
15 import java.util.TreeMap;
16 import java.util.SortedMap;
17 import java.util.Iterator;
18 import java.util.Collections;
```

## 步骤2：处理输入参数

示例3-19展示了第2步，我们要在这一步处理输入参数。

示例3-19：步骤2：处理输入参数

```
1    // 步骤2：处理输入参数
2    if (args.length < 2) {
3       System.err.println("Usage: Top10NonUnique <input-path> <topN>");
4       System.exit(1);
5    }
6    System.out.println("args[0]: <input-path>="+args[0]);
7    System.out.println("args[1]: <topN>="+args[1]);
8    final String inputPath = args[0];
9    final int N = Integer.parseInt(args[1]);
```

## 步骤3：创建一个Java Spark上下文对象

这一步如示例3-20所示，我们要创建一个Java Spark上下文对象。

示例3-20：步骤3：创建一个Java Spark上下文对象

```
1 // 步骤3：创建一个Java Spark上下文对象
2 JavaSparkContext ctx = SparkUtil.createJavaSparkContext("Top10NonUnique");
```

## 步骤4：将topN广播到所有集群节点

要在所有集群节点间广播或共享对象和数据结构，可以使用Spark的`Broadcast`类（参见示例3-21）。

示例3-21：步骤4：将topN广播到所有集群节点

```
1 // 步骤4：将topN广播到所有集群节点
2 final Broadcast<Integer> topN = ctx.broadcast(N);
3 // 现在可以从所有集群节点读取topN
```

## 步骤5：从输入创建一个RDD

在示例3-22中，将从HDFS读取输入数据，并创建第一个RDD。

示例3-22：步骤5：从输入创建一个RDD

```
1 //  步骤5：从输入创建一个RDD
2 //     输入记录格式：
3 //        <string-key><,><integer-value-count>
4 JavaRDD<String> lines = ctx.textFile(inputPath, 1);
5 lines.saveAsTextFile("/output/1");
```

为了调试第5步，下面打印出这个RDD：

```
# hadoop fs -cat /output/1/part*
A,2
B,2
C,3
D,2
E,1
G,2
A,1
B,1
C,3
E,1
F,1
G,2
A,2
B,2
C,1
D,2
E,1
F,1
G,2
```

## 步骤6：RDD分区

RDD分区如示例3-23所示，这里需要艺术和科学的完美结合。需要有多少个分区？计算分区数时，并没有万全之策。这要取决于集群节点数、每个服务器的内核数及可用的RAM大小。我的经验是，要通过不断尝试和校正来设置适当的分区数。一般经验是每个执行器使用（2 × num_executors × cores_per_executor）个分区。

示例3-23：步骤6：RDD分区

```
1    // 步骤6：RDD分区
2    // public JavaRDD<T> coalesce(int numPartitions)
3    // 返回一个新的RDD，归约到numPartitions个分区。
4    JavaRDD<String> rdd = lines.coalesce(9);
```

## 步骤7：将输入（T）映射到（K,V）对

这一步完成基本映射，如示例3-24所示。它将各个输入记录转换为一个（K, V）对，这里*K*是键（如URL），*V*是值（如访问数）。这一步可能会生成重复的键。

示例3-24：步骤7：将输入(T)映射到(K,V)对

```
// 步骤7：将输入(T)映射到（K,V）对
// PairFunction<T, K, V>
// T => Tuple2<K, V>
JavaPairRDD<String,Integer> kv =
   rdd.mapToPair(new PairFunction<String,String,Integer>() {
   public Tuple2<String,Integer> call(String s) {
      String[] tokens = s.split(","); // url,789
      return new Tuple2<String,Integer>(tokens[0],
                                        Integer.parseInt(tokens[1]));
   }
});
kv.saveAsTextFile("/output/2");
```

这一步会生成以下输出：

```
# hadoop fs -cat /output/2/part*
(A,2)
(B,2)
(C,3)
(D,2)
(E,1)
(G,2)
(A,1)
(B,1)
(C,3)
(E,1)
(F,1)
(G,2)
(A,2)
(B,2)
(C,1)
(D,2)
(E,1)
(F,1)
(G,2)
```

## 步骤8：归约重复键

第7步可能生成重复的键，而这一步将创建唯一的键，并聚集相关的值。例如，如果对应一个键K有以下数据：

$$\{(K, V_1), (K, V_2), ..., (K, V_n)\}$$

这些数据会聚集/归约为一个（$K$, $V$），其中：

$$V = (V_1 + V_2 + ... + V_n)$$

在示例3-25中，归约器由JavaPairRDD.reduceByKey()实现。

示例3-25：步骤8：归约重复键

```
// 步骤8：归约重复K
JavaPairRDD<String, Integer> uniqueKeys =
    kv.reduceByKey(new Function2<Integer, Integer, Integer>() {
    public Integer call(Integer i1, Integer i2) {
        return i1 + i2;
    }
});
uniqueKeys.saveAsTextFile("/output/3");
```

这一步生成的输出如下：

```
# hadoop fs -cat /output/3/part*
(B,5)
(E,3)
(C,7)
(F,2)
(G,6)
(A,5)
(D,4)
```

## 步骤9：创建本地top N

这一步（参见示例3-26）的目标是为各个分区创建一个本地top 10。

示例3-26：步骤9：创建本地top N

```
// 步骤9：创建本地top N
JavaRDD<SortedMap<Integer, String>> partitions = uniqueKeys.mapPartitions(
   new FlatMapFunction<
                        Iterator<Tuple2<String,Integer>>,
                        SortedMap<Integer, String>
                       >() {
   @Override
   public Iterable<SortedMap<Integer, String>>
      call(Iterator<Tuple2<String,Integer>> iter) {
      final int N = topN.value();
      SortedMap<Integer, String> localTopN = new TreeMap<Integer, String>();
      while (iter.hasNext()) {
         Tuple2<String,Integer> tuple = iter.next();
         localTopN.put(tuple._2, tuple._1);
         // 只保留top N
         if (localTopN.size() > N) {
            // 删除频度最小的元素
            localTopN.remove(localTopN.firstKey());
         }
       }
       return Collections.singletonList(localTopN);
   }
});
partitions.saveAsTextFile("/output/4");
```

这一步会生成以下输出：

```
# hadoop fs -ls /output/4/
Found 4 items
-rw-r--r-- 3 hadoop hadoop 0 2014-08-31 13:11 /output/4/_SUCCESS
-rw-r--r-- 3 hadoop hadoop 11 2014-08-31 13:11 /output/4/part-00000
-rw-r--r-- 3 hadoop hadoop 11 2014-08-31 13:11 /output/4/part-00001
-rw-r--r-- 3 hadoop hadoop 11 2014-08-31 13:11 /output/4/part-00002

# hadoop fs -cat /output/4/part-00000
{3=E, 5=B}

# hadoop fs -cat /output/4/part-00001
{2=F, 7=C}

# hadoop fs -cat /output/4/part-00002
{5=A, 6=G}

# hadoop fs -cat /output/4/part*
{3=E, 5=B}
{2=F, 7=C}
{5=A, 6=G}
```

## 步骤10：查找最终top N

这一步（参见示例3-27）得到所有本地top N列表并生成最终的top N列表。

示例3-27：步骤10：查找最终top N

```
1    // 步骤10：查找最终top N
2    SortedMap<Integer, String> finalTopN = new TreeMap<Integer, String>();
3    List<SortedMap<Integer, String>> allTopN = partitions.collect();
4    for (SortedMap<Integer, String> localTopN : allTopN) {
5        for (Map.Entry<Integer, String> entry : localTopN.entrySet()) {
6            // count = entry.getKey()
7            // url = entry.getValue()
8            finalTopN.put(entry.getKey(), entry.getValue());
9            // 只保留top N
10           if (finalTopN.size() > N) {
11               finalTopN.remove(finalTopN.firstKey());
12           }
13       }
14   }
```

你也可以使用`JavaRDD.reduce()`操作实现第10步（完整的解决方案参见本章前面的小节）。

## 步骤11：发出最终top N

这一步会在标准输出设备上发出最终的输出，如示例3-28所示。

示例3-28：步骤11：发出最终top N

```
1 // 步骤11：发出最终top N
2 System.out.println("--- Top-N List ---");
3 System.out.println("------------------");
```

```
4 for (Map.Entry<Integer, String> entry : finalTopN.entrySet()) {
5     System.out.println("key="+ entry.getValue()+ " value = " +entry.getKey());
6 }
```

得到的最终top *N*列表如下所示：

```
--- Top-N List ---
------------------
key=G value=6
key=C value=7
```

# 运行示例

## 脚本

```
 1 # cat ./run_top10_nonunique.sh
 2 export JAVA_HOME=/usr/java/jdk7
 3 export SPARK_HOME=/usr/local/spark-1.1.0
 4 export SPARK_MASTER=spark://myserver100:7077
 5 BOOK_HOME=/mp/data-algorithms-book
 6 APP_JAR=$BOOK_HOME/dist/data_algorithms_book.jar
 7 INPUT=/top10/top10input
 8 TOPN=4
 9 prog=org.dataalgorithms.chap03.spark.Top10NonUnique
10 # 在Spark独立集群上运行
11 $SPARK_HOME/bin/spark-submit \
12   --class $prog \
13   --master $SPARK_MASTER \
14   --executor-memory 2G \
15   --total-executor-cores 20 \
16     $APP_JAR $INPUT $TOPN
```

## 使用Spark集群运行Top 10示例

这里给出示例运行的输出，为适应版面这里做了删减和编辑。运行这个示例的Spark集群包括3个服务器节点： myserver100、myserver200和myserver300。

```
 1 # ./run_top10_nonunique.sh
 2 args[0]: <input-path>=/top10/top10input
 3 args[1]: <topN>=2
 4 ...
 5 INFO : Total input paths to process : 3
 6 INFO : Starting job: saveAsTextFile at Top10NonUnique.java:73
 7 INFO : Got job 0 (saveAsTextFile at Top10NonUnique.java:73) with
 8 ...
 9 INFO : Completed ResultTask(4, 0)
10 INFO : Finished TID 12 in 244 ms on myserver100 (progress: 1/3)
11 INFO : Completed ResultTask(4, 2)
12 INFO : Finished TID 14 in 304 ms on myserver200 (progress: 2/3)
13 INFO : Completed ResultTask(4, 1)
14 INFO : Finished TID 13 in 362 ms on myserver300 (progress: 3/3)
15 ...
```

```
16 INFO : Adding task set 6.0 with 3 tasks
17 ...
18 INFO : Job finished: collect at Top10NonUnique.java:121, took 0.076470196 s
19 --- Top-N List ---
19 ------------------
20 key=G value=6
21 key=C value=7
```

# 使用takeOrdered()的Spark Top 10解决方案

这个解决方案与Top10NonUnique.java对应，这里使用了Spark的一个非常强大的特性：takeOrdered()函数。这个函数的签名如下：

```
java.util.List<T> takeOrdered(int N,
                              java.util.Comparator<T> comp)
Description:
   Returns the first N elements from this RDD as defined
   by the specified Comparator[T] and maintains the order.

Parameters:
   N - the number of top elements to return
   comp - the comparator that defines the order

Returns:
   an array of top N elements
```

找到所有唯一键之后（我们称这个RDD为uniqueKeys），通过执行以下代码找到top N：

```
JavaPairRDD<String, Integer> uniqueKeys = kv.reduceByKey(...);
List<JavaPairRDD<String, Integer>>
    uniqueKeys.takeOrdered(N, MyTupleComparator.INSTANCE);
```

这里MyTupleComparator.INSTANCE是比较器，定义了（$K$, $V$）对的顺序。MyTuple-Comparator类定义如下：

```
static class MyTupleComparator implements
   Comparator<Tuple2<String, Integer>> ,java.io.Serializable {
   final static MyTupleComparator INSTANCE = new MyTupleComparator();
   // 需要指出，这里将根据键的频度完成比较
   public int compare(Tuple2<String, Integer> t1,
                      Tuple2<String, Integer> t2) {
      return -t1._2.compareTo(t2._2); // 返回top N
      // return -t1._2.compareTo(t2._2); // 返回bottom N
   }
}
```

需要说明，比较器类MyTupleComparator必须实现java.io.Serializable接口；否则，Spark会抛出以下异常：

```
Caused by: org.apache.spark.SparkException: Task not serializable
at org.apache.spark.util.ClosureCleaner$.ensureSerializable
```

```
    (ClosureCleaner.scala:166)
```

## 完整的Spark实现

示例3-29给出了完整的Spark top 10解决方案。

示例3-29：对应非唯一键的完整Spark Top 10程序

```
1 package org.dataalgorithms.chap03;
2 // 步骤1：导入必要的类和接口
3 /**
4  * 假设：对于所有输入(K, V)，K是不唯一的。
5  * 这个类实现了Top N设计模式（N > 0）。
6  * 主要假设为对于所有输入（K, V）对，K
7  * 非唯一。这说明，我们可能会看找到类似
8  * (A, 2), ..., (A, 5),.. 的项。
9  *
10 * 这是一个通用的Top N算法，适用于唯一键以及
11 * 非唯一键。
12 *
13 * 这个类也可以用来找出bottom N
14 * （只需要保留集合中最小的N个元素）。
15 *
16 * Top 10 设计模式：Top 10结构
17 *
18 * 1. 映射(输入) => (K, V)
19 *
20 * 2. 归约(K, List<V1, V2, ..., Vn>) => (K, V),
21 * 其中V = V1+V2+...+Vn;
22 * 现在所有K都是唯一的。
23 *
24 * 3. 使用以下高层Spark API找出top N:
25 * java.util.List<T> takeOrdered(int N, java.util.Comparator<T> comp)
26 * 根据指定Comparator[T]的定义从这个RDD返回前N个元素，
27 * 并维护顺序。
28 *
29 * @author Mahmoud Parsian
30 *
31 */
32 public class Top10UsingTakeOrdered implements Serializable {
33    public static void main(String[] args) throws Exception {
34       // 步骤2：处理输入参数
35       // 步骤3：创建一个Java Spark上下文对象
36       // 步骤4：从输入创建一个RDD
37       // 步骤5：RDD分区
38       // 步骤6：输入（T）映射到（K,V）对
39       // 步骤7：归约重复的K
40       // 步骤8：调用takeOrdered()查找最终top N
41       // 步骤9：发出最终top N
42       System.exit(0);
43    }
44 }
```

## 输入

这个例子的输入与上一节完全相同（见上一节“输入”小节）。

## 步骤1：导入必要的类和接口

第1步如示例3-30所示，要导入这个解决方案需要的所有类和接口。

示例3-30：步骤1：导入必要的类和接口

```
// 步骤1：导入必要的类和接口
import org.dataalgorithms.util.SparkUtil;

import scala.Tuple2;
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.api.java.JavaPairRDD;
import org.apache.spark.api.java.JavaSparkContext;
import org.apache.spark.api.java.function.FlatMapFunction;
import org.apache.spark.api.java.function.PairFunction;
import org.apache.spark.api.java.function.Function2;

import java.util.List;
import java.util.Map;
import java.util.TreeMap;
import java.util.SortedMap;
import java.util.Iterator;
import java.util.Collections;
import java.util.Comparator;
import java.io.Serializable;
```

## 步骤2：处理输入参数

示例3-31将处理Spark驱动器程序的输入参数。

示例3-31：步骤2：处理输入参数

```
    // 步骤2：处理输入参数
    if (args.length < 2) {
       System.err.println("Usage: Top10UsingTakeOrdered <input-path> <topN>");
       System.exit(1);
    }
    System.out.println("args[0]: <input-path>="+args[0]);
    System.out.println("args[1]: <topN>="+args[1]);
    final String inputPath = args[0];
    final int N = Integer.parseInt(args[1]);
```

## 步骤3：创建一个Java Spark上下文对象

下一步中（参见示例3-32），我们要创建一个Java Spark上下文对象。

示例3-32：步骤3：创建一个Java Spark上下文对象

```
    // 步骤3：创建一个Java Spark上下文对象
    JavaSparkContext ctx = SparkUtil.createJavaSparkContext("top-10");
```

## 步骤4：从输入创建一个RDD

在示例3-33中，从HDFS（或其他持久存储文件系统，如Linux NFS文件系统）读取输入数据，并创建第一个RDD。

示例3-33：步骤4：从输入创建一个RDD

```
1    // 步骤4：从输入创建一个RDD
2    //     输入记录格式：
3    //         <string-key><,><integer-value-count>
4 JavaRDD<String> lines = ctx.textFile(inputPath, 1);
5 lines.saveAsTextFile("/output/1");
```

为了调试第4步，下面打印出第一个RDD：

```
# hadoop fs -cat /output/1/part*
A,2
B,2
C,3
D,2
E,1
G,2
A,1
B,1
C,3
E,1
F,1
G,2
A,2
B,2
C,1
D,2
E,1
F,1
G,2
```

## 步骤5：RDD分区

RDD分区如示例3-34所示，这里需要艺术和科学的完美结合。需要有多少个分区？计算分区数时，并没有万全之策。这要取决于集群节点数、每个服务器的内核数以及可用的RAM大小。我的经验是，要通过不断尝试和校正来确定适当的分区数。

示例3-34：步骤5：RDD分区

```
1    // 步骤5：RDD分区
2    // public JavaRDD<T> coalesce(int numPartitions)
3    // 返回一个新的RDD，归约到numPartitions个分区
4    JavaRDD<String> rdd = lines.coalesce(9);
```

## 步骤6：将输入（T）映射到（K,V）对

这一步将完成基本映射，如示例3-35所示。它将各个输入记录转换为一个（K, V）对，这里*K*是键（如URL），*V*是值（如访问数）。这一步可能生成重复的键。

示例3-35：步骤6：将输入（T）映射到（K,V）对

```
// 步骤6：输入(T)映射到（K,V）对
// PairFunction<T, K, V>
// T => Tuple2<K, V>
JavaPairRDD<String,Integer> kv =
   rdd.mapToPair(new PairFunction<String,String,Integer>() {
   public Tuple2<String,Integer> call(String s) {
       String[] tokens = s.split(","); // url,789
       return new Tuple2<String,Integer>(tokens[0],
                                         Integer.parseInt(tokens[1]));
   }
});
kv.saveAsTextFile("/output/2");
```

这一步生成的输出如下所示：

```
# hadoop fs -cat /output/2/part*
(A,2)
(B,2)
(C,3)
(D,2)
(E,1)
(G,2)
(A,1)
(B,1)
(C,3)
(E,1)
(F,1)
(G,2)
(A,2)
(B,2)
(C,1)
(D,2)
(E,1)
(F,1)
(G,2)
```

## 步骤7：归约重复键

第6步可能会生成重复的键，而这一步将创建唯一的键，并聚集相关的值。例如，如果对应一个键K有以下数据：

$$\{(K, V_1), (K, V_2), \ldots, (K, V_n)\}$$

这些数据则会聚集/归约为一个（*K*, *V*），其中：

$$V = (V_1 + V_2 + ... + V_n)$$

在示例3-36中，归约器由JavaPairRDD.reduceByKey()实现。

示例3-36：步骤7：归约重复键

```
1    // 步骤7：归约重复K
2    JavaPairRDD<String, Integer> uniqueKeys =
3        kv.reduceByKey(new Function2<Integer, Integer, Integer>() {
4        public Integer call(Integer i1, Integer i2) {
5           return i1 + i2;
6        }
7    });
8 uniqueKeys.saveAsTextFile("/output/3");
```

这一步会生成以下输出：

```
# hadoop fs -cat /output/3/part*
(B,5)
(E,3)
(C,7)
(F,2)
(G,6)
(A,5)
(D,4)
```

## 步骤8：调用takeOrdered()查找最终top *N*

这一步使用一个非常强大的Spark函数takeOrdered()来查找一个给定（*K*,*V*）对列表中的top *N*。takeOrdered()函数会对给定RDD排序，并返回top *N*元素，如示例3-37所示。

示例3-37：步骤8：调用takeOrdered()查找最终top *N*

```
1    // 步骤8：调用takeOrdered()查找最终top N
2    List<Tuple2<String, Integer>> topNResult =
3       uniqueKeys.takeOrdered(N, MyTupleComparator.INSTANCE);
```

## 步骤9：发出最终的top *N*

这一步会发出最终的top *N*，如示例3-38所示。

示例3-38：步骤9：发出最终的top *N*

```
1    // 步骤9：发出最终top N
2    for (Tuple2<String, Integer> entry : topNResult) {
3        System.out.println(entry._2 + "--" + entry._1);
4 }
```

这一步的输出如下所示：

```
7--C
6--G
```

```
5--A
5--B
```

### 运行Spark程序的Shell脚本

下面是运行这个Spark程序的Shell脚本：

```
# cat run_top10usingtakeordered_yarn.sh
#!/bin/bash
export MP=/home/hadoop/testspark
export THE_SPARK_JAR=$MP/spark-assembly-1.1.0-hadoop2.5.0.jar
export JAVA_HOME=/usr/java/jdk7
export APP_JAR=$MP/mp.jar
export SPARK_HOME=/home/hadoop/spark-1.1.0
#
export HADOOP_HOME=/usr/local/hadoop/hadoop-2.5.0
export HADOOP_CONF_DIR=$HADOOP_HOME/etc/hadoop
INPUT=/top10/top10input
TOPN=4
DRIVER=org.dataalgorithms.chap03.spark.Top10UsingTakeOrdered
$SPARK_HOME/bin/spark-submit --class $DRIVER \
    --master yarn-cluster \
    --num-executors 12 \
    --driver-memory 3g \
    --executor-memory 7g \
    --executor-cores 12 \
    --conf "spark.yarn.jar=$THE_SPARK_JAR" \
    $APP_JAR $INPUT $TOPN
```

## 查找Bottom *N*

如果想生成bottom *N*而不是top *N*该怎么做呢？可以在比较器类中实现：只需要将compare()函数的结果取负。

示例3-39使用MyTupleComparatorAscending查找bottom *N*元素。

示例3-39：元组比较器：升序

```
static class MyTupleComparatorAscending implements
   Comparator<Tuple2<String, Integer>> ,java.io.Serializable {
   final static MyTupleComparatorAscending INSTANCE =
     new MyTupleComparatorAscending();
   // 需要指出，要根据键的频度完成比较
   public int compare(Tuple2<String, Integer> t1,
                      Tuple2<String, Integer> t2) {
      // 升序排序，返回bottom N
      return t1._2.compareTo(t2._2);
   }
}
```

示例3-40使用MyTupleComparatorDescending查找top *N*元素。

示例3-40：元组比较器：降序

```
 1   static class MyTupleComparatorDescending implements
 2      Comparator<Tuple2<String, Integer>> ,java.io.Serializable {
 3      final static MyTupleComparatorDescending INSTANCE =
 4        new MyTupleComparatorDescending();
 5      // 需要指出，要根据键的频度完成比较
 6      public int compare(Tuple2<String, Integer> t1,
 7                         Tuple2<String, Integer> t2) {
 8          // 降序排序，返回top N
 9          return -t1._2.compareTo(t2._2);
10      }
11 }
```

最终的输出如下：

```
2--F
3--E
4--D
5--A
```

## 使用takeOrdered()的替代方案

使用takeOrdered()函数的另一种替代方案是使用Spark的top()函数，参见示例3-41中的定义。

示例3-41：使用takeOrdered()的替代方案

```
1 import java.util.List;
2 import java.util.Comparator;
3 ...
4 List<T> top(int N, Comparator<T> comp)
5 // 根据指定Comparator[T]的定义，
6 // 从这个RDD返回Ton N元素。
7 // 参数:
8 //   N－返回前N个元素
9 //   comp－定义排序顺序的比较器
```

# MapReduce/Hadoop Top 10解决方案：非唯一键

这是Top N设计模式的最后一个解决方案。这里假设所有输入键是不唯一的，也就是说，对于一个给定的输入（*K*, *V*），所有K不是唯一的。例如，输入中可能包含（*K*, *2*）、（*K*, *3*）和（*K*, *36*）（这些记录有相同的键）。

这个MapReduce/Hadoop top *N*解决方案分为两个阶段。*N*需要参数化，也就是说可以作为参数，从而可以查找top 5、top 20或top 100。



这一节的所有Java类都已在GitHub（*http://bit.ly/da_book*）上提供。

这两个阶段如下：

*阶段1*

将不唯一的键转换为唯一的键。例如，如果我们的输入包含键-值对（*K*, *2*）、（*K*, *3*）和（*K*, *36*），就将所有K聚集为一个键-值对：（*K*, *41*），这里41 = 2+3+36。这个阶段由3个类实现：

- `AggregateByKeyDriver`
- `AggregateByKeyMapper`
- `AggregateByKeyReducer`

在这个阶段中，由于映射器聚集之后的输出（也就是累加频度值）会构成一个幺半群（monoid），我们还会利用组合器。这个工作在驱动器类（`AggregateByKeyDriver`）的`run()`方法中实现：

```
...
job.setMapperClass(AggregateByKeyMapper.class);
job.setReducerClass(AggregateByKeyReducer.class);
job.setCombinerClass(AggregateByKeyReducer.class);
...
```

*阶段2*

这个阶段的输入是阶段1生成的唯一键-值对。在这个阶段中，每个映射器创建一个本地top N列表，所有这些top N列表都将传递到一个归约器。在这里，这个归约器并不会成为一个瓶颈，因为每个映射器只生成很少量的数据（它的本地top *N*列表）。例如，如果我们有1000个映射器，每个映射器生成一个本地top *N*，那么归约器只需要处理（1000 × *N*）的数据，如果*N*为5、20或100，这个数据量并不大。这个阶段由以下3个类实现：

- `TopNDriver`
- `TopNMapper`
- `TopNReducer`

## 运行示例

下面各小节将展示一个运行示例，这里使用`TopNDriver`作为驱动器。为适应版面这里对日志输出做了删减，并对格式有所调整。

## 输入

```
$ hadoop fs -ls /kv/input/
Found 3 items
-rw-r--r-- 3 hadoop haoop 42 2014-11-30 15:18 /kv/input/file1.txt
-rw-r--r-- 3 hadoop haoop 33 2014-11-30 15:18 /kv/input/file2.txt
-rw-r--r-- 3 hadoop haoop 28 2014-11-30 15:18 /kv/input/file3.txt

$ hadoop fs -cat /kv/input/file1.txt
A,2
B,2
C,3
D,2
E,1
G,2
A,3
B,4
Z,100
Z,1

$ hadoop fs -cat /kv/input/file2.txt
A,1
B,1
C,3
E,1
F,1
G,2
A,65
A,3

$ hadoop fs -cat /kv/input/file3.txt
A,2
B,2
C,1
D,2
E,1
F,1
G,2
```

## 脚本

```
$ cat ./run_top_N_mapreduce.sh
#!/bin/bash
export HADOOP_HOME=/usr/local/hadoop-2.5.0
export PATH=$HADOOP_HOME/bin:$PATH
export BOOK_HOME=/mp/data-algorithms-book
export APP_JAR=$BOOK_HOME/dist/data_algorithms_book.jar
#
# Generate unique (K,V) pairs
#
INPUT=/kv/input
OUTPUT=/kv/output
hadoop fs -rmr $OUTPUT
AGGREGATOR=org.dataalgorithms.chap03.mapreduce.AggregateByKeyDriver
hadoop jar $APP_JAR $AGGREGATOR $INPUT $OUTPUT
```

```
#
# Find Top N
#
N=5
TopN_INPUT=/kv/output
TopN_OUTPUT=/kv/final
hadoop fs -rmr $TopN_OUTPUT
TopN=org.dataalgorithms.chap03.mapreduce.TopNDriver
hadoop jar $APP_JAR $TopN $N $TopN_INPUT $TopN_OUTPUT
```

## 示例运行日志

```
$ ./run_top_N_mapreduce.sh
...
14/11/30 16:08:04 INFO mapreduce.AggregateByKeyDriver: inputDir=/kv/input
14/11/30 16:08:04 INFO mapreduce.AggregateByKeyDriver: outputDir=/kv/output
...
14/11/30 16:08:07 INFO mapreduce.Job: Running job: job_1416535300416_0017
14/11/30 16:08:14 INFO mapreduce.Job: map 0% reduce 0%
...
14/11/30 16:08:31 INFO mapreduce.Job: map 100% reduce 100%
14/11/30 16:08:32 INFO mapreduce.Job: Job job_1416535300416_0017
  completed successfully
...
14/11/30 16:08:32 INFO mapreduce.AggregateByKeyDriver: run(): status=true
...
14/11/30 16:08:35 INFO mapreduce.TopNDriver: N=5
14/11/30 16:08:35 INFO mapreduce.TopNDriver: inputDir=/kv/output
14/11/30 16:08:35 INFO mapreduce.TopNDriver: outputDir=/kv/final
...
14/11/30 16:08:39 INFO mapreduce.Job: Running job: job_1416535300416_0018
14/11/30 16:08:45 INFO mapreduce.Job: map 0% reduce 0%
...
14/11/30 16:09:28 INFO mapreduce.Job: map 100% reduce 100%
14/11/30 16:09:28 INFO mapreduce.Job: Job job_1416535300416_0018
  completed successfully
...
14/11/30 16:09:29 INFO mapreduce.TopNDriver: run(): status=true
```

## 中间输出

```
$ hadoop fs -text /kv/output/part*
A 76
B 9
Z 101
C 7
D 4
E 3
F 2
G 6
```

## 最终top N输出

```
$ hadoop fs -text /kv/final/part*
```

```
101 Z
76 A
9 B
7 C
6 G
```

这一章分别使用MapReduce/Hadoop和Spark实现了一个非常重要的过滤分类设计模式Top N。下一章会提供另一个重要设计模式（称为“左外连接”）的MapReduce解决方案，这种设计模式常用于分析商业交易。

# 第4章

# 左外连接

这一章将介绍如何在MapReduce环境中实现左外连接（left outer join）。这里会提供MapReduce/Hadoop和Spark中的3个不同实现：

- MapReduce/Hadoop解决方案：使用传统`map()`和`reduce()`函数。
- Spark解决方案：不使用内置`JavaPairRDD.leftOuterJoin()`。
- Spark解决方案：使用内置`JavaPairRDD.leftOuterJoin()`。

## 左外连接示例

考虑一家公司，比如Amazon，它拥有超过2亿的用户，每天要完成数亿次交易。为了理解左外连接的概念，假设我们有两类数据：用户和交易。用户数据包括用户的地址信息（例如`location_id`），交易数据包括用户身份信息（例如`user_id`），但是不包括用户地址的直接信息。给定`users`和`transactions`，如下：

```
users(user_id, location_id)
transactions(transaction_id, product_id, user_id, quantity, amount)
```

我们的目标是得出每个售出商品对应的唯一用户地址数。

不过，到底什么是左外连接？令$T_1$（左表）和$T_2$（右表）是以下两个关系（其中$t_1$是$T_1$的属性，$t_2$是$T_2$的属性）：

$$T_1 = (K, t_1)$$
$$T_2 = (K, t_2)$$

关系$T_1$和$T_2$在连接键K上左外连接的结果将包含左表（$T_1$）的所有记录，即使连接条件在

右表（$T_2$）中未找到任何匹配的记录。如果关于键K的ON子句匹配$T_2$中的0条记录（对于$T_1$中的一个给定记录），这个连接仍会在结果中返回一行（对应这条记录），不过$T_2$的各个列都为NULL。左外连接会返回内连接的所有值以及左表中未与右表匹配的所有值。可以形式化表示为：

$$\text{LeftOuterJoin}(T_1, T_2, K) = \{(k, t_1, t_2) \text{ where } k \in T_1.k \text{ and } k \in T_2.k\} \cup \{(k, t_1, null) \text{ where } k\ T_1 \in K \text{ and } kT_2.k\}$$

在SQL中，可以将左外连接表示如下（在这里，$T_1$和$T_2$在K列上连接）：

```
SELECT field_1, field_2, ...
   FROM T1 LEFT OUTER JOIN T2
      ON T1.K = T2.K;
```

左外连接如图4-1所示（连接结果包含有颜色的部分，而不包含白色的部分）。

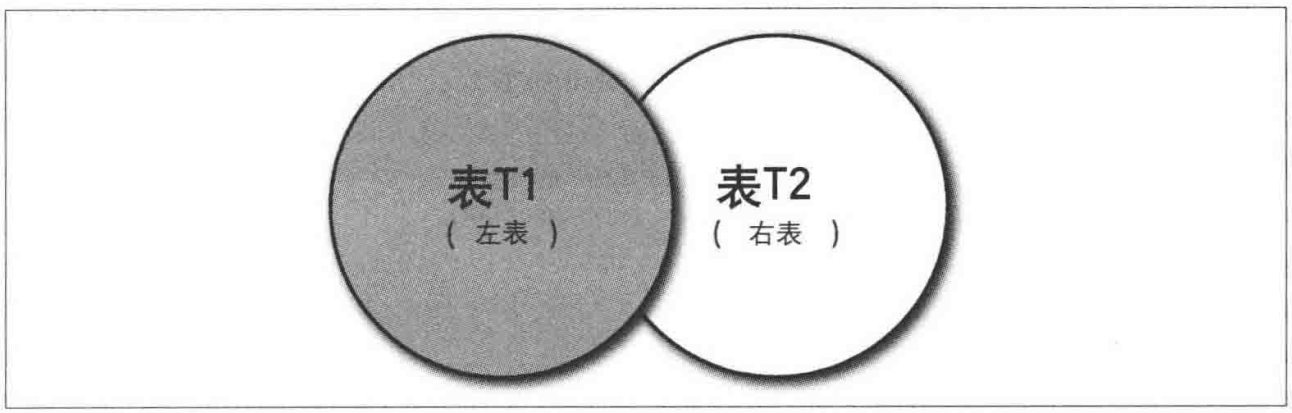



图4-1：左外连接

下面来看表4-1和表4-2的值，它们分别对应用户和交易信息（需要说明，这些值只是一些例子，用来展示MapReduce环境中左外连接的概念）。

表4-1：左外连接中的用户信息

| user_id | location_id |
|---|---|
| u1 | UT |
| u2 | GA |
| u3 | CA |
| u4 | CA |
| u5 | GA |

表4-2：左外连接中的交易信息

| transaction_id | product_id | user_id | quantity | amount |
|---|---|---|---|---|
| t1 | p3 | u1 | 1 | 300 |
| t2 | p1 | u2 | 1 | 100 |
| t3 | p1 | u1 | 1 | 100 |
| t4 | p2 | u2 | 1 | 10 |
| t5 | p4 | u4 | 1 | 9 |
| t6 | p1 | u1 | 1 | 100 |
| t7 | p4 | u1 | 1 | 9 |
| t8 | p4 | u5 | 2 | 40 |

## 示例查询

下面是与这个左外连接有关的一些SQL查询示例：

- 查询1：找出已售出的所有商品（及关联的用户位置）：

```
mysql> SELECT product_id, location_id
    -> FROM transactions LEFT OUTER JOIN users
    ->   ON transactions.user_id = users.user_id;
+------------+-------------+
| product_id | location_id |
+------------+-------------+
| p3         | UT          |
| p1         | GA          |
| p1         | UT          |
| p2         | GA          |
| p4         | CA          |
| p1         | UT          |
| p4         | UT          |
| p4         | GA          |
+------------+-------------+
8 rows in set (0.00 sec)
```

- 查询2：找出已售出的所有商品（及关联的用户位置数）：

```
mysql> SELECT product_id, count(location_id)
    -> FROM transactions LEFT OUTER JOIN users
    ->   ON transactions.user_id = users.user_id
    ->   group by product_id;
+------------+--------------------+
| product_id | count(location_id) |
+------------+--------------------+
| p1         |                  3 |
| p2         |                  1 |
| p3         |                  1 |
| p4         |                  3 |
+------------+--------------------+
4 rows in set (0.00 sec)
```

- 查询3：找出已售出的所有商品（及唯一位置数）：

```
mysql> SELECT product_id, count(distinct location_id)
    -> FROM transactions LEFT OUTER JOIN users
    -> ON transactions.user_id = users.user_id
    -> group by product_id;
+------------+----------------------------+
| product_id | count(distinct location_id) |
+------------+----------------------------+
| p1         |                          2 |
| p2         |                          1 |
| p3         |                          1 |
| p4         |                          3 |
+------------+----------------------------+
4 rows in set (0.00 sec)
```

# MapReduce左外连接实现

上一节的SQL查询3可以提供我们所要的输出，它会找出所有交易中各个售出商品对应的不同（唯一）用户地址。我们将分两个阶段提供左外连接问题的解决方案：

- MapReduce阶段1：找出所有售出的商品（以及关联的地址）。可以使用上一节中的SQL查询1完成这个任务。
- MapReduce阶段2：找出所有售出的商品（以及关联的唯一地址数）。可以使用上一节中的SQL查询3来实现。

## MapReduce阶段1：找出商品地址

这个阶段将用一个MapReduce作业完成左外连接操作，这里会利用两个映射器（一个对应用户，另一个对应交易），这个作业的归约器会发出一个键-值对，键为`product_id`，值为`location_id`。利用`MultipleInputs`类可以使用多个映射器（需要说明，如果只有一个映射器，那么可以使用`Job.setMapperClass()`）：

```
import org.apache.hadoop.mapreduce.lib.input.MultipleInputs;
...
Job job = new Job(...);
...
Path transactions = <hdfs-path-to-transactions-data>;
Path users = <hdfs-path-to-users-data>;

MultipleInputs.addInputPath(job,
                            transactions,
                            TextInputFormat.class,
                            TransactionMapper.class);
MultipleInputs.addInputPath(job,
                            users,
                            TextInputFormat.class,
                            UserMapper.class);
```

图4-2和4-3展示了左外连接算法的MapReduce工作流，其中包括两个MapReduce作业（阶段1和阶段2）。

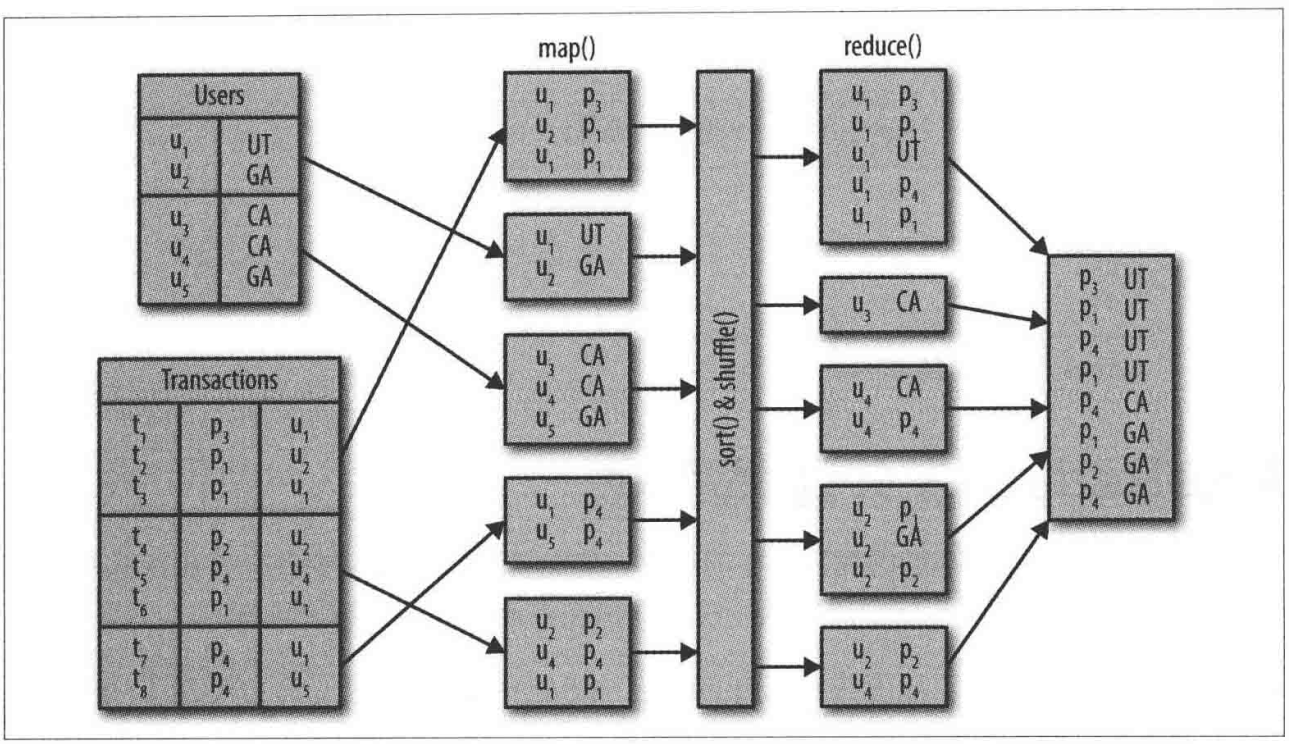



图4-2：左外连接数据流，阶段1

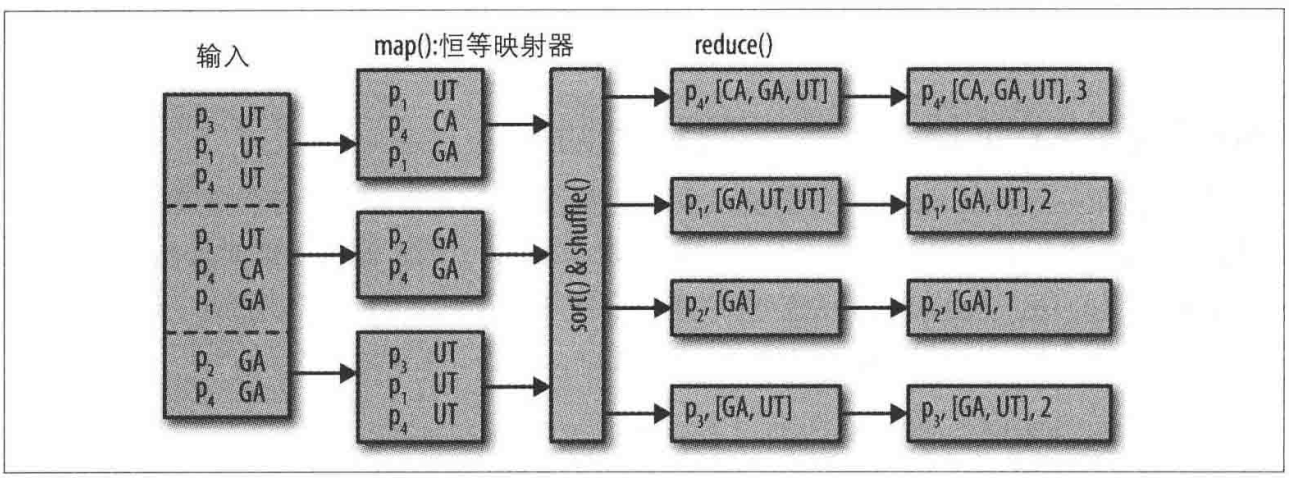



图4-3：左外连接数据流，阶段2

左外连接数据流的核心部分如下：

*交易映射器*

交易map()读取（transaction_id, product_id, user_id, quantity, amount），发出一个键-值对（user_id, product_id）。

*用户映射器*

用户map()读取（user_id, location_id），发出一个键-值对（user_id, location_id）。

阶段1的归约器得到用户的location_id和product_id，发出（product_id, location_id）。现在的问题是这个归约器如何区分location_id和product_id。Hadoop中并未定义归约器值的顺序。因此，对应一个特定键（user_id）的归约器得不到如何处理值的任何线索。为了解决这个问题，我们要对交易和用户映射器/归约器做一些修改（这称为版本2）：

*交易映射器（版本2）*

如示例4-1所示，交易map()读取（transaction_id, product_id, user_id, quantity, amount），发出键对（user_id, 2）和值对（"P", product_id）。通过为归约器键增加一个“2”，可以保证product_id总是在末尾。这可以通过第1章和第2章介绍的二次排序技术来实现。我们为值增加了“P”来标识商品。在Hadoop中，要实现Pair(String, String)，我们将使用PairOfStrings类[注1]。

示例4-1：交易映射器（版本2）

```
1 /**
2  * @param key 由框架生成，在这里忽略
3  * @param value 为
4  * transaction_id<TAB>product_id<TAB>user_id<TAB>quantity<TAB>amount
5  */
6 map(key, value) {
7    String[] tokens = StringUtil.split(value, "\t");
8    String productID = tokens[1];
9    String userID = tokens[2];
10   outputKey = Pair(userID, 2);
11   outputValue = Pair("P", productID);
12   emit(outputKey, outputValue);
13 }
```

*用户映射器（版本2）*

如示例4-2所示，用户map()读取（user_id, location_id），发出键（user_id, 1）和值对（"L", location_id）。通过为归约器键增加一个“1”，可以保证location_id总会先到达。这可以通过第1章和第2章介绍的二次排序技术来实现。我们为值增加了“L”来标识地址。

示例4-2：用户映射器（版本2）

```
1 /**
2  * @param key 由框架生成，在这里忽略
3  * @param value 为user_id<TAB>location_id
```

注1：edu.umd.cloud9.io.pair.PairOfStrings（它实现了WritableComparable<PairOfStrings>）。

```
 4  */
 5 map(key, value) {
 6    String[] tokens = StringUtil.split(value, "\t");
 7    String userID = tokens[0];
 8    String locationID = tokens[1];
 9    outputKey = Pair(userID, 1); // 确保位置显示在商品之前
10    outputValue = Pair("L", locationID);
11    emit(outputKey, outputValue);
12 }
```

如示例4-3所示，阶段1归约器（版本2）会得到（"L", location_id）和（"P", product_id）对，发出一个键-值对（product_id,location_id）。需要说明，由于1 < 2，这意味着用户的location_id会先到达。

示例4-3：阶段1归约器（版本2）

```
 1 /**
 2  * @param key为user_id
 3  * @param values 为List<Pair<left, right>>，在这里
 4  * values = List<{
 5  *                     Pair<"L", locationID>,
 6  *                     Pair<"P", productID1>,
 7  *                     Pair<"P", productID2>,
 8  *                     ...
 9  *                   }
10  * 需要说明， Pair<"L", locationID>在所有商品对之前到达。
11  * 第一个值是位置；
12  * 如果不是，说明得到的不是一个用户记录，
13  * 所以要把locationID设置为"undefined"。
14  */
15 reduce(key, values) {
16    locationID = "undefined";
17    for (Pair<left, right> value: values) {
18        // 第一次迭代时，
19        // 下面的if-stmt将为true
20        if (value.left.equals("L")) {
21            locationID = value.right;
22            continue;
23        }
24
25        // 这里有一个商品： value.left.equals("P")
26        productID = value.right;
27        emit(productID, locationID);
28    }
29 }
```

## MapReduce阶段2：统计唯一地址

这个阶段将使用阶段1的输出（这是一个（product_id, location_id）对序列），生成（product_id, number_of_unique_locations）对。这个阶段的映射器是一个恒等映射器（参见示例4-4），归约器（使用一个Set数据结构）统计每个商品对应的唯一地址数（参见示例4-5）。

示例4-4：阶段2映射器：统计唯一地址

```
/**
 * @param key为product_id
 * @param value为location_id
 */
map(key, value) {
   emit(key, value);
}
```

示例4-5：阶段2归约器：统计唯一地址

```
/**
 * @param key为product_id
 * @param value为List<location_id>
 */
reduce(key, values) {
   Set<String> set = new HashSet<String>();
   for (String locationID : values) {
      set.add(locationID);
   }

   int uniqueLocationsCount = set.size();
   emit(key, uniqueLocationsCount);
}
```

## Hadoop中的实现类

利用表4-3中所示的类，可以使用MapReduce/Hadoop框架实现左外连接设计模式的两个阶段。

**表4-3：Hadoop中的实现类**

| 阶段 | 类名 | 类描述 |
|---|---|---|
| 阶段1 | LeftJoinDriver | 提交阶段1作业的驱动器 |
| | LeftJoinReducer | 左连接归约器 |
| | LeftJoinTransactionMapper | 左连接交易映射器 |
| | LeftJoinUserMapper | 左连接用户映射器 |
| | SecondarySortPartitioner | 如何对自然键分区 |
| | SecondarySortGroupComparator | 如何按自然键分组 |
| 阶段2 | LocationCountDriver | 提交阶段2作业的驱动器 |
| | LocationCountMapper | 定义map()完成地址统计 |
| | LocationCountReducer | 定义reduce()完成地址统计 |

# 运行示例

## 阶段1输入

```
# hadoop fs -cat /left_join/zbook/users/users.txt
u1 UT
u2 GA
u3 CA
u4 CA
u5 GA

# hadoop fs -cat /left_join/zbook/transactions/transactions.txt
t1 p3 u1 3 330
t2 p1 u2 1 400
t3 p1 u1 3 600
t4 p2 u2 10 1000
t5 p4 u4 9 90
t6 p1 u1 4 120
t7 p4 u1 8 160
t8 p4 u5 2 40
```

## 运行阶段1

```
# ./run_phase1_left_join.sh
...
13/12/29 21:17:48 INFO input.FileInputFormat: Total input paths to process : 1
...
13/12/29 21:17:48 INFO input.FileInputFormat: Total input paths to process : 1
13/12/29 21:17:49 INFO mapred.JobClient: Running job: job_201312291929_0004
13/12/29 21:17:50 INFO mapred.JobClient:  map 0% reduce 0%
...
13/12/29 21:18:41 INFO mapred.JobClient:  map 100% reduce 100%
13/12/29 21:18:41 INFO mapred.JobClient: Job complete: job_201312291929_0004
...
13/12/29 21:18:41 INFO mapred.JobClient: Map-Reduce Framework
13/12/29 21:18:41 INFO mapred.JobClient:   Map input records=13
...
13/12/29 21:18:41 INFO mapred.JobClient:   Reduce input records=13
13/12/29 21:18:41 INFO mapred.JobClient:   Reduce input groups=5
13/12/29 21:18:41 INFO mapred.JobClient:   Combine output records=0
13/12/29 21:18:41 INFO mapred.JobClient:   Reduce output records=8
13/12/29 21:18:41 INFO mapred.JobClient:   Map output records=13
```

## 阶段1输出（阶段2输入）

```
# hadoop fs -text /left_join/zbook/output/part*
p4 GA
p3 UT
p1 UT
p1 UT
p4 UT
p1 GA
p2 GA
p4 CA
```

### 运行阶段2

```
# ./run_phase2_location_count.sh
...
13/12/29 21:19:28 INFO input.FileInputFormat: Total input paths to process : 10
13/12/29 21:19:28 INFO mapred.JobClient: Running job: job_201312291929_0005
13/12/29 21:19:29 INFO mapred.JobClient: map 0% reduce 0%
...
13/12/29 21:20:24 INFO mapred.JobClient: map 100% reduce 100%
13/12/29 21:20:25 INFO mapred.JobClient: Job complete: job_201312291929_0005
...
13/12/29 21:20:25 INFO mapred.JobClient: Map-Reduce Framework
13/12/29 21:20:25 INFO mapred.JobClient: Map input records=8
...
13/12/29 21:20:25 INFO mapred.JobClient: Reduce input records=8
13/12/29 21:20:25 INFO mapred.JobClient: Reduce input groups=4
13/12/29 21:20:25 INFO mapred.JobClient: Combine output records=0
13/12/29 21:20:25 INFO mapred.JobClient: Reduce output records=4
13/12/29 21:20:25 INFO mapred.JobClient: Map output records=8
```

### 阶段2输出

```
# hadoop fs -cat /left_join/zbook/output2/part*
p1 2
p2 1
p3 1
p4 3
```

# Spark左外连接实现

与MapReduce/Hadoop API相比，由于Spark提供了一个更高层的Java API，这里将在一个Java类（名为LeftOuterJoin）中提供完整的解决方案，包括一系列`map()`、`groupByKey()`和`reduce()`函数。在MapReduce/Hadoop实现中，我们使用了MultipleInputs类，由两个不同的映射器处理两种不同类型的输入。你已经了解到，Spark为映射器和归约器提供了一个更为丰富的API。不再需要特殊的插件类，它提供了很多不同类型的映射器（使用`map()`、`flatMap()`和`flatMapToPair()`函数）。在Spark中，并不使用Hadoop的`MultipleInputs`类，我们要用`JavaRDD.union()`函数返回两个JavaRDD的并集（一个用户RDD和一个交易RDD），这两个RDD将合并来创建一个新的RDD。`JavaRDD.union()`函数定义如下：

```
JavaRDD<T> union(JavaRDD<T> other)
JavaPairRDD<T> union(JavaPairRDD<T> other)

Description: Return the union of this JavaRDD and another one.
             Any identical elements will appear multiple
             times (use .distinct() to eliminate them).
```

你只能对相同类型（`T`）的JavaRDD应用`union()`函数。所以，我们将为用户和交易创建相同的RDD类型。做法如下：

```
JavaPairRDD<String,Tuple2<String,String>> usersRDD = users.map(...);
JavaPairRDD<String,Tuple2<String,String>> transactionsRDD =
   transactions.map(...);

// 这里对usersRDD和transactionsRDD完成union()
JavaPairRDD<String,Tuple2<String,String>> allRDD =
   transactionsRDD.union(usersRDD);
```

union()工作流如图4-4所示。

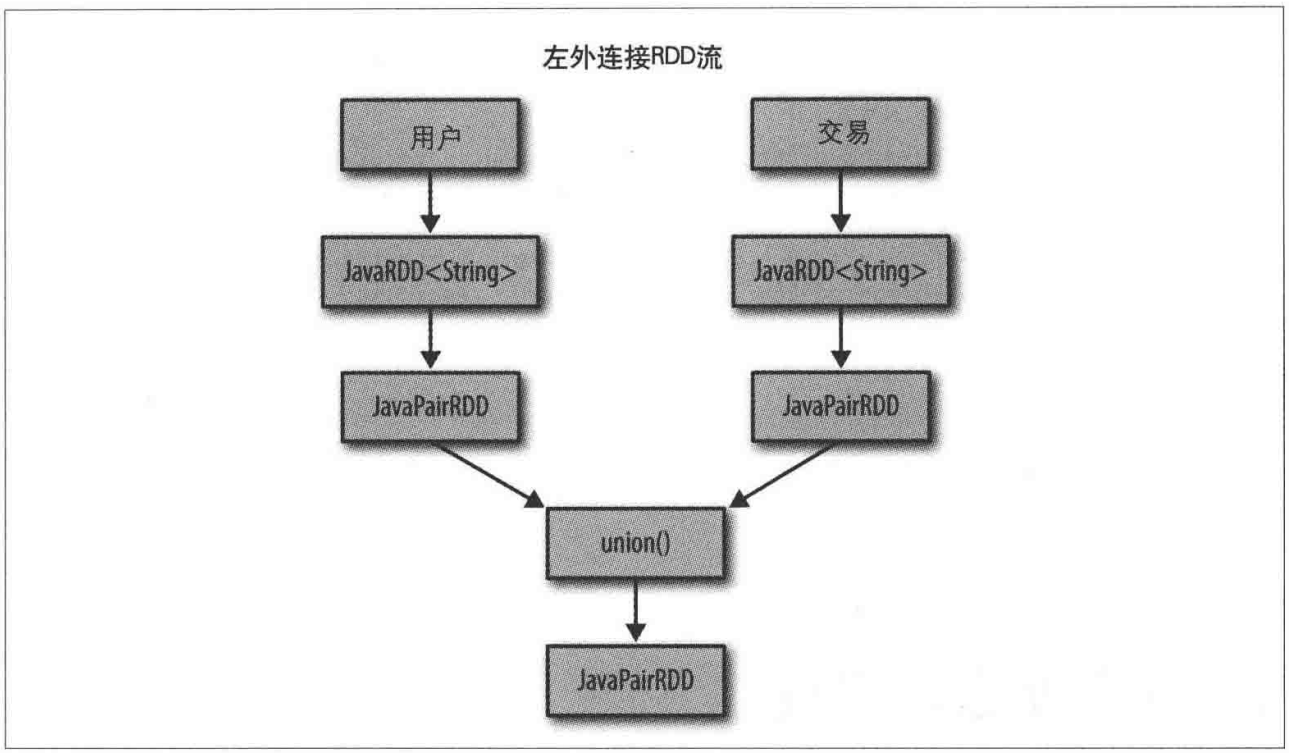



图4-4：union()数据流

作为复习，下面给出用户和交易的数据（这些数据将用作为这一章最后运行示例的输入文件）：

```
# hadoop fs -cat /data/leftouterjoins/users.txt
u1 UT
u2 GA
u3 CA
u4 CA
u5 GA

# hadoop fs -cat /data/leftouterjoins/transactions.txt
t1 p3 u1 3 330
t2 p1 u2 1 400
t3 p1 u1 3 600
t4 p2 u2 10 1000
t5 p4 u4 9 90
t6 p1 u1 4 120
t7 p4 u1 8 160
t8 p4 u5 2 40
```

下面来看这个算法是如何工作的。对于用户和交易数据，可以生成：

```
users => (userID, T2("L", location))
transactions => (userID, T2("P", product))
```

（在这里，T2代表Tuple2）。

接下来，创建这两个数据集的一个并集：

```
all = transactions.union(users);
    = { (userID1, T2("L", location)),
        (userID1, T2("P", P11)),

        (userID1, T2("P", P12)),
        ...
        (userID1, T2("P", P1n)),
        ...
    }
```

这里$P_{ij}$是一个商品ID。

下一步按userID对数据分组。这会生成以下结果：

```
{
  (userID1, List<T2("L", L1), T2("P", P11), T2("P", P12), T2("P", P13), ...>),
  (userID2, List<T2("L", L2), T2("P", P21), T2("P", P22), T2("P", P23), ...>),
  ...
}
```

这里$L_i$是一个locationID，$P_{ij}$是一个商品ID。

## Spark程序

首先，我先给出一个包含11个步骤的高层解决方案（参见示例4-6），然后通过一个实际的Spark代码示例详细分析每一个步骤。

示例4-6：LeftOuterJoin高层解决方案

```
// 步骤1：导入必要的类和接口
public class LeftOuterJoin {
  public static void main(String[] args) throws Exception {
    // 步骤2：读取输入参数
    // 步骤3：创建一个JavaSparkContext对象
    // 步骤4：为用户创建一个JavaRDD
    // 步骤5：为交易创建一个JavaRDD
    // 步骤6：为第4步和第5步生成的RDD创建一个并集
    // 步骤7：调用groupByKey()创建一个JavaPairRDD(userID, List<T2>)
    // 步骤8：创建productLocationsRDD为JavaPairRDD<String,String>
    // 步骤9：查找一个商品的所有地址
    // 结果是一个JavaPairRDD<String, List<String>>
    // 步骤10：对输出做最终处理，将"值"从List<String>
    // 改为Tuple2<Set<String>, Integer>，包含
```

```
15      // 唯一地址及相应数目的一个集合。
16      // 步骤11：打印最终结果RDD
17      System.exit(0);
18    }
19 }
```

## 步骤1：导入必要的类

首先从Spark框架二进制发布版本提供的JAR文件导入所需的类和接口。Spark提供了两个Java包（org.apache.spark.api.java和org.apache.spark.api.java.function），用来创建和管理RDD。这个步骤如示例4-7所示。

示例4-7：步骤1：导入必要的类和接口

```
 1 // 步骤1：导入必要的类和接口
 2 import scala.Tuple2;
 3 import org.apache.spark.api.java.JavaRDD;
 4 import org.apache.spark.api.java.JavaPairRDD;
 5 import org.apache.spark.api.java.JavaSparkContext;
 6 import org.apache.spark.api.java.function.Function;
 7 import org.apache.spark.api.java.function.PairFlatMapFunction;
 8 import org.apache.spark.api.java.function.FlatMapFunction;
 9 import org.apache.spark.api.java.function.PairFunction;
10
11 import java.util.Set;
12 import java.util.HashSet;
13 import java.util.Arrays;
14 import java.util.List;
15 import java.util.ArrayList;
16 import java.util.Collections;
```

## 步骤2：读取输入参数

如示例4-8所示，接下来我们要读取两个输入参数：用户数据和交易数据。用户和交易数据作为HDFS文本文件提供。

示例4-8：步骤2：读取输入参数

```
1    // 步骤2：读取输入参数
2    if (args.length < 2) {
3      System.err.println("Usage: LeftOuterJoin <users> <transactions>");
4      System.exit(1);
5    }
6    String usersInputFile = args[0]; // HDFS文本文件
7    String transactionsInputFile = args[1]; // HDFS文本文件
8    System.out.println("users="+ usersInputFile);
9    System.out.println("transactions="+ transactionsInputFile);
```

这一步的输出如下：

```
users=/data/leftouterjoins/users.txt
transactions=/data/leftouterjoins/transactions.txt
```

## 步骤3：创建一个JavaSparkContext对象

如示例4-9所示，接下来我们要创建一个JavaSparkContext对象。这个对象用来创建第一个RDD。

示例4-9：步骤3：创建一个JavaSparkContext对象

```
// 步骤3：创建一个JavaSparkContext对象
JavaSparkContext ctx = new JavaSparkContext();
```

## 步骤4：为用户创建一个JavaRDD

在这一步中（参见示例4-10），我们要创建一个用户JavaRDD<String>，在这里RDD元素是文本文件中的一个记录（表示userID和locationID）。接下来，使用JavaRDD<String>.mapToPair()函数创建一个新的JavaPairRDD<String,Tuple2<String,String>>，其中键是userID，值是一个Tuple2("L", location)。后面将为交易数据创建Tuple2("P", product)对。标记"L"和"P"分别标识地址和商品。

示例4-10：步骤4：为用户创建一个JavaRDD

```
// 步骤4：为用户创建一个JavaRDD
JavaRDD<String> users = ctx.textFile(usersInputFile, 1);
// <K2,V2> JavaPairRDD<K2,V2> mapToPair(PairFunction<T,K2,V2> f)
// 通过对这个RDD的所有元素应用一个函数返回一个新的RDD。
// PairFunction<T, K, V> 其中 T => Tuple2<K, V>
JavaPairRDD<String,Tuple2<String,String>> usersRDD =
   users.mapToPair(new PairFunction<
                                    String, // T
                                    String, // K
                                    Tuple2<String,String> // V
                                  >() {
 public Tuple2<String,Tuple2<String,String>> call(Strings) {
    String[] userRecord = s.split("\t");
    Tuple2<String,String> location =
       new Tuple2<String,String>("L", userRecord[1]);
    return new Tuple2<String,Tuple2<String,String>>(userRecord[0], location);
 }
});
```

## 步骤5：为交易创建一个JavaRDD

在这一步中（参见示例4-11），我们要创建一个交易JavaRDD<String>，这里的RDD元素是文本文件中的一个记录（表示一个交易记录）。接下来，我们使用JavaRDD<String>.mapToPair()函数创建一个新的JavaPairRDD<String,Tuple2<String,String>>，这里键是userID，值是一个Tuple2("P", product)。上一步中，我们已经为用户创建了Tuple2("L", location)对。标记"L"和"P"分别标识地址和商品。

示例4-11：步骤5：为交易创建一个JavaRDD

```
// 步骤5：为交易创建一个JavaRDD
JavaRDD<String> transactions = ctx.textFile(transactionsInputFile, 1);

// mapToPair
// <K2,V2> JavaPairRDD<K2,V2> mapToPair(PairFunction<T,K2,V2> f)
// 通过对这个RDD的所有元素应用一个函数返回一个新的RDD。
// PairFunction<T, K, V>
// T => Tuple2<K, V>
JavaPairRDD<String,Tuple2<String,String>> transactionsRDD =
      transactions.mapToPair(new PairFunction<
                                              String,               // T
                                              String,               // K
                                              Tuple2<String,String> // V
                                             >() {
  public Tuple2<String,Tuple2<String,String>> call(String s) {
    String[] transactionRecord = s.split("\t");
    Tuple2<String,String> product =
        new Tuple2<String,String>("P", transactionRecord[1]);
    return new Tuple2<String,Tuple2<String,String>>(transactionRecord[2],
                                                    product);
  }
});
```

## 步骤6：为第4步和第5步生成的RDD创建一个并集

这一步中（参见示例4-12），我们将创建两个JavaPairRDD<String,Tuple2<String,String>>实例的一个并集。JavaPairRDD.union()方法要求两个RDD必须有相同的类型。

示例4-12：步骤6：创建RDD并集

```
// 步骤6：为第4步和第5步生成的RDD创建一个并集
JavaPairRDD<String,Tuple2<String,String>> allRDD =
      transactionsRDD.union(usersRDD);
```

示例4-13给出了与这个步骤有相同语义的一个实现，我们只是改变了合并参数的顺序。

示例4-13：步骤6：创建RDD并集（替代实现）

```
// 这里对usersRDD和transactionsRDD完成union()。
JavaPairRDD<String,Tuple2<String,String>> allRDD =
      usersRDD.union(transactionsRDD);
```

两个JavaPairRDD合并的结果是以下键-值对：

```
{
  (userID, Tuple2("L", location)),
  ...
  (userID, Tuple2("P", product))
  ...
}
```

### 步骤7：调用groupByKey()创建JavaPairRDD(userID, List(T2))

接下来，按userID对步骤6创建的数据分组。在示例4-14中可以看到，这一步由JavaPairRDD.groupByKey()实现。

示例4-14：步骤7：创建一个JavaPairRDD

```
1    // 步骤7：调用groupBy()创建JavaPairRDD (userID, List<T2>)
1    // 按userID对allRDD分组
2    JavaPairRDD<String, Iterable<Tuple2<String,String>>> groupedRDD =
3        allRDD.groupByKey();
4    // 现在groupedRDD记录如下所示：
5    // <userIDi, List[T2("L", location),
6    //                T2("P", Pi1),
7    //                T2("P", Pi2),
8    //                T2("P", Pi3), ...
9    //               ]
10   // >
```

这一步的结果如下：

```
(userID1, List[T2("L", location1),
               T2("P", P11),
               T2("P", P12),
               T2("P", P13), ...]),
(userID2, List[T2("L", location2),
               T2("P", P21),
               T2("P", P22),
               T2("P", P23), ...]),
...
```

这里$P_{ij}$是一个productID。

### 步骤8：创建一个JavaPairRDD(String,String)作为productLocationsRDD

在这一步中，从RDD中去除userID。对于一个给定的RDD元素：

```
(userID, List[T2("L", location),
              T2("P", p1),
              T2("P", p2),
              T2("P", p3), ...])
```

创建如下的JavaPairRDD<String,String>：

```
(p1, location)
(p2, location)
(p3, location)
...
```

这一步由JavaPairRDD.flatMapToPair()函数完成，我们把它实现为一个PairFlatMapFunction.call()方法。PairFlatMapFunction的工作如下：

```
PairFlatMapFunction<T, K, V>
T => Iterable<Tuple2<K, V>>
```

在我们的例子中：T是输入，
我们将创建(K, V)对作为输出：

```
T = Tuple2<String, Iterable<Tuple2<String,String>>>
K = String
V = String
```

示例4-15给出了PairFlatMapFunction.call()方法的完整实现。

示例4-15：步骤8：创建 productLocationsRDD

```
// 步骤8：创建productLocationsRDD为JavaPairRDD<String,String>
// PairFlatMapFunction<T, K, V>
// T => Iterable<Tuple2<K, V>>
JavaPairRDD<String,String> productLocationsRDD =
    groupedRDD.flatMapToPair(new PairFlatMapFunction<
       Tuple2<String, Iterable<Tuple2<String,String>>>, // T
       String,                                          // K
       String>() {                                      // V
  public Iterable<Tuple2<String,String>>
     call(Tuple2<String, Iterable<Tuple2<String,String>>> s) {
       // String userID = s._1; // 不需要
       Iterable<Tuple2<String,String>> pairs = s._2;
        String location = "UNKNOWN";
        List<String> products = new ArrayList<String>();
        for (Tuple2<String,String> t2 : pairs) {
              if (t2._1.equals("L")) {
                    location = t2._2;
              }
              else {
                     // t2._1.equals("P")
                     products.add(t2._2);
              }
       }

       // 现在发出(K, V)对
       List<Tuple2<String,String>> kvList =
          new ArrayList<Tuple2<String,String>>();
       for (String product : products) {
          kvList.add(new Tuple2<String, String>(product, location));
       }
       // 需要说明，边必须是双向的；也就是说，
       // 每个{source, destination} 边必然有一个
       // 相应的{destination, source}。
    return kvList;
  }
});
```

## 步骤9：查找一个商品的所有地址

在这一步中，RDD对（product, location）按商品分组。使用JavaPairRDD.

groupByKey()来完成这一步，如示例4-16所示。这一步还会调用JavaPairRDD.collect()函数完成一些基本的调试。

示例4-16：步骤9：查找一个商品的所有地址

```
// 步骤9：查找一个商品的所有地址
// 结果将是JavaPairRDD <String, List<String>>
JavaPairRDD<String, Iterable<String>> productByLocations =
   productLocationsRDD.groupByKey();

// debug3
List<Tuple2<String, List<String>>> debug3 = productByLocations.collect();
System.out.println("--- debug3 begin ---");
for (Tuple2<String, Iterable<String>> t2 : debug3) {
  System.out.println("debug3 t2._1="+t2._1);
  System.out.println("debug3 t2._2="+t2._2);
}
System.out.println("--- debug3 end ---");
```

## 步骤10：改变值，对输出做最终处理

步骤9生成了一个JavaPairRDD<String, List<String>>对象，在这里键是商品（作为一个字符串），值是一个List<String>，这是一个地址列表，可能存在重复。为了从值中删除重复的元素，我们使用了JavaPairRDD.mapValues()函数。实现这个函数时，要把List<String>转换为一个Set<String>。需要说明，键并没有改变。映射值由Function(T, R).call()实现，这里T是输入（List<String>），R是输出（Tuple2<Set<String>, Integer>）。如示例4-17所示。

示例4-17：步骤10：对输出做最终处理

```
// 步骤10：对输出做最终处理，将"值"从List<String>
// 改为Tuple2<Set<String>, Integer>，其中包含
// 唯一地址及相应数目的一个集合
JavaPairRDD<String, Tuple2<Set<String>, Integer>> productByUniqueLocations =
      productByLocations.mapValues(
         new Function<Iterable<String>, // 输入
         Tuple2<Set<String>, Integer>   // 输出
>() {
  public Tuple2<Set<String>, Integer> call(Iterable<String> s) {
    Set<String> uniqueLocations = new HashSet<String>();
    for (String location : s) {
        uniqueLocations.add(location);
    }
    return new Tuple2<Set<String>, Integer>(uniqueLocations,
                                            uniqueLocations.size());
  }
});
```

## 步骤11：打印最终结果RDD

最后一步如示例4-18所示，使用JavaPairRDD.collect()方法发出结果。

示例4-18：步骤11：打印最终结果RDD

```
// 步骤11：打印最终结果RDD
// debug4
System.out.println("=== Unique Locations and Counts ===");
List<Tuple2<String, Tuple2<Set<String>, Integer>>> debug4 =
   productByUniqueLocations.collect();
System.out.println("--- debug4 begin ---");
for (Tuple2<String, Tuple2<Set<String>, Integer>> t2 : debug4) {
  System.out.println("debug4 t2._1="+t2._1);
  System.out.println("debug4 t2._2="+t2._2);
}
System.out.println("--- debug4 end ---");
```

# 运行Spark解决方案

## Shell脚本

```
# cat run_left_outer_join.sh
#!/bin/bash
export JAVA_HOME=/usr/java/jdk7
export SPARK_HOME=/usr/local/spark-1.1.0
export SPARK_MASTER=spark://myserver100:7077
export BOOK_HOME=/home/data-algorithms-book
export APP_JAR=$BOOK_HOME/dist/data_algorithms_book.jar
USERS=/data/leftouterjoins/users.txt
TRANSACTIONS=/data/leftouterjoins/transactions.txt
# 在一个Spark集群上运行
prog=org.dataalgorithms.chap04.spark.SparkLeftOuterJoin
$SPARK_HOME/bin/spark-submit \
  --class $prog \
  --master $SPARK_MASTER \
  --executor-memory 2G \
  --total-executor-cores 20 \
    $APP_JAR $USERS $TRANSACTIONS
```

## 运行Shell脚本

运行示例的日志输出如下所示。为适应版面，这里做了删减，并且对格式有所调整：

```
# ./run_left_outer_join.sh
users=/data/leftouterjoins/users.txt
transactions=/data/leftouterjoins/transactions.txt
...
14/06/03 17:52:01 INFO scheduler.DAGScheduler: Stage 0
  (collect at LeftOuterJoin2.java:112) finished in 0.163 s
14/06/03 17:52:01 INFO spark.SparkContext: Job finished:
  collect at LeftOuterJoin2.java:112, took 6.365762312 s
--- debug3 begin ---
debug3 t2._1=p2
debug3 t2._2=[GA]
debug3 t2._1=p4
debug3 t2._2=[GA, UT, CA]
```

```
debug3 t2._1=p1
debug3 t2._2=[GA, UT, UT]
debug3 t2._1=p3
debug3 t2._2=[UT]
--- debug3 end ---
=== Unique Locations and Counts ===
14/06/03 17:52:01 INFO spark.SparkContext: Starting job:
  collect at LeftOuterJoin2.java:137
14/06/03 17:52:01 INFO spark.MapOutputTrackerMaster: Size
  of output statuses for shuffle 1 is 156 bytes
...
14/06/03 17:52:01 INFO scheduler.DAGScheduler: Stage 3
  (collect at LeftOuterJoin2.java:137) finished in 0.058 s
14/06/03 17:52:01 INFO spark.SparkContext: Job finished:
  collect at LeftOuterJoin2.java:137, took 0.081830132 s
--- debug4 begin ---
debug4 t2._1=p2
debug4 t2._2=([GA],1)
debug4 t2._1=p4
debug4 t2._2=([UT, GA, CA],3)
debug4 t2._1=p1
debug4 t2._2=([UT, GA],2)
debug4 t2._1=p3
debug4 t2._2=([UT],1)
--- debug4 end ---
...
14/06/03 17:52:02 INFO scheduler.DAGScheduler: Stage 6
  (saveAsTextFile at LeftOuterJoin2.java:144) finished in 1.060 s
14/06/03 17:52:02 INFO spark.SparkContext: Job finished: saveAsTextFile
  at LeftOuterJoin2.java:144, took 1.169724354 s
```

# 在YARN上运行Spark

这一节中，我们会介绍如何将Spark的ApplicationMaster提交到Hadoop YARN的ResourceManager，并指示Spark运行这个左外连接程序。另外，还会要求这个Spark程序把最终的结果保存到一个HDFS文件中。创建productByUniqueLocations RDD之后可以通过下面这行代码将文件保存到HDFS（*/left/output*是一个HDFS输出目录）：

```
productByUniqueLocations.saveAsTextFile("/left/output");
```

## 在YARN上运行Spark的脚本

在YARN上运行Spark的脚本如下：

```
# cat leftjoin.sh
export SPARK_HOME=/usr/local/spark-1.0.0
export SPARK_JAR=spark-assembly-1.0.0-hadoop2.3.0.jar
export SPARK_ASSEMBLY_JAR=$SPARK_HOME/assembly/target/scala-2.10/$SPARK_JAR
export JAVA_HOME=/usr/java/jdk6
export HADOOP_HOME=/usr/local/hadoop-2.3.0
export HADOOP_CONF_DIR=$HADOOP_HOME/conf
```

```
export YARN_CONF_DIR=$HADOOP_HOME/conf
export BOOK_HOME=/mp/data-algorithms-book
export APP_JAR=$BOOK_HOME/dist/data_algorithms_book.jar
# 将Spark的ApplicationMaster提交到YARN的ResourceManager,
# 指示Spark运行LeftOuterJoin2示例
prog=org.dataalgorithms.chap04.spark.SparkLeftOuterJoin
SPARK_JAR=$SPARK_ASSEMBLY_JAR \
      $SPARK_HOME/bin/spark-class org.apache.spark.deploy.yarn.Client \
      --jar $APP_JAR \
      --class $prog \
      --args yarn-standalone \
      --args /left/users.txt \
      --args /left/transactions.txt \
      --num-workers 3 \
      --master-memory 4g \
      --worker-memory 2g \
      --worker-cores 1
```

关于Spark参数（如num-workers和worker-memory）和环境变量的详细内容，可以参考Spark Summit slides（*http://bit.ly/spark_summit*）。其中大部分参数都依赖工作节点数、每个集群节点的内核数及可用的RAM大小。要想找出最优的设置，这需要对不同大小的输入数据做一些尝试和校正，通过反复试验来得出结论。

## 运行脚本

运行示例的日志输出如下所示。为适应版面，这里做了删减，并且对格式有所调整：

```
# ./leftjoin.sh
14/05/28 16:49:31 INFO RMProxy: Connecting to
.. ResourceManager at myserver100:8032
14/05/28 16:49:31 INFO Client:
  Got Cluster metric info from ApplicationsManager (ASM),
number of NodeManagers: 13
...
14/05/28 16:49:33 INFO Client: Command for starting
.. the Spark ApplicationMaster:
$JAVA_HOME/bin/java -server -Xmx4096m -Djava.io.tmpdir=$PWD/tmp
org.apache.spark.deploy.yarn.ApplicationMaster
--class LeftOuterJoin2 --jar /usr/local/spark/tmp/data_algorithms_book.jar
--args 'yarn-standalone' --args '/left/users.txt'
--args '/left/transactions.txt' --worker-memory 2048 --worker-cores 1
--num-workers 3 1> <LOG_DIR>/stdout 2> <LOG_DIR>/stderr
14/05/28 16:49:33 INFO Client: Submitting application to ASM
...
yarnAppState: FINISHED
distributedFinalState: SUCCEEDED
appTrackingUrl: http://myserver100:50030/proxy/application_1401319796895_0008/A
appUser: hadoop
```

## 检查期望输出

下面检查生成的HDFS输出：

```
# hadoop fs -ls /left/output
Found 3 items
-rw-r--r-- 2 hadoop supergroup 0 2014-05-28 16:49 /left/output/_SUCCESS
-rw-r--r-- 2 hadoop supergroup 36 2014-05-28 16:49 /left/output/part-00000
-rw-r--r-- 2 hadoop supergroup 32 2014-05-28 16:49 /left/output/part-00001

# hadoop fs -cat /left/output/part*
(p2,([GA],1))
(p4,([UT, GA, CA],3))
(p1,([UT, GA],2))
(p3,([UT],1))
```

# 使用leftOuterJoin()的Spark实现

这一节使用Spark的内置JavaPairRDD.leftOuterJoin()方法实现左外连接（注意，MapReduce/Hadoop没有提供类似leftOuterJoin()方法的高层API功能）：

```
import scala.Tuple2;
import com.google.common.base.Optional;
import org.apache.spark.api.java.JavaPairRDD;

JavaPairRDD<K,Tuple2<V,Optional<W>>> leftOuterJoin(JavaPairRDD<K,W> other)
  // 完成this和other的左外连接。对于this中的
  // 各个元素(k, v)，结果RDD将
  // 包含other中对应w的所有（k, (v, Some(w)))对，或者
  // 如果other中没有键为k的元素，结果中则包含（k, (v, None)）对
```

使用Sparks的JavaPairRDD.leftOuterJoin()方法可以帮助我们避免：

- 对users和transactions使用JavaPairRDD.union()操作，这个操作的开销很大。
- 引入定制标志，如为地址增加"L"，为商品增加"P"。
- 使用额外的RDD转换来区分定制标志。

利用JavaPairRDD.leftOuterJoin()方法，我们可以很高效地生成结果。transactions-RDD 是左表，usersRDD是右表：

```
JavaPairRDD<String,String> usersRDD = ...;        // (K=userID, V=location)
JavaPairRDD<String,String> transactionsRDD = ...; // (K=userID, V=product)
// 由内置leftOuterJoin()完成左外连接
JavaPairRDD<String, Tuple2<String,Optional<String>>> joined =
    transactionsRDD.leftOuterJoin(usersRDD);
```

现在，joined RDD包含：

```
(u4,(p4,Optional.of(CA)))
(u5,(p4,Optional.of(GA)))
(u2,(p1,Optional.of(GA)))
(u2,(p2,Optional.of(GA)))
(u1,(p3,Optional.of(UT)))
```

```
(u1,(p1,Optional.of(UT)))
(u1,(p1,Optional.of(UT)))
(u1,(p4,Optional.of(UT)))
```

由于我们只对商品和唯一地址感兴趣，所以下面的步骤中将忽略userID（键）。这要通过另一个JavaPairRDD.mapToPair()函数来完成。忽略userID之后，将生成：

```
(p4,CA)
(p4,GA)
(p1,GA)
(p2,GA)
(p3,UT)
(p1,UT)
(p1,UT)
(p4,UT)
```

其中包含生成商品和唯一地址列表所需的信息。

## Spark程序

示例4-19中给出了高层步骤，展示了如何使用Spark的内置JavaPairRDD.leftOuterJoin()方法。后面将通过示例详细讨论每一个步骤。

示例4-19：高层步骤

```
// 步骤1：导入必要的类和接口
public class SparkLeftOuterJoin {
  public static void main(String[] args) throws Exception {
    // 步骤2：读取输入参数
    // 步骤3：创建Spark的上下文对象
    // 步骤4：为用户数据创建RDD
    // 步骤5：为用户（右表）创建（userID,location）对
    // 步骤6：为交易数据创建RDD
    // 步骤7：为交易（左表）创建（userID,product）对
    // 步骤8：使用Spark的内置JavaPairRDD.leftOuterJoin()方法
    // 步骤9：创建（product, location）对
    // 步骤10：按键对（product, location）对分组
    // 步骤11：按键创建最终输出（product, Set<location>）对
    System.exit(0);
  }
}
```

### 步骤1：导入必要的类和接口

首先导入必要的类和接口（参见示例4-20）。Optional[注2]表示一个不可变的对象，可能包含另一个对象的非null引用（这对于左外连接很有用，因为如果键在左表中出现但在右

注2：com.google.common.base.Optional<T>是一个抽象类。

表中未出现，连接的值可能包含null）。使用工厂类JavaSparkContext来创建新RDD。

示例4-20：步骤1：导入必要的类和接口

```
// 步骤1：导入必要的类和接口
import scala.Tuple2;
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.api.java.JavaPairRDD;
import org.apache.spark.api.java.JavaSparkContext;
import org.apache.spark.api.java.function.Function;
import org.apache.spark.api.java.function.PairFunction;
import com.google.common.base.Optional;
import java.util.Set;
import java.util.HashSet;
```

## 步骤2：读取输入参数

这个步骤中（参见示例4-21），将读取包含用户和交易数据的HDFS输入文件中的地址。这些输入文件将用来创建左表（transactionsRDD）和右表(usersRDD)。

示例4-21：步骤2：读取输入参数

```
    // 步骤2：读取输入参数
    if (args.length < 2) {
      System.err.println("Usage: LeftOuterJoin <users> <transactions>");
      System.exit(1);
    }

    String usersInputFile = args[0];
    String transactionsInputFile = args[1];
    System.out.println("users="+ usersInputFile);
    System.out.println("transactions="+ transactionsInputFile);
```

## 步骤3：创建Spark的上下文对象

这个步骤中（参见示例4-22），将创建一个JavaSparkContext对象，它将用来创建一个新的RDD。

示例4-22：步骤3：创建Spark的上下文对象

```
    // 步骤3：创建Spark的上下文对象
    JavaSparkContext ctx = new JavaSparkContext();
```

## 步骤4：为用户数据创建一个RDD

在这个步骤中（参见示例4-23）将创建usersRDD，这是一组（userID, location）对。usersRDD表示左外连接操作的“右”表。

示例4-23：步骤4：为用户数据创建RDD

```
1    // 步骤4：为用户数据创建RDD
2    JavaRDD<String> users = ctx.textFile(usersInputFile, 1);
```

## 步骤5：创建usersRDD（右表）

在示例4-24中可以看到，这一步将创建右表usersRDD，其中包含用户输入数据中的（userID,location）对。

示例4-24：步骤5：为用户创建（userID,location）对

```
1 // 步骤5：为用户创建 (userID,location)对
2 // <K2,V2> JavaPairRDD<K2,V2> mapToPair(PairFunction<T,K2,V2> f)
3 // 通过对这个RDD的所有元素应用一个函数返回一个新的RDD。
4 // PairFunction<T, K, V>
5 // T => Tuple2<K, V>
6 JavaPairRDD<String,String> usersRDD =
7 // T K V
8 users.mapToPair(new PairFunction<String, String, String>() {
9 public Tuple2<String,String> call(String s) {
10 String[] userRecord = s.split("\t");
11 String userID = userRecord[0];
12 String location = userRecord[1];
13 return new Tuple2<String,String>(userID, location);
14 }
15 });
```

## 步骤6：为交易数据创建一个RDD

这个步骤中（参见示例4-25），将创建transactionsRDD，这是一组（userID, product）对。transactionsRDD表示左外连接操作的“左”表。

示例4-25：步骤6：为交易数据创建RDD

```
1    // 步骤6：为交易数据创建RDD
2    JavaRDD<String> transactions = ctx.textFile(transactionsInputFile, 1);
```

## 步骤7：创建transactionsRDD（左表）

这个步骤中（参见示例4-26），将创建左表transactionsRDD，其中包含交易输入数据中的（userID,product）对。

示例4-26：步骤7：为交易创建（userID,product）对

```
1   // 步骤7：为交易创建（userID,product）对
2   // PairFunction<T, K, V>
3   // T => Tuple2<K, V>
4   // 示例交易输入：t1 p3 u1 3 330
5   JavaPairRDD<String,String> transactionsRDD =
6         //                                    T         K          V
7         transactions.mapToPair(new PairFunction<String, String, String>() {
8     public Tuple2<String,String> call(String s) {
9       String[] transactionRecord = s.split("\t");
```

```
10        String userID = transactionRecord[2];
11        String product = transactionRecord[1];
12        return new Tuple2<String,String>(userID, product);
13      }
14    });
```

## 步骤8：使用Spark的内置JavaPairRDD.leftOuterJoin()方法

这是完成左外连接操作的核心步骤，将使用Spark的`JavaPairRDD.leftOuterJoin()`方法（参见示例4-27）。

示例4-27：步骤8：使用Spark的内置JavaPairRDD.leftOuterJoin()方法

```
1   // 步骤8：使用Spark的内置JavaPairRDD.leftOuterJoin()方法。
2   // JavaPairRDD<K,Tuple2<V,Optional<W>>> leftOuterJoin(JavaPairRDD<K,W> other)
3   // 完成this和other的左外连接。对于this中的各个元素（k, v），
4   //结果RDD将包含other中对应w的所有（k, (v, Some(w)）对，或者
5   // 如果other中没有键为k的元素，结果中则包含（k, (v, None)）对。
6   //
7   // 这里我们完成一个transactionsRDD.leftOuterJoin(usersRDD)
8   JavaPairRDD<String, Tuple2<String,Optional<String>>> joined =
9       transactionsRDD.leftOuterJoin(usersRDD);
10  joined.saveAsTextFile("/output/1");
```

这一步将创建以下输出（左外连接操作的结果）：

```
# hadoop fs -cat /output/1/part*
(u4,(p4,Optional.of(CA)))
(u5,(p4,Optional.of(GA)))
(u2,(p1,Optional.of(GA)))
(u2,(p2,Optional.of(GA)))
(u1,(p3,Optional.of(UT)))
(u1,(p1,Optional.of(UT)))
(u1,(p1,Optional.of(UT)))
(u1,(p4,Optional.of(UT)))
```

## 步骤9：创建（product, location）对

这一步会建立另一个`JavaPairRDD`，其中包含（`product`, `location`）对。注意，在示例4-28中，完全忽略了`userID`，因为我们只对商品及相应的唯一用户地址感兴趣。

示例4-28：步骤9：创建（product, location）对

```
1   // 步骤9：创建（product, location）对
2   JavaPairRDD<String,String> products =
3      joined.mapToPair(new PairFunction<
4          Tuple2<String, Tuple2<String,Optional<String>>>, // T
5          String, // K
6          String> // V
7      () {
8      public Tuple2<String,String> call(Tuple2<String,
9                                         Tuple2<String,Optional<String>>> t) {
10        Tuple2<String,Optional<String>> value = t._2;
```

```
11          return new Tuple2<String,String>(value._1, value._2.get());
12       }
13 });
14 products.saveAsTextFile("/output/2");
```

这一步会创建以下输出：

```
# hadoop fs -cat /output/2/part*
(p4,CA)
(p4,GA)
(p1,GA)
(p2,GA)
(p3,UT)
(p1,UT)
(p1,UT)
(p4,UT)
```

## 步骤10：按键完成（K=product, V=location）对的分组

这一步按键完成（K=product, V=location）对的分组。结果将是（K, V2），其中V2是一个地址列表（包含重复的地址）。

示例4-29：步骤10：按键完成（K=product, V=location）对的分组

```
1   // 步骤10：按键完成（K=product, V=location）对的分组
2   JavaPairRDD<String, Iterable<String>> productByLocations = products.groupByKey();
3   productByLocations.saveAsTextFile("/output/3");
```

这一步生成以下输出：

```
# hadoop fs -cat /output/3/p*
(p1,[GA, UT, UT])
(p2,[GA])
(p3,[UT])
(p4,[CA, GA, UT])
```

## 步骤11：创建最终输出（K=product, V=Set(location)）

最后一步如示例4-30所示，将删除重复的地址，创建（K, V2），其中V2是一个Tuple2<Set<location>, size>。

示例4-30：步骤11：按键创建最终输出（K=product, V=Set<location>）对

```
1    // 步骤11：按键创建最终输出（K=product, V=Set<location>）对
2    JavaPairRDD<String, Tuple2<Set<String>, Integer>> productByUniqueLocations =
3       productByLocations.mapValues(
4          new Function< Iterable<String>,            // 输入
5                        Tuple2<Set<String>, Integer> // 输出
6          >() {
7       public Tuple2<Set<String>, Integer> call(Iterable<String>s) {
8          Set<String> uniqueLocations = new HashSet<String>();
9          for (String location : s) {
10            uniqueLocations.add(location);
```

```
11          }
12          return new Tuple2<Set<String>, Integer>(uniqueLocations,
13                                                  uniqueLocations.size());
14        }
15    });
16
17    productByUniqueLocations.saveAsTextFile("/output/4");
```

这一步会创建以下最终输出：

```
# hadoop fs -cat /output/4/p*
(p1,([UT, GA],2))
(p2,([GA],1))
(p3,([UT],1))
(p4,([UT, GA, CA],3))
```

## 合并步骤10和11

可以把步骤10和步骤11合并为一个Spark操作。我们利用Spark的combineByKey()完成这个工作，这是按键聚集函数的最一般的形式，允许采用灵活的方式组合键相同的值。reduceByKey()和combineByKey()的主要区别是什么？reduceByKey()可以将类型V的值归约为V（相同的数据类型，例如，可以将整数值相加或相乘）。不过，combineByKey()可以将类型V的值组合/转换为另一种类型C。例如，我们可能希望将整数(V)值组合/转换为一个整数集（Set<Integer>）。基本说来，可以利用combineByKey()返回与输入数据类型不同的值。要使用combineByKey()，需要提供一组函数。最简单的combineByKey()签名如下所示：

```
public <C> JavaPairRDD<K,C> combineByKey(
                                          Function<V,C> createCombiner,
                                          Function2<C,V,C> mergeValue,
                                          Function2<C,C,C> mergeCombiners
                                          )
```

描述：这是一个泛型函数，使用一组定制的聚集函数合并对应各个键的元素。将一个JavaPairRDD[(K, V)]转换为类型为JavaPairRDD[(K, C)]的结果，这里的C为"组合类型"。注意，V和C可能是不同的类型。例如，可能希望将一个类型为（Int, Int）的RDD分组为类型为（Int, List[Int]）的RDD。
Users提供了3个函数：
- createCombiner，将一个V转换为一个C（例如，创建一个单元素列表）。
- mergeValue，将一个V合并到一个C（例如，把它增加到列表末尾）。
- mergeCombiners，合并两个C来创建一个新的C

我们的目标是为各个键分别创建一个Set<String>（每个键有一个值列表，各个值分别是一个字符串）。要完成这个工作，需要实现3个基本函数，如示例4-31所示。

示例4-31：combineByKey()使用的基本函数

```
1 Function<String, Set<String>> createCombiner =
2   new Function<String, Set<String>>() {
3   @Override
```

```
 4   public Set<String> call(String x) {
 5     Set<String> set = new HashSet<String>();
 6     set.add(x);
 7     return set;
 8   }
 9 };
10 Function2<Set<String>, String, Set<String>> mergeValue =
11   new Function2<Set<String>, String, Set<String>>() {
12   @Override
13   public Set<String> call(Set<String> set, String x) {
14     set.add(x);
15     return set;
16   }
17 };
18  Function2<Set<String>, Set<String>, Set<String>> mergeCombiners =
19    new Function2<Set<String>, Set<String>, Set<String>>() {
20    @Override
21    public Set<String> call(Set<String> a, Set<String> b) {
22      a.addAll(b);
23      return a;
24   }
25 };
```

实现这3个基本函数之后，下面可以使用combineByKey()合并步骤10和11。在此之前，首先明确输入和输出，如表4-4所示。

表4-4：输入、输出和转换器

| | |
|---|---|
| 输入 | products(K:String, V:String) |
| 输出 | productUniqueLocations(K: String, V: Set<String>) |
| 转换器 | combineByKey() |

示例4-32显示了如何使用combineByKey()转换器。

示例4-32：使用combineByKey()

```
1 JavaPairRDD<String, Set<String>> productUniqueLocations =
2     products.combineByKey(createCombiner, mergeValue, mergeCombiners);
3 // 发出最终输出
4 Map<String, Set<String>> productMap = productLocations.collectAsMap();
5 for (Entry<String, Set<String>> entry : productMap.entrySet()) {
6   System.out.println(entry.getKey() + ":" + entry.getValue());
7 }
```

# 在YARN上运行示例

## 输入（右表）

```
# hadoop fs -cat /data/leftouterjoins/users.txt
u1   UT
u2   GA
u3   CA
```

```
u4   CA
u5   GA
```

## 输入（左表）

```
# hadoop fs -cat /data/leftouterjoins/transactions.txt
t1   p3   u1   3    330
t2   p1   u2   1    400
t3   p1   u1   3    600
t4   p2   u2  10   1000
t5   p4   u4  9      90
t6   p1   u1  4     120
t7   p4   u1  8     160
t8   p4   u5  2      40
```

## 脚本

```
# cat ./run_left_outer_join_spark.sh
#!/bin/bash
export JAVA_HOME=/usr/java/jdk7
export SPARK_HOME=/usr/local/spark-1.0.0
export HADOOP_HOME=/usr/local/hadoop-2.5.0
export HADOOP_CONF_DIR=$HADOOP_HOME/etc/hadoop
export YARN_CONF_DIR=$HADOOP_HOME/etc/hadoop
BOOK_HOME=/mp/data-algorithms-book
APP_JAR=$BOOK_HOME/dist/data_algorithms_book.jar
USERS=/data/leftouterjoins/users.txt
TRANSACTIONS=/data/leftouterjoins/transactions.txt
prog=org.dataalgorithms.chap04.spark.SparkLeftOuterJoin
$SPARK_HOME/bin/spark-submit
    --class $prog \
    --master yarn-cluster \
    --num-executors 3 \
    --driver-memory 1g \
    --executor-memory 1g \
    --executor-cores 10 \
    $APP_JAR $USERS $TRANSACTIONS
```

## 生成的HDFS输出

```
# hadoop fs -cat /output/1/p*
(u5,(p4,Optional.of(GA)))
(u2,(p1,Optional.of(GA)))
(u2,(p2,Optional.of(GA)))
(u1,(p3,Optional.of(UT)))
(u1,(p1,Optional.of(UT)))
(u1,(p1,Optional.of(UT)))
(u1,(p4,Optional.of(UT)))

# hadoop fs -cat /output/2/p*
(p4,CA)
(p4,GA)
(p1,GA)
(p2,GA)
```

```
(p3,UT)
(p1,UT)
(p1,UT)
(p4,UT)

# hadoop fs -cat /output/3/p*
(p1,[GA, UT, UT])
(p2,[GA])
(p3,[UT])
(p4,[CA, GA, UT])

# hadoop fs -cat /output/4/p*
(p1,([UT, GA],2))
(p2,([GA],1))
(p3,([UT],1))
(p4,([UT, GA, CA],3))
```

这一章实现了左外连接设计模式，左外连接常用于在分布式编程环境分析商业交易。下一章将介绍另一个设计模式：反转排序（Order Inversion），我们将采用MapReduce范式实现这个设计模式。

# 第5章

# 反转排序

这一章重点介绍反转排序（Order Inversion，OI）设计模式，这种设计模式可以用来控制MapReduce框架中归约器值的顺序（这很有用，因为一些计算需要有序的数据）。通常会在数据分析阶段应用OI模式。在Hadoop和Spark中，值到达归约器的顺序是未定义的（没有明确的顺序，除非我们利用MapReduce的排序阶段将计算所需的数据推至归约器）。OI模式适用成对模式（使用更简单的数据结构，需要更少的归约器内存），因为归约器阶段不需要额外的归约器值排序。

为了帮助理解OI模式，下面首先来看一个简单的例子。考虑一个对应组合键（$K_1$, $K_2$）的归约器，假定$K_1$是这个组合键的自然键部分。假设这个归约器接收到下面的值（这些值没有确定的顺序）：

$$V_1, V_2, \ldots, V_n$$

通过实现OI模式，可以对到达归约器（对应键（$K_1$, $K_2$））的值进行排序和分类。使用OI模式的唯一目的是适当地确定提供给归约器的数据的顺序。为了展示OI设计模式，下面假设$K_1$是组合键的固定部分，在这里$K_2$只有3个不同的值 $\{K_{2a}, K_{2b}, K_{2c}\}$（实际可以有任意多个值），这将生成表5-1所示的值（需要说明，必须把键 $\{(K_1, K_{2a}), (K_1, K_{2b}), (K_1, K_{2c})\}$ 发送到相同的归约器）。

**表5-1：对应自然键$K_1$的键和值**

| 组合键 | 值 |
|---|---|
| $(K_1, K_{2a})$ | $\{A_1, A_2, \ldots, A_m\}$ |
| $(K_1, K_{2b})$ | $\{B_1, B_2, \ldots, B_p\}$ |
| $(K_1, K_{2c})$ | $\{C_1, C_2, \ldots, C_q\}$ |

在这个表中：

- m + p + q = n
- 排序顺序如下：

  $K_{2a} < K_{2b} < K_{2c}$（升序）

  或：

  $K_{2a} > K_{2b} > K_{2c}$（降序）
- $A_i, B_j, C_k \in \{V_1, V_2, ..., V_n\}$

利用适当的OI模式实现，可以对归约器值排序，如下所示：

$A_1, A_2, ..., A_m, B_1, B_2, ..., B_p, C_1, C_2, ..., C_q$

由于归约器值是有序的，这就允许我们首先从$A_i$开始，再到$B_j$，最后到$C_k$完成一些计算。需要说明，这里不需要在内存中缓存值。关键问题是如何得到所需的行为。答案就是定义一个定制分区器，它只关注组合键（$K_1, K_2$）左边的部分（$K_1$，即自然归约器键）。也就是说，定制分区器只根据左键（$K_1$）的散列进行分区。

# 反转排序模式示例

这一节中，我们来研究一个简单的例子，通过计算一个给定文档集中单词的相对频度来展示OI模式。这里的目标是建立一个$N \times N$矩阵（我们称为矩阵$M$），其中$N = |V|$（所有给定文档的单词量），每个单元$M_{ij}$包含一个特定上下文中单词$W_i$与单词$W_j$共同出现的次数。为简单起见，我们将这个上下文定义为$W_i$的邻域。例如，给定以下单词：

$W_1, W_2, W_3, W_4, W_5, W_6$

如果定义一个单词的邻域为这个单词的前两个单词和后两个单词，那么这6个单词的邻域表则如表5-2所示。

**表5-2：邻域表**

| 单词 | 邻域 ±2 |
|---|---|
| $W_1$ | $W_2, W_3$ |
| $W_2$ | $W_1, W_3, W_4$ |
| $W_3$ | $W_1, W_2, W_4, W_5$ |
| $W_4$ | $W_2, W_3, W_5, W_6$ |
| $W_5$ | $W_3, W_4, W_6$ |
| $W_6$ | $W_4, W_5$ |

对于这个例子，计算相对频度需要得到边缘计数，也就是行和列总数。不过，在得到所有计数之前，将无法计算边缘计数。因此，要让边缘计数在联合计数之前到达归约器。可以把这些值缓存在内存中，不过如果这些值不能完全放在内存中，这种方法就不可伸缩。需要说明，要计算相对频度，我们并不会使用绝对单词数。Jimmy Lin和Chris Dyer[17]指出："绝对计数的缺点在于，它没有考虑到一些单词会比另外一些单词更频繁地出现。单词$W_i$可能经常与$W_j$共同出现，只是因为其中一个单词相当常用。对于这个问题，一个简单的补救方法是把绝对计数转换为相对频度$f(W_j|W_i)$。也就是说，在$W_i$的上下文中$W_j$出现次数的比例有多大？"可以使用下面的公式来计算相对频度：

$$f(W_j|W_i)=\frac{N(w_i,w_j)}{\sum_w N(W_i,w)}$$

在这里，$N(a, b)$表示语料库（作为输入的所有文档的一个集合）中观察到的一个特定共现单词对出现的次数。因此，我们要得到这个联合事件（单词共现）的次数，除以边缘计数（条件变量与所有其他单词共现的次数总和）。

# 反转排序模式的MapReduce/Hadoop实现

继续来看我们的例子：计算一个给定文档集的相对频度。我们要生成两个数据序列。第一个序列是单词的总邻域计数（一个单词的总共现次数）；用组合键（$W$, *）表示，其中$W$表示单词。第二个序列是这个单词与其他特定单词的共现次数；用组合键（$W$, $W_2$）表示。因此，归约器会接收表5-3中的有序键-值对集。

**表5-3：归约器键-值对（有序）**

| 键 | 值（整数） |
|---|---|
| ($W$, *) | $A_1, A_2, ..., A_m$ |
| ($W$, $W_1$) | $B_1, B_2, ..., B_p$ |
| ($W$, $W_2$) | $C_1, C_2, ..., C_q$ |
| ... | ... |

表5-4给出一个具体的例子。

**表5-4：归约器键-值对（有序）具体示例**

| 键 | 值 |
|---|---|
| (dog, *) | 100, 129, 500, ... |
| (dog, bite) | 2, 1, 1, 1 |
| (dog, best) | 1, 1, 1, 1, 1 |
| ... | ... |

现在我们来讨论相对主题词共现和OI设计模式。为此，要使用各个词对的相对频度计算共现矩阵，而不是使用绝对数值。因此，我们要统计各个词对（$W_i, W_j$）的出现次数，除以包含$W_i$的词对总数（边缘计数）。在我们的MapReduce解决方案中，将提供一个map()函数、一个reduce()函数和一个定制分区器（OrderInversionPartitioner类），只根据PairOfWords[注1]中的第一个元素应用分区器，将关于同一个单词的所有数据发送给同一个归约器。另外，我们将用PairOfWords表示组合键（包括两个单词）。

## 定制分区器

Hadoop提供了一个插件体系结构来建立定制分区器，如示例5-1所示。

示例5-1：定制分区器：OrderInversionPartitioner

```
1 ...
2 import org.apache.hadoop.mapreduce.Job;
3 import org.apache.hadoop.conf.Configuration;
4 ...
5
6 public class RelativeFrequencyDriver {
7
8     public static void main(String[] args) throws Exception {
9         Job job = Job.getInstance(new Configuration());
10        ...
11        job.setPartitionerClass(OrderInversionPartitioner.class);
12        ...
13    }
14 }
```

Hadoop还提供了一个API来插入定制分区器：

```
job.setPartitionerClass(OrderInversionPartitioner.class);
```

### Hadoop的定制分区器实现

定制分区器必须确保包含相同左词（即自然键）的所有词对要发送到相同的归约器。例如，组合键{(man, tall), (man, strong), (man, moon), ...}都要发送到同一个归约器。要把这些键发送到相同的归约器，必须定义一个定制分区器，它只关心左词（在我们的例子中，左词就是man）。也就是说，这个分区器只根据左词的散列完成分区。参见示例5-2。

示例5-2：定制分区器实现：OrderInversionPartitioner

```
1 import org.apache.hadoop.io.IntWritable;
2 import org.apache.hadoop.mapreduce.Partitioner;
```

注1：我们将使用PairOfWords类表示一个词对（$W_i, W_j$）。方法PairOfWords.getLeftElement((Wi,Wj))返回$w_i$，方法PairOfWords.getRightElement((Wi,Wj))返回$w_j$。

```
3
4 public class OrderInversionPartitioner
5     extends Partitioner<PairOfWords, IntWritable> {
6
7     @Override
8     public int getPartition(PairOfWords key,
9                             IntWritable value,
10                            int numberOfPartitions) {
11        // key = (leftWord, rightWord)
12        String leftWord = key.getLeftElement();
13        return Math.abs(hash(leftWord) % numberOfPartitions);
14    }
15
16    // 由String.hashCode()改写
17    private static long hash(String str) {
18       long h = 1125899906842597L; // 质数
19       int length = str.length();
20       for (int i = 0; i < length; i++) {
21          h = 31*h + str.charAt(i);
22       }
23       return h;
24    }
25 }
```

## 相对频度映射器

这个映射器类发出一个单词邻域（也就是它之前的两个单词和之后的两个单词）的相对频度。例如，如果一个映射器接收到以下输入：

```
w1 w2 w3 w4 w5 w6
```

它会发出表5-5所示的键-值对。

**表5-5：映射器生成的键-值对**

| 键 | 值 |
|---|---|
| (w1, w2) | 1 |
| (w1, w3) | 1 |
| (w1, *) | 2 |
| | |
| (w2, w1) | 1 |
| (w2, w3) | 1 |
| (w2, w4) | 1 |
| (w2, *) | 3 |
| | |
| (w3, w1) | 1 |

表5-5：映射器生成的键-值对（续）

| 键 | 值 |
|---|---|
| (w3, w2) | 1 |
| (w3, w4) | 1 |
| (w3, w5) | 1 |
| (w3, *) | 4 |
| | |
| (w4, w2) | 1 |
| (w4, w3) | 1 |
| (w4, w5) | 1 |
| (w4, w6) | 1 |
| (w4, *) | 4 |
| | |
| (w5, w3) | 1 |
| (w5, w4) | 1 |
| (w5, w6) | 1 |
| (w5, *) | 3 |
| | |
| (w6, w4) | 1 |
| (w6, w5) | 1 |
| (w6, *) | 2 |

映射器算法如示例5-3所示。

示例5-3：映射器类：RelativeFrequencyMapper

```
1 public class RelativeFrequencyMapper ... {
2
3     private int neighborWindow = 2;
4     private PairOfWords pair = new PairOfWords();
5
6     public void setup(Context context) {
7         // 驱动器将设置"neighbor.window"
8         neighborWindow = context.getConfiguration().getInt("neighbor.window", 2);
9     }
10
11    // key 由系统生成，在这里忽略
12    // value是一个String（单词集）
13    public void map(Object key, String value) {
14       String[] tokens = StringUtils.split(value, " " );
15       if (tokens.length < 2) {
16         return;
17       }
```

```
18
19        for (int i = 0; i < tokens.length; i++) {
20           String word = tokens[i];
21           pair.setWord(word);
22           int start = (i - neighborWindow < 0) ? 0 : i - neighborWindow;
23           int end = (i + neighborWindow >= tokens.length) ?
24                 tokens.length - 1 : i + neighborWindow;
25           for (int j = start; j <= end; j++) {
26              if (i == j) {
27                 continue;
28              }
29              pair.setNeighbor(tokens[j]);
30              emit(pair, 1);
31              }
32              pair.setNeighbor("*");
33              int totalCount = end - start;
34              emit(pair, totalCount);
35           }
36        }
37 }
```

## 相对频度归约器

既然已经有了定制分区器和OI模式的实现，归约器接收的值将基于所有映射器生成的键的自然键。对于映射器阶段给出的例子，我们有6个归约器，键-值输入对如表5-6所示。

表5-6：归约器的输入

| 键 | 值 |
|---|---|
| (w1, *),(w1, w2),(w1, w3) | 2,1,1 |
| (w2, *),(w2, w1),(w2, w3),(w2, w4) | 3,1,1,1 |
| (w3, *),(w3, w1),(w3, w2),(w3, w4),(w3, w5) | 4,1,1,1,1 |
| (w4, *),(w4, w2),(w4, w3),(w4, w5),(w4, w6) | 4,1,1,1,1 |
| (w5, *),(w5, w3),(w5, w4),(w5, w6) | 3,1,1,1 |
| (w6, *),(w6, w4),(w6, w5) | 2,1,1 |

归约器算法如示例5-4所示。

示例5-4：归约器类：RelativeFrequencyReducer

```
1 public class RelativeFrequencyReducer ... {
2     private double totalCount = 0;
3     private String currentWord = "NOT_DEFINED";
4
5     protected void reduce(PairOfWords key, Iterable<Integer> values) {
6         if (key.getRight().equals("*")) {
7             if (key.getLeft().equals(currentWord)) {
8                 totalCount += getTotalCount(values);
9             }
10            else {
```

```
                currentWord = key.getLeft();
                totalCount = getTotalCount(values);
            }
        }
        else {
            int count = getTotalCount(values);
            double relativeCount = count / totalCount;
            emit(key, relativeCount);
        }

    }

    private int getTotalCount(Iterable<Integer> values) {
        int count = 0;
        for (Integer value : values) {
            count += value.get();
        }
        return count;
    }
}
```

## Hadoop实现类

在MapReduce/Hadoop中为相对频度问题实现OI设计模式时要使用表5-7中的类。

表5-7：Hadoop实现类

| 类名 | 类描述 |
|---|---|
| RelativeFrequencyDriver | 提交作业的驱动器 |
| RelativeFrequencyMapper | 定义map() |
| RelativeFrequencyReducer | 定义reduce() |
| RelativeFrequencyCombiner | 定义combine() |
| OrderInversionPartitioner | 定制分区器：如何对自然键分区 |
| PairOfWords | 表示词对（*Word1*, *Word2*） |

# 运行示例

## 输入

```
# hadoop fs -cat /order_inversion/input/sample_input.txt
java is a great language
java is a programming language
java is green fun language
java is great
programming with java is fun
```

## 运行MapReduce作业

下面给出运行示例的日志输出，为适应版面这里做了编辑，并且调整了格式：

```
# ./run.sh
...
Deleted hdfs://localhost:9000/lib/order_inversion.jar
Deleted hdfs://localhost:9000/order_inversion/output
...
14/01/04 14:56:56 INFO input.FileInputFormat:
Total input paths to process : 1
...
14/01/04 14:56:57 INFO mapred.JobClient:
Running job: job_201401041453_0002
14/01/04 14:56:58 INFO mapred.JobClient: map 0% reduce 0%
...
14/01/04 14:57:21 INFO mapred.JobClient: map 100% reduce 100%
14/01/04 14:57:21 INFO mapred.JobClient:
Job complete: job_201401041453_0002
...
14/01/04 14:57:21 INFO mapred.JobClient: Map-Reduce Framework
14/01/04 14:57:21 INFO mapred.JobClient: Map input records=5
14/01/04 14:57:21 INFO mapred.JobClient: Combine input records=85
14/01/04 14:57:21 INFO mapred.JobClient: Reduce input records=53
14/01/04 14:57:21 INFO mapred.JobClient: Reduce input groups=53
14/01/04 14:57:21 INFO mapred.JobClient: Combine output records=53
14/01/04 14:57:21 INFO mapred.JobClient: Reduce output records=44
14/01/04 14:57:21 INFO mapred.JobClient: Map output records=85
14/01/04 14:57:21 INFO RelativeFrequencyDriver:
Job Finished in milliseconds: 24869
```

## 生成的输出

```
# hadoop fs -cat /order_inversion/output/part*
(great, a)        0.2
(great, is)        0.4
(great, java)        0.2
(great, language)        0.2
(with, is)        0.3333333333333333
(with, java)        0.3333333333333333
(with, programming)        0.3333333333333333
(fun, green)        0.2
(fun, is)        0.4
(fun, java)        0.2
(fun, language)        0.2
(programming, a)        0.2
(programming, is)        0.2
(programming, java)        0.2
(programming, language)        0.2
(programming, with)        0.2
(a, great)        0.125
(a, is)        0.25
(a, java)        0.25
```

```
(a, language)       0.25
(a, programming)       0.125
(green, fun)        0.25
(green, is)        0.25
(green, java)        0.25
(green, language)        0.25
(is, a)        0.14285714285714285
(is, fun)        0.14285714285714285
(is, great)        0.14285714285714285
(is, green)        0.07142857142857142
(is, java)        0.35714285714285715
(is, programming)        0.07142857142857142
(is, with)        0.07142857142857142
(java, a)        0.16666666666666666
(java, fun)        0.08333333333333333
(java, great)        0.08333333333333333
(java, green)        0.08333333333333333
(java, is)        0.4166666666666667
(java, programming)        0.08333333333333333
(java, with)        0.08333333333333333
(language, a)        0.3333333333333333
(language, fun)        0.16666666666666666
(language, great)        0.16666666666666666
(language, green)        0.16666666666666666
(language, programming)        0.16666666666666666
```

这一章介绍了反转排序设计模式（用来控制归约器值的顺序），并使用MapReduce/Hadoop来实现。下一章将采用MapReduce范式实现移动平均算法（用于时间序列数据分析）。

第6章

# 移动平均

这一章将介绍MapReduce/Hadoop中的移动平均解决方案。在给出MapReduce解决方案之前，先来看移动平均（moving average）的基本概念。不过，首先需要理解时间序列数据。时间序列数据表示一个变量在一段时间内的值，如1秒、1分钟、1小时、1天、1周、1月、1季度或1年。可以不太严格地把时间序列数据形式化表示为三元组序列：

```
(k, t, v)
```

这里k是键（如股票代码），t是时间（小时、分钟或秒），v是相关联的值（如一只股票在时间点t的值）。一般地，只要在一段时间内记录相同的度量值，就会得到时间序列数据。例如，一个公司股票的收盘价就是基于分钟、小时或天的时间序列数据。多个连续周期的时间序列数据平均值（按相同时间间隔得到的观察值，如每小时一次或每天一次）称为移动平均。之所以称之为移动，这是因为随着新的时间序列数据的到来，要不断重新计算这个平均值，由于会删除最早的值同时增加最新的值，这个平均值会相应地“移动”。

## 示例1：时间序列数据（股票价格）

考虑表6-1所示的数据，这是一个名为MY-STOCK的公司的股票收盘价（注意这是一个虚构的股票代码）。

表6-1：MY-STOCK收盘价时间序列数据

| 时间序列 | 日期 | 收盘价 |
|---|---|---|
| 1 | 2013-10-01 | 10 |
| 2 | 2013-10-02 | 18 |
| 3 | 2013-10-03 | 20 |
| 4 | 2013-10-04 | 30 |
| 5 | 2013-10-07 | 24 |
| 6 | 2013-10-08 | 33 |
| 7 | 2013-10-09 | 27 |
| ... | ... | ... |

要计算3天的移动平均数，可以得到表6-2所示的输出。

表6-2：MY-STOCK收盘价3天的移动平均数

| 时间序列 | 日期 | 移动平均 | 如何计算 |
|---|---|---|---|
| 1 | 2013-10-01 | 10.00 | =(10)/(1) |
| 2 | 2013-10-02 | 14.00 | =(10+18)/(2) |
| 3 | 2013-10-03 | 16.00 | =(10+18+20)/(3) |
| 4 | 2013-10-04 | 22.66 | =(18+20+30)/(3) |
| 5 | 2013-10-07 | 24.66 | =(20+30+24)/(3) |
| 6 | 2013-10-08 | 29.00 | =(30+24+33)/(3) |
| 7 | 2013-10-09 | 28.00 | =(24+33+27)/(3) |

# 示例2：时间序列数据（URL访问数）

第二个例子是计算一个特定时间窗口内各个日期访问不同URL的不同访问者人数的移动平均数。我们将使用表6-3所示的输入。

表6-3：URL访问数的时间序列数据

| URL | 日期 | 不同访问者人数 |
|---|---|---|
| URL1 | 2013-10-01 | 400 |
| URL1 | 2013-10-02 | 200 |
| URL1 | 2013-10-03 | 300 |
| URL1 | 2013-10-04 | 700 |
| URL1 | 2013-10-05 | 800 |
| URL2 | 2013-10-01 | 10 |

表6-3：URL访问数的时间序列数据（续）

| URL | 日期 | 不同访问者人数 |
|---|---|---|
| URL2 | 2013-10-02 | 20 |
| URL2 | 2013-10-03 | 30 |
| URL2 | 2013-10-04 | 70 |

要计算3天的移动平均数，可以得到表6-4所示的输出。

表6-4：3天的URL访问数的移动平均数

| URL | 日期 | 移动平均数 |
|---|---|---|
| URL1 | 2013-10-01 | 400 |
| URL1 | 2013-10-02 | 300 |
| URL1 | 2013-10-03 | 300 |
| URL1 | 2013-10-04 | 400 |
| URL1 | 2013-10-05 | 600 |
| URL2 | 2013-10-01 | 10 |
| URL2 | 2013-10-02 | 15 |
| URL2 | 2013-10-03 | 20 |
| URL2 | 2013-10-04 | 40 |

# 形式定义

令A为一组有序对象的序列：

$$A = (a_1, a_2, a_3, \ldots, a_N)$$

可以把A表示为：

$$\{a_i\}_{i=1}^{N}$$

$n$移动平均序列是由$a_i$定义的一个新序列

$$\{s_i\}_{i=1}^{N-n+1}$$

这是通过计算$n$项子序列的算术平均值来得到的：

$$s_i = \frac{1}{n}\sum_{j=i}^{i+n-1} a_j$$

所以$n$移动平均序列$S_n$计算如下：

$$S_2 = \frac{1}{2}[(a_1 + a_2),(a_2 + a_3),....,(a_{n-1} + a_n)]$$

$$S_3 = \frac{1}{3}[(a_1 + a_2 + a_3),(a_2 + a_3 + a_4),....,(a_{n-2} + a_{n-1} + a_n)]$$

$$S_4 = \frac{1}{4}[(a_1 + a_2 + a_3 + a_4),(a_2 + a_3 + a_4 + a_5),....,(a_{n-3} + a_{n-2} + a_{n-1} + a_n] \ ...$$

# POJO移动平均解决方案

要解决移动平均问题，这里将提供两个简单Java对象（plain old Java object，POJO）解决方案：

- 解决方案1: 使用java.util.Queue。
- 解决方案2: 使用数组模拟队列。

在这两个解决方案中，我们将窗口实现为一个队列数据结构，以一种先进先出（first-in-first-out，FIFO）的方式填入时间序列数据点，直到其中包含N个数据点（这N个点的平均值就是移动平均数）。

## 解决方案1: 使用队列

这个解决方案如示例6-1所示，这里使用了java.util.Queue实现的一个队列数据结构。任何时候都要确保队列中只有必要的项（移动平均窗口的大小）。

示例6-1：SimpleMovingAverage类

```
1 import java.util.Queue;
2 import java.util.LinkedList;
3 public class SimpleMovingAverage {
4
5     private double sum = 0.0;
6     private final int period;
7     private final Queue<Double> window = new LinkedList<Double>();
8
9     public SimpleMovingAverage(int period) {
10        if (period < 1) {
11           throw new IllegalArgumentException("period must be > 0");
12        }
13        this.period = period;
14    }
15
16    public void addNewNumber(double number) {
17        sum += number;
18        window.add(number);
19        if (window.size() > period) {
20            sum -= window.remove();
```

```
        }
    }

    public double getMovingAverage() {
        if (window.isEmpty()) {
            throw new IllegalArgumentException("average is undefined");
        }
        return sum / window.size();
    }
}
```

## 解决方案2：使用数组

这个解决方案如示例6-2所示，这里使用一个简单数组（名为`window`）模拟入队和出队操作。这个解决方案更为高效，但不那么直观。

示例6-2：SimpleMovingAverageUsingArray类

```
public class SimpleMovingAverageUsingArray {

    private double sum = 0.0;
    private final int period;
    private double[] window = null;
    private int pointer = 0;
    private int size = 0;

    public SimpleMovingAverageUsingArray(int period) {
        if (period < 1) {
           throw new IllegalArgumentException("period must be > 0");
        }
        this.period = period;
        window = new double[period];
    }

    public void addNewNumber(double number) {
        sum += number;
        if (size < period) {
            window[pointer++] = number;
            size++;
        }
        else {
            // size = period (size cannot be > period)
            pointer = pointer % period;
            sum -= window[pointer];
            window[pointer++] = number;
        }
    }

    public double getMovingAverage() {
        if (size == 0) {
            throw new IllegalArgumentException("average is undefined");
        }
        return sum / size;
    }
```

```
37 }
```

## 测试移动平均

示例6-3给出了一个程序，用来测试使用SimpleMovingAverage类的移动平均算法。这里分别用两个窗口大小{3, 4}进行测试。

示例6-3：测试SimpleMovingAverage

```
 1 import org.apache.log4j.Logger;
 2 public class TestSimpleMovingAverage {
 3     private static final Logger THE_LOGGER =
 4         Logger.getLogger(TestSimpleMovingAverage.class);
 5     public static void main(String[] args) {
 6         // 时间序列1 2 3 4 5 6 7
 7         double[] testData = {10, 18, 20, 30, 24, 33, 27};
 8         int[] allWindowSizes = {3, 4};
 9         for (int windowSize : allWindowSizes) {
10             SimpleMovingAverage sma = new SimpleMovingAverage(windowSize);
11             THE_LOGGER.info("windowSize = " + windowSize);
12             for (double x : testData) {
13                 sma.addNewNumber(x);
14                 THE_LOGGER.info("Next number = " + x + ", SMA = " +
15                                 sma.getMovingAverage());
16             }
17             THE_LOGGER.info("---");
18         }
19     }
20 }
```

## 运行示例

```
# javac SimpleMovingAverage.java
# javac TestSimpleMovingAverage.java
# java TestSimpleMovingAverage
13/10/10 22:24:08 INFO TestSimpleMovingAverage: windowSize = 3
13/10/10 22:24:08 INFO TestSimpleMovingAverage: Next number = 10.0, SMA = 10.00
13/10/10 22:24:08 INFO TestSimpleMovingAverage: Next number = 18.0, SMA = 14.00
13/10/10 22:24:08 INFO TestSimpleMovingAverage: Next number = 20.0, SMA = 16.00
13/10/10 22:24:08 INFO TestSimpleMovingAverage: Next number = 30.0, SMA = 22.66
13/10/10 22:24:08 INFO TestSimpleMovingAverage: Next number = 24.0, SMA = 24.66
13/10/10 22:24:08 INFO TestSimpleMovingAverage: Next number = 33.0, SMA = 29.00
13/10/10 22:24:08 INFO TestSimpleMovingAverage: Next number = 27.0, SMA = 28.00
13/10/10 22:24:08 INFO TestSimpleMovingAverage: ------
13/10/10 22:24:08 INFO TestSimpleMovingAverage: windowSize = 4
13/10/10 22:24:08 INFO TestSimpleMovingAverage: Next number = 10.0, SMA = 10.00
13/10/10 22:24:08 INFO TestSimpleMovingAverage: Next number = 18.0, SMA = 14.00
13/10/10 22:24:08 INFO TestSimpleMovingAverage: Next number = 20.0, SMA = 16.00
13/10/10 22:24:08 INFO TestSimpleMovingAverage: Next number = 30.0, SMA = 19.50
13/10/10 22:24:08 INFO TestSimpleMovingAverage: Next number = 24.0, SMA = 23.00
13/10/10 22:24:08 INFO TestSimpleMovingAverage: Next number = 33.0, SMA = 26.75
13/10/10 22:24:08 INFO TestSimpleMovingAverage: Next number = 27.0, SMA = 28.50
13/10/10 22:24:08 INFO TestSimpleMovingAverage: ------
```

# MapReduce/Hadoop移动平均解决方案

我们已经了解了时间序列数据移动平均的概念，下面重点讨论解决移动平均问题的MapReduce解决方案。具体实现使用Apache Hadoop提供。可以看到，在解决大量时间序列数据（如很长一段时间的股价）的移动平均问题时，MapReduce是一个非常强大的框架。移动平均的计算由reduce()函数处理，在这里可以得到一个特定时间序列的所有所需数据。

## 输入

这个MapReduce解决方案的输入采用以下格式：

```
<name-as-string><,><date-as-timestamp><,><value-as-double>
```

在这个简单例子中，我提供了3个公司/股票代码（GOOG、AAPL和IBM）的股价。第一列是股票代码，第二列是时间戳，第三列是当天的已调整收盘价：

```
GOOG,2004-11-04,184.70
GOOG,2004-11-03,191.67
GOOG,2004-11-02,194.87
AAPL,2013-10-09,486.59
AAPL,2013-10-08,480.94
AAPL,2013-10-07,487.75
AAPL,2013-10-04,483.03
AAPL,2013-10-03,483.41
IBM,2013-09-30,185.18
IBM,2013-09-27,186.92
IBM,2013-09-26,190.22
IBM,2013-09-25,189.47
GOOG,2013-07-19,896.60
GOOG,2013-07-18,910.68
GOOG,2013-07-17,918.55
```

## 输出

这个MapReduce解决方案的输出有以下格式：

```
<name-as-string><,><date-as-timestamp><,><moving-average-as-double>
```

下面给出一些示例：

```
Name: GOOG, Date: 2013-07-17 Moving Average: 912.52
Name: GOOG, Date: 2013-07-18 Moving Average: 912.01
Name: GOOG, Date: 2013-07-19 Moving Average: 916.39
```

既然已经了解了移动平均算法，现在只需要根据股票代码对数据分组，然后按时间戳对这些值排序，最后应用移动平均算法。对时间序列数据排序至少有两种方法：

*方案1*

对于各个归约器键，在内存中（RAM）对时间序列数据排序。这个方法存在一个问题：如果没有足够的RAM来完成归约器的排序操作，这种方法就不可行。

*方案2*

让MapReduce框架完成时间序列数据的排序（MapReduce框架的主要特性之一就是按键值排序和分组，Hadoop很擅长这个工作）。与方案1相比，这个方案可伸缩性要好得多，排序由MapReduce框架的`sort()`和`shuffle()`函数完成。如果采用这个方案，我们需要修改键－值对，并编写一些定制插件类来完成二次排序。

我会给出这两种方法的解决方案。

# 方案1：在内存中排序

映射器接受一个输入行，如：

```
<name-as-string><,><timestamp><,><value-as-double>
```

然后发出键-值对，其中键是`<name-as-string>`，值是`<timestamp><,><value-as-double>`。归约器会接收键-值对，其中键是`<name-as-string>`，值是一个无序的`<timestamp><,><value-as-double>`列表。

## 时间序列数据

时间序列数据表示为一个`TimeSeriesData`对象，如示例6-4所示。这个类实现了`Writable`（因为这些对象会在Hadoop中持久存储）和`Comparable<TimeSeriesData>`（因为我们将对这些对象完成内存中排序）。

示例6-4：TimeSeriesData类

```
 1 import org.apache.hadoop.io.Writable;
 2 ...
 3 /**
 4  *
 5  * TimeSeriesData 表示
 6  * 一个(time-series-timestamp, time-series-value)对。
 7  *
 8  */
 9 public class TimeSeriesData
10    implements Writable, Comparable<TimeSeriesData> {
11
12     private long timestamp;
13     private double value;
14
15     public static TimeSeriesData copy(TimeSeriesData tsd) {
16        return new TimeSeriesData(tsd.timestamp, tsd.value);
17     }
18
```

```
19     public TimeSeriesData(long timestamp, double value) {
20        set(timestamp, value);
21     }
22     ...
23 }
```

## 映射器函数

map()函数的定义参见示例6-5，这个函数会发出时间序列数据。

示例6-5：映射器函数

```
1 /**
2  * @param key为<name-as-string>
3  * @param value为<timestamp><,><value-as-double>
4  */
5 map(key, value) {
6    TimeSeriesData timeseries =
7       new TimeSeriesData(value.timestamp, value.value-as-double);
8    emit(key, timeseries);
9 }
```

## 归约器函数

reduce()函数将聚集时间序列数据，并发出每个键的移动平均数，这个函数的定义如示例6-6所示。

示例6-6：归约器函数

```
 1 public class MovingAverageSortInRAM_Reducer {
 2
 3     private int windowSize = 4; // 默认
 4
 5     /**
 6      * 归约器任务开始时调用一次
 7      */
 8     setup() {
 9        // "moving.average.window.size" 由驱动器设置
10       // configuration是MRF的配置对象
11       windowSize = configuration.get("moving.average.window.size");
12     }
13
14     /**
15      * @param key为<name-as-string>
16      * @param value为List<TimeSeriesData>
17      * 这里TimeSeriesData表示一个(timestamp, value)对
18      */
19     reduce(key, values) {
20        List<TimeSeriesData> sortedTimeSeries = sort(values);
21          // 调用 movingAverage(sortedTimeSeries, windowSize)并发出输出
22          // 对sortedTimeSeries应用移动平均算法
23          double sum = 0.0;
24          // 计算前缀和
25          for (int i=0; i < windowSize-1; i++) {
```

```
            sum += sortedTimeSeries.get(i).getValue();
        }

        // 现在我们有足够的时间序列数据来计算移动平均
        for (int i = windowSize-1; i < sortedTimeSeries.size(); i++) {
            sum += sortedTimeSeries.get(i).getValue();
            double movingAverage = sum / windowSize;
            long timestamp = sortedTimeSeries.get(i).getTimestamp();
            Text outputValue = timestamp+ "," + movingAverage;
            // 将输出发送到分布式文件系统
            emit(key, outputValue);

           // 准备下一次迭代
           sum -= sortedTimeSeries.get(i-windowSize+1).getValue();
        }

    }
}
```

## Hadoop实现类

这个Hadoop实现包括表6-5所示的类。

**表6-5：Hadoop移动平均实现中的类**

| 类名 | 描述 |
|---|---|
| SortInMemory_MovingAverageDriver | 提交Hadoop作业的驱动器程序 |
| SortInMemory_MovingAverageMapper | 定义map() |
| SortInMemory_MovingAverageReducer | 定义reduce() |
| TimeSeriesData | 将时间序列数据点表示为一个（timestamp, double）对 |
| DateUtil | 提供基本数据转换工具 |
| HadoopUtil | 提供基本Hadoop工具 |

# 运行示例

下面各小节给出这个运行示例的输入、脚本和期望输出。

## 输入

```
# hadoop fs -cat /moving_average/sort_in_memory/input/*
GOOG,2004-11-04,184.70
GOOG,2004-11-03,191.67
GOOG,2004-11-02,194.87
AAPL,2013-10-09,486.59
AAPL,2013-10-08,480.94
AAPL,2013-10-07,487.75
AAPL,2013-10-04,483.03
AAPL,2013-10-03,483.41
IBM,2013-09-30,185.18
```

```
IBM,2013-09-27,186.92
IBM,2013-09-26,190.22
IBM,2013-09-25,189.47
GOOG,2013-07-19,896.60
GOOG,2013-07-18,910.68
GOOG,2013-07-17,918.55
```

## 脚本

```
# cat ./run.sh
#/bin/bash
export HADOOP_HOME=/usr/local/hadoop-2.3.0
export HADOOP_HOME_WARN_SUPPRESS=true
export JAVA_HOME=/usr/java/jdk6
export PATH=$PATH:$HADOOP_HOME/bin:$JAVA_HOME/bin
BOOK_HOME=/mp/data-algorithms-book
APP_JAR=$BOOK_HOME/dist/data_algorithms_book.jar
CLASSPATH=.:$HADOOP_HOME/conf:$APP_JAR
export INPUT=/moving_average/sort_in_memory/input
export OUTPUT=/moving_average/sort_in_memory/output
$HADOOP_HOME/bin/hadoop fs -put $APP_JAR /lib/
$HADOOP_HOME/bin/hadoop fs -rmr $OUTPUT
driver=org.dataalgorithms.chap06.memorysort.SortInMemory_MovingAverageDriver
export WINDOW_SIZE=2
$HADOOP_HOME/bin/hadoop jar $JAR $driver $WINDOW_SIZE $INPUT $OUTPUT
```

## 运行示例日志

为适应版面，这里对输出做了编辑，并对格式有所调整。

```
# ./run.sh
...
Deleted hdfs://localhost:9000/moving_average/sort_in_memory/output
args[0]: <window_size>=2
args[1]: <input>=/moving_average/sort_in_memory/input
args[2]: <output>=/moving_average/sort_in_memory/output
...
14/05/12 15:07:03 INFO mapred.JobClient: Running job: job_201405121349_0009
14/05/12 15:07:04 INFO mapred.JobClient: map 0% reduce 0%
...
14/05/12 15:07:54 INFO mapred.JobClient: map 100% reduce 100%
14/05/12 15:07:55 INFO mapred.JobClient: Job complete: job_201405121349_0009
...
14/05/12 15:07:55 INFO mapred.JobClient: Map-Reduce Framework
14/05/12 15:07:55 INFO mapred.JobClient: Map input records=15
14/05/12 15:07:55 INFO mapred.JobClient: Spilled Records=30
14/05/12 15:07:55 INFO mapred.JobClient: Map output bytes=311
14/05/12 15:07:55 INFO mapred.JobClient: Combine input records=0
14/05/12 15:07:55 INFO mapred.JobClient: Reduce input records=15
14/05/12 15:07:55 INFO mapred.JobClient: Reduce input groups=3
14/05/12 15:07:55 INFO mapred.JobClient: Combine output records=0
14/05/12 15:07:55 INFO mapred.JobClient: Reduce output records=12
14/05/12 15:07:55 INFO mapred.JobClient: Map output records=15
```

### 检查输出

```
# hadoop fs -cat /moving_average/sort_in_memory/output/part*
GOOG 2004-11-03,193.26999999999998
GOOG 2004-11-04,188.18499999999997
GOOG 2013-07-17,551.625
GOOG 2013-07-18,914.615
GOOG 2013-07-19,903.6400000000001
AAPL 2013-10-04,483.22
AAPL 2013-10-07,485.39
AAPL 2013-10-08,484.345
AAPL 2013-10-09,483.765
IBM 2013-09-26,189.845
IBM 2013-09-27,188.57
IBM 2013-09-30,186.05
```

## 方案2: 使用MapReduce框架排序

在上一节中，你已经看到如何使用MapReduce框架解决移动平均问题，首先按name对大量时间序列数据分组，再在内存中（按timestamp）对时间序列值排序。不过，在内存中完成数据排序存在一个问题：如果有大量数据，可能无法完全放在内存中（除非所有集群节点上都有大量RAM，而目前可能做不到这一点，因为内存还不像硬盘那么廉价）。在这一节中，我会介绍如何使用MapReduce框架（MRF）完成数据排序，而不用在内存中排序。例如，在Hadoop中，可以在MRF的洗牌阶段对数据排序。MRF通过洗牌完成排序时，这称为“二次排序”（在第1章和第2章已经了解）。现在的问题是如何完成这个二次排序。我们要编写另外一些定制Java类，然后插入到MRF中。

如何插入定制Java类利用MRF完成排序？首先，在驱动器类中建立示例6-7所示的类。

示例6-7：SortByMRF_MovingAverageDriver类

```
1 import ...
2 /**
3  * MapReduce作业：计算时间序列数据的移动平均
4  * 这里使用了MapReduce的二次排序（由shuffle函数排序）。
5  */
6 public class SortByMRF_MovingAverageDriver {
7    public static void main(String[] args) throws Exception {
8       Configuration conf = new Configuration();
9       JobConf jobconf = new JobConf(conf, SortByMRF_MovingAverageDriver.class);
10      ...
11      // 下面3个设置是二次排序的必要设置。
12      // 分区器根据映射器输出键确定
13      // 哪个映射器输出发送到哪个归约器。一般地，不同的键会在
14      // 不同的组中（对于归约器就是不同的迭代器）。不过，有时
15      // 我们希望不同的键在同一个组中。这种情况下要使用
16      // 输出值分组比较器，用来对映射器输出分组
17      // （类似SQL中的group by条件）。输出键比较器
18      // 在排序阶段用来比较映射器输出键
19      jobconf.setPartitionerClass(NaturalKeyPartitioner.class);
```

```
20        jobconf.setOutputKeyComparatorClass(CompositeKeyComparator.class);
21        jobconf.setOutputValueGroupingComparator(
22           NaturalKeyGroupingComparator.class);
23
24        JobClient.runJob(jobconf);
25     }
26 }
```

要为移动平均实现二次排序，映射器的输出键应当是自然键（name-as-string）和自然值（timeseries-timestamp）的一个组合。图6-1展示了所需的自然键和组合键。

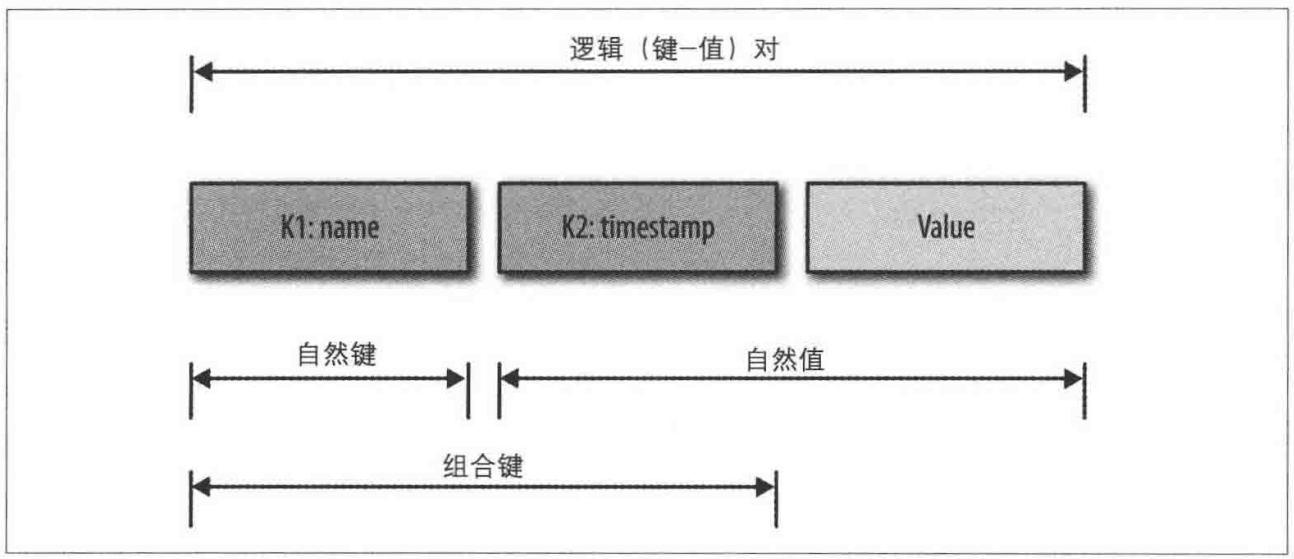



图6-1：组合键和自然键

定制CompositeKey（由"name"和"timestamp"组成）的定义如示例6-8所示。

示例6-8：CompositeKey类定义

```
1 import org.apache.hadoop.io.WritableComparable;
2 ...
3 public class CompositeKey
4    implements WritableComparable<CompositeKey> {
5    // 组合键是一个(name, timestamp)对
6    private String name;
7    private long timestamp;
8    ...
9 }
```

CompositeKey类必须实现WritableComparable接口，因为这些对象将在HDFS中持久存储。因此，定制CompositeKey类会在洗牌阶段中为Hadoop提供必要的信息，根据两个字段（"name" 和"timestamp"）完成排序，而不只是一个"name"字段。不过，如何对CompositeKey对象排序呢？为此需要提供一个类来比较组合键对象。这个比较由CompositeKeyComparator类完成（它的主要功能就是提供compare()方法），如示例6-9所示。

示例6-9：CompositeKeyComparator类定义

```
1 import org.apache.hadoop.io.WritableComparable;
2 import org.apache.hadoop.io.WritableComparator;
3 public class CompositeKeyComparator extends WritableComparator {
4    protected CompositeKeyComparator() {
5        super(CompositeKey.class, true);
6    }
7
8    @Override
9    public int compare(WritableComparable w1, WritableComparable w2) {
10       CompositeKey key1 = (CompositeKey) w1;
11       CompositeKey key2 = (CompositeKey) w2;
12
13       int comparison = key1.getName().compareTo(key2.getName());
14       if (comparison == 0) {
15            // name相同
16            if (key1.getTimestamp() == key2.getTimestamp()) {
17                return 0;
18            }
19            else if (key1.getTimestamp() < key2.getTimestamp()) {
20                return -1;
21            }
22            else {
23                return 1;
24            }
25       }
26       else {
27           return comparison;
28       }
29   }
30 }
```

WritableComparator是完成WritableComparable比较的一个Comparator。这是一个基实现，使用了自然顺序。如果要定义其他顺序，必须覆盖compare(WritableComparable,WritableComparable)方法。所以，CompositeKey对象的排序由定制CompositeKeyComparator类处理。

既然已经知道如何对组合键排序，下一个问题是数据如何到达归约器。这由另一个定制类NaturalKeyPartitioner处理，它实现了Partitioner接口[注1]，会对所有映射器生成的键空间分区。尽管键按完整的CompositeKey（由name和timestamp组成）排序，但分区时只使用name，因为我们希望有相同name的所有待排序值都到达同一个归约器。

示例6-10给出了这个定制分区器NaturalKeyPartitioner的代码。

---

注1：org.apache.hadoop.mapred.Partitioner。Partitioner控制中间映射输出键的分区。键（或键子集）要用来得出分区，通常由一个散列函数完成。分区的总数与完成这个作业的归约任务数相同。定制分区器根据归约器数划分数据，使得一个分区中的所有数据都由一个归约器执行（资料来源：*http://hadoop.apache.org/docs*）。

示例6-10：NaturalKeyPartitioner类定义

```
import org.apache.hadoop.mapred.JobConf;
import org.apache.hadoop.mapred.Partitioner;
public class NaturalKeyPartitioner implements
    Partitioner<CompositeKey, TimeSeriesData> {

    @Override
    public int getPartition(CompositeKey key,
                            TimeSeriesData value,
                            int numberOfPartitions) {
        return Math.abs((int) (hash(key.getName()) % numberOfPartitions));
    }

    @Override
    public void configure(JobConf jobconf) {
    }

    /**
     * 由String.hashCode()改写
     */
    static long hash(String str) {
        long h = 1125899906842597L; // 质数
        int length = str.length();
        for (int i = 0; i < length; i++) {
           h = 31*h + str.charAt(i);
        }
        return h;
     }
}
```

示例6-11给出了我们需要的最后一个插件类NaturalKeyGroupingComparator，在Hadoop的洗牌阶段，将用这个类按键的自然键部分对组合键分组（在我们的例子中，就是按name部分分组）。

示例6-11：NaturalKeyGroupingComparator类定义

```
import org.apache.hadoop.io.WritableComparable;
import org.apache.hadoop.io.WritableComparator;
/**
 *
 * NaturalKeyGroupingComparator
 *
 * 这个类在Hadoop的洗牌阶段用来
 * 按自然键对组合键分组。
 * 这个时间序列数据的自然键是name。
 *
 */
public class NaturalKeyGroupingComparator extends WritableComparator {
   protected NaturalKeyGroupingComparator() {
      super(CompositeKey.class, true);
   }

   @Override
   public int compare(WritableComparable w1, WritableComparable w2) {
```

```
19          CompositeKey key1 = (CompositeKey) w1;
20          CompositeKey key2 = (CompositeKey) w2;
21          // 自然键为: key1.getName()和key2.getName()
22          return key1.getName().compareTo(key2.getName());
23      }
24  }
```

### Hadoop实现类

实现移动平均算法的类如表6-6所示，这里使用MRF对归约器值排序。

**表6-6：MRF中实现移动平均算法使用的类**

| 类名 | 描述 |
|---|---|
| MovingAverage | 一个简单的移动平均算法 |
| SortByMRF_MovingAverageDriver | 提交Hadoop作业的驱动器程序 |
| SortByMRF_MovingAverageMapper | 定义map() |
| SortByMRF_MovingAverageReducer | 定义reduce() |
| TimeSeriesData | 将时间序列数据点表示为一个（timestamp, double）对 |
| CompositeKey | 定义一个定制组合键（string, timestamp） |
| CompositeKeyComparator | 定义CompositeKey的排序顺序 |
| NaturalKeyPartitioner | 映射阶段的数据输出发送到洗牌阶段之前先分区 |
| NaturalKeyGroupingComparator | 在Hadoop的洗牌阶段按键的第一部分（自然键部分）对组合键分组 |
| DateUtil | 提供基本日期转换工具 |
| HadoopUtil | 提供基本Hadoop工具 |

## 运行示例

下面各小节会提供一个运行示例，包括输入、脚本和期望的输出。

### HDFS输入

```
# hadoop fs -cat /moving_average/sort_by_mrf/input/*
GOOG,2004-11-04,184.70
GOOG,2004-11-03,191.67
GOOG,2004-11-02,194.87
AAPL,2013-10-09,486.59
AAPL,2013-10-08,480.94
AAPL,2013-10-07,487.75
AAPL,2013-10-04,483.03
AAPL,2013-10-03,483.41
IBM,2013-09-30,185.18
IBM,2013-09-27,186.92
IBM,2013-09-26,190.22
IBM,2013-09-25,189.47
```

```
GOOG,2013-07-19,896.60
GOOG,2013-07-18,910.68
GOOG,2013-07-17,918.55
```

## 脚本

```
# cat ./run.sh
#/bin/bash
export HADOOP_HOME=/usr/local/hadoop-2.3.0
export HADOOP_HOME_WARN_SUPPRESS=true
export JAVA_HOME=/usr/java/jdk6
export PATH=$PATH:$HADOOP_HOME/bin:$JAVA_HOME/bin
BOOK_HOME=/mp/data-algorithms-book
APP_JAR=$BOOK_HOME/dist/data_algorithms_book.jar
CLASSPATH=.:$HADOOP_HOME/conf:$APP_JAR
$HADOOP_HOME/bin/hadoop fs -put $APP_JAR /lib/
export input=/moving_average/sort_by_mrf/input
export output=/moving_average/sort_by_mrf/output
$HADOOP_HOME/bin/hadoop fs -rmr $output
driver=org.dataalgorithms.chap06.secondarysort.SortByMRF_MovingAverageDriver
export window_size=2
$HADOOP_HOME/bin/hadoop jar $APP_JAR $driver $window_size $input $output
```

## 运行示例日志

为适应版面，这里对输出做了编辑，并对格式有所调整：

```
1 # ./run.sh
2 ...
3 Deleted hdfs://localhost:9000/moving_average/sort_by_mrf/output
4 ...
5 14/05/12 16:15:29 INFO mapred.JobClient: Running job: job_201405121612_0002
6 14/05/12 16:15:30 INFO mapred.JobClient: map 0% reduce 0%
7 ...
8 14/05/12 16:16:32 INFO mapred.JobClient: map 100% reduce 100%
9 14/05/12 16:16:33 INFO mapred.JobClient: Job complete: job_201405121612_0002
10 ...
11 14/05/12 16:16:33 INFO mapred.JobClient: Map-Reduce Framework
12 14/05/12 16:16:33 INFO mapred.JobClient: Map input records=15
13 14/05/12 16:16:33 INFO mapred.JobClient: Spilled Records=30
14 ...
15 14/05/12 16:16:33 INFO mapred.JobClient: Reduce input records=15
16 14/05/12 16:16:33 INFO mapred.JobClient: Reduce input groups=3
17 14/05/12 16:16:33 INFO mapred.JobClient: Combine output records=0
18 14/05/12 16:16:33 INFO mapred.JobClient: Reduce output records=15
19 14/05/12 16:16:33 INFO mapred.JobClient: Map output records=15
```

## 生成的输出

```
1 # hadoop fs -cat /moving_average/sort_by_mrf/output/part*
2 GOOG 2004-11-02,194.87
3 GOOG 2004-11-03,193.26999999999998
4 GOOG 2004-11-04,188.185
```

```
5 GOOG 2013-07-17,551.625
6 GOOG 2013-07-18,914.6149999999999
7 GOOG 2013-07-19,903.64
8 AAPL 2013-10-03,483.41
9 AAPL 2013-10-04,483.22
10 AAPL 2013-10-07,485.39
11 AAPL 2013-10-08,484.345
12 AAPL 2013-10-09,483.765
13 IBM 2013-09-25,189.47
14 IBM 2013-09-26,189.845
15 IBM 2013-09-27,188.57
16 IBM 2013-09-30,186.04999999999995
```

这一章使用MapReduce框架提供了移动平均算法的一个基本实现。我们还在实现中回顾了第1章和第2章介绍的二次排序设计模式。下一章将使用MapReduce/Hadoop和Spark实现购物篮分析（一个数据挖掘算法）。

第7章

# 购物篮分析

购物篮分析（Market Basket Analysis，MBA）是一个流行的数据挖掘技术，市场营销和电子商务专业人员经常用这个技术来揭示不同商品或商品组之间的相似度。数据挖掘的一般目标是从庞大的数据集合中提取有趣的关联信息，例如数百万超市或信用卡销售交易。购物篮分析可以帮助我们找出很可能会一起购买的商品，关联规则挖掘会发现一个交易集中商品之间的相关性。然后市场营销人员可以使用这些关联规则在商店货架上或在线将相关的商品摆放在相邻的位置，使顾客能购买更多的商品。为购物篮分析挖掘关联规则时要找出频繁商品集，这是一个计算密集型问题，所以非常适合利用MapReduce来解决。

这一章将提供两个购物篮分析解决方案：

- 对应N阶元组（N = 1, 2, 3, ...）的MapReduce/Hadoop解决方案。这个解决方案可以查找频繁模式。
- Spark解决方案，不仅可以找出频繁模式，还会为它们生成关联规则。

## MBA目标

这一章为数据挖掘分析提供一个MapReduce解决方案，来查找一个给定超市或网店购物篮中最常出现的商品对（阶数为1, 2, ...）。我们的MapReduce解决方案可以扩展为查找购物篮中最常出现的N阶商品`TupleN`（其中`N` = 1, 2, 3, ...）。“阶数N”（作为一个整数）将作为参数传递到MapReduce驱动器，驱动器再在Hadoop的`Configuration`对象中设置这个参数。最后，`map()`方法在`setup()`方法中从Hadoop的`Configuration`对象读取这个参数。一旦得到最频繁项集$F_i$ ($i$ = 1, 2, ...)，可以用它们生成交易的一个关联规则。例如，如果有5个商品`{A, B, C, D, E}`，对应以下6个交易：

```
Transaction 1: A, C
Transaction 2: B, D
Transaction 3: A, C, E
Transaction 4: C, E
Transaction 5: A, B, E
Transaction 6: B, E
```

我们的目标是构建项集$F_1$（大小= 1）和$F_2$（大小= 2）：

- $F_1$ = {[C, 3], [A, 3], [B, 3], [E, 4]}
- $F_2$ = {[<A,C>, 2], [<C,E>, 2], [<A,E>, 2], [<B,E>, 2]}

需要说明，在这个例子中，我们使用的最小支持度为2，支持度是一个模式在整个交易集中出现的次数。因此，要从$F_1$中去除 [D, 1] 。项集$F_1$和$F_2$可以用来生成交易的关联规则。关联规则形式：

```
LHS => RHS
可乐 => 薯片
如果顾客购买可乐（称为LHDS-左件），
他们也会购买薯片（称为RHS-右件）。
```

在数据挖掘中，关联规则有两个度量标准：

*支持度（Support）*
: 一个项集的出现频度。例如，Support({A,C}) = 2表示项A和C只在两个交易中一起出现。

*置信度（Confidence）*
: 关联规则左件与右件共同出现的频繁程度。

利用购物篮分析，我们可以通过回答以下问题来了解购物者的行为：

- 哪些商品会一起购买？
- 每个购物篮里有哪些商品？
- 哪些商品应当降价销售？
- 商品应当如何相邻摆放以实现最大利润？例如，如果一个超市有10000或更多不同的商品，那么就有5000万种两项商品组合，1000亿种三项商品组合。
- 如何确定一个电子商务网站的商品目录？

为了理解MBA的主要思想，下面假设一家超市收银台前有一个购物车，其中装满了一个顾客购买的商品。这个购物篮里包含下面这些商品：金枪鱼、牛奶、橙汁、香蕉、鸡蛋、牙膏、门窗清洁器和清洁剂。这个购物篮会告诉我们一个顾客一次购买的商品。不过，所有顾客购买的所有商品的一个完整清单（超市的所有交易）提供的信息会比这多

得多。它会描述一个零售企业最重要的部分，顾客在购买什么商品以及何时购买。因此，MBA的一个目的就是帮助市场营销人员确定哪些商品应当在货架上相邻摆放，以及如何设计电子商务网站商品目录中商品的布局来促进销售。MBA的最终目标是自动生成关联规则。

# MBA的应用领域

购物篮分析适用于超市的购物者，不过还有很多其他领域也可以应用MBA。这包括：

- 信用卡交易分析。
- 电话呼叫模式分析。
- 医疗保险欺诈识别（考虑哪些情况会违反一般规则）。
- 电信服务交易分析。
- 大型在线零售商（如Amazon.com）的每日/周交易分析。

# 使用MapReduce的购物篮分析

由于有大量大数据，所以MapReduce是完成购物篮分析的理想框架。我们将采用Jongwook Woo和Yuhang Xu[33]提出的购物篮分析算法。给定一个交易集（其中每个交易是一个商品集），我们要回答这样一个问题：哪两个商品经常会一起购买？购物篮分析的高层MapReduce解决方案如图7-1所示。

MBA的`map()`和`reduce()`函数是怎样的？每个`map()`函数接受一个交易，这是一个顾客购买的一个商品集$\{I_1, I_2, \ldots, I_n\}$。映射器首先对这些商品排序（升序或降序），这会生成$\{S_1, S_2, \ldots, S_n\}$，然后发出（key, 1）对，这里key = Tuple2($S_i$, $S_j$)，$S_i \leqslant S_j$，而且value为1（表示这个键已经见过一次）。组合器和归约器的任务是聚集和统计频度。

详细讨论MapReduce算法之前，下面来看输入和期望的输出。

## 输入

假设输入是一个交易序列（每行表示一个交易）。交易商品用逗号分隔。下面给出示例输入（“:”后面才是具体的交易数据）：

```
Transaction 1: crackers, icecream, coke, apple
Transaction 2: chicken, pizza, coke, bread
Transaction 3: baguette, soda, shampoo, crackers, pepsi
Transaction 4: baguette, cream cheese, diapers, milk
...
```

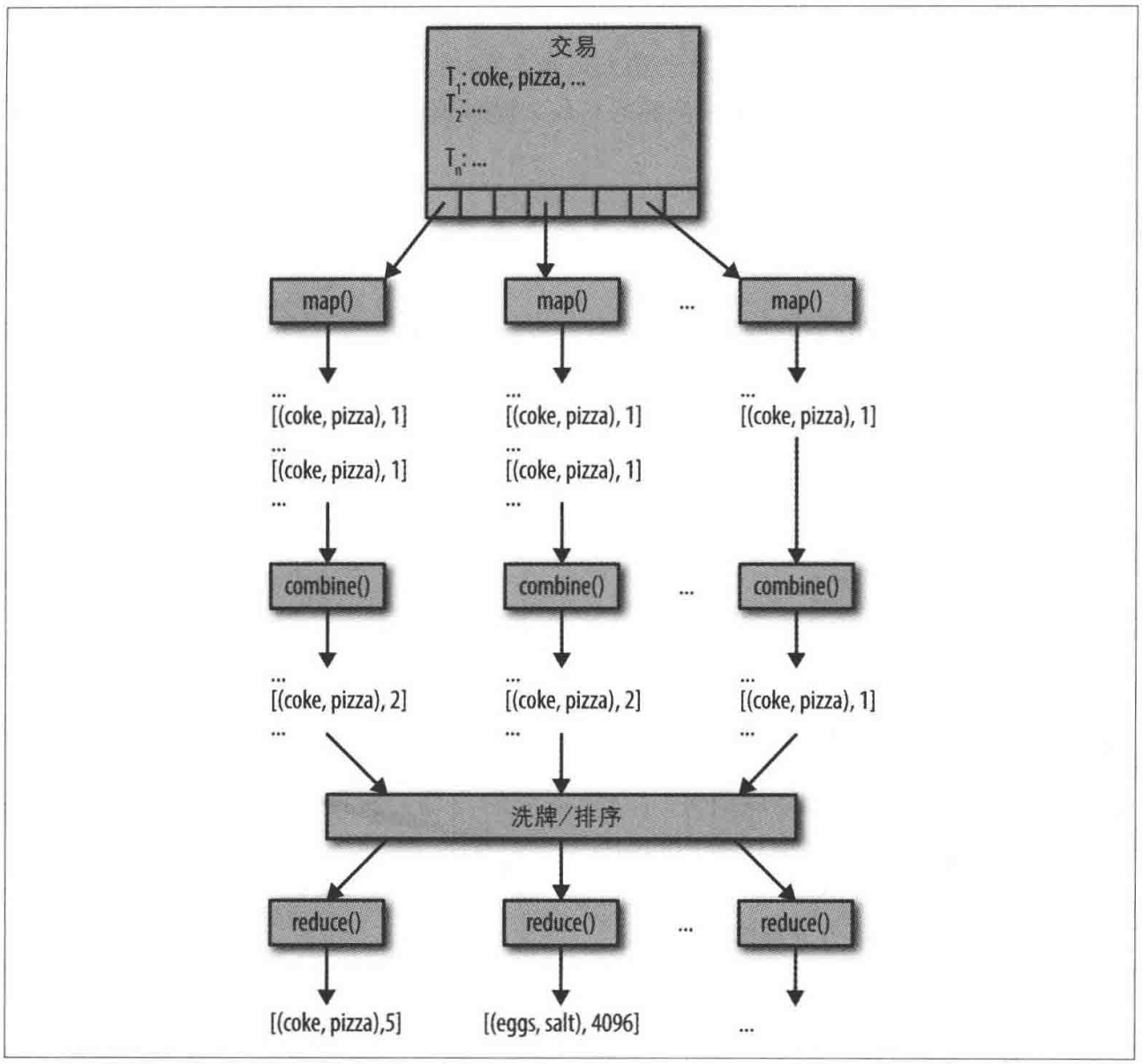



图7-1：MapReduce购物篮分析算法

## Tuple2（2阶）期望输出

对于2阶元组，我们希望找出所有不同（$Item_1$, $Item_2$）对的商品频度。表7-1给出了示例期望输出。

表7-1：Tuple2（2阶）期望输出

| 商品对 | 频度 |
|---|---|
| ... | ... |
| (bread, crackers) | 8709 |
| (baguette, eggs) | 7524 |
| (crackers, coke) | 4300 |
| ... | ... |

## Tuple3（3阶）期望输出

对于3阶元组，我们希望找出所有不同（$Item_1$, $Item_2$, $Item_3$）集的商品频度。表7-2给出了示例期望输出。

**表7-2：Tuple3（3阶）期望输出**

| 商品三元组 | 频度 |
|---|---|
| ... | ... |
| (bread, crackers, eggs) | 1940 |
| (baguette, diapers, eggs) | 1900 |
| (crackers, coke, meat) | 1843 |
| ... | ... |

## 非形式化映射器

map()函数根据一个交易生成一组将由reduce()处理的键-值对。映射器会配对交易商品作为键，并以它在购物篮中出现的次数作为值（包括所有交易，而没有交易号）。

例如，对应交易1的map()会生成以下键-值对：

```
[<crackers, icecream>, 1]
[<crackers, coke>, 1]
[<crackers, apple>, 1]
[<icecream, coke>, 1]
[<icecream, apple>, 1]
[<coke, apple>, 1]
```

需要说明，如果选择一个购物篮中的两个商品作为键，这些商品成对出现的次数是不正确的。例如，如果交易$T_1$和$T_2$有以下商品（相同的商品，但是顺序不同）：

```
T1: crackers, icecream, coke
T2: icecream, coke, crackers
```

那么，对于交易$T_1$，map()会生成：

```
[(crackers, icecream), 1]
[(crackers, coke), 1]
[(icecream, coke), 1]
```

对于交易$T_2$，map()会生成：

```
[(icecream, coke), 1]
[(icecream, crackers), 1]
[(coke, crackers), 1]
```

从交易$T_1$和$T_2$的map()输出可以看到，我们总共得到6对不同的商品，它们分别只出

现1次，但实际上我们应该只得到3对不同的商品。也就是说，尽管键（crackers, icecream）和（icecream, crackers）是一样的，但在这里却被看作是不一样的，这是不正确的。如果在生成键-值对之前先对交易商品按字母顺序排序，就可以避免这个问题。对交易中的商品排序之后，我们可以得到：

```
Sorted T1: coke, crackers, icecream
Sorted T2: coke, crackers, icecream
```

现在每个交易（$T_1$和$T_2$）都有以下3个键-值对：

```
[(coke, crackers), 1]
[(coke, icecream), 1]
[(crackers, icecream), 1)
```

通过对交易商品排序，我们就能得到映射器生成的键-值对的正确计数。这样一来，对于交易$T_1$和$T_2$，combiner()函数会生成正确的输出：

```
[(coke, crackers), 2],
[(coke, icecream), 2],
[(crackers, icecream),2]
```

## 形式化映射器

映射器的MBA算法如示例7-1所示。这个映射器读取输入数据，并创建各个交易的一个商品列表。对于每个交易，其时间复杂度为O(n)，这里n是一个交易的商品数目。然后对交易列表中的商品排序，以避免类似（crackers,coke）和（coke, crackers）的重复键。快速排序（Quicksort）的时间复杂度是O($n$ log $n$)。接下来，已排序的交易商品将转换为商品对作为键，这是一个交叉操作，来生成列表中商品的交叉对。

示例7-1：MBA map()函数

```
1 // key是交易ID，在这里忽略
2 // value = 交易商品(I1, I2, ...,In)
3 map(key, value) {
4     (S1, S2, ..., Sn) = sort(I1, I2, ...,In);
5     // 现在可以确保： S1 < S2 < ... < Sn
6     List<Tuple2<Si, Sj>> listOfPairs =
7          Combinations.generateCombinations(S1, S2, ..., Sn);
8     for ( Tuple2<Si, Sj> pair : listOfPairs) {
9         // 归约器键为：Tuple2<Si, Sj>
10        // 归约器值为整数1
11        emit([Tuple2<Si, Sj>, 1]);
12    }
13 }
```

Combinations是一个简单的Java工具类，可以为给定的“商品列表”和“对数”（2, 3, 4, ...）生成购物篮商品组合。假设（$S_1$, $S_2$, ..., $S_n$）是有序的（也就是说，$S_1 \leq S_2 \leq$ ...

$\leq S_n$）。Combinations.generateCombinations($S_1$, $S_2$, ..., $S_n$)方法会生成一个给定交易中的所有两商品组合。例如，generateCombinations($S_1$, $S_2$, $S_3$, $S_4$) 会返回以下商品对：

```
(S1, S2)
(S1, S3)
(S1, S4)
(S2, S3)
(S2, S4)
(S3, S4)
```

## 归约器

归约器的MBA算法如示例7-2所示。这个归约器会累加对应各个归约器键的值数目。因此，它的时间复杂度是O($n$)，这里n是各个键的值数目。

示例7-2：MBA reduce()函数

```
// key形式为Tuple2(Si, Sj)
// value = List<integer>, 其中各个元素分别是一个整数
reduce(Tuple2(Si, Sj) key, List<integer> values) {
    integer sum = 0;
    for (integer i : values) {
        sum += i;
    }

    emit(key, sum);
}
```

# MapReduce/Hadoop实现类

Hadoop实现包括表7-3所示的Java类。

表7-3：MapReduce/Hadoop中的实现类

| 类名 | 类描述 |
|---|---|
| Combination | 创建项组合的工具类 |
| MBADriver | 向Hadoop提交作业 |
| MBAMapper | 定义map() |
| MBAReducer | 定义reduce() |

## 查找有序组合

Combination.findSortedCombinations()是一个递归函数，可以为任意阶数N (N = 2, 3, ...)创建一个唯一组合，如示例7-3所示。

示例7-3：查找有序组合

```
/**
 * 如果 elements = { a, b, c, d },
 * 则 findCollections(elements, 2) 会返回:
 * { [a, b], [a, c], [a, d], [b, c], [b, d], [c, d] }
 *
 * findCollections(elements, 3) 会返回:
 * { [a, b, c], [a, b, d], [a, c, d], [b, c, d] }
 *
 */
public static <T extends Comparable<? super T>> List<List<T>>
        findSortedCombinations(Collection<T> elements, int n) {
    List<List<T>> result = new ArrayList<List<T>>();

    // 处理递归的初始步骤
    if (n == 0) {
        result.add(new ArrayList<T>());
        return result;
    }

    // 处理n-1递归
    List<List<T>> combinations = findSortedCombinations(elements, n - 1);
    for (List<T> combination: combinations) {
        for (T element: elements) {
            if (combination.contains(element)) {
                continue;
            }

            List<T> list = new ArrayList<T>();
            list.addAll(combination);

            if (list.contains(element)) {
                continue;
            }

            list.add(element);
            // 对元素排序，以避免重复的元素
            // 示例：如果未排序，（a, b, c）和（a, c, b）可能会被统计为
            // 不同的元素
            Collections.sort(list);

            if (result.contains(list)) {
                continue;
            }
            result.add(list);
        }
    }
    return result;
}
```

## 购物篮分析驱动器：MBADriver

如示例7-4所示，这个驱动器类接收3个参数，并将作业提交给MapReduce/Hadoop框架。这个驱动器还设置了要由映射器读取"number.of.pairs"配置参数。

示例7-4：MBADriver

```
public class MBADriver extends Configured implements Tool {
    ...
    public int run(String args[]) throws Exception {
        String inputPath = args[0];
        String outputPath = args[1];
        int numberOfPairs = Integer.parseInt(args[2]);
        ...
        // 作业配置
        Job job = new Job(getConf());
        ...
        job.getConfiguration().setInt("number.of.pairs", numberOfPairs);

        // 设置输入/输出路径
        FileInputFormat.setInputPaths(job, new Path(inputPath));
        FileOutputFormat.setOutputPath(job, new Path(outputPath));

        // 映射器K, V输出
        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(IntWritable.class);
        // 输出格式
        job.setOutputFormatClass(TextOutputFormat.class);

        // 归约器K, V输出
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);

        // 设置映射器/归约器/组合器
        job.setMapperClass(MBAMapper.class);
        job.setCombinerClass(MBAReducer.class);
        job.setReducerClass(MBAReducer.class);

        //如果输出路径已经存在，则删除，以避免"existing dir/file" 错误
        Path outputDir = new Path(outputPath);
        FileSystem.get(getConf()).delete(outputDir, true);

        // 提交作业
        boolean status = job.waitForCompletion(true);
        ...
    }
```

## 购物篮分析映射器：MBAMapper

map()函数一次读取一个交易，生成一个键-值对集，其中键是一个给定交易中商品的N阶组合，值是一个整数1（指示这个组合已经见过1次）。映射器的setup()方法从Hadoop的Configuration对象读取"number.of.pairs"。参见示例7-5到示例7-7。

示例7-5：MBAMapper

```
public class MBAMapper extends Mapper<LongWritable, Text, Text, IntWritable> {

    public static final int DEFAULT_NUMBER_OF_PAIRS = 2;

```

```
    // 输出key2：成对项列表，可以是2或3 ...
    private static final Text reducerKey = new Text();

    // 输出value2：项列表中的成对项数
    private static final IntWritable NUMBER_ONE = new IntWritable(1);

    int numberOfPairs; // 由setup()读取，由驱动器设置

    protected void setup(Context context)
        throws IOException, InterruptedException {
        this.numberOfPairs = context.getConfiguration()
              .getInt("number.of.pairs", DEFAULT_NUMBER_OF_PAIRS);
    }

    public void map(LongWritable key, Text value, Context context)
        throws IOException, InterruptedException {
        String line = value.toString().trim();
        List<String> items = convertItemsToList(line);
        if ((items == null) || ( items.isEmpty())) {
           // 不会生成映射器输出
           return;
        }
        generateMapperOutput(numberOfPairs, items, context);
     }

     private static List<String> convertItemsToList(String line) {
          // 参见示例7-6
     }

     private void generateMapperOutput(...) {
        // 参见示例7-7
     }
}
```

示例7-6：MBAMapper辅助方法：convertItemsToList

```
   private static List<String> convertItemsToList(String line) {
       if ((line == null) || ( line.length() == 0)) {
           // 不会生成映射器输出
           return null;
       }
       String[] tokens = StringUtils.split(line, ",");
       if (( tokens == null) || ( tokens.length == 0) ) {
           return null;
       }
       List<String> items = new ArrayList<String>();
       for (String token : tokens) {
           if (token != null) {
               items.add(token.trim());
           }
       }
       return items;
   }
```

示例7-7：MBAMapper辅助方法：generateMapperOutput

```
    /**
     *
     * 这个方法通过对输入列表排序建立一个键-值对集合。
     * 键是交易中的一个商品组合，值= 1。
     * 这里需要排序以确保(a, b)和(b, a)
     * 表示相同的键。
     * @param numberOfPairs是关联商品对数。
     * @param items 是商品列表（来自输入行）。
     * @param context 是Hadoop作业上下文。
     * @throws IOException
     * @throws InterruptedException
     */
    private void generateMapperOutput(int numberOfPairs,
                                      List<String> items,
                                      Context context)
        throws IOException, InterruptedException {
        List<List<String>> sortedCombinations =
            Combination.findSortedCombinations(items, numberOfPairs);
        for (List<String> itemList: sortedCombinations) {
            reducerKey.set(itemList.toString());
            context.write(reducerKey, NUMBER_ONE);
        }
    }
}
```

# 运行示例

## 输入

```
# hadoop fs -cat /market_basket_analysis/input/input.txt
crackers,bread,banana
crackers,coke,butter,coffee
crackers,bread
crackers,bread
crackers,bread,coffee
butter,coke
butter,coke,bread,crackers
```

## 运行示例日志

```
# INPUT=/market_basket_analysis/input
# OUTPUT=/market_basket_analysis/output
# $HADOOP_HOME/bin/hadoop fs -rmr $OUTPUT
# $HADOOP_HOME/bin/hadoop jar $JAR MBADriver $INPUT $OUTPUT 2
...
Deleted hdfs://localhost:9000/market_basket_analysis/output
14/05/02 13:36:36 INFO MBADriver: inputPath: /market_basket_analysis/input
14/05/02 13:36:36 INFO MBADriver: outputPath: /market_basket_analysis/output
14/05/02 13:36:36 INFO MBADriver: numberOfPairs: 2
...
14/05/02 13:36:37 INFO mapred.JobClient: Running job: job_201405021309_0003
14/05/02 13:36:38 INFO mapred.JobClient: map 0% reduce 0%
```

```
...
14/05/02 13:37:29 INFO mapred.JobClient: map 100% reduce 100%
14/05/02 13:37:30 INFO mapred.JobClient: Job complete: job_201405021309_0003
...
14/05/02 13:37:30 INFO mapred.JobClient: Map-Reduce Framework
14/05/02 13:37:30 INFO mapred.JobClient: Map input records=7
...
14/05/02 13:37:30 INFO mapred.JobClient: Combine input records=21
14/05/02 13:37:30 INFO mapred.JobClient: Reduce input records=12
14/05/02 13:37:30 INFO mapred.JobClient: Reduce input groups=12
14/05/02 13:37:30 INFO mapred.JobClient: Combine output records=12
14/05/02 13:37:30 INFO mapred.JobClient: Reduce output records=12
14/05/02 13:37:30 INFO mapred.JobClient: Map output records=21
14/05/02 13:37:30 INFO MBADriver: job status=true
14/05/02 13:37:30 INFO MBADriver: Elapsed time: 52737 milliseconds
14/05/02 13:37:30 INFO MBADriver: exitStatus=0
```

### 输出

```
# hadoop fs -cat /market_basket_analysis/output/part*
[bread, coffee]     1
[butter, coffee]    1
[banana, crackers]   1
[butter, coke]      3
[coffee, crackers]   2
[bread, butter]     1
[banana, bread]     1
[bread, crackers]    5
[coke, crackers]     2
[bread, coke]       1
[coffee, coke]      1
[butter, crackers]   2
```

# Spark解决方案

这一节将提供一个Spark解决方案，可以为交易集生成所有关联规则。关联规则是形如$X \rightarrow Y$的if-then蕴涵式，这表示如果找出购物篮中的所有$X$项（表示为一个交易），则很有可能找到$Y$。在详细分析如何生成关联规则之前，首先需要明确一些基本定义。令$I = \{I_1, I_2, \ldots, I_n\}$：这是一个项集。交易t定义为$t = \{S_1, S_2, \ldots, S_m\}$，其中$\{S_i \in I\}$而且$m \leq n$。交易数据库T定义为一个交易集$T = \{T_1, T_2, \ldots, T_k\}$。因此，每个交易$\{T_i\}$包含I中的一个项集。给定这些定义，下面可以定义关联规则为：

$$X \rightarrow Y, \text{ where } X, Y \in I, \text{ and } X \cap Y = \emptyset$$

例如，假设有$X$={milk, bread}和$Y$={cereal, sugar, butter}。$X$（包含两元素的项集）和$Y$（包含三元素的项集）分别称为一个关联规则的左件和右件。因此，基本说来，可以说关联规则是一个模式，表示如果$X$出现，则$Y$会以某个特定概率出现。在回顾关联规则的两个重要度量（这一章前面讨论过的支持度和置信度）之前，还需要再给出几

个定义。项集（item set）是包含一个或多个项的集合。例如，项集$Y$={cereal, sugar, butter}包含3个元素。支持数（用θ表示）是一个项集在所有交易中的出现频度。例如：

$$\theta(\{I_1, I_2\}) = 3$$

表示在所有交易中，项$I_1$和$I_2$只在3个交易中一起出现。

现在可以重新描述关联规则的支持度和置信度：

- 支持度（Support）是包含某个项集的交易的百分比。所以，对于一个给定的关联规则$X \rightarrow Y$，支持度就是同时包含X和Y的交易的百分比。可以写为：

$$\text{support} = \frac{\theta(\{I_1, I_2\})}{\#\text{ of transactions}}$$

例如，如果$\theta(\{I_1, I_2\}) = 3$，交易总数为24，则$\{I_1, I_2\}$的支持度定义为$\frac{3}{24}$。生成频繁项集时会使用一个支持度阈值。

- 置信度（Confidence）是指，对于一个给定的关联规则$X \rightarrow Y$，$Y$中的项出现在包含$X$的交易中的频度。生成关联规则时会使用一个置信度阈值。置信度可以表示为：

$$\text{confidence} = \frac{\theta(Y)}{\theta(X)}$$

我们的目标是找出满足用户指定的最小支持度（minimum-support）和最小置信度（minimum-confidence）的所有关联规则。可以传入这两个参数来限制所生成的频繁项集数以及相应的关联规则数。总之，可以认为关联规则（表示为$X \rightarrow Y$）就是两个不相交项集X和Y之间的关系，表示交易中的“若$X$出现，则$Y$也出现”模式。

在我们的解决方案中，没有按minimum-support（最小支持度）来限制频繁项集，不过我们的算法会计算生成的每一个关联规则的置信度。这个Spark解决方案会找出所有频繁项集，生成所有适当的关联规则，最后计算生成的这些关联规则的置信度。计算支持度相当简单：只需要将一个项集的频度除以所有交易总数。

## MapReduce算法工作流

MapReduce算法工作流如图7-2所示。它包括两个阶段：

- MapReduce阶段1: 映射器将交易转换为模式，归约器找出频繁模式。
- MapReduce阶段2: 映射器将频繁模式转换为子模式，最后归约器生成关联规则和相关的置信度。

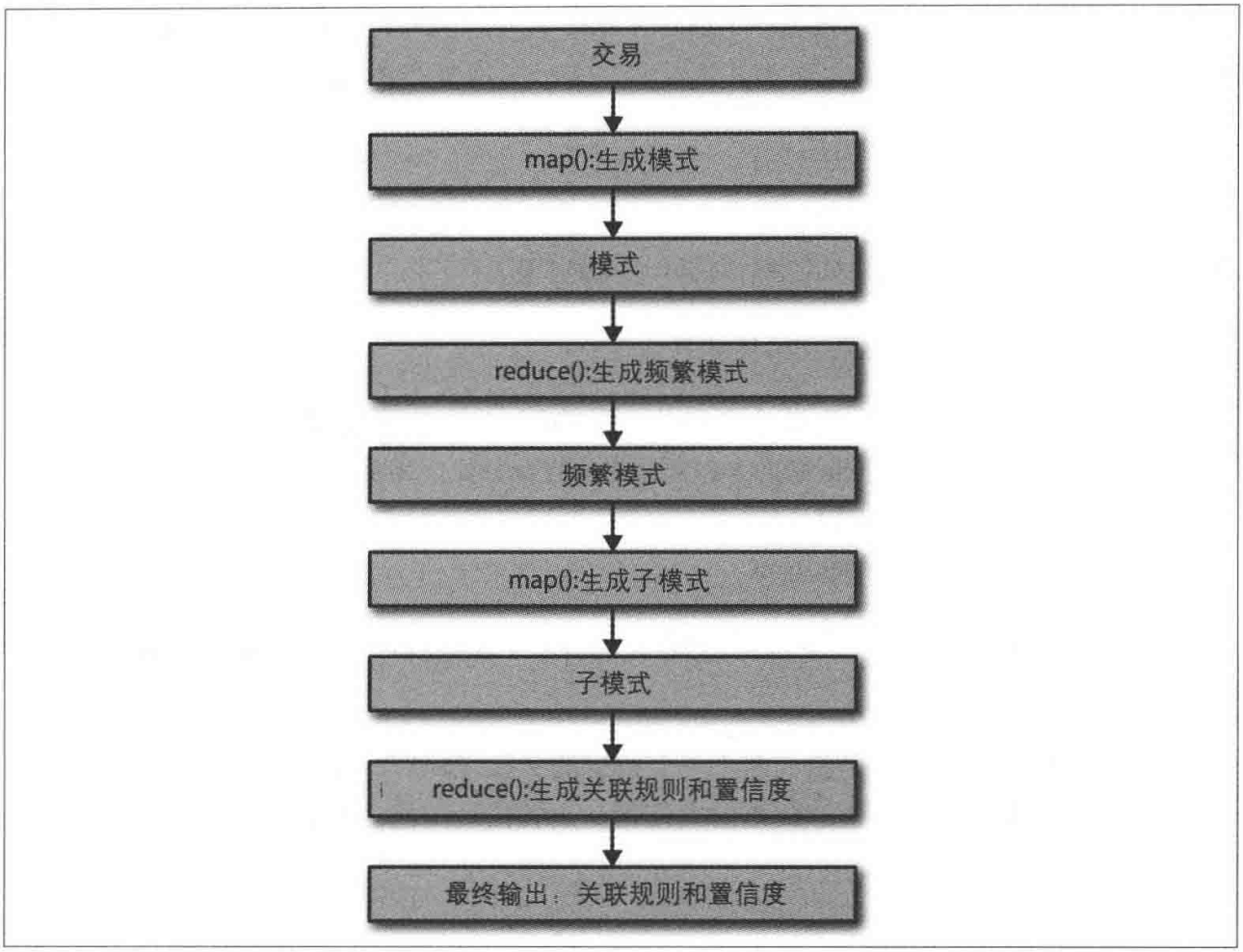



图7-2：MapReduce算法工作流

## 输入

输入是一个交易集（每行表示一个交易）。例如，为了进行测试，我们将使用以下输入（包括4个交易）：

```
transaction-1: a,b,c
transaction-2: a,b,d
transaction-3: b,c
transaction-4: b,c
```

## Spark实现

Spark实现遵循前面给出的两阶段MapReduce算法。首先，这里会给出所有高层步骤，然后再详细分析各个步骤。

示例7-8利用一个Java驱动器类来提供这个Spark解决方案。由于Spark API提供了一个高层抽象，所以这是可行的，有关内容在前面几章已经介绍过。

示例7-8：高层步骤

```
// 步骤1：导入必要的类和接口
public class FindAssociationRules {

   static JavaSparkContext createJavaSparkContext() {...}
   static List<String> toList(String transaction) {...}
   static List<String> removeOneItem(List<String> list, int i) {...}

   public static void main(String[] args) throws Exception {
      // 步骤2：处理输入参数
      // 步骤3：创建一个Spark上下文对象
      // 步骤4：从HDFS读取所有交易并创建第一个RDD
      // 步骤5：生成频繁模式(map()阶段1)
      // 步骤6：组合/归约频繁模式 (reduce()阶段1)
      // 步骤7：生成所有子模式 (map()阶段2)
      // 步骤8：组合子模式
      // 步骤9：生成关联规则 (reduce()阶段2)
      System.exit(0);
   }
}
```

步骤5和步骤6处理MapReduce过程的阶段1，如图7-3所示。步骤7到步骤9解决第2阶段，如图7-4所示。我们将在后面的小节中更详细地讨论各个步骤。

## 步骤1：导入必要的类和接口

示例7-9导入Spark驱动器程序中需要用到的Java类和接口。

示例7-9：步骤1：导入必要的类和接口

```
// 步骤1：导入必要的类和接口
import java.util.List;
import java.util.ArrayList;
import scala.Tuple2;
import scala.Tuple3;
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.api.java.JavaPairRDD;
import org.apache.spark.api.java.JavaSparkContext;
import org.apache.spark.api.java.function.PairFlatMapFunction;
import org.apache.spark.api.java.function.FlatMapFunction;
import org.apache.spark.api.java.function.Function;
import org.apache.spark.api.java.function.Function2;
import org.apache.spark.SparkConf;
```

## 创建一个Spark上下文对象

要创建RDD，需要创建一个`JavaSparkContext`对象。可以使用`createJavaSparkContext()`方法来创建，如示例7-10所示。`JavaSparkContext`是一个用来创建新RDD的工厂类。

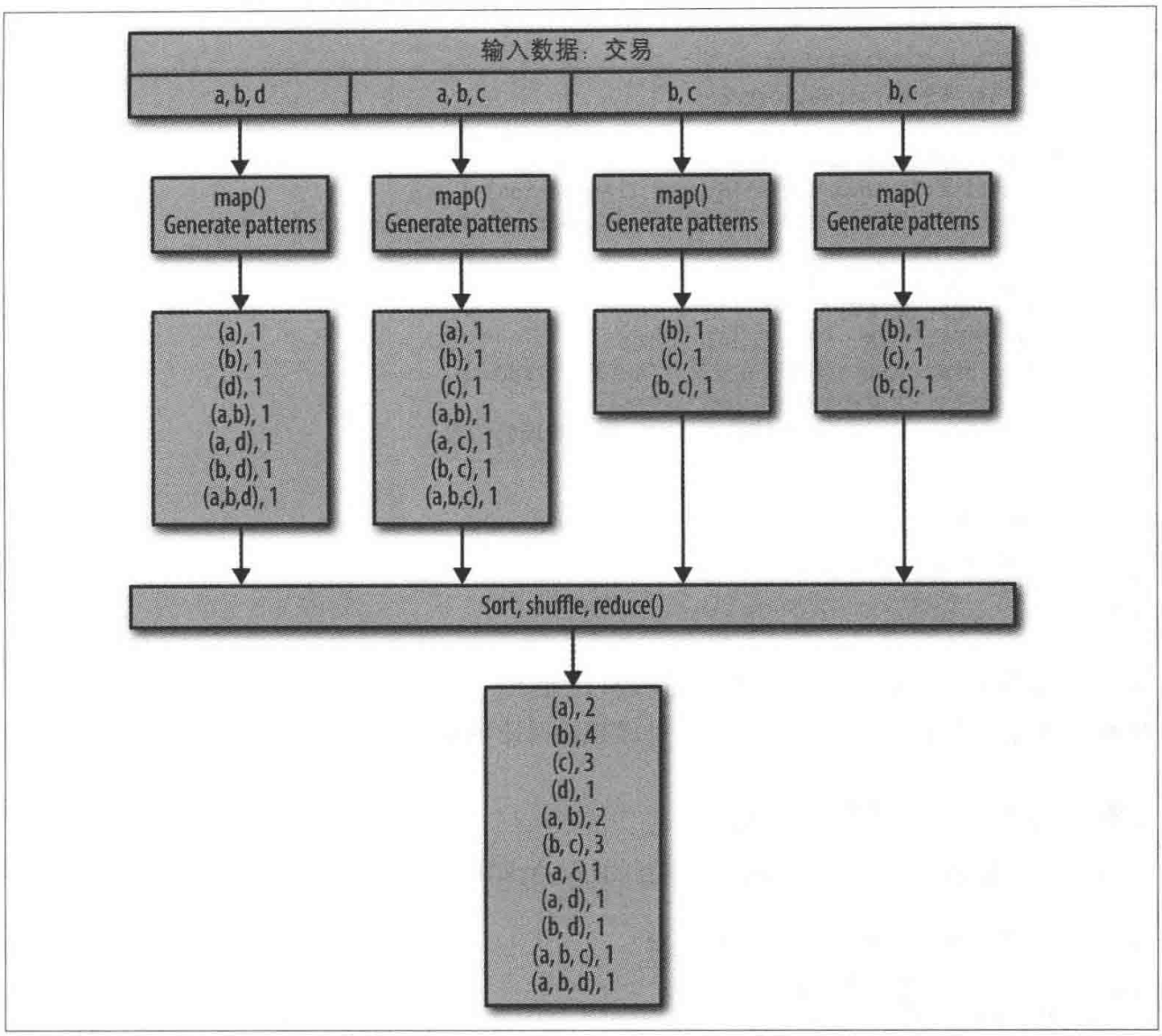



图7-3：MapReduce阶段1：交易转换为模式并找出频繁模式

示例7-10：创建一个Spark上下文对象

```
1 static JavaSparkContext createJavaSparkContext() throws Exception {
2    SparkConf conf = new SparkConf();
3    conf.setAppName("market-basket-analysis");
4    // 建立一个快速串行化器
5    conf.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer");
6    // 现在默认为32 MB缓冲区而不是64 MB
7    // 参见: https://github.com/cloudera/spark/blob/master/docs/configuration.md
8    conf.set("spark.kryoserializer.buffer.mb","32");
9    JavaSparkContext ctx = new JavaSparkContext(conf);
10   return ctx;
11 }
```

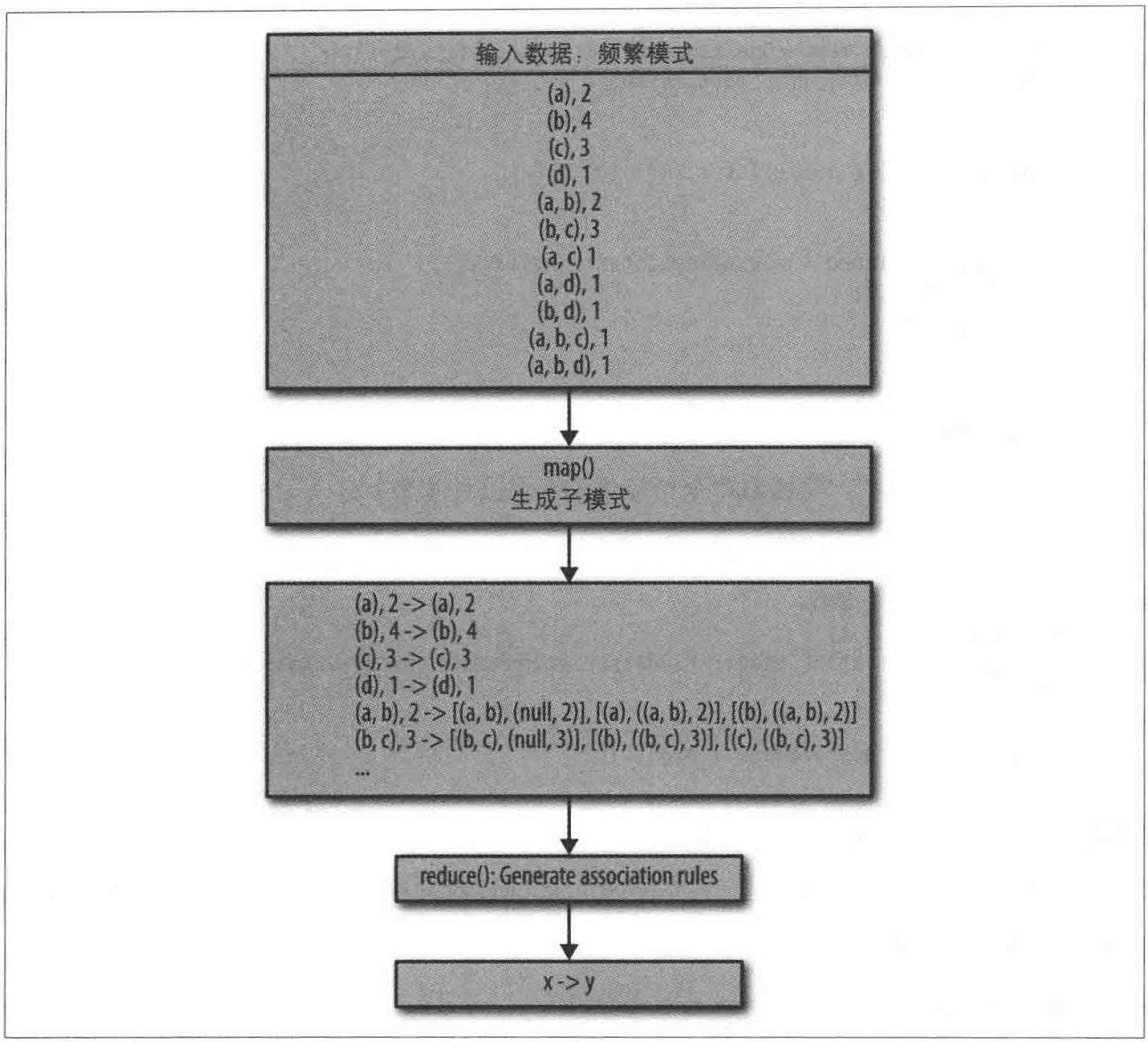



图7-4：MapReduce阶段2: 生成关联规则

## 工具函数

示例7-11给出了这个解决方案中使用的两个工具函数。toList()函数接受一个交易（各项用逗号分隔），并返回一个项列表。removeOneItem()函数从一个给定列表删除一项，返回删除了这一项的一个新列表（用来生成关联规则的左件）。

示例7-11：工具列表函数

```
1 static List<String> toList(String transaction) {
2    String[] items = transaction.trim().split(",");
3    List<String> list = new ArrayList<String>();
4    for (String item : items) {
5      list.add(item);
6    }
7    return list;
8 }
```

```
 9
10 static List<String> removeOneItem(List<String> list, int i) {
11    if (( list == null) || ( list.isEmpty()) ) {
12       return list;
13    }
14    if (( i < 0) || ( i > (list.size()-1)) ) {
15       return list;
16    }
17    List<String> cloned = new ArrayList<String>(list);
18    cloned.remove(i);
19    return cloned;
20 }
```

## 步骤2：处理输入参数

这一步如示例7-12所示，将读取命令行提供的交易文件参数。

示例7-12：步骤2：处理输入参数

```
1    // 步骤2：处理输入参数
2    if (args.length < 1) {
3       System.err.println("Usage: FindAssociationRules <transactions>");
4       System.exit(1);
5    }
6    String transactionsFileName = args[0];
```

## 步骤3：创建一个Spark上下文对象

这一步如示例7-13所示，将创建一个JavaSparkContext对象（在示例7-10中已经看到），这是一个用来创建新RDD的工厂对象。

示例7-13：步骤3：创建一个Spark上下文对象

```
1    // 步骤3：创建一个Spark上下文对象
2    JavaSparkContext ctx = createJavaSparkContext();
```

## 步骤4：从HDFS读取交易并创建一个RDD

如示例7-14所示，这一步会从一个HDFS文件读取所有交易，并创建第一个RDD（JavaRDD<String>），其中的每一项分别是一个交易。

示例7-14：步骤4：从HDFS读取交易并创建一个RDD

```
1    // 步骤4：从HDFS读取所有交易并创建第一个RDD
2    JavaRDD<String> transactions = ctx.textFile(transactionsFileName, 1);
3    transactions.saveAsTextFile("/rules/output/1");
```

为了进行调试，我们将中间数据写入HDFS。这里只是简单地发出所有原始交易：

```
# hadoop fs -cat /rules/output/1/part*
a,b,c
a,b,d
```

```
b,c
b,c
```

## 步骤5：生成频繁模式

映射器会生成一个给定交易的所有模式。例如，给定一个包含3项的交易{a， b， c}，映射器会发出以下键-值对，如表7-4所示。

**表7-4：映射器发出的键-值对**

| 键 | 值 |
|---|---|
| a | 1 |
| b | 1 |
| c | 1 |
| a,b | 1 |
| a,c | 1 |
| b,c | 1 |
| a,b,c | 1 |

一般地，在MBA中，可以选择长度为2或3的模式。如果选择所有模式，这个过程可能非常耗费时间。在这里，为了生成所有模式，我们使用了一个递归函数：

```
List<List<String>> Combination.findSortedCombinations(List<String>)
```

例如，findSortedCombinations(List<a, b, c, d>)会生成：

```
[
  [],
  [a],
  [b],
  [c],
  [d],
  [a, b],
  [a, c],
  [a, d],
  [b, c],
  [b, d],
  [c, d],
  [a, b, c],
  [a, b, d],
  [a, c, d],
  [b, c, d],
  [a, b, c, d]
]
```

需要说明，这些组合总是有序的。例如，对于一个包含两项的列表{a， b}，总是会创建

[a, b] 而不会创建 [b, a] 。由于项集是有序的，这样可以避免重复。这一章的最后会给出Combination类的定义。

示例7-15给出了这一步的代码，这一步将生成频繁模式。

示例7-15：步骤5：生成频繁模式

```
// 步骤5：生成频繁模式
// PairFlatMapFunction<T, K, V>
// T => Iterable<Tuple2<K, V>>
JavaPairRDD<List<String>,Integer> patterns =
    transactions.flatMapToPair(new PairFlatMapFunction<
                                                    String,       // T
                                                    List<String>, // K
                                                    Integer       // V
                                                   >() {
    public Iterable<Tuple2<List<String>,Integer>> call(String transaction) {
       List<String> list = toList(transaction);
       List<List<String>> combinations =
               Combination.findSortedCombinations(list);
       List<Tuple2<List<String>,Integer>> result =
               new ArrayList<Tuple2<List<String>,Integer>>();
       for (List<String> combList : combinations) {
            if (combList.size() > 0) {
                result.add(new Tuple2<List<String>,Integer>(combList, 1));
            }
       }
       return result;
    }
 });
 patterns.saveAsTextFile("/rules/output/2");
```

下面给出这一步的输出：

```
# hadoop fs -cat /rules/output/2/part*
([a],1)
([b],1)
([c],1)
([a, b],1)
([a, c],1)
([b, c],1)
([a, b, c],1)
([a],1)
([b],1)
([d],1)
([a, b],1)
([a, d],1)
([b, d],1)
([a, b, d],1)
([b],1)
([c],1)
([b, c],1)
([b],1)
([c],1)
```

```
([b, c],1)
```

## 步骤6：组合/归约频繁模式

这一步如示例7-16所示，这里实现了MapReduce算法阶段1的reduceByKey()函数。它会在所有交易中找出所有不同的频繁模式及相关的频度。

示例7-16：步骤6：组合/归约频繁模式

```
// 步骤6：组合/归约频繁模式
JavaPairRDD<List<String>, Integer> combined =
    patterns.reduceByKey(new Function2<
                                        Integer, // 输入 i1
                                        Integer, // 输入 i2
                                        Integer  // i1+i2的结果
                                       >() {
    public Integer call(Integer i1, Integer i2) {
        return i1 + i2;
    }
});
combined.saveAsTextFile("/rules/output/3");

// 现在可以得到： patterns(K,V)
// K = 模式（List<String>）
// V = 模式频度
// 现在，给定 （K,V）为（List<a,b,c>, 2），将生成
// 以下（K2,V2） 对：
//
// (List<a,b,c>, T2(null, 2))
// (List<a,b>, T2(List<a,b,c>, 2))
// (List<a,c>, T2(List<a,b,c>, 2))
// (List<b,c>, T2(List<a,b,c>, 2))
```

下面给出这一步的输出：

```
# hadoop fs -cat /rules/output/3/part*
([a, b],2)
([a, b, d],1)
([c],3)
([b, d],1)
([d],1)
([a],2)
([b, c],3)
([a, b, c],1)
([a, c],1)
([a, d],1)
([b],4)
```

## 步骤7：生成所有子模式

这一步将创建所有子模式，这是创建关联规则所必需的。给定一个频繁模式：

$(K = List < A_1, A_2, .., A_n >, V = Frequency)$

会创建以下子模式（$K_2$, $V_2$）：

($K_2$ = List < $A_1$, $A_2$, .., $A_n$ >, $V_2$ = Tuple2(null, Frequency))

($K_2$ = List < $A_1$, $A_2$, .., $A_{n-1}$ >, $V_2$ = Tuple2(K, Frequency))

($K_2$ = List < $A_1$, $A_2$, .., $A_{n-2}$, $A_n$ >, $V_2$ = Tuple2(K, Frequency))

...

例如，给定（K, V）为（(List(a,b,c), 2），会生成如表7-5所示的（$K_2$, $V_2$）对。

表7-5：子模式（$K_2$,$V_2$）

| $K_2$ | $V_2$ |
|---|---|
| List(a,b,c) | Tuple2(null, 2) |
| List(a,b) | Tuple2(List(a,b,c), 2) |
| List(a,c) | Tuple2(List(a,b,c), 2) |
| List(b,c) | Tuple2(List(a,b,c), 2) |

这一步如示例7-17所示。

示例7-17：步骤7：生成所有子模式

```
// 步骤7：生成所有子模式
// PairFlatMapFunction<T, K, V>>
// T => Iterable<Tuple2<K, V>>
JavaPairRDD<List<String>,Tuple2<List<String>,Integer>> subpatterns =
    combined.flatMapToPair(new PairFlatMapFunction<
              Tuple2<List<String>, Integer>, // T
              List<String>,                  // K
              Tuple2<List<String>,Integer>   // V
                                                >() {
    public Iterable<Tuple2<List<String>,Tuple2<List<String>,Integer>>>
        call(Tuple2<List<String>, Integer> pattern) {
        List<Tuple2<List<String>,Tuple2<List<String>,Integer>>> result =
            new ArrayList<Tuple2<List<String>,Tuple2<List<String>,Integer>>>();
        List<String> list = pattern._1;
        Integer frequency = pattern._2;
        result.add(new Tuple2(list, new Tuple2(null,frequency)));
        if (list.size() == 1) {
            return result;
        }

        // 模式中包含多个商品
        // result.add(new Tuple2(list, new Tuple2(null,size)));
        for (int i=0; i < list.size(); i++) {
            List<String> sublist = removeOneItem(list, i);
            result.add(new Tuple2(sublist, new Tuple2(list, frequency)));
        }
        return result;
    }
```

```
29 });
30 subpatterns.saveAsTextFile("/rules/output/4");
```

下面给出这一步的输出：

```
# hadoop fs -cat /rules/output/4/part*
([a, b],(null,2))
([b],([a, b],2))
([a],([a, b],2))
([a, b, d],(null,1))
([b, d],([a, b, d],1))
([a, d],([a, b, d],1))
([a, b],([a, b, d],1))
([c],(null,3))
([b, d],(null,1))
([d],([b, d],1))
([b],([b, d],1))
([d],(null,1))
([a],(null,2))
([b, c],(null,3))
([c],([b, c],3))
([b],([b, c],3))
([a, b, c],(null,1))
([b, c],([a, b, c],1))
([a, c],([a, b, c],1))
([a, b],([a, b, c],1))
([a, c],(null,1))
([c],([a, c],1))
([a],([a, c],1))
([a, d],(null,1))
([d],([a, d],1))
([a],([a, d],1))
([b],(null,4))
```

## 步骤8：组合子模式

如示例7-18所示，这一步将使用Spark的groupByKey()方法按键对子模式分组。

示例7-18：步骤8：组合子模式

```
1   // 步骤8：组合子模式
2   JavaPairRDD<List<String>,Iterable<Tuple2<List<String>,Integer>>> rules =
3       subpatterns.groupByKey();
4   rules.saveAsTextFile("/rules/output/5");
```

下面给出这一步的输出：

```
# hadoop fs -cat /rules/output/5/part*
([a, b],[(null,2), ([a, b, d],1), ([a, b, c],1)])
([a, b, d],[(null,1)])
([c],[(null,3), ([b, c],3), ([a, c],1)])
([b, d],[([a, b, d],1), (null,1)])
([d],[([b, d],1), (null,1), ([a, d],1)])
([a],[([a, b],2), (null,2), ([a, c],1), ([a, d],1)])
```

```
([b, c],[(null,3), ([a, b, c],1)])
([a, b, c],[(null,1)])
([a, c],[([a, b, c],1), (null,1)])
([a, d],[([a, b, d],1), (null,1)])
([b],[([a, b],2), ([b, d],1), ([b, c],3), (null,4)])
```

## 步骤9：生成关联规则

现在已经有了频繁模式和所有子模式，接下来可以生成所有关联规则和相应的置信度值。这一步由JavaPairRDD.map()函数实现，如示例7-19所示。

示例7-19：步骤9：生成关联规则

```
// 步骤9：生成关联规则
// Now, use (K=List<String>, V=Iterable<Tuple2<List<String>,Integer>>)
//  生成关联规则
// JavaRDD<R> map(Function<T,R> f)
// 通过对这个RDD的所有元素应用一个函数返回一个新的RDD。
JavaRDD<List<Tuple3<List<String>,List<String>,Double>>> assocRules =
    rules.map(new Function<
          Tuple2<List<String>,Iterable<Tuple2<List<String>,Integer>>>,
          List<Tuple3<List<String>,List<String>,Double>>
          // T: 输入
          // R: ( ac => b, 1/3): T3(List(a,c), List(b), 0.33)
          //    ( ad => c, 1/3): T3(List(a,d), List(c), 0.33)
         >() {
    public List<Tuple3<List<String>,List<String>,Double>>
        call(Tuple2<List<String>,Iterable<Tuple2<List<String>,Integer>>> in) {
        List<Tuple3<List<String>,List<String>,Double>> result =
            new ArrayList<Tuple3<List<String>,List<String>,Double>>();
        List<String> fromList = in._1;
            Iterable<Tuple2<List<String>,Integer>> to = in._2;
        List<Tuple2<List<String>,Integer>> toList =
            new ArrayList<Tuple2<List<String>,Integer>>();
        Tuple2<List<String>,Integer> fromCount = null;
        for (Tuple2<List<String>,Integer> t2 : to) {
            // 找到"count"对象
            if (t2._1 == null) {
                fromCount = t2;
            }
            else {
                toList.add(t2);
            }
        }

        // 现在我们得到了生成关联规则所需的对象：
        // "fromList", "fromCount"和"toList"
        if (toList.isEmpty()) {
            // 没有生成输出，不过由于Spark不接受
            // null对象，我们将模拟一个null对象
            return result; // 一个空列表
        }

        // 现在使用3个对象"from", "fromCount"和"toList",
```

```
            // 创建关联规则:
            for (Tuple2<List<String>,Integer> t2 : toList) {
                double confidence = (double) t2._2 / (double) fromCount._2;
                List<String> t2List = new ArrayList<String>(t2._1);
                t2List.removeAll(fromList);
                result.add(new Tuple3(fromList, t2List, confidence));
             }
             return result;
        }
    });
    assocRules.saveAsTextFile("/rules/output/6");
```

下面给出这一步的输出（也就是最终输出）。每个输出项是一个Tuple3(X, Y, confidence)列表，表示X → Y而且confidence（置信度）为double值：

```
# hadoop fs -cat /rules/output/6/part*
[([a, b],[d],0.5), ([a, b],[c],0.5)]
[]
[([c],[b],1.0), ([c],[a],0.3333333333333333)]
[([b, d],[a],1.0)]
[([d],[b],1.0), ([d],[a],1.0)]
[([a],[b],1.0), ([a],[c],0.5), ([a],[d],0.5)]
[([b, c],[a],0.3333333333333333)]
[]
[([a, c],[b],1.0)]
[([a, d],[b],1.0)]
[([b],[a],0.5), ([b],[d],0.25), ([b],[c],0.75)]
```

# 运行Spark实现的YARN脚本

下面是在YARN环境中运行这个Spark实现的脚本：

```
1 # cat run_assoc_rules.sh
2 #!/bin/bash
3 export JAVA_HOME=/usr/java/jdk7
4 export HADOOP_HOME=/usr/local/hadoop2.5.0
5 export SPARK_HOME=/usr/local/spark-1.1.0
6 export CLASSPATH=$CLASSPATH:$HADOOP_HOME/etc/hadoop
7 export HADOOP_CONF_DIR=$HADOOP_HOME/etc/hadoop
8 export YARN_CONF_DIR=$HADOOP_HOME/etc/hadoop
9 export BOOK_HOME=/mp/data-algorithms-book
10 export APP_JAR=$BOOK_HOME/dist/data_algorithms_book.jar
11 export SPARK_JAR=$BOOK_HOME/lib/spark-assembly-1.1.0-hadoop2.5.0.jar
12 INPUT=/data/data_mining_transactions.txt
13 prog=org.dataalgorithms.chap07.spark.FindAssociationRules
14 $SPARK_HOME/bin/spark-submit --class $prog \
15     --master yarn-cluster \
16     --num-executors 12 \
17     --driver-memory 3g \
18     --executor-memory 7g \
19     --executor-cores 12 \
20     $APP_JAR $INPUT
```

## 由交易创建项集

给定一个交易T = {$I_1$, $I_2$, ...,$I_n$}（包括一个项集），我们使用了一个简单的POJO类（Combination）来生成有序项集的所有组合。Combination类如示例7-20所示。

示例7-20：Combination类

```
import java.util.List;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collection;
import java.util.Collections;

public class Combination {

    public static <T extends Comparable<? super T>> List<List<T>>
       findSortedCombinations(Collection<T> elements) {...}

    public static <T extends Comparable<? super T>> List<List<T>>
        findSortedCombinations(Collection<T> elements, int n) {...}

    public static void main(String[] args) throws Exception {
       test();
    }

    public static void test() throws Exception {
        List<String> list = Arrays.asList("a", "b", "c", "d");
        System.out.println("list="+list);
        List<List<String>> comb = findSortedCombinations(list);
        System.out.println(comb.size());
        System.out.println(comb);
    }
}
```

方法定义参见示例7-21和示例7-22。

示例7-21：Combination类: findSortedCombinations(List)

```
   /**
    * 返回所有不同大小的组合...
    * 如果elements = { a, b, c }, 则findCollections(elements)会返回:
    * { [], [a], [b], [c], [a, b], [a, c], [b, c], [a, b, c] }
    *
    */
   public static <T extends Comparable<? super T>> List<List<T>>
       findSortedCombinations(Collection<T> elements) {
       List<List<T>> result = new ArrayList<List<T>>();
       for (int i = 0; i <= elements.size(); i++) {
           result.addAll(findSortedCombinations(elements, i));
       }
       return result;
   }
```

示例7-22：Combination类：findSortedCombinations(List, n)

```
/**
 * 如果elements = { a, b, c }，则findCollections(elements, 2)会返回：
 * { [a, b], [a, c], [b, c] }
 *
 */
public static <T extends Comparable<? super T>> List<List<T>>
    findSortedCombinations(Collection<T> elements, int n) {
    List<List<T>> result = new ArrayList<List<T>>();
    if (n == 0) {
        result.add(new ArrayList<T>());
        return result;
    }

    List<List<T>> combinations = findSortedCombinations(elements, n - 1);
    for (List<T> combination: combinations) {
        for (T element: elements) {
            if (combination.contains(element)) {
                continue;
            }

            List<T> list = new ArrayList<T>();
            list.addAll(combination);
            if (list.contains(element)) {
                continue;
            }

            list.add(element);
            // 对项排序，以避免重复的项，例如，如果不排序，（a, b, c）和
            // (a, c, b) 可能会记为不同的项
            Collections.sort(list);
            if (result.contains(list)) {
                continue;
            }
            result.add(list);
        }
    }
    return result;
}
```

这一章介绍了一个著名的数据挖掘算法：购物篮分析。这里给出了用于查找频繁模式和相应关联规则的多个分布式算法（使用MapReduce/Hadoop和Spark）。下一章将提供用来查找社交网络环境中共同好友的一个简单MapReduce算法。

# 第8章

# 共同好友

给定一个包含上千万用户的社交网络，这一章中我们会实现一个MapReduce程序，在所有用户对中找出“共同好友”。令U为包含所有用户的一个集合：$\{U_1, U_2, \ldots, U_n\}$。我们的目标是为每个（$U_i, U_j$）对（$i \neq j$）找出共同好友。

要找出共同好友，这里将提供3个解决方案：

- MapReduce/Hadoop解决方案，使用基本数据类型。
- MapReduce/Hadoop解决方案，使用定制数据类型。
- Spark解决方案，使用RDD。

如今大多数社交网络网站（如Facebook、hi5和LinkedIn）都提供了有关的服务，可以帮助你与好友共享消息、图片和视频。有些网站甚至还提供了视频聊天服务，帮助你与好友保持联系。根据定义，“好友”是你认识、喜欢和信任的一个人。例如，你在Facebook上有一个好友列表，这个网站上好友关系是双向的。如果我是你的好友，那么你也是我的好友。一般地，社交网络会尽可能预先完成计算来减少请求的处理时间，其中一个常见的处理请求就是共同好友特性（如“你和玛丽（你的好友）有185个共同好友”）。访问某个人的个人资料时，你会看到你们的共同好友列表。这个列表不会频繁改变，所以如果每次访问这个人的个人资料时网站都重新计算，这就会很浪费。

有很多方法可以找出一个社交网络中用户之间的共同好友。下面给出两种可能的解决方案：

- 使用一个缓存策略，将共同好友保存在一个缓存中（如Redis或memcached）。
- 使用MapReduce每天计算一次每个人的共同好友并存储这些结果。

首先来看查找共同好友的MapReduce解决方案，先来简要分析示例输入和一个POJO解决方案。

# 输入

我们准备了一组记录作为输入，每个记录有以下格式：

```
<person><,><friend1 ><friend2 >...<friendN>
```

这里<friend1><friend2> ... <friendN>是<person>的好友。需要说明，在实际项目/应用中，每个人/好友将由一个唯一的用户ID标识。下面是一个很简单但很完整的输入示例：

```
100, 200 300 400 500 600
200, 100 300 400
300, 100 200 400 500
400, 100 200 300
500, 100 300
600, 100
```

在这个例子中，用户500有两个好友，分别由用户ID 100和300标识，用户600只有一个好友：用户100。

# POJO共同好友解决方案

令$\{A_1, A_2, ..., A_m\}$是User1的好友集合，$\{B_1, B_2, ..., B_n\}$是$User_2$的好友集合。因此，$User_1$和$User_2$的共同好友可以定义为这两个集合的交集（共同元素）。POJO（简单Java对象）解决方案如示例8-1所示。

示例8-1：CommonFriends类

```
 1 import java.util.Set;
 2 import java.util.TreeSet;
 3
 4 public class CommonFriends {
 5    // user1friends = {A1, A2, ..., Am}
 6    // user2friends = {B1, B2, ..., Bn}
 7    public static Set<Integer> intersection(Set<Integer> user1friends,
 8                                            Set<Integer> user2friends) {
 9       if ((user1friends == null) || (user1friends.isEmpty())) {
10          return null;
11       }
12
13       if ((user2friends == null) || (user2friends.isEmpty())) {
14          return null;
15       }
16
17       // 两个集合都非null
```

```
18        if (user1friends.size() < user2friends.size()) {
19           return intersect(user1friends, user2friends);
20        }
21        else {
22           return intersect(user2friends, user1friends);
23        }
24     }
25
26     private static Set<Integer> intersect(Set<Integer> smallSet,
27                                           Set<Integer> largeSet) {
28        Set<Integer> result = new TreeSet<Integer>();
29        // 迭代处理小集合来提高性能
30        for (Integer x : smallSet) {
31           if (largeSet.contains(x)) {
32              result.add(x);
33           }
34        }
35        return result;
36     }
37 }
```

# MapReduce算法

查找“共同好友”的MapReduce解决方案包括map()和reduce()函数。如示例8-2所示，映射器接受一个（$key_1$， $value_1$）对，其中$key_1$是一个人，$value_1$是这个人的相关好友列表。映射器发出一组新的（$key_2$,$value_2$）对；$key_2$是一个Tuple2($key_1$， $friend_i$)，其中$friend_i \in value_1$，$value_2$等同于$value_1$（$key_1$的所有好友列表）。归约器的key是一个用户对（$User_j$， $User_k$），value是一个好友集合列表。reduce()函数得到所有好友集合的交集，从而找出（$User_j$， $User_k$）对的共同好友。

示例8-2：查找共同好友：map()函数

```
1 // 键是person。
2 // 值是对应这个key=person的好友列表。
3 // value = (<friend_1> <friend_2> ... <friend_N>)
4 map(key, value) {
5    reducerValue = (<friend_1>< friend_2> ...< friend_N>);
6    foreach friend in (<friend_1>< friend_2> ... <friend_N>) {
7       reducerKey = buildSortedKey(person, friend);
8       emit(reducerKey, reducerValue);
9    }
10 }
```

通过buildSortedKey()函数对映射器的输出键排序，来避免重复的键（参见示例8-3）。需要说明，这里假设好友关系是双向的：如果Alex是Bob的一个好友，那么Bob也是Alex的一个好友。

示例8-3：查找共同好友：buildSortedKey()函数

```
1 Tuple2 buildSortedKey(person1, person2) {
```

```
2    if (person1 < person2) {
3       return new Tuple2(person1, person2);
4    }
5    else {
6      return new Tuple2(person2, person1);
7    }
8 }
```

reduce()函数通过得到好友集合的交集来找出每一对用户的共同好友，参见示例8-4。

示例8-4：查找共同好友：reduce()函数

```
1 // key = Tuple2(person1, person2)
2 // value = List {List_1, List_2, ..., List_M}
3 // 这里各个List_i = { 唯一的用户ID集合 }
4 reduce(key, value) {
5    outputKey = key;
6    outputValue = intersection (List_1, List_1, ..., List_M);
7    emit (outputKey, outputValue);
8 }
```

## MapReduce算法的实际使用

为了帮助理解我们的MapReduce算法，这里将给出映射器和归约器生成的所有键-值对。下面首先对示例输入应用map()。

map(100, (200 300 400 500 600))将生成：

```
([100, 200], [200 300 400 500 600])
([100, 300], [200 300 400 500 600])
([100, 400], [200 300 400 500 600])
([100, 500], [200 300 400 500 600])
([100, 600], [200 300 400 500 600])
```

map(200, (100 300 400))将生成：

```
([100, 200], [100 300 400])
([200, 300], [100 300 400])
([200, 400], [100 300 400])
```

map(300, (100 200 400 500))将生成：

```
([100, 300], [100 200 400 500])
([200, 300], [100 200 400 500])
([300, 400], [100 200 400 500])
([300, 500], [100 200 400 500])
```

map(400, (100 200 300))将生成：

```
([100, 400], [100 200 300])
([200, 400], [100 200 300])
([300, 400], [100 200 300])
```

map(500, (100 300))将生成：

```
([100, 500], [100 300])
([300, 500], [100 300])
```

map(600, (100))将生成：

```
([100, 600], [100])
```

所以，映射器会生成以下键-值对：

```
([100, 200], [200 300 400 500 600])
([100, 300], [200 300 400 500 600])
([100, 400], [200 300 400 500 600])
([100, 500], [200 300 400 500 600])
([100, 600], [200 300 400 500 600])
([100, 200], [100 300 400])
([200, 300], [100 300 400])
([200, 400], [100 300 400])
([100, 300], [100 200 400 500])
([200, 300], [100 200 400 500])
([300, 400], [100 200 400 500])
([300, 500], [100 200 400 500])
([100, 400], [100 200 300])
([200, 400], [100 200 300])
([300, 400], [100 200 300])
([100, 500], [100 300])
([300, 500], [100 300])
([100, 600], [100])
```

将这些键-值对发送给归约器之前，按键对它们分组：

```
([100, 200], [200 300 400 500 600])
([100, 200], [100 300 400])
=> ([100, 200], ([200 300 400 500 600], [100 300 400]))

([100, 300], [200 300 400 500 600])
([100, 300], [100 200 400 500])
=> ([100, 300], ([200 300 400 500 600],[100 200 400 500]))

([100, 400], [200 300 400 500 600])
([100, 400], [100 200 300])
=> ([100, 400], ([200 300 400 500 600], [100 200 300]))

([100, 500], [200 300 400 500 600])
([100, 500], [100 300])
=> ([100, 500], ([200 300 400 500 600], [100 300]))

([200, 300], [100 300 400])
([200, 300], [100 200 400 500])
=> ([200, 300], ([100 300 400],[100 200 400 500]))

([200, 400], [100 300 400])
([200, 400], [100 200 300])
```

```
=> ([200, 400], ([100 300 400],[100 200 300]))

([300, 400], [100 200 400 500])
([300, 400], [100 200 300])
=> ([300, 400], ([100 200 400 500][100 200 300]))

([300, 500], [100 200 400 500])
([300, 500], [100 300])
=> ([300, 500], ([100 200 400 500],[100 300]))

([100, 600], [200 300 400 500 600])
([100, 600], [100])
=> ([100, 600], ([200 300 400 500 600]), [100])
```

所以，归约器会接收以下键-值对集合：

```
([100, 200], ([200 300 400 500 600], [100 300 400]))
([100, 300], ([200 300 400 500 600],[100 200 400 500]))
([100, 400], ([200 300 400 500 600], [100 200 300]))
([100, 500], ([200 300 400 500 600], [100 300]))
([200, 300], ([100 300 400],[100 200 400 500]))
([200, 400], ([100 300 400],[100 200 300]))
([300, 400], ([100 200 400 500][100 200 300]))
([300, 500], ([100 200 400 500],[100 300]))
([100, 600], ([200 300 400 500 600], [100])
```

最后，归约器将生成：

```
([100, 200], [300, 400])
([100, 300], [200, 400, 500])
([100, 400], [200, 300])
([100, 500], [300])
([200, 300], [100, 400])
([200, 400], [100, 300])
([300, 400], [100, 200])
([300, 500], [100])
([100, 600], [])
```

在生成的输出中，最后可以看到用户100和600没有共同好友。用户100访问用户200的个人资料时，现在就能快速查找[100, 200]键，可以看到他们有两个共同好友：[300, 400]。另外，用户100和用户500有一个共同好友（由用户ID 300标识）。

# 解决方案1: 使用文本的Hadoop实现

假设每个用户ID是一个long数据类型，在这个实现中，我们将“好友列表”表示为一个字符串对象。例如，包括3个用户ID的列表（100, 200, 300）可以表示为字符串对象"100,200,300"。要遍历这种列表，需要对这个字符串对象完成词法分析（tokenize），然后获取其中的各项。

表8-1列出了使用基本数据类型查找共同好友用到的Hadoop实现类。

表8-1：查找共同好友的实现类

| 类名 | 描述 |
| --- | --- |
| CommonFriendsDriver | 提交Hadoop作业的驱动器程序 |
| CommonFriendsMapper | 定义map() |
| CommonFriendsReducer | 定义reduce() |
| HadoopUti | Hadoop的工具方法 |

# 解决方案1运行示例

## 脚本

下面的shell脚本可以启动查找共同好友的MapReduce程序驱动器：

```
$ cat run.sh
export JAVA_HOME=/usr/java/jdk7
export BOOK_HOME=/mp/data-algorithms-book
export APP_JAR=$BOOK_HOME/dist/data_algorithms_book.jar
INPUT=/common_friends/input
OUTPUT=/common_friends/output
$HADOOP_HOME/bin/hadoop fs -rmr $OUTPUT
PROG=org.dataalgorithms.chap08.mapreduce.CommonFriendsDriver
$HADOOP_HOME/bin/hadoop jar $APP_JAR $PROG $INPUT $OUTPUT
```

## 准备输入/输出

```
$ hadoop fs -mkdir /common_friends
$ hadoop fs -mkdir /common_friends/input
$ hadoop fs -mkdir /common_friends/output
$ cat input/file1.txt
100 200 300 400 500
200 100 300 400
300 100 200 400 500
400 100 200 300

$ cat input/file2.txt
500 100 300
600 100

$ hadoop fs -copyFromLocal input/file1.txt /common_friends/input/
$ hadoop fs -copyFromLocal input/file2.txt /common_friends/input/

$ hadoop fs -ls /common_friends/input/
Found 2 items
-rw-r--r-- ... 74 ... /common_friends/input/file1.txt
-rw-r--r-- ... 20 ... /common_friends/input/file2.txt
```

## 运行脚本

```
$ ./run.sh
```

```
...
Deleted hdfs://localhost:9000/lib/common_friends.jar
Deleted hdfs://localhost:9000/common_friends/output
13/09/21 17:17:25 INFO CommonFriendsDriver: inputDir=/common_friends/input
13/09/21 17:17:25 INFO CommonFriendsDriver: outputDir=/common_friends/output
...
13/09/21 17:17:26 INFO mapred.JobClient: Running job: job_201309211704_0003
13/09/21 17:17:27 INFO mapred.JobClient:  map 0% reduce 0%
...
13/09/21 17:18:20 INFO mapred.JobClient:  map 100% reduce 100%
13/09/21 17:18:21 INFO mapred.JobClient: Job complete: job_201309211704_0003
13/09/21 17:18:21 INFO mapred.JobClient: Counters: 26
13/09/21 17:18:21 INFO mapred.JobClient:   Job Counters
...
13/09/21 17:18:21 INFO mapred.JobClient:   Map-Reduce Framework
13/09/21 17:18:21 INFO mapred.JobClient:     Map input records=6
13/09/21 17:18:21 INFO mapred.JobClient:     Reduce shuffle bytes=510
13/09/21 17:18:21 INFO mapred.JobClient:     Spilled Records=34
...
13/09/21 17:18:21 INFO mapred.JobClient:     Map output records=17
13/09/21 17:18:21 INFO CommonFriendsDriver: run(): status=true
13/09/21 17:18:21 INFO CommonFriendsDriver: jobStatus=0
```

### 检查输出

```
$ hadoop fs -cat /common_friends/output/part*
100,200 [300, 400]
100,300 [200, 400, 500]
100,400 [300, 200]
100,500 [300]
100,600 []
200,300 [400, 100]
200,400 [300, 100]
300,400 [200, 100]
300,500 [100]
```

## 解决方案2: 使用ArrayListOfLongsWritable的Hadoop实现

这个实现使用ArrayListOfLongsWritable[注1]类（一个定制类）表示一个“长整数列表”，这个类扩展了ArrayListOfLongs，并实现了WritableComparable<ArrayListOfLongsWritable>对象。只要对象（ArrayListOfLongsWritable）实现了Writable[注2]接口，映

注1：edu.umd.cloud9.io.array.ArrayListOfLongsWritable。

注2：org.apache.hadoop.io.Writable（这是一个可串行化的对象，基于DataInput和DataOutput实现了一个简单高效的串行化协议）。注意，Hadoop MapReduce框架中的所有键或值类型都实现了这个接口。

射器和归约器就可以将它用作为键或值。一般经验是，在Hadoop中，可以使用定制对象作为键或值（只要它实现了Writable接口）。Hadoop就是基于这一点在HDFS中持久存储对象。

表8-2所示的Hadoop类提供了一个使用定制数据类型查找共同好友的MapReduce解决方案。

**表8-2：使用定制数据类型查找共同好友用到的类**

| 类名 | 描述 |
|---|---|
| CommonFriendsDriverUsingList | 提交Hadoop作业的驱动器程序 |
| CommonFriendsMapperUsingList | 定义map() |
| CommonFriendsReducerUsingList | 定义reduce() |
| HadoopUtil | Hadoop的工具方法 |

# 解决方案2运行示例

## 脚本

下面的shell脚本可以启动使用定制数据类型查找共同好友的MapReduce程序驱动器：

```
$ cat run.sh
export JAVA_HOME=/usr/java/jdk7
export BOOK_HOME=/home/mp/data-algorithms-book
export APP_JAR=$BOOK_HOME/dist/data_algorithms_book.jar
input=/common_friends_using_lists/input
output=/common_friends_using_lists/output
prog=CommonFriendsDriverUsingList
$HADOOP_HOME/bin/hadoop fs -rmr $output
$HADOOP_HOME/bin/hadoop jar $APP_JAR $prog $input $output
```

## 运行脚本

```
$ ./run.sh
...
Deleted hdfs://localhost:9000/lib/common_friends_using_lists.jar
Deleted hdfs://localhost:9000/common_friends_using_lists/output
13/09/23 15:13:33 INFO ... inputDir=/common_friends_using_lists/input
13/09/23 15:13:33 INFO ... outputDir=/common_friends_using_lists/output
...
13/09/23 15:13:33 INFO mapred.JobClient: Running job: job_201309231108_0007
13/09/23 15:13:34 INFO mapred.JobClient: map 0% reduce 0%
...
13/09/23 15:14:24 INFO mapred.JobClient: map 100% reduce 100%
13/09/23 15:14:25 INFO mapred.JobClient: Job complete: job_201309231108_0007
13/09/23 15:14:25 INFO mapred.JobClient: Counters: 26
...
13/09/23 15:14:25 INFO mapred.JobClient: Map output bytes=636
```

```
...
13/09/23 15:14:25 INFO mapred.JobClient: Reduce output records=9
13/09/23 15:14:25 INFO mapred.JobClient: Map output records=17
13/09/23 15:14:25 INFO CommonFriendsDriverUsingList: run(): status=true
13/09/23 15:14:25 INFO CommonFriendsDriverUsingList: jobStatus=0
```

### 检查输出

```
$ hadoop fs -cat /common_friends_using_lists/output/part*
100,200 [400, 300]
100,300 [200, 500, 400]
100,400 [200, 300]
100,500 [300]
100,600 []
200,300 [100, 400]
200,400 [100, 300]
300,400 [100, 200]
300,500 [100]
```

## Spark解决方案

接下来提供一个Spark解决方案，这里使用Spark的RDD编写map()和reduce()函数。用户的数据表示为HDFS文本文件，每个记录行采用以下格式（P是某个人，{$F_1$, $F_2$, ..., $F_n$}是P的直接好友）：

P, $F_1$, $F_2$, ..., $F_n$

我们的Spark解决方案将建立在以下map()和reduce()函数基础上。下面给出映射器函数：

```
map(P, {F_1, F_2, ..., F_n}) {
   friends = {F_1, F_2, ..., F_n};
   for (f : friends) {
      key = buildSortedTuple(P, f);
      emit(key, friends);
   }
}
```

下面是归约器函数：

```
// key = Tuple2<user1,user2>
// values = List<List<user>>
reduce(key, values) {
   commonFriends = intersection(values);
   emit(key, friends);
}
```

## Spark程序

与传统MapReduce/Hadoop相比，由于Spark提供了一个更高层的API，整个解决方案可以用一个Java驱动器类表示。首先，我会给出全部步骤，然后再详细分析各个步骤。这个高层解决方案如示例8-5所示。

示例8-5：高层步骤

```
// 步骤1：导入必要的类和接口
public class FindCommonFriends {
   public static void main(String[] args) throws Exception {

     // 步骤2：检查输入参数
     // 步骤3：创建一个JavaSparkContext对象
     // 步骤4：从HDFS读取输入文本文件并
     // 创建第一个JavaRDD表示输入文件
     // 步骤5：将JavaRDD<String>映射到键-值对，
     // 其中key=Tuple<user1,user2>, value=好友列表
     // 步骤6：将(key=Tuple2<u1,u2>, value=List<friends>)对归约为
     // (key=Tuple2<u1,u2>, value=List<List<friends>>)
     // 步骤7：利用所有List<List<Long>>的交集查找共同好友

     System.exit(0);
   }

   // 建立一个有序的Tuple来避免重复
   static Tuple2<Long,Long> buildSortedTuple(long a, long b) {...}
}
```

接下来详细分析各个步骤。

### 步骤1：导入必要的类

Spark的主要Java类和接口在`org.apache.spark.api.java`包中定义。示例8-6展示了如何导入这些类和接口。

示例8-6：步骤1：导入必要的类和接口

```
// 步骤1：导入必要的类和接口
import scala.Tuple2;
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.api.java.JavaPairRDD;
import org.apache.spark.api.java.JavaSparkContext;
import org.apache.spark.api.java.function.PairFlatMapFunction;
import org.apache.spark.api.java.function.FlatMapFunction;
import org.apache.spark.api.java.function.Function;

import java.util.Arrays;
import java.util.List;
import java.util.ArrayList;
import java.util.Map;
import java.util.HashMap;
```

### 步骤2：检查输入参数

对于这个程序，需要一个输入参数：存储在HDFS中的输入文本文件（表示为args[0]，如示例8-7所示）。

示例8-7：步骤2：检查输入参数

```
1    // 步骤2：检查输入参数
2    if (args.length < 1) {
3      System.err.println("Usage: FindCommonFriends <file>");
4      System.exit(1);
5    }
6    System.out.println("HDFS input file ="+args[0]);
```

### 步骤3：创建一个JavaSparkContext对象

下一步如示例8-8所示，用来创建一个JavaSparkContext对象。这是创建新RDD的一个工厂对象。

示例8-8：步骤3：创建一个JavaSparkContext对象

```
1    // 步骤3：创建一个JavaSparkContext对象
2    JavaSparkContext ctx = new JavaSparkContext();
```

### 步骤4：读取输入文件并创建RDD

在示例8-9中，我们使用步骤3创建的上下文对象（JavaSparkContext）从一个输入文件创建了第一个RDD。这个新RDD（JavaRDD<String>）表示输入文件的所有记录。每个输入记录表示为一个Java字符串对象。

示例8-9：步骤4：读取输入并创建RDD

```
1    // 步骤4：从HDFS读取输入文本文件，并
2    //        创建第一个JavaRDD表示输入文件
3    JavaRDD<String> records = ctx.textFile(args[0], 1);
```

这里使用JavaRDD.collect()函数来调试步骤4：

```
// debug0
List<String> debug0 = records.collect();
for (String t : debug0) {
   System.out.println("debug0 record="+t);
}
```

### 步骤5：应用映射器

这一步将每个记录[P, $F_1$, $F_2$, ..., $F_n$]映射为一个键-值对集合，其中键是一个Tuple2(P, $F_i$)，值是一个列表[$F_1$, $F_2$, ..., $F_n$]。如果好友列表的大小为1，则不会创建好友列表（在这种情况下，不会有任何共同好友）。为了实现映射器，要使用

JavaRDD.flatMapToPair()函数。可以使用PairFlatMapFunction来实现，如下所示（这里T是输入，将生成（K, V）作为输出）：

```
PairFlatMapFunction<T, K, V>
T => Iterable<Tuple2<K, V>>
```

示例8-10给出了步骤5的完整实现。

示例8-10：步骤5：应用映射器

```
// 步骤5：将JavaRDD<String>映射为(key, value)对，
// 其中key=Tuple<user1,user2>, value=好友列表
//
// PairFlatMapFunction<T, K, V>
// T => Iterable<Tuple2<K, V>>
JavaPairRDD<Tuple2<Long,Long>,Iterable<Long>> pairs =
      records.flatMapToPair(new PairFlatMapFunction<
                                                    String,            // T
                                                    Tuple2<Long,Long>, // K
                                                    Iterable<Long>     // V
                                                   >() {
  public Iterable<Tuple2<Tuple2<Long,Long>,Iterable<Long>>> call(String s) {
    String[] tokens = s.split(",");
    long person = Long.parseLong(tokens[0]);
    String friendsAsString = tokens[1];
    String[] friendsTokenized = friendsAsString.split(" " );
    if (friendsTokenized.length == 1) {
      Tuple2<Long,Long> key = buildSortedTuple(person,
                                    Long.parseLong(friendsTokenized[0]));
      return Arrays.asList(new Tuple2<Tuple2<Long,Long>,
                           Iterable<Long>>(key, new ArrayList<Long>()));
    }
    List<Long> friends = new ArrayList<Long>();
    for (String f : friendsTokenized) {
      friends.add(Long.parseLong(f));
    }

    List<Tuple2<Tuple2<Long, Long>,Iterable<Long>>> result =
        new ArrayList<Tuple2<Tuple2<Long, Long>, Iterable<Long>>>();
    for (Long f : friends) {
      Tuple2<Long,Long> key = buildSortedTuple(person, f);
      result.add(new Tuple2<Tuple2<Long,Long>, Iterable<Long>>(key,
                 friends));
    }
    return result;
  }
 });
```

这里使用了JavaRDD.collect()函数来调试步骤5：

```
// debug1
List<Tuple2<Tuple2<Long, Long> ,Iterable<Long>>> debug1 = pairs.collect();
for (Tuple2<Tuple2<Long,Long>,Iterable<Long>> t2 : debug1) {
  System.out.println("debug1 key="+t2._1+"\t value="+t2._2);
```

```
}
```

## 步骤6：应用归约器

通过JavaPairRDD.groupByKey()方法来应用归约器，如示例8-11所示。

示例8-11：步骤6：应用归约器

```
// 步骤6：将(key=Tuple2<u1,u2>, value=Iterable<friends>)对归约为
// (key=Tuple2<u1,u2>, value=Iterable<Iterable<friends>>)
JavaPairRDD<Tuple2<Long, Long>, Iterable<Iterable<Long>>> grouped =
pairs.groupByKey();
```

我们使用了以下代码来调试这个步骤：

```
// debug2
List<Tuple2<Tuple2<Long, Long> ,Iterable<Iterable<Long>>>> debug2 =
    grouped.collect();
for (Tuple2<Tuple2<Long,Long>, Iterable<Iterable<Long>>> t2 : debug2) {
    System.out.println("debug2 key="+t2._1+"\t value="+t2._2);
}
```

## 步骤7：查找共同好友

要查找共同好友，只需要修改值来得到两个用户所有好友的交集。可以使用JavaPairRDD.mapValues()方法来完成（无需改变键），如示例8-12所示。

示例8-12：步骤7：查找所有好友的交集

```
// 步骤7：查找所有List<List<Long>>的交集
// mapValues[U](f: (V) => U): JavaPairRDD[K, U]
// 通过一个映射函数传入键-值对RDD中的各个值，
// 而不改变键；
// 这还会保留原RDD的分区
JavaPairRDD<Tuple2<Long, Long>, Iterable<Long>> commonFriends =
    grouped.mapValues(new Function< Iterable< Iterable<Long>>, // 输入
                                                Iterable<Long> // 输出
                                              >() {
  public Iterable<Long> call(Iterable<Iterable<Long>> s) {
     Map<Long, Integer> countCommon = new HashMap<Long, Integer>();
     int size = 0;
     for (Iterable<Long> iter : s) {
        size++;
        List<Long> list = iterableToList(iter);
        if ((list == null) || (list.isEmpty())) {
           continue;
        }
        for (Long f : list) {
           Integer count = countCommon.get(f);
           if (count == null) {
              countCommon.put(f, 1);
           }
           else {
             countCommon.put(f, ++count);
```

```
26              }
27           }
28        }
29
30        // 如果countCommon.Entry<f, count> == countCommon.Entry<f, s.size()>
31        // 则有一个共同好友
32        List<Long> finalCommonFriends = new ArrayList<Long>();
33        for (Map.Entry<Long, Integer> entry : countCommon.entrySet()) {
34           if (entry.getValue() == size) {
35              finalCommonFriends.add(entry.getKey());
36           }
37        }
38        return finalCommonFriends;
39     }
40  });
```

我们使用了以下代码来调试步骤7：

```
// debug3
List<Tuple2<Tuple2<Long, Long>, Iterable<Long>>> debug3 =
   commonFriends.collect();
for (Tuple2<Tuple2<Long, Long>, Iterable<Long>> t2 : debug3) {
   System.out.println("debug3 key="+t2._1+ "\t value="+t2._2);
}
```

最后，使用以下函数确保不会得到重复的Tuple2<user1,user2>对象：

```
static Tuple2<Long,Long> buildSortedTuple(long a, long b) {
   if (a < b) {
      return new Tuple2<Long, Long>(a,b);
   }
   else {
      return new Tuple2<Long, Long>(b,a);
   }
}
```

## 合并步骤6和步骤7

可以把步骤6和步骤7合并到一个Spark步骤/操作中，这可以通过Spark的combineByKey()或reduceByKey()函数来完成（这两个转换器会把有相同键的值组合在一起）。下面使用reduceByKey()，这是combineByKey()的一个简单版本。reduceByKey()可以把类型为V的值归约为V（相同的数据类型）。例如，整数值相加或相乘，另外，也可以将类型为V的值组合/转换为另一种类型C。例如，可以组合/转换整数（V）值为一个整数集合（Set<Integer>）。最简形式的reduceByKey()签名如下：

```
public JavaPairRDD<K,V> reduceByKey(Function2<V,V,V> func)
// 使用一个关联的归约函数合并各个键的值。
// 在把结果发送到归约器之前，
// 还会在各个映射器上完成本地合并，
// 类似MapReduce中的一个"组合器"。输出将采用
// 现有的分区器/并行等级散列分区
```

下面使用reduceByKey()合并步骤6和步骤7。首先，明确输入和输出，如表8-3所示。

**表8-3：输入、输出和转换器**

| | |
|---|---|
| 输入 | pairs (K: <Tuple2<Long,Long>, V: Iterable<Long>) |
| 输出 | commonFriends (K: String, V: Iterable<Long>) |
| 转换器 | reduceByKey() |

示例8-13展示了如何利用reduceByKey()转换器，它会找出好友的交集，表示为一个Iterable<Long>。

示例8-13：使用combineByKey()

```
import com.google.common.collect.Sets;
...
JavaPairRDD<Tuple2<Long,Long>, Iterable<Long>>
    commonFriends = pairs.reduceByKey(new Function2<
                                                    Iterable<Long>,
                                                    Iterable<Long>
                                                   >() {
    // 返回a和b的交集
    public Iterable<Long> call(Iterable<Long> a, Iterable<Long> b) {
       Set<Long> x = Sets.newHashSet(a);
       Set<Long> intersection = new HashSet<Long>();
       for (Long item : b) {
          if (x.contains(item)) {
             intersection.add(item);
          }
       }
       return intersection;
    }
});

// 发出最终输出
Map<String, Set<String>> commonFriendsMap = commonFriends.collectAsMap();
for (Entry<Tuple2<Long,Long>, Iterable<Long>> entry :
    commonFriendsMap.entrySet()) {
    System.out.println(entry.getKey() + ":" + entry.getValue());
}
```

# Spark程序运行示例

下面的小节给出这个Spark程序的示例输入、脚本和期望输出。

## HDFS输入

下面是Spark解决方案的HDFS输入：

```
# hadoop fs -cat /data/users_and_friends.txt
100,200 300 400 500
200,100 300 400
```

```
300,100 200 400 500
400,100 200 300
500,100 300
600,100
```

## 脚本

运行这个Spark程序的脚本如下：

```
$ cat run_find_common_friends.sh
#!/bin/bash
export JAVA_HOME=/usr/java/jdk7
export SPARK_HOME=/home/hadoop/spark-1.1.0
export SPARK_MASTER=spark://myserver100:7077
BOOK_HOME=/mp/data-algorithms-book
APP_JAR=$BOOK_HOME/dist/data_algorithms_book.jar
USERS=/data/users_and_friends.txt
# 在一个Spark集群上运行
prog=org.dataalgorithms.chap08.spark.FindCommonFriends
$SPARK_HOME/bin/spark-submit \
--class $prog \
--master $SPARK_MASTER \
--executor-memory 2G \
--total-executor-cores 20 \
$APP_JAR \
$USERS
```

## 运行示例日志

我们在一个集群环境中运行这个程序，这里使用3个服务器，分别标识为myserver100（Spark主节点）、myserver200和myserver300。为适应版面，这里对实际输出日志做了删减，并对格式有所调整：

```
# ./run_find_common_friends.sh
HDFS input file =/data/users_and_friends.txt
...
14/05/31 21:33:40 INFO Remoting: Starting remoting
14/05/31 21:33:40 INFO Remoting: Remoting started; listening on addresses :
  [akka.tcp://spark@myserver100 :33722]
14/05/31 21:33:40 INFO Remoting: Remoting now listens on addresses:
  [akka.tcp://spark@myserver100 :33722]
...
14/05/31 21:33:46 INFO scheduler.TaskSchedulerImpl: Removed TaskSet 0.0,
  whose tasks have all completed, from pool
14/05/31 21:33:46 INFO spark.SparkContext: Job finished: collect at
  FindCommonFriends.java:33, took 3.76636011 s
debug0 record=100,200 300 400 500
debug0 record=200,100 300 400
debug0 record=300,100 200 400 500
debug0 record=400,100 200 300
debug0 record=500,100 300
debug0 record=600,100
```

```
14/05/31 21:33:46 INFO spark.SparkContext: Starting job: collect at
  FindCommonFriends.java:69
14/05/31 21:33:46 INFO scheduler.DAGScheduler: Got job 1 (collect at
  FindCommonFriends.java:69) with 1 output partitions (allowLocal=false)
...
14/05/31 21:33:46 INFO scheduler.DAGScheduler: Stage 1 (collect at
  FindCommonFriends.java:69) finished in 0.043 s
14/05/31 21:33:46 INFO spark.SparkContext: Job finished: collect at
  FindCommonFriends.java:69, took 0.051576509 s
debug1 key=(100,200) value=[200, 300, 400, 500]
debug1 key=(100,300) value=[200, 300, 400, 500]
debug1 key=(100,400) value=[200, 300, 400, 500]
debug1 key=(100,500) value=[200, 300, 400, 500]
debug1 key=(100,200) value=[100, 300, 400]
debug1 key=(200,300) value=[100, 300, 400]
debug1 key=(200,400) value=[100, 300, 400]
debug1 key=(100,300) value=[100, 200, 400, 500]
debug1 key=(200,300) value=[100, 200, 400, 500]
debug1 key=(300,400) value=[100, 200, 400, 500]
debug1 key=(300,500) value=[100, 200, 400, 500]
debug1 key=(100,400) value=[100, 200, 300]
debug1 key=(200,400) value=[100, 200, 300]
debug1 key=(300,400) value=[100, 200, 300]
debug1 key=(100,500) value=[100, 300]
debug1 key=(300,500) value=[100, 300]
debug1 key=(100,600) value=[]
14/05/31 21:33:46 INFO spark.SparkContext: Starting job: collect at
  FindCommonFriends.java:78
14/05/31 21:33:46 INFO scheduler.DAGScheduler: Registering RDD 2 (flatMap
  at FindCommonFriends.java:41)
14/05/31 21:33:46 INFO scheduler.DAGScheduler: Got job 2 (collect at
  FindCommonFriends.java:78) with 1 output partitions (allowLocal=false)
...
14/05/31 21:33:46 INFO scheduler.DAGScheduler: Stage 2 (collect at
  FindCommonFriends.java:78) finished in 0.212 s
14/05/31 21:33:46 INFO spark.SparkContext: Job finished: collect at
  FindCommonFriends.java:78, took 0.335984028 s
debug2 key=(200,300) value=[[100, 300, 400], [100, 200, 400, 500]]
debug2 key=(100,300) value=[[200, 300, 400, 500], [100, 200, 400, 500]]
debug2 key=(100,200) value=[[200, 300, 400, 500], [100, 300, 400]]
debug2 key=(300,400) value=[[100, 200, 400, 500], [100, 200, 300]]
debug2 key=(100,500) value=[[200, 300, 400, 500], [100, 300]]
debug2 key=(200,400) value=[[100, 300, 400], [100, 200, 300]]
debug2 key=(100,400) value=[[200, 300, 400, 500], [100, 200, 300]]
debug2 key=(100,600) value=[[]]
debug2 key=(300,500) value=[[100, 200, 400, 500], [100, 300]]
14/05/31 21:33:46 INFO spark.SparkContext: Starting job: collect at
  FindCommonFriends.java:122
14/05/31 21:33:46 INFO scheduler.DAGScheduler: Got job 3 (collect at
  FindCommonFriends.java:122) with 1 output partitions (allowLocal=false)
...
14/05/31 21:33:47 INFO scheduler.DAGScheduler: Stage 4 (collect at
  FindCommonFriends.java:122) finished in 0.068 s
14/05/31 21:33:47 INFO scheduler.TaskSchedulerImpl: Removed TaskSet 4.0,
  whose tasks have all completed, from pool
```

```
14/05/31 21:33:47 INFO spark.SparkContext: Job finished: collect at
  FindCommonFriends.java:122, took 0.084085412 s
debug3 key=(200,300) value=[100, 400]
debug3 key=(100,300) value=[200, 500, 400]
debug3 key=(100,200) value=[400, 300]
debug3 key=(300,400) value=[100, 200]
debug3 key=(100,500) value=[300]
debug3 key=(200,400) value=[100, 300]
debug3 key=(100,400) value=[200, 300]
debug3 key=(100,600) value=[]
debug3 key=(300,500) value=[100]
```

这一章提供了一个简单的MapReduce解决方案，用来查找所有社交网络用户的共同好友。下一章将介绍如何使用MapReduce框架实现基本的推荐引擎。

第9章

# 使用MapReduce实现推荐引擎

这一章讨论如何使用MapReduce算法实现推荐引擎。

如果你经常光顾Amazon.com，可能很熟悉这个网站提供的相关商品列表（图书、音像制品等），这些列表可以帮助顾客找到他们想要找的东西。Amazon.com在每个页面上会提供很多这样的列表，包括“经常一起购买的商品”和“购买过该商品的顾客还购买了哪些商品”等。这些特性的基础就是推荐引擎和系统。一般地，推荐引擎和系统会在以下方面改善用户体验：

- 帮助用户查找信息。
- 减少搜索和导航时间。
- 提高用户满意度，而且可以激励用户经常光顾网站。

推荐引擎或系统可以预测或推荐：

- 用户还没有评级、购买或访问的商品。
- 用户还没有考虑过的电影或图书。
- 用户还没有去过的餐馆或场所。

近年来，推荐系统已经非常普遍。下面给出这种系统的一些例子：

- Amazon.com和MyBuys.com，这些网站提供的推荐系统可以向用户推荐他们可能购买的类似商品，换句话说，用户会看到其他顾客在购买他们当前选择的商品时还同时购买了哪些其他商品。
- Tripbase.com，这是一家旅游网站，会根据用户的输入或偏好推荐旅游团。

- Netflix，可以根据用户之前的评分和观看习惯（相对于其他用户的行为）来预测用户可能喜欢看哪些电影。

这一章我们将研究以下方面，这些特性的基础都是推荐引擎和系统：

- 购买了该商品的顾客还购买了哪些商品。
- 经常一起购买的商品。
- 推荐连接。

关于推荐系统的详细信息，可以参考[1]、[26]和[11]。

# 购买过该商品的顾客还购买了哪些商品

大多数电子商务经销商，包括Amazon.com，都在网站上使用“购买过该商品的顾客还购买了哪些商品”（CWBTIAB）特性来推荐图书、音像制品或其他商品。这里我们将建立一个简单的推荐系统来实现这个CWBTIAB特性。

假设Amazon.com日志中对应每个销售记录包含一个`user-id`和`bought-item`。我们将使用MapReduce范式实现CWBTIAB功能。只要显示一个商品，Amazon.com就会推荐购买过这个商品的用户最常购买的其他5件商品。

## 输入

假设输入是一个大交易集（交易日志包含大量数据，包括交易ID、日期、价格等），其中有以下字段：

```
<user-id><,><bought-item>
```

## 期望输出

推荐引擎要发出键-值对，其中键是商品，值是一个列表，包含购买过这个商品的顾客最常购买的5件商品。

## MapReduce解决方案

我们将利用MapReduce的两次迭代实现CWBTIAB功能：

- 阶段1: 生成同一个用户购买的所有商品的列表。分组由Hadoop框架处理，其中映射器和归约器都会完成一个恒等函数。
- 阶段 2: 解决列表商品的共现问题。我们使用Stripes（条纹）设计模式，只发出5个最常见的共现商品。

在讨论这两个阶段之前，下面先通过一个简单的示例解释Stripes设计模式的概念。

## Stripes设计模式

Stripes（条纹）作为一个设计模式，它的主要思想是将键-值对分组为一个关联数组。考虑表9-1所示的传统案例，这里给出了一个映射器发出的键-值对（需要说明，在这个例子中，映射器的输出键是一个组合键Tuple2）。

**表9-1：映射器输出：传统方法**

| 键 | 值 |
|---|---|
| $(k, k_1)$ | 3 |
| $(k, k_2)$ | 2 |
| $(k, k_3)$ | 4 |
| $(k, k_4)$ | 6 |
| $(z, z_1)$ | 7 |
| $(z, z_2)$ | 8 |
| $(z, z_3)$ | 5 |

Stripes方法的基本思想则不是发出很多键-值对，我们只对每个条纹发出一个键-值对，如表9-2所示（注意在这个例子中，k和z是自然键）。

**表9-2：映射器输出：Stripes方法**

| 键 | 值 |
|---|---|
| k | $\{(k_1, 3), (k_2, 2), (k_3, 4), (k_4, 6)\}$ |
| z | $\{(z_1, 7), (z_2, 8), (z_3, 5)\}$ |

Stripes方法为每个自然键创建一个关联数组（或散列表），并归约各个映射器发出的键-值对数。各个映射器发出的值转换为一个复杂对象（关联数组）时，如果采用Stripes方法，可以减少键-值对的排序和洗牌。

Stripes方法中归约器是如何工作的？归约器会对关联数组完成一个元素级求和。考虑下面的例子，这是一个归约器的输入，包含3个键-值对：

```
K -> { (a, 1), (b, 2), (c, 4), (d, 3) }
K -> { (a, 2), (c, 2) }
K -> { (a, 3), (b, 5), (d, 5) }
```

将生成以下输出：

```
K -> { (a, 1+2+3), (b, 2+5), (c, 4+2), (d, 3+5) }
```

或：

```
K -> { (a, 6), (b, 7), (c, 6), (d, 8) }
```

Stripes方法的优点如下：

- 与传统方法相比，由于映射器生成的键-值对更少，所以需要的排序和洗牌也更少。
- 采用Stripes方法，我们可以充分利用组合器（完成各个节点上的本地优化）。
- Stripes方法可以提供更好的性能[14]。

Stripes方法的缺点包括：

- 较难实现（因为各个映射器发出的值是一个关联数组，必须为这个关联数组写一个串行化器和逆串行化器）。
- 底层对象（映射器生成的值，即关联数组）是更重量级的对象。
- Stripes方法在事件空间（event space）大小方面存在一个基本限制（因为要为每个自然键创建一个关联数组，需要确保映射器有足够的RAM来保存这些散列表）。

## MapReduce阶段1

第1个MapReduce阶段会生成同一个用户购买的所有商品的列表。由MapReduce框架根据userID（作为键）完成分组。映射器和归约器都会完成一个恒等函数。阶段1的目标就是找出所有用户购买的所有商品。

映射器是一个恒等函数，会发出接收到的键-值对（参见示例9-1）。

示例9-1：映射器：阶段1

```
1 // key = userID
2 // value = userID购买的商品
3 map(userID, item) {
4    emit(userID, item);
5 }
```

归约器也是一个恒等函数，会对一个用户的所有商品分组（参见示例9-2）。

示例9-2：归约器：阶段1

```
1 // key = userID
2 // value = userID购买的商品的列表
3 reduce(userID, items[I1, I2, ..., In]) {
4     emit(userID, items);
5 }
```

## MapReduce阶段2

第2个MapReduce阶段解决列表商品的共现问题。采用Stripes方法，映射器会完成大部分工作，聚集数据，然后把数据传递到组合器和归约器。归约器再发出期望的输出（首先是商品，然后是包含5个最常见共现商品的列表）。

由于我们可能要为各个映射器/归约器创建多个散列表（关联数组），需要确保有足够的内存来保存这些数据结构。如果用户数或商品数足够大，就可能无法全部放在内存中。如果你的内存/RAM是有限的，可能要考虑在硬盘上创建这些散列表（MapDB提供了这样一个解决方案）[注1]。

在示例9-3中可以看到，这个阶段中的映射器包含组合器功能（使用Stripes方法时，映射器会完成大部分工作，主要由map()和combine()函数实现）。前面已经提到，Stripes方法会尽可能减少生成的键数，因此MapReduce执行框架需要完成的洗牌和排序会更少。不过，由于我们为值使用了一个非基本类型（关联数组），所以需要更多的串行化和逆串行化。

示例9-3：映射器：阶段2

```
1 // key = userID
2 // value = userID购买的商品的列表
3 map(userID, items[I1, I2, ..., In]) {
4     for (Item item : items) {
5       Map<Item, Integer> map = new HashMap<Item, Integer>();
6       for (Item j : items) {
7          map(j) = map(j) + 1;
8       }
9       emit(item, map);
10    }
11 }
```

如示例9-4所示，在这个阶段，对于所有交易中的每一件商品，归约器会生成与这个商品最常一起购买的“top 5”商品。归约器对一个给定商品的所有条纹（表示为一个关联数组）完成元素级求和。

示例9-4：归约器：阶段2

```
1 // key = 商品
2 // value = 条纹列表[M1, M2, ..., Mm]
3 reduce(item, stripes[M1, M2, ..., Mm]) {
4     Map<Item, Integer> final = new HashMap<Item, Integer>();
5     for (Map<Item, Integer> map : stripes) {
6        for (all (k, v) in map) {
7           final(k) = final(k) + v;
8        }
9     }
10    emit(key, top(5, final))
11 }
```

top(N, Map<Item, Integer>)会返回N项，表示一个给定关联数组的最高/最大频度项。

---

注1：MapDB（*http://www.mapdb.org/*）由Jan Kotek开发，提供了由硬盘存储空间或堆外内存支持的并发映射、集合和队列。这是一个内嵌的Java数据库引擎，速度很快而且很容易使用。

# 经常一起购买的商品

这一节使用MapReduce/Hadoop实现“经常一起购买的商品” (FBT)特性。FBT是一种用户行为定向技术，充分利用用户之前的购买历史来选择和显示这个用户可能希望购买的其他相关商品。假设你在Amazon.com上搜索Donald Knuth的《The Art of Computer Programming》。在商品详细页面上，会看到“经常一起购买的商品”部分，其中会列出你搜索的这件商品，另外还会列出经常与它一同购买的其他图书（见图9-1）。

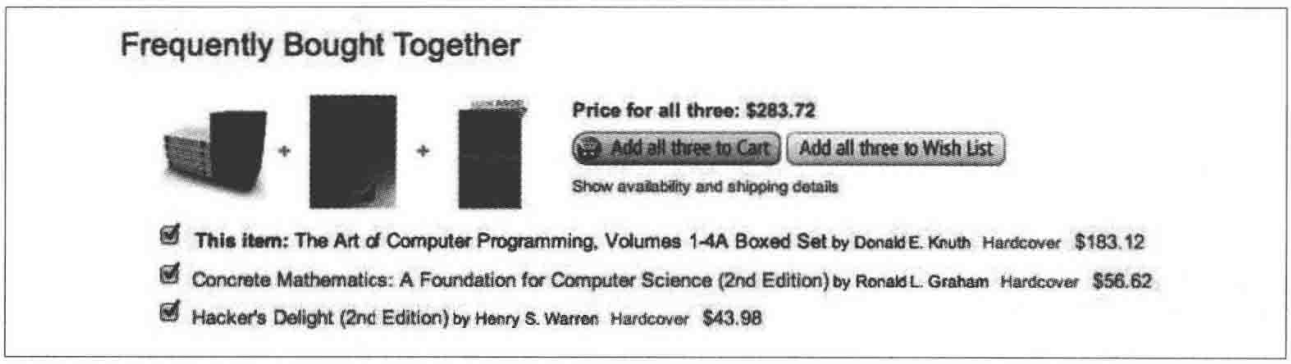



图9-1：“经常一起购买的商品”特性

Amazon是如何为大部分商品提供这个列表的呢？基本说来，它会对商品之间的关系完成一个搜索（如图书和音像制品）。一般的，类似Amazon.com的电子商务网站会收集顾客购买习惯的其他有关数据。通过使用关联规则学习，这些网站可以确定哪些商品经常一起购买，并利用这个信息来进行市场营销。有时这称为购物篮分析的一个变种（购物篮分析是一个有名的数据挖掘技术，本书第7章做过介绍）。

# 输入和期望输出

假设我们得到所有顾客的商品销售交易作为输入。下面再假设有$n$个交易（标记为$T_1$, ..., $T_n$）和$m$个商品（标记为$P_1$, ..., $P_m$），组合得到的输入如表9-3所示。

**表9-3：所有顾客的商品销售交易**

| 交易 | 购买的商品 |
|---|---|
| $T_1$ | $\{P_{1,1}, P_{1,2}, ..., P_{1,k1}\}$ |
| $T_2$ | $\{P_{2,1}, P_{2,2}, ..., P_{2,k2}\}$ |
| ... | ... |
| $T_n$ | $\{P_{n,1}, P_{n,2}, ..., P_{n,kn}\}$ |

在这个表中：

- $P_{i,j} \in \{P_1, ..., P_m\}$.

- $k_i$是交易$T_i$中购买的商品数量。
- 每个输入行包括一个交易ID，后面是购买的商品列表。

我们的目标是建立一个散列表，键为$P_i$（$i = 1, 2, ..., m$），值是一起购买的商品的列表。

例如，假设有表9-4所示的输入。

**表9-4：FBT示例的输入**

| 交易 | 购买的商品 |
|---|---|
| $T_1$ | $\{P_1, P_2, P_3\}$ |
| $T_2$ | $\{P_2, P_3\}$ |
| $T_3$ | $\{P_2, P_3, P_4\}$ |
| $T_4$ | $\{P_5, P_6\}$ |
| $T_5$ | $\{P_3, P_4\}$ |

那么期望的输出如表9-5所示。

**表9-5：FBT示例的期望输出**

| 商品 | 经常一起购买的商品 |
|---|---|
| $P_1$ | $\{P_2, P_3\}$ |
| $P_2$ | $\{P_1, P_3, P_4\}$ |
| $P_3$ | $\{P_1, P_2, P_4\}$ |
| $P_4$ | $\{P_2, P_3\}$ |
| $P_5$ | $\{P_6\}$ |
| $P_6$ | $\{P_5\}$ |

因此，如果一个顾客在浏览商品$P_3$，就可以得出经常一起购买的商品是$P_1$、$P_2$和$P_4$。

## MapReduce解决方案

`map()`函数由一个交易生成一组键-值对，再由`reduce()`处理/使用。映射器将交易项（即商品）配对作为键，这个键在交易中的出现次数作为相应的值（包括所有交易但不带交易ID，这里会忽略交易ID）。

例如，对于交易1（$T_1$），`map()`会发出以下键-值对：

```
[<P1, P2>, 1]
[<P1, P3>, 1]
[<P2, P3>, 1]
```

需要说明，如果简单地选择一个交易中的两个商品作为键，商品对的出现次数并不正确。例如，如果交易$T_1$和$T_2$有以下商品（商品相同，但是顺序不同）：

```
T1: (P1, P2, P3)
T2: (P1, P3, P2)
```

那么对于交易$T_1$，map()将生成：

```
[<P1, P2>, 1]
[<P1, P3>, 1]
[<P2, P3>, 1]
```

对于交易$T_2$，map()将生成：

```
[<P1, P3>, 1]
[<P1, P2>, 1]
[<P3, P2>, 1]
```

从交易$T_1$和$T_2$的map()输出可以看到，总共得到了6个不同的商品对，每个商品对只出现1次，但实际上只有3个不同的商品对。也就是说，键（P2, P3）和（P3, P2）原本是一样的，在这里却视为不同的键。我们知道这是不正确的。如果在生成键-值对之前按字母顺序对交易商品排序，就可以避免这个问题。对交易的商品排序之后，我们可以得到：

```
sorted T1: (P1, P2, P3)
sorted T2: (P1, P2, P3)
```

现在每个交易（$T_1$和$T_2$）都会有以下3个键-值对：

```
[<P1, P2>, 1]
[<P1, P3>, 1]
[<P2, P3>, 1)
```

如下累加这两个交易的出现值：[<P1, P2>, 2], [<P1, P3>, 2], [<P2, P3>,2]。可以得到正确的总出现数。

## 映射器

映射器如示例9-5所示，它会读取输入数据，并为每个交易创建一个商品列表。对于每个交易，其时间复杂度为O($n$)，这里$n$是一个交易的商品数。然后，对交易列表中的商品排序，以避免（P2, P3）和（P3, P2）之类重复的键。快速排序的时间复杂度是O($n$ log $n$)。接下来，已排序的交易商品转换为商品对作为键，这是一个交叉操作，从而生成列表中商品的交叉对。

示例9-5：经常一起购买的商品：map()函数

```
1 // key 是交易ID，在这里忽略
2 // value = 交易商品(P1, P2, ...,Pm)
3 map(key, value) {
```

```
    (S1, S2, ..., Sm) = sort(P1, P2, ...,Pm);
    // 现在可以保证： S1 < S2 < ... < Sm
    ListOfPair<Si, Sj> = generateCombinations(S1, S2, ..., Sm)
    for ( (Si, Sj) pair : ListOfPair<Si, Sj>) {
        // 归约器键: (Si, Sj)
        // 归约器值: integer 1
        emit([(Si, Sj), 1]);
    }
}
```

generateCombinations(S1, S2, ..., Sm)生成一个给定交易中的所有两两商品组合。例如，generateCombinations(S1, S2, S3, S4)会返回以下商品对：

```
(S1, S2)
(S1, S3)
(S1, S4)
(S2, S3)
(S2, S4)
(S3, S4)
```

最后，map()会输出以下键-值对：

```
<P1, P2> 1
<P1, P3> 1
<P2, P3> 1
<P2, P3> 1
<P2, P3> 1
<P3, P4> 1
<P5, P6> 1
<P1, P5> 1
<P3, P4> 1
```

## 归约器

归约器的FBT算法如示例9-6所示。这个归约器对各个归约器键的值数求和。因此，它的时间复杂度是O($v$)，这里v是每个键的值数。

示例9-6：经常一起购买的商品: reduce()函数

```
// key 形式为(Si, Sj)
// value = List<integer>, 其中各个元素分别是一个整数
reduce(key, value) {
    int sum = 0;
    for (int i : List<integer>) {
        sum += i;
    }
    emit(key, sum);
}
```

## 归约器输出

归约器会创建以下输出格式：

```
<Pi, Pj>, N
```

这里$N$是交易数，商品$P_i$和$P_j$是一起购买的商品。$N$越大，这两个商品之间的关系就越紧密。现在，利用这个输出，可以创建所需的散列表，其中键是$\{P_1, P_2, ..., P_m\}$。

对于我们的输入，归约器会生成以下输出：

```
<P1, P2> 1
<P1, P3> 1
<P2, P3> 3
<P2, P4> 1
<P3, P4> 2
<P5, P6> 1
```

# 推荐连接

在这一节中，我会提供一个完整的基于Spark的MapReduce算法，来推荐相互有关联的人。Spark提供了一个GraphX API（*http://spark.apache.org/graphx/*）来完成图和图并行计算，不过在我们不打算使用这个API，这里只使用Spark API。

目前有大量社交网络网站（Facebook、Instagram、LinkedIn、Pinterest等）。它们有一个共同的特性，就是可以推荐联系人。例如，LinkedIn的“你可能认识的人”特性允许会员查看他们可能想联系的人。基本思想是：如果Alex是Bob的好友，而Alex又是Barbara的好友（也就是说，Alex是Bob和Barbara的共同好友，但是Bob和Barbara彼此并不认识），那么社交网络系统就会推荐Bob与Barbara联系，或者反之，向Barbara推荐Bob。换句话说，如果两个人有一组共同好友，但是这两个人本身不是好友，那么这个MapReduce解决方案就会推荐他们相互联络。

所有用户之间的好友关系可以表示为一个图。对于我们这个简单例子，可以使用图9-2来表示。

在数学中，图是一个有序对$G= (V, E)$，包括顶点或节点集$V$，以及边或线集$E$，边集$E$是$V$的两元素子集（也就是说，边与两个顶点相关，这个关系表示为关于这个特定边的一个无序的顶点对）。这种图可以准确地描述为无向图或简单图。在我们的MapReduce解决方案中，假设人们之间的好友关系可以用一个无向图表示（如果$A$是$B$的一个好友，那么$B$也是$A$的好友）。大多数社交网络（如Facebook和LinkedIn）都使用双向好友关系，不过Twitter上的好友关系是单向的。

在这里，图是一个有序对$G= (V, E)$，其中：

- $V$是由人（社交网络的用户）构成的一个有限集。
- $E$是$V$上的一个二元关系，称为边集，其中的元素称为一个好友关系。

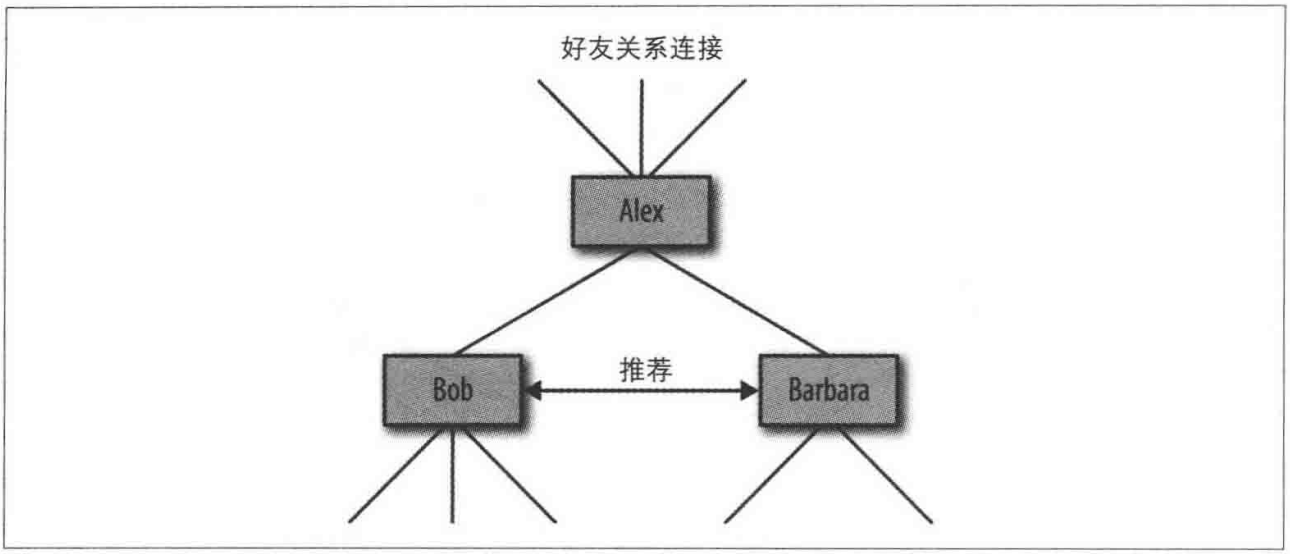



图9-2：好友关系连接

从图论的角度来看，对于一个特定社交网络的各个人或成员来说，如果他在某个人A的两度以内，我们会统计这个人与A之间存在多少条不同的路径（包括两个连接边）。然后根据路径数对这个列表中的所有人评分，显示出A要联系的前10个人。我会介绍如何使用MapReduce解决方案为每个人计算这个top 10好友连接列表。因此，我们的目标是为社交网络中的每一个成员预计算前10名推荐好友（作为一个批作业）。

好友推荐问题可以表述为：对于每个人P，我们要确定一个列表$P_1$, $P_2$, ..., $P_{10}$，这是与P有最多共同好友的10个人。

## 输入

社交网络图通常非常稀疏。这里我们假设输入记录是一个按名字排序的邻接表[注2]。因此，输入中的每一行分别包括一个成员ID（如用户ID，表示为P），后面是他的直接好友列表，表示为$F_1$, $F_2$, ..., $F_n$：

$$P\ F_1\ F_2\ \ldots\ F_n$$

这里$F_1 < F_2 < \ldots < F_n$。

考虑一个包括8个人的社交网络，这8个人分别用数字1到8表示。进一步假设他们的好友关系图如图9-3所示（好友关系图）。

注2：在图论和计算机科学中，图的邻接表表示是一个无序列表集合，图中的各个顶点分别有一个列表。每个列表描述这个顶点的相邻顶点集合（资料来源：*http://en.wikipedia.org/wiki/Adjacency_list*）。

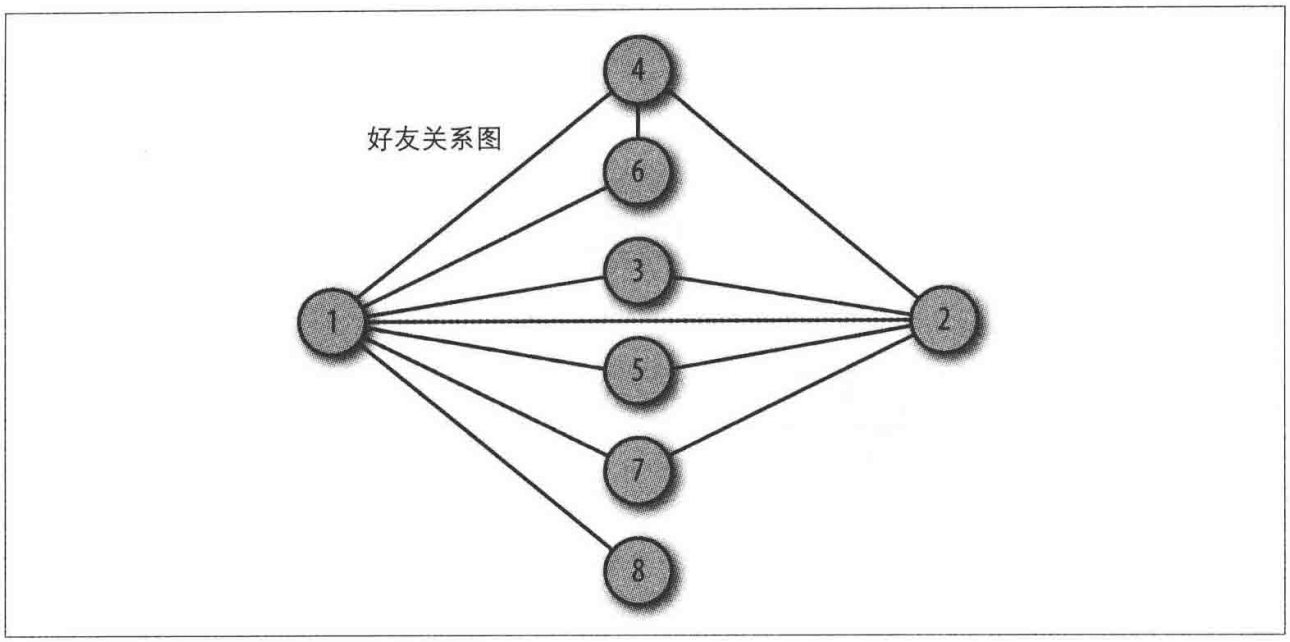



图9-3：8个好友的好友关系图

这个图的邻接表（按ID排序）如下表示（我们将从HDFS读取输入）：

```
# hadoop fs -cat /data/friendS2.txt
1 2,3,4,5,6,7,8
2 1,3,4,5,7
3 1,2
4 1,2,6
5 1,2
6 1,4
7 1,2
8 1
```

用户1与所有人都是好友，所以不用向这个用户推荐任何人。另一方面，用户3与用户1和2是好友；我们可以向用户3推荐用户4、5、6、7和8，因为他们是用户3与用户1和/或用户2的共同好友。

## 输出

这个Spark程序的输出采用以下格式：

```
<USER><:><F(M: [I1, I2, I3, ...]), ...>
```

其中：

- F是推荐给USER的一个好友。
- M是共同好友数。
- I1, I2, I3, ...是共同好友的ID。

对于我们的示例输入，期望的输出为：

```
4: 3 (2: [1, 2]),5 (2: [1, 2]),7 (2: [1, 2]),8 (1: [1]),
2: 6 (2: [1, 4]),8 (1: [1]),
6: 2 (2: [1, 4]),3 (1: [1]),5 (1: [1]),7 (1: [1]),8 (1: [1]),
8: 2 (1: [1]),3 (1: [1]),4 (1: [1]),5 (1: [1]),6 (1: [1]),7 (1: [1]),
3: 4 (2: [1, 2]),5 (2: [1, 2]),6 (1: [1]),7 (2: [1, 2]),8 (1: [1]),
1:
7: 3 (2: [1, 2]),4 (2: [1, 2]),5 (2: [1, 2]),6 (1: [1]),8 (1: [1]),
5: 3 (2: [1, 2]),4 (2: [1, 2]),6 (1: [1]),7 (2: [1, 2]),8 (1: [1]),
```

再次说明，用户1没有得到任何好友推荐，因为这个用户与所有人都已经是好友。这个输出从图中也可以得到验证。

## MapReduce解决方案

这一节将提供一个通用MapReduce解决方案，这里没有使用任何特定的框架实现。要实现好友推荐特性，我们只需要一个映射器和一个归约器。映射器如示例9-7所示，可以找出直接好友和将来可能的好友（也就是推荐的好友）。

示例9-7：推荐连接：映射器

```
// key = person (对应userID的Long)
// value = friends = List<userID> = person的直接好友 ({F1, F2, F3, ...})
map(key, friends) {
   // 发出所有直接好友关系
   for (friend : friends) {
      // -1指示直接好友关系
      directFriend = Tuple2(friend, -1);
      emit(key, directFriend);
   }

   // 发出所有可能的好友关系，
   // 他们有共同好友person(作为键)
   for (int i = 0; i < friends.size(); i++) {
      for (int j = i + 1; j < friends.size(); j++) {
         // 可能的好友1
         possibleFriend1 = Tuple2(friends.get(j), person);
         emit(friends.get(i), possibleFriend1);
         // 可能的好友2
         possibleFriend2 = Tuple2(friends.get(i), person);
         emit(friends.get(j), possibleFriend2);
      }
   }
}
```

归约器如示例9-8所示，它会找出键（某个特定的人）与值（Tuple2<Long, Long>列表）的共同好友。如果某个value有一个共同好友（由特殊long值-1表示），则不做推荐，因为他们已经是好友。最后，要提供一个格式化输出，以便其他程序读取来进一步处理。

示例9-8：推荐连接：归约器

```
// key = person
// values = 可能的推荐好友（List<Tuple2<userID1, userID2>>）
reduce(key, values) {

    // mutualFriends.key 是推荐的好友
    // mutualFriends.value 是共同好友列表
    Map<Long, List> mutualFriends = new HashMap<Long, List>();

    for (Tuple2<toUser, mutualFriend> t2 : values) {
          Long toUser = t2.toUser; // t2._1
          Long mutualFriend = t2.mutualFriend; // t2._2
          boolean alreadyFriend = (mutualFriend == -1);

          if (mutualFriends.containsKey(toUser)) {
               if (alreadyFriend) {
                    mutualFriends.put(toUser, null);
               }
               else if (mutualFriends.get(toUser) != null) {
                    mutualFriends.get(toUser).add(mutualFriend);
               }
          }
          else {
               if (alreadyFriend) {
                    mutualFriends.put(toUser, null);
               }
               else {
                    mutualFriends.put(toUser, List<mutualFriend>)
               }
          }
    }

    String reducerOutput = buildOutput(mutualFriends);
    emit(key, reducerOutput);
}
```

buildOutput()方法的定义参见示例9-9。

示例9-9：推荐连接：buildOutput()方法

```
String buildOutput(Map<Long, List> map) {
   String output = "";
   for (Map.Entry<Long, List> entry : map.entrySet()) {
     String K = entry.getKey();
     List V = entry.getValue();
     output += K + " (" + V.size() + ": " + V + "),";
   }
   return output;
}
```

## Spark实现

我们的MapReduce解决方案实现使用了Spark-1.1.0 Java API。与MapReduce/Hadoop API

相比，由于Spark提供了一个更高层的Java API，所以我们可以把整个解决方案用一个Java驱动器类（SparkFriendRecommendation）表示，其中包括一系列flatMapToPair()和groupBy()函数。Spark的MapReduce抽象基于RDD。我们使用JavaRDD<T>表示一组类型为T的对象，另外使用JavaPairRDD<K,V>表示一组键-值对。

首先给出一个高层解决方案，其中包括6个基本步骤（参见示例9-10），然后我们会详细分析各个步骤，并提供相应的Spark代码。需要说明，其中一些步骤可以合并（以建立更为复杂的步骤），不过为简单起见，这里没有这么做。

示例9-10：SparkFriendRecommendation: 高层步骤

```
// 步骤1：导入必要的类和接口
public class SparkFriendRecommendation {
   public static void main(String[] args) throws Exception {
     // 步骤2：处理输入参数
     // 步骤3：创建一个Spark上下文对象（ctx）
     // 步骤4：读取HDFS输入文本并创建第一个RDD
     // 步骤5：实现map()函数
     // 步骤6：实现reduce()函数
     // 步骤7：生成所需的最终输出

     // 完成
     ctx.close();
     System.exit(0);
   }
}
```

示例9-11给出了Spark解决方案中使用的一些基本工具函数。

示例9-11：工具方法

```
   static String buildRecommendations(Map<Long, List<Long>> mutualFriends) {
      StringBuilder recommendations = new StringBuilder();
      for (Map.Entry<Long, List<Long>> entry : mutualFriends.entrySet()) {
         if (entry.getValue() == null) {
            // 已经是好友，不需要再次推荐！
            continue;
         }
         recommendations.append(entry.getKey());
         recommendations.append(" (");
         recommendations.append(entry.getValue().size());
         recommendations.append(":" );
         recommendations.append(entry.getValue());
         recommendations.append("),");
      }
      return recommendations.toString();
   }

   static Tuple2<Long,Long> T2(long a, long b) {
      return new Tuple2<Long,Long>(a, b);
   }

```

```
22    static Tuple2<Long,Tuple2<Long,Long>> T2(long a, Tuple2<Long,Long> b) {
23       return new Tuple2<Long,Tuple2<Long,Long>>(a, b);
24    }
```

## 步骤1：导入必要的类

这一步如示例9-12所示，将导入所需的类和接口。这里使用了Spark的两个重要的包：

`org.apache.spark.api.java`

这个包定义了RDD（JavaRDD，JavaPairRDD和JavaSpark-Context）。

`org.apache.spark.api.java.function`

这个包定义了转换函数（Function和PairFlatMapFunction）。

示例9-12：步骤1：导入必要的类

```
1 // 步骤1：导入必要的类和接口
2 import scala.Tuple2;
3 import org.apache.spark.api.java.JavaRDD;
4 import org.apache.spark.api.java.JavaPairRDD;
5 import org.apache.spark.api.java.JavaSparkContext;
6 import org.apache.spark.api.java.function.Function;
7 import org.apache.spark.api.java.function.PairFlatMapFunction;
8 import java.util.Arrays;
9 import java.util.List;
10 import java.util.ArrayList;
11 import java.util.Map;
12 import java.util.HashMap;
```

## 步骤2：处理输入参数

这一步如示例9-13所示，我们将读取一个输入参数：HTFS文本文件的输入路径，这个文本文件中包含用户和他们的好友。

示例9-13：步骤2：处理输入参数

```
1    // 步骤2：处理输入参数
2    if (args.length < 1) {
3      System.err.println("Usage: SparkFriendRecommendation <users-and-friends>");
4      System.exit(1);
5    }
6    String hdfsInputFile = args[0];
```

## 步骤3：创建一个Spark上下文对象

在示例9-14中，将创建一个`JavaSparkContextorg.apache.spark.api.java.`JavaSparkContext对象，这个对象是创建新RDD的一个工厂。创建JavaSparkContext对象有很多方法，有关的详细信息可以参考Spark的Java API。

示例9-14：步骤3：创建一个Spark上下文对象

```
1 // 步骤3：创建一个Spark上下文对象
2 JavaSparkContext ctx = new JavaSparkContext();
```

## 步骤4：读取HDFS输入文件并创建一个RDD

在示例9-15中，JavaSparkContext返回第一个JavaRDD<String>对象，这表示HDFS输入文件中的所有记录。这里使用JavaRDD.collect()进行调试，确保得到了正确的输入数据。

示例9-15：步骤4：读取HDFS输入文件并创建RDD

```
1 // 步骤4：读取HDFS文件并创建第一个RDD
2 JavaRDD<String> records = ctx.textFile(hdfsInputFile, 1);
```

可以使用以下代码调试步骤4：

```
// debug0
List<String> debug1 = records.collect();
for (String t : debug1) {
  System.out.println("debug1 record="+t);
}
```

## 步骤5：实现map()函数

这一步将实现MapReduce算法中的map()函数。映射器将各个记录：

```
<person><TAB><friend1><,><friend2><,><friend3><,>...
```

转换为一个键-值对集，其中键是<person>，值是一个Tuple2<friend1,friend2>，表示直接好友关系和将来可能的好友关系。映射器由JavaRDD.flatMapToPair()函数实现。我们提供了call()方法来完成这个实现，如下所示：

```
JavaPairRDD<K2,V2> flatMapToPair(PairFlatMapFunction<T,K2,V2> f)
// 首先对这个RDD中的所有元素应用一个函数，
// 返回一个新的RDD，然后扁平化结果

PairFlatMapFunction<T, K, V>
T => Iterable<Tuple2<K, V>>
```

在这里：

- T是一个输入参数/记录。
- 键-值对由T生成。

示例9-16给出了详细的映射器实现。

示例9-16：步骤5：实现map()函数

```
// 步骤5：实现map()函数
// flatMapToPair
// <K2,V2> JavaPairRDD<K2,V2> flatMapToPair(PairFlatMapFunction<T,K2,V2> f)
// 首先对这个RDD中的所有元素应用一个函数，
// 返回一个新的RDD，然后扁平化结果
//
// PairFlatMapFunction<T, K, V>
// T => Iterable<Tuple2<K, V>>
JavaPairRDD<Long, Tuple2<Long,Long>> pairs =
      records.flatMapToPair(new PairFlatMapFunction<
                                                    String,           // T
                                                    Long,             // K
                                                    Tuple2<Long,Long> // V
                                                   >() {
 public Iterable<Tuple2<Long,Tuple2<Long,Long>>> call(String record) {
    // record=<person><TAB><friend1><,><friend2><,><friend3><,>...
    String[] tokens = record.split("\t");
    long person = Long.parseLong(tokens[0]);
    String friendsAsString = tokens[1];
    String[] friendsTokenized = friendsAsString.split(",");

    List<Long> friends = new ArrayList<Long>();
    List<Tuple2<Long,Tuple2<Long, Long>>> mapperOutput =
        new ArrayList<Tuple2<Long,Tuple2<Long, Long>>>();
    for (String friendAsString : friendsTokenized) {
       long toUser = Long.parseLong(friendAsString);
       friends.add(toUser);
       Tuple2<Long,Long> directFriend = T2(toUser, -1L);
       mapperOutput.add(T2(person, directFriend));
    }

    for (int i = 0; i < friends.size(); i++) {
       for (int j = i + 1; j < friends.size(); j++) {
            // 可能的好友1
            Tuple2<Long,Long> possibleFriend1 = T2(friends.get(j), person);
            mapperOutput.add(T2(friends.get(i), possibleFriend1));
            // 可能的好友2
            Tuple2<Long,Long> possibleFriend2 = T2(friends.get(i), person);
            mapperOutput.add(T2(friends.get(j), possibleFriend2));
        }
     }
     return mapperOutput;
   }
 });
```

我们使用了以下代码来调试步骤5：

```
// debug2
List<Tuple2<Long,Tuple2<Long,Long>>> debug2 = pairs.collect();
for (Tuple2<Long,Tuple2<Long,Long>> t2 : debug2) {
  System.out.println("debug2 key="+t2._1+"\t value="+t2._2);
}
```

## 步骤6：实现reduce()函数

这里通过调用groupByKey()和collect()方法应用reduce()函数，如示例9-17所示。

示例9-17：步骤6：实现reduce()函数

```
// 步骤6：实现reduce()函数
JavaPairRDD<Long, Iterable<Tuple2<Long, Long>>> grouped = pairs.groupByKey();
```

我们使用了以下代码来调试这一步：

```
// debug3
List<Tuple2<Long, Iterable<Tuple2<Long, Long>>>> debug3 = grouped.collect();
for (Tuple2<Long, Iterable<Tuple2<Long, Long>>> t2 : debug3) {
  System.out.println("debug3 key="+t2._1+"\t value="+t2._2);
}
```

## 步骤7：生成最终输出

我们将聚集步骤6中分组的所有值，创建最终输出。这一步会生成最后推荐的将来好友。这里使用了JavaPairRDD.mapValues()方法：

```
public <U> JavaPairRDD<K,U> mapValues(Function<V,U> f)
// 描述：通过一个映射函数将各个值传入键-值对RDD，
// 而不改变键，
// 还会保留原来的RDD分区。
```

这一步将Iterable<Tuple2<Long, Long>>（作为输入）转换为一个字符串对象（作为输出）。实现细节参见示例9-18。

示例9-18：步骤7：生成所需的最终输出

```
// 步骤7：生成所需的最终输出
// 找出所有 List<List<Long>>的交集
// mapValues[U](f: (V) => U): JavaPairRDD[K, U]
// 通过一个映射函数将各个值传入键-值对RDD，
// 而不改变键，
// 这还会保留原来的RDD分区。
JavaPairRDD<Long, String> recommendations =
   grouped.mapValues(new Function<
        Iterable<Tuple2<Long, Long>>, // 输入
        String                        // 最终输出
                                >() {
     public String call(Iterable<Tuple2<Long, Long>> values) {
        // mutualFriends.key = 推荐的好友
        // mutualFriends.value = 共同好友列表
        final Map<Long, List<Long>> mutualFriends =
           new HashMap<Long, List<Long>>();
        for (Tuple2<Long, Long> t2 : values) {
           final Long toUser = t2._1;
           final Long mutualFriend = t2._2;
           final boolean alreadyFriend = (mutualFriend == -1);
```

```
21
22              if (mutualFriends.containsKey(toUser)) {
23                 if (alreadyFriend) {
24                     mutualFriends.put(toUser, null);
25                 }
26                 else if (mutualFriends.get(toUser) != null) {
27                     mutualFriends.get(toUser).add(mutualFriend);
28                 }
29              }
30              else {
31                 if (alreadyFriend) {
32                    mutualFriends.put(toUser, null);
33                 }
34                 else {
35                     List<Long> list1 =
36                        new ArrayList<Long>(Arrays.asList(mutualFriend));
37                     mutualFriends.put(toUser, list1);
38                 }
39              }
40           }
41           return buildRecommendations(mutualFriends);
42        }
43     });
```

我们使用了以下代码来调试步骤7：

```
// debug4
List<Tuple2<Long,String>> debug4 = recommendations.collect();
for (Tuple2<Long,String> t2 : debug4) {
  System.out.println("debug4 key="+t2._1+ "\t value="+t2._2);
}
```

### 合并步骤6和步骤7

步骤6和步骤7可以利用Spark的combineByKey()转换器合并为一个步骤。一般地，groupByKey()后如果紧跟着mapValues()，这可以替换为reduceByKey()或combineByKey()。

## Spark程序运行示例

### HDFS输入

```
# hadoop fs -cat /data/friendS2.txt
1 2,3,4,5,6,7,8
2 1,3,4,5,7
3 1,2
4 1,2,6
5 1,2
6 1,4
7 1,2
8 1
```

## 运行Spark程序的脚本

```
# cat run_friends_recommendationS2.sh
#!/bin/bash
export JAVA_HOME=/usr/java/jdk7
export SPARK_HOME=/home/hadoop/spark-1.1.0
export SPARK_MASTER=spark://myserver100:7077
BOOK_HOME=/mp/data-algorithms-book
APP_JAR=$BOOK_HOME/dist/data_algorithms_book.jar
USERS=/data/friendS2.txt
# 在Spark独立集群上运行
prog=org.dataalgorithms.chap09.spark.SparkFriendRecommendation
$SPARK_HOME/bin/spark-submit \
  --class $prog \
  --master $SPARK_MASTER \
  --executor-memory 2G \
  --total-executor-cores 20 \
  $APP_JAR $USERS
```

## 程序运行日志

因为版面所限，我们对日志做了删减和编辑：

```
# ./run_friends_recommendationS2.sh
...
14/06/02 12:46:51 INFO Remoting: Starting remoting
14/06/02 12:46:51 INFO Remoting: Remoting started; listening on addresses :
  [akka.tcp://spark@myserver100:38397]
14/06/02 12:46:51 INFO Remoting: Remoting now listens on addresses:
  [akka.tcp://spark@myserver100:38397]
...
14/06/02 12:46:51 INFO server.AbstractConnector:
  Started SocketConnector@0.0.0.0:43523
14/06/02 12:46:51 INFO broadcast.HttpBroadcast:
  Broadcast server started at http://myserver100:43523
...
14/06/02 12:46:52 INFO client.AppClient$ClientActor:
  Connecting to master spark://myserver100:7077...
...
14/06/02 12:46:57 INFO scheduler.DAGScheduler: Stage 0
  (collect at SparkFriendRecommendation.java:34) finished in 3.446 s
14/06/02 12:46:57 INFO spark.SparkContext: Job finished:
  collect at SparkFriendRecommendation.java:34, took 3.543265173 s
debug1 record=1 2,3,4,5,6,7,8
debug1 record=2 1,3,4,5,7
debug1 record=3 1,2
debug1 record=4 1,2,6
debug1 record=5 1,2
debug1 record=6 1,4
debug1 record=7 1,2
debug1 record=8 1
14/06/02 12:46:57 INFO spark.SparkContext: Starting job:
  collect at SparkFriendRecommendation.java:79
...
```

```
14/06/02 12:46:58 INFO scheduler.DAGScheduler: Stage 1
  (collect at SparkFriendRecommendation.java:79) finished in 1.643 s
14/06/02 12:46:58 INFO spark.SparkContext: Job finished:
  collect at SparkFriendRecommendation.java:79, took 1.654938027 s
debug2 key=1 value=(2,-1)
debug2 key=1 value=(3,-1)
debug2 key=1 value=(4,-1)
debug2 key=1 value=(5,-1)
debug2 key=1 value=(6,-1)
debug2 key=1 value=(7,-1)
debug2 key=1 value=(8,-1)
debug2 key=2 value=(3,1)
debug2 key=3 value=(2,1)
debug2 key=2 value=(4,1)
debug2 key=4 value=(2,1)
debug2 key=2 value=(5,1)
debug2 key=5 value=(2,1)
debug2 key=2 value=(6,1)
debug2 key=6 value=(2,1)
debug2 key=2 value=(7,1)
debug2 key=7 value=(2,1)
debug2 key=2 value=(8,1)
debug2 key=8 value=(2,1)
debug2 key=3 value=(4,1)
debug2 key=4 value=(3,1)
debug2 key=3 value=(5,1)
debug2 key=5 value=(3,1)
debug2 key=3 value=(6,1)
debug2 key=6 value=(3,1)
debug2 key=3 value=(7,1)
debug2 key=7 value=(3,1)
debug2 key=3 value=(8,1)
debug2 key=8 value=(3,1)
debug2 key=4 value=(5,1)
debug2 key=5 value=(4,1)
debug2 key=4 value=(6,1)
debug2 key=6 value=(4,1)
debug2 key=4 value=(7,1)
debug2 key=7 value=(4,1)
debug2 key=4 value=(8,1)
debug2 key=8 value=(4,1)
debug2 key=5 value=(6,1)
debug2 key=6 value=(5,1)
debug2 key=5 value=(7,1)
debug2 key=7 value=(5,1)
debug2 key=5 value=(8,1)
debug2 key=8 value=(5,1)
debug2 key=6 value=(7,1)
debug2 key=7 value=(6,1)
debug2 key=6 value=(8,1)
debug2 key=8 value=(6,1)
debug2 key=7 value=(8,1)
debug2 key=8 value=(7,1)
debug2 key=2 value=(1,-1)
debug2 key=2 value=(3,-1)
```

```
debug2 key=2 value=(4,-1)
debug2 key=2 value=(5,-1)
debug2 key=2 value=(7,-1)
debug2 key=1 value=(3,2)
debug2 key=3 value=(1,2)
debug2 key=1 value=(4,2)
debug2 key=4 value=(1,2)
debug2 key=1 value=(5,2)
debug2 key=5 value=(1,2)
debug2 key=1 value=(7,2)
debug2 key=7 value=(1,2)
debug2 key=3 value=(4,2)
debug2 key=4 value=(3,2)
debug2 key=3 value=(5,2)
debug2 key=5 value=(3,2)
debug2 key=3 value=(7,2)
debug2 key=7 value=(3,2)
debug2 key=4 value=(5,2)
debug2 key=5 value=(4,2)
debug2 key=4 value=(7,2)
debug2 key=7 value=(4,2)
debug2 key=5 value=(7,2)
debug2 key=7 value=(5,2)
debug2 key=3 value=(1,-1)
debug2 key=3 value=(2,-1)
debug2 key=1 value=(2,3)
debug2 key=2 value=(1,3)
debug2 key=4 value=(1,-1)
debug2 key=4 value=(2,-1)
debug2 key=4 value=(6,-1)
debug2 key=1 value=(2,4)
debug2 key=2 value=(1,4)
debug2 key=1 value=(6,4)
debug2 key=6 value=(1,4)
debug2 key=2 value=(6,4)
debug2 key=6 value=(2,4)
debug2 key=5 value=(1,-1)
debug2 key=5 value=(2,-1)
debug2 key=1 value=(2,5)
debug2 key=2 value=(1,5)
debug2 key=6 value=(1,-1)
debug2 key=6 value=(4,-1)
debug2 key=1 value=(4,6)
debug2 key=4 value=(1,6)
debug2 key=7 value=(1,-1)
debug2 key=7 value=(2,-1)
debug2 key=1 value=(2,7)
debug2 key=2 value=(1,7)
debug2 key=8 value=(1,-1)
14/06/02 12:46:58 INFO spark.SparkContext: Starting job:
  collect at SparkFriendRecommendation.java:88
14/06/02 12:46:58 INFO scheduler.DAGScheduler:
  Registering RDD 2 (flatMapToPair at SparkFriendRecommendation.java:42)
...
14/06/02 12:46:59 INFO scheduler.DAGScheduler: Stage 2
```

```
  (collect at SparkFriendRecommendation.java:88) finished in 0.319 s
14/06/02 12:46:59 INFO spark.SparkContext: Job finished:
  collect at SparkFriendRecommendation.java:88, took 0.408234901 s
debug3 key=4 value=[(2,1), (3,1), (5,1), (6,1), (7,1), (8,1),
                    (1,2), (3,2), (5,2), (7,2), (1,-1), (2,-1),
                    (6,-1), (1,6)]
debug3 key=2 value=[(3,1), (4,1), (5,1), (6,1), (7,1), (8,1),
                    (1,-1), (3,-1), (4,-1), (5,-1), (7,-1),
                    (1,3), (1,4), (6,4), (1,5), (1,7)]
debug3 key=6 value=[(2,1), (3,1), (4,1), (5,1), (7,1), (8,1),
                    (1,4), (2,4), (1,-1), (4,-1)]
debug3 key=8 value=[(2,1), (3,1), (4,1), (5,1), (6,1), (7,1), (1,-1)]
debug3 key=3 value=[(2,1), (4,1), (5,1), (6,1), (7,1), (8,1), (1,2),
                    (4,2), (5,2), (7,2), (1,-1), (2,-1)]
debug3 key=1 value=[(2,-1), (3,-1), (4,-1), (5,-1), (6,-1), (7,-1),
                    (8,-1), (3,2), (4,2), (5,2), (7,2), (2,3), (2,4),
                    (6,4), (2,5), (4,6), (2,7)]
debug3 key=7 value=[(2,1), (3,1), (4,1), (5,1), (6,1), (8,1), (1,2),
                    (3,2), (4,2), (5,2), (1,-1), (2,-1)]
debug3 key=5 value=[(2,1), (3,1), (4,1), (6,1), (7,1), (8,1), (1,2),
                    (3,2), (4,2), (7,2), (1,-1), (2,-1)]
14/06/02 12:46:59 INFO spark.SparkContext: Starting job:
  collect at SparkFriendRecommendation.java:135
14/06/02 12:46:59 INFO scheduler.DAGScheduler: Got job 3 (collect at
  SparkFriendRecommendation.java:135) with 1 output partitions (allowLocal=false)
...
14/06/02 12:46:59 INFO scheduler.DAGScheduler: Stage 4 (collect at
  SparkFriendRecommendation.java:135) finished in 0.109 s
14/06/02 12:46:59 INFO spark.SparkContext: Job finished: collect
  at SparkFriendRecommendation.java:135, took 0.124233834 s
debug4 key=4 value=3 (2: [1, 2]),5 (2: [1, 2]),7 (2: [1, 2]),8 (1: [1]),
debug4 key=2 value=6 (2: [1, 4]),8 (1: [1]),
debug4 key=6 value=2 (2: [1, 4]),3 (1: [1]),5 (1: [1]),7 (1: [1]),
                     8 (1: [1]),
debug4 key=8 value=2 (1: [1]),3 (1: [1]),4 (1: [1]),5 (1: [1]),
                     6 (1: [1]),7 (1: [1]),
debug4 key=3 value=4 (2: [1, 2]),5 (2: [1, 2]),6 (1: [1]),
                     7 (2: [1, 2]),8 (1: [1]),
debug4 key=1 value=
debug4 key=7 value=3 (2: [1, 2]),4 (2: [1, 2]),5 (2: [1, 2]),
                     6 (1: [1]),8 (1: [1]),
debug4 key=5 value=3 (2: [1, 2]),4 (2: [1, 2]),6 (1: [1]),
                     7 (2: [1, 2]),8 (1: [1]),
```

这一章介绍了推荐引擎，这是数据挖掘中的一个很重要的概念。这里通过一些简单的例子讲述了如何使用MapReduce范式实现一个基本的推荐引擎。下一章将通过电影推荐用例继续讨论推荐引擎。

# 第10章

# 基于内容的电影推荐

你是不是很想知道Netflix是如何为用户创建推荐电影的？或者Amazon如何为用户推荐图书？肯定有某种魔法算法生成这些推荐，是吗？Netflix甚至斥资100万美元奖励找出最佳的电影推荐解决方案[20]。基于内容的推荐系统（如Netflix和Amazon使用的推荐系统）会检查项目（如电影）的属性来为用户做出推荐。例如，如果一个用户看了很多动作片，推荐系统就会为他推荐这一类的电影。

这一章将提供一个基本的基于内容的MapReduce推荐解决方案，这个解决方案以Edwin Chen的博客[6]为基础。假设你在运营一个在线电影企业，希望生成电影推荐。现在有一个评分系统（人们可以对电影打分，评级为1到5星），另外假设为简单起见，所有评分都存储在HDFS中的一个TSV（tab-separated value，tab分隔值）文件中。我会提供一个通用的MapReduce解决方案，然后给出一个具体的电影推荐Spark实现。

需要说明，在基于内容的推荐系统中，我们得到的关于内容的信息（如领域和元数据）越多，算法就会变得越复杂（因为要涉及更多的变量），不过推荐也会更准确、更合理。例如，要实现电影推荐，系统应当有一些元数据，如演员、导演和制片人。在这里的例子中，我们的算法仅限于电影评分。一般地，在一个推荐系统应用中，会有两类实体：用户和项目。在我们的例子中，用户是看电影和给电影评分的人，项目是电影。下一节中的输入数据会展示用户和电影之间的关系。

这一章将提供一个3阶段MapReduce解决方案来实现电影推荐：

- 阶段1：找出各个电影的评分人总数。
- 阶段2：对于每个电影对A和B，找出所有同时对A和B评分的人。
- 阶段3：找出每两个相关电影之间的关联。在这个阶段，我会展示如何应用3个不同的

关联度算法（Pearson、Cosine和Jaccard）。一般地，要根据具体的数据需求来选择关联度算法，不过，这个阶段可以应用你想要的任何关联度算法。

# 输入

假设原始输入数据采用以下格式（我们将它称为格式1）：

```
<user> <movie> <rating>
```

这里user是对movie评分的那个人的ID，rating是一个从1到5的整数。表10-1给出了一个例子：

**表10-1：用户和电影**

| 用户 | 电影 | 评分 |
|---|---|---|
| User1 | Movie1 | 1 |
| User1 | Movie2 | 2 |
| User1 | Movie3 | 3 |
| User2 | Movie1 | 1 |
| User2 | Movie2 | 2 |
| User2 | Movie3 | 3 |
| User2 | Movie5 | 5 |
| ... | ... | ... |

使用格式1，我们将生成另一个输入，它有以下格式（称为格式2），这里增加了一个新列numberOfRaters，表示对这个特定电影评分的人数，而且我们会按movie分组：

```
<user> <movie> <rating> <numberOfRaters>
```

# MapReduce阶段1

这个阶段的目标是读取格式1的数据，生成格式2的数据。为了找出numberOfRaters，我们写了一个简单的MapReduce作业，其中包括一个映射器和一个归约器。给定一个输入记录<user> <movie> <rating>，映射器发出一个（$K_2$, $V_2$）对，其中$K_2$是<movie>，$V_2$是一个Tuple2(user, rating)。归约器确定一个电影被评分的次数，然后生成我们需要的输出（如前定义）。

阶段1的映射器如示例10-1所示。

示例10-1：映射器：阶段1

```
1 map( <user> <movie> <rating>) {
2    K2 = <movie>;
3    V2 = Tuple2(user, rating);
4    emit(K2, V2);
5 }
```

阶段1的归约器如示例10-2所示。

示例10-2：归约器：阶段1

```
1  //key = <movie>
2  //values = List<Tuple2<user,rating>>
3 reduce(key, values) {
4    numberOfRaters = values.size();
5    for (Tuple2<user,rating> t2 : values) {
6       K3 = t2.user;
7       V3 = Tuple3(key, t2.rating, numberOfRaters);
8       emit(K3, V3);
9    }
10 }
```

阶段1的输出将用作为阶段2的输入。

# MapReduce阶段2和阶段3

我们的MapReduce解决方案将计算两个电影的相似度，如果有人看过电影《狮子王》，我们就会向他推荐《玩具总动员》之类的电影。那么如何定义两个电影的相似度呢？一种方法是使用它们的关联度[32]：

1. 对于每一对电影A和B，找出同时对A和B评分的所有人。
2. 用这些评分来建立一个电影A向量和一个电影B向量。
3. 计算这两个向量之间的关联度（两项或多项的相互关系：在这里，关联度是一种度量两个电影关联或相关程度的方法）。
4. 只要有人看过一个电影，我们就可以推荐与它关联度最高的电影。

前两个步骤将在MapReduce阶段2中完成（在这个阶段中，我们将生成一个电影A向量和一个电影B向量）。后两个步骤在MapReduce阶段3中完成（计算每一对相关电影之间的关联度）。

要得到所有相关电影对，我们将让输入（所有电影评分）与自身连接（类似SQL中的数据库表与自身连接）。为了让评分输入与自身连接，映射器要收集各个用户的所有电影，另外归约器将生成一个用户评分的所有电影的唯一组合。

为了更好地理解这一点，下面为两个用户创建一个示例输入（注意，这个输入在阶段1生成，每个记录包含<user>、<movie>、<rating>和<numberOfRaters>），如表10-2所示。

**表10-2：用户和电影：示例输入**

| 用户 | 电影 | 评分 | 评分人数 |
|---|---|---|---|
| User1 | Movie1 | 1 | 10 |
| User1 | Movie2 | 2 | 20 |
| User1 | Movie3 | 3 | 30 |
| User2 | Movie1 | 1 | 10 |
| User2 | Movie2 | 2 | 20 |
| User2 | Movie3 | 3 | 30 |
| User2 | Movie5 | 5 | 50 |
| ... | ... | ... | ... |

## MapReduce阶段2：映射器

map()函数接受一个输入行：

```
<user> <movie> <rating> <numberOfRaters>
```

发出一个键-值对：

```
    key: <user>
  value: <movie> <rating> <numberOfRaters>
```

示例10-3定义了这个map()函数。

示例10-3：映射器：阶段2

```
// key = <user>
// value = <movie> <rating> <NumOfRaters>
map(key, value) {
   String[] tokens = value.split("\t");
   movie = tokens[0];
   rating = tokens[1];
   numberOfRaters = tokens[2];
   Tuple3<String, Integer, Integer> t3 =
       new Tuple3(movie, rating, numberOfRaters);
   emit(key, t3);
}
```

对于我们的示例输入，映射器将生成表10-3所示的输出（将由归约器处理）。

表10-3：映射器输出

| 键 | 值 |
|---|---|
| User1 | <Movie1, 1, 10> |
| User1 | <Movie2, 2, 20> |
| User1 | <Movie3, 3, 30> |
| User2 | <Movie1, 1, 10> |
| User2 | <Movie2, 2, 20> |
| User2 | <Movie3, 3, 30> |
| User2 | <Movie5, 5, 50> |
| ... | ... |

## MapReduce阶段2：归约器

reduce()函数接受一个用户（作为键）和一个Tuple3(movie, rating, numOfRaters)列表（作为值），发出各个用户评分的所有电影的唯一组合。因此，归约器的输入如表10-4所示。

表10-4：排序和洗牌的输出

| 键 | 值 |
|---|---|
| User1 | [<Movie1, 1, 10>,<br><Movie2, 2, 20>,<br><Movie3, 3, 30>] |
| User2 | [<Movie1, 1, 10>,<br><Movie2, 2, 20>,<br><Movie3, 3, 30>,<br><Movie5, 5, 50>] |
| ... | ... |

示例10-4定义了reduce()函数，它将“排序和洗牌”阶段的输出与自身连接（在“user”键上完成这个连接）。

示例10-4：归约器：阶段2

```
1  // key = user（由map()生成）
2  // value = List{ Tuple3(<movie> <rating> <numOfRaters>) }
3 reduce(key, value) {
4      // 生成所有电影的唯一组合。
5      // 确保不要创建重复的项，如（movie1, movie2） 和
6      // (movie2, movie1)。
7      // 为了避免这种情况，我们将创建（movie1, movie2）
8      // 这里 movie1 < movie2
```

```
    List[ Tuple2( Tuple3(<movie1> <rating1> <numOfRaterS1>),
          Tuple3(<movie2> <rating2> <numOfRaterS2>) ]
          list = generateUniqueCombinations(value);

    for (Tuple2( Tuple3(<movie1> <rating1> <numOfRaterS1>),
                 Tuple3(<movie2> <rating2> <numOfRaterS2>) pair : list) {
        m1 = pair._1; // = Tuple3(<movie1> <rating1> <numOfRaterS1>)
        m2 = pair._2; // = Tuple3(<movie2> <rating2> <numOfRaterS2>)

        // 定义归约器键
       reducerKey = Tuple2(m1.movie, m2.movie);

       // 计算额外信息,
       // 相关函数将使用这些信息
       int ratingProduct = m1.rating * m2.rating;
       int rating1Squared = m1.rating * m1.rating;
       int rating2Squared = m2.rating * m2.rating;

        // 定义归约器值
       reducerValue = Tuple7(m1.rating,
                             m1.NumOfRaters,
                             m2.rating,
                             m2.NumOfRaters,
                             ratingProduct,
                             rating1Squared,
                             rating2Squared);
       emit(reducerKey, reducerValue);
    } // for循环结束
}
```

归约器将生成表10-5和表10-6所示的键-值对。

**表10-5：由key=User1生成的归约器输出**

| 键 | 值 |
|---|---|
| <Movie1, Movie2> | <1 10 2 20 2 1 4> |
| <Movie1, Movie3> | <1 10 3 30 3 1 9> |
| <Movie2, Movie3> | <2 20 3 30 6 4 9> |

**表10-6：由key=User2生成的归约器输出**

| 键 | 值 |
|---|---|
| <Movie1, Movie2> | <2 10 3 20 6 4 9> |
| <Movie1, Movie3> | <2 10 4 30 8 4 16> |
| <Movie1, Movie5> | <2 10 5 50 10 4 25> |
| <Movie2, Movie3> | <3 20 4 30 12 9 16> |
| <Movie2, Movie5> | <3 20 5 50 15 9 25> |
| <Movie3, Movie5> | <4 30 5 50 20 16 25> |

## MapReduce阶段3：映射器

阶段3映射器是一个恒等映射器。map()函数接受键-值输入对，如下所示：

```
    key: Tuple2(<movie1>, <movie2>)
  value: Tuple7(<rating1>,
                <numOfRaterS1>,
                <rating2>,
                <numOfRaterS2>,
                <ratingProduct>,
                <rating1Squared>,
                <rating2Squared>
                )
```

这个映射器只是发出接收到的键-值对。map()函数的定义参见示例10-5。

示例10-5：映射器：阶段3

```
1 // key = Tuple2(<movie1>, <movie2>)由阶段2的归约器生成
2 // value = Tuple7(<rating1> <numOfRaterS1> <rating2> <numOfRaterS2>
3 // <ratingProduct> <rating1Squared> <rating2Squared>)
4 map(key, value) {
5    emit(key, value);
6 }
```

这个映射器将由以上示例输入生成表10-7所示的输出（这个输出将用作为阶段3归约器的输入）。

**表10-7：映射器输出将用作为阶段3归约器的输入**

| 键 | 值 |
|---|---|
| <Movie1, Movie2> | <1 10 2 20 2 1 4> |
| <Movie1, Movie3> | <1 10 3 30 3 1 9> |
| <Movie2, Movie3> | <2 20 3 30 6 4 9> |
| <Movie1, Movie2> | <2 10 3 20 6 4 9> |
| <Movie1, Movie3> | <2 10 4 30 8 4 16> |
| <Movie1, Movie5> | <2 10 5 50 10 4 25> |
| <Movie2, Movie3> | <3 20 4 30 12 9 16> |
| <Movie2, Movie5> | <3 20 5 50 15 9 25> |
| <Movie3, Movie5> | <4 30 5 50 20 16 25> |

## MapReduce阶段3：归约器

reduce()函数接受（<movie1>, <movie2>）作为键，另外接受以下列表作为值：

```
Tuple7(<rating1>
       <numOfRaterS1>,
```

<*rating*$_2$>
<*numOfRaterS*$_2$>
<*ratingProduct*>
<*rating*$_1$*Squared*>
<*rating*$_2$*Squared*>)

这个函数将发出每两个相关电影的关联度。所以，归约器的输入为表10-8所示的键-值对。

表10-8：阶段3的归约器输入

| Key | Value |
|---|---|
| <Movie1, Movie2> | [<1 10 2 20 2 1 4>, <2 10 3 20 6 4 9>] |
| <Movie1, Movie3> | [<1 10 3 30 3 1 9>, <2 10 4 30 8 4 16>] |
| <Movie2, Movie3> | [<2 20 3 30 6 4 9>, <3 20 4 30 12 9 16>] |
| <Movie1, Movie5> | <2 10 5 50 10 4 25> |
| <Movie2, Movie5> | <3 20 5 50 15 9 25> |
| <Movie3, Movie5> | <4 30 5 50 20 16 25> |

示例10-6给出了reduce()的定义。

示例10-6：归约器：阶段3

```
// key = Tuple2(<movie1, movie2>)
// values = List { Tuple7(<rating1>,
//                        <numOfRaterS1>,
//                        <rating2>,
//                        <numOfRaterS2>,
//                        <ratingProduct>,
//                        <rating1Squared>,
//                        <rating2Squared>)
// }
reduce(key, value) {
     // 计算额外信息
     // 相关函数将使用这些信息
    int groupSize = value.size(); // 各个向量的长度
    int dotProduct = 0; // ratingProd之和
    int rating1Sum = 0; // rating1之和
    int rating2Sum = 0; // rating2之和
    int rating1NormSq = 0; // rating1Squared之和
    int rating2NormSq = 0; // rating2Squared之和
    int maxNumOfumRaterS1 = 0; // numOfRaterS1最大值
    int maxNumOfumRaterS2 = 0; // numOfRaterS2最大值
    for (Tuple7(<rating1>
                <numOfRaterS1>
                <rating2>
```

```
                <numOfRaterS2>
                <ratingProduct>
                <rating1Squared>
                <rating2Squared>) : values) {
        dotProduct += ratingProd;
        rating1Sum += rating1;
        rating2Sum += rating2;
        rating1NormSq += rating1Squared;
        rating2NormSq += rating2Squared;
        if (numOfRaterS1 > maxNumOfumRaterS1) {
           maxNumOfumRaterS1 = numOfRaterS1;
        }
        if (numOfRaterS2 > maxNumOfumRaterS2) {
           maxNumOfumRaterS2 = numOfRaterS2;
        }
     }

     double pearson = calculatePearsonCorrelation(
                groupSize,
                dotProduct,
                rating1Sum,
                rating2Sum,
                rating1NormSq,
                rating2NormSq);

    double cosine = calculateCosineCorrelation(dotProduct,
                                               Math.sqrt(rating1NormSq),
                                               Math.sqrt(rating2NormSq));
    double jaccard = calculateJaccardCorrelation(groupSize,
                                                 maxNumOfumRaterS1,
                                                 maxNumOfumRaterS2);
    return Tuple3(pearson, cosine, jaccard);
}
```

最后，归约器将生成每两个电影之间的相似度：

```
<Movie1, Movie2> pearson1, cosine1, jaccard1
<Movie1, Movie3> pearson2, cosine2, jaccard2
<Movie2, Movie3> pearson3, cosine3, jaccard3
<Movie1, Movie2> pearson4, cosine4, jaccard4
<Movie1, Movie3> pearson5, cosine5, jaccard5
<Movie1, Movie5> pearson6, cosine6, jaccard6
<Movie2, Movie3> pearson7, cosine7, jaccard7
<Movie2, Movie5> pearson8, cosine8, jaccard8
<Movie3, Movie5> pearson9, cosine9, jaccard9
```

## 相似度度量

对于前面的关联度计算，我们使用了皮尔逊（Pearson）积矩相关系数、杰卡德（Jaccard）相似系数和余弦相似系数。皮尔逊关联度可以定义如下：

$$\text{correlation}(X,Y)=\frac{n\Sigma xy-\Sigma x\Sigma y}{\sqrt{n\Sigma x^2-(\Sigma x)^2}\sqrt{n\Sigma y^2-(\Sigma y)^2}}$$

杰卡德相似度和余弦相似度是另外两个常用的基于向量的相似度度量。当然，还有很多其他的度量方法可以用来计算关联度。其他关联度或相似度度量包括：

- 正则关联度。
- 欧氏距离。
- 曼哈顿距离。

要根据具体的领域/问题选择一个适当的关联度或相似度度量算法，需要仔细研究来确定哪个算法可以提供更好（实际最优）的结果。

# Spark电影推荐实现

这一节将提供本章前面介绍的MapReduce算法3个阶段的Spark实现。整个Spark解决方案用一个Java驱动器类实现，名为`MovieRecommendationsWithJoin`。对于这个Spark实现，我会提供以下内容：

- 所有高层步骤。
- 各个步骤的详细描述，并提供相应的完整Spark代码（Java代码）。
- 使用Spark-1.1.0的一个运行示例。

## Spark高层解决方案

这个Spark解决方案包括13个步骤，其中使用了RDD。应该知道，Spark的主要数据抽象和编程模型都基于RDD。下面回顾一下RDD的概念，RDD是有以下属性的不可变数据结构：

- 由HDFS文件或“并行化”数组创建。
- 可以用`map()`和`filter()`转换。
- 在并行操作（如`reduce()`、`collect()`和`foreach()`）之间可以缓存。

驱动器类`MovieRecommendationsWithJoin`如示例10-7所示。

*示例10-7：高层解决方案：所有步骤*

```
1  //步骤1: 导入所需的类和接口
2 public class MovieRecommendationsWithJoin {
3
4    public static void main(String[] args) throws Exception {
```

```
5        //步骤2: 处理输入参数
6        //步骤3: 创建一个Spark上下文对象
7        //步骤4: 读取HDFS文件并创建第一个RDD
8        //步骤5: 找出谁曾对这个电影评分
9        //步骤6: 按movie对moviesRDD分组
10       //步骤7: 得出每个电影的评分人数，然后创建（K, V）为
11       //       usersRDD = <K=user, V=<movie,rating,numberOfRaters>>
12       //步骤8: 将usersRDD与自身连接，找出所有(movie1, movie2)对
13       //       joinedRDD = usersRDD.join(usersRDD);
14       //       joinedRDD = (user, T2((m1,r1,n1), (m2,r2,n2)))
15       //步骤9: 删除重复的（movie1, movie2）对。
16       //       需要说明（movie1, movie2）和（movie2, movie1）是一样的
17       //步骤10: 生成所有（movie1, movie2）组合。
18       // 这一步的目标是创建(K, V)对
19       //    K: Tuple2(movie1, movie2)
20       //    V: Tuple7(movie1.rating,
21       //              movie1.numOfRaters,
22       //              movie2.rating,
23       //              movie2.numOfRaters,
24       //              ratingProduct,
25       //              rating1Squared, = movie1.rating * movie1.rating
26       //              rating2Squared = movie2.rating * movie2.rating
27       //              )
28       //步骤11: 按键(movie1,movie2)对moviePairs分组
29       //步骤12: 计算每个(movie1,movie2)对的关联度/相似度
30       //步骤13: 打印最终结果
31     System.exit(0);
32   }
33
34   static Tuple3<Double,Double,Double> calculateCorrelations(...);
35   static double calculatePearsonCorrelation(...);
36   static double calculateCosineCorrelation(...);
37   static double calculateJaccardCorrelation(...);
38 }
```

## 步骤1: 导入所需的类

我们需要的大部分类和接口都包含在以下两个包中：org.apache.spark.api.java和org.apache.spark.api.java.function。示例10-8展示了如何导入这些类和接口。

示例10-8：步骤1: 导入所需的类

```
1 //步骤1: 导入所需的类和接口
2 import scala.Tuple2;
3 import java.util.List;
4 import java.util.ArrayList;
5 import org.apache.spark.api.java.JavaRDD;
6 import org.apache.spark.api.java.JavaPairRDD;
7 import org.apache.spark.api.java.JavaSparkContext;
8 import org.apache.spark.api.java.function.Function;
9 import org.apache.spark.api.java.function.PairFlatMapFunction;
10 import org.apache.spark.api.java.function.FlatMapFunction;
11 import org.apache.spark.api.java.function.PairFunction;
```

## 步骤2：处理输入参数

运行Spark程序需要一个参数：HDFS输入文件（表示用户和评分）。示例10-9展示了如何处理这个参数。

示例10-9：步骤2: 处理输入参数

```
1    //步骤2: 处理输入参数
2   if (args.length < 1) {
3     System.err.println("Usage: MovieRecommendationsWithJoin <users-ratings>");
4     System.exit(1);
5   }
6   String usersRatingsInputFile = args[0];
7   System.out.println("usersRatingsInputFile="+ usersRatingsInputFile);
```

## 步骤3：创建一个Spark上下文对象

这一步如示例10-10所示，这里创建了一个JavaSparkContext对象，它会返回JavaRDD并处理Java集合。JavaSparkContext是一个工厂类，将创建JavaRDD<T>和JavaPairRDD<K,V>对象。

示例10-10：步骤3: 创建一个Spark上下文对象

```
1    //步骤3: 创建一个Spark上下文对象
2    JavaSparkContext ctx = new JavaSparkContext();
```

## 步骤4：读取输入文件并创建RDD

在这一步（参见示例10-11），将使用JavaSparkContext对象读取一个HDFS输入文本，并创建第一个JavaRDD<String>对象（String类型的记录集），这表示输入文件。

示例10-11：步骤4: 读取输入文件并创建RDD

```
1    //步骤4: 读取HDFS文件并创建第一个RDD
2    JavaRDD<String> usersRatings = ctx.textFile(usersRatingsInputFile, 1);
```

执行步骤4之后，usersRating RDD包含以下列表：

```
{
  "user1 movie1 1",
  "user1 movie2 2",
  "user1 movie3 3",
  "user2 movie1 1",
  "user2 movie2 2"
  "user2 movie3 3"
  "user2 movie5 5"
  ...
}
```

## 步骤5：找出谁曾对这个电影评分

usersRating RDD是步骤5的输入。步骤5～7会找出每个电影的评分人数。步骤5实现了一个映射器，它接受一个记录（由user、movie和rating组成），然后创建一个键-值对（movie, user,rating）。JavaRDD.mapToPair()接受一个输入记录，创建一个JavaPairRDD<K2,V2>，其中K2=movie，V2=Tuple2(user,rating)。如示例10-12所示。

示例10-12：步骤5: 找出谁曾对这个电影评分

```
// 步骤5: 找出谁曾对这个电影评分
// <K2,V2> JavaPairRDD<K2,V2> mapToPair(PairFunction<T,K2,V2> f)
// 通过对这个RDD的所有元素应用一个函数返回一个新的RDD
// PairFunction<T, K2, V2>
//    T => Tuple2<K2, V2>
//    T = <user> <movie> <rating>
//    K2 = <movie>
//    V2 = Tuple2<user, rating>
JavaPairRDD<String,Tuple2<String,Integer>> moviesRDD =
    usersRatings.mapToPair(
        new PairFunction<
                        String,                // T
                        String,                // K
                        Tuple2<String,Integer> // V
                       >() {
    public Tuple2<String,Tuple2<String,Integer>> call(String s) {
       String[] record = s.split("\t");
       String user = record[0];
       String movie = record[1];
       Integer rating = new Integer(record[2]);
       Tuple2<String,Integer> userAndRating =
           new Tuple2<String,Integer>(user, rating);
           return new Tuple2<String,Tuple2<String,Integer>>
               (movie, userAndRating);
    }
 });
```

我们使用了JavaPairRDD.collect()方法来调试步骤5：

```
System.out.println("=== debug1: moviesRDD: K = <movie>, " +
                   "= Tuple2<user, rating> ===");
List<
     Tuple2<String,Tuple2<String,Integer>>
    >  debug1 = moviesRDD.collect();
for (Tuple2<String,Tuple2<String,Integer>> t2 : debug1) {
  System.out.println("debug1 key="+t2._1 + "\t value="+t2._2);
}
```

执行步骤5之后，moviesRDD是一个键-值对列表，如表10-9所示。

表10-9：moviesRDD键-值对

| K2: Movie | V2: Tuple2(user, rating) |
|---|---|
| movie1 | (user1, 1) |
| movie2 | (user1, 2) |
| movie3 | (user1, 3) |
| movie1 | (user2, 1) |
| movie2 | (user2, 2) |
| movie3 | (user2, 3) |
| movie5 | (user2, 5) |
| ... | ... |

## 步骤6：按movie对moviesRDD分组

步骤5创建了一个JavaPairRDD<K2,V2>，其中K2=movie，V2=Tuple2(user,rating)。这一步按键对moviesRDD分组。其结果是：

```
JavaPairRDD<K,V>
   where K=movie
         V=List(Tuple2(user,rating))
```

这一步由JavaPairRDD.groupByKey()方法实现，如示例10-13所示。

示例10-13：步骤6: 按movie对moviesRDD分组

```
1    // 步骤6: 按movie对moviesRDD分组
2    JavaPairRDD<String, Iterable<Tuple2<String,Integer>>> moviesGrouped =
3       moviesRDD.groupByKey();
4
5    System.out.println("=== debug2: moviesGrouped: K = <movie>, " +
6                       "V = Iterable<Tuple2<user, rating>> ===");
7    List<Tuple2<String,Iterable<Tuple2<String,Integer>>> debug2 =
8       moviesGrouped.collect();
8    for (Tuple2<String,Iterable<Tuple2<String,Integer>>> t2 : debug2) {
9       System.out.println("debug2 key="+t2._1 + "\t value="+t2._2);
10   }
```

执行步骤6之后，moviesGrouped是以下键-值对列表。显然，现在可以统计各个电影的评分人数（就是每个电影的列表值的大小）：

```
{
  (movie1, [(user1, 1), (user2, 1), ...]),
  (movie2, [(user1, 2), (user2, 2), ...]),
  (movie3, [(user1, 3), (user2, 3), ...]),
  (movie5, [(user2, 5), ...])
  ...
}
```

## 步骤7：找出每个电影的评分人数

这一步如示例10-14所示，将统计各个电影的评分人数，并生成另一个JavaPairRDD<K,V>，其中K=user，V=Tuple3(movie, rating, numberOfRaters)。这一步由JavaPairRDD.flatMapToPair()方法实现。

示例10-14：步骤7：找出每个电影的评分人数

```
// 步骤7: 得出每个电影的评分人数，然后创建(K, V)
//       K = user
//       V = Tuple3<movie, rating, numberOfRaters>
//
// PairFlatMapFunction<T, K, V>
// T => Iterable<Tuple2<K, V>>
JavaPairRDD<String,Tuple3<String,Integer,Integer>> usersRDD =
     moviesGrouped.flatMapToPair(new PairFlatMapFunction<
        Tuple2<String, Iterable<Tuple2<String,Integer>>>, // T
        String,                                           // K
        Tuple3<String,Integer,Integer>>() {               // V
  public Iterable<Tuple2<String,Tuple3<String,Integer,Integer>>>
     call(Tuple2<String, Iterable<Tuple2<String,Integer>>> s) {
     List<Tuple2<String,Integer>> listOfUsersAndRatings =
       new ArrayList<Tuple2<String,Integer>>();
      // 现在读取输入，并生成所需的(K,V)对
     String movie = s._1;
     Iterable<Tuple2<String,Integer>> pairsOfUserAndRating = s._2;
     int numberOfRaters = 0;
     for (Tuple2<String,Integer> t2 : pairsOfUserAndRating) {
         numberOfRaters++;
         listOfUsersAndRatings.add(t2);
     }

      // 现在发出(K, V)对
     List<Tuple2<String,Tuple3<String,Integer,Integer>>> results =
        new ArrayList<Tuple2<String,Tuple3<String,Integer,Integer>>>();
     for (Tuple2<String,Integer> t2 : listOfUsersAndRatings) {
         String user = t2._1;
         Integer rating = t2._2;
         Tuple3<String,Integer,Integer> t3 =
           new Tuple3<String,Integer,Integer>(movie, rating, numberOfRaters);
           results.add(new Tuple2<String, Tuple3<String,Integer,Integer>>
             (user, t3));
     }
     return results;
  }
});
```

我们使用了JavaPairRDD.collect()方法来调试步骤7：

```
System.out.println("=== debug3: moviesGrouped: K = user, "+
    "V = Tuple3<movie, rating, numberOfRaters> ===");
List<
      Tuple2<String,Tuple3<String,Integer,Integer>>
     >  debug3 = usersRDD.collect();
```

```
for (Tuple2<String,Tuple3<String,Integer,Integer>> t2 : debug3) {
  System.out.println("debug3 key="+t2._1 + "\t value="+t2._2);
}
```

执行步骤7之后，usersRDD是以下键-值对列表，包含每个电影的评分人数。在这里K=user，V=(movie，rating,numberOfRaters)：

```
{
  (user1, (movie1, 1, 2)),
  (user1, (movie2, 2, 2))
  (user1, (movie3, 3, 2)),
  (user2, (movie1, 1, 2)),
  (user2, (movie2, 2, 2)),
  (user2, (movie3, 3, 2)),
  (user2, (movie5, 5, 1)),
  ...
}
```

## 步骤8：完成自连接

这一步使用了Spark API的一个非常强大的特性：join操作。可以如下在usersRDD上完成自连接（将在user键上完成连接）：

```
joinedRDD = usersRDD.join(usersRDD);
```

这里usersRDD和joinedRDD都是JavaRDD类型。join()的签名如下：

```
public <W> JavaPairRDD<K,scala.Tuple2<V,W>> join(JavaPairRDD<K,W> other)
```

描述：返回一个RDD，其中包含this和other中有相同键的所有元素对。每个元素对作为一个（k，(V1，V2)）元组返回，其中(k，V1)在this中，（k，V2）在other中。跨集群完成一个散列连接。

示例10-15展示了JavaPairRDD.join(JavaPairRDD)提供的自连接实现（需要这个方法来查找（movie1，movie2）组合）。

示例10-15：步骤8: 完成自连接

```
1   // 步骤8: 将usersRDD与自身连接，找出所有（movie1，movie2）对
2   // usersRDD = <K=user，V=<movie,rating,numberOfRaters>>
3   // 结果是joinedRDD = (user，T2((m1,r1,n1)，(m2,r2,n2))).
4   // 连接在"user"键上完成。
5   JavaPairRDD<String,
6               Tuple2<Tuple3<String,Integer,Integer>,
7                      Tuple3<String,Integer,Integer>
8                     >
9               >
10              joinedRDD = usersRDD.join(usersRDD);
11  List<
12       Tuple2<String,
13              Tuple2<Tuple3<String,Integer,Integer>,
14                     Tuple3<String,Integer,Integer>
```

```
15                    >
16               >
17        > debug5 = joinedRDD.collect();
18    for (Tuple2<String, Tuple2<Tuple3<String,Integer,Integer>,
19                                Tuple3<String,Integer,Integer>>
20               > t2 : debug5) {
21       System.out.println("debug5 key="+t2._1 + "\t value="+t2._2);
22    }
```

执行步骤8之后，joinedRDD是以下键-值对列表：

```
{
  (user1, [(movie1, 1, 2), (movie2, 2, 2)]),
  (user1, [(movie1, 1, 2), (movie3, 3, 2)]),
  (user1, [(movie2, 2, 2), (movie1, 1, 2)]),
  (user1, [(movie2, 2, 2), (movie3, 3, 2)]),
  (user1, [(movie3, 3, 2), (movie1, 1, 2)]),
  (user1, [(movie3, 3, 2), (movie2, 2, 2)]),
  ...
}
```

可以看到，对于每个K=user，会有重复的电影对：需要说明，（movie1,movie2）与（movie2, movie1）是一样的。下一步将通过filter()方法删除重复的电影对。

## 步骤9：删除重复的（movie1, movie2）对

在上一步中，我们可能会生成有重复的（movie1, movie2）对。这一步将使用JavaPairRDD.filter()方法删除这些重复的电影对。filter()方法定义如下：

```
public JavaPairRDD<K,V> filter(Function<scala.Tuple2<K,V>,Boolean> f)

Description: Return a new RDD containing only the
             elements that satisfy a predicate.
```

示例10-16展示了过滤器实现。只有当movie1 < movie2时，filter()方法才会保留（movie1, movie2）对。

示例10-16：步骤9: 删除重复的（movie1, movie2）对

```
1    // 步骤9: 删除重复的（movie1, movie2）对。
2    // 需要说明（movie1, movie2）和（movie2, movie1）是一样的
3    // 现在我们有filteredRDD = (user, T2((m1,r1,n1), (m2,r2,n2))),
4    // 在这里 m1 < m2，以保证每个用户不会有重复的
5    // 电影对
6    JavaPairRDD<
7                String,
8                Tuple2<Tuple3<String,Integer,Integer>,
9                       Tuple3<String,Integer,Integer>
10                     >
11                     >
12                     filteredRDD = joinedRDD.filter(new Function<
13                  Tuple2<
```

```
14                        String,
15                        Tuple2<Tuple3<String,Integer,Integer>,
16                              Tuple3<String,Integer,Integer>
17                             >
18                       >, Boolean>() {
19              public Boolean call(Tuple2<String,
20                                         Tuple2<Tuple3<String,Integer,Integer>,
21                                                Tuple3<String,Integer,Integer>
22                                               >
23                                        > s){
24                 // 要删除重复项，确保对于所有（movie1, movie2）对
25                 // 都有movie1 < movie2
26                 Tuple3<String,Integer,Integer> movie1 = s._2._1;
27                 Tuple3<String,Integer,Integer> movie2 = s._2._2;
28                 String movieName1 = movie1.first();
29                 String movieName2 = movie2.first();
30                 if (movieName1.compareTo(movieName2) < 0) {
31                    return true;
32                 }
33                 else {
34                    return false;
35                 }
36           }
37     });
```

我们使用了JavaPairRDD.collect()方法来调试步骤9:

```
List<
     Tuple2<
           String,
           Tuple2<Tuple3<String,Integer,Integer>,
                  Tuple3<String,Integer,Integer>>
           >
     > debug55 = filteredRDD.collect();
for (Tuple2<String,
            Tuple2<Tuple3<String,Integer,Integer>,
                   Tuple3<String,Integer,Integer>>
            > t2 : debug55) {
  System.out.println("debug55 key="+t2._1 + "\t value="+t2._2);
}
```

执行步骤9之后，filteredRDD是以下键-值对列表（无重复）：

```
{
  (user1, [(movie1, 1, 2), (movie2, 2, 2)]),
  (user1, [(movie1, 1, 2), (movie3, 3, 2)]),
  (user1, [(movie2, 2, 2), (movie3, 3, 2)]),
  ...
}
```

## 步骤10：生成所有(movie1, movie2)组合

对于各个（movie1, movie2）对，这一步（参见示例10-17）会创建一些导出数据，将用

于计算3个关联度：Pearson（皮尔逊关联度）、cosine（余弦关联度）和Jaccard（杰卡德关联度）。我们使用JavaPairRDD.mapToPair()通过另一个变换来得到这个结果。

示例10-17：步骤10: 生成所有（movie1, movie2）组合

```
//步骤10: 生成所有（movie1, movie2）组合
// 这一步的目标是创建（K, V）对
// K: Tuple2(movie1, movie2)
// V: Tuple7(movie1.rating,
//           movie1.numOfRaters,
//           movie2.rating,
//           movie2.numOfRaters,
//           ratingProduct,
//           rating1Squared, = movie1.rating * movie1.rating
//           rating2Squared = movie2.rating * movie2.rating
//          )
// PairFunction<T, K, V>
// T => Tuple2<K, V>
// 这个阶段将去除user属性。
//           K                  V
JavaPairRDD<Tuple2<String,String>,
            Tuple7<Integer,Integer,Integer,Integer,
                   Integer,Integer,Integer>>
      moviePairs = filteredRDD.mapToPair(new PairFunction
           <Tuple2<String,Tuple2<Tuple3<String,Integer,Integer>,
                                 Tuple3<String,Integer,Integer>>>, // T
            Tuple2<String,String>,                                 // K
            Tuple7<Integer,Integer,Integer,Integer,
                   Integer,Integer,Integer> // V
           >() {
   public Tuple2<Tuple2<String,String>, Tuple7<Integer,Integer,Integer,
                 Integer,Integer,Integer,Integer>>
     call(Tuple2<String,Tuple2<Tuple3<String,Integer,Integer>,
                               Tuple3<String,Integer,Integer>>> s) {
         // String user = s._1; // 将其去除
         Tuple3<String,Integer,Integer> movie1 = s._2._1;
         Tuple3<String,Integer,Integer> movie2 = s._2._2;
         Tuple2<String,String> m1m2Key =
            new Tuple2<String,String>(movie1.first(), movie2.first());
         // movie1.rating * movie2.rating;
         int ratingProduct = movie1.second() * movie2.second();
         // movie1.rating * movie1.rating;
         int rating1Squared = movie1.second() * movie1.second();
         // movie2.rating * movie2.rating;
         int rating2Squared = movie2.second() * movie2.second();
         Tuple7<Integer,Integer,Integer,Integer,Integer,Integer,Integer> t7 =
           new Tuple7<Integer,Integer,Integer,Integer,
                      Integer,Integer,Integer>(
                movie1.second(), // movie1.rating,
                movie1.third(),  // movie1.numberOfRaters,
                movie2.second(), // movie2.rating,
                movie2.third(),  // movie2.numberOfRaters,
                ratingProduct,   // movie1.rating * movie2.rating
                rating1Squared,  // movie1.rating * movie1.rating
                rating2Squared   // movie2.rating * movie2.rating
```

```
            );
        return new Tuple2<Tuple2<String,String>,
                          Tuple7<Integer,Integer,Integer,Integer,
                                 Integer,Integer,Integer>>(m1m2Key, t7);
      }
  });
```

## 步骤11：电影对分组

这一步如示例10-18所示，将按电影对（movie1, movie2）将数据分组,来收集计算关联度所需的数据。

示例10-18：步骤11: 电影对分组

```
//步骤11: 按键(movie1,movie2)对moviePairs分组
JavaPairRDD<Tuple2<String,String>,
            Iterable<Tuple7<Integer,Integer,Integer,Integer,
                            Integer,Integer,Integer>>
          > corrRDD = moviePairs.groupByKey();
```

## 步骤12：计算关联度

这一步如示例10-19所示，使用了3种不同的关联度算法来找出每一对电影之间的相似度。一个特定的应用具体选择什么关联度算法，对此并没有黄金定律。应当选择一种适合具体应用环境的关联度算法。

示例10-19：步骤12: 计算关联度

```
// 步骤12: 计算每两个电影（movie1,movie2）
// 之间的关联度/相似度：
// coor.key = (movie1,movie2)
// corr.value = (pearson, cosine, jaccard)关联度
JavaPairRDD<Tuple2<String,String>, Tuple3<Double,Double,Double>> corr =
      corrRDD.mapValues(new Function<
          Iterable<Tuple7<Integer,Integer,Integer,Integer,
                          Integer,Integer,Integer>>,   // 输入
          Tuple3<Double,Double,Double>                 // 输出
      >() {
   public Tuple3<Double,Double,Double> call(
      Iterable<Tuple7<Integer,Integer,Integer,
                      Integer,Integer,Integer,Integer>> s) {
       return calculateCorrelations(s);
    }
  });
```

## 步骤13：打印最终结果

最后一步要打印最终结果。如示例10-20所示。

示例10-20：步骤13: 打印最终结果

```
// 步骤13: 打印最终结果
System.out.println("=== Movie Correlations ===");
```

```
3    List<
4         Tuple2< Tuple2<String,String>, Tuple3<Double,Double,Double>>
5        >  debug6 = corr.collect();
6    for (Tuple2<Tuple2<String,String>, Tuple3<Double,Double,Double>> t2 : debug6) {
7      System.out.println("debug5 key="+t2._1 + "\t value="+t2._2);
8    }
9
10   corr.saveAsTextFile("/movies/output");
```

## 辅助方法

示例10-21给出了辅助方法calculateCorrelations()的定义。

示例10-21：辅助方法：calculateCorrelations()

```
1   static Tuple3<Double,Double,Double> calculateCorrelations(
2      Iterable<Tuple7<Integer,Integer,Integer,Integer,Integer,Integer,Integer>>
3      values) {
4      int groupSize = 0; //各个向量的长度
5      int dotProduct = 0; // ratingProd之和
6      int rating1Sum = 0; // rating1之和
7      int rating2Sum = 0; // rating2之和
8      int rating1NormSq = 0; // rating1Squared之和
9      int rating2NormSq = 0; // rating2Squared之和
10     int maxNumOfumRaterS1 = 0; // numOfRaterS1最大值
11     int maxNumOfumRaterS2 = 0; // numOfRaterS2最大值
12     for (Tuple7<Integer,Integer,Integer,Integer,Integer,Integer,Integer> t7 :
13        values) {
14              //Tuple7(<rating1>:   t7._1
15              // <numOfRaterS1>:    t7._2
16              // <rating2>:         t7._3
17              // <numOfRaterS2> :   t7._4
18              // <ratingProduct> :  t7._5
19              // <rating1Squared> : t7._6
20              // <rating2Squared>): t7._7
21          groupSize++;
22          dotProduct += t7._5;     // ratingProduct;
23          rating1Sum += t7._1;     // rating1;
24          rating2Sum += t7._3;     // rating2;
25          rating1NormSq += t7._6;  // rating1Squared;
26          rating2NormSq += t7._7;  // rating2Squared;
27          int numOfRaterS1 = t7._2;
28          if (numOfRaterS1 > maxNumOfumRaterS1) {
29            maxNumOfumRaterS1 = numOfRaterS1;
30          }
31          int numOfRaterS2 = t7._4;
32          if (numOfRaterS2 > maxNumOfumRaterS2) {
33             maxNumOfumRaterS2 = numOfRaterS2;
34          }
35     }
36
37     double pearson = calculatePearsonCorrelation(
38               groupSize,
39               dotProduct,
40               rating1Sum,
```

```
41                rating2Sum,
42                rating1NormSq,
43                rating2NormSq);
44
45      double cosine = calculateCosineCorrelation(dotProduct,
46                                                 Math.sqrt(rating1NormSq),
47                                                 Math.sqrt(rating2NormSq));
48
49      double jaccard = calculateJaccardCorrelation(groupSize,
50                                                 maxNumOfumRaterS1,
51                                                 maxNumOfumRaterS2);
52
53      return new Tuple3<Double,Double,Double>(pearson, cosine, jaccard);
54 }
```

示例10-22定义了辅助方法calculatePearsonCorrelation()，会计算两个电影的皮尔逊关联度。

示例10-22：辅助方法：calculatePearsonCorrelation()

```
 1 static double calculatePearsonCorrelation(
 2                 double size,
 3                 double dotProduct,
 4                 double rating1Sum,
 5                 double rating2Sum,
 6                 double rating1NormSq,
 7                 double rating2NormSq) {
 8
 9    double numerator = size * dotProduct - rating1Sum * rating2Sum;
10      double denominator =
11        Math.sqrt(size * rating1NormSq - rating1Sum * rating1Sum) *
12        Math.sqrt(size * rating2NormSq - rating2Sum * rating2Sum);
13      return numerator / denominator;
14    }
```

示例10-23定义了辅助方法calculateCosineCorrelation()。

示例10-23：辅助方法：calculateCosineCorrelation()

```
1   /**
2    * 两个向量A和B之间的余弦相似度为
3    * dotProduct(A, B) / (norm(A) * norm(B))
4    */
5  static double calculateCosineCorrelation(double dotProduct,
6                                           double rating1Norm,
7                                           double rating2Norm) {
8    return dotProduct / (rating1Norm * rating2Norm);
9  }
```

示例10-24定义了辅助方法calculateJaccardCorrelation()，可以计算两个电影的杰卡德关联度。

示例10-24：辅助方法：calculateJaccardCorrelation()

```
/**
 * 两个集合A和B之间的杰卡德相似度为
 * |Intersection(A, B)| / |Union(A, B)|
 */
static double calculateJaccardCorrelation(double inCommon,
                                          double totalA,
                                          double totalB) {
  double union = totalA + totalB - inCommon;
  return inCommon / union;
}
```

# Spark程序运行示例

下面各小节将给出这个Spark程序的输入、脚本、示例运行日志和期望的输出。

## HDFS输入

重申一次，这里假设我们的输入数据作为一个文本文件存储在HDFS中：

```
# hadoop fs -cat /movies.txt
User1   Movie1   3
User1   Movie2   4
User1   Movie3   3
User2   Movie1   2
User2   Movie2   5
User2   Movie3   3
User2   Movie5   5
User3   Movie1   2
User3   Movie2   3
User3   Movie3   2
User4   Movie1   5
User4   Movie2   3
User4   Movie3   3
User4   Movie4   2
User4   Movie5   3
```

## Script

下面是Spark程序的脚本：

```
# cat run_movies.sh
#!/bin/bash
export JAVA_HOME=/usr/java/jdk7
export SPARK_HOME=/home/hadoop/spark-1.1.0
export SPARK_MASTER=spark://myserver100:7077
BOOK_HOME=/mp/data-algorithms-book
APP_JAR=$BOOK_HOME/dist/data_algorithms_book.jar
MOVIES=/movies.txt
# 在Spark独立集群上运行
$SPARK_HOME/bin/spark-submit \
  --class org.dataalgorithms.chap10.spark.MovieRecommendationsWithJoin \
```

```
  --master $SPARK_MASTER \
  --executor-memory 2G \
  --total-executor-cores 20 \
$APP_JAR \
$MOVIES
```

## 示例运行日志

下面给出示例运行的实际输出，由于版面所限，这里对输出做了编辑：

```
# ./run_movies.sh
usersRatingsInputFile=/movies.txt
...
14/06/07 23:18:13 INFO scheduler.DAGScheduler: Stage 0
  (collect at MovieRecommendationsWithJoin.java:73) finished in 4.199 s
14/06/07 23:18:13 INFO spark.SparkContext: Job finished: collect at
  MovieRecommendationsWithJoin.java:73, took 4.290270699 s
debug1 key=Movie1 value=(User1,3)
debug1 key=Movie2 value=(User1,4)
debug1 key=Movie3 value=(User1,3)
debug1 key=Movie1 value=(User2,2)
debug1 key=Movie2 value=(User2,5)
debug1 key=Movie3 value=(User2,3)
debug1 key=Movie5 value=(User2,5)
debug1 key=Movie1 value=(User3,2)
debug1 key=Movie2 value=(User3,3)
debug1 key=Movie3 value=(User3,2)
debug1 key=Movie1 value=(User4,5)
debug1 key=Movie2 value=(User4,3)
debug1 key=Movie3 value=(User4,3)
debug1 key=Movie4 value=(User4,2)
debug1 key=Movie5 value=(User4,3)
=== debug2: moviesGrouped: K = <movie>, V = Iterable<Tuple2<user, rating>> ===
14/06/07 23:18:13 INFO spark.SparkContext: Starting job: collect at
  MovieRecommendationsWithJoin.java:85
14/06/07 23:18:13 INFO scheduler.DAGScheduler: Registering RDD 2
  (mapToPair at MovieRecommendationsWithJoin.java:56)
...
14/06/07 23:18:13 INFO scheduler.DAGScheduler: Stage 1

  (collect at MovieRecommendationsWithJoin.java:85) finished in 0.752 s
14/06/07 23:18:13 INFO spark.SparkContext: Job finished: collect at
  MovieRecommendationsWithJoin.java:85, took 0.833602896 s
debug2 key=Movie2 value=[(User1,4), (User2,5), (User3,3), (User4,3)]
debug2 key=Movie5 value=[(User2,5), (User4,3)]
debug2 key=Movie4 value=[(User4,2)]
debug2 key=Movie3 value=[(User1,3), (User2,3), (User3,2), (User4,3)]
debug2 key=Movie1 value=[(User1,3), (User2,2), (User3,2), (User4,5)]
=== debug3: moviesGrouped: K = user, V = Tuple3<movie, rating, numberOfRaters>
14/06/07 23:18:13 INFO spark.SparkContext: Starting job: collect at
..MovieRecommendationsWithJoin.java:126
...
14/06/07 23:18:14 INFO scheduler.DAGScheduler: Stage 3
  (collect at MovieRecommendationsWithJoin.java:126) finished in 0.141 s
```

```
14/06/07 23:18:14 INFO spark.SparkContext: Job finished:
  collect at MovieRecommendationsWithJoin.java:126, took 0.159691236 s
debug3 key=User1 value=Tuple3[Movie2,4,4]
debug3 key=User2 value=Tuple3[Movie2,5,4]
debug3 key=User3 value=Tuple3[Movie2,3,4]
debug3 key=User4 value=Tuple3[Movie2,3,4]
debug3 key=User2 value=Tuple3[Movie5,5,2]
debug3 key=User4 value=Tuple3[Movie5,3,2]
debug3 key=User4 value=Tuple3[Movie4,2,1]
debug3 key=User1 value=Tuple3[Movie3,3,4]
debug3 key=User2 value=Tuple3[Movie3,3,4]
debug3 key=User3 value=Tuple3[Movie3,2,4]
debug3 key=User4 value=Tuple3[Movie3,3,4]
debug3 key=User1 value=Tuple3[Movie1,3,4]
debug3 key=User2 value=Tuple3[Movie1,2,4]
debug3 key=User3 value=Tuple3[Movie1,2,4]
debug3 key=User4 value=Tuple3[Movie1,5,4]
14/06/07 23:18:14 INFO spark.SparkContext: Starting job: collect
  at MovieRecommendationsWithJoin.java:142
...
14/06/07 23:18:15 INFO scheduler.DAGScheduler: Stage 5 (collect at
  MovieRecommendationsWithJoin.java:142) finished in 0.728 s
14/06/07 23:18:15 INFO spark.SparkContext: Job finished: collect at
  MovieRecommendationsWithJoin.java:142, took 0.846712724 s

debug5 key=User3 value=(Tuple3[Movie2,3,4],Tuple3[Movie2,3,4])
debug5 key=User3 value=(Tuple3[Movie2,3,4],Tuple3[Movie3,2,4])
debug5 key=User3 value=(Tuple3[Movie2,3,4],Tuple3[Movie1,2,4])
debug5 key=User3 value=(Tuple3[Movie3,2,4],Tuple3[Movie2,3,4])
debug5 key=User3 value=(Tuple3[Movie3,2,4],Tuple3[Movie3,2,4])
debug5 key=User3 value=(Tuple3[Movie3,2,4],Tuple3[Movie1,2,4])
debug5 key=User3 value=(Tuple3[Movie1,2,4],Tuple3[Movie2,3,4])
debug5 key=User3 value=(Tuple3[Movie1,2,4],Tuple3[Movie3,2,4])
debug5 key=User3 value=(Tuple3[Movie1,2,4],Tuple3[Movie1,2,4])
debug5 key=User2 value=(Tuple3[Movie2,5,4],Tuple3[Movie2,5,4])
debug5 key=User2 value=(Tuple3[Movie2,5,4],Tuple3[Movie5,5,2])
debug5 key=User2 value=(Tuple3[Movie2,5,4],Tuple3[Movie3,3,4])
debug5 key=User2 value=(Tuple3[Movie2,5,4],Tuple3[Movie1,2,4])
debug5 key=User2 value=(Tuple3[Movie5,5,2],Tuple3[Movie2,5,4])
debug5 key=User2 value=(Tuple3[Movie5,5,2],Tuple3[Movie5,5,2])
debug5 key=User2 value=(Tuple3[Movie5,5,2],Tuple3[Movie3,3,4])
debug5 key=User2 value=(Tuple3[Movie5,5,2],Tuple3[Movie1,2,4])
debug5 key=User2 value=(Tuple3[Movie3,3,4],Tuple3[Movie2,5,4])
debug5 key=User2 value=(Tuple3[Movie3,3,4],Tuple3[Movie5,5,2])
debug5 key=User2 value=(Tuple3[Movie3,3,4],Tuple3[Movie3,3,4])
debug5 key=User2 value=(Tuple3[Movie3,3,4],Tuple3[Movie1,2,4])
debug5 key=User2 value=(Tuple3[Movie1,2,4],Tuple3[Movie2,5,4])
debug5 key=User2 value=(Tuple3[Movie1,2,4],Tuple3[Movie5,5,2])
debug5 key=User2 value=(Tuple3[Movie1,2,4],Tuple3[Movie3,3,4])
debug5 key=User2 value=(Tuple3[Movie1,2,4],Tuple3[Movie1,2,4])
debug5 key=User4 value=(Tuple3[Movie2,3,4],Tuple3[Movie2,3,4])
debug5 key=User4 value=(Tuple3[Movie2,3,4],Tuple3[Movie5,3,2])
debug5 key=User4 value=(Tuple3[Movie2,3,4],Tuple3[Movie4,2,1])
debug5 key=User4 value=(Tuple3[Movie2,3,4],Tuple3[Movie3,3,4])
debug5 key=User4 value=(Tuple3[Movie2,3,4],Tuple3[Movie1,5,4])
```

```
debug5 key=User4 value=(Tuple3[Movie5,3,2],Tuple3[Movie2,3,4])
debug5 key=User4 value=(Tuple3[Movie5,3,2],Tuple3[Movie5,3,2])
debug5 key=User4 value=(Tuple3[Movie5,3,2],Tuple3[Movie4,2,1])
debug5 key=User4 value=(Tuple3[Movie5,3,2],Tuple3[Movie3,3,4])
debug5 key=User4 value=(Tuple3[Movie5,3,2],Tuple3[Movie1,5,4])
debug5 key=User4 value=(Tuple3[Movie4,2,1],Tuple3[Movie2,3,4])
debug5 key=User4 value=(Tuple3[Movie4,2,1],Tuple3[Movie5,3,2])
debug5 key=User4 value=(Tuple3[Movie4,2,1],Tuple3[Movie4,2,1])
debug5 key=User4 value=(Tuple3[Movie4,2,1],Tuple3[Movie3,3,4])
debug5 key=User4 value=(Tuple3[Movie4,2,1],Tuple3[Movie1,5,4])
debug5 key=User4 value=(Tuple3[Movie3,3,4],Tuple3[Movie2,3,4])
debug5 key=User4 value=(Tuple3[Movie3,3,4],Tuple3[Movie5,3,2])
debug5 key=User4 value=(Tuple3[Movie3,3,4],Tuple3[Movie4,2,1])
debug5 key=User4 value=(Tuple3[Movie3,3,4],Tuple3[Movie3,3,4])
debug5 key=User4 value=(Tuple3[Movie3,3,4],Tuple3[Movie1,5,4])
debug5 key=User4 value=(Tuple3[Movie1,5,4],Tuple3[Movie2,3,4])
debug5 key=User4 value=(Tuple3[Movie1,5,4],Tuple3[Movie5,3,2])
debug5 key=User4 value=(Tuple3[Movie1,5,4],Tuple3[Movie4,2,1])
debug5 key=User4 value=(Tuple3[Movie1,5,4],Tuple3[Movie3,3,4])
debug5 key=User4 value=(Tuple3[Movie1,5,4],Tuple3[Movie1,5,4])
debug5 key=User1 value=(Tuple3[Movie2,4,4],Tuple3[Movie2,4,4])
debug5 key=User1 value=(Tuple3[Movie2,4,4],Tuple3[Movie3,3,4])
debug5 key=User1 value=(Tuple3[Movie2,4,4],Tuple3[Movie1,3,4])
debug5 key=User1 value=(Tuple3[Movie3,3,4],Tuple3[Movie2,4,4])
debug5 key=User1 value=(Tuple3[Movie3,3,4],Tuple3[Movie3,3,4])
debug5 key=User1 value=(Tuple3[Movie3,3,4],Tuple3[Movie1,3,4])
debug5 key=User1 value=(Tuple3[Movie1,3,4],Tuple3[Movie2,4,4])
debug5 key=User1 value=(Tuple3[Movie1,3,4],Tuple3[Movie3,3,4])
debug5 key=User1 value=(Tuple3[Movie1,3,4],Tuple3[Movie1,3,4])
14/06/07 23:18:15 INFO spark.SparkContext: Starting job: collect at
  MovieRecommendationsWithJoin.java:171
...
14/06/07 23:18:15 INFO scheduler.DAGScheduler: Stage 9 (collect at
  MovieRecommendationsWithJoin.java:171) finished in 0.109 s
14/06/07 23:18:15 INFO spark.SparkContext: Job finished: collect at
  MovieRecommendationsWithJoin.java:171, took 0.13069563 s
debug55 key=User3 value=(Tuple3[Movie2,3,4],Tuple3[Movie3,2,4])
debug55 key=User3 value=(Tuple3[Movie1,2,4],Tuple3[Movie2,3,4])
debug55 key=User3 value=(Tuple3[Movie1,2,4],Tuple3[Movie3,2,4])
debug55 key=User2 value=(Tuple3[Movie2,5,4],Tuple3[Movie5,5,2])
debug55 key=User2 value=(Tuple3[Movie2,5,4],Tuple3[Movie3,3,4])
debug55 key=User2 value=(Tuple3[Movie3,3,4],Tuple3[Movie5,5,2])
debug55 key=User2 value=(Tuple3[Movie1,2,4],Tuple3[Movie2,5,4])
debug55 key=User2 value=(Tuple3[Movie1,2,4],Tuple3[Movie5,5,2])
debug55 key=User2 value=(Tuple3[Movie1,2,4],Tuple3[Movie3,3,4])
debug55 key=User4 value=(Tuple3[Movie2,3,4],Tuple3[Movie5,3,2])
debug55 key=User4 value=(Tuple3[Movie2,3,4],Tuple3[Movie4,2,1])
debug55 key=User4 value=(Tuple3[Movie2,3,4],Tuple3[Movie3,3,4])
debug55 key=User4 value=(Tuple3[Movie4,2,1],Tuple3[Movie5,3,2])
debug55 key=User4 value=(Tuple3[Movie3,3,4],Tuple3[Movie5,3,2])
debug55 key=User4 value=(Tuple3[Movie3,3,4],Tuple3[Movie4,2,1])
debug55 key=User4 value=(Tuple3[Movie1,5,4],Tuple3[Movie2,3,4])
debug55 key=User4 value=(Tuple3[Movie1,5,4],Tuple3[Movie5,3,2])
debug55 key=User4 value=(Tuple3[Movie1,5,4],Tuple3[Movie4,2,1])
debug55 key=User4 value=(Tuple3[Movie1,5,4],Tuple3[Movie3,3,4])
```

```
debug55 key=User1 value=(Tuple3[Movie2,4,4],Tuple3[Movie3,3,4])
debug55 key=User1 value=(Tuple3[Movie1,3,4],Tuple3[Movie2,4,4])
debug55 key=User1 value=(Tuple3[Movie1,3,4],Tuple3[Movie3,3,4])
=== Movie Correlations ===
14/06/07 23:18:15 INFO spark.SparkContext: Starting job: collect at
  MovieRecommendationsWithJoin.java:242
...
14/06/07 23:18:15 INFO scheduler.DAGScheduler: Stage 13 (collect
  at MovieRecommendationsWithJoin.java:242) finished in 0.075 s
14/06/07 23:18:15 INFO spark.SparkContext: Job finished: collect
  at MovieRecommendationsWithJoin.java:242, took 0.216924159 s
debug5 key=(Movie2,Movie3)
       value=Tuple3[0.5222329678670935,0.9820699844444069,1.0]
debug5 key=(Movie1,Movie3)
       value=Tuple3[0.47140452079103173,0.9422657923052785,1.0]
debug5 key=(Movie3,Movie4)
       value=Tuple3[NaN,1.0,0.25]
debug5 key=(Movie3,Movie5)
       value=Tuple3[NaN,0.970142500145332,0.5]
debug5 key=(Movie4,Movie5)
       value=Tuple3[NaN,1.0,0.5]
debug5 key=(Movie1,Movie2)
       value=Tuple3[-0.492365963917331,0.8638091589670809,1.0]
debug5 key=(Movie2,Movie5)
       value=Tuple3[1.0,1.0,0.5]
debug5 key=(Movie1,Movie5)
       value=Tuple3[-1.0,0.7961621941231025,0.5]
debug5 key=(Movie1,Movie4)
       value=Tuple3[NaN,1.0,0.25]
debug5 key=(Movie2,Movie4)
       value=Tuple3[NaN,1.0,0.25]
...
14/06/07 23:18:16 INFO scheduler.DAGScheduler: Stage 18 (saveAsTextFile at
  MovieRecommendationsWithJoin.java:247) finished in 1.014 s
14/06/07 23:18:16 INFO spark.SparkContext: Job finished: saveAsTextFile at
  MovieRecommendationsWithJoin.java:247, took 1.131578238 s
```

## HDFS输出

下面是这个Spark程序的期望HDFS输出：

```
# hadoop fs -ls /movies/output/
Found 2 items
-rw-r--r-- ...            0 2014-06-07 23:18 /movies/output/_SUCCESS
-rw-r--r-- ...          504 2014-06-07 23:18 /movies/output/part-00000
# hadoop fs -cat /movies/output/part-00000
((Movie2,Movie3),Tuple3[0.5222329678670935,0.9820699844444069,1.0])
((Movie1,Movie3),Tuple3[0.47140452079103173,0.9422657923052785,1.0])
((Movie3,Movie4),Tuple3[NaN,1.0,0.25])
((Movie3,Movie5),Tuple3[NaN,0.970142500145332,0.5])
((Movie4,Movie5),Tuple3[NaN,1.0,0.5])
((Movie1,Movie2),Tuple3[-0.492365963917331,0.8638091589670809,1.0])
((Movie2,Movie5),Tuple3[1.0,1.0,0.5])
((Movie1,Movie5),Tuple3[-1.0,0.7961621941231025,0.5])
```

```
((Movie1,Movie4),Tuple3[NaN,1.0,0.25])
((Movie2,Movie4),Tuple3[NaN,1.0,0.25])
```

这一章提供了使用相似度的基本分布式推荐算法，如皮尔逊关联系数、杰卡德相似度和余弦相似度。需要说明，要构建实际的推荐引擎，还需要考虑另外一组元数据，如演员名、导演、电影类型和其他相关的属性。

下一章会介绍如何使用马尔可夫（Markov）模型预测将来的事件。

第11章

# 使用马尔可夫模型的智能邮件营销

这一章介绍如何使用马尔可夫模型（最简形式称为马尔可夫链（Markov chain））根据顾客的交易历史观测“下一个智能邮件营销日期”。给定一组随机变量（如顾客的最近交易日期），马尔可夫模型只根据前一个状态（前一个最近交易日期）的分布指示该变量（即最近交易日期）的分布。对于这个“智能邮件营销”问题，我们提供了两个不同的解决方案：

- 使用传统`map()`和`reduce()`函数的MapReduce/Hadoop解决方案。
- 使用有向无环图（一组任意的变换和动作）的Spark解决方案。

写这一章时，我受到Pranab Ghosh博客文章“Smarter Email Marketing with Markov Model”的启发。对于这一章中各个MapReduce阶段的实现，我开发了全新的模块化Java类来展示其核心特性（如通过MapReduce的二次排序技术减少值的排序，定义定制分区器类，以及定义分组比较器类）。因此，给定一个顾客的交易历史［表示为（`purchase-date,amount-purchased`）］，我们的目标就是使用MapReduce和马尔可夫模型来观测下一次向这个顾客发送营销邮件的合理日期。这是一种机器学习算法。一般地，基于机器学习的解决方案包括两个不同阶段：

- 阶段1：使用历史训练数据建立一个模型。
- 阶段2：使用阶段1建立的模型对新数据做出预测。

# 马尔可夫链基本原理

令$S = \{S_1, S_2, S_3, ...\}$是一个有限状态集。我们希望收集以下概率：

$$P(S_n|S_{n-1},S_{n-2}, ..., S_1)$$

一阶马尔可夫假设如下：

$$P(S_n|S_{n-1},S_{n-2}, ..., S_1) \approx P(S_n|S_{n-1})$$

这个近似公式描述了马尔可夫性质（Markov property）：系统在时间$t$+1的状态只基于系统在时间$t$的状态。

二阶马尔可夫假设如下：

$$P(S_n|S_{n-1},S_{n-2}, ..., S_1) \approx \mathrm{P}(S_n|S_{n-1},S_{n-2})$$

下面使用马尔可夫假设描述联合概率：

$$P(S_1,S_2,...,S_n)=\prod_{i=1}^{n}P(S_i,S_{i-1})$$

马尔可夫随机过程可以总结如下[注1]：

- 如果一个随机序列的分布仅由其当前状态确定，则具有马尔可夫性质。具有这个性质的随机过程则为马尔可夫随机过程（Markov random process）。
- 对于可观察的状态序列（即状态由数据可知），可以得到一个马尔可夫链模型（Markov chain model），我们可以使用这个模型来预测下一个有效的邮件营销日期。
- 对于不可观察状态，会得到一个隐式马尔可夫模型（hidden Markov model，HMM）。

现在来给出本章将用到的马尔可夫链的形式化表示。我们的马尔可夫链包括3个部分：

*状态空间（State space）*

有限状态集$S = \{ S_1, S_2, S_3 ...\}$

*转移概率（Transition probabilities）*

函数$f$: $S \times S \rightarrow R$：

- $0 \leqslant f(a, b) \leqslant 1$ （对于所有$a, b \in S$）

注1：参考了Mehmet Yunus Dnmez的幻灯片：*http://bit.ly/hidden_markov_mod*。

- $\sum_{b \in S} f(a,b) = 1$ （对于各个$a \in S$）

*初始分布（Initial distribution）*

函数$g$: $S \times R$：

- $0 \leqslant g(a) \leqslant 1$ （对于各个$a \in S$）
- $\sum_{a \in S} g(a) = 1$

则马尔可夫链是$S$中的一个随机过程：

- 时间0时，这个链的状态用分布函数$g$描述。
- 如果时间$t$时马尔可夫链的状态为$a$，则时间$t + 1$时，对于各个$b \in S$，其状态为$b$的概念为$f(a, b)$。

下面来看一个例子：假设一个城市的天气变化包括4种状态——sunny（晴天），cloudy（多云），rainy（有雨）和foggy（有雾），进一步假设一天中天气状态不会改变。表11-1中每行的概率之和为1.00。

**表11-1：城市天气变化（每天仅一个状态）**

| 今天天气 \ 明天天气 | 晴天 | 有雨 | 多云 | 有雾 |
|---|---|---|---|---|
| 晴天 | 0.6 | 0.1 | 0.2 | 0.1 |
| 有雨 | 0.5 | 0.2 | 0.2 | 0.1 |
| 多云 | 0.1 | 0.7 | 0.1 | 0.1 |
| 有雾 | 0.0 | 0.3 | 0.4 | 0.3 |

现在我们可以回答以下问题：

- 如果今天的天气状态是晴天，那么明天多云而且后天有雾的概率是多大？可以如下计算：

$$P(S_2 = \text{cloudy}, S_3 = \text{foggy} | S_1 = \text{sunny})$$

$$= P(S_3 = \text{foggy} | S_2 = \text{cloudy}, S_1 = \text{sunny}) \times$$

$$P(S_2 = \text{cloudy} | S_1 = \text{sunny})$$

$$= P(S_3 = \text{foggy} | S_2 = \text{cloudy}) \times$$

$$P(S_2 = \text{cloudy} | S_1 = \text{sunny})$$

$$= 0.1 \times 0.2$$

$$= 0.02$$

- 如果今天有雾，那么两天后有雨的概率是多大（这说明，第二天的天气可以是晴天、多云、有雨或有雾）？

$P(S_3 = \text{foggy} | S_1 = \text{foggy}) =$

$P(S_3 = \text{foggy}, S_2 = \text{sunny} | S_1 = \text{foggy}) +$

$P(S_3 = \text{foggy}, S_2 = \text{cloudy} | S_1 = \text{foggy}) +$

$P(S_3 = \text{foggy}, S_2 = \text{rainy} | S_1 = \text{foggy}) +$

$P(S_3 = \text{foggy}, S_2 = \text{foggy} | S_1 = \text{foggy}) +$

$= P(S_3 = \text{foggy} | S_2 = \text{sunny}) \times P(S_2 = \text{sunny} | S_1 = \text{foggy}) +$

$P(S_3 = \text{foggy} | S_2 = \text{cloudy}) \times P(S_2 = \text{cloudy} | S_1 = \text{foggy}) +$

$P(S_3 = \text{foggy} | S_2 = \text{rainy}) \times P(S_2 = \text{rainy} | S_1 = \text{foggy}) +$

$P(S_3 = \text{foggy} | S_2 = \text{foggy}) \times P(S_2 = \text{foggy} | S_1 = \text{foggy})$

$= 0.1 \times 0.0 +$

$0.1 \times 0.4 +$

$0.1 \times 0.3 +$

$0.3 \times 0.3$

$= 0.00 + 0.04 + 0.03 + 0.09$

$= 0.16$

这一章的主要目标之一就是建立这样一个模型（也就是转移概率表），可以定义所有$a \in S$的$f(a, b)$。一旦创建了这个模型，其余的任务就容易了。

# 使用MapReduce的马尔可夫模型

假设我们有顾客交易的历史数据，包括transaction-id（交易ID）、customer-id（顾客ID）、purchase-date（交易日期）和amount（金额）。因此，每个输入记录有以下格式：

```
<customerID><,><transactionID><,><purchaseDate><,><amount>
```

整个解决方案包括两个MapReduce作业和一组Ruby脚本（Ruby脚本由Pranab Ghosh开发，用到这些脚本时我会提供相应的链接）。

整个工作流如图11-1所示，下面给出这个解决方案（图11-1中所示的步骤）的概要描述：

1. 使用一个脚本生成虚拟的顾客数据(1)。

2. MapReduce投影（2）接受顾客数据（1）作为输入，生成有序数据（3）。有序数据(3)包括按升序排序的交易日期。
3. 状态转换器脚本(4)接受这个有序数据（3），生成状态序列（5）。
4. MapReduce马尔可夫状态转移模型（6）接受这个状态序列(5)作为输入，生成一个马尔可夫链模型（7）。利用这个模型可以预测下一个状态。
5. 最后，下一状态预测脚本（9）接受新的顾客数据(8)和马尔可夫链模型（7），预测发出下一封营销邮件的最佳日期。

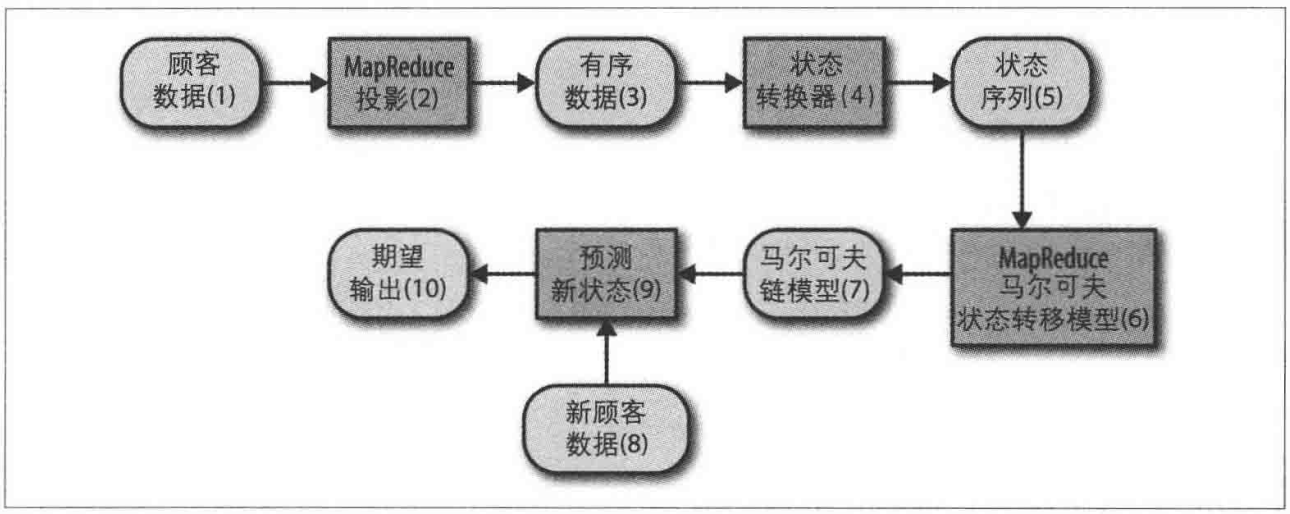



图11-1：马尔可夫工作流

## 使用MapReduce生成时序转移

这个MapReduce阶段的目标是接受顾客历史交易数据，为每一个customer-id生成以下输出：

$$\text{customerID}\ (\text{Date}_1, \text{Amount}_1);(\text{Date}_2, \text{Amount}_2);...(\text{Date}_N, \text{Amount}_N)$$

使得：

$$\text{Date}_1 \leqslant \text{Date}_2 \leqslant \ldots \leqslant \text{Date}_N$$

这个MapReduce输出按交易日期的升序排序。要生成有序的输出，可以采用两种方法：各个归约器按交易日期的升序对输出排序（这需要足够的RAM来保存所有要排序的数据），或者也可以使用MapReduce的二次排序技术按日期对数据排序（采用这种方法时，并不需要太多的RAM）。生成输出之后，我们将把（Date, Amount）转换为一个两字母代号（这个步骤由一个脚本完成），表示一个马尔可夫链状态。我会给出这两种方法的解决方案。这个阶段生成的最终输出将有以下格式：

$$\text{customerID}, \text{State}_1, \text{State}_2, \ldots, \text{State}_n$$

示例11-1定义了时序交易的map()函数。

示例11-1：时序交易：map()函数

```
1 /**
2  * @param key忽略
3  * @param value为transaction-id, customer-id, purchase-date, amount
4  */
5 map(key, value) {
6     pair(Date, Integer) pair = (value.purchase-date, value.amount);
7     emit(value.customer-id, pair);
8 }
```

示例11-2定义了时序交易的reduce()函数。

示例11-2：时序交易：reduce()函数

```
1 /**
2  * @param key为customer-id
3  * @param values为Pair(Date, Integer)对列表
4  */
5 reduce(String key, List<Pair<Date, Integer>> values) {
6    sortedValues = sortbyDateInAscendingOrder(values);
7    emit(key, sortedValues);
8 }
```

## Hadoop解决方案1：时序交易

在这个解决方案中，映射器发出键-值对，其中键是一个customer-id，值是一个（purchase-date, amount）对。到达归约器的数据是无序的。这个解决方案在归约器中对交易排序。如果到达归约器的交易数太大，可能会导致归约器中出现内存溢出异常。我们的第二个Hadoop解决方案不存在这个限制：对于每个归约器，并不是在内存中对数据排序，而会使用二次排序（应该记得，本书前面介绍过），这是MapReduce范式提供的一个特性，可以用来完成归约器值的排序。

表11-2列出了Hadoop解决方案1中需要的实现类。

**表11-2：Hadoop解决方案1：实现类**

| 类名 | 描述 |
|---|---|
| SortInMemoryProjectionDriver | 提交作业的驱动器类 |
| SortInMemoryProjectionMapper | 映射器类 |
| SortInMemoryProjectionReducer | 归约器类 |
| DateUtil | 基本日期工具类 |
| HadoopUtil | 基本Hadoop工具类 |

## 部分输入

Pranab Ghosh提供了一个Ruby脚本（*buy_xaction.rb*）（*https://github.com/pranab/avenir/blob/master/resource/buy_xaction.rb*），可以用这个脚本生成虚拟的顾客交易数据：

```
$ # 使用一个ruby脚本创建测试数据：
$ ./buy_xaction.rb 80000 210 .05 > training.txt

$ # 将测试数据复制到Hadoop/HDFS
$ hadoop fs -copyFromLocal training.txt
    /markov/projection_by_sorting_in_ram/input/

$ # 检查HDFS中的数据
$ hadoop fs -cat
   /markov/projection_by_sorting_in_ram/input/training.txt
...
EY2I3D12PZ,1382705171,2013-07-29,28
VC38QFM2IF,1382705172,2013-07-29,84
1O22R2QPWG,1382705173,2013-07-29,27
4GO2MW73CK,1382705174,2013-07-29,31
VKV2K1SOD2,1382705175,2013-07-29,28
LDFK8WZQFH,1382705176,2013-07-29,25
8874144Q11,1382705177,2013-07-29,180
...
```

## 示例运行日志

日志输出如下所示,因篇幅所限，这里对日志做了编辑，并对格式有所调整：

```
# ./run.sh
...
Deleted hdfs://localhost:9000/lib/projection_by_sorting_in_ram.jar
Deleted hdfs://localhost:9000/markov/projection_by_sorting_in_ram/output
...
13/11/27 12:03:16 INFO mapred.JobClient: Running job: job_201311271011_0012
13/11/27 12:03:17 INFO mapred.JobClient: map 0% reduce 0%
...
13/11/27 12:04:16 INFO mapred.JobClient: map 100% reduce 100%
13/11/27 12:04:17 INFO mapred.JobClient: Job complete: job_201311271011_0012
...
13/11/27 12:04:17 INFO mapred.JobClient: Map-Reduce Framework
13/11/27 12:04:17 INFO mapred.JobClient: Map input records=832280
13/11/27 12:04:17 INFO mapred.JobClient: Reduce input records=832280
13/11/27 12:04:17 INFO mapred.JobClient: Reduce input groups=79998
13/11/27 12:04:17 INFO mapred.JobClient: Reduce output records=79998
13/11/27 12:04:17 INFO mapred.JobClient: Map output records=832280
13/11/27 12:04:17 INFO SortInMemoryProjectionDriver: jobStatus: true
13/11/27 12:04:17 INFO SortInMemoryProjectionDriver:
elapsedTime (in milliseconds): 62063
```

## 部分输出

```
...
ZTOBR28AH2 2013-01-06,190;2013-04-02,109;2013-04-09,26;...
```

```
ZV2A56WNI6 2013-01-22,51;2013-01-24,34;2013-02-09,52;...
ZXN7727FBA 2013-02-07,164;2013-02-23,30;2013-03-28,107;...
ZY44ATNBK7 2013-03-27,191;2013-04-27,51;2013-05-06,31;...
...
```

下一步是建立模型的转移概率，也就是要定义：

$$0.0 \leqslant P(state_1, state_2) \leqslant 1.0$$

在这里，$state_1$和$state_2 \in \{SL, SE, SG, ML, ME, MG, LL, LE, LG\}$。

## Hadoop解决方案2：时序交易

这个实现提供了一个解决方案，可以使用二次排序技术对归约器值排序（这是Hadoop解决方案1的一个替代方法；通过使用二次排序设计模式，不再需要将所有归约器值缓存在内存/RAM中来完成排序）。要做到这一点，我们需要一些定制类，并插入到MapReduce框架实现中。映射器发出键-值对，其中键是一个（customer-id, purchase-date）对，值是一个（purchase-date, amount）对。到达归约器的数据是有序的。可以看到，为了各个归约器键生成有序的值，我们在映射器键中加入了purchase-date（也就是说，发出的映射器值的第一部分）。所以，CompositeKey由（customer-id, purchase-date）对组成。值（purchase-date, amount）用类edu.umd.cloud9.io.pair.PairOfLongInt表示，在这里Long部分表示交易日期，Int表示交易额。

MapReduce框架指出，一旦数据值到达归约器，所有数据都按键分组。前面已经提到，我们有一个CompositeKey，所以要确保记录只按自然键（即customer-id）分组。为此我们编写了一个定制分区器类：NaturalKeyPartitioner。另外还需要提供其他一些插件类：

```
Configuration conf = new Configuration();
JobConf jobconf = new JobConf(conf, SecondarySortProjectionDriver.class);
...
jobconf.setPartitionerClass(NaturalKeyPartitioner.class);
jobconf.setOutputKeyComparatorClass(CompositeKeyComparator.class);
jobconf.setOutputValueGroupingComparator(NaturalKeyGroupingComparator.class);
```

现在来回顾第1章和第2章介绍的二次排序模式。映射器会生成键-值对。到达归约器的值的顺序未指定，运行不同作业期间可能会变化。例如，假设所有映射器生成了（K, V1），（K, V2）和（K, V3）。所以，对于键K，我们有3个值V1, V2, V3。处理键K的归约器可能得到以下的某个值（共6个顺序不同的值）：

```
V1, V2, V3
V1, V3, V2
V2, V1, V3
V2, V3, V1
V3, V1, V2
```

```
V3, V2, V1
```

大多数情况下（取决于你的需求以及如何处理归约器值），这个值以什么顺序到达可能并不重要。不过，如果希望你接收和处理的值以某种顺序排序（如升序或降序），首先想到的做法就是得到所有值V1, V2, V3，然后对它们应用一个排序函数来得到你希望的顺序。我们已经讨论过，如果你的服务器没有足够的RAM，不能存放所有值，那么这种排序技术可能就不可行。不过，还有另外一种更可取的方法，具有很好的可伸缩性，而不用担心“大RAM”需求：可以使用MapReduce框架的排序和洗牌特性。我们已经知道，这个技术称为二次排序，它允许MapReduce程序员控制`reduce()`函数调用中值出现的顺序。为了实现这一点，需要使用一个组合键，其中包含按键和按值排序所需的信息，然后解耦合中间数据的分组和排序。接下来，利用二次排序，可以定义映射器生成的值的排序顺序，再在键以及值上完成排序。更进一步，通过分组，可以确定哪些键-值对要放在一个`reduce()`函数调用中。对于这里的例子，组合键如图11-2所示。

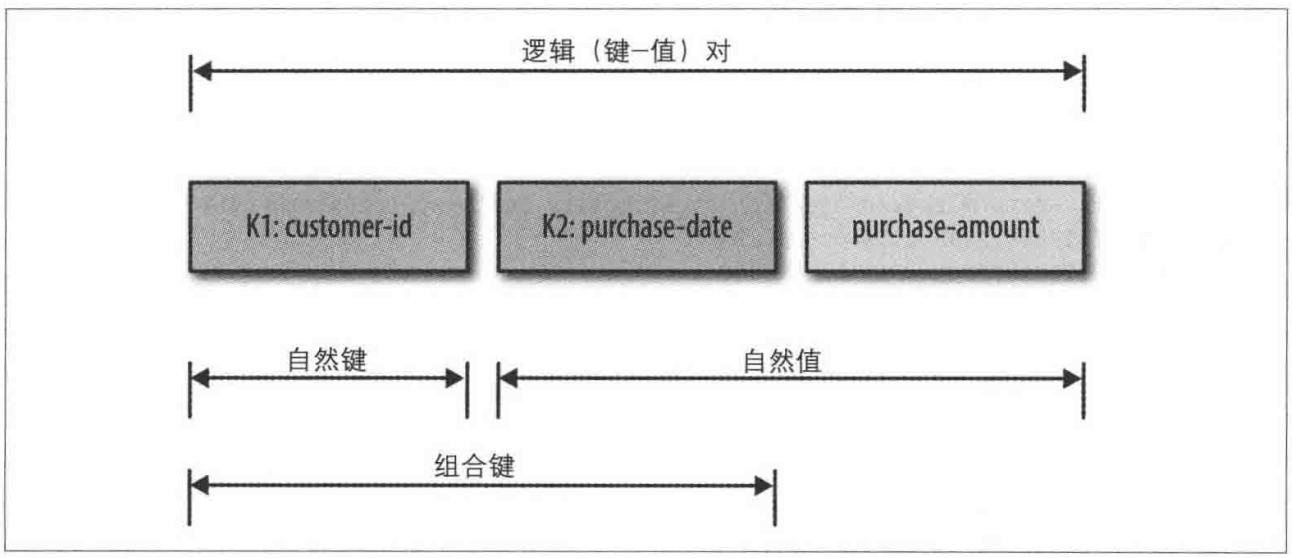



图11-2：二次排序的组合键

在Hadoop中，要在两个地方控制分组：分区器（将映射器输出发送到归约器任务），以及分组比较器（将数据分组到一个归约器任务）。分区器和分组比较器的功能都可以通过各个MapReduce作业相应的插件类来完成。要对归约器值完成排序，必须定义一个插件类设置输出键比较器。因此，如果使用传统MapReduce，实现二次排序可能需要做一点工作：必须定义一个组合键，并指定3个函数（为MapReduce作业定义插件类，详细信息见表11-3），这3个函数分别采用不同的方法使用这个组合键。需要说明，对于我们的例子，自然键就是`customer-id`（一个字符串对象），没有必要把它包装在可能名为自然键的另一个类中。通常会用自然键作为键或“分组”操作符。

表11-3：Hadoop解决方案2: 实现类

| 类名 | 描述 |
|---|---|
| SecondarySortProjectionDriver | 提交作业的驱动器类 |
| SecondarySortProjectionMapper | 映射器类 |
| SecondarySortProjectionReducer | 归约器类 |
| CompositeKey | 包含（customer-id, purchase-date）对的一个定制类，这是自然键和要排序的自然值的一个组合 |
| CompositeKeyComparator | 如何对CompositeKey对象排序，比较两个组合键来完成排序 |
| NaturalKeyGroupingComparator | 考虑自然键，确保一个归约器看到分组的一个定制视图（如何对customer-id分组） |
| NaturalKeyPartitioner | 如何为归约器按自然键（customer-id）分区，将所有数据划分到一个逻辑组中，我们希望其中对自然值完成二次排序 |
| DateUtil | 基本日期工具类 |
| HadoopUtil | 基本Hadoop工具类 |

## 部分输入

```
# hadoop fs -cat /markov/projection_by_secondary_sort/input/training.txt | head
V31E55G4FI,1381872898,2013-01-01,123
301UNH7I2F,1381872899,2013-01-01,148
PP2KVIR4LD,1381872900,2013-01-01,163
AC57MM3WNV,1381872901,2013-01-01,188
BN020INHUM,1381872902,2013-01-01,116
UP8R2SOR77,1381872903,2013-01-01,183
VD91210MGH,1381872904,2013-01-01,204
COI40XHET1,1381872905,2013-01-01,78
76S34ZE89C,1381872906,2013-01-01,105
6K3SNF2EG1,1381872907,2013-01-01,214
```

## 示例运行日志

下面给出日志输出，因篇幅所限，这里对日志做了编辑，并对格式有所调整：

```
# ./run.sh
JAVA_HOME=/System/Library/Java/JavaVirtualMachines/1.6.0.jdk/Contents/Home
...
Deleted hdfs://localhost:9000/lib/projection_by_secondary_sort.jar
Deleted hdfs://localhost:9000/markov/projection_by_secondary_sort/output
...
13/11/27 15:14:34 INFO mapred.FileInputFormat: Total input paths to process : 1
13/11/27 15:14:34 INFO mapred.JobClient: Running job: job_201311271459_0003
13/11/27 15:14:35 INFO mapred.JobClient: map 0% reduce 0%
...
13/11/27 15:16:02 INFO mapred.JobClient: map 100% reduce 100%
13/11/27 15:16:03 INFO mapred.JobClient: Job complete: job_201311271459_0003
...
```

```
13/11/27 15:16:03 INFO mapred.JobClient: Map-Reduce Framework
13/11/27 15:16:03 INFO mapred.JobClient: Map input records=832280
13/11/27 15:16:03 INFO mapred.JobClient: Combine input records=0
13/11/27 15:16:03 INFO mapred.JobClient: Reduce input records=832280
13/11/27 15:16:03 INFO mapred.JobClient: Reduce input groups=79998
13/11/27 15:16:03 INFO mapred.JobClient: Combine output records=0
13/11/27 15:16:03 INFO mapred.JobClient: Reduce output records=79998
13/11/27 15:16:03 INFO mapred.JobClient: Map output records=832280
13/11/27 15:16:03 INFO SecondarySortProjectionDriver: elapsedTime
  (in milliseconds): 89809
```

### 部分输出

```
...
ZSY4ONVPS6 2013-01-01,110;2013-01-11,32;2013-03-04,111;2013-04-09,65;...
ZTLNFOO4LN 2013-01-16,55;2013-03-21,164;2013-05-14,66;2013-06-29,81;...
ZV2OAIXG8L 2013-01-13,210;2013-02-03,32;2013-02-10,48;2013-02-23,27;...
...
```

## 生成状态序列

这一节的目标是将交易序列（transaction sequence）转换为一个状态序列（state sequence）。我们的两个Hadoop解决方案都会生成以下输出（也就是交易序列）：

$$\text{customer-id } (\text{Date}_1, \text{Amount}_1);(\text{Date}_2, \text{Amount}_2);...(\text{Date}_N, \text{Amount}_N)$$

使得：

$$\text{Date}_1 \leqslant \text{Date}_2 \leqslant ... \leqslant \text{Date}_N$$

我们要把这个输出（“交易序列”）转换为一个“状态序列”，如下所示：

$$\text{customer-id}, \text{State}_1, \text{State}_2, ..., \text{State}_n$$

下一个任务是（使用Pranab Ghosh开发的另一个Ruby脚本）将（purchase-date, amount）对的有序序列转换为一组马尔可夫链状态。状态由一个两字母代号表示，各个字母的定义如表11-4所示。

**表11-4：指示马尔可夫链状态的字母**

| 上一次交易后的经过时间 | 与前次交易相比的交易额 |
|---|---|
| S: 小 | L: 显著小于 |
| M: 中 | E: 基本相同 |
| L: 大 | G: 显著大于 |

因此，我们可以得到9个状态（3 × 3），如表11-5所示。

表11-5：两字母马尔可夫链状态名和定义

| 状态名 | 上一次交易后的经过时间:与前次交易相比的交易额 |
|---|---|
| SL | 小:显著小于 |
| SE | 小:基本相同 |
| SG | 小:显著大于 |
| ML | 中:显著小于 |
| ME | 中:基本相同 |
| MG | 中:显著大于 |
| LL | 大:显著小于 |
| LE | 大:基本相同 |
| LG | 大:显著大于 |

可以看到，我们的马尔可夫链模型有9个状态（9 × 9转移矩阵）。

下面的一组shell命令展示了如何使用Ruby脚本*xaction_state.rb*（*http://bit.ly/xaction_state*）由交易序列生成状态序列：

```
$ cat transaction_sequence.txt
00VVD1E210,2012-06-18,87
00W6TWFW4S,2012-03-24,22,2012-05-22,80,2012-06-15,33
00W86YOGFT,2012-02-15,141,2012-03-10,30,2012-03-25,49,2012-05-17,107
00W92K8A1W,2012-04-19,25
00W9W3Y3XH,2012-03-25,123
00XL1QERUO,2012-01-07,81,2012-05-10,154
00XPR1XW1P,2012-04-26,103
00Y1BOY4CO,2012-03-10,81
00YR97DWWO,2012-07-15,118
00Z5SOHKED,2012-01-28,43,2012-02-25,27
00ZLLMHKND,2012-02-21,185,2012-04-02,63,2012-04-03,30

$ ./xaction_state.rb transaction_sequence.txt
00W6TWFW4S,ML,SG
00W86YOGFT,SG,SL,ML
00XL1QERUO,LL
00Z5SOHKED,SG
00ZLLMHKND,MG,SG
```

对于类似这样的少量数据，这个Ruby脚本能很好地完成工作。不过，对于大量的数据，可能就需要使用一个映射器程序，如示例11-3所示。

示例11-3：生成状态序列的映射器

```
/**
 * @param key由MapReduce生成，在这里忽略
 * @param value为：
 * <customerID><,><Date1><,><Amount1><,><Date2><,><Amount2>...
 * 这里 Date1 <= Date2 <= ...
```

```
 */
map(key, value) {
   tokens[] = value.split(",");
   if (tokens.length < 5) {
        // 没有足够的数据
        return;
   }
   customerID = tokens[0];
   sequence = [];
   i = 4;
   while (i < tokens.length) {
     amount = tokens[i];
     priorAmount = tokens[i-2];
     date = tokens[i-1];
     priorDate = tokens[i-3];
     daysDiff = date - priorDate;
     amountDiff = amount - priorAmount;

     if (daysDiff < 30) {
        dd = "S";
     }
     elsif (daysDiff < 60) {
        dd = "M";
     }
     else {
        dd = "L";
     }

     if (priorAmount < 0.9 * amount) {
        ad = "L";
     }
     elsif (priorAmount < 1.1 * amount) {
        ad = "E";
     }
     else {
        ad = "G";
     }

     sequence << (dd + ad);
     i += 2;
   }
   emit (customerID, ",", sequence.join(","));
}
```

## 使用MapReduce生成马尔可夫状态转移矩阵

这个MapReduce阶段的目标是生成一个马尔可夫状态转移矩阵。这个阶段的输入是状态序列，格式如下：

$$\text{customer-id}, \text{State}_1, \text{State}_2, \ldots, \text{State}_n$$

输出是一个$N \times N$的矩阵，这里$N$是马尔可夫链模型的状态数（在我们的模型中，$N$=

9）。这个矩阵中的各项指示了从一个状态转移到另一个状态的概率。这个MapReduce阶段将统计状态转移的实例数。由于我们的模型中状态数为9，所以会有9 × 9=81 个可能的状态转移。

示例11-4定义了马尔可夫状态转移的map()函数。

示例11-4：马尔可夫状态转移：map()函数

```
/**
 * @param key为Customer-ID, 忽略
 * @param value为状态序列= {S1, S2, ..., Sn}
 * 假设值是一个包含n个状态的数组（索引从0到n-1）。
 */
map(key, value) {
    for (i=0, i < n-1, i++) {
       // value[i] 指示"从状态"
       // value[i+1] 指示"到状态"
       reducerKey = pair(value[i], value[i+1]);
       emit(reducerKey, 1);
    }
}
```

示例11-5定义了马尔可夫状态转移的combine()函数。

示例11-5：马尔可夫状态转移：combine()函数

```
/**
 * @param key为Pair(state1, state2)
 * @param value为整数列表（"state1"到"state2"的部分计数）
 */
combine(Pair(state1, state2) key, List<integer> values) {
    int partialSum = 0;
    for (int count : values) {
        partialSum += count;
    }
    emit(key, partialSum);
}
```

示例11-定义了马尔可夫状态转移的reduce()函数。

示例11-6：马尔可夫状态转移：reduce()函数

```
/**
 * @param key为Pair(state1, state2)
 * @param value为整数列表（"state1"到"state2"的部分计数）
 */
reduce(Pair(state1, state2) key, List<integer> value) {
    int sum = 0;
    for (int count : value) {
        sum += count;
    }
    emit(key, sum);
}
```

## 状态转移模型的Hadoop实现

表11-6展示了状态转移模型Hadoop实现所需的类。

**表11-6：状态转移模型：实现类**

| 类名 | 描述 |
|---|---|
| MarkovStateTransitionModelDriver | 提交作业的驱动器类 |
| MarkovStateTransitionModelMapper | 映射器类 |
| MarkovStateTransitionModelCombiner | 组合器类 |
| MarkovStateTransitionModelReducer | 归约器类 |
| ReadDataFromHDFS | 从HDFS读取数据，创建List<TableItem> |
| StateTransitionTableBuilder | 建立转移表并定义P(state1, state2) |
| TableItem | 表示（fromState, toState, count）三元组 |
| HadoopUtil | 基本Hadoop工具类 |

## 输入

```
# hadoop fs -cat /markov/state_transition_model/input/state_seq.txt | head
000IA1PHVZ,SG,SL,SG,SL,ML,MG,SG,SL,SG,SL,ML
000KH3DK15,SG,SL,SG,ML,SG,SL,SG,SL,SG,SL,SG,ML,SG,SL,SG
001KD25DTD,SG,SL,SG,SL,SG,SL,SG
00241F24T4,SG,SL,SG,SL,SG,SL,SG,ML,SG,SL,ML,SG,ML,SG
002C11GB8Y,SG,SL,SG,SL,SG,SL,SG,ML,SG,SL,SG,ML,SG
002SG5SKJT,SG,SL,SG,ML,SG,SL,SG
0030B44HD0,SG,SL,SG,SL,SG,SL,SG,SL,ML,SG
004ADRKOEW,SG,SL,SG,ML,MG,SG,SL,SG,LL
004MT1M5BY,SG,SL,SG,SL,SG,ML,SG,ML,SG,SL,ML
007DI3WJ5B,SL,SL,ML,MG,SG,SE,SL,SG,SG,SL,SG
```

## 示例运行日志

下面给出日志输出的一个缩减版本：

```
# ./run.sh
...
adding: HadoopUtil.class(in = 1797) (out= 840)(deflated 53%)
...
adding: TableItem.class(in = 375) (out= 263)(deflated 29%)
...
13/11/29 21:16:17 INFO mapred.JobClient: Running job: job_201311291911_0003
13/11/29 21:16:18 INFO mapred.JobClient: map 0% reduce 0%
...
13/11/29 21:17:10 INFO mapred.JobClient: map 100% reduce 100%
13/11/29 21:17:11 INFO mapred.JobClient: Job complete: job_201311291911_0003
...
13/11/29 21:17:11 INFO mapred.JobClient: Map-Reduce Framework
13/11/29 21:17:11 INFO mapred.JobClient: Map input records=79977
13/11/29 21:17:11 INFO mapred.JobClient: Reduce shuffle bytes=844
```

```
...
13/11/29 21:17:11 INFO mapred.JobClient: Combine input records=672461
13/11/29 21:17:11 INFO mapred.JobClient: Reduce input records=56
13/11/29 21:17:11 INFO mapred.JobClient: Reduce input groups=56
13/11/29 21:17:11 INFO mapred.JobClient: Combine output records=212
13/11/29 21:17:11 INFO mapred.JobClient: Reduce output records=56
13/11/29 21:17:11 INFO mapred.JobClient: Map output records=672305
```

### 输出

```
# hadoop fs -ls /markov/state_transition_model/output/part*
-rw-r--r-- ... 58 2013-11-29 21:16 /markov/state_transition_model/output/
  part-r-00000
...
-rw-r--r-- ... 46 2013-11-29 21:17 /markov/state_transition_model/output/
  part-r-00009
# hadoop fs -cat /markov/state_transition_model/output/part*
LL,MG 2990
ME,SG 172
MG,LL 803
...
SL,SE 2099
LE,LG 2
LG,LE 1
MG,SG 19485
ML,SL 268
...
SL,ME 151
LG,SL 510
LL,SG 17062
...
SG,SG 5090
SL,SL 2772
...
SG,LE 140
# javac ReadDataFromHDFS.java TableItem.java
# java ReadDataFromHDFS /markov/state_transition_model/output
Nov 30 2013 12:24:18 [main] [INFO ] [ReadDataFromHDFS] - path=...
Nov 30 2013 12:24:18 [main] [INFO ] [ReadDataFromHDFS] - line=ME,SG 172
Nov 30 2013 12:24:18 [main] [INFO ] [ReadDataFromHDFS] - line=MG,LL 803
...
Nov 30 2013 12:24:18 [main] [INFO ] [ReadDataFromHDFS] - line=LL,LG 507
Nov 30 2013 12:24:18 [main] [INFO ] [ReadDataFromHDFS] - line=SE,LG 2
Nov 30 2013 12:24:18 [main] [INFO ] [ReadDataFromHDFS] - line=SG,LE 140
Nov 30 2013 12:27:45 [main] [INFO ] [ReadDataFromHDFS] -
  list=[{LL,MG,2990}, ..., {MG,LL,803}]
```

## 使用马尔可夫模型预测下一个智能邮件营销日期

既然已经由给定的训练数据建立了马尔可夫模型，可以使用以下代码持久保存这个模型：

```
# cat StateTransitionTableBuilder.java
```

```
...
public class StateTransitionTableBuilder {
    ...
   public static void main(String[] args) {
      String hdfsDirectory = args[0];
      generateStateTransitionTable(hdfsDirectory);
   }
}

# export hdfsDir="/markov/state_transition_model/output"
# java StateTransitionTableBuilder $hdfsDir > model.txt
```

我们已经有了model.txt(表示为一个二维表)，下面可以使用Ruby脚本*buy_xaction.rb*和*mark_plan.rb*（*http://bit.ly/mark_plan*）（由Pranab Ghosh开发）来预测下一个智能邮件营销日期：

```
# Generate validation data
# ------------------------
./buy_xaction.rb 80000 30 .05 > validation.txt
head validation.txt
XURQDBEHME,1385141945,2013-01-01,98
3RT4PONSUP,1385141946,2013-01-01,53
4NYCEUD3YG,1385141947,2013-01-01,164
SF9KAY8F42,1385141948,2013-01-01,204
LKNCID1DRV,1385141949,2013-01-01,83
4EZJDVB4W1,1385141950,2013-01-01,116
ITJ39B3NX3,1385141951,2013-01-01,72
D8VVPAHG8I,1385141952,2013-01-01,124
21XHZJY561,1385141953,2013-01-01,103
F7LS37RO8X,1385141954,2013-01-01,211
# Predict email marketing time
# ----------------------------
./mark_plan.rb validation.txt model.txt
XURQDBEHME, 2013-04-27
4NYCEUD3YG, 2013-04-14
SF9KAY8F42, 2013-04-07
LKNCID1DRV, 2013-04-30
4EZJDVB4W1, 2013-02-02
ITJ39B3NX3, 2013-04-27
D8VVPAHG8I, 2013-04-29
21XHZJY561, 2013-01-18
F7LS37RO8X, 2013-02-14
...
```

现在就能根据用户的交易历史预测下一个发送智能营销邮件的日期了。

# Spark解决方案

在这一节中，我会提供一个使用马尔可夫链算法实现智能邮件营销的Spark解决方案。我们的Spark解决方案包含一组变换，如mapToPair()、groupByKey()、reduceByKey()和

mapValues()。首先会给出这个解决方案的一组高层步骤，然后会详细介绍各个步骤，并提供相应的Spark变换或动作。最后还会提供一个运行示例和相应的脚本。

## 输入格式

每个输入记录包括以下4个字段：

```
<customerID><,><transactionID><,><purchaseDate><,><amount>
```

下面给出一些示例输入数据：

```
$ ls -l smart_email_training.txt
-rw-r--r-- 1 hadoop hadoop 30089506 Nov 5 16:57 smart_email_training.txt
$ wc -l smart_email_training.txt
832280 smart_email_training.txt
$ hadoop fs -copyFromLocal smart_email_training.txt /home/hadoop/testspark/
$ hadoop fs -cat /home/hadoop/testspark/smart_email_training.txt | head
V31E55G4FI,1381872898,2013-01-01,123
301UNH7I2F,1381872899,2013-01-01,148
PP2KVIR4LD,1381872900,2013-01-01,163
AC57MM3WNV,1381872901,2013-01-01,188
BNO20INHUM,1381872902,2013-01-01,116
UP8R2SOR77,1381872903,2013-01-01,183
VD91210MGH,1381872904,2013-01-01,204
COI4OXHET1,1381872905,2013-01-01,78
76S34ZE89C,1381872906,2013-01-01,105
6K3SNF2EG1,1381872907,2013-01-01,214
```

## 高层步骤

我们的Spark解决方案表示为7个基本步骤：

1. 处理输入参数。这个程序中只有一个参数。这一步会读取交易数据的输入路径，接下来将分析和处理这些交易数据。
2. 创建一个上下文对象，把输入转换为一个JavaRDD<String>，其中各个元素分别是一个输入记录（见上一小节中的定义）。
3. 将JavaRDD<String>转换为一个JavaPairRDD<K,V>，其中K是一个customerID，V是一个Tuple2<purchaseDate, amount>。

   需要说明，这里忽略了transactionID字段（因为后面的处理不需要这个字段）。
4. 按customerID对交易分组。我们将对步骤3的输出应用groupByKey()，结果是一个JavaPairRDD<K2,V2>，其中K2是一个customerID，V2是一个Iterable<Tuple2-<purchaseDate, Amount>>。
5. 创建一个马尔可夫状态序列：

$State_1$, $State_2$, ..., $State_N$

对JavaPairRDD<K2,V2>应用mapValues()变换，生成一个JavaPairRDD<K4, V4>。首先将（K2, V2）对转换为（K3, V3）对，这里K2 = K3 = K4 = customerID，V3是排序后的V2（按purchaseDate有序），也就是说，这是一个有序的交易序列。然后，使用V3创建一个马尔可夫状态序列（V4）。

6. 生成马尔可夫状态转移，输入/输出如下：

   *输入*

   JavaPairRDD<K4, V4> 对。

   *输出*

   一个状态矩阵 {S1, S2, S3, ...}（见表11-7），定义了从一个状态转移到另一个状态的概率。建立这个矩阵后，我们可以使用新数据来预测下一个智能营销日期。

   在下面的表中，$P_{ij}$是从状态$S_i$到状态$S_j$的概率。

   **表11-7：马尔可夫状态转移概率**

| States | $S_1$ | $S_2$ | $S_3$ | ... |
|---|---|---|---|---|
| $S_1$ | $P_{11}$ | $P_{12}$ | $P_{13}$ | ... |
| $S_2$ | $P_{21}$ | $P_{22}$ | $P_{23}$ | ... |
| $S_3$ | $P_{31}$ | $P_{32}$ | $P_{33}$ | ... |
| ... | ... | ... | ... | ... |

7. 发出最终输出，定义从一个状态转移到另一个状态的概率。

## Spark程序

整个Spark解决方案由一个Java驱动器类提供，如示例11-7所示。

示例11-7：Spark程序的结构

```
1 package org.dataalgorithms.chap11.spark;
2 // 步骤0: 导入所需的类和接口
3 //源代码参考:
4 //    https://github.com/mahmoudparsian/data-algorithms-book/
5 //    .../src/main/java/org/dataalgorithms/chap11/spark/SparkMarkov.java
6 public class SparkMarkov implements Serializable {
7
8   static List<Tuple2<Long,Integer>> toList(Iterable<Tuple2<Long,Integer>>
9 iterable) {...}
10   static List<String> toStateSequence(List<Tuple2<Long,Integer>> list) {...}
11   static class TupleComparatorAscending implements
12       Comparator<Tuple2<Long, Integer>>, Serializable {...}
13
```

```
14    public static void main(String[] args) throws Exception {
15       // 步骤1: 处理输入参数
16       // 步骤2: 创建上下文对象 (ctx) 并把输入转换为
17       //        JavaRDD<String>, 其中各个元素分别是一个输入记录
18       // 步骤3: 将RDD<String>转换为JavaPairRDD<K,V>, 其中
19       //        K: customerID
20       //        V: Tuple2<purchaseDate, Amount> : Tuple2<Long, Integer>
21       // 步骤4: 按customerID对交易分组: 应用groupByKey()
22       // 步骤5: 创建马尔可夫状态序列: State1, State2, ..., StateN
23       // 步骤6: 生成马尔可夫状态转移矩阵
24       // 步骤7: 发出最终输出
25
26       // 完成
27       ctx.close();
28       System.exit(0);
29    }
30 }
```

## 步骤1：处理输入参数

这个步骤如示例11-8所示，从一个shell脚本读取交易数据的输入路径。这个输入路径可能是一个Linux路径或HDFS路径（Linux和HDFS文件Spark都可以读取）。

示例11-8：步骤1: 处理输入参数

```
1 // 步骤1: 处理输入参数
2 if (args.length != 1) {
3     System.err.println("Usage: SparkMarkov <input-path>");
4     System.exit(1);
5 }
6 final String inputPath = args[0];
7 System.out.println("inputPath:args[0]="+args[0]);
```

## 步骤2：创建Spark上下文对象，并把输入转换为RDD

这个步骤如示例11-9所示，读取输入文件，并为这个程序创建第一个RDD（使用工厂类JavaSparkContext）。

示例11-9：步骤2: 创建上下文对象，并把输入转换为RDD

```
1   // 步骤2: 创建Spark上下文对象并把输入转换为
2   //        JavaRDD<String>,
3   // 其中各个元素分别是一个输入记录
4   JavaSparkContext ctx = new JavaSparkContext();
5   JavaRDD<String> records = ctx.textFile(inputPath, 1);
6   records.saveAsTextFile("/output/2");
7
8   // 可以对RDD分区
9   // public JavaRDD<T> coalesce(int N)
10  // 返回一个新RDD，归约到N个分区。
11  // JavaRDD<String> records = ctx.textFile(inputPath, 1).coalesce(9);
```

我们将使用以下RDD元素来调试这个步骤：

```
$ hadoop fs -cat /output/2/part* | head -3
V31E55G4FI,1381872898,2013-01-01,123
301UNH7I2F,1381872899,2013-01-01,148
PP2KVIR4LD,1381872900,2013-01-01,163
```

## 步骤3：将RDD转换为JavaPairRDD

如示例11-10所示，这一步将各个记录分别转换为一个JavaPairRDD<K,V>，其中K是一个customerID，V是一个Tuple2<purchaseDate,Amount> （purchaseDate转换为一个Long数据类型，用毫秒表示日期）。

示例11-10：步骤3: RDD转换为JavaPairRDD

```
// 步骤3: JavaRDD<String>转换为JavaPairRDD<K,V>，其中
//   K: customerID
//   V: Tuple2<purchaseDate, Amount> : Tuple2<Long, Integer>
//   PairFunction<T, K, V>
//   T => Tuple2<K, V>
JavaPairRDD<String, Tuple2<Long,Integer>> kv = records.mapToPair(
    new PairFunction<
                    String,              // T
                    String,              // K
                    Tuple2<Long,Integer> // V
                   >() {
    public Tuple2<String,Tuple2<Long,Integer>> call(String rec) {
        String[] tokens = StringUtils.split(rec, ",");
        if (tokens.length != 4) {
        // 不是正确的格式
        return null;
        }
        // tokens[0] = customer-id
        // tokens[1] = transaction-id
        // tokens[2] = purchase-date
        // tokens[3] = amount
        long date = 0;
        try {
            date = DateUtil.getDateAsMilliSeconds(tokens[2]);
        }
        catch(Exception e) {
            // 暂时忽略—必须处理
        }
        int amount = Integer.parseInt(tokens[3]);
        Tuple2<Long,Integer> V = new Tuple2<Long,Integer>(date, amount);
        return new Tuple2<String,Tuple2<Long,Integer>>(tokens[0], V);
    }
});
kv.saveAsTextFile("/output/3");
```

我们将使用以下RDD元素来调试这个步骤：

```
$ hadoop fs -cat /output/3/part* | head
(V31E55G4FI,(1357027200000,123))
(301UNH7I2F,(1357027200000,148))
```

```
(PP2KVIR4LD,(1357027200000,163))
(AC57MM3WNV,(1357027200000,188))
(BNO20INHUM,(1357027200000,116))
(UP8R2SOR77,(1357027200000,183))
(VD91210MGH,(1357027200000,204))
(COI4OXHET1,(1357027200000,78))
(76S34ZE89C,(1357027200000,105))
(6K3SNF2EG1,(1357027200000,214))
```

## 步骤4：按customerID对交易分组

这个步骤如示例11-11所示，使用groupByKey()按customerID对所有交易分组。

示例11-11：步骤4: 按customerID对交易分组

```
// 步骤4: 按customerID对交易分组。对步骤2的输出应用groupByKey()；
// 结果将是
// JavaPairRDD<K2,V2>，其中
// K2: customerID
// V2: Iterable<Tuple2<purchaseDate, Amount>>
JavaPairRDD<String, Iterable<Tuple2<Long,Integer>>> customerRDD =
    kv.groupByKey();
customerRDD.saveAsTextFile("/output/4");
```

我们将使用以下RDD元素来调试这个步骤：

```
$ hadoop fs -cat /output/4/part* | head -3
(OIROUCA5O2,[(1361347200000,86), (1362643200000,30), (1362816000000,45),
(1364886000000,27), (1366009200000,40), (1366182000000,28),
(1369724400000,115), (1370502000000,32), (1371970800000,42),
(1372575600000,32), (1374649200000,43)])
(4NOB1U5HVG,[(1358668800000,81), (1359446400000,33), (1363071600000,98),
(1365750000000,50), (1366614000000,29), (1367218800000,48),
(1369378800000,30), (1369810800000,41), (1370674800000,28),
(1373353200000,107)])
(3KJR1907D9,[(1361088000000,105), (1362211200000,26), (1366182000000,103),
(1366182000000,28), (1370415600000,111), (1373266800000,61),
(1373439600000,34)])
```

## 步骤5：创建马尔可夫状态序列

如示例11-12所示，这一步将交易序列转换为状态序列。首先，按交易日期对交易序列排序，然后将这些有序的交易序列转换为状态序列。

示例11-12：步骤5: 创建马尔可夫状态序列

```
// 步骤5: 创建马尔可夫状态序列：State1, State2, ..., StateN。
//        对JavaPairRDD<K2,V2>应用mapValues()，生成JavaPairRDD<K4, V4>。
//        首先将(K2, V2)转换为(K3, V3)对[K2 = K3 = K4]。
//        V3是有序的V2（按purchaseDate有序）；
//        也就是说，这是一个有序的交易序列。
//        然后使用V3创建马尔可夫状态序列（ V4）。
// mapValues[U](f: (V) => U): JavaPairRDD[K, U]
// 通过一个映射函数传递键-值对RDD中的各个值，而不
```

```
 9    // 改变键；还会保留原来的RDD分区。
10    JavaPairRDD<String, List<String>> stateSequence = customerRDD.mapValues(
11        new Function<
12                    Iterable<Tuple2<Long,Integer>>, // 输入
13                    List<String>                    // 输出 ("状态序列")
14                   >() {
15        public List<String> call(Iterable<Tuple2<Long,Integer>> dateAndAmount) {
16            List<Tuple2<Long,Integer>> list = toList(dateAndAmount);
17            Collections.sort(list, TupleComparatorAscending.INSTANCE);
18            // 现在将有序列表（按日期）转换为一个状态序列
19            List<String> stateSequence = toStateSequence(list);
20            return stateSequence;
21        }
22    });
23    stateSequence.saveAsTextFile("/output/5");
```

我们将使用以下RDD元素来调试这个步骤：

```
$ hadoop fs -cat /output/5/part* | head
(OIROUCA5O2,[SG, SL, SG, SL, SG, ML, SG, SL, SG, SL])
(4NOB1U5HVG,[SG, ML, MG, SG, SL, SG, SL, SG, ML])
(3KJR1907D9,[SG, ML, SG, ML, MG, SG])
(8555DQOK14,[SG, ML, LL])
(J6VXOTY7IA,[SG, ML, SG, SL, SG, ML, SG])
(T29MOVFTO4,[SG, SL, SG, SL, ML, SG, SL, SG, SL, SG, SL, SG, SL, SG])
(JOBO64O93C,[SG, SL, SG, SL, ML, SG, SG, SL, SG, SL, SG, SL, ML, SG, SL])
(NT58RT7KK4,[MG, SG, SL, SG, SL, SG, SL, SG, SL, SG])
(HBD6YAC69Y,[SG, SL, SG, SL, SG, SL, SG, SL, ML, MG])
(1BNFI5D3Z1,[SG, SL, SG, SL, SG, SL, SG, SL])
```

## 步骤6：生成马尔可夫状态转移矩阵

这一步是生成马尔可夫状态转移表的映射器阶段。如果我们的训练数据集中有一个从fromState到toState的状态转移，则发出（K，V）对，其中K是一个Tuple2(fromState, toState)，V为1。

示例11-13：步骤6：生成马尔可夫状态转移矩阵

```
 1    // 步骤6: 生成马尔可夫状态转移矩阵
 2    //         输入是JavaPairRDD<K4, V4>对
 3    //           其中 K4 = customerID, V4 = List<State>
 4    //         输出是一个状态矩阵{S1, S2, S3, ...}
 5    //
 6    //            | S1 S2 S3 ...
 7    //         ---+----------------------
 8    //         S1 | <probability-value>
 9    //            |
10    //         S2 |
11    //            |
12    //         S3 |
13    //            |
14    //         ...|
15    //
```

```
//          这会定义从一个状态到另一个状态的概率。
//          建立这个矩阵后，可以使用
//          这个新数据预测下一个智能营销日期。
// 这一步的实现中，我们使用了：
//     PairFlatMapFunction<T, K, V>
//     T => Iterable<Tuple2<K, V>>
JavaPairRDD<Tuple2<String,String>, Integer> model = stateSequence.flatMapToPair(
    new PairFlatMapFunction<
                            Tuple2<String, List<String>>, // T
                            Tuple2<String,String>,        // K
                            Integer                       // V
                           >() {
    public Iterable<Tuple2<Tuple2<String,String>, Integer>>
        call(Tuple2<String, List<String>> s) {
        List<String> states = s._2;
        if ( (states == null) || (states.size() < 2) ) {
            return Collections.emptyList();
        }

        List<Tuple2<Tuple2<String,String>, Integer>> mapperOutput =
            new ArrayList<Tuple2<Tuple2<String,String>, Integer>>();
        for (int i = 0; i < (states.size() -1); i++) {
            String fromState = states.get(i);
            String toState = states.get(i+1);
            Tuple2<String,String> k = new Tuple2<String,String>(fromState,
                                                                toState);
            mapperOutput.add(new Tuple2<Tuple2<String,String>, Integer>(k, 1));
        }
        return mapperOutput;
    }
});
model.saveAsTextFile("/output/6.1");
```

我们将使用以下RDD元素来调试这个步骤：

```
$ hadoop fs -cat /output/6.1/part* | head
((SG,SL),1)
((SL,SG),1)
((SG,SL),1)
((SL,SG),1)
((SG,ML),1)
((ML,SG),1)
((SG,SL),1)
((SL,SG),1)
((SG,SL),1)
((SG,ML),1)
```

**组合/归约（fromState, toState）的频度。**reduceByKey()变换用来组合/归约从一个状态（fromState）到另一个状态（toState）的频度，如示例11-14所示。

示例11-14：组合/归约（fromState, toState）的频度

```
// 组合/归约频度模式（fromState, toState）
JavaPairRDD<Tuple2<String,String>, Integer> markovModel =
```

```
3       model.reduceByKey(new Function2<Integer, Integer, Integer>() {
4       public Integer call(Integer i1, Integer i2) {
5           return i1 + i2;
6       }
7   });
8   markovModel.saveAsTextFile("/output/6.2");
```

我们将使用以下RDD元素来调试这个步骤：

```
$ hadoop fs -cat /output/6.2/part*
((SL,LL),7890)
((SG,LL),11140)
...
((MG,SG),19769)
((LL,MG),2885)
...
((SG,SL),254532)
((SG,ML),50112)
...
((ML,LL),2450)
((ML,SG),66275)
```

输出：

```
((fromState, toState), count)
```

将用来规范化数据，生成P(fromState, toState)（即马尔可夫概率表）。这是通过StateTransitionTableBuilder类完成的，稍后会介绍。

## 步骤7：发出最终输出

在这个步骤中（参见示例11-15），我们将发出最终的输出，格式为：

```
<fromState><,><toState><TAB><frequency-count>
```

示例11-15：步骤7: 发出最终输出

```
1   // 步骤7: 发出最终输出
2   // 将markovModel转换为"<fromState><,><toState><TAB><count>"
3   // 使用map()将JavaPairRDD转换为JavaRDD:
4   // <R> JavaRDD<R> map(Function<T,R> f)
5   // 通过对这个RDD的所有元素应用一个函数返回一个新的RDD。
6   JavaRDD<String> markovModelFormatted = markovModel.map(
7       new Function<Tuple2<Tuple2<String,String>, Integer>, String>() {
8       public String call(Tuple2<Tuple2<String,String>, Integer> t) {
9           return t._1._1 + "," + t._1._2 + "\t" + t._2;
10      }
11   });
12   markovModelFormatted.saveAsTextFile("/output/6.3");
```

我们将使用以下RDD元素来调试这个步骤。因篇幅所限，下面的输出有所修改。这个输

出显示了（(fromState, toState), frequency），其中frequency指示状态从fromState转移到toState的次数：

```
$ export hdfsDir=/output/6.3
$ java org.dataalgorithms.chap11.statemodel.ReadDataFromHDFS $hdfsDir
INFO : path=hdfs://hnode01319.nextbiosystem.net:8020/output/6.3/part-00000
INFO : line=SL,LL 7890
INFO : line=SL,MG 209
INFO : line=SG,LL 11140
...
INFO : line=ML,LL 2450
INFO : line=ML,SG 66275
INFO : list=[{SL,LL,7890},
             {SL,MG,209},
             {SG,LL,11140},

             ...,
             {ML,LL,2450},
             {ML,SG,66275}]
```

## 生成马尔可夫概率模型

StateTransitionTableBuilder类将规范化数据，生成一个马尔可夫模型（定义从一个状态转移到另一个状态的概率）。由于我们有9个状态，所以生成的马尔可夫概率模型将是一个9×9的矩阵：

```
$ export hdfsDir=/output/6.3
$ export prog=org.dataalgorithms.chap11.statemodel.StateTransitionTableBuilder
$ java $prog $hdfsDir > model.txt
$ cat model.txt
0.001882,4.376e-06,0.8060,0.1567,4.376e-06,0.0009190,0.03453,4.376e-06,4.376e-06
0.1000,0.1000,0.2000,0.1000,0.1000,0.1000,0.1000,0.1000,0.1000
0.8056,3.165e-06,0.0004653,0.1586,3.165e-06,3.165e-06,0.03526,3.165e-06,3.165e-06
1.234e-05,1.234e-05,0.8178,0.0004812,1.234e-05,0.1511,0.03024,1.234e-05,0.0003208
0.01075,0.01075,0.7419,0.01075,0.01075,0.1290,0.06452,0.01075,0.01075
0.003234,4.146e-05,0.8197,0.1444,4.146e-05,0.0004561,0.03205,4.146e-05,4.146e-05
4.828e-05,4.828e-05,0.8370,4.828e-05,4.828e-05,0.1393,4.828e-05,9.655e-05,0.02332
0.01852,0.01852,0.6852,0.01852,0.01852,0.1481,0.01852,0.01852,0.05556
0.0005238,0.0005238,0.7973,0.001048,0.0005238,0.1650,0.03353,0.001048,0.0005238
```

## 辅助方法

toList()方法如示例11-16所示，将一个Iterable<Tuple2<Long,Integer>>转换为List<Tuple2<Long,Integer>>。

示例11-16：toList()方法

```
1 static List<Tuple2<Long,Integer>> toList(Iterable<Tuple2<Long,Integer>> iterable) {
2   List<Tuple2<Long,Integer>> list = new ArrayList<Tuple2<Long,Integer>>();
3   for (Tuple2<Long,Integer> element: iterable) {
4       list.add(element);
5   }
```

```
6   return list;
7 }
```

toStateSequence()方法如示例11-17所示，将一个有序的交易序列（List<Tuple2<Date,Amount>>）转换为一个状态序列（List<String>），其中各个元素分别表示一个马尔可夫状态。

示例11-17：toStateSequence()方法

```
1 /**
2  * @param list : List<Tuple2<Date,Amount>>
3  * list = [T2(Date1,Amount1), T2(Date2,Amount2), ..., T2(DateN,AmountN)
4  * 这里 Date1 <= Date2 <= ... <= DateN
5  */
6 static List<String> toStateSequence(List<Tuple2<Long,Integer>> list) {
7    if (list.size() < 2) {
8      // 没有足够的数据
9      return null;
10   }
11   List<String> stateSequence = new ArrayList<String>();
12   Tuple2<Long,Integer> prior = list.get(0);
13   for (int i = 1; i < list.size(); i++) {
14      Tuple2<Long,Integer> current = list.get(i);
15
16      long priorDate = prior._1;
17      long date = current._1;
18      // 1天= 24*60*60*1000 = 86400000 毫秒
19      long daysDiff = (date - priorDate) / 86400000;
20
21      int priorAmount = prior._2;
22      int amount = current._2;
23      int amountDiff = amount - priorAmount;
24
25      String dd = null;
26      if (daysDiff < 30) {
27         dd = "S";
28      }
29      else if (daysDiff < 60) {
30         dd = "M";
31      }
32      else {
33         dd = "L";
34      }
35
36      String ad = null;
37      if (priorAmount < 0.9 * amount) {
38         ad = "L";
39      }
40      else if (priorAmount < 1.1 * amount) {
41         ad = "E";
42      }
43      else {
44        ad = "G";
45      }
```

```
46
47        String element = dd + ad;
48        stateSequence.add(element);
49        prior = current;
50     }
51     return stateSequence;
52 }
```

### 比较器类

Comparator类如示例11-18所示，这是一个插件类，用来对List<Tuple2<Long, Integer>>排序。排序在“日期”字段上完成，日期用一个Long数据类型表示。

示例11-18：比较器类

```
1 static class TupleComparatorAscending implements
2     Comparator<Tuple2<Long, Integer>>, Serializable {
3     final static TupleComparatorAscending INSTANCE = new TupleComparatorAscending();
4     public int compare(Tuple2<Long, Integer> t1, Tuple2<Long, Integer> t2) {
5          // return -t1._1.compareTo(t2._1);      // 对RDD元素降序排序
6          return t1._1.compareTo(t2._1);          // 对RDD元素升序排序
7     }
8 }
```

## 运行Spark程序的脚本

下面的shell脚本将把我们的Spark程序提交给一个由3个节点组成的Spark集群：

```
$ cat run_build_markov_model.sh
#!/bin/bash
export JAVA_HOME=/usr/java/jdk7
export SPARK_HOME=/usr/local/spark-1.1.0
export SPARK_MASTER=spark://myserver100:7077
export BOOK_HOME=/mp/data-algorithms-book
export APP_JAR=$BOOK_HOME/dist/data_algorithms_book.jar
INPUT=/home/hadoop/testspark/smart_email_training.txt
# 在Spark独立集群上运行
prog=org.dataalgorithms.chap11.spark.SparkMarkov
$SPARK_HOME/bin/spark-submit\
--class $prog \
--master $SPARK_MASTER \
--executor-memory 2G \
--total-executor-cores 20 \
$APP_JAR \
$INPUT
```

## 运行示例

在前面提到的包含3个节点的Spark集群中运行这个Spark程序时，输出如下：

```
$ ./run_build_markov_model.sh
```

```
inputPath:args[0]=/home/hadoop/testspark/smart_email_training.txt
...
INFO : Executor updated: app-20141106000336-0011/0 is now RUNNING
INFO : Executor updated: app-20141106000336-0011/1 is now RUNNING
INFO : Executor updated: app-20141106000336-0011/2 is now RUNNING
...
INFO : Stage 8 (saveAsTextFile at SparkMarkov.java:277) finished in 0.300 s
INFO : Removed TaskSet 8.0, whose tasks have all completed, from pool
INFO : Job finished: saveAsTextFile at SparkMarkov.java:277, took 3.769665517 s
...
```

这一章介绍了一个非常重要的数学统计概念：马尔可夫模型。我们了解了如何在分布式编程环境中使用马尔可夫模型根据给定的前一状态来预测下一状态。

下一章将讨论如何在MapReduce环境中实现K-均值聚类。

# 第12章

# K-均值聚类

这一章将为K-均值聚类算法提供一个MapReduce解决方案。K-均值聚类算法是一个很有意思的MapReduce算法，而且与其他MapReduce算法有所区别。这是一个迭代算法（也就是说，它需要多个Map?Reduce阶段），需要根据不同的质心（centroids）执行多次，直至其收敛（这表示多次迭代相同的MapReduce作业后找到最优的簇）。

那么什么是聚类？什么是K-均值算法（K-Means）呢？基本说来，给定K > 0 （这里K是簇数）和一个集合（其中包括需要聚类的*N*个*d*-维对象）：

- 聚类是将*N*个*d*-维（2-维、3-维等）对象分组为K个类似对象簇的过程。
- 同一个簇中的对象彼此类似，而不同簇中的对象彼此相异。

K-均值算法是一个基于距离的聚类算法。K-均值聚类有很多有用的应用。例如，可以用来找出有共同行为的一组消费者，或者根据文档内容的相似性对文档完成聚类。首要的问题是：如何确定K值（即希望由输入数据生成多少个组或簇）？K的选择与具体的应用或问题域有关。关于找出K，并没有一个万能的公式。

作为K-均值算法的一个例子，图12-1和图12-2展示了*K*=3的K-均值算法。图12-1展示了原始数据，图12-2展示了K-均值算法的应用。在图12-2中，红点表示第1个簇，黑点表示第2个簇，最后的绿点表示第3个簇。K-均值算法的目标就是将原始数据转换为簇。

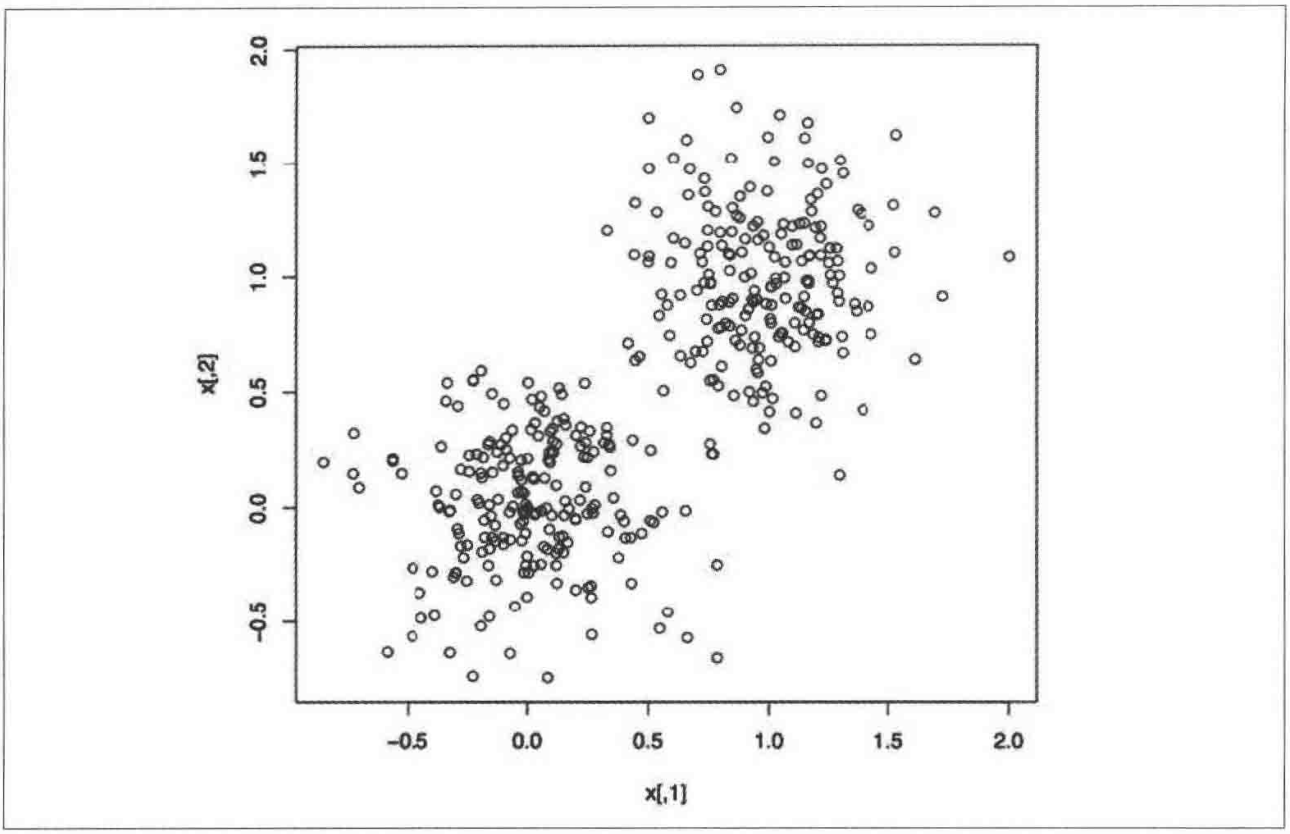



图12-1：原始数据（http://bit.ly/data-algorithms_12-1_color）

K-均值算法的工作如下：为这个算法提供一个集合（其中包含$N$个$d$-维点）和希望的簇数K作为输入。为简单起见，我们会考虑欧氏空间中的点。不过，只要能提供一个距离度量作为输入，K-均值聚类算法同样适用于任何空间。因此，我们的输入（$N$个$d$-维点）如下：（这里会交替使用$n$和$N$）

$$p_1 = (a_{11}, a_{12}, ..., a_{1d})$$

$$p_2 = (a_{21}, a_{22}, ..., a_{2d})$$

...

$$p_n = (a_{n1}, a_{n2}, ..., a_{nd})$$

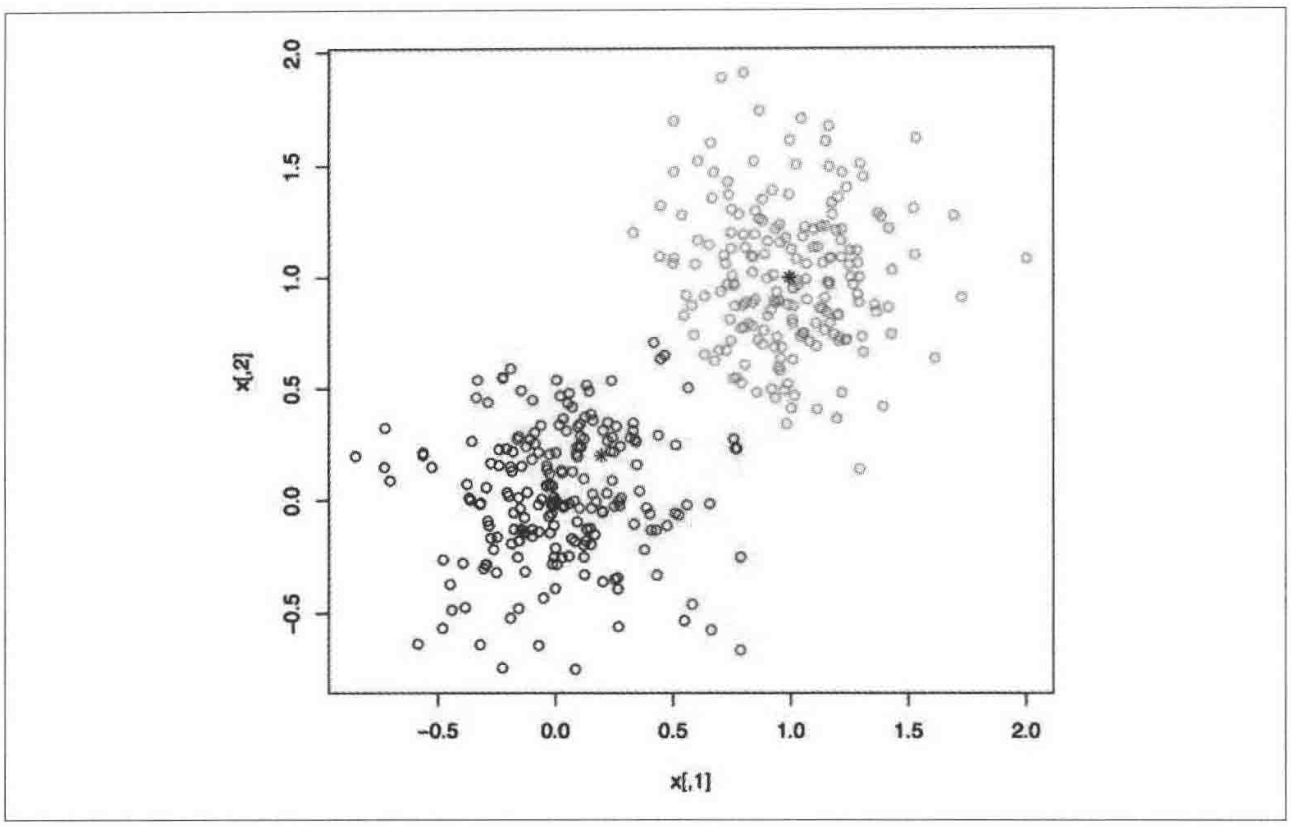



图12-2：聚类的数据（http://bit.ly/data-algorithms_12-2_color）

例如，在一个2-维环境中，输入数据可能如下，其中每一行表示一个点（$x$, $y$）：

```
p1 = (1,1)
p2 = (2,1)
p3 = (1,2)
p4 = (5,5)
p5 = (6,5)
p6 = (5,6)
p7 = (7,7)
p8 = (9,6)
```

开始时，会选择K个点作为簇中心，这些点称为簇质心（centroids）。可以采用很多方法来初始化簇质心，其中一种方法是从$n$个点的样本中随机选择K个点。一旦选择了K个初始的簇质心，下面可以计算输入集合中各个点到这K个中心点的距离，然后将各个点分配到与它距离最近的簇中心。所有对象都分配之后，再重新计算K个质心的位置。这两步反复迭代，直到簇质心不再改变（或者变化非常小）。

给出我们的MapReduce解决方案之前，首先来看K-均值聚类的基本定义，然后我们将简要介绍K-均值聚类的形式化和非形式化算法。对K-均值聚类了解越充分，就能越容易地理解MapReduce解决方案。

# 什么是K-均值聚类？

基本说来，K-均值聚类［又称为无监督学习（unsupervised learning）］是一个数据挖掘算法，可以根据对象的属性或特性将N个对象聚类、分类或分组到K个组（称为簇）。K是一个正整数（K = 2, 3, 4, ...）。如果对一个包含N个对象的集合应用K-均值算法，结果是K个不相交的组（也就是说，这K个组中的所有对象相加可以得到N个对象）。那么，如何将这N个对象分组到K个不相交的组中呢？要完成这个分组，我们的做法是让数据与相应簇质心间距离的平方和最小。接下来还有两个问题，为什么要使用聚类，另外如何找到簇质心？第一个问题的答案是：我们将使用K-均值聚类对数据分类，例如，可以把学生的分数分为5类（K = 5）：A（优秀）、B（良好）、C（一般）、D（差）和F（未通过）。或者再看另一个例子，可以使用K-均值算法预测学生的学业表现[27]。后面的小节中我会再来讨论第二个问题，即如何找到簇质心。

接下来给出K-均值聚类的形式化描述。给定$n$个$d$-维点：

$X_1 = (x_{11}, x_{12}, ..., x_{1d})$

$X_2 = (x_{21}, x_{22}, ..., x_{2d})$

...

$X_n = (x_{n1}, x_{n2}, ..., x_{nd})$

我们的目标是将这个$\{X_1, X_2, ..., X_n\}$划分为K个簇：$\{C_1, C_2, ..., C_k\}$。K-均值算法的目的是找出这些簇的位置$\mu_i$ $(i = 1, ..., K)$，使数据点到簇质心的距离最小。K-均值聚类可以解决以下最小代价算法：

$$\arg \min_{C} \sum_{i=1}^{k} \sum_{X_j \in C_i} || X_j - \mu_i ||^2$$

这里$\mu_i$是$C_i$中点的平均值。

# 聚类的应用领域

聚类算法有很多可能的应用，包括：

*市场营销*

给定一个很大的顾客交易集，找出有类似购买行为的顾客分组。

*文档分类*

对Web日志数据聚类，发现有类似访问模式的分组。

*保险*

通过识别可能的欺诈行为找出平均索赔支出很高的车险投保人群。

# K-均值聚类方法非形式化描述：分区方法

这一章开始时我们提到过，K-均值聚类算法是迭代的。相应地，后面的小节将提供一个MapReduce实现，我们会不断运行这个实现，直到找到一个合适的最优方案，使K-均值聚类算法收敛。非正式地，K-均值聚类算法可以总结如下：给定一个$N$个对象的集合，要将这些对象分组到K个簇中，K-均值算法需要完成以下步骤：

1. 将$N$个对象划分到K个非空子集中。
2. 计算当前分区中的簇质心（质心是这个簇的中心或平均点）。对于所有$i$=1, 2, ..., $K$，$\mu_i$ 计算为：

$$\mu_i = \frac{1}{|\ c_i\ |}\sum_{j \in C_i} \underline{x}_j, \forall i$$

3. 将各个对象分配到有最近质心的簇。这样会为各个数据点找到最近的簇：

   $c_i = \{j : d(x_j,\mu_i) \leq d(x_j,\mu_1), 1 \neq i, j = 1, ..., n\}$

   其中$d(a, b)$是两个点$a$和$b$的距离函数。

4. 如果不再有新的分配，则停止计算。否则，回到步骤2。一般需要重复步骤2和3直到算法收敛。

这个算法会反复迭代，直到质心不再变化，此时就找到了我们想要的K个簇。由于K-均值算法是一个迭代过程，每一步都会根据各个现有簇的当前中心重新计算簇中各个对象的成员关系。这个过程会反复进行，直到达到期望的簇数（或对象数）。

# K-均值距离函数

K-均值聚类算法的第一步需要把各个数据点分配到最近的簇质心。对于一个给定的（$d$维）数据点，可以使用一个距离函数来确定最近的簇质心，这个函数会计算这个质心生成该数据点的可能性有多大。有很多可用的距离函数，包括：

- 欧氏距离。
- 曼哈顿距离。
- 内积空间。
- 最大模。
- 你自己的定制函数（可以是$d$维空间上定义的任何度量函数）。

例如，很多K-均值算法中都使用了欧氏距离。要得出两个数据点实例$X$和$Y$（表示为$d$个连续属性，即$d$-维）之间的欧氏距离：

令：

$$X = (X_1, X_2, ..., X_d)$$

$$Y = (Y_1, Y_2, ..., Y_d)$$

则有：

$$\text{distance}(X,Y)=\sqrt{(X_1-Y_1)^2+(X_2-Y_2)^2+...+(X_d-Y_d)^2}$$

欧氏距离函数有一些有趣的性质：

- distance(i,j) ⩾ 0
- distance(i,i) = 0
- distance(i,j) = distance(j,i)
- distance(i,j) ⩽ distance(i,k) + distance(k,j)

下面给出一个类Java的欧氏距离函数：

```
public class EuclideanDistance {
   // 令Vector[i] 为Vector的第i个位置
   public static double calculateDistance(Vector center, Vector data) {
      double sum = 0.0;
      // center.length = data.length
      int length = center.length;
      for (int i = 0; i < length; i++) {
         sum += Math.pow((center[i] - data[i]), 2.0);
      }
      return Math.sqrt(sum);
    }
}
```

实现K-均值算法时，必须确保算法收敛，而且可以在数据空间上正确地计算平均值和距离函数。

# K-均值聚类形式化描述

K-均值算法是一个完成聚类分析的简单学习算法。K-均值聚类算法的目标是找出n项的最佳划分，也就是将n个对象或点划分到K个组中，使得一个组中的成员与其相应的质心（表示这个组）之间的总距离最小。采用形式化表示，目标就是将n项划分到K个集合$\{S_i, i = 1, 2, ..., K\}$，使得簇内平方和或组内平方和（within-cluster sum of squares，WCSS）最小，WCSS定义为：

$$\min \sum_{j=1}^{k}\sum_{i=1}^{n} \| x_i^j - c_j \|$$

这里 $\| x_i^j - c_j \|$ 提供了实体点与簇质心之间的距离。

# K-均值聚类的MapReduce解决方案

K-均值聚类的MapReduce解决方案是一个迭代方案。其中每一次迭代实现为一个MapReduce作业。K-均值算法需要一个迭代版本的MapReduce，这不是MapReduce范式的标准形式。要实现一个迭代的MapReduce解决方案，我们需要有一个驱动器或者客户端控制程序来初始化K个质心位置，迭代调用MapReduce作业，并确定应当继续迭代还是应当停止。映射器需要获取数据点和所有簇质心，簇质心必须由所有映射器共享。对此，可以采用很多不同的方法来管理，例如可以使用HDFS或者一个全局数据结构服务器（如Redis或memcached）。归约器会重新计算新的平均值。

需要说明，算法的每一次迭代都实现为一个MapReduce作业，这会反复改进K个簇的数据划分。总的MapReduce伪代码解决方案如示例12-1所示。质心没有太大变化时，将使用change()函数终止MapReduce迭代。

示例12-1：K-均值聚类算法

```
// k = 期望的簇数
// delta = 可接受的收敛误差
// data = 输入数据
kmeans(k, delta, data) {
    // 初始化簇质心
    initial_centroids = pick(k, data);

    // 利用这个方法向映射器广播中心
    writeToHDFS(initial_centroids);

    // 必要时迭代
    current_centroids = initial_centroids;
    while (true) {
        // theMapReduceJob()完成2个任务：
        //    1. map()中使用current_centroids
        //    2. reduce()创建new_centroids并写至HDFS
        theMapReduceJob();
        new_centroids = readFromHDFS();
        if change(new_centroids, current_centroids) <= delta {
           // 完成，终止循环迭代
           break;
        }
        else {
           current_centroids = new_centroids;
        }
   }
```

```
27
28   result = readFromHDFS();
29   return result;
30 }
```

change()方法如示例12-2所示。

示例12-2：K-均值聚类算法：change()方法

```
1 change(new_centroids, current_centroids) {
2    new_distance = [sum of squared distance in the new_centroids];
3    current_distance = [sum of squared distance in the current_centroids];
4    changed = absoulteValue(new_distance - current_distance);
5    return changed;
6 }
```

下面3小节将介绍MapReduce作业的详细内容，包括3个函数：map()、combine()和reduce()。

## MapReduce解决方案：map()

map()函数完成分类。它会使用簇质心，并把各个点$p \in 1, 2, ..., n$分配到最近的中心。需要说明，第一次迭代时，质心是随机选择的一个点，或者手动创建。map()接受数据集中的点，并为每个点输出一个（Cluster-ID, Vector）对，其中Cluster-ID是与这个点最近的簇的整数ID，Vector是一个$d$维向量（double数据类型的一个数组）。

map()函数将完成下面的工作，如示例12-3所示：

- 从SequenceFile[注1]将簇质心读入内存（需要说明，也可以使用Redis或memcached来持久存储簇质心）。在Hadoop实现中，这将在映射器类的setup()方法中完成。
- 迭代处理对应各个输入键-值对的各个簇质心。在Hadoop实现中，对于map()函数，键由Hadoop生成并忽略（不使用）。
- 计算欧氏距离，并保存与输入点（作为一个$d$维向量）有最小距离的最近中心。
- 写出将由归约器处理的键-值对，其中键是离输入点最近的簇中心（值是一个$d$维向量）。键和值都是Vector数据类型。

示例12-3：K-均值MapReduce解决方案：map()函数

```
1 public class KmeansMapper ... {
2
3    private List<Vector> centers = null;
4
```

注1：org.apache.hadoop.io.SequenceFile。

```
5    private List<Vector> readCentersFromSequenceFile() {
6       // 从 SequenceFile读取簇质心，
7       // 这是一个键-值对集合
8       ...
9    }
10
11   // 映射任务开始时调用一次
12   public void setup(Context context) {
13      this.centers = readCentersFromSequenceFile();
14   }
15
16   /**
17    * @param key由MapReduce生成，在这里将忽略
18    * @param value是d维向量 (V1, V2, ..., Vd)
19    */
20   map(Object key, Vector value) {
21      Vector nearest = null;
22      double nearestDistance = Double.MAX_VALUE;
23      for (Vector center : centers) {
24          double distance = EuclideanDistance.calculateDistance(center, value);
25          if (nearest == null) {
26              nearest = center;
27              nearestDistance = distance;
28          }
29          else {
30             if (nearestDistance > distance) {
31                 nearest = center;
32                 nearestDistance = distance;
33             }
34          }
35      }
36
37      // 为归约器准备键-值对
38      emit(nearest, value);
39   }
40
41 } // 类KmeansMapper结束
```

## MapReduce解决方案: combine()

各个映射任务之后，会应用组合器来组合映射任务的中间数据。组合器将累加向量对象各个维的值（需要这个累加和来计算平均值）。在combine()函数中，我们会部分累加分配到相同簇的点（作为向量）的值。combine()函数可以提高算法的效率，因为通过减少从节点（即工作节点）之间传输的数据，可以减少网络流量。如示例12-4所示。

示例12-4：K-均值MapReduce解决方案: combine()函数

```
1 /**
2  * @param key是质心
3  * @param values是一个向量列表
4  */
5 combine(Vector key, Iterable<Vector> values) {
6   // sum向量中的所有维都初始化为0.0
```

```
7   Vector sum = new Vector();
8   for (Vector value : values) {
9      // 注意value.length = d,
10     // 这里d是输入对象的维数
11     for (int i = 0; i < value.length; i++) {
12        sum[i] += value[i];
13     }
14  }
15
16  emit(key, sum);
17 }
```

## MapReduce解决方案: reduce()

归约器重新计算中心：它会重新计算所有簇的平均值，从而重新创建所有簇的质心（参见示例12-5）。在reduce()阶段，map()函数的输出按Cluster-ID分组，对于各个Cluster-ID，将计算与这个Cluster-ID关联的点的质心。每一个reduce()函数会生成一个（Cluster-ID, Centroid）对，这表示新计算得到的簇中心。reduce()函数可以总结如下：

- reduce()函数的主要任务是重新计算中心。
- 每个归约器迭代处理各个值向量，并计算平均值向量。一旦找到平均值，这就是新的中心；最后一步就是将它存储在一个持久存储库中（如SequenceFile）。

示例12-5：K-平均MapReduce解决方案: reduce()函数

```
1 /**
2  * @param key是质心
3  * @param values是一个向量列表
4  */
5 reduce(Vector key, Iterable<Vector> values) {
6   // newCenter中的所有维都初始化为0.0
7   Vector newCenter = new Vector();
8   int count = 0;
9   for (Vector value : values) {
10     count++;
11     for (int i = 0; i < value.length; i++) {
12        newCenter[i] += value[i];
13     }
14  }
15
16  for (int i = 0; i < key.length; i++) {
17     // 设置各个维的新平均值
18     newCenter[i] = newCenter[i] / count;
19  }
20
21  emit(key.ID, newCenter);
22 }
```

# K-均值算法Spark实现

Spark框架在一个MLlib库中提供了一些常用的机器学习（machine learning，ML）功能。MLlib支持以下机器学习算法：

- 二元分类（Binary classification）。
- 回归（Regression）。
- 聚类（Clustering，包括K-均值算法）。
- 协同过滤（Collaborative filtering）。
- 梯度下降优化原语（Gradient descent optimization primitive）。

MLlib支持K-均值聚类，可以将数据点分组到预定数目的一组簇中。MLlib实现包括k-means++方法的一个并行化版本，称为kmeansII（也就是K-均值算法的一个并行实现）。K-均值算法的Spark实现有以下参数：

- K是期望的簇数。
- maxIterations是要运行的最大迭代次数。
- initializationMode指定随机初始化还是通过kmeans||初始化。
- runs是运行K-均值算法的次数（需要指出，不能保证K-均值算法一定能找到一个全局最优的解决方案，即使在一个给定的数据集上运行多次，也不能保证这个算法会返回最佳的聚类结果）。
- initializiationSteps确定kmeans||算法中的步数。
- epsilon确定距离阈值，达到这个阈值时就可以认为K-均值算法已经收敛。

Spark提供了JavaKMeans类[注2]作为K-均值算法的示例实现.在示例12-6中，我给出了这个JavaKMeans类，后面将展示一个运行示例。

示例12-6：从Java使用MLlib K-均值算法

```
1 package org.apache.spark.examples.mllib;
2
3 import java.util.regex.Pattern;
4 import org.apache.spark.SparkConf;
5 import org.apache.spark.api.java.JavaRDD;
6 import org.apache.spark.api.java.JavaSparkContext;
7 import org.apache.spark.api.java.function.Function;
8 import org.apache.spark.mllib.clustering.KMeans;
9 import org.apache.spark.mllib.clustering.KMeansModel;
10 import org.apache.spark.mllib.linalg.Vector;
```

注2：org.apache.spark.examples.mllib.JavaKMeans。

```
import org.apache.spark.mllib.linalg.Vectors;

/**
 * 从Java使用MLlib K-均值算法示例
 */
public final class JavaKMeans {

  private static class ParsePoint implements Function<String, Vector> {
    private static final Pattern SPACE = Pattern.compile(" ");
    @Override
      public Vector call(String line) {
        String[] tok = SPACE.split(line);
        double[] point = new double[tok.length];
        for (int i = 0; i < tok.length; ++i) {
          point[i] = Double.parseDouble(tok[i]);
        }
        return Vectors.dense(point);
    }
  }

  public static void main(String[] args) {
    if (args.length < 3) {
      System.err.println(
        "Usage: JavaKMeans <input_file> <k> <max_iterations> [<runs>]");
      System.exit(1);
    }
    String inputFile = args[0];
    int k = Integer.parseInt(args[1]);
    int iterations = Integer.parseInt(args[2]);
    int runs = 1;

    if (args.length >= 4) {
      runs = Integer.parseInt(args[3]);
    }

    SparkConf sparkConf = new SparkConf().setAppName("JavaKMeans");
    JavaSparkContext sc = new JavaSparkContext(sparkConf);
    JavaRDD<String> lines = sc.textFile(inputFile);
    JavaRDD<Vector> points = lines.map(new ParsePoint());
    KMeansModel model = KMeans.train(points.rdd(), k, iterations, runs,
                                     KMeans.K_MEANS_PARALLEL());

    System.out.println("Cluster centers:");
    for (Vector center : model.clusterCenters()) {
      System.out.println(" " + center);
    }
    double cost = model.computeCost(points.rdd());
    System.out.println("Cost: " + cost);

    sc.stop();
  }
}
```

下面对JavaKMeans类的要点做几点说明：

- 第47行：创建JavaSparkContext对象，这是Spark master的一个连接对象，也是在Spark集群中运行作业的一个入口点。
- 第48行：读取输入文件（作为一个字符串记录），并创建一个新的JavaRDD<String>（字符串对象集合）。
- 第49行：将JavaRDD<String>中的各个字符串记录转换为一个double Vector。我们使用ParsePoint类来完成这个工作，它取一个字符串对象并完成词法分析。
- 第50行：KMeans类[注3]调用K-均值算法，这会生成一个KMeansMode[注4]（K-均值算法的一个聚类模型，各个点分别属于中心与之最近的簇）。
- 第53～55行：打印簇中心。
- 第57行：返回这个模型在给定数据上的K-均值算法代价（各个点与其最近中心的距离的平方和）。

## K-均值算法Spark实现运行示例

这一节给出K-均值算法Spark实现的脚本、示例输入和运行示例的日志。

### 脚本

下面的示例shell脚本将运行这个K-均值程序：

```
 1 $ cat run_spark_kmeans_example.sh
 2 #!/bin/bash
 3 export JAVA_HOME=/usr/java/jdk7
 4 export SPARK_HOME=/home/hadoop/spark-1.1.0
 5 export SPARK_MASTER=spark://myserver100:7077
 6 SPARK_JAR=spark-examples-1.1.0-hadoop2.5.0.jar
export APP_JAR=$SPARK_HOME/examples/target/scala-2.10/$SPARK_JAR
 7 # 在一个Spark集群上运行
 8 prog=org.apache.spark.examples.mllib.JavaKMeans
 9 inputfile=/home/hadoop/testspark/kmeans_input_file.txt
10 K=3
11 iterations=10
12 $SPARK_HOME/bin/spark-submit \
13   --class $prog \
14   --master $SPARK_MASTER \
15   --executor-memory 2G \
16   --total-executor-cores 20 \
17   $APP_JAR \
18   $inputfile $K $iterations
```

注3：org.apache.spark.mllib.clustering.KMeans。

注4：org.apache.spark.mllib.clustering.KMeansModel。

## 输入

```
$hadoop fs -cat /home/hadoop/testspark/kmeans_input_file.txt
1.0 2.0
1.0 3.0
1.0 4.0
2.0 5.0
2.0 6.0
2.0 7.0
2.0 8.0
3.0 100.0
3.0 101.0
3.0 102.0
3.0 103.0
3.0 104.0
```

## 示例运行日志

```
$ ./run_spark_kmeans_example.sh
...
INFO : Job finished: collectAsMap at KMeans.scala:193, took 0.174752238 s
INFO : Run 0 finished in 1 iterations
INFO : Iterations took 0.193 seconds.
INFO : KMeans converged in 1 iterations.
INFO : The cost for the best run is 17.750000000000007.
INFO : Removing RDD 3 from persistence list
INFO : Removing RDD 3
Cluster centers:
[3.0, 102.0]
[1.25, 3.5]
[2.0, 7.0]
...
INFO : Job finished: sum at KMeansModel.scala:56, took 0.090478143 s
Cost: 17.750000000000007
...
```

这一章介绍了K-均值聚类技术。这里提供了一个迭代MapReduce算法来解决K-均值聚类问题，另外研究了K-均值聚类的一个Spark解决方案。

下一章将讨论另一种机器学习算法，称为kNN（k-近邻算法）。

# 第13章

# k-近邻

这一章重点介绍一个重要的机器学习算法，称为k-近邻（k-Nearest Neighbors，kNN），这里k是一个大于0的整数。kNN分类问题是找出一个数据集中与一个给定查询数据点最近的k 个数据点。这个操作也称为kNN连接（kNN join），可以定义为：给定两个数据集R和S，对于R中的每一个对象，我们希望从S中找出k个最近的相邻对象。在数据挖掘中，R和S分别称为查询（query）和训练（training）数据集。训练数据集(S)表示已经分类的数据，而查询数据集（R）表示将利用S中的分类来进行分类的数据。

这一章的目的是使用Spark为kNN连接算法提供一个MapReduce解决方案。kNN算法在很多机器学习书中已经做了广泛深入的讨论，如《Machine Learning for Hackers》[8]和《Machine Learning in Action》[10]。kNN是一个重要的聚类算法，在数据挖掘（如图像识别）和生物信息（如乳腺癌诊断[23]、天气数据生成模型和商品推荐系统）中有很多应用。kNN算法的开销可能很昂贵，特别是有一个庞大的训练集时，正是因为这个原因，MapReduce非常适用。

那么到底什么是kNN？近邻分析或近邻搜索是一个根据*n*-维对象[注1]与其他*n*维对象的相似度完成分类的算法。在机器学习中，近邻分析被作为一种识别数据模式的方法，而不需要与任何已存储的模式或对象完全匹配。类似的*n*-维对象相互靠近，而不同的*n*-维对象相距很远。因此，两个对象间的距离是其不相似度的一个度量。根据Wikipedia[注2]：

> 在模式识别中，k-近邻算法（kNN）是一个根据特性空间中最近的训练示例对对象分类的一种非参数化方法。kNN是一种基于实例的学习方法，或一种懒学习，在局部只

注1：*n*-维对象*x*有*n*个属性，表示为（$x_1$, $x_2$, ..., $x_n$）。

注2：2014年9月时Wikipedia给出的描述。

是近似函数，所有计算将延迟到真正分类时才完成。kNN算法是所有机器学习算法中最简单的算法之一：对象根据其相邻对象的多数投票来完成分类，对象将分配到其$k$个近邻中大多数对象所在的类（$k$是一个正数，通常很小）。如果$k$=1，那么这个对象就会直接分配到其近邻所在的类。

# kNN分类

kNN的中心思想是建立一个分类方法，使得对于将$y$（响应变量）与$x$（预测变量）关联的“平滑”函数$f$的形式没有任何假设：

$$x = (x_1, x_2, \ldots, x_n)$$

$$y = f(x)$$

函数$f$是非参数化的，因为它不涉及任何形式的参数估计。在kNN中，给定一个新的点$p=(p_1, p_2, \ldots, p_n)$，要动态地识别训练数据集中与p相似的k个观察（k个近邻）。近邻由一个距离或不相似度来定义，可以根据独立变量计算不同观察之间的距离。为简单起见，我们将使用欧氏距离。

点$(x_1, x_2, \ldots, x_n)$和$(p_1, p_2, \ldots, p_n)$之间的欧氏距离定义如下：

$$\sqrt{(x_1-p_1)^2+(x_2-p_2)^2+\ldots+(x_n-p_n)^2}$$

那么如何找出$k$个近邻呢？对于每一个要查询的$n$维对象（也就是有$n$个属性的对象），计算该查询对象与所有训练数据对象之间的欧氏距离，然后将这个查询对象分配到$k$个最近训练数据中大多数对象所在的类。需要说明，kNN算法假设所有数据都对应一个$n$维数值空间（表示为$R_n$），这说明，所有属性的数据类型都是`double`。必须使用这个数据类型，因为我们需要计算$n$维对象之间的欧氏距离。

# 距离函数

前面已经提到，很多数据挖掘聚类算法中都使用了欧氏距离函数。下面给出两个$n$维对象$X$和$Y$之间的欧氏距离。令$X$和$Y$定义如下：

$$X = (X_1, X_2, \ldots, X_n)$$

$$Y = (Y_1, Y_2, \ldots, Y_n)$$

$distance(X, Y)$可以定义如下：

$$\text{distance}(X, Y)=\sqrt{\sum_{i=1}^{n}(X_i - Y_i)^2}$$

$$\text{distance}(X, Y)=\sqrt{(X_1 - Y_1)^2+(X_2 - Y_2)^2+\cdots+(X_n - Y_n)^2}$$

再次说明，欧氏距离函数只适用于数值数据，要求所有属性都使用基本数据类型double。不过，如果属性是非数值数据（例如，"big spender", "medium spender", "small spender"和"non spender"），该怎么做呢？在这种情况下，需要提供定制距离函数来适应不同的数据类型；也就是说，我们需要回答distance("big spender", "small spender")等问题。大多数情况下，在实际应用中都需要定制的距离函数。

取决于具体的应用或问题域，对于kNN，可能还希望考虑其他一些距离函数，如：

- 曼哈顿距离（Manhattan）：

  $$\sum_{i=1}^{n}| X_i - Y_i |$$

- 闵可夫斯基距离（Minkowski）：

  $$\left(\sqrt{\sum_{i=1}^{n}(| X_i - Y_i |)^q}\right)^{1/q}$$

# kNN示例

先对kNN算法做个简单的复习：kNN算法是一种对未分类数据（称为输入查询）进行分类的直观方法，它会根据未分类数据与训练数据集中数据的相似度或距离完成分类。在我们的例子中，假设训练数据集有4个类 $\{C_1, C_2, C_3, C_4\}$，如图13-1所示。

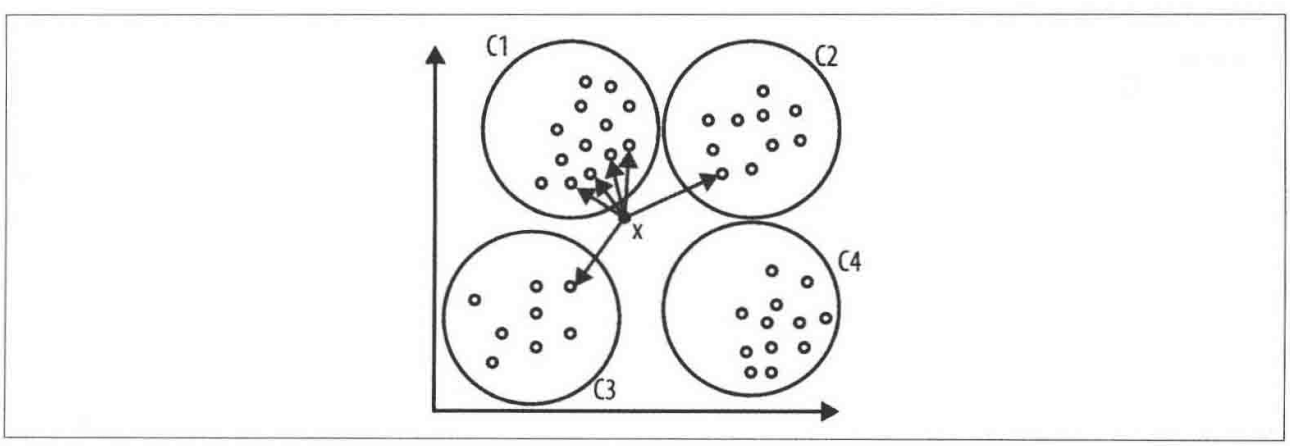



图13-1：4个类 $\{C_1, C_2, C_3, C_4\}$ 的kNN分类

给定$k$=6和$X = (X_1, X_2, \ldots, X_n)$，我们的目标是从$\{C_1, C_2, C_3, C_4\}$找出未知数据$X$的类标签。从图13-1可以看到，在6个近邻中，其中4个属于$C_1$，1个属于$C_2$，还有1个属于$C_3$，没有属于$C_4$的近邻点。因此，按照多数投票，$X$会分配到$C_1$，这是主导类。

# kNN算法非形式化描述

kNN算法可以总结为以下的简单步骤：

1. 确定$k$（$k$的选择取决于具体的数据和项目需求，对于$k$的确定并没有万能的魔法公式）。
2. 计算新输入与所有训练数据之间的距离（与$k$一样，距离函数的选择也取决于数据的类型）。
3. 对距离排序，并根据第$k$个最小距离确定$k$个近邻。
4. 收集这些近邻所属的类别。
5. 根据多数投票确定类别。

# kNN算法形式化描述

要形式化地定义kNN算法，需要首先了解一些术语：

*距离函数（Distance function）*

距离函数取决于具体的应用领域，不过大多数情况下，对于大多数数值数据，欧氏距离函数就很适用。两个点$(x_1, x_2, \ldots, x_n)$和$(y_1, y_2, \ldots, y_n)$之间的距离定义为$|x, y|$。

*K近邻（k nearest neighbors）*

给定一个对象$q$（称为一个查询对象）、一个训练数据集$S$和一个整数$k$，$S$中$q$的$k$个近邻表示为kNN($q$, $S$)，这是$S$中$k$个对象的集合，使得：

$$\forall o \in \text{kNN}(q, S), \forall s \in \{S - \text{kNN}(q, S)\}, |o, q| \leqslant |s, q|$$

*kNN连接（kNN join）*

给定两个数据集$R$和$S$（其中$S$是一个训练数据集）和一个整数$k$，$R$和$S$的kNN连接定义为：

$$\text{kNNjoin}(R, S) = \{(r, s) | \forall r \in R, \forall s \in \text{kNN}(r, S)\}$$

基本说来，这会将各个对象$r \in R$与它在$S$中的$k$个近邻组合。

# kNN的类Java非MapReduce 解决方案

给定$K > 0$, 并给定一个查询集（即我们想要分类的数据）和一个训练集（也就是已经分类

的数据对象），接下来，我会为kNN算法提供一个类Java的非MapReduce解决方案。在示例13-1中，使用Point类表示*n*-维数据。

示例13-1：classify()方法

```
/**
 * 使用kNN分析对查询数据分类
 * @param k 是一个> 0的整数
 * @param querySet是我们想要分类的数据对象集合
 * @param trainingSet是一个训练数据集合（已经分类）
 */
public static void classify(int k,
                            Point[] querySet,
                            Point[] trainingSet) {
  foreach (Point query : querySet) {
      knn(k, query, trainingSet);
  }
}
```

KNN算法（非MapReduce版本）如示例13-2所示。

示例13-2：kNN算法

```
/**
 * kNN分析
 * @param k是一个>0的整数
 * @param query是我们想要分类的一个数据对象
 * @param trainingSet是一个训练数据集合（已经分类）
 */
public static void knn(int k,
                       Point query,
                       Point[] trainingSet) {
  foreach (Point training : trainingSet) {
     // 创建长度为k的一个固定长度有序映射，
     // 将距离映射到一个训练点。
     SortedMap<Double, Point> map = new TreeMap<Double, Point>(k);

     // 计算测试点到训练点的距离。
     double d = calculateEuclidianDistance(query, training);

     // 将训练点插入有序列表，如果
     // 训练点不在查询点的k个近邻中，
     // 将其删除。
      map.put(d, training);
  }

  // 对k个近邻完成多数投票，
  // 并为查询点分配
  // 相应的类标签。
  query.label = majorityVote(map, k);
}
```

示例13-3定义了欧氏距离函数。

示例13-3：欧氏距离函数

```
/**
 * @param query 是一个n-维查询数据对象
 * @param reference是一个n-维参考数据对象
 * @return 欧氏距离的平方
 */
public static double calculateEuclidianDistance(Point query,
                                                Point reference) {
  double sum = 0.0;

  // n是向量空间中的维数。
  // 在这里n等于query.length，
  // 这也等于reference.length。
  int n = query.length;

  for (int i = 0; i < n; i++) {
     double difference = reference.vector[i] - query.vector[i];
     sum += difference * difference;
  }

  return Math.sqrt(sum);
}
```

# Spark的kNN算法实现

给定两个数据集$R$和$S$，我们的目标是从$S$中为$R$中的各个对象找出k个近邻。前面已经了解到，在数据挖掘中，$S$称为训练数据集。这个kNN连接算法的复杂性是$O(N^2)$，因为对于$R$中的每一个对象$r$，需要计算对应$S$中每一个$s$的距离$distance(r, s)$。要找出$distance(r, s)$，这需要得到$R$和$S$的笛卡尔积。Spark提供了一个高层API来计算两个数据集之间的笛卡尔积。例如，JavaPairRDD.cartesian(JavaPairRDD)可以得到两个数据集之间的笛卡尔积。cartesian()函数的签名定义如下：

```
<U> JavaPairRDD<T,U> cartesian(JavaRDDLike<U,?> other)

// 描述：返回这个RDD与另一个RDD的笛卡尔积；
         也就是说，所有元素对(a, b)的RDD，
         其中a在this中，b在other中。
```

下面的例子展示了如何在Spark中计算两个数据集之间的笛卡尔积：

```
// 数据集R
List<Tuple2<String,String>> R = new ArrayList<Tuple2<String,String>>();
R.add(new Tuple2<String,String>("r1", "R1"));
R.add(new Tuple2<String,String>("r2", "R2"));
R.add(new Tuple2<String,String>("r3", "R3"));

// 数据集S
List<Tuple2<String,String>> S = new ArrayList<Tuple2<String,String>>();
S.add(new Tuple2<String,String>("S1", "S1"));
S.add(new Tuple2<String,String>("S2", "S2"));
```

```
S.add(new Tuple2<String,String>("s3", "S3"));
S.add(new Tuple2<String,String>("s4", "S4"));

JavaPairRDD<String,String> R_RDD = ctx.parallelizePairs(R);
JavaPairRDD<String,String> S_RDD = ctx.parallelizePairs(S);

// 完成R和S的笛卡尔积计算
JavaPairRDD<Tuple2<String,String>, Tuple2<String,String>> cart =
   R_RDD.cartesian(S_RDD);

// 保存输出
cart.saveAsTextFile("/output/z");
```

$R$和$S$的笛卡尔积计算的输出如下：

```
# hadoop fs -cat /output/z/part*
((r1,R1),(S1,S1))
((r1,R1),(S2,S2))
((r1,R1),(s3,S3))
((r1,R1),(s4,S4))
((r2,R2),(S1,S1))
((r2,R2),(S2,S2))
((r2,R2),(s3,S3))
((r2,R2),(s4,S4))
((r3,R3),(S1,S1))
((r3,R3),(S2,S2))
((r3,R3),(s3,S3))
((r3,R3),(s4,S4))
```

## kNN Spark实现的形式化描述

设$R$（查询数据集）和$S$（训练数据集）是$d$-维数据集，我们想找出其kNN($R$, $S$)。进一步假设所有训练数据($S$)已经分类到$C = \{C_1, C_2, ...\}$，这里C表示所有可能的分类。$R$、$S$和$C$定义如下：

- $R = \{R_1, R_2, ..., R_m\}$
- $S = \{S_1, S_2, ..., S_n\}$
- $C = \{C_1, C_2, ..., C_p\}$

在这里：

- $R_i = (r_i, a_1, a_2, ..., a_d)$
- $S_j = (s_j, b_1, b_2, ..., b_d, C_j)$
- $r_i$是$R_i$的唯一记录ID。
- $a_1, a_2, ..., a_d$是$R_i$的属性。
- $s_j$是$S_j$的唯一记录ID。

- $b_1, b_2, ..., b_d$是$S_j$的属性。
- $C_j$是$S_j$的分类标识符。

我们的目标是找出kNN($R$, $S$)。为此，要计算$R$和$S$的笛卡尔积，然后按$r_i$（$R_i$的唯一记录ID）对数据分组。分组之后，将从$S$中找出$k$个最近数据点（按升序对距离排序，然后选出前$k$个元素）。一旦从$S$中找到$k$个最近的数据点，再将按多数原则选择分类。图13-2展示了这个过程。

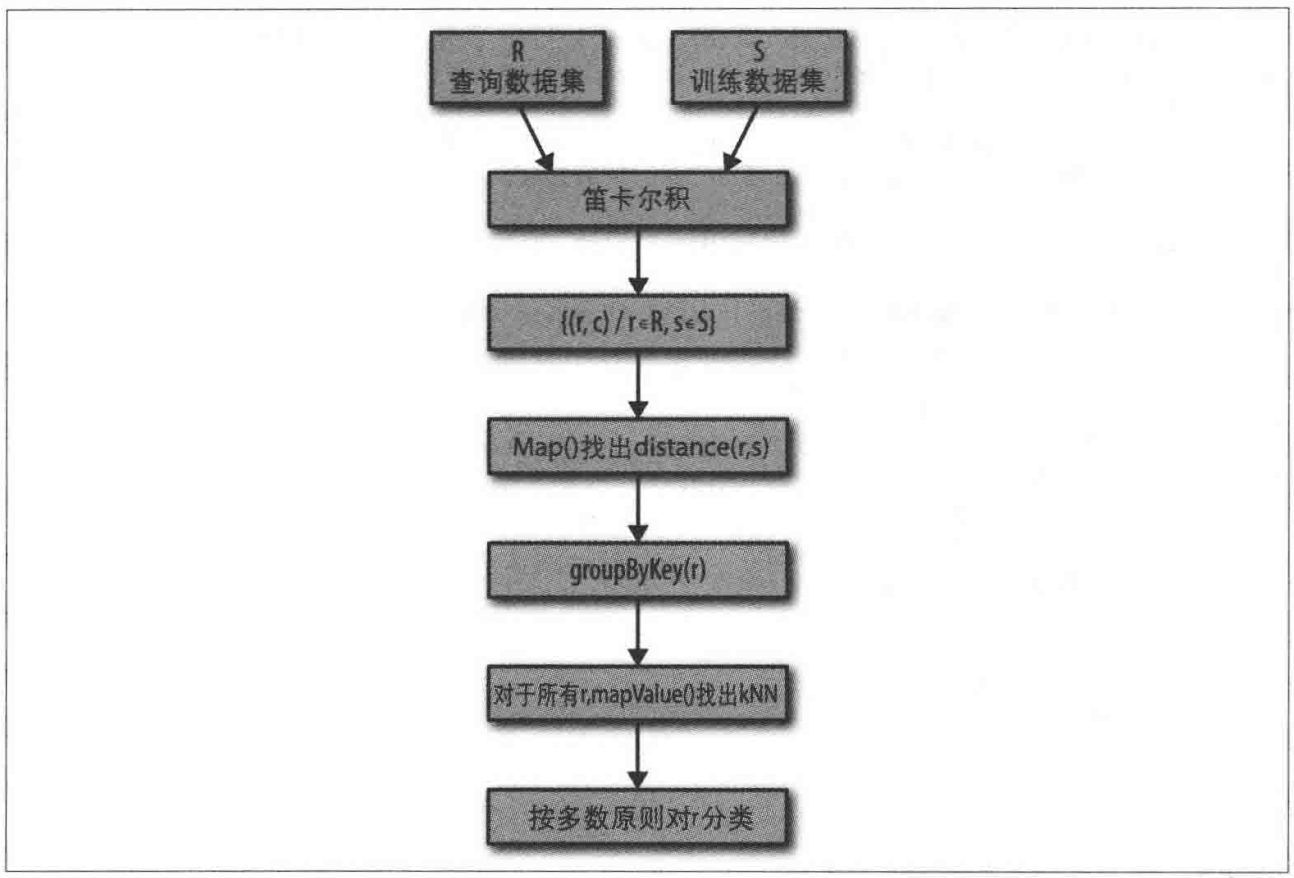



图13-2：kNN Spark实现

## 输入数据集格式

假设$R$和$S$都表示$d$-维数据点。查询数据集（$R$）的输入记录格式如下：

```
<unique-record-id><;><a-1><,><a-2><,>...<,><a-d>
```

训练数据集（$S$）的输入记录有以下格式：

```
<unique-record-id><;><classification-id><;><b-1><,><b-2><,>...<,><b-d>
```

## Spark实现

示例13-4展示了kNN算法Spark实现的高层步骤。在后面的小节中，我还会更详细地分析各个步骤。

示例13-4：高层步骤

```
// 步骤1：导入所需的类和接口

public class kNN {
    static JavaSparkContext createJavaSparkContext() throws Exception {...}
    static List<Double> splitOnToListOfDouble(String str, String delimiter) {...}
    static double calculateDistance(String r, String s, int d) {...}
    static SortedMap<Double, String> findNearestK(
          Iterable<Tuple2<Double,String>> neighbors, int k) {...}
    static Map<String, Integer> buildClassificationCount(
          Map<Double, String> nearestK) {...}
    static String classifyByMajority(Map<String, Integer> majority) {...}

    public static void main(String[] args) throws Exception {
       // 步骤2：  处理输入参数
       // 步骤3：  创建一个Spark上下文对象（ctx）
       // 步骤4：  广播共享对象
       // 步骤5：  为查询和训练数据集创建RDD
       // 步骤6：  计算（R, S）的笛卡尔积
       // 步骤7：  找出R中的r与S中的s之间的距离distance(r, s)
       // 步骤8：  按R中的r对距离分组
       // 步骤9：  找出k个近邻并对r分类

       // 完成
       ctx.close();
       System.exit(0);
    }
}
```

### 步骤1：导入所需的类和接口

大多数Spark类和接口都从两个包导入：`org.apache.spark.api.java`和`org.apache.spark.api.java.function`。示例13-5展示了如何导入这些类和接口。Broadcast类用来向所有集群节点广播共享对象和数据结构（这类似Hadoop的`Configuration`对象，可以在映射器和归约器中使用`set()`和`get()`设置和获取对象）。

示例13-5：步骤1：导入所需的类和接口

```
// 步骤1：导入所需的类和接口
import scala.Tuple2;
import org.apache.spark.SparkConf;
import org.apache.spark.broadcast.Broadcast;
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.api.java.JavaPairRDD;
import org.apache.spark.api.java.JavaSparkContext;
import org.apache.spark.api.java.function.Function;
import org.apache.spark.api.java.function.PairFunction;
```

```
10
11 import java.util.Map;
12 import java.util.HashMap;
13 import java.util.SortedMap;
14 import java.util.TreeMap;
15 import java.util.List;
16 import java.util.ArrayList;
17 import com.google.common.base.Splitter;
```

## createJavaSparkContext()方法

createJavaSparkContext()方法（如示例13-6所示）创建了JavaSparkContext对象的一个实例，这是一个工厂对象，可以用来创建RDD。

示例13-6：createJavaSparkContext()方法

```
1 static JavaSparkContext createJavaSparkContext() throws Exception {
2     SparkConf conf = new SparkConf();
3     // 使用一个快速串行化器
4     conf.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer");
5     return ctx;
6 }
```

## splitOnToListOfDouble()方法

splitOnToListOfDouble()方法如示例13-7所示，接受所有属性（作为一个字符串对象），返回一个List<Double>。

示例13-7：splitOnToListOfDouble()方法

```
 1 /**
 2  * @param str是一个逗号或分号分隔的double值列表
 3  * str 形如"1.1,2.2,3.3" 或 "1.1;2.2;3.3"
 4  *
 5  * @param delimiter是一个分隔符，如 ",", ";", ...
 6  * @return List<Double>（一个数据集记录的所有属性）
 7  */
 8 static List<Double> splitOnToListOfDouble(String str, String delimiter) {
 9     Splitter splitter = Splitter.on(delimiter).trimResults();
10     Iterable<String> tokens = splitter.split(str);
11     if (tokens == null) {
12         return null;
13     }
14     List<Double> list = new ArrayList<Double>();
15     for (String token: tokens) {
16         double data = Double.parseDouble(token);
17         list.add(data);
18     }
19     return list;
20 }
```

## calculateDistance()方法

calculateDistance()方法如示例13-8所示，接受两个向量$R$和$S$，计算它们之间的欧氏距离。两个点$r = (r_1, r_2, ..., r_d)$和$s = (s_1, s_2, ..., s_d)$之间的欧氏距离定义为：

$$\text{distance}(r,s) = \sqrt{(r_1 - s_1)^2 + (r_2 - s_2)^2 + ... + (r_d - s_d)^2}$$

根据具体的项目和数据需求，可以选择和使用其他的距离函数[注3]。

示例13-8：calculateDistance()方法

```
/**
 * @param rAsString = "r.1,r.2,...,r.d"
 * @param sAsString = "s.1,s.2,...,s.d"
 * @param d是R和S的维数
 */
static double calculateDistance(String rAsString, String sAsString, int d) {
   List<Double> r = splitOnToListOfDouble(rAsString, ",");
   List<Double> s = splitOnToListOfDouble(sAsString, ",");

   // d是向量中的维数
   if (r.size() != d) {
      return Double.NaN;
   }
   if (s.size() != d) {
      return Double.NaN;
   }

   // 这里 (r.size() == s.size() == d)
   double sum = 0.0;
   for (int i = 0; i < d; i++) {
     double difference = r.get(i) - s.get(i);
     sum += difference * difference;
   }
   return Math.sqrt(sum);
}
```

## findNearestK()方法

给定{(distance, classification)}，findNearestK()方法（如示例13-9所示）会根据这个距离找出$k$个近邻。

示例13-9：findNearestK()方法

```
static SortedMap<Double, String> findNearestK(
  Iterable<Tuple2<Double,String>> neighbors,
  int k) {
    // 只保留k个近邻
    SortedMap<Double, String> nearestK = new TreeMap<Double, String>();
    for (Tuple2<Double,String> neighbor : neighbors) {
```

注3：关于距离函数的有关详细内容，可以参考*http://www.saedsayad.com/k_nearest_neighbors.htm*。

```
        Double distance = neighbor._1;
        String classificationID = neighbor._2;
        nearestK.put(distance, classificationID);
        // 只保留k个近邻
        if (nearestK.size() > k) {
            // 从nearestK删除上一个最大距离邻点
            nearestK.remove(nearestK.lastKey());
        }
    }
    return nearestK;
}
```

## buildClassificationCount()方法

buildClassificationCount()如示例13-10所示，这是一个统计分类的简单方法（根据多数计数选择分类）。

示例13-10：buildClassificationCount()方法

```
static Map<String, Integer> buildClassificationCount(Map<Double, String>
                                                      nearestK) {
    Map<String, Integer> majority = new HashMap<String, Integer>();
    for (Map.Entry<Double, String> entry : nearestK.entrySet()) {
        String classificationID = entry.getValue();
        Integer count = majority.get(classificationID);
        if (count == null) {
            majority.put(classificationID, 1);
        }
        else {
            majority.put(classificationID, count+1);
        }
    }
    return majority;
}
```

## classifyByMajority()方法

classifyByMajority()方法如示例13-11所示，根据多数原则选择分类。例如，对于一个给定的查询点$r$，如果$k$=6，而且分类为$\{C_1, C_2, C_3, C_3, C_3, C_4\}$，则会根据多数原则选择$C_3$。

示例13-11：classifyByMajority()方法

```
static String classifyByMajority(Map<String, Integer> majority) {
    int votes = 0;
    String selectedClassification = null;
    for (Map.Entry<String, Integer> entry : majority.entrySet()) {
        if (selectedClassification == null) {
            selectedClassification = entry.getKey();
            votes = entry.getValue();
        }
        else {
            int count = entry.getValue();
```

```
            if (count > votes) {
                selectedClassification = entry.getKey();
                votes = count;
            }
        }
    }
    return selectedClassification;
}
```

## 步骤2：处理输入参数

这个步骤如示例13-12所示，将读取4个输入参数：

- k为kNN要用到一个整数。
- d为表示R和S向量维数的一个整数。
- 查询数据集R（一个HDFS文件）。
- 训练数据集S（一个HDFS文件）。

示例13-12：步骤2：处理输入参数

```
// 步骤2: 处理输入参数
if (args.length < 4) {
    System.err.println("Usage: kNN <k-knn> <d-dimension> <R> <S>");
    System.exit(1);
}
Integer k = Integer.valueOf(args[0]); // kNN中的k
Integer d = Integer.valueOf(args[1]); // d-维
String datasetR = args[2];
String datasetS = args[3];
```

## 步骤3：创建一个Spark上下文对象

这一步如示例13-13所示，会创建一个JavaSparkContext对象，这是创建新RDD的工厂对象。

示例13-13：步骤3：创建一个Spark上下文对象

```
// 步骤3: 创建一个Spark上下文对象
JavaSparkContext ctx = createJavaSparkContext("knn");
```

## 步骤4：广播共享对象

为了能够从所有集群节点访问共享对象和数据结构，Spark提供了一个Broadcast类，利用这个类可以注册对象，然后可以从任意节点读取这些对象。在示例13-14中，我们广播了两个整数值（k和d），可以从所有集群节点访问这两个整数值。

示例13-14：步骤4：广播共享对象

```
// 步骤4: 广播共享对象
// 广播k和d作为全局共享对象,
```

```
3   // 这样就可以从所有集群节点访问这两个共享对象
4   final Broadcast<Integer> broadcastK = ctx.broadcast(k);
5   final Broadcast<Integer> broadcastD = ctx.broadcast(d);
```

## 步骤5：为查询和训练数据集创建RDD

如示例13-15所示，这一步会创建两个RDD（一个对应*R*，另一个对应*S*）。这些RDD将原始数据表示为字符串对象。

示例13-15：步骤5：为查询和训练数据集创建RDD

```
1   // 步骤5：为查询和训练数据集创建RDD
2   JavaRDD<String> R = ctx.textFile(datasetR, 1);
3   R.saveAsTextFile("/output/R");
4   JavaRDD<String> S = ctx.textFile(datasetS, 1);
5   S.saveAsTextFile("/output/S");
```

为了调试并帮助理解，这一步会为查询数据集(R)创建以下输出：

```
# hadoop fs -cat /output/R/part*
1000;3.0,3.0
1001;10.1,3.2
1003;2.7,2.7
1004;5.0,5.0
1005;13.1,2.2
1006;12.7,12.7
```

并为训练数据集(S)创建下面的输出：

```
# hadoop fs -cat /output/S/part*
100;c1;1.0,1.0
101;c1;1.1,1.2
102;c1;1.2,1.0
103;c1;1.6,1.5
104;c1;1.3,1.7
105;c1;2.0,2.1
106;c1;2.0,2.2
107;c1;2.3,2.3
208;c2;9.0,9.0
209;c2;9.1,9.2
210;c2;9.2,9.0
211;c2;10.6,10.5
212;c2;10.3,10.7
213;c2;9.6,9.1
214;c2;9.4,10.4
215;c2;10.3,10.3
300;c3;10.0,1.0
301;c3;10.1,1.2
302;c3;10.2,1.0
303;c3;10.6,1.5
304;c3;10.3,1.7
305;c3;10.0,2.1
306;c3;10.0,2.2
307;c3;10.3,2.3
```

## 步骤6：计算(R, S)的笛卡尔积

这个步骤如示例13-16所示，要找出查询数据集(R)与训练数据集(S)的笛卡尔积，将会创建以下RDD：

$$\{(r, s)/r \in R, s \in S\}$$

示例13-16：步骤6：计算（R, S）的笛卡尔积

```
// 步骤6：计算（R, S）的笛卡尔积
//<U> JavaPairRDD<T,U> cartesian(JavaRDDLike<U,?> other)
// 返回这个RDD和另一个RDD的笛卡尔积；
// 也就是说，所有元素对(a, b)的RDD，
// 其中a在this中，b在other中。
JavaPairRDD<String,String> cart = R.cartesian(S);
cart.saveAsTextFile("/output/cart");
```

为了调试并帮助理解，这一步会创建以下输出（为适应版面，这里的代码有所删减）：

```
# hadoop fs -cat /output/cart/part*
(1000;3.0,3.0,100;c1;1.0,1.0)
(1000;3.0,3.0,101;c1;1.1,1.2)
...
(1000;3.0,3.0,306;c3;10.0,2.2)
(1000;3.0,3.0,307;c3;10.3,2.3)
(1001;10.1,3.2,100;c1;1.0,1.0)
(1001;10.1,3.2,101;c1;1.1,1.2)
...
(1001;10.1,3.2,306;c3;10.0,2.2)
(1001;10.1,3.2,307;c3;10.3,2.3)
...
...
(1006;12.7,12.7,306;c3;10.0,2.2)
(1006;12.7,12.7,307;c3;10.3,2.3)
```

## 步骤7：找出R中的r与S中的s之间的距离distance(r, s)

这一步如示例13-17所示，找出各个(R, S)的欧氏距离。根据具体的数据和项目需求，可能会选择和使用不同的距离算法（如Minkowski）。需要指出，距离算法的选择会影响kNN分类的偏差。这一步会创建以下RDD：

$$\{(r, (\text{distance}, \text{classification})\}$$

示例13-17：步骤7：找出R中的r与S中的s之间的距离distance(r, s)

```
// 步骤7：找出R中的r与S中的s之间的距离distance(r, s)
// (K,V)，其中 K = unique-record-id-of-R, V=Tuple2(distance, classification)
// distance = distance(r, s) 这里r属于R，s属于S
// 分类从s提取
JavaPairRDD<String,Tuple2<Double,String>> knnMapped =
        cart.mapToPair(new PairFunction<
                                        Tuple2<String,String>, // 输入
```

```
8                                          String,                 // K
9                                          Tuple2<Double,String>   // V
10                                        >() {
11   public Tuple2<String,Tuple2<Double,String>> call(
12      Tuple2<String,String> cartRecord) {
13           String rRecord = cartRecord._1;
14           String sRecord = cartRecord._2;
15           String[] rTokens = rRecord.split(";");
16           String rRecordID = rTokens[0];
17           String r = rTokens[1]; // r.1, r.2, ..., r.d
18           String[] sTokens = sRecord.split(";");
19           // sTokens[0] = s.recordID
20           String sClassificationID = sTokens[1];
21           String s = sTokens[2]; // s.1, s.2, ..., s.d
22           Integer d = broadcastD.value();
23           double distance = calculateDistance(r, s, d);
24           String K = rRecordID; // r.recordID
25           Tuple2<Double,String> V = new Tuple2<Double,String>(
26           distance, sClassificationID);
27           return new Tuple2<String,Tuple2<Double,String>>(K, V);
28      }
29   });
30   knnMapped.saveAsTextFile("/output/knnMapped");
```

为了调试并帮助理解，这一步会创建以下输出：

```
# hadoop fs -cat /output/knnMapped/part*
(1000,(2.8284271247461903,c1))
(1000,(2.6172504656604803,c1))
...
(1000,(7.045565981523415,c3))
(1000,(7.333484846919642,c3))
(1001,(9.362157870918434,c1))
...
(1001,(0.9219544457292893,c3))
...
(1006,(10.84158659975559,c3))
(1006,(10.673331251301065,c3))
```

## 步骤8：按R中的r对距离分组

得到{(r, s)}的距离后，要找出k个近邻，我们要按r对数据分组（参见示例13-18）。数据按r分组后，可以扫描组值，找出k个最近距离（参见步骤9的讨论）。

示例13-18：步骤8：按R中的r对距离分组

```
1   // 步骤8: 按R中的r对距离分组
2   // 现在将结果按r.recordID分组，然后找出k个近邻。
3   JavaPairRDD<String, Iterable<Tuple2<Double,String>>> knnGrouped =
4     knnMapped.groupByKey();
```

这一步会创建以下RDD：

{(*r*, {(distance, classification)}}

## 步骤9：找出k个近邻并对r分类

这个步骤如示例13-19所示，这里会扫描组值，找出k个近邻。要找到距离最小的k个元素，我们使用了SortedMap，它只保留k个近邻。每次迭代时，我们要确保只保留k个最近的元素。一旦找到k个近邻，可以按多数原则对查询数据分类。

示例13-19：步骤9：找出k个近邻并对r分类

```
// 步骤9: 找出k个近邻并对r分类
// mapValues[U](f: (V) => U): JavaPairRDD[K, U]
// 通过map()函数将各个值传入键-值对RDD，
// 而不改变键；
// 这还会保留原来的RDD分区。
// 生成(K,V)对，这里K=r.recordID, V = classificationID.
JavaPairRDD<String, String> knnOutput =
   knnGrouped.mapValues(new Function<
      Iterable<Tuple2<Double,String>>, // 输入
      String // 输出 (分类)
      >() {
      public String call(Iterable<Tuple2<Double,String>> neighbors) {
          Integer k = broadcastK.value();
          SortedMap<Double, String> nearestK = findNearestK(neighbors, k);
          // 现在nearestK中有k个近邻，
          // 需要统计分类，
          // 按多数原则找出分类
          Map<String, Integer> majority = buildClassificationCount(nearestK);

          // 使用多数投票得到classificationID
          String selectedClassification = classifyByMajority(majority);
          return selectedClassification;
      }
 });
 knnOutput.saveAsTextFile("/output/knnOutput");
```

这一步会创建最终输出，其中包括整个查询数据集的分类：

```
# hadoop fs -cat /output/knnOutput/part*
(1005,c3)
(1001,c3)
(1000,c1)
(1004,c1)
(1006,c2)
(1003,c1)
```

## 合并步骤8和9

在这里，我会展示如何使用`reduceByKey()`或`combineByKey()`将`groupByKey()`和`mapValues()`操作合并到一个步骤中。我们使用了以下RDD以及步骤8和9中的变换：

- RDD：
  - knnMapped: JavaPairRDD<String,Tuple2<Double,String>>

— knnGrouped: JavaPairRDD<String, Iterable<Tuple2<Double,String>>>

— knnOutput: JavaPairRDD<String, String>

- 变换：

— knnMapped --> groupBy() --> knnGrouped

— knnGrouped --> mapValues() --> knnOutput

根据这些RDD，我们无法使用reduceByKey()，因为它要求输出类型（knnOutput.V中的String）等同于输入值的类型（knnMapped.V中的Tuple2<Double,String>）。聚集的返回类型不同于聚集值的类型时就要使用combineByKey()变换。因此，我们将使用combineByKey()把步骤8和9合并到一步。这个合并步骤将使用以下RDD，并实现一个变换：

- RDD：

— knnMapped: JavaPairRDD<String,Tuple2<Double,String>>

— knnOutput: JavaPairRDD<String, String>

- 变换：

— knnMapped --> combineByKey() --> knnOutput

要把groupBy()和mapValues()变换合并为一个变换combineByKey()，需要编写3个基本函数。不过，在此之前，先来看combineByKey()的基本签名：

```
public <C> JavaPairRDD<K,C> combineByKey(
      Function<V,C> createCombiner,
      Function2<C,V,C> mergeValue,
      Function2<C,C,C> mergeCombiners
)

描述：这是一个通用函数，使用一组定制聚集函数为各个键组合相应的元素。
    将一个JavaPairRDD[(K, V)]转换为一个类型为JavaPairRDD[(K, C)]的结果，
    对应一个“组合类型”C * 注意V和C可以不同。例如，可能
    将一个类型为（Int, Int）的RDD分组为一个类型为（Int, List[Int]）的RDD；
    用户提供3个函数：
    - createCombiner，将一个V转换为一个C
    （例如，创建一个单元素列表）
    - mergeValue，将V合并到一个C（例如，把元素增加到列表末尾）
    - mergeCombiners，将两个C合并到一个C中。
```

下面给出使用combineByKey()所需的3个函数：

```
Function<Tuple2<Double,String>, String> createCombiner =
   new Function<Tuple2<Double,String>, String>() {
   @Override
   public String call(Tuple2<Double,String> x) {
      // 这个函数留作练习，有兴趣的读者可以自行完成
```

```
    }
};

Function2<String, Tuple2<Double,String>, String> mergeValue =
new Function2<String, Tuple2<Double,String>, String>() {
    @Override
    public AvgCount call(String a, Tuple2<Double,String> x) {
        //这个函数留作练习，有兴趣的读者可以自行完成
    }
};

Function2<String, String, String> mergeCombiners =
    new Function2<String, String, String>() {
    @Override
    public String call(String a, String b) {
        //这个函数留作练习，有兴趣的读者可以自行完成
    }
};

JavaPairRDD<String, String> knnOutput =
    knnMapped.combineByKey(createCombiner, mergeValue, mergeCombiners);
```

## YARN shell脚本

下面是在YARN环境中运行这个Spark kNN实现的脚本：

```
1 $ cat run_knn.sh
2 #!/bin/bash
3 export JAVA_HOME=/usr/java/jdk7
4 export SPARK_HOME=/usr/local/spark-1.0.0
5 export HADOOP_HOME=/usr/local/hadoop-2.5.0
6 export HADOOP_CONF_DIR=$HADOOP_HOME/etc/hadoop
7 export YARN_APPLICATION_CLASSPATH=$HADOOP_CONF_DIR
8 BOOK_HOME=/mp/data-algorithms-book
9 APP_JAR=$BOOK_HOME/dist/data_algorithms_book.jar
10 #
11 k=4
12 d=2
13 R=/knn/R.txt
14 S=/knn/S.txt
15 prog=org.dataalgorithms.chap13.spark.kNN
16 $SPARK_HOME/bin/spark-submit
17 --class $prog \
18 --master yarn-cluster \
19 --num-executors 12 \
20 --driver-memory 3g \
21 --executor-memory 7g \
22 --executor-cores 12 \
23 $APP_JAR $k $d $R $S
```

这一章使用Spark实现了一个kNN分布式算法。利用Spark的高层抽象，我们可以采用一种直接的方式表述这个算法。下一章将提供一个分布式朴素贝叶斯（Naive Bayes）算法，这是最重要的分类技术之一。

# 第14章

# 朴素贝叶斯

在数据挖掘和机器学习中，有很多分类算法。其中朴素贝叶斯分类器（Naive Bayes classifier，NBC）（*http://bit.ly/naive_bayes*）是最简单但最有效的算法之一。这一章的重点就是提供NBC的一个分布式MapReduce实现（使用Spark），这个实现结合了监督学习方法和概率分类器。朴素贝叶斯（Naive Bayes）是一个线性分类器。要理解这个概念，首先需要了解一些基本和条件概率。处理数值数据时，最好使用聚类技术（如K-均值（*http://bit.ly/k-means_clustering*）和k-近邻（*http://bit.ly/k-nearest_algorithm*）方法和算法），不过对于名字、符号、电子邮件和文本的分类，则最好使用概率方法，如NBC。在某些情况下，NBC也可以用来对数值数据分类。在下一节中，你会看到符号和数值数据的例子。

NBC是一个基于强（朴素）独立假设应用贝叶斯定理实现的概率分类器。基本说来，NBC根据输入的一些属性（特性）将输入分配到k个类$\{C_1, C_2, ..., C_k\}$中的某一类。NBC有很多应用，如垃圾邮件过滤和文档分类。

例如，使用朴素贝叶斯分类器的垃圾邮件过滤器将把各个电子邮件分配到两个簇之一：垃圾邮件（spam mail）或非垃圾邮件（not a spam mail）。由于朴素贝叶斯是一个监督型学习方法，它有两个不同的阶段：

*阶段1：训练（见图14-1）*

这个阶段使用一个有限的数据样本实例集合中的训练数据建立一个分类器（将在阶段2中使用）。这就是所谓的监督型学习方法（supervised learning method），即从一个样本集学习，然后使用这个信息来完成新数据分类。

*阶段2: 分类（见图14-2）*

在这个阶段中，我们使用训练数据和贝叶斯定理将新数据分类到阶段1中明确的某一个类别中。

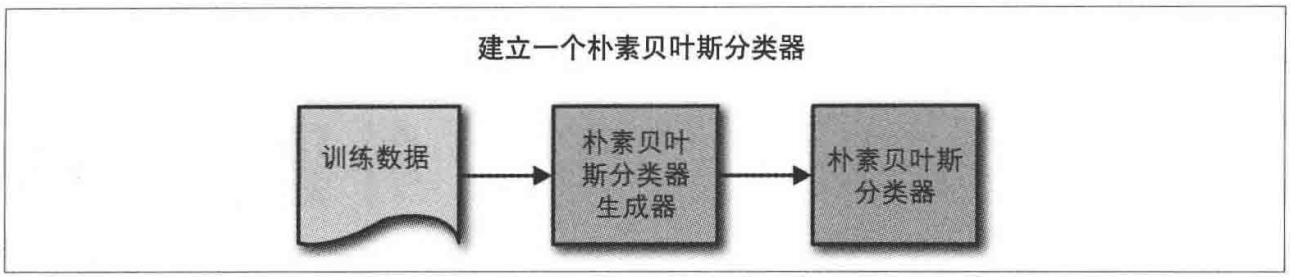



图14-1：朴素贝叶斯：训练阶段

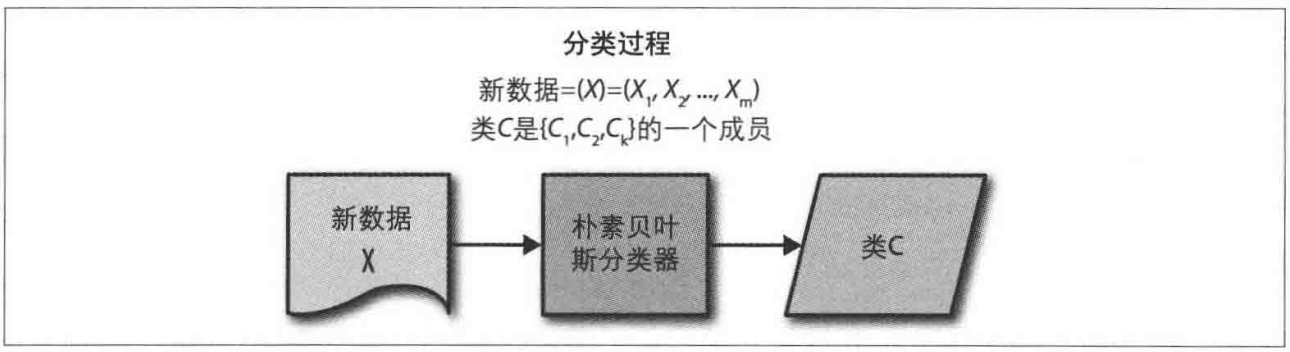



图14-2：朴素贝叶斯：分类

# 训练和学习示例

令各个数据集有m个属性［$X = (X_1, X_2, ..., X_m)$］，训练数据的大小为$n$，另外将有$k$个不同的类别$\{C_1, C_2, ..., C_k\}$，其中 $k \leqslant n$。则有：

$(X_{11}, X_{12}, ..., X_{1m}), (C_1)$

$(X_{21}, X_{22}, ..., X_{2m}), (C_2)$

...

$(X_{n1}, X_{n2}, ..., X_{nm}), (C_n)$

这里，$C_i$是$\{C_1, C_2, ..., C_k\}$的一个成员。

## 数值训练数据

表14-1是数值训练数据的一个例子。需要说明，第一列（Person）是元数据（也就是说，它不是具体数据的一部分）。

表14-1：数值训练数据示例

| 人 | 身高（英尺） | 体重（磅） | 脚长（英寸） | 性别（分类） |
|---|---|---|---|---|
| 1 | 6.00 | 180 | 12 | 男 |
| 2 | 5.92 | 190 | 11 | 男 |
| 3 | 5.58 | 170 | 12 | 男 |
| 4 | 5.92 | 165 | 10 | 男 |
| 5 | 5.00 | 100 | 6 | 女 |
| 6 | 5.50 | 150 | 8 | 女 |
| 7 | 5.42 | 130 | 7 | 女 |
| 8 | 5.75 | 150 | 9 | 女 |

下面是关于这个数值训练数据的一些性质：

- Gender列是分类列。这里有两个（$k = 2$）类别{male, female}，其中 $C_1 = male$，$C_2 = female$。
- 每个数据集有3个属性（$m = 3$）：身高（`height`）、体重（`weight`）和脚长（`foot size`）。
- 我们认为每个数据实例是一个$m$-维属性值向量：$X = (X_1, X_2, ..., X_m)$。
- 训练数据大小为8($n = 8$)，由第1列标识（列号分别为1、2、...、8）。
- 我们的目标是用这个训练数据建立一个分类系统（使用贝叶斯理论），这个系统能帮我们确定一个人的性别。这个分类要基于3个属性（身高（`height`）、体重（`weight`）和脚长（`foot size`））的值来完成。

## 符号训练数据

表14-2是符号训练数据的一个例子，选自Tom Mitchell的《Machine Learning》[19]。注意第一列是元数据（也就是说，这不是具体数据的一部分）。

表14-2：示例符号训练数据

| 日期 | 天气 | 温度 | 湿度 | 风力 | 是否可以打网球（分类） |
|---|---|---|---|---|---|
| $D_1$ | Sunny | Hot | High | Weak | No |
| $D_2$ | Sunny | Hot | High | Strong | No |
| $D_3$ | Overcast | Hot | High | Weak | Yes |
| $D_4$ | Rain | Mild | High | Weak | Yes |
| $D_5$ | Rain | Cool | Normal | Weak | Yes |
| $D_6$ | Rain | Cool | Normal | Strong | No |

表14-2：示例符号训练数据（续）

| 日期 | 天气 | 温度 | 湿度 | 风力 | 是否可以打网球（分类） |
|---|---|---|---|---|---|
| $D_7$ | Overcast | Cool | Normal | Strong | Yes |
| $D_8$ | Sunny | Mild | High | Weak | No |
| $D_9$ | Sunny | Cool | Normal | Weak | Yes |
| $D_{10}$ | Rain | Mild | Normal | Weak | Yes |
| $D_{11}$ | Sunny | Mild | Normal | Strong | Yes |
| $D_{12}$ | Overcast | Mild | High | Strong | Yes |
| $D_{13}$ | Overcast | Hot | Normal | Weak | Yes |
| $D_{14}$ | Rain | Mild | High | Strong | No |

关于这个符号训练数据有以下性质：

- “是否可以打网球”（PlayTennis）列是分类列。这里有两个（$k = 2$）类别：{Yes, No}，其中 $C_1$ = Yes ，$C_2$ = No。
- 每个数据集有4个属性（$m = 4$）：天气（outlook）、温度（temperature）、湿度（humidity）和风力（wind）。我们认为各个数据实例是一个$m$维属性值向量：$X = (X_1, X_2, ..., X_m)$。
- 训练数据大小为14 ($n = 14$)，由第1列标识 $\{D_1, D_2, ..., D_{14}\}$。
- 我们的目标是使用这个训练数据建立一个分类系统（使用贝叶斯理论），这个分类系统会帮助我们根据天气条件确定是否可以打网球（也就是说，我们希望对应PlayTennis把数据分为两类：Yes或No）。这个分类要基于4个属性的值：天气（outlook）、温度（temperature）、湿度（humidity）和风力（wind）。

现在真正的问题是，如果输入数据如下：

```
X = (X1 = u1, X2=u2, X3=u3, X4=u4)
  = (Outlook = Overcast,
     Temperature = Hot,
     Humidity = High,
     Wind = Strong)
```

那么PlayTennis分类为Yes还是No呢？朴素贝叶斯分类器（使用贝叶斯理论）可以根据训练/学习阶段给定的数据回答这个问题。在这里，“朴素”（naive）是指我们认为数据属性的所有概率都是独立的；也就是说，对于一个给定的数据集$X = (X_1, X_2, ..., X_m)$，我们假设这些属性相互独立，因此每个属性的概率也是独立的。这是一个非常强（“朴素”）的假设。有些统计学家会有异议，认为使用NBC是有问题的，因为这个“朴素”的独立性假设在真实世界中几乎都是不可行的。不过，事实上可以看到，朴素贝叶斯方法在大量实际应用中都能很好地使用。

# 条件概率

由于朴素贝叶斯基于概率分类（特别是条件概率），所以这里对概率和条件概率做一个很简短的介绍。给定事件B，事件A的条件概率（表示为$P$）定义如下：

$$P(A|B)=\frac{P(A\cap B)}{P(B)}$$

另外，事件$A$和$B$是独立的，当且仅当：

$$P(A \cap B) = P(A)P(B)$$

# 深入分析朴素贝叶斯分类器

前面已经了解到，朴素贝叶斯分类器是一个简单而且稳定的分类和预测方法。下面给出贝叶斯理论的一般形式（根据贝叶斯统计），这是这个分类器的基础。令$A$是一个互斥事件序列$\{A_1, A_2, ..., A_n\}$，其并集是整个样本空间，另外令E是某个事件。假设所有事件概率均不为0($P(E) > 0$，而且对于所有$i$，$P(A_i) > 0$)。贝叶斯理论指出：

对于所有$j \in \{1, 2, ..., n\}$

$$P(A_j|E)=\frac{P(A_j)P(E|A_j)}{\sum_{i=1}^{n}P(A_i)P(E|A_i)}$$

贝叶斯理论的一个更简单的形式化表示可以表述为：令A和B是两个事件（属性值在一个给定的统计空间中）。那么，贝叶斯理论指定了A和B的概率［也就是说，$P(A)$和$P(B)$］与给定$B$时$A$的条件概率［表示为$P(A|B)$］和给定$A$时$B$的条件概率（表示为$P(B|A)$）之间的关系。贝叶斯理论可以表示如下（对于两个事件$A$和$B$，这里假设它们的概率都不为0）：

$$P(A|B)=\frac{P(B|A)P(A)}{P(B)}$$

关于贝叶斯理论的更多详细信息，参见[4]。

接下来给出贝叶斯理论用于分类的形式化描述：令$X = (X_1 = u_1, ..., X_m = u_m)$是一个需要分类的数据实例，另外令$C = \{C_1, C_2, ..., C_k\}$是不同类别的一个有限集合（由训练数据生成/归约）。然后，使用贝叶斯理论，可以预测一个给定$X$的类别$C^{\text{predict}} \in \{C_1, C_2, ..., C_k\}$：

$$C^{\text{predict}} = \arg \max_c P(C=c \mid X_1=u_1,...,X_m=u_m)$$
$$= \arg \max_c \frac{P(C=c, X_1=u_1,...,X_m=u_m)}{P(X_1=u_1,...,X_m=u_m)}$$
$$= \arg \max_c \frac{P(X_1=u_1,...,X_m=u_m \mid C=c)P(C=c)}{P(X_1=u_1,...,X_m=u_m)}$$
$$= \arg \max_c P(X_1=u_1,...,X_m=u_m \mid C=c)P(C=c)$$
$$= \arg \max_c P(C=c)\prod_{j=1}^{m} P(X_j=u_j \mid C=c)$$

需要说明，我们从分类算法中去掉了分母$P(X_1 = u_1, ..., X_m = u_m)$，因为对于所有计算这都是一个常量，它不会改变分类算法的结果。因此，从这个概率模型构建分类器时，可以表述如下：

$$classify(X_1=u_1,..., X_m=u_m)=C^{\text{predict}}$$
$$C^{\text{predict}} = \arg \max_c P(C=c)\prod_{j=1}^{m} P(X_j=u_j \mid C=c)$$

## 朴素贝叶斯分类器示例

在这个NBC示例中，我们想知道如何对以下输入数据分类：

```
X = (Outlook = Overcast,
     Temperature = Hot,
     Humidity = High,
     Wind = Strong)
     X = (Overcast, Hot, High, Strong)
     X = (X1, X2, X3, X4)
```

答案是Yes（可以打网球）还是No（不能打网球）？对于这个例子，我们有两个类：

- $C = (C_1, C_2)$ = (Yes, No)
- $P(C_1) = P$(Yes) = 9/14
- $P(C_2) = P$(No) = 5/14

现在，根据贝叶斯分类器，可以有：

$$C^{\text{predict}} = \arg \max_c P(C=c)\prod_{j=1}^{m} P(X_j=u_j \mid C=c) = \max\{V_1, V_2\}$$

在这里：

$V_1=\{P(C=C_1)P(X_1|C=C_1)P(X_2|C=C_1)P(X_3|C=C_1)P(X_4|C=C_1)\}$

$V_2=\{P(C=C_2)P(X_1|C=C_2)P(X_2|C=C_2)P(X_3|C=C_2)P(X_4|C=C_2)\}$

如果$V_1 > V_2$，则X的分类为$C_1$ = Yes；否则，分类为$C_2$ = No。

下面是$C_1$ = Yes的条件概率计算：

$P(X_1|C = C_1) = P(\text{“}Overcast\text{”}|C = \text{Yes}) = ?$

在$PlayTennis$=Yes的9种情况中，有4种情况Outlook=“Overcast”；因此，$P$ ($Outlook$=“$Overcast$”|$PlayTennis$=$Yes$)=4/9。采用这个公式的记法，可以写为$P(X_1$= $Overcast|C_1) = 4/9$。

$P(X_2|C=C_1) = P(\text{“}Hot\text{”}|C=\text{Yes}) = ?$

在$PlayTennis$=Yes的9种情况中，有2种情况$Temperature$=“Hot”；因此，$P$($Temperature$=“$Hot$”|$PlayTennis$=$Yes$)=2/9。采用这个公式的记法，可以写为$P(X_2$=$Hot|C_1)$=2/9。

$P(X_3|C = C_1) = P(\text{“}High\text{”}|C = \text{Yes}) = ?$

在$PlayTennis$ = Yes的9种情况中，有3种情况$Humidity$=“High”；因此，$P$($Humidity$ = “$High$”|$PlayTennis$ = $Yes$) = 3/9。采用这个公式的记法，可以写为$P(X_3 = High|C_1)$ = 3/9。

$P(X_4|C = C_1) = P(\text{“}Strong\text{”}|C = \text{Yes}) = ?$

在$PlayTennis$=Yes的9种情况中，有3种情况$Wind$=“$Strong$”；因此，$P$($Wind$ = “$Strong$”|$PlayTennis$=$Yes$) = 3/9。采用这个公式的记法，可以写为$P(X_4$=$Strong|C_1)$ = 3/9。

下面是$C_2$ = No的条件概率计算：

$P(X_1|C = C_2) = P(\text{“}Overcast\text{”}|C = \text{No}) = ?$

在$PlayTennis$=No的5种情况中，有0种情况$Outlook$=“$Overcast$”；因此，$P$($Outlook$ = “$Overcast$”|$PlayTennis$=$No$)=0/5。采用这个公式的记法，可以写为$P(X_1$ = $Overcast|C_2)$ = 0/5。

$P(X_2|C = C_2) = P(\text{“}Hot\text{”}|C = \text{No}) = ?$

在$PlayTennis$=No的5种情况中，有0种情况$Temperature$=“$Hot$”；因此，$P$($Temperature$=“$Hot$”|$PlayTennis$=$No$)=2/5。采用这个公式的记法，可以写为$P(X_2$=$Hot|C_2)$ = 2/5。

$P(X_3|C = C_2) = P(\text{“}High\text{”}|C = \text{No}) = ?$

在$PlayTennis$=No的5种情况中，有4种情况$Humidity$=“$High$”；因此，$P$($Humidity$ = “$High$”|$PlayTennis$ = $No$) = 4/5。采用这个公式的记法，可以写为$P(X_3$=$High|C_2)$ = 4/5。

$P(X_4|C = C_2) = P(\text{“Strong”}|C = \text{No}) = ?$

在$PlayTennis$=No的5种情况中，有3种情况$Wind$=“Strong”；因此，$P(Wind =$ “$Strong$”$|PlayTennis=No)$=3/5。采用这个公式的记法，可以写为$P(X_4 = Strong|C_2) =$ 3/5。

插入这些值，可以得到：

$$V_1=\left(\frac{9}{14}\right)\left(\frac{4}{9}\right)\left(\frac{2}{9}\right)\left(\frac{3}{9}\right)\left(\frac{3}{9}\right)=\frac{648}{91854}$$

$$V_2=\left(\frac{5}{14}\right)\left(\frac{0}{5}\right)\left(\frac{2}{5}\right)\left(\frac{4}{5}\right)\left(\frac{3}{5}\right)=0$$

由于$V_1 > V_2$,我们将$X$分类为$C_1$ = Yes。

# 朴素贝叶斯分类器：符号数据的MapReduce解决方案

这一节会提供一个MapReduce解决方案，可以对上百万甚至数十亿符号（非数值）数据完成分类。要建立MapReduce解决方案，这里假设已经有适当的训练数据（可以基于这些训练数据使用贝叶斯理论计算概率和条件概率）。这个MapReduce解决方案的目标是将数据分类到一个明确定义的类集合（由训练数据定义），其中包含k个不同的类，分别为 $\{C_1, C_2, ..., C_k\}$ 。

## 阶段1：使用符号训练数据建立分类器

这个阶段的目标是使用训练数据建立一个分类器函数，它接受一个数据实例$X = (X_1 = u_1, ..., X_m = u_m)$，输出一个类$c \in \{C_1, C_2, ..., C_k\}$ (假设所有训练数据映射到其中一个类)。所以，要为训练数据中所有不同$X_i$以及所有不同$C_j(j = 1, 2, ..., k)$计算$P(X_i = u_i|C = C_j)$ 。训练数据集很小时，可以写一个非MapReduce程序来建立分类器，不过，对于很大的训练数据集，就要写一个MapReduce作业计算$P(X_i = u_i|C = C_j)$。

示例14-1定义了建立分类器的map()函数。

示例14-1：建立分类器的map()函数

```
1 /**
2  * @param key由MapReduce框架生成，在这里忽略
3  * @param value是一个String，有以下格式：
4  * <Data1><,><Data2><,><...><DataM><,><Class>
5  */
6 map(key, value) {
```

```
 7    String[] tokens = value.split(",");
 8    int classIndex = tokens.length -1;
 9    String theClass = tokens[classIndex];
10    for(int i=0, i < (classIndex-1); i++) {
11       String reducerKey = tokens[i] + "," + theClass;
12       emit(reducerKey, 1);
13    }
14
15    String reducerKey = "CLASS," + theClass;
16    emit(reducerKey, 1);
17 }
```

为了理解这个映射器，接下来我们对所有训练数据应用这个map()函数，这会生成一组输入，由reduce()函数处理。这个map()函数只是统计这些属性以及它们与分类类别之间的关联。

map(Sunny, Hot, High, Weak, No)将生成：

```
<Sunny,No>, <1>
<Hot,No>, <1>
<High,No>, <1>
<Weak,No>, <1>
<CLASS,No>, <1>
```

map(Sunny, Hot, High, Strong, No)将生成：

```
<Sunny,No>, <1>
<Hot,No>, <1>
<High,No>, <1>
<Strong,No>, <1>
<CLASS,No>, <1>
```

map(Overcast, Hot, High, Weak, Yes)将生成：

```
<Overcast,Yes>, <1>
<Hot,Yes>, <1>
<High,Yes>, <1>
<Weak,Yes>, <1>
<CLASS,Yes>, <1>
```

map(Rain, Mild, High, Weak, Yes)将生成：

```
<Rain,Yes>, <1>
<Mild,Yes>, <1>
<High,Yes>, <1>
<Weak,Yes>, <1>
<CLASS,Yes>, <1>
```

map(Rain, Cool, Normal, Weak, Yes)将生成：

```
<Rain,Yes>, <1>
<Cool,Yes>, <1>
```

```
<Normal,Yes>, <1>
<Weak,Yes>, <1>
<CLASS,Yes>, <1>
```

map(Rain, Cool, Normal, Strong, No)将生成：

```
<Rain,No>, <1>
<Cool,No>, <1>
<Normal,No>, <1>
<Strong,No>, <1>
<CLASS,No>, <1>
```

map(Overcast, Cool, Normal, Strong, Yes)将生成：

```
<Overcast,Yes>, <1>
<Cool,Yes>, <1>
<Normal,Yes>, <1>
<Strong,Yes>, <1>
<CLASS,Yes>, <1>
```

map(Sunny, Mild, High, Weak, No)将生成：

```
<Sunny,No>, <1>
<Mild,No>, <1>
<High,No>, <1>
<Weak,No>, <1>
<CLASS,No>, <1>
```

map(Sunny, Cool, Normal, Weak, Yes)将生成：

```
<Sunny,Yes>, <1>
<Cool,Yes>, <1>
<Normal,Yes>, <1>
<Weak,Yes>, <1>
<CLASS,Yes>, <1>
```

map(Rain, Mild, Normal, Weak, Yes)将生成：

```
<Rain,Yes>, <1>
<Mild,Yes>, <1>
<Normal,Yes>, <1>
<Weak,Yes>, <1>
<CLASS,Yes>, <1>
```

map(Sunny, Mild, Normal, Strong, Yes)将生成：

```
<Sunny,Yes>, <1>
<Mild,Yes>, <1>
<Normal,Yes>, <1>
<Strong,Yes>, <1>
<CLASS,Yes>, <1>
```

map(Overcast, Mild, High, Strong, Yes)将生成：

```
<Overcast,Yes>, <1>
<Mild,Yes>, <1>
<High,Yes>, <1>
<Strong,Yes>, <1>
<CLASS,Yes>, <1>
```

map(Overcast, Hot, Normal, Weak, Yes)将生成：

```
<Overcast,Yes>, <1>
<Hot,Yes>, <1>
<Normal,Yes>, <1>
<Weak,Yes>, <1>
<CLASS,Yes>, <1>
```

最后map(Rain, Mild, High, Strong, No)将生成：

```
<Rain,No>, <1>
<Mild,No>, <1>
<High,No>, <1>
<Strong,No>, <1>
<CLASS,No>, <1>
```

示例14-2给出了建立这个分类器的归约器。由于我们只对值（计数）完成聚集，reduce()函数也可以用作为一个组合器。对于这个例子，这个归约器将接收表14-3中所示的键-值对。

**表14-3：归约器输入**

| 键 | 值 |
|---|---|
| <CLASS,No> | [<1>,<1>,<1>,<1>,<1>] |
| <CLASS,Yes> | [<1>,<1>,<1>,<1>,<1>,<1>,<1>,<1>,<1>] |
| <Cool,No> | [<1>] |
| <Cool,Yes> | [<1>,<1>,<1>] |
| <High,No> | [<1>,<1>,<1>,<1>] |
| <High,Yes> | [<1>,<1>,<1>] |
| <Hot,No> | [<1>,<1>] |
| <Hot,Yes> | [<1>,<1>] |
| <Mild,No> | [<1>,<1>] |
| <Mild,Yes> | [<1>,<1>,<1>,<1>] |
| <Normal,No> | [<1>] |
| <Normal,Yes> | [<1>,<1>,<1>,<1>,<1>,<1>] |
| <Overcast,Yes> | [<1>,<1>,<1>,<1>] |

表14-3：归约器输入（续）

| 键 | 值 |
| --- | --- |
| <Rain,No> | [<1>,<1>] |
| <Rain,Yes> | [<1>,<1>,<1>] |
| <Strong,No> | [<1>,<1>,<1>] |
| <Strong,Yes> | [<1>,<1>,<1>] |
| <Sunny,No> | [<1>,<1>,<1>] |
| <Sunny,Yes> | [<1>,<1>] |
| <Weak,No> | [<1>,<1>] |
| <Weak,Yes> | [<1>,<1>,<1>,<1>,<1>,<1>] |

示例14-2：建立分类器的reduce()函数

```
/**
 * @param key为<Data, Class>或<CLASS, Class>
 * @param values是一个整数列表
 */
reduce(key, values) {
    int total = 0;
    for(int value : values) {
        total += value;
    }

    emit(key, total);
}
```

这个reduce()函数只是将频度累加（基本说来，它相当于计数器）。这个归约器会生成表14-4所示的输出。

表14-4：归约器输出

| 键 | 值 |
| --- | --- |
| <CLASS,No> | 5 |
| <CLASS,Yes> | 9 |
| <Cool,No> | 1 |
| <Cool,Yes> | 3 |
| <High,No> | 4 |
| <High,Yes> | 3 |
| <Hot,No> | 2 |
| <Hot,Yes> | 2 |
| <Mild,No> | 2 |
| <Mild,Yes> | 4 |

表14-4：归约器输出（续）

| 键 | 值 |
| --- | --- |
| <Normal,No> | 1 |
| <Normal,Yes> | 6 |
| <Overcast,Yes> | 4 |
| <Rain,No> | 2 |
| <Rain,Yes> | 3 |
| <Strong,No> | 3 |
| <Strong,Yes> | 3 |
| <Sunny,No> | 3 |
| <Sunny,Yes> | 2 |
| <Weak,No> | 2 |
| <Weak,Yes> | 6 |

我们将定制这个归约器来生成两类输出：

- 类输出。
- 条件概率输出。

类输出如表14-5所示。

表14-5：类输出

| 键 | 值 |
| --- | --- |
| <CLASS,No> | 5 |
| <CLASS,Yes> | 9 |

条件概率输出如表14-6所示。

表14-6：条件概率输出

| 键 | 值 |
| --- | --- |
| <Cool,No> | 1 |
| <Cool,Yes> | 3 |
| <High,No> | 4 |
| <High,Yes> | 3 |
| <Hot,No> | 2 |
| <Hot,Yes> | 2 |
| <Mild,No> | 2 |

表14-6：条件概率输出（续）

| 键 | 值 |
|---|---|
| <Mild,Yes> | 4 |
| <Normal,No> | 1 |
| <Normal,Yes> | 6 |
| <Overcast,Yes> | 4 |
| <Rain,No> | 2 |
| <Rain,Yes> | 3 |
| <Strong,No> | 3 |
| <Strong,Yes> | 3 |
| <Sunny,No> | 3 |
| <Sunny,Yes> | 2 |
| <Weak,No> | 2 |
| <Weak,Yes> | 6 |

从这两类归约器输出（类输出和条件概率输出）可以生成一个最终的概率表，下面将使用这个概率表对新数据进行分类（见表14-7）。

表14-7：概率表

| 键 | 概率 |
|---|---|
| <CLASS,No> | 5/14 |
| <CLASS,Yes> | 9/14 |
| <Cool,No> | 1/5 |
| <Cool,Yes> | 3/9 |
| <High,No> | 4/5 |
| <High,Yes> | 3/9 |
| <Hot,No> | 2/5 |
| <Hot,Yes> | 2/9 |
| <Mild,No> | 2/5 |
| <Mild,Yes> | 4/9 |
| <Normal,No> | 1/5 |
| <Normal,Yes> | 6/9 |
| <Overcast,Yes> | 4/9 |
| <Rain,No> | 2/5 |
| <Rain,Yes> | 3/9 |
| <Strong,No> | 3/5 |

表14-7：概率表（续）

| 键 | 概率 |
|---|---|
| <Strong,Yes> | 3/9 |
| <Sunny,No> | 3/5 |
| <Sunny,Yes> | 2/9 |
| <Weak,No> | 2/5 |
| <Weak,Yes> | 6/9 |

如果某个键不在我们的概率表中，则其概率为0。一般地，训练数据应当涵盖所有属性，而且要保证不会发生概率为0的情况，不过这个需求实际上依赖于具体的项目和数据挖掘需求。

## 阶段2：使用分类器对新符号数据分类

既然已经建立了分类器（也就是生成了概率表），下面可以用它对新数据进行分类。我们的目标是将$X = \{X_1 = u_1, X_2 = u_2, ..., X_m = u_m\}$分类到$\{C_1, C_2, ..., C_k\}$中的某个类中。这里假设MapReduce输入的各个记录分别是需要分类的一个数据实例。

完成分类的map()函数是一个恒等映射器，可以帮助我们避免将相同的数据分类多次。由于分类是一个开销很昂贵的操作，我们要消除重复的计算。既然已经由给定的训练数据建立了概率表，下面由归约器完成所有分类。

对新符号数据分类的map()函数如示例14-3所示。

示例14-3：朴素贝叶斯分类器的map()函数

```
1 public class NaiveBayesClassifierMapper ... {
2    ...
3    /**
4     * @param key由MapReduce框架生成，在这里忽略
5     * @param value是一个String ，有以下格式：
6     * X = (X1, X2, ..., Xm) = <Data1><,><Data2><,><...><,><DataM>
7     */
8    map(key, value) {
9       // 如果需要，可以得出
10      // 每个输入数据重复多少次
11      emit(value, 1);
12   }
13   ...
14 }
```

对新符号数据分类的reduce()函数如示例14-4所示。

示例14-4：朴素贝叶斯分类器的reduce()

```
1 public class NaiveBayesClassifierReducer ... {
```

```
2
3     private theProbabilityTable= ...;
4     // classifications = {C1, C2, ..., Ck}
5     private List<String> classifications = ...;
6     public void setup() {
7        theProbabilityTable = buildTheProbabilityTable();
8        classifications = buildClassifications();
9     }
10
11    /**
12     * @param key为X = (X1, X2, ..., Xm)
13     * = <Data1><,><Data2><,><...><,><DataM>
14     *
15     * @param values是一个整数列表（显示重复记录）
16     *
17     */
18    reduce(key, values) {
19       // key = (X1, X2, ..., Xm)
20       String[] attributes = key.split(",");
21       String selectedClass = null;
22       double maxPosterior = 0.0;
23       for(String aClass : classifications) {
24          double posterior = theProbabilityTable.getClassProbability(aClass);
25          for (int i=0; i < attributes.length; i++) {
26              posterior *= theProbabilityTable.getConditionalProbability(
27                    attributes[i], aClass);
28          }
29          if (selectedClass == null) {
30             // 计算第一个分类的值
31             selectedClass = aClass;
32             maxPosterior = posterior;
33          }
34          else {
35             if (posterior > maxPosterior) {
36                 selectedClass = aClass;
37                 maxPosterior = posterior;
38             }
39          }
40       }
41
42       reducerOutputValue = selectedClass + "," + maxPosterior;
43       emit(key, reducerOutputValue);
44    }
45 }
```

## 朴素贝叶斯分类器：数值数据的MapReduce解决方案

这一节会提供一个MapReduce解决方案，可以对上百万甚至数十亿数值或所谓的连续（continuous）数据进行分类。要建立这个MapReduce解决方案，我们假设已经有适当的训练数据（可以基于这些训练数据使用贝叶斯理论计算概率和条件概率）。这个MapReduce解决方案的目标是将数据分类到一个明确定义的类集合（由训练数据定义），其中包含$k$个不同的类，分别为$\{C_1, C_2, ..., C_k\}$。

要处理数值数据，我们要计算训练数据的均值和方差。然后在分类器中使用这些值（均值和方差）对新的数值数据进行分类。在这个数值示例中，我们有两个类$\{C_1, C_2\} = \{male, female\}$。下面给出由采用高斯分布的训练集创建的分类器，如表14-8所示。

**表14-8：使用高斯分布的训练集分类器**

| 性别(类) | 均值(身高) | 方差(身高) | 均值(体重) | 方差(体重) | 均值(脚长) | 方差(脚长) |
|---|---|---|---|---|---|---|
| Male | 5.8550 | 0.035033 | 176.25 | 122.92 | 11.25 | 0.9167 |
| Female | 5.4175 | 0.097225 | 132.50 | 558.33 | 7.50 | 1.6667 |

由于训练数据中有4个男性和4个女性，所以类是等概率的：$P(male) = P(female) = 0.5$。

对于数值数据（连续属性，如身高、体重和脚长），建议像这里一样采用高斯分布。令$x$是一个连续属性（也就是数值）。首先，按类对数据分段，然后计算各个类中$x$的均值($\mu$)和方差($\sigma^2$)。条件概率的高斯正态分布可以表示为：

$$P(x=v \mid c)=\frac{!}{\sigma_c\sqrt{2\pi}}e^{-\frac{(v-\mu_c)^2}{(2\sigma_c^2)}}$$

在这里：

- $\mu_c$是与类$c$关联的$x$中所有值的均值。
- $\sigma^2_c$是与类$c$关联的$x$中所有值的方差（$\sigma_c$是与类$c$关联的$x$中所有值的标准差）。

下面对新数据分类，如表14-9所示。

**表14-9：新数据分类**

| 身高(英尺) | 体重(磅) | 脚长(英寸) | 性别 |
|---|---|---|---|
| 6.00 | 130 | 8 | ? |

在贝叶斯统计中，后验概率（posterior probability）是给定证据X时参数T的概率，可以写为：

$$P(T \mid X)$$

我们的目标是将这个数据分类为男/女（也就是说，我们希望确定哪一个后验概率更大，是男还是女）。根据贝叶斯理论，可以表示为：

```
posterior(male) = evidenceMale / evidence
posterior(female) = evidenceFemale / evidence
```

evidenceMale、evidenceFemale和evidence变量（也称为归一化常量（normalizing constants））可以如下计算，因为后验概率之和等于1：

```
evidenceMale = P(male) *
               P(height|male) *
               P(weight|male) *
               P(footsize|male)
evidenceFemale = P(female) *
                 P(height|female) *
                 P(weight|female) *
                 P(footsize|female)

evidence = evidenceMale + evidenceFemale
```

evidence可以忽略，因为这是一个正常量。现在可以确定这个样本的性别（分类）：

```
P(male) = 0.5
P(height|male) = 1.5789 (概率密度大于1.00是有可能的。
                         这是等于1.00的
                         钟形曲线以下的区域)
P(weight|male) = 5.9881e-06
P(footsize|male) = 1.3112e-3

posterior numerator (male) =
                              P(male) *
                              P(height|male) *
                              P(weight|male) *
                              P(footsize|male)
                            = 6.1984e-09
P(female) = 0.5
P(height|female) = 2.2346e-1
P(weight|female) = 1.6789e-2
P(footsize|female) = 2.8669e-1
posterior numerator (female) =
                                P(female) *
                                P(height|female) *
                                P(weight|female) *
                                P(footsize|female)
                              = 5.3778e-04
```

由于posterior numerator (female) > posterior numerator (male)，可以得出结论，这个样本是女性。

# 朴素贝叶斯分类器Spark实现

Spark实现包括两个阶段，如图14-3所示。

*阶段1：使用训练数据建立一个朴素贝叶斯分类器*

这个任务使用BuildNaiveBayesClassifier类来完成。这个类读取训练数据，建立朴素贝叶斯分类器。令$C = \{C_1, C_2, ..., C_k\}$是一个分类集合，我们的训练数据定义如下：

- 令$X = \{X_1, X_2, ..., X_n\}$是原始数据。
- 每个$X_i$有$m$个属性，可以分类为：

$$X_1 = \{X_{11}, X_{12}, ..., X_{1m}\} \rightarrow c_1 \in C$$

$$X_2 = \{X_{21}, X_{22}, ..., X_{2m}\} \rightarrow c_2 \in C$$

$$...$$

$$X_n = \{X_{n1}, X_{n2}, ..., X_{nm}\} \rightarrow c_n \in C$$

- 我们的目标是创建以下概率表（ProbabilityTable）函数：

```
ProbabilityTable(Ci) = pValue
ProbabilityTable(Aj, Ci) = pValue
其中Aj是X的一个属性，而且0.00 ≤ pValue ≤ 1.00
```

*阶段2：使用新建立的NBC对新数据分类*

一旦建立了朴素贝叶斯分类器，可以使用`ProbabilityTable()`函数对新数据分类。这个任务使用`NaiveBayesClassifier`类来完成。这个类会读取分类器（表示为`ProbabilityTable()`函数）和新数据，并使用读入的分类器对这些新数据分类。

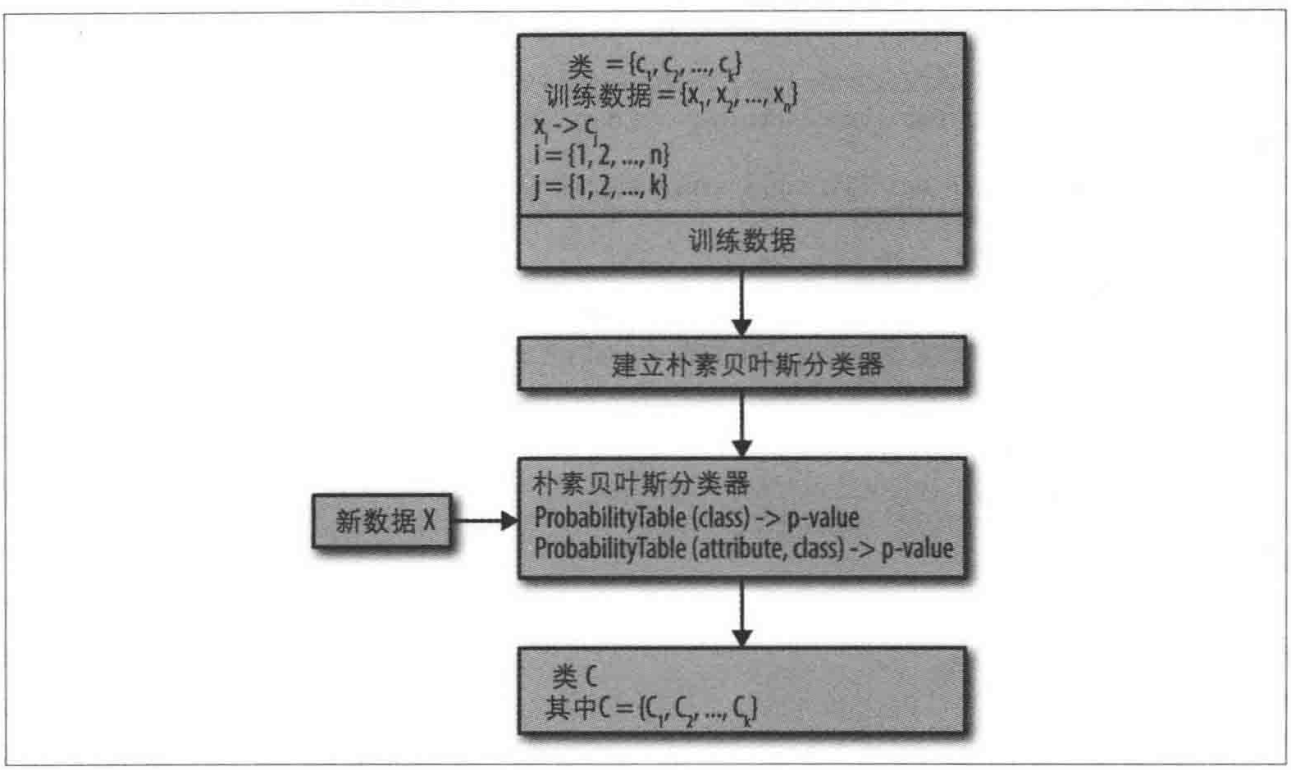



图14-3：朴素贝叶斯：训练阶段

## 阶段1：使用训练数据建立一个分类器

这个阶段由`BuildNaiveBayesClassifier`类实现，它接受训练数据（已经分类的数据），建立一个朴素贝叶斯分类器。这个朴素贝叶斯分类器是一组概率表（PT），见上一节的

讨论。首先，示例14-5给出了BuildNaiveBayesClassifier类，这里提供了高层步骤，接下来我们会详细解释每一个步骤。

示例14-5：建立分类器：高层步骤

```
// 步骤1：导入所需的类和接口
/**
 * 建立朴素贝叶斯分类器。目标是建立以下
 * 数据结构（概率和条件概率）
 * 将用来对新数据分类。
 * 令C = {C1, C2, ..., Ck}是分类集合，
 * 另外令各个训练数据元素有m个属性A = {A1, A2, ..., Am}。
 * 要建立
 *    ProbabilityTable(c) = p-value 其中c属于C
 *    ProbabilityTable(c, a) = p-value 其中c属于C ，a属于A
 * 这里1 >= p-value >=0
 *
 * 训练数据中的记录示例：
 *    <attribute_1><,><attribute_2><,>...<,><attribute_m><,><classification>
 *
 *
 * @author Mahmoud Parsian
 */
public class BuildNaiveBayesClassifier implements java.io.Serializable {

   static List<Tuple2<PairOfStrings, DoubleWritable>>
      toWritableList(Map<Tuple2<Strinwg,String>, Double> PT) {...}

   public static void main(String[] args) throws Exception {
      // 步骤2：处理输入参数
      // 步骤3：创建一个Spark上下文对象 (ctx)
      // 步骤4：读取训练数据
      // 步骤5：对训练数据的所有元素实现map()函数
      // 步骤6：对训练数据的所有元素实现reduce()函数
      // 步骤7：收集归约数据为Map
      // 步骤8：建立分类器
      // 步骤9：保存分类器
      //    9.1：用于对新记录分类的PT（概率表）
      //    9.2：分类列表（CLASSIFICATIONS）

      // 完成
      ctx.close();
      System.exit(0);
   }
}
```

## toWritableList()方法

toWritableList()方法的定义参见示例14-6，这个方法准备将分类器保存在HDFS中。为了能持久存储分类器，数据类型必须实现Hadoop的Writable接口。

示例14-6：toWritableList()方法

```
   static List<Tuple2<PairOfStrings, DoubleWritable>>
```

```
2       toWritableList(Map<Tuple2<String,String>, Double> PT) {
3       List<Tuple2<PairOfStrings, DoubleWritable>> list =
4          new ArrayList<Tuple2<PairOfStrings, DoubleWritable>>();
5       for (Map.Entry<Tuple2<String,String>, Double> entry : PT.entrySet()) {
6          list.add(new Tuple2<PairOfStrings, DoubleWritable>(
7             new PairOfStrings(entry.getKey()._1, entry.getKey()._2),
8             new DoubleWritable(entry.getValue())
9          ));
10      }
11      return list;
12   }
```

## 步骤1：导入所需的类和接口

这一步导入所需的Java类和接口 (参见示例14-7)。

示例14-7：步骤1：导入所需的类和接口

```
1 // 步骤1: 导入所需的类和接口
2 import java.util.Map;
3 import java.util.HashMap;
4 import java.util.List;
5 import java.util.ArrayList;
6 import scala.Tuple2;
7 import org.apache.spark.api.java.JavaRDD;
8 import org.apache.spark.api.java.JavaPairRDD;
9 import org.apache.spark.api.java.JavaSparkContext;
10 import org.apache.spark.api.java.function.PairFunction;
11 import org.apache.spark.api.java.function.PairFlatMapFunction;
12 import org.apache.spark.api.java.function.FlatMapFunction;
13 import org.apache.spark.api.java.function.Function;
14 import org.apache.spark.api.java.function.Function2;
15 import org.apache.spark.broadcast.Broadcast;
16 import edu.umd.cloud9.io.pair.PairOfStrings;
17 import org.apache.hadoop.mapred.SequenceFileOutputFormat;
18 import org.apache.hadoop.io.DoubleWritable;
```

## 步骤2：处理输入参数

这个步骤如示例14-8所示，它会读取一个输入参数<training-datafilename>，这是一个HDFS文件，表示用来建立朴素贝叶斯分类器的训练数据。

示例14-8：步骤2：处理输入参数

```
1       // 步骤2: 处理输入参数
2       if (args.length < 1) {
3          System.err.println("Usage: BuildNaiveBayesClassifier
4                              <training-data-filename>");
5          System.exit(1);
6       }
7       final String trainingDataFilename = args[0];
```

## 步骤3：创建一个Spark上下文对象

这个步骤如示例14-9所示，这里会创建一个SparkContextObject，这是一个工厂对象，用来创建新的RDD。SparkUtil类提供了一些static方法来创建JavaSparkContext实例，这里可以使用YARN的资源管理器，或者也可以使用Spark主节点URL。

示例14-9：步骤3：创建一个Spark上下文对象

```
1       // 步骤3：创建一个Spark上下文对象
2       JavaSparkContext ctx = SparkUtil.createJavaSparkContext("naive-bayes");
```

## 步骤4：读取训练数据

这个步骤如示例14-10所示，这里使用一个JavaSparkContext实例读取训练数据，创建一个JavaRDD<String>，其中各个元素分别是训练数据集中的一个记录，有以下格式：

```
<attribute_1><,><attribute_2><,>...<,><attribute_m><,><classification>
```

示例14-10：步骤4：读取训练数据

```
1       // 步骤4：读取训练数据
2       JavaRDD<String> training = ctx.textFile(trainingDataFilename, 1);
3       training.saveAsTextFile("/output/1");
4       // 得到训练数据大小，这个大小将
5       // 在计算条件概率时用到
6       long trainingDataSize = training.count();
```

为了进行调试，查看所创建的RDD：

```
# hadoop fs -cat /output/1/part*
Sunny,Hot,High,Weak,No
Sunny,Hot,High,Strong,No
Overcast,Hot,High,Weak,Yes
Rain,Mild,High,Weak,Yes
Rain,Cool,Normal,Weak,Yes
Rain,Cool,Normal,Strong,No
Overcast,Cool,Normal,Strong,Yes
Sunny,Mild,High,Weak,No
Sunny,Cool,Normal,Weak,Yes
Rain,Mild,Normal,Weak,Yes
Sunny,Mild,Normal,Strong,Yes
Overcast,Mild,High,Strong,Yes
Overcast,Hot,Normal,Weak,Yes
Rain,Mild,High,Strong,No
```

## 步骤5：对训练数据的所有元素实现map()函数

这个步骤如示例14-11所示，这里会映射训练数据中的所有元素，创建属性关于分类的一个计数。然后使用这些计数来计算条件概率。

示例14-11：步骤5：实现map()函数

```
// 步骤5：对训练数据的所有元素实现map()函数
// PairFlatMapFunction<T, K, V>
// T => Iterable<Tuple2<K, V>>
// K = <CLASS,classification> 或 <attribute,classification>
JavaPairRDD<Tuple2<String,String>,Integer> pairs =
    training.flatMapToPair(new PairFlatMapFunction<
    String,                     // A1,A2, ...,An,classification
    Tuple2<String,String>,      // K = Tuple2(CLASS,classification) or
                                // Tuple2(attribute,classification)
    Integer                     // V = 1
    >() {
    public Iterable<Tuple2<Tuple2<String,String>,Integer>> call(String rec) {
       List<Tuple2<Tuple2<String,String>,Integer>> result =
          new ArrayList<Tuple2<Tuple2<String,String>,Integer>>();
       String[] tokens = rec.split(",");
       // tokens[0] = A1
       // tokens[1] = A2
       // ...
       // tokens[m-1] = Am
       // token[m] = classification
       int classificationIndex = tokens.length -1;
       String theClassification = tokens[classificationIndex];
       for(int i=0; i < (classificationIndex-1); i++) {
          Tuple2<String,String> K = new Tuple2<String,String>("CLASS",
                                theClassification);
          result.add(new Tuple2<Tuple2<String,String>,Integer>(K, 1));
       }

       Tuple2<String,String> K = new Tuple2<String,String>("CLASS",
                           theClassification);
       result.add(new Tuple2<Tuple2<String,String>,Integer>(K, 1));
       return resultt;
    }
});
pairs.saveAsTextFile("/output/2");
```

为了进行调试，查看所创建的RDD：

```
# hadoop fs -cat /output/2/part*
((Sunny,No),1)
((Hot,No),1)
((High,No),1)
((CLASS,No),1)
((Sunny,No),1)
((Hot,No),1)
((High,No),1)
((CLASS,No),1)
((Overcast,Yes),1)
((Hot,Yes),1)
((High,Yes),1)
((CLASS,Yes),1)
((Rain,Yes),1)
((Mild,Yes),1)
```

```
((High,Yes),1)
((CLASS,Yes),1)
((Rain,Yes),1)
((Cool,Yes),1)
((Normal,Yes),1)
((CLASS,Yes),1)
((Rain,No),1)
((Cool,No),1)
((Normal,No),1)
((CLASS,No),1)
((Overcast,Yes),1)
((Cool,Yes),1)
((Normal,Yes),1)
((CLASS,Yes),1)
((Sunny,No),1)
((Mild,No),1)
((High,No),1)
((CLASS,No),1)
((Sunny,Yes),1)
((Cool,Yes),1)
((Normal,Yes),1)
((CLASS,Yes),1)
((Rain,Yes),1)
((Mild,Yes),1)
((Normal,Yes),1)
((CLASS,Yes),1)
((Sunny,Yes),1)
((Mild,Yes),1)
((Normal,Yes),1)
((CLASS,Yes),1)
((Overcast,Yes),1)
((Mild,Yes),1)
((High,Yes),1)
((CLASS,Yes),1)
((Overcast,Yes),1)
((Hot,Yes),1)
((Normal,Yes),1)
((CLASS,Yes),1)
((Rain,No),1)
((Mild,No),1)
((High,No),1)
((CLASS,No),1)
```

## 步骤6：实现reduce()函数

这个步骤如示例14-12所示，将归约上一步得到的计数，计算条件概率（由分类器使用）。

示例14-12：步骤6: 实现reduce()函数

```
// 步骤6: 对训练数据的所有元素实现reduce()函数
JavaPairRDD<Tuple2<String,String>, Integer> counts =
   pairs.reduceByKey(new Function2<Integer, Integer, Integer>() {
   public Integer call(Integer i1, Integer i2) {
```

```
5             return i1 + i2;
6           }
7       });
8 counts.saveAsTextFile("/output/3");
```

为了进行调试，查看所创建的RDD：

```
# hadoop fs -cat /output/3/part*
((Rain,Yes),3)
((Mild,No),2)
((Cool,No),1)
((Mild,Yes),4)
((Sunny,Yes),2)
((High,Yes),3)
((Hot,No),2)
((Sunny,No),3)
((Overcast,Yes),4)
((CLASS,No),5)
((High,No),4)
((Cool,Yes),3)
((Rain,No),2)
((Hot,Yes),2)
((CLASS,Yes),9)
((Normal,Yes),6)
((Normal,No),1)
```

## 步骤7：收集归约数据为Map

如示例14-13所示，这一步使用了Spark API的一个强大特性来收集JavaPairRDD<K,V>（作为一个Map<K,V>）。接下来，我们将使用所生成的Map<K,V>建立分类器。

示例14-13：步骤7: 收集归约数据为Map

```
1     // 步骤7: 收集归约数据为Map
2     // java.util.Map<K,V> collectAsMap()
3     // 向master返回这个RDD中的键-值对作为一个Map。
4     Map<Tuple2<String,String>, Integer> countsAsMap = counts.collectAsMap();
```

## 步骤8：建立分类器数据结构

这个步骤如示例14-14所示，这一步会建立分类器，这包括：

- 概率表（PT）。
- 分类列表（CLASSIFICATIONS）。

示例14-14：步骤8: 建立分类器数据结构

```
1     // 步骤8: 建立分类器数据结构，将用来
2     // 对新数据分类。我们需要建立以下结构：
3     //   1. 概率表(PT)
4     //   2. 分类列表（CLASSIFICATIONS）
5     Map<Tuple2<String,String>, Double> PT = new HashMap<Tuple2<String,String>,
```

```
6                                                          Double>();
7     List<String> CLASSIFICATIONS = new ArrayList<String>();
8     for (Map.Entry<Tuple2<String,String>, Integer> entry :
9                   countsAsMap.entrySet()) {
10       Tuple2<String,String> k = entry.getKey();
11       String classification = k._2;
12        if (k._1.equals("CLASS")) {
13           PT.put(k, ((double) entry.getValue()) /
14                  ((double) trainingDataSize));
12            CLASSIFICATIONS.add(k._2);
13        }
14        else {
15            Tuple2<String,String> k2 = new Tuple2<String,String>("CLASS",
16                                                              classification);
17            Integer count = countsAsMap.get(k2);
18            if (count == null) {
19               PT.put(k, 0.0);
20            }
21            else {
22               PT.put(k, ((double) entry.getValue()) /
23                      ((double) count.intValue()));
24            }
25         }
26    }
27    System.out.println("PT="+PT);
```

为了进行调试，查看所创建的PT：

```
PT={
    (Normal,No)=0.2,
    (Mild,Yes)=0.4444444444444444,
    (Normal,Yes)=0.6666666666666666,
    (Overcast,Yes)=0.4444444444444444,
    (CLASS,No)=0.35714285714285715,
    (CLASS,Yes)=0.6428571428571429,
    (Hot,Yes)=0.2222222222222222,
    (Hot,No)=0.4,
    (Cool,No)=0.2,
    (Sunny,No)=0.6,
    (High,No)=0.8,
    (Rain,No)=0.4,
    (Sunny,Yes)=0.2222222222222222,
    (Cool,Yes)=0.3333333333333333,
    (Rain,Yes)=0.3333333333333333,
    (Mild,No)=0.4,
    (High,Yes)=0.3333333333333333
}
```

## 步骤9：保存分类器数据结构

这个步骤如示例14-15所示，这里会保存这个分类器，包括：

- 概率表（PT）。

- 分类列表（CLASSIFICATIONS）。

要在Hadoop中保存数据，持久存储的类必须实现Hadoop的org.apache.hadoop.io.Writable[注1]接口。在这个代码中，我使用了PairOfStrings和DoubleWritable类，它们都实现了Hadoop的Writable接口。

示例14-15：步骤9: 保存分类器数据结构

```
// 步骤9: 保存以下数据结构，它们将用来对新数据分类：
//   1. 用于对新记录分类的PT（概率表）
//   2. 分类列表（CLASSIFICATIONS）

// 步骤9.1: 保存PT
// public <K,V> JavaPairRDD<K,V>
//    parallelizePairs(java.util.List<scala.Tuple2<K,V>> list)
// 分布存储一个本地Scala集合来构成一个RDD。
List<Tuple2<PairOfStrings, DoubleWritable>> list = toWritableList(PT);
JavaPairRDD<PairOfStrings, DoubleWritable> ptRDD = ctx.parallelizePairs(list);
ptRDD.saveAsHadoopFile("/naivebayes/pt",              // 路径名
                       PairOfStrings.class,           // 键类
                       DoubleWritable.class,          // 值类
                       SequenceFileOutputFormat.class // 输出格式类
                       );

 // 步骤9.2: 保存分类列表（CLASSIFICATIONS）
 // List<Text> writableClassifications = toWritableList(CLASSIFICATIONS);
 JavaRDD<String> classificationsRDD = ctx.parallelize(CLASSIFICATIONS);
 classificationsRDD.saveAsTextFile("/naivebayes/classes"); // 路径名
```

为了进行调试，查看所保存的分类器的内容：

```
# hadoop fs -text /classifier.seq/part*
(Normal, No)    0.2
(Mild, Yes)     0.4444444444444444
(Normal, Yes)   0.6666666666666666
(Overcast, Yes) 0.4444444444444444
(CLASS, No)     0.35714285714285715
(CLASS, Yes)    0.6428571428571429
(Hot, Yes)      0.2222222222222222
(Hot, No)       0.4
(Cool, No)      0.2
(Sunny, No)     0.6
(High, No)      0.8
(Rain, No)      0.4
(Sunny, Yes)    0.2222222222222222
(Cool, Yes)     0.3333333333333333
(Rain, Yes)     0.3333333333333333
(Mild, No)      0.4
(High, Yes)     0.3333333333333333
```

注1：这是一个可串行化对象，基于DataInput和DataOutput实现了一个简单、高效的串行化协议。

```
# hadoop fs -cat /naivebayes/classes/part*
Yes
No
```

### 建立NBC的YARN脚本

这里给出的shell脚本可以在YARN环境中运行我们的Spark程序建立一个NBC：

```
 1 # cat run_build_naive_bayes_classifier.sh
 2 #!/bin/bash
 3 #... 相应地设置CLASSPATH...
 4 BOOK_HOME=/mp/data-algorithms-book
 5 APP_JAR=$BOOK_HOME/dist/data_algorithms_book.jar
 6 THE_JARS=$BOOK_HOME/lib/cloud9-1.3.2.jar
 7 INPUT=/naivebayes/training_data.txt
 8 prog=org.dataalgorithms.chap14.spark.BuildNaiveBayesClassifier
 9 $SPARK_HOME/bin/spark-submit --class $prog \
10     --master yarn-cluster \
11     --num-executors 12 \
12     --driver-memory 3g \
13     --executor-memory 7g \
14     --executor-cores 12 \
15     --jars $THE_JARS \
16     $APP_JAR $INPUT
```

## 阶段2：使用分类器对新数据分类

在阶段1中，使用所提供的训练数据建立了一个朴素贝叶斯分类器。阶段2的目标是使用这个分类器对新数据分类。在阶段2中，我们从HDFS读取分类器，然后相应地对输入数据分类。将使用以下算法对新数据完成朴素贝叶斯分类：

$$C^{\text{predict}} = \arg\max_{c} P(C=c)\prod_{j=1}^{m} P(X_j = u_j \mid C=c)$$

下面使用Spark API通过一个驱动器类（NaiveBayesClassifier）实现这个分类。同样地，这里先给出高层步骤（参见示例14-16），然后再详细讨论每一个步骤。

示例14-16：Spark NBC解决方案的高层步骤

```
 1 // 步骤1：导入所需的类和接口
 2 /**
 3  * 朴素贝叶斯分类器，会对新数据进行分类（使用
 4  * 由BuildNaiveBayesClassifier类
 5  * 建立的分类器）。
 6  *
 7  * 对于一个给定的X = (X1, X2, ..., Xm)，我们要
 8  * 使用以下数据结构
 9  * （由BuildNaiveBayesClassifier类构建）对它进行分类：
10  *
11  *      ProbabilityTable(c) = p-value 这里c属于C
12  *      ProbabilityTable(c, a) = p-value 这里c属于C，a属于A
```

```
 *
 * 因此，给定X，可以将它分类为C，这里C属于{C1, C2, ..., Ck}
 *
 * @author Mahmoud Parsian
 */
public class NaiveBayesClassifier implements java.io.Serializable {
   public static void main(String[] args) throws Exception {
     // 步骤2: 处理输入参数
     // 步骤3: 创建一个Spark上下文对象 (ctx)
     // 步骤4: 读取要分类的新数据
     // 步骤5: 从Hadoop读取分类器
     // 步骤6: 缓存分类器组件，
     // 集群中的任何节点都可以使用这些组件
     // 步骤7: 对新数据分类

     // 完成
     ctx.close();
     System.exit(0);
   }
}
```

## 步骤1：导入所需的类和接口

第1步如示例14-17所示，要导入这个解决方案所需的类和接口。

示例14-17：步骤1：导入所需的类和接口

```
// 步骤1: 导入所需的类和接口
import java.util.Map;
import java.util.HashMap;
import java.util.List;
import java.util.ArrayList;
import scala.Tuple2;
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.api.java.JavaPairRDD;
import org.apache.spark.api.java.JavaSparkContext;
import org.apache.spark.api.java.function.PairFunction;
import org.apache.spark.api.java.function.PairFlatMapFunction;
import org.apache.spark.api.java.function.FlatMapFunction;
import org.apache.spark.api.java.function.Function;
import org.apache.spark.api.java.function.Function2;
import org.apache.spark.broadcast.Broadcast;
import edu.umd.cloud9.io.pair.PairOfStrings;
import org.apache.hadoop.mapred.SequenceFileInputFormat;
import org.apache.hadoop.io.DoubleWritable;
```

## 步骤2：处理输入参数

这一步如示例14-18所示，会处理这个解决方案的输入参数。

示例14-18：步骤2: 处理输入参数

```
      // 步骤2: 处理输入参数
      if (args.length != 2) {
         System.err.println("Usage: NaiveBayesClassifier +
```

```
4                           <input-data-filename> <NB-PT-path>");
5          System.exit(1);
6       }
7       final String inputDataFilename = args[0];       // 要分类的数据
8       final String nbProbabilityTablePath = args[1]; // 分类器的路径
```

## 步骤3：创建一个Spark上下文对象

通过使用SparkUtil类（如示例14-19所示），这里要创建JavaSparkContext的一个实例，它将用来创建新的RDD。

示例14-19：步骤3：创建一个Spark上下文对象

```
1 // 步骤3: 创建一个Spark上下文对象
2 JavaSparkContext ctx = SparkUtil.createJavaSparkContext("naive-bayes");
```

## 步骤4：读取要分类的新数据

要分类的原始数据（参见示例14-20）有以下记录格式：

```
<attribute_1><,><attribute_2><,>...<,><attribute_m>
```

示例14-20：步骤4：读取要分类的新数据

```
1 // 步骤4: 读取要分类的新数据
2 JavaRDD<String> newdata = ctx.textFile(inputDataFilename, 1);
```

## 步骤5：从Hadoop读取分类器

这个步骤如示例14-21所示，这里会读取（阶段1）BuildNaiveBayesClassifier类建立的分类器组件和数据结构。读入分类器组件后，下面就可以对新数据分类。

示例14-21：步骤5: 从Hadoop读取分类器

```
1    // 步骤5: 从Hadoop读取分类器
2    // JavaPairRDD<K,V> hadoopFile(String path,
3    //                             Class<F> inputFormatClass,
4    //                             Class<K> keyClass,
5    //                             Class<V> valueClass)
6    // 得到一个Hadoop文件的RDD，可能有任意的InputFormat。
7    // '''注意:''' 由于Hadoop的RecordReader类对各个记录重用
8    // 同一个Writable对象，直接缓存
9    // 所返回的RDD会为同一个对象创建多个引用。
10   // 如果计划直接缓存Hadoop Writable对象，应当
11   // 首先用一个map函数复制这些对象。
12   JavaPairRDD<PairOfStrings, DoubleWritable> ptRDD = ctx.hadoopFile(
13                   nbProbabilityTablePath,          // "/naivebayes/pt"
14                   SequenceFileInputFormat.class,   // 输入格式类
15                   PairOfStrings.class,             // 键类
16                   DoubleWritable.class             // 值类
17               );
18
19   // <K2,V2> JavaPairRDD<K2,V2> mapToPair(PairFunction<T,K2,V2> f)
```

```
20    // 通过对这个RDD的所有元素应用一个函数返回一个新的RDD。
21    JavaPairRDD<Tuple2<String,String>, Double> classifierRDD = ptRDD.mapToPair(
22        new PairFunction<
23                Tuple2<PairOfStrings,DoubleWritable>, // T
24                Tuple2<String,String>,                // K2,
25                Double                                // V2
26        >() {
27        public Tuple2<Tuple2<String,String>,Double>
28           call(Tuple2<PairOfStrings,DoubleWritable> rec) {
29           PairOfStrings pair = rec._1;
30           Tuple2<String,String> K2 =
31              new Tuple2<String,String>(pair.getLeftElement(),
32                                        pair.getRightElement());
33           Double V2 = new Double(rec._2.get());
34           return new Tuple2<Tuple2<String,String>,Double>(K2, V2);
35        }
36        });
```

## 步骤6：缓存分类器组件

这个步骤如示例14-22所示，这里会缓存分类器组件（使用Spark的Broadcast类），这样就可以从集群的任何节点访问和使用这些分类器组件。

示例14-22：步骤6: 缓存分类器组件

```
1    // 步骤6: 缓存分类器组件,
2    // 集群中的任何节点都可以使用这些组件。
3    Map<Tuple2<String,String>, Double> classifier = classifierRDD.collectAsMap();
4    final Broadcast<Map<Tuple2<String,String>, Double>> broadcastClassifier =
5      ctx.broadcast(classifier);
6
7    // 需要所有分类类别,
8    // 由BuildNaiveBayesClassifier类创建。
9    JavaRDD<String> classesRDD = ctx.textFile("/naivebayes/classes", 1);
10   List<String> CLASSES = classesRDD.collect();
11   final Broadcast<List<String>> broadcastClasses =
12     ctx.broadcast(CLASSEDS);
```

## 步骤7：对新数据分类

这个步骤如示例14-23所示，使用分类器组件对新数据进行分类。

示例14-23：步骤7: 对新数据分类

```
1 // 步骤7: 对新数据分类
2 // 现在，我们已经有一个朴素贝叶斯分类器和新数据。
3 // 使用这个分类器对新数据分类。
4 // PairFlatMapFunction<T, K, V>
5 // T => Iterable<Tuple2<K, V>>
6 // K = <CLASS,classification> 或 <attribute,classification>
7 JavaPairRDD<String,String> classified = newdata.mapToPair(new PairFunction<
8     String,                      // T = A1,A2,...,Am（要分类的数据）
9     String,                      // K = A1,A2,...,Am （要分类的数据）
10    String                       // V = T的分类
```

```
    >() {
    public Tuple2<String,String> call(String rec) {
        // 从Spark缓存得到分类器
        Map<Tuple2<String,String>, Double> CLASSIFIER =
            broadcastClassifier.value();
        // 从Spark缓存得到类
        List<String> CLASSES = broadcastClasses.value();

        // rec = (A1, A2, ..., Am)
        String[] attributes = rec.split(",");
        String selectedClass = null;
        double maxPosterior = 0.0;
        for (String aClass : CLASSES) {
            double posterior = CLASSIFIER.get(new Tuple2<String,String>("CLASS",
                                                                        aClass));
            for (int i=0; i < attributes.length; i++) {
                 Double probability = CLASSIFIER.get(new Tuple2<String,String>(
                                                    attributes[i], aClass));
                 if (probability == null) {
                     posterior = 0.0;
                     break;
                 }
                 else {
                     posterior *= probability.doubleValue();
                 }
            }
            if (selectedClass == null) {
                // 计算第1个分类的值
                selectedClass = aClass;
                maxPosterior = posterior;
            }
            else {
                if (posterior > maxPosterior) {
                    selectedClass = aClass;
                    maxPosterior = posterior;
                }
            }
        }

        return new Tuple2<String,String>(rec, selectedClass);
    }
});
classified.saveAsTextFile("/output/classified");
```

输入（也就是要分类的数据）如下所示：

```
# hadoop fs -cat /naivebayes/new_data_to_be_classified.txt
Rain,Hot,High,Strong
Overcast,Mild,Normal,Weak
Sunny,Mild,Normal,Week
```

下面是相应的输出（也就是分类后的数据）：

```
# hadoop fs -cat /output/classified/part*
```

```
(Rain,Hot,High,Strong,Yes)
(Overcast,Mild,Normal,Weak,Yes)
(Sunny,Mild,Normal,Week,Yes)
```

### Spark的YARN脚本

下面给出这个Spark NBC解决方案的YARN脚本：

```
# cat run_classify_new_data.sh
#... 建立所需的CLASSPATH
BOOK_HOME=/mp/data-algorithms-book
APP_JAR=$BOOK_HOME/dist/data_algorithms_book.jar
THE_JARS=$BOOK_HOME/lib/cloud9-1.3.2.jar
prog=org.dataalgorithms.chap14.spark.NaiveBayesClassifier
NEW_DATA=/naivebayes/new_data_to_be_classified.txt
CLASSIFIER_PT=/naivebayes/pt
CLASSIFIER_CLASSES=/naivebayes/classes
$SPARK_HOME/bin/spark-submit --class $prog \
    --master yarn-cluster \
    --num-executors 12 \
    --driver-memory 3g \
    --executor-memory 7g \
    --executor-cores 12 \
    --jars $THE_JARS \
    $MY_JAR $NEW_DATA $CLASSIFIER_PT $CLASSIFIER_CLASSES
```

# 使用Spark和Mahout

Apache Spark和Apache Mahout都提供了机器学习算法。如果不想开发你自己的机器学习算法，可以使用它们提供的算法。

## Apache Spark

MLlib是Apache Spark提供的可伸缩机器学习算法库。MLlib的根Java包是`org.apache.spark.mllib`。MLlib提供了以下功能：

- 朴素贝叶斯方法。
- 聚类和K-均值算法。
- 基本统计和总结统计。
- 关联度分析。
- 假设检验。
- 分类和回归。
- 线性模型（逻辑回归、线性回归等）。

MLlib为朴素贝叶斯分析提供了两个重要的类：

`org.apache.spark.mllib.classification.NaiveBayes`
: 给定（label, features）对的一个RDD，可以训练朴素贝叶斯模型。

`org.apache.spark.mllib.classification.NaiveBayesModel`
: 朴素贝叶斯分类器的模型。

示例14-24给出了这两个类的基本用法（这个代码取自*http://bit.ly/mllib_naive_bayes*）。

示例14-24：Spark的朴素贝叶斯类

```
JavaRDD<LabeledPoint> training = ... // 训练集
JavaRDD<LabeledPoint> test = ... // 测试集

double theSmoothingParameter = 1.0;
// 给定(label, features)对的一个RDD，训练一个朴素贝叶斯模型
final NaiveBayesModel model = NaiveBayes.train(training.rdd(),
                                               theSmoothingParameter);
JavaPairRDD<Double, Double> predictionAndLabel =
 test.mapToPair(new PairFunction<LabeledPoint, Double, Double>() {
 @Override
  public Tuple2<Double, Double> call(LabeledPoint p) {
      return new Tuple2<Double, Double>(model.predict(p.features()), p.label());
  }
 });

double accuracy = 1.0 * predictionAndLabel.filter(new Function<
      Tuple2<Double, Double>,
      Boolean>() {
   @Override
   public Boolean call(Tuple2<Double, Double> pl) {
     return pl._1() == pl._2();
   }
}).count() / test.count();
```

## Apache Mahout

Apache Mahout项目的目标是建立一个可伸缩的机器学习算法库。下面的两篇博客文章描述了如何使用Mahout对微博分类：

- “Using the Mahout Naive Bayes classifier to automatically classify Twitter messages”（使用Mahout朴素贝叶斯分类器自动对微博消息分类）。
- “Using the Mahout Naive Bayes Classifier to automatically classify Twitter messages(part 2: distribute classification with Hadoop)” [使用Mahout朴素贝叶斯分类器自动对微博消息分类（第2部分：用Hadoop发布分类）]。

# 第15章

# 情感分析

情感（Sentiment）是指“一种想法、意见、感情、情绪、观点或感觉”，Wikipedia（*http://en.wikipedia.org/wiki/Sentiment_analysis*）将情感分析（sentiment analysis）（也称为观点挖掘（opinion mining））描述为“使用自然语言处理、文本分析和计算语言学识别和提取原素材中的主观信息”。Bo Pang和Lillian Lee[21]指出“情感分析的目的是找出一段文本中表达的观点，将一个影评分类为‘赞成’（thumbs up）或‘反对’（thumbs down）就是情感分析的一个示例应用。”要完成关于某个事件的情感分析，我们要让计算机知道什么是情感（也就是说，如何定义“正面”或“负面”，以及如何区分“好”与“坏”）。这里就可以引入机器学习：我们必须教（teach）计算机学会正面、负面等的含义。这个过程中，第一步就是从一组训练数据建立一个模型。建立模型之后，再用它分析新数据。

那么什么是情感数据（sentiment data）？一般地，这是一种非结构化数据，表示原素材中的观点和情绪，如特殊的新闻报道、客户支持邮件、社交媒体帖子（如微博和Facebook评论）及在线商品评论。

要准确地完成情感分析，情感分析引擎必须完成某种层次的语音分析和语义消歧。因此，一个文本文档的情感分析不只是要完成词法分析，再根据一个“正面”和“负面”词汇列表检查词法分析得到的词汇。有时，我们需要了解词汇感情色彩的强烈程度（intensity），而且要考虑到另外一些因素，如否定和轻视。例如，考虑下面这个句子：

The movie was not good.

如果单看这个句子中的每个词，而不考虑它们之间的关系，你可能会认为它表达的情感是“中立的”［因为not（不）是负面的，而good（好）是正面的］。不过，如果看整个句子的语义，很显然，它表达了一种负面的情感。因此，情感分析的目的就是理解关于

某个主题所表达的观点，提取到不同的类别，如高兴、悲伤和愤怒，或者更简单地，可以是正面、负面和中立。

# 情感示例

下面来看情感分析应用的一些例子：

- 博客关于一些汽车品牌的评价，例如，部分丰田汽车因为刹车系统问题而被召回后，博客对丰田车的态度是怎样的？
- 观众在观看《钢铁侠》之前和之后分别有什么想法（我们可以分析他们在观看前和观看后发表的微博来回答这个问题）？人们是不是真的喜欢这部电影？
- 发布新款iPhone之前和之后顾客的态度如何？
- 总统辩论之前和之后选民有什么看法？有没有更倾向于民主党或共和党？
- 关于一项客户服务体验，顾客的感觉如何（是满意还是不满意）？在这里，要完成情感分析，我们需要有客户服务日志数据以及一个“满意”和“不满意”的词汇表。

# 情感分数：正面或负面

给定一个简短的句子（如一条微博，要求少于140个字符），如何确定它表达一种正面还是负面的情绪呢？如果要根据句法得到结论（忽略上下文语义），可以将这个句子通过词法分析分解为单词，然后再根据一个正面和负面词汇表检查这些单词[注1]。

例如，给定下面的句子（粗体单词是正面的，有下划线的单词是负面的）：

- 句子1：“The movie was **great** and I **loved** it.”
- 句子2：“The hamburger had a bad taste and was terrible, but I loved the fries.”

那么句子1和句子2的情感分数是怎样的呢？

```
sentimentScore(Sentence-1) = 1 正面 +
                             1 正面
                           = 2 正面 (正面情感)

sentimentScore(Sentence-2) = 1 负面   +
                             1 负面   +
                             1 正面
                           = 1 负面 (负面情感)
```

注1：Hu和Liu发表的文章“opinion lexicon”对6800个单词做了大致分类，将它们分为正面和负面词汇，这篇文章可以从*http://bit.ly/opinion_lexicon*下载。例如，正面词汇包括*love*、*best*、*cool*、*great*和*good*，负面词汇包括*terrible*、*bad*、*hate*、*worst*和*sucks*。

对于大多数情感分析，如果只是根据一个正面和负面词汇表检查单个单词，这种做法并不够，可能得不到我们期望的结果。在实际的情感分析中，最好不要简单地依赖于纯语法，而是要理解句子的语义，这包括单词的强烈程度，以及否定和轻视等其他因素。例如，考虑下面的句子：

"The movie was not great and I did not love it."

在现实世界的情感分析中，如果这里只是检查纯语法，可能得不到正确的结果。前面已经提到，如果只是查看单个单词，而不考虑它们之间的关系，纯语法分析算法可能会认为这里表达的情感是"中立"（因为*not*是负面的，而*great*和*love*是正面的）。不过查看整个句子的语义时，可以很清楚地看到这是一种负面情感。

所以，基本说来，可以认为情感分析任务是要确定文本中关于一个特定主题或趋势所表达的立场/观点是正面、负面，还是中立的。

# 一个简单的MapReduce情感分析示例

现在我们来看一个例子，这里要对给定的一组微博完成情感分析。假设我们感兴趣的关键字是*Obama*和*Romney*，进一步假设现在是选举年，你想知道人们关于这两个总统候选人的微博所反映出的态度。下面假设你想找出每天对各个候选人的情感趋势。映射器将接受一个微博，规范化文本，查找感兴趣的关键字（*Obama*或*Romney*），统计正面和负面关键字，然后用正面词汇比例减去负面词汇比例。要完成这个映射，这里需要两个词汇集：

- 一个正面词汇集（如*good*、*like*和*enjoy*）。
- 一个负面词汇集（如*hate*、*bad*和*terrible*）。

这两个集合由驱动器程序传入映射器。在Hadoop中，可以使用一个分布式缓存来实现（DistributedCache类[注2]）。

## 情感分析的map()函数

这里假设对于一个给定的微博，已经有一个规范化器和一个词法分析器函数，如示例15-1所示。

注2：DistributedCache（org.apache.hadoop.filecache.DistributedCache）是一个工具类，由Hadoop的MapReduce框架提供，用来缓存应用所需的文本和归档文件。

示例15-1：映射器函数

```
1 private static List<String> normalizeAndTokenize(String tweet) {
2    List<String> tokens = <normalize and tokenize all
3                           the words in the tweet text>;
4    return tokens;
5 }
```

通过使用normalizeAndTokenize()方法，可以完成map()方法，如示例15-2所示。

示例15-2：映射器函数，包括完整的map()方法

```
 1 private Set<String> positiveWords = null;
 2 private Set<String> negativeWords = null;
 3 // 默认候选人值
 4 private Set<String> allCandidates = {"obama", "romney"};
 5
 6 setup() {
 7    positiveWords = <load positive words from distributed cache>;
 8    negativeWords = <load negative words from distributed cache>;
 9    allCandidates = <load all candidates from distributed cache>;
10 }
11
12 /**
13  * @param key为微博日期： YYYY-MM-DD:hh:mm:ss
14  * @param value是一个微博
15  */
16 map(key, value) {
17     date = key; // 微博日期：YYYY-MM-DD
18     List<String> tweetWords = normalizeAndTokenize(value);
19     int positiveCount = 0;
20     int negativeCount = 0;
21     for (String candidate : allCandidates) {
22         if (candidate is in the tweetWords) {
23             int positiveCount = <count of positive words in tweetWords>;
24             int negativeCount = <count of negative words in tweetWords>;
25             double positiveRatio = positiveCount / tweetWords.size();
26             double negativeRatio = negativeCount / tweetWords.size();
27             outputKey = Pair(date, candidate);
28             outputValue = positiveRatio - negativeRatio;
29             emit (outputKey, outputValue);
30         }
31     }
32 }
```

## 情感分析的reduce()函数

示例15-3提供了基本情感分析的reduce()函数。

示例15-3：归约器函数

```
1 /**
2  * @param key为Pair(Date, String)
3  * 这里
4  * Date = YYYY-MM-DD
```

```
 5  * String = 一个候选人 ("obama"或"romney")
 6  * @param values是一个List<Double> , 这里Double表示一个比例
 7  */
 8 reduce(key, values) {
 9    double sumOfRatio = 0.0;
10    int n = 0; // 比例
11    for (Double ratio : values) {
12       n++;
13       sumOfRatio += ratio;
14    }
15    emit (key, sumOfRatio / n);
16 }
```

# 真实世界的情感分析

这一章中描述的算法非常简化（可能只适用于最简单的问题）。要实现一个现实的基于上下文的情感分析，这将是一个非常复杂的过程。要想充分理解数据，需要付出大量努力。如果想了解更多有关内容，Siddharth Batra和Deepak Rao[3]中描述了如何对微博完成现实的基于实体的情感分析，可以在MapReduce范式中实现。

第16章

# 查找、统计和列出大图中的所有三角形

社交网络分析人员往往使用图和矩阵来表示和分析社交节点（用户、好友和“好友的好友”）之间的关系模式。在网络分析中，数据通常建模为一个图或一个图集。图是一个数据结构，包括一个有限的节点集，这称为顶点（vertices），另外还包括一个有限的线集，称为边（edges），边会连接其中一些或全部节点。在使用三角形个数定义一些度量参数之前，首先来定义三联体（triad）和三角形（triangle）。令$T=(a,b,c)$是图（表示为G）中3个不同节点构成的一个集合，如果其中两个节点相连（$\{(a,b),(a,c)\}$），$T$就是一个三联体（triad），如果所有3个节点都相连（$\{(a,b),(a,c),(b,c)\}$），这就是一个三角形（triangle）。

在图论分析中，有3个很重要的度量参数：

- 全局集聚系数（Global clustering coefficient）。
- 传递比，定义为：

$$T(G)=\frac{3\times(\text{图中三角形个数})}{(\text{互连的顶点三联体个数})}$$

- 局部集聚系数（Local clustering coefficient）。

要为一个大图计算这3个度量参数，需要知道图中的三角形个数。关于这些度量参数的更多详细信息，可以参考[28]、[25]和[34]。

图可能包含数亿个节点（例如，一个社交网络中的用户）和边（这些用户之间的关

系），统计三角形个数是一个非常耗费时间的任务。这一章将提供两个MapReduce解决方案，可以用来查找、统计和列出一个给定图或图集中的所有三角形。这里分别给出Hadoop和Spark中的解决方案：

- MapReduce/Hadoop解决方案包括3个步骤，每一步是一个单独的MapReduce作业。
- Spark解决方案包括一个Java驱动器类，它要管理多个JavaRDD对象。在前面的章节中已经了解到，Spark的API比MapReduce/Hadoop API更高层，所以可以把所有`map()`和`reduce()`函数都放在一个Java驱动器类中。Spark提供了GraphX，这是Apache Spark为图和图并行计算提供的API，不过在这一章中，我们只使用了Spark API而没有用到GraphX库。

# 基本的图概念

令$V$是一个有限的节点集（再次说明，节点可以表示一个用户、一台计算机或者是一个有形/无形的物体），$E$是一个有限的边集（一个边表示两个节点之间的一个关系和互连性）。可以定义无向简单图为$G=(V, E)$。这里使用$n$表示节点个数，$m$表示边数。节点$v$的度$d(v) = |u \in V: \exists \{v, u\} \in E|$ 定义为$V$中与$v$邻接的节点个数。图$G=(V, E)$中的三角形$\Delta = (V_\Delta, E_\Delta)$是一个包含3个节点的子图，而且有$V_\Delta = \{u, v, w\} \subset V$和$E_\Delta = \{\{u, v\}, \{v, w\}, \{w, u\}\} \subset E$。我们使用符号$T(G)$表示图$G$中的三角形个数。

例如，图16-1所示的图中有下面这2个三角形：

- {{2, 3}, {3, 4}, {4, 2}}
- {{2, 4}, {4, 5}, {5, 2}}

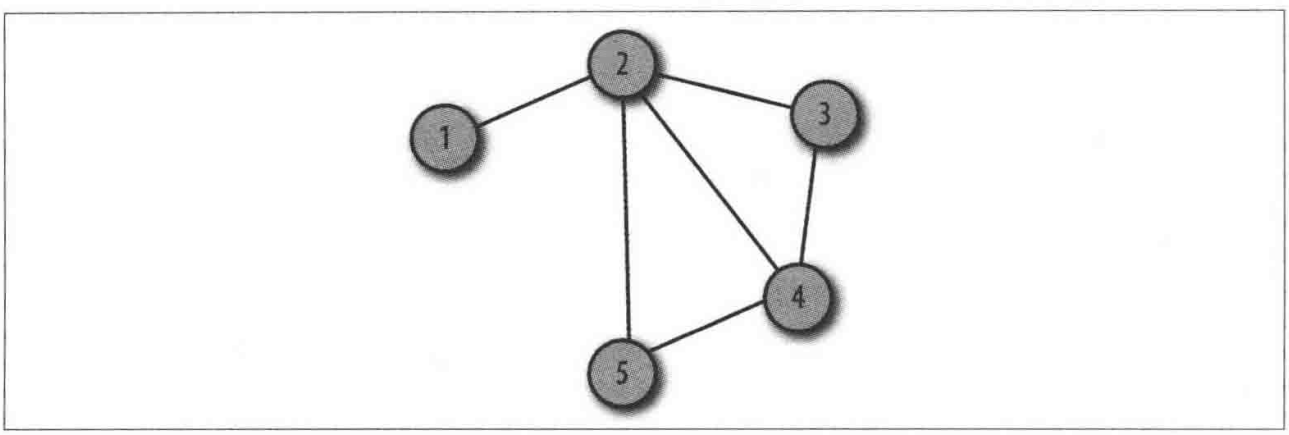



图16-1：图中的三角形

下面给出这些图度量参数的形式化描述。给定一个无向图$G=(V, E)$，可以定义图的一些度量参数。不过，首先我们需要一些基本定义：

- 节点$v$的度表示为$d(v)$，这是$V$中与$v$邻接的节点个数。
- 节点$v$的三角形数定义为：

$$T(v) = |(u, w) \in E : (v, u) \in E \text{而且} (v, w) \in E|$$

- 图$G$的三角形数定义为：

$$T(G) = \frac{1}{3}\sum_{v \in V} T(v)$$

- 节点$v$的三联体数定义为：

$$t(v) = \begin{pmatrix} d(v) \\ 2 \end{pmatrix}$$

- 图$G$的三联体数定义为：

$$t(G) = \sum_{v \in V} t(v)$$

使用这些基本定义，现在我们可以定义集聚系数：

- 节点$v$的集聚系数表示为$c(v)$，其中$d(v) \geqslant 2$：

$$c(v) = T(v)/t(v)$$

- 图$G$的集聚系数表示为$C(G)$,这是其节点集聚系数的平均数：

$$C(G) = \frac{1}{|W|}\sum_{w \in W} c(w)$$

这里$W$是$d(w) \geqslant 2$的节点$w$的集合。

# 三角形计数的重要性

识别和统计三角形可以帮助我们得出大图的两个重要度量参数（集聚系数和传递比），统计三角形数还可以帮助我们找出社交图中的模式，这对于入侵检测和识别社群和垃圾邮件很有用。类似地，生物网络中的三角形检测可以帮助我们找出蛋白质-蛋白质相互作用网络。

三角形和集聚系数在使用图数据结构的复杂网络分析中扮演着重要的角色。三角形的存在以及随之而来的高集聚系数可以揭示出社交网络、生物网络、Web和其他网络的重要特征。

由于统计一个图中的三角形个数非常耗费时间（有关的详细信息参见[24]），我们可以使用MapReduce来分布图数据和相应的计算。前面已经提到，这一章的目标是提供MapReduce解决方案来查找、列出和统计大图中的三角形。Schank和Wagner[24]提供了一个顺序（非MapReduce）算法来查找和统计大图中的所有三角形。

# MapReduce/Hadoop解决方案

我会分3步展示这个MapReduce解决方案（每一步都是一个单独的MapReduce作业）：

1. 生成经过$u$的长度为2的路径，并复制从u发出的所有边作为键（没有关联的数据）。我们将从这些路径生成可能的三角形。这一步可以通过以下`map()`和`reduce()`函数解决：

$$\begin{aligned}
&\text{mapper}_1\text{: (k,v)} \rightarrow \{(k,v),\\
&\qquad\qquad\qquad\qquad\quad (v,k)\}\\
&\text{reducer}_1\text{:}\{(k,V_1),(k,V_2),\ldots(k,V_n)\} \rightarrow \{[(V_1,V_2),(k)],\\
&\qquad\qquad\qquad\qquad\qquad\quad [(V_1,v_3),(k)],\\
&\qquad\qquad\qquad\qquad\qquad\quad \ldots,\\
&\qquad\qquad\qquad\qquad\qquad\quad [(V_{n-1},V_n),(k)],\\
&\qquad\qquad\qquad\qquad\qquad\quad [(k,V_1),(-)],\\
&\qquad\qquad\qquad\qquad\qquad\quad \ldots,\\
&\qquad\qquad\qquad\qquad\qquad\quad [(k,V_n),(-)]\}
\end{aligned}$$

2. 识别三角形，表示为$\{\{u, v\}, \{v, w\}, \{w, u\}\}$。这一步可能生成重复的三角形，可以通过以下`map()`和`reduce()`函数实现：

$$\begin{aligned}
&\text{mapper}_2\text{:[(u,v),(w)]} \rightarrow \text{[(u,v),(w)]}\\
&\text{reducer}_2\text{:}\{\text{[(u,v),(-)]},\text{[(u,v),}(w_1)\text{]}, \ldots, \text{[(u,v),}(w_n)\text{]} \rightarrow \{\text{[(u,v,}w_1\text{)]}\ ,\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \ldots,\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{[(u,v,}w_n\text{)]}\}\\
&\qquad\qquad\text{reducer}_2\text{:}\{\text{[(u,v),}(w_1)\text{]}, \ldots, \text{[(u,v),}(w_n)\text{]}\} \rightarrow \text{NoOutpwt}
\end{aligned}$$

3. 删除重复的三角形［$(a, b, c)$ 等同于 $(a, c, b)$］。这一步可以通过以下`map()`和`reduce()`函数解决：

$$\begin{aligned}
&\text{mapper}_3\text{: } (K_3, V_3) \rightarrow (\text{sort}(K_3), V_3)\\
&\text{reducer}_3\text{: } \{(K_3, V_{31}), (K_3, V_{32}), (K_3, V_{33}), \ldots\} \rightarrow \{(K_3, \text{null})
\end{aligned}$$

## 步骤1：MapReduce实现

第1步（也就是步骤1的$mapper_1$）的输入是一个给定无向图中的边。每个输入行采用（$u$, $v$）的格式，其中$u < v$（$\{u, v\}$是图中的一条边）。对于图16-1中的图，可以得到以下记录，如表16-1所示。

表16-1：步骤1中$mapper_1$的输入

| 键(startNode) | 值(endNode) |
|---|---|
| 1 | 2 |
| 2 | 3 |
| 2 | 4 |
| 2 | 5 |
| 3 | 4 |
| 4 | 5 |

步骤1的$mapper_1$会生成表16-2所示的输出（这将作为步骤1中$reducer_1$的输入）。需要说明，边必须是相互的。也就是说，每个{startNode, endNode}边必然有一个相应的{endNode, startNode}。

表16-2：步骤1中$mapper_1$的输出

| 键(startNode) | 值(endNode) |
|---|---|
| 1 | 2 |
| 2 | 1 |
| 2 | 3 |
| 3 | 2 |
| 2 | 4 |
| 4 | 2 |
| 2 | 5 |
| 5 | 2 |
| 3 | 4 |
| 4 | 3 |
| 4 | 5 |
| 5 | 4 |

洗牌和排序阶段之后，表16-3中的数据将作为步骤1中$reducer_1$的输入。

表16-3：步骤1中$reducer_1$的输入

| 键 | 值（节点列表） |
|---|---|
| 1 | [2] |
| 2 | [1, 3, 4, 5] |
| 3 | [2, 4] |
| 4 | [2, 3, 5] |
| 5 | [2, 4] |

步骤1中$reducer_1$定义如下：

$$
\begin{aligned}
&\text{mapper}_1\text{: (K, V)} \rightarrow \{(K, V),\ (V, K)\} \\
&\text{reducer}_1\text{: } \{(K, V_1), (K, V_2), \ldots, (K, V_n)\} \rightarrow \{[(V_1, V_2), (K)],\ [(V_1, V_3), (K)],\ \ldots,\ [(V_{n-1}, Vn), (K)],\ [(K,V_1), (-)],\ \ldots,\ [(K,V_n), (-)]\}
\end{aligned}
$$

步骤1中$reducer_1$的输入/输出如表16-4所示。

**表16-4：步骤1中$reducer_1$的输入/输出**

| 输入 | 输出 |
|---|---|
| [1],[2] | [(1, 2), (-)] |
| [2],[1, 3, 4, 5] | [(2, 1), (-)] |
| | [(2, 3), (-)] |
| | [(2, 4), (-)] |
| | [(2, 5), (-)] |
| | [(1, 3), (2)] |
| | [(1, 4), (2)] |
| | [(1, 5), (2)] |
| | [(3, 4), (2)] |
| | [(3, 5), (2)] |
| | [(4, 5), (2)] |
| [3],[2, 4] | [(3, 2), (-)] |
| | [(3, 4), (-)] |
| | [(2, 4), (3)] |
| [4],[2, 3, 5] | [(4, 2), (-)] |
| | [(4, 3), (-)] |
| | [(4, 5), (-)] |
| | [(2, 3), (4)] |
| | [(2, 5), (4)] |
| | [(3, 5), (4)] |
| [5],[2, 4] | [(5, 2), (-)] |
| | [(5, 4), (-)] |
| | [(2, 4), (5)] |

## 步骤2：识别三角形

步骤2的$mapper_2$是一个恒等映射器：它只是传递键和值，而不做任何进一步的处理。这使得我们可以为步骤2的$reducer_2$生成适当的键和值。步骤2的$reducer_2$有以下输入/输出，如表16-5所示。

**表16-5：步骤2中$reducer_2$的输入/输出**

| 输入 | 输出（三角形） |
|---|---|
| (1, 2) [-] | |
| (2, 1) [-] | |
| (2, 3) [-, 4] | (2, 3, 4) |
| (2, 4) [-, 3, 5] | (2, 4, 3), (2, 4, 5) |
| (2, 5) [-, 4] | (2, 4, 5) |
| (1, 3) [2] | |
| (1, 4) [2] | |
| (1, 5) [2] | |
| (3, 4) [2, -] | (3, 4, 2) |
| (3, 5) [2, 4] | |
| (4, 5) [2, -] | (4, 5, 2) |
| (3, 2) [-] | |
| (4, 2) [-] | |
| (4, 3) [-] | |
| (5, 4) [-] | |
| (5, 2) [-] | |

可以看到，每个三角形都列出了3次，这也强调了三角形的定义。例如，一个包括3个节点$u$、$v$和$w$的三角形可以等价地表述为：

```
{{u, v},{v, w},{w, u}}
{{v, w},{w, u},{u, v}}
{{w, u},{u, v},{v, w}}
```

## 步骤3：删除重复三角形

这一步要删除重复的三角形，从而得到唯一的三角形。这个步骤的输入形式为（a, b, c），其中a、 b和c分别表示三角形的一个顶点。

### 映射器

这个映射器接受形式为（a, b, c）的三角形，发出有序的顶点序列。

例如：

```
(1, 2, 3)将发出 (1, 2, 3)
(1, 3, 2)将发出 (1, 2, 3)
(2, 1, 3)将发出 (1, 2, 3)
(2, 3, 1)将发出 (1, 2, 3)
(3, 1, 2)将发出 (1, 2, 3)
(3, 2, 1)将发出 (1, 2, 3)
```

示例16-1定义了这个步骤的映射器。

示例16-1：步骤3: 映射器函数

```
1 // key未使用
2 // value = (a, b, c)
3 map(key, value) {
4   triangle = sort(value);
5   // triangle = (x, y, z)
6   // x < y < z
7   // x, y, z in {a, b, c}
8   emit(triangle, null)
9 }
```

### 归约器

由于映射器输出按映射器键分组，所以归约器只发出接收到的键（参见示例16-2）。这会生成唯一的三角形。我们根本不关心这个归约器的值（因为键就表示唯一的三角形）。

示例16-2：步骤3: 归约器函数

```
1 // key是一个三角形（a, b, c）
2 // 这里 a < b < c
3 // values = null列表
4 reduce(key, values) {
5   emit(key, null)
6 }
```

## Hadoop实现类

表16-6给出了这个解决方案的Hadoop实现类。

表16-6：Hadoop实现类

| 阶段 | 类名 | 类描述 |
|---|---|---|
| 作业提交 | TriangleCounterDriver | 向所有3个阶段提交作业的驱动器 |
| 1 | GraphEdgeMapper | 生成可能的三角形路径 |
| | GraphEdgeReducer | 识别可能的三角形 |
| 2 | TriadsMapper | 恒等映射器 |
| | TriadsReducer | 识别有重复的三角形 |
| 3 | UniqueTriadsMapper | 恒等映射器 |
| | UniqueTriadsReducer | 生成唯一的三角形 |

# 运行示例

下面各小节将提供这个MapReduce解决方案的脚本、HDFS输入、示例运行日志和期望的输出。

## 脚本

下面给出运行这个MapReduce/Hadoop解决方案的脚本：

```
$ cat run.sh
#/bin/bash
export JAVA_HOME=/usr/java/jdk7
export BOOK_HOME=/mp/data-algorithms-book
export APP_JAR=$BOOK_HOME/dist/data_algorithms_book.jar
#
export HADOOP_HOME=/usr/local/hadoop-2.3.0
export HADOOP_CONF_DIR=$HADOOP_HOME/etc/hadoop
export PATH=.:$JAVA_HOME/bin:$HADOOP_HOME/bin:$PATH
#
$HADOOP_HOME/bin/hadoop fs -rm /lib/data_algorithms_book.jar
$HADOOP_HOME/bin/hadoop fs -put $APP_JAR /lib/
export INPUT=/triangles/input
export OUTPUT=/triangles/output
$HADOOP_HOME/bin/hadoop fs -rmr $OUTPUT
$HADOOP_HOME/bin/hadoop fs -rmr /triangles/tmp1
$HADOOP_HOME/bin/hadoop fs -rmr /triangles/tmp2
$HADOOP_HOME/bin/hadoop jar $APP_JAR TriangleCounterDriver $INPUT $OUTPUT
```

## HDFS输入

```
# hadoop fs -ls /triangles/input/
-rw-r--r-- .. 24 2014-05-25 11:06 /triangles/input/sample_graph.txt
# hadoop fs -cat /triangles/input/sample_graph.txt
1 2
2 3
2 4
2 5
3 4
```

```
4 5
```

## 示例运行日志

日志输出如下所示；为适应版面，这里对输出做了编辑，并对格式有所调整：

```
# ./run.sh
...
14/05/25 13:40:17 INFO input.FileInputFormat: Total input paths to process : 1
...
14/05/25 13:40:17 INFO mapred.JobClient: Running job: job_201405251213_0025
14/05/25 13:40:18 INFO mapred.JobClient: map 0% reduce 0%
...
14/05/25 13:41:08 INFO mapred.JobClient: map 100% reduce 100%
14/05/25 13:41:09 INFO mapred.JobClient: Job complete: job_201405251213_0025
...
14/05/25 13:41:09 INFO mapred.JobClient: Map-Reduce Framework
14/05/25 13:41:09 INFO mapred.JobClient: Map input records=6
...
14/05/25 13:41:09 INFO mapred.JobClient: Reduce input records=12
14/05/25 13:41:09 INFO mapred.JobClient: Reduce output records=23
14/05/25 13:41:09 INFO mapred.JobClient: Map output records=12
14/05/25 13:41:10 INFO input.FileInputFormat: Total input paths to process : 10
14/05/25 13:41:10 INFO mapred.JobClient: Running job: job_201405251213_0026
14/05/25 13:41:11 INFO mapred.JobClient: map 0% reduce 0%
...
14/05/25 13:42:06 INFO mapred.JobClient: map 100% reduce 100%
14/05/25 13:42:06 INFO mapred.JobClient: Job complete: job_201405251213_0026
...
14/05/25 13:42:06 INFO mapred.JobClient: Map-Reduce Framework
14/05/25 13:42:06 INFO mapred.JobClient: Map input records=23
14/05/25 13:42:06 INFO mapred.JobClient: Reduce input records=23
14/05/25 13:42:06 INFO mapred.JobClient: Combine output records=0
14/05/25 13:42:06 INFO mapred.JobClient: Reduce output records=6
14/05/25 13:42:06 INFO mapred.JobClient: Map output records=23
14/05/25 13:42:06 INFO input.FileInputFormat: Total input paths to process : 10
14/05/25 13:42:06 INFO mapred.JobClient: Running job: job_201405251213_0027
14/05/25 13:42:07 INFO mapred.JobClient: map 0% reduce 0%
...
14/05/25 13:43:00 INFO mapred.JobClient: map 100% reduce 100%
14/05/25 13:43:01 INFO mapred.JobClient: Job complete: job_201405251213_0027
...
14/05/25 13:43:01 INFO mapred.JobClient: Map-Reduce Framework
14/05/25 13:43:01 INFO mapred.JobClient: Map input records=6
14/05/25 13:43:01 INFO mapred.JobClient: Reduce input records=6
14/05/25 13:43:01 INFO mapred.JobClient: Combine output records=0
14/05/25 13:43:01 INFO mapred.JobClient: Reduce output records=2
14/05/25 13:43:01 INFO mapred.JobClient: Map output records=6
```

## 期望的输出

```
# hadoop fs -cat /triangles/tmp1/part*
1,2 0
2,1 0
```

```
2,3 0
2,4 0
2,5 0
1,3 2
1,4 2
1,5 2
3,4 2
3,5 2
4,5 2
3,2 0
3,4 0
2,4 3
4,2 0
4,3 0
4,5 0
2,3 4
2,5 4
3,5 4
5,2 0
5,4 0
2,4 5

# hadoop fs -cat /triangles/tmp2/part*
4,5,2
2,3,4
2,4,3
2,4,5
2,5,4
3,4,2

# hadoop fs -cat /triangles/output/part*
2,3,4
2,4,5
```

# Spark解决方案

查找所有唯一三角形的Hadoop解决方案包括3个Map-Reduce作业（每个作业分别有自己的map()和reduce()函数）。由于Spark为映射器和归约器提供了一个高层API，所以这个Spark解决方案将通过一个Java驱动器类提供，其中包括10个基本步骤。首先，我会给出所有10个步骤，在提供运行示例之前会详细分析每一个步骤。

## 高层步骤

我们的Spark解决方案如示例16-3所示，这里完全依照为Hadoop实现提供的3阶段MapReduce算法来实现。

示例16-3：Spark的高层解决方案

```
1 // 步骤1: 导入所需的类和接口
2 public class CountTriangles {
3
```

```
4    public static void main(String[] args) throws Exception {
5       // 步骤2: 读取和验证输入参数
6       // 步骤3: 创建一个JavaSparkContext对象(ctx)
7       // 步骤4: 读取表示一个图的HDFS输入文本文件;
8       // 记录表示为JavaRDD<String>
9       // 步骤5: 为所有边创建一个新的JavaPairRDD，包括
10      // (source, destination)和(destination, source)
11      // 步骤6: 创建一个新的JavaPairRDD，将生成三联体
12      // 步骤7: 创建一个新的JavaPairRDD，表示所有可能的三联体
13      // 步骤8: 创建一个新的JavaPairRDD，将生成三角形
14      // 步骤9: 创建一个新的JavaPairRDD，表示所有三角形
15      // 步骤10: 消除重复的三角形，创建唯一的三角形
16
17      // 完成
18      ctx.close();
19      System.exit(0);
20   }
21 }
```

## 步骤1：导入所需的类和接口

如示例16-4所示，首先从Spark二进制版本提供的JAR文件导入必要的类和接口。大多数类和接口都在org.apache.spark.api.java包中定义。

示例16-4：步骤1: 导入所需的类和接口

```
1 // 步骤1: 导入所需的类和接口
2 import scala.Tuple2;
3 import scala.Tuple3;
4
5 import org.apache.spark.api.java.JavaRDD;
6 import org.apache.spark.api.java.JavaPairRDD;
7 import org.apache.spark.api.java.JavaSparkContext;
8 import org.apache.spark.api.java.function.PairFlatMapFunction;
9 import org.apache.spark.api.java.function.FlatMapFunction;
10
11 import java.util.Arrays;
12 import java.util.List;
13 import java.util.ArrayList;
14 import java.util.Collections;
```

## 步骤2：读取输入参数

这个步骤如示例16-5所示，将读取输入数据（作为一个HDFS文件），它将输入图表示为一个边集，边的形式为（source, destination）和（destination, source）。

示例16-5：步骤2: 读取和验证输入参数

```
1    // 步骤2: 读取和验证输入参数
2    if (args.length < 1) {
3      System.err.println("Usage: CountTriangles <hdfs-file>");
4      System.exit(1);
5    }
6    String hdfsFile = args[0];
```

## 步骤3：创建一个Spark上下文对象

在示例16-6中，我们创建了一个JavaSparkContext对象，它会返回JavaRDD并处理Java集合。创建一个JavaSparkContext对象后，就可以用RDD创建和管理数据。这里使用了Spark主节点名。例如，如果Spark集群由4个服务器组成{myserver100, myserver200, myserver300, myserver400}，Spark主节点表示为myserver100，那么Spark主节点URL就是*spark://myserver100:7077*（可以从命令行调用进行配置）。

示例16-6：步骤3: 创建一个Spark上下文对象

```
1    // 步骤3: 创建一个JavaSparkContext对象
2    // 这个对象将用来创建第一个RDD。
3    JavaSparkContext ctx = new JavaSparkContext();
```

## 步骤4：通过HDFS输入读取图

这一步如示例16-7所示，在这个步骤中，由JavaSparkContext对象读取我们的输入图（表示为hdfsFile），并创建第一个JavaRDD<String>。例如，这里使用HDFS文件*/triangles/sample_graph.txt*作为输入文件，表示我们的输入图。应该知道，Spark的主要抽象是一个分布式的项集合，称为弹性分布式数据集（resilient distributed data set，RDD）。可以从Hadoop InputFormat（如HDFS文件）创建RDD，或者通过转换其他RDD来创建。可以简单做个回顾，Spark RDD包括动作［如reduce()和collect()）和转换（如map(), flatMap(), flatMapToPair(), union()和filter()］，动作会返回值，转换会返回新RDD的指针。

示例16-7：步骤4: 读取表示一个图的HDFS输入文本文件

```
1      // 步骤4: 读取表示一个图的HDFS输入文本文件;
2      // 记录表示为JavaRDD<String>
3      // args[1] = HDFS文本文件: /triangles/sample_graph.txt
4 JavaRDD<String> lines = ctx.textFile(hdfsFile, 1);
```

## 步骤5：创建图的所有边

这一步如示例16-8所示，由边集中的各个边创建一个新的JavaPairRDD<Long,Long>，边表示为lines (这是一个JavaRDD<String>)。将各个边（nodeA nodeB）转换为两个节点对，如下所示：

```
(nodeA, nodeB)
(nodeB, nodeA)
```

注意，边必须是相互的，也就是说，每个（source, destination）边必然有一个相应的 （destination, source）。为了完成这个映射，这里使用PairFlatMapFunction<T, K, V>，其中T是输入，K和V是输出（创建一个(key=K, value=V)对）。我们使用Spark的JavaRDD.flatMapToPair()方法来生成所有必要的键-值对（表示为JavaPairRDD<Long,Long>）。

示例16-8：步骤5: 为所有边创建一个新的JavaPairRDD

```
// 步骤5：为所有边创建一个新的JavaPairRDD，
// 包括(source, destination)和(destination, source)
// PairFlatMapFunction<T, K, V>
// T => Iterable<Tuple2<K, V>>
JavaPairRDD<Long,Long> edges =
        lines.flatMapToPair(new PairFlatMapFunction<
                                                    String, // T
                                                    Long,   // K
                                                    Long    // V
                                                   >() {
    public Iterable<Tuple2<Long,Long>> call(String s) {
        String[] nodes = s.split(" ");
        long start = Long.parseLong(nodes[0]);
        long end = Long.parseLong(nodes[1]);
        // 需要说明，边必须是相互的；也就是说，
        // 每个(source, destination)边必然有一个相应的
        // (destination, source)。
        return Arrays.asList(new Tuple2<Long, Long>(start, end),
                             new Tuple2<Long, Long>(end, start));
    }
});
```

## 步骤6：创建RDD生成三联体

既然已经创建了所有边，下面来建立最终创建三角形的数据结构。对于每个节点，我们要找出可能构成三联体的所有相应终点。要完成这个任务，这里使用了`JavaPairRDD.groupByKey()`，如示例16-9所示。新的RDD表示为：

```
JavaPairRDD<Long, Iterable<Long>>
```

需要说明，值作为`Iterable<Long>`返回而不是作为`List<Long>`。这是Spark提供的另一个高层抽象，可以对用户隐藏具体实现（Spark平台可能实现优化；Spark可以由`ArrayList`、`LinkedList`或其他适当的数据结构实现`Iterable<Long>`）。

示例16-9：步骤6: 创建一个新的JavaPairRDD生成三联体

```
// 步骤6：创建一个新的JavaPairRDD生成三联体
JavaPairRDD<Long, Iterable<Long>> triads = edges.groupByKey();
```

我们收集并显示所有`triads`对象来调试步骤6：

```
// 步骤6.1: debug1
List<Tuple2<Long, Iterable<Long>>> debug1 = triads.collect();
for (Tuple2<Long, Iterable<Long>> t2 : debug1) {
   System.out.println("debug1 t2._1="+t2._1);
   System.out.println("debug1 t2._2="+t2._2);
}
```

### 步骤7：创建所有可能的三联体

这个步骤如示例16-10所示，将创建所有可能的三联体（后面将进一步检查这些三联体，查看是否构成一个三角形）。为了实现这一步，我们使用了JavaPairRDD.flatMapToPair()函数创建适当的键-值对，其中键是一个边（表示为Tuple2<node1,node2>），值是一个节点（表示为Long）。注意，所有RDD都是不可变的，这说明它们是只读的；不能以任何方式修改这些RDD或对它们排序。如果想要对RDD排序，首先需要克隆，然后再完成转换或动作。

示例16-10：步骤7: 创建一个新的JavaPairRDD，生成可能的三联体

```
// 步骤7: 创建一个新的JavaPairRDD，生成可能的三联体
JavaPairRDD<Tuple2<Long,Long>, Long> possibleTriads =
        triads.flatMapToPair(
        new PairFlatMapFunction<
                                Tuple2<Long, Iterable<Long>>, // 输入
                                Tuple2<Long,Long>,            // 键(输出)
                                Long                          // 值(输出)
                               >() {
    public Iterable<Tuple2<Tuple2<Long,Long>, Long>>
              call(Tuple2<Long, Iterable<Long>> s) {

       // s._1 = Long (作为键)
       // s._2 = Iterable<Long> (作为值)
       Iterable<Long> values = s._2;
       // 假设没有ID为0的节点
       List<Tuple2<Tuple2<Long,Long>, Long>> result =
       new ArrayList<Tuple2<Tuple2<Long,Long>, Long>>();

    // 生成可能的三联体。
    for (Long value : values) {
       Tuple2<Long,Long> k2 = new Tuple2<Long,Long>(s._1, value);
       Tuple2<Tuple2<Long,Long>, Long> k2V2 =
          new Tuple2<Tuple2<Long,Long>, Long>(k2, 0l);
       result.add(k2V2);
    }

    // RDD值是不可变的，所以必须复制这些值；
    // 将values复制到valuesCopy。
    List<Long> valuesCopy = new ArrayList<Long>();
    for (Long item : values) {
        valuesCopy.add(item);
    }
    Collections.sort(valuesCopy);

    // 生成可能的三联体。
    for (int i=0; i< valuesCopy.size() -1; ++i) {
       for (int j=i+1; j< valuesCopy.size(); ++j) {
          Tuple2<Long,Long> k2 =
             new Tuple2<Long,Long>(valuesCopy.get(i), valuesCopy.get(j));
          Tuple2<Tuple2<Long,Long>, Long> k2V2 =
             new Tuple2<Tuple2<Long,Long>, Long>(k2, s._1);
          result.add(k2V2);
```

```
43          }
44        }
45        return result;
46        }
47    });
```

我们收集并显示所有possibleTriads对象来调试步骤7：

```
// 步骤7.1: debug2
List<Tuple2<Tuple2<Long,Long>, Long>> debug2 =
  possibleTriads.collect();
for (Tuple2<Tuple2<Long,Long>, Long> t2 : debug2) {
   System.out.println("debug2 t2._1="+t2._1);
   System.out.println("debug2 t2._2="+t2._2);
}
```

## 步骤8：创建RDD生成三角形

这个步骤如示例16-11所示，将创建一个RDD，用来生成具体的三角形。对于每一个可能的三角形，得到的JavaPairRDD中将有相应的一个边（表示为Tuple2<Long,Long>）和一个节点集（表示为Iterable<Long>）。

示例16-11：步骤8: 创建一个新的JavaPairRDD，生成三角形

```
1     // 步骤8: 创建一个新的JavaPairRDD，生成三角形
2     JavaPairRDD<Tuple2<Long,Long>, Iterable<Long>> triadsGrouped =
3           possibleTriads.groupByKey();
```

我们使用JavaRDD.collect()方法来调试这一步：

```
// 步骤8.1: debug3
List<Tuple2<Tuple2<Long,Long>, Iterable<Long>>> debug3 =
  triadsGrouped.collect();
for (Tuple2<Tuple2<Long,Long>, Iterable<Long>> t2 : debug3) {
    System.out.println("debug3 t2._1="+t2._1);
    System.out.println("debug3 t2._2="+t2._2);
}
```

## 步骤9：创建所有三角形

这一步会生成所有可能的三角形，不过其中会包括重复的三角形。这里使用了一个FlatMapFunction，它的工作如下：

```
FlatMapFunction<T, R>
T => Iterable<R>
```

T是输入，Iterable<R>是输出。在这里：

```
T = Tuple2<Tuple2<Long,Long>, Iterable<Long>>
  = Tuple2 (Tuple2(nodeA,nodeB), <node1,node2, ...>)
```

```
R = Tuple3<Long,Long,Long>
  = Tuple3(nodeA,nodeB,nodeC>
  = 作为一个三角形，其中nodeC在{node1, node2, ...}中
```

这一步通过调用JavaPairRDD.flatMap()函数来完成，如示例16-12所示。

示例16-12：步骤9: 创建一个新的JavaPairRDD，生成所有三角形

```
// 步骤9: 创建一个新的JavaPairRDD，生成所有三角形
JavaRDD<Tuple3<Long,Long,Long>> trianglesWithDuplicates =
          triadsGrouped.flatMap(new FlatMapFunction<
               Tuple2<Tuple2<Long,Long>, Iterable<Long>>, // 输入
               Tuple3<Long,Long,Long>                     // 输出
                                                  >() {
     public Iterable<Tuple3<Long,Long,Long>>
            call(Tuple2<Tuple2<Long,Long>, Iterable<Long>> s) {

          // s._1 = Tuple2<Long,Long> (作为键) = "<nodeA><,><nodeB>"
          // s._2 = Iterable<Long> (作为值) = {0, n1, n2, n3, ...} 或
          //                                  {n1, n2, n3, ...}
          // 注意0是一个虚拟节点，并不存在这样一个节点。
          Tuple2<Long,Long> key = s._1;
          Iterable<Long> values = s._2;
          // 这里假设没有ID为0的节点。

          List<Long> list = new ArrayList<Long>();
          boolean haveSeenSpecialNodeZero = false;
          for (Long node : values) {
             if (node == 0) {
                haveSeenSpecialNodeZero = true;
             }
             else {
                list.add(node);
             }
          }

          List<Tuple3<Long,Long,Long>> result =
             new ArrayList<Tuple3<Long,Long,Long>>();
          if (haveSeenSpecialNodeZero) {
             if (list.isEmpty()) {
                // 没有找到三角形。
                return result;
             }
             // 发出三角形。
             for (long node : list) {
                long[] aTriangle = {key._1, key._2, node};
                Arrays.sort(aTriangle);
                Tuple3<Long,Long,Long> t3 =
                   new Tuple3<Long,Long,Long>(aTriangle[0], aTriangle[1],
                                              aTriangle[2]);
                result.add(t3);
             }
          }
          else {
             // 没有找到三角形。
```

```
48                return result;
49            }
50
51             return result;
52         }
53     });
```

我们使用JavaRDD.collect()方法来调试这一步：

```
// 步骤9.1: debug4: 打印所有三角形（包括重复的三角形）
System.out.println("=== Triangles with Duplicates ===");
List<Tuple3<Long,Long,Long>> debug4 = trianglesWithDuplicates.collect();
for (Tuple3<Long,Long,Long> t3 : debug4) {
  //System.out.println(t3._1 + "," + t3._2+ "," + t3._3);
  System.out.println("t3="+t3);
}
```

### 步骤10：创建唯一三角形

Spark提供了一个非常强大的API来查找给定RDD的唯一元素。JavaRDD<Tuple3<Long,Long,Long>>.distinct()将创建一个新的JavaRDD<Tuple3<Long,Long,Long>>，其中所有元素都是唯一的（参见示例16-13）。

示例16-13：步骤10: 消除重复三角形，创建唯一的三角形

```
1     // 步骤10: 消除重复三角形，创建唯一的三角形
2     JavaRDD<Tuple3<Long,Long,Long>> uniqueTriangles =
3           trianglesWithDuplicates.distinct();
```

我们使用JavaRDD.collect()方法来调试这一步：

```
// 步骤10.1: 打印唯一三角形
System.out.println("=== Unique Triangles ===");
List<Tuple3<Long,Long,Long>> output = uniqueTriangles.collect();
for (Tuple3<Long,Long,Long> t3 : output) {
   //System.out.println(t3._1 + "," + t3._2+ "," + t3._3);
   System.out.println("t3="+t3);
}
```

## 运行示例

这里提供了Spark统计三角形程序的完整运行示例。

### 输入

我们使用HDFS作为输入：

```
# hadoop fs -ls /triangles/
Found 1 items
-rw-r--r-- 3 hadoop root,hadoop 24 2014-05-25 17:45 /triangles/sample_graph.txt
```

```
# hadoop fs -cat /triangles/sample_graph.txt
1 2
2 3
2 4
2 5
3 4
4 5
```

## 脚本

这里使用了一个方便的shell脚本来运行我们的Spark程序：

```
# cat run_count_triangles.sh
#!/bin/bash
export JAVA_HOME=/usr/java/jdk7
export BOOK_HOME=/mp/data-algorithms-book
export APP_JAR=$BOOK_HOME/dist/data_algorithms_book.jar
export SPARK_HOME=/usr/local/spark-1.0.0
export SPARK_MASTER=spark://myserver100:7077
export SPARK_JAR=$DAB/lib/spark-assembly-1.0.0-hadoop2.3.0.jar
export INPUT=/triangles/sample_graph.txt
prog=org.dataalgorithms.chap16.spark.CountTriangles
$SPARK_HOME/bin/spark-submit
    --class $prog \
    --master $SPARK_MASTER \
    --num-executors 12 \
    --driver-memory 3g \
    --executor-memory 7g \
    --executor-cores 12 \
    $APP_JAR $INPUT
```

## 运行脚本

运行示例的输出（使用shell脚本*run_count_triangles.sh*）如下所示，这里对输出做了编辑，格式也有所调整，缩减为几个调试段：

```
Output log for debug1:
======================
debug1 t2._1=4
debug1 t2._2=[2, 3, 5]
debug1 t2._1=1
debug1 t2._2=[2]
debug1 t2._1=3
debug1 t2._2=[2, 4]
debug1 t2._1=5
debug1 t2._2=[2, 4]
debug1 t2._1=2
debug1 t2._2=[1, 3, 4, 5]

Output log for debug2:
======================
debug2 t2._1=(4,2)
debug2 t2._2=0
```

```
debug2 t2._1=(4,3)
debug2 t2._2=0
debug2 t2._1=(4,5)
debug2 t2._2=0
debug2 t2._1=(2,3)
debug2 t2._2=4
debug2 t2._1=(2,5)
debug2 t2._2=4
debug2 t2._1=(3,5)
debug2 t2._2=4
debug2 t2._1=(1,2)
debug2 t2._2=0
debug2 t2._1=(3,2)
debug2 t2._2=0
debug2 t2._1=(3,4)
debug2 t2._2=0
debug2 t2._1=(2,4)
debug2 t2._2=3
debug2 t2._1=(5,2)
debug2 t2._2=0
debug2 t2._1=(5,4)
debug2 t2._2=0
debug2 t2._1=(2,4)
debug2 t2._2=5
debug2 t2._1=(2,1)
debug2 t2._2=0
debug2 t2._1=(2,3)
debug2 t2._2=0
debug2 t2._1=(2,4)
debug2 t2._2=0
debug2 t2._1=(2,5)
debug2 t2._2=0
debug2 t2._1=(1,3)
debug2 t2._2=2
debug2 t2._1=(1,4)
debug2 t2._2=2
debug2 t2._1=(1,5)
debug2 t2._2=2
debug2 t2._1=(3,4)
debug2 t2._2=2
debug2 t2._1=(3,5)
debug2 t2._2=2
debug2 t2._1=(4,5)
debug2 t2._2=2

Output log for debug3:
======================
debug3 t2._1=(4,5)
debug3 t2._2=[0, 2]
debug3 t2._1=(1,4)
debug3 t2._2=[2]
debug3 t2._1=(4,2)
debug3 t2._2=[0]
debug3 t2._1=(3,5)
debug3 t2._2=[4, 2]
```

```
debug3 t2._1=(2,3)
debug3 t2._2=[4, 0]
debug3 t2._1=(5,4)
debug3 t2._2=[0]
debug3 t2._1=(2,4)
debug3 t2._2=[3, 5, 0]
debug3 t2._1=(1,2)
debug3 t2._2=[0]
debug3 t2._1=(3,2)
debug3 t2._2=[0]
debug3 t2._1=(2,5)
debug3 t2._2=[4, 0]
debug3 t2._1=(1,5)
debug3 t2._2=[2]
debug3 t2._1=(3,4)
debug3 t2._2=[0, 2]
debug3 t2._1=(4,3)
debug3 t2._2=[0]
debug3 t2._1=(2,1)
debug3 t2._2=[0]
debug3 t2._1=(1,3)
debug3 t2._2=[2]
debug3 t2._1=(5,2)
debug3 t2._2=[0]

=== Triangles with Duplicates ===
t3=(2,4,5)
t3=(2,3,4)
t3=(2,3,4)
t3=(2,4,5)
t3=(2,4,5)
t3=(2,3,4)

=== Unique Triangles ===
t3=(2,3,4)
t3=(2,4,5)
```

这一章提供了两个不同的MapReduce解决方案（Hadoop和Spark），用来识别、列出和统计一个给定图集中的三角形，这对于理解社交图有重要的意义。下一章将提供统计DNA序列K-mer的MapReduce解决方案。

# 第17章

# K-mer计数

K-mer是一个长度为$K$（$K$>0）的子串，统计所有这些子串的出现次数是很多DNA序列数据分析中的中心环节。一个DNA序列的K-mers计数是指得出整个序列中K-mer出现的频次。在生物信息科学中，K-mer计数用于基因组和转录组组装、宏基因组测序以及序列读取的错误修正。尽管原理上很简单，但K-mer计数是一个大数据难题，因为一个DNA样本可能包含数十亿DNA序列。Schatz实验室（*http://bit.ly/k-mer_counting*）给出了K-mer计数问题的定义。

这一章将使用MapReduce/Hadoop和Spark提供一个完整的K-mer计数解决方案。给定$K$>0，我们的实现能够发现所有K-mer，并且能找出对于给定$N$>0的top N个K-mer。完整的MapReduce/Hadoop和Spark解决方案可以从GitHub（*http://bit.ly/da_book*）获得。

K-mer是一个长度为$K$的DNA序列。例如，如果一个样本DNA序列为CACACACAGT，对于这个序列，它的3-mers和5-mers如下：

```
original sequence: CACACACAGT
3-mers:            CAC
                    ACA
                     CAC
                      ACA
                       CAC
                        ACA
                         CAG
                          AGT

original sequence: CACACACAGT
5-mers:            CACAC
                    ACACA
                     CACAC
                      ACACA
                       CACAG
                        ACAGT
```

给定一个由{*A*, *C*, *G*, *T*}中任意字符（表示主要碱基）构成的DNA序列（我们称为字符串S），想要统计S中每个长度为K的子串出现的次数。

# K-mer计数的输入数据

输入将使用FASTQ（*http://maq.sourceforge.net/fastq.shtml*），这是一种在文本文件中存储DNA序列的简洁而紧凑的格式。FASTQ文件中每一组连续的4行表示一个序列。例如，下面的4行表示一个序列：

```
@EAS54_6_R7_30800
GTTGCTTCCCGCGTGGGTGGGTCGGGG
+EAS54_6_R7
;;;;;;;33;;;;9;7;;.7;393333
```

要读取FASTQ文件格式，首先需要理解这个格式规范。第1行以@开头，第3行以+开头，第4行表示序列的质量。第2行表示要分析的序列。因此，可以删除第2行以外的其他各行。

## K-mer计数的示例数据

可以使用以下示例数据来测试你的K-mer计数解决方案：

- 大肠杆菌基因组（*http://bit.ly/e_coli_genome*）。
- 人类基因组（*http://bit.ly/hum_genome*）。注意*hg19*表示人类基因组版本19。

然后可以使用这个K-mer计数解决方案来回答以下问题：

- 大肠杆菌基因组中最常出现的前10个9-mers是什么？
- 人类基因组hg19中最常出现的前10个9-mers是什么？
- 人类基因组hg19中最常出现的前10个21-mers是什么？

# K-mer计数应用

DNA序列中的K-mer计数在很多生物信息应用中都是一个非常重要的步骤。下面给出K-mer计数的一些示例应用：

- 确定序列读取之间的偏差是测序错误还是序列的基因差异。
- 检测重复的序列，如转位因子，这是生物角色应用的一个重要因素。
- 修正短读组装错误。

- 计算亲缘度（relatedness）和特异度（unique enough）等度量参数（这在宏基因组应用中很有用）。

# K-mer计数MapReduce/Hadoop解决方案

从概念上讲，使用MapReduce的K-mer计数类似“字数统计”程序，不过，由于人类基因组中没有空格，所以我们要统计重叠的K-mers而不是分开的单词。

## map()函数

如果基因组序列为CACACACAGT而且$K$=3，要统计3-mers个数，`map()`函数（参见示例17-1）会输出以下键-值对：

```
CAC 1
ACA 1
CAC 1
ACA 1
CAC 1
ACA 1
CAG 1
AGT 1
```

示例17-1：K-mer计数: map()函数

```
/**
 * @param key由MapReduce生成，在这里忽略
 * @param sequence（作为值）是对应一个给定基因组序列的输入行
 */
map(Long key, String sequence) {
   // MapReduce/Hadoop: 从setup()得到K的值（K-mer）
   // Spark: 使用广播，调用一次
   for (int i=0; i < sequence.length()-K+1 ; i++) {
      String kmer = sequence.substring(i, K+i);
      emit(kmer, 1));
   }
}
```

## Reduce()函数

排序和洗牌阶段会对`map()`的输出排序，相同的键分组在一起，如下所示：

```
ACA 1
ACA 1
ACA 1
CAC 1
CAC 1
CAC 1
CAG 1
AGT 1
```

最后，reduce()函数（参见示例17-2）将输出以下结果：

```
ACA 3
CAC 3
CAG 1
AGT 1
```

示例17-2：K-mer计数: reduce()函数

```
/**
 * @param key是映射器生成的一个唯一K-mer
 * @param value是一个整数列表（K-mer的部分计数）
 */
reduce(String key, List<integer> value) {
   int sum = 0;
   for (int count : value) {
      sum += count;
   }
   emit(key, sum);
}
```

## Hadoop实现类

这里提供一个完整的K-mer计数解决方案，使用了4个很小的类，如表17-1所示。

表17-1：K-mer计数解决方案

| 类名 | 描述 |
| --- | --- |
| KmerCountDriver | MapReduce/Hadoop作业驱动器 |
| KmerCountMapper | 定义map()函数 |
| KmerCountReduce | 定义reduce()函数 |
| KmerUtil | 为给定的K生成K-mers |

# K-mer计数Spark解决方案

K-mer MapReduce工作流如图17-1所示。在介绍Spark解决方案之前，先来分析这个工作流中的步骤。

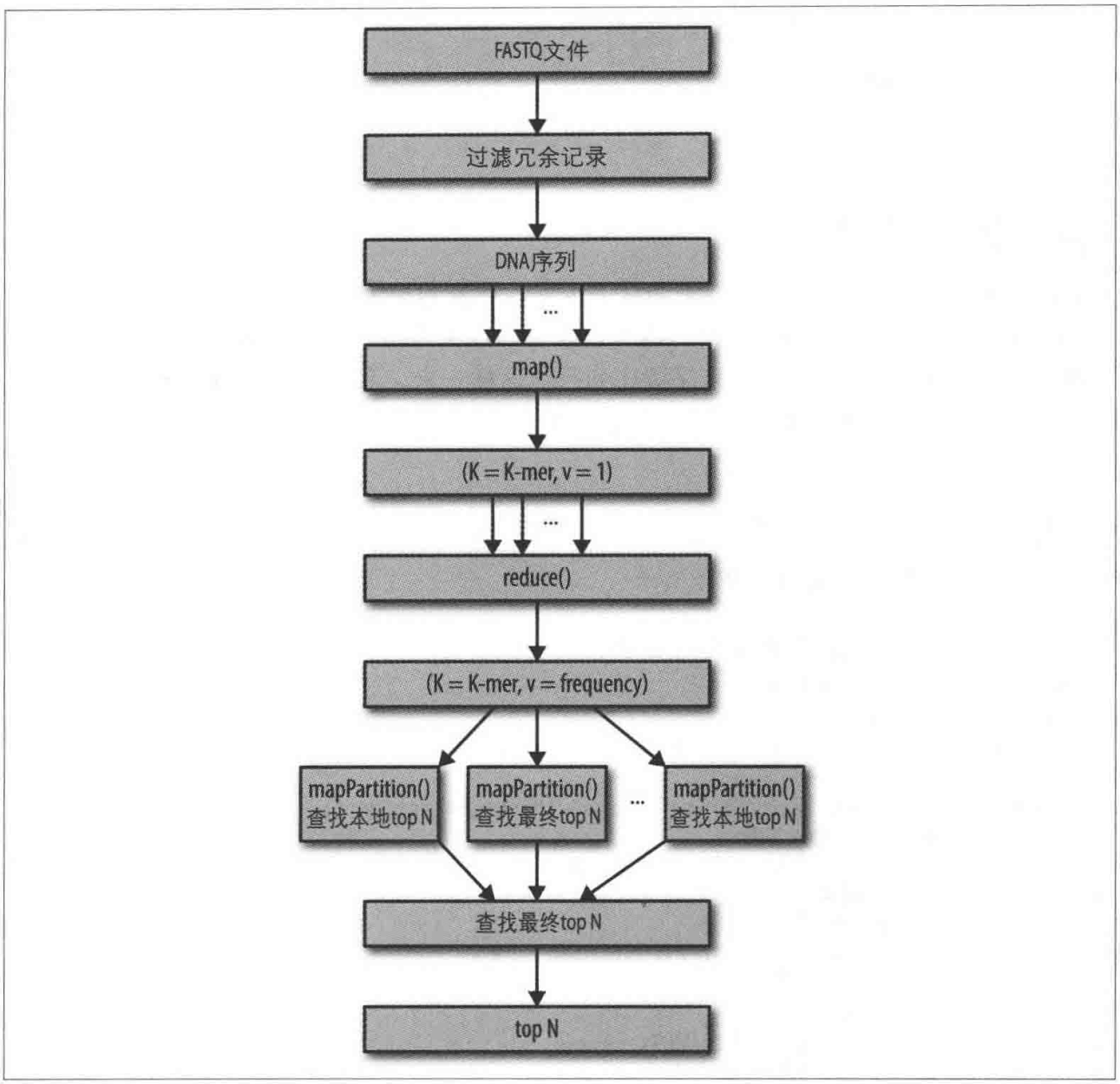



图17-1：K-mer MapReduce工作流

对于一组FASTQ文件，假设我们希望找出所有K-mers（给定$K$>0）和top $N$（给定$N > 0$）。由于FASTQ文件格式已经明确定义，首先要为给定的FASTQ文件创建一个`JavaRDD<String>`。接下来，删除非DNA序列的记录（类似之前的输入数据中的第1、3和4行），这个过滤工作由`JavaRDD.filter()`函数实现。得到真正的DNA序列后，接下来要创建（$K$, 1）对，其中K是一个K-mer。然后得出K-mer的频次。最后，可以找到top $N$的K-mer（$N > 0$）。查找top $N$很简单：假设（$K_2$,$V_2$）已经分区（$K_2$是一个K-mer，$V_2$是$K_2$的频次），将各个分区映射为top $N$。得到一个top $N$列表后（其中包括每个分区的top $N$），再完成最终的归约来找出最终的top $N$。

我们的Spark程序有3个输入：

- 存储在HDFS的FASTQ文件。
- $K > 0$：查找K-mers。
- $N > 0$：查找top $N$的K-mers。

## Spark解决方案

我们的Spark解决方案用一个Java驱动器类实现，这要归功于Spark提供的高层API。这个Spark解决方案读取FASTQ文件作为输入，将它们转换为JavaRDD<String>。接下来，要过滤冗余的记录，每4个记录中只保留序列行。这样一来，我们只会保留输入文件中真正的DNA序列。然后生成K-mers，并得出它们的频次。最后，以降序顺序应用Top $N$设计模式。

首先将给出整个解决方案，这里包含11个高层步骤（参见示例17-3），然后分别实现各个步骤（作为一个Java/Spark API）。

示例17-3：K-mer Spark解决方案高层步骤

```
// 步骤1：导入所需的类和接口
public class Kmer {
   static JavaSparkContext createJavaSparkContext() throws Exception {...}

   public static void main(String[] args) throws Exception {
      // 步骤2：处理输入参数
      // 步骤3：创建一个Spark上下文对象（ctx）
      // 步骤4：广播K和N作为全局共享对象，
      // 可以由所有集群节点访问
      // 步骤5：从HDFS读取FASTQ文件并创建第一个RDD
      // 步骤6：过滤冗余的记录
      // 步骤7：生成K-mers
      // 步骤8：组合/归约多次出现的K-mers
      // 步骤9：为所有分区创建一个本地top N
      // 步骤10：收集所有分区的本地top N
      // 并从所有分区找出最终的top N
      // 步骤11：按降序发出最终的top N

      // 完成
      ctx.close()
      System.exit(0);
   }
}
```

### 步骤1：导入所需的类和接口

步骤1如示例17-4所示，这里将导入所需的所有Java类和接口。

示例17-4：步骤1：导入所需的类和接口

```
// 步骤1：导入所需的类和接口
import java.util.Map;
```

```
 3 import java.util.SortedMap;
 4 import java.util.TreeMap;
 5 import java.util.List;
 6 import java.util.ArrayList;
 7 import java.util.Iterator;
 8 import java.util.Collections;
 9 import scala.Tuple2;
10 import scala.Tuple3;
11 import org.apache.spark.SparkConf;
12 import org.apache.spark.api.java.JavaRDD;
13 import org.apache.spark.api.java.JavaPairRDD;
14 import org.apache.spark.api.java.JavaSparkContext;
15 import org.apache.spark.api.java.function.PairFlatMapFunction;
16 import org.apache.spark.api.java.function.FlatMapFunction;
17 import org.apache.spark.api.java.function.Function;
18 import org.apache.spark.api.java.function.Function2;
19 import org.apache.spark.broadcast.Broadcast;
```

### createJavaSparkContext()方法

要创建RDD，需要JavaSparkContext的一个实例，这是一个工厂类，用来创建JavaRDD和JavaPairRDD对象（参见示例17-5）。

示例17-5：createJavaSparkContext()方法

```
1    static JavaSparkContext createJavaSparkContext() throws Exception {
2       JavaSparkContext ctx = new JavaSparkContext();
3       return ctx;
4    }
```

### 步骤2：处理输入参数

这个步骤如示例17-6所示，这里会从命令行读取3个必要的输入：FASTQ文件、K（K-mer的大小）和N（对应top $N$）。

示例17-6：步骤2: 处理输入参数

```
1    // 步骤2: 处理输入参数
2    if (args.length < 3) {
3      System.err.println("Usage: Kmer <fastq-file> <K> <N>");
4      System.exit(1);
5    }
6    final String fastqFileName = args[0];    // FASTQ文件作为输入
7    final int K = Integer.parseInt(args[1]); // 查找K-mers
8    final int N = Integer.parseInt(args[2]); // 查找top N
```

### 步骤3：创建一个Spark上下文对象

这一步使用Kmer.createJavaSparkContext()创建一个JavaSparkContext对象（参见示例17-7）。

示例17-7：步骤3: 创建一个Spark上下文对象

```
1    // 步骤3：创建一个Spark上下文对象
2    JavaSparkContext ctx = createJavaSparkContext();
```

## 步骤4：广播全局共享对象

在Spark框架中，要在所有集群节点中使用共享对象，可以广播这些对象，并从map()和reduce()函数读取。要保证一个对象可共享，可以使用以下API：

```
T t = <an-instance-of-T>;
Broadcast<T> broadcastT = ctx.broadcast(t);
```

要从任意的集群节点访问这个广播的（共享）对象，可以使用以下API：

```
T t = broadcastT.value();
```

由于K（K-mer大小）和N（对应top *N*）对所有集群节点都是必要的，所以我们要广播这两个值，如示例17-8所示。

示例17-8：步骤4: 广播K和N作为全局共享对象

```
1    // 步骤4：广播K和N作为全局共享对象，
2    // 可以由所有集群节点访问
3    final Broadcast<Integer> broadcastK = ctx.broadcast(K);
4    final Broadcast<Integer> broadcastN = ctx.broadcast(N);
```

## 步骤5：从HDFS读取FASTQ文件并创建第一个RDD

这个步骤如示例17-9所示，这里会读取FASTQ文件，并创建一个JavaRDD<String>对象，其中FASTQ文件的各个记录表示为一个String对象。

示例17-9：步骤5:从HDFS读取FASTQ文件并创建第一个RDD

```
1    // 步骤5：从HDFS读取FASTQ文件并创建第一个RDD
2    JavaRDD<String> records = ctx.textFile(fastqFileName, 1);
3    records.saveAsTextFile("/kmers/output/1");
```

下面是这一步的输出：

```
# hadoop fs -cat /kmers/output/1/part*
@EAS54_6_R1_2_1_413_324
CCCTTCTTGTCCCCAGCGTTTCTCC
+
;;3;;;;;;;;;;;;7;;;;;;;88
@EAS54_6_R1_2_1_540_792
TTGGCAGGCCAAGGCCGATGGATCA
+
;;;;;;;;;;;7;;;;;-;;;3;83
@EAS54_6_R1_2_1_443_348
GTTGCTTCTGGCGTGGGTGGGGGGG
+EAS54_6_R1_2_1_443_348
```

```
;;;;;;;;;;;9;7;;.7;393333
@EAS54_6_R1_2_1_413_324
CCCCCCTTGTCTTCAGCCCTTCTCC
+
;;3;;;;;;;;;;;;7;;;;;;;88
@EAS54_6_R1_2_1_540_792
TTTTCAGGCCAAGGCCGATGGATCA
+
;;;;;;;;;;;7;;;;;-;;;3;83
@EAS54_6_R1_2_1_443_348
GTTGTTTCTGGCGTGGGTGGGGGGG
+EAS54_6_R1_2_1_443_348
;;;;;;;;;;;9;7;;.7;393333
@EAS54_6_R1_2_1_443_348
GTTGTTTCTGGCGTGGGTGGCCCCC
+EAS54_6_R1_2_1_443_348
;;;;;;;;;;;9;7;;.7;393333
```

## 步骤6：过滤冗余的记录

这个步骤如示例17-10所示，这里使用了一个非常强大的Spark API：JavaRDD.filter()，在应用map()函数之前先过滤掉冗余的记录。在一个FASTQ文件中，我们只保留表示DNA序列的记录。

示例17-10：步骤6: 过滤冗余的记录

```
1    // 步骤6: 过滤冗余的记录
2    // JavaRDD<T> filter(Function<T,Boolean> f)
3    // 返回一个新的RDD，其中只包含满足一个谓词条件的元素。
4    JavaRDD<String> filteredRDD = records.filter(new Function<String,Boolean>() {
5      public Boolean call(String record) {
6         String firstChar = record.substring(0,1);
7         if ( firstChar.equals("@") ||
8              firstChar.equals("+") ||
9              firstChar.equals(";") ||
10             firstChar.equals("!") ||
11             firstChar.equals("~") ) {
12           return false; // 不返回这些记录
13        }
14        else {
15          return true;
16        }
17      }
18   });
19   filteredRDD.saveAsTextFile("/kmers/output/1.5");
```

下面是这一步的输出：

```
# hadoop fs -cat /kmers/output/1.5/part*
CCCTTCTTGTCCCCAGCGTTTCTCC
TTGGCAGGCCAAGGCCGATGGATCA
GTTGCTTCTGGCGTGGGTGGGGGGG
CCCCCCTTGTCTTCAGCCCTTCTCC
```

```
TTTTCAGGCCAAGGCCGATGGATCA
GTTGTTTCTGGCGTGGGTGGGGGGG
GTTGTTTCTGGCGTGGGTGGCCCCC
```

## 步骤7：生成K-mers

这个步骤如示例17-11所示，这里使用JavaRDD.flatMapToPair()函数为K-mers实现了一个映射器。这个映射器接受一个序列和K（K-mers的大小），然后生成所有（kmer, 1）对。

示例17-11：步骤7: 生成K-mers

```
1    // 步骤7: 生成K-mers
2    // PairFlatMapFunction<T, K, V>
3    // T => Iterable<Tuple2<K, V>>
4    JavaPairRDD<String,Integer> kmers =
5      filteredRDD.flatMapToPair(new PairFlatMapFunction<
6       String,       // T
7       String,       // K
8       Integer       // V
9     >() {
10     public Iterable<Tuple2<String,Integer>> call(String sequence) {
11        int K = broadcastK.value();
12        List<Tuple2<String,Integer>> list =
13           new ArrayList<Tuple2<String,Integer>>();
14        for (int i=0; i < sequence.length()-K+1 ; i++) {
15           String kmer = sequence.substring(i, K+i);
16           list.add(new Tuple2<String,Integer>(kmer, 1));
17        }
18           return list;
19        }
20    });
21    kmers.saveAsTextFile("/kmers/output/2");
```

下面是这一步的部分输出：

```
# hadoop fs -cat /kmers/output/2/part*
(CCC,1)
(CCT,1)
(CTT,1)
...
(GGC,1)
(GCC,1)
(CCC,1)
(CCC,1)
(CCC,1)
```

## 步骤8：组合/归约多次出现的K-mers

如示例17-12所示，这一步使用JavaPairRDD.reduceByKey()函数为K-mers实现了一个归约器。这个归约器接受一个键（K-mers）和值（K-mers的频次），然后生成K-mers的最终计数。

示例17-12：步骤8: 组合/归约多次出现的K-mers

```
// 步骤8: 组合/归约多次出现的K-mers
JavaPairRDD<String, Integer> kmersGrouped =
   kmers.reduceByKey(new Function2<Integer, Integer, Integer>() {
   public Integer call(Integer i1, Integer i2) {
      return i1 + i2;
   }
});
kmersGrouped.saveAsTextFile("/kmers/output/3");
```

下面是这一步的输出：

```
# hadoop fs -cat /kmers/output/3/part*
(CTC,2)
(AGC,2)
(CAA,2)
(TGC,1)
(GGC,9)
(GCC,6)
(GCT,1)
(CCT,3)
(TTT,5)
(TGG,12)
(TCC,3)
(CGA,2)
(CCC,11)
(AGG,4)
(GGT,3)
(GCA,1)
(CTG,3)
(TGT,4)
(TCA,4)
(GTG,6)
(CTT,6)
(TTC,8)
(CGT,4)
(GGG,13)
(CAG,4)
(GAT,4)
(TTG,6)
(CCA,3)
(AAG,2)
(TCT,7)
(GTT,6)
(GTC,2)
(GCG,4)
(ATG,2)
(CCG,2)
(GGA,2)
(ATC,2)
```

## 步骤9：为每个分区创建一个本地top N

这个步骤如示例17-13所示，将所有（kmer，frequency）对划分到多个分区，然后找出每

个分区的top *N*。对于每一个分区，我们只保留top *N*的（frequency, kmer）对（作为一个SortedMap），这里TreeMap实现了SortedMap接口。查找每个分区本地top *N*的工作由JavaPairRDD.mapPartitions()方法处理。

示例17-13：步骤9: 为所有分区创建一个本地top N

```
// 步骤9: 为所有分区创建一个本地top N
// 现在已经得到：(K=kmer,V=frequency)。
// 下一步是找出top N K-mers
JavaRDD<SortedMap<Integer, String>> partitions = kmersGrouped.mapPartitions(
 new FlatMapFunction<Iterator<Tuple2<String,Integer>>,
                     SortedMap<Integer, String>
                    >() {
 @Override
 public Iterable<SortedMap<Integer, String>>
     call(Iterator<Tuple2<String,Integer>> iter) {
     int N = broadcastN.value();
     SortedMap<Integer, String> topN = new TreeMap<Integer, String>();
     while (iter.hasNext()) {
       Tuple2<String,Integer> tuple = iter.next();
       String kmer = tuple._1;
       int frequency = tuple._2;
       topN.put(frequency, kmer);
       // 只保留top N
       if (topN.size() > N) {
          topN.remove(topN.firstKey());
       }
     }
     System.out.println("topN="+topN);
     return Collections.singletonList(topN);
   }
 });
```

## 步骤10：找出最终top N

这个步骤如示例17-14所示，这里会聚集从各个分区生成的所有top *N*，并为所有分区创建最终top *N*。在这一步中，我们需要得到N的值，这个值从一个广播变量（broadcastN）读取。

示例17-14：步骤10: 找出最终top N

```
// 步骤10: 收集所有分区的本地top N
// 并从所有分区找出最终的top N
SortedMap<Integer, String> finaltopN = new TreeMap<Integer, String>();
List<SortedMap<Integer, String>> alltopN = partitions.collect();
for (SortedMap<Integer, String> localtopN : alltopN) {
    // frequency = tuple._1
    // kmer = tuple._2
    for (Map.Entry<Integer, String> entry : localtopN.entrySet()) {
        finaltopN.put(entry.getKey(), entry.getValue());
        // 只保留top N
        if (finaltopN.size() > N) {
           finaltopN.remove(finaltopN.firstKey());
```

```
13          }
14       }
15    }
```

## 步骤11：发出最终top *N*

这个步骤如示例17-15所示，将发出所有K-mers的最终top *N*。

示例17-15：步骤11: 发出最终top N

```
1    // 步骤11: 按降序发出最终的top N
2    System.out.println("=== top " + N + " ===");
3    List<Integer> frequencies = new ArrayList<Integer>(finaltopN.keySet());
4    for(int i = frequencies.size()-1; i>=0; i--) {
5       System.out.println(frequencies.get(i) + "\t" +
6                          finaltopN.get(frequencies.get(i)));
7    }
```

## 合并步骤9和10为一个函数

步骤9和10可以合并到一个Spark函数JavaPairRDD.top()。top()函数定义如下：

```
import java.util.List;
import java.util.Comparator;
...
List<T> top(int N, Comparator<T> comp)
// 根据指定Comparator[T]的定义
// 从这个RDD返回top N个元素
// 参数:
// N-要返回前几个元素
// comp-定义排序顺序的比较器
```

top()的第2个参数（Comparator<T>）定义了RDD元素如何排序（可以按升序或降序对元素排序）。接下来，我们要定义一个Comparator。输入RDD为kmersGrouped，这是一个JavaPairRDD<String,Integer>。因此，这个比较器必须根据K-mers（作为String）的频次（Integer）对元素排序。

示例17-16展示了如何按降序对元组排序。

示例17-16：降序比较器：TupleComparatorDescending

```
1 static class TupleComparatorDescending
2    implements Comparator<Tuple2<String, Integer>>, Serializable {
3    final static TupleComparatorDescending INSTANCE =
4      new TupleComparatorDescending();
5    public int compare(Tuple2<String, Integer> t1,
6                       Tuple2<String, Integer> t2) {
7       // 按降序对RDD元素排序（用于top N)
8       return -t1._2.compareTo(t2._2);
9    }
10 }
```

示例17-17展示了如何按升序对元组排序。

示例17-17：升序比较器：TupleComparatorAscending

```
static class TupleComparatorAscending
   implements Comparator<Tuple2<String, Integer>>, Serializable {
   final static TupleComparatorAscending INSTANCE =
     new TupleComparatorAscending();
   public int compare(Tuple2<String, Integer> t1,
                      Tuple2<String, Integer> t2) {
      //按升序对RDD元素排序（用于bottom N)
      return -t1._2.compareTo(t2._2);
   }
}
```

现在可以应用这个top()函数（注意你的比较器必须实现Serializable接口，因为排序算法将在一个分布式环境中使用）：

```
// 查找top 10
List<Tuple2<String, Integer>> finalTop10 =
   kmersGrouped.top(10, TupleComparatorDescending.INSTANCE);
// 查找bottom 10
List<Tuple2<String, Integer>> finalBottom10 =
   kmersGrouped.top(10, TupleComparatorAscending.INSTANCE);
```

# 运行示例

## Spark的YARN脚本

下面给出的脚本可以在YARN环境中运行我们的Spark程序：

```
$ cat run_kmer.sh
#!/bin/bash
export JAVA_HOME=/usr/java/jdk7
export SPARK_HOME=/usr/local/spark-1.1.0
export HADOOP_HOME=/usr/local/hadoop-2.5.0
export HADOOP_CONF_DIR=$HADOOP_HOME/etc/hadoop
export YARN_CONF_DIR=$HADOOP_HOME/etc/hadoop
export BOOK_HOME=/mp/data-algorithms-book
export APP_JAR=$BOOK_HOME/dist/data_algorithms_book.jar
export SPARK_JAR=$BOOK_HOME/lib/spark-assembly-1.1.0-hadoop2.5.0.jar
#
export INPUT=/data/sample.fastq
export K=3
export N=5
prog=org.dataalgorithms.chap17.spark.Kmer
$SPARK_HOME/bin/spark-submit --class $prog \
    --master yarn-cluster \
    --num-executors 12 \
    --driver-memory 3g \
    --executor-memory 7g \
    --executor-cores 12 \
    $APP_JAR $INPUT $K $N
```

## HDFS输入

接下来，检查HDFS中的输入数据：

```
# hadoop fs -cat /data/sample.fastq
@EAS54_6_R1_2_1_413_324
CCCTTCTTGTCCCCAGCGTTTCTCC
+
;;3;;;;;;;;;;;;7;;;;;;;88
@EAS54_6_R1_2_1_540_792
TTGGCAGGCCAAGGCCGATGGATCA
+
;;;;;;;;;;;7;;;;;-;;;3;83
@EAS54_6_R1_2_1_443_348
GTTGCTTCTGGCGTGGGTGGGGGGG
+EAS54_6_R1_2_1_443_348
;;;;;;;;;;;9;7;;.7;393333
@EAS54_6_R1_2_1_413_324
CCCCCCTTGTCTTCAGCCCTTCTCC
+
;;3;;;;;;;;;;;;7;;;;;;;88
@EAS54_6_R1_2_1_540_792
TTTTCAGGCCAAGGCCGATGGATCA
+
;;;;;;;;;;;7;;;;;-;;;3;83
@EAS54_6_R1_2_1_443_348
GTTGTTTCTGGCGTGGGTGGGGGGG
+EAS54_6_R1_2_1_443_348
;;;;;;;;;;;9;7;;.7;393333
@EAS54_6_R1_2_1_443_348
GTTGTTTCTGGCGTGGGTGGCCCCC
+EAS54_6_R1_2_1_443_348
;;;;;;;;;;;9;7;;.7;393333
```

## 最终top *N*输出

下面给出最终的top 5列表（降序）：

```
=== top 5 ===
13 GGG
12 TGG
11 CCC
9 GGC
8 TTC
```

这一章提供了针对一组给定的DNA序列完成Kmers计数的MapReduce/Hadoop和Spark解决方案。可以应用第3章介绍的Top N设计模式来找出前几个K-mers。下一章会介绍基因组中的一个非常重要的概念：DNA测序，也就是对于给定的一个很大的DNA序列集合，找出单核苷酸多态性（single nucleotide polymorphisms，SNP）。

# 第18章

# DNA测序

如今，基因组测序仪（如Illumina的HiSeq 4000）可以在几小时内生成数千兆亿碱基对（Gb）的DNA和RNA测序数据，而成本不超过1000美元（几年前，成本则在100000美元以上，第一个人类基因组测序的成本则高达30亿美元）。生物和人类领域获得的成功很大程度上取决于是否能正确地分析这些技术生成的大数据集，而这又进一步要求我们采用信息科学领域取得的新进展。利用MapReduce/Hadoop和Spark，我们能够在数小时内完成成千上万GB/PB数据的计算和分析（而不再是几天甚至几星期）。例如，最近就曾使用Spark在23分钟内用206台机器完成了100 TB数据的排序[注1]。

简单地讲，DNA测序是对整个基因组（如人类基因组）的测序。根据*http://dnasequencing.com*的描述："如果找到DNA是发现包含基因组成信息的具体物质。那么DNA测序则是发现能读取这个信息的过程。"DNA测序的主要功能是找出一个DNA分子中核甘酸的具体排列顺序。另外，DNA测序还用来确定一个DNA片段中4种碱基的排序顺序，包括腺嘌呤（A）、鸟嘌呤（G）、胞嘧啶（C）与胸腺嘧啶（T）。

DNA测序存在哪些难题？难题确实有很多，不过这里只讨论其中最重要的一些问题：

- 有很多测序技术可以生成FASTQ文件，对于每种测序技术，DNA序列的长度会有所不同。
- 输入数据（FASTQ数据）很庞大（一个DNA序列样本的大小可能高达900GB）。
- 即便是使用一个计算能力很强的服务器，处理一个DNA序列并抽取单核苷酸多态性（single nucleotide polymorphisms，SNP）等变异也需要花费很长时间（多达80小时）。

注1：*http://bit.ly/spark_2014_gray_sort*

- DNA测序涉及很多算法和步骤，所以选择适当的开源工具组合是一个很大的挑战。例如，有非常多的映射/比对算法和参数。
- 很难实现可伸缩性（即优化映射器和归约器数）。

图18-1给出了一个高层DNA测序工作流。这一章的重点是利用一组MapReduce程序实现DNA测序，这些程序接受一个DNA数据集（FASTQ文件），最终生成一个VCF（variant call format，变体调用格式）文件，其中包含给定DNA数据集对应的变异。

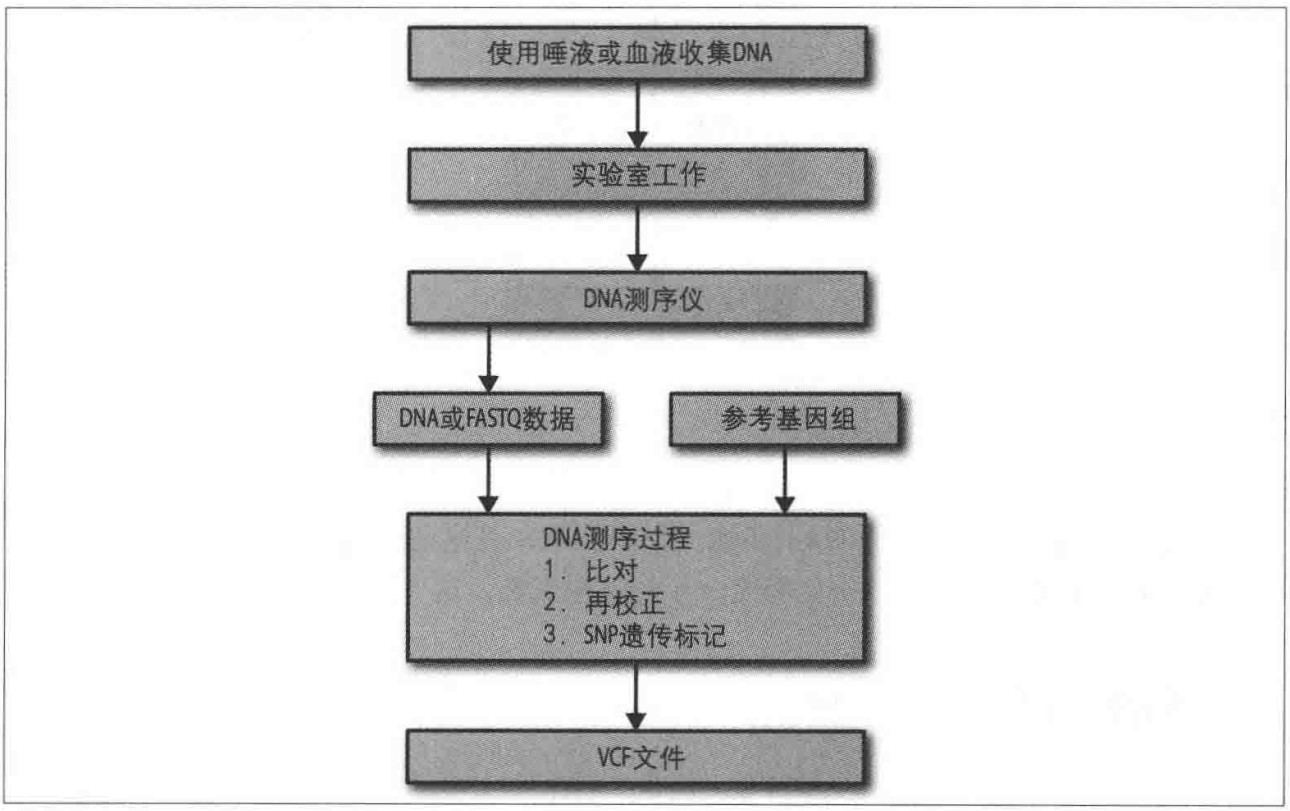



图18-1：DNA测序的高层视图

DNA测序的主要目标之一是找出变异，因为我们的大部分DNA都是相同的：不同的人只有很少一部分DNA有区别。识别单核苷酸多态性（SNP）就是一个重要的例子。从原始基因序列识别和抽取SNP涉及很多算法，还要应用一组不同的工具。

DNA测序管道如图18-2所示，其中包括以下重要步骤：

1. 输入数据验证：对输入数据（如FASTQ文件）完成质量控制。
2. 比对：将短序列（short reads）映射到参考基因组。
3. 再校正：对比对完成可视化显示和后处理，包括碱基质量再校正。
4. 变异检测：执行SNP调用过程，同时过滤SNP候选标记。

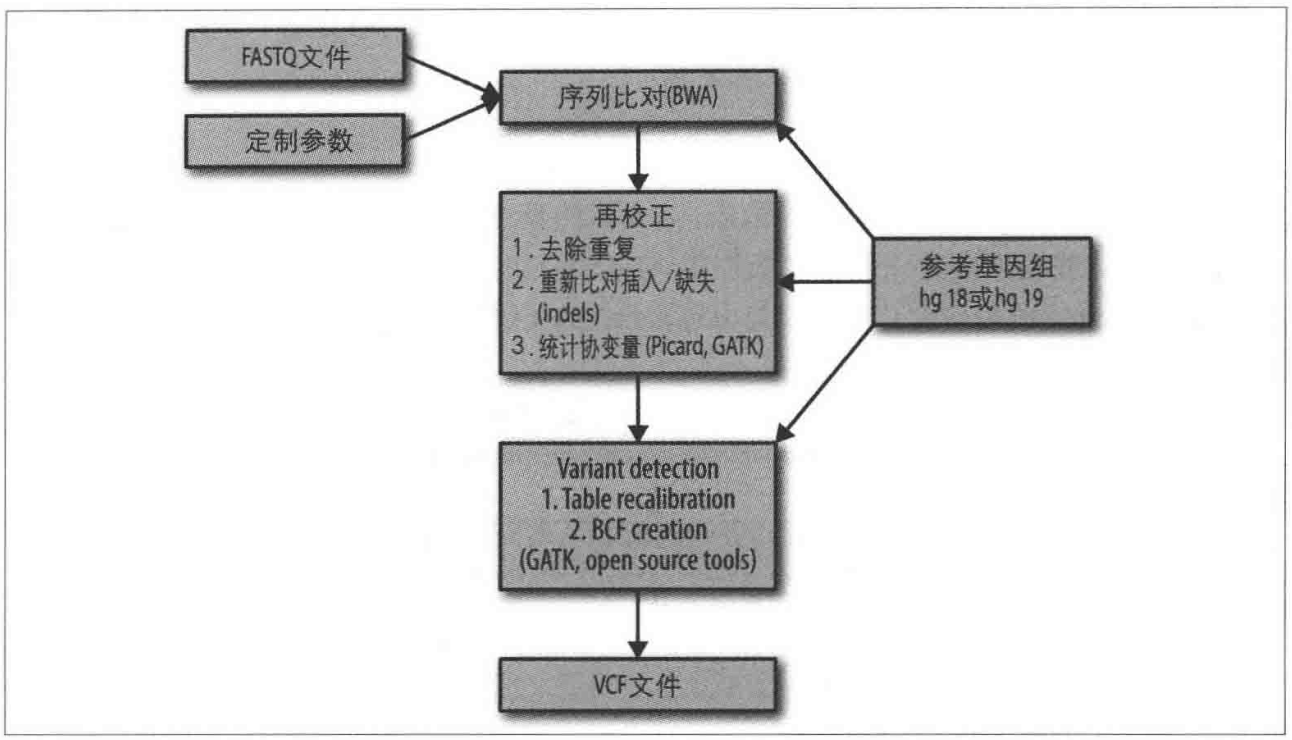



图18-2：DNA测序管道

需要分析和应用DNA测序的数据量相当庞大（*http://www.1000genomes.org/data*），另外有大量开源算法可以完成前面的4个步骤。需要注意，在这些开源工具中，你的选择对最终结果会有显著的影响，所以一定要充分了解这些工具，这一点非常重要。

# DNA测序的输入数据

DNA测序数据最常用的格式是FASTQ，这是一种基本文本的格式，用于存储一个生物序列及其质量评分。对于一个给定的FASTQ文件，每4行表示一个DNA序列。

FASTQ文件的一般语法如下：

```
<fastq>:= <block>+
<block>:=@<seqname>\n<seq>\n[<seqname>]\n<qual>\n
<seqname>:= [A-Za-z0-9_.:-]+
<seq>:= [A-Za-z\n\.~]+
<qual>:= [!-~\n]+
```

下面给出一个例子：

```
@NCYC361-11a03.q1k bases 1 to 1576
GCGTGCCCGAAAAAATGCTTTTGGAGCCGCGCGTGAAAT...
+NCYC361-11a03.q1k bases 1 to 1576
!)))))****(((***%%((((*(((+,**(((+**+,-...
```

FASTQ数据可以是成对的也可以不成对。如果是成对的，那么DNA测序的输入就是一个文件对：*left_file.fastq*和*right_file.fastq*。如果不成对，就只有一个文件：*file.fastq*。

既然对输入数据已经有所了解，下面再来仔细分析DNA测序管道的前几个步骤。然后我们会介绍MapReduce如何帮助我们解决DNA测序问题。

# 输入数据验证

DNA测序管道中的第一步是验证FASTQ文件的格式。通过验证，要检验输入文件的质量。利用输入数据验证工具，可以对来自高通量测序管道的原始序列数据（如采用FASTQ文件格式）完成一些质量控制检查。

有很多开源工具可以完成输入数据验证。例如，对于FASTQ验证，可以有以下选择：

- FastQValidator（*http://bit.ly/fastqvalidator*）。
- FastQC（*http://bit.ly/FastQC*）。

输入数据验证步骤非常简单，也很直接，所以这里不做过多的介绍。我们的重点是使用MapReduce的映射/比对、再校正和变异检测算法。

# DNA序列比对

序列比对（Sequence alignment）是指比较两个或多个DNA或蛋白质序列。序列比对的主要用途是明确序列之间的相似性。

对于全局序列比对，考虑下面的例子，这里有两个输入序列，它们使用相同的字母表：

- 序列1：GCGCATGGATTGAGCGA
- 序列2：TGCGCCATTGATGACCA

输出是这两个序列的一个可能的比对：

- **-GCGC-ATGGATTGAGCGA**
- **TGCGCCATTGAT-GACC-A**

可以观察到这个可能的比对输出中有3种元素：

- 完全匹配（用**粗体**显示）。
- 不匹配（加下划线）。
- 插入和缺失（称为*indels*，没有特殊格式）。

在比对阶段，我们将使用MapReduce/Hadoop并结合以下开源工具：

- Burrows-Wheeler Aligner (BWA)，这是一个很高效的程序，可以根据一个长参考序列（如人类基因组）比对相对短的核甘酸序列（参见*http://bio-bwa.sourceforge.net/*）。
- Sequence Alignment/Map(SAM)工具，提供了一组工具处理SAM格式序列的比对，包括排序、合并、索引和生成逐位置格式的比对（参见*http://samtools.sourceforge.net/*）。

我们将处理BAM格式的文件，这是SAM文件的二进制格式。

# DNA测试的MapReduce算法

对于一个约400～900 GB的数据样本（采用FASTQ文件格式），在一个计算能力很强的服务器上，一般的DNA测序需要70小时以上的时间才能完成。这里的MapReduce算法的目标是在几小时内就找出答案，而且要保证这个解决方案具有可伸缩性。

由于完成比对、再校正和变异检测的大多数开源工具（如BWA、SAMtools和GATK（*https://www.broadinstitute.org/gatk/*））都提供了Linux命令行界面，所以各个MapReduce阶段中的`map()`和`reduce()`函数将调用Linux shell脚本，并提供适当的参数。要执行这些shell脚本，我们将使用FreeMarker模板语言（*http://freemarker.org/*），用一个模板合并Java对象和数据结构来创建一个适当的shell脚本（见图18-3）。为了区分不同的DNA序列，每次分析时，将指定并使用一个唯一的GUID，称为“分析ID”（analysis ID），这可以帮助我们适当地组织输入和输出目录。

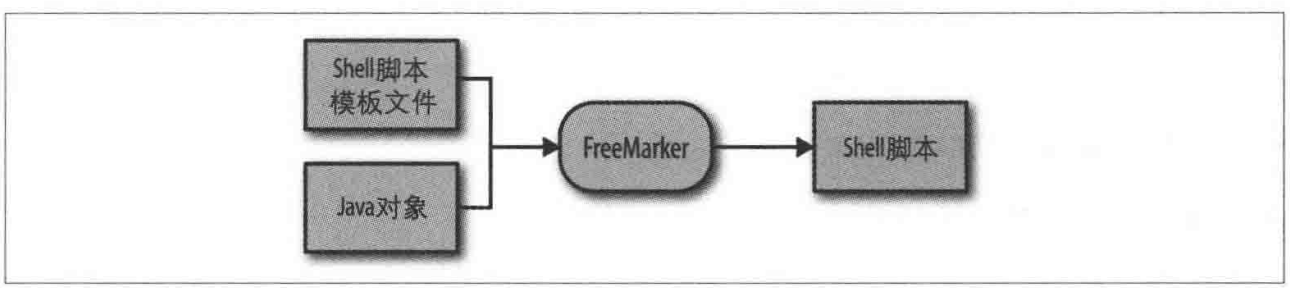



图18-3：FreeMarker模板引擎

MapReduce解决方案分为3步，如图18-4和图18-5所示。它们分别对应这一章最前面介绍的DNA测序管道中的步骤1～3。

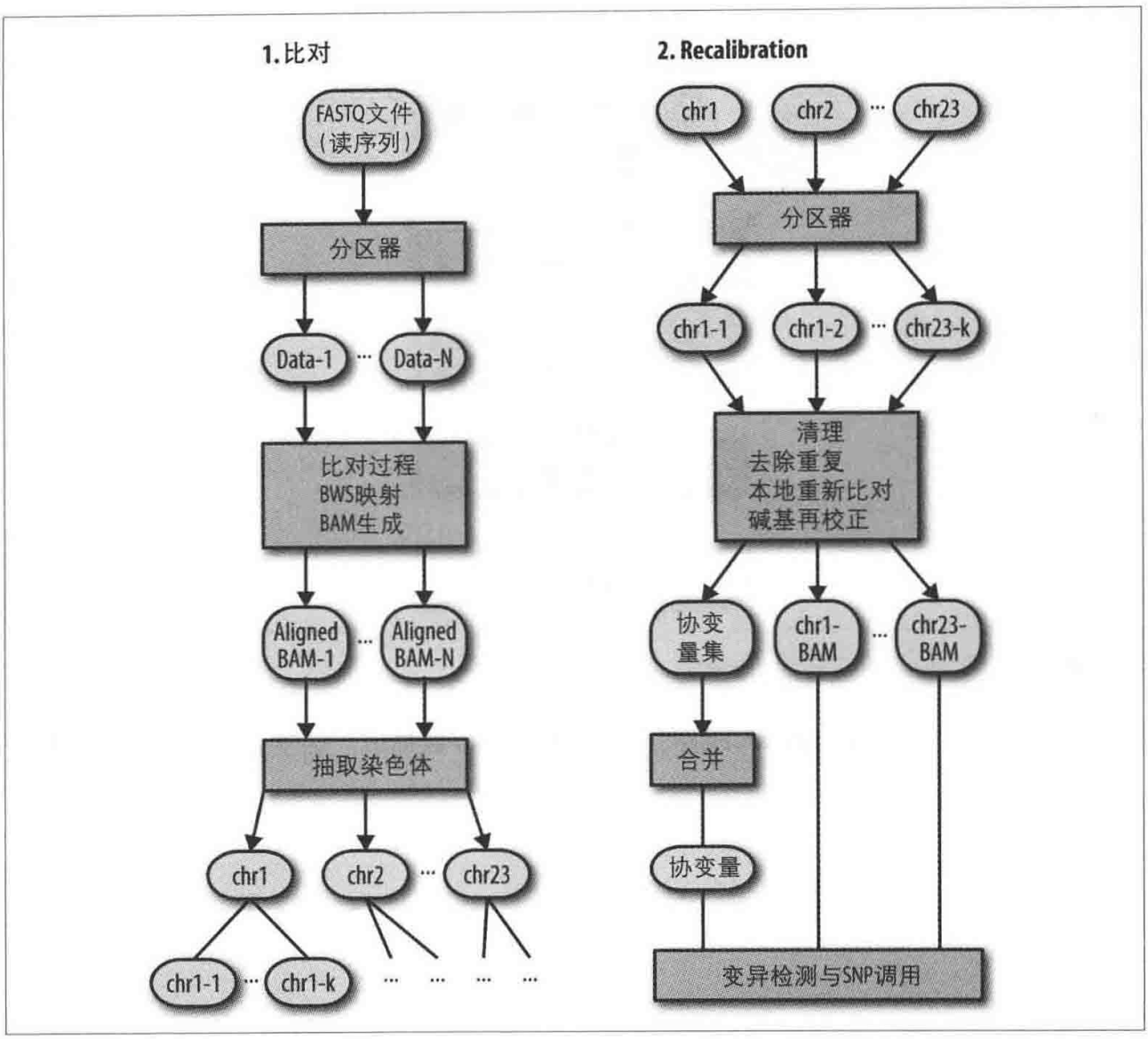



图18-4：MapReduce解决方案（步骤1和2）

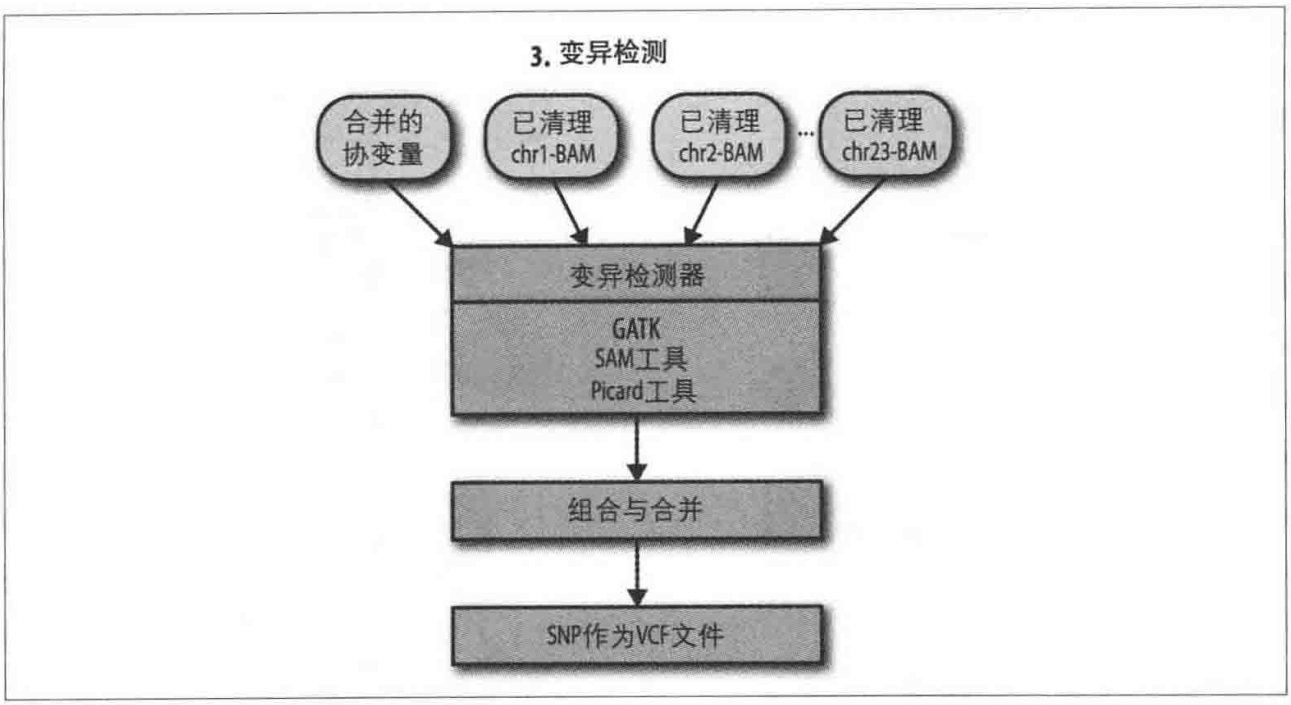



图18-5：MapReduce解决方案 (步骤3)

图18-6给出了这个3步MapReduce解决方案的DNA测序数据流。这里显示了各个阶段如何划分和合并数据。

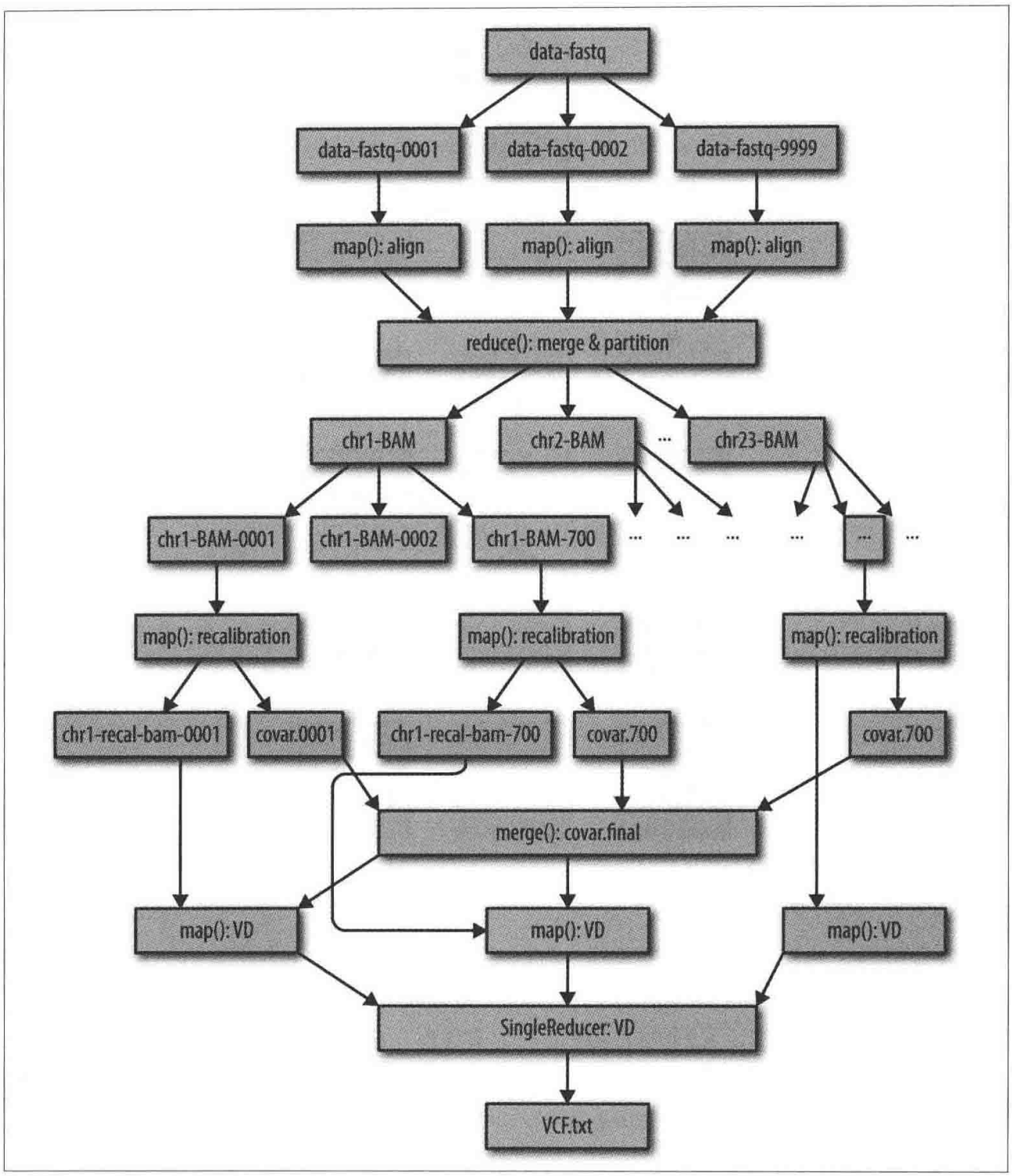



图18-6：DNA测序数据流

## 步骤1：比对

比对阶段的高层工作流如图18-7所示。

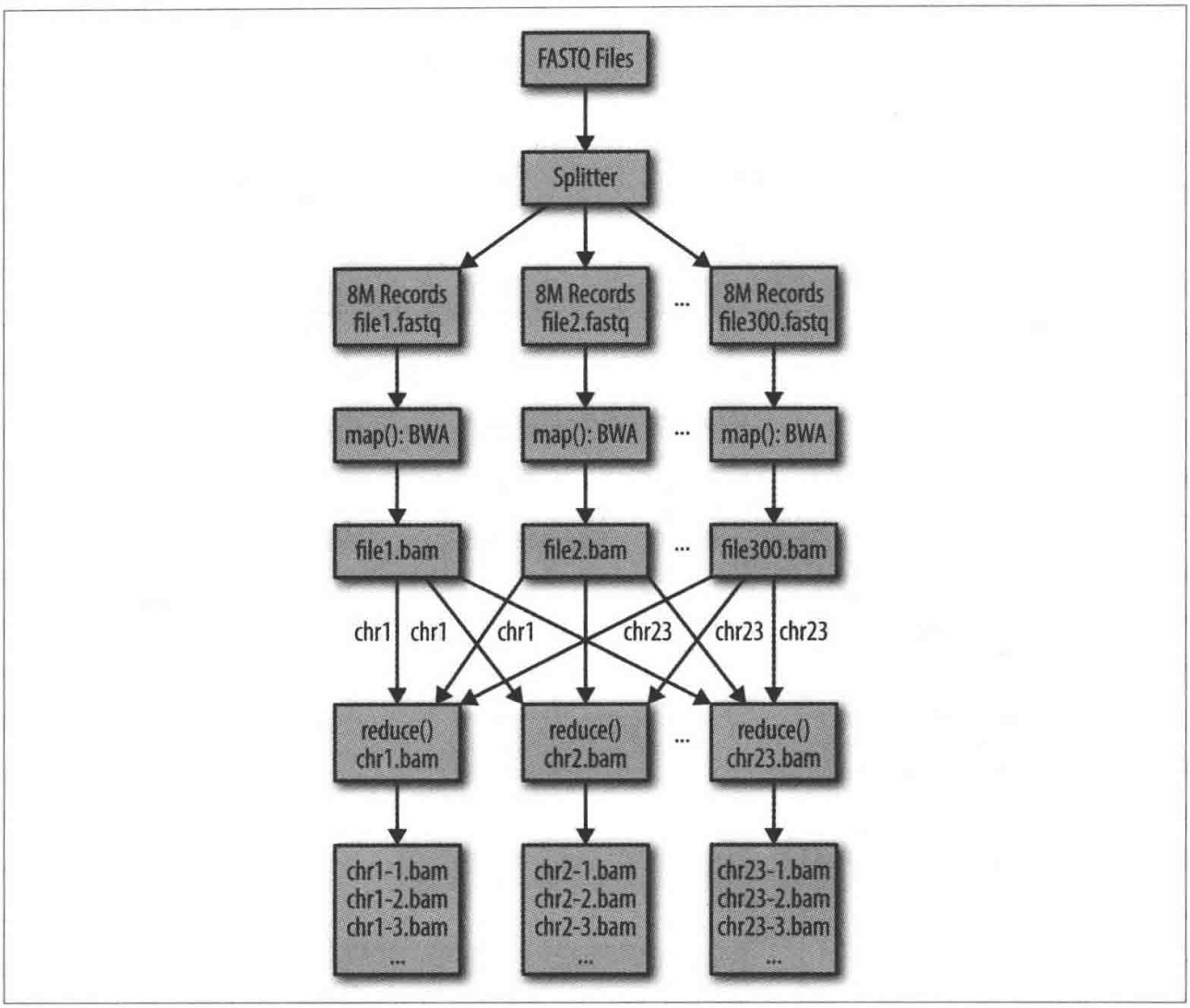



图18-7：DNA测序比对工作流

比对步骤开始之前，要把DNA序列FASTQ文件分区为包含800万行的集合（或者200万个序列；应该记得，采用FASTQ格式时，每4行一组表示一个DNA序列）。前面提到过，如果FASTQ数据是成对的，我们的输入就是一个文件对：*left_file.fastq*和*right_file.fastq*。如果不成对，那么输入将是一个文件：*file.fastq*。成对数据要如下分区（对于成对数据，比对`map()`函数会同时处理*left_file.fastq.NNNN*和*right_file.fastq.NNNN*）：

```
left_file.fastq.0000 right_file.fastq.0000
left_file.fastq.0001 right_file.fastq.0001
left_file.fastq.0002 right_file.fastq.0002
...                  ...
```

不成对数据要如下分区（对于不成对数据，比对`map()`函数会处理*file.fastq.NNNN*）：

```
file.fastq.0000
file.fastq.0001
file.fastq.0002
...
```

每个分区（也称为块）由一个map()函数处理。需要注意，这里使用的分区大小为800万行，等于200万个序列，这只是为了介绍有关概念，分区的实际大小应当由Hadoop集群的大小来确定。也就是说，如果你的集群有50个节点，每个节点可以处理4个映射器，就可以把FASTQ文件分割为200个分区。例如，如果总的输入大小约为400 GB，就要将输入分区为2GB个块（这样一来，就能最大程度地利用你的所有映射器）。map()函数会读取输入文件（一个块），并生成一个BAM格式的比对文件。在这里map()函数使用BWA来完成比对过程。完成比对之后，它会抽取所有染色体（1, 2, ..., 22, 23[注2]），把它们保存在MapReduce文件系统中（对于Hadoop就是HDFS）。例如，如果我们有800个分区，那么每个染色体就会生成800个文件（23 × 800 = 18400个文件）。但只有23个归约器（每个染色体对应一个归约器）。归约器会连接（合并和排序）对应一个特定染色体ID的所有染色体。所有染色体1都会连接到一个名为*chr1.bam*的文件中，所有染色体2会连接到一个名为*chr2.bam*的文件中，依此类推。然后每个归约器将合并得到的BAM文件分区为小文件，用作为再校正阶段的输入。

## 比对阶段的映射器

对于示例18-1所示的比对映射器，我们的解决方案接收FASTQ格式的文件作为输入，生成染色体分区（chr1,chr2, ..., chr22, chr23）。

示例18-1：比对阶段： map()函数

```
/**
 * @param key是MapReduce框架生成的一个键
 * @param value是一个分区的FASTQ文件（可能为8M行=2M序列）
 */
map(key, value) {
   // note: chr23 = concat(chrX, chrY, chrM)
   alignedBAMFile = alignByBWA(value);
   (chr1File, chr2File, ..., ch23File) = partitionByChromosome(alignedFile);
   for (i=1, i < 24; i++) {
      emit(chr<i>, chr<i>File);
   }
}
```

alignByBWA()函数接受一个分区的FASTQ文件，完成比对，最后按染色体将比对文件分区。所有这些动作都由一个shell脚本模板完成。示例18-2列出了这个模板的一些部分。

示例18-2：比对阶段：不成对输入

```
#!/bin/bash
...
export BWA=<bwa-install-dir>/bwa
export SAMTOOLS=<samtools-install-dir>/samtools
export BCFTOOLS=<bcftools-install-dir>/bcftools
```

注2： 并没有染色体23，不过可以连接染色体X, Y和M，把得到的结果称为染色体23。

```
 6 export VCFUTILS=<bcftools-install-dir>/vcfutils.pl
 7 export HADOOP_HOME=<hadoop-install-dir>
 8 export HADOOP_CONF_DIR=<hadoop-install-dir>/conf
 9 ...
10 # 数据目录
11 export TMP_HOME=<root-tmp-dir>/tmp
12 export BWA_INDEXES=<root-index-dir>/ref/bwa
13 ...
14 # 定义参考基因组
15 export REF=<root-reference-dir>/hg19.fasta
16
17 ### 步骤1：比对
18 # KEY唯一标识输入文件
19 KEY={key}
20 # input_file
21 export INPUT_FILE=${input_file}
22 export ANALYSIS_ID=${analysis_id}
23 NUM_THREAD=3
24 cd $TMP_HOME
25 $BWA aln -t $NUM_THREAD $REF $INPUT_FILE > out.sai
26 $BWA samse -r $REF out.sai $INPUT_FILE | $SAMTOOLS view -Su -F 4 - | \
27      $SAMTOOLS sort - aln.flt
28
29 # 开始索引aln.flt.bam文件
30 $SAMTOOLS index aln.flt.bam
31
32 # 将比对数据分区
33 for i in {1..22}
34 do
35    CHR=chr$i
36    $SAMTOOLS view -b -o $CHR.bam aln.flt.bam $CHR
37    output_file=/genome/dnaseq/output/$ANALYSIS_ID/$CHR/$KEY.$CHR.bam
38    $HADOOP_HOME/bin/hadoop fs -put $CHR.bam $output_file
39 done
40
41 # 对 X、Y 和 M染色体做同样的处理
42 $SAMTOOLS view -b -o chr23.bam aln.flt.bam chrX chrY chrM
43 output_file=/genome/dnaseq/output/$ANALYSIS_ID/chr23/$KEY.chr23.bam
44 $HADOOP_HOME/bin/hadoop fs -put chr23.bam $output_file
45
46 exit 0
```

上面提供的shell脚本只处理不成对的数据。如果输入文件是成对的，就要把第25～27行替换为示例18-3中所示的代码。

示例18-3：比对阶段：成对输入

```
25 $BWA aln -t $NUM_THREAD $REF $INPUT_FILE_1 > out1.sai
26 $BWA aln -t $NUM_THREAD $REF $INPUT_FILE_2 > out2.sai
27 $BWA sampe -r $INFO_RG $REF out1.sai out2.sai $INPUT_FILE_1 $INPUT_FILE_2 | \
28      $SAMTOOLS view -Su -F 4 - | $SAMTOOLS sort - aln.flt
```

### 比对阶段的归约器

比对阶段会有23个归约器（每个染色体对应一个归约器）。归约器键是一个组合键<chrID><;><analysisID>，其中，染色体ID标记为{01, 02, 03, ..., 23}。需要说明，染色体ID为23时，这包括chrM、chrX和chrY。每个归约器会把所有已比对的.bam 文件合并到一个*chr<i>.bam*文件：

```
chr<i>.bam = merge the following files:
             chr<i>.bam.0000
             chr<i>.bam.0001
             ...
             chr<i>.bam.0437
             ...
```

将所有文件合并到一个*chr<i>.bam*文件之后，我们将*chr<i>.bam*分区为多个小*.bam*文件，作为步骤2再校正映射器的输入。分区文件将是：

```
chr<i>.bam.j (j = 1, 2, 3, ..., 100+)
```

比对阶段的Reduce()函数如示例18-4所示。

示例18-4：比对阶段：reduce()函数

```
/**
 * @param key是一个<chrID><;><analysis_id>
 * 这里chrID 为(1, 2, 3, ..., 23)中一个ID
 * @param value将忽略（未使用）
 */
reduce(key, value) {
   DNASeq.mergeAllChromosomesAndPartition(key);
}
```

所有工作都在DNASeq.mergeAllChromosomesAndPartition()方法中完成，它会合并对应一个特定染色体的所有已比对.bam文件（参见示例18-5）。前面提到过，将对最后合并得到的文件进行分区，由再校正阶段（步骤2）进一步处理。

示例18-5：mergeAllChromosomesAndPartition()方法

```
/**
 * reducerKey=<chrID>;<analysis_id>
 * 在这里chrID=1, 2, ..., 22, 23 (23包括chrM, chrX, chrY)
 */
public static void mergeAllChromosomesAndPartition(String reducerKey)
   throws Exception {
   // 分解行：每一行有两个字段（字段由";"分隔)
   String[] tokens = reducerKey.split(";");
   String chrID = tokens[0];
   String analysisID = tokens[1];
   Map<String, String> templateMap = new HashMap<String, String>();
   templateMap.put("chr_id", chrID);
   templateMap.put("analysis_id", analysisID);
   mergeAllChromosomesBamFiles(templateMap);
```

```
15    partitionSingleChromosomeBam(templateMap);
16 }
```

从mergeAllChromosomesAndPartition()方法可以看到，这两个辅助方法mergeAllChromosomesBamFiles()和partitionSingleChromosomeBam()都使用FreeMarker模板引擎来传递所需的Java对象，然后代表归约器执行shell脚本。mergeAllChromosomesBamFiles()方法的定义参见示例18-6。

示例18-6：mergeAllChromosomesBamFiles()方法

```
/**
 * 这个方法会合并以下文件，并创建一个chr<i>.bam文件
 * 其中i属于{1, 2, ..., 23}:
 *
 * HDFS: /.../chr<i>/chr<i>.bam.0000
 * HDFS: /.../chr<i>/chr<i>.bam.0001
 * ...
 * HDFS: /.../chr<i>/chr<i>.bam.0437
 *
 * 然后合并所有这些 (.0000, .0001, ..., .0437) 文件，并把结果
 * 保存在/data/tmp/<analysis_id>/chr<i>/chr<i>.bam中。
 *
 * 一旦创建chr<i>.bam，将它分区为小.bam文件，
 * 这些小文件将输入到RecalibrationDriver(DNA测序的步骤2)
 *
 */
public static void mergeAllChromosomesBamFiles(Map<String, String> templateMap)
   throws Exception {
   TemplateEngine.initTemplatEngine();
   String templateFileName = <freemarker-template-file-as-a-bash-script>;
   // 从模板文件创建具体的脚本
   String chrID = templateMap.get("chr_id");
   String analysisID = templateMap.get("analysis_id");
   String scriptFileName = createScriptFileName(chrID, analysisID);
   String logFileName = createLogFileName(chrID, analysisID);
   File scriptFile = TemplateEngine.createDynamicContentAsFile(templateFileName,
                                                               templateMap,
                                                               scriptFileName);
   if (scriptFile != null) {
      ShellScriptUtil.callProcess(scriptFileName, logFileName);
   }
}
```

TemplateEngine.createDynamicContentAsFile()方法会完成所有魔法：它取两个输入（templateFileName和templateMap），生成一个scriptFileName。基本说来，所有参数都要传递到templateFileName，然后生成一个新的shell脚本scriptFileName，再代表归约器执行这个脚本。这里有两个重要的类ShellScriptUtil和TemplateEngine需要进一步讨论。ShellScriptUtil.callProcess()方法接受一个shell脚本文件（第一个参数），这是它要执行的脚本文件。然后将执行脚本的所有日志写至一个日志文件（第二个参数）。日志记录是异步的，这说明，在你执行脚本的同时可以立即访问这个日志文件。

示例18-7中定义了TemplateEngine类。它实现了模板引擎的基本概念：接受一个模板（包含键占位符的文本文件）及键-值对（Java映射），然后创建一个全新的文件，这个文件中将把模板中的所有键替换为具体的值。

示例18-7：TemplateEngine类

```
import java.io.File;
import java.io.Writer;
import java.io.FileWriter;
import java.util.Map;
import java.util.concurrent.atomic.AtomicBoolean;
import freemarker.template.Template;
import freemarker.template.Configuration;
import freemarker.template.DefaultObjectWrapper;

/**
 * 这个类使用FreeMarker (http://freemarker.sourceforge.net/).
 * FreeMarker是一个模板引擎，这是一个根据模板生成文本输出的通用工具
 * (从shell脚本到自动生成的源代码都是文本输出)
 * 这是一个Java包，面向Java程序员的一个类库。
 * 它本身并不面向最终用户，
 * 要由程序员嵌入在他们的程序中。
 *
 * @author Mahmoud Parsian
 *
 */
public class TemplateEngine {

   // 通常在整个应用生命周期中只执行一次
   private static Configuration TEMPLATE_CONFIGURATION = null;
   private static AtomicBoolean initialized = new AtomicBoolean(false);

   // 下面的模板目录将从配置文件加载
   private static String TEMPLATE_DIRECTORY = "/home/dnaseq/template";

   public static void init() throws Exception {
        if (initialized.get()) {
        // 已经初始化并返回...
        return;
        }
        initConfiguration();
        initialized.compareAndSet(false, true);
   }

   static {
      if (!initialized.get()) {
         try {
            init();
         }
         catch(Exception e) {
            theLogger.error("TemplateEngine init failed at initialization.", e);
         }
      }
   }
```

```
49
50    // 这支持一个模板目录
51    private static void initConfiguration() throws Exception {
52       TEMPLATE_CONFIGURATION = new Configuration();
53       TEMPLATE_CONFIGURATION.setDirectoryForTemplateLoading(
54          new File(TEMPLATE_DIRECTORY));
55       TEMPLATE_CONFIGURATION.setObjectWrapper(new DefaultObjectWrapper());
56       TEMPLATE_CONFIGURATION.setWhitespaceStripping(true);
57       // 如果设置了这个属性，未定义的键将设置为""
58       TEMPLATE_CONFIGURATION.setClassicCompatible(true);
59    }
60
61    public static File createDynamicContentAsFile(...){...}
```

示例18-8中定义了TemplateEngine类最重要的方法createDynamicContentAsFile()。这个方法接受一个包含键占位符的模板文件及一个键-值对集合，然后生成一个新文件，将键占位符替换为给定的键。

示例18-8：TemplateEngine.createDynamicContentAsFile()方法

```
1     /**
2      * @param templateFile是一个模板文件名，如script.sh.template
3      * @param keyValuePairs是一个(K,V)键值对集合
4      * @param outputFileName是从templateFile生成的文件名
5      */
6     public static File createDynamicContentAsFile(String templateFile,
7                                                    Map<String,String> keyValuePairs,
8                                                    String outputFileName)
9        throws Exception {
10       if ((templateFile == null) || (templateFile.length() == 0)) {
11          return null;
12       }
13
14       Writer writer = null;
15       try {
16          // 创建一个模板： 例如"cb_stage1.sh.template2"
17          Template template = TEMPLATE_CONFIGURATION.getTemplate(templateFile);
18          // 合并数据模型和模板
19          File outputFile = new File(outputFileName);
20          writer = new BufferedWriter(new FileWriter(outputFile));
21          template.process(keyValuePairs, writer);
22          writer.flush();
23          return outputFile;
24       }
25       finally {
26         if (writer != null) {
27            writer.close();
28         }
29       }
30    }
31 }
```

## 步骤2：再校正

再校正（Recalibration）是MapReduce DNA测序管道的第2个步骤。在再校正步骤中，每个`map()`函数处理一个特定的已比对染色体。映射器会完成重复标记、本地重新比对和再校正。`map()`的目标是创建一个包含协变量（covariates）的本地再校正表。各个归约器将合并这些本地协变量来创建最终的全局文件（再校正表），这个全局文件将在DNA测序的第3步（也就是最后的一个步骤）变异检测的`map()`函数中使用。

再校正MapReduce算法（数据流）如图18-8所示。

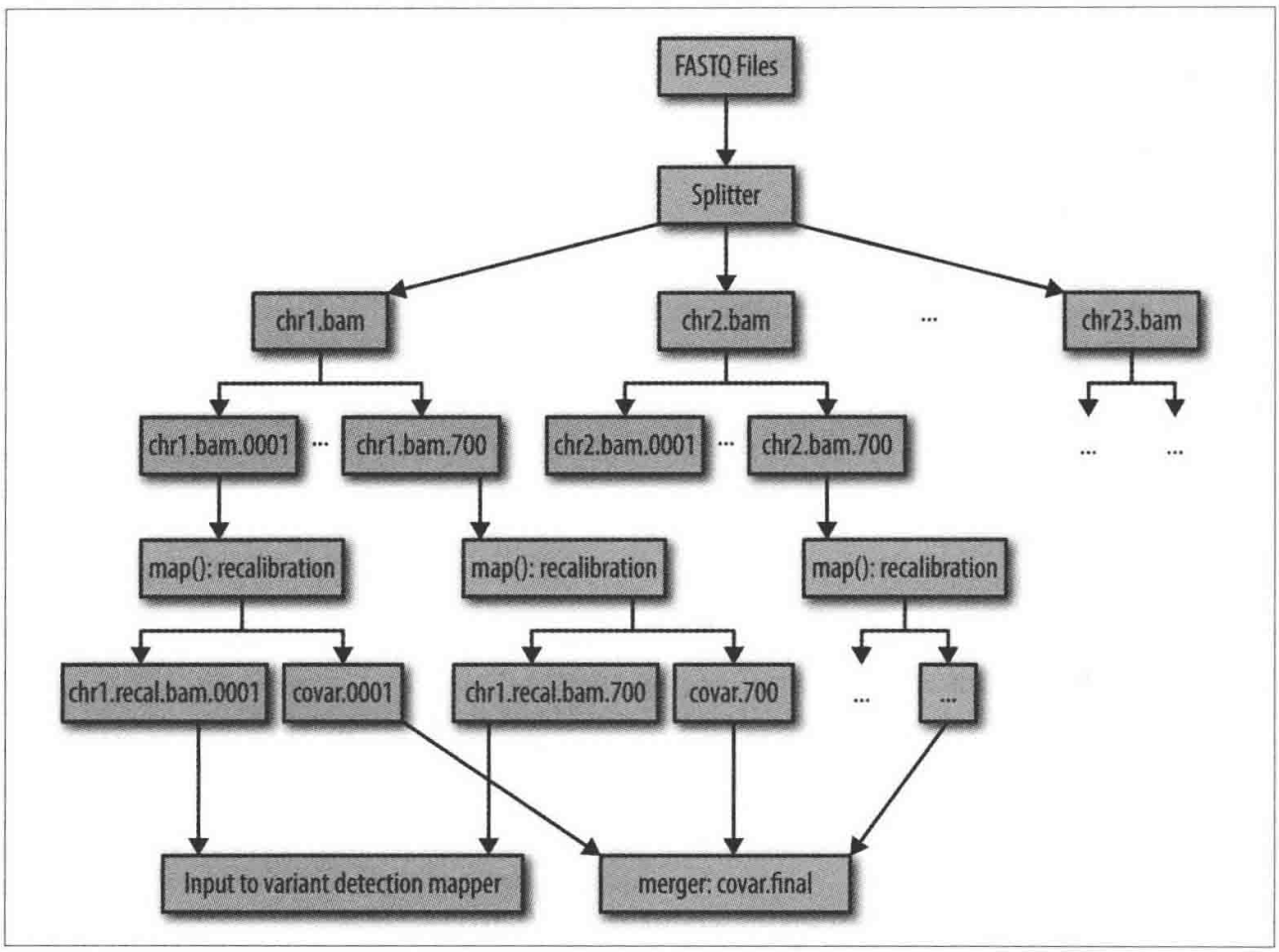



图18-8：DNA测序：再校正

比对阶段之后，我们要创建将由再校正映射器使用的特殊元数据。这个元数据有以下格式：

```
<counter><;><partitioned-bam-file><;><ref_genome><;><analysis_id>
```

其中：

- `<counter>`是一个自动生成的数字序列0000, 0001, 0002, ....
- `<partitioned-bam-file>`是一个已分区的比对文件块。

- <ref_genome>指示hg18或hg19。
- <analysis_id>是完成DNA测序的一个GUID（用来区分不同的分析）。

下面给出一些示例输入：

```
0001;chr07.bam.0001;hg19;208
0002;chr07.bam.0002;hg19;208
0003;chr07.bam.0003;hg19;208
...
```

再校正映射器如示例18-9所示。

示例18-9：再校正阶段：map()函数

```
// key 由MR生成，在这里忽略
// value 为: <counter><;><partitioned-bam-file><;><ref_genome><;><analysis_id>
map(key, value) {
   // 具体文件位置为：
   // /data/dnaseq/align/ANALYSIS_ID/merged.bam.<KEY>
   Map<String, String> tokens =
      DNASeq.tokenizeRecalibrationMapperInput(value);
   String reducerKey = tokens.get("analysis_id");
   DNASeq.recalibrationMapper(tokens);
   emit(reducerKey, value);
}

public static void recalibrationMapper(Map<String, String> templateMap)
   throws Exception {
   TemplateEngine.init();
   // 从一个模板文件创建具体的脚本
   String key = templateMap.get("key");
   String analysisID = templateMap.get("analysis_id");
   String scriptFileName = createScriptFileName("recalibration_mapper",
                                                key,
                                                analysisID);
   String logFileName = createLogFileName("recalibration_mapper",
                                          key,
                                          analysisID);
   File scriptFile = TemplateEngine.createDynamicContentAsFile(
        "recalibration_mapper.template",
        templateMap,
        scriptFileName);
   if (scriptFile != null) {
      ShellScriptUtil.callProcess(scriptFileName, logFileName);
   }
}
```

对应每个<analysis_id>只有一个归约器（参见示例18-10）。

示例18-10：再校正阶段：reduce()函数

```
// key: analysisID
// values: 忽略
reduce(key, Iterable<Object> values) {
```

```
4    DNASeq.recalibrationReducer(key);
5    emit(key, key);
6 }
```

示例18-11定义了recalibrationReducer()方法。

示例18-11：recalibrationReducer()方法

```
public static void recalibrationReducer(String analysisID)
   throws Exception {
   TemplateEngine.init();
   String[] tokens = valueAsString.split(";");
   Map<String, String> templateMap = new HashMap<String, String>();
   templateMap.put("key", "-"); // key未定义
   templateMap.put("analysis_id", key);
   // 从一个模板文件创建具体的脚本
   String scriptFileName = createScriptFileName("recalibration_reducer",
                                                analysisID);
   String logFileName = createLogFileName("recalibration_reducer",
                                          analysisID);
   File scriptFile = TemplateEngine.createDynamicContentAsFile(
      "recalibration_reducer.template",
      templateMap,
      scriptFileName);
   if (scriptFile != null) {
      ShellScriptUtil.callProcess(scriptFileName, logFileName);
   }
}
```

再校正映射器模板的定义参见示例18-12。

示例18-12：再校正映射器模板

```
#!/bin/bash

###
### 再校正映射器模板
###
### 一旦创建recal.table.csv，
### 调用snp(获得变异)来计算 ...recal.table.csv，这将保存在HDFS
### 输入文件：已比对的bam文件（从一个chr<i>.bam分区得到）
...
##
## 输入文件：已比对的bam文件（从一个chr<i>.bam分区得到）
## 将HDFS_BAM_FILE复制到LOCAL_BAM_FILE
HDFS_BAM_FILE=${hdfs_bam_file}
BAM_FILE='basename $HDFS_BAM_FILE';
$HADOOP_HOME/bin/hadoop fs -copyToLocal $HDFS_BAM_FILE.
...
#
# 将4.recal.table.csv放在GLOBAL/SHARED目录中
#
export SHARED_RECAL_DIR=/dnaseq/recal/${analysis_id}
...
## 标记重复
```

```
java       -Xmx4g\
       -Djava.io.tmpdir=$JAVA_IO_TMPDIR \
       -jar $PICARD_JAR/MarkDuplicates.jar \
       I=$BAM_FILE \
       O=2.mark.out.bam \
       M=2.mark.out.metrics \
       AS=true

## 本地重新比对
samtools index 2.mark.out.bam
java       -Xmx4g \
       -Djava.io.tmpdir=$JAVA_IO_TMPDIR \
       -jar $GATK_JAR/GenomeAnalysisTK.jar \
       -T IndelRealigner \
       -I 2.mark.out.bam \
       -o 3.realigned.out.bam \
       -R $REF \
       -targetIntervals $DBSNP/dbsnp_indel.intervals \
       -known $DBSNP/dbsnp_indel.vcf \
       --consensusDeterminationModel KNOWNS_ONLY \
       -LOD 0.4
## 碱基质量再校正
java       -Xmx4g \
       -Djava.io.tmpdir=$JAVA_IO_TMPDIR \
       -jar $GATK_JAR/GenomeAnalysisTK.jar \
       -T CountCovariates \
       -I 3.realigned.out.bam \
       -recalFile 4.recal.table.csv \
       -R $REF \
       -knownSites $DBSNP/dbsnp.vcf \
       -cov QualityScoreCovariate \
       -cov ReadGroupCovariate \
       -cov PositionCovariate \
       -cov DinucCovariate
# 将结果复制到共享目录
cp -f 4.recal.table.csv $SHARED_RECAL_DIR/$KEY.4.recal.table.csv

##
## 还需要保存3.realigned.out.bam (
## variant_detection_mapper.sh.template中需要用到)
##
cp -f 3.realigned.out.bam $SHARED_RECAL_DIR/$BAM_FILE.3.realigned.out.bam
###
### 所以可以得到：
### $SHARED_RECAL_DIR/$KEY.4.recal.table.csv 对应KEY=1, 2, 3, ....
### $SHARED_RECAL_DIR/$BAM_FILE.3.realigned.out.bam 对应 KEY=1, 2, 3, ....
###  （将是variant_detection_mapper.sh.template的输入）
###
```

再校正归约器模板的定义参见示例18-13。

示例18-13：再校正归约器模板

```
#!/bin/bash
###
```

```
### 将所有 -.4.recal.table.csv 文件（由各个.bam文件生成）
### 合并为一个recal.table.merged.final.txt文件。
###
### 一旦创建了recal.table.merged.final.txt ，它将保存在
### /dnaseq/recal/${analysis_id}/ ，并输入到VariantDetectionMapper。
###
...
#
# 所有 -.4.recal.table.csv文件都在$SHARED_RECAL_DIR目录中。
#
export SHARED_RECAL_DIR=/dnaseq/recal/$ANALYSIS_ID/
recal_files='find $SHARED_RECAL_DIR -name '*.4.recal.table.csv' | sort'
num_of_recal_files='find $SHARED_RECAL_DIR -name '*.4.recal.table.csv' | wc -1'
...
### 注意：所有计算将在$SHARED_RECAL_DIR中完成
# 准备java输入文件
java_input_files=""
for file in $recal_files
do
      echo "preparing java input file=$file"
      java_input_files="$file $java_input_files"
done
...
cd $SHARED_RECAL_DIR
current_dir='pwd'
export MERGE_COVARIATES=JavaMergeCovariates
$JAVA_HOME/bin/java -Xms4g -Xmx12g $MERGE_COVARIATES \
-i "$java_input_files" -o recal.txt.unsorted
#
# 相应地对文件排序
#
/bin/sort -t, -k 2,2n -k3,3n -k4,4 recal.txt.unsorted > recal.txt.sorted
#
# recal.txt.sorted文件将在变异检测映射器中使用
```

## 步骤3：变异检测

变异检测（也称为SNP调用）是DNA测序的最后一个阶段。这一步的目标是生成VCF格式的变异（VCF表示变体调用格式（variant call format），由1000 Genomes Project开发）。 map()函数将使用再校正步骤中map()函数生成的BAM文件，以及最后的“再校正表”文件。这个map()函数会使用开源工具（如GATK和SAMtools）生成部分变异（原始BCF（二进制调用格式，binary call format）文件）。归约器将连接（排序和合并）这些原始BCF文件，生成一个VCF文件。一旦创建VCF文件，可以由很多分析算法来使用，如等位基因频率（第21章介绍）、家族分析和Cochran-Armitage（科克伦-阿米蒂奇）趋势检验。

变异检测过程中要查找NGS（下一代测序）数据中与参考基因组（如hg18或hg19）不同的碱基（hg18或hg19是人类基因组版本），并确定相应的参考注释（有关的详细信息参见*http://bit.ly/build_36_1_genome*）。

## 变异检测阶段的映射器

这个映射器接受一个分块的“已比对.*bam*”文件，并对这个文件完成以下转换：

- 碱基质量再校正。
- 变异调用和过滤。

这部分工作由DNASeq.theVariantDetectionMapper()方法完成，它接受所需的参数，并从一个给定的模板创建适当的shell脚本，最后会执行这个shell脚本。变异检测阶段的映射器如示例18-14所示。

示例18-14：变异检测阶段：map()函数

```
// key: 忽略，未使用
// value: <counter><;><3.realigned.out.bam.<key>><;><ref_genome><;><analysis_id>
// index < 0 > < 1 > < 2 > < 3 >
// value example-1: 0001;/<dir>/realigned.out.bam.0001;hg19;208
// value example-2: 0007;/<dir>/realigned.out.bam.0007;hg19;208
// 注意：变异检测将有一个归约器：
// 归约器输出的键将是<analysis_id>
map(key, value) {
   Map<String, String> tokens = DNASeq.tokenizeTheVariantDetectionMapper(value);
   String reducerKey = tokens.get("analysis_id");
   DNASeq.theVariantDetectionMapper(tokens);
   emit(reducerKey, reducerKey);
}
```

theVariantDetectionMapper()方法如示例18-15所示，它接受analysis_id作为参数(这标识一次DNA测序分析的所有文件目录)。

示例18-15：theVariantDetectionMapper()方法

```
public static void theVariantDetectionMapper(Map<String, String> templateMap)
   throws Exception {
   TemplateEngine.init();
   // 从一个模板文件创建具体的脚本
   String scriptFileName = "/dnaseq/variant_detection_mapper_" +
     templateMap.get("analysis_id") + "_" + templateMap.get("key") +".sh";
   String logFileName = "/dnaseq/variant_detection_mapper_" +
     templateMap.get("analysis_id") + "_" + templateMap.get("key") +".log";
   File scriptFile = TemplateEngine.createDynamicContentAsFile(
      "variant_detection_mapper.template",
      templateMap,
      scriptFileName);
   if (scriptFile != null) {
      ShellScriptUtil.callProcess(scriptFileName, logFileName);
   }
}
```

示例18-16给出了*variant_detection_mapper.template*的部分代码。

示例18-16：变异检测映射器模板

```
#!/bin/bash
...

# 1 完成碱基质量再校正：
# GATK要求BAM文件扩展名必须是.bam
samtools index $REALIGNED_OUT_BAM_FILE
#
java -Xmx4g \
    -Djava.io.tmpdir=$JAVA_IO_TMPDIR \
    -jar $GATK_JAR/GenomeAnalysisTK.jar \
    -T TableRecalibration \
    -I $REALIGNED_OUT_BAM_FILE \
    -o 4.recal.out.bam \
    -R $REF \
    -recalFile $SHARED_RECAL_DIR/recal.table.merged.final.txt
...
# 2. 变异调用和过滤
samtools mpileup -Duf $REF -q 1 4.recal.out.bam | bcftools view -bvg ] - > \
$REALIGNED_OUT_BAM_FILE.raw.bcf
```

## 变异检测阶段的归约器

前面已经提到，对应所有映射器只有一个归约器（参见示例18-17）。这是因为，我们要合并所有值来创建一个输出：一个VCF文件。相应地，这个归约器只做一件事：创建一个VCF文件（参见示例18-18）。

示例18-17：变异检测阶段：reducer()函数

```
// key: <analysis_id>, 唯一地标识所有数据
// values: 忽略
reduce(key, values) {
   DNASeq.theVariantDetectionReducer(key);
   emit(key, key);
}
```

示例18-18：theVariantDetectionReducer()方法

```
public static void theVariantDetectionReducer(String analysisID)
   throws Exception {
   TemplateEngine.init();
   Map<String, String> templateMap = new HashMap<String, String>();
   templateMap.put("key", "-");
   templateMap.put("analysis_id", analysisID);
   // 从一个模板文件创建具体的脚本
    String scriptFileName = "/dnaseq/variant_detection_reducer_" +
                    templateMap.get("analysis_id") +".sh";
   String logFileName = "/dnaseq/variant_detection_reducer_" +
                    templateMap.get("analysis_id") +".log";
   File scriptFile = TemplateEngine.createDynamicContentAsFile(
        "variant_detection_reducer.template",
        templateMap,
        scriptFileName);
   if (scriptFile != null) {
```

```
17        ShellScriptUtil.callProcess(scriptFileName, logFileName);
18    }
19 }
```

示例18-19给出了*variant_detection_reducer.template*的部分代码。

示例18-19：变异检测归约器模板

```
1 #!/bin/bash
2 ...
3 # 调用snp(获得变异)
4 # 连接所有$KEY.raw.bcf文件
5 #
6 FINAL_BCF_FILE=$FINAL_DIR/all.raw.bcf
7 VCF_FILE=$FINAL_DIR/var.flt.vcf
8 ...
9 ##
10 ## 连接BCF文件。输入文件必须排序
11 ## 而且相同的样本要以同样的顺序出现。
12 ##
13 ALL_BCF_FILES='find $RECAL_DIR/ -name '*.raw.bcf' | sort'
14 $BCFTOOLS cat $ALL_BCF_FILES > $FINAL_BCF_FILE
15 #
16 # 启动bcftools & 创建最终的VCF文件
17 $BCFTOOLS view $FINAL_BCF_FILE | $VCFUTILS varFilter > $VCF_FILE
```

这一章提供了DNA测序的一个MapReduce解决方案，这是基因组分析生态系统中非常重要的一个任务。一般的，一个计算能力很强的计算机可以在70小时内完成DNA测序，不过如果采用MapReduce解决方案，使用一个包含100个节点的集群，完成这个工作的时间可以锐减到几分钟。下一章会提供一个可伸缩的Cox回归解决方案(也称为生存分析)。

# 第19章

# Cox回归

在医学统计中，生存分析（survival analysis）是要描述一个连续变量（如基因表达式）对生存时间的影响。Cox比例风险回归（Cox proportional hazards regression）是生存分析中使用的一个非常重要而且相当流行的算法。这个算法很简单，对生存分布不做任何假设，因此，变量的单元变化会带来相对风险。例如，一个特定基因的表达式中的一个单元变化会让相对风险成倍增加。Cox回归的一个简单例子是：根据含酒精饮料的饮用情况，男人和女人患脑癌的风险是否有所不同？以饮酒量(每天饮用多少盎司)和输入的性别作为协变量，性别和饮酒量对脑癌发作时间的影响会有一些变化，通过构造一个Cox回归模型，可以对这些假设进行测试。

Cox回归模型是一个统计技术，用来研究病人的生存时间与一些解释变量（如time和censor）之间的关系。Cox回归模型由统计学教授Sir David Cox最早开发。Cox回归的一个重要特征是它会估计相对风险而不是绝对风险，而且没有假设任何绝对风险知识。根据定义，实现比例风险模型的Cox回归设计可以用来分析某个事件发生前的时间或者事件之间的间隔时间。Cox回归使用一个或多个预测变量来预测一个状态或事件变量，这些预测变量称为协变量（covariates）。在生存分析例子中，时间是协变量，死亡是事件（也就是说，我们要根据死亡事件前的疾病诊断来分析时间）。

Cox回归在临床和医疗领域得到了广泛的使用。下面是一些示例应用：

- 使用连续变量(如基因表达式或白血球数)的生存分析（[29]）。
- 组合基因标签(gene signatures)来改进乳腺癌存活率的预测（[36]）。

关于Cox回归的更多详细内容，可以参见David Garson的书[9]和*http://bit.ly/cox_regression*。

R编程语言的Cox回归实现是一个足以信赖的行业级标准的实现（基于coxph()函数），我们的MapReduce解决方案中将使用这个实现。遗憾的是，Java中却没有很好的Cox回归实现。

这一章将提供Cox回归的一个两阶段MapReduce解决方案：

- 阶段1：为Cox回归准备适当的输入。在这里，我们将聚集、分组，并生成数据，准备调用R的coxph()函数。
- 阶段2：使用阶段1的输出，并应用R的coxph()函数，最后分析所生成的结果。

# Cox模型剖析

Cox回归是一个基于时间的相当复杂的统计函数。它的主要作用是为时间-事件数据构建一个预测模型。基本说来，Cox回归模型要生成一个生存函数，根据预测变量的给定值预测给定时间t某个感兴趣的事件的发生概率（例如，在肺癌分析中，预测变量可以是不同性别的人所抽的香烟数量）。Cox回归要根据观察对象和一个时间段内发生的事件进行估计。

Cox比例风险模型可以表述如下：

$$h_i(t) = h_0(t) \exp(\beta_1 x_{i1} + \beta_2 x_{i2} + \ldots + \beta_n x_{in})$$

在这里：

- $(x_{i1}, x_{i2}, \ldots, x_{in})$ 是预测变量，与时间无关。
- $(\beta_1, \beta_2, \ldots, \beta_n)$是一个回归参数向量，由Cox回归估计。
- $h_0(t)$是基准点风险，这是一个不确定函数，因此这是一个半参数化模型，也就是说，它依赖于时间，而不是依赖于协变量。
- $\exp(\beta X)$依赖于协变量而不是时间。

两个观察的风险比（hazard ratio）与时间$t$无关，定义了比例风险性质，如下：

$$\frac{h_i(t)}{h_j(t)} = \frac{h_0(t)e^{\theta_i}}{h_0(t)e^{\theta_j}} = \frac{e^{\theta_i}}{e^{\theta_j}}$$

可以在等式两边同除以$h_0(t)$。然后取对数，可以得到：

$$\ln\left(\frac{h_i(t)}{h_0(t)}\right) = (\beta_1 x_{i1} + \beta_2 x_{i2} + \ldots + \beta_n x_{in})$$

我们称 $\frac{h_i(t)}{h_0(t)}$ 为风险比。

## Cox回归基本术语

要在生存分析中使用Cox回归，首先要了解一些重要的定义。我们从Michael Walker（*http://bit.ly/surv_analysis*）借用了以下定义：

*事件（Event）*

这是一个依赖于具体应用的概念。例如，死亡、疾病复发或恢复都可以是事件。

*时间（Time）*

这个参数在Cox回归定义中扮演着一个特别重要的角色。事件是基于时间的，例如，自一个观察期（如手术）开始到某个事件或研究结束或者失去联系或退出研究之间的时间。

*受限观察（Censoring/censored observation）*

测试主体在观察时间内没有发生任何事件，则描述为是受限的（censored），这表示我们无法观察后续发生了什么。观察时间结束后受限主体可能有也可能没有事件。

*生存函数（Survivor function），S(t)*

主体生存时间大于时间t的概率。

# 使用R的Cox回归

R编程语言通过coxph {survival}提供了Cox回归模型的一个实现。coxph()函数实现了一个Cox比例风险回归模型[注1]。

R的Surv()函数创建一个生存对象作为响应变量。它接受两个参数：Surv(time, censor)，其中：

- time是一个事件时间向量。
- censor是一个指示值向量（指示是否观察到事件，或事件是否是受限的）。

下面的例子显示了time和censor的一些示例值：

```
> time=c(5.880903491,11.07186858,10.97330596,1.347022587,8.246406571,
         6.209445585,1.80698152,6.899383984,1.281314168,5.650924025,
         10.25051335,2.036960986,6.800821355,6.932238193,6.800821355,
         2.595482546,8.344969199,5.848049281,4.238193019,5.815195072,
         6.340862423,2.529774127,2.628336756,0.689938398,1.347022587,
         2.694045175,6.078028747,1.21560575,8.410677618,8.509240246)
```

注1：使用Andersen和Gill的计数过程公式结合时间依赖变量、时间依赖阶层（strata）、每个主体的多个事件以及其他扩展（资料来源：*http://bit.ly/coxph*）。

```
> censor=c(1,1,0,1,1,1,1,0,1,1,0,1,0,1,1,1,0,0,1,1,1,1,1,1,1,0,1,1,0,0)
```

可以如下使用coxph()函数：

```
> library(survival) # load the package
> coxph(Surv(time, censor) ~ logRatio)
Call:
coxph(formula = Surv(time, censor) ~ logRatio)

            coef exp(coef) se(coef)      z    p
logRatio -0.0247     0.976   0.0798 -0.31 0.76

Likelihood ratio test=0.1 on 1 df, p=0.758 n= 30, number of events= 21
```

## 表达式数据

一个特定基因的数据（也就是表达式值中的变化）表示为差异倍数（foldchange）。差异倍数作为一个度量，描述了一个初始值到一个最终值之间的数量变化：

```
> foldchange=c(20.3,-15.5,-8.04,4.85,5.5,2.16,1.94,-2.13,-52.5,-1.07,
              6.23,7.19,4.97,-39.8,-2.11,3.19,-1.24,1.24,2.73,-44.4,
              -35,-58.5,-1.79,1.74,-2.15,3.22,-1.7,-3.07,2.57,-1.41)
> convert=function(x){return(log2(max(x, -1/x)))}
> logRatio = sapply(foldchange, convert)
> logRatio
 [1] 4.3434078 -3.9541963 -3.0071955 2.2779847 2.4594316 1.1110313
 [7] 0.9560567 -1.0908534 -5.7142455 -0.0976108 2.6392322 2.8459918
[13] 2.3132459 -5.3146965 -1.0772430 1.6735564 -0.3103401 0.3103401
[19] 1.4489010 -5.4724878 -5.1292830 -5.8703647 -0.8399596 0.7990873
[25] -1.1043367 1.6870607 -0.7655347 -1.6182387 1.3617684 -0.4956952
```

对于coxph()函数，我们只对两个值感兴趣：

- coef = -0.0247
- pValue = 0.758

# Cox回归应用

下面来考虑Cox回归的一个示例应用。假设有一组病人。每个病人有一组基因集（bioset），每个bioset包含一组基因（例如，RNA基因表达式数据类型的基因）及其相关的基因值（差异倍数）。Bioset是单独分析得到的数据标签，包含试验样本比较形式的数据（对应转录组、表观遗传和拷贝数变异数据），以及基因型标签（对应GWAS（全基因组关联研究）和突变数据）。通常会把Bioset称为基因标签（gene signature）。

每个bioset可以有多达60000个基因。这个数随基因标签的类型不同而有所不同；例如，ioset可以有最多20000个基因，而基因表达式bioset最多可以有50,000个基因。这个问题可以描述如下：给定一组bioset、time和censor，找出所有bioset中包含的全部基因的coef

（系数）和pValue（概率值）。如果为一个包含100000个样本的集合完成Cox回归，可能必须分析100000 × 60000= 60亿个数据记录。这个问题很适合使用MapReduce来解决。

# Cox回归 POJO解决方案

为了帮助你更好地理解Cox回归，现在先给出一个非MapReduce解决方案。对于每个bioset数据类型（基因表达式、拷贝数变异、甲基化），基因数目是固定的，由geneIdList标识。这个算法的目标是创建类似下面的输入：

```
geneID_1 bioset1_value11 bioset2_value12, ..., biosetN_value1N
geneID_2 bioset1_value21 bioset2_value22, ..., biosetN_value2N
...
geneID_M bioset1_valueM1 bioset2_valueM2, ..., biosetN_valueMN
```

然后将这个输入传递到R的coxph()函数。例如，对于拷贝数变异（copynumber variation,cnv）类型，算法如示例19-1所示。

示例19-1：非MapReduce Cox回归算法

```
input: double[] time;
input: int[] censor;
input: List<Long> biosetIDs;
input: List<Long> cnvGeneIdList; // 预建的(key, value)数据库
output: List<Tuple3<String, Double, Double>>
// 作为 List<Tuple3<geneID, coef, p-value>>
//
// 迭代处理所有基因
for (geneID : cnvGeneIdList) {
    // 每个geneID有一个键-值库
    // 其中键是biosetID，值是基因值
    mapDB = getMapDB(geneID);
    int index = 0;
    double[] geneValues = new double[time.length];
    for (biosetID : biosetIDs) {
        double geneValue = mapDB.get(biosetID);
        geneValues[index++] = geneValue;
    }

    // 调用R的Cox回归coxph()函数
    double[] result = coxph(time, censor, geneValues);
    double coef = result[0];
    double pValue = result[1];
    emit(geneID, (coef, pValue));
}
```

例如，如果有5000个bioset，我们希望为这些bioset中包含的所有基因（最多40000个）计算coxph()。这需要40000个coxph()调用，每个调用涉及3个数组：time[5000]、censor[5000]和foldchange[5000]（这里foldchange表示基因值）。

运行这个算法会生成以下输出：

```
geneID_1 coef_1 pValue_1
geneID_2 coef_2 pValue_2
...
geneID_M coef_M pValue_M
```

我们已经知道，这个解决方案没有使用MapReduce。完成这个解决方案时，要为每个基因创建一个持久的（key, value）库：数据库名为geneID，key是biosetID，value是与geneID关联的值。这里使用了MapDB存储这些持久库[注2]。

如果仅仅有几百个bioset，这是一个很高效的解决方案。不过，如果有成千上万个bioset，与这个非MapReduce解决方案相比，MapReduce/Hadoop解决方案可以更好地扩展。

# MapReduce输入

一般的，一个病人有一组bioset，每个bioset可能有不同的类型（如拷贝数变异、甲基化或基因表达式）。重申一次，每个bioset有成千上万个记录，每个记录包含geneID、参考类型（{r1, r2, r3, r4}）和与这个geneID关联的一个值。bioset由一个biosetID标识。例如，biosetID 8800包含以下数据：

```
$ head 8800.csv
13972,r1;2.45
4082,r1;1.8
40583,r1;1.8
16422,r1;1.8
21602,r1;1.8
45735,r1;1.8
43936,r1;1.8
26446,r1;1.8
16030,r1;-3
828,r1;0
```

这里每一行/记录表示了geneID、参考类型和关联值（例如，对于第一行，13972是geneID，r1是一个正常参考，2.45是关联的变异倍数值）。由于每个geneID的值顺序在Cox回归中很重要，所以我们将由现有的键-值库生成另外一组数据，其中还将包含biosetID。下面是对应biosetID 8800所需的输入（将在MapReduce/Hadoop实现中使用）：

注2：MapDB提供了由磁盘存储或离堆内存支持的并发TreeMap和HashMap功能。这是一个快速、可伸缩而且易于使用的嵌入式Java数据库引擎。这个数据库引擎很小（只是一个160 KB的JAR文件），不过它包含了丰富的特性，如事务、高效使用空间的串行化、实例缓存及透明压缩/加密。它的性能也很突出，与原生嵌入式数据库引擎不相上下。MapDB由Jan Kotek开发。

```
$ head 8800_for_cox.csv
13972,r1;8800,2.45
4082,r1;8800,1.8
40583,r1;8800,1.8
16422,r1;8800,1.8
21602,r1;8800,1.8
45735,r1;8800,1.8
43936,r1;8800,1.8
26446,r1;8800,1.8
16030,r1;8800,-3
828,r1;8800,0
```

## 输入格式

每个bioset记录的格式如下：

```
<geneID><,><referenceID><;><biosetID><,><geneValue>
```

这里referenceID可以是r1 = normal、r2 = disease、r3 = paired或r4 =unknown。

需要说明，我们的数据中使用了两个不同的分隔符（;和,），它们在MapReduce的map()和reduce()函数中标记化（tokenizing）数据时对我们会很有帮助。

# 使用MapReduce的Cox回归

这一章前面曾经说过，我们要用一个两阶段MapReduce算法实现Cox回归。下面来回顾这两个阶段，并给出各个阶段中映射器和归约器的描述：

- 阶段1: 生成数据，准备调用coxph()函数。
  - map()：按<geneID><,><referenceID>(键)对数据分组。
  - reduce()：为每个归约器生成<geneID><,><referenceID>，后面是所有基因值，基因值要按正确的bioset顺序排列（必须保持这个顺序，否则coxph()调用将是无意义的）。
- 阶段2: 调用coxph()并分析生成的结果。
  - map()：对阶段1中reduce()函数生成的数据调用coxph()。
  - reduce()：无。

## Cox回归阶段1: map()

阶段1的map()函数如示例19-2所示，这是一个恒等映射器。它接受键-值对，并发出同样的键-值对。

示例19-2：Cox回归算法阶段1: map()函数

```
/**
 * 每个输入记录表示一个键-值对
 * 输入记录有以下格式：
 * <geneID><,><referenceID><;><biosetID><,><geneValue>
 =* 这个映射器发出以下(K,V)对：
 * K: <geneID><,><referenceID>
 * V: <biosetID><,><geneValue>
 */
map(key, value) {
   String[] tokens = StringUtil.split(value, ";");
   String reducerKey = tokens[0];   // <geneID><,><referenceID>
   String reducerValue = tokens[1]; // <biosetID><,><geneValue>
   emit(reducerKey, reducerValue);
}
```

## Cox回归阶段1: reduce()

每个归约器接受一个（key, values）对，其中key是一个（geneID, referenceID）对，values是一个（biosetID, geneValue）对列表。如示例19-3所示，归约器根据归约器类setup()方法接收的bioset的顺序对geneValue排序。例如，如果values = {(B1, G1),(B3, G3), (B4, G4), (B2, G2)}，接收到的bioset是（B1, B2, B3, B4），归约器生成的有序基因值则为（G1, G2, G3, G4）。

示例19-3：Cox回归算法阶段1: reduce()

```
/**
 * @param key是<geneID><referenceID>
 * @param values是一个{<biosetID><,><geneValue>}列表
 * 注释: biosetID (List<Long>)从MapReduce驱动器传出，
 * 将在归约器的setup()函数中保存（只在reduce()开始之前执行一次）
 */
reduce(key, values) {
    String[] doubleValues = new String[biosets.size()];
    int numberOfValues = 0;
    for (pair : values) {
        String[] tokens = StringUtils.split(pair, ",");
        String biosetID = tokens[0];
        String geneValue = tokens[1];

        int index = biosets.indexOf(biosetID);
        if (index == -1) {
            // biosetID未找到
            return;
        }

        //在位置"index"找到biosetID
        doublevalues[index] = geneValue;
        numberOfValues++;
    }

    // 这些值可以完成Cox回归分析
```

```
27      if (numberOfValues != biosets.size()) {
28          // 对应这些bioset没有足够的基因值,
29          // 无法完成Cox回归
30          return;
31      }
32
33      // numberOfValues == biosets.size()
34      StringBuilder builder = new StringBuilder();
35      builder.append(key.toString());
36      builder.append(",");
37      for (int i=0; i < doublevalues.length; i++) {
38          builder.append(doublevalues[i]);
39          // 查看是否需要增加逗号
40          if (i < (doublevalues.length -1)) {
41              builder.append(",");
42          }
43      }
44
45      // 准备归约器来提供输出
46      String reducerValue = builder.toString();
47      // reducerValue = <geneID><,><referenceID>, value1, value2, ..., valueN
48      context.write(null, reducerValue);
49  }
```

## Cox回归阶段2: map()

这个阶段的映射器接受一个HDFS输入文件，然后利用这个输入执行coxph()，再将输出传递到HDFS来完成最终分析。对于这个映射器，我们需要一个setup()函数，这个函数只执行一次。（setup()和map()分别参见示例19-4和示例19-5）。

示例19-4：Cox回归算法阶段2: setup()

```
 1 /**
 2  * map(key) = LongWritable（由Hadoop生成，在这里忽略）
 3  * map(value) = 文本 (HDFS文件名: /biomarker/output/rnae/0/part-r-00000)
 4  * 可以用作为Cox回归分析的输入
 5  *
 6  * reduce(key) = none
 7  * reduce(value) = none
 8  */
 9 public class CoxRegressionMapperPhase2
10    extends Mapper<LongWritable, Text, LongWritable, Text> {
11
12    private Configuration conf = null;
13    private FileSystem fs = null;
14    private String timeAsCommaSeparatedString = null;
15    private String censorAsCommaSeparatedString = null;
16    private String hadoopOutputPathAsString = null;
17
18    ...
19
20    // 只运行一次
21    public void setup(Context context)
```

```
22        throws IOException, InterruptedException {
23        this.conf = context.getConfiguration();
24        this.fs = FileSystem.get(conf);
25        hadoopOutputPathAsString = conf.get("hadoopOutputPathAsString");
26        timeAsCommaSeparatedString = conf.get("time");
27        censorAsCommaSeparatedString = conf.get("censor");
28
29        //
30        // 确保本地UNIX文件系统中存在/hadoop/home/cox_regression.r.template文件；
31        // 如果不存在，则从
32        // hdfs:/biomarker/template/cox_regression.r.template复制
33        //
34        if (IOUtil.fileExists(
35            CoxRegressionUsingR.COX_REGRESSION_TEMPLATE_FILE_FULL_NAME)) {
36           THE_LOGGER.info("template file does exist: "+
37           CoxRegressionUsingR.COX_REGRESSION_TEMPLATE_FILE_FULL_NAME);
38        }
39        else {
40           // 从HDFS复制
41           copyHDFSFileToLocal(
41                 CoxRegressionUsingR.COX_REGRESSION_TEMPLATE_FILE_HDFS_PATH,
42                     CoxRegressionUsingR.COX_REGRESSION_TEMPLATE_FILE_FULL_NAME);
43        }
44     }
45
46     // map()见下面的定义
```

示例19-5：Cox回归算法阶段2: map()

```
 1 /**
 2  * @param key是MapReduce框架生成的键（这里未使用）
 3  * @param value为阶段1中reduce()生成的HDFS输入文件名
 4  * (例如：HDFS文件</biomarker/output/rnae/0><,><part-r-00019>)
 5  * 注意：以下变量
 6  * timeAsCommaSeparatedString,
 7  * censorAsCommaSeparatedString和
 8  * hadoopOutputPathAsString
 9  * 在归约器的setup()函数中初始化（只执行一次）。
10  */
11 map(key, value) {
12    String coxRegressionInputFileName = "/tmp/" + HadoopUtil.getRandomUUID();
13    // coxRegressionInputFileName = /tmp/a99817a0-c149-4cb2-a771-dbc7da86b56a
14    String coxRegressionOutputFileName = coxRegressionInputFileName + ".out.txt";
15    File coxRegressionInputFile = new File(coxRegressionInputFileName);
16
17    Path hdfsPartFile = new Path(valueAsString);
18    FileUtil.copy(fs,                    // 文件系统
19                  hdfsPartFile,          // 源文件
20                  coxRegressionInputFile, // 目标
21                  false,                 // 布尔值deleteSource
22                  conf);
23
24    int coxCallStatus = -1; // 失败
25    // 完成Cox回归
26    coxCallStatus = CoxRegressionUsingR.callCoxRegression(
```

```
27          coxRegressionInputFileName,
28          coxRegressionOutputFileName,
29          timeAsCommaSeparatedString,
30          censorAsCommaSeparatedString);
31
32    if (coxCallStatus == 0) {
33       // 大功告成! 将coxRegressionOutputFileName
34       // 复制到HDFS(hadoopOutputPathAsString)
35       File coxRegressionOutputFile = new File(coxRegressionOutputFileName);
36       if (coxRegressionOutputFile.exists()) {
37          FileUtil.copy(coxRegressionOutputFile, fs,
38                      new Path(hadoopOutputPathAsString), false, conf);
39       }
40    }
41
42    // 发送到归约器 (用作指示以方便调试)
43    context.write(key, value);
44 }
```

## 阶段1中 reduce()函数生成的示例输出

下面是阶段1中reduce()函数生成的示例输出文件：

```
$ hadoop fs -ls /biomarker/output/rnae/0/part*
Found 30 items
-rw-r--r--   3 ... 1047918 ... /biomarker/output/rnae/0/part-r-00000
-rw-r--r--   3 ... 1053398 ... /biomarker/output/rnae/0/part-r-00001
...
-rw-r--r--   3 ... 1030567 ... /biomarker/output/rnae/0/part-r-00026
-rw-r--r--   3 ... 1028483 ... /biomarker/output/rnae/0/part-r-00027
```

如果我们的测试示例有5000个bioset，则可以得到：

```
$ hadoop fs -cat /biomarker/output/rnae/0/part-r-00005 | head
100174,r1,1.8619553641448698,1.8925853151926535, ..., <5000th-value>
100181,r1,0.40490312214513074,2.67581593117227, ..., <5000th-value>
100191,r2,-13.287712379549449,13.28771237954945, ..., <5000th-value>
100237,r1,0.8147554828098739,-0.3391373849195852, ..., <5000th-value>
100314,r1,2.103665482765696,2.254594043033141, ..., <5000th-value>
100478,r2,0.0,0.0, ..., <5000th-value>
100482,r1,1.5464623581670915,-2.5864044748950628, ..., <5000th-value>
100545,r1,-2.454965473634712,0.41359408240917517, ..., <5000th-value>
100555,r2,-13.287712379549449,-0.15055967657538144, ..., <5000th-value>
```

## 阶段2中 map()函数生成的示例输出

在下面的示例输出中，第一列是<geneID>,<referenceID>，第二列是coef，最后一列是pValue：

```
94893,r2  -0.04106195  0.7144141
94963,r2   0.2514287   0.02973554
```

```
95037,r1  -0.1822651   0.326605
95047,r2  -0.02349459  0.8174117
95338,r1   0.06733862  0.4429394
95425,r2  -0.1370884   0.3265356
95719,r2  -0.02158204  0.9071648
95891,r1   0.1033474   0.6061742
```

## MapReduce的Cox回归脚本

CoxRegressionUsingR类包含了Cox回归的核心功能。这个类由模板文件创建一个Rscript。示例19-6给出了Rscript的一个实例。

示例19-6：Cox回归算法（Rscript）

```
#!/usr/local/bin/Rscript

input_file = "/tmp/f11d10f4-e0a6-4796-9a1c-34c6914be1e7"
cat("input_file=", input_file, "\n")
output_file = "/tmp/f11d10f4-e0a6-4796-9a1c-34c6914be1e7.out.txt"
cat("output_file=", output_file, "\n")

library("survival")

# 转换函数
convert=function(x){return(log2(max(x, -1/x)))}

# 完成Cox回归的函数
cox_regression <- function(line){
      #cat(line, "\n")
      # 每一行有以下格式:
      # <geneID><;><V1,V2,...,Vn><;><T1,T2,...,Tn><;><C1,C2,...,Cn>
      items = unlist(strsplit(line, ";"))
      #
      geneID = items[[1]]
      #
      value = as.double(unlist(strsplit(items[[2]], ",")))
      #
      time = as.double(unlist(strsplit(items[[3]], ",")))
      #
      censor = as.double(unlist(strsplit(items[[4]], ",")))
      #
      coxphoutput = coxph(Surv(time,censor) ~ value)
      pValue = summary(coxphoutput)$waldtest[3]
      cat(geneID, coxphoutput$coef, pValue, "\n", file=output_file, append=TRUE)
}

# Rscript的驱动器部分
conn <- file(input_file, open="r")
while(length(line <- readLines(conn, 1)) > 0) {
   try.output <- try( cox_regression(line) )
}
close.connection(conn)
```

下面是这个代码的详细说明：

*第14～31行*

这里我们定义了定制函数cox_regression()，它会调用R的coxph()函数。第18行对输入记录标记化。最后coxph()生成coef和pValue。

*第30行*

这一行将geneID、coef和pValue追加到输出文件。

*第34～38行*

这是Rscript的驱动器。它会打开文件，逐行读取，并应用cox_regression()函数。最后会关闭输入文件来释放资源。

下面是示例输出：

```
$ head -4 /tmp/f11d10f4-e0a6-4796-9a1c-34c6914be1e7.out.txt
100007,r1,0.1038095,0.3594686
10017,r2,0.00613293,0.892313
100205,r1,-0.01681699,0.6164844
101583,r1,0.03383865,0.4736812
```

这一章使用MapReduce/Hadoop提供了一个分布式可伸缩的Cox回归算法。Cox回归是临床应用中完成生存分析的一个非常重要的技术。下一章会为Cochran-Armitage趋势检验实现一个MapReduce解决方案，可以用来完成VCF(变体调用格式)文件的分析。

# 第20章

# Cochran-Armitage趋势检验

Cochran-Armitage趋势检验（Cochran-Armitage test for trend，CATT）用于分析生殖细胞数据。例如，DNA测序生成的VCF（变体调用格式）文件中的变异可以标记为生殖细胞数据。CATT是一种对窄候选（narrow alternatives）完成卡方检验的统计方法。如果R是一个响应变量集，$E$是一个实验变量集，CATT对$R(s)$与$E(s)$之间的线性关系很敏感，可以检测其趋势。可以用另一种方式来描述CATT：如果B是某个事件的二进制结果{PASSED, FAILED}，$C$是一个有序的分类集合$\{C_1, ..., C_n\}$，那么CATT可以用作为$C$的各个层次上$B$比例的线性趋势。要应用CATT，需要构建一个列联表（contingency table）：包含对应结果值{PASSED, FAILED}的两行，另外有$n$列$\{C_1, ..., C_n\}$。CATT的列联表将在后面的小节中详细解释。

*Wikipedia对Cochran-Armitage趋势检验的描述如下（http://bit.ly/cochran-armitage）：*

> Cochran-Armitage趋势检验（The Cochran-Armitage test for trend）以William Cochran和Peter Armitage命名，在分类数据分析中，如果目标是得到一个的变量(包含两个分类)与另一个变量（包含k个分类）之间的关联关系，就可以使用这种检验方法。它修改了泊松卡方检验，对第二个变量k个分类的效果增加了一个猜想顺序。例如，药物用量可以排序为“low”、“medium”和“high”，可以猜想随着用药量的增加，治疗效果不会减弱。趋势检验通常用作为一种基于基因型的检验来完成病例-对照遗传关联研究。

*Stefan Wellek和Andreas Ziegler关于Cochran-Armitage趋势检验的描述如下[30]：*

> 基于线性回归模型的Cochran-Armitage趋势检验已经成为病例-对照研究中完成关联检验的标准过程。

# Cochran-Armitage算法

数据采用2 × $k$列联表形式时，可以应用CATT。行数(2)指示实验的结果，列数指示一个可变的实验数（$k$）。例如，如果$k = 3$，列联表则如表20-1所示。

**表20-1：2×3列联表**

| 组 | B = 1 | B = 2 | B = 3 |
|---|---|---|---|
| $A = 1$ | $N_{11}$ | $N_{12}$ | $N_{13}$ |
| $A = 2$ | $N_{21}$ | $N_{22}$ | $N_{23}$ |

这个列联表可以再加上两个变量的边缘总和，如表20-2所示。

**表20-2：包含边缘总和的2 × 3列联表**

| 组 | B = 1 | B = 2 | B = 3 | 总和 |
|---|---|---|---|---|
| $A = 1$ | $N_{11}$ | $N_{12}$ | $N_{13}$ | $R_1$ |
| $A = 2$ | $N_{21}$ | $N_{22}$ | $N_{23}$ | $R_2$ |
| 总和 | $C_1$ | $C_2$ | $C_3$ | $N$ |

在这里：

- $R_1 = N_{11} + N_{12} + N_{13}$
- $R_2 = N_{21} + N_{22} + N_{23}$
- $C_1 = N_{11} + N_{21}$
- $C_2 = N12 + N_{22}$
- $C_3 = N_{13} + N_{23}$
- $N = R_1 + R_2 = C_1 + C_2 + C_3 = N_{11} + N_{12} + N_{13} + N_{21} + N_{22} + N_{23}$

趋势检验统计量为：

$$T \equiv \sum_{i=1}^{k} w_i (N_{1i} R_2 - N_{2i} R_1)$$

这里$w_i$为权重。对生殖细胞数据的等位基因使用CATT时，可以根据权重值应用3种不同的检验：

- 权重 = {0, 1, 2}: 对应加性模型。
- 权重 = {1, 1, 0}: 对应显性模型。
- 权重 = {0, 1, 1}: 对应隐性模型。

无关联假设（也称为零假设或虚无假设（null hypothesis））可以表述为：

$$\Pr(A = 1|B = 1) = \ldots = \Pr(A = 1|B = k)$$

假设满足零假设，使用迭代期望可以写为：

$$E(T) = E(E(T|R_1, R_2)) = E(0) = 0$$

给定两个离散的随机变量$X$和$Y$，可以定义条件期望为：

$$E[X|Y = y] = \sum_x x \cdot P(X = x|Y = y)$$

现在使用所有这些定义和公式，用Java编写我们的Cochran-Armitage算法（参见示例20-1）。CATT的一个主要目标是计算p值（p-value，这是0.00～1.00之间的一个概率值）。使用Wikipedia（*http://bit.ly/cochran-armitage*）中定义的算法，可以把CATT实现为一个POJO类`CochranArmitage`。我们的MapReduce解决方案中将使用这个Java类。

示例20-1：Cochran-Armitage算法

```
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;

import org.apache.log4j.Logger;
import org.apache.commons.math3.distribution.NormalDistribution;

/**
 * 这个类可以在一个2x3列联表上计算Cochran-Armitage趋势检验。
 * 用来估计基因型数据加性
 * 遗传模型中的关联。
 */
public class CochranArmitage {

   private static final Logger THE_LOGGER =
      Logger.getLogger(CochranArmitage.class);

   // 使用对应加性/共显性模型的权重
   private static final int[] WEIGHTS = { 0, 1, 2 };

   // 传入的列联表的维度 ， 必须是2行x 3列
   private static final int NUMBER_OF_ROWS = 2;
   private static final int NUMBER_OF_COLUMNS = 3;

   // 这些变量保存方差、原始统计量、标准化统计量和
   // p值
   private double stat = 0.0;
   private double standardStatistics = 0.0;
   private double variance = 0.0;
   private double pValue = -1.0; // 范围是0.0～1.0（-1.0表示未定义）
```

```
33
34    // Apache的NormalDistribution类，用来计算p值
35    private static NormalDistribution normDist = new NormalDistribution();
36
37    /**
38     * 得到方差variance
39     */
40    public double getVariance() {
41        return variance;
42    }
43
44    /**
45     * 得到原始统计量Stat
46     */
47    public double getStat() {
48        return stat;
49    }
50
51    /**
52     * 得到标准化统计量StandardStatistics
53     */
54    public double getStandardStatistics() {
55        return standardStatistics;
56    }
57
58    /**
59     * 得到p值（p-value）
60     */
61    public double getpValue() {
62        return pValue;
63    }
64
65    /**
66     * 为传入的2行3列的列联表
67     * 计算Cochran-Armitage趋势检验
68     * @param countTable= 2x3列联表。
69     * @return 所传入列联表Cochran-Armitage统计量的p值
70     */
71    public double callCochranArmitageTest(int[][]countTable) {
72       // 在示例20-2中定义。
73    }
74
75
76    /**
77     * @param args用于测试/调试的输入/输出文件
78     * args[0] 作为输入文件
79     * args[1] 作为输出文件
80     */
81    public static void main(String[]args) throws IOException {
82       // 在示例20-3中定义。
83    }
84
85 }
```

callCochranArmitageTest()方法在示例20-2中定义，这是Cochran-Armitage算法的核心。

示例20-2：Cochran-Armitage算法：callCochranArmitageTest()

```
/**
 *  为传入的2行3列的列联表
 * 计算Cochran-Armitage趋势检验
 *  @param countTable= 2x3列联表。
 * @return 所传入列联表Cochran-Armitage统计量的p值
 */
public double callCochranArmitageTest(int[][]countTable) {

   if (countTable== null) {
      throw new IllegalArgumentException(
          "contingency table cannot be null/empty.");
   }

   if ( (countTable.length != NUMBER_OF_ROWS) ||
        (countTable[0].length != NUMBER_OF_COLUMNS) ) {
      throw new IllegalArgumentException(
          "contingency table must be 2 rows by 3 columns");
   }

   int totalSum=0;
   int[]rowSum = new int[NUMBER_OF_ROWS];
   int[]colSum = new int[NUMBER_OF_COLUMNS];

   // 计算列联表的边缘总和和总计
   for (int i=0; i<NUMBER_OF_ROWS; i++) {
      for (int j=0; j<NUMBER_OF_COLUMNS; j++) {
         rowSum[i] += countTable[i][j];
         colSum[j] += countTable[i][j];
         totalSum += countTable[i][j];
      }
   }

   // 根据http://en.wikipedia.org/wiki/Cochran-Armitage_test_for_trend的公式
   // 计算检验统计量和方差
   stat = 0.0;
   variance = 0.0;
   for (int j=0; j<NUMBER_OF_COLUMNS; j++) {
      stat += WEIGHTS[j] * (countTable[0][j]*rowSum[1] -
                  countTable[1][j]*rowSum[0]);
      variance += WEIGHTS[j]*WEIGHTS[j]*colSum[j]*(totalSum-colSum[j]);

      if (j!=NUMBER_OF_COLUMNS-1) {
         for (int k=j+1;k<NUMBER_OF_COLUMNS;k++) {
            variance -= 2*WEIGHTS[j]*WEIGHTS[k]*colSum[j]*colSum[k];
         }
      }
   }
   variance *= rowSum[0]*rowSum[1]/totalSum;

   // 标准化统计量为stat除以SD
   standardStatistics = stat/Math.sqrt(variance);

```

```
53    // 使用Apache Commons正态分布来计算双尾检验p值
54    pValue = 2*normDist.cumulativeProbability(-Math.abs(standardStatistics));
55
56    // 返回p值
57    return pValue;
58 }
```

可以用示例20-3中所示的程序测试这个Cochran-Armitage算法。

示例20-3：Cochran-Armitage算法：main()

```
 1 /**
 2  * @param args用于测试/调试的输入/输出文件
 3  * args[0] 作为输入文件
 4  * args[1] 作为输出文件
 5  */
 6 public static void main(String[]args) throws IOException {
 7    if (args.length != 2) {
 8       THE_LOGGER.info("usage: java CochranArmitage " +
 9              "<input-filename> <output-filename>");
10       throw new IOException("must provide input and output files for testing.");
11    }
12
13    long startTime = System.currentTimeMillis();
14    String inputFileName = args[0];
15    String outputFileName = args[1];
16    BufferedWriter outfile = new BufferedWriter(new FileWriter(outputFileName));
17    outfile.write("score\tp-value\n");
18    BufferedReader infile = new BufferedReader(new FileReader(inputFileName));
19
20    int[][] countTable= new int[2][3];
21    String line = null;
22    while (( line = infile.readLine()) != null) {
23       String[] tokens = line.split("\t");
24       int index=0;
25
26       // 填充2x3列联表
27       for(int i=0; i<2; i++) {
28          for(int j=0; j<3; j++) {
29             countTable[i][j] = Integer.parseInt(tokens[index++]);
30          }
31       }
32
33       CochranArmitage catest = new CochranArmitage();
34       double pValue = catest.callCochranArmitageTest(countTable);
35       outfile.write(String.format("%f\t%f\n",
36         catest.getStandardStatistics(), pValue));
37    }
38
39    long elapsedTime = System.currentTimeMillis() - startTime;
40    THE_LOGGER.info("run time (in milliseconds): " + elapsedTime);
41
42    infile.close();
43    outfile.close();
44 }
```

下面是Cochran-Armitage算法的一个运行示例。首先来编译这个算法，并定义输入：

```
$ javac CochranArmitage.java
$ cat test3.txt
1386  1565  401  1342  1579  434
2716  672   13   2689   695    9
2062  1144  151  2021  1184  173
```

然后运行这个算法：

```
$java CochranArmitage test3.txt test3.txt.out
[main] [INFO ] [CochranArmitage] - run time (in milliseconds): 9
```

期望输出如下所示：

```
$ cat test3.txt.out
score       p-value
-1.414843   0.157114
-0.488857   0.624943
-1.555344   0.119864
```

# Cochran-Armitage应用

在基因组分析中，CATT可以用来完成基因型频率（genotype frequency）差异的统计检验，其中每个个体要根据这个个体中特定变异等位基因的个数编码为{0, 1, 2}。这些计数可以用来建立一个包含2行3列的列联表（每一行表示一个特定的组，每一列表示一次实验的结果，如一个等位基因计数），可以用标准统计方法分析这个列联表。CATT用于估计一个加性遗传模型。

下面来描述基因组分析中的一个实际例子：令组A表示一组病人的一个bioset集（由VCF文件生成），另外组B表示另一组病人的另一个bioset集（也由VCF文件生成）。这里的目标是对特定染色体[注1]（指定了开始位置和结束位置）上找到的一个等位基因集应用CATT（关于染色体的详细信息，可以访问National Institute of Health网站（*http://bit.ly/chromosome_nih*））。这个数据可能相当庞大。从一个（人类样本）VCF文件生成的bioset可能包含超过4000000个染色体。如果组A有3000个样本，组B有5000个样本，要比较基因型频率，我们必须分析320亿个记录（很显然，这是一个大数据问题！）：

组A记录= 3000 × 4000000 = 12000000000
组B记录= 5000 × 4000000 = 20000000000
总记录= 12000000000 + 20000000000 = 32000000000

注1：正常人每个细胞有46个染色体，分为23对。染色体1的两个副本（分别从双亲继承得来）构成了其中一个染色体对（资料来源：*http://ghr.nlm.nih.gov/chromosome/1*）。

要使用CATT得到基因型频率，需要为特定染色体（由其开始位置和结束位置标识）公共键上找到的各个等位基因分别建立一个2 × 3列联表。如表20-3所示。

**表20-3 每个等位基因的列联表**

| 组 | 0的计数 | 1的计数 | 2的计数 |
|---|---|---|---|
| 组A | $N_{11}$ | $N_{12}$ | $N_{13}$ |
| 组B | $N_{21}$ | $N_{22}$ | $N_{23}$ |

我会通过下面的例子来展示如何为各个等位基因分别建立一个2 × 3的列联表。令组A是一个包含6个bioset的集合，标识为$\{B_1, B_2, B_3, B_4, B_5, B_6\}$（见表20-4），另外令组B是一个包含5个bioset的集合，标识为$\{B_7, B_8, B_9, B_{10}, B_{11}\}$ (见表20-5)。注意这个数据对应一个特定的染色体（有指定的开始和结束位置）。

**表20-4：组A的bioset**

| Bioset ID | 等位基因1 | 等位基因2 |
|---|---|---|
| $B_1$ | A | C |
| $B_2$ | A | A |
| $B_3$ | A | C |
| $B_4$ | G | G |
| $B_5$ | A | A |
| $B_6$ | AC | T |

**表20-5：组B的bioset**

| Bioset ID | 等位基因1 | 等位基因2 |
|---|---|---|
| $B_7$ | A | A |
| $B_8$ | C | C |
| $B_9$ | A | C |
| $B_{10}$ | A | A |
| $B_{11}$ | A | A |

生成/构建列联表之前，需要先建立一些数据结构，如表20-6所述。

**表20-6：基因型频率**

| Bioset ID | 组 | A计数 | C计数 | G计数 | T计数 | AC计数 |
|---|---|---|---|---|---|---|
| $B_1$ | 组A | 1 | 1 | 0 | 0 | 0 |
| $B_2$ | 组A | 2 | 0 | 0 | 0 | 0 |
| $B_3$ | 组A | 1 | 1 | 0 | 0 | 0 |
| $B_4$ | 组A | 0 | 0 | 2 | 0 | 0 |

表20-6：基因型频率（续）

| Bioset ID | 组 | A计数 | C计数 | G计数 | T计数 | AC计数 |
|---|---|---|---|---|---|---|
| $B_5$ | 组A | 2 | 0 | 0 | 0 | 0 |
| $B_6$ | 组A | 0 | 0 | 0 | 1 | 1 |
| $B_7$ | 组B | 2 | 0 | 0 | 0 | 0 |
| $B_8$ | 组B | 0 | 2 | 0 | 0 | 0 |
| $B_9$ | 组B | 1 | 1 | 0 | 0 | 0 |
| $B_{10}$ | 组B | 2 | 0 | 0 | 0 | 0 |
| $B_{11}$ | 组B | 2 | 0 | 0 | 0 | 0 |

现在可以为各个等位基因分别生成一个列联表（A，C，G，T和AC。见表20-7～表20-11），然后应用CATT算法。在我们的MapReduce算法中，（$Key_2$，$Value_2$）的各个归约器会生成一组列联表（其中$Key_2$是一个组合键chromosomeID:start:stop）。

表20-7：等位基因A的列联表

| 组 | 0的计数 | 1的计数 | 2的计数 |
|---|---|---|---|
| 组A | 2 | 2 | 2 |
| 组B | 1 | 1 | 3 |

表20-8：等位基因C的列联表

| 组 | 0的计数 | 1的计数 | 2的计数 |
|---|---|---|---|
| 组A | 4 | 2 | 0 |
| 组B | 3 | 1 | 1 |

表20-9：等位基因G的列联表

| 组 | 0的计数 | 1的计数 | 2的计数 |
|---|---|---|---|
| 组A | 5 | 0 | 1 |
| 组B | 5 | 0 | 0 |

表20-10：等位基因T的列联表

| 组 | 0的计数 | 1的计数 | 2的计数 |
|---|---|---|---|
| 组A | 5 | 1 | 0 |
| 组B | 5 | 0 | 0 |

表20-11：等位基因AC的列联表

| 组 | 0的计数 | 1的计数 | 2的计数 |
|---|---|---|---|
| 组A | 5 | 1 | 0 |
| 组B | 5 | 0 | 0 |

# MapReduce解决方案

这一节将提供CATT的一个MapReduce算法，可以用Hadoop和Spark实现。这里的实现使用了MapReduce/Hadoop。

## 输入

由于可能会为两组bioset（A和B）选择相同的bioset，我们将生成两个数据类型（它们唯一的区别在于GROUP-NAME，对于组A，GROUP-NAME为a；对于组B，GROUP-NAME为b，这样就能区别这两个组）。

每个bioset记录有以下格式：

```
<chromosome-ID>
<:>
<chromosome-start-position>
<:>
<chromosome-stop-position>
<;>
<GROUP-NAME>
<:>
<allele1>
<:>
<allele2>
<:>
<reference>
<:>
<snp-id>
<:>
<mutation-class-ID>
<:>
<gene-id>
<:>
<bioset_Id>
```

例如，如果为组A选择6个bioset，则有：

```
7:10005296:10005296;a:A:C:A:snpid:mc:geneid:1000
7:10005296:10005296;a:A:A:A:snpid:mc:geneid:2000
7:10005296:10005296;a:A:C:C:snpid:mc:geneid:3000
7:10005296:10005296;a:G:G:G:snpid:mc:geneid:4000
7:10005296:10005296;a:A:A:A:snpid:mc:geneid:5000
```

```
7:10005296:10005296;a:AC:T:A:snpid:mc:geneid:6000
```

如果为组B选择5个bioset，则有：

```
7:10005296:10005296;b:A:A:A:snpid:mc:geneid:7000
7:10005296:10005296;b:C:C:C:snpid:mc:geneid:7100
7:10005296:10005296;b:A:C:C:snpid:mc:geneid:7200
7:10005296:10005296;b:A:A:A:snpid:mc:geneid:7300
7:10005296:10005296;b:A:A:A:snpid:mc:geneid:7400
```

## 期望输出

每个结果记录（Cochran-Armitage检验生成的p值）有以下格式：

```
<pValue>
<:>
<chromosome-ID>
<:>
<chromosome-start-position>
<:>
<chromosome-stop-position>
<:>
<gene-id>
<:>
<mutation-class-ID>
<:>
<reference>
<:>
<alternate-allele>
<:>
<N11>
<:>
<N12>
<:>
<N13>
<:>
<N21>
<:>
<N22>
<:>
<N23>
<:>
<snp-id>
```

## 映射器

映射器（参见示例20-4）将为各个记录生成一个键-值对，其中，键为：

```
<chromosome-ID><:><chromosome-start-position><:><chromosome-stop-position>
```

值为其余属性：

```
<GROUP-NAME>
<:>
<allele1>
<:>
<allele2>
<:>
<reference>
<:>
<snp-id>
<:>
<mutation-class-ID>
<:>
<gene-id>
<:>
<bioset_Id>
```

示例20-4：Cochran-Armitage: map()函数

```
/**
 * @param key为Hadoop生成的键，在这里忽略
 * @param value有以下格式：
 * <chr-ID><:><start><:><stop><;><GROUP><|><allele1><|><allele2><|>
 * <ref><|><snp-id><|><mutation-class><|><gene-id>|<bioset_Id>
 * 这里GROUP属于{"a" , "b"}
 *
 * reducerKey = Text: <chr-ID><:><start><:><stop>
 * reducerValue = Text: <GROUP><|><allele1><|><allele2><|><ref><|>
 * <snp-id><|><mutation-class><|><gene-id>|<bioset_Id>
 */
map(Object key, Text value) {
   String[] tokens = StringUtils.split(line, ";");
   if (tokens.length == 2) {
      String reducerKey = tokens[0];
      String reducerValue = tokens[1];
      emit(reducerKey, reducerValue);
   }
}
```

## 归约器

CATT的主要任务由归约器完成。每个归约器的键就是映射器生成的键。各个归约器的值是映射器生成的一个值列表。因此，键为7:10005296:10005296的归约器将有以下值：

```
a:A:C:A:snpid:mc:geneid:1000
a:A:A:A:snpid:mc:geneid:2000
a:A:C:C:snpid:mc:geneid:3000
a:G:G:G:snpid:mc:geneid:4000
a:A:A:A:snpid:mc:geneid:5000
a:AC:T:A:snpid:mc:geneid:6000
b:A:A:A:snpid:mc:geneid:7000
b:C:C:C:snpid:mc:geneid:7100
b:A:C:C:snpid:mc:geneid:7200
b:A:A:A:snpid:mc:geneid:7300
```

```
b:A:A:A:snpid:mc:geneid:7400
```

要实现reduce()（参见示例20-5），我们需要3个数据结构：

```
// 组A和组B中的所有等位基因
Map<String, String[]> allAlleles = new HashMap<String, String[]>();

//组A中的所有等位基因
List<String[]> groupA = new ArrayList<String[]>();

//组B中的所有等位基因
List<String[]> groupB = new ArrayList<String[]>();
```

填充这3个数据结构之后，可以生成列联表，然后应用Cochran-Armitage趋势检验。

示例20-5：Cochran-Armitage: reduce()函数

```
/**
 * @param key为键: Text: <chr-ID><:><start><:><stop>
 * @param values是一个{value}列表，其中各个值有以下格式：
 * <GROUP><:><allele1><:><allele2><:><ref><:><snp-id><:>
 * <mutation-class><:><gene-id>:<bioset_Id>
 * 这里GROUP属于{"a" , "b"}
 *
 * outputKey = null
 * outputValue = 由buildOutputValue()方法建立
 */
reduce(Text key, Iterable<Text> values) {
   // tokens[index] 0 1 2 3 4
   // reduce(value) = Text: <GROUP><:><allele1><:><allele2><:><ref><:><snp-id><:>
   // tokens[index] 5 6 7
   // reduce(value) = Text: <mutation-class><:><gene-id>:<bioset_Id>

   Map<String, String[]> allAlleles = new HashMap<String, String[]>();
   List<String[]> groupA = new ArrayList<String[]>();
   List<String[]> groupB = new ArrayList<String[]>();

   for (Text valueAsText : values) {
      String value = valueAsText.toString();
      String[] tokens = StringUtils.split(value, "|");
      if (tokens.length != 8) {
         continue;
      }

      String group = tokens[0];
      String allele1 = tokens[1];
      String allele2 = tokens[2];
      if (!allAlleles.containsKey(allele1)) {
         allAlleles.put(allele1, tokens);
      }
      if (!allAlleles.containsKey(allele2)) {
         allAlleles.put(allele2, tokens);
      }

      if (group.equals("a")) {
```

```
39          // 可能是组"a"
40          groupA.add(tokens);
41        }
42        else {
43          // 或者是组"b"
44          groupB.add(tokens);
45        }
46      } // for-loop结束
47
48    if ( (groupA.isEmpty()) & (groupB.isEmpty()) ) {
49      return;
50    }
51
52    // 迭代处理allAlleles，完成分析
53    int[][] contingencyTable= new int[2][3];    // 创建一个列联表
54    for (Map.Entry<String, String[]> entry : allAlleles.entrySet()) {
55      String allele = entry.getKey();          // 键
56      String[]tokens = entry.getValue();       // 值
57
58      // -------------------------------------------------
59      // 创建以下表:
60      //
61      //            0     1     2   <-- 计数
62      //         -------------------
63      // groupA    n10   n11   n12   (索引0为groupA)
64      // groupB    n20   n21   n22   (索引1为groupB)
65      //
66      // 然后将这个2x3数组传递到CochranArmitage检验
67      // -------------------------------------------------
68
69      clearContingencyTable(contingencyTable);
70      fillContingencyTable(groupA, groupB, contingencyTable);
71      double pValue = CochranArmitage.callTrendTest(contingencyTable);
72      String outputValue = buildOutputValue(pValue, key, allele, tokens,
73                                            contingencyTable);
74      // 准备归约器提供输出
75      emit(null, outputValue);
76    }
77 }
```

示例20-6定义了Cochran-Armitage趋势检验的辅助函数。

示例20-6：Cochran-Armitage: 辅助函数

```
 1 private static void clearContingencyTable(int[][] contingencyTable) {
 2    for (int i=0; i < 2; i++) {
 3       for (int j=0; j < 3; j++) {
 4         contingencyTable[i][j] = 0;
 5       }
 6    }
 7 }
 8
 9 private static void fillContingencyTable(List<String[]> groupA,
10                                           List<String[]> groupB,
11                                           int[] contingencyTable) {
```

```
12      for (String[]tokensA: groupA) {
13         int count = countAlleles(tokensA, allele);
14         // 这里count = 0, 1或2
15         contingencyTable[0][count]++; //  (索引0为groupA)
16      }
17
18      for (String[] tokensB: groupB) {
19         int count = countAlleles(tokensB, allele);
20         // 这里count = 0,1或2
21         contingencyTable[1][count]++; // (索引1为groupB)
22     }
23 }
24
25 private static int countAlleles(String[] tokens, String allele) {
26     int count = 0;
27     if (allele.equals(tokens[1])) {
28         count++;
29     }
30     if (allele.equals(tokens[2])) {
31         count++;
32     }
33     // 这里 count = 0, 1或2
34     return count;
35 }
```

buildOutputValue()方法的定义参见示例20-7，这个方法将生成最终输出，由归约器发出。

示例20-7：Cochran-Armitage: buildOutputValue()函数

```
1 String buildOutputValue(pValue, key, allele, tokens, contingencyTable) {
2    // tokens:              <GROUP></><allele1></><allele2></><ref></><snp-id></>
3    // tokens[index]           0         1          2          3         4
4    // tokens:              <mutation-class><|><gene-id>|<bioset_Id>
5    // tokens[index]           5              6          7
6    StringBuilder outputValue    = new StringBuilder();
7    outputValue.append(pValue);                  // p-value
8    outputValue.append(":");
9    outputValue.append(key.toString()); // reduce(key)=<chr-ID><:><start><:><stop>
10   outputValue.append(":");
11   outputValue.append(tokens[6]);               // <gene-id>
12   outputValue.append(":");
13   outputValue.append(tokens[5]);               // <mutation-class>
14   outputValue.append(":");
15   outputValue.append(tokens[3]);               // <ref>
16   outputValue.append(":");
17   outputValue.append(allele);                  // 候选等位基因
18   outputValue.append(":");
19   outputValue.append(contingencyTable[0][0]);  // n10 (组A, 计数0)
20   outputValue.append(":");
21   outputValue.append(contingencyTable[0][1]);  // n11 (组A, 计数1)
22   outputValue.append(":");
23   outputValue.append(contingencyTable[0][2]);  // n12 (组A, 计数2)
24   outputValue.append(":");
25   outputValue.append(contingencyTable[1][0]);  // n20 (组B, 计数0)
```

```
26    outputValue.append(":");
27    outputValue.append(contingencyTable[1][1]);   // n21 (组B, 计数1)
28    outputValue.append(":");
29    outputValue.append(contingencyTable[1][2]);   // n22 (组B, 计数2)
30    outputValue.append(":");
31    outputValue.append(tokens[4]);                // snp-id
32    return outputValue.toString();
33 }
```

# MapReduce/Hadoop实现类

CATT的Hadoop实现包括表20-12所示的类。

**表20-12：Hadoop实现类**

| 类名 | 类描述 |
|---|---|
| CochranArmitage | Cochran-Armitage趋势检验算法 |
| CochranArmitageAnalyzer | 读取和分析MapReduce程序生成的结果 |
| CochranArmitageClient | 提交MapReduce作业的简单类 |
| CochranArmitageDriver | 提交MapReduce作业的具体驱动器 |
| CochranArmitageItem | 输出记录会映射到这个bean类 |
| CochranArmitageItemFactory | 生成CochranArmitageItem对象的工厂类 |
| CochranArmitageItemImpl | CochranArmitageItem的实现 |
| CochranArmitageMapper | 为Cochran-Armitage趋势检验算法定义Hadoop map() |
| CochranArmitageReducer | 为Cochran-Armitage趋势检验算法定义Hadoop reduce() |
| PaginatedObject | 如何对结果分页以便在前端显示 |
| ResultObject | Cochran-Armitage趋势检验算法的结果对象 |

# 运行示例

下面各小节给出了Cochran-Armitage趋势检验MapReduce/Hadoop实现运行示例的输入、运行日志及生成的输出。

## 输入

```
$ cat run_CochranArmitage.sh
#!/bin/bash
client=CochranArmitageClient
java $client interactive 0out groupA_0.txt groupB_0.txt

$ cat groupA_0.txt
0

$ cat groupB_0.txt
0
```

```
$ hadoop fs -cat /germline/groupA/0/*
7:100:200;a|A|C|Ref|snpid|mc|geneid|1000
7:100:200;a|A|A|Ref|snpid|mc|geneid|2000
7:100:200;a|A|C|Ref|snpid|mc|geneid|3000
7:100:200;a|G|G|Ref|snpid|mc|geneid|4000
7:100:200;a|A|A|Ref|snpid|mc|geneid|5000
7:100:200;a|AC|T|Ref|snpid|mc|geneid|6000

$ hadoop fs -cat /germline/groupB/0/*
7:100:200;b|A|A|Ref|snpid|mc|geneid|7000
7:100:200;b|C|C|Ref|snpid|mc|geneid|7100
7:100:200;b|A|C|Ref|snpid|mc|geneid|7200
7:100:200;b|A|A|Ref|snpid|mc|geneid|7300
7:100:200;b|A|A|Ref|snpid|mc|geneid|7400
```

## 示例运行日志

下面给出运行示例的日志输出，因篇幅所限，这里对日志做了编辑，并对格式有所调整：

```
$ ./run_CochranArmitage.sh
16:34:40 [main] [INFO ] [CATT] - executionType: interactive
16:34:40 [main] [INFO ] [CATT] - requestID: 0out
16:34:40 [main] [INFO ] [CATT] - biosetIDsFilenameGroupA: groupA_0.txt
16:34:40 [main] [INFO ] [CATT] - biosetIDsFilenameGroupB: groupB_0.txt
...
16:34:42 [main] [INFO ] [...JobClient] - map 0% reduce 0%
...
16:35:13 [main] [INFO ] [...JobClient] - map 100% reduce 100%
16:35:18 [main] [INFO ] [CATT] - run(): Job Finished in 37.782 seconds
16:35:18 [main] [INFO ] [CATT - run(): jobid=job_201307110717_0285
16:35:18 [main] [INFO ] [CATT] - submitJob(): runStatus=0
```

## 生成的输出

```
$ hadoop fs -cat /biomarker/output/germline/0out/part*
0.2635524772829726:7:100:200:geneid:mc:Ref:AC:5:1:0:5:0:0:snpid
0.2635524772829726:7:100:200:geneid:mc:Ref:T:5:1:0:5:0:0:snpid
0.2635524772829726:7:100:200:geneid:mc:Ref:G:5:0:1:5:0:0:snpid
0.3545394797735012:7:100:200:geneid:mc:Ref:A:2:2:2:1:1:3:snpid
0.43276758066778453:7:100:200:geneid:mc:Ref:C:4:2:0:3:1:1:snpid
```

这一章使用MapReduce/Hadoop为生殖细胞数据输入（表示为VCF文件）实现了Cochran-Armitage趋势检验。下一章将为等位基因频率分析实现一个可伸缩的解决方案，其中也会使用生殖细胞数据作为输入。

# 第21章

# 等位基因频率

等位基因频率分析是查找基因组（*http://bit.ly/genomic_data*）数据（特别是生殖细胞数据类型（*http://bit.ly/germline_cells*））中等位基因频率的一种技术。等位基因频率（*http://bit.ly/allele_freq*）定义为“一个物种种群中指定染色体基因座上有某个特定等位基因的百分比”。这一章中，我们将开发一个MapReduce解决方案来聚集各个指定键（由[chromosome, start-position,stop-position]组成）对应的所有基因组数据，然后应用Fisher精确检验（Fisher’s Exact Test（*http://bit.ly/fishers_exact_test*）），这是一种统计检验，用来确定两组变量之间（这两组变量可以是病人的bioset，稍后会讨论）是否存在非随机关联。然后将对MapReduce程序的输出进行分析并作图。等位基因频率计算的输入来自DNA测序管道生成的VCF文件。一般地，每个VCF记录包括chromosome、start-position、stop-position、genomereference和两个等位基因（标记为allele1和allele2，一个来自母亲，另一个来自父亲）。这些信息足以对两个数据集完成等位基因频率分析。

这一章的主要目标是使用Fisher精确检验为等位基因频率计算提供一个MapReduce解决方案，由3个MapReduce作业组成。

为了充分理解等位基因频率的重要性和影响，首先必须了解突变（mutations）、迁移（migrations）和选择（selections）的含义。这些概念的详细信息参见Understanding Evolution项目的文章“DNA and Mutations”（*http://bit.ly/dna_and_mutations*）。计算等位基因频率对于生物学家非常重要：如果等位基因频率经过一段时间发生改变，这就表明出现了遗传漂变（genetic drift）或者种群中发生新的突变。

图21-1给出了突变的一个例子。从图中可以看到种群基因如何偏离哈迪-温伯格平衡（Hardy-Weinberg equilibrium）（*http://bit.ly/hardy-weinberg_equil*）。

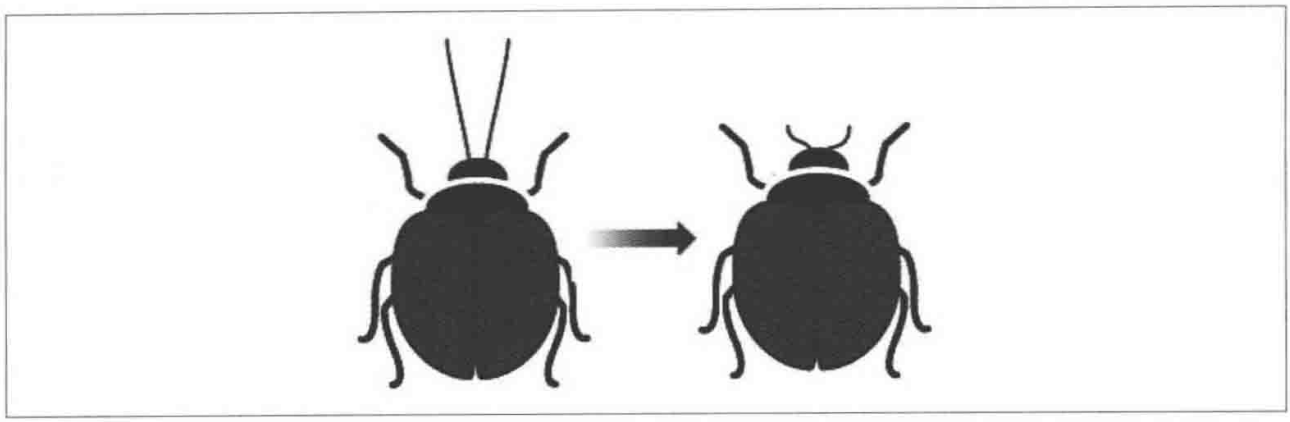


图21-1：突变示例

这一章中我们会完成以下工作：

1. 找出两个给定bioset组（A和B）（稍后定义）之间各个三元组（`chromosome`、`start-position`、`stop-position`）的p值（范围介于0.00~1.00之间的概率值）。这将由MapReduce阶段1完成。为了完成高效的运行时处理，对于给定的bioset，我们将创建原始数据的两个相同的副本，再增加一个`groupID a`或`b`来区别这两个副本。
2. 找出所有染色体中100个最小的p值，也就是说，这些p值最接近0.00（因为p值越接近0.00，就越有意义）。这说明，如果按升序对所有p值排序，那么前100个p值就是我们所要的答案。这将由MapReduce阶段2程序实现。阶段1的输出将用作为阶段2的输入。
3. 找出每个染色体的100个最小的p值，也就是最接近0.00的p值。这由MapReduce阶段3程序实现。阶段1的输出也将用作为阶段3的输入。

# 基本定义

介绍我们的MapReduce解决方案之前，首先需要定义几个重要的术语。

## 染色体（Chromosome）

染色体是细胞中DNA、蛋白质和RNA构成的一个有组织的结构。人类细胞有23对染色体，标记为`{1, 2, ..., 22, X, Y}`。

## 基因标签（Bioset）

bioset是单独分析得到的数据标签，其中包含试验样本比较形式的数据（对应转录组、表观遗传和拷贝数变异数据），以及基因型标签（对应GWAS（全基因组关联研究）和突变数据）。通常会把bioset称为基因标签（gene signature）。一个bioset的样本记录包含一个染色体、它的开始和结束位置、2个等位基因及其他相关的信息。bioset还可以有一个

关联的数据类型，这可以是基因表达式、蛋白质表达式、甲基化、拷贝数变异、miRNA或体细胞突变。假设有一组病人。每个病人有一组bioset（直接由VCF文件创建，VCF文件的每个记录包含基因组中一个位置的有关信息），每个bioset包含一组基因和关联的差异倍数值。每个bioset的记录数取决于它的数据类型。例如，一个生殖细胞bioset可以有430万条记录，而一个基因表达式bioset可能最多有50000个记录。

## 等位基因和等位基因频率

等位基因（allele）是染色体中一个特定基因座（位置）上的一个非常重要的DNA（脱氧核糖核酸，deoxyribonucleic acid）编码。每个染色体位置上有两个等位基因：`allele1`和`allele2`。如前所述，等位基因频率可以度量“一个物种种群中指定染色体基因座上有某个特定等位基因的百分比”。或者，等位基因频率也可以定义为种群中一个等位基因相对于相同基因其他等位基因的频率。一般会使用Fisher精确检验来计算等位基因频率的p值（概率值）。Fisher精确检验使用一个2×2列联表，表示不同的治疗会有怎样不同的结果。另外，Fisher精确检验基于一个零假设（null hypothesis），认为治疗不会影响结果（也就是说，二者是独立的）。

## 等位基因频率的数据源

鸟枪法测序仪（Shotgun sequencing machines）可以由生物（包括人类）样本生成DNA序列数据，这可能采用FASTQ（*http://bit.ly/fastq_format*）和其他流行格式。然后再利用DNA测序（*http://bit.ly/dna_sequencing*）软件和工作流分析这个FASTQ数据（这些样本的大小可以多达800 GB），最终生成VCF文件（*http://bit.ly/vc_format*）。我们要处理的VCF文件是对生殖细胞数据完成DNA测序的结果。VCF是一个文本文件格式，这些VCF文件会划分为元数据（所表示数据的类型、数据质量等有关信息）和具体数据。得到一个VCF文件后，可以创建bioset和生物标记物（biomarkers，将在进一步分析bioset/生物标志物组中包含的基因组时使用）。等位基因频率分析就是这样一种分析，这也是这一章的重点。图21-2给出了等位基因频率分析工作流图。

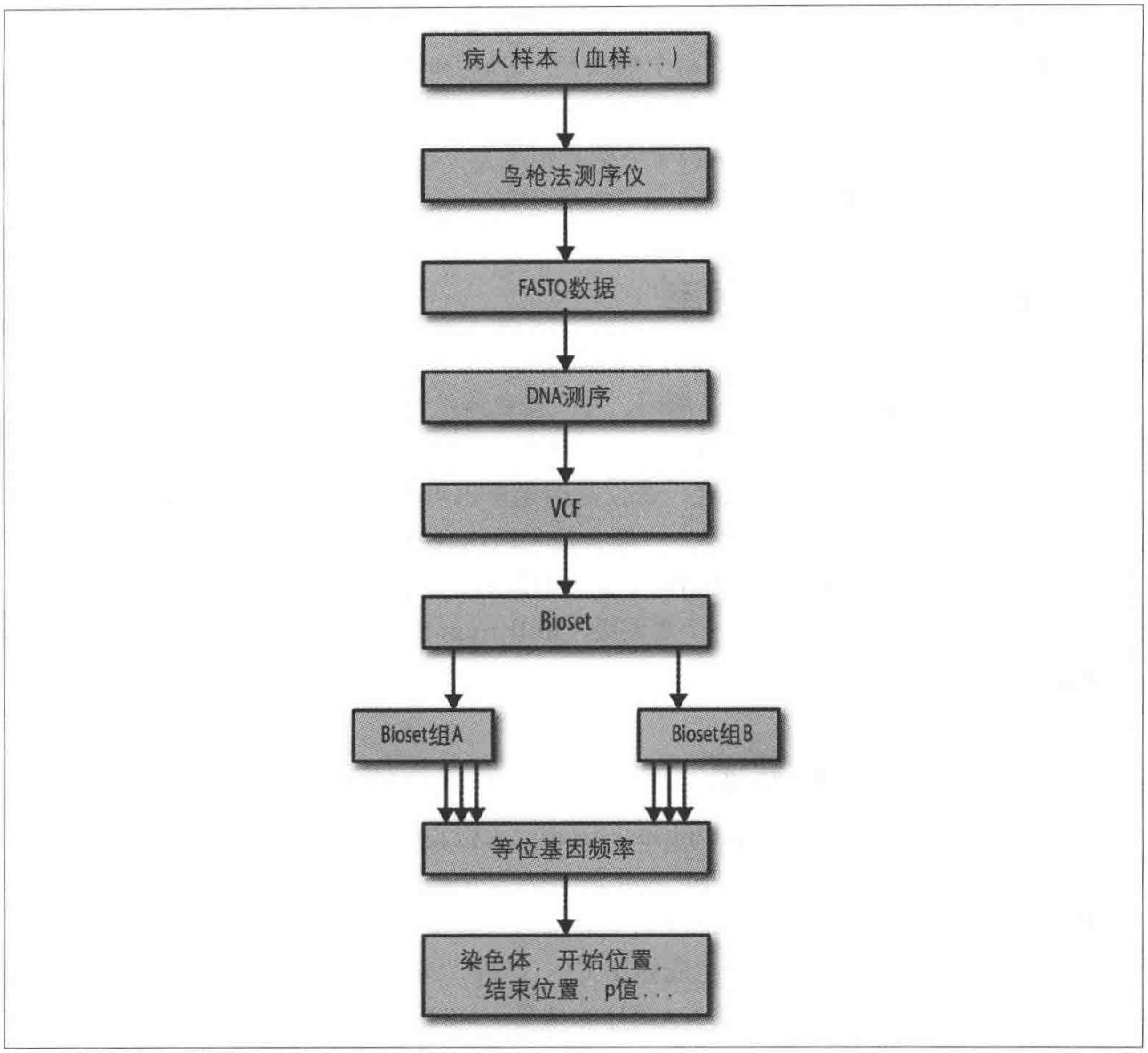



图21-2：等位基因频率分析工作流

生殖细胞数据的等位基因频率分析是一个大数据问题：每个生殖细胞bioset有约430万条记录，包含染色体ID、染色体开始位置、染色体结束位置、参考基因组、`allele1`和`allele2`等基因组信息。假设有两组生殖细胞bioset：A和B。如果组A有2000个bioset，组B有4000个bioset，我们想找出这两组bioset的等位基因频率，就必须分析：

$$4300000 \times (2000 + 4000) = 25800000000 \text{ 条记录}$$

要对组A和B中所有等位基因（`allele1,allele2`等）完成等位基因频率分析（如Fisher精确检验），如果使用一个包含15个节点的Hadoop集群，这需要约20分钟；如果有100个节点，可能只需要不到4分钟的时间。

## 使用Fisher精确检验的等位基因频率分析

一般地，得到一个给定染色体（由其染色体ID、开始位置和结束位置标识）的等位基因频率后，可以应用Fisher精确检验来找出pValue（0.0 ≤ pValue ≤ 1.0）。当pValue逐渐接近1.0，则认为行（组）与列（结果）之间的关联不具有统计意义；而当pValue接近0.0时，认为行（组）与列（结果）之间的关联具有统计意义。

## Fisher精确检验

Fisher精确检验（*http://bit.ly/fishers_exact_def*）是“医学研究中大量使用的一种独立性统计检验方法，根据观察频率的精确抽样分布检验一个2×2列联表（2个水平行，2个垂直列，共有4个数据）中行与列的独立性。因此这是一个精确检验”。Fisher精确检验 是Sir Ronald Aylmer Fisher设计提出的一种检验方法。

一般地，会在包含两种结果（结果1和结果2）的两个组（组1和组2）上完成Fisher精确检验。如表21-1所示。具体数据值用粗体显示。

**表21-1：Fisher精确检验**

| | 结果1 | 结果2 | 行总计 |
|---|---|---|---|
| 组1 | ***a*** | ***b*** | *a*+*b* |
| 组2 | ***c*** | ***d*** | *c*+*d* |
| 列总计 | *a*+*c* | *b*+*d* | *n*=*a*+*b*+*c*+*d* |

Fisher精确检验的公式如下：

$$p=\frac{\begin{pmatrix} a+b \\ a \end{pmatrix}\begin{pmatrix} c+d \\ c \end{pmatrix}}{\begin{pmatrix} n \\ a+c \end{pmatrix}}=\frac{(a+b)!(c+d)!(a+c)!(b+d)!}{a!\,b!\,c!\,d!\,n!}$$

其中 $\binom{n}{k}$ 是一个二项式系数，$n!$是$n$的阶乘。

利用Fisher精确检验可以确定什么？使用这个检验，我们可以统计确定两个类别变量（分别包含两个层次）之间是否存在关联。它会返回一个p值，指示观测数据达到已观测数据极限或者更极端事件发生的概率（如果满足零假设条件，也就是说两个层次之间没有差别）。

下面来看两个简单的例子。

表21-2显示了一个2×2列联表。

**表21-2：2×2列联表 (n = 34)**

| | 成功 | 失败 | 行总计 |
|---|---|---|---|
| 组1 | 4 | 12 | 16 |
| 组2 | 8 | 10 | 18 |
| 列总计 | 12 | 22 | $n = 34$ |

根据Fisher精确检验，双尾p值等于0.2966，可以认为行（组）与列（结果）之间的关联不具有统计意义。可以采用双尾或单尾检验来计算p值。建议使用双尾（也称为双端）p值（关于单尾和双尾p值的更多详细内容参见[2]）。

例如，在R编程语言中，可以如下完成Fisher精确检验：

```
# R
R version 2.15.1 (2012-06-22)
> mytable = rbind ( c(4, 12), c(8, 10) );
> mytable
     [,1] [,2]
[1,]    4    12
[2,]    8    10
> fisher.test(mytable)
  Fisher's Exact Test for Count Data
  data:   mytable
  p-value = 0.2966
```

下面来看另一个例子，如表21-3所示。

**表21-3：2×2列联表（n = 26）**

| | 成功 | 失败 | 行总计 |
|---|---|---|---|
| 组1 | 12 | 2 | 14 |
| 组2 | 2 | 10 | 12 |
| 列总计 | 14 | 12 | $n = 26$ |

根据Fisher精确检验，这里的双尾p值等于0.0011。可以认为行（组）与列（结果）之间的关联非常具有统计意义。

# 形式化问题描述

令A (由groupID a表示)和B（由groupID b表示）是两个生殖细胞数据bioset集（我们已经知道，这些bioset由DNA测序生成的VCF文件创建）。集合A的大小为$m$，集合B的大小$n$。需要说明，$m$和$n$可以相同，也可以不同。给定A和B，我们希望使用Fisher精确检验得出这两个组的等位基因频率。每个bioset记录表示一个变异，由以下属性标识。注意，一

个bioset记录可能包含超过50个属性，不过，对于等位基因频率分析来说，使用以下属性就足够了：

- Bioset的键属性：
  - 染色体ID: `1, 2, 3`等。
  - 染色体开始位置。
  - 染色体结束位置。
- Bioset的值属性：
  - 组ID: `a, b`。
  - `allele1`。
  - `allele2`。
  - 参考值。
  - SNP ID。
  - 突变类ID。
  - Gene ID。
  - Bioset ID。

# 等位基因频率分析的MapReduce解决方案

我们将通过一个3阶段的MapReduce算法实现等位基因频率分析：

- 阶段1：调用Fisher精确检验对数据完成聚集和分组，并生成p值。
- 阶段2：使用阶段1的输出，找出所有染色体中bottom 100（最小）的p值。我们特别将这个算法设计为可以找出bottom *N*个p值（N可以为任意值），例如，可以是bottom 50或者bottom 500。
- 阶段3：找出对应每个染色体的最小的100个p值，也就是说，最接近0.00的p值。例如，对于染色体5，最小的100个p值是什么？

# MapReduce解决方案，阶段1

这个阶段要找出两个给定bioset组（A和B）之间每个三元组（`chromosome, start-position, stop-position`）的p值。`map()`函数将按键（`chromosome, start-position, stopposition`）对输入记录分组。`Reduce()`函数会构造一个2×2列联表，最后调用Fisher精确检验。这里不能提供`combine()`函数，因为对于每个归约器需要所有等位基因。

## 输入

由于两个集合（A和B）可能选择相同的bioset，所以我们将生成两类数据（唯一的区别是group-name，对于组A，GROUP-NAME为a，对组B，GROUP-NAME为b，从而可以区别这两个组）。

每个bioset记录有以下格式：

```
<chromosome-ID>
<:>
<chromosome-start-position>
<:>
<chromosome-stop-position>
<;>
<GROUP-NAME>
<:>
<allele1>
<:>
<allele2>
<:>
<reference>
<:>
<snpID>
<:>
<mutationClassID>
<:>
<geneID>
<:>
<biosetID>
```

例如，如果组A选择6个bioset，那么对于染色体7，可能有以下数据：

```
7:10005296:10005296;a:A:C:A:snpid:mc:geneid:1000
7:10005296:10005296;a:A:A:A:snpid:mc:geneid:2000
7:10005296:10005296;a:A:C:C:snpid:mc:geneid:3000
7:10005296:10005296;a:G:G:G:snpid:mc:geneid:4000
7:10005296:10005296;a:A:A:A:snpid:mc:geneid:5000
7:10005296:10005296;a:AC:T:A:snpid:mc:geneid:6000
```

如果对组B选择5个bioset，对于染色体7，可能有以下数据：

```
7:10005296:10005296;b:A:A:A:snpid:mc:geneid:7000
7:10005296:10005296;b:C:C:C:snpid:mc:geneid:7100
7:10005296:10005296;b:A:C:C:snpid:mc:geneid:7200
7:10005296:10005296;b:A:A:A:snpid:mc:geneid:7300
7:10005296:10005296;b:A:A:A:snpid:mc:geneid:7400
```

## 输出/结果

每个输出/结果记录（对应每个chromosome-ID、chromosome-start-position和chromosome-stop-position，由Fisher精确检验生成的p值）将有以下格式：

```
<pValue>
<:>
<chromosome-ID>
<:>
<chromosome-start-position>
<:>
<chromosome-stop-position>
<:>
<geneID>
<:>
<mutationClassID>
<:>
<reference>
<:>
<alternate-allele>
<:>
<N11>
<:>
<N12>
<:>
<N21>
<:>
<N22>
<:>
<snpID>
```

需要说明，$N_{11}$, $N_{12}$, $N_{21}$和$N_{22}$指示生成的2 × 2 列联表中的值，如表21-4所示。

**表21-4：2×2列联表（$N_{11}$, $N_{12}$, $N_{21}$, $N_{22}$值）**

| | 已知 | 其他 | 行总和 |
|---|---|---|---|
| 组A | $N_{11}$ | $N_{12}$ | $N_{11} + N_{12}$ |
| 组B | $N_{21}$ | $N_{22}$ | $N_{21} + N_{22}$ |
| 列总和 | $N_{11} + N_{21}$ | $N_{12} + N_{22}$ | $n = N_{11} + N_{12} + N_{21} + N_{22}$ |

使用Fisher精确检验的等位基因频率分析将生成以下输出：

```
1.0:7:10005296:10005296:geneid:mc:A:AC:1:11:0:10:snpid
1.0:7:10005296:10005296:geneid:mc:A:T:1:11:0:10:snpid
0.480519480519484:7:10005296:10005296:geneid:mc:G:G:2:10:0:10:snpid
0.4148606811145488:7:10005296:10005296:geneid:mc:A:A:6:6:7:3:snpid
0.6240601503759411:7:10005296:10005296:geneid:mc:C:C:2:10:3:7:snpid
```

## 阶段1映射器

这个映射器如示例21-1所示，将为每个记录生成一个键-值对，其中键为：

```
<chromosome-ID><:><chromosome-start-position><:><chromosome-stop-position>
```

值包含其余属性：

```
<GROUP-NAME>
<:>
<allele1>
<:>
<allele2>
<:>
<reference>
<:>
<snpID>
<:>
<mutationClassID>
<:>
<geneID>
<:>
<biosetID>
```

示例21-1：等位基因频率: map()函数

```
/**
 * @param key为Hadoop生成的键，在这里忽略
 * @param value 有以下格式:
 *  <chrID><:><start><:><stop><;><GROUP><|><allele1><|><allele2><|><ref><|>
 *     <snpID><|><mutationClassID><|><geneID>|<biosetID>
 * 其中GROUP属于{"a" , "b"}
 *
 * reducerKey = Text: <chrID><:><start><:><stop>
 * reducerValue = Text: <GROUP><|><allele1><|><allele2><|><ref><|>
       <snpID><|><mutationClassID><|><geneID>|<biosetID>
 */
map(Text key, Text value) {
    String[] tokens = StringUtils.split(line, ";");
    if (tokens.length == 2) {
        String reducerKey = tokens[0];
        String reducerValue = tokens[1];
        emit(reducerKey, reducerValue);
    }
}
```

映射器函数生成键-值对，其中键包括{chrID, start, stop}，值包括{GROUP, allele1, allele2,ref, snpID, mutationClassID, geneID, biosetID}。因此，每个归约器将接受键<chrID><:><start><:><stop>，另外接受映射器发出的值列表作为值。

## 阶段1归约器

主要任务由归约器完成。每个归约器的键由映射器生成。每个归约器的值则是映射器生成的值列表。因此，键为7:10005296:10005296的归约器将有以下值：

```
a:A:C:A:snpid:mc:geneid:1000
a:A:A:A:snpid:mc:geneid:2000
a:A:C:C:snpid:mc:geneid:3000
a:G:G:G:snpid:mc:geneid:4000
a:A:A:A:snpid:mc:geneid:5000
```

```
a:AC:T:A:snpid:mc:geneid:6000
b:A:A:A:snpid:mc:geneid:7000
b:C:C:C:snpid:mc:geneid:7100
b:A:C:C:snpid:mc:geneid:7200
b:A:A:A:snpid:mc:geneid:7300
b:A:A:A:snpid:mc:geneid:7400
```

要实现这个归约器（参见示例21-2），需要3个基本散列表：

groupA

只出现在组A中的等位基因。

```
Map<String, Integer> groupA = new HashMap<String, Integer>();
```

groupB

只出现在组B中的等位基因。

```
Map<String, Integer> groupB = new HashMap<String, Integer>();
```

globalMap

组A或组B中的等位基因。

```
Map<String, String[]> globalMap = new HashMap<String, String[]>();
```

为各个键（<chrID><:><start><:><stop>）填充这3个散列表之后，将迭代处理globalMap来找出所有等位基因频率（参见示例21-3~示例21-5）。

示例21-2：等位基因频率: reduce()函数

```
/**
 * @param key为键: Text: <chrID><:><start><:><stop>
 * @param values是一个{value}列表，其中 各个值有以下格式:
 * <GROUP><:><allele1><:><allele2><:><ref><:><snpID><:>
 * <mutationClassID><:><geneID>:<biosetID>
 * 其中GROUP属于{"a" , "b"}
 *
 * outputKey = null (不需要键)
 * outputValue = 由buildOutputValue()方法构建
 */
reduce(Text key, Iterable<Text> values) {

    int totalNumOfBiosetsInGroupA = 0;
    int totalNumOfBiosetsInGroupB = 0;

    Map<String, String[]> globalMap = new HashMap<String, String[]>();
    Map<String, Integer> groupA = new HashMap<String, Integer>();
    Map<String, Integer> groupB = new HashMap<String, Integer>();

    // 步骤1: 迭代处理值来填充表 （参见示例21-3）

    // 步骤2: 创建Fisher精确检验的一个实例
    FisherExactTest theFisherExactTest = new FisherExactTest();

    // 步骤3: 迭代处理所建立的表
```

```
26    // 得出等位基因频率（参见示例21-4）
27 }
```

示例21-3：步骤1:迭代处理值来填充表

```
1    for (Text valueAsText: values) {
2        String value = valueAsText.toString();
3        String[] tokens = StringUtils.split(value, ":");
4        if (tokens.length != 8) {
5            continue;
6        }
7
8       // reduce(value) = <GROUP><:><allele1><:><allele2><:><ref><:><snpID><:>
9       // tokens[index]      0         1           2          3       4
10      // reduce(value) = <mutationClassID><:><geneID>:<biosetID>
11      // tokens[index]         5              6         7
12      String group = tokens[0];
13      String allele1 = tokens[1];
14      String allele2 = tokens[2];
15       if (group.equals("a")) {
16          // 组"a"
17          totalNumOfBiosetsInGroupA++;
18          // 处理 allele1
19          Integer count = groupA.get(allele1);
20          if (count == null) {
21              groupA.put(allele1, 1);
22              globalMap.put(allele1, tokens);
23          }
24          else {
25              groupA.put(allele1, Integer.valueOf(count.intValue() + 1));
26          }
27          // 处理 allele2
28          Integer count2 = groupA.get(allele2);
29          if (count2 == null) {
30              groupA.put(allele2, 1);
31              globalMap.put(allele2, tokens);
32          }
33          else {
34              groupA.put(allele2, Integer.valueOf(count2.intValue() + 1));
35          }
36       }
37       else {
38          // 处理组"b"的allele1
39          totalNumOfBiosetsInGroupB++;
40          Integer count = groupB.get(allele1);
41          if (count == null) {
42             groupB.put(allele1, 1);
43             // 查看是否要增加这个等位基因
44             if (!globalMap.containsKey(allele1)) {
45                 globalMap.put(allele1, tokens);
46             }
47          }
48          else {
49              groupB.put(allele1, Integer.valueOf(count.intValue() + 1));
50          }
51          // 处理allele2
```

```
52          Integer count2 = groupB.get(allele2);
53          if (count2 == null) {
54             groupB.put(allele2, 1);
55             // 查看是否要增加这个等位基因
56             if (!globalMap.containsKey(allele2)) {
57                 globalMap.put(allele2, tokens);
58             }
59           }
60           else {
61               groupB.put(allele2, Integer.valueOf(count2.intValue() + 1));
62           }
63        }
64
65    } // while-loop结束
66
67    if ( (groupA.size() == 0) || (groupB.size() == 0) ) {
68       return;
69    }
```

示例21-4：步骤3:迭代处理所建立的表得出等位基因频率

```
1    // 现在已经创建了所需的3个散列表：
2    // groupA<String, Integer>
3    // groupB<String, Integer>
4    // globalMap<allele, tokens[]>
5    //
6    // 接下来，迭代处理globalMap<allele, tokens[]) 并完成分析;
7    // 需要说明globalMap<allele, tokens[])包含所有等位基因
8    for (Map.Entry<String, String[]> entry : globalMap.entrySet()) {
9        String allele = entry.getKey(); // 键
10       String[] tokens = entry.getValue();
11
12       int n11 = 0;
13       int n12 = 0;
14       int n21 = 0;
15       int n22 = 0;
16
17       // 查看groupA是否有这个等位基因：
18       Integer countOfAllelesInGroupA = groupA.get(allele);
19       Integer countOfAllelesInGroupB = groupB.get(allele);
20       if (countOfAllelesInGroupA == null) {
21           // 那么这个等位基因只可能在groupB中
22           n11 = 0;
23           n12 = (2 * totalNumOfBiosetsInGroupA);
24           n21 = countOfAllelesInGroupB;
25           n22 = (2 * totalNumOfBiosetsInGroupB) - n21;
26       }
27       else if (countOfAllelesInGroupB == null) {
28           // 那么这个等位基因只可能在groupA中
29           n11 = countOfAllelesInGroupA;
30           n12 = (2 * totalNumOfBiosetsInGroupA) - n11;
31           n21 = 0;
32           n22 = (2 * totalNumOfBiosetsInGroupB);
33       }
34       else {
35           // 那么这个等位基因必然在两个组中都出现（groupA和groupB）
```

```
            n11 = countOfAllelesInGroupA;
            n12 = (2 * totalNumOfBiosetsInGroupA) - n11;
            n21 = countOfAllelesInGroupB;
            n22 = (2 * totalNumOfBiosetsInGroupB) - n21;
        }

        theFisherExactTest.init(n11, n12, n21, n22);
        double pValue = theFisherExactTest.get2Tail();
        String outputValue = buildOutputValue(pValue, key, allele,
                                              tokens, n11, n12, n21, n22);
        // 准备归约器提供输出
        emit(null, outputValue);
    } // for-loop结束
}
```

示例21-5：等位基因频率：buildOutputValue()函数

```
String buildOutputValue(pValue, key, allele, tokens, n11, n12, n21, n22) {
    StringBuilder outputValue = new StringBuilder();
    outputValue.append(pValue);         // p值
    outputValue.append(":");
    outputValue.append(key.toString()); // key = <chrID><:><start><:><stop>
    outputValue.append(":");
    outputValue.append(tokens[6]);      // <geneID>
    outputValue.append(":");
    outputValue.append(tokens[5]);      // <mutationClassID>
    outputValue.append(":");
    outputValue.append(tokens[3]);      // <ref>
    outputValue.append(":");
    outputValue.append(allele);         // 候选等位基因
    outputValue.append(":");
    outputValue.append(n11);            // n11
    outputValue.append(":");
    outputValue.append(n12);            // n12
    outputValue.append(":");
    outputValue.append(n21);            // n21
    outputValue.append(":");
    outputValue.append(n22);            // n22
    outputValue.append(":");
    outputValue.append(tokens[4]);      // snpID
    return outputValue.toString();
}
```

# MapReduce/Hadoop实现中阶段1的运行示例

下面各小节将提供计算等位基因频率的MapReduce/Hadoop实现中阶段1的脚本、输入、运行日志和所生成的输出。

## 脚本

```
# cat run_test.sh
#!/bin/bash
groupA=/bioset/input/groupA/groupA.txt
```

```
groupB=/bioset/input/groupB/groupB.txt
output=/bioset/output
java AllelicFrequencyClient interactive $groupA $groupB $output
```

## 输入

```
# hadoop fs -cat /bioset/input/groupA/groupA.txt
7:100:200;a|A|C|Ref|snpid|mc|geneid|1000
7:100:200;a|A|A|Ref|snpid|mc|geneid|2000
7:100:200;a|A|C|Ref|snpid|mc|geneid|3000
7:100:200;a|G|G|Ref|snpid|mc|geneid|4000
7:100:200;a|A|A|Ref|snpid|mc|geneid|5000
7:100:200;a|AC|T|Ref|snpid|mc|geneid|6000

# hadoop fs -cat /bioset/input/groupB/groupB.txt
7:100:200;b|A|A|Ref|snpid|mc|geneid|7000
7:100:200;b|C|C|Ref|snpid|mc|geneid|7100
7:100:200;b|A|C|Ref|snpid|mc|geneid|7200
7:100:200;b|A|A|Ref|snpid|mc|geneid|7300
7:100:200;b|A|A|Ref|snpid|mc|geneid|7400
```

## 运行脚本

这里给出一个运行示例的日志输出，因篇幅所限，这里对日志做了编辑，并对格式有所调整：

```
# ./run_test.sh
...
16:01:50 [AllelicFrequencyClient] - biosetIDsFilenameGroupA: groupA.txt
16:01:50 [AllelicFrequencyClient] - biosetIDsFilenameGroupB: groupB.txt
...
16:01:51 [AllelicFrequencyDriver] - GroupA biosetPath::/bioset/input/groupA::
16:01:51 [AllelicFrequencyDriver] - GroupB biosetPath::/bioset/input/groupB::
...
16:01:51 [org.apache.hadoop.mapred.JobClient] - Running job: job_201307110717
16:01:52 [org.apache.hadoop.mapred.JobClient] - map 0% reduce 0%
...
16:03:07 [org.apache.hadoop.mapred.JobClient] - map 100% reduce 100%
16:03:12 [AllelicFrequencyDriver] - run(): Job Finished in 80.862 seconds
16:03:12 [AllelicFrequencyDriver] - submitJob(): runStatus=0
```

## 生成的输出

```
# hadoop fs -cat /bioset/output/part*
1.0:7:100:200:geneid:mc:Ref:AC:1:11:0:10:snpid
1.0:7:100:200:geneid:mc:Ref:T:1:11:0:10:snpid
0.480519480519484:7:100:200:geneid:mc:Ref:G:2:10:0:10:snpid
0.4148606811145488:7:100:200:geneid:mc:Ref:A:6:6:7:3:snpid
0.6240601503759411:7:100:200:geneid:mc:Ref:C:2:10:3:7:snpid
```

## p值示例图示

假设我们将等位基因频率输出中的p值抽取到一个名为*pvalue.txt*的文件中，想要利用这些值作图（在这里我只从430万个数据中抽取了约40个p值）。使用R编程语言，可以很容易地做到：

```
# R
R version 2.15.1 (2012-06-22) -- "Roasted Marshmallows"
> p = read.table('pvalue.txt');
> p
    V1
1 0.00
2 0.05
...
36 0.90
37 1.00
38 1.00
> plot(table(p));
> title(main="Allelic Frequency");
> q();
```

输出将是一个PDF文件，如图21-3所示。

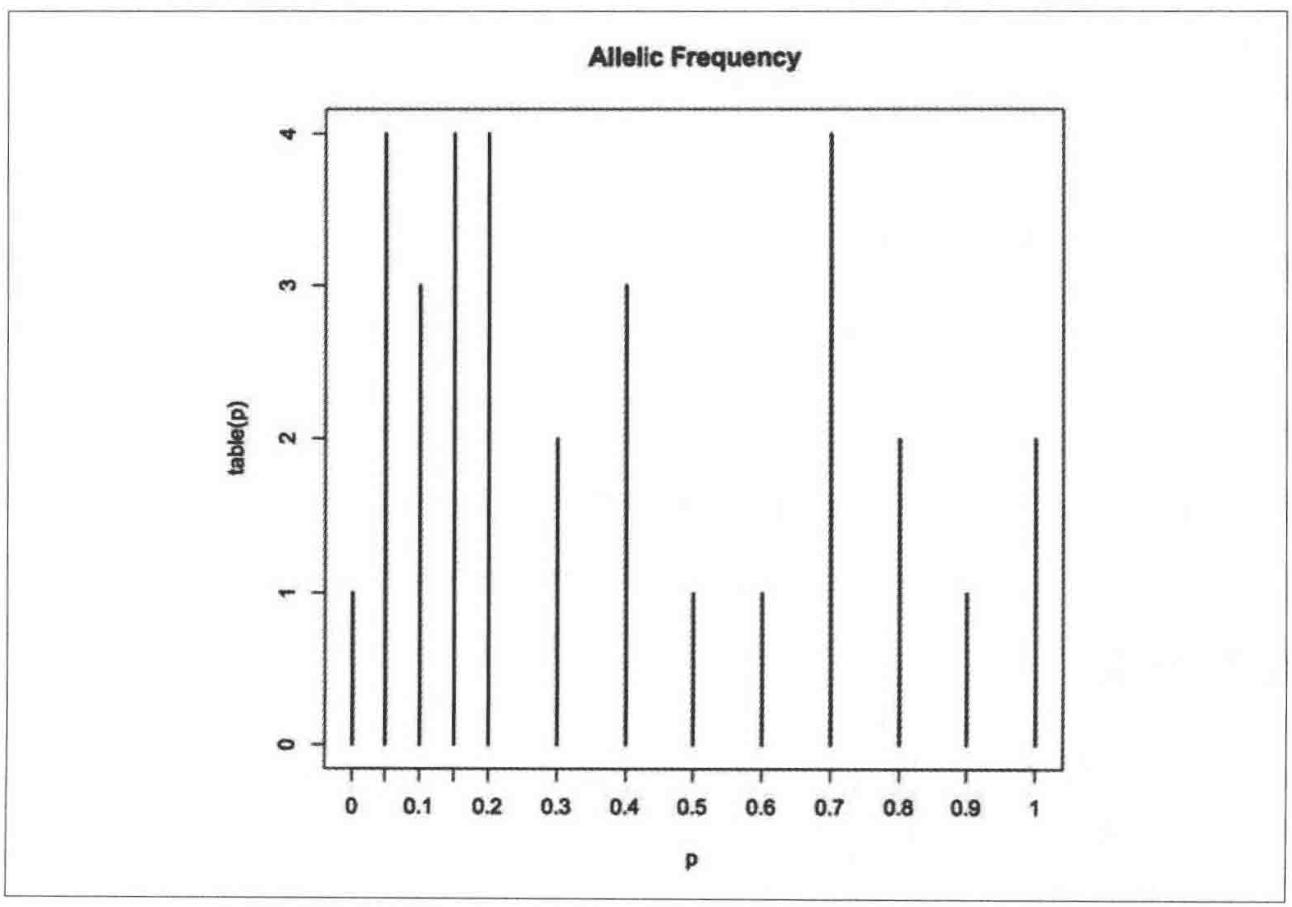



图21-3：等位基因频率的p值

# MapReduce解决方案，阶段2

这个阶段的目标是找出所有染色体中前100个最小的p值，也就是说，最接近0.00的100个p值。阶段1的输出将用作为阶段2的输入。最简单的解决方案是按p值对阶段1的输出（以升序）排序，然后输出前100个记录（这就是阶段2所要的输出）。不过，由于阶段1的输出可能非常庞大，排序并不是一个好办法，所以我们会用另一个MapReduce作业解决这个问题。如果使用一个关系数据库，如Oracle或MySQL，就能用一个SQL查询回答这个问题（假设allele_frequency_table是一个表，其中包含pValue和<other>列，并将阶段1的输出作为数据加载到这个表中）：

```
SELECT *
   FROM allele_frequency_table
      ORDER BY pValue LIMIT 100;
```

不过，本书前面解释过，由于我们要处理的数据量非常庞大，所以这个任务最好用一个MapReduce解决方案来处理。

通过使用等位基因频率算法（阶段1），对于两个组中所有给定bioset的所有染色体，可以生成（pValue, chromosomeID, start, stop, ...）。现在需要完成排序（以升序）来找出所生成记录中100个最小的p值。不过，如前所述，对等位基因频率算法生成的所有输出进行排序并不是一个好的解决方案。如果阶段1的输出非常庞大，这种排序有可能会耗尽内存。所以我们选择另一种做法：使用另一个MapReduce程序读取等位基因频率算法的输出，然后生成所有染色体的最小的100个p值。这个新MapReduce算法相当简单：

- 映射器。
  - — 每个映射器找出本地的bottom 100 p值列表，将这个bottom 100列表发送到归约器。
  - — 我们将使用多个映射器。
- 归约器。
  - — 归约器从映射器发送的所有bottom 100列表中找到最终的bottom 100 p值列表。
  - — 我们将使用一个归约器来找出最终的bottom 100。

## 阶段2 Bottom 100 p值：映射器

这些映射器要处理阶段1中等位基因频率算法生成的输出。每个映射器将读取（pValue, chromosomeID, start, stop, ...），然后发出这个映射器的bottom 100 p值，这里会使

用一个列表数据结构。我们将使用Java的TreeMap[注1]数据结构来维护bottom 100列表。将根据p值（作为一个Double键，$0 \leq pValue \leq 1$）保持列表有序：

```
import java.util.TreeMap;
import java.util.SortedMap;
SortedMap<Double, String> bottom100 = new TreeMap<Double, String>();
```

必须确保bottom100数据结构中只包含bottom 100 p值。示例21-6定义了Bottom100Mapper类中的map()函数：这个map()函数本身不发出任何键-值对，只是维护和更新bottom100数据结构。map()完成工作后，cleanup()函数会发出bottom 100列表。

示例21-6：Bottom100Mapper类

```
1 public class Bottom100Mapper ... {
2
3    private SortedMap<Double, String> bottom100 =
4        new TreeMap<Double, String>();
5
6    // 键是Double类型的p值，范围在0.00~1.00之间
7    // 值是整个等位基因频率输出记录（包括p值）
8    map(Double key, String entireRecord) {
9       bottom100.put(key, value); // 按p值排序
10      if (bottom100.size() > 100) {
11         // 删除最大的p值
12         bottom100.remove(bottom100.lastKey());
13      }
14   }
15
16   // 映射器任务结束时调用一次
17   cleanup() {
18       for (Map.Entry<Double, String> entry : bottom100.entrySet() {
19          Double pValue = entry.getKey();
20          String entireRecord = entry.getValue();
21          String outputValue = pair(pValue, entireRecord);
22          // NULL键会向一个归约器
23          // 发送所有键-值对
24          emit(NULL, outputValue);
25       }
26   }
27 }
```

## 阶段2 Bottom 100 p值：归约器

bottom 100算法设计为由一个归约器聚集所有映射器生成的所有bottom 100列表。这个归约器的键是NULL（在Hadoop中，这是一个NullWritable实例）。为了保证只有一个归约

注1：java.util.TreeMap是一个基于红黑树的NavigableMap实现。这个映射根据键的自然顺序排序，或者按创建映射时提供的一个Comparator来排序，具体取决于使用哪一个构造函数。这个实现可以保证containsKey、get、put和remove操作的时间开销为$\log(n)$。

器，还要在作业驱动器类中设置归约任务数为1（setNumOfReduceTasks()），如示例21-7所示。

示例21-7：Bottom100Driver

```
public class Bottom100Driver {
   ...
   void run(String[] args) {
      ...
      Job job = new Job(...);
      ...
      job.setNumReduceTasks(1);
      ...
   }
   ...
}
```

这个归约器的任务是从所有映射器生成的所有bottom 100列表生成最终的bottom 100列表。所以，这个归约器的功能（在示例21-8中定义）与映射器的功能非常类似。

示例21-8：Bottom100Reducer

```
public class Bottom100Reducer ... {

    reduce(NullWritable key, Iterable<pair<Double, String>> values) {
        SortedMap<Double, String> finalBottom100 =
            new TreeMap<Double, String>();

        for (pair(Double, String) value : values) {
            Double pValue = value.pValue;
            String entireRecord = value.entireRecord;
            finalBottom100.put(pValue, entireRecord);

            if (finalBottom100.size() > 100) {
               // 删除最大的p值
               finalBottom100.remove(finalBottom100.lastKey());
            }
         }

         // 现在就有了最终的bottom 100列表
         for (Map.Entry<Double, String> entry : finalBottom100.entrySet() {
            Double pValue = entry.getKey();
            String entireRecord = entry.getValue();
            emit(pValue, entireRecord);
         }
    }
}
```

## Bottom 100列表是一个幺半群吗？

接下来我们要确定这个bottom 100列表是否是一个幺半群（monoid）[注2]。如果是，就能

注2：第28章将解释MapReduce中的Monoids。

为这个bottom 100列表提供一个组合器。可以看到，bottom100是一个满足交换律的幺半群。

bottom100定义如下：

```
bottom100 = MONOID(S, e, f),
```

在这里：

S

{(p, w), 0 ≤ p ≤ 1, w是一个唯一的染色体记录}

e

{}是一个空集

f

是bottom100函数。

bottom100的交换律定义如下：

```
bottom100(a) = {(p1, w1), ..., (p100, w100)
其中 0 ≤ p1 ≤ ... ≤ p100 ≤ 1}

bottom100(a, b) = bottom100(a ∪ b) =
bottom100(b ∪ a) = bottom100(b, a)

bottom100(a, b) = c ∈ S
```

最后，bottom100的单位元特性表述如下：令a,b∈S。

则有：

```
bottom100(a, e) = a
bottom100(e, a) = a
```

采用数学记法，可以写为：

*闭包（Closure）*

```
∀a, b ∈ S : bottom100(a, b) ∈ S
```

*结合律（Associativity）*

```
∀a, b, c ∈ S : bottom100(a, bottom100(b, c))=bottom100(bottom100(a, b), c))
```

*单位元（Identity element）*

```
∃e∈S : ∀a ∈ S : bottom100(a, e) = a = bottom100(e, a)
```

*交换律（Commutative）*

$\forall a, b \in S$ : bottom100(a, b) = bottom100(b, a)

## Bottom 100列表的Hadoop实现类

表21-5中列出了bottom 100列表MapReduce解决方案中使用的Java类。在我们的解决方案中，利用了`PairOfDoubleString`[注3]类作为Hadoop `WritableComparable`，表示由一个`Double`和一个`String`组成的对。

**表21-5：bottom 100列表Hadoop解决方案中使用的Java类**

| 类名 | 描述 |
|---|---|
| `Bottom100Driver` | 提交Hadoop作业的驱动器程序 |
| `Bottom100Mapper` | 定义`map()` |
| `Bottom100Combiner` | 定义`combine()` |
| `Bottom100Reducer` | 定义`reduce()` |
| `HadoopUtil` | 定义一些工具函数 |
| `PairOfDoubleString` | WritableComparable，表示一个`Double`和`String` |

# MapReduce解决方案，阶段3

在阶段2中我们找出了所有染色体中最小的100个p值（也就是最接近0.00的p值）。在阶段3中，我们将找出每个染色体的100个最小的p值（同样的，最接近0.00的p值）。也就是说，我们想要分别找出以下染色体的最小的100个p值：

- Chromosome 1
- Chromosome 2
- ...
- Chromosome 22
- Chromosome X
- Chromosome Y

因此，我们要生成24个输出。阶段1的输出将用作为阶段3的输入。最容易的解决方法是对阶段1的输出按p值（以升序）排序，然后按染色体分组，最后输出每个染色体的前100个记录。不过，由于阶段1的输出可能相当庞大，排序并不是一个好办法。我们将使用另

注3：这是Jimmy Lin开发的Cloud9工具包中的一个类，Cloud9是一个Hadoop工具集，可以用来帮助处理大数据。

一个MapReduce作业解决这个问题。如果使用一个关系数据库（如Oracle或MySQL），可以用一个SQL查询回答这个问题（假设allele_frequency_table是一个表，其中包含pValue和<other>列，并将阶段1的输出作为数据加载到这个表中）。可以为每一个染色体运行以下SQL查询：

```
for (id in (1, 2, 3, ..., 21, 22, X, Y)) {
   SELECT *
      FROM allele_frequency_table
         WHERE chromosome_id = id
         ORDER BY pValue LIMIT 100;
}
```

MapReduce阶段3的算法相当简单：

- 映射器。
  - 每个映射器为各个染色体找出本地bottom 100 p值，将这个bottom 100列表发送到归约器（每个染色体对应一个归约器，总共24个归约器）。
  - 将使用多个映射器。
- 归约器。
  - 每个归约器从映射器发送的所有bottom 100列表找出最终的bottom 100 p值。
  - 将为每个染色体使用一个归约器来得到最终的bottom 100。

## 阶段3 Bottom 100 p值：映射器

映射器处理阶段1等位基因频率算法生成的输出。每个映射器会读取（pValue, chromosomeID, start, stop, ...），然后发出各个染色体的bottom 100 p值，这里会使用一个列表数据结构。如示例21-9所示，我们使用Java的TreeMap数据结构来维护bottom 100列表：将根据p值（作为一个Double键，0 ≤ pValue ≤ 1）保持列表有序。

*示例21-9：所需的数据结构*

```
1 import java.util.Map;
2 import java.util.HashMap;
3 import java.util.SortedMap;
4 import java.util.TreeMap;
5 ...
6 // map<chromosomeID, SortedMap<p-value, record>>
7 private Map<String, SortedMap<Double, String>> chromosomes =
8    new HashMap<String, TreeMap<Double, String>>();
```

要确保bottom100数据结构只包含bottom 100 p值。示例21-10定义了Bottom100MapperPhase3类中的map()函数：map()函数本身不会发出任何键-值对，不过它会维护和更新bottom100数据结构。map()完成工作后，cleanup()函数会发出各个染色体的bottom 100列表。

示例21-10：Bottom100MapperPhase3

```
public class Bottom100MapperPhase3 ... {
   // map<chromosomeID, treemap<p-value, record>>
   private Map<String, SortedMap<Double, String>> chromosomes =
       new HashMap<String, TreeMap<Double, String>>();

   //键是Double类型的p值，范围是0.00~1.00之间
   // 值是整个等位基因频率输出记录（包括p值）
   map(Double key, String entireRecord) {
      String chromosomeID = extract(entireRecord);
      SortedMap<Double, String> bottom100 = chromosomes.get(chromosomeID);
      if (bottom100 == null) {
         bottom100 = new TreeMap<Double, String>();
      }
      bottom100.put(key, value); // 按p值排序
      if (bottom100.size() > 100) {
         // 删除大的p值
         bottom100.remove(bottom100.lastKey());
      }
   }

   // 在映射器任务结束时调用一次
   cleanup() {
       for (Map.Entry<String, SortedMap<Double, String>>> entry :
         chromosomes.entrySet() {
         String chromosomeID = entry.getKey();
         SortedMap<Double, String> bottom100 = = entry.getValue();
         for (Map.Entry<Double, String> row : bottom100.entrySet() {
             Double pValue = row.getKey();
             String entireRecord = row.getValue();
             String outputValue = pair(pValue, entireRecord);
             // 对应特定染色体的输出对
             emit(chromosomeID, outputValue);
         }
      }
   }
}
```

## 阶段3 Bottom 100 p值：归约器

这个bottom 100算法设计为：对应每个染色体，只有一个归约器聚集所有映射器生成的所有bottom 100列表（注意，每个归约器会处理一个染色体的bottom 100，这与阶段2中不同，阶段2中会处理所有染色体）。归约器的键是chromosomeID。

每个归约器的任务是由（对应一个特定染色体的）所有映射器生成的所有bottom 100列表生成最终bottom 100列表。所以，这个归约器的功能（参见示例21-11中的定义）与映射器的功能非常相似。

示例21-11：Bottom100ReducerPhase3

```
public class Bottom100ReducerPhase3 ... {

```

```
    // key是一个chromosomeID , 属于{1, 2, ..., 22, X, Y}
    reduce(String key, Iterable<pair<Double, String>> values) {
        SortedMap<Double, String> finalBottom100 =
            new TreeMap<Double, String>();

        for (pair(Double, String) value : values) {
            Double pValue = value.pValue;
            String entireRecord = value.entireRecord;
            finalBottom100.put(pValue, entireRecord);

            if (finalBottom100.size() > 100) {
               // 删除大的p值
               finalBottom100.remove(finalBottom100.lastKey());
            }
        }

       // 现在我们就得到了最终的bottom 100列表
       for (Map.Entry<Double, String> entry : finalBottom100.entrySet() {
          Double pValue = entry.getKey();
          String entireRecord = entry.getValue();
          emit(key, (pValue + entireRecord));
       }
    }
}
```

## 各染色体Bottom 100列表的Hadoop实现

表21-6列出了各染色体bottom 100列表MapReduce解决方案中使用的Java类。在这个解决方案中，使用PairOfDoubleString类作为一个Hadoop WritableComparable，表示由一个Double和一个String组成的对。

**表21-6：各染色体bottom 100列表Hadoop解决方案中使用的Java类**

| 类名 | 描述 |
|---|---|
| Bottom100DriverPhase3 | 提交Hadoop作业的驱动器程序 |
| Bottom100MapperPhase3 | 定义map() |
| Bottom100CombinerPhase3 | 定义combine() |
| Bottom100ReducerPhase3 | 定义reduce() |
| HadoopUtil | 定义一些工具函数 |
| PairOfDoubleString | WritableComparable，表示一个Double和String |

# 染色体X 和Y的特殊处理

为了正确地计算一个给定生物标志物/bioset的等位基因频率，与其他染色体不同，我们要用另一种方式处理性染色体（染色体X和Y）。这说明，我们要修改算法来处理给定生殖细胞数据集的染色体X和Y。具体地，对于男性样本，染色体X和Y的非拟常染色体区

（non-pseudoautosomal regions，non-PARs）的等位基因应当只计一次，而不是两次。染色体X和Y上有两个PAR：

- PAR1:
  - 染色体X: 60001 – 2699520
  - 染色体Y: 10001 – 2649520
- PAR2:
  - 染色体X: 154931044 – 155260560
  - 染色体Y: 59034050 – 59373566

这个算法的实现（正确地处理染色体X和Y）留作练习，有兴趣的读者可以自行完成。

这一章提供了一个可伸缩的MapReduce解决方案，可以找出庞大生殖细胞数据集（表示为VCF文件）的等位基因频率。下一章将提供T检验的MapReduce和Spark解决方案。

# 第22章

# T检验

T检验（t-test，也称为双样本T检验（two-sample t-test））在临床应用和基因组分析中用来检验统计假设。独立样本T检验会比较两个样本的均值（$\mu$，也称为平均值）。在统计学中，要比较两个数据集，我们会把数据转换为一种更简单的形式，如数据的均值，然后计算和比较其均值。由于我们要比较随机样本，所以有可能存在随机误差（通常表示为样本的标准差或均方差$\sigma$）。有$N$个样本的种群的标准差公式定义如下：

$$\sigma = \sqrt{\frac{\sum_{i=1}^{N}(X_i - \mu)^2}{N}}$$

其中：

$\sigma$ = 标准差；
$X_i$ = 种群中的第$i$个值；
$\mu$ = 种群中值的均值。

因此，考虑到随机误差，我们可以比较$\mu \pm \sigma$。Sarah Boslaugh在《Statistics in a Nutshell》（O'Reilly出版）（*http://shop.oreilly.com/product/0636920023074.do*）中指出，“[T检验的]目的是确定抽取样本的种群是否有相同的均值。这里假设两个样本中的主体是无关的，而且要从其种群中独立选取。”

这一章将提供T检验的MapReduce/Hadoop和Spark解决方案。这里给出的MapReduce算法是通用的，可以用于检验任何大数据。

# 对bioset完成T检验

在基因组分析（特别是体细胞突变）中，可以对从相同基因（由geneID标识）抽取的一对样本使用T检验。给定一个biosets/生物标志物集合（标识为$B = \{B_1, B_2, ..., B_n\}$，其中每个bioset包含“geneID和geneValue对”[注1]），另外给定生存时间，标识为$\{S_1, S_2, ..., S_n\}$（注意，$S_i$对应$B_i$），我们的目标是创建两个样本集：Exist-Set和Non-Exist-Set。根据这个输入，我们要创建$\{G_1, G_2, ..., G_m\}$，这里$B_i$表示一个bioset集，$G_i$是其中一个bioset中的geneID（$G_i$相当于bioset的一个反向索引）。所以可以得到：

- Exist-Set样本：$\{S_i | B_i \in G_k\}$
- Non-Exist-Set样本：$\{S_i | B_i \notin G_k\}$

然后对Exist-Set和Non-Exist-应用ttest()函数。表22-1是一个包含4个bioset的具体示例（$B_i$是一个GUID，$T_i$为double基本数据类型）。

**表22-1：示例数据**

| biosetID | 生存时间 |
|---|---|
| $B_1$ | $T_1$ |
| $B_2$ | $T_2$ |
| $B_3$ | $T_3$ |
| $B_4$ | $T_4$ |

另外，假设bioset包含表22-2所示的geneID和geneValue。

**表22-2：Bioset数据**

| Bioset | geneID | geneValue |
|---|---|---|
| $B_1$ | G1 | V11 |
| | G2 | V12 |
| | G3 | V13 |
| $B_2$ | G1 | V21 |
| | G2 | V22 |
| | G4 | V24 |

注1：需要指出，geneID和geneValue对数取决于bioset类型。例如，对于RNA基因表达式数据类型，可能有40000+对，对于甲基化数据类型，则可能有20000+对。

表22-2：Bioset数据（续）

| Bioset | geneID | geneValue |
|---|---|---|
| $B_3$ | G1 | V31 |
| | G3 | V32 |
| | G4 | V33 |
| | G5 | V34 |
| | G6 | V35 |
| $B_4$ | G2 | V41 |
| | G5 | V42 |
| | G6 | V43 |

由此可以得到包含表（containment table），如表22-3所示。

表22-3：包含表

| | 时间 | | | |
|---|---|---|---|---|
| geneID | $T_1$ | $T_2$ | $T_3$ | $T_4$ |
| $G_1$ | $B_1$ | $B_2$ | $B_3$ | - |
| $G_2$ | $B_1$ | $B_2$ | - | $B_4$ |
| $G_3$ | $B_1$ | - | $B_3$ | - |
| $G_4$ | - | $B_2$ | $B_3$ | - |
| $G_5$ | - | - | $B_3$ | $B_4$ |
| $G_6$ | - | - | $B_3$ | $B_4$ |

例如，表22-2中的第一行指示G1包含在bioset{$B_1$,$B_2$,$B_3$}中，但是未包含在{$B_4$}中。根据这个表中的信息，我们会调用以下函数：

```
G1: ttest({T1, T2, T3}, {T4})
G2: ttest({T1, T2, T4}, {T3})
G3: ttest({T1, T3}, {T2, T4})
G4: ttest({T2, T3}, {T1, T4})
G5: ttest({T3, T4}, {T1, T2})
G6: ttest({T3, T4}, {T1, T2})
```

在ttest()实现中，我们将使用Apache Commons Math项目提供的以下对象：

- 接口：org.apache.commons.math.stat.inference.TTest
- 实现类：org.apache.commons.math.stat.inference.TTestImpl

ttest()实现（在TTestImpl类中使用）可以描述如下。这个统计可以用来完成双样本（sample1和sample2)T检验来比较样本均值。返回的T统计量如下所示：

$$t = \frac{(m_1 - m_2)}{\sqrt{\frac{v_1}{n_1} + \frac{v_2}{n_2}}}$$

其中:

- $n_1$是第一个样本（*sample1*）的大小。
- $n_2$是第二个样本（*sample2*）的大小。
- $m_1$是第一个样本的均值。
- $m_2$是第二个样本的均值。
- $v_1$是第一个样本的方差。
- $v_2$是第二个样本的方差。

如何使用这些对象来实现ttest()呢？示例22-1给出了一个简单的实现。

示例22-1：ttest()实现

```
import org.apache.commons.math.stat.inference.TTest;
import org.apache.commons.math.stat.inference.TTestImpl;
import org.apache.commons.math.MathException;

public class MathUtil {
   /**
    * @param sample1 = 样本数据值数组
    * @param sample2 =样本数据值数组
    * @return 观察的与双样本双尾T检验相关的显著性水平，或
    * p值，
    * 这个T检验会比较输入数组的均值
    */
    public static double ttest(double[] sample1, double[] sample2) {
       if ( (sample1 == null) ||
            (sample2 == null) ||
            (sample1.length == 0) ||
            (sample2.length == 0) ) {
          // 返回一个不存在的值
          return Double.MAX_VALUE;
       }

       if ((sample1.length == 1) && (sample2.length == 1)) {
          // 返回一个不存在的值
          return Double.MAX_VALUE;
       }
       return calculateTtest(sample1, sample2);
   }

   private static double calculateTtest(double[] sample1, double[] sample2) {
      ...
   }
```

```
32 }
```

calculateTtest()方法的实现参见示例22-2。

示例22-2：calculateTtest()方法

```
/**
 * @param sample1 = 样本数据值数组
 * @param sample2 = 样本数据值数组
 * @return p-value, 与sample1和sample2相关
 */
private static double calculateTtest(double[] sample1, double[] sample2) {
   double pValue = 0.0d;
   TTest ttest = new TTestImpl();
   try {
      if (sample1.length == 1) {
         pValue = ttest.tTest(sample1[0], sample2);
      }
      else if (sample2.length == 1) {
         pValue = ttest.tTest(sample2[0], sample1);
      }
      else {
         pValue = ttest.tTest(sample1, sample2);
      }
   }
   catch(MathException me) {
      System.out.println("ttest() failed. +", me.getMessage());
      pValue = 0.0d;
   }

   return pValue;
}
```

# MapReduce问题描述

给定一个bioset集[注2]，其中每个bioset可能有多达100000个基因，我们希望对所有bioset的各个基因应用ttest()函数。一般地，在基因组分析中，会对基因表达式、甲基化、体细胞突变和拷贝数变异bioset数据类型应用T检验。这种分析需要读取和处理数十亿条记录。例如，要处理200000个bioset（在病人为中心的应用中，这是一个正常的bioset数），每个bioset有50000个唯一的geneID，MapReduce解决方案就必须处理100亿（10000000000）条记录。这个数据量以及ttest()处理是一个服务器所无法胜任的。

# 输入

T检验的输入包括两部分（$B_i$对应$S_i$）：

注2：对于bioset的详细信息，参考附录A。

- bioset列表（$B_1, B_2, ..., B_n$）
- 这些bioset的生存时间（$S_1, S_2, ..., S_n$）

每个bioset记录有以下格式：

```
<geneID><,><biosetID><,><geneValue>
```

例如：

```
7562135,778800,1.04
```

或：

```
7570769,778800,-1.09
```

这些bioset文件持久存储在HDFS中，由MapReduce框架读取（map()执行期间完成读取）。如何从MapReduce驱动器将这两个额外的动态参数（bioset ID列表和这些bioset的生存时间）传递到map()和reduce()函数？通过使用MapReduce/Hadoop的Configuration对象，可以在map()或reduce()的setup()函数中通过Configuration.set()设置这些值，并利用Configuration.get()获取这些值。另一种做法是使用Hadoop的DistributedCache类[注3]，它能高效地分布存储应用特定的庞大的只读文件。

# 期望输出

对于（包含在所有bioset中的）各个geneID，需要生成：

```
<geneID><,><RESULT-of-TTEST-FUNCTION>
```

其中<RESULT-of-TTEST-FUNCTION>是一个p值。

# MapReduce解决方案

我们的MapReduce解决方案的目标是对bioset中包含的所有基因应用T检验算法。映射器得到<geneID><,><biosetID><,><geneValue>，发出一个键-值对，其中键是geneID，值是biosetID。归约器函数的任务是创建两个列表（Exist-Set和Non-Exist-Set），然后对这两个列表应用T检验。

如示例22-3所示，map()是一个简单函数，将发出（geneID,biosetID）对。

注3：这个类的全名为org.apache.hadoop.filecache.DistributedCache。DistributedCache是Hadoop MapReduce框架提供的一个类，用来缓存应用需要的文件。

示例22-3：T检验: map()函数

```
/**
 * @param key由MapReduce框架生成，在这里忽略
 * @param value是一个String，有以下格式：
 * <geneID><,><biosetID><,><geneValue>
 * 需要说明，<geneValue>在T检验中未使用。
 */
map(key, value) {
   String[] tokens = value.split(",");
   String geneID = tokens[0];
   String biosetID = tokens[1];
   emit(geneID, biosetID);
}
```

归约器（在示例22-4中定义）的主要功能是根据biosetID创建基因的一个反向索引，最后对两个集合应用T检验算法：Exist-Set和Non-Exist-Set。Reduce()函数实现T检验，并保持biosetID和生存时间的顺序。

示例22-4：T检验归约器

```
public class TtestReducer {

   // 实例变量
   private Configuration conf = null;
   // biosetID为String: 注意，bioset的顺序非常非常重要
   public List<String> biosets = null;
   // 生存时间：注意时间（time）项的顺序非常非常重要
   public List<Double> time = null;

   // 只运行一次
   public void setup(Context context) {
        this.conf = context.getConfiguration();

      // 从Hadoop配置得到参数
      String biosetsAsString = conf.get("biosets");
      this.biosets = DataStructuresUtil.splitOnToListOfString(biosetsAsString,
                                                               ",");

      String timeAsCommaSeparatedString = conf.get("time");
      this.time = DataStructuresUtil.toListOfDouble(timeAsCommaSeparatedString);
   }

   // key = geneID
   // values = { biosetID }列表
   public void reduce(Text key, Iterable<Text> values) {
      // 下面给出解决方案
   }
}
```

示例22-5中给出了T检验实现的Reduce()函数。

示例22-5：T检验：reduce()函数

```
// key = geneID
// values = { biosetID }列表
public void reduce(Text key, Iterable<Text> values) {
  Iterator<Text> iter = values.iterator();
  List<Double> exist = new ArrayList<Double>();
  List<Double> notexist = new ArrayList<Double>();
  List<String> genebiosets = new ArrayList<String>();
  while (iter.hasNext()) {
     String biosetID = iter.next();
     genebiosets.add(biosetID);
  }

  for (int i=0; i < biosets.size(); i++) {
     // 查看基因中是否存在这个biosetID
     int index = genebiosets.indexOf(biosets.get(i));
     if (index == -1) {
        // biosetID未找到
        notexist.add(time.get(i));
     }
     else {
        //找到biosetID
        exist.add(time.get(i));
     }
  }

  if (exist.isEmpty()) {
     // 不需要包括它
     return;
  }

  if (notexist.isEmpty()) {
     // 不需要包括它
     return;
  }

  //
  // 这里有：
  // (exist.size() > 0) && (notexist.size() > 0)
  //
  // 准备归约器的最终值
  //
  double pValue = MathUtil.ttest(exist, notexist); // 调用 ttest()
  emit(key, pValue);
}
```

## Hadoop实现类

这一节使用Hadoop实现T检验的一个MapReduce解决方案。这个Hadoop解决方案包括表22-4所示的Java类。

表22-4：使用MapReduce/Hadoop的T检验实现类

| 类名 | 描述 |
| --- | --- |
| TtestDriver | 提供MapReduce作业的驱动器类 |
| TtestMapper | 定义map()函数 |
| TtestReducer | 定义reduce()函数 |
| TtestAnalyzer | 读取归约器生成的输出数据 |

# Spark实现

这一节使用Spark实现T检验的一个MapReduce解决方案。这个Spark解决方案用一个驱动器Java类实现，其中要使用RDD。T检验涉及3类输入：

- biosetID列表 $\{B_1, B_2, ..., B_n\}$，其中 $n > 0$，每个$B_i$分别是一个bioset ID。
- 生成时间列表$\{T_1, T_2, ..., T_n\}$，其中每个$T_i$是一个Double数据类型。各个$B_i$分别与$T_i$对应。
- $n$个文件的列表（每个bioset对应一个文件）：{B1.txt, B2.txt, ..., Bn.txt}。bioset文件中的各个记录有以下格式（需要说明，对于ttest，未使用geneValue）：

```
<geneID><,><biosetID><,><geneValue>
```

由于每个$B_i$对应$T_i$，我们将这两个数据组合到一个文件中，名为*timetable.txt*，其中每个记录有两列：<bioset_id>和<survival_time>。因此，假设有以下输入文件：

```
/<hdfs-directory>/timetable/timetable.txt
/<hdfs-directory>/biosets/bioset_1.txt
/<hdfs-directory>/biosets/bioset_2.txt
/<hdfs-directory>/biosets/bioset_3.txt
...
/<hdfs-directory>/biosets/bioset_n.txt
```

## 高层步骤

使用这些输入文件，T检验的Spark工作流如图22-1所示。

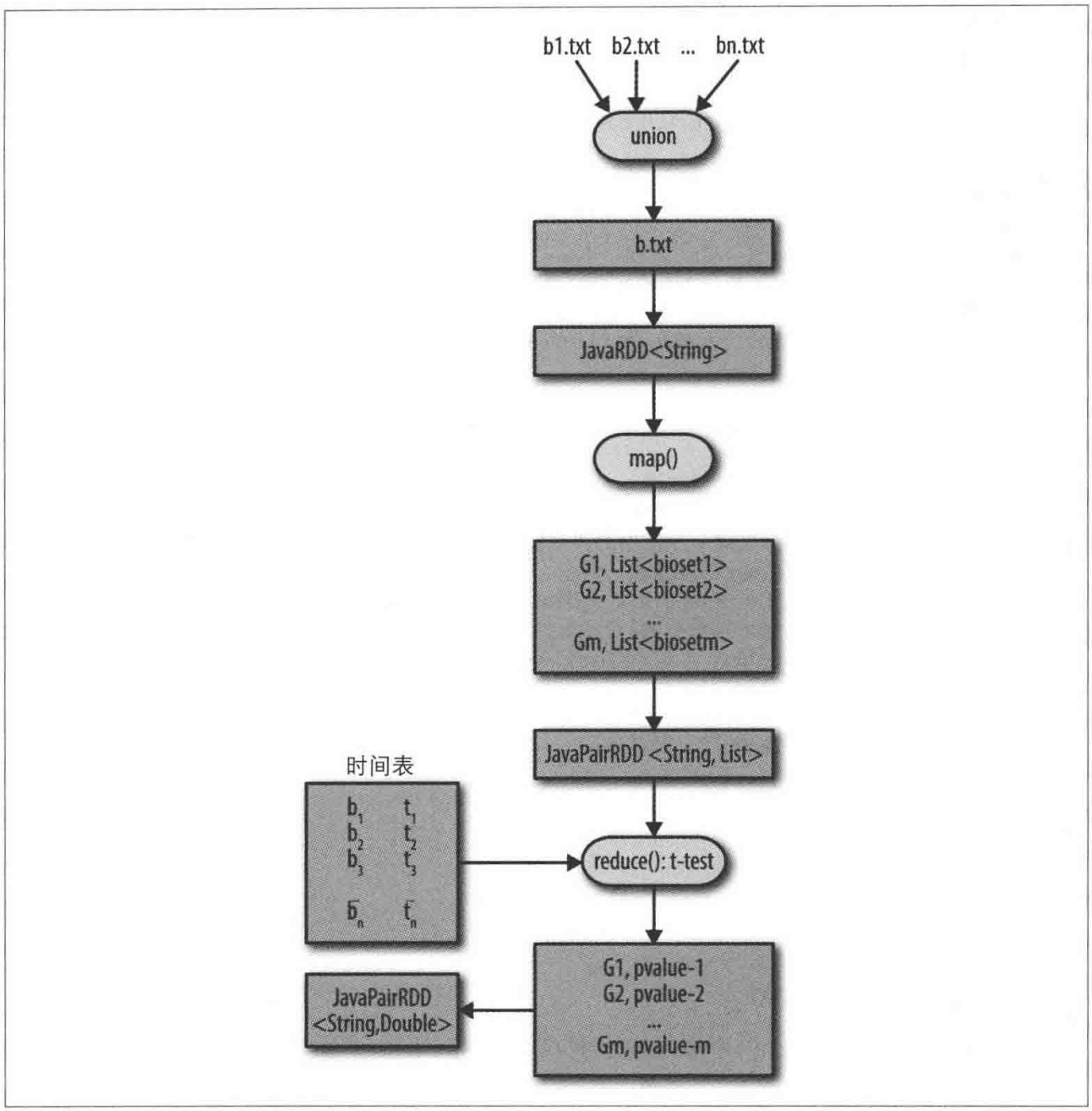



图22-1：T检验的Spark工作流

Spark工作流的高层实现如下所述（参见示例22-6）：

1. 连接所有bioset文件（*B1.txt*, *B2.txt*, ..., *Bn.txt*），创建一个`JavaRDD<String>`，这个RDD的每一项（作为一个`String`对象）有以下格式：

   ```
   <geneID><,><biosetID><,><geneValue>
   ```

2. 将`JavaRDD<String>`映射到一个`JavaRDD<String, Iterable<String>>`，其中键（作为一个`String`）是一个`geneID`，值是一个bioset列表。

3. 读取时间表数据得到（`biosetID,survival-time`）对，使用T检验算法将`JavaRDD-`

<String, Iterable<String>>归约为最终的JavaRDD<String,Double>，其中键（作为一个String）是一个geneID，值是一个p值（T检验算法的输出）。

4. 将最终的JavaPairRDD保存在HDFS中。

示例22-6：Spark解决方案：高层步骤

```
// 步骤1：导入所需的类和接口
public class SparkTtest {
   static JavaSparkContext createJavaSparkContext() {...}
   static Map<String, Double> createTimeTable(String filename) {...}
   static JavaRDD<String> readBiosetFiles(JavaSparkContext ctx,
                                          String biosetFiles) {...}

   public static void main(String[] args) throws Exception {
      // 步骤2：处理输入参数
      // 步骤3：创建时间表数据结构
      // 步骤4：创建一个Spark上下文对象（ctx）
      // 步骤5：广播所有集群节点使用的共享变量
      // 步骤6：为所有bioset创建RDD，其中
      // 每个RDD元素包括：<geneID><,><biosetID><,><geneValue>
      // 步骤7：将bioset记录映射为JavaPairRDD<K,V>对
       // 其中 K = <geneID>, V = <biosetID>
       // 步骤8：按geneID对bioset分组；
       // 现在，对于每个geneID，会有一个List<biosetID>
       // 步骤9：对每个geneID完成T检验

       // 完成
       ctx.close();
       System.exit(0);
   }
}
```

## 创建JavaSparkContext

createJavaSparkContext()方法（如示例22-7所示）用来创建一个JavaSparkContext对象，这是一个创建新RDD的工厂对象。

示例22-7：createJavaSpark()方法

```
   static JavaSparkContext createJavaSparkContext() throws Exception {
      return new JavaSparkContext();
   }
```

## createTimeTable()方法

createTimeTable()方法如示例22-8所示，将创建我们的TimeTable数据结构，这是一个（K,V）对散列表，其中K是一个biosetID，V是与K关联的时间。

示例22-8：创建TimeTable数据结构

```
   static Map<String, Double> createTimeTable(String filename) throws Exception {
      Map<String, Double> map = new HashMap<String, Double>();
```

```
      BufferedReader in = null;
      try {
         in = new BufferedReader(new FileReader(filename));
         String line = null;
         while ((line = in.readLine()) != null) {
            String value = line.trim();
            String[] tokens = value.split("\t");
            String biosetID = tokens[0];
            Double time = new Double(tokens[1]);
            map.put(biosetID, time);
         }
         System.out.println("createTimeTable() map="+map);
      }
      finally {
        if (in != null) {
         in.close();
        }
      }
      return map;
   }
```

## readBiosetFiles()方法

我们将使用这个JavaSparkContext为所有给定的bioset创建一个RDD，并通过readBiosetFiles()方法读取bioset文件来填充这个RDD。这个方法再将RDD划分为14个分区来完成进一步的分布和并行处理（参见示例22-9）（这里只是要展示这个过程，所以使用了任意的分区数。实际需要的分区数取决于具体数据和集群的大小）。确定适当的分区数对程序的性能非常重要。

示例22-9：为所有bioset创建RDD并分区

```
static JavaRDD<String> readBiosetFiles(JavaSparkContext ctx,
                                       String biosetFiles)
      throws Exception {
      StringBuilder unionPath = new StringBuilder();
      BufferedReader in = null;
      try {
           in = new BufferedReader(new FileReader(biosetFiles));
           String singleBiosetFile = null;
           while ((singleBiosetFile = in.readLine()) != null) {
              singleBiosetFile = singleBiosetFile.trim();
              unionPath.append(singleBiosetFile);
              unionPath.append(",");
           }
           //System.out.println("readBiosetFiles() unionPath="+unionPath);
      }
      finally {
        if (in != null) {
          in.close();
        }
      }
      // 删除最后一个逗号","
      String unionPathAsString = unionPath.toString();
```

```
23        unionPathAsString = unionPathAsString.substring(0,
24                          unionPathAsString.length()-1);
25        // 创建RDD
26        JavaRDD<String> allBiosets = ctx.textFile(unionPathAsString);
27        JavaRDD<String> partitioned = allBiosets.coalesce(14);
28        return partitioned;
29  }
```

## 步骤1：导入所需的类和接口

示例22-10导入了这个Spark实现所需的类和接口。

示例22-10：步骤1: 导入所需的类和接口

```
1     // 步骤1: 导入所需的类和接口
2     import scala.Tuple2;
3     import org.apache.spark.api.java.JavaRDD;
4     import org.apache.spark.api.java.JavaPairRDD;
5     import org.apache.spark.api.java.JavaSparkContext;
6     import org.apache.spark.api.java.function.Function;
7     import org.apache.spark.api.java.function.PairFunction;
8     import org.apache.commons.lang.StringUtils;
9     import org.apache.spark.broadcast.Broadcast;
10    import org.apache.spark.SparkConf;
11
12    import java.util.Map;
13    import java.util.HashMap;
14    import java.util.Set;
15    import java.util.HashSet;
16    import java.util.List;
17    import java.util.ArrayList;
18    import java.io.FileReader;
19    import java.io.BufferedReader;
```

## 步骤2：处理输入参数

这个步骤如示例22-11所示，这里读取两个输入参数：时间表文件和bioset文件。

示例22-11：步骤2: 处理输入参数

```
1 // 步骤2: 处理输入参数
2 if (args.length != 2) {
3    System.err.println("Usage: SparkTtest <timetable-file> <bioset-file>");
4    System.exit(1);
5 }
6 String timeTableFileName = args[0];
7 System.out.println("<timetable-file>="+timeTableFileName);
8 String biosetFileNames = args[1];
9 System.out.println("<bioset-file>="+biosetFileNames);
```

## 步骤3：创建TimeTable数据结构

这个步骤如示例22-12所示，将创建TimeTable数据结构，这是一个Map<biosetID, time>散列表。

示例22-12：步骤3: 创建时间表数据结构

```
// 步骤3：创建时间表数据结构
Map<String, Double> timetable = createTimeTable(timeTableFileName);
```

## 步骤4：创建一个Spark上下文对象

这个步骤如示例22-13所示，将创建一个JavaSparkContext实例，这是一个工厂类，用来创建新的RDD。

示例22-13：步骤4: 创建一个Spark上下文对象

```
// 步骤4：创建一个Spark上下文对象
JavaSparkContext ctx = createJavaSparkContext();
```

## 步骤5：广播共享数据结构

Spark的Broadcast类可以用来在所有集群节点间广播和共享全局数据结构。一旦广播一个数据结构，就可以从任意的集群节点（利用Spark变换）使用Broadcast.value()方法读取/获取这个数据。参见示例22-14。

示例22-14：步骤5: 广播共享数据结构

```
// 步骤5：广播所有集群节点使用的共享数据结构和变量；
// 按geneID对数据分组后，
// 想要查找Exist-Set和Not-Exist-Set集合时，
// 就会需要这个共享数据结构。
final Broadcast<Map<String, Double>> broadcastTimeTable=
  ctx.broadcast(timetable);
```

## 步骤6：为所有bioset创建RDD

这个步骤如示例22-15所示，读取所有输入数据，并为所有输入bioset创建一个RDD。

示例22-15：步骤6: 为所有bioset创建RDD

```
// 步骤6：为所有bioset创建RDD;
// 每个RDD元素包括： <geneID><,><biosetID><,><geneValue>
JavaRDD<String> biosets = readBiosetFiles(ctx, biosetFileNames);
biosets.saveAsTextFile("/ttest/output/1");
```

## 步骤7：将bioset记录映射为JavaPairRDD<K,V>对

这个步骤如示例22-16所示，将JavaRDD<String> bioset转换为JavaPairRDD<geneID, biosetID>对。

示例22-16：步骤7: 将bioset记录映射为JavaPairRDD<K,V>对

```
// 步骤7：将bioset记录映射为JavaPairRDD<K,V>对
// 这里 K = <geneID>
// V = <biosetID>
// 需要说明，对于T检验，未使用<geneValue>（在这里忽略）。
```

```
5     JavaPairRDD<String, String> pairs = biosets.mapToPair(
6         new PairFunction<
7                         String, // T
8                         String, // K
9                         String  // V
10                       >() {
11        public Tuple2<String, String> call(String biosetRecord) {
12            String[] tokens = StringUtils.split(biosetRecord, ",");
13            String geneID = tokens[0];   // K
14            String biosetID = tokens[1]; // V
15            return new Tuple2<String,String>(geneID, biosetID);
16        }
17    });
18    pairs.saveAsTextFile("/ttest/output/2");
```

## 步骤8：按geneID对bioset分组

如示例22-17所示，这一步使用JavaPairRDD.groupByKey()对一个geneID对应的所有基因值分组。

示例22-17：步骤8: 按geneID对bioset分组

```
1     // 步骤8: 按geneID对bioset分组
2     JavaPairRDD<String, Iterable<String>> grouped = pairs.groupByKey();
3     // 现在，对于每个geneID，会有一个List<biosetID>
4     grouped.saveAsTextFile("/ttest/output/3");
```

## 步骤9：对每个geneID完成T检验

在这一步中（如示例22-18所示），对基因值应用具体的T检验算法。

示例22-18：步骤9: 对每个geneID完成T检验

```
1     // 步骤9: 对每个geneID完成T检验
2     // mapValues[U](f: (V) => U): JavaPairRDD[K, U]
3     // 通过一个映射函数传入键-值对RDD中的各个值
4     // 键不改变；还将保留原来的RDD分区。
5     JavaPairRDD<String, Double> ttest = grouped.mapValues(
6         new Function<
7                     Iterable<String>, // 输入
8                     Double            // 输出（T检验的结果）
9                    >() {
10        public Double call(Iterable<String> biosets) {
11            Set<String> geneBiosets = new HashSet<String>();
12            for (String biosetID : biosets) {
13                geneBiosets.add(biosetID);
14            }
15
16            // 现在需要一个共享的Map数据结构来迭代处理各项
17            Map<String, Double> timetable = broadcastTimeTable.value();
18            // 完成T检验需要下面两个列表 (exist, notexist)
19            List<Double> exist = new ArrayList<Double>();
20            List<Double> notexist = new ArrayList<Double>();
21            for (Map.Entry<String, Double> entry : timetable.entrySet()) {
```

```
                String biosetID = entry.getKey();
                Double time = entry.getValue();
                if (geneBiosets.contains(biosetID)) {
                   exist.add(time);
                }
                else {
                   notexist.add(time);
                }
             }

            // 完成T检验ttest(exist, notexist)
            double ttest = MathUtil.ttest(exist, notexist);
            return ttest;
         }
      });
      ttest.saveAsTextFile("/ttest/output/4");
```

## T检验算法

我们使用Apache Commons Math的解决方案来提供T检验算法（参见示例22-19），其中：

`TTest`

是一个接口，指定了多个T检验算法。

`TTestImpl`

是实现TTest接口的类。

示例22-19：Apache Commons Math的T检验算法解决方案

```
import java.util.List;
import java.util.ArrayList;
import org.apache.commons.math.stat.inference.TTest;
import org.apache.commons.math.stat.inference.TTestImpl;

public class MathUtil {

   private static final TTest ttest = new TTestImpl();

   public static double ttest(double[] arrA, double[] arrB) {
      if ((arrA.length == 1) && (arrB.length == 1)) {
         //对score返回一个NULL值（无意义）
         return Double.NaN;
      }

      double score = Double.NaN;
      try {
         if (arrA.length == 1) {
            score = ttest.tTest(arrA[0], arrB);
         }
         else if (arrB.length == 1) {
            score = ttest.tTest(arrB[0], arrA);
```

```
23          }
24          else {
25             score = ttest.tTest(arrA, arrB);
26          }
27       }
28       catch(Exception e) {
29          e.printStackTrace();
30          score = Double.NaN;
31       }
32       return score;
33    }
34
35    public static double ttest(List<Double> groupA, List<Double> groupB) {
36       if ((groupA.size() == 1) && (groupB.size() == 1)) {
37          return Double.NaN;
38       }
39
40       double score;
41       if (groupA.size() == 1) {
42          score = tTest(groupA.get(0), groupB);
43       }
44       else if (groupB.size() == 1) {
45          score = tTest(groupB.get(0), groupA);
46       }
47       else {
48          score = tTest(groupA, groupB);
49       }
50       return score;
51    }
52
53    private static double tTest(double d, List<Double> group) {
54       try {
55          double[] arr = listToArray(group);
56          return ttest.tTest(d, arr);
57       }
58       catch(Exception e) {
59          e.printStackTrace();
60          return Double.NaN;
61       }
62    }
63
64    private static double tTest(List<Double> groupA, List<Double> groupB) {
65       try {
66          double[] arrA = listToArray(groupA);
67          double[] arrB = listToArray(groupB);
68          return ttest.tTest(arrA, arrB);
69       }
70       catch(Exception e) {
71          e.printStackTrace();
72          return 0.0d;
73       }
74    }
75
76    static double[] listToArray(List<Double> list) {
77       if ( (list == null) || (list.isEmpty()) ) {
```

```
78          return null;
79       }
80
81       double[] arr = new double[list.size()];
82       for (int i=0; i < arr.length; i++) {
83          arr[i] = list.get(i);
84       }
85       return arr;
86    }
87 }
```

# 运行示例

## 输入

这个T检验Spark程序有两个输入，分别是时间表和bioset，描述如下：

```
# cat ttestinput/timetable.txt
b1        0.9
b2        0.75
b3        0.5
b4        1.1

# cat ttestinput/biosets.txt
/ttest/input/b1.txt
/ttest/input/b2.txt
/ttest/input/b3.txt
/ttest/input/b4.txt

# hadoop fs -ls /ttest/input/
Found 4 items
-rw-r--r--   3 hadoop root,hadoop   52 2014-07-06 22:10 /ttest/input/b1.txt
-rw-r--r--   3 hadoop root,hadoop   33 2014-07-06 22:10 /ttest/input/b2.txt
-rw-r--r--   3 hadoop root,hadoop   31 2014-07-06 22:10 /ttest/input/b3.txt
-rw-r--r--   3 hadoop root,hadoop   31 2014-07-06 22:10 /ttest/input/b4.txt

# hadoop fs -cat /ttest/input/b1.txt
G1,b1,1.0
G2,b1,0.6
G3,b1,0.09
G4,b1,2.2
G5,b1,1.03

# hadoop fs -cat /ttest/input/b2.txt
G1,b2,2.09
G2,b2,1.07
G3,b2,1.00

# hadoop fs -cat /ttest/input/b3.txt
G2,b3,2.9
G3,b3,1.03
G5,b3,2.9

# hadoop fs -cat /ttest/input/b4.txt
```

```
G1,b4,2.9
G2,b4,1.8
G4,b4,1.05
```

## YARN shell脚本

下面的shell脚本可以在YARN环境中运行我们的Spark程序：

```
# cat ./run_ttest.sh
#!/bin/bash
export JAVA_HOME=/usr/java/jdk7
export BOOK_HOME=/mp/data-algorithms-book
export SPARK_HOME=/usr/local/spark-1.0.0
export HADOOP_HOME=/usr/local/hadoop-2.5.0
export HADOOP_CONF_DIR=$HADOOP_HOME/etc/hadoop
export YARN_CONF_DIR=$HADOOP_HOME/etc/hadoop
APP_JAR=$BOOK_HOME/dist/data_algorithms_book.jar
LIB=$BOOK_HOME/lib
EXTRA_JARS=$LIB/commons-math-2.2.jar,$LIB/commons-math3-3.0.jar
TIMETABLE=$BOOK_HOME/data/timetable.txt
BIOSETS=$DAB_HOME/data/biosets.txt
$SPARK_HOME/bin/spark-submit
    --class org.dataalgorithms.chap22.spark.SparkTtest \
    --master yarn-cluster \
    --num-executors 12 \
    --driver-memory 3g \
    --executor-memory 7g \
    --executor-cores 12 \
    --jars $EXTRA_JARS \
    $APP_JAR $TIMETABLE $BIOSETS
```

## 示例运行日志

下面是运行以上脚本得到的示例日志输出：

```
# ./run_ttest.sh
<timetable-file>=/home/mp/data-algorithms-book/ttestinput/timetable.txt
<bioset-file>=/home/mp/data-algorithms-book/ttestinput/biosets.txt
createTimeTable() map={b1=0.9, b3=0.5, b2=0.75, b4=1.1}
...
```

## 生成的调试输出

下面是脚本的调试输出：

```
# hadoop fs -cat /ttest/output/1/part*
G1,b1,1.0
G2,b1,0.6
G3,b1,0.09
G4,b1,2.2
G5,b1,1.03
G1,b4,2.9
G2,b4,1.8
```

```
G4,b4,1.05
G1,b2,2.09
G2,b2,1.07
G3,b2,1.00
G2,b3,2.9
G3,b3,1.03
G5,b3,2.9

# hadoop fs -cat /ttest/output/2/part*
(G1,b1)
(G2,b1)
(G3,b1)
(G4,b1)
(G5,b1)
(G1,b4)
(G2,b4)
(G4,b4)
(G1,b2)
(G2,b2)
(G3,b2)
(G2,b3)
(G3,b3)
(G5,b3)

# hadoop fs -cat /ttest/output/3/part*
(G3,[b3, b1, b2])
(G4,[b1, b4])
(G5,[b1, b3])
(G1,[b1, b4, b2])
(G2,[b1, b4, b2, b3])
```

## 生成的最终输出

下面是这个程序的最终输出：

```
# hadoop fs -cat /ttest/output/4/part*
(G3,0.08146847659449757)
(G4,0.14994028688738084)
(G5,0.48769401782708255)
(G1,0.05441315690970782)
(G2,0.0)
```

这一章为使用bioset的T检验提供了可伸缩的MapReduce/Hadoop和Spark解决方案（如果愿意，也可以使用不同类型的数据，不过通常都会在使用bioset的临床试验领域完成T检验）。我们已经了解到，T检验可以检查两个组的均值在统计意义上是否不同。

# 第23章

# 皮尔逊相关系数

## 概述

皮尔逊（Pearson）[注1]相关系数可以度量两个数据集的相关关系（线性关系）。这是数学和统计学中最常用的相关度量方法。基本来说，皮尔逊相关系数回答了这样一个问题：能不能画一个折线图表示数据？根据onlinestatbook（*http://bit.ly/pearson-correlation*）的描述：

> 皮尔逊积差相关系数（Pearson product-moment correlation coefficient）是对两个变量间线性关系强度的一个度量。这称为皮尔逊相关系数（Pearson's correlation）或者简称为相关系数（correlation coefficient）。如果变量间的关系不是线性的，那么这个相关系数不能充分表示变量间关系的强度。

这一章将提供两个计算皮尔逊相关系数的MapReduce解决方案：

- 使用传统MapReduce/Hadoop的简单解决方案。
- 实现全相关（all vs. all correlation）的Spark解决方案（稍后会给出全相关的定义）

可以很容易地将皮尔逊相关系数算法调整为计算斯皮尔曼等级相关系数（Spearman ranked correlations）（*http://bit.ly/spearman_coeff*）。为了完成斯皮尔曼等级相关分析，我提供了一个Java包装器类`Spearman.java`。在这一章的最后，你就能将皮尔逊相关系数算法替换为斯皮尔曼等级相关系数算法。

---

注1：Karl Pearson（1857－1936）是英国数学家，也是数理统计理论的奠基人。

# 皮尔逊相关系数公式

皮尔逊相关系数的计算公式有很多不同的等价形式。令$x = (x_1, x_2, \ldots, x_n)$，$y = (y_1, y_2, \ldots, y_n)$，$x$和$y$的皮尔逊相关系数可以表述为：

$$r = \frac{\sum (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum (x_i - \bar{x})^2 \sum (y_i - \bar{y})^2}}$$

$$r = \frac{\sum xy - \frac{\sum x \sum y}{n}}{\sqrt{\left(\sum x^2 - \frac{\left(\sum x\right)^2}{n}\right)\left(\sum y^2 - \frac{\left(\sum y\right)^2}{n}\right)}}$$

其中：

$$\bar{x} = \frac{\sum x}{n}$$

$$\bar{y} = \frac{\sum y}{n}$$

皮尔逊相关系数有以下性质：

- 范围为 $-1.00 \leqslant r \leqslant 1.00$。
- 相关系数是两个变量间关联强度的一个无量纲指标：
  - $r > 0$ 表示正关联。
  - $r < 0$ 表示负关联。
  - $r = 0$ 表示没有关联。
- 度量$x$和$y$之间的线性关系。

给定两个数据集，皮尔逊相关系数可能取哪些值？其结果将落在$-1.00 \sim 1.00$之间。这些值可以如下分类：

- 高/正相关：0.6～1.0或−0.6～1.0 （见图23-1）。
- 中相关：0.3～0.6或−0.3～0.6。
- 低/负相关：0.1～0.3或−0.1～0.3 （见图23-2）。
- 无相关：接近0.00（见图23-3）。

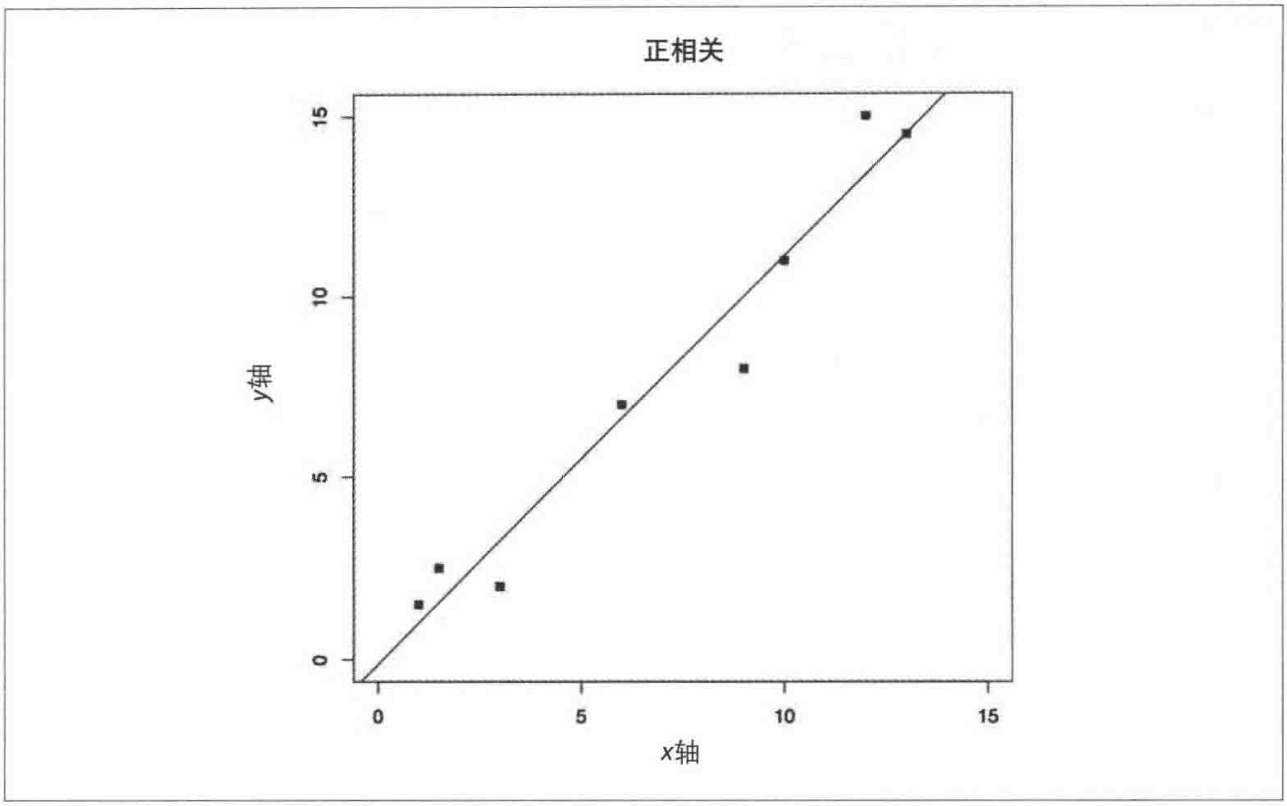



图23-1：正相关

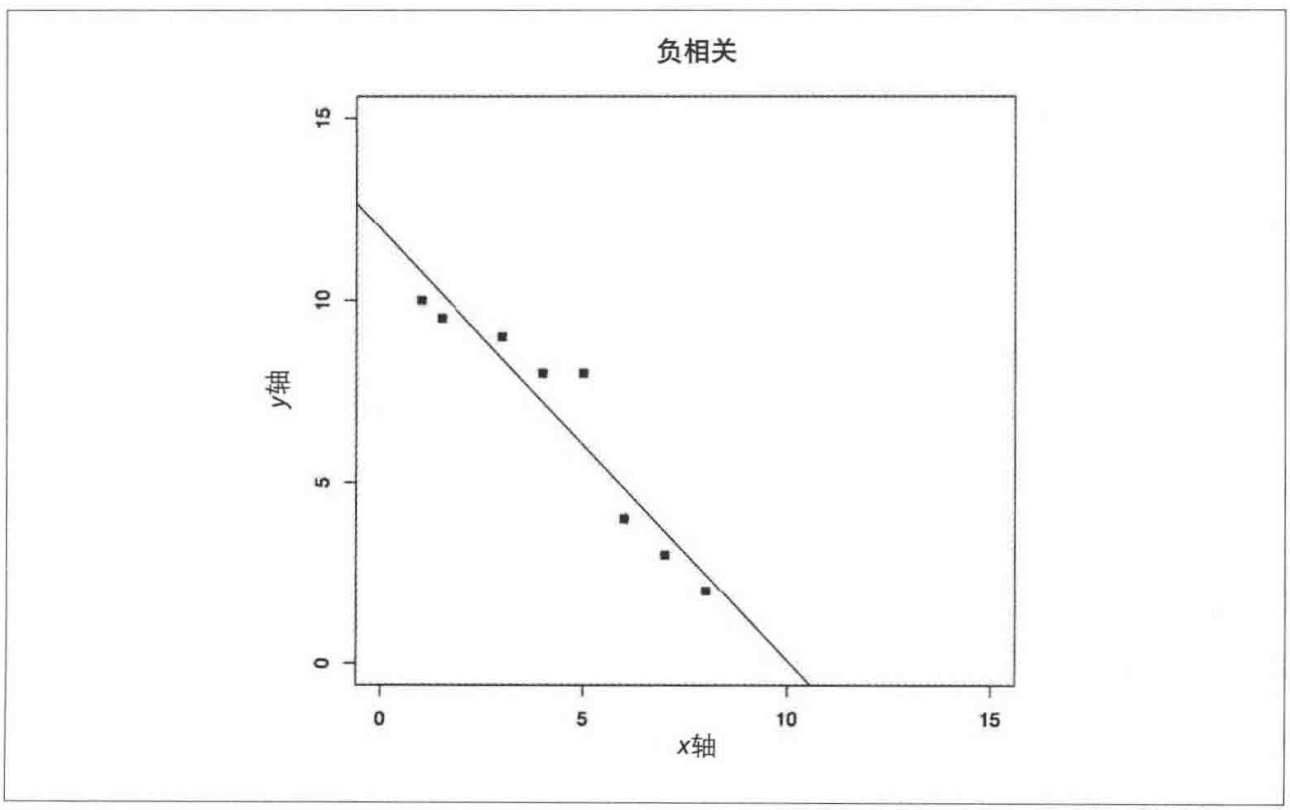



图23-2：负相关

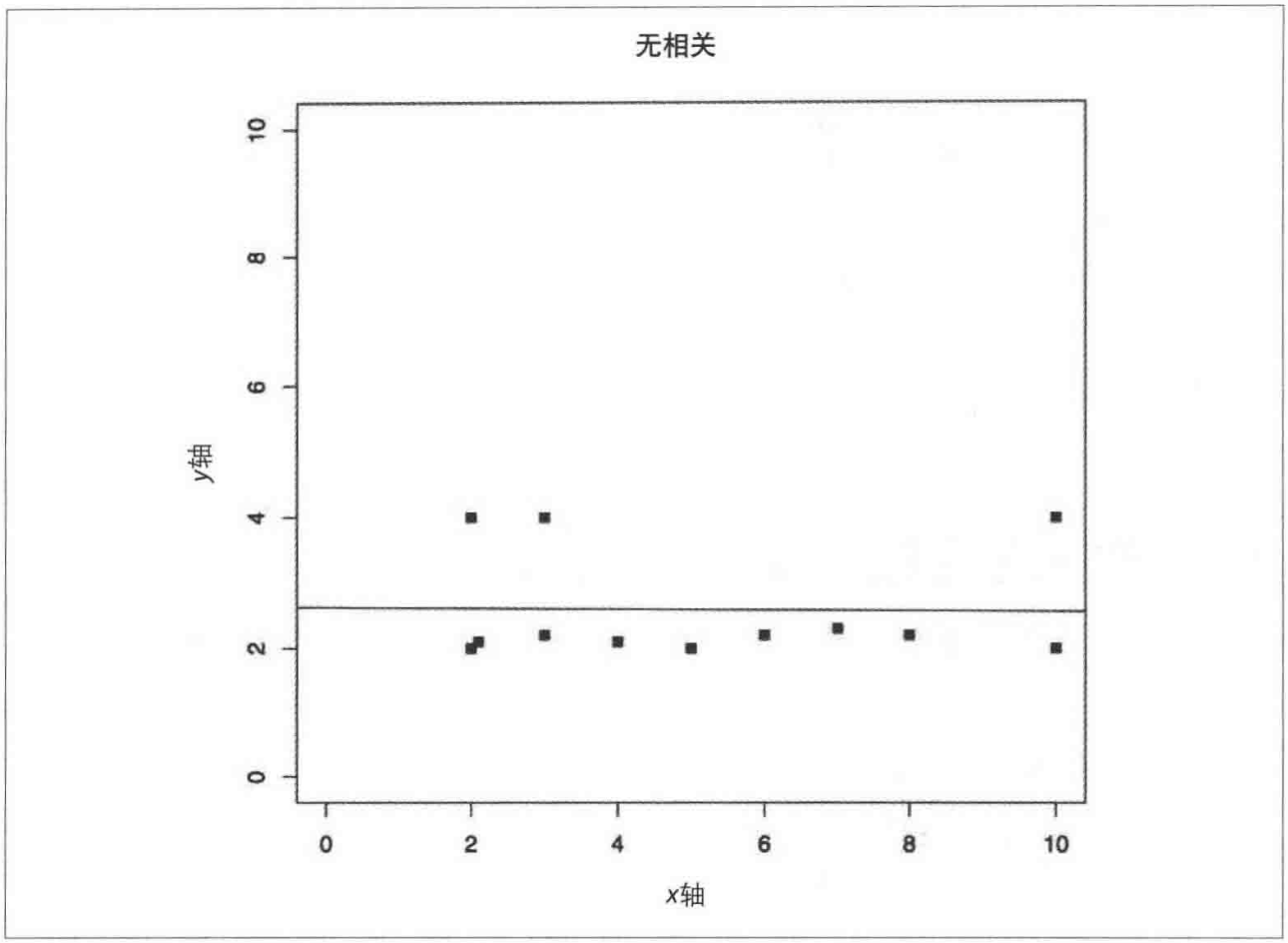



图23-3：无相关

# 皮尔逊相关系数示例

表23-1显示了7位候选人在大学里学习的年数($x$)和以后的年收入($y$)。这里收入单位为千美元，不过这一点对我们的计算并没有任何影响。

**表23-1：候选人在大学里学习的年数和年收入**

| 候选人 | x | y | $x^2$ | $y^2$ | xy |
|---|---|---|---|---|---|
| Alex | 0 | 15 | 0 | 225 | 0 |
| Mary | 1 | 18 | 0 | 324 | 18 |
| Jane | 3 | 20 | 9 | 400 | 60 |
| John | 4 | 25 | 16 | 625 | 100 |
| Rafa | 4 | 30 | 16 | 900 | 120 |
| Roger | 6 | 35 | 36 | 1225 | 210 |
| Ken | 7 | 40 | 36 | 1600 | 280 |
| | $\Sigma x = 25$ | $\Sigma y = 183$ | $\Sigma x^2 = 127$ | $\Sigma y^2 = 5299$ | $\Sigma xy = 788$ |

将这些值插入皮尔逊相关系数公式，可以得到$r = 0.95$，这是一个很高的相关系数。

# 皮尔逊相关系数数据集

我们想要处理一个通用数据集（这是一个矩阵，采用CSV格式），其中列对应变量（如$x$, $y$, $z$），行对应实例。示例数据可能如下所示：

```
ROW-1: 1, 1, 3, -1
ROW-2: 2, 2, 1, -2
ROW-3: 3, 3, 8, -3
...
```

# 皮尔逊相关系数POJO 解决方案

令data是一个二维double数组；这样就可以计算成对相关系数，如示例23-1所示。

示例23-1：计算成对相关系数

```
public void computeAllPairwiseCorrelations(double[][] data) {
   int numColumns = data[0].length;
   for(int i=0; i < numColumns; i++) {
      for(int j=i+1; j < numColumns; j++) {
         // 计算第i个和第j个列/变量之间的相关系数
         double correlation = computeCorrelation(i, j, data);
         System.out.println("i="+i+" j="+j+" correlation="+correlation);
      }
   }
}
```

计算成对皮尔逊相关系数的Java方法如示例23-2所示。

示例23-2：计算皮尔逊相关系数

```
public double computePearsonCorrelation(int i, int j, double[][] data) {
   double x = 0;
   double y = 0;
   double xx = 0;
   double yy = 0;
   double xy = 0;
   double n = data.length;
   for(int row=0; row < data.length; row++) {
      x += data[row][i];
      y += data[row][j];
      xx += Math.pow(data[row][i], 2.0d);
      yy += Math.pow(data[row][j], 2.0d);
      xy += data[row][i] * data[row][j];
   }
   double numerator = xy - ((x * y) / n);
   double denominator1 = xx - (Math.pow(x, 2.0d) / n);
   double denominator2 = yy - (Math.pow(y, 2.0d) / n);
   double denominator = Math.sqrt(xx * yy);
```

```
19    double correlation = numerator / denominator;
20    return correlation;
21 }
```

## POJO解决方案测试

这里使用一个小例子测试皮尔逊相关系数，如示例23-3所示。

示例23-3：皮尔逊相关系数测试

```
1 public class PearsonCorrelation {
2    public void computeAllPairwiseCorrelations(double[][] data) {...}
3    public double computeCorrelation(int i, int j, double[][] data) {...}
4
5    public static void main(String[] args) throws Exception {
6       double[][] matrix = new double[][] {
7          {1, 1, 3, -1},
8          {2, 2, 1, -2},
9          {3, 3, 8, -3}
10      };
11      PearsonCorrelation pc = new PearsonCorrelation();
12      pc.computeAllPairwiseCorrelations(matrix);
13      System.exit(0);
14   }
15 }
```

下面是运行这个示例的输出：

```
$ javac PearsonCorrelation.java
$ java PearsonCorrelation
i=0 j=1 correlation=0.14285714285714285
i=0 j=2 correlation=0.15534244150030002
i=0 j=3 correlation=-0.14285714285714285
i=1 j=2 correlation=0.15534244150030002
i=1 j=3 correlation=-0.14285714285714285
i=2 j=3 correlation=-0.15534244150030002
```

从这个输出可以得出结论，我们的MapReduce解决方案必须生成6个唯一的归约器键。

# 皮尔逊相关系数MapReduce解决方案

我们的MapReduce解决方案中主要考虑map()和reduce()函数。

## 皮尔逊相关系数的map()函数

示例23-4定义了皮尔逊相关系数的映射器。

示例23-4：皮尔逊相关系数：map()函数

```
1 /**
2  * key 由MapReduce生成，在这里忽略
3  * value 是矩阵中的一行
4  */
5 map(key, value) {
6   double[] arr = line.split(",");
7   int size = arr.length;
8   for(int i=0; i < size -1; i++) {
9      for(int j=i+1; j < size; j++) {
10        reducerKey = PairOfLongs(i, j);
11        reducerValue = PairOfDoubles(arr[i], arr[j]);
12        emit(reducerKey, reducerValue);
13     }
14   }
15 }
```

现在来看每个map()函数会生成什么结果，map(ROW-1)会生成：

- K2=(0,1)和V2=(1,1)
- K2=(0,2)和V2=(1,3)
- K2=(0,3)和V2=(1,-1)
- K2=(1,2)和V2=(1,3)
- K2=(1,3)和V2=(1,-1)
- K2=(2,3)和V2=(3,-1)

map(ROW-2)将生成：

- K2=(0,1)和V2=(2,2)
- K2=(0,2)和V2=(2,1)
- K2=(0,3)和V2=(2,-2)
- K2=(1,2)和V2=(2,1)
- K2=(1,3)和V2=(1,-2)
- K2=(2,3)和V2=(1,-2)

map(ROW-3)将生成：

- K2=(0,1)和V2=(3,3)
- K2=(0,2)和V2=(3,8)
- K2=(0,3)和V2=(3,-3)
- K2=(1,2)和V2=(3,8)

- K2=(1,3)和V2=(3,-3)
- K2=(2,3)和V2=(8,-3)

## 皮尔逊相关系数的reduce()函数

前面已经提到，要为归约器生成6个唯一的键：

- K2=(0,1)和List_of_V2=[(1,1), (2,2), (3,3)]
- K2=(0,2)和List_of_V2=[(1,3), (2,1), (3,8)]
- K2=(0,3)和List_of_V2=[(1,-1), (2,-2), (3,-3)]
- K2=(1,2)和List_of_V2=[(1,3), (2,1), (3,8)]
- K2=(1,3)和List_of_V2=[(1,-1), (1,-2), (3,-3)]
- K2=(2,3)和List_of_V2=[(3,-1), (1,-2), (8,-3)]

示例23-5定义了皮尔逊相关系数的归约器。

示例23-5：皮尔逊相关系数：reduce()函数

```
1 reduce(PairOfLongs key, Iterable<PairOfDoubles> values) {
2    double x = 0.0d;
3    double y = 0.0d;
4    double xx = 0.0d;
5    double yy = 0.0d;
6    double xy = 0.0d;
7    double n = 0.0d;
8
9    for(PairOfDoubles pair : values) {
10      x += pair.getLeftElement();
11      y += pair.getRightElement();
12      xx += Math.pow(pair.getLeftElement(), 2.0d);
13      yy += Math.pow(pair.getRightElement(), 2.0d);
14      xy += (pair.getLeftElement() * pair.getRightElement());
15      n += 1.0d;
16   }
17
18   PearsonComputation pearson = new PearsonComputation(x, y, xx, yy, xy, n);
19   emit(key, pearson);
20 }
```

## Hadoop实现类

在这个Hadoop实现中，我们提供了一个驱动器、定义map()和reduce()函数的类，以及一些定制数据结构和类（如表23-2所示），这些数据结构和类用来保存计算皮尔逊相关系数的中间值。

表23-2：皮尔逊相关系数Hadoop实现的类和数据结构

| 类名 | 描述 |
| --- | --- |
| PearsonCorrelationDriver | 提交Hadoop作业的驱动器程序 |
| PearsonCorrelationMapper | 定义map() |
| PearsonCorrelationReducer | 定义reduce() |
| PearsonCorrelation | 实现皮尔逊相关系数算法 |
| HadoopUtil | 定义一些工具函数 |
| TestDataGeneration | 生成皮尔逊相关系数的测试数据 |
| PairOfWritables<L,R> | edu.umd.cloud9.io.pair.PairOfWritables<L,R> |

在这个Hadoop实现中，我们使用了一个PairOfWritables<L,R>类，可以实例化这个类来生成PairOfWritables<Long,Long>和PairOfWritables<Double,Double>。PairOfWritables（这个类表示一对Writable）的定义参见示例23-6。

示例23-6：PairOfWritables类

```
1 package edu.umd.cloud9.io.pair;
2
3 import java.io.DataInput;
4 import java.io.DataOutput;
5 import java.io.IOException;
6 import org.apache.hadoop.io.Writable;
7
8 public class PairOfWritables<L extends Writable, R extends Writable>
9    implements Writable {
10
11   private L leftElement;
12   private R rightElement;
13   ...
14 }
```

这里使用PairOfWritables.getLeftElement()和PairOfWritables.getRightElement()分别获取leftElement和rightElement。

# 皮尔逊相关系数的Spark 解决方案

这一节中我们将计算全相关（all vs. all correlation）。这是什么意思？下面先定义一个特定的问题用例，然后我会解释什么是全相关。需要说明，针对你的特定输入数据（对于不同的应用领域，输入数据的格式肯定会有所不同），可以很容易地调整这里给出的算法。令$P = \{P_1, P_2, ..., P_n\}$是一个病人集合（每个病人由一个Patient-ID标识）。令每个病人分别有一个生物标志物数据的有限集合，格式如下：

```
<Gene-ID><,><Reference><,><Patient-ID><,><Gene-Value-As-Double-Data-Type>
```

在这里：

```
Reference = {
                    r1, // 正常
                    r2, // 患病
                    r3, // 成对
                    r4 // 未知
}
```

为了得到正确的相关结果，我们只使用一个“reference”类型（用作为过滤器）。例如，下面给出几个样本生物标志物数据行：

```
G1234,r3,P100,0.04
G1345,r1,P200,0.90
G2155,r2,P200,0.86
```

令$G = \{G_1, G_2, ..., G_m\}$是一个基因集（表示为Gene-ID）。例如，对于RNA基因表达式数据类型，唯一基因个数可能约为40000。因此，全相关将生成：

$$40000 \times 40000=1600000000$$

个皮尔逊相关数据点。不过，由于（$G_i$,$G_j$）的相关系数与（$G_j$, $G_i$）的相关系数相同，这个计算可以简化如下（$N = 40000$）：

$$\frac{N\times(N-1)}{2}$$

因此，如果$G = \{G_1, G_2, G_3, G_4, G_5\}$，就要为以下基因对生成皮尔逊相关系数：

$(G_1, G_2)$

$(G_1, G_3)$

$(G_1, G_4)$

$(G_1, G_5)$

$(G_2, G_3)$

$(G_2, G_4)$

$(G_2, G_5)$

$(G_3, G_4)$

$(G_3, G_5)$

$(G_4, G_5)$

要生成正确的基因对计算相关系数，需要确保：

$G_i < G_j$

这可以保证不会同时生成（$G_i$, $G_j$）和（$G_j$, $G_i$）。

## 输入

我们将从HDFS读取生物标志物数据。假设HDFS中有以下数据（生物标志物数据采用文本格式，由文本文件{b1, b2,b3, ...}标识）：

```
/biomarker/input/b1
/biomarker/input/b2
/biomarker/input/b3
...
```

这些生物标志物文件中的各个记录有以下格式：

```
<Gene-ID><,><Reference><,><Patient-ID><,><Gene-Value-As-Double-Data-Type>
```

## 输出

令$G = \{G_1, G_2, ..., G_m\}$ 是一个基因ID集合。目标是为每个基因对（$G_i$, $G_j$）生成以下输出记录集：

($G_i$, $G_j$), (pearson-correlation, pValue)

其中：

- $G_i < G_j$
- $G_i \in \{G_1, G_2, ..., G_m\}$
- $G_j \in \{G_1, G_2, ..., G_m\}$
- $0.00 \leq$ pValue $\leq 1.00$
- $1.00 \leq$ pearson-correlation $\leq +1.00$

## Spark 解决方案

这里用一个Java驱动器类提供我们的Spark解决方案，其中使用了丰富的Spark API（包括filter()和cartesian()等函数）来找出所有基因对的相关系数。需要说明，尽管这里是为所有基因对提供的Spark解决方案，但是完全可以对任何类型的数据应用这个技术。这个Spark程序的输入是一组生物标志物文件和一个reference（取值为{$r_1$, $r_2$, $r_3$, $r_4$}中的某个值）。所有生物标志物输入都从HDFS读入一个JavaRDD，然后使用以下Spark方法分区：

```
public JavaRDD<T> coalesce(int numberOfPartitions)
// 描述：返回一个新的RDD
// 将归约为numberOfPartitions个分区。
```

创建和划分JavaRDD之后，按reference值过滤所有生物标志物。例如，如果reference = r2，那么所有reference值不为r2的记录都将被剔除。要用JavaRDD.filter()函数完成这个过滤。接下来使用JavaRDD.mapToPair()函数映射所有通过过滤的记录，这会生成以下JavaPairRDD<K,V>：

- K = Gene-ID（作为一个String）
- V = Tuple2<String, Double>(patientID, geneValue)

下一步是按Gene-ID对所有生物标志物数据分组，这会生成以下JavaPairRDD（称为grouped）：

```
JavaPairRDD<String, Iterable<Tuple2<String,Double>>>
```

其中：

- K = Gene-ID
- V = Iterable<Tuple2(patientID, geneValue)>

要完成所有基因对的相关分析，必须计算grouped与grouped的笛卡儿积；这会生成所有可能的（$G_i$, $G_j$）组合。令这个笛卡儿积的结果名为cart。完成其他相关计算之前，首先要过滤这个笛卡儿积结果（cart），只保留$G_i < G_j$的对。这个过滤由以下函数完成：

```
cart.filter(...);
```

过滤笛卡儿积之后，接下来准备计算所有可能的（$G_i$, $G_j$）组合的皮尔逊相关系数和相应的p值。为了计算（$G_i$, $G_j$）的皮尔逊相关系数，要建立表23-3所示的矩阵。

表23-3：计算($G_i$,$G_j$)皮尔逊相关系数的矩阵

| | Patient-ID1 | Patient-ID2 | Patient-ID3 | ... |
|---|---|---|---|---|
| $G_i$ | avg(values) | avg(values) | avg(values) | ... |
| $G_j$ | avg(values) | avg(values) | avg(values) | ... |

下面使用Spark API提供一个完整的解决方案。首先描述高层步骤，给出这个Spark程序的完整结构，然后详细讨论各个步骤。

## 高层步骤

Spark解决方案的工作流如示例23-7所示（另外参见图23-4）。

示例23-7：Spark解决方案：高层步骤

```
// 步骤1：导入所需的类和接口
/**
 * 全相关是什么意思？
 *
 * 令选择的基因为：G = (g1, g2, g3, g4).
 * 全相关将计算以下基因对之间的相关系数：
 *
 * (g1, g2)
 * (g1, g3)
 * (g1, g4)
 * (g2, g3)
 * (g2, g4)
 * (g3, g4)
 *
 * 注意当且仅当（Ga < Gb）时生成（Ga, Gb）对。
 * （Ga, Gb）和（Gb, Ga）的相关系数相同，所以
 * 没有必要计算重复基因的相关系数。
 *
 * 生物标志物记录示例：
 * 格式：  <geneID><,><r{1,2,3,4}><,><patientID><,><geneValue>
 * 示例：  37761,r2,p10001,1.287
 *
 * @author Mahmoud Parsian
 *
 */
public class AllVersusAllCorrelation implements java.io.Serializable {

   static boolean smaller(String g1, String g2) {...}
   static class MutableDouble implements java.io.Serializable {...}
   static Map<String, MutableDouble> toMap(List<Tuple2<String,Double>> list)
       {...}
   static List<String> toListOfString(Path hdfsFile) throws Exception {...}
   static JavaRDD<String> readBiosets(JavaSparkContext ctx, List<String>
                                      biosets) {...}

   public static void main(String[] args) throws Exception {
     // 步骤2：处理输入参数
     // 步骤3：创建Spark上下文对象（ctx）
     // 步骤4：创建输入文件/生物标志物列表
     // 步骤5：作为全局共享对象广播参考值，
     //        可以由所有集群节点访问
     // 步骤6：从HDFS读取所有生物标志物并创建第一个RDD
     // 步骤7：按参考值过滤生物标志物
     // 步骤8：创建（Gene-ID, (Patient-ID, Gene-Value)）对
     // 步骤9：按Gene-ID对生物标志物分组
     // 步骤10：创建所有基因的笛卡儿积
     // 步骤11：过滤冗余基因对
     // 步骤12：皮尔逊相关系数和p值

     // 完成
     ctx.close();
     System.exit(0);
   }
}
```

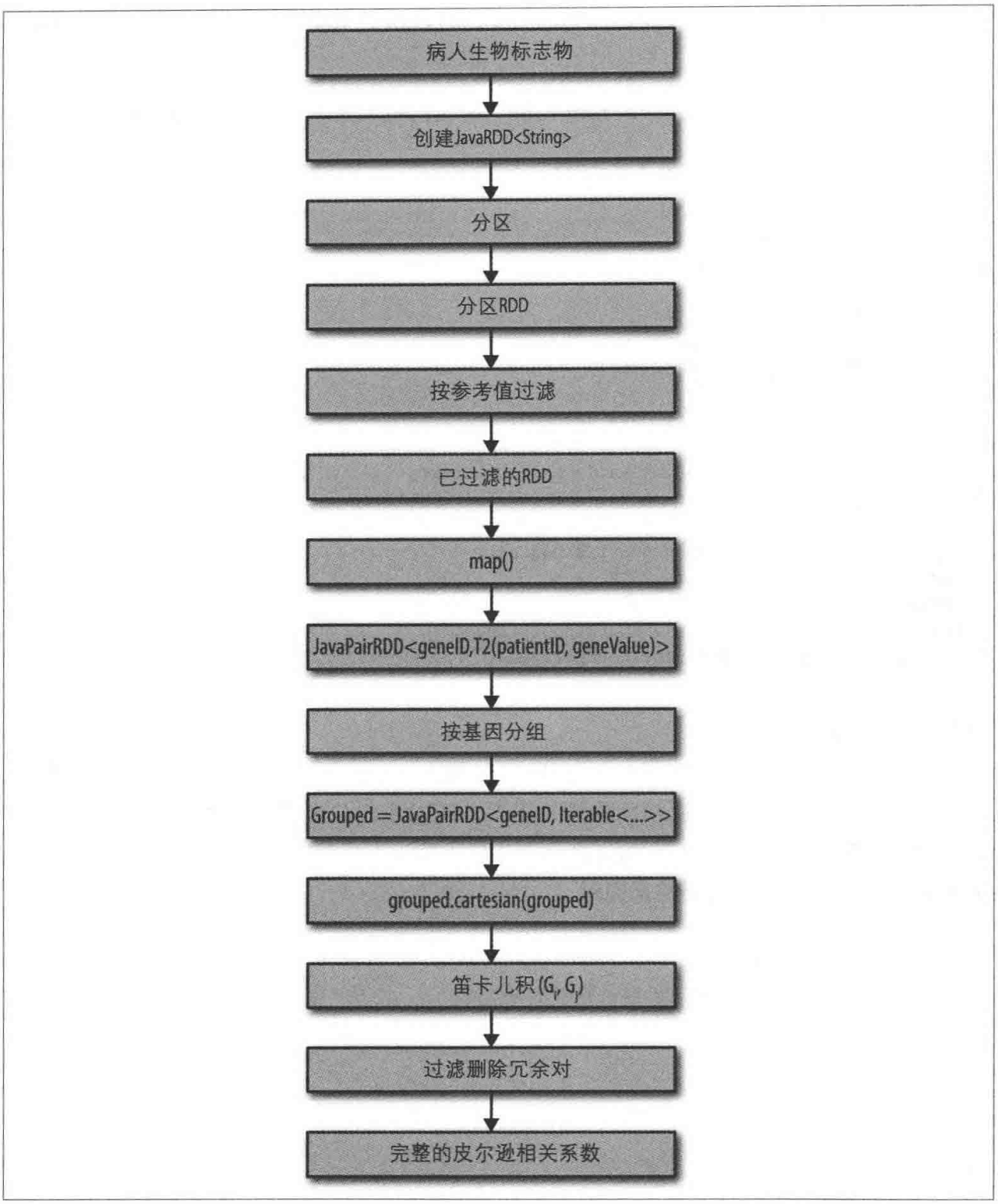



图23-4：所有基因对的皮尔逊相关系数

## 步骤1：导入所需的类和接口

这个步骤如示例23-8所示，要导入这个解决方案所需的类和接口，它们包含在以下Spark包中：

- org.apache.spark.api.java（包含JavaPairRDD、JavaRDD和JavaSparkContext类）。
- org.apache.spark.api.java.function（包含Function和PairFunction接口）。

示例23-8：步骤1：导入所需的类和接口

```
// 步骤1：导入所需的类和接口
import java.util.Map;
import java.util.HashMap;
import java.util.List;
import java.util.ArrayList;
import java.io.BufferedReader;
import java.io.InputStreamReader;
import scala.Tuple2;
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.api.java.JavaPairRDD;
import org.apache.spark.api.java.JavaSparkContext;
import org.apache.spark.api.java.function.PairFunction;
import org.apache.spark.api.java.function.Function;
import org.apache.spark.broadcast.Broadcast;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.FSDataInputStream;
import org.apache.hadoop.conf.Configuration;
```

## smaller()方法

示例23-9中定义了smaller()方法，利用这个方法可以避免生成重复的基因对。基因对（$g_1$, $g_2$）和（$g_2$, $g_1$）会生成相同的相关系数，所以我们只生成一个基因对（$g_1$, $g_2$）的相关系数，其中$g_1$小于$g_2$。（利用cartesian()函数）生成基因的笛卡儿积之后，将使用smaller()方法过滤掉不必要的基因对。

示例23-9：smaller()方法

```
static boolean smaller(String g1, String g2) {
    if (g1.compareTo(g2) < 0) {
        return true;
    }
    else {
        return false;
    }
}
```

## MutableDouble类

MutableDouble类的定义见示例23-10，这个类用来存储一个病人的所有生物标志物值。这个类可以对double值求和和计数，而且提供了一个方便的avg()方法。MutableDouble类包含一个基因和一个病人的所有值。

示例23-10：MutableDouble类

```
static class MutableDouble implements java.io.Serializable {
    private double value = 0.0;
    private double count = 0.0;

    public MutableDouble(double d) {
        value = d;
        count = 1.0;
    }
    public MutableDouble(Double d) {
        if (d == null) {
           value = 0.0;
        }
        else {
           value = d;
        }
        count = 1.0;
    }

    public void increment(Double d) {
        if (d == null) {
           // 值不变
        }
        else {
           value += d;
        }
        count++;
    }

    public void increment(double d) {
        count++;
        value += d;
    }

    public double avg() {
        return value/count;
    }
}
```

## toMap()方法

toMap()方法的定义见示例23-11，它将一个Tuple2<PatientID, GeneValue>项列表转换为一个Map<PatientID, MutableDouble>（也就是说，将对应一个基因聚集所有病人的值）。

示例23-11：toMap()方法

```
static Map<String, MutableDouble> toMap(Iterable<Tuple2<String,Double>> list) {
    Map<String, MutableDouble> map = new HashMap<String, MutableDouble>();
    for (Tuple2<String,Double> entry : list) {
        MutableDouble md = map.get(entry._1);
        if (md == null) {
            map.put(entry._1, new MutableDouble(entry._2));
        }
```

```
 8         else {
 9             md.increment(entry._2);
10         }
11     }
12     return map;
13 }
```

### toListOfString()方法

toListOfString()方法要创建一个List<String>，其中每个元素分别是一个HDFS文件，表示一个病人的一个生物标志物文件（每个病人可能有多个生物标志物文件）。这个方法的输入是一个HDFS文件，其中包含计算皮尔逊相关系数涉及的所有生物标志物文件。为了帮助你理解这个方法，我提供了以下演示示例：

```
# hadoop fs -cat /biomarkers/biomarkers.txt
/biomarker/input/b1
/biomarker/input/b2
/biomarker/input/b3
/biomarker/input/b4
/biomarker/input/b5
/biomarker/input/b6
/biomarker/input/b7
/biomarker/input/b8
/biomarker/input/b9
/biomarker/input/b10
/biomarker/input/b11
/biomarker/input/b12
/biomarker/input/b13
/biomarker/input/b14
/biomarker/input/b15

# hadoop fs -cat /biomarker/input/b1
g1,r2,p1,1.86
g2,r2,p1,0.74
g3,r2,p1,1.24
...

# hadoop fs -cat /biomarker/input/b2
g1,r2,p2,2.46
g2,r2,p2,3.24
g3,r1,p2,1.44
...
```

这个例子的结果是一个List<String>，包含以下String：

```
/biomarker/input/b1
/biomarker/input/b2
...
/biomarker/input/b14
/biomarker/input/b15
```

这个方法的定义见示例23-12。

示例23-12：toListOfString()方法

```
static List<String> toListOfString(Path hdfsFile) throws Exception {
    FSDataInputStream fis = null;
    BufferedReader br = null;
    FileSystem fs = FileSystem.get(new Configuration());
    List<String> list = new ArrayList<String>();
    try {
        fis = fs.open(hdfsFile);
        br = new BufferedReader(new InputStreamReader(fis));
        String line = null;
        while ((line = br.readLine()) != null) {
            String value = line.trim();
            list.add(value);
        }
    }
    finally {
       if (br != null) {
           br.close();
       }
    }
    return list;
}
```

## readBiosets()方法

readBiosets()方法的定义见示例23-13，它会读取所有生物标志物（HDFS中的文件），生成第一个RDD做进一步处理。每个RDD元素是生物标志物文件的一个记录。

示例23-13：readBiosets()方法

```
static JavaRDD<String> readBiosets(JavaSparkContext ctx,
                                   List<String> biosets) {
    int size = biosets.size();
    int counter = 0;
    StringBuilder paths = new StringBuilder();
    for (String biosetFile : biosets) {
        counter++;
        paths.append(biosetFile);
        if (counter < size) {
            paths.append(",");
        }
    }
    JavaRDD<String> rdd = ctx.textFile(paths.toString());
    return rdd;
}
```

可以使用Spark API中的coalesce()方法对这个RDD进一步分区，如下所示：

```
public JavaRDD<T> coalesce(int numPartitions)
// 描述：返回一个新RDD，它会归约为
// numPartitions个分区。
```

现在的问题是如何为RDD选择适当的分区数。答案取决于集群节点数、每个服务器的内

核数以及RAM大小。要得出适当的RDD分区数，并没有一个“银弹”公式（也就是说，你要做一些尝试，经历一些失败，才能最后找出和设置适合当前环境的分区数）。

现在，利用以上方法定义，我们可以回过头来继续讨论示例23-7给出的高层解决方案中的各个步骤。

## 步骤2：处理输入参数

这个步骤如示例23-14所示，它会读取两个参数：

- 一个参考值，这是{r1, r2, r3, r4}中的一个值。
- 一个HDFS文件，其中包含计算皮尔逊相关系数所需的所有生物标志物文件的一个列表（这些生物标志物文件都持久存储为HDFS文件）。

示例23-14：步骤2: 处理输入参数

```
// 步骤2: 处理输入参数
if (args.length < 2) {
    System.err.println(
        "Usage: AllVersusAllCorrelation <reference> <all-bioset-ids-as-filename>");
    System.err.println(
        "Usage: OneVersusAllCorrelation r2 <all-bioset-ids-as-filename>");
    System.exit(1);
}
final String reference = args[1]; // {"r1", "r2", "r3", "r4"}
final String biomarkersFileName = args[2];
```

## 步骤3：创建一个Spark上下文对象

这个步骤会创建一个JavaSparkContext对象。可以采用很多不同方法创建JavaSparkContext的一个实例，例如，SparkUtil类提供了一些便利方法来完成这个工作。SparkUtil的方法如示例23-15所示，Spark上下文对象的创建见示例23-16。

示例23-15：创建JavaSparkContext的SparkUtil方法

```
public class SparkUtil {
   /**
    * 创建一个JavaSparkContext对象
    *
    * @param appName是一个应用名
    * @return 一个JavaSparkContext
    *
    */
   public static JavaSparkContext createJavaSparkContext(String appName)
      throws Exception {...}

   /**
    * 由给定的一个Spark主URL创建一个JavaSparkContext对象
    *
    * @param sparkMasterURL Spark主URL
```

```
16      * "spark://<spark-master-host-name>:7077"
17      * @param description 程序描述
18      * @return 一个JavaSparkContext
19      *
20      */
21     public static JavaSparkContext
22        createJavaSparkContext(String sparkMasterURL, String description)
23        throws Exception {...}
24 }
```

示例23-16：步骤3: 创建一个Spark上下文对象

```
1    // 步骤3: 创建Spark上下文对象
2    JavaSparkContext ctx = SparkUtil.createJavaSparkContext("pearson-correlation");
```

## 步骤4：创建输入文件/生物标志物列表

这个步骤如示例23-17所示，这里会读取biomarkersFileName标识的文件，这是一个文本HDFS文件，其中包含完成皮尔逊相关系数计算所需的所有输入生物标志物文件（每一行是一个生物标志物文件）。

示例23-17：步骤4: 创建输入文件/生物标志物列表

```
1    步骤4: 创建输入文件/生物标志物列表
2    List<String> list = toListOfString(new Path(biomarkersFileName));
```

## 步骤5：作为全局共享对象广播参考值

由于reference值({r1, r2, r3, r4}) 要用来过滤RDD元素，应当将这个值广播到所有集群节点。在Spark中，要广播一个数据结构（作为一个只读共享对象），可以使用Broadcast类，如下所示。

- 要广播一个类型为T的共享数据结构：

```
T t = <create-object-of-type-T>;
final Broadcast<T> broadcastT = ctx.broadcast(t);
```

- 要读取/访问一个类型为T的共享数据结构：

```
T t = broadcastT.value();
```

在MapReduce/Hadoop中，可以使用Hadoop的Configuration对象将一个共享数据结构广播到map()或reduce()函数。可以使用Configuration.set(...)广播，使用Configuration.get(...)来读取/访问所广播的对象。Spark的API比Hadoop的API丰富得多，因为Spark API允许广播任意类型的数据结构。这一步的实现如示例23-18所示。

示例23-18：步骤5: 作为全局共享对象广播参考值

```
1    // 步骤5: 作为全局共享对象广播参考值
2    // 可以从所有集群节点访问
```

```
3    final Broadcast<String> REF = ctx.broadcast(reference); // "r2"
```

## 步骤6：从HDFS读取所有生物标志物并创建第一个RDD

这个步骤如示例23-19所示，将读取所有生物标志物文件，并创建一个JavaRDD<String>，可以利用以下方法进一步对它分区：

```
JavaRDD<T> coalesce(int numPartitions)
```

示例23-19：步骤6: 从HDFS读取所有生物标志物文件，并创建第一个RDD

```
1 // 步骤6: 从HDFS读取所有生物标志物并创建第一个RDD
2 JavaRDD<String> biosets = readBiosets(ctx, list);
3 biosets.saveAsTextFile("/output/1");
```

这里给出这一步的示例输出来进行调试：

```
# hadoop fs -cat /output/1/*
g1,r2,p1,1.86
g2,r2,p1,0.74
g3,r2,p1,1.24
g4,r1,p1,2.44
g5,r2,p1,1.69
g6,r2,p1,0.93
g7,r2,p1,1.44
g8,r2,p1,2.11
g1,r2,p2,2.46
g2,r2,p2,3.24
...
```

注意，实际上这个Spark解决方案并不需要方法JavaRDD.saveAsTextFile()和JavaPairRDD.saveAsTextFile()（这两个方法用于在HDFS中保存RDD）。提供这些方法只是为了进行调试，帮助你理解这个过程。

## 步骤7：按参考值过滤生物标志物

Spark提供了一个非常简单而强大的API来完成RDD的过滤。要过滤元素，只需要实现一个filter()函数：对于你想要保留的记录，这个函数返回true，对于想要剔除的记录则返回false，如示例23-20所示。

示例23-20：步骤7: 按参考值过滤生物标志物

```
1 // JavaRDD<T> filter(Function<T,Boolean> f)
2 // 返回一个新RDD，只包含满足某个谓词条件的元素。
3 JavaRDD<String> filtered = biosets.filter(new Function<String,Boolean>() {
4     public Boolean call(String record) {
5         String ref = REF.value();
6         String[] tokens = record.split(",");
7         if (ref.equals(tokens[1])) {
8             return true; // 返回这些记录
9         }
```

```
10          else {
11              return false; // 不返回这个记录
12          }
13      }
14  });
15  filtered.saveAsTextFile("/output/2");
```

提供这一步的示例输出是为了进行调试；要注意，输出中只出现了r2参考值（所有其他参考值{r1, r3, r4}都会从结果RDD中剔除）：

```
# hadoop fs -cat /output/2/*
g1,r2,p1,1.86
g2,r2,p1,0.74
g3,r2,p1,1.24
g5,r2,p1,1.69
g6,r2,p1,0.93
g7,r2,p1,1.44
g8,r2,p1,2.11
g1,r2,p2,2.46
g2,r2,p2,3.24
g4,r2,p2,2.11
g5,r2,p2,1.69
g6,r2,p2,1.25
...
```

## 步骤8：创建（Gene-ID, (Patient-ID, Gene-Value)）对

这个步骤如示例23-21所示，这里实现一个map()函数，它会将：

```
<Gene-ID><,><reference><,><Patient-ID><,><Gene-Value>
```

转换为一个（K, V）对，其中：

```
K = Gene-ID
V = Tuple2<Patient-ID, Gene-Value>
```

需要说明，从这一步开始，不再需要参考值（reference）。

示例23-21：步骤8: 创建（Gene-ID, (Patient-ID, Gene-Value)）对

```
1   // 步骤8: 创建（Gene-ID, (Patient-ID, Gene-Value)）对
2   // PairMapFunction<T, K, V>
3   // T => Tuple2<K, V> = Tuple2<geneID, Tuple2<patientID, geneValue>>
4   //
5   JavaPairRDD<String,Tuple2<String,Double>> pairs =
        filtered.mapToPair(new PairFunction<
6       String,                         // T
7       String,                         // K = g1234 (一个Gene-ID)
8       Tuple2<String,Double>           // V = <Patient-ID, Gene-Value>
9       >() {
10      public Tuple2<String,Tuple2<String,Double>> call(String rec) {
11          String[] tokens = rec.split(",");
12          // tokens[0] = 1234
```

```
            // tokens[1] = 2 （这是{"1", "2", "3", "4"}中的一个参考值）
            // tokens[2] = patientID
            // tokens[3] = value
            Tuple2<String,Double> V =
               new Tuple2<String,Double>(tokens[2], Double.valueOf(tokens[3]));
            return new Tuple2<String,Tuple2<String,Double>>(tokens[0], V);
        }
    });
    pairs.saveAsTextFile("/output/3");
```

提供这一步的示例输出是为了进行调试：

```
# hadoop fs -cat /output/3/*
(g1,(p1,1.86))
(g2,(p1,0.74))
(g3,(p1,1.24))
(g5,(p1,1.69))
(g6,(p1,0.93))
(g7,(p1,1.44))
(g8,(p1,2.11))
(g1,(p2,2.46))
(g2,(p2,3.24))
...
```

## 步骤9：按基因分组

这个步骤如示例23-22所示，按Gene-ID对数据分组。这个分组的结果是一个新RDD，格式如下：

```
JavaPairRDD<String, Iterable<Tuple2<String,Double>>>
```

其中：

```
K = Gene-ID
V = Iterable<Tuple2<Patient-ID, Gene-Value>>
```

示例23-22：步骤9: 按基因分组

```
// 步骤9: 按Gene-ID对生物标志物分组
JavaPairRDD<String, Iterable<Tuple2<String,Double>>> grouped =
  pairs.groupByKey();
grouped.saveAsTextFile("/output/4");
// grouped = (K, V)
// 其中
// K = Gene-ID
// V = Iterable<Tuple2<Patient-ID,Gene-Value>>
grouped.saveAsTextFile("/output/5");
```

提供这一步的部分示例输出是为了进行调试：由于篇幅限制，这里对输出的格式稍有调整：

```
# hadoop fs -cat /output/5/*
(g1,[(p1,1.86), (p1,1.76), (p1,1.16), (p3,1.06),
```

```
        (p1,1.86), (p2,1.46), (p2,1.33), (p2,2.46),
        (p2,2.46), (p2,1.33), (p3,2.61), (p1,2.86),
        (p2,2.06), (p2,1.43)])
    (g2,[(p2,3.24), (p2,1.24), (p2,2.0), (p3,1.55),
        (p1,1.74), (p2,3.2), (p2,2.5), (p1,0.74),
        (p1,2.84), (p1,1.33), (p3,1.24), (p3,2.1),
        (p1,2.74), (p2,2.24), (p2,2.0)])
    ...
```

## 步骤10：创建所有基因的笛卡儿积

要完成所有基因对的相关系数计算，必须创建所有基因的一个笛卡儿积。这个工作利用JavaPairRDD.cartesian()函数来完成，如示例23-23所示。

示例23-23：步骤10: 创建所有基因的笛卡儿积

```
// 步骤10: 创建所有基因的笛卡儿积
// <U> JavaPairRDD<T,U> cartesian(JavaRDDLike<U,?> other)
// 返回当前RDD与另一个RDD的笛卡儿积；
// 也就是所有元素对(a, b)的RDD
// 其中a在this中，b在other中。
JavaPairRDD< Tuple2<String, Iterable<Tuple2<String,Double>>>,
             Tuple2<String, Iterable<Tuple2<String,Double>>>
           > cart = grouped.cartesian(grouped);
cart.saveAsTextFile("/output/6");
// cart =
//        (g1, g1), (g1, g2), (g1, g3), (g1, g4)
//        (g2, g1), (g2, g2), (g2, g3), (g2, g4)
//        (g3, g1), (g3, g2), (g3, g3), (g3, g4)
//        (g4, g1), (g4, g2), (g4, g3), (g4, g4)
```

## 步骤11：过滤冗余基因对

令g1和g2是两个基因；由于（g1, g2）的皮尔逊相关系数与（g2, g1）的皮尔逊相关系数相同，为了减少计算时间，我们将过滤重复的基因对（参见示例23-24）。当且仅当g1 < g2时才会保留基因对（g1, g2）。

示例23-24：步骤11: 过滤冗余基因对

```
// 步骤11: 过滤冗余基因对
// 当且仅当（g1 < g2）才保留（g1, g2）。
// 过滤后，将有：
// filtered2 =
//            (g1, g2), (g1, g3), (g1, g4)
//            (g2, g3), (g2, g4)
//            (g3, g4)
//
// JavaRDD<T> filter(Function<T,Boolean> f)
 // 返回一个新RDD，其中只包含满足某个谓词条件的元素。
 JavaPairRDD<Tuple2<String, Iterable<Tuple2<String,Double>>>,
        Tuple2<String, Iterable<Tuple2<String,Double>>>> filtered2 =
              cart.filter(new Function<
              Tuple2<Tuple2<String, Iterable<Tuple2<String,Double>>>,
```

```
                Tuple2<String, Iterable<Tuple2<String,Double>>>
              >,
          Boolean>() {
      public Boolean call(Tuple2<
          Tuple2<String, Iterable<Tuple2<String,Double>>>,
          Tuple2<String, Iterable<Tuple2<String,Double>>>> pair) {
          // pair._1 = Tuple2<String, Iterable<Tuple2<String,Double>>>
          // pair._2 = Tuple2<String, Iterable<Tuple2<String,Double>>>
          if (smaller(pair._1._1, pair._2._1)) {
            return true; // 返回这些记录
          }
          else {
            return false; // 不返回这些记录
          }
      }
  });
  filtered2.saveAsTextFile("/output/7");
```

这里给出这一步的示例输出是为了进行调试。由于篇幅限制，这里对输出的格式稍有调整：

```
# hadoop fs -cat /output/7/*

...
((g1,[(p2,2.46), (p2,2.46), (p2,1.33), (p3,2.61), (p1,2.86),
      (p2,2.06), (p2,1.43), (p1,1.86), (p1,1.76), (p1,1.16),
      (p3,1.06), (p1,1.86), (p2,1.46), (p2,1.33)]),
(g2,[(p2,3.24), (p2,1.24), (p2,2.0), (p3,1.55), (p1,1.74),
     (p2,3.2), (p1,0.74), (p1,2.84), (p1,1.33), (p3,1.24),
     (p3,2.1), (p1,2.74), (p2,2.24), (p2,2.0), (p2,2.5)])
)
...
((g3,[(p1,1.24), (p1,1.24), (p1,1.64), (p1,2.66),
      (p3,2.22), (p1,1.24)]),
(g4,[(p2,2.11), (p2,2.11), (p2,1.77), (p2,2.01),
     (p3,2.87), (p2,1.11), (p2,1.77), (p2,1.78)])
)
...
```

## 步骤12：皮尔逊相关系数和p值

现在将生成的基因对与适当的数据聚集，这一步将完成皮尔逊相关系数的计算，并找出相关的p值（见示例23-25）。如果没有足够的数据来计算相关系数，则为相关系数和p值返回Double.NaN。这里使用Apache Commons Math3包完成皮尔逊相关系数计算（将在后面的小节中给出）。

示例23-25：步骤12: 计算皮尔逊相关系数和p值

```
// 步骤12：皮尔逊相关系数和p值
// 接下来，迭代处理所有映射值
// JavaPairRDD<String, List<Tuple2<String,Double>>> mappedvalues
// 创建(K,V)，其中
```

```
//     K = Tuple2<String,String>(g1, g2)
//     V = Tuple2<Double,Double>(corr, pValue)
//
JavaPairRDD<Tuple2<String,String>,Tuple2<Double,Double>> finalresult =
         filtered2.mapToPair(new PairFunction<
            Tuple2<Tuple2<String,Iterable<Tuple2<String,Double>>>,
                   Tuple2<String,Iterable<Tuple2<String,Double>>>>, // 输入
            Tuple2<String,String>,                                  // K
            Tuple2<Double,Double>                                   // V
         >() {
      public Tuple2<Tuple2<String,String>,Tuple2<Double,Double>>
        call(Tuple2<Tuple2<String,Iterable<Tuple2<String,Double>>>,
                Tuple2<String,Iterable<Tuple2<String,Double>>>> t) {
      Tuple2<String,Iterable<Tuple2<String,Double>>> g1 = t._1;
      Tuple2<String,Iterable<Tuple2<String,Double>>> g2 = t._2;

      Map<String, MutableDouble> g1map = toMap(g1._2);
      Map<String, MutableDouble> g2map = toMap(g2._2);
      // 现在完成correlation(one, other)
      // 确保相应地根据Patient-ID对值排序
      // 每个Patient-ID可能有一个或多个值
      List<Double> x = new ArrayList<Double>();
      List<Double> y = new ArrayList<Double>();
      for (Map.Entry<String, MutableDouble> g1Entry : g1map.entrySet()) {
          String g1PatientID = g1Entry.getKey();
          MutableDouble g2MD = g2map.get(g1PatientID);
          if (g2MD != null) {
             // 对应Patient-ID的one和other都有值
             x.add(g1Entry.getValue().avg());
             y.add(g2MD.avg());
          }
      }

      System.out.println("x="+x);
      System.out.println("y="+y);
      // K = 基因对
      Tuple2<String,String> K = new Tuple2<String,String>(g1._1,g2._1);
       if (x.size() < 3) {
         // 没有足够的数据完成相关系数计算
         return new Tuple2<Tuple2<String,String>,Tuple2<Double,Double>>
             (K, new Tuple2<Double,Double>(Double.NaN, Double.NaN));

         }
         else {
            // 皮尔逊相关系数
            double correlation = Pearson.getCorrelation(x, y);
            double pValue = Pearson.getpValue(correlation, (double) x.size() );
            return new Tuple2<Tuple2<String,String>,Tuple2<Double,Double>>
                (K, new Tuple2<Double,Double>(correlation, pValue));
       }
    }
});
finalresult.saveAsTextFile("/output/corr");
```

这里提供了最后一步的完整输出来进行调试。需要说明，如果两个基因没有足够的数据完成相关系数计算，由于缺少适当的值，所以会发出Double.NaN：

```
# hadoop fs -cat /output/corr/part*
((g1,g2),(-0.5600331663273436,0.6215767617117369))
((g1,g3),(NaN,NaN))
((g1,g4),(NaN,NaN))
((g1,g5),(-0.02711004213333685,0.9827390963782845))
((g1,g6),(0.19358340989347553,0.8759779754044315))
((g1,g7),(-0.8164277145788058,0.3919024816433061))
((g1,g8),(0.1671231007563918,0.8931045335800389))
((g1,g9),(0.6066217061857698,0.5850485651167254))
((g2,g3),(NaN,NaN))
((g2,g4),(NaN,NaN))
((g2,g5),(-0.8129831655334596,0.3956841419099777))
((g2,g6),(-0.9212118275674606,0.25440121369269475))
((g2,g7),(0.9356247601371344,0.22967428006843038))
((g2,g8),(-0.9104130933946509,0.271527771868302))
((g2,g9),(-0.9983543136507176,0.036528196595012385))
((g3,g4),(NaN,NaN))
((g3,g5),(NaN,NaN))
((g3,g6),(NaN,NaN))
((g3,g7),(NaN,NaN))
((g3,g8),(NaN,NaN))
((g3,g9),(NaN,NaN))
((g4,g5),(NaN,NaN))
((g4,g6),(NaN,NaN))
((g4,g7),(NaN,NaN))
((g4,g8),(NaN,NaN))
((g4,g9),(NaN,NaN))
((g5,g6),(0.9754751741958164,0.1412829282172836))
((g5,g7),(-0.5551020210096935,0.625358421978409))
((g5,g8),(0.9810429471659294,0.12415637004167612))
((g5,g9),(0.7782528977513095,0.4322123385049901))
((g6,g7),(-0.7245714063087034,0.4840754937611247))
((g6,g8),(0.9996381540187659,0.017126558175602602))
((g6,g9),(0.8973843455132996,0.2909294102877069))
((g7,g8),(-0.7057703774748613,0.5012020519367326))
((g7,g9),(-0.9543282445223156,0.19314608347341866))
((g8,g9),(0.885190414128154,0.30805596846331396))
```

## 皮尔逊相关系数包装器类

Pearson是一个包装器类，示例23-26中给出了这个类的定义。它提供了两个方法：getCorrelation()和getpValue()。底层实现使用了Apache Commons Math3包。

示例23-26：Pearson包装器类

```
1 import java.util.List;
2 import java.util.Arrays;
3 import org.apache.commons.math3.distribution.TDistribution;
4 import org.apache.commons.math3.stat.correlation.PearsonsCorrelation;
5 /**
```

```
 6  * 计算皮尔逊相关系数和p值的类
 7  *
 8  */
 9 public class Pearson {
10
11    final static PearsonsCorrelation PC = new PearsonsCorrelation();
12
13    public static double getCorrelation(List<Double> X, List<Double> Y) {
14        double[] xArray = toDoubleArray(X);
15        double[] yArray = toDoubleArray(Y);
16        double corr = PC.correlation(xArray, yArray);
17        return corr;
18    }
19
20    private static double[] toDoubleArray(List<Double> list) {
21        if (list == null) {
22           return null;
23        }
24        double[] arr = new double[list.size()];
25        for (int i=0; i < list.size(); i++) {
26            arr[i] = list.get(i);
27        }
28        return arr;
29    }
30
31    public static double getpValue(final double corr, final int n) {
32       return getpValue(corr, (double) n);
33    }
34
35    public static double getpValue(final double corr, final double n) {
36       double t = Math.abs(corr * Math.sqrt( (n-2.0) / (1.0 - (corr * corr)) ));
37       System.out.println(" t =" + t);
38       TDistribution tdist = new TDistribution(n-2);
39       double pValue = 2* (1.0 - tdist.cumulativeProbability(t));
40       return pValue;
41    }
42 }
```

## 测试Pearson类

在示例23-27中，我们来测试这个包装器类。

示例23-27：测试Pearson包装器类

```
 1 import java.util.List;
 2 import java.util.Arrays;
 3 public class TestPearson {
 4
 5    public static void main(String[]args) {
 6       test(args);
 7    }
 8
 9    /**
10     * 测试/调试
```

```
11     */
12    public static void test(String[] args) {
13         // 索引                          0   1   2   3   4
14       List<Double> X = Arrays.asList(2.0, 4.0, 45.0, 6.0, 7.0);
15       List<Double> Y = Arrays.asList(23.0, 5.0, 54.0, 6.0, 7.0);
16       double n = X.size(); // 5.0;
17       double corr = Pearson.getCorrelation(X, Y);
18       double pValue = Pearson.getpValue(corr, n);
19       System.out.println("corr =" + corr);
20       System.out.println("pValue =" + pValue);
21    }
22 }
```

## 使用R计算皮尔逊相关系数

示例23-28展示了如何使用R语言计算皮尔逊相关系数。通过这个例子，可以对我们的Java实现与R做个比较。

示例23-28：使用R计算皮尔逊相关系数

```
 1 > x=c(2,4,45,6,7)
 2 > y=c(23,5,54,6,7)
 3 > cor.test(x,y)
 4
 5 Pearson's product-moment correlation
 6
 7 data: x and y
 8 t = 3.6026, df = 3, p-value = 0.03669
 9 alternative hypothesis: true correlation is not equal to 0
10 95 percent confidence interval:
11  0.09266873 0.99352355
12 sample estimates:
13       cor
14 0.9012503
```

# 运行Spark程序的YARN脚本

要在YARN中运行我们的Spark程序*AllVersusAllCorrelation*，可以使用以下shell脚本：

```
# cat run_all_vs_all.sh
#!/bin/bash
SPARK_HOME=/usr/local/spark-1.0.0
# app jar:
BOOK_HOME=/mp/data-algorithms-book
APP_JAR=$BOOK_HOME/dist/data_algorithms_book.jar
prog=org.dataalgorithms.chap23.spark.AllVersusAllCorrelation
reference=r2
biomarkers=/mp/biomarkers.txt
$SPARK_HOME/bin/spark-submit --class $prog \
    --master yarn-cluster \
    --num-executors 12 \
    --driver-memory 3g \
```

```
    --executor-memory 7g \
    --executor-cores 12 \
    $APP_JAR $reference $biomarkers
```

# 使用Spark计算斯皮尔曼相关系数

要计算斯皮尔曼相关系数（Spearman correlation）而不是皮尔逊相关系数，只需要把步骤12中的下面两行（参见“步骤12: 计算皮尔逊相关系数和p值”）：

```
// 皮尔逊相关系数
double correlation = Pearson.getCorrelation(x, y);
double pValue = Pearson.getpValue(correlation, (double) x.size() );
```

替换为：

```
// 斯皮尔曼相关系数
double correlation = Spearman.getCorrelation(x, y);
double pValue = Spearman.getpValue(correlation, (double) x.size() );
```

然后使用下一节提供的Spearman包装器类。

## 斯皮尔曼相关系数包装器类

Spearman包装器类的定义见示例23-29，它提供了两个方法：getCorrelation()和getpValue()。底层实现使用了Apache Commons Math3包。

示例23-29：Spearman包装器类

```
 1 import java.util.Arrays;
 2 import java.util.List;
 3 import java.util.ArrayList;
 4 import org.apache.commons.math3.distribution.TDistribution;
 5 import org.apache.commons.math3.stat.correlation.SpearmansCorrelation;
 6
 7 /**
 8  * 计算两个向量之间斯皮尔曼等级相关系数的类。
 9  *
10  * @author Mahmoud Parsian
11  *
12  */
13 public class Spearman {
14
15     final static SpearmansCorrelation SC = new SpearmansCorrelation();
16
17     public static double getCorrelation(List<Double> X, List<Double> Y) {
18        double[] xArray = toDoubleArray(X);
19        double[]yArray = toDoubleArray(Y);
20        double corr = SC.correlation(xArray, yArray);
21        return corr;
22     }
```

```
23
24     public static double getpValue(double corr, double n) {
25        double t = Math.abs(corr * Math.sqrt( (n-2.0) / (1.0 - (corr * corr)) ));
26        System.out.println(" t =" + t);
27        TDistribution tdist = new TDistribution(n-2);
28        double pValue = 2.0 * (1.0 - tdist.cumulativeProbability(t));
29        return pValue;
30     }
31
32     static double[] toDoubleArray(List<Double> list) {
33        double[] arr = new double[list.size()];
34        for (int i=0; i < list.size(); i++) {
35           arr[i] = list.get(i);
36        }
37        return arr;
38     }
39 }
```

## 测试斯皮尔曼相关系数包装器类

示例23-30测试Spearman包装器类。

示例23-30：测试Spearman包装器类

```
 1 import java.util.Arrays;
 2 import java.util.List;
 3 public class TestSpearman {
 4
 5     public static void main(String[] args) {
 6         test(args);
 7     }
 8
 9     public static void test(String[] args) {
10        //                              1    2    3    4    5
11        List<Double> X = Arrays.asList(2.0, 4.0, 45.0, 6.0, 7.0);
12        List<Double> Y = Arrays.asList(23.0, 5.0, 54.0, 6.0, 7.0);
13        double n = X.size(); // 5.0;
14        double corr = getCorrelation(X, Y);
15        double pValue = getpValue(corr, n);
16        System.out.println("corr =" + corr);
17        System.out.println("pValue =" + pValue);
18        //                               1      2     3     4      5
19        List<Double> X2 = Arrays.asList(12.0, 14.0, 45.0, 6.0, 17.0);
20        List<Double> Y2 = Arrays.asList(3.0, 5.0, 15.0, 16.0, 17.0);
21        double n2 = X2.size(); // 5.0;
22        double corr2 = getCorrelation(X2, Y2);
23        double pValue2 = getpValue(corr2, n2);
24        System.out.println("corr2 =" + corr2);
25        System.out.println("pValue2 =" + pValue2);
26     }
27 }
```

这一章提供了两个可伸缩的分布式解决方案（MapReduce/Hadoop和Spark）来完成皮尔逊

相关系数计算，这在临床应用中非常重要。我们还介绍了如何使用我们的Spark解决方案计算斯皮尔曼相关系数。下一章将提供DNA碱基计数的分布式算法，这些算法可以用于基因组算法和应用中。

# 第24章

# DNA碱基计数

这一章将为DNA碱基计数提供4个解决方案：

- 使用FASTA格式的MapReduce/Hadoop解决方案。
- 使用FASTQ格式的MapReduce/Hadoop解决方案。
- 使用FASTA格式的Spark解决方案。
- 使用FASTQ格式的Spark解决方案。

这一章的目的是统计DNA[注1]碱基。人类DNA编码只使用4个字母（A, C, T和G），如果无法识别编码，就把它标为N。DNA编码的含义由字母*A*, *T*, *C*和*G*的序列确定，这就类似于英语单词的含义由字母表字母 (A~Z)的序列来确定。

在这一章中，我们将得出一组给定DNA序列中A, T, C, G和N的频率（或百分比），另外还为Hadoop的输入文件提供了定制记录阅读器。

那么，在DNA领域字母*ATCG*到底代表什么？它们分别表示与DNA关联的4种含氮碱基：

- A =腺嘌呤（Adenine）。

注1：脱氧核糖核酸（Deoxyribonucleic acid，DNA）是一种包含遗传指令的分子，用于所有已知生物体以及很多病毒的发育和机能运作。DNA是一种核酸；核酸以及蛋白质和碳水化合物是构成所有已知生命形式基础的3种大分子。大多数DNA分子都包括2个生物聚合物长链，它们相互反向平行盘旋构成一个双螺旋结构。这两个DNA长链称为多核苷酸链，这是因为它们由称为核苷酸的更简单的单元组成。每个核苷酸包括一个含氮核碱基（可以是鸟嘌呤（G）、腺嘌呤（A）、胸腺嘧啶（T）或胞嘧啶（C）），另外还包括一个称为脱氧核糖的单糖和一个磷酸基团（资料来源：Wikipedia）。

- T =胸腺嘧啶（Thymine）。
- C =胞嘧啶（Cytosine）。
- G =鸟嘌呤（Guanine）。

例如，ACGGGTACGAAT是一个非常小的DNA序列。DNA序列可能很庞大[注2]。对于这个例子，DNA碱基计数将生成表24-1所示的结果。

**表24-1：DNA碱基计数示例**

| 碱基 | 数目 |
|---|---|
| a | 4 |
| t | 2 |
| c | 2 |
| g | 4 |
| n | 0 |

DNA序列可以采用多种不同的格式表示，包括常用的基于文本的FASTA和FASTQ格式，这也是我们的解决方案中将要使用的格式。需要说明，Hadoop的默认记录阅读器会逐行地读取记录，因此我们无法使用Hadoop的默认记录阅读器读取FASTQ格式的文件，这是因为FASTQ格式的文件中，一个记录由包括4行的一个序列组成。我们要插入`FastaInputFormat`和`FastqInputFormat`来分别读取FASTA和FASTQ文件格式（注入定制记录阅读器）。如果不使用定制记录阅读器，就必须在DNA碱基计数中剔除不需要的记录（行），对于FASTA格式这很容易做到，但是对于FASTQ就不那么容易了。

# FASTA格式

FASTA格式的序列文件可以包含多个序列。FASTA格式的各个序列以一个单行描述开头，后面是一行或多行序列数据。这个描述行必须以第1列上的一个大于号（>）开头。

## FASTA格式示例

下面是一个FASTA格式的示例序列（这个文件包含4个序列，不区分大小写）：

```
# cat test_fasta.fasta
>seq1
cGTAaccaataaaaaaacaagcttaacctaattc
>seq2
agcttagTTTGGatctggccgggg
```

注2：单倍体人类基因组（包含在精子和卵子细胞中）包含30亿个DNA碱基对，而二倍体基因组（包含在体细胞中）中DNA数量则加倍”（资料来源：Wikipedia）。

```
>seq3
gcggatttactcCCCCCAAAAANNaggggagagcccagataaatggagtctgtgcgtccaca
gaattcgcacca
AATAAAACCTCACCCAT
agagcccagaatttactcCCC
>seq4
gcggatttactcaggggagagcccagGGataaatggagtctgtgcgtccaca
gaattcgcacca
```

# FASTQ格式

FASTQ是一个基于文本的格式，用于存储一个生物序列（通常是一个核苷酸序列）以及相应的质量评分。为简单起见，序列字母和质量评分都编码为单个ASCII字符。FASTQ文件中，通常使用4行表示一个序列，如下所示：

- 第1行以一个@字符开头，后面是一个序列标识符和一个可选的描述。
- 第2行是原始序列字母（如A, T, C, G）。
- 第3行以一个+字符开头，后面可以再次出现相同的序列标识符（和可能的描述），不过这是可选的。
- 第4行编码第2行中序列的质量值，包含的符号个数必须与原始序列中的字母个数相同。

## FASTQ格式示例

下面给出一个FASTQ文件，其中包含一个DNA序列：

```
@SEQ_ID
GATTTGGGGTTCAAAGCAGTATCGATCAAATAGTAAATCCATTTGTTCAACTCACAGTTT
+
!''*((((***+))%%%++)(%%%%).1***-+*''))**55CCF>>>>>>CCCCCCC65
```

# MapReduce解决方案：FASTA格式

要完成DNA碱基计数，我们将使用一个映射器、一个归约器和一个定制FASTA格式阅读器（实现FASTA阅读器时，将在`Job.setInputFormatClass()`方法中注入`FastaInputFormat`类）。

## 读取FASTA文件

如何让MapReduce/Hadoop框架读取FASTA数据？Hadoop中的默认阅读器会逐行读取输入记录，而FASTA记录会跨多行。Hadoop为“输入格式”和“记录阅读器”提供了一个插件框架。我们开发了两个定制插件类`FastaInputFormat`和`FastaRecordReader`来支

持读取FASTA格式。在Hadoop中，由TextInputFormat类提供默认输入格式。我们的定制类FastaInputFormat扩展了这个类，并覆盖了两个方法：createRecordReader()和isSplitable()。覆盖的createRecordReader()方法会返回一个新的定制记录阅读器，名为FastaRecordReader。

## MapReduce FASTA解决方案：map()

要得出DNA碱基数，map()函数将获得一个FASTA记录，这可能是一行或多行DNA序列。map()函数再对各行完成词法分析，并统计碱基。为了让映射器更为高效，并不是对每一个字母执行emit(letter, 1)，而是使用一个散列表（Map<Character, Long>）为各个碱基发出一个总数。最后，通过使用Hadoop的cleanup()方法，迭代处理这个散列表并执行emit(letter, countOfLetter)。散列表可以提供一个非常高效的解决方案，因为散列键限制为仅有的几个DNA字母。另外，这个“聪明”的map()算法只向MapReduce框架发出最少的(letter, countOfLetter)对，因此可以减少网络流量。映射器类FastaCountBaseMapper的结构如示例24-1所示。

示例24-1：FastaCountBase映射器类

```
public class FastaCountBaseMapper ... {

   Map<Character, Long> dnaBasesCounter = null;

   // 这个函数在映射任务开始时调用一次。
   setup() {
      dnaBasesCounter = new HashMap<Character, Long>();
   }

   map(Object key, String value) {
      ...
   }

   // 在映射任务结束时调用一次
   cleanup() {
       // 现在迭代处理baseCounter并发出<key, value>
       for (Map.Entry<Character, Long> entry : dnaBaseCounter.entrySet()) {
          emit(entry.getKey(), entry.getValue());
       }
    }
}
```

map()函数如示例24-2所示。

示例24-2：DNA碱基计数：map()函数

```
/**
 * @param key为Hadoop生成的键（在这里忽略）
 * @param value为一个给定文档的一行输入
 */
map(Object key, String value) {
```

```
    // fasta是多个DNA序列行构成的一个字符串
    String fasta = value.trim().toLowerCase();
    String[] lines = fasta.split("[\\r\\n]+");
    for (int i=1; i < lines.length; i++) {
       char[] array = lines[i].toCharArray();
       for(char c : array) {
          Long v = dnaBaseCounter.get(c);
          if (v == null) {
             dnaBaseCounter.put(c, 1);
          }
          else {
             dnaBaseCounter.put(c, v+1);
          }
       }
    }
}
```

## MapReduce FASTA解决方案: reduce()

由于DNA字母表中的字母很有限，我们可以直接将归约器个数设定为5（每个DNA字母对应一个归约器）。这个归约器与传统单词计数归约器几乎是一样的，只不过它会累加各个DNA字母的计数器（见示例24-3）。

示例24-3：DNA碱基计数: reduce()函数

```
/**
 * @param key为映射器生成的唯一DNA字母
 * @param value是一个整数列表（唯一单词的部分计数）
 */
reduce(String key, List<long> value) {
    long sum = 0;
    for (int count : value) {
        sum += count;
    }
    emit(key, sum);
}
```

# 运行示例

这一节提供一个运行示例，并给出这个FASTA格式Hadoop解决方案生成的输出。

## 示例运行日志

这是Hadoop解决方案运行示例的日志输出（因为篇幅限制，对输出做了编辑和格式调整）：

```
# ./run.sh
...
Deleted hdfs://localhost:9000/dna-base-count/output
```

```
13/03/15 09:16:22 INFO CountBaseDriver: inputDir=/dna-base-count/input
13/03/15 09:16:22 INFO CountBaseDriver: outputDir=/dna-base-count/output
...
13/03/15 09:16:24 INFO mapred.JobClient: map 0% reduce 0%
...
13/03/15 09:17:53 INFO mapred.JobClient: map 100% reduce 100%
...
13/03/15 09:17:58 INFO mapred.JobClient: Map-Reduce Framework
13/03/15 09:17:58 INFO mapred.JobClient: Map input records=7
13/03/15 09:17:58 INFO mapred.JobClient: Reduce input records=44
13/03/15 09:17:58 INFO mapred.JobClient: Reduce input groups=9
13/03/15 09:17:58 INFO mapred.JobClient: Reduce output records=9
13/03/15 09:17:58 INFO mapred.JobClient: Map output records=44
13/03/15 09:17:58 INFO CountBaseDriver: run(): status=true
13/03/15 09:17:58 INFO CountBaseDriver: returnStatus=0
```

## 生成的输出

下面是这个Hadoop FASTA解决方案生成的输出：

```
$ hadoop fs -cat /dna-base-count/output/p*
c   82
G   10
g   80
T   5
A   7
t   82
C   7
a   110
N   5
```

## 定制排序

如果想把a和A发送到同一个归约器，而且希望将a的计数与A的计数累加（因为它们是相同的DNA字母），该怎么做呢？至少有两种方法可以完成这个任务：

*做法1*：

将map()的所有输入转换为小写字母。可以在映射器代码（map()函数）中很容易地做到这一点，只需要把下面这一行代码：

```
String fasta = value.toString();
```

替换为：

```
String fasta = value.toString().toLowerCase();
```

*做法2*：

提供一个结果排序器，注入到MapReduce框架中。我们可以定义一个比较器，控制在将键传递到归约器之前如何对键进行排序，如下所示：

```
job.setSortComparatorClass(BaseComparator.class);
```

示例24-4显示了如何为做法1实现比较器。

示例24-4：定制比较器：BaseComparator

```
1 import org.apache.hadoop.io.Text;
2 import org.apache.hadoop.io.WritableComparator;
3 import org.apache.hadoop.io.WritableComparable;
4
5 public class BaseComparator extends WritableComparator {
6    protected BaseComparator() {
7       super(Text.class, true);
8    }
9
10   @Override
11   public int compare(WritableComparable w1, WritableComparable w2) {
12      Text t1 = (Text) w1;
13      Text t2 = (Text) w2;
14      String S1 = t1.toString().toUpperCase();
15      String S2 = t2.toString().toUpperCase();
16      int cmp = S1.compareTo(S2);
17      return cmp;
18   }
19 }
```

在Hadoop中，要编写一个定制比较器，只需要扩展`org.apache.hadoop.io.WritableComparator`类并实现`compare()`方法。在这里，定制比较器（`BaseComparator`类）会对键（a, A, t, T, c, C, g, G, n, N）排序（忽略大小写），使得DNA碱基的大写和小写版本相邻。默认地，注入到`setSortComparatorClass()`和`setGroupingComparatorClass()`方法的类会使用相同的比较器。例如，如果设置`setSortComparatorClass()`而`setGroupingComparatorClass()`未设置，`setGroupingComparatorClass()`就会使用我们为`setSortComparatorClass()`设置的比较器（不过反过来不一定成立）。可以用这个例子来测试和验证。

## 定制分区

如果想要将所有键只发送到5个归约器（每个DNA字母对应一个归约器），该怎么做呢？答案就是提供一个定制分区器。Hadoop中的默认分区器是`HashPartitioner`，它会对映射器生成的键计算散列，来确定这个（`key, value`）属于哪一个分区（哪一个归约器）。这样一来，分区数就等于作业的归约任务数。所以关键问题是，`map()`生成一个（`key, value`）时，必须发送到一个归约器吗？如果是，应当发送到哪一个归约器？可以使用`Job.setPartitionerClass()`方法来提供定制分区器，将字母a/A的所有键发送到归约器1，字母b/B的所有键发送到归约器2，依此类推（参见示例24-5）。

示例24-5：定制分区器

```
import org.apache.hadoop.mapreduce.Partitioner;

/** 按碱基{A,T,G,C,a,t,g,c}对键分区。 */
public class BasePartitioner<K, V> extends Partitioner<K, V> {
   public int getPartition(K key, V value, int numReduceTasks) {
      String base = key.toString();
      if (base.compareToIgnoreCase("A") == 0) {
         return 0;
      }
      else if (base.compareToIgnoreCase("C") == 0) {
         return 1;
      }
      else if (base.compareToIgnoreCase("G") == 0) {
         return 2;
      }
      else if(base.compareToIgnoreCase("T") == 0) {
         return 3;
      }
      else {
         return 4;
      }
   }
}
```

在我们驱动器类中注入这个定制类，如示例24-6所示。

示例24-6：DNA碱基计数驱动器

```
public class CountBaseDriver extends Configured implements Tool {

    public int run(String[] args) throws Exception {
       Job job = new Job(getConf(), "count-dns-bases");
       job.setJarByClass(CountBaseMapper.class);
       job.setMapperClass(CountBaseMapper.class);
       job.setReducerClass(CountBaseReducer.class);
       job.setNumReduceTasks(5);
        // job.setCombinerClass(CountBaseCombiner.class);
       job.setInputFormatClass(FastaInputFormat.class);
       job.setPartitionerClass(BasePartitioner.class);
       job.setSortComparatorClass(BaseComparator.class);
       job.setGroupingComparatorClass(BaseComparator.class);
       job.setOutputKeyClass(Text.class);
       job.setOutputValueClass(LongWritable.class);
       FileInputFormat.addInputPath(job, new Path(args[0]));
       FileOutputFormat.setOutputPath(job, new Path(args[1]));

       boolean status = job.waitForCompletion(true);
       theLogger.info("run(): status="+status);
       return status ? 0 : 1;
    }
    ...
}
```

现在运行这个作业时，会有以下输出（因为篇幅有限，这里对输出做了编辑和格式调整）：

```
$ ./run.sh
...
Deleted hdfs://localhost:9000/dna-base-count/output
13/03/15 09:44:26 INFO CountBaseDriver: inputDir=/dna-base-count/input
13/03/15 09:44:26 INFO CountBaseDriver: outputDir=/dna-base-count/output
13/03/15 09:44:27 INFO input.FileInputFormat: Total input paths to process : 1
13/03/15 09:44:27 INFO mapred.JobClient: Running job: job_201303150852_0004
13/03/15 09:44:28 INFO mapred.JobClient: map 0% reduce 0%
...
13/03/15 09:45:58 INFO mapred.JobClient: map 100% reduce 100%
13/03/15 09:46:03 INFO mapred.JobClient: Job complete: job_201303150852_0004
...
13/03/15 09:46:03 INFO mapred.JobClient: Map-Reduce Framework
13/03/15 09:46:03 INFO mapred.JobClient: Map input records=7
13/03/15 09:46:03 INFO mapred.JobClient: Reduce input records=44
13/03/15 09:46:03 INFO mapred.JobClient: Reduce input groups=5
13/03/15 09:46:03 INFO mapred.JobClient: Reduce output records=5
13/03/15 09:46:03 INFO mapred.JobClient: Map output records=44
13/03/15 09:46:03 INFO CountBaseDriver: run(): status=true
13/03/15 09:46:03 INFO CountBaseDriver: returnStatus=0
```

现在我们只有5个归约器：

```
$ hadoop fs -ls /dna-base-count/output/p*
-rw-r--r-- ... 6 2013-03-15 10:06 /dna-base-count/output/part-r-00000
-rw-r--r-- ... 5 2013-03-15 10:06 /dna-base-count/output/part-r-00001
-rw-r--r-- ... 5 2013-03-15 10:06 /dna-base-count/output/part-r-00002
-rw-r--r-- ... 5 2013-03-15 10:06 /dna-base-count/output/part-r-00003
-rw-r--r-- ... 4 2013-03-15 10:06 /dna-base-count/output/part-r-00004

$ hadoop fs -cat /dna-base-count/output/p*

a 117
c 89
g 90
t 87
n 5
```

# MapReduce解决方案: FASTQ格式

为了完成DNA碱基计数，我们要使用一个映射器和一个归约器。这个FASTQ解决方案与FASTA解决方案非常相似，只不过要注入一个不同的类来处理输入格式。对于FASTQ格式，我们要注入FastqInputFormat类：

```
job.setInputFormatClass(FastqInputFormat.class);
```

通过使用FastqInputFormat，映射器会得到一个键-值对，其中键是一个LongWritable，

值是一个Text对象，格式如下（行分隔符是,;,）：

```
<line-1><,;,><line-2><,;,><line-3><,;,><line-4>
```

对于每个映射器值，将有4行，因为每个FASTQ记录对应4行数据。FASTQ格式的映射器类会与FASTA类有所区别，因为输入格式是不同的，不过归约器类没有变化。

## 读取FASTQ文件

如何让MapReduce/Hadoop框架读取FASTQ数据？Hadoop中的默认阅读器会逐行读取输入记录，但是对于FASTQ格式，我们需要4行、4行地读取，因为每4行一组才是一个FASTQ记录。如果只看一行中包含的字母，并不能可靠地确定这一行是一个序列名、序列本身还是质量行，所以需要读取全部4行。每一组中的第2行包含序列，所以我们要使用FASTQ文件的行号。Hadoop为“输入格式”和“记录阅读器”提供了一个插件框架。我们开发了两个定制插件类FastqInputFormat和FastqRecordReader来支持读取FASTQ格式。在Hadoop中，由TextInputFormat类提供默认输入格式。我们的定制类FastqInputFormat扩展了这个类，并覆盖了两个方法：createRecordReader()和isSplitable()。所覆盖的createRecordReader()方法返回一个新的定制记录阅读器，名为FastqRecordReader。示例24-7显示了FastqInputFormat类如何实现。

示例24-7：FastqInputFormat: 定制InputFormat类

```
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.InputSplit;
import org.apache.hadoop.mapreduce.RecordReader;
import org.apache.hadoop.mapreduce.TaskAttemptContext;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import org.apache.hadoop.mapreduce.JobContext;
/**
 * 这个类针对FASTQ文件为Hadoop MapReduce框架
 * 定义了一个InputFormat。
 */
public class FastqInputFormat extends TextInputFormat {
  @Override
  public RecordReader<LongWritable, Text> createRecordReader(
              InputSplit inputSplit,
              TaskAttemptContext taskAttemptContext) {
    return new FastqRecordReader();
  }

  @Override
  public boolean isSplitable(JobContext context, Path file) {
      return false;
  }
}
```

另一个定制类是FastqRecordReader，它扩展了RecordReader<LongWritable, Text>类（参见示例24-8）。这个FastqRecordReader类将输入数据分解为键-值对，作为映射器的输入。在这里，FastqRecordReader会创建值（String对象），其中可以包含一个FASTQ记录的4行。

示例24-8：FastqRecordReader: 定制RecordReader类

```
import ...
import org.apache.hadoop.mapreduce.RecordReader;

/**
 * 这个类针对FASTQ文件为Hadoop MapReduce框架
 * 定义了一个RecordReader。
 */
public class FastqRecordReader extends RecordReader<LongWritable, Text> {
   ...
}
```

## MapReduce FASTQ 解决方案: map()

要得出DNA碱基数，map()函数要得到一个FASTQ记录，这是一个4行文本，其中只有第2行包含一个DNA序列。然后map()函数会对第2行（包含DNA序列）完成词法分析，并统计碱基。为了让我们的映射器更为高效，这里不会对每一个字母执行emit(letter,1)，而是使用一个散列表（Map<Character, Long>）为各个碱基发出一个总数。最后，通过使用Hadoop的cleanup()方法，迭代处理这个散列表并执行emit(letter, countOfLetter)。散列表可以提供一个非常高效的解决方案，因为散列键限制为仅有的几个DNA字母。另外，这个聪明的map()算法只向MapReduce框架发出最少的（letter, countOfLetter）对，因此可以减少网络流量。如示例24-9所示。

示例24-9：DNA碱基计数：使用FASTQ格式的map()函数

```
Map<Character, Long> dnaBasesCounter = null;
/**
 * 这个函数在映射任务开始时调用一次。
 */
setup() {
   dnaBasesCounter = new HashMap<Character, Long>();
}

/**
 * @param key为Hadoop生成的键（在这里忽略）
 * @param value为给定文档中的一行输入。
 */
map(Object key, String value) {
   String fastq = value.toString();
   // 4行，用一个特殊的定界符分隔: ",;,"
   String[] lines = fastq.split(",;,");
   // 第2行= lines[1] = DNA序列
   char[] array = lines[1].toCharArray();
```

```
19    for(char c : array) {
20        Long v = dnaBaseCounter.get(c);
21        if (v == null) {
22           dnaBaseCounter.put(c, 1);
23        }
24        else {
25           dnaBaseCounter.put(c, v+1);
26        }
27     }
28 }
29
30 /**
31  * 在映射任务结束时调用一次。
32  */
33 cleanup() {
34    // 现在迭代处理baseCounter，并发出<key, value>
35    for (Map.Entry<Character, Long> entry : dnaBaseCounter.entrySet()) {
36       emit(entry.getKey(), entry.getValue());
37    }
38 }
```

## MapReduce FASTQ解决方案: reduce()

由于DNA字母表中的字母很有限，我们可以直接将归约器个数设定为5（每个DNA字母对应一个归约器）。这个归约器同样与传统单词计数归约器几乎是一样的；只不过它会累加各个DNA字母的计数器。如示例24-10所示。

示例24-10：DNA碱基计数：使用FASTQ格式的reduce()函数

```
1 /**
2  * @param key为映射器生成的唯一DNA字母
3  * @param value是一个整数列表（唯一单词的部分计数）
4  */
5 reduce(String key, List<long> value) {
6    long sum = 0;
7    for (int count : value) {
8        sum += count;
9    }
10   emit(key, sum);
11 }
```

## Hadoop实现类：FASTQ格式

这个Hadoop实现包括表24-2所示的类。

表24-2：Hadoop实现类

| 类名 | 描述 |
|---|---|
| BaseComparator | DNA碱基的比较器类 |
| BasePartitioner | 定制分区器类 |

表24-2：Hadoop实现类（续）

| 类名 | 描述 |
| --- | --- |
| FastqCountBaseDriver | 提交Hadoop作业的驱动器类 |
| FastqCountBaseMapper | 映射器类 |
| FastqCountBaseReducer | 归约器类 |
| FastqInputFormat | 定制InputFormat类 |
| FastqRecordReader | 定制RecordReader类 |

# 运行示例

这一节提供一个运行示例以及FASTQ格式Hadoop解决方案生成的输出。

## 示例运行日志

下面给出这个FASTQ格式Hadoop解决方案的运行示例日志（由于篇幅限制，对这个输出做了编辑和格式调整）：

```
$ ./run_fastq.sh
...
Deleted hdfs://localhost:9000/fastq/output
...
13/03/19 09:24:45 INFO FastqCountBaseDriver: run(): input args[0]=/fastq/input
13/03/19 09:24:45 INFO FastqCountBaseDriver: run(): output args[1]=/fastq/output
13/03/19 09:24:47 INFO mapred.JobClient: map 0% reduce 0%
...
13/03/19 09:26:03 INFO mapred.JobClient: map 100% reduce 100%
...
13/03/19 09:26:08 INFO mapred.JobClient: Map input records=5
...
13/03/19 09:26:08 INFO mapred.JobClient: Reduce output records=4
13/03/19 09:26:08 INFO mapred.JobClient: Map output records=4
13/03/19 09:26:08 INFO FastqCountBaseDriver: run(): status=true
```

## 生成的输出

这个Hadoop FASTQ解决方案生成的输出如下：

```
$ hadoop fs -ls /fastq/output/
-rw-r--r-- 1 ... 5 2013-03-19 09:25 /fastq/output/part-r-00000
...
-rw-r--r-- 1 ... 5 2013-03-19 09:25 /fastq/output/part-r-00008
-rw-r--r-- 1 ... 0 2013-03-19 09:25 /fastq/output/part-r-00009

$ hadoop fs -cat /fastq/output/p*
c 36
g 22
t 67
a 55
```

# Spark 解决方案: FASTA格式

这一节使用Spark来解决FASTA格式的DNA碱基计数问题。与FASTQ格式相比，识别FASTA格式的DNA序列要容易得多：如果一行以>开头，这就是一个命令行（也就是说，它不是DNA序列的一部分，要将其忽略）；否则，这就是一个合法的DNA序列。下面是我们的算法：读取FASTA格式的输入，对输入分区，为各个分区创建一个散列表(散列表的键是DNA编码{A, T, C, G}，值是这些DNA编码的频率)，最后将所有分区的散列表组合/合并为一个最终的散列表。通过使用Spark API，整个解决方案可以使用一个Java类提供。首先，我会给出高层步骤，然后提供各个步骤（作为一个Spark变换或动作）。

## 高层步骤

下面给出DNA碱基计数问题的FASTA格式Spark解决方案的高层步骤。

示例24-11：高层步骤

```
1 package org.dataalgorithms.spark.chap24;
2 // 步骤1: 导入所需的类和接口
3 public class SparkDNABaseCountFASTA {
4    public static void main(String[] args) throws Exception {
5        // 步骤2: 处理输入参数
6        // 步骤3: 从FASTA输入创建一个RDD
7        // 步骤4: 映射分区
8        // 步骤5: 收集所有DNA碱基数目
9        // 步骤6: 发出最终碱基数
10
11       // 完成
12       ctx.close(); // 关闭JavaSparkContext对象
13       System.exit(0);
14   }
15 }
```

### 步骤1：导入所需的类和接口

在这个步骤中，如示例24-12所示，要为这个解决方案导入所需的类和接口。

示例24-12：步骤1: 导入所需的类和接口

```
1 // 步骤1: 导入所需的类和接口
2 import java.util.Map;
3 import java.util.List;
4 import java.util.HashMap;
5 import java.util.Iterator;
6 import java.util.Collections;
7 import org.apache.spark.api.java.JavaRDD;
8 import org.apache.spark.api.java.JavaSparkContext;
9 import org.apache.spark.api.java.function.FlatMapFunction;
```

## 步骤2：处理输入参数

这一步展示了如何处理输入参数。

示例24-13：步骤2: 处理输入参数

```
// 步骤2: 处理输入参数
if (args.length != 1) {
    System.err.println("Usage: SparkDNABaseCountFASTA <input-path>");
    System.exit(1);
}
final String inputPath = args[0];
```

## 步骤3：从FASTA输入创建一个RDD

在这一步中，如示例24-14所示，将从FASTA输入创建一个RDD。

示例24-14：步骤3: 从FASTA输入创建一个RDD

```
// 步骤3: 从FASTA输入创建一个RDD
JavaSparkContext ctx = new JavaSparkContext();
JavaRDD<String> fastaRDD = ctx.textFile(inputPath, 1);
```

## 步骤4：映射分区

在示例24-15中，为每个分区创建一个散列表（K,V），其中K是一个DNA编码，V是这个DNA编码关联的频率。

示例24-15：步骤4: 映射分区

```
// 步骤4: 映射分区
// <U> JavaRDD<U> mapPartitions(FlatMapFunction<Iterator<T>,U> f)
//为这个RDD的每个分区应用一个函数来返回一个新RDD。
JavaRDD<Map<Character, Long>> partitions = fastaRDD.mapPartitions(
    new FlatMapFunction<Iterator<String>, Map<Character,Long>>() {
    @Override
    public Iterable<Map<Character,Long>> call(Iterator<String> iter) {
        Map<Character,Long> baseCounts = new HashMap<Character,Long>();
        while (iter.hasNext()) {
            String record = iter.next();
            if (record.startsWith(">")) {
               // 这是一个FASTA注释记录，将其忽略
            }
           else {
              String str = record.toUpperCase();
              for (int i = 0; i < str.length(); i++) {
                  char c = str.charAt(i);
                  Long count = baseCounts.get(c);
                  if (count == null) {
                     baseCounts.put(c, 1l);
                  }
                  else {
                     baseCounts.put(c, count+1l);
                  }
```

```
                }
            }
        }
        return Collections.singletonList(baseCounts);
    }
});
```

### 步骤5：收集所有DNA碱基数目

如示例24-16所示，要收集所有DNA碱基数。

示例24-16：步骤5: 收集所有DNA碱基数

```
// 步骤5: 收集所有DNA碱基数
List<Map<Character, Long>> list = partitions.collect();
System.out.println("list="+list);
Map<Character, Long> allBaseCounts = list.get(0);
for (int i=1; i < list.size(); i++) {
    Map<Character, Long> aBaseCount = list.get(i);
    for (Map.Entry<Character, Long> entry : aBaseCount.entrySet()) {
      char base = entry.getKey();
      Long count = allBaseCounts.get(base);
      if (count == null) {
          allBaseCounts.put(base, entry.getValue());
      }
      else {
         allBaseCounts.put(base, (count + entry.getValue()));
      }
    }
}
```

### 步骤6：发出最终碱基数

在这一步中，如示例24-17所示，我们要发出最终的DNA碱基数。

示例24-17：步骤6：发出最终碱基数

```
// 步骤6: 发出最终碱基数
for (Map.Entry<Character, Long> entry : allBaseCounts.entrySet()) {
    System.out.println(entry.getKey() + "\t" + entry.getValue());
}
```

## 运行示例

下面的各个小节会提供这个Spark FASTA解决方案运行示例的输入、脚本和生成的输出。

### 输入

```
$ hadoop fs -cat /home/hadoop/testspark/fasta.txt
>seq1
cGTAaccaataaaaaaacaagcttaacctaattc
>seq2
```

```
agcttagTTTGGatctggccgggg
>seq3
gcggatttactcCCCCCAAAAANNaggggagagcccagataaatggagtctgtgcgtccaca
gaattcgcacca
AATAAAACCTCACCCAT
agagcccagaatttactcCCC
>seq4
gcggatttactcaggggagagcccagGGataaatggagtctgtgcgtccaca
gaattcgcacca
```

## 脚本

下面是运行这个DNA碱基计数问题FASTA格式Spark解决方案的脚本：

```
$ cat run_spark_dna_base_count_fasta.sh
#!/bin/bash
export JAVA_HOME=/usr/java/jdk7
export SPARK_HOME=/usr/local/spark-1.1.0
export SPARK_MASTER=spark://myserver100:7077
export BOOK_HOME=/mp/data-algorithms-book
export APP_JAR=$BOOK_HOME/dist/data_algorithms_book.jar
INPUT=/home/hadoop/testspark/fasta.txt
# 在Spark独立集群上运行
$SPARK_HOME/bin/spark-submit \
  --class org.dataalgorithms.chap24.spark.SparkDNABaseCountFASTA \
  --master $SPARK_MASTER \
  --executor-memory 2G \
  --total-executor-cores 20 \
  $APP_JAR \
  $INPUT
```

## 示例运行日志

我们的脚本在包括3个节点（{myserver100, myserver200, myserver300}）的一个Spark集群环境中运行。由于篇幅限制，这里对输出做了编辑：

```
# ./run_spark_dna_base_count_fasta.sh
...
INFO : Remoting started; listening on addresses :
  [akka.tcp://sparkDriver@myserver100:59451]
INFO : Remoting now listens on addresses:
  [akka.tcp://sparkDriver@myserver100:59451]
INFO : Successfully started service 'sparkDriver' on port 59451.
...
INFO : Executor added: app-20141114113127-0023/0 on worker-20141016164917-
  myserver200-34042 (myserver200:34042) with 2 cores
INFO : Granted executor ID app-20141114113127-0023/0 on hostPort
  myserver200:34042 with 2 cores, 2.0 GB RAM
INFO : Executor added: app-20141114113127-0023/1 on
  worker-20141016164917-myserver300-52001 (myserver300:52001) with 2 cores
INFO : Granted executor ID app-20141114113127-0023/1 on hostPort
  myserver300:52001 with 2 cores, 2.0 GB RAM
INFO : Executor added: app-20141114113127-0023/2 on
```

```
  worker-20141016164917-myserver100-58455 (myserver100:58455) with 2 cores
INFO : Granted executor ID app-20141114113127-0023/2 on hostPort
  myserver100:58455 with 2 cores, 2.0 GB RAM
INFO : Executor updated: app-20141114113127-0023/0 is now RUNNING
INFO : Executor updated: app-20141114113127-0023/1 is now RUNNING
INFO : Executor updated: app-20141114113127-0023/2 is now RUNNING
...
INFO : Job finished: collect at SparkDNABaseCountFASTA.java:67,
  took 0.069880929 s
INFO : Removed TaskSet 2.0, whose tasks have all completed, from pool
list=[{T=45, N=2, G=53, A=73, C=61}]
T       45
N       2
G       53
A       73
C       61
...
```

# Spark解决方案: FASTQ格式

这一节使用Spark来解决FASTQ格式的DNA碱基计数问题。采用FASTQ格式时，每个DNA序列是一个列表，包含4个连续的行/记录。对于DNA碱基计数，我们只对第2行感兴趣，这才是具体的DNA序列（另外3行将被忽略）。在Spark中，默认的记录阅读器会逐行读取输入记录。对于FASTQ，我们无法使用默认的记录阅读器来读取格式，因为不能从一个给定的行/记录明确地识别出具体的DNA序列。如果可以做到这一点，我们也能逐行地读取，只处理具体的DNA序列。由于无法从FASTQ文件的一个给定行识别出DNA序列，所以要一次读取FASTQ文件的4行（同样地，每一组4行表示FASTQ文件的一个记录）。

不过如何按记录读取FASTQ文件呢（也就是说，一次读取4行）？答案是编写一个定制`InputFormat`（作为一个插件类），它会返回4行输入，而不是1行输入。我们将用`FastqInputFormat`完成这个任务，这个类扩展了`TextInputFormat`（需要说明，FASTQ文件是常规的文本文件）。下面给出我们的算法：用一个定制`FastqInputFormat`读取FASTQ格式的输入（这会返回4行，不过只会处理表示DNA序列的那一行），对输入分区，为各个分区创建一个散列表(散列表的键是DNA编码{A, T, C, G}，值是这些DNA编码的频率)，最后将所有分区的散列表组合/合并为一个最终的散列表。通过使用Spark API，整个解决方案可以使用一个Java类提供。首先，我会给出高层步骤，然后提供各个步骤（作为一个Spark变换或动作）。

## 高层步骤

示例24-18给出了DNA碱基计数问题的FASTQ格式Spark解决方案的高层步骤。

示例24-18：高层步骤

```
package org.dataalgorithms.spark.chap24;
// 步骤1: 导入所需的类和接口
public class SparkDNABaseCountFASTQ {
    public static void main(String[] args) throws Exception {
       // 步骤2: 处理输入参数
       // 步骤3: 从FASTQ输入格式创建一个JavaPairRDD
       // 步骤4: 映射分区
       // 步骤5: 收集所有DNA碱基数
       // 步骤6: 发出最终碱基数

       // 完成
       ctx.close(); // 关闭JavaSparkContext对象
       System.exit(0);
   }
}
```

## 步骤1：导入所需的类和接口

在这个步骤中，如示例24-19所示，要为这个解决方案导入所需的类和接口。

示例24-19：步骤1: 导入所需的类和接口

```
// 步骤1: 导入所需的类和接口
import scala.Tuple2;

import java.util.Map;
import java.util.List;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Collections;

import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.api.java.JavaPairRDD;
import org.apache.spark.api.java.JavaSparkContext;
import org.apache.spark.api.java.function.FlatMapFunction;

import org.apache.hadoop.io.Text;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.conf.Configuration;
```

## 步骤2：处理输入参数

示例24-20显示了如何处理输入参数。

示例24-20：步骤2: 处理输入参数

```
// 步骤2: 处理输入参数
if (args.length != 1) {
    System.err.println("Usage: SparkDNABaseCountFASTQ <input-path>");
    System.exit(1);
}
final String inputPath = args[0];
```

## 步骤3：从FASTQ输入创建一个JavaPairRDD

在这个步骤中，如示例24-21所示，要从FASTQ输入创建一个RDD。

示例24-21：步骤3: 从FASTQ输入创建一个JavaPairRDD

```
// 步骤3: 从FASTQ输入创建一个JavaPairRDD
JavaSparkContext ctx = new JavaSparkContext();

// import org.apache.hadoop.mapreduce.InputFormat;
// public <K,V,F extends InputFormat<K,V>> JavaPairRDD<K,V> newAPIHadoopFile(
//             String path,
//             Class<F> fClass,
//             Class<K> kClass,
//             Class<V> vClass,
//             org.apache.hadoop.conf.Configuration conf)
// 利用任意的一个新API InputFormat以及传入输入格式的额外配置选项
// 为一个给定的Hadoop文件创建一个RDD。
// 注意：由于Hadoop的RecordReader类为每个记录重用同样的Writable
// 对象，直接缓存所返回的RDD
// 会创建同一个对象的多个引用。如果计划直接缓存Hadoop
// Writable对象，首先要使用一个map函数进行复制。

//// 可以利用coalesce()对数据分区
////   public JavaPairRDD<T> coalesce(int N)
////   返回一个新RDD，它将归约到N个分区。
JavaPairRDD<LongWritable ,Text> fastqRDD = ctx.newAPIHadoopFile(
                inputPath,              // 输入路径
                FastqInputFormat.class, // 定制InputFormat
                LongWritable.class,     // 定制InputFormat返回的键
                Text.class,             // 定制InputFormat返回的值
                new Configuration()     //  Hadoop配置对象
);
```

## 步骤4：映射分区

步骤4如示例24-22所示，展示了如何映射分区并创建本地Map<Character，Long>对象。

示例24-22：步骤4: 映射分区

```
// 步骤4: 映射分区
// <U> JavaRDD<U> mapPartitions(FlatMapFunction<Iterator<T>,U> f)
// 对这个RDD的各个分区应用一个函数返回一个新RDD
JavaRDD<Map<Character, Long>> partitions = fastqRDD.mapPartitions(
   new FlatMapFunction<Iterator<Tuple2<LongWritable ,Text>>,
                       Map<Character,Long>
                      >() {
   @Override
   public Iterable<Map<Character,Long>> call(
      Iterator<Tuple2<LongWritable ,Text>> iter) {
      Map<Character,Long> baseCounts = new HashMap<Character,Long>();
      while (iter.hasNext()) {
         Tuple2<LongWritable ,Text> kv = iter.next();
         String fastqRecord = kv._2.toString(); // 得到一个FASTQ记录
         String[] lines = fastqRecord.split(",;,");
```

```
16          // 第2行(也就是lines[1])是DNA序列。
17          String sequence = lines[1].toUpperCase();
18          for (int i = 0; i < sequence.length(); i++) {
19             char c = sequence.charAt(i);
20             Long count = baseCounts.get(c);
21             if (count == null) {
22                baseCounts.put(c, 1l);
23             }
24             else {
25                baseCounts.put(c, count+1l);
26             }
27          }
28       }
29       return Collections.singletonList(baseCounts);
30    }
31 });
```

## 步骤5：收集所有DNA碱基数

在示例24-23中，我们收集了所有DNA碱基数。

示例24-23：步骤5: 收集所有DNA碱基数

```
1 // 步骤5: 收集所有DNA碱基数
2 List<Map<Character, Long>> list = partitions.collect();
3 System.out.println("list="+list);
4 Map<Character, Long> allBaseCounts = list.get(0);
5 for (int i=1; i < list.size(); i++) {
6     Map<Character, Long> aBaseCount = list.get(i);
7     for (Map.Entry<Character, Long> entry : aBaseCount.entrySet()) {
8         char base = entry.getKey();
9         Long count = allBaseCounts.get(base);
10        if (count == null) {
11            allBaseCounts.put(base, entry.getValue());
12        }
13        else {
14           allBaseCounts.put(base, (count + entry.getValue()));
15        }
16    }
17 }
```

## 步骤6：发出最终碱基数

这一步中，如示例24-24所示，要发出最终的碱基数目。

示例24-24：步骤6: 发出最终碱基数

```
1 // 步骤6: 发出最终碱基数
2 for (Map.Entry<Character, Long> entry : allBaseCounts.entrySet()) {
3     System.out.println(entry.getKey() + "\t" + entry.getValue());
4 }
```

## 运行示例

下面的小节会给出这个Spark FASTQ解决方案的运行示例的输入、脚本和生成的输出。

### 输入

```
$ hadoop fs -cat /home/hadoop/testspark/sample.fastq
@EAS54_6_R1_2_1_413_324
CCCTTCTTGTCTTCAGCGTTTCTCC
+
;;3;;;;;;;;;;;;7;;;;;;;88
@EAS54_6_R1_2_1_540_792
TTGGCAGGCCAAGGCCGATGGATCA
+
;;;;;;;;;;;7;;;;;-;;;3;83
@EAS54_6_R1_2_1_443_348
GTTGCTTCTGGCGTGGGTGGGGGGG
+EAS54_6_R1_2_1_443_348
;;;;;;;;;;;9;7;;.7;393333
```

### 脚本

下面是运行DNA碱基计数问题FASTQ格式Spark解决方案的脚本：

```
$ cat ./run_spark_dna_base_count_fastq.sh
#!/bin/bash
export JAVA_HOME=/usr/java/jdk7
export BOOK_HOME=/mp/data-algorithms-book
export SPARK_HOME=/usr/local/spark-1.1.0
export SPARK_MASTER=spark://myserver100:7077
export SPARK_JAR=$BOOK_HOME/lib/spark-assembly-1.1.0-hadoop2.5.0.jar
export APP_JAR=$BOOK_HOME/dist/data_algorithms_book.jar
#
# 构建OTHER_JARS中的所有其他jar
JARS='find $BOOK_HOME/lib -name '*.jar''
OTHER_JARS=""
for J in $JARS ; do
   OTHER_JARS=$J,$OTHER_JARS
done
#
INPUT=/home/hadoop/testspark/sample.fastq
DRIVER=org.dataalgorithms.chap24.spark.SparkDNABaseCountFASTQ
  --class $DRIVER \
  --master $SPARK_MASTER \
  --jars $OTHER_JARS \
  $APP_JAR $INPUT
```

### 示例运行日志

我们的脚本将在一个包括3个节点（myserver100，myserver200，myserver300）的Spark集群环境中运行。由于篇幅限制，这里对输出页面做了编辑：

```
$ ./run_spark_dna_base_count_fastq.sh
...
INFO : Remoting started; listening on addresses :
  [akka.tcp://sparkDriver@myserver100:52511]
INFO : Remoting now listens on addresses:
  [akka.tcp://sparkDriver@myserver100:52511]
INFO : Successfully started service 'sparkDriver' on port 52511.
...
INFO : Stage 0 (collect at SparkDNABaseCountFASTQ.java:102)
  finished in 7.367 s
INFO : Job finished: collect at SparkDNABaseCountFASTQ.java:102,
  took 7.446341308 s
T 22
G 27
A 7
C 19
```

这一章为DNA碱基计数问题提供了多个MapReduce和Spark解决方案，可以在基因组相关应用中使用。下一章会为RNA测序问题提供一个MapReduce/Hadoop解决方案，可以用于基因组和临床应用。

# 第25章

# RNA测序

近年来，RNA (核糖核酸)测序为基因表达式领域的探索带来了变革。基于RNA测序方法取得的进展，研究人员可以快速分析和研究转录组。Dr. Ananya Mandel（*http://bit.ly/what_is_rna*）将RNA定义为“一种由核苷酸长链组成的重要分子。核苷酸由一个含氮碱基、一个五碳糖和一个磷酸构成。与DNA类似，RNA对所有生物至关重要。”RNA的主要功能是将创建蛋白质所需的基因密码从细胞核转移到核糖体。Dr. Mandel把它描述为：“这个过程可以防止DNA离开细胞核。这可以保护DNA和基因密码不被破坏。如果没有RNA，就无法合成蛋白质。”

这一章将提供一个完整的计算管道MapReduce解决方案，用来分析不同基因表达式的RNA测序（RNA-Seq）数据。在我们的实现中，会利用两个开源包：

*TopHat(http://bit.ly/TopHat-tool)*

这是一个处理RNA测序read(短序列)的快速剪接映射器。它使用超高通短序列比对器Bowtie将RNA测序短序列与哺乳类基因组进行比对，然后分析映射结果来找出外显子之间的剪接。

*Cufflinks(http://bit.ly/cufflinks_tool)*

组装转录本，估计其丰度，并检验RNA测序样本中的差异表达和调控。Cufflinks得到比对过的RNA测序短序列，把这些比对结果组装为一组最大简约的(parsimonious)转录本。然后根据多少个短序列支持各个转录本来估计这些转录本的相对丰度，这里会考虑库准备协议（library preparation protocols）的偏差。

## 数据大小和格式

RNA测序数据可以用FASTA、FASTQ、BAM和很多其他格式表示。TopHat包可以处理

FASTA和FASTQ数据格式。每个RNA测序数据样本的大小可以从30～300 GB（一个RNA测序分析可以达到1 TB甚至更多数据）。

# MapReduce工作流

RNA测序管道包括以下主要步骤：

1. 输入数据验证（输入数据(如FASTQ文件)的质量控制）。
2. 比对（将短序列映射到参考基因组）。
3. 转录本组装（使用Cufflinks和Cuffdiff工具）。

RNA测序工作流如图25-1所示。

## 输入数据验证

这一步要验证FASTQ文件的格式，希望通过验证来保证输入文件的质量。利用输入数据验证工具，可以对来自高通测序管道的原始序列数据（例如，采用FASTQ文件格式）完成一些质量控制检查。

有很多开源工具可以完成输入数据验证。例如，对于FASTQ验证，有以下选择：

- FastQValidator（*http://bit.ly/fastqvalidator*）
- FastQC（*http://bit.ly/FastQC*）

输入数据验证步骤非常简单，也很直接，所以这里不做详细介绍。我们的重点是RNA测序的核心：映射/比对和检验RNA测序样本中的差异表达和调控。

# RNA测序分析概述

具体介绍RNA测序的MapReduce算法之前，首先需要了解RNA测序分析中如何使用数据/样本。要完成一个RNA测序分析，需要比较两组RNA测序数据样本（这是一个比较，RNA测序分析可能包含一个或多个比较）：

*组1*

一个样本集，其中每个样本可能有一个或多个FASTQ文件。例如，这个组可能是一个“正常”样本集。

*组2*

一个样本集，其中每个样本可能有一个或多个FASTQ文件。例如，这个组可能是“癌症”（患病）样本集。

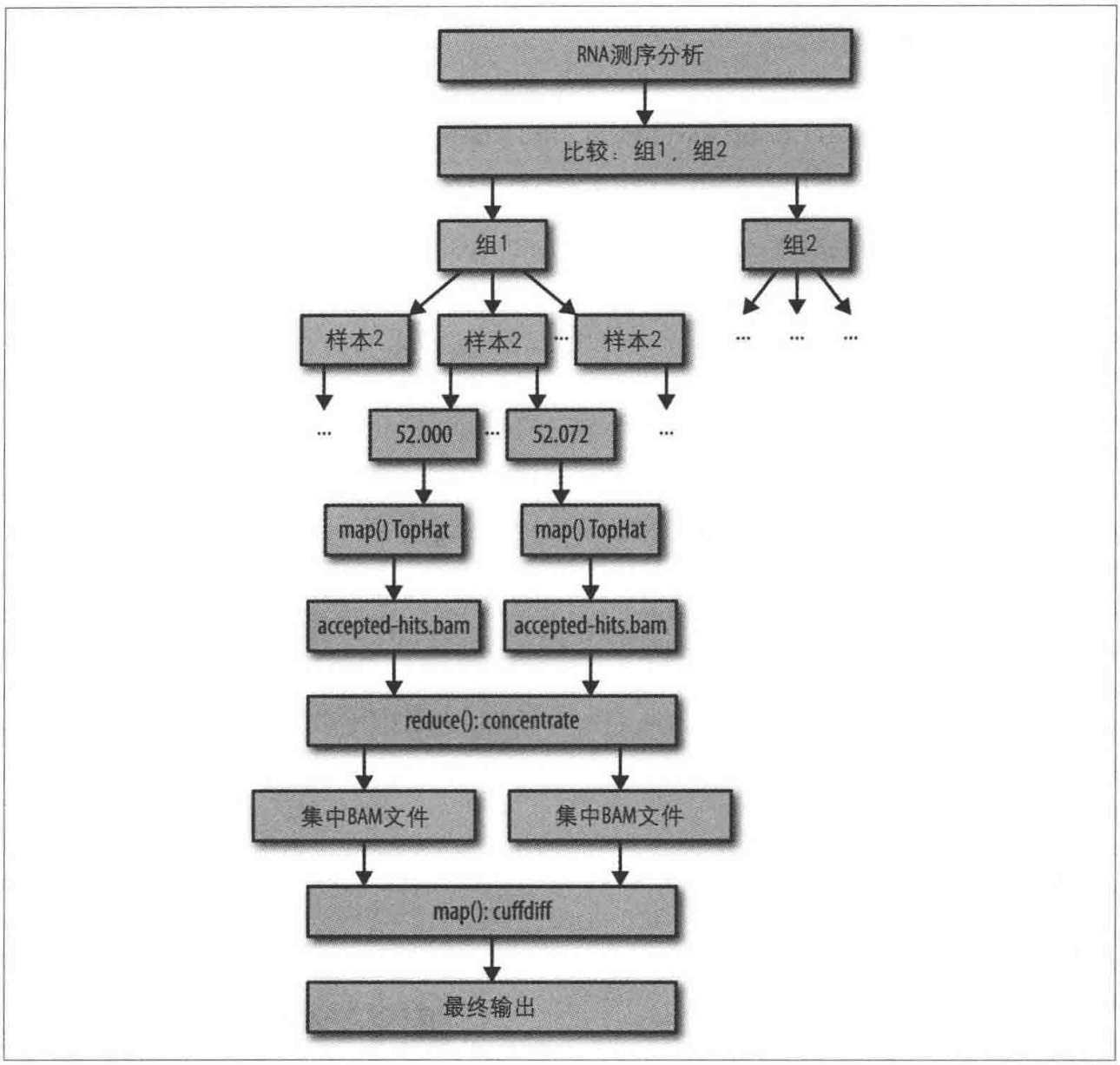



图25-1：RNA测序工作流

目标是找出组1和组2的样本之间转录本表达、剪接和启动子使用的显著差异（一般地，我们会使用Cufflinks的Cuffdiff程序完成两个组之间的RNA测序分析）。通常，RNA测序分析的输入包括一个或多个比较（comparison）。每个比较只包括两个组（如前面所讨论的，就是组1和组2）。

例如，下面是一个RNA测序分析的样本输入。这个分析包括3个比较（比较1、比较2和比较3）。每个比较中有两个组：组1和组2。比较1的组1有4个样本，组2有3个样本（这里没有给出比较2和比较3的样本数据）：

- 比较1

  组1：{样本1, 样本2, 样本3, 样本4}

组2：{样本7, 样本8, 样本9}

这里：

样本1 = {file11.fastq, file12.fastq, file13.fastq}

样本2 = {file21.fastq, file22.fastq}

样本3 = {file31.fastq, file32.fastq, file33.fastq, file34.fastq}

样本4 = {file41.fastq}

...

样本7 = {file71.fastq, file72.fastq}

样本8 = {file81.fastq, file82.fastq, file83.fastq}

样本9 = {file91.fastq, file92.fastq, file93.fastq}

- 比较2

  组1: {...}

  组2: {...}

- 比较3

  组1: {...}

  组2: {...}

RNA测序数据结构如图25-2所示。

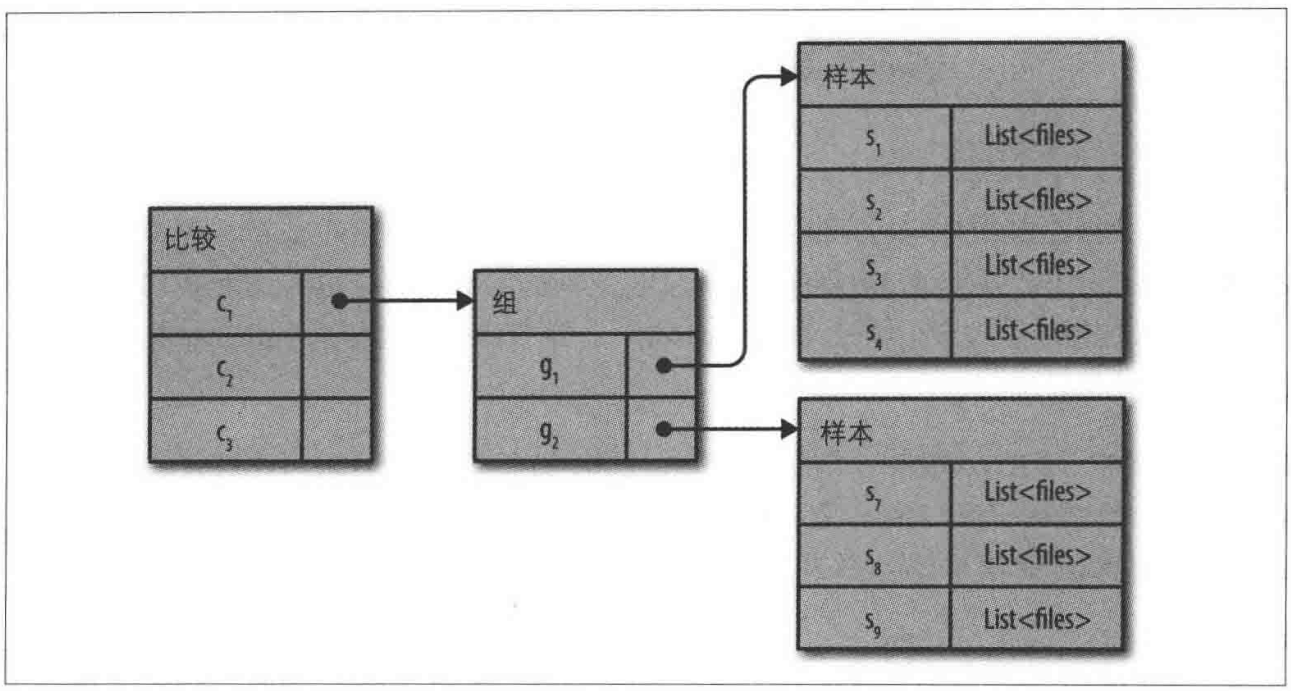



图25-2：RNA测序数据结构

RNA测序的输入数据很容易地用Java类和数据结构表示，如示例25-1所示。

示例25-1：RNA测序数据结构

```
import java.util.Arrays;
import java.util.Map;
import java.util.HashMap;
import java.util.List;
import java.util.ArrayList;

public class Comparison {
    private Map<String, List<String>> group1 = null;
    private Map<String, List<String>> group2 = null;
    public void setGroup1(Map<String, List<String>> group1) {
       this.group1 = group1;
    }
    public void setGroup2(Map<String, List<String>> group2) {
       this.group2 = group2;
    }
    public Map<String, List<String>> getGroup1() {
       return this.group1;
    }
    public Map<String, List<String>> getGroup2() {
       return this.group2;
    }
}
```

现在可以使用Comparison类创建适当的RNA测序数据结构，如示例25-2所示。

示例25-2：RNA测序数据结构使用

```
// 创建3个比较：
List<Comparison> comparisons = new ArrayList<Comparison>();
Comparison comparison_1 = new Comparison();
Comparison comparison_2 = new Comparison();
Comparison comparison_3 = new Comparison();
comparisons.add(comparison_1);
comparisons.add(comparison_2);
comparisons.add(comparison_3);

// 准备comparison_1的组1
Map<String, List<String>> group1 = new HashMap<String, List<String>>()
group1.put("Sample-1w"1, new ArrayList<String>(
  Arrays.asList("file11.fastq", "file12.fastq", "file13.fastq")));
group1.put("Sample-2", new ArrayList<String>(
  Arrays.asList("file21.fastq", "file22.fastq")));
group1.put("Sample-3", new ArrayList<String>(
  Arrays.asList("file31.fastq", "file32.fastq", "file33.fastq", "file34.fastq")));
group1.put("Sample-4", new ArrayList<String>(Arrays.asList("file41.fastq")));

// 准备comparison_1的组2
Map<String, List<String>> group2 = new HashMap<String, List<String>>()
group2.put("Sample-7", new ArrayList<String>(
  Arrays.asList("file71.fastq", "file72.fastq")));
group2.put("Sample-8", new ArrayList<String>(
```

```
25   Arrays.asList("file81.fastq", "file82.fastq", "file83.fastq")));
26 group2.put("Sample-9", new ArrayList<String>(
27   Arrays.asList("file91.fastq", "file92.fastq", "file93.fastq")));
28 comparison_1.setGroup1(group1);
29 comparison_1.setGroup2(group2);
30 ...
```

进一步（使用Linux的`split()`函数）将*<filename>.fastq*文件划分为800万个记录（即200万个FASTQ记录，因为每4行是一个FASTQ记录）。这个划分的原因是为了确保每个映射器可以得到适当数量的输入记录。

# RNA测序MapReduce算法

我们的RNA测序分析解决方案包括两个MapReduce算法步骤：

1. MapReduce TopHat映射。
2. MapReduce Cuffdiff调用。

## 步骤1： MapReduce TopHat映射

这个步骤包括`map()`和`reduce()`函数：映射器使用TopHat比对输入数据，然后归约器连接映射器的输出。步骤1的驱动器程序将数据划分为小块，并将各个输入文件传递到一个映射器。例如，RNA测序分析的输入可以是HDFS目录*/rnaseq/input/2397/*（这里2397是一个`analysisID`，唯一标识这个RNA测序分析）：

```
# hadoop fs -ls /rnaseq/input/2397/
0000.txt
0001.txt
0002.txt
..
0090.txt
0091.txt
```

每个输入文件分别包含一个记录，其中包括多个token（注意，由于记录太长，这里对格式有所调整以便打印，每一行包含一个token和一个分隔符;）：

```
# hadoop fs -cat /rnaseq/input/2397/0090.txt
0090;                          # counter
g1_vs_g2;                      # comparison ID
10332;                         # group ID
62586;                         # sample ID
annotation;                    # RNA-Seq input type
2397;                          # GUID as analysis ID
2;                             # segment mismatches
fr-unstranded;                 # library type
55;                            # segment length
Refseq;                        # gene model
hg18;                          # reference genome
```

```
/data/GSE29006/SRR192334_1_0026, # paired left file
/data/GSE29006/SRR192334_2_0026  # paired right file
```

## map()函数

map()函数调用一个bash shell脚本[注1]来应用TopHat并生成中间BAM文件（见示例25-3）。各个TopHat映射器的输出是一个*accepted_hits.bam*文件，这是SAM格式的短序列比对列表。SAM是一种紧凑的短序列比对格式，已经得到越来越多的应用。

示例25-3：步骤1: map()函数

```
// key 由MR生成，在这里忽略
// value 表示一个输入记录
map(key, value) {
   Map<String, String> tokens = tokenizeMapperRecord(value);
   // 归约器键为：:
   // "<analysis_id><;><comparison_id><;><group_id><;><sample_id>"
   String reducerKey = getReducerKey(tokens);
   RNASeq.tophat(tokens);
   emit(reducerKey, value);
}

public static void tophat(Map<String, String> tokens)
   throws Exception {
   TemplateEngine.init();
   Map<String, String> templateMap = new HashMap<String, String>();
   templateMap.put("key", tokens.get("counter"));
   templateMap.put("counter", tokens.get("counter"));
   templateMap.put("comparison_id", tokens.get("comparison_id"));
   templateMap.put("group_id", tokens.get("group_id"));
   templateMap.put("sample_id", tokens.get("sample_id"));
   templateMap.put("rna_seq_input_type", tokens.get("rna_seq_input_type"));
   templateMap.put("analysis_id", tokens.get("analysis_id"));
   templateMap.put("rna_seq_tophat_segment_mismatches",
      tokens.get("rna_seq_tophat_segment_mismatches"));
   templateMap.put("rna_seq_tophat_library_type",
      tokens.get("rna_seq_tophat_library_type"));
   templateMap.put("rna_seq_tophat_segment_length",
      tokens.get("rna_seq_tophat_segment_length"));
   templateMap.put("rna_seq_tophat_gene_model",
      tokens.get("rna_seq_tophat_gene_model"));
   templateMap.put("rna_seq_tophat_reference_genome",
      tokens.get("rna_seq_tophat_reference_genome"));
   // 从一个模板文件创建具体的脚本
   String scriptFileName = "/rnaseq/scripts/tophat." +
     tokens.get("analysis_id") + "." + tokens.get("counter") + ".sh";
   String logFileName = "/rnaseq/logs/tophat." +
     tokens.get("analysis_id") + "." + tokens.get("counter") + ".log";
   File scriptFile = TemplateEngine.createDynamicContentAsFile("tophat.template",
```

注1：我使用了FreeMarker（*http://freemarker.sourceforge.net/*），这是一个根据模板生成文本输出的通用工具，这里用它来生成shell脚本。

```
39       templateMap, scriptFileName);
40    if (scriptFile != null) {
41       ShellScriptUtil.callProcess(scriptFileName, logFileName);
42    }
43 }
```

## Reduce()函数

Reduce()函数调用一个bash shell脚本，使用SAMtools cat函数连接短序列比对BAM文件（*accepted_hits.bam*）。这个脚本再使用SAMtools sort函数生成有序BAM文件。所以，在我们的例子中，对于比较1（Comparison 1），归约器会生成以下HDFS格式的有序BAM文件：

```
/<hdfs-dir>/${analysis_id}/comparison_1/group_1/sample_1/reducer/sorted.bam
/<hdfs-dir>/${analysis_id}/comparison_1/group_1/sample_2/reducer/sorted.bam
/<hdfs-dir>/${analysis_id}/comparison_1/group_1/sample_3/reducer/sorted.bam
/<hdfs-dir>/${analysis_id}/comparison_1/group_1/sample_4/reducer/sorted.bam

/<hdfs-dir>/${analysis_id}/comparison_1/group_2/sample_7/reducer/sorted.bam
/<hdfs-dir>/${analysis_id}/comparison_1/group_2/sample_8/reducer/sorted.bam
/<hdfs-dir>/${analysis_id}/comparison_1/group_2/sample_9/reducer/sorted.bam
```

归约器的定义见示例25-4。

示例25-4：步骤1: reduce()函数

```
1 // key: "<analysis_id><;><comparison_id><;><group_id><;><sample_id>"
2 // values: 未使用
3 reduce(key, values) {
4    RNASeqTophat.catenatePartitionedBamFiles(key);
5    emit(key, key);
6 }
7
8 /**
9  * 这个方法会连接小/分区的accepted_hits.bam文件
10 * 并创建一个有序的.bam文件。
11 * @param key: "<analysis_id><;><comparison_id><;><group_id><;><sample_id>"
12 */
13 public static void catenatePartitionedBamFiles(String key) {
14    throws Exception {
15    TemplateEngine.init();
16    // 分解键（字段由";"分隔)
17    String[] tokens = reducerKey.split(";");
18    String analysisID = tokens[0];
19    String comparisonID = tokens[1];
20    String groupID = tokens[2];
21    String sampleID = tokens[3];
22    // 创建一个模板映射，传递到FreeMarker模板
23    Map<String, String> templateMap = new HashMap<String, String>();
24    templateMap.put("analysis_id", analysisID);
25    templateMap.put("comparison_id", comparisonID);
26    templateMap.put("group_id", groupID);
27    templateMap.put("sample_id", sampleID);
```

```
28
29    // 从一个模板文件创建具体的脚本
30    String scriptFileName = "/rnaseq/scripts/catenate_partitioned_bam_files." +
31       analysisID + "." + comparisonID + "." + groupID + "." + sampleID + ".sh";
32    String logFileName = "/rnaseq/scripts/catenate_partitioned_bam_files." +
33       analysisID + "." + comparisonID + "." + groupID + "." + sampleID + ".log";
34    File scriptFile = TemplateEngine.createDynamicContentAsFile(
35       "cat_bam_files_partitioned.template.sh", templateMap, scriptFileName);
36    if (scriptFile != null) {
37      ShellScriptUtil.callProcess(scriptFileName, logFileName);
38    }
39 }
```

## 步骤2：MapReduce调用Cuffdiff

这个步骤也包括`map()`和`reduce()`函数（归约器是可选的，不过可以用来完成将来的分析）。映射器对给定比较中的两个组调用`cuffdiff`函数，然后归约器生成最终需要的输出作为biosets/生物标志物以便将来分析。

## map()函数

映射器会为每个比较调用`cuffdiff`函数（也就是说，在我们的例子中，由于有3个比较，所以会调用3次`cuffdiff`）。例如，对于比较1（Comparison 1），会如下调用`cuffdiff`：

```
group_1= {
/<hdfs-dir>/${analysis_id}/comparison_1/group_1/sample_1/reducer/sorted.bam
/<hdfs-dir>/${analysis_id}/comparison_1/group_1/sample_2/reducer/sorted.bam
/<hdfs-dir>/${analysis_id}/comparison_1/group_1/sample_3/reducer/sorted.bam
/<hdfs-dir>/${analysis_id}/comparison_1/group_1/sample_4/reducer/sorted.bam
}

group_2= {
/<hdfs-dir>/${analysis_id}/comparison_1/group_2/sample_7/reducer/sorted.bam
/<hdfs-dir>/${analysis_id}/comparison_1/group_2/sample_8/reducer/sorted.bam
/<hdfs-dir>/${analysis_id}/comparison_1/group_2/sample_9/reducer/sorted.bam
}

cuffdiff $group_1 $group_2
```

`cuffdiff`将生成以下输出文件，保存在HDFS中：

- *isoform_exp.diff*
- *isoforms.read_group_tracking*
- *read_groups.info*

步骤2的映射器如示例25-5所示。

示例25-5：步骤2: map()函数

```
/**
 * 映射器为两组数据调用cuffdiff函数。
 *
 * @param key 为：
 * "<analysis_id><;><comparison_id><;><group_id><;><sample_id>"
 * @param value是一个包含11个token的字符串：
 * index content
 * ---- ---------
 * 0 <counter-as-key>; (such as 0000, 0001, )
 * 1 <comparison_id>;
 * 2 <group1_id>;
 * 3 <group2_id>;
 * 4 constant "cuffdiff"; (ignored)
 * 5 <analysis_id>;
 * 6 <segment_mismatches>;
 * 7 <library_type>;
 * 8 <segment_length>;
 * 9 <gene_model>;
 * 10 <reference_genome>;
 *
 */
map(key, value) {
    Map<String, String> tokens = tokenizeMapperRecord(value);
    //  reducerKey = <analysis_id><;><comparison_id>
    String reducerKey = getReducerKey(tokens);
    RNASeqCuffDiff.cuffdiff(tokens);
    emit(reducerKey, value);
}

public static void cuffdiff(Map<String, String> tokens) throws Exception {
   TemplateEngine.init();
   String scriptFileName = "/rnaseq/scripts/run_cuffdiff." +
      tokens.get("analysis_id") + "." + tokens.get("comparison_id") + ".sh";
   String logFileName = "/rnaseq/logs/run_cuffdiff." +
      tokens.get("analysis_id") + "." + tokens.get("comparison_id") + ".log";
   File scriptFile = TemplateEngine.createDynamicContentAsFile(
      "cuffdiff.sh.template", tokens, scriptFileName);
   if (scriptFile != null) {
      ShellScriptUtil.callProcess(scriptFileName, logFileName);
   }
}
```

## Reduce()函数

步骤2没有归约器，不过也可以使用归约器读取cuffdiff函数的输出，并生成biosets/生物标志物，将来对RNA测序样本进行分析和估计时这可能很有用。

这一章为RNA测序提供了一个通用的高层MapReduce解决方案。这里给出了一个具体的MapReduce解决方案，并介绍了一些开源工具。下一章将为基因聚合（在临床应用中也称为标志物频度）提供一些可伸缩的分布式解决方案。

# 第26章

# 基因聚合

这一章将分别用MapReduce/Hadoop和Spark提供基因聚合（在临床应用中也称为标志物频度）的4种不同的解决方案。基因聚合的输入数据是病人的bioset。前几章中已经讨论过，bioset也称为基因标签（gene signature），其中包含试验样本比较形式的数据（对应转录组、表观遗传和拷贝数变异数据），以及基因型标签（对应GWAS（全基因组关联研究）和突变数据）。简单地说，bioset是一个键-值对列表，其中键是一个geneID，值是一个相关属性的列表。基因聚合在临床应用中用来标识转录标签和基因表达数据的模式。基因聚合还可以用来查看基因如何分组，以及这对整个分析有什么影响。基因聚合是一种革命性的方法，依赖于染色体折叠和更高阶的结构。

可以通过3种度量来得到基因聚合：

- 参考类型（Reference type）指示病人数据的类型：
  - `r1` = 正常（normal）。
  - `r2` = 患病（disease）。
  - `r3` = 成对（paired）。
  - `r4` = 未知（unknown）。
- 基因过滤器类型（Gene filter type ）指示应用到数据的过滤器类型。过滤器类型指示如何分组和分析基因值。例如，如果过滤器类型是`up`，那么只会对大于过滤器阈值的基因值做进一步分析。有3种基因过滤器类型：
  - 绝对值（Absolute value，`abs`）。
  - 大于（Greater than，`up`）。
  - 小于（Less than，`down`）。

- 过滤器阈值，基因过滤器类型使用这个阈值来排除不满足阈值要求的基因。

每个bioset属于一个病人（由patientID标识）。给定一个bioset集合（包含数千或上万个bioset），其中每个bioset可能有多达50000个基因，我们希望找出满足之前提到的3个量度（参考类型、基因过滤器类型和过滤器阈值）的各个基因的频度。过滤器类型指示基因值要如何与过滤器阈值比较（也就是说，要使用绝对值、大于还是小于）。每个bioset的基因数取决于bioset数据类型：拷贝数变异、基因表达式、甲基化或体细胞突变。深入讨论基因聚合的MapReduce算法之前，下面来讨论输入/输出数据。

# 输入

每个bioset可能有20000到50000个记录，每个记录有以下格式（在下面的例子中，病人标识为p100和p200）：

```
<geneID><,><referenceType><;><patientID><,><geneValue>
```

例如：

```
7562135,r1;p100,1.04
```

或：

```
7570769,r1;p200,-1.09
```

# 输出

基因聚合的目标是找出各个基因结合其参考类型（r1, r2, r3或r4）的频度。我们的MapReduce解决方案将在一个分布式文件系统中（例如HDFS）生成以下输出格式：

```
<geneID><,><referenceType><TAB><frequency-count>
```

例如：

```
7562135,r1 1205
```

或：

```
7570769,r3 14067
```

最终，可以把这个输出读入一个Java对象，如GeneAggregationFrequencyCount，有关内容将在“输出分析”一节中讨论。

# MapReduce解决方案（按单个值过滤和按平均值过滤）

基因聚合算法中存在这样一个问题：对于每个病人，是按单个基因值过滤（忽略patientID）还是按基因值平均值过滤（使用patientID）？例如，对于3个病人（分别标识为p100、p200和p300），GENE-1有10个值（如果使用真实数据，每个geneID可能有数千个值；为了便于理解这个算法，这里只使用简单的模拟数据），参考类型为r1，基因过滤器类型为up，过滤器阈值为1.04：

```
GENE-1,r1;p100,1.00
GENE-1,r1;p100,1.06
GENE-1,r1;p100,1.10
GENE-1,r1;p100,1.20

GENE-1,r1;p200,1.00
GENE-1,r1;p200,1.02
GENE-1,r1;p200,1.04

GENE-1,r1;p300,1.01
GENE-1,r1;p300,1.06
GENE-1,r1;p300,1.08
```

根据按单个基因值过滤还是按基因值平均值过滤，可以得到不同的值：

- 如果按单个基因值过滤，会得到以下输出：

  ```
  GENE-1,r1    6
  ```

  因为只有6个记录的基因值大于或等于1.04，这些记录能通过过滤。

- 如果按基因值的平均值过滤，会得到以下输出：

  ```
  GENE-1,r1    2
  ```

  因为对于病人p100，值的平均值$\left(\frac{1.00+1.06+1.10+1.20}{4}\right)$为1.09，这大于1.04（可以通过过滤），对于病人p200，值的平均值$\left(\frac{1.00+1.02+1.04}{3}\right)$为1.02（未通过过滤），对于病人p300，值的平均值$\left(\frac{1.01+1.06+1.08}{3}\right)$为1.05（可以通过过滤）。因此，按平均值过滤时，只有两个病人可以通过测试，频度为2。

我们提供了两个不同的MapReduce解决方案，分别处理按单个基因值过滤和按基因值平均值过滤。

如何将这3个动态参数（参考类型、基因过滤器类型和过滤器阈值）从MapReduce驱动器传递到map()和reduce()函数？通过使用MapReduce/Hadoop Configuration对象，就可以利用Configuration.set()设置这些值，然后在map()或reduce()的setup()函数中利用Configuration.get()获取这些值。

映射器得到一个bioset的一个记录，对输入完成词法分析，得到<geneID><,><referenceType>和geneValue。如果<geneID><,><referenceType>包含所需的引用类型，则检查geneValue，查看它是否满足过滤器阈值条件。如果两个条件都满足，就发出键-值对，其中键是<geneID><,><referenceType>，值是整数1。归约器会累加特定<geneID><,><referenceType>的频度，并发出一个键-值对，其中键是<geneID><,><referenceType>，值是最终的频度计数。

## 映射器：按单个值过滤

对于按单个值过滤的解决方案，映射器的setup()函数（在示例26-1中定义）将获取map()函数中要使用的参数（也就是基因聚合所需要的度量），map()函数的定义见示例26-2。 checkFilter()函数的定义如示例26-3所示。

示例26-1：基因聚合器：setup()函数（按单个值过滤）

```
public class GeneAggregatorMapperByIndividual ... {

  private String referenceType = null;
  private String filterType = null;
  private double filterValueThreshold;

/**
 * 只运行一次
 */
public void setup(Context context) {
   Configuration conf = context.getConfiguration();
   this.referenceType = conf.get("gene.reference.type");
   this.filterType = conf.get("gene.filter.type");
   this.filterValueThreshold =
       Double.parseDouble(conf.get("gene.filter.value.threshold"));
}
```

示例26-2：基因聚合器：map()函数（按单个值过滤）

```
/**
 * @param key是MapReduce分区器生成的键（在这里忽略）
 * @param value是一个bioset的一个记录：
 * <geneID><,><referenceType><;><patientID><,><geneValue>
 */
map(Long key, String value) {
   String[] tokens = StringUtil.split(value, ";");
   String geneIDAndReferenceType = tokens[0];
   String patientIDAndGeneValue = tokens[1];
   String[] val = StringUtil.split(patientIDAndGeneValue, ",");
   // val[0] = patientID
   // val[1] = geneValue
   double geneValue = Double.parseDouble(val[1]);

   String[] arr = StringUtil.split(geneIDAndReferenceType, ",");
   // arr[0] = geneID
   // arr[1] = referenceType
```

```
18    // 检查referenceType
19    if (arr[1].equals(this.referenceType)) {
20       if (checkFilter(geneValue)) {
21          // 然后为归约器创建一个计数器；
22          // 否则不写任何结果。
23          // 为归约器准备键-值对，并发送到归约器。
24          emit(geneIDAndReferenceType, 1);
25       }
26    }
27 }
```

示例26-3：基因聚合器：checkFilter ()函数（按单个值过滤）

```
1 public boolean checkFilter(double value) {
2    if (filterType.equals("abs")) {
3       if (Math.abs(value) >= this.filterValueThreshold) {
4          return true;
5       }
6       else {
7          return false;
8       }
9    }
10   if (filterType.equals("up")) {
11      if (value >= this.filterValueThreshold) {
12         return true;
13      }
14      else {
15         return false;
16      }
17   }
18   if (filterType.equals("down")) {
19      if (value <= this.filterValueThreshold) {
20         return true;
21      }
22      else {
23         return false;
24      }
25   }
26   return false;
27 }
```

由于filterType是从MapReduce驱动器传递到map()，所以在map()开始之前就能知道filterType的值。因此，在我们的MapReduce/Hadoop实现中，可以使用定制映射器来避免每个map()中执行太多次if语句。需要说明，在我们的MapReduce解决方案中，为了保证可读性，这里没有加入异常检查和处理。

## 归约器：按单个值过滤

归约器会接收一个（key，List<Integer>）对，其中key是唯一的geneAndReferenceType，List<Integer>是唯一geneAndReferenceAsString的部分

频度。归约器的任务是累计唯一geneAndReferenceType的出现次数。示例26-4给出了reduce()函数的定义。

示例26-4：基因聚合器：reduce()函数（按单个值过滤）

```
/**
 * @param key为是映射器生成的geneAndReferenceAsString
 * @param values是一个整数列表
 *  （唯一geneIDAndReferenceType的部分计数）
 */
reduce(String key, List<Integer> values) {
   int sum = 0;
   for (int count : values) {
      sum += count;
   }
   emit(key, sum);
}
```

## 映射器：按平均值过滤

对于按平均值过滤的解决方案，映射器的setup()函数（在示例26-5中定义）将获取map()函数中需要使用的参数（也就是基因聚合所需要的度量），map()函数的定义见示例26-6。由于这里按每个病人的基因值的平均值过滤，所以除了基因值还要同时将patientID传入归约器。

示例26-5：基因聚合器：setup()函数（按平均值过滤）

```
public class GeneAggregatorMapperByIndividual ... {

   private String referenceType = null;
   private String filterType = null;
   private double filterValueThreshold;

   /**
    * 只运行一次
    */
   public void setup(Context context) {
      Configuration conf = context.getConfiguration();
      this.referenceType = conf.get("gene.reference.type");
      this.filterType = conf.get("gene.filter.type");
      this.filterValueThreshold =
         Double.parseDouble(conf.get("gene.filter.value.threshold"));
}
```

示例26-6：基因聚合器：map()函数（按平均值过滤）

```
/**
 * @param key是MapReduce分区器生成的键（在这里忽略）
 * @param value是一个bioset的一个记录：
 * <geneID><,><referenceType><;><patientID><,><geneValue>
 */
map(Long key, String value) {
   String[] tokens = StringUtil.split(value, ";");
```

```
 8    String geneIDAndReferenceType = tokens[0];
 9    String patientIDAndGeneValue = tokens[1];
10
11    String[] arr = StringUtil.split(geneIDAndReferenceType, ",");
12    // arr[0] = geneID
13    // arr[1] = referenceType
14    // 检查referenceType
15    if (arr[1].equals(this.referenceType)) {
16       // 为归约器准备键-值对，并发送到归约器
17       emit(geneIDAndReferenceType, patientIDAndGeneValue);
18    }
19 }
```

由于filterType是从MapReduce驱动器传递到map()，所以在map()开始之前就能知道filterType的值。因此，在我们的MapReduce/Hadoop实现中，可以使用定制映射器来避免每个map()中执行太多次if语句。需要说明，在我们的MapReduce解决方案中，为了保证可读性，这里没有加入异常检查和处理。

## 归约器：按平均值过滤

归约器会接收一个（K, List<V>）对，其中K是唯一的geneIDAndReferenceType，V是一个patientIDAndGeneValue。归约器的任务是通过过滤基因值的平均值累计唯一geneIDAndReferenceType的出现次数。

示例26-7定义了reduce()函数。

示例26-7：基因聚合器：reduce()函数（按平均值过滤）

```
 1 /**
 2  * @param key是映射器生成的唯一geneIDAndReferenceType
 3  * @param values是一个List<V> ，V是一个patientIDAndGeneValue
 4  */
 5 reduce(String key, List<String> values) {
 6       Map<String, Tuple2<Double,Integer>> patients =
 7          GeneAggregatorUtil.buildPatientsMap(values);
 8       int passedTheTest = GeneAggregatorUtil.getNumberOfPatientsPassedTheTest(
 9            patients,
10            this.filterType,
11            this.filterValueThreshold
12       );
13
14       // 发出归约器的输出
15       emit(key, passedTheTest);
16 }
```

## 计算基因聚合

计算基因聚合由一个名为GeneAggregatorUtil的工具类处理。这个类有两个static方法：

buildPatientsMap()

这个方法的定义见示例26-8，它接收一个List<Tuple2<patientID, geneValue>>，并构建一个Map<patientID, sumOfGeneValues>。

getNumberOfPatientsPassedTheTest()

这个方法的定义如示例26-9所示，它会统计多少个病人通过测试（也就是满足过滤器条件）。这是由buildPatientsMap()方法检查Map构建的各个记录来实现的。

另外，我们还使用了一个简单的PairOfDoubleInteger类表示Tuple2<Double,Integer>，并允许更新它的值。

示例26-8：GeneAggregatorUtil.buildPatientsMap()

```
/**
 * @param values = List<Tuple2<patientID, geneValue>>
 *
 * 结果Map:
 * 确保同一个病人不会统计多次
 *   patients.key = patientID
 *   patients.value = Tuple2<D, I>
 *      在这里 D = 对应patientID的值总和
 *             I = 值的个数（计数器）
 */
public static Map<String, PairOfDoubleInteger> buildPatientsMap(
   Iterable<Text> values) throws IOException, InterruptedException {
   Map<String, PairOfDoubleInteger> patients =
      new HashMap<String, PairOfDoubleInteger>();
   for (Text patientIdAndGeneValue : values) {
      String[] tokens =
         StringUtils.split(patientIdAndGeneValue.toString(), ",");
      String patientID = tokens[0];
      //tokens[1] = geneValue
      double geneValue = Double.parseDouble(tokens[1]);
      PairOfDoubleInteger pair = patients.get(patientID);
      if (pair == null) {
         pair = new PairOfDoubleInteger(geneValue, 1);
         patients.put(patientID, pair);
      }
      else {
         pair.increment(geneValue);
      }
   }
   return patients;
}
```

示例26-9：GeneAggregatorUtil.getNumberOfPatientsPassedTheTest()

```
public static int getNumberOfPatientsPassedTheTest(
        Map<String, PairOfDoubleInteger> patients,
        double filterValue,
        String filterType) { // filterType = {"up", "down", "abs"}
    if (patients == null) {
       return 0;
```

```
    }

    // 现在来计算值的平均值，并查看
    // 哪些病人通过阈值条件
    int passedTheTest = 0;
    for (Map.Entry<String, PairOfDoubleInteger>
        entry : patients.entrySet()) {
        //String patientID = entry.getKey();
        PairOfDoubleInteger pair = entry.getValue();
        double avg = pair.avg();
        if (filterType.equals("up")) {
           if (avg >= filterValue) {
               passedTheTest++;
           }
        }
        if (filterType.equals("down")) {
           if (avg <= filterValue) {
               passedTheTest++;
           }
        }
        else if (filterType.equals("abs")) {
           if (Math.abs(avg) >= filterValue) {
               passedTheTest++;
           }
        }
    }
    return passedTheTest;
}
```

## Hadoop实现类

我们为基因聚合提供了一个简单的MapReduce解决方案。下面的类将用来实现这个MapReduce Hadoop解决方案。

`GeneAggregationDriverByIndividual`
: 设置输入/输出并启动作业的驱动器。

`GeneAggregationMapperByIndividual`
: 按单个值过滤的映射器。

`GeneAggregationReducerByIndividual`
: 按单个值过滤的归约器。

`GeneAggregationDriverByAverage`
: 设置输入/输出并启动作业的驱动器。

`GeneAggregationMapperByAverage`
: 按平均值过滤的映射器。

GeneAggregationReducerByAverage

按平均值过滤的归约器。

为了得到更高的性能，我们将使用不同的类处理各个过滤器类型（up, down, abs）。因此，GeneAggregationMapperByIndividual可以替换为以下3个插件类：

GeneAggregationMapperByIndividualUP

(filterType = up时)

GeneAggregationMapperByIndividualDOWN

(filterType = down时)

GeneAggregationMapperByIndividualABS

(filterType = abs时)

需要说明，为了提高性能，我提供了3个映射器（分别对应各个不同的基因过滤器类型）。这样就可以避免if语句带来的基因过滤器类型检查。例如，如果处理20000个bioset，每个bioset有超过40000个geneID，就可以避免执行8亿次不必要的if语句。

类似地，GeneAggregationReducerByAverage归约器类也可以替换为以下3个插件类来进行优化：

GeneAggregationReducerByAverageUP

(filterType = up时)

GeneAggregationReducerByAverageDOWN

(filterType = down时)

GeneAggregationReducerByAverageABS

(filterType = abs时)

类似于映射器插件类，这些定制归约器插件类可以避免不必要的if语句检查。

按单个值检查基因值时，可以在驱动器类中设置这些映射器插件类，如示例26-10所示。

示例26-10：GeneAggregationDriverByIndividual类

```
1 public class GeneAggregationDriverByIndividual extends Configured
2    implements Tool {
3    String filterType = ...;
4
5    public int run(String[] args) throws Exception {
6       Job job = ...;
7       ...
8      // 设置优化的映射器类
9      if (filterType.equals("up")) {
10        job.setMapper(GeneAggregationMapperByIndividualUP.class);
11     else if (filterType.equals("down")) {
```

```
12          job.setMapper(GeneAggregationMapperByIndividualDOWN.class);
13       }
14       else if (filterType.equals("abs")) {
15          job.setMapper(GeneAggregationMapperByIndividualABS.class);
16       }
17       else {
18          throw new Exception("filterType is undefined");
19       }
20
21      // 设置归约器类
22      job.setReducer(GeneAggregationReducerByIndividualDOWN.class);
23       ...
24    }
25 }
```

按平均值检查基因值时，可以在驱动器类中设置这些归约器插件类，如示例26-11所示。

示例26-11：GeneAggregationDriverByAverage类

```
1 public class GeneAggregationDriverByAverage extends Configured implements Tool {
2    String filterType = ...;
3
4    public int run(String[] args) throws Exception {
5       Job job = ...;
6       ...
7       // 设置映射器类
8       job.setMapper(GeneAggregationMapperByAverage.class);
9
10      // 设置优化的归约器类
11      if (filterType.equals("up")) {
12         job.setReducer(GeneAggregationReducerByAverageUP.class);
13      else if (filterType.equals("down")) {
14         job.setReducer(GeneAggregationReducerByAverageDOWN.class);
15      }
16      else if (filterType.equals("abs")) {
17         job.setReducer(GeneAggregationReducerByAverageABS.class);
18      }
19      else {
20         throw new Exception("filterType is undefined");
21      }
22      ...
23   }
24 }
```

接下来，checkFilter()方法分解为3部分（每个映射器类对应一部分），如示例26-12～示例26-14所示。由于对应每个filterType有一个定制映射器插件类，所以不再需要if语句。现在，每个映射器类都知道具体的filterType值。

示例26-12：GeneAggregationMapperByIndividualUP类的checkFilter()

```
1 public boolean checkFilter(double value) {
2    //if (filterType.equals("up")) {
3       if (value >= this.filterValueThreshold) {
4          return true;
```

```
5       }
6       else {
7          return false;
8       }
9    //}
10 }
```

示例26-13：GeneAggregationMapperByIndividualDOWN类的checkFilter()

```
1 public boolean checkFilter(double value) {
2    //if (filterType.equals("down")) {
3       if (value <= this.filterValueThreshold) {
4          return true;
5       }
6       else {
7          return false;
8       }
9    //}
10 }
```

示例26-14：GeneAggregationMapperByIndividualABS类的checkFilter()

```
1 public boolean checkFilter(double value) {
2    //if (filterType.equals("abs")) {
3       if (Math.abs(value) >= this.filterValueThreshold) {
4          return true;
5       }
6       else {
7          return false;
8       }
9    //}
10 }
```

## 输出分析

这个MapReduce/Hadoop解决方案的输出（在HDFS中保存为一个SequenceFile，也就是二进制键-值对）有以下格式：

```
<geneID><,><referenceType><TAB><frequency-count>
```

例如：

```
7562135,r1    1205
```

或：

```
7570769,r1    14067
```

很容易读取这些输出文件，将值传递到Java对象中。

FrequencyItem接口如示例26-15所示，可以用来返回基因聚合的结果。

示例26-15：FrequencyItem接口

```
public interface FrequencyItem {
    public Integer getGeneId();
    public void setGeneId(Integer id);

    public Integer getFrequency();
    public void setFrequency(Integer frequency);

    public String getReferenceType();
    public void setReferenceType(String referenceType);

    public String toString();
}
```

然后调用以下代码，其中dir是Hadoop输出路径：

```
List<FrequencyItem> list = readDirectoryIntoFrequencyItem(dir);
```

GeneAggregatorAnalyzerUsingFrequencyItem类在示例26-16～示例26-20中定义。这个类读取HDFS的输出目录，将结果作为一个List<FrequencyItem>对象返回。

示例26-16：GeneAggregatorAnalyzerUsingFrequencyItem类

```
GeneAggregatorAnalyzerUsingFrequencyItem {

  static List<FrequencyItem> readDirectoryIntoFrequencyItem(String pathAsString) {
     return readDirectoryIntoFrequencyItem(new Path(pathAsString));
  }
  static List<FrequencyItem> readDirectoryIntoFrequencyItem(Path path) {
     // 定义见示例26-17
  }

  @SuppressWarnings("unchecked")
  public static List<FrequencyItem> readFileIntoFrequencyItem(Path path,
                                                              FileSystem fs) {
     // 定义见示例26-18
  }

  private static FrequencyItem createFrequencyItem(Text key,
                                                   IntWritable frequency) {
     // 定义在示例26-19中提供
  }

  static void closeAndIgnoreException(SequenceFile.Reader reader) {
      // 定义见示例26-20
  }

  public static void main(String[] args) throws Exception {
     //test1(args);
     Path path = new Path(args[0]);
     List<FrequencyItem> list = readDirectoryIntoFrequencyItem(path);
     THE_LOGGER.info("list="+list.toString());
  }
}
```

示例26-17：readDirectoryIntoFrequencyItem()方法

```
static List<FrequencyItem> readDirectoryIntoFrequencyItem(Path path) {
    FileSystem fs = null;
    try {
        fs = FileSystem.get(new Configuration());
    }
    catch (IOException e) {
        THE_LOGGER.error("Unable to access the hadoop file system!", e);
        throw new RuntimeException("Unable to access the hadoop file system!");
    }

    List<FrequencyItem> list = new ArrayList<FrequencyItem>();
    try {
        FileStatus[] status = fs.listStatus(path);
        for (int i = 0; i < status.length; ++i) {
           Path hdfsFile = status[i].getPath();
           if (hdfsFile.getName().startsWith("part")) {
             List<FrequencyItem> pairs = readFileIntoFrequencyItem(hdfsFile, fs);
             list.addAll(pairs);
           }
        }
    }
    catch (IOException e) {
        THE_LOGGER.error("Unable to access the hadoop file system!", e);
        throw new RuntimeException("Error reading the hadoop file system!");
    }
    return list;
}
```

示例26-18：readFileIntoFrequencyItem()方法

```
@SuppressWarnings("unchecked")
public static List<FrequencyItem> readFileIntoFrequencyItem(Path path,
                                                            FileSystem fs) {
    List<FrequencyItem> list = new ArrayList<FrequencyItem>();
    SequenceFile.Reader reader = null;
    try {
        reader = new SequenceFile.Reader(fs, path, fs.getConf());
        Text key = (Text) reader.getKeyClass().newInstance();
        IntWritable value = (IntWritable) reader.getValueClass().newInstance();
        while (reader.next(key, value)) {
           list.add(createFrequencyItem(key, value));
           key = (Text) reader.getKeyClass().newInstance();
           value = (IntWritable) reader.getValueClass().newInstance();
        }
    }
    catch (Exception e) {
        THE_LOGGER.error("Error reading SequenceFile " + path, e);
        throw new RuntimeException("Error reading SequenceFile " + path);
    }
    finally {
       closeAndIgnoreException(reader);
    }
    return list;
}
```

示例26-19：createFrequencyItem()方法

```
private static FrequencyItem createFrequencyItem(Text key,
                                                  IntWritable frequency) {
    // key = <geneID><,><referenceType> where referenceType in {r1, r2, r3, r4}
    // value = integer
    if (key == null) {
       return null;
    }
    String geneIDAndReference = key.toString();
    String[] tokens = StringUtils.split(geneIDAndReference, ",");
     String geneID = tokens[0];
     String referenceType = tokens[0];
     FrequencyItem item = FrequencyItemFactory.createFrequencyItem(
          geneID, referenceType, frequency.get());
     return item;
}
```

示例26-20：closeAndIgnoreException()方法

```
static void closeAndIgnoreException(SequenceFile.Reader reader) {
    if (reader == null) {
         return;
    }

    try {
        reader.close();
    }
    catch(Exception ignore) {
        THE_LOGGER.error("Error closing SequenceFile.Reader.", ignore);
    }
}
```

# 基因聚合的Spark解决方案

下面各小节提供了基因聚合的两个Spark解决方案。可以从HDFS输入文件创建RDD，并在Spark中处理。Spark的主要数据表示是RDD，它支持数据的并行操作（如`map()`、`reduce()`和`groupByKey()`函数）。Spark可以读取和保存不同来源的RDD（包括HDFS和Linux文件系统）。另外还可以从Java集合对象创建RDD。

与Hadoop解决方案类似，我们将为基因聚合提供两个不同的Spark解决方案（这两种情况下，都是用一个Java驱动器类提供整个Spark解决方案）：

- 按单个基因值过滤的解决方案由`SparkGeneAggregationByIndividual`类实现。
- 按基因值平均值过滤的解决方案由`SparkGeneAggregationByAverage`类实现。

# Spark解决方案：按单个值过滤

按单个值过滤的第一步是为所有输入bioset创建一个`JavaRDD<String>`，其中RDD的

各项分别是bioset的一个记录。第二步过滤掉不满足参考类型和过滤器阈值的值。第三步将（由第二步生成的）RDD项映射为JavaPairRDD<K,V>对，其中K是一个<geneID><,><referenceType>，V是整数1（类似经典的单词计数示例）。最后一步是按键K分组。

## 在集群节点间共享数据

现在的问题是如何将3个度量值（referenceType、filterType和filterValueThreshold）传递到Spark的动作和变换来处理RDD。Spark提供了Broadcast<T>[注1]类来支持在所有集群节点之间共享只读变量。在MapReduce/Hadoop中，映射器和归约器之间通过Hadoop Configuration对象共享变量。（向MapReduce框架提交作业的）驱动器类可以设置/定义共享变量，map()或reduce()函数可以使用这些共享变量。共享是通过映射器或归约器的setup()函数实现的，对于每个映射器或归约器只会调用一次setup()函数。另外，Hadoop中还可以使用DistributedCache[注2]。

在Spark中，我们为3个度量值定义了3个Broadcast对象；需要这些度量值来处理RDD时（使用map()和reduce()函数），我们会读取这些对象。这些Broadcast对象的定义如下所示：

```
JavaSparkContext ctx = new JavaSparkContext();
...
Broadcast<String> broadcastVarReferenceType =
   ctx.broadcast(referenceType);
Broadcast<String> broadcastVarFilterType =
   ctx.broadcast(filterType);
Broadcast<Double> broadcastVarFilterValueThreshold =
   ctx.broadcast(filterValueThreshold);
```

需要这些度量值时，可以如下使用/读取：

```
String referenceType = (String) broadcastVarReferenceType.value();
String filterType = (String) broadcastVarFilterType.value();
Double filterValueThreshold =
   (Double) broadcastVarFilterValueThreshold.value();
```

使用Broadcast<T>的一般形式定义如下：

---

注1：org.apache.spark.broadcast.Broadcast。Spark的Broadcast变量允许程序员在各个集群节点上缓存只读变量，而不是用MapReduce任务传递副本。Broadcast变量采用一种很高效的方式为每个集群节点提供输入数据集的一个副本。Spark会尝试采用高效的广播算法发布Broadcast变量来减少通信开销。

注2：org.apache.hadoop.filecache.DistributedCach是MapReduce框架提供的一个工具，可以缓存应用需要的文件（文本文件、归档文件、JAR等）。

定义：

```
JavaSparkContext ctx = new JavaSparkContext();
T t = <some-data-structure-of-type-T>;
final Broadcast<T> broadcastVariable = ctx.broadcast(t);
```

使用：

```
T t = (T) broadcastVariable.value();
```

Spark API指出，创建Broadcast变量之后，集群上运行的所有函数都应当使用这些变量，而不是值t，这样就不用多次为节点传递t。

## 高层步骤

Spark解决方案是一个Java驱动器类，将处理多个RDD。这一节会提供这个Java驱动器类的主要步骤（参见示例26-21），接下来会介绍各个步骤的详细内容。

示例26-21：SparkGeneAggregationByIndividual类

```
1 //步骤1: 导入所需的类和接口
2 public class SparkGeneAggregationByIndividual {
3
4    public static void main(String[] args) throws Exception {
5       //步骤2: 处理输入参数
6       //步骤3: 创建一个Spark上下文对象 (ctx)
7       //步骤4: 广播共享变量
8       //步骤5: 由所有bioset文件创建一个JavaRDD
9       //步骤6: 将bioset记录映射为JavaPairRDD<K,V>对
10      //       在这里K = "<geneID><,><referenceType>", V = 1
11       //步骤7: 过滤掉冗余的RDD元素
12       //步骤8: 按键归约并累计频度数
13       //步骤9: 准备最终的输出
14
15       // 完成
16       ctx.close();
17       System.exit(0);
18   }
19 }
```

### 步骤1：导入所需的类和接口

要使用RDD，首先需要Spark的org.apache.spark.api.java包。org.apache.spark.api.java.function包定义了动作和变换的函数。另外，我们还需要org.apache.spark.broadcast.Broadcast类来定义共享全局数据结构并在所有集群节点之间广播。

示例26-22：步骤1: 导入所需的类和接口

```
1 //步骤1: 导入所需的类和接口
2 import scala.Tuple2;
3 import org.apache.spark.api.java.JavaPairRDD;
4 import org.apache.spark.api.java.JavaRDD;
```

```
 5 import org.apache.spark.api.java.JavaSparkContext;
 6 import org.apache.spark.api.java.function.Function;
 7 import org.apache.spark.api.java.function.Function2;
 8 import org.apache.spark.api.java.function.PairFunction;
 9 import org.apache.commons.lang.StringUtils;
10 import org.apache.spark.broadcast.Broadcast;
11
12 import java.util.Arrays;
13 import java.util.List;
14 import java.util.ArrayList;
15 import java.io.FileReader;
16 import java.io.BufferedReader;
```

### 步骤2：处理输入参数

这个步骤如示例26-23所示，将读取基因聚合的3个参数（referenceType，filterType，filterValueThreshold）以及分析中涉及的所有bioset。每个bioset由一个HDFS文件标识。

示例26-23：步骤2: 处理输入参数

```
1     //步骤2: 处理输入参数
2     if (args.length != 4) {
3        System.err.println("Usage: SparkGeneAggregationByIndividual "
4                           + "<referenceType><filterType> "
5                           + "<filterValueThreshold> <biosets>");
6        System.exit(1);
7     }
8
9     final String referenceType = args[0];   // {"r1", "r2", "r3", "r4"}
10    final String filterType = args[1];      // {"up", "down", "abs"}
11    final Double filterValueThreshold = new Double(args[2]);
12    final String biosets = args[3];
13
14    System.out.println("args[0]: <referenceType>="+referenceType);
15    System.out.println("args[1]: <filterType>="+filterType);
16    System.out.println("args[2]: <filterValueThreshold>="+filterValueThreshold);
17    System.out.println("args[3]: <biosets>="+biosets);
```

### 步骤3：创建一个Spark上下文对象

这个步骤如示例26-24所示，这一步会创建JavaSparkContext对象，创建RDD时要使用这个对象。可以采用很多不同方式来创建这个上下文对象。

示例26-24：步骤3: 创建一个Spark上下文对象

```
1     //步骤3: 创建一个Spark上下文对象
2     JavaSparkContext ctx = new JavaSparkContext();
```

### 步骤4：广播共享变量

如前所述，Spark允许通过Broadcast<T>类在集群节点之间共享变量和数据结构，其中 T

是共享数据的类型（参见示例26-25）。Spark提供了一个高效的机制来广播共享变量和对象。可以在Spark的map()和reduce()函数中访问共享变量。

示例26-25：步骤4: 广播共享变量

```
1 //步骤4: 广播共享变量
2 final Broadcast<String> broadcastVarReferenceType = ctx.broadcast(referenceType);
3 final Broadcast<String> broadcastVarFilterType = ctx.broadcast(filterType);
4 final Broadcast<Double> broadcastVarFilterValueThreshold =
5   ctx.broadcast(filterValueThreshold);
```

## 步骤5：为bioset创建JavaRDD

基因聚合分析的主要数据来自于bioset文件。这一步如示例26-26所示，会为分析中涉及的所有bioset记录创建一个JavaRDD<String>。这里还包括一个调试步骤。

示例26-26：步骤5: 由所有bioset文件创建一个JavaRDD

```
1     //步骤5: 由所有bioset文件创建一个JavaRDD
2     JavaRDD<String> records = readInputFiles(ctx, biosets);
```

调试这个步骤的代码如下所示：

```
1 // debug1
2 List<String> debug1 = records.collect();
3 for (String rec : debug1) {
4    System.out.println("debug1 => "+ rec);
5 }
```

## 步骤6：将bioset记录映射为JavaPairRDD<K,V>对

这个步骤由各个bioset记录创建一个JavaPairRDD<K,V>对象，其中K="<geneID><,><referenceType>" 而且V = 1。利用这个对象，可以统计bioset表示的所有病人的基因频度。如果一个bioset记录与我们的度量标准不匹配，则创建一个哑对象Tuple2("null", 0)。在最终得到输出之前，这些哑对象将被过滤掉。在Spark中，mapToPair()不允许返回Java null对象，正是因为这个原因，我们要创建类似Tuple2Null的替代对象。

示例26-27：步骤6: 将bioset记录映射为JavaPairRDD<K,V>对

```
1     //步骤6: 将bioset记录映射为JavaPairRDD<K,V>对
2     // where K = "<geneID><,><referenceType>", V = 1
3     JavaPairRDD<String, Integer> genes =
4             records.mapToPair(new PairFunction<String, String, Integer>() {
5         public Tuple2<String, Integer> call(String record) {
6            String[] tokens = StringUtils.split(record, ";");
7            String geneIDAndReferenceType = tokens[0];
8            String patientIDAndGeneValue = tokens[1];
9            String[] val = StringUtils.split(patientIDAndGeneValue, ",");
10           // val[0] = patientID
11           // val[1] = geneValue
12           double geneValue = Double.parseDouble(val[1]);
```

```
13            String[] arr = StringUtils.split(geneIDAndReferenceType, ",");
14            // arr[0] = geneID
15            // arr[1] = referenceType
16            // 检查referenceType和geneValue
17            String referenceType = (String) broadcastVarReferenceType.value();
18            String filterType = (String) broadcastVarFilterType.value();
19            Double filterValueThreshold =
20               (Double) broadcastVarFilterValueThreshold.value();
21
22            if ( (arr[1].equals(referenceType)) &&
23                 (checkFilter(geneValue, filterType, filterValueThreshold)) ) {
24               // 为归约器准备键-值对，并发送到归约器
25               return new Tuple2<String, Integer>(geneIDAndReferenceType, 1);
26            }
27            else {
28               // 否则不会统计
29               // 后面将过滤掉这些"null"键
30               return Tuple2Null;
31            }
32         }
33      });
```

下面的代码可以用来调试步骤6：

```
// debug2
List<Tuple2<String, Integer>> debug2 = genes.collect();
for (Tuple2<String, Integer> pair : debug2) {
   System.out.println("debug2 => key="+ pair._1 + "\tvalue="+pair._2);
}
```

## 步骤7：过滤掉冗余的RDD元素

步骤6会创建冗余的（K,V） = ("null", 0)对。这一步如示例26-28所示，会过滤掉不必要的数据，提供一个干净的JavaPairRDD<String,Integer>。这一步利用Spark的filter()函数实现，定义如下：

```
public JavaPairRDD<K,V> filter(Function<Tuple2<K,V>,Boolean> f)

描述：返回一个新的RDD，其中只包含
      满足谓词条件的元素；
```

示例26-28显示了这个过滤器实现。

示例26-28：步骤7: 过滤掉冗余的RDD元素

```
1     //步骤7：过滤掉冗余的RDD元素
2     // 如果计数器(即V)为0，则将其剔除。
3     JavaPairRDD<String,Integer> filteredGenes =
4           genes.filter(new Function<Tuple2<String,Integer>,Boolean>() {
5        public Boolean call(Tuple2<String, Integer> s) {
6           int counter = s._2;
7           if (counter > 0) {
```

```
            return true;
         }
         else {
            return false;
         }
      }
   });
```

### 步骤8：按键归约并累计频度数

这个步骤如示例26-29所示，实现了reduce()函数。基本说来，这里会按geneID将基因分组，并累计频度数。

示例26-29：步骤8: 按键归约，并累计频度数

```
//步骤8: 按键归约并累计频度数
JavaPairRDD<String, Integer> counts =
    filteredGenes.reduceByKey(new Function2<Integer, Integer, Integer>() {
   public Integer call(Integer i1, Integer i2) {
      return i1 + i2;
   }
});
```

### 步骤9：准备最终的输出

这一步（示例26-30）会发出基因聚合分析的最终输出。

示例26-30：步骤9: 准备最终的输出

```
//步骤9: 准备最终的输出
List<Tuple2<String, Integer>> output = counts.collect();
for (Tuple2<String, Integer> tuple : output) {
    System.out.println("final output => "+ tuple._1 + ": " + tuple._2);
}
```

## 工具函数

示例26-31到示例26-33提供了Spark实现中使用的工具方法。toList()方法接受一个文件，其中包含分析阶段中涉及的所有bioset（见示例26-31）。输入文件的每个记录分别是一个HDFS文件，表示一个bioset。

示例26-31：toList()工具函数

```
/**
 * 将所有bioset文件转换为一个List<biosetFileName>
 *
 * @param biosets是一个文件名，包含所有bioset文件（作为HDFS项）
 * 例如：
 *      /biosets/1000.txt
 *      /biosets/1001.txt
 *      ...
 *      /biosets/1408.txt
 */
```

```
11     private static List<String> toList(String biosets) throws Exception {
12        List<String> biosetFiles = new ArrayList<String>();
13        BufferedReader in = new BufferedReader(new FileReader(biosets));
14        String line = null;
15        while ((line = in.readLine()) != null) {
16           String aBiosetFile = line.trim();
17           biosetFiles.add(aBiosetFile);
18        }
19        in.close();
20        return biosetFiles;
21     }
```

示例26-32：readInputFiles()工具函数

```
1 static JavaRDD<String> readInputFiles(JavaSparkContext ctx,
2                                       String filename)
3    throws Exception {
4    List<String> biosetFiles = toList(filename);
5    int counter = 0;
6    JavaRDD[] rdds = new JavaRDD[biosetFiles.size()];
7    for (String biosetFileName : biosetFiles) {
8       System.out.println("readInputFiles(): biosetFileName=" + biosetFileName);
9       JavaRDD<String> record = ctx.textFile(biosetFileName);
10      rdds[counter] = record;
11      counter++;
12   }
13   JavaRDD<String> allBiosets = ctx.union(rdds);
14   return allBiosets;
15 }
```

示例26-33：checkFilter()工具函数

```
1 static boolean checkFilter(double value,
2                            String filterType,
3                            Double filterValueThreshold) {
4     if (filterType.equals("abs")) {
5         if (Math.abs(value) >= filterValueThreshold) {
6            return true;
7         }
8         else {
9            return false;
10        }
11     }
12     if (filterType.equals("up")) {
13         if (value >= filterValueThreshold) {
14            return true;
15         }
16         else {
17            return false;
18         }
19     }
20     if (filterType.equals("down")) {
21         if (value <= filterValueThreshold) {
22            return true;
23         }
24         else {
```

```
25              return false;
26          }
27      }
28      return false;
29 }
```

# 运行示例

下面各小节将提供YARN环境中这个Spark解决方案运行示例的输入、脚本和日志输出。

## 输入

下面是SparkGeneAggregationByIndividual的输入：

```
# cat biosets.txt
/biosets/b1.txt
/biosets/b2.txt
/biosets/b3.txt
# hadoop fs -cat /biosets/b1.txt
GENE-1,r1;p100,1.00
GENE-1,r1;p100,1.06
GENE-1,r1;p100,1.10
GENE-1,r1;p100,1.20
# hadoop fs -cat /biosets/b2.txt
GENE-1,r1;p200,1.00
GENE-1,r1;p200,1.02
GENE-1,r1;p200,1.04
# hadoop fs -cat /biosets/b3.txt
GENE-1,r1;p300,1.01
GENE-1,r1;p300,1.06
GENE-1,r1;p300,1.08
```

## 脚本

下面是运行SparkGeneAggregationByIndividual的脚本：

```
$ cat run_gene_aggregation_by_individual_yarn.sh
#!/bin/bash
export JAVA_HOME=/usr/java/jdk7
export HADOOP_HOME=/usr/local/hadoop-2.5.0
export HADOOP_CONF_DIR=$HADOOP_HOME/etc/hadoop
export YARN_CONF_DIR=$HADOOP_CONF_DIR
export SPARK_HOME=/home/hadoop/spark-1.1.0
export BOOK_HOME=/home/data-algorithms-book
export APP_JAR=$BOOK_HOME/data_algorithms_book.jar
prog=org.dataalgorithms.chap26.spark.SparkGeneAggregationByIndividual
biosets=$BOOK_HOME/data/biosets.txt
referenceType=r1
#{"r1", "r2", "r3", "r4"}
filterType=up
# {"up", "down", "abs"}
filterValueThreshold=1.04
```

```
$SPARK_HOME/bin/spark-submit --class $prog \
   --master yarn-cluster \
   --num-executors 6 \
   --driver-memory 1g \
   --executor-memory 1g \
   --executor-cores 12 \
   $APP_JAR $referenceType $filterType $filterValueThreshold $biosets
```

### 示例运行日志

```
# ./run_gene_aggregation_by_individual_yarn.sh

args[0]: <referenceType>=r1
args[1]: <filterType>=up
args[2]: <filterValueThreshold>=1.04
args[3]: <biosets>=/home/data-algorithms-book/data/biosets.txt

readInputFiles(): biosetFileName=/biosets/b1.txt
readInputFiles(): biosetFileName=/biosets/b2.txt
readInputFiles(): biosetFileName=/biosets/b3.txt

debug2 => key=null value=0
debug2 => key=GENE-1,r1 value=1
debug2 => key=GENE-1,r1 value=1
debug2 => key=GENE-1,r1 value=1
debug2 => key=null value=0
debug2 => key=null value=0
debug2 => key=GENE-1,r1 value=1
debug2 => key=null value=0
debug2 => key=GENE-1,r1 value=1
debug2 => key=GENE-1,r1 value=1
final output => GENE-1,r1: 6
```

# Spark解决方案：按平均值过滤

这一节采用按平均值过滤（filter by average）的方法为基因聚合提供一个Spark解决方案。按平均值过滤的第一步是为所有输入bioset创建一个JavaRDD<String>，其中RDD的各项分别是bioset的一个记录。第二步是为每一项创建一个JavaPairRDD<K,V>对象，其中K是一个<geneID><,><referenceType>，V是一个<patientID><,><geneValue>。第三步是按K分组。最后一步是按基因的平均值（对应各个病人）得出各个geneID的频度，这些基因应满足所有给定的度量条件。这个例子将展示按平均值过滤的基因聚合的核心。

为了得到对应各个病人的平均值，利用V，我们构建了一个散列表，如下所示：

```
Map(patientID, Tuple2(sum(geneValue), count))
```

接下来，找出各个病人的平均值Map(patientID, average)，其中average = sum(geneValue)/count。如果average通过了过滤器阈值条件，则计为1（表示它通过了测试）；否则，计为0（表示未通过测试）。

## 高层步骤

与前面一样，首先会给出这个Spark解决方案的主要步骤（见示例26-34），在后面的小节中会逐一给出各个步骤的详细信息。

示例26-34：按平均值过滤的基因聚合：高层步骤

```
//步骤1：导入所需的类和接口

public class SparkGeneAggregationByAverage {

    // 将要过滤的哑对象
    static final Tuple2<String, String> Tuple2Null =
       new Tuple2<String, String>("n", "n");

    public static void main(String[] args) throws Exception {
        //步骤2：处理输入参数
        //步骤3：创建一个Java Spark上下文对象（ctx）
        //步骤4：在所有集群节点中共享全局变量
        //步骤5：读取所有bioset记录，并创建一个RDD
        //步骤6：映射bioset记录，并由各个记录创建一个JavaPairRDD<K, V>
        //    其中 K = "<geneID><,><referenceType>",
        //         V = "<patientID><,><geneValue>"
        //步骤7：过滤步骤6创建的冗余记录
        //步骤8：按"<geneID><,><referenceType>"对bioset分组
        //     genesByID = JavaPairRDD<K, List<V>>
        //           其中 K = "<geneID><,><referenceType>"
        //                V = "<patientID><,><geneValue>"
        //步骤9：准备最终所需的输出
        //步骤10：发出最终输出

        // 完成
        ctx.close();
        System.exit(0);
    }
}
```

### 步骤1：导入所需的类和接口

在这个步骤中（如示例26-35所示），要导入按平均值过滤的Spark解决方案所需的类和接口。

示例26-35：步骤1：导入所需的类和接口

```
//步骤1：导入所需的类和接口
import scala.Tuple2;
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.api.java.JavaPairRDD;
import org.apache.spark.api.java.JavaSparkContext;
import org.apache.spark.api.java.function.Function;
import org.apache.spark.api.java.function.PairFunction;
```

```
 8 import org.apache.commons.lang.StringUtils;
 9 import org.apache.spark.broadcast.Broadcast;
10
11 import java.io.FileReader;
12 import java.io.BufferedReader;
13 import java.util.Map;
14 import java.util.HashMap;
15 import java.util.List;
16 import java.util.ArrayList;
```

## 步骤2：处理输入参数

示例26-36展示了如何处理输入参数。

示例26-36：步骤2: 处理输入参数

```
1     //步骤2: 处理输入参数
2     if (args.length != 4) {
3       System.err.println("Usage: SparkGeneAggregationByAverage"+
4         " <referenceType> <filterType> <filterValueThreshold> <biosets>");
5       System.exit(1);
6     }
7
8     final String referenceType = args[0];      // {"r1", "r2", "r3", "r4"}
9     final String filterType = args[1];         // {"up", "down", "abs"}
10    final Double filterValueThreshold = new Double(args[2]);
11    final String biosets = args[3];
12
13    System.out.println("args[0]: <referenceType>="+referenceType);
14    System.out.println("args[1]: <filterType>="+filterType);
15    System.out.println("args[2]: <filterValueThreshold>="+filterValueThreshold);
16    System.out.println("args[3]: <biosets>="+biosets);
```

## 步骤3：创建一个Java Spark上下文对象

需要创建一个JavaSparkContext对象（如示例26-37所示）来创建和管理RDD。创建这个上下文对象有很多方法。有关的详细信息参见Spark API文档。

示例26-37：步骤3: 创建一个Java Spark上下文对象

```
1     //步骤3: 创建一个Java Spark上下文对象
2     JavaSparkContext ctx = new JavaSparkContext();
```

## 步骤4：在所有集群节点中共享全局变量

在Spark中，可以利用Broadcast<T>类在所有集群节点间共享类型为T的数据结构。基本说来，可以在驱动器程序（向Spark或YARN集群提交作业的类）中共享/广播数据结构，然后在map()和reduceByKey()函数中从任意集群节点读取这些数据结构。

示例26-38展示了如何共享我们的3个全局变量。

示例26-38：步骤4: 在所有集群节点中共享全局变量

```
1    //步骤4: 在所有集群节点中共享全局变量
2    final Broadcast<String> broadcastVarReferenceType =
3      ctx.broadcast(referenceType);
4    final Broadcast<String> broadcastVarFilterType =
6      ctx.broadcast(filterType);
7    final Broadcast<Double> broadcastVarFilterValueThreshold =
8      ctx.broadcast(filterValueThreshold);
```

## 步骤5：读取所有bioset记录，并创建一个RDD

这个步骤如示例26-39所示，从HDFS读取所有bioset文件，并创建一个RDD（JavaRDD<String>）。这个功能将在讨论readInputFiles()方法时详细解释。

示例26-39：步骤5: 读取所有bioset记录，并创建一个RDD

```
1    //步骤5: 读取所有bioset记录，并创建一个RDD
2    JavaRDD<String> records = readInputFiles(ctx, biosets);
```

我们使用JavaRDD.collect()方法来调试这个步骤：

```
1 // debug1
2 List<String> debug1 = records.collect();
3 for (String rec : debug1) {
4   System.out.println("debug1 => "+ rec);
5 }
```

## 步骤6：将bioset记录映射到JavaPairRDD <K,V>对

这个步骤如示例26-40所示，将从所有bioset记录创建JavaPairRDD<K,V>对象，其中 K = <geneID><,><referenceType> ，V = <patientID><,><geneValue>。由此可以对bioset表示的所有病人按平均值统计基因的频度。如果一个bioset记录不满足我们的度量条件，则创建一个哑对象（标志为一个null对象）Tuple2("n", "n")，最终返回输出之前将过滤掉这些哑对象。在Spark中，RDD元素不允许是Java null对象，正是因为这个原因，我们创建了类似Tuple2Null的替代对象。

需要说明，在忽略RDD元素方面，Spark的方法与MapReduce/Hadoop的方法有所不同，后者会忽略RDD元素而不发出任何冗余的键-值对。例如，在MapReduce/Hadoop中，如果映射器/归约器输入不满足所要求的条件，可能会生成0个键-值对（也就是说，根本没有输出）。不过在Spark中，mapToPair()方法必须返回一个非null的键-值对（可以使用filter()方法过滤）。

示例26-40：步骤6: 将bioset记录映射到JavaPairRDD <K,V>对

```
1    //步骤6: 将bioset记录映射到JavaPairRDD<K, V>对
2    // where K = "<geneID><,><referenceType>",
3    //       V = "<patientID><,><geneValue>"
4    JavaPairRDD<String, String> genes =
```

```
        records.mapToPair(new PairFunction<String, String, String>() {
      public Tuple2<String, String> call(String record) {
        String[] tokens = StringUtils.split(record, ";");
        String geneIDAndReferenceType = tokens[0];
        String patientIDAndGeneValue = tokens[1];
        String[] arr = StringUtils.split(geneIDAndReferenceType, ",");
        // arr[0] = geneID
        // arr[1] = referenceType
        // 检查referenceType和geneValue
        String referenceType = (String) broadcastVarReferenceType.value();
        if ( arr[1].equals(referenceType) ) {
            // 为归约器准备键-值对并发送到归约器
            return new Tuple2<String, String>(geneIDAndReferenceType,
                                              patientIDAndGeneValue);
        }
        else {
          // 否则不计数
          // 后面会过滤到这些"null"键
          return Tuple2Null;
        }
      }
    });
```

## 步骤7：过滤步骤6创建的冗余记录

步骤6会创建冗余的 (K,V) = Tuple2("n", "n")对来替代Java的null值。这一步会过滤掉这些冗余的数据，提供一个干净的JavaPairRDD<String,String>，这通过实现Spark的filter()函数来完成，定义如下：

```
public JavaPairRDD<K,V> filter(Function<Tuple2<K,V>,Boolean> f)
// 描述：返回一个新RDD，其中只包含
//       满足函数f定义的谓词条件的元素。
```

示例26-41显示了这个过滤器实现。

示例26-41：步骤7: 过滤步骤6创建的冗余记录

```
    //步骤7: 过滤步骤6创建的冗余记录
    // public JavaPairRDD<K,V> filter(Function<Tuple2<K,V>,Boolean> f)
    // 返回一个新RDD，其中只包含满足某个谓词条件的元素;
    // 如果K = "n"，则将其剔除
    JavaPairRDD<String,String> filteredGenes =
        genes.filter(new Function<Tuple2<String,String>,Boolean>() {
      public Boolean call(Tuple2<String, String> s) {
        String value = s._1;
        if (value.equals("n")) {
          // 剔除null项
          return false;
        }
        else {
          return true;
        }
```

```
16      }
17    });
```

## 步骤8：按geneID和referenceType对bioset分组

这个步骤如示例26-42所示，按<geneID><,><referenceType>对所有bioset记录分组。分组后的值可以完成最后的归约，我们将对每个病人按平均值计算基因聚合。

示例26-42：步骤8: 按<geneID><,><referenceType>对Bioset分组

```
1 // 步骤8: 按<geneID><,><referenceType>对bioset分组
2 // genesByID = JavaPairRDD<K, List<V>>
3 //    在这里K = <geneID><,><referenceType>
4 //          V = <patientID><,><geneValue>
5 JavaPairRDD<String, Iterable<String>> genesByID = filteredGenes.groupByKey();
```

## 步骤9：准备最终输出

这一步对所有bioset记录应用“按平均值计算基因聚合”算法。现在对于各个<geneID><,><referenceType>分别有一个<patientID><,><geneValue>列表。这个步骤由mapValues()方法实现，如下所示：

```
mapValues[U](f: (V) => U): JavaPairRDD[K, U]

描述: 通过一个map函数传入
      键-值对RDD中的各个值
      而不改变键;
      仍保留原RDD的分区。
```

这个步骤（如示例26-43所示）使用了Spark的广播全局变量（即共享变量）。可以创建Broadcast<T>来定义共享数据结构（这里T是所需的数据结构）。要使用共享/广播变量，可以使用Broadcast.value()方法（我们就使用这个方法在所有集群节点中读取共享变量：value()方法会返回类型为T的数据结构）。

示例26-43：步骤9: 准备最终所需的输出

```
1 // 步骤9: 准备最终所需的输出
2 JavaPairRDD<String, Integer> frequency =
3   genesByID.mapValues(new Function<Iterable<String>, // 输入
4                                    Integer           // 输出
5                                   >() {
6   public Integer call(Iterable<String> values) {
7       Map<String, PairOfDoubleInteger> patients = buildPatientsMap(values);
8      String filterType = (String) broadcastVarFilterType.value();
9      Double filterValueThreshold =
10         (Double) broadcastVarFilterValueThreshold.value();
11     int passedTheTest = getNumberOfPatientsPassedTheTest(
12          patients,
13         filterType,
```

```
14          filterValueThreshold);
15      return passedTheTest;
16    }
17 });
```

### 步骤10：发出最终输出

这个步骤如示例26-44所示，打印最终的输出。

示例26-44：步骤10: 发出最终输出

```
1 //步骤10: 发出最终输出
2 List<Tuple2<String, Integer>> finalOutput = frequency.collect();
3 for (Tuple2<String, Integer> tuple : finalOutput) {
4     System.out.println("final output => "+ tuple._1 + ": " + tuple._2);
5 }
```

下面各小节会解释采用平均值基因聚合解决方案时过滤器使用的一些重要方法。

## 工具函数

`toList()`方法如示例26-45所示，会收集所有HDFS bioset文件，把它们保存为`List<String>`，其中每个列表元素分别是一个HDFS bioset文件。

示例26-45：toList()支持方法

```
1    /**
2     * 将所有bioset文件转换为一个List<biosetFileName>
3     *
4     * @param biosets是一个文件名，其中包含所有bioset文件（HDFS项）；
5     * 例如：
6     *      /biosets/1000.txt
7     *      /biosets/1001.txt
8     *      ...
9     *     /biosets/1408.txt
10    */
11    private static List<String> toList(String biosets) throws Exception {
12       List<String> biosetFiles = new ArrayList<String>();
13       BufferedReader in = new BufferedReader(new FileReader(biosets));
14       String line = null;
15       while ((line = in.readLine()) != null) {
16          String aBiosetFile = line.trim();
17          biosetFiles.add(aBiosetFile);
18       }
19       in.close();
20       return biosetFiles;
21    }
```

`readInputFiles()`方法如示例26-46所示，会读取所有HDFS bioset文件并创建一个新的RDD（`JavaRDD<String>`），其中每个元素分别是一个bioset记录。通过`JavaSparkContext.union()`方法合并所有bioset记录。然后通过应用`JavaRDD.coalesce()`方法对这个RDD分区，来完成进一步的并行处理。

示例26-46：readInputFiles()支持方法

```
1    private static JavaRDD<String> readInputFiles(JavaSparkContext ctx,
2                                                  String filename)
3       throws Exception {
4        List<String> biosetFiles = toList(filename);
5        int counter = 0;
6        JavaRDD[] rdds = new JavaRDD[biosetFiles.size()];
7        for (String biosetFileName : biosetFiles) {
8          System.out.println("debug1 biosetFileName=" + biosetFileName);
9          JavaRDD<String> record = ctx.textFile(biosetFileName);
10         rdds[counter] = record;
11         counter++;
12       }
13       JavaRDD<String> allBiosets = ctx.union(rdds);
14       return allBiosets.coalesce(9, false);
15   }
```

buildPatientsMap()方法如示例26-47所示，会建立一个Map<String,PairOfDoubleInteger>，其中键是一个patientID，值是一个PairOfDoubleInteger对象，将跟踪基因值总和和计数。

示例26-47：buildPatientsMap()支持方法

```
1 private static Map<String,PairOfDoubleInteger>
2   buildPatientsMap(Iterable<String> values) {
3   Map<String, PairOfDoubleInteger> patients =
4      new HashMap<String, PairOfDoubleInteger>();
5   for (String patientIdAndGeneValue : values) {
6      String[] tokens = StringUtils.split(patientIdAndGeneValue, ",");
7      String patientID = tokens[0];
8      // tokens[1] = geneValue
9      double geneValue = Double.parseDouble(tokens[1]);
10      PairOfDoubleInteger pair = patients.get(patientID);
11      if (pair == null) {
12         pair = new PairOfDoubleInteger(geneValue, 1);
13         patients.put(patientID, pair);
14      }
15      else {
16         pair.increment(geneValue);
17      }
18   }
19   return patients;
20 }
```

getNumberOfPatientsPassedTheTest()方法如示例26-48所示，会迭代处理buildPatientsMap()方法建立的Map<String, PairOfDoubleInteger>。这个映射中的每一项指示一个病人。如果一个病人满足所要求的度量条件（也就是说，基因值的平均值通过了所定义的度量要求filterType和filterValueThreshold），则认为这个病人通过了测试。

示例26-48：getNumberOfPatientsPassedTheTest()支持方法

```
1 private static int getNumberOfPatientsPassedTheTest(
```

```
2           Map<String, PairOfDoubleInteger> patients,
3           String filterType,
4           Double filterValueThreshold) {
5     if (patients == null) {
6         return 0;
7     }
8
9     // 现在计算值的平均值，查看
10    // 哪些病人满足阈值条件
11    int passedTheTest = 0;
12    for (Map.Entry<String, PairOfDoubleInteger> entry : patients.entrySet()) {
13        //String patientID = entry.getKey();
14        PairOfDoubleInteger pair = entry.getValue();
15        double avg = pair.avg();
16        if (filterType.equals("up")) {
17            if (avg >= filterValueThreshold) {
18                passedTheTest++;
19            }
20        }
21        if (filterType.equals("down")) {
22            if (avg <= filterValueThreshold) {
23                passedTheTest++;
24            }
25        }
26        else if (filterType.equals("abs")) {
27           if (Math.abs(avg) >= filterValueThreshold) {
28               passedTheTest++;
29           }
30       }
31    }
32    return passedTheTest;
33 }
```

# 运行示例

下面各小节会给出这个Spark解决方案在YARN环境中的一个运行示例的输入、脚本和日志输出。

## 输入

下面是YARN环境中这个Spark解决方案的输入：

```
# cat biosets.txt
/biosets/b1.txt
/biosets/b2.txt
/biosets/b3.txt
# hadoop fs -cat /biosets/b1.txt
GENE-1,r1;p100,1.00
GENE-1,r1;p100,1.06
GENE-1,r1;p100,1.10
GENE-1,r1;p100,1.20
# hadoop fs -cat /biosets/b2.txt
GENE-1,r1;p200,1.00
```

```
GENE-1,r1;p200,1.02
GENE-1,r1;p200,1.04
# hadoop fs -cat /biosets/b3.txt
GENE-1,r1;p300,1.01
GENE-1,r1;p300,1.06
GENE-1,r1;p300,1.08
```

## 脚本

下面是在YARN中运行SparkGeneAggregationByAverage的脚本：

```
$ cat run_gene_aggregation_by_average_yarn.sh
#!/bin/bash
export JAVA_HOME=/usr/java/jdk7
export HADOOP_HOME=/usr/local/hadoop-2.5.0
export HADOOP_CONF_DIR=$HADOOP_HOME/etc/hadoop
export YARN_CONF_DIR=$HADOOP_CONF_DIR
export SPARK_HOME=/usr/local/spark-1.1.0
export BOOK_HOME=/home/data-algorithms-book
export APP_JAR=$BOOK_HOME/dist/data_algorithms_book.jar
prog=org.dataalgorithms.chap26.spark.SparkGeneAggregationByAverage
biosets=$BOOK_HOME/data/biosets.txt
referenceType=r1
#{"r1", "r2", "r3", "r4"}
filterType=up
# {"up", "down", "abs"}
filterValueThreshold=1.04
$SPARK_HOME/bin/spark-submit --class $prog \
    --master yarn-cluster \
    --num-executors 6 \
    --driver-memory 1g \
    --executor-memory 1g \
    --executor-cores 12 \
    $APP_JAR $referenceType $filterType $filterValueThreshold $biosets
```

## 示例运行日志

运行示例的日志输出如下：

```
# ./run_gene_aggregation_by_average_yarn.sh
args[0]: <referenceType>=r1
args[1]: <filterType>=up
args[2]: <filterValueThreshold>=1.04
args[3]: <biosets>=/home/data-algorithms-book/data/biosets.txt
debug1 biosetFileName=/biosets/b1.txt
debug1 biosetFileName=/biosets/b2.txt
debug1 biosetFileName=/biosets/b3.txt
final output => GENE-1,r1: 2
```

这一章为基因聚合（也就是标志物频度）提供了多个MapReduce和Spark解决方案，另外提供了过滤以及向所有集群节点广播数据结构的Spark高级API。第27章将讨论线性回归，并提供相应的可伸缩分布式解决方案。

# 第27章

# 线性回归

这一章讨论一个非常重要的统计概念：线性回归（linear regression）[注1]。线性回归有很多用途，包括临床应用，如使用病人样本数据的基因组分析。Wikipedia对线性回归有以下描述："线性回归（*http://bit.ly/linear_regression*）在生物、行为和社会科学领域广泛用于描述变量间可能的关系，它是这些学科使用的最为重要的工具之一。"对于小数据，使用线性回归相当简单：有很多现成的Java类可以使用，如Apache Commons的`SimpleRegression`[注2]。不过，由于一个服务器的内存和CPU资源有限，这些类和包无法处理大量的数据。这一章的主要目标就是对大数据集实现线性回归（如对应多个病人的样本数据，相应bioset所表示的基因组数据）。

这一章将提供线性回归的两个不同的MapReduce/Hadoop解决方案：

- 第一个解决方案使用Apache Commons的`SimpleRegression`。
- 第二个解决方案使用R的线性模型实现MapReduce。

Spark提供了机器学习库（Machine Learning Library）包，即MLlib，其中就包含一些线性方法（MLlib目前正在积极开发当中）。

线性回归最常用的形式是最小二乘拟合（least squares fitting）（*http://bit.ly/least_*

注1：线性回归模型可以用来分析响应变量或依赖变量与一组自变量或预测变量之间的关系。这种关系可以表示为一个公式，预测响应变量是参数的一个线性函数。可以调整这些参数来得到最优的拟合度。关于模型拟合的很多工作主要考虑尽可能减少残差大小，以及确保模型预测随机分布（资料来源：*http://en.wikipedia.org/wiki/Predictive_analytics*）。

注2：`org.apache.commons.math3.stat.regression.SimpleRegression`。

*squares_fitting*）。在具体讨论实现线性回归的细节之前，下面先来定义最小二乘拟合是什么，它会告诉我们什么。简单地讲，我们要对一个实际的数据集拟合一个方程（这有些类似于通过观察一个实际数据集预测未来）。

# 基本定义

下面是关于线性回归的一些概念，其中一些描述摘选自*http://bit.ly/linear_regression_intro*：

- 线性回归建模为一个线性方程$y = ax + b$。基本说来，回归分析就是要找出与数据拟合的方程，其中数据表示为一个（$x, y$）集合。
- 线性回归使用了这样一个事实：两个变量之间存在有统计意义的相关性，从而可以根据你对一个变量的知识对另一个变量做出预测。
- 除非相关系数有统计意义，否则不要完成线性回归。
- 要有效地使用线性回归，变量之间必须存在一种线性关系。

接下来我们还需要了解回归和线性回归的定义。回归是一种统计分析，用来评估变量（如回归方程$y = ax + b$中的$x$和$y$）之间的关联。采用最简形式时，回归可以用来查找两个变量x和y之间的关系，不过也可以使用多个变量。因此，线性回归就是在回归方程为线性方程时定义/估计变量之间的关系，例如，$y = ax + b$，其中b是回归线与y轴的截距，a是回归线的斜率。所以，线性回归的主要目的就是对应给定的（$x, y$）集合找出$b$（也就是截距，即$x = 0$时$y$的值)和$a$（斜率）。通常$x$是解释变量，$y$是依赖变量。

# 简单示例

这里我会给出一个简单的例子来说明线性回归的基本概念。线性回归分析的目标是找出与数据拟合的线性方程。一旦得到这个线性方程（$y = ax + b$），就可以使用这个线性回归模型（表示为一个线性方程）做出预测。在这里，我们会展示如何使用样本数据计算线性回归，以及如何找出线性方程$y = ax + b$。这里使用的两个变量是一个很小的病人集合的年龄和血糖水平。首先，为数据建立一个表，如表27-1所示。

**表27-1：病人年龄和血糖水平的数据表**

| 病人 | 年龄(x) | 血糖水平(y) | xy | $x^2$ | $y^2$ |
|---|---|---|---|---|---|
| 1 | 41 | 90 | 3690 | 1681 | 8100 |
| 2 | 42 | 93 | 3906 | 1764 | 8649 |
| 3 | 43 | 98 | 4214 | 1849 | 9604 |
| 4 | 20 | 64 | 1280 | 400 | 4096 |

表27-1：病人年龄和血糖水平的数据表（续）

| 病人 | 年龄(x) | 血糖水平(y) | xy | $x^2$ | $y^2$ |
|---|---|---|---|---|---|
| 5 | 25 | 78 | 1950 | 625 | 6084 |
| 6 | 40 | 71 | 2840 | 1600 | 5041 |
| 7 | 58 | 88 | 5104 | 3364 | 7744 |
| 8 | 60 | 86 | 5160 | 3600 | 7396 |
| 总计 | 329 | 668 | 28144 | 14883 | 56714 |

接下来，使用以下公式来计算$a$和$b$：

$$a=\frac{\left(\sum y\right)\left(\sum x^2\right)-\left(\sum x\right)\left(\sum xy\right)}{n\left(\sum x^2\right)-\left(\sum x\right)^2}$$

$$b=\frac{n\left(\sum xy\right)-\left(\sum x\right)\left(\sum y\right)}{n\left(\sum x^2\right)-\left(\sum x\right)^2}$$

需要说明，$n$是回归的样本数（在这个例子中，$n = 8$）。现在插入预计算的值，可以得到：

$$a = 63.05$$

$$b = 0.497$$

接下来，使用R编程语言绘制线性回归图（见图27-1）：

```
# R
R version 2.15.1 (2012-06-22) -- "Roasted Marshmallows"
> x <- c(41, 42, 43, 20, 25, 40, 58, 60)
> y <- c(90, 93, 98, 64, 78, 71, 88, 86)
> mod1 <- lm(y ~ x)
> plot(x, y, xlim=c(min(x)-5, max(x)+5), ylim=c(min(y)-10, max(y)+10))
> abline(mod1, lwd=2)
```

从这个图中可以观察到，线性回归要根据所观察的数据拟合一个线性方程，来为两个变量（$x$和$y$）间的关系建模（拟合回归线的最常用的算法是最小二乘法）。

# 问题描述

假设有一组病人数据，用来完成某种药物或治疗的临床试验。进一步假设每个病人可能有一组bioset/生物标志物，而且每个bioset/生物标志物有一组键-值对，其中键是Gene-ID，值可以表示段值（segment value）、拷贝变异数（copy change number）或其他相关的基因组数据值（这里的值表示变量$x$，变量$y$是接受这种治疗的病人的生存时间）。

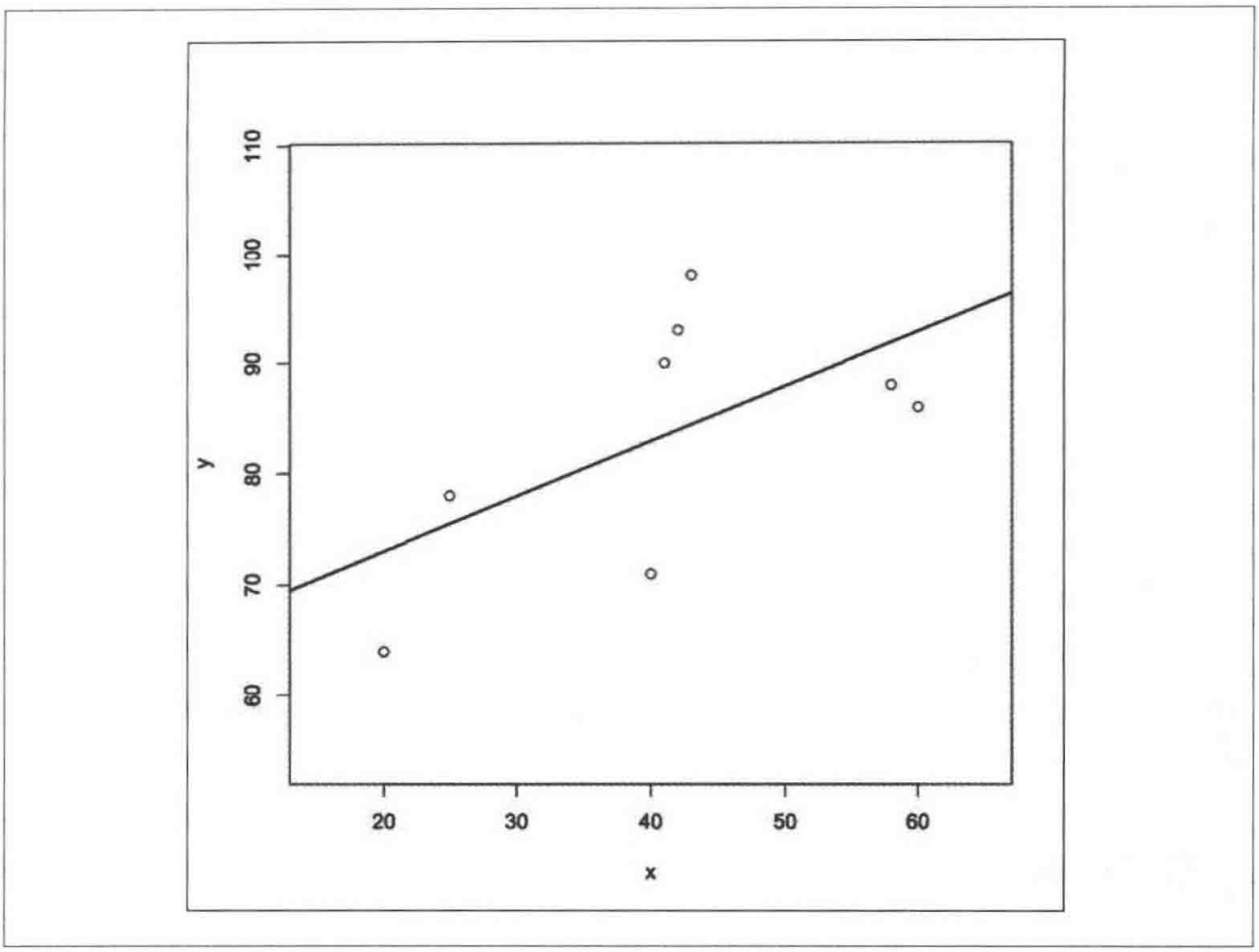



图27-1：用R绘制线性回归图

每个生物标志物的键-值对数取决于bioset的类型（bioset数据类型包括甲基化、拷贝数变异、基因表达式和体细胞突变）。例如，对于基因表达式类型，每个生物标志物可以有多达50000个键-值对。假设希望对40000个bioset完成线性回归：这就会产生20亿个数据点，所以根本无法在一个服务器上完成线性回归计算。在这里，我们感兴趣的是对各个Gene-ID完成线性回归。具体地，我们的目标是找出每个Gene-ID的以下信息：

- 截距（估计的回归线的拦截点）。
- 斜率 （估计的回归线的斜率）。
- 显著性（斜率相关的p值或显著性水平）。

# 输入数据

我们的输入数据包括一组biosets/生物标志物，每个bioset由一个唯一的bioset_id标识。每个bioset包括最多60000个记录，每个记录的格式如下（在这里，我们认为gene_value是变量$x$）：

```
<gene_id><,><reference><;><bioset_id><,><gene_value>
```

<gene_id>是一个基因，表示为一个ID，<reference>可以有以下值：

- r1 = 正常病人的参考值。
- r2 = 患病的参考值。
- r3 = 配对的参考值。
- r4 = 无。

例如：

```
1234,r2;3344550,0.43
```

或：

```
122765,r1;3344550,1.78
```

另外还有一个输入向量（表示变量$y$），如病人的生存时间（这是一个Double数据类型的数组；每一项对应一个bioset）。例如，要分析5000个bioset，则需要传入一个大小为5000的生存时间数组。

# 期望输出

完成线性回归分析之后，分析结果将有以下格式（每个Gene-ID对应一个记录）：

```
<gene_id><,><reference><,><slope><,><intercept><,><pValue>
```

例如：

```
1234,r2,1.002,3.12007,0.000098
```

# 使用SimpleRegression的MapReduce解决方案

我们的线性回归MapReduce解决方案包括一个映射器和一个归约器。map()函数的主要任务是标识变量$x$，把它传递到归约器。每个归约器会接收和处理一个键-值对。键是gene_id和reference，相应的值是gene_value（线性回归中的变量$x$）。在我们的例子中，变量$y$是生存时间，将从驱动器类传递到MapReduce/Hadoop Configuration对象。Reduce()函数从Configuration对象得到$y$值（在Hadoop中，这将由setup()方法完成）。一旦有了$x$和$y$变量，再使用SimpleRegression[注3]类完成线性回归，然后将结果（slope,intercept和pValue）写回到HDFS。使用SimpleRegression类非常简单：

注3：org.apache.commons.math3.stat.regression.SimpleRegression (Apache Commons Math)。

```
// y = intercept + slope * x
SimpleRegression sr = new SimpleRegression();
sr.addData(1d, 2d); // x = 1, y = 2
sr.addData(3d, 3d); // x = 3, y = 3
sr.addData(2d, 4d); // x = 2, y = 4
// 可以增加任意多个sr.addData(x, y)
// 现在所有统计量都已经定义。
double pValue = sr.getSignificance());
double intercept = sr.getIntercept());
double slope = sr.getSlope());
```

需要说明，gene_value表示$x$，还要传递bioset_id和生存时间（表示$y$）。我们要传递bioset_id，因为bioset_id[i]对应survival_time[i]，必须确保使用适当的$x$和$y$来完成线性回归。映射器如示例27-1所示。

示例27-1：线性回归：map()函数

```
/**
 * @param key由Hadoop生成（在这里忽略）
 * @param value的格式：<gene_id><,><reference><;><bioset_id><,><gene_value>
 */
map(key, value) {
   String line = value.toString().trim();
   if ((line == null) || (line.length() == 0)) {
      return;
   }

   String[] tokens = StringUtils.split(line, ";");
   // tokens[0] = <gene_id><,><reference>
   // tokens[1] = <bioset_id><,><gene_value>
   if (tokens.length == 2) {
      // 为归约器准备(key, value)
      emit(tokens[0], tokens[1]);
   }
}
```

归约器类LinearRegressionReducer如示例27-2所示，reduce()函数的定义见示例27-3 。

示例27-2：线性回归的LinearRegressionReducer

```
public class LinearRegressionReducer ... {
   // 实例变量
   private Configuration conf = null;
   private List<Double> time = null;
   // biosetID表示为String: 注意bioset的顺序非常重要
   private List<String> biosets = null;

   // 只运行一次
public void setup(Context context)
    this.conf = context.getConfiguration();
    // 从Hadoop的Configuration对象得到参数
    String biosetsAsString = conf.get("biosets");
     this.biosets =
        DataStructuresUtil.splitOnToListOfString(biosetsAsString, ",");
```

```
15      String timeAsCommaSeparatedString = conf.get("time");
16      this.time = DataStructuresUtil.splitOnToListOfDouble(
17         timeAsCommaSeparatedString, ",");
18   }
19
20    // key = <gene_id><,><reference>
21    // values = { bioset_id,value }
22    // biosets: B1, B2, B3, ..., Bn
23    // times: T1, T2, T3, ..., Tn
24    public void reduce(Text key, Iterable<Text> values) {
25       // 见示例27-3 ...
26    }
27 }
```

示例27-3：线性回归：reduce()函数

```
 1 public class LinearRegressionReducer ... {
 2     ...
 3     // key = <gene_id><,><reference>
 4     // values = { bioset_id,value }
 5     // biosets: B1, B2, B3, ..., Bn
 6     // times: T1, T2, T3, ..., Tn
 7     public void reduce(Text key, Iterable<Text> values) {
 8        int numberOfValues = 0;
 9        SimpleRegression sr = new SimpleRegression();
10        Iterator<Text> iter = values.iterator();
11        while (iter.hasNext()) {
12           Text pairAsText = iter.next();
13           if (pairAsText == null) {
14              continue;
15           }
16           String pairAsString = pairAsText.toString();
17           String[] tokens = StringUtils.split(pairAsString, ",");
18           // biosetID = tokens[0]
19           // value = tokens[1]
20           if (tokens.length != 2) {
21              // 则回归中忽略这个(value, time)
22              continue;
23           }
24
25           int index = biosets.indexOf(tokens[0]);
26           if (index == -1) {
27              // biosetID未找到
28              continue;
29           }
30
31           // 在index找到biosetID
32           // sr.addData(xPos.get(i), yPos.get(i));
33           double dvalue = Double.parseDouble(tokens[1]);
34           sr.addData(dvalue, time.get(index));
35           numberOfValues++;
36        }
37
38        if (numberOfValues > 0) {
39           StringBuilder builder = new StringBuilder();
```

```
40          builder.append(key.toString()); // gene_id_and_reference: 1234,r2
41          builder.append(",");
42          builder.append(sr.getSignificance()); // p值
43          builder.append(",");
44          builder.append(sr.getIntercept()); // 截距
45          builder.append(",");
46          builder.append(sr.getSlope()); // 斜率
47          // 准备归约器来输出
48          // p值、截距和斜率
49          Text reducerValue = new Text(builder.toString());
50          emit(null, reducerValue);
51      }
52  }
53 }
```

# Hadoop实现类

我们的MapReduce/Hadoop实现包括表27-2所示的类。

表27-2：MapReduce/Hadoop实现所需的类

| 类名 | 类描述 |
|---|---|
| LinearRegressionDriver | 驱动器类，定义输入/输出，并注册插件类 |
| LinearRegressionMapper | 定义map()函数 |
| LinearRegressionReducer | 定义reduce()函数 |
| LinearRegressionAnalyzer | 定义如何从HDFS读取输出数据 |
| LinearRegressionClient | 提交作业的客户类 |

# 使用R线性模型的MapReduce解决方案

使用R线性模型的MapReduce解决方案包括两个MapReduce作业。第一个作业聚集所需的数据，并生成第二个MapReduce作业中读取/使用的输入文件。第二个MapReduce作业读取这个输入，然后应用R的线性模型函数lm()（详细信息参见*http://bit.ly/r_linear_models*）。R的线性模型比Apache的SimpleRegression类更为准确和全面。之所以要有第二个MapReduce作业，这是为了避免为每一个gene_id分别调用lm()函数（需要说明，如果为每一个gene_id分别调用R的lm()函数，将不能很好地伸缩，不过对多个输入调用R确实是可伸缩的）。要尽可能多地收集文本文件中的ID，然后调用lm()函数。

介绍这个两阶段MapReduce解决方案之前，下面先介绍如何使用R的lm()函数：

```
# R
R version 2.15.1 (2012-06-22) -- "Roasted Marshmallows"
> x <- c(-1.0, 1.1, 2.2, 3.3, 4.4, 5.5, -1.5)
> y <- c(-1.1, 1.88, 2.88, 3.44, 4.44, 5.99, -1.8)
> fit = lm(x ~ y)
```

```
> intercept = fit$coef[1]
> intercept
(Intercept)
-0.07142071
> slope = fit$coef[2]
> slope
       y
0.921802
> pValue = anova(fit)$'Pr(>F)'[1]
> pValue
[1] 1.279226e-05
> rsquare = summary(fit)$r.squared
> rsquare
[1] 0.9830423
> plot(x, y, xlim=c(min(x)-2, max(x)+2), ylim=c(min(y)-5, max(y)+5))
> abline(fit, lwd=1)
```

利用R的plot()和abline()函数作图，可以生成线性模型，如图27-2所示。

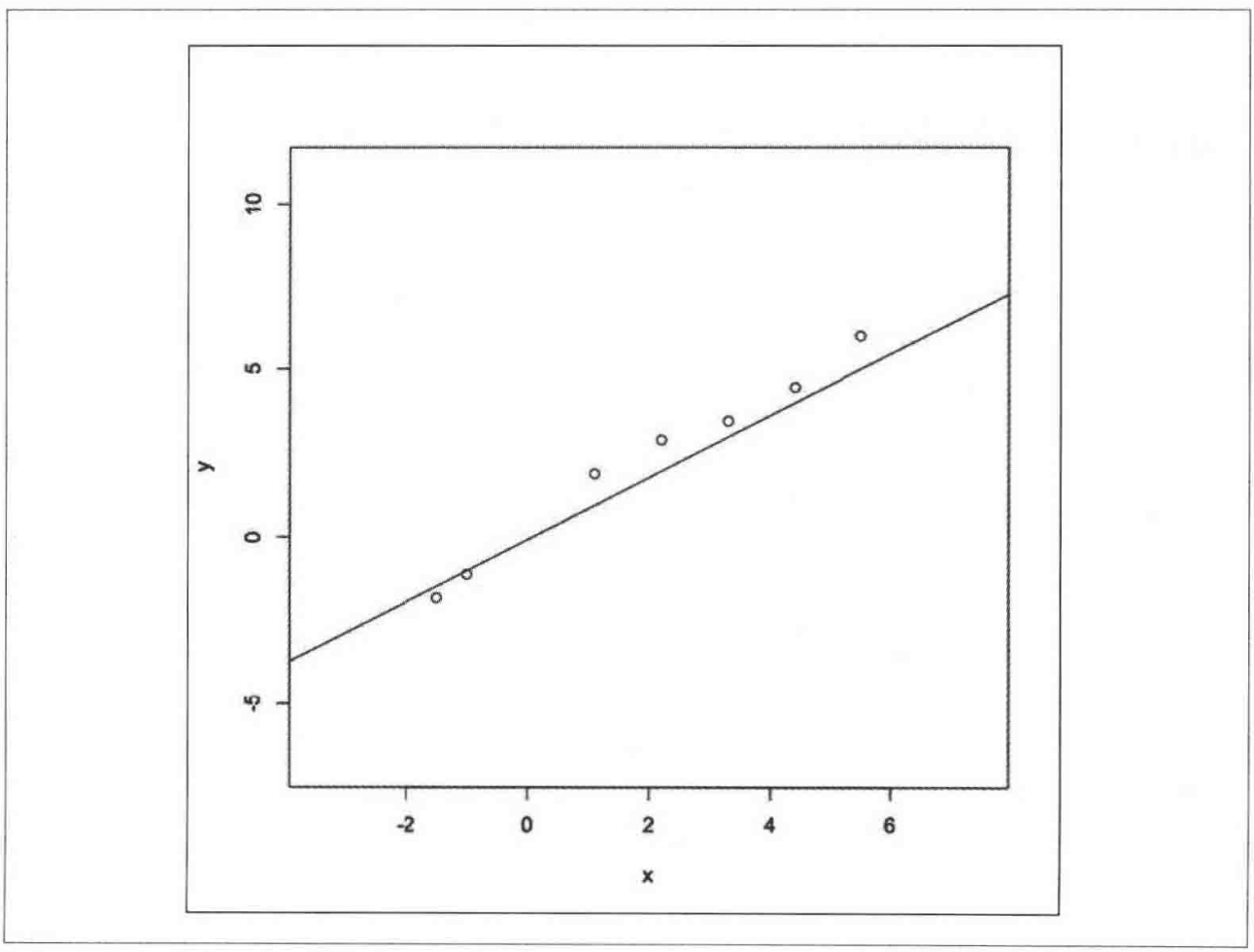



图27-2：线性回归模型

## 阶段1

阶段1中map()函数的主要任务是找出变量$x$，将它传递到一个归约器。这个归约器会

接收gene_id和reference作为键，相关的值是gene_value（线性回归中的变量$x$）。这个例子中，变量$y$是生存时间；生存时间将从驱动器类传递到MapReduce/Hadoop的Configuration对象，由阶段1归约器函数获取这个对象（用来生成适当的输入数据，由阶段2的map()函数处理）。

所以，我们要通过MapReduce/Hadoop的Configuration对象传递bioset_id和survival_time，由阶段1的归约器函数获取这些变量。令bioset_id为$\{B_1, B_2, ..., B_n\}$，survival_time为$\{T_1, T_2, ..., T_n\}$（注意，bioset_id和survival_time数目必须相同）。

每个归约器将接收和处理一个键-值对。键是gene_id和reference，值是一个gene_value列表。Reduce()函数从MapReduce/Hadoop的Configuration对象得到$y$值（在Hadoop中，这会由setup()方法完成）。一旦有了$x$和$y$变量，把它们写至一个文本文件，这个文本文件将保存在HDFS中，由阶段2的map()函数处理。

例如，归约器将生成以下文本文件：

```
GENE-ID-1; V11, V12, ..., V1n; T11, T12,..., T1n
GENE-ID-2; V21, V22, ..., V2n; T21, T22,..., T2n
...
GENE-ID-m; Vm1, Vm2, ..., Vmn; Tm1, Tm2,..., Tmn
```

注意gene_value表示$x$，还要传递bioset_id和生存时间（由y表示）。我们要用Hadoop的Configuration对象传递bioset_id，因为bioset_id[i]与survival_time[i]对应，要确保使用适当的$x$和$y$完成线性回归。

假设阶段1有200个映射器和30个归约器。根据这个配置，阶段1的MapReduce作业将创建30个输出文件（在HDFS中）：

```
/<hdfs-output-directory>/part-00000
/<hdfs-output-directory>/part-00001
...
/<hdfs-output-directory>/part-00029
```

阶段2中map()函数的输入就是这些HDFS文件（由阶段1作业的归约器生成），如示例27-4所示。

示例27-4：线性回归阶段1: map()函数

```
/**
 * @param key由Hadoop生成（在这里忽略）
 * @param value格式: <gene_id_and_ref><;><bioset_id><,><gene_value>
 */
map(key, value) {
   String line = value.toString().trim();
   if ((line == null) || (line.length() == 0)) {
      return;
   }
```

```
10
11    String[] tokens = StringUtils.split(line, ";");
12    String gene_id_and_ref = tokens[0];
13    String bioset_id_and_value = tokens[1];
14    if (tokens.length == 2) {
15       // 为归约器准备(key, value)
16       emit(gene_id_and_ref, bioset_id_and_value);
17    }
18 }
```

Reduce()函数的定义如示例27-5所示。

示例27-5：线性回归阶段1: reduce()函数

```
 1 public class LinearModelReducer extends Reducer<Text, Text, NullWritable, Text> {
 2       // 实例变量
 3       private Configuration conf = null;
 4       private String type = null;
 5       // biosetID表示为String：注意bioset的顺序非常重要
 6       public List<String> biosets = null;
 7       // 注意：时间项的顺序非常重要
 8       public List<Double> time = null;
 9
10       // 只运行一次
11       public void setup(Context context) {
12         this.conf = context.getConfiguration();
13         this.type = conf.get("type");
14         // 从Hadoop的Configuration对象得到参数
15         String biosetsAsString = conf.get("biosets");
16         this.biosets =
17            DataStructuresUtil.splitOnToListOfString(biosetsAsString, ",");
18         String timeAsCommaSeparatedString = conf.get("time");
19         this.time = DataStructuresUtil.toListOfDouble(timeAsCommaSeparatedString);
20       }
21
22       // key = geneid_and_ref
23       // values = { biosetID,value }
24       public void reduce(Text key, Iterable<Text> values, Context context) {
25             //
26             // 将为每个键生成以下结果：
27             // (( key=<geneID><r{1,2,3,4}> [键示例：1234r2] ))
28             // <geneID><;><V1, V2, ..., Vn><;><T1,T2, ..., Tn>
29             // 其中V是基因值，T是时间值
30             //
31             // T1对应V1
32             // T2对应V2
33             // ...
34             // Tn对应Vn
35             //
36             int numberOfValues = 0;
37             Iterator<Text> iter = values.iterator();
38             // 生成geneID; V1, V2, ...
39             StringBuilder valuebuilder = new StringBuilder();
40             // 生成T1, T2, ...
41             StringBuilder timebuilder = new StringBuilder();
```

```
42
43              valuebuilder.append(key.toString());
44              valuebuilder.append(";");
45
46              while (iter.hasNext()) {
47                  Text pairAsText = iter.next();
48                  if (pairAsText == null) {
49                      continue;
50                  }
51                  String pairAsString = pairAsText.toString();
52                  String[] tokens = StringUtils.split(pairAsString, ",");
53                  // biosetID = tokens[0]
54                  // value = tokens[1]
55                  if (tokens.length != 2) {
56                     // 完成，所有值都必须提供给
57                     // 线性模型，不需要向Hadoop写任何结果
58                     continue;
59                  }
60
61                  int index = biosets.indexOf(tokens[0]);
62                  if (index == -1) {
63                      // biosetID未找到
64                      continue;
65                  }
66
67                  // 在index找到biosetID
68                  // biosets.get(index) = biosetID;
69                  double valueAsDouble = Double.parseDouble(tokens[1]);
70
71                  // 更新值
72                  valuebuilder.append(valueAsDouble);
73                  valuebuilder.append(",");
74
75                  // 更新时间
76                  timebuilder.append(time.get(index));
77                  timebuilder.append(",");
78
79                  numberOfValues++;
80              }
81
82              if (numberOfValues > 2) {
83              // 准备归约器提供输出
84              String geneAndValues = valuebuilder.toString();
85              // 去除最后一个","
86              String geneAndValuesFinal =
87                 geneAndValues.substring(0, geneAndValues.length()-1);
88              String timesAsString = timebuilder.toString();
89              // 去除最后一个","
90              String timeFinal =
91                 timesAsString.substring(0, timesAsString.length()-1);
92              // reducerValue = <geneID><;><V1, V2, ..., Vn><;><T1,T2, ..., Tn>
93              Text reducerValue = new Text(geneAndValuesFinal + ";" + timeFinal);
94              context.write(null, reducerValue);
95              }
96      }
```

```
97 }
```

## 阶段2

阶段2有一个映射器，不过不需要归约器。map()函数将读取文本文件（（阶段1中reduce()函数的输出），然后调用R的lm()函数。

map()函数的输入是阶段1归约器生成的以下HDFS文件（如果阶段1有30个归约器，那么阶段2的映射器就有30个输入文件）：

```
/<hdfs-output-directory>/part-00000
/<hdfs-output-directory>/part-00001
...
/<hdfs-output-directory>/part-00029
```

阶段2的map()函数将把各个文件传递到一个shell脚本，它会调用R的lm()函数。如果在文本文件中打包大量输入，R处理就会非常高效（这样一来，可以只启动一个R进程，对从输入文件读取的每个记录调用一次lm()）。示例27-6中给出了这个R脚本*linear_model.template.r*（我们使用FreeMarker作为模板引擎来生成具体的Linux shell脚本），可以看到这里如何调用lm()函数。

示例27-6：MapReduce解决方案阶段2的R脚本

```
#!/usr/local/bin/Rscript
# 模板名: linear_model.template.r
# input_file是本地文件系统中的part-nnnnn文件
input_file = "${input_file}"
cat("input_file=", input_file, "\n")
output_file = "${output_file}"
cat("output_file=", output_file, "\n")

#
# 文件中的各个记录将有以下格式
# <geneID><;><V1, V2, ..., Vn><;><T1,T2, ..., Tn>
#
# > a = "g1;1,2,3; 4,5,6"
# > items = unlist(strsplit(a, ";"))
# > items
# [1] "g1" "1,2,3" " 4,5,6"
# > geneID = items[[1]]
# > geneID
# [1] "g1"
# > value = as.double(unlist(strsplit(items[[2]], ",")))
# >value
# [1] 1 2 3
# > time = as.double(unlist(strsplit(items[[3]], ",")))
# >time
# [1] 4 5 6
#
linear_model_fun <- function(line) {
   items = unlist(strsplit(line, ";"))
```

```
    geneID = items[[1]]
    value = as.double(unlist(strsplit(items[[2]], ",")))
    time = as.double(unlist(strsplit(items[[3]], ",")))
    fit = lm(value ~ time)
    intercept = fit$coef[1]
    slope = fit$coef[2]
    pValue = anova(fit)$'Pr(>F)'[1]
    rsquare = summary(fit)$r.squared
    cat(geneID, intercept, slope, pValue, rsquare, "\n",
        file=output_file, append=TRUE)
}

# 主驱动器
conn <- file(input_file, open="r")
while(length(line <- readLines(conn, 1)) > 0) {
    try.output <- try( linear_model_fun(line) )
}
close.connection(conn)
```

# 使用类的Hadoop实现

使用R线性模型的Hadoop实现包括表27-3所示的类（注意，阶段2没有归约器）。

**表27-3：使用R的Hadoop实现所需的类**

| 类名 | 类描述 |
|---|---|
| LinearRegressionClientPhase2 | 提交作业的客户类 |
| LinearRegressionDriverPhase2 | 驱动器类，定义输入/输出并注册插件类 |
| LinearRegressionMapperPhase2 | 定义map()函数 |
| LinearRegressionAnalyzerPhase2 | 定义如何从HDFS读取输出数据 |

这一章为线性回归问题提供了多个MapReduce解决方案，这些方案可以随数据量的增长很好地伸缩。第28章将在MapReduce上下文讨论幺半群（monoid），并介绍如何使用幺半群优化MapReduce编程。

# 第28章

# MapReduce和幺半群

## 概述

这一章以Jimmy Lin的一篇文章“Monoidify! Monoids as a Design Principle for Efficient MapReduce Algorithms”[15]为基础。在这篇文章中，Lin清晰地介绍了幺半群作为高效MapReduce算法的一个设计原则。不过，什么是幺半群？哪些性质可以定义幺半群？另外幺半群对MapReduc范式有什么帮助？Lin指出如果一个MapReduce操作不是幺半群，则很难高效地使用组合器。

另外，David Saile[13]指出：

> 幺半群（monoid）是带有一个可结合的二元运算和一个单位元的代数结构。例如，带有单位元0和加法的自然数[注1]$N$就构成一个幺半群。在传统MapReduce中，映射器不是约束的，不过要求归约器是一个可结合运算（的迭代应用）。最近的研究指出，在已知的MapReduce应用中，归约实际上具有幺半群的性质。也就是说，事实上，归约就是以一个幺元u迭代地应用一个关联运算“•”。在单词计数例子中，归约会迭代地完成加法“+”（“0”作为幺元）。如果除了结合性，还需要有可交换性，并行执行调度会更为灵活。

关于基于幺半群的常见MapReduce计算，[5]给出了一个详细的分析。接下来，我们会简要地回顾MapReduce的组合器和抽象代数的幺半群，看看它们相互之间有什么关系。将不具有结合性的MapReduce运算转换为可结合的运算相当重要，因为这样一来，我们

注1：自然数（natural numbers）是指正整数集合{1, 2, 3, ...}或非负整数集合{0, 1, 2, 3, ...}。

就能充分利用组合器（而且将组合器作为幺半群使用时，可以提高MapReduce作业的性能）。

在MapReduce框架中，组合器（作为一个可选的插件组件）是一个“本地归约”过程，只处理一个服务器生成的数据。成功地使用组合器可以减少给定服务器上映射器生成的中间数据量（正因如此，我们称之为一个本地归约器）。组合器可以作为一个MapReduce优化来减少映射器和归约器之间的网络流量（减少瞬时数据的大小）。一般地，组合器的接口与归约器相同。组合器必须有以下特性：

- 组合器接收一个给定服务器上映射器实例发出的所有数据作为输入（这称为一个本地聚合）。
- 组合器的输出发送给归约器；有些程序员称之为本地服务器归约。
- 组合器必须没有任何副作用；组合器可以运行无限次。
- 组合器必须有相同的输入和输出键类型（参见示例28-1）。
- 组合器必须有相同的输入和输出值类型（参见示例28-1）。
- 映射阶段后组合器在内存中运行。

示例28-1给出了组合器模板的定义。

示例28-1：组合器模板

```
1 public class MyCombiner {
2    ...
3    public void combine(KeyType key, Iterable<ValueType> values) {
4       ...
5       KeyType key2 = ...;
6       ValueType value2 = ...;
7       ...
8       emit(key2, value2);
9       ...
10   }
11 }
```

这个模板指示组合器生成的键-值对必须与归约器（作为输入）接收的键-值对匹配。例如，如果映射器输出（$T_1$, $T_2$）对（键为$T_1$类型，值为$T_2$类型），那么组合器也必须发出（$T_1$, $T_2$）对。

Lin[15]总结指出：“设计高效的MapReduce算法的一个原则可以描述如下：由映射器发出的中间值创建一个幺半群。一旦将对象转换为幺半群（monoidify），组合器的使用和映射器中的组合技术就变得很简单。”

MapReduce/Hadoop实现并没有`combine()`函数，在Hadoop中我们只是使用`reduce()`来实现组合器，不过可以使用插件`Job.setCombinerClass()`定义一个组合器类。

Haskell编程语言对幺半群提供了直接支持（*http://bit.ly/monoid_structure*）。在Haskell（*http://bit.ly/haskell_monoids*）中，“幺半群是一种类型，满足以下规则：该类型的两个元素可以组合为相同类型的另一个元素。”

# 幺半群的定义

幺半群是一个三元组（S,f,e），其中S是一个集合（也就是幺半群的底层集合），f: S×S→S是一个映射（即幺半群的二元运算），e∈S是幺半群的单位元（幺元）。带有二元运算?的幺半群满足以下3个性质（需要说明，f(a, b) = a•b）：

- 闭包：对于S中的所有a和b，运算a •b的结果也在S中。
- 结合律: 对于S中的所有a, b,和c，满足以下等式：

  (a • b) •c = a • (b•c)
- 单位元：S中存在一个元素e，使得对于S中的所有元素a，满足以下两个等式：

  e • a = a

  a • e = a

用数学符号表示，可以写为：

- 闭包：∀a, b∈S : a•b∈S
- 结合律: ∀a, b, c∈S : (a•b)• c = a•(b•c)
- 单位元: ∃e∈S : ∀a∈S : e • a = a • e = a

幺半群可能还有其他性质。例如，幺半群运算可能（但不一定）有以下性质：

- 幂等性: ∀a∈S : a • a = a
- 交换律: ∀a, b,∈S : a • b = b • a

# 如何构成幺半群

要构成一个幺半群，首先需要一个类型S，它定义了一个值集合，如整数集：{0, -1, +1, -2, +2, ...}。第二个部分是一个二元函数：

•: S × S → S

然后需要确保对于任意的两个值x∈S和y∈S，会得到一个结果对象，即x和y的组合：

x • y : S

例如，如果S是一个整数集，则二元运算•可以是加法（+）、乘法（×）或除法（÷）。最后，作为第3个也是最重要的一个部分，我们要求•遵循一组规则。如果确实满足这些规则，那么S以及•就称为一个幺半群。我们称（S, •, e）是一个幺半群，其中 e ∈ S是单位元（如对于加法，单位元为0，对于乘法，单位元为1）。

另外，注意一个实数集上的二元除法运算（÷）并不是一个幺半群：

$((12 \div 4) \div 2) \neq (12 \div (4 \div 2))$

$((12 \div 4) \div 2) = (3 \div 2) = 1.5$

$(12 \div (4 \div 2)) = (12 \div 2) = 6.0$

正如“Monoids for Programmers”（*http://bit.ly/monoids_4_prgrmmrs*）所指出的：

> 幺半群描述了这样一个概念：基于一个空事物（幺元）的概念，将任意多个事物组合为一个事件....自然数之上的加法就是这样一个例子。加法函数+允许我们把任意多个自然数组合为一个自然数，即总和。空的总和为0。另一个例子是字符串连接。利用连接运算，我们可以把任意多个字符串组合为一个字符串。空连接就是...空串。

# 幺半群和非幺半群示例

下面的小节将给出一些例子来帮助你理解幺半群的概念。

## 整数集的最大值

集合`S = {0, 1, 2, ...}`上的MAX（求最大值）运算定义了一个满足交换律的幺半群，其单位元为0：

```
MAX(a, MAX(b, c)) = MAX(MAX(a, b), c)

MAX(a, 0) = MAX(0, a) = a

MAX(a, b) ∈ S
```

## 整数集的减法

整数集上的减法运算（−）不能构成一个幺半群；这个运算不满足结合律：

$(1-2)-3 \neq 1-(2-3)$

$(1-2)-3 = -4$

$1-(2-3) = 2$

## 整数集的加法

整数集上的加法运算（+）定义了一个幺半群；这个运算满足交换律和结合律，单位元为0：

(1 + 2) + 3 = 6

1 + (2 + 3) = 6

n + 0 = n

0 + n = n

可以把这个幺半群形式化描述如下（其中e(+)定义了一个单位元）：

S = {0, −1, +1, −2, +2, −3, +3, ...}

e(+) = 0, 单位元为0

m(a, b) = m(b, a) = a + b

## 整数集的乘法

自然数N = {0, 1, 2, 3, ...}上的乘法构成了一个可交换的幺半群（单位元为1）。

## 整数集的均值

另一方面，自然数N = {0, 1, 2, 3, ...}上的均值（平均数）函数不能构成幺半群。下面的例子可以说明一个值集的任意子集均值的均值并不等于这个值集的均值：

MEAN( 1, 2, 3, 4, 5) ≠ MEAN(MEAN(1, 2, 3) ,MEAN (4, 5))

$$\mathrm{MEAN}(1,2,3,4,5)=\frac{(1+2+3+4+5)}{5}=\frac{15}{5}=3$$

$$\mathrm{MEAN}\left(\mathrm{MEAN}(1,2,3),\mathrm{MEAN}\left(4,5\right)\right)=\mathrm{MEAN}(2,4,5)=\frac{(2+4.5)}{2}=3.25$$

## 不满足交换律的示例

作为不满足交换律的例子，可以考虑任意二进制串的集合：其中的元素是0和1的有限有序序列。二元运算是连接（例如，concat(1011, 001001) = 1011001001）。单位元是空串。因此，二进制串的连接是一个幺半群。

## 整数集的中值

自然数上的中值函数不能构成一个幺半群：

$$\text{EMDIAN}(1,2,3,4,5,6,7,8,9) \neq \text{MEDIAN}(\text{MEDIAN}(1,2,3),\text{MEDIAN}(5,6,7,8,9)))$$

$$\text{EMDIAN}(1,2,3,4,5,6,7,8,9) = \frac{(5+6)}{2} = 5.5$$

$$\text{EMDIAN}(\text{MEDIAN}(1,2,3),\text{MEDIAN}(5,6,7,8,9)) = \text{MEDIAN}(2,7) = \frac{(2+7)}{2} = 4.5$$

## 列表的连接

基于空列表（表示为[]）的列表连接(+)是一个幺半群。对于任意的列表，可以写为：

```
L +[] = L
[] + L = L
```

而且连接运算是可结合的。使用+将[]与任意列表组合可以得到相同的列表。例如，[]+[1,2,3] = [1,2,3]和[1,2,3] + [] = [1,2,3]。

可以使用+来连接任意的两个列表。例如，[1,2,3] + [7,8] = [1,2,3,7,8]。

## 整数的并集/交集

整数集合的并集和交集运算构成了一个幺半群。

## 函数示例

这个例子选自Mike Stay的Monoids博客（*http://bit.ly/stay_monoids*）。对于集合T到自身（集合T）的函数，其组合（composition）构成了一个幺半群：

S = { 形如a : T → T 的所有函数}

e(•) = 单位函数

f(a, b) = b ○ a，其中 ○ 是函数的组合：

(b ○ a)(x) = b(a(x))

例如，给定集合T = {0, 1}，则有：

- S包含4个可能的（T到自身的）函数：$S = \{k_0, 1_T, NOT, k_1\}$
  - 常量函数将所有元素映射到0：

    $k_0: T \to T$

$k_0(t) = 0$

— 单位函数：

$1_T: T \to T$
$1_T(t) = t$

— 反转输入的函数：

$NOT: T \to T$
$NOT(t) = 1 - t$

— 将所有元素映射到1的常量函数：

$k_1: T \to T$
$k_1(t) = 1$

- $e(\bullet) = 1_T$ (这是单位函数)
- $f(a, b) = b \bigcirc a$

需要说明，单位函数是组合的单位元（幺元）：

$$(a \bigcirc 1_T)(t) = a(1_T(t)) = a(t)$$

$$(1_T \bigcirc a)(t) = 1_T(a(t)) = a(t)$$

这里给读者留一个练习：请证明这个函数组合满足结合律。

## 矩阵示例

这个矩阵示例选自John Perry[22]。令$N = \{1, 2, 3, ...\}$，另外令$m, n \in N$。那么，$m \times n$整数矩阵集合（写为$Z^{m \times n}$）的加法满足幺半群所要求的性质：

- 闭包可以由定义保证。
- 结合律由其元素的结合律保证。
- 加法的单位元是0，即0矩阵。

# MapReduce示例：非幺半群

在这一节中，我们将给出Jimmy Lin[15]提供的例子。假设有大量的键-值对，其中键是字符串，值是整数，我们的目标是按键找出所有值的平均值。在SQL中，可以如下完成（假设mytable包含key和value列）：

1. 选择所有数据：

```
SELECT key, value FROM mytable;
```

| key | value |
|---|---|
| key1 | 10 |
| key1 | 20 |
| key1 | 30 |
| key2 | 40 |
| key2 | 60 |
| key3 | 20 |
| key3 | 30 |

2. 选择所有数据并按键分组：

```
SELECT key, AVG(value) FROM mytable GROUP BY key;
```

| key | value |
|---|---|
| key1 | 20 |
| key2 | 50 |
| key3 | 25 |

示例28-2和示例28-3提供了这个MapReduce算法的第一个版本，这里映射器没有为均值/平均数函数生成幺半群输出。

示例28-2：映射器：不生成幺半群

```
/**
 * @param key是一个字符串对象
 * @param value是与键关联的一个long
 */
map(String key, Long value) {
    emit(key, value);
}
```

示例28-3：归约器：不接收幺半群

```
/**
 * @param key是一个字符串对象
 * @param value是一个long列表: [i1, i2, ...]
 */
reduce(String key, List<Long> list) {
   Long sum = 0;
   Integer count = 0;
   for (Long i : list) {
      sum = sum + i;
      count++;
   }
   double average = sum/count;
   emit(key, average);
}
```

从这个MapReduce算法的第一个版本可以观察到两点：

- 这个算法效率不高，因为MapReduce框架的shuffle()和sort()函数必须完成大量工作。
- 不能使用归约器作为组合器，因为我们知道一个值集的任意子集均值的均值不等于这个值集的均值（不是一个幺半群）。

需要说明，使用组合器可以让MapReduce算法更高效，这样能减少网络流量（要确保组合器提供了足够的聚合），并减少MapReduce框架的shuffle()和sort()函数的工作。现在的问题是：如何让我们的归约器作为一个组合器？答案就是让映射器的输出成为一个幺半群，也就是说，我们要改变映射器的输出。一旦映射器输出构成一个幺半群，组合器和归约器就能正确而高效地工作。

# MapReduce示例：幺半群

在示例28-4中，我们修改了映射器，它会生成键-值对，其中键是字符串，值是一个（sum, count）对，这是一个幺半群。

示例28-4：映射器：生成幺半群

```
1 /**
2  * @param key是一个字符串对象
3  * @param value是与键关联的一个（long : sum, int: count）对
4  */
5 map(String key, Long value) {
6    emit(key, Pair(value, 1));
7 }
```

可以看到，键与之前是一样的，不过值是一个（sum, count）对。现在映射器的输出是一个幺半群，单位元是（0, 0）。元素的求和运算可以如下完成：

$$(a, b) \oplus (c, d) = (a + c, b + d)$$

现在可以正确地计算均值函数，因为映射器会输出幺半群：

$$\text{MEAN}\ (1, 2, 3, 4, 5) = \text{MEAN}\ (\text{MEAN}\ (1, 2, 3)\ ,\ \text{MEAN}\ (4, 5)\ )$$

$$\text{MEAN}\ (1, 2, 3, 4, 5) = \frac{(1+2+3+4+5)}{5} = \frac{15}{5} = 3$$

$$\text{MEAN}\ (\text{MEAN}\ (1, 2, 3)\ ,\ \text{MEAN}\ (4, 5)\ ) =$$

$$\text{MEAN}\ (\text{MEAN}\ (6, 3)\ ,\ \text{MEAN}\ (9, 2)\ ) = \text{MEAN}\ (15, 5) = 3$$

示例28-5和28-6给出了修改后的算法，其中映射器的输出是一个幺半群。

示例28-5：新组合器：接收幺半群值

```
/**
 * @param key是一个字符串对象
 * @param value是一个列表 = [(V1, c1), (V2, c2), ...]
 */
combine(String key, List<Pair<Long, Integer>> list) {
   Long sum = 0;
   Integer count = 0;
   for (Pair<Long, Integer> pair : list) {
      sum += pair.v;
      count += pair.c
   }
   emit(key, new Pair(sum, count));
}
```

示例28-6：新归约器：接收幺半群值

```
/**
 * @param key 是一个字符串对象
 * @param value是一个列表 = [(V1, c1), (V2, c2), ...]
 */
reduce(String key, List<Pair<Long, Integer>> list) {
   Long sum = 0;
   Integer count = 0;
   for (Pair<Long, Integer> pair : list) {
      sum += pair.v;
      count += pair.c
   }
   Pair<Long, Integer> partialPair = new Pair<Long, Integer>(sum, count);
   emit(key, partialPair);
}
```

## Hadoop实现类

使用幺半群的MapReduce解决方案包括表28-1所示的Java类。在这个解决方案中，使用PairOfLongInt[注2]类作为一个Hadoop WritableComparable，表示一个Long和Int对。

**表28-1：使用幺半群的MapReduce/Hadoop解决方案中用到的Java类**

| 类名 | 类描述 |
|---|---|
| MeanDriver | 提交Hadoop作业的驱动器程序 |
| MeanMonoidizedMapper | 定义map() |
| MeanMonoidizedCombiner | 定义combiner() |
| MeanMonoidizedReducer | 定义reduce() |

注2：这是Jimmy Lin开发的Cloud9工具包中的一个类，Cloud9工具包是一个帮助处理大数据的Hadoop工具集。

表28-1：使用幺半群的MapReduce/Hadoop解决方案中用到的Java类（续）

| 类名 | 类描述 |
| --- | --- |
| SequenceFileWriterDemo | 创建一个示例SequenceFile |
| HadoopUtil | 定义一些工具函数 |
| edu.umd.cloud9.io.pair.PairOfLongInt WritableComparable | 表示一个Long和Int对 |

# 运行示例

下面的小节会概要介绍使用幺半群的MapReduce/Hadoop运行示例的步骤。

## 创建输入文件（作为SequenceFile）

使用SequenceFileWriterDemo类生成示例输入SequenceFile。与文本文件相比，SequenceFile的优点在于：在map()函数中，不需要解析输入来找出key和value字段。

SequenceFile还可以压缩，可以逐记录地压缩，也可以逐块地压缩。要创建一个SequenceFile，这里使用了SequenceFile.createWriter()方法，它会返回一个SequenceFile.Writer实例。然后使用File.Writer.append(key,value)方法写键-值对。完成这些工作之后，最后要调用close()方法。我们的示例输入如下所示：

```
$ java SequenceFileWriterDemo test.seq
key1 1
key1 2
key1 3
key1 4
key1 5
key2 2
key2 4
key2 6
key2 8
key2 10
key3 3
key3 6
key3 9
key3 12
key3 15
key4 4
key4 8
key4 12
key4 16
key4 20
key5 5
key5 10
key5 15
key5 20
key5 25
```

创建HDFS输入和输出目录

接下来，创建Hadoop分布式文件系统输入和输出目录，如下所示：

```
$ hadoop fs -mkdir /monoid
$ hadoop fs -mkdir /monoid/input
$ hadoop fs -mkdir /monoid/output
```

## 将文件复制到HDFS并验证

这里要把输入文件复制到HDFS，然后验证文件的内容：

```
$ hadoop fs -copyFromLoal test.seq /monoid/input/
$ hadoop fs -text /monoid/input/test.seq
key1 1
key1 2
key1 3
key1 4
key1 5
key2 2
key2 4
key2 6
key2 8
key2 10
key3 3
key3 6
key3 9
key3 12
key3 15
key4 4
key4 8
key4 12
key4 16
key4 20
key5 5
key5 10
key5 15
key5 20
key5 25
```

## 准备脚本

下面给出运行这个MapReduce作业的shell脚本：

```
$ cat run_monoids.sh
#/bin/bash
export JAVA_HOME=/usr/java/jdk7
export HADOOP_HOME=/usr/local/hadoop/hadoop-2.5.0
export PATH=$PATH:$HADOOP_HOME/bin:$JAVA_HOME/bin
export BOOK_HOME=/mp/data-algorithms-book
export JAR=$BOOK_HOME/dist/data_algorithms_book.jar
$HADOOP_HOME/bin/hadoop fs -rmr /monoid/output
driver=org.dataalgorithms.chap28.mapreduce.MeanDriver
$HADOOP_HOME/bin/hadoop jar $JAR $driver /monoid/input /monoid/output
```

### 运行MapReduce作业

最后运行这个MapReduce作业，如下所示：

```
$ ./run_monoids.sh
...
14/10/24 22:19:09 INFO mapreduce.Job: Running job: job_1412870576870_0036
14/10/24 22:19:18 INFO mapreduce.Job: map 0% reduce 0%
14/10/24 22:19:34 INFO mapreduce.Job: map 100% reduce 0%
...
14/10/24 22:19:54 INFO mapreduce.Job: map 100% reduce 83%
14/10/24 22:19:55 INFO mapreduce.Job: map 100% reduce 100%
        ...
Map-Reduce Framework
Map input records=25
Map output records=25
Combine input records=25
Combine output records=5
Reduce input groups=5
Reduce input records=5
Reduce output records=5
         ...
```

## 查看Hadoop输出

下面给出Hadoop生成的输出：

```
$ hadoop fs -text /monoid/output/part*
key2    6.0
key3    9.0
key4   12.0
key5   15.0
key1    3.0
```

# 使用幺半群的Spark示例

在Spark中，调用reduceByKey()为各个键计算全局总和之前会自动在各个机器上完成本地组合。程序员不需要指定组合器。由于组合是自动的，所以要特别注意reduceByKey()，确保使用组合器不会改变函数的语义（在这个例子中，我们期望得到均值函数）。接下来，要通过提供幺半群结构实现均值函数的一个解决方案，从而保留均值函数的正确语义。实际上，如果你的数据结构有一个幺半群形式，那么可以直接将这个数据结构插入到MapReduce或DAG环境，有效地使用组合器和归约器。

需要说明，均值的均值不是一个幺半群。因此，为了保证一个Long数据类型数值集合上均值函数的语义，必须提供一个幺半群结构，从而能有效而且正确地使用组合器。那么均值函数的幺半群结构是什么呢？一个可能的幺半群是值的（V,C）对，其中V是值，C是计数。例如：

- 不使用幺半群（组合器会生成错误的结果）：

```
mean(1, 2, 3, 4, 5)
   = mean ( mean(1,2), mean(3,4,5) )
   = mean ( 3/2, 12/3 )
   = mean (1.5, 4)
   = 2.75 [WRONG ANSWER]
```

- 使用幺半群（组合器会生成正确的结果）：

```
mean(1, 2, 3, 4, 5)
   = mean ( (1,1), (2,1), (3,1), (4,1), (5,1) )
   = mean ( mean ((1,1), (2,1), (3,1)), mean((4,1), (5,1)) )
   = mean ( (1+2+3, 1+1+1) ), (4+5, 1+1))
   = mean ( (1+2+3+4+5, 1+1+1+1+1) )
  = 15/5
   = 3 [CORRECT ANSWER]
```

如果没有幺半群结构，就不能保证使用组合器时的正确性。如果有幺半群结构，可以写为：

```
mean( (V, C) )
   = V / C

mean( (V1, C1), (V2, C2) )
   = mean ( (V1+V2), (C1+C2) )
   = (V1+V2) / (C1+C2)

mean( (V1, C1), (V2, C2), (V3, C3), (V4, C4) )
 = mean( mean((V1, C1),(V2, C2)), mean((V3, C3), (V4, C4)))
 = mean( ((V1+V2), (C1+C2)), ((V3+V4), (C3+C4)) )
 = (V1+V2+V3+V4) / (C1+C2+C3+C4)
```

可以看到，幺半群结构可以保证组合器的正确使用。这样一来，可以使用组合器优化，同时不会丢失所需的功能（在这个例子中，就是均值函数）。

## 高层步骤

这里通过一个Java类提供整个Spark解决方案。Spark的`reduceByKey()`会默认使用组合器，幺半群结构可以保留值集合上均值函数的语义。下面给出这个Spark解决方案的高层步骤（如示例28-7所示），然后我们会详细分析每一个步骤。

示例28-7：SparkMeanMonoidized的高层步骤

```
1 package org.dataalgorithms.chap28.spark;
2 // 步骤1: 导入所需的类和接口
3 /**
4  * 给定{(K:String, V:Long)}，我们的目标是找出对应一个给定K的所有值的均值。
5  * 这里将采用适当的方式创建结构，使得使用组合器时，
6  * 均值的均值能正确地返回所有值的均值。
7  * 在这个例子中，我们会创建一个幺半群，
8  * 从而能使用组合器而不会丢失均值函数的语义。
```

```
 */
public class SparkMeanMonoidized {
  public static void main(String[] args) throws Exception {
     // 步骤2: 处理输入参数
     // 步骤3: 由输入创建一个RDD
     // 步骤4: 创建一个幺半群
     // 步骤5: 通过保持幺半群归约多次出现的K
     // 步骤6: 通过映射值查找均值
     System.exit(0);
  }
}
```

## 步骤1：导入所需的类和接口

这个步骤如示例28-8所示，将导入所需的类和接口。

示例28-8：步骤1: 导入所需的类和接口

```
// 步骤1: 导入所需的类和接口
import scala.Tuple2;
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.api.java.JavaPairRDD;
import org.apache.spark.api.java.JavaSparkContext;
import org.apache.spark.api.java.function.Function;
import org.apache.spark.api.java.function.Function2;
import org.apache.spark.api.java.function.PairFunction;
```

## 步骤2：处理输入参数

这个步骤如示例28-9所示，读取输入路径来查找一个数值集合的均值。每个记录的格式如下：

```
<key-as-string><TAB><value-as-long>
```

示例28-9：步骤2: 处理输入参数

```
    // 步骤2: 处理输入参数
    if (args.length != 1) {
       System.err.println("Usage: SparkMeanMonoidized <input-path>");
       System.exit(1);
    }
    final String inputPath = args[0];
```

## 步骤3：由输入创建一个RDD

这个步骤如示例28-10所示，通过读取输入路径创建第一个RDD（`JavaRDD<String>`）。这个RDD中的各个元素分别是一个记录，格式如前所示。

示例28-10：步骤3: 由输入创建一个RDD

```
    // 步骤3: 由输入创建一个RDD
    //    输入记录格式:
    //       <string-key><TAB><long-value>
```

```
4     JavaSparkContext ctx = new JavaSparkContext();
5     JavaRDD<String> records = ctx.textFile(inputPath, 1);
6     records.saveAsTextFile("/output/2");
```

下面列出这个RDD以便于调试：

```
$ hadoop fs -cat /output/2/part*
key1 1
key1 2
key1 3
key1 4
key1 5
key2 2
key2 4
key2 6
key2 8
key2 10
key3 3
key3 6
key3 9
key3 12
key3 15
key4 4
key4 8
key4 12
key4 16
key4 20
key5 5
key5 10
key5 15
key5 20
key5 25
```

## 步骤4：创建一个幺半群

这个步骤创建一个幺半群结构，这样一来，就可以有效地使用组合器和归约器，而不会破坏所需功能的语义。在示例28-11中，组合器和归约器的主要功能就是查找数值的均值。

示例28-11：步骤4: 创建一个幺半群

```
1     // 步骤4: 创建一个幺半群
2     // 将input(T)映射到(K,V)对，这是一个幺半群
3     // mean( (V1,C1), (V2,C2) ) => ( (V1+V2), (C1+C2) )
4     JavaPairRDD<String,Tuple2<Long,Integer>> monoid =
5         records.mapToPair(new PairFunction<String,
6                                            String,
7                                            Tuple2<Long,Integer>
8                                           >() {
9         public Tuple2<String,Tuple2<Long,Integer>> call(String s) {
10           String[] tokens = s.split("\t"); // s = <key><TAB><value>
11           String K = tokens[0];
12           Tuple2<Long,Integer> V =
```

```
13              new Tuple2<Long,Integer>(Long.parseLong(tokens[1]), 1);
14          return new Tuple2<String,Tuple2<Long,Integer>>(K, V);
15        }
16    });
```

这里给出RDD以便于调试：

```
$ hadoop fs -cat /output/3/part*
(key1,(1,1))
(key1,(2,1))
(key1,(3,1))
(key1,(4,1))
(key1,(5,1))
(key2,(2,1))
(key2,(4,1))
(key2,(6,1))
(key2,(8,1))
(key2,(10,1))
(key3,(3,1))
(key3,(6,1))
(key3,(9,1))
(key3,(12,1))
(key3,(15,1))
(key4,(4,1))
(key4,(8,1))
(key4,(12,1))
(key4,(16,1))
(key4,(20,1))
(key5,(5,1))
(key5,(10,1))
(key5,(15,1))
(key5,(20,1))
(key5,(25,1))
```

## 步骤5：通过保持幺半群归约多次出现的键

在Spark中，调用reduceByKey()为各个键计算全局总和之前会自动在各个机器上完成本地组合。程序员不需要指定组合器。由于组合是自动的，所以要特别注意reduceByKey()，确保使用组合器不会改变函数的语义（即均值函数）。基于幺半群结构，我们可以使用组合器优化同时提供归约器功能。参见示例28-12。

示例28-12：步骤5: 通过保持幺半群归约多次出现的K

```
1     // 步骤5: 通过保持幺半群归约多次出现的K
2     // 可以使用组合器而不会破坏均值的语义。
3     JavaPairRDD<String, Tuple2<Long,Integer>> reduced = monoid.reduceByKey(
4         new Function2<
5                       Tuple2<Long,Integer>,
6                       Tuple2<Long,Integer>,
7                       Tuple2<Long,Integer>
8                      >() {
9         public Tuple2<Long,Integer> call(Tuple2<Long,Integer> V1,
10                                         Tuple2<Long,Integer> V2) {
```

```
11              return new Tuple2<Long,Integer>(V1._1+ V2._1, V1._2+ V2._2);
12          }
13      });
14      reduced.saveAsTextFile("/output/4");
15      // 现在归约的RDD包含所需的值来提供最终输出
```

这里给出RDD以便于调试（注意，输出有一个幺半群结构）：

```
$ hadoop fs -cat /output/4/part*
(key4,(60,5))
(key5,(75,5))
(key2,(30,5))
(key3,(45,5))
(key1,(15,5))
```

### 步骤6：通过映射值查找均值

查找均值被延至最后一步。示例28-13将一个幺半群结构转换为具体的均值。

示例28-13：步骤6: 通过映射值查找均值

```
1     // 步骤6: 通过映射值查找均值
2     // mapValues[U](f: (V) => U): JavaPairRDD[K, U]
3     // 通过一个映射函数传递键-值对RDD中的各个值
4     // 而不改变键；
5     // 这还会保留原来的RDD分区。
6     JavaPairRDD<String,Double> mean = reduced.mapValues(
7         new Function<
8                      Tuple2<Long,Integer>,   // 输入
9                      Double                  // 输出
10                    >() {
11        public Double call(Tuple2<Long, Integer> s) {
12            return ( (double) s._1 / (double) s._2 );
13        }
14    });
15    mean.saveAsTextFile("/output/5");
```

下面给出最终的输出：

```
$ hadoop fs -cat /output/5/part*
(key4,12.0)
(key5,15.0)
(key2,6.0)
(key3,9.0)
(key1,3.0)
```

## 运行示例

### 运行SparkMeanMonoidized的脚本

下面是运行这个Spark解决方案的脚本：

```
$ cat run_spark_monoidized.sh
#!/bin/bash
export JAVA_HOME=/usr/java/jdk7
export SPARK_HOME=/usr/local/spark-1.1.0
export SPARK_MASTER=spark://myserver100:7077
export BOOK_HOME=/mp/data-algorithms-book
export APP_JAR=$BOOK_HOME/dist/data_algorithms_book.jar
INPUT=/home/hadoop/testspark/kv.txt
# 在Spark独立集群上运行
$SPARK_HOME/bin/spark-submit \
--class org.dataalgorithms.chap28.spark.SparkMeanMonoidized \
--master $SPARK_MASTER \
--executor-memory 2G \
--total-executor-cores 20 \
$APP_JAR \
$INPUT
```

### 示例运行日志

下面给出在一个非常小的集群上的示例运行日志输出，这个集群包括3个节点（myserver100、myserver200和myserver300）。由于篇幅所限，这里对输出做了编辑：

```
$ ./run_spark_monoidized.sh
...
INFO : Added broadcast_5_piece0 in memory on myserver100:60093
 (size: 21.2 KB, free: 265.4 MB)
INFO : Updated info of block broadcast_5_piece0
INFO : Submitting 1 missing tasks from Stage 4 (MappedRDD[8]
  at saveAsTextFile
  at SparkMeanMonoidized.java:76)
INFO : Adding task set 4.0 with 1 tasks
INFO : Starting task 0.0 in stage 4.0 (TID 4, myserver200,
  PROCESS_LOCAL, 996 bytes)
INFO : Added broadcast_5_piece0 in memory on myserver300:51784
  (size: 21.2 KB, free: 1060.2 MB)
INFO : Finished task 0.0 in stage 4.0 (TID 4) in 224 ms on myserver200 (1/1)
INFO : Stage 4 (saveAsTextFile at SparkMeanMonoidized.java:76)
  finished in 0.226 s
INFO : Removed TaskSet 4.0, whose tasks have all completed, from pool
INFO : Job finished: saveAsTextFile at SparkMeanMonoidized.java:76,
  took 0.322557953 s
```

# 使用幺半群的结论

我们观察到，在MapReduce中，如果映射器能生成幺半群，就可以利用组合器完成优化并高效地进行处理（也就是说，使用组合器可以减少网络流量，而且由于处理的数据更少，所以MapReduce的sort()和shuffle()函数更为高效）。另外。你还了解了如何在MapReduce算法中使用幺半群。这很有难度。一般来讲，如果想要应用的函数是可交换而且可结合的（幺半群的性质），就可以使用组合器。例如，传统的单词计数即整数集

上的+运算就是一个幺半群（这里可以使用组合器）。不过，整数集上的均值函数不构成幺半群（不满足结合律，如前面的计数器例子所示）。因此，如果适当地使用组合器，可以显著减少从映射器转换到归约器的数据量。

幺半群在函数式编程中也有大量应用和使用。作为计算机科学家、软件工程师和博主，MarkCC（*http://bit.ly/monoids_markcc*）指出：

> 我们为什么要考虑数据结构是不是幺半群呢？因为这样可以利用代数构造编写非常通用的代码，然后用于各种不同的运算。幺半群提供了构建折叠操作所需的工具。可以用幺半群来定义各种类型的折叠（这种操作会把一系列其他操作折叠为一个值）。所以，可以编写适用于列表、字符串、数值、可选值、映射和几乎所有其他结构的折叠操作。只要是一个幺半群，任何数据结构都有一个有意义的折叠操作：幺半群封装了可折叠性需求。

# 函子和幺半群

既然已经知道什么是幺半群，而且了解了它们在MapReduce框架中的使用，下面可以对幺半群应用更高阶的函数（如函子）。函子（functor）是作为函数的一个对象（它既是一个函数，同时也是一个对象）。Java[注3]编程语言（JDK6和JDK7）中没有直接的函子概念，因为在Java中，函数不是首类对象；这说明不能把一个函数名作为参数传递到另一个函数。不过，Java中可以通过定义一个接口和一个方法来模拟函子（这是一个非常简化的模拟）：

```
public interface FunctorSimulation<T1, T2> {
   T2 apply(T1 input);
}
```

关于Java中函子的实现/模拟的详细内容，参见Guava的Function接口（*http://bit.ly/guava-libraries*）和Apache Commons Functor接口（*http://bit.ly/commons_functor*）。Haskell编程语言（http://bit.ly/functors_haskell）未提供对幺半群和函子的直接支持。Bruno P.Kinoshita在他的网站上提供了一个很好的例子（*http://bit.ly/functor_w_java8*），介绍了如何结合Java 8 lambda使用Apache Commons Functor函数式接口。

首先，通过一个简单的例子来提供幺半群上的一个函子。令MONOID = (t,e, f)是一个幺半群，其中t是一个类型（值集合），e是单位元，f是+二元加法函数：

```
MONOID = {
   type t
   val e : t
```

注3：JDK8通过Project Lambda提供了对函子的直接支持（有关的详细信息，参见*http://openjdk.java.net/projects/lambda/*）。

```
    val plus : t x t -> t
}
```

然后定义一个函子Prod，如下所示：

```
functor Prod (M : MONOID) (N : MONOID) = {
    type t = M.t * N.t
    val e = (M.e, N.e)
    fun plus((x1,y1), (x2,y2)) = (M.plus(x1,x2), N.plus(y1,y2))
}
```

然后可以定义其他函子，如Square，如下所示：

```
functor Square (M : MONOID) : MONOID = Prod M M
```

接下来，定义两个幺半群之间的一个函子。令$(M_1, f_1, e_1)$和$(M_2, f_2, e_2)$为幺半群。函子由一个对象映射（幺半群分类为单个对象）和一个箭头映射F: M1 → M2指定：

$$F: (M_1, f_1, e_1) \rightarrow (M_2, f_2, e_2)$$

而且要满足以下条件：

$$\forall a, b \in M_1, F(f_1(a, b)) = f_2(F(a), F(b))$$
$$F(e_1) = e_2$$

两个幺半群之间的函子就是一个幺半群同态（monoid homomorphism）。例如，对于String数据类型，统计单词中字母个数的函数Length()就是一个幺半群同态。

- Length('''') = 0 (空串的长度为0)。
- 如果Length(x) = m而且Length(y) = n，则x + y的连接有m + n个字母。例如：

```
Length("String" + "ology") = Length("Stringology")
                           = 11
                           = 6 + 5
                           = Length("String") + Length("ology")
```

接下来，给出幺半群同态的形式化定义。令M和N是幺半群，如下所示：

$$M: (m, \times, e_m)$$

$$N: (n, +, e_n)$$

则两个幺半群M和N之间的同态是一个函数：

$$f: M \rightarrow N$$

其中：

$$f(x \times y) = f(x) + f(y) \quad \text{对于所有} x, y \in M$$

$$f(e_m) = e_n$$

经典的单词计数（下面称之为f）是一个幺半群同态：

$$f: String \rightarrow Map[String,Integer]$$

令$s \in S$是一个字符串，$S_1 + S_2$表示字符串连接。令$m \in M$是一个计数映射，$m_1 \oplus m_2$表示逐键相加。单词计数(f)函数可以定义如下：

$$f: S \rightarrow M$$

幺半群同态性质可以定义为：

$$f(S_1 + S_2) = f(S_1) \oplus f(S_2)$$

这一章讨论了使用幺半群的MapReduce优化技术。这里展示了通过使用幺半群可以正确地使用组合器（每个集群节点上的本地优化），从而提高密集型计算的性能。第29章将解决Hadoop世界中的“小文件问题”，这是MapReduce编程的另一个优化技术。

# 第29章

# 小文件问题

这一章会为“小文件”问题提供一个高效的解决方案。在MapReduce/Hadoop环境中，什么是小文件？Hadoop世界里，小文件（small file）是指大小远远小于HDFS块大小的文件。默认的HDFS块大小是64 MB（或67108864字节），所以，例如，2MB、5MB或7MB的文件就可以认为是小文件。不过，块大小是可以配置的：这由一个名为dfs.block.size的参数定义。如果一个应用可以处理超大文件（如DNA测序），就可以把这个参数设置得更大，如256 MB。

一般地，Hadoop可以很好地处理大文件，不过当文件很小时，它会把每一个小文件传递到一个map()函数，这样做效率并不高，因为这会创建大量映射器。如果使用和存储小文件，通常就会有很多映射器。例如，对于基因表达式数据类型，表示一个bioset的文件可能有2～3 MB。所以，要处理1000个bioset，就需要1000个映射器（也就是说，每个文件将发送到一个映射器，效率会非常低下）。所以，如果有太多的小文件，在Hadoop环境中这就可能会有问题。要解决这个问题，我们要把多个小文件合并为一个文件，然后再进行处理。对于bioset，可以把20～25个文件合并为一个文件（这样大小就接近64 MB）。通过合并这些文件，最后就只需要40～50个映射器（而不是1000个）。需要说明，Hadoop主要设计用来批处理大量数据，而不是处理很多小文件。解决小文件问题的主要目的是通过将小文件合并为更大的文件来加快Hadoop程序的执行（例如，从3小时缩短到15分钟）。解决小文件问题可以减少map()函数的执行次数，相应地提高Hadoop作业的整体性能。

这一章将为小文件问题提供两个解决方案：

- 解决方案1：使用小文件的定制合并；这个解决方案会在客户端将小文件合并为大文件。

- 解决方案2：使用CombineFileInputFormat<K,V>的一个定制实现[注1]。

# 解决方案1：在客户端合并小文件

假设要处理20000个小文件（每个文件的大小远远小于64 MB），另外我们希望在MapReduce/Hadoop环境中高效地处理这些文件。如果只是直接作为输入通过FileInputFormat.addInputPath(Job, Path)发送这些文件，那么每个输入文件都会发送到一个单独的映射器，最后就会有20000个映射器，效率会非常低下。令dfs.block.size为64MB，进一步假设这些文件的大小在2和3 MB之间（所以假设每个小文件的大小平均为2.5 MB）。另外，假设有M（如100，200，300，...）个可用的映射器。下面的多线程算法（这是一个非MapReduce的POJO解决方案）可以解决小文件问题。由于小文件平均大小为2.5 MB，所以可以把25(25 × 2.5 ≈ 64 MB)个小文件放在一个HDFS块中，我们称之为一个桶（bucket）。现在只需要800(20000 ÷ 25 = 800)个映射器，与20000个映射器相比，这会高效得多。我们的算法将*N*个文件（在这个例子中，就是25个文件）放在各个桶中，然后并发地将这些小文件合并为一个大小接近dfs.block.size的文件。

将小文件提交到MapReduce/Hadoop之前，要把它们合并到大文件中；然后再把这些大文件提交给MapReduce驱动器程序。示例29-1（选自提交MapReduce/Hadoop作业的驱动器程序）展示了如何将小文件合并为一个大文件。

示例29-1：将小文件合并为一个大文件

```
// 准备输入
int NUMBER_OF_MAP_SLOTS_AVAILABLE = <M>;
Job job = <define-a-job>;
List<Path> smallFiles = <HDFS small input files: file1, file2, ...>;
int numberOfSmallFiles = smallFiles.size();
if ( NUMBER_OF_MAP_SLOTS_AVAILABLE >= numberOfSmallFiles ) {
    // 有足够的映射器，所以不需要
    // 合并或组合小文件；每个
    // 小文件会作为一个块发送给一个映射器。
    for (Path path : smallFiles) {
        FileInputFormat.addInputPath(job, path);
    }
}
else {
    // 映射器个数小于小文件数
    // 创建桶，并在桶中填充合并的小文件

    // 步骤1：创建空桶（每个桶可以包含多个小文件）
    BucketThread[] buckets = SmallFilesConsolidator.createBuckets(
              smallFiles,
              NUMBER_OF_MAP_SLOTS_AVAILABLE);

    // 步骤2：用小文件填充桶
```

注1： org.apache.hadoop.mapred.lib.CombineFileInputFormat<K,V>。

```
24    SmallFilesConsolidator.fillBuckets(buckets, smallFiles, job);
25
26    // 步骤3：逐个桶地合并小文件
27    // 每个桶是一个线程（实现Runnable接口）
28    // 各个桶的合并工作将并发完成
29    SmallFilesConsolidator.mergeEachBucket(buckets, job);
30 }
```

SmallFilesConsolidator类接受一组小Hadoop文件，然后将这些小文件合并在一起，构成更大的Hadoop文件，这些文件的大小小于或等于dfs.block.size（也就是HDFS块大小）。最优的解决方案是创建尽可能少的文件（应该记得，对应每个文件会有一个映射器，所以每个文件应当尽可能接近HDFS块大小。这些大文件（作为GUID）在HDFS的/*tmp*/目录下生成（当然，可以配置所使用的目录）：

```
// 这个目录可以配置
private static String MERGED_HDFS_ROOT_DIR = "/tmp/";
...
private static String getParentDir() {
   String guid = UUID.randomUUID().toString();
   return MERGED_HDFS_ROOT_DIR + guid + "/";
}
```

BucketThread类允许我们把小文件连接为一个小于HDFS块大小的大文件。这样一来，可以把这些大的输入文件提交给更少的映射器。BucketThread类实现了Runnable接口，还提供了copyMerge()方法，它会把小文件合并为一个大文件。由于每个BucketThread对象实现了Runnable接口，所以它会在自己的线程中运行。这样一来，所有BucketThread对象会并发地合并它们的小文件。BucketThread.copyMerge()是核心的方法，它会把一个桶中的所有小文件合并为另一个临时HDFS文件。例如，如果一个桶中包含小文件{File1, File2, File3, File4}，那么合并得到的文件则如图29-1所示（注意，MergedFile是所有4个小文件连接的结果）。

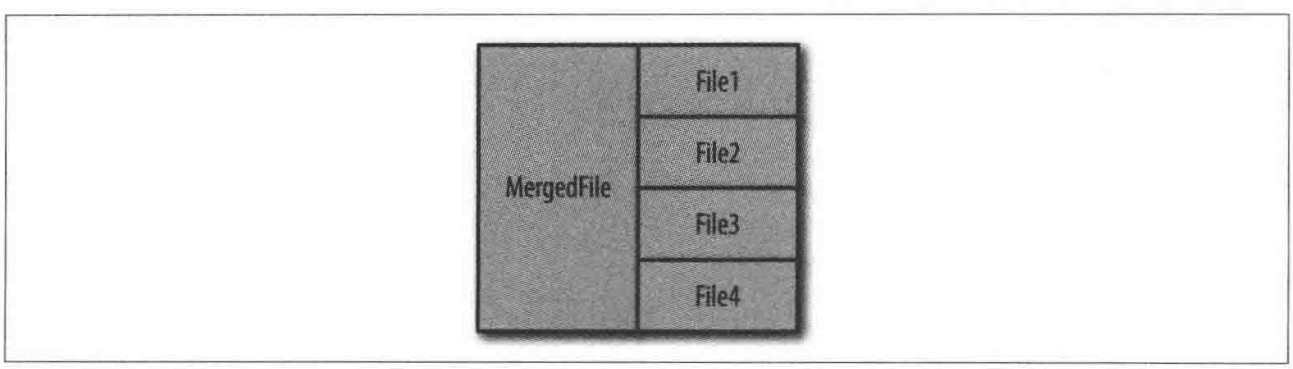



图29-1：小文件合并为更大的文件

示例29-2显示了BucketThread.copyMerge()方法如何实现。

示例29-2：copyMerge()方法

```
/**
 * 将多个目录中的所有文件复制到一个输出文件(mergedFile)。
 *
 * parentDir将是"/tmp/<guid>/"
 * targetDir将是"/tmp/<guid>/id/"
 * targetFile将是"/tmp/<guid>/id/id"
 *
 * 合并桶中的所有路径，并返回一个新目录
 * (targetDir)，其中包含合并的路径
 */
public void copyMerge() throws IOException {

    // 如果桶中只有一个路径/目录，
    // 则没有必要合并
    if ( size() < 2 ) {
       return;
    }

    // 这里bucket.size() >= 2
    Path hdfsTargetFile = new Path(targetFile);
    OutputStream mergedFile = fs.create(hdfsTargetFile);
    try {
        for (int i = 0; i < bucket.size(); i++) {
            FileStatus contents[] = fs.listStatus(bucket.get(i));
            for (int k = 0; k < contents.length; k++) {
               if (!contents[k].isDir()) {
                   InputStream smallFile = fs.open(contents[k].getPath());
                   try {
                       IOUtils.copyBytes(smallFile, mergedFile, conf, false);
                   }
                   finally {
                     HadoopUtil.close(smallFile);
                   }
               }
         } // for k
      } // for i
    }
    finally {
      HadoopUtil.close(mergedFile);
    }
}
```

因此SmallFilesConsolidator类提供了3个功能：

1. 创建所需的空桶。每个桶将包含一组小文件。这将由SmallFilesConsolidator.createBuckets()完成。
2. 填充桶。我们将把足够多的小文件放在一个桶中，使得所有小文件的总大小约为dfs.block.size。这由SmallFilesConsolidator.fillBuckets()实现。
3. 合并各个桶。这里我们将合并桶中的所有小文件来创建一个大文件，其大小约为dfs.block.size。这由SmallFilesConsolidator.mergeEachBucket()完成。

为了说明小文件问题，我们将分别使用和不使用SmallFilesConsolidator类来运行经典的单词计数程序。作为输入，两种情况下都将使用30个小文件。可以清楚地看到，使用SmallFilesConsolidator时，性能远远优于原来的解决方案；它会在58235毫秒内完成，而原来使用小文件的单词计数程序需要80435毫秒才能完成。

## 输入数据

两个解决方案都使用以下输入数据（30个小文件）：

```
# hadoop fs -ls /small_input_files/input/
Found 30 items
-rw-r--r-- 1 ... /small_input_files/input/Document-1
-rw-r--r-- 1 ... /small_input_files/input/Document-2
...
-rw-r--r-- 1 ... /small_input_files/input/Document-29
-rw-r--r-- 1 ... /small_input_files/input/Document-30
```

## 使用 SmallFilesConsolidator的解决方案

在这个解决方案中，我们使用SmallFilesConsolidator类将小文件合并为一个更大的文件。

### Hadoop实现类

表29-1给出了这个解决方案中需要的Java类。

**表29-1：使用SmallFilesConsolidator的解决方案中需要的Java类**

| 类名 | 类描述 |
|---|---|
| BucketThread | 用来将小文件合并为更大的文件 |
| HadoopUtil | 定义一些基本Hadoop工具 |
| SmallFilesConsolidator | 管理小文件合并为更大的文件 |
| WordCountDriverWithConsolidator | 使用合并器的单词计数驱动器 |
| WordCountMapper | 定义map() |
| WordCountReducer | 定义reduce()和combine() |

SmallFilesConsolidator类是合并小文件的驱动器类，它将小文件合并为一个大小接近HDFS块大小的更大的文件。主要方法包括：

getNumberOfBuckets()

确定将所有文件合并到更大文件所需的桶数。

```
public static int getNumberOfBuckets(int totalFiles,
                                      int numberOfMapSlotsAvailable,
                                      int maxFilesPerBucket)
```

createBuckets()

创建所需的桶。

```
public static BucketThread[] createBuckets(
                                    int totalFiles,
                                    int numberOfMapSlotsAvailable,
                                    int maxFilesPerBucket)
```

fillBuckets()

用小文件填充各个桶。

```
public static void fillBuckets(
        BucketThread[] buckets,
        List<String> smallFiles, // 小文件列表
        Job job,
        int maxFilesPerBucket)
```

mergeEachBucket()

合并小文件来创建一个更大的文件。

```
public static void mergeEachBucket(BucketThread[] buckets,
                                   Job job)
```

## 运行示例

下面给出这个解决方案的一个运行示例（由于篇幅所限，这里做了编辑和格式调整）：

```
# ./run_with_consolidator.sh
...
Deleted hdfs://localhost:9000/small_input_files/output
13/11/05 10:54:04 ...: inputDir=/small_input_files/input
13/11/05 10:54:04 ...: outputDir=/small_input_files/output
13/11/05 10:54:05 ...added path: /tmp/906e6c30-c411-4a70-b68f-114ba7511e63/
...
13/11/05 10:54:05 ...added path: /tmp/906e6c30-c411-4a70-b68f-114ba7511e63/
...
13/11/05 10:54:05 INFO input.FileInputFormat: Total input paths to process : 8
...
13/11/05 10:54:05 INFO mapred.JobClient: Running job: job_201311051023_0002
13/11/05 10:54:06 INFO mapred.JobClient: map 0% reduce 0%
...
13/11/05 10:55:01 INFO mapred.JobClient: map 100% reduce 100%
13/11/05 10:55:02 INFO mapred.JobClient: Job complete: job_201311051023_0002
13/11/05 10:55:02 INFO mapred.JobClient: Launched reduce tasks=10
13/11/05 10:55:02 INFO mapred.JobClient: Launched map tasks=8
13/11/05 10:55:02 INFO mapred.JobClient: Data-local map tasks=8
...
13/11/05 10:55:02 INFO mapred.JobClient: Map input records=48
13/11/05 10:55:02 INFO mapred.JobClient: Reduce input records=48
13/11/05 10:55:02 INFO mapred.JobClient: Reduce input groups=7
13/11/05 10:55:02 INFO mapred.JobClient: Reduce output records=7
13/11/05 10:55:02 INFO mapred.JobClient: Map output records=201
13/11/05 10:55:02 INFO WordCountDriverWithConsolidator: returnStatus=0
```

```
13/11/05 10:55:02 INFO WordCountDriverWithConsolidator:
   Finished in milliseconds: 58235
```

从运行示例的日志可以看到，这里将30个HDFS小文件合并为8个HDFS大文件。

# 不使用SmallFilesConsolidator的解决方案

这个解决方案就是一个基本的单词计数应用，没有使用SmallFilesConsolidator类。从运行示例的以下片段可以看到，要处理的输入路径的总数为30，这正好是要处理的小文件个数：

```
...
13/11/05 10:29:13 INFO input.FileInputFormat: Total input paths to process : 30
...
```

这个解决方案并不是一个最优方案，因为每个小文件都会分别发送到一个映射器。我们知道，理想情况下，发送的输入文件大小应当稍小于或等于HDFS块大小（因为Hadoop就是设计用来处理大文件的）。

## Hadoop实现类

表29-2显示了这个不使用SmallFilesConsolidator的解决方案所需的Java类。

**表29-2：不使用SmallFilesConsolidator的解决方案所需的Java类**

| 类名 | 类描述 |
|---|---|
| HadoopUtil | 定义一些基本Hadoop工具 |
| WordCountDriverWithoutConsolidator | 不使用合并器的单词计数驱动器 |
| WordCountMapper | 定义map() |
| WordCountReducer | 定义reduce()和combine() |

## 运行示例

下面是这个不使用SmallFilesConsolidator的解决方案的一个运行示例（由于篇幅所限，这里做了编辑和格式调整）：

```
# ./run_without_consolidator.sh
...
Deleted hdfs://localhost:9000/small_input_files/output
13/11/05 10:29:12 ... inputDir=/small_input_files/input
13/11/05 10:29:12 ... outputDir=/small_input_files/output
...
13/11/05 10:29:13 INFO input.FileInputFormat: Total input paths to process : 30
...
13/11/05 10:29:13 INFO mapred.JobClient: Running job: job_201311051023_0001
13/11/05 10:29:14 INFO mapred.JobClient: map 0% reduce 0%
...
```

```
13/11/05 10:30:32 INFO mapred.JobClient: map 100% reduce 100%
13/11/05 10:30:33 INFO mapred.JobClient: Job complete: job_201311051023_0001
...
13/11/05 10:30:33 INFO mapred.JobClient: Map-Reduce Framework
13/11/05 10:30:33 INFO mapred.JobClient: Map input records=48
13/11/05 10:30:33 INFO mapred.JobClient: Reduce input records=153
13/11/05 10:30:33 INFO mapred.JobClient: Reduce input groups=7
13/11/05 10:30:33 INFO mapred.JobClient: Combine output records=153
13/11/05 10:30:33 INFO mapred.JobClient: Reduce output records=7
13/11/05 10:30:33 INFO mapred.JobClient: Map output records=201
13/11/05 10:30:33 INFO WordCountDriverWithoutConsolidator:
   run(): status=true
13/11/05 10:30:33 INFO WordCountDriverWithoutConsolidator:
   Finished in milliseconds: 80435
```

# 解决方案2：用CombineFileInputFormat解决小文件问题

这一节使用Hadoop API（抽象类CombineFileInputFormat）来解决小文件问题。Hadoop 2.5.0中的CombineFileInputFormat抽象类定义如下：

```
package org.apache.hadoop.mapred.lib;
...
@InterfaceAudience.Public
@InterfaceStability.Stable
public abstract class CombineFileInputFormat<K,V>
extends CombineFileInputFormat<K,V>
implements InputFormat<K,V>
```

抽象类CombineFileInputFormat的基本思想是通过使用一个定制InputFormat允许将小文件合并到Hadoop的分片（split）或块（chunk）中。要使用抽象类CombineFileInputFormat，需要提供/实现3个定制类：

- CustomCFIF要扩展CombineFileInputFormat （CombineFileInputFormat是一个没有提供具体实现的抽象类，所以必须创建子类来支持定制输入格式）。
- PairOfStringLong是一个Writable类，会存储小文件名（String)及其偏移量（Long），另外覆盖compareTo()方法：首先比较文件名，再比较偏移量。
- CustomRecordReader是一个定制RecordReader：

```
public class CustomRecordReader
    extends RecordReader<PairOfStringLong, Text> {
    ...
}
```

CombineFileInputFormat的定制实现在示例29-3中给出。

示例29-3：CustomCFIF类

```
import java.io.IOException;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.InputSplit;
import org.apache.hadoop.mapreduce.JobContext;
import org.apache.hadoop.mapreduce.RecordReader;
import org.apache.hadoop.mapreduce.TaskAttemptContext;
import org.apache.hadoop.mapreduce.lib.input.CombineFileSplit;
import org.apache.hadoop.mapreduce.lib.input.CombineFileInputFormat;
import org.apache.hadoop.mapreduce.lib.input.CombineFileRecordReader;

import edu.umd.cloud9.io.pair.PairOfStringLong;
// PairOfStringLong = Tuple2<String, Long> = Tuple2<FileName, Offset>
// https://github.com/lintool/Cloud9/

/**
 * 定制文件输入格式，将较小的文件组合/合并
 * 到大文件，文件大小由 MAX_SPLIT_SIZE控制
 *
 * @author Mahmoud Parsian
 *
 */
public class CustomCFIF extends CombineFileInputFormat<PairOfStringLong, Text> {
   final static long MAX_SPLIT_SIZE = 67108864; // 64 MB

   public CustomCFIF() {
      super();
      setMaxSplitSize(MAX_SPLIT_SIZE);
   }

   public RecordReader<PairOfStringLong, Text> createRecordReader
        (InputSplit split,
         TaskAttemptContext context)
      throws IOException {
      return new CombineFileRecordReader<PairOfStringLong, Text>(
        (CombineFileSplit)split,
        context,
        CustomRecordReader.class);
   }

   @Override
   protected boolean isSplitable(JobContext context, Path file) {
      return false;
   }
}
```

要根据HDFS块大小（默认为64 MB）来设置MAX_SPLIT_SIZE。如果大多数文件都大于64 MB，就可以将HDFS块大小设置为128MB，甚至256MB （在一些基因组中，HDFS块大小甚至设置为512MB）。在Hadoop 2.5.1中，HDFS块大小默认设置为128 MB（134217728字节）。可以在*hdfs-site.xml*文件（这是配置Hadoop群集的文件之一）中通过dfs.blocksize属性控制HDFS块大小：

```
$ cat $HADOOP_HOME/etc/hadoop/hdfs-site.xml
<?xml version="1.0" encoding="UTF-8"?>

<configuration>

   <property>
      <name>dfs.blocksize</name>
      <value>268435456</value>
      <description>256MB</description>
   </property>

   <property>
   ...
   </property>

   ...

</configuration>
```

设置最大分片大小（MAX_SPLIT_SIZE）将确定所需的映射器个数。例如，考虑以下HDFS目录[注2]：

```
# hadoop fs -ls /small_input_files | wc -l
10004

# hadoop fs -ls /small_input_files | head -3
-rw-r--r-- 3 ... 9184 2014-10-06 15:20 /small_input_files/file1.txt
-rw-r--r-- 3 ... 27552 2014-10-06 15:20 /small_input_files/file2.txt
-rw-r--r-- 3 ... 27552 2014-10-06 15:20 /small_input_files/file3.txt

# hadoop fs -ls /small_input_files | tail -3
-rw-r--r-- 3 ... 27552 2014-10-06 15:28 /small_input_files/file10002.txt
-rw-r--r-- 3 ... 27552 2014-10-06 15:28 /small_input_files/file10003.txt
-rw-r--r-- 3 ... 27552 2014-10-06 15:28 /small_input_files/file10004.txt

# hadoop fs -dus /small_input_files
275584288 /small_input_files
#
```

可以看到，HDFS目录*/small_input_files*有10004个小文件，需要275584288字节。如果不使用CustomCFIF作为输入格式，那么基本MapReduce作业就要使用10004个映射器（在一个包括3节点的集群上执行这个作业需要34分钟）。不过，如果使用CustomCFIF，我们只需要5个映射器（在包括3节点的集群上执行这个作业只需要不到2分钟）。为什么需要5个映射器呢？下面的计算可以回答这个问题：

```
HDFS-split-size = 64MB = 64*1024*1024 = 67108864
Required Bytes for 10004 small files = 275584288
275584288/67108864 = 4
Therefore we need 5 splits:
```

注2：为了进行测试，可以使用bash创建大量小文件，有关的详细信息参见*http://bit.ly/many_small_files*。

```
    67108864+67108864+67108864+67108864+7148832 = 275584288

Therefore, 5 input splits are required
=> this will launch 5 mappers (one per split)
```

如果把MAX_SPLIT_SIZE设置为128 MB（134217728字节），那么这个Hadoop作业只需要启动3个映射器，如下：

```
HDFS-split-size = 128MB = 128*1024*1024 = 134217728
Required Bytes for 10004 small files = 275584288
275584288/134217728 = 2
Therefore we need 3 splits:
     134217728+134217728+7148832 = 275584288

Therefore, 3 input splits are required
=> this will launch 3 mappers (one per split)
```

## 定制CombineFileInputFormat

CustomCFIF类将小文件组合到分片中（分片由MAX_SPLIT_SIZE确定，这实际上就是合并小文件构成的更大文件的最大大小），从而解决小文件问题。这个定制类（扩展了抽象类CombineFileInputFormat）有以下功能：

- 在构造函数中调用setMaxSplitSize(MAX_SPLIT_SIZE)设置最大分片大小。小文件的最大组合不能超过这个大小。
- 由createRecordReader()定义一个定制记录阅读器，然后提供一个插件类CustomRecordReader，它将小文件读取到大分片（最大大小由MAX_SPLIT_SIZE确定）。
- 定义输入到映射器的键-值对。我们使用PairOfStringLong作为键，使用Text（文本文件中的一行）作为值。PairOfStringLong表示两部分信息：文件名（String）和偏移量（Long）。
- 指示组合/合并的文件不应分片，这由isSplitable()方法设置，它会返回false。

## 使用CustomCFIF的运行示例

下面各小节提供了使用CustomCFIF的运行示例的相应脚本、运行日志和输出（由于篇幅所限，这里对输出做了编辑和格式调整）。

### 脚本

```
# cat run_combine_small_files.sh
#!/bin/bash
BOOK_HOME=/mp/data-algorithms-book
CLASSPATH=.:$BOOK_HOME/dist/data_algorithms_book.jar
APP_JAR=$BOOK_HOME/dist/data_algorithms_book.jar
```

```
CLASSPATH=$CLASSPATH:$BOOK_HOME/lib/spark-assembly-1.2.0-hadoop2.6.0.jar
INPUT=/small_input_files
OUTPUT=/output/1
PROG=org.dataalgorithms.chap29.combinesmallfiles.CombineSmallFilesDriver
hadoop jar $APP_JAR $PROG $INPUT $OUTPUT
```

## 运行示例日志

```
# ./run_combine_small_files.sh
input path = /small_input_files
output path = /output/1
14/10/06 15:51:39 INFO input.FileInputFormat:
..Total input paths to process : 10003
14/10/06 15:51:40 INFO input.CombineFileInputFormat:
  DEBUG: Terminated node allocation
  with : CompletedNodes: 3, size left: 7108416
14/10/06 15:51:40 INFO mapreduce.JobSubmitter: number of splits:5
...
14/10/06 15:51:41 INFO impl.YarnClientImpl:
  Submitted application application_1411149084067_0006
14/10/06 15:51:41 INFO mapreduce.Job: The url to track the job:
  http://myserver100:8088/proxy/application_1411149084067_0006/
14/10/06 15:51:41 INFO mapreduce.Job: Running job: job_1411149084067_0006
14/10/06 15:51:49 INFO mapreduce.Job: Job job_1411149084067_0006
  running in uber mode : false
14/10/06 15:51:49 INFO mapreduce.Job: map 0% reduce 0%
...
14/10/06 15:54:07 INFO mapreduce.Job: map 100% reduce 100%
14/10/06 15:54:08 INFO mapreduce.Job: Job job_1411149084067_0006
   completed successfully
14/10/06 15:54:08 INFO mapreduce.Job: Counters: 50
...
        Job Counters
                Launched map tasks=5
                Launched reduce tasks=12
...
        Map-Reduce Framework
                Map input records=1650385
                Map output records=41499681
                Reduce input groups=617
                Reduce input records=41499681
                Reduce output records=617
                Shuffled Maps =60
                Merged Map outputs=60
                CPU time spent (ms)=822100
                Total committed heap usage (bytes)=18386780160
```

## 输出

```
# hadoop fs -ls /output/1
Found 13 items
-rw-r--r-- 3 hadoop,hadoop 0 2014-10-06 15:54 /output/1/_SUCCESS
-rw-r--r-- 3 hadoop,hadoop 647 2014-10-06 15:53 /output/1/part-r-00000
-rw-r--r-- 3 hadoop,hadoop 778 2014-10-06 15:53 /output/1/part-r-00001
```

```
...
-rw-r--r-- 3 hadoop,hadoop 767 2014-10-06 15:54 /output/1/part-r-00010
-rw-r--r-- 3 hadoop,hadoop 534 2014-10-06 15:54 /output/1/part-r-00011
# hadoop fs -cat /output/1/part-r-00000
...
about 30007
against 30007
also 120028
assay 60014
cannot 30007
carefully 30007
...
```

### Hadoop URL的输出

```
Job Output from Hadoop URL:
http://myserver100:8088/cluster/app/application_1411149084067_0007

User: hadoop
Name: CombineSmallFilesDriver
Application Type: MAPREDUCE
Application Tags:
State: FINISHED
FinalStatus: SUCCEEDED
Started: 6-Oct-2014 16:25:02
Elapsed: 1mins, 54sec
Tracking URL: History
Diagnostics:
```

## 其他解决方案

filecrush工具（*http://bit.ly/filecrush*）也可以作为解决小文件问题的解决方案。它会把多个小文件合并为更少的大文件。另外，filecrush工具可以将文件从文本改为SequenceFile格式（二进制键-值文件），而且可以在一趟处理中执行其他压缩处理。

这一章为“小文件”问题提供了两个可伸缩的解决方案。利用这里提供的解决方案，可以对使用大量小文件的MapReduce作业进行适当的优化。第30章会讨论MapReduce作业的另一种优化：对map()和reduce()函数使用“大容量缓存”。

# 第30章

# MapReduce的大容量缓存

这一章会介绍如何在MapReduce算法中使用和读取大容量缓存（也就是说，可能包括数十亿键-值对，而无法放在一个商用服务器的内存中）。这一章提供的算法非常通用，可以在任何MapReduce范式中使用（如MapReduce/Hadoop和Spark）。

有些MapReduce算法可能需要访问一些庞大的静态参考关系表（可能包含数十亿条记录）。一般来讲，这些参考关系表在很长一段时间内都不会改变，不过在MapReduce程序的map()或reduce()阶段会用到。“位置特性”表就是这样一个例子，用于生殖细胞[注1]数据类型采集和变体分类（variant classification）。位置特性表可能有表30-1所示的性质（组合键为(chromosome_id, position)）。

**表30-1：位置特性表的性质**

| 列名 | 性质 |
|---|---|
| chromosome_id | Key-1 |
| position | Key-2 |
| feature_id | 基本属性 |
| mrna_feature_id | 基本属性 |
| sequence_data_type_id | 基本属性 |
| mapping | 基本属性 |

描述MapReduce解决方案时（可能在map()或reduce()中），给定一个key=(chromosome_id, position)，可能希望返回一个List<String>，这个列表的各个元素由其余的属性组

注1：生殖细胞（Germline）是指从受精卵到原始生殖细胞的包括所有能够形成配子的细胞，有别于体细胞（所有其他体细胞）。生殖细胞突变会转移到后代，体细胞突变则不会（资料来源：*http://bit.ly/germline_def*）。

成{feature_id, mrna_feature_id, sequence_data_type_id, mapping}。对于生殖细胞数据类型，一个位置特性表可能有多达120亿个记录（在MySQL或Oracle数据库系统中，这可能约占2 TB的磁盘空间）。现在假设映射器或归约器希望访问这个位置特性表来查找给定key=(chromosome_id, position)。由于可能会发出很多这种类型的请求（每秒数百万个），这会让数据库服务器疲于奔命，而且这种做法不可伸缩。一个可能的解决方案是将所有静态数据缓存在一个散列表中，然后使用这个散列表而不是一个关系表。不过，在下一节可以看到，这并不是一个合适的解决方案，它不具有可伸缩性，这样一个散列表的大小可能超过4 TB (散列表的元数据会占据很大空间)，因此无法放在当今商用服务器的内存中。

# 实现方案

那么，要想在MapReduce环境中缓存120亿个记录，有哪些实用的最优方案呢？在这一节中，我会给出一些方案，并讨论这些方案实现的可行性：

*方案1*

使用一个关系数据库（如MySQL或Oracle）。这并不是一个可靠的解决方案，不能很好地伸缩。映射器或归约器会不断地访问数据库，数据库服务器无法每秒处理成千上万甚至数百万的请求。尽管可以使用一组复制关系数据库，这个方案仍不能很好地横向扩容（因为即使对于一组庞大的Hadoop集群，数据库连接数仍是有限的）。

*方案2*

使用一个memcached（内存缓存）服务器。Memcached（*http://memcached.org/*）是一个内存中的键-值存储库，用于存储由数据库调用、API调用或页面呈现的结果得到的小块任意数据（例如，字符串、对象）。如果每几个从节点就有一个专用的内存缓存集群（在一个大规模集群环境中这样成本会很高），这就是一个可靠而且合适的方案。但是由于需要太多昂贵的内存缓存服务器，这个方案不能很好地横向扩容。

*方案3*

使用一个Redis服务器。Redis（*http://redis.io/*）是一个有BSD许可的开源高级键-值存储库。通常称之为数据结构服务器（data structure server），因为键可以包含字符串、散列、列表、集合和有序集。类似于内存缓存，如果对于每几个从节点有一个专用的Redis服务器（在一个大规模集群环境中这样成本会很高），这就是一个可靠而且合适的方案。不过，同样的，由于需要太多的Redis服务器（对于2 TB的键-值对，Redis至少需要12 TB的RAM），这个方案不能很好地横向扩容。

*方案4*

在输入和120亿静态记录之间使用一个MapReduce连接。具体想法是扁平化120亿记录，然后在这些扁平化的记录和输入之间使用一个MapReduce连接。在key=(chromosome_id, position)上完成连接。如果输入也很庞大（这说明有数十亿

记录而不只是几百万记录），这就是一个可行的方案。例如，对于一个生殖细胞采集过程，VCF[注2]文件的记录不会有超过600万条，而将一个有600万记录的表与120亿记录连接是不合适的（因为这会浪费大量时间）。

*方案5*

将120亿静态记录划分为小块（每个块有64MB，可以很容易地将这些小块加载到RAM，而且不再需要这些小块时可以再将其从RAM移出），并使用一个LRU-Map散列表(LRU表示最近最少使用（least recently used））。这个想法很简单，但很巧妙，也很实用：它能很好地横向扩容，而不需要专门的缓存服务器。这个解决方案将在后面的小节中详细讨论。不管怎样，在一个分布式环境中，映射器或归约器不可能访问无限量的内存/RAM。这个解决方案很适用于MapReduce环境，而且不需要额外的RAM来实现缓存。LRU映射是Linux上的一个本地缓存，每个集群工作节点都有这个缓存的一个相同的副本（占据不到1 TB的硬盘空间）。与常态硬盘相比，使用固态硬盘可以让缓存性能提高8到10倍。

# 缓存问题形式化描述

基本说来，给定一组120亿个记录，我们的目标是找出对应给定组合键key=(chromosome_id, position)的关联值。假设要把一个VCF文件（这是我们的输入文件，包括对应生殖细胞数据类型的约500万个记录）采集到基因组系统。对于一个给定VCF的每一个记录，需要找出：

- 突变类（Mutation class）。
- 基因列表（List of genes）。
- 特性列表（List of features）。

要找出每个VCF记录的这些详细信息，需要一组静态表，例如这一章开始时简单介绍的位置特性表。如前所述，位置特性表可能包含多达120亿个记录。生殖细胞采集过程的第一步就是从一个位置特性表找出对应一个给定key=(chromosome_id, position)的记录（每个VCF记录都包含chromosome_id和position字段）。

# 一个精巧、可伸缩的解决方案

这里给出的解决方案是一个本地缓存解决方案。这说明，一个MapReduce/Hadoop或Spark集群的每一个工作/从节点都有自己的本地缓存（位于硬盘上，成本廉价），而且无需网

注2：变体调用格式（Variant call format）是一个文本文件格式。包含元信息行、一个标题行和数据行，每个数据行包含基因组中一个位置的有关信息。有关的详细信息，参见*http://bit.ly/variant_call_format*。

络流量来访问缓存组件和数据。本地缓存位于一个SSD或HDD中，通过LRU映射缓存技术按需加载到内存。假设有一个名为position_feature的关系表（这一章最前面给出了这个关系表的定义），其中包含超过120亿记录，它的工作如下：

1. 首先，按chromosome_id（1, 2, 3, ..., 24, 25）划分这120亿个记录。这会得到25个文件（假设这些文件分别名为*chr1.txt*, *chr2.txt*, ..., *chr25.txt*），其中每个记录有以下格式（需要说明，根据key=(chromosome_id,position)，可能会从关系表返回多个记录）：

   ```
   <position><;><Record1><:><Record2><:>...<:><RecordN>
   ```

   而且各个Record<i>包括以下内容：

   ```
   <feature_id><,><mrna_fea
   ture_id><,><sequence_data_type_id><,><mapping>
   ```

   因此，每个原始缓存数据文件（*chr1.txt*, *chr2.txt*, ..., *chr25.txt*）对应以下SQL查询（可以对每个chromosome_id重复这个脚本；下面的查询对应chromosome_id=1）：

   ```
   select position,
         GROUP_CONCAT(feature_id, ',',
                      mrna_feature_id, ',',
                      seq_datatype_id, ',',
                      mapping SEPARATOR ':')
         INTO OUTFILE '/tmp/chr1.txt'
         FIELDS TERMINATED BY ';'
         LINES TERMINATED BY '\n'
      from position_feature where chromosome_id = 1 Group By position;
   ```

2. 接下来，分别对这些文件（*chr1.txt*, *chr2.txt*, ..., *chr25.txt*）按position排序，生成*chr1.txt.sorted*, *chr2.txt.sorted*, ..., *chr25.txt.sorted*。

3. 由于对应一个MapReduce作业的各个映射器/归约器的内存是有限的（假设不超过4 GB），要把各个已排序文件划分为64MB的块（但不能中断记录行）。为此，要执行以下命令：

   ```
   #!/bin/bash
   sorted=/data/positionfeature.sorted
   output=/data/partitioned/
   for i in {1..25} ; do
      echo "i=$i"
      mkdir -p $output/$i
      cd $output/$i/
      split -a 3 -d -C 64m $sorted/$i.txt.sorted $i.
   done
   exit
   ```

   例如，对于chromosome_id=1，就会得到：

   ```
   # ls -l /data/partitioned/1/
   -rw-rw-r-- 1 hadoop hadoop 67108634 Feb 2 09:30 1.000
   ```

```
-rw-rw-r-- 1 hadoop hadoop 67108600 Feb 2 09:30 1.001
-rw-rw-r-- 1 hadoop hadoop 67108689 Feb 2 09:30 1.002
...
-rw-rw-r-- 1 hadoop hadoop 11645141 Feb 2 09:33 1.292
```

需要说明，这些分区文件分别有一个position值范围（因为它们按position排序），可以在缓存实现中使用这些范围。因此，给定chromosome_id=1和一个position，可以准确地知道哪个分区包含所查询的结果。下面来看其中一个已排序分区文件的内容：

```
# head -2 /data/partitioned/1/1.000
6869;35304872,35275845,2,1
6870;35304872,35275845,2,1

# tail -2 /data/partitioned/1/1.000
790279;115457,21895578,12,2:115457,35079912,3,3:...
790280;115457,21895578,12,2:115457,35079912,3,3:...
```

可以看到，各个分区中所有位置都是有序的。对于使用LRU Map的所有分区，为了支持元数据，还需要另外一个数据结构跟踪（begin, end）位置。对于每个分区文件，我们会记录（partition name, begin, end）信息。例如，对于chromosome id=1和chromosome id=2，可以得到：

```
# cat 1/begin_end_position.txt
1.000;6869;790280
1.001;790281;1209371
1.002;1209372;1461090
...
1.292;249146130;249236242
# cat 2/begin_end_position.txt
2.000;33814;1010683
2.001;1010684;1494487
2.002;1494488;2132388
...
2.279;242617420;243107469
```

分区数据结构如图30-1所示。

4. 接下来，要把各个分区（64 MB）转换为一个散列表，由MapDB实现[注3]。

对于每个映射器/归约器来说，内存是有限的，所以我们使用LRUMap<K,V>(N)来存储最多N个MapDB数据结构（每个MapDB持久存储的映射对应一个64 MB的有序分区）。LRUMap<K,V>的想法是保存最多N个分区，保证N × 64 MB小于各个映射器/归约器可用的内存。将第N + 1个记录插入到LRUMap<K,V>(N)时，最老的记录会移出内存（然后可以适当地关闭MapDB对象，这会释放所有内存，并关闭所有文件句柄）。我们使用org.apache.commons.collections4.map.LRUMap<K,V>来提供LRU Map实现。

注3：MapDB由Jan Kotek实现，源代码存放在*https://github.com/jankotek/MapDB*。MapDB提供了并发映射、集合和队列，由磁盘存储空间或离堆内存提供后备支持。这是一个快速而且易于使用的嵌入式Java数据库引擎。

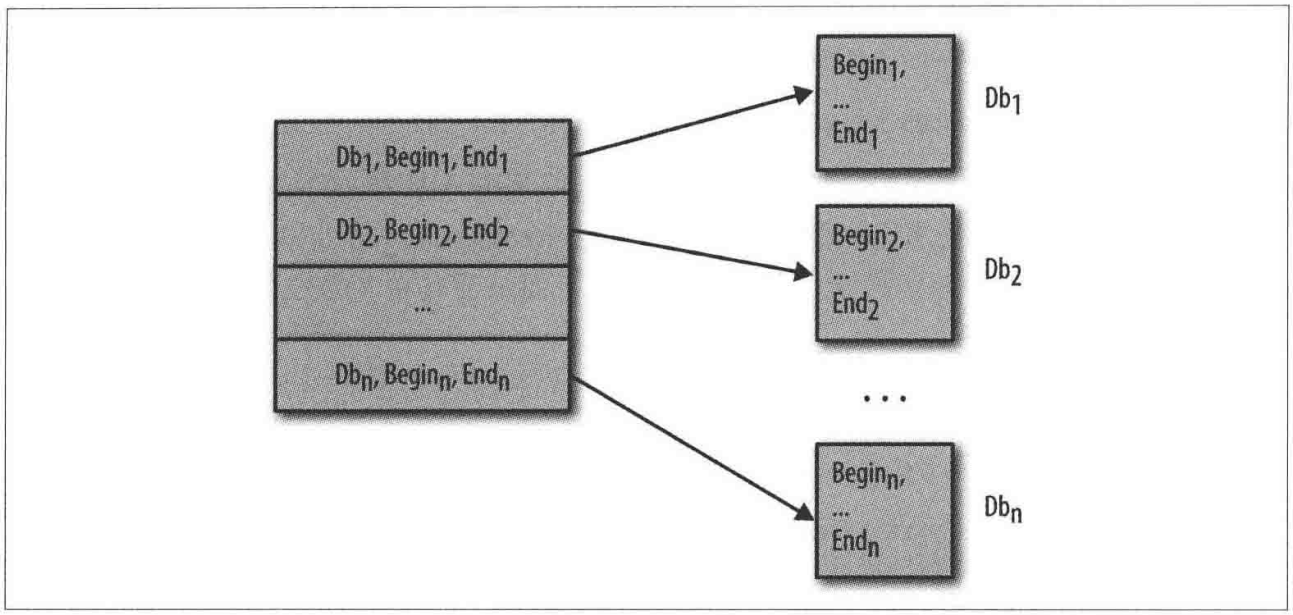



图30-1：超大缓存分区数据结构

5. 最后一步是按key=(chromosome_id, position)对输入进行排序（这会使用缓存。需要说明，对于VCF文件等输入，可以在不到3秒时间内完成排序）。通过排序，可以尽可能减少从LRUMap<K,V>(N)移出MapDB记录。输入文件的排序是一个很重要的设计/实现标准，这对于MapReduce作业（生殖细胞采集）的性能有很大影响。如果没有对输入文件排序，移出率可能非常高。反之，如果对输入文件排序，就可以尽可能减少移出率。需要说明，输入文件的排序速度很快；应该不会超过几秒的时间（可以通过一个Linux sort命令完成，或者使用MapReduce/Hadoop来实现）。

# 实现LRUMap缓存

这一节中，我们会逐步介绍如何实现这个精巧、可伸缩的MapReduce解决方案来管理大缓存。

## 扩展LRUMap类

我选择了LRUMap<K,V>[注4]来实现LRU映射，这是一个Map实现，最大大小是固定的，如果映射已满，还要增加一个映射项，就会删除最近最少使用的（LRU）项。LRU映射算法只适用于get和put操作。任何迭代处理（包括迭代设置值）都不会改变项的顺序（有关的详细信息可以参考Apache Commons Collections）。在这个缓存实现中，我们扩展了LRUMap类（MapDBEntry类是一个简单类，它将一个64 MB的有序分区表示为MapDB中实现的一个Map数据结构）。参见示例30-1。

注4：org.apache.commons.collections4.map.LRUMap。

示例30-1：定制LRUMap

```
import org.apache.commons.collections4.map.LRUMap;
import org.apache.commons.collections4.map.AbstractLinkedMap.LinkEntry;

public class CustomLRUMap<K, V> extends LRUMap<K, V> {

    private K key = null;
    private V value = null;
    private LinkEntry<K, V> entry = null;

    public CustomLRUMap(final int size) {
        super(size);
    }

    @Override
    protected boolean removeLRU(final LinkEntry<K, V> entry) {
        System.out.println("begin remove LRU entry ...");
        this.entry = entry;
        this.key = entry.getKey();
        this.value = entry.getValue();

        if (key instanceof String) {
            String keyAsString = (String) key;
            System.out.println("evicting key="+keyAsString);
        }

        if (value instanceof MapDBEntry) {
            // 释放MapDBEntry占用的资源
            MapDBEntry mapdbEntry = (MapDBEntry) value;
            mapdbEntry.close();
        }

        return true; // 从LRU映射删除项
    }
}
```

## 测试定制类

示例30-2显示了CustomLRUMap<K,V>如何工作。CustomLRUMap<K,V>类扩展了LRUMap<K,V>类，并且重新定义了removeLRU()方法的删除策略。在这个例子中，我们每次只保留3个映射项。不论为LRUMap对象增加多少个映射项，LRUMap.size()都不能超过3（创建CustomLRUMap<K,V>对象时会指定这个最大大小）。

示例30-2：定制LRUMap测试

```
# cat CustomLRUMapTest.java

import org.apache.commons.collections4.map.LRUMap;

public class CustomLRUMapTest {
   public static void main(String[] args) throws Exception {
      CustomLRUMap<String, String> map = new CustomLRUMap<String, String>(3);
```

```
 8        map.put("k1", "V1");
 9        map.put("k2", "V2");
10        map.put("k3", "v3");
11        System.out.println("map="+map);
12        map.put("k4", "v4");
13        String v = map.get("k2");
14        System.out.println("v="+v);
15        System.out.println("map="+map);
16        map.put("k5", "v5");
17        System.out.println("map="+map);
18        map.put("k6", "v6");
19        System.out.println("map="+map);
20    }
21 }
```

运行这个测试会生成以下输出：

```
# javac CustomLRUMapTest.java
# java CustomLRUMapTest
map={k1=V1, k2=V2, k3=v3}
begin removeLRU...
evicting key=k1
v=V2
map={k3=v3, k4=v4, k2=V2}
begin removeLRU...
evicting key=k3
map={k4=v4, k2=V2, k5=v5}
begin removeLRU...
evicting key=k4
map={k2=V2, k5=v5, k6=v6}
```

## MapDBEntry类

示例30-3中的MapDBEntry类定义了MapDB对象中的一个映射项。

示例30-3：MapDBEntry类

```
 1 import java.io.File;
 2 import java.util.Map;
 3 import org.mapdb.DB;
 4 import org.mapdb.DBMaker;
 5
 6 public class MapDBEntry {
 7     private DB db = null;
 8     private Map<String, String> map = null;
 9
10     public MapDBEntry(DB db, Map<String, String> map) {
11         this.db = db;
12         this.map = map;
13     }
14
15     public String getValue(String key) {
16         if (map == null) {
17             return null;
```

```
        }
        return map.get(key);
    }

    // 删除策略
    public void close() {
        closeDB();
        closeMap();
    }

    private void closeDB() {
        if (db != null) {
            db.close();
        }
    }

    private void closeMap() {
        if (map != null) {
            map = null;
        }
    }

    public static MapDBEntry create(String dbName) {
        DB db = DBMaker.newFileDB(new File(dbName))
                    .closeOnJvmShutdown()
                    .readOnly()
                    .make();
        Map<String, String> map = map = db.getTreeMap("collectionName");
        MapDBEntry entry = new MapDBEntry(db, map);
        return entry;
    }
}
```

# 使用MapDB

如何使用MapDB创建持久Map<K,V>？示例30-4显示了如何使用GenerateMapDB类为一个有序的分区文件创建一个MapDB数据库。

示例30-4：GenerateMapDB类

```
import org.mapdb.DB;
import org.mapdb.DBMaker;
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.File;
import java.util.concurrent.ConcurrentNavigableMap;

public class GenerateMapDB {

    public static void main(String[] args) throws Exception {
        String inputFileName = args[0];
        String mapdbName = args[1];
        create(inputFileName, mapdbName);
    }
```

```
15
16      public static void create(String inputFileName, String mapdbName)
17          throws Exception {
18          // 使用生成器模式配置和打开数据库。
19          // 代码自动完成特性会提供所有选项。
20          DB db = DBMaker.newFileDB(new File(mapdbName))
21                  .closeOnJvmShutdown()
22                  .make();
23
24          // 打开一个集合。TreeMap的性能优于HashMap。
25          ConcurrentNavigableMap<String,String> map =
26             db.getTreeMap("collectionName");
27
28          //
29          // line = <position><;><v><:><v>:...:<v>
30          // 其中<v> 有以下格式:
31          // <feature_id><,><mrna_feature_id><,><sequence_data_type_id><,><mapping>
32          //
32          String line = null;
33          BufferedReader reader = null;
34          try {
35              reader = new BufferedReader(new FileReader(inputFileName));
36              while ((line = reader.readLine()) != null) {
37                  line = line.trim();
38                  String[] tokens = line.split(";");
39                  if (tokens.length == 2) {
40                      map.put(tokens[0], tokens[1]);
41                  }
42                  else {
43                      System.out.println("error line="+line);
44                  }
45              }
46          }
47          finally {
48             reader.close(); // 关闭输入文件。
49             db.commit(); // 将所做的修改持久存储到磁盘。
50             db.close(); // 关闭数据库资源。
51          }
52      }
53  }
```

要查询这个MapDB数据库，可以编写一个非常简单的QueryMapDB类，如示例30-5所示。

示例30-5：QueryMapDB类

```
 1 import java.io.File;
 2 import java.util.Map;
 3 import org.mapdb.DB;
 4 import org.mapdb.DBMaker;
 5 import org.apache.log4j.Logger;
 6 /**
 7  * 这个类定义了MapDB上的一个基本查询。
 8  *
 9  * @author Mahmoud Parsian
10  *
```

```
 */
public class QueryMapDB {

   public static void main(String[] args) throws Exception {
       long beginTime = System.currentTimeMillis();
       String mapdbName = args[0];
       String position = args[1];
       THE_LOGGER.info("mapdbName="+mapdbName);
       THE_LOGGER.info("position="+position);
       String value = query(mapdbName, position);
       THE_LOGGER.info("value="+value);
       long elapsedTime = System.currentTimeMillis() -beginTime;
       THE_LOGGER.info("elapsedTime (in millis) ="+elapsedTime);
       System.exit(0);
   }

   public static String query(String mapdbName, String key) throws Exception {
       String value = null;
       DB db = null;
       try {
           db = DBMaker.newFileDB(new File(mapdbName))
               .closeOnJvmShutdown()
               .readOnly()
               .make();
           Map<String, String> map = map = db.getTreeMap("collectionName");
           value = map.get(key);
       }
       finally {
          if (db != null) {
              db.close();
          }
       }
       return value;
   }
}
```

## 测试MapDB: put()

下面的代码段显示了如何创建MapDB记录：

```
# javac GenerateMapDB.java
# javac QueryMapDB.java

# cat test.txt
19105201;35302633,35292056,2,1:20773813,35399339,2,1
19105202;35302633,35292056,2,1:20773813,35399339,2,1
19105203;35302633,35284930,2,1:35302633,35292056,2,1:20773813,35399339,2,1
19105204;35302633,35284930,2,1:35302633,35292056,2,1:20773813,35399339,2,1

# java GenerateMapDB test.txt mapdbtest
# ls -l mapdbtest*
-rw-r--r-- 1 mahmoud staff 32984 Feb 11 16:40 mapdbtest
-rw-r--r-- 1 mahmoud staff 13776 Feb 11 16:40 mapdbtest.p
```

从生成的输出可以看到，MapDB会为每个持久存储的散列表生成两个文件（*mapdbtest*和*mapdbtest.p*）。

## 测试MapDB: get()

下面的代码段显示了如何查询MapDB记录：

```
# java QueryMapDB mapdbtest 19105201
16:41:16 [QueryMapDB] - mapdbName=mapdbtest
16:41:16 [QueryMapDB] - position=19105201
16:41:16 [QueryMapDB] - value=35302633,35292056,2,1:20773813,35399339,2,1
16:41:16 [QueryMapDB] - elapsedTime (in millis) = 2

# java QueryMapDB mapdbtest 19105202
16:41:21 [QueryMapDB] - mapdbName=mapdbtest
16:41:21 [QueryMapDB] - position=19105202
16:41:21 [QueryMapDB] - value=35302633,35292056,2,1:20773813,35399339,2,1
16:41:21 [QueryMapDB] - elapsedTime (in millis) = 2

# java QueryMapDB mapdbtest 191052023333
16:41:55 [QueryMapDB] - mapdbName=mapdbtest
16:41:55 [QueryMapDB] - position=191052023333
16:41:55 [QueryMapDB] - value=null
16:41:55 [QueryMapDB] - elapsedTime (in millis) = 1
```

# 使用LRUMap的MapReduce解决方案

既然已经有一个高效的LRU映射缓存，我们可以在`map()`或`reduce()`函数中使用这个缓存。要在`map()` 或`reduce()`中使用LRUMap缓存，需要完成以下步骤（可以在映射器或归约器类中实现）：

1. 初始化并建立LRUMap缓存。这一步在`setup()`方法中完成。这个工作只完成一次。
2. 在`map()`或`reduce()`中使用LRUMap缓存。这可能会使用多次。
3. 关闭LRUMap缓存对象（释放所有不再需要的资源）。这个步骤可以在`cleanup()`方法中完成。这个工作只完成一次。

为了更方便地完成这些步骤，我们定义了一个服务类，名为`CacheManager`，这个类可以如下使用，如示例30-6所示。

示例30-6：CacheManager的使用

```
try {
    //
    // 初始化缓存
    //
    CacheManager.init();

    //
```

```
 8       // 使用缓存
 9       //
10       String chrID = ...;
11       String position = ...;
12       List<String> valueAsList = CacheManager.get(chrID, position);
13  }
14  finally {
15     //
16     // 关闭缓存
17     //
18     CacheManager.close();
19  }
```

## CacheManager定义

CacheManager的定义如示例30-7所示，这是一个服务类，提供了3个基本功能：打开缓存、服务（获取所需的值）以及关闭缓存。由于这是一个服务类，所有方法都定义为static。BeginEndPosition对象实现了分区数据结构，这样就能得到对应一个给定键（chrID, position）的数据库名（一个64 MB的散列表）。

示例30-7：CacheManager定义

```
 1 public class CacheManager {
 2
 3    private static final int DEFAULT_LRU_MAP_SIZE = 128;
 4    private static int theLRUMapSize = DEFAULT_LRU_MAP_SIZE;
 5
 6    private static CustomLRUMap<String, MapDBEntry<String, String>>
 7       theCustomLRUMap = null;
 8    private static BeginEndPosition beginend = null;
 9    private static String mapdbRootDirName =
10       "/cache/mapdb/pf";
11    private static String mapdbBeginEndDirName =
12       "/cache/mapdb/pf/begin_end_position";
13    private static boolean initialized = false;
14
15    public static void setLRUMapSize(int size) {
16       theLRUMapSize = size;
17    }
18
19    public static int getLRUMapSize() {
20       return theLRUMapSize;
21    }
22
23    // 初始化LRUMap
24    public static void init() throws Exception {...}
25
26    // 初始化LRUMap
27    public static void init(int size) throws Exception {...}
28
29    // 关闭LRUMap中的缓存数据库
30    public static void close() throws Exception {...}
31
```

```
32    // 从缓存数据库得到对应一个给定键(chrID, position)的值。
33    public static String get(int chrID, int position) throws Exception {...}
34
35    //从缓存数据库得到对应一个给定键(chrID, position)的值。
36    public static String get(String chrID, String position) throws Exception {...}
37
38    //关闭LRUMap中的缓存数据库
39    public static void close() throws Exception {...}
40 }
```

## 初始化缓存

示例30-8显示了CacheManager如何初始化LRUMap缓存。

示例30-8：CacheManager: *初始化缓存*

```
 1 public static void setLRUMapSize(int size) {
 2     theLRUMapSize = size;
 3 }
 4
 5 public static int getLRUMapSize() {
 6     return theLRUMapSize;
 7 }
 8
 9 // 初始化LRUMap
10 public static void init() throws Exception {
11     if (initialized) {
12         return;
13     }
14     theCustomLRUMap =
15        new CustomLRUMap<String, MapDBEntry<String, String>>(theLRUMapSize);
16     beginend = new BeginEndPosition(mapdbBeginEndDirName);
17     beginend.build(mapdbRootDirName);
18     initialized = true;
19 }
20
21 // 初始化LRUMap
22 public static void init(int size) throws Exception {
23     if (initialized) {
24         return;
25     }
26     setLRUMapSize(size);
27     init();
28 }
```

## 使用缓存

示例30-9显示了CacheManager如何访问LRUMap缓存来得到所需的值。

示例30-9：CacheManager的使用

```
/**
 * 从LRUMap缓存得到对应一个给定键(chrID, position)的值。
 * @param chrID=chrID
 * @param position=position
 */
public static String get(int chrID, int position) throws Exception {
    return get(String.valueOf(chrID), String.valueOf(position));
}

/**
 * 从缓存数据库得到对应一个给定键(chrID, position)的值
 * （值为snpID）。
 * @param chrID=chrID
 * @param position=position
 */
public static String get(String chrID, String position) throws Exception {
    String dbName = getDBName(chrID, position);
    if (dbName == null) {
        return null;
    }
    // 现在返回缓存值
    MapDBEntry<String, String< entry = theCustomLRUMap.get(dbName);
    if (entry == null) {
        entry = MapDBEntryFactory.create(dbName);
        theCustomLRUMap.put(dbName, entry);
    }
    return entry.getValue(position);
}

private static String getDBName(String chrID, String position) {
    // 查询参数为： chrID和position
    List<Interval> results = beginend.query(chrID, position);
    if ((results == null) || (results.isEmpty()) || (results.size() == 0)) {
        return null;
    }
    else {
       return results.get(0).db();
    }
}
```

# 关闭缓存

示例30-10显示了CacheManager如何关闭LRUMap缓存。

示例30-10：CacheManager: 关闭缓存

```
public static void close() throws Exception {
    if (theCustomLRUMap != null) {
        for (Map.Entry<String, MapDBEntry<String, String>>
           entry : theCustomLRUMap.entrySet()) {
           entry.getValue().close();
        }
```

```
7     }
8 }
```

这一章提供了一个可伸缩的超大缓存解决方案，可以在`map()`和`reduce()`函数中使用。我们的解决方案非常简单，只使用商用服务器就可以实现。第31章将提供Bloom过滤器的简单介绍，并展示在MapReduce范式中如何使用Bloom过滤器。

第31章

# Bloom过滤器

这一章将介绍Bloom过滤器的概念，并在归约端连接中使用，可以在MapReduce作业的映射阶段使用Bloom过滤器。那么什么是Bloom过滤器呢？如何在MapReduce环境中使用？它对于加快两个庞大关系/表之间的连接操作有什么帮助？根据Wikipedia（*http://en.wikipedia.org/wiki/Bloom_filter*）的描述：

> Bloom过滤器是一个空间高效的概率型数据结构，由Burton Howard Bloom在1970年提出，用来检验一个元素是否属于一个集合。有可能会得到假阳性匹配判断，不过不会得到假阴性错误……。换句话说，查询会返回“可能在集合中”或“肯定不在集合中”。元素可以增加到集合中，但不能删除（不过这个问题可以利用一个“计数”过滤器解决）。增加到集合的元素越多，假阳性的概率就越大。

Bloom过滤器数据结构可能会对某些实际上并不是集合成员的元素返回true（这称为假阳性错误（false-positive error）），不过对于确实在集合中的元素，绝不会返回false；对于集合中的各个元素，Bloom过滤器必然返回true。关于Bloom过滤器，Bill Mill提供了一个非常棒的教程（*http://billmill.org/bloomfilter-tutorial/*）。如果你希望更深入地研究这个数据结构，Jacob Honoroff还写过一篇关于Bloom过滤器数据结构的很好的介绍文章（*http://bit.ly/bloom_filter_intro*）。

## Bloom过滤器性质

Bloom过滤器的性质可以总结如下：

- 给定一个大集合$S = \{x_1, x_2, ..., x_n\}$，Bloom过滤器是一个快速而且空间高效的概率型缓存生成器；基本说来，它近似于集合成员操作：

$x$是否属于$S$（$x \in S$）？

- Bloom过滤器会尝试回答查找问题：项x是否在集合S中？
- Bloom过滤器可能存在假阳性错误（false positive errors），因为这只会带来一个额外的数据集访问操作，而不会导致错误答案。这说明，对于不在集合中的某个$x$，Bloom过滤器可能会指示这个$x$在这个集合中。
- Bloom过滤器不会有假阴性错误（false negative errors），因为这会导致错误的答案。这意味着，如果$x$在集合中，Bloom过滤器必然会指出这个$x$确实在集合中。

存在两种可能的错误：

- 假阳性：$x \notin S$, 但答案是$x \in S$
- 假阴性：$x \in S$, 但答案是$x \notin S$

下面来重点考虑两个关系/表之间的一个简单的连接示例。假设我们希望根据一个公共字段$a$连接$R(a, b)$和$S(a, c)$。进一步假设：

$size(R)$ = 1000000000 (较大的数据集)
$size(S)$ = 10000000 (较小的数据集)

要完成一个基本连接，需要检查10000000000000000个记录，这是一个庞大而且很耗费时间的任务。为了减少$R$和$S$之间连接操作的时间和复杂性，一种想法是在关系$S$（较小的数据集）上使用Bloom过滤器，然后在关系$R$上使用生成的Bloom过滤器数据结构。这样可以消除$R$中不必要的记录（可能可以把集合缩小到20000000），从而快速而高效地实现连接。

接下来给出Bloom过滤器数据结构的半形式化描述来回答下面这些问题：Bloom过滤器概率数据结构包括什么？如何构造一个Bloom过滤器？什么是假阳性错误的概率？如何降低假阳性错误的概率？具体如下，给定一个大集合$S = \{x_1, x_2, ..., x_n\}$：

- 令B是一个大小为$m(m>1)$的位数组，其中的元素都初始化为0。$B$的元素为$B[0]$, $B[1]$, $B[2]$, ..., $B[m-1]$。存储数组B所需的内存只是存储整个集合$S$所需内存的很小一部分。通过选择更大的位向量（数组$B$），可以减少假阳性错误的概率。
- 令$\{H_1, H_2, ..., H_k\}$是包含k个散列函数的集合。如果$H_i(x_j) = a$, 则设置$B[a] = 1$。可以使用SHA1、MD5和Murmur作为散列函数。例如，可以使用以下散列函数：
  - $H_i(x) = MD5(x + i)$
  - $H_i(x) = MD5(x||i)$
- 要检查$x$是否属于$S$，可以根据$H_i(x)$检查$B$。所有$k$值必须为1。
- 有可能存在假阳性错误；所有k值都为1，但是$x$不在$S$中。

- 假阳性错误的概率为：

$$\left(1-\left[1-\frac{1}{m}\right]^{kn}\right)^{k} \approx \left(1-e^{-kn/m}\right)^{k}$$

- 最优的散列函数是什么？最好有多少个散列函数（散列函数最佳数目）？对于一个给定的$m$（为Bloom过滤器选择的位数）和n（大数据集的大小），保证假阳性错误概率最小的k值（散列函数数目）为（*ln* 表示“自然对数”[注1]）：

$$k=\frac{m}{n}\ln(2)$$

其中：$m=\frac{n\ln(p)}{(\ln(2))^2}$

因此，一个特定位翻转为1的概率为：

$$1-\left(1-\frac{1}{m}\right)^{kn} \approx 1-e^{-\frac{kn}{m}}$$

根据Wikipedia（*http://bit.ly/bloom_filter_article*）的描述：

> 不同于标准散列表，固定大小的Bloom过滤器可以表示包含任意多个元素的集合，增加一个元素绝不会导致这个数据结构“上溢”。不过，随着元素的增加，假阳性率会不断增加，直到过滤器中的所有位都设置为1，此时所有查询都会得到一个阳性结果。

# 一个简单的Bloom过滤器示例

在这个例子中，你会了解如何插入和查询一个大小为10（$m$=10）的Bloom过滤器，这里使用了3个散列函数$H = \{H_1, H_2, H_3\}$。令$H(x)$指示这3个散列函数的结果($H(x) = \{H_1(x),H_2(x),H_3(x)\}$)。首先将一个长度为10位的数组$B$初始化为0：

```
Array B:
   initialized:
         index 0 1 2 3 4 5 6 7 8 9
         value 0 0 0 0 0 0 0 0 0 0

   insert element a, H(a) = (2, 5, 6)
         index 0 1 2 3 4 5 6 7 8 9
         value 0 0 1 0 0 1 1 0 0 0

   insert element b, H(b) = (1, 5, 8)
         index 0 1 2 3 4 5 6 7 8 9
         value 0 1 1 0 0 1 1 0 1 0
```

注1：自然对数是以$e$为底的对数，其中 e是欧拉数: 2.71828183。令$x = e^y$; 则$ln(x) = \log_e(x) = y$。

```
query element c
H(c) = (5, 8, 9) => c is not a member (since B[9]=0)

query element d
H(d) = (2, 5, 8) => d is a member (False positive)

query element e
H(e) = (1, 2, 6) => e is a member (False positive)

query element f
H(f) = (2, 5, 6) => f is a member (Positive)
```

# Guava 库中的Bloom过滤器

Guava库（*http://bit.ly/guava-libraries*）提供了一个Bloom过滤器实现。它的工作如下。首先，定义将在Bloom过滤器中使用的基本类型（在这里是Person）（这个例子选自*http://bit.ly/hashingexplained*）：

```
class Person {
   final int id;
   final String firstName;
   final String lastName;
   final int birthYear;
   Person(int id, String firstName, String lastName, int birthYear) {
      this.id = id;
      this.firstName = firstName;
      this.lastName = lastName;
      this.birthYear = birthYear;
   }
}
```

然后定义这个基本类型的一个Funnel，我们希望在这个Funnel上使用Bloom过滤器。Funnel描述了如何将特定的对象类型分解为基本字段值。这里的Funnel如下所示：

```
Funnel<Person> personFunnel = new Funnel<Person>() {
   @Override
   public void funnel(Person person, PrimitiveSink into) {
      into
       .putInt(person.id)
       .putString(person.firstName, Charsets.UTF_8)
       .putString(person.lastName, Charsets.UTF_8)
       .putInt(birthYear);
   }
};
```

一旦定义了Funnel<T>，下面可以定义和使用Bloom过滤器：

```
List<Person> friendsList = {Person1, Person2, ...};
BloomFilter<Person> friends = BloomFilter.create(personFunnel, 500, 0.01);
for(Person friend : friendsList) {
   friends.put(friend);
}
```

```
// 后面会使用Bloom过滤器
Person dude = new Person(100, "Alex", "Smith", 1967);
if (friends.mightContain(dude)) {
   // 如果这个dude不是一个好友，到达这个位置的概率
   // 为1% (0.01)；例如，
   // 可能在完成一个开销很高的完全检查的同时
   // 异步加载
}
```

Guava库提供了以下散列函数来建立Bloom过滤器数据结构：

- `md5()`
- `murmur3_128()`
- `murmur3_32()`
- `sha1()`
- `sha256()`
- `sha512()`
- `goodFastHash(int bits)`

# MapReduce中使用Bloom过滤器

Bloom过滤器是一个紧凑、快速的小数据结构，可以用来确定集合成员关系。我们已经了解到，可以在两个关系/表的连接中使用Bloom过滤器，如$R(a, b)$和$S(a, c)$，其中一个关系的记录数目庞大（例如，R可能有1000000000个记录），而另一个关系的记录数较少（例如，$S$可能有10000000个记录）。$R$和$S$在字段$a$上完成连接需要耗费很长时间，效率相当低。可以如下使用Bloom过滤器数据结构：根据关系$S(a, c)$建立一个Bloom过滤器，然后使用所建立的这个Bloom过滤器检验值$R(a, b)$来确定成员关系。需要说明，对于归约端连接优化，我们会在映射任务中使用Bloom过滤器，这会减少MapReduce作业的I/O开销。示例31-1显示了使用Bloom过滤器优化的归约端连接。这个方法包括两个步骤：

1. 构建Bloom过滤器。这是一个MapReduce作业，使用两个关系/表中较小的一个来构建Bloom过滤器数据结构。在这一步中，只需要映射器（将构建多个Bloom过滤器，对应每个映射器分别建立一个Bloom过滤器）。
2. 在归约端连接中使用步骤1中建立的Bloom过滤器数据结构。

示例31-1：连接R和S

```
1 joinResult = {};
2 for all tuples r in R {
3   for all tuples s in S {
4     if (joinConditionSatisfied(r, s)) {
5       // 连接条件类似于r.a == s.b
```

```
6         add (r;s) to the joinResult;
7       }
8     }
9 }
```

这一章介绍了Bloom过滤器数据结构的基本概念，并展示了如何在MapReduce环境中使用Bloom过滤器完成优化。在生产集群中部署Bloom过滤器时需要仔细测试。这是我们讨论的最后一个优化技术。附录A和附录B会提供关于biosets和Spark RDD概念的一些支持材料。

# 附录A

# Bioset

Bioset也称为基因标签（gene signature）[注1]或assays[注2]，包括试验样本比较形式的数据（对应转录组、表观遗传和拷贝数变异数据），以及基因型标签（对应全基因组关联研究[GWAS]和突变数据）。

Bioset有一个关联的数据类型，这可以是基因表达式、蛋白质表达式、甲基化、拷贝数变异、miRNA或体细胞突变。另外每个bioset项/记录有一个关联的参考类型，这可以是r1=normal（正常）、r2=disease（患病）、r3=paired（成对）或r4=unknown（未知）。需要说明，参考类型不适用体细胞突变数据类型。

每个bioset的项/记录数取决于其数据类型（见表A-1）。

注1：基因标签是细胞中的一组基因，其组合表达式模式可以唯一标识一个生物学表型或身体状况的特征。理论上讲，表型由一个基因表达式标签定义，这可以是用来区别一种疾病不同亚型的基因表达式、预测患病个体生存或预后率的基因表达式，或者是预测某种特定治疗方案有效率的基因表达式。理想情况下，基因签名可以用来选择某种特定治疗方案对其有效的一组患者（资料来源：*http://en.wikipedia.org/wiki/Gene_signature*）。

注2：assay是实验室医学、药理学、环境生物学、连续输送药物和分子生物学中使用的一个研究（分析）过程，用来定性地评估或定量地测量一个目标实体（分析对象）的存在性、数量或机能活动。分析对象可以是一种药物、生化物质或某个有机体或有机物样本中的一个细胞（资料来源：*http://en.wikipedia.org/wiki/Assay*）。

表A-1：每个bioset数据类型的记录数

| Bioset数据类型 | 项/记录数 |
| --- | --- |
| 体细胞突变 | 3000～20000 |
| 甲基化 | 30000 |
| 基因表达式 | 50000 |
| 拷贝数变异 | 40000 |
| 生殖细胞 | 4300000 |
| 蛋白质表达式 | 30000 |
| miRNA | 30000 |

# 附录B

# Spark RDD

Apache Spark是一个“快速而通用的集群计算系统”，其主要抽象是一个分布式项集合（如日志记录、FASTQ序列或员工记录），这称为一个弹性分布式数据集（resilient distributed data set，RDD）。我们可以从Hadoop InputFormat（如HDFS文件）通过转换其他RDD或者通过转换“集合”数据结构（如List和Map）来创建RDD，还可以由Java/Scala集合对象以及其他持久数据存储库创建RDD。RDD的主要作用是通过一个简单API（如JavaRDD和JavaPairRDD）支持更高层的数据并行操作。

这个附录将通过一些简单的Java示例介绍Spark RDD。这里的目的不是深入研究RDD的体系结构[35]细节，而只是向你展示如何在MapReduce或通用程序中使用RDD（如有向无环图或DAG）。可以把RDD看作是一个类型T的项集合的一个句柄，这些项是某个计算的结果。类型T可以是任何标准Java数据类型（如String，Integer，Map或List）或者任何定制对象（如Employee或Mutation）。Spark RDD可以完成动作（如reduce()，collect()，count()和saveAsTextFile()）和变换（如map()，filter()，union()，groupByKey()和reduceByKey()），这些可以用来完成更复杂的计算。这里给出的所有示例都基于Spark-1.1.0。

一般来讲，与相应的MapReduce/Hadoop程序相比，如果为集群节点提供更多RAM，Spark程序可以更快地运行。Spark提供了StorageLevel类，其中包括一些用来控制RDD存储的标志，包括：

- MEMORY_ONLY（只使用内存来存储RDD）。
- DISK_ONLY（只使用硬盘存储RDD）。
- MEMORY_AND_DISK（组合使用内存和硬盘存储RDD）。

要想详细地了解Spark和RDD，可以参考《Learning Spark》[12]。

# Spark操作

Spark作业完成对RDD的工作（动作和变换）。需要说明，Spark的主程序在Spark驱动器上执行，而变换在Spark工作节点上执行。在Spark中创建一个RDD对象后，可以通过调用Spark API完成一组操作。Spark主要有以下两类操作：

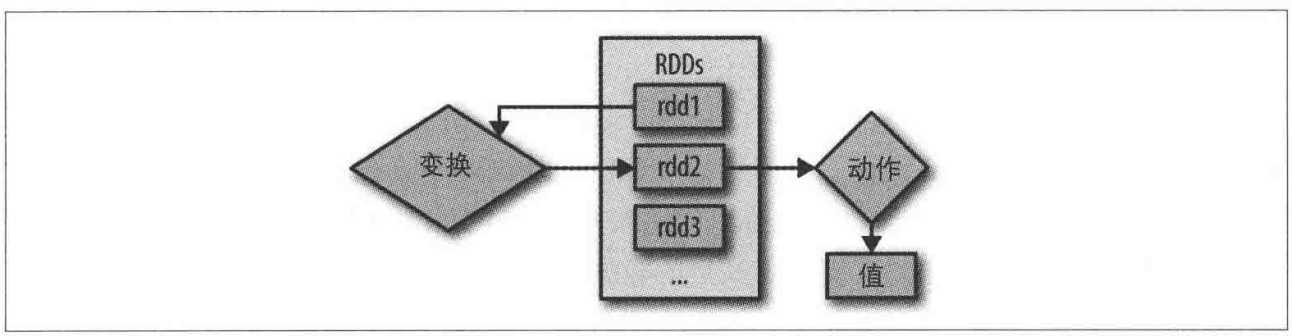
*变换（Transformations）*

这类操作根据原来的RDD返回一个新RDD。可以通过以下Spark API（包括`map()`, `filter()`, `sample()`和`union()`）完成变换。

*动作（Actions）*

这一类操作根据RDD上完成的某个计算返回一个值。可以通过以下Spark API（包括`reduce()`,`count()`, `first()`和`foreach()`）完成动作。

变换和动作如图B-1所示。




图B-1：Spark操作：变换和动作

# Tuple<N>

Spark大量使用了元组，如`scala.Tuple2`（两元素元组)和`scala.Tuple3`（三元素元组）。`scala.Tuple<N>`表示N个元素的一个组合数据结构。Spark API使用了组合数据类型`scala.Tuple2`（表示两元素元组，参见示例B-1）和`scala.Tuple3`（表示三元素元组，参见示例B-2）。

示例B-1：Tuple2

```
import scala.Tuple2;
import java.util.List;
import java.util.Arrays;
...
Tuple2<String,Integer> t21 = new Tuple2<String,Integer>("abc", 234);
String str21 = t21._1; // str21 包含 "abc"
Integer int21 = t21._2; // int21 包含 234

List<Integer> listofint = Arrays.asList(1, 2, 3);
Tuple2<String,List<Integer>> t22 =
```

```
11    new Tuple2<String,List<Integer>>("z1z2", listofint);
12 String str22 = t22._1;          // str22 包含 "z1z2"
13 List<Integer> list22 = t22._2; // int21 指向List(1, 2, 3)
```

示例B-2：Tuple3

```
1 import scala.Tuple3;
2 import java.util.List;
3 import java.util.Arrays;
4 ...
5 List<String> listofstrings = Arrays.asList("a1", "a2", "a3");
6 Tuple3<String,Integer, List<String>> t3 =
7    new Tuple3<String,Integer, List<String>>("a1a2", 567, listofstrings);
8 String str3 = t3._1;          // str3 包含 "a1a2"
9 Integer int3 = t3._2;         // int3 包含 567
10 List<String> list3 = t3._3; // list3 指向List("a1", "a2", "a3")
```

注意，Java Spark API中还定义了scala.Tuple4到scala.Tuple22。

# RDD

我们已经知道，RDD表示弹性分布式数据集，这是Spark中的基本数据抽象。Spark提供了很多子类和包装器（如JavaRDD、JavaPairRDD和JavaHadoopRDD）。要完成mapToPair()、reduceByKey()或groupByKey()操作（称为变换），可以使用一个RDD作为输入和输出。根据定义，RDD表示一个不可变的[注1]分区的元素集合，可以并行处理。这说明，如果有一个JavaRDD<String>，其中包含1亿个元素，可以把它划分为10、100或1000个区段，每个区段可以独立地处理。选择适当的分区数很重要，这取决于你的集群大小、每个服务器可用的内核数以及集群中可用的RAM总量。要得出最合适的分区数，并没有百试百灵的灵丹妙药，不过可以通过反复尝试来适当地设置。

JavaRDD和JavaPairRDD在org.apache.spark.api.java包中定义，如示例B-3所示。

示例B-3：JavaRDD和JavaPairRDD

```
1   public class JavaRDD<T> ...
2   // 表示类型为T的元素
3   //   JavaRDD<T> 的例子包括：
4   //   JavaRDD<String>
5   //   JavaRDD<Integer>
6   //   JavaRDD<Map<String, Long>>
7   //   JavaRDD<Employee>
8   //   JavaRDD<Tuple2<String, Integer>>
9   //   ...
10
11   public class JavaPairRDD<K,V> ...
12   // 表示类型K和V的(key,value)对
13   //   JavaPairRDD<K,V>的例子包括：
```

注1：RDD是只读的，不能修改或更新。

```
14  //   JavaPairRDD<String,Integer>
15  //   JavaPairRDD<String, Tuple2<Integer,Integer>>
16  //   JavaPairRDD<Integer, Map<String, Long>>
17  //   JavaPairRDD<Tuple2<Integer,Integer>, Tuple2<String,Double>>
18  //   ...
```

## 如何创建RDD

对于大多数MapReduce应用，RDD由一个JavaSparkContext对象创建，或者由JavaRDD和JavaPairRDD创建。最初的RDD从JavaContextObject创建，后续的RDD可以由很多其他的类创建（包括JavaRDD和JavaPairRDD）。对于大多数应用，创建和处理RDD的顺序如下：

1. 创建一个JavaSparkContext。完成这一步有很多方法，例如，可以使用Spark主URL创建，或者使用YARN的资源管理器主机名来创建。还可以使用SparkUtil类（*http://bit.ly/sparkutil*）来创建。也可以使用以下代码来创建：

```
1 import org.apache.spark.SparkConf;
2 import org.apache.spark.api.java.JavaSparkContext;
3 ...
4 // 创建JavaSparkContext
5 SparkConf sparkConf = new SparkConf().setAppName("my-app-name");
6 JavaSparkContext ctx = new JavaSparkContext(sparkConf);
```

2. 一旦有了JavaSparkContext的一个实例，可以由很多不同来源创建RDD：Java集合对象、文本文件及HDFS文件（文本或二进制）。

3. 一旦有了JavaRDD或JavaPairRDD的一个实例，可以通过使用map()、reduceByKey()、filter()或groupByKey()方法创建新的RDD。还可以使用Java集合对象或者通过读取文件（文本文件或文本/二进制HDFS文件）来创建RDD。

## 使用集合对象创建RDD

在上一节中提到，可以使用Java集合对象创建RDD。示例B-4展示了由一个java.utl.List对象创建一个RDD（名为rdd1）。

示例B-4：JavaRDD创建

```
1 import java.util.List;
2 import com.google.common.collect.Lists;
3 import org.apache.spark.SparkConf;
4 import org.apache.spark.api.java.JavaSparkContext;
5 ...
6 // 创建JavaSparkContext
7 SparkConf sparkConf = new SparkConf().setAppName("myapp");
8 JavaSparkContext ctx = new JavaSparkContext(sparkConf);
9
10 // 创建所需的集合对象
```

```
final List<String> list1 = Lists.newArrayList(
    "url1,2",
    "url1,3",
    "url2,7",
    "url2,6",
    "url3,4",
    "url3,5",
    "url3,6"
);

// 从Java集合对象创建一个RDD
JavaRDD<String> rdd1 = ctx.parallelize(list1);
```

现在rdd1是一个JavaRDD<String>，其中包含7个元素（分别为<url><,><count>格式）。一旦创建了RDD，接下来可以应用函数和变换（如map()、collect()、reduceByKey()和很多其他方法）。

## 收集RDD元素

生成的rdd1可以收集为一个List<String>，如示例B-5所示。

示例B-5：收集RDD的元素

```
// 从Java集合对象创建RDD
JavaRDD<String> rdd1 = ctx.parallelize(list1);

// 收集RDD的元素：
// java.util.List<T> collect()
// 返回一个数组，其中包含这个RDD中的所有元素。
java.util.List<String> collected = rdd1.collect();
```

一般地，collect()可以用于调试（查看一个RDD的内容）或者保存为最终结果。将Spark程序部署到生产集群时，一定要删除所有用于调试的collect()（collect()操作会花很长时间收集RDD元素，在生产阶段这会严重影响性能）。注意，调用collect()会导致调度运行目前的有向无环图（DAG），性能会大幅下降。如果在一个很大的RDD上调用collect()，可能会触发一个OutOfMemoryError错误。另外还要注意，collect()会把所有分区发送到一个驱动器（这可能是产生OOM错误的主要原因）。建议不要在很大的RDD上调用collect()。

## 将已有的RDD转换为一个新RDD

以rdd1为例，把它转换为键-值对（作为一个JavaPairRDD对象）。对于这个变换，我们可以使用JavaRDD.mapToPair()方法，如示例B-6所示。

示例B-6：JavaPairRDD变换

```
// 从Java集合对象创建一个RDD
JavaRDD<String> rdd1 = ctx.parallelize(list1);
```

```
3
4 // 创建 (K,V)对，其中K是一个URL，V是一个计数
5 JavaPairRDD<String, Integer> kV1 = rdd1.mapToPair(
6       new PairFunction<
7                       String, // T 作为输入
8                       String, // K 作为输出
9                       Integer // V 作为输出
10                     >() {
11      @Override
12      public Tuple2<String, Integer> call(String element) {
13         String[] tokens = element.split(",");
14         // tokens[0] = URL
15         // tokens[1] = count
16         return new Tuple2<String, Integer>(tokens[0], new Integer(tokens[1]));
17      }
18 });
```

## 读取文件创建RDD

对于大多数MapReduce应用，可以从HDFS读取数据再创建RDD，还可以将结果保存到HDFS中。下面假设我们有两个HDFS文件，如下所示：

```
$ hadoop fs -ls /myinput/logs/
... /myinput/logs/file1.txt
... /myinput/logs/file2.txt

$ hadoop fs -cat /myinput/logs/file1.txt
url5,2
url5,3
url6,7
url6,8
$ hadoop fs -cat /myinput/logs/file2.txt
url7,1
url7,2
url8,5
url8,6
url9,7
url9,8
url5,9
$
```

接下来从这两个文件创建一个RDD，如示例B-7所示。

示例B-7：从HDFS文件创建RDD

```
1 // 创建JavaSparkContext
2 SparkConf sparkConf = new SparkConf().setAppName("myapp");
3 JavaSparkContext ctx = new JavaSparkContext(sparkConf);
4
5 // 从HDFS读取输入文件
6 // 输入记录格式: <string-key><,><integer-value>
7 String hdfsPath = "/myinput/logs";
8 // public JavaRDD<String> textFile(String path)
```

```
// 从HDFS、本地文件系统(所有节点)、
//或Hadoop支持的任何
// 文件系统URI读取一个文本文件，作为一个String RDD返回。
JavaRDD<String> rdd2 = ctx.textFile(hdfsPath);
```

此时，rdd2包含11个String元素（其中4个来自*file1.txt*，7个来自*file2.txt*）。现在可以使用上一节的技术来创建一个JavaPairRDD<String,Integer>，其中键是一个URL，值是这个URL相关的计数。

## 按键分组

在示例B-8中，我们对rdd2按键完成元素分组。

示例B-8：对rdd2按键完成元素分组

```
// rdd2已经创建（见上一节）
// 创建(K,V)对，其中K是一个URL，V是一个计数
JavaPairRDD<String, Integer> kV2 = rdd2.mapToPair(
      new PairFunction<
                        String, // T 作为输入
                        String, // K 作为输出
                        Integer // V 作为输出
                       >() {
    @Override
    public Tuple2<String, Integer> call(String element) {
       String[] tokens = element.split(",");
       // tokens[0] = URL
       // tokens[1] = count
       return new Tuple2<String, Integer>(tokens[0], new Integer(tokens[1]));
    }
});

// 接下来按键对RDD的元素分组
JavaPairRDD<String, Iterable<Integer>> grouped2 = kV2.groupByKey();
```

得到的RDD（group2）将有以下内容，见表B-1。

表B-1：groupByKey()方法生成的键-值对

| 键 | 值 |
|---|---|
| url5 | [2, 3, 9] |
| url6 | [7, 8] |
| url7 | [1, 2] |
| url8 | [5, 6] |
| url9 | [7, 8] |

## 映射值

如果有一个JavaPairRDD，希望改变它的值来创建一个新的值集（而不改变键），为此可以使用JavaPairRDD.mapValues()方法。在示例B-9中，仍以上一节创建的grouped2 RDD为例，将返回最大URL计数。

示例B-9：JavaPairRDD.mapValues()

```
// public <U> JavaPairRDD<K,U> mapValues(Function<V,U> f)
// 通过一个映射函数传递键-值对RDD中的各个值
// 而不改变键；仍然保持
// 原来的RDD分区。
JavaPairRDD<String,Integer> mapped2 = grouped2.mapValues(
    new Function<
                 Iterable<Integer>, // 输入
                 Integer            // 输出
                >() {
    public Integer call(Iterable<Integer> list) {
       Integer max = null;
       for (Integer element : list) {
          if (max == null) {
              max = element;
       }
       else {
         if (element > max) {
             max = element;
         }
       }
     }
     return max;
    }
});
```

得到的RDD（mapped2）有以下内容，如表B-2所示。

表B-2：mapValues()方法生成的键-值对

| 键 | 值 |
|---|---|
| url5 | 9 |
| url6 | 8 |
| url7 | 2 |
| url8 | 6 |
| url9 | 8 |

## 按键归约

在示例B-10中，将按键对rdd2的元素完成归约：我们的目标是累加相同URL的计数（也就是说，创建各个URL的最终计数）。这里将使用JavaPairRDD.reduceByKey()方法。

示例B-10：JavaPairRDD.mapToPair()和reduceByKey()

```
// rdd2已经创建（见上一节）
// 创建 (K,V) 对，其中K是一个URL ，V是一个计数
JavaPairRDD<String, Integer> kV2 = rdd2.mapToPair(
      new PairFunction<
                       String, // T 作为输入
                       String, // K 作为输出
                       Integer // V 作为输出
                      >() {
    @Override
    public Tuple2<String, Integer> call(String element) {
       String[] tokens = element.split(",");
       // tokens[0] = URL
       // tokens[1] = count
       return new Tuple2<String, Integer>(tokens[0], new Integer(tokens[1]));
    }
});

// 接下来按键归约JavaPairRDD的元素
// public JavaPairRDD<K,V> reduceByKey(Function2<V,V,V> func)
// 使用一个可结合的归约函数合并各个键对应的值。
// 在把结果发送到一个归约器之前，
// 这还会在各个映射器上完成本地合并，
// 类似于MapReduce中的组合器。输出将根据现有的分区器/并行化等级
// 完成散列分区。
JavaPairRDD<String, Integer> counts = kV2.reduceByKey(
    new Function2<
                  Integer, // 输入 T1
                  Integer, // 输入 T2
                  Integer // 输出 T
                 >() {
    @Override
    public Integer call(Integer count1, Integer count2) {
      return count1 + count2;
    }
});
```

需要说明，`reduceByKey()`作用于两个类型为V的输入值，并生成相同类型的输出（也就是说，输出类型仍是V）。如果想创建类型不为V的输出，可以使用`combineByKey()`变换。得到的RDD (counts) 将包含以下内容，如表B-3所示。

表B-3：reduceByKey()生成的键-值对

| 键 | 值 |
|---|---|
| url5 | 14 |
| url6 | 15 |
| url7 | 3 |
| url8 | 11 |
| url9 | 15 |

## 按键组合

Spark提供了一个强大的通用函数，可以使用一组定制的聚合函数组合对应各个键的元素。这里将按键组合JavaPairRDD(kV2)的元素：我们的目标是得到一个唯一的列表，其中包含相同URL的全部计数（删除重复的计数）。要完成这个工作，我们将使用JavaPairRDD.combineByKey()方法。不能用reduceByKey()方法来解决这个问题，因为输出类型不同于输入类型。对于这个例子，输入和输出如下：

输入：(K: String, V: Integer)
输出：(K: String, V: Set<Integer>)

最简形式的combineByKey()的函数签名为：

```
public <C> JavaPairRDD<K,C> combineByKey(
                                          Function<V,C> createCombiner,
                                          Function2<C,V,C> mergeValue,
                                          Function2<C,C,C> mergeCombiners
                                         )
// 这是一个通用函数，可以使用一组定制的聚合函数组合对应各个键的元素。
// 将一个JavaPairRDD[(K, V)] 转换为一个
// 类型为JavaPairRDD[(K, C)]的结果（对应一个"组合类型" C）。需要说明V和C
// 可能不同—例如，可能将一个类型为(Int, Int)的RDD组合为
// 类型为(Int, List[Int])的RDD。用户提供了3个函数：
// - createCombiner，将V转换为一个C（例如，创建一个单元素列表）
// - mergeValue，将V合并为一个C（例如，把它增加到一个列表的末尾）
// - mergeCombiners，将2个C组合到一个C。
```

因此，要使用combineByKey()，我们必须提供/实现3个基本函数，如示例B-11所示。

示例B-11：combineByKey()的聚合函数

```
Function<Integer, Set<Integer>> createCombiner =
  new Function<Integer, Set<Integer>>() {
  @Override
  public Set<Integer> call(Integer x) {
     Set<Integer> set = new HashSet<Integer>();
     set.add(x);
     return set;
  }
};
Function2<Set<Integer>, Integer, Set<Integer>> mergeValue =
  new Function2<Set<Integer>, Integer, Set<Integer>>() {
  @Override
  public Set<Integer> call(Set<Integer> set, Integer x) {
     set.add(x);
     return set;
  }
};
Function2<Set<Integer>, Set<Integer>, Set<Integer>> combine =
  new Function2<Set<Integer>, Set<Integer>, Set<Integer>>() {
  @Override
```

```
21   public Set<Integer> call(Set<Integer> a, Set<Integer> b) {
22      a.addAll(b);
23      return a;
24   }
25 };
```

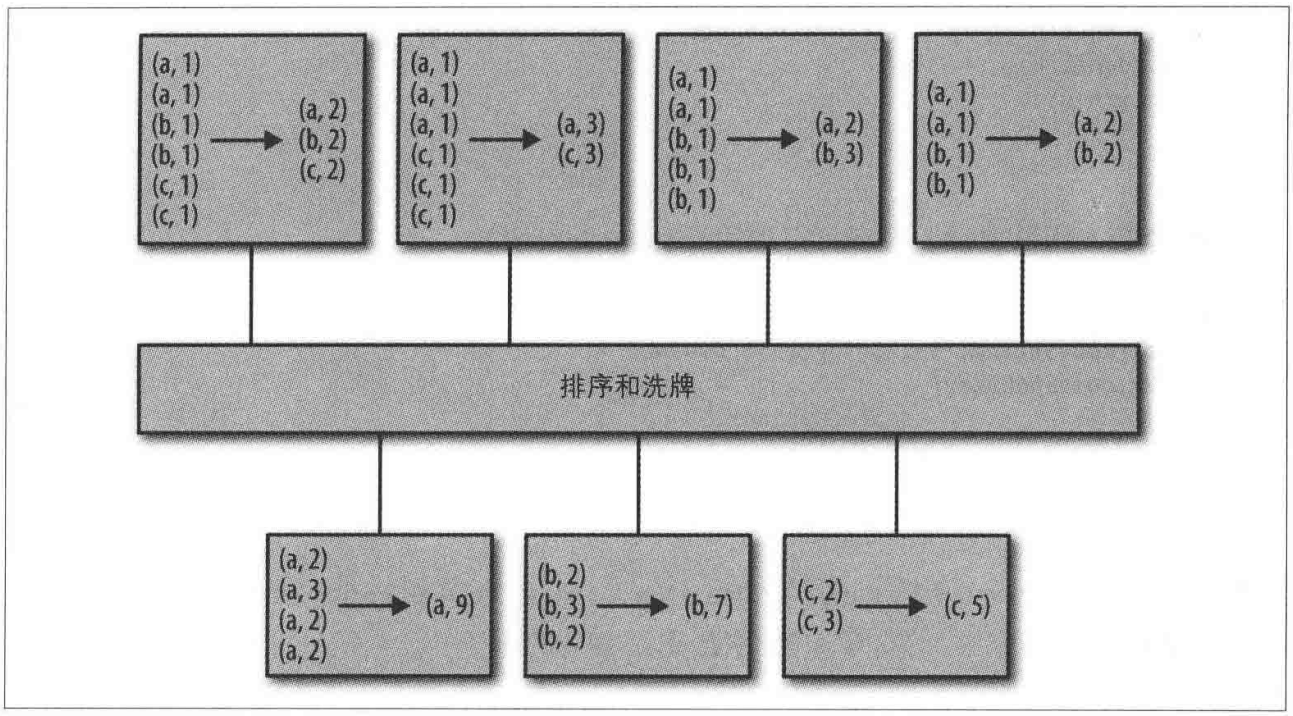
实现这3个基本函数之后，现在可以使用combineByKey()：

```
JavaPairRDD<String, Integer> kV2 = ...;
JavaPairRDD<String, Set<Integer>> combined =
    kV2.combineByKey(
                      createCombiner,
                      mergeValue,
                      combine
                     );
```

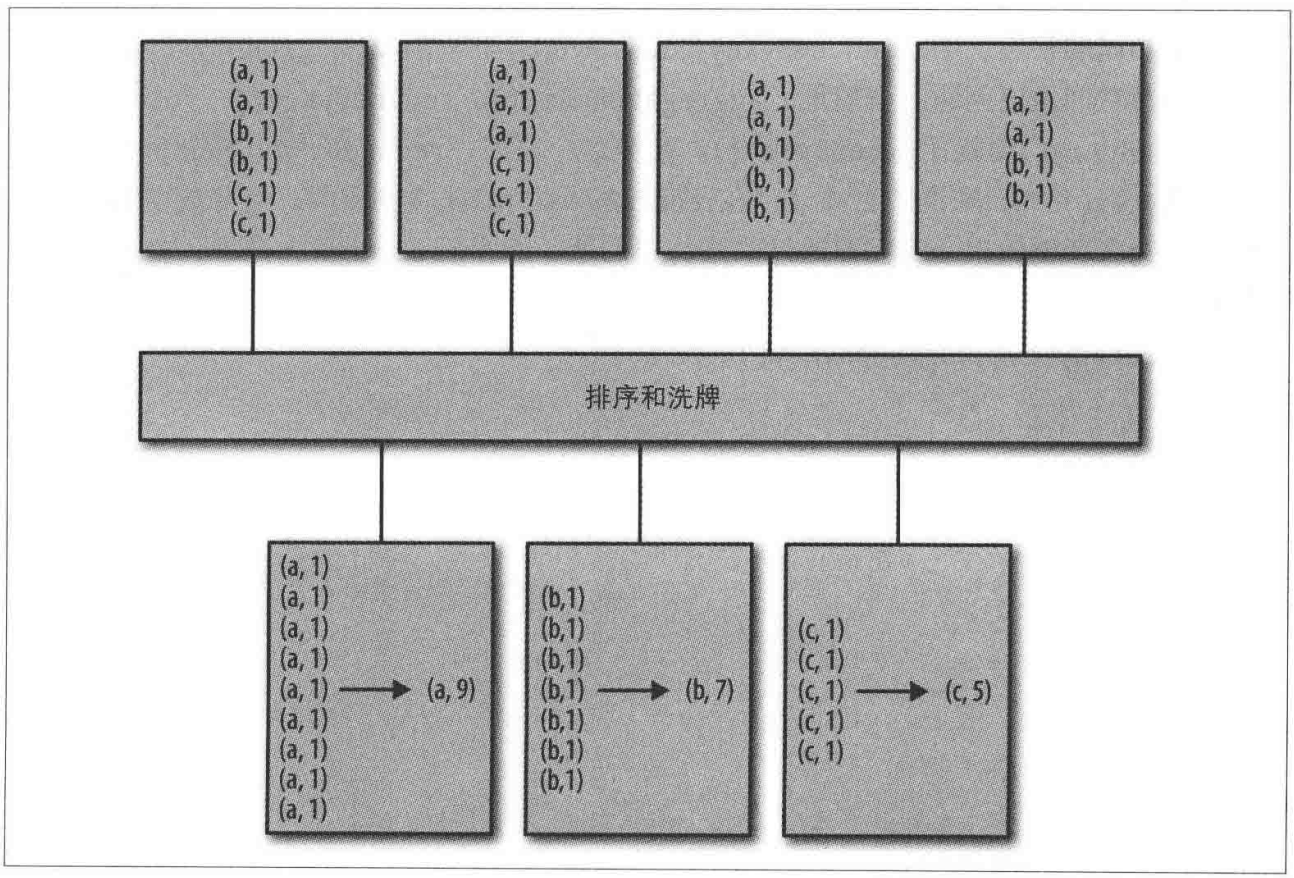
一般地，可以把groupByKey()和mapValues()合并到一个reduceByKey()或combineByKey()操作中（第4章和第8章给出了有关的例子）。

## reduceByKey()与groupByKey()

应当使用哪一个方法呢？是reduceByKey()还是groupByKey()？根据其语义，这两个方法可以给出相同的答案。不过reduceByKey()更为高效。有些情况下，groupByKey()甚至还会导致磁盘空间溢出问题。一般地，aggregateByKey()、reduceByKey()、foldByKey()和combineByKey()要优于groupByKey()。由于这个原因，reduceByKey()的Spark洗牌要比groupByKey()的Spark洗牌更高效：在reduceByKey()的洗牌步骤中，组合数据时，每个分区最多为各个键输出一个需要通过网络传送的值，而在groupByKey()中，会在网络上发送所有数据并由归约器收集，这很浪费。为了帮助了解这二者的差别，图B-2和图B-3展示了reduceByKey()和groupByKey()的洗牌过程。




图B-2：reduceByKey()的排序和洗牌步骤




图B-3：groupByKey()的排序和洗牌步骤

## 过滤RDD

Spark提供了一个很简单但很强大的API来实现RDD过滤。要过滤RDD元素，只需要实现一个filter()函数，对想保留的元素返回true，而对想要剔除的元素返回false。考虑示例B-12中名为logRDD的RDD。我们编写了一个filter()函数，它会保留包含normal关键字的元素，剔除其余的元素。

示例B-12：过滤JavaRDD

```
import java.util.List;
import com.google.common.collect.Lists;
import org.apache.spark.SparkConf;
import org.apache.spark.api.java.JavaSparkContext;
...
// 创建JavaSparkContext
SparkConf sparkConf = new SparkConf().setAppName("logs");
JavaSparkContext ctx = new JavaSparkContext(sparkConf);

// 创建所需的集合对象
final List<String> logRecords = Lists.newArrayList(
   "record 1: normal",
   "record 2: error",
   "record 3: normal",
   "record 4: error",
   "record 5: normal"
);

// 从Java集合对象创建一个RDD
JavaRDD<String> logRDD = ctx.parallelize(logRecords);

// 要过滤元素，需要实现一个filter函数：
// JavaRDD<T> filter(Function<T,Boolean> f)
// 返回一个新RDD，其中只包含满足指定谓词条件的元素。
JavaRDD<String> filteredRDD = logRDD.filter(new Function<String,Boolean>() {
    public Boolean call(String record) {
       if (record.contains("normal")) {
           return true; // 要返回这些记录
       }
       else {
           return false; // 不返回这些记录
       }
    }
});

// 在这里，filteredRDD将包含以下3个元素：
//   "record 1: normal"
//   "record 3: normal"
//   "record 5: normal"
```

## 将RDD保存为一个HDFS文本文件

可以把RDD保存为Hadoop文件（文本或序列文件）。例如，可以把上一节创建的counts RDD 作为一个文本文件保存在HDFS中：

```
// 将RDD保存为一个HDFS文本文件
// void saveAsTextFile(String path)
// 把这个RDD保存为一个文本文件，这里使用元素的字符串表示。
// 确保输出路径-- "/my/output" --不存在
counts.saveAsTextFile("/my/output");
```

保存之后，可以查看所保存的数据，如下所示:

```
$ hadoop fs -cat /my/output/part*
```

## 将RDD保存为一个HDFS序列文件

还可以把上一节创建的counts RDD作为一个序列文件保存在HDFS中（序列文件是键-值对形式的二进制文件）。要把一个RDD保存为序列文件，必须将其元素转换为Hadoop Writable对象。由于counts是一个类型为JavaPairRDD<String, Integer>的RDD，我们不能把它直接作为序列文件写入HDFS；首先需要创建一个新的RDD，类型为JavaPairRDD<Text, IntWritable>，在这里把String转换为Text，将Integer转换为IntWritable。示例B-13显示了如何将counts作为一个序列文件写入HDFS。

示例B-13：创建一个序列文件

```
import scala.Tuple2;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.mapred.SequenceFileOutputFormat;
...

// 首先创建一个Writable RDD:
JavaPairRDD<Text, IntWritable> countsWritable =
    counts.mapToPair(new PairFunction<
                                    Tuple2<String,Integer>, // T
                                    Text,                   // K
                                    IntWritable             // V
                                   >() {
    @Override
    public Tuple2<Text, IntWritable> call(Tuple2<String,Integer> t) {
        return new Tuple2<Text, IntWritable>(
                                            new Text(t._1),
                                            new IntWritable(t._2)
                                            );
    }
});

// 接下来，作为一个序列文件写入HDFS。
// 确保输出路径-- "/my/output2" --不存在。
```

```
25 countsWritable.saveAsHadoopFile(
26         "/my/output2",                  // 路径名
27         Text.class,                     // 键类
28         IntWritable.class,              // 值类
29         SequenceFileOutputFormat.class // 输出格式类
30 );
```

保存之后，可以查看所保存的数据，如下所示:

```
$ hadoop fs -text /my/output2/part*
```

## 从一个HDFS序列文件读取RDD

可以读取Hadoop序列文件创建一个新的RDD。在示例B-14中，读取我们在上一节创建的序列文件，并创建一个新的RDD。

示例B-14：读取一个序列文件

```
 1 import scala.Tuple2;
 2 import org.apache.hadoop.io.Text;
 3 import org.apache.hadoop.io.IntWritable;
 4 import org.apache.hadoop.mapred.SequenceFileInputFormat;
 5 ...
 6
 7 // JavaPairRDD<K,V> hadoopFile(String path,
 8 //                              Class<F> inputFormatClass,
 9 //                              Class<K> keyClass,
10 //                              Class<V> valueClass)
11 //对应一个有任意InputFormat的Hadoop文件，得到一个RDD
12 // '''注意:''' 由于Hadoop的RecordReader类会对各个记录重用
13 // 相同的Writable对象，所以直接缓存
14 // 返回的RDD会创建同一个对象的多个引用。
15 // 如果计划直接缓存Hadoop Writable对象，
16 // 应当首先使用一个映射函数复制这些对象。
17 JavaPairRDD<Text, IntWritable> seqRDD = ctx.hadoopFile(
18           "/my/output2",                        // HDFS路径
19           SequenceFileInputFormat.class,        // 输入格式类
20           Text.class,                           // 键类
21           IntWritable.class                     // 值类
22 );
```

## 统计RDD项

count()方法如示例B-15所示，会返回一个RDD中存储的项数。

示例B-15：统计RDD项

```
1 // 统计JavaPairRDD
2 JavaPairRDD<Text, IntWritable> pairRDD = ...;
3 int count1 = pairRDD.count();
4
5 // 统计JavaRDD
```

```
6 JavaRDD<String> strRDD = ...;
7 int count2 = strRDD.count();
```

## Scala的Spark RDD示例

如果对在Scala中使用RDD感兴趣，可以参考Zhen He的“The RDD API By Example”（*http://bit.ly/rdd_api_examples*）。

## PySpark示例

如果对在Python中使用RDD感兴趣，可以参考*https://github.com/mahmoudparsian/pyspark-tutorial*。

## 如何打包和运行Spark作业

这一节简要介绍在一个Spark集群或Hadoop的YARN环境中如何打包和运行Spark作业。提交Spark应用的详细信息参见Spark文档（*http://bit.ly/submitting_apps*）。

第一步是把应用类打包在一个JAR文件中（可以把它称为一个应用JAR）。可以使用一个构建工具（如Ant或Maven）生成你的应用JAR，或者从命令行使用jar脚本（*http://bit.ly/jar_files*）手动生成：

```
$ export JAVA_HOME=/usr/java/jdk7
$ JAVA_HOME/bin/jar cvf my_app.jar *.class
```

例如，运行本书中示例所需的JAR（名为*data_algorithms_book.jar*)就可以用一个Ant构建脚本生成（参见*http://bit.ly/da_book*的介绍）。一旦生成应用JAR，就可以运行你的程序了。

## 创建本书示例的JAR

这里会给出一个示例来说明如何创建JAR来运行这本书中的例子。这里假设你已经从GitHub（*http://bit.ly/da_book*）复制了这本书的源代码：

```
$ pwd
/home/mp/data-algorithms-book
$ ant clean
Buildfile: build.xml
clean:
BUILD SUCCESSFUL
Total time: 0 seconds

$ ant
Buildfile: build.xml
init:
```

```
    ...
copy_jar:
  [echo] copying spark-assembly-1.1.0-hadoop2.5.0.jar...
build_jar:
[echo] javac
[echo] compiling src...
[javac] Compiling 151 source files to /home/mp/data-algorithms-book/build
...
[jar] Building jar: /home/mp/data-algorithms-book/dist/data_algorithms_book.jar
BUILD SUCCESSFUL
Total time: 3 seconds
```

# 在Spark集群中运行作业

示例B-16展示了如何在Spark集群环境中运行一个Spark程序（Top10程序）。

示例B-16：在Spark集群中运行作业

```
1 $ cat run_top10_spark_cluster.sh
2 #!/bin/bash
3 export JAVA_HOME=/usr/java/jdk7
4 export SPARK_HOME=/usr/local/spark-1.1.0
5 export SPARK_MASTER=spark://myserver100:7077
6 export BOOK_HOME=/home/mp/data-algorithms-book
7 export APP_JAR=$BOOK_HOME/dist/data_algorithms_book.jar
8 INPUT=$BOOK_HOME/data/top10data.txt
9 # 在Spark独立集群上运行
10 $SPARK_HOME/bin/spark-submit \
11   --class org.dataalgorithms.chap03.spark.Top10 \
12   --master $SPARK_MASTER \
13   --executor-memory 2G \
14   --total-executor-cores 20 \
15   $APP_JAR \
16   $INPUT
```

表B-4中给出了这个脚本的描述

**表B-4：在Spark集群中运行作业的脚本的相关描述**

| 行号 | 描述 |
|---|---|
| 3 | JAVA_HOME是JDK7的主目录 |
| 4 | SPARK_HOME是Spark安装的主目录 |
| 5 | BOOK_HOME是GitHub中本书的安装目录 |
| 6 | SPARK_MASTER定义了Spark主URL |
| 7 | APP_JAR定义了定制应用JAR |
| 8 | INPUT定义Spark作业的输入路径 |
| 10 | 将Spark作业提交到Spark集群 |

# 在Hadoop的YARN环境中运行作业

示例B-17展示了如何在YARN环境中运行一个Spark程序（Top10程序）。

示例B-17：在Hadoop的YARN环境中运行作业

```
1 $ cat run_top10_yarn.sh
2 #!/bin/bash
3 export JAVA_HOME=/usr/java/jdk7
4 export SPARK_HOME=/usr/local/spark-1.1.0
5 export BOOK_HOME=/home/mp/data-algorithms-book
6 export SPARK_JAR=spark-assembly-1.1.0-hadoop2.5.0.jar
7 export THE_SPARK_JAR=$SPARK_HOME/assembly/target/scala-2.10/$SPARK_JAR
8 export APP_JAR=$BOOK_HOME/dist/data_algorithms_book.jar
9 #
10 export HADOOP_HOME=/usr/local/hadoop-2.5.0
11 export HADOOP_CONF_DIR=$HADOOP_HOME/etc/hadoop
12 INPUT=$BOOK_HOME/data/top10data.txt
13 $SPARK_HOME/bin/spark-submit \
14     --class org.dataalgorithms.chap03.spark.Top10 \
15     --master yarn-cluster \
16     --num-executors 12 \
17     --driver-memory 3g \
18     --executor-memory 7g \
19     --executor-cores 12 \
20     --conf "spark.yarn.jar=$THE_SPARK_JAR" \
21     $APP_JAR \
22     $INPUT
```

表B-5中给出了这个脚本的描述。

**表B-5：在YARN中运行作业的脚本的相关描述**

| 行号 | 描述 |
|---|---|
| 3 | JAVA_HOME是JDK7的主目录 |
| 4 | SPARK_HOME是Spark安装的主目录 |
| 5 | BOOK_HOME是GitHub中本书的安装目录 |
| 6 | 在YARN上运行程序必须有spark-assembly-1.1.0-hadoop2.5.0.jar |
| 7 | APP_JAR定义了定制应用JAR |
| 9~10 | 在YARN上运行程序必须有HADOOP_HOME和HADOOP_CONF_DIR |
| 12 | 将Spark作业提交到YARN环境 |

如果计划使用Spark读写HDFS，Spark的CLASSPATH上必须包括两个Hadoop配置文件：

- *hdfs-site.xml*, 提供HDFS客户端的默认行为。
- *core-site.xml*, 设置默认的文件系统名。

要让这些文件对Spark可见，在*$SPARK_HOME/sparkenv.sh*中将HADOOP_CONF_DIR设置为包含Hadoop配置文件的位置。

# 参考书目


[1] Alag, Satnam. *Collective Intelligence in Action*. Greenwich, CT: Manning Publications Co., 2009.

[2] Anderson, Carolyn J. "Exact Tests for 2-Way Tables." Urbana-Champaign: University of Illinois Department of Education Psychology, Spring 2014. *http://bit.ly/2-way_table_exact*.

[3] Batra, Siddharth, and Deepak Rao. "Entity Based Sentiment Analysis on Twitter." Stanford, CA: Stanford University Department of Computer Science, 2010. *http:// bit.ly/ batra_rao*.

[4] Boslaugh, Sarah. *Statistics in a Nutshell, Second Edition*. Sebastopol, CA: O'Reilly Media, Inc., 2013.

[5] Brandt, Andreas. "Algebraic Analysis of MapReduce Samples." Bachelor's thesis, Institute for Computer Science, 2010.

[6] Chen, Edwin. "Movie Recommendations and More via MapReduce and Scalding." Blog post. February 9, 2012. *http://bit.ly/movie_recs_mapreduce*.

[7] Conway, Drew, and John Myles White. *Machine Learning for Hackers*. Sebastopol, CA: O'Reilly Media, Inc., 2012.

[8] Dean, Jeffrey, and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters." Paper presented at the Sixth Symposium on Operating System Design and Implementation, San Francisco, CA, December 2004.

[9] Garson, G. David. *Cox Regression*. Asheboro, NC: Statistical Associates Publishers, 2012.

[10] Harrington, Peter. *Machine Learning in Action*. Greenwich, CT: Manning Publications Co., 2012.

[11] Jannach, Dietmar, Markus Zanker, Alexander Felfernig, and Gerhard Friedrich. *Recommender Systems: An Introduction*. Cambridge, UK: Cambridge University Press, 2011.

[12] Karau, Holden, Andy Konwinski, Patrick Wendell, and Matei Zaharia. Learning Spark. Sebastopol, CA: O'Reilly Media, 2015.

[13] Lammel, Ralf, and David Saile. "MapReduce with Deltas." Landau in der Pfalz, Germany: University of Koblenz-Landau, Software Languages Team, 2011. *http:// bit.ly/ lammel_saile*.

[14] Lin, Jimmy. "MapReduce Algorithm Design." Slideshow presented at WWW2013, Rio de Janeiro, Brazil, May 13, 2013. *http://bit.ly/mapreduce_tutorial*.

[15] Lin, Jimmy. "Monoidify! Monoids as a Design Principle for Efficient MapReduce Algorithms." College Park: University of Maryland, 2013. *http://bit.ly/monoidify*.

[16] Lin, Jimmy, and Chris Dyer. *Data-Intensive Text Processing with MapReduce*. College Park: University of Maryland, 2010.

[17] Lin, Jimmy, and Chris Dyer. *Data-Intensive Text Processing with MapReduce*. San Rafael, CA: Morgan & Claypool Publishers, 2013.

[18] Miner, Donald, and Adam Shook. *MapReduce Design Patterns*. Sebastopol, CA: O'Reilly Media, Inc., 2013.

[19] Mitchell, Tom M. *Machine Learning*. New York: McGraw-Hill, 1997.

[20] Netflix. "Netflix Prize." 2009. *http://www.netflixprize.com/*.

[21] Pang, Bo, and Lillian Lee. "A Sentimental Education: Sentiment Analysis Using Subjectivity Summarization Based on Minimum Cuts." Ithaca, NY: Cornell University, 2004. *http://bit.ly/pang_and_lee*.

[22] Perry, John. Foundations of Nonlinear Algebra. Hattiesburg: University of Southern Mississippi, 2012. *http://bit.ly/nonlinear_algebra*.

[23] Sarkar, Manish, and Tze-Yun Leong. "Application of K-Nearest Neighbors Algorithm on Breast Cancer Diagnosis Problem." Medical Computing Laboratory, Department of Computer Science, School of Computing, The National University of Singapore, 2000.

[24] Schank, Thomas, and Dorothea Wagner. "Finding, Counting and Listing All Triangles in Large Graphs: An Experimental Study." Karlsruhe, Germany: University of Karlsruhe, 2005. *http://bit.ly/schank_and_wagner.*

[25] Schank, Thomas, and Dorothea Wagner. "Approximating Clustering Coefficient and Transitivity." *Journal of Graph Algorithms and Applications* 9, no. 2 (2005): 265－275.

[26] Segaran, Toby. *Programming Collective Intelligence: Building Smart Web 2.0 Applications*. Sebastopol, CA: O'Reilly Media, Inc., 2007.

[27] Sembiring, Sajadin, M. Zarlis, Dedy Hartama, S. Ramliana, and Elvi Wani. "Prediction of Student Academic Performance by an Application of Data Mining Techniques." Paper presented at the 2011 International Conference on Management and Artificial Intelligence, Bali, Indonesia, April 2011.

[28] Tsourakakis, Charalampos E., Petros Drineas, Eirinaios Michelakis, Ioannis Koutis, and Christos Faloutsos. "Spectral Counting of Triangles in Power-Law Networks via Element-Wise Sparsification." Pittsburgh, PA: Carnegie Mellon School of Computer Science, 2009. *http://bit.ly/tsourakakis_et_al.*

[29] Walker, Michael G. "Survival Analysis." *http://bit.ly/surv_analysis.*

[30] Wellek, Stefan, and Andreas Ziegler. "Cochran-Armitage Test Versus Logistic Regression in the Analysis of Genetic Association Studies." *Human Heredity* 73, no. 1 (2012): 14－17.

[31] White, Tom. *Hadoop*: *The Definitive Guide, Fourth Edition*. Sebastopol, CA: O'Reilly Media, Inc., 2015.

[32] Wikipedia. "Correlation and Dependence." *http://bit.ly/correlation_article.*

[33] Woo, Jongwook, and Yuhang Xu. "Market Basket Analysis Algorithm with Map/Reduce of Cloud Computing." Los Angeles: Computer Information Systems Department, California State University, 2013. *http://bit.ly/mba_w_mapreduce.*

[34] Yang, Changyu. "Triangle Counting in Large Networks." Master's thesis, University of Minnesota, Duluth, 2012.

[35] Zaharia, Matei. "An Architecture for Fast and General Data Processing on Large Clusters." Berkeley: University of California, 2014. *http://bit.ly/zaharia.*

[36] Zhao, Xi, Einar Andreas Rodland, Therese Sorlie, Bjorn Naume, Anita Langerod, Arnoldo Frigessi, Vessela N Kristensen, Anne-Lise Borresen-Dale, and Ole Christian

Lingjarde. "Combining Gene Signatures Improves Prediction of Breast Cancer Survival." Oslo, Norway: Department of Genetics, Institute for Cancer Research, Oslo University Hospital, 2011. *http://bit.ly/zhao_et_al.*

Document generated by Anna’s Archive around 2023-2024 as part of the DuXiu collection (https://annas-blog.org/duxiu-exclusive.html).

Images have been losslessly embedded. Information about the original file can be found in PDF attachments. Some stats (more in the PDF attachments):

{
  "filename": "MTQxMDk3MDcuemlw",
  "filename_decoded": "14109707.zip",
  "filesize": 104817214,
  "md5": "b7577c149da41d2c4858135b9f0dbfeb",
  "header_md5": "d8a3bae032f1ce6a83d76a39af253e26",
  "sha1": "ab9911e923bffc6b73338b3cb86d33c6420794c3",
  "sha256": "08e86e0b372264e431b088031bb4bb13b190949e79c38cfaf1886cc615bba5cc",
  "crc32": 505521620,
  "zip_password": "",
  "uncompressed_size": 104650632,
  "pdg_dir_name": "14109707",
  "pdg_main_pages_found": 680,
  "pdg_main_pages_max": 680,
  "total_pages": 694,
  "total_pixels": 3543031008,
  "pdf_generation_missing_pages": false
}